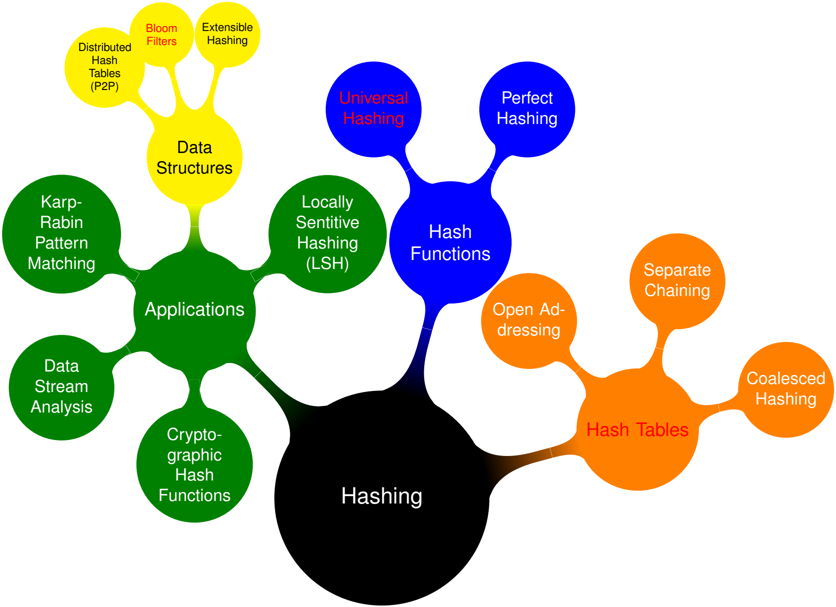
Advanced Data Structures

Conrado Martínez
U. Politècnica Catalunya

April 4, 2017

1. Hashing
   - Universal Hashing
   - Bloom Filters
2. Skip lists
3. Binomial Queues

Introduction



Introduction

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

1. Hashing
   - Universal Hashing
   - Bloom Filters
2. Skip lists
3. Binomial Queues

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)

1. Hashing
   - Universal Hashing
   - Bloom Filters
2. Skip lists
3. Binomial Queues

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

```cpp
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 ($\implies$ 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.

1. Hashing
2. Skip lists
3. Binomial Queues

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

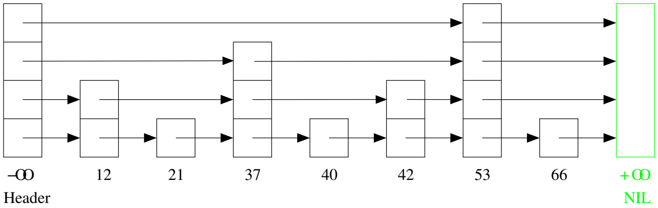
- Skip lists were invented by William Pugh (C. ACM, 1990) as a simple alternative to balanced trees
- The algorithms to search, insert, delete, etc. are very simple to understand and to implement, and they have very good expected performance—independent of any assumption on the input

Skip lists

A skip list S for a set X consists of:

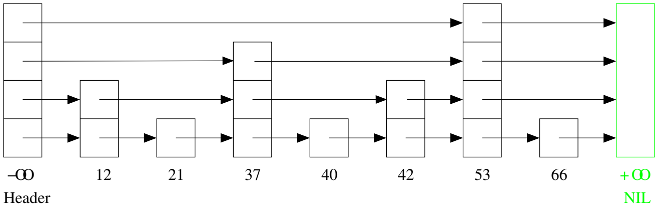
1. A sorted linked list L_1, called level 1, contains all elements of X
2. A collection of non-empty sorted lists $L_2, L_3, \ldots$, called level 2, level 3, ... such that for all $i \geq 1$, if an element x belongs to L_i then x belongs to L_{i+1} with probability p, for some $0 < p < 1$

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:

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

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:

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

Skip lists

- The height of the skip list S is the number of non-empty lists,
$$\text{height}(S) = \max_{x \in S}\{\text{height}(x)\}$$
- The random variable H_n giving the height of a random skip list of n is the maximum of n i.i.d. Geom(p)
- Several performance measures of skip lists are expressed in terms of the probabilistic behavior of a sequence of n i.i.d. geometric r.v. of parameter p

Skip lists

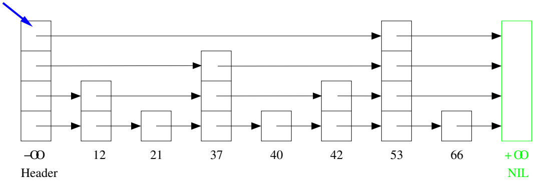
- The height of the skip list S is the number of non-empty lists,
$$\text{height}(S) = \max_{x \in S}\{\text{height}(x)\}$$
- The random variable H_n giving the height of a random skip list of n is the maximum of n i.i.d. Geom(p)
- Several performance measures of skip lists are expressed in terms of the probabilistic behavior of a sequence of n i.i.d. geometric r.v. of parameter p

Skip lists

- The height of the skip list S is the number of non-empty lists,
$$\text{height}(S) = \max_{x \in S}\{\text{height}(x)\}$$
- The random variable H_n giving the height of a random skip list of n is the maximum of n i.i.d. Geom(p)
- Several performance measures of skip lists are expressed in terms of the probabilistic behavior of a sequence of n i.i.d. geometric r.v. of parameter p

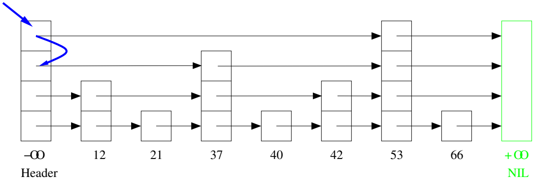
Searching in a skip list

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



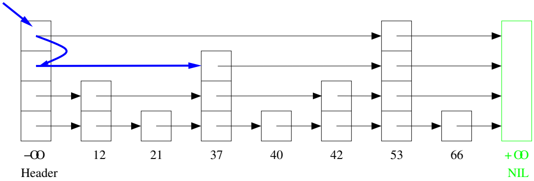
Searching in a skip list

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



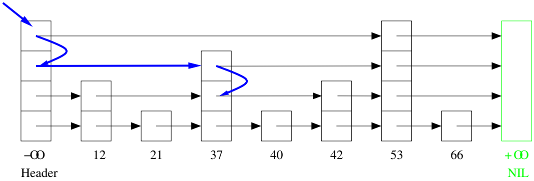
Searching in a skip list

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



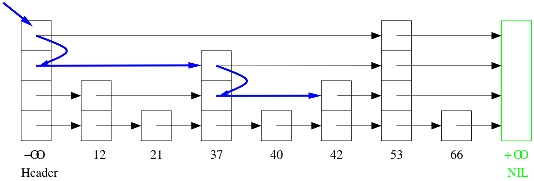
Searching in a skip list

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



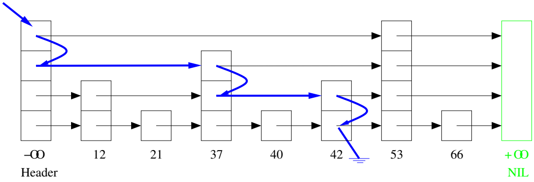
Searching in a skip list

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



Searching in a skip list

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

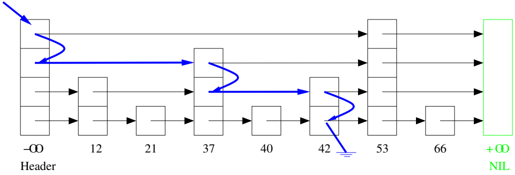


Searching in a skip list

```
procedure SEARCH(S, x)
    p ← S.header
    ℓ ← S.height
    while ℓ ≠ 0 do
        if p.item < x then
            p ← p.next[ℓ]
        else
            ℓ ← ℓ − 1
        end if
    end while
end procedure
```

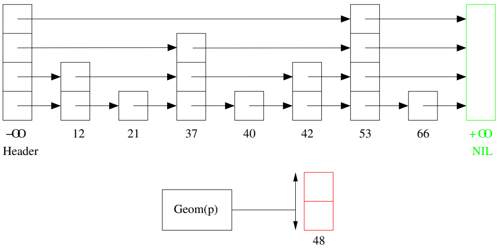
Insertion in a skip list

Inserting an item $x = 48$



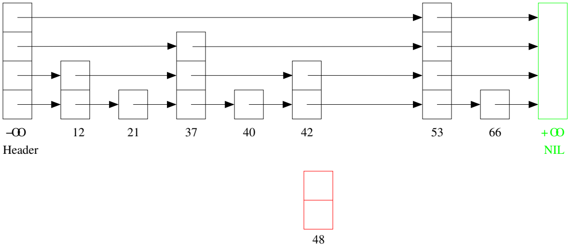
Insertion in a skip list

Inserting an item $x = 48$



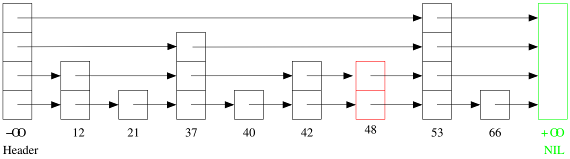
Insertion in a skip list

Inserting an item $x = 48$



Insertion in a skip list

Inserting an item $x = 48$



Implementing skip lists

```
template <typename Key, typename Value>
class Dictionary {
public:
    ...
private:
  struct node_skip_list {
     Key _k;
     Value _v;
     int _height;
     node_skip_list** _next;

     node_skip_list(const Key& k, const Value& v, int h) :
                    _k(k), _v(v), _height(h),
                    _next(new node_skip_list*[h]) {
     }
  };
  node_skip_list* _header;
  int _height;
  double _p; // e.g., _p = 0.5
  ...
};
```

Implementing skip lists

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

   node_skip_list* p =
       lookup_skip_list(_header, _height-1, k);
   if (p == nullptr)
      exists = false;
   else {
      exists = true;
      v = p -> _v;
   }
}

template <typename Key, typename Value>
Dictionary<Key,Value>::node_skip_list*
   Dictionary<Key,Value>::lookup_skip_list(
      node_skip_list* p,
      int l, const Key& k) const throw() {

   while (l >= 0)
      if  (p -> _next[l] == nullptr or k <= p ->_next[l] -> _k)
         --l;
      else
         p = p -> _next[l];

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

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

```
template <typename Key, typename Value>
void Dictionary<Key,Value>::insert_skip_list(...) {
   node_skip_list* p = _header;
   int l = _height - 1;
   node_skip_list** pred = new node_skip_list*[_height];
   while (l >= 0)
      if  (p -> _next[l] == nullptr or k <= p ->_next[l] -> _k) {
         pred[l] = p; // <====== keep track of predecessor at level l
         --l;
      } else {
         p = p -> _next[l];
      }

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

Implementing skip lists

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

Implementing skip lists

```
template <typename Key, typename Value>
class Dictionary {
public:
   ...
private:
   ...
   Random _rng; // associate a random number generator
                // to the skip list
};

template <typename Key, typename Value>
void Dictionary<Key,Value>::insert_skip_list(...) {

   ...
   // adding new node
   // generate random height
   int h = 1; while (_rng() > _p) ++h;
   node_skip_list* nn = new node_skip_list(k, v, h);
   if (h > _height) {
      // add new levels to the header
      // make pred[i] = _header for all i = _height .. h-1
      ...
   }

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

Implementing skip lists

```
...
if (h > _height) {
   node_skip_list** _new_header = new node_skip_list*[h];
   node_skip_list** new_pred = new node_skip_list*[h];

   // copying
   for (int i = _height - 1; i >= 0; --i) {
      _new_header -> _next[i] = _header -> _next[i];
      new_pred -> _next[i] = pred -> _next[i];
   }

   // empty upper levels
   for (int i = h - 1; i >= _height; --i) {
      _new_header -> _next[i] = nullptr;
      new_pred -> _next[i] = nullptr;
   }

   // delete old header and pred
   delete[] _header;
   delete[] pred;

   // update the skip list
   _header = _new_header;
   pred = new_pred;
   _height = h;
}
...
```

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

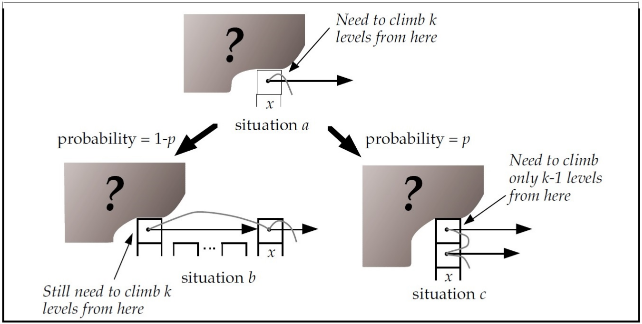


Figure from W. Pugh's *Skip Lists: A Probabilistic Alternative to Balanced Trees* (C. ACM, 1990)—the meaning of p is the opposite of what we have used!

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 level for which the expected number of nodes that have height $\geq L_n$ is $\leq 1/q$
- Probability that a node has height $\geq k$ is

$$\begin{aligned}\Pr\{\text{height}(x_i) \geq k\} &= \sum_{i \geq k} pq^{i-1} \\ &= pq^{k-1} \sum_{i \geq 0} q^i = q^{k-1}\end{aligned}$$

- Number of nodes with height $\geq k$ is a binomial r.v. with parameters n and q^{k-1}, hence the expected number is 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 level for which the expected number of nodes that have height $\geq L_n$ is $\leq 1/q$
- Probability that a node has height $\geq k$ is

$$\begin{aligned}\Pr\{\text{height}(x_i) \geq k\} &= \sum_{i \geq k} pq^{i-1} \\ &= pq^{k-1} \sum_{i \geq 0} q^i = q^{k-1}\end{aligned}$$

- Number of nodes with height $\geq k$ is a binomial r.v. with parameters n and q^{k-1}, hence the expected number is 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 level for which the expected number of nodes that have height $\geq L_n$ is $\leq 1/q$
- Probability that a node has height $\geq k$ is

$$\begin{aligned} \Pr\{\text{height}(x_i) \geq k\} &= \sum_{i \geq k} pq^{i-1} \\ &= pq^{k-1} \sum_{i \geq 0} q^i = q^{k-1} \end{aligned}$$

- Number of nodes with height $\geq k$ is a binomial r.v. with parameters n and q^{k-1}, hence the expected number is 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 level for which the expected number of nodes that have height $\geq L_n$ is $\leq 1/q$
- Probability that a node has height $\geq k$ is

$$\begin{aligned}\Pr\{\text{height}(x_i) \geq k\} &= \sum_{i \geq k} pq^{i-1} \\ &= pq^{k-1} \sum_{i \geq 0} q^i = q^{k-1}\end{aligned}$$

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

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

Performance of skip lists

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

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

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

- it follows that

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

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

Performance of skip lists

Summing up, the expected path length of a search is

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

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

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

A more refined analysis

- The cost of insertions, deletions and searches is essentially that of searching, with

$$\text{Cost of search} = \text{\# of forward steps} + \text{height}(S)$$

- More formally, with $X = \{x_1, x_2, \ldots, x_n\}$, $x_0 = -\infty < x_1 < \cdots < x_n < x_{n+1} = +\infty$, for $0 \leq k \leq n$,

$$C_{n,k} = F_{n,k} + H_n \qquad \text{cost of searching a key in } (x_k, x_{k+1}]$$

$$F_{n,k} = \text{\# of forward steps to } (x_k, x_{k+1}]$$

$$H_n = \text{height of the skip list}$$

A more refined analysis

- The cost of insertions, deletions and searches is essentially that of searching, with

$$\text{Cost of search} = \text{\# of forward steps} + \text{height}(S)$$

- More formally, with $X = \{x_1, x_2, \ldots, x_n\}$, $x_0 = -\infty < x_1 < \cdots < x_n < x_{n+1} = +\infty$, for $0 \le k \le n$,

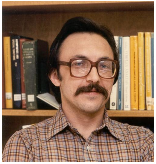
$$\begin{aligned} C_{n,k} &= F_{n,k} + H_n \qquad \text{cost of searching a key in } (x_k, x_{k+1}] \\ F_{n,k} &= \text{\# of forward steps to } (x_k, x_{k+1}] \\ H_n &= \text{height of the skip list} \end{aligned}$$

Analysis of the height

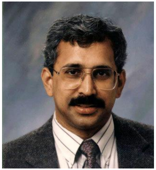
$$
\begin{aligned}
a_i &= \text{height}(x_i) \sim Geom(p) \\
H_n &= \text{height}(S) = \max\{a_1, \dots, a_n\} \\
\mathbb{E}[H_n] &= \sum_{k>0} \Pr\{H_n > k\} = \sum_{k>0} (1 - \Pr\{H_n \le k\}) \\
&= \sum_{k>0} \left(1 - \prod_{1 \le i \le n} \Pr\{a_i \le k\}\right) = \sum_{k>0} \left(1 - (\Pr\{a_i \le k\})^n\right) \\
&= \sum_{k>0} \left(1 - \left(1 - q^k\right)^n\right)
\end{aligned}
$$

with $q := 1 - p$.

Analysis of the height



W. Szpankowski



V. Rego

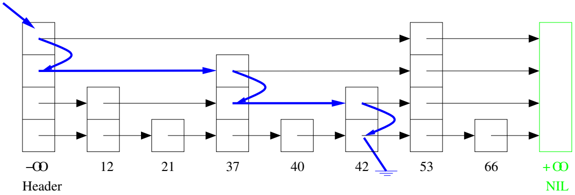
Theorem (Szpankowski and Rego,1990)

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

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

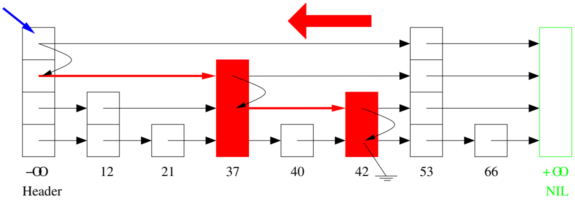
Analysis of the forward cost

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



Analysis of the forward cost

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



Analysis of the forward cost

- Total unsuccessful search cost

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

- Total forward cost

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

Analysis of the forward cost

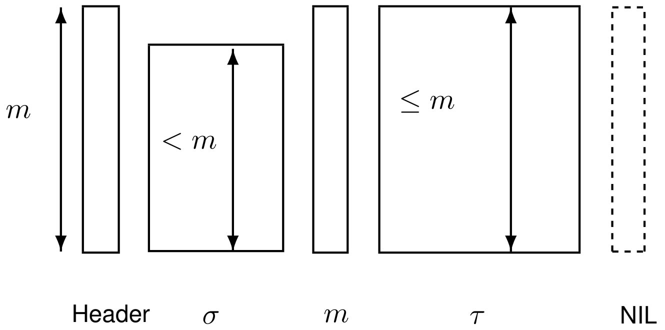
- Total unsuccessful search cost

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

- Total forward cost

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

Analysis of the forward cost



A recursive decomposition of the skip list S

Analysis of the forward cost

- $F(S) =$ total forward cost of the skip list S
- The recursive decomposition $S = \langle \sigma, m, \tau \rangle$ gives

$$F(S) = F(\sigma) + F(\tau) + |\tau| + 1$$

- Let $\mathcal{S}^{[\text{cond}]}$ denote the set of all skip lists whose height satisfies the condition *cond*

$$F^{[\text{cond}]}(z, u) = \sum_{S \in \mathcal{S}^{[\text{cond}]}} z^{|S|} u^{F(S)} \Pr(S),$$

with

$$\Pr(S) = \Pr(\sigma) \cdot pq^{m-1} \cdot \Pr(\tau)$$

Analysis of the forward cost

- $F(S) =$ total forward cost of the skip list S
- The recursive decomposition $S = \langle \sigma, m, \tau \rangle$ gives

$$F(S) = F(\sigma) + F(\tau) + |\tau| + 1$$

- Let $\mathcal{S}^{[\text{cond}]}$ denote the set of all skip lists whose height satisfies the condition *cond*

$$F^{[\text{cond}]}(z, u) = \sum_{S \in \mathcal{S}^{[\text{cond}]}} z^{|S|} u^{F(S)} \Pr(S),$$

with

$$\Pr(S) = \Pr(\sigma) \cdot pq^{m-1} \cdot \Pr(\tau)$$

Analysis of the forward cost

- $F(S) =$ total forward cost of the skip list S
- The recursive decomposition $S = \langle \sigma, m, \tau \rangle$ gives

$$F(S) = F(\sigma) + F(\tau) + |\tau| + 1$$

- Let $\mathcal{S}^{[\text{cond}]}$ denote the set of all skip lists whose height satisfies the condition *cond*

$$F^{[\text{cond}]}(z, u) = \sum_{S \in \mathcal{S}^{[\text{cond}]}} z^{|S|} u^{F(S)} \Pr(S),$$

with

$$\Pr(S) = \Pr(\sigma) \cdot pq^{m-1} \cdot \Pr(\tau)$$

Analysis of the forward cost

- The recursion translates to

$$F^{=m}(z,u) = pq^{m-1}zu^2F^{\leq m-1}(z,u)F^{\leq m}(z,u), \qquad m>0$$
$$F^{=0}(z,u) = 1$$

- Taking derivatives w.r.t. u and setting $u=1$, we obtain a recurrence for the GF of expectations:

$$f^{=m}(z) = \frac{2pq^{m-1}z}{[\![m-1]\!][\![m]\!]} + \frac{f^{\leq m-1}(z)}{[\![m]\!]} + \frac{f^{\leq m}(z)}{[\![m-1]\!]},$$

with $[\![m]\!] := 1 - z(1-q^m)$

Analysis of the forward cost

- The recursion translates to

$$F^{=m}(z,u) = pq^{m-1} z u^2 F^{\leq m-1}(z,u) F^{\leq m}(z,u), \qquad m > 0$$
$$F^{=0}(z,u) = 1$$

- Taking derivatives w.r.t. u and setting $u = 1$, we obtain a recurrence for the GF of expectations:

$$f^{=m}(z) = \frac{2pq^{m-1}z}{[\![m-1]\!][\![m]\!]} + \frac{f^{\leq m-1}(z)}{[\![m]\!]} + \frac{f^{\leq m}(z)}{[\![m-1]\!]},$$

with $[\![m]\!] := 1 - z(1 - q^m)$

Analysis of the forward cost

- We solve the recurrence by iteration, with $f^{=m} = f^{\leq m} - f^{\leq m-1}$ and finally take the limit $f(z) := \lim_{m\to\infty} f^{\leq m}(z)$

$$f(z) = \frac{z^2}{(1-z)^2} \sum_{i\geq 1} \frac{pq^{i-1}(1-q^i)}{[\![i]\!]}$$

- Using Euler transform we can easily extract the nth coefficient of $f(z)$, $[z^n]f(z) = \mathbb{E}[F_n]$

$$\mathbb{E}[F_n] = \frac{p}{q} \sum_{k=2}^{n} \binom{n}{k} (-1)^k \frac{1}{Q^{k-1}-1},$$

$$q := 1-p, Q := 1/q$$

Analysis of the forward cost

- We solve the recurrence by iteration, with $f^{=m} = f^{\leq m} - f^{\leq m-1}$ and finally take the limit $f(z) := \lim_{m\to\infty} f^{\leq m}(z)$

$$f(z) = \frac{z^2}{(1-z)^2} \sum_{i\geq 1} \frac{pq^{i-1}(1-q^i)}{[\![i]\!]}$$

- Using Euler transform we can easily extract the nth coefficient of $f(z)$, $[z^n]f(z) = \mathbb{E}[F_n]$

$$\mathbb{E}[F_n] = \frac{p}{q} \sum_{k=2}^{n} \binom{n}{k} (-1)^k \frac{1}{Q^{k-1}-1},$$

$$q := 1-p, Q := 1/q$$

Analysis of the forward cost

The asymptotic behavior of F_n (and other quantities that arise in the analysis of skip lists) can be analyzed using Mellin transforms or Rice's method

$$\sum_{k=a}^{n} \binom{n}{k} (-1)^k f(k) = -\frac{1}{2\pi\mathbf{i}} \int_{\mathcal{C}} \frac{\Gamma(n+1)\Gamma(-z)}{\Gamma(n+1-z)} f(z)\, dz$$

with $\mathcal{C}$ a positively oriented curve enclosing a, $a+1$, ..., n, and $f(z)$ an analytic continuation of $f(k)$

Analysis of the forward cost

P. Kirschenhofer

H. Prodinger

Theorem (Kirschehofer, Prodinger, 1994)

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

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

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

To learn more

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

To learn more (2)

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

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

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

1. Hashing
2. Skip lists
3. Binomial Queues

Binomial Queues

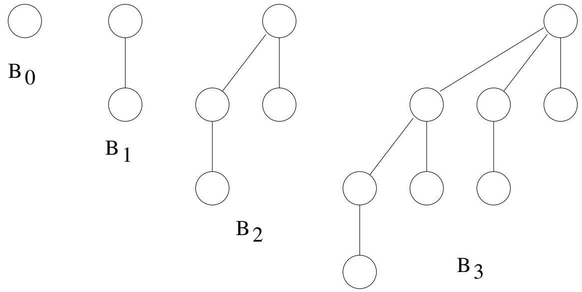
J. Vuillemin

- 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.

```cpp
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();

  ...
};
```

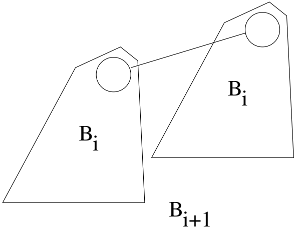
Binomial Queues

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



Binomial Queues

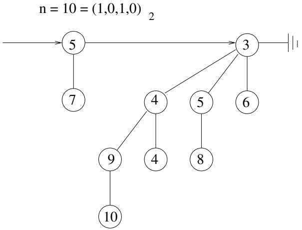
- A binomial tree of order $i + 1$ is (recursively) built by planting a binomial tree B_i as a child of the root of another binomial tree B_i.



- 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, ...

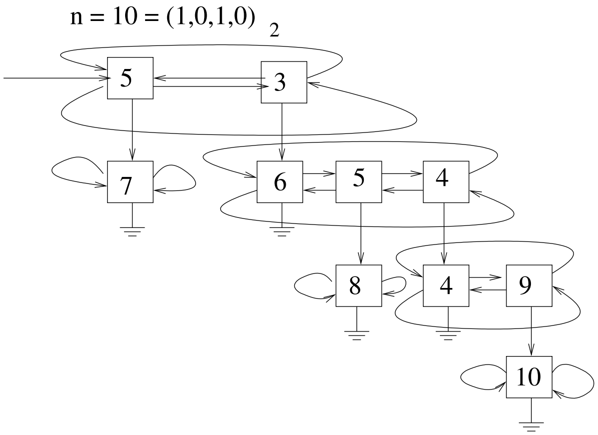


- A binomial queue for n elements contains at most $\lceil \log_2(n+1) \rceil$ binomial trees
- The n elements of the binomial queue are stored in the binomial trees in such a way that each binomial tree satisfies the heap property: the priority of the element at any given node is $\leq$ than the priority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

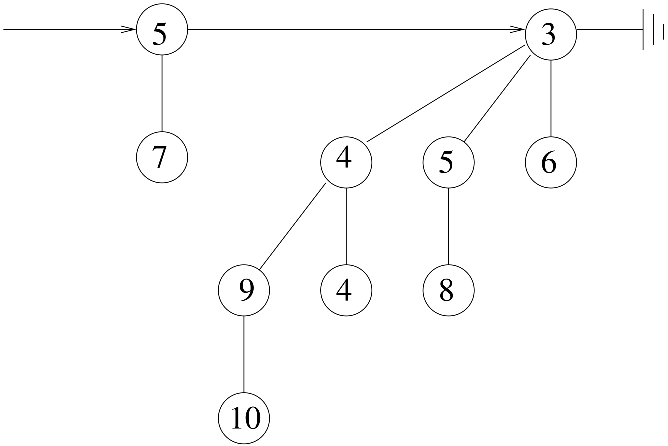
```
void insert(const Elem& x, const Prio& p) throw(error) {
    node_bq* nn = new node_bq(x, p);
    _first = meld(_first, nn);
    ++_nelems;
}
```

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'

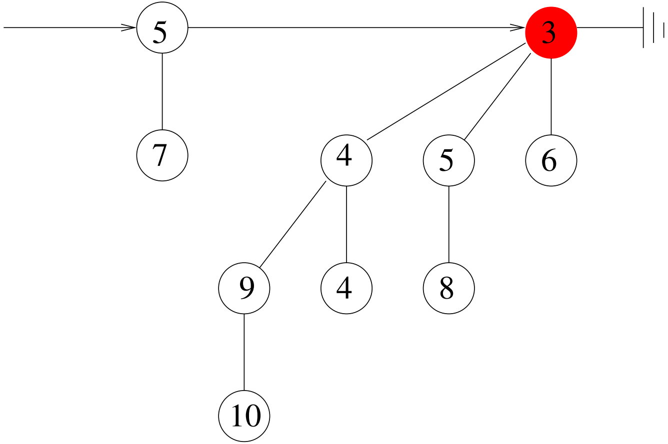
Binomial Queues

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



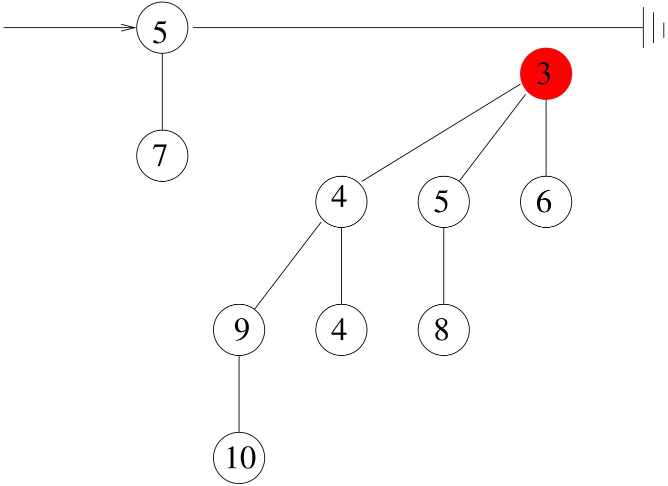
Binomial Queues

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

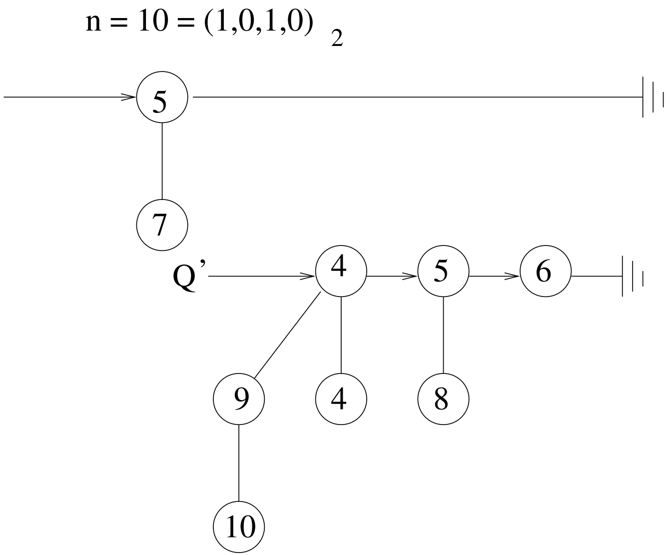


Binomial Queues

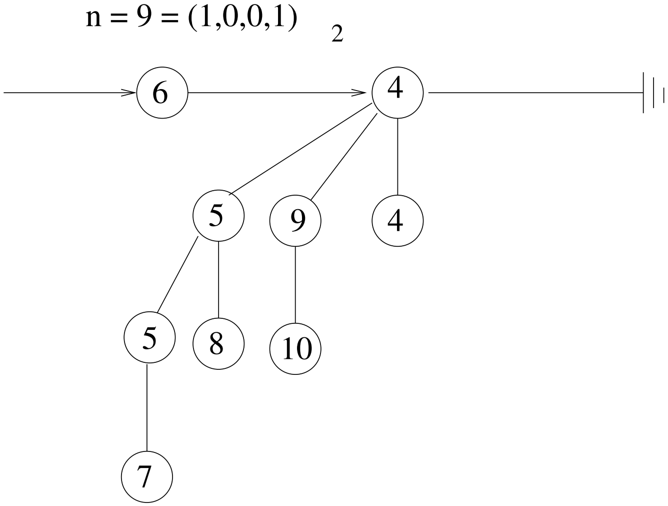
$n = 10 = (1,0,1,0)_2$



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

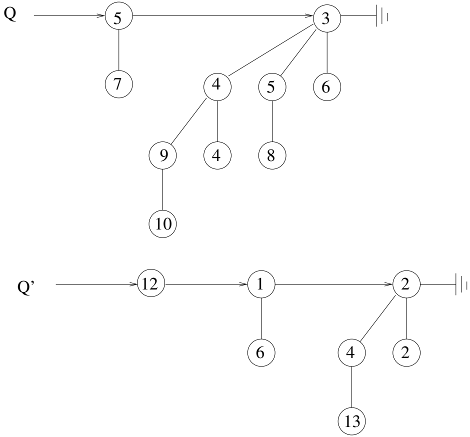
Binomial Queues

- 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 $C = \emptyset$

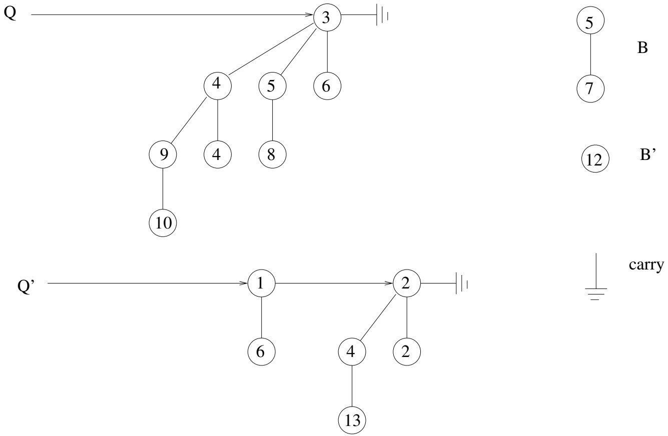
Binomial Queues

- 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

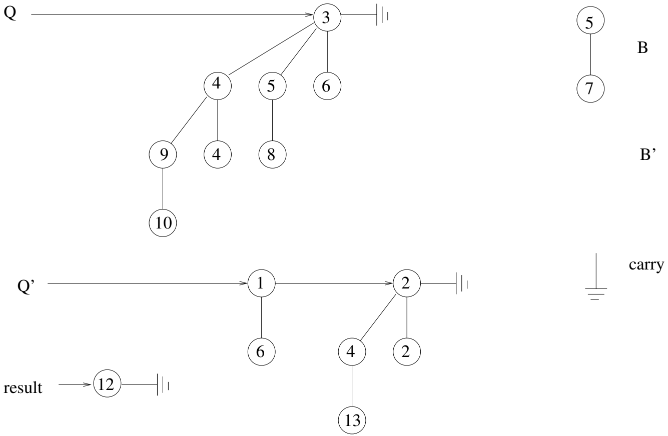
Binomial Queues



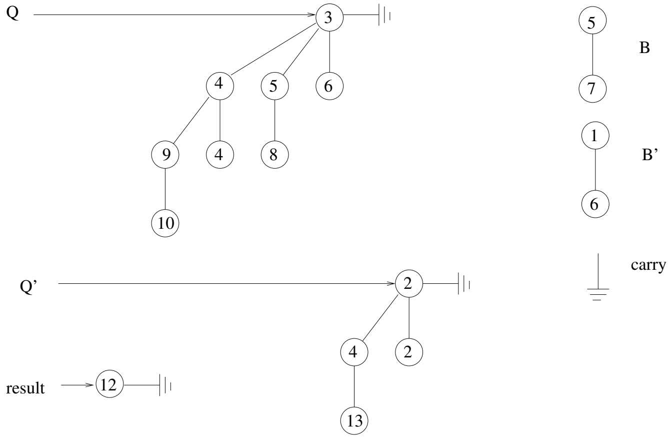
Binomial Queues



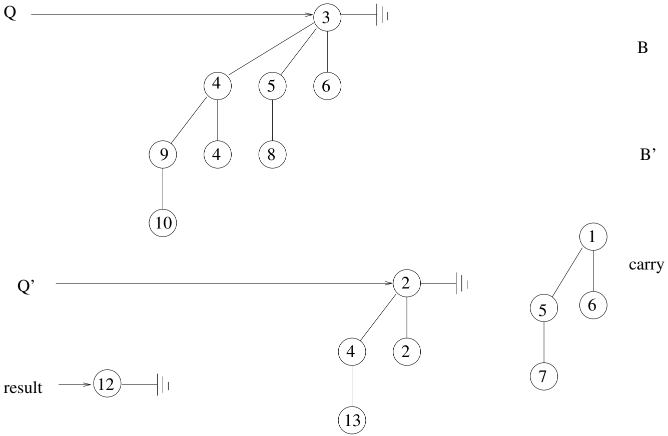
Binomial Queues



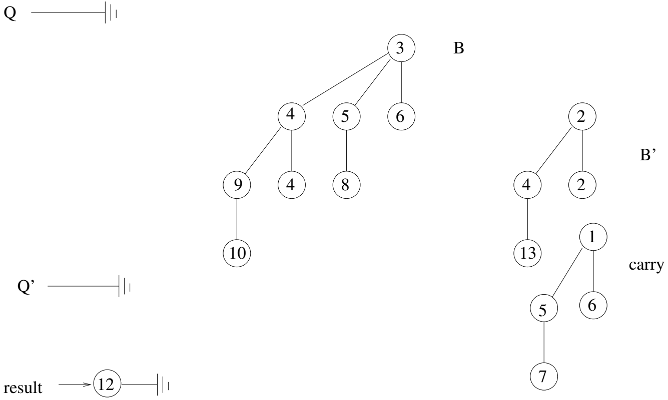
Binomial Queues



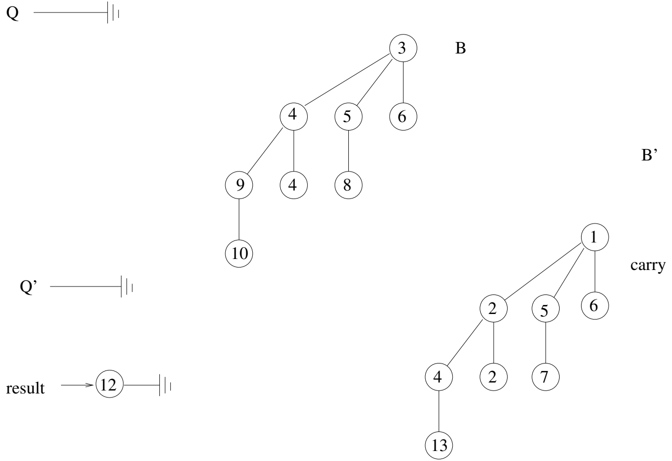
Binomial Queues



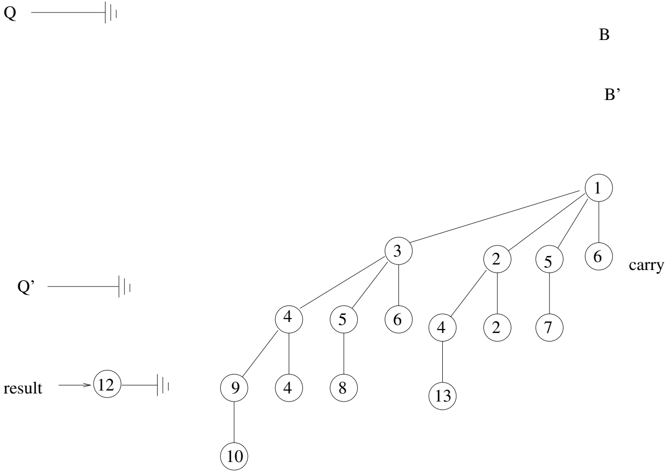
Binomial Queues



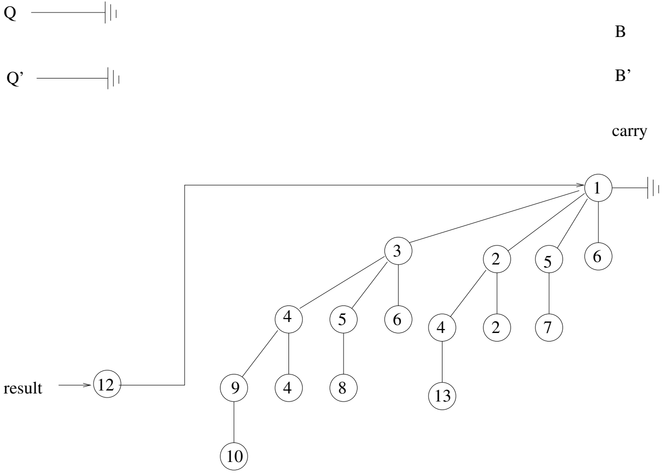
Binomial Queues



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

$$\begin{aligned}\sum_{0\le i<n} \Theta(\ell_i + 1) &= \sum_{r=1}^{\lceil \log_2(n+1)\rceil} \Theta(r)\cdot \frac{n}{2^r}\\ &\le n\Theta\left(\sum_{r\ge 0}\frac{r}{2^r}\right) = \Theta(n),\end{aligned}$$

  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.