per viure millor

Dedicatòria

a la Joana i al Xavier,

als pares i als germans,

a la Montse,

i als companys de dinar.

Agraïments

A l'Angel Messeguer per les correccions, suggeriments, comentaris, observacions, etc., que m'ha fet,

al Joaquim Gabarró per la confiança que m'ha demostrat i l'ajut constant que m'ha ofert,

al Jorge Castro perquè no hi ha res millor per treballar bé que tenir un bon company de despatx,

a Conrado Martínez i Ricardo Baeza per tots els suggeriments i correccions,

i a Felipe Cucker, Rafael Casas, José Luis Balcázar, Josep Diaz i Horacio Rodríguez pels seus consells i ajuts.

A les secretàries i a la gent de Sistemes del Departament per la seva amable predisposició a solventar qualsevol problema.

I finalment, al propi Departament de Llenguatges i Sistemes Informàtics i al Centre Europeu de Paral·lelisme de Barcelona pels medis posats al meu abast [1].

[1] Gran part de la feina s'ha fet dins els Projectes ALCOM I i II del Programa de la Comunitat Europea ESPRIT BRA.

English / Català

This PhD. thesis has been written in English to facilitate its diffusion. Nevertheless, as this pragmatic point of view pushes my own language into the background, I have added a Catalan version of the most generic chapters, *i.e.* those which non-specialists will find most easily understable, such as the introduction and conclusion.

Aquesta tesi ha estat escrita en anglès per no posar obstacles a la seva difusió. No obstant, com aquest punt de vista pragmàtic arracona, en certa manera, la meva llengua, he afegit la versió catalana dels capítols més genèrics, és a dir, d'aquells capítols més a l'abast dels no especialistes en la matèria, com ho són l'introductori i el de les conclusions.

It is important to take care of our own language, and if this responsability is neglected, it will be very difficult to maintain this language alive.

Cal tenir cura de la llengua, i si no ho fa qui la té per seva, difícilment es mantindrà viva.

Contents

Chapter 1

Introduction / Introducció

Computer science deals with the automatic management of **data sets**. The operations to be performed on a data set define an **abstract data type**, and some of these data sets have become classical research material. A goog example is the **dictionary**, which is defined by the following operations:

1. **testing** membership in the set,
2. **insertion** of elements into the set,
3. **deletion** of elements from the set.

The operations are represented by **algorithms**, which describe the sequence of elementary steps needed to perform these operations.

Dictionaries can be represented by many **data types**, such as **lists**, **hash tables** and **search trees**, and the selection of one of them depends on the time and space requirements of the dictionary. We restrict ourselves to those cases in which the data can be sorted and the **worst case** time is suitably bounded. This means search trees, namely **balanced search trees**. As the algorithms take time to work in proportion to the height of trees, the balanced search trees ensure a worst case logarithmic time with respect to the number of dictionary elements.

La informàtica tracta de la manipulació automàtica de **conjunts de dades**. Les operacions que els manipulen defineixen un **tipus abstracte de dades**. Alguns d'aquests conjunts han esdevingut clàssics camps de recerca, com el **diccionari**, definit per les següents operacions:

1. **consulta** d'un element del conjunt,
2. **inserció** d'elements dins del conjunt,
3. **supressió** d'elements del conjunt.

Les operacions estan representades pels **algorismes**, els quals descriuen la seqüència d'instruccions elementals necessàries per realitzar les operacions.

Els diccionaris poden ser representats per molts **tipus de dades**, tals com **llistes**, **taules de dispersió** i **arbres de cerca**, i l'elecció d'un d'ells ve determinada per les exigències en temps i espai de cada diccionari. Nosaltres ens limitarem a aquells diccionaris en què el conjunt de dades està ordenat i el temps d'accés en el **cas pitjor** està convenientment afitat. Això significa arbres de cerca, concretament **arbres de cerca equilibrats** ja que, com els algorismes triguen un temps proporcional a l'altura de l'arbre, asseguren en aquest cas un temps logarítmic respecte del nombre d'elements del diccionari.

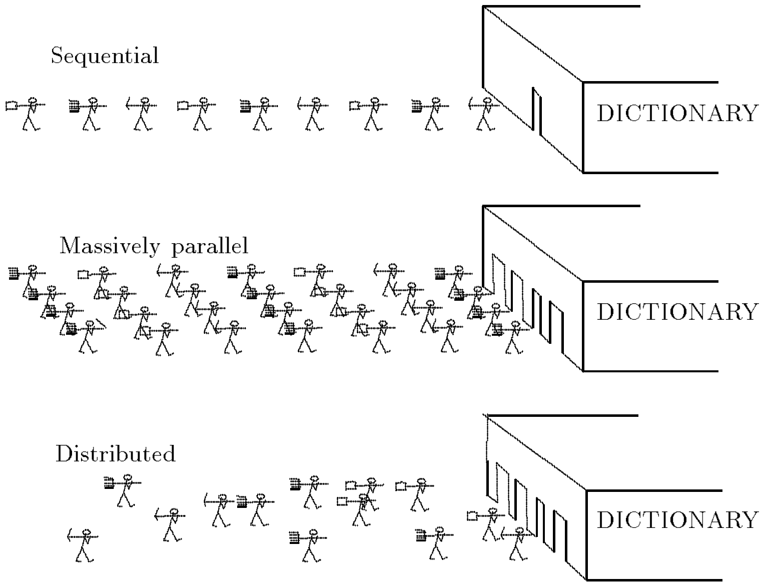


Figure 1.1: A picture of algorithms

When the dictionary is consulted by only one user it is not possible nowadays to design significant algorithmic improvements. Conversely, if there are many users, then we can find more efficient strategies than the **sequential** strategy, which assigns the dictionary to one user after another as if it were in book format. In fact, the simultaneous (or **concurrent**) use by many users is desirable. For this purpose **parallel** or **concurrent** algorithms must be designed.

Quan el diccionari és consultat per només un sol usuari no és possible, per avui, dissenyar millores algorísmiques remarcables. Per contra, si hi ha molts usuaris, podem buscar estratègies de consulta millors que la **seqüencial**, la qual assigna el diccionari a un usuari rera l'altre, com si fos en format de llibre. S'ha de permetre que molts usuaris el consultin simultàniament (o **concurrentment**). Llavors cal dissenyar algorismes **paral·lels** o **concurrents**.

We assume that users create a **process** for each **query** (test) or **update** (insertion or deletion) that they carry out. Each process is controlled by a sequential algorithm because it is devoted to only one user, and all processes are managed by **parallel algorithms**. The parallel algorithm processes can run either synchronously or not. In the first case we call the algorithm **massively parallel**, and in the second case we call it **distributed** (see Figure 1.1).

Diem que l'usuari crea un **procés** per a cada **consulta** o **actualització** (insercions i supressions) que faci. Cada procés està controlat per un **algorisme seqüencial** degut a que atany a un sol usuari, i tots els processos són controlats per un **algorisme paral·lel**. Els processos poden ser executats sincrònicament o asincrònicament. En el primer cas parlem d'algorismes **massivament paral·lels**, i en el segon d'algorismes **distribuïts** (vegis figura 1.1).

Parallel computers have been available for many years. They range from the asynchronous **distributed computers** with few but very powerful processors, to the synchronous **massively parallel** or **data parallel computers** with thousands of simple processors. The gap between these two types of machines is currently filled by **parallel virtual machines**, which are made of computer networks.

Des de fa anys existeixen ordinadors paral·lels, i van des dels **ordinadors distribuïts** asíncrons amb pocs processadors però molt potents, fins els **ordinadors massivament paral·lels** amb milers de processadors simples. Entre aquests dos extrems existeixen, actualment, els **ordinadors paral·lels virtuals**, formats per xarxes d'ordinadors.

1.1 This thesis / Aquesta tesi

This thesis deals with **massively parallel** and **distributed** algorithms on **balanced search trees**.

Distributed algorithms: These consist of those parallel algorithms composed by asynchronous processes which manage data types in a concurrent approach. This means that data can be simultaneously queried and updated. Problems arise when an updating process interferes with another process and induces erroneous responses. We separate distributed algorithms into two groups:

Global scope : Algorithms composed of processes which emulate sequential algorithms. This means that the processes leave the data type balanced.

Consequently, there are only two kinds of processes: **queries** (or readers) which only find elements and **updaters** (or writers) which insert or delete elements.

Aquesta tesi tracta d'algorismes **distribuïts i massivament paral·lels** sobre **arbres de cerca equilibrats**.

Algorismes distribuïts: Són aquells algorismes paral·lels que manipulen tipus de dades de forma asíncrona. Això significa que es poden consultar i actualitzar les dades simultàniament. Els problemes sorgeigen quan una consulta interfereix amb una actualització, la qual cosa pot comportar respostes equivocades. Dividim els algorismes distribuïts en dos grups:

Àmbit global: Algorismes composats per processos que emulen algorismes seqüencials. Això significa que els processos deixen l'estructura equilibrada.

Llavors només hi ha dos tipus de processos: **consultors** (o lectors) que tan sols busquen elements i **actualitzadors** (o escriptors) que insereixen o suprimeixen elements.

Partial scope: Algorithms whose update processes do not balance the structure. This means that updaters only add/remove the data to/from the structure and that these distributed algorithms must include some kind of rules to balance the structure. The rules are said to be **local** if they are composed of a small number of sentences that access a small and fixed number of neighbour nodes. For instance, a distributed algorithm on AVL trees may include rotations as a local rule.

We are interested in partial scope algorithms with local rules because they accept a higher concurrence degree and, with an adequate scheduler, can emulate global scope algorithms. Our development follows the research lines started by J.L.W. Kessels on AVL trees in 1983 [25].

Massively parallel algorithms: These consist of those parallel algorithms composed by processes that manage data types in a synchronized manner. They can be envisaged as many sequential algorithms running simultaneously and executing the same sentence at the same time.

Therefore, it may happen that more than one process reads or writes on the same memory location at the same time (**concurrent access**). These conflicts are more easily addressed if the design of the algorithms is based on sets of local actions.

Àmbit parcial: Algorismes composats per actualitzadors que no equilibren l'estructura. Això significa que els actualitzadors només afegeixen/eliminen elements de l'estructura i que cal afegir noves regles que l'equilibrin. Si les regles estan controlades per algorismes amb poques instruccions que accedeixen a un nombre fix i petit de nodes veins, llavors parlem de **regles locals**. Per exemple, un algorisme distribuït pels arbres AVL pot incloure com a regla local les rotacions.

Nosaltres estem interessats en algorismes d'àmbit parcial amb regles locals perque accepten un major grau de concurrència i poden emular, amb un planificador adequat, algorismes d'àmbit global. La nostra recerca continua la iniciada el 1983 per en J.L.W. Kessels sobre els arbres AVL [25].

Algorismes massivament paral·lels: Són aquells algorismes paral·lels que manipulen dades sincronitzadament. Poden ser entesos com molts algorismes seqüencials que executen simultàniament la mateixa instrucció.

Per tant, pot passar que més d'un procés llegeixi o escrigui en la mateixa posició de memòria en el mateix moment (**accés concurrent**). Aquests conflictes es poden encarar millor si l'algorisme es basa en regles locals.

This class of algorithms was under active theorical research in the early 80's. Later on, this interest decreased until the appearance of data parallel machines.

Note that the time needed to search k elements with k processors on balanced search trees is $O(\log n + \log k)$ which is very close to the optimal speedup of $O(\log n)$ (sequential time for one search or update). The first algorithm on balanced search trees was designed by W. Paul, U. Vishkin and H. Wagener on 2–3 trees in 1983 [43].

Aquest tipus d'algorismes va ser un actiu camp de recerca teòrica al principi dels anys 80. Posteriorment l'interès minvà fins que aparesqueren els ordinadors paral·lels.

Cal observar que el temps de cerca de k elements amb k processadors en arbres balancejats de cerca és $O(\log n + \log k)$, molt proper al temps òptim seqüencial $O(\log n)$ (temps de consulta o actualització d'un element). El primer algorisme dissenyat fou el de W. Paul, U. Vishkin and H. Wagener sobre arbres 2–3 l'any 1983 [43].

Balanced search trees: These consists of those balanced trees whose elements, denoted **keys**, are ordered in a depth-first, left-right traversal. The balance criteria define different tree types, from which we consider the following to be the most representative: **2–3 trees**, **AVL trees** and **Skip lists**.

Arbres de cerca equilibrats: Són aquells arbres equilibrats tal que els seus elements, anomenats **claus**, estan ordenats en inordre. El criteri d'equilibri defineix diferents tipus d'arbres, dels quals considerem com a més representatius els següents: **arbres 2–3**, **arbres AVL** i **"Skip lists"**.

2–3 trees: They were developed by J.E. Hopcroft in 1970 and are considered the precursors of B-trees, which were introduced by R. Bayer and C. McCreight in 1972 [6].

Arbres 2–3: Foren dissenyats per J.E. Hopcroft al 1970 i són considerats els precursors dels arbres B introduïts per R. Bayer and C. McCreight al 1972 [6].

AVL trees: They were the first balanced trees ever introduced and were designed by G.M. Adel'son-Vel'skiĭ and Y.M. Landis in 1962 [1].

Arbres AVL: Foren els primers arbres balancejats desenvolupats, i foren dissenyats per G.M. Adel'son-Vel'skiĭ and Y.M. Landis.

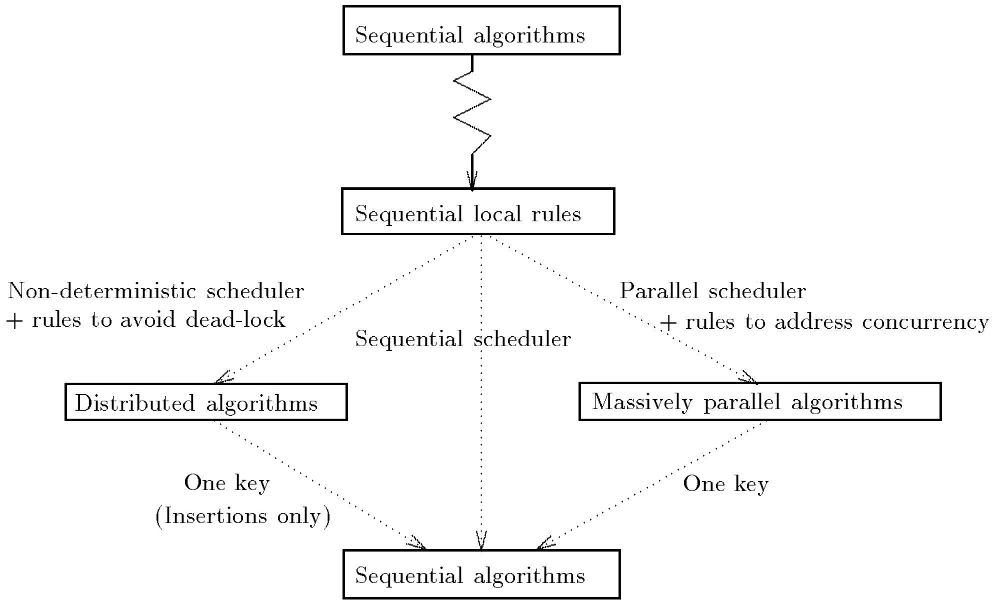


Figure 1.2: Common design framework

Skip lists: They were introduced by W. Pugh in 1990 [47]. As they are composed of a set of linked lists, they are not actually search trees. However, Skip lists behave very much like trees as for as searching, insertion and deletion are concerned. Balancing this data type is achieved through randomization.

"Skip lists": Foren introduïdes per W. Pugh al 1990 [47]. Com les **"Skip lists"** estan composades per llistes enllaçades no són realment arbres de cerca. Però si ens fixem en el comportament dels algorismes, llavors podem veure les "Skip lists" com a arbres de cerca. L'estructura s'equilibra de forma aleatòria.

We would like to consider massively parallel and distributed algorithms in a very general and common setting. In fact, we were looking for massively parallel algorithms for a long time, but we were only successful in obtaining them after a study of the properties of distributed algorithms. Invariants and local rules from distributed algorithms gave us enough freedom to go on and a good basis to address massively parallel algorithms. This fact suggested that a good design methodology could consists of the following:

Volíem tractar els algorismes distribuïts i massivament paral·lels dins d'un marc comú. De fet ens hem trobat que aconseguíem dissenyar algorismes massivament paral·lels després d'analitzar les propietats dels algorismes distribuïts. Els invariants i les regles locals dels algorismes distribuïts ens donaven la suficient llibertat alhora que una bona base per dissenyar els algorismes massivament paral·lels. Aquest fet suggeria que un bon mètode de disseny podria ser:

First, try to solve the problem for a distributed environment with a set of local rules and adequate global invariants and then, try to achieve the massive parallel algorithm by coupling the rules of the distributed algorithm with a tightly synchronized scheduler.

Primer resoldre el problema en un entorn distribuït amb un conjunt de regles locals i invariants globals, i posteriorment aconseguir algorismes massivament paral·lels acbplant les regles locals amb un bon planificador.

Note that local rules play a fundamental role in our development because both types of algorithms are based on them. Then, we propose a design method (see Figure 1.2) that can be summarized as follows:

Observem que les regles locals juguen un paper fonamental ja que ambdós tipus d'algorisme paral·lels s'hi basen. Llavors proposem un mètode de disseny (vegis figura 1.2) que passem a concretar:

Common design framework

1. Extraction of the local rules from sequential algorithms.
2. Design of the distributed algorithm based on this set of local rules and on others designed to avoid dead-locks, and also on a set of very weak invariants.
3. Design of the specific parallel scheduler for the data parallel algorithm based on this set of local rules and on some rules dealing with concurrent access.

Marc comú de disseny

1. Extracció de les regles locals dels algorismes seqüencials.
2. Disseny de l'algorisme distribuït a partir d'aquestes regles locals i algunes més per evitar la mort-per-inanició, i d'un conjunt d'invariants molt febles.
3. Disseny del planificador de l'algorisme massivament paral·lel a partir de les regles locals i d'algunes més que tractin l'accés concurrent.

Note that this method does not investigate how the data structures can be efficiently scanned in distributed algorithms. It is only assumed that these algorithms are controled by a **fair** scheduler, which ensures that all requests for rules will be served.

Observem que aquest mètode no estudia com recorrer eficientment les estructures de dades en els algorismes distribuïts. No més suposa que aquests algorismes són controlats per un planificador **equitable**, que ens assegura que totes les peticions de regles seran satisfetes.

The initial objectives were determined by the state of the art (see Table 1.1) in 1990: filling the empty entries of the table, namely $\dagger^{1,2,5}$. These goals have been achieved as the following list shows:

Els propòsits inicials foren determinats per l'estat de la recerca (vegis taula 1.1) al 1990: el disseny dels algorismes corresponents a les cel·les buides $\dagger^{1,2,5}$. Aquests objectius han estat aconseguits tal i com ho mostra la següent llista:

	Skip lists	2–3 trees and B-trees	AVL trees
Sequential	W. Pugh 1990 [47]	J.E. Hopcroft 1970 R. Bayer and C. McCreight 1972 [6]	G.M. Adel'son-Vel'skiĭ and Y.M. Landis 1962 [1]
Concurrent with local rules	†[2]	O. Nurmi, E. Soisalon-Soininen and D. Wood 1987 [36] †[3]	J.L.W. Kessels 1983 [25] O. Nurmi, E. Soisalon-Soininen and D. Wood 1987 [36, 37] †[4]
Massively parallel	†[1]	W. Paul, U. Vishkin and H. Wagener 1983 [43] L. Higham and E. Schenks 1994 [22]	†[5]

Table 1.1: State of the art with our contributions (†)

Our algorithmic contribution

†[1]: Massively parallel algorithms for Skip lists.

†[2]: Design of a new type of search trees, one-to-one mapped to Skip lists, whose algorithms commute with this mapping. Design of the distributed algorithm.

†[3]: Distributed algorithm for 2–3 trees.

†[4]: Fine-grained distributed algorithm for AVL trees.

†[5]: Massively parallel algorithms for AVL trees.

La nostre aportació algorísmica

†[1]: Algorisme massivament paral·lel per les “Skip lists”.

†[2]: Disseny d'un nou tipus d'arbres de cerca isomorf a les “Skip lists” respecte dels algorismes. Disseny de l'algorisme distribuït.

†[3]: Algorisme distribuït per arbres 2–3.

†[4]: Algorisme distribuït de granularitat fina pels arbres AVL.

†[5]: Algorisme massivament paral·lel pels arbres AVL.

1.2 Overview / Contingut

The contents of the chapters are outlined here. First, we begin with the introductory chapters to explain the basic tools upon which we have based our research. Thereafter, we introduce the chapters which contain our main algorithmic contributions. Finally, we include a chapter that comprises the analysis of the evolution of tree leaves once the insertion of keys in a parallel manner has been accomplished.

Comentarem el contingut dels capítols. En primer lloc tenim els capítols introductoris que expliquen les eines bàsiques a partir de les quals hem realitzat el nostre treball de recerca. Després introduïm els capítols que contenen la nostra aportació algorísmica més important. Finalment incloem un capítol que conté l'anàlisi de l'evolució de les fulles dels arbres en inserir claus en paral·lel.

1.2.1 Introductory chapters / Capítols introductoris

Since sequential local rules play a fundamental role in this work, the second chapter addresses their extraction from sequential algorithms.

Chapter 2. Local rules of sequential algorithms: We explain how the local rules are extracted and, in particular, how to extract them from the data type selected: 2–3 trees, AVL trees and Skip lists.

This set of rules allows us to rewrite the most relevant previously reported work. Next, an explanation of our methods and proposals is presented.

Chapter 3. Establishment of objectives: We explain the design of J.L.W. Kessels for distributed insertion algorithm on AVL trees and the design of W. Paul, U. Vishkin and H. Wagener for massively parallel algorithms on 2–3 trees. Then, a widely explanation of the design method and a detailed explanation of our proposals are presented. Finally, we sketch some historical notes.

Com les regles locals juguen un paper fonamental en aquest treball, dediquem el segon capítol a extreure-les dels algorismes seqüencials.

Capítol 2. Regles locals dels algorismes seqüencials: Expliquem com s'extreuen les regles locals i posteriorment ho fem per als tipus de dades escollides: arbres 2–3, arbres AVL i "Skip lists".

Aquest conjunt de regles ens permet reescriure les aportacions prèvies més importants. Seguidament expliquem el nostre mètode i els nostres propòsits

Capítol 3. Establiment dels objectius: Expliquem el disseny de J.L.W. Kessels de l'algorisme distribuït d'inserció sobre els arbres AVL i el disseny de W. Paul, U. Vishkin and H. Wagener dels algorisme massivament paral·lels sobre arbres 2–3. Després fem una àmplia explicació del mètode de disseny i dels nostres propòsits. Finalment afegim algunes notes històriques.

1.2.2 Algorithmic chapters / Capítols algorísmics

These chapters contains our algorithmic contribution.

The first chapter is devoted to a top-down design and to the implementation of massively parallel algorithms for Skip lists.

Aquests capítols contenen la nostra aportació algorísmica.

El primer capítol algorísmic conté el disseny descendent i la implementació de l'algorisme massivament paral·lel sobre les "Skip lists".

Chapter 4. Skip lists: massively parallel algorithms: We design the massively parallel search, insertion and deletion algorithms for Skip lists. For each algorithm we analyse the concurrent access conflicts and their efficiency. Finally, we sketch the main lines of an implementation in `C*` and we show some of the experimental results obtained.

Capítol 4. "Skip lists": algorismes massivament paral·lels: Dissenyem els algorismes massivament paral·lels de cerca, inserció i supressió per les "Skip lists". Per a cada algorisme analitzem els conflictes d'accés concurrent i la seva eficiència. Finalment fem una breu introducció a la implementació en `C*` i mostrem alguns resultat experimentals.

The non-locality of Skip lists rules makes it impossible to design the distributed algorithm by using local rules. This fact was overcome by creating new search trees, denoted Skip trees, which can be one-to -one mapped to Skip lists. Moreover, this mapping commutes with the algorithms of both data types.

La no localitat de les regles de les Skip lists fan impossible el disseny de l'algorisme distribuït amb regles locals. Aquest malaurat fet s'ha resolt amb la creació d'un nou tipus d'arbre de cerca, anomenat "Skip trees". Aquests arbres són isomorfs a les "Skip lists", la qual cosa vol dir que l'aplicació bijectiva commuta amb els algorismes.

Chapter 5. Skip trees (and 2–3 trees): We define this new class of trees and we prove that a one-to-one mapping exists between Skip lists and Skip trees, which commutes with search and update algorithms, and that both structures have equal expected performance. Finally, we design the corresponding distributed algorithm.

Since Skip trees can be stored in secondary storage, we analyse their relationship with B-trees. The similarities allow us to include in this chapter a distributed algorithm on 2–3 trees.

Capítol 5. "Skip trees" (i arbres 2–3): Definim aquesta nova classe d'arbres. Demostrem que existeix una aplicació bijectiva entre les "Skip lists" i els arbres "Skip trees", que commuta amb els algorismes de cerca i actualització, i que ambdos estructures tenen idèntiques magnituds esperades. Finalment dissenyem l'algorisme distribuït corresponent.

Com els arbres "Skip trees" poden ser enregistrats en memòria secundària, analitzem la seva relació amb els arbres B. Aquesta proximitat ens convida a incloure en aquest capítol l'algorisme distribuït sobre els arbres 2–3.

The last two algorithmic chapters, namely 6 and 7, are devoted to AVL trees.

Els dos darrers capítols algorísmics 6 i 7 estan dedicats als arbres AVL.

	Skip lists	2–3 trees and Skip trees	AVL trees
Concurrent with local rules		X. Messeguer [34] 1996 (Chap. 5)	L. Bougé, J. Gabarró and X. Messeguer [9, 10] 1995 (Chap. 6)
Massively parallel	J. Gabarró, C. Martinez and X. Messeguer [19] 1995 (Chap.4)		J. Gabarró and X. Messeguer [20] 1996 (Chap. 7)

Table 1.2: Our algorithmic contributions

The first of these chapters includes the design of a more fine-grained algorithm than those previously published, and solves affirmatively the question proposed by H.T. Kung and P.L. Lehman: is it possible to uncouple rotations which balance trees from information propagation on a distributed algorithm?

Chapter 6. AVL trees: a fine-grained distributed algorithm: First, we address a balancing distributed algorithm which transforms any kind of binary search tree with only local information into an AVL tree. Then, we address the concurrent search, insertion and deletion algorithm. Finally, we rewrite previously published algorithms by using our rules.

It is worth remembering that AVL trees modify their structure in a different way from 2–3 trees: the algorithms modify the height (or the depth) of some nodes by adding or removing leaves and by rotating nodes. This fact leads to an important problem that we have to face: the synchronization of processes in the massively parallel algorithms.

Chapter 7. AVL trees: massively parallel algorithms: First, we introduce the virtual magnitudes that make the synchronization processes possible. Then, we design the insertion and the deletion algorithms.

El primer conté un algorisme distribuït de granularitat més fina que els existents, i respon afirmativament la pregunta proposada per H.T. Kung i P.L. Lehman: és possible separar les rotacions que equilibren els arbres de la propagació d'informació en un algorisme distribuït?

Capítol 6. Arbres AVL: algorisme distribuït de granularitat fina: Primerament dissenyem un algorisme distribuït que transforma qualsevol tipus d'arbre de cerca binari amb només informació local, en un arbre AVL. Després encarem l'algorisme de cerca, inserció i supressió. Finalment reescrivim els algorismes apareguts anteriorment amb les nostres regles.

Recordem que els arbres AVL modifiquen la seva estructura de forma diferent als arbres 2–3: els algorismes modifiquen l'altura (o la profunditat) dels nodes en afegir o suprimir fulles i en fer rotacions. Aquest fet planteja un problema greu: la sincronització dels processos en els algorismes massivament paral·lels.

Capítol 7. Arbres AVL: algorismes massivament paral·lels: Primerament introduïm les magnituds virtuals que ens permetran sincronitzar els processos. Després dissenyem els algorismes d'inserció i supressió.

Table 1.2 summarizes our algorithmic contributions.

La taula 1.2 mostra la nostra contribució algorísmica

1.2.3 Analytical chapter / Capítol analític

Finally, the last chapter includes the **fringe analysis** of parallel insertions on 2–3 trees. This analysis studies the distribution evolution of tree leaves.

Chapter 8. Fringe analysis on parallel 2–3 trees: First, we study the evolution under parallel insertion of two and three keys. These two cases allow us to conjecture and prove the evolution on the k keys parallel insertion case.

Finalment l'últim capítol conté l'**anàlisi del serrell** de les insercions en paral·lel sobre els arbres 2–3. Aquest tipus d'anàlisi estudia l'evolució de la distribució de les fulles dels arbres

Capítol 8. Anàlisi del serrell en arbres 2–3: Primerament estudiem l'evolució sota l'inserció de dues i tres claus. Aquest dos casos ens permeten conjecturar i demostrar l'evolució en el cas general d'inserció de k claus en paral·lel.

Chapter 2

Local rules of sequential algorithms

Local rules play a fundamental role in our design method of parallel algorithms. After an introductory section, we address how local rules can be extracted from sequential algorithms. Finally, the last three sections are devoted to the extraction of local rules from the selected search trees, namely from 2–3 trees, AVL trees and Skip lists.

2.1 Introduction

Each balanced search tree has a particular set of rules, but some of them are so similar that their development does not bring significant improvements. Therefore we selected the following search trees because they have different set of rules and are representative of many others as well. We deal with:

2–3 trees: They are characterized by local rules which split and join nodes, and are representatives of B-trees, 2–3–4 trees, etc.

AVL trees: They are characterized by local rules that propagate information and perform rotations, and are representatives of Red-black trees, various kind of weight balanced search trees, etc.

Skip lists : They are characterized by the use of probabilistic rules.

We note that Skip lists have a double nature: although being a set of linked lists, they are organized as search trees. The first view keeps out the local properties of rules while the second can support them. Therefore, although our development speaks about *local* rules and search *trees* to unify the management of all three structures, the reader should remember this double nature.

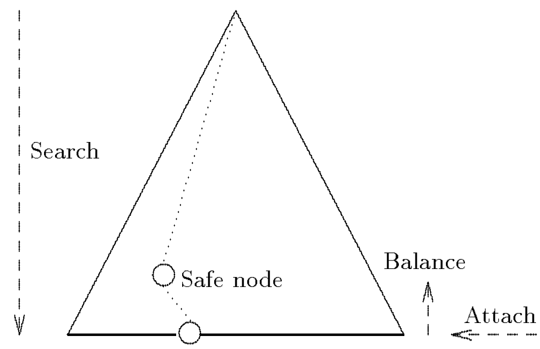
We establish the meaning of some basic notions about algorithms or parts of them.

The **search** algorithm tests membership and consists only of a **search** phase.

The **insertion** algorithm adds a new key into the set and consists of three phases: a **search** phase that carries the key down through the structure, an **attach** phase that adds the key to the structure and a **balance** phase that arranges the perturbation created by the new key.

The **deletion** algorithm removes the key from the set and consists of a **search** phase that finds the key and a **remove** phase that eliminates the key and arranges the structure. Sometimes this *remove* phase is composed by an **unattach** phase that strictly removes the key followed by a **balance** phase that rearranges the tree.

Procedure insertion (T: 2–3 tree; x : TKey)
  $\{T$ is not empty$\}$
  $n :=$ root(T)
    $\{n$ is the current node $\}$
  while $\neg$leaf(son(n)) **do**
      – percolate x to adequate son –
  end
  – if necessary attach new leaf to n –
  while $|$key(n)$| > 2$ **and** $n \neq$ **root**(T) **do**
    – split n –
  end
  – if necessary create new root –



Algorithm 2.1: Insertion algorithm on 2–3 trees

Both the insertion and deletion algorithms are collectively called **update** algorithms.

2.2 How to extract the local rules

The local rules extracted from sequential algorithms will be called **sequential local rules** and are based on **local actions** performed by sequential algorithms. Recall that a local action is an algorithm composed by a small number of sentences which access a small and fixed number of "neighbourhood" nodes.

Although the iterative version of some classical algorithms contains non local updates, such as pointers to far nodes (for instance, the deletion algorithm on 2–3 trees), there exists iterative (or recursive) versions with only local structure changes. The sentences of these algorithms can be grouped into local actions used by the main algorithm, and from these local actions we design the sequential local rules. Then, the extraction of local rules is composed by the following two steps:

Extraction of local actions from sequential algorithms: Local actions are extracted from sequential algorithms in a very natural manner. As the meaning of *local action* is that it is composed by a fixed number of sentences and access a fixed number of neighbourhood nodes, loops cannot be incorporated into them. Therefore we essay to isolate such loops. For instance, take the iterative version of insertion algorithm on 2–3 tree (Algorithm 2.1). The search phase is composed by a loop and a set of sentences, such as assignments and tests, which can be grouped into a local action referred as **Percolation**; the bottom-up phase is composed by the attach phase which determines the local action **Attach**; and the balance phase is composed by a loop and a set of sentences which we group into the local action **Split**. These actions access three neighbourhood nodes at most and have a fixed number of sentences.

Introduction of local registers from global variables: Local actions are transformed into local rules by taking away the global variables of the sequential algorithm (for instance, the key x to be inserted in the insertion algorithm 2.1). Then, global variables must be stored into nodes as new local registers, and these registers will be examined by processes to accomplish their tasks.

We introduce now the most significant registers added to nodes.

Search algorithm: We need a new register, denoted **waiting** bag, to store the key to be searched. The algorithm starts by adding the key in the **waiting** bag of the root. The **Percolation** rule tests whether the node's waiting is empty or not, and, if so, sends the key down from that waiting bag to the waiting bag of the appropiate descendants.

Insertion algorithm: We need a new register, denoted **red**, to mark the unbalanced nodes. Then, if a new key is attached to the tree and the new node does not holds the balance conditions, we colour it **red**. For instance, on 2–3 trees, red nodes are those nodes with three keys, and the **Split** rule must examine the colour of the nodes.

Deletion algorithm: The balance phase of deletion algorithm needs two registers. First, the **green** register which is active on nodes to be removed. For instance, on 2–3 trees a node is green if it does not contain keys.

Second, the **update** register that contains pairs of keys (x, y) whose meaning is that the first key x should be remplaced by second key y. Recall that sequential deletion algorithms substitute the key to be removed, say it x, by its precedent key, say it y, and that this substitution is made by means of a pointer. As local rules cannot support far pointers, this substitution should be performed by a bottom-up process, in which the substitute key is brought up through the tree.

We address now the script of rules. Each one contains the items **Guard, Behaviour, Variant function** and **Spatial scope**. The rules syntax and the meaning of items are the following:

Rule : Name

Guard: *Specifies the conditions under which the rule can be applied.*

Behaviour: *Contains the sequence of actions performed by the rule.*

Variant function: *Shows a function that strictly decreases after each execution of the rule.*

Spatial scope: *Shows the set of elements (nodes) accessed by the rule.*

Note that there can exists many variant functions that strictly decrease for each rule.
As different search trees have similar rules, it is not necessary to write for each one the whole script with the four items. In these cases, we will repeat the name of the rule

Rule : Name

and the items that have been modified.

2.3 Rules of 2–3 trees

A 2–3 tree (see Figure 2.1) is a search tree with the following properties:

$P_1 \equiv$ every node has one key and two sons, or two keys and three sons.

$P_2 \equiv$ all leaves appear at the same depth.

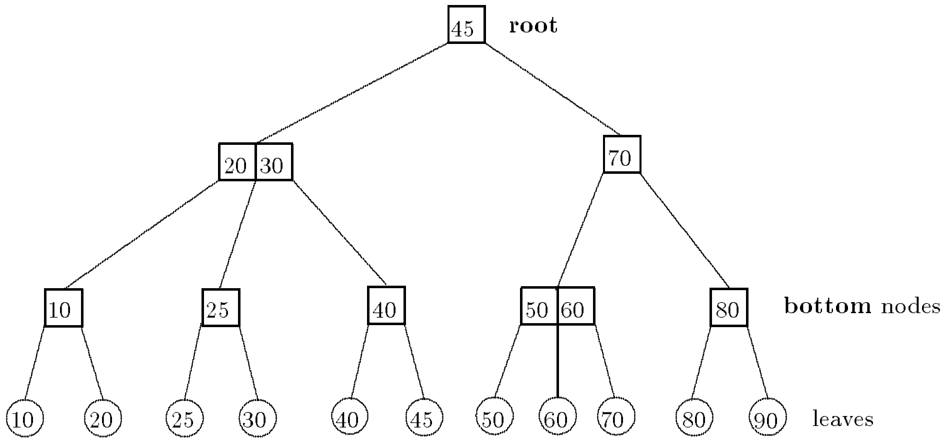


Figure 2.1: 2–3 tree

They are highly balanced search trees. Every node n stores one or two keys in **lkey**(n) and **rkey**(n), two or three pointers to its sons in **left**(n), **middle**(n) and **right**(n), and a pointer **parent**(n) to the father. We assume that any leaf has two *nil* nodes as left and right sons. As usual, the *height* of a node n can be defined bottom-up:

$$\begin{aligned}&\mathbf{height}(nil) = 0\\&\mathbf{height}(n) = 1 + \max(\mathbf{height}(\mathbf{left}(n)), \mathbf{height}(\mathbf{middle}(n)), \mathbf{height}(\mathbf{right}(n))),\end{aligned}$$

therefore all the leaves have height 1. The *depth* can be defined top-down starting with:

$$\begin{aligned}&\mathbf{depth}(\mathbf{root}) = 0\\&\mathbf{depth}(n) = 1 + \mathbf{depth}(\mathbf{parent}(n)).\end{aligned}$$

We have the following invariant for any node n belonging to a 2–3 tree:

$$\mathbf{depth}(n) + \mathbf{height}(n) = \mathbf{height}(\mathbf{root}) = \mathbf{depth}(nil).$$

A **safe** node (see Algorithm 2.1) is the last singleton node (node that contains only one key) in the path from the root to a leaf, and the **bottom** nodes are the deepest nodes of tree, *i.e.* those nodes whose sons are leaves.

Search

Algorithm: Assume that we search for key x and n is the current node which stores two keys a and b such that $a < b$. Key x is tested against a and b: if $x \le a$ then the search continues in the left son of n, if $a < x \le b$ in the middle son, otherwise in the right son because $x > b$. When n is a singleton, the algorithm acts in a similar manner.

Extraction of local rules: Note that the previous algorithm compares repeatedly the key x against the keys stored in the nodes, and since this action is composed by a small number of sentences and access a small number of nodes, then it can be viewed as a local action. The local rule is obtained by adding to each node the new register **waiting** bag which will contain a key waiting to be percolated. We define the **Percolation** rule:

Rule : Percolation

Guard: *Node n such that* **waiting**$(n) = \{x\}$.

Behaviour: *Key x is extracted from* **waiting**(n) *bag. If n is a bottom node then x is found iff x is a leaf of n, otherwise compare x against the keys of n and give it into the waiting bag of the adequate son.*

Variant function: **height**(n), *where n is the unique node in the tree such that* **waiting**$(n) = \{x\}$.

Spatial scope: *Node n and one of its sons.*

Insertion

Algorithm: First, we search for the bottom node at which the new leaf is attached, and we store the new key on the bottom node. If this node stores two keys, the process is over; but if it stores three keys, we start the balance phase in which this node is **split** and the middle key is sent up to father's node, and so on (see Algorithm 2.1) until the **safe** node is reached.

Extraction of local rules: Recall that we must design the local rules by isolating the local actions and storing the global variables information into local registers of nodes. Also, we must colour **red** those nodes which unbalance the tree.

We define two variant function:

$$\begin{aligned} \text{RED} &= \text{number of red nodes} \\ \text{NWAITING} &= \text{number of keys contained into waiting bags.} \end{aligned}$$

The first rule finds the adequate bottom node:

Rule : Percolation

Guard: *Node n such that* **waiting**$(n) = \{x\}$ *and* $\neg$**bottom**(n).

Behaviour: *Assume that key x, extracted from* **waiting**(n), *should be inserted. If it is found, i.e. is equal to some key of n, then remove it from waiting bag; otherwise, propagate it to the adequate son's waiting bag.*

Variant function: **height**(n).

Spatial scope: *Node n and one of its sons.*

The second rule attaches the new leaf and if necessary colours node red:

Rule : Red attach

Guard: *Node n such that* **waiting**$(n) = \{x\}$ *and* **bottom**(n).

Behaviour: *Key x is extracted from* **waiting**(n). *If it is equal to some leaf of n then remove it; otherwise attach a new leaf with key x from node n and insert x into n. If n stores three keys then colour it red.*

Variant function: NWAITING.

Spatial scope: *Node n.*

The third rule achieves the balance phase and takes place when there are red nodes, *i.e.* node with three keys. The rule must split red nodes:

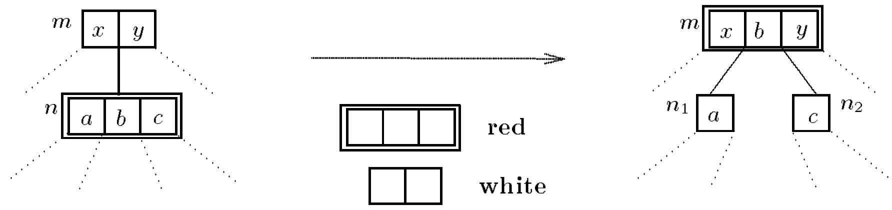


Figure 2.2: **Red Split** rule

Rule : Red Split

Guard: *Node n such that* $\mathbf{red}(n)$ *and* $\neg\mathbf{root}(n)$.

Behaviour: *Let a, b, c be the keys of n such that $a < b < c$ (See Figure 2.2). The middle key b is stored at the father's node m of n and n is split into two nodes n_1 and n_2. Node n_1 is attached at the left of key b and n_2 at the right. If m contains three keys then we colour it red.*

Variant function: *The sorted pair* (RED,$\mathbf{depth}(n)$).

Spatial scope: *Node n and its parent.*

Note: *Sequential environment implies that* $\neg\mathbf{red}(m)$, *but this rule can be also applied with* $\mathbf{red}(m)$.

The last rule creates a new root if the root is red:

Rule : Red Root

Guard: *Node n such that* $\mathbf{red}(n)$ *and* $\mathbf{root}(n)$.

Behaviour: *A new node is created with the middle key of n. Node n is split into n_1, n_2 which are then attached to the new root.*

Variant function: RED.

Spatial scope: *Root n.*

Deletion

We cannot use the classical deletion algorithm [2]. Assume that key x must be deleted, then this algorithm first searchs for the bottom node n which has attached a leave with key x. But note that key x is also stored into another node m of the search path. Therefore, when the current node is n, a pointer to m is needed to replace key x by its predecessor key, but m could not be a n neighbour, so this replacement is made with a far pointer which cannot be suported by local rules.

We explain an algorithm that uses only local actions.

Algorithm: First we search for the bottom node n that has attached leaf x, then it is removed. If n stores key x then also remove key x from n, and if n has become empty then balance the tree. Otherwise x is stored into an ancester node m of n, *i.e.* a node in the path from the root to n, and the predecessor key y of x is stored into the same bottom node n. Then we kept the predecessor key y and start a bottom-up process to bring key y up to ancester node m. When m is reached, we substitute key y for key x. Finally, if node n which stored y has become empty, then we must balance the tree. Observe that both substitution and balancing can be made with the same bottom-up process.

Extraction of local rules: In this case we color **green** the nodes to be removed until the tree is balanced. We define the variant function

$$\text{GREEN} = \text{ number of } \mathbf{green} \text{ nodes}$$

The first rule

Rule : Percolation (as in searches and insertions),

reaches the bottom node that has attached the searched leaf. Next, the following rule removes leaf x and starts the bottom-up phase.

Rule : Request manager

Guard: *Node n such that* $\mathbf{waiting}(n) = \{x\}$ *and* $\mathbf{bottom}(n)$.

Behaviour: *Remove leaf x. If node n stores key x then remove it, otherwise remove y (predecessor of x) from its node n and store the pair (x, y) into* **update***(n). Finally if n has no keys color it green.*

Variant function: NWAITING.

Spatial scope: *Nodes n.*

This request rule can create two perturbations which have to be sent up: one is created by the local register **update**, and the other by the local register **green**. Recall that the sequential deletion algorithm solves them with an unique bottom-up phase. Now, we design three rules to deal with them. The first rule deals with the **update** register:

Rule : Update

Guard: *Node n such that* $\mathbf{update}(n) = \{(x, y)\}$.

Behaviour: *If node n stores key x then replace x by y. Otherwise store the pair into* $\mathbf{update}(\mathbf{parent}(n))$

Variant function: $\mathbf{depth}(n)$.

Spatial scope: *Node n and its parent.*

The following rule propagate or remove green nodes.

Rule : Green Propagation

Guard: *Node n such that* $\mathbf{green}(n), \neg\mathbf{root}(n)$ *and* $\mathbf{update}(n) = \emptyset$.

Behaviour: *If one of their brothers has two keys, keep one of them and stores it into n. If their brothers have only one key then keep one key of the parent, add it to a brother and attach* $\mathbf{son}(n)$*. If* $\mathbf{parent}(n)$ *stores no keys, i.e. it becomes empty, then color parent green.*

Variant function: $(\text{GREEN}, \mathbf{depth}(n))$.

Spatial scope: *Node n, brothers and parent.*

And the last rule deals with green roots:

Rule : Green Root

Guard: *Node n such that* $\mathbf{green}(n)$ *and* $\mathbf{root}(n)$.

Behaviour: *Remove n.*

Variant function: GREEN.

Spatial scope: *Node n.*

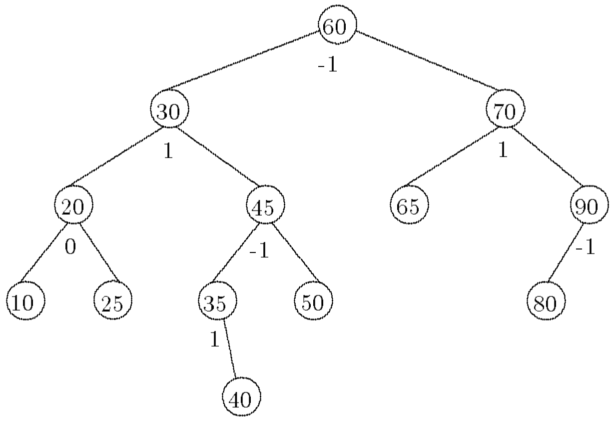


Figure 2.3: AVL tree

2.4 Rules of AVL trees

AVL trees are highly balanced search trees (see Figure 2.3). Every node n stores a key, denoted $\mathbf{key}(n)$, $\mathbf{left}(n)$ and $\mathbf{right}(n)$ pointers to their sons and a pointer $\mathbf{parent}(n)$ to the father. To compute the height we define

$$\begin{aligned}&\mathbf{height}(nil) = 0\\&\mathbf{height}(n) = 1 + \max(\mathbf{height}(\mathbf{left}(n)), \mathbf{height}(\mathbf{right}(n))).\end{aligned}$$

The depth of nodes is defined as

$$\begin{aligned}&\mathbf{depth}(\mathbf{root}) = 0\\&\mathbf{depth}(n) = 1 + \mathbf{depth}(\mathbf{parent}(n)),\end{aligned}$$

and the **balance field** of a node n is given by

$$\mathbf{bal}(n) = \mathbf{height}(\mathbf{right}(n)) - \mathbf{height}(\mathbf{left}(n)).$$

AVL trees are highly balanced because for any node n of the tree, $\mathbf{bal}(n) \in \{-1, 0, 1\}$.

We adopt the following notation: $n(A, B)$ denotes the (sub)tree with root n, left son A and right son B (sons may possibly be empty). Also, we present only the leftwise versions of the rules. The rightwise version can be obtained by symmetry. As a convention, the final state of a node n is denoted n'. Unless specified, it is identical to the initial state. We name the rotation sub-rules after the position of the subtrees with dominant heights.

We define the following numbers because they determine most of the variant functions. Let n be any node in the tree, then

$$\begin{aligned}\mathbf{inside}(n) &= \text{size of the subtree rooted at } n(\text{ including } n).\\\mathbf{outside}(n) &= \text{number of nodes out of the subtree rooted at } n.\end{aligned}$$

Note that for any node n, $\mathbf{inside}(n) + \mathbf{outside}(n)$ is the size of the tree.

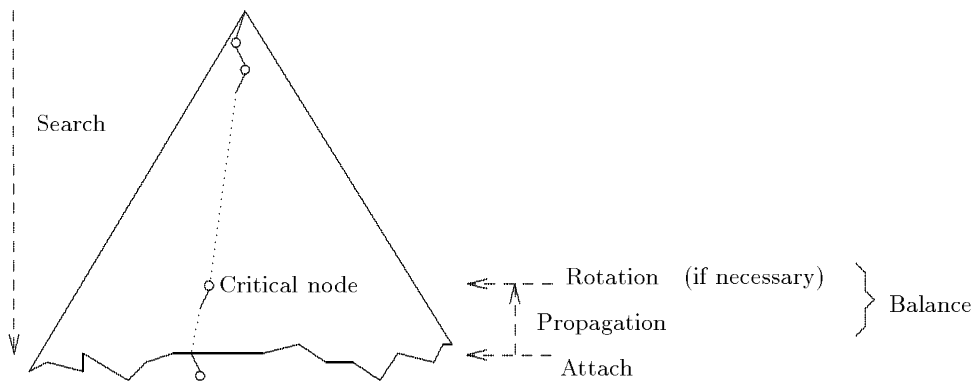


Figure 2.4: Insertion algorithm on AVL trees

Search

Algorithm: We search for x by testing it against the key of the current node n. If n is the *nil* node then x does not belong to the tree; otherwise we keep three cases: if $x <$ **key**(n) then **left**(n) becomes the current node; if $x >$ **key**(n) then **right**(n) becomes the current node; otherwise the key has been found.

Extraction of local rules: We add the **waiting** register to all nodes and keep a **Percolation** rule to search keys. Note that all search trees has similar search rules.

Rule : Percolation

Guard: *Node n such that* **waiting**$(n) = \{x\}$.

Behaviour: *Key x is extracted from* **waiting**(n) *bag. If n is a nil node then remove n, otherwise if $x \neq$* **key**(n) *send x to adequate son's bag.*

Variant function: **inside**(n).

Spatial scope: *Node n and one son.*

Insertion

Algorithm: We make a top-down search for the leaf with key x. Along this search we cross over the **critical node** m which is the last node such that **bal**$(m) \neq 0$ (See Figure 2.4). After the leaf is found, we attach a new leaf and we rearrange bottom-up the balance of those nodes located before the critical node. Note that these nodes, which lie between the new leaf and the critical node, have balance field zero before the insertion, and -1 or 1 after it. Then this phase can be envisaged as the propagation of the tree growth which has been created by the new leaf.

Finally, when the propagation reaches the critical node, the balance of this node lies into $\{-2, 0, 2\}$. Then, if it has become unbalanced ($|$**bal**$(m)| = 2$), then the critical node is arranged with a rotation.

Extraction of local rules: We extract the rules that search and balance the tree. The search is made with

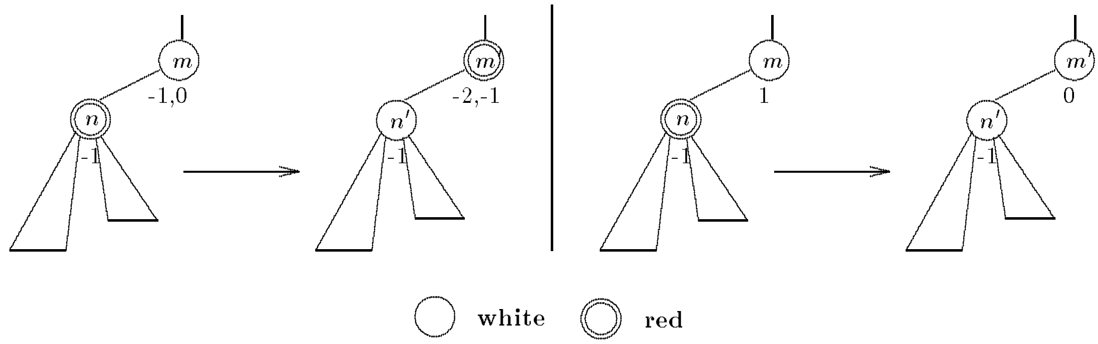


Figure 2.5: **Red Propagation** rule (left case)

Rule : Percolation (as in search algorithm).

The attachment of the new leaves is made with rule

Rule : Red attach

Guard: *Node n such that* **waiting**$(n) = \{x\}$ *and* **leaf**(n).

Behaviour: *Extract key x from* **waiting**(n). *If it is equal to* **key**(n) *then remove it, otherwise attach a new red leaf with key x to leaf n.*

Variant function: NWAITING.

Spatial scope: *Node n.*

The analysis of the balance phase, which is composed by a bottom-up propagation followed by rotation, suggest the following design of rules. First the propagation rules:

Rule : Red Propagation

Guard: *Node n such that* **red**(n), $\neg$**root**(n) *and* $|$**bal**$(n)| < 2$. *(Figure 2.5)*

Behaviour: *Update n, m into n', m' with*

$$\mathbf{red}(m') \equiv \mathbf{bal}(m) \in \{-1, 0\},\ \neg\mathbf{red}(n') \text{ and } \mathbf{bal}(m') = \mathbf{bal}(m) - 1$$

Variant function: **outside**(n).

Spatial scope: *Node n and its father m.*

Note: *Symmetrically if n is the right son of m.*

Rule : Whitening Red Root

Guard: *Node n such that* **red**(n), **root**(n) *and* $|$**bal**$(n)| < 2$.

Behaviour: *Colours root white.*

Variant function: RED.

Spatial scope: *Root n.*

If rule propagation colours red the critical node and the balance becomes 2 or -2, it is necessary to rotate around the critical node. Note that the son which participates in rotations has the balance field different from zero, then the following rotation rules take into account only this fact by taking **bal**$(p) \neq 0$.

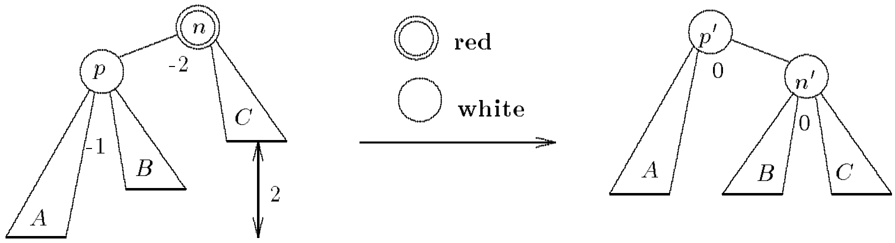


Figure 2.6: **Red Rotation** rule (single left-left case)

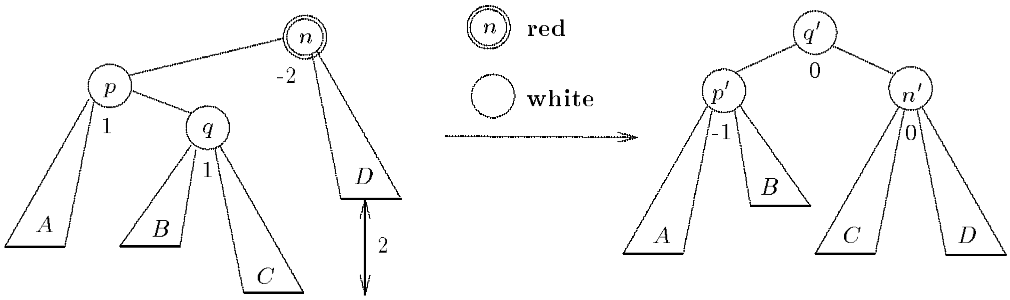


Figure 2.7: **Red Rotation** rule (double left-right case)

Rule : Red Rotation (single left-left case)

Guard: *A subtree* $n(p(A, B), C)$ *such that* $\mathbf{red}(n), \mathbf{bal}(n) = -2$, *and* $\mathbf{bal}(p) = -1$*(Figure 2.6)*

Behaviour: *Restructure the tree into* $p'(A, n'(B, C))$ *with the updating of the balance* $\mathbf{bal}(n') = \mathbf{bal}(p') = 0$.

Variant function: RED

Spatial scope: *Node* n *and its left son* p.

Note: *Symmetrically for the right-right case:* $n(A, q(B, C)), \mathbf{bal}(n) = 2, \mathbf{red}(n)$, *and* $\mathbf{bal}(q) = 1$.

Rule : Red Rotation (double left-right case)

Guard: *A subtree* $n(p(A, q(B, C)), D)$, $\mathbf{red}(n), \mathbf{bal}(n) = -2$, $\mathbf{bal}(p) = 1$ *(Figure 2.7).*

Behaviour: *Restructure the tree into* $q'(p'(A, B), n'(C, D))$ *with the updating of the balance:*

		$\mathbf{bal}(q) = 1$	$\mathbf{bal}(q) = -1$
$\mathbf{bal}(n')$	$=$	0	1
$\mathbf{bal}(p')$	$=$	-1	0
$\mathbf{bal}(q')$	$=$	0	0

Variant function: RED.

Spatial scope: *Node n, its son p and p's son q.*

Note: *Symmetrically for the right-left case: $n(A, q(p(B, C), D))$, $\mathbf{bal}(n) = 2, \mathbf{red}(n)$ and $\mathbf{bal}(q) = -1$.*

Recall that rotations appear only when the involved son has balance field zero. This fact implies that in both rotation cases, the height of trees decreases and the root become white. But, if the rotation involves sons with the same height (for instance, if $\mathbf{bal}(p) = 0$ in Figure 2.6), as occurs in distributed algorithm, the tree has the same height after rotation. That means that the height does not decreases with some rotations, and that in this cases the colour red remains active.

Deletion

As in 2–3 trees we cannot deal with the classical deletion algorithm because it contains far pointers. We propose the following deletion algorithm:

Algorithm: First we search for key x. If x is a leaf then we remove it and start a bottom-up balance process. Otherwise we search for the predecessor key y of x and start a bottom-up balance process which sends key y up, propagates the decreasement up and eventually rotates nodes.

Extraction of local rules: The first rule

Rule : Percolation (as in search phase),

sends key x down the tree until a leaf with key x (or key predecessor of x) is found. Then the rule **Request manager** starts the bottom-up balance phase. Like in 2–3 trees, we need the register **update** which contains the pair (x, y) with the meaning that key x should be replaced by key y. We define the numbers

$$\text{UPDATE} \equiv \text{number of pairs } (x, y)$$
$$\text{BAL} \equiv \text{the addition of balance fields of all nodes}$$

Rule : Request manager

Guard: *Node n such that* $\mathbf{waiting}(n) = \{x\}$ *and* $\mathbf{leaf}(n)$.

Behaviour: *If leaf n stores key x then remove leaf; otherwise store the pair* $(x, \mathbf{key}(n))$ *into* $\mathbf{update}(\mathbf{parent}(n))$. *Afterthat, colour* $\mathbf{parent}(n)$ *green iff n was the unique son,i.e.* $\mathbf{parent}(n)$ *becomes a leaf.*

Variant function: NWAITING.

Spatial scope: *Node n.*

As in 2–3 trees, two kind of perturbation can be created, so we must design two kind of rules to face them. The first kind of rules send the substitute key up:

Rule : Update

Guard: *Node n such that* $\mathbf{update}(n) = (x, y)$.

Behaviour: *If $x \in$* $\mathbf{key}(n)$ *then replace x by y. Otherwise stores the pair into* $\mathbf{update}(\mathbf{parent}(n))$

Variant function: $\mathbf{outside}(n)$, UPDATE.

Spatial scope: *Node n and its parent.*

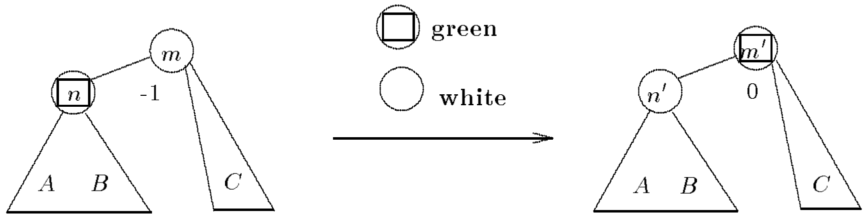


Figure 2.8: **Green Propagation** rule(left case with $\mathbf{bal}(m) = -1$)

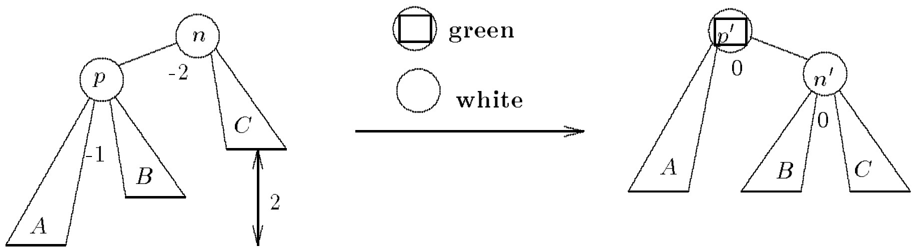


Figure 2.9: **Green Rotation** rule(single left-left case with $\mathbf{bal}(p) = -1$)

A green node means that the tree rooted at it has decreased its height, then we design a rule that propagates this decreasement up.

Rule : Green Propagation

Guard: *A subtree* $m(n(A, B), C)$ *such that* $\mathbf{green}(n)$

Behaviour: $\mathbf{bal}(m') := \mathbf{bal}(m) + 1$ *and* n' *becomes white. If* $\mathbf{bal}(m) = -1$, *then* m' *becomes green (Figure 2.8).*

Variant function: $\mathbf{outside}(n)$.

Spatial scope: *Node* n *and its parent* m.

Note: *Symmetrically if* n *is the right son of* m.

When *Green Propagation* reaches the root we apply

Rule : Whitening Green Root (as **Whitening Red Root**).

Finally we need the rotation rule because **Green Propagation** rule unbalances nodes:

Rule : Green Rotation

Guard: *A subtree rooted at* n *such that* $\mathbf{bal}(n) = -2$ *and nodes* n, p, q *are white.*

Behaviour: *If* $\mathbf{bal}(p) \in \{-1, 0\}$ *(single rotation case) restructure the tree* $n(p(A, B), C)$ *into* $p'(A, n'(B, C))$ *with the usual updating for balance fields and let* $\mathbf{green}(p')$ *iff* $\mathbf{bal}(p) = -1$.

If $\mathbf{bal}(p) = 1$, *(double rotation case) restructure the tree* $n(p(A, q(B, C)), D)$ *into* $q'(p'(A, B), n'(C, D))$, *the balance fields and let* $\mathbf{green}(p')$.

Variant function: BAL.
Spatial scope: *Node n and its left son p.*
Note: *The symmetrical case.*

Note that the propagation of green nodes can unbalance them, and the rotation of nodes can colours them green.

2.5 Rules of Skip lists

Skip lists (see Figure 2.10) are randomized data structures introduced by W. Pugh in 1990 [47]. Sequential skip list algorithms are very simple to implement, and they provide significant constant factor improvements over balanced and self-adjusting trees. On the other hand, skip lists are also space efficient, requiring an average of 2 (or less) pointers per item and no balance, priority or weight information. Furthermore, the probability of the search time or space complexity exceeding their expected values rapidly approaches 0 as the items number in the skip list increases [51]. Skip lists have been widely analysed: the expected search cost [40, 41] by T. Papadakis, J.I. Munro and P.V. Poblete, the expected length of maximal search paths [12] by L. Devroye, and the expectation and variance of total search cost (total path length) [27] by P. Kirschenhofer and H. Prodinger.

A non-empty Skip list consists of several non-empty sorted linked lists. All the items are stored in the list of *level* 1. Some of them also belong to the list of level 2, and so forth.

Each item x in S has a key denoted as $\mathbf{key}(x)$ and a positive integer $\mathbf{level}(x)$. If $\mathbf{level}(x) = \ell$, it means that x belongs to the linked lists of level $1, 2, \ldots, \ell$.

To implement a skip list, we need to allocate a node for each item. Each node x contains the item and $\mathbf{level}(x)$ pointers. The successor of x at level l, denoted $\mathbf{forward}(x, \ell)$, is given by the ℓ-th *forward* pointer of x. A header node, $\mathbf{header}(S)$, points to the first node of each linked list. We write $\mathbf{level}(S)$ to denote the maximum level among the levels of its items. The level of S is also called as its *height*. We will use the following conventions:

- There is a node called **NIL** pointed by the last node of each of the linked lists. By convention, $\mathbf{level}(\mathbf{NIL}) = \mathbf{level}(S)$, and stores a key greater than any legal key: $\mathbf{key}(\mathbf{NIL}) = +\infty$;
- The level of $\mathbf{header}(S)$ is $\mathbf{level}(\mathbf{header}(S)) = \mathbf{level}(S)$ and it stores a dummy key smaller than any legal key: $\mathbf{key}(\mathbf{header}(S)) = -\infty$;
- Each node also stores as many copies of its key as its level indicates. This convention simplifies the analysis of read conflicts (see section 4.2).

Let $x \neq \mathbf{NIL}$ be an item of S and ℓ be some integer such that $0 \leq \ell \leq \mathbf{level}(x)$; we define the $\mathbf{wall}(x, \ell)$ of x at level ℓ as the first node y to the right of x,*i.e.* $\mathbf{key}(y) > key(x)$, such that $\mathbf{level}(y) > \ell$ For instance, in Figure 2.10, $\mathbf{wall}(\mathbf{header}(S), 3)$ is the 5th item in S and its key is 50. Note that $\mathbf{wall}(x, 0) = \mathbf{forward}(x, 1)$. Let $I(x, \ell)$ be the interval:

$$I(x,\ell) = (\mathbf{key}(x), \mathbf{key}(\mathbf{wall}(x,\ell))] = \{k \mid \mathbf{key}(x) < k \leq \mathbf{key}(\mathbf{wall}(x,\ell))\}.$$

If some key a belongs to $I(x, \ell)$, it must be the key of a node or fall between the keys of two consecutive nodes, for some of the nodes between x and $\mathbf{wall}(x, \ell)$.

Lemma 2.1 *Let S be a skip list and let a be a key such that $a \in I(x, \ell)$, for $x \neq \mathbf{NIL}$ and $1 \leq \ell \leq \mathbf{level}(x)$. Let b be the key stored at the successor of x at level ℓ, that is, $b = \mathbf{key}(\mathbf{forward}(x, \ell))$. Then,*

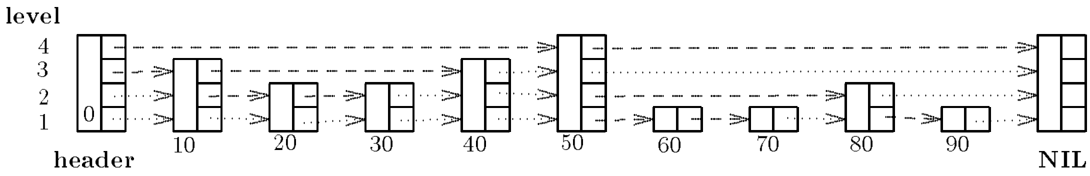


Figure 2.10: Skip list

- *If* $a > b$ *then* $a \in I(\mathbf{forward}(x, \ell), \ell)$ *and* $\mathbf{level}(\mathbf{forward}(x, \ell)) = \ell$.
- *If* $a \leq b$ *then* $a \in I(x, \ell - 1)$.

Moreover, if $a \in I(x, 0)$ *then* $a \in S \equiv (a = \mathbf{key}(\mathbf{forward}(x, 1)))$.

Each node/level (x, ℓ) defines a sub-skip list denoted $S_{(x,\ell)}$. It is composed by all keys belonging to $I(x, \ell)$, its level is ℓ, the header is x and the **NIL** node is $\mathbf{wall}(x, \ell)$. Then the preceding lemma can be rewritten by saying that a key that belongs to $S_{(x,\ell)}$ also belongs either to $S_{\mathbf{forward}(x,\ell),\ell}$ or $S_{(x,\ell-1)}$. We define the number

$$\mathbf{inside}(x, \ell) \equiv \text{ the size of the sub-skip list } S_{(x,\ell)}.$$

The Skip lists are generated by the forthcoming updates algorithms starting on an empty list. As the insertion algorithm is random, this data type should be denoted (random)Skip lists; but following W. Pugh we refer to them as Skip lists [47].

The random distribution considered has been the negative binomial distribution with random parameter q, also designated Pascal or geometric distribution, and denoted $NB(1, q)$. Recall that a negative random variable is the number of failures observed before a success in a series of independents trials, where the probability of success in a trial is q. Let X be a random variable with this distribution $NB(1, q)$, then $Prob\{X = k\} = p^k q$ and the expected value is p/q being $p + q = 1$. [1]

The levels of nodes of a Skip lists with parameter q is given by the independent distributed random variables $NB(1, q) + 1$ with expected mean $1/q$.

We give now the algorithms and the "**local**" rules. We use quotes around local because some rules read global variables. These rules appear implicitly in the work of Pugh [45] about concurrent Skip lists.

Search

Algorithm: Given S and a key a, the search procedure returns the unique *node* in S such that $\mathbf{key}(node) < a \leq \mathbf{key}(\mathbf{forward}(node, 1))$. It works moving the key a forward or down through S until it reaches *node*. In any given stage the key is said to be at a node/level (x, ℓ), designated the *current node/level*. Initially the current node/level is set to $(\mathbf{header}(S), \mathbf{level}(S))$. The search procedure iterates and maintains the invariant $a \in I(x, \ell)$ until the current level ℓ is 0. In each iteration the current

[1] Some authors count the number of trials instead of the number of failures (see [44]).

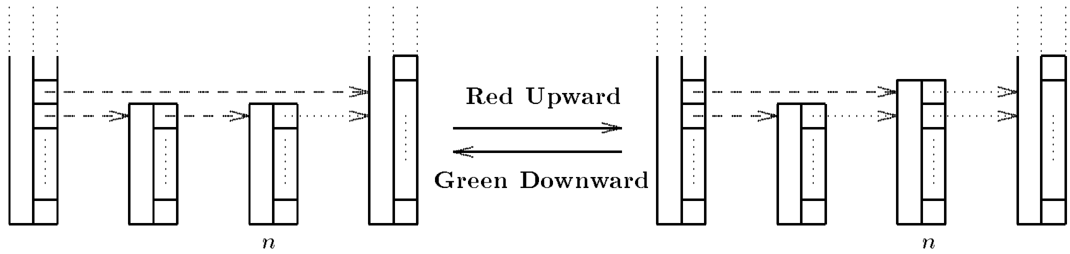


Figure 2.11: Rules of Skip lists

node/level is changed according to Lemma 2.1. It is useful to collect the key, the current node/level and the loop condition in a *packet* p

$$p \equiv \langle a, x, \ell, (\ell > 0)\rangle \equiv \langle \mathbf{key}(p), \mathbf{node}(p), \mathbf{level}(p), \mathbf{active}(p)\rangle,$$

and rephrase the search algorithm in terms of a packet moving through the skip list, until the packet becomes inactive.

Extraction of "local" rules: Like in search trees we add the **waiting** bag to each node/level and define the rule

Rule : Percolation

Guard: *Node/level* (n, ℓ) *such that* $\mathbf{waiting}(n, \ell) = \{x\}$ *and* $\ell > 0$.
Behaviour: *The keys* x *is sent forward or down.*
Variant function: $\mathbf{inside}(n, \ell) \cdot \ell$.
Spatial scope: *Node/level* n *and* $\mathbf{forward}(n, \ell)$.

Insertion

Algorithm: Assume, w.l.o.g., that the key a to be inserted does not belong to S. The insertion has three main phases. First, we search for a to locate the insertion point for the new item. It is also necessary to collect information about the would-be predecessors of the new item in each list. This can be done by means of an array called *update* (see [47]) such that

$$\begin{aligned} update[\ell] \quad &= \quad \text{"the unique node in } S \text{ such that} \\ &\quad \mathbf{key}(update[\ell]) < a \leq \mathbf{key}(\mathbf{forward}(update[\ell], \ell))\text{"}, \quad 1 \leq \ell \leq \mathbf{level}(S). \end{aligned}$$

As $update[\ell]$ is the last visited node at level ℓ during the search for key a, it is easy to modify the standard search procedure to compute also the *update* array as defined above. Second, a random level is chosen and a new node is allocated for the new item. Finally, the third phase modifies the necessary links to add the new node, by using the *update* array.

Extraction of "local" rules: Note that the height of a new node increases until it reaches the random chosen height, which is stored into a global variable. We define three rules:

Rule : Percolation (as in searchs),

translates the keys through the waiting bags until it reaches level zero. After a new node is created with

Rule : Red Attach

Guard: *Node/level* $(n, 0)$ *such that* **waiting**$(n, 1) = \{x\}$.

Behaviour: *If* **forward**$(n, 1) = x$ *then remove* x, *otherwise insert between nodes* n *and* **forward**$(n, 1)$ *a new node with level one, compute randomly the level* L *of the node, and if the level is bigger than one colour node red.*

Variant function: NWAITING.

Spatial scope: *Node/levels* $(n, 0), (n, 1)$.

and finally we increase its height with

Rule : Red Upward

Guard: *Node* n *such that* **red**(n) *(Figure 2.11).*

Behaviour: *Increase the level of* n *by one, reconstruct the Skip list* S *and if the level* L *is bigger than* **level**(n) *then colour node red.*

Variant function: $L -$ **level**(n).

Spatial scope: *node* n *and the predecessor.*

Deletion

Algorithm: It goes along the same lines as the insertion algorithm. After the search for the item to be deleted, we colour it green.

Extraction of "local" rules: with rule

Rule : Percolation (as in searchs),

we reach the highest level/node of node, we apply the inverse of **Red Upward** defined as

Rule : Green Downward

Guard: *Node* n *such that* **green**(n) *and* **level**$(n) > 1$ *(Figure 2.11).*

Behaviour: *decrease the level of* n *by one and reconstruct the Skip list* S. *If necessary arrange* **level**(S). *Color* n *green.*

Variant function: **level**(n)

Spatial scope: *Node* n *and predecessor.*

Finally the node with level one is unattached by rule

Rule : Green Unattach

Guard: *Nodes* n *such that* **green**(n) *and* **level**$(n) = 1$

Behaviour: *remove* n *and arrange the predecessor.*

Var: GREEN.

Spatial scope: *node* n *and predecessor.*

Chapter 3

Establishment of objectives

In this chapter we explain the objectives of this dissertation and the previous approaches which established them. The first two sections are devoted to a brief description of previous works on distributed and massively parallel algorithms to dinamically mantain a dictionary, namely those by J.L.W. Kessels [25] and W. Paul, U. Vishkin and H. Wagener [43]. We rewrite that works with our rules, but we cannot address them by using only the sequential local rules extracted in the previous chapter because these rules do not cover all states of the parallel computation. We need new rules either to avoid dead-locks on distributed algorithms or to deal with concurrent access to nodes on massively parallel algorithms. The third section contains a detailed explanation about our method for designing of parallel algorithms which allows an unified framework for distributed and massively parallel algorithms. The fourth section includes our algorithmic proposals, namely the parallel algorithms we wish to design. Finally, the last section contains some historical notes.

3.1 Distributed algorithms

As it was pointed out in the introduction, we are interested on partial scope distributed algorithms with local rules. Recall that the meaning of partial scope is that update processes do not balance the structure, then the distributed algorithm need balance processes. We represent processes with local rules. J.L.W. Kessels was the first one to apply this approach to search trees. He was inspired by E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten and E.F.M. Steffens [14] who address the garbage collection problem. We sketch this last paper, and next we rewrite the J.L.W. Kessels [25] distributed algorithm.

E.W. Dijkstra *et al.* addressed the **Garbage collection** problem in a **multiprocessor** environment with an **organizing in cooperation** strategy in 1978 [14]. They assumed that the *main algorithm* dynamically puts nodes onto the structure and takes them away, and the *garbage collector* marks the removed nodes and gathers them together into a garbage list. Both algorithms are executed concurrently but by keeping exclusion and synchronization constraints as weak as possible: the ultimate aim is no more interference that the mutual exclusion of a single read and write of the same variable. They introduce the term **shared variables** for those variables accessed by at least two processes, and **atomic operations** for those operations that cannot be interfered by other ones, and force shared variables to be updated by atomic actions. They designed the garbage collector with a set of atomic operations controlled by a *garbage algorithm*, and a set of invariant predicates which serves to prove the algorithm correctness. Note that the *main algorithm* and the *garbage collection* can be viewed as components of a major distributed algorithm.

J.L.W. Kessels translates this point of view to algorithms on balance search trees. As an insertion algorithm is composed by a *percolate-and-attach* phase and a *balance* one, it is possible to interpret the first phase *percolate-and-attach* as the *main algorithm* and the second phase *balance* as the arrangement phase done by a set of atomic operations. Then the *garbage collector algorithm* can be viewed as a background scheduler that scans nodes in arbitrary order and applies the balance rules. The correctness proofs are based on a set of invariant predicates.

3.1.1 J.L.W. Kessels algorithm on AVL trees

We present the work of J.L.W. Kessels [25] because of his influence in our own work. He adressed the design of a concurrent insertion algorithm on AVL trees by defining local rules that balance the tree concurrently with the *search_and_attach* processes. Note that many such balancing processes can be running in parallel at a given moment.

He coloured red the new attached nodes and introduced the **dynamic height** of a tree which does not take into account these coloured nodes. He defined

$$\mathbf{dheight}(nil) = 0$$
$$\mathbf{dheight}(n) = \max(\mathbf{dheight}(\mathbf{left}(n)), \mathbf{dheight}(\mathbf{right}(n))) + 1 - \mathbf{red}(n).$$

where $\mathbf{red}(n) = 1$ if n is a red node and 0 otherwise. This notion serves him to define the **dynamic balance** of a node as

$$\mathbf{dbal}(n) = \max(\mathbf{dheight}(\mathbf{right}(n)) - \mathbf{dheight}(\mathbf{left}(n))) + 1,$$

then balance constraints do only apply to uncoulored nodes. Note that an AVL tree with red nodes, denoted **dynamical** tree, may not be an AVL tree because of coloured nodes, which unbalance the tree by increasing the height of some nodes. So if the red nodes are omitted, *i.e.* they do not increase the height, then the tree is an AVL tree.

The rules of the distributed algorithm follow from the invariant predicates

$$P_1 \equiv \mathbf{dbal}(n) \in \{-1, 0, 1\}$$
$$P_2 \equiv \mathbf{dheight}(n) \neq 0 \ \wedge \ \mathbf{red}(n) \implies \mathbf{dbal}(n) \neq 0$$

The first predicate defines what we denote **dynamical** AVL tree and forces on rules to compose propagation with rotation; and the second rule forces the internal red nodes (non-leaves) to be unbalanced.

Assume that several new keys are located at the bottom of the tree and coloured red. The rules must send up the colour until it disappears. We address now the rules which deal with only one red node. Depending on the balancing aspects J.L.W. Kessels defines three types of rules. Let n be the left son of m such that $\mathbf{red}(n)$ and $\neg\mathbf{red}(m)$:

1. $\mathbf{dbal}(m) \in \{0, 1\}$; no balancing. He uses the

   Rule : Red Propagation (as the sequential rule of AVL trees (Page 22)),

   but adding the predicate $\neg\mathbf{red}(n)$ as a guard. Note that in sequential environment this condition holds implicitly.

2. $\mathbf{dbal}(m) = -1$ and $\mathbf{dbal}(n) = -1$ balancing with single rotations. This case couples propagation of one red node with single rotations (Page 23). As only one red node is propagated these rules can be replaced by a composition of **Red Propagation** with **Red Rotation** (single case) rules of the sequential algorithm, so we denote it

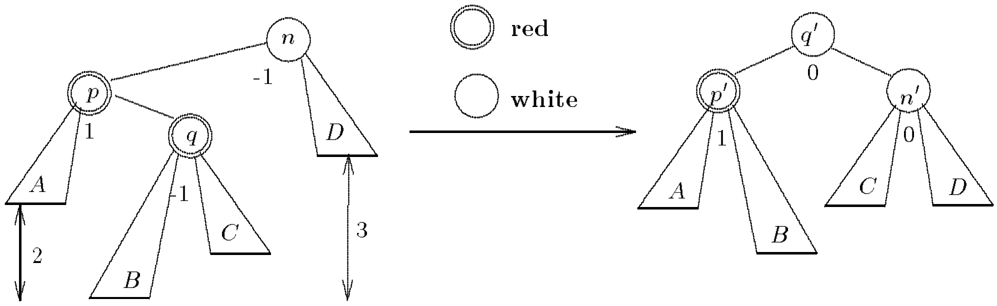


Figure 3.1: **Consecutive Red Nodes** rule (left case)

Rule : Red Rebalancing (single case).

3. $\mathbf{dbal}(m) = -1$ and $\mathbf{dbal}(m) \neq \mathbf{dbal}(n)$ requires balancing with double rotations. Double rotations take into account three nodes; then, either one or two of them can be red. In the first case, where only one node is red, the rule is obtained by composing **Red Propagation** with **Red Rotation** (double case) and is denoted

   Rule : Red Rebalancing (double case).

We address now the rule which deals with consecutive red nodes; we need to prevent the *deadlock*.

Rule : Consecutive Red Nodes

Guard: *A subtree* $n(p(A, q(B, C)), D)$ *with* $\mathbf{red}(p)$, $\mathbf{red}(q)$, $\mathbf{dbal}(n) = -1$ *and* $dbal(p) = 1$.

Behaviour: *The subtree* $q'(p'(A, B), n'(C, D))$ *is obtained. If* $\mathbf{dbal}(q) = -1$ *(see Figure 3.1), then update* $\mathbf{red}(p')$, $\mathbf{dbal}(p') = 1$ *and* $\mathbf{dbal}(q') = \mathbf{dbal}(n') = 0$. *Otherwise* $dbal(q) = 1$, *then update* $\mathbf{red}(n')$, $\mathbf{dbal}(p') = \mathbf{dbal}(q') = 0$ *and* $\mathbf{dbal}(n') = -1$.

Variant function: RED.

Spatial scope: *Nodes* n, p, q.

Theorem 3.1 *(J.L.W. Kessels):*

1. *The algorithm is deadlock free.*
2. *(Safety property) If no rule applies then the tree is an AVL tree.*
3. *(Liveness property) Eventually no rule applies.*

Proof: *The safety property follows from the invariance of predicates* P_1 *and* P_2 *and the guards of rules. The liveness follows from the number*

$$\mathrm{LOSS} = \sum_{\mathbf{red}(n)} \mathbf{outside}(n)$$

*and the claim that the variant function (*RED,LOSS*) strictly decreases at each application of any rule.* ■

3.2 Massively parallel algorithms

We are interested on those algorithms that follow from the k-PRAM model [3]. This model consists of k processors and a shared memory, where each processor is a random-access machine. It is a synchronous model because all processors are controlled by the same algorithm and execute the same sentence on different data. Our aim is to resolve the memory access conflicts with the exclusive-read exclusive-write (EREW) policy.

W. Paul, U. Vishkin and H. Wagener [43] designed the first , and for severals years the only one, massively parallel algorithm on balanced search structures, namely 2–3 trees. Their ideas were applied to B-trees by L. Higham and E. Schenk [22] in 1994 with the improvement focused on the efficient management of large nodes.

3.2.1 W. Paul, U. Vishkin and H. Wagener algorithm on 2–3 trees

They assume that the set of k keys to be searched or updated are sorted and distinct. Note that that last assumptions are non-restrictive because, in this model, the sorting may be done in time $O(\log(k))$.

Search

They introduced the technique of *routing packets* of keys. As it is widely explained in chapter 4, we only sketch it. They introduce the notion of *chain* (which we call **packet** from now) that contains subsets of consecutive keys. Although there is a processor for each key, packets can be managed by only one processor. We define for any node n the number

$$|\text{packets}(n)| = \text{number of keys contained into packets located at node } n$$

The algorithm starts at the root by leaving a packet with all keys. At any stage the packets are routed or split through the tree until they arrive at the deepest nodes, denoted **bottom** nodes.

Rule : Percolation on Packets

Guard: *Node n that is pointed to by packets and* $\neg\mathbf{bottom}(n)$.

Behaviour: *Split and send the packets.*

Variant function: $\sum_n \mathbf{depth}(n)|\text{packets}(n)|$

Spatial scope: *Node n.*

Insert

The *search* phase translates packets to bottom nodes. As bottom nodes store two (one) keys, they have at most three (two) packets which hang from it. We define the number

$$\text{PACKET} = \sum_n |\text{packets}(n)| = \text{number of keys contained into packets.}$$

The massively parallel attach of nodes is performed by rule:

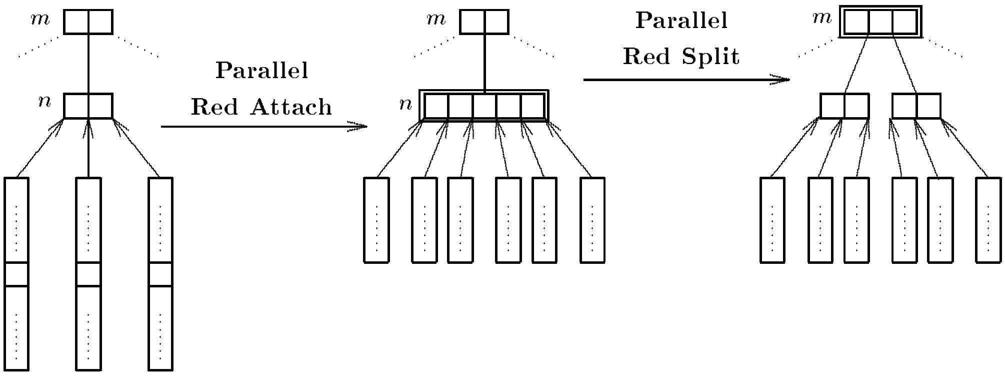


Figure 3.2: Parallel red rules

Rule : Parallel Red Attach

Guard: *Node n such that* $\mathbf{bottom}(n)$ *and has attached packets.*

Behavior: *Node n selects the middle key of each packet, inserts it into itself and , splits the packets. If n stores more than two keys then colour it red. (Figure 3.2, first transformation).*

Variant function: PACKET.

Spatial scope: *Node n.*

After that, the algorithms applies 3 times the following rule

Rule : Parallel Red Split

Guard: *Node m with, at least, one red son.*

Behavior: *Select one key of each red son and store them into n. Then attach the split nodes adequately. If it has more that two keys colour it red. (Figure 3.2, second transformation).*

Variant function: RED, $L - \mathbf{height}(n)$

Spatial scope: *Node n and its red sons.*

Note that all leaves have the same depth and that the following predicate remains invariant

$$\mathbf{depth}(n) + \mathbf{height}(n) = \mathbf{height}(\mathbf{root}) = \mathbf{depth}(nil).$$

At each stage a **dynamic plane red wave** is created at leaves because all parallel attach rules are synchronously applied over nodes with same depth. Then, the wave is sent up 3 steps by applying the parallel red split rule. Then the algorithm pipelines waves whilst there are non empty packets. Consequently, the balance phase can be envisaged as a **stationary travelling waves** with **wave length** 3.

The algorithm correctness is based on the following lemmas:

Lemma 3.1 *(Plane wave) All red nodes that belong to the same wave have the same height and depth.*

Lemma 3.2 *The waves do not modify the height of nodes.*

Lemma 3.3 *(Liveness) At each stage, the height of red nodes strictly increases but it is bounded by the height of the tree.*

Finally they prove the following theorem:

Theorem 3.2 *(W. Paul, U. Vishkin and H. Wagener) The algorithm inserts k keys into a tree with n keys in time $O(\log(n) + \log(k))$.*

Proof: *At each stage the length of packets is divided by two; then, at most $O(\log(k))$ waves will be created. Each wave takes time $O(log(n))$* ■

Delete

The search phase translates packets to bottom nodes. The rule **Parallel Request Manager** creates green waves that are sent up by rule **Parallel Green Propagation**. If we add to each node the maximum key of its subtrees, the key to be deleted can be substituted for it.

Rule : Parallel Request Manager

Guard: *Node n that is pointed to by packets and* **bottom**(n).

Behavior: *If all leaves of n must be deleted, then select one of them, remove the others and colour node green. Otherwise remove all deleted keys and if $|key(n)| = 1$ colour node green.*

Variant function: PACKET.

Spatial scope: *Node n.*

Rule : Parallel Green Propagation and Updating

Guard: *Node n whith some green son.*

Behavior: *Rearrange* **parent**(n) *and colour green if* $|\mathbf{parent}(n)| = 1$.

Variant function: $\sum_{\mathbf{green}(n)}(L - \mathbf{height}(n))$.

Spatial scope: *Nodes* **parent**(n), *n and their brothers and sons.*

Note that at each stage one key is left, at least, at each leaf, and if these keys must be deleted, then the algorithm pipeline waves. When the green wave arrives to the root, then it is removed by rule **Green Root**.

Theorem 3.3 *(W. Paul, U. Vishkin and H. Wagener) The algorithm deletes k keys from a tree with n keys in time $O(\log(n) + \log(k))$.*

3.3 Common design framework

We propose a method that allows us to unify the design of parallel algorithms with local rules on balanced search trees. We underline that

Our goal in distributed algorithms is to design rules by keeping the invariants as weak as possible, and in massively parallel algorithms is to achieve time costs as good as those for 2–3 trees.

We extract from the previous approaches the following two facts:

J.L.W. Kessels defines local rules for AVL trees in order to maintain the invariance of two predicates. Some rules come from the set of sequential ones, such as **Red Propagation** or **Red Rebalancing**, and some others are created to avoid deadlock, such as **Consecutive Red Nodes** rule.

W. Paul, U. Vishkin and H. Wagener create a waves pipeline which maintain some invariants predicates. Each wave is composed of local parallel rules, such as **Parallel Red Attach** or **Parallel Red Split**, which are the natural extension of sequential rules **Red Attach** and **Red Split**.

Then it can be possible to create an unified framework to distributed and massively parallel algorithms design, based on local rules. We propose the follow

Common design framework

1. *Extract local rules from the sequential algorithms*, as it has been done in chapter 2.

2. *Design the distributed algorithm using that set of local rules.* This means

   (a) Specify an invariant predicate,

   (b) Extract and design sequential local rules in order to maintain the predicate invariant.

   (c) Design local rules to avoid deadlock.

   Deletion algorithms on trees cannot be designed from sequential rules if they use pointers to far nodes. In this case, *create new rules to deal with deletions.*

   The correctness criterion for the distributed algorithm follows from the invariant and is based on proofs of:

   Deadlock free: If at leats one guard is open, then some rule can be applied.

   Safety property: If all guards are closed, then the data type is well constructed.

   Liveness property: Eventually no rule applies, *i.e.* there is no infinite sequence of rule application.

3. *Design the data parallel algorithm with a specific parallel scheduler based on that set of local rules.* This means that we generalize the sequential rules to deal with parallel cases and we design a waves pipeline. The correctness follows from a predicate invariant and is based on:

   Invariance of waves: Each wave maintains the invariants.

   Independence of waves: The support of different waves is disjoint, *i.e.* different waves access different set of nodes.

The benefits of this method should include the achievement of distributed algorithms which are more fine-grained than the existing ones and of massively parallel algorithms as fast as the existing ones.

3.4 Objectives

Recall that our aim is the design of parallel algorithms with local rules for the most commonly used search trees. Namely those search trees which have different set of rules and are representatives of many others, such as Skip lists, 2–3 trees and AVL trees. (see Tablepage 1.1).

We explain the proposals for each data structure selected.

Skip lists: We address the design of the massively parallel algorithms. Recal that the sequential set of rules, extracted in section 2.5, read global variables. This fact, which makes difficult the massively parallel design, also makes impossible the design of a distributed algorithm with local rules. We solve this problem by storing the global variables information into local registers of nodes. As these informations are composed of pointers to nodes, this recording suggests the creation of a new data type with a tree structure, with similar performances than Skip lists and denoted **Skip trees**. Finally, we design a distributed algorithms with local rules for this new data type.

2–3 trees: Recall that these trees have two basic properties: there are 2 or 3 keys at each node, and all leaves have the same depth.

We address the design of a distributed algorithm by taking only the second property as invariant. This means that we admit nodes as wide as necessary. The distributed algorithm will be composed of the rules

{ **Percolation** , **Red Attach** , **Red Split** ,
Whitening Red Root , rules to delete keys }

The algorithm design is similar to that of Skip trees. Because of this reason we include this design in the chapter devoted to Skip trees. Conversely, O. Nurmi, E. Soisalon-Soininen and D. Wood have developed their design by maintaining the first property and by introducing unary nodes between parents and sons (therefore leaves have different depth).

Finally we address the analysis of the parallel evolution of the 2–3 trees leaves,*i.e.* once several parallel insertions have been made. In fact we wish to extend the fringe analysis from the sequential case to the parallel case.

AVL trees: After J.L.W. Kessels, there are the improvements made by O. Nurmi, E. Soisalon-Soininen and D. Wood, and K.S. Larsen, but in both cases the rotation should be made after some propagation. We design the first distributed algorithm that uncouples rotations from propagations whith the set of rules:

{ **Percolation** , **Red Attach** , **Red Propagation** ,
Whitening Red Root , **Red Rotation** , rules to delete keys }

Finally we address the design of the massively parallel algorithm. It has more disadvantageous aspects than the 2–3 trees design because, the leaves depth is different and the waves modify the height of interior nodes. These facts forces us to define *virtual* magnitudes that difficult the design comprehension.

3.5 Historical notes

Recall that we separate distributed algorithms into two groups. First, the **global scope** group, which includes those algorithms which emulate sequential algorithms, *i.e.* that leave the data type correctly structured after its execution. Second, the **partial scope** group, which includes those algorithms whose update process does not balances the structure.

The first distributed algorithms designed on balanced search trees belong to the **global scope** approach, such as those for B-trees designed by Bayer and Schkolnick [7] in 1977, Lehman and Yao [31] in 1981 or Kwong and Wood [29] in 1982, and those for 2–3 trees [16] and AVL trees [17] by Ellis in 1980. All of them are inspired on sequential algorithms and incorporate different type of locks depending on the task the process has to perform. Recall that searchers lock a small and bounded number of nodes, but updaters could lock larger and unbounded parts of the data structure. Then the major improvement has been focused on the diminution of the locked zone. Some complex algorithms were designed to that aim but their correctness were very difficult to prove.

The **partial scope** approach was first undertaken by H.T. Kung and P.L. Lehman [28]. They gave safe concurrent algorithms to deal with insertions and deletions in binary search trees. They did not wish to restrict to any specific type of balanced tree. In a footnote, they suggest to attack this problem using the dichromatic trees introduced by L.J. Guibas and R. Sedgewick [21]. Recall that in the dichromatic approach, tree nodes are painted red or black. Red nodes do not count to compute height: all paths from an internal node to external nodes contain the same number of black nodes. Later on, inspired by on-the-fly garbage collection algorithms [14], J.L.W. Kessels [25] found the first safe and live algorithm. This approach was further applied by O. Nurmi, E. Soisalon-Soininen and D. Wood on AVL trees [37] and on B-trees [36] in 1987, and by O. Nurmi and E. Soisalon-Soininen on chromatic binary search trees in 1991 [35].

Chapter 4

Skip lists: massively parallel algorithms

In this chapter we present a top down design and implementation of massively parallel algorithms using Skip lists (PRAM dictionaries). Recall that Skip lists are composed of a set of linked lists which can be accessed as search trees and are balanced through randomization. As Skip lists do not support local rules, we do not refer to them in this chapter.

The first section gives detailed algorithms to search for, insert and delete k keys in a Skip lists in parallel, and the second section contains a discussion of some implementation issues in `C*`.

4.1 Algorithmic design

In this section we design PRAM EREW algorithms to search, insert and delete an ordered set of k keys on a Skip list of size n.

We recall that in the PRAM model there is a collection of independent synchronizated sequential process with a unique identifier for each one, and a shared memory which stores the data. At each time, each processor can read a location in the shared memory, perform a computation and then write to a location on the shared memory. In the EREW PRAM model no two processors can simultaneous read or write the same location. Conversely, in the CREW PRAM model processors can simultaneous read but they cannot simultaneous write.

Our algorithms run in time $O(\log(n) + \log(k))$ as those of W. Paul, U. Vishkin and H. Wagener. The assumption that the k keys are ordered and have no duplicates is nonrestrictive because the time needed to sort k keys and to remove the duplicates with k processors is $O(\log(k))$.

4.1.1 Massively parallel searchs

4.1.1.1 Algorithm

The algorithm searches for k keys, given in an ordered array $[1..k]$, by routing packets along the skip list S. A similar approach was developed for 2–3 trees by W. Paul, U. Vishkin and H. Wagener [43] and by Higham and Schenk [22].

Fundamental to our search algorithm is the notion of *packet*. A packet p stores a current node/level, an active status, and two indexes i and j, $1 \le i \le j \le k$, representing

```
var p : packet;
    P : set_of_packets;
    i : integer;

p := ⟨[1..k], header(S), level(S), true⟩;
P := {p};

do active(P) ≠ ∅
   for all p : p ∈ active(P) do in parallel
         route(p, P)
   end
end

for all p : p ∈ P do in parallel
      for all i : first(p) ≤ i ≤ last(p) do in parallel
            node[i] := node(p)
      end
end
```

Algorithm 4.1: Search algorithm

the subarray $a[i..j]$ of $a[1..k]$. We write $\text{first}(p) = i$ and $\text{last}(p) = j$. Formally,

$$p \equiv \langle[\text{first}(p)\ ..\ \text{last}(p)], \text{node}(p), \text{level}(p), \text{active}(p)\rangle.$$

We denote by P the set of all the packets that exist in any given stage of the search. This set induces a full partition of the array $a[1..k]$, *i.e.* , each key is "held" by one and only one packet. This property is mantained along the search. A packet p is said to be active if its level is not null: $\text{active}(p) \equiv (\text{level}(p) > 0)$. Inactive packets are those that have reached their final "destination". The subset of active packets is denoted by $\text{active}(P)$.

Given a skip list S of n keys and an ordered array $a[1..k]$ that contains the k keys, the search algorithm returns an array $node[1..k]$ such that:

$$\begin{aligned} node[i] \quad = \quad & \text{a pointer to the unique node } x \text{ in } S \\ & \text{such that } \mathbf{key}(x) < a[i] \leq \mathbf{key}(\mathbf{forward}(x,1)), \quad 1 \leq i \leq k. \end{aligned}$$

The search algorithm is specified by

```
AKey = array [1..k] of TKey
AItem = array [1..k] of TItem
Procedure search (in a: AKey; in S: SkipList; out node: AItem;
  PRE: {ordered(a)}
  POST: {∀i ∈ [1..k]: node[i] points to adequate node}
```

At the very start of the algorithm (see Algorithm 4.1), an active packet containing all k keys is "injected" into the skip list S by making $(\mathbf{header}(S), \mathbf{level}(S))$ its current node/level. In each stage, each active packet is routed by moving it forward or down, or it is split into two packets and one of these is moved forward or down. The main loop of the search ends when all the packets become inactive. At most k processors are needed to execute this loop; we need one processor to route in parallel each active packet. After the main loop, the array *node* is filled by spreading the current node of each packet to its keys (see Algorithm 4.1).

We now describe each procedure used by the search algorithm in more detail.

```
Procedure route(in p : packet; in/out P : set_of_packets)
  var b : TKey;

  b := key(forward(node(p), level(p)));
  if
    b < a[first(p)] —→push_forward(p)
    b ≥ a[last(p)] —→push_down(p)
    a[first(p)] ≤ b < a[last(p)] —→split_and_push (p, P)
  fi
end route


Procedure split_and_push(in p : packet ; in/out P : set_of_packets)
  var b : TKey; p1, p2: packet; m : 1..k;

  b := key(forward(node(p), level(p)));
  m := (first(p) + last(p)) div 2;
  split_packet(p, m, p1, p2);
  P := P − {p} ∪ {p1, p2};
  if
    b < a[first(p2)] —→push_forward (p2)
    b ≥ a[last(p1)] —→push_down(p1)
  fi
end split_and_push
```

Algorithm 4.2: Route algorithm

Procedure route: At each stage, all active packets are routed through S. The procedure *route*(p, P) moves or splits p in order to mantain the invariant of the main loop:

$$\forall p: \ a[\text{first}(p) \ .. \ \text{last}(p)] \subseteq I(\text{node}(p), \text{level}(p)).$$

(Recall the definition of $I(x, l)$ in section 2.5). Given a packet p whose current node/level is (x, l), the procedure *route* compares $a[\text{first}(p)]$ and $a[\text{last}(p)]$ with $b = \mathbf{key}(\mathbf{forward}(x, l))$. Then it pushes forward, down or splits p, depending on the outcome of the comparisons, using the procedures *push_forward, push_down* and *split_and_push*, respectively (see Algorithm 4.2).

Procedure push_forward: If $a[\text{first}(p)] > b$ then all the keys in the packet p must be in $I(\mathbf{forward}(x, l), l)$ provided that all them were in $I(x, l)$, by Lemma 2.1. By hypotesis, this is indeed the case and the packet must be forwarded replacing the current node of p, x, by $\mathbf{forward}(x, l)$.

Procedure push_down: Assume now that $a[\text{last}(p)] \leq b$. Then for any key a in p it holds that $a \in I(x, l-1)$, and the packet must be pushed down, decrementing its level. If the level of p becomes 0, then the packet is made inactive, so it will not be routed in the next stages.

Procedure split_and_push: Finally, b could *hit* the packet, i.e. $a[\text{first}(p)] \leq b < a[\text{last}(p)]$. In that case *route* calls procedure *split_and_push*. It first calls *split_packet*, which halves $p = \langle [i..j], x, l, true \rangle$ into two packets $p_1 = \langle [i..m], x, l, true \rangle$ and $p_2 = \langle [m+1..j], x, l, true \rangle$, with $m = (i+j)$ **div** 2. The key b must not hit at least one of p_1 and p_2. Therefore, either it is true that all the keys in p_1 are in $I(x, l-1)$ or all the keys in p_2 are in $I(\mathbf{forward}(x, l), l)$. Hence, one of these packets can be pushed forward or down applying one of the rules for the procedures above, while the other remains at the same node and level.

While active$(P) \neq \emptyset$, the *main loop* of the search executes the sentence:

for all $p : p \in \text{active}(P)$ **do in parallel** route(p, P) **end**

We call an execution of this **for all** a *stage*. We assume that stages are succesivelly numbered. The first execution of the **for all** is stage 1, the second one is stage 2, ... Hence, the notion of a packet p located at (x, l), *i.e.* the value of its current node/level is (x, l), at the begining (at the end) of stage t is well defined. Similarly, if a packet p belongs to active(P) before the execution of stage t and *route*(p, P) pushes it forward, we say that p goes from (x, l) to $(\mathbf{forward}(x, l), l)$ during stage t. We should also say that p is forwarded (or moves forward) from (x, l) to its successor during stage t. If p is at (x, l) at the beginning of stage t and *route*(p, P) pushes it down, we say that p at stage t goes (moves down) from (x, l) to $(x, l - 1)$ during stage t. Finally, if p is at (x, l) at the beginning of stage t and *route*(p, P) applies *split_and_push* to it, we say that p splits at this stage. Notice that both the set of packets and the set of active packet may change during any stage t. In fact, if we consider the sum of the distances of each of the keys $a[i]$ to their final destinations (measured in number of *push_forward*'s and *push_down*'s) its value decreases after each stage and when active$(P) = \emptyset$ the value of this sum is zero.

4.1.1.2 Analysis of read conflicts

During any given stage all active packets perform the statement $b := \mathbf{key}(\mathbf{forward}(x, l))$. Hence, read conflicts arise whenever several packets are at the same node/level (x, l). It is clear that each stage takes time $O(1)$ in a CREW model, but we will like to execute each stage in a EREW PRAM using constant time. We prove in this subsection that there are at most three packets at a given node/level of the skip list in any stage of the search. Later, we show an explicit protocol that allows the execution of each stage in constant parallel time without read conflicts.

First of all, we introduce several useful definitions and notation to study the relationship between packets and their flow throught the skip list. For each packet p, we define two new fields remains(p) and split_side(p). If p has been generated by an split operation during stage t at (x, l) and it remains at this same node/level at the end of stage t then remains(p) is true; otherwise, it is false. The value of the split_side field depends on the way the packet has been generated. If p has been generated splitting a packet q and first(p) = first(q) then split_side(p) = right; if last(p) = last(q) then split_side(p) = left. By convention, the split_side of the initial packet $\langle [1..k], \mathbf{header}(S), \mathbf{level}(S), \ldots \rangle$ is left. Recall that, except for this first packet, all packets are generated by the split of some other.

Given a packet p and one of its fields f, the value of p at the end of stage t will be denoted $f_t(p)$. We assume that stage 0 ends before we enter the main loop of the search for the first time. Let us consider in detail the set of packets located at (x, l) at the end of stage t:

$$\{p \mid (\mathbf{node}_t(p), \mathbf{level}_t(p)) = (x, l)\}.$$

A packet p can belong to the set above due to two different reasons: either it was generated by the split of a packet located at (x, l) during stage t and p remains in (x, l), or p arrives to (x, l) from another node/level. In the first case remains$_t(p)$ is true, whilst in the second

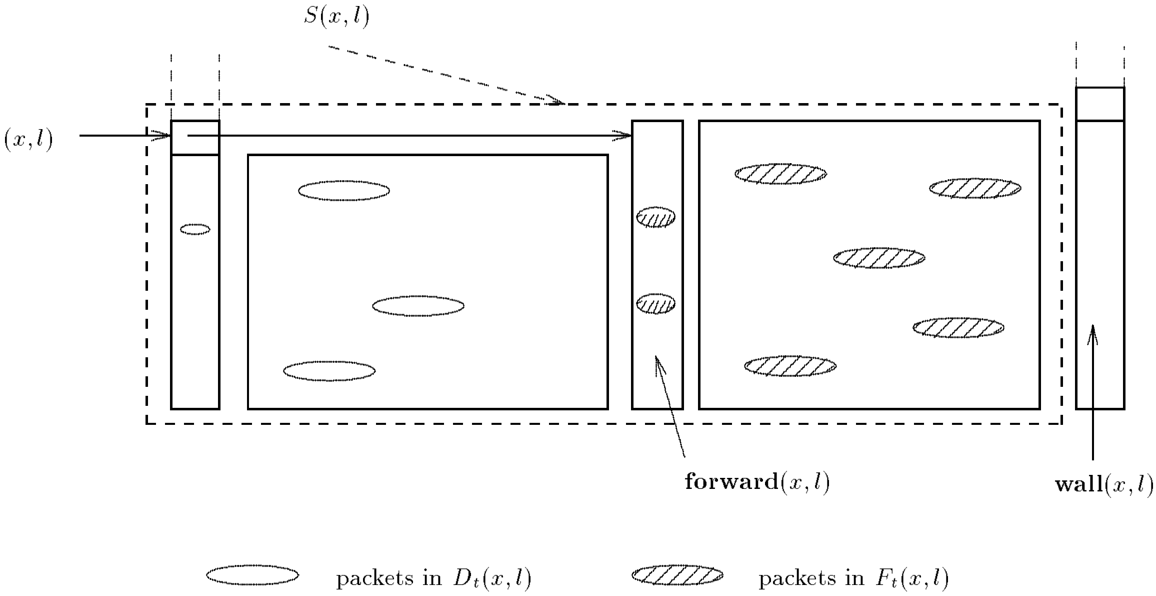


Figure 4.1: The sets of packets $F_t(x,l)$ and $D_t(x,l)$

case $\text{remains}_t(p)$ is false. Hence,

$$\begin{aligned}\{p \mid (\mathbf{node}_t(p), \mathbf{level}_t(p)) = (x,l)\} &= \{p \mid (\mathbf{node}_t(p), \mathbf{level}_t(p)) = (x,l) \ \wedge \text{remains}_t(p)\} \\ &\cup \ \{p \mid (\mathbf{node}_t(p), \mathbf{level}_t(p)) = (x,l) \ \wedge \neg\text{remains}_t(p)\}.\end{aligned}$$

A little more of structure can be added according to the split side of the packets. We introduce the following notations:

$$\begin{aligned}\mathrm{L}_t(x,l) &= \{p \mid (\mathbf{node}_t(p), \mathbf{level}_t(p)) = (x,l) \ \wedge \neg\text{remains}_t(p) \ \wedge \text{split\_side}_t(p) = \text{left}\} \\ \mathrm{M}_t(x,l) &= \{p \mid (\mathbf{node}_t(p), \mathbf{level}_t(p)) = (x,l) \ \wedge \text{remains}_t(p)\} \\ \mathrm{R}_t(x,l) &= \{p \mid (\mathbf{node}_t(p), \mathbf{level}_t(p)) = (x,l) \ \wedge \neg\text{remains}_t(p) \ \wedge \text{split\_side}_t(p) = \text{right}\}\end{aligned}$$

and have the following partition:

$$\{p \mid (\mathbf{node}_t(p), \mathbf{level}_t(p)) = (x,l)\} = \mathrm{L}_t(x,l) \cup \mathrm{M}_t(x,l) \cup \mathrm{R}_t(x,l)$$

Whenever it is clear from the context, we will omit the explicit mention to the node/level (x,l), thus writing L_t, M_t, etc.

The *subskiplist at* (x,l) of a skip list S, denoted $S(x,l)$, is the skip list of height l, where x acts as a header and $\mathbf{wall}(x,l)$ acts as **NIL**. Node/levels in $S(x,l)$ are those reachable from (x,l). We will denote $\mathrm{P}_t(x,l)$ the set of packets such that its current node/level at the end of stage t belongs to $S(x,l)$. We can keep a record of the packets that have traversed (x,l) from the initial stage up to stage t (see Figure 4.1).

- $\mathrm{F}_t(x,l)$ contains the set of packets located in the subskiplist $S(\mathbf{forward}(x,l),l)$ at the end of stage t. A packet p belongs to $\mathrm{F}_t(x,l)$ iff it has been generated through succesive splits from a packet q that was *forwarded* from (x,l) to $(\mathbf{forward}(x,l),l)$ at some stage $t' \leq t$. The set $\mathrm{F}_t(x,l)$ is defined only if $x \neq \mathbf{NIL}$.

- $\mathrm{D}_t(x,l)$ contains the packets located in the subskiplist $S(x,l-1)$ at the end of stage t. A packet p belongs to $\mathrm{D}_t(x,l)$ iff it has been generated through succesive splits from a packet q that was *pushed down* from (x,l) to $(x,l-1)$ at some stage $t' \leq t$. The set $\mathrm{D}_t(x,l)$ is defined only if $l > 0$.

Formally,

$$\mathrm{D}_t(x,l) = \mathrm{P}_t(x,l-1), \quad \text{for } l > 0 \qquad \mathrm{F}_t(x,l) = \mathrm{P}_t(\mathbf{forward}(x,l),l), \quad \text{for } x \neq \mathbf{NIL}.$$

Note that $\mathrm{P}_t = \mathrm{L}_t \cup \mathrm{D}_t \cup \mathrm{M}_t \cup \mathrm{F}_t \cup \mathrm{R}_t$; moreover, these sets are mutually disjoint. It is immediate to see that each node/level $(x,l) \neq (\mathbf{header}(S), \mathbf{level}(S))$ has a unique *predecessor*, that is, a node/level that precedes it in any search path that passes throught (x,l). We denote the predecessor of (x,l) by $\mathbf{pred}(x,l)$. If $l = \mathbf{level}(x)$ then $\mathbf{pred}(x,l) = (y,l)$ where $\mathbf{forward}(y,l) = x$. Then (x,l) can only "receive" packets coming from (y,l) and $\mathrm{F}_t(y,l) = \mathrm{P}_t(x,l)$. Analogously, if $l < \mathbf{level}(x)$ then (x,l) can only "receive" packets pushed down from $\mathbf{pred}(x,l) = (x,l+1)$ and $\mathrm{D}_t(x,l+1) = \mathrm{P}_t(x,l)$.

In order to compare packets we introduce the following notation:

$$\begin{aligned} p < q &\equiv \mathrm{last}(p) < \mathrm{first}(q), \\ p \subseteq q &\equiv \mathrm{first}(q) \leq \mathrm{first}(p) \ \wedge \ \mathrm{last}(p) \leq \mathrm{last}(q), \\ p \cap q = \emptyset &\equiv [\mathrm{first}(p) \mathrel{..} \mathrm{last}(p)] \cap [\mathrm{first}(q) \mathrel{..} \mathrm{last}(q)] = \emptyset. \end{aligned}$$

If $p < q$ we say that p is *at the left* of q (resp. q is *at the right* of p). A packet p is said to be a *subpacket* of q if $p \subseteq q$. Note that, at the beginning and the end of any stage t, all packets are disjoint, *i.e.* $p \cap q = \emptyset$, for any pair of different packets p, q in P. Two disjoint packets can be always compared in the sense that either $p < q$ or $q < p$ holds. Moreover, if p and q are disjoint, $q' \subseteq q$ and $p < q$ then $p < q'$. The ordering among packets can be easily extended to sets of packets:

$$A < B \equiv \forall p, q : p \in A \ \wedge \ q \in B : p < q.$$

We should note that $<$ is transitive and that both $\emptyset < A$ and $A < \emptyset$ hold for any set of packets A.

Lemma 4.1 *Let p be a packet generated by an split during stage t at (x,l). If p is pushed down or forward during stage t then*

- $\mathrm{split\_side}(p) = \mathrm{left}$ *if and only if p has been pushed forward ($p \in \mathrm{F}_t$).*
- $\mathrm{split\_side}(p) = \mathrm{right}$ *if and only if p has been pushed down ($p \in \mathrm{D}_t$).*

Proof: *Let $b = \mathbf{key}(\mathbf{forward}(x,l))$. Let q be the packet that originates p at stage t. Therefore, $p \subseteq q$ and $a[\mathrm{first}(q)] \leq b < a[\mathrm{last}(q)]$. Moreover, as we assume that p is propagated during stage t, either $a[\mathrm{last}(p)] \leq b$ or $b < a[\mathrm{first}(p)]$. If $\mathrm{split\_side}(p) = \mathrm{left}$, we have $a[\mathrm{last}(p)] = a[\mathrm{last}(q)]$ and $a[\mathrm{last}(p)] \leq b$ cannot hold because this would imply $a[\mathrm{last}(q)] \leq b$, thus contradicting our hypothesis that q was split at stage t. Therefore, $b < a[\mathrm{first}(p)]$ should be true and p must have been pushed forward. If $\mathrm{split\_side}(p) = \mathrm{right}$, we have $a[\mathrm{first}(p)] = a[\mathrm{first}(q)]$ and an analogous reasoning proves that p must have been pushed down. If we assume $b < a[\mathrm{first}(p)]$ it follows that $\mathrm{last}(p) = \mathrm{last}(q)$ and hence, $\mathrm{split\_side}(p) = \mathrm{left}$. On the other hand, if $a[\mathrm{last}(p)] \leq b$ then $\mathrm{split\_side}(p) = \mathrm{right}$, because necessarily $\mathrm{first}(p) = \mathrm{first}(q)$.* ■

Lemma 4.2 *For any node/level (x, l) in S and any stage t: 1) if $\mathrm{L}_t < \mathrm{D}_t \cup \mathrm{M}_t \neq \emptyset$, then any packet $p \in \mathrm{L}_t$ is pushed down in stage $t+1$ ($p \in \mathrm{D}_{t+1}$); 2) if $\emptyset \neq \mathrm{M}_t \cup \mathrm{F}_t < \mathrm{R}_t$, then any packet $p \in \mathrm{R}_t$ is pushed forward in stage $t+1$ ($p \in \mathrm{F}_{t+1}$).*

Proof: *Let $b = \mathbf{key}(\mathbf{forward}(x, l))$. Consider the case where $p \in \mathrm{L}_t$ and $\mathrm{M}_t \neq \emptyset$. As $\mathrm{M}_t \neq \emptyset$ there exists a packet q such that $\mathrm{remains}_t(q)$. Moreover, during stage t there was a packet q' at (x, l) such that $q \subseteq q'$ and $a[\mathrm{first}(q')] \leq b < a[\mathrm{last}(q')]$. Since $p \in \mathrm{L}_t$ the packet p must have been pushed from the predecessor of (x, l) in stage t. Therefore, $p \cap q' = \emptyset$. By hypothesis, $\mathrm{L}_t < \mathrm{M}_t$ so it follows that $p < q$ and $p < q'$. Since $a[\mathrm{last}(p)] \leq a[\mathrm{first}(q')] \leq b$, p must be pushed down during the $(t+1)$-th routing step. Assume now that $\mathrm{M}_t = \emptyset$, but $\mathrm{D}_t \neq \emptyset$. Then, there exists a packet $q \in \mathrm{D}_t$ such that $a[\mathrm{last}(q)] \leq b$. By hypothesis, $p < q$. Hence, $a[\mathrm{last}(p)] \leq a[\mathrm{first}(q)] \leq a[\mathrm{last}(q)] \leq b$ and p must be pushed down in stage $t+1$. For the second claim, when $p \in \mathrm{R}_t$, the proof goes along the same lines.* ■

Lemma 4.3 *Let (x, l) be a node/level such that at the end of stage t, each of $\mathrm{L}_t, \mathrm{M}_t, \mathrm{R}_t$ has at most one packet and $\mathrm{L}_t < \mathrm{M}_t < \mathrm{R}_t$. Then, at most two packets can be propagated from (x, l) to one of its adjacent node/levels in stage $t+1$; moreover, if two packets are pushed from (x, l) to the same node/level then they must have different* split_side.

Proof: *We prove the lemma assuming that $\mathrm{M}_t \neq \emptyset$. The case where $\mathrm{M}_t = \emptyset$ can be proved using a similar reasoning. Let $b = \mathbf{key}(\mathbf{forward}(x, l))$, as usual. By the hypothesis of the lemma and our initial assumption, M_t should be a singleton: $\mathrm{M}_t = \{p_m\}$. As $\mathrm{remains}_t(p_m)$ is true, the packet p_m was generated at (x, l) during stage t from the split of some other packet p'_m. Therefore, $p_m \subseteq p'_m$ and $a[\mathrm{first}(p'_m)] \leq b < a[\mathrm{last}(p'_m)]$. Assume that $\mathrm{L}_t = \{p_l\}$. By Lemma 4.2, the packet p_l will be pushed down to $(x, l-1)$ in stage $t+1$ and $\mathrm{split\_side}(p_l) = \mathrm{left}$ does not change. Similarly, if $\mathrm{R}_t = \{p_r\}$ then p_r will be pushed to $(\mathbf{forward}(x, l), l)$ in stage $t+1$ and $\mathrm{split\_side}(p_r) = \mathrm{right}$. Let us consider now the evolution of p_m in stage $t+1$. There are two possibilities: either p_m is split or it is propagated. If p_m splits into p_{mr} and p_{ml} with $\mathrm{split\_side}(p_{mr}) = \mathrm{right}$ and $\mathrm{split\_side}(p_{ml}) = \mathrm{left}$, one of the following relations holds: $a[\mathrm{last}(p_{mr})] \leq b$ or $b < a[\mathrm{first}(p_{ml})]$. Then one of these subpackets will be pushed forward or down. If p_{mr} is pushed down to $(x, l-1)$ we can conclude that at most one packet, p_r, was pushed forward; and at most two packets, p_l and p_{mr}, have been pushed down during stage $t+1$, their* split_sides *being different: $\mathrm{split\_side}(p_l) = \mathrm{left} \neq \mathrm{split\_side}(p_{mr}) = \mathrm{right}$. If, on the contrary, p_{mr} remains at (x, l) and p_{ml} is pushed forward then we conclude that at most two packets, p_r and p_{ml}, have been forwarded to the same node/level with $\mathrm{split\_side}(p_r) = \mathrm{right} \neq \mathrm{split\_side}(p_{ml}) = \mathrm{left}$. Consider now what happens when p_m is not split in stage $t+1$. Then p_m should be at the left of b (case $a[\mathrm{last}(p_m)] \leq b$) with $\mathrm{split\_side}(p_m) = \mathrm{right}$ or at the right of b (case $b < a[\mathrm{first}(p_m)]$) with $\mathrm{split\_side}(p_m) = \mathrm{left}$. In both cases p_m has a different* split_side *than the other packet (p_l or p_r) that could be pushed to the same node/level.* ■

Lemma 4.4 *For any node/level (x, l) in S and any stage t, each one of the sets L_t, M_t and R_t has at most one packet. Moreover, it also holds $\mathrm{L}_t < \mathrm{D}_t < \mathrm{M}_t < \mathrm{F}_t < \mathrm{R}_t$.*

Proof: *We prove the lemma by induction on the number of stages t.*

Inductive basis ($t = 0$): At the end of stage $t = 0$, we only have a packet p pointing to $(\mathbf{header}(S), \mathbf{level}(S))$. Furthermore, $\mathrm{remains}_0(p) = \mathrm{false}$ and $\mathrm{split\_side}_0(p) = \mathrm{left}$.

The lemma trivially holds since $\mathrm{L}_0(\mathbf{header}(S), \mathbf{level}(S)) = \{p\}$ *and all other sets are empty.*

Inductive step ($t \Rightarrow t+1$)*: Assume* $\mathrm{M}_t \neq \emptyset$*. Then, by induction hipothesis,* M_t *has exactly only one packet* p_m*. Lemma 4.3 guarantees that at most two packets, say* pp_l *and* pp_r*, come from* $\mathbf{pred}(x,l)$ *during stage* $t+1$*. As they have different* split_side *we can assume* $\text{split\_side}(pp_l) = \text{left}$ *and* $\text{split\_side}(pp_r) = \text{right}$*. When* $\mathrm{L}_t = \{p_l\}$*, the induction hypotesis and Lemma 4.2 imply* $p_l \in \mathrm{D}_{t+1}$*. In that same stage* $t+1$ *the packet* pp_l *comes from* $pred(x,l)$*. Since* $\text{split\_side}(pp_l) = \text{left}$ *and* p_l *has been pushed down, we have* $\mathrm{L}_{t+1} = \{pp_l\}$*. Note that* $\mathrm{L}_{t+1} = \emptyset$ *if no packet is pushed from* $\mathbf{pred}(x,l)$ *with* $\text{split\_side} = \text{left}$*, no matter there existed a packet* p_l *in* L_t *or not. By a similar argument, if* $\mathrm{R}_t = \{p_r\}$ *then* $p_r \in \mathrm{F}_{t+1}$ *and* $\mathrm{R}_{t+1} = \{pp_r\}$*. In the case that such* pp_r *does not exist, then* $\mathrm{R}_{t+1} = \emptyset$*. The packet* p_m *that belongs to* M_t *will be pushed or split in stage* $t+1$ *into two packets and only one of them would remain at* (x,l)*. Therefore,* $\mathrm{M}_{t+1} = \emptyset$ *if* p_m *moves to some other node/level and* $\mathrm{M}_{t+1} = \{p'_m\} \neq \emptyset$ *when* p_m *splits, where* $p'_m \subseteq p_m$ *is the subpacket that remains at* (x,l)*. Therefore, when* $\mathrm{M}_t \neq \emptyset$ *the node/level* (x,l) *can contain at most the packets* pp_l*,* p'_m *and* pp_r *at the end of stage* $t+1$ *and consequently, each of* L_{t+1}*,* M_{t+1} *and* R_{t+1} *has at most one packet.*

We now analyze the relation between the packets in P_t *and the packets* pp_l *and* pp_r *that move from* $pred(x,l)$ *to* (x,l) *in stage* $t+1$*. For each one of* pp_l *and* pp_r*, there are two possibilities: either it is generated during stage* $t+1$ *at* $\mathbf{pred}(x,l)$ *or it is not. If* pp_l *is not generated by an split in stage* $t+1$ *and since we assume that* $\text{split\_side}(pp_l) = \text{left}$ *it follows that* $pp_l \in \mathrm{L}_t(pred(x,l))$*. The induction hypotesis, now applied to* $pred(x,l)$*, yields* $\{pp_l\} < \mathrm{D}_t(pred(x,l)) \cup \mathrm{F}_t(pred(x,l))$*. If* $\mathbf{pred}(x,l) = (x, l+1)$ *then* pp_l *has to be pushed down and* $\mathrm{P}_t(x,l) = \mathrm{D}_t(pred(x,l))$*; otherwise,* pp_l *has to be pushed forward and* $\mathrm{P}_t(x,l) = \mathrm{F}_t(pred(x,l))$*. In both cases, it holds* $\{pp_l\} < \mathrm{P}_t(x,l)$*. If* pp_l *is generated by the splitting of a packet* q *during stage* $t+1$ *it cannot be pushed down, i.e.* $pred(x,l) \neq (x, l+1)$*, because it would contradict Lemma 4.1 otherwise. Hence,* $\mathbf{forward}(pred(x,l), l) = (x,l)$*. Since* $\emptyset \neq \mathrm{M}_t(x,l) \subseteq \mathrm{F}_t(pred(x,l)) < \mathrm{R}_t(pred(x,l))$*, Lemma 4.2 implies* $q \in \mathrm{L}_t(pred(x,l)) \cup \mathrm{M}_t(pred(x,l))$*. Applying once again the induction hypothesis, we have*

$$pp_l \subseteq q \in \mathrm{L}_t(pred(x,l)) \cup \mathrm{M}_t(pred(x,l)) < \mathrm{F}_t(pred(x,l)) = \mathrm{P}_t(x,l),$$

so $\{pp_l\} < \mathrm{P}_t(x,l)$ *is also true if* pp_l *is generated by an split in stage* $t+1$*. A similar analysis yields* $\mathrm{P}_t(x,l) < \{pp_r\}$*, whether* pp_r *is generated in stage* $t+1$ *or not.*

Let us consider the relationship between $\mathrm{L}_{t+1}(x,l)$ *and* $\mathrm{D}_{t+1}(x,l)$*. This last set contains: the packets (or subpackets of these) already in* D_t*, the packet in* L_t *if there were one and, in some cases, the packet or a subpacket of the packet in* M_t*. Since all packets in* D_{t+1} *were also in* P_t *or are subpackets of packets in* P_t *and* $\{pp_l\} < \mathrm{P}_t$*, it follows that* $\mathrm{L}_{t+1} < \mathrm{D}_{t+1}$*. Moreover, the inductive hypothesis implies* $\mathrm{D}_t < \mathrm{M}_t$*, and it is easy to show that* $\mathrm{D}_{t+1} < \mathrm{M}_{t+1}$*, irrespective of the fact that* p_m *moves or splits. Of course, the relationships above trivially hold unless the sets are nonempty. It is not difficult to show, using similar arguments, that* $\mathrm{M}_{t+1} < \mathrm{F}_{t+1} < \mathrm{R}_{t+1}$ *since* $\mathrm{P}_t < \{pp_r\}$*. Hence, if* $\mathrm{M}_t(x,l) \neq \emptyset$

$$\mathrm{L}_{t+1} < \mathrm{D}_{t+1} < \mathrm{M}_{t+1} < \mathrm{F}_{t+1} < \mathrm{R}_{t+1}.$$

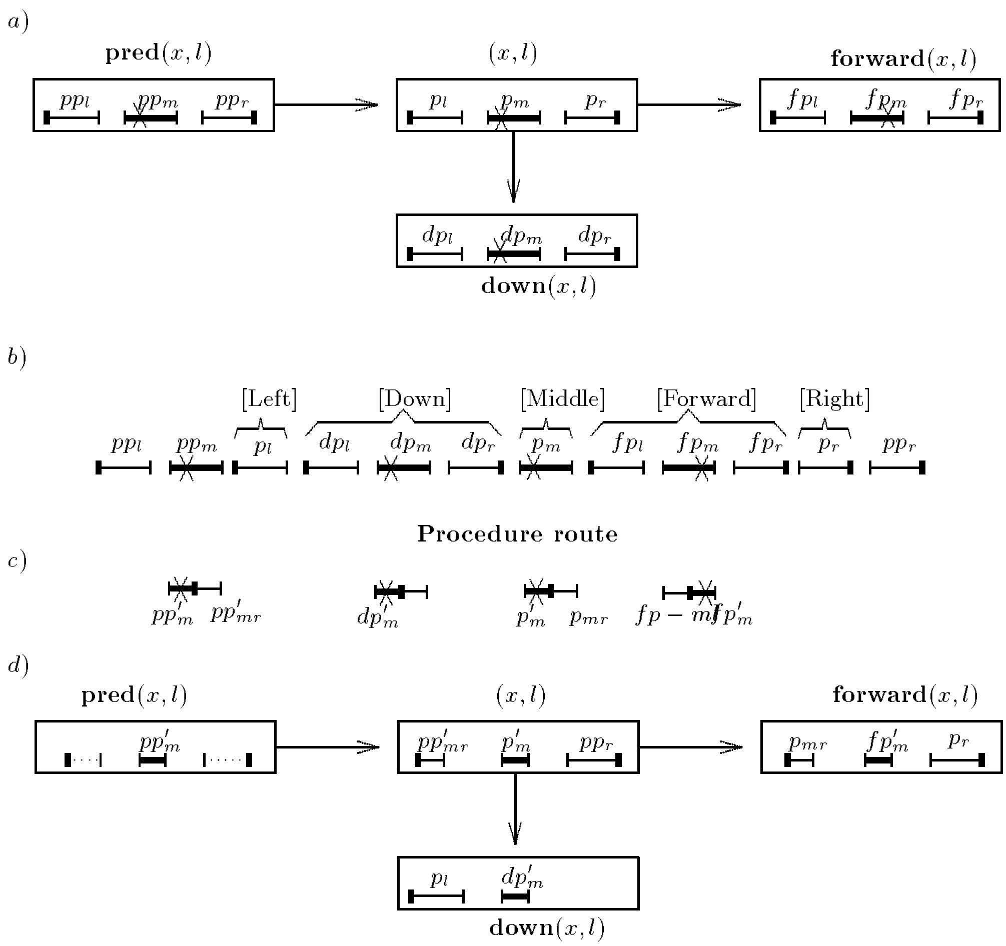


Figure 4.2: Evolution of packets

The lemma holds also true when $\mathrm{M}_t(x,l) = \emptyset$. *The proof in this case is rather similar to the one yet shown and independently considers the cases* $\mathrm{D}_t \cup \mathrm{F}_t \neq \emptyset$ *and* $\mathrm{D}_t \cup \mathrm{F}_t = \emptyset$. ∎

The figure 4.2 shows the changes of states in the particular case that all the node/levels has tree packets and l =level(x). The symbol $\times$ marks the place where the key hits the packet. In diagram a) we see the state at the beginning of the routing step $t+1$. We use the following conventions: a packet is represented by a segment limited by its first and last keys, the split_side by a wide vertical line and the property remains by a wide horizontal line; finally, if a packet p is drawed at left of a packet q it means that last(p)<first(q). The second diagram b) shows the relative position of the subpackets, defined by the hypothesis of the lemma, before the routing process starts. The diagram c) allows us to understand the relative position of the packets split, the ones represented by a superscript *prime*; for instance, the packets p_1 and p_2 created by the split of p_m are represented by p'_m and p'_{mr}. The diagram d) shows the final state of the packets.

If (x,l) is not a top node/level then the $\mathbf{pred}(x,l) = (x, l+1)$. By considering the same figure with the same notation, the routing algorithm will be send the packet pp_r to the forward and pp_l to (x,l). The packet pp_m will be split and depending on the forward

```
Procedure exclusive_read(out b : TKey; in p : packet)
  if remains(p) ⟶b := key(forward(node(p), level(p)) fi ;
  if ¬ remains(p) ∧ split_side(p)= left⟶b := key(forward(node(p), level(p)) fi ;
  if ¬ remains(p) ∧ split_side(p)= right⟶b := key(forward(node(p), level(p)) fi ;
end exclusive_read
```

Algorithm 4.3: Exclusive read of keys

key of $(x, l+1)$ one of them will be forwarded or will be sent down. Is easy to prove that the final state accomplishes the hipothesis of lemma.

From Lemma 4.4 it is plain to see that if we use the procedure *exclusive_read* given in Algorithm 4.3 to access the forward pointers and the keys of the nodes they point to, each routing step can be done in constant time avoiding read conflicts.

Lemma 4.5 *The procedure* route(p, P) *can be implemented in the EREW model in constant parallel time using* k *processors.*

4.1.1.3 Efficiency of the algorithm

We analyze in this subsection the expected number of routing steps or stages in a search. The performance of the parallel search is stated in Theorem 4.1, at the end of the section. It follows directly from the results of the previous subsection and the results in this one.

Before we consider the expected number of stages, we should note that we check for active packets before the execution of each stage or routing step: (**do** active$(P) \neq \emptyset$). If there is at least one active packet, each processor locally "decides", in constant time, whether its associated packet has to be routed or not (see Algorithm 4.4.a):

for all $p : p \in \text{active}(P)$ **do in parallel** ...

Note that the status of a packet (active/inactive) is not the status of the processor associated with that packet. When we check if there is any active packet, all the processors perform a parallel **or** with the active status of all the packets. Since this check is made before the start of each new stage, each stage would have cost $O(\log k)$ and the whole search would have expected cost

$$O(\log k \cdot \text{expected number of stages}).$$

To avoid this $O(\log k)$ cost per stage we execute *runs* of $\lceil L(n) \rceil + \lceil \log_2 k \rceil$ stages, where $L(n) = \log_{1/p} n$. After each run we test whether there remains any active packet or not. If there is at least an active packet in P, then a new run of $\lceil L(n) \rceil + \lceil \log_2 k \rceil$ stages is executed, etc. (see Algorithm 4.4.b). We prove in Lemma 4.6 that the expected number of stages is $O(\log n + \log k)$; hence, the expected number of runs is constant and the expected performance is $O(\log n + \log k)$. Note that we assume that stages can be executed even if there are not active packets at all: the host computer issues each of the instructions in the *route* procedure to each processor, but they execute **skip**s or NOPs instead.

For any two random variables X and Y, we say that Y is a stochastic upper bound for X if and only if, for any t, $\Pr\{X > t\} \leq \Pr\{Y > t\}$. We write $X \leq_{\text{prob}} Y$ (see [47]). Note that $X \leq_{\text{prob}} Y$ implies $\text{E}(X) \leq \text{E}(Y)$. We use $B(n, p)$ to denote a random variable *a*) Main loop of Algorithm 4.1.

```
...
do active(P) ≠ ∅
   for all p : p ∈ active(P) do in parallel
      route(p, P)
   end
enddots
```

b) Main loop of the search by runs.

```
...
do active(P) ≠ ∅
   for j : 1 . . ⌈L(n)⌉ + ⌈log_2 k⌉ do
      ∀p : p ∈ active(P) do in parallel
         route(p, P)
      end
   end
end
...
```

Algorithm 4.4: Main loops of search algorithm

with *binomial distribution*. It is equal to the number of successes seen in a series of n independent random trials, where the probability of success in a trial is p. We denote a random variable with *negative binomial distribution* as $NB(r, p)$. It is equal to the number of failures seen before the r–th success in a series of random independent trials, where the probability of success in a trial is p. If $r = 1$, the random variable is said to be geometric. Finally, to avoid the use of ceiling function we assume that $\log_2 k$ is an integer number.

Lemma 4.6 *Let $C_{n,k}$ be the random variable denoting the number of stages needed to search for k keys in a random skip list of size n. The expected value of $C_{n,k}$ is $O(\log n + \log k)$.*

Proof: *Let C_i be the random variable whose value is the the number of* push_forward, push_down *and* split *operations where key $a[i]$ gets involved before it reaches its final destination in the skip list of n items. Clearly, $C_{n,k} = \max_{1 \leq i \leq k}\{C_i\}$. The number of* push_down *operations applied to the packets containing a given key $a[i]$ is bounded by* **level**(S)*, the height of the skip list. The expected height $\mathrm{E}(H_n)$ for random skip lists of size n is $\Theta(\log n)$ (see [40]). Similarly, the number of* push_forward *operations applied to packets containing a given key is bounded by the* width *of the skip list. The width of a skip list S is the maximum number of forward pointers followed on any search path. Let W_n denote the random variable corresponding to the width of a random skip list of size n. Devroye [12] has shown that $\mathrm{E}(W_n) = \Theta(\log n)$. Finally, the number of split operations where a key gets involved never exceeds $\log_2 k$, since each split halves the size of the packet containing that key. Therefore,*

$$\begin{aligned} C_{n,k} &\leq_{\text{prob}} H_n + W_n + \log_2 k, \\ \mathrm{E}(C_{n,k}) &\leq \mathrm{E}(H_n) + \mathrm{E}(W_n) + \log_2 k = O(\log n + \log k). \end{aligned}$$

■

Our next lemma states that significant deviations between the expected number of stages and the actual number of stages are unlikely to occur. The lemma is proved using

Chernoff bounds. Similar statements have been proved for the time complexity of the sequential search algorithm in skip lists and for the storage requirements of skip lists [51]. Let us recall that, using Chernoff tail bound lemma [11], for a binomial random variable $X = B(n,p)$, any value a $(0 \le a \le n)$ and any $t > 0$

$$\Pr\{X \ge a\} \le \left(q + pe^t\right)^n e^{-ta}, \qquad q \equiv 1 - p. \tag{4.1}$$

For a negative binomial random variable $Y = NB(r,p)$, we have the following bound for any $a \ge 1$ and $t > 0$

$$\Pr\{Y \ge a\} \le e^{-ta} \left(\frac{pe^t}{1 - qe^t}\right)^r, \qquad q \equiv 1 - p. \tag{4.2}$$

Lemma 4.7 *Let $C_{n,k}$ be the number of stages in a search for k keys in a random skip list of size n. Then, for any $\theta > c$ and sufficiently large n, there exists $\gamma = \gamma(\theta) > 0$ such that*

$$\Pr\{C_{n,k} \ge \theta\, L(n) + \log_2 k\} = \Theta(n^{-\gamma}),$$

where $c = c(p)$ is the unique solution, larger than $1/p$, of the equation

$$\log(1-p) = \log(c-1) - \frac{c}{c-1}\log c.$$

Proof: *This proof essentially follows the argument given by Devroye in [12]. Let D_k denote the length of the search path from the top most level of the header to the kth item in a random skip list. Then $\max_{1 \le k \le n} D_k = H_n + W_n$. It turns out that each of the D_k is stochastically smaller than D_n and hence, for any z*

$$\Pr\{C_{n,k} \ge z + \log_2 k\} \le n \Pr\{D_n \ge z\}. \tag{4.3}$$

Let $X_1 = NB(L(n) - 1, p)$, $X_2 = NB(1, 1-p)$ and $X_3 = B(n, 1/np)$. The random variable $L(n) + X_1 + X_2 + X_3$ is an stochastic upper bound for D_n, the path length to the last item in a random skip list of size n [47].

We now define $z := \theta L(n)$, with $\theta > 1$. Applying the Chernoff bounds above (Eqs. (4.1) and (4.2)) we have

$$\begin{aligned}
\Pr\{X_2 \ge z\} &\le \left(\frac{1-p}{p^{\epsilon/\theta} - p}\right) n^{-\epsilon} = \Theta(n^{-\epsilon}),\\
\Pr\{X_3 \ge z\} &\le \exp\left(p^{-1}(p^{-\epsilon/\theta} - 1)\right) n^{-\epsilon} = \Theta(n^{-\epsilon}),
\end{aligned}$$

where for both inequalities $\epsilon > 0$ is any arbitrary constant. We take $t = (\epsilon/\theta)\ \ln(1/p) > 0$ (however, one should avoid $\epsilon = \theta$ for obvious reasons in the first inequality).

Finally, let $\ell := L(n) - 1$ and $u := \ell/z$. For X_1 we have

$$\begin{aligned}
\Pr\{X_1 \ge z\} &\le e^{-tz}\left(\frac{pe^t}{1 - qe^t}\right)^{\ell} \le p^{\ell} q^{z-\ell} \frac{z^z}{\ell^{\ell}(z-\ell)^{z-\ell}}\\
&= \left\{\left(\frac{q}{1-u}\right)^{1-u} \left(\frac{p}{u}\right)^{u}\right\}^{z}.
\end{aligned}$$

where we take $e^t = (z - \ell)/qz$ (and hence θ must be larger than $1/p$ to ensure that $t > 0$).

The lemma follows from Eq. 4.3 and the bounds for the tails of X_1, X_2 and X_3:

$$\begin{aligned} n\Pr\{D_n \geq \theta\, L(n)\} &\leq n\Pr\{X_1 + X_2 + X_3 + L(n) \geq \theta\, L(n)\} \\ &\leq \Theta(n^{1-\epsilon}) + \exp\left(L(n)\left(\ln(1/p) + \theta\left((1-u)\ln\frac{q}{1-u} + u\ln\frac{p}{u}\right)\right)\right) \\ &= \Theta(n^{-\gamma}), \end{aligned}$$

where $-\gamma$ is the coefficient of $L(n)$ and we make ϵ arbitrarily large. To complete the proof, recall that for n large enough $u \to 1/\theta$ and therefore, the constant γ is positive if θ is larger than the unique root $c > 1/p$ of the equation:

$$(c-1)\ln(1-p) - (c-1)\ln\frac{c-1}{c} + \ln c = 0.$$

■

From Lemmas 4.4, 4.6 and 4.7, and the fact that the current node of each packet at the end of the main loop can be optimally spread (segmented copy) in time $O(\log k)$ using $k/\log k$ processors [8, 49], we have the next Theorem:

Theorem 4.1 *The parallel search in a skip list can be implemented in a EREW PRAM model with expected time $O(\log n + \log k)$ and using k processors, where k is the number of keys to be searched for and n is the size of the skip list. Moreover, the probability that its performance significantly deviates from its expected value, say, it is θ times larger decreases as $O(n^{-\gamma})$, for some $\gamma > 0$ that depends on p and θ.*

4.1.2 Massively parallel insertions

4.1.2.1 Algorithm

Assume, for the sake of simplicity and without loss of generality, that we want to insert k distinct items not already present in the skip list. The insertion algorithm has four main phases (see Algorithm 4.5). First, the procedure *search_with_update* searches for the k keys and builds a collection of *update* arrays with information about the would-be predecessors of the new items. Second, *create_new_nodes* allocates k new nodes to hold the items to be inserted. Third, *make_succ* builds an *skip array* called *succ*. This array, together with the *update* arrays, provides information about the predecessor and successor of each node/level to be inserted. Fourth, the procedure *merge* actually inserts the k new nodes in the appropriate places of the skip list.

Procedure search_with_update: Upon termination a set of *update* arrays, one for each new item, has been computed. The *update* array for the i-th new item has information about its would-be predecessors for each level (See Figure 4.3). We have

$$\begin{aligned} \mathit{update}[l,i] \quad = \quad & \text{"the unique node } x \text{ in } S \text{ such that} \\ & \mathbf{key}(x) < a[i] \leq \mathbf{key}(\mathbf{forward}(x,l))\text{"},\ 1 \leq l \leq \mathbf{level}(S),\ 1 \leq i \leq k. \end{aligned}$$

Note that $\mathit{update}[l,i]$ will be the predecessor of the i-th new item at level l only if a level larger or equal to l is assigned to the new item and there is not any key to be inserted, say $a[j]$, such that $\mathbf{key}(\mathit{update}[l,i]) < a[j] < a[i]$. To compute these *update* arrays we maintain an *update* array for each packet p. Let p be a packet that has

```
Procedure insert (in a : AKey; in/out S :skip_list)

  typeskip_array = array[1..??, 0..k + 1]of0..k + 1;
    {the number of rows is computed at run-time}
      update_array = array[1..??, 1..k] ofItem;
      node_array = array[1..k] ofItem;
      level_array = array[0..k + 1] ofinteger;

  Procedure search_with_update( ... );
  Procedure create_new_nodes( ... );
  Procedure make_succ( in level : level_array;
  Procedure merge( ... );

  var succ : skip_array;
      update : update_array;
      node : node_array;
      level : level_array;
      m :integer;

  search_with_update (a, S, update);
  create_new_nodes (a, node, level);
  m:= max(level[i] : 1 ≤ i ≤ k);
  make_succ (level, m, succ);
  merge (succ, update, node, m, S)
```

Algorithm 4.5: Insertion algorithm

been generated at node/level (x, l). We keep track of the path followed by p in its $update(p)$ array as follows

$$\begin{aligned} update(p)[k] &= \text{rightmost node of level} \geq k \text{ in the search path from} \\ &\quad (x, l) \text{ to the current node/level } (x', l') \text{ of } p, \quad l' < k \leq l. \end{aligned}$$

In fact, the *update* array of a packet p is the one associated to its first key. To collect the information in the *update* arrays, the procedure push_down is modified by adding

$$update[\text{level}(p), \text{first}(p)] := \mathbf{node}(p);$$

just before we decrement the current level of p.

When the search loop finishes we must spread the *update* arrays of the packets to all the keys. If the key $a[i]$ was in a packet p the last time that key was at level l, then the l-th component of the update array of p should be copied to $update[l, i]$. This is not very difficult since the needed value is held, by construction, by $update[l, j]$, where j is the maximum index such that $j \leq i$ and $update[l, j] \neq \mathbf{NIL}$.

Procedure create_new_nodes: To create the new nodes for the new k items, k random levels are independently generated in the first step. This is achieved by making each processor call, in parallel, a function *random_level*. The value it returns follows a geometric $NB(1, p)$ distribution. The random level of the i-th new item is stored in $level[i]$. Finally, the procedure allocates k new nodes with as many forward pointers per node as its level indicates and stores each of the k keys in its corresponding node. It returns an array *node* of pointers to these new nodes, as well as the array *level*.

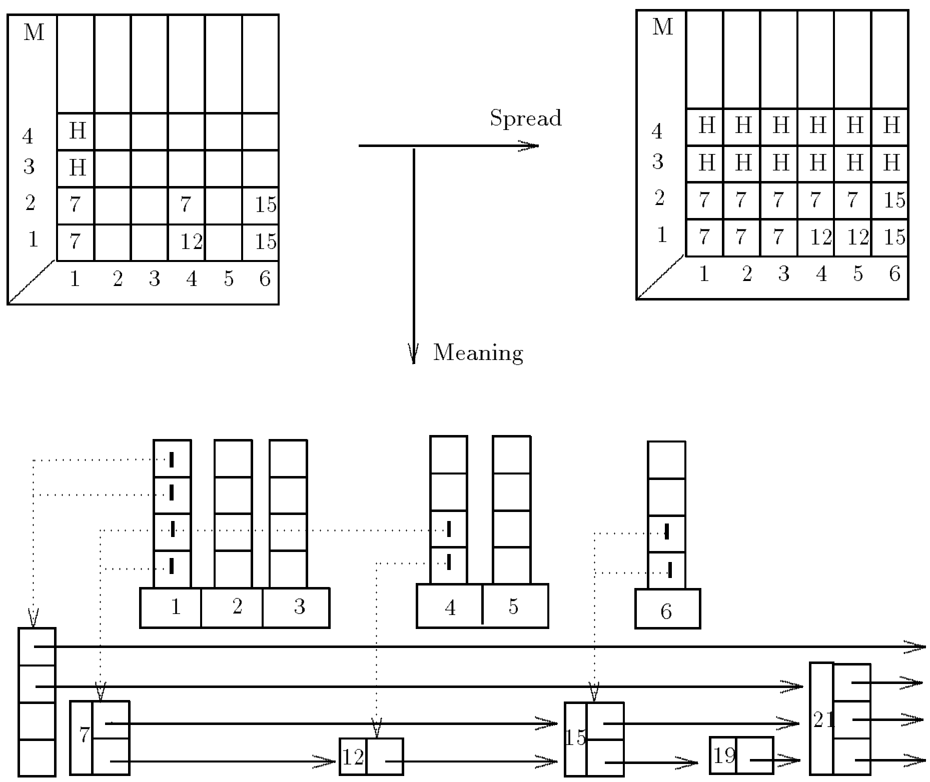


Figure 4.3: Meaning of the *update* array

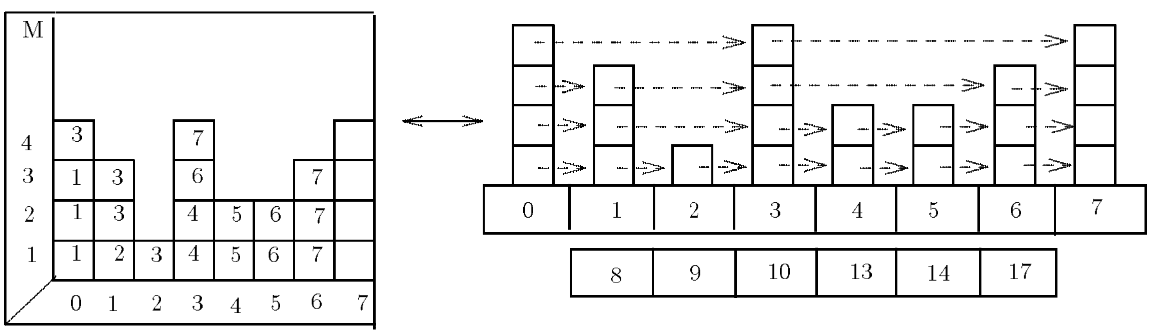


Figure 4.4: Meaning of the *succ* array

```
Procedure make_succ( in level : level_array; in m : integer; out succ : skip_array)
  var i : 0..k; l : integer;

  level[0] := m;
  level[k + 1] := m;
  for all l : 1 ≤ l ≤ m do in parallel
    for all i : 0 ≤ i ≤ k ∧ level[i] ≥ l do in parallel
      succ[l, i] := min(j : i < j ≤ k + 1 ∧ level[j] ≥ l)
    end
  end
```

Algorithm 4.6: Make_succ algorithm

Procedure make_succ: (see Algorithm 4.6). It builds the skip array *succ*. This array mimics a new skip list correspoding to the keys in a alone (see Figure 4.4). An skip array is a two-dimensional array with $m = \max\{level[i] : 1 \leq i \leq k\}$ rows and $k+2$ columns. Any column i with $1 \leq i \leq k$ has $level[i]$ useful rows. Columns 0 and $k+1$ are sentinels with m rows. The array *succ*, for $0 \leq i \leq k$, is defined as follows:

$$succ[l, i] = \text{minimum } j \text{ to the right of } i,\ i < j \leq k+1, \text{ such that } level[j] \geq l.$$

Recall that we must be able to decide whether $update[l, i]$ is the actual predecessor of the node $node[i]$ at level l or not. The skip array *succ* brings exactly this information (see the description of the procedure merge). Note that $succ[l, 0]$ is the index of the first new node whose level is at least l. All the rows of *succ* can be computed independently and can be filled using easy variants of the tree or parallel prefix computations [8, 49].

Procedure merge: This phase merges *succ* with S (see Algorithm 4.7). If $m > \textbf{level}(S)$, we fill with pointers to **header**(S) the components **level**(S) up to m of the arrays *update*. Later, we insert the new nodes level by level in S using the procedure merge_level.

Let us consider it in more detail. Consider a level l and a node $node[i]$ with corresponding key $a[i]$ and $level[i] \geq l$. To insert $node[i]$ in the linked list of level l we need to know both its predecessor and its successor for that level. There are four different cases (see Algorithm 4.7 and Figure 4.5). We consider only two of them, the others are similar. Given i with $1 \leq i \leq k$, call $succ[l, i] = j$ and suppose that

$$j \neq k+1 \wedge update[l, i] \neq update[l, j].$$

As j is different from the sentinel $k+1$ there exists a node $node[j]$ such that its level is at least l and $a[j] > a[i]$. Moreover, as $update[l, i] \neq update[l, j]$ there exist nodes in S at level l whose keys lie between $a[i]$ and $a[j]$. Hence, $node[j]$ is not the successor of $node[i]$ at level l in the new skip list and $update[l, j]$ is the actual predecessor of $node[j]$. The successor of $node[i]$ at level l is the node that was the successor of $update[l, i]$ at level l. The following assignments are thus needed:

$$\text{forward}(node[i], l) := \text{forward}(update[l, i], l); \quad \text{forward}(update[l, j], l) := node[j];$$

Now assume that $j \neq k+1$, but $update[l, i]$ and $update[l, j]$ coincide. Then we can be sure that $update[l, j]$ will not be the actual predecessor of $node[j]$; $node[i]$ will be

```
Procedure merge (in succ : skip_array; in update : update_array;
                 in node : node_array; in m : integer; in/out S :skip_list)

  Procedure merge_level (in succ : skip_array; in update : update_array;
                         in node : node_array; in l : integer)
    ivar i : 1..k; j : 0..k;

    for all i : 1 ≤ i ≤ k ∧ level[i] ≥ l do in parallel
      j := succ[l, i];
      if
        (3) j ≠ k + 1 ∧ update[l, i] ≠ update[l, j]⟶ forward(node[i], l) := forward(update[l, i], l);
                                                    forward(update[j, l], l) := node[j]
        (2) j ≠ k + 1 ∧ update[l, i] = update[l, j]⟶forward(node[i], l) := node[j]
        (1) j = k + 1⟶forward(node[i], l) := forward(update[l, i], l)
      fi
    end;
    j := succ[0, l];
    (4) forward(update[j, l], l) := node[j]
  end merge_level

  var l :integer; i : 1..k;

  for all i : 1 ≤ i ≤ kdo in parallel
    for l : level(S) ≤ l ≤ m do
      update[l,i]:= header(S)
    end
  end;for l : 1 ≤ l ≤ m do
    merge_level (succ, update, node, l)
  end;
  level(S):= max(level(S), m)
end merge
```

Algorithm 4.7: Merge algorithm

the predecessor of $node[j]$ at level l. The procedure merge_level performs in this case the assignment:

$$\text{forward}(node[i], l) := node[j];$$

All the tests and pointer updates can be done in parallel constant time, since only the local information provided by the *update* and *succ* arrays is needed.

4.1.2.2 Efficiency of the algorithm

The search for the places to insert the k new items needs parallel expected time $O(\log n + \log k)$ using k processors (Theorem 4.1). Gathering the information in the *update* arrays during the search does only introduce an additional constant cost for each routing step. The initialization of the *update* arrays components to **NIL** can be done by k processors in expected time $O(\mathbf{level}(S)) = O(\log n)$. The information collected in these *update* arrays must be spread to all the processors when the main loop of the search finishes; now one processor is associated to every key. Each level of the update arrays can be optimally spread using $O(\log k)$ time and $O(k/\log k)$ processors in an EREW PRAM. We can allocate $k/\mathbf{level}(S)$ processors for each level in the *update* arrays, and do the broadcasting of the *update* arrays in time $O(\mathbf{level}(S) + \log(k/\mathbf{level}(S)))$ using k processors. Since the expected level of S is $O(\log n)$ the expected time to spread the update arrays is $O(\log n + \log k)$.

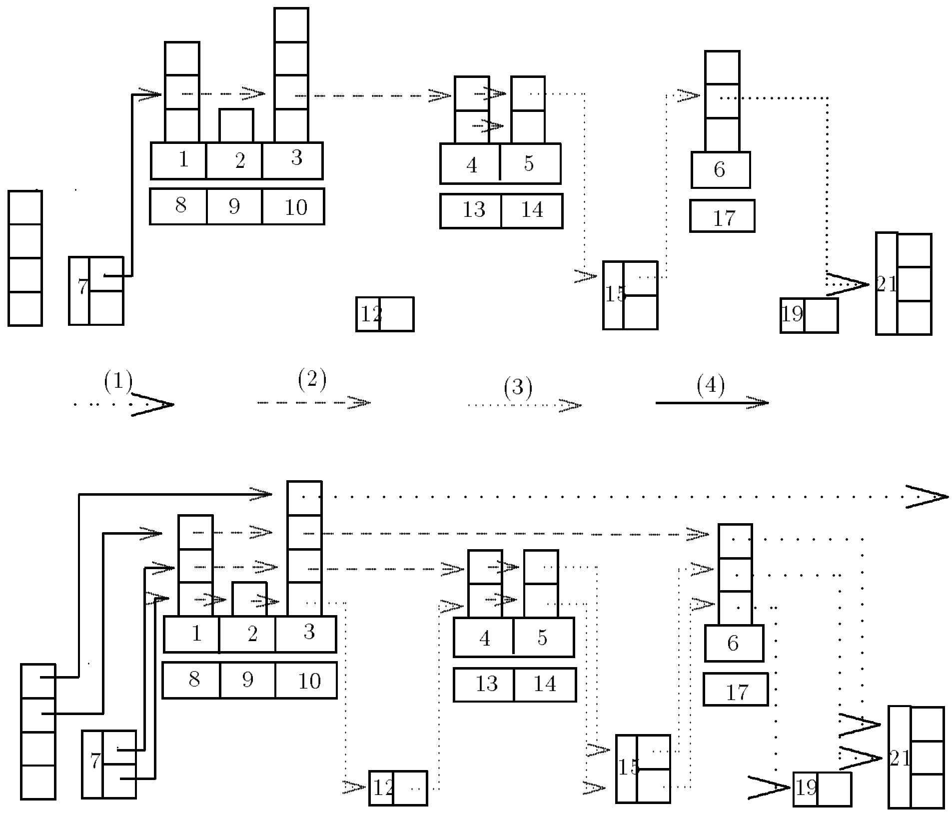


Figure 4.5: Meaning of the merge_level algorithm

The next step in the insertion procedure requires the computation of a random level for each new item to be inserted. The expected time to generate the k random levels is proportional to the expected height of a skip list of size k. Therefore, after an expected number $O(\log k)$ of iterations, the level of each new item has been independently computed. Later, k new nodes must be allocated to store the new items. The parallel dynamic memory manager needs $O(\log k)$ steps to allocate the k new nodes [23, 42, 43].

To create the array *succ* with the information about possible successors of the new items, we need first to compute the maximum level m among the levels of the new items. This can be obviously done in $O(\log k)$ steps using k processors. Using k/m processors for each row, it takes $O(m + \log(k/m))$ parallel time to fill all the rows of *succ*, using a slight variation of common parallel prefix computations. Since the expected value of m is $O(\log k)$, this part of the insertion has expected cost $O(\log k)$ with k processors. In order to avoid concurrent reads in the *level* array, the make_succ procedure should fill a bidimensional array of bits, say lb, such that $lb[l,i] = 1$ if and only if $level[i] \geq l$. The array lb can be filled by k processors in expected time $O(\log k)$ (the expected number of rows in lb) in an EREW model. Then, the computation of each row of the *succ* array can be done using the corresponding row of the lb array and concurrent read conflicts are avoided.

Finally, a call to the procedure merge is performed. The execution of this procedure with k processors takes parallel expected time $O(\log k)$, since merge_level has cost $O(1)$ using k processors and merge calls merge_level m times, once for each level. It is not difficult to see that the procedure merge_level can be written without using concurrent reads. Taking all these contributions into account Theorem 4.2 follows. Moreover, we can prove that there is a very small probability of bad performance for the insertion algorithm using Chernoff bounds, following similar arguments to those used in Subsection 4.1.1.3. In particular, an $O(n^{-\gamma})$ bound follows from the search phase. The phase were new nodes are created needs $O(\log k)$ in the worst-case. Both the construction of the skip array and the merge phases take $O(\log k)$ expected time. In both cases the performance is proportional to the height of a random skip list of k elements. Then, it is not difficult to prove, using Chernoff bounds as in Subsection 4.1.1.3, that the probability of large deviations from the expected time is $O(k^{-2})$. Furthermore, the cost of the search phase is independent of that of the next phases.

Theorem 4.2 *The insertion algorithm for skip lists can be implemented in a EREW PRAM model with expected time $O(\log n + \log k)$ using k processors, where k is the number of keys to be inserted and n is the length of the skip list. Moreover, the probability that the performance of the insertion algorithm deviates from its expected value decreases as $O(n^{-\gamma} + k^{-\gamma})$.*

4.1.3 Massively parallel deletions

We now consider the deletion algorithm. First, it uses the *search_with_update* procedure to find where are the keys to be deleted and its predecessors at each level. After this step, it constructs three skip arrays: *succ*, *pred* and *last*, giving information about sucessors, predecessors and blocks of consecutive nodes to be deleted. For each node to be removed and each of its levels, we must know whether its predecessor and successor will remain or be also removed. Which one of these cases holds can be checked in parallel using the *update* and the three mentioned skip arrays. Each case can be managed with simple parallel address arithmetic techniques similar to those used in the insertion algorithm.

Both the procedures search_with_update and make_succ are the ones that we already described for the insertion. The number of rows in the *succ* array is computed from the actual levels of the items to be deleted. We now give a short description of the other procedures used in the deletion algorithm (see Algorithm 4.8 and Figure 4.6).

Procedure make_pred: This procedure computes an skip array called *pred*, that provides information on the predecessor of the items to be deleted, given their levels.

$$pred[l, i] = \text{maximum } j \text{ to the left of } i,\ 0 \leq j < i, \text{ such that } level[j] \geq l.$$

The procedure is quite similar to the procedure make_succ.

Procedure make_last: Consider a node $node[i]$ to be deleted and a level $l \leq level[i]$. Maybe a block of consecutive node/levels in the list of level l and starting from $(node[i], l)$ will be deleted too. In order to chain the nodes that will not be deleted, we need to know the index of the rightmost element in this block. We call it $last[l, i]$. Therefore, the nodes

$$node[i], \mathbf{forward}(node[i], l), \mathbf{forward}(\mathbf{forward}(node[i], l), l), \ldots, node[last[l, i]]$$

will be deleted, but $\mathbf{forward}(node[last[l, i]], l)$ should remain in S. Formally,

$$\begin{aligned} last[l, i] \quad = \quad & \text{minimum index } j,\ i \leq j \leq k, \text{ such that } level[j] \geq l \text{ and } node[j] \\ & \text{will be deleted but } \mathbf{forward}(node[j], l) \text{ will not.} \end{aligned}$$

```
Procedure delete (in a : array[1..k] ofTKey; in/out S : skip_list);

  Procedure search_with_update( ... )
  Procedure make_pred( ... )
  Procedure make_succ( ... )
  Procedure make_last(in level : level_array; in m :integer; in node :node_array;
                      in succ : skip_array; out last : skip_array)
  Procedure remove( ... )
  var pred, succ, last : skip_array;
      update : update_array;
      node : node_array;
      level : level_array;
      m :integer; i :1..k;

  search_with_update (a, S, update);
  for all i : 1 ≤ i ≤ k do
      level[i] := level(node[i])
  iend;
  m := max(level[i] : 1 ≤ i ≤ k);
  make_pred (level, m, pred);
  make_succ (level, m, succ);
  make_last (level, m, node, succ, last);
  remove (pred, last, update, node, m, S)
end delete
```

Algorithm 4.8: Deletion algorithm.

There are two cases in the computation of $last[l,i]$ (see Algorithm 4.9). When $succ[l,i] = k+1$, it means that $node[i]$ is the last whose level is greater or equal to l and will be deleted. In particular, the forward pointer of $node[i]$ at level l points to a node that will not be removed from S. Therefore, $last[i,l] = i$. When $succ[l,i] \neq k+1$ we have that "**forward**$(node[j], l)$ will not be deleted from S" is equivalent to "**forward**$(node[j], l) \neq node[succ[l,j]]$)", and the element $last[l,i]$ is given by

$$last[l,i] = min(j : \ i \leq j \leq k \ \wedge \ level[j] \geq l \ \wedge \ \textbf{forward}(node[j], l) \neq node[succ[l,j]])$$

Procedure remove: It actually removes the elements from the skip list (see Algorithm 4.10). Given a $node[i]$ to be deleted and a level l, its predecessor at level l is $update[l,i]$. There are two possible cases. The first one happens when $update[l,i]$ has also to be removed. This can be easily checked because

$$\text{“}update[l,i] \text{ will be deleted”} \equiv \text{“}pred[l,i] \neq 0 \wedge update[l,i] = node[pred[l,i]]\text{”}$$

As $node[pred[l,i]]$ will also be deleted the pointer **forward**$(node[pred[l,i]], l)$ is redundant and the algorithm does not take care of it. The second one occurs if $update[l,i]$ has not to be removed from S:

$$\text{“}update[l,i] \text{ remains”} \equiv \text{“}pred[l,i] = 0 \vee update[l,i] \neq node[pred[l,i]]\text{”}$$

The node $update[l,i]$ remains in the list and its forward pointer at level l is set to **forward**$(node[last[l,i]], l)$.

Procedure make_last(**in** $level$: level_array; **in** m : integer; **in** $node$:node_array;
  in $succ$:skip_array; **in** $last$:skip_array)
  var $i : 0..k; l$: integer;

  for all $l : 1 \leq l \leq m$ **do in parallel**
    for all $i : 1 \leq i \leq k \wedge level[i] \geq l$ **do in parallel**
      if
      $succ[l,i] = k+1 \longrightarrow last[l,i] := i$
      $succ[l,i] \neq k+1 \longrightarrow last[l,i] :=$ **min**$(j : i \leq j \leq k \wedge level[j] \geq l \wedge$
        forward$(node[j],l) \neq node[succ[l,j]])$
      fi
    end
  end
end make_last

Algorithm 4.9: Make_last algorithm

The analysis of the deletion algorithm goes along the same lines as that of the insertion algorithm. The searching phase needs $O(\log n + \log k)$ expected time with k processors. The computation of the skip arrays *succ*, *pred* and *last* reduces to parallel tree computations with expected cost $O(\log k)$ using k processors. Finally, each call to remove_level consumes constant parallel time using k processors; the procedure is called $O(\log k)$ times on the average. Summing up these contributions the next Theorem follows.

Theorem 4.3 *The deletion algorithm for skip lists can be implemented in a EREW PRAM model with expected time $O(\log n + \log k)$ and using k processors, where k is the number of keys to be deleted and n is the length of the skip list. Moreover, the probability that the performance of the deletion algorithm significantly deviates from its expected value decreases as $O(n^{-\gamma} + k^{-\gamma})$.*

4.2 Implementation issues

We discuss in appendix A.1 several implementation-related issues and give some examples using the programming language `C*`. In this section, a preliminary implementation of the search and insertion algorithms has already been developed for the Connection Machine CM-2 and their performance has been empirically studied [33].

4.2.1 Implementation

The notions of packets and packet routing are useful to explain the search algorithm, but it is not easy to express these notions in most current parallel languages. A simple way to implement Algorithm 4.1 is to define a parallel array of size k, so each processor holds a key, an active status, etc. For instance, in `C*`, we can dynamically declare a shape of size k using `allocate_shape` and then a parallel variable of that shape with `palloc`, where each component of the parallel variable is a record with several fields. The field `key` holds the keys to be searched for. The other fields are: `active`, `first`, `last`, `node`, `level`, `remains` and `split_side`. We can think of each processor containing a *slice* associated

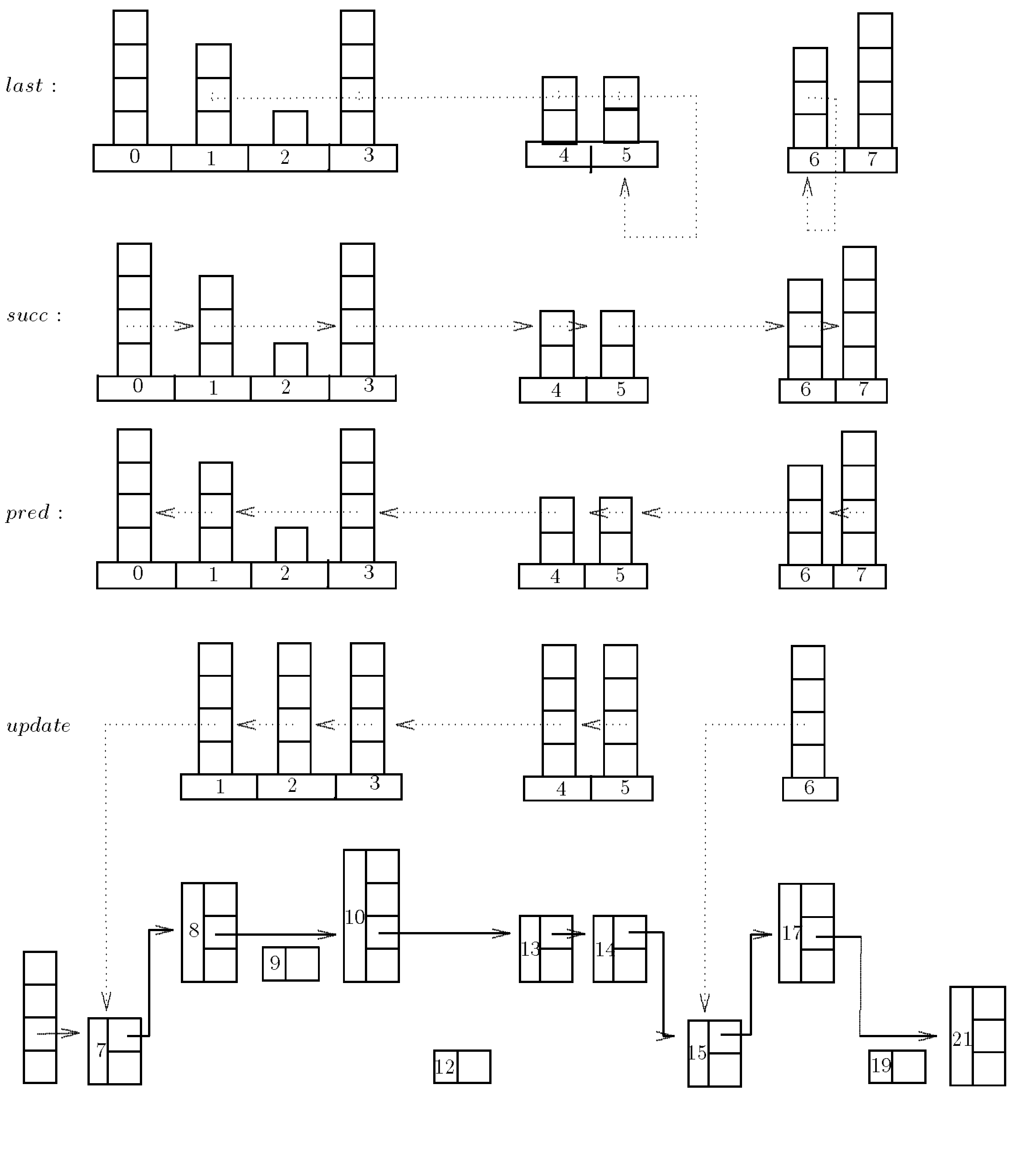


Figure 4.6: Meaning of the remove algorithm

Procedure remove (**in** *pred*, *last* : skip_array; **in** *update* : update_array;
in *node* : node_array; **in** m : integer; **in/out** S : skip_list)

Procedure remove_level (**in** *pred*, *last* : skip_array; **in** *update* : update_array;
in *node* : node_array; **in** l : integer)
var $i : 1..k; j : 0..k;$

for all $i : 1 \leq i \leq k \wedge l \leq level[i]$ **do in parallel**
$j := pred[l, i];$
if
$j = 0 \vee update[l, i] \neq node[j] \longrightarrow$forward$(update[l, i], l) :=$ forward$(node[last[l, i]], l)$
$j \neq 0 \wedge update[l, i] = node[j] \longrightarrow \emptyset$
fi
end
for all $i : 1 \leq i \leq k$ **do in parallel**
$free(node[i])$
end
end remove_level

var l : integer;

for $l : 1 \leq l \leq m$ **do**
remove_level $(pred, last, update, node, l)$
end;
do forward(header(S), level(S)) = NIL$\longrightarrow$
level(S) := level(S) $- 1$
end

end remove

Algorithm 4.10: Remove and remove_level algorithms

with a particular key: the i-th processor contains the i-th key and the i-th value for each of the other fields.

Consider a packet p that holds the keys $a[i], \ldots, a[j]$ (first$(p) = i$ and last$(p) = j$) at some stage of the search algorithm. We use the convention that the proccesor associated to the first key of a packet is "responsible" of the whole packet. This situation is represented by storing first(p) and last(p) in the i-th components of the fields `first` and `last`, and storing the current node/level of the packet in the i-th components of the fields `node` and `level`. If p is an active packet, then the value of the i-th component of the `active` field is 1 (true), whereas the rest of the components of the `active` field that are associated to other keys in the same packet p are set to 0 (false). If p is not active, the `active` field is set to 0 for all keys within p.

We give some useful `C*` declarations for the search procedure:

```
shape []packet_shape; /* number of positions in the shape is left
                         unspecified; it will be specified at run-time */
typedef struct {
        key_type key;
        int first, last, level;
        refnode_type node;
        bool active, remains, split_side; } packet_CM;

typedef packet_CM:packet_shape set_of_packets;
```

We now sketch the C* code for the search procedure. It takes as arguments a pointer to a `set_of_packets` and a skip list. We assume that the `key` field of the `set_of_packets` is already initialized with the k keys. The result is returned through the `node` field.

The first lines of the code create a single active packet located at $(\mathbf{header}(S), \mathbf{level}(S))$. The test for active packets can be accomplished by performing a parallel **or** reduction (`|=`) of all components of the `active` field. Since the parallel **or** reduction can be very efficiently performed in the Connection Machine, we do not execute the main loop of the search by runs. Inside the main loop of the search, only those packets that are still active have to be routed. Therefore, we select only the processors where the `active` field is true. The last step in the search can be efficiently done using *scan sets* and the `scan` function.

The implementation of the procedure *route* should also be easy, although three separate reading phases are needed.

```
void search(set_of_packets *a, int k, skip_list S) {
    int i;
    bool:packet_shape segbit;

      /* initializations */
    a -> first = -1; a -> last = -1;
    [0](a -> first) = 0; [0](a -> last) = k - 1;
    [0](a -> node) = header(S); [0](a -> level) = level(S);
    with (packet_shape)   /* only processor nr.  0 is active */
    a -> active = (pcoord(0) == 0);
    ...
      /* main loop */
    with(packet_shape)
        while(|= a -> active)
             where(a -> active)
                  route(a, S);

      /* spread the current node of each packet to fill 'a -> node' */
    ... }
```

The number of stages in a search can be reduced if, whenever a packet is split, we try to route the two resulting subpackets in the same stage. This is possible if the key that hits the packet does not hit any of the two subpackets. The main lemmas about read conflicts, as well as Theorem 4.1, hold for this variant. Note that in this variant, any packet p such that remains(p) is true after stage t must be split in stage $t+1$. Although this version of the search guarantees at most the same number of stages as the original algorithm, the execution of the main loop by runs (see Subsection 4.1.1.3) and the increased complexity of each routing step makes this choice less interesting that it could seem at a first glance.

We can benefit from the use of C*'s `scan` function to implement the insertion and deletion procedures. For instance, it can be used to spread the information collected in the *update* arrays during the initial search phase. It is also useful to efficiently build the skip arrays *succ*, *pred* and *last*, since `scan` is a kind of generalized parallel prefix facility. The *update* and skip arrays used in the insertion and deletion procedures are bidimensional parallel variables. The number of positions in each rank or dimension can be dynamically declared using `allocate_shape` and then the variables themselves allocated using `palloc`. Note that this simple approach requires $O(k \log n)$ processors (on the average) to store the *update* array and $O(k \log k)$ processors for each skip array. Since the skip list itself must be stored in the local memories of the processors (not in the host) an expected number of $O(n)$ processors is needed, anyway.

In the case of the insertion, it will be useful to generate the levels of the new nodes and compute its maximum level before allocating the *update* array. In general, the *update* needs $O(k \log n)$ space (on the average) to be stored. But since we know in advance the maximum level m among the new items, only $O(k \log k)$ space will be needed for the *update* array. If $m <$ **level**(S) then this strategy should be used. The code of the procedure *search_with_update* gets messier, but it is more space-efficient.

Finally, we discuss a simple way to save both space and expensive key comparisons. We have assumed that every node/level in a skip list stored a copy of the key of the node. That assumption is useful since we have shown that there is at most a constant number of packets at a node/level in any given stage, but it would be interesting to avoid such a space consuming solution. We cannot simply let all packets at the predecessors of a given node access a single copy of the key, because the number of such packets is not bounded by a constant: there can be an expected number of $O(\log n)$ packets trying to look at the single copy of the key of a node during any given stage and, in principle, $O(\log \log n)$ time will be needed to execute a stage in an EREW model.

The trick is to keep a pointer *already_checked*(p) in each packet p. It points to the last node to which the packet "looked at" in the previous current level. The pointer is initialized to **NIL** and updated each time the packet drops one level. The procedure *route* is slightly modified so each routing step begins comparing the pointer *already_checked*(p) and the forward pointer of the current node/level, say **forward**(x, l). If they are equal, the packet is pushed down. Otherwise, the key of **forward**(x, l) is read and the packet is split, pushed forward or down according to the rules given in Section 4.1.1.1. The protocol for exclusive read (see Subsection 4.1.1.2) must be used for both reading the forward pointer and the key of the node it points to. The crucial point is that, if the level of the node **forward**(x, l) is greater than l, then *already_checked*$(p) =$ **forward**(x, l). Hence, we can keep a single copy of the key of each node, since the keys are exclusively read by packets "looking" through forward pointers that access top node/levels. In other words, a comparison between keys is needed only if a packet is at some node/level (x, l) and **level**(**forward**$(x, l)) = l$. Moreover, the key of a node has not to be read more than once by any given packet if we keep a copy the last read key in each packet. Then, if a packet was generated by a split and was not moved during the same step, the copy of the last read key can be used in the next stage.

The idea of *already_checked* pointers was suggested by Pugh in [46] as an optimization to reduce the number of key comparisons in the sequential search algorithm. A detailed performance analysis can be found in [26].

4.2.2 Performance

This section studies the performance of parallel and sequential implementation of Skip lists. The sequential algorithm was codified in `C` and the parallel one in the parallel extension `C*`. The sequential program was executed in a SUN Sparc Station. The parallel one in a CM-200 with 2048 processors and a SUN Sparc Station as a Front End.

Accuracy of the data: The first question is to determine the number of times that the algorithms should be run. To decide it we run one hundred times the insertion algorithm of a 16000 keys into an Skip list of size 16000. The distribution of data follows the Normal Distribution with mean 3.237 and variance 0.0438. Therefore, as we want that the relative error of data was smaller than the 5%, from

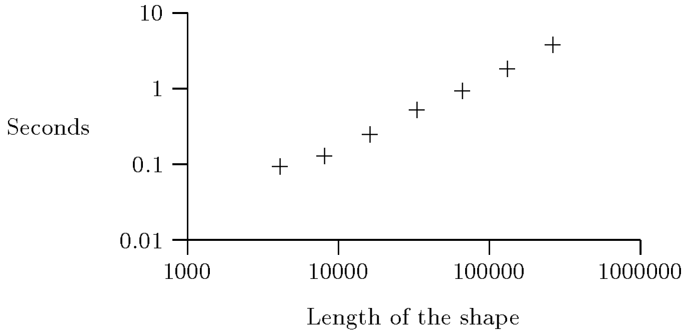


Figure 4.7: Running time with differents shapes

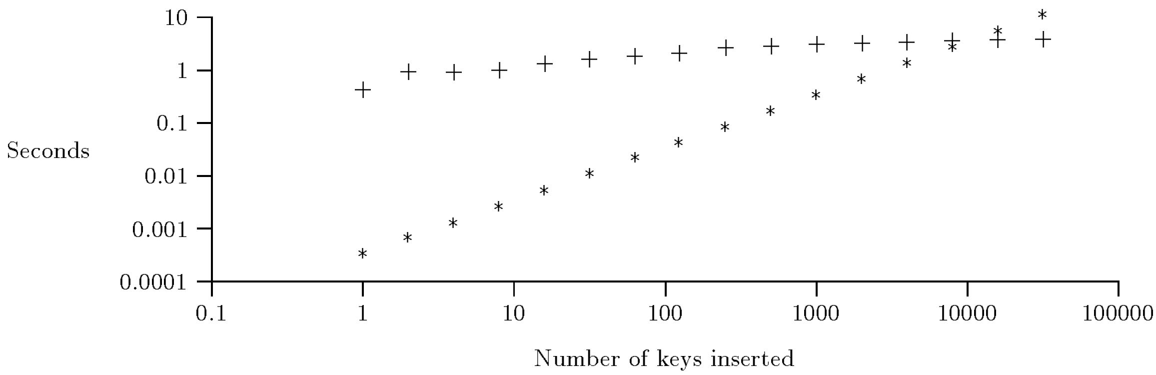


Figure 4.8: Comparation between sequential (*) and parallel (+) running insertion time

$$\text{Prob}\left(\mu \in \left[3.237 - 1.96 \cdot \sqrt{\frac{0.0438}{z}}\ ,\ 3.237 + 1.96 \cdot \sqrt{\frac{0.0.0438}{z}}\right]\right) \geq 95\%$$

where z is the size, we conclude that the sample of the size z can be 25.

Size of shapes: We analyze the relation between the length of the shapes and the running time. Althought the hardware contains small number of physical processors, we can work with many virtual processors,*i.e.* each one can support $8, 16, 32, \cdots$ virtual ones.This fact implies a linear increment of the time spent in the execution (see Figure 4.7), Figure 4.7 represents the time spent by the Connection Machine. The linearity of the representation confirms the proporcionality between the running time and the number of virtual processors.

Performance of sequential and parallel insertions: Figure 4.8 contains the running times of sequential $(*)$ and parallel $(+)$ insertion of k keys into a skip list of size 32000.

Note that both running times coincide when 10000 keys are inserted. This means that the sequential implementation is advisable for the insertion of smaller number of key.

Chapter 5

Skip trees (and 2–3 trees)

Skip trees follow from the attempt to design concurrent algorithms with local rules for Skip lists. After the introduction we devote the second section to Skip trees and the third section to 2–3 trees. Both search trees are included into the same chapter because they have very close features.

The second section contains the definition of Skip trees and of local rules, the formal definition of the one-to one mapping between Skip lists and Skip trees, and the commutation between their local rules. We advance that the update algorithms of both data types are syntactically identical; they differ only in local calls. We close the section with the distributed algorithm.

The third section includes a distributed algorithm on 2–3 trees.

5.1 Introduction

The concurrent algorithm of W. Pugh [45] is far from being composed of local rules and it needs some kind of circular pointers to maintain information. Both facts raise objections to the comprehensiveness and correctness of the algorithm. We think that the main difficulty derives from the sequential update algorithms: recall that the insertion and deletion ones are based on the path followed in the search stage, therefore, global information is needed to modify local data.

As the main objection to design concurrent algorithms takes place in the structure of Skip lists (a set of linked lists), we propose a new data type denoted Skip trees, composed by trees and inspired in the path followed by sequential search algorithm on Skip lists[1]. The improvement is that we do not have to store the path followed in the search process because this information has been added to nodes of trees, therefore the design of algorithms is able to be based on sets of local rules. Moreover, as there is a one-to-one mapping between Skip lists and Skip trees that commutes with the update algorithms, the wide amount of race properties of Skip lists can be translated to Skip trees. Let us explain the main ideas of the mapping: consider the Skip list of figure 2.10, we group consecutive items with same level, as 60 and 70, into the same node, and attach from them the nodes with smaller level (possible empty) in a sorted order; the Skip tree of figure 5.1 is obtained.

From a dynamic point of view, as the level of each key of Skip lists is given by a random procedure, the height of keys (or the height of the subtree rooted at this keys) is given by the same random procedure; but the interesting observation is that the length (the number of keys) of nodes inherits a random distribution. Therefore, the Skip trees can be

[1] This idea was applied by T. Papadakis in his PhD. thesis [39] to introduce deterministic Skip lists

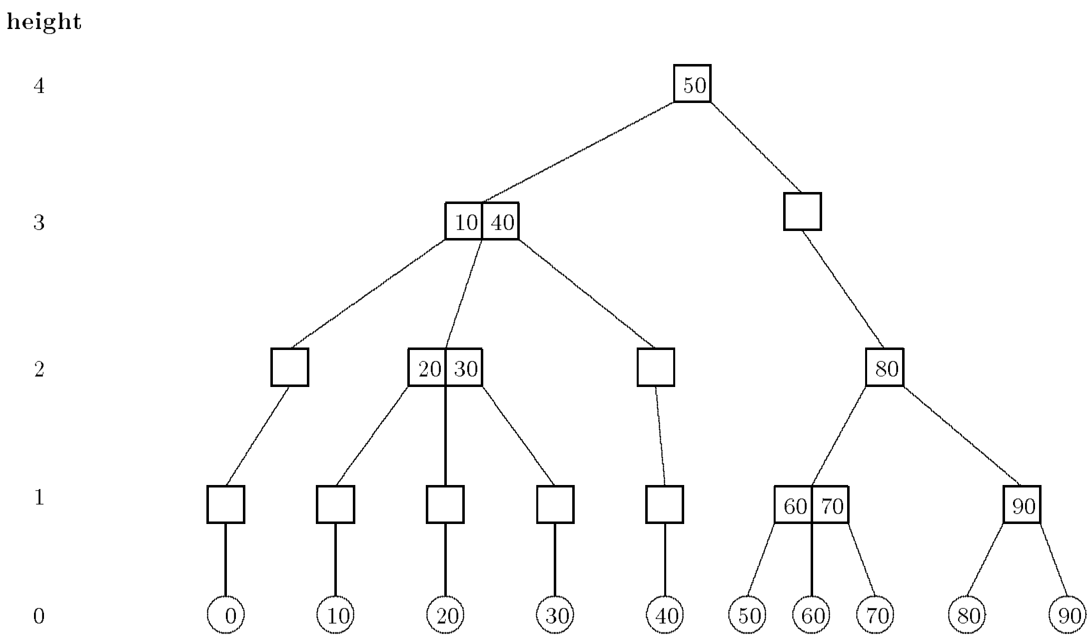


Figure 5.1: Skip tree

considered as a class of unbounded random B-trees because all the leaves have the same depth; we say *unbounded random* because the length of nodes is unbounded and have a random distribution. Moreover, as the expected length of nodes is proportional to a certain parameter $q : 0 < q < 1$ of the distribution, it holds that Skip trees are close to 2–3 trees for $q \simeq .5$ and close to B-trees for $q \simeq 1$.

From a practical point of view a Skip list should be in main memory, but if the data is very large it is necessary to map the structure. This paging is based on sets of consecutive keys, so all update algorithms becomes slower because they have to change pages many times. The Skip trees allows a more efficient paging. Moreover, by means of its relationships with B-trees, the improvements proposed by G. Diehr and B. Faaland [13] can be applied.

Finally, Skip trees have the same performance of the random search trees by C. Aragon and R. Seidel [50]. However random search trees should be applied into a different context, because the probability of a key is given by a continuous identically distributed random variable.

5.2 Skip trees

A Skip tree is a search tree. Every internal node n has three registers, **height**(n) gives us the height of the node, **key**(n) stores a possible empty ordered list of keys and **child**(n) stores an ordered list of pointers. If **key**(n) has r keys **child**(n) contains $r+1$ pointers to its children. Every leaf l contains one key and the information associated with this key in the registers **key**(l) and $\inf(l)$. The leaves does not have any children and all of them have the same depth.

Observe that the number of keys of nodes is not bounded and that we allow internal nodes without keys (empty nodes in the figure) having only one child.

5.2.1 Sequential algorithms and local rules

Before to design the algorithms we select the random distribution that determines the height of keys. As the height is equal to the level on the Skip list, it is determined by the the random variable $1 + NB(1, q)$, where $NB(1, q)$ is the negative binomial distribution with random parameter q. Thus by a "Skip tree with parameter q" we refer to Skip trees whose all keys have been inserted following the previous random distribution. These algorithms can be easily derived from Skip lists ones.

Search

Algorithm: The key is percolated until it is found or it falls from a leaf.

Extraction of local rules: Recall that we add to each node the **waiting** bag which contains the key waiting for to be percolated. The algorithm introduce the key into the bag of the root, and applies the rule that percolates it through the tree.

Rule : Percolation

Guard: *Node n such that* **height**$(n) > 1$ *and* **waiting**$(n) = \{a\}$.

Behavior: *The key a is extracted from* **waiting**(n) *and is handed adequately.*

Variant function: **height**(n).

Spatial scope: *Node n and one son.*

Insertion

Algorithm: The key is percolated: if the search was unsuccessful a new key is attached and the height is randomly computed following the negative binomial distribution. After, we split the key until it reaches its height.

Extraction of local rules: First the keys falls down the tree with

Rule : Percolation (as in searchs)

We colour red the new attached keys in order to send up them. Note that on 2–3 trees or Skip lists the subject which is coloured is the node and the key which is sent up is the middle key. We need a rule that attaches new keys.

Rule : Red Attach

Guard: *Node n such that* **height**$(n) = 1$ *and* **waiting**$(n) = \{a\}$.

Behaviour: *Attach a new leaf with value a at height 0, locate the key a adequately at height 1 and compute randomly the height h. If $h > 1$ colour key red.*

Variant function: NWAITING.

Spatial scope: *Node n.*

Rule : Red Split

Guard: *Node n which stores key a such that* **red**(a).

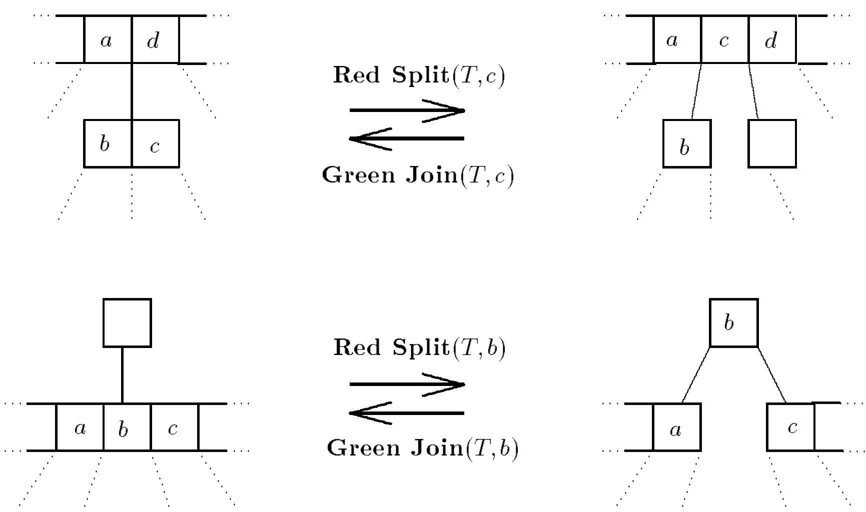


Figure 5.2: **Red Split** and **Green Join** with empty nodes: in the first case when c is split an empty node is generated, and in the second case the split of b takes off the empty node.

Behaviour: *If* **root**(n) *the node* n *is split into two other ones that are hung from a new node that stores* a*. Otherwise* $\neg$**root**(n) *and the node* n *is split into two other ones that are hung adequately at* **parent**(n) *and key* a *is located into father of* n*. If* $h >$ **height**(**parent**(n)) *colour key red.*

Variant function: $h -$ **height**(a)

Spatial scope: *Nodes* n *and* **parent**(n).

In both cases, when necessary empty nodes will be created or propagated. The different context of a into nodes give us different types of split that can be deduced straightforward (see Figure 5.2).

Deletion

Algorithm: The key is percolated until it is found and coloured, or it falls from a leaf. In the first case we join the green key until it is removed.

Extraction of local rules: The first rule is slightly modified:

Rule : Percolation

Guard: *Node* n *such that* **height**$(n) > 1$ *and* **waiting**$(n) = \{a\}$.

Behavior: *The key* a *is extracted from* **waiting**(n)*. If* $a \in$ **key**(n) *colour* a *green, otherwise handed it to the adequate son.*

Variant function: **height**(n).

Spatial scope: *Node* n *and one son.*

Rule : Green Join

Guard: *Node* n *with* $height(n) > 1$ *and stores key* a *such that* **green**(a).

Behaviour: *Group together the nodes of sons located at both sides of the key* a *into only one internal node, and puts* a *between them. Colour key* a *green.*

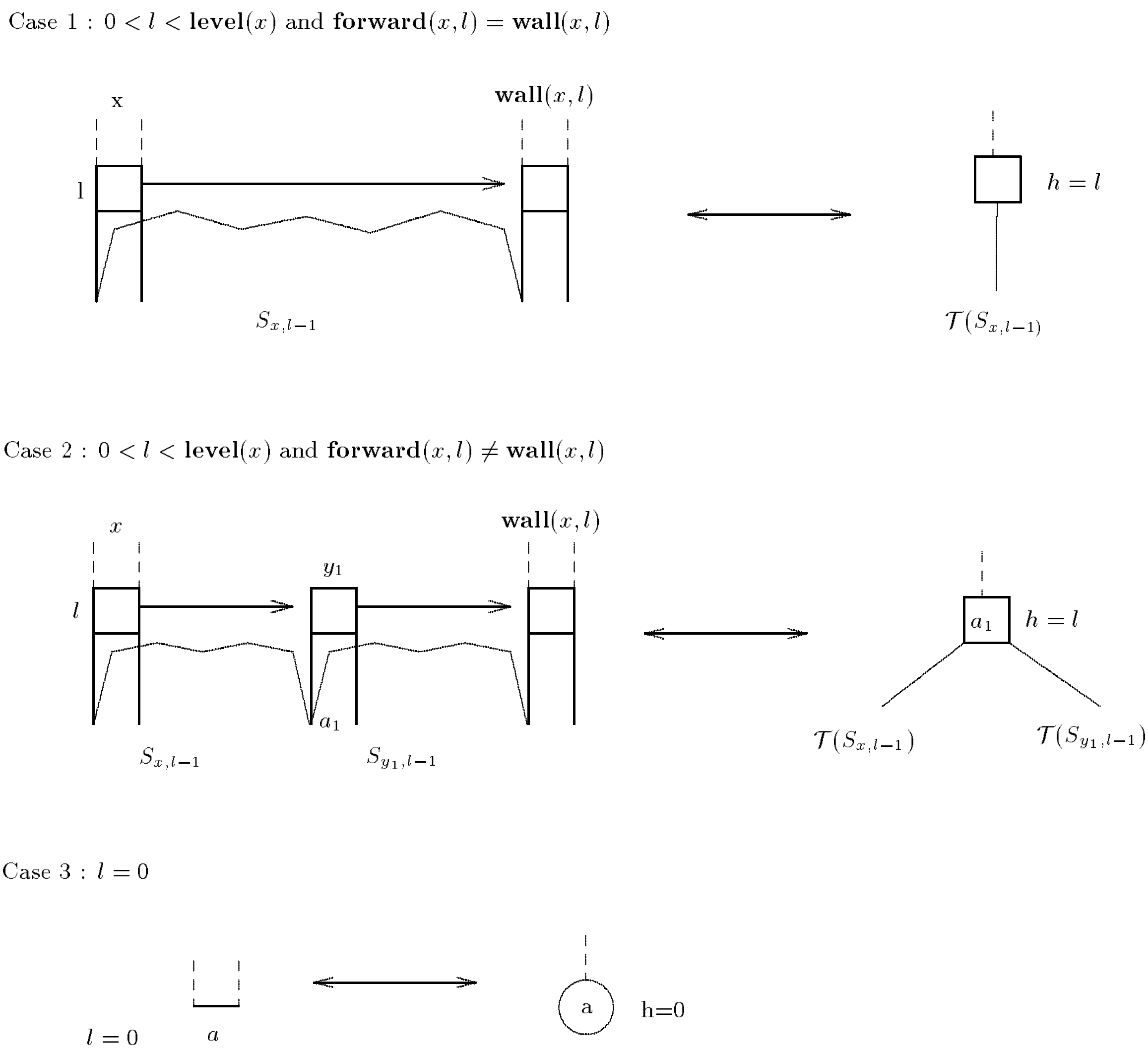


Figure 5.3: Mapping $\mathcal{T}$ between Skip lists and Skip trees

Variant function: $\textbf{height}(n)$.
Spatial scope: *Nodes n and two sons.*

With this rule the height of a decreases by one. It follows:

$$\textbf{Green Join}\,(\textbf{Red Split}(T,a),a) = \textbf{Red Split}\,(\textbf{Green Join}(T,a),a) = T$$

Rule : Green Unattach

Guard: *Node n with* $\textbf{height}(n) = 1$ *and stores key a such that* $\textbf{green}(a)$.
Behaviour: *Removes leaf a and key a from node n.*
Variant function: GREEN.
Spatial scope: *Node n.*

5.2.2 One-to-one mapping between Skip-trees and Skip-lists

We define the mapping between Skip trees and Skip lists, and we prove that it is one-to-one.

Recall that mappings between data structures are a well known topic. L. Guibas and R. Sedgewick [21] embed in a dichromatic frame balanced tree schemes. T. Papadakis [39] gives us a one-to-one mapping between 2–3 trees and deterministic Skip lists. Later, T. Ottmann, H. Six and D. Wood [38] proves that there exists a one-to-one mapping that commutes with updates between AVL trees and 1–2 Brother trees.

Let us recall some basic notations in Skip lists [18]. Given a Skip list S and a node $x \neq$ NIL and some integer $0 \leq l \leq \mathbf{level}(x)$, we write

$$\begin{aligned}\mathbf{wall}(x,l) &= \text{“the first node } y \text{ to the right of } x,\ \textit{i.e.}\\ &\quad\ \mathbf{key}(x) < \mathbf{key}(y), \text{ such that } \mathbf{level}(y) > l\text{”}.\end{aligned}$$

For instance, in Figure 2.10, $\mathbf{wall}(\mathbf{header}(S), 3)$ is the node having key 50. Moreover the *subskiplist at* (x,l) of S, denoted $S_{x,l}$ for short, is the Skip list of height l, where x acts as a header and $\mathbf{wall}(x,l)$ acts as NIL. Node/levels in $S_{x,l}$ are those reachable from (x,l).

Before to give an accurate definition of the mapping

$$\mathcal{T} : \textit{Skip lists} \longrightarrow \textit{Skip trees},$$

let us explain this top-down transformation informally, see figure 5.3

There are three main cases:

- The first case happens when we have a subskiplist $S_{x,l}$ such that $0 < l < \mathbf{level}(x)$ and $\mathbf{forward}(x,l) = \mathbf{wall}(x,l)$. The transformed Skip tree $\mathcal{T}(S_{x,l})$ start with a white node of height l having as child $\mathcal{T}(S_{x,l-1})$. Remark that the smallest key of $\mathcal{T}(S_{x,l-1})$ will be be $\mathbf{key}(x)$.

- The second case corresponds to $l > 0$ and $l < \mathbf{level}(x)$ and $\mathbf{forward}(x,l) \neq \mathbf{wall}(x,l)$. Therefore there is at least one node between $\mathbf{forward}(x,l)$ and $\mathbf{wall}(x,l)$: if there is only one node y_1 between $\mathbf{forward}(x,l)$ and $\mathbf{wall}(x,l)$, then Skip tree $\mathcal{T}(S_{x,l})$ has an internal node with key $a_1 = \mathbf{key}(y_1)$ and two children corresponding to the transformations of $S_{x,l-1}$ and $S_{y_1,l-1}$.

- Case 3. The last case happens when we have $l = 0$ or $S_{x,0}$. Therefore we get a leaf having the key a of x.

In a formal way we define $\mathcal{T}$ as:

Definition 5.1 *Given a Skip list $S = S(\mathbf{header}, \text{NIL})$ we define $\mathcal{T}(S)$ recursively starting from $\mathcal{T}(S(\mathbf{header}, \text{NIL}))$. Given a node/level (x,l) we consider the integer $f \geq 1$ such that $\mathbf{forward}^f(x,l) = \mathbf{wall}(x,l)$, therefore*

1. *When $f = 1$ and $l > 0$ we have*

$$\mathcal{T}(S_{x,l}) \quad = \quad \begin{array}{c} [] \\ | \\ \mathcal{T}(S_{x,l-1}) \end{array}$$

2. *When* $f > 1$ *and* $l > 0$ *we write* $y_i = \mathbf{forward}^i(x, l)$ *(in particular* $x = y_0 = \mathbf{forward}^0(x, l)$*),* $a_i = \mathbf{key}(y_i)$ *and* $T_i = \mathcal{T}(S_{y_i, l-1})$ *we have:*

$$\mathcal{T}(S_{x,l}) \quad = \quad \begin{array}{ccccccccc} [& a_1 & | & a_2 & | & \dots & | & a_{f-1} &] \\ / & & | & & | & & | & & \backslash \\ T_0 & & T_1 & & T_2 & & T_{f-2} & & T_{f-1} \end{array}$$

3. *When* $l = 0$ *we have*

$$\mathcal{T}(S(x, l)) \quad = \quad \begin{array}{c} | \\ \mathbf{key}(x) \end{array}$$

We can easily verify that Skip list given in the figure 2.10 transforms into the Skip tree of the figure 5.1.

Lemma 5.1 *The mapping* $\mathcal{T}$: *Skip lists* $\longrightarrow$ *Skip trees is a one-to-one function.*

5.2.3 Commutation between local rules and mapping

We prove that that there are local rules on Skip lists that commute with local rules of Skip trees.

The local rules of Skip lists are **Red Attach, Red Upward, Green Downward** and **Green Unattach**. They match with rules on Skip trees, more precisely:

Lemma 5.2 *Given a Skip list* S *having all the keys different, there is the following relationship between rules on Skip list* S *and the Skip tree* $\mathcal{T}(S)$*:*

1. *Given a keys* a *belonging to* S *it holds*

$$\mathcal{T}(\mathbf{Red\ Upward}(S, a)) = \mathbf{Red\ Split}(\mathcal{T}(S), a).$$

2. *Given a key* a *which does not appears in* S*, it holds*

$$\mathcal{T}(\mathbf{Red\ Attach}(S, a)) = \mathbf{Red\ Attach}(\mathcal{T}(S), a).$$

Proof: *First, we give a proof outline of (1). Assume that* $a = \mathbf{key}(x)$ *and* $l = \mathbf{level}(x)$*. As in* $\mathbf{Red\ Upward}(S, a)$ *the level of* x *will be* $l + 1$*, we consider in* S *the smallest subskip* S' *containing* x *with* $\mathbf{level}(S') > l$*. When* x *goes up one level it will cut the forward pointer at level* $l + 1$ *of a node* y *belonging to* S'*. We would need to consider three main cases depending on* y *and* $z = \mathbf{forward}(y, l + 1)$*.*

- *Both* y *and* z *verify* $\mathbf{level}(y) = \mathbf{level}(z) = l + 1$*.*
- *Both* y *and* z *verify* $\mathbf{level}(y) > l + 1$ *and* $\mathbf{level}(z) > l + 1$*.*
- *The node* y *verifies* $\mathbf{level}(y) > l + 1$ *but node* z *verifies* $\mathbf{level}(z) = l + 1$*.*
- *Reciprocally we have,* $\mathbf{level}(y) = l + 1$ *but node* z *verifies* $\mathbf{level}(z) > l + 1$*.*

For every one of these cases there are another four possibilities, depending on the brothers surrounding x*.*

- *The node* x *is surrounded by left and right brothers having exactly level* l*.*

- *The node x has only a left brother with level exactly l.*
- *The node x has only a right brother having level exactly l.*
- *There are no brothers surrounding x and having level l.*

The preceding considerations allow us a total of 16 different context for x inside S' straightforward to prove ∎

Lemma 5.3 *Given a Skip list S having all the keys different, there is the following relationship between rules on Skip list S and the Skip tree $\mathcal{T}(S)$:*

1. *Given a keys a belonging to S such that* $\mathbf{level}(a) > 1$, *it holds*

$$\mathcal{T}(\mathbf{Green\ Downward}(S, a)) = \mathbf{Green\ Join}(\mathcal{T}(S), a).$$

2. *Given a key a which does not appears in S, it holds*

$$\mathcal{T}(\mathbf{Green\ Unattach}(S, a)) = \mathbf{Green\ Unattach}(\mathcal{T}(S), a).$$

5.2.4 Expected magnitudes

The time spent in updating is the same as in Skip lists. Given a Skip tree T, the inverse $S = \mathcal{T}^{-1}(T)$ is a Skip list. Therefore we translate the results obtained in Skip lists [47, 19]:

Theorem 5.1 *Let T be a Skip tree of size n and random parameter q. The expected time to search, insert or delete a key is $O(\log_{1/p} n)$ and the probability that the expected height deviates k times from the expected value decreases as $O(n^{-k})$.*

The distribution function of the following magnitudes can be deduced:

Height: it is determined by the random variable $H = 1 + NB(1, q)$ like the height of Skip lists. The expected value is $1/q$.

Length: Let L be the random variable whose value is the number of keys of nodes, by means of the one-to-one mapping L is equal to the number of consecutive keys with the same level, magnitude that is denoted *gap* by T. Papadakis [39] and that is given by the random distribution $L = NB(1, p)$ whose expected value is q/p.

The Skip trees inherits from Skip lists the following expected properties:

Theorem 5.2 *Let T be a Skip tree with parameter q and size n ($q + p = 1$):*

1. *The expected height of T is $O(\log_{1/p} n)$, and the probability that the expected height deviates k times from the expected value decreases as $O(n^{-k})$.*

2. *The expected length of internal nodes is q/p, and the probability that it deviates c times from the expected value is $q^{\lceil cq/p \rceil}$.*

3. *The expected number of splits or joins while updating a key is $1/q$.*

Proof: 1. *Given a random Skip tree T with parameter q and n keys, the Skip list $S = \mathcal{T}^{-1}(T)$ is an usual Skip list with n keys where the height of nodes is determined by the random variable $1 + NB(1,q)$, then the height of S holds $O(\log_{1/p} n)$. To prove that the expected height decreases exponentially we apply the Chernoff tail bound lemma [11],*

$$\mathrm{Prob}\{H \geq a\} \leq E(e^{tH})/e^{ta} \qquad \forall a, t > 0.$$

The expectation of H is $E(e^{tH}) = q/(e^{-t} - p)$ being $q = 1 - p$. Then

$$\mathrm{Prob}\{H \geq c \log_{1/p} n\} \leq \frac{q}{e^{-t} - p} / e^{tc \log_{1/p} n} = \frac{q}{p^{k/c} - p} n^{-k}$$

taking $t = \frac{k}{c} ln(1/p)$. Observe that this result is well defined for $1 \leq c < k$.

2. *Let L be the random variable whose value is the number of keys in a node of a tree T, then*

$$\mathrm{Prob}\{L > cE(L)\} = \mathrm{Prob}\{L > cq/p\} = \sum_{k \geq cq/p} \mathrm{Prob}\{L = k\} = \sum_{k \geq cq/p} pq^k = q^{\lceil c \frac{q}{p} \rceil}$$

3. *The expected number of splits and joins is equal to the expected height of one key, then*

$$E(1 + NB(1,q)) = 1 + E(NB(1,q)) = 1 + p/q = 1/q$$

■

5.2.5 Close to B-trees

Although Skip list can not be efficiently broken (or paginated) to store some parts in secondary or external memory, Skip trees have this capability by means of their structure. This fact and the closeness between B-trees and Skip trees with parameter $q \simeq 1$ suggest the use of Skip trees to handle index of data bases, as B-trees do. The following lemma explores this possibility.

Lemma 5.4 *Let T be a Skip tree of size n and parameter q, recall L and H as random variables giving the length of nodes and the height of tree:*

1. $\mathrm{Prob}\{L > cE(L)\} \simeq (eq)^{-c}$ *for larger values of q.*
2. $\mathrm{Prob}\{H > 1 + \log_{1/p}(n)\} = 1 - \left(1 - \frac{p}{n}\right)^n \simeq p$ *for $n >> 1$ and $p \simeq 0$.*

Proof: 1. *Recall that* $\mathrm{Prob}\{L > cE(G)\} \leq \left(q^{q/p}\right)^c$. *This expression can be approximated by*

$$q^{\frac{q}{p}} = (1-p)^{\frac{1-p}{p}} = \frac{(1-p)^{1/p}}{(1-p)} \simeq (qe)^{-1}$$

2. *Recall that the height of each node is also a negative binomial distribution but with parameter q: $H = NB(1,q) + 1$. Say $L(n) = \log_{1/p} n$, then*

$$\mathrm{Prob}\{H_n > L(n)+1\} = 1 - \mathrm{Prob}\{H_n \leq 1 + L(n)\} = 1 - \prod_{k=1}^{n} \mathrm{Prob}\{H \leq 1 + L(n)\}$$

because all the keys should have smaller height,

$$= 1 - \prod_{k=1}^{n}\left(1 - p^{L(n)+1}\right) = 1 - \left(1 - p^{L(n)+1}\right)^n = 1 - \left(1 - \frac{p}{n}\right)^n \overset{n>>1}{\simeq} 1 - e^{-p} \overset{p\simeq 0}{\simeq} p$$

because they are independent random variables.

■

This previous lemma suggest that Skip trees manage Data Base indexes as well as large are n and q, but taking care about the length of nodes. Note that the probability that the height was greater than the expected one is very small (p), then the height of the Skip tree will be similar to those of B-tree. But this similarity does not holds for the length of nodes: for instance, $c = 1$ determines that about n/e nodes grows more than the expected value. Therefore Skip trees will have an overflow of keys from nodes; this fact suggests to implement Skip trees with variable length nodes as E. McCreight [32] or G. Diehr and B. Faaland [13] propose for B-trees.

5.2.6 Distributed algorithm

We design the concurrent algorithms with local rules in which some keys are search, others ones inserted an the remainder deleted. As all them are gathered into waiting bags we assume that there exists a flag to differentiate them. **Percolation** rule first test this flag before to manage the extracted key.

The Skip trees holds two properties:

$$P_1 \equiv \text{the height of the keys has been randomly computed}$$
$$P_2 \equiv \text{all leaves have the same depth}$$

Our development maintains P_2 invariant and a weakest version of P_1: the height af a key is smaller than the random computed value.

Search and insertion algorithm

We consider first the case in which there are keys to be search and inserted. We take into account the rules **Percolation, Red Attach** and **Red Split** of section 5.2.1 and we sligtly modify them because nodes can store non empty waiting bags.

Rule : Percolation

Guard: *Node n such that* $\mathbf{height}(n) > 1$ *and* $\mathbf{waiting}(n) \neq \emptyset$.

Rule : Red Split

Aditional behaviour: *: Split also the waiting bag.*

We define the following variant functions: let n and k denote nodes and keys respectively, and h_k the randomly computed height of key k, then

$$\text{NWAITING} = \sum_n |\mathbf{waiting}(n)|$$

$$\text{WAIT} = \sum_n \mathbf{height}(n) \cdot |\mathbf{waiting}(n)|$$

$$\text{RDEPTH} = \sum_k (h_k - \mathbf{height}(k)) \cdot \mathbf{red}(k)$$

Lemma 5.5 *(Interference) The following holds:*

1. *Rule* **Red Attach** *strictly decreases variant function* NWAITING.
2. *Rule* **Percolation** *strictly decreases* WAIT *and decreases* NWAITING.
3. *Rule* **Red Split** *strictly decreases* RDEPTH *and leaves unchanged* (NWAITING, WAIT).

Lemma 5.6 *The distributed algorithm composed of the following rules is correct.*

{**Percolation**, **Red Attach**, **Red Split**}

Proof: Safety: *If no rule applies all rules have been search and inserted.* Liveness: *The variant function* (NWAITING, WAIT, RDEPTH) *strictly decreases at each application of any rule.* ■

Search, insertion and deletion algorithm

Now we address the general case in which there are keys waiting for to be search, inserted or deleted (green keys). We define the variant functions

$$\begin{aligned} \mathrm{GREEN} &= \sum_k \mathbf{green}(k) \\ \mathrm{GHEIGHT} &= \sum_k \mathbf{height}(k) \cdot \mathbf{green}(k) \end{aligned}$$

We take into account insertion previous rules and rules **Green Join** and **Green Unattach**. We modify the second one:

Rule : Green Join

Aditional behaviour: *Join the keys contined into waiting bags of sons.*

Lemma 5.7 *(Interference) The following holds:*

1. *Rules* **Green Join** *and* **Green Unattach** *leave unchanged*

   (NWAITING, WAIT, RDEPTH)

2. *Rule* **Green Unattach** *strictly decreases variant function* GREEN.
3. *Rule* **Green Join** *strictly decreases* GHEIGHT *and leaves unchanged* GREEN.

Theorem 5.3 *The distributed algorithm composed of the following rules is correct.*

{**Percolation**, **Red Attach**, **Red Split**, **Green Join**, **Green Unattach**}

Proof: Safety: *if no rules applies all items have been percolated and the keys have been updated.* Liveness: *The variant function*

(NWAITING, WAIT, RDEPTH, GREEN, GHEIGHT)

strictly decreases at each step of any rule. ■

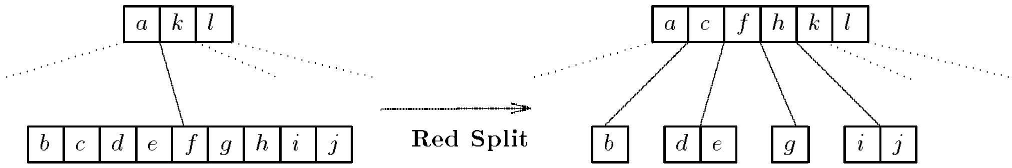


Figure 5.4: **Red Split**

5.3 2–3 trees: a distributed algorithm

Recall that all nodes have one or two keys (P_1) and all leaves have the same depth (P_2). We keep as invariant the second property, and we let nodes to store any number of keys during the algorithms. Assume that H is a constant as big as necessary.

Search and insertion algorithm

We first access the single case which searchs and inserts keys. We recall from section 2.3 the rules **Percolation, Red Attach, Red Split** and **Whitening Red Root**. We assume that there are many full **waiting** bags, then the last two rules have to be revised:

Rule : Red Split

Guard: *Node n such that* $\mathbf{red}(n)$ *and* $\neg\mathbf{root}(n)$.

Behavior: *Separate keys in groups of two or three consecutive items and insert the rightmost key (except the last group) into the parent node. Finally attach new nodes adequately and colour parent red if it stores more that two keys (Figure 5.4).*

Variant function: *The sorted pair (*RED,$H-\mathbf{height}(n)$*).*

Spatial scope: *Node n and its parent.*

Rule : Whitening Red Root

Additional behaviour: *Create parent node, insert keys and colour red if it has more that two keys.*

Lemma 5.8 *All rules leave P_2 invariant.*

We recall the variant functions. Let n denote nodes and H a big constant:

$$\text{NWAITING} = \sum_n |\mathbf{waiting}(n)|$$

$$\text{WAIT} = \sum_n \mathbf{height}(n) \cdot |\mathbf{waiting}(n)|$$

$$\text{RKEYS} = \sum_n \mathbf{red}(n) \cdot |\mathbf{key}(n)|$$

$$\text{RDEPTH} = \sum_n (H - \mathbf{height}(n)) \cdot \mathbf{red}(n)$$

Lemma 5.9 *(Interference) The following holds:*

1. *Rule* **Red Attach** *strictly decreases variant function* NWAITING.
2. *Rule* **Percolation** *strictly decreases* WAIT *and decreases* NWAITING.
3. *Rule* **Red Split** *strictly decreases* RDEPTH *and leaves unchanged* (NWAITING, WAIT).
4. *Rule* **Whitening Red Root** *strictly decreases* RKEYS *and leaves unchanged*

$$(\mathrm{NWAITING}, \mathrm{WAIT}, \mathrm{RDEPTH})$$

.

5. *Rule* **Red Split** *strictly decreases* RDEPTH *and leaves unchanged* (NWAITING, WAIT).

Theorem 5.4 *The distributed algorithm composed of the following rules is correct.*

{**Percolation**, **Red Attach**, **Whitening Red Root** , **Red Split**}

Proof: *We prove safety and lveness properties.*

Safety: *If no rules applies all nodes have at most two keys, all keys have been percolated and updated. The invariant predicate P_2 ensures that all leaves have the same depth.*

Liveness: *The variant function*

$$(\mathrm{NWAITING}, \mathrm{WAIT}, \mathrm{RDEPTH}, \mathrm{RKEYS})$$

strictly decreases at each step of any rule. ■

Search, insertion and deletion algorithm

We address the general case in which the keys can be search and updated. We can not apply the **Update** rule of sequential algorithm. Recall that this rule sends up a pair (x, y) whose meaning is the substitution of key x by key y. But before an update reaches node x rule **Percolation** falls down a key z such that $y < z < x$ by sending it through the left son of x: this fact creates the inconsistency of data.

As there can be red nodes with green keys, *i.e.* nodes with more than two keys being some of them green, we modify the meaning of red

red$(n) \equiv$ n contains more than two keys, and none of them are green.

We recall deletion rules from Skip trees:

Rule : Percolation

Aditional behaviour: *if the key extracted from a waiting bag is stored into n, colour key green.*

Rule : Red Split

Aditional behaviour: *Green keys cannot be sent up and there is not possible to create empty nodes.*

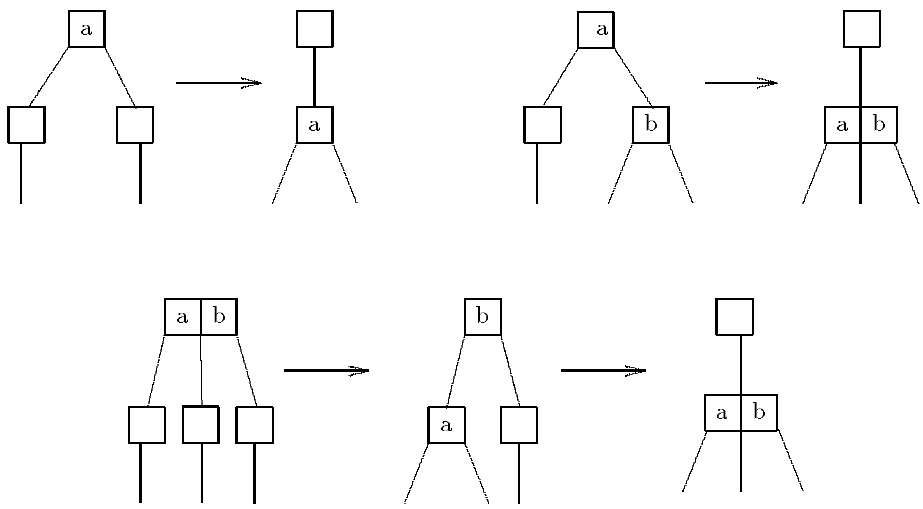


Figure 5.5: Cases of **Garbage Filler**

This behaviour determines the prior claim of the removing rules than the inserting ones.

Rule : Green Join

Guard: *Node n with $height(n) > 1$ stores a key a such that* **green**(a).

Behaviour: *Group together the nodes and the waiting bags of sons, locate them at both sides of the key a into only one internal node, and puts a between them. Colour key a green.*

Variant function: height(a).

Spatial scope: *Nodes n and two sons.*

Rule : Green Unattach

Guard: *Node n with* **height**$(n) = 1$ *and stores key a such that* **green**(a).

Behaviour: *Removes leaf a and key a from node n.*

Variant function: GREEN.

Spatial scope: *Node n.*

But these rules have an important drawback: some nodes can became empty, *i.e.* store no keys. Then we need to define another rule that sends this nodes up. We define the predicate and the variant function

$$\mathbf{empty}(n) \equiv \text{n has no keys.}$$

$$\text{EMPTY} = \sum_n \mathbf{empty}(n)$$

The rule that deals with empty nodes is

Rule : Garbage Filler

Guard: *Node n with no keys. If* **root**(n) *then* **waiting**$(n) = \emptyset$.

Behaviour: *If* **root**(n) *then remove it, otherwise there are empty nodes with non-empty parents. Keep a key from* **parent**(n) *and join to a son (see Figure 5.5).*

Variant function: *(*EMPTY*,H-height(n)).*

Spatial scope: *Nodes n, their sons and its parent.*

Lemma 5.10 *All rules leave P_2 invariant.*

We define the variant functions

$$\text{GHEIGHT} = \sum_k height(k) \cdot \mathbf{green}(k)$$
$$\text{EDEPTH} = \sum_n (H - \mathbf{height}(n)) \cdot \mathbf{empty}(n)$$

Lemma 5.11 *The following holds:*

1. *Rules* **Green Join, Green Unattach** *and* **Garbage Filler** *decrease*

   $$(\text{NWAITING}, \text{WAIT}).$$

2. *Rules* **Green Join** *and* **Green Unattach** *strictly decrease*

   $$(GREENHEIGHT, \text{GREEN}).$$

3. *Rule* **Garbage Filler** *strictly decrease*

   $$(\text{EDEPTH}, \text{EMPTY})$$

   and decrease

   $$(GREENHEIGHT, \text{GREEN}).$$

4. *Rules* **Red Split** *and* **Whitening Red Root** *strictly decrease*

   $$(\text{RDEPTH}, \text{RKEYS})$$

   and leave unchanged all others.

Theorem 5.5 *The distributed algorithm composed of the following rules is correct*

{ **Percolation**, **Red Attach**, **Whitening Red Root** , **Red Split**, **Green Join**, **Green Unattach**, **Garbage Filler** }

Chapter 6

AVL trees: a fine-grained distributed algorithm

We face in this chapter the distributed algorithm with local rules on AVL trees. The distributed algorithm is designed with sequential rules extracted in section 2.4 without uncoupling them. This fact makes our design more fine-grained than the previous ones reported by J.L.W. Kessels and O. Nurmi, E. Soisalon-Soininen and D. Wood, because the propagation of information is separated from the balance of trees. As some new registers are added to nodes, we recall them and we rewrite the known rules showing how the new register must be updated.

The first section contains a sketch of previous approaches by J.L.W. Kessels and O. Nurmi, E. Soisalon-Soininen and D. Wood. The second section contains the self-balancing distributed algorithm which transforms binary trees with only local information into AVL trees. The third and four sections are devoted to the search, insert and delete distributed algorithm on AVL trees. Finally, the last section explains how previous coarse-grained algorithmic approaches can be emulated by our algorithm.

6.1 Related works

In the J.L.W. Kessels's approach, the height of every node n is adjusted with the contents of a private *red* register. This register defines a *dynamic height* as

$$\mathbf{dheight}(n) = 1 - \mathbf{red}(n) + \max\{\mathbf{dheight}(p), \mathbf{dheight}(q)\}$$

whose meaning is that *red* nodes do not contribute to the dynamic height (see section 3.1.1). In Kessels' work, a *dynamic balance* of a node, written $\mathbf{dbal}(n)$, is defined in terms of the dynamic height of the subtrees. The eight local transformations are designed to maintain $\mathbf{bal}(n) \in \{-1, 0, +1\}$. To do it, it is necessary to deal with the **Red Rebalancing** rule which couple the **Red Propagation** rule with **Red Rotation**(see Page 33). Intuitively, a **Red Propagation** improves the consistency of the data structure: when all the nodes verify $\neg\mathbf{red}(n)$, then the $\mathbf{dheight}(n)$ coincides with the real height. Otherwise, a **Red Rotation** attempts to balance subtrees. In Kessels's approach, the question about rotations is solved as follows: one rotates (if necessary) just after a red propagation in order to maintain the balance.

O. Nurmi, E. Soisalon-Soininen and D. Wood [36, 37] extended Kessels's work by giving safe and fair algorithms to deal with insertions and deletions in external AVL tree (i.e., the actual keys are stored only in the leaves of the tree). They introduced a *tag* such that $\mathbf{tag}(n) \in \{-1, 0, 1, 2, \ldots\}$, a relaxed height defined as

$$\mathbf{rheight}(n) = 1 + \mathbf{tag}(n) + \max\{\mathbf{rheight}(p), \mathbf{rheight}(q)\}$$

and a relaxed balance factor $\mathbf{rbal}(n)$. As before the upward propagation of a unit of tag is followed by a rotation in order to maintain $\mathbf{rbal}(n) \in \{-1, 0, +1\}$. Finally, K.S. Larsen [30] modified the rules given in [36] in order to increase the degree of concurrency and to study the complexity of the rebalancing.

In our approach, we answer the question of Kung and Lehman about rotations as follows: a rotation *can* just take place at any unbalanced node. Therefore, when no more rotations are possible, then we have an AVL tree. This approach is intuitively clear but working out the technical details is far from obvious. For instance, if rotations are separated from propagations, then it is harder to maintain any notion of local balance and we will get $\mathbf{bal}(n) \in \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. Moreover, the local information can be arbitrarily unfaithful, and one may decide to rotate the wrong subtree. Liveness is then questionable.

6.2 Self-balancing distributed search trees

We consider binary search trees as a distributed system, where the processes are the nodes and the links are the father/son edges. Each node n is thus equipped with a number of *private registers* used to store the local information it holds about the global system:

lefth(n): The apparent height of the left son of n, at the best of its knowledge;

righth(n): The apparent height of the right son of n at the best of its knowledge.

We define three additional quantities:

lheight(n): The apparent local height of node n, at the best of its knowledge.

$$\mathbf{lheight}(n) = \max(\mathbf{righth}(n), \mathbf{lefth}(n)) + 1$$

lbal(n): The apparent balance of node n, at the best of its knowledge.

$$\mathbf{lbal}(n) = \mathbf{righth}(n) - \mathbf{lefth}(n)$$

delta(n): The difference between the height known by the father m of node n and the current apparent height of node n, at the best of its knowledge.

$$\mathbf{delta}(n) = \begin{cases} \mathbf{lefth}(m) - \mathbf{lheight}(n) & \text{if } n \text{ is the left son of } m \\ \mathbf{righth}(m) - \mathbf{lheight}(n) & \text{if } n \text{ is the right son of } m \end{cases}$$

By convention, we set $\mathbf{delta}(n) = 0$ if n is the root of the tree.

In the following sections we present a safe and live distributed algorithm to update AVL trees. It can be described easily by a set of evolution rules. Any rule can be selected and applied in any order as soon as its guards are satisfied. The application of a rule modifies the values stored into the local registers. Observe that these quantities may be arbitrarily different from their *real* values. We do not try to define accurately what we mean by *real*. Informally, to get the *(real) height* or *(real) balance* we freeze the tree and we compute these values as usual. Whenever a local register is updated with the information sent to it (at some preceding moment) we call this information *apparent*. Of course, the value of the apparent information can be very different from the value of the real information. We say that node n is *(apparently) balanced* if $\mathbf{lbal}(n)$ is 0, 1 or -1. Node n is apparently balanced if it is according to the information transmitted to it by its sons at some preceding moments. As the information can be delayed by some of the sons, $\mathbf{lbal}(n)$ reflects roughly the reality. When $\mathbf{lbal}(n) \notin \{-1, 0, 1\}$ we call n *(apparently) unbalanced*.

Let us say what we mean by *(apparently) faithful* information or *(apparent) stability*. Intuitively a node n has apparent faithful information if it is correct from the point of view of its neighbors. The quantity $\mathbf{delta}(n)$ serves to measure the degree of faithfulness. If needed, it can be considered as a new internal register of n. In this case, n knows when its father m has faithful information about $\mathbf{lheight}(n)$. If n is a left son of m, then node m has faithful information about $\mathbf{lheight}(n)$ whenever $\mathbf{lheight}(n) = \mathbf{lefth}(m)$ (that means $\mathbf{delta}(n) = 0$) otherwise it has unfaithful information. When $\mathbf{delta}(n) = 0$, node n does not need to send any new information to update the values of m. We say node n is *(apparently) stable*. Otherwise, $\mathbf{delta}(n) \neq 0$, and some useful information is still being held in n: we say n is *(apparently) unstable*.

As we will see later, a lot of unfaithful information may be generated along the whole tree. In order to guarantee a correct final result, we need to anchor the correct values of the local height at some nodes. A distributed sorting tree is *well-founded* if $\mathbf{lefth}(n) = 0$ (resp. $\mathbf{righth}(n) = 0$) for any node n with an empty left (resp. right) node. This means that (at least) the border nodes have an accurate knowledge about their height. We have the following easy result.

Lemma 6.1 *Let T be a distributed, well-founded search tree.*

- *If all nodes have faithful information, that is, are (apparently) stable, then the local information coincides with the real information. For all n we have* $\mathbf{lheight}(n) = \mathbf{height}(n)$ *and* $\mathbf{lbal}(n) = \mathbf{bal}(n)$.

- *If 1) all nodes have faithful information, are (apparently) stable, and 2) all nodes are (apparently) balanced, then the tree is (really) balanced. It is thus an AVL tree in the usual sense.*

Proof: *As* all *nodes are* apparently *stable, the* apparent *value* $\mathbf{lefth}(n)$ *and* $\mathbf{righth}(n)$ *are just the* real *heights of the sons of any node n: is it true for empty sons, as the tree is well-founded, and for all sons by induction. As the nodes are* apparently *balanced, they are* really *balanced.* ■

We can now state our basic framework. Consider a distributed, well-founded search tree T. We aim at transforming T through a sequence of atomic evolution rules into a tree T' with the following properties:

1. T' holds the same keys as T, also in sorted order, and T' is well-founded;

2. All nodes of T' are (apparently) stable and (apparently) balanced.

By the lemma above, T' is an AVL tree, as wanted. Observe that we make absolutely no assumption on the initial knowledge of the internal nodes of T. In this sense, the behavior is *self-balancing*, very much in the same sense as for self-stabilizing distributed algorithms.

6.2.1 The rules

The algorithm should eventually *yield an AVL tree* in spite of *unfaithful information.* Two problems have to be faced. First, as the information can be delayed at any descendent m of n (we mean $\mathbf{delta}(m) \neq 0$), node n has a very rough knowledge of its real height and balance. We must allow information to flow upwards (our tree is well-founded) to get faithful information. Lemma 6.1 states that local information coincides with real information when all information is faithful. Second, we have to end up with an AVL tree. Every node should thus try to improve the situation by doing rotations. To take a decision, node n can only access local information. A rotation around n may seem very good from the local point of view whereas it actually worsens the global situation. This leads to two well known evolution rules:

Propagation: It propagates information upwards from a son to his father. Applying this rule increases the quality of the father's knowledge, that is the *global stability.* Note that it is a generalization of **Red Propagation** rule.

Rotation: It is a generalization of the rule **Red Rotation** of the sequential AVL tree algorithm. Applying this rule increases the *global balance* of the tree. In contrast with the original AVL tree algorithm, we have no control on rule application: we can no longer assume that the balance of a node to be rotated is 2 or -2. It may happen that the imbalance of a node suddenly reaches large values.

Let us start with a rule allowing to any son n to update the information of its father concerning its apparent height.

Rule : Propagation

Guard: *A subtree* $m(n(A, B), C)$ *with* $\mathbf{delta}(n) \neq 0$

Behavior: *Update* m *into* m' *with*

$$\mathbf{lefth}(m') := \mathbf{lheight}(n)$$

Spatial scope: *Node* n *and its father* m*.*

Note: *Symmetrically if* n *is the right son of* m*.*

We can easily prove that applying repeatedly the propagation rule to a well-founded tree will eventually correct the local values stored into the nodes. At the end, local information will coincide with the real information. Therefore, the Propagation Rule lets the correct information flow up. Now, we consider rotations to improve the balance of the tree. First of all, it makes no sense to require a locally better tree if even local data are unfaithful. We therefore require the sons to be stable. We could think that as rotations equilibrate (at least locally) subtrees, they also make these subtrees smaller (at least with respect to the local height). Unfortunately this is not always the case. This leads us to split single rotations into two cases according to this fact. As we will see later, the local height of the rotated subtree strictly decreases when $\mathbf{lbal}(p) \neq 0$, and it remains constant when $\mathbf{lbal}(p) = 0$.

Rule : Rotation (single left-left case)

Guard: *A subtree* $n(p(A, B), C)$, $\mathbf{lbal}(n) \leq -2$, *p is stable, and* $\mathbf{lbal}(p) < 0$.

Behavior: *Restructure the tree into $p'(A, n'(B, C))$ with the obvious updating of the registers:*

$$\begin{array}{ll} \mathbf{key}(n') := \mathbf{key}(n) & \mathbf{key}(p') := \mathbf{key}(p) \\ \mathbf{lefth}(n') := \mathbf{righth}(p) & \mathbf{lefth}(p') := \mathbf{lefth}(p) \\ \mathbf{righth}(n') := \mathbf{righth}(n) & \mathbf{righth}(p') := \mathbf{lheight}(n') \end{array}$$

Spatial scope: *Node n and its left son p.*

Note: *Symmetrically for the single right-right case: $n(A, q(B, C))$, $\mathbf{lbal}(n) \geq 2$, q is stable, and $\mathbf{lbal}(q) > 0$.*

Rule : Rotation (single left-equal case)

Guard: *A subtree $n(p(A, B), C)$, $\mathbf{lbal}(n) \leq -2$, p is stable, and $\mathbf{lbal}(p) = 0$.*

Behavior: *Restructure the tree into $p'(A, n'(B, C))$ with the obvious updating of the registers:*

$$\begin{array}{ll} \mathbf{key}(n') := \mathbf{key}(n) & \mathbf{key}(p') := \mathbf{key}(p) \\ \mathbf{lefth}(n') := \mathbf{righth}(p) & \mathbf{lefth}(p') := \mathbf{lefth}(p) \\ \mathbf{righth}(n') := \mathbf{righth}(n) & \mathbf{righth}(p') := \mathbf{lheight}(n') \end{array}$$

Spatial scope: *Node n and its left son p.*

Note: *Symmetrically for the right-equal case: $n(A, q(B, C))$, $\mathbf{lbal}(n) \geq 2$, q is stable, and $\mathbf{lbal}(q) = 0$.*

In contrast to the single case (as we will see), double rotations always make the local height of the subtree to decrease.

Rule : Rotation (double left-right case)

Guard: *A subtree $n(p(A, q(B, C)), D)$, $\mathbf{lbal}(n) \leq -2$, p and q are stable, $\mathbf{lbal}(p) > 0$.*

Behavior: *Restructure the tree into $q'(p'(A, B), n'(C, D))$ with the obvious updating of the registers:*

$$\begin{array}{lll} \mathbf{key}(n') := \mathbf{key}(n) & \mathbf{key}(p') := \mathbf{key}(p) & \mathbf{key}(q') := \mathbf{key}(q) \\ \mathbf{lefth}(n') := \mathbf{righth}(q) & \mathbf{lefth}(p') := \mathbf{lefth}(p) & \mathbf{lefth}(q') := \mathbf{lheight}(p') \\ \mathbf{righth}(n') := \mathbf{righth}(n) & \mathbf{righth}(p') := \mathbf{lefth}(q) & \mathbf{righth}(q') := \mathbf{lheight}(n') \end{array}$$

Spatial scope: *Node n, its son p and p's son q.*

Note: *Symmetrically for the right-left case: $n(A, q(p(B, C), D))$, $\mathbf{lbal}(n) \geq 2$, q and p are stable, and $\mathbf{lbal}(q) < 0$.*

We have decoupled the rules designed to improve the consistence of the local data from the rules aimed at balancing the tree. **Propagation** rule deals with registers $\mathbf{delta}(n)$. These registers measure the faithfulness between local data. **Rotation** rule work with quantities $\mathbf{lbal}(n)$ measuring the imbalance.

6.2.2 Safety

The Safety Property guarantees that, whenever we start with faithful information at the border nodes of a well-founded tree, any tree obtained through the evolution rules is fine: *nothing bad may happen!* Moreover, if the algorithm stops, then we get a good result.

Lemma 6.2 *Assume that T is a distributed, well-founded search tree. 1) Let T' be obtained by the application of some rule, then T' is a well-founded search tree, too. 2) Moreover, if no rule applies on T, as it is locally stable and locally balanced, T is an AVL tree.*

6.2.3 Liveness

It remains to prove the difficult part, that is, the evolution rules do not admit any infinite sequence of applications: *something good, e.g. termination, must eventually happen!* As mentioned above, Propagation and Rotation Rules can fight ones against others.

Propagation Rule: On one side, this rule increases the quality of the father's knowledge, that is the *global stability*. This is good. But on the other side, it may reveal to it that it is in fact more unbalanced than previously believed. It may therefore decrease the *global balance* and this is bad.

Rotation Rule: Intuitively, this rule increases the *global balance* of the tree. This is good. But, applying a rotation at a node usually changes the height of this node. As a side effect, it invalidates the knowledge of its father. It may therefore worsen the *global stability*. This is bad.

As we have to cope with two contradictory effects, it is by no means obvious that the whole process eventually stops. Some oscillating behavior could result, where stability and balance alternatively improve. An in-depth study of their interaction is therefore necessary.

The proof of Liveness Property proceeds through a number of lemmas to capture the precise action of each rule on stability and balance. Unless specified, all lemmas are proved by easy but tedious case enumerations. The key point is that a loss in one quantity is always matched by a larger gain in the other! First of all, observe that we now apply rotation rules to highly unbalanced nodes n (we mean $|\mathbf{lbal}(n)| > 2$). The resulting tree is *not* balanced in general, in contrast with the usual AVL tree algorithm. Yet, we can still easily catch the global effect by an exhaustive case enumeration: the height may not increase.

Lemma 6.3 (Evolution of lheight)

Rotation rule (single left-left case): $\mathbf{lheight}(p') = \mathbf{lheight}(n) - 1$.

Rotation rule (double left-right case): $\mathbf{lheight}(q') = \mathbf{lheight}(n) - 1$.

Rotation rule (single left-equal case): $\mathbf{lheight}(p') = \mathbf{lheight}(n)$.

Thus, **lheight** may not increase on applying a rotation at node n, and it is left unchanged for all other nodes in the tree. Remark that rotations around n cannot decrease the stability of n (we mean $\mathbf{delta}(p') \geq \mathbf{delta}(n)$, where p is the new root). When $\mathbf{delta}(n) > 0$ we say that *n has positive unstability*. In this case $\mathbf{lefth}(n)$ has become smaller but this variation has not been yet communicated to m. This information is being delayed at n and m thinks that n is higher than really it is. Intuitively,

$$\begin{aligned}\mathbf{delta}(n) > 0 \quad &\equiv \text{“The father of } n \text{ believes that } n \text{ is } \mathbf{delta}(n) \text{ units } \textit{higher} \text{ than } n\text{”}\\ &\equiv \text{“The father of } n\ \textit{overestimates} \text{ in } \mathbf{delta}(n) \text{ units the local height of } n\text{”}\end{aligned}$$

The variation of the local apparent height through rotations has an interesting consequence: positive unstability is preserved by the rules. Say a tree is *consistent* if $\mathbf{delta}(n) \geq 0$ for all nodes n.

Lemma 6.4 *Let T be a search tree. Let T' be obtained by the application of some rule. If T is consistent, then T' is, too.*

Proof: *This is clear if the rule is a rotation at node n:* $\mathbf{delta}(n)$ *may not decrease, and all other values are left unchanged . Assume it is a propagation at node n with father m, and that node m has a father o. Then* $\mathbf{delta}(n') = 0$. *As* $\mathbf{lheight}(n) < \mathbf{lefth}(m)$ *by hypothesis, updating* $\mathbf{lefth}(m)$ *let* $\mathbf{lheight}(m)$ *decrease. As* $\mathbf{lheight}(m) \leq \mathbf{lefth}(o)$, *this lets* $\mathbf{delta}(m)$ *increase:* $\mathbf{delta}(m') \geq \mathbf{delta}(m)$. *Thus, all values remain positive.* ■

When $\mathbf{delta}(n) < 0$ we say that *n has negative unstability*. In this case the n apparent local height has grown up from the last message sent its the father m. Therefore m thinks that n is lower that really is it.

$$\begin{aligned}\mathbf{delta}(n) < 0 \quad &\equiv \text{“The father of } n \text{ believes that } n \text{ is } |\mathbf{delta}(n)| \text{ units } \textit{smaller} \text{ than } n\text{”}\\ &\equiv \text{“The father of } n \textit{ underestimates } n \text{ local height in } |\mathbf{delta}(n)| \text{ units”}\end{aligned}$$

In the following two sections we deal separately with

Consistent trees: All the nodes satisfy $\mathbf{delta}(n) \geq 0$. In words, all fathers overestimate the height of their respective sons. Consistency is preserved through the rules.

General trees: At least one node satisfies $\mathbf{delta}(n) < 0$. At least one father underestimates the height of one of its sons.

Defining the variants

We recall from section 2.4 the meaning of the following functions

$$\begin{aligned}\mathbf{inside}(n) &= \text{size of the subtree rooted at } n(\text{ including} n).\\ \mathbf{outside}(n) &= \text{number of nodes out of the subtree rooted at } n.\end{aligned}$$

As we pointed out above, the lack of stability splits into two different cases.

Consistent trees: All nodes satisfy $\mathbf{delta}(n) \geq 0$. Thus, all nodes overestimate the (apparent) height of their respective sons. Let us define variant functions to deal with this case.

Balance: We define the *global (apparent) balance B* as

$$\text{LBAL} = \sum_n |\mathbf{lbal}(n)|$$

If LBAL $= 0$, then the tree is balanced. Now, observe that on a left-equal single rotation, the global balance strictly betters *if we disregard balanced nodes*. We therefore define

$$\text{RLBAL} = \sum_{|\mathbf{lbal}(n)|>1} |\mathbf{lbal}(n)|$$

Positive unstability: The overestimation of the apparent height of the whole tree from the father's point of view is given by:

$$\text{EXCESS} = \sum_{\mathbf{delta}(n)>0} \mathbf{delta}(n)$$

Applying the rules sometimes result in nice interactions between LBAL and EXCESS. Let us sketch a case appearing in the propagation. Let n be the left son of m and assume that m overestimates the height of n (we mean $\mathbf{delta}(n) > 0$) and that the right subtree of m dominates the balance. As we will see in the following section, when propagation from n to m takes place, we get $\text{EXCESS}' = \text{EXCESS} - \mathbf{delta}(n)$ but $\text{LBAL}' = \text{LBAL} + \mathbf{delta}(n)$. The increase in stability (the variant EXCESS decreases by $\mathbf{delta}(n)$ units) has been compensated by a similar increase of the unbalance (LBAL increases by $\mathbf{delta}(n)$ units). The transformation of inconsistency into unbalance is acceptable if it occurs in a more or less similar amount. We therefore define a new variant combining balance and unstability with different weights:

$$\text{TRADE\_OFF} = \text{LBAL} + 2 \cdot \text{EXCESS}$$

In the preceding case variant TRADE_OFF strictly decreases because $\text{TRADE\_OFF}' = \text{TRADE\_OFF} - \mathbf{delta}(n)$. Let us define:

$$\boxed{\text{Variant for consistent trees: } (\text{TRADE\_OFF}, \text{RLBAL})}$$

In the following section we will prove:

Lemma 6.5 (Consistent Liveness) *For consistent trees, the variant* (TRADE_OFF, RLBAL) *strictly decreases for the lexicographic order on any rule application and it is is greater than* $(0, 0)$. *Therefore, if we start with a consistent tree, no infinite sequence of rule applications is possible.*

General trees: We now relax the hypothesis of consistency. Any father can have an over or under estimation of the apparent height of some of its sons. In addition to the phenomenon appearing in the consistent case (we will need all the preceding variants), we need to consider negative unstability.

Negative unstability: Recall that a node has negative unstability if its father underestimates its apparent height. Intuitively, *negative* unstability flows upwards in the tree and eventually reaches the root where it vanishes. The rotations help decreasing negative unstability, as they cannot increase the apparent height of subtrees. Upwards flowing can be captured by weighting the lack of stability with a measure of its depth within the tree. We define

$$\text{LOSS} = \sum_{\mathbf{delta}(n)<0} \mathbf{outside}(n).|\mathbf{delta}(n)|$$

The tree is consistent if LOSS = 0. Thus, LOSS is a measure of the inconsistency of the tree. This last fact allows us to reduce the liveness of general trees to the liveness of consistent trees.

Let us define:

$$\boxed{\text{Variant for general trees: } (\text{LOSS}, \text{TRADE\_OFF}, \text{RLBAL})}$$

In the following section we will prove:

Lemma 6.6 (General Liveness) *The variant* (LOSS, TRADE_OFF, RLBAL) *strictly decreases for the lexicographic order on any rule application and it is is greater than* $(0, 0, 0)$. *Therefore, no infinite sequence of rule applications is possible.*

Proof of Liveness

The following result is crucial for analyzing the various cases.

Lemma 6.7 *None of the quantities* **outside**, **delta**, **lbal** *is modified outside of the spatial scope of the rules.*

Consistent trees: Although all nodes verify $\mathbf{delta}(n) \geq 0$, we underline that the following analysis only takes into account the local properties of the nodes.

Propagation Rule: We consider the left case only. Consider a node n, its father m and m's father o. By the precondition of the Propagation Rule, $\mathbf{lheight}(n) < \mathbf{lefth}(m)$. On propagating the height of n, one modifies $\mathbf{delta}(n)$, $\mathbf{lefth}(m)$, $\mathbf{lheight}(m)$ and $\mathbf{delta}(m)$. The global effect of a propagation on LBAL and EXCESS depends therefore on the value of $\mathbf{delta}(m)$, and on the relative values of $\mathbf{lheight}(n) \leq \mathbf{lefth}(m)$ and $\mathbf{righth}(m)$. As the tree is consistent, we have $\mathbf{delta}(m) \geq 0$. The global positive unstability and the global unbalance progress in opposite directions. Remember that $\mathbf{lefth}(m') \leq \mathbf{lefth}(m)$ by hypothesis.

- If the left subtree was "very" dominant at m, then updating $\mathbf{lefth}(m)$ improves the balance $\mathbf{lbal}(m)$ but decreases the height of m. Thus $\mathbf{delta}(m)$ increases of the same quantity as $\mathbf{delta}(n)$ decreases. Thus, LBAL strictly decreases and EXCESS remains unchanged. Thus, TRADE_OFF strictly decreases.
- If the right subtree was dominant at m, then updating $\mathbf{lefth}(m)$ worsens the balance $\mathbf{lbal}(m)$ by $\mathbf{delta}(n)$, and the leaves the height unchanged. Thus, LBAL increases but EXCESS strictly decreases by the same quantity. Therefore TRADE_OFF strictly decreases.
- For intermediate cases, a detailed analysis shows that

$$\mathrm{LBAL}' = \mathrm{LBAL} + 2.\alpha - \mathbf{delta}(n) \qquad \mathrm{EXCESS}' = \mathrm{EXCESS} - \alpha$$

  for some α with $0 \leq \alpha \leq \mathbf{delta}(n)$, thus TRADE_OFF strictly decreases because $\mathrm{TRADE\_OFF}' = \mathrm{TRADE\_OFF} - \mathbf{delta}(n)$.

Rotation Rules: A careful analysis with $\mathbf{delta}(n) \geq 0$ shows the following global effect.

- Left-left rotation: $\mathrm{TRADE\_OFF}' \leq \mathrm{TRADE\_OFF} - 1$ because $\mathrm{LBAL}' \leq \mathrm{LBAL} - 3$ and $\mathrm{EXCESS}' = \mathrm{EXCESS} + 1$.
- Left-equal rotation: $\mathrm{TRADE\_OFF}' = \mathrm{TRADE\_OFF}$ because $\mathrm{LBAL}' = \mathrm{LBAL}$ and $\mathrm{EXCESS}' = \mathrm{EXCESS}$.
- Left-right rotation: $\mathrm{TRADE\_OFF}' \leq \mathrm{TRADE\_OFF} - 1$ because $\mathrm{LBAL}' \leq \mathrm{LBAL} - 3$ and $\mathrm{EXCESS}' = \mathrm{EXCESS} + 1$.

We can sum up all the analysis above by the following lemma.

Lemma 6.8 (Variant function) *For any rule application, one of these 2 cases holds:*

1. TRADE_OFF *strictly decreases;*
2. TRADE_OFF *remain unchanged: this occurs only in the case of a left-equal rotation. In this case* RLBAL *strictly decreases.*

Therefore, the variant function (TRADE_OFF, RLBAL) *strictly decreases for the lexicographic order on any rule application and it is is greater than* $(0, 0)$.

General trees: Consider a node n and its father m, let us make a case analysis based on its sign of **delta**(n). We assume that there are nodes with negative and positive unstability but the analysis will be done only for nodes of the first case. This is feasible because for nodes with positive unstability we repeat the analysis of the consistent trees: recall that we have only taken into account the local properties of the nodes, and that these local properties are unchanged in this kind of nodes. Now, we undertake a case analysis based on values of the **delta**(n) not yet considered in the consistent case.

Case delta$(n) < 0$.

As claimed above, this case is easier, as the Propagation Rule and the Rotation Rules cooperate. On applying the Propagation Rule, the negative unstability is moved one node upwards: the associated **outside** is thus strictly decreased. Thus LOSS$' \leq$ LOSS-1. On applying a Rotation Rule, the height of the subtree decreases, and the associated **delta** is thus decreased. The only case where there is no strict decrease is the Rotation Rule, left-equal case. Thus

- Left-left and left-right rotations: LOSS$' \leq$ LOSS $- 1$.
- Left-equal rotation: LOSS$'$ = LOSS and TRADE_OFF$'$ = TRADE_OFF.

Case delta$(n) \geq 0$ and delta$(m) < 0$.

A careful but tedious case analysis shows the following effect.

- If the left subtree of m is dominant, then the height of m decreases, and LOSS$' \leq$ LOSS $- 1$.
- If the right subtree is dominant, then **delta**(m) does not change: LOSS$'$ = LOSS. The unbalance at m gets worse: LBAL$'$ = LBAL + **delta**(n). But the global positive unstability improves: EXCESS$'$ = EXCESS $-$ **delta**(n). Therefore TRADE_OFF = TRADE_OFF $-$ **delta**(n).

We can sum up all the analysis above by the following lemma.

Lemma 6.9 (Variant function) *For any rule application, one of these 3 cases holds:*

1. LOSS *strictly decreases;*
2. LOSS *remains unchanged and* TRADE_OFF *strictly decreases;*
3. LOSS *and* TRADE_OFF *remain unchanged: this occurs only in the case of a left-equal rotation. In this case* RLBAL *strictly decreases.*

Therefore, the variant function (LOSS, TRADE_OFF, RLBAL) *strictly decreases for the lexicographic order on any rule application and it is is greater than* $(0, 0, 0)$.

This concludes the proof of Liveness.

It seems to be very difficult to give any general description of the successive "shapes" of the tree on the way towards balance. But we can at least give a bound on the register values as follows. Let **height**(n) be the *real height* of the subtree rooted at node n.

Lemma 6.10 *Let T be a distributed search tree such that* $\mathbf{lheight}(n) \leq \mathbf{height}(n)$ *for all nodes n in T. Let T' be obtained by any rule application. Then, this holds for T', too.*

This Lemma could be used to give a very rough bound on the number of steps needed to get an AVL tree from any search tree with N nodes. Just defines initially $\mathbf{lefth}(n) = \mathbf{righth}(n) = 0$ for all the nodes: it makes it a well-founded (but *not* consistent!) tree. We have: LOSS $\leq N^2$, EXCESS $= 0$, LBAL $= 0$, RLBAL $= 0$.

It is clear that our tree checks the condition of the lemma. Then, any tree derived from it do so. In particular, at any step, $\mathbf{lefth}(n), \mathbf{righth}(n) \leq \mathbf{lheight}(n) \leq \mathbf{height}(n) \leq N$ for any node. Thus LOSS $\leq N^3$, LBAL $\leq N^2$, EXCESS $\leq N^2$, RLBAL $\leq$ LBAL $\leq N^2$. The variant function can thus take at most $2.N^7$ values. This gives an obviously extremely rough bound of the actual number of steps needed to reach balance.

Exactly, N. Schabanel [48] decreases our theorical bound from this $O(n^7)$ to $O(n^2)$. Morever, he shows experimentally that the worst case self-balance is so improbable that the number of steps could decrease until $\Theta(n)$.

6.3 Concurrent insertion of keys

Using this basic framework, we can easily describe an algorithm to manage the concurrent insertion of keys in a distributed search tree. The idea is to simulate the percolation of keys in the original sequential algorithm with a new register $\mathbf{waiting}(n)$ which holds the keys waiting at node n for downwards percolation. To handle the possibility of equal keys, $\mathbf{waiting}(n)$ is managed as a *bag*. Operation $+$ adds a key to the bag. Operation $-$ removes it. This register is called the *waiting bag* at n. As usual, $|\mathbf{waiting}(n)|$ denotes the number of keys held in bag $\mathbf{waiting}(n)$.

We say that a distributed search tree is *strongly sorted* if the following condition holds: If n is in the left (resp. right) subtree of m, and $a \in \mathbf{waiting}(n)$, then $a \leq \mathbf{key}(m)$ (resp. $a \geq \mathbf{key}(m)$). As a simple example of such a tree, consider a single node n and N keys $a_1, \ldots, a_N$ with

$$\mathbf{key}(n) := a_1 \qquad \mathbf{waiting}(n) := \{a_2, \ldots, a_N\} \qquad \mathbf{lefth}(n) := 0 \qquad \mathbf{righth}(n) := 0$$

Rule : Percolation

Guard: *Node n, key $a \in$* $\mathbf{waiting}(n)$, $a \leq \mathbf{key}(n)$.

Behavior: *If n has a left son p, then*

$$\mathbf{waiting}(n) := \mathbf{waiting}(n) - a \qquad \mathbf{waiting}(p) := \mathbf{waiting}(p) + a$$

Otherwise, create a new node p, left son of n, with the following registers:

$$\mathbf{waiting}(n) := \mathbf{waiting}(n) - a \quad \mathbf{key}(p) := a \quad \mathbf{waiting}(p) := \emptyset$$

$$\mathbf{lefth}(p) := 0 \quad \mathbf{righth}(p) := 0$$

Spatial scope: *Node n and the potential new node p.*

Note: *Symmetrically with $a \geq$* $\mathbf{key}(n)$ *and node q the right son of n.*

It remains to refine the Propagation Rule and the Rotation Rules to take into account the waiting bags. The former causes no problem. But simply leaving the waiting bags hanging at the rotated nodes does not preserve strong sortedness in the latter. An easy solution is the following.

Rule : Rotation with waiting bags

> **Additional guard:** *A Rotation Rule may be applied only if the waiting bags at the nodes in the spatial scope of the rule are all empty.*

It is easy to check that the restricted version of Rotation Rules preserves strong sortedness. It is also clear that the global effects on LOSS, LBAL, EXCESS and RLBAL are not altered.

Lemma 6.11 *Let T be a distributed, well-founded and strongly sorted search tree. Let T' be the tree obtained by applying any rule. Then, T' is well-founded and strongly sorted, too.*

Lemma 6.12 (Safety) *Let T be a distributed, well-founded and strongly sorted search tree. Assume no rule applies to T. Then, T is an AVL tree whose all waiting bags are empty.*

Proof: *As the Percolation Rule does not apply, then all waiting bags are empty. As the Propagation Rule does not apply, all nodes are stable. As the Rotation Rules do not apply and all the waiting bags are empty, then all nodes are balanced.* ■

Lemma 6.13 *Let T be a distributed, well-founded and strongly sorted search tree. Let T' be the tree obtained by applying a Propagation Rule or a Rotation Rules. Then*

$$(\mathrm{LOSS}', \mathrm{TRADE\_OFF}', \mathrm{RLBAL}') < (\mathrm{LOSS}, \mathrm{TRADE\_OFF}, \mathrm{RLBAL})$$

.

To prove liveness, it remains to find an additional variant function to cover the two cases of the Percolation Rule. We introduce the following quantity: $\mathrm{IN}(n)$ is the number of nodes in the subtree rooted at n. Define first

$$\mathrm{NWBAG} = \sum_n 1$$

Quantity NWBAG is the current number of nodes of the tree. Define also

$$\mathrm{WAIT} = \sum_n \mathrm{IN}(n).|\mathbf{waiting}(n)|$$

Quantity WAIT is the current number of waiting keys, somehow weighted by their respective distances from the leaves. It is easy to check the that the first case of Percolation Rule lets NWBAG remain invariant and WAIT strictly decrease. Moreover, NWBAG and WAIT are invariant under the Propagation and Rotation rules (recall that a rotation may only take place if all bags in the spatial scope are empty). Also, the second case of Percolation Rule lets NWBAG strictly decrease. We can therefore define

$$\boxed{\text{Variant for general trees: } (\mathrm{NWBAG}, \mathrm{WAIT}, \mathrm{LOSS}, \mathrm{TRADE\_OFF}, \mathrm{RLBAL})}$$

Lemma 6.14 (Liveness) *The variant function* (NWBAG, WAIT, LOSS, TRADE_OFF, RLBAL) *strictly decreases on any rule application.*

6.4 Concurrent insertion and deletion of keys

In this section, we add the possibility to delete keys in a concurrent schema with the previous concurrent insertion algorithm. We start by letting the key to be deleted percolate down until the node with equal key is found and marked. Then, the node is sent down by successive rotations around it, until one of its sons, at least, gets empty. The node can then be safely removed from the tree.

The **waiting** bag contains now two kind of keys, the keys to be inserted and the keys to be deleted, which are differentiated by a flag. We need another flag, denoted $\mathbf{green}(n)$, to mark the nodes to be rotated down and removed. We say that a node is *alive* if $\mathbf{green}(n) = true$, otherwise it is said *dead*.

Let $\{a_1, a_2, \ldots, a_N\}$ the keys to be deleted. We start by locating them into the **waiting** bag of the root of the tree and by setting to true all the flags **green**. We modify the Percolation rule in order to consider the new kind of keys.

Rule : Percolation

Guard: *Node n, key $a \in \mathbf{waiting}(n)$ and should be deleted.*

Behavior: *a is extracted from $\mathbf{waiting}(n)$. If $a = \mathbf{key}(n)$ then $\mathbf{green}(n)$ becomes false. Assume that $a < \mathbf{key}(n)$ and n has left son p, then $\mathbf{waiting}(p) := \mathbf{waiting}(p) + a$, but if n has no left son then a is removed. The case $a > \mathbf{key}(n)$ is handed in the same way.*

Spatial scope: *Nodes n and p.*

The following rule deals with *dead* nodes: it sends down the marked nodes by doing rotations and eventually removes them. We underline that the rotations considered here are not the rules explained in previous sections, they are the obvious transformations that send down the root and leave invariant the order of keys.

Rule : Garbage Percolation

Guard: *Node n is dead. If both sons are empty then $\mathbf{waiting}(n)$ does not contains keys to be inserted, and if both sons are non empty then one of them, say p, at least, is alive, stable and $\mathbf{waiting}(p)$ is empty.*

Behavior: *Let m be the father of n.*

If both sons of n are empty then node n is removed.

If only one son of n is empty, then node n is removed and the other son p is attached to m and $\mathbf{waiting}(p') := \mathbf{waiting}(p) + \mathbf{waiting}(n)$.

If both sons are non empty and only the left son p is alive, stable and $\mathbf{waiting}(p) = \emptyset$, then a right rotation around n is made and $\mathbf{waiting}(p') := \mathbf{waiting}(m)$ and $\mathbf{waiting}(m') := \emptyset$.

If both sons are alive, stable and have an empty waiting bag a nondeterministic selection can be done between right and left rotations. The registers should then be updated as suggested in the previous paragraph.

Spatial scope: *Node n and its sons.*

Theorem 6.1 (Safety) *Let T be a distributed, sorted, well-founded search tree. Assume no rule applies to T. Then, T is an AVL tree whose all waiting bags are empty.*

We now turn to the Liveness property. Note that the WAIT function has been slightly modified by including the keys to be deleted.

Lemma 6.15 *The Percolation Rule strictly decreases function* WAIT.

Let $|T|$ be the number of initial items of the tree T. We define the variant function

$$\text{DEAD} = \sum_{\mathbf{green}(n)=\text{false}} \mathbf{inside}(n)$$

Lemma 6.16 *The Garbage Percolation Rule strictly decreases function* (WAIT, DEAD).

Proof: *Function* WAIT *decreases in the first two cases (one son, at least, of the dead node is empty) and remains unchanged in the last two cases.*

Function DEAD *strictly decreases if the node is removed. Assume that n is dead and that it is sent down by a rotation around itself and leaves its son p as the new root, then* WAIT(n) *strictly decreases because, at least, the node p, that is alive by the guard, has been subtracted.*

■

Before establishing the total correctness, we add to the rotation rules an important condition that must avoid to send up a dead node: the nodes involved should be alive. This condition determines the prior claim of the removing rule than the equilibrating or inserting rules. We define:

$$\boxed{\text{Variant for general trees: } (\text{WAIT}, \text{DEAD}, \text{LOSS}, \text{TRADE\_OFF}, \text{RLBAL})}$$

Theorem 6.2 (Total correctness) *The Propagation, Rotation, Percolation and Garbage Percolation Rules strictly decrease the function* (WAIT, DEAD, LOSS, TRADE_OFF, RLBAL). *There does not exist any infinite sequence of rule applications. One eventually gets an AVL tree.*

6.5 Discussion

The approach given here has a very high degree of concurrency and atomicity. We can use this fact to emulate some of the other existing algorithms based on local rules. Recall that an algorithm A can be emulated by an algorithm B if any rule of A can be simulated by a concatenation of a fixed and finite number of rules of B. We keep the discussion informal, but it could be rewritten formally with the notion of homomorphisms between models of parallel computation systems introduced by T. Kasai and R. Miller [24]. We will only take into account the reconstruction phase of the tree (when all the keys have been transformed into new leaves).

6.5.1 Sequential algorithm

In the original sequential algorithm, every node n holds a register $\mathbf{bal}(n)$ with values in $\{-1, 0, 1\}$. If n has left and right sons p and q, then it holds $\mathbf{bal}(n) = \mathbf{real\text{-}height}(q) - \mathbf{real\text{-}height}(p)$. Essentially, sequential insertion reconstructs the tree bottom up in order to maintain balances. Let us consider what happens with our approach. Assume that a new key has just been added to the tree at the end of the *percolation* process. We can apply the propagation rule bottom up changing the value of $\mathbf{bal}(n)$ along the nodes of the restructuring path (the path going from the new leaf to the last node of the insertion path with a non-zero balance [16]). When we arrive at the critical node (last node having non zero balance), only a rotation can be applied if ever necessary. Therefore, the distributed algorithm mimics a bottom-up version of the sequential one.

Let us consider the emulation of some distributed algorithms based on local rules [25, 36, 37, 30]. As we have seen in section 2 these algorithms are based on some new approximate notion of height. To approach the problem we introduce the *up apparent height*, written $\mathbf{up\text{-}height}(n)$ of a node n, defined as:

$$\mathbf{up\text{-}height}(n) = \mathbf{lheight}(n) + \mathbf{delta}(n)$$

Given a node n having left son p and right son q we have $\mathbf{up\text{-}height}(p) = \mathbf{lefth}(n)$ and $\mathbf{up\text{-}height}(q) = \mathbf{righth}(n)$. The apparent height $\mathbf{up\text{-}height}(n)$ give us the information known by the father of n about the height of its son n. This function behaves as an approximate height because it verifies:

$$\mathbf{up\text{-}height}(n) = 1 + \mathbf{delta}(n) + \max\{\mathbf{up\text{-}height}(p), \mathbf{up\text{-}height}(q)\}$$

Remark that $\mathbf{delta}(n) \in \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. Moreover the balance can be rewritten as:

$$\mathbf{lbal}(n) = \mathbf{up\text{-}height}(q) - \mathbf{up\text{-}height}(p)$$

6.5.2 J.L.W. Kessels' approach

Let us see how to emulate the algorithm [25]. We start giving a brief description of it. Every node n has the registers $\mathbf{bal}(n) \in \{-1, 0, 1\}$, $\mathbf{red}(n)$ (denoted **carry** by J.L.W. Kessels) and there is a dynamic height defined as:

$$\mathbf{dheight}(n) = 1 - \mathbf{red}(n) + \max\{\mathbf{dheight}(p), \mathbf{dheight}(q)\}$$

Once all the keys have been inserted, the tree can be highly unbalanced and the algorithm starts a *restructuring process*. Eventually, all nodes n satisfy $\mathbf{red}(n) = 0$. Then, $\mathbf{real\text{-}height}(n) = \mathbf{dheight}(n)$ and the tree is an AVL. This process is described by the transformations (a), (b) and (c) (Figures 1, 2 and 3 in [25]). All these transformations maintain the balance and the red constraints. The first one moves the red up when this fact does not unbalance the tree. The second one couples a red updating with a single rotation. The last one modifies the red adequately under double rotations. For instance the Transformation (b) can be rewritten as follows:

Rule : Transformation (b)

Guard: *Node n is a left son of m and* $\mathbf{bal}(n) < 0$, $\mathbf{red}(n) = 1$, $\mathbf{bal}(m) < 0$, $\mathbf{red}(m) = 0$.

Behavior: *A right rotation takes place (left-left case) and the new values of balance and carries are* $\mathbf{bal}(n') = \mathbf{bal}(m') = 0$ *and* $\mathbf{red}(n') = \mathbf{red}(m') = 0$.

Spatial Scope: *Nodes n and m.*

Note: *Symmetrically n is the right son of m.*

The comparison between $\mathbf{dheight}(n)$ and $\mathbf{up\text{-}height}(n)$ suggests us the identifications:

$$\mathbf{red}(n) = -\mathbf{delta}(n) \qquad \mathbf{dheight}(n) = \mathbf{up\text{-}height}(n)$$

Let us see how to emulate Transformations (a), (b) and (c) with our rules. All the cases given in (a) can be emulated by our propagation. The cases appearing in (b) can be emulated concatenating a propagation with a single rotation. Let us consider in more detail the five cases appearing in (c). The first three of them can be emulated with a propagation concatenated with a double rotation. The last two cases cannot be obtained in this way (because a red node is located in the wrong place), however our approach (concatenating a propagation with a double rotation) gives us alternative and equally correct solutions. As we identify $\mathbf{delta}(n) = -\mathbf{red}(n)$ and $\mathbf{red}(n) \in \{0, 1\}$ we get $\mathbf{delta}(n) \in \{0, -1\}$. Therefore any emulated transformation needs to maintain $\mathbf{delta}(n) \in \{0, -1\}$. Let us explain what it means. The condition $\mathbf{red}(n) \in \{0, 1\}$ holds before and after the application of any transformation. It does not make sense to take $\mathbf{red}(n)$ into account during the transformation (a transformation can be thought as instantaneous and without internal structure). Therefore, in our approach, the constraint $\mathbf{delta}(n) \in \{0, -1\}$ holds before and after the emulation of any rule. As any transformation is emulated by a concatenation of a small number of rules, the emulated transformation has an "internal structure" induced by this concatenation. We relax the constraint $\mathbf{delta}(n) \in \{0, -1\}$ during the emulation. For instance, in the emulation of b, just after the propagation and before a rotation we can have $\mathbf{delta}(n) = -2$. A rotation takes place precisely to correct this situation.

6.5.3 O. Nurmi et al.'s approach

Now we consider the following extension of the J.L.W. Kessels' algorithm developed by O. Nurmi, E. Soisalon-Soininen and D. Wood in [37] (see also [36]). Every node holds the registers $\mathbf{rbal}(n) \in \{-1, 0, 1\}$ and $\mathbf{tag}(n) \in \{-1, 0, 1, 2, \ldots\}$ and they define a relaxed height as:

$$\mathbf{rheight}(n) = 1 + \mathbf{tag}(n) + \max\{\mathbf{rheight}(p), \mathbf{rheight}(q)\}$$

As before, we consider only the restructuring process. Let n be a node with father m. The transformations assume $\mathbf{tag}(n) \neq 0$ and $\mathbf{tag}(m) = 0$ and have two phases. Phase 1 is designed to decrease the tag value of n. At the end of this phase, we can have $\mathbf{rbal}(m) \in \{-2, 2\}$. Phase 2 adjusts $\mathbf{rbal}(m)$ readjusting tags and making a rotation if necessary. The algorithm will eventually reach $\mathbf{tag}(n) = 0$. As they are local and need to maintain the relaxed balance, only small variations on the tag values can be correctly absorbed with rotations and tag adjustments. Therefore, the authors consider only a tag decrease of one unit, $\mathbf{tag}(n') = \mathbf{tag}(n) \pm 1$ (maintaining the constraint $\mathbf{tag}(n') \in \{-1, 0, 1, 2, \ldots\}$). This small decrease of the tag adds a level of complexity in the description of the transformations. Let us consider how to emulate this algorithm. The definitions of $\mathbf{rheight}(n)$ and $\mathbf{up\text{-}height}(n)$, suggest us the identifications:

$$\mathbf{rheight}(n) = \mathbf{up\text{-}height}(n) \qquad \mathbf{rbal}(n) = \mathbf{lbal}(n) \qquad \mathbf{tag}(n) = \mathbf{delta}(n)$$

A problem appears with these identifications. As tag values can only decrease in one unit, we are forced to have $\mathbf{delta}(n') = \mathbf{delta}(n) \pm 1$. However our Rule 1 allows node n to propagate all the information in only one step (for any value $\mathbf{delta}(n) \neq 0$ we have $\mathbf{delta}(n') = 0$). In order to accept small modifications like $\mathbf{delta}(n') = \mathbf{delta}(n) \pm 1$, we add a new private register $\mathbf{delta}(n)$. This register keeps the information to be transmitted to the father in order to update its knowledge about the height. In this way node n can transmit only a very small part of its information, keeping an account of the remaining one to be transmitted. The information about the height flows slowly along the tree and in a propagation we have $\textbf{up-height}(n') = \textbf{up-height}(n) \pm 1$. All our rules can be rewritten to take care of this fact. As a major level of detail has been added, rules are more complex. For instance, with a little bit of thought, we get the following new rule to propagate one unit of $\textbf{delta}(n)$ (we give the rule for both, left and right case because of the lack of symmetry in the evolution of balance field):

Rule : One unit propagation

Guard: *Node n is a left son of m and* $\textbf{delta}(n) \neq 0$.

Behavior: *When* $\textbf{delta}(n) > 0$ *we have* $\textbf{delta}(n') = \textbf{delta}(n) - 1$. *If n is a left son of m then* $\textbf{lefth}(m') = \textbf{lefth}(m) - 1$, $\textbf{delta}(m') = \textbf{delta}(m) + (\textbf{lbal}(m) < 0)?1 : 0$ *else n is a right son of m and* $\textbf{righth}(m') = \textbf{righth}(m) - 1$, $\textbf{delta}(m') = \textbf{delta}(m) + (\textbf{lbal}(m) > 0)?1 : 0$. *All other registers remain constant.*

When $\textbf{delta}(n) < 0$ *we have* $\textbf{delta}(n') = \textbf{delta}(n) + 1$. *If n is a left son of m then* $\textbf{lefth}(m') = \textbf{lefth}(m) + 1$, $\textbf{delta}(m') = \textbf{delta}(m) - (\textbf{lbal}(m) \leq 0)?1 : 0$ *otherwise n is a right son of m and* $\textbf{righth}(m') = \textbf{righth}(m) + 1$, $\textbf{delta}(m') = \textbf{delta}(m) - (\textbf{lbal}(m) \geq 0)?1 : 0$. *All other registers remain constant.*

Spatial Scope: *Nodes n and m.*

We can get (with patience!) new versions for rules dealing with rotations and double rotations which take into account the small variations of $\textbf{delta}(n)$. Using these new rules we can mimic the restructuring transformations given in [37] , by mainly coupling propagations and rotations.

6.5.4 K.L. Larsen's approach

Finally let us consider the approach taken by K. L. Larsen [30]. Recall that, given a node n having a father m, the transformations given by O. Nurmi, E. Soisalon-Soininen and D. Wood in [37] require precondition $\textbf{tag}(n) \neq 0$ and $\textbf{tag}(m) = 0$. This precondition has been relaxed by Larsen accepting nonzero values for $\textbf{tag}(m)$. To do it, K.L. Larsen modifies the preceding transformations to avoid the accumulation of negative values in $\textbf{tag}(m)$. As before he also couples a propagation with the rotation. More precisely, he defines 13 rules and he needs an scheduler to joint sequentially some of them. For instance, transformation 3 given in the Appendix of [30] is a propagation in the case $\textbf{delta}(n) = \textbf{tag}(n) = -1$, Transformation 4 is a propagation when $\textbf{delta}(n) = \textbf{tag}(n) = 1$ and Transformation 5 is a right rotation. As before we can emulate this approach.

Chapter 7

AVL trees: massively parallel algorithms

In this chapter we address the massively parallel algorithms on AVL trees. These algorithms are designed by coupling propagation and rotation rules, and by adding rules to address the parallel behaviour. The main drawback of this design that differentiates AVL trees from 2–3 trees and Skip trees is that leaves do not have the same depth and some rules modify the depth of some leaves. Then, it is not obvious that pipeline waves created at leaves with length of wave l yield a stationary travelling waves motion.

7.1 Introduction

The first massively parallel algorithm on search trees was first designed for 2–3 trees in 1983 by W. Paul, U. Vishkin and H. Wagener [43]. They gave effective parallel algorithms to maintain dynamically a parallel dictionary working on "batches" of k keys simultaneously. They designed pipelines to deal with massively parallel insertions and deletions. These pipelines can be explained intuitively in terms of traveling plane waves. Recall, for instance, the basic insertion case in which every leaf incorporates at most one new key, something like a *straight wave* (or *plane wave*) is generated at the bottom of the tree and is sent up in further steps. The wave goes to the root step by step increasing the height and decreasing the depth; it looks like a straight line going up in a picture of the tree. Formally, every pair of nodes p and q belonging to the wave front verify: $\mathbf{height}(p) = \mathbf{height}(q)$ and $\mathbf{depth}(p) = \mathbf{depth}(q)$. In the general insertion case, a pipeline of waves is generated to get something like harmonic traveling waves.

The main drawback in AVL trees is that in general only the following holds

$$\mathbf{depth}(n) + \mathbf{height}(n) \leq \mathbf{height}(\mathbf{root})$$

because the height of the root is determined by the deepest leaf. This fact means that this quantities cannot be taken into account to define the invariant of the waves.

We will say that n is *lower* if $\mathbf{lower}(n) \equiv (\mathbf{height}(n) < \mathbf{height}(\mathbf{brother}(n)))$ holds (we will identify true with 1 and false with 0). Finally, let us define a new depth so called *virtual depth* (see Figure 7.1) such that $\mathbf{vdepth}(root) = 0$ and $\mathbf{vdepth}(n) = 1 + \mathbf{lower}(n) + \mathbf{vdepth}(\mathbf{parent}(n))$. This new depth give us the invariant:

$$\mathbf{vdepth}(n) + \mathbf{height}(n) = \mathbf{height}(root) = \mathbf{vdepth}(nil).$$

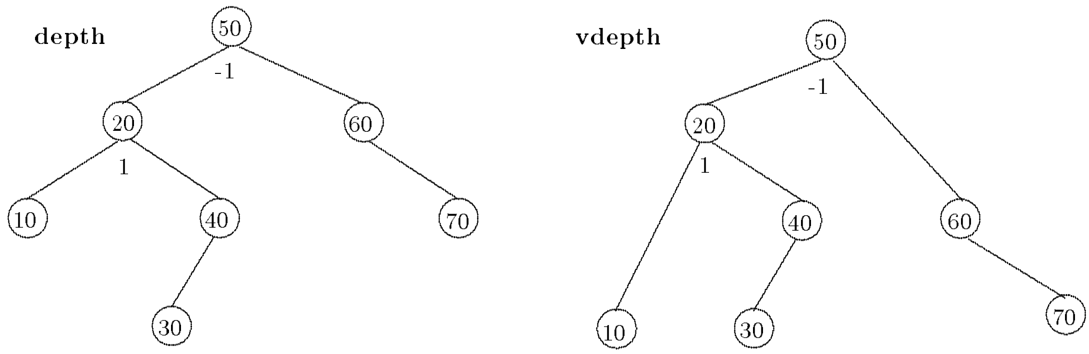


Figure 7.1: Meaning of the **depth** and the **vdepth**

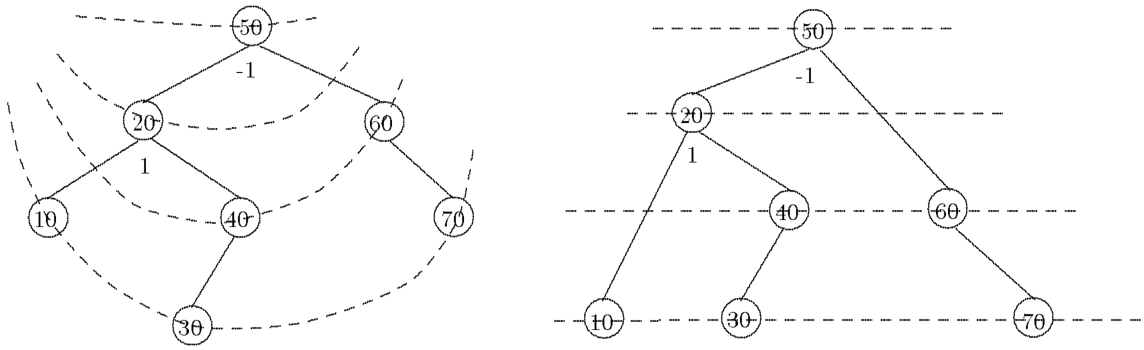


Figure 7.2: Virtual plane waves

Therefore it is possible to design *virtually plane waves* such that every pair of nodes p and q in to the wave front verify: $\mathbf{height}(p) = \mathbf{height}(q)$ and $\mathbf{vdepth}(p) = \mathbf{vdepth}(q)$ as in 2-3 trees. Based on the preceding ideas we can design a pipeline flowing bottom-up the tree. If we assume that information's flow does not modifies the AVL tree, we will call this tree *static* (in relation to the information) and we derive the following sheme:

PIPELINE SCHEME FOR STATIC AVL trees: using *virtual depth*, the leaves become aligned at the bottom of the AVL trees. It is possible to start and move up a *virtual plane wave*. When a wave moves up the *height* increases and *virtual depth* decreases. We can chain virtual plane waves to get a pipeline in a static AVL tree (see Figure 7.2).

The preceding scheme shows us how to pipeline information across AVL trees in a very regular fashion. This can be done chaining virtual plane waves. The preceding scheme assumes that the tree does not change while information flows up. A major problem occurs when the wave front dynamically perturbes the structure of the tree. This happens in insertions and deletions. When a wave front modifies the tree, succesive waves will be highly perturbed and collisions become possible. Therefore, to design massive parallel algorithms to deal with insertions or deletions is not straightforward.

7.2 Massively parallel insertions

Let us deal with parallel insertions. Following the idea of J.L.W. Kessels [25] we will assume that a newly inserted node does not count to compute the height, and as in previous chapters we colour red this kind of nodes.

7.2.1 Red relaxed AVL trees

We assume that the old nodes on AVL trees are whiten and the new ones to be inserted are red. Viewing nil nodes as nodes to by filled with new information, it makes sense to colour them red: $\mathbf{colour}(nil) = \mathbf{red}$. We define the predicates $\mathbf{red}(n) \equiv (\mathbf{colour}(n) = \mathbf{red})$ and $\mathbf{white}(n) \equiv (\mathbf{colour}(n) = \mathbf{white})$. Intuitively, red nodes mean two things, first they do not count to compute the height and second, they represent an unstable perturbation to be propagated up or erased as soon as possible. To develop the first idea we recall from [25] the definitions of the dynamic magnitude height, balance and lower of a node. The *dynamic height*, written $\mathbf{dheight}(n)$, is defined such that if n is a red leaf $\mathbf{dheight}(n) = 0$, if n is a white leaf $\mathbf{dheight}(n) = 1$, otherwise

$$\mathbf{dheight}(n) = \mathbf{white}(n) + \max(\mathbf{dheight}(\mathbf{left}(n)), \mathbf{dheight}(\mathbf{right}(n))).$$

To keep consistency we assume $\mathbf{dheight}(nil) = 0$. Based on this height, the *dynamic balance* of a node is defined as 0 when n is a leaf or *nil*, otherwise

$$\mathbf{dbal}(n) = \mathbf{dheight}(\mathbf{right}(n)) - \mathbf{dheight}(\mathbf{left}(n))$$

and the predicate lower can be redefined calling it *dynamic lower*, such that $\mathbf{dlower}(n) \equiv (\mathbf{dheight}(n) < \mathbf{dheight}(\mathbf{brother}(n)))$. A tree is called a *relaxed AVL tree with red nodes* (or a *red relaxed AVL tree*) when any node n verifies $\mathbf{dbal}(n) \in \{-1, 0, 1\}$, see [25]. Observe that when all the nodes are white, a red relaxed AVL tree is an AVL tree.

Let us modify the virtual depth in order to take care of the color. If a node does not count to compute the dynamic height, it seems obvious that it cannot count to compute the depth. We define the *virtual dynamic depth* of a node n, written as $\mathbf{vddepth}(n)$, in such a way that $\mathbf{vddepth}(\mathbf{root}) = 0$ and for any node, different from the root:

$$\mathbf{vddepth}(n) = \mathbf{white}(n) + \mathbf{dlower}(n) + \mathbf{vddepth}(\mathbf{parent}(n)).$$

The following easy lemma gives us a precise statement of the preceding intuitions.

Lemma 7.1 *In a* red relaxed AVL tree *any node n verifies*

$$\mathbf{dheight}(n) + \mathbf{vddepth}(n) + \mathbf{red}(n) = \mathbf{dheight}(root) + \mathbf{red}(\mathbf{root}).$$

All the nil nodes have the same virtual dynamic depth *because*

$$\mathbf{dheight}(\mathbf{root}) + \mathbf{red}(\mathbf{root}) = 1 + \mathbf{vddepth}(nil).$$

Based on the preceding lemma we come to our second design scheme. It will allow us to deal with waves perturbing the data structure dynamically. The red color and the dynamic magnitudes will allow us to maintain the virtual plane character of the waves.

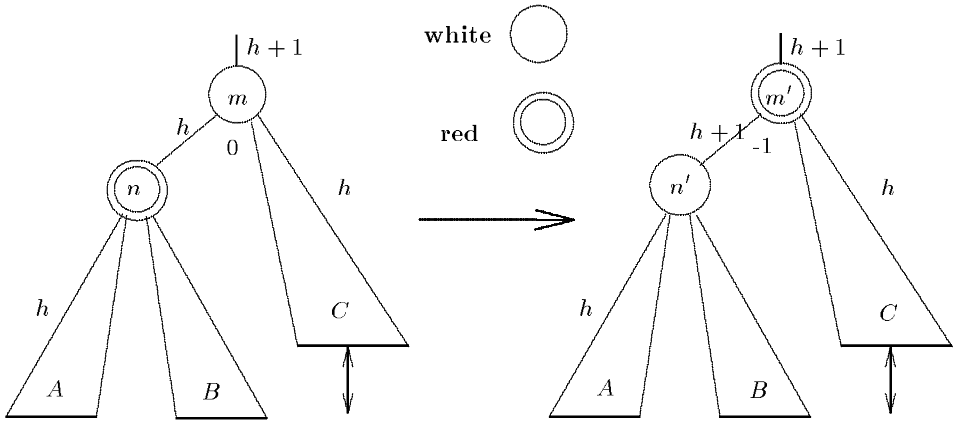


Figure 7.3: **Red Propagation** rule

RED DYNAMIC PIPELINE SCHEME: Using *virtual dynamic depth*, the leaves become aligned at the bottom of a red relaxed AVL tree. Thus it is possible to start and move up a *virtual dynamic plane wave* (also called *virtual plane waves* or *plane waves* or just *waves*). When the wave moves up the *dynamic height* increases and the *virtual dynamic depth* decreases. We can chain waves to get a pipeline in red relaxed AVL trees.

7.2.2 Local rules for insertions

We recall the local rules extracted from the sequential insertion algorithm and we extend this set of rules to take care of parallelism and of prevention of deadlock.

The rules are an extension (induced by the parallel case) of those given by J.L.W. Kessels [25]. Essentially, the rules move red nodes up (maintaining the dynamic height and balance) until they come to the root and disappear transforming a red relaxed AVL tree into an AVL tree. We only give complete versions of the most important rules, the others will be sketched. We adopt the following notation: $n(A, B)$ denotes the (sub)tree with root n, left son A and right son B. By convention, the final state of a node n is denoted as n'.

First, let us recall the rules issues from the sequential case. In this case *just one* red node moves up the color. There are two cases (propagations or rotations) depending on the dynamic balance of the father. Figure 7.3 deals with one of the cases of the following propagation rule. We have wrote the height of each node to underline that red nodes (two concentric circles) are not took into account. We recall from page 22 or from page 32

Rule : Red Propagation

> **Additional Guard:** *A subtree* $m(n(A, B), C)$ *such that* m *is white,* n *is red with a white son and* $\mathbf{dbal}(m) \in \{0, 1\}$.

We need a rule to deal with a red root

Rule : Whitening red root (Page 22)

This rule colours the root white (increasing the dynamic height by one). There is another class of rules

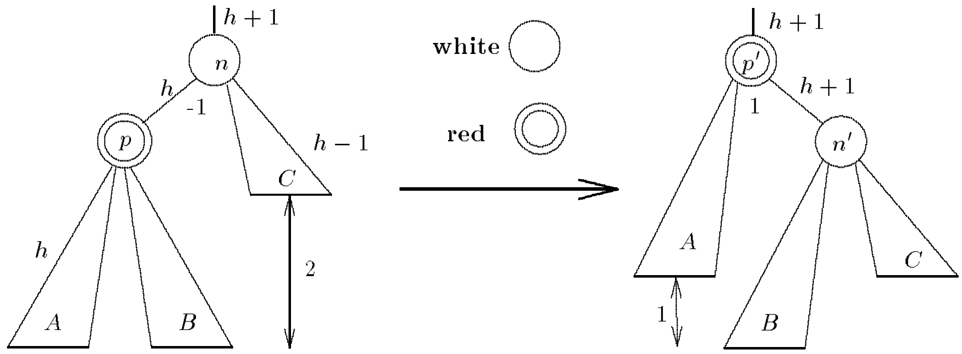


Figure 7.4: **Red Rebalancing** rule (single left-equal case)

Rule : Red Rebalancing (Page 33)

coupling the propagation of a red propagation with a usual rotation in order to keep the relaxed nodes balanced. As usual we have a single and double version.

Let us consider now the rules issues from the parallel case. These rules extend those given by J.L.W. Kessels [25]. The first case happens in a subtree $n(p(A, B), q(C, D))$ when both sons of n are red (this case cannot appear with only one key to be inserted).

Rule : Parallel Red Propagation

> **Guard:** *A subtree $n(p(A, B), q(C, D))$ such that n is white and its sons p and q are red. There are no conditions on the dynamic balances.*
>
> **Behavior:** *The dynamic balance of the nodes remain unchanged and:*
>
> $$\mathbf{colour}(n') := \mathbf{red} \quad \mathbf{colour}(p') := \mathbf{white} \quad \mathbf{colour}(q') := \mathbf{white}$$
>
> **Spatial scope:** *Nodes n, p and q.*

As the preceding propagation allows us to have red balanced nodes, the set of rotations have to be extended outside of J.L.W. Kessels. Then

Rule : Red Rebalancing

should be extended. Figure 7.4 explains one of the two symmetric cases: $\mathbf{dbal}(n) = -1$ and $\mathbf{dbal}(p) = 0$.

All the preceding rules have been designed to obtain the following lemmas. If all the rules were given explicitly, both could be proved by (lenghtly) straightforward case analysis.

Lemma 7.2 *Any application of a rule, keeps the tree dynamically balanced and*

(1) a propagation rule verifies $\mathbf{dheight}(m') = \mathbf{dheight}(m)$,

(2) a colored rebalancing maintains the dynamic height of the subtree, for instance in single right rotation $\mathbf{dheight}(p') = \mathbf{dheight}(n)$,

(3) *when the root changes from red to white it holds* $\mathbf{dheight}(\mathbf{root}') = 1+\mathbf{dheight}(\mathbf{root})$.

White nodes can be seen as stable nodes while red nodes are like bubbles going up. It seems to bre a good idea to "give the control" to the white nodes and call a node *n active* if it is white but has at least a red son.

Lemma 7.3 *Let n be an active node in a red relaxed AVL tree.*

(1) *Any propagation rule keeps constant the dynamic balanced depth of any node different from n and* $\mathbf{vddepth}(n') + \mathbf{red}(n') = \mathbf{vddepth}(n)$.

(2) *Any colored rebalancing can only modify the dynamic balanced depth of nodes in its scope. If r' is the new root of the rotated subtree we have* $\mathbf{vddepth}(r') + \mathbf{red}(r') = \mathbf{vddepth}(n)$.

7.2.3 Algorithms for insertions

We will take the usual set of rules and apply them in a synchronized way giving a PRAM approach close to that of W. Paul, U. Vishkin and H. Wagener [43] for 2–3 trees. It is based on our second design scheme. We consider three cases.

(1) Assume all the keys to be inserted are attached to a white node.More formally, any red node (different from nil) with dynamic height 0 has a white father. The massive parallel algorithm can be sketched as follows. While there are active nodes,

- all active nodes with dynamic height 1, applies to each node the corresponding rule,
- all active nodes with dynamic height 2, applies to each one the corresponding rule,
- iterate, increasing the dynamic height until it becomes the dynamic height of the tree,

if the root becomes red, white it and increase the dynamic height of the tree. In this algorithm, we can see the set of active nodes as a unique wave going bottom-up the tree. The following lemma considers if this wave is "plane" going up.

Lemma 7.4 *The nodes that belong to the front wave have the same* virtual dynamic depth.

Proof: *Observe that in the preceding algorithm two active nodes p and q belonging to the front wave verify* $\mathbf{dheight}(p) = \mathbf{dheight}(q)$. *Since throughout the whole algorithm the tree is a red relaxed AVL tree, for any node n it follows that*

$$\mathbf{dheight}(n) + \mathbf{vddepth}(n) + \mathbf{red}(n) = \mathbf{dheight}(root) + \mathbf{red}(\mathbf{root}).$$

Using that any active node n in the wave is white and also is the root (except perhaps at the very last step), any node in the front wave verifies

$$\mathbf{dheight}(n) + \mathbf{vddepth}(n) = \mathbf{dheight}(root).$$

■

Therefore, the expression *virtual red plane wave* makes sense because the wave behaves as a plane wave using the dynamic height and the virtual dynamic depth. As all the active white nodes have the same dynamic height and dynamic balanced depth it makes sense to assign a dynamic height and a dynamic balanced depth to the virtual plane wave w written $\mathbf{dheight}(w)$ and $\mathbf{vddepth}(w)$.

(2) Let us consider the case of an AVL tree with a "red barbe on the bottom". By this we mean that all the red nodes, containing a new key to be inserted, are at the bottom of the tree and all the other nodes are white. To solve this case we would like to pipeline different virtual plane waves. To do this we need to prove that two different waves do not collide if they are initially separated from one another. Let us prove that, any virtual plane wave will not be affected by the behaviour of other waves higher in the tree. It holds:

Theorem 7.1 *Take a relaxed AVL tree having red nodes at the bottom. If we start from the bottom a virtual plane wave w moving up and, at least, three steps later we start another one w' moving up at the same rate, the wave w' will remain virtually plane and moreover*

$$\mathbf{dheight}(w) - \mathbf{dheight}(w') = \mathbf{vddepth}(w') - \mathbf{vddepth}(w) = \lambda$$

while w and w' are moving up.

(3) Finally let us consider the case where we have an AVL tree with n keys and we have to insert an ordered set of k new keys stored in the array $a[1..k]$. We recall the search algorithm by packet routing [43, 22, 19]. Moreover, as this search algorithm in a binary search tree is formally identical to a search by packet routing on Skip lists, the results about read conflicts [19] apply directly and we have that at most three subpackets can remain on a node. When the subpackets are located at the bottom of the AVL tree, the divide and conquer approach given in [43] allows us to start a pipeline and the results of (2) assure that

Theorem 7.2 *The massively parallel insertion of k keys into a red relaxed AVL tree tree with n keys takes time $O(\log n + \log k)$ using k processors on an EREW PRAM.*

7.3 Massively parallel deletions

When some nodes are deleted the tree can become lower. In order to maintain the dynamic magnitudes we introduce *green nodes* which count twice (green nodes are a complementary notion of red nodes). We assume that the set of keys to be deleted does not contain repetitions and any key to be deleted really appears in the tree. Let us sketch the version of the sequential deletion algorithm we will use.

- Request phase: A *request to delete* the key k is introduced at the root and percolated down through the tree until it stops at a node n at the bottom of the tree. If $k \leq \mathbf{key}(n)$ the value k is stored in the so called *left request register* otherwise it is stored in the *right request register*. A *request manager* of n (its father or grandfather) will take care of this request. Depending on the cases, the node n will be deleted, a *green node* will be generated (to maintain the AVL character), an *update set* written $\{\mathbf{key}(n) := k\}$, meaning "the value $\mathbf{key}(n)$ will substitute k on the tree" will be generated.

- Balance and update phase: a green node will move up the tree until it disappears or comes to the root. Update comes up until it arrives at a node holding the key k and substitutes k by $\mathbf{key}(n)$.

Based on these ideas, we develop green relaxed AVL tree. Later on we sketch a set of local rules maintaining the relaxed balance and finally we sketch massively parallel algorithm.

7.3.1 Green relaxed AVL trees

The deletion algorithm starts by percolating down any deletion request. Therefore we assume that any node n has two new registers called **left request**(n) and **right request**(n). At the end of this percolation phase any deletion request will be located at one request register. The following easy property holds,

Lemma 7.5 *If the set of keys to be deleted does not contain any repetition and all these keys belong to the tree, any leaf n will contain at most two deletion requests: one located at* **left request**(n) *and the second in* **right request**(n).

If a leaf n has a request we will say that n is the *request holder* of this request. Moreover if n has a father m, this father will we charged to deal initially with this request, and we will say that m is the *request manager* of this request. As before we will color the nodes of the tree to get a dynamic balance. The strategy is just the opposite of the insertion case. For instance, when a leaf p is deleted in the subtree $n(p, nil)$, it becomes just a leaf n', therefore node n' should *count twice* in order to maintain the height. To take care of this fact, we color n' green. Therefore, nodes can be white or green, and green nodes counts twice. In deletions, the *dynamic height* of a node, is given by $\mathbf{dheight}(n) = 2$ if n is a green leaf, $\mathbf{dheight}(n) = 1$ if n is a white leaf, otherwise

$$\mathbf{dheight}(n) = 1 + \mathbf{green}(n) + \max(\mathbf{dheight}(\mathbf{left}(n)), \mathbf{dheight}(\mathbf{right}(n))).$$

We define $\mathbf{dheight}(nil) = 0$ and $\mathbf{colour}(nil) = \mathbf{white}$, however in deletions nil nodes do not play an important role. Of course, all the request holders and all the request managers appear at the bottom of the tree. Formally, any *request holder* has dynamic height 1 or 2 and *request managers* have dynamic height 2 or 3.

As we said before, we need one more register. For instance, assume we have a subtree $n(p, q)$ and at the very bottom of the tree leaves p and q. Further suppose that p, with key k, holds only a right request d, meaning "the key d has to be deleted", written $\mathbf{right\ request}(q) = d$. In this case following the ideas sketched in the sequential algorithm, the key k will be used to substitute the key d upward in the tree. Therefore, the algorithm deletes node q and the tree evolves to $n'(p', nil)$ and node n' stores the set $\{d := k\}$ meaning "the key k will substitute the key d upwards in the tree". When updates go up they can be pack together, we allow a new register $\mathbf{update}(n) = \{d_1 := k_1, d_2 := k_2, \ldots\}$.

The *virtual dynamic depth* of a node n is given by $\mathbf{vddepth}(\mathbf{root}) = 0$ and for any node, different from the root:

$$\mathbf{vddepth}(n) = 1 + \mathbf{green}(n) + \mathbf{lower}(n) + \mathbf{vddepth}(\mathbf{parent}(n)).$$

Lemma 7.6 *In a* green relaxed AVL tree *any node n verifies*

$$\mathbf{dheight}(n) + \mathbf{vddepth}(n) - \mathbf{green}(n) = \mathbf{dheight}(root) - \mathbf{red}(\mathbf{root}).$$

All the nil nodes have the same dynamic balanced depth *because*

$$\mathbf{dheight}(\mathbf{root}) - \mathbf{green}(\mathbf{root}) = \mathbf{vddepth}(nil).$$

As in the case of red relaxed AVL trees we can introduce a pipeline scheme for this kind of tree based on dynamic magnitudes.

GREEN DYNAMIC PIPELINE SCHEME: With the *virtual dynamic depth* we can pipeline *virtual waves* in green relaxed AVL trees.

7.3.2 Local rules for deletions

Let us *sketch* the set of rules to deal with deletions. Recall that we need rules to take into account three concepts: requests, green color and update sets. As in the insertions case rules keep the tree dynamically balanced. First, let us briefly consider the rule

Rule : Parallel Request Manager

acting on request managers located at (dynamic) height 2 or 3. This rule processes only one request at any application. If necessary, the manager:

- colours green a node at the bottom of the tree in order to maintain the dynamic height(Figure 7.5.d),
- generates a new update set $(d, k))$ (Figure 7.5.e),
- rotates in order to maintain the red relaxed AVL character. (Figure 7.5.e).

To reduce the number of cases, if both sons have requests we always give priority to the left son over the right one. Figures 7.5.a), b) and c) consider in detail the case of request manager of height 2 with $\mathbf{dbal}(n) = 0$, and p and q leaves. A request manager of dynamic height 3 (Figure 7.5d) is considered if their sons have no request.
The following rule forces the update sets to move up and, in some cases to replace the key to be deleted by the new key. This rule does not change the "topology" of the AVL tree.

Rule : Parallel Update

Guard: *A subtree $n(p(A, B), q(C, D))$ all three nodes have an update set. Nodes can have any color (white or green), can have requests and some of them can be leaves.*

Behavior: *The tree evolves to $n'(p'(A, B), q'(C, D))$. The nodes p', q' are equal to p, q' without updates. The nodes n and n' have the same color. As the update of p has the form $\{n := k\}$, therefore $(\mathbf{key}(n'), k)$ and $\mathbf{update}(n') = \mathbf{update}(n) \cup \mathbf{update}(q)$.*

Spatial scope: *Nodes n, p, q.*

Note: *All other cases are quite similar and easy.*

We recall the rule

Rule : Green Propagation

from page 25 that forces the green nodes to move up. As before, this rule does not change the "topology" of the AVL tree. Note the difference of guards between green and red propagation.

Finally we develop **green rebalancing** rule. (Figure 7.6). As before, it is the complementary notion of **red rebalancing** rule.

Rule : Green Rebalancing

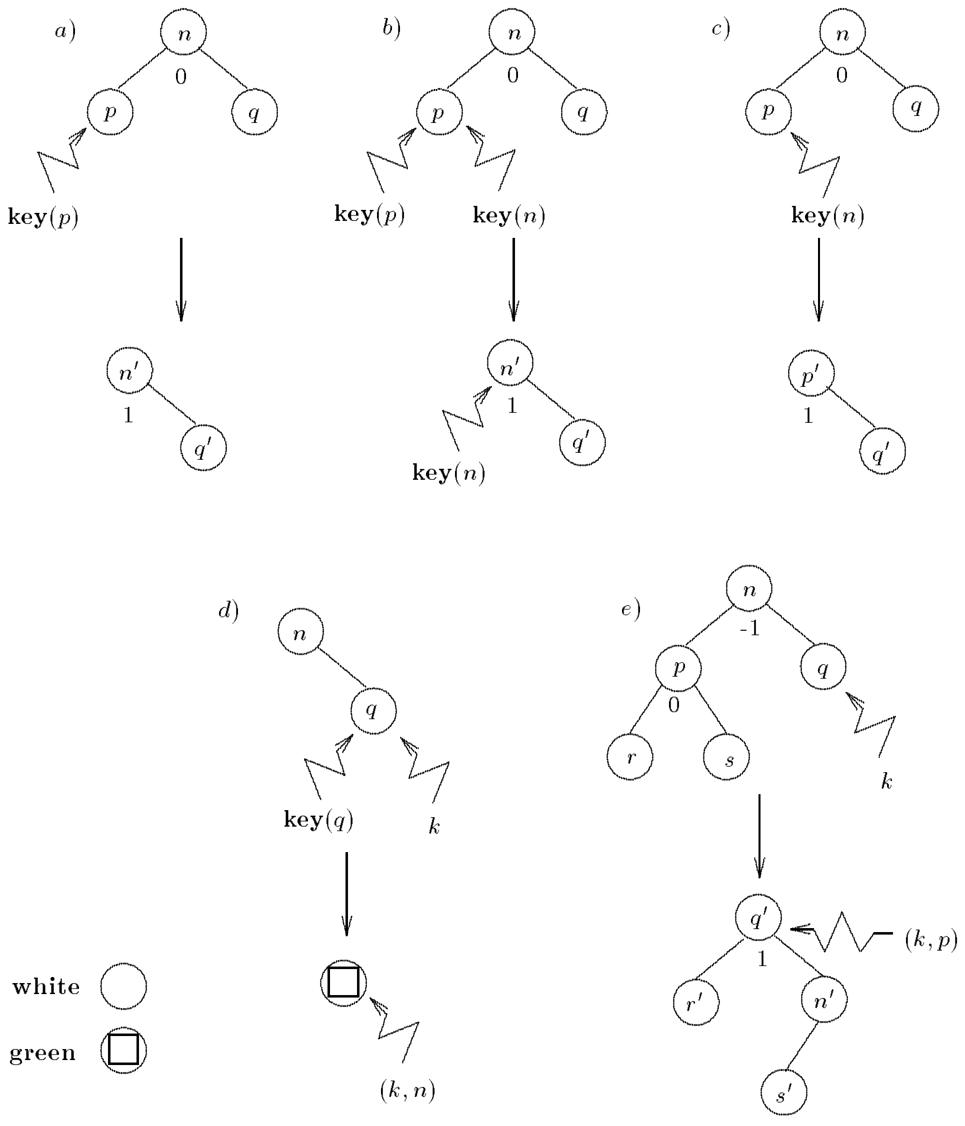


Figure 7.5: **Parallel request manager** rule

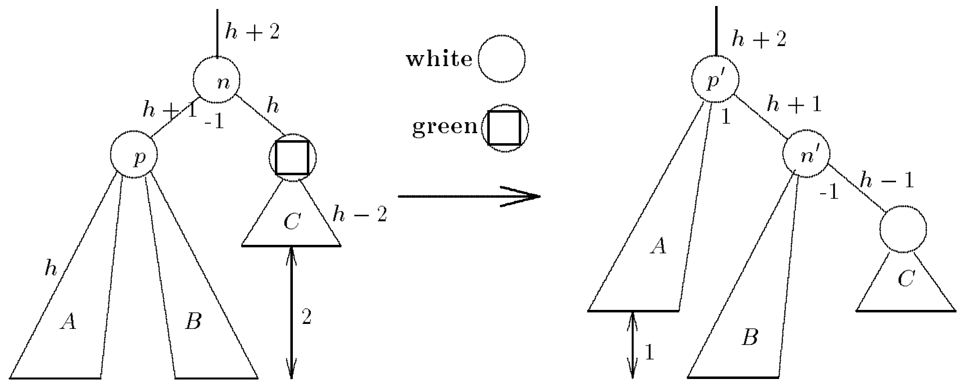


Figure 7.6: **Green Rebalancing** rule

Guard: *A subtree* $n(p(A,B),C)$, $\mathbf{dbal}(n) = -1$, $\mathbf{dbal}(p) \in \{0,-1\}$ *the node* n, p *are white and the root of* C *is green. Neither* n, p, *or the root of* C *have updates.*

Behavior: *Restructure the tree into* $p'(A, n'(B,C))$ *with the usual updating for keys and left and right pointers. If* $\mathbf{dbal}(p) = 0$, *the nodes* n', p' *and the root of* C *becomes white*

$$\mathbf{dbal}(p') := 1 \quad \mathbf{dbal}(n') := -1$$

Otherwise $\mathbf{dbal}(p) = -1$, *nodes* n', p' *become balanced, the root of* C *becomes white and*

$$\mathbf{colour}(p') := \mathbf{green} \quad \mathbf{colour}(n') := \mathbf{white}$$

Spatial scope: *Node* n *and its left son* p.

Note: *The symmetrical cases for single rotation and double green colored rotations also exist.*

7.3.3 Algorithms for deletions

Assume that we have k keys to be deleted, all of them in the tree, with no repetitions. As in the insertion case we generate waves. Here we have two different types of waves:

- *Update-wave* dealing with the update sets. This wave is charged to move up the update sets and to replace the deleted keys by the new ones. This wave is generated applying the request manager rule and moves up with the update rule.
- *Green-wave* moving up green nodes. It is also generated by the manager rule. Moves up by propagation and rotation rules. This green-wave moves up applying green propagation and rotation rules. A node is active in an *green-wave* if it has a green son.

The algorithm works in phases. In each one the following happens: all request managers of dynamic height 2 start up a request. Green nodes and update sets appear. An *update-wave* starts going up and a “little bit” later a green-wave goes up. A “little bit” later the entire scheme is repeated with the request managers having dynamic height 3.

Note that a request can give us a maximum of one component in an update set an at most one green node, therefore it is possible to implement the preceding phase with at most $2k$ processors, but it is really easy to decrease this to k processors. Moreover waves look like straight waves using dynamic balanced depth. As in a phase the number of requests decreases by a constant fraction; we can pipeline phases to get:

Theorem 7.3 *The massively parallel deletion of k keys into an AVL tree tree with n keys takes time $O(\log n + \log k)$ using k processors on an EREW PRAM.*

Chapter 8

Fringe analysis on parallel 2–3 trees

The last chapter of the dissertation is devoted to an introduction of analysis of search trees under massively parallel insertions.

8.1 Introduction

Massively parallel insertion algorithm on search trees is composed of two steps: the first one which routes packets of keys down and the second one which generates a pipeline of waves. It has been proved that the time spent to insert k keys with k processors into a tree of size n is $O(\log n + \log k)$.

Each wave is created and sent up as follows. Once the packets have been routed, they are attached to leaves. Then the middle key of each packet is selected and stored into a new leaf which is coloured red. The set of new red leaves is viewed as a red wave which is sent up until they disappear or reach the root. This bottom-up process can be viewed as a travelling wave. At each step the red nodes are handled with the corresponding balancing rule, namely **Parallel Red Split** (see page 35) in case of 2–3 trees, or propagations and rotations (see page 106) in case of AVL trees. Note that $O(\log k)$ waves are created, and that the time spent by each travelling wave is $O(\log n)$ because the time spent at each step is constant.

The performance analysis of this pipeline insertion algorithm is very difficult. For this reason we propose the following modification: the creation of only one wave which manages all the keys of packets. For instance, on 2–3 trees we use the rule **Red Split** (see Figure 5.4 on page 80) in place of rule **Parallel Red Split** (see Figure 3.2 on page 35). Note that this modification implies that the time spent at each wave step becomes linear on k.

The substitution of the pipeline-waves analysis for the one-wave analysis implies the usage of new rules which should mimic previous rules. Conversely, we can analyze only those magnitudes which new rules emulate accuratement, as, for instance, the distribution of leaves. This distribution is determined by rule **Red Split**, but this rule can be modified, for instance, to create the minimum number of single nodes or the maximum number (see Figure 8.1). Then the two distributions created by these cases bound the distribution created by pipeline algorithm.

In the following sections we develop the analysis of the evolution of leaves distribution of 2–3 trees under one-wave parallel insertions, called **fringe analysis** [4]. Our analysis can be applied to AVL trees with minor changes.

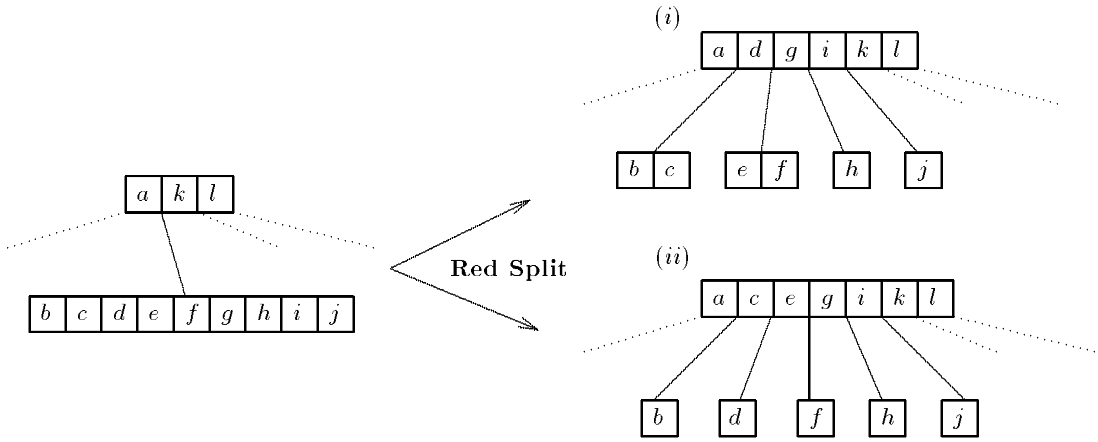


Figure 8.1: **Red Split** rules which create the minimum (i) or the maximum (ii) number of single nodes

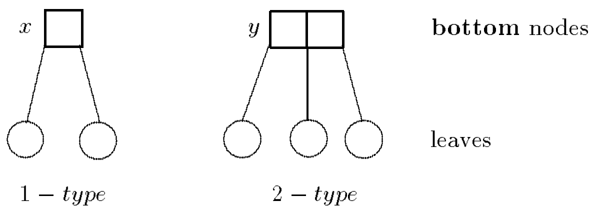


Figure 8.2: x and y nodes of 2–3 trees.

8.2 Overview

The 2–3 trees have been introduced in section 2.3 (see Figure 2.1 on page 16). They are search trees such that all leaves appear at the same depth and every node has either one key and two sons, or two keys and three sons. The first node type is designated x node, and the second one y node (see Figure 8.2). Note that bottom nodes separate leaves into $1-type$ leaves if their parents are x nodes; otherwise, $2-type$ leaves.

We denote k-random tree to those trees which have been generated randomly in parallel by inserting k keys at each stage. Take a k-random tree with n keys; let X_n be the expected number of $1{-}type$ leaves and Y_n the expected number of $2{-}type$ leaves, then $X_n{+}Y_n = n{+}1$. We define the probability that a random chosen leaf belongs to the $i-type$ as

$$P_n^i = \frac{\text{Expected number of leaves of } i-type}{n+1}.$$

Assume that k keys are in a random parallel manner inserted into a k-random tree with n keys. The random variable X_{n+k} (also denoted $X_{n,k}$) depends on X_n and Y_n only. This means that the current value depends on the history of the process only through the most recent value. Therefore we deal with a Markov chain and the evolution can be analysed through the matrix transition probabilities $T_{n,k}$ such that $\overrightarrow{P_{n,k}} = T_{n,k}\overrightarrow{P_{n,0}}$.

k	x node	y node
1	y	xx
2	xx	xy
3	xy	xxx or yy
4	xxx or yy	xxy
5	xxy	$xxxx$ or xyy
6	$xxxx$ or xyy	$xxxy$ or yyy

Table 8.1: Transformation of x and y bottom node once k keys reach them

n	$k=1$	$k=2$	$k=3$
1	1	1	1
2	0		
3	1	1	
4	0.4		0.4
5	0.6	0.5	
6	0.5714=4/7		
7	"	0.5625	0.466
8	"		
9	"	0.5609	
10	"		0.5394
31	"	0.5689	0.5656
61	"	0.5702	0.5687
91	"	0.5706	0.5697

Table 8.2: Probability of $1-type$ leaves once k keys have been inserted repeatedly

The transition matrix is computed by considering the equallylikely distribution of keys and the transformation of the bottom nodes. Let us explain this last point. Assume that k keys have been inserted, then, at most, k keys can reach a node. If the node becomes red, *i.e.* it stores more that two keys, it must be split. Table 8.1 shows some splits of x and y nodes; for instance, the first row shows the x node transformation into an y and the y node transformation into xx nodes under the one key insertion, and the fourth row shows how x and y nodes with new four new keys can be split (in some cases there are different possibilities). Note that y node transformation when k keys reach it is the same as the x node transformation when $k+1$ keys reach it.

The sequential case $k=1$ has been widely analysed [4, 5, 15]. The transition matrix is

$$T_{n,1} = I + \frac{1}{n+2}\begin{pmatrix} -3 & 4 \\ 3 & -4 \end{pmatrix} \qquad (I \text{ is the identity matrix})$$

and it implies the stationary values of the probability for $P_1^1 = 4/7$ and $P_1^2 = 3/7$ and $n \geq 6$ (see first column of Table 8.2).

The parallel case for two and three keys, studied in section 8.3, define the following transition probabilities

$$T_{n,2} = I + \frac{2}{n+3}\begin{pmatrix}-3 & 4\\ 3 & -4\end{pmatrix} + \frac{1}{(n+3)(n+1)}\begin{pmatrix}12 & -18\\ -12 & 18\end{pmatrix}$$

$$T_{n,3} = I + \frac{3}{n+4}\begin{pmatrix}-3 & 4\\ 3 & -4\end{pmatrix} + \frac{3}{(n+4)(n+1)}\begin{pmatrix}12 & -18\\ -12 & 18\end{pmatrix} + \frac{1}{(n+4)(n+1)^2}\begin{pmatrix}-48 & 54\\ 48 & -54\end{pmatrix}$$

Recall that the y node transformation under the insertion of three keys has two possibilities, and we have developed the second one (y, y). The columns of Table 8.2 show the evolution of the probalility values of x nodes (the initial tree had one x node). Note that these values rapidly converge to 4/7, therefore this value seem to be an **attractor** in the parallel case. The same table shows that the parallel insertion determines a leaves distribution different than those determined by sequential insertions. Let $T^{seq}_{n,k}$ be the transition matrix of the k keys sequential insertion, then

$$T^{seq}_{n,k} = T_{n+k-1,1} \circ T_{n+k-2,1} \circ \ldots \circ T_{n,1}$$

For $k = 2, 3$ it yields

$$T^{seq}_{n,2} = I + \frac{2}{n+3}\begin{pmatrix}-3 & 4\\ 3 & -4\end{pmatrix} + \frac{6}{(n+3)(n+2)}\begin{pmatrix}3 & -4\\ -3 & 4\end{pmatrix}$$

$$T^{seq}_{n,3} = I + \frac{3}{n+4}\begin{pmatrix}-3 & 4\\ 3 & -4\end{pmatrix} + \frac{18}{(n+4)(n+3)}\begin{pmatrix}3 & -4\\ -3 & 4\end{pmatrix} + \frac{30}{(n+4)(n+3)(n+2)}\begin{pmatrix}-3 & 4\\ 3 & -4\end{pmatrix}$$

The comparison of $T_{n,k}$ with $T^{seq}_{n,k}$ shows that they coincide in the first two terms and differs slightly in the remainder terms; something like $T_{n,k} = T^{seq}_{n,k} + O(1/n^2)$.

The transition probabilities for cases $k = 1, 2, 3$ suggest us to conjecture that the transition matrix for parallel insertion of k keys is

$$T_{n,k} = I + \frac{1}{n+k+1}\sum_{j=1}^{k}\frac{(-1)^{j-1}\binom{k}{j}}{(n+1)^{j-1}}\begin{pmatrix}\alpha_j & \beta_j\\ -\alpha_j & -\beta_j\end{pmatrix}.$$

And this is indeed the case, as demonstrated in section 8.4 and appendix C.1.

Finally, note that the transition matrix can be approximated by

$$T_{n,k} = I + \frac{k}{n+k+1}\begin{pmatrix}-3 & 4\\ 3 & 4\end{pmatrix} + O\left(\frac{1}{n^2}\right)\cdot I.$$

Therefore, the values of parallel insertion case can be approximated by the first two terms of the sequential case.

8.3 Two particular cases

We develop the parallel insertion of two and three keys. We follow the same technique applied in [15] to sequential insertions.

$(\cdot,\cdot)$	$P(\cdot,\cdot)$	$E_{..}(X_{n,2}\mid X_n,Y_n)$	$E_{..}(Y_{n,2}\mid X_n,Y_n)$
(x,x)	$\frac{X_n}{n+1}\frac{2}{n+1}$	X_n+2	Y_n
(x_1,x_2)	$\frac{X_n}{n+1}\frac{X_n-2}{n+1}$	X_n-4	Y_n+6
(x,y)	$2\frac{X_n}{n+1}\frac{Y_n}{n+1}$	X_n+2	Y_n
(y,y)	$\frac{Y_n}{n+1}\frac{3}{n+1}$	X_n+2	Y_n
(y_1,y_2)	$\frac{Y_n}{n+1}\frac{Y_n-3}{n+1}$	X_n+8	Y_n-6

Table 8.3: Parallel insertion of two keys

8.3.1 Parallel insertion of two keys

Assume that we have a 2-random tree with n keys and we insert randomly in parallel two additional keys. These keys fall through the tree until they reach bottom nodes. As at most two keys can reach the same bottom node, we have no election, *i.e.* the transformation of bottom nodes is unique (second row of table 8.1).

Both keys can be either at the same bottom node or at different bottom nodes, and in each case bottom nodes can be of type x or y. Let $P(x,x)$ be the probability that both keys reach the same x node, $P(x_1,x_2)$ the probability to reach different x nodes and so on for the remainder probabilities $P(x,y)$ and $P(y_1,y_2)$. We denote the generic case as $P(\cdot,\cdot)$, being $(.,.)$ the generic pair of nodes accessed.

The expected number of $1-type$ leaves conditional on X_n and Y_n is

$$E(X_{n,2}|X_n,Y_n) = \sum_{(\cdot,\cdot)} P(\cdot,\cdot)\ E_{(.,.)}(X_{n,2}|X_n,Y_n)$$

being $E_{..}(X_{n,2}|X_n,Y_n)$ the expected number of x nodes when keys reach nodes $(\cdot,\cdot)$ conditioned to initial expected number of nodes X_n and Y_n (we identify X_n, Y_n with $X_{n,0}, Y_{n,0}$). For instance, if both keys reach different x nodes then it holds

$$P(x_1,x_2) = \frac{X_n}{n+1}\frac{X_n-2}{n+1}$$

and the expectations of $1-type$ leaves is

$$E_{(x1,x2)}(X_{n,2}|X_n,Y_n) = X_n - 4.$$

Table 8.3 contains the remainder values.

The inconditional value of $X_{n,2}$ is

$$E(X_{n,2}) = E(E(X_{n,2}|X_n,Y_n)),$$

and similarly for $Y_{n,2}$.

Lemma 8.1 *Let X_n and Y_n the expectations of $1-type$ and $2-type$ leaves on a random tree of size n. After the random parallel insertion of two keys it holds*

$$E(X_{n,2}) = \left(1+\frac{-4}{n+1}+\frac{12}{(n+1)^2}\right)E(X_n)+\left(\frac{8}{n+1}+\frac{-18}{(n+1)^2}\right)E(Y_n)$$
$$E(Y_{n,2}) = \left(\frac{6}{n+1}+\frac{-12}{(n+1)^2}\right)E(X_n)+\left(1+\frac{-6}{n+1}+\frac{18}{(n+1)^2}\right)E(Y_n)$$

Proof: *We compute the conditional expectation only for $X_{n,2}$ (the term $Y_{n,2}$ has a similar development).*

$$\begin{aligned} E(X_{n,2}|X_n,Y_n) =& \sum_{(\cdot,\cdot)} P(\cdot,\cdot)\ E_{(.,.)}(X_{n,2}|X_n,Y_n) \\ =& \frac{1}{(n+1)^2}\Big(2X_n(X_n+2)+X_n(X_n-2)(X_n-4)+2X_nY_n(X_n+2) \\ & \quad + 3Y_n(X_n+2)+Y_n(Y_n-3)(X_n+8)\Big) \\ =& X_n + \frac{1}{(n+1)^2}\Big(12X_n - 4X_n^2 + 4X_nY_n + 8Y_n^2 - 18Y_n)\Big) \\ =& \left(1-\frac{4}{n+1}+\frac{12}{(n+1)^2}\right)X_n + \left(\frac{8}{n+1}-\frac{18}{(n+1)^2}\right)Y_n \end{aligned}$$

■

Theorem 8.1 *With the same hypothesis as the above lemma the transition matrix is*

$$T_{n,2} = I + \frac{2}{n+3}\begin{pmatrix} -3 & 4 \\ 3 & -4 \end{pmatrix} + \frac{1}{(n+3)(n+1)}\begin{pmatrix} 12 & -18 \\ -12 & 18 \end{pmatrix}$$

Proof: *The transition matrix is defined by $\overrightarrow{P_{n+k}} = T_{n,k}\overrightarrow{P_n}$, and $P^1_{n,2} = E(X_{n,2})/(n+3)$ and $P^2_{n,2} = E(Y_{n,2})/(n+3)$.* ■

8.3.2 Parallel insertion of three keys

The theoretic development is similar to the previous case, then we do not prove the lemmas. Recall that in this case in which three keys can reach the same node there are two possibilities (third row of table 8.1). We have selected the second transformation.

Table 8.4 shows the probability values for this case.

Lemma 8.2 *Let X_n and Y_n the expectations $1-type$ and $2-type$ leaves on a random tree of size n. After the random parallel insertion of three keys it holds*

$$E(X_{n,3}) = \left(1+\frac{-6}{n+1}+\frac{36}{(n+1)^2}+\frac{-48}{(n+1)^3}\right)E(X_n) + \left(\frac{12}{n+1}+\frac{-54}{(n+1)^2}+\frac{54}{(n+1)^3}\right)E(Y_n)$$
$$E(Y_{n,3}) = \left(\frac{9}{n+1}+\frac{-36}{(n+1)^2}+\frac{48}{(n+1)^3}\right)E(X_n) + \left(1+\frac{-9}{n+1}+\frac{54}{(n+1)^2}+\frac{-54}{(n+1)^3}\right)E(Y_n)$$

Theorem 8.2 *With the same hypothesis as the above lemma the transition matrix is*

$$T_{n,3} = I + \frac{3}{n+4}\begin{pmatrix} -3 & 4 \\ 3 & -4 \end{pmatrix} + \frac{3}{(n+4)(n+1)}\begin{pmatrix} 12 & -18 \\ -12 & 18 \end{pmatrix} + \frac{1}{(n+4)(n+1)^2}\begin{pmatrix} -48 & 54 \\ 48 & -54 \end{pmatrix}$$

8.4 Transition matrix of the k keys insertion

In this section we give the transition matrix of k keys parallel insertion case.

Assume that we insert $k > 1$ additional keys on a k-random tree, and one of them is selected and denoted κ. This key can reach an x or y an bottom node; the first case is

Reached nodes	$P(\cdot,\cdot,\cdot)$	$E_{\ldots}(X_{n,3}\|X_n,Y_n)$	$E_{\ldots}(Y_{n,3}\|X_n,Y_n)$
(x,x,x)	$\frac{X_n}{n+1}\left(\frac{2}{n+1}\right)^2$	X_n	Y_n+3
(x_1,x_1,x_2)	$3\frac{X_n}{n+1}\frac{2}{n+1}\frac{X_n-2}{n+1}$	X_n	Y_n+3
(x_1,x_2,x_3)	$\frac{X_n}{n+1}\frac{X_n-2}{n+1}\frac{X_n-4}{n+1}$	X_n-6	Y_n+9
(x,x,y)	$3\frac{X_n}{n+1}\frac{2}{n+1}\frac{Y_n}{n+1}$	X_n+6	Y_n-3
(x,y,y)	$3\frac{X_n}{n+1}\frac{Y_n}{n+1}\frac{3}{n+1}$	X_n	Y_n+3
(x_1,x_2,y)	$3\frac{X_n}{n+1}\frac{X_n-2}{n+1}\frac{Y_n}{n+1}$	X_n	Y_n+3
(x,y_1,y_2)	$3\frac{X_n}{n+1}\frac{Y_n}{n+1}\frac{Y_n-3}{n+1}$	X_n+6	Y_n-3
(y_1,y_2,y_3)	$\frac{Y_n}{n+1}\frac{Y_n-3}{n+1}\frac{Y_n-6}{n+1}$	X_n+12	Y_n-9
(y_1,y_1,y_2)	$3\frac{Y_n}{n+1}\frac{3}{n+1}\frac{Y_n-3}{n+1}$	X_n+6	Y_n-3
(y,y,y)	$\frac{Y_n}{n+1}\left(\frac{3}{n+1}\right)^2$	X_n	Y_n+3

Table 8.4: Parallel insertion of three keys

denoted χ case and the second one γ case. Then the expectations of $1-type$ leaves are given by

$$E(X_{n,k}) = P(\chi)E_\chi(X_{n,k}) + P(\gamma)E_\gamma(X_{n,k})$$

where $E_\chi(X_{n,k}|X_n,Y_n)$ means the expected number of $1-type$ leaves once k keys have been inserted and one of them, κ, has reach an x node. It holds

$$P(\chi) = \frac{X_n}{n+1} \quad , \quad P(\gamma) = \frac{Y_n}{n+1} \qquad \text{and} \qquad X_n + Y_n = n+1.$$

We compute the expectations E_χ and E_γ. Assume that i keys of the remainder $k-1$ ones reach the same node than key κ. We denote this case as χ^i (or γ^i if the node hit by κ is an y node). It holds

$$E_\chi(X_{n,k}) = \sum_{i=0}^{k-1} P(\chi^i)\Big(X_{x,i+1} + E(X_{n-2,k-1-i})\Big)$$
$$E_\gamma(X_{n,k}) = \sum_{i=0}^{k-1} P(\gamma^i)\Big(X_{y,i+1} + E(X_{n-3,k-1-i})\Big)$$

The term $X_{x,i+1}$ $(X_{y,i+1})$ is the number of $1-type$ leaves after the insertion of $i+1$ keys into an x (y) node; for instance, the second row of table 8.1 shows that $X_{x,2}=4$ and $X_{y,2}=2$. As there are $\binom{k-1}{i}$ different ways for i keys to reach the same node than κ the probabilities hold

$$P(\chi^i) = \binom{k-1}{i}\left(\frac{2}{n+1}\right)^i\left(\frac{n-1}{n+1}\right)^{k-1-i}$$
$$P(\gamma^i) = \binom{k-1}{i}\left(\frac{3}{n+1}\right)^i\left(\frac{n-2}{n+1}\right)^{k-1-i}.$$

Lemma 8.3 *The probability that a chosen leaf belongs to a $1-$ type after the random parallel insertion of k keys on a random tree with n keys is*

$$P^1_{n,k} = \sum_{i=0}^{k-1} P(\chi^i)\Big(X_{x,i+1} + (n-2+k-i)P^1_{n-2,k-1-i}\Big)\frac{P^1_{n,0}}{n+k+1}$$
$$+\sum_{i=0}^{k-1} P(\gamma^i)\Big(X_{y,i+1} + (n-3+k-i)P^1_{n-3,k-1-i}\Big)\frac{P^2_{n,0}}{n+k+1}.$$

Theorem 8.3 *The transition matrix when k keys are inserted in parallel is*

$$T_{n,k} = I + \frac{1}{n+k+1}\sum_{j=1}^{k}\frac{(-1)^{j-1}\binom{k}{j}}{(n+1)^{j-1}}\begin{pmatrix}\alpha_j & \beta_j\\ -\alpha_j & -\beta_j\end{pmatrix},$$

and the coefficients are

$$\alpha_1 = -3 \qquad \alpha_2 = -12 \qquad \textit{and}$$
$$\textit{for} \qquad j \geq 3: \;\alpha_j = 2^j\left(-(j-2)(j+1) - 3 + 2\sum_{i=2}^{j-1}(-1)^i\binom{j}{i+1}X_{x,i+1}\right)$$
$$\beta_1 = 4 \qquad \beta_2 = 18 \qquad \textit{and}$$
$$\textit{for} \qquad j \geq 3: \;\beta_j = 3^{j-1}\left(-(j-2)(j-3) - 6 + \sum_{i=2}^{j-1}(-1)^i\binom{j}{i+1}X_{y,i+1}\right)$$

The proof appears in the appendix C.1.

Chapter 9

Conclusions / Conclusions

Design method

We have shown that sequential, distributed and massively parallel paradigms have many common traits in algorithm design. This fact suggested the development of a more **unified design method**. It is worth remembering that in the classical approach, the paradigm is first selected and then the corresponding algorithm design is performed.

Our proposal (see Figure 9.1) suggests the initial design of the distributed algorithm, because it does not take into consideration time control features such as sequentialization or synchronization. Thereafter, the computer paradigm which determines the time control features must be added.

This approach is feasible if we accept that the algorithm design can be separated into two steps:

First, the identification of the local rules that manage the problem data, followed by the design of the distributed algorithm. In this step, the time control characteristics are not considered. We are only interested in the data structure and, above all, the operations and their relationships.

Mètode de disseny

Hem mostrat que els paradigmes seqüencial, distribuït i massivament paral·lel tenen molts trets en comú a l'hora de dissenyar algorismes. Aquest fet ens ha suggerit el desenvolupament d'un **mètode de disseny més unificat**. Recordem que en el punt de vista clàssic es decideix en primer lloc el paradigma, i posteriorment es dissenya l'algorisme corresponent.

La nostra proposta (vegis figura 9.1) suggereix, en primer lloc, el disseny de l'algorisme distribuït, atès que és el més lliure en no existir controls de tipus temporal, és a dir, no cal seqüencialitzar ni sincronitzar processos. Posteriorment escollim el paradigma, el qual determina les característiques de control temporal que cal afegir.

Aquesta aproximació és factible ja que hem dividit el disseny d'algorismes en dues fases:

En primer lloc, la identificació de les regles locals del problema i el disseny l'algorisme distribuït. En aquesta fase no intervenen les característiques de control temporal. Només estem interessats en l'estructura de dades, principalment en les operacions i les seves relacions.

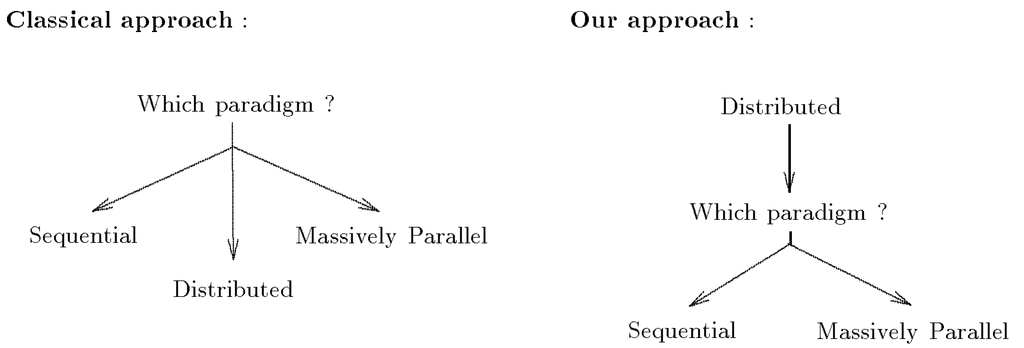


Figure 9.1: Classical approaches *vs* our approach

Second, the introduction of time control characteristics that yield the sequential or the massively parallel algorithm.

En segon lloc, la introducció de les característiques temporals que ens donaran l'algorisme seqüencial o el massivament paral·lel.

Data types and algorithms

We have designed the **massively parallel algorithms for Skip lists**. As Skip list rules access pointers stored in global variables and the design method translates these pointers to local registers, the *list* structure of Skip lists has been transformed into a *tree* structure. These new types of search trees are called **Skip trees**.

We have proved that Skip trees can be one-to-one mapped on Skip lists and that both data types have similar performances. As Skip trees can be stored in external memory (disk) they can be used to manage Data Base indexes. Finally, we have designed a **distributed algorithm with local rules**

We have designed a **fine-grained distributed algorithm with local rules for AVL trees**. Note that we have uncoupled rotations from propagations. We have also designed the **massively parallel algorithms**.

Tipus de dades i algorismes

Hem dissenyat l'**algorisme massivament paral·lel per les "Skip lists"**. Com les regles de les "Skip lists" accedeixen a punters enregistrats en variables globals i el mètode de disseny trasllada aqests punters cap a registres locals, l'estructura en *llistes* de les "Skip lists" s'ha transformat en *arbre*. Aquests nous arbres de cerca han estat anomenats **"Skip trees"**.

Hem demostrat que els "Skip trees" són isomorfos a les "Skip lists" i que tenen similars magnituds. Com els "Skip trees" poden ser enregistrats en memòria externa (disc) poden ser usats per indexar bases de dades. Finalment, hem dissenyat un **algorisme distribuït amb regles locals**.

Hem dissenyat un **algorisme distribuït de granularitat fina pels arbres AVL**. Remarquem que hem separat les rotacions de les propagacions. També hem dissenyat els **algorismes massivament parall·lels**

Our methodology has been widely applied to AVL trees. These trees have presented the most difficult algorithmic problems and our method has been able to solve them satisfactorily. The validity of our approach has been confirmed.

La metodologia ha estat ampliament aplicada en els arbres AVL. Com aquest tipus d'arbres han plantejat els problemes algorísmics més difícils i el mètode ha estat capaç de resoldre'ls satisfactòriament. La validesa de la nostra aproximació ha quedat confirmada.

We feel that algorithms of other search trees, possessing rules similar to those of AVL trees, can be designed in a similar manner. Note, for instance, Red-black trees, Brother trees and Splay trees. This assumption can be extended to those search trees with more simple rules; for instance, 2–3 trees, Skip trees, 2–3–4 trees, B-trees, etc.

Considerem que els algorismes d'altres arbres de cerca amb regles similars a les dels arbres AVL, poden dissenyar-se de forma anàloga. Pensem, per exemple, en els arbres Roig-i-negre, els arbres "Brother" i els arbres "Splay". Aquesta suposició es pot extendre als arbres de cerca amb regles més senzilles, com per exemple, els arbres 2–3, els "Skip trees", els arbres 2–3–4, els arbres B, etc.

We believe that this approach can be applied to other structures such as arrays or lists. For instance, let us assume that an array must be sorted with a local rule which compares two consecutive locations and interchanges them if they are not sorted. The distributed algorithm contains only this rule. The sequential design adds a sequential scheduler which determines how the rule scans the array. Note, for instance, the "bubble-sort" algorithm. The massively parallel algorithms add a parallel scheduler and determine how many processes can scan the array. Note, for instance, the "odd-even transposition" algorithm.

Creiem que aquesta aproximació pot aplicar-se també a d'altres estructures, com ara taules (o vectors) o llistes. Per exemple, suposem que cal ordenar una taula amb una regla local que compara dues posicions consecutives i intercanvia els valors si no estan ben ordenats. L'algorisme distribuït només té aquesta regla. El disseny seqüencial afegeix un distribuïdor seqüencial que determina com la regla recorre la taula; recordem, per exemple, l'algorisme "bombolla". El disseny massivament paral·lel afegeix un distribuïdor paral·lel que determina com els processos accedeixen a la taula; recordem, per exemple, l'algorisme de "transposició sanar-parell".

Parallel case analysis

Tree evolution under parallel insertions has been analyzed for the first time. Namely, the evolution of the leaves type has been analysed by using a similar technique to that employed in the sequential case. The results show how the parallel case tends rapidly towards the sequential case.

Anàlisi del cas paral·lel

Per primer cop s'ha analitzat l'evolució dels arbres 2–3 en inserir claus en paral·lel. Concretament, s'ha analitzat l'evolució del tipus de fulles, generalitzant la tècnica emprada en insercions seqüencials, i el resultat mostra la rapidesa amb què el cas el cas paral·lel tendeix al cas seqüencial.

We believe that this first step will allow the analysis of other evolution aspects of 2–3 trees and other search trees. There are some parallel phenomena that would also be interesting subjects of research since they do not appear in the sequential case. These include the dispersion of insertion or deletion waves, or the mean height at which waves stop.

In summary,

we conclude this dissertation with the conviction that the parallel algorithmic design in main balanced search trees has been accomplished, and that the first steps towards the evolution of search trees under parallel insertions have been made.

Pensem que és un primer pas que permetrà analitzar altres aspectes de l'evolució dels mateixos arbres 2–3 i d'altres arbres de cerca. Serà molt interessant estudiar fenòmens que no es produeixen en el cas seqüencial, com l'amortiment de les ones d'inserció o supressió, o l'altura mitja a la que arriben.

En definitiva,

acabem aquesta tesi amb la convicció que el disseny d'algorismes paral·lels en els principals arbres de cerca s'ha completat, i que s'han donat les primeres passes en l'anàlisi de l'evolució dels arbres de cerca quan s'insereixen les claus en paral·lel.

Appendix A

A.1 C*: Implementation issues

We discuss in this appendix several implementation-related issues and give some examples using the programming language C*.

C* is a data parallel language for the Connection Machine CM-2 system [23, 52]. In the data parallel model of C*, there are many tiny processors, each one capable of storing local data, performing computations and communicating with other processors. Parallel instructions are synchronously issued from the front end to all the processors; each processor can execute the instruction using its own locally stored data or just wait for the next instruction; a status bit controls the behavior of each processor. *Shapes* and *parallel variables* are basic notions of C*. A shape explains how to configure the data in the processors. For instance, a shape declaration like

```
shape [128]my_vector;
```

tells that `my_vector` consists of 128 positions numbered 0, 1, ... , 127 arranged in a linear fashion. There will be a processor associated to each position and each processor can be uniquely identified by its position i. The shape `my_vector` is said to have *rank* 1; bidimensional shapes (of rank 2) and k-dimensional shapes are also possible. Any C* parallel variable has a type and a shape. The following declaration

```
int a_vector:my_vector;
```

introduces a new parallel variable `a_vector` with shape `my_vector` and where each position stores an `integer`. Just as common arrays can be created at runtime in C, both shapes and parallel variables can be dynamically allocated by means of the library functions `allocate_shape` and `palloc`. Shapes and parallel variables can be accessed through pointers, passed as parameters to functions, etc.

In any moment of the execution of a C* program, there must be a *current shape*. Computation and communication can only be done between parallel variables of the current shape. The clause `with` sets the current shape.

Standard C operators such as `=` and `+=` have been given new meanings. For instance, if both `p`, `q` and `r` are parallel variables of shape `my_vector`, the statement `p = q + r;` is equivalent to

for all $i : 0 \leq i < 128$ **do in parallel**
  $p[i] := q[i] + r[i]$
end

The operator `+=` (and others of the type *op*`=`) are parallel reduction operators in C*. For instance, if `s` is an scalar `int`, the statement `s += p;` does in parallel the following

```
s = s + [0]p + [1]p + ··· + [127]p;
```

where `[i]p` denotes the i-th element of the parallel variable `p` (this notation is called *left-indexing*). Besides operators such as `+=` (parallel sum) and `|=` (parallel logical or), `C*` offers library functions such as `scan` that performs sophisticated parallel prefix computations [8].

General communication between processors is possible using *send* or *get* operations. Consider two parallel variables `dst` and `src` of shape `my_vector` and

```
int address:my_vector;
```

Assume furthermore that all values of the parallel variable `address` are in the range 0..127 and different, *i.e.* `address` stores a permutation. The get operation `dst = [address]src;` is equivalent to

for all $i : 0 \leq i < 128$ **do in parallel**
  $dst[i] := src[address[i]]$
end

The send operation `[address]dst = src;` is equivalent to

for all $i : 0 \leq i < 128$ **do in parallel**
  $dst[address[i]] := src[i]$
end

A function worth mentioning in this context is `pcoord(j)` which evaluates to the index of each position along the j-th dimension. For a linear shape such as `my_vector`, the evaluation of `pcoord(0)` at its i-th position returns i.

Finally, processors can be activated and deactivated at will before executing any parallel computation or communication instruction. The clause `where` serves this purpose. In a `where` clause, each (active) processor evaluates a condition; if the result is false, the processor becomes inactive until the `where` clause ends. For instance,

```
with(my_vector)
   where(q >= 0)
      p = q + r;
```

performs the parallel assignment only for those positions i such that `[i]q` is positive.

Appendix B

B.1 Generalized rotations

In this section, we give the precise computation of the new balances generated by the generalized rotations.

For the sake of simplicity, we denote the height of subtree A by a, etc. Also, we restrict ourselves to the contribution of the nodes under study to B and B'.

Single rotation, left-left case

The original tree is $n(p(A, B), C)$, with $\mathbf{lbal}(n) \leq 2$ and $\mathbf{lbal}(p) \leq 0$. The new tree is $p(A, n(B, C))$. Node p is stable. We have

$$a \geq b \qquad a \geq c + 1$$

then

$$\begin{aligned}
&\mathbf{lbal}(p) = b - a \leq 0 \\
&\mathbf{lbal}(n) = c - (a + 1) \leq -2 \\
&\mathbf{lheight}(n) = a + 1 \\
&B = 2a - b - c + 1
\end{aligned}$$

Assume $b \geq c$: Then $\mathbf{lbal}'(n) = c - b \leq 0$.

Assume $a \geq b + 1$: Then

$$\begin{aligned}
&\mathbf{lbal}'(p) = (b + 1) - a \leq 0 \\
&B' = a - c - 1 \\
&B' - B = a - c - 1 - 2a + b + c - 1 = (b - a) - 2 \leq -1 - 2 = -3 \\
&\mathbf{lheight}'(p) = a
\end{aligned}$$

Assume $a < b + 1$: Then, $a = b$.

$$\begin{aligned}
&\mathbf{lbal}'(p) = 1 \\
&B' = b - c + 1 \\
&B' - B = b - c + 1 - 2a + b + c - 1 = 0 \\
&\mathbf{lheight}'(p) = a + 1
\end{aligned}$$

Assume $c \geq b$: Then $a \geq b+1$.

$$\begin{aligned}
&\mathbf{lbal}'(n) = c - b \geq 0 \\
&\mathbf{lbal}'(p) = (c+1) - a \leq 0 \\
&B' = a - b - 1 \\
&B' - B = a - b - 1 - 2a + b + c - 1 = (c-a) - 2 \leq -1 - 2 = -3 \\
&\mathbf{lheight}'(p) = a
\end{aligned}$$

Double rotation, left-right case

The original tree is $n(p(A, q(B, C)), D)$, with $\mathbf{lbal}(n) \leq 2$ and $\mathbf{lbal}(p) > 0$. The new tree is $p(A, n(B, C))$. Nodes p and q are stable. We have

$$\max(b, c) \geq a \qquad \max(b, c) \geq d$$

Assume $b \geq c$: Then $b \geq a$ and $b \geq d$.

$$\begin{aligned}
&\mathbf{lbal}(q) = c - b \leq 0 \\
&\mathbf{lbal}(p) = (b+1) - a \geq 1 \\
&\mathbf{lbal}(n) = d - (b+2) \leq -2 \\
&\mathbf{lheight}(n) = b + 2 \\
&B = 3b - c - a - d + 3 \\
&\mathbf{lbal}'(p) = b - a \geq 0 \\
&\mathbf{lheight}'(q) = b + 1
\end{aligned}$$

Assume $c \geq d$: Then

$$\begin{aligned}
&\mathbf{lbal}'(n) = d - c \leq 0 \\
&\mathbf{lbal}'(q) = c - b \leq 0 \\
&B' = 2b - a - d \\
&B' - B = 2b - a - b - 3b + c + a + d - 3 = -b + c - 3 \leq 0 - 3 = -3
\end{aligned}$$

Assume $d \geq c$: Then

$$\begin{aligned}
&\mathbf{lbal}'(n) = d - c \leq 0 \\
&\mathbf{lbal}'(q) = d - b \leq 0 \\
&B' = 2b - a - c \\
&B' - B = 2b - a - c - 3b + c + a + d - 3 = -b + d - 3 \leq 0 - 3 = -3
\end{aligned}$$

Assume $c \geq b$: Then $c \geq a$ and $c \geq d$.

$$\begin{aligned}
&\mathbf{lbal}(q) = c - b \geq 0 \\
&\mathbf{lbal}(p) = (c+1) - a \geq 1 \\
&\mathbf{lbal}(n) = d - (c+2) \leq -2 \\
&\mathbf{lheight}(n) = c + 2 \\
&B = 3c - a - b - d + 3 \\
&\mathbf{lbal}'(n) = d - c \leq 0 \\
&\mathbf{lheight}'(q) = c + 1
\end{aligned}$$

Assume $a \geq b$: Then

$$\begin{aligned}
&\mathbf{lbal}'(p) = b - a \leq 0 \\
&\mathbf{lbal}'(q) = c - a \geq 0 \\
&B' = 2c - b - d \\
&B' - B = 2c - b - d - 3c + a + b + d - 3 = -b + a - 3 \leq 0 - 3 = -3
\end{aligned}$$

Assume $b \geq a$: Then

$$\begin{aligned}
&\mathbf{lbal}'(p) = b - a \geq 0 \\
&\mathbf{lbal}'(q) = c - b \geq 0 \\
&B' = 2c - a - d \\
&B' - B = 2c - a - d - 3c + a + b + d - 3 = -c + b - 3 \leq 0 - 3 = -3
\end{aligned}$$

Appendix C

C.1 Proofs of chapter 8

Before to address the proof of the main theorem that gives us a recurrence for the coefficients α_j and β_j we introduce some functions. First we adopt the boolean generalization of the function Delta of Kronecker: let p be a boolean expression, then

$$\delta(p) = \begin{cases} 1 \text{ if } p \text{ is true} \\ 0 \text{ if } p \text{ is false} \end{cases}$$

Second we introduce the combinatorial expression

$$\sigma(k,j,i) = (-1)^{k-1-i-j} \binom{k-1}{i} \binom{k-1-i}{j} \text{ for } \quad \begin{cases} k \geq 1 \\ 0 \leq j \leq k \\ 1 \leq i \leq k-1-j \end{cases} \quad .$$

And third, we introduce the functions which are determided by the insertion algorithm features

$$g(x,k,j) = 2^{j-1} \sum_{i=0}^{j-1} \sigma(k,k-j,i) X_{x,i+1}$$

and

$$g(y,k,j) = 3^{j-1} \sum_{i=0}^{j-1} \sigma(k,k-j,i) X_{y,i+1}.$$

Recall that the number $X_{x,i+1}$ ($X_{y,i+1}$) means the number of $1 - type$ leaves after the transformation of an x (y) node when $i+1$ new keys have reached it.

Lemma C.1 *It holds*

$$\sum_{i=0}^{k-1} P(\chi^i) = \sum_{j=0}^{k-1} \frac{2^{k-1-j}}{(n+1)^{k-1-j}} \sum_{i=1}^{k-1-j} \sigma(k,j,i)$$

The addition for $P(\gamma^i)$ is achieved by changing number 2 by 3.

Proof: *From section 8.4*

$$\begin{aligned}
\sum_{i=0}^{k-1} P(\chi^i) =& \sum_{i=0}^{k-1} \binom{k-1}{i} \left(\frac{2}{n+1}\right)^i \left(\frac{n-1}{n+1}\right)^{k-1-i} \\
=& \sum_{i=0}^{k-1} \binom{k-1}{i} \frac{2^i}{(n+1)^i} \sum_{j=0}^{k-1-i} \binom{k-1-i}{j} (-1)^{k-1-i-j} \frac{(n+1)^j 2^{k-1-i-j}}{(n+1)^{k-1-i}} \\
=& \sum_{i=0}^{k-1} \sum_{j=0}^{k-1-i} \binom{k-1}{i} \binom{k-1-i}{j} 2^{k-1-j} (-1)^{k-1-i-j} \frac{1}{(n+1)^{k-1-j}} \\
=& \sum_{j=0}^{k-1} \frac{2^{k-1-j}}{(n+1)^{k-1-j}} \sum_{i=0}^{k-1-j} \binom{k-1}{i} \binom{k-1-i}{j} (-1)^{k-1-i-j} \\
=& \sum_{j=0}^{k-1} \frac{2^{k-1-j}}{(n+1)^{k-1-j}} \sum_{i=0}^{k-1-j} \sigma(k,j,i)
\end{aligned}$$

■

Lemma C.2 *It holds*

$$\sigma(k,j,i) = \frac{k-1}{j} \sigma(k-1, j-1, i) \quad \textit{for} \quad \begin{cases} k > 1 \\ 1 < j \leq k-1 \\ 1 \leq i \leq k-1-i \end{cases}$$

Proof:

$$\begin{aligned}
\sigma(k,j,i) =& \binom{k-1}{i} \binom{k-1-i}{j} (-1)^{k-1-i-j} \\
=& \frac{k-1}{j} \binom{k-2}{i} \binom{k-2-i}{j-1} (-1)^{k-1-i-j} = \frac{k-1}{j} \sigma(k-1, j-1, i)
\end{aligned}$$

■

Lemma C.3 *It holds*

$$\sum_{i=0}^{k-1-j} \sigma(k,j,i) = \delta(k-1=j) \qquad \textit{for} \qquad 1 \leq j < k$$

Proof: *The base of the induction is* $j = 1$,

$$\sum_{i=0}^{k-2} \sigma(k,1,i) = \sum_{i=0}^{k-2} \binom{k-1}{i} (k-1-i)(-1)^{k-i} = (k-1) \sum_{i=0}^{k-2} \binom{k-2}{i} (-1)^{k-i}$$

because of $(k-1-i)\binom{k-1}{i} = (k-1)\binom{k-2}{i}$. *As* $\sum_{i=0}^{k-2} \binom{k-2}{i}(-1)^{k-i} = \delta(k=2)$ *the basis holds for any* $k > j = 1$. *We refer the case* k, j *to* $k-1, j-1$. *Applying previous lemma C.2*

$$\sum_{i=0}^{k-1-j} \sigma(k,j,i) = \sum_{i=0}^{k-1-j} \frac{k-1}{j} \sigma(k-1, j-1, i) = \frac{k-1}{j} \delta(k-2 = j-1) = \delta(k-1, j)$$

■

Lemma C.4 *It holds*

$$\sigma(k,j,i) = -\frac{k-1}{k-1-i-j}\sigma(k-1,j,i) \quad \textit{for} \quad \begin{cases} k > 1 \\ 1 < j \leq k-1 \\ 1 \leq i \leq k-1-i \end{cases}$$

Proof:

$$\begin{aligned}\sigma(k,j,i) =& \binom{k-1}{i}\binom{k-1-i}{j}(-1)^{k-1-i-j} \\ =& \frac{k-1}{k-1-i}\binom{k-2}{i}\frac{k-1-i}{k-1-i-j}\binom{k-2-i}{j}(-1)(-1)^{k-2-i-j}\end{aligned}$$

■

Lemma C.5 *It holds for* $j = 1, 2, \ldots k-2$

$$\sum_{i=0}^{k-2-j} \sigma(k,j,i)\,(k-1-i-j) = -(k-1)\,\delta(k-2=j) \quad \textit{for} \quad \begin{cases} k \geq 1 \\ 1 \leq j \leq k \end{cases}$$

Proof: *By lemma C.4*

$$\begin{aligned}\sum_{i=0}^{k-2-j} \sigma(k,j,i)\,(k-1-i-j) =& -\sum_{i=0}^{k-2-j} \frac{k-1}{k-1-i-j}\sigma(k-1,j,i)\,(k-1-i-j) \\ =& -(k-1)\sum_{i=0}^{k-2-j} \sigma(k-1,j,i)\end{aligned}$$

and by lemma C.3 it is equal to $-(k-1)\,\delta(k-2,j)$. ■

Lemma C.6 *It holds*

$$\sum_{i=0}^{k-1} P(\chi^i)\frac{k-1-i}{n-1} = \frac{k-1}{n+1} \quad \textit{and} \quad \sum_{i=0}^{k-1} P(\gamma^i)\frac{k-1-i}{n-2} = \frac{k-1}{n+1} \quad \textit{for} \quad k \geq 1$$

Proof:

$$\begin{aligned}\sum_{i=0}^{k-1} P(\chi^i)\frac{k-1-i}{n-1} =& \sum_{i=0}^{k-1}\binom{k-1}{i}\left(\frac{2}{n+1}\right)^i\left(\frac{n-1}{n+1}\right)^{k-1-i}\frac{k-1-i}{n-1} \\ =& \sum_{i=0}^{k-2}\binom{k-1}{i}\frac{2^i}{(n+1)^{k-1}}\left((n+1)-2\right)^{k-2-i}(k-1-i) \\ =& \sum_{i=0}^{k-2}\sum_{j=0}^{k-2-i}\binom{k-1}{i}\binom{k-2-i}{j}2^{k-2-j}(-1)^{k-i-j}\frac{1}{(n+1)^{k-1-j}}(k-1-j) \\ =& \sum_{j=0}^{k-2}\frac{2^{k-2-j}}{(n+1)^{k-1-j}}\sum_{i=0}^{k-2-j}\binom{k-1}{i}\binom{k-2-j}{j}(-1)^{k-i-j}(k-1-j) \\ =& \sum_{j=0}^{k-2}\frac{2^{k-2-j}}{(n+1)^{k-1-j}}\sum_{i=0}^{k-2-j}\sigma(k,j,i)(-1)(k-1-i-j)\end{aligned}$$

And by lemma C.5 it is equal to

$$\sum_{j=0}^{k-2} \frac{2^{k-2-j}}{(n+1)^{k-1-j}} \delta(k-2,j)(k-1)$$

■

Lemma C.7 *It holds for $k > 3$*

$$\sum_{i=0}^{k-3} \sum_{l=1}^{k-2-i} \binom{k-1-i}{l} \binom{k-1}{i} (k-1-i-l) 2^{k-2-l} (-1)^{k-i-1} \alpha_l = (-1)^{k-1}(k-1)\alpha_{k-2}$$

Proof: *The addition is equal to*

$$\begin{aligned} &\sum_{l=1}^{k-2} 2^{k-2-l}\ \alpha_l \sum_{i=0}^{k-2-l} \binom{k-1-i}{l} \binom{k-1}{i} (k-1-i-l)(-1)^{k-i-1} \\ =&\sum_{l=1}^{k-2} (-1)^l 2^{k-2-l}\ \alpha_l \sum_{i=0}^{k-2-l} \sigma(k,l,i)\ (k-1-i-l). \end{aligned}$$

By lemma C.5

$$= \sum_{l=1}^{k-2} (-1)^{l+1} 2^{k-2-l} \alpha_l (k-1)\ \delta(k-2=l)$$

■

Lemma C.8 *It holds*

$$\sum_{i=0}^{k-2-j} \sum_{l=1}^{k-1-i-j} (-1)^{k-i-j} 2^{k-1-l-j} \binom{k-1-i}{l} \binom{k-1}{i} \binom{k-1-i-l}{j} \alpha_l$$

$$= (-1)^{k-j} \binom{k-1}{j} \alpha_{k-1-j}$$

Proof: *We prove by induction: the case k,j will be based on the case $k-1, j-1$. As $j \leq k$ the induction is based on the case $k \geq j = 1$, which has been dealed in the previous lemma C.7. The addition is equal to*

$$\frac{k-1}{j} \sum_{i=0}^{k-2-j} \sum_{l=1}^{k-1-i-j} (-1)^{k-i-j} 2^{k-1-l-j} \binom{k-2-i}{l} \binom{k-2}{i} \binom{k-2-i-l}{j-1} \alpha_l.$$

We apply the induction

$$= \frac{k-1}{j} (-1)^{k-j} \binom{k-2}{j-1} \alpha_{k-1-j}.$$

■

Lemma C.9 *It holds*

$$\frac{1}{n-1} \sum_{i=0}^{k-1} P(\chi^i) \sum_{l=1}^{k-1-i} \frac{(-1)^{l-1} \binom{k-1-i}{l} \alpha_l}{(n-1)^{l-1}} = \sum_{j=0}^{k-2} \frac{(-1)^{k-j} \binom{k-1}{j} \alpha_{k-j-1}}{(n+1)^{k-1-j}}$$

and similarly for β coefficients.

Proof:

$$\begin{aligned}
&\frac{1}{n-1}\sum_{i=0}^{k-1} P(\chi^i)\sum_{l=1}^{k-1-i}\frac{(-1)^{l-1}\binom{k-1-i}{l}\alpha_l}{(n-1)^{l-1}}\\
&=\frac{1}{n-1}\sum_{i=0}^{k-2}\binom{k-1}{i}\frac{2^i}{(n+1)^{k-1}}(n-1)^{k-1-i}\sum_{l=1}^{k-1-i}\frac{(-1)^{l-1}\binom{k-1-i}{l}\alpha_l}{(n-1)^{l-1}}\\
&=\sum_{i=0}^{k-2}\sum_{l=1}^{k-1-i}(-1)^{l-1}\binom{k-1-i}{l}\binom{k-1}{i}2^i\frac{(n-1)^{k-1-i-l}}{(n+1)^{k-1}}\alpha_l\\
&=\sum_{i=0}^{k-2}\sum_{l=1}^{k-1-i}\sum_{j=0}^{k-1-i-l}(-1)^{l-1}\binom{k-1-i}{l}\binom{k-1}{i}\binom{k-1-i-l}{j}\frac{2^{k-1-l-j}(-1)^{k-i-j}}{(n+1)^{k-1-j}}\alpha_l
\end{aligned}$$

But

$$\sum_{i=0}^{k-2}\sum_{l=1}^{k-1-i}\sum_{j=0}^{k-1-i-l}=\sum_{j=0}^{k-2}\sum_{i=0}^{k-2-j}\sum_{l=1}^{k-1-i-j}$$

and applying the last previous lemma C.8 we obtain the desired result. ■

C.1.1 Main theorem

From the matrix transition

$$T_{n,k}=I+\frac{1}{n+k+1}\sum_{j=1}^{k}\frac{(-1)^{j-1}\binom{k}{j}}{(n+1)^{j-1}}\begin{pmatrix}\alpha_j & \beta_j\\ -\alpha_j & -\beta_j\end{pmatrix},$$

we obtain the probability

$$P^1_{n,k}=P^1_{n,0}+\frac{1}{n+k+1}\sum_{j=1}^{k}(-1)^{j-1}\binom{k}{j}\frac{\alpha_j\,P^1_{n,0}+\beta_j\,P^2_{n,0}}{(n+1)^{j-1}}.$$

and a similar expression for $P^2_{n,k}$. From now we only develops the first component only.

Proof: *We prove the theorem by induction on k. For $k=1$ we have*

$$P^1_{n,k}=P^1_{n,0}+\frac{\alpha_1\,P^1_{n,0}+\beta_1\,P^2_{n,0}}{n+2}$$

with $\alpha_1=-3$ and $\beta_1=4$. We assume that the theorem holds for smaller values than k, then

$$\begin{aligned}
P^1_{n-2,k-1-i}&=\left(1+\frac{1}{n-2+k-i}\sum_{l=1}^{k-1-i}\frac{(-1)^{l-1}\binom{k-1-i}{l}\alpha_l}{(n-1)^{l-1}}\right)P^1_{n-2,0}\\
&\qquad+\frac{1}{n-2+k-i}\sum_{l=1}^{k-1-i}\frac{(-1)^{l-1}\binom{k-1-i}{l}\beta_l}{(n-1)^{l-1}}\,P^2_{n-2,0}\\
P^1_{n-3,k-1-i}&=\left(1+\frac{1}{n-3+k-i}\sum_{l=1}^{k-1-i}\frac{(-1)^{l-1}\binom{k-1-i}{l}\alpha_l}{(n-2)^{l-1}}\right)P^1_{n-3,0}\\
&\qquad+\frac{1}{n-3+k-i}\sum_{l=1}^{k-1-i}\frac{(-1)^{l-1}\binom{k-1-i}{l}\beta_l}{(n-2)^{l-1}}\,P^2_{n-2,0}
\end{aligned}$$

where

$$P^1_{n-2,0} = \frac{(n+1)\ P^1_{n,0} - 2}{n-1} \qquad P^2_{n-2,0} = \frac{(n+1)\ P^2_{n,0}}{n-1}$$
$$P^1_{n-3,0} = \frac{(n+1)\ P^1_{n,0}}{n-2} \qquad P^2_{n-3,0} = \frac{(n+1)\ P^2_{n,0} - 3}{n-2}$$

Therefore the recurrence becomes

$$\begin{aligned}
P^1_{n,k} = \sum_{i=0}^{k-1} P(\chi^i) \Bigg[& X_{x,i+1} \\
& + \left((n-1) + (k-1-i) + \sum_{l=1}^{k-1-i} \frac{(-1)^{l-1}\binom{k-1-i}{l}\alpha_l}{(n-1)^{l-1}} \right) \frac{(n+1)\ P^1_{n,0} - 2}{n-1} \\
& + \sum_{l=1}^{k-1-i} \frac{(-1)^{l-1}\binom{k-1-i}{l}\beta_l}{(n-1)^{l-1}} \cdot \frac{(n+1)\ P^2_{n,0}}{n-1} \Bigg] \frac{P^1_{n,0}}{n+k+1} \\
+ \sum_{i=0}^{k-1} P(\gamma^i) \Bigg[& X_{y,i+1} \\
& + \left((n-2) + (k-1-i) + \sum_{l=1}^{k-1-i} \frac{(-1)^{l-1}\binom{k-1-i}{l}\alpha_l}{(n-2)^{l-1}} \right) \frac{(n+1)\ P^1_{n,0}}{n-2} \\
& + \sum_{l=1}^{k-1-i} \frac{(-1)^{l-1}\binom{k-1-i}{l}\beta_l}{(n-2)^{l-1}} \cdot \frac{(n+1)\ P^2_{n,0} - 3}{n-2} \Bigg] \frac{P^2_{n,0}}{n+k+1}
\end{aligned}$$

that is equal to

$$\begin{aligned}
P^1_{n,k} = \sum_{i=0}^{k-1} P(\chi^i) \Bigg[& (n+1)P^1_{n,0} - 2 + X_{x,i+1} + \frac{n+1}{n-1}(k-1-i)P^1_{n,0} - \frac{2}{n-1}(k-1-i) \\
& + \frac{n+1}{n-1} \sum_{l=1}^{k-1-i} (-1)^{l-1} \binom{k-1-i}{l} \frac{\alpha_l P^1_{n,0} + \beta_l P^2_{n,0}}{(n-1)^{l-1}} \\
& - \frac{2}{n-1} \sum_{l=1}^{k-1-i} \frac{(-1)^{l-1}\binom{k-1-i}{l}\alpha_l}{(n-1)^{l-1}} \Bigg] \frac{P^1_{n,0}}{n+k+1} \\
+ \sum_{i=0}^{k-1} P(\gamma^i) \Bigg[& (n+1)P^1_{n,0} + X_{y,i+1} + \frac{n+1}{n-2}(k-1-i)P^1_{n,0} \\
& + \frac{n+1}{n-2} \sum_{l=1}^{k-1-i} (-1)^{l-1} \binom{k-1-i}{l} \frac{\alpha_l P^1_{n,0} + \beta_l P^2_{n,0}}{(n-2)^{l-1}} \\
& - \frac{3}{n-2} \sum_{l=1}^{k-1-i} \frac{(-1)^{l-1}\binom{k-1-i}{l}\beta_l}{(n-2)^{l-1}} \Bigg] \frac{P^2_{n,0}}{n+k+1}.
\end{aligned}$$

We apply lemma C.1 to $X_{x,i+1}$ term and lemma C.6 to the last two terms of the first line and to the last term of the fourth line.

$$\begin{aligned}
P^1_{n,k} = \Bigg[&(n+1)P^1_{n,0}-2+\sum_{j=0}^{k-1}\frac{2^{k-1-j}}{(n+1)^{k-1-j}}\sum_{i=0}^{k-1-j}\sigma(k,j,i)X_{x,i+1}+(k-1)P^1_{n,0}-\frac{2(k-1)}{n+1}\\
&+\frac{n+1}{n-1}\sum_{i=0}^{k-1}P(\chi^i)\sum_{l=1}^{k-1-i}(-1)^{l-1}\binom{k-1-i}{l}\frac{\alpha_l P^1_{n,0}+\beta_l P^2_{n,0}}{(n-1)^{l-1}}\\
&-\frac{2}{n-1}\sum_{i=0}^{k-1}P(\chi^i)\sum_{l=1}^{k-1-i}\frac{(-1)^{l-1}\binom{k-1-i}{l}\alpha_l}{(n-1)^{l-1}}\Bigg]\frac{P^1_{n,0}}{n+k+1}\\
+\Bigg[&(n+1)P^1_{n,0}+\sum_{j=0}^{k-1}\frac{3^{k-1-j}}{(n+1)^{k-1-j}}\sum_{i=0}^{k-1-j}\sigma(k,j,i)X_{y,i+1}+(k-1)P^1_{n,0}\\
&+\frac{n+1}{n-2}\sum_{i=0}^{k-1}P(\gamma^i)\sum_{l=1}^{k-1-i}(-1)^{l-1}\binom{k-1-i}{l}\frac{\alpha_l P^1_{n,0}+\beta_l P^2_{n,0}}{(n-2)^{l-1}}\\
&-\frac{3}{n-2}\sum_{i=0}^{k-1}P(\gamma^i)\sum_{l=1}^{k-1-i}\frac{(-1)^{l-1}\binom{k-1-i}{l}\beta_l}{(n-2)^{l-1}}\Bigg]\frac{P^2_{n,0}}{n+k+1}.
\end{aligned}$$

By lemma C.9 the second, third, fifth and sixth lines become

$$\begin{aligned}
P^1_{n,k} = \Bigg[&(n+1)P^1_{n,0}-2+\sum_{j=0}^{k-1}\frac{2^{k-1-j}}{(n+1)^{k-1-j}}\sum_{i=0}^{k-1-j}\sigma(k,j,i)X_{x,i+1}+(k-1)P^1_{n,0}-\frac{2(k-1)}{n+1}\\
&+\sum_{j=0}^{k-2}(-1)^{k-j}\binom{k-1}{j}\frac{\alpha_{k-j-1}P^1_{n,0}+\beta_{k-j-1}P^2_{n,0}}{(n+1)^{k-2-j}}\\
&-2\sum_{j=0}^{k-2}\frac{(-1)^{k-j}\binom{k-1}{j}\alpha_{k-j-1}}{(n+1)^{k-1-j}}\Bigg]\frac{P^1_{n,0}}{n+k+1}\\
+\Bigg[&(n+1)P^1_{n,0}+\sum_{j=0}^{k-1}\frac{3^{k-1-j}}{(n+1)^{k-1-j}}\sum_{i=0}^{k-1-j}\sigma(k,j,i)X_{y,i+1}+(k-1)P^1_{n,0}\\
&+\sum_{j=0}^{k-2}(-1)^{k-j}\binom{k-1}{j}\frac{\alpha_{k-j-1}P^1_{n,0}+\beta_{k-j-1}P^2_{n,0}}{(n+1)^{k-2-j}}\\
&-3\sum_{j=0}^{k-2}\frac{(-1)^{k-j}\binom{k-1}{j}\beta_{k-j-1}}{(n+1)^{k-1-j}}\Bigg]\frac{P^2_{n,0}}{n+k+1}.
\end{aligned}$$

As function $g(x,k,k.j)$ is defined as follows

$$2^{k-j-1}\sum_{i=0}^{k-j-1}\sigma(k,j,i)X_{x,i+1}=g(x,k,k-j)$$

and a similar expression for $g(y,k,k-j)$, and the repetition of the term

$$\sum_{j=0}^{k-2}(-1)^{k-j}\binom{k-1}{j}\frac{\alpha_{k-j-1}P^1_{n,0}+\beta_{k-j-1}P^2_{n,0}}{(n+1)^{k-2-j}}$$

into the second and fifth lines, we obtain

$$\begin{aligned}P^1_{n,k} = &\Bigg[(n+k+1) - 3 + \sum_{j=0}^{k-1} \frac{g(x,k,k-j)}{(n+1)^{k-1-j}} - \frac{2(k-1)}{n+1} \\ &+ \sum_{j=0}^{k-2} \frac{(-1)^{k-j}\binom{k-1}{j}\alpha_{k-j-1}}{(n+1)^{k-2-j}} - 2\sum_{j=0}^{k-2} \frac{(-1)^{k-j}\binom{k-1}{j}\alpha_{k-j-1}}{(n+1)^{k-1-j}}\Bigg] \frac{P^1_{n,0}}{n+k+1} \\ + &\Bigg[\sum_{j=0}^{k-1} \frac{g(y,k,k-j)}{(n+1)^{k-1-j}} + \sum_{j=0}^{k-2} \frac{(-1)^{k-j}\binom{k-1}{j}\beta_{k-j-1}}{(n+1)^{k-2-j}} - 3\sum_{j=0}^{k-2} \frac{(-1)^{k-j}\binom{k-1}{j}\beta_{k-j-1}}{(n+1)^{k-1-j}}\Bigg] \frac{P^2_{n,0}}{n+k+1}.\end{aligned}$$

As this matrix is equal to

$$P^1_{n,k} = P^1_{n,0} + \frac{1}{n+k+1} \sum_{j=1}^{k} (-1)^{j-1} \binom{k}{j} \frac{\alpha_j\, P^1_{n,0} + \beta_j\, P^2_{n,0}}{(n+1)^{j-1}}.$$

we obtain the values of matrix coefficients. The terms $(n+1)^0$ *give the values of* α_1 *and* β_1*:*

$$\begin{aligned} k\alpha_1 =& -3 + g(x,k,1) + (k-1)\alpha_1 \\ k\beta_1 =& g(y,k,1) - (k-1)\beta_1; \end{aligned}$$

the terms $(n+1)^{-1}$ *give the values of* α_2 *and* β_2*:*

$$\begin{aligned} -\binom{k}{2} =& g(x,k,2) - 2(k-1) - \binom{k-1}{2}\alpha_2 - 2\binom{k-1}{1}\alpha_1 \\ -\binom{k}{2} =& g(y,k,2) - \binom{k-1}{2}\beta_2 - 3\binom{k-1}{1}\beta_1; \end{aligned}$$

and the remainder terms give the values of α_2 *and* β_2 *for* $j \geq 3$*:*

$$\begin{aligned} (-1)^{j-1}\binom{k}{j}\alpha_j =& g(x,k-j,j) + (-1)^{j-1}\binom{k-1}{k-j-1}\alpha_j - 2(-1)^j\binom{k-1}{k-j}\alpha_{j-1} \\ (-1)^{j-1}\binom{k}{j}\beta_j =& g(x,k-j,j) + (-1)^{j-1}\binom{k-1}{k-j-1}\beta_j - 3(-1)^j\binom{k-1}{k-j}\beta_{j-1} \end{aligned}$$

As $g(x,k,1) = 0$ *and* $g(y,k,1) = 4$ *then* $\alpha_1 = -3$ *and* $\beta_1 = 4$*. As* $g(x,k,2) = 8(k-1)$ *and* $g(y,k,2) = -6(k-1)$ *then* $\alpha_2 = -12$ *and* $\beta_2 = 18$*.*

As $\binom{k}{j} - \binom{k-1}{k-j-1} = \binom{k-1}{j-1}$ *and* $\sigma(x,k-j,j) = (-1)^j - i - 1\binom{k-1}{j-1}\binom{j-1}{i}$*, for* $j \geq 3$ *it holds*

$$\alpha_j = 2^{j-1} \sum_{i=0}^{j-1} (-1)^{i+1} \binom{j-1}{i} X_{x,i+1} - 2\alpha_{j-1}$$

$$\beta_j = 3^{j-1} \sum_{i=0}^{j-1} (-1)^{i+1} \binom{j-1}{i} X_{y,i+1} - 2\beta_{j-1}$$

These linear recurrences can be solved. For instance, the coefficients α_j become

$$\begin{aligned}
\alpha_j &= 2^{j-1}\sum_{i=0}^{j-1}(-1)^{i+1}\binom{j-1}{i}X_{x,i+1} - 2\alpha_{j-1} \qquad \text{for } j \geq 3 \\
&= \sum_{l=0}^{j-3} 2^{j-1} \sum_{i=0}^{j-l-1}(-1)^{i+1}\binom{j-l-1}{i}X_{x,i+1} - 2^{j-2}\alpha_2 \\
&= 2^{j-1}\sum_{l=0}^{j-3} X_{x,1} - 2^{j-1}\sum_{l=0}^{j-3}(j-l-1)X_{x,2} + 2^{j-1}\sum_{l=0}^{j-3}\sum_{i=2}^{j-1}(-1)^i\binom{j-l-i}{i}X_{x,i+1} - 3\,2^j
\end{aligned}$$

As $X_{x,1} = 0,\ X_{x,2} = 4$ and

$$\sum_{l=l}^{j-3}(j-l-1) = \frac{(j-2)(j+1)}{2} \qquad \text{and} \qquad \sum_{l=0}^{j-l-1}\binom{j-l-i}{i} = \binom{j}{i+1}$$

we conclude

$$\alpha_j = 2^j\left(-(j-2)(j+1) - 3 + 2\sum_{i=2}^{j-1}(-1)^i\binom{j}{i+1}X_{x,i+1}\right).$$

In a similar manner the coefficients β_j become

$$\beta_j = 3^{j-1}\left(-(j-2)(j-3) - 6 + \sum_{i=2}^{j-1}(-1)^i\binom{j}{i+1}X_{y,i+1}\right).$$

■

Bibliography

[1] G.M. Adel'son-Vel'skiĭ and Y.M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, (3):1259–1263, 1962.

[2] A.V. Aho, J.E. Hopcroft, and Ullman J.D. *Data structures and algorithms*. Addison-Wesley, 1983.

[3] S. Akl. *The design and analysis of parallel algorithms*. Prentice-Hall, 1989.

[4] R.A. Baeza-Yates. Fringe analysis revisited. *ACM Computing Surveys*, 27(1):109–119, 1995.

[5] R.A. Baeza-Yates and Poblete P.V. Higher-order analysis of 2–3 trees. *International Journal of Foundations of Computer Science*, 6(1):1–10, 1995.

[6] R. Bayer and C. McCreight. Organization and maintenance of large ordered indexes. *Acta Inf.*, 1(3):173–189, 1972.

[7] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Inf.*, 9(1):1–21, 1977.

[8] G.E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. on Computers*, 38(11):1526–1538, 1989.

[9] L. Bougé, J. Gabarró, and X. Messeguer. Concurrent AVL revisited: self-balancing distributed search trees. Technical Report LSI-95-54, LSI-UPC, 1995. Also appeared as Tech.Rep. RR95-45,LIP,ENS Lyon.

[10] L. Bougé, J. Gabarró, and X. Messeguer. Where should concurrent rotations take place to rebalance a distributed arbitrary search tree? Technical Report LSI-95-57, LSI-UPC, 1995.

[11] H. Chernoff. A measure of asymptotic efficiency for tests of hypothesis based on the sum of observations. *Ann. Math. Statist.*, 23:493–507, 1952.

[12] L. Devroye. A limit theory for random skip lists. *The Annals of Applied Probability*, 2(3):597–609, 1992.

[13] G. Diehr and B. Faaland. Optimal pagination of B trees with variable-length items. *Communications of the ACM*, 27(3):241–247, 1984.

[14] E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the fly garbage collection: an exercice in cooperation. *Communications of the ACM*, 21:966–975, 1978.

[15] B. Eisenbarth, N. Ziviani, G.H. Gonnet, K. Mehlhorn, and D. Wood. The theory of fringe analysis and its application to 2–3 trees and B-trees. *Information and control*, 55(1-3):125–174, 1982.

[16] C.S. Ellis. Concurrent search and insertion in 2-3 trees. *Acta Informatica*, 14(1):63–86, 1980.

[17] C.S. Ellis. Concurrent search and insertion in AVL trees. *IEEE Transactions on Computers*, C–29(9):811–817, September 1980.

[18] J. Gabarró, C. Martínez, and X. Messeguer. Parallel update and search in skip lists. In R. Baeza-Yates, editor, *Computer Science 2: Research and Applications*, pages 15–26. Plenum Press, New York, 1994.

[19] J. Gabarró, C. Martínez, and X. Messeguer. A design of a parallel dictionary using skip lists. *Theoretical Computer Science*, (158):1–33, 1996.

[20] J. Gabarró and X. Messeguer. Massively parallel and distributed dictionaries on AVL trees and brother trees. In ISCA, editor, *Proc. of 9th International Conference on Parallel and Distributed Computing Systems*, pages 14–17, 1996. An extended version appeared as Technical Report LSI-96-27-R, LSI-UPC,1996.

[21] L. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In IEEE, editor, *Proc. of 19th Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[22] L. Higham and E. Schenks. Maintaining B-trees on an EREW PRAM. *J. of Parallel and Dist. Comp.*, 22:329–335, 1994.

[23] W.D. Hillis and G.L. Steele. Data parallel algorithms. *Comm. ACM*, 29(12):1170–1183, 1986.

[24] T. Kasai and R. Miller. Homomorphisms between models of parallel computation. *J. Comp. Sys. Sci.*, 25:285–331, 1982.

[25] J.L.W. Kessels. On-the-fly optimization of data structures. *Comm. ACM*, 26(11):895–901, 1983.

[26] P. Kirschenhofer, C. Martínez, and H. Prodinger. Analysis of an optimized search algorithm for skip lists. Technical Report LSI-94-12-R, Dept. Llenguatges i Sistemes Informàtics, UPC, 1994.

[27] P. Kirschenhofer and H. Prodinger. The path length of random skip lists. *Acta Informatica*, 1994. to appear.

[28] H.T. Kung and P.L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Systems*, 5(3):354–382, 1980.

[29] Y.S. Kwong and D. Wood. A new method for concurrency in B-trees. *IEEE Trans. on Soft. Engin.*, SE-8(3):211–222, 1982.

[30] K. Larsen. AVL trees with relaxed balance. In IEEE Computer Society, editor, *Proc. of 8th International Parallel Processing Symposium*, pages 888–893, 1994.

[31] P. Lehman and Yao B. Efficient locking for concurrent operations on B trees. *ACM Transactions on Database Systemsd*, 6(4):650–670, 1981.

[32] E. McCreight. Pagination of B* trees with variable length records. *Communications of the ACM*, 20:670–674, 1977.

[33] X. Messeguer. A sequential and parallel implementation of skip lists. Technical Report LSI-94-41-R, LSI-UPC, 1994.

[34] X. Messeguer. Skip trees, an alternative data structure to skip lists in a concurrent approach. Technical Report LSI-96-4-R, LSI-UPC, 1996. Submitted to RAIRO.

[35] O. Nurmi and E. Soisalon-Soininen. Chromatic binary search trees: A structure for concurrent rebalancing. *Acta Informatica*, 33(6):547–557, 1995. Also appeared in 10th ACM PODS, 1991.

[36] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency control in database structures with relaxed balance. In *6th ACM PODS*, pages 170–176, 1987.

[37] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrent balancing and updating of avl trees. In *6 ACM PODS*, 1987.

[38] T. Ottmann, H. Six, and D. Wood. On the correspondende between AVL trees and brother trees. *Computing*, 23(1):43–54, 1979.

[39] T. Papadakis. *Skip Lists and Probabilistic Analysis of Algorithms*. PhD thesis, University of Waterloo, 1993. Available as Technical Report CS-93-28.

[40] T. Papadakis, J. I. Munro, and P. V. Poblete. Analysis of the expected search cost in skip lists. In J. R. Gilbert and R. Karlsson, editors, *Proc. 2nd Scandinavian Workshop on Algorithm Theory, LNCS 447*, pages 160–172. Springer-Verlag, 1990.

[41] T. Papadakis, J. I. Munro, and P. V. Poblete. Average search and update costs in skip lists. *BIT*, 32:316–332, 1992.

[42] W. Paul, U. Vishkin, and H. Wagener. Parallel computation on 2–3 trees. *RAIRO Informatique Théorique*, pages 398–404, 1983.

[43] W. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2–3 trees. In J. Díaz, editor, *Proc. 10th International Colloquium on Automata, Programming and Languages, LNCS 154*, pages 597–609. Springer-Verlag, 1983. Also appeared as "Parallel computation on 2–3 trees" in RAIRO Informatique Théorique, pages 397-404, 1983.

[44] P.E. Pfeiffer. *Probability for applications*. Springer-Verlag, 1992.

[45] W. Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222.1, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, MD, Apr 1989. Also published as UMIACS-TR-90-80.

[46] W. Pugh. A skip list cookbook. Technical Report CS-TR-2286.1, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, MD, 1990. Also published as UMIACS-TR-89-72.1.

[47] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[48] N. Schabanel. Equilibrage AVL distribué d'arbres binaires de recherche. Technical report, Laboratoire de l'Informatique du Parallélisme, Ecole Normal Supérieure de Lyon, Printemps 1996.

[49] J.T. Schwartz. Ultracomputers. *ACM Trans. Programming Languages Sys.*, 4(2):484–521, 1980.

[50] R. Seidel and R.Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996. Appeared in Proc. of 30th Symposium on Foundations of Computer Science, 1989.

[51] S. Sen. Some observations on skip-lists. *Information Processing Letters*, 39(3):173–176, 1991.

[52] MA Thinking Machines Corporation, Cambridge. *Programming guide. Version 6.0.2.* 1991.