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.

Let us have a closer look at how orders are assigned 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. The founders of TMG 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 at only a small number of values to decide who will be assigned the next order?

Next year, the rules will be changed. In order to encourage timely delivery, orders will now be assigned to the tailor with the smallest value of unfinished *orders, i.e., whenever an order is assigned to a tailor, you have to increase the backlog of the tailor, and whenever a finished order arrives, you have to deduct the value of that order from the backlog of the tailor who executed it. In order to assign an order, you have to find the tailor with the smallest backlog. Is your strategy for assigning orders flexible enough to handle this efficiently?*

Priority queues[1] are the data structure required for the problem above and for many other applications. We start our discussion with the precise specification. (Nonaddressable) priority queues maintain a set M of *Element*s with *Key*s supporting the following operations:

[1] The photograph shows people queueing at the Eiffel Tower (Doods Dumaguing `www.flickr.com/photos/xianl2/8620507361`).

- $M.build(\{e_1,\ldots,e_n\})$: $M := \{e_1,\ldots,e_n\}$.
- $M.insert(e)$: $M := M \cup \{e\}$.
- $M.\min$: **return** $\min M$ (an element with minimum key).
- $M.deleteMin$: $e := \min M$; $M := M \setminus \{e\}$; **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 0 for each contracted 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.

Addressable priority queues support additional operations. The elements in an addressable priority queue are accessible through a handle. The handle is established when the element is inserted into the queue. The additional operations are:

- $M.insert(e)$: $M := M \cup \{e\}$; Return a handle to e.
- $remove(h)$: Remove the element specified by the handle h.
- $decreaseKey(h,k)$: Decrease the key of the element at 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 assigned or delivered, we remove the *Element* for the contractor who executed the order, update its backlog value, and reinsert the *Element*. *DecreaseKey* streamlines the actions for a delivery 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 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 some number, maybe zero, of 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.

Scheduling tasks of different importance in a parallel system is an application of parallel priority queues. For example, when parallelizing branch-and-bound, the

partial solutions might be viewed as such tasks and the importance of a task might be a bound on its objective value.

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 size of the queue is bounded by 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.

In Sect. 6.1 we begin with a very simple array-based data structure that is well suited to nonaddressable priority queues. We then discuss pointer-based addressable priority queues in Sect. 6.2. Section 6.3 shows how nonaddressable queues can be implemented efficiently in external memory. Section 6.4 discusses parallel priority queues. We shall describe a version with bulk operations that accesses many elements at once.

6.1 Binary Heaps

Heaps are a simple and efficient implementation of nonaddressable bounded priority queues [331]. They can be made unbounded in the same way as bounded arrays can be made unbounded; see Sect. 3.4. Heaps can also be made addressable, but we shall see better addressable queues in later sections.

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. Node 1 is the root of the tree, and the children of node i are the nodes with numbers $2i$ and $2i+1$. The parent of a node $i \ge 2$ is node $\lfloor i/2 \rfloor$. A heap of size

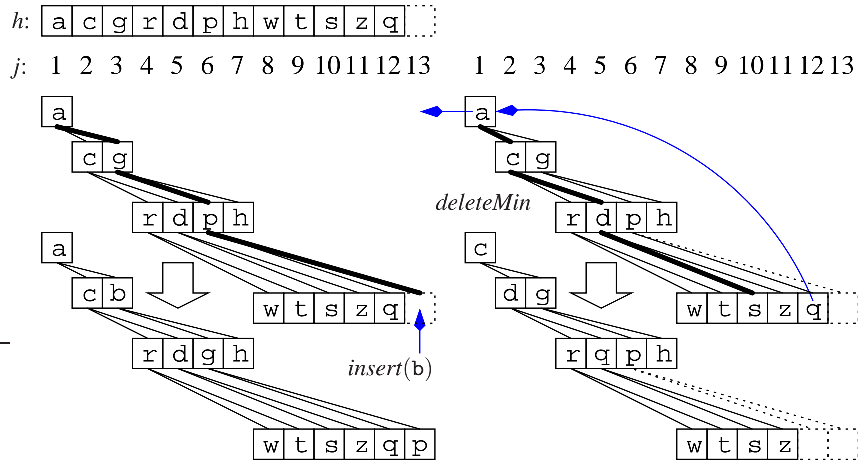


Fig. 6.1. The *top part* shows a heap with $n = 12$ elements stored in an array h with $w = 13$ entries. The root corresponds to index 1. The children of the root correspond to indices 2 and 3. The children of node i have indices $2i$ and $2i+1$ (if they exist). The parent of a node i, $i \geq 2$, has index $\lfloor i/2 \rfloor$. If n elements are stored in the heap, they are stored in nodes 1 to n. The keys are characters, with their usual alphabetic order. The invariant states that the key of a parent is no larger than the keys of its children, i.e., the tree is heap-ordered. The *left part* shows the effect of inserting b. We add node 13 as a new leaf to the heap. The thick edges mark a path from node 13 to the root. The new element b is moved up this path until its parent is not larger. 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 stored in the node n is provisionally moved to the root and then moves down p until its successor in p is not smaller anymore. The remaining elements move up to make room for q.

n uses nodes 1 to n. In a heap, the key of a parent is no larger than the keys of the children. In particular, a minimum element is stored in the root (= array position 1). Thus the operation 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:

Class *BinaryHeapPQ*$(w : \mathbb{N})$ **of** *Element*
  h : *Array* $[1..w]$ **of** *Element* // The *heap* h is
  $n = 0 : \mathbb{N}$ // initially *empty* and has the
  invariant $\forall j \in 2..n : h[\lfloor j/2 \rfloor] \leq h[j]$ // *heap property*, which implies that
  Function min **assert** $n > 0$; **return** $h[1]$ // the *root* contains the *min*imum.

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 easily proven correct. Section 6.5 gives some hints towards a more efficient implementation. An *insert* puts a new element e tentatively at the end of the heap h, i.e., it increments n and tentatively puts the new element into $h[n]$. This may violate the heap property at position n. To repair the heap property, we move 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*(n) moves the contents of node n towards the root until either the root is reached or the key in the parent is no larger anymore; see Fig. 6.1. We have to prove that this restores the heap property. We write "heap except maybe at i" if h is a heap or $i > 1$, $h[i] < h[\lfloor i/2 \rfloor]$ and replacing $h[i]$ by $h[\lfloor i/2 \rfloor]$ turns h into a heap. When we put the new element into $h[n]$, h is a heap except maybe at n. Assume now that h is a heap except maybe at i when *siftUp*(i) is called. By the preceding sentence, this is true for the first call with $i = n$. If $i = 1$ or $h[\lfloor i/2 \rfloor] \leq h[i]$, h is a heap and we are done. If $i > 1$ and $h[\lfloor i/2 \rfloor] > h[i]$, we swap $h[i]$ and $h[\lfloor i/2 \rfloor]$. The heap property now holds for the children of i, since it sufficed to replace $h[i]$ by $h[\lfloor i/2 \rfloor]$ to restore the heap property. It clearly holds for i, and it holds for the sibling of i since we have replaced $h[\lfloor i/2 \rfloor]$ by something smaller. Hence h is a heap except maybe at $\lfloor i/2 \rfloor$, and we have established the invariant for the recursive call.

Procedure *siftUp*$(i : \mathbb{N})$
  assert h is a heap except maybe at i.
  if $i = 1 \vee h[\lfloor i/2 \rfloor] \leq h[i]$ **then return**
  swap$(h[i], h[\lfloor i/2 \rfloor])$
  siftUp$(\lfloor i/2 \rfloor)$

Exercise 6.4. Show that the running time of *siftUp*(n) is $\mathrm{O}(\log n)$ and hence an *insert* takes time $\mathrm{O}(\log n)$. Reformulate *siftUp* as a while-loop.

A *deleteMin* returns the content of the root and replaces it by the content of node n. Since $h[n]$ might be larger than $h[2]$ or $h[3]$, this manipulation may violate the heap property at position 2 or 3. This possible violation is repaired using *siftDown*:

Function *deleteMin* : *Element*
  assert $n > 0$
  $result = h[1] : Element$
  $h[1] := h[n];\quad n{-}{-}$
  siftDown(1)
  return *result*

The procedure *siftDown*(1) moves the new content of the root, which we call e, 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 of the last node of the path (there may be zero, one, or two) have a key no smaller than e. We put e into this position and move all elements on path p up by one position. In this way, the heap property is restored. Clearly, e is no larger than the elements stored in the children. Also moving the elements on p up by one position maintains the heap property because p is always extended to the child with the smaller key. The strategy is most easily formulated as a recursive procedure. A call of the following procedure, $\mathit{siftDown}(i)$, 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. Why is it important that the path is always extended to the child with the smaller key? Reformulate *siftDown* as a while-loop.

Exercise 6.6. Our current implementation of *siftDown* needs about $2\log n$ element comparisons. Show how to reduce this to $\log n + \mathrm{O}(\log\log n)$. Hint: Determine the path p first and then perform a binary search on this path to find the proper position for $h[1]$. Section 6.6 has more on variants of *siftDown*.

We can obviously build a heap from n elements by inserting them one after the other in $\mathrm{O}(n \log n)$ 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 6.7 (*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 integer keys 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 height $k = \lfloor \log n \rfloor$. *insert* and *deleteMin* explore one root-to-leaf path and hence have logarithmic running time; min returns the content 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.15). □

Heaps are the basis of *heapsort*. We first *build* a heap from the elements and then repeatedly perform *deleteMin*. Before the ith *deleteMin* operation, the ith 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 of *insert* and of deleting the maximum.

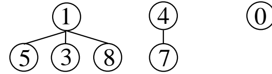
Heaps do not directly implement the addressable priority queues, since elements are moved around in the array h during insertion and deletion. Thus the array indices cannot be used as handles.

Exercise 6.8 (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 6.9 (bulk insertion).** Design an algorithm for inserting k new elements into an n-element heap. Give an algorithm that runs in time $O(k \log k + \log n)$. 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 more flexible structures. The single, left-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. Here is an example of a heap-ordered forest for the set $\{0,1,3,4,5,7,8\}$:



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* of 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 6.2 summarizes the commonalities. In order to keep track of the current minimum, we maintain a 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 on its own.

An *insert* adds a new single-node tree to the forest. So, a sequence of n *insert*s 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 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.

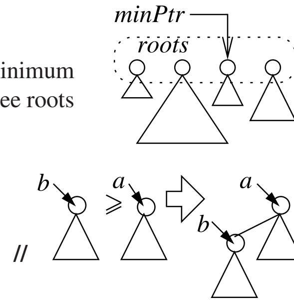
Class *Handle* = **Pointer to** *PQItem*

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 $a \le b$

remove b from *roots*

make a the parent of b

Procedure *combine*$(a,b : Handle)$

assert a and b are tree roots

if $a \le b$ **then** $link(a,b)$ **else** $link(b,a)$

Procedure *newTree*$(h : Handle)$

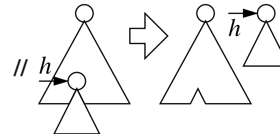
$roots := roots \cup \{h\}$

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*

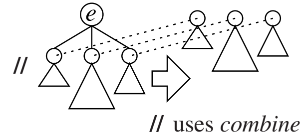
e:= the *Element* stored in *minPtr*

foreach child h of the root at *minPtr* **do** $cut(h)$

dispose *minPtr*

perform some rebalancing and update *minPtr*

return e



// uses *combine*

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. 6.2. Addressable priority queues

6.2.1 Pairing Heaps

Pairing heaps [114] use a very simple technique for rebalancing. Pairing heaps are very efficient in practice. However, a full theoretical analysis is missing.

We present a simple variant of pairing heap that also has good provable bounds. There is always exactly one tree, and nodes may have an arbitrary number of children. Whenever an operation creates several roots, a rebalancing operation is necessary. The most complex rebalancing is done after a *deleteMin*. The root contains an element with a minimum key. After removal of the root, the children of the old root form a sequence $\langle r_1, \ldots, r_k \rangle$ of roots. They are combined into a single tree in the following *two-pass* process. In the first pass, the trees are combined in pairs, i.e., the trees with roots r_1 and r_2, r_3 and r_4, and so on, are joined by calls of *combine*. The resulting $\lceil k/2 \rceil$ trees are then combined into a single tree in the second pass. The last tree is joined with the next to last, the resulting tree is joined with the last tree but two, and so on. Figure 6.3 shows an example. The operations *insert*, *decreaseKey* and *merge* generate pairs of roots. They are simply combined into a single tree by a call of *combine*.

Exercise 6.10 (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 6.6 gives an example.

***Exercise 6.11 (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 6.6 gives an example.

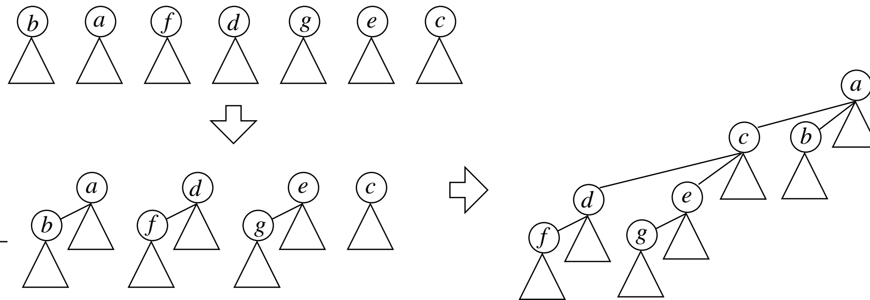


Fig. 6.3. The *deleteMin* operation for pairing heaps makes two passes over the nodes that became roots after the deletion of the old root. In the first pass, roots are combined pairwise. In the second pass, the roots are scanned sequentially from right to left and, in each step, the last two roots are joined. In this example, e becomes the child of c, then d becomes the child of c, and finally c becomes the child of a.

6.2.2 *Fibonacci Heaps

Fibonacci heaps [115] use more intensive balancing operations than do pairing heaps. This paves the way for a theoretical analysis. In particular, we obtain logarithmic 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 link it to its parent, one child, and two siblings; see Fig. 6.6. The children of each node form a doubly linked circular list using the sibling pointers. The sibling pointers of the root nodes are used to represent the set *roots* of all 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 simply the number of its children. In Fibonacci heaps, *deleteMin* links only 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 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 all roots. When scanning a root of rank i, inspect the ith bucket. If the ith bucket is empty, then put the root there. If the bucket is nonempty, then combine the two trees into one. This empties the ith 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 Fig. 6.4.

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 $\mathrm{O}(maxRank)$. The reader must be familiar with the technique of amortized analysis (see Sect. 3.5) 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* $\mathrm{O}(maxRank)$.

Proof. A *deleteMin* first calls *newTree* at most *maxRank* times (since the degree of the root containing the old minimum is bounded by *maxRank*) and then initializes an array of size *maxRank*. So far, the running time is $\mathrm{O}(maxRank)$, and at most *maxRank* new tokens need to be created. The remaining time is proportional to the

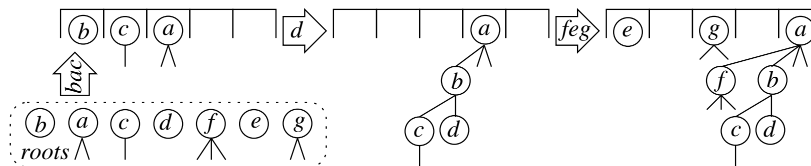


Fig. 6.4. An example of the development of the bucket array during execution of *deleteMin* for a Fibonacci heap. The arrows indicate the roots scanned. Note that scanning d leads to a cascade of three combine operations.

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 *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. 6.5. 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.

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 [324] is to keep all trees in the heap binomial. However, this causes logarithmic cost for a *decreaseKey*.

***Exercise 6.12 (binomial heaps).** Work out the details of this idea. Assume that the key of v is decreased and becomes smaller than the key stored in its parent. Cut the following links. For each nonroot ancestor w of v (this includes v), cut the link to its parent. Moreover, for each such node w, cut the links from all siblings of w of rank higher than w to their parents. Show that all resulting trees are binomial. Then combine trees of equal rank until there is at most one tree of each rank. Argue that the cost of *decreaseKey* is logarithmic.

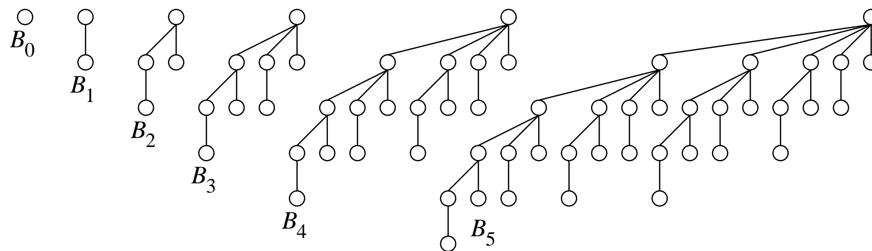


Fig. 6.5. The binomial trees of ranks 0 to 5

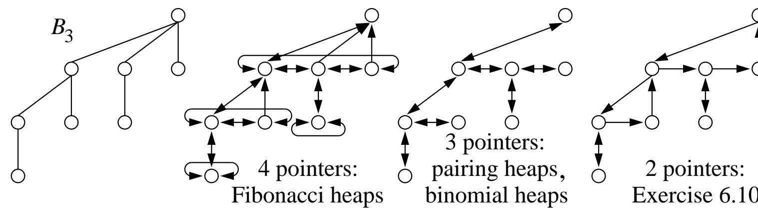


Fig. 6.6. Three ways to represent trees of nonuniform degree. The binomial tree of rank three, B_3, is used as an example.

Fibonacci heaps allow the trees to go out of shape but in a controlled way. The idea is surprisingly simple and is based on the amortized analysis of binary counters; see Sect. 3.4.3. We introduce an additional flag for each node. A node may or may not be marked. 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 has already been marked, we cut x, remove the mark from x, and attempt to mark x's parent. If x's parent is already marked, then we continue in the same way. 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 6.7 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 $0,1,1,2,3,5,8,\ldots$ 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 $F_{i+2} \geq ((1+\sqrt{5})/2)^i \geq 1.618^i$ for all $i \geq 0$.

Exercise 6.13. Prove that $F_{i+2} \geq ((1+\sqrt{5})/2)^i \geq 1.618^i$ for all $i \geq 0$ by induction.

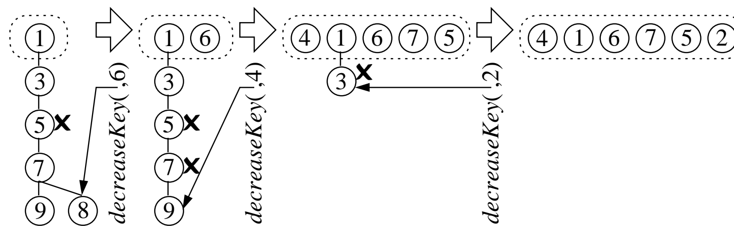


Fig. 6.7. Cascading cuts. Marks are drawn as crosses. Note that roots are never marked.

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 jth 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 smallest number S_i of nodes in a tree whose root has rank i. Clearly, $S_0 = 1$ and $S_1 = 2$. Also, $S_i \geq 2 + S_0 + S_1 + \cdots + S_{i-2}$, since for $j \geq 2$ the number of nodes in the subtree with root w_j is at least S_{j-2}, and there are 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. remove and deleteMin take an amortized time logarithmic in the size of the queue.*

Exercise 6.14. Describe a variant of Fibonacci heaps where all roots have distinct ranks. Hint: Whenever a new root comes into existence, immediately check whether there is already a root of the same rank. If so, combine.

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 (nonaddressable) 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. 5.12.1. Figure 6.8 depicts the basic design. The data structure consists of two priority queues Q and Q' (e.g., binary heaps) and k

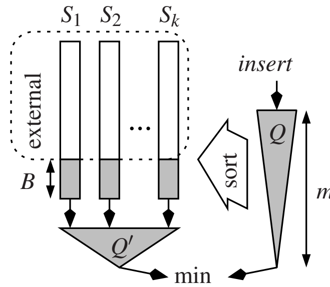


Fig. 6.8. Schematic view of an external-memory priority queue.

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**
    $k{+}{+};\quad S_k := sort(Q);\quad Q := \emptyset;\quad Q'.insert((S_k.popFront, k))$
  $Q.insert(e)$

Q or Q' contains an element with the minimum key. We find it by comparing their minimum elements. If the minimum is in Q' and comes from sequence S_i, the next element in S_i is inserted into Q':

Function *deleteMin*
  if $\min Q \le \min Q'$ **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 algorithm described in Sect. 5.12.1. Each inserted element is written to external memory at most once and fetched back to internal memory at most once. Since all accesses to external memory transfer full blocks, the I/O requirement of our algorithm is at most n/B for n queue operations.

The total requirement for internal memory is at most the space for $m+kB+2k$ elements. 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 and few deletions, 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 8 GB of main memory, 8-byte elements, and 1 MB disk blocks, we have $M=2^{30}$ and $B=2^{17}$ (measured in elements). We can then perform about 2^{41} insertions – enough for 16 TB of data. Similarly to external mergesort, we can handle larger amounts of data by performing multiple phases of multiway merging; see [54, 271]. 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 [271]. There are also implementations for external memory [88].

6.4 Parallel Priority Queues

We first have to decide what a parallel priority queue should be. For example, are we allowing concurrent queue operations or are we only parallelizing single operations of an otherwise sequential queue? A simple answer is that we only want to parallelize single queue operations. There are indeed such data structures; see [55]. However, the maximum speedup we can hope for is $\mathrm{O}(\log n)$, and the constant factor for PE interactions is likely to eat up much of that advantage. We are not aware of practical implementations achieving high speedups. In practice, a little bit of speedup can be obtained by parallelizing the sorting and multiway merging operations in priority queues based on the external queues described in Sect. 6.3; see [35, 45] for details.

Another view on parallel priority queues asks for concurrent access to a priority queue. However, as with the concurrent FIFOs discussed in Sect. 3.7, this raises two severe issues. First of all, it becomes nonobvious how to define the semantics of the queue operations. For example, if several PEs want to extract the minimum, should they all receive the same element? This definition restricts parallelism on the level of the program using the queue and – the second problem – leads to contention, as several PEs will have to access the same element.

We shall therefore concentrate on a third view of parallel priority queues – bulk parallel deletion on a distributed-memory machine. The operation $\mathit{deleteMin}^*(k)$ is executed collectively by all PEs and removes the k globally smallest elements from the queue.[2] Insertions still insert individual elements in an asynchronous fashion.

Our strategy for this kind of parallel priority queue is very simple. Each PE maintains a local priority queue Q. Inserted elements are sent asynchronously to a PE chosen uniformly at random, and inserted there. Thus, each PE has a representative

[2] We should point out that not all application programs can make use of bulk deletion. For example, Dijkstra's algorithm for computing shortest paths (Section 10.3) loses its label-setting property (see Theorem 10.6) when we scan several nodes at the same time.

```
Function deleteMin*(k : ℕ) : Sequence of Element
  result:=⟨⟩
  m := initialBufferSize(k, p)          // choose initial # of removed elements such that with
  repeat                               //  high probability, one iteration of this loop is enough.
     for i := 1 to m do  result.pushBack(Q.deleteMin)          // extract result candidates
     (e_k, k_here) := parSelect2(result, k)                              // see Fig. 5.16
  until ∀i : e_k ≤ Q@i.min                         // no result is missing. Needs all-reduce-and
  for i := 1 to |result| − k_here do  Q.insert(result.popBack)      // reinsert nonresult elements
  return result
```

Fig. 6.9. SPMD pseudocode for bulk *deleteMin* of the k globally smallest elements from a parallel priority queue.

sample of the overall data set. In particular, it can be shown that all globally small elements are among the locally small elements with high probability.

Lemma 6.6. *If $\ell \geq 1$ and $k = \Omega(p \log p)$, then with probability at least $1 - 1/k^{\ell}$, the k globally smallest elements are among the $\mathrm{O}(\ell k/p)$ locally smallest elements of each queue.*

Exercise 6.15. Prove Lemma 6.6. Hint: Use the Chernoff bound (A.7).

Exercise 6.16. Refine Lemma 6.6 and derive the constant factors in the O-terms. What happens for $k = o(p \log p)$?

The operation *deleteMin** exploits Lemma 6.6; see also the pseudocode in Fig. 6.9 and the example in Fig. 6.10. It suffices to look at the locally smallest elements of each local queue to find the globally smallest ones. The function *initialBufferSize* makes an initial guess m of how many local elements will be needed. A simple choice that works well for $k = \Omega(p \log p)$ is $m = 2k/p$. For $k \gg p \log p$, we can use a value of m very close to k/p. Exercise 6.16 asks you to work out the constants in more detail. The m locally smallest elements are removed. They are tentativly moved to the result set. Then, parallel selection (see Fig. 5.16) is used to identify the k globally smallest elements among these result candidates. Let e_k denote the kth smallest result candidate. If there is no element smaller than e_k in any local queue, the result set contains the k globally smallest elements. Otherwise, we continue to remove elements from the local queues. We complete the operation by reinserting the result candidates that have not been selected for the final result.

Exercise 6.17. Our implementation of *deleteMin** delivers the results locally irrespective of load imbalance. Explain how to modify *deleteMin** such that every PE gets the same number of result elements up to rounding up or down. Hint: Use prefix sums.

Theorem 6.7. *For $k = \Omega(p \log p)$, the operation deleteMin* works in expected time $\mathrm{O}(\frac{k}{p} \log n)$, where n is the total queue size.*

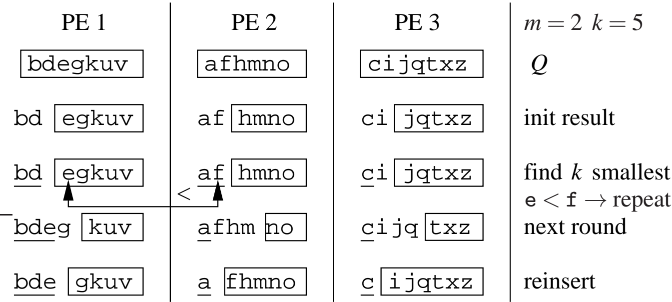


Fig. 6.10. Deleting the $k = 5$ smallest elements from a queue containing $\langle a,b,c,d,e,f,g,h,i,j,k,m,n,o,q,t,x,z\rangle$ distributed over three PEs

Proof. (Outline.) By Lemma 6.6, we can choose $m = \Theta(k/p)$ and achieve a constant expected number of iterations of the repeat loop. An iteration of the loop takes time $O(m\log n)$ for local *deleteMin* operations and expected time $O(k/p+\log p)$ for parallel selection; see Fig. 5.16. Testing for loop termination takes time $O(\log p)$ for an all-reduce-and operation; see Sect. 13.2. Reinserting unneeded elements takes no more time than deleting them in the first place. □

Exercise 6.18. Strengthen Theorem 6.7 and show a time bound of $O(\frac{k}{p}\log\frac{n}{p})$ for $n = \Omega(p\log p)$.

Exercise 6.19. Show that a *deleteMin** is communication-efficient in the sense that it has total communication cost $O(\alpha\log p)$ regardless of n and k. Are insertions equally efficient with respect to communication overhead?

6.4.1 Refinements

An inefficiency of our implementation of *deleteMin** is that it moves elements back and forth between the result buffer and Q. For $k = o(p\log p)$ this also becomes an issue asymptotically. For example, for $k = \Theta(p)$, it is known that we need $m = \Omega(\log p/\log\log p)$ locally removed elements, although on average only a constant number of elements per PE is actually returned. This problem can be avoided by keeping the result buffer around, emptying it only occasionally. It can be shown that this allows efficient operation all the way down to $k = O(p)$ [270].

We can reduce communication overhead by a significant constant factor if we allow some fluctuations in the size of the returned result set. Rather than running a full-fledged selection algorithm with several iterations of sample sorting and partitioning, we can just sort a single sample to determine a single pivot whose expected global rank is k. We then simply return all result candidates bounded by this rank; see also [157].

6.5 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 6.20. Give iterative versions of *siftDown* and *siftUp*. Also, replace the *swap* operations.

Some compilers do the recursion elimination for you.

As with 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 the bit string representing x as an integer.

6.5.1 C++

The STL class *priority_queue* offers nonaddressable priority queues implemented using binary heaps. The external-memory library STXXL [88] offers an external-memory priority queue. LEDA [194] and LEMON (Library for Efficient Modeling and Optimization in Networks) [200] implement a wide variety of addressable priority queues, including pairing heaps and Fibonacci heaps.

6.5.2 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 $\Theta(n)$. JGraphT [166] offers an implementation of Fibonacci heaps.

6.6 Historical Notes and Further Findings

There is an interesting internet survey[3] 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 $\log n + \mathrm{O}(\log\log n)$ 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, does the rightmost element fill the hole and it is subsequently sifted up. When used for sorting, the resulting *bottom-up heapsort* requires $\frac{3}{2}n\log n + \mathrm{O}(n)$ comparisons in the worst case and $n\log n + \mathrm{O}(1)$ in the average case [105, 283, 327]. 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 at first. The explanation is that the bottom-up variant usually causes more cache faults that the standard variant and that the number of hard-to-predict branch operations is not reduced. Cache faults and incorrectly predicted branch operations have a larger influence on running time than does the number of comparisons; see [279] for more discussion. d-ary heaps, in which a node has d children instead of only two, outperform binary heaps in practice; see [**?**] for experiments.

The recursive *buildHeap* routine in Exercise 6.7 is an example of a *cache-oblivious algorithm* [116]. This algorithm is efficient in the external-memory model even though it does not explicitly use the block size or cache size.

Pairing heaps [114] have constant amortized complexity for *insert* and *merge* [159] and logarithmic amortized complexity for *deleteMin*. The best analysis is due to Pettie [252]. Fredman [112] has given operation sequences consisting of $\mathrm{O}(n)$ insertions and *deleteMin*s and $\mathrm{O}(n\log n)$ *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. Haeupler et al. [140] introduced a variant of pairing heaps that match the performance of Fibonacci heaps.

The family of addressable priority queues is large. Vuillemin [324] introduced binomial heaps, and Fredman and Tarjan [115] invented Fibonacci heaps. Høyer [156] described additional balancing operations that are akin to the operations used for search trees. One such operation yields *thin heaps* [173], which have performance guarantees similar to those of 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 [53]. The basic idea is to tolerate violations of the heap property and to continuously invest some work in

[3] www.leekillough.com/heaps/survey_results.html

reducing these violations. Other interesting variants are *fat heaps* [173] and *hollow heaps* [144].

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 $\mathrm{O}(\log\log n)$ time for *deleteMin* [223, 313]. Using randomization, the time bound can even be reduced to $\mathrm{O}\left(\sqrt{\log\log n}\right)$ [142]. The algorithms are fairly complex. However, integer priority queues for operation sequences satisfying a *monotonicity property* are simple and practical. Section 10.3 gives examples. *Calendar queues* [58] are popular in the discrete-event simulation community. These are a variant of the *bucket queues* described in Sect. 10.5.1.