Advanced Data Structures

Conrado Martínez
U. Politècnica Catalunya

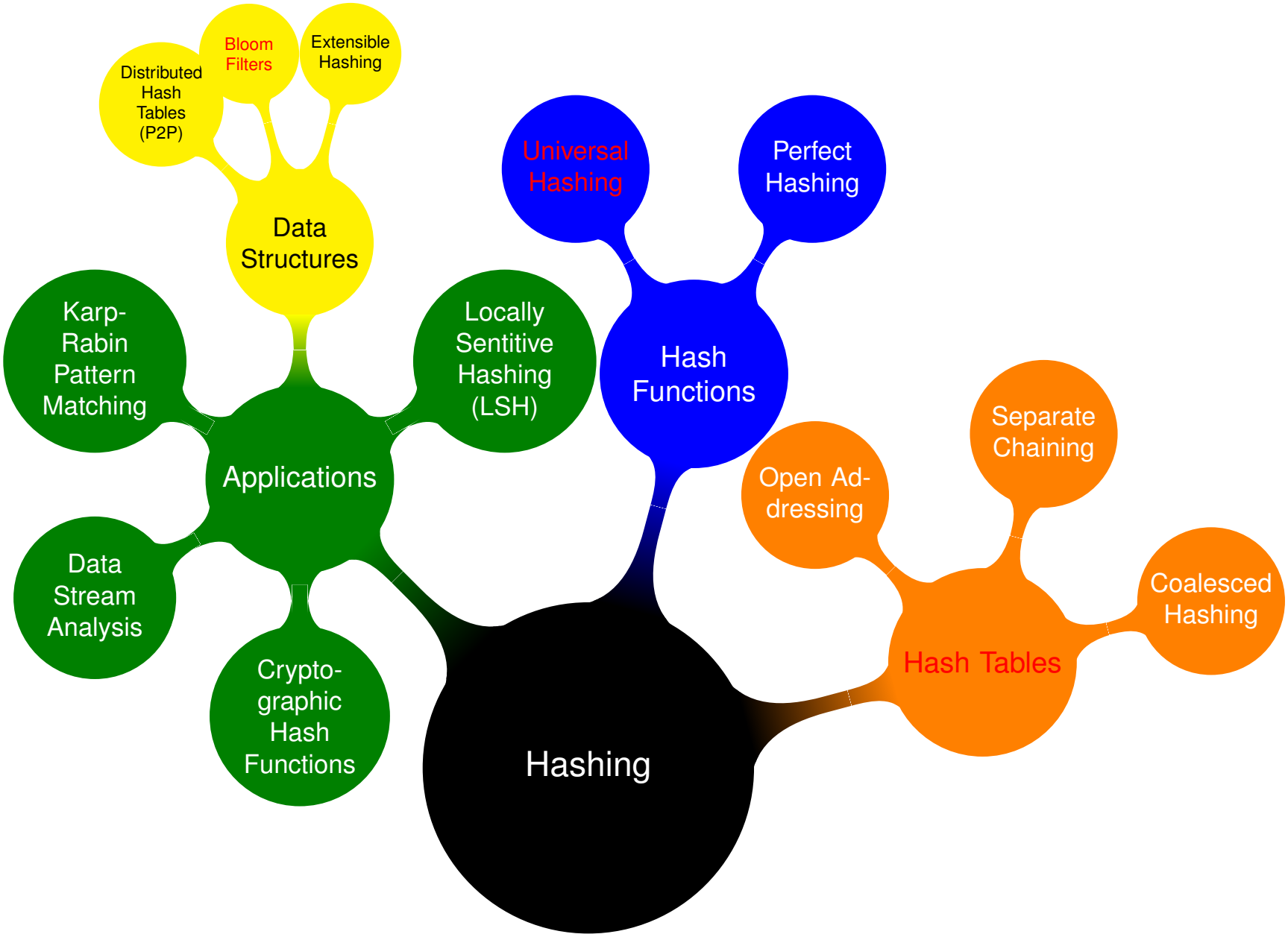
March 22, 2020

Part I

Hashing

1. Universal Hashing
2. Hash Tables
3. Bloom Filters

Hashing


Hashing
Data Structures
Bloom Filters
Extensible Hashing
Distributed Hash Tables (P2P)
Applications
Karp-Rabin Pattern Matching
Locally Sentitive Hashing (LSH)
Data Stream Analysis
Crypto-graphic Hash Functions
Hash Functions
Universal Hashing
Perfect Hashing
Hash Tables
Separate Chaining
Open Ad-dressing
Coalesced Hashing

Hashing

A hash function h maps the elements (keys) of a given domain (or *universe*) $\mathcal{U}$ in a finite range $0..M-1$.
Hash functions must:

1. Be easy and fast to compute
2. Be represented with little memory
3. Spread the universe as evenly as possible

$$\mathcal{U}_i = \{x \in \mathcal{U} \mid h(x) = i\}, \qquad 0 \leq i < M$$

$$|\mathcal{U}_i| \approx \frac{|\mathcal{U}|}{M}$$

4. Give very different hash values to “similar” keys

Part I

Hashing

Universal Hashing

M.N. Wegman

Definition

A class

$$\mathcal{H} = \{h \,|\, h : \mathcal{U} \to [0..M-1]\}$$

of hash functions is universal iff, for all $x, y \in \mathcal{U}$ with $x \neq y$ we have

$$\mathbb{P}[h(x) = h(y)] \leq \frac{1}{M},$$

where h is a hash function randomly drawn from $\mathcal{H}$

Universal Hashing

A stronger property is pairwise independence (a.k.a. strong universality). A class is strongly universal iff, for all $x, y \in \mathcal{U}$ with $x \neq y$ and any two values $i, j \in [0..M-1]$

$$\mathbb{P}[h(x) = i \wedge h(y) = j] = \frac{1}{M^2}$$

Strong universality implies universality; moreover

$$\mathbb{P}[h(x) = i] = \frac{1}{M}$$

for any x and i.

Universal Hashing

Let $\mathcal{H}$ be a universal class and $h \in \mathcal{H}$ drawn at random. For any fixed set of n keys $S \subseteq \mathcal{U}$ we have the following properties:

1. For any $x \in S$, the expected number of elements in S that hash to $h(x)$ is n/M.
2. The expected number of collisions is $O(n^2/M)$. If $M = \Theta(n)$ then the expected number of collisions is $O(n)$.

Universal Hashing

The big questions are:

- Are there universal classes? Strongly universal classes?
- If so, how complicated are its members? How much effort does it take to compute and represent the functions in the class?

Universal Hashing

In 1977 Carter and Wegman introduced the concept of universal class of hash functions and gave the first construction.
Put the universe $\mathcal{U}$ into one-to-one correspondence with $[0..U-1]$ ($U = |\mathcal{U}|$) and let p be a prime $\geq U$.
The class

$$\mathcal{H} = \{h_{a,b} \,|\, 0 < a < p, 0 \leq b < p\}$$

is (strongly) universal, with

$$h_{a,b}(x) = ((ax + b) \mod p) \mod M$$

Universal Hashing

The ingredients we need are thus a BIG prime p; picking a hash function at random from $\mathcal{H}$ amounts to choosing two integers a and b at random.
Let $r = \lceil \log_2(U+1) \rceil$. The prime number p and the numbers a and b will need roughly r bits each. For instance, if our universe are ASCII strings of length at most 30, $U \approx 256^{30}$ and $r \approx 240$ bits; these are huge numbers and a fast primality test is a must to have a practical scheme.

Universal Hashing

Suppose that $h_{a,b}$ has been picked at random and let x and y be two distinct keys that collide

$$h_{a,b}(x) = h_{a,b}(y)$$

Therefore

$$ax + b \equiv ay + b + \lambda \cdot M \pmod{p}$$

for some integer $\lambda \geq 0$, $\lambda \leq p/M$.

Universal Hashing

Since $x \neq y$, $x - y \neq 0$, hence $x - y$ has an inverse multiplicative in the ring $\mathbb{Z}_p$, denote it $(x-y)^{-1}$. Hence

$$\begin{aligned} ax &\equiv ay + \lambda \cdot M \pmod{p} \\ a(x-y) &\equiv \lambda \cdot M \pmod{p} \\ a &\equiv (x-y)^{-1} \cdot \lambda \cdot M \pmod{p} \end{aligned}$$

Universal Hashing

There are $p-1$ possible choices for a and $\lfloor p/M \rfloor$ possible values for λ; hence the probability of collision is

$$\leq \frac{\lfloor p/M \rfloor}{p-1} \approx \frac{1}{M}$$

for sufficiently large p.

Universal Hashing

Notice that b plays no rôle in the universality of the family. We might have choosen $b = 0$ or any other convenient fixed value. However, picking b at random makes the class strongly universal.

Universal Hashing

To learn more:


L. Carter and M.N. Wegman.
Universal Classes of Hash Functions.
Journal of Computer and System Sciences, 18 (2): 143–154, 1979.


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


O. Kaser and D. Lemire.
Strongly universal string hashing is fast.
Computer Journal (published on-line in 2013)

Part I

Hashing

Hash Tables

A hash table (cat: *taula de dispersió*, 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 Tables

```
template <typename T> class Hash {
public:
  int operator()(const T& x) const ;
};

template <typename Key, typename Value,
          template <typename> class HashFunct = Hash>
class Dictionary {
public:
   ...
private:
   struct node {
       Key _k;
       Value _v;
       ...
   };
   nat _M; // capacity of the table
   nat _n; // number of elements (size) of the table
   double _alpha_max; // max. load factor
   HashFunct<Key> h;

   // open addressing
   vector<node> _Thash; // an array with pairs <key,value>

   // separate chaining
   // vector<list<node>> _Thash; // an array of lists of synonyms

   int hash(const Key& k)  {
        return h(k) % _M;
   }
};
```

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, $0 \leq i < M$

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

Hash Functions

In our implementation, the class `Hash<T>` overloads operator `()` so that for an object `h` of the class `Hash<T>`, `h(x)` is the result of "applying" h to the object `x` of class `T`. The operation returns a positive integer.

The private method `hash` in class Dictionary computes

```
h(x) % _M
```

to obtain a valid position into the table, an index between 0 and `_M - 1`.

Hash Functions

```
// specialization of the template for T = string
template <> class Hash<string> {
public:
  int operator()(const string& x) const  {

      int s = 0;
      for (int i = 0; i < x.length(); ++i)
          s = s * 37 + x[i];
      return s;
};

// specialization of the template for T = int
template <> class Hash<int> {
static long const MULT = 31415926;
public:
  int operator()(const int& x) const  {

      long y = ((x * x * MULT) << 20) >> 4;
      return y;
  }
};
```

Other sophisticated hash functions use weighted sums or non-linear transformations (e.g., they square the number represented by the k central bits of the key).

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

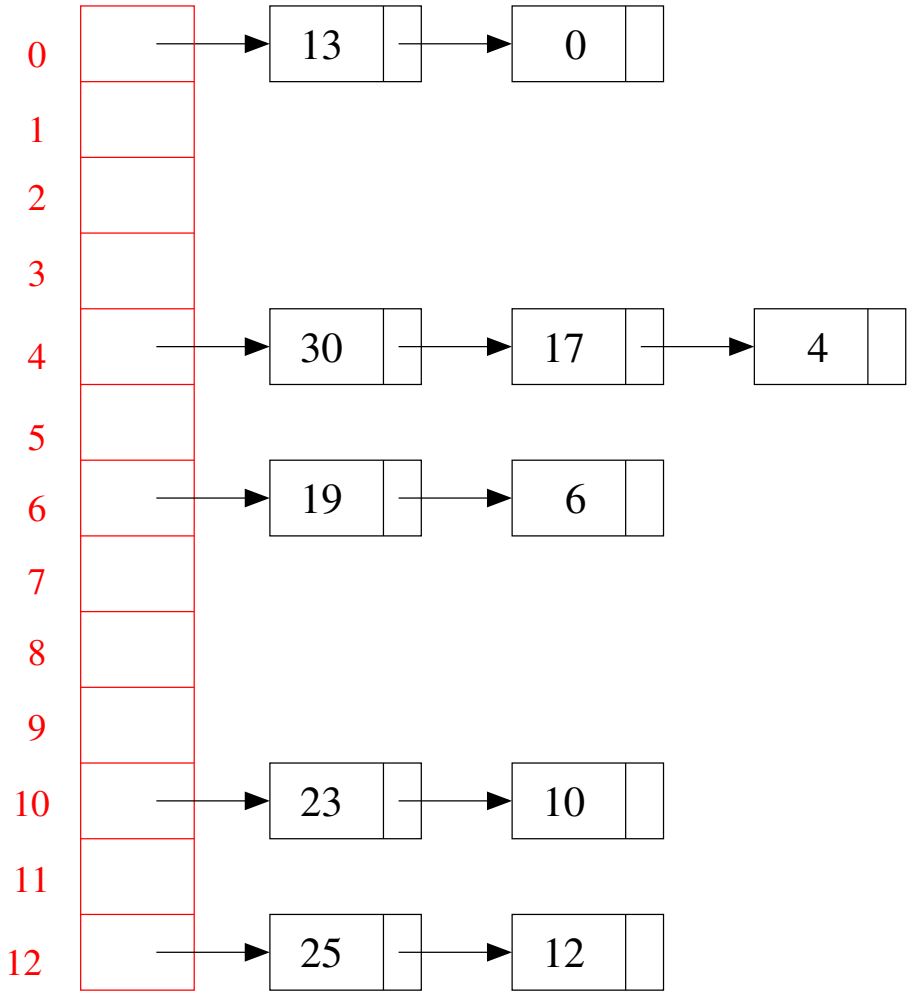
In separate chaining, each slot in the hash table has a pointer to a linked list of synonyms.

```
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

   list<node>::const_iterator lookup_sep_chain(const Key& k, int i) const ;
   void insert_sep_chain(const Key& k,
           const Value& v);
   void remove_sep_chain(const Key& k) ;
};
```

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

```
template <typename Key, typename Value,
          template <typename> class HashFunct>
void Dictionary<Key,Value,HashFunct>::insert(const Key& k,
     const Value& v) {
   insert_sep_chain(k, v);
   if (_n / _M > _alpha_max)
      // the current load factor is too large, raise here an exception or
      // resize the table and rehash
}

template <typename Key, typename Value,
          template <typename> class HashFunct>
void Dictionary<Key,Value,HashFunct>::insert_sep_chain(
           const Key& k, const Value& v) {
   int i = hash(k);
   list<node>::const_iterator p = lookup_sep_chain(k,i);
   // insert as first item in the list
   // if not present
   if (p == _thash[i].end()) {

      _thash[i].push_back(node(k, v));
      ++_n;
   }
   else
      p -> _v = v;
}
```

Separate Chaining

```
template <typename Key, typename Value,
          template <typename> class HashFunct>
void Dictionary<Key,Value,HashFunct>::lookup(const Key& k,
   bool& exists, Value& v) const {
   int i = hash(k);
   list<node>::const_iterator p = lookup_sep_chain(k, i);
   if (p == _thash[i].end())
     exists = false;
   else {
     exists = true;
     v = p -> _v;
  }
}

template <typename Key, typename Value,
          template <typename> class HashFunct>
list<Dictionary<Key,Value,HashFunct>::node >::const_iterator
   Dictionary<Key,Value,HashFunct>::lookup_sep_chain(const Key& k,
                                                     int i) const  {

   list<node>::const_iterator p = _Thash[i].begin();
   // sequential search in the i-th list of synonyms
   while (p != _thash[i].end() and p -> _k != k)
      ++p;

   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.

The Cost of Separate Chaining

- $L_n^{(i)}$: the number of elements hashing to the i-th list, $0 \leq i < M$, after the insertion of n items.
- Standard assumption: the probability that the j-th inserted item hashes to position i, $0 \leq i < M$, is $1/M$
- The random variables $L_n^{(i)}$, $0 \leq i < M$, are **not** independent, but they are identically distributed
- Set $L_n := L_n^{(0)}$. Let $Y_j = 1$ iff the j-th inserted item goes to list 0, and $Y_j = 0$ otherwise.

$$
\begin{aligned}
L_n &= Y_1 + \ldots + Y_n \\
\mathbb{E}[L_n] &= \mathbb{E}[Y_1 + \ldots + Y_n] = \mathbb{E}[Y_1] + \ldots + \mathbb{E}[Y_n] \\
&= 1/M + \ldots + 1/M = n/M = \alpha
\end{aligned}
$$

The Cost of Separate Chaining

- Cost of unsuccessful search $U_n \approx$ cost of insertion of the $(n+1)$-th item

$$\begin{aligned}
\mathbb{E}[U_n] &= \sum_{0 \le i < M} \mathbb{E}[U_n | \text{search in list } i] \cdot \mathbb{P}[\text{search in list } i] \\
&= \frac{1}{M} \sum_{0 \le i < M} \mathbb{E}[U_n | \text{search in list } i] = \frac{1}{M} \sum_{0 \le i < M} (1 + \mathbb{E}\left[L_n^{(i)}\right]) \\
&= 1 + \alpha
\end{aligned}$$

The Cost of Separate Chaining

- Cost of succesful search of a random item $S_n \approx$ cost of deletion of a random item

$$\mathbb{E}[S_n] = \sum_{0 \le i < M : L_n^{(i)} > 0} \mathbb{E}[S_n | \text{search in list } i] \cdot \mathbb{P}[\text{search in list } i]$$

$$= \sum_{0 \le i < M : L_n^{(i)} > 0} \left(\sum_{\ell > 0} \frac{\ell + 1}{2} \mathbb{P}\left[L_n^{(i)} = \ell \right] \right) \cdot \frac{L_n^{(i)}}{n}$$

$$= \sum_{0 \le i < M : L_n^{(i)} > 0} \frac{1 + \alpha}{2} \frac{L_n^{(i)}}{n}$$

$$= \frac{1 + \alpha}{2} \sum_{0 \le i < M : L_n^{(i)} > 0} \frac{L_n^{(i)}}{n} = \frac{1 + \alpha}{2}$$

The Cost of Separate Chaining

- The Poisson model: in order to avoid the dependence between $L_n^{(i)}$ we can consider a Poisson random model in which "balls" (items) are thrown into "bins" (slots in the hash table) at a rate α, then the length of each list $\mathcal{L}_i \sim \text{Poisson}(\alpha)$ is independent of all other
- We have, for instance, $\mathbb{E}[\mathcal{L}_i] = \alpha = \mathbb{E}\left[L_n^{(i)}\right]$
- In general we can make our computations in the easier Poisson model then (rigorously) transfer these results to the "exact model"

The Cost of Separate Chaining

- Let $L_n^* = \max\{L_n^{(0)}, \ldots, L_n^{(M-1)}\}$. This random variable gives the worst-case cost of search, insertions and deletions
- An important identity for positive discrete r.v.

$$\mathbb{E}[X] = \sum_{k \geq 0} k\, \mathbb{P}[X = k] = \sum_{k \geq 0} \mathbb{P}[X > k]$$

The Cost of Separate Chaining

- In th Poisson model, we have M i.i.d. Poisson r.v. $\mathcal{L}_i$, all with parameter $\alpha = n/M$, giving the length of the i-th list, $0 \leq i < M$
- Then for $\mathcal{L}^* = \max_{0 \leq i < M} \{\mathcal{L}_i\}$ we have

$$\mathbb{P}[\mathcal{L}^* \leq k] = \prod_i \mathbb{P}[\mathcal{L}_i \leq k]$$

$$= \left(\sum_{0 \leq j \leq k} \frac{\alpha^j e^{-\alpha}}{j!} \right)^M$$

and

$$\mathbb{E}[\mathcal{L}^*] = \sum_{k>0} \left(1 - \left(\sum_{0 \leq j \leq k} \frac{\alpha^j e^{-\alpha}}{j!} \right)^M \right)$$

- But this path leads us nowhere.

The Cost of Separate Chaining

We will try a different way:

- Compute (or give useful bounds) for the median of $\mathcal{L}^*$, i.e., the value of j such that $\mathbb{P}[\mathcal{L}^* \leq j] = 1/2$
- Show that the expectation (mean) of $\mathcal{L}^*$ is close to its median, namely we show that

$$\frac{\mathbb{E}[\mathcal{L}^*]}{j} \to 1$$

if n is large enough (and $\alpha = n/M$ is kept constant)

The Cost of Separate Chaining

For which value j do we have

$$\mathbb{P}[\mathcal{L}^* \leq j] = \prod_i \mathbb{P}[\mathcal{L}_i \leq j] = \left(\sum_{0 \leq k \leq j} \frac{\alpha^k e^{-\alpha}}{k!} \right)^M = 1/2?$$

The summation

$$\sum_{0 \leq k \leq j} \frac{\alpha^k}{k!} \approx e^\alpha - \frac{\alpha^{j+1}}{(j+1)!},$$

hence

$$\mathbb{P}[\mathcal{L}^* \leq j] \approx \left(1 - \frac{\alpha^{j+1}}{(j+1)!} e^{-\alpha} \right)^M$$

The Cost of Separate Chaining

We want j such that

$$\left(1 - \frac{\alpha^{j+1}}{(j+1)!}e^{-\alpha}\right)^{M} = \frac{1}{2}$$

Taking natural logs on both sides

$$M \ln\left(1 - \frac{\alpha^{j+1}e^{-\alpha}}{(j+1)!}\right) = -\ln 2$$

Since $\ln(1-x) \sim x + x^2/2 + O(x^3)$

$$\left(\frac{\alpha^{j+1}e^{-\alpha}}{(j+1)!} + \ldots\right) \approx \frac{-\ln 2}{M}$$

The Cost of Separate Chaining

Hence

$$\frac{\alpha^{j+1}e^{-\alpha}}{(j+1)!} = \Theta(1/M)$$

- $\alpha \to 1$ implies $(j+1)! = \frac{M}{e \ln 2}$, that is, $j = \Gamma^{(-1)}(M/(e \ln 2))$.
- For $\alpha < 1$ we also have $j = \Theta(\Gamma^{(-1)}(M))$.
- Since $\Gamma^{(-1)}(n) \sim \ln n / \ln \ln n$, and $n = \alpha M$ we have that the median j of $\mathcal{L}^*$ is $j = \Theta(\log n / \log \log n)$.

(see next slide for definitions and remarks)

The Cost of Separate Chaining

Note:

- $\Gamma^{(-1)} =$ inverse of the Gamma function $\Gamma(z)$
- Γ generalizes factorials to complex numbers ($\Gamma(z+1) = z\Gamma(z)$).
- Since $\ln n! \sim n \ln n - n + O(1)$ (Stirling's approximation) we can easily prove $\Gamma^{(-1)}(n) \sim \ln n / \ln \ln n$ if $n \to \infty$.

For the rest of the proof (showing that the expected value of $\mathcal{L}^*$ has the same order of growth as its median) you can check Section 2.2 in [Gon81]

d-way Chaining

Azar, Broder, Karlin and Upfal [ABKU99] have shown the following important result

Theorem

Suppose n balls are sequentailly placed in $m \geq n$ bins, so that for each ball $d \geq 2$ random bins are chosen and the ball is placed in the least full bin —with ties broken arbitrarily. Then with high probability, as $n \to \infty$, the fullest bin contains

$$(1 + o(1)) \ln \ln n / \ln d + \Theta(m/n)$$

balls.

d-way Chaining

This result has many applications, not only for data structures design. In the context of hashing, the hashing scheme suggested by this result is very straightforward:

- To insert an item x, compute $i = h_1(x)$ and $j = h_2(x)$ with two (or in general d) independent hash functions and insert x in the synonym list which is shorter, i.e., list i if $L_i \leq L_j$ and vice-versa
- To search (or delete) an item x compute $i = h_1(x)$ and $j = h_2(x)$ and search for x in both lists (why? why not only the shortest?) Clearly if x is present it must be in one of these two lists

d-way Chaining

In d-way chaining we basically multiply by d the expected costs of all operations, as compared to separate chaining, as we need to evaluate d hash functions and search in that many lists. We also need to keep the size of each list.
It is very easy to show that with d-way chaining the expected length of each list is $\alpha = n/m$ like in ordinary separate chaining. However:

- the variance of each $L_n^{(i)}$ is smaller than in separate chaining
- the expected longest list has length $\Theta(\log \log n)$, a huge improvement w.r.t. the $\Theta(\log n / \log \log n)$ in separate chaining

For those interested in the details (in particular the proof of the result) check [ABKU99].

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.

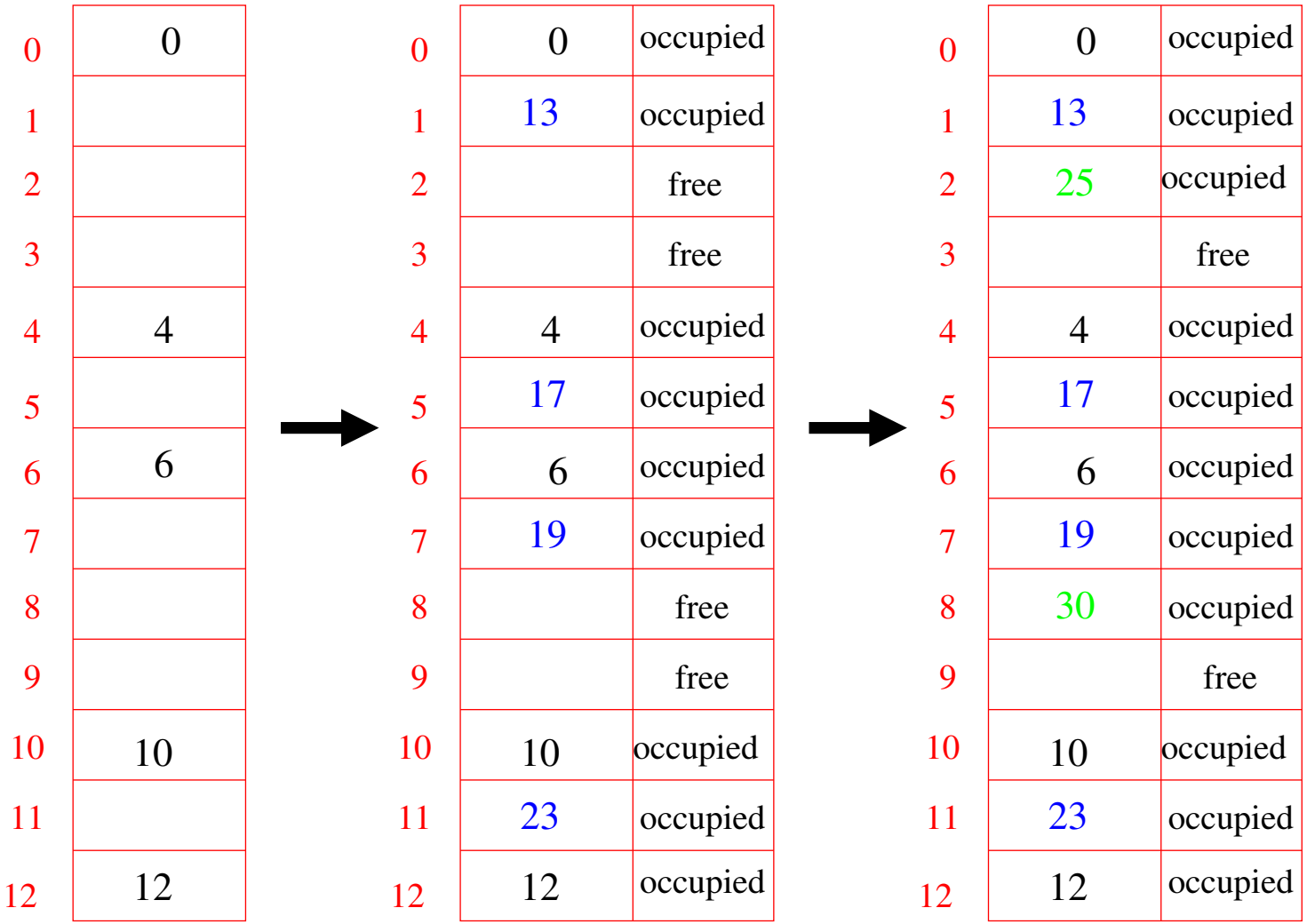
Linear Probing

```
template <typename Key, typename Value,
          template <typename> class HashFunct = Hash>

class Dictionary {
   ...
private:
struct node {
   Key _k;
   Value _v;
   bool _free;
   // constructor for class node
   node(const Key& k, const Value& v, bool free = true);
};
vector<node>  _Thash;  // array of nodes
int _M;         // capacity of the table
int _n;         // number of elements
double _alpha_max;  // max. load factor (must be < 1)


int lookup_linear_probing(const Key& k) const ;
void insert_linear_probing(const Key& k,
        const Value& v);
void remove_linear_probing(const Key& k) ;
};
```

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

```
template <typename Key, typename Value,
          template <typename> class HashFunct>
int Dictionary<Key,Value,HashFunct>::lookup(
        const Key& k,
        bool& exists, Value& v) const {
   int i = lookup_linear_probing(k);
   if (not _Thash[i]._free and _Thash[i]._k == k) {
      exists = true; v = _Thash[i]. _v;
   }
   else
      exists = false;
}

template <typename Key, typename Value,
          template <typename> class HashFunct>
int Dictionary<Key,Value,HashFunct>::lookup_linear_probing(
        const Key& k) const  {
   int i = hash(k);
   int visited = 0;  // this is only necessary if
                     // _n == _M, otherwise there is at least
                     // a free position
   while (not _Thash[i]._free and _Thash[i]._k != k
          and visited < _M) {
      ++visited;
      i = (i + 1) % _M;
   }
   return i;
}
```

Deletions in Open Addressing

There is no general solution for true deletions in open addressing tables. It is not enough to mark the position of the element to be removed as “free”, since later searches might report as not present some element which is stored in the table.

The general technique that can be used is lazy deletions. Each slot can be free, occupied or **deleted**. Deleted slots can be used to store there a new element, but they are not free and searches must pass them over and continue.

Deletions in Linear Probing

For linear probing, we can do true deletions. The deletion algorithm must continue probing the positions after the removed element, and moving to the emptied slot any element whose hash address is equal (or smaller in the cyclic order) to the address of the emptied slot. Moving an element creates a new emptied slot, and the procedure is repeated until an empty slot is found. In our implementation we will use the function $\texttt{displ}(j, i)$ which gives us the distance between j e i in the cyclic order: if $j > i$ we must turn around position _$M - 1$ and go back to position 0.

```
int displ(j, i, M) {
  if (i >= j)
     return i - j;
  else
     return M + (i - j);
}
```

Deletions in Linear Probing

```
// we assume _n < _M

template <typename Key, typename Value,
          template <typename> class HashFunct>
int Dictionary<Key,Value,HashFunct>::remove_linear_probing(
       const Key& k) const  {
    int i = lookup_linear_probing(k);
    if (not _Thash[i]._free) {
       // _Thash[i] is the element to remove
       int free = i; i = (i + 1) % _M; int d = 1;
       while (not _Thash[i]._free) {
             int i_home = hash(_Thash[i]._k);
             if (displ(i_home, i, _M) >= d) {
                _Thash[free] = _Thash[i]; free = i; d = 0;
             }
             i = (i + 1) % _M;  ++d;
       }
       _Thash[free]._free = true; --_n;
    }
}
```

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

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

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 $H(j,x) = f(j)$ (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 (see next slides), 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

$$U_n = 1 \times (1-\alpha) + 2 \times \alpha \cdot (1-\alpha) + 3 \times \alpha^2 \cdot (1-\alpha)$$

$$= \sum_{k>0} k\alpha^{k-1} \cdot (1-\alpha) = (1-\alpha) \sum_{k>0} \frac{d(\alpha^k)}{d\alpha}$$

$$= (1-\alpha) \frac{d}{d\alpha} \sum_{k>0} \alpha^k = \frac{1}{1-\alpha}.$$

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.

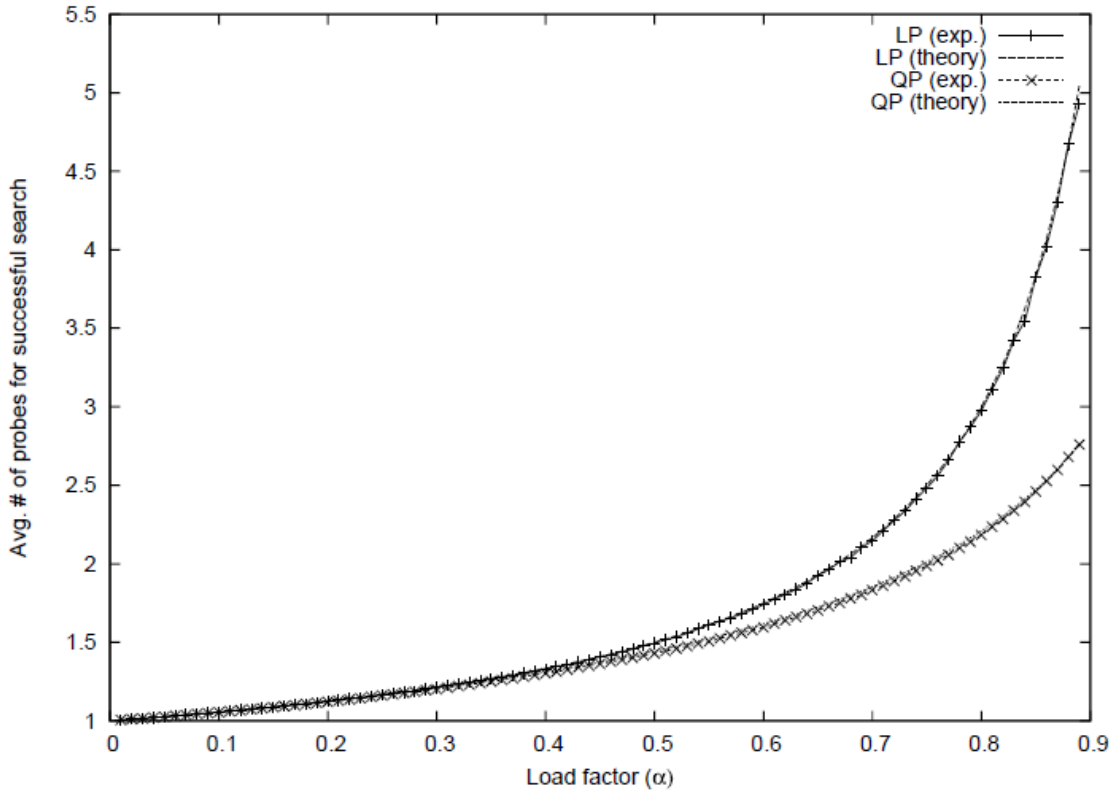
- The average cost of unsuccessful search is

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

- 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

Insertion Policies in Open Addresing

- The standard insertion policuy in case of a collision is FCFS (first-come-first-served): the item x that occupies an slot remains there, and the colliding item y continues with its probe sequence
- But other policies are also possible and have been proposed in the literature:
  - LCFS (last-come-first-served): y kicks out x, x continues with its probe sequence
  - Ordered hashing: If $x \leq y$, x remains and y continues, and the other way around otherwise
  - Robin Hood: the item farthest away from its home location stays, the other continues, ties are resolved arbitrarily

Insertion Policies in Open Addresing

All these strategies lead to the same average successful search cost as FCFS, but:

- the variance is significantly reduced
- most importantly, the expected worst case is reduced from $\Theta(\log n)$ to $\Theta(\log \log n)$

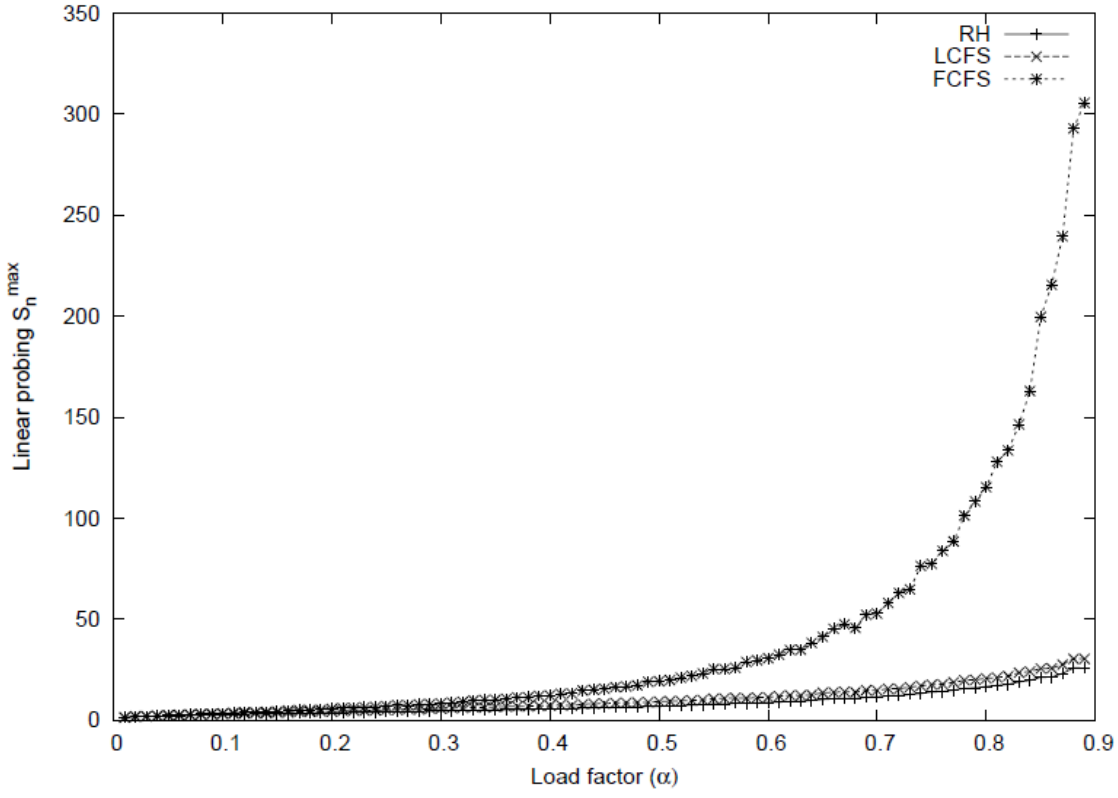
Think for instance in linear probing. The length of clusters will be the same for FCFS, LCFS, OH and RH, and the **sum of the distances** of all items to their respective home locations also changes but the distribution of distances to home location will vastly differ—like in the two sums below

$$1 + 3 + 7 = 3 + 4 + 4$$

Insertion Policies in Open Addresing

- Both ordered hashing and Robin Hood have the very nice feature that, given a set X of items to be inserted the final table is always the same, irrespective of the order in which item are inserted.
- This invariance with respect the prder of insertions notably simplifies some analysis.
- The unsuccessful search cost in OH and RH can be greatly improved; no need to continue until an empty slot is found (why?)
- But the insertion cost is the same, we either stop at an empty location or we kick out some item to insert the new item, but then we must continue

Insertion Policies in Open Addresing



Comparison of the maximum expected cost of a successful search in linear probing with FCFS (standard), LCFS and RH

Cuckoo Hashing

Hash Tables

To learn more:


Y. Azar, A. Broder, A. Karlin and E. Upfal.
Ballanced Allocations
SIAM J. Computing, 29(1):180-200, 1999.


Gaston H. Gonnet
Expected Length of the Longest Probe Sequence in Hash Code Searching
Journal of the ACM, 28 (2): 289–304, 1981.


Leo J. Guibas
The Analysis of Hashing Techniques That Exhibit k-ary Clustering
Journal of the ACM, 25 (4): 544–555, 1978.

Hash Tables

To learn more:

Donald E. Knuth
The Art of Computer Programming. Volume 3: Sorting and Searching (2nd ed)
Addison-Wesley, Reading, MA, 1998.

Rasmus Pagh and Flemming F. Rodler
Cuckoo Hashing
Journal of Algorithms 51:122-144, 2004

Part I

Hashing

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

```
template <class T>
class BloomFilter {
public:
   // creates a Bloom filter to store at most nmax items
   // with an upper bound 'fp' for false positives
   BloomFilter(int nmax, double fp = 0.05);
void insert(const T& x);
bool contains(const T& x) const;
private:
   ...
}
```

Bloom Filters

```
template <class T>
class HashFunction {
public:
   HashFunction(int M);
int operator()(const T& x) const;
   ...
};

template <class T>
class BloomFilter {
   ...
private:
   bitvector F;
   vector<HashFunction<T> > h;
   int M, k;
   ...
};

template <class T>
BloomFilter::BloomFilter(int nmax, double fp = 0.05) {
    // compute here M and k to achieve the guarantee on false
    // positives
    F = bitvector(M, 0);
    for (int i = 0; i < k; ++i)
       h.push_back(HashFunction<T>(M));
}
```

Bloom Filters

```
template <class T>
void BloomFilter::insert(const T& x) {
  for (int i = 0; i < k; ++i)
     F[h[i](x)] = 1;
}

template <class T>
void BloomFilter::contains(const T& x) {
  for (int i = 0; i < k; ++i)
     if (F[h[i](x)] == 0)
        return false;
  return true; // might be a false positive!
}
```

Bloom Filters

- Probability that the j-th bit is not updated in an insertion

$$\prod_{i=0}^{k-1} \mathbb{P}[h_i(x) \neq j] = \left(1 - \frac{1}{M}\right)^k$$

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

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

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

Bloom Filters

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

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

- Probability that the 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 $\alpha > 0$

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

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

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

Bloom Filters

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

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

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

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

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

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 74GB 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 3TB.
- The optimal number k^* of hash functions for the example above is 4.32 ($\Longrightarrow$ use 4 or 5 hash functions for optimal performance)

Bloom Filters

```
template <class T>
BloomFilter::BloomFilter(int nmax, double fp = 0.05) {
    // compute here M and k to achieve the guarantee on false
    // positives
    M = int(log(1/P)*nmax/log(2)*log(2));
    k = int(log(2)* M/nmax);
    ...
}
```

Bloom Filters

M. Mitzenmacher and E. Upfal.
Probability and computing: Randomized algorithms and probabilistic analysis.
Cambridge University Press, 2005.

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

Part II

Amortized Analysis

Amortized Analysis

To be continued

...

Part III

Priority Queues

4. Binary Heaps
5. Heapsort
6. Binomial Queues
7. Fibonacci Heaps

Priority Queues

A priority queue (cat: *cua de prioritat*; 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

```
// 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

- 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

Heaps

Definition

A heap is a binary tree such that

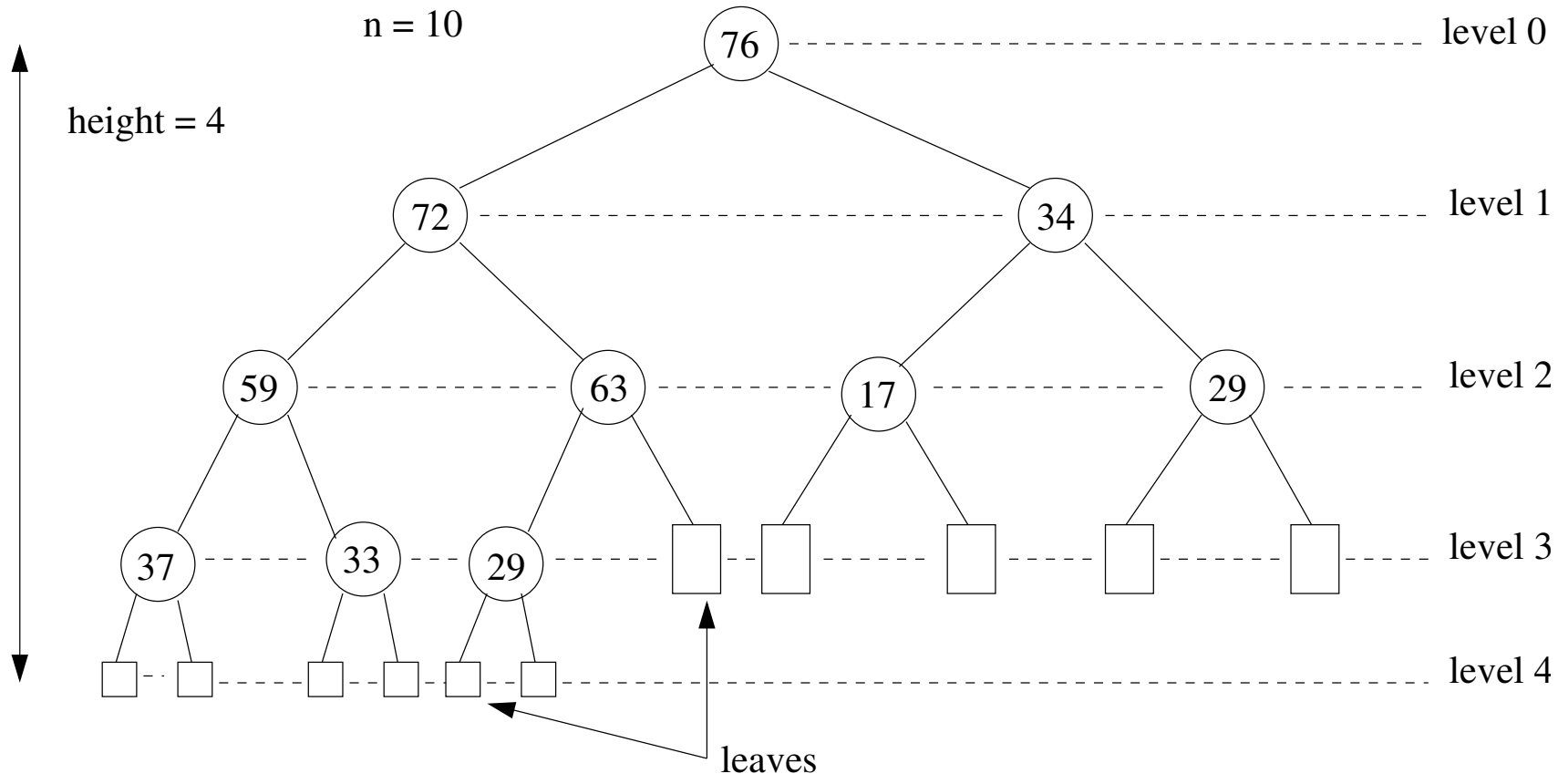
1. All empty subtrees are located in the last two levels of the tree.
2. If a node has an empty left subtree then its right subtree is also empty.
3. The priority of any element is larger or equal than the priority of any element in its descendants.

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.

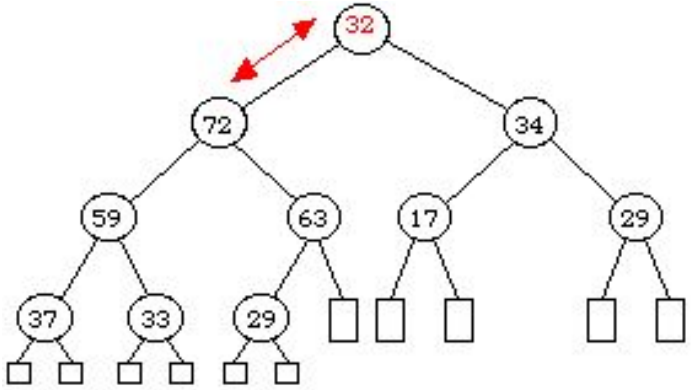
Heaps: Removing the maximum

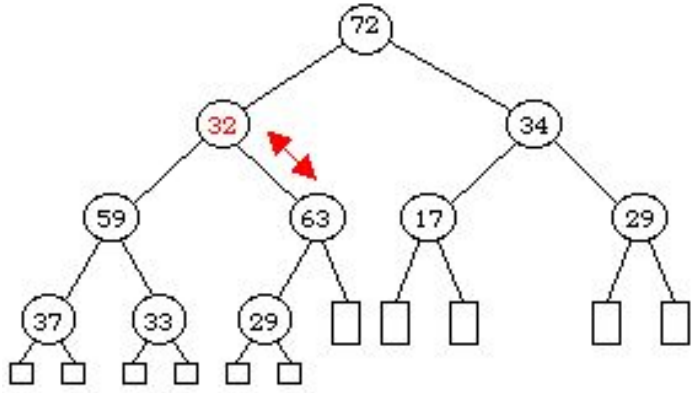
1. Replace the root of the heap with the last element (the rightmost element in the last level)
2. Reestablish the invariant (heap order) sinking the root: The function `sink` exchanges a given node with its largest priority child, if its priority is smaller than the priority of its child, and repeats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

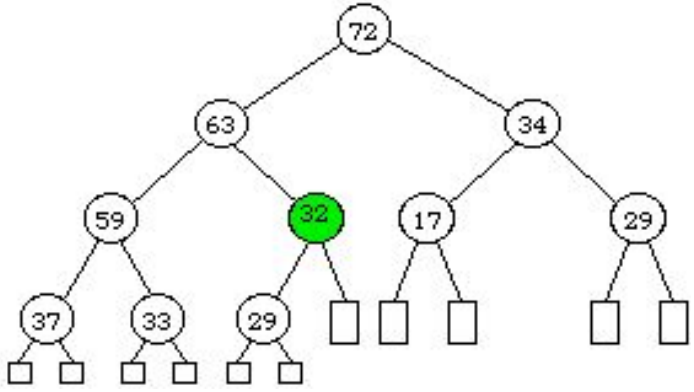
Heaps: Removing the maximum

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

Heapsort

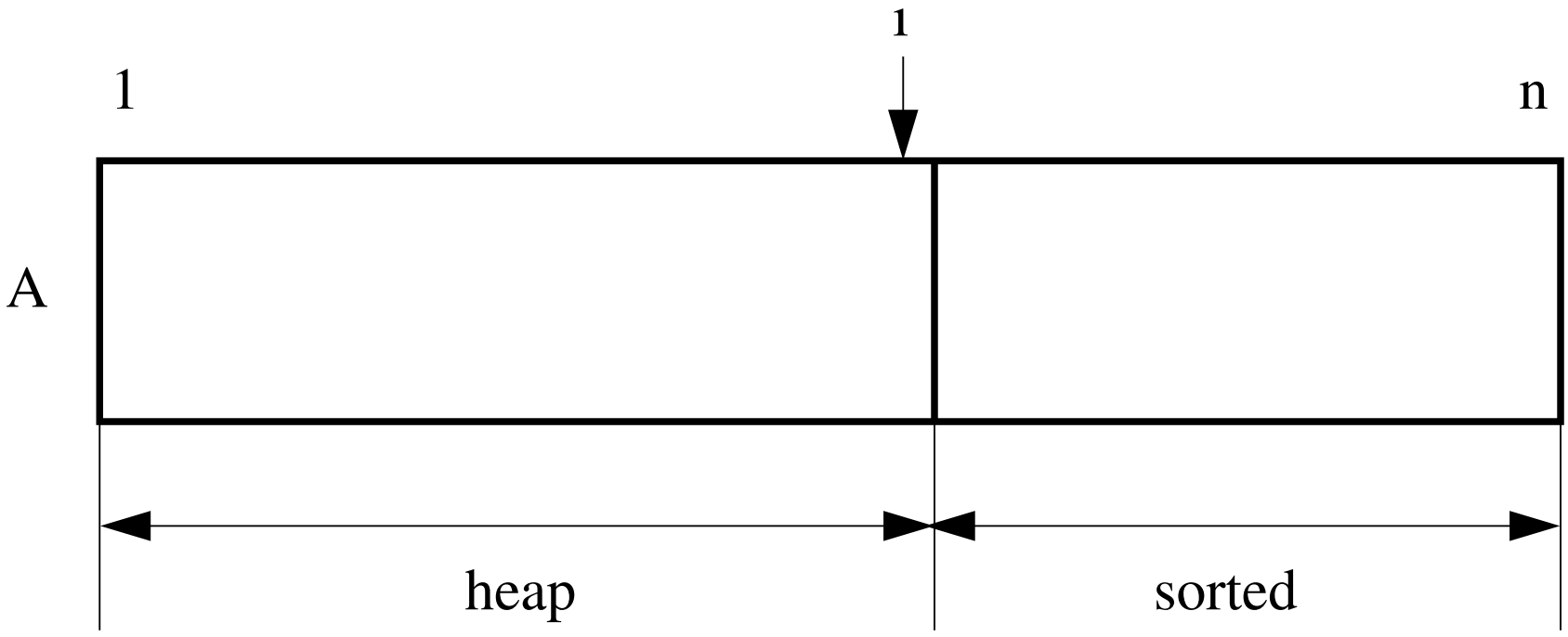
Heapsort (Williams, 1964) sorts an array of n elements building a heap with the n elements and extracting them, one by one, from the heap (cif. our example of the atoic weights and chemical symbols).

The originally given array is used to build the heap; heapsort works in-place and only some constant auxiliary memory space is needed.
Since insertions and deletions in heaps have cost $\mathcal{O}(\log n)$ the cost of the algorithm is $\mathcal{O}(n \log n)$.

Heapsort

```
template <typename Elem>
void sink(Elem v[], int sz, int pos);

// Sort v[1..n] in ascending order
// (v[0] is unused)
template <typename Elem>
void heapsort(Elem v[], int n) {

   make_heap(v, n);
   for (int i = n; i > 0; --i) {
       // extract largest element from the heap
       swap(v[1], v[i]);

       // sink the root to reestablish max-heap order
       // in the subarray v[1..i-1]
       sink(v, i-1, 1);
   }
}
```

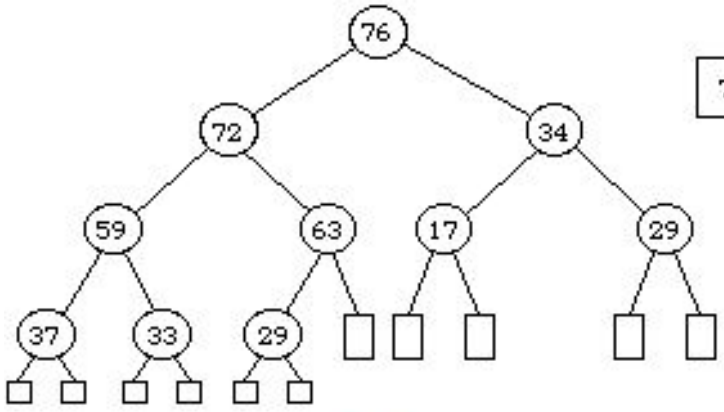
Heapsort

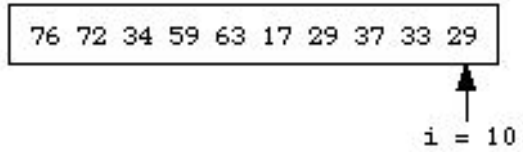


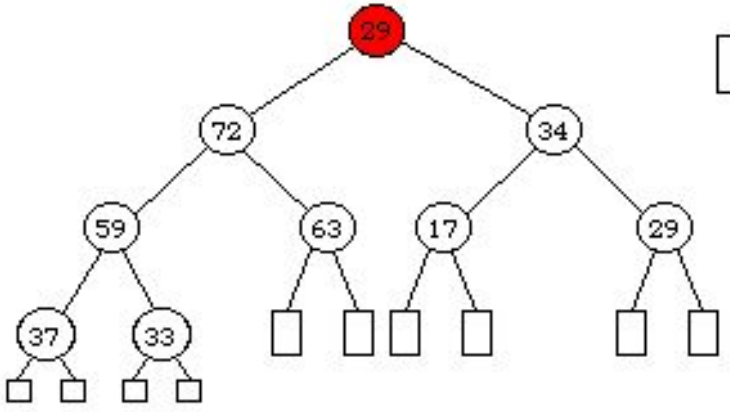
$A[1] \leq A[i+1] \leq A[i+2] \leq \ldots \leq A[n]$

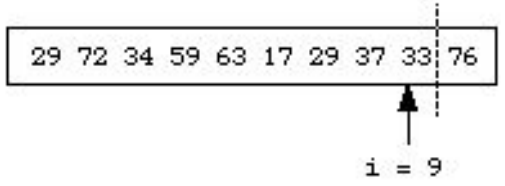
$A[1] = \max_{1 \leq k \leq i} A[k]$

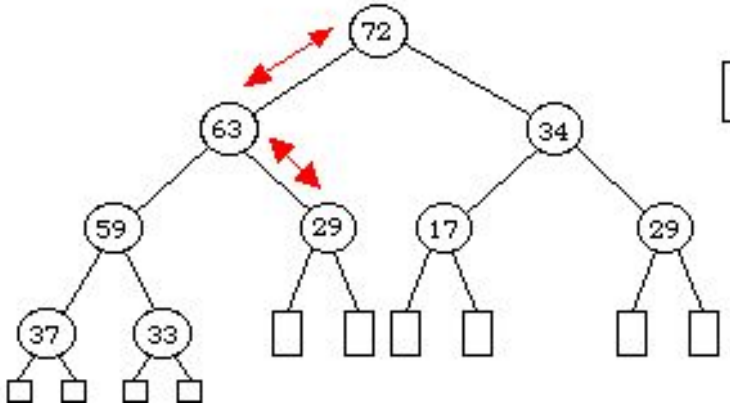
Heapsort

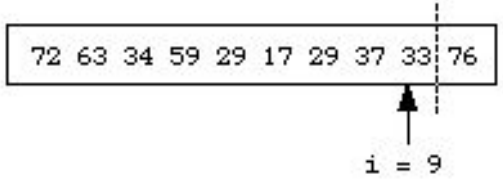

76
72
34
59
63
17
29
37
33
29
29


76 72 34 59 63 17 29 37 33 29
i = 10

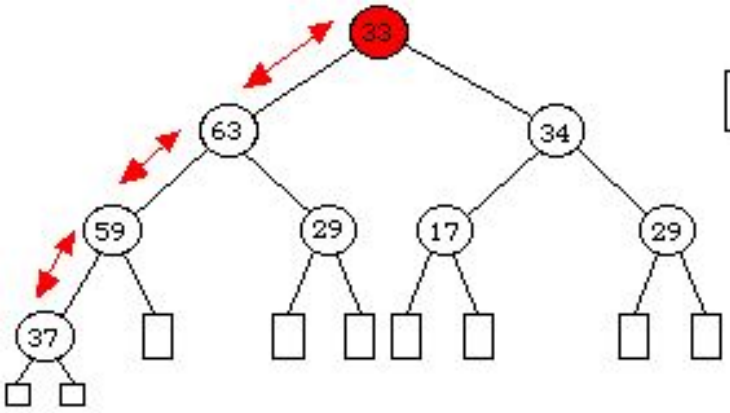

29
72
34
59
63
17
29
37
33

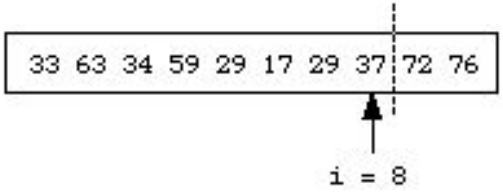

29 72 34 59 63 17 29 37 33 76
i = 9

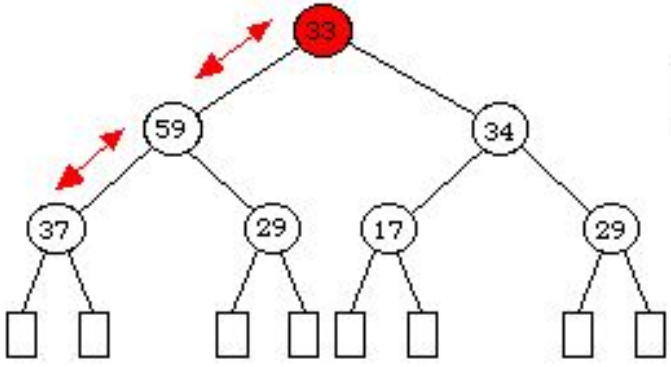

72
63
34
59
29
17
29
37
33

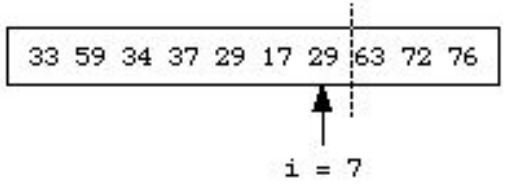

72 63 34 59 29 17 29 37 33 76
i = 9

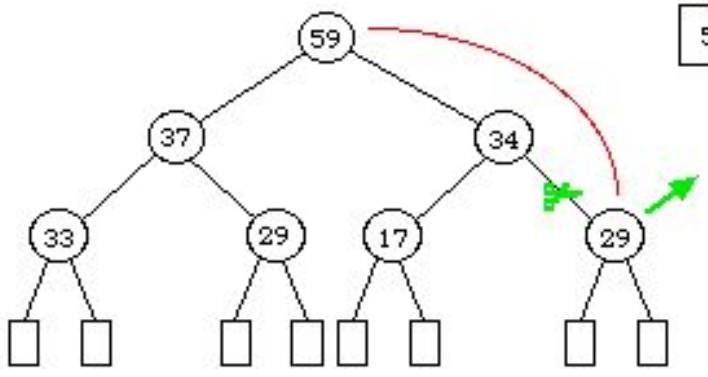
Heapsort

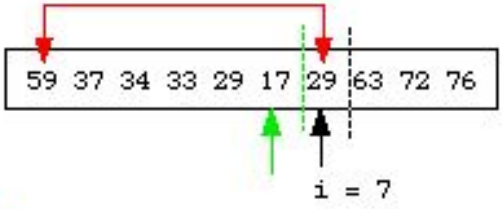

33
63
34
59
29
17
29
37


33 63 34 59 29 17 29 37 72 76
i = 8


33
59
34
37
29
17
29


33 59 34 37 29 17 29 63 72 76
i = 7

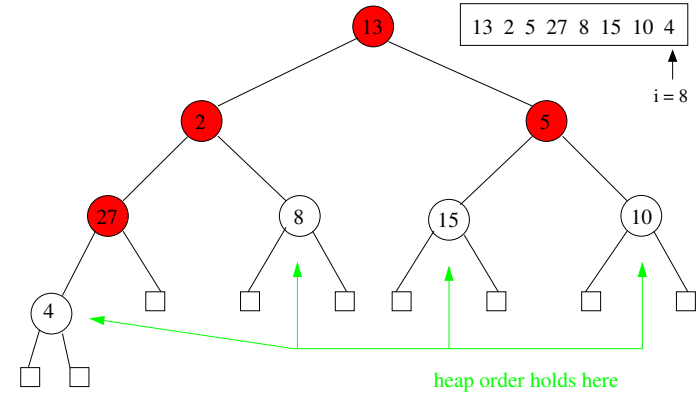

59
37
34
33
29
17
29

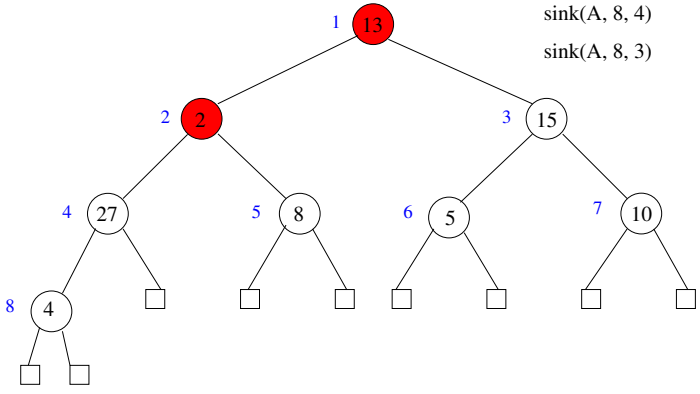

59 37 34 33 29 17 29 63 72 76
i = 7

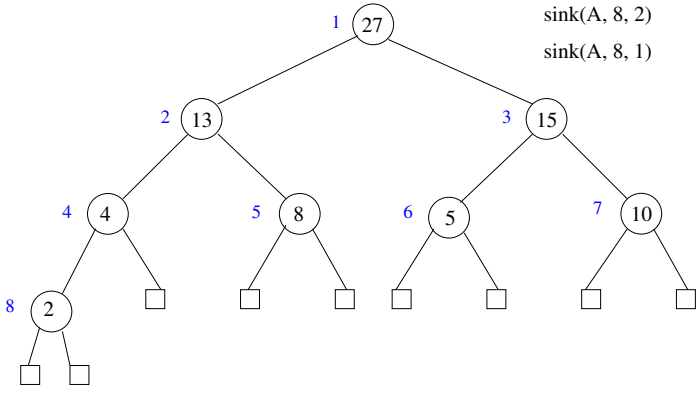
Heapify

```
// Establish (max) heap order in the
// array v[1..n] of Elem's; Elem == priorities
// this is a.k.a. as heapify
template <typename Elem>
void make_heap(Elem v[], int n) {
   for (int i = n/2; i > 0; --i)
       sink(v, n, i);
}
```

Heapify


13 2 5 27 8 15 10 4
i = 8
heap order holds here


sink(A, 8, 4)
sink(A, 8, 3)


sink(A, 8, 2)
sink(A, 8, 1)

The Cost of Heapsort

Let $H(n)$ be the worst-case cost of `heapsort` and $B(n)$ the cost `make_heap`. Since the worst-case cost of `sink`$(v, i-1, 1)$ is $\mathcal{O}(\log i)$ we have

$$
\begin{aligned}
H(n) &= B(n) + \sum_{i=1}^{i=n} \mathcal{O}(\log i) \\
&= B(n) + \mathcal{O}\left(\sum_{1 \le i \le n} \log_2 i \right) \\
&= B(n) + \mathcal{O}(\log(n!)) = B(n) + \mathcal{O}(n \log n)
\end{aligned}
$$

A rough analysis of $B(n)$ shows that $B(n) = \mathcal{O}(n \log n)$ since it makes $\Theta(n)$ calls to `sink`, each one with cost $\mathcal{O}(\log n)$. Hence, $H(n) = \mathcal{O}(n \log n)$; actually, $H(n) = \Theta(n \log n)$ in any case if all elements are different.

The Cost of Heapify

A refined analysis of $B(n)$:

$$\begin{aligned} B(n) &= \sum_{1 \le i \le \lfloor n/2 \rfloor} \mathcal{O}(\log(n/i)) \\ &= \mathcal{O}\left(\log \frac{n^{n/2}}{(n/2)!}\right) \\ &= \mathcal{O}\left(\log(2e)^{n/2}\right) = \mathcal{O}(n) \end{aligned}$$

Since $B(n) = \Omega(n)$, we conclude $B(n) = \Theta(n)$.

The Cost of Heapify

Alternative proof: Let $h = \lceil \log_2(n+1) \rceil$ the height of the heap. Level $h-1-k$ contains at most

$$2^{h-1-k} < \frac{n+1}{2^k}$$

elements; in the worst-case each one will sink down to level $h-1$ with cost $\mathcal{O}(k)$

The Cost of Heapify

$$
\begin{aligned}
B(n) &= \sum_{0 \le k \le h-1} \mathcal{O}(k) \frac{n+1}{2^k} \\
&= \mathcal{O}\left(n \sum_{0 \le k \le h-1} \frac{k}{2^k} \right) \\
&= \mathcal{O}\left(n \sum_{k \ge 0} \frac{k}{2^k} \right) = \mathcal{O}(n),
\end{aligned}
$$

since

$$
\sum_{k \ge 0} \frac{k}{2^k} = 2.
$$

In general, if $0 < |r| < 1$,

$$
\sum_{k \ge 0} k \cdot r^k = \frac{r}{(1-r)^2}.
$$

The Cost of Heapify

Despite $H(n) = \Theta(n \log n)$, the refined analysis of $B(n)$ is important: using a *min-heap* we can get the smallest k elements in an array in ascending order with cost:

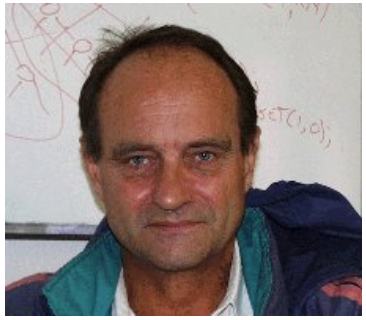
$$\begin{aligned} S(n,k) &= B(n) + k \cdot \mathcal{O}(\log n) \\ &= \mathcal{O}(n + k \log n). \end{aligned}$$

If $k = \mathcal{O}(n/\log n)$ then $S(n,k) = \mathcal{O}(n)$.

Part III

Priority Queues

Binomial Queues



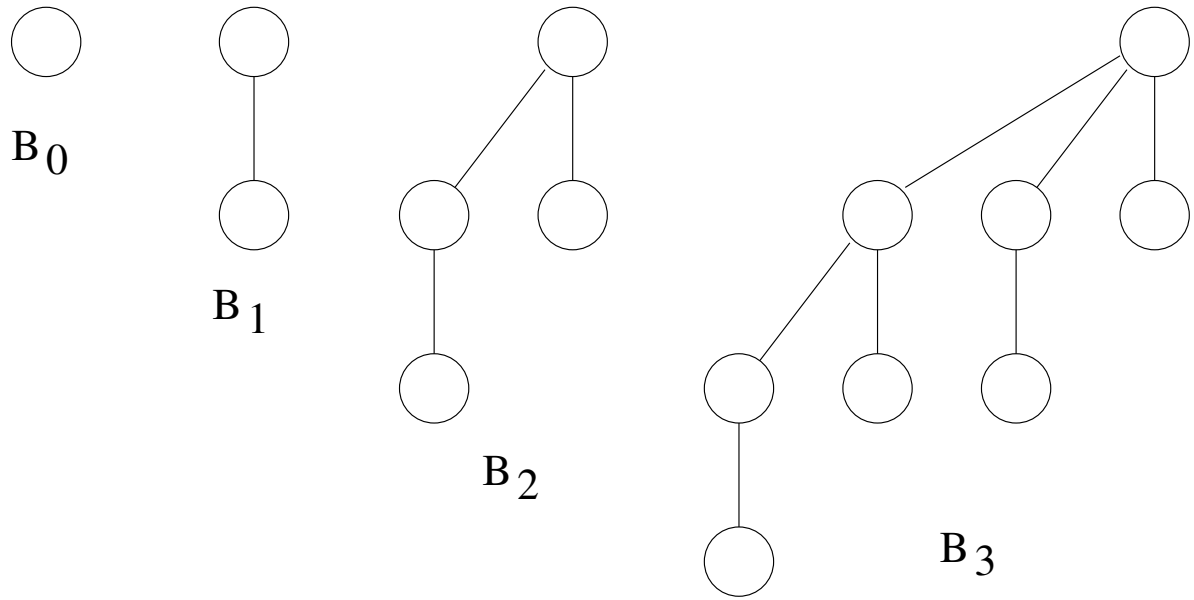
J. Vuillemin

- 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();

  ...
};
```

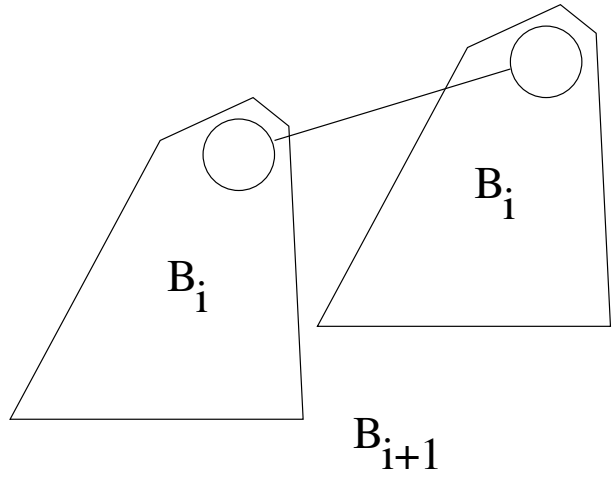
Binomial Queues

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



Binomial Queues

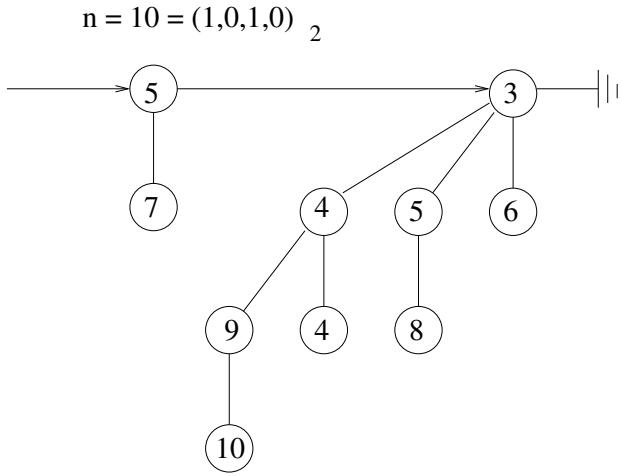
- A binomial tree of order $i+1$ is (recursively) built by planting a binomial tree B_i as a child of the root of another binomial tree B_i.



- 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, ...

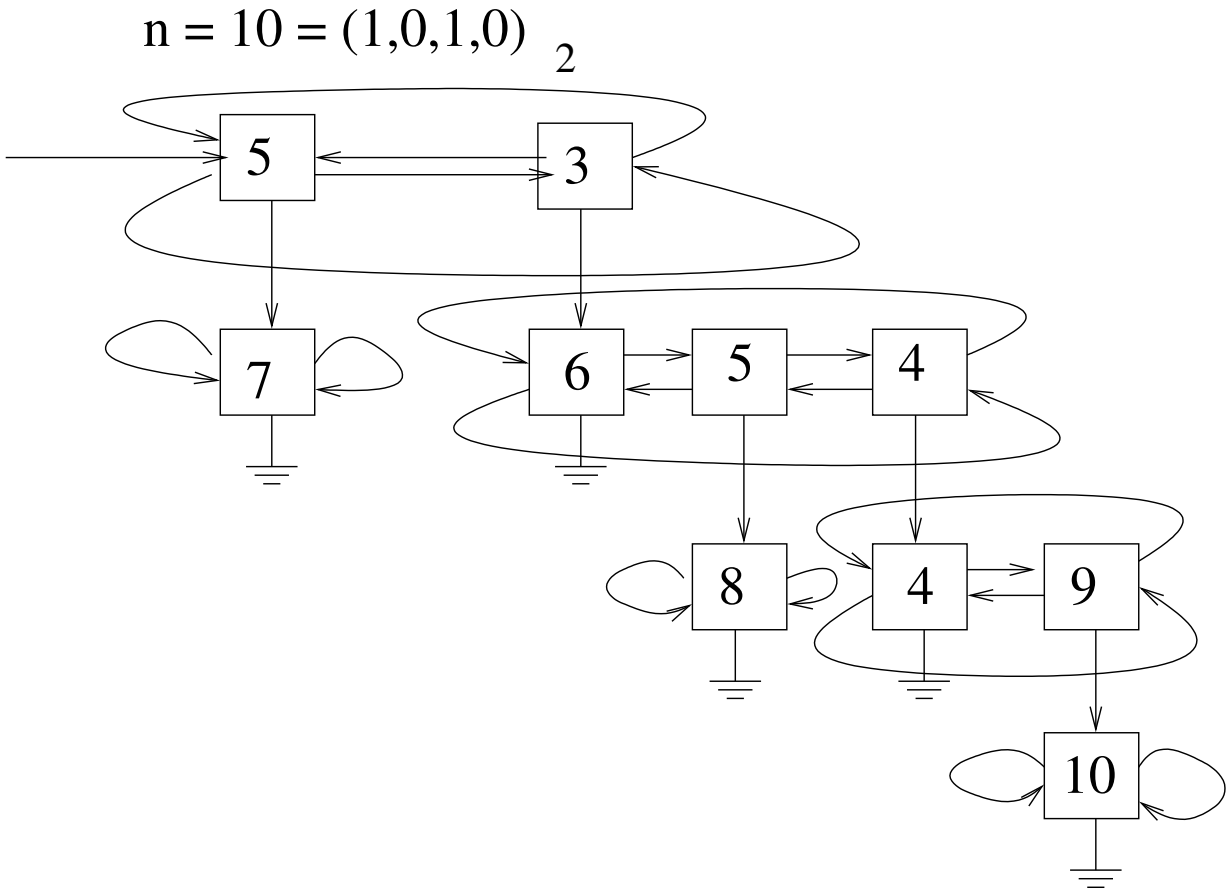


- A binomial queue for n elements contains at most $\lceil \log_2(n+1) \rceil$ binomial trees
- The n elements of the binomial queue are stored in the binomial trees in such a way that each binomial tree satisfies the heap property: the priority of the element at any given node is $\leq$ than the priority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

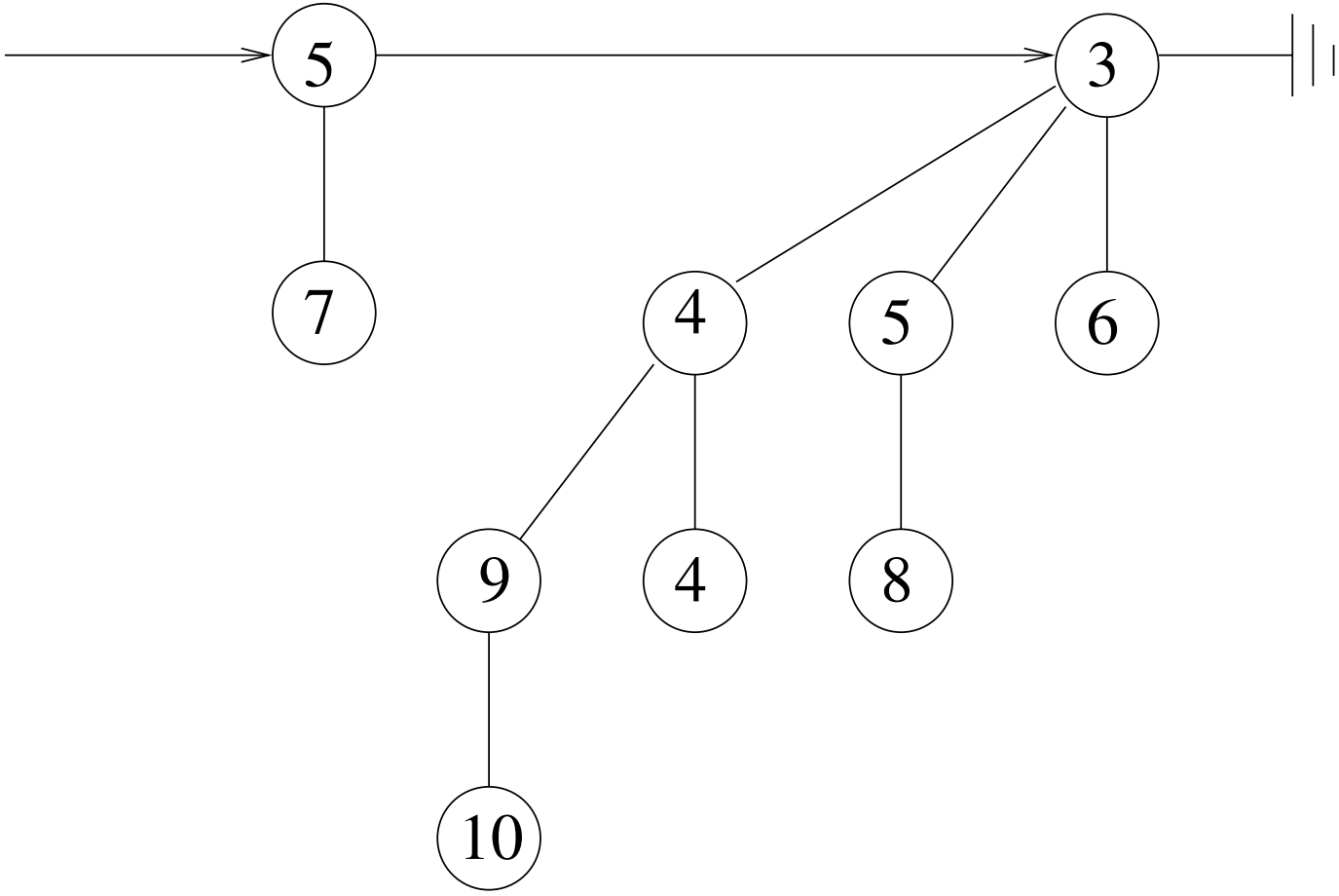
- To delete an element of minimum priority from a queue Q, we start locating such an element, say x; it must be at the root of some B_i
- The root of B_i is dettached from Q and thus B_i is no longer part of the original queue Q; the list of x's children is a binomial queue Q' with $2^i - 1$ elements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

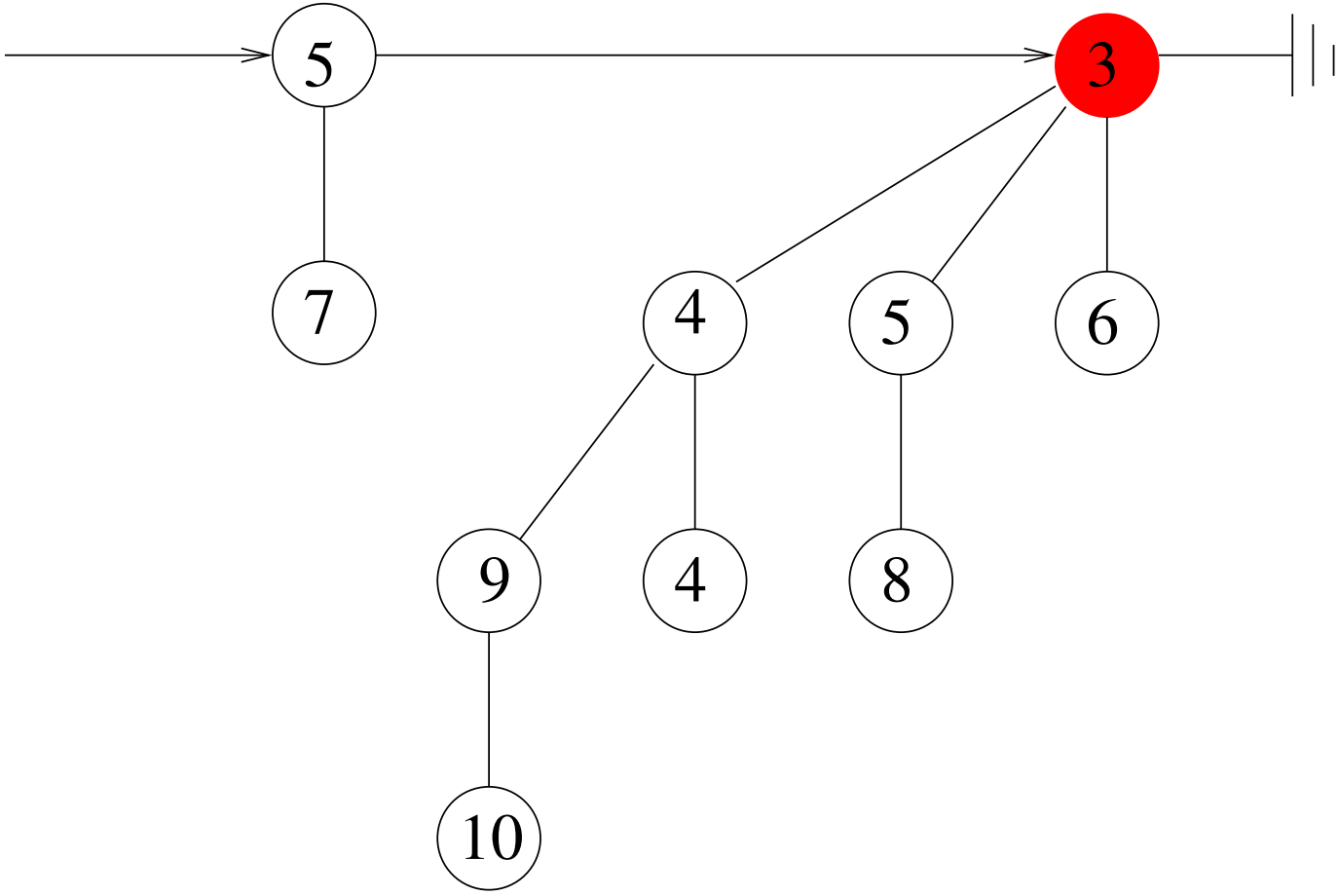
Binomial Queues

$n = 10 = (1,0,1,0)_2$



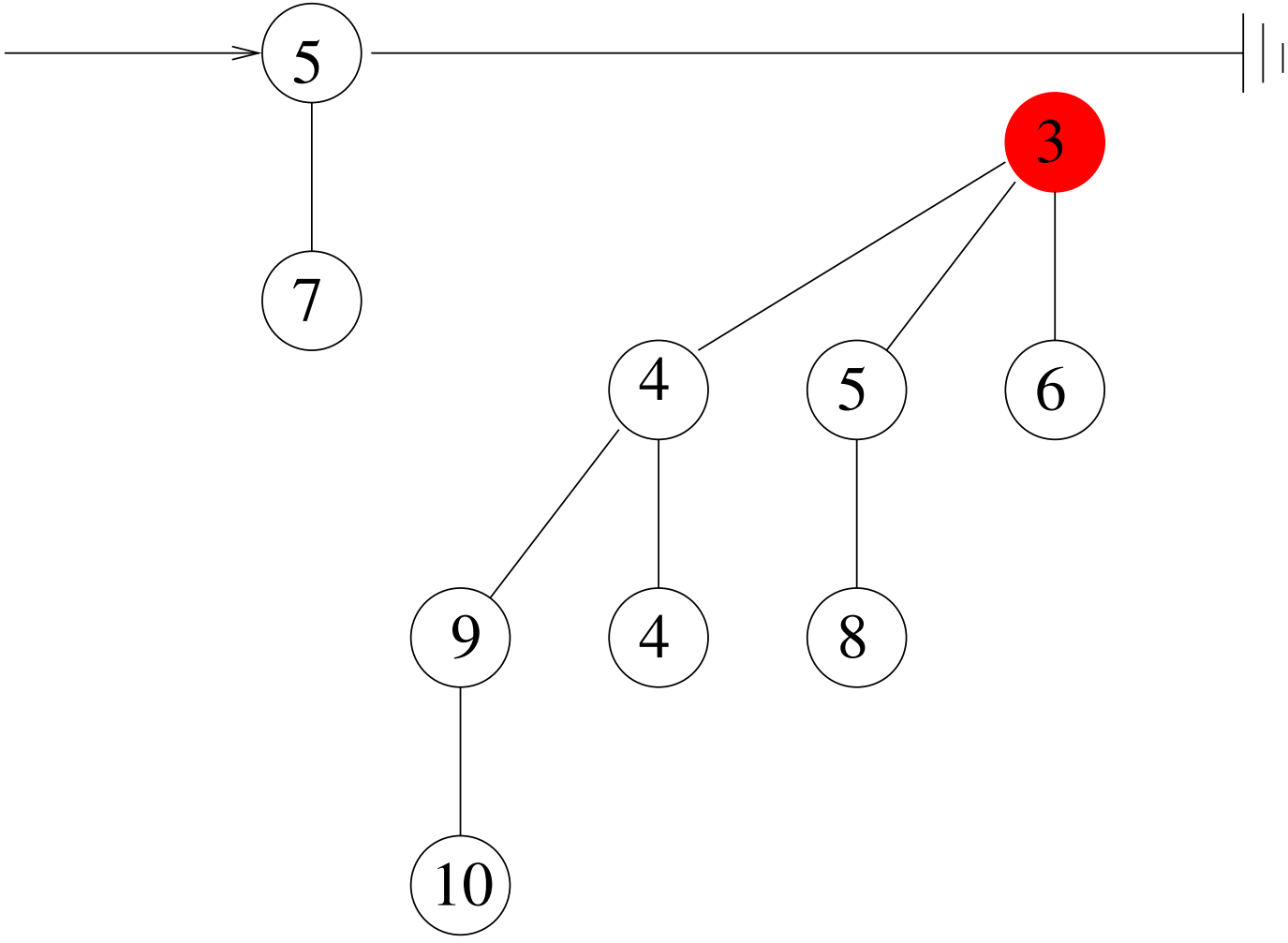
Binomial Queues

$n = 10 = (1,0,1,0)_2$

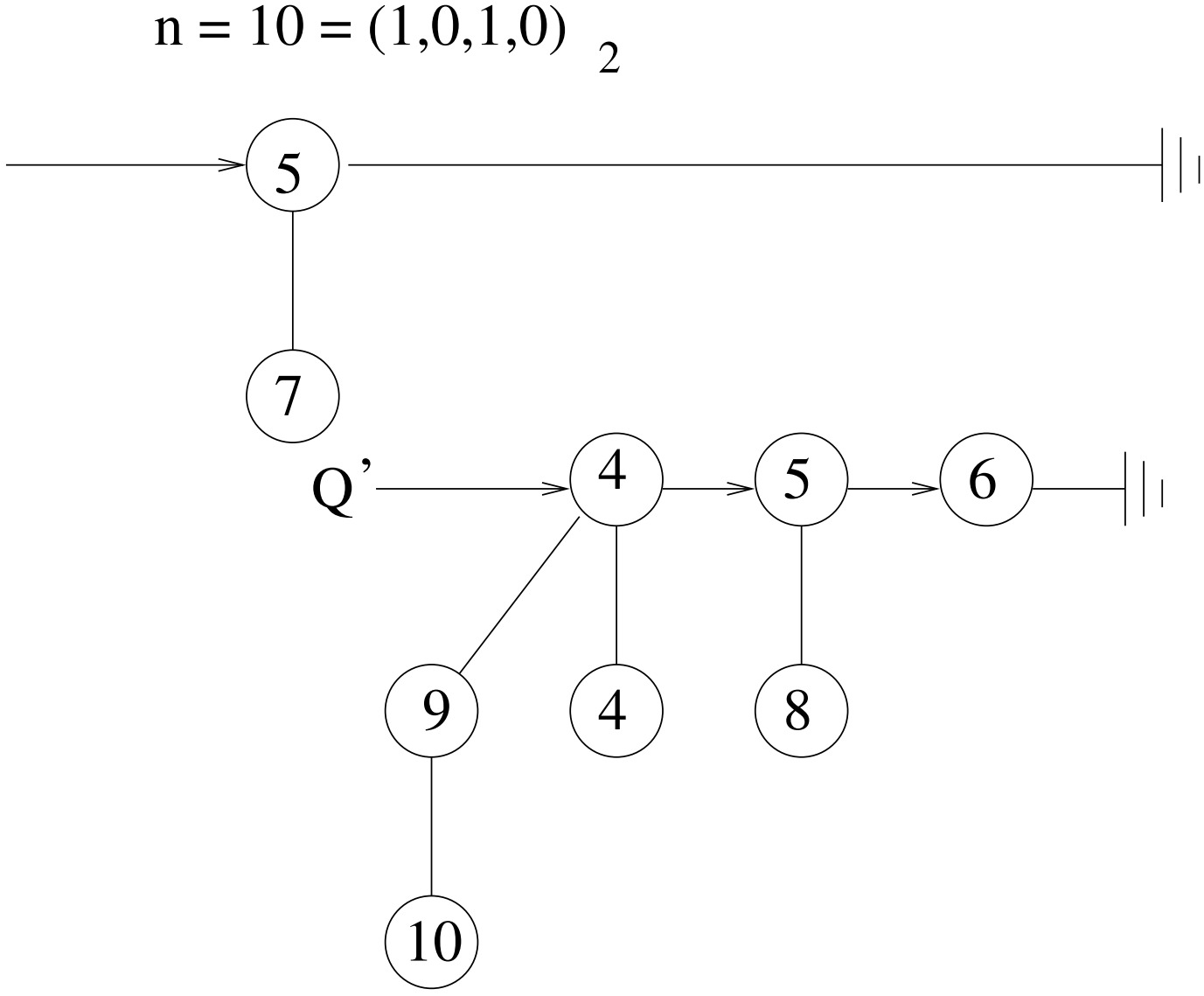


Binomial Queues

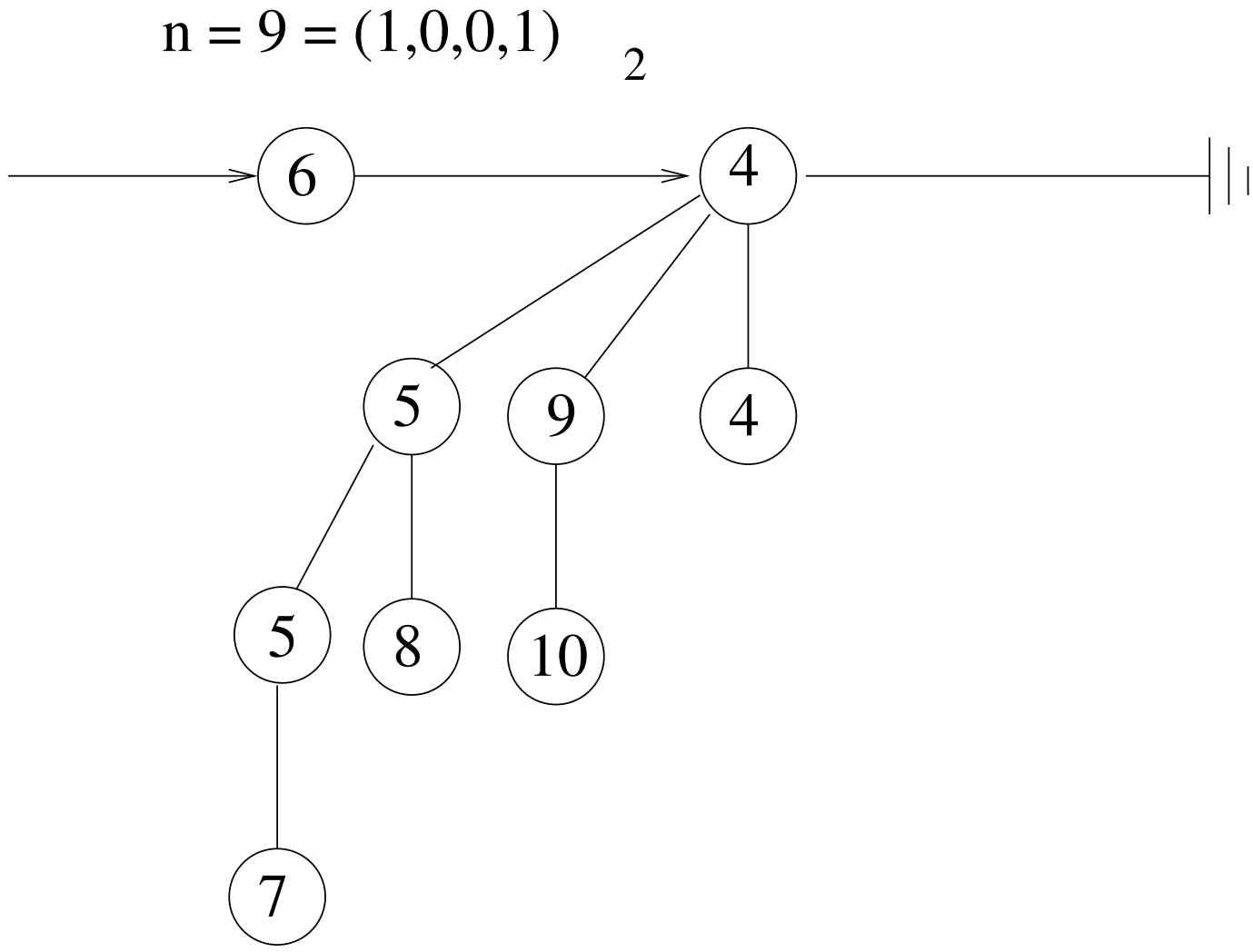
$n = 10 = (1,0,1,0)_2$



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

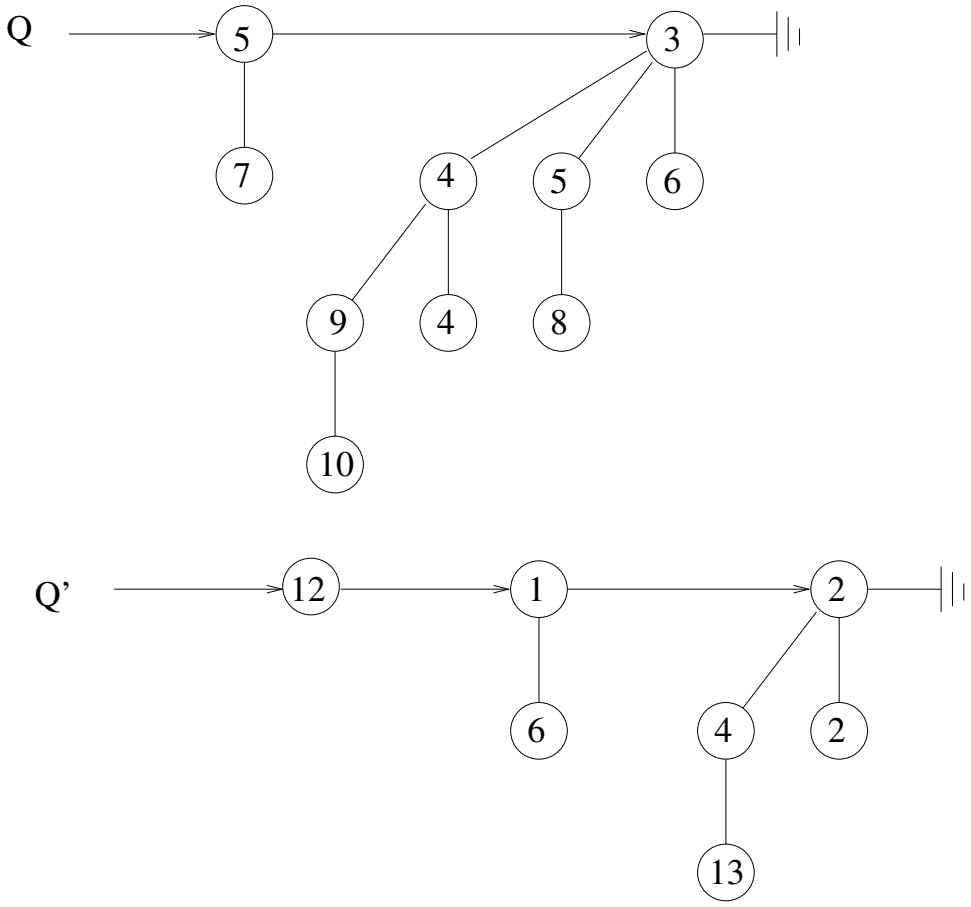
Binomial Queues

- 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 $C = \emptyset$

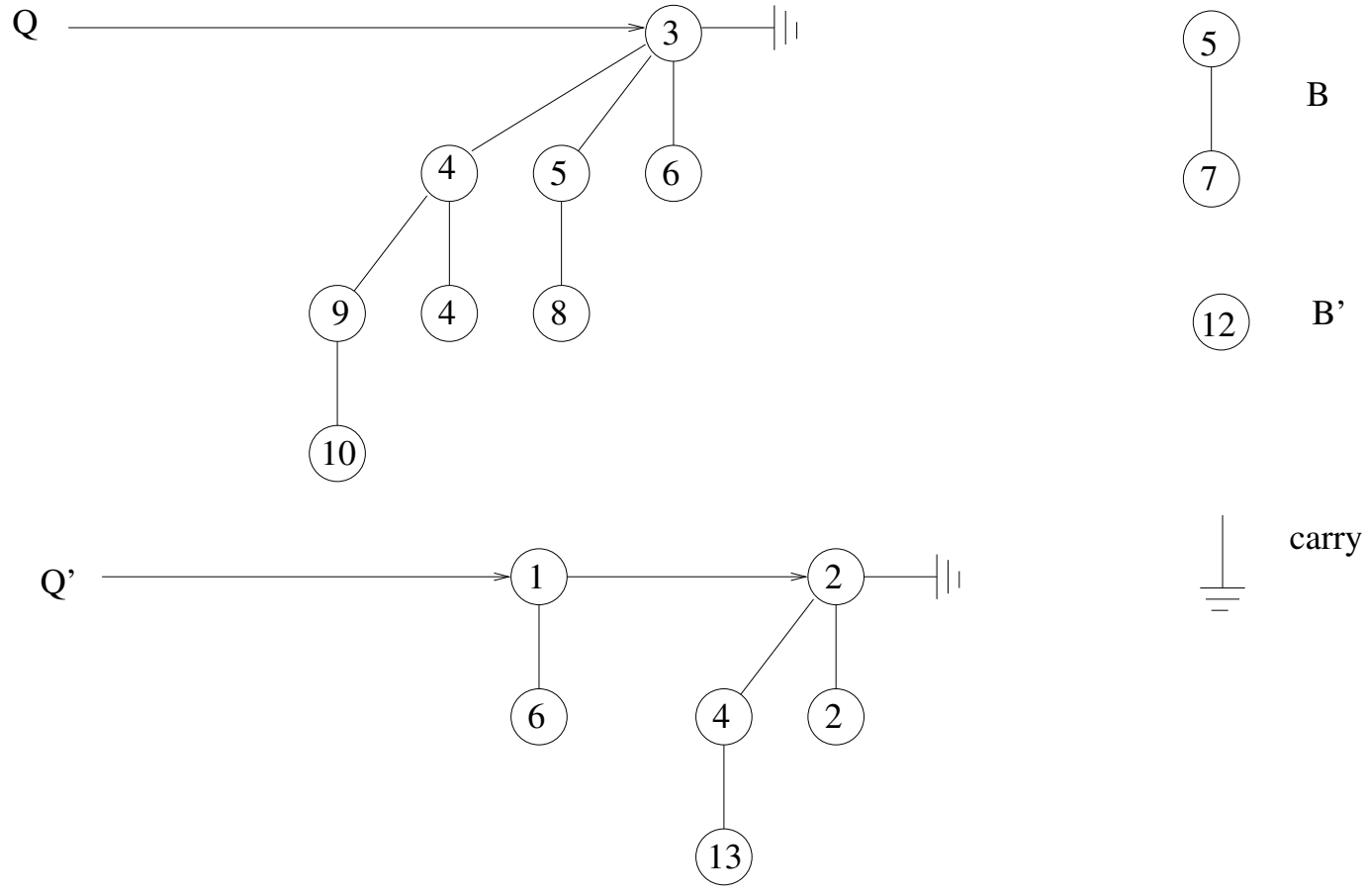
Binomial Queues

- 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

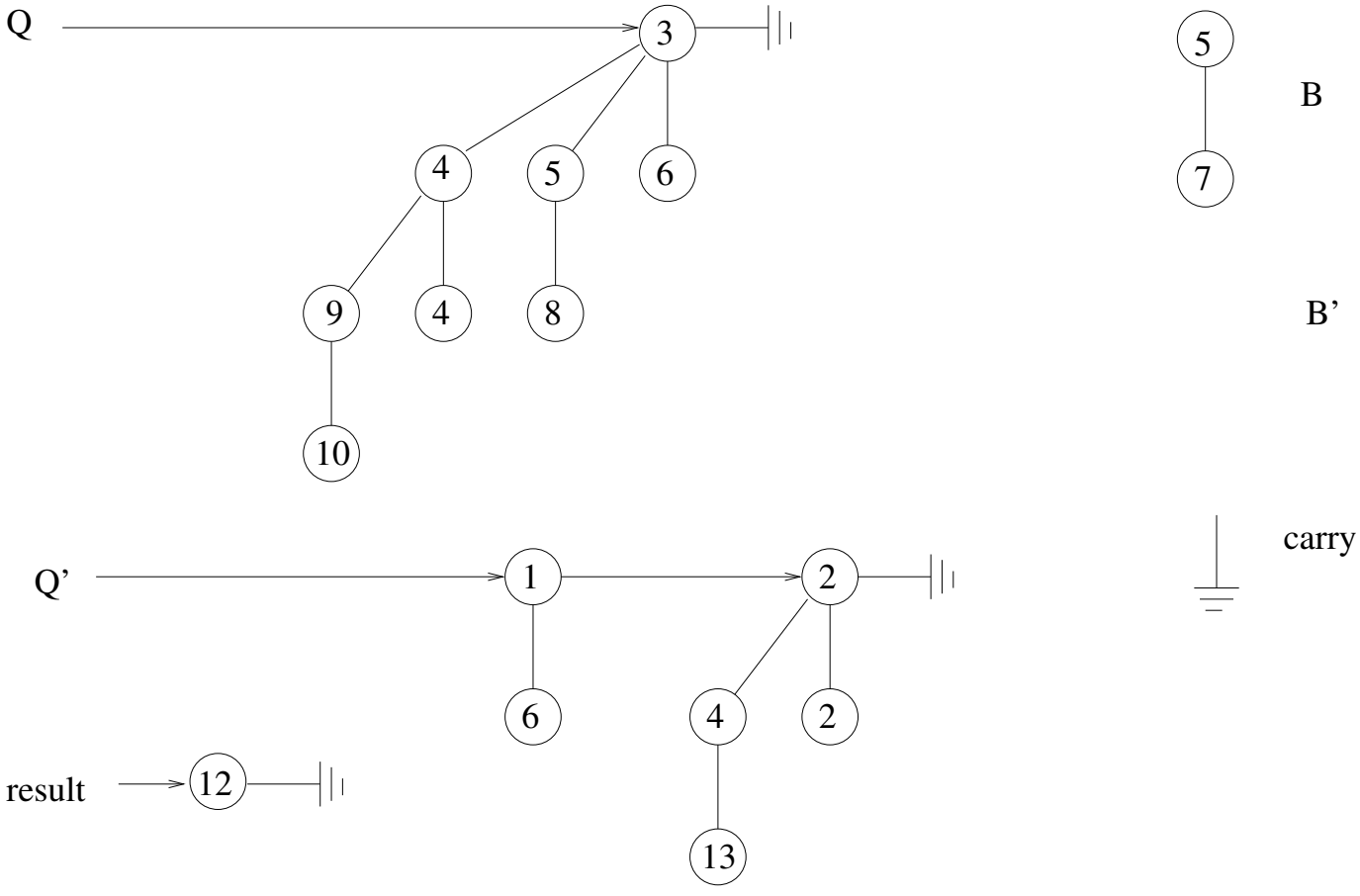
Binomial Queues



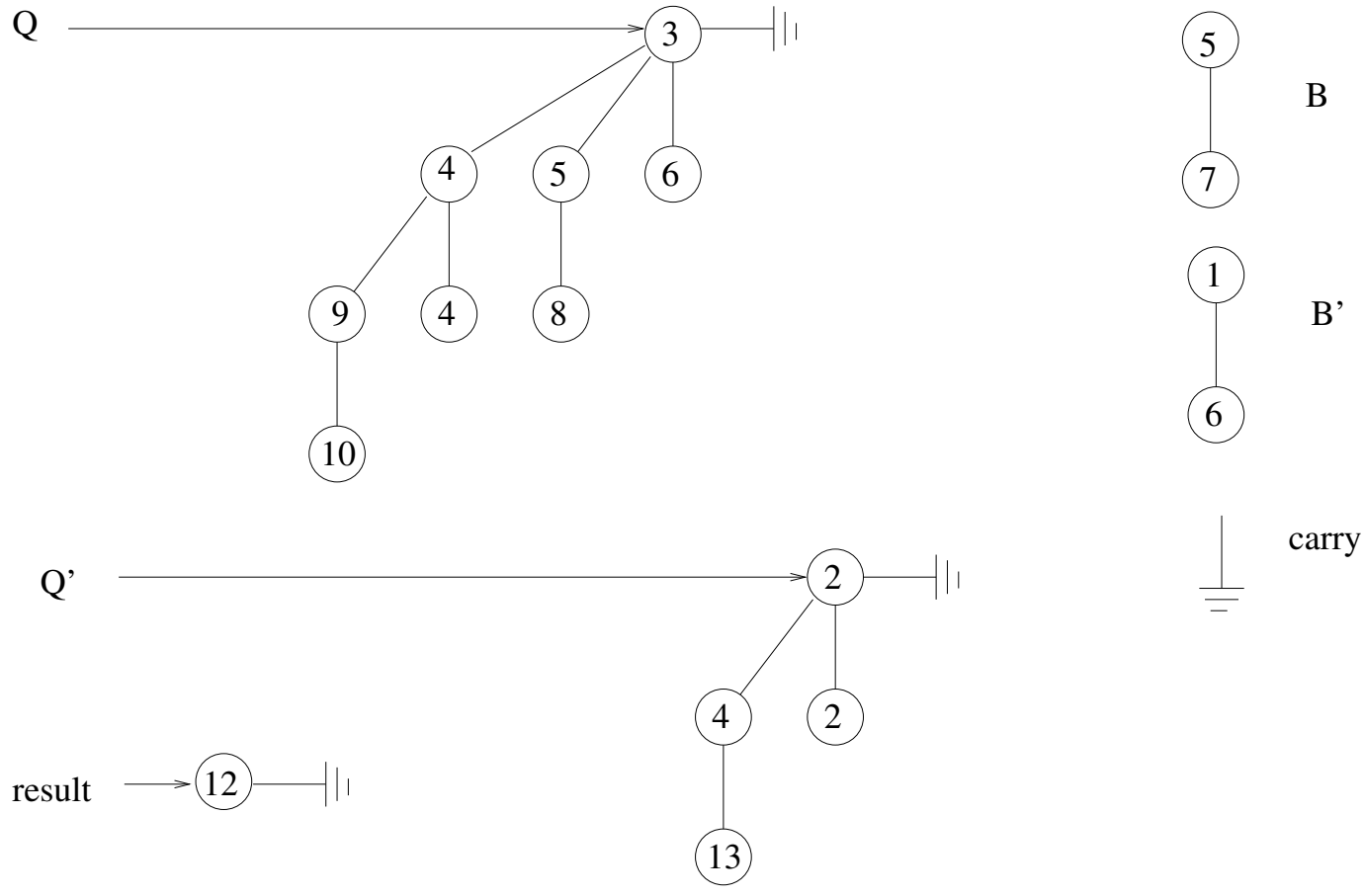
Binomial Queues



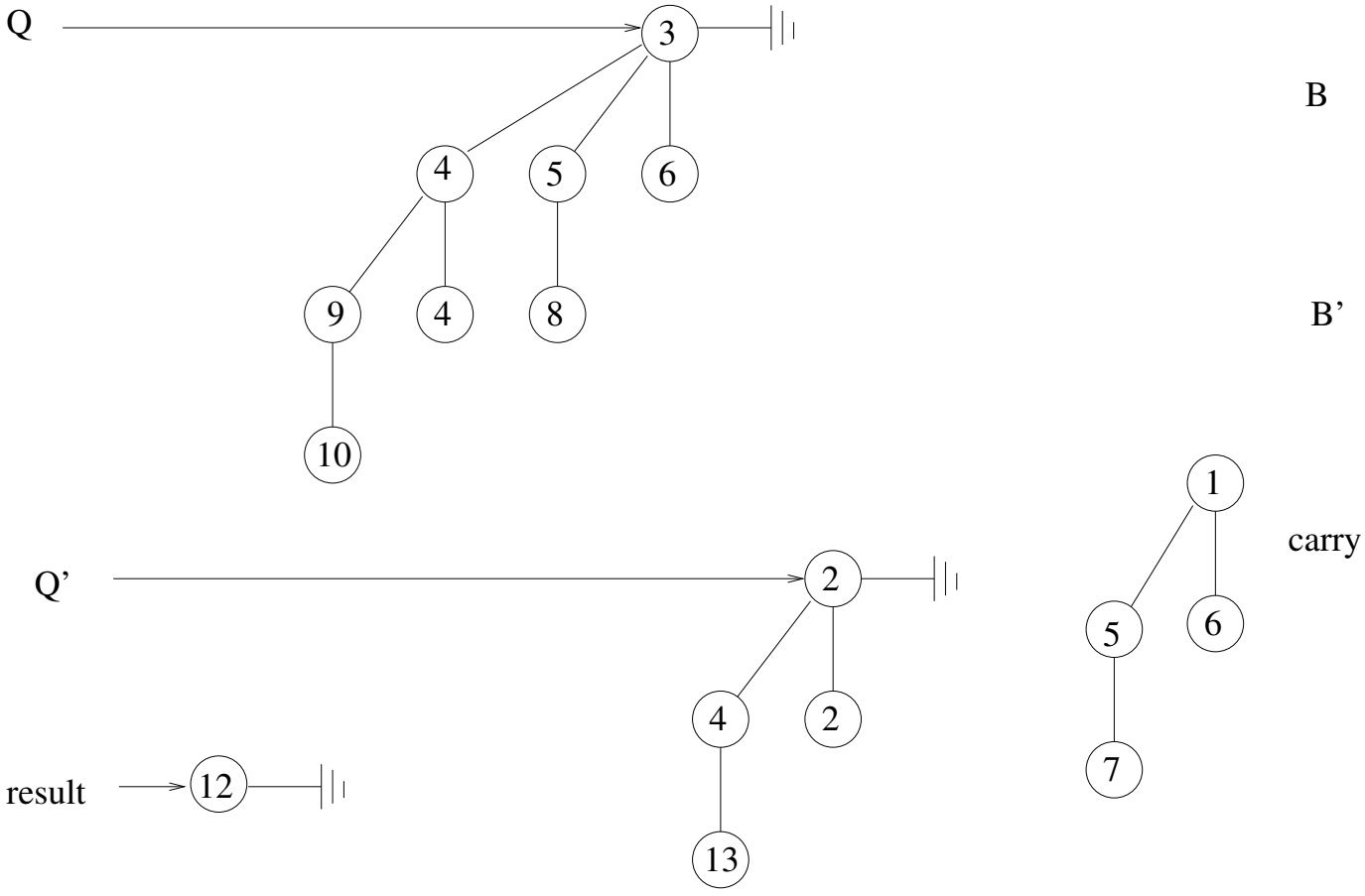
Binomial Queues



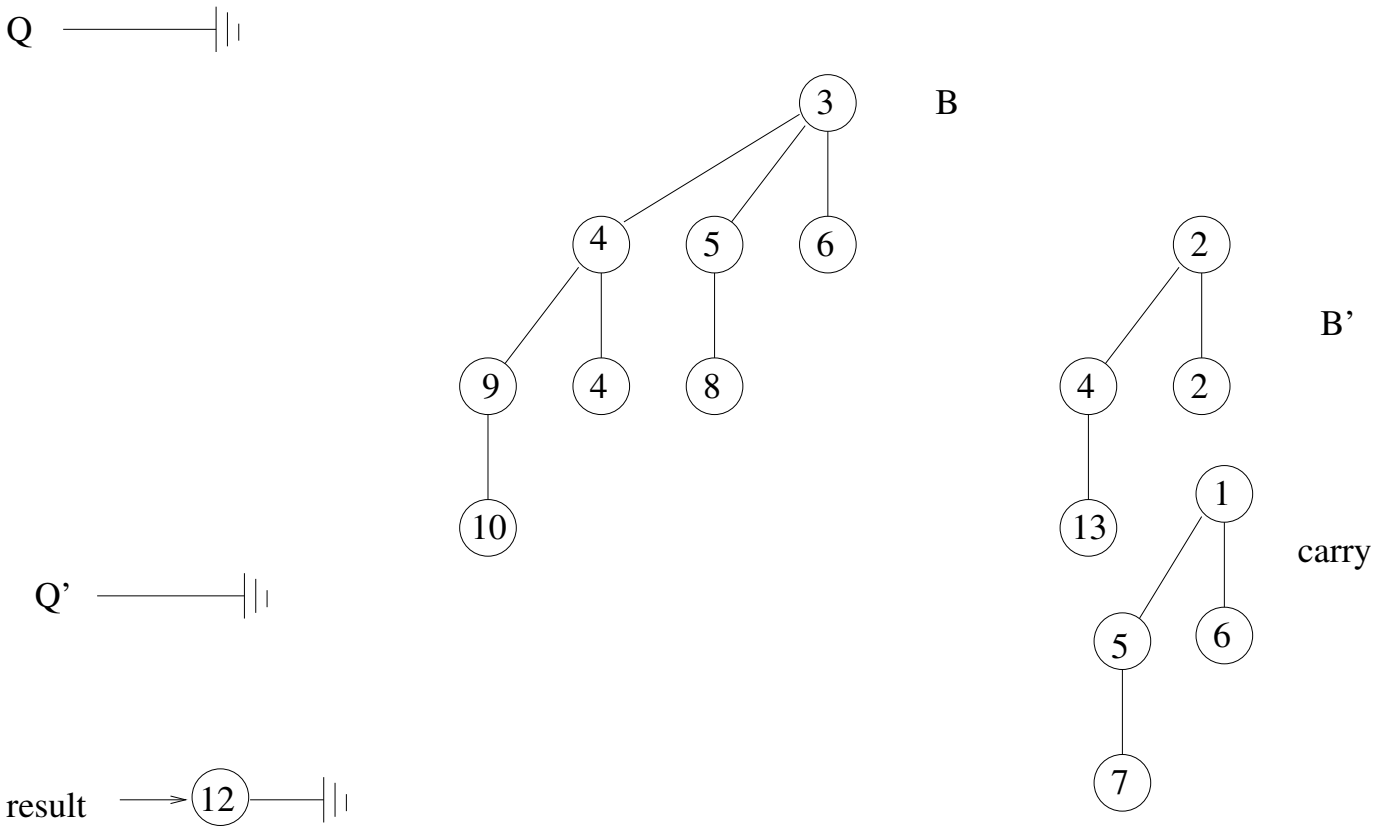
Binomial Queues



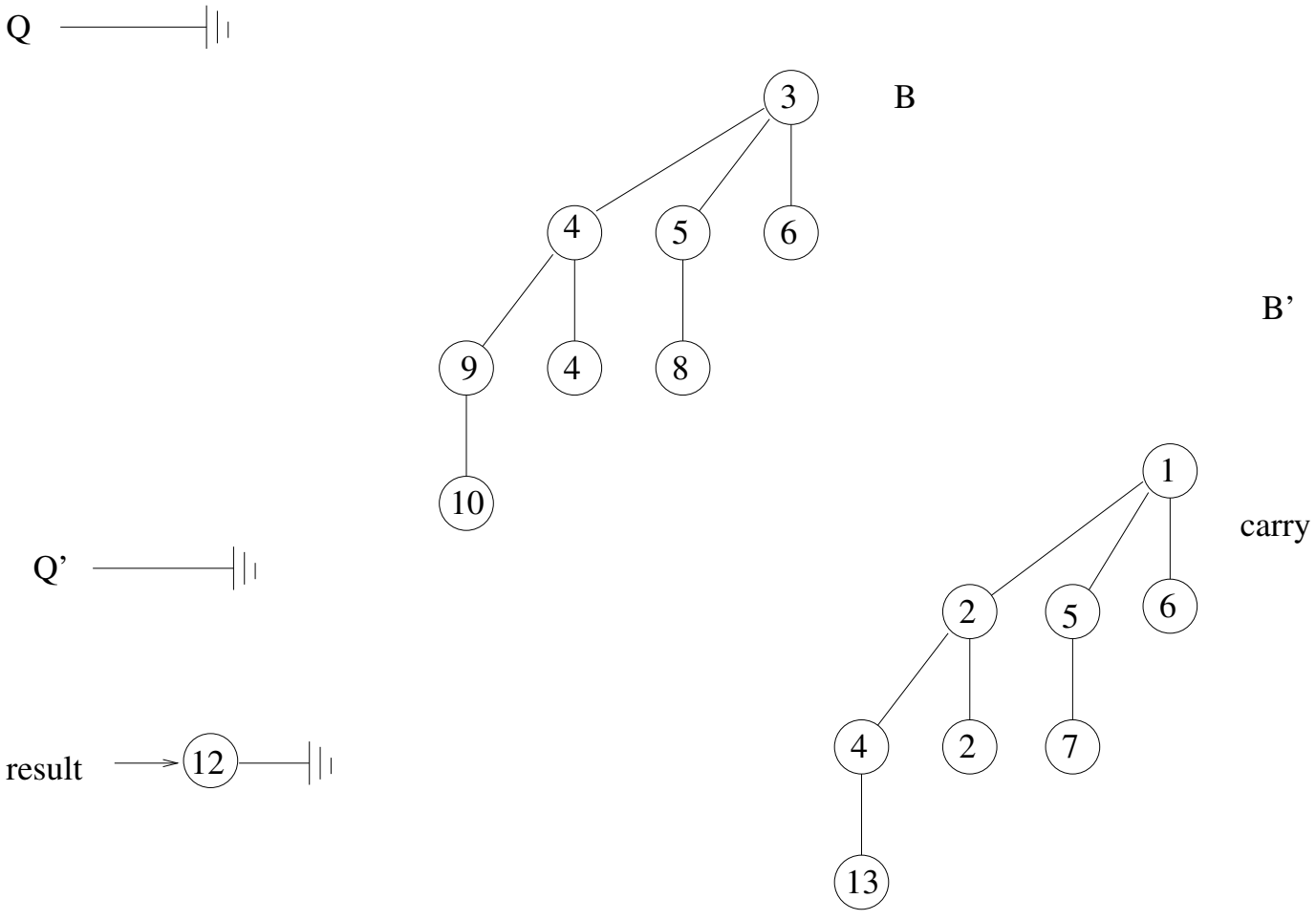
Binomial Queues



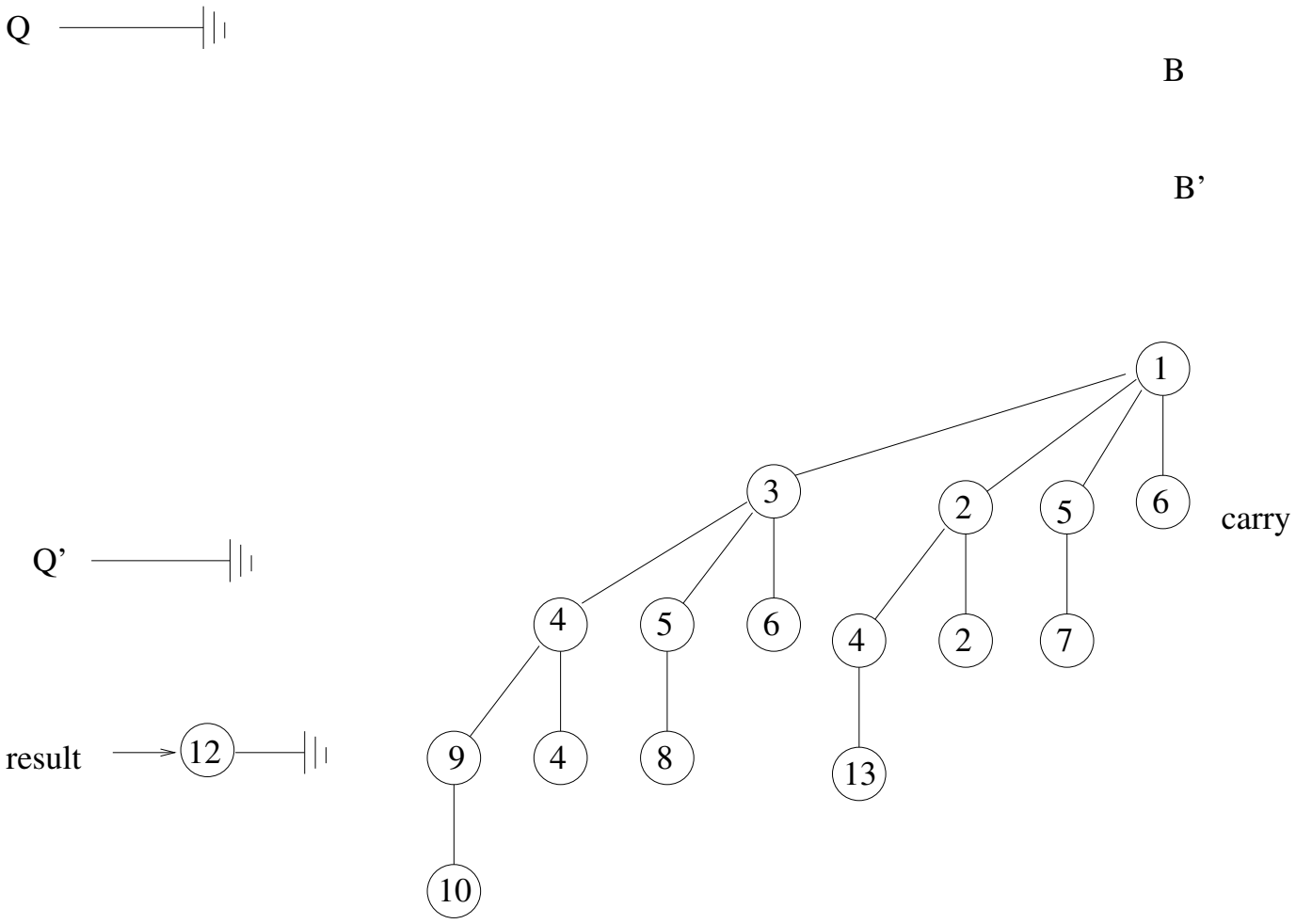
Binomial Queues



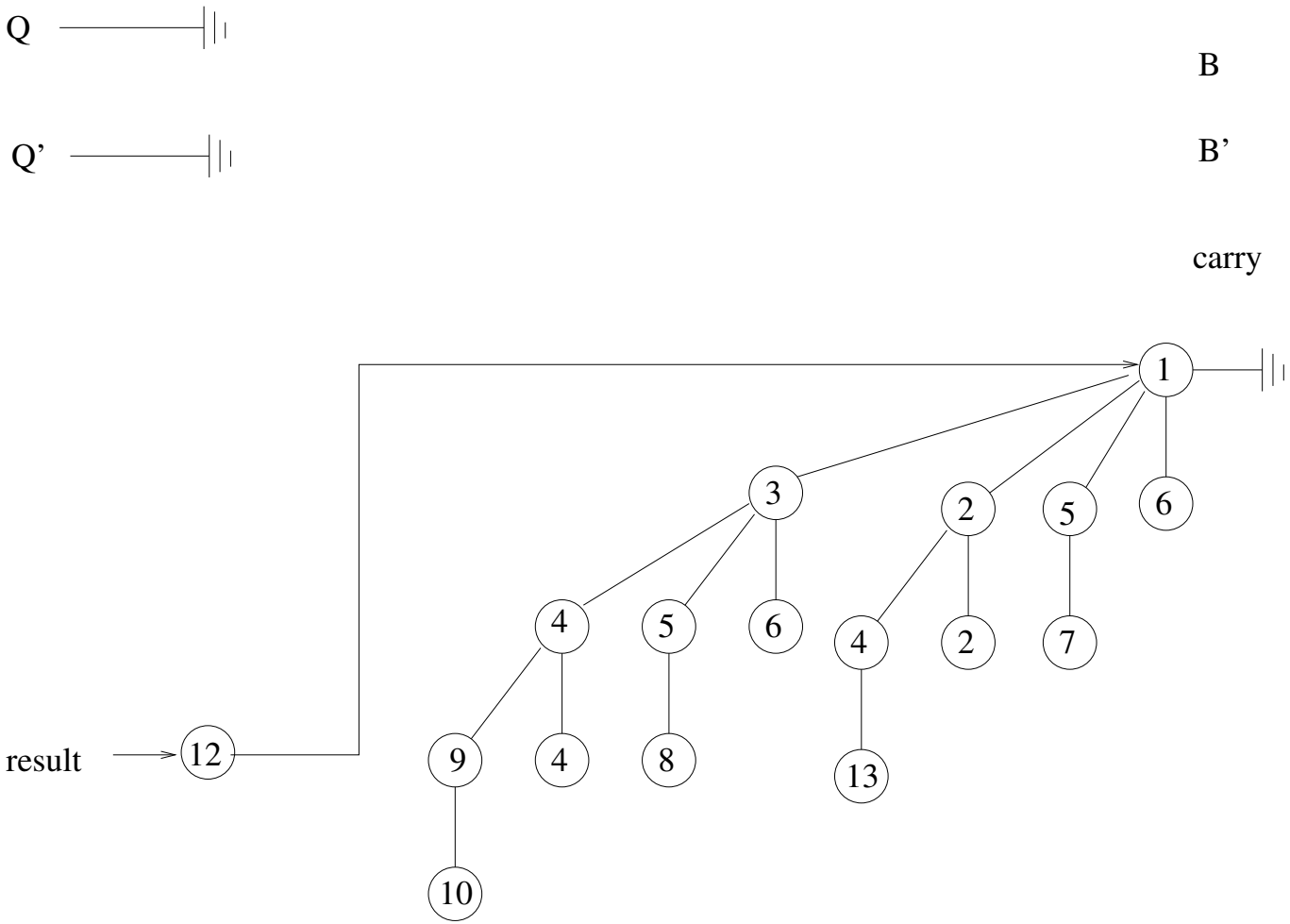
Binomial Queues



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:

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

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

Part III

Priority Queues

Fibonacci Heaps

To be

continued. . .