Advanced Data Structures

Conrado Martínez

Univ. Politècnica de Catalunya, Spain

36ª Escuela de Ciencias Informáticas (ECI 2023)
July 24–28, 2023
Buenos Aires, Argentina

Outline of the course

1. Analysis of Algorithms:
   Review of basic concepts. Probabilistic tools. The continuous master theorem. Amortized analysis.
2. Probabilistic & Randomized Dictionaries:
   Review of basic techniques (search trees, hash tables). Randomized binary search trees. Skip lists. Cuckoo hashing. Bloom filters.
3. Priority Queues:
   Review of basic techniques. Binomial queues. Fibonacci heaps. Applications: Dijkstra's algorithm for shortest paths.

Outline of the course

4. Disjoint Sets:
   Union by weight and by rank. Path compression heuristics. Applications: Kruskal's algorithm for minimum spanning trees.
5. Data Structures for String Processing:
   Tries. Patricia. Ternary search trees.
6. Multidimensional Data Structures:
   Associative queries. K-dimensional search trees. Quadtrees.

Part I

Analysis of Algorithms

1 Some Probabilistic Tools

2 The Continuous Master Theorem

3 Amortized Analysis

Linearity of Expectations

For any random variables X and Y, independent or not,

$$\mathbb{E}[aX + bY] = a\,\mathbb{E}[X] + b\,\mathbb{E}[Y]\,.$$

If X and Y are independent then $\mathbb{V}[X + Y] = \mathbb{V}[X] + \mathbb{V}[Y]$.

Indicator variables

It is often useful to introduce indicator random variables $X_i = \mathbb{I}_{A_i}$ such that $X_i = 1$ if the event A_i is true and $X_i = 0$ if the event A_i is false. Let $p_i = \mathbb{P}[\text{event } A_i \text{ happens}]$. Then the X_i are Bernouilli random variables with

$$\mathbb{E}[X_i] = \mathbb{P}[X_i = 1] = p_i.$$

In many cases we can express or bound a random variable X as a linear combination of indicator random variables and then exploit linearity of expectations to derive $\mathbb{E}[X]$.

Union Bound

For any sequence (finite or denumerable) of events $\{A_i\}_{i\geq 0}$

$$\mathbb{P}\left[\bigcup_{i\geq 0} A_i\right] \leq \sum_{i\geq 0} \mathbb{P}[A_i]\,.$$

Markov's Inequality

Theorem

Let X be a positive random variable. For any $a > 0$

$$\mathbb{P}[X > a] \leq \frac{\mathbb{E}[X]}{a}.$$

Markov's Inequality

Proof.

Let A be the event $X > a$. Let $\mathbb{I}_A$ denote the indicator variable of that event. Then

$$\mathbb{P}[X > a] = \mathbb{P}[\mathbb{I}_A = 1] = \mathbb{E}[\mathbb{I}_A],$$

but $a \cdot \mathbb{I}_A < X$, therefore $a\,\mathbb{E}[\mathbb{I}_A] < \mathbb{E}[X]$ and

$$\mathbb{P}[X > a] < \frac{\mathbb{E}[X]}{a}.$$

□

Markov's Inequality

Example

Suppose we throw a fair coin n times. Let H_n denote number of heads in the n throws. We have $\mathbb{E}[H_n] = n/2$. Using Markov's inequality

$$\mathbb{P}[H_n > 3n/4] \leq \frac{n/2}{3n/4} = \frac{2}{3}.$$

In general,

$$\mathbb{P}[X > c \cdot \mathbb{E}[X]] \leq \frac{\mathbb{E}[X]}{c\,\mathbb{E}[X]} = \frac{1}{c}.$$

Chebyshev's Inequality

Theorem

Let X be a positive random variable. For any $a > 0$

$$\mathbb{P}[|X - \mathbb{E}[X]| \geq a] \leq \frac{\mathbb{V}[X]}{a^2}.$$

Corollary

$$\mathbb{P}[|X - \mathbb{E}[X]| \geq c \cdot \sigma_X] \leq \frac{1}{c^2},$$

with $\sigma_X = \sqrt{\mathbb{V}[X]}$, the standard deviation of X.

Chebyshev's Inequality

Proof.

We have

$$\begin{aligned}\mathbb{P}[|X - \mathbb{E}[X]| \geq a] &= \mathbb{P}\Big[(X - \mathbb{E}[X])^2 \geq a^2\Big] \\ &\leq \frac{\mathbb{E}\big[(X - \mathbb{E}[X])^2\big]}{a^2} = \frac{\mathbb{V}[X]}{a^2}.\end{aligned}$$

□

Chebyshev's Inequality

Example

Again $H_n =$ the number of heads in n throws of a fair coin. Since $H_n \sim \text{Binomial}(n, 1/2)$, $\mathbb{E}[H_n] = n/2$ and $\mathbb{V}[H_n] = n/4$. Using Chebyshev's inequality

$$\mathbb{P}[H_n > 3n/4] \leq \mathbb{P}\left[|H_n - \frac{n}{2}| \geq \frac{n}{4}\right] \leq \frac{\mathbb{V}[H_n]}{(n/4)^2} = \frac{4}{n}.$$

Chebyshev's Inequality

Example

The expected number of comparisons $\mathbb{E}[q_n]$ in standard quicksort is $2n \ln n + o(n \log n)$. It can be shown that $\mathbb{V}[q_n] = \left(7 - \frac{2\pi^2}{3}\right) n^2 + o(n^2)$. Hence, the probability that we deviate more than c times from the expected value goes to 0 as $1/\log n$:

$$\mathbb{P}[|q_n - \mathbb{E}[q_n]| \geq c\,\mathbb{E}[q_n]] \leq \frac{\left(7 - \frac{2\pi^2}{3}\right) n^2 + o(n^2)}{2c^2 n^2 \ln^2 n + o(n^2 \log^2 n)}$$
$$= \frac{\left(7 - \frac{2\pi^2}{3}\right)}{2c \ln n} + o\left(1/\log n\right).$$

Jensen's Inequality

Theorem

If f is a convex function then

$$\mathbb{E}[f(X)] \geq f\left(\mathbb{E}[X]\right).$$

Example

For any random variable X, $\mathbb{E}[X^2] \geq (\mathbb{E}[X])^2$, since $f(x) = x^2$ is convex.

Chernoff Bounds

Theorem

Let $\{X_i\}_{i=0}^n$ *be independent Bernouilli trials, with* $\mathbb{P}[X_i = 1] = p_i$. *Then, if* $X = \sum_{i=1}^n X_i$, *and* $\mu = \mathbb{E}[X]$, *we have*

1. $\mathbb{P}[X \leq (1-\delta)\mu] \leq \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{\mu}$, *for* $\delta \in (0,1)$.
2. $\mathbb{P}[X \geq (1+\delta)\mu] \leq \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu}$ *for any* $\delta > 0$.

Chernoff Bounds

Corollary (Corollary 1)

Let $\{X_i\}_{i=0}^n$ be independent Bernouilli trials, with $\mathbb{P}[X_i = 1] = p_i$. Then if $X = \sum_{i=1}^n X_i$, and $\mu = \mathbb{E}[X]$, we have

1. $\mathbb{P}[X \leq (1-\delta)\mu] \leq e^{-\mu\delta^2/2}$, *for* $\delta \in (0,1)$.
2. $\mathbb{P}[X \geq (1+\delta)\mu] \leq e^{-\mu\delta^2/3}$, *for* $\delta \in (0,1]$.

Corollary (Corollary 2)

Let $\{X_i\}_{i=0}^n$ be independent Bernouilli trials, with $\mathbb{P}[X_i = 1] = p_i$. Then if $X = \sum_{i=1}^n X_i$, $\mu = \mathbb{E}[X]$ and $\delta \in (0,1)$, we have

$$\mathbb{P}[|X - \mu| \geq \delta\mu] \leq 2e^{-\mu\delta^2/3}.$$

Chernoff Bounds

Back to an old example: We flip n times a fair coin, we wish an upper bound on the probability of having at least $\frac{3n}{4}$ heads.
Recall Let $H_n \sim \text{Binomial}(n, 1/2)$, then,
$\mu = \mathbb{E}[H_n] = n/2, \mathbb{V}[H_n] = n/4.$

We want to bound $\mathbb{P}\left[H_n \geq \frac{3n}{4}\right]$.

- Markov: $\mathbb{P}\left[H_n \geq \frac{3n}{4}\right] \leq \frac{\mu}{3n/4} = 2/3.$
- Chebyshev: $\mathbb{P}\left[H_n \geq \frac{3n}{4}\right] \leq \mathbb{P}\left[|H_n - \frac{n}{2}| \geq \frac{n}{4}\right] \leq \frac{\mathbb{V}[H_n]}{(n/4)^2} = \frac{4}{n}.$
- Chernoff: Using Cor. 1.2,
  $\mathbb{P}\left[H_n \geq \frac{3n}{4}\right] = \mathbb{P}\left[H_n \geq (1+\delta)\frac{n}{2}\right] \Rightarrow (1+\delta) = \frac{3}{2} \Rightarrow \delta = \frac{1}{2}$
  $\Longrightarrow \mathbb{P}\left[H_n \geq \frac{3n}{4}\right] \leq e^{-\mu\delta^2/3} = e^{-\frac{n}{24}}.$

Example

- If $n = 100$, Cheb. $= 0.04$, Chernoff $= 0.0155$
- If $n = 10^6$, Cheb. $= 4 \times 10^{-6}$, Chernoff $= 2.492 \times 10^{-18095}$

Part I

Analysis of Algorithms

1 Some Probabilistic Tools

2 The Continuous Master Theorem

3 Amortized Analysis

The Continuous Master Theorem

CMT considers divide-and-conquer recurrences of the following type:

$$F_n = t_n + \sum_{0 \le j < n} \omega_{n,j} F_j, \qquad n \ge n_0,$$

for some positive integer n_0, a function t_n, called the toll function, and a sequence of weights $\omega_{n,j} \ge 0$. The weights must satisfy two conditions:

1. $W_n = \sum_{0 \le j < n} \omega_{n,j} \ge 1$ (at least one recursive call).
2. $Z_n = \sum_{0 \le j < n} \frac{j}{n} \cdot \frac{\omega_{n,j}}{W_n} < 1$ (the size of the subinstances is a fraction of the size of the original instance).

The next step is to find a shape function $\omega(z)$, a continuous function approximating the discrete weights $\omega_{n,j}$.

The Continuous Master Theorem

Definition

Given the sequence of weights $\omega_{n,j}$, $\omega(z)$ is a shape function for that set of weights if

1. $\int_0^1 \omega(z)\,dz \geq 1$
2. there exists a constant $\rho > 0$ such that

$$\sum_{0 \leq j < n} \left| \omega_{n,j} - \int_{j/n}^{(j+1)/n} \omega(z)\,dz \right| = \mathcal{O}(n^{-\rho})$$

A simple trick that works very often, to obtain a convenient shape function is to substitute j by $z \cdot n$ in $\omega_{n,j}$, multiply by n and take the limit for $n \to \infty$:

$$\omega(z) = \lim_{n \to \infty} n \cdot \omega_{n,z \cdot n}$$

The Continuous Master Theorem

The extension of many discrete functions to functions in the real domain is immediate, e.g., $j^2 \to z^2$. For binomial numbers one might use the approximation

$$\binom{z \cdot n}{k} \sim \frac{(z \cdot n)^k}{k!}.$$

The continuation of factorials to the real numbers is given by Euler's Gamma function $\Gamma(z)$ and that of harmonic numbers by Ψ function: $\Psi(z) = \frac{d \ln \Gamma(z)}{dz}$.
For instance, in quicksort's recurrence all weights are equal: $\omega_{n,j} = \frac{2}{n}$. Hence a simple valid shape function is $\omega(z) = \lim_{n \to \infty} n \cdot \omega_{n,z \cdot n} = 2$.

The Continuous Master Theorem

Theorem (Roura, 1997)

Let F_n satisfy the recurrence

$$F_n = t_n + \sum_{0 \le j < n} \omega_{n,j} F_j,$$

with $t_n = \Theta(n^a (\log n)^b)$, for some constants $a \ge 0$ and $b > -1$, and let $\omega(z)$ be a shape function for the weights $\omega_{n,j}$. Let $\mathcal{H} = 1 - \int_0^1 \omega(z) z^a \, dz$ and $\mathcal{H}' = -(b+1) \int_0^1 \omega(z) z^a \ln z \, dz$.

The Continuous Master Theorem

Theorem (Roura, 1997; cont'd)

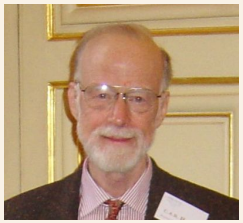
Then

$$F_n = \begin{cases} \frac{t_n}{\mathcal{H}} + o(t_n) & \text{if } \mathcal{H} > 0, \\ \frac{t_n}{\mathcal{H}'} \ln n + o(t_n \log n) & \text{if } \mathcal{H} = 0 \text{ and } \mathcal{H}' \neq 0, \\ \Theta(n^\alpha) & \text{if } \mathcal{H} < 0, \end{cases}$$

where $x = \alpha$ is the unique non-negative solution of the equation

$$1 - \int_0^1 \omega(z) z^x \, dz = 0$$

Example #1: QuickSort

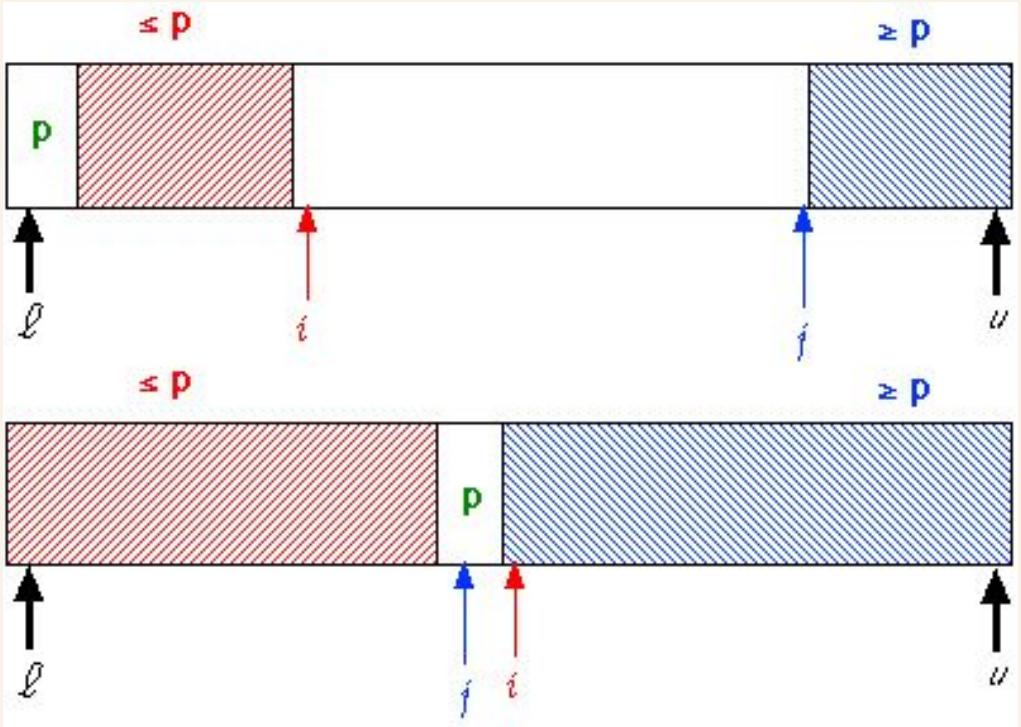


C.A.R. Hoare (1934–)

QUICKSORT (Hoare, 1962) is a sorting algorithm using the divide-and-conquer principle too, but contrary to the previous examples, it does not guarantee that the size of each subinstance will be a fraction of the size of the original given instance.

QuickSort

The basis of *Quicksort* is the procedure PARTITION: given an element p, called the pivot, the (sub)array is rearranged as shown in the picture below.



QuickSort

PARTITION puts the pivot in its final place. Hence it suffices to recursively sort the subarrays to its left and to its right.

procedure QUICKSORT(A, ℓ, u)
Ensure: Sorts subarray $A[\ell..u]$
  if $u - \ell + 1 \leq M$ **then**
    use a simple sorting method, e.g., insertion sort
  else
    PARTITION(A, ℓ, u, k)
    $\triangleright$ $A[\ell..k-1] \leq A[k] \leq A[k+1..u]$
    QUICKSORT$((A, \ell, k-1)$
    QUICKSORT$(A, k+1, u)$

QuickSort

There are many ways to do the partition; not them all are equally good. Some issues, like repeated elements, have to be dealt with carefully. Bentley & McIlroy (1993) discuss a very efficient partition procedure, which works seamlessly even in the presence of repeated elements. Here, we will examine a basic algorithm, which is reasonably efficient.
We will keep two indices i and j such that $A[\ell + 1..i - 1]$ contains elements less than or equal to the pivot p, and $A[j + 1..u]$ contains elements greater than or equal to the pivot p. The two indices scan the subarray locations, i from left to right, j from right to left, until $A[i] > p$ and $A[j] < p$, or until they cross ($i = j + 1$).

QuickSort

procedure PARTITION(A, ℓ, u, k)
Require: $\ell \leq u$
Ensure: $A[\ell..k-1] \leq A[k] \leq A[k+1..u]$

```
i := ℓ + 1; j := u; p := A[ℓ]
while i < j + 1 do
    while i < j + 1 ∧ A[i] ≤ p do
        i := i + 1
    while i < j + 1 ∧ A[j] ≥ p do
        j := j − 1
    if i < j + 1 then
        A[i] :=: A[j]
        i := i + 1; j := j − 1
A[ℓ] :=: A[j]; k := j
```

The Cost of QuickSort

The worst-case cost of QUICKSORT is $\Theta(n^2)$, hence not very attractive. But it only occurs is all or most recursive call one of the subarrays contains very few elements and the other contains almost all. That would happen if we systematically choose the first element of the current subarray as the pivot and the array is already sorted!
The cost of the partition is $\Theta(n)$ and we would have then

$$\begin{aligned} Q(n) &= \Theta(n) + Q(n-1) + Q(0) \\ &= \Theta(n) + Q(n-1) = \Theta(n) + \Theta(n-1) + Q(n-2) \\ &= \cdots = \sum_{i=0}^{n} \Theta(i) = \Theta\left(\sum_{0 \le i \le n} i\right) \\ &= \Theta(n^2). \end{aligned}$$

The Cost of QuickSort

However, on average, there will be a fraction of the elements that are less than the pivot (and will be to its left) and a fraction of elements that are greater than the pivot (and will be to its right). It is for this reason that QUICKSORT belongs to the family of divide-and-conquer algorithms, and indeed it has a good average-case complexity.

The Cost of QuickSort

To analyze the performance of QUICKSORT it only matters the relative order of the elements to be sorted, hence we can safely assume that the input is a permutation of the elements 1 to n. Furthermore, we can concentrate in the number of comparisons between the elements since the total cost will be proportional to that number of comparisons.

The Cost of QuickSort

Let us assume that all $n!$ possible input permutations is equally likely and let q_n be the expected number of comparisons to sort the n elements. Then

$$
\begin{aligned}
q_n &= \sum_{1 \le j \le n} \mathbb{E}[\#\text{ compar.} \mid \text{pivot is the } j\text{-th}] \times \Pr\{\text{pivot is the } j\text{-th}\} \\
&= \sum_{1 \le j \le n} (n - 1 + q_{j-1} + q_{n-j}) \times \frac{1}{n} \\
&= n + \mathcal{O}(1) + \frac{1}{n} \sum_{1 \le j \le n} (q_{j-1} + q_{n-j}) \\
&= n + \mathcal{O}(1) + \frac{2}{n} \sum_{0 \le j < n} q_j
\end{aligned}
$$

Solving QuickSort's Recurrence

We apply CMT to quicksort's recurrence with the set of weights $\omega_{n,j} = 2/n$ and toll function $t_n = n - 1$. As we have already seen, we can take $\omega(z) = 2$, and the CMT applies with $a = 1$ and $b = 0$. All necessary conditions to apply CMT are met. Then we compute

$$\mathcal{H} = 1 - \int_0^1 2z\,dz = 1 - \left. z^2 \right|_{z=0}^{z=1} = 0,$$

hence we will have to apply CMT's second case and compute

$$\mathcal{H}' = -\int_0^1 2z \ln z\,dz = \left. \frac{z^2}{2} - z^2 \ln z \right|_{z=0}^{z=1} = \frac{1}{2}.$$

Finally,

$$\begin{aligned} q_n &= \frac{n \ln n}{1/2} + o(n \log n) = 2n \ln n + o(n \log n) \\ &= 1.386\ldots n \log_2 n + o(n \log n). \end{aligned}$$

Example #2: QuickSelect

The selection problem is to find the j-th smallest element in a given set of n elements. More specifically, given an array $A[1..n]$ of size $n > 0$ and a rank j, $1 \leq j \leq n$, the selection problem is to find the j-th element of A if it were in ascending order.
For $j = 1$ we want to find the minimum, for $j = n$ we want to find the maximum, for $j = \lceil n/2 \rceil$ we are looking for the median, etc.

QuickSelect

The problem can be trivially but inefficiently (because it implies doing much more work than needed!) solved with cost $\Theta(n \log n)$ sorting the array. Another solution keeps an unsorted table of the j smallest elements seen so far while scanning the array from left to right; it has cost $\Theta(j \cdot n)$, and using clever data structures the cost can be improved to $\Theta(n \log j)$. This is not a real improvement with respect the first trivial solution if $j = \Theta(n)$.

QUICKSELECT (Hoare, 1962), also known as FIND and as *one-sided* QUICKSORT, is a variant of QUICKSORT adapted to select the j-th smallest element out of n.

QuickSelect

Assume we partition the subarray $A[\ell..u]$, that contains the elements of ranks ℓ to u, $\ell \leq j \leq u$, with respect some pivot p. Once the partition finishes, suppose that the pivot ends at position k.
Then $A[\ell..k-1]$ contains the elements of ranks ℓ to $(k-1)$ in A and $A[k+1..u]$ contains the elements of ranks $(k+1)$ to u. If $j = k$ we are done since we have found the sought element. If $j < k$ then we need to recursively continue in the left subarray $A[\ell..k-1]$, whereas if $j > k$ then the sought element must be located in the right subarray $A[k+1..u]$.

QuickSelect

Example

We are looking the fourth element ($j = 4$) out of $n = 15$ elements

9	5	10	12	3	1	11	15	7	2	8	13	6	4	14

QuickSelect

Example

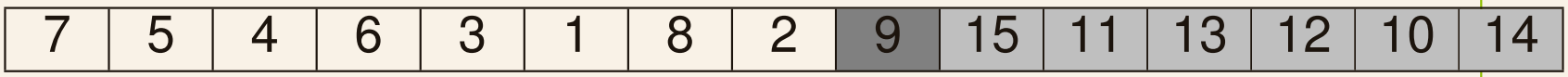
We are looking the fourth element ($j = 4$) out of $n = 15$ elements

9	5	10	12	3	1	11	15	7	2	8	13	6	4	14

QuickSelect

Example

We are looking the fourth element ($j = 4$) out of $n = 15$ elements

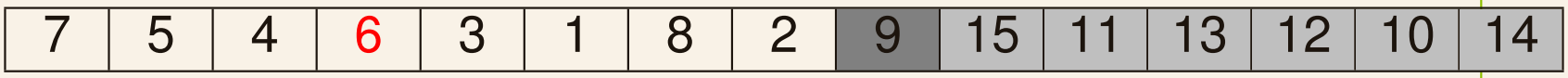


pivot ends at position $k = 9 > j$

QuickSelect

Example

We are looking the fourth element ($j = 4$) out of $n = 15$ elements



QuickSelect

Example

We are looking the fourth element ($j = 4$) out of $n = 15$ elements

pivot ends at position $k = 6 > j$

QuickSelect

Example

We are looking the fourth element ($j = 4$) out of $n = 15$ elements

QuickSelect

Example

We are looking the fourth element ($j = 4$) out of $n = 15$ elements

pivot ends at position $k = 4 = j \Rightarrow$ DONE!

QuickSelect

procedure QUICKSELECT(A, ℓ, j, u)
Ensure: Returns the $(j+1-\ell)$-th smallest element in $A[\ell..u]$, $\ell \leq j \leq u$

```
if ℓ = u then
    return A[ℓ]
PARTITION(A, ℓ, u, k)
if j = k then
    return A[k]
if j < k then
    return QUICKSELECT(A, ℓ, j, k − 1)
else
    return QUICKSELECT(A, k + 1, j, u)
```

QuickSelect

In the worst-case, the cost of QUICKSELECT is $\Theta(n^2)$. However, its average cost is $\Theta(n)$, with the proportionality constant depending on the ratio j/n. Knuth (1971) proved that $C_n^{(j)}$, the expected number of comparisons to find the smallest j-th element among n is:

$$\begin{aligned} C_n^{(j)} = 2\big((n+1)H_n - (n+3-j)H_{n+1-j} \\ - (j+2)H_j + n + 3\big) \end{aligned}$$

The maximum average cost corresponds to finding the median ($j = \lfloor n/2 \rfloor$); then we have

$$C_n^{(\lfloor n/2 \rfloor)} = 2(\ln 2 + 1)n + o(n).$$

QuickSelect

Let us now consider the analysis of the expected cost C_n when j takes any value between 1 and n with identical probability. Then

$$C_n = n + \mathcal{O}(1) + \frac{1}{n} \sum_{1 \leq k \leq n} \mathbb{E}[\text{remaining number of comp.} \mid \text{pivot is the } k\text{-th element}],$$

as the pivot will be the k-th smallest element with probability $1/n$ for all k, $1 \leq k \leq n$.

QuickSelect

The probability that $j = k$ is $1/n$, then no more comparisons are need since we would be done. The probability that $j < k$ is $(k-1)/n$, then we will have to make C_{k-1} comparisons. Similarly, with probability $(n-k)/n$ we have $j > k$ and we will then make C_{n-k} comparisons. Thus

$$\begin{aligned} C_n &= n + \mathcal{O}(1) + \frac{1}{n} \sum_{1 \le k \le n} \frac{k-1}{n} C_{k-1} + \frac{n-k}{n} C_{n-k} \\ &= n + \mathcal{O}(1) + \frac{2}{n} \sum_{0 \le k < n} \frac{k}{n} C_k. \end{aligned}$$

Applying the CMT with the shape function

$$\lim_{n \to \infty} n \cdot \frac{2}{n} \frac{z \cdot n}{n} = 2z$$

we obtain $\mathcal{H} = 1 - \int_0^1 2z^2 \, dz = 1/3 > 0$ and $C_n = 3n + o(n)$.

Part I

Analysis of Algorithms

1 Some Probabilistic Tools

2 The Continuous Master Theorem

3 Amortized Analysis

Amortized Analysis

In amortized analysis we find the (worst/best/average) cost C_n of a sequence of n operations; the amortized cost per operation is

$$a_n = \frac{C_n}{n}$$

Sometimes we compute the cost $C(n_1, \ldots, n_k)$ of a sequence involving n_1 operations of type 1, n_2 operations of type 2, ... The amortized cost is then

$$A(n_1, \ldots, n_k) = \frac{C(n_1, \ldots, n_k)}{n_1 + \cdots + n_k}$$

Amortized Analysis

Amortized cost is interesting when we consider that a sequence of operations must be performed, and some are expensive, but some are cheap; bounding the total cost by n times the cost of the most expensive operation is overly pessimistic.

A first example: Binary counter

Suppose we have a counter that we initialize to 0 and increment it (mod 2) n times. The counter has k bits. How many bit flips are needed?

Theorem

Starting from 0, a sequence of n increments makes $\mathcal{O}(nk)$ bit flips.

A first example: Binary counter

Counter	B[5]	B[4]	B[3]	B[2]	B[1]	B[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
5	0	0	0	1	0	1
6	0	0	0	1	1	0
7	0	0	0	1	1	1
8	0	0	1	0	0	0
9	0	0	1	0	0	1
10	0	0	1	0	1	0
11	0	0	1	0	1	1
12	0	0	1	1	0	0
13	0	0	1	1	0	1
14	0	0	1	1	1	0
15	0	0	1	1	1	1
16	0	1	0	0	0	0

Proof

Any increment flips $\mathcal{O}(k)$ bits. □

Aggregate method

- Determine (an upper bound on) the number $N(c)$ of operations of cost c in the sequence. Then the cost of the sequence is $\leq \sum_{c>0} c \cdot N(c)$.
- An alternative is to count how many operations $N'(c)$ have cost $\geq c$, then the cost of the sequence is $\leq \sum_{c>0} N'(c)$.

Aggregate method

In the binary counter problem, we observe that bit 0 flips n times, bit 1 flips $\lfloor n/2 \rfloor$ times, bit 2 flips $\lfloor n/4 \rfloor$ times, ...

Theorem

Starting from 0, a sequence of n increments makes $\Theta(n)$ bit flips.

Proof

Each increment flips at least 1 bit, thus we make at least $\Omega(n)$ flips. But total cost is $\mathcal{O}(n)$. Indeed

$$\sum_{j=0}^{k-1} \left\lfloor \frac{n}{2^j} \right\rfloor \leq n \sum_{j=0}^{k-1} \frac{1}{2^j} < n \sum_{j=0}^{\infty} \frac{1}{2^j} = n \frac{1}{1-(1/2)} = 2n.$$

□

Accounting method (banker's viewpoint)

We associate "charges" to different operations, these charges may be smaller or larger than the actual cost.

- When the charge or amortized cost $\hat{c}_i$ of an operation is larger than the actual cost c_i then the difference is seen as credits that we store in the data structure to pay for future operations.
- When the amortized cost $\hat{c}_i$ is smaller than c_i the difference must be covered from the credits stored in the data structure.
- The initial data structure D_0 has 0 credits.

Invariant: For all ℓ,

$$\sum_{i=1}^{\ell}(\hat{c}_i - c_i) \geq 0,$$

that is, at all moments, there must be a positive number of credits in the data structure.

Accounting method (banker's viewpoint)

Theorem

The total cost of processing a sequence of n operations starting with D_0 is bounded by the sum of amortized costs.

Proof

$$\text{Invariant} \implies \sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i = \text{total cost.}$$

□

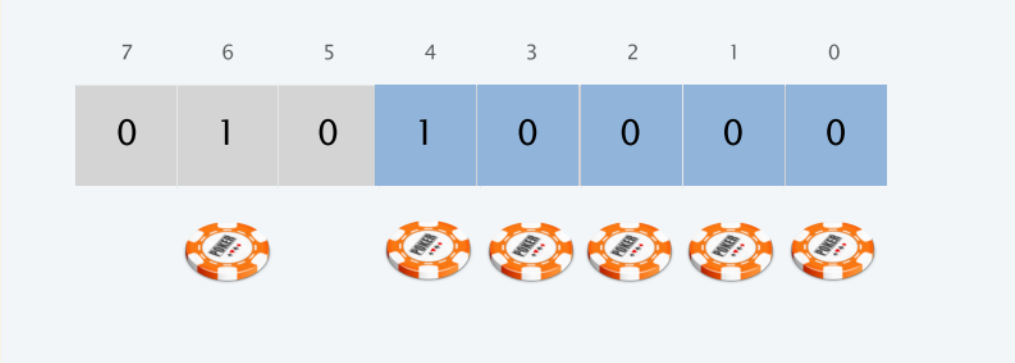
Accounting method (banker's viewpoint)

In the binary counter problem, we charge 2 credits every time we flip a bit from 0 to 1, we pay 1 credit every time we flip a bit from 1 to 0.
We consider that each time we flip a bit from 0 to 1 we store one credit in the data structure and use the other credit to pay the flip. When the bit is flip from 1 to 0, we use the stored credit for that. Thus 1-bits all store 1 credit each, whereas 0-bits do not store credits.

Source: Kevin Wayne
(https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/AmortizedAnalysis.pdf)

Accounting method (banker's viewpoint)

In the binary counter problem, we charge 2 credits every time we flip a bit from 0 to 1, we pay 1 credit every time we flip a bit from 1 to 0.
We consider that each time we flip a bit from 0 to 1 we store one credit in the data structure and use the other credit to pay the flip. When the bit is flip from 1 to 0, we use the stored credit for that. Thus 1-bits all store 1 credit each, whereas 0-bits do not store credits.



Source: Kevin Wayne
(https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/AmortizedAnalysis.pdf)

Accounting method (banker's viewpoint)

In the binary counter problem, we charge 2 credits every time we flip a bit from 0 to 1, we pay 1 credit every time we flip a bit from 1 to 0.
We consider that each time we flip a bit from 0 to 1 we store one credit in the data structure and use the other credit to pay the flip. When the bit is flip from 1 to 0, we use the stored credit for that. Thus 1-bits all store 1 credit each, whereas 0-bits do not store credits.

Source: Kevin Wayne
(https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/AmortizedAnalysis.pdf)

Accounting method (banker's viewpoint)

Theorem

Starting from 0, a sequence of n increments makes $\Theta(n)$ bit flips.

Proof

Every increment flips at least one bit, thus $\sum_i c_i \geq n$. Every increment flips a 0-bit to a 1-bit once (the rightmost 0 in the counter before the increment is the only 0-bit flipped). Hence $\hat{c}_i = 2$ because all the other flips during the i-th increment are from 1-bits to 0-bits, and their amortized cost is 0. Thus

$$\sum_i \hat{c}_i = 2n \geq \sum_i c_i .$$

As the number of credits per bit is ≥ 0 then the number of credits stored at the counter are ≥ 0 at all times, that is, the invariant is preserved. □

Potential method (physicist's viewpoint)

In the potential method we define a potential function Φ that associates a non-negative real to every possible configuration D of the data structure.

1. $\Phi(D) \geq 0$ for all possible configurations D of the data structure.
2. $\Phi(D_0) = 0 \leftarrow$ the potential of the initial configuration is 0

- $D_i =$ configuration of the data structure after i-th operation
- $c_i =$ actual cost of the i-th operation in the sequence
- $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) =$ amortized cost of the i-th operation, it is the actual cost c_i of the operation plus the change in potential $\Delta\Phi_i = \Phi(D_i) - \Phi(D_{i-1})$.

Potential method (physicist's viewpoint)

Theorem

The total cost of processing a sequence of n operations starting with D_0 is bounded by the sum of amortized costs.

Proof

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Delta\Phi_i = \sum_{i=1}^{n} c_i + \sum_{i=1}^{n} (\Phi(D_i) - \Phi(D_{i-1}))$$

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) = \sum_{i=1}^{n} c_i + \Phi(D_n) \geq \sum_{i=1}^{n} c_i,$$

since $\Phi(D_n) \geq 0$ and $\Phi(D_0) = 0$. □

Potential method (physicist's viewpoint)

For the binary counter problem, we take $\Phi(D) =$ number of 1-bits in the binary counter D. Notice that $\Phi(D_0) = 0$ and $\Phi(D) \geq 0$ for all D.

- The actual cost c_i of the i-th increment is $\leq 1 + p$, where p, $0 \leq p \leq k$ is the position of the rightmost 0-bit. We flip the p 1's to the right of the rightmost 0-bit, then the rightmost 0-bit (except when the counter is all 1's and we reset it, then the cost is p).
- The change in potential is $\leq 1 - p$ because we add one 1-bit (flipping the rightmost 0-bit to a 1-bit, except if $p = k$) and we flip p 1-bits to 0-bits, those to the right of the rightmost 0-bit. Hence

$$\hat{c}_i = c_i + \Delta\Phi_i \leq 1 + p + (1 - p) = 2,$$

$$\implies 2n \geq \sum_i \hat{c}_i \geq \sum_i c_i.$$

Stacks with multi-pop

Example

Suppose we have a stack that supports:

- PUSH(x)
- POP(): pops the top of the stack and returns it, stack must be non-empty
- MPOP(k): pops k items, the stack must contain at least k items

The cost of PUSH and POP is $\mathcal{O}(1)$ and the cost of MPOP(k) is $\Theta(k) = \mathcal{O}(n)$ ($n =$ size of the stack), but saying that the worst-case cost of a sequence of N stack operations is $\mathcal{O}(N^2)$ is too pessimistic!

Stacks with multi-pop

Example

Accounting: Assign 2 credits to each PUSH. One is used to do the operation and the other credit to pop (with pop or multi-pop) the element at a later time. The total number of credits in the stack $=$ size of the stack.

- $\hat{c}_{\text{PUSH}} = 2$.
- $\hat{c}_{\text{POP}} = \hat{c}_{\text{MPOP}} = 0$.

$$\implies 2N \geq \sum_{i=1}^{N} \hat{c}_i \geq \sum_{i=1}^{N} c_i.$$

Stacks with multi-pop

Example

Potential: $\Phi(S) = \text{size}(S)$. Then: $\Phi(S_0) = 0$ and $\Phi(S) \geq 0$ for all stacks S.

- $\hat{c}_{\text{PUSH}} = 1 + \Delta\Phi_i = 2.$
- $\hat{c}_{\text{POP}} = 1 + \Delta\Phi_i = 1 + (-1) = 0.$
- $\hat{c}_{\text{MPOP}} = k + \Delta\Phi_i = k + (-k) = 0 \qquad \leftarrow |S_{i-1}| \geq k$

$$\implies 2N \geq \sum_{i=1}^{N} \hat{c}_i \geq \sum_{i=1}^{N} c_i.$$

Dynamic arrays

Example

We often use dynamic arrays (a.k.a. vectors in C++), the array dynamically grows as we add items (using `v.push_back(x)`, say).

A common way to implement dynamic arrays is to allocate an array of some size from dynamic memory; in a given moment, we use only part of the array, we have then

- size: number of elements in the array
- capacity: number of memory cells in the array, size $\leq$ capacity

Dynamic arrays

Example

When a new element has to be added and $n = \text{size} = \text{capacity}$ a new array with double capacity is allocated from dynamic memory, the contents of the old array copied into the new and the old array freed back to dynamic memory, with total cost $\Theta(n)$. The program sets the array name (a pointer) to point to the new array instead of poiting to the old.

This procedure is called resizing, and it implies that a single `push_back` can be very costly if it has to invoke a resizing to accomplish its task.

Dynamic arrays

Example

Cost to fill a dynamic array using n `push_back`'s?

Aggregate:

- The cost c_i of the i-th push_back is $\Theta(1)$ except if $i = 2^k + 1$ for some k, $0 \leq k \leq \log_2(n-1)$.
- When $i = 2^k + 1$, it triggers a resizing with cost $\Theta(i)$.

$$\text{Total cost} = \sum_{i=1}^{n} c_i = \Theta\left(\sum_{i:i\neq 2^k+1} 1 + \sum_{i:i=2^k+1} i\right)$$

$$= n - \Theta(\log n) + \sum_{k=0}^{\lfloor \log_2(n-1) \rfloor} (2^k + 1)$$

$$\leq n - \Theta(\log n) + \Theta(\log n) + (2^{\log_2(n-1)+1} - 1) = \Theta(n).$$

Dynamic arrays

Example

Cost to fill a dynamic array using n `push_back`'s?
Accounting:

- Charge 3 credits to the assignment $v[j] := x$ in which we add x to the first unused array slot j; every `push_back` does it, sometimes a resizing is also needed. Use 1 credit for the assignment, and store the remaining 2 credits in slot j.
- When resizing an array v of size n to an array v' with capacity $2n$, each $j \in v[n/2..n-1]$ stores 2 credits; use one credit for the copying $v'[j] := v[j]$ and use the other credit for the copying of $v[j - n/2]$ to $v'[j - n/2]$

Dynamic arrays

Source: Kevin Wayne
(https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/AmortizedAnalysis.pdf)

Dynamic arrays

Example

Cost to fill a dynamic array using n `push_back`'s?
Accounting: The total number of credits in the dynamic array v is $2 \times (\text{size}(v)/2) = \text{size}(v)$, therefore always ≥ 0.

$$\sum_{i=1}^{n} \hat{c}_i = 3n \geq \sum_{i=1}^{n} c_i.$$

Dynamic arrays

Example

Instead of 3 credits for the assignment $v[j] := x$ we might charge some other constant quantity $c \geq 3$ so that we use 1 credit for the assignment $v[j] := x$ proper, and we store $c - 1$ credits at every used slot j in the upper half of v; these $c - 1$ credits will be used to pay for the copying of $v[j]$ and of $v[j - n/2]$, but also the creation of a unused slot $v'[j + n]$ in the new array and the destruction of $v[j - n/2]$ and $v[j]$ in the old array, if such construction/destruction costs need to be taken into account.

Dynamic arrays

Example

Cost to fill a dynamic array using n `push_back`'s?
Potential:

- When there is no resizing: $c_i = 1$.
- When there is resizing: $c_i = 1 + k \cdot \text{cap}(v_i)$, for some constant k, v_i is the dynamic array after the i-th `push_back`
- $\Phi(v) = 2k\,(2 \cdot \text{size}(v) - \text{cap}(v) + 1)$.

N.B. We will take $k = 1/2$ to simplify the calculations

Dynamic arrays

Example

Cost to fill a dynamic array using n `push_back`'s?

Potential:

- When there is no resizing: $\mathrm{cap}(v_i) = \mathrm{cap}(v_{i-1})$,

$$\begin{aligned}\Phi(v_i) - \Phi(v_{i-1}) &= 2(\mathrm{size}(v_{i-1}) + 1) - \mathrm{cap}(v_{i-1}) + 1 \\ &\quad - \{2 \cdot \mathrm{size}(v_{i-1}) - \mathrm{cap}(v_{i-1}) + 1\} \\ &= 2,\end{aligned}$$

and $\hat{c}_i = c_i + \Delta\Phi_i = 3$.

Dynamic arrays

Example

Cost to fill a dynamic array using n `push_back`'s?
Potential:

- When there is resizing:
  $\mathrm{cap}(v_i) = 2 \cdot \mathrm{cap}(v_{i-1}) = 2 \cdot \mathrm{size}(v_{i-1})$,

  $$\begin{aligned}\Phi(v_i) - \Phi(v_{i-1}) &= 2(\mathrm{size}(v_{i-1}) + 1) - 2 \cdot \mathrm{cap}(v_{i-1}) + 1 \\ &\quad - \{2 \cdot \mathrm{size}(v_{i-1}) - \mathrm{cap}(v_{i-1}) + 1\} \\ &= 2 - \mathrm{cap}(v_{i-1}),\end{aligned}$$

  and

  $$\hat{c}_i = c_i + \Delta\Phi_i = 1 + \mathrm{cap}(v_{i-1}) + 2 - \mathrm{cap}(v_{i-1}) = 3.$$

  Hence $\sum_{i=1}^{n} \hat{c}_i = 3n \geq \sum_{i=1}^{n} c_i$.

Part II

Probabilistic & Randomized Dictionaries

4 Randomized Binary Search Trees

5 Skip Lists

6 Hash Tables
- Separate Chaining
- Open Addressing
- Cuckoo Hashing

7 Bloom Filters

Random BSTs

- In a random binary search tree (built by a random permutation) any of its n elements is the root with probability $1/n$
- Idea: To obtain random BST –independently of any assumption on the distribution of the input– insert a new item in a tree of size n as follows:
  - insert it at the root with probability $1/(n+1)$,
  - otherwise proceed recursively

Random BSTs

- In a random binary search tree (built by a random permutation) any of its n elements is the root with probability $1/n$
- Idea: To obtain random BST –independently of any assumption on the distribution of the input– insert a new item in a tree of size n as follows:
  - insert it at the root with probability $1/(n+1)$,
  - otherwise proceed recursively

Random BSTs

- In a random binary search tree (built by a random permutation) any of its n elements is the root with probability $1/n$
- Idea: To obtain random BST –independently of any assumption on the distribution of the input– insert a new item in a tree of size n as follows:
  - insert it at the root with probability $1/(n+1)$,
  - otherwise proceed recursively

Random BSTs

- In a random binary search tree (built by a random permutation) any of its n elements is the root with probability $1/n$
- Idea: To obtain random BST –independently of any assumption on the distribution of the input– insert a new item in a tree of size n as follows:
  - insert it at the root with probability $1/(n+1)$,
  - otherwise proceed recursively

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 Martínez and Roura (ESA 1996, JACM 1998) use subtree sizes and top-down balancing

Randomized binary search trees


C. Aragon


R. Seidel

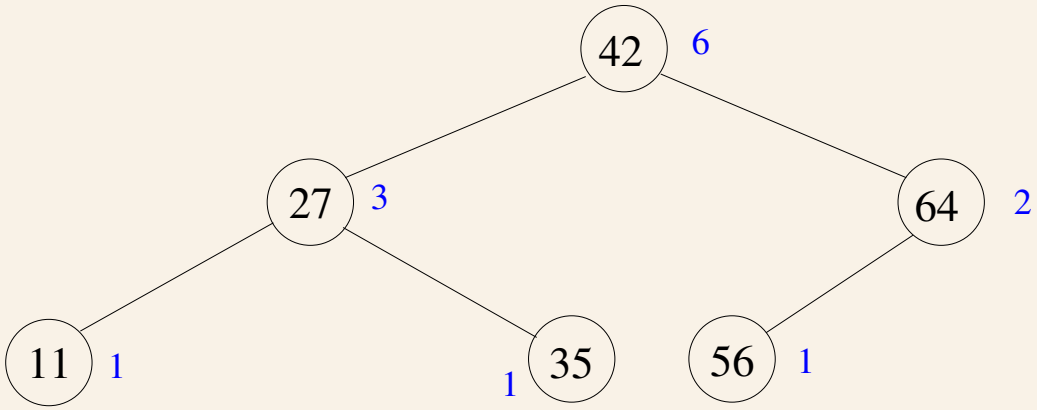

S. Roura

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 Martínez and Roura (ESA 1996, JACM 1998) use subtree sizes and top-down balancing

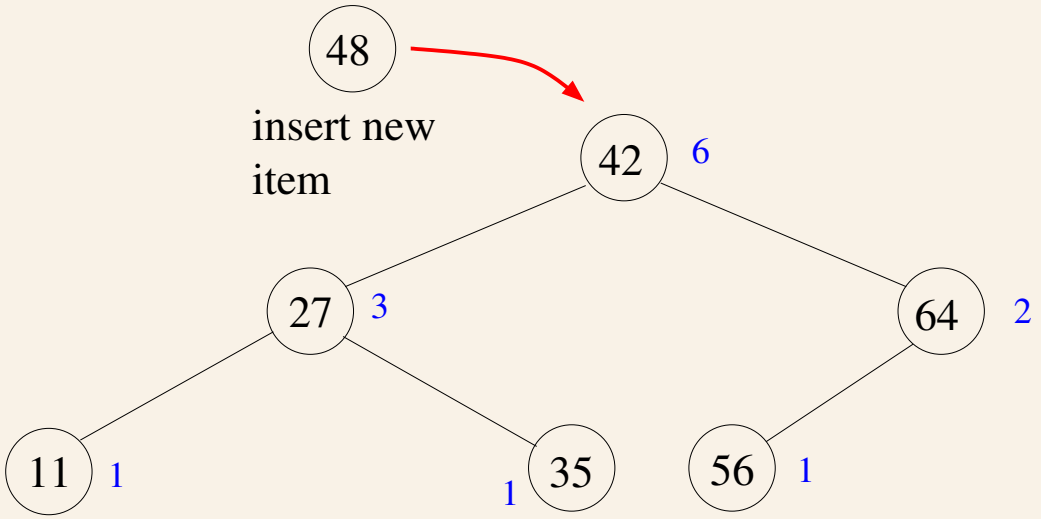
Insertion in a RBST

Inserting an item $x = 48$



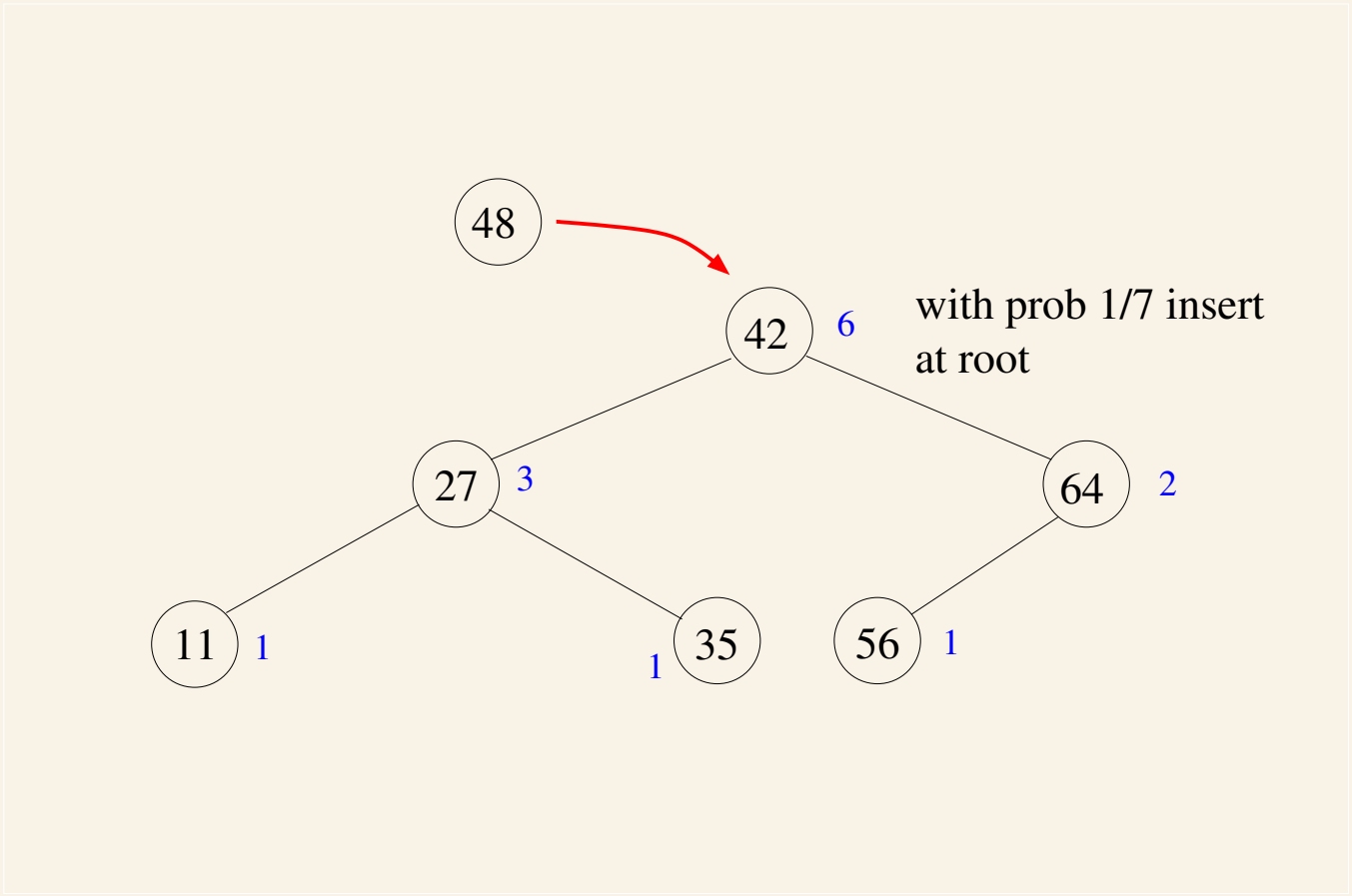
Insertion in a RBST

Inserting an item $x = 48$



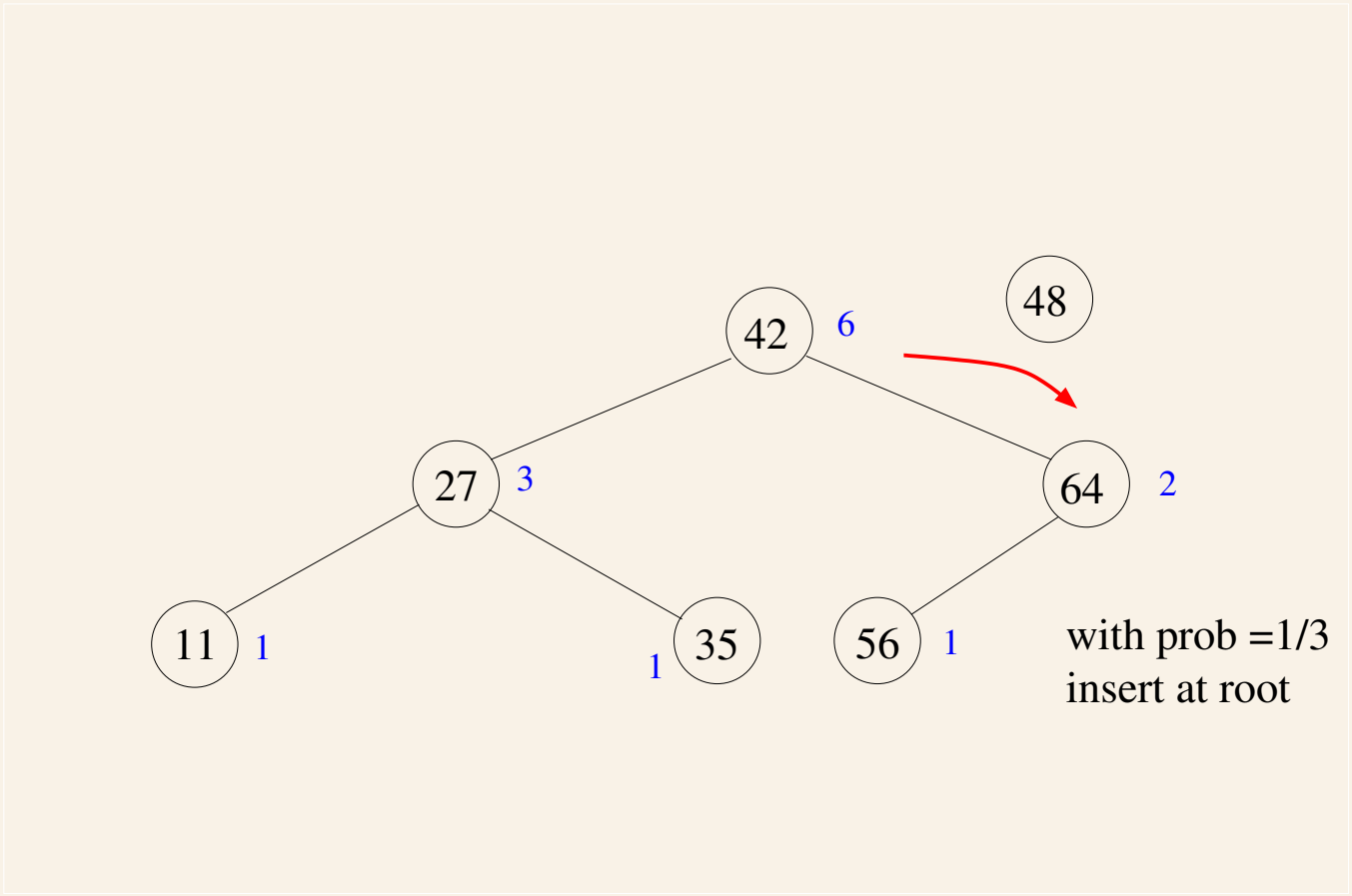
Insertion in a RBST

Inserting an item $x = 48$



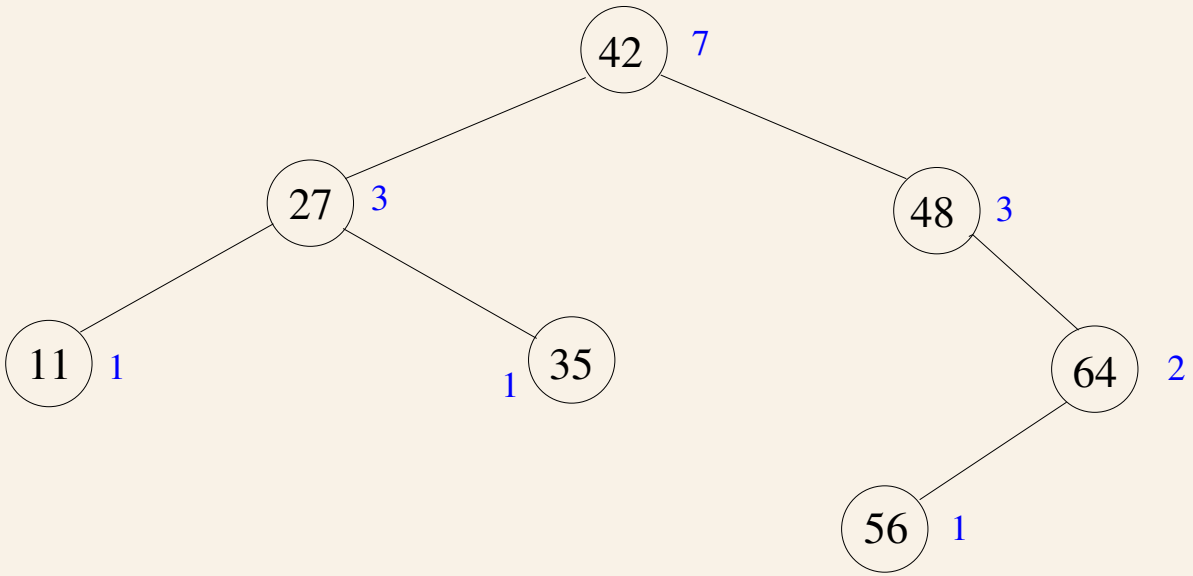
Insertion in a RBST

Inserting an item $x = 48$



Insertion in a RBST

Inserting an item $x = 48$



Insertion in a RBST

```
procedure INSERT(T, k, v)
    n := T → size   ▷ n = 0 if T = □
    if UNIFORM(0, n) = 0 then
        ▷ this will always succeed if T = □
        return INSERT-AT-ROOT(T, k, v)
    if k < T → key then
        T → left := INSERT(T → left, k, v)
    else
        T → right := INSERT(T → right, k, v)
    Update T → size
    return T
```

Insertion in a RBST

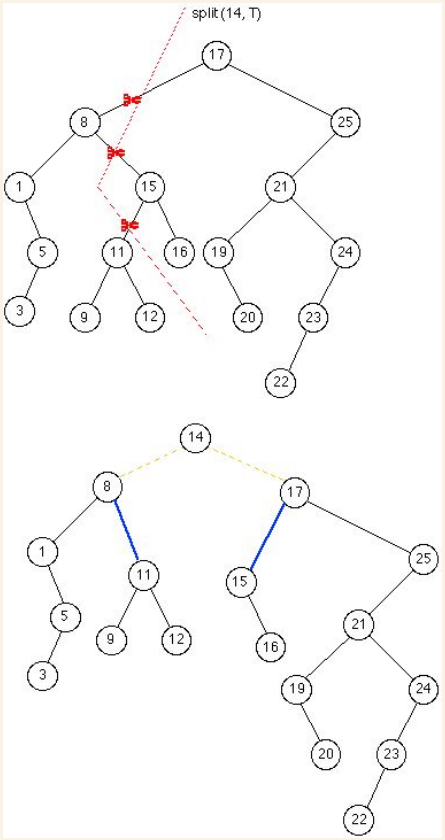
- To insert a new item x at the root of T, we use the algorithm SPLIT that returns two RBSTs T^- and T^+ with element smaller and larger than x, resp.

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

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

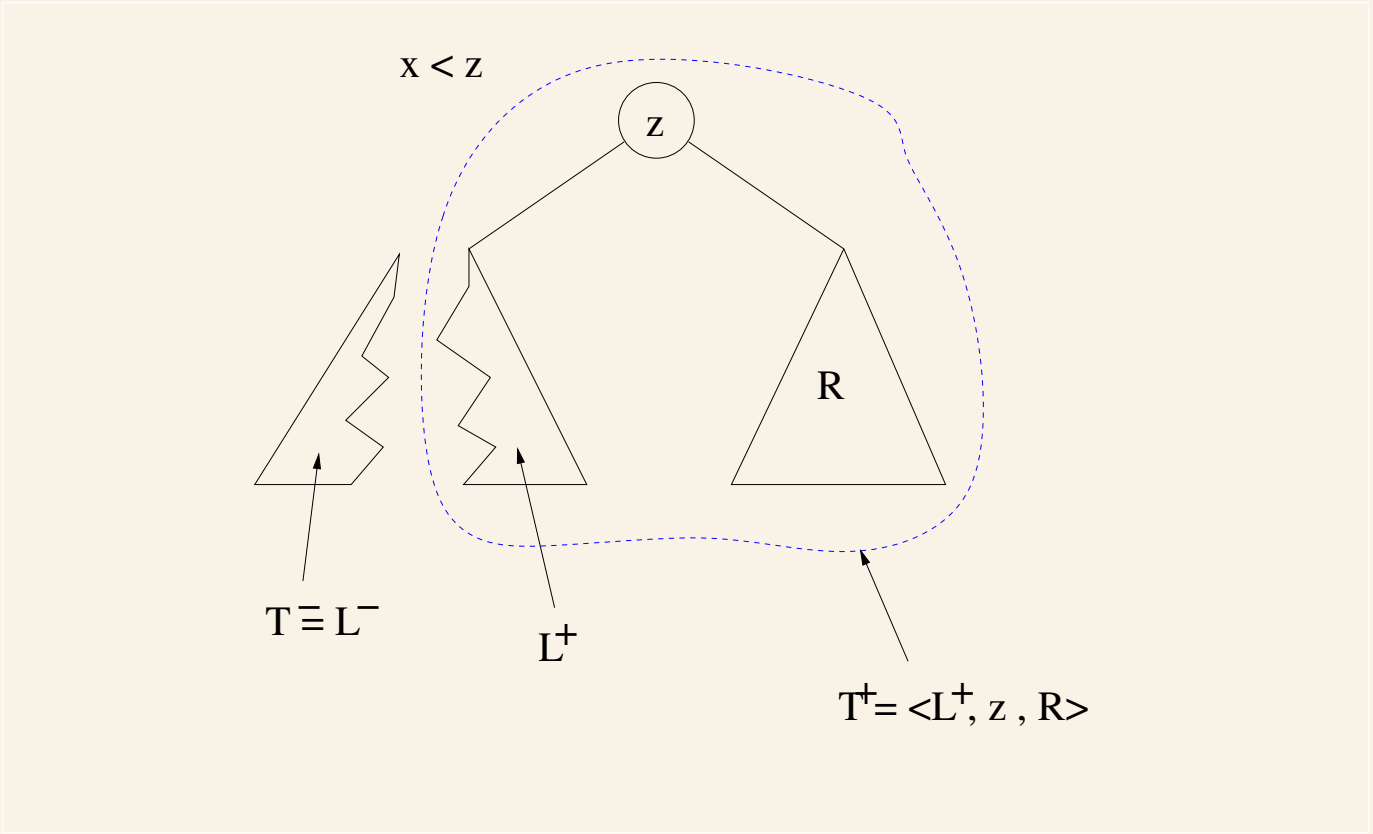
Splitting a RBST

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

```
▷ Pre: k is not present in T
procedure SPLIT(T, k, T⁻, T⁺)
    if T = null then
        T⁻ := null; T⁺ := null; return
    if k < T → key then
        SPLIT(T → left, k, L⁻, L⁺)
        T → left := L⁺
        Update T → size
        T⁻ := L⁻
        T⁺ := T
    else
        ▷ "Symmetric" code for k > T → key
```

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 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:

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

The Cost of Insertions

- The recurrence of $\mathbb{E}[I_n]$:

$$\mathbb{E}[I_n] = 1 + \frac{1}{n} \sum_{1 \leq j \leq n} \frac{j}{n+1} \mathbb{E}[I_{j-1}] + \frac{n-j+1}{n+1} \mathbb{E}[I_{n-j}]$$

- To solve this recurrence the Continuous Master Theorem (Roura, 2001) comes handy
- We need to produce $\mathcal{O}(\log n)$ random numbers on average to insert an item

The Cost of Insertions

- The recurrence of $\mathbb{E}[I_n]$:

$$\mathbb{E}[I_n] = 1 + \frac{1}{n} \sum_{1 \leq j \leq n} \frac{j}{n+1} \mathbb{E}[I_{j-1}] + \frac{n-j+1}{n+1} \mathbb{E}[I_{n-j}]$$

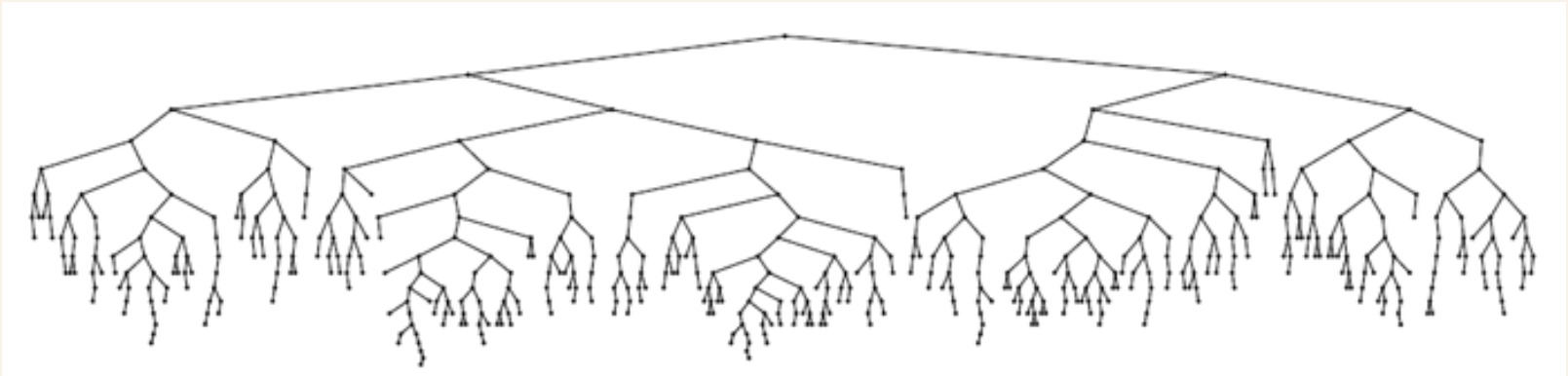
- To solve this recurrence the Continuous Master Theorem (Roura, 2001) comes handy
- We need to produce $\mathcal{O}(\log n)$ random numbers on average to insert an item

The Cost of Insertions

- The recurrence of $\mathbb{E}[I_n]$:

$$\mathbb{E}[I_n] = 1 + \frac{1}{n} \sum_{1 \leq j \leq n} \frac{j}{n+1} \mathbb{E}[I_{j-1}] + \frac{n-j+1}{n+1} \mathbb{E}[I_{n-j}]$$

- To solve this recurrence the Continuous Master Theorem (Roura, 2001) comes handy
- We need to produce $\mathcal{O}(\log n)$ random numbers on average to insert an item



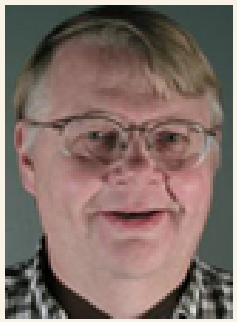
RBST resulting from the insertion of 500 keys in ascending order

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


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

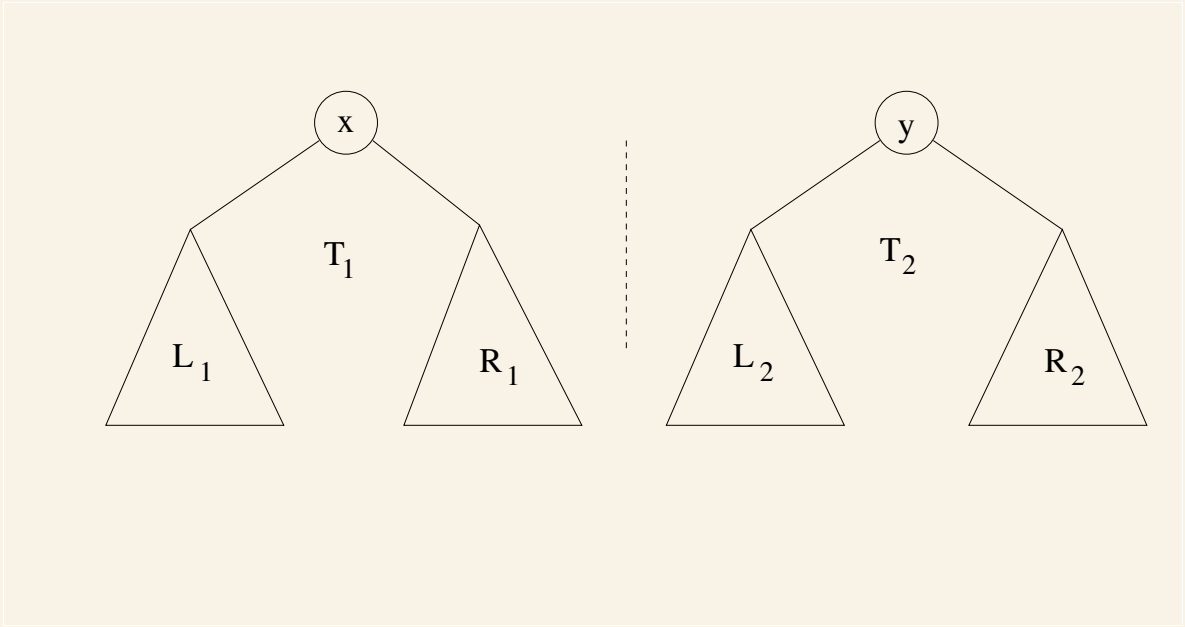
```
procedure DELETE(T, k)
    if T = □ then
        return T
    if k = T → key then
        return DELETE-ROOT(T)
    if x < T → key then
        T → left := DELETE(T → left, k)
    else
        T → right := DELETE(T → right, k)
    Update T → size
    return T
```

Deletions in RBSTs

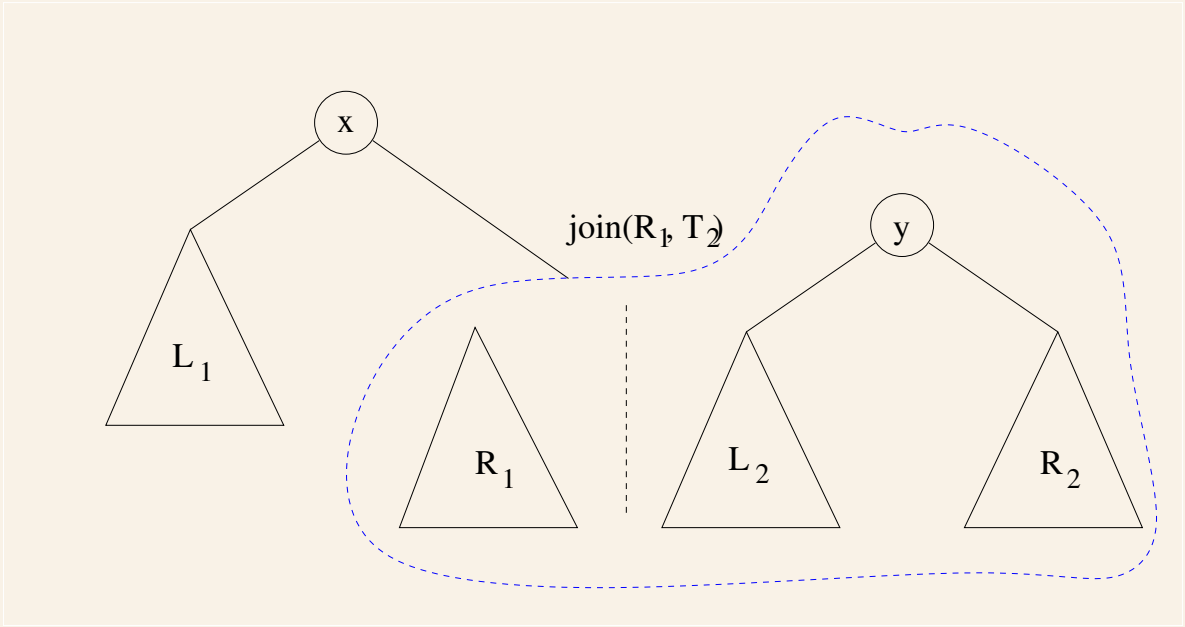
We delete the root using a procedure $\textsc{Join}(T_1, T_2)$. Given two BSTs such that for all $x \in T_1$ and all $y \in T_2$, $x \leq y$, it returns a new BST that contains all the keys in T_1 and T_2.

$$\textsc{Join}(\square, \square) = \square$$
$$\textsc{Join}(T, \square) = \textsc{Join}(\square, T) = T$$
$$\textsc{Join}(T_1, T_2) = ?, \qquad T_1 \neq \square, T_2 \neq \square$$

Joining two BSTs



Joining two BSTs



Joining two BSTs

- If we systematically choose the root of T_1 as the root of $\textsc{Join}(T_1, T_2)$, 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 $T = \textsc{Join}(L, R)$ 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

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

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

- 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 Martínez, 1998) and quadtrees (Duch, 1999) (stay tuned!)

To learn more

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

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

To learn more (2)

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

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

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

Part II

Probabilistic & Randomized Dictionaries

4 Randomized Binary Search Trees

5 Skip Lists

6 Hash Tables
- Separate Chaining
- Open Addressing
- Cuckoo Hashing

7 Bloom Filters

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

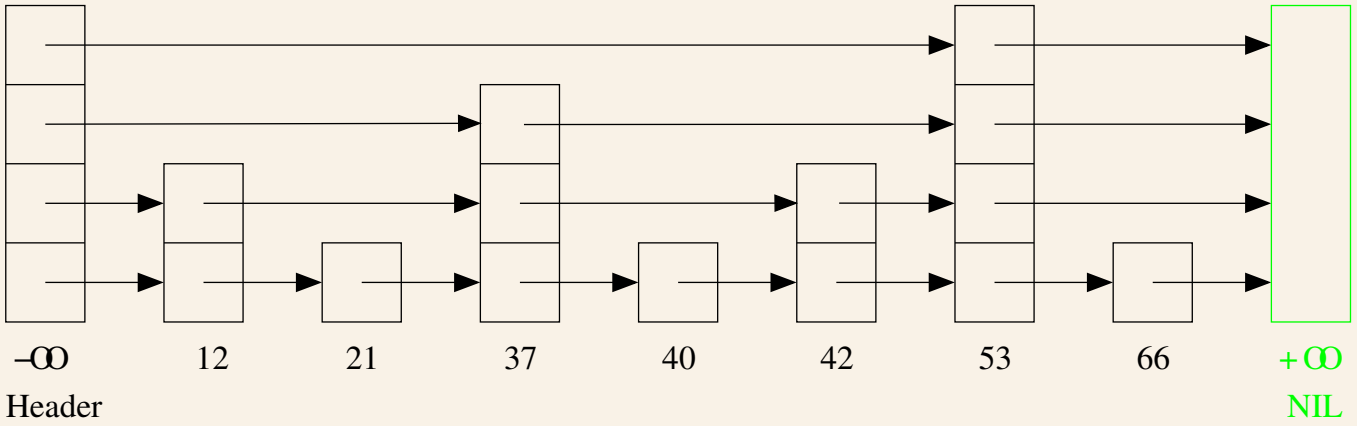
- Skip lists were invented by William Pugh (C. ACM, 1990) as a simple alternative to balanced trees
- The algorithms to search, insert, delete, etc. are very simple to understand and to implement, and they have very good expected performance—independent of any assumption on the input

Skip lists

A skip list S for a set X consists of:

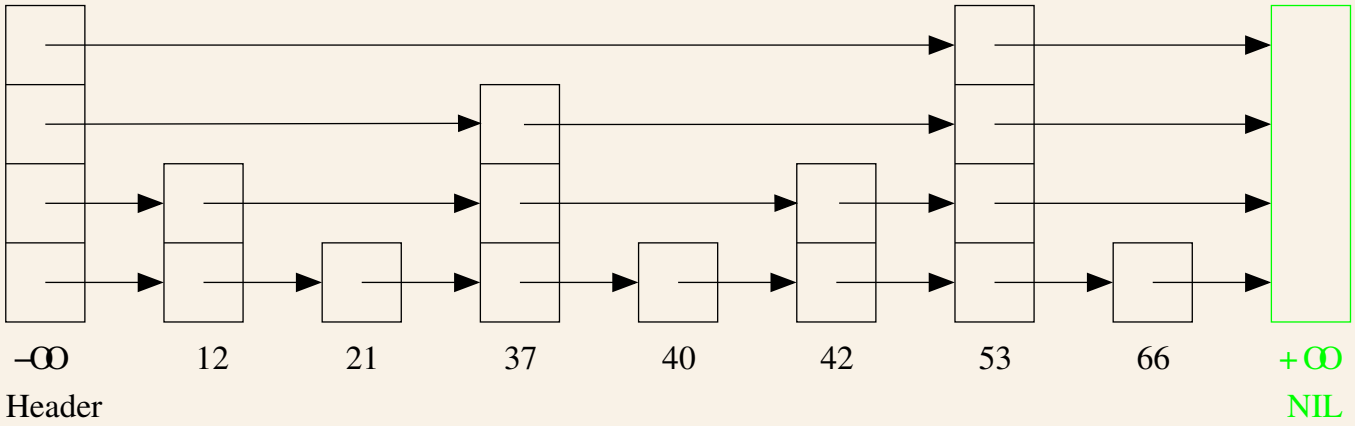
1. A sorted linked list L_1, called level 1, contains all elements of X
2. A collection of non-empty sorted lists L_2, L_3, ..., called level 2, level 3, ... such that for all $i \geq 1$, if an element x belongs to L_i then x belongs to L_{i+1} with probability q, for some $0 < q < 1$, $p := 1 - q$

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

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:

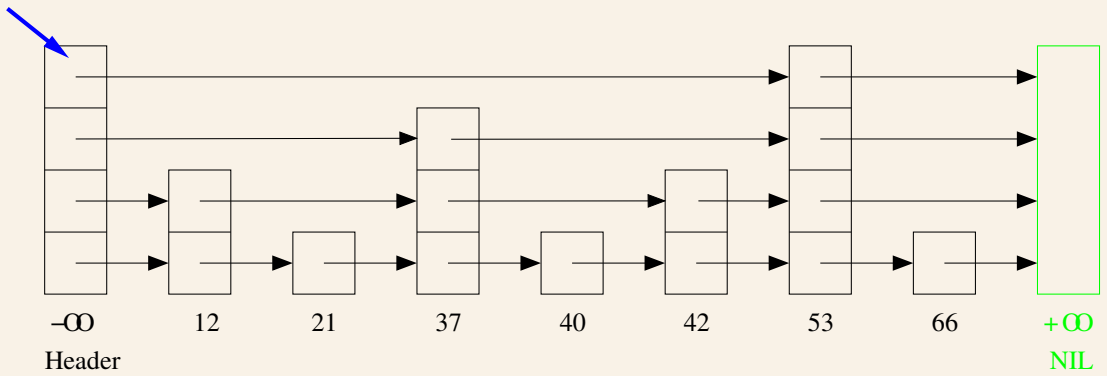
$$\Pr\{\text{height}(x) = k\} = pq^{k-1}, \qquad q = 1 - p$$

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

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

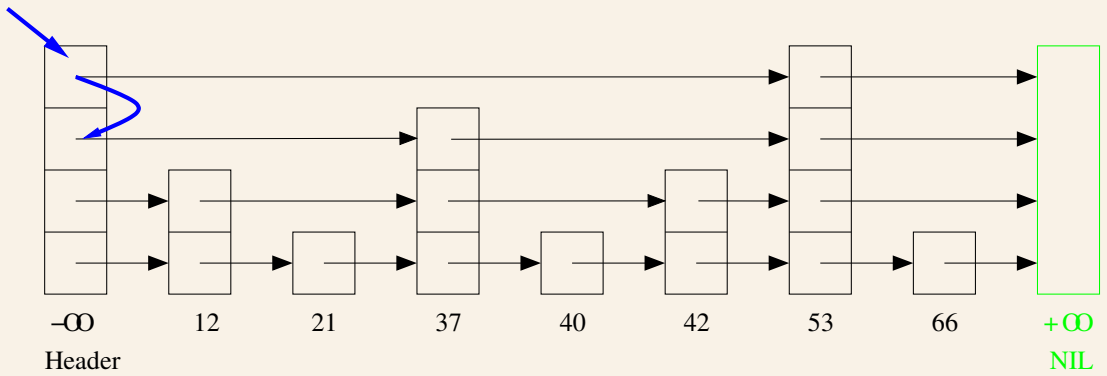
Searching in a skip list

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



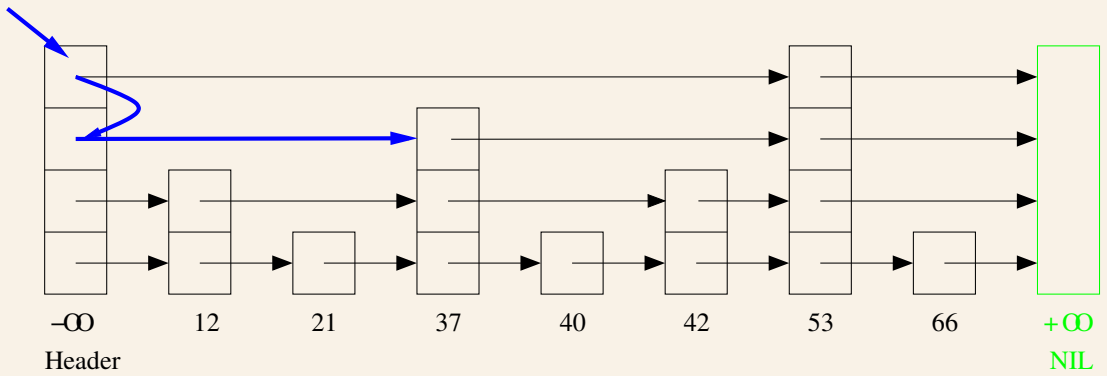
Searching in a skip list

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



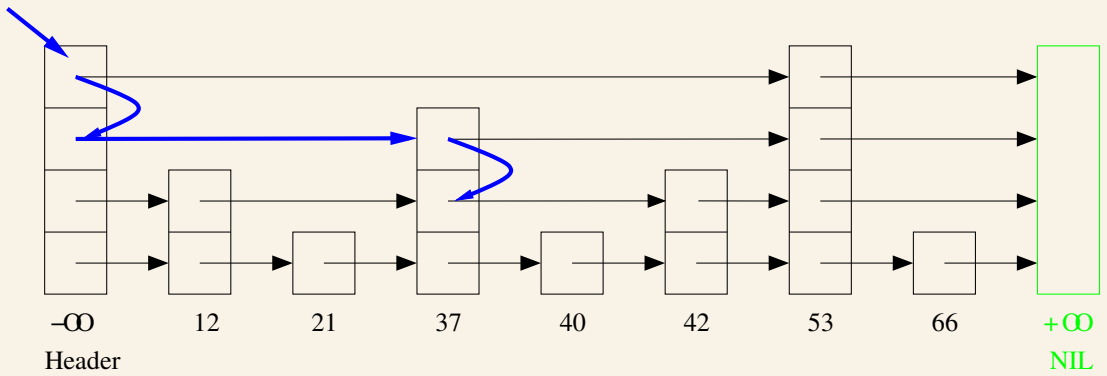
Searching in a skip list

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



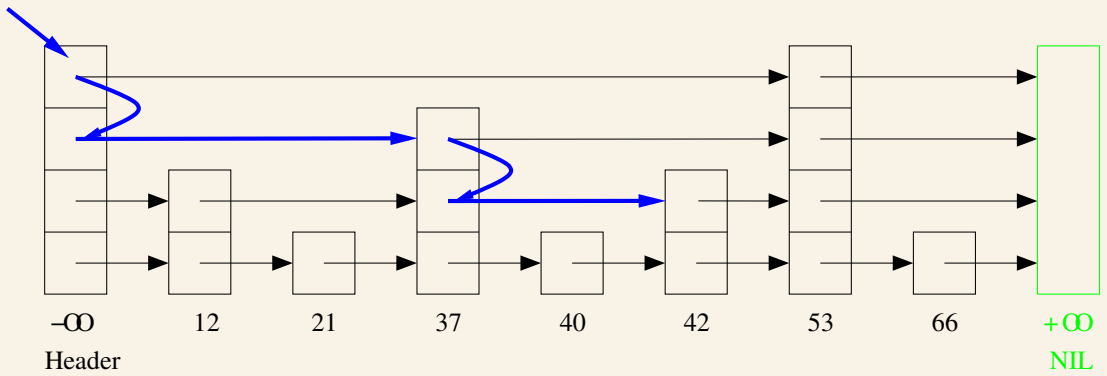
Searching in a skip list

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



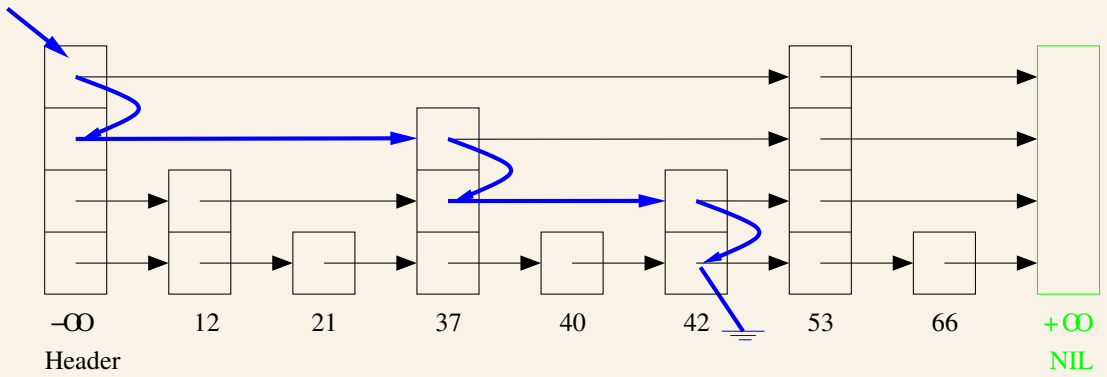
Searching in a skip list

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



Searching in a skip list

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

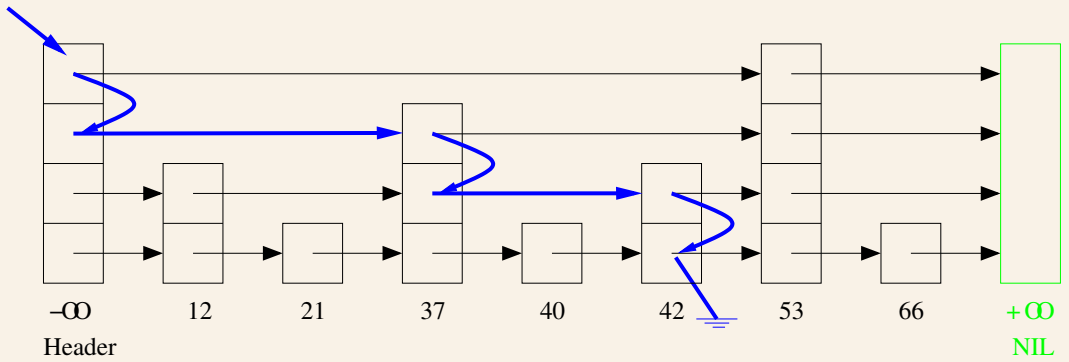


Implementing skip lists

```
▷ Returns pointer to item with key k or null
▷ if not such item exists in the skip list S
procedure SEARCH(k, S)
    p := S.header
    ℓ := S.height
    while ℓ > 0 do
        if p → next[ℓ] = null ∨ k ≤ p → next[ℓ] → key then
            ℓ := ℓ − 1
        else
            p := p → next[ℓ]
    if p → next[1] = null ∨ k ≠ p → next[1] → key then
        ▷ k is not present
        return null
    else  ▷ k is present, return pointer to the node
        return p → next[1]
```

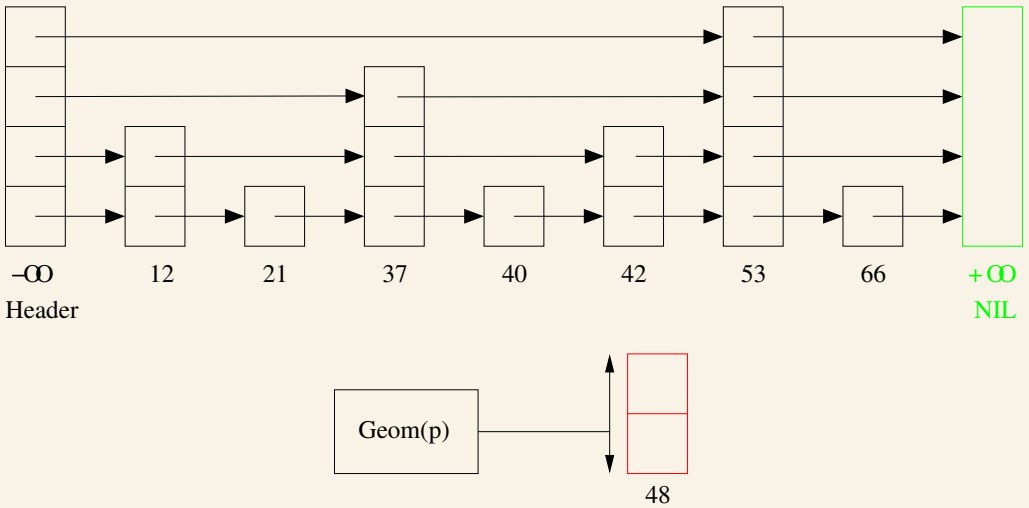
Insertion in a skip list

Inserting an item $x = 48$



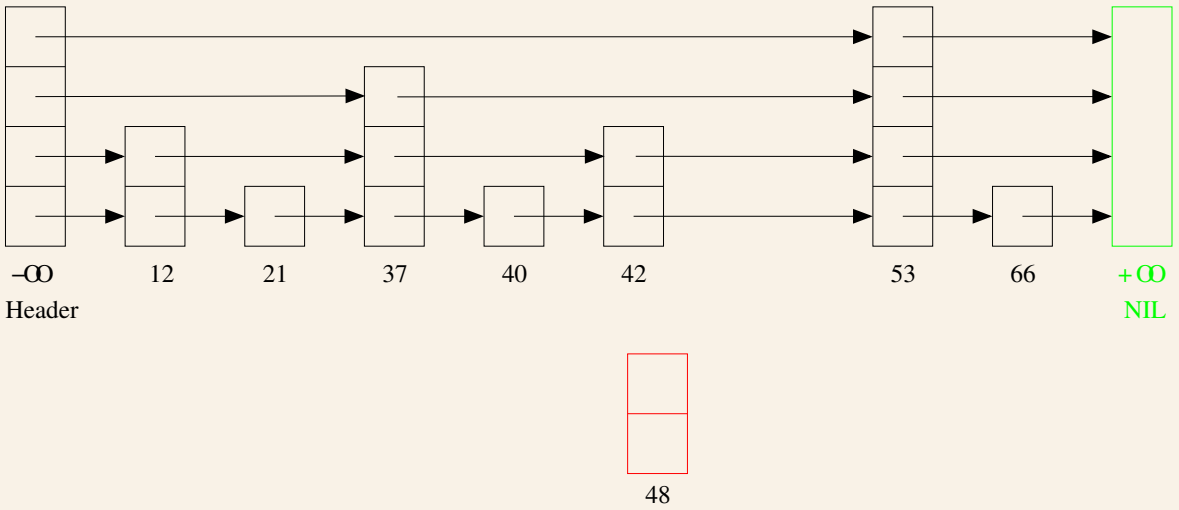
Insertion in a skip list

Inserting an item $x = 48$



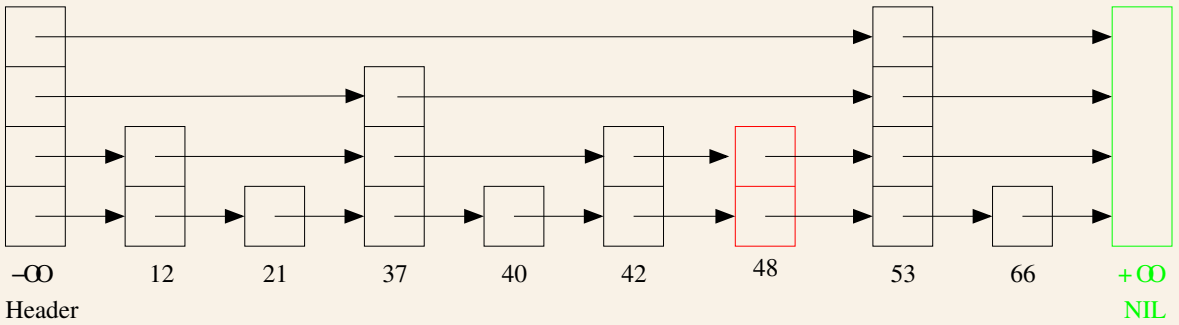
Insertion in a skip list

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

▷ Inserts new item $\langle k, v \rangle$ or
▷ updates value if key k is present in the skip list S

procedure INSERT(k, v, S)
    $p := S$.header; $\ell := S$.height
    create array $pred$ of pointers of size S.height
    for $i := 1$ **to** S.height **do** $pred[i] := S$.header
    while $\ell > 0$ **do**
        if $p \to \text{next}[\ell] = \textbf{null} \vee k \leq p \to \text{next}[\ell] \to \text{key}$ **then**
            ▷ p should be the predecessor of the new item
            ▷ at level ℓ
            $pred[\ell] := p$; $\ell := \ell - 1$
        else
            $p := p \to \text{next}[\ell]$
    ...

Implementing skip lists

procedure INSERT(k, v, S)

...

while ... **do**

▷ loop to locate whether k is present or not

▷ and to determine predecessors at each level

if $p \to \text{next}[1] = \textbf{null} \vee k \neq p \to \text{next}[1] \to \text{key}$ **then**

▷ k is not present

▷ Insert new item, see next slide

else

▷ k is present, update its value

$p \to \text{next}[1] \to \text{value} := v$

Implementing skip lists

3) When k is not present, create a new node with key k and value 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

▷ Insert new item
▷ RNG() generates a random number $U(0,1)$

$h := 1$;
while RNG() $> p$ **do** $h := h + 1$

$nn :=$ **new** NODE(k, v, h)
if $h > S$.height **then**
    Resize S.header and $pred$ with $h - S$.height
    new pointers, all set to **null** and S.header, resp.
    S.height $:= h$

for $i := 1$ **to** h **do**
    $nn \rightarrow$ next$[i] := pred[i] \rightarrow$ next$[i]$
    $pred[i] \rightarrow$ next$[i] := nn$

Other Operations

- Deletions are also very easy to implement
- Ordered raversal of the keys is trivially implemented
- Skip lists 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 raversal of the keys is trivially implemented
- Skip lists 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 raversal of the keys is trivially implemented
- Skip lists 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 raversal of the keys is trivially implemented
- Skip lists 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

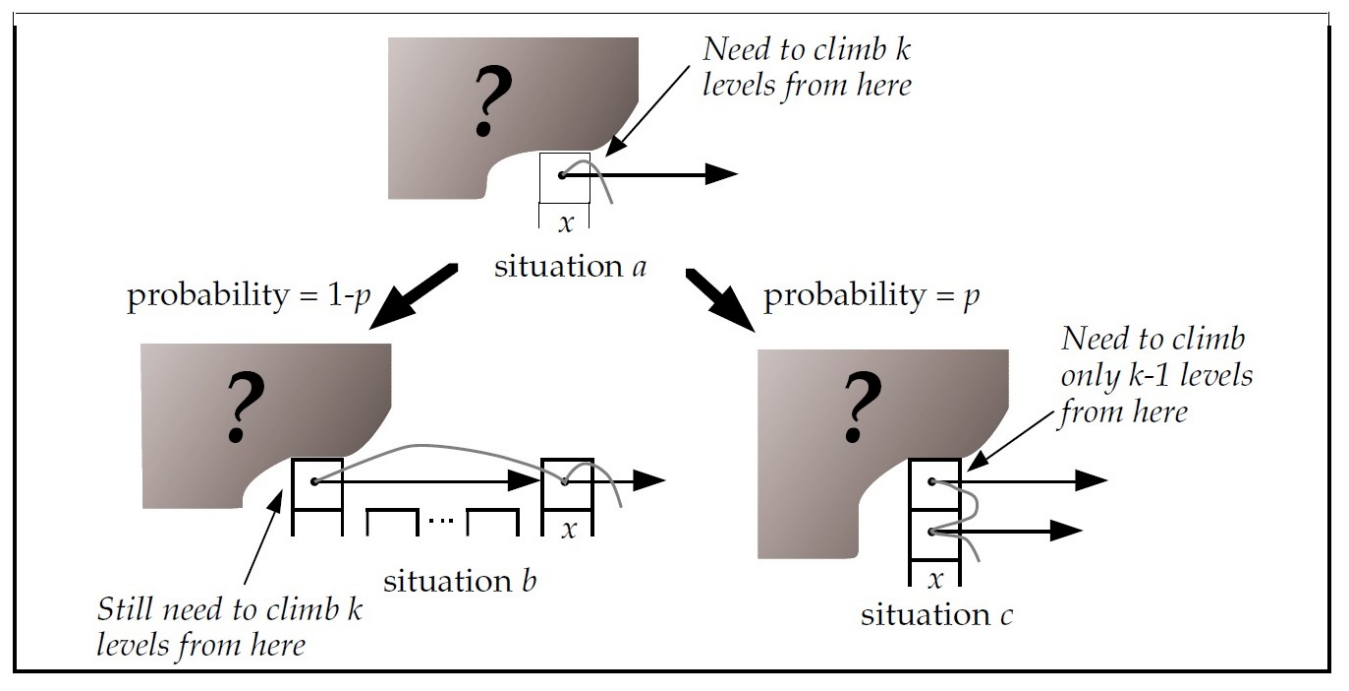


Figure from W. Pugh's *Skip Lists: A Probabilistic Alternative to Balanced Trees* (C. ACM, 1990)—the meaning of p is the opposite of what we have used!

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

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

- the expected value of the height H_n can be bounded as

$$\begin{aligned}\mathbb{E}[H_n] &= \sum_{k\geq 0} \mathbb{P}[H_n > k] = \sum_{0\leq k<L_n} \mathbb{P}[H_n > k] + \sum_{k\geq L_n} \mathbb{P}[H_n > k] \\ &\leq L_n + \sum_{k\geq 0} \mathbb{P}[H_n > L_n + k] = L_n + nq^{L_n} \sum_{k\geq 0} q^k \\ &= L_n + 1/p\end{aligned}$$

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

Performance of skip lists

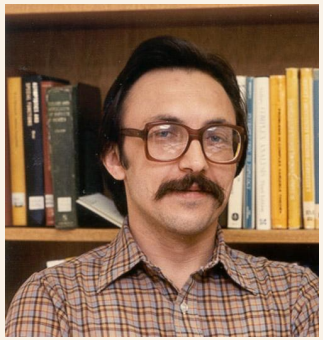
Summing up, the expected path length of a search is

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

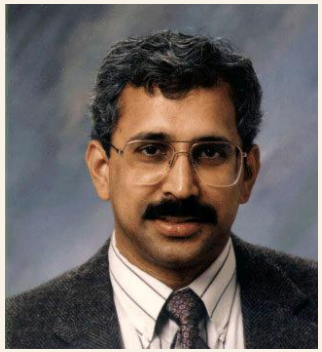
On the other hand, the average number of pointers per node is $1/p$ so there is a trade-off between space and time:

- $p \to 0, q \to 1 \implies$ very tall “nodes”, short horizontal cost
- $p \to 1, q \to 0 \implies$ flat skip lists
- Pugh suggested $p = 3/4$ as a good practical choice; the optimal choice minimizes factor $(q \ln(1/q))^{-1} \implies$ $q = e^{-1} = 0.36\ldots, p = 1 - e^{-1} \approx 0.632\ldots$

Analysis of the height



W. Szpankowski



V. Rego

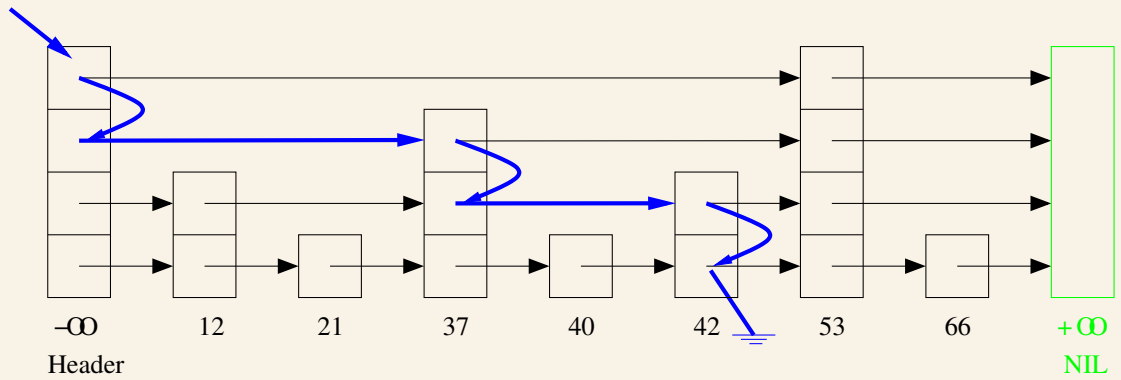
Theorem (Szpankowski and Rego,1990)

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

where $\gamma = 0.577\ldots$ *is Euler's constant and* $\chi(t)$ *a fluctuation of period 1, mean 0 and small amplitude.*

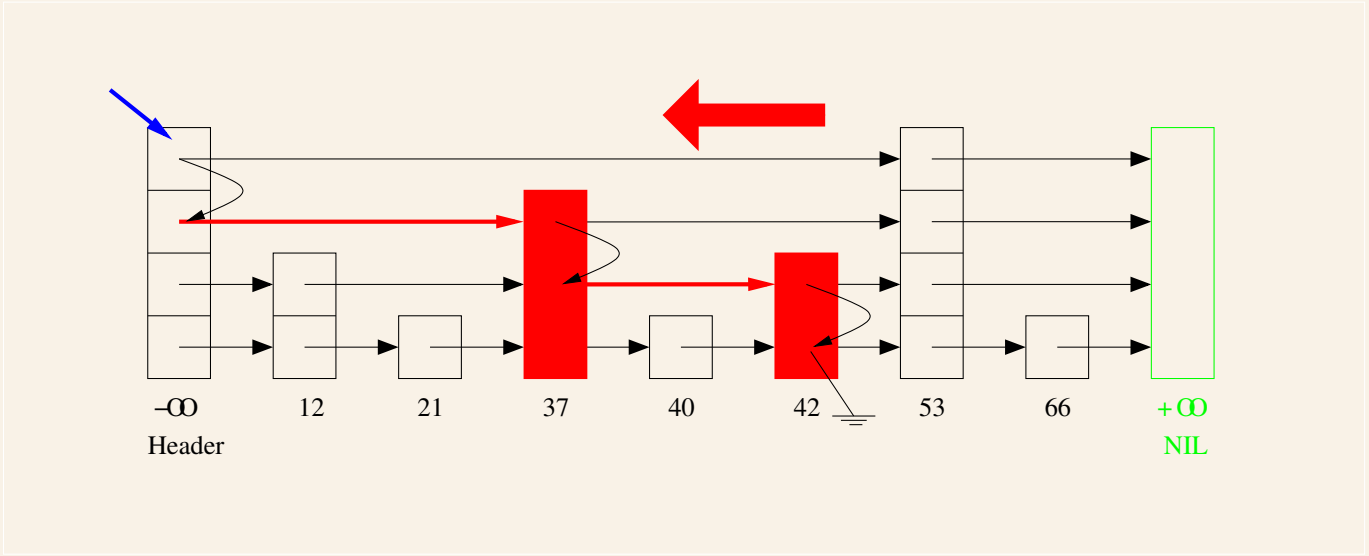
Analysis of the forward cost

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 total forward cost in a random skip list of size n is

$$\mathbb{E}[F_n] = \left(\frac{1}{q} - 1\right) \cdot n \cdot \left(\log_{1/q} n + \frac{\gamma - 1}{\ln(1/q)} - \frac{1}{2} + \frac{1}{\ln(1/q)} \chi(\log_{1/q} n)\right) + \mathcal{O}(\log n),$$

where $\gamma = 0.577\ldots$ is Euler's constant 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

[1] L. Devroye.
A limit theory for random skip lists.
The Annals of Applied Probability, 2(3):597–609, 1992.

[2] P. Kirschenhofer and H. Prodinger.
The path length of random skip lists.
Acta Informatica, 31(8):775–792, 1994.

[3] P. Kirschenhofer, C. Martínez and H. Prodinger.
Analysis of an Optimized Search Algorithm for Skip Lists.
Theoretical Computer Science, 144:199–220, 1995.

To learn more (2)

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

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

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

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

Part II

Probabilistic & Randomized Dictionaries

4 Randomized Binary Search Trees

5 Skip Lists

6 Hash Tables
- Separate Chaining
- Open Addressing
- Cuckoo Hashing

7 Bloom Filters

Hash Tables

A hash table (esp: *tabla de dispersión*) allows us to store a set of elements (or pairs $\langle key, value \rangle$) using a hash function $h : K \Longrightarrow I$, where I is the set of indices or addresses into the table, e.g., $I = [0..M-1]$.

Ideally, the hash function h would map every element (their keys) to a distinct address of the table, but this is hardly possible in a general situation, and we should expect to find collisions (different keys mapping to the same address) as soon as the number of elements stored in the table is $n = \Omega(\sqrt{M})$.

Hash Tables

If the hash function evenly “spreads” the keys, the hash table will be useful as there will be a small number of keys mapping to any given address of the table.

Given two distinct keys x and y, we say that they are synonyms, also that they collide if $h(x) = h(y)$.

A fundamental problem in the implementation of a dictionary using a hash table is to design a collision resolution strategy.

Hash Functions

A good hash function h must enjoy the following properties

1. It is easy to compute
2. It must evenly spread the set of keys K: for all i, $0 \leq i < M$

$$\frac{\#\{k \in K \mid h(k) = i\}}{\#\{k \in K\}} \approx \frac{1}{M}$$

Collision Resolution

Collision resolution strategies can be grouped into two main families. By historical reasons (not very logically) they are called

- Open hashing: separate chaining, 2-way chaining, coalesced hashing, ...
- Open addressing: linear probing, double hashing, quadratic hashing, cuckoo hashing, ...

Separate Chaining

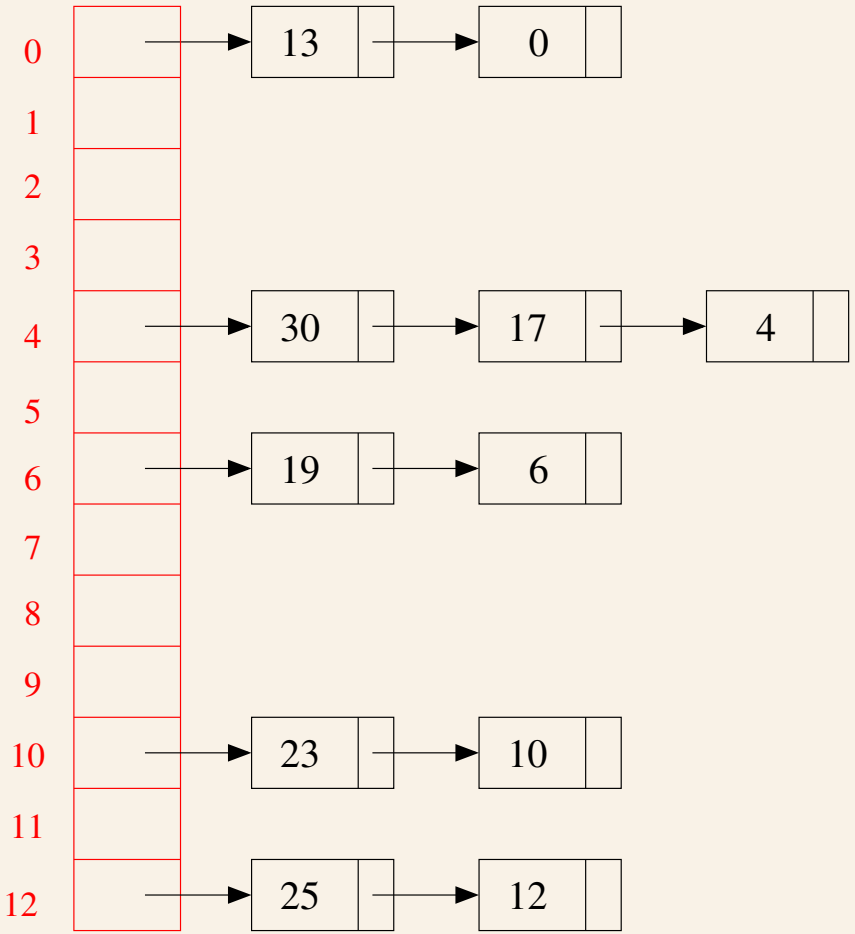
In separate chaining, each slot in the hash table has a pointer to a linked list of synonyms.

```
template <typename Key, typename Value,
          template <typename> class HashFunct = Hash>
class Dictionary {
   ...
private:

   struct node {
      Key _k;
      Value _v;
      ..-
   };
   vector<list<node>> _Thash; // array of linked lists of synonyms
   int _M;         // capacity of the table
   int _n;         // number of elements
   double _alpha_max;  // max. load factor
};
```

Separate Chaining

M = 13 X = { 0, 4, 6, 10, 12, 13, 17, 19, 23, 25, 30}

h (x) = x mod M



Separate Chaining

For insertions, we access the apropriate linked list using the hash function, and scan the list to find out whether the key was already present or not. If present, we modify the associated value; if not, a new node with the pair $\langle key, value \rangle$ is added to the list.
Since the lists contain very few elements each, the simplest and more efficient solution is to add elements to the front. There is no need for double links, sentinels, etc. Sorting the lists or using some other sophisticated data structure instead of linked lists does not report real practical benefits.

Separate Chaining

Searching is also simple: access the apropriate linked list using the hash function and sequentially scan it to locate the key or to report unsuccessful search.

Separate Chaining

```
procedure INSERT(T, k, v)
    if n/M > α_max then
        RESIZE(T)
    i := HASH(k)
    p := __LOOKUP(T, i, k)
    if p = null then
        p := new NODE(k, v)
        p.next := T[i]
        T[i] := p
        n := n + 1
    else
        p.value := v
```

Separate Chaining

```
procedure LOOKUP(T, k, found, v)
    i := HASH(k)
    p := __LOOKUP(T, i, k)
    if p = null then
        found := false
    else
        found := true
        v := p.value
procedure __LOOKUP(T, i, k)
    p := T[i]
    while p ≠ null ∧ p.key ≠ k do
        p := p.next
    return p
```

The Cost of Separate Chaining

Let n be the number of elements stored in the hash table. On average, each linked list contains $\alpha = n/M$ elements and the cost of lookups (either successful or unsuccessful), of insertions and of deletions will be proportional to α. If α is a small constant value then the cost of all basic operations is, on average, $\Theta(1)$. However, it can be shown that the expected length of the largest synonym list is $\Theta(\log n / \log \log n)$.
The value α is called load factor, and the performance of the hash table will be dependent on it.

Open Addressing

In open addressing, synonyms are stored in the hash table. Searches and insertions probe a sequence of positions until the given key or an empty slot is found. The sequence of probes starts in position $i_0 = h(k)$ and continues with $i_1, i_2, \ldots$ The different open addressing strategies use different rules to define the sequence of probes. The simplest one is linear probing:

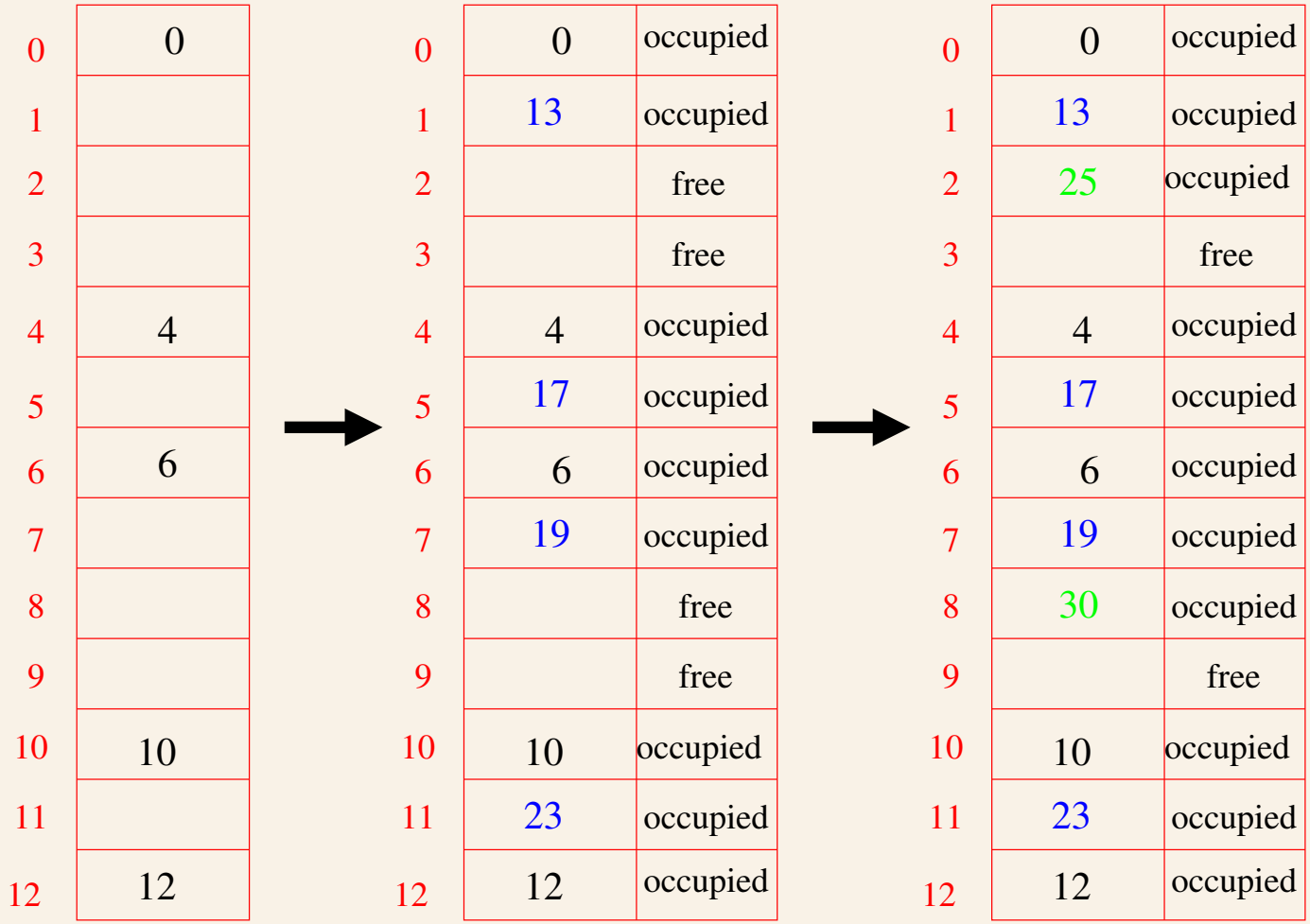
$$i_1 = i_0 + 1, i_2 = i_1 + 1, \ldots,$$

taking modulo M in all cases.

Linear Probing

M = 13 X = { 0, 4, 6, 10, 12, 13, 17, 19, 23, 25, 30}

h (x) = x mod M (incremento 1)



+ {0, 4, 6, 10, 12} + {13, 17, 19, 23} + {25, 30}

Linear Probing

```
procedure LOOKUP(T, k, found, v)
    i := __LOOKUP(T, k)
    if T[i].free then
        found := false
    else
        found := true
        v := T[i].value
procedure __LOOKUP(T, k)
    ▷ we assume at least one free slot
    i := HASH(k)
    while ¬T[i].free ∧ T[i].key ≠ k do
        i := (i + 1) mod M
    return i
```

Other Open Addressing Schemes

As we have already mention different probe sequences give us different open addressing strategies. In general, the sequence of probes is given by

$$\begin{aligned} i_0 &= h(x), \\ i_j &= i_{j-1} \oplus \Delta(j, x), \end{aligned}$$

where $x \oplus y$ denotes $x + y \pmod{M}$.

Other Open Addressing Schemes

1. Linear Probing: $\Delta(j, x) = 1$ (or a constant); $i_j = h(x) \oplus j$
2. Quadratic Hashing: $\Delta(j, x) = a \cdot j + b$; $i_j = h(x) \oplus (Aj^2 + Bj + C)$; constants a and b must be carefully choosen to guarantee that the probe sequence will ultimately explore all the table if necessary
3. Double Hashing: $\Delta(j, x) = h_2(x)$ for a second independent hash function h_2 such that $h_2(x) \neq 0$; $i_j = h(x) \oplus j \cdot h_2(x)$
4. Uniform Hashing: $i_0, i_1, \ldots$ is a random permutation of $\{0, \ldots, M-1\}$
5. Random Probing: $i_0, i_1, \ldots$ is a random sequence such that $0 \leq i_k < M$, for all k, and it contains every value in $\{0, \ldots, M-1\}$ at least once

Other Open Addressing Schemes

Uniform Hashing and Random Probing are completely impractical algorithms; they are interesting as idealizations —they do not suffer from clustering

- Linear Probing suffers primary clustering. There are only M distinct probe sequences, the M circular permutations of $0, 1 \ldots, M-1$
- Quadratic Hashing and other methods with $\Delta(j, x) = f(j)$ (a non-constant function only of j) behave almost as the schemes with *secondary clustering*: two keys such that $h(x) = h(y)$ will probe exactly the same sequence of slots, but if a key x probes i_j in the j-th step and y probes i'_k in the k-th step then i_{j+1} and i'_{k+1} will be probably different
- Double Hashing is even better and generalizations, they exhibit secondary (more generally k-ary clustering) as they depend on $(k-1)$ evaluations of independent hash functions

Other Open Addressing Schemes

- In linear probing two keys will have the same probe sequence with probability $1/M$; in an scheme with secondaty clustering that probability drops to $1/M(M-1)$
- The average performance of schemes with k-ary clustering, $k \geq 2$, is close to that of uniform hashing (no clustering)
- Random probing also approximates well the performance of uniform hashing

The Cost of Open Addressing

We will focus in the following parameters (we assume M is fixed):

1. $\mathcal{U}_n$: number of probes in an unsuccessful search that starts at a random slot in a table with n items
2. $\mathcal{S}_{n,i}$: number of probes in the successful search of the i-th inserted item when the table contains n items, $1 \leq i \leq n$

We will actually be more interested in $\mathcal{S}_n := \mathcal{S}_{n,U_n}$ where U_n is a random uniform value in $\{1, \ldots, n\}$, that is, $\mathcal{S}_n$ is the cost of a successful search of a random item in a table with n items

The Cost of Open Addressing

- The cost of the $(n+1)$-th insertion is given by $\mathcal{U}_n$
- With the FCFS insertion policy, an item will be inserted where the unsuccessful search terminated and never be moved from there, hence

$$\mathcal{S}_{n,i} \stackrel{\mathcal{D}}{=} \mathcal{U}_{i-1}$$

where $\stackrel{\mathcal{D}}{=}$ denotes equal distribution

The Cost of Open Addressing

Consider random probing. What is $U_n = \mathbb{E}[\mathcal{U}_n]$?
With one probe we land in an empty slot and we are done. Probability is $(1-\alpha)$. If the first place is occupated, probability α, we probe a second slot, which is empty with probability $1-\alpha$. And so on. Thus

$$\begin{aligned} U_n &= 1 \times (1-\alpha) + 2 \times \alpha \cdot (1-\alpha) + 3 \times \alpha^2 \cdot (1-\alpha) \\ &= \sum_{k>0} k\alpha^{k-1} \cdot (1-\alpha) = (1-\alpha) \sum_{k>0} \frac{d(\alpha^k)}{d\alpha} \\ &= (1-\alpha) \frac{d}{d\alpha} \sum_{k>0} \alpha^k = \frac{1}{1-\alpha}. \end{aligned}$$

The Cost of Open Addressing

And for the expected successful search we have

$$S_n = \mathbb{E}[\mathcal{S}_n] = \frac{1}{n} \sum_{1 \leq i \leq n} \mathbb{E}[\mathcal{S}_{n,i}] = \frac{1}{n} \sum_{1 \leq i \leq n} \mathbb{E}[\mathcal{U}_{i-1}] = \frac{1}{n} \sum_{1 \leq i \leq n} U_{i-1}$$

Using Euler-McLaurin

$$S_n = \frac{1}{\alpha M} \sum_{1 \leq i \leq n} U_{i-1} = \frac{1}{\alpha} \int_0^{\alpha} \frac{1}{1-\beta} \, d\beta = \frac{1}{\alpha} \ln \left(\frac{1}{1-\alpha} \right)$$

The Cost of Open Addressing

The actual expected costs of hashing with uniform hashing (and thus of quadratic hashing, double hashing) are slightly different from those of random probing, a few small corrections must be introduced:

- $U_n = 1/(1-\alpha) - \alpha - \ln(1-\alpha)$
- $S_n = 1/\alpha \int_0^\alpha U(\beta)\, d\beta = 1 - \alpha/2 - \ln(1-\alpha) \qquad (*)$

The Cost of Open Addressing

The analysis of linear probing turns out to be more challenging than one could think at first.

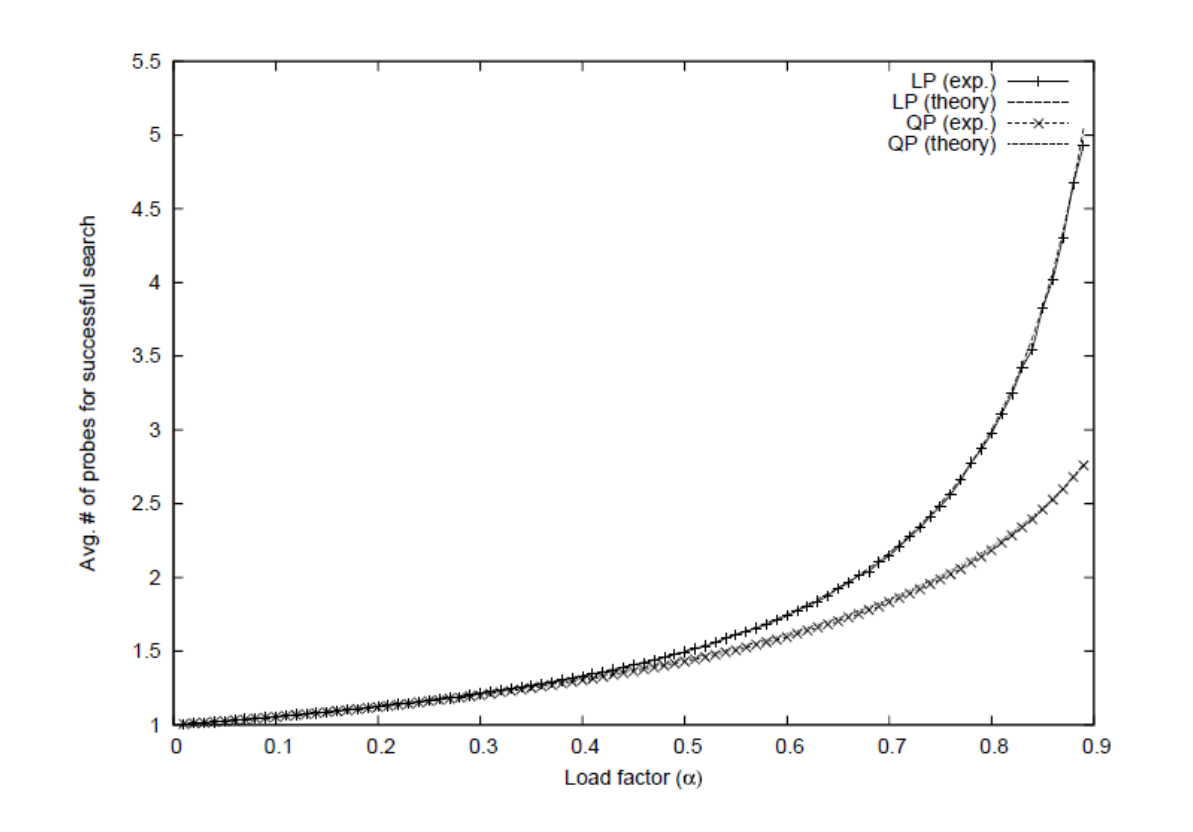
- The average cost of unsuccessful search is

$$U_n = \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

- The average cost of successful search is

$$S_n = \frac{1}{\alpha}\int_0^{\alpha} U(\beta)\, d\beta = \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \qquad (**)$$

The Cost of Open Addressing



Comparison of experimental vs. theoretical expected cost of successful search in linear probing and quadratic hashing

Cuckoo Hashing

Rasmus Pagh

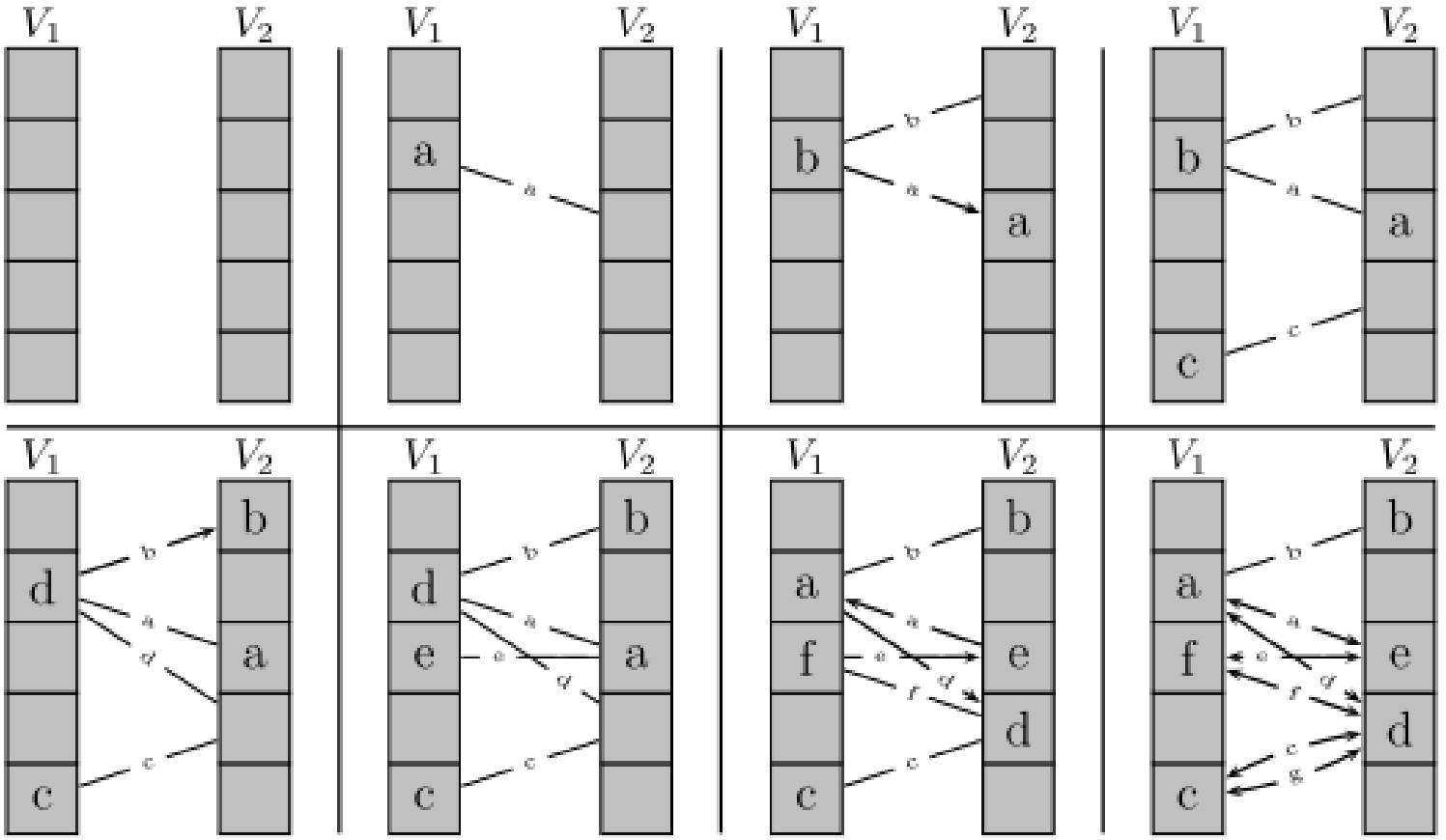
Flemming F. Rodler

In cuckoo hashing we have **two** tables T_1 and T_2 of size M each, and two hash functions $h_1, h_2 : 0 \to M - 1$.
The worst-case complexity of searches and deletions in a cuckoo hash table is $\Theta(1)$. We can insert in such table $n < M$ items: the load factor $\alpha = n/2M$ must be strictly less than 1/2 in order to guarantee constant expected time for insetions.

Cuckoo Hashing

To insert a new item x, we probe slot $T_1[h_1(x)]$, if it is empty, we put x there and stop. Otherwise if y sits already in that slot, then x kicks out y— x is put in $T_1[h_1(x)]$ and y moves to $T_2[h_2(y)]$.
If that slot in T_2 is empty, we're done, but if some z occupies $T_2[h_2(y)]$, then y is put in its second "nest" and z is kicked out to $T_1[h_1(z)]$, and so on.
These "kicks out" give the name to this strategy. If this procedure succeeds to insert n keys then each key x **can only appear in one of its two nests**: $T_1[h_1(x)]$ or $T_2[h_2(x)]$, nowhere else!

Cuckoo Hashing



	a	b	c	d	e	f	g
h_1	2	2	5	2	3	3	5
h_2	3	1	4	4	3	4	4

Cuckoo Hashing

```
procedure LOOKUP(T, k, found, v)
    n1 := T_1[h_1(k)]
    if ¬n1.free ∧ n1.key = k then
        found := true; v := n1.value
    else
        n2 := T_2[h_2(k)]
        if ¬n2.free ∧ n2.key = k then
            found := true; v := n2.value
        else
            found := false
```

Only two probes are necessary in the worst-case! To delete we localate with ≤ 2 probes the key to remove and mark the slot as free.

Cuckoo Hashing

- The insertion of an item x can fail because we enter in an infinite loop of items each kicking out the next in the cycle . . .
- The solution to the problem: nuke the table! Draw two new hash functions, and rehash everything again with the two new functions.
- This rehashing is clearly quite costly; moreover, we don't have a guarantee that the new functions will succeed where the old failed!

We will see, however, that insertion has expected amortized constant cost, or equivalently, that the expected cost of n insertions is $\Theta(n)$

Cuckoo Hashing

```
procedure INSERT(T = ⟨T_1, T_2⟩, k, v)
    if k ∈ T then ▷ update v and return
    else
        if n = M − 1 then ▷ M = |T_1| = |T_2|
            RESIZE(T)
            REHASH(T) ▷ can't insert ≥ M elements
        x := NODE(k, v); x.free = false
        for i := 1 to MAXITER(n, M) do
            ▷ for example, MAXITER(n, M) = 2n
            x :=: T_1[h_1(k)]
            if x.free then return
            x :=: T_2[h_2(k)]
            if x.free then return
        ▷ Insertion failed! pick new functions h_1 and h_2
        REHASH(T)
        INSERT(T, k, v) ▷ retry with the new functions
```

Cuckoo Hashing

We say that an insertion is *good* if it does not run into a infinite loop (our implementation protects from ∞-loops by bounding the number of iterations).
A "high-level analysis" of the cost of insertions follows from:

1. The expected number of steps/iterations in a good insertion is $\Theta(1)$
2. The probability that the insertion of an item is not good is $O(1/n^2)$

Cuckoo Hashing

3. By the union bound, the probability that we fail to make n consecutive good insertions is $O(1/n)$
4. The expected total cost of making n good insertions—conditioned on the event that we can make them—is $n \times \Theta(1) = \Theta(n)$

Cuckoo Hashing

1. The expected number of times we need to rehash a set of n items until we can insert all with good insertions is given by a geometric r.v. with probability of success $1 - O(1/n)$:

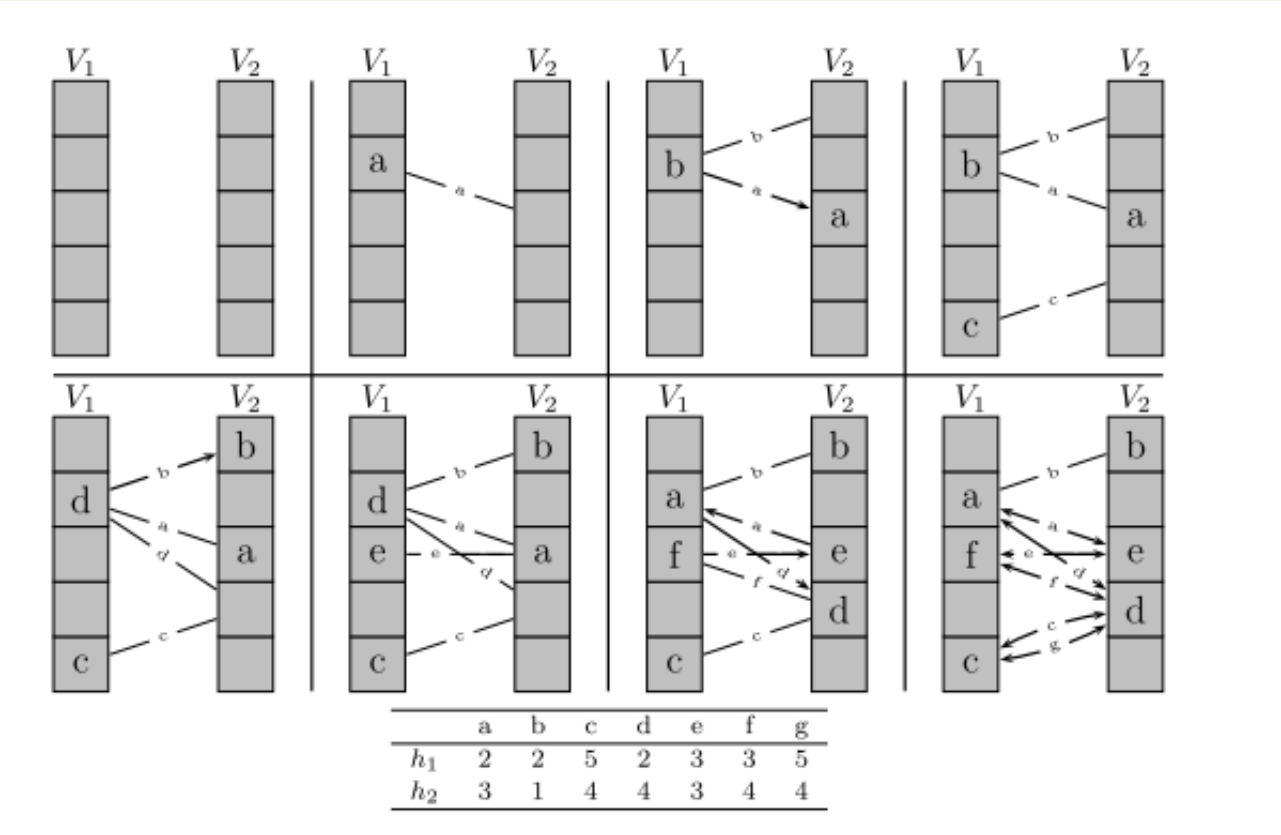
$$\mathbb{E}[\#\text{ rehashes}] = \frac{1}{1 - O(1/n)} = 1 + O(1/n)$$

2. Each rehash plus the attempt to insert with good insertions the n items has expected cost $\Theta(n)$
3. By Wald's lemma, the expected cost of the insertion will be

$$\mathbb{E}[\#\text{rehashes}] \times \mathbb{E}[\text{cost of rehash}] = (1 + O(1/n)) \times O(n) = O(n)$$

Cuckoo Hashing

To prove facts #1 (good insertion needs expected O(1) time) and #2 (probability of a good insertion is $1 - O(1/n^2)$) we formulate the problem in graph theoretic terms.



	a	b	c	d	e	f	g
h_1	2	2	5	2	3	3	5
h_2	3	1	4	4	3	4	4

Cuckoo Hashing

Cuckoo graph:

- Vertices: $V = \{v_{1,i}, v_{2,i} \mid 0 \leq i < M\} =$ the set of $2M$ slots in the tables
- Edges: If $T_1[j]$ is occupied by x then there's an edge $(v_{1,j}, v_{2,h_2(x)})$, where $v_{\ell,j}$ is the vertex associated to $T_\ell[j]$; x is the label of the edge. If $T_2[k]$ is occupied by y then there is an edge $(v_{2,k}, v_{1,h_1(y)})$ with label y.

This is a labeled directed "bipartite" multigraph—all edges go from $v_{1,j}$ to $v_{2,k}$ or from $v_{2,k}$ to $v_{1,j}$.

Cuckoo Hashing

Consider the connected components of the cuckoo graph. A component can be either a tree (no cycles), unicyclic (exactly one cycle–with trees "hanging") or complex (two or more cycles). Trees with k nodes have exactly $k-1$ edges, unicycles have exactly k edges and complex components have $> k$ edges.

- Fact 1: An insertion that creates a complex component is not good $\implies$ if the cuckoo graph contains no complex components then all insertions were good
- Fact 2: the expected time of a good insertion is bounded by the expected diameter of the component in which we make the insertion (also by the size)

Cuckoo Hashing

Then we convert the analysis to that of the cuckoo graph as a random bipartite graph with $2M$ vertices and $n = (1 - \epsilon)M$ edges—each item gives us an edge.
This is a very "sparse" graph, but if the density n/M grew to 1/2 there will be an complex component with very high probability (a similar thing happens in random Erdös-Renyi graphs).

Cuckoo Hashing

The most detailed analysis of the cuckoo graph has been made by Drmota and Kutzelnigg (2012). They prove, among many other things:

1. The probability that the cuckoo graph contains no complex component is
$$1 - h(\epsilon)\frac{1}{M} + O(1/M^2)$$
We do not reproduce their explicit formula for $h(\epsilon)$ here ($h(\epsilon) \to \infty$ as $\epsilon \to 0$)
2. The expected number of steps in n good insertions is
$$\leq n \cdot \min\left(4, \frac{\ln(1/\epsilon)}{1-\epsilon}\right) + O(1)$$

These two results prove the two Facts that we needed for our analysis

Cuckoo Hashing

- Several variants of Cuckoo hashing have appeared in the literature, for instance, using $d > 2$ tables and d hash functions. With such d-Cuckoo Hashing higher load factor, approaching 1 can be achieved
- An interesting variant puts all items in one single table, all the $d \geq 2$ hash functions map keys into the range $0..M-1$; the load factor n/M must be below some threshold α_d. We need to now which function was used to put the item at an occupied location—easily using $\log_2 d$ bits.

Part II

Probabilistic & Randomized Dictionaries

4 Randomized Binary Search Trees

5 Skip Lists

6 Hash Tables
- Separate Chaining
- Open Addressing
- Cuckoo Hashing

7 Bloom Filters

Bloom filters

A Bloom Filter is a probabilistic data structure representing a set of items; it supports:

- Addition of items: $F := F \cup \{x\}$
- Fast lookup: $x \in F$?

Bloom filters do require very little memory and are specially well suited for unsuccessful search (when $x \notin F$)

Bloom filters

- The price to pay for the reduced memory consumption and very fast lookup is the non-null probability of false positives.
- If $x \in F$ then a lookup in the filter will always return true; but if $x \notin F$ then there is some probability that we get a positive answer from the filter.
- In other words, if the filter says $x \notin F$ we are sure that's the case, but if the filter says $x \in F$ there is some probability that this is an error.

Bloom filters

Bloom filters are the most basic example of the so-called Approximate Membership Query Filters (AMQ filters) and support the following operations:

1. $F :=$ CREATEBF($N_{\max}, fp$): creates an empty Bloom filter F that might store up to $N_{\max}$ items, and sets an upper bound fp on the *false positive rate* allowed
2. F.INSERT(x): add item x to filter F
3. F.LOOKUP(x): returns whether x belongs to the filter F or not
   - if the answer is **true**, it might be wrong with probability $\leq fp$
   - if the answer is **false**, then $x \notin F$ for sure

Implementing Bloom filters

To represent a Bloom filter for a subset of items drawn from the domain $\mathcal{U}$ we will use:

1. A bitvector A of size M
2. A set of k pairwise independent hash functions $\{h_1, \ldots, h_k\}$, each $h_i : \mathcal{U} \to \{0, \ldots, M-1\}$

The values of M and k are carefully chosen as a function of N_{max} and fp

Implementing Bloom filters

procedure CREATEBF($N_{\max}$, fp)
  $M := \ldots$; $k := \ldots$
  A : bitvector$[0..M-1]$
  for $i := 0$ **to** $M-1$ **do** $A[i] := 0$
  for $j := 1$ **to** k **do** $h_i :=$ a random hash function

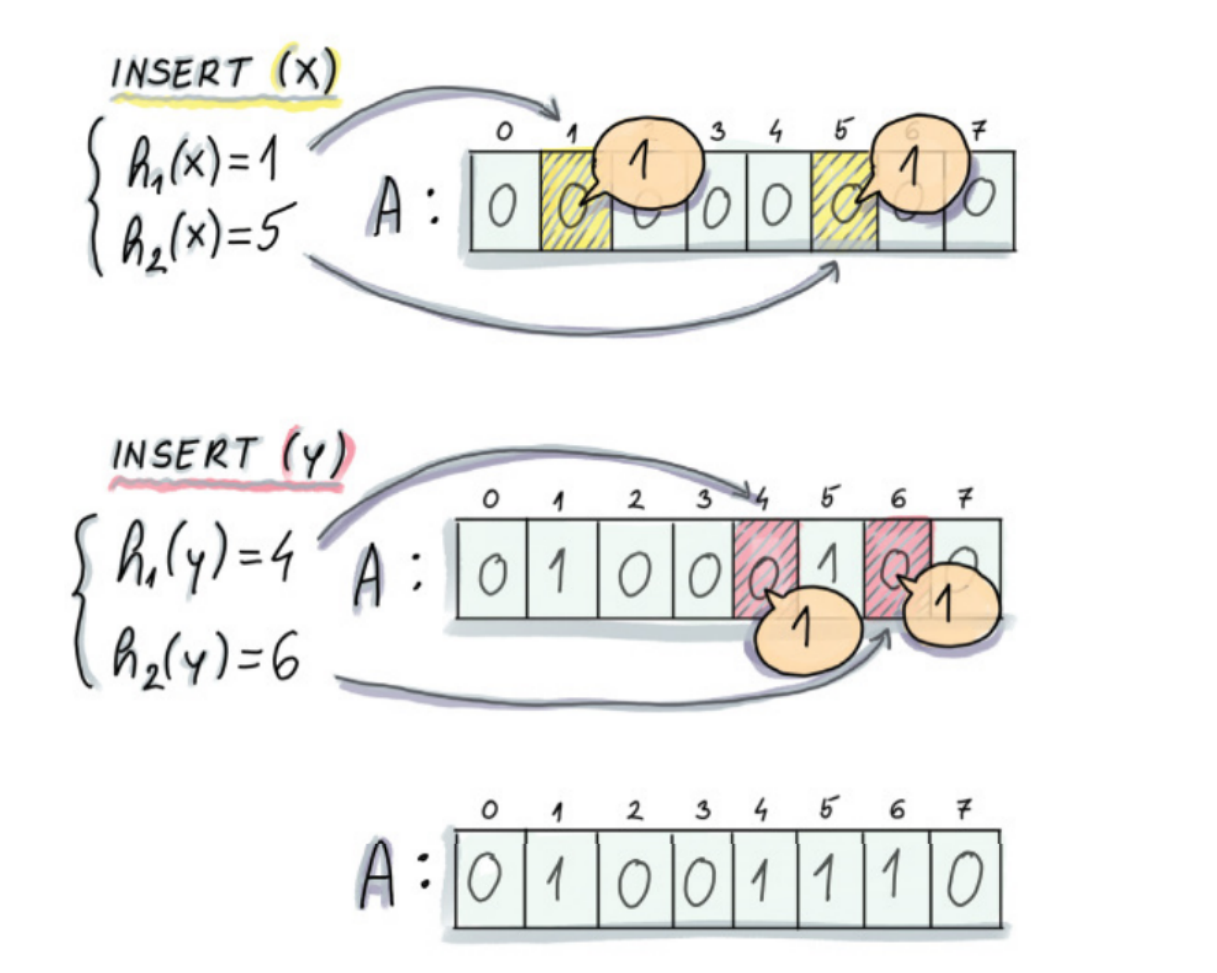
The k independent hash functions can be choosen from a universal class of hash functions (later in this course)

Insertion & lookup

```
procedure INSERT(x)
    for j := 1 to k do
        A[h_j(x)] := 1
```

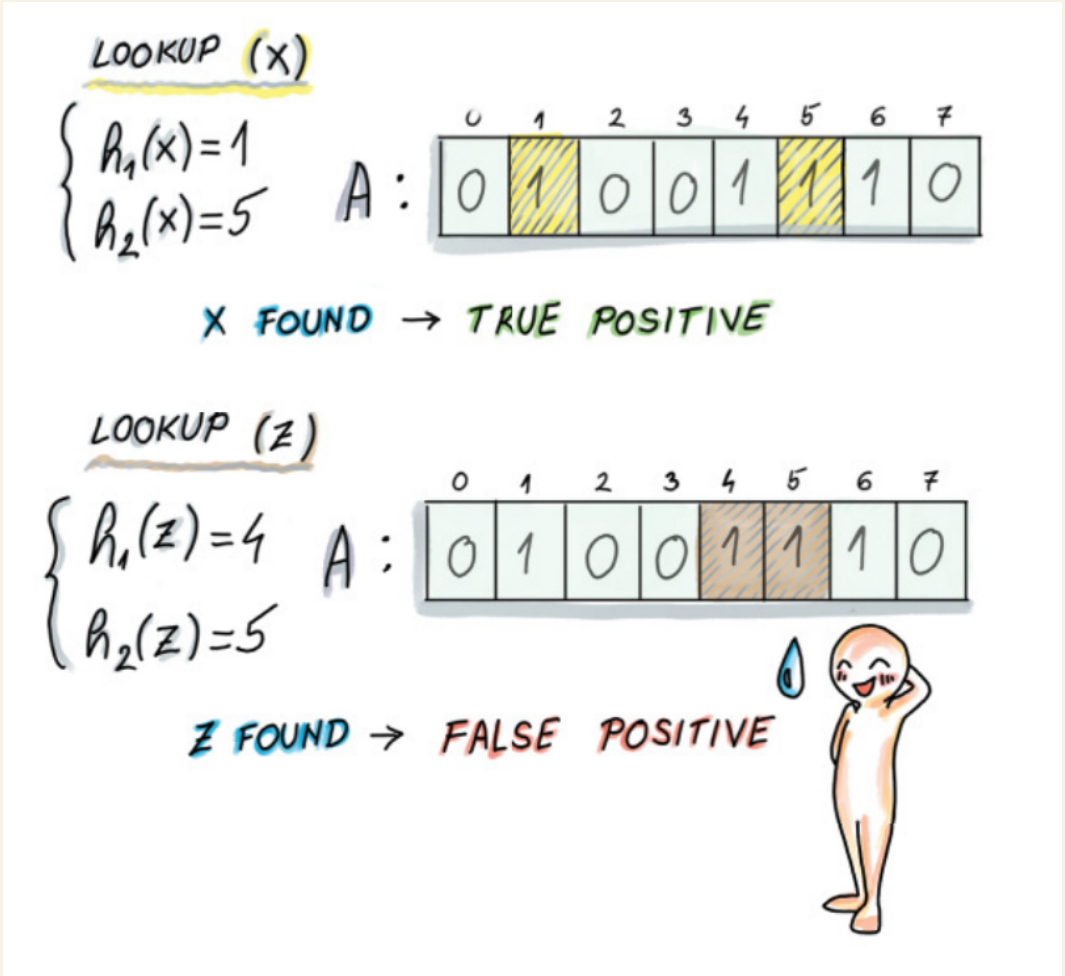
```
procedure LOOKUP(x)
    for j := 1 to k do
        if A[h_j(x)] = 0 then
            return false
    return true
```

Insertion & lookup



Source: D. Medjedovic & E. Tahirovic, *Algorithms and Data Structures for Massive Datasets*, 2022

Insertion & Lookup



Source: D. Medjedovic & E. Tahirovic, *Algorithms and Data Structures for Massive Datasets*, 2022

Analysis of Bloom filters

- Probability that the j-th bit is not updated when inserting x

$$\prod_{i=1}^{k} \mathbb{P}[h_i(x) \neq j] = \left(1 - \frac{1}{M}\right)^k$$

- Probability that the j-th bit is not updated after n insertions

$$\prod_{\ell=1}^{n} \mathbb{P}[A[j] \text{ is not updated in } \ell\text{-th insertion}] =$$

$$\left(\left(1 - \frac{1}{M}\right)^k\right)^n = \left(1 - \frac{1}{M}\right)^{k \cdot n}$$

Analysis of Bloom filters

- Probability that $A[j] = 1$ after n insertions

$$1 - \left(1 - \frac{1}{M}\right)^{k \cdot n}$$

- Probability that k checked bits are set to 1 $\approx$ probability of a false positive

$$\left(1 - \left(1 - \frac{1}{M}\right)^{k \cdot n}\right)^k \approx \left(1 - e^{-kn/M}\right)^k$$

if $n = \alpha M$, for some $\alpha > 0$

$$\left(1 - \frac{a}{x}\right)^{bx} \to e^{-ba}, \quad x \to \infty$$

Analysis of Bloom filters

- The derivation above is the so-called classic model for Bloom filters—but it is not the formula that Bloom himself derived in his paper!
- The approximation fails for small filters; correct formulas have been derived by Bose et al. (2008) and Christensen et al. (2010)
- For the rest of the presentation we will take

$$\mathbb{P}[x \text{ is a false positive}] = \mathbb{P}[x \notin F \wedge F.\texttt{contains}(x) = \textbf{true}]$$
$$\approx \left(1 - e^{-kn/M}\right)^k,$$

where x is drawn at random. Be careful! The formula does not give the probability that the filter reports x as a positive, conditioned to x being negative!

Optimal parameters for Bloom filters

- Fix n and M. The optimal value k^* minimizes the probability of false positive, thus

$$\frac{d}{dk}\left[\left(1-e^{-kn/M}\right)^k\right]_{k=k^*}=0$$

  which gives

$$k^* \approx \frac{M}{n}\ln 2 \approx 0.69\frac{M}{n}$$

- Call p the probability of a false positive. This probability is a function of k, $p=p(k)$; for the optimal choice k^* we have

$$p(k^*) \approx \left(1-e^{-\ln 2}\right)^{\frac{M}{n}\ln 2} = \left(\frac{1}{2}\right)^{\ln 2\frac{M}{n}} \approx 0.6185^{\frac{M}{n}}$$

Optimal parameters for Bloom filters

- Suppose that you want the probability of false positive $p^* = p(k^*)$ to remain below some bound P

$$p^* \leq P \implies \ln p^* = -\frac{M}{n}(\ln 2)^2 \leq \ln P$$

$$\frac{M}{n}(\ln 2)^2 \geq -\ln P = \ln(1/P)$$

$$\frac{M}{n} \geq \frac{1}{\ln 2}\log_2(1/P) \approx 1.44 \log_2(1/P)$$

$$M \geq 1.44 \cdot n \cdot \log_2(1/P)$$

Optimal parameters for Bloom filters

procedure CREATEBF(N_{max}, fp)
$M := 1.44 \cdot N_{\text{max}} \cdot \log_2(1/fp)$;
$k := \log_2(1/fp)$
...

Optimal parameters for Bloom filters

- If we want a Bloom filter for a database that will store about $n \approx 10^8$ elements and a false positive rate $\leq 5\%$, we need a bitvector of size $M \geq 624 \cdot 10^6$ bits (that's around 74MB of memory).
- Despite this amount of memory is big, it is only a small fraction of the size of the database itself: even if we store only keys of 32 bytes each, the database occupies more than 3GB.
- The optimal number k^* of hash functions for the example above is 4.32 ($\implies$ use 4 or 5 hash functions for optimal performance)

To learn more

[1] B.H. Bloom.
Space/Time Trade-offs in Hash Coding with Allowable Errors.
Communications of the ACM 13 (7): 422–426, 1970.

[2] A. Broder and M. Mitzenmacher.
Network Applications of Bloom Filters: A Survey
Internet Mathematics 1 (4):485–509, 2003.

To learn more (2)

[3] P. Bose, H. Guo, E.Kranakis et al.
On the False-Positive Rate of Bloom Filters
Information Processing Letters 108 (4):210–213, 2004.

[4] K. Christensen, A. Roginsky and M. Jimeneo.
A New Analysis of the False-Positive Rate of a Bloom Filter
Information Processing Letters 110 (21):944–949, 2010.

Part III

Priority Queues

8 Priority Queues: Introduction

9 Heaps

10 Binomial Queues

11 Fibonacci Heaps

Priority Queues: Introduction

A priority queue (esp: *cola de prioridad*) stores a collection of elements, each one endowed with a value called its priority.

Priority queues support the insertion of new elements and the query and removal of an element of minimum (or maximum) priority.

Introduction

```
template <typename Elem, typename Prio>
class PriorityQueue {
public:
   ...
// Adds an element x with priority p to the priority queue.
void insert(cons Elem& x, const Prio& p);

// Returns an element of minimum priority; throws an
// exception if the queue is empty.
Elem min() const;

// Returns the priority of an element of minimum priority; throws an
// exception if the queue is empty.
Prio min_prio() const;

// Removes an element of minimum priority; throws an
// exception if the queue is empty.
void remove_min();

// Returns true iff the priority queue is empty
bool empty() const;
};
```

Priority Queues: Introduction

```
// We have two arrays Weight and Symb with the atomic
// weights and the symbols of n chemical elements, e.g.,
// Symb[i] = "C" y Weight[i] = 12.2, for some i.
// We use a priority queue to sort the information in alphabetic
// ascending order

PriorityQueue<double, string> P;
for (int i = 0; i < n; ++i)
   P.insert(Weigth[i], Symb[i]);
int i = 0;
while(not P.empty()) {
   Weight[i] = P.min();
   Symb[i] = P.min_prio();
   ++i;
   P.remove_min();
}
```

Priority Queues: Introduction

- Several techniques that used for the implementation of dictionaries can also be used for priority queues (not hash tables).
- For instance, balanced search trees such as AVLs can be used to implement a PQ with cost $\mathcal{O}(\log n)$ for insertions and deletions

Part III

Priority Queues

8 Priority Queues: Introduction

9 Heaps

10 Binomial Queues

11 Fibonacci Heaps

Heaps

Definition

A heap is a binary tree such that

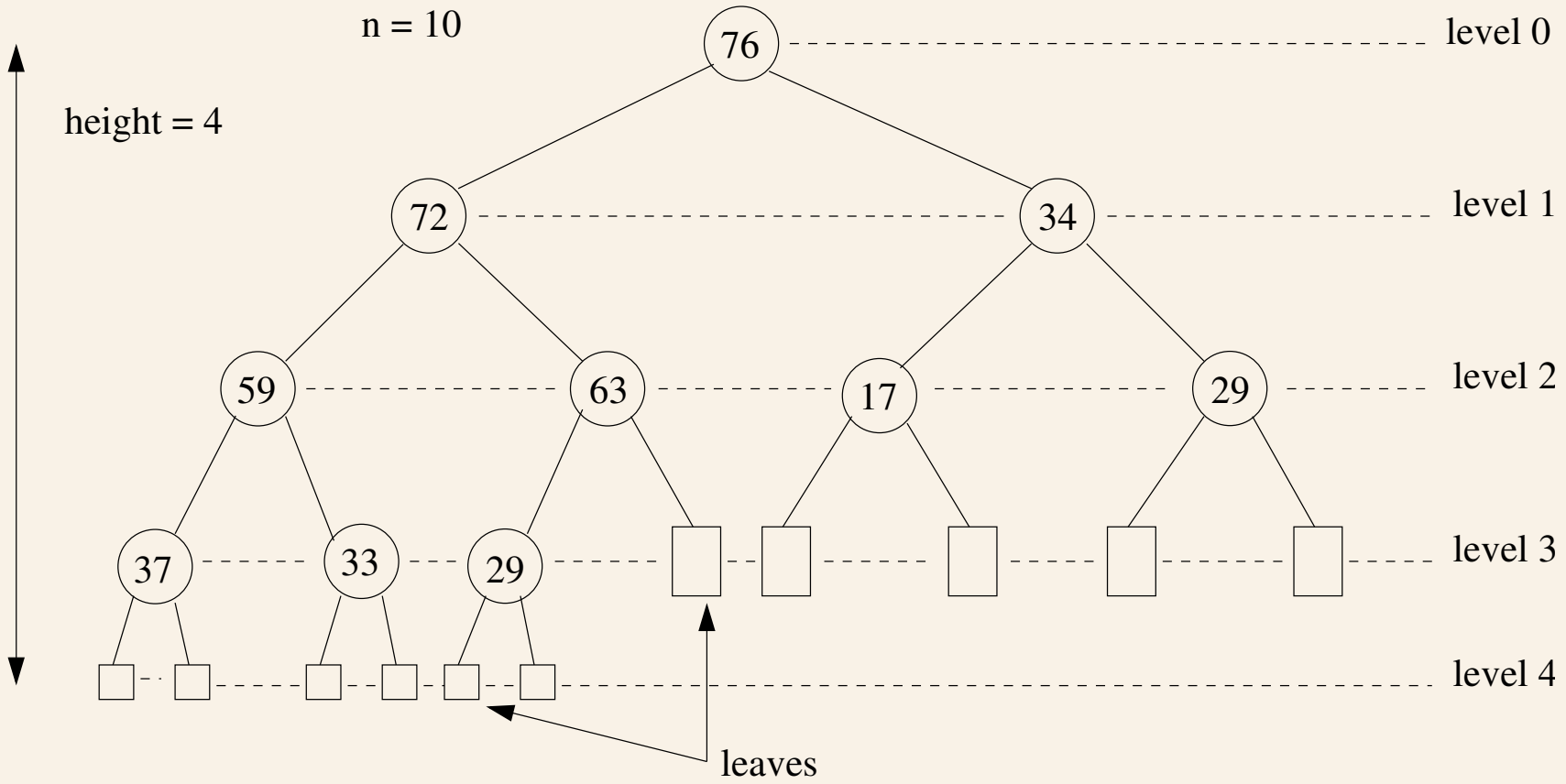
1. All empty subtrees are located in the last two levels of the tree.
2. If a node has an empty left subtree then its right subtree is also empty.
3. The priority of any element is larger or equal than the priority of any element in its descendants.

Heaps

Because of properties 1–2 in the definition, a heap is a quasi-complete binary tree. Property #3 is called heap order (for max-heaps).

If the priority of an element is smaller or equal than that of its descendants then we talk about min-heaps.

Heaps


n = 10
height = 4
76
72
34
59
63
17
29
37
33
29
level 0
level 1
level 2
level 3
level 4
leaves

Heaps

Proposition

1. *The root of a max-heap stores an element of maximum priority.*
2. *A heap of size n has height*

$$h = \lceil \log_2(n+1) \rceil .$$

If heaps are used to implement a PQ the query for a max/min element and its priority is trivial: we need only to examine the root of the heap.

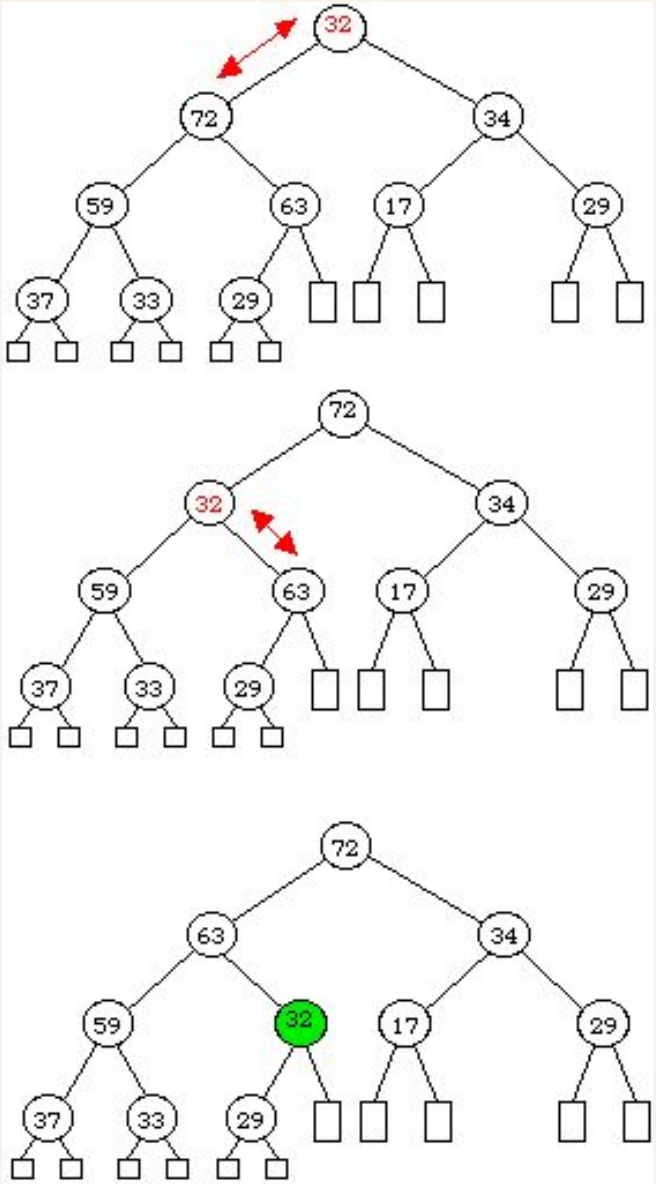
Heaps: Removing the maximum

1. Replace the root of the heap with the last element (the rightmost element in the last level)
2. Reestablish the invariant (heap order) sinking the root: The function `sink` exchanges a given node with its largest priority child, if its priority is smaller than the priority of its child, and repeats the same until the heap order is reestablished.

Heaps: Removing the maximum

1. Replace the root of the heap with the last element (the rightmost element in the last level)
2. Reestablish the invariant (heap order) sinking the root: The function `sink` exchanges a given node with its largest priority child, if its priority is smaller than the priority of its child, and repeats the same until the heap order is reestablished.

Heaps: Removing the maximum



Heaps: Adding a new element

1. Add the new element as rightmost node of the last level of the heap (or as the first element of a new deeper level)
2. Reestablish the heap order sifting up (a.k.a. *floating*) the new added element:
   The function `siftup` compares the given node to its father, and they are exchanged if its priority is larger than that of its father; the process is repeated until the heap order is reestablished.

Heaps: Adding a new element

1. Add the new element as rightmost node of the last level of the heap (or as the first element of a new deeper level)
2. Reestablish the heap order sifting up (a.k.a. *floating*) the new added element:
   The function `siftup` compares the given node to its father, and they are exchanged if its priority is larger than that of its father; the process is repeated until the heap order is reestablished.

The Cost of Heaps

Since the height of a heap is $\Theta(\log n)$, the cost of removing the maximum and the cost of insertions is $\mathcal{O}(\log n)$.

We can implement heaps with dynamically allocated nodes, and three pointers per node (left, right, father) . . . But it is much easier and efficient to implement heaps with vectors!

Since the heap is a quasi-complete binary tree this allows us to avoid wasting memory: the n elements are stored in the first n components of the vector, which implicitly represent the tree.

Implementing Heaps

To make the rules easier we will use a vector A of size $n+1$ and discard $A[0]$. Resizing can be used to allow unlimited growth.

1. $A[1]$ contains the root
2. If $2i \leq n$ then $A[2i]$ contains the left child of $A[i]$ and if $2i+1 \leq n$ then $A[2i+1]$ contains the right subtree of $A[i]$
3. If $i \geq 2$ then $A[i/2]$ contains the father of $A[i]$

Implementing Heaps

To make the rules easier we will use a vector A of size $n+1$ and discard $A[0]$. Resizing can be used to allow unlimited growth.

1. $A[1]$ contains the root
2. If $2i \leq n$ then $A[2i]$ contains the left child of $A[i]$ and if $2i+1 \leq n$ then $A[2i+1]$ contains the right subtree of $A[i]$
3. If $i \geq 2$ then $A[i/2]$ contains the father of $A[i]$

Implementing Heaps

To make the rules easier we will use a vector A of size $n+1$ and discard $A[0]$. Resizing can be used to allow unlimited growth.

1. $A[1]$ contains the root
2. If $2i \leq n$ then $A[2i]$ contains the left child of $A[i]$ and if $2i+1 \leq n$ then $A[2i+1]$ contains the right subtree of $A[i]$
3. If $i \geq 2$ then $A[i/2]$ contains the father of $A[i]$

Implementing Heaps

```
template <typename Elem, typename Prio>
class PriorityQueue {
public:
  ...
private:
   // Component of index 0 is not used
   vector<pair<Elem, Prio> > h;
   int nelems;

   void siftup(int j) throw();
   void sink(int j) throw();
};
```

Implementing Heaps

```
template <typename Elem, typename Prio>
bool PriorityQueue<Elem,Prio>::empty() const {

   return nelems == 0;
}

template <typename Elem, typename Prio>
Elem PriorityQueue<Elem,Prio>::min() const {

   if (nelems == 0) throw EmptyPriorityQueue;
   return h[1].first;
}

template <typename Elem, typename Prio>
Prio PriorityQueue<Elem,Prio>::min_prio() const {

   if (nelems == 0) throw EmptyPriorityQueue;
   return h[1].second;
}
```

Implementing Heaps

```
template <typename Elem, typename Prio>
void PriorityQueue<Elem,Prio>::insert(cons Elem& x,
                          cons Prio& p) {
   ++nelems;
   h.push_back(make_pair(x, p));
   siftup(nelems);
}

template <typename Elem, typename Prio>
void PriorityQueue<Elem,Prio>::remove_min() const {

   if (nelems == 0) throw EmptyPriorityQueue;
   swap(h[1], h[nelems]);
   --nelems;
   h.pop_back();
   sink(1);
}
```

Implementing Heaps

```
// Cost: O(log(n/j))
template <typename Elem, typename Prio>
void PriorityQueue<Elem,Prio>::sink(int j) {

   // if j has no left child we are at the last level
   if (2 * j > nelems) return;

   int minchild = 2 * j;
   if (minchild < nelems and
       h[minchild].second > h[minchild + 1].second)
      ++minchild;

   // minchild is the index of the child with minimum priority
   if (h[j].second > h[minchild].second) {
      swap(h[j], h[minchild]);
      sink(minchild);
   }
}
```

Implementing Heaps

```
// Cost: O(log j)
template <typename Elem, typename Prio>
void PriorityQueue<Elem,Prio>::siftup(int j) {

   // if j is the root we are done
   if (j == 1) return;

   int father = j / 2;
   if (h[j].second < h[father].second) {
      swap(h[j], h[father]);
      siftup(father);
   }
}
```

Part III

Priority Queues

8 Priority Queues: Introduction

9 Heaps

10 Binomial Queues

11 Fibonacci Heaps

Binomial Queues

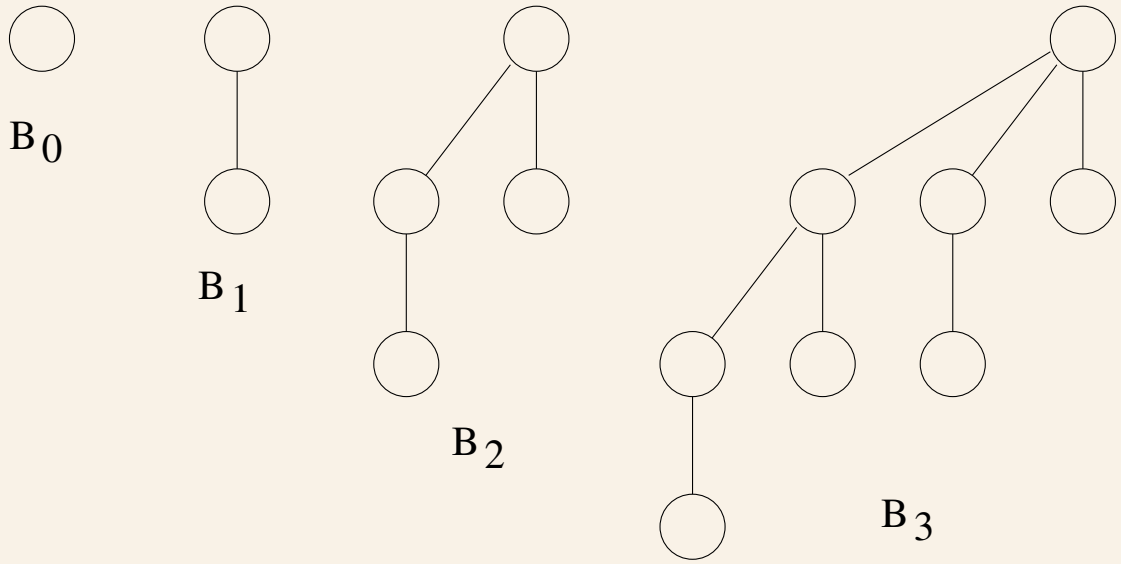
J. Vuillemin

- A binomial queue is a data structure that efficiently supports the standard operations of a priority queue (`insert`, `min`, `extract_min`) and additionally it supports the melding (merging) of two queues in time $\mathcal{O}(\log n)$.
- Note that melding two ordinary heaps takes time $\mathcal{O}(n)$.
- Binomial queues (aka *binomial heaps*) were invented by J. Vuillemin in 1978.

```
template <typename Elem, typename Prio>
class PriorityQueue {
public:
  PriorityQueue() throw(error);
  ~PriorityQueue() throw();
  PriorityQueue(const PriorityQueue& Q) throw(error);
  PriorityQueue& operator=(const PriorityQueue& Q) throw(error);

  // Add element x with priority p to the priority queue
  void insert(cons Elem& x, const Prio& p) throw(error)

  // Returns an element of minimum priority. Throws an exception if
  // the priority queue is empty
  Elem min() const throw(error);

  // Returns the minimum priority in the queue. Throws an exception
  // if the priority queue is empty
  Prio min_prio() const throw(error);

  // Removes an element of minimum priority from the queue. Throws
  // an exception if the prioirty queue is empty
  void remove_min() throw(error);

  // Returns true if and only if the queue is empty
  bool empty() const throw();

  // Melds (merges) the priority queue with the priority queue Q;
  // the priority queue Q becomes empty
  void meld(PriorityQueue& Q) throw();

  ...
};
```

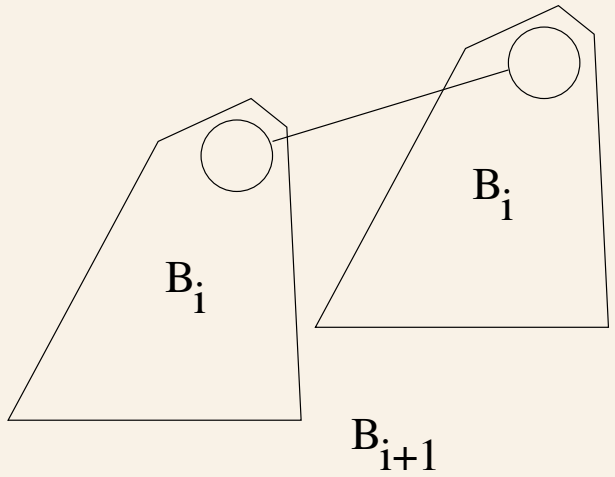
Binomial Queues

- A binomial queue is a collection of binomial trees.
- The binomial tree of order i (called B_i) contains 2^i nodes



Binomial Queues

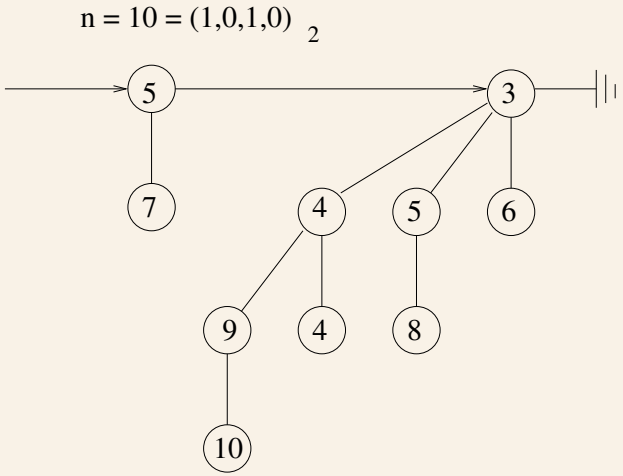
- A binomial tree of order $i+1$ is (recursively) built by planting a binomial tree B_i as a child of the root of another binomial tree B_i.



- The size of B_i is 2^i; indeed $|B_0| = 2^0 = 1$, $|B_{i+1}| = 2 \cdot |B_i| = 2 \cdot 2^i = 2^{i+1}$
- A binomial tree of order i has exactly $\binom{i}{k}$ descendants at level k (the root is at level 0); hence their name
- A binomial tree of order i has height $i = \log_2 |B_i|$

Binomial Queues

- Let $(b_{k-1}, b_{k-2}, \ldots, b_0)_2$ be the binary representation of n. Then a binomial queue for a set of n elements contains b_0 binomial trees of order 0, b_1 binomial trees of order 1, …, b_j binomial trees of order j, …

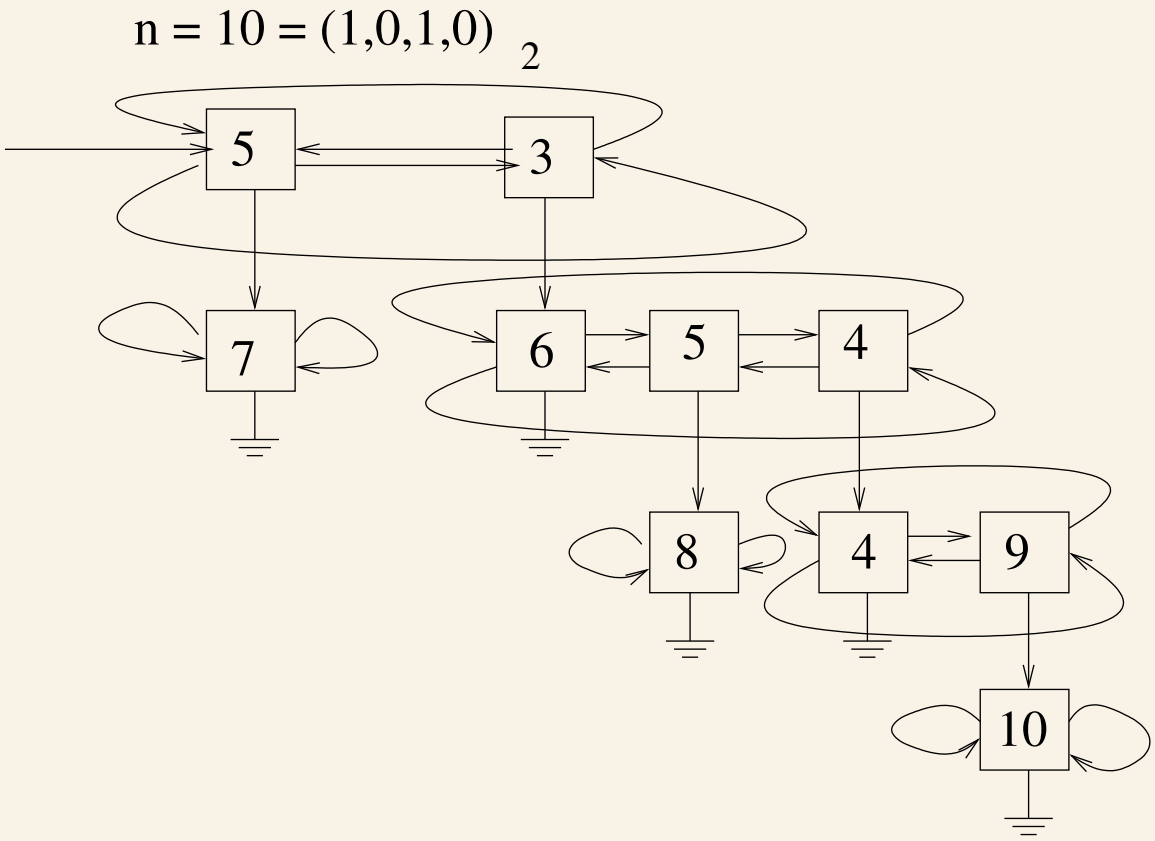


- A binomial queue for n elements contains at most $\lceil \log_2(n+1) \rceil$ binomial trees
- The n elements of the binomial queue are stored in the binomial trees in such a way that each binomial tree satisfies the heap property: the priority of the element at any given node is $\leq$ than the priority of its descendants

Binomial Queues

- Each node in the binomial queue will store an `Elem` and its priority (any type that admits a total order)
- Each node will also store the order of the binomial subtree of which the node is the root
- We will use the usual *first-child/next-sibling* representation for general trees, with a twist: the list of children of a node will be double linked and circularly closed
- We need thus three pointers per node: `first_child`, `next_sibling`, `prev_sibling`
- The binomial queue is simply a pointer to the root of the first binomial tree
- We will impose that all lists of children are in increasing `order`

Binomial Queues



Binomial Queues

```
template <typename Elem, typename Prio>
class PriorityQueue {
  ...
private:
  struct node_bq {
    Elem _info;
    Prio _prio;
    int _order;
    node_bq* _first_child;
    node_bq* _next_sibling;
    node_bq* _prev_sibling;
    node_bq(const Elem& x, const Prio& p, int order = 0) : _info(x), _prio(p),
                                                           _order(order), _first_child(NULL) {
      _next_sibling = _prev_sibling = this;
    };
  };
  node_bq* _first;
  int _nelems;
```

Binomial Queues

- To locate an element of minimum priority it is enough to visit the roots of the binomial trees; the minimum of each binomial tree is at its root because of the heap property.
- Since there are at most $\lceil \log_2(n+1) \rceil$ binomial trees, the methods `min()` and `min_prio()` take $\mathcal{O}(\log n)$ time and both are very easy to implement.

Binomial Queues

- We can also keep a pointer to the root of the element with minimum priority, and update it after each insertion or removal, when necessary. The complexity of updates does not change and `min()` and `min_prio()` take $\mathcal{O}(1)$ time

Binomial Queues

```
static node_bq* min(node_bq* f) const throw(error) {
    if (f == NULL) throw error(EmptyQueue);
    Prio minprio = f -> _prio;
    node_bq* minelem = f;
    node_bq* p = f-> _next_sibling;
    while (p != f) {
      if (p -> _prio < minprio) {
        minprio = p -> _prio;
        minelem = p;
      };
      p = p -> _next_sibling;
    }
    return minelem;
}

Elem min() const throw(error) {
    return min(_first) -> _info;
}

Prio min_prio() const throw(error) {
    return min(_first) -> _prio;
}
```

Binomial Queues

- To insert a new element x with priority p, a binomial queue with just that element is trivially built and then the new queue is melded with the original queue
- If the cost of melding two queues with a total number of items n is $M(n)$, then the cost of insertions is $\mathcal{O}(M(n))$

Binomial Queues

```
void insert(const Elem& x, const Prio& p) throw(error) {
    node_bq* nn = new node_bq(x, p);
    _first = meld(_first, nn);
    ++_nelems;
}
```

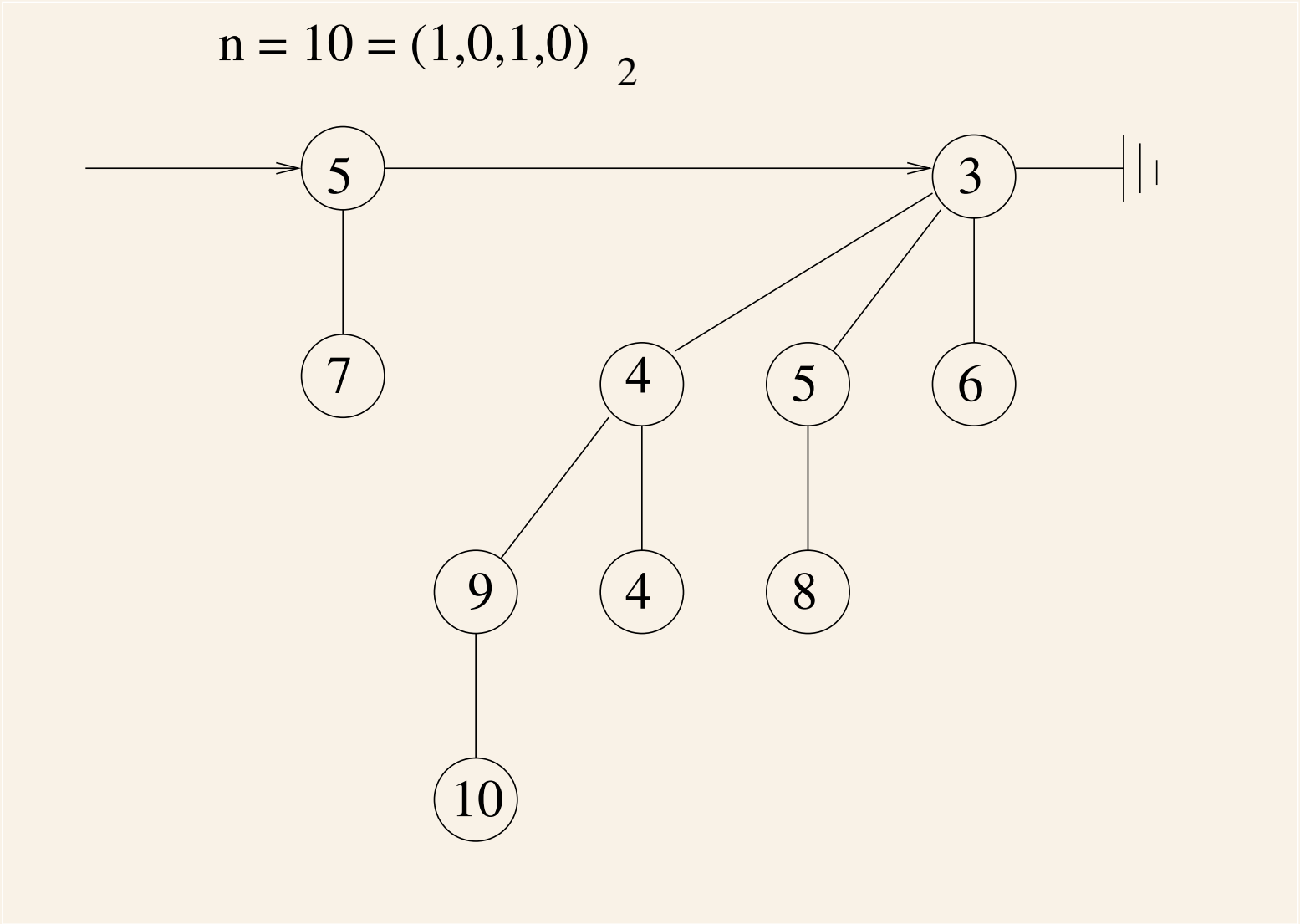
Binomial Queues

- To delete an element of minimum priority from a queue Q, we start locating such an element, say x; it must be at the root of some B_i
- The root of B_i is dettached from Q and thus B_i is no longer part of the original queue Q; the list of x's children is a binomial queue Q' with $2^i - 1$ elements
- The queue Q' has i binomial trees of orders 0, 1, 2, ... up to $i - 1$

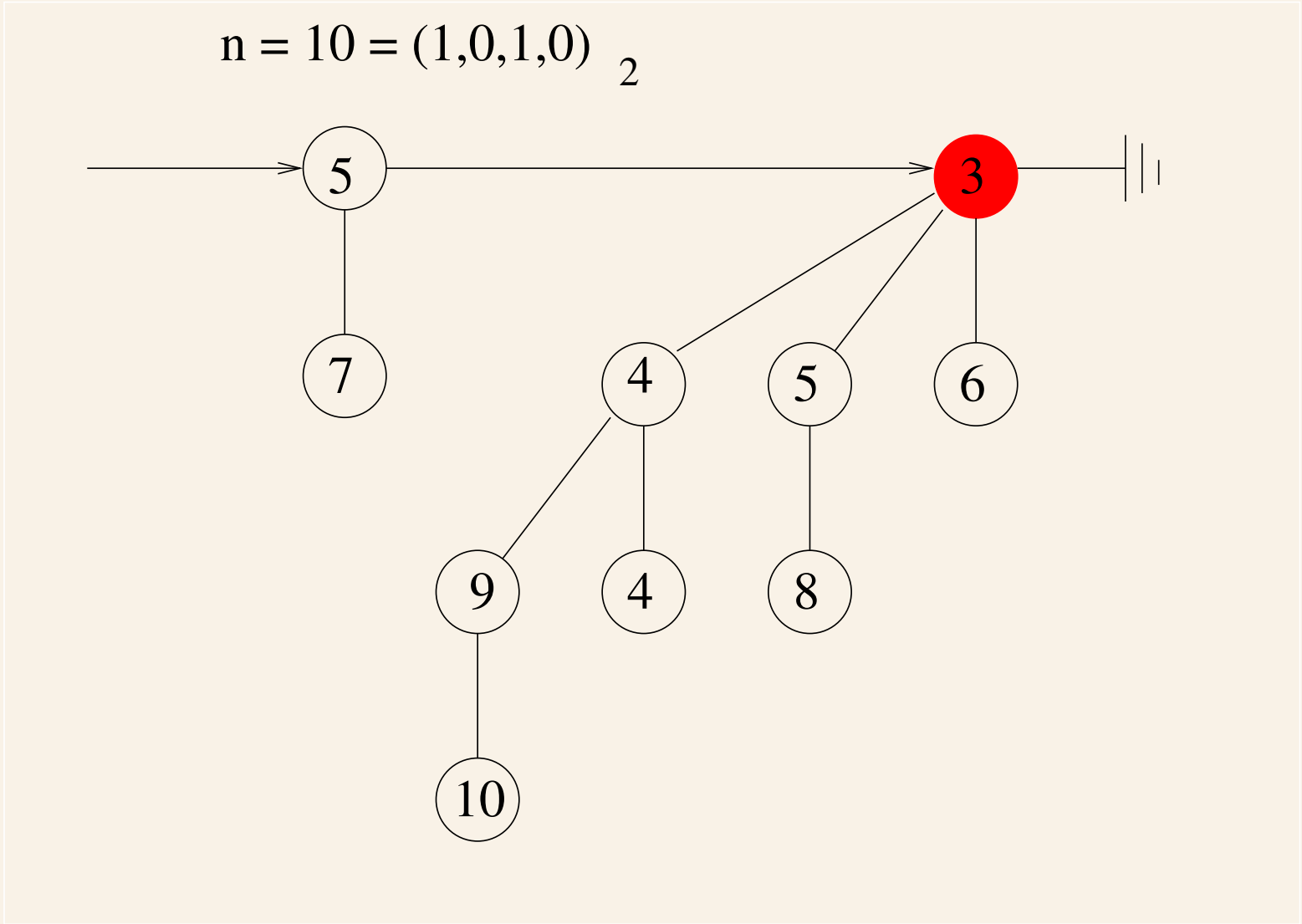
$$1 + 2 + \ldots + 2^{i-1} = 2^i - 1$$

- The queue $Q \setminus B_i$ is then melded with Q'

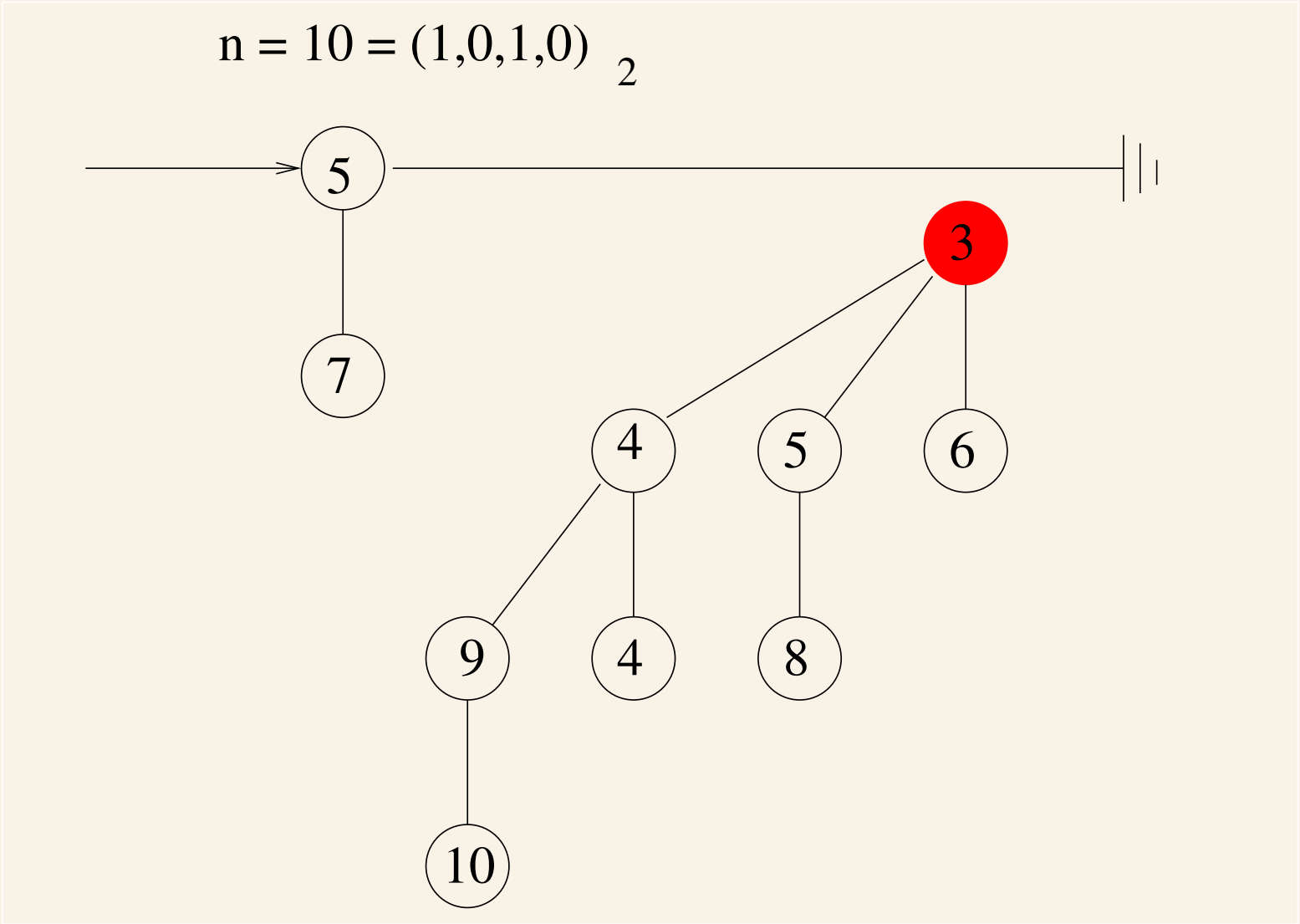
Binomial Queues



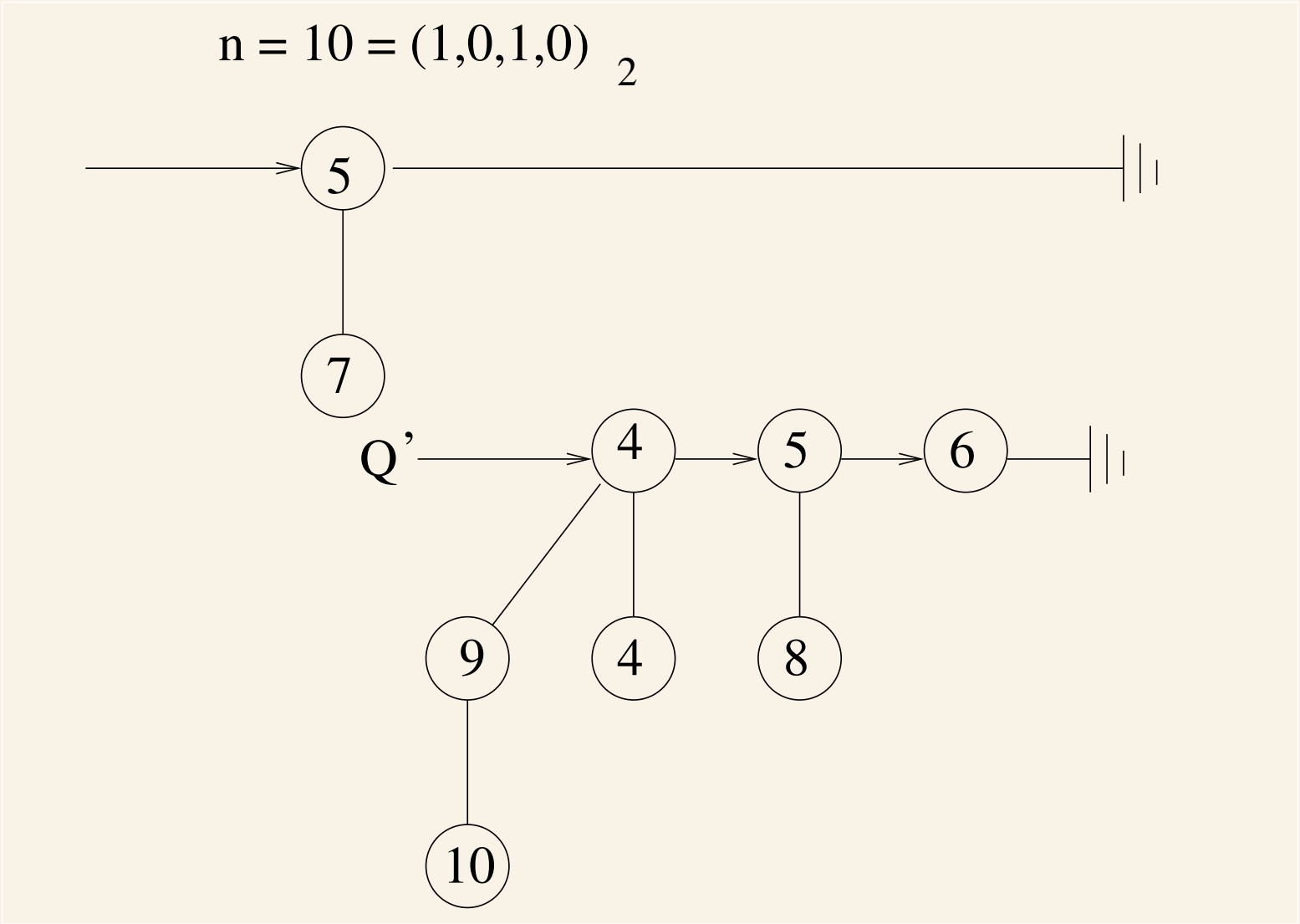
Binomial Queues



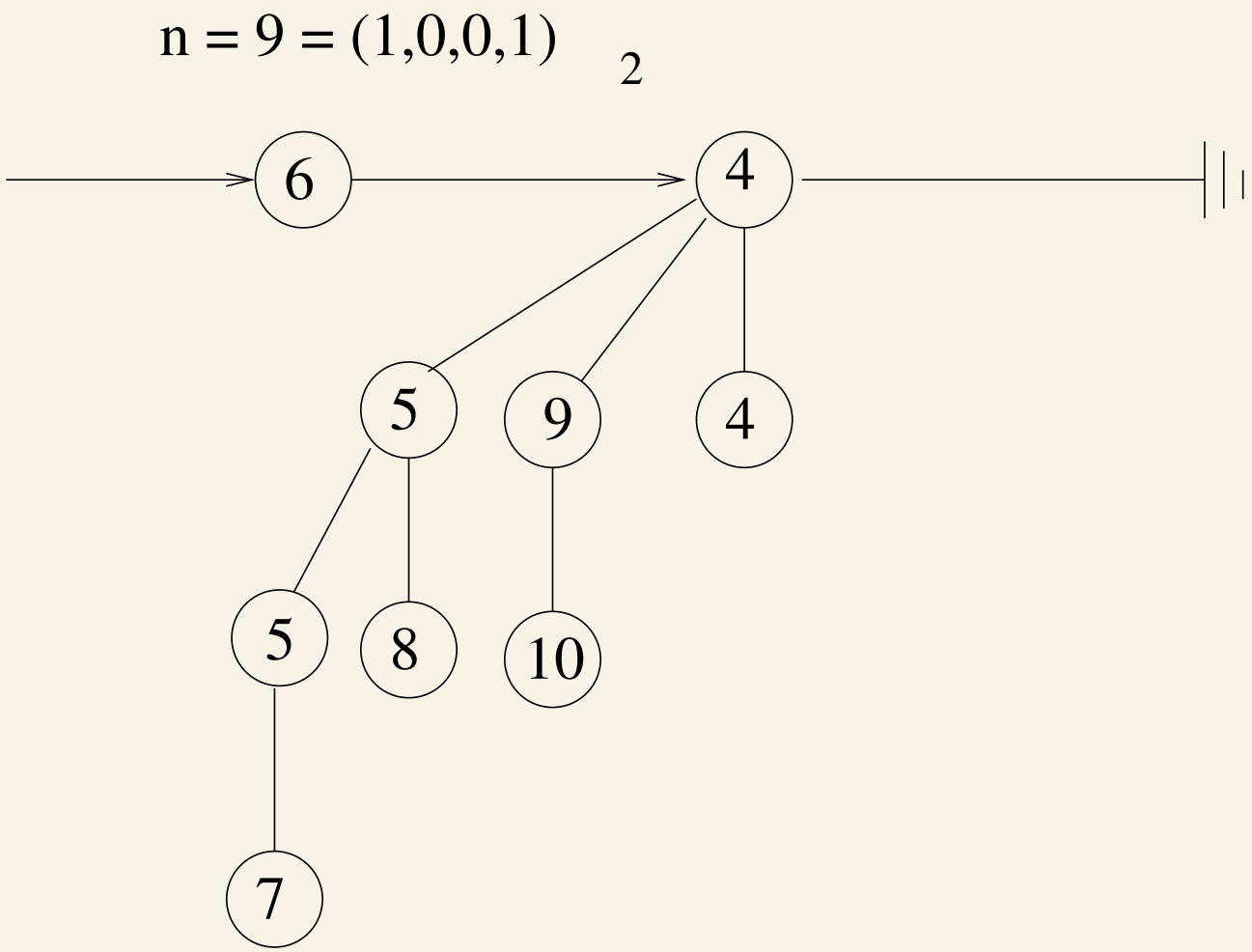
Binomial Queues



Binomial Queues



Binomial Queues



Binomial Queues

```
void remove_min() throw(error) {
  node_bq* m = min(_first);
  node_bq* children = m -> _first_child;
  if (m != m -> _next_sibling) { // there is more than one
                                 // binomial tree
    m -> _prev_sibling -> _next_sibling = m -> _next_sibling;
    m -> _next_sibling -> _prev_sibling = m -> _prev_sibling;
  } else {
    _first = NULL;
  }
  node_bq* qaux = m -> _first_child;
  m -> _first_child = m -> _next_sibling = m -> _prev_sibling = NULL;
  delete m;
  _first = meld(_first, qaux);
  --_nelems;
}
```

Binomial Queues

- The cost of extracting an element of minimum priority:
  - To locate the minimum priority has cost $\mathcal{O}(\log n)$
  - Melding $Q \setminus B_i$ and Q' has cost $\mathcal{O}(M(n))$, since $|Q \setminus B_i| + |Q'| = n - 2^i + 2^i - 1 = n - 1$
- In total: $\mathcal{O}(\log n + M(n))$

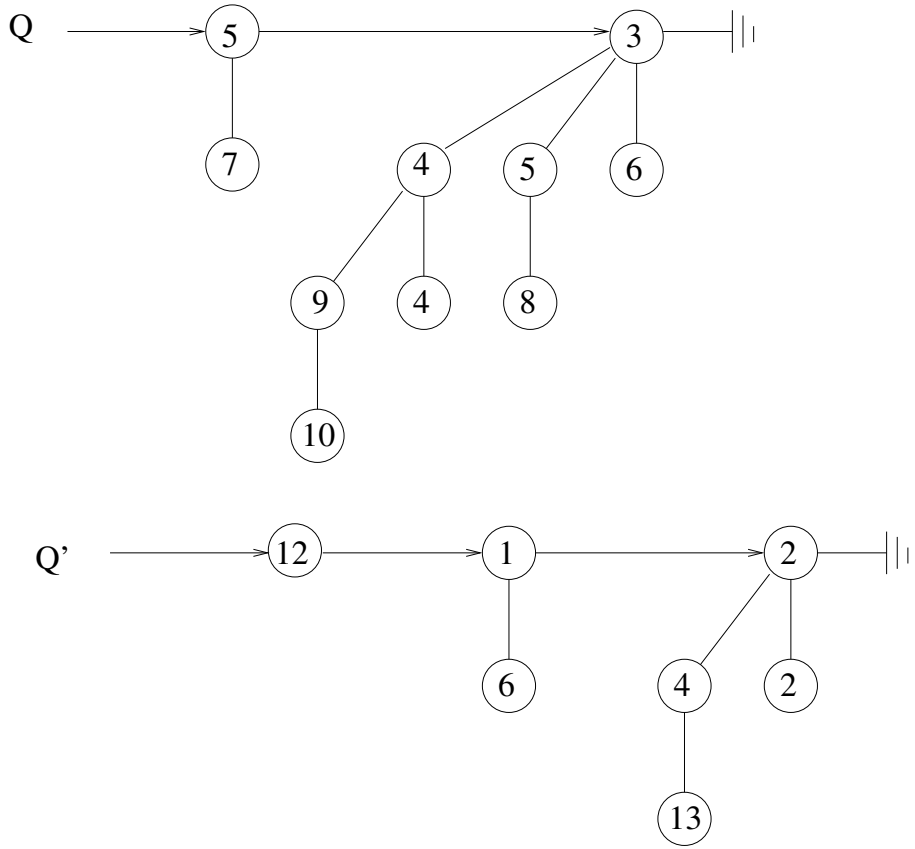
Binomial Queues

- Melding two binomial queues Q and Q' is very similar to the addition of two binary numbers bitwise
- The procedure iterates along the two lists of binomial trees; at any given step we consider two binomial trees B_i and B'_j, and a *carry* $C = B''_k$ or $C = \emptyset$

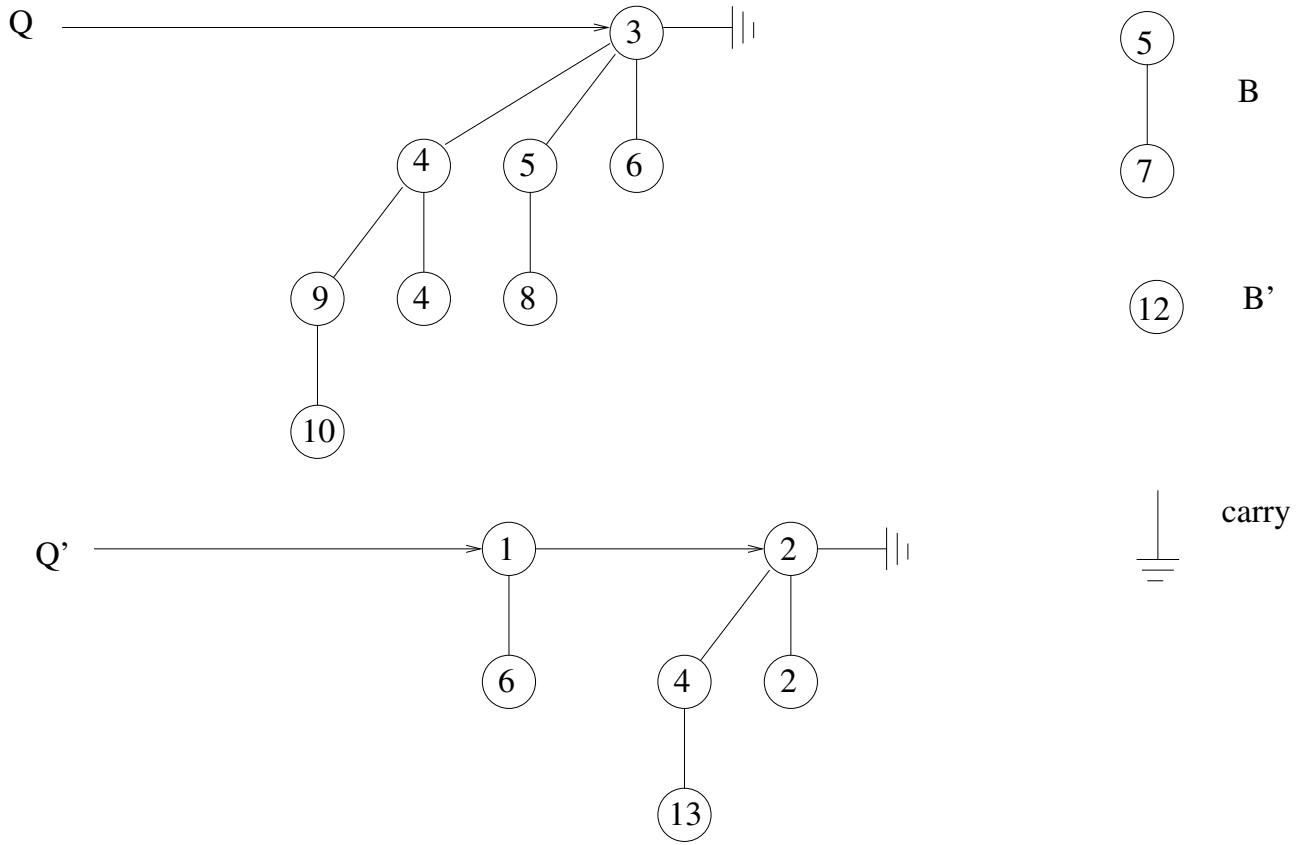
Binomial Queues

- Let $r = \min(i, j, k)$.
  - If there is only one binomial tree in $\{B_i, B'_j, C\}$ of order r, put that binomial tree in the result and advance to the next binomial tree in the corresponding queue (or set $C = \emptyset$)
  - If exactly two binomial trees in $\{B_i, B'_j, C\}$ are of order r, set $C = B_{r+1}$ by joining the two binomial trees (while preserving the heap property), remove the binomial trees from the respective queues, and advance to the next binomial tree where appropiate
  - If the three binomial trees are of order r, put B''_k in the result, remove B_i from Q and B'_j from Q', set $C = B_{r+1}$ by joining B_i and B'_j, and advance in both Q and Q' to the next binomial trees

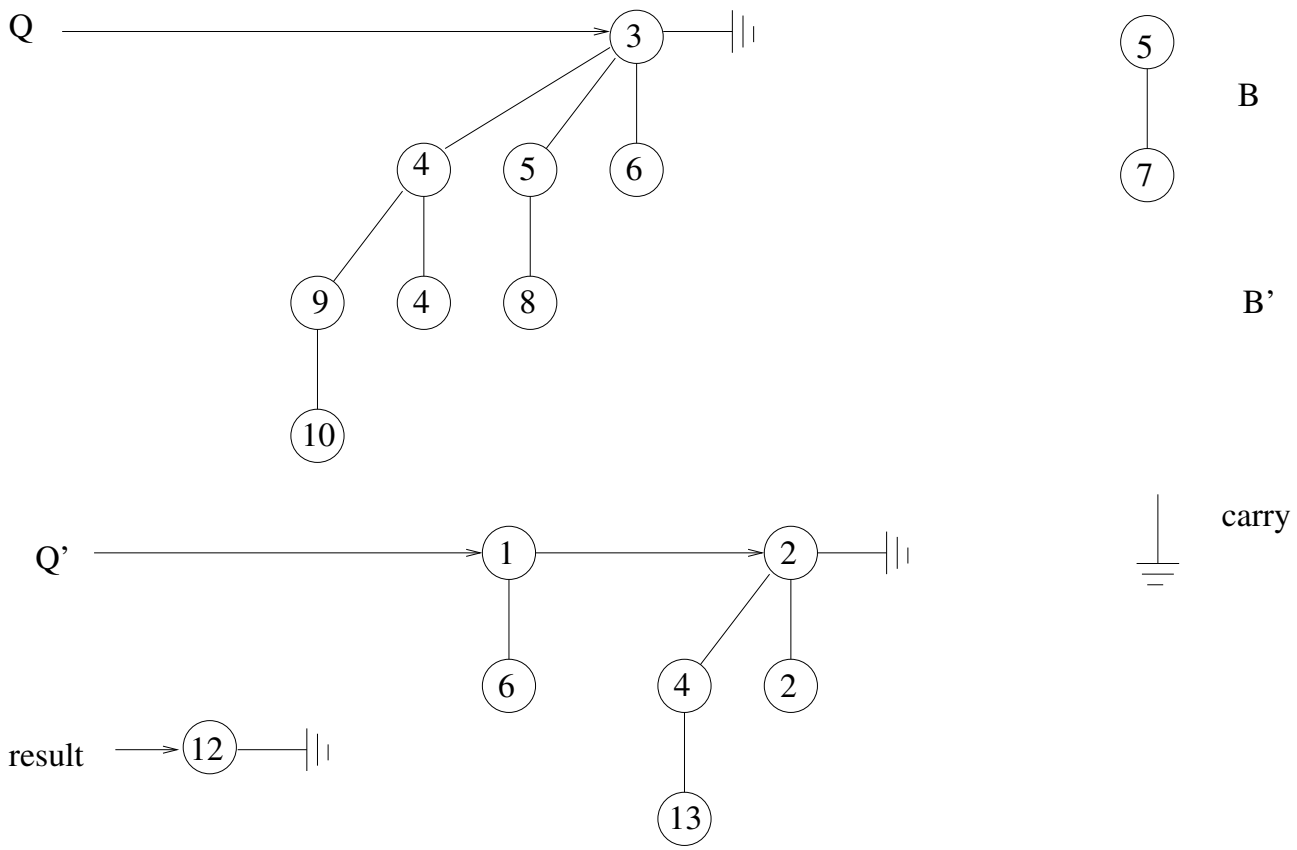
Binomial Queues



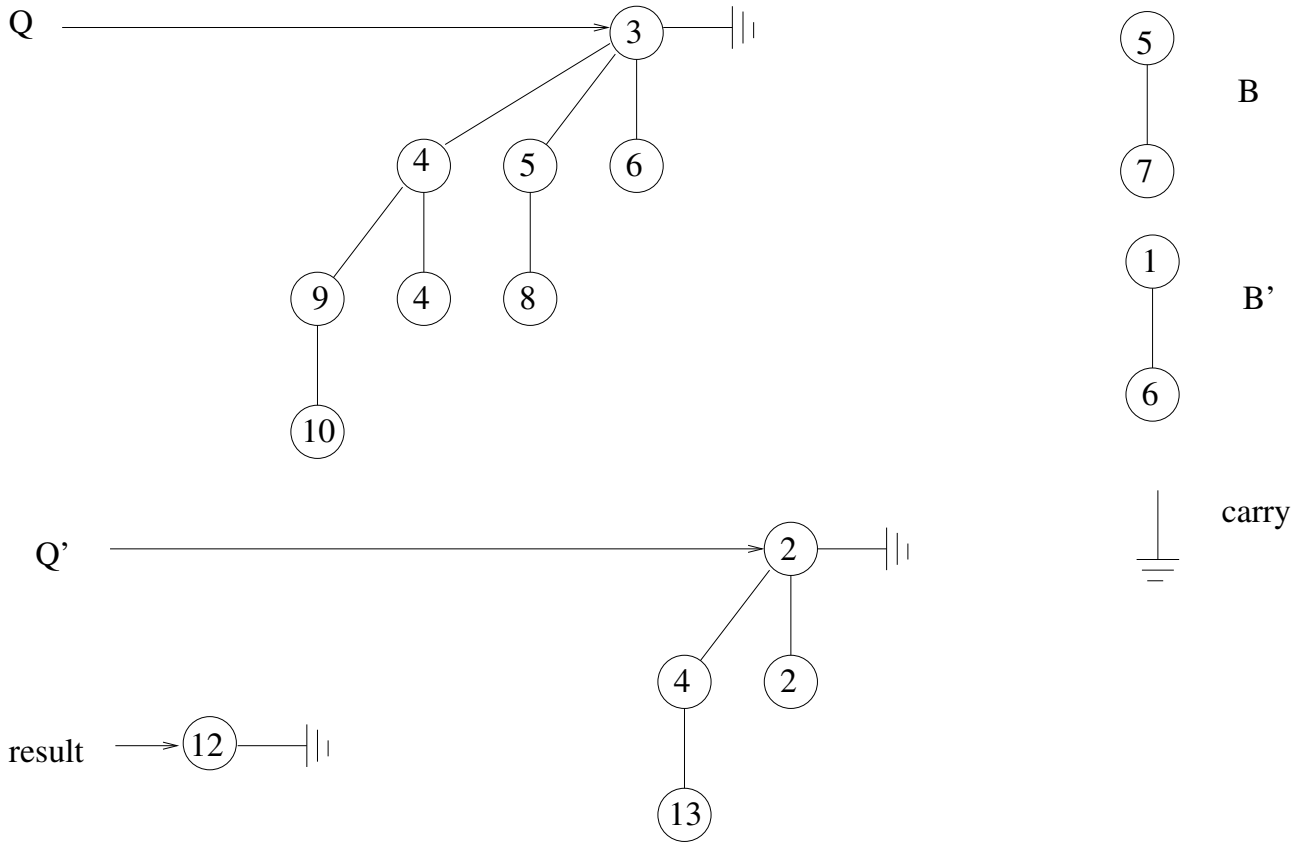
Binomial Queues



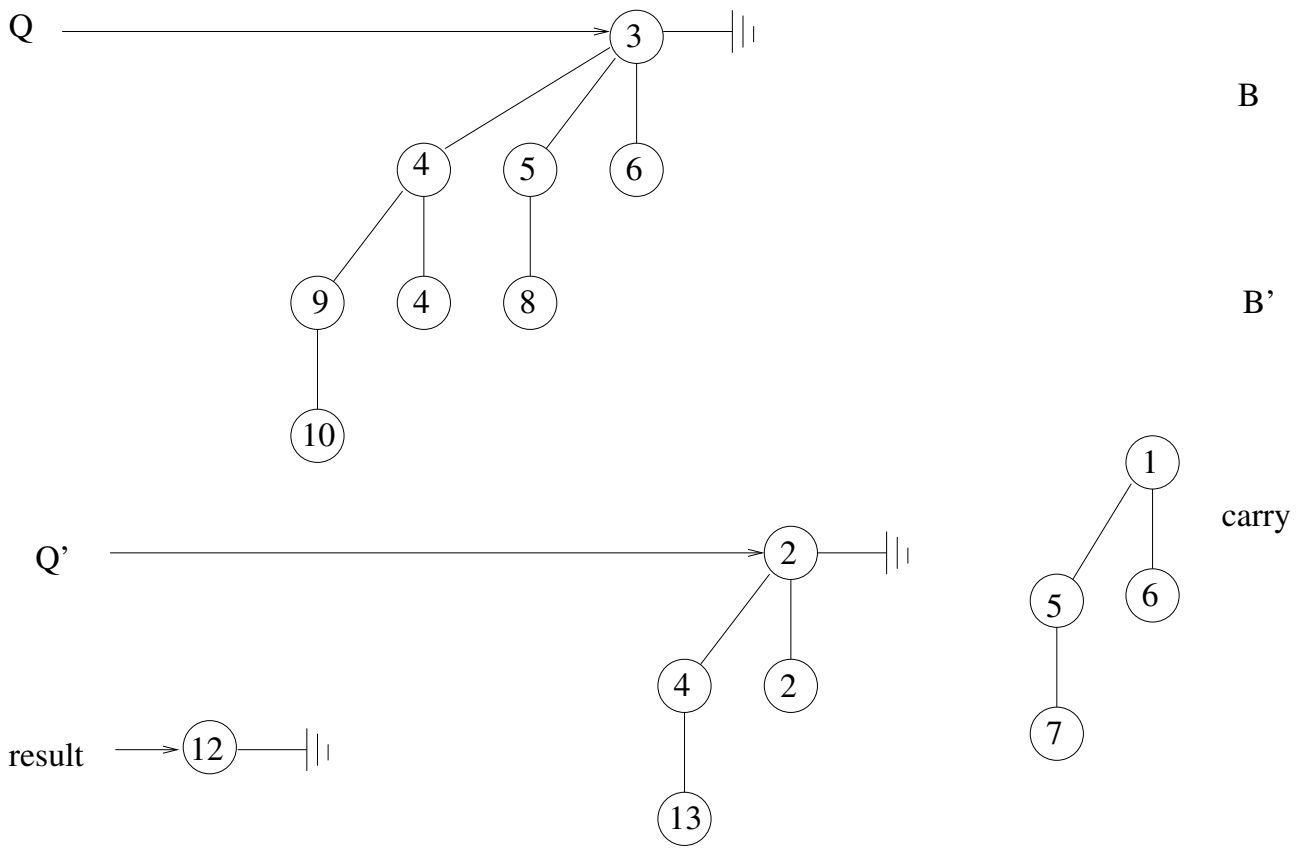
Binomial Queues



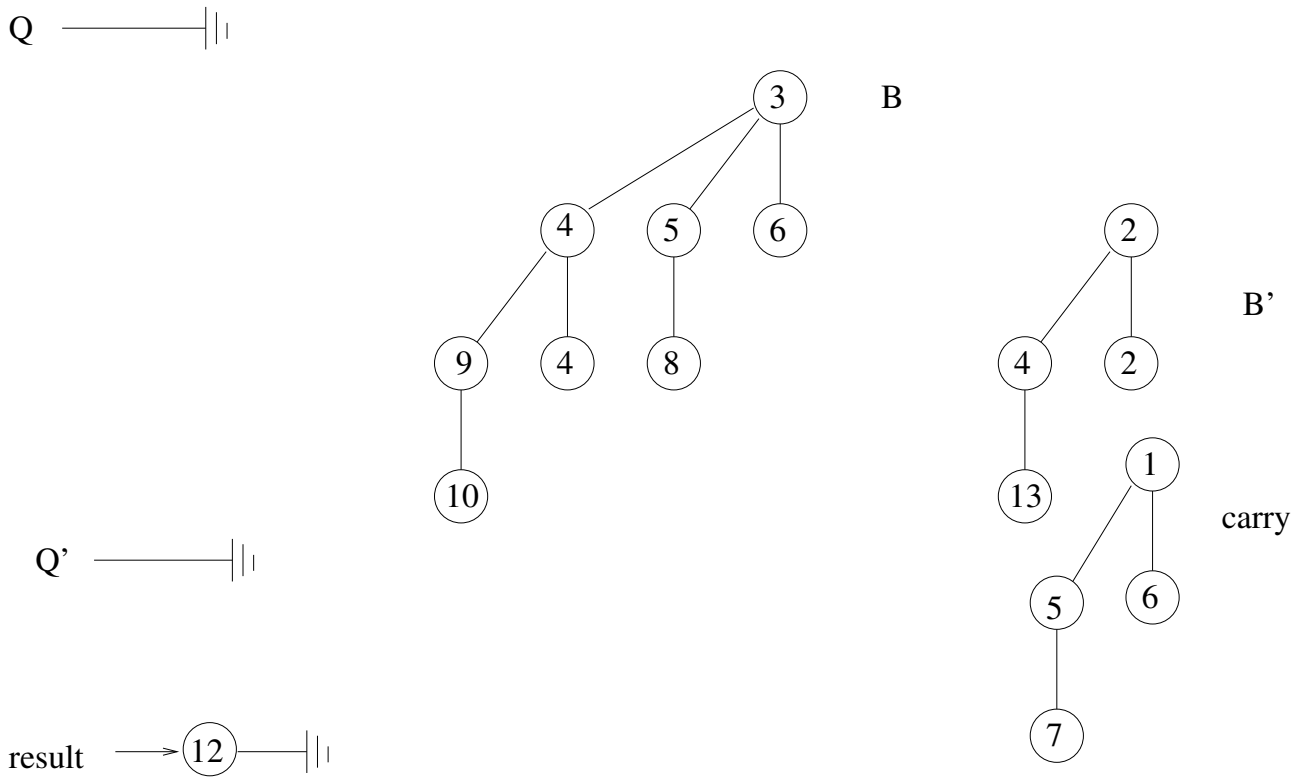
Binomial Queues



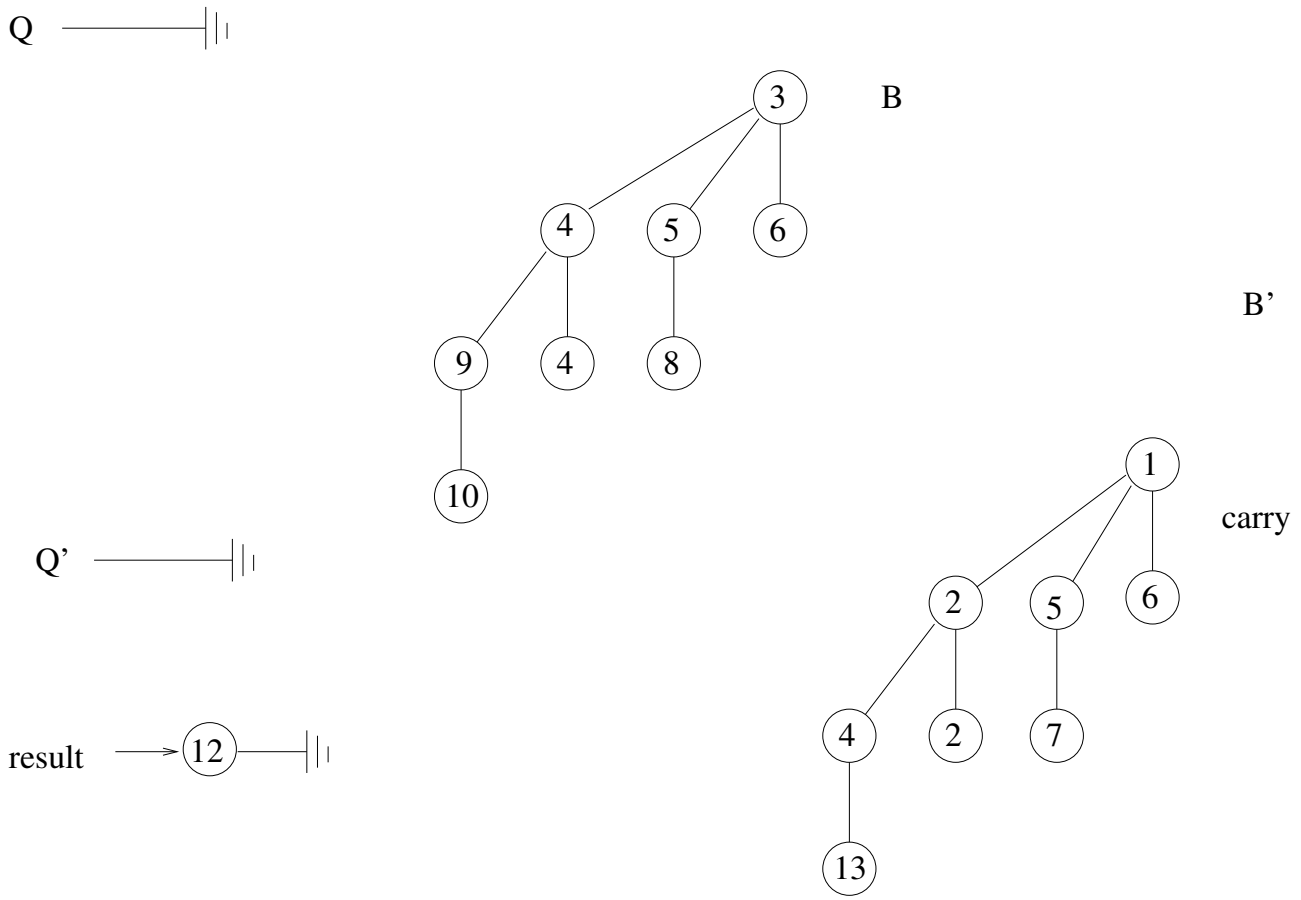
Binomial Queues



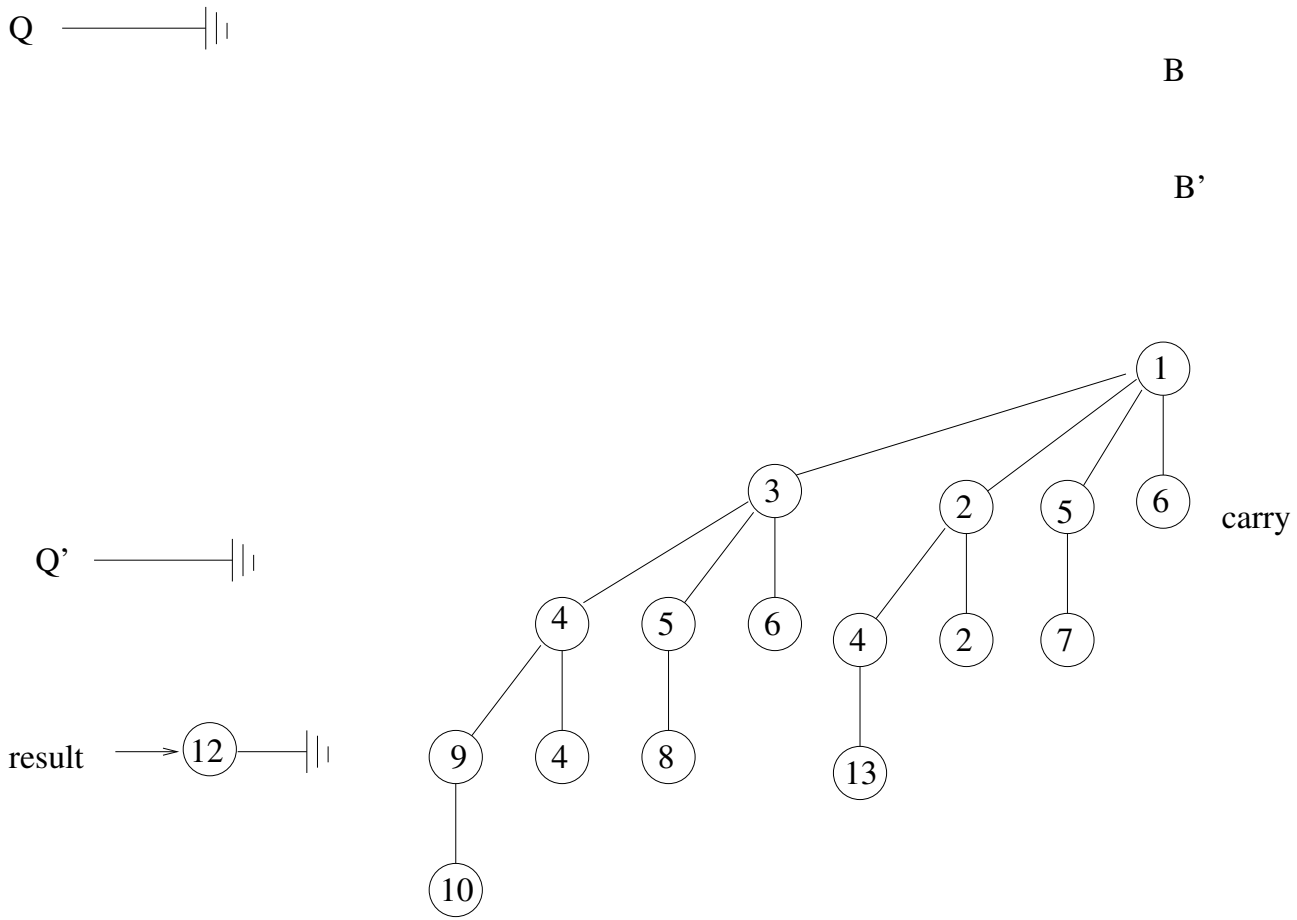
Binomial Queues



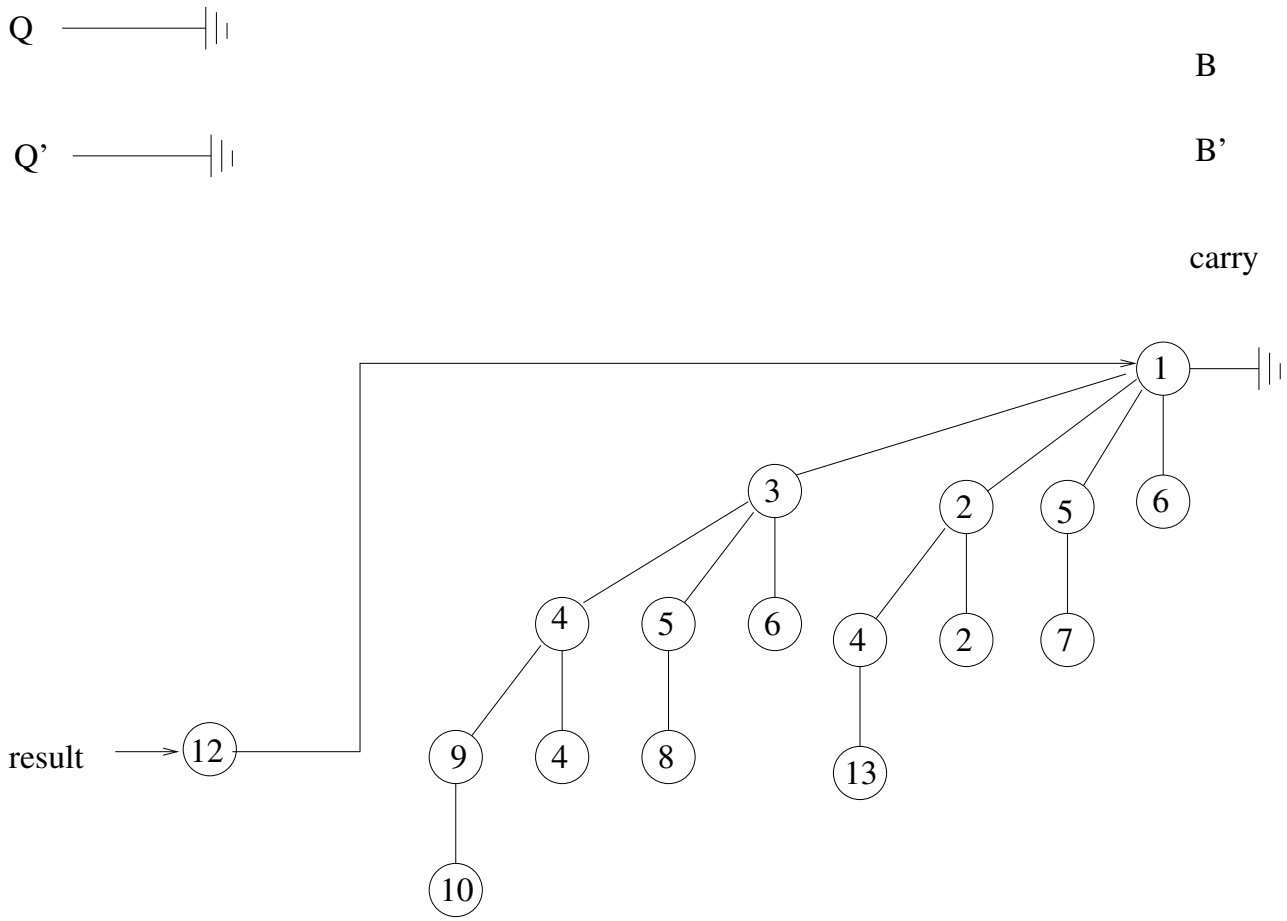
Binomial Queues



Binomial Queues



Binomial Queues



Binomial Queues

```
// removes the first binomial tree from the binomial queue q
// and returns it; if the queue q is empty, returns NULL: cost: Theta(1)
static node_bq* pop_front(node_bq*& q) throw();

// adds the binomial queue b (typically consisting of a single tree)
// at the end of the binomial queue q;
// does nothing if b == NULL; cost: Theta(1)
static void append(node_bq*& q, node_bq* b) throw();

// melds Q and Qp, destroying the two binomial queues
static node_bq* meld(node_bq*& Q, node_bq*& Qp) throw() {
    node_bq* B = pop_front(Q);
    node_bq* Bp = pop_front(Qp);
    node_bq* carry = NULL;
    node_bq* result = NULL;
    while (non-empty(B, Bp, carry) >= 2) {
       node_bq* s = add(B, Bp, carry);
       append(result, s);
       if (B == NULL) B = pop_front(Q);
       if (Bp == NULL) Bp = pop_front(Qp);
    }
    // append the remainder t othe result
    append(result, Q);
    append(result, Qp);
    append(result, carry);
    return result;
}
```

Binomial Queues

```
static node_bq* add(node_bq*& A, node_bq*& B, node_bq*& C) throw() {
  int i = order(A); int j = order(B); int k = order(C);
  int r = min(i, j, k);
  node_bq* a, b, c;
  a = b = c = NULL;
  if (i == r) { a = A; A = NULL; }
  if (j == r) { b = B; B = NULL; }
  if (k == r) { c = C; C = NULL; }
  if (a != NULL and b == NULL and c == NULL) {
    return a;
  }
  if (a == NULL and b != NULL and c == NULL) {
    return b;
  }
  if (a == NULL and b == NULL and c != NULL) {
    return c;
  }
  if (a != NULL and b != NULL and c == NULL) {
    C = join(a, b);
    return NULL;
  }
  if (a != NULL and b == NULL and c != NULL) {
    C = join(a,c);
    return NULL;
  }
  if (a == NULL and b != NULL and c != NULL) {
    C = join(b,c);
    return NULL;
  }
  /// a != NULL and b != NULL and c != NULL
  C = join(a,b);
  return c;
```

Binomial Queues

```
static int order(node_bq* q) throw() {
   // no binomial queue will ever be of order as high as 256 ...
   // unless it had 2^256 elements, more than elementary particles in
   // this Universe; to all practical purposes 256 = infinity
   return q == NULL ? 256 : q -> _order;
}

// plants p as rightmost child of q or q as rightmost child of p
// to obtain a new binomial tree of order + 1 and preserving
// the heap property
static node_bq* join(node_bq* p, node_bq* q) {
  if (p -> _prio <= q -> _prio) {
    push_back(p -> _first_child, q);
    ++p -> _order;
    return p;
  } else {
    push_back(q -> _first_child, p);
    ++q -> _order;
    return q;
  }
}
```

Binomial Queues

- Melding two queues with ℓ and m binomial trees each, respectively, has cost $\mathcal{O}(\ell + m)$ because the cost of the body of the iteration is $\mathcal{O}(1)$ and each iteration removes at least one binomial tree from one of the queues
- Suppose that the queues to be melded contain n elements in total; hence the number of binomial trees in Q is $\leq \log n$ and the same is true for Q', and the cost of `meld` is $M(n) = \mathcal{O}(\log n)$
- The cost of inserting a new element is $\mathcal{O}(M(n))$ and the cost of removing an element of minimum priority is

$$\mathcal{O}(\log n + M(n)) = \mathcal{O}(\log n)$$

Binomial Queues

- Note that the cost of inserting an item in a binomial queue of size n is $\Theta(\ell_n + 1)$ where ℓ_n is the weight of the rightmost zero in the binary representation of n.
- The cost of n insertions

$$\sum_{0 \le i < n} \Theta(\ell_i + 1) = \sum_{r=1}^{\lceil \log_2(n+1) \rceil} \Theta(r) \cdot \frac{n}{2^r}$$

$$\le n\Theta\left(\sum_{r \ge 0} \frac{r}{2^r}\right) = \Theta(n),$$

  as $\approx n/2^r$ of the numbers between 0 and $n-1$ have their rightmost zero at position r, and the infinite series in the last line above is bounded by a positive constant
- This gives a $\Theta(1)$ amortized cost for insertions

Binomial Queues

To learn more:

[1] J. Vuillemin
A Data Structure for Manipulating Priority Queues.
Comm. ACM 21(4):309–315, 1978.

[2] T. Cormen, C. Leiserson, R. Rivest and C. Stein.
Introduction to Algorithms, 2e.
MIT Press, 2001.

Part III

Priority Queues

8 Priority Queues: Introduction

9 Heaps

10 Binomial Queues

11 Fibonacci Heaps

Fibonacci Heaps

M.L. Fredman

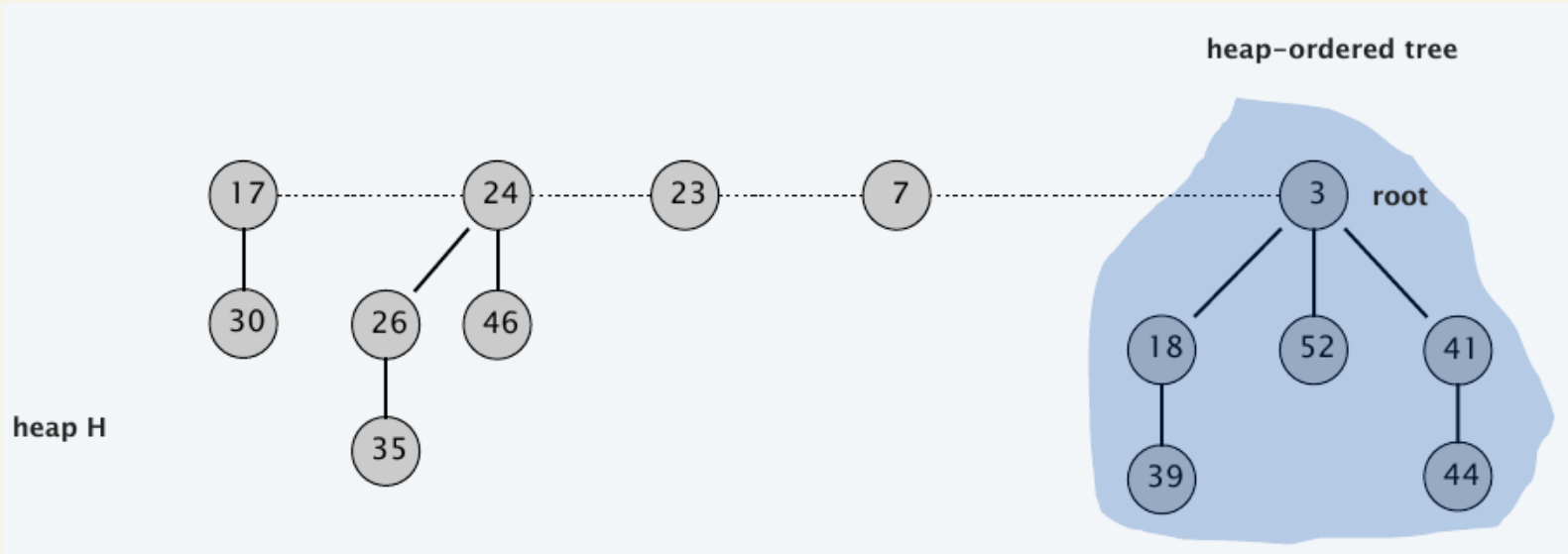
R.E. Tarjan

- A Fibonacci heap is a data structure that efficiently supports the standard operations of a priority queue (`insert`, `min`, `extract_min`) and additionally it supports: (1) the melding (merging) of two queues in amortized time $\mathcal{O}(\log n)$, (2) the deletion of an arbitrary element in amortized time $\mathcal{O}(\log n)$ and (3) the decrease of the priority of an element in amortized constant time $\mathcal{O}(1)$.
- Fibonacci heaps were invented by Michael L. Fredman and Robert E. Tarjan in 1986.

```
template <typename Elem>
class FibonacciHeap {
public:
  class FH_handle;

  // creates an empty Fibonacci heap
  FibonacciHeap();
  ...

  // Add element x with integer priority p to the priority queue
  // returns a "pointer" (FH_handle) to the inserted element
  FH_handle insert(const Elem& x, int p);

  // Returns a handle to the element of minimum priority. Throws an exception if
  // the priority queue is empty
  FH_handle min() const;

  // Returns the minimum priority in the queue. Throws an exception
  // if the priority queue is empty
  int min_prio();

  // Removes an element of minimum priority from the queue. Throws
  // an exception if the prioirty queue is empty
  void extract_min();

  // Removes the element pointed to by the given handle h
  void delete(FH_handle p);

  // Decrease the priority of the element pointed to by the handle
  // to make it p (< old priority of h)
  void decrease_prio(FH_handle h, int p);

  // Returns true if and only if the queue is empty
  bool empty() const;

  // Melds (merges) the priority queue with the priority queue Q;
  // the priority queue Q becomes empty
  void meld(FibonacciHeap& F);
```

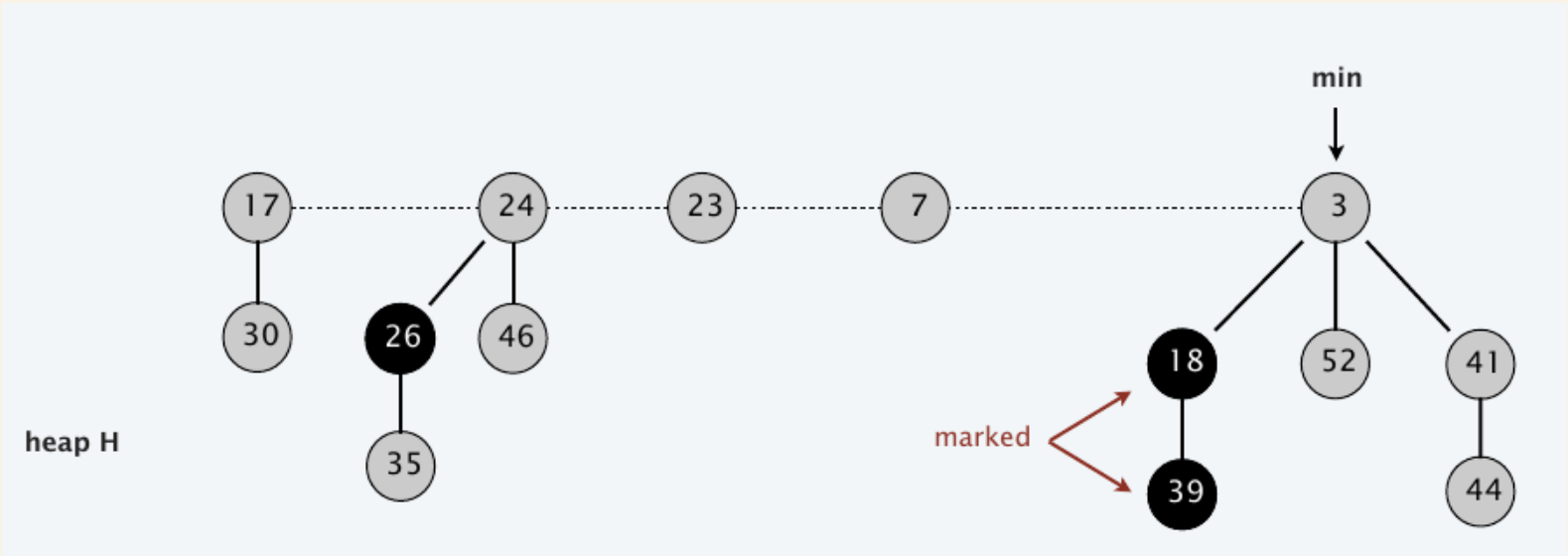
Fibonacci Heaps

- A Fibonacci heap is a collection of heap-ordered trees (every node contains a priority smaller or equal to that of its descendents).



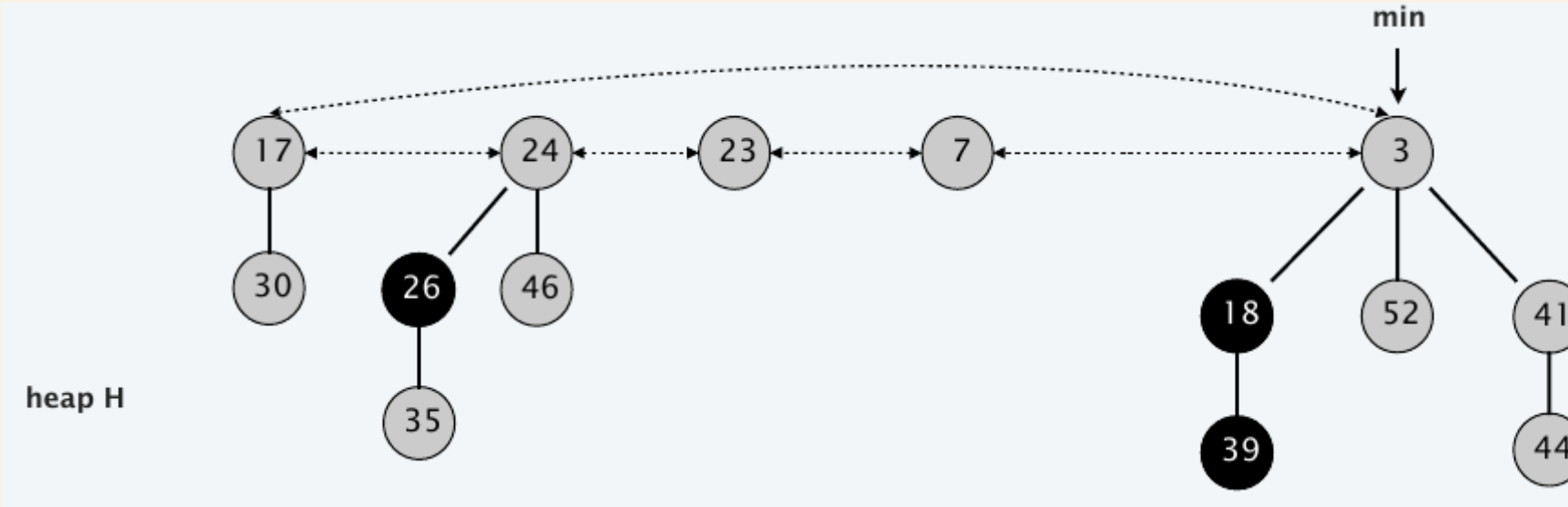
Fibonacci Heaps

- Some nodes in a FH can be marked, other are *unmarked*.



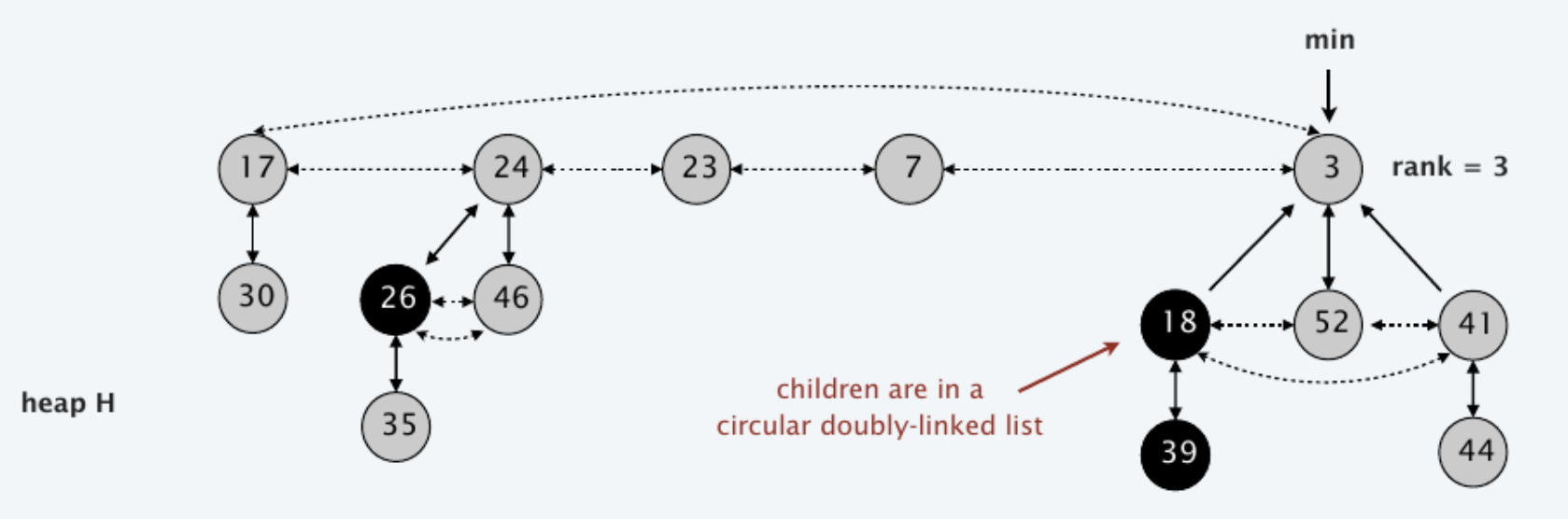
Fibonacci Heaps

- The roots of the collection of trees in a FH are maintained in a double circular linked list
- A pointer `min` points to a root with minimum priority



Fibonacci Heaps

- The children of each node are also maintained in a circular double linked list
- Each node has a pointer to one of its children, and also a pointer to its parent
- The rank of a node is the number of children of the node
- The rank of the FH is the maximum rank



Fibonacci Heaps

```
class FibonacciHeap {
...
private:
   struct FH_node {
      FH_node* parent;
      FH_node* a_child;
      FH_node* left_sibling, * right_sibling;
      int rank;
      bool mark;
      Elem info;
      int prio;
   };
   FH_node* min;
   int rank;
   ...
}
```

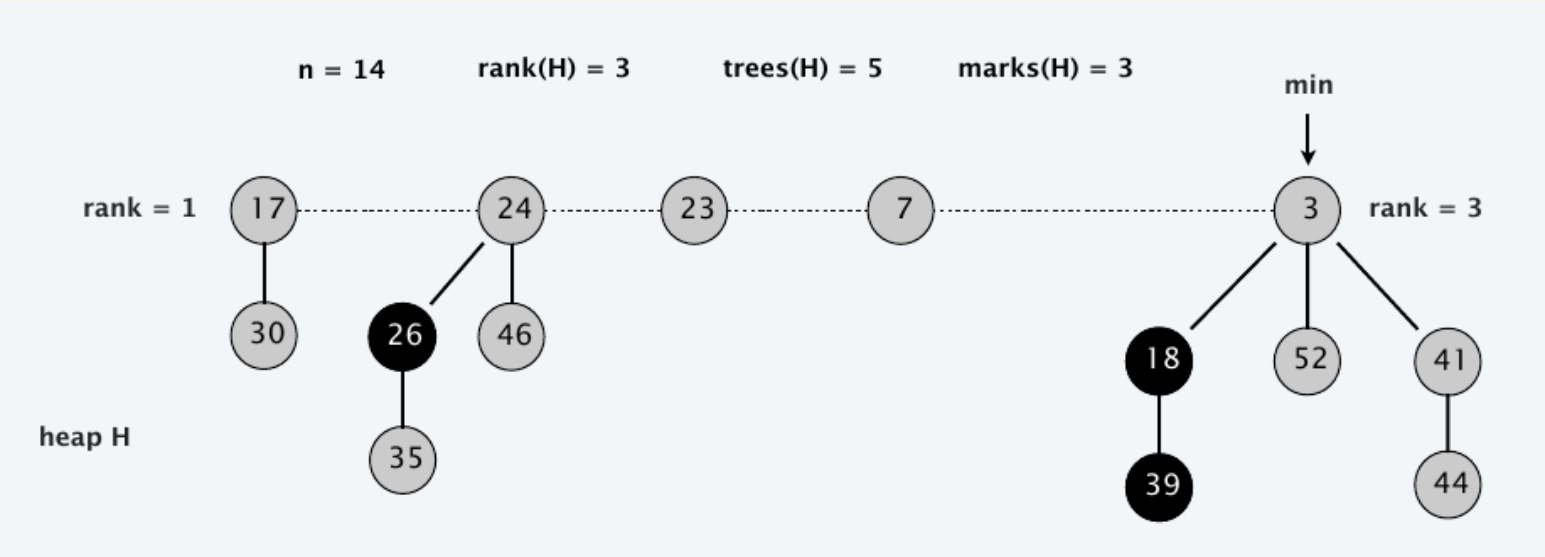
Fibonacci Heaps

Operations in constant time

- Find an element of minimum priority.
- Merge to root lists (= concatenate the two linked lists)
- Obtain the rank of a given node (via a `FH_handle`)
- Add or remove a tree (via a handle to its root) to a root list
- Remove a subtree of some node and add it to the root list
- Add a subtree as a child of some node (add it to the linked lists of children of that node)

Fibonacci Heaps

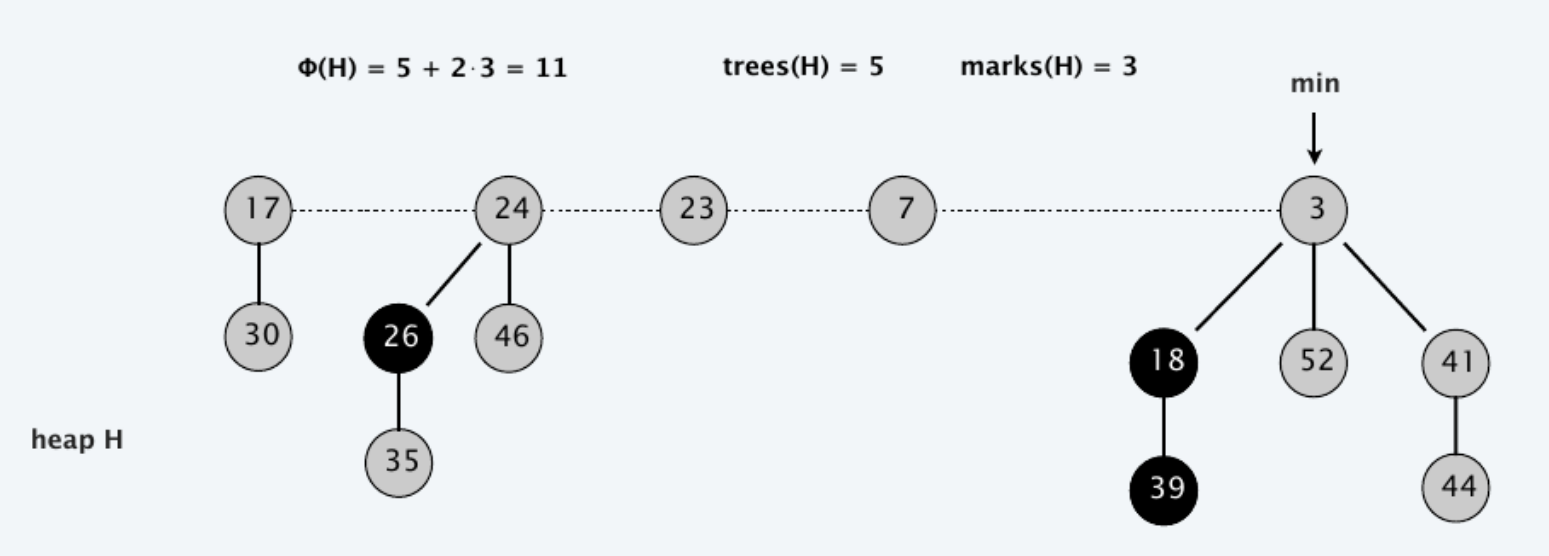
Notation	Meaning
n	number of elements
rank(x)	rank of x = number of children of node x
rank(H)	max. rank of any node in H
trees(H)	number of trees in H
marks(H)	number of marked nodes in H



Fibonacci Heaps

Potential function:

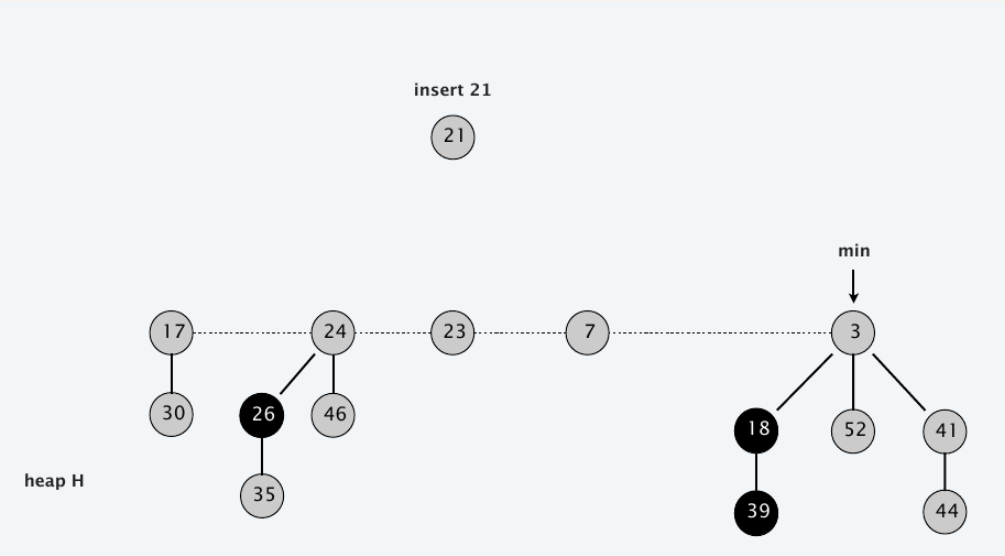
$$\Phi(H) = \text{trees}(H) + 2\text{marks}(H)$$



Fibonacci Heaps

To insert a new element x with priority p

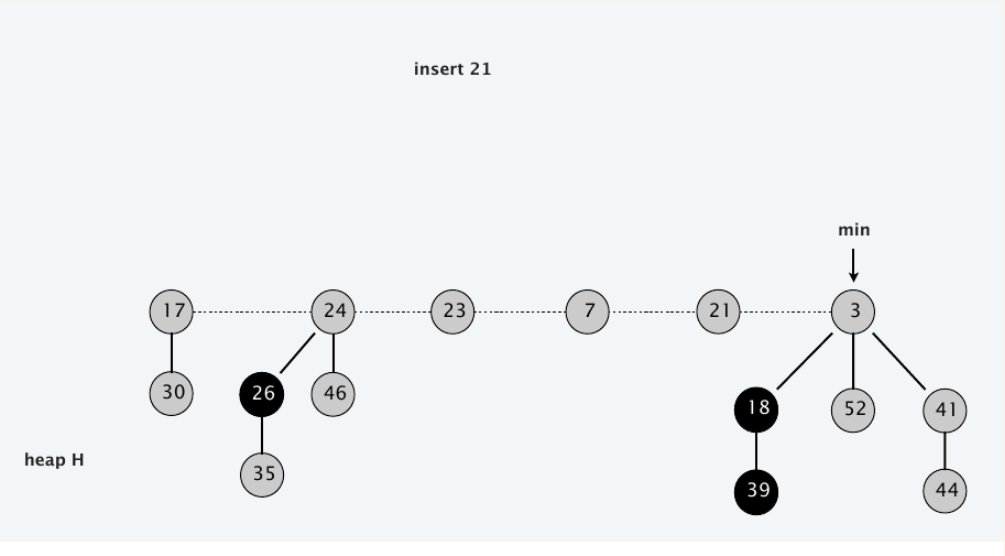
- Create a new node with x and p (and no children, no parent, . . .)
- Add the “tree” with x to the root list, and update `min` if needed



Fibonacci Heaps

To insert a new element x with priority p

- Create a new node with x and p (and no children, no parent, . . .)
- Add the “tree” with x to the root list, and update `min` if needed

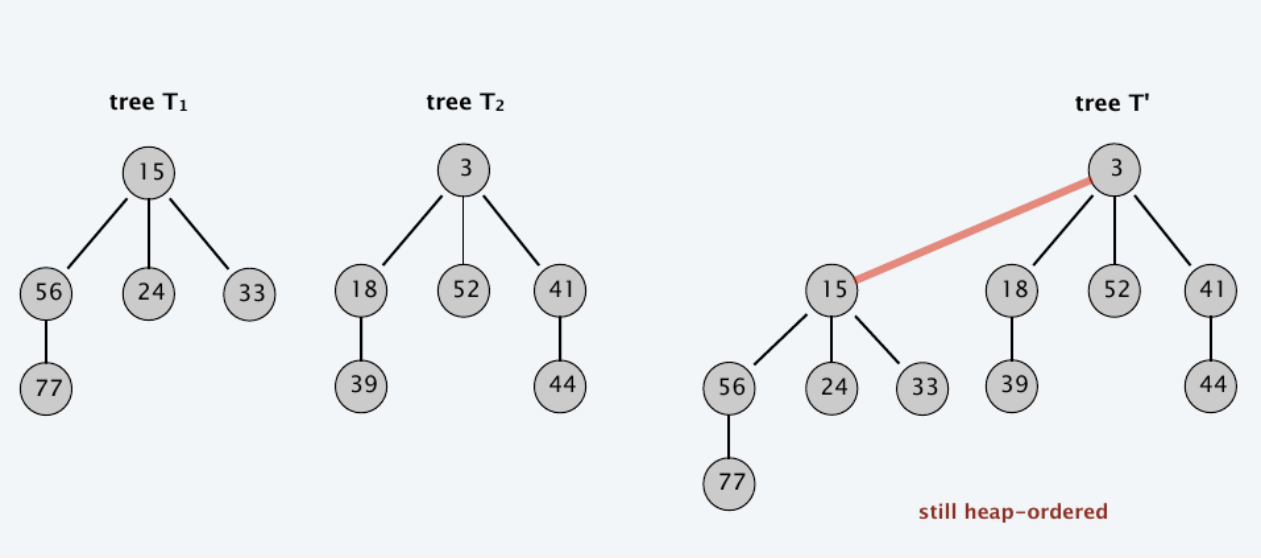


Fibonacci Heaps

- Actual cost: $c_i = \mathcal{O}(1)$
- Change in potential: $\Delta\Phi = \Phi(H') - \Phi(H) = 1$, there is one more tree after insertion
- Amortized cost: $\hat{c}_i = \mathcal{O}(1) + \Delta\Phi = \mathcal{O}(1)$

Fibonacci Heaps

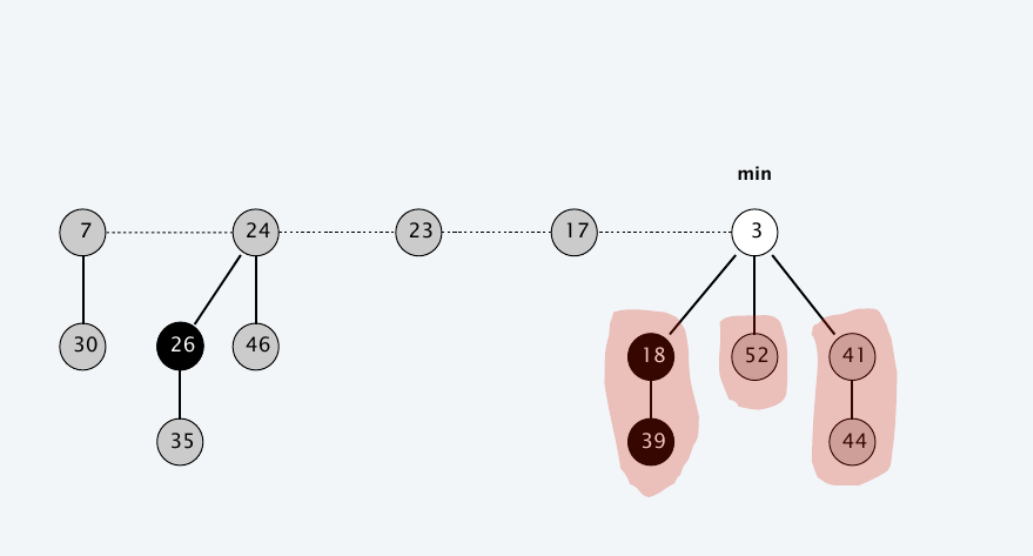
Linking:
Given two trees of rank k (rank of a tree = rank of its root), linking T_1 and T_2 yields a tree of rank $k+1$, adding the root with larger priority as a child of the root with smaller priority.



Fibonacci Heaps

To extract an element of minimum priorty:

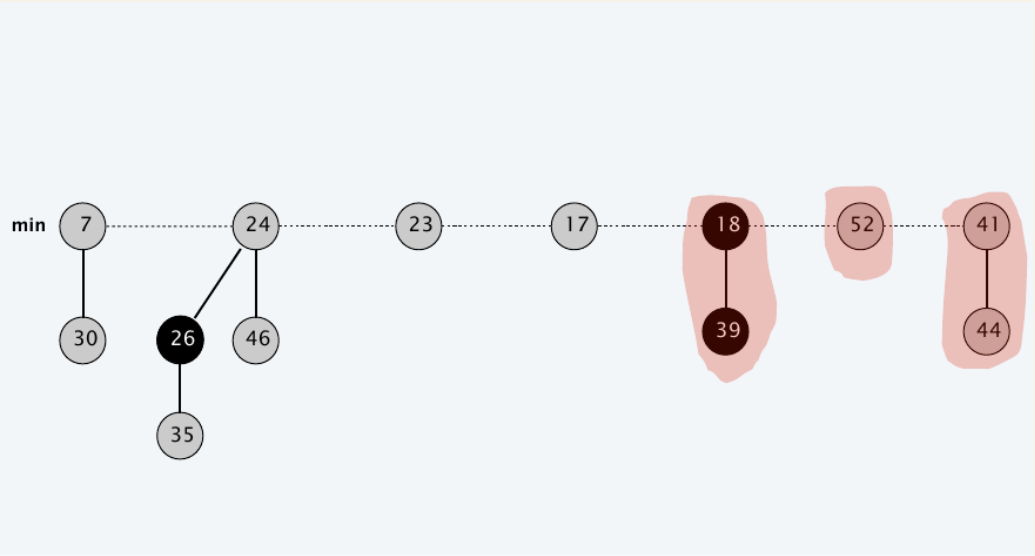
1. Remove the `min`, merging the children with the root list, and update `min`
2. Consolidate the root list (= keep linking trees until no two trees of the same rnk remain)



Fibonacci Heaps

To extract an element of minimum priorty:

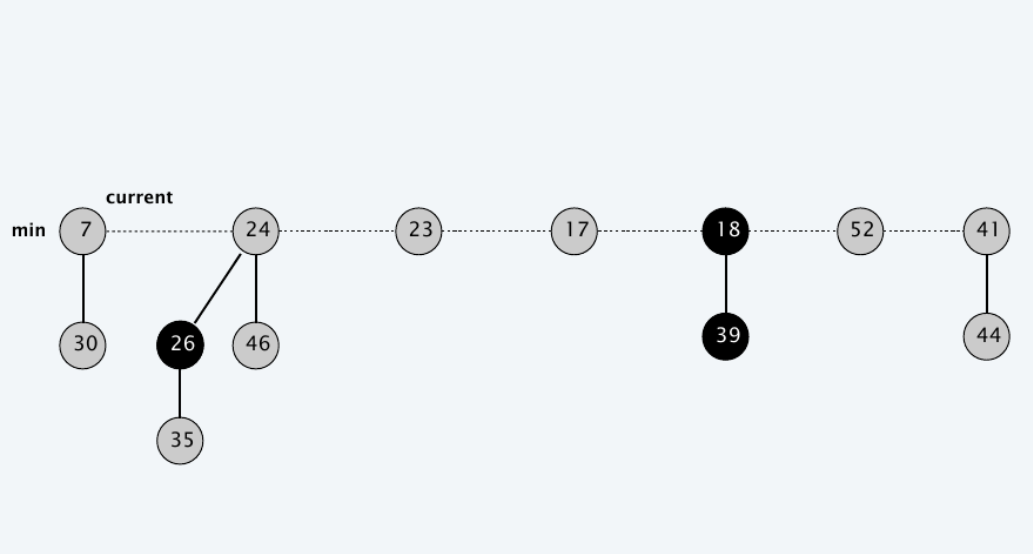
1. Remove the `min`, merging the children with the root list, and update `min`
2. Consolidate the root list (= keep linking trees until no two trees of the same rnk remain)



Fibonacci Heaps

To extract an element of minimum priorty:

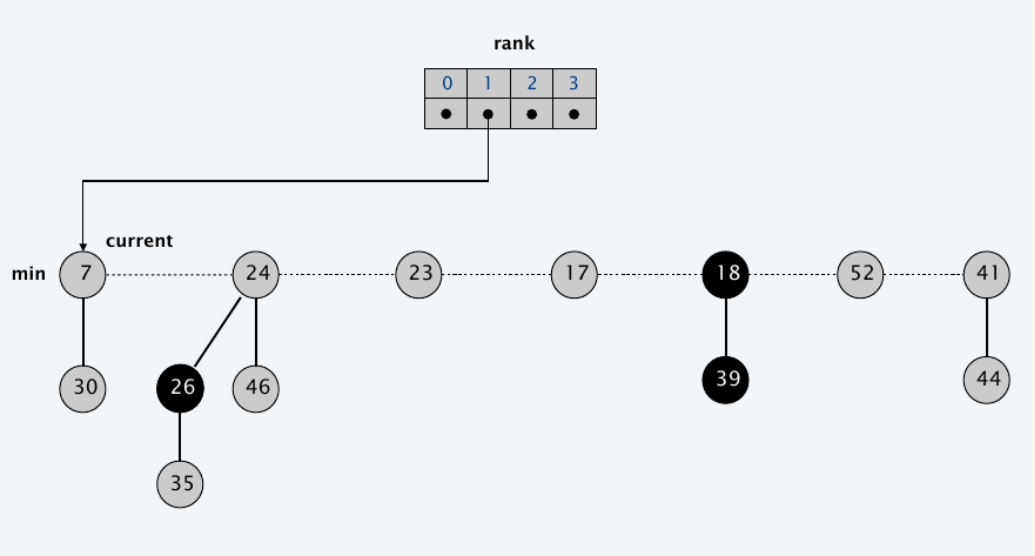
1. Remove the root pointed to by min, merging the children with the root list, and update min
2. Consolidate the root list (= link trees so that no two trees of the same rnk remain)



Fibonacci Heaps

To extract an element of minimum priorty:

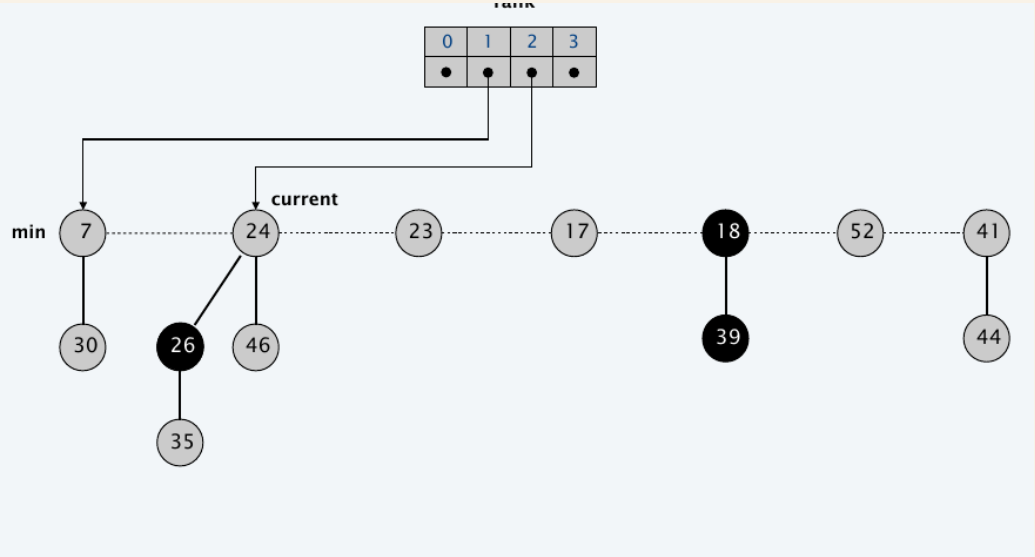
1. Remove the root pointed to by min, merging the children with the root list, and update min
2. Consolidate the root list (= link trees so that no two trees of the same rnk remain)



Fibonacci Heaps

To extract an element of minimum priorty:

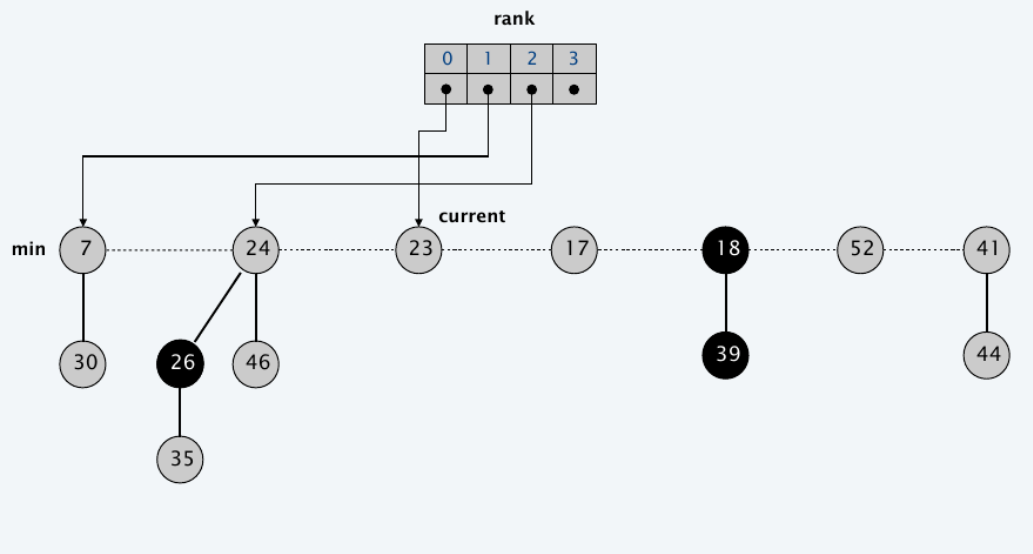
1. Remove the root pointed to by min, merging the children with the root list, and update min
2. Consolidate the root list (= link trees so that no two trees of the same rnk remain)



Fibonacci Heaps

To extract an element of minimum priorty:

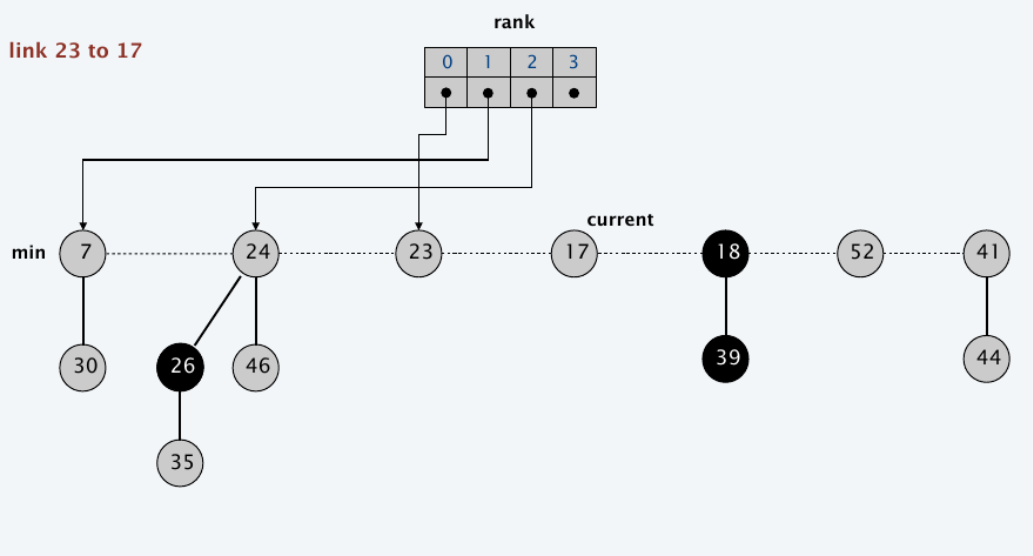
1. Remove the root pointed to by min, merging the children with the root list, and update min
2. Consolidate the root list (= link trees so that no two trees of the same rnk remain)



Fibonacci Heaps

To extract an element of minimum priorty:

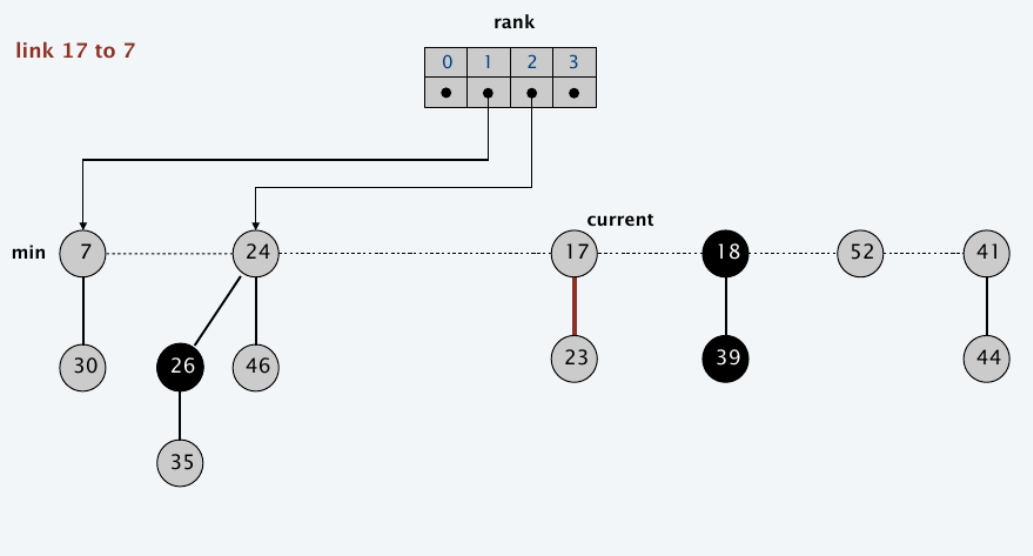
1. Remove the root pointed to by min, merging the children with the root list, and update min
2. Consolidate the root list (= link trees so that no two trees of the same rnk remain)



Fibonacci Heaps

To extract an element of minimum priorty:

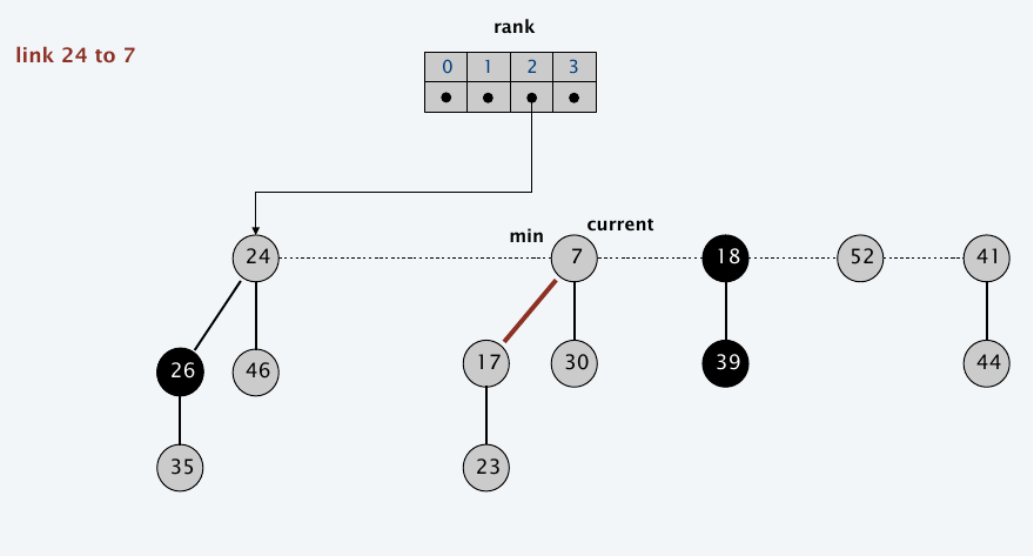
1. Remove the root pointed to by min, merging the children with the root list, and update min
2. Consolidate the root list (= link trees so that no two trees of the same rnk remain)



Fibonacci Heaps

To extract an element of minimum priorty:

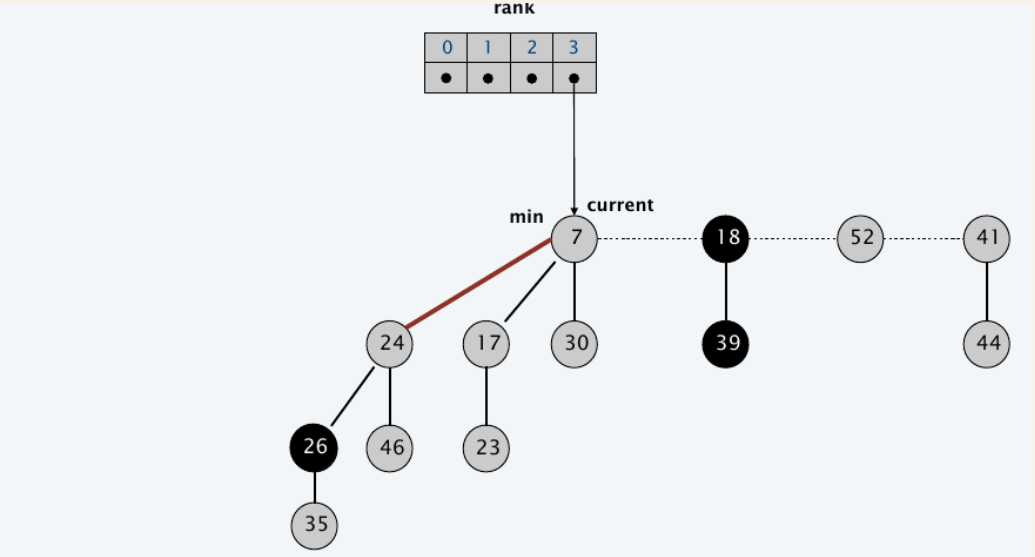
1. Remove the root pointed to by min, merging the children with the root list, and update min
2. Consolidate the root list (= link trees so that no two trees of the same rnk remain)



Fibonacci Heaps

To extract an element of minimum priorty:

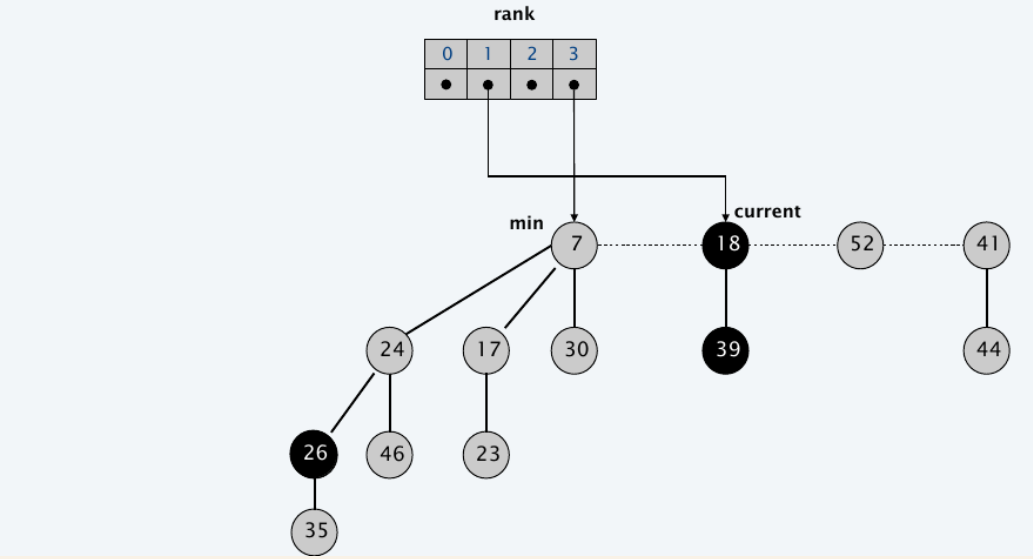
1. Remove the root pointed to by min, merging the children with the root list, and update min
2. Consolidate the root list (= link trees so that no two trees of the same rnk remain)



Fibonacci Heaps

To extract an element of minimum priorty:

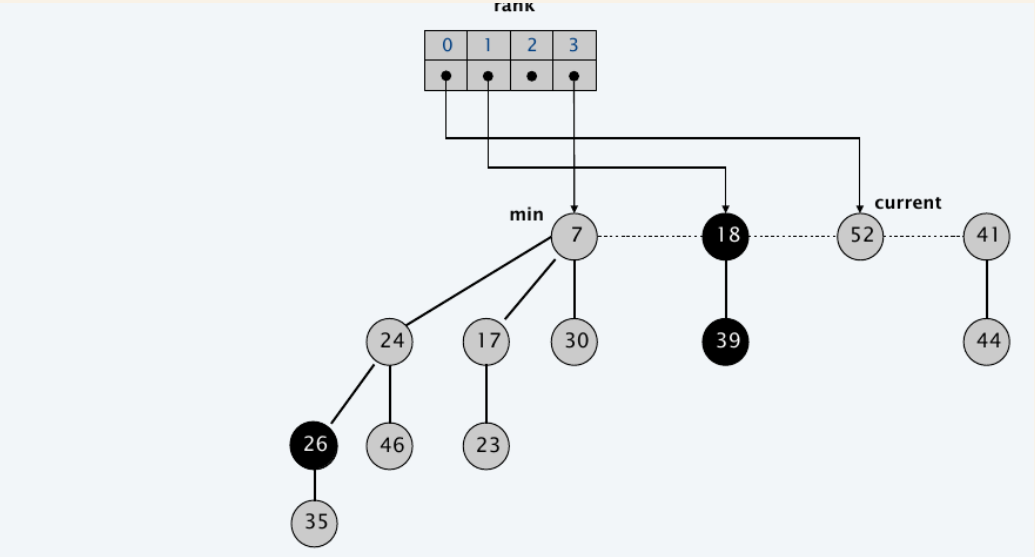
1. Remove the root pointed to by min, merging the children with the root list, and update min
2. Consolidate the root list (= link trees so that no two trees of the same rnk remain)



Fibonacci Heaps

To extract an element of minimum priorty:

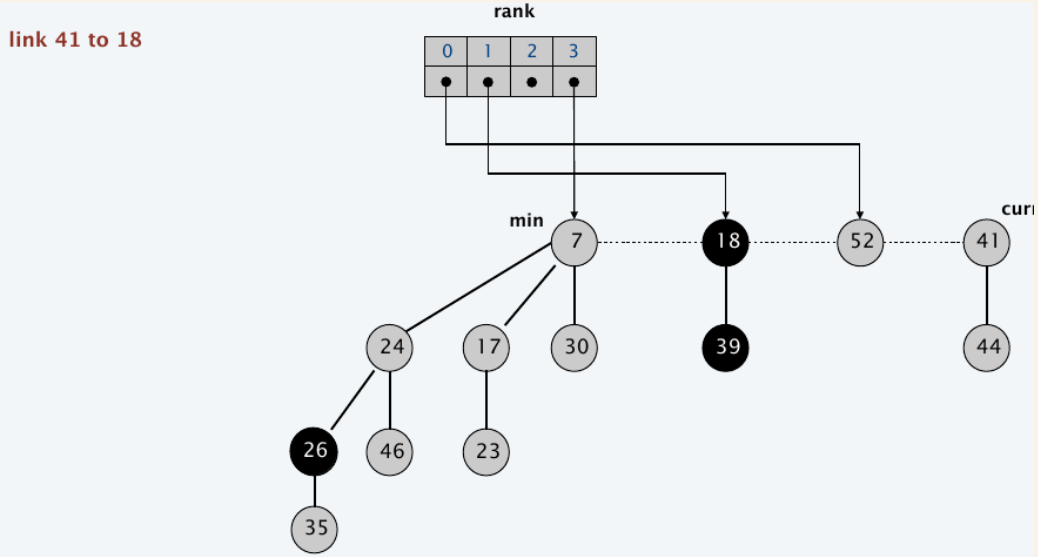
1. Remove the root pointed to by min, merging the children with the root list, and update min
2. Consolidate the root list (= link trees so that no two trees of the same rnk remain)



Fibonacci Heaps

To extract an element of minimum priorty:

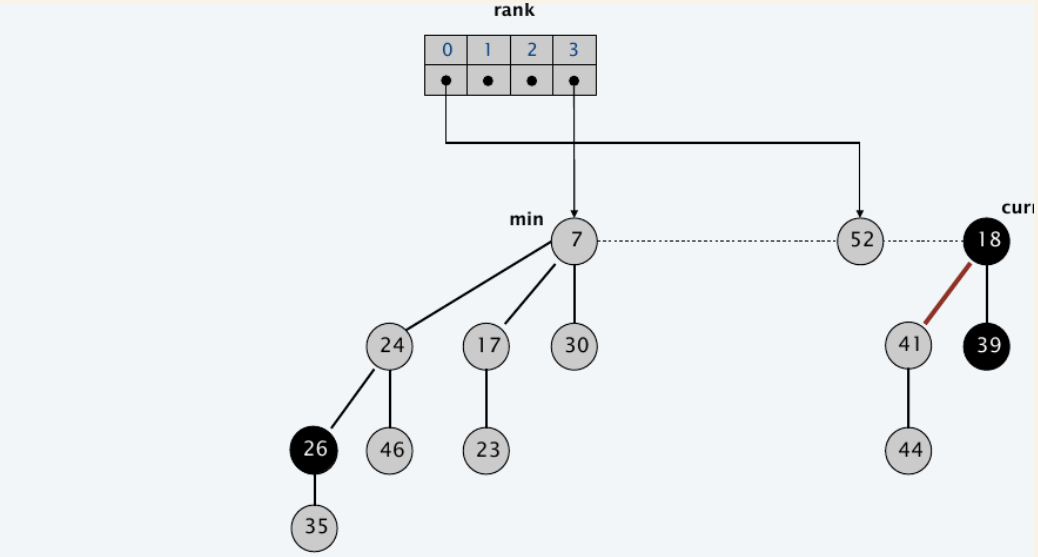
1. Remove the root pointed to by min, merging the children with the root list, and update min
2. Consolidate the root list (= link trees so that no two trees of the same rnk remain)



Fibonacci Heaps

To extract an element of minimum priorty:

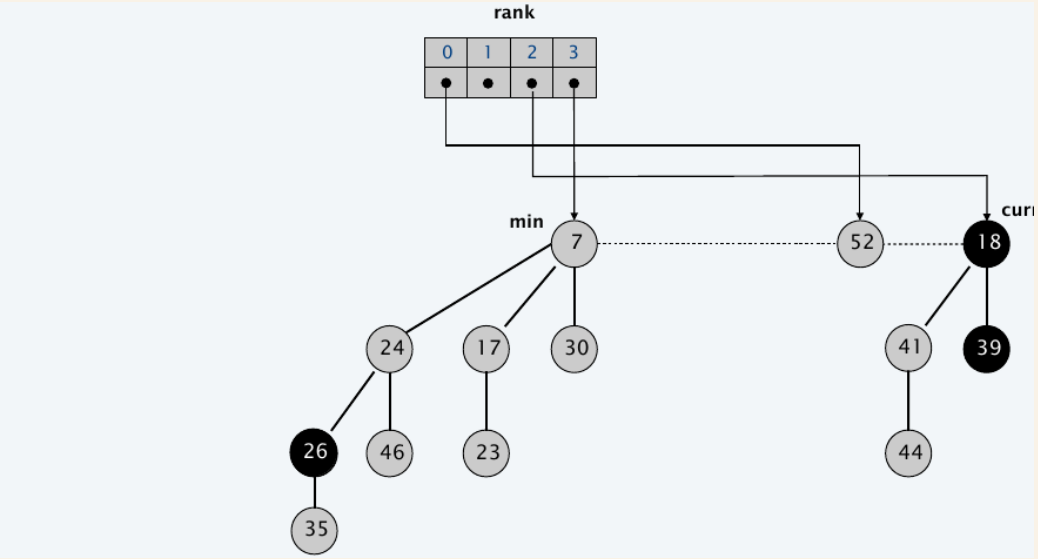
1. Remove the root pointed to by min, merging the children with the root list, and update min
2. Consolidate the root list (= link trees so that no two trees of the same rnk remain)



Fibonacci Heaps

To extract an element of minimum priorty:

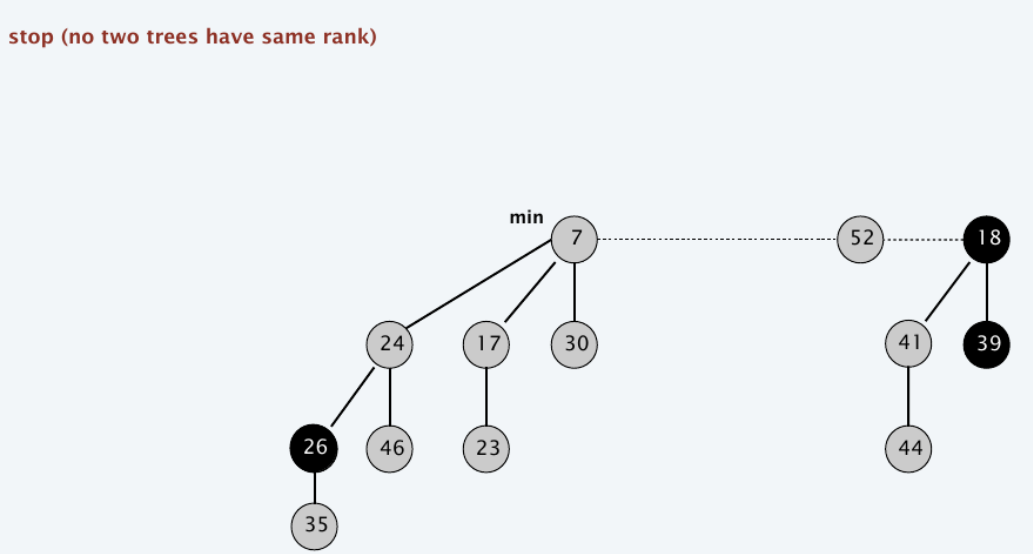
1. Remove the root pointed to by min, merging the children with the root list, and update min
2. Consolidate the root list (= link trees so that no two trees of the same rnk remain)



Fibonacci Heaps

To extract an element of minimum priorty:

1. Remove the root pointed to by min, merging the children with the root list, and update min
2. Consolidate the root list (= link trees so that no two trees of the same rank remain)



Fibonacci Heaps

- Actual cost: $c_i = \mathcal{O}(\text{rank}(H)) + \text{trees}(H)$
  - Removing min + merging children: $\mathcal{O}(\text{rank}(H))$
  - Update min: $\mathcal{O}(\text{rank}(H)) + \text{trees}(H) - 1$
  - Consolidate: $\mathcal{O}(\text{rank}(H)) + \text{trees}(H) - 1$
- Change in potential: $\Delta\Phi = \Phi(H') - \Phi(H) = \text{trees}(H') - \text{trees}(H) \leq \text{rank}(H') + 1 - \text{trees}(H)$, as no two trees have same rank after consolidation
- Amortized cost: $\hat{c}_i = c_i + \Delta\Phi = \mathcal{O}(\text{rank}(H')) + \mathcal{O}(\text{rank}(H))$. We will show later that $\text{rank}(H) = \mathcal{O}(\log n)$ if H has n elements.

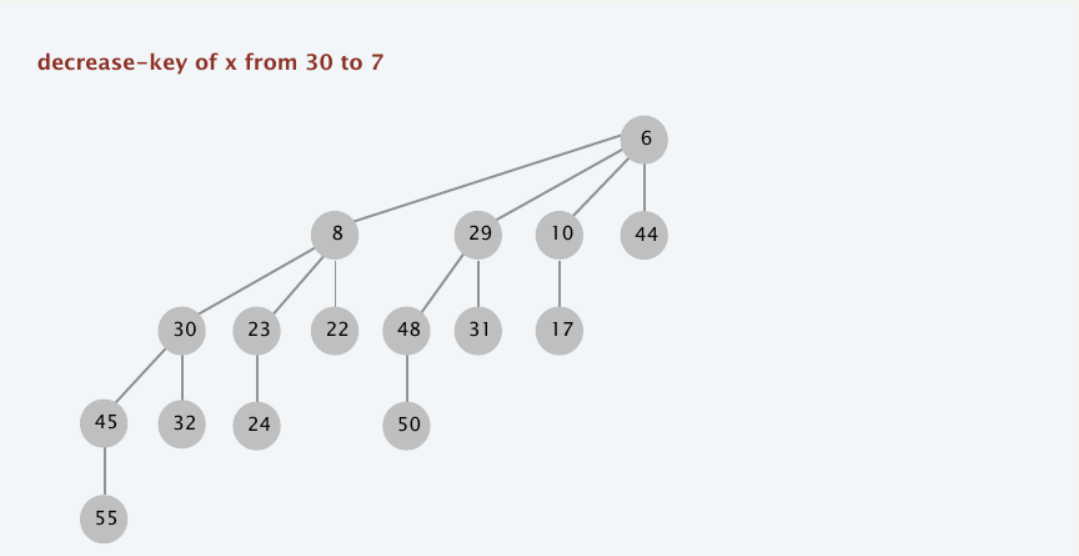
Fibonacci Heaps

- If there are only insertions and extract_min, all trees in a Fibonacci heap H are binomial trees (rank = order of the binomial trees)
- Hence $\text{rank}(H) \leq \log_2 n$
- Decrease_priority: trees are no longer binomial, but $\text{rank}(H) \leq \log_\phi n$, with $\phi = (1 + \sqrt{5})/2 \approx 1.618\ldots$

Fibonacci Heaps

To decrease the pripority of node x from p to p'

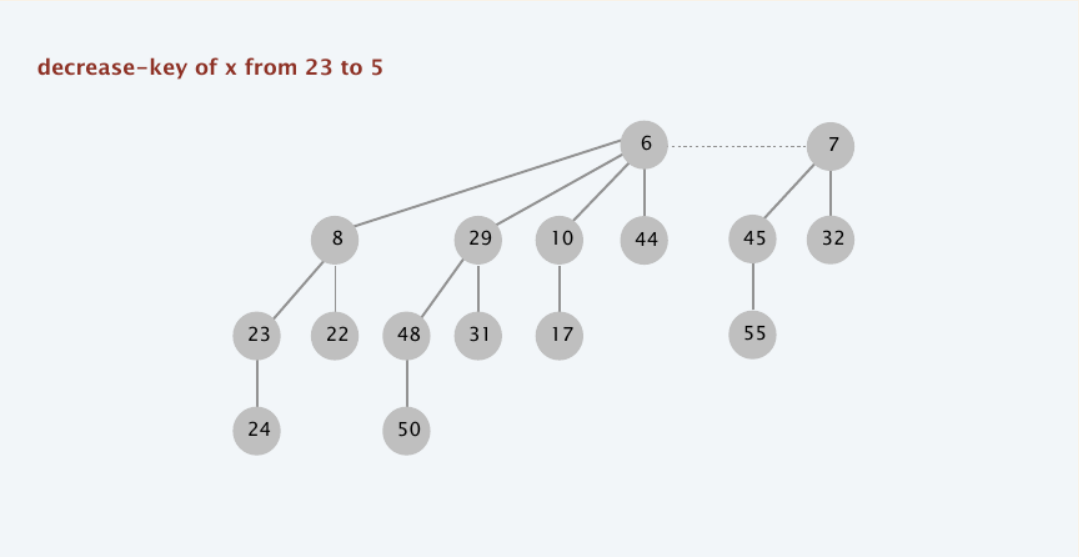
- If heap-order is preserved, update priority to p'
- Otherwise, cut the subtree rooted at x and add it to the root list
- Update `min` if necessary



Fibonacci Heaps

To decrease the pripority of node x from p to p'

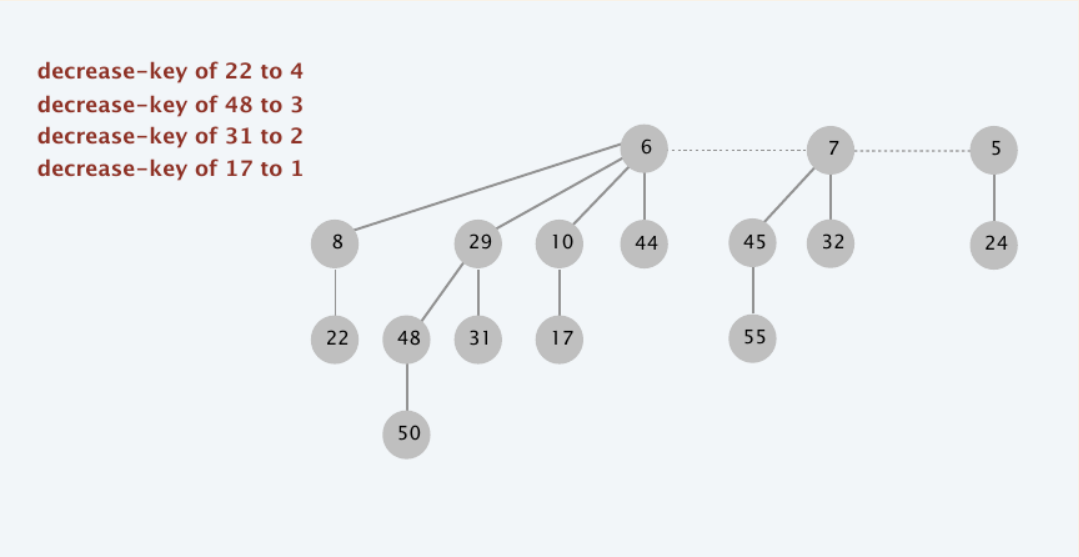
- If heap-order is preserved, update priority to p'
- Otherwise, cut the subtree rooted at x and add it to the root list
- Update `min` if necessary



Fibonacci Heaps

To decrease the pripority of node x from p to p'

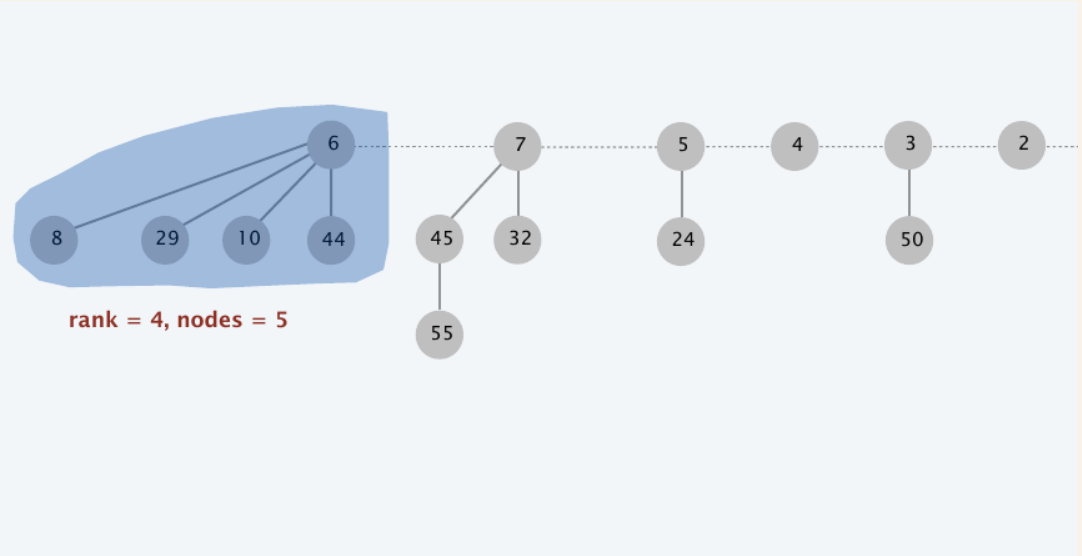
- If heap-order is preserved, update priority to p'
- Otherwise, cut the subtree rooted at x and add it to the root list
- Update `min` if necessary



Fibonacci Heaps

To decrease the pripority of node x from p to p'

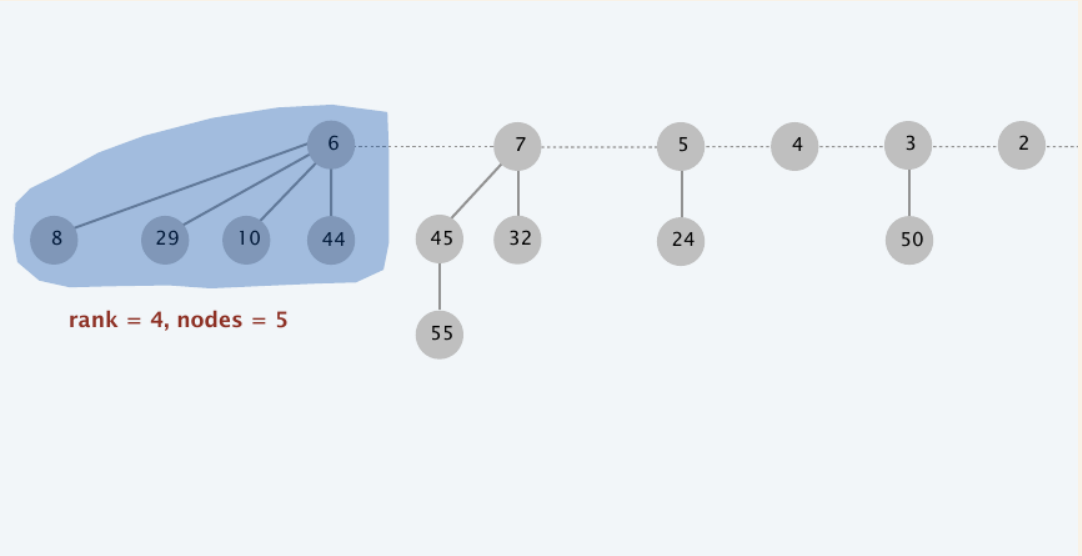
- If heap-order is preserved, update priority to p'
- Otherwise, cut the subtree rooted at x and add it to the root list
- Update `min` if necessary



Fibonacci Heaps

To decrease the pripority of node x from p to p'

- If heap-order is preserved, update priority to p'
- Otherwise, cut the subtree rooted at x and add it to the root list
- The number of nodes can be less than exponential w.r.t. the rank (e.g., trees with size $\approx$ rank)



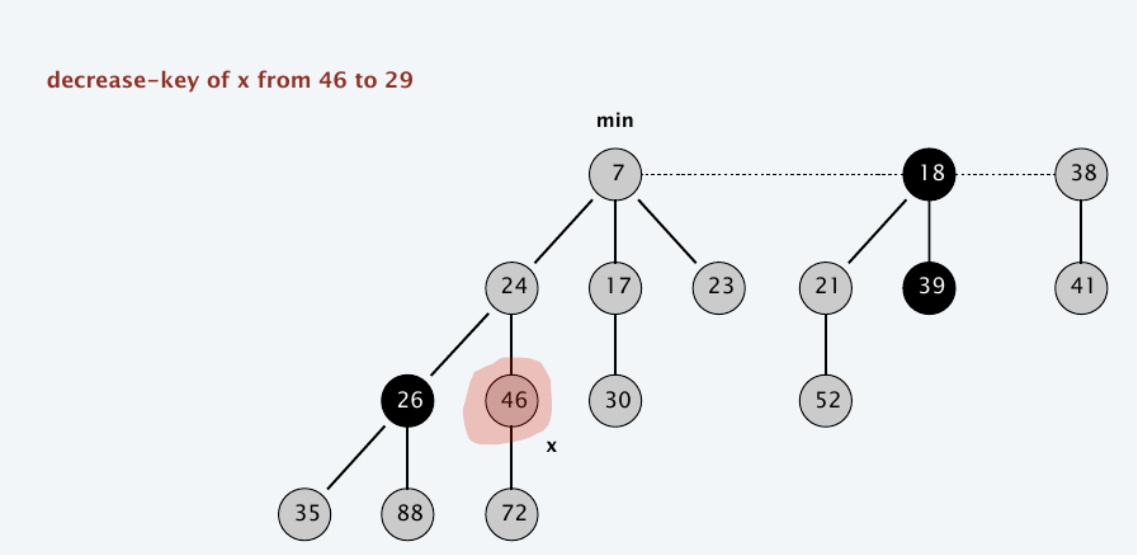
Fibonacci Heaps

To solve the problem:

1. If a node x loses a child for the first time, cut the child but also mark x
2. If you cut a child from a marked node x then cut the subtree rooted at x too and merge it with the root list
3. The roots of cut subtrees that are added to the root list lose their marks if they had marks

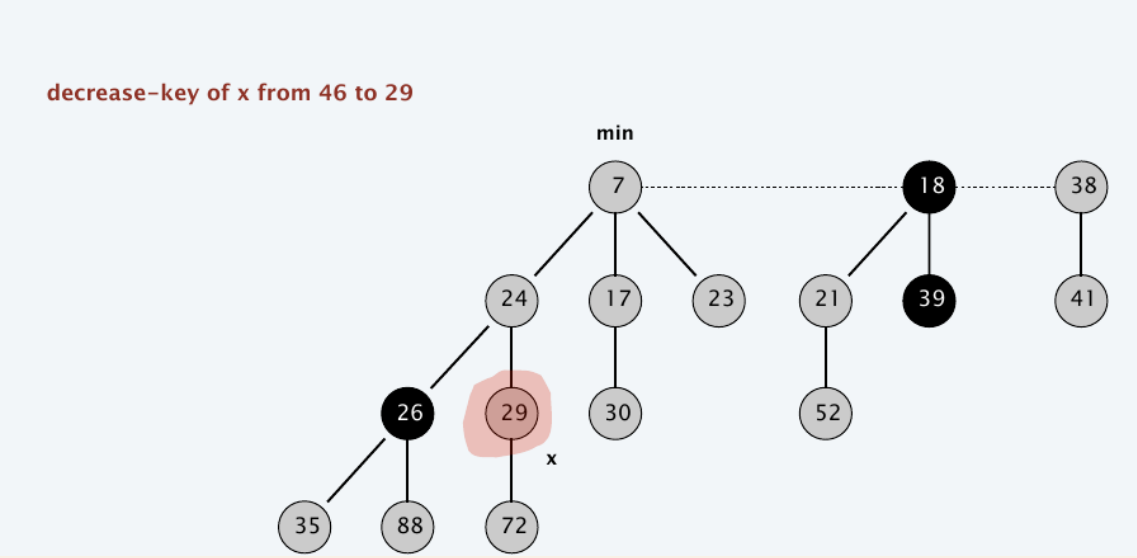
Fibonacci Heaps

Case 1: heap-order is preserved



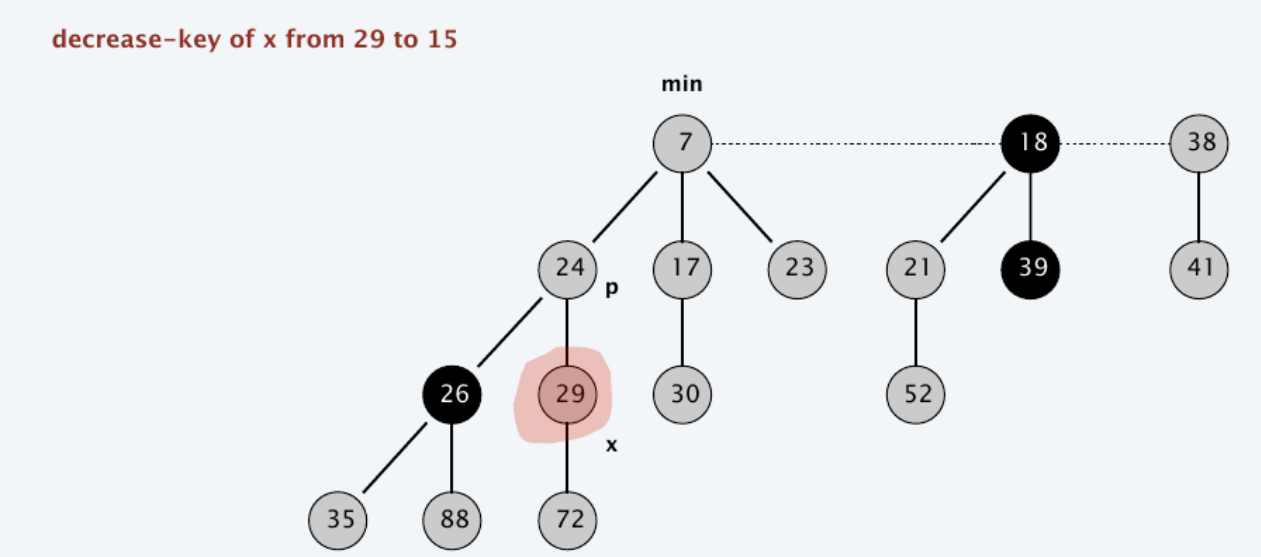
Fibonacci Heaps

Case 1: heap-order is preserved



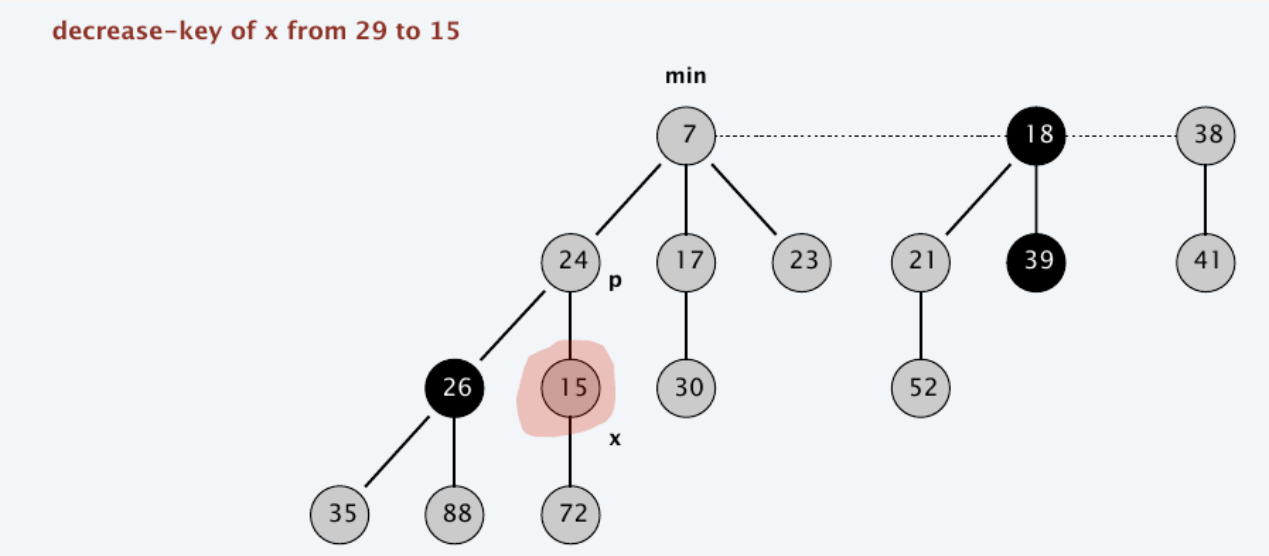
Fibonacci Heaps

Case 2: parent is not marked



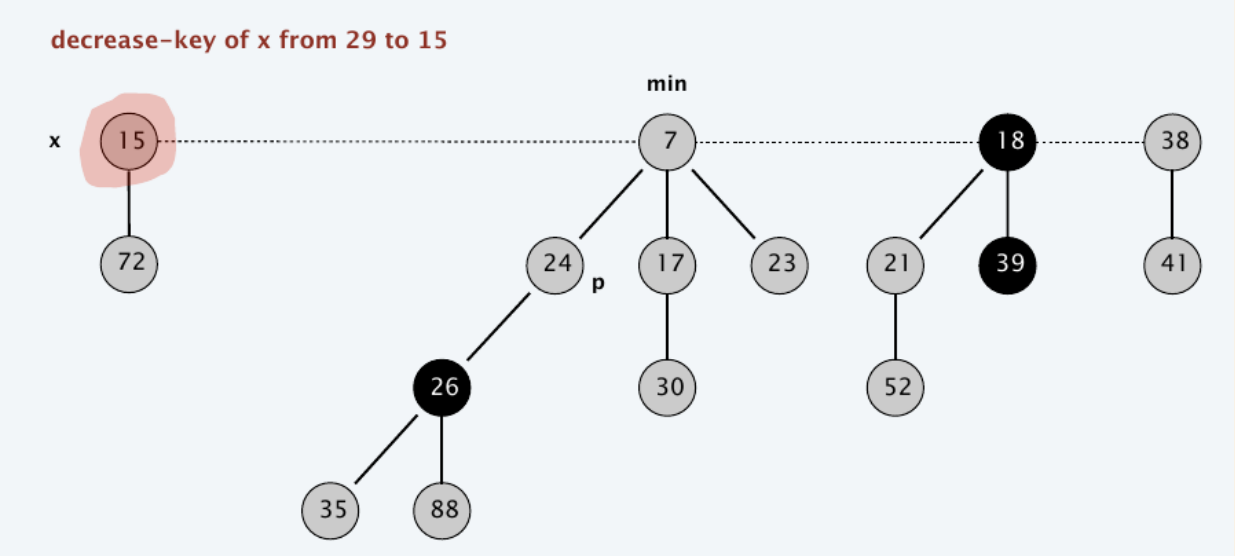
Fibonacci Heaps

Case 2: parent is not marked



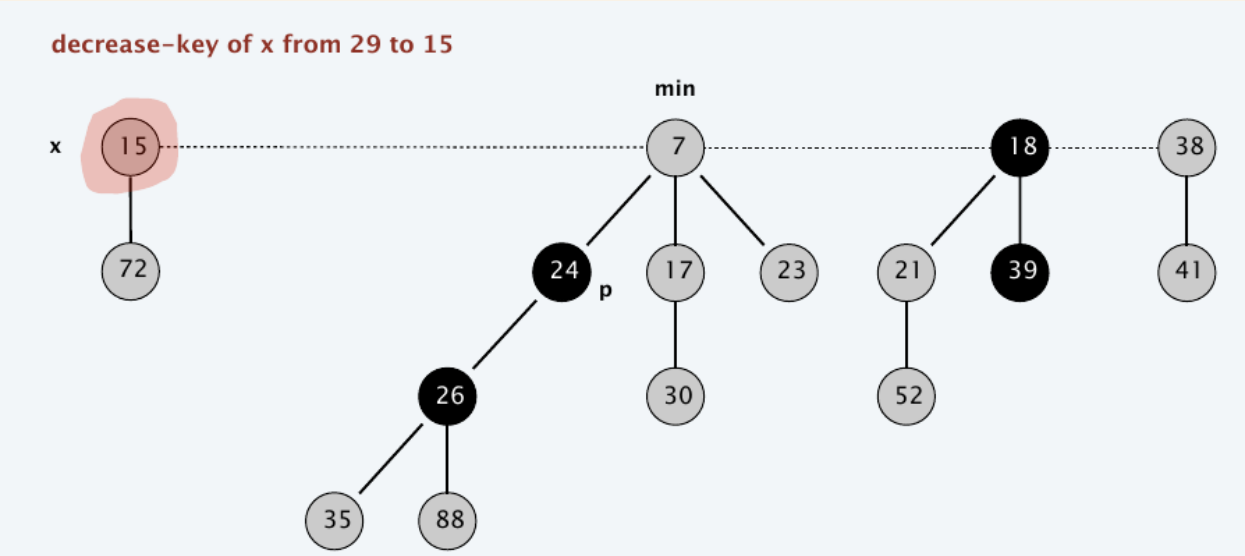
Fibonacci Heaps

Case 2: parent is not marked



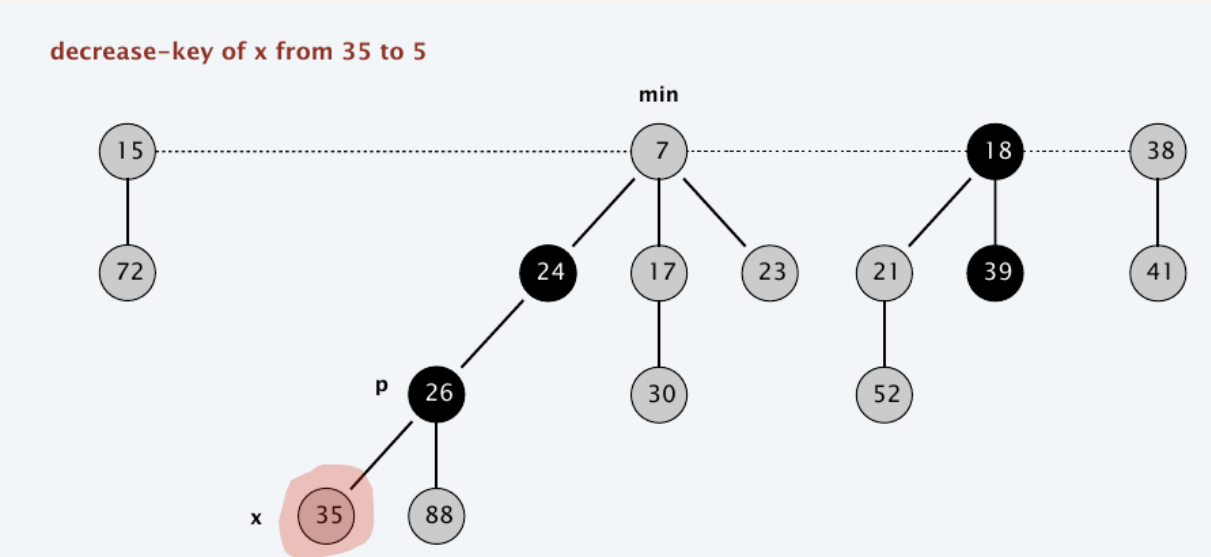
Fibonacci Heaps

Case 2: parent is not marked



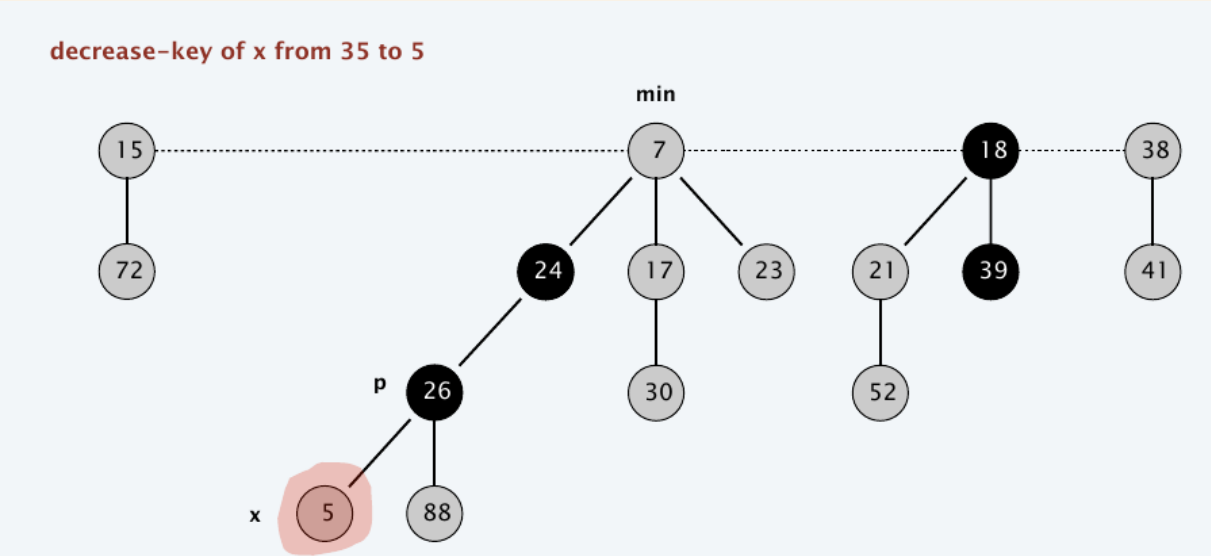
Fibonacci Heaps

Case 3: parent is marked



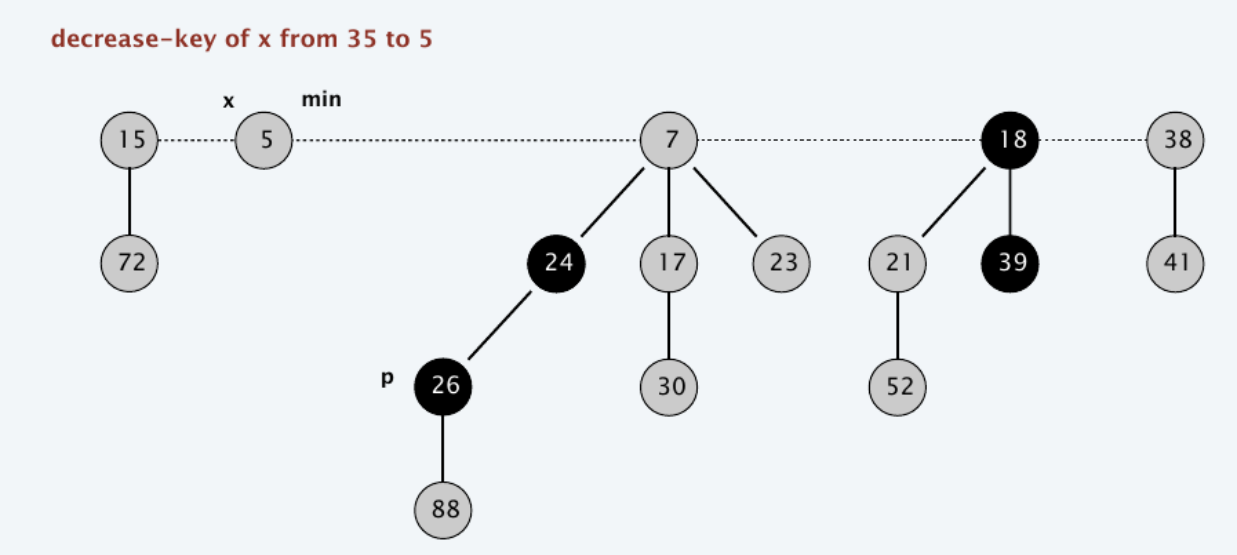
Fibonacci Heaps

Case 3: parent is marked



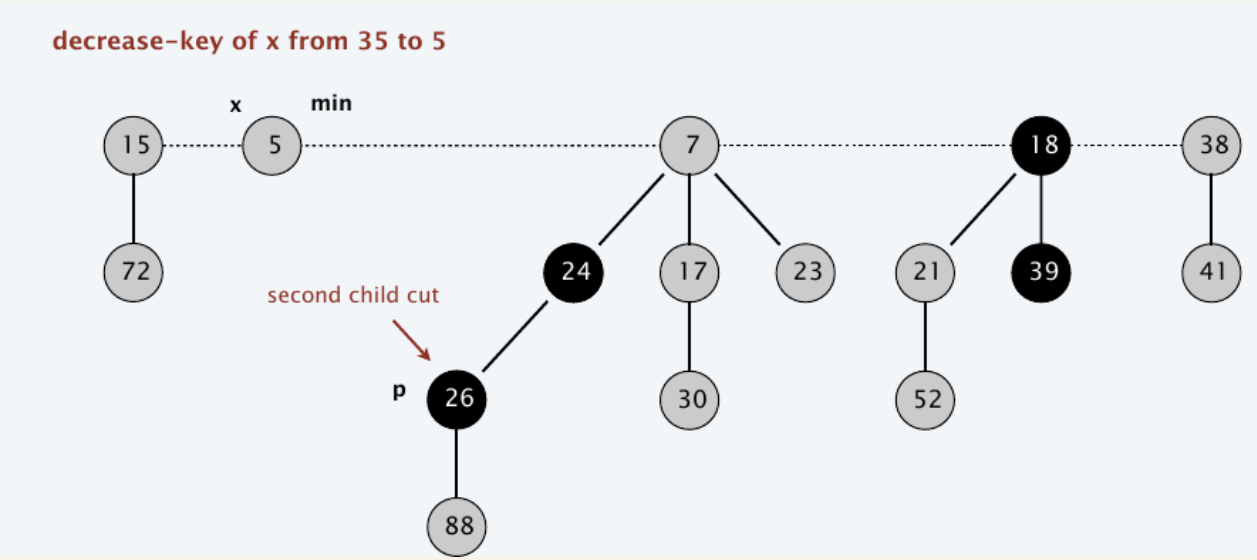
Fibonacci Heaps

Case 3: parent is marked



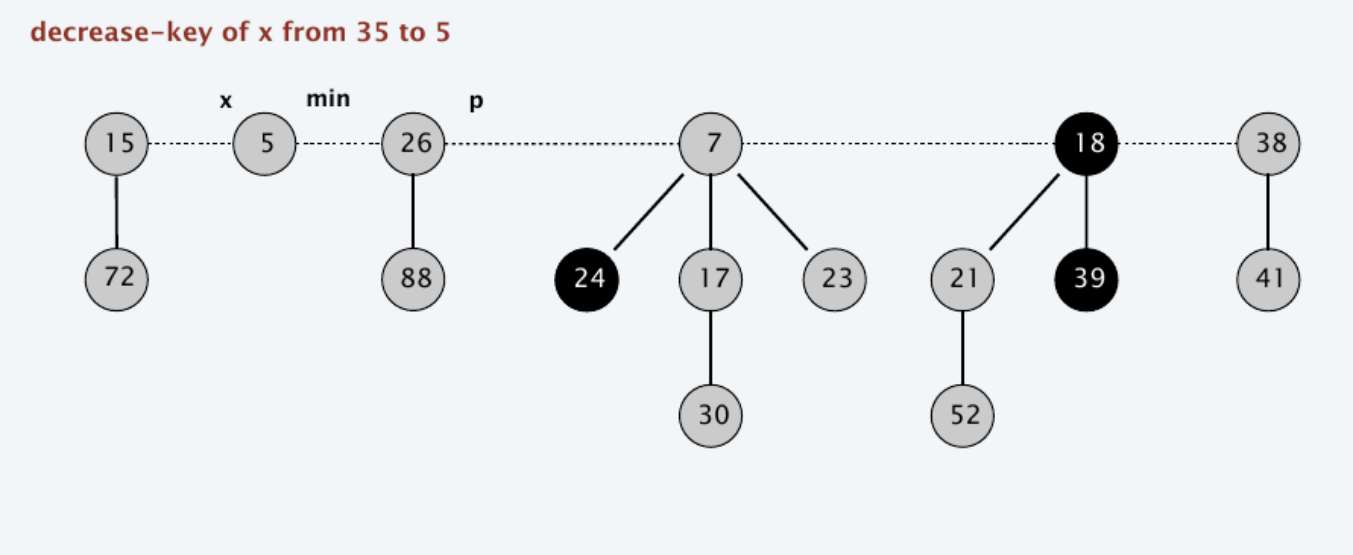
Fibonacci Heaps

Case 3: parent is marked



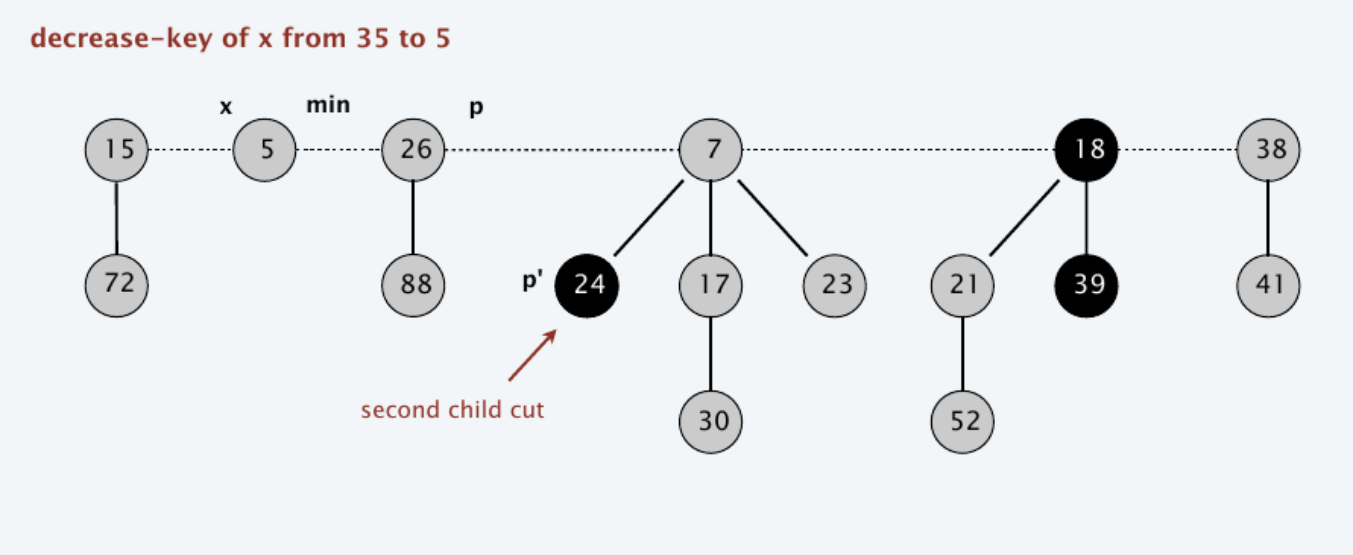
Fibonacci Heaps

Case 3: parent is marked



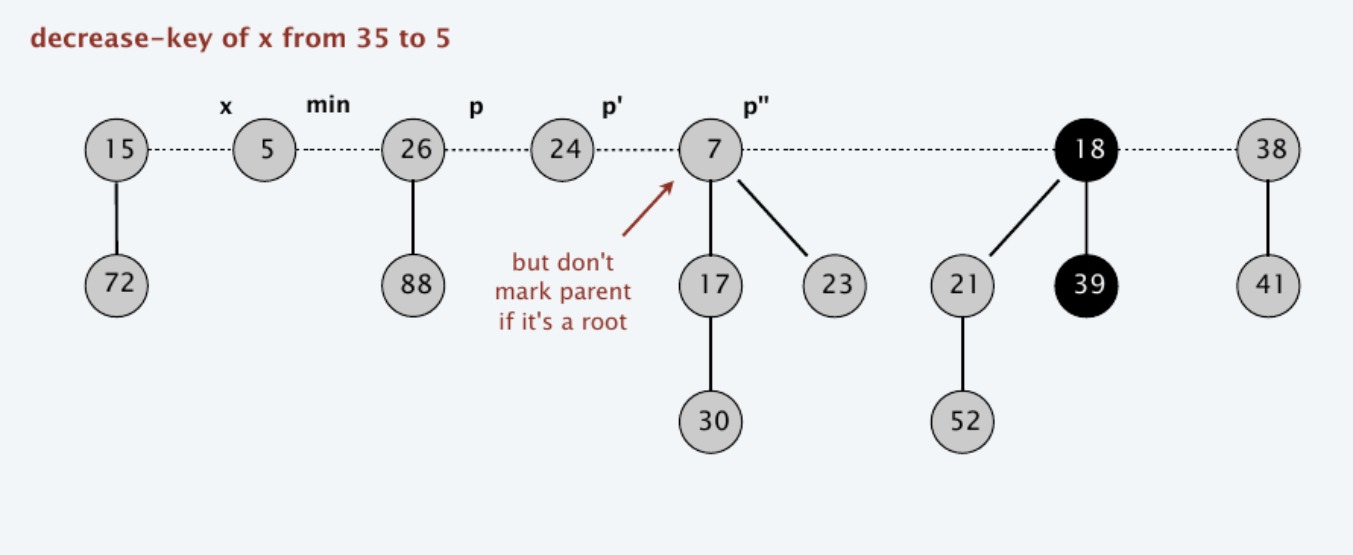
Fibonacci Heaps

Case 3: parent is marked



Fibonacci Heaps

Case 3: parent is marked



Fibonacci Heaps

- Actual cost: $c_i = c + 1$, $c =$ number of cuts; includes changing the key, merging each cutted subtree in the root list
- Change in potential: $\Delta\Phi = \mathcal{O}(1) - c$
  - $\text{trees}(H') = \text{trees}(H) + c$
  - $\text{marks}(H') \leq \text{marks}(H) + 2 - c$; every cut, except the first and the last, removes a mark; the last might or might not remove a mark
  - $\Delta\Phi \leq c + 2 \cdot (2 - c) = 4 - c$
- Amortized cost: $\hat{c}_i = c_i + \Delta\Phi = \mathcal{O}(1)$.

Fibonacci Heaps

Summary:

- Insert: $\mathcal{O}(1)$
- Extract min: $\mathcal{O}(\text{rank}(H))$ amortized
- Decrease priority: $\mathcal{O}(1)$ amortized

Last step: Fibonacci lemma.

Lemma

Let H be a Fibonacci heap with n elements. Then

$$\textit{rank}(H) \leq \log_\phi n$$

Fibonacci Heaps

Lemma

Fix some moment in time and consider a tree of rank k with root x. Denote $y_1, \ldots, y_k$ the k children of x in the order in which they have been attached as children of x. Then

$$\mathit{rank}(y_i) \geq \begin{cases} 0 & \text{if } i = 1, \\ i - 2 & \text{if } i \geq 2. \end{cases}$$

Fibonacci Heaps

1. When y_i gets linked to x both must be at least of rank $i-1$ (as x had $y_1, \ldots, y_{i-1}$ as children and possibly others that x lost later)
2. Thus $\text{rank}(x) = \text{rank}(y_{i-1}) \geq i-1$
3. Since y_i gets linked as a child of x it can lose one child, but not more; otherwise y_i would have been cut too
4. Therefore $\text{rank}(y_i) \geq i-2$

Fibonacci Heaps

Lemma

Let s_k be the smallest possible number of elements in a Fibonacci heap of rank k. Then $s_k \geq F_{k+2}$, where F_k denotes the k-th Fibonacci number

Proof

Consider a FH consisting of a single tree with root x.

- Basis: $s_0 = 1$, $s_1 = 2$. Inductive hyp: $s_i \geq F_{i+2}$ for all i, $0 \leq i < k$.
- Let $y_1, \ldots, y_k$ denote the children of x.

$$\begin{aligned} s_k &\geq 1 + 1 + (s_0 + \cdots + s_{k-2}) && \text{because of Lemma 1} \\ &\geq 1 + F_1 + (F_2 + \cdots + F_k) && \text{because of inductive hyp.} \\ &= F_k. \end{aligned}$$

Fibonacci Heaps

Two facts (both easily proved by induction):

1. For all $k \geq 0$, $F_{k+2} = 1 + F_0 + F_1 + \cdots + F_k$
2. For all $k \geq 0$, $F_{k+2} \geq \phi^k$, where $\phi = (1 + \sqrt{5})/2 \approx 1.618\ldots$

Let H be a Fibonacci heap with n elements and rank k. Then by Lemma 2, $n \geq s_k \geq F_{k+2} \geq \phi^k$. Hence,

$$k = \text{rank}(H) \leq \log_\phi n = 1.44 \log_2 n$$

This implies that the amortized cost of extract min is $\mathcal{O}(\log n)$.

Part IV

Disjoint Sets

12 Disjoint Sets: Introduction

13 Implementation of Union-Find

14 Analysis of Union-Find

Disjoint Sets

A set of *disjoint sets* or *partition* Π of a non-empty set $\mathcal{A}$ is a collection of non-empty subsets $\Pi = \{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$ such that

1. $i \neq j \implies \mathcal{A}_i \cap \mathcal{A}_j = \emptyset$
2. $\mathcal{A} = \bigcup_{1 \leq i \leq k} \mathcal{A}_i$

Each $\mathcal{A}_k$ is often called *block* or *class*; we might see a partition as an equivalence relation and each $\mathcal{A}_k$ as one of its equivalence classes.

Disjoint Sets

Given a partition Π of $\mathcal{A}$, it induces an equivalence relation $\equiv_\Pi$

$$x \equiv_\Pi y \iff \text{there is } \mathcal{A}_i \in \Pi \text{ such that } x, y \in \mathcal{A}_i$$

Conversely, an equivalence relation of a non-empty set $\mathcal{A}$ induces a partition $\Pi = \{\mathcal{A}_x\}_{x \in \mathcal{A}}$, with

$$\mathcal{A}_x = \{y \in \mathcal{A} \,|\, y \equiv x\}.$$

Disjoint Sets

Without loss of generality we will assume that the *support* $\mathcal{A}$ of the disjoint sets is $\{1, \ldots, n\}$ (or $\{0, 1, \ldots, n-1\}$). If that were not the case, we can have a dictionary to map the actual elements of $\mathcal{A}$ into the range $\{1, \ldots, n\}$.
We shall also assume that $\mathcal{A}$ is static, that is, no elements are added or removed. Efficient representations for partitions of a dynamic set can be obtain with some extra but small effort.

Disjoint Sets

Two fundamental operations supported by a DISJOINTSETS abstract data type are:

1. Given i and j, determine if the items i and j belong to the same block (class), or not. Alternatively, given an item i find the representative of the block (class) to which i belongs; i and j belong to the same block
   $\iff \texttt{Find}(i) = \texttt{Find}(j)$
2. Given i and j, perform the union (a.k.a. merge) of the blocks of i and j into a single block; the operation might require i and j to be the representatives of their respective blocks

It is because of these two operations that these data structures are usually called union-find sets or merge-find sets (mfsets, for short).

Union-Find

```
class UnionFind {
public:
// Creates the partition {{0}, {1}, ..., {n-1}}
UnionFind(int n);

// Returns the representative of the class to which
// i belongs (should be const, but it is not to
// allow path compression)
int Find(int i);

// Performs the union of the classes with representatives
// ri and rj, ri ≠ rj
void Union(int ri, int rj);

// Returns the number of blocks in the union-find set
int nr_blocks() const;
...
};
```

Part IV

Disjoint Sets

12 Disjoint Sets: Introduction

13 Implementation of Union-Find

14 Analysis of Union-Find

Implementation #1: Quick-find

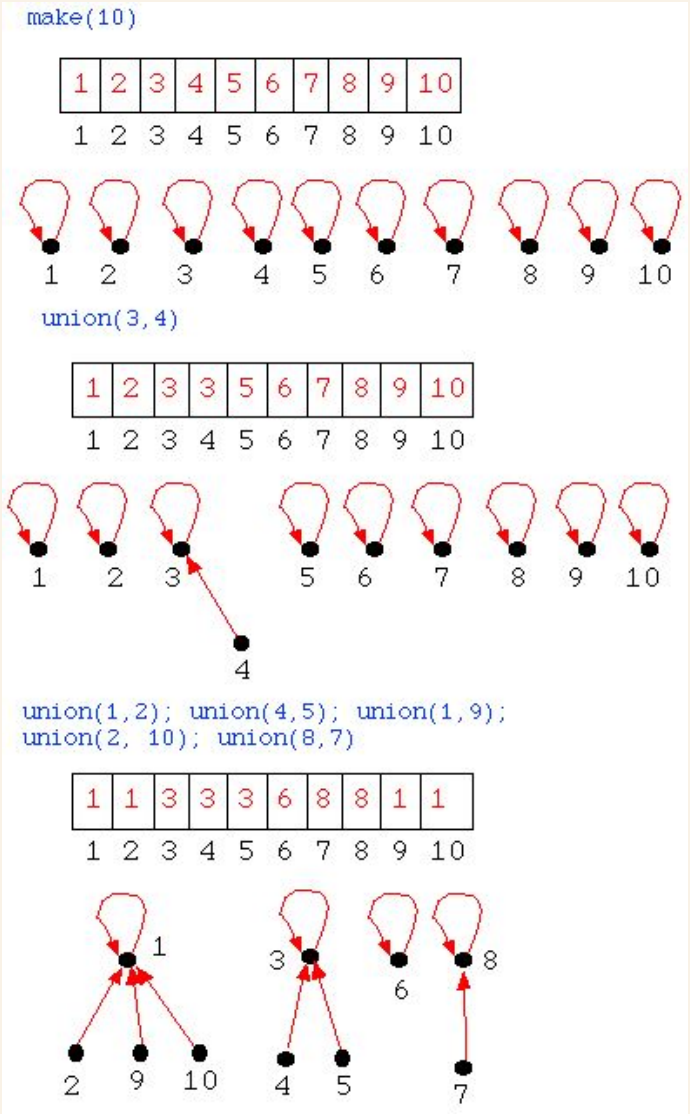
- We represent the partition with a vector P:
  $P[i] =$ the representive of the block of i
- Initially, $P[i] = i$ for all i, $1 \leq i \leq n$
- FIND(i) is trivial: just return $P[i]$
- For the union of two blocks with representatives ri and rj, simply scan the vector P and set all elements in the block of ri now belong to the block of rj, that is, set $P[k] := rj$ whenever $P[k] = ri$ (or vice-versa, transfer elements in the block rj to block ri)
- FIND(i) is very cheap ($\Theta(1)$), but UNION(ri, rj) is very expensive ($\Theta(n)$).

Implementation #1: Quick-find

We can avoid scanning the entire array to perform a union; but we will still have to change the representative of all the elements in one block to point to the representative of the other block, and this has linear cost in the worst-case too.

Despite it is not very natural in this case, it is very convenient to think of the union-find set as a collection of trees, one tree per block, and see $P[i]$ as a pointer to the parent of i in its tree; $P[i] = i$ indicates that i is the root of the tree—i is the representative of the block. With *quick-find*, all trees have height 1 (blocks with a single item) or 2 (the representative is the root and all other items in the block are its children).

Implementation #1: Quick-find



Implementation #2: Quick-union

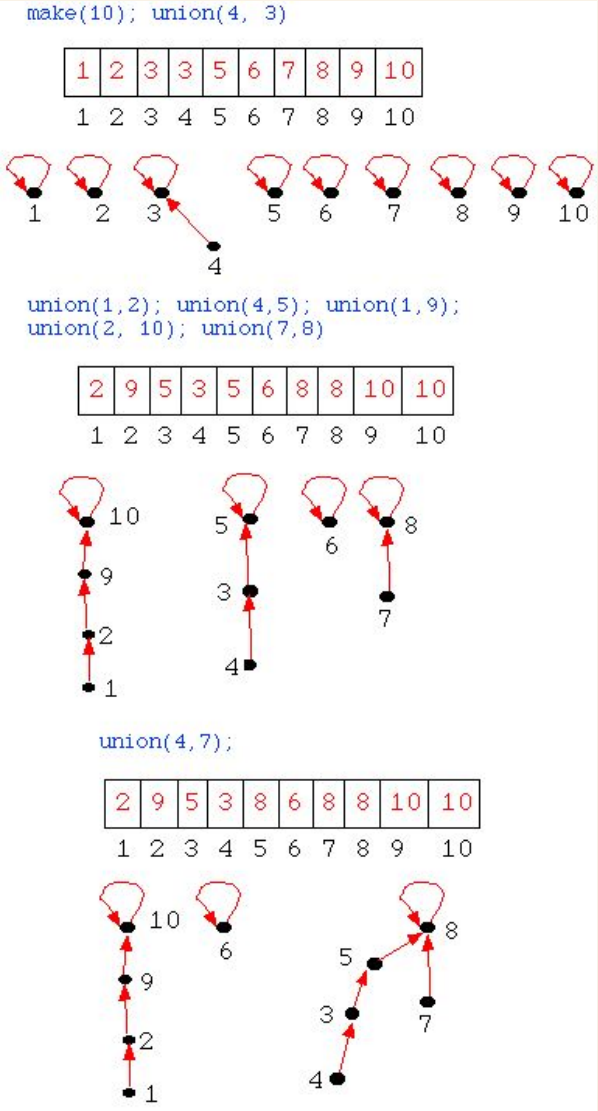
In quick-union, to merge two blocks with representatives ri and rj, it is enough to set $P[ri] := rj$ or $P[rj] := ri$. That makes UNION(ri, rj) trivial and cheap (cost is $\Theta(1)$).
If we allow UNION(i, j) with whatever i and j, we must find the corresponding representatives ri and rj, check that they are different and proceed as above. The operation can now be costly, but that's because of the calls to FIND.

A call FIND(i) can be expensive in the worst-case, it is proportional to the maximum height of the tree that contains i, and that can be as much as $\Theta(n)$.

Implementation #2: Quick-union

```
class UnionFind {
  ...
private:
  vector<int> P;
  int nr_blocks;
};

UnionFind::UnionFind(int n) : P(vector<int>(n)) {
  // constructor
  for (int j = 0; j < n; ++j)
     P[j] = j;
  nr_blocks = n;
}
void UnionFind::Union(int i, int j) {
  int ri = Find(i); int rj = Find(j);
  if (ri != rj) {
    P[ri] = rj; --nr_blocks;
  }
}
int UnionFind::Find(int i) {
  while (P[i] != i) i = P[i];
  return i;
}
```

Implementation #2: Quick-union



Implementation #3: Union by weight or by rank

To overcome the problem of unbalanced trees (leading to trees which are too high) it is enough to make sure that in a union

1. The smaller tree becomes the child of the bigger tree (union-by-weight), or
2. The tree with smaller rank becomes the child of the tree with larger rank (union-by-rank)

Unless we use path compression (stay tuned!) *rank*$\equiv$*height*.

Implementation #3: Union by weight or by rank

- To implement one of these two strategies we will need to know, for each block, its size (number of elements) or its rank (=height).
- We can use an auxiliary array to store that information. But we can avoid the extra space as follows: if i is the representative of its block, instead of setting $P[i] := i$ to mark it as the root we can have
  1. $P[i] = -$the size of the tree rooted at i
  2. $P[i] = -$the rank of the tree rooted at i

  We use the negative sign to indicate that i is the root of a tree.

Implementation #3: Union by weight or by rank

```
class UnionFind {
  ...
private:
  vector<int> P;
  int nr_blocks;
};

UnionFind::UnionFind(int n) : P(vector<int>(n)) {
  // constructor
  for (int j = 0; j < n; ++j)
     P[j] = -1; // all items are roots of trees of size 1 (or rank 1)
  nr_blocks = n;
}
int UnionFind::Find(int i) {
  // P[i] < 0 when i is a root
  while (P[i] > 0) i = P[i];
  return i;
}
 ...
```

Implementation #3: Union by weight or by rank

```
void UnionFind::Union(int i, int j) {
  int ri = Find(i); int rj = Find(j);
  if (ri != rj) {
    if (P[ri] >= P[rj]) {
    // ri is the smallest/shortest
       P[ri] = rj;
       P[rj] += P[ri]; // <= union-by-weight
       // P[rj] = min(P[rj], P[ri]-1); // <= union-by-rank
    } else {
    // rj is the smallest/shortest
       P[rj] = ri;
       P[ri] += P[rj]; // <= union-by-weight
       // P[ri] = min(P[ri], P[rj]-1); // <= union-by-rank
    }
    --nr_blocks;
  }
}
```

Implementation #3: Union by weight or by rank

Lemma

The height of a tree that represents a block of size k is $\leq 1 + \log_2 k$, using union-by-weight.

Proof

We prove it by induction. If $k = 1$ the lemma is obviously true, the height of a tree of one element is 1. Let T be a tree of size k resulting from the union-by-weight of two trees T_1 and T_2 of sizes r and s, respectively, assume $r \leq s < k = r + s$. Then T has been obtained putting T_1 as child of T_2.

Implementation #3: Union by weight or by rank

Proof (cont'd)

By inductive hypothesis, height$(T_1) \leq 1 + \log_2 r$ and height$(T_2) \leq 1 + \log_2 s$. The height of T is that of T_2 unless height$(T_1) =$ height(T_2), then height$(T) =$ height$(T_1) + 1$. That is,

$$\begin{aligned}
\text{height}(T) &= \max(\text{height}(T_2), \text{height}(T_1) + 1) \\
&\leq 1 + \max(\log_2 s, 1 + \log_2 r) \\
&= 1 + \max(\log_2 s, \log_2(2r)) \\
&\leq 1 + \log_2 k,
\end{aligned}$$

since $s \leq k$ and $2r \leq r + s = k$. □

Implementation #3: Union by weight or by rank

An analogous lemma can be proved if we perform union by rank.

We might be satisfied with union-by-rank or union-by-weight, but we can improve even further the cost of FIND applying some path compression heuristic.

Path Compression

While we look for the representative of i in a FIND(i), we follow the pointers from i up to the root, and we could make the pointers along that path change so that the path becomes shorter and therefore we may speeed up future calls to FIND. There are several heuristics for path compression:

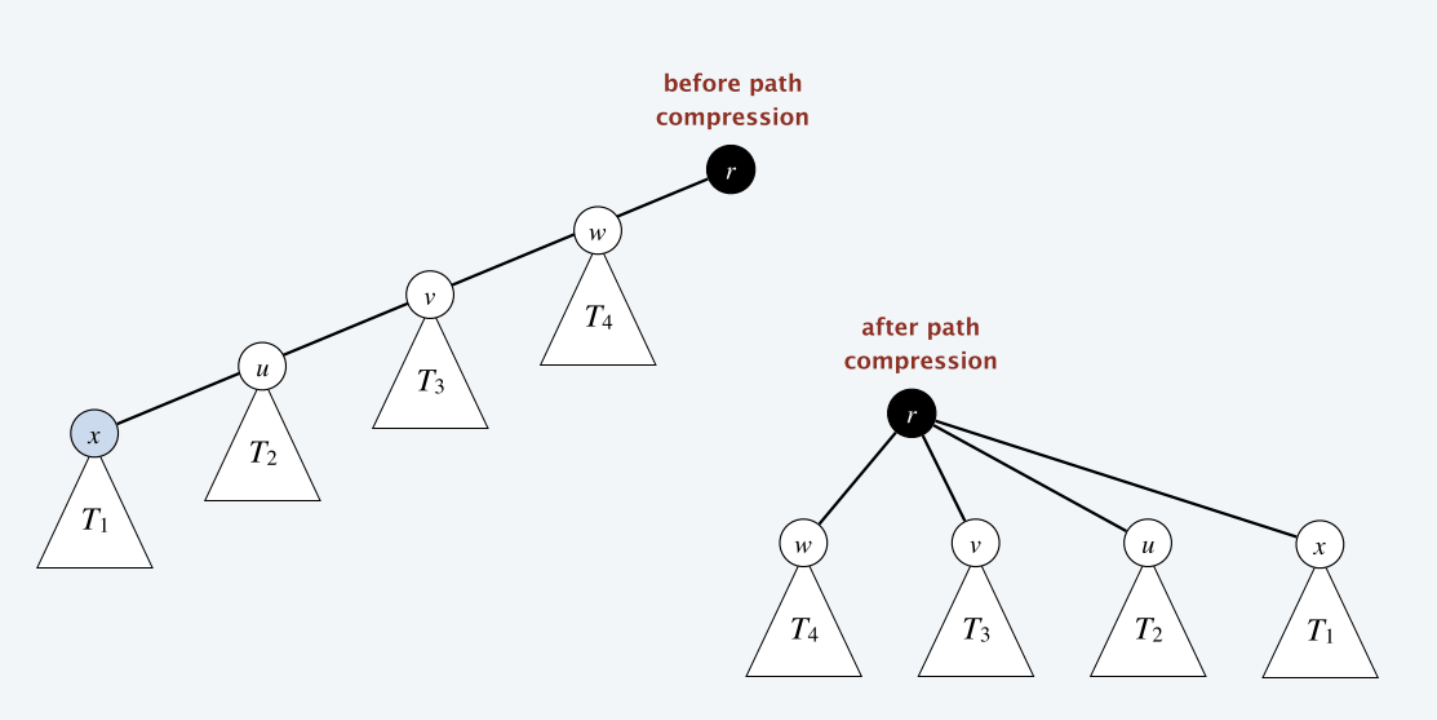
1. In full path compression, we traverse the path from i to its representative twice: first to determine that ri is such representative; second, to set $P[k] := ri$ for all k along the path, as all these items have ri as their representative; this only doubles the cost of FIND(i).
2. In path splitting, we maintain two consecutive items in the path $i1$ and $i2 = P[i1]$, then when we go up in the tree we make $P[i1] := P[i2]$; at the end of this traversal, all k along the path, except the root and its immediate child, will point to the element that was previously their grand-parent; we reduce the length of the path roughly by half.

Path Compression

3. In path halving, we traverse the path from i to its representative, making every other node point to its grand-parent.

Path compression: full path compression

Make every node point to its representative.

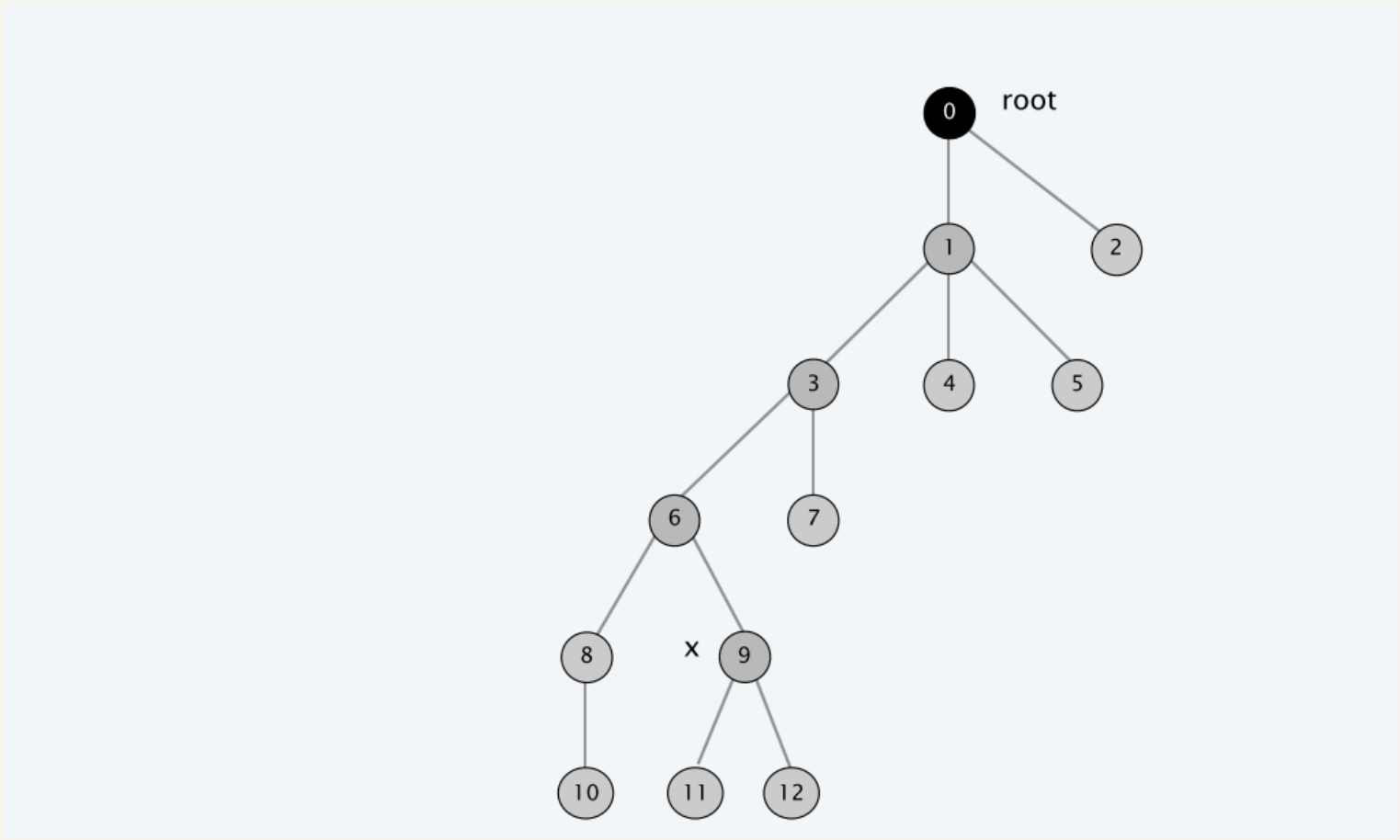


Source: Kevin Wayne
(https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/UnionFind.pdf)

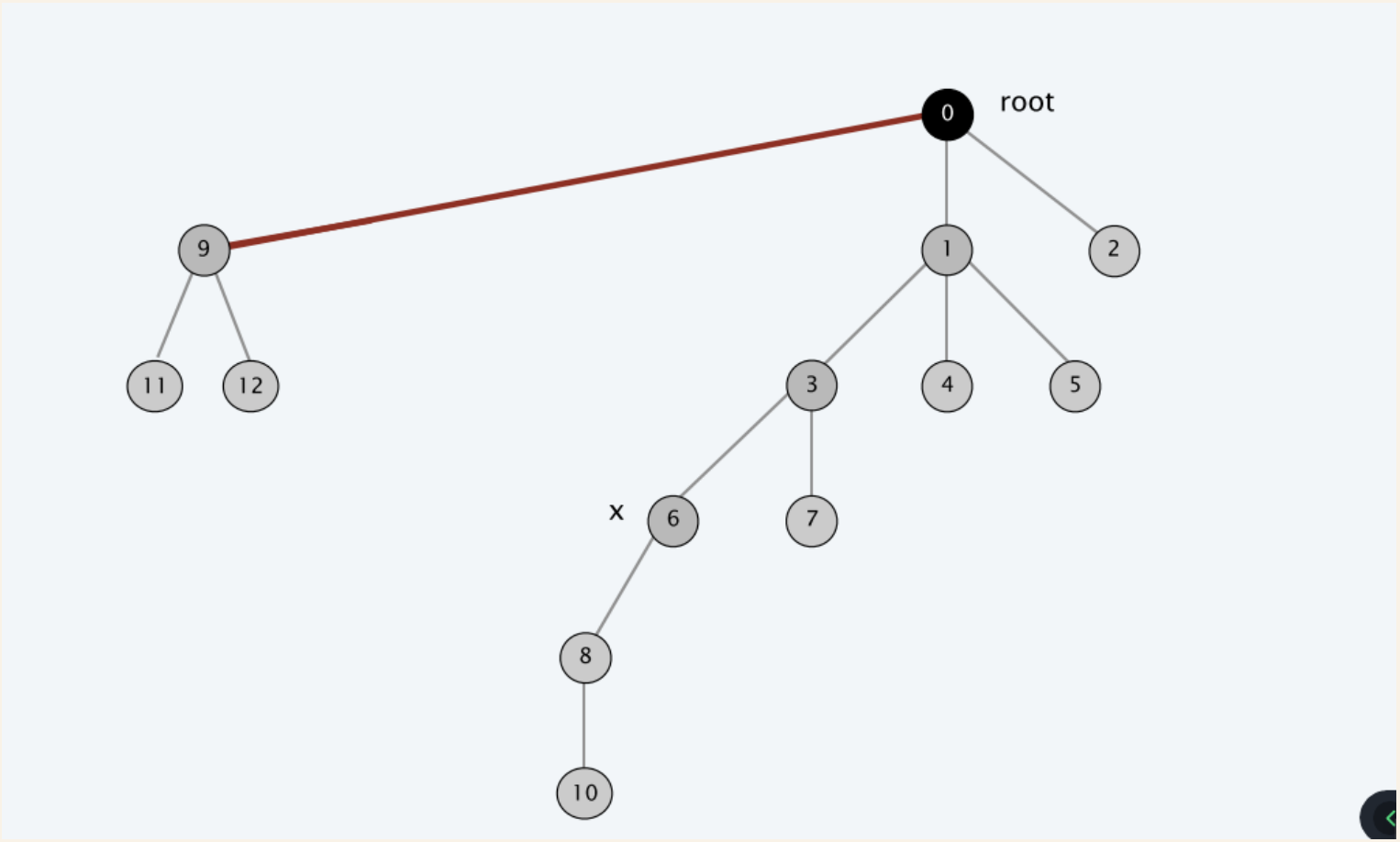
Path compression: full path compression

```
    // iterative full path compression
    // with the convention that P[i] = -the rank of i if P[i] < 0
int UnionFind::Find(int i) {
   int ri = i;
   // P[ri] < 0 when ri is a root
   while (P[ri] > 0) ri = P[ri];

   // traverse the path again making everyone point to ri
   while (P[i] > 0) {
      int aux = i;
      i = P[i];
      P[aux] = ri;
   }
   return ri;
}

// recursive full path compression
// with the convention that P[i] = -the rank of i if P[i] < 0
int UnionFind::Find(int i) {
  if (P[i] < 0) return i;
  else {
    P[i] = Find(P[i]);
    return P[i];
  }
}
```

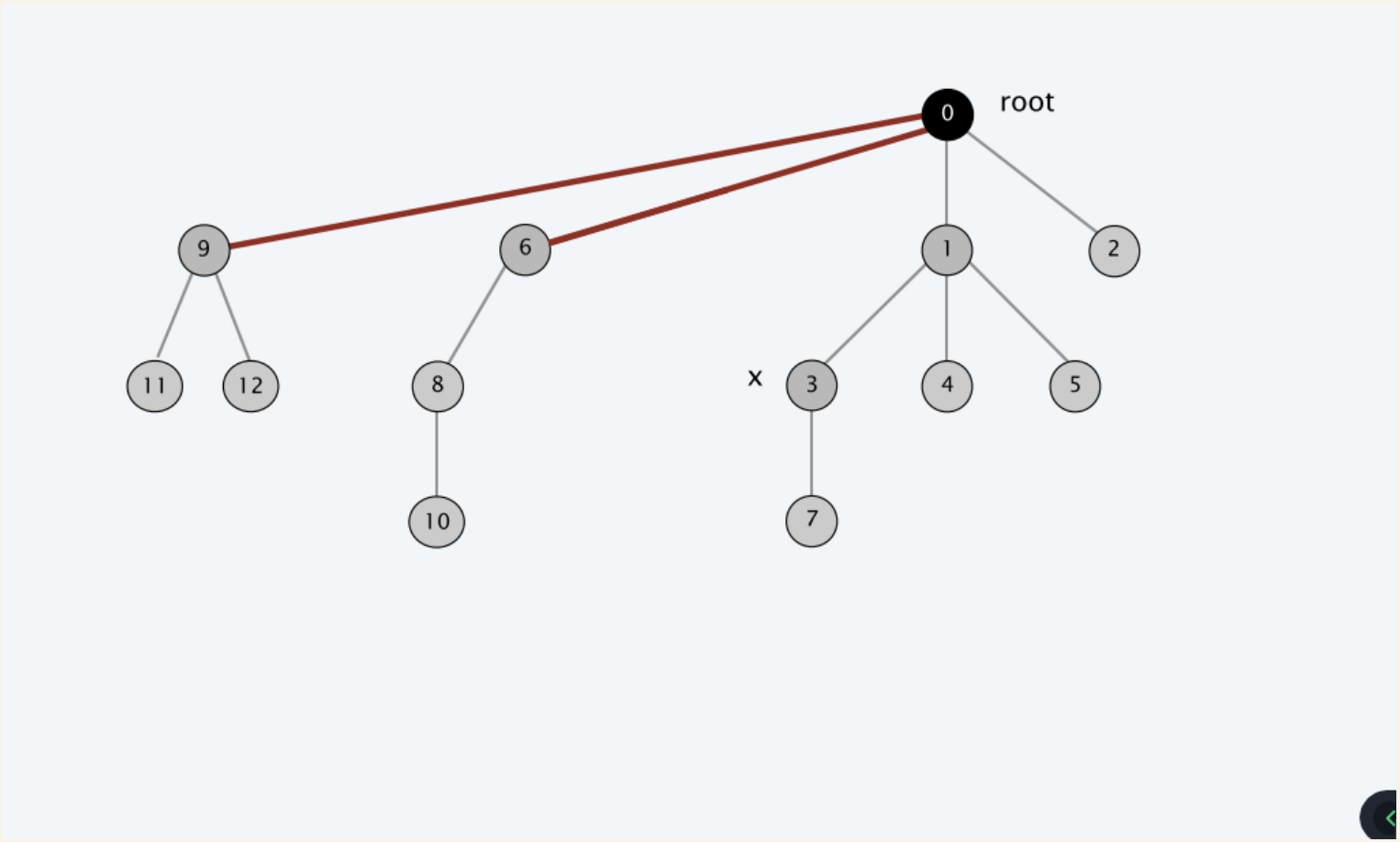
Path compression: full path compression



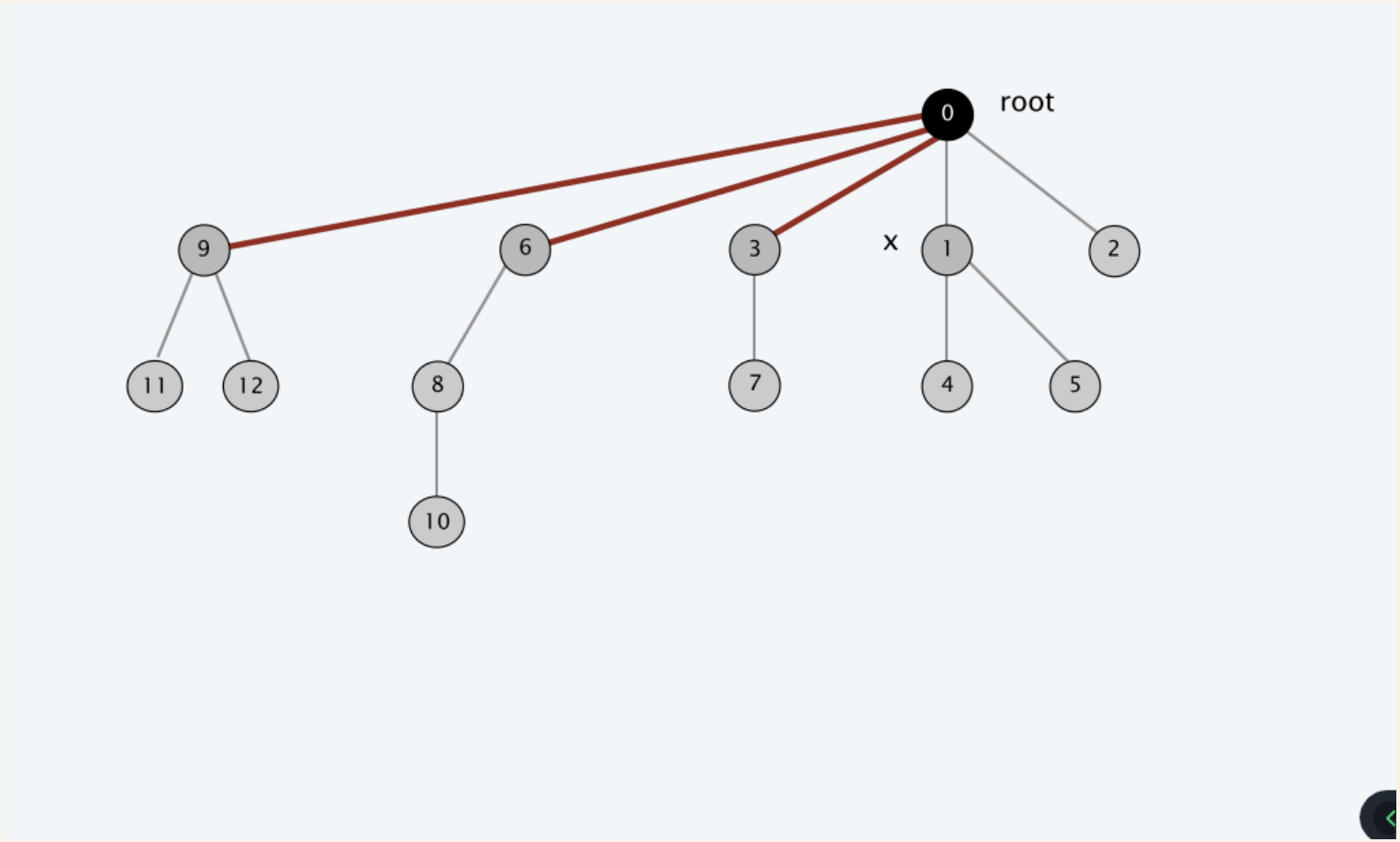
Path compression: full path compression



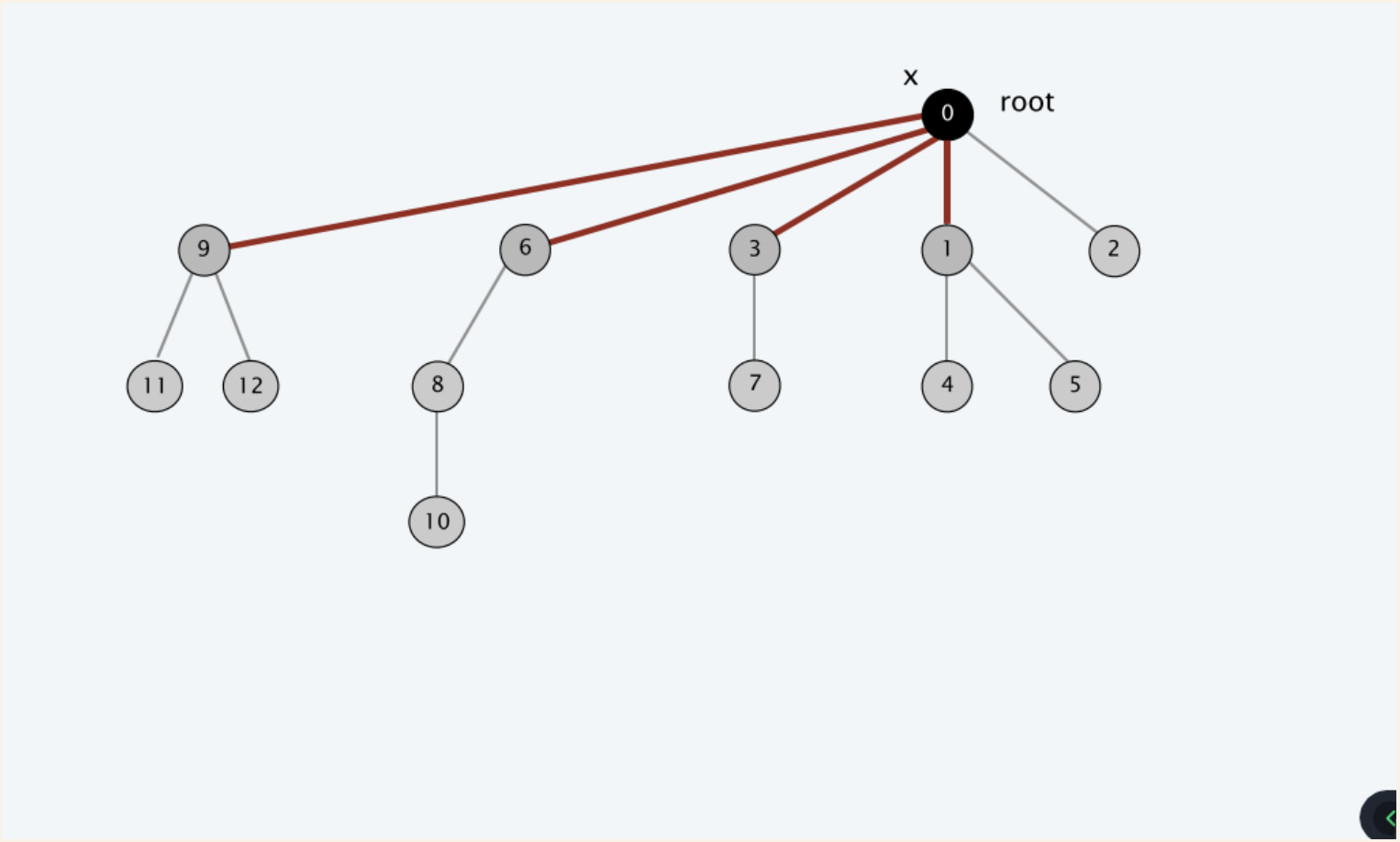
Path compression: full path compression



Path compression: full path compression

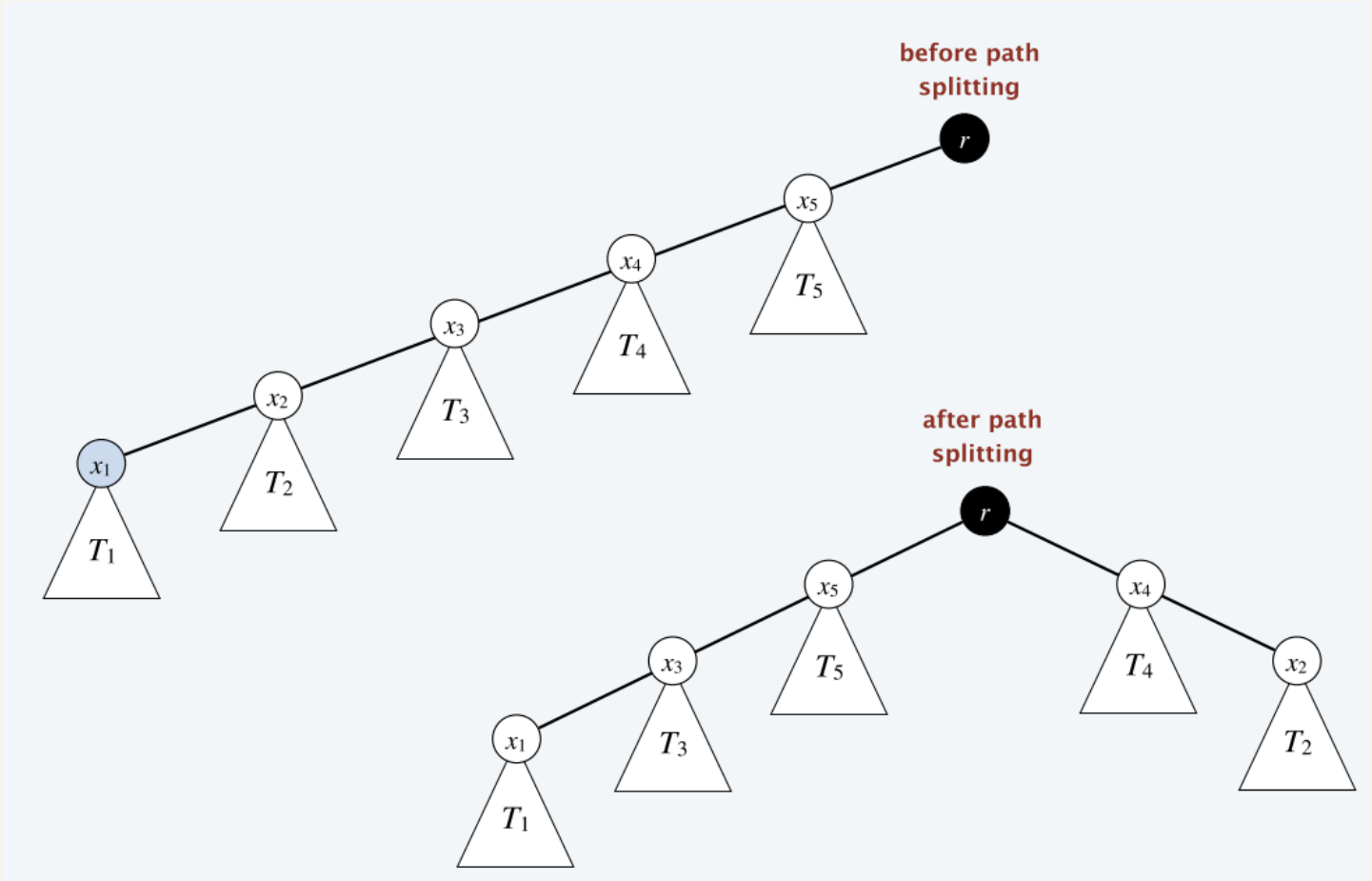


Path compression: full path compression



Path compression: path splitting

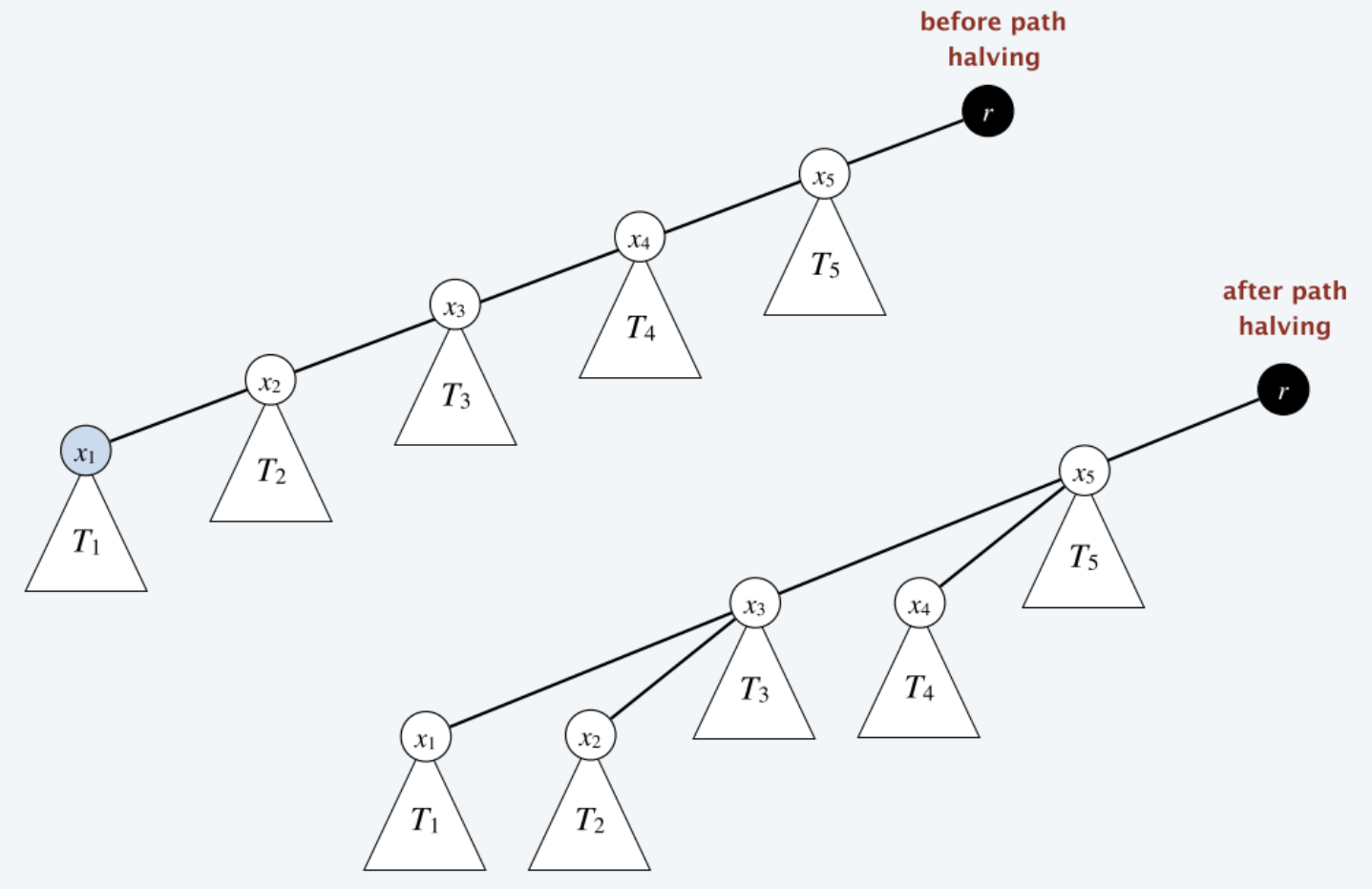
Make every node point to its grandparent (except if it is the root or a child of the root).



Source: Kevin Wayne
(https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/UnionFind.pdf)

Path compression: path halving

Make every other node in the path point to its grandparent (except if it is the root or a child of the root).



Source: Kevin Wayne
(https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/UnionFind.pdf)

Part IV

Disjoint Sets

12 Disjoint Sets: Introduction

13 Implementation of Union-Find

14 Analysis of Union-Find

Amortized analysis of Union-Find

The analysis of Union-Find with union by weight (or by rank) using some path compression heuristic must be amortized: the union of two representatives (roots) is always cheap, and the cost of any FIND is bounded by $\mathcal{O}(\log n)$, but if we apply many FIND's the trees become bushier, and we approach rather quickly the situation of *Quick-Find* while we avoid costly UNION's.

In what follows we will analyze the cost of a sequence of m intermixed UNIONs and FINDs performed in a Union-Find data structure with $n \leq m$ elements, using union-by-rank and full path compression.

Similar results hold for the various combinations of union-by-weight/union-by-rank and path compression heuristics.

Amortized analysis of Union-Find

1. Observation #1. Path compression does not change the rank of any node, hence $\text{rank}(x) \geq \text{height}(x)$ for any node x.
2. In what follows we assume that we initialize the rank of all nodes to 0 and keep the ranks in a different array, so we will have:

```
int UnionFind::Find(int i) {
  if (P[i] == i) return i;
  else {
    P[i] = Find(P[i]);
    return P[i];
  }
}

void UnionFind::Union(int i, int j) {
  int ri = Find(i); int rj = Find(j);
  if (ri != rj) {
    if (rank[ri] <= rank[rj]) {
      P[ri] = rj;
      rank[rj] = max(rank[rj], 1+rank[ri]);
    } else {
      P[rj] = ri;
      rank[ri] = max(rank[ri], 1+rank[rj]);
    }
  }
}
```

Amortized analysis of Union-Find

Proposition

The tree roots, node ranks and elements within a tree are the same with or without path compression.

Proof

Path compression only changes some parent pointers, nothing else. It does not create new roots, does not change ranks or move elements from one tree to another. □

Amortized analysis of Union-Find

Properties:

1. If x is not a root node then rank(x) $<$ rank(parent(x)).
2. If x is not a root node then its rank will not change.
3. Let $r_t =$ rank(parent(x)) at time t. If at time $t+1$ x changes its parent then $r_t < r_{t+1}$
4. A root node of rank k has $\geq 2^k$ descendants.
5. The rank of any node is $\leq \lceil \log_2 n \rceil$
6. For any $r \geq 0$, the Union-Find data structure contains at most $n/2^r$ elemnts of rank r

Amortized analysis of Union-Find

All the six properties hold for Union-Find with union-by-rank. By the previous proposition, propeties #2, #4, #5 and #6 immediately hold for any variant using path compression

Only properties #1 and #3 might not hold as path compression makes changes to parent pointers. However, they still hold if we are doing path compression.

Amortized analysis of Union-Find

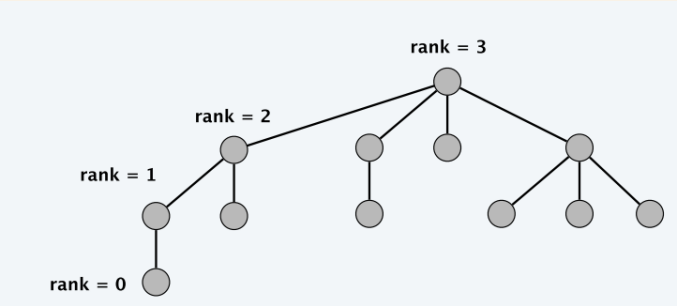
Proof of Property #1

A node of rank k can only be created by joining two nodes of rank $k - 1$. Path compression can't change ranks. However it might change the parent of x; in that case, x will point to some ancestor of its previous parent, hence $\text{rank}(x) < \text{rank}(\text{parent}(x))$ at all times. □

Proof of Property #2

The rank of a node can only change in union-by-rank if x was a root and becomes a non-root. Once a root becomes a non-root it will never become a root node again. Path compression never changes ranks and never changes roots. □

Amortized analysis of Union-Find



Example of property #1

Amortized analysis of Union-Find

Proof of Property #3

When the parent of x changes it is because either

1. x becomes a non-root and union-by-rank guarantees that $r_t = \text{rank}(\text{parent}(x)) = \text{rank}(x)$ and $r_{t+1} > r_t$ as x becomes a chlid of a node whose rank is larger than r_t
2. x is a non-root at time t but path compression changes its parent. Because x will be pointing to some ancestor of its parent then $r_{t+1} > r_t$ (because of Property #1)

Amortized analysis of Union-Find

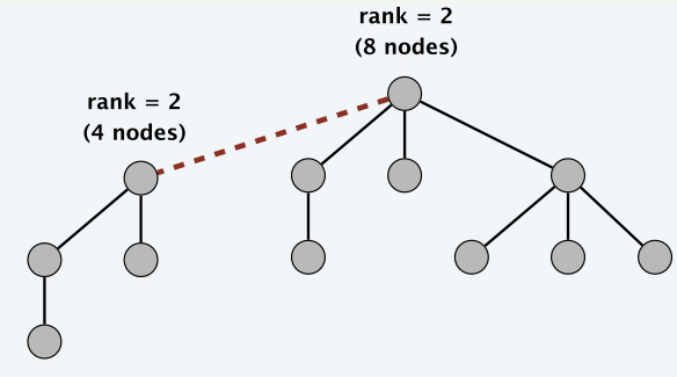
Proof of Property #4

By induction on k.
Base: If $k = 0$ then x is the root of a tree with only one node, so the number of descendants is $\geq 2^k$.
Inductive hypothesis: a node x of rank k can only get that rank because of the union of two nodes of rank $k - 1$, hence x was the root of one of the trees involved and its rank was $k - 1$ before the union. By hypothesis, each tree contained $\geq 2^{k-1}$ and the result must then contain $\geq 2^k$ nodes. □

Proof of Property #5

Immediate from Properties #1 and #4. □

Amortized analysis of Union-Find



An example of property #4

Amortized analysis of Union-Find

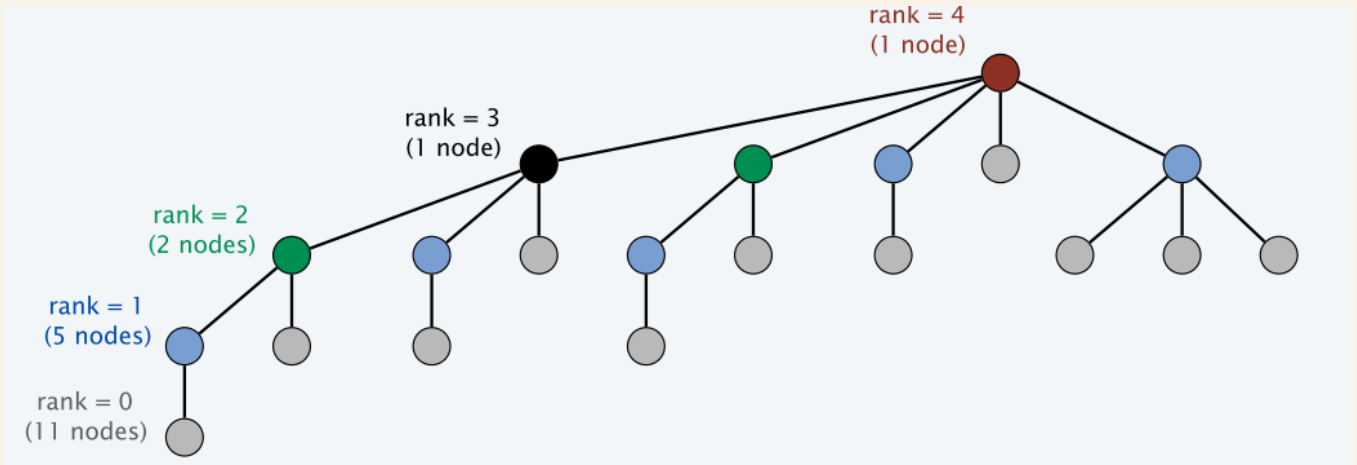
Proof of Property #6

Because of Property #4, any node x of rank k is the root of a subtree with $\geq \quad 2^k$ nodes. Indeed, if x is a root that is the statement of the property. Else, inductively, because x had the property just before becoming a non-root; since neither its rank nor the set of descendants can change afterwards, the property is also true for non-root nodes. Because of Property #1, two distinct nodes of rank k can't be one ancestor of the other and then they can't have common ancestors.

Therefore, there can be at most $n/2^r$ nodes of rank r.

□

Amortized analysis of Union-Find



An example of property #6

Amortized analysis of Union-Find

Definition

The iterated logarithm function is

$$\lg^* x = \begin{cases} 0, & \text{if } x \leq 1 \\ 1 + \lg^*(\lg x), & \text{otherwise} \end{cases}$$

We consider only logarithms base 2, hence write $\lg \equiv \log_2$.

n	$\lg^* n$
$(0, 1]$	0
$(1, 2]$	1
$(2, 4]$	2
$(4, 16]$	3
$(16, 65536]$	4
$(65536, 2^{65536}]$	5

$\lg^* n \leq 5$ in this Universe.

Amortized analysis of Union-Find

Given k, let $2 \uparrow\uparrow k = \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{k \text{ exponentiations}}$. Inductively: $2 \uparrow\uparrow 0 = 1$, and $2 \uparrow\uparrow k = 2^{2 \uparrow\uparrow (k-1)}$. Then $\lg^*(2 \uparrow\uparrow k) = k$. Define groups

$$
\begin{aligned}
G_0 &= \{1\} \\
G_1 &= \{2\} \\
G_2 &= \{3, 4\} \\
G_3 &= \{5, \ldots, 16\} \\
G_4 &= \{17, \ldots, 65536\} \\
G_5 &= \{65537, \ldots, 2^{65536}\} \\
\ldots &= \ldots \\
G_k &= \{1 + 2 \uparrow\uparrow (k-1), \ldots, 2 \uparrow\uparrow k\}
\end{aligned}
$$

Amortized analysis of Union-Find

For any $n > 0$, n belongs to $G_{\lg^* n}$. The rank of any node in a Union-Find data structure of n elements will belong to one of the first $\lg^* n$ groups (as all ranks are between 0 and $\lg n$).

Amortized analysis of Union-Find

Accounting scheme: We assign credits during a UNION to the node that ceases to be a root; if its rank belongs to group G_k we assign $2 \uparrow\uparrow k$ credits to the item.

Proposition

The number of credits assigned in total among all nodes is $\leq n \lg^* n$.

Proof

By Property #6, the number of nodes with rank $\geq x+1$ is at most

$$\frac{n}{2^{x+1}} + \frac{n}{2^{x+2}} + \ldots \leq \frac{n}{2^x}$$

Consider nodes that belong (their ranks) to group $G_k = \{x+1, \ldots, 2^x\}$ $(x = 2 \uparrow\uparrow (k-1))$.

Amortized analysis of Union-Find

Proof (cont'd)

As the group contains $\leq 2^x$ nodes the number of credits assigned to nodes in the group is $\leq n$. All the ranks belong in the first $\lg^* n$ groups, hence the total number of credits is $\leq n \lg^* n$. $\square$

Amortized analysis of Union-Find

The cost of Union is constant. We need to find the amortized cost of Find. The actual cost is the number of parent pointers followed:

1. parent(x) is a root $\Longrightarrow$ this is true for at most one of the nodes visited during the execution of a FIND,
2. rank(parent(x)) belongs to a group G_j higher that rank(x) $\Longrightarrow$ this might happen at most for $\lg^* n$ visited x's during the execution of a FIND
3. rank(parent(x)) and rank(x) belong to the same group $\Longrightarrow$ see next slide

Amortized analysis of Union-Find

We make any node x such that rank(parent(x)) and rank(x) are in the same group to pay 1 credit to follow and update the parent pointer during the FIND.

- Then rank(parent(x)) strictly increases (Property #1).
- If the node was in group $G = \{x + 1, \ldots, 2^x\}$ then it had 2^x credits to spend and the rank of its parent will belong to a higher group before x has been been updated 2^x times by FIND operations.
- Once the parent's rank belongs to a higher group than the rank of x, the situation will remain, as rank(x) (hence the group) never changes and rank(parent(x)) never decreases.

Therefore x has enough credits to pay for all FIND's in which it gets involved before it becomes a node in Case #2.

Amortized analysis of Union-Find

Theorem

Starting from an initial Union-Find data structures for n elements with n disjoint blocks, any sequence of $m \geq n$ UNION *and* FIND *using union-by-rank and full path compression have total cost $\mathcal{O}(m \lg^* n)$.*

Proof

The amortized cost of FIND is $\mathcal{O}(\lg^* n)$, and that of any UNION is constant, hence the sequence of m operations has total cost $\mathcal{O}(m \lg^* n)$. □

Amortized analysis of Union-Find

1972: Fischer: $\mathcal{O}(m \log \log n)$

1973: Hopcroft & Ullman: $\mathcal{O}(m \lg^* n)$

1975: Tarjan: $\mathcal{O}(m\,\alpha(m,n))$. Ackermann's inverse $\alpha(m,n)$ is an extremely slowly growing function $\alpha(m,n) \ll \lg^* n$

1984: Tarjan & van Leeuwen: $\mathcal{O}(m\,\alpha(m,n))$. For all combinations of union-by-weight/rank and path compression heuristics (full/splitting/halving).

1989: Fredman & Saks: $\Omega(m\,\alpha(m,n))$. A non-trivial lower bound for amortized complexity of Union-Find in the cell probe model.

Disjoint Sets

To learn more:

[1] Michael J. Fischer
Efficiency of Equivalence Algorithms
Symposium on Complexity of Computer Computations,
IBM Thomas J. Watson Research Center, 1972.

[2] J.E. Hopcroft and J.D. Ullman
Set Merging Algorithms
SIAM J. Computing 2(4):294–303, 1973.

[3] Robert E. Tarjan
Efficiency of a Good But Not Linear Set Union Algorithm
J. ACM 22(2):215–225, 1975.

Disjoint Sets

To learn more:

[1] Robert E. Tarjan and Jan van Leeuwen
Worst-Case Analysis of Set Union Algorithms
J. ACM 31(2):245–281, 1984.

[2] Michael L. Fredman and Michael E. Saks
The Cell Probe Complexity of Dynamic Data Structures
Proc. 21st Symp. Theory of Computing (STOC), p. 345–354, 1989.

[3] Z. Galil and G. Italiano
Data Structures and Algorithms for Disjoint Set Union Problems
ACM Computing Surveys 23(3):319–344, 1991.

Part V

Data Structures for String Processing

15 Tries

16 Suffix Trees

Tries

We often deal with keys which are sequences of symbols (characters, decimal digits, bits, ...). Such a decomposition is usually very natural and can be exploited to implement efficient dictionaries.
Moreover, we usually want, besides lookups and updates, operations in which the keys as symbol sequences matter: for example, we might want an operation that, given a collection of words C and some word w, returns all words in C that contain w as a subsequence. Or as a prefix, or a suffix, etc.

Tries

Consider a finite alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_m\}$ with $m \geq 2$ symbols. We denote Σ^*, as usual in the literature, the set of all strings that can be formed with symbols from Σ. Given two strings u and v in Σ^* we write $u \cdot v$ for the string which results from the concatenation of u and v. We will use λ to dente the empty string or string of length 0.

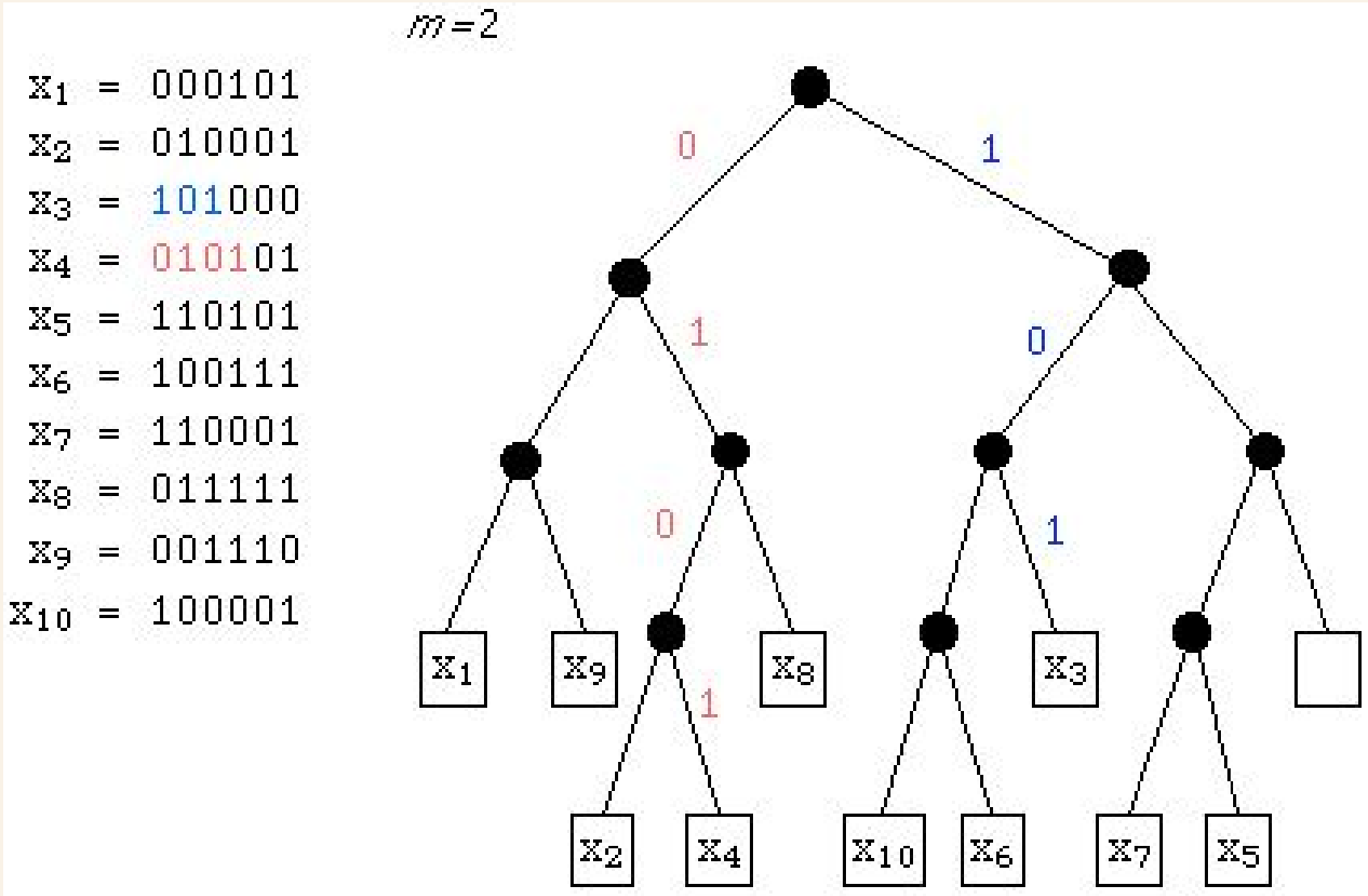
Definition

Given a finite set of strings $X \subset \Sigma^*$, all of identical length, the *trie* T of X is an m-ary tree recursively defined as follows:

1. If X contains a single element x or none, then T is a tree consisting on a single node that contains x or is empty.
2. If $|X| \geq 2$, let T_i be the trie for the subset

$$X_i = \{y \mid x = \sigma_i \cdot y \in X \wedge \sigma_i \in \Sigma\}$$

Tries


m = 2
x1 = 000101
x2 = 010001
x3 = 101000
x4 = 010101
x5 = 110101
x6 = 100111
x7 = 110001
x8 = 011111
x9 = 001110
x10 = 100001
0
1
1
0
0
1
1
x1
x9
x8
x3
x2
x4
x10
x6
x7
x5

Tries

Lemma

If the edges in the trie T for X are labelled in such a way that the edge connecting the root of T with subtree T_i has label σ_i, $1 \leq i \leq m$, then the sequence of labels in the path from the root to a non-empty leaf that contains x form the shortest prefix that univoquely identifies x, that is, the shortest prefix of x which is not shared by any other element of X

Lemma

Let p be the sequence of labels in a path from the root of the trie T to some node v (either internal or leaf); the subtree rooted at v is a trie for the subset of all strings in X starting with the prefix p (and no other strings)

Tries

Lemma

Given a finite set $X \subset \Sigma^*$ of strings of equal length, the trie T for X is unique; in particular, T does not depen on the order in which we "present" or "insert" the elements of X

Lemma

The height of a trie T is the minimum length of prefixes needed to distinguish to elements in X; in other words, the length of the longest prefix which is common to ≥ 2 elements in X; of course, if ℓ is the length of the string in X then

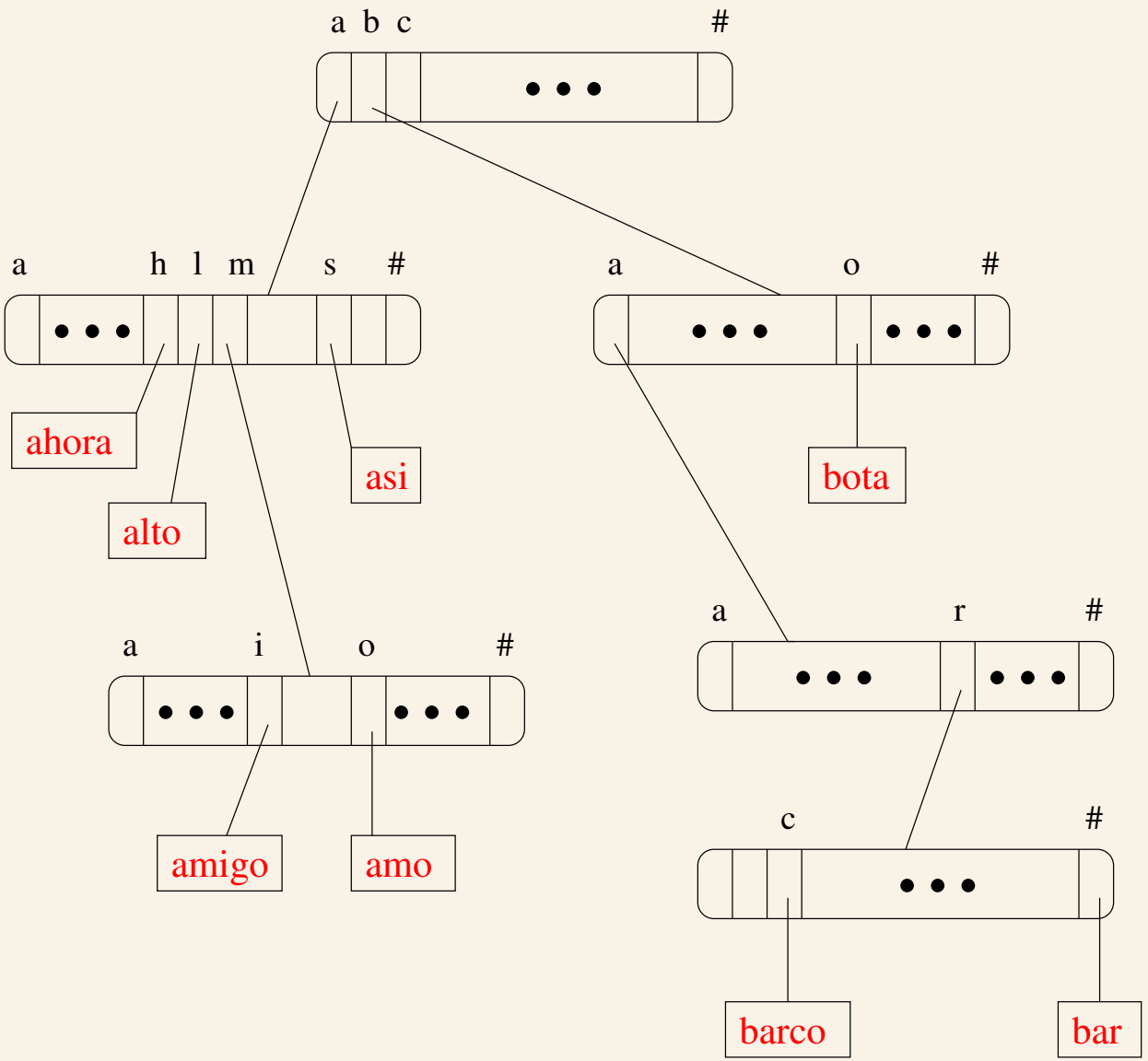
$$height(T) \leq \ell$$

Tries

Our definition of tries requires all string being of the same length; that's too restrictive. What we actually need is that no string in X is a proper prefix of another string in X.
The standard solution to the problem is to extend Σ with a special symbol (e.g. $\sharp$) to mark the end of strings. If we append $\sharp$ to the end of all strings in X, then no (marked) string is a proper prefix of another string. The modest price to pay is to work with an alphabet of $m + 1$ symbols

Tries

X = {ahora, alto, amigo, amo, asi, bar, barco, bota, ...}



Tries

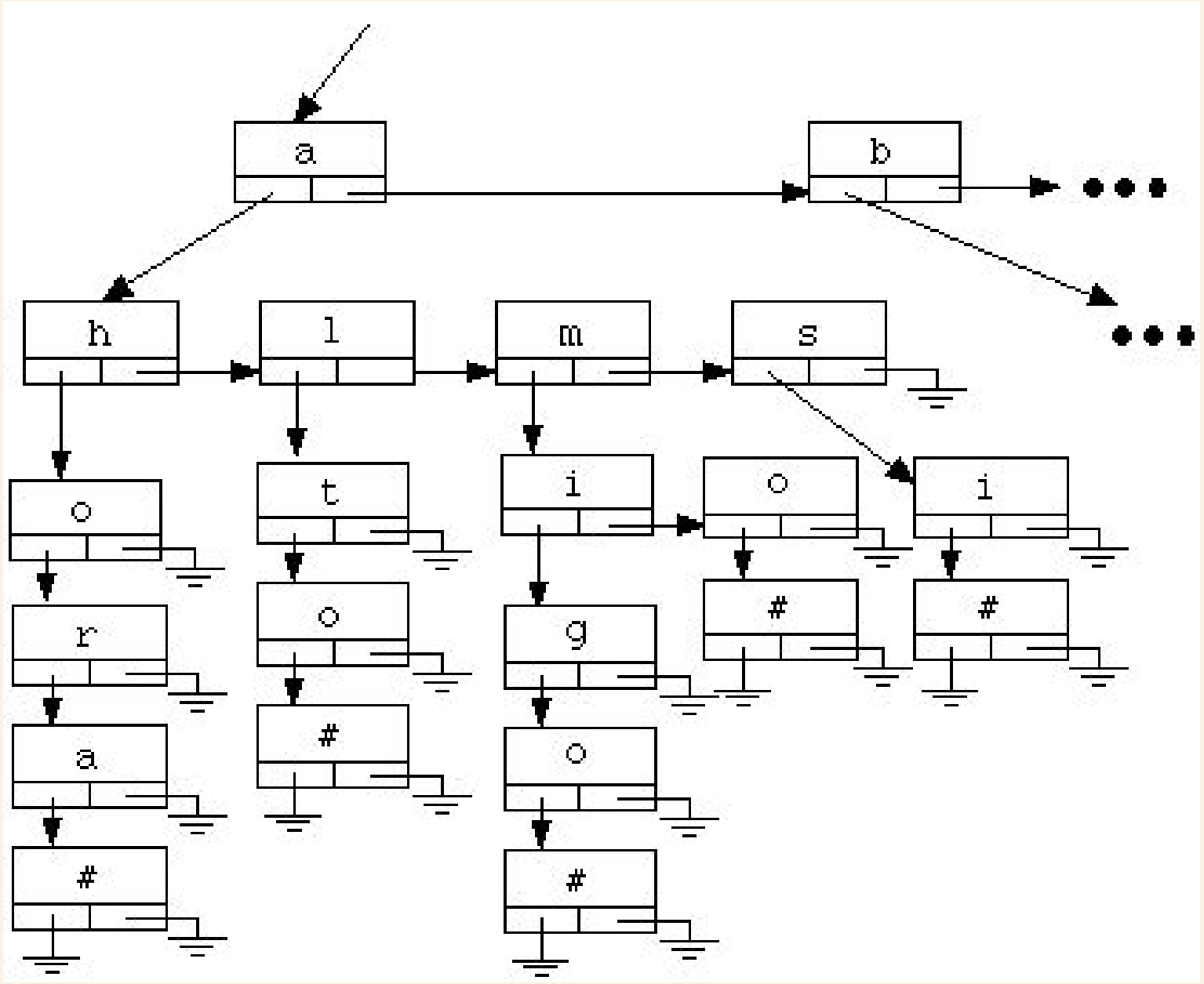
In order to implement tries we can use standard techniques to implement m-ary trees, namely, use an array of pointers for each internal node or use the first child-next sibling representation.

If using arrays of pointers, the pointers give us access to the root node of every non-empty subtree of the current node, and the symbols of Σ can be used to indices into the array of pointers (eventually with some easy bijective mapping $f : \Sigma \to \{0, \ldots, m-1\}$. Leaves that are not empty can contain the remaining suffix of the element, the prefix given by the path from the root to the leave.

Tries

When using first child-next sibling, each node stores a symbol c; the pointer to the first child points to the root of the trie of words that have c at that level, while the next sibling points to the node giving us access to some other trie, now with some other symbol d. It is usual that since the symbols in Σ typically admit a natural total order then the list of children of a node in the trie are incrreasingly sorted according to that order.

Tries



Tries

Despite it is more costly in space to use nodes to store full string, symbol by symbol, instead of storing suffixes once a leaf is reached, it is advantadgeous to avoid different types of nodes, different types of pointers or forcing pointers of one type point to nodes of some other type, using wasteful `union`s,...

```
// We asssume that the class Key supports
// x.length() = length() >= 0 of key x
int Key::length() const;

// x[i] = i-th symbol of key x;
// throws an exception if i < 0 or i >= x.length()
template <typename Symbol>
Symbol Key::operator[](const Key& x, int i);

template <typename Symbol, typename Key, typename Value>
class DigitalDictionary {
public:
   ...
private:
   struct trie_node {
      Symbol _c;
      trie_node* _first_child;
      trie_node* _next_sibl;
      Value _v;
   };
   trie_node* root;
   ...
```

Tries

```
template  <typename Symbol,
          typename Key,
          typename Value>
void DigitalDictionary<Symbol,Key,Value>::lookup(
     const Key& k, bool& exists, Value& v) const {

   trie_node* p = _lookup(root, k, 0);
   if (p == nullptr)
      exists = false;
   else {
      exists = true;
      v = p -> _v;
   }
}
```

Tries

```
// Pre: p points to the root of the subtree that contains
// all elements such that their first i-1 symbols
// coincide with the first i-1 symbols of the key k
// Post: returns a pointer to the node that stores the value
// associated to $k$ if $\pair{k,v}$ belongs to the dictionary
// and a null pointer if not such pair exists
// Cost: O(|k| · m)
template  <typename Symbol, typename Key,
          typename Value>
DigitalDictionary<Symbol,Key,Value>::trie_node*
DigitalDictionary<Symbol,Key,Value>::_lookup(trie_node* p,
     const Key& k, int i) const {

   if (p == nullptr)    return nullptr;
   if (i == k.length()) return p;
   if (p -> _c > k[i])  return nullptr;
   if (p -> _c < k[i])
      return _lookup(p -> _next_sibl, k, i);
   // p -> _c == k[i]
   return _lookup(p -> _first_child, k, i+1);
}
```

Tries

```
template  <typename Symbol,
          typename Key,
          typename Value>
void DigitalDictionary<Symbol,Key,Value>::insert(
     const Key& k, const Value& v) {

   _root = _insert(root, k, 0);
}
```

Tries

```
// Pre: p points to the root of the subtree that contains
// all elements such that their first i-1 symbols
// coincide with the first i-1 symbols of the key k
// Post: returns a pointer to the root of the tree resulting from
// the insertion of the pair $\pair{k[i..],v}$ in the subtree
// Cost: O(|k| · m)
template  <typename Symbol, typename Key,
           typename Value>
DigitalDictionary<Symbol,Key,Value>::trie_node*
DigitalDictionary<Symbol,Key,Value>::_insert(trie_node* p,
     const Key& k, int i) const {

   if (i == k.length()) {
     if (p == nullptr) p = new trie_node;
     p -> _c = Symbol(); // Symbol() is the end-of-string symbol
                 // e.g. Symbol() == '\0' or Symbol() == '\sharp'
     p -> _v = v;
     return p;
   }
   if (p == nullptr or p -> _c > k[i])    {
     trie_node* p = new trie_node;
     p -> _next_sibl = p;
     p -> _c = k[i];
     p -> _first_child = _insert(nullptr, k, i+1);
     return p;
   }
   if (p -> _c < k[i])
      p -> _next_sibl = _insert(p -> _next_sibl, k, i);
   else // p -> _c == k[i]
      p -> _first_child = _insert(p -> _first_child, k, i+1);
   return p;
}
```

Ternary Search Trees

One alternative to implement tries is to represent the trie nodes as binary search trees with pointers to roots of subtrees, instead of as array of pointers to roots, or as linked lists of pointers to roots (of non-empty subtrees).

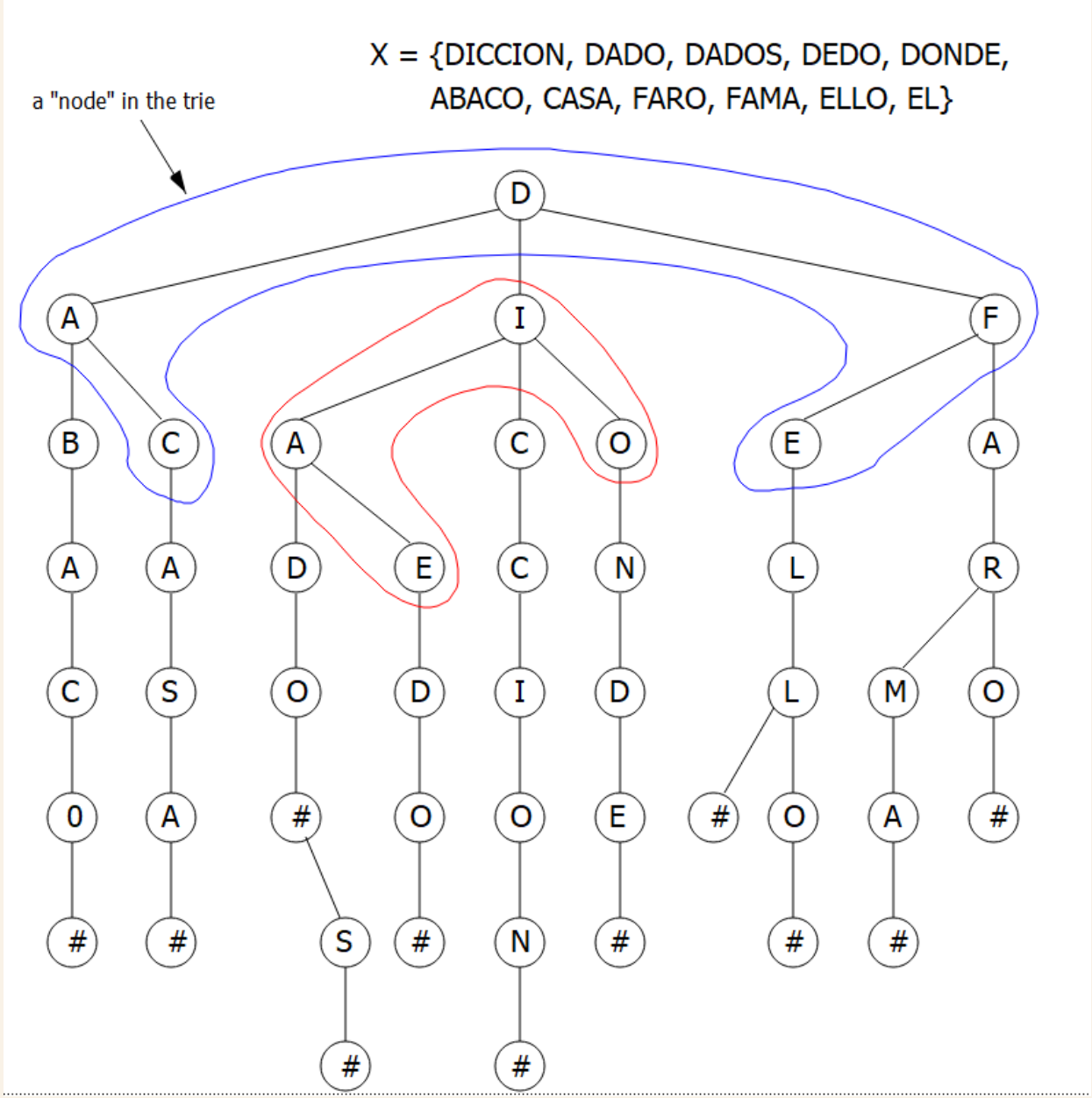
The new data structure, invented by Bentley and Sedgewick (1997), is called an ternary search tree (TST). It tries to combine the efficiency in space of list-tries (we avoid the large number of null pointers when using arrays) and the efficiency in time (we avoid the linear "scans" when using lists to navigate to the appropriate subtree).

Nodes in TSTs have a symbol c and 3 pointers each: pointers to the left and right child of the node in the BST that represents the trie "node", and a central pointer to the root of the subtree with symbol c at that level.

Ternary Search Trees

```
template <typename Symbol,
          typename Key,
          typename Value>
class DigitalDictionary {
public:
   ...
   void lookup(const Key& k, bool& exists, Value& v) const;
   void insert(const Key& k, const Value& v);
   ...
private:
   struct tst_node {
      Symbol _c;
      tst_node* _left;
      tst_node* _cen;
      tst_node* _right;
      Value _v;
   };
   tst_node* root;
   ...
static  tst_node* _lookup(tst_node* t,
        int i, const Key& k, const Value& v);
static  tst_node* _insert(tst_node* t,
        int i, const Key& k, const Value& v);
   ...
};
```

Ternary Search Trees



Ternary Search Trees

```
// Pre: p points to the root of the subtree that contains
// all elements such that their first i-1 symbols
// coincide with the first i-1 symbols of the key k
// Post: returns a pointer to the node that stores the value
// associated to $k$ if $\pair{k,v}$ belongs to the dictionary
// and a null pointer if not such pair exists
// Expected cost: O(|k| log m)}
template  <typename Symbol, typename Key,
          typename Value>
DigitalDictionary<Symbol,Key,Value>::trie_node*
DigitalDictionary<Symbol,Key,Value>::_lookup(tst_node* p,
     const Key& k, int i) const {

   if (p == nullptr)    return nullptr;
   if (i == k.length()) return p;
   if (k[i] < p -> _c > k[i]) return _lookup(p -> _left, k, i);;
   if (k[i] == p -> _c) return _lookup(p -> _cen, k, i+1);;
   if (p -> _c < k[i]) return _lookup(p -> _right, k, i);
}
```

Ternary Search Trees

```
template <typename Symbol,
          typename Key,
          typename Value>
void DigitalDictionary<Symbol,Key,Value>::insert(
     const Key& k, const Value& v) {
   // Symbol() is the end-of-string symbol
   // e.g. Symbol() == '\0' or Symbol() == '\sharp'
   k[k.length()] = Symbol(); // add end-of-string
   root = _insert(root, 0, k, v);
}
```

Ternary Search Trees

```
template <typename Symbol,
          typename Key,
          typename Value>
DigitalDictionary<Symbol,Key,Value>::tst_node*
DigitalDictionary<Symbol,Key,Value>::_insert(
          tst_node* t, int i,
          const Key& k, const Value& v) {

   if (t == nullptr) {
      t = new tst_node;
      t -> _left = t -> _right = t -> cen = nullptr;
      t -> _c = k[i];
      if (i < k.length() - 1) {
         t -> _cen = _insert(t -> _cen, i + 1, k, v);
      } else { // i == k.length() - 1; k[i] == Symbol()
         t -> _v = v;
      }
   } else {
      if (t -> _c == k[i])
         t -> _cen = _insert(t -> _cen, i + 1, k, v);
      if (k[i] < t -> _c)
         t -> _left = _insert(t -> _left, i, k, v);
      if (t -> _c < k[i])
         t -> _right = _insert(t -> _right, i, k, v);
   }
   return t;
}
```

Performance of Tries

There are several measures of the performance of tries in terms of space and time of the different operations.

For example, in tries using arrays of pointers and considering an extended alphabet with $m + 1$ symbols a tree for a set of n elements will contain $\geq n$ leaves; how many of them?

A common model to study the average behavior of tries (array, list or TST) is to consider that the n strings are produced by some memoryless source (so that the r-th symbol of the element is symbol σ_i with a probability p_i irresp. of the previous symbols and r) or some Markovian model there is a probability $p_{i,j}$ that some symbol is σ_i given that the preceding sysmbol was σ_j

Performance of Tries

Theorem (Clément, Flajolet, Vallée (1998))

The external path length (EPL) in a random trie of n elements produced by a random source S is

$$\frac{C_S}{H_S} n \log n + o(n \log n)$$

where both C_S are H_S are constants dependening on the random source S; moreover C_S depends on the specific implementation of the trie (array, list, TST).

The EPL is the sum of the length of all paths from the root of the trie to all leaves in the trie; a random search (successful or unsuccessful) will cost, on average $C_S / H_S \log n$

Performance of Tries

For example, if the source is memoryless with probabilities p_1, $\ldots$, p_m, for the symbols of the alphabet ($\sum_i p_i = 1$) then $H_S = -\sum_i p_i \log p_i$ is the entropy of the source and

Type	C_S
Array	1
List	$\sum_i (i-1)p_i$
TST	$2\sum_{i<j} \frac{p_i p_j}{p_i+\ldots+p_j}$

When all $p_i = 1/m$ we have that $H_S = \log m$ and hence the cost of random searches is

Type	Cost of random search
Array	$\log_m n$
List	$\approx \frac{m}{2} \log_m n$
TST	$\approx 2 \ln m \log_m n = 2 \ln n$

Performance of Tries

Let N denote the total number of symbols (including end-of-string, if needed) required for our n strings. The shared prefixes will provide for a more compact representation of the set of strings and thus we expect to need $\ll N$ nodes (assuming we store one symbol per node like in list-tries and TSTs).
On the other hand, it has been shown that to store n strings in a trie we will need, on average, n/H_S internal nodes (Knuth 1968, Regnier 1988), hence the average number of pointers is $n \cdot (m+1)/H_S$ (array-tries), $2n/H_S$ (list-tries) and $3n/H_s$ (TSTs).

Performance of Tries

For example, for a memoryless source with m symbols all equally likely, we will need on average $n/\ln m$ nodes, e.g., $n/\ln 2$ nodes in a binary trie.
In list-tries and TSTs the number of nodes coincides, in the worst-case, with N as each node holds one symbol. But in practice the common prefixes will be exploited giving a reduction by a factor of $1/H_S$ ($\approx 1/\ln m$ for string made out of equally likely independent symbols)

Patricia (a.k.a. Compressed Tries)

When an internal node in a trie has only one non-empty subtree we say that such a node is redundant. In a compressed trie or Patricia (*Practical Algorithm To Retrieve Information Coded in Alphanumeric, D. Morrison, 1968*) chains of redudant nodes are substitutedby a single node, and edges labeled by subsequence of one or more symbols.
In a Patricia the n strings are stored in the n leaves, and by definition there are no empty leaves with one or more non-empty siblings.

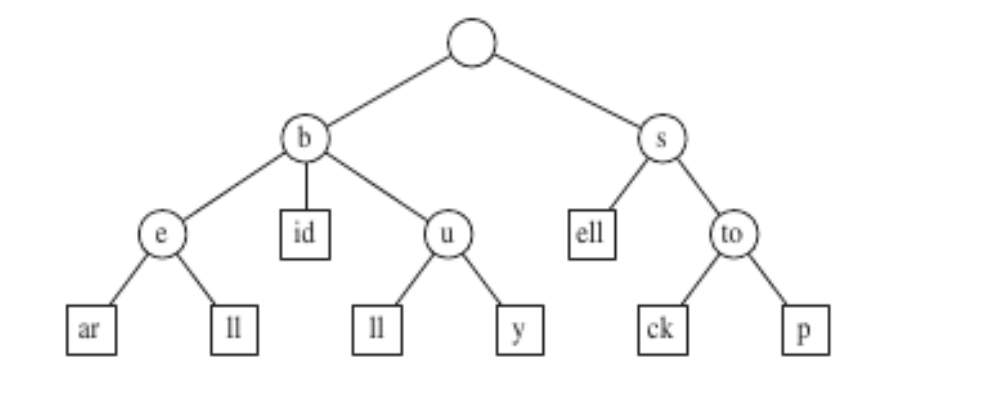
Patricia

To implement Patricia we can use the first child/next sibling or the TST representation, but instead of storing one symbol per node, we can store a substring of ≥ 1 symbols, or keep `skip` attributes that indicate which position has to be examined next if we move to a descendant node in the trie.

Thus during a search, if we reach a node x at level i which we have matched the first $i-1$ symbols of the given key and the node has the substring $w = w_0 \ldots w_{j-1}$ as a label, then we will have to see if $k[i-1..i+j-2]$ matches w, and if so, we "descend", otherwise we will continue looking for the apropriate subtree or declare the search unsuccessful–this will depend on whether we are compacting a list-trie or a TST.

Patricia

- In a list-Patricia, the first child pointer will point to the subtree that contains all strings with prefix $k[0..i-2] \cdot w$, and the next sibling will give us access to the subtrees that store words with a prefix $\geq k[0..i-2] \cdot w_0'$, where w_0' is the successor of w_0 i Σ in the alphabetic order.
- In a bst-Patricia, the central child contains all strings with prefix $k[0..i-2] \cdot w$, the left child all strings with a smaller prefix, the right child all strings with a larger prefix



A Patricia for the set $X = \{\text{bear, bell, . . . , stock, stop}\}$.

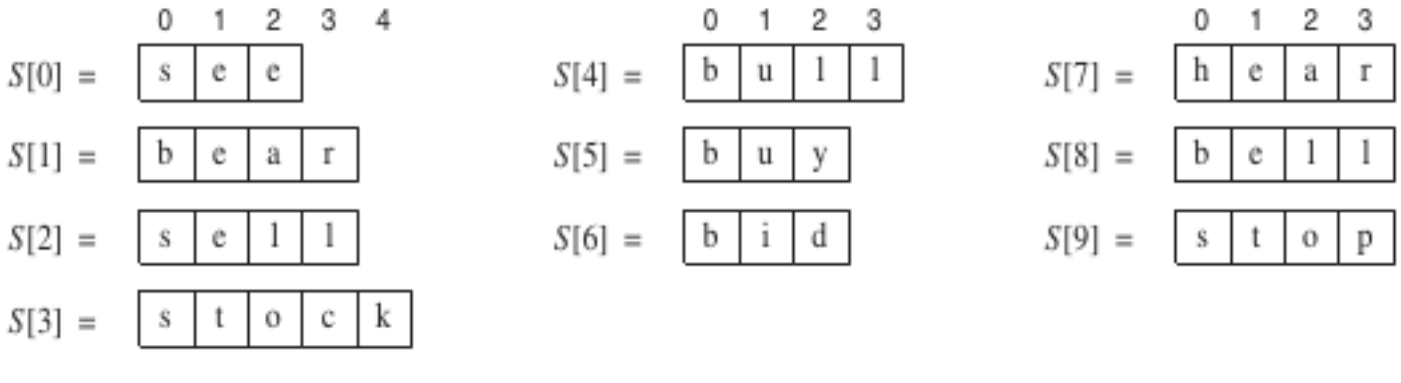
Source: M. T. Goodrich & R. Tamassia

Patricia

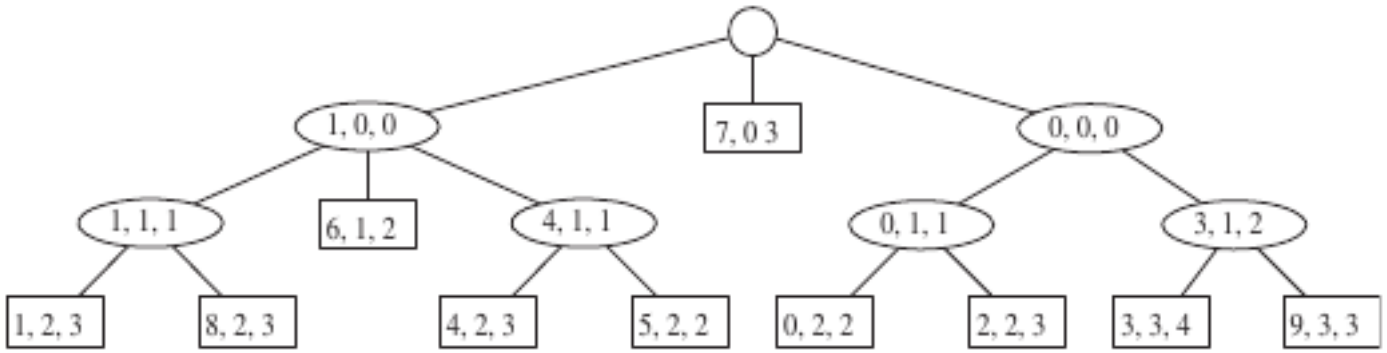
Despite Patricia saves on empty leaves and it might be slightly more efficient to store common infixes in the internal nodes instead of the full expanded infixes, one symbol per node, but the major advantadge of Patricia occurs when the string are externally store and Patricia is an index into the external storage.

For example, if we have an array of strings $S[0..M-1]$, we can designate the substring between positions i and j of $S[k]$ by a triplet $\langle k; i, j \rangle$; we can build our Patricia storing such triplets in the nodes instead of symbols or subsequences of symbols.

Patricia


0 1 2 3 4
S[0] = s e e
S[1] = b e a r
S[2] = s e l l
S[3] = s t o c k
0 1 2 3
S[4] = b u l l
S[5] = b u y
S[6] = b i d
0 1 2 3
S[7] = h e a r
S[8] = b e l l
S[9] = s t o p

(a)


1, 0, 0
7, 0 3
0, 0, 0
1, 1, 1
6, 1, 2
4, 1, 1
0, 1, 1
3, 1, 2
1, 2, 3
8, 2, 3
4, 2, 3
5, 2, 2
0, 2, 2
2, 2, 3
3, 3, 4
9, 3, 3

(b)

Patricia

```
template  <typename Symbol, typename Key,
          typename Value>
DigitalDictionary<Symbol,Key,Value>::patricia_node*
DigitalDictionary<Symbol,Key,Value>::_lookup(patricia_node* p,
     const Key& k, int i) const {

   if (p == nullptr)    return nullptr;
   if (i == k.length()) return p;
   triplet x = p -> _x; // x=(idx,first,last)
   int len = x.last - x.first + 1; // length of the subsequence
   if (k[i..i+len-1] <  S[x.idx][first..last])  return nullptr;
   if (S[idx][first..last] < k[i..i+len-1])
      return _lookup(p -> _next_sibl, k, i);
   // S[idx][first..last] == k[i..i+len-1]
   return _lookup(p -> _first_child, k, i+len);
}
```

Patricia

The lookup algorithm in Patricia has cost $\mathcal{O}(\ell \cdot m)$ in the worst-case, where ℓ is the length of the longest string in the set, and m the size of the alphabet. If we combine TST & compression we get expected cost $\mathcal{O}(\ell \cdot \log m)$ for searches. Likewise the cost of insertions and deletions will be like that of search.

Inverted files

One interesting application of tries and Patricia is inverted files (a.k.a. inverted indices).
Suppose we have a large collection of documents $D_1, \ldots, D_T$.
For each document we extract the unique set of words (vocabulary, index terms) of each document (we eventually record the positions of the document at which each unique word occurs). It is also frequent to remove common words such as pronouns, articles, connectives, ... known as stopwords.

Inverted files

We then proceed to insert/update, one by one, each index term of each document, in a trie (or TST or Patricia). When a word appears in several different documents we will keep track in a occurrence list. Because of their sheer volume, occurrence lists will be typically stored in secondary memory, and they won't be kept in any particular order.
When we process a word w from document D_i, we consider three cases

1. w is a stopword: discard it and proceed to the next word/term in D_i (or start processing a new document)
2. w was already in the inverted file: use the (compressed) trie to located the occurrence list and add a reference to document D_i to the list associated to word w; or
3. w wasn't yet in the inverted list: create a new occurrence list with (w, D_i), append the new occurrence list to the set of occurrence lists, and add a link from the (compressed) trie to the new occurence list

Inverted files

Inverted indices are used in search engines to retrieve relevant documents as follows.

1. The user query Q is normalized and stopwords removed
2. For each term/word t in Q, use the compressed trie (in main memory) to get the link to the corresponding occurrence list for t and retrieve it from secondary memory
3. Intersect (*) the occurence lists and sort the resulting set of documents according to some *relevance parameter* (e.g. the PageRank when the documents are wbe pages)

(*) This is what we do when it is assumed that we want the subset of documents that contain **all** the terms in Q; in some case, the user will use operators or will allow a certain degree of mismatch, or we will have to produce results discarding terms that do not appear in the index

Inverted files

If $|V_i|$ is the size of the vocabulary V_i of D_i, without stopwords, then we will be building a (compressed) trie for

$$N = \left| \bigcup_{i=1}^{T} V_i \right|$$

words/terms, and the space that it will occupy will be roughly $\Theta(N/H_S)$, H_S being the empirical entropy for Σ, based upon the set of documents. On the other hand, we will have to store the N words and their respective occurrence lists somewhere else.

To construct the inverted file we will have to perform $D = \sum_{1 \leq i \leq T} |D_i|$ insertions/updates in the index and each such operation will have cost $\mathcal{O}(\ell)$, where ℓ is the length of the longest word in the collection of documents.

Inverted files

The cost of processing a query will be the cost of searching the Q terms in the (compressed) trie ($\mathcal{O}(|Q| \cdot \ell)$ plus the cost of merging the $|Q|$ occurrence lists and sorting the final result by relevance.
In practical situations the occurrence lists can be long, but not extremely long (words that **are not** stopwords do not appear in a significative fraction of the documents in the collection!). This is even more true for the final result (the "merging" of all occurrence lists) and the cost of sorting will be small compared to the others, hence the cost will be $\ll \Theta(|Q| \cdot T + T \log T)$, indeed, it will be close to $\Theta(|Q|)$, as the length of the occurrence lists can be though as $\mathcal{O}(1)$

Tries

To learn more:

[1] D. E. Knuth
The Art of Computer Programming, Volume 3: Sorting and Searching, 2nd ed
Addison-Wesley, 1998

[2] M. T. Goodrich and R. Tamassia
Algorithm Design and Applications
John Wiley & Sons, 2015

Tries

To learn more:

[1] J. L. Bentley and R. Sedgewick
Fast algorithms for sorting and searching strings
Proc. SODA, pp. 360–369, 1997

[2] J. Clément, Ph. Flajolet and B. Vallée
The Analysis of Hybrid Trie Structures
Proc. SODA, pp. 531–539, 1998

Part V

Data Structures for String Processing

15 Tries

16 Suffix Trees

Suffix Trees

A suffix tree (or *suffix trie*) is simply a trie for all the suffixes of a string, the text, $T[0..n-1]$.

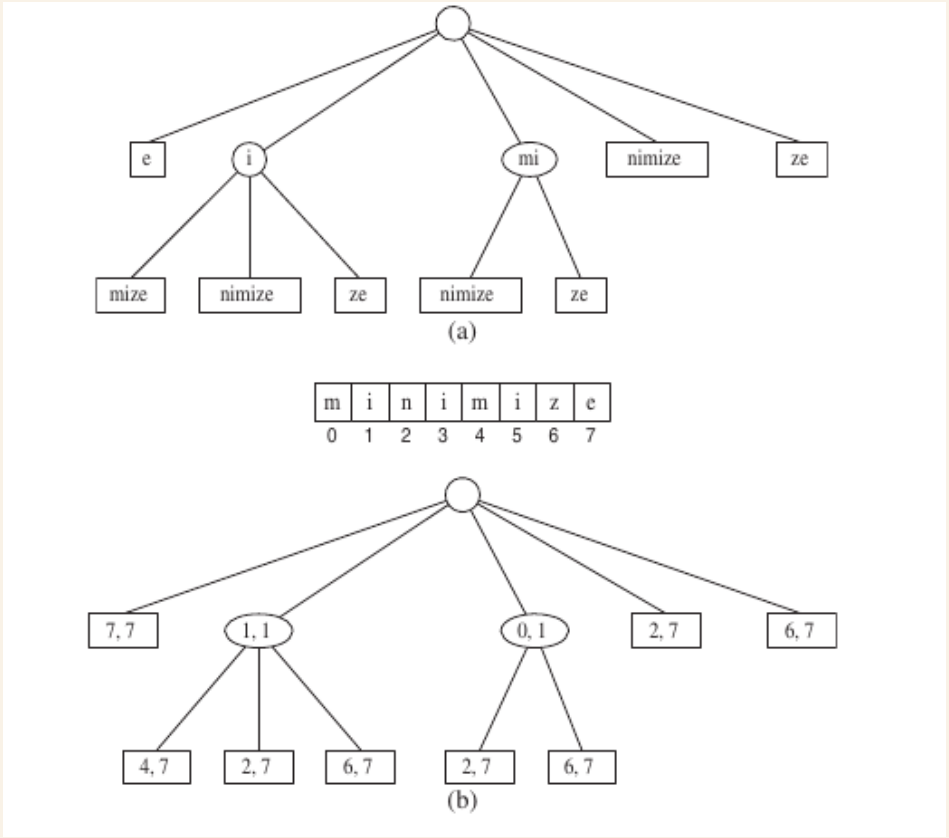
Thus we will form a trie with the M suffixes of the text T. Since the length of the text is n there are The total number of (proper) suffixes to store is n ($n+1$ if we count the empty suffix) and the total number of symbols involved is

$$\leq \sum_{i=0}^{n} n - i = \frac{n(n+1)}{2}$$

as the suffix $T[i..n-1]$ has length $n-i$, $0 \leq i \leq n$

Suffix Tries

A compact representation of the trie, storing the pair (i, j) to represent a substring $T[i..j]$ will be most convenient; in any case, using Patricia for the suffixes guarantees that the space used is $\Theta(n)$.



(a) the suffix trie for the text $T =$ minimize

Suffix Tries

A naÃ¯ve approach to the construction of suffix tries would need $\Theta(n^2)$ in the worst case—assuming that we use the array of pointers representation with $\Theta(1)$ cost to find which children to use in the next level, otherwise an extra factor m or $\log m$ appears

However there exist several linear-time algorithms for the construction of suffix tries

1. Weiner, 1973 (*position trees*)
2. McCreight, 1976 (more space efficient than Weiner's)
3. Ukkonen, 1995 (same bounds as McCreight's, simpler)

They won't be covered here

Suffix Tries

Suffix tries have many applications. The most immediate one is the substring matching problem. We are given a text $T[0..n-1]$ and a pattern $P[0..k-1]$, typically $k \ll n$ and we need to show if P occurs as a substring of T, and if so, where.

Well known algorithms like Knuth-Morris-Pratt (KMP) or Boyer-Moore (BM) —there many other— solve this important problem in time by preprocessing the pattern in time $\Theta(k)$ and then scanning the text in time $\Theta(n)$, giving a total cost $\Theta(n+k)$ in the worst-case

Suffix Tries

The same bound is achieved with suffix tries: but we invest time $\Theta(n)$ in the preprocessing of the text ($\rightarrow$ build the suffix trie!) and then search the pattern in the suffix trie with cost $\Theta(k)$.

The great news is that we can search many patterns very efficiently, the cost of building the suffix trie was paid once, the search of each pattern is lightning fast. This simply not possible with KMP, BM and many of the string mathcing algorithms since they preprocess each pattern and will need to scan the text with cost $\Theta(n)$ for every pattern. Unless we were given all the patterns in advance, in which case we can preprocess the whole set of p patterns at once (with cost $\Theta(k \cdot p + n)$ instead of cost $\Theta(k + n) \cdot p)$). But this is not the case in many situations where the patterns to be searched are not known in advance.

Suffix tries

```
int k = P.size();
int j = 0;
suffix_trie_node* p = T.root;
do {
  bool fin = true;
  for(q:children of p) {
    int i = q.first;
    if (P[j] == T[i]) {
      int len = q.last - i + 1;
      if (k <= len) {
        // suffix is shorter that node label
        if (P[j..j+k-1] == T[i..i+k-1])
          return ``match at i-j''
        else
          return ``P not a substring of T''
      } else { // k > len
        if (P[j..j+len-1]==T[i..i+len-1])
        k -= len; j += len; p = q;
        fin = false;
        break; // end the for(q:children of p) loop
      }
    }
  }
} while (not fin and p is not a leaf);
return ``P not a substring of T'';
```

Tries

To learn more:

[1] D. E. Knuth
The Art of Computer Programming, Volume 3: Sorting and Searching, 2nd ed
Addison-Wesley, 1998

[2] M. T. Goodrich and R. Tamassia
Algorithm Design and Applications
John Wiley & Sons, 2015

[3] Dan Gusfield
Algorithms on Strings, Trees & Sequences
Cambridge Univ. Press, 1997

Part VI

Multidimensional Data Structures

17 Multidimensional Data Structures: Introduction

18 K-Dimensional Search Trees

19 Quad Trees

20 Analysis of the Cost of Associative Queries

Why Multidimensional?

Multidimensional data everywhere:

- Points, lines,
- rivers, maps, cities, roads,
- hyperplanes, cubes, hypercubes,
- mp3, mp4 and mp5 files,
- jpeg files, pixels,
- . . . ,

Used in applications such as:

- database design, geographic information systems (GIS),
- computer graphics, computer vision, computational geometry, image processing,
- pattern recognition,
- very large scale integration (VLSI) design,
- . . .

Why Multidimensional?

- Data: File of K-dimensional points, K-tuples of the form:

$$x = (x_0, x_1, \ldots, x_{K-1})$$

- Retrieval: associative queries that involve more than one of the K dimensions
- Data structures:
  - K-Dimensional Search Trees (a.k.a. K-d trees)
  - Quad Trees
  - ...

Associative Retrieval

Multidimensional data structures must support:

- Insertions, deletions and (exact) search
- Associative queries such as:

  Partial Match Queries: Find the data points that match some specified coordinates of a given query point q.

  Orthogonal Range Queries: Find the data points that fall within a given hyper rectangle Q (specified by K ranges).

  Nearest Neighbor Queries: Find the closest data point to some given query point q (under a predefined distance).

Associative Queries

Associative Queries

Associative Queries

Partial Match Queries

Definition

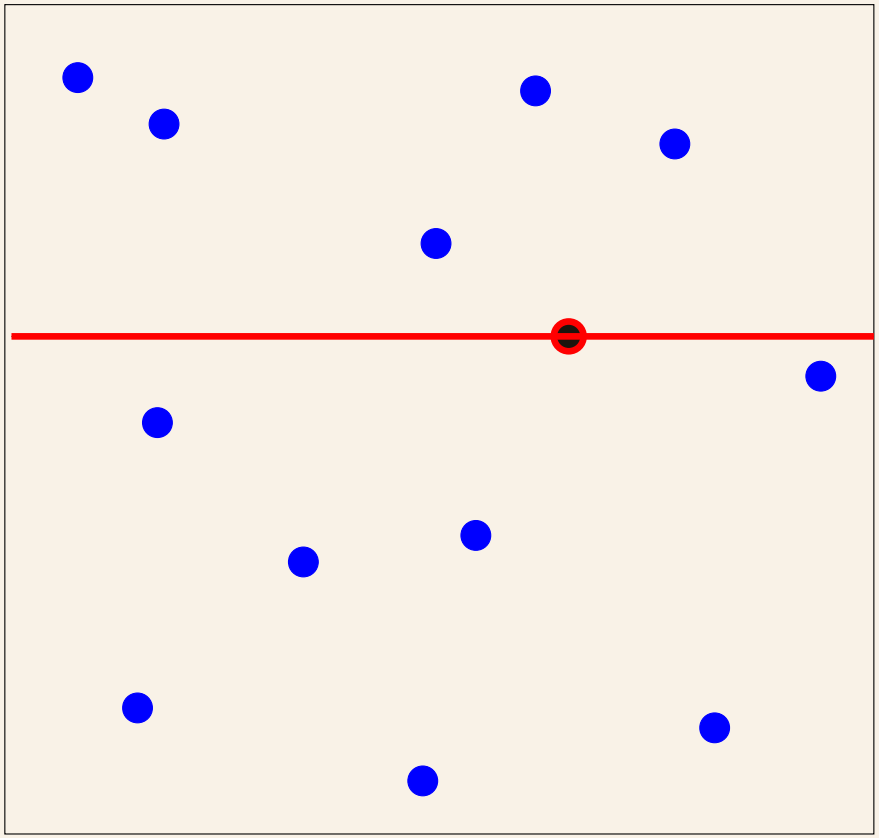
Given a file F of n K-dimensional records and a query $q = (q_0, q_1, \ldots, q_{K-1})$ where each q_i is either a value in D_i (it is specified) or $*$ (it is unspecified), a *partial match query* returns the subset of records x in F whose attributes coincide with the specified attributes of q. This is,

$$\{x \in F \mid q_i = * \text{ or } q_i = x_i,\ \forall i \in \{0, \ldots, K-1\}\}.$$

Partial Match Queries

Example

Query: $q = (*, q_2)$ or $q = (q_1, q_2)$ with specification pattern: 01



Part VI

Multidimensional Data Structures

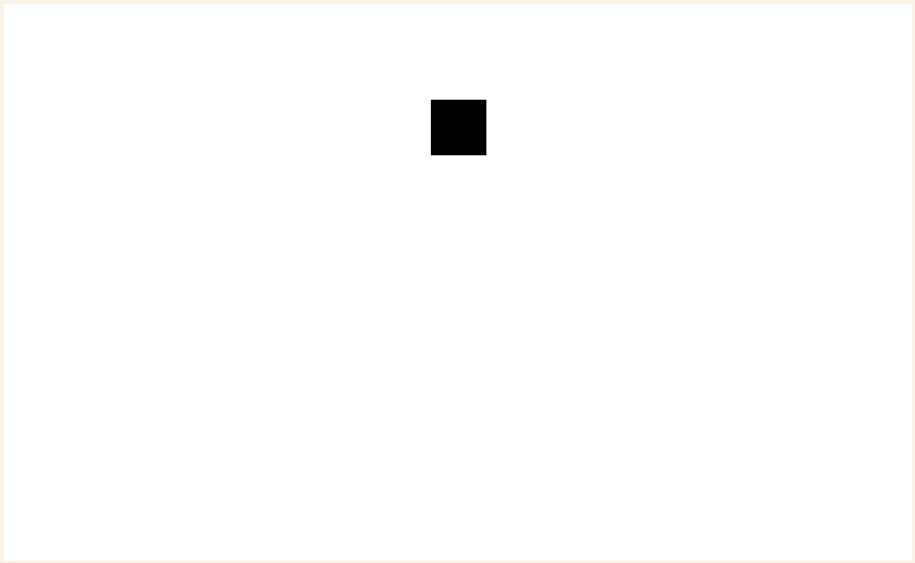
17 Multidimensional Data Structures: Introduction

18 K-Dimensional Search Trees

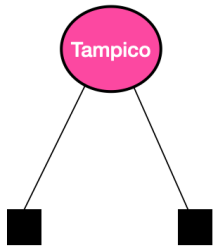
19 Quad Trees

20 Analysis of the Cost of Associative Queries

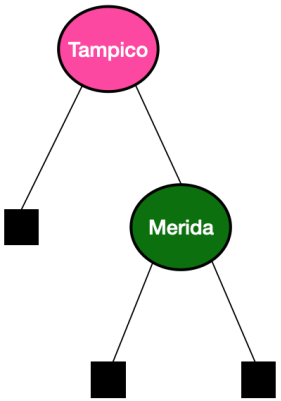
Standard K-d trees



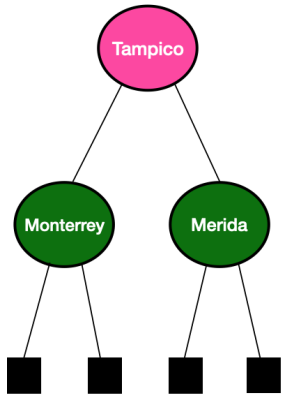
Standard K-d trees



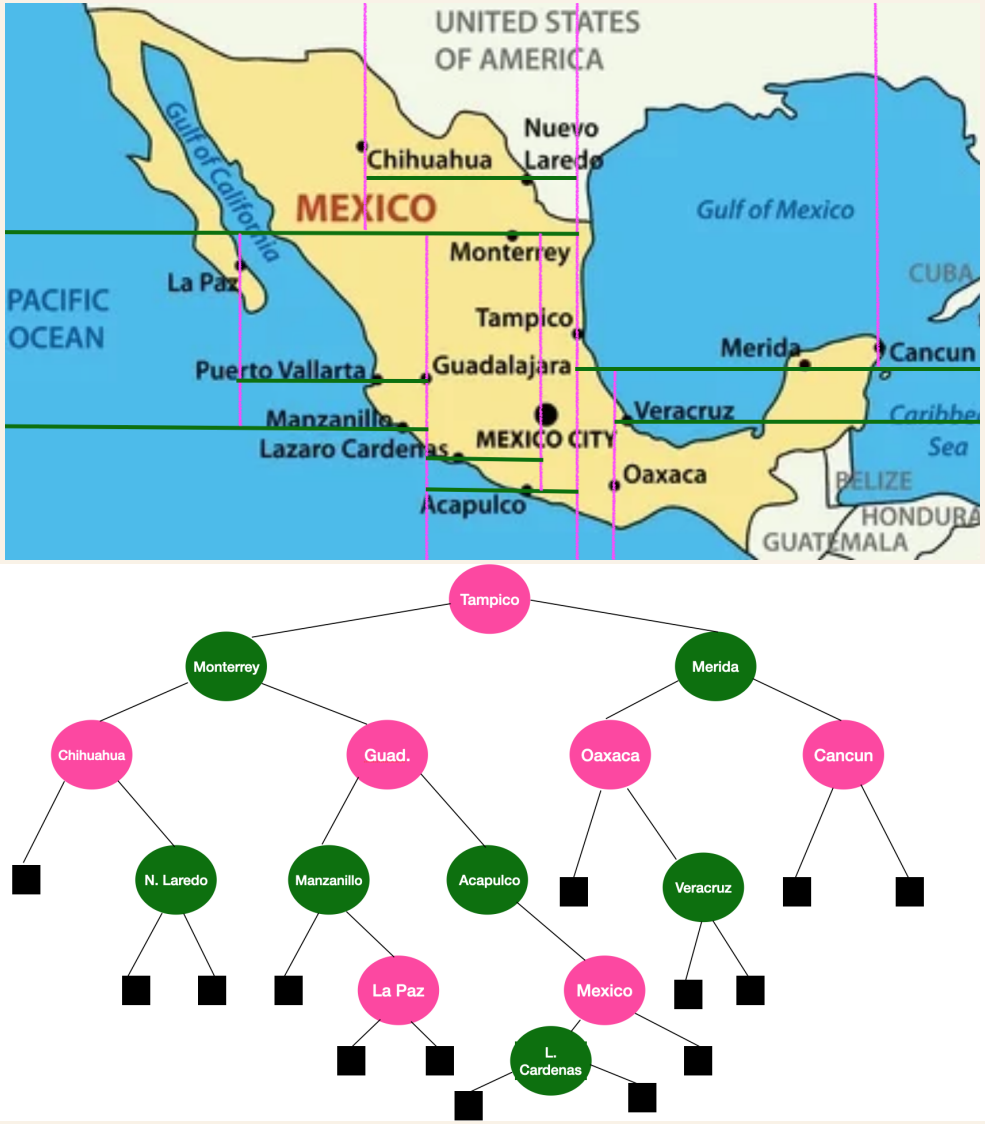
Standard K-d trees



Standard K-d trees



Standard K-d trees



Standard K-d Trees

Definition (Bentley75)

A standard K-d tree T of size $n \geq 0$ is a binary tree that stores a set of n K-dimensional points, such that

- it is empty when $n = 0$, or
- its root stores a key x and a discriminant $j =$ level of the root $\bmod K$, $0 \leq j < K$, and the remaining $n - 1$ records are stored in the left and right subtrees of T, say L and R, in such a way that both L and R are K-d trees; furthermore, for any key $u \in L$, it holds that $u_j < x_j$, and for any key $v \in R$, it holds that $x_j < v_j$.

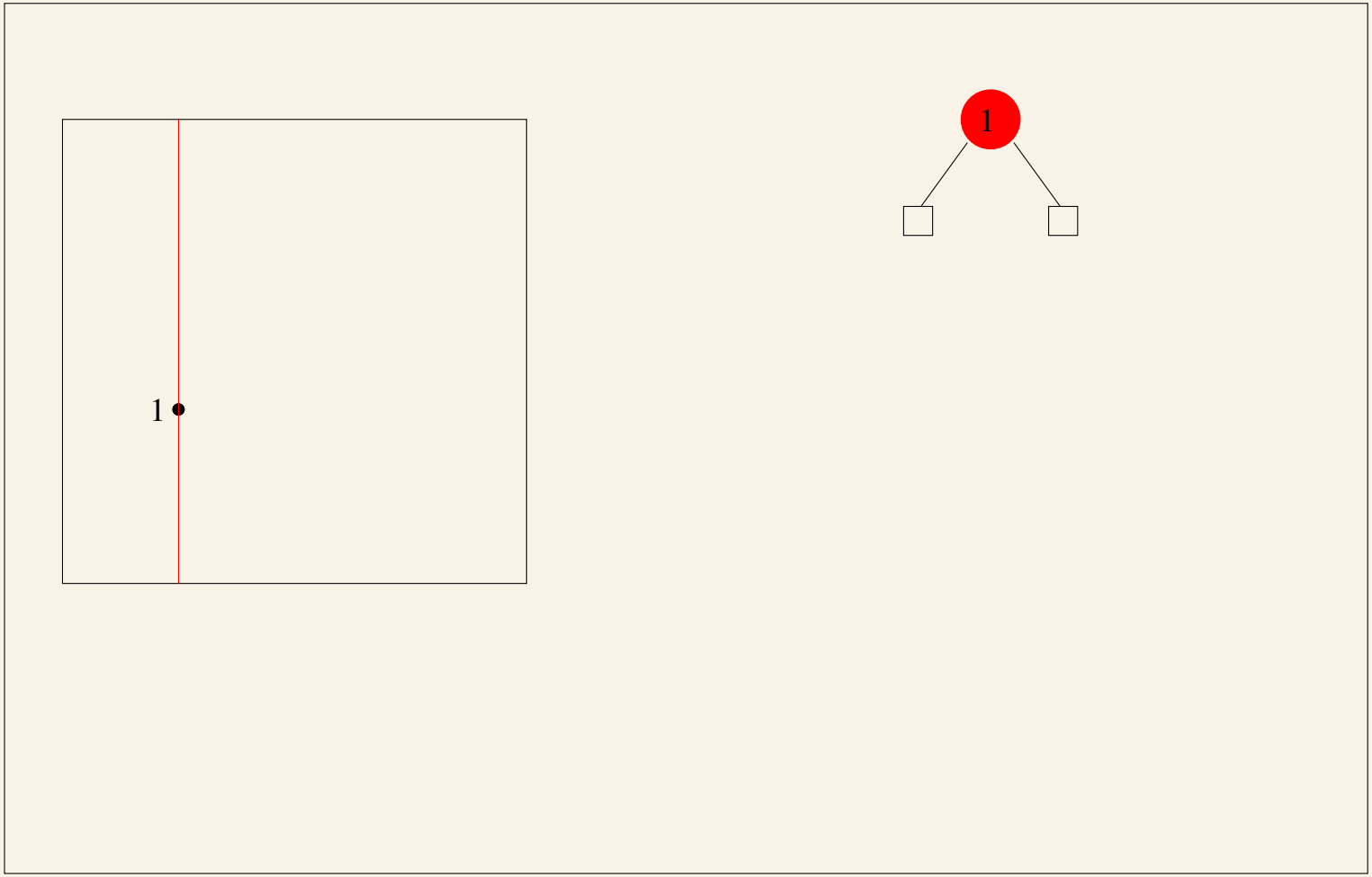
Relaxed K-d trees

A relaxed K-d tree (Duch, Estivill-Castro, Martínez, 1998) for a set of K-dimensional keys is a binary tree in which:

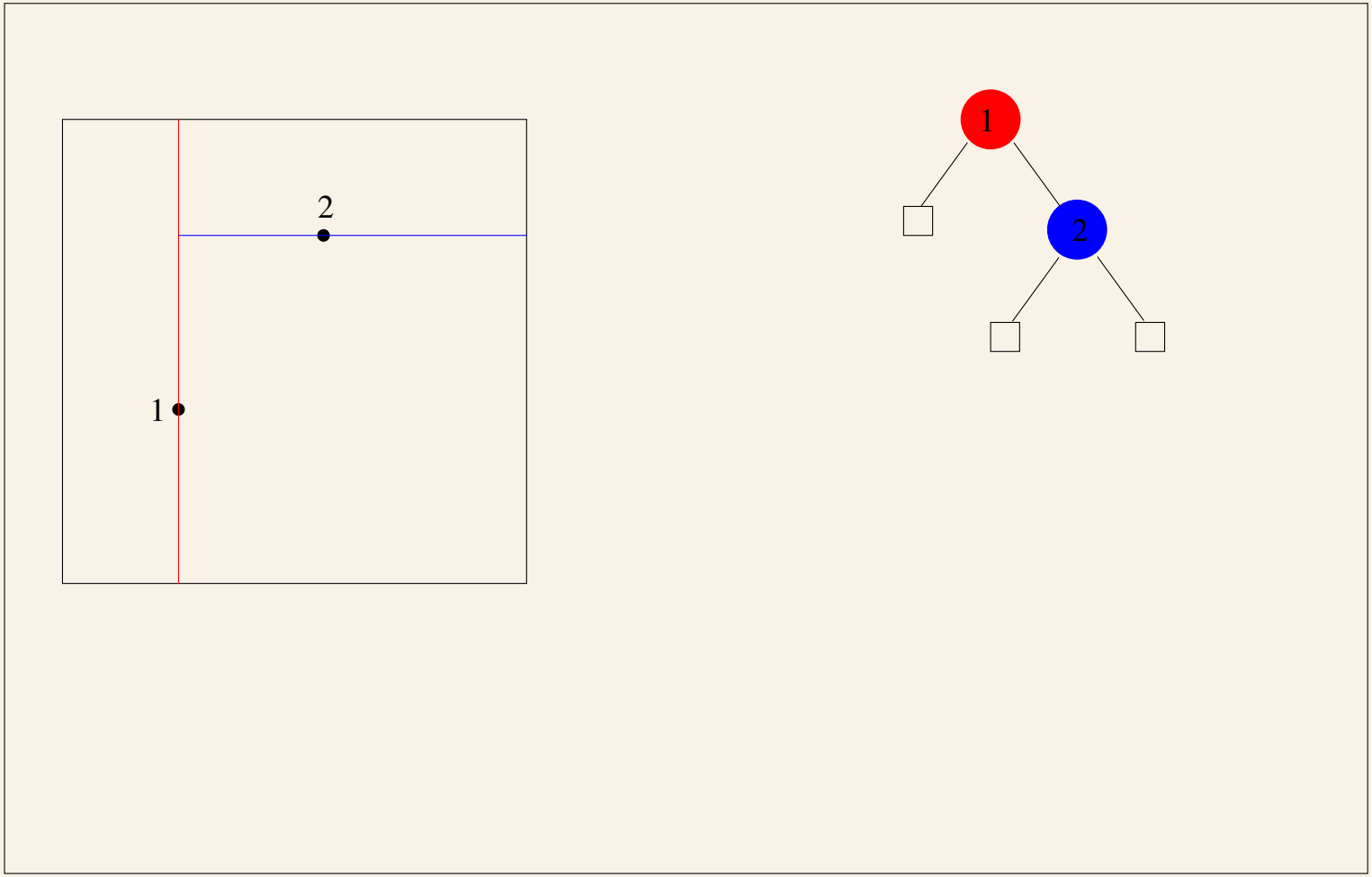
1. Each node contains a K-dimensional record and has associated an arbitrary discriminant $j \in \{0, 1, \ldots, K-1\}$.
2. For every node with key x and discriminant j, the following invariant is true: any record in the right subtree with key y satisfies $y_j < x_j$ and any record in the left subtree with key y satisfies $y_j \geq x_j$.

Relaxed K-d trees

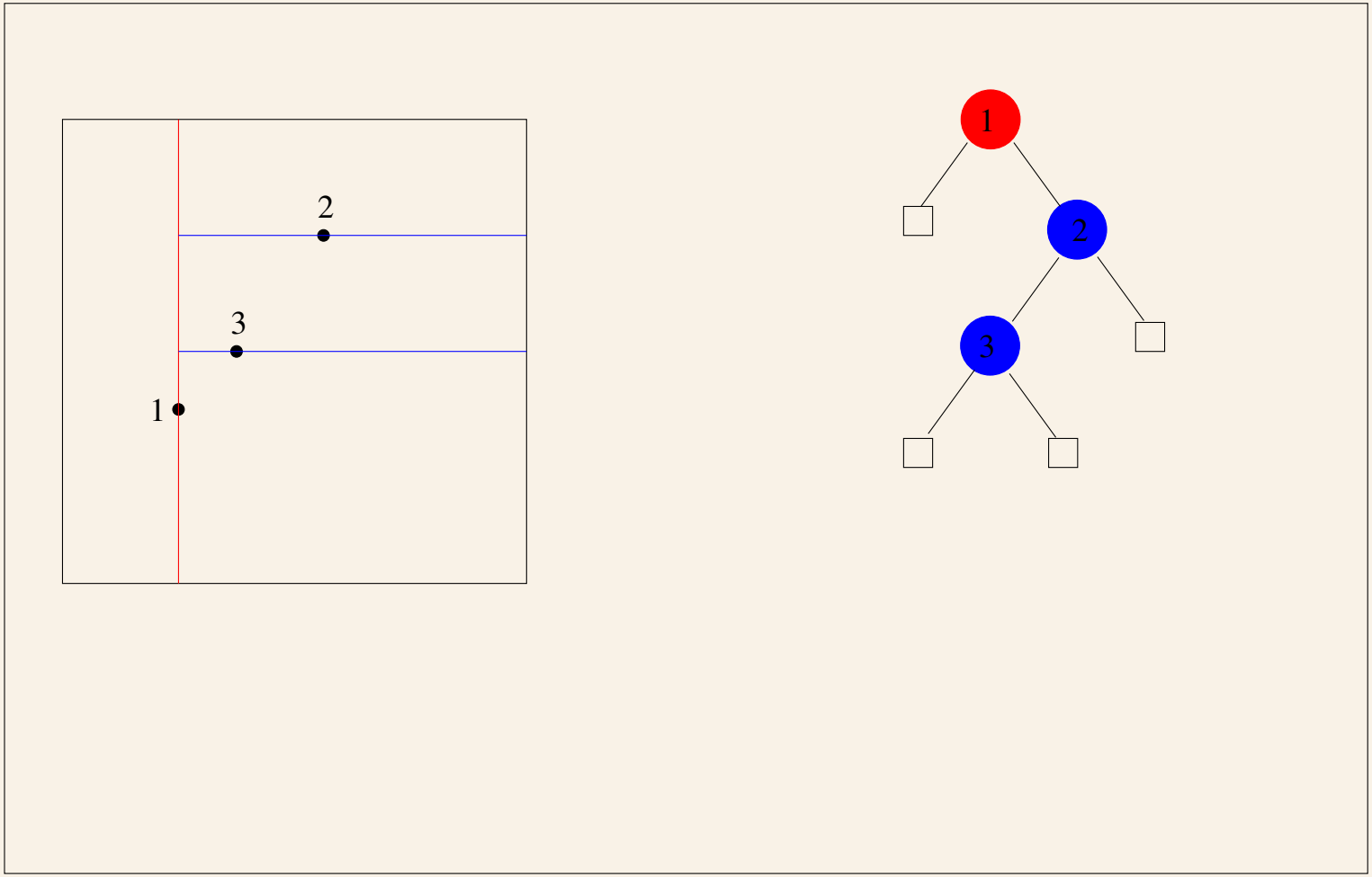
Relaxed K-d trees



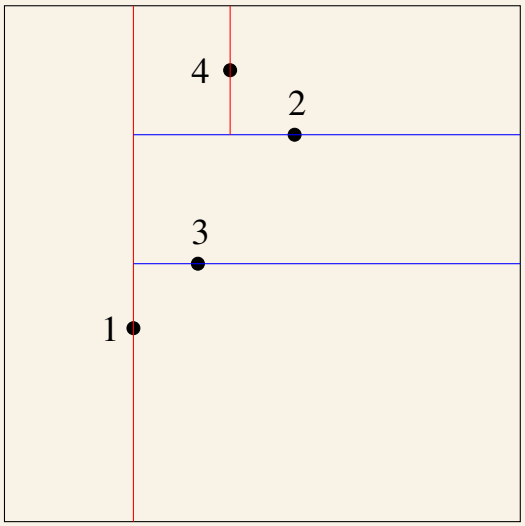
Relaxed K-d trees

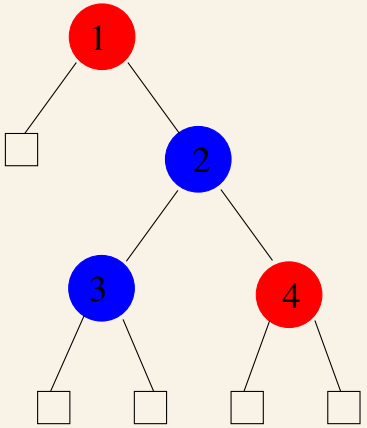


Relaxed K-d trees

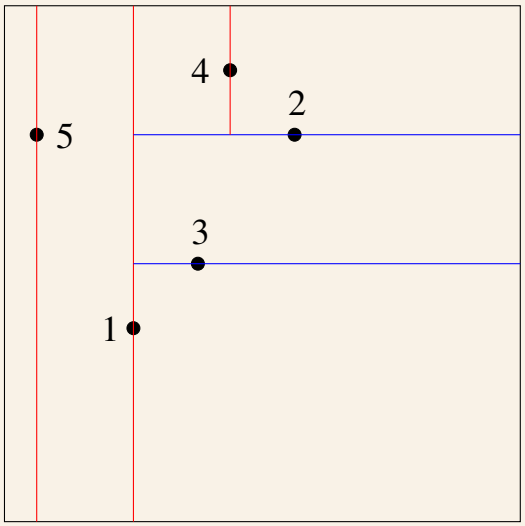


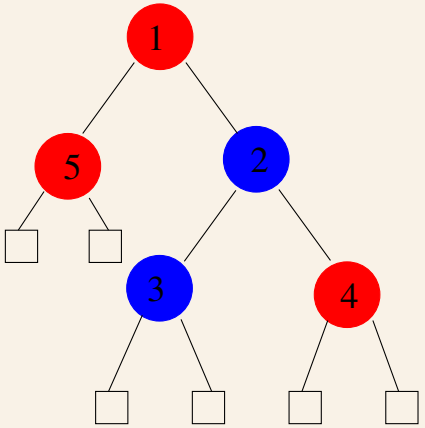
Relaxed K-d trees





Relaxed K-d trees





K-d trees

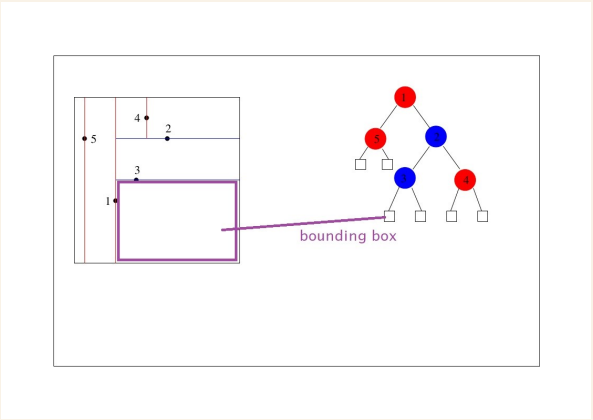
Several other variants of K-d trees have been proposed in the literature, they differ by the way in which discriminants are assigned to nodes. For example:

Squarish K-d trees: Use as discriminant the coordinate which cuts the longest edge of the bounding box

Median K-d trees: Use as discriminant the corrdinate whic is close to the bounding box's center corresponding coordinate

K-d trees

The bounding box of a leaf in the K-d tree T is a region of the domain associated to that leaf, namely, the set of points in the K-dimensional space which will replace that leaf if any of them were inserted into T.



The bounding box of a node x is that of the leaf which that node replaced.

Partial Match in K-d Trees

Partial match search in K-d trees works as follows:

- At each node of the tree we verify if it satisfies the query and we examine its discriminant.
- If the discriminant is specified in the query then the algorithm recursively follow in the appropriate subtree depending on the result of the comparison between the key and the query.
- Otherwise the algorithm recursively follows the two subtrees of the node.

Partial Match Algoritm

```
procedure PartialMatch(T, q)
    ▷ T: tree, q: query
    if T ≠ □ then  ▷ nothing to do if T were empty
        i = T → discr
        if Match(T → key, q) then
            Report(T → key)
        if q[i] ≠ * then  ▷ Coordinate i specified
            if q[i] < T → key[i] then
                PartialMatch(T → left, q)
            else
                PartialMatch(T → right, q)
        else  ▷ Coordinate i not specified, q[i] = *
            PartialMatch(T → left, q)
            PartialMatch(T → right, q)
```

Orthogonal Range and Region Queries in K-d Trees

Orthogonal range and region (even for complex regions) in K-d trees work as follows:

- At each node of the tree we verify if it satisfies the query or not, e.g., the key at the node is inside the orthogonal range (hyper)rectangle, or it is within the given distance from the query "center"
- For each subtree T' of T, we check if the bounding box of T' intersects the region defined by the query; if so we recursively visit T'

Orthogonal Range Algoritm

procedure OrthogonalRange(T, Q)
  ▷ T: tree, $Q = [\ell_0, u_0] \times [\ell_{K-1}, u_{K-1}]$: query
  if $T \neq \square$ **then** ▷ nothing to do if T were empty
    $i = T \to$ discr
    if Inside($T \to$ key, Q) **then**
      Report($T \to$ key)
    if $Q[i].\ell \leq T \to$ key$[i]$ **then**
      OrthogonalRange($T \to$ left, Q)
    if $T \to$ key$[i] \leq Q[i].u$ **then**
      OrthogonalRange($T \to$ right, q)

Nearest Neighbors

In order to find the k nearest neighbors of point q (the query) in the K-d tree T, we carry out a region search but instead of a fixed distance we update the distance at the search proceeds; in particular the distance is at all moment the k-th smallest distance observed so far.

The algorithm maintains a list S of the k points of the tree that are closest to the query so far. It is convenient to maintain such list as a (min) priority queue, with distances to q as priorities.

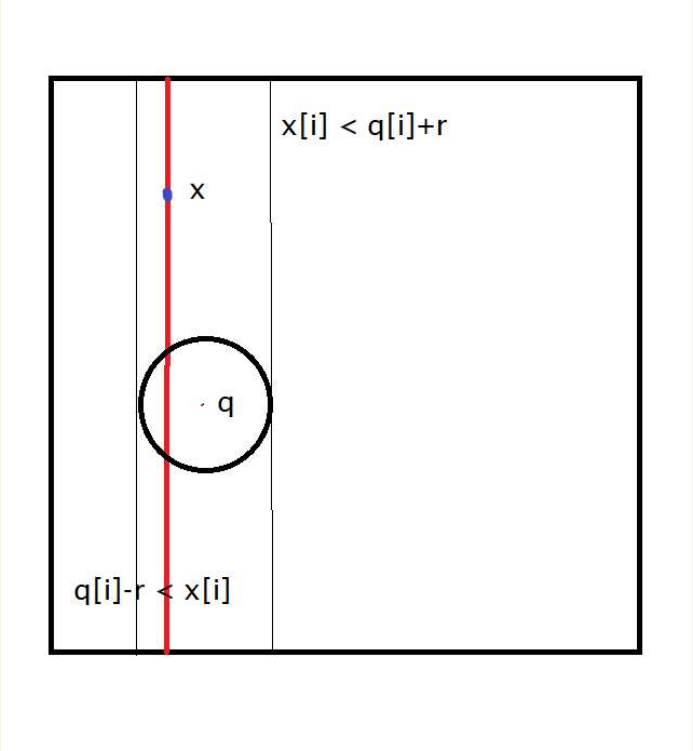
If the current node x is at distance less than $r \equiv S.\texttt{min\_prio}()$ then an element of minimum priority is extracted from S and x is added to S.

Nearest Neighbors

Let i be the discriminant at the current node x, and $r = S.\texttt{min\_prio}()$ the current radius of search. If $q[i] - r \leq x[i]$ then the left subtree bounding box intersects the query region and the left subtree must be explored. Likewise if $x[i] \leq q[i] + r$ then the right subtree bounding box intersects the query region and the right subtree must be explored.

If both conditions were true we have to make both recursive calls; but it must turn out that after completion of the first recursive call the value of r has been update and the visit to the other subtree is no longer needed. For that reason the algorithm chooses to visit first the left or the right subtree, depending on which side is most likely help avoid the second call.

Nearest Neighbors



Nearest Neighbors Algoritm

```
procedure NN(T, q, S, r)
    ▷ T: tree, q: query, S: result, initially empty
    ▷ r: search radius, initially +∞
    if T ≠ □ then  ▷ nothing to do if T were empty
        i = T → discr
        if d := DISTANCE(T → key, q) ≤ r then
            if |S| ≥ k then
                S.EXTRACT_MIN()
                S.INSERT(T → key, d)
                r := S.MIN_PRIO()
            else
                S.INSERT(T → key, d)
        ... ▷ see next slide
```

Nearest Neighbors Algoritm

```
procedure NN(T, q, S, r)
    ...
    if q[i] − r ≤ T → key[i] ∧ T → key[i] ≤ q[i] + r then
        if q[i] ≤ T → key[i] then
            NN(T → left, q, S, r)
            if T → key[i] ≤ q[i] + r then
                NN(T → right, q, S, r)
        else
            NN(T → right, q, S, r)
            if q[i] − r ≤ T → key[i] then
                NN(T → left, q, S, r)
    else
        ▷ query region does no intersect both BB's
        ▷ visit the subtree that corresponds
    ...
```

Part VI

Multidimensional Data Structures

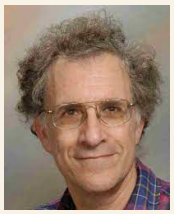
17 Multidimensional Data Structures: Introduction

18 K-Dimensional Search Trees

19 Quad Trees

20 Analysis of the Cost of Associative Queries

2-d Quad Trees

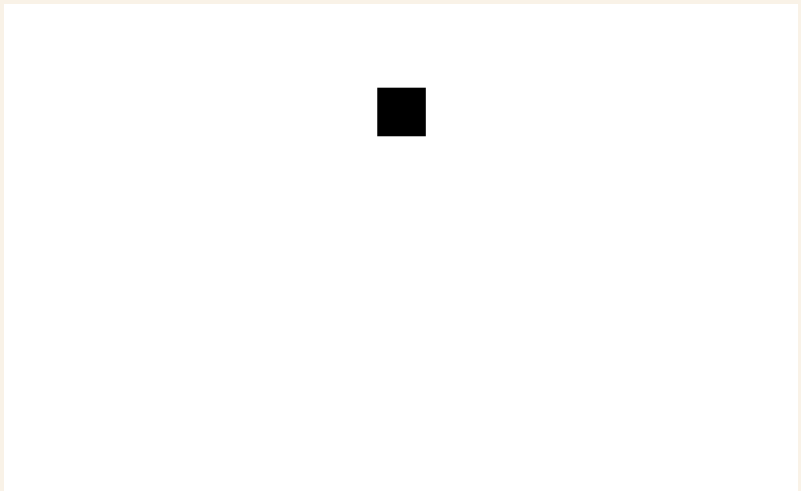
Definition (Bentley & Finkel, 1974)

A quad tree for a file of 2-dimensional records, is a quaternary tree in which:

1. Each node contains a 2-dimensional key and has associated four subtrees corresponding to the quadrants NW, NE, SE and SW.
2. For every node with key x the following invariant is true: any record in the NW subtree with key y satisfies $y_1 < x_1$ and $y_2 \geq x_2$; any record in the NE subtree with key y satisfies $y_1 \geq x_1$ and $y_2 \geq x_2$; any record in the SE subtree with key y satisfies $y_1 \geq x_1$ and $y_2 < x_2$; and, any record in the SW

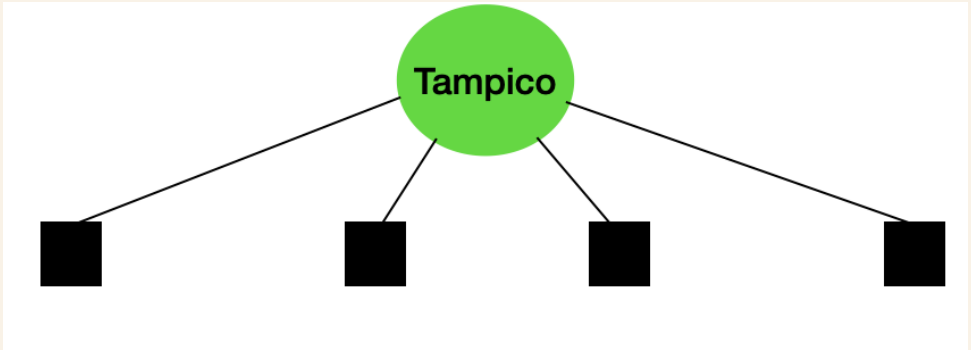
Quad Trees


UNITED STATES
OF AMERICA
Gulf of California
Chihuahua
Nuevo
Laredo
MEXICO
Gulf of Mexico
Monterrey
CUBA
La Paz
PACIFIC
OCEAN
Tampico
Merida
Cancun
Puerto Vallarta
Guadalajara
Veracruz
Manzanillo
MEXICO CITY
Lazaro Cardenas
Caribbe
Sea
Oaxaca
BELIZE
Acapulco
HONDURA
GUATEMALA



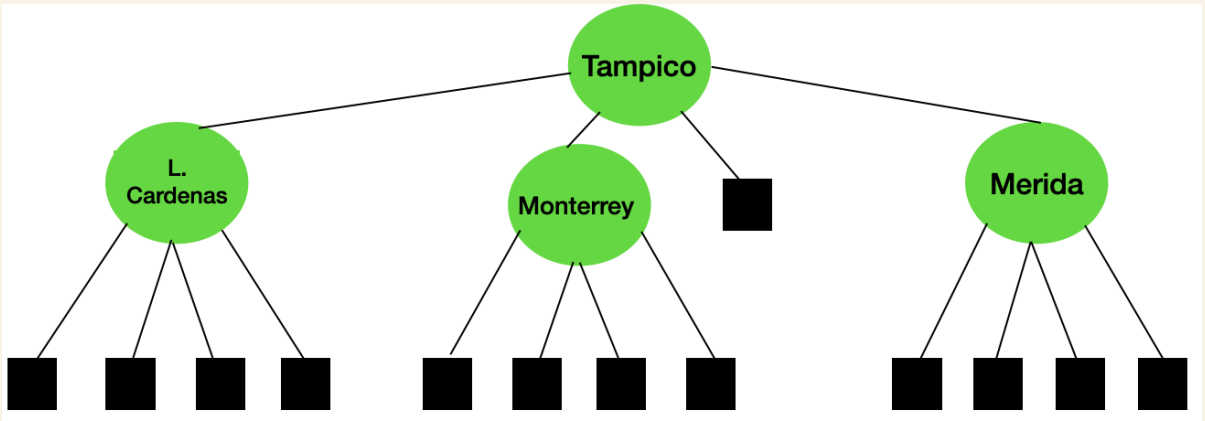
Quad Trees


UNITED STATES
OF AMERICA
Gulf of California
Chihuahua
Nuevo
Laredo
MEXICO
Gulf of Mexico
Monterrey
La Paz
CUBA
PACIFIC
OCEAN
Tampico
Merida
Cancun
Puerto Vallarta
Guadalajara
Veracruz
Manzanillo
Lazaro Cardenas
MEXICO CITY
Caribbe
Sea
Oaxaca
BELIZE
Acapulco
HONDURA
GUATEMALA


Tampico

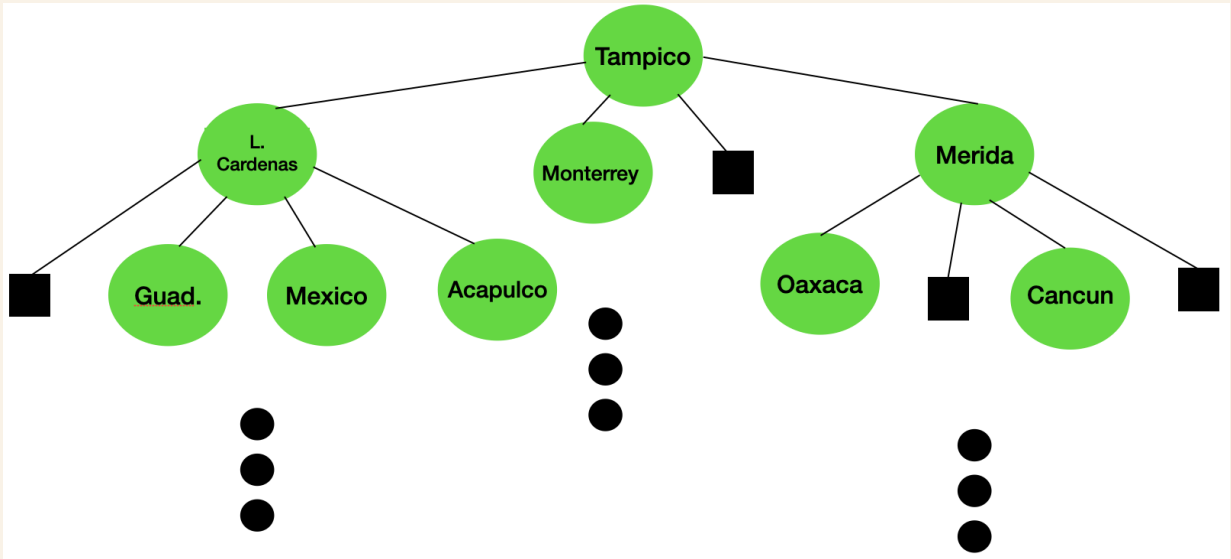
Quad Trees


UNITED STATES
OF AMERICA
Gulf of California
Chihuahua
Nuevo
Laredo
MEXICO
Gulf of Mexico
Monterrey
La Paz
CUBA
PACIFIC
OCEAN
Tampico
Merida
Cancun
Puerto Vallarta
Guadalajara
Veracruz
Manzanillo
MEXICO CITY
Caribbe
Sea
Oaxaca
BELIZE
Acapulco
HONDURA
GUATEMALA


Tampico
L.
Cardenas
Monterrey
Merida

Quad Trees


UNITED STATES
OF AMERICA
Gulf of California
Chihuahua
Nuevo
Laredo
MEXICO
Gulf of Mexico
Monterrey
CUBA
La Paz
PACIFIC
OCEAN
Tampico
Merida
Cancun
Puerto Vallarta
Guadalajara
Veracruz
Manzanillo
Caribbean
Sea
MEXICO CITY
Oaxaca
BELIZE
Acapulco
HONDURAS
GUATEMALA


Tampico
L.
Cardenas
Monterrey
Merida
Guad.
Mexico
Acapulco
Oaxaca
Cancun

Quad Trees

Definition (Bentley & Finkel, 1974)

A quad tree T of size $n \geq 0$ stores a set of n K-dimensional records. The quad tree T is a 2^K-ary tree such that

- either it is empty and $n = 0$, or
- its root stores a record with key x and has 2^K subtrees, each one associated to a K-bitstring $w = w_0 w_1 \ldots w_{K-1} \in \{0,1\}^K$, and the remaining $n - 1$ records are stored in one of these subtrees, let's say T_w, in such a way that $\forall w \in \{0,1\}^K$: T_w is a quad tree, and for any key $y \in T_w$, it holds that $y_j < x_j$ if $w_j = 0$ and $y_j > x_j$ otherwise, $0 \leq j < K$.

Part VI

Multidimensional Data Structures

17 Multidimensional Data Structures: Introduction

18 K-Dimensional Search Trees

19 Quad Trees

20 Analysis of the Cost of Associative Queries

The Cost of Partial Match Searches

- With probability $\frac{s}{K}$ the discriminant will be specified in the query and the algorithm will follow one of the subtrees.
- With probability $\frac{K-s}{K}$ the algorithm will follow the two subtrees

In a random K-d tree the size of the left subtree of a tree of size n is j with equal probability for all j, $0 \leq j < n$. Hence, the expected cost P_n of a partial match in a relaxed K-d tree of size n is

$$P_n = 1 + \frac{s}{K}\frac{1}{n}\sum_{j=0}^{n-1}\left(\frac{j+1}{n+1}P_j + \frac{n-j}{n+1}P_{n-1-j}\right)$$
$$+ \frac{K-s}{K}\frac{1}{n}\sum_{j=0}^{n-1}(P_j + P_{n-1-j})$$

The Cost of Partial Match Searches

The shape function for the recurrence is, with $\rho := s/K$,

$$\omega(z) = 2\rho z + 2(1-\rho)$$

If we compute

$$\mathcal{H} = 1 - \int_0^1 \omega(z)dz = 1 = \rho - 1 < 0$$

We need to find $\alpha \in [0,1]$ such that

$$\int_0^1 z^\alpha \omega(z)dz = 1,$$

that is

$$\frac{2\rho}{\alpha+1} + \frac{2(1-\rho)}{\alpha+2} = 1.$$

The solution of the quadratic equation is

$$\alpha = (\sqrt{9-8\rho} - 1)/2$$

The Cost of Partial Match

Theorem (Duch et al., 1998)

The expected cost P_n (measured as the number of comparisons) of a PM query with s out of K coordinates specified in a random relaxed K-d tree of size n is

$$P_n = \beta n^{\alpha} + \mathcal{O}(1),$$

where

$$\alpha = (\sqrt{9 - 8\rho} - 1)/2$$

$$\beta = \frac{\Gamma(2\alpha + 1)}{(1 - \rho)(\alpha + 1)\Gamma^3(\alpha + 1)}$$

$\Gamma(z)$ *is Euler's Gamma function.*

The Cost of Partial Match

Theorem (Flajolet and Puech, 1986)

The expected cost P_n (measured as the number of comparisons) of a PM query q with s out of K coordinates specified in a standard K-d tree of size n is

$$P_n = \beta_u n^{\alpha} + \mathcal{O}(1),$$

where α is the uniques solution in $[0, 1]$ of

$$(\alpha + 2)^{\rho} \cdot (\alpha + 1)^{1-\rho} = 2,$$

and β_u is a constant depending on the query pattern u ($u[i] =$ specified/non-specificied)

The Cost of Partial Match

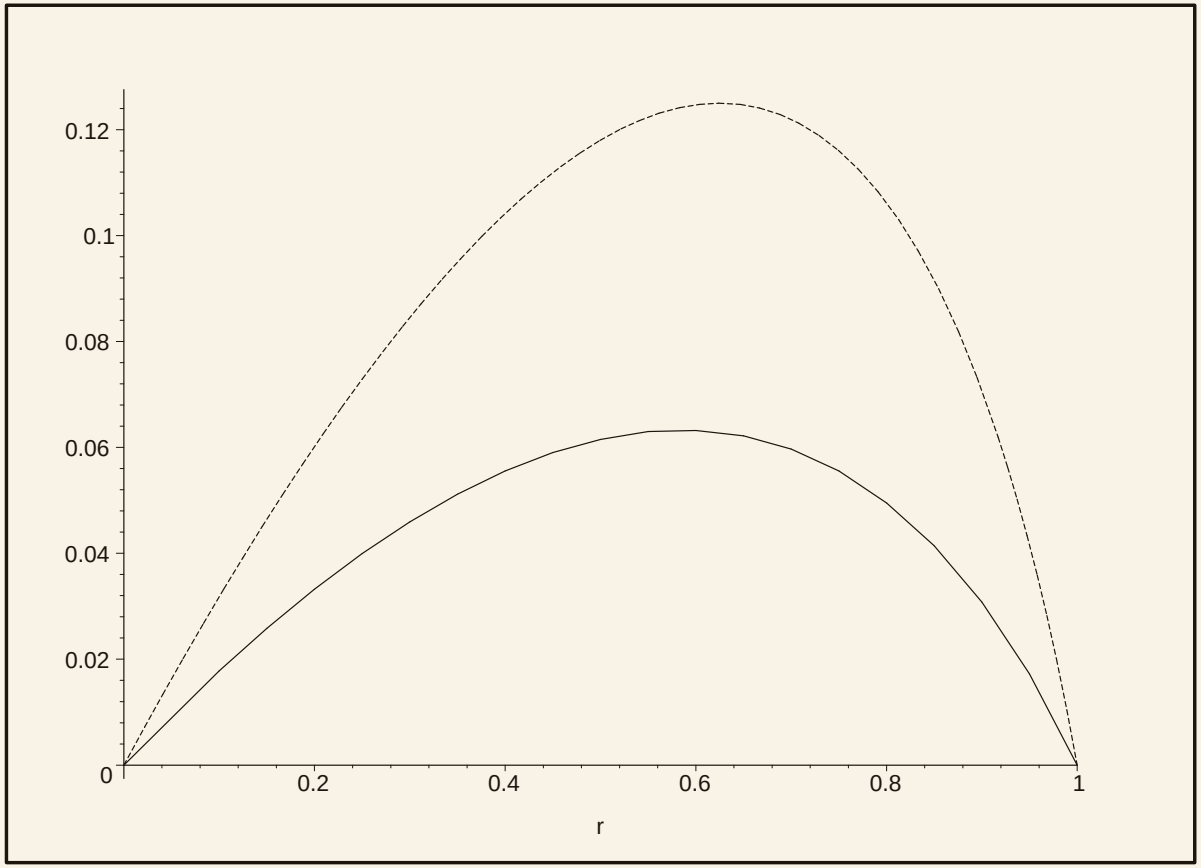
The exponent α in standard K-d trees is samller than the α for relaxed K-d trees. If we consider the excess

$$\vartheta = \alpha - (1 - \rho)$$

it is very close to 0 (and never greater than 0.07) for satdnard K-d trees. The excess for relaxed K-d trees is not very big but can be as much as 0.12.
Squarish K-d trees achieve the ultimate optimal expected performance as they have excess = 0, thanks to their more (heuristically) balanced space partition, induced by the choice of discriminants.

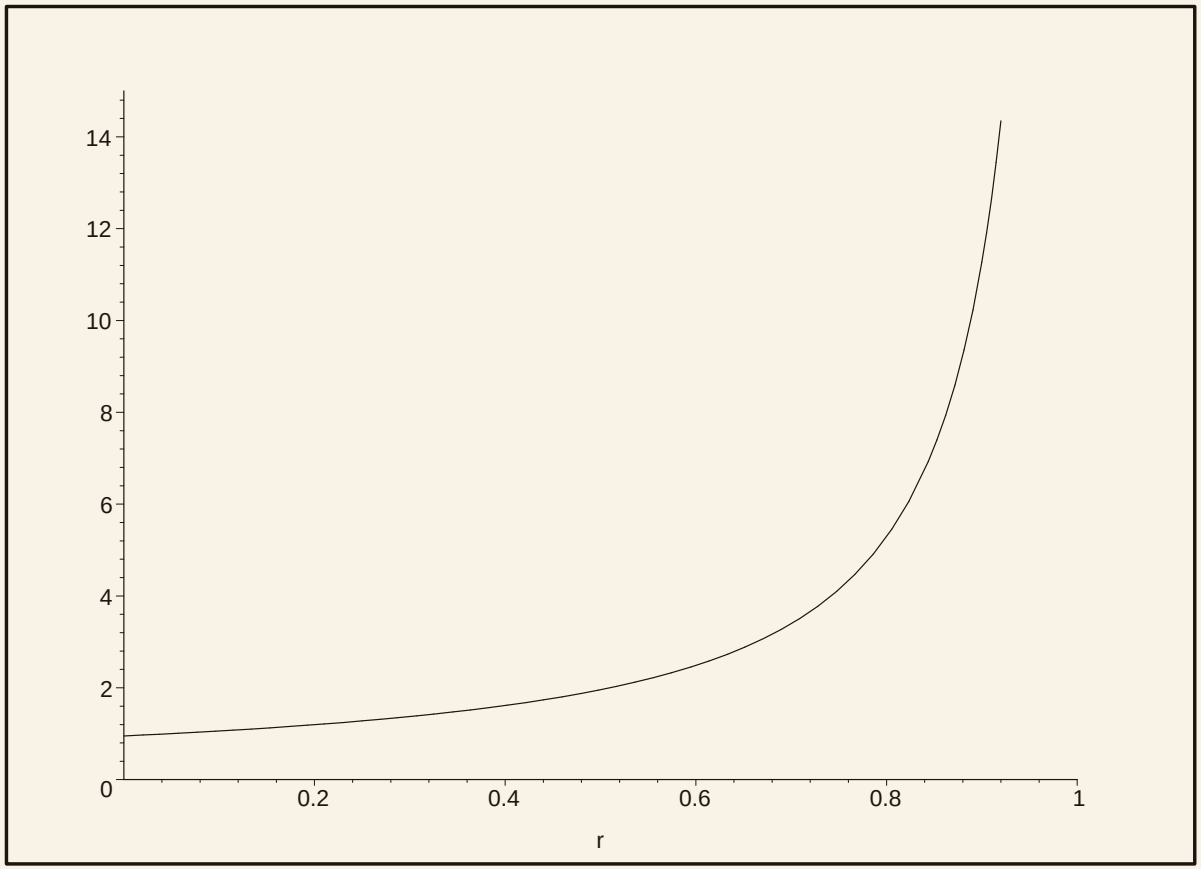
The Cost of Partial Match Searches

Excess of the exponent α with respect to $1 - s/K$. Solid line: standard K-d trees. Dotted line: relaxed K-d trees



The Cost of Partial Match Searches

Plot of the coefficient β for relaxed K-d trees



The Cost of Orthogonal Range Queries

The expected cost of a orthogonal range query Q with side lengths $\Delta_0 = u_0 - \ell_0$, $\Delta_1 = u_1 - \ell_1$, ... is

$$v_K(\Delta) \cdot n + v_{K-1}(\Delta) \cdot n^{\alpha(1/K)} + v_{K-2}(\Delta) \cdot n^{\alpha(2/K)} + \\ \cdots + v_1 \cdot n^{\alpha((K-1)/K)} + 2v_0 \ln n + \mathcal{O}(1)$$

where v_K is, roughly, the "volume" of the K-dimensional boundary of Q. For example, if $K = 2$ then $v_2 = \Delta_0 \cdot \Delta_1$, $v_1 = \Delta_0(1 - \Delta_1) + \Delta_1(1 - \Delta_0) \approx \Delta_0 + \Delta_1$, $v_0 = (1 - \Delta_0) \cdot (1 - \Delta_1) = 1 - \Delta_0 - \Delta_1 + \Delta_0 \cdot \Delta_1$. Intuition: the

orthogonal range search behaves in a region of "'volume" v_j exactly as a partial match with $K - j$ specified coordinates.

The Cost of Neareast Neighbors Queries

The expected cost of a nearest neighbors query is

$$S_n = \Theta(n^{\vartheta} + \log n)$$

where $\vartheta = \max_j\{\alpha(j/K) - 1 + j/K\}$ is the maximum excess.
Intuition: the nearest neighbor search behaves like an

orthogonal range search where the query has side lenghts $\Delta_i = \Theta(n^{1/K})$.

To learn more

[1] J. L. Bentley.
Multidimensional binary search trees used for associative retrieval.
Communications of the ACM, 18(9):509–517, 1975.

[2] J. L. Bentley and R. A. Finkel.
Quad trees: A data structure for retrieval on composite keys.
Acta Informatica, 4:1–9, 1974.

[3] H. H. Chern and H. K. Hwang.
Partial match queries in random k-d trees.
SIAM J. on Computing, 35(6):1440–1466, 2006.

[4] H. H. Chern and H. K. Hwang.
Partial match queries in random quad trees.
SIAM Journal on Computing, 32(4):904–915, 2003.

To learn more (2)

[5] L. Devroye.
Branching processes in the analysis of the height of trees.
Acta Informatica, 24:277–298, 1987.

[6] L. Devroye and L. Laforest.
An analysis of random d-dimensional quadtrees.
SIAM Journal on Computing, 19(5):821–832, 1990.

[7] A. Duch.
Randomized insertion and deletion in point quad trees.
In *Int. Symposium on Algorithms and Computation (ISAAC)*, LNCS. Springer–Verlag, 2004.

[8] A. Duch, V. Estivill-Castro, and C. Martínez.
Randomized K-dimensional binary search trees.
In K.-Y. Chwa and O. H. Ibarra, editors, *Int. Symposium on Algorithms and Computation (ISAAC'98)*, volume 1533 of *LNCS*, pages 199–208. Springer-Verlag, 1998.

To learn more (3)

[9] A. Duch and C. Martínez.
On the average performance of orthogonal range search in multidimensional data structures.
Journal of Algorithms, 44(1):226–245, 2002.

[10] A. Duch and C. Martínez.
Updating relaxed k-d trees.
ACM Transactions on Algorithms (TALG), 6(1):1–24, 2009.

[11] Ph. Flajolet, G. Gonnet, C. Puech, and J. M. Robson.
Analytic variations on quad trees.
Algorithmica, 10:473–500, 1993.

[12] Ph. Flajolet and C. Puech.
Partial match retrieval of multidimensional data.
Journal of the ACM, 33(2):371–407, 1986.

To learn more (4)

[13] C. Martínez, A. Panholzer, and H. Prodinger.
Partial match queries in relaxed multidimensional search trees.
Algorithmica, 29(1–2):181–204, 2001.

[14] R. Neininger.
Asymptotic distributions for partial match queries in K-d trees.
Random Structures and Algorithms, 17(3–4):403–4027, 2000.

[15] R. L. Rivest.
Partial-match retrieval algorithms.
SIAM Journal on Computing, 5(1):19–50, 1976.

[16] H. Samet.
Deletion in two-dimensional quad-trees.
Communications of the ACM, 23(12):703–710, 1980.

General References

[1] R. Sedgewick & Ph. Flajolet
An Introduction to the Analysis of Algorithms.
Addison-Wesley, 2nd edition, 2013.

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

[3] D.P. Mehta and S. Sahni, editors.
Handbook of Data Structures and Applications.
Chapman & Hall, CRC, 2005.

General References (2)

[4] T. Cormen, C. Leiserson, R. Rivest & C. Stein.
Introduction to Algorithms.
The MIT Press, 3rd edition, 2009.

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

[6] M. Mitzenmacher and E. Upfal.
Probability and computing: Randomized algorithms and probabilistic analysis.
Cambridge University Press, 2005.

General References (3)

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

[8] R. Sedgewick and K. Wayne.
Algorithms.
Addison-Wesley, 4th edition, 2011.

[9] D. Gusfield.
Algorithms on String, Trees, and Sequences.
Cambridge Univ. Press, 1997.

General References (3)

[10] H. Samet.
Foundations of Multidimensional and Metric Data Structures.
Morgan Kaufmann, 2006.