6

Priority Queues

The company TMG markets tailor-made first-rate garments. It organizes marketing, measurements, etc., but outsources the actual fabrication to independent tailors. The company keeps 20% of the revenue. When the company was founded in the 19th century, there were five subcontractors. Now it controls 15% of the world market and there are thousands of subcontractors worldwide.

Your task is to assign orders to the subcontractors. The rule is that an order is assigned to the tailor who has so far (in the current year) been assigned the smallest total value of orders. Your ancestors used a blackboard to keep track of the current total value of orders for each tailor; in computer science terms, they kept a list of values and spent linear time to find the correct tailor. The business has outgrown this solution. Can you come up with a more scalable solution where you have to look only at a small number of values to decide who will be assigned the next order?

In the following year the rules are changed. In order to encourage timely delivery, the orders are now assigned to the tailor with the smallest value of unfinished *orders, i.e., whenever a finished order arrives, you have to deduct the value of the order from the backlog of the tailor who executed it. Is your strategy for assigning orders flexible enough to handle this efficiently?*

Priority queues are the data structure required for the problem above and for many other applications. We start our discussion with the precise specification. Priority queues maintain a set M of *Element*s with *Key*s under the following operations:

- M.*build*$(\{e_1,\ldots,e_n\})$: $M := \{e_1,\ldots,e_n\}$.
- M.*insert*(e): $M := M \cup \{e\}$.
- M.min: **return** $\min M$.
- M.*deleteMin*: $e := \min M$; $M := M \setminus \{e\}$; **return** e.

This is enough for the first part of our example. Each year, we build a new priority queue containing an *Element* with a *Key* of zero for each contract tailor. To assign an order, we delete the smallest *Element*, add the order value to its *Key*, and reinsert it. Section 6.1 presents a simple, efficient implementation of this basic functionality.

[0] The photograph shows a queue at the Mao Mausoleum (V. Berger, see `http://commons.wikimedia.org/wiki/Image:Zhengyangmen01.jpg`).

Addressable priority queues additionally support operations on arbitrary elements addressed by an element handle h:

- *insert*: as before, but return a handle to the element inserted.
- *remove*(h): remove the element specified by the handle h.
- *decreaseKey*(h,k): decrease the key of the element specified by the handle h to k.
- M.*merge*(Q): $M := M \cup Q$; $Q := \emptyset$.

In our example, the operation *remove* might be helpful when a contractor is fired because he/she delivers poor quality. Using this operation together with *insert*, we can also implement the "new contract rules": when an order is delivered, we remove the *Element* for the contractor who executed the order, subtract the value of the order from its *Key* value, and reinsert the *Element*. *DecreaseKey* streamlines this process to a single operation. In Sect. 6.2, we shall see that this is not just convenient but that decreasing keys can be implemented more efficiently than arbitrary element updates.

Priority queues have many applications. For example, in Sect. 12.2, we shall see that our introductory example can also be viewed as a greedy algorithm for a machine-scheduling problem. Also, the rather naive selection-sort algorithm of Sect. 5.1 can be implemented efficiently now: first, insert all elements into a priority queue, and then repeatedly delete the smallest element and output it. A tuned version of this idea is described in Sect. 6.1. The resulting *heapsort* algorithm is popular because it needs no additional space and is worst-case efficient.

In a *discrete-event simulation*, one has to maintain a set of pending events. Each event happens at some scheduled point in time and creates zero or more new events in the future. Pending events are kept in a priority queue. The main loop of the simulation deletes the next event from the queue, executes it, and inserts newly generated events into the priority queue. Note that the priorities (times) of the deleted elements (simulated events) increase monotonically during the simulation. It turns out that many applications of priority queues have this monotonicity property. Section 10.5 explains how to exploit monotonicity for integer keys.

Another application of monotone priority queues is the *best-first branch-and-bound* approach to optimization described in Sect. 12.4. Here, the elements are partial solutions of an optimization problem and the keys are optimistic estimates of the obtainable solution quality. The algorithm repeatedly removes the best-looking partial solution, refines it, and inserts zero or more new partial solutions.

We shall see two applications of addressable priority queues in the chapters on graph algorithms. In both applications, the priority queue stores nodes of a graph. Dijkstra's algorithm for computing shortest paths (Sect. 10.3) uses a monotone priority queue where the keys are path lengths. The Jarník–Prim algorithm for computing minimum spanning trees (Sect. 11.2) uses a (nonmonotone) priority queue where the keys are the weights of edges connecting a node to a partial spanning tree. In both algorithms, there can be a *decreaseKey* operation for each edge, whereas there is at most one *insert* and *deleteMin* for each node. Observe that the number of edges may be much larger than the number of nodes, and hence the implementation of *decreaseKey* deserves special attention.

Exercise 6.1. Show how to implement bounded nonaddressable priority queues using arrays. The maximal size of the queue is w and when the queue has a size n, the first n entries of the array are used. Compare the complexity of the queue operations for two implementations: one by unsorted arrays and one by sorted arrays.

Exercise 6.2. Show how to implement addressable priority queues using doubly linked lists. Each list item represents an element in the queue, and a handle is a handle of a list item. Compare the complexity of the queue operations for two implementations: one by sorted lists and one by unsorted lists.

6.1 Binary Heaps

Heaps are a simple and efficient implementation of nonaddressable bounded priority queues [208]. They can be made unbounded in the same way as bounded arrays can be made unbounded (see Sect. 3.2). Heaps can also be made addressable, but we shall see better addressable queues in later sections.

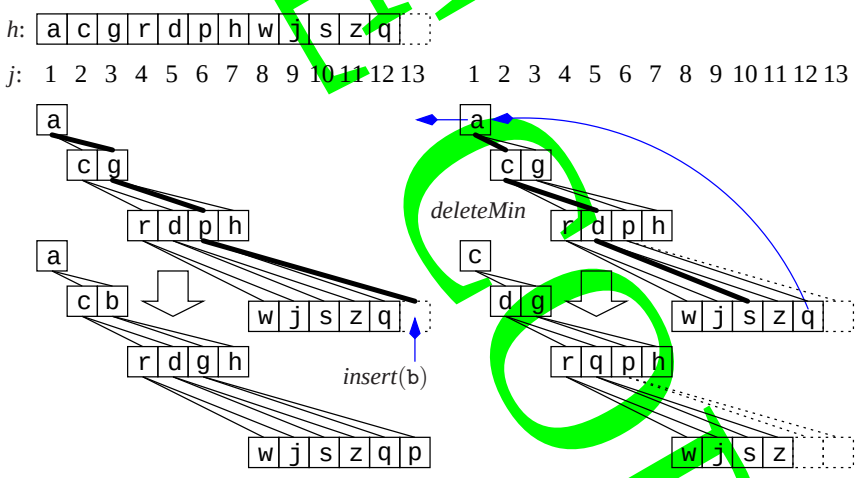


Fig. 6.1. The *top part* shows a heap with $n = 12$ elements stored in an array h with $w = 13$ entries. The root corresponds to index 1. The children of the root correspond to indices 2 and 3. The children of node i have indices $2i$ and $2i+1$ (if they exist). The parent of a node i, $i \geq 2$, has index $\lfloor i/2 \rfloor$. The elements stored in this implicitly defined tree fulfill the invariant that parents are no larger than their children, i.e., the tree is heap-ordered. The *left part* shows the effect of inserting b. The thick edges mark a path from the rightmost leaf to the root. The new element b is moved up this path until its parent is smaller. The remaining elements on the path are moved down to make room for b. The *right part* shows the effect of deleting the minimum. The thick edges mark the path p that starts at the root and always proceeds to the child with the smaller *Key*. The element q is provisionally moved to the root and then moves down p until its successors are larger. The remaining elements move up to make room for q

```
Class BinaryHeapPQ(w : ℕ) of Element
    h : Array [1..w] of Element                    // The heap h is
    n = 0 : ℕ                                      // initially empty and has the
    invariant ∀ j ∈ 2..n : h[⌊j/2⌋] ≤ h[j]          // heap property which implies that
    Function min assert n > 0 ; return h[1]        // the root is the minimum.
```

Fig. 6.2. A class for a priority queue based on binary heaps whose size is bounded by w

We use an array $h[1..w]$ that stores the elements of the queue. The first n entries of the array are used. The array is *heap-ordered*, i.e.,

$$\text{for } j \text{ with } 2 \le j \le n: \quad h[\lfloor j/2 \rfloor] \le h[j].$$

What does "heap-ordered" mean? The key to understanding this definition is a bijection between positive integers and the nodes of a complete binary tree, as illustrated in Fig. 6.1. In a heap the minimum element is stored in the root (= array position 1). Thus min takes time $O(1)$. Creating an empty heap with space for w elements also takes constant time, as it only needs to allocate an array of size w. Figure 6.2 gives pseudocode for this basic setup.

The minimum of a heap is stored in $h[1]$ and hence can be found in constant time; this is the same as for a sorted array. However, the heap property is much less restrictive than the property of being sorted. For example, there is only one sorted version of the set $\{1,2,3\}$, but both $\langle 1,2,3\rangle$ and $\langle 1,3,2\rangle$ are legal heap representations.

Exercise 6.3. Give all representations of $\{1,2,3,4\}$ as a heap.

We shall next see that the increased flexibility permits efficient implementations of *insert* and *deleteMin*. We choose a description which is simple and can be easily proven correct. Section 6.4 gives some hints toward a more efficient implementation. An *insert* puts a new element e tentatively at the end of the heap h, i.e., into $h[n]$, and then moves e to an appropriate position on the path from leaf $h[n]$ to the root:

```
Procedure insert(e : Element)
    assert n < w
    n++; h[n] := e
    siftUp(n)
```

Here $siftUp(s)$ moves the contents of node s toward the root until the heap property holds (see. Fig. 6.1).

```
Procedure siftUp(i : ℕ)
    assert the heap property holds except maybe at position i
    if i = 1 ∨ h[⌊i/2⌋] ≤ h[i] then return
    assert the heap property holds except for position i
    swap(h[i], h[⌊i/2⌋])
    assert the heap property holds except maybe for position ⌊i/2⌋
    siftUp(⌊i/2⌋)
```

Correctness follows from the invariants stated.

Exercise 6.4. Show that the running time of *siftUp*(n) is $O(\log n)$ and hence an *insert* takes time $O(\log n)$.

A *deleteMin* returns the contents of the root and replaces them by the contents of node n. Since $h[n]$ might be larger than $h[1]$ or $h[2]$, this manipulation may violate the heap property at position 1 or 2. This possible violation is repaired using *siftDown*:

```
Function deleteMin : Element
    assert n > 0
    result = h[1] : Element
    h[1] := h[n]; n--
    siftDown(1)
    return result
```

The procedure *siftDown*(1) moves the new contents of the root down the tree until the heap property holds. More precisely, consider the path p that starts at the root and always proceeds to the child with the smaller key (see Fig. 6.1); in the case of equal keys, the choice is arbitrary. We extend the path until all children (there may be zero, one, or two) have a key no larger than $h[1]$. We put $h[1]$ into this position and move all elements on path p up by one position. In this way, the heap property is restored. This strategy is most easily formulated as a recursive procedure. A call of the following procedure *siftDown*(i) repairs the heap property in the subtree rooted at i, assuming that it holds already for the subtrees rooted at $2i$ and $2i+1$; the heap property holds in the subtree rooted at i if we have $h[\lfloor j/2 \rfloor] \leq h[j]$ for all proper descendants j of i:

```
Procedure siftDown(i : ℕ)
    assert the heap property holds for the trees rooted at j = 2i and j = 2i+1
    if 2i ≤ n then                                   // i is not a leaf
        if 2i+1 > n ∨ h[2i] ≤ h[2i+1] then m := 2i else m := 2i+1
        assert the sibling of m does not exist or it has a larger key than m
        if h[i] > h[m] then                          // the heap property is violated
            swap(h[i], h[m])
            siftDown(m)
    assert the heap property holds for the tree rooted at i
```

Exercise 6.5. Our current implementation of *siftDown* needs about $2\log n$ element comparisons. Show how to reduce this to $\log n + O(\log\log n)$. Hint: determine the path p first and then perform a binary search on this path to find the proper position for $h[1]$. Section 6.5 has more on variants of *siftDown*.

We can obviously build a heap from n elements by inserting them one after the other in $O(n\log n)$ total time. Interestingly, we can do better by establishing the heap property in a bottom-up fashion: *siftDown* allows us to establish the heap property for a subtree of height $k+1$ provided the heap property holds for its subtrees of height k. The following exercise asks you to work out the details of this idea.

Exercise 6.6 (*buildHeap*). Assume that you are given an arbitrary array $h[1..n]$ and want to establish the heap property on it by permuting its entries. Consider two procedures for achieving this:

```
Procedure buildHeapBackwards
    for i := ⌊n/2⌋ downto 1 do siftDown(i)
Procedure buildHeapRecursive(i : ℕ)
    if 4i ≤ n then
        buildHeapRecursive(2i)
        buildHeapRecursive(2i + 1)
    siftDown(i)
```

(a) Show that both *buildHeapBackwards* and *buildHeapRecursive*(1) establish the heap property everywhere.

(b) Implement both algorithms efficiently and compare their running times for random integers and $n \in \{10^i : 2 \le i \le 8\}$. It will be important how efficiently you implement *buildHeapRecursive*. In particular, it might make sense to unravel the recursion for small subtrees.

*(c) For large n, the main difference between the two algorithms is in memory hierarchy effects. Analyze the number of I/O operations required by the two algorithms in the external-memory model described at the end of Sect. 2.2. In particular, show that if the block size is B and the fast memory has size $M = \Omega(B \log B)$, then *buildHeapRecursive* needs only $O(n/B)$ I/O operations.

The following theorem summarizes our results on binary heaps.

Theorem 6.1. *The heap implementation of nonaddressable priority queues realizes creating an empty heap and finding the minimum element in constant time, deleteMin and insert in logarithmic time* $O(\log n)$*, and build in linear time.*

Proof. The binary tree represented by a heap of n elements has a height of $k = \lfloor \log n \rfloor$. *insert* and *deleteMin* explore one root-to-leaf path and hence have logarithmic running time; min returns the contents of the root and hence takes constant time. Creating an empty heap amounts to allocating an array and therefore takes constant time. *build* calls *siftDown* for at most 2^ℓ nodes of depth ℓ. Such a call takes time $O(k-\ell)$. Thus total the time is

$$O\left(\sum_{0 \le \ell < k} 2^\ell (k-\ell)\right) = O\left(2^k \sum_{0 \le \ell < k} \frac{k-\ell}{2^{k-\ell}}\right) = O\left(2^k \sum_{j \ge 1} \frac{j}{2^j}\right) = O(n) .$$

The last equality uses (A.14). □

Heaps are the basis of *heapsort*. We first *build* a heap from the elements and then repeatedly perform *deleteMin*. Before the i-th *deleteMin* operation, the i-th smallest element is stored at the root $h[1]$. We swap $h[1]$ and $h[n-i+1]$ and sift the new root down to its appropriate position. At the end, h stores the elements sorted in

decreasing order. Of course, we can also sort in increasing order by using a *max-priority queue*, i.e., a data structure supporting the operations *insert* and of deleting the maximum.

Heaps do not immediately implement the data type addressable priority queue, since elements are moved around in the array h during insertion and deletion. Thus the array indices cannot be used as handles.

Exercise 6.7 (addressable binary heaps). Extend heaps to an implementation of addressable priority queues. How many additional pointers per element do you need? There is a solution with two additional pointers per element.

***Exercise 6.8 (bulk insertion).** Design an algorithm for inserting k new elements into an n-element heap. Give an algorithm that runs in time $O(k + \log n)$. Hint: use a bottom-up approach similar to that for heap construction.

6.2 Addressable Priority Queues

Binary heaps have a rather rigid structure. All n elements are arranged into a single binary tree of height $\lfloor \log n \rfloor$. In order to obtain faster implementations of the operations *insert*, *decreaseKey*, *remove*, and *merge*, we now look at structures which are more flexible. The single, complete binary tree is replaced by a collection of trees (i.e., a forest) with arbitrary shape. Each tree is still *heap-ordered*, i.e., no child is smaller than its parent. In other words, the sequence of keys along any root-to-leaf path is nondecreasing. Figure 6.4 shows a heap-ordered forest. Furthermore, the elements of the queue are now stored in *heap items* that have a persistent location in memory. Hence, pointers to heap items can serve as *handle*s to priority queue elements. The tree structure is explicitly defined using pointers between items.

We shall discuss several variants of addressable priority queues. We start with the common principles underlying all of them. Figure 6.3 summarizes the commonalities.

In order to keep track of the current minimum, we maintain the handle to the root containing it. We use *minPtr* to denote this handle. The forest is manipulated using three simple operations: adding a new tree (and keeping *minPtr* up to date), combining two trees into a single one, and cutting out a subtree, making it a tree of its own.

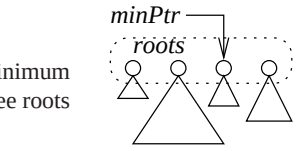
An *insert* adds a new single-node tree to the forest. So a sequence of n inserts into an initially empty heap will simply create n single-node trees. The cost of an insert is clearly $O(1)$.

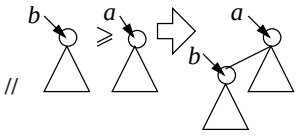
A *deleteMin* operation removes the node indicated by *minPtr*. This turns all children of the removed node into roots. We then scan the set of roots (old and new) to find the new minimum, a potentially very costly process. We also perform some rebalancing, i.e., we combine trees into larger ones. The details of this process distinguish different kinds of addressable priority queue and are the key to efficiency.

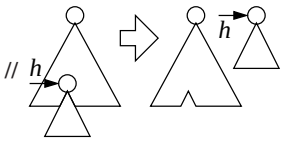
We turn now to *decreaseKey*(h,k) which decreases the key value at a handle h to k. Of course, k must not be larger than the old key stored with h. Decreasing the

Class *Handle* = **Pointer to** *PQItem*

Class *AddressablePQ*
minPtr : *Handle* // root that stores the minimum
roots : *Set* **of** *Handle* // pointers to tree roots
Function min **return** element stored at *minPtr*
Procedure *link*(*a*,*b* : *Handle*)
assert $a \leq b$
remove *b* from *roots*
make *a* the parent of *b*
Procedure *combine*(*a*,*b* : *Handle*)
assert *a* and *b* are tree roots
if $a \leq b$ **then** *link*(a,b) **else** *link*(b,a)
Procedure *newTree*(*h* : *Handle*)
$roots := roots \cup \{h\}$
if $*h <$ min **then** $minPtr := h$
Procedure *cut*(*h* : *Handle*)
remove the subtree rooted at *h* from its tree
newTree(*h*)
Function *insert*(*e* : *Element*) : *Handle*
i:=a *Handle* for a new *PQItem* storing *e*
newTree(*i*)
return *i*
Function *deleteMin* : *Element*
e:= the *Element* stored in *minPtr*
foreach child *h* of the root at *minPtr* **do** *cut*(*h*)
dispose *minPtr*
perform some rebalancing and update *minPtr* // uses *combine*
return *e*
Procedure *decreaseKey*(*h* : *Handle*, *k* : *Key*)
change the key of *h* to *k*
if *h* is not a root **then**
cut(*h*); possibly perform some rebalancing
Procedure *remove*(*h* : *Handle*) *decreaseKey*$(h,-\infty)$; *deleteMin*
Procedure *merge*(*o* : *AddressablePQ*)
if $*minPtr > *(o.minPtr)$ **then** $minPtr := o.minPtr$
$roots := roots \cup o.roots$
$o.roots := \emptyset$; possibly perform some rebalancing







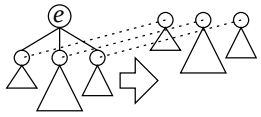


Fig. 6.3. Addressable priority queues

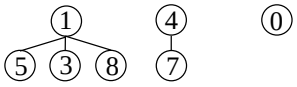


Fig. 6.4. A heap-ordered forest representing the set $\{0,1,3,4,5,7,8\}$

key associated with h may destroy the heap property because h may now be smaller than its parent. In order to maintain the heap property, we cut the subtree rooted at h and turn h into a root. This sounds simple enough, but may create highly skewed trees. Therefore, some variants of addressable priority queues perform additional operations to keep the trees in shape.

The remaining operations are easy. We can *remove* an item from the queue by first decreasing its key so that it becomes the minimum item in the queue, and then perform a *deleteMin*. To merge a queue o into another queue we compute the union of *roots* and o.*roots*. To update *minPtr*, it suffices to compare the minima of the merged queues. If the root sets are represented by linked lists, and no additional balancing is done, a merge needs only constant time.

In the remainder of this section we shall discuss particular implementations of addressable priority queues.

6.2.1 Pairing Heaps

Pairing heaps [67] use a very simple technique for rebalancing. Pairing heaps are efficient in practice; however a full theoretical analysis is missing. They rebalance only in *deleteMin*. If $\langle r_1, \ldots, r_k \rangle$ is the sequence of root nodes stored in *roots*, then *deleteMin combine*s r_1 with r_2, r_3 with r_4, etc., i.e., the *roots* are *paired*. Figure 6.5 gives an example.

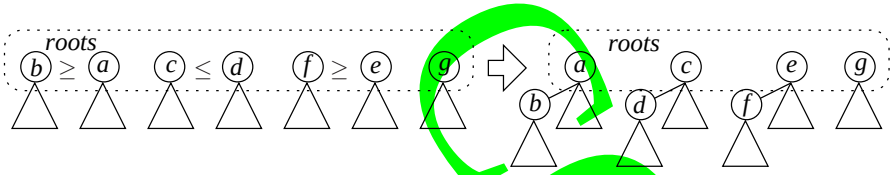


Fig. 6.5. The *deleteMin* operation for pairing heaps combines pairs of root nodes

Exercise 6.9 (three-pointer items). Explain how to implement pairing heaps using three pointers per heap item i: one to the oldest child (i.e., the child linked first to i), one to the next younger sibling (if any), and one to the next older sibling. If there is no older sibling, the third pointer goes to the parent. Figure 6.8 gives an example.

***Exercise 6.10 (two-pointer items).** Explain how to implement pairing heaps using two pointers per heap item: one to the oldest child and one to next younger sibling. If there is no younger sibling, the second pointer goes to the parent. Figure 6.8 gives an example.

6.2.2 *Fibonacci Heaps

Fibonacci heaps [68] use more intensive balancing operations than do pairing heaps. This paves the way to a theoretical analysis. In particular, we obtain logarithmic

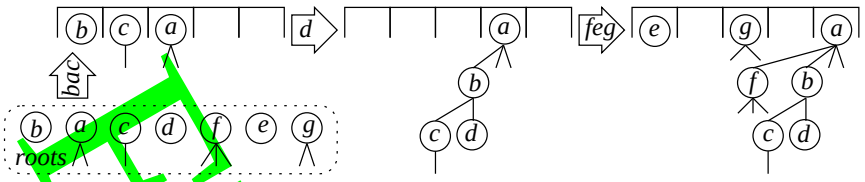


Fig. 6.6. An example of the development of the bucket array during execution of *deleteMin* for a Fibonacci heap. The arrows indicate the roots scanned. Note that scanning *d* leads to a cascade of three combine operations

amortized time for *remove* and *deleteMin* and worst-case constant time for all other operations.

Each item of a Fibonacci heap stores four pointers that identify its parent, one child, and two siblings (see Fig. 6.8). The children of each node form a doubly linked circular list using the sibling pointers. The sibling pointers of the root nodes can be used to represent *roots* in a similar way. Parent pointers of roots and child pointers of leaf nodes have a special value, for example, a null pointer.

In addition, every heap item contains a field *rank*. The *rank* of an item is the number of its children. In Fibonacci heaps, *deleteMin* links roots of equal rank r. The surviving root will then obtain a rank of $r+1$. An efficient method to combine trees of equal rank is as follows. Let *maxRank* be an upper bound on the maximal rank of any node. We shall prove below that *maxRank* is logarithmic in n. Maintain a set of buckets, initially empty and numbered from 0 to *maxRank*. Then scan the list of old and new roots. When scanning a root of rank i, inspect the i-th bucket. If the i-th bucket is empty, then put the root there. If the bucket is nonempty, then combine the two trees into one. This empties the i-th bucket and creates a root of rank $i+1$. Treat this root in the same way, i.e., try to throw it into the $i+1$-th bucket. If it is occupied, combine .... When all roots have been processed in this way, we have a collection of trees whose roots have pairwise distinct ranks (see Figure 6.6).

A *deleteMin* can be very expensive if there are many roots. For example, a *deleteMin* following n insertions has a cost $\Omega(n)$. However, in an amortized sense, the cost of *deletemin* is $O(maxRank)$. The reader must be familiar with the technique of amortized analysis (see Sect. 3.3) before proceeding further. For the amortized analysis, we postulate that each root holds one token. Tokens pay for a constant amount of computing time.

Lemma 6.2. *The amortized complexity of deleteMin is* $O(maxRank)$.

Proof. A *deleteMin* first calls *newTree* at most *maxRank* times (since the degree of the old minimum is bounded by *maxRank*) and then initializes an array of size *maxRank*. Thus its running time is $O(maxRank)$ and it needs to create *maxRank* new tokens. The remaining time is proportional to the number of *combine* operations performed. Each *combine* turns a root into a nonroot and is paid for by the token associated with the node turning into a nonroot. □

How can we guarantee that *maxRank* stays small? Let us consider a simple situation first. Suppose that we perform a sequence of insertions followed by a one *deleteMin*. In this situation, we start with a certain number of single-node trees and all trees formed by combining are *binomial trees*, as shown in Fig. 6.7. The binomial tree B_0 consists of a single node, and the binomial tree B_{i+1} is obtained by combining two copies of B_i. This implies that the root of B_i has rank i and that B_i contains exactly 2^i nodes. Thus the rank of a binomial tree is logarithmic in the size of the tree.

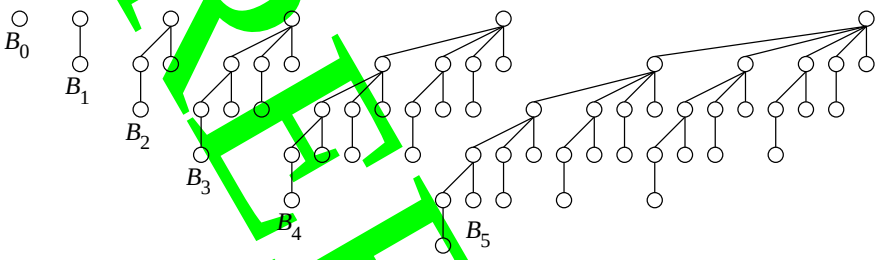


Fig. 6.7. The binomial trees of ranks zero to five

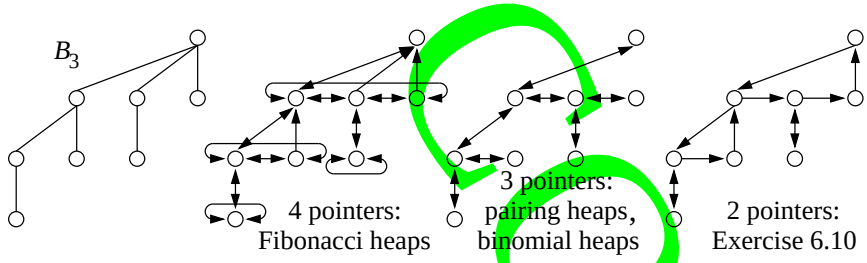


Fig. 6.8. Three ways to represent trees of nonuniform degree. The binomial tree of rank three, B_3, is used as an example

Unfortunately, *decreaseKey* may destroy the nice structure of binomial trees. Suppose an item v is cut out. We now have to decrease the rank of its parent w. The problem is that the size of the subtrees rooted at the ancestors of w has decreased but their rank has not changed, and hence we can no longer claim that the size of a tree stays exponential in the rank of its root. Therefore, we have to perform some rebalancing to keep the trees in shape. An old solution [202] is to keep all trees in the heap binomial. However, this causes logarithmic cost for a *decreaseKey*.

***Exercise 6.11 (binomial heaps).** Work out the details of this idea. Hint: cut the following links. For each ancestor of v and for v itself, cut the link to its parent. For

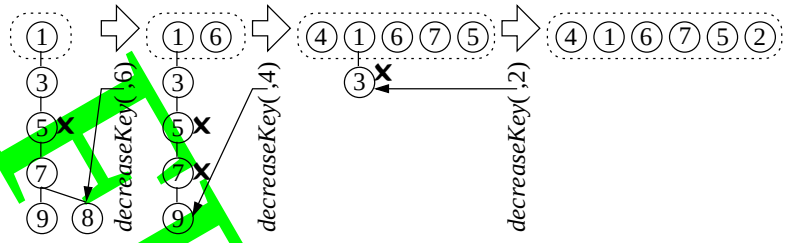


Fig. 6.9. An example of cascading cuts. Marks are drawn as crosses. Note that roots are never marked

each sibling of v of rank higher than v, cut the link to its parent. Argue that the trees stay binomial and that the cost of *decreaseKey* is logarithmic.

Fibonacci heaps allow the trees to go out of shape but in a controlled way. The idea is surprisingly simple and was inspired by the amortized analysis of binary counters (see Sect. 3.2.3). We introduce an additional flag for each node. A node may be marked or not. Roots are never marked. In particular, when *newTree*(h) is called in *deleteMin*, it removes the mark from h (if any). Thus when *combine* combines two trees into one, neither node is marked.

When a nonroot item x loses a child because *decreaseKey* has been applied to the child, x is marked; this assumes that x is not already marked. Otherwise, when x was already marked, we cut x, remove the mark from x, and attempt to mark x's parent. If x's parent is already marked, then .... This technique is called *cascading cuts*. In other words, suppose that we apply *decreaseKey* to an item v and that the k nearest ancestors of v are marked. We turn v and the k nearest ancestors of v into roots, unmark them, and mark the $k+1$-th nearest ancestor of v (if it is not a root). Figure 6.9 gives an example. Observe the similarity to carry propagation in binary addition.

For the amortized analysis, we postulate that each marked node holds two tokens and each root holds one token. Please check that this assumption does not invalidate the proof of Lemma 6.2.

Lemma 6.3. *The amortized complexity of decreaseKey is constant.*

Proof. Assume that we decrease the key of item v and that the k nearest ancestors of v are marked. Here, $k \geq 0$. The running time of the operation is $O(1+k)$. Each of the k marked ancestors carries two tokens, i.e., we have a total of $2k$ tokens available. We create $k+1$ new roots and need one token for each of them. Also, we mark one unmarked node and need two tokens for it. Thus we need a total of $k+3$ tokens. In other words, $k-3$ tokens are freed. They pay for all but $O(1)$ of the cost of *decreaseKey*. Thus the amortized cost of *decreaseKey* is constant. □

How do cascading cuts affect the size of trees? We shall show that it stays exponential in the rank of the root. In order to do so, we need some notation. Recall

the sequence 0, 1, 1, 2, 3, 5, 8, ... of Fibonacci numbers. These are defined by the recurrence $F_0 = 0$, $F_1 = 1$, and $F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$. It is well known that $F_{i+1} \geq ((1+\sqrt{5})/2)^i \geq 1.618^i$ for all $i \geq 0$.

Exercise 6.12. Prove that $F_{i+2} \geq ((1+\sqrt{5})/2)^i \geq 1.618^i$ for all $i \geq 0$ by induction.

Lemma 6.4. *Let v be any item in a Fibonacci heap and let i be the rank of v. The subtree rooted at v then contains at least F_{i+2} nodes. In a Fibonacci heap with n items, all ranks are bounded by* $1.4404 \log n$.

Proof. Consider an arbitrary item v of rank i. Order the children of v by the time at which they were made children of v. Let w_j be the j-th child, $1 \leq j \leq i$. When w_j was made a child of v, both nodes had the same rank. Also, since at least the nodes $w_1, \ldots, w_{j-1}$ were children of v at that time, the rank of v was at least $j-1$ then. The rank of w_j has decreased by at most 1 since then, because otherwise w_j would no longer be a child of v. Thus the current rank of w_j is at least $j-2$.

We can now set up a recurrence for the minimal number S_i of nodes in a tree whose root has rank i. Clearly, $S_0 = 1$, $S_1 = 2$, and $S_i \geq 2 + S_0 + S_1 + \cdots + S_{i-2}$. The latter inequality follows from the fact that for $j \geq 2$, the number of nodes in the subtree with root w_j is at least S_{j-2}, and that we can also count the nodes v and w_1. The recurrence above (with = instead of $\geq$) generates the sequence 1, 2, 3, 5, 8, ... which is identical to the Fibonacci sequence (minus its first two elements).

Let us verify this by induction. Let $T_0 = 1$, $T_1 = 2$, and $T_i = 2 + T_0 + \cdots + T_{i-2}$ for $i \geq 2$. Then, for $i \geq 2$, $T_{i+1} - T_i = 2 + T_0 + \cdots + T_{i-1} - 2 - T_0 - \cdots - T_{i-2} = T_{i-1}$, i.e., $T_{i+1} = T_i + T_{i-1}$. This proves $T_i = F_{i+2}$.

For the second claim, we observe that $F_{i+2} \leq n$ implies $i \cdot \log((1+\sqrt{5})/2) \leq \log n$, which in turn implies $i \leq 1.4404 \log n$. □

This concludes our treatment of Fibonacci heaps. We have shown the following result.

Theorem 6.5. *The following time bounds hold for Fibonacci heaps: min, insert, and merge take worst-case constant time; decreaseKey takes amortized constant time, and remove and deleteMin take an amortized time logarithmic in the size of the queue.*

Exercise 6.13. Describe a variant of Fibonacci heaps where all roots have distinct ranks.

6.3 *External Memory

We now go back to nonaddressable priority queues and consider their cache efficiency and I/O efficiency. A weakness of binary heaps is that the *siftDown* operation goes down the tree in an unpredictable fashion. This leads to many cache faults and makes binary heaps prohibitively slow when they do not fit into the main memory.

We now outline a data structure for (nonadressable) priority queues with more regular memory accesses. It is also a good example of a generally useful design principle: construction of a data structure out of simpler, known components and algorithms.

In this case, the components are internal-memory priority queues, sorting, and multiway merging (see also Sect. 5.7.1). Figure 6.10 depicts the basic design. The data structure consists of two priority queues Q and Q' (e.g., binary heaps) and k sorted sequences $S_1, \ldots, S_k$. Each element of the priority queue is stored either in the *insertion queue* Q, in the *deletion queue* Q', or in one of the sorted sequences. The size of Q is limited to a parameter m. The *deletion queue* Q' stores the smallest element of each sequence, together with the index of the sequence holding the element.

New elements are inserted into the insertion queue. If the insertion queue is full, it is first emptied. In this case, its elements form a new sorted sequence:

Procedure *insert*$(e : Element)$
  if $|Q| = m$ **then**
    k++; $S_k := sort(Q)$; $Q := \emptyset$; $Q'.insert((S_k.popFront, k))$
  $Q.insert(e)$

The minimum is stored either in Q or in Q'. If the minimum is in Q' and comes from sequence S_i, the next largest element of S_i is inserted into Q':

Function *deleteMin*
  if $\min Q \leq \min Q'$ **then** $e := Q.deleteMin$ // assume $\min \emptyset = \infty$
  else $(e, i) := Q'.deleteMin$
    if $S_i \neq \langle\rangle$ **then** $Q'.insert((S_i.popFront, i))$
  return e

It remains to explain how the ingredients of our data structure are mapped to the memory hierarchy. The queues Q and Q' are stored in internal memory. The size bound m for Q should be a constant fraction of the internal-memory size M and a multiple of the block size B. The sequences S_i are largely kept externally. Initially, only the B smallest elements of S_i are kept in an internal-memory buffer b_i. When the last element of b_i is removed, the next B elements of S_i are loaded. Note that we are effectively merging the sequences S_i. This is similar to our multiway merging

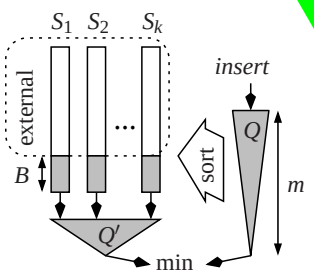


Fig. 6.10. Schematic view of an external-memory priority queue

algorithm described in Sect. 5.7.1. Each inserted element is written to disk at most once and fetched back to internal memory at most once. Since all disk accesses are in units of at least a full block, the I/O requirement of our algorithm is at most n/B for n queue operations.

Our total requirement for internal memory is at most $m+kB+2k$. This is below the total fast-memory size M if $m = M/2$ and $k \leq \lfloor (M/2-2k)/B \rfloor \approx M/(2B)$. If there are many insertions, the internal memory may eventually overflow. However, the earliest this can happen is after $m(1+\lfloor (M/2-2k)/B \rfloor) \approx M^2/(4B)$ insertions. For example, if we have 1 Gbyte of main memory, 8-byte elements, and 512 Kbyte disk blocks, we have $M = 2^{27}$ and $B = 2^{16}$ (measured in elements). We can then perform about 2^{36} insertions – enough for 128 Gbyte of data. Similarly to external mergesort, we can handle larger amounts of data by performing multiple phases of multiway merging (see, [31, 164]). The data structure becomes considerably more complicated, but it turns out that the I/O requirement for n insertions and deletions is about the same as for sorting n elements. An implementation of this idea is two to three times faster than binary heaps for the hierarchy between cache and main memory [164]. There are also implementations for external memory [48].

6.4 Implementation Notes

There are various places where *sentinels* (see Chap. 3) can be used to simplify or (slightly) accelerate the implementation of priority queues. Since sentinels may require additional knowledge about key values, this could make a reusable implementation more difficult, however.

- If $h[0]$ stores a *Key* no larger than any *Key* ever inserted into a binary heap, then *siftUp* need not treat the case $i = 1$ in a special way.
- If $h[n+1]$ stores a *Key* no smaller than any *Key* ever inserted into a binary heap, then *siftDown* need not treat the case $2i+1 > n$ in a special way. If such large keys are stored in $h[n+1..2n+1]$, then the case $2i > n$ can also be eliminated.
- Addressable priority queues can use a special dummy item rather than a null pointer.

For simplicity we have formulated the operations *siftDown* and *siftUp* for binary heaps using recursion. It might be a little faster to implement them iteratively instead. Similarly, the *swap* operations could be replaced by unidirectional move operations thus halving the number of memory accesses.

Exercise 6.14. Give iterative versions of *siftDown* and *siftUp*. Also replace the *swap* operations.

Some compilers do the recursion elimination for you.

As for sequences, memory management for items of addressable priority queues can be critical for performance. Often, a particular application may be able to do this more efficiently than a general-purpose library. For example, many graph algorithms use a priority queue of nodes. In this case, items can be incorporated into nodes.

There are priority queues that work efficiently for integer keys. It should be noted that these queues can also be used for floating-point numbers. Indeed, the IEEE floating-point standard has the interesting property that for any valid floating-point numbers a and b, $a \leq b$ if and only if $bits(a) \leq bits(b)$, where $bits(x)$ denotes the reinterpretation of x as an unsigned integer.

6.4.1 C++

The STL class *priority_queue* offers nonaddressable priority queues implemented using binary heaps. The external-memory library STXXL [48] offers an external-memory priority queue. LEDA [118] implements a wide variety of addressable priority queues, including pairing heaps and Fibonacci heaps.

6.4.2 Java

The class *java.util.PriorityQueue* supports addressable priority queues to the extent that *remove* is implemented. However, *decreaseKey* and *merge* are not supported. Also, it seems that the current implementation of *remove* needs time $\Theta(n)$! JDSL [78] offers an addressable priority queue *jdsl.core.api.PriorityQueue*, which is currently implemented as a binary heap.

6.5 Historical Notes and Further Findings

There is an interesting Internet survey[1] of priority queues. It lists the following applications: (shortest-) path planning (see Chap. 10), discrete-event simulation, coding and compression, scheduling in operating systems, computing maximum flows, and branch-and-bound (see Sect. 12.4).

In Sect. 6.1 we saw an implementation of *deleteMin* by top-down search that needs about $2\log n$ element comparisons, and a variant using binary search that needs only $\log n + O(\log\log n)$ element comparisons. The latter is mostly of theoretical interest. Interestingly, a very simple "bottom-up" algorithm can be even better: The old minimum is removed and the resulting hole is sifted down all the way to the bottom of the heap. Only then, the rightmost element fills the hole and is subsequently sifted up. When used for sorting, the resulting *Bottom-up heapsort* requires $\frac{3}{2}n\log n + O(n)$ comparisons in the worst case and $n\log n + O(1)$ in the average case [204, 61, 169]. While bottom-up heapsort is simple and practical, our own experiments indicate that it is not faster than the usual top-down variant (for integer keys). This surprised us. The explanation might be that the outcomes of the comparisons saved by the bottom-up variant are easy to predict. Modern hardware executes such predictable comparisons very efficiently (see [167] for more discussion).

The recursive *buildHeap* routine in Exercise 6.6 is an example of a *cache-oblivious algorithm* [69]. This algorithm is efficient in the external-memory model even though it does not explicitly use the block size or cache size.

[1] `http://www.leekillough.com/heaps/survey_results.html`

Pairing heaps [67] have constant amortized complexity for *insert* and *merge* [96] and logarithmic amortized complexity for *deleteMin*. The best analysis is that due to Pettie [154]. Fredman [65] has given operation sequences consisting of $O(n)$ insertions and *deleteMin*s and $O(n \log n)$ *decreaseKey*s that require time $\Omega(n \log n \log\log n)$ for a family of addressable priority queues that includes all previously proposed variants of pairing heaps.

The family of addressable priority queues is large. Vuillemin [202] introduced binomial heaps, and Fredman and Tarjan [68] invented Fibonacci heaps. Høyer [94] described additional balancing operations that are akin to the operations used for search trees. One such operation yields *thin heaps* [103], which have performance guarantees similar to Fibonacci heaps and do without parent pointers and mark bits. It is likely that thin heaps are faster in practice than Fibonacci heaps. There are also priority queues with worst-case bounds asymptotically as good as the amortized bounds that we have seen for Fibonacci heaps [30]. The basic idea is to tolerate violations of the heap property and to continuously invest some work in reducing these violations. Another interesting variant is *fat heaps* [103].

Many applications need priority queues for integer keys only. For this special case, there are more efficient priority queues. The best theoretical bounds so far are constant time for *decreaseKey* and *insert* and $O(\log\log n)$ time for *deleteMin* [193, 136]. Using randomization, the time bound can even be reduced to $O(\sqrt{\log\log n})$ [85]. The algorithms are fairly complex. However, integer priority queues that also have the *monotonicity property* can be simple and practical. Section 10.3 gives examples. *Calendar queues* [33] are popular in the discrete-event simulation community. These are a variant of the *bucket queues* described in Sect. 10.5.1.