Searching with Dice

A survey on randomized data structures

Conrado Martnez

Univ. Politcnica de Catalunya, Spain

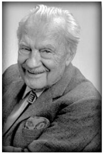
February 2018

Papers We Love

$f(x) = x$

Introduction

R. Karp



N. C. Metropolis

M. O. Rabin

The usefulnees of randomization in the design of algorithms has been known for a long time:

- Metropolis' algorithms
- Rabin's primality test
- Rabin-Karp's string search

Introduction

M.N. Wegman

- Hashing is another early success of randomization for the design of data structures.
- Selecting the hash function from a universal class (Carter and Wegman, 1977) guarantees expected performance

Introduction

Randomization yields algorithms:

- Simple and elegant
- Practical
- With guaranteed expected performance
- Without assumptions on the probabilistic distribution of the input

Introduction

- The usual worst-case analysis is not useful for randomized algorithms
- The probabilistic model to use in the analysis is under control; it is not a working hypothesis, but built-in

Introduction

In this talk:

- Skip lists
- Randomized binary search trees

Skip lists

W. Pugh

- Skip lists were invented by William Pugh (C. ACM, 1990) as a simple alternative to balanced trees
- The algorithms to search, insert, delete, etc. are very simple to understand and to implement, and they have very good expected performance—independent of any assumption on the input

Skip lists

W. Pugh

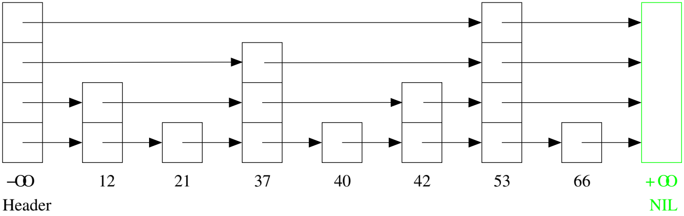
- Skip lists were invented by William Pugh (C. ACM, 1990) as a simple alternative to balanced trees
- The algorithms to search, insert, delete, etc. are very simple to understand and to implement, and they have very good expected performance—independent of any assumption on the input

Skip lists

A skip list S for a set X consists of:

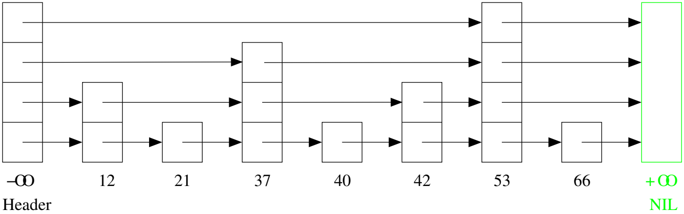
1. A sorted linked list L_1, called level 1, contains all elements of X
2. A collection of non-empty sorted lists L_2, L_3, ..., called level 2, level 3, ... such that for all $i \geq 1$, if an element x belongs to L_i then x belongs to L_{i+1} with probability q, for some $0 < q < 1$, $p := 1 - q$

Skip lists



To implement this, we store the items of X in a collection of nodes each holding an item and a variable-size array of pointers to the item's successor at each level; an additional dummy node gives access to the first item of each level

Skip lists



To implement this, we store the items of X in a collection of nodes each holding an item and a variable-size array of pointers to the item's successor at each level; an additional dummy node gives access to the first item of each level

Implementing skip lists

```
template <typename Key, typename Value>
class Dictionary {
public:
   ...
private:
  struct node_skip_list {
     Key k;
     Value v;
     vector<node_skip_list*> next;

     node_skip_list(const Key& the_key, const Value& the_value, int h) :
                    k(the_key), v(the_value), next(h, nullptr) {
     }
  };
  node_skip_list* header;
  int height;
  double p; // e.g., p = 0.5
  ...
};
```

Skip lists

- The level or height of a node x, height(x), is the number of lists it belongs to.
- It is given by a geometric r.v. of parameter p:

$$\Pr\{\text{height}(x) = k\} = pq^{k-1}, \qquad q = 1 - p$$

- The height of the skip list S is the number of non-empty lists,

$$\text{height}(S) = \max_{x \in S}\{\text{height}(x)\}$$

Skip lists

- The level or height of a node x, height(x), is the number of lists it belongs to.
- It is given by a geometric r.v. of parameter p:

$$\Pr\{\text{height}(x) = k\} = pq^{k-1}, \qquad q = 1 - p$$

- The height of the skip list S is the number of non-empty lists,

$$\text{height}(S) = \max_{x \in S}\{\text{height}(x)\}$$

Skip lists

- The level or height of a node x, height(x), is the number of lists it belongs to.
- It is given by a geometric r.v. of parameter p:

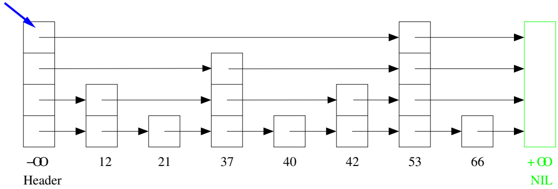
$$\Pr\{\mathsf{height}(x) = k\} = pq^{k-1}, \qquad q = 1 - p$$

- The height of the skip list S is the number of non-empty lists,

$$\mathsf{height}(S) = \max_{x \in S}\{\mathsf{height}(x)\}$$

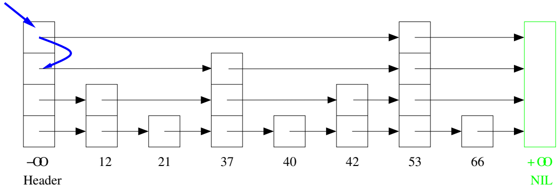
Searching in a skip list

Searching for an item x, $42 < x \leq 53$



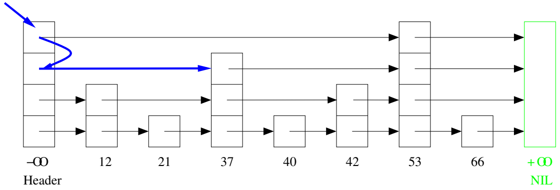
Searching in a skip list

Searching for an item x, $42 < x \leq 53$



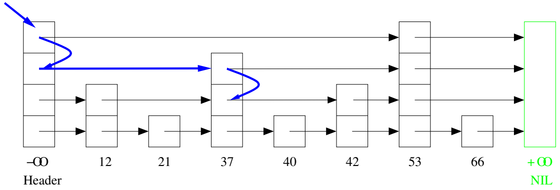
Searching in a skip list

Searching for an item x, $42 < x \leq 53$



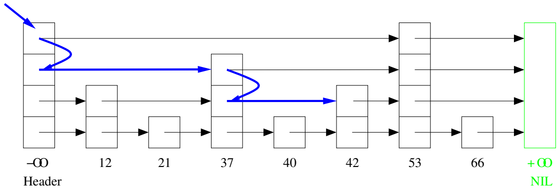
Searching in a skip list

Searching for an item x, $42 < x \leq 53$



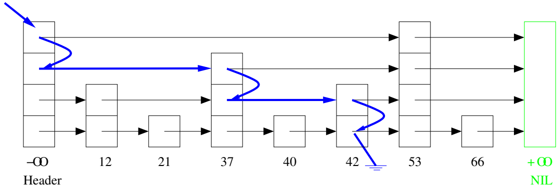
Searching in a skip list

Searching for an item x, $42 < x \leq 53$



Searching in a skip list

Searching for an item x, $42 < x \leq 53$



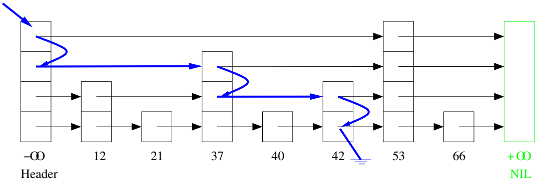
Implementing skip lists

```
// search for an item with key k
// return pointer to item with key k or nullptr if not
// such item exists
node_skip_list*  lookup_skip_list(const Key& k) const {
   node_skip_list* p = header;
   int l = height - 1;
   while (l >= 0)
      if  (p -> next[l] == nullptr or k <= p -> next[l] -> k)
         --l;
      else
         p = p -> next[l];

   if (p -> next[0] == nullptr or p -> next[0] -> k != k)
      // k is not present
      return nullptr;
   else // k is present, return pointer to the node
      return p -> next[0];
}
```

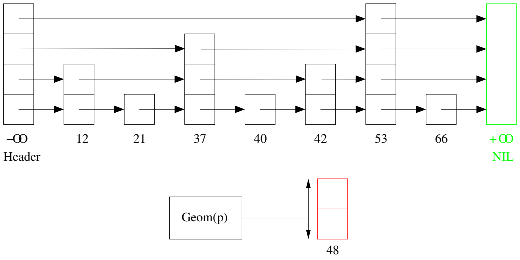
Insertion in a skip list

Inserting an item $x = 48$



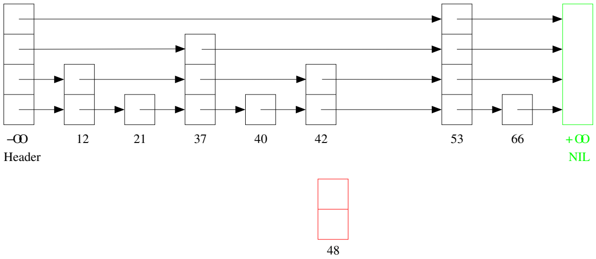
Insertion in a skip list

Inserting an item $x = 48$



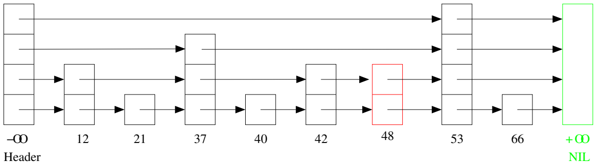
Insertion in a skip list

Inserting an item $x = 48$



Insertion in a skip list

Inserting an item $x = 48$



Implementing skip lists

To insert a new item we go through four phases:

1) Search the given key. The search loop is slightly different from before, since we need to keep track of the last node seen at each level before descending from that level to the one immediately below.
2) If the given key is already present we only update the associated value and finish.

Implementing skip lists

```
void insert_skip_list(const Key& k, const Value& v) {
   // search for insertion point for the new item with key k
   // (or detect it is duplicate)
   node_skip_list* p = header;
   int l = height - 1;
   vector<node_skip_list*> pred(height, header);
   while (l >= 0)
      if  (p -> next[l] == nullptr or k <= p -> next[l] -> k) {
         pred[l] = p; // <====== keep track of predecessor at level l
         --l;
      } else {
         p = p -> next[l];
      }

   if (p -> next[0] == nullptr or p -> next[0] -> _k != k) {
      // k is not present, add new node here
      ...
   }
   else // k is present, update associated value
      p -> next[0] -> v = v;
}
```

Implementing skip lists

3) When k is not present, create a new node with k and v, and assign a random level r to the new node, using geometric distribution
4) Link the new node in the first r lists, adding empty lists if r is larger than the maximum level of the skip list

Implementing skip lists

```
void insert_skip_list(...) {
   ...
   // adding new node
   // generate random height
   // each call to rng() produces a (pseudo)random
   // number uniformly distr. in (0,1)
   int h = 1; while (rng() > p) ++h;

   // create new node
   node_skip_list* nn = new node_skip_list(k, v, h);
   if (h > height) {
      // add new levels to the header and to pred, if necessary
      // make pred[i] = _header for all i = _height .. h-1
      (header -> next).resize(h, nullptr);
      pred.resize(h, header);
   }

   // link the new node to h linked lists
   for (int i = h - 1; i >= 0; --i) {
       nn -> next[i] = pred[i] -> next[i];
       pred[i] -> next[i] = nn;
   }
   ...
}
```

Other Operations

- Deletions are also very easy to implement
- Ordered iterators are trivially implemented
- Skip list can also support many other operations, e.g., merging, search and deletion by rank, finger search, ...
- They can also support concurrency and massive parallelism without too much effort

Other Operations

- Deletions are also very easy to implement
- Ordered iterators are trivially implemented
- Skip list can also support many other operations, e.g., merging, search and deletion by rank, finger search, ...
- They can also support concurrency and massive parallelism without too much effort

Other Operations

- Deletions are also very easy to implement
- Ordered iterators are trivially implemented
- Skip list can also support many other operations, e.g., merging, search and deletion by rank, finger search, ...
- They can also support concurrency and massive parallelism without too much effort

Other Operations

- Deletions are also very easy to implement
- Ordered iterators are trivially implemented
- Skip list can also support many other operations, e.g., merging, search and deletion by rank, finger search, . . .
- They can also support concurrency and massive parallelism without too much effort

Performance of skip lists

A preliminary rough analysis considers the search path backwards. Imagine we are at some node x and level i:

- The height of x is $> i$ and we come from level $i + 1$ since the sought key k is smaller than the key of the successor of x at level $i + 1$
- The height of x is i and we come from x's predecessor at level i since k is larger or equal to the key at x

Performance of skip lists

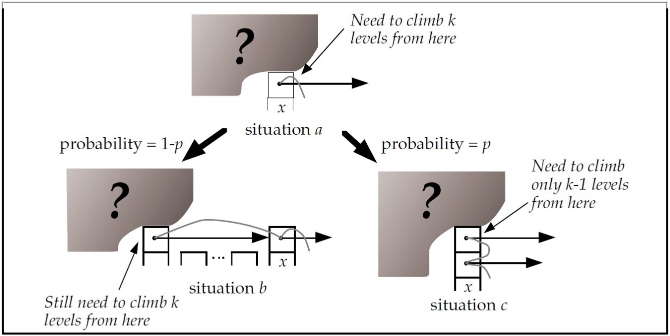


Figure from W. Pugh's *Skip Lists: A Probabilistic Alternative to Balanced Trees* (C. ACM, 1990)—the meaning of p is the opposite of what we have used!

Performance of skip lists

The expected number $C(k)$ of steps to "climb" k levels in an infinite list

$$\begin{aligned} C(k) &= p(1 + C(k)) + (1-p)(1 + C(k-1)) \\ &= 1 + pC(k) + qC(k-1) = \frac{1}{q}(1 + qC(k-1)) \\ &= \frac{1}{q} + C(k-1) = k/q \end{aligned}$$

since $C(0) = 0$.

Performance of skip lists

The analysis above is pessimistic since the list is not infinite and we might "bump" into the header. Then all remaining backward steps to climb up to a level k are vertical—no more horizontal steps. Thus the expected number of steps to climb up to level L_n is

$$\leq (L_n - 1)/q$$

Performance of skip lists

- L_n = the largest level L for which

$$\mathbb{E}[\text{\# of nodes with height} \geq L] \leq 1/q$$

- Probability that a node has height $\geq k$ is

$$\Pr\{\text{height}(x) \geq k\} = \sum_{i \geq k} pq^{i-1} = pq^{k-1} \sum_{i \geq 0} q^i = q^{k-1}$$

- Number of nodes with height $\geq k$ is a binomial r.v. with parameters n and q^{k-1}, hence

$$\mathbb{E}[\text{\# of nodes with height} \geq k] = nq^{k-1}$$

- Then

$$nq^{L_n - 1} = 1/q \implies L_n = \log_q(1/n) = \log_{1/q} n$$

Performance of skip lists

- L_n = the largest level L for which

$$\mathbb{E}[\#\text{ of nodes with height} \geq L] \leq 1/q$$

- Probability that a node has height $\geq k$ is

$$\Pr\{\text{height}(x) \geq k\} = \sum_{i \geq k} pq^{i-1} = pq^{k-1} \sum_{i \geq 0} q^i = q^{k-1}$$

- Number of nodes with height $\geq k$ is a binomial r.v. with parameters n and q^{k-1}, hence

$$\mathbb{E}[\#\text{ of nodes with height} \geq k] = nq^{k-1}$$

- Then

$$nq^{L_n - 1} = 1/q \implies L_n = \log_q(1/n) = \log_{1/q} n$$

Performance of skip lists

- L_n = the largest level L for which

$$\mathbb{E}[\text{\# of nodes with height} \geq L] \leq 1/q$$

- Probability that a node has height $\geq k$ is

$$\Pr\{\text{height}(x) \geq k\} = \sum_{i \geq k} pq^{i-1} = pq^{k-1} \sum_{i \geq 0} q^i = q^{k-1}$$

- Number of nodes with height $\geq k$ is a binomial r.v. with parameters n and q^{k-1}, hence

$$\mathbb{E}[\text{\# of nodes with height} \geq k] = nq^{k-1}$$

- Then

$$nq^{L_n - 1} = 1/q \implies L_n = \log_q(1/n) = \log_{1/q} n$$

Performance of skip lists

- L_n = the largest level L for which

$$\mathbb{E}[\text{\# of nodes with height} \geq L] \leq 1/q$$

- Probability that a node has height $\geq k$ is

$$\Pr\{\text{height}(x) \geq k\} = \sum_{i \geq k} pq^{i-1} = pq^{k-1} \sum_{i \geq 0} q^i = q^{k-1}$$

- Number of nodes with height $\geq k$ is a binomial r.v. with parameters n and q^{k-1}, hence

$$\mathbb{E}[\text{\# of nodes with height} \geq k] = nq^{k-1}$$

- Then

$$nq^{L_n - 1} = 1/q \implies L_n = \log_q(1/n) = \log_{1/q} n$$

Performance of skip lists

Then the steps remaining to reach H_n (=the height of a random skip list of size n) can analyzed this way:

- we need not more horizontal steps than nodes with height $\geq L_n$, the expected number is $\leq 1/q$, by definition
- the probability that $H_n > k$ is

$$1 - \left(1 - q^k\right)^n \leq nq^k$$

- It follows that

$$\mathbb{E}[H_n] \leq L_n + 1/p$$

  and the expected additional vertical steps need to reach H_n from L_n is $\leq 1/p$

Performance of skip lists

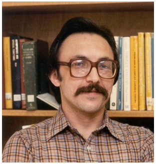
Summing up, the expected path length of a search is

$$\leq (L_n - 1)/q + 1/q + 1/p = \frac{1}{q} \log_{1/q} n + 1/p$$

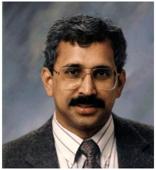
On the other hand, the average number of pointers per node is $1/p$ so there is a trade-off between space and time:

- $p \to 0, q \to 1 \implies$ very tall "nodes", short horizontal cost
- $p \to 1, q \to 0 \implies$ flat skip lists
- Pugh suggests $p = 3/4$, optimal choice minimizes factor $(q \ln(1/q))^{-1}$ is $q = e^{-1} = 0.36\ldots, p = 1 - e^{-1} \approx 0.632\ldots$

Analysis of the height



W. Szpankowski



V. Rego

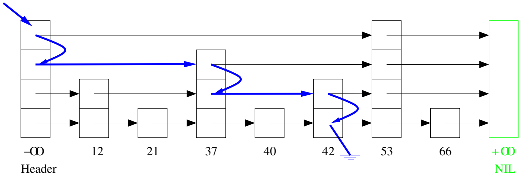
Theorem (Szpankowski and Rego,1990)

$$\mathbb{E}[H_n] = \log_Q n + \frac{\gamma}{L} - \frac{1}{2} + \chi(\log_Q n) + \mathcal{O}(1/n)$$

with $Q := 1/q$, $L := \ln Q$, $\chi(t)$ a fluctuation of period 1, mean 0 and small amplitude.

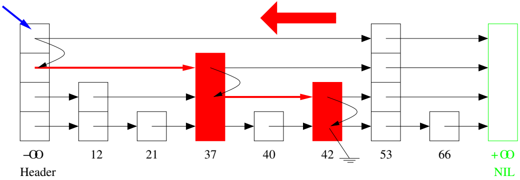
Analysis of the forward cost

The number of forward steps $F_{n,k}$ is the number of weak left-to-right maxima in $a_k, a_{k-1}, \ldots, a_1$, with $a_i = \text{height}(x_i)$



Analysis of the forward cost

The number of forward steps $F_{n,k}$ is the number of weak left-to-right maxima in $a_k, a_{k-1}, \ldots, a_1$, with $a_i = \text{height}(x_i)$



Analysis of the forward cost

- Total unsuccessful search cost

$$C_n = \sum_{0 \le k \le n} C_{n,k} = nH_n + F_n$$

- Total forward cost

$$F_n = \sum_{0 \le k \le n} F_{n,k}$$

Analysis of the forward cost

- Total unsuccessful search cost

$$C_n = \sum_{0 \le k \le n} C_{n,k} = nH_n + F_n$$

- Total forward cost

$$F_n = \sum_{0 \le k \le n} F_{n,k}$$

Analysis of the forward cost

P. Kirschenhofer

H. Prodinger

Theorem (Kirschehofer, Prodinger, 1994)

The expected forward cost in a random skip list of size n is

$$\mathbb{E}[F_n] = (Q-1)n\left(\log_Q n + \frac{\gamma - 1}{L} - \frac{1}{2} + \frac{1}{L}\chi(\log_Q n)\right) + \mathcal{O}(\log n),$$

with $Q := 1/q$, $L = \ln Q$ and χ a periodic fluctuation of period 1, mean 0 and small amplitude.

Skip Lists in Real Life

Usages [edit]

List of applications and frameworks that use skip lists:

- MemSQL uses skip lists as its prime indexing structure for its database technology.
- Cyrus IMAP server offers a "skiplist" backend DB implementation (source file)
- Lucene uses skip lists to search delta-encoded posting lists in logarithmic time.[citation needed]
- QMap (up to Qt 4) template class of Qt that provides a dictionary.
- Redis, an ANSI-C open-source persistent key/value store for Posix systems, uses skip lists in its implementation of ordered sets.[7]
- nessDB, a very fast key-value embedded Database Storage Engine (Using log-structured-merge (LSM) trees), uses skip lists for its memtable.
- skipdb is an open-source database format using ordered key/value pairs.
- ConcurrentSkipListSet and ConcurrentSkipListMap in the Java 1.6 API.
- Speed Tables are a fast key-value datastore for Tcl that use skiplists for indexes and lockless shared memory.
- leveldb, a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values
- Con Kolivas' MuQSS[8] Scheduler for the Linux kernel uses skip lists
- SkiMap uses skip lists as base data structure to build a more complex 3D Sparse Grid for Robot Mapping systems.[9]

Skip lists are used for efficient statistical computations of running medians (also known as moving medians). Skip lists are also used in distributed applications (where the nodes represent physical computers, and pointers represent network connections) and for implementing highly scalable concurrent priority queues with less lock contention,[10] or even without locking,[11][12][13] as well as lockless concurrent dictionaries.[14] There are also several US patents for using skip lists to implement (lockless) priority queues and concurrent dictionaries.[15]

See also [edit]

- Bloom filter

Source: Wikipedia

To learn more

- L. Devroye.
  A limit theory for random skip lists.
  The Annals of Applied Probability, 2(3):597–609, 1992.
- P. Kirschenhofer and H. Prodinger.
  The path length of random skip lists.
  Acta Informatica, 31(8):775–792, 1994.
- P. Kirschenhofer, C. Martnez and H. Prodinger.
  Analysis of an Optimized Search Algorithm for Skip Lists.
  Theoretical Computer Science, 144:199–220, 1995.

To learn more (2)

T. Papadakis, J. I. Munro, and P. V. Poblete.
Average search and update costs in skip lists.
BIT, 32:316–332, 1992.

H. Prodinger.
Combinatorics of geometrically distributed random variables: Left-to-right maxima.
Discrete Mathematics, 153:253–270, 1996.

W. Pugh.
Skip lists: a probabilistic alternative to balanced trees.
Comm. ACM, 33(6):668–676, 1990.

W. Pugh.
A Skip List Cookbook.
Technical Report UMIACS–TR–89–72.1. U. Maryland, College Park, 1989.

Randomized binary search trees

C. Aragon

R. Seidel

Two incarnations

- Randomized treaps (tree+heap) invented by Aragon and Seidel (FOCS 1989, Algorithmica 1996) use random priorities and bottom-up balancing
- Randomized binary search trees (RBSTs) invented by M. and Roura (ESA 1996, JACM 1998) use subtree sizes and top-down balancing

Randomized binary search trees

C. Aragon

R. Seidel

S. Roura

Two incarnations

- Randomized treaps (tree+heap) invented by Aragon and Seidel (FOCS 1989, Algorithmica 1996) use random priorities and bottom-up balancing
- Randomized binary search trees (RBSTs) invented by M. and Roura (ESA 1996, JACM 1998) use subtree sizes and top-down balancing

Randomized binary search trees

- In a random binary search tree (built using random insertions) any of its n elements is the root with probability $1/n$
- Idea: to insert a new item, insert it at the root with probability $1/(n+1)$, otherwise proceed recursively
- The random priorities of treaps "simulate" random timestamps (cif. Vuillemin's Cartesian trees 1980); rotations are used to maintain the BST invariant w.r.t. keys and the heap invariant w.r.t. priorities

Randomized binary search trees

- In a random binary search tree (built using random insertions) any of its n elements is the root with probability $1/n$
- Idea: to insert a new item, insert it at the root with probability $1/(n+1)$, otherwise proceed recursively
- The random priorities of treaps “simulate” random timestamps (cif. Vuillemin’s Cartesian trees 1980); rotations are used to maintain the BST invariant w.r.t. keys and the heap invariant w.r.t. priorities

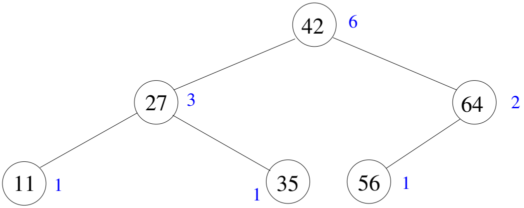
Randomized binary search trees

J. Vuillemin

- In a random binary search tree (built using random insertions) any of its n elements is the root with probability $1/n$
- Idea: to insert a new item, insert it at the root with probability $1/(n+1)$, otherwise proceed recursively
- The random priorities of treaps "simulate" random timestamps (cif. Vuillemin's Cartesian trees 1980); rotations are used to maintain the BST invariant w.r.t. keys and the heap invariant w.r.t. priorities

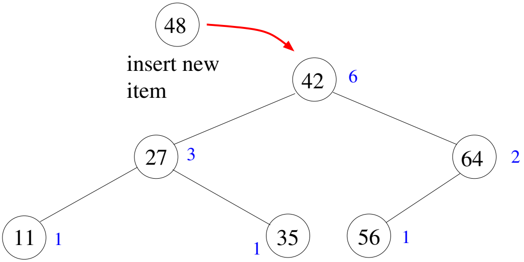
Insertion in a RBST

Inserting an item $x = 48$



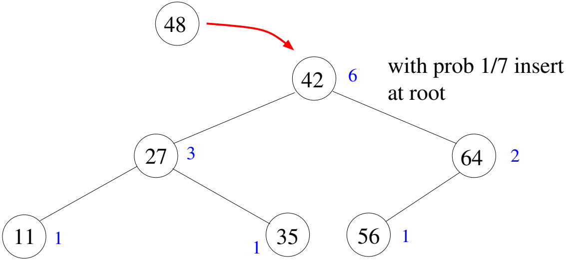
Insertion in a RBST

Inserting an item $x = 48$



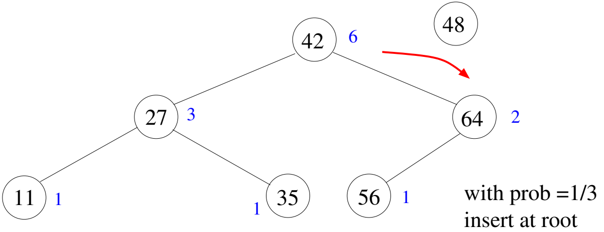
Insertion in a RBST

Inserting an item $x = 48$



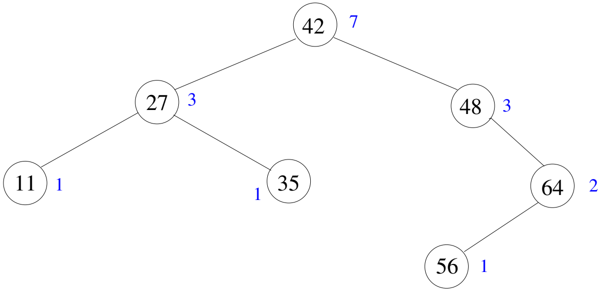
Insertion in a RBST

Inserting an item $x = 48$



Insertion in a RBST

Inserting an item $x = 48$



Insertion in a RBST

```
int size(node* p) {
     return (T == nullptr) ? 0 : T -> size;
}

void update_size(node* p) {
     if (p != nullptr)
         p -> size = size(p -> left) + size(p -> right) + 1;
}
```

```
// we assume here that k is not present in T
node* insert(node* T, const Key& k, const Value& v) {
  int n = size(T); // size of the subtree
  if (Uniform(0,n) == 0) // with probability 1/(n+1)
     return insert_at_root(T, k, v);
  else { // with probability n/(n+1)
     if (k < T -> k)
        T -> left = insert(T -> left, k, v);
     else
        T -> right = insert(T -> right, k, v);
     update_size(T);
     return T;
  }
}
```

Insertion in a RBST

- To insert a new item x at the root of T, we use the algorithm Split that returns two RBSTs T^- and T^+ with element smaller and larger than x, resp.

$$\begin{aligned}\langle T^-, T^+\rangle &= \mathsf{Split}(T, x)\\ T^- &= \mathsf{BST\ for}\ \{y \in T \,|\, y < x\}\\ T^+ &= \mathsf{BST\ for}\ \{y \in T \,|\, x < y\}\end{aligned}$$

- Split is like partition in Quicksort
- Insertion at root was invented by Stephenson in 1976

Insertion in a RBST

- To insert a new item x at the root of T, we use the algorithm Split that returns two RBSTs T^- and T^+ with element smaller and larger than x, resp.

$$\begin{aligned} \langle T^-, T^+ \rangle &= \mathsf{Split}(T, x) \\ T^- &= \mathsf{BST\ for}\ \{y \in T \,|\, y < x\} \\ T^+ &= \mathsf{BST\ for}\ \{y \in T \,|\, x < y\} \end{aligned}$$

- Split is like partition in Quicksort
- Insertion at root was invented by Stephenson in 1976

Insertion in a RBST

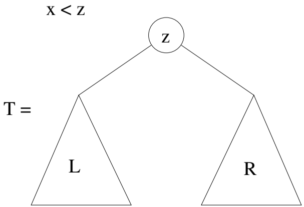
- To insert a new item x at the root of T, we use the algorithm Split that returns two RBSTs T^- and T^+ with element smaller and larger than x, resp.

$$\begin{aligned}\langle T^-, T^+\rangle &= \mathsf{Split}(T, x)\\ T^- &= \mathsf{BST\ for}\ \{y \in T \,|\, y < x\}\\ T^+ &= \mathsf{BST\ for}\ \{y \in T \,|\, x < y\}\end{aligned}$$

- Split is like partition in Quicksort
- Insertion at root was invented by Stephenson in 1976

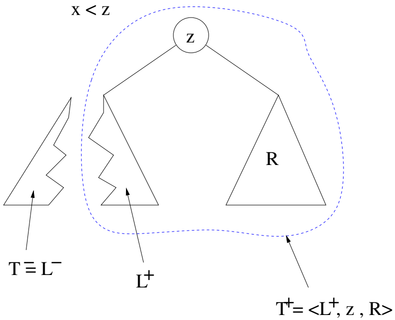
Splitting a RBST

To split a RBST T around x, we need just to follow the path from the root of T to the leaf where x falls



Splitting a RBST

To split a RBST T around x, we need just to follow the path from the root of T to the leaf where x falls



Splitting a RBST & Insertion at Root

```
// splits the RBST T (destructively) into two trees, one with
// keys smaller than k, the other with keys larger than k
pair<node*, node*> split(node* T, const Key& k) {
  if (T == nullptr) return make_pair(nullptr, nullptr);
  if (k < T -> k) {
     pair<node*, node*> result = split(T -> left, k);
     T -> left = result.second;
     update_size(T);
     result.second = T;
     return result;
  } else { // idem, change left <-> right
             // first <-> second
     ...
  }
}

node* insert_at_root(node* T, const Key& k, const Value& v) {
  pair<node*, node*> LR = split(T, k); // $LR = \langle T^-, T^+\rangle$
  node* nn = new node(k, v);
  nn -> left = LR.first;
  nn -> right = LR.second;
  update_size(nn);
  return nn;
}
```

Splitting a RBST

Lemma

Let T^- *and* T^+ *be the BSTs produced by* Split(T, x). *If* T *is a random BST containing the set of keys* K, *then* T^- *and* T^+ *are independent random BSTs containing the sets of keys* $K^- = \{y \in T \mid y < x\}$ *and* $K^+ = \{y \in T \mid y > x\}$, *respectively.*

Insertion in RBSTs

Theorem

If T is a random BST that contains the set of keys K and x is any key not in K, then $\text{Insert}(T, x)$ *produces a random BST containing the set of keys $K \cup \{x\}$.*

The Cost of Insertions

- The cost of the insertion at root (measured # of visited nodes) is exactly the same as the cost of the standard insertion
- For a random(ized) BST the cost of insertion is the depth of a random leaf in a random binary searh tree:

$$\mathbb{E}[I_n] = 2\log n + \mathcal{O}(1)$$

- We need to produce $\mathcal{O}(\log n)$ random numbers on average to insert an item

The Cost of Insertions

- The cost of the insertion at root (measured # of visited nodes) is exactly the same as the cost of the standard insertion
- For a random(ized) BST the cost of insertion is the depth of a random leaf in a random binary searh tree:

$$\mathbb{E}[I_n] = 2\log n + \mathcal{O}(1)$$

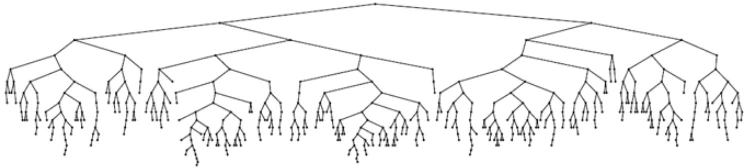
- We need to produce $\mathcal{O}(\log n)$ random numbers on average to insert an item

The Cost of Insertions

- The cost of the insertion at root (measured # of visited nodes) is exactly the same as the cost of the standard insertion
- For a random(ized) BST the cost of insertion is the depth of a random leaf in a random binary searh tree:

$$\mathbb{E}[I_n] = 2\log n + \mathcal{O}(1)$$

- We need to produce $\mathcal{O}(\log n)$ random numbers on average to insert an item



RBST resulting from the insertion of 500 keys in ascending order
Source: R. Sedgewick, *Algorithms in C* (3rd edition), 1997

Deletions in RBSTs

- The fundamental problem is how to remove the root node of a BST, in particular, when both subtrees are not empty
- The original deletion algorithm by Hibbard was assumed to preserve randomness
- In 1975, G. Knott discovered that Hibbard's deletion preserves randomness of shape, but an insertion following a deletion would destroy randomness (Knott's paradox)

Deletions in RBSTs

- The fundamental problem is how to remove the root node of a BST, in particular, when both subtrees are not empty
- The original deletion algorithm by Hibbard was assumed to preserve randomness
- In 1975, G. Knott discovered that Hibbard's deletion preserves randomness of shape, but an insertion following a deletion would destroy randomness (Knott's paradox)

Deletions in RBSTs

- The fundamental problem is how to remove the root node of a BST, in particular, when both subtrees are not empty
- The original deletion algorithm by Hibbard was assumed to preserve randomness
- In 1975, G. Knott discovered that Hibbard's deletion preserves randomness of shape, but an insertion following a deletion would destroy randomness (Knott's paradox)

Deletions in RBSTs

J. Culberson

J.L. Eppinger

D.E. Knuth

- Several theoretical and experimental work aimed at understanding what was the effect of deletions, e.g.,
    - Jonassen & Knuth's *An Algorithm whose Analysis Isn't* (JCSS, 1978)
    - Knuth's *Deletions that Preserve Randomness* (IEEE Trans. Soft. Eng., 1977)
    - Eppinger's experiments (CACM, 1983)
    - Culberson's paper on deletions of the left spine (STOC, 1985)
    - ...
- These studies showed that deletions degraded performance in the long run

Deletions in RBSTs

J. Culberson

J.L. Eppinger

D.E. Knuth

- Several theoretical and experimental work aimed at understanding what was the effect of deletions, e.g.,
  - Jonassen & Knuth's *An Algorithm whose Analysis Isn't* (JCSS, 1978)
  - Knuth's *Deletions that Preserve Randomness* (IEEE Trans. Soft. Eng., 1977)
  - Eppinger's experiments (CACM, 1983)
  - Culberson's paper on deletions of the left spine (STOC, 1985)
  - ...
- These studies showed that deletions degraded performance in the long run

Deletions in RBSTs

J. Culberson

J.L. Eppinger

D.E. Knuth

- Several theoretical and experimental work aimed at understanding what was the effect of deletions, e.g.,
  - Jonassen & Knuth's *An Algorithm whose Analysis Isn't* (JCSS, 1978)
  - Knuth's *Deletions that Preserve Randomness* (IEEE Trans. Soft. Eng., 1977)
  - Eppinger's experiments (CACM, 1983)
  - Culberson's paper on deletions of the left spine (STOC, 1985)
  - ...
- These studies showed that deletions degraded performance in the long run

Deletions in RBSTs

J. Culberson

J.L. Eppinger

D.E. Knuth

- Several theoretical and experimental work aimed at understanding what was the effect of deletions, e.g.,
  - Jonassen & Knuth's *An Algorithm whose Analysis Isn't* (JCSS, 1978)
  - Knuth's *Deletions that Preserve Randomness* (IEEE Trans. Soft. Eng., 1977)
  - Eppinger's experiments (CACM, 1983)
  - Culberson's paper on deletions of the left spine (STOC, 1985)
  - ...
- These studies showed that deletions degraded performance in the long run

Deletions in RBSTs

J. Culberson

J.L. Eppinger

D.E. Knuth

- Several theoretical and experimental work aimed at understanding what was the effect of deletions, e.g.,
  - Jonassen & Knuth's *An Algorithm whose Analysis Isn't* (JCSS, 1978)
  - Knuth's *Deletions that Preserve Randomness* (IEEE Trans. Soft. Eng., 1977)
  - Eppinger's experiments (CACM, 1983)
  - Culberson's paper on deletions of the left spine (STOC, 1985)
  - ...
- These studies showed that deletions degraded performance in the long run

Deletions in RBSTs


J. Culberson


J.L. Eppinger


D.E. Knuth

- Several theoretical and experimental work aimed at understanding what was the effect of deletions, e.g.,
  - Jonassen & Knuth's *An Algorithm whose Analysis Isn't* (JCSS, 1978)
  - Knuth's *Deletions that Preserve Randomness* (IEEE Trans. Soft. Eng., 1977)
  - Eppinger's experiments (CACM, 1983)
  - Culberson's paper on deletions of the left spine (STOC, 1985)
  - ...
- These studies showed that deletions degraded performance in the long run

Deletions in RBSTs

J. Culberson

J.L. Eppinger

D.E. Knuth

- Several theoretical and experimental work aimed at understanding what was the effect of deletions, e.g.,
  - Jonassen & Knuth's *An Algorithm whose Analysis Isn't* (JCSS, 1978)
  - Knuth's *Deletions that Preserve Randomness* (IEEE Trans. Soft. Eng., 1977)
  - Eppinger's experiments (CACM, 1983)
  - Culberson's paper on deletions of the left spine (STOC, 1985)
  - ...
- These studies showed that deletions degraded performance in the long run

Deletions in RBSTs

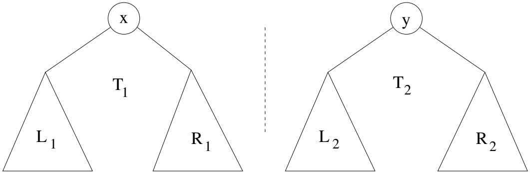
```
node* remove(node* T, const Key& k) {
  if (T == nullptr) return nullptr;
  if (T -> k == k) { // to delete the root of the subtree, join the subtrees
    node* result = join(T -> left, T -> right);
    T -> left = T -> right = nullptr;
    free(T); // release node
    return result;
  }
  if (k < T -> k)
    T -> left = remove(T -> left, k);
  else
    T -> right = remove(T -> right, k);
  update_size(T);
  return T;
}
```

Deletions in RBSTs

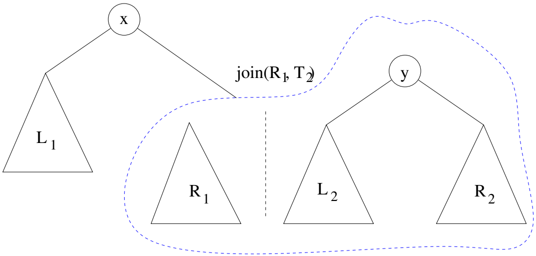
We delete the root using a procedure Join(T_1, T_2). Given two BSTs such that for all $x \in T_1$ and all $y \in T_2$, $x \le y$, it returns a new BST that contains all the keys in T_1 and T_2.

$$\begin{aligned} \text{Join}(\square, \square) &= \square \\ \text{Join}(T, \square) &= \text{Join}(\square, T) = T \\ \text{Join}(T_1, T_2) &= ?, \qquad T_1 \neq \square, T_2 \neq \square \end{aligned}$$

Joining two BSTs



Joining two BSTs



Joining two BSTs

- If we systematically choose the root of T_1 as the root of Join(T_1, T_2), or the other way around, we will introduce an undesirable bias
- Suppose both T_1 and T_2 are random. Let m and n denote their sizes. Then x is the root of T_1 with probability $1/m$ and y is the root of T_2 with probability $1/n$
- Choose x as the common root with probability $m/(m+n)$, choose y with probability $n/(m+n)$

$$\frac{1}{m} \times \frac{m}{m+n} = \frac{1}{m+n}$$
$$\frac{1}{n} \times \frac{n}{m+n} = \frac{1}{m+n}$$

Joining two BSTs

- If we systematically choose the root of T_1 as the root of Join(T_1, T_2), or the other way around, we will introduce an undesirable bias
- Suppose both T_1 and T_2 are random. Let m and n denote their sizes. Then x is the root of T_1 with probability $1/m$ and y is the root of T_2 with probability $1/n$
- Choose x as the common root with probability $m/(m+n)$, choose y with probability $n/(m+n)$

$$\frac{1}{m} \times \frac{m}{m+n} = \frac{1}{m+n}$$
$$\frac{1}{n} \times \frac{n}{m+n} = \frac{1}{m+n}$$

Joining two BSTs

- If we systematically choose the root of T_1 as the root of Join(T_1, T_2), or the other way around, we will introduce an undesirable bias
- Suppose both T_1 and T_2 are random. Let m and n denote their sizes. Then x is the root of T_1 with probability $1/m$ and y is the root of T_2 with probability $1/n$
- Choose x as the common root with probability $m/(m+n)$, choose y with probability $n/(m+n)$

$$\frac{1}{m} \times \frac{m}{m+n} = \frac{1}{m+n}$$
$$\frac{1}{n} \times \frac{n}{m+n} = \frac{1}{m+n}$$

Joining two RBSTs

Lemma

Let L and R be two independent random BSTs, such that the keys in L are strictly smaller than the keys in R. Let K_L and K_R denote the sets of keys in L and R, respectively. Then $T = \mathrm{Join}(L, R)$ is a random BST that contains the set of keys $K = K_L \cup K_R$.

Joining two RBSTs

- The recursion for Join(T_1, T_2) traverses the rightmost branch (right spine) of T_1 and the leftmost branch (left spine) of T_2
- The trees to be joined are the left and right subtrees L and R of the ith item in a RBST of size n; then

$$\text{length of left spine of } L = \text{path length to } i\text{th leaf}$$
$$\text{length of right spine of } R = \text{path length to } (i+1)\text{th leaf}$$

- The cost of the joining phase is the sum of the path lengths to the leaves minus twice the depth of the ith item; the expected cost follows from well-known results

$$\left(2 - \frac{1}{i} - \frac{1}{n+1-i}\right) = \mathcal{O}(1)$$

Joining two RBSTs

- The recursion for Join(T_1, T_2) traverses the rightmost branch (right spine) of T_1 and the leftmost branch (left spine) of T_2
- The trees to be joined are the left and right subtrees L and R of the ith item in a RBST of size n; then

$$\text{length of left spine of } L = \text{path length to } i\text{th leaf}$$
$$\text{length of right spine of } R = \text{path length to } (i+1)\text{th leaf}$$

- The cost of the joining phase is the sum of the path lengths to the leaves minus twice the depth of the ith item; the expected cost follows from well-known results

$$\left(2 - \frac{1}{i} - \frac{1}{n+1-i}\right) = \mathcal{O}(1)$$

Joining two RBSTs

- The recursion for Join(T_1, T_2) traverses the rightmost branch (right spine) of T_1 and the leftmost branch (left spine) of T_2
- The trees to be joined are the left and right subtrees L and R of the ith item in a RBST of size n; then

$$\text{length of left spine of } L = \text{path length to } i\text{th leaf}$$
$$\text{length of right spine of } R = \text{path length to } (i+1)\text{th leaf}$$

- The cost of the joining phase is the sum of the path lengths to the leaves minus twice the depth of the ith item; the expected cost follows from well-known results

$$\left(2 - \frac{1}{i} - \frac{1}{n+1-i}\right) = \mathcal{O}(1)$$

Deletions in RBSTs

Theorem

If T is a random BST that contains the set of keys K, then Delete(T, x) *produces a random BST containing the set of keys $K \setminus \{x\}$.*

Deletions in RBSTs

Theorem

If T is a random BST that contains the set of keys K, then Delete(T, x) *produces a random BST containing the set of keys* $K \setminus \{x\}$.

Corollary

The result of any arbitary sequence of insertions and deletions, starting from an initially empty tree is always a random BST.

Additional remarks

- Arbitrary insertions and deletions yield always random BSTs
- A deletion algorithm for BSTs that preserved randomness was a long standing open problem (10-15 yr)
- Properties of random BSTs have been investigated in depth and for a long time
- Treaps only need to generate a single random number per node (with $\mathcal{O}(\log n)$ bits)
- RBSTs need $\mathcal{O}(\log n)$ calls to the random generator per insertion, and $\mathcal{O}(1)$ calls per deletion (on average)

Additional remarks

- Arbitrary insertions and deletions yield always random BSTs
- A deletion algorithm for BSTs that preserved randomness was a long standing open problem (10-15 yr)
- Properties of random BSTs have been investigated in depth and for a long time
- Treaps only need to generate a single random number per node (with $\mathcal{O}(\log n)$ bits)
- RBSTs need $\mathcal{O}(\log n)$ calls to the random generator per insertion, and $\mathcal{O}(1)$ calls per deletion (on average)

Additional remarks

- Arbitrary insertions and deletions yield always random BSTs
- A deletion algorithm for BSTs that preserved randomness was a long standing open problem (10-15 yr)
- Properties of random BSTs have been investigated in depth and for a long time
- Treaps only need to generate a single random number per node (with $\mathcal{O}(\log n)$ bits)
- RBSTs need $\mathcal{O}(\log n)$ calls to the random generator per insertion, and $\mathcal{O}(1)$ calls per deletion (on average)

Additional remarks

- Arbitrary insertions and deletions yield always random BSTs
- A deletion algorithm for BSTs that preserved randomness was a long standing open problem (10-15 yr)
- Properties of random BSTs have been investigated in depth and for a long time
- Treaps only need to generate a single random number per node (with $\mathcal{O}(\log n)$ bits)
- RBSTs need $\mathcal{O}(\log n)$ calls to the random generator per insertion, and $\mathcal{O}(1)$ calls per deletion (on average)

Additional remarks

- Arbitrary insertions and deletions yield always random BSTs
- A deletion algorithm for BSTs that preserved randomness was a long standing open problem (10-15 yr)
- Properties of random BSTs have been investigated in depth and for a long time
- Treaps only need to generate a single random number per node (with $\mathcal{O}(\log n)$ bits)
- RBSTs need $\mathcal{O}(\log n)$ calls to the random generator per insertion, and $\mathcal{O}(1)$ calls per deletion (on average)

Additional remarks

- Storing subtree sizes for balancing is more useful: they can be used to implement search and deletion by rank, e.g., find the ith smallest element in the tree
- Other operations, e.g., union and intersection are also efficiently supported by RBSTs
- Similar ideas have been used to randomize other search trees, namely, K-dimensional binary search trees (Duch and M., 1998) and quadtrees (Duch, 1999)

Additional remarks

- Storing subtree sizes for balancing is more useful: they can be used to implement search and deletion by rank, e.g., find the ith smallest element in the tree
- Other operations, e.g., union and intersection are also efficiently supported by RBSTs
- Similar ideas have been used to randomize other search trees, namely, K-dimensional binary search trees (Duch and M., 1998) and quadtrees (Duch, 1999)

Additional remarks

- Storing subtree sizes for balancing is more useful: they can be used to implement search and deletion by rank, e.g., find the ith smallest element in the tree
- Other operations, e.g., union and intersection are also efficiently supported by RBSTs
- Similar ideas have been used to randomize other search trees, namely, K-dimensional binary search trees (Duch and M., 1998) and quadtrees (Duch, 1999)

To learn more

C. Martínez and S. Roura.
Randomized binary search trees.
J. Assoc. Comput. Mach., 45(2):288–323, 1998.

R. Seidel and C. Aragon.
Randomized search trees.
Algorithmica, 16:464–497, 1996.

To learn more

J. L. Eppinger.
An empirical study of insertion and deletion in binary search trees.
Comm. of the ACM, 26(9):663—669, 1983.

W. Panny.
Deletions in random binary search trees: A story of errors.
J. Statistical Planning and Inference, 140(8):2335–2345, 2010.

H. M. Mahmoud.
Evolution of Random Search Trees.
Wiley Interscience, 1992.

General References

Ph. Flajolet and R. Sedgewick.
Analytic Combinatorics.
Cambridge University Press, 2008.

D. E. Knuth.
The Art of Computer Programming: Sorting and Searching, volume 3.
Addison-Wesley, 2nd edition, 1998.

C. Pandu Rangan.
Randomized Data Structures, in *Handbook of Data Structures and Applications*.
D.P. Mehta and S. Sahni, editors.
Chapman & Hall, CRC, 2005.

General References (2)

P. Raghavan and R. Motwani.
Randomized Algorithms.
Cambridge University Press, 1995.

R. Sedgewick.
Algorithms in C.
Addison-Wesley, 3rd edition, 1997.

THANK YOU FOR YOUR ATTENTION!