Jubilee lecture, October 2004/January 2005

Title:

Anatomy of a worst-case efficient priority queue

Speaker:

Jyrki Katajainen

Co-workers:

Amr Elmasry and Claus Jensen

These slides are available at

`http://www.cphstl.dk/`.

Priority Queues

Types

$\mathcal{E}$: elements manipulated

$\mathcal{C}$: compartments where elements are stored

$\mathcal{F}$: ordering used in element comparisons

$\mathcal{A}$: allocator used in memory management

Assumptions

- The elements are only moved and compared, both operations having a cost of $O(1)$.
- It is possible to get any information stored at a compartment at a cost of $O(1)$.
- Both allocation and deallocation have a cost of $O(1)$.

Priority-Queue Operations

Let Q be a priority queue with type parameters $\langle \mathcal{E}, \mathcal{C}, \mathcal{F}, \mathcal{A} \rangle$.

$\mathcal{E}$ **find-min():** Return a minimum element stored in Q. The minimum is taken with respect to $\mathcal{F}$.

$\mathcal{C}$ **insert(**$\mathcal{E}$ e**):** Insert element e into Q and return its compartment for later use.

void **delete-min():** Remove a minimum element and its compartment from Q.

void **delete(**$\mathcal{C}$ p**):** Remove both the element stored at compartment p and p from Q.

void **decrease(**$\mathcal{C}$ p**,** $\mathcal{E}$ e**):** Replace the element stored at compartment p with a smaller element e.

void **unite(**priority queue $\langle \mathcal{E}, \mathcal{C}, \mathcal{F}, \mathcal{A} \rangle$ R**):** Move all elements stored in R to Q.

Some additional operations like a constructor, a destructor, empty(), and size() are necessary to make the data structure useful.

Warning

When an element is inserted, the reference to the compartment, where it is stored, should remain the same so that possible later references made by delete and decrease operations are valid.

Our solution to this potential problem is simple: we do not move the elements after they have been inserted into the data structure.

In the C++ standard this is called **iterator validity**.

Comparison of Priority Queues

operation	binary heap worst case	Fibonacci heap amortized	pruned binomial queue worst case
find-min	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
insert	$\Theta(\lg n)$	$\Theta(1)$	$\Theta(1)$
delete-min	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(\lg n)$
delete	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(\lg n)$
decrease	$\Theta(\lg n)$	$\Theta(1)$	$\Theta(1)$
unite	$\Theta(n)$	$\Theta(1)$	$\Theta(\lg n)$

n denotes the number of elements in the data structure just prior to the operation.

C++ Standard

The following complexity requirements — no other time or space bounds — are given.

find-min(): constant time

insert(): at most $\lg n$ element comparisons

delete-min(): at most $2 \lg n$ element comparisons

general constructor: at most $3n$ element comparisons

priority-queue sort: at most $n \lg n$ element comparisons.

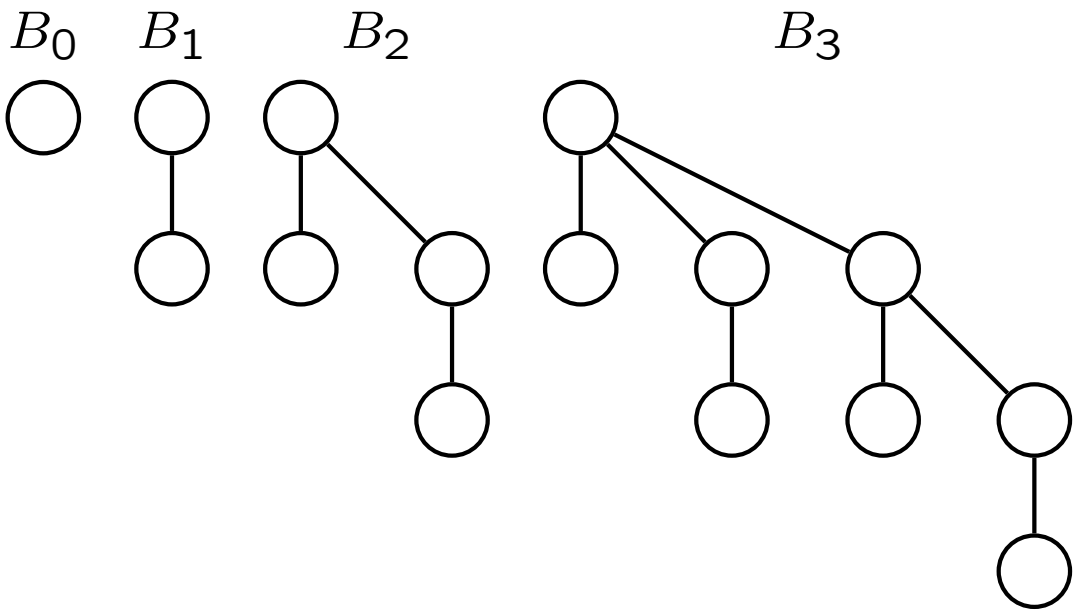
However, the standard does not demand any compulsory support for

- external references,
- delete(), or
- decrease().

Binomial Trees

A **binomial tree** B_k, $k \geq 0$, is a rooted, ordered tree defined recursively as follows:

1. B_0 consists of a single node.
2. For $k > 0$, B_k comprises the root and its k binomial subtrees $B_0, \ldots, B_{k-1}$ in this order.



Properties of Binomial Trees

Fact: A binomial tree B_k contains 2^k nodes.

Proof: By induction on k. ∎

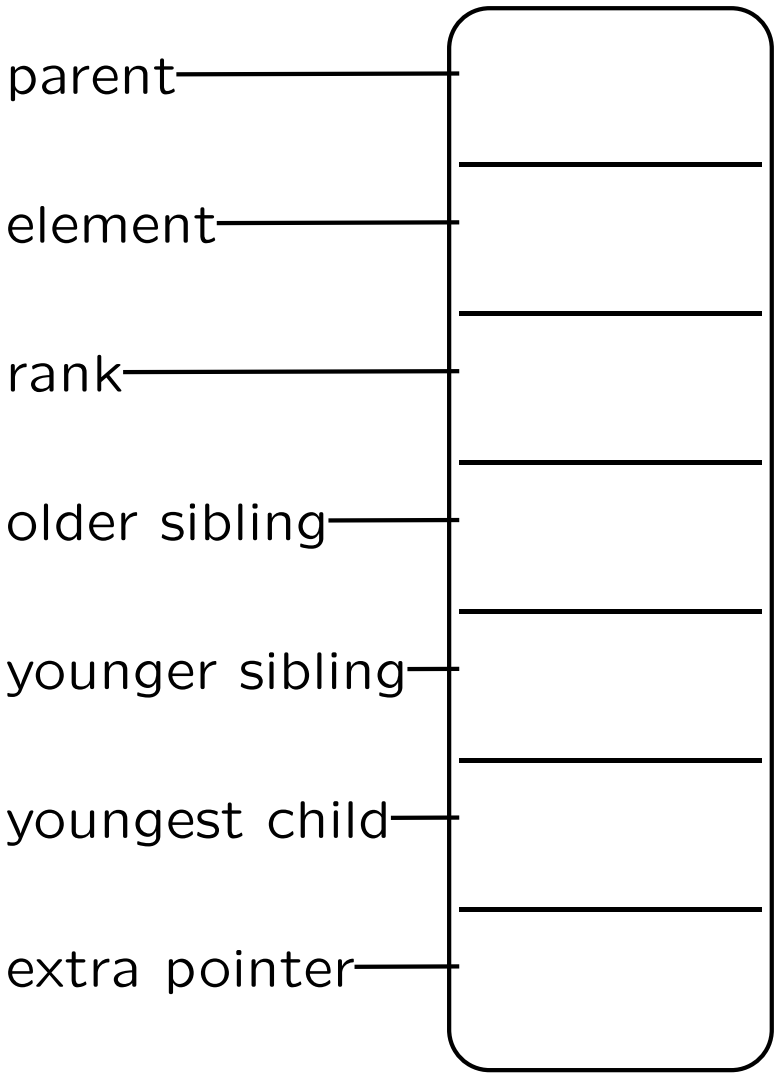
Fact: The height of B_k is k.

Proof: By induction on k. ∎

Fact: The number of nodes $n_k(j)$ at level j in B_k, $j \in \{0, \ldots, k\}$, is given by the binomial coefficient $\binom{k}{j}$.

Proof: By induction on k. ∎

Representing a Binomial Tree



- The parent pointer of a root points to a fixed sentinel.
- For the youngest child of a node one of the sibling pointers points to the oldest child, and vice versa.
- Unused child pointers have the value null.

Binomial Queues

A **binomial queue** Q storing n elements is a collection of binomial trees with the following properties:

1. Consider the binary representation of n

$$n = \sum_{i=0}^{\lfloor \lg n \rfloor} b_i 2^i ,$$

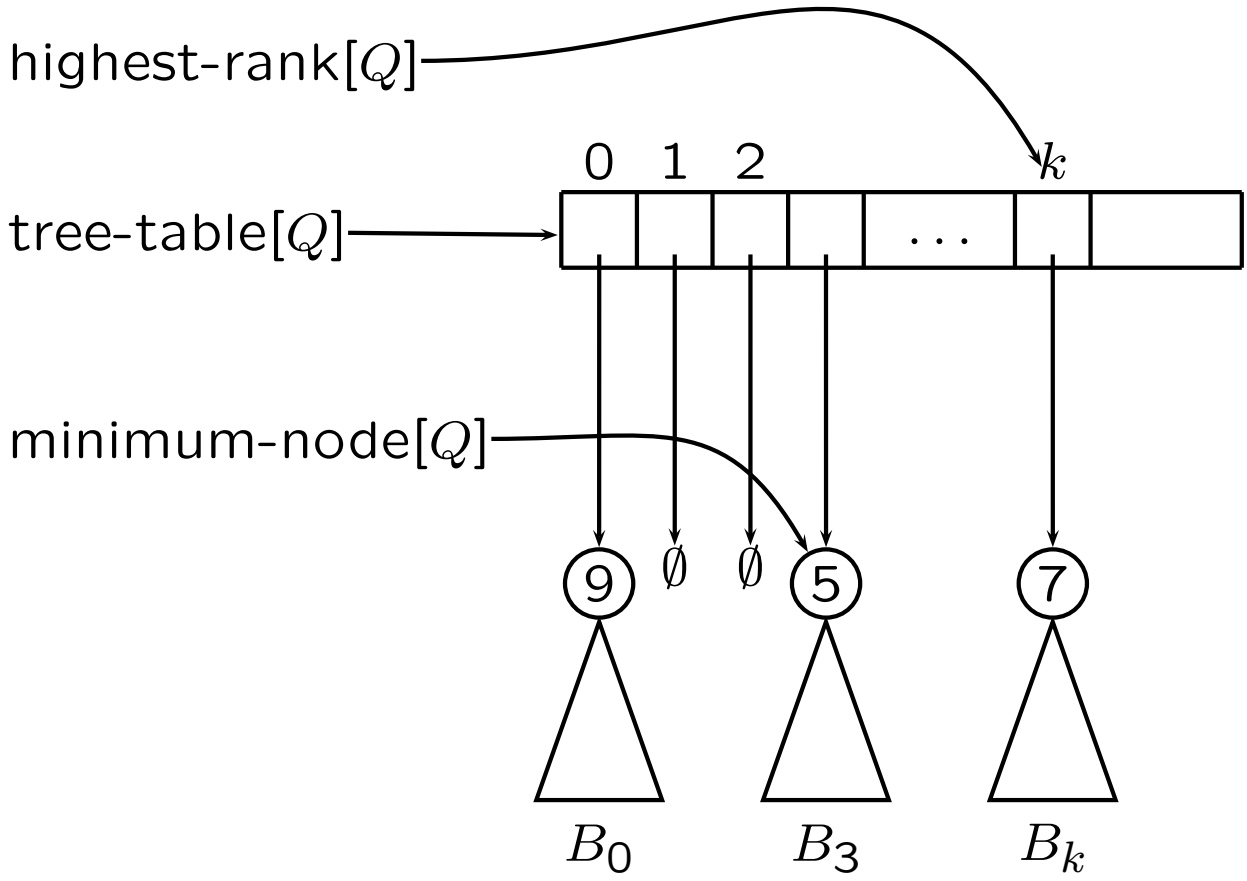
where $b_i \in \{0, 1\}$ for all $i \in \{0, \ldots, \lfloor \lg n \rfloor\}$. A binomial tree B_i is in Q if and only if $b_i = 1$, i.e., Q is a forest of binomial trees

$$F_n = \{B_i \mid n = \sum_{i=0}^{\lfloor \lg n \rfloor} b_i 2^i \text{ and } b_i = 1\}.$$

2. Each node stores exactly one element.

3. Each binomial tree is **heap ordered**, i.e. the element stored at a node is no greater than the elements stored at the children of that node.

Representing a Binomial Queue



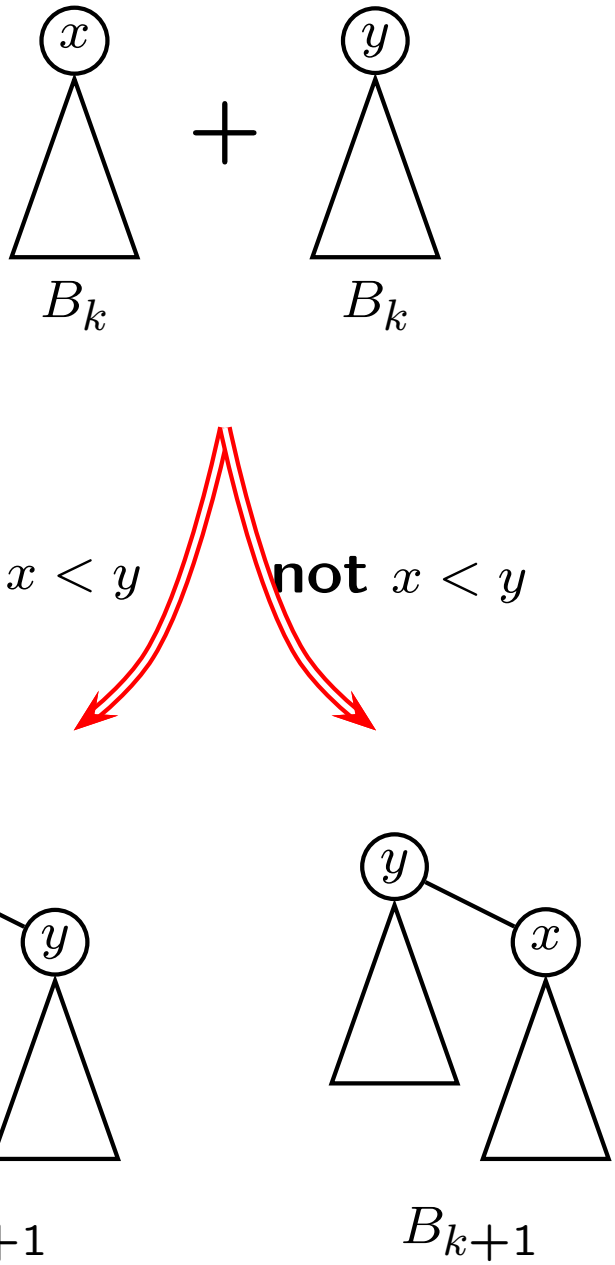
The **tree table** is a resizable array which must support growing and shrinking at the tail at the worst-case cost of $O(1)$.

find-min()

Follow the pointer to the minimum node and return the element stored there.

Worst-case cost: $\Theta(1)$

Joining Two Binomial Trees



Worst-case cost: $\Theta(1)$

insert()

1. Create a new B_0 and put the given element there.
2. Correct the minimum-node pointer to point to the new node if the new element is smaller than the element stored at the node pointed to by it.
3. Do binary addition (see illustration below).

				B_1		
	B_4	B_3	$\emptyset$	B_1	B_0	F_{27}
$+$					B_0	F_1
	B_4	B_3	B_2	$\emptyset$	$\emptyset$	F_{28}

Worst-case cost: $\Theta(\lg n)$ because of the carry

1. How to reduce the cost to $\Theta(1)$?
2. How to make it possible to insert trees of arbitrary rank into the structure?

Solution

Represent integer n in a **redundant binary system** such that

$$n = \sum_{i=0}^{\lfloor \lg n \rfloor} d_i 2^i$$

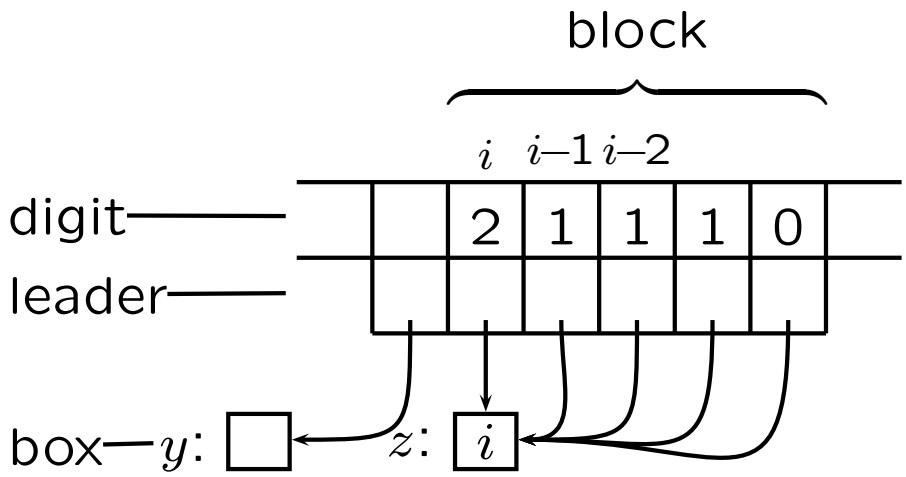
and $d_i \in \{0, 1, 2\}$ for all $i \in \{0, 1, \ldots, \lfloor \lg n \rfloor\}$.

Moreover, keep the representation **regular** such that any 2 is preceded by one 0, possibly having a sequence of 1s in between.

Do at most one join per insert and give a preference for a join involving small trees.

$$\begin{array}{cccccccccc}
 & & & B_2 & & & & & & \\
 & B_4 & \emptyset & B_2 & \emptyset & B_0 & & & F_{25} & \\
+ & & & & & B_0 & & & F_1 & \\
\hline
 & B_4 & \emptyset & B_2 & B_1 & \emptyset & & & F_{26} & \\
 & & & B_2 & & & & & &
\end{array}$$

Guides



A **guide** supports three operations:

void **fix-up($\mathcal{N}$ i):** $digit[i] \leftarrow 0$; $digit[i+1]$++
(Precondition: $digit[i] = 2$)

Case 1: `*y` = `null`

$*z \leftarrow null$;
$digit[i] \leftarrow 0$;
$digit[i+1]$++;
if $digit[i+1] = 2$:
  $x \leftarrow$ new box;
  $*x \leftarrow i+1$;
  $leader[i] \leftarrow x$;
  $leader[i+1] \leftarrow x$;

Case 2: `*y` $\neq$ `null`

$*z \leftarrow null$;
$digit[i] \leftarrow 0$;
$digit[i+1]$++;
$leader[i] \leftarrow y$;

void **increment($\mathcal{N}$ i):** $digit[i]$++ $(digit[i] < 2)$

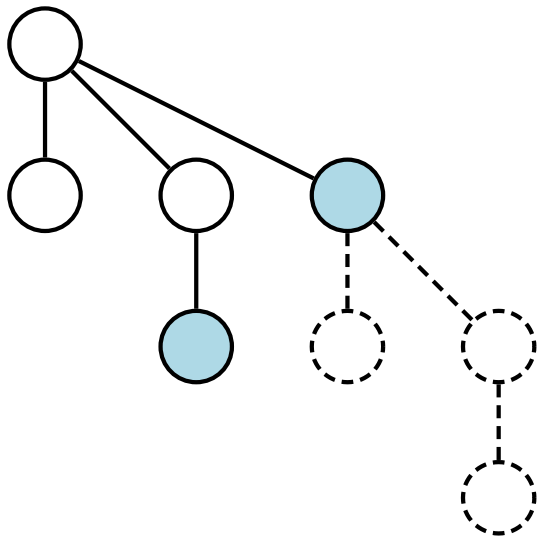
void **decrement($\mathcal{N}$ i):** $digit[i]$-- $(digit[i] > 0)$

All operations fix-up(), increment(), and decrement() have a cost of $O(1)$.

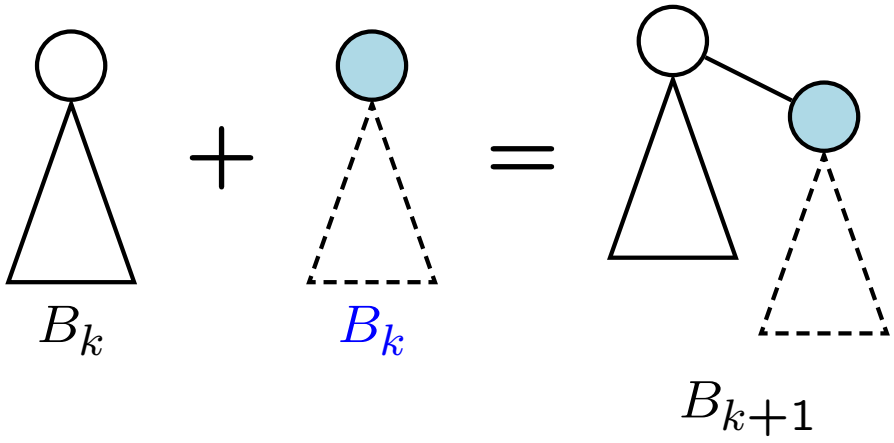
Pruned Binomial Trees

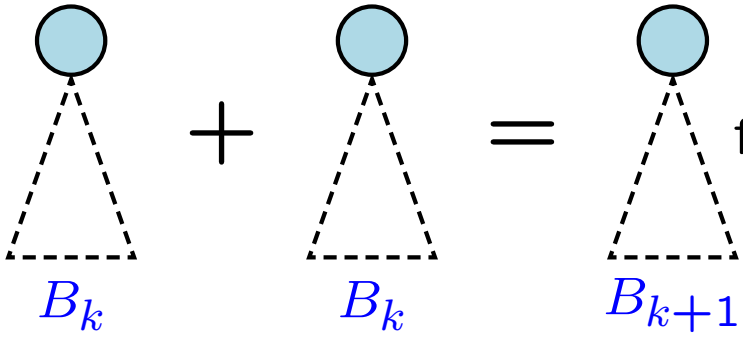
Some nodes may have lost some of their children. Technically, this can be handled by storing a **phantom node** in the place of any missing node. To distinguish a phantom node from the other nodes, its child pointer points to the node itself.

Example: B_3 with two phantom nodes

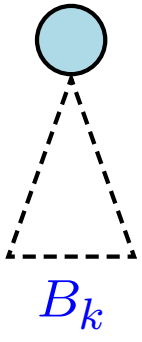


Phantom Arithmetic





free the other node



becomes a root; free the node

Local Violation Rule

1. Make sure that a node may have lost its last child (if any).
2. Allow between zero and two trees of each rank, i.e. use a guide to keep track of the trees.

A forest of pruned binomial trees fulfilling these properties is called a **thin binomial queue**.

Fact: In a thin binomial queue storing n elements, the rank of a tree can never be larger than $1.44 \lg n$.

Proof: As for Fibonacci heaps. ∎

Also, a decrease operation can be supported so that is has an amortized cost of $\Theta(1)$.

Global Violation Rule

1. Restrict the **total** number of phantom nodes to be no larger than $\lceil \lg n \rceil + 1$, n being the number of elements stored.
2. Allow between zero and two trees of each rank, i.e. use a guide to keep track of the trees.

A forest of pruned binomial trees fulfilling these properties is called a **pruned binomial queue**.

Some Properties

Fact: In a pruned binomial queue storing n elements, the rank of a tree can never be higher than $2 \lg n + O(1)$.

Proof: Trivial. ∎

Fact: A pruned binomial queue storing n elements can never contain more than $2 \lg n + O(1)$ trees.

Proof: Follows from the definition of a regular counter. ∎

Fact: In a pruned binomial queue storing n elements, the root of any subtree can never have more than $\lg n + O(\sqrt{\lg n})$ real children.

Proof: Painful. ∎

Bookkeeping of Phantom Nodes

A **run** is a maximal sequence of two or more neighbouring phantom nodes. A **singleton** is a phantom node that is not in a run. To keep track of the phantom nodes, a **run-singleton structure** is maintained.

Singleton table: A resizable array accessed by rank. Singletons of the same rank are kept in a list.

Pair list: A list of ranks that have more than two singletons.

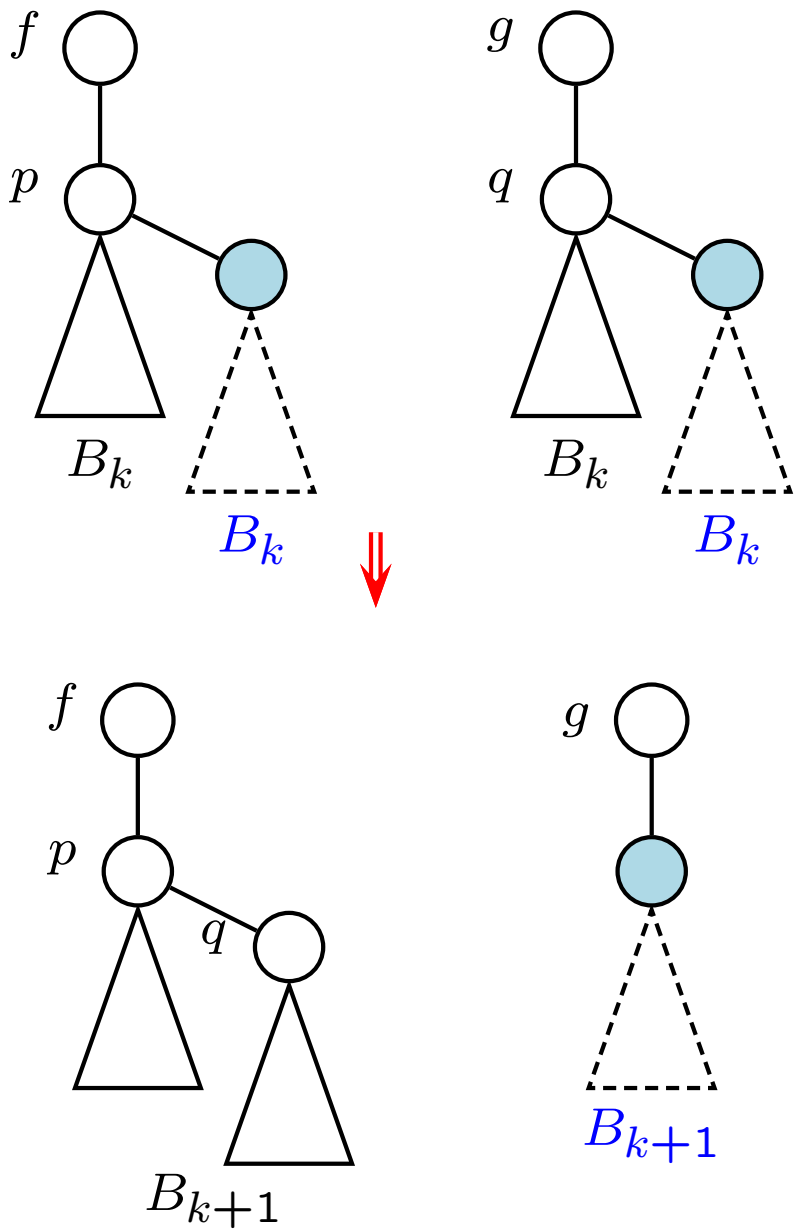
Run list: A list of phantom nodes that are the youngest in their respective run. None of the phantom nodes in a run are in the singleton table.

All lists are doubly linked, and each phantom node should have a pointer to its occurrence in a list (if any).

Removing Phantom Nodes

Let λ denote the number of phantom nodes. There are 8 transformations that are used to reduce λ if $\lambda > \lceil \lg n \rceil + 1$. The cost of each transformation is $O(1)$.

Example: Singleton transformation I



decrease()

1. Make the element replacement.
2. If the heap order is violated, cut off the subtree rooted at the given node.
3. Put a phantom node in the place of the given node.
4. Add the tree cut off to the guide.
5. Correct the minimum-node pointer if necessary.
6. Reduce λ if necessary.

Worst-case cost: $\Theta(1)$

delete-min()

1. Remove the root of the tree that contains a minimum element.
2. Join the subtrees of the root with B_0.
3. Scan through all roots to update the pointer to a minimum node if necessary.
4. Make λ smaller if it is too large.

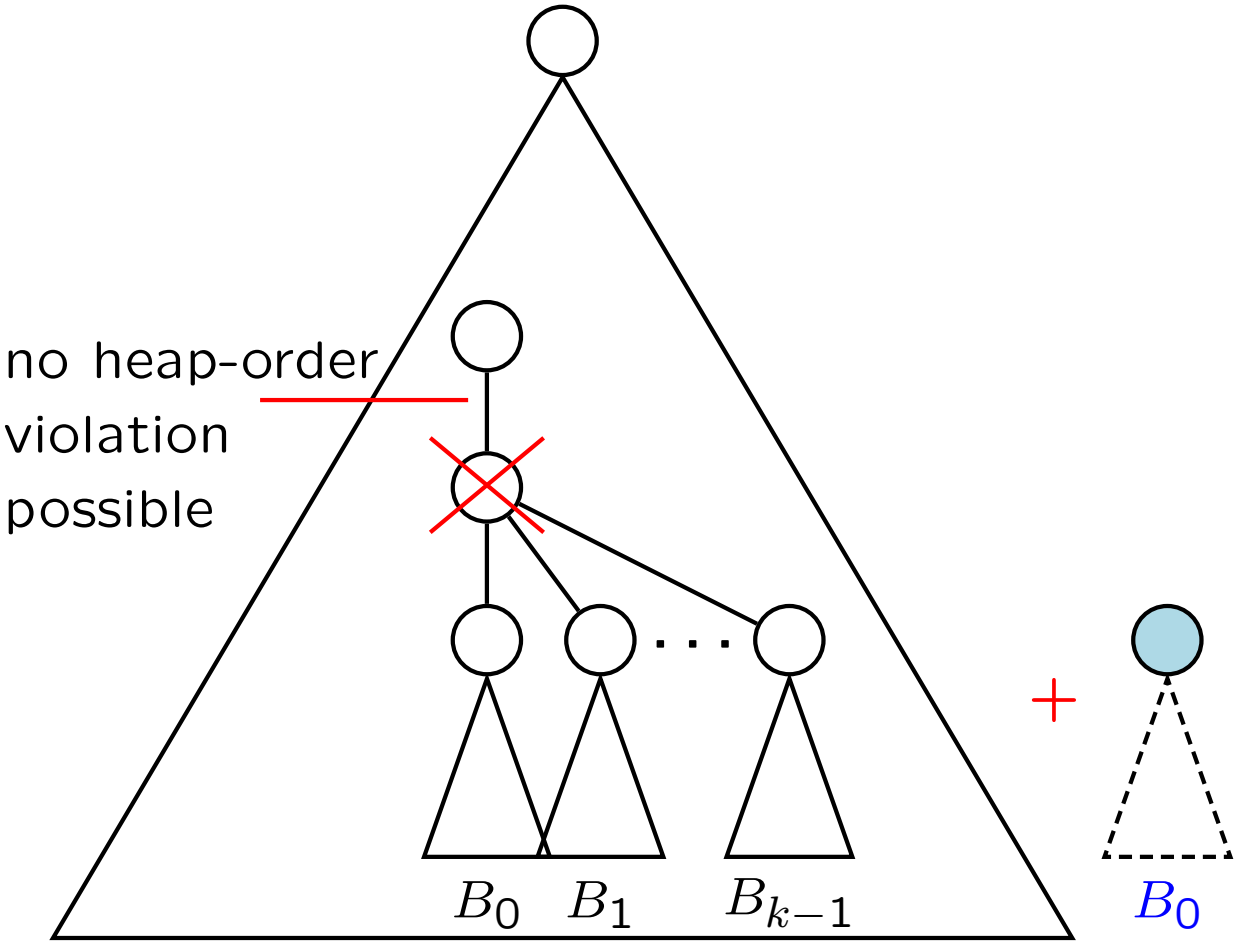
$$
\begin{array}{cccccccc}
 & B_4 & B_3 & \emptyset & B_1 & B_0 & & F_{27} \\
- & & B_3 & \emptyset & \emptyset & \emptyset & & F_8 \\
\hline
 & B_4 & \emptyset & \emptyset & B_1 & B_0 & & F_{19}
\end{array}
$$

$$
\begin{array}{cccccccc}
 & & B_3 & B_2 & B_1 & & & \\
 & B_4 & \emptyset & \emptyset & B_1 & B_0 & & F_{19} \\
 & & & B_2 & B_1 & B_0 & & F_7 \\
+ & & & & & B_0 & & F_1 \\
\hline
 & B_4 & B_3 & \emptyset & B_1 & B_0 & & F_{27}
\end{array}
$$

Worst-case cost: $\Theta(\lg n)$ with at most $3 \lg n + O(\sqrt{\lg n})$ element comparisons.

delete()

Identical to delete-min().



Worst-case cost: $\Theta(\lg n)$ with at most $3 \lg n + O(\sqrt{\lg n})$ element comparisons.

unite()

1. Merge the run-singleton structures of the two priority queues.
2. Insert the trees of the other priority queue one by one into Q,
3. Reduce λ until it is small enough.

Worst-case cost: $\Theta(\lg n)$

Conclusions

- In a pruned binomial queue, the complexity is hidden in the guide, in the run-singleton structure, and in the phantom-removing transformations.
- Hopefully, you agree that algorithmists have a good sense of humour.
- Hopefully, you understand why pruned binomial queues are not in your favourite program library.