II

Priority Queues

5

Leftist Trees

Sartaj Sahni
University of Florida

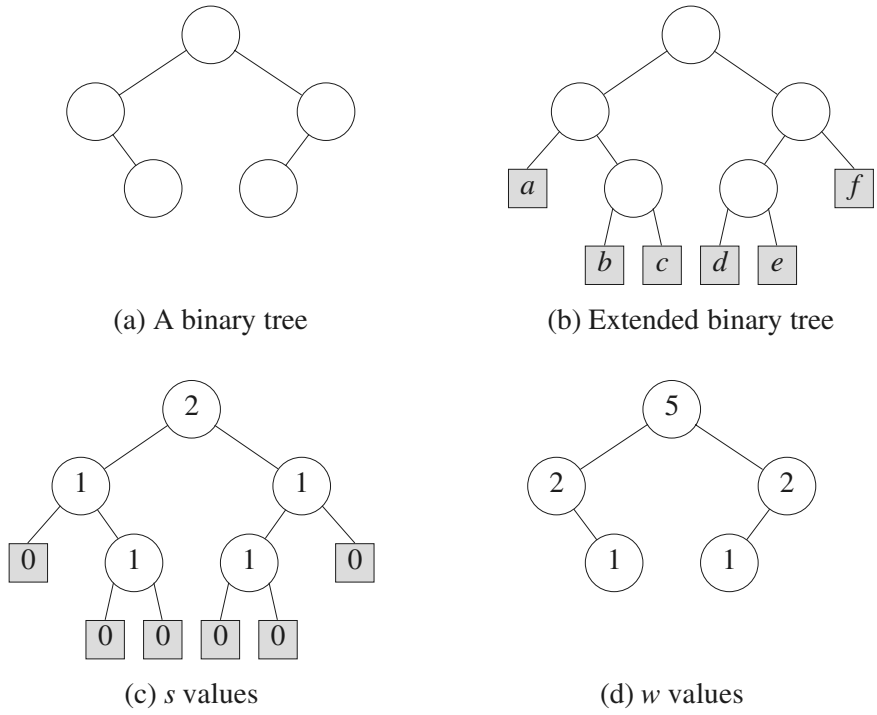
5.1 Introduction

A single-ended priority queue (or simply, a priority queue) is a collection of elements in which each element has a priority. There are two varieties of priority queues—max and min. The primary operations supported by a max (min) priority queue are (a) find the element with maximum (minimum) priority, (b) insert an element, and (c) delete the element whose priority is maximum (minimum). However, many authors consider additional operations such as (d) delete an arbitrary element (assuming we have a pointer to the element), (e) change the priority of an arbitrary element (again assuming we have a pointer to this element), (f) meld two max (min) priority queues (i.e., combine two max (min) priority queues into one), and (g) initialize a priority queue with a nonzero number of elements.

Several data structures: e.g., heaps (Chapter 3), leftist trees [2, 5], Fibonacci heaps [7] (Chapter 7), binomial heaps [1] (Chapter 7), skew heaps [11] (Chapter 6), and pairing heaps [6] (Chapter 7) have been proposed for the representation of a priority queue. The different data structures that have been proposed for the representation of a priority queue differ in terms of the performance guarantees they provide. Some guarantee good performance on a per operation basis while others do this only in the amortized sense. Max (min) heaps permit one to delete the max (min) element and insert an arbitrary element into an n element priority queue in $O(\log n)$ time per operation; a find max (min) takes $O(1)$ time. Additionally, a heap is an implicit data structure that has no storage overhead associated with it. All other priority queue structures are pointer-based and so require additional storage for the pointers.

Max (min) leftist trees also support the insert and delete max (min) operations in $O(\log n)$ time per operation and the find max (min) operation in $O(1)$ time. Additionally, they permit us to meld pairs of priority queues in logarithmic time.

The remaining structures do not guarantee good complexity on a per operation basis. They do, however, have good amortized complexity. Using Fibonacci heaps, binomial queues, or skew heaps, find max (min), inserts and melds take $O(1)$ time (actual and amortized) and a delete max (min) takes $O(\log n)$ amortized time. When a pairing heap is



(a) A binary tree (b) Extended binary tree

(c) s values (d) w values

FIGURE 5.1: s and w values.

used, the amortized complexity is $O(1)$ for find max (min) and insert (provided no decrease key operations are performed) and $O(logn)$ for delete max (min) operations [12]. Jones [8] gives an empirical evaluation of many priority queue data structures.

In this chapter, we focus on the leftist tree data structure. Two varieties of leftist trees– height-biased leftist trees [5] and weight-biased leftist trees [2] are described. Both varieties of leftist trees are binary trees that are suitable for the representation of a single-ended priority queue. When a max (min) leftist tree is used, the traditional single-ended priority queue operations– find max (min) element, delete/remove max (min) element, and insert an element–take, respectively, $O(1)$, $O(\log n)$ and $O(\log n)$ time each, where n is the number of elements in the priority queue. Additionally, an n-element max (min) leftist tree can be initialized in $O(n)$ time and two max (min) leftist trees that have a total of n elements may be melded into a single max (min) leftist tree in $O(\log n)$ time.

5.2 Height-Biased Leftist Trees

5.2.1 Definition

Consider a binary tree in which a special node called an **external node** replaces each empty subtree. The remaining nodes are called **internal nodes**. A binary tree with external nodes added is called an **extended binary tree**. Figure 5.1(a) shows a binary tree. Its corresponding extended binary tree is shown in Figure 5.1(b). The external nodes appear as shaded boxes. These nodes have been labeled a through f for convenience.

Let $s(x)$ be the length of a shortest path from node x to an external node in its subtree. From the definition of $s(x)$, it follows that if x is an external node, its s value is 0.

Furthermore, if x is an internal node, its s value is

$$\min\{s(L), s(R)\} + 1$$

where L and R are, respectively, the left and right children of x. The s values for the nodes of the extended binary tree of Figure 5.1(b) appear in Figure 5.1(c).

DEFINITION 5.1 [Crane [5]] A binary tree is a **height-biased leftist tree (HBLT)** iff at every internal node, the s value of the left child is greater than or equal to the s value of the right child.

The binary tree of Figure 5.1(a) is not an HBLT. To see this, consider the parent of the external node a. The s value of its left child is 0, while that of its right is 1. All other internal nodes satisfy the requirements of the HBLT definition. By swapping the left and right subtrees of the parent of a, the binary tree of Figure 5.1(a) becomes an HBLT.

THEOREM 5.1 *Let x be any internal node of an HBLT.*

(a) *The number of nodes in the subtree with root x is at least $2^{s(x)} - 1$.*
(b) *If the subtree with root x has m nodes, $s(x)$ is at most $\log_2(m+1)$.*
(c) *The length, $rightmost(x)$, of the right-most path from x to an external node (i.e., the path obtained by beginning at x and making a sequence of right-child moves) is $s(x)$.*

Proof From the definition of $s(x)$, it follows that there are no external nodes on the $s(x)-1$ levels immediately below node x (as otherwise the s value of x would be less). The subtree with root x has exactly one node on the level at which x is, two on the next level, four on the next, $\cdots$, and $2^{s(x)-1}$ nodes $s(x)-1$ levels below x. The subtree may have additional nodes at levels more than $s(x)-1$ below x. Hence the number of nodes in the subtree x is at least $\sum_{i=0}^{s(x)-1} 2^i = 2^{s(x)} - 1$. Part (b) follows from (a). Part (c) follows from the definition of s and the fact that, in an HBLT, the s value of the left child of a node is always greater than or equal to that of the right child.

DEFINITION 5.2 A **max tree** (**min tree**) is a tree in which the value in each node is greater (less) than or equal to those in its children (if any).

Some max trees appear in Figure 5.2, and some min trees appear in Figure 5.3. Although these examples are all binary trees, it is not necessary for a max tree to be binary. Nodes of a max or min tree may have an arbitrary number of children.

DEFINITION 5.3 A **max HBLT** is an HBLT that is also a max tree. A **min HBLT** is an HBLT that is also a min tree.

The max trees of Figure 5.2 as well as the min trees of Figure 5.3 are also HBLTs; therefore, the trees of Figure 5.2 are max HBLTs, and those of Figure 5.3 are min HBLTs. A max priority queue may be represented as a max HBLT, and a min priority queue may be represented as a min HBLT.

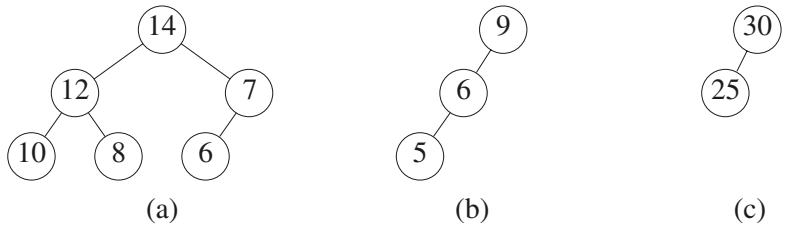


FIGURE 5.2: Some max trees.

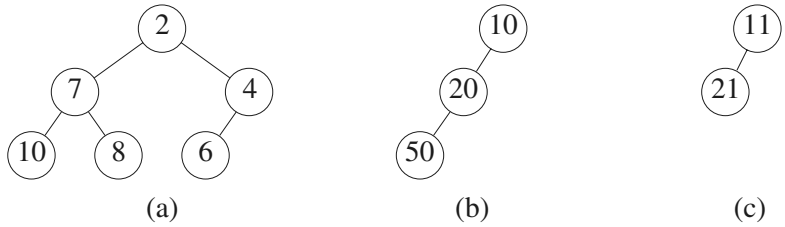


FIGURE 5.3: Some min trees.

5.2.2 Insertion into a Max HBLT

The insertion operation for max HBLTs may be performed by using the max HBLT meld operation, which combines two max HBLTs into a single max HBLT. Suppose we are to insert an element x into the max HBLT H. If we create a max HBLT with the single element x and then meld this max HBLT and H, the resulting max HBLT will include all elements in H as well as the element x. Hence an insertion may be performed by creating a new max HBLT with just the element that is to be inserted and then melding this max HBLT and the original.

5.2.3 Deletion of Max Element from a Max HBLT

The max element is in the root. If the root is deleted, two max HBLTs, the left and right subtrees of the root, remain. By melding together these two max HBLTs, we obtain a max HBLT that contains all elements in the original max HBLT other than the deleted max element. So the delete max operation may be performed by deleting the root and then melding its two subtrees.

5.2.4 Melding Two Max HBLTs

Since the length of the right-most path of an HBLT with n elements is $O(\log n)$, a meld algorithm that traverses only the right-most paths of the HBLTs being melded, spending $O(1)$ time at each node on these two paths, will have complexity logarithmic in the number of elements in the resulting HBLT. With this observation in mind, we develop a meld algorithm that begins at the roots of the two HBLTs and makes right-child moves only.

The meld strategy is best described using recursion. Let A and B be the two max HBLTs that are to be melded. If one is empty, then we may use the other as the result. So assume that neither is empty. To perform the meld, we compare the elements in the two roots. The root with the larger element becomes the root of the melded HBLT. Ties may be broken

arbitrarily. Suppose that A has the larger root and that its left subtree is L. Let C be the max HBLT that results from melding the right subtree of A and the max HBLT B. The result of melding A and B is the max HBLT that has A as its root and L and C as its subtrees. If the s value of L is smaller than that of C, then C is the left subtree. Otherwise, L is.

Example 5.1

Consider the two max HBLTs of Figure 5.4(a). The s value of a node is shown outside the node, while the element value is shown inside. When drawing two max HBLTs that are to be melded, we will always draw the one with larger root value on the left. Ties are broken arbitrarily. Because of this convention, the root of the left HBLT always becomes the root of the final HBLT. Also, we will shade the nodes of the HBLT on the right.

Since the right subtree of 9 is empty, the result of melding this subtree of 9 and the tree with root 7 is just the tree with root 7. We make the tree with root 7 the right subtree of 9 temporarily to get the max tree of Figure 5.4(b). Since the s value of the left subtree of 9 is 0 while that of its right subtree is 1, the left and right subtrees are swapped to get the max HBLT of Figure 5.4(c).

Next consider melding the two max HBLTs of Figure 5.4(d). The root of the left subtree becomes the root of the result. When the right subtree of 10 is melded with the HBLT with root 7, the result is just this latter HBLT. If this HBLT is made the right subtree of 10, we get the max tree of Figure 5.4(e). Comparing the s values of the left and right children of 10, we see that a swap is not necessary.

Now consider melding the two max HBLTs of Figure 5.4(f). The root of the left subtree is the root of the result. We proceed to meld the right subtree of 18 and the max HBLT with root 10. The two max HBLTs being melded are the same as those melded in Figure 5.4(d). The resultant max HBLT (Figure 5.4(e)) becomes the right subtree of 18, and the max tree of Figure 5.4(g) results. Comparing the s values of the left and right subtrees of 18, we see that these subtrees must be swapped. Swapping results in the max HBLT of Figure 5.4(h).

As a final example, consider melding the two max HBLTs of Figure 5.4(i). The root of the left max HBLT becomes the root of the result. We proceed to meld the right subtree of 40 and the max HBLT with root 18. These max HBLTs were melded in Figure 5.4(f). The resultant max HBLT (Figure 5.4(g)) becomes the right subtree of 40. Since the left subtree of 40 has a smaller s value than the right has, the two subtrees are swapped to get the max HBLT of Figure 5.4(k). Notice that when melding the max HBLTs of Figure 5.4(i), we first move to the right child of 40, then to the right child of 18, and finally to the right child of 10. All moves follow the right-most paths of the initial max HBLTs.

5.2.5 Initialization

It takes $O(n \log n)$ time to initialize a max HBLT with n elements by inserting these elements into an initially empty max HBLT one at a time. To get a linear time initialization algorithm, we begin by creating n max HBLTs with each containing one of the n elements. These n max HBLTs are placed on a FIFO queue. Then max HBLTs are deleted from this queue in pairs, melded, and added to the end of the queue until only one max HBLT remains.

Example 5.2

We wish to create a max HBLT with the five elements 7, 1, 9, 11, and 2. Five single-element max HBLTs are created and placed in a FIFO queue. The first two, 7 and 1,

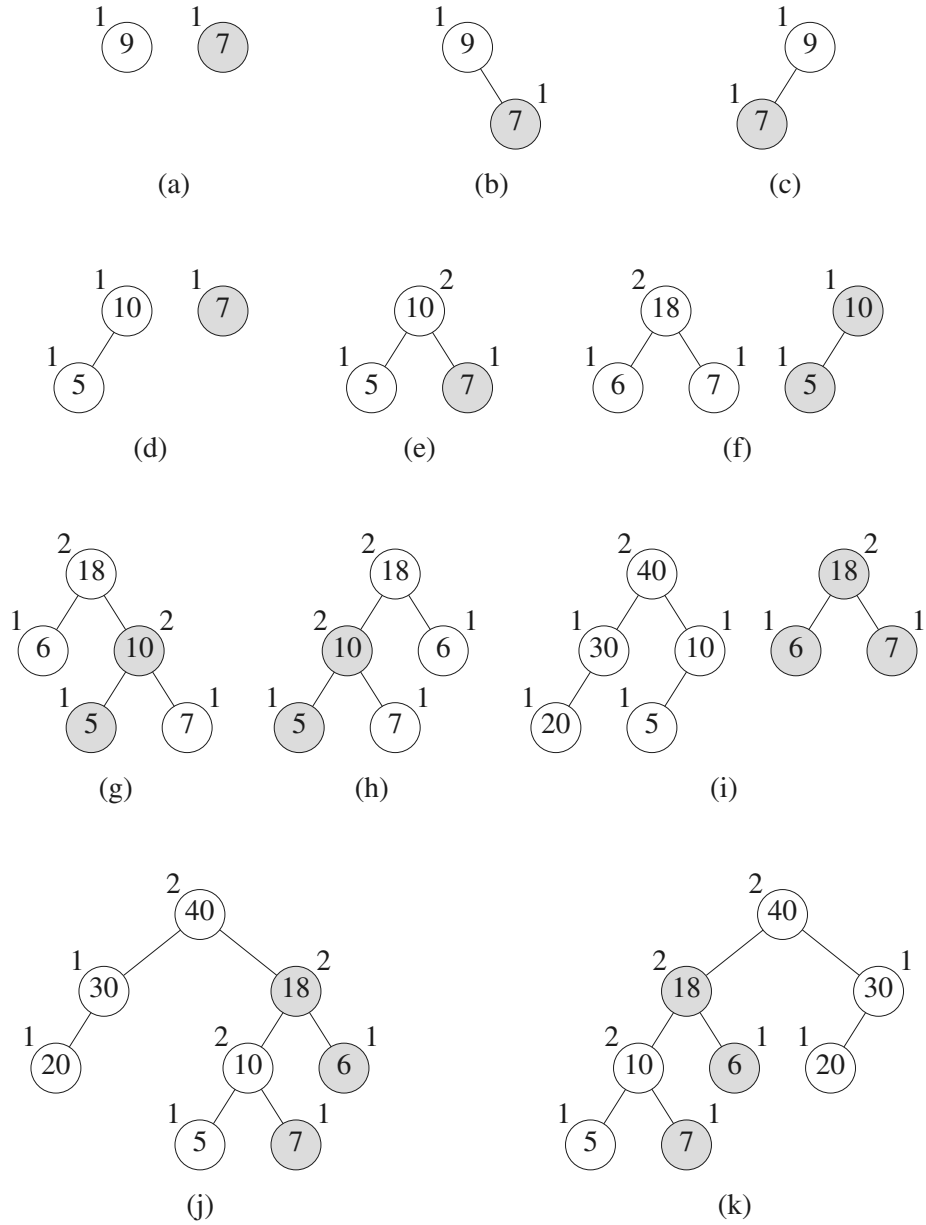


FIGURE 5.4: Melding max HBLTs.

are deleted from the queue and melded. The result (Figure 5.5(a)) is added to the queue. Next the max HBLTs 9 and 11 are deleted from the queue and melded. The result appears in Figure 5.5(b). This max HBLT is added to the queue. Now the max HBLT 2 and that of Figure 5.5(a) are deleted from the queue and melded. The resulting max HBLT (Figure 5.5(c)) is added to the queue. The next pair to be deleted from the queue consists of the max HBLTs of Figures Figure 5.5 (b) and (c). These HBLTs are melded to get the max HBLT of Figure 5.5(d). This max HBLT is added to the queue. The queue now has just one max HBLT, and we are done with the initialization.

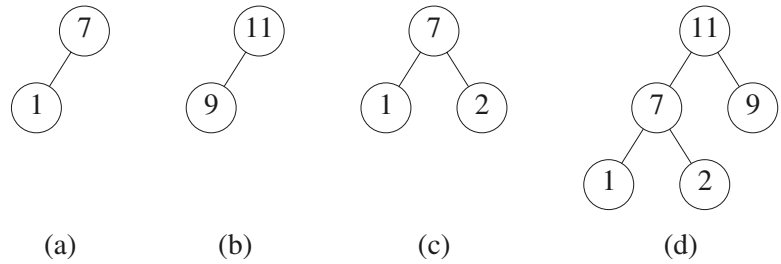


FIGURE 5.5: Initializing a max HBLT.

For the complexity analysis of of the initialization operation, assume, for simplicity, that n is a power of 2. The first $n/2$ melds involve max HBLTs with one element each, the next $n/4$ melds involve max HBLTs with two elements each; the next $n/8$ melds are with trees that have four elements each; and so on. The time needed to meld two leftist trees with 2^i elements each is $O(i+1)$, and so the total time for the initialization is

$$O(n/2 + 2 * (n/4) + 3 * (n/8) + \cdots) = O(n \sum \frac{i}{2^i}) = O(n)$$

5.2.6 Deletion of Arbitrary Element from a Max HBLT

Although deleting an element other than the max (min) element is not a standard operation for a max (min) priority queue, an efficient implementation of this operation is required when one wishes to use the generic methods of Cho and Sahni [3] and Chong and Sahni [4] to derive efficient mergeable double-ended priority queue data structures from efficient single-ended priority queue data structures. From a max or min leftist tree, we may remove the element in any specified node *theNode* in $O(\log n)$ time, making the leftist tree a suitable base structure from which an efficient mergeable double-ended priority queue data structure may be obtained [3, 4].

To remove the element in the node *theNode* of a height-biased leftist tree, we must do the following:

1. Detach the subtree rooted at *theNode* from the tree and replace it with the meld of the subtrees of *theNode*.
2. Update s values on the path from *theNode* to the root and swap subtrees on this path as necessary to maintain the leftist tree property.

To update s on the path from *theNode* to the root, we need parent pointers in each node. This upward updating pass stops as soon as we encounter a node whose s value does not change. The changed s values (with the exception of possibly $O(\log n)$ values from moves made at the beginning from right children) must form an ascending sequence (actually, each must be one more than the preceding one). Since the maximum s value is $O(\log n)$ and since all s values are positive integers, at most $O(\log n)$ nodes are encountered in the updating pass. At each of these nodes, we spend $O(1)$ time. Therefore, the overall complexity of removing the element in node *theNode* is $O(\log n)$.

5.3 Weight-Biased Leftist Trees

5.3.1 Definition

We arrive at another variety of leftist tree by considering the number of nodes in a subtree, rather than the length of a shortest root to external node path. Define the weight $w(x)$ of node x to be the number of internal nodes in the subtree with root x. Notice that if x is an external node, its weight is 0. If x is an internal node, its weight is 1 more than the sum of the weights of its children. The weights of the nodes of the binary tree of Figure 5.1(a) appear in Figure 5.1(d)

DEFINITION 5.4 [Cho and Sahni [2]] A binary tree is a **weight-biased leftist tree (WBLT)** iff at every internal node the w value of the left child is greater than or equal to the w value of the right child. A max (min) WBLT is a max (min) tree that is also a WBLT.

Note that the binary tree of Figure 5.1(a) is not a WBLT. However, all three of the binary trees of Figure 5.2 are WBLTs.

THEOREM 5.2 *Let x be any internal node of a weight-biased leftist tree. The length, $rightmost(x)$, of the right-most path from x to an external node satisfies*

$$rightmost(x) \leq \log_2(w(x)+1).$$

Proof The proof is by induction on $w(x)$. When $w(x) = 1$, $rightmost(x) = 1$ and $\log_2(w(x)+1) = \log_2 2 = 1$. For the induction hypothesis, assume that $rightmost(x) \leq \log_2(w(x)+1)$ whenever $w(x) < n$. Let $RightChild(x)$ denote the right child of x (note that this right child may be an external node). When $w(x) = n$, $w(RightChild(x)) \leq (n-1)/2$ and $rightmost(x) = 1 + rightmost(RightChild(x)) \leq 1 + \log_2((n-1)/2+1) = 1 + \log_2(n+1) - 1 = \log_2(n+1)$.

5.3.2 Max WBLT Operations

Insert, delete max, and initialization are analogous to the corresponding max HBLT operation. However, the meld operation can be done in a single top-to-bottom pass (recall that the meld operation of an HBLT performs a top-to-bottom pass as the recursion unfolds and then a bottom-to-top pass in which subtrees are possibly swapped and s-values updated). A single-pass meld is possible for WBLTs because we can determine the w values on the way down and so, on the way down, we can update w-values and swap subtrees as necessary. For HBLTs, a node's new s value cannot be determined on the way down the tree.

Since the meld operation of a WBLT may be implemented using a single top-to-bottom pass, inserts and deletes also use a single top-to-bottom pass. Because of this, inserts and deletes are faster, by a constant factor, in a WBLT than in an HBLT [2]. However, from a WBLT, we cannot delete the element in an arbitrarily located node, $theNode$, in $O(\log n)$ time. This is because $theNode$ may have $O(n)$ ancestors whose w value is to be updated. So, WBLTs are not suitable for mergeable double-ended priority queue applications [3, 8].

C++ and Java codes for HBLTs and WBLTs may be obtained from [9] and [10], respectively.

Acknowledgment

This work was supported, in part, by the National Science Foundation under grant CCR-9912395.

References

[1] M. Brown, Implementation and analysis of binomial queue algorithms, *SIAM Jr. on Computing*, 7, 3, 1978, 298-319.

[2] S. Cho and S. Sahni, Weight biased leftist trees and modified skip lists, *ACM Jr. on Experimental Algorithmics*, Article 2, 1998.

[3] S. Cho and S. Sahni, Mergeable double-ended priority queues. *International Journal on Foundations of Computer Science*, 10, 1, 1999, 1-18.

[4] K. Chong and S. Sahni, Correspondence based data structures for double ended priority queues. *ACM Jr. on Experimental Algorithmics*, Volume 5, 2000, Article 2, 22 pages.

[5] C. Crane, Linear Lists and Priority Queues as Balanced Binary Trees, Tech. Rep. CS-72-259, Dept. of Comp. Sci., Stanford University, 1972.

[6] M. Fredman, R. Sedgewick, D. Sleator, and R.Tarjan, The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1, 1986, 111-129.

[7] M. Fredman and R. Tarjan, Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms, *JACM*, 34, 3, 1987, 596-615.

[8] D. Jones, An empirical comparison of priority-queue and event-set implementations, *Communications of the ACM*, 29, 4, 1986, 300-311.

[9] S. Sahni, *Data Structures, Algorithms, and Applications in C++*, McGraw-Hill, NY, 1998, 824 pages.

[10] S. Sahni, *Data Structures, Algorithms, and Applications in Java*, McGraw-Hill, NY, 2000, 846 pages.

[11] D. Sleator and R. Tarjan, Self-adjusting heaps, *SIAM Jr. on Computing*, 15, 1, 1986, 52-69.

[12] J. Stasko and J. Vitter, Pairing heaps: Experiments and analysis, *Communications of the ACM*, 30, 3, 1987, 234-249.

6

Skew Heaps

C. Pandu Rangan
Indian Institute of Technology, Madras

6.1 Introduction

Priority Queue is one of the most extensively studied *Abstract Data Types* (ADT) due to its fundamental importance in the context of resource managing systems, such as operating systems. *Priority Queues* work on finite subsets of a totally ordered universal set U. Without any loss of generality we assume that U is simply the set of all non-negative integers. In its simplest form, a *Priority Queue* supports two operations, namely,

- *insert*(x, S) : update S by adding an arbitrary $x \in U$ to S.
- *delete-min*(S) : update S by removing from S the minimum element of S.

We will assume for the sake of simplicity, all the items of S are distinct. Thus, we assume that $x \notin S$ at the time of calling *insert*(x, S). This increases the cardinality of S, denoted usually by $|S|$, by one. The well-known data structure *Heaps*, provide an elegant and efficient implementation of *Priority Queues*. In the *Heap* based implementation, both *insert*(x, S) and *delete-min*(S) take $O(\log n)$ time where $n = |S|$.

Several extensions for the basic *Priority Queues* were proposed and studied in response to the needs arising in several applications. For example, if an operating system maintains a set of jobs, say print requests, in a priority queue, then, always, the jobs with 'high priority' are serviced irrespective of when the job was queued up. This might mean some kind of 'unfairness' for low priority jobs queued up earlier. In order to straighten up the situation, we may extend priority queue to support *delete-max* operation and arbitrarily mix *delete-min* and *delete-max* operations to avoid any undue stagnation in the queue. Such priority queues are called *Double Ended Priority Queues*. It is easy to see that *Heap* is not an appropriate data structure for *Double Ended Priority Queues*. Several interesting alternatives are available in the literature [1] [3] [4]. You may also refer Chapter 8 of this handbook for a comprehensive discussion on these structures.

In another interesting extension, we consider adding an operation called *melding*. A *meld* operation takes two disjoint sets, S_1 and S_2, and produces the set $S = S_1 \cup S_2$. In terms of an implementation, this requirement translates to building a data structure for S, given

the data structures of S_1 and S_2. A *Priority Queue* with this extension is called a *Meldable Priority Queue*. Consider a scenario where an operating system maintains two different priority queues for two printers and one of the printers is down with some problem during operation. *Meldable Priority Queues* naturally model such a situation.

Again, maintaining the set items in *Heaps* results in very inefficient implementation of *Meldable Priority Queues*. Specifically, designing a data structure with $O(\log n)$ bound for each of the *Meldable Priority Queue* operations calls for more sophisticated ideas and approaches. An interesting data structure called *Leftist Trees*, implements all the operations of *Meldable Priority Queues* in $O(\log n)$ time. *Leftist Trees* are discussed in Chapter 5 of this handbook.

The main objective behind the design of a data structure for an ADT is to implement the ADT operations as efficiently as possible. Typically, efficiency of a structure is judged by its worst-case performance. Thus, when designing a data structure, we seek to minimize the worst case complexity of each operation. While this is a most desirable goal and has been theoretically realized for a number of data structures for key ADTs, the data structures optimizing worst-case costs of ADT operations are often very complex and pretty tedious to implement. Hence, computer scientists were exploring alternative design criteria that would result in simpler structures without losing much in terms of performance. In Chapter 13 of this handbook, we show that incorporating randomness provides an attractive alternative avenue for designers of the data structures. In this chapter we will explore yet another design goal leading to simpler structural alternatives without any degrading in overall performance.

Since the data structures are used as basic building blocks for implementing algorithms, a typical execution of an algorithm might consist of a sequence of operations using the data structure over and again. In the worst case complexity based design, we seek to reduce the cost of each operation as much as possible. While this leads to an overall reduction in the cost for the sequence of operations, this poses some constraints on the designer of data structure. We may relax the requirement that the cost of each operation be minimized and perhaps design data structures that seek to minimize the total cost of any sequence of operations. Thus, in this new kind of design goal, we will not be terribly concerned with the cost of any individual operations, but worry about the total cost of any sequence of operations. At first thinking, this might look like a formidable goal as we are attempting to minimize the cost of an arbitrary mix of ADT operations and it may not even be entirely clear how this design goal could lead to simpler data structures. Well, it is typical of a novel and deep idea; at first attempt it may puzzle and bamboozle the learner and with practice one tends to get a good intuitive grasp of the intricacies of the idea. This is one of those ideas that requires some getting used to. In this chapter, we discuss about a data structure called *Skew heaps*. For any sequence of a *Meldable Priority Queue* operations, its total cost on *Skew Heaps* is asymptotically same as its total cost on *Leftist Trees*. However, *Skew Heaps* are a bit simpler than *Leftist Trees*.

6.2 Basics of Amortized Analysis

We will now clarify the subtleties involved in the new design goal with an example. Consider a typical implementation of *Dictionary* operations. The so called Balanced Binary Search Tree structure (BBST) implements these operations in $O(m \log n)$ worst case bound. Thus, the total cost of an arbitrary sequence of m dictionary operations, each performed on a tree of size at most n, will be $O(\log n)$. Now we may turn around and ask: Is there a data structure on which the cost of a sequence of m dictionary operations is $O(m \log n)$ but

individual operations are not constrained to have $O(\log n)$ bound? Another more pertinent question to our discussion - Is that structure simpler than BBST, at least in principle? An affirmative answer to both the questions is provided by a data structure called *Splay Trees*. *Splay Tree* is the theme of Chapter 12 of this handbook.

Consider for example a sequence of m dictionary operations $S_1, S_2, ..., S_m$, performed using a BBST. Assume further that the size of the tree has never exceeded n during the sequence of operations. It is also fairly reasonable to assume that we begin with an empty tree and this would imply $n \leq m$. Let the actual cost of executing S_i be C_i. Then the total cost of the sequence of operations is $C_1 + C_2 + \cdots + C_m$. Since each C_i is $O(\log n)$ we easily conclude that the total cost is $O(m \log n)$. No big arithmetic is needed and the analysis is easily finished. Now, assume that we execute the same sequence of m operations but employ a *Splay Tree* in stead of a BBST. Assuming that c_i is the actual cost of S_i in a *Splay Tree*, the total cost for executing the sequence of operation turns out to be $c_1 + c_2 + \ldots + c_m$. This sum, however, is tricky to compute. This is because a wide range of values are possible for each of c_i and no upper bound other than the trivial bound of $O(n)$ is available for c_i. Thus, a naive, worst case cost analysis would yield only a weak upper bound of $O(nm)$ whereas the actual bound is $O(m \log n)$. But how do we arrive at such improved estimates?

This is where we need yet another powerful tool called *potential function*.

The potential function is purely a conceptual entity and this is introduced only for the sake of computing a sum of widely varying quantities in a convenient way. Suppose there is a function $f : D \rightarrow R^+ \cup \{0\}$, that maps a configuration of the data structure to a non-negative real number. We shall refer to this function as potential function. Since the data type as well as data structures are typically dynamic, an operation may change the configuration of data structure and hence there may be change of potential value due to this change of configuration. Referring back to our sequence of operations $S_1, S_2, \ldots, S_m$, let D_{i-1} denote the configuration of data structure before the executing the operation S_i and D_i denote the configuration after the execution of S_i. The potential difference due to this operation is defined to be the quantity $f(D_i) - f(D_{i-1})$. Let c_i denote the actual cost of S_i. We will now introduce yet another quantity, a_i, defined by

$$a_i = c_i + f(D_i) - f(D_{i-1}).$$

What is the consequence of this definition?

Note that $\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} c_i + f(D_m) - f(D_0)$.

Let us introduce one more reasonable assumption that $f(D_0) = f(\phi) = 0$. Since $f(D) \geq 0$ for all non empty structures, we obtain,

$$\sum a_i = \sum c_i + f(D_m) \geq \sum c_i$$

If we are able to choose cleverly a 'good' potential function so that a_i's have tight, uniform bound, then we can evaluate the sum $\sum a_i$ easily and this bounds the actual cost sum $\sum c_i$. In other words, we circumvent the difficulties posed by wide variations in c_i by introducing new quantities a_i which have uniform bounds. A very neat idea indeed! However, care must be exercised while defining the potential function. A poor choice of potential function will result in a_is whose sum may be a trivial or useless bound for the sum of actual costs. In fact, arriving at the right potential function is an ingenious task, as you will understand by the end of this chapter or by reading the chapter on *Splay Trees*.

The description of the data structures such as *Splay Trees* will not look any different from the description of a typical data structures - it comprises of a description of the organization of the primitive data items and a bunch of routines implementing ADT operations. The key difference is that the routines implementing the ADT operations will not be analyzed for their individual worst case complexity. We will only be interested in the the cumulative effect of these routines in an arbitrary sequence of operations. Analyzing the average potential contribution of an operation in an arbitrary sequence of operations is called *amortized analysis*. In other words, the routines implementing the ADT operations will be analyzed for their *amortized cost*. Estimating the amortized cost of an operation is rather an intricate task. The major difficulty is in accounting for the wide variations in the costs of an operation performed at different points in an arbitrary sequence of operations. Although our design goal is influenced by the costs of sequence of operations, defining the notion of amortized cost of an operation in terms of the costs of sequences of operations leads one nowhere. As noted before, using a potential function to off set the variations in the actual costs is a neat way of handling the situation.

In the next definition we formalize the notion of amortized cost.

DEFINITION 6.1 [Amortized Cost] Let A be an ADT with basic operations $O = \{O_1, O_2, \cdots, O_k\}$ and let D be a data structure implementing A. Let f be a potential function defined on the configurations of the data structures to non-negative real number. Assume further that $f(\Phi) = 0$. Let D' denote a configuration we obtain if we perform an operation O_k on a configuration D and let c denote the actual cost of performing O_k on D. Then, the amortized cost of O_k operating on D, denoted as $a(O_k, D)$, is given by

$$a(O_k, D) = c + f(D') - f(D)$$

If $a(O_k, D) \leq c'g(n)$ for all configuration D of size n, then we say that the amortized cost of O_k is $O(g(n))$.

THEOREM 6.1 *Let D be a data structure implementing an ADT and let $s_1, s_2, \cdots, s_m$ denote an arbitrary sequence of ADT operations on the data structure starting from an empty structure D_0. Let c_i denote actual cost of the operation s_i and D_i denote the configuration obtained which s_i operated on D_{i-1}, for $1 \leq i \leq m$. Let a_i denote the amortized cost of s_i operating on D_{i-1} with respect to an arbitrary potential function. Then,*

$$\sum_{i=1}^{m} c_i \leq \sum_{i=1}^{m} a_i.$$

Proof Since a_i is the amortized cost of s_i working on the configuration D_{i-1}, we have

$$a_i = a(s_i, D_{i-1}) = c_i + f(D_i) - f(D_{i-1})$$

Therefore,

$$\begin{aligned}\sum_{i=1}^{m} a_i &= \sum_{i=1}^{m} c_i + (f(D_m) - f(D_0)) \\ &= f(D_m) + \sum_{i=1}^{m} c_i \text{ (since } f(D_0) = 0) \\ &\geq \sum_{i=1}^{m} c_i\end{aligned}$$

REMARK 6.1 The potential function is common to the definition of amortized cost of all the ADT operations. Since $\sum_{i=1}^{m} a_i \geq \sum_{i=1}^{m} c_i$ holds good for any potential function, a clever choice of the potential function will yield tight upper bound for the sum of actual cost of a sequence of operations.

6.3 Meldable Priority Queues and Skew Heaps

DEFINITION 6.2 [Skew Heaps] A Skew Heap is simply a binary tree. Values are stored in the structure, one per node, satisfying the *heap-order property*: A value stored at a node is larger than the value stored at its parent, except for the root (as root has no parent).

REMARK 6.2 Throughout our discussion, we handle sets with distinct items. Thus a set of n items is represented by a skew heap of n nodes. The minimum of the set is always at the root. On any path starting from the root and descending towards a leaf, the values are in increasing order.

6.3.1 Meldable Priority Queue Operations

Recall that a *Meldable Priority queue* supports three key operations: *insert*, *delete-min* and *meld*. We will first describe the meld operation and then indicate how other two operations can be performed in terms of the *meld* operation.

Let S_1 and S_2 be two sets and H_1 and H_2 be *Skew Heaps* storing S_1 and S_2 respectively. Recall that $S_1 \cap S_2 = \phi$. The *meld* operation should produce a single *Skew Heap* storing the values in $S_1 \cup S_2$. The procedure *meld* (H_1, H_2) consists of two phases. In the first phase, the two right most paths are merged to obtain a single right most path. This phase is pretty much like the merging algorithm working on sorted sequences. In this phase, the left subtrees of nodes in the right most paths are not disturbed. In the second phase, we simply swap the children of every node on the merged path except for the lowest. This completes the process of *melding*.

Figures 6.1, 6.2 and 6.3 clarify the phases involved in the *meld* routine.

Figure 6.1 shows two *Skew Heaps* H_1 and H_2. In Figure 6.2 we have shown the scenario after the completion of the first phase. Notice that right most paths are merged to obtain the right most path of a single tree, keeping the respective left subtrees intact. The final

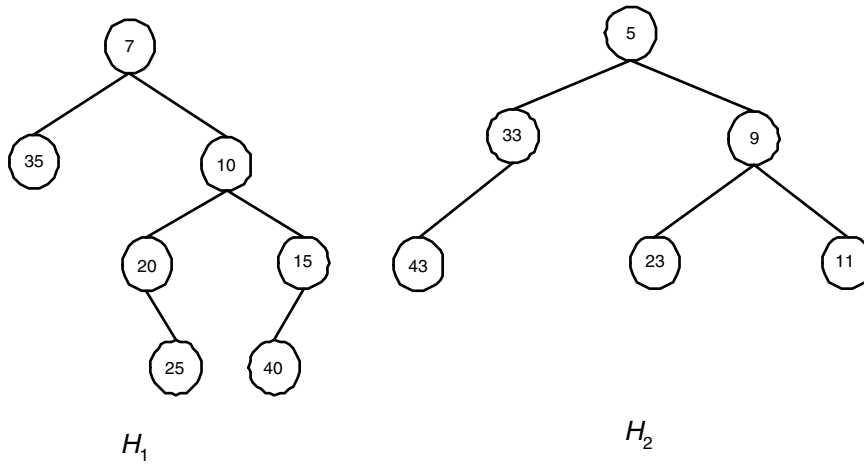


FIGURE 6.1: Skew Heaps for meld operation.

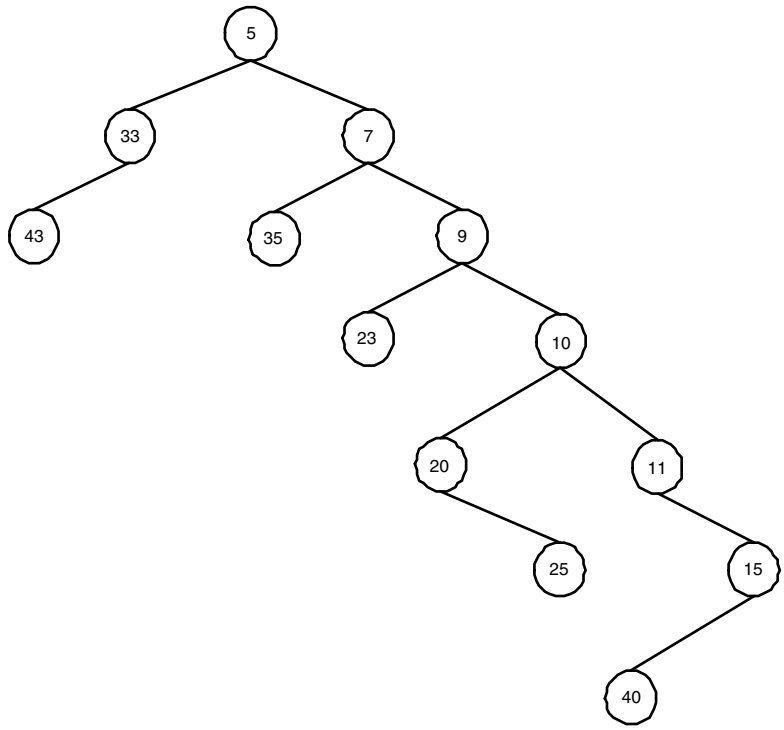


FIGURE 6.2: Rightmost paths are merged. Left subtrees of nodes in the merged path are intact.

Skew Heap is obtained in Figure 6.3. Note that left and right child of every node on the right most path of the tree in Figure 6.2 (except the lowest) are swapped to obtain the final *Skew Heap*.

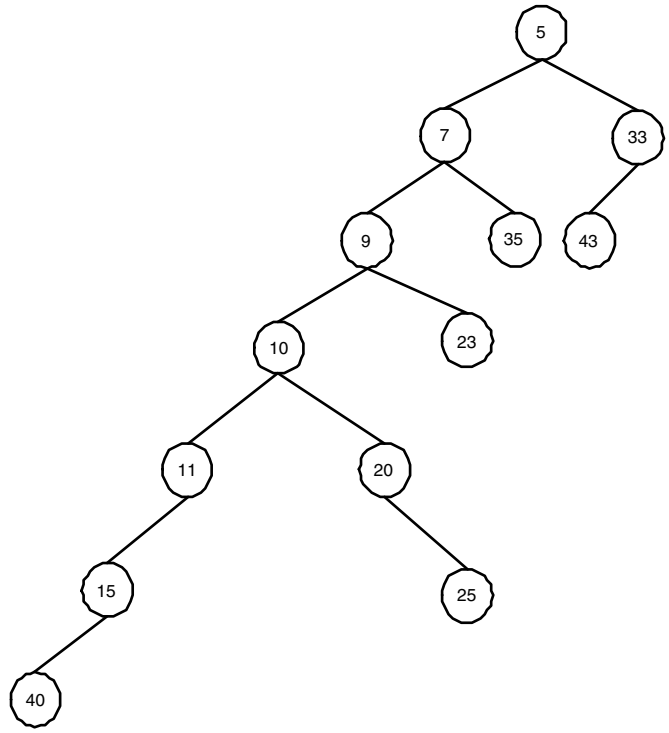


FIGURE 6.3: Left and right children of nodes (5), (7), (9), (10), (11) of Figure 2 are swapped. Notice that the children of (15) which is the lowest node in the merged path, are not swapped.

It is easy to implement *delete-min* and *insert* in terms of the *meld* operation. Since minimum is always found at the root, *delete-min* is done by simply removing the root and *melding* its left subtree and right subtree. To *insert* an item x in a *Skew Heap* H_1, we create a Skew Heap H_2 consisting of only one node containing x and then *meld* H_1 and H_2. From the above discussion, it is clear that cost of *meld* essentially determines the cost of *insert* and *delete-min*. In the next section, we analyze the amortized cost of *meld* operation.

6.3.2 Amortized Cost of Meld Operation

At this juncture we are left with the crucial task of identifying a suitable potential function. Before proceeding further, perhaps one should try the implication of certain simple potential functions and experiment with the resulting amortized cost. For example, you may try the function $f(D)$ = number of nodes in D(and discover how ineffective it is!).

We need some definitions to arrive at our potential function.

DEFINITION 6.3 For any node x in a binary tree, the weight of x, denoted $wt(x)$, is the number of descendants of x, including itself. A non-root node x is said to be heavy if $wt(x) > wt(parent(x))/2$. A non-root node that is not heavy is called light. The root is neither light nor heavy.

The next lemma is an easy consequence of the definition given above. All logarithms in this section have base 2.

LEMMA 6.1 For any node, at most one of its children is heavy. Furthermore, any root to leaf path in a n-node tree contains at most $\lfloor \log n \rfloor$ light nodes.

DEFINITION 6.4 [Potential Function] A non-root is called *right* if it is the right child of its parent; it is called *left* otherwise. The potential of a skew heap is the number of right heavy node it contains. That is, $f(H) =$ number of right heavy nodes in H. We extend the definition of potential function to a collection of skew heaps as follows: $f(H_1, H_2, \cdots, H_t) = \sum_{i=1}^{t} f(H_i)$.

Here is the key result of this chapter.

THEOREM 6.2 *Let H_1 and H_2 be two heaps with n_1 and n_2 nodes respectively. Let $n = n_1 + n_2$. The amortized cost of meld (H_1, H_2) is $O(\log n)$.*

Proof Let h_1 and h_2 denote the number of heavy nodes in the right most paths of H_1 and H_2 respectively. The number of light nodes on them will be at most $\lfloor \log n_1 \rfloor$ and $\lfloor \log n_2 \rfloor$ respectively. Since a node other than root is either heavy or light, and there are two root nodes here that are neither heavy or light, the total number of nodes in the right most paths is at most

$$2 + h_1 + h_2 + \lfloor \log n_1 \rfloor + \lfloor \log n_2 \rfloor \leq 2 + h_1 + h_2 + 2\lfloor \log n \rfloor$$

Thus we get a bound for actual cost c as

$$c \leq 2 + h_1 + h_2 + 2\lfloor \log n \rfloor \tag{6.1}$$

In the process of swapping, the $h_1 + h_2$ nodes that were *right heavy*, will lose their status as *right heavy*. While they remain heavy, they become left children for their parents hence they do not contribute for the potential of the output tree and this means a drop in potential by $h_1 + h_2$. However, the swapping might have created new heavy nodes and let us say, the number of new heavy nodes created in the swapping process is h_3. First, observe that all these h_3 new nodes are attached to the left most path of the output tree. Secondly, by Lemma 6.1, for each one of these right heavy nodes, its sibling in the left most path is a light node. However, the number of light nodes in the left most path of the output tree is less than or equal to $\lfloor \log n \rfloor$ by Lemma 6.1.

Thus $h_3 \leq \lfloor \log n \rfloor$. Consequently, the net change in the potential is $h_3 - h_1 - h_2 \leq \lfloor \log n \rfloor - h_1 - h_2$.

$$\begin{aligned} \text{The amortized cost} &= c + \text{ potential difference} \\ &\leq 2 + h_1 + h_2 + 2\lfloor \log n \rfloor + \lfloor \log n \rfloor - h_1 - h_2 \\ &= 3\lfloor \log n \rfloor + 2. \end{aligned}$$

Hence, the amortized cost of meld operation is $O(\log n)$ and this completes the proof.

Since *insert* and *delete-min* are handled as special cases of *meld* operation, we conclude

THEOREM 6.3 *The amortized cost complexity of all the Meldable Priority Queue operations is* $O(\log n)$ *where* n *is the number of nodes in skew heap or heaps involved in the operation.*

6.4 Bibliographic Remarks

Skew Heaps were introduced by Sleator and Tarjan [7]. *Leftist Trees* have $O(\log n)$ worst case complexity for all the *Meldable Priority Queue* operations but they require heights of each subtree to be maintained as additional information at each node. *Skew Heaps* are simpler than *Leftist Trees* in the sense that no additional 'balancing' information need to be maintained and the *meld* operation simply swaps the children of the right most path without any constraints and this results in a simpler code. The bound $3\log_2 n + 2$ for *melding* was significantly improved to $\log_\phi n$(here ϕ denotes the well-known *golden ratio* $(\sqrt{5}+1)/2$ which is roughly 1.6) by using a different potential function and an intricate analysis in [6]. Recently, this bound was shown to be tight in [2]. *Pairing Heap*, introduced by Fredman et al. [5], is yet another self-adjusting heap structure and its relation to *Skew Heaps* is explored in Chapter 7 of this handbook.

References

[1] A. Aravind and C. Pandu Rangan, Symmetric Min-Max heaps: A simple data structure for double-ended priority queue, *Information Processing Letters*, 69:197-199, 1999.

[2] B. Schoenmakers, A tight lower bound for top-down skew heaps, *Information Processing Letters*, 61:279-284, 1997.

[3] S. Carlson, The Deap - A double ended heap to implement a double ended priority queue, *Information Processing Letters*, 26: 33-36, 1987.

[4] S. Chang and M. Du, Diamond dequeue: A simple data structure for priority dequeues, *Information Processing Letters*, 46:231-237, 1993.

[5] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan, The pairing heap: A new form of self-adjusting heap, *Algorithmica*, 1:111-129, 1986.

[6] A. Kaldewaij and B. Schoenmakers, The derivation of a tighter bound for top-down skew heaps, *Information Processing Letters*, 37:265-271, 1991.

[7] D. D. Sleator and R. E. Tarjan, Self-adjusting heaps, *SIAM J Comput.*, 15:52-69, 1986.

7

Binomial, Fibonacci, and Pairing Heaps

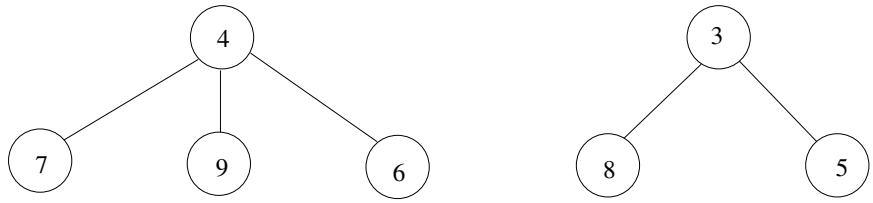
Michael L. Fredman
Rutgers University at New Brunswick

7.1 Introduction

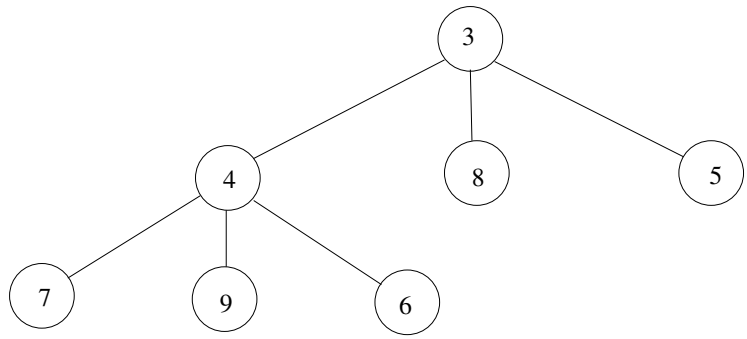
This chapter presents three algorithmically related data structures for implementing meldable priority queues: binomial heaps, Fibonacci heaps, and pairing heaps. What these three structures have in common is that (a) they are comprised of heap-ordered trees, (b) the comparisons performed to execute extractmin operations exclusively involve keys stored in the roots of trees, and (c) a common side effect of a comparison between two root keys is the linking of the respective roots: one tree becomes a new subtree joined to the other root.

A tree is considered heap-ordered provided that each node contains one item, and the key of the item stored in the parent $p(x)$ of a node x never exceeds the key of the item stored in x. Thus, when two roots get linked, the root storing the larger key becomes a child of the other root. By convention, a linking operation positions the new child of a node as its leftmost child. Figure 7.1 illustrates these notions.

Of the three data structures, the binomial heap structure was the first to be invented (Vuillemin [13]), designed to efficiently support the operations insert, extractmin, delete, and meld. The binomial heap has been highly appreciated as an elegant and conceptually simple data structure, particularly given its ability to support the meld operation. The Fibonacci heap data structure (Fredman and Tarjan [6]) was inspired by and can be viewed as a generalization of the binomial heap structure. The raison d'être of the Fibonacci heap structure is its ability to efficiently execute decrease-key operations. A decrease-key operation replaces the key of an item, specified by location, by a smaller value: e.g. decrease-key(P,k_{new},H). (The arguments specify that the item is located in node P of the priority queue H, and that its new key value is k_{new}.) Decrease-key operations are prevalent in many network optimization algorithms, including minimum spanning tree, and shortest path. The pairing heap data structure (Fredman, Sedgewick, Sleator, and Tarjan [5]) was



(a) before linking.



(b) after linking.

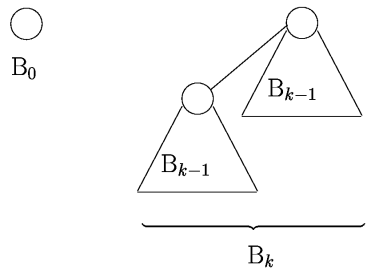
FIGURE 7.1: Two heap-ordered trees and the result of their linking.

devised as a self-adjusting analogue of the Fibonacci heap, and has proved to be more efficient in practice [11].

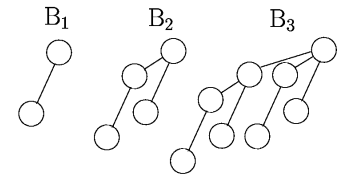
Binomial heaps and Fibonacci heaps are primarily of theoretical and historical interest. The pairing heap is the more efficient and versatile data structure from a practical standpoint. The following three sections describe the respective data structures. Summaries of the various algorithms in the form of pseudocode are provided in section 7.5.

7.2 Binomial Heaps

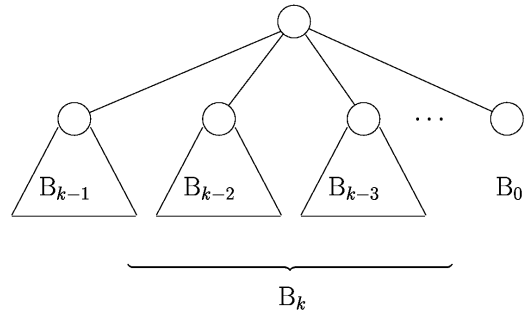
We begin with an informal overview. A single binomial heap structure consists of a forest of specially structured trees, referred to as binomial trees. The number of nodes in a binomial tree is always a power of two. Defined recursively, the binomial tree B_0 consists of a single node. The binomial tree B_k, for $k > 0$, is obtained by linking two trees B_{k-1} together; one tree becomes the leftmost subtree of the other. In general B_k has 2^k nodes. Figures 7.2(a-b) illustrate the recursion and show several trees in the series. An alternative and useful way to view the structure of B_k is depicted in Figure 7.2(c): B_k consists of a root and subtrees (in order from left to right) $B_{k-1}, B_{k-2}, \cdots, B_0$. The root of the binomial tree B_k has k children, and the tree is said to have rank k. We also observe that the height of B_k (maximum number of edges on any path directed away from the root) is k. The name "binomial heap" is inspired by the fact that the root of B_k has $\binom{k}{j}$ descendants at distance j.



(a) Recursion for binomial trees.



(b) Several binomial trees.



(c) An alternative recursion.

FIGURE 7.2: Binomial trees and their recursions.

Because binomial trees have restricted sizes, a forest of trees is required to represent a priority queue of arbitrary size. A key observation, indeed a motivation for having tree sizes being powers of two, is that a priority queue of arbitrary size can be represented as a union of trees of *distinct* sizes. (In fact, the sizes of the constituent trees are uniquely determined and correspond to the powers of two that define the binary expansion of n, the size of the priority queue.) Moreover, because the tree sizes are unique, the number of trees in the forest of a priority queue of size n is at most $\lg(n+1)$. Thus, finding the minimum key in the priority queue, which clearly lies in the root of one of its constituent trees (due to the heap-order condition), requires searching among at most $\lg(n+1)$ tree roots. Figure 7.3 gives an example of binomial heap.

Now let's consider, from a high-level perspective, how the various heap operations are performed. As with leftist heaps (cf. Chapter 6), the various priority queue operations are to a large extent comprised of melding operations, and so we consider first the melding of two heaps.

The melding of two heaps proceeds as follows: (a) the trees of the respective forests are combined into a single forest, and then (b) *consolidation* takes place: pairs of trees having common rank are linked together until all remaining trees have distinct ranks. Figure 7.4 illustrates the process. An actual implementation mimics binary addition and proceeds in much the same was as merging two sorted lists in ascending order. We note that insertion is a special case of melding.

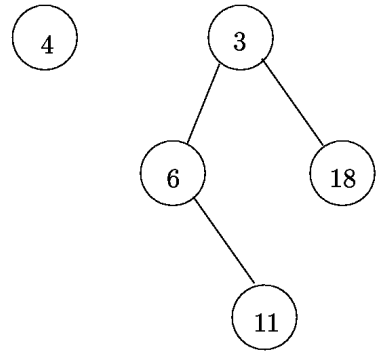
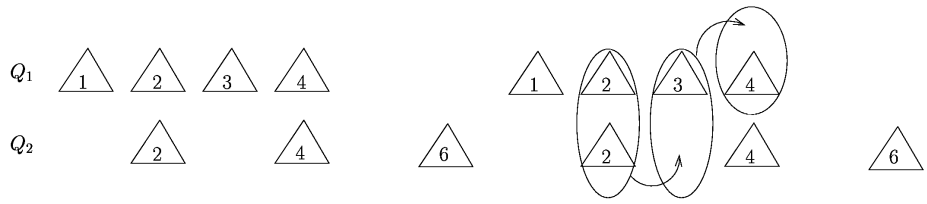


FIGURE 7.3: A binomial heap (showing placement of keys among forest nodes).



(a) Forests of two heaps Q_1 and Q_2 to be melded.

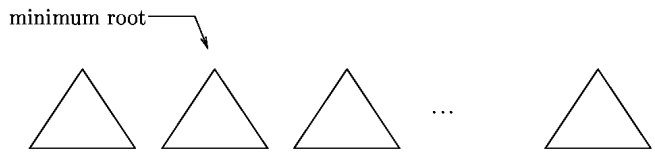
(b) Linkings among trees in the combined forest.

(c) Forest of meld(Q_1,Q_2).

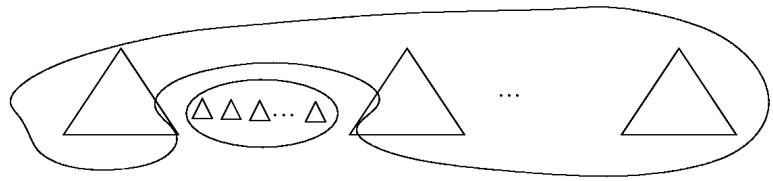
FIGURE 7.4: Melding of two binomial heaps. The encircled objects reflect trees of common rank being linked. (Ranks are shown as numerals positioned within triangles which in turn represent individual trees.) Once linking takes place, the resulting tree becomes eligible for participation in further linkings, as indicated by the arrows that identify these linking results with participants of other linkings.

The extractmin operation is performed in two stages. First, the *minimum root*, the node containing the minimum key in the data structure, is found by examining the tree roots of the appropriate forest, and this node is removed. Next, the forest consisting of the subtrees of this removed root, whose ranks are distinct (see Figure 7.2(c)) and thus viewable as

constituting a binomial heap, is melded with the forest consisting of the trees that remain from the original forest. Figure 7.5 illustrates the process.



(a) Initial forest.

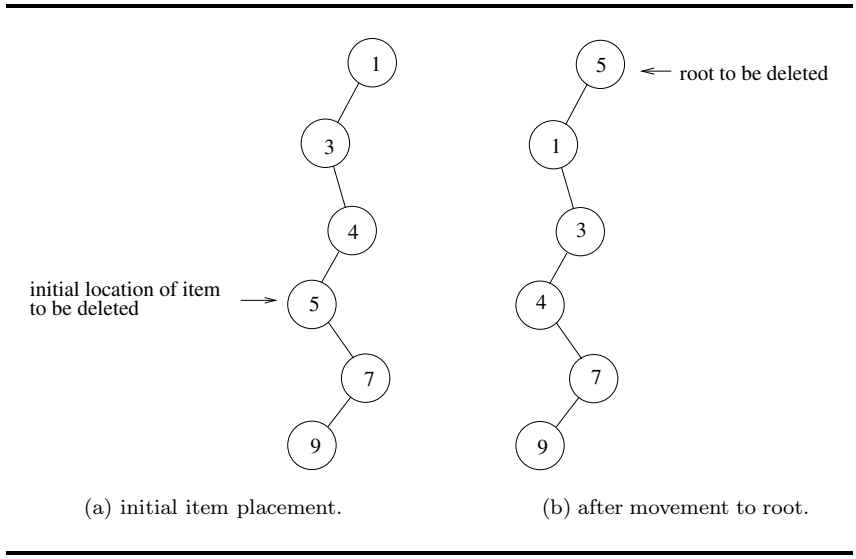


(b) Forests to be melded.

FIGURE 7.5: Extractmin Operation: The location of the minimum key is indicated in (a). The two encircled sets of trees shown in (b) represent forests to be melded. The smaller trees were initially subtrees of the root of the tree referenced in (a).

Finally, we consider arbitrary deletion. We assume that the node ν containing the item to be deleted is specified. Proceeding up the path to the root of the tree containing ν, we permute the items among the nodes on this path, placing in the root the item x originally in ν, and shifting each of the other items down one position (away from the root) along the path. This is accomplished through a sequence of exchange operations that move x towards the root. The process is referred to as a *sift-up* operation. Upon reaching the root r, r is then removed from the forest as though an extractmin operation is underway. Observe that the re-positioning of items in the ancestors of ν serves to maintain the heap-order property among the remaining nodes of the forest. Figure 7.6 illustrates the re-positioning of the item being deleted to the root.

This completes our high-level descriptions of the heap operations. For navigational purposes, each node contains a leftmost child pointer and a sibling pointer that points to the next sibling to its right. The children of a node are thus stored in the linked list defined by sibling pointers among these children, and the head of this list can be accessed by the leftmost child pointer of the parent. This provides the required access to the children of



(a) initial item placement. (b) after movement to root.

FIGURE 7.6: Initial phase of deletion – sift-up operation.

a node for the purpose of implementing extractmin operations. Note that when a node obtains a new child as a consequence of a linking operation, the new child is positioned at the head of its list of siblings. To facilitate arbitrary deletions, we need a third pointer in each node pointing to its parent. To facilitate access to the ranks of trees, we maintain in each node the number of children it has, and refer to this quantity as the node rank. Node ranks are readily maintained under linking operations; the node rank of the root gaining a child gets incremented. Figure 7.7 depicts these structural features.

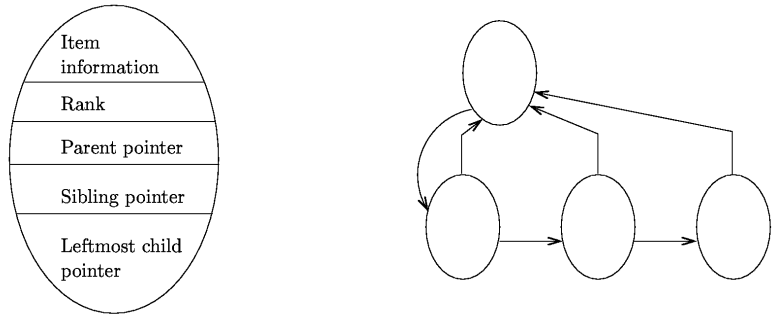
As seen in Figure 7.2(c), the ranks of the children of a node form a descending sequence in the children's linked list. However, since the melding operation is implemented by accessing the tree roots in ascending rank order, when deleting a root we first reverse the list order of its children before proceeding with the melding.

Each of the priority queue operations requires in the worst case $O(\log n)$ time, where n is the size of the heap that results from the operation. This follows, for melding, from the fact that its execution time is proportional to the combined lengths of the forest lists being merged. For extractmin, this follows from the time for melding, along with the fact that a root node has only $O(\log n)$ children. For arbitrary deletion, the time required for the sift-up operation is bounded by an amount proportional to the height of the tree containing the item. Including the time required for extractmin, it follows that the time required for arbitrary deletion is $O(\log n)$.

Detailed code for manipulating binomial heaps can be found in Weiss [14].

7.3 Fibonacci Heaps

Fibonacci heaps were specifically designed to efficiently support decrease-key operations. For this purpose, the binomial heap can be regarded as a natural starting point. Why? Consider the class of priority queue data structures that are implemented as forests of heap-ordered trees, as will be the case for Fibonacci heaps. One way to immediately execute a



(a) fields of a node.

(b) a node and its three children.

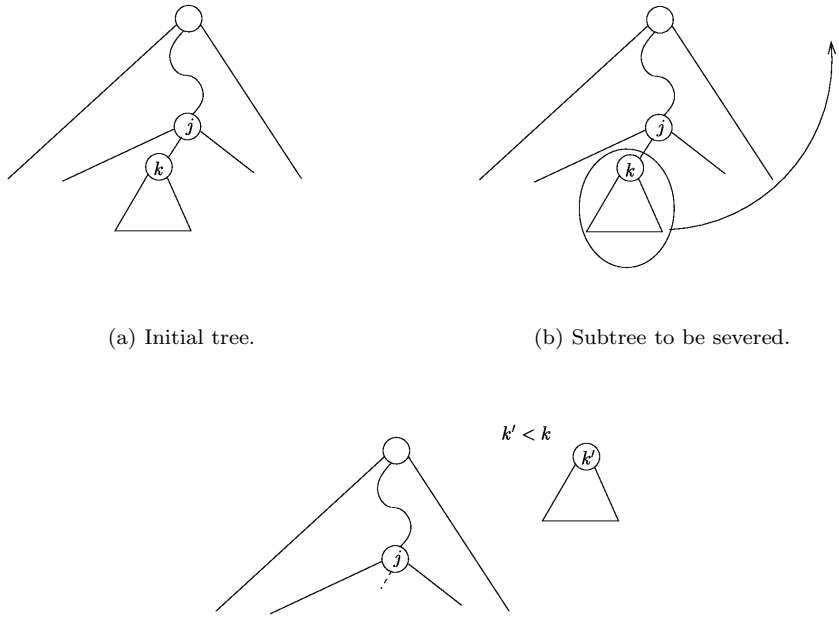
FIGURE 7.7: Structure associated with a binomial heap node. Figure (b) illustrates the positioning of pointers among a node and its three children.

decrease-key operation, remaining within the framework of heap-ordered forests, is to simply change the key of the specified data item and sever its link to its parent, inserting the severed subtree as a new tree in the forest. Figure 7.8 illustrates the process. (Observe that the link to the parent only needs to be cut if the new key value is smaller than the key in the parent node, violating heap-order.) Fibonacci heaps accomplish this without degrading the asymptotic efficiency with which other priority queue operations can be supported. Observe that to accommodate node cuts, the list of children of a node needs to be doubly linked. Hence the nodes of a Fibonacci heap require two sibling pointers.

Fibonacci heaps support findmin, insertion, meld, and decrease-key operations in constant amortized time, and deletion operations in $O(\log n)$ amortized time. For many applications, the distinction between worst-case times versus amortized times are of little significance. A Fibonacci heap consists of a forest of heap-ordered trees. As we shall see, Fibonacci heaps differ from binomial heaps in that there may be many trees in a forest of the same rank, and there is no constraint on the ordering of the trees in the forest list. The heap also includes a pointer to the tree root containing the minimum item, referred to as the *min-pointer*, that facilitates findmin operations. Figure 7.9 provides an example of a Fibonacci heap and illustrates certain structural aspects.

The impact of severing subtrees is clearly incompatible with the pristine structure of the binomial tree that is the hallmark of the binomial heap. Nevertheless, the tree structures that can appear in the Fibonacci heap data structure must sufficiently approximate binomial trees in order to satisfy the performance bounds we seek. The linking constraint imposed by binomial heaps, that trees being linked must have the same size, ensures that the number of children a node has (its rank), grows no faster than the logarithm of the size of the subtree rooted at the node. This *rank versus subtree size* relation is key to obtaining the $O(\log n)$ deletion time bound. Fibonacci heap manipulations are designed with this in mind.

Fibonacci heaps utilize a protocol referred to as *cascading cuts* to enforce the required rank versus subtree size relation. Once a node ν has had two of its children removed as a result of cuts, ν's contribution to the rank of its parent is then considered suspect in terms of rank versus subtree size. The cascading cut protocol requires that the link to ν's parent



(a) Initial tree.

(b) Subtree to be severed.

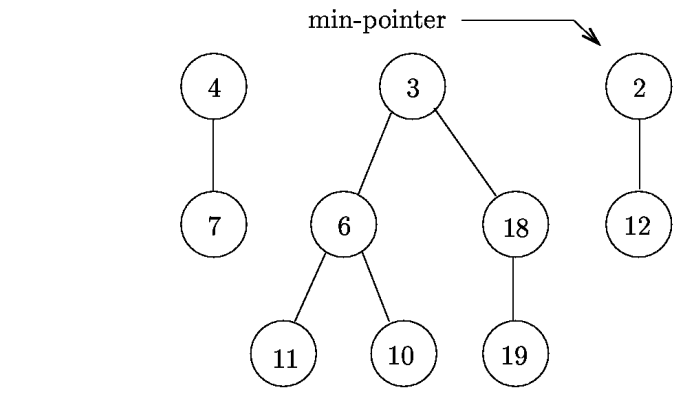
(c) Resulting changes

FIGURE 7.8: Immediate decrease-key operation. The subtree severing (Figures (b) and (c)) is necessary only when $k' < j$.

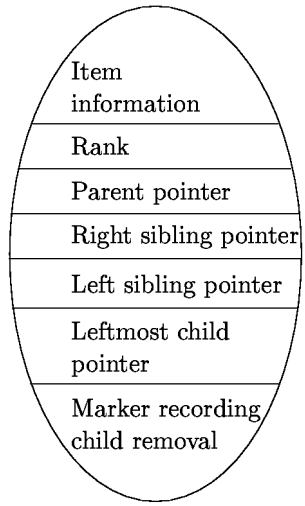
be cut, with the subtree rooted at ν then being inserted into the forest as a new tree. If ν's parent has, as a result, had a second child removed, then it in turn needs to be cut, and the cuts may thus cascade. Cascading cuts ensure that no non-root node has had more than one child removed subsequent to being linked to its parent.

We keep track of the removal of children by marking a node if one of its children has been cut. A marked node that has another child removed is then subject to being cut from its parent. When a marked node becomes linked as a child to another node, or when it gets cut from its parent, it gets unmarked. Figure 7.10 illustrates the protocol of cascading cuts.

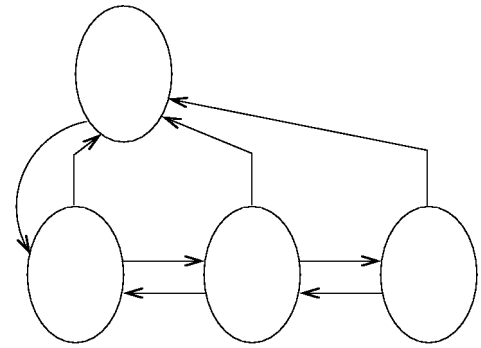
Now the induced node cuts under the cascading cuts protocol, in contrast with those primary cuts immediately triggered by decrease-key operations, are bounded in number by the number of primary cuts. (This follows from consideration of a potential function defined to be the total number of marked nodes.) Therefore, the burden imposed by cascading cuts can be viewed as effectively only doubling the number of cuts taking place in the absence of the protocol. One can therefore expect that the performance asymptotics are not degraded as a consequence of proliferating cuts. As with binomial heaps, two trees in a Fibonacci heap can only be linked if they have equal rank. With the cascading cuts protocol in place,



(a) a heap



(b) fields of a node.

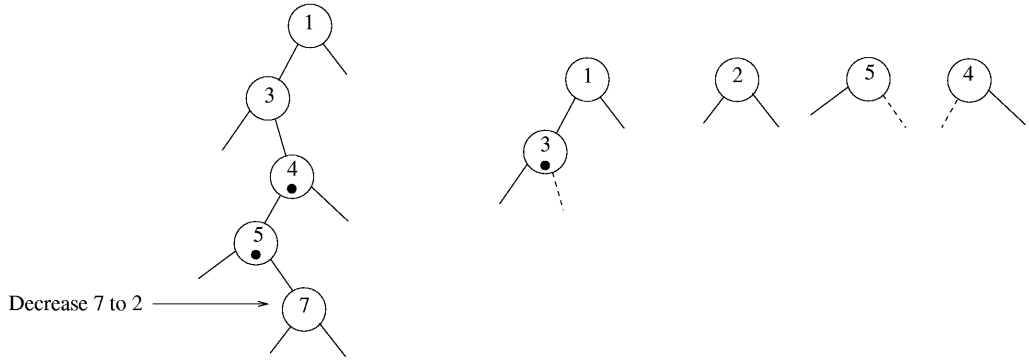


(c) pointers among a node and its three children.

FIGURE 7.9: A Fibonacci heap and associated structure.

we claim that the required rank versus subtree size relation holds, a matter which we address next.

Let's consider how *small* the subtree rooted at a node ν having rank k can be. Let ω be the mth child of ν from the right. At the time it was linked to ν, ν had at least $m-1$ other children (those currently to the right of ω were certainly present). Therefore ω had rank at least $m-1$ when it was linked to ν. Under the cascading cuts protocol, the rank of ω could have decreased by at most one after its linking to ν; otherwise it would have been removed as a child. Therefore, the current rank of ω is at least $m-2$. We minimize the size of the subtree rooted at ν by minimizing the sizes (and ranks) of the subtrees rooted at



(a) Before decrease-key. (b) After decrease-key.

FIGURE 7.10: Illustration of cascading cuts. In (b) the dashed lines reflect cuts that have taken place, two nodes marked in (a) get unmarked, and a third node gets marked.

ν's children. Now let F_j denote the minimum possible size of the subtree rooted at a node of rank j, so that the size of the subtree rooted at ν is F_k. We conclude that (for $k \geq 2$)

$$F_k = \underbrace{F_{k-2} + F_{k-3} + \cdots + F_0 + 1}_{k \text{ terms}} + 1$$

where the final term, 1, reflects the contribution of ν to the subtree size. Clearly, $F_0 = 1$ and $F_1 = 2$. See Figure 7.11 for an illustration of this construction. Based on the preceding recurrence, it is readily shown that F_k is given by the $(k+2)$th Fibonacci number (from whence the name "Fibonacci heap" was inspired). Moreover, since the Fibonacci numbers grow exponentially fast, we conclude that the rank of a node is indeed bounded by the logarithm of the size of the subtree rooted at the node.

We proceed next to describe how the various operations are performed.

Since we are not seeking worst-case bounds, there are economies to be exploited that could also be applied to obtain a variant of Binomial heaps. (In the absence of cuts, the individual trees generated by Fibonacci heap manipulations would all be binomial trees.) In particular we shall adopt a lazy approach to melding operations: the respective forests are simply combined by concatenating their tree lists and retaining the appropriate min-pointer. This requires only constant time.

An item is deleted from a Fibonacci heap by deleting the node that originally contains it, in contrast with Binomial heaps. This is accomplished by (a) cutting the link to the node's parent (as in decrease-key) if the node is not a tree root, and (b) appending the list of children of the node to the forest. Now if the deleted node happens to be referenced by the min-pointer, considerable work is required to restore the min-pointer – the work previously deferred by the lazy approach to the operations. In the course of searching among the roots

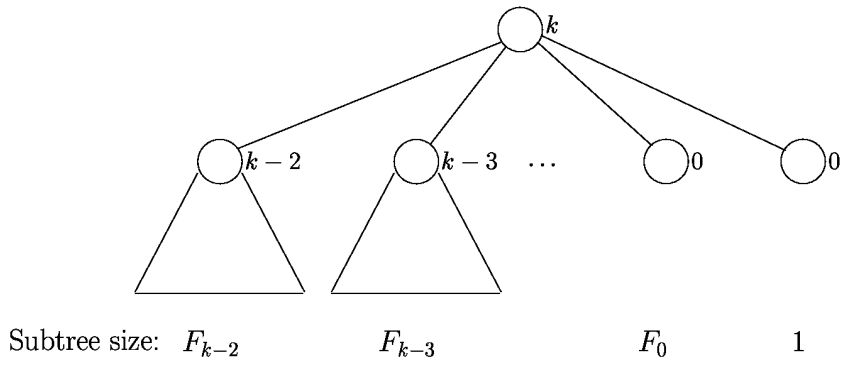


FIGURE 7.11: Minimal tree of rank k. Node ranks are shown adjacent to each node.

of the forest to discover the new minimum key, we also link trees together in a *consolidation* process.

Consolidation processes the trees in the forest, linking them in pairs until there are no longer two trees having the same rank, and then places the remaining trees in a new forest list (naturally extending the melding process employed by binomial heaps). This can be accomplished in time proportional to the number of trees in forest plus the maximum possible node rank. Let max-rank denote the maximum possible node rank. (The preceding discussion implies that max-rank = $O(\log$ heap-size$)$.) Consolidation is initialized by setting up an array A of trees (initially empty) indexed by the range [0,max-rank]. A non-empty position A[d] of A contains a tree of rank d. The trees of the forest are then processed using the array A as follows. To process a tree T of rank d, we insert T into A[d] if this position of A is empty, completing the processing of T. However, if A[d] already contains a tree U, then T and U are linked together to form a tree W, and the processing continues as before, but with W in place of T, until eventually an empty location of A is accessed, completing the processing associated with T. After all of the trees have been processed in this manner, the array A is scanned, placing each of its stored trees in a new forest. Apart from the final scanning step, the total time necessary to consolidate a forest is proportional to its number of trees, since the total number of tree pairings that can take place is bounded by this number (each pairing reduces by one the total number of trees present). The time required for the final scanning step is given by max-rank = log(heap-size).

The amortized timing analysis of Fibonacci heaps considers a potential function defined as the total number of trees in the forests of the various heaps being maintained. Ignoring consolidation, each operation takes constant actual time, apart from an amount of time proportional to the number of subtree cuts due to cascading (which, as noted above, is only constant in amortized terms). These cuts also contribute to the potential. The children of a deleted node increase the potential by $O(\log$ heap-size$)$. Deletion of a minimum heap node additionally incurs the cost of consolidation. However, consolidation reduces our potential, so that the amortized time it requires is only $O(\log$ heap-size$)$. We conclude therefore that all non-deletion operations require constant amortized time, and deletion requires $O(\log n)$ amortized time.

An interesting and unresolved issue concerns the protocol of cascading cuts. How would the performance of Fibonacci heaps be affected by the absence of this protocol?

Detailed code for manipulating Fibonacci heaps can found in Knuth [9].

7.4 Pairing Heaps

The pairing heap was designed to be a self-adjusting analogue of the Fibonacci heap, in much the same way that the skew heap is a self-adjusting analogue of the leftist heap (See Chapters 5 and 6). The only structure maintained in a pairing heap node, besides item information, consists of three pointers: leftmost child, and two sibling pointers. (The leftmost child of a node uses it left sibling pointer to point to its parent, to facilitate updating the leftmost child pointer its parent.) See Figure 7.12 for an illustration of pointer structure.

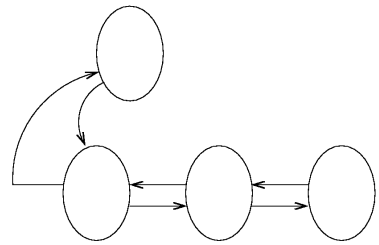


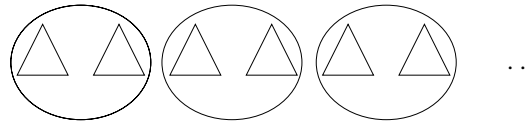
FIGURE 7.12: Pointers among a pairing heap node and its three children.

The are no cascading cuts – only simple cuts for decrease-key and deletion operations. With the absence of parent pointers, decrease-key operations uniformly require a single cut (removal from the sibling list, in actuality), as there is no efficient way to check whether heap-order would otherwise be violated. Although there are several varieties of pairing heaps, our discussion presents the two-pass version (the simplest), for which a given heap consists of only a single tree. The minimum element is thus uniquely located, and melding requires only a single linking operation. Similarly, a decrease-key operation consists of a subtree cut followed by a linking operation. Extractmin is implemented by removing the tree root and then linking the root's subtrees in a manner described below. Other deletions involve (a) a subtree cut, (b) an extractmin operation on the cut subtree, and (c) linking the remnant of the cut subtree with the original root.

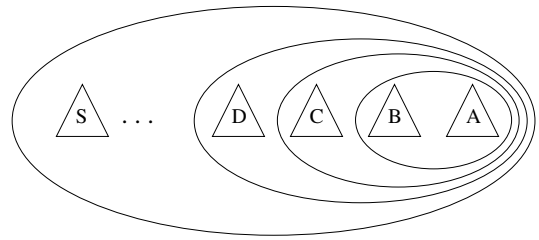
The extractmin operation combines the subtrees of the root using a process referred to as *two-pass pairing*. Let $x_1, \cdots, x_k$ be the subtrees of the root in left-to-right order. The first pass begins by linking x_1 and x_2. Then x_3 and x_4 are linked, followed by x_5 and x_6, etc., so that the odd positioned trees are linked with neighboring even positioned trees. Let $y_1, \cdots, y_h$, $h = \lceil k/2 \rceil$, be the resulting trees, respecting left-to-right order. (If k is odd, then $y_{\lceil k/2 \rceil}$ is x_k.) The second pass reduces these to a single tree with linkings that proceed from right-to-left. The rightmost pair of trees, y_h and y_{h-1} are linked first, followed by the linking of y_{h-2} with the result of the preceding linking etc., until finally we link y_1 with the structure formed from the linkings of $y_2, \cdots, y_h$. See Figure 7.13.

Since two-pass pairing is not particularly intuitive, a few motivating remarks are offered. The first pass is natural enough, and one might consider simply repeating the process on the remaining trees, until, after logarithmically many such passes, only one tree remains. Indeed, this is known as the multi-pass variation. Unfortunately, its behavior is less understood than that of the two-pass pairing variation.

The second (right-to-left) pass is also quite natural. Let H be a binomial heap with exactly 2^k items, so that it consists of a single tree. Now suppose that an extractmin followed by



(a) first pass.



(b) second pass.

FIGURE 7.13: Two-pass Pairing. The encircled trees get linked. For example, in (b) trees A and B get linked, and the result then gets linked with the tree C, etc.

an insertion operation are executed. The linkings that take place among the subtrees of the deleted root (after the new node is linked with the rightmost of these subtrees) entail the right-to-left processing that characterizes the second pass. So why not simply rely upon a single right-to-left pass, and omit the first? The reason, is that although the second pass preserves existing balance within the structure, it doesn't improve upon poorly balanced situations (manifested when most linkings take place between trees of disparate sizes). For example, using a single-right-to-left-pass version of a pairing heap to sort an increasing sequence of length n (n insertions followed by n extractmin operations), would result in an n^2 sorting algorithm. (Each of the extractmin operations yields a tree of height 1 or less.) See Section 7.6, however, for an interesting twist.

In actuality two-pass pairing was inspired [5] by consideration of splay trees (Chapter 12). If we consider the child, sibling representation that maps a forest of arbitrary trees into a binary tree, then two-pass pairing can be viewed as a splay operation on a search tree path with no bends [5]. The analysis for splay trees then carries over to provide an amortized analysis for pairing heaps.

The asymptotic behavior of pairing heaps is an interesting and unresolved matter. Reflecting upon the tree structures we have encountered in this chapter, if we view the binomial trees that comprise binomial heaps, their structure highly constrained, as likened to perfectly spherical masses of discrete diameter, then the trees that comprise Fibonacci heaps can be viewed as rather rounded masses, but rarely spherical, and of arbitrary (non-discrete) size. Applying this imagery to the trees that arise from pairing heap manipulations, we can aptly liken these trees to chunks of clay subject to repeated tearing and compaction, typically irregular in form. It is not obvious, therefore, that pairing heaps should be asymptotically efficient. On the other hand, since the pairing heap design dispenses with the rather complicated, carefully crafted constructs put in place primarily to facilitate proving the time bounds enjoyed by Fibonacci heaps, we can expect efficiency gains at the level of elemen-

tary steps such as linking operations. From a practical standpoint the data structure is a success, as seen from the study of Moret and Shapiro [11]. Also, for those applications for which decrease-key operations are highly predominant, pairing heaps provably meet the optimal asymptotic bounds characteristic of Fibonacci heaps [3]. But despite this, as well as empirical evidence consistent with optimal efficiency in general, pairing heaps are in fact asymptotically sub-optimal for certain operation sequences [3]. Although decrease-key requires only constant worst-case time, its execution can asymptotically degrade the efficiency of extractmin operations, even though the effect is not observable in practice. On the positive side, it has been demonstrated [5] that under all circumstances the operations require only $O(\log n)$ amortized time. Additionally, Iacono [7] has shown that insertions require only constant amortized time; significant for those applications that entail many more insertions than deletions.

The reader may wonder whether some alternative to two-pass pairing might provably attain the asymptotic performance bounds satisfied by Fibonacci heaps. However, for information-theoretic reasons *no* such alternative exists. (In fact, this is how we know the two-pass version is sub-optimal.) A precise statement and proof of this result appears in Fredman [3].

Detailed code for manipulating pairing heaps can be found in Weiss [14].

7.5 Pseudocode Summaries of the Algorithms

This section provides pseudocode reflecting the above algorithm descriptions. The procedures, link and insert, are sufficiently common with respect to all three data structures, that we present them first, and then turn to those procedures having implementations specific to a particular data structure.

7.5.1 Link and Insertion Algorithms

```
Function link(x,y){
   // x and y are tree roots. The operation makes the root with the
   // larger key the leftmost child of the other root. For binomial and
   // Fibonacci heaps, the rank field of the prevailing root is
   // incremented.  Also, for Fibonacci heaps, the node becoming the child
   // gets unmarked if it happens to be originally marked.  The function
   // returns a pointer to the node x or y that becomes the root.
}

Algorithm Insert(x,H){
   //Inserts into heap H the item x
   I = Makeheap(x)
      // Creates a single item heap I containing the item x.
   H = Meld(H,I).
}
```

7.5.2 Binomial Heap-Specific Algorithms

```
Function Meld(H,I){
   // The forest lists of H and I are combined and consolidated -- trees
   // having common rank are linked together until only trees of distinct
   // ranks remain.  (As described above, the process resembles binary
   // addition.)  A pointer to the resulting list is returned.  The
   // original lists are no longer available.
}

Function Extractmin(H){
   //Returns the item containing the minimum key in the heap H.
   //The root node r containing this item is removed from H.
   r = find-minimum-root(H)
   if(r = null){return "Empty"}
   else{
      x = item in r
      H = remove(H,r)
         // removes the tree rooted at r from the forest of H
      I = reverse(list of children of r)
      H = Meld(H,I)
      return x
   }
}

Algorithm Delete(x,H)
   //Removes from heap H the item in the node referenced by x.
   r = sift-up(x)
      // r is the root of the tree containing x.  As described above,
      // sift-up moves the item information in x to r.
   H = remove(H,r)
      // removes the tree rooted at r from the forest of H
   I = reverse(list of children of r)
   H = Meld(H,I)
}
```

7.5.3 Fibonacci Heap-Specific Algorithms

```
Function Findmin(H){
   //Return the item in the node referenced by the min-pointer of H
   //(or "Empty" if applicable)
}

Function Meld(H,I){
   // The forest lists of H and I are concatenated.  The keys referenced
   // by the respective min-pointers of H and I are compared, and the
   // min-pointer referencing the larger key is discarded.  The concatenation
   // result and the retained min-pointer are returned.  The original
   // forest lists of H and I are no longer available.
}
```

```
Algorithm Cascade-Cut(x,H){
   //Used in decrease-key and deletion.  Assumes parent(x) != null
   y = parent(x)
   cut(x,H)
      // The subtree rooted at x is removed from parent(x) and inserted into
      // the forest list of H.  The mark-field of x is set to FALSE, and the
      // rank of parent(x) is decremented.
   x = y
   while(x is marked and parent(x) != null){
       y = parent(x)
       cut(x,H)
       x = y
   }
   Set mark-field of x = TRUE
}

Algorithm Decrease-key(x,k,H){
   key(x) = k
   if(key of min-pointer(H) > k){ min-pointer(H) = x}
   if(parent(x) != null and key(parent(x)) > k){ Cascade-Cut(x,H)}
}

Algorithm Delete(x,H){
   If(parent(x) != null){
      Cascade-Cut(x,H)
      forest-list(H) = concatenate(forest-list(H), leftmost-child(x))
      H = remove(H,x)
         // removes the (single node) tree rooted at x from the forest of H
   }
   else{
      forest-list(H) = concatenate(forest-list(H), leftmost-child(x))
      H = remove(H,x)
      if(min-pointer(H) = x){
         consolidate(H)
            // trees of common rank in the forest list of H are linked
            // together until only trees having distinct ranks remain.  The
            // remaining trees then constitute the forest list of H.
            // min-pointer is reset to reference the root with minimum key.
       }
   }
}
```

7.5.4 Pairing Heap-Specific Algorithms

```
Function Findmin(H){
   // Returns the item in the node referenced by H (or "empty" if applicable)
}

Function Meld(H,I){
   return link(H,I)
}
```

```
Function Decrease-key(x,k,H){
   If(x != H){
      Cut(x)
         // The node x is removed from the child list in which it appears
      key(x) = k
      H = link(H,x)
   }
   else{ key(H) = k}
}

Function Two-Pass-Pairing(x){
   // x is assumed to be a pointer to the first node of a list of tree
   // roots.  The function executes two-pass pairing to combine the trees
   // into a single tree as described above, and returns a pointer to
   // the root of the resulting tree.
}

Algorithm Delete(x,H){
   y = Two-Pass-Pairing(leftmost-child(x))
   if(x = H){ H = y}
   else{
      Cut(x)
         // The subtree rooted at x is removed from its list of siblings.
      H = link(H,y)
   }
}
```

7.6 Related Developments

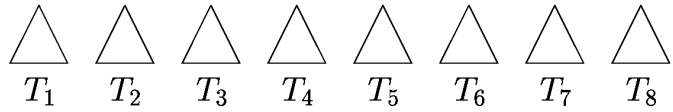
In this section we describe some results pertinent to the data structures of this chapter. First, we discuss a variation of the pairing heap, referred to as the skew-pairing heap. The skew-pairing heap appears as a form of "missing link" in the landscape occupied by pairing heaps and skew heaps (Chapter 6). Second, we discuss some adaptive properties of pairing heaps. Finally, we take note of soft heaps, a new shoot of activity emanating from the primordial binomial heap structure that has given rise to the topics of this chapter.

Skew-Pairing Heaps

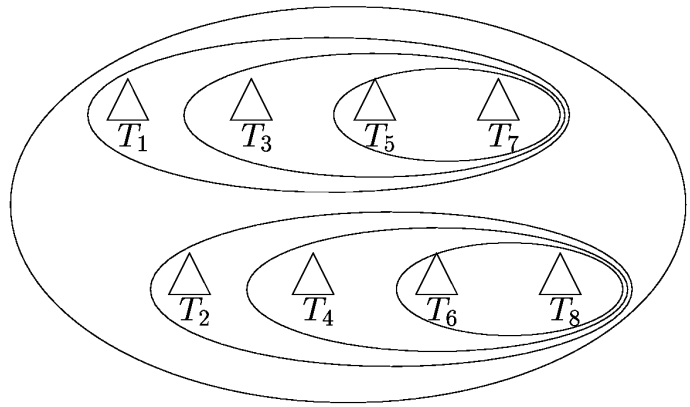
There is a curious variation of the pairing heap which we refer to as a *skew-pairing* heap – the name will become clear. Aside from the linking process used for combining subtrees in the extractmin operation, skew-pairing heaps are identical to two-pass pairing heaps. The skew-pairing heap extractmin linking process places greater emphasis on right-to-left linking than does the pairing heap, and proceeds as follows.

First, a right-to-left linking of the subtrees that fall in odd numbered positions is executed. Let H_{odd} denote the result. Similarly, the subtrees in even numbered positions are linked in right-to-left order. Let H_{even} denote the result. Finally, we link the two trees, H_{odd} and H_{even}. Figure 7.14 illustrates the process.

The skew-pairing heap enjoys $O(\log n)$ time bounds for the usual operations. Moreover, it has the following curious relationship to the skew heap. Suppose a finite sequence S of



(a) subtrees before linking.



(b) linkings.

FIGURE 7.14: Skew-pairing heap: linking of subtrees performed by extractmin. As described in Figure 7.13, encircled trees become linked.

meld and extractmin operations is executed (beginning with heaps of size 1) using (a) a skew heap and (b) a skew-pairing heap. Let C_s and C_{s-p} be the respective sets of comparisons between keys that actually get performed in the course of the respective executions (ignoring the order of the comparison executions). Then $C_{s-p} \subset C_s$ [4]. Moreover, if the sequence S terminates with the heap empty, then $C_{s-p} = C_s$. (This inspires the name "skew-pairing".) The relationship between skew-pairing heaps and splay trees is also interesting. The child, sibling transformation, which for two-pass pairing heaps transforms the extractmin operation into a splay operation on a search tree path having no bends, when applied to the skew-pairing heap, transforms extractmin into a splay operation on a search tree path having a bend at each node. Thus, skew-pairing heaps and two-pass pairing heaps demarcate opposite ends of a spectrum.

Adaptive Properties of Pairing Heaps

Consider the problem of merging k sorted lists of respective lengths $n_1, n_2, \cdots, n_k$, with $\sum n_i = n$. The standard merging strategy that performs $\lg k$ rounds of pairwise list merges requires $n \lg k$ time. However, a merge pattern based upon the binary Huffman tree, having minimal external path length for the weights $n_1, n_2, \cdots, n_k$, is more efficient when the lengths n_i are non-uniform, and provides a near optimal solution. Pairing heaps can be utilized to provide a rather different solution as follows. Treat each sorted list as a

linearly structured pairing heap. Then (a) meld these k heaps together, and (b) repeatedly execute extractmin operations to retrieve the n items in their sorted order. The number of comparisons that take place is bounded by

$$O(\log \binom{n}{n_1, \cdots, n_k})$$

Since the above multinomial coefficient represents the number of possible merge patterns, the information-theoretic bound implies that this result is optimal to within a constant factor. The pairing heap thus self-organizes the sorted list arrangement to approximate an optimal merge pattern. Iacono has derived a "working-set" theorem that quantifies a similar adaptive property satisfied by pairing heaps. Given a sequence of insertion and extractmin operations initiated with an empty heap, at the time a given item x is deleted we can attribute to x a contribution bounded by $O(\log \mathrm{op}(x))$ to the total running time of the sequence, where $\mathrm{op}(x)$ is the number of heap operations that have taken place since x was inserted (see [8] for a slightly tighter estimate). Iacono has also shown that this same bound applies for skew and skew-pairing heaps [8]. Knuth [10] has observed, at least in qualitative terms, similar behavior for leftist heaps . Quoting Knuth:

> Leftist trees are in fact already obsolete, except for applications with a strong tendency towards last-in-first-out behavior.

Soft Heaps

An interesting development (Chazelle [1]) that builds upon and extends binomial heaps in a different direction is a data structure referred to as a *soft heap*. The soft heap departs from the standard notion of priority queue by allowing for a type of error, referred to as *corruption*, which confers enhanced efficiency. When an item becomes corrupted, its key value gets increased. Findmin returns the minimum current key, which might or might not be corrupted. The user has no control over which items become corrupted, this prerogative belonging to the data structure. But the user does control the overall amount of corruption in the following sense.

The user specifies a parameter, $0 < \epsilon \leq 1/2$, referred to as the error rate, that governs the behavior of the data structure as follows. The operations findmin and deletion are supported in constant amortized time, and insertion is supported in $O(\log 1/\epsilon)$ amortized time. Moreover, no more than an ϵ fraction of the items present in the heap are corrupted at any given time.

To illustrate the concept, let x be an item returned by findmin, from a soft heap of size n. Then there are no more than ϵn items in the heap whose original keys are less than the original key of x.

Soft heaps are rather subtle, and we won't attempt to discuss specifics of their design. Soft heaps have been used to construct an optimal comparison-based minimum spanning tree algorithm (Pettie and Ramachandran [12]), although its actual running time has not been determined. Soft heaps have also been used to construct a comparison-based algorithm with known running time $m\alpha(m, n)$ on a graph with n vertices and m edges (Chazelle [2]), where $\alpha(m, n)$ is a functional inverse of the Ackermann function. Chazelle [1] has also observed that soft heaps can be used to implement median selection in linear time; a significant departure from previous methods.

References

[1] B. Chazelle, The Soft Heap: An Approximate Priority Queue with Optimal Error Rate, *Journal of the ACM*, 47 (2000), 1012–1027.

[2] B. Chazelle, A Faster Deterministic Algorithm for Minimum Spanning Trees, *Journal of the ACM*, 47 (2000), 1028–1047.

[3] M. L. Fredman, On the Efficiency of Pairing Heaps and Related Data Structures, *Journal of the ACM*, 46 (1999), 473–501.

[4] M. L. Fredman, A Priority Queue Transform, *WAE: International Workshop on Algorithm Engineering* LNCS 1668 (1999), 243–257.

[5] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan, The Pairing Heap: A New Form of Self-adjusting Heap, *Algorithmica*, 1 (1986), 111–129.

[6] M. L. Fredman and R. E. Tarjan, Fibonacci Heaps and Their Uses in Improved Optimization Algorithms, *Journal of the ACM*, 34 (1987), 596–615.

[7] J. Iacono, New upper bounds for pairing heaps, *Scandinavian Workshop on Algorithm Theory*, LNCS 1851 (2000), 35–42.

[8] J. Iacono, Distribution sensitive data structures, *Ph.D. Thesis, Rutgers University*, 2001.

[9] D. E. Knuth, *The Stanford Graph Base*, ACM Press, New York, N.Y., 1994.

[10] D. E. Knuth, *Sorting and Searching* 2d ed., Addison-Wesley, Reading MA., 1998.

[11] B. M. E. Moret and H. D. Shapiro, An Empirical Analysis of Algorithms for Constructing a Minimum Spanning Tree, *Proceedings of the Second Workshop on Algorithms and Data Structures* (1991), 400–411.

[12] S. Pettie and V. Ramachandran, An Optimal Minimum Spanning Tree Algorithm, *Journal of the ACM* 49 (2002), 16–34.

[13] J. Vuillemin, A Data Structure for Manipulating Priority Queues, *Communications of the ACM*, 21 (1978), 309–314.

[14] M. A. Weiss, *Data Structures and Algorithms in C*, 2d ed., Addison-Wesley, Reading MA., 1997.

8

Double-Ended Priority Queues

Sartaj Sahni
University of Florida

8.1 Definition and an Application

A *double-ended priority queue (DEPQ)* is a collection of zero or more elements. Each element has a priority or value. The operations performed on a double-ended priority queue are:

1. $getMin()$ ... return element with minimum priority
2. $getMax()$ ... return element with maximum priority
3. $put(x)$ ... insert the element x into the DEPQ
4. $removeMin()$ ... remove an element with minimum priority and return this element
5. $removeMax()$ ... remove an element with maximum priority and return this element

One application of a DEPQ is to the adaptation of quick sort, which has the the best expected run time of all known internal sorting methods, to external sorting. The basic idea in quick sort is to partition the elements to be sorted into three groups L, M, and R. The middle group M contains a single element called the *pivot*, all elements in the left group L are $\leq$ the pivot, and all elements in the right group R are $\geq$ the pivot. Following this partitioning, the left and right element groups are sorted recursively.

In an external sort, we have more elements than can be held in the memory of our computer. The elements to be sorted are initially on a disk and the sorted sequence is to be left on the disk. When the internal quick sort method outlined above is extended to an

external quick sort, the middle group M is made as large as possible through the use of a DEPQ. The external quick sort strategy is:

1. Read in as many elements as will fit into an internal DEPQ. The elements in the DEPQ will eventually be the middle group of elements.
2. Read in the remaining elements. If the next element is $\leq$ the smallest element in the DEPQ, output this next element as part of the left group. If the next element is $\geq$ the largest element in the DEPQ, output this next element as part of the right group. Otherwise, remove either the max or min element from the DEPQ (the choice may be made randomly or alternately); if the max element is removed, output it as part of the right group; otherwise, output the removed element as part of the left group; insert the newly input element into the DEPQ.
3. Output the elements in the DEPQ, in sorted order, as the middle group.
4. Sort the left and right groups recursively.

In this chapter, we describe four implicit data structures—symmetric min-max heaps, interval heaps, min-max heaps, and deaps—for DEPQs. Also, we describe generic methods to obtain efficient DEPQ data structures from efficient data structures for single-ended priority queues (PQ).[1]

8.2 Symmetric Min-Max Heaps

Several simple and efficient implicit data structures have been proposed for the representation of a DEPQ [1, 2, 4, 5, 16, 17, 21]. All of these data structures are adaptations of the classical heap data structure (Chapter 2) for a PQ. Further, in all of these DEPQ data structures, $getMax$ and $getMin$ take $O(1)$ time and the remaining operations take $O(\log n)$ time each (n is the number of elements in the DEPQ). The symmetric min-max heap structure of Arvind and Pandu Rangan [1] is the simplest of the implicit data structures for DEPQs. Therefore, we describe this data structure first.

A *symmetric min-max heap* (SMMH) is a complete binary tree in which each node other than the root has exactly one element. The root of an SMMH is empty and the total number of nodes in the SMMH is $n + 1$, where n is the number of elements. Let x be any node of the SMMH. Let $elements(x)$ be the elements in the subtree rooted at x but excluding the element (if any) in x. Assume that $elements(x) \neq \emptyset$. x satisfies the following properties:

1. The left child of x has the minimum element in $elements(x)$.
2. The right child of x (if any) has the maximum element in $elements(x)$.

Figure 8.1 shows an example SMMH that has 12 elements. When x denotes the node with 80, $elements(x) = \{6, 14, 30, 40\}$; the left child of x has the minimum element 6 in $elements(x)$; and the right child of x has the maximum element 40 in $elements(x)$. You may verify that every node x of this SMMH satisfies the stated properties.

Since an SMMH is a complete binary tree, it is stored as an implicit data structure using the standard mapping of a complete binary tree into an array. When $n = 1$, the minimum and maximum elements are the same and are in the left child of the root of the SMMH.

[1] A *minPQ* supports the operations $getmin()$, $put(x)$, and $removeMin()$ while a *maxPQ* supports the operations $getMax()$, $put(x)$, and $removeMax()$.

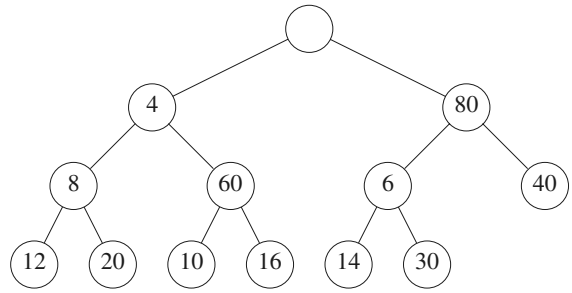


FIGURE 8.1: A symmetric min-max heap.

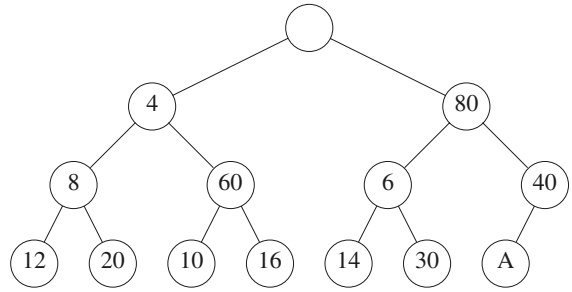


FIGURE 8.2: The SMMH of Figure 8.1 with a node added.

When $n > 1$, the minimum element is in the left child of the root and the maximum is in the right child of the root. So the $getMin$ and $getMax$ operations take $O(1)$ time.

It is easy to see that an $n + 1$-node complete binary tree with an empty root and one element in every other node is an SMMH iff the following are true:

P1: For every node x that has a right sibling, the element in x is less than or equal to that in the right sibling of x.

P2: For every node x that has a grandparent, the element in the left child of the grandparent is less than or equal to that in x.

P3: For every node x that has a grandparent, the element in the right child of the grandparent is greater than or equal to that in x.

Notice that if property P1 is satisfied at node x, then at most one of $P2$ and $P3$ may be violated at x. Using properties P1 through P3 we arrive at simple algorithms to insert and remove elements. These algorithms are simple adaptations of the corresponding algorithms for min and max heaps. Their complexity is $O(\log n)$. We describe only the insert operation. Suppose we wish to insert 2 into the SMMH of Figure 8.1. Since an SMMH is a complete binary tree, we must add a new node to the SMMH in the position shown in Figure 8.2; the new node is labeled A. In our example, A will denote an empty node.

If the new element 2 is placed in node A, property P2 is violated as the left child of the grandparent of A has 6. So we move the 6 down to A and obtain Figure 8.3.

Now we see if it is safe to insert the 2 into node A. We first notice that property P1 cannot be violated, because the previous occupant of node A was greater than 2. Similarly, property P3 cannot be violated. Only P2 can be violated only when $x = A$. So we check P2 with $x = A$. We see that P2 is violated because the left child of the grandparent of A

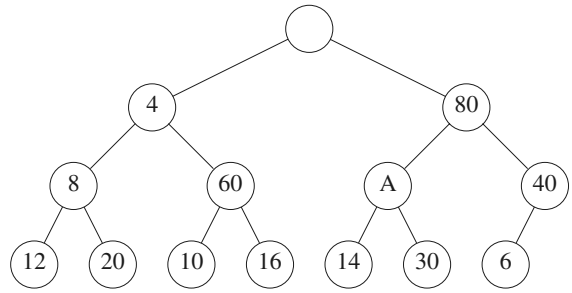


FIGURE 8.3: The SMMH of Figure 8.2 with 6 moved down.

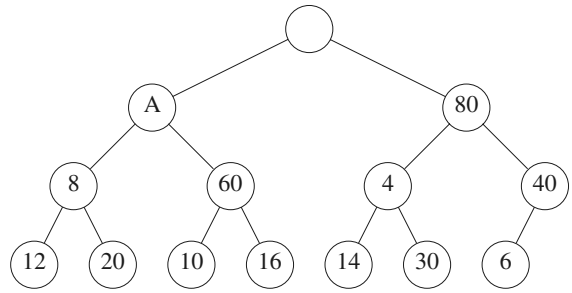


FIGURE 8.4: The SMMH of Figure 8.3 with 4 moved down.

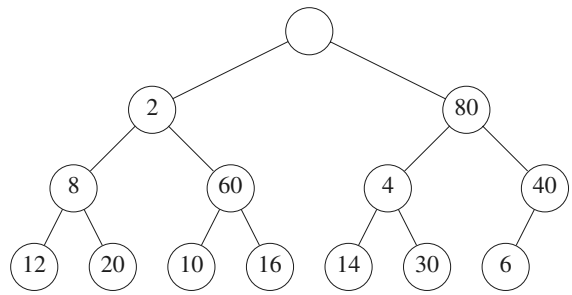


FIGURE 8.5: The SMMH of Figure 8.4 with 2 inserted.

has the element 4. So we move the 4 down to A and obtain the configuration shown in Figure 8.4.

For the configuration of Figure 8.4 we see that placing 2 into node A cannot violate property P1, because the previous occupant of node A was greater than 2. Also properties P2 and P3 cannot be violated, because node A has no grandparent. So we insert 2 into node A and obtain Figure 8.5.

Let us now insert 50 into the SMMH of Figure 8.5. Since an SMMH is a complete binary tree, the new node must be positioned as in Figure 8.6.

Since A has a right child of its parent, we first check P1 at node A. If the new element (in this case 50) is smaller than that in the left sibling of A, we swap the new element and the element in the left sibling. In our case, no swap is done. Then we check P2 and P3.

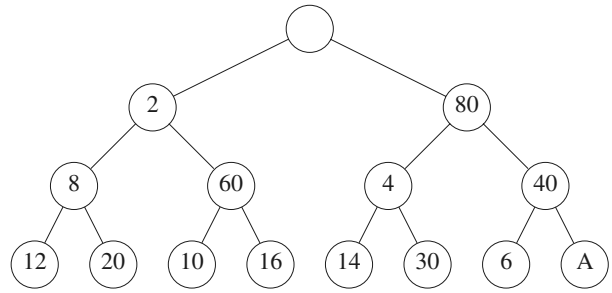


FIGURE 8.6: The SMMH of Figure 8.5 with a node added.

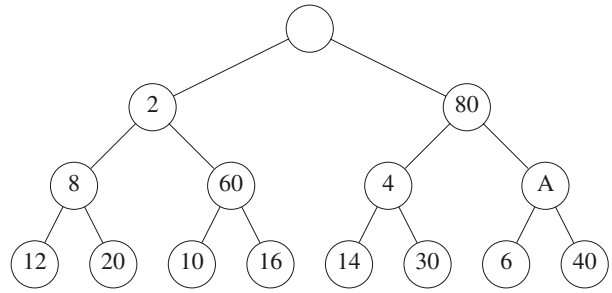


FIGURE 8.7: The SMMH of Figure 8.6 with 40 moved down.

We see that placing 50 into A would violate P3. So the element 40 in the right child of the grandparent of A is moved down to node A. Figure 8.7 shows the resulting configuration. Placing 50 into node A of Figure 8.7 cannot create a P1 violation because the previous occupant of node A was smaller. Also, a P2 violation isn't possible either. So only P3 needs to be checked at A. Since there is no P3 violation at A, 50 is placed into A.

The algorithm to remove either the min or max element is a similar adaptation of the trickle-down algorithm used to remove an element from a min or max heap.

8.3 Interval Heaps

The twin heaps of [21], the min-max pair heaps of [17], the interval heaps of [11, 16], and the diamond deques of [5] are virtually identical data structures. In each of these structures, an n element DEPQ is represented by a min heap with $\lceil n/2 \rceil$ elements and a max heap with the remaining $\lfloor n/2 \rfloor$ elements. The two heaps satisfy the property that each element in the min heap is $\leq$ the corresponding element (two elements correspond if they occupy the same position in their respective binary trees) in the max heap. When the number of elements in the DEPQ is odd, the min heap has one element (i.e., element $\lceil n/2 \rceil$) that has no corresponding element in the max heap. In the twin heaps of [21], this is handled as a special case and one element is kept outside of the two heaps. In min-max pair heaps, interval heaps, and diamond deques, the case when n is odd is handled by requiring element $\lceil n/2 \rceil$ of the min heap to be $\leq$ element $\lfloor n/4 \rfloor$ of the max heap.

In the twin heaps of [21], the min and max heaps are stored in two arrays *min* and *max*

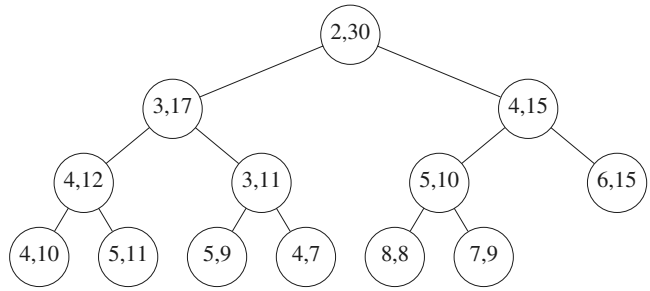


FIGURE 8.8: An interval heap.

using the standard array representation of a complete binary tree[2] [15]. The correspondence property becomes $min[i] \leq max[i]$, $1 \leq i \leq \lfloor n/2 \rfloor$. In the min-max pair heaps of [17] and the interval heaps of [16], the two heaps are stored in a single array *minmax* and we have $minmax[i].min$ being the i'th element of the min heap, $1 \leq i \leq \lceil n/2 \rceil$ and $minmax[i].max$ being the i'th element of the max heap, $1 \leq i \leq \lfloor n/2 \rfloor$. In the diamond deque [5], the two heaps are mapped into a single array with the min heap occupying even positions (beginning with position 0) and the max heap occupying odd positions (beginning with position 1). Since this mapping is slightly more complex than the ones used in twin heaps, min-max pair heaps, and interval heaps, actual implementations of the diamond deque are expected to be slightly slower than implementations of the remaining three structures.

Since the twin heaps of [21], the min-max pair heaps of [17], the interval heaps of [16], and the diamond deques of [5] are virtually identical data structures, we describe only one of these—interval heaps—in detail. An *interval heap* is a complete binary tree in which each node, except possibly the last one (the nodes of the complete binary tree are ordered using a level order traversal), contains two elements. Let the two elements in node P be a and b, where $a \leq b$. We say that the node P represents the closed interval $[a, b]$. a is the left end point of the interval of P, and b is its right end point.

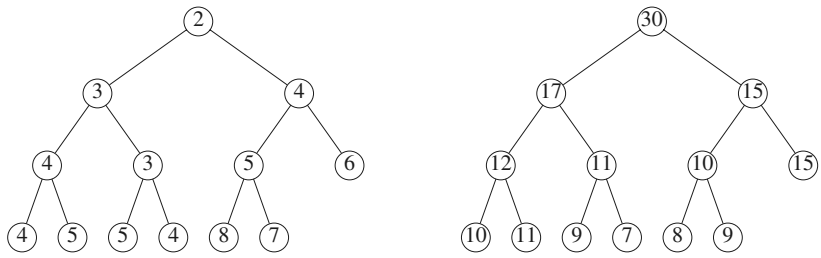
The interval $[c, d]$ is contained in the interval $[a, b]$ iff $a \leq c \leq d \leq b$. In an interval heap, the intervals represented by the left and right children (if they exist) of each node P are contained in the interval represented by P. When the last node contains a single element c, then $a \leq c \leq b$, where $[a, b]$ is the interval of the parent (if any) of the last node.

Figure 8.8 shows an interval heap with 26 elements. You may verify that the intervals represented by the children of any node P are contained in the interval of P.

The following facts are immediate:

1. The left end points of the node intervals define a min heap, and the right end points define a max heap. In case the number of elements is odd, the last node has a single element which may be regarded as a member of either the min or max heap. Figure 8.9 shows the min and max heaps defined by the interval heap of Figure 8.8.
2. When the root has two elements, the left end point of the root is the minimum element in the interval heap and the right end point is the maximum. When

[2] In a *full* binary tree, every non-empty level has the maximum number of nodes possible for that level. Number the nodes in a full binary tree 1, 2, $\cdots$ beginning with the root level and within a level from left to right. The nodes numbered 1 through n define the unique *complete binary tree* that has n nodes.



(a) min heap (b) max heap

FIGURE 8.9: Min and max heaps embedded in Figure 8.8.

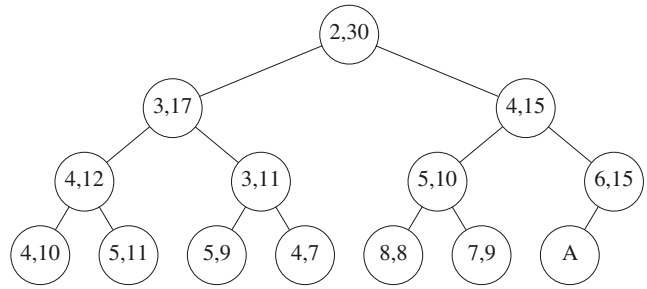


FIGURE 8.10: Interval heap of Figure 8.8 after one node is added.

the root has only one element, the interval heap contains just one element. This element is both the minimum and maximum element.

3. An interval heap can be represented compactly by mapping into an array as is done for ordinary heaps. However, now, each array position must have space for two elements.
4. The height of an interval heap with n elements is $\Theta(\log n)$.

8.3.1 Inserting an Element

Suppose we are to insert an element into the interval heap of Figure 8.8. Since this heap currently has an even number of elements, the heap following the insertion will have an additional node A as is shown in Figure 8.10.

The interval for the parent of the new node A is $[6, 15]$. Therefore, if the new element is between 6 and 15, the new element may be inserted into node A. When the new element is less than the left end point 6 of the parent interval, the new element is inserted into the min heap embedded in the interval heap. This insertion is done using the min heap insertion procedure starting at node A. When the new element is greater than the right end point 15 of the parent interval, the new element is inserted into the max heap embedded in the interval heap. This insertion is done using the max heap insertion procedure starting at node A.

If we are to insert the element 10 into the interval heap of Figure 8.8, this element is put into the node A shown in Figure 8.10. To insert the element 3, we follow a path from node A towards the root, moving left end points down until we either pass the root or reach a node whose left end point is ≤ 3. The new element is inserted into the node that now has

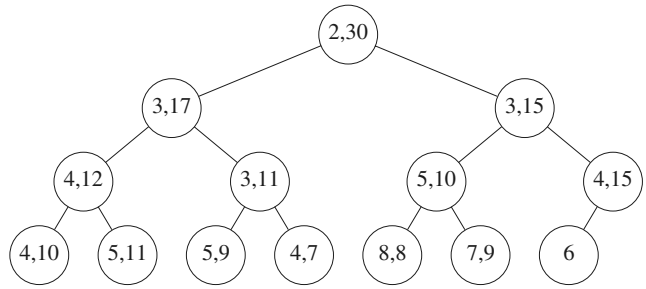


FIGURE 8.11: The interval heap of Figure 8.8 with 3 inserted.

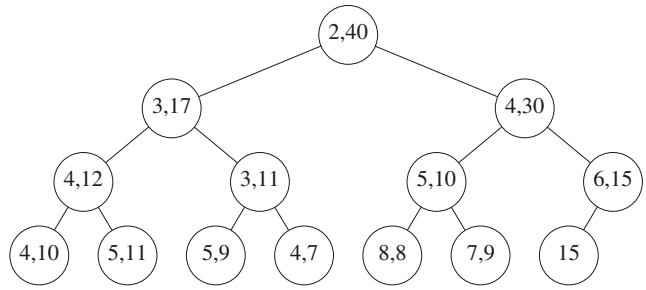


FIGURE 8.12: The interval heap of Figure 8.8 with 40 inserted.

no left end point. Figure 8.11 shows the resulting interval heap.

To insert the element 40 into the interval heap of Figure 8.8, we follow a path from node A (see Figure 8.10) towards the root, moving right end points down until we either pass the root or reach a node whose right end point is ≥ 40. The new element is inserted into the node that now has no right end point. Figure 8.12 shows the resulting interval heap.

Now, suppose we wish to insert an element into the interval heap of Figure 8.12. Since this interval heap has an odd number of elements, the insertion of the new element does not increase the number of nodes. The insertion procedure is the same as for the case when we initially have an even number of elements. Let A denote the last node in the heap. If the new element lies within the interval $[6, 15]$ of the parent of A, then the new element is inserted into node A (the new element becomes the left end point of A if it is less than the element currently in A). If the new element is less than the left end point 6 of the parent of A, then the new element is inserted into the embedded min heap; otherwise, the new element is inserted into the embedded max heap. Figure 8.13 shows the result of inserting the element 32 into the interval heap of Figure 8.12.

8.3.2 Removing the Min Element

The removal of the minimum element is handled as several cases:

1. When the interval heap is empty, the *removeMin* operation fails.
2. When the interval heap has only one element, this element is the element to be returned. We leave behind an empty interval heap.
3. When there is more than one element, the left end point of the root is to be

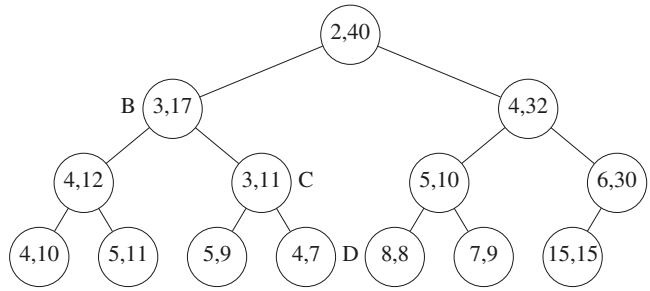


FIGURE 8.13: The interval heap of Figure 8.12 with 32 inserted.

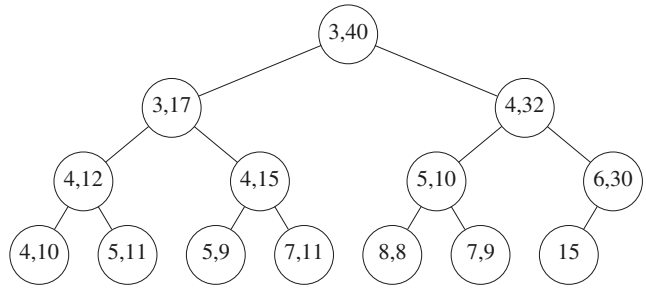


FIGURE 8.14: The interval heap of Figure 8.13 with minimum element removed.

returned. This point is removed from the root. If the root is the last node of the interval heap, nothing more is to be done. When the last node is not the root node, we remove the left point p from the last node. If this causes the last node to become empty, the last node is no longer part of the heap. The point p removed from the last node is reinserted into the embedded min heap by beginning at the root. As we move down, it may be necessary to swap the current p with the right end point r of the node being examined to ensure that $p \leq r$. The reinsertion is done using the same strategy as used to reinsert into an ordinary heap.

Let us remove the minimum element from the interval heap of Figure 8.13. First, the element 2 is removed from the root. Next, the left end point 15 is removed from the last node and we begin the reinsertion procedure at the root. The smaller of the min heap elements that are the children of the root is 3. Since this element is smaller than 15, we move the 3 into the root (the 3 becomes the left end point of the root) and position ourselves at the left child B of the root. Since, $15 \leq 17$ we do not swap the right end point of B with the current $p = 15$. The smaller of the left end points of the children of B is 3. The 3 is moved from node C into node B as its left end point and we position ourselves at node C. Since $p = 15 > 11$, we swap the two and 15 becomes the right end point of node C. The smaller of left end points of Cs children is 4. Since this is smaller than the current $p = 11$, it is moved into node C as this node's left end point. We now position ourselves at node D. First, we swap $p = 11$ and Ds right end point. Now, since D has no children, the current $p = 7$ is inserted into node D as Ds left end point. Figure 8.14 shows the result.

The max element may removed using an analogous procedure.

8.3.3 Initializing an Interval Heap

Interval heaps may be initialized using a strategy similar to that used to initialize ordinary heaps–work your way from the heap bottom to the root ensuring that each subtree is an interval heap. For each subtree, first order the elements in the root; then reinsert the left end point of this subtree's root using the reinsertion strategy used for the $removeMin$ operation, then reinsert the right end point of this subtree's root using the strategy used for the $removeMax$ operation.

8.3.4 Complexity of Interval Heap Operations

The operations $isEmpty()$, $size()$, $getMin()$, and $getMax()$ take $O(1)$ time each; $put(x)$, $removeMin()$, and $removeMax()$ take $O(\log n)$ each; and initializing an n element interval heap takes $O(n)$ time.

8.3.5 The Complementary Range Search Problem

In the *complementary range search* problem, we have a dynamic collection (i.e., points are added and removed from the collection as time goes on) of one-dimensional points (i.e., points have only an x-coordinate associated with them) and we are to answer queries of the form: what are the points outside of the interval $[a, b]$? For example, if the point collection is $3, 4, 5, 6, 8, 12$, the points outside the range $[5, 7]$ are $3, 4, 8, 12$.

When an interval heap is used to represent the point collection, a new point can be inserted or an old one removed in $O(\log n)$ time, where n is the number of points in the collection. Note that given the location of an arbitrary element in an interval heap, this element can be removed from the interval heap in $O(\log n)$ time using an algorithm similar to that used to remove an arbitrary element from a heap.

The complementary range query can be answered in $\Theta(k)$ time, where k is the number of points outside the range $[a, b]$. This is done using the following recursive procedure:

1. If the interval tree is empty, *return*.
2. If the root interval is contained in $[a, b]$, then all points are in the range (therefore, there are no points to report), *return*.
3. Report the end points of the root interval that are not in the range $[a, b]$.
4. Recursively search the left subtree of the root for additional points that are not in the range $[a, b]$.
5. Recursively search the right subtree of the root for additional points that are not in the range $[a, b]$.
6. *return*

Let us try this procedure on the interval heap of Figure 8.13. The query interval is $[4, 32]$. We start at the root. Since the root interval is not contained in the query interval, we reach step 3 of the procedure. Whenever step 3 is reached, we are assured that at least one of the end points of the root interval is outside the query interval. Therefore, each time step 3 is reached, at least one point is reported. In our example, both points 2 and 40 are outside the query interval and so are reported. We then search the left and right subtrees of the root for additional points. When the left subtree is searched, we again determine that the root interval is not contained in the query interval. This time only one of the root interval points (i.e., 3) is to be outside the query range. This point is reported and we proceed to search the left and right subtrees of B for additional points outside the query range. Since

the interval of the left child of B is contained in the query range, the left subtree of B contains no points outside the query range. We do not explore the left subtree of B further. When the right subtree of B is searched, we report the left end point 3 of node C and proceed to search the left and right subtrees of C. Since the intervals of the roots of each of these subtrees is contained in the query interval, these subtrees are not explored further. Finally, we examine the root of the right subtree of the overall tree root, that is the node with interval $[4, 32]$. Since this node's interval is contained in the query interval, the right subtree of the overall tree is not searched further.

The complexity of the above six step procedure is Θ(number of interval heap nodes visited). The nodes visited in the preceding example are the root and its two children, node B and its two children, and node C and its two children. Thus, 7 nodes are visited and a total of 4 points are reported.

We show that the total number of interval heap nodes visited is at most $3k + 1$, where k is the number of points reported. If a visited node reports one or two points, give the node a count of one. If a visited node reports no points, give it a count of zero and add one to the count of its parent (unless the node is the root and so has no parent). The number of nodes with a nonzero count is at most k. Since no node has a count more than 3, the sum of the counts is at most $3k$. Accounting for the possibility that the root reports no point, we see that the number of nodes visited is at most $3k + 1$. Therefore, the complexity of the search is $\Theta(k)$. This complexity is asymptotically optimal because every algorithm that reports k points must spend at least $\Theta(1)$ time per reported point.

In our example search, the root gets a count of 2 (1 because it is visited and reports at least one point and another 1 because its right child is visited but reports no point), node B gets a count of 2 (1 because it is visited and reports at least one point and another 1 because its left child is visited but reports no point), and node C gets a count of 3 (1 because it is visited and reports at least one point and another 2 because its left and right children are visited and neither reports a point). The count for each of the remaining nodes in the interval heap is 0.

8.4 Min-Max Heaps

In the min-max heap structure [2], all n DEPQ elements are stored in an n-node complete binary tree with alternating levels being min levels and max levels (Figure 8.15, nodes at max levels are shaded). The root level of a min-max heap is a min level. Nodes on a min level are called min nodes while those on a max level are max nodes. Every min (max) node has the property that its value is the smallest (largest) value in the subtree of which it is the root. Since 5 is in a min node of Figure 8.15, it is the smallest value in its subtree. Also, since 30 and 26 are in max nodes, these are the largest values in the subtrees of which they are the root.

The following observations are a direct consequence of the definition of a min-max heap.

1. When $n = 0$, there is no min nor max element.
2. When $n = 1$, the element in the root is both the min and the max element.
3. When $n > 1$, the element in the root is the min element; the max element is one of the up to two children of the root.

From these observations, it follows that $getMin()$ and $getMax()$ can be done in $O(1)$ time each.

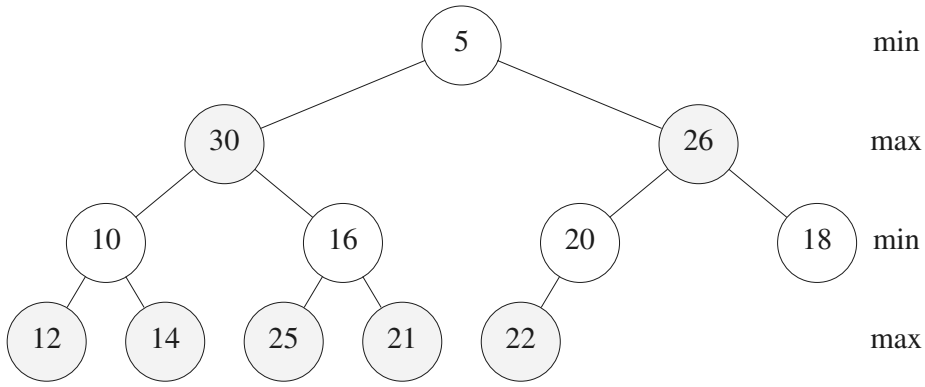


FIGURE 8.15: A 12-element min-max heap.

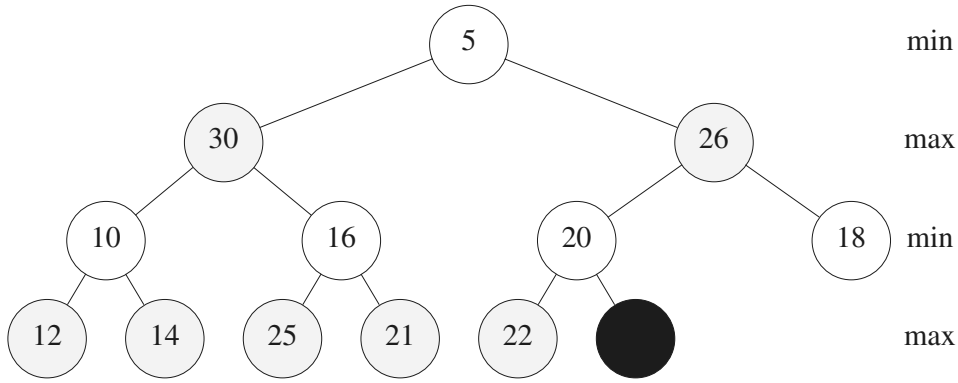


FIGURE 8.16: A 13-node complete binary tree.

8.4.1 Inserting an Element

When inserting an element *newElement* into a min-max heap that has n elements, we go from a complete binary tree that has n nodes to one that has $n+1$ nodes. So, for example, an insertion into the 12-element min-max heap of Figure 8.15 results in the 13-node complete binary tree of Figure 8.16.

When $n = 0$, the insertion simply creates a min-max heap that has a single node that contains the new element. Assume that $n > 0$ and let the element in the parent, *parentNode*, of the new node j be *parentElement*. If *newElement* is placed in the new node j, the min- and max-node property can be violated only for nodes on the path from the root to *parentNode*. So, the insertion of an element need only be concerned with ensuring that nodes on this path satisfy the required min- and max-node property. There are three cases to consider.

1. $parentElement = newElement$
   In this case, we may place *newElement* into node j. With such a placement, the min- and max-node properties of all nodes on the path from the root to *parentNode* are satisfied.
2. $parentNode > newElement$

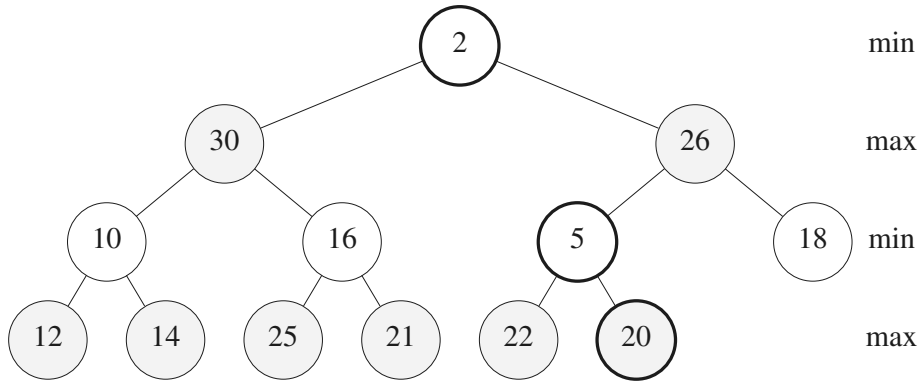


FIGURE 8.17: Min-max heap of Figure 8.15 following insertion of 2.

If *parentNode* is a min node, we get a min-node violation. When a min-node violation occurs, we know that all max nodes on the path from the root to *parentNode* are greater than *newElement*. So, a min-node violation may be fixed by using the trickle-up process used to insert into a min heap; this trickle-up process involves only the min nodes on the path from the root to *parentNode*. For example, suppose that we are to insert $newElement = 2$ into the min-max heap of Figure 8.15. The min nodes on the path from the root to *parentNode* have values 5 and 20. The 20 and the 5 move down on the path and the 2 trickles up to the root node. Figure 8.17 shows the result. When $newElement = 15$, only the 20 moves down and the sequence of min nodes on the path from the root to j have values 5, 15, 20.
The case when *parentNode* is a max node is similar.

3. $parentNode < newElement$
When *parentNode* is a min node, we conclude that all min nodes on the path from the root to *parentNode* are smaller than *newElement*. So, we need be concerned only with the max nodes (if any) on the path from the root to *parentNode*. A trickle-up process is used to correctly position *newElement* with respect to the elements in the max nodes of this path. For the example of Figure 8.16, there is only one max node on the path to *parentNode*. This max node has the element 26. If $newElement > 26$, the 26 moves down to j and *newElement* trickles up to the former position of 26 (Figure 8.18 shows the case when $newElement = 32$). If $newElement < 26$, *newElement* is placed in j.
The case when *parentNode* is a max node is similar.

Since the height of a min-max heap is $\Theta(\log n)$ and a trickle-up examines a single element at at most every other level of the min-max heap, an insert can be done in $O(\log n)$ time.

8.4.2 Removing the Min Element

When $n = 0$, there is no min element to remove. When $n = 1$, the min-max heap becomes empty following the removal of the min element, which is in the root. So assume that $n > 1$. Following the removal of the min element, which is in the root, we need to go from an n-element complete binary tree to an $(n-1)$-element complete binary tree. This causes the element in position n of the min-max heap array to drop out of the min-max

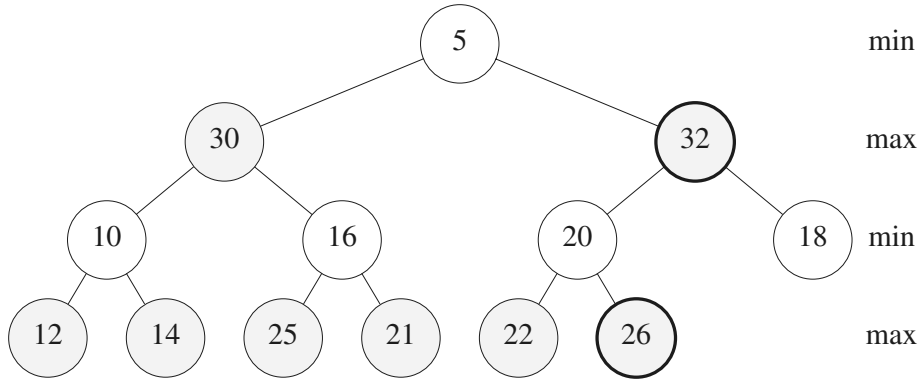


FIGURE 8.18: The min-max heap of Figure 8.15 following the insertion of 32.

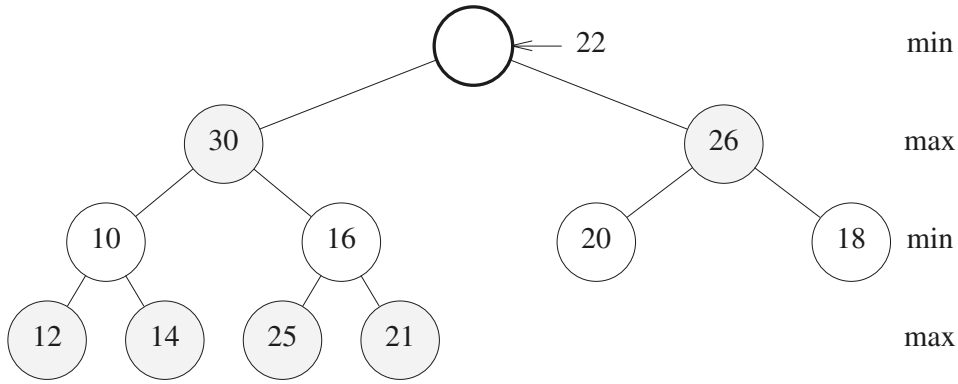


FIGURE 8.19: Situation following a remove min from Figure 8.15.

heap. Figure 8.17 shows the situation following the removal of the min element, 5, from the min-max heap of Figure 8.15. In addition to the 5, which was the min element and which has been removed from the min-max heap, the 22 that was in position $n = 12$ of the min-max heap array has dropped out of the min-max heap. To get the dropped-out element 22 back into the min-max heap, we perform a trickle-down operation that begins at the root of the min-max heap.

The trickle-down operation for min-max heaps is similar to that for min and max heaps. The root is to contain the smallest element. To ensure this, we determine the smallest element in a child or grandchild of the root. If 22 is $\leq$ the smallest of these children and grandchildren, the 22 is placed in the root. If not, the smallest of the children and grandchildren is moved to the root; the trickle-down process is continued from the position vacated by the just moved smallest element.

In our example, examining the children and grandchildren of the root of Figure 8.19, we determine that the smallest of these is 10. Since $10 < 22$, the 10 moves to the root and the 22 trickles down (Figure 8.20). A special case arises when this trickle down of the 22 by 2 levels causes the 22 to trickle past a smaller element (in our example, we trickle past a larger element 30). When this special case arises, we simply exchange the 22 and the smaller element being trickled past. The trickle-down process applied at the vacant node

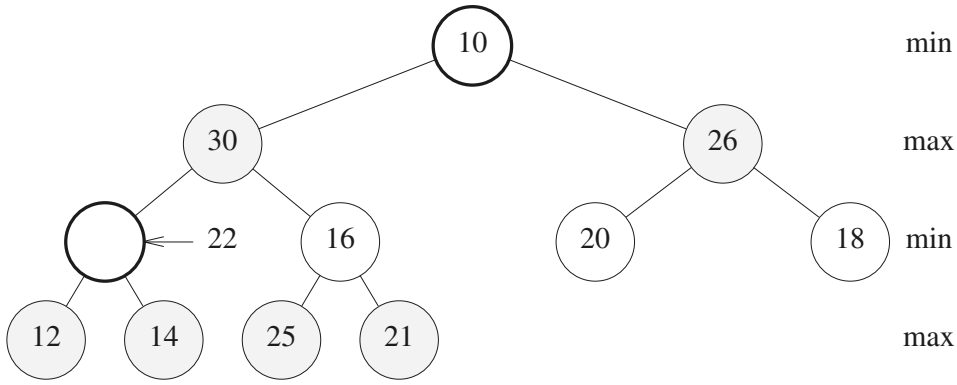


FIGURE 8.20: Situation following one iteration of the trickle-down process.

of Figure 8.20 results in the 22 being placed into the vacant node.

Suppose that *droppedElement* is the element dropped from $minmaxHeap[n]$ when a remove min is done from an n-element min-max heap. The following describes the trickle-down process used to reinsert the dropped element.

1. *The root has no children.*
   In this case *droppedElement* is inserted into the root and the trickle down terminates.
2. *The root has at least one child.*
   Now the smallest key in the min-max heap is in one of the children or grandchildren of the root. We determine which of these nodes has the smallest key. Let this be node k. The following possibilities need to be considered:
   (a) $droppedElement \leq minmaxHeap[k]$.
   droppedElement may be inserted into the root, as there is no smaller element in the heap. The trickle down terminates.
   (b) $droppedElement > minmaxHeap[k]$ and k is a child of the root.
   Since k is a max node, it has no descendants larger than $minmaxHeap[k]$. Hence, node k has no descendants larger than *droppedElement*. So, the $minmaxHeap[k]$ may be moved to the root, and *droppedElement* placed into node k. The trickle down terminates.
   (c) $droppedElement > minmaxHeap[k]$ and k is a grandchild of the root.
   $minmaxHeap[k]$ is moved to the root. Let p be the parent of k. If $droppedElement > minmaxHeap[p]$, then $minmaxHeap[p]$ and *droppedElement* are interchanged. This interchange ensures that the max node p contains the largest key in the subheap with root p. The trickle down continues with k as the root of a min-max (sub) heap into which an element is to be inserted.

The complexity of the remove-min operation is readily seen to be $O(\log n)$. The remove-max operation is similar to the remove-min operation, and min-max heaps may be initialized in $\Theta(n)$ time using an algorithm similar to that used to initialize min and max heaps [15].

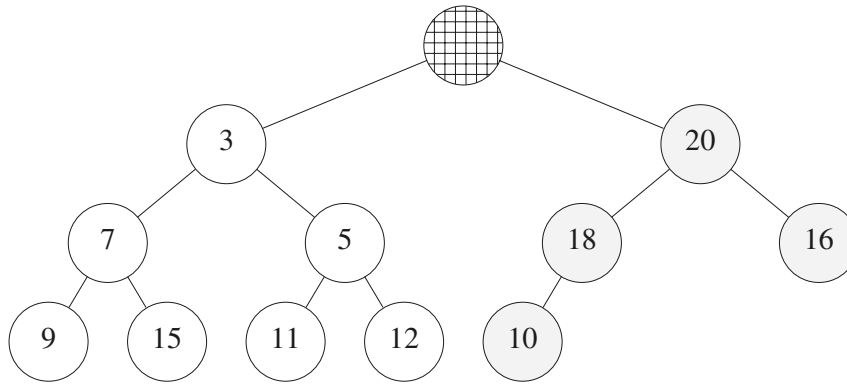


FIGURE 8.21: An 11-element deap.

8.5 Deaps

The deap structure of [4] is similar to the two-heap structures of [5, 16, 17, 21]. At the conceptual level, we have a min heap and a max heap. However, the distribution of elements between the two is not $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$. Rather, we begin with an $(n+1)$-node complete binary tree. Its left subtree is the min heap and its right subtree is the max heap (Figure 8.21, max-heap nodes are shaded). The correspondence property for deaps is slightly more complex than that for the two-heap structures of [5, 16, 17, 21].

A *deap* is a complete binary tree that is either empty or satisfies the following conditions:

1. The root is empty.
2. The left subtree is a min heap and the right subtree is a max heap.
3. *Correspondence property.* Suppose that the right subtree is not empty. For every node x in the left subtree, define its corresponding node y in the right subtree to be the node in the same position as x. In case such a y doesn't exist, let y be the corresponding node for the parent of x. The element in x is $\leq$ the element in y.

For the example complete binary tree of Figure 8.21, the corresponding nodes for the nodes with 3, 7, 5, 9, 15, 11, and 12, respectively, have 20, 18, 16, 10, 18, 16, and 16.

Notice that every node y in the max heap has a unique corresponding node x in the min heap. The correspondence property for max-heap nodes is that the element in y be $\geq$ the element in x. When the correspondence property is satisfied for all nodes in the min heap, this property is also satisfied for all nodes in the max heap.

We see that when $n = 0$, there is no min or max element, when $n = 1$, the root of the min heap has both the min and the max element, and when $n > 1$, the root of the min heap is the min element and the root of the max heap is the max element. So, both $getMin()$ and $getMax()$ may be implemented to work in $O(1)$ time.

8.5.1 Inserting an Element

When an element is inserted into an n-element deap, we go form a complete binary tree that has $n+1$ nodes to one that has $n+2$ nodes. So, the shape of the new deap is well defined. Following an insertion, our 11-element deap of Figure 8.21 has the shape shown in Figure 8.22. The new node is node j and its corresponding node is node i.

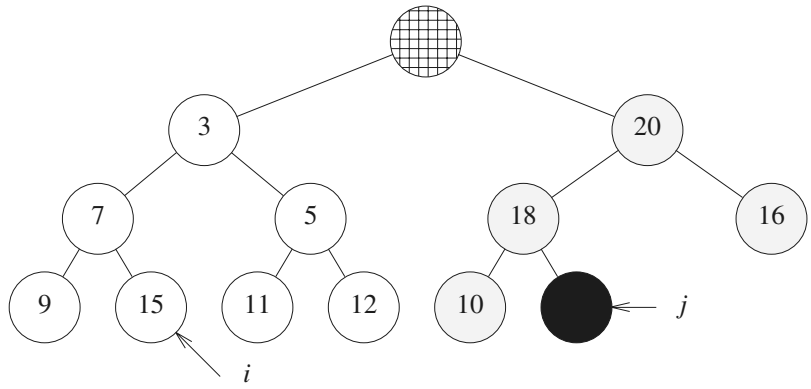


FIGURE 8.22: Shape of a 12-element deap.

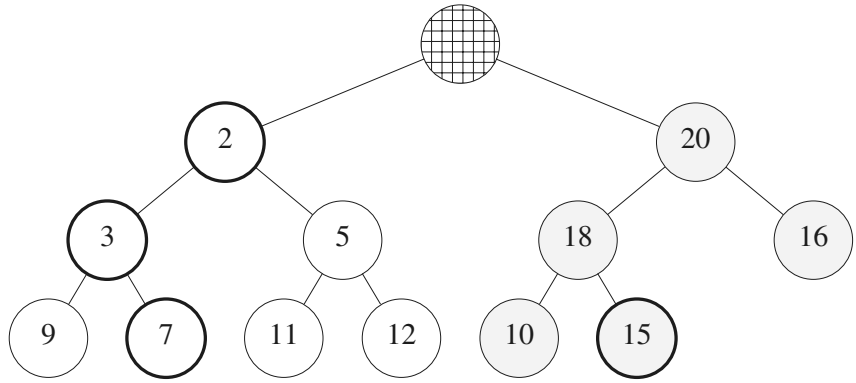


FIGURE 8.23: Deap of Figure 8.21 with 2 inserted.

To insert *newElement*, temporarily place *newElement* into the new node j and check the correspondence property for node j. If the property isn't satisfied, swap *newElement* and the element in its corresponding node; use a trickle-up process to correctly position *newElement* in the heap for the corresponding node i. If the correspondence property is satisfied, do not swap *newElement*; instead use a trickle-up process to correctly place *newElement* in the heap that contains node j.

Consider the insertion of $newElement = 2$ into Figure 8.22. The element in the corresponding node i is 15. Since the correspondence property isn't satisfied, we swap 2 and 15. Node j now contains 15 and this swap is guaranteed to preserve the max-heap properties of the right subtree of the complete binary tree. To correctly position the 2 in the left subtree, we use the standard min-heap trickle-up process beginning at node i. This results in the configuration of Figure 8.23.

To insert $newElement = 19$ into the deap of Figure 8.22, we check the correspondence property between 15 and 19. The property is satisfied. So, we use the trickle-up process for max heaps to correctly position *newElement* in the max heap. Figure 8.24 shows the result.

Since the height of a deap is $\Theta(\log n)$, the time to insert into a deap is $O(\log n)$.

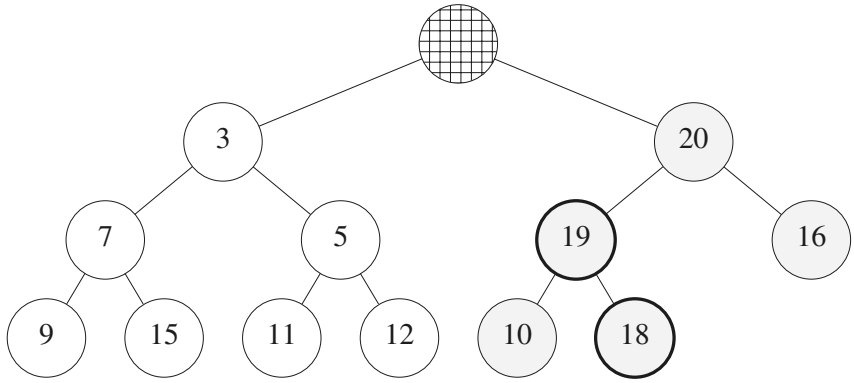


FIGURE 8.24: Deap of Figure 8.21 with 19 inserted.

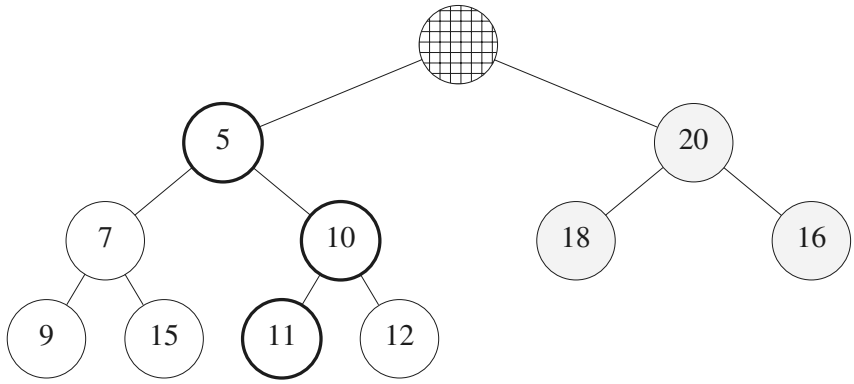


FIGURE 8.25: Deap of Figure 8.21 following a remove min operation.

8.5.2 Removing the Min Element

Assume that $n > 0$. The min element is in the root of the min heap. Following its removal, the deap size reduces to $n-1$ and the element in position $n+1$ of the deap array is dropped from the deap. In the case of our example of Figure 8.21, the min element 3 is removed and the 10 is dropped. To reinsert the dropped element, we first trickle the vacancy in the root of the min heap down to a leaf of the min heap. This is similar to a standard min-heap trickle down with ∞ as the reinsert element. For our example, this trickle down causes 5 and 11 to, respectively, move to their parent nodes. Then, the dropped element 10 is inserted using a trickle-up process beginning at the vacant leaf of the min heap. The resulting deap is shown in Figure 8.25.

Since a *removeMin* requires a trickle-down pass followed by a trickle-up pass and since the height of a deap is $\Theta(\log n)$, the time for a *removeMin* is $O(\log n)$. A *removeMax* is similar. Also, we may initialize a deap in $\Theta(n)$ time using an algorithm similar to that used to initialize a min or max heap [15].

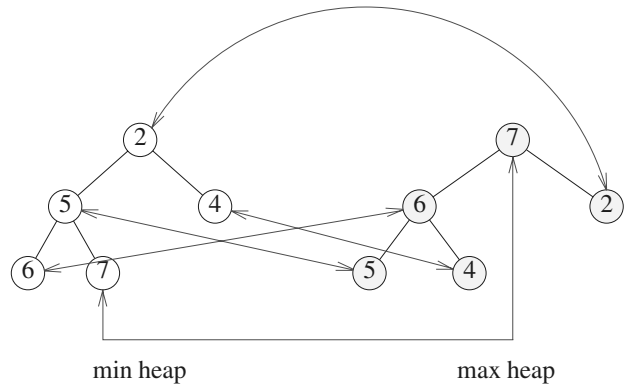


FIGURE 8.26: Dual heap.

8.6 Generic Methods for DEPQs

8.6.1 Dual Priority Queues

General methods [8] exist to arrive at efficient DEPQ data structures from single-ended priority queue data structures that also provide an efficient implementation of the *remove* (*theNode*) operation (this operation removes the node *theNode* from the PQ). The simplest of these methods, *dual structure method*, maintains both a min PQ (called *minPQ*) and a max PQ (called *maxPQ*) of all the DEPQ elements together with *correspondence pointers* between the nodes of the min PQ and the max PQ that contain the same element. Figure 8.26 shows a dual heap structure for the elements $6, 7, 2, 5, 4$. Correspondence pointers are shown as double-headed arrows.

Although Figure 8.26 shows each element stored in both the min and the max heap, it is necessary to store each element in only one of the two heaps.

The minimum element is at the root of the min heap and the maximum element is at the root of the max heap. To insert an element x, we insert x into both the min and the max heaps and then set up correspondence pointers between the locations of x in the min and max heaps. To remove the minimum element, we do a $removeMin$ from the min heap and a $remove(theNode)$, where *theNode* is the corresponding node for the removed element, from the max heap. The maximum element is removed in an analogous way.

8.6.2 Total Correspondence

The notion of total correspondence borrows heavily from the ideas used in a twin heap [21]. In the twin heap data structure n elements are stored in a min heap using an array $minHeap[1:n]$ and n other elements are stored in a max heap using the array $maxHeap[1:n]$. The min and max heaps satisfy the inequality $minHeap[i] \leq maxHeap[i]$, $1 \leq i \leq n$. In this way, we can represent a DEPQ with $2n$ elements. When we must represent a DEPQ with an odd number of elements, one element is stored in a buffer, and the remaining elements are divided equally between the arrays $minHeap$ and $maxHeap$.

In total correspondence, we remove the positional requirement in the relationship between pairs of elements in the min heap and max heap. The requirement becomes: for each element a in $minPQ$ there is a distinct element b in $maxPQ$ such that $a \leq b$ and vice versa. (a, b) is a corresponding pair of elements. Figure 8.27(a) shows a twin heap with 11

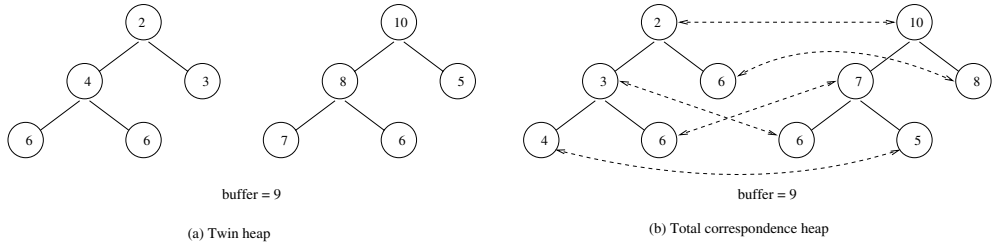


FIGURE 8.27: Twin heap and total correspondence heap.

elements and Figure 8.27(b) shows a total correspondence heap. The broken arrows connect corresponding pairs of elements.

In a twin heap the corresponding pairs $(minHeap[i], maxHeap[i])$ are implicit, whereas in a total correspondence heap these pairs are represented using explicit pointers.

In a total correspondence DEPQ, the number of nodes is either n or $n-1$. The space requirement is half that needed by the dual priority queue representation. The time required is also reduced. For example, if we do a sequence of inserts, every other one simply puts the element in the buffer. The remaining inserts put one element in $maxPQ$ and one in $minPQ$. So, on average, an insert takes time comparable to an insert in either $maxPQ$ or $minPQ$. Recall that when dual priority queues are used the insert time is the sum of the times to insert into $maxPQ$ and $minPQ$. Note also that the size of $maxPQ$ and $minPQ$ together is half that of a dual priority queue.

If we assume that the complexity of the insert operation for priority queues as well as 2 $remove(theNode)$ operations is no more than that of the delete max or min operation (this is true for all known priority queue structures other than weight biased leftist trees [6]), then the complexity of $removeMax$ and $removeMin$ for total correspondence DEPQs is the same as for the $removeMax$ and $removeMin$ operation of the underlying priority queue data structure.

Using the notion of total correspondence, we trivially obtain efficient DEPQ structures starting with any of the known priority queue structures (other than weight biased leftist trees [6]).

The $removeMax$ and $removeMin$ operations can generally be programmed to run faster than suggested by our generic algorithms. This is because, for example, a $removeMax()$ and $put(x)$ into a max priority queue can often be done faster as a single operation $changeMax(x)$. Similarly a $remove(theNode)$ and $put(x)$ can be programmed as a $change$ $(theNode, x)$ operation.

8.6.3 Leaf Correspondence

In leaf correspondence DEPQs, for every leaf element a in $minPQ$, there is a distinct element b in $maxPQ$ such that $a \leq b$ and for every leaf element c in $maxPQ$ there is a distinct element d in $minPQ$ such that $d \leq c$. Figure 8.28 shows a leaf correspondence heap.

Efficient leaf correspondence DEPQs may be constructed easily from PQs that satisfy the following requirements [8]:

(a) The PQ supports the operation $remove(Q, p)$ efficiently.

(b) When an element is inserted into the PQ, no nonleaf node becomes a leaf node (except possibly the node for the newly inserted item).

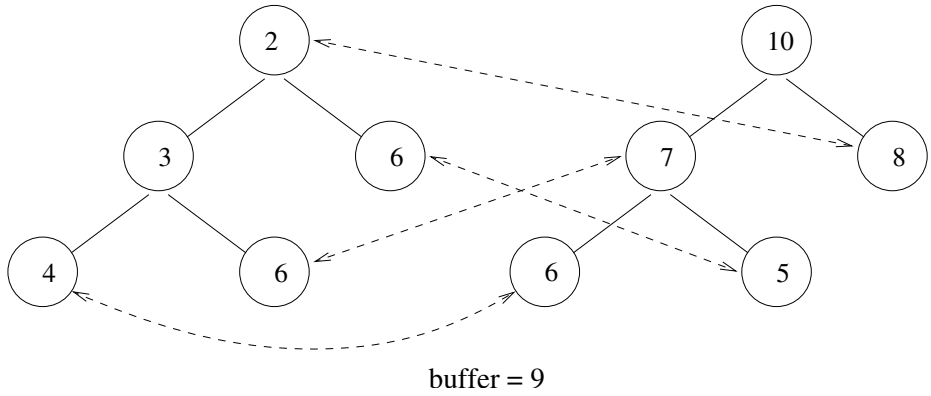


FIGURE 8.28: Leaf correspondence heap.

(c) When an element is deleted (using *remove*, *removeMax* or *removeMin*) from the PQ, no nonleaf node (except possibly the parent of the deleted node) becomes a leaf node.

Some of the PQ structures that satisfy these requirements are height-biased leftist trees (Chapter 5) [9, 15, 20], pairing heaps (Chapter 7) [12, 19], and Fibonacci heaps [13] (Chapter 7). Requirements (b) and (c) are not satisfied, for example, by ordinary heaps and the FMPQ structure of [3]. Although heaps and Brodal's FMPQ structure do not satisfy the requirements of our generic approach to build a leaf correspondence DEPQ structure from a priority queue, we can nonetheless arrive at leaf correspondence heaps and leaf correspondence FMPQs using a customized approach.

8.7 Meldable DEPQs

A *meldable DEPQ (MDEPQ)* is a DEPQ that, in addition to the DEPQ operations listed above, includes the operation

$meld(x, y)$ ... meld the DEPQs x and y into a single DEPQ

The result of melding the double-ended priority queues x and y is a single double-ended priority queue that contains all elements of x and y. The meld operation is destructive in that following the meld, x and y do not remain as independent DEPQs.

To meld two DEPQs in less than linear time, it is essential that the DEPQs be represented using explicit pointers (rather than implicit ones as in the array representation of a heap) as otherwise a linear number of elements need to be moved from their initial to their final locations. Olariu et al. [17] have shown that when the min-max pair heap is represented in such a way, an n element DEPQ may be melded with a k element one ($k \leq n$) in $O(\log(n/k) * \log k)$ time. When $k = \sqrt{n}$, this is $O(\log^2 n)$. Hasham and Sack [14] have shown that the complexity of melding two min-max heaps of size n and k, respectively, is $\Omega(n + k)$. Brodal [3] has developed an MDEPQ implementation that allows one to find the min and max elements, insert an element, and meld two priority queues in O(1) time. The time needed to delete the minimum or maximum element is $O(\log n)$. Although the asymptotic complexity provided by this data structure are the best one can hope for [3], the data structure has practical limitations. First, each element is represented twice using

a total of 16 fields per element. Second, even though the delete operations have $O(\log n)$ complexity, the constant factors are very high and the data structure will not perform well unless find, insert, and meld are the primary operations.

Cho and Sahni [7] have shown that leftist trees [9, 15, 20] may be adapted to obtain a simple representation for MDEPQs in which *meld* takes logarithmic time and the remaining operations have the same asymptotic complexity as when any of the aforementioned DEPQ representations is used. Chong and Sahni [8] study MDEPQs based on pairing heaps [12, 19], Binomial and Fibonacci heaps [13], and FMPQ [3].

Since leftist heaps, pairing heaps, Binomial and Fibonacci heaps, and FMPQs are meldable priority queues that also support the $remove(theNode)$ operation, the MDEPQs of [7, 8] use the generic methods of Section 8.6 to construct an MDEPQ data structure from the corresponding MPQ (meldable PQ) structure.

It is interesting to note that if we use the FMPQ structure of [3] as the base MPQ structure, we obtain a total correspondence MDEPQ structure in which $removeMax$ and $removeMin$ take logarithmic time, and the remaining operations take constant time. This adaptation is superior to the dual priority queue adaptation proposed in [3] because the space requirements are almost half. Additionally, the total correspondence adaptation is faster. Although Brodal's FMPQ structure does not satisfy the requirements of the generic approach to build a leaf correspondence MDEPQ structure from a priority queue, we can nonetheless arrive at leaf correspondence FMPQs using a customized approach.

Acknowledgment

This work was supported, in part, by the National Science Foundation under grant CCR-9912395.

References

[1] A. Arvind and C. Pandu Rangan, Symmetric min-max heap: A simpler data structure for double-ended priority queue, *Information Processing Letters*, 69, 1999, 197-199.

[2] M. Atkinson, J. Sack, N. Santoro, and T. Strothotte, Min-max heaps and generalized priority queues, *Communications of the ACM*, 29, 996-1000, 1986.

[3] G. Brodal, Fast meldable priority queues, *Workshop on Algorithms and Data Structures*, 1995.

[4] S. Carlsson, The deap — A double ended heap to implement double ended priority queues, *Information Processing Letters*, 26, 33-36, 1987.

[5] S. Chang and M. Du, Diamond deque: A simple data structure for priority deques, *Information Processing Letters*, 46, 231-237, 1993.

[6] S. Cho and S. Sahni, Weight biased leftist trees and modified skip lists, *ACM Jr. on Experimental Algorithms*, Article 2, 1998.

[7] S. Cho and S. Sahni, Mergeable double ended priority queue, *International Journal on Foundation of Computer Sciences*, 10, 1, 1999, 1-18.

[8] K. Chong and S. Sahni, Correspondence based data structures for double ended priority queues, *ACM Jr. on Experimental Algorithmics*, Volume 5, 2000, Article 2, 22 pages.

[9] C. Crane, *Linear lists and priority queues as balanced binary trees*, Technical Report CS-72-259, Computer Science Department, Stanford University,

[10] Y. Ding and M. Weiss, The Relaxed Min-Max Heap: A Mergeable Double-Ended Priority Queue, *Acta Informatica*, 30, 215-231, 1993.

[11] Y. Ding and M. Weiss, On the Complexity of Building an Interval Heap, *Information Processing Letters*, 50, 143-144, 1994.

[12] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan, The paring heap : A new form of self-adjusting heap, *Algorithmica*, 1:111-129, 1986.
[13] M. Fredman and R. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, *JACM*, 34:3, 596-615, 1987.
[14] A. Hasham and J. Sack, Bounds for min-max heaps, *BIT*, 27, 315-323, 1987.
[15] E. Horowitz, S. Sahni, D. Mehta, *Fundamentals of Data Structures in C++*, Computer Science Press, NY, 1995.
[16] J. van Leeuwen and D. Wood, Interval heaps, *The Computer Journal*, 36, 3, 209-216, 1993.
[17] S. Olariu, C. Overstreet, and Z. Wen, A mergeable double-ended priority queue, *The Computer Journal*, 34, 5, 423-427, 1991.
[18] D. Sleator and R. Tarjan, Self-adjusting binary search trees, *JACM*, 32:3, 652-686, 1985.
[19] J. T. Stasko and J. S. Vitter, Pairing heaps : Experiments and Analysis, *Communication of the ACM*, 30:3, 234-249, 1987.
[20] R. Tarjan, *Data structures and network algorithms*, SIAM, Philadelphia, PA, 1983.
[21] J. Williams, Algorithm 232, *Communications of the ACM*, 7, 347-348, 1964.