and Lohman in the paper "Differential files: Their application to the maintenance of large databases," by D. Severence and G. Lohman, *ACM Transactions on Database Systems*, 1:3, 1976, pp. 256-267. This paper also provides several advantages of using differential files. The assumptions of uniformity made in the filter error analysis are unrealistic, as, in practice, future accesses are more likely to be for records previously accessed. Several authors have attempted to take this into account. Two references are "A practical guide to the design of differential file architectures," by H. Aghili and D. Severance, *ACM Transactions on Database Systems*, 7:2, 1982, pp. 540-565; and "A regression approach to performance analysis for the differential file architecture," by T. Hill and A. Srinivasan, *Proceedings of the Third IEEE International Conference on Data Engineering*, 1987, pp. 157-164.

Bloom filters have found application in the solution to problems in a variety of domains. Some applications to network related problems may be found in "Space-code Bloom filter for efficient traffic flow measurement," by A. Kumar, J. Xu, L. Li and J. Wang, *ACM Internet Measurement Conference*, 2003; "Hash-based paging and location update using Bloom filters," by P. Mutaf and C. Castelluccia, *Mobile Networks and Applications*, Kluwer Academic, 9, 627-631, 2004; and "Approximate caches for packet classification," by F. Chang, W. Feng and K. Li, *IEEE INFOCOM*, 2004.

CHAPTER 9

Priority Queues

9.1 SINGLE- AND DOUBLE-ENDED PRIORITY QUEUES

A *priority queue* is a collection of elements such that each element has an associated priority. We study two varieties of priority queues—single- and double-ended—in this chapter. Single-ended priority queues, which were first studied in Section 5.6, may be further categorized as min and max priority queues. As noted in Section 5.6.1, the operations supported by a min priority queue are:

SP1: Return an element with minimum priority.

SP2: Insert an element with an arbitrary priority.

SP3: Delete an element with minimum priority.

The operations supported by a max priority queue are the same as those supported by a min priority queue except that in SP1 and SP3 we replace minimum by maximum. The heap structure of Section 5.6 is a classic data structure for the representation of a priority queue. Using a min (max) heap, the minimum (maximum) element can be found

in O(1) time and each of the other two single-ended priority queue operations may be done in O(log n) time, where n is the number of elements in the priority queue. In this chapter, we consider several extensions of a single-ended priority queue. The first extension, *meldable (single-ended) priority queue*, augments the operations SP1 through SP3 with a *meld* operation that melds together two priority queues. One application for the meld operation is when the server for one priority queue shuts down. At this time, it is necessary to meld its priority queue with that of a functioning server. Two data structures for meldable priority queues—leftist trees and binomial heaps—are developed in this chapter.

A further extension of meldable priority queues includes operations to delete an arbitrary element (given its location in the data structure) and to decrease the key/priority (or to increase the key, in case of a max priority queue) of an arbitrary element (given its location in the data structure). Two data structures—Fibonacci heaps and pairing heaps—are developed for this extension. The section on Fibonacci heaps describes how Fibonacci heaps may be used to improve the run time of Dijkstra's shortest paths algorithm of Section 6.4.1.

A *double-ended priority queue* (DEPQ) is a data structure that supports the following operations on a collection of elements.

DP1: Return an element with minimum priority.

DP2: Return an element with maximum priority.

DP3: Insert an element with an arbitrary priority.

DP4: Delete an element with minimum priority.

DP5: Delete an element with maximum priority.

So, a DEPQ is a min and a max priority queue rolled into one structure.

Example 9.1: A DEPQ may be used to implement a network buffer. This buffer holds packets that are waiting their turn to be sent out over a network link; each packet has an associated priority. When the network link becomes available, a packet with the highest priority is transmitted. This corresponds to a *DeleteMax* operation. When a packet arrives at the buffer from elsewhere in the network, it is added to this buffer. This corresponds to an *Insert* operation. However, if the buffer is full, we must drop a packet with minimum priority before we can insert one. This is achieved using a *DeleteMin* operation. □

Example 9.2: In Section 7.10, we saw how to adapt merge sort to the external sorting environment. We now consider a similar adaptation of quick sort, which has the best expected run time of all known internal sorting methods. Recall that the basic idea in quick sort (Section 7.3) 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 (Section 7.10), 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) Process the remaining elements one at a time. 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. □

9.2 LEFTIST TREES

9.2.1 Height-Biased Leftist Trees

Leftist trees provide an efficient implementation of meldable priority queues. Consider the meld operation. Let n be the total number of elements in the two priority queues (throughout this section, we use the term priority queue to mean a single-ended priority queue) that are to be melded. If heaps are used to represent meldable priority queues, then the meld operation takes $O(n)$ time (this may, for example, be accomplished using the heap initialization algorithm of Section 7.6). Using a leftist tree, the meld operation as well as the insert and delete min (or delete max) operations take logarithmic time; the minimum (or maximum) element may be found in $O(1)$ time.

Leftist trees are defined using the concept of an extended binary tree. An *extended binary tree* is a binary tree in which all empty binary subtrees have been replaced by a square node. Figure 9.1 shows two examples of binary trees. Their corresponding extended binary trees are shown in Figure 9.2. The square nodes in an extended binary tree are called *external nodes*. The original (circular) nodes of the binary tree are called *internal nodes*.

There are two varieties of leftist trees—height biased (HBLT) and weight biased (WBLT). We study HBLTs in this section and WBLTs in the next. HBLTs were invented first and are generally referred to simply as leftist trees. We continue with this tradition

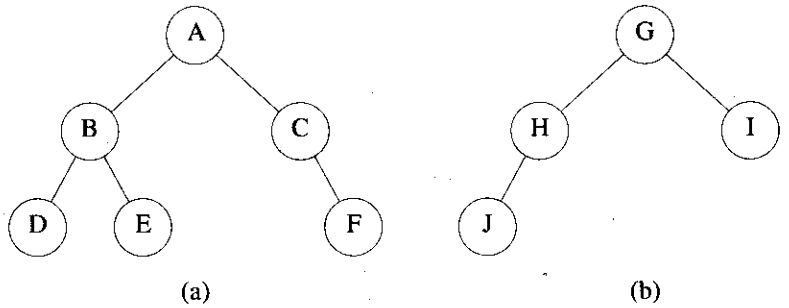


Figure 9.1: Two binary trees

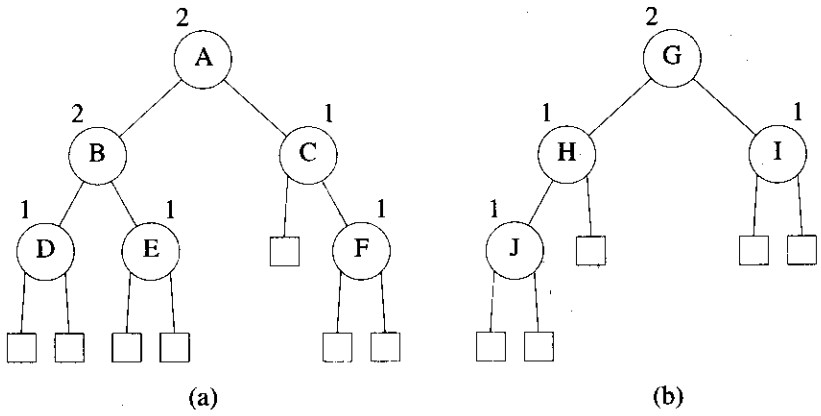


Figure 9.2: Extended binary trees corresponding to Figure 9.1

and refer to HBLTs simply as leftist trees in this section.

Let x be a node in an extended binary tree. Let *leftChild* (x) and *rightChild* (x), respectively, denote the left and right children of the internal node x. Define *shortest* (x) to be the length of a shortest path from x to an external node. It is easy to see that *shortest* (x) satisfies the following recurrence:

$$\text{shortest}(x) = \begin{cases} 0 & \text{if } x \text{ is an external node} \end{cases}$$

The number outside each internal node x of Figure 9.2 is the value of *shortest* (x).

Definition: A *leftist tree* is a binary tree such that if it is not empty, then

$$shortest\,(leftChild\,(x)) \geq shortest\,(rightChild\,(x))$$

for every internal node x. □

The binary tree of Figure 9.1(a), which corresponds to the extended binary tree of Figure 9.2(a), is not a leftist tree, as *shortest* (*leftChild* (C)) = 0, whereas *shortest* (*rightChild*(C)) = 1. The binary tree of Figure 9.1(b) is a leftist tree.

Lemma 9.1: Let r be the root of a leftist tree that has n (internal) nodes.

(a) $n \geq 2^{shortest(r)} - 1$

(b) The rightmost root to external node path is the shortest root to external node path. Its length is $shortest\,(r) \leq \log_2(n+1)$.

Proof: (a) From the definition of *shortest* (r) it follows that there are no external nodes on the first *shortest* (r) levels of the leftist tree. Hence, the leftist tree has at least

$$\sum_{i=1}^{shortest(r)} 2^{i-1} = 2^{shortest(r)} - 1$$

internal nodes. (b) This follows directly from the definition of a leftist tree. □

We represent leftist trees with nodes that have the fields *leftChild*, *rightChild*, *shortest*, and *data*. We assume that *data* is a **struct** with at least a *key* field. We note that we introduced the concept of an external node merely to arrive at clean definitions. The external nodes are never physically present in the representation of a leftist tree. Rather the appropriate child field (*leftChild* or *rightChild*) of the parent of an external node is set to *NULL*. The C declarations are:

```
typedef struct {
        int key;
        /* other fields */
        } element;
typedef struct leftist *leftistTree;
        struct {
               leftistTree leftChild;
               element data;
               leftistTree rightChild;
               int shortest;
               } leftist;
```

Definition: A *min leftist tree* (*max leftist tree*) is a leftist tree in which the key value in each node is no larger (smaller) than the key values in its children (if any). In other words, a min (max) leftist tree is a leftist tree that is also a min (max) tree. □

Two min leftist trees are shown in Figure 9.3. The number inside a node x is the priority of the element in x, and the number outside x is *shortest*(x). *For convenience, all priority queue figures in this chapter show only element priority rather than the complete element.* The operations insert, delete min (delete-max), and meld can be performed in logarithmic time using a min (max) leftist tree. We shall continue our discussion using min leftist trees.

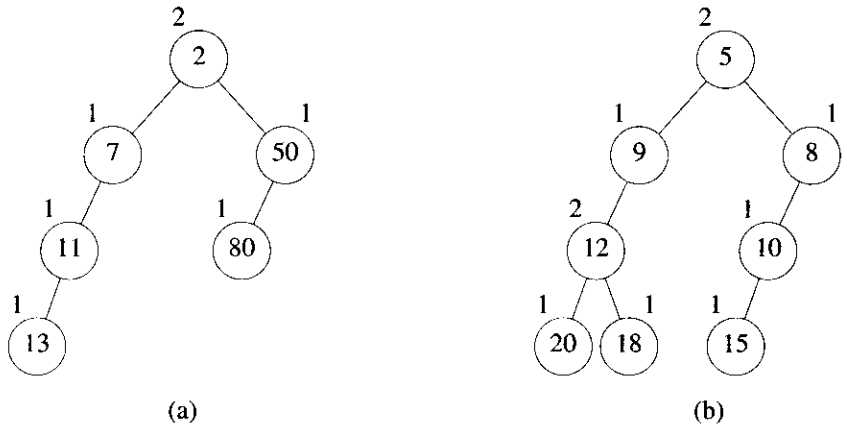


Figure 9.3: Examples of min leftist trees

The insert and delete min operations can both be performed by using the meld operation. To insert an element, x, into a min-leftist tree, a, we first create a min-leftist tree, b, that contains the single element x. Then we meld the min-leftist trees a and b. To delete the min element from a nonempty min-leftist tree, a, we meld the min-leftist trees $a \rightarrow leftChild$ and $a \rightarrow rightChild$ and delete node a.

The meld operation is itself simple. Suppose that we wish to meld the min-leftist trees a and b. First, we obtain a new binary tree containing all elements in a and b by following the rightmost paths in a and/or b. This binary tree has the property that the key in each node is no larger than the keys in its children (if any). Next, we interchange the left and right subtrees of nodes as necessary to convert this binary tree into a leftist tree.

As an example, consider melding the two min leftist trees of Figure 9.3. To obtain a binary tree that contains all the elements in each tree and that satisfies the required

relationship between parent and child keys, we first compare the root keys 2 and 5. Since 2 < 5, the new binary tree should have 2 in its root. We shall leave the left subtree of 2 unchanged and meld 2's right subtree with the entire binary tree rooted at 5. The resulting binary tree will become the new right subtree of 2. When melding the right subtree of 2 and the binary tree rooted at 5, we notice that 5 < 50. So, 5 should be in the root of the melded tree. Now, we proceed to meld the subtrees with root 8 and 50. Since 8 < 50 and 8 has no right subtree, we can make the subtree with root 50 the right subtree of 8. This gives us the binary tree of Figure 9.4(a). Hence, the result of melding the right subtree of 2 and the tree rooted at 5 is the tree of Figure 9.4(b). When the tree of Figure 9.4(b) is made the right subtree of 2, we get the binary tree of Figure 9.4(c). The leftist tree that is made a subtree is represented by shading its nodes, in each step. To convert the tree of Figure 9.4(c) into a leftist tree, we begin at the last modified root (i.e., 8) and trace back to the overall root, ensuring that *shortest* (*leftChild* ( )) ≥ *shortest* (*rightChild* ( )). This inequality holds at 8 but not at 5 and 2. Simply interchanging the left and right subtrees at these nodes causes the inequality to hold. The result is the leftist tree of Figure 9.4(d). The pointers that were interchanged are represented by dotted lines in the figure.

The function *minMeld* (Program 9.1) contains the code to meld two leftist trees. This function uses the recursive function *minUnion* (Program 9.2) to actually meld two nonempty leftist trees. The function *minMeld* intertwines the two steps:

(1) Create a binary tree that contains all elements while ensuring that the root of each subtree has the smallest key in that subtree.

(2) Ensure that each node has a left subtree whose *shortest* value is greater than or equal to that of its right subtree.

Analysis of *minMeld*: Since *minUnion* moves down the rightmost paths in the two leftist trees being combined and since the lengths of these paths is at most logarithmic in the number of elements in each tree, the combining of two leftist trees with a total of n elements is done in time $O(\log n)$. □

9.2.2 Weight-Biased Leftist Trees

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 trees of Figure 9.2 appear in Figure 9.5.

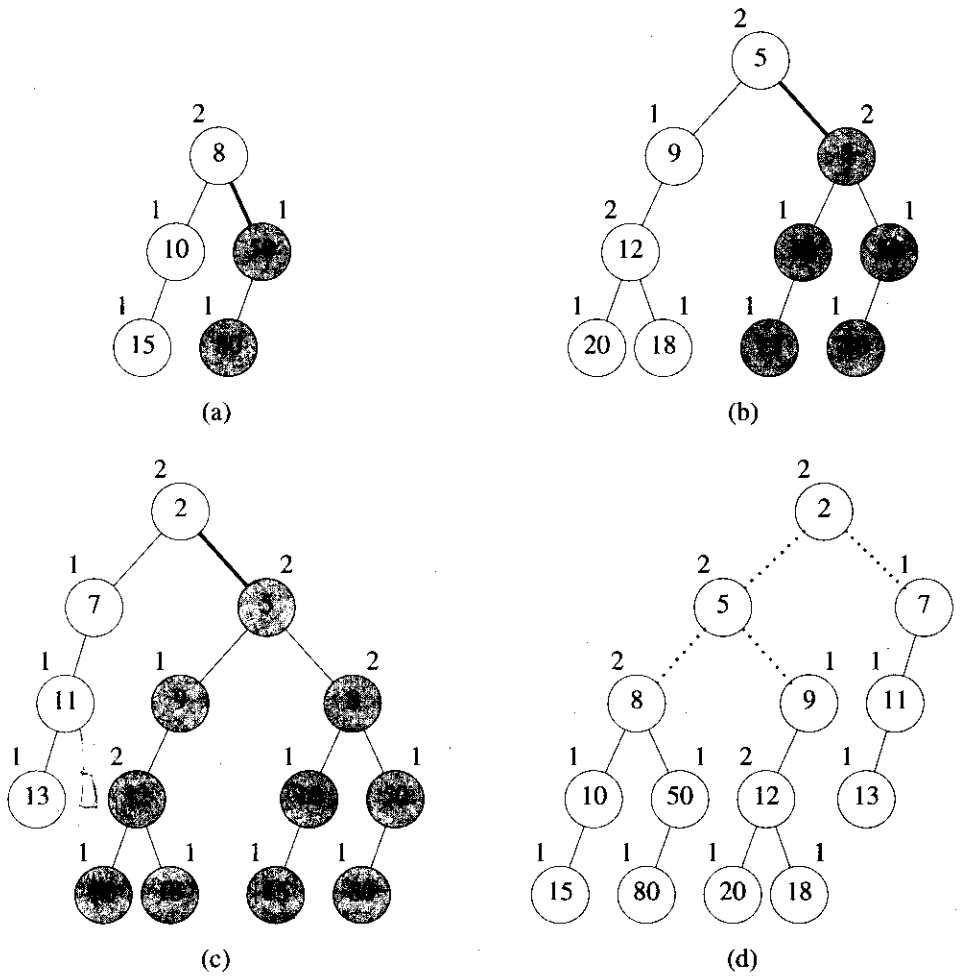


Figure 9.4: Melding the min leftist trees of Figure 9.3

Definition: 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 9.5(a) is not a WBLT while that of Figure 9.5(b) is.

```
void minMeld(leftistTree *a, leftistTree *b)
{/* meld the two min leftist trees *a and *b.  The
    resulting min leftist tree is returned in *a, and *b
    is set to NULL */
   if (!*a) *a = *b;
   else if (*b) minUnion(a,b);
   *b = NULL;
}
```

Program 9.1: : Melding two min-leftist trees

```
void minUnion(leftistTree *a, leftistTree *b)
{/* recursively combine two nonempty min leftist trees */
   leftistTree temp;
   /* set a to be the tree with smaller root */
   if ((*a)→data.key > (*b)→data.key) SWAP(*a,*b,temp);

   /* create binary tree such that the smallest key in each
      subtree is in the root */
   if (!(*a)→rightChild) (*a)→rightChild = *b;
   else minUnion(&(*a)→rightChild, b);

   /*leftist tree property */
   if (!(*a)→leftChild) {
      (*a)→leftChild = (*a)→rightChild;
      (*a)→rightChild = NULL ;
   }
   else if ((*a)→leftChild→shortest <
            (*a)→rightChild→shortest)
      SWAP((*a)→leftChild,(*a)→rightChild, temp);
   (*a)→shortest = (!(*a)→rightChild) ? 1 :
                     (*a)→rightChild→shortest + 1;
}
```

Program 9.2: Melding two nonempty min-leftist trees

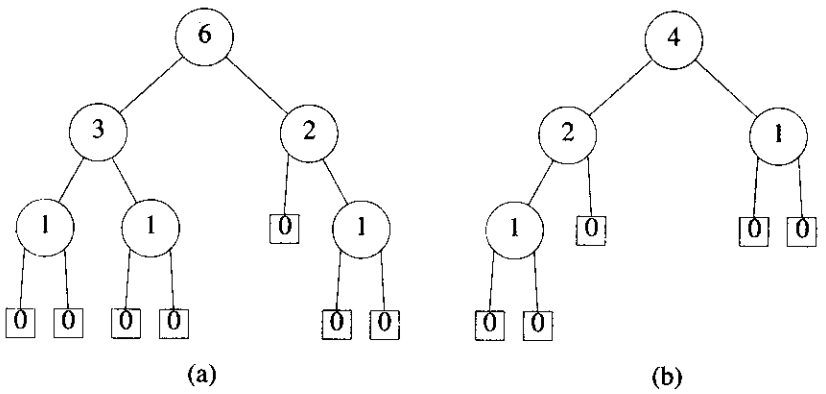


Figure 9.5: Extended binary trees of Figure 9.2 with weights shown

Lemma 9.2: Let x be any internal node of a weight-biased leftist tree. The length, *rightmost* (x), of the rightmost 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

$$\begin{aligned} rightmost(x) &= 1+rightmost(rightChild(x)) \\ &\leq 1+\log_2((n-1)/2+1) \\ &= 1+\log_2(n+1)-1 \\ &= \log_2(n+1). \ \square \end{aligned}$$

The insert, delete max, and initialization operations are analogous to the corresponding max HBLT operations. 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 *shortest*-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 *shortest* value cannot be determined on the way down the tree.

Experiments indicate that meldable single-ended priority queue operations are

faster, by a constant factor, when we use WBLTs rather than HBLTs.

EXERCISES

1. Give an example of a binary tree that is not a leftist tree. Label the nodes of your binary tree with their *shortest* value.
2. Let t be an arbitrary binary tree represented using the node structure for a leftist tree.
   (a) Write a function to initialize the *shortest* field of each node in t.
   (b) Write a function to convert t into a leftist tree.
   (c) What is the complexity of each of these two functions?
3. (a) Into an empty min leftist tree, insert elements with priorities 20, 10, 5, 18, 6, 12, 14, 4, and 22 (in this order). Show the min leftist tree following each insert.
   (b) Delete the min element from the final min leftist tree of part (a). Show the resulting min leftist tree.
4. Compare the performance of leftist trees and min heaps under the assumption that the only operations to be performed are insert and delete min. For this, do the following:
   (a) Create a random list of n elements and a random sequence of insert and delete-min operations of length m. The latter sequence is created such that the probability of an insert or delete-min operation is approximately 0.5. Initialize a min leftist tree and a min heap to contain the n elements in the first random list. Now, measure the time to perform the m operations using the min leftist tree as well as the min heap. Divide this time by m to get the average time per operation. Do this for n = 100, 500, 1000, 2000, $\cdots$, 5000. Let m be 5000. Tabulate your computing times.
   (b) Based on your experiments, make some statements about the relative merits of the two priority-queue schemes.
5. Write a function to initialize a min leftist tree with n elements. Assume that the node structure is the same as that used in the text. Your function must run in $\Theta(n)$ time. Show that this is the case. Can you think of a way to do this initialization in $\Theta(n)$ time such that the resulting min leftist tree is also a complete binary tree?
6. Write a function to delete the element in node x of a min leftist tree. Assume that in addition to the fields stated in the text, each node has the field *parent*, which points to its parent in the leftist tree. What is the complexity of your function?
7. [***Lazy deletion***] Another way to handle the deletion of arbitrary elements from a min-leftist tree is to use a field, *deleted*, in place of the parent field of the previous exercise. When we delete an element, we set its *deleted* field to *TRUE*. However,

we do not physically delete the node. When we perform a *deleteMin* operation, we first search for the minimum element not deleted by carrying out a limited preorder search. This preorder search traverses only the upper part of the tree as needed to identify the min element. All deleted elements encountered are physically deleted, and their subtrees are melded to obtain the new min leftist tree.

(a) Write a function to delete the element in node x of a min leftist tree.

(b) Write another function to delete the min element from a min leftist tree from which several elements have been deleted using the former function.

(c) What is the complexity of your function of part (b)? Provide this as a function of the number of deleted elements encountered and the number of elements in the entire tree?

8. [***Skew heaps***] A *skew heap* is a min tree that supports the min leftist tree operations: insert, delete min, and meld in amortized time (see Section 9.4 for a definition of amortized time) O(log n) per operation. As in the case of min leftist trees, insertions and deletions are performed using the meld operation, which is carried out by following the rightmost paths in the two heaps being melded. However, unlike min leftist trees, the left and right subtrees of all nodes (except the last) on the rightmost path in the resulting heap are interchanged.

   (a) Write insert, delete min, and meld functions for skewed heaps.

   (b) Compare the running times of these with those for the same operations on a min leftist tree. Use random sequences of insert, delete min, and meld operations.

9. [***WBLT***] Develop a complete implementation of a weight-biased min leftist tree. You must include functions to delete and return the min element, insert an arbitrary element, and meld two min WBLTs. Your meld function should perform only a top-to-bottom pass over the WBLTs being melded. Show that the complexity of these three functions is $O(n)$. Test all functions using your own test data.

10. Give an example of an HBLT that is not a WBLT as well as one that is a WBLT but not an HBLT.

9.3 BINOMIAL HEAPS

9.3.1 Cost Amortization

A *binomial heap* is a data structure that supports the same functions (i.e., insert, delete min (or delete-max), and meld) as those supported by leftist trees. Unlike leftist trees, where an individual operation can be performed in O(log n) time, it is possible that certain individual operations performed on a binomial heap may take O(n) time. However, if we amortize (spread out) part of the cost of expensive operations over the inexpensive

ones, then the amortized complexity of an individual operation is either O(1) or O(log n) depending on the type of operation.

Let us examine more closely the concept of cost amortization (we shall use the terms *cost* and *complexity* interchangeably). Suppose that a sequence I1, I2, D1, I3, I4, I5, I6, D2, I7 of insert and delete-min operations is performed. Assume that the *actual cost* of each of the seven inserts is one. By this, we mean that each insert takes one unit of time. Further, suppose that the delete-min operations D1 and D2 have an actual cost of eight and ten, respectively. So, the total cost of the sequence of operations is 25.

In an amortization scheme we charge some of the actual cost of an operation to other operations. This reduces the charged cost of some operations and increases that of others. The *amortized cost* of an operation is the total cost charged to it. The cost transferring (amortization) scheme is required to be such that the sum of the amortized costs of the operations is greater than or equal to the sum of their actual costs. If we charge one unit of the cost of a delete-min operation to each of the inserts since the last delete-min operation (if any), then two units of the cost of D1 get transferred to I1 and I2 (the charged cost of each increases by one), and four units of the cost of D2 get transferred to I3 to I6. The amortized cost of each of I1 to I6 becomes two, that of I7 is equal to its actual cost (i.e., one), and that of each of D1 and D2 becomes 6. The sum of the amortized costs is 25, which is the same as the sum of the actual costs.

Now suppose we can prove that no matter what sequence of insert and delete-min operations is performed, we can charge costs in such a way that the amortized cost of each insertion is no more than two and that of each deletion is no more than six. This will enable us to make the claim that the actual cost of any insert / delete min sequence is no more than $2i + 6d$ where i and d are, respectively, the number of insert and delete min operations in the sequence. Suppose that the actual cost of a deletion is no more than ten, and that of an insertion is one. Using actual costs, we can conclude that the sequence cost is no more than $i + 10d$. Combining these two bounds, we obtain $\min\{2i + 6d, i + 10d\}$ as a bound on the sequence cost. Hence, using the notion of cost amortization, we can obtain tighter bounds on the complexity of a sequence of operations. This is useful, because in many applications, we are concerned more with the time it takes to perform a sequence of priority queue operations than we are with the time it takes to perform an individual operation. For example, when we sort using the heap sort method, we are concerned with the time it takes to complete the entire sort; not with the time it takes to remove the next element from the heap. In applications such as sorting, where we are concerned only with the overall time rather than the time per operation, it is adequate to use a data structure that has a good amortized complexity for each operation type.

We shall use the notion of cost amortization to show that although individual delete operations on a binomial heap may be expensive, the cost of any sequence of binomial heap operations is actually quite small.

9.3.2 Definition of Binomial Heaps

As in the case of heaps and leftist trees, there are two varieties of binomial heaps, min and max. A *min binomial heap* is a collection of min trees; a *max binomial heap* is a collection of max trees. We shall explicitly consider min binomial heaps only. These will be referred to as *B-heaps*. Figure 9.6 shows an example of a B-heap that is made up of three min trees.

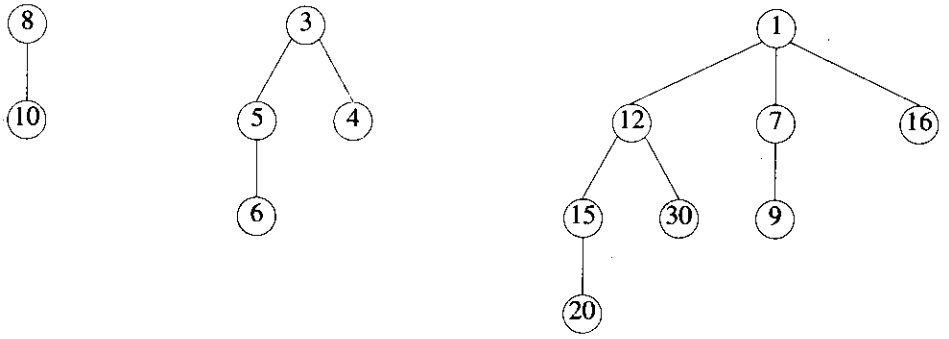


Figure 9.6: A B-heap with three min trees

Using B-heaps, we can perform an insert and a meld in O(1) actual and amortized time and a delete min in O(log n) amortized time. B-heaps are represented using nodes that have the following fields: *degree*, *child*, *link*, and *data*. The *degree* of a node is the number of children it has; the *child* data member is used to point to any one of its children (if any); the *link* data member is used to maintain singly linked circular lists of siblings. All the children of a node form a singly linked circular list, and the node points to one of these children. Additionally, the roots of the min trees that comprise a B-heap are linked to form a singly linked circular list. The B-heap is then pointed at by a single pointer *min* to the min tree root with smallest key.

Figure 9.7 shows the representation for the example of Figure 9.6. To enhance the readability of this figure, we have used bidirectional arrows to join together nodes that are in the same circular list. When such a list contains only one node, no such arrows are drawn. Each of the key sets {10}, {6}, {5,4}, {20}, {15, 30}, {9}, {12, 7, 16}, and {8, 3, 1} denotes the keys in one of the circular lists of Figure 9.7. *min* is the pointer to the B-heap. Note that an empty B-heap has a 0 pointer.

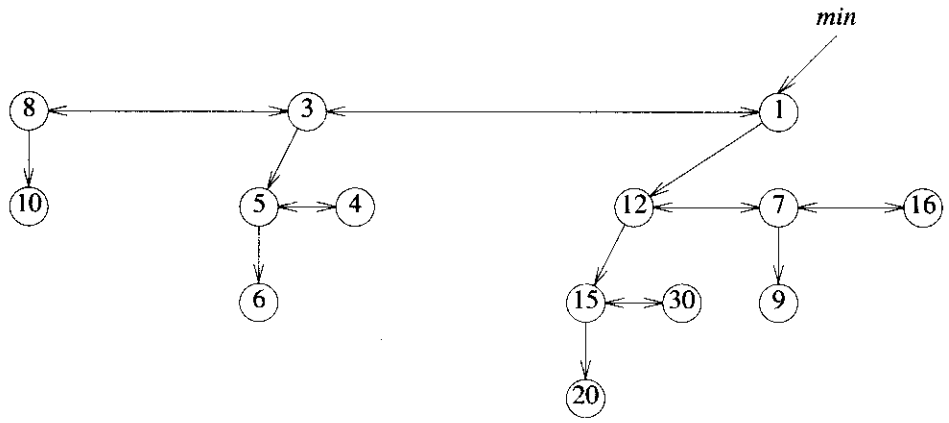


Figure 9.7: B-heap of Figure 9.6 showing child pointers and sibling lists

9.3.3 Insertion into a Binomial Heap

An element x may be inserted into a B-heap by first putting x into a new node and then inserting this node into the circular list pointed at by *min*. The pointer *min* is reset to this new node only if *min* is 0 or the key of x is smaller than the key in the node pointed at by *min*. It is evident that these insertion steps can be performed in O(1) time.

9.3.4 Melding Two Binomial Heaps

To meld two nonempty B-heaps, we meld the top circular lists of each into a single circular list. The new B-heap pointer is the *min* pointer of one of the two trees, depending on which has the smaller key. This can be determined with a single comparison. Since two circular lists can be melded into a single one in O(1) time, the total time required to meld two B-heaps is O(1).

9.3.5 Deletion of Min Element

If *min* is 0, then the B-heap is empty, and a deletion cannot be performed. Assume that *min* is not 0. *min* points to the node that contains the min element. This node is deleted from its circular list. The new B-heap consists of the remaining min trees and the sub-

min trees of the deleted root. Figure 9.8 shows the situation for the example of Figure 9.6.

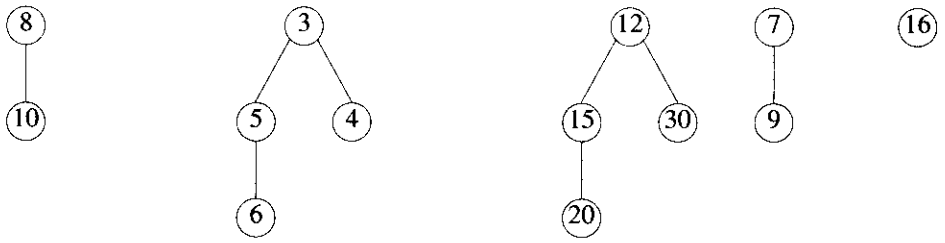


Figure 9.8: The B-heap of Figure 9.6 following the deletion of the min element

Before forming the circular list of min tree roots, we repeatedly join together pairs of min trees that have the same degree (the degree of a nonempty min tree is the degree of its root). *This min tree joining is done by making the min tree whose root has a larger key a subtree of the other (ties are broken arbitrarily).* When two min trees are joined, the degree of the resulting min tree is one larger than the original degree of each min tree, and the number of min trees decreases by one. For our example, we may first join either the min trees with roots 8 and 7 or those with roots 3 and 12. If the first pair is joined, the min tree with root 8 is made a subtree of the min tree with root 7. We now have the min tree collection of Figure 9.9. There are three min trees of degree two in this collection. If the pair with roots 7 and 3 is picked for joining, the resulting min tree collection is that of Figure 9.10. Shaded nodes in Figure 9.9 and Figure 9.10 denote the min tree that was made a subtree in the previous step. Since the min trees in this collection have different degrees, the min tree joining process terminates.

The min tree joining step is followed by a step in which the min tree roots are linked together to form a circular list and the B-heap pointer is reset to point to the min tree root with smallest key. The steps involved in a delete-min operation are summarized in Program 9.3.

Steps 1 and 2 take O(1) time. Step 3 may be implemented by using an array *tree* indexed from 0 to the maximum possible degree, *MAX_DEGREE*, of a min-tree. Initially all entries in this array are *NULL*. Let s be the number of min-trees in a and y. The lists a and y created in step 2 are scanned. For each min-tree p in the lists a and y created in step 2, the code of Program 9.4 is executed. The function *joinMinTrees* makes the input tree with larger root a sub tree of the other tree. The resulting tree is returned in the first parameter. In the end, the array *tree* contains pointers to the min-trees that are to be linked together in step 4. Since each time a pair of min-trees is joined the total number of min-trees decreases by one, the number of joins is at most $s-1$. Hence, the

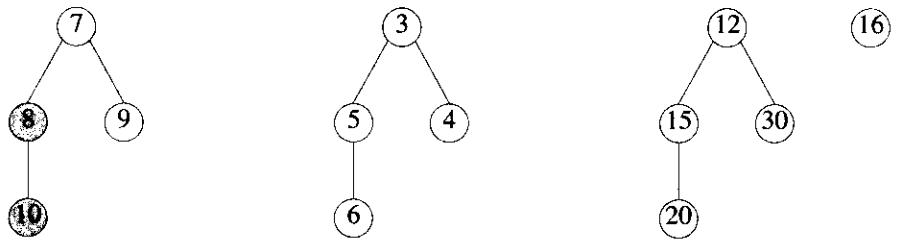


Figure 9.9: The B-heap of Figure 9.8 following the joining of the two degree-one min trees

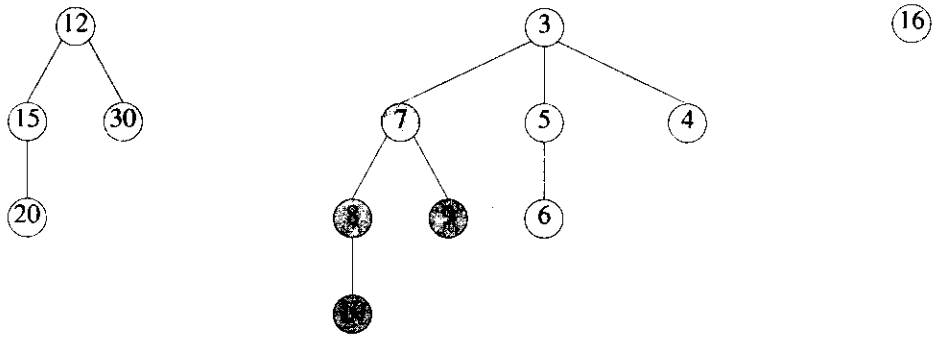


Figure 9.10: The B-heap of Figure 9.9 following the joining of two degree-two min trees

complexity of step 3 is $O(MAX\_DEGREE + s)$.

Step 4 is accomplished by scanning *tree* and linking together the min-trees found. During this scan, the min-tree with minimum key may also be determined. The complexity of step 4 is $O(MAX\_DEGREE)$.

{Delete the min element from a B-heap a, this element is returned in x}

Step 1: [Handle empty B-heap] **if** (a = *NULL*) *deletionError* **else** perform steps 2 - 4;

Step 2: [Deletion from nonempty B-heap] $x = a \rightarrow data$; $y = a \rightarrow child$; delete a from its doubly linked circular list; now, a points to any remaining node in the resulting list; **if** there is no such node, then a = *NULL*;

Step 3: [Min-tree joining] Consider the min-trees in the lists a and y; join together pairs of min-trees of the same degree until all remaining min-trees have different degree;

Step 4: [Form min-tree root list] Link the roots of the remaining min-trees (if any) together to form a doubly linked circular list; set a to point to the root (if any) with minimum key;

Program 9.3: Steps in a delete min

```
for (degree = p→degree; tree[degree]; degree++) {
    joinMinTrees(p,tree[degree]);
    tree[degree] = NULL;
}
tree[degree] = p;
```

Program 9.4: Code to handle min-tree p encountered during a scan of lists a and y

9.3.6 Analysis

Definition: The *binomial tree, B_k, of degree k* is a tree such that if $k = 0$, then the tree has exactly one node, and if $k > 0$, then the tree consists of a root whose degree is k and whose subtrees are $B_0, B_1, \cdots, B_{k-1}$. □

The min trees of Figure 9.6 are B_1, B_2, and B_3, respectively. One may verify that B_k has exactly 2^k nodes. Further, if we start with a collection of empty B-heaps and perform inserts, melds, and delete mins only, then the min trees in each B-heap are binomial trees. These observations enable us to prove that when only inserts, melds, and delete mins are performed, we can amortize costs such that the amortized cost of each insert and meld is O(1), and that of each delete min is O(log n).

Lemma 9.3: Let a be a B-heap with n elements that results from a sequence of insert, meld, and delete-min operations performed on a collection of initially empty B-heaps.

Each min tree in a has degree $\leq \log_2 n$. Consequently, *maxDegree* $\leq \lfloor \log_2 n \rfloor$, and the actual cost of a delete-min operation is $O(\log n + s)$.

Proof: Since each of the min trees in a is a binomial tree with at most n nodes, none can have degree greater than $\lfloor \log_2 n \rfloor$. □

Theorem 9.1: If a sequence of n insert, meld, and delete-min operations is performed on initially empty B-heaps, then we can amortize costs such that the amortized time complexity of each insert and meld is $O(1)$, and that of each delete-min operation is $O(\log n)$.

Proof: For each B-heap, define the quantities *#insert* and *LastSize* in the following way: When an initially empty B-heap is created or when a delete-min operation is performed on a B-heap, its *#insert* value is set to zero. Each time an insert is done on a B-heap, its *#insert* value is increased by one. When two B-heaps are melded, the *#insert* value of the resulting B-heap is the sum of the *#insert* values of the B-heaps melded. Hence, *#insert* counts the number of inserts performed on a B-heap or its constituent B-heaps since the last delete-min operation performed in each. When an initially empty B-heap is created, its *LastSize* value is zero. When a delete-min operation is performed on a B-heap, its *LastSize* is set to the number of min trees it contains following this delete. When two B-heaps are melded, the *LastSize* value for the resulting B-heap is the sum of the *LastSize* values in the two B-heaps that were melded. One may verify that the number of min trees in a B-heap is always equal to *#insert* + *LastSize*.

Consider any individual delete-min operation in the operation sequence. Assume this is from the B-heap a. Observe that the total number of elements in all the B-heaps is at most n, as only inserts add elements, and at most n inserts can be present in a sequence of n operations. Let $u = a.\ min \rightarrow degree \leq \log_2 n$.

From Lemma 9.3, the actual cost of this delete-min operation is $O(\log n + s)$. The $\log n$ term is due to *maxDegree* and represents the time needed to initialize the array *tree* and to complete Step 4. The s term represents the time to scan the lists *min* and y and to perform the $s-1$ (at most) min tree joins. We see that s = *#insert* + *LastSize* + $u - 1$. If we charge *#insert* units of cost to the insert operations that contribute to the count *#insert* and *LastSize* units to the delete mins that contribute to the count *LastSize* (each such delete-min operation is charged a number of cost units equal to the number of min trees it left behind), then only $u - 1$ of the s cost units remain. Since $u \leq \log_2 n$, and since the number of min trees in a B-heap immediately following a delete-min operation is $\leq \log_2 n$, the amortized cost of a delete-min operation becomes $O(\log_2 n)$.

Since this charging scheme adds at most one unit to the cost of any insert, the amortized cost of an insert becomes $O(1)$. The amortization scheme used does not charge anything extra to a meld. So, the actual and amortized costs of a meld are also $O(1)$. □

From the preceding theorem and the definition of cost amortization, it follows that

the actual cost of any sequence of i inserts, c melds, and dm delete-min operations is $O(i + c + dm \log i)$.

EXERCISES

1. Let S be an initially empty stack. We wish to perform two kinds of operations on S: *add* (x) and *deleteUntil* (x). These are defined as follows:
   (a) *add* (x) ... add the element x to the top of the stack S. This operation takes $O(1)$ time per invocation.
   (b) *deleteUntil* (x) ... delete elements from the top of the stack upto and including the first x encountered. If p elements are deleted, the time taken is $O(p)$.
   Consider any sequence of n stack operations (*add*s and *deleteUntil*s). Show how to amortize the cost of the *add* and *deleteUntil* operations so that the amortized cost of each is $O(1)$. From this, conclude that the time needed to perform any such sequence of operations is $O(n)$.
2. Let x be an unsorted array of n elements. The function *search* (x,n,i,y) searches x for y by examining $x[i], x[i+1], \ldots$, in that order, for the least j such that $x[j] = y$. In case no such j is found, j is set to $n+1$. On termination, *search* sets i to j. Assume that the time required to examine a single element of x is $O(1)$.
   (a) What is the worst case complexity of *search*?
   (b) Suppose that a sequence of m searches is performed beginning with $i = 0$. Use a cost amortization scheme that assigns costs to both elements and search operations. Show that it is always possible to amortize costs so that the amortized cost of each element is $O(1)$ and that of each search is also $O(1)$. From this, conclude that the cost of the sequence of m searches is $O(m + n)$.
3. (a) Into an empty B-heap, insert elements with priorities 20, 10, 5, 18, 6, 12, 14, 4, and 22 (in this order). Show the final B-heap.
   (b) Delete the min element from the final B-heap of part (a). Show the resulting B-heap. Show how you arrived at this final B-heap.
4. Prove that the binomial tree B_k has 2^k nodes, $k \geq 0$.
5. Can all the functions on a B-heap be performed in the same time using singly linked circular lists rather than doubly linked circular lists? Note that we can delete from an arbitrary node x of a singly linked circular list by copy over the data from the next node and then deleting the next node rather than the node x.
6. Compare the performance of leftist trees and B-heaps under the assumption that the only permissible operations are insert and delete min. For this, do the following:
   (a) Create a random list of n elements and a random sequence of insert and delete min operations of length m. The number of delete mins and inserts

should be approximately equal. Initialize a min-leftist tree and a B-heap to contain the n elements in the first random list. Now, measure the time to perform the m operations using the min-leftist tree as well as the B-heap. Divide this time by m to get the average time per operation. Do this for n = 100, 500, 1000, 2000, $\cdots$, 5000. Let m be 5000. Tabulate your computing times.

(b) Based on your experiments, make some statements about the relative merits of the two data structures?

7. Is the height of every tree in a Binomial heap that has n elements O(log n)? If not, what is the worst-case height as a function of n?

9.4 FIBONACCI HEAPS

9.4.1 Definition

There are two varieties of Fibonacci heaps: min and max. A *min Fibonacci heap* is a collection of min trees; a *max Fibonacci heap* is a collection of max trees. We shall explicitly consider min Fibonacci heaps only. These will be referred to as *F-heaps*. B-heaps are a special case of F-heaps. Thus, all the examples of B-heaps in the preceding section are also examples of F-heaps. As a consequence, in this section, we shall refer to these examples as F-heaps.

An F-heap is a data structure that supports the three binomial heap operations: insert, delete min or max, and meld as well as the operations:

(1) *Delete*, delete the element in a specified node. We refer to this delete operation as *arbitrary delete*.

(2) *Decrease key*, decrease the key of a specified node by a given positive amount.

When an F-heap, is used, the *delete* operation takes O(log n) amortized time and the *decrease key* takes O(1) amortized time. The B-heap operations can be performed in the same asymptotic times using an F-heap as they can be using a B-heap.

To represent an F-heap, the B-heap representation is augmented by adding two fields, *parent* and *childCut* to each node. The *parent* field is used to point to the node's parent (if any). The significance of the *childCut* field will be described later. The basic operations: insert, delete min, and meld are performed exactly as for the case of B-heaps. Let us examine the remaining two operations.

9.4.2 Deletion From An F-heap

To delete an arbitrary node b from the F-heap a, we do the following:

(1) If $a = b$, then do a delete min; otherwise do steps 2, 3, and 4 below.

(2) Delete b from the doubly linked list it is in.

(3) Combine the doubly linked list of b's children with the doubly linked list of a's min-tree roots to get a single doubly linked list. Trees of equal degree are not joined together as in a delete min.

(4) Dispose of node b.

For example, if we delete the node containing 12 from the F-heap of Figure 9.6, we get the F-heap of Figure 9.11. The actual cost of an arbitrary delete is O(1) unless the min element is being deleted. In this case the deletion time is the time for a delete min operation.

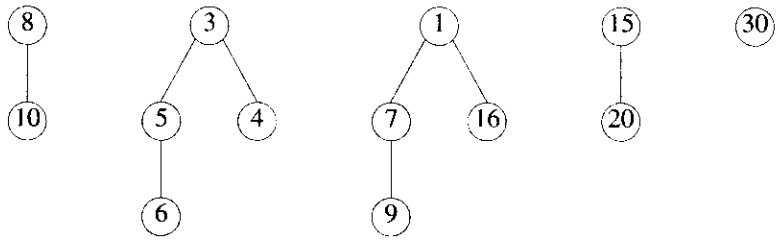


Figure 9.11: F-heap of Figure 9.6 following the deletion of 12

9.4.3 Decrease Key

To decrease the key in node b we do the following:

(1) Reduce the key in b.

(2) If b is not a min-tree root and its key is smaller than that in its parent, then delete b from its doubly linked list and insert it into the doubly linked list of min-tree roots.

(3) Change a to point to b in case the key in b is smaller than that in a.

Suppose we decrease the key 15 in the F-heap of Figure 9.6 by 4. The new value for this key is 11 and the resulting F-heap is shown in Figure 9.12. The cost of performing a decrease key is O(1).

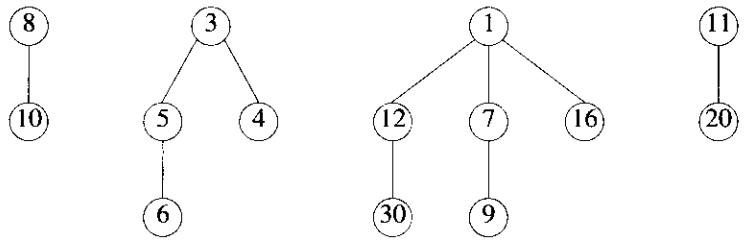


Figure 9.12: F-heap of Figure 9.6 following the reduction of 15 by 4

9.4.4 Cascading Cut

With the addition of the delete and decrease key operations, the min-trees in an F-heap need not be binomial trees. In fact, it is possible to have degree k min-trees with as few as $k+1$ nodes. As a result, the analysis of Theorem 9.1 is no longer valid. The analysis of Theorem 9.1 requires that each min-tree of degree k have an exponential (in k) number of nodes. When decrease key and delete operations are performed as described above, this is no longer true. To ensure that each min-tree of degree k has at least c^k nodes for some c, $c > 1$, each delete and decrease key operation must be followed by a *cascading cut* step. For this, we add the boolean field *childCut* to each node. The value of this field is useful only for nodes that are not a min-tree root. In this case, the *childCut* field of node x has the value *TRUE* iff one of the children of x was cut off (i.e., removed) after the most recent time x was made the child of its current parent. This means that each time two min-trees are joined in a delete min operation, the *childCut* field of the root with the larger key should be set to *FALSE*. Further, whenever a delete or decrease key operation deletes a node q that is not a min-tree root from its doubly linked list (step 2 of delete and decrease key), then the cascading cut step is invoked. During this, we examine the nodes on the path from the parent p of the deleted node q up to the nearest ancestor of the deleted node with *childCut* = *FALSE*. In case there is no such ancestor, then the path goes from p to the root of the min-tree containing p. All non root nodes on this path with *childCut* field *TRUE* are deleted from their respective doubly linked lists and added to the doubly linked list of min-tree root nodes of the F-heap. If the path has a node with *childCut* field *FALSE*, this field is changed to *TRUE*.

Figure 9.13 gives an example of a cascading cut. Figure 9.13(a) is the min-tree containing 14 before a decrease key operation that reduces this key by 4. The *childCut* fields are shown only for the nodes on the path from the parent of 14 to its nearest ancestor with *childCut* = *FALSE*. A *TRUE* value is indicated by T. During the decrease key operation, the min-tree with root 14 is deleted from the min-tree of Figure 9.13(a) and

becomes a min-tree of the F-heap. Its root now has key 10. This is the first min-tree of Figure 9.13(b). During the cascading cut, the min-trees with roots 12, 10, 8, and 6 are cut off from the min tree with root 2. Thus the single min-tree of Figure 9.13(a) becomes six min-trees of the resulting F-heap. The *childCut* value of 4 becomes *TRUE*. All other *childCut* values are unchanged.

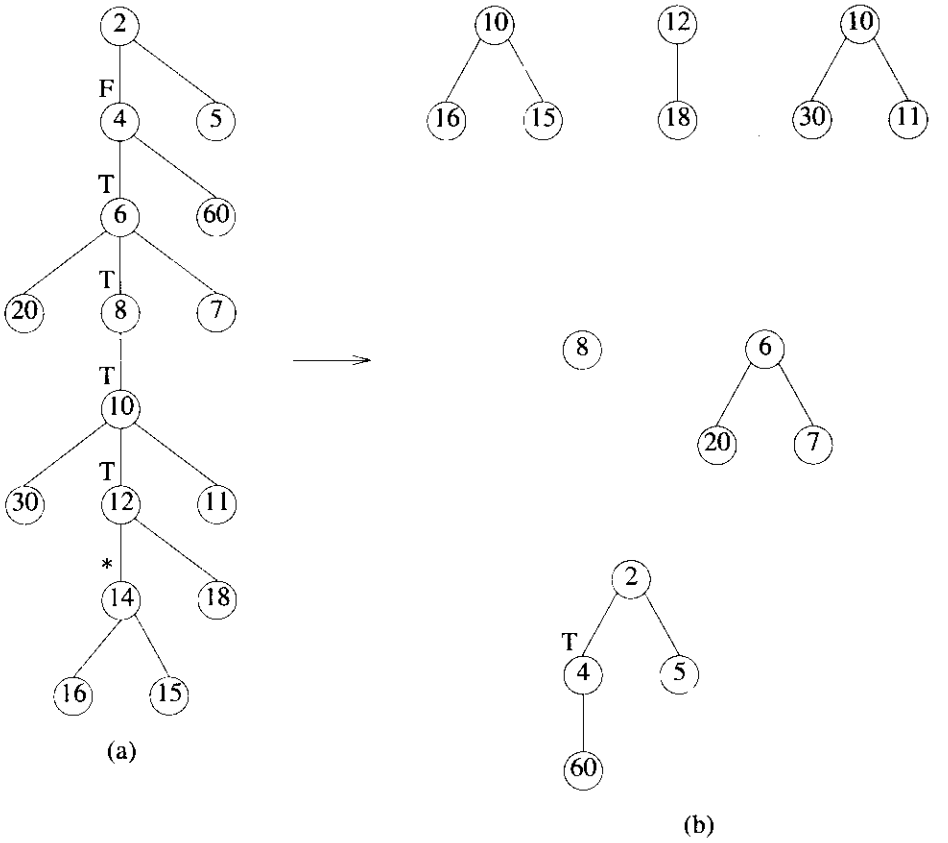


Figure 9.13: A cascading cut following a decrease of key 14 by 4

9.4.5 Analysis

Lemma 9.4: Let a be an F-heap with n elements that results from a sequence of insert, meld, delete min, delete, and decrease key operations performed on initially empty F-heaps.

(a) Let b be any node in any of the min-trees of a. The degree of b is at most $\log_\phi m$, where $\phi = (1+\sqrt{5})/2$ and m is the number of elements in the subtree with root b.

(b) $MAX\_DEGREE \leq \lfloor \log_\phi n \rfloor$ and the actual cost of a delete min is $O(\log n + s)$.

Proof: We shall prove (a) by induction on the degree of b. Let N_i be the minimum number of elements in the subtree with root b when b has degree i. We see that $N_0 = 1$ and $N_1 = 2$. So, the inequality of (a) holds for degrees 0 and 1. For $i > 1$, let $c_1, \cdots, c_i$ be the i children of b. Assume that c_j was made a child of b before c_{j+1}, $j < i$. Hence, when c_k, $k \leq i$ was made a child of b, the degree of b was at least $k-1$. The only F-heap operation that makes one node a child of another is delete min. Here, during a join min-tree step, one min-tree is made a sub tree of another min-tree of equal degree. Hence, at the time of joining, the degree of c_k must have been equal to that of b. Subsequent to joining, its degree can decrease as a result of a delete or decrease key operation. However, following such a join, the degree of c_k can decrease by at most one as an attempt to cut off a second child of c_k results in a cascading cut at c_k. Such a cut causes c_k to become the root of a min-tree of the F-heap. Hence, the degree, d_k, of c_k is at least $\max\{0, k-2\}$. So, the number of elements in c_k is at least N_{d_k}. This implies that

$$N_i = N_0 + \sum_{k=0}^{i-2} N_k + 1 = \sum_{k=0}^{i-2} N_k + 2$$

One may show (see the exercises) that the Fibonacci numbers satisfy the equality

$$F_h = \sum_{k=0}^{h-2} F_k + 1, h > 1, F_0 = 0, \text{ and } F_1 = 1$$

From this we may obtain the equality $N_i = F_{i+2}$, $i \geq 0$. Further, since $F_{i+2} \geq \phi^i$, $N_i \geq \phi^i$. Hence, $i \leq \log_\phi m$.

(b) is a direct consequence of (a). □

Theorem 9.2: If a sequence of n insert, meld, delete min, delete, and decrease key operations is performed on an initially empty F-heap, then we can amortize costs such that the amortized time complexity of each insert, meld, and decrease key operation is $O(1)$ and that of each delete min and delete is $O(\log n)$. The total time complexity of the entire sequence is the sum of the amortized complexities of the individual operations in

the sequence.

Proof: The proof is similar to that of Theorem 9.1. The definition of *#insert* is unchanged. However, that of *lastSize* is augmented by requiring that following each delete and decrease key *lastSize* be changed by the net change in the number of min-trees in the F-heap (in the example of Figure 9.13 *lastSize* is increased by 5). With this modification, we see that at the time of a delete min operation s = *#insert* + *lastSize* + $u - 1$. *#insert* units of cost may me charged, one each, to the *#insert* insert operations that contribute to this count and *lastSize* units may be charged to the delete min, delete, and decrease key operations that contribute to this count. This results in an additional charge of at most $\log_\phi n$ to each contributing delete min and delete operation and of one to each contributing decrease key operation. As a result, the amortized cost of a delete min is $O(\log n)$.

Since the total number of cascading cuts is limited by the total number of deletes and decrease key operations (as these are the only operations that can set *childCut* to *TRUE*), the cost of these cuts may be amortized over the delete and decrease key operations by adding one to their amortized costs. The amortized cost of deleting an element other than the min element becomes $O(\log n)$ as its actual cost is $O(1)$ (excluding the cost of the cascading cut sequence that may be performed); at most one unit is charged to it from the amortization of all the cascading cuts; and at most $\log_\phi n$ units are charged to it from a delete min.

The amortized cost of a decrease key operation is $O(1)$ as its actual cost is $O(1)$ (excluding the cost of the ensuing cascading cut); at most one unit is charged to it from the amortization of all cascading cuts; and at most one unit is charged from a delete min.

The amortized cost of an insert is $O(1)$ as its actual cost is one and at most one cost unit is charged to it from a delete min. Since the amortization scheme transfers no charge to a meld, its actual and amortized costs are the same. This cost is $O(1)$. □

From the preceding theorem, it follows that the complexity of any sequence of F-heap operations is $O(i + c + dk + (dm + d)\log i)$ where i, c, dk, dm, and d are, respectively, the number of insert, meld, decrease key, delete min, and delete operations in the sequence.

9.4.6 Application to the Shortest-Paths Problem

We conclude this section on F-heaps by considering their application to the single-source/all-destinations algorithm of Chapter 6. Let S be the set of vertices to which a shortest path has been found and let $dist(i)$ be the length of a shortest path from the source vertex to vertex i, $i \in S$, that goes through only vertices in S. On each iteration of the shortest-path algorithm, we need to determine an i, $i \in \bar{S}$, such that $dist(i)$ is minimum and add this i to S. This corresponds to a delete min operation on $\bar{S}$. Further, the *dist* values of the remaining vertices in $\bar{S}$ may decrease. This corresponds to a

decrease-key operation on each of the affected vertices. The total number of decrease-key operations is bounded by the number of edges in the graph, and the number of delete-min operations is $n - 2$. S begins with $n - 1$ vertices. If we implement S as an F-heap using *dist* as the key, then $n - 1$ inserts are needed to initialize the F-heap. Additionally, $n - 2$ delete-min operations and at most e decrease-key operations are needed. The total time for all these operations is the sum of the amortized costs for each. This is $O(n \log n + e)$. The remainder of the algorithm takes $O(n)$ time. Hence if an F-heap is used to represent S, the complexity of the shortest-path algorithm becomes $O(n \log n + e)$. This is an asymptotic improvement over the implementation discussed in Chapter 6 if the graph does not have $\Omega(n^2)$ edges. If this single-source algorithm is used n times, once with each of the n vertices in the graph as the source, then we can find a shortest path between every pair of vertices in $O(n^2 \log n + ne)$ time. Once again, this represents an asymptotic improvement over the $O(n^3)$ dynamic programming algorithm of Chapter 6 for graphs that do not have $\Omega(n^2)$ edges. It is interesting to note that $O(n \log n + e)$ is the best possible implementation of the single-source algorithm of Chapter 6, as the algorithm must examine each edge and may be used to sort n numbers (which takes $O(n \log n)$ time).

EXERCISES

1. Prove that if we start with empty F-heaps and perform only the operations insert, meld, and delete min, then all min-trees in the F-heaps are binomial trees.
2. Can all the functions on an F-heap be performed in the same time using singly linked circular lists rather than doubly linked circular lists? Note that we can delete from an arbitrary node x of a singly linked circular list by copy over the data from the next node and then deleting the next node rather than the node x.
3. Show that if we start with empty F-heaps and do not perform cascading cuts, then it is possible for a sequence of F-heap operations to result in degree k min-trees that have only $k + 1$ nodes, $k \geq 1$.
4. Is the height of every tree in a Fibonacci heap that has n elements $O(\log n)$? If not, what is the worst-case height as a function of n?
5. Suppose we change the rule for a cascading cut so that such a cut is performed only when a node loses a third child rather than when it loses a second child. For this, the *childCut* field is changed so that it can have the values 0, 1, and 2. When a node acquires a new parent, its *childCut* field is set to 1. Each time a node has a child cut off (during a delete or decrease key operation), its *childCut* field is increased by one (unless this field is already two). In case the *childCut* field is already two, a cascading cut is performed.
   (a) Obtain a recurrence equation for N_i, the minimum number of nodes in a min-tree with degree i. Assume that we start with an empty F-heap and that all operations (except cascading cut) are performed as described in the text.

Cascading cuts are performed as described above.

(b) Solve the recurrence of part (a) to obtain a lower bound on N_i.

(c) Does the modified rule for cascading cuts ensure that the minimum number of nodes in any min-tree of degree i is exponential in i?

(d) For the new cascading cut rule, can you establish the same amortized complexities as for the original rule? Prove the correctness of your answer.

(e) Answer parts (c) and (d) under the assumtion that cascading cuts are performed only after k children of a node have been cut off. Here, k is a fixed constant ($k = 2$ for the rule used in the text and $k = 3$ for the rule used earlier in this exercise).

(f) How do you expect the performance of F-heaps to change as larger values of k (see part (e)) are used?

6. Write C functions to do the following:

(a) Create an empty F-heap

(b) Insert element x into an F-heap

(c) Perform a delete min from an F-heap. The deleted element is to be returned to the invoking function.

(d) Delete the element in node b of an F-heap a. The deleted element is to be returned to the invoking function.

(e) Decrease the key in the node b of an F-heap a by some positive amount c.

Note that all operations must leave behind properly structured F-heaps. Your functions for (d) and (e) must perform cascading cuts. Test the correctness of your procedures by running them on a computer using suitable test data.

7. For the Fibonacci numbers F_k and the numbers N_i of Lemma 9.4, prove the following:

(a) $F_h = \sum_{k=0}^{h-2} F_k + 1, h > 1$

(b) Use (a) to show that $N_i = F_{i+2}, i \geq 0$.

(c) Use the equality $F_k = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^k - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^k, k \geq 0$ to show that $F_{k+2} \geq \phi^k, k \geq 0$, where $\phi = (1+\sqrt{5})/2$.

8. Implement the single source shortest path algorithm of Chapter 6 using the data structures recommended there as well as using F-heaps. However, use adjacency lists rather than an adjacency matrix. Generate 10 connected undirected graphs with different edge densities (say 10%, 20%, $\cdots$, 100% of maximum) for each of the cases $n = 100, 200, \cdots, 500$. Assign random costs to the edges (use a uniform random number generator in the range [1, 1000]). Measure the run times of the

two implementations of the shortest path algorithms. Plot the average times for each n.

9.5 PAIRING HEAPS

9.5.1 Definition

The pairing heap supports the same operations as supported by the Fibonacci heap. Pairing heaps come in two varieties—min pairing heaps and max pairing heaps. Min pairing heaps are used when we wish to represent a min priority queue, and max pairing heaps are used for max priority queues. In keeping with our discussion of Fibonacci heaps, we explicitly discuss min pairing heaps only. Max pairing heaps are analogous. Figure 9.14 compares the actual and amortized complexities of the Fibonacci and pairing heap operations.

Operation	Fibonacci Heap		Pairing Heap	
	Actual	Amortized	Actual	Amortized
getMin	O(1)	O(1)	O(1)	O(1)
insert	O(1)	O(1)	O(1)	O(1)
deleteMin	O(n)	O(log n)	O(n)	O(log n)
meld	O(1)	O(1)	O(1)	O(log n)
delete	O(n)	O(log n)	O(n)	O(log n)
decreaseKey	O(n)	O(1)	O(1)	O(log n)

Figure 9.14: Complexity of Fibonacci and pairing heap operations

Although the amortized complexities given in Figure 9.14 for pairing heap operations are not known to be tight (i.e., no one knows of an operation sequence whose run time actually grows logarithmically with the number of decrease key operations (say)), it is known that the amortized complexity of the decrease key operation is $\Omega(\log\log n)$ (see the section titled References and Selected Readings at the end of this chapter).

Although the amortized complexity is better when a Fibonacci heap is used rather than when a pairing heap is used, extensive experimental studies employing these structures in the implementation of Dijkstra's shortest paths algorithm (Section 6.4.1) and Prim's minimum cost spanning tree algorithm (Section 6.3.2) indicate that pairing heaps actually outperform Fibonacci heaps.

Definition: A *min pairing heap* is a min tree in which the operations are performed in a manner to be specified later.

Figure 9.15 shows four example min pairing heaps. Notice that a pairing heap is a single tree, which need not be a binary tree. The min element is in the root of this tree and hence this element may be found in O(1) time.

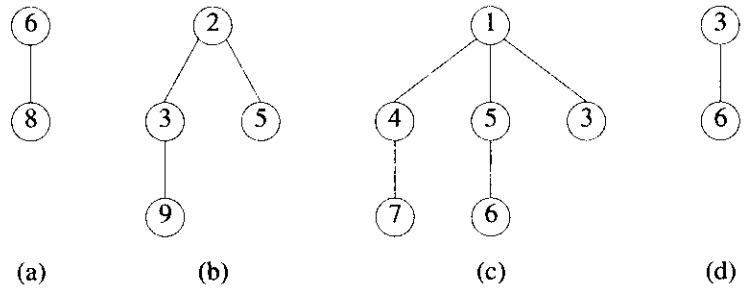


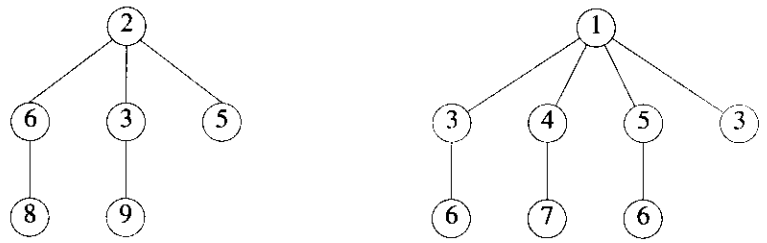
Figure 9.15: Example min pairing heaps

9.5.2 Meld and Insert

Two min pairing heaps may be melded into a single min pairing heap by performing a *compare-link* operation. In a compare-link, the roots of the two min trees are compared and the min tree that has the larger root is made the leftmost subtree of the other tree (ties are broken arbitrarily).

To meld the min trees of Figures 9.15 (a) and (b), we compare the two roots. Since tree (a) has the larger root, this tree becomes the leftmost subtree of tree (b). Figure 9.16 (a) is the resulting pairing heap. Figure 9.16 (b) shows the result of melding the pairing heaps of Figures 9.15 (c) and (d). When we meld the pairing heaps of Figures 9.16 (a) and (b), the result is the pairing heap of Figure 9.17.

To insert an element x into a pairing heap p, we first create a pairing heap q with the single element x, and then meld the two pairing heaps p and q.



(a) Meld of Figures 9.15 (a) and (b)

(b) Meld of Figures 9.15 (c) and (d)

Figure 9.16: Melding pairing heaps

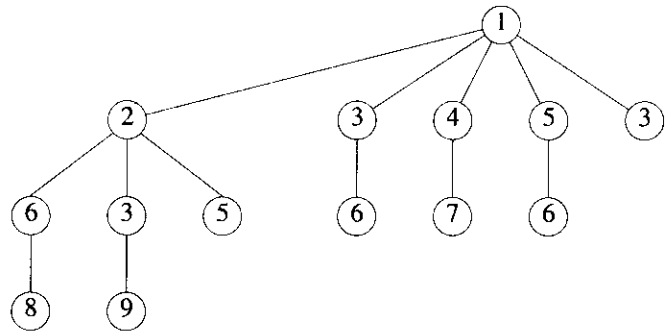


Figure 9.17: Meld of Figures 9.16 (a) and (b)

9.5.3 Decrease Key

Suppose we decrease the key/priority of the element in node N. When N is the root or when the new key in N is greater than or equal to that in its parent, no additional work is to be done. However, when the new key in N is less than that in its parent, the min tree property is violated and corrective action is to be taken. For example, if the key in the root of the tree of Figure 9.15 (c) is decreased from 1 to 0, or when the key in the leftmost child of the root of Figure 9.15 (c) is decreased from 4 to 2 no additional work is necessary. When the key in the leftmost child of the root of Figure 9.15 (c) is decreased

from 4 to 0 the new value is less than that in the root (see Figure 9.18 (a)) and corrective action is needed.

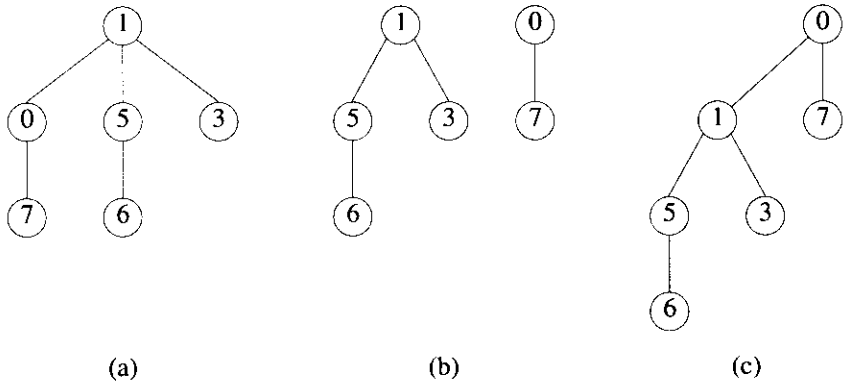


Figure 9.18: Decreasing a key

Since pairing heaps are normally not implemented with a parent pointer, it is difficult to determine whether or not corrective action is needed following a key reduction. Therefore, corrective action is taken regardless of whether or not it is needed except when N is the tree root. The corrective action consists of the following steps:

Step 1: Remove the subtree with root N from the tree. This results in two min trees.

Step 2: Meld the two min trees together.

Figure 9.18 (b) shows the two min trees following Step 1, and Figure 9.18 (c) shows the result following Step 2.

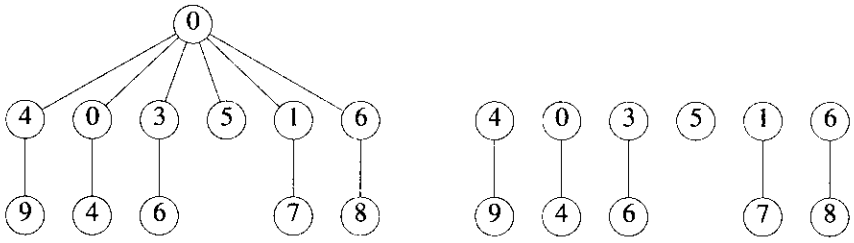
9.5.4 Delete Min

The min element is in the root of the tree. So, to delete the min element, we first delete the root node. When the root is deleted, we are left with zero or more min trees (i.e., the subtrees of the deleted root). When the number of remaining min trees is two or more, these min trees must be melded into a single min tree. In *two pass pairing heaps*, this melding is done as follows:

Step 1: Make a left to right pass over the trees, melding pairs of trees.

Step 2: Start with the rightmost tree and meld the remaining trees (right to left) into this tree one at a time.

Consider the min pairing heap of Figure 9.19 (a). When the root is removed, we get the collection of 6 min trees shown in Figure 9.19 (b).



(a) A min pairing heap (b) After deletion of root

Figure 9.19: Deleting the min element

In the left to right pass of Step 1, we first meld the trees with roots 4 and 0. Next, the trees with roots 3 and 5 are melded. Finally, the trees with roots 1 and 6 are melded. Figure 9.20 shows the resulting three min trees.

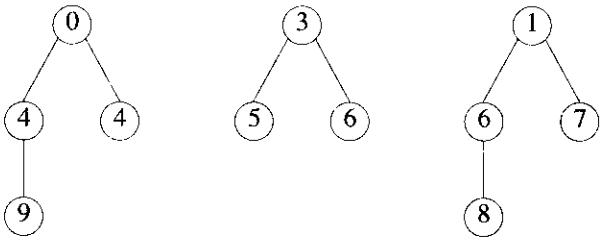


Figure 9.20: Trees following first pass

In Step 2 (which is a right to left pass), the two rightmost trees of Figure 9.20 are first melded to get the tree of Figure 9.21 (a).

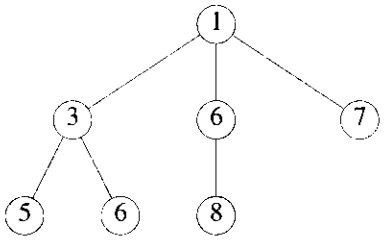


Figure 9.21: First stage of second pass

Then the tree of Figure 9.20 with root 0 is melded with the tree of Figure 9.21 to get the final min tree, which is shown in Figure 9.22.

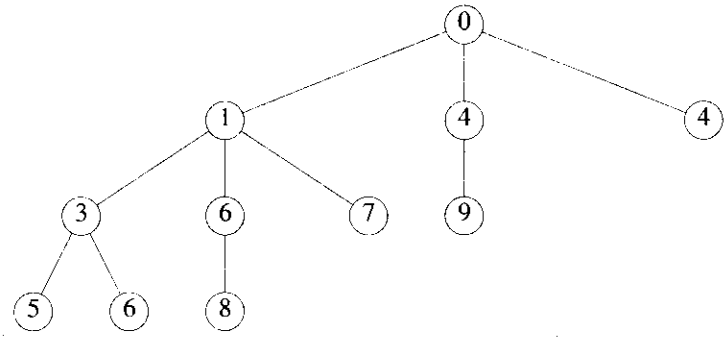


Figure 9.22: Final min pairing heaping following a delete min

Note that if the original pairing heap had 8 subtrees, then following the left to right melding pass we would be left with 4 min trees. In the right to left pass, we would first meld trees 3 and 4 to get tree 5. Then trees 2 and 5 would be melded to get tree 6. Finally, we would meld trees 1 and 6.

In *multi pass pairing heaps*, the min trees that remain following the removal of the root are melded into a single min tree as follows:

Step 1: Put the min trees onto a FIFO queue.

Step 2: Extract two trees from the front of the queue, meld them and put the resulting tree at the end of the queue. Repeat this step until only one tree remains.

Consider the six trees of Figure 9.19 (b) that result when the root of Figure 9.19 (a) is deleted. First, we meld the trees with roots 4 and 0 and put the resulting min tree at the end of the queue. Next, the trees with roots 3 and 5 are melded and the resulting min tree is put at the end of the queue. And then, the trees with roots 1 and 6 are melded and the resulting min tree added to the queue end. The queue now contains the three min trees shown in Figure 9.20. Next, the min trees with roots 0 and 3 are melded and the result put at the end of the queue. We are now left with the two min trees shown in Figure 9.23.

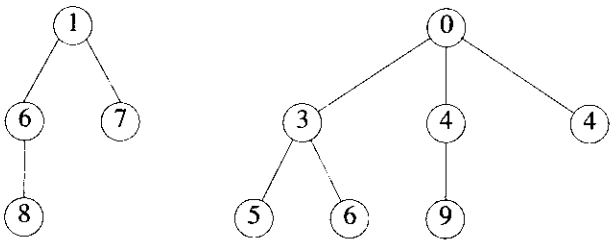


Figure 9.23: Next to last state in multi pass delete

Finally, the two min trees of Figure 9.23 are melded to get the min tree of Figure 9.24.

9.5.5 Arbitrary Delete

Deletion from an arbitrary node N is handled as a delete-min operation when N is the root of the pairing heap. When N is not the tree root, the deletion is done as follows:

Step 1: Detach the subtree with root N from the tree.

Step 2: Delete node N and meld its subtrees into a single min tree using the two pass scheme if we are implementing a two pass pairing heap or the multi pass scheme if we are implementing a multi pass pairing heap.

Step 3: Meld the min trees from Steps 1 and 2 into a single min tree.

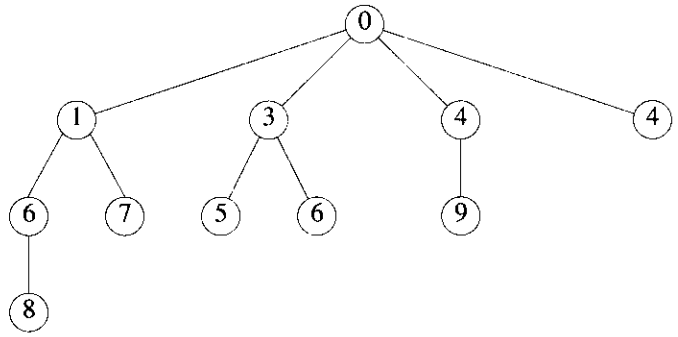


Figure 9.24: Result of multi pass delete min

9.5.6 Implementation Considerations

Although we can implement a pairing heap using nodes that have a variable number of children fields, such an implementation is expensive because of the need to dynamically increase the number of children fields as needed. An efficient implementation results when we use the binary tree representation of a tree (see Section 5.1.2.2). Siblings in the original min tree are linked together using a doubly linked list. In addition to a *data* field, each node has the three pointer fields *previous*, *next*, and *child*. The leftmost node in a doubly linked list of siblings uses its *previous* pointer to point to its parent. A leftmost child satisfies the property $x \rightarrow previous \rightarrow child = x$. The doubly linked list makes it is possible to remove an arbitrary element (as is required by the *delete* and *decrease key* operations) in O(1) time.

9.5.7 Complexity

You can verify that using the described binary tree representation, all pairing heap operations (other than *delete* and *delete min* can be done in O(1) time. The complexity of the *delete* and *delete min* operations is $O(n)$, because the number of subtrees that have to be melded following the removal of a node is $O(n)$.

The amortized complexity of the pairing heap operations is established in the paper by Fredman et al. cited in the References and Selected Readings section. Experimental studies conducted by Stasko and Vitter (see their paper that is cited in the References and Selected Readings section) establish the superiority of two pass pairing heaps

over multipass pairing heaps.

EXERCISES

1. (a) Into an empty two pass min pairing heap, insert elements with priorities 20, 10, 5, 18, 6, 12, 14, 9, 8 and 22 (in this order). Show the min pairing heap following each insert.

   (b) Delete the min element from the final min pairing heap of part (a). Show the resulting pairing heap.

2. (a) Into an empty multi pass min pairing heap, insert elements with priorities 20, 10, 5, 18, 6, 12, 14, 9, 8 and 22 (in this order). Show the min pairing heap following each insert.

   (b) Delete the min element from the final min pairing heap of part (a). Show the resulting pairing heap.

3. Fully code and test the class *MultiPassPairingHeap*, which impements a multi pass min pairing heap. Your class must include the functions *GetMin*, *Insert*, *DeleteMin*, *Meld*, *Delete* and *DecreaseKey*. The function *Insert* should return the node into which the new element was inserted. This returned information can later be used as an input to *Delete* and *DecreaseKey*.

4. What are the worst-case height and degree of a pairing heap that has n elements? Show how you arrived at your answer.

5. Define a *one pass pairing heap* as an adaptation of a two pass pairing heap in which Step 1 (Make a left to right pass over the trees, melding pairs of trees.) is eliminated. Show that the amortized cost of either insert or delete min must be $\Theta(n)$.

9.6 SYMMETRIC MIN-MAX HEAPS

9.6.1 Definition and Properties

A double-ended priority queue (DEPQ) may be represented using a symmetric min-max heap (SMMH). An *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 N be any node of the SMMH. Let *elements* (N) be the elements in the subtree rooted at N but excluding the element (if any) in N. Assume that *elements* (N)$\neq\phi$. N satisfies the following properties:

Q1: The left child of N has the minimum element in *elements* (N).

Q2: The right child of N (if any) has the maximum element in *elements* (N).

Figure 9.25 shows an example SMMH that has 12 elements. When N denotes the node with 80, *elements* $(N)=\{6,14,30,40\}$; the left child of N has the minimum element 6 in *elements* (N); and the right child of N has the maximum element 40 in *elements* (N). You may verify that every node N of this SMMH satisfies properties Q1 and Q2.

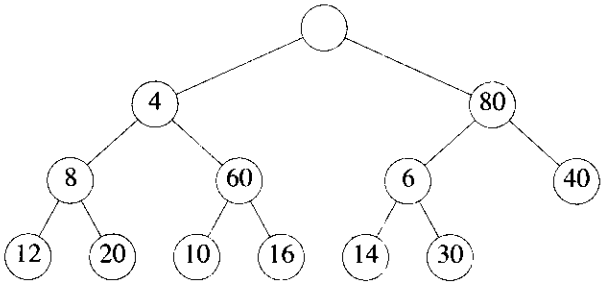


Figure 9.25: A symmetric min-max heap

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: The element in each node is less than or equal to that in its right sibling (if any).

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

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

Properties P2 and P3, respectively, state that the grandchildren of each node M have elements that are greater than or equal to that in the left child of M and less than or equal to that in the right child of M. Hence, P2 and P3 follow from Q1 and Q2, respectively. Notice that if property P1 is satisfied, then at most one of P2 and P3 may be violated at any node N. Using properties P1 through P3 we arrive at simple algorithms to insert and delete elements. These algorithms are simple adaptations of the corresponding algorithms for heaps.

As we shall see, the standard DEPQ operations can be done efficiently using an SMMH.

9.6.2 SMMH Representation

Since an SMMH is a complete binary tree, it is efficiently represented as a one-dimensional array (say h) using the standard mapping of a complete binary tree into an array (Section 5.2.3.1). Position 0 of h is not used and position 1, which represents the root of the complete binary tree, is empty. We use the variable *last* to denote the right-most position of h in which we have stored an element of the SMMH. So, the size (i.e., number of elements) of the SMMH is *last*−1. The variable *arrayLength* keeps track of the current number of positions in the array h.

When $n=1$, the minimum and maximum elements are the same and are in the left child of the root of the SMMH. 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 min and max elements may be found in $O(1)$ time each.

9.6.3 Inserting into an SMMH

The algorithm to insert into an SMMH has three steps.

Step 1: Expand the size of the complete binary tree by 1, creating a new node E for the element x that is to be inserted. This newly created node of the complete binary tree becomes the candidate node to insert the new element x.

Step 2: Verify whether the insertion of x into E would result in a violation of property P1. Note that this violation occurs iff E is a right child of its parent and x is greater than the element in the sibling of E. In case of a P1 violation, the element in the sibling of E is moved to E and E is updated to be the now empty sibling.

Step 3: Perform a bubble-up pass from E up the tree verifying properties P2 and P3. In each round of the bubble-up pass, E moves up the tree by one level. When E is positioned so that the insertion of x into E doesn't result in a violation of either P2 or P3, insert x into E.

Suppose we wish to insert 2 into the SMMH of Figure 9.25. Since an SMMH is a complete binary tree, we must add a new node to the SMMH in the position shown in Figure 9.26; the new node is labeled E. In our example, E will denote an empty node.

If the new element 2 is placed in node E, property P2 is violated as the left child of the grandparent of E has 6. So we move the 6 down to E and move E up one level to obtain the configuration of Figure 9.27.

Now we determine if it is safe to insert the 2 into node E. We first notice that such an insertion cannot result in a violation of property P1, because the previous occupant of node E was greater than 2. For properties P2 and P3, let $N=E$. P3 cannot be violated for this value of N as the previous occupant of this node was greater than 2. So, only P2 can

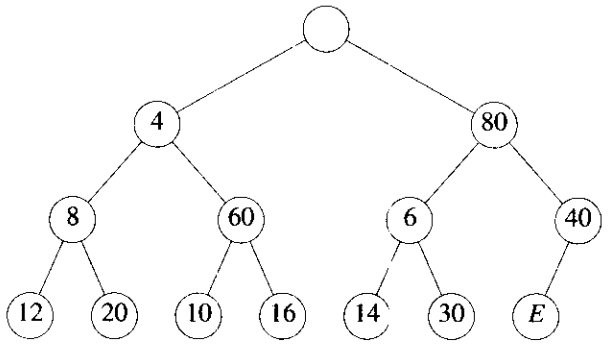


Figure 9.26: The SMMH of Figure 9.25 with a node added

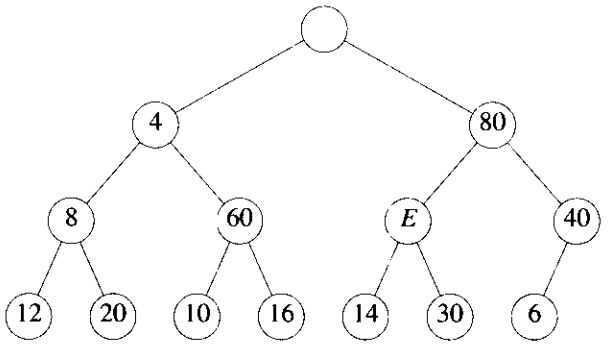


Figure 9.27: The SMMH of Figure 9.26 with 6 moved down

be violated. Checking P2 with N=E, we see that P2 will be violated if we insert x=2 into E, because the left child of the grandparent of E has the element 4. So we move the 4 down to E and move E up one level to the node that previously contained the 4. Figure 9.28 shows the resulting configuration.

For the configuration of Figure 9.28 we see that placing 2 into node E cannot violate property P1, because the previous occupant of node E was greater than 2. Also properties P2 and P3 cannot be violated, because node E has no grandparent. So we

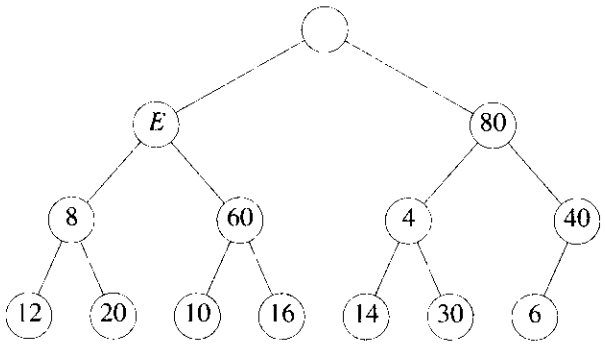


Figure 9.28: The SMMH of Figure 9.27 with 4 moved down

insert 2 into node E and obtain Figure 9.29.

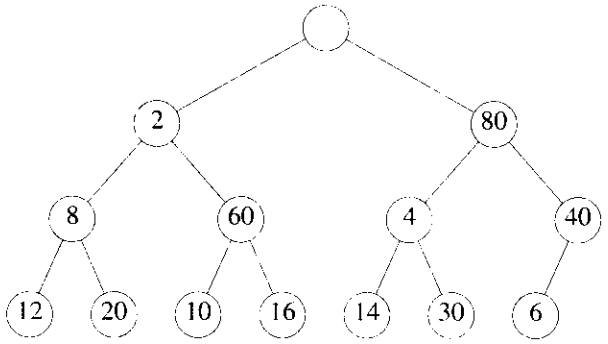


Figure 9.29: The SMMH of Figure 9.28 with 2 inserted

Let us now insert 50 into the SMMH of Figure 9.29. Since an SMMH is a complete binary tree, the new node must be positioned as in Figure 9.30.

Since E is the right child of its parent, we first check P1 at node E. If the new element (in this case 50) is smaller than that in the left sibling of E, 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. We see that placing 50 into E would violate P3. So the element 40 in the right

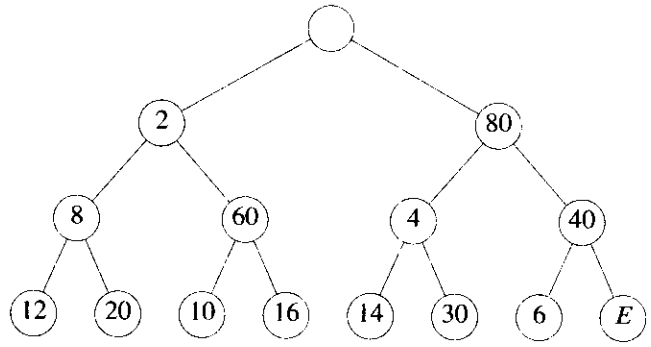


Figure 9.30: The SMMH of Figure 9.29 with a node added

child of the grandparent of E is moved down to node E. Figure 9.31 shows the resulting configuration. Placing 50 into node E of Figure 9.31 cannot create a P1 violation because the previous occupant of node E was smaller. A P2 violation isn't possible either. So only P3 needs to be checked at E. Since there is no P3 violation at E, 50 is placed into E.

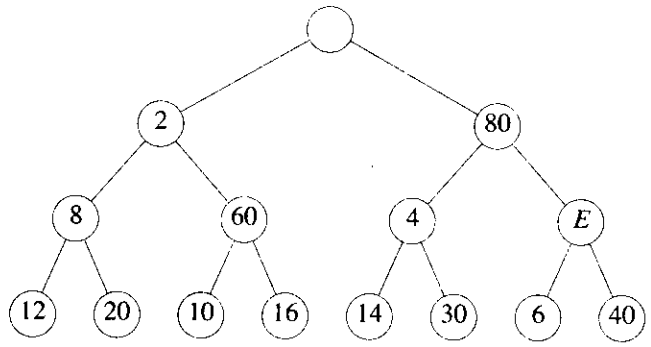


Figure 9.31: The SMMH of Figure 9.30 with 40 moved down

Program 9.5 gives the C code for the insert operation; the variable *currentNode* refers to the empty node E of our example. This code assumes that the SMMH state

```
void insert(int x)
{/* insert x into the SMMH */
   int currentNode, done, gp, lcgp, rcgp;

   /* increase array length if necessary */
   if (last == arrayLength - 1)
   {/* double array length
      REALLOC(h, 2 * arrayLength *sizeof(*h));
      arrayLength *= 2;
   }
   /* find place for x */
   /* currentNode starts at new leaf and moves up tree */
   currentNode = ++last;
   if (last % 2 == 1 && x < h[last - 1])
   {/* left sibling must be smaller, P1 */
      h[last] = h[last - 1]; currentNode--;
   }
   done = FALSE;
   while (!done && currentNode >= 4)
   {/* currentNode has a grandparent
      gp = currentNode / 4;     /* grandparent */
      lcgp = 2 * gp;            /* left child of gp */
      rcgp = lcgp + 1;          /* right child of gp */
      if (x < h[lcgp])
      {/* P2 is violated */
         h[currentNode] = h[lcgp]; currentNode = lcgp;
      }
      else if (x > h[rcgp])
           {/* P3 is violated */
              h[currentNode] = h[rcgp];
              currentNode = rcgp;
           }
           else done = TRUE; /* neither P2 nor P3 violated */
   }
   h[currentNode] = x;
}
```

Program 9.5: Insertion into a symmetric min-max heap

variables h, *arrayLength*, and *last* are global. For simplicity, we assume that the data-type of the SMMH elements is **int**. Since the height of a complete binary tree is $O(\log n)$ and Program 9.5 does $O(1)$ work at each level of the SMMH, the complexity of the insert function is $O(\log n)$.

9.6.4 Deleting from an SMMH

The algorithm to delete either the min or max element is an adaptation of the trickle-down algorithm used to delete an element from a min or a max heap. We consider only the case when the minimum element is to be deleted. If the SMMH is empty, the deletion cannot be performed. So, assume we have a non-empty SMMH. The minimum element is in $h[2]$. If $last=2$, the SMMH becomes empty following the deletion. Assume that $last \neq 2$. Let $x=h[last]$ and decrement *last* by 1. To complete the deletion, we must reinsert x into an SMMH whose $h[2]$ node is empty. Let E denote the empty node. We follow a path from E down the tree, as in the delete algorithm for a min or max heap, verifying properties P1 and P2 until we reach a suitable node into which x may be inserted. In the case of a delete-min operation, the trickle-down process cannot cause a P3 violation. So, we don't explicitly verify P3.

Consider the SMMH of Figure 9.31 with 50 in the node labeled E. A delete min results in the removal of 2 from the left child of the root (i.e., $h[2]$) and the removal of the last node (i.e., the one with 40) from the SMMH. So, $x=40$ and we have the configuration shown in Figure 9.32. Since $h[3]$ has the maximum element, P1 cannot be violated at E. Further, since E is the left child of its parent, no P3 violations can result from inserting x into E. So we need only concern ourselves with P2 violations. To detect such a violation, we determine the smaller of the left child of E and the left child of E's right sibling. For our example, the smaller of 8 and 4 is determined. This smaller element 4 is, by definition of an SMMH, the smallest element in the SMMH. Since $4<x=40$, inserting x into E would result in a P2 violation. To avoid this, we move the 4 into node E and the node previously occupied by 4 becomes E (see Figure 9.33). Notice that if $4 \geq x$, inserting x into E would result in a properly structured SMMH.

Now the new E becomes the candidate node for the insertion of x. First, we check for a possible P1 violation that may result from such an insertion. Since $x=40<50$, no P1 violation results. Then we check for a P2 violation. The left children of E and its sibling are 14 and 6. The smaller child, 6, is smaller than x. So, x cannot be inserted into E. Rather, we swap E and 6 to get the configuration of Figure 9.34.

We now check the P1 property at the new E. Since E doesn't have a right sibling, there is no P1 violation. We proceed to check the P2 property. Since E has no children, a P2 violation isn't possible either. So, x is inserted into E. Let's consider another delete-min operation. This time, we delete the minimum element from the SMMH of Figure 9.34 (recall that the node labeled E contains 40). The min element 4 is removed from $h[2]$ and the last element, 40, is removed from the SMMH and placed in x. Figure 9.35

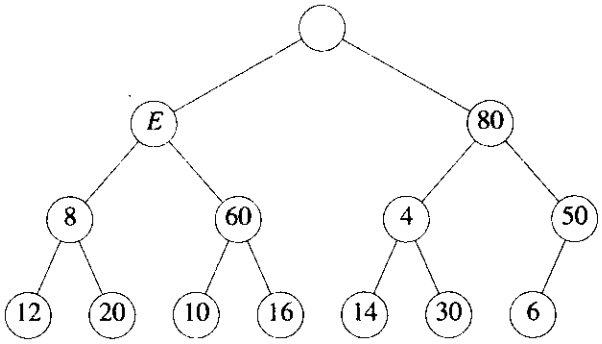


Figure 9.32: The SMMH of Figure 9.31 with 2 deleted

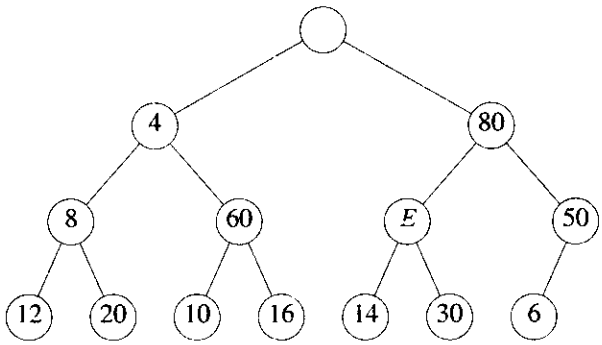
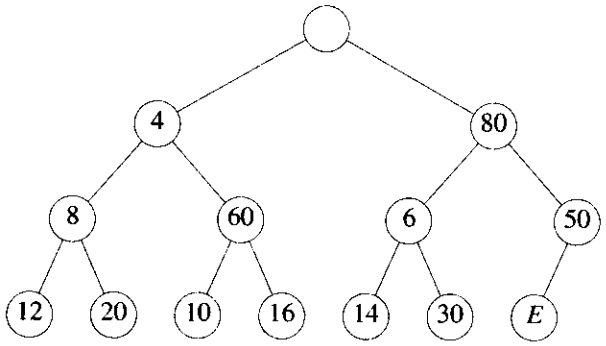


Figure 9.33: The SMMH of Figure 9.32 with E and 4 interchanged

shows the resulting configuration.

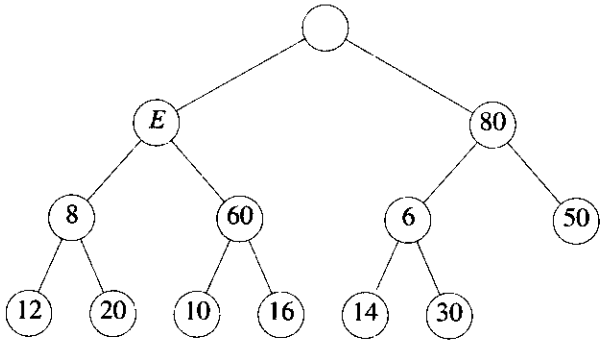
As before, a P1 violation isn't possible at $h[2]$. The smaller of the left children of E and its sibling is 6. Since $6<x=40$, we interchange 6 and E to get Figure 9.36.

Next, we check for a possible P1 violation at the new E. Since the sibling of E is 50 and $x=40 \leq 50$, no P1 violation is detected. The smaller left child of E and its sibling



x=40

Figure 9.34: The SMMH of Figure 9.33 with E and 6 interchanged



x=40

Figure 9.35: First step of another delete min

is 14 (actually, the sibling doesn't have a left child, so we just use the left child of E), which is $<x$. So, we swap E and 14 to get Figure 9.37.

Since there is a P1 violation at the new E, we swap x and 30 to get Figure 9.38 and proceed to check for a P2 violation at the new E. As there is no P2 violation here, x=30 is inserted into E.

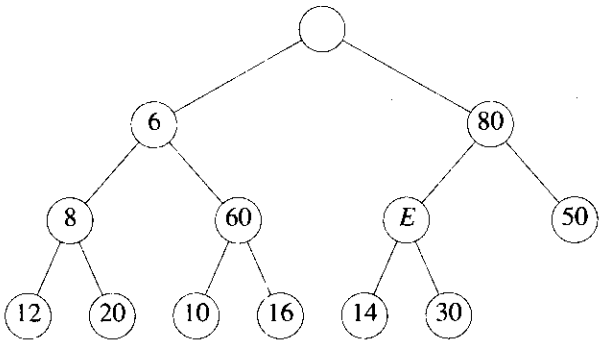


Figure 9.36: The SMMH of Figure 9.35 with E and 6 interchanged

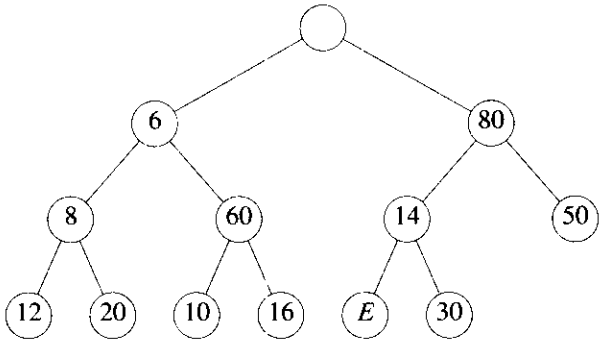
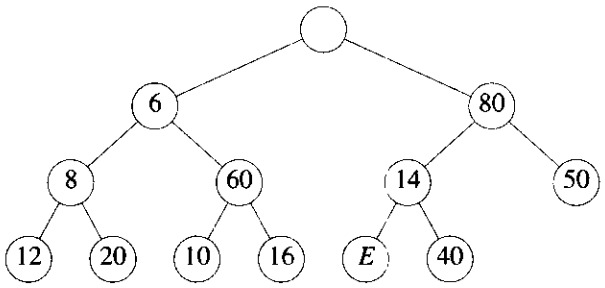


Figure 9.37: The SMMH of Figure 9.36 with E and 14 interchanged

We leave the development of the code for the delete operations as an exercise. However, you should note that these operations spend $O(1)$ time per level during the trickle-down pass. So, their complexity is $O(\log n)$.



x=30

Figure 9.38: The SMMH of Figure 9.37 with x and 30 interchanged

EXERCISES

1. Show that every complete binary tree with an empty root and one element in every other node is an SMMH iff P1 through P3 are true.
2. Start with an empty SMMH and insert the elements 20, 10, 40, 3, 2, 7, 60, 1 and 80 (in this order) using the insertion algorithm developed in this section. Draw the SMMH following each insert.
3. Perform 3 delete-min operations on the SMMH of Figure 9.38 with 30 in the node E. Use the delete min strategy described in this section. Draw the SMMH following each delete min.
4. Perform 4 delete max operations on the SMMH of Figure 9.38 with 30 in the node E. Adapt the delete min strategy of this section to the delete max operation. Draw the SMMH following each delete max operation.
5. Develop the code for all SMMH operations. Test all functions using your own test data.

9.7 INTERVAL HEAPS

9.7.1 Definition and Properties

Like an SMMH, an interval heap is a heap inspired data structure that may be used to represent a DEPQ. 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 a node be a and b,

where $a \leq b$. We say that the node represents the closed interval $[a,b]$. a is the left end point of the node's interval 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 9.39 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.

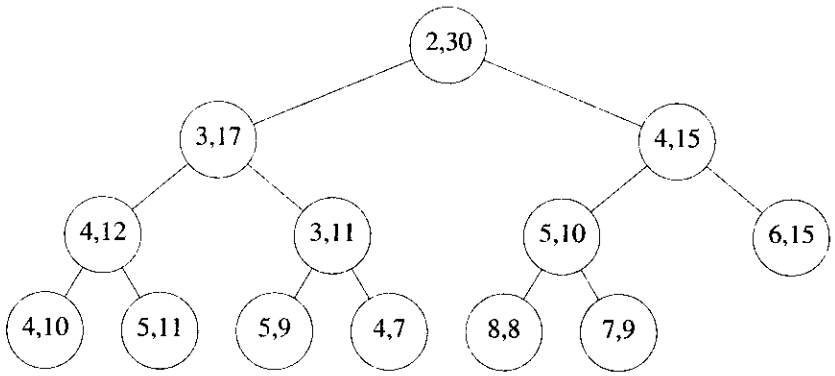


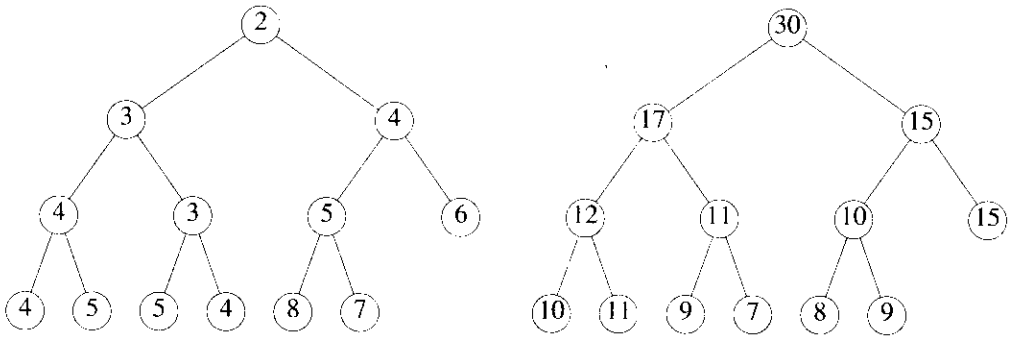
Figure 9.39: An interval heap

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 9.40 shows the min and max heaps defined by the interval heap of Figure 9.39.

(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 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.



(a) min heap (b) max heap

Figure 9.40: Min and max heaps embedded in Figure 9.39

(4) The height of an interval heap with n elements is $\Theta(\log n)$.

9.7.2 Inserting into an Interval Heap

Suppose we are to insert an element into the interval heap of Figure 9.39. 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 9.41.

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 9.39, this element is put into the node A shown in Figure 9.41. 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 no left end point. Figure 9.42 shows the resulting interval heap.

To insert the element 40 into the interval heap of Figure 9.39, we follow a path from node A (see Figure 9.41) towards the root, moving right end points down until we

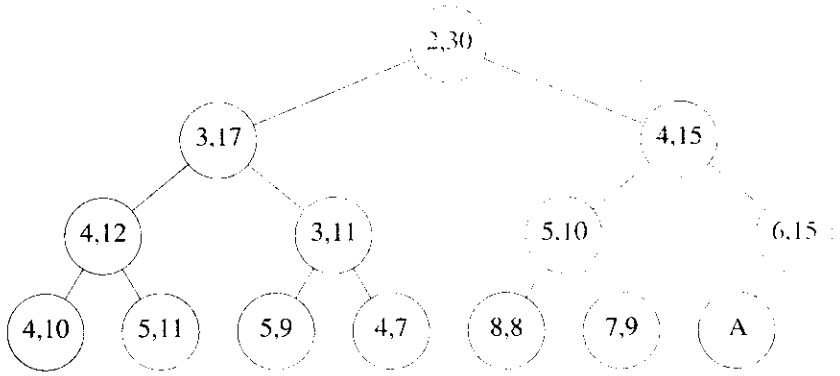


Figure 9.41: Interval heap of Figure 9.39 after one node is added

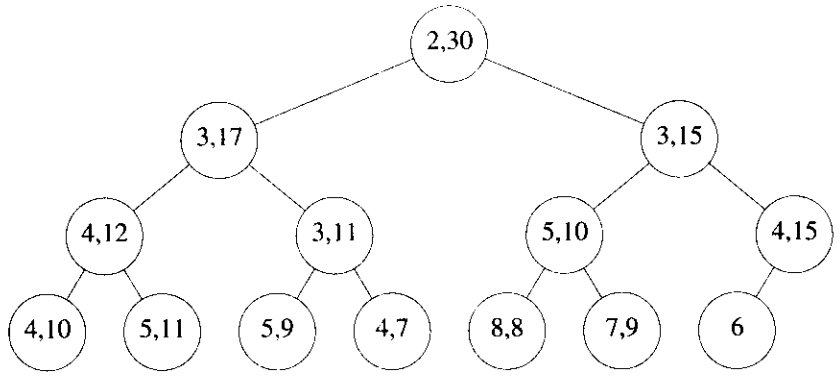


Figure 9.42: The interval heap of Figure 9.39 with 3 inserted

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 9.43 shows the resulting interval heap.

Now, suppose we wish to insert an element into the interval heap of Figure 9.43. Since this interval heap has an odd number of elements, the insertion of the new element

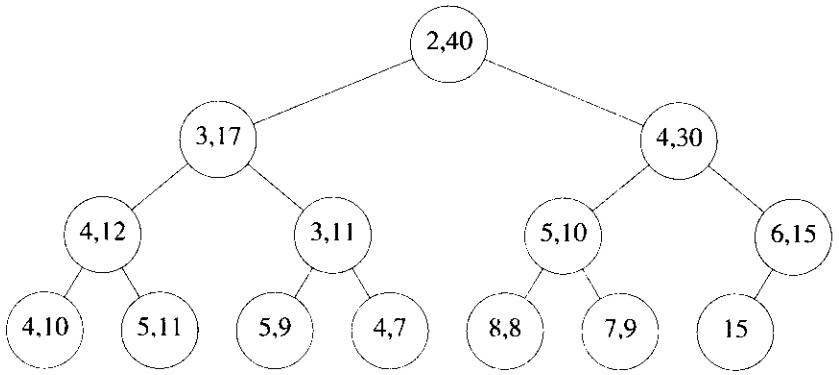


Figure 9.43: The interval heap of Figure 9.39 with 40 inserted

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 9.44 shows the result of inserting the element 32 into the interval heap of Figure 9.43.

9.7.3 Deleting the Min Element

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

(1) When the interval heap is empty, the *delete min* 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 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

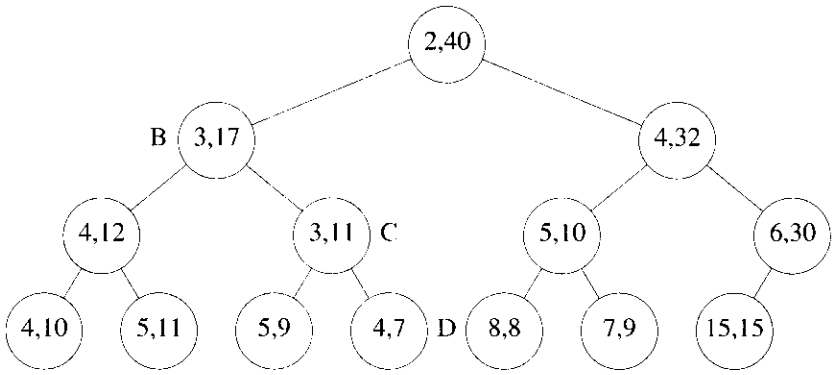


Figure 9.44: The interval heap of Figure 9.43 with 32 inserted

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 9.44. 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 9.45 shows the result.

The max element may be removed using an analogous procedure.

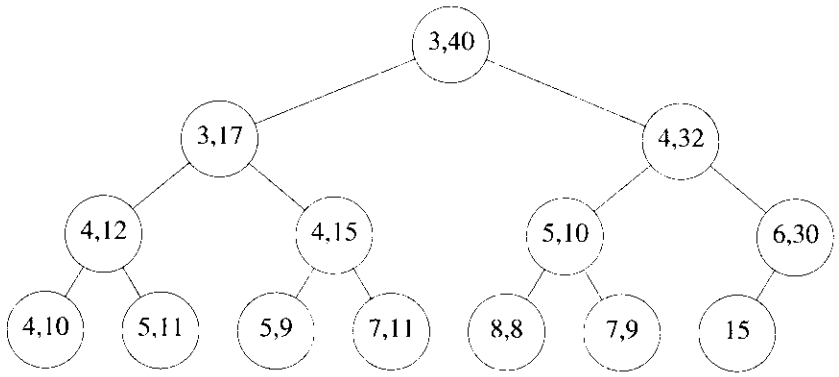


Figure 9.45: The interval heap of Figure 9.44 with minimum element removed

9.7.4 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 *DeleteMin* operation, then reinsert the right end point of this subtree's root using the strategy used for the *delete max* operation.

9.7.5 Complexity of Interval Heap Operations

The min and max elements may be found in O(1) time each; insertion as well as deletion of the min and max elements take O(log n) each; and initializing an n element interval heap takes $\Theta(n)$ time.

9.7.6 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:

Step 1: If the interval tree is empty, **return**.

Step 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**.

Step 3: Report the end points of the root interval that are not in the range $[a,b]$.

Step 4: Recursively search the left subtree of the root for additional points that are not in the range $[a,b]$.

Step 5: Recursively search the right subtree of the root for additional points that are not in the range $[a,b]$.

Step 6: **return**.

Let us try this procedure on the interval heap of Figure 9.44. 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 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 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.

We say that a node is *visited* if its interval is examined in Step 2. With this definition of *visited*, we see that the complexity of the above six step procedure is

Θ(number of nodes visited). The nodes visited in the preceding example are the root and its two children, the two children of node B, and the two children of node C. So, 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.

EXERCISES

1. Start with an empty interval heap and insert the elements 20, 10, 40, 3, 2, 7, 60, 1 and 80 (in this order) using the insertion algorithm developed in this section. Draw the interval heap following each insert.
2. Perform 3 delete-min operations on the interval heap of Figure 9.45. Use the delete min strategy described in this section. Draw the interval heap following each delete min.
3. Perform 4 delete max operations on the interval heap of Figure 9.45. Adapt the delete min strategy of this section to the delete max operation. Draw the interval heap following each delete max operation.
4. Develop the code for all interval heap operations. You also must code the initialization function and a function for the complementary range search operation. Test all functions using your own test data.
5. The min-max heap is an alternative heap inspired data structure for the representation of a DEPQ. A *min-max heap* is a complete binary tree in which each node has exactly one element. Alternating levels of this tree are min levels and max levels, respectively. The root is on a min level. Let x be any node in a min-max heap. If x is on a min (max) level then the element in x has the minimum (maximum) priority from among all elements in the subtree with root x. A node on a min (max) level is called a *min* (*max*) node. Figure 9.46 shows an example 12-elements min-max heap. We use shaded circles for max nodes and unshaded circles for min

nodes.

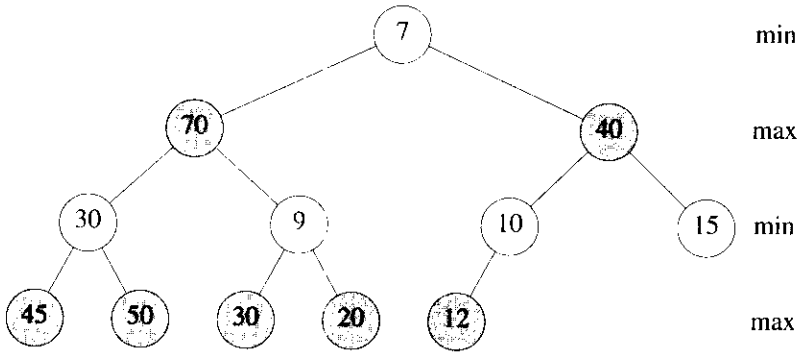


Figure 9.46: A 12-element min-max heap

Fully code and test the double-ended priority queue operations using a min-max heap that is stored using the array representation of a complete binary tree. The complexity of the functions to return the min and max elements should be O(1) and that for the remaining DEPQ functions should be O(log n).

9.8 REFERENCES AND SELECTED READINGS

Height-biased leftist trees were invented by C. Crane. See, "Linear lists and priority queues as balanced binary trees", Technical report CS-72-259, Computer Science Dept., Stanford University, Palo Alto, CA, 1972. Weight-biased leftist trees were developed in "Weight biased leftist trees and modified skip lists," S. Cho and S. Sahni, *ACM Jr. on Experimental Algorithms*, Article 2, 1998.

The exercise on lazy deletion is from "Finding minimum spanning trees," by D. Cheriton and R. Tarjan, *SIAM Journal on Computing*, 5, 1976, pp. 724-742.

B-heaps and F-heaps were invented by M. Fredman and R. Tarjan. Their work is reported in the paper "Fibonacci heaps and their uses in improved network optimization algorithms," *JACM*, 34:3, 1987, pp. 596-615. This paper also describes several variants of the basic F-heap as discussed here, as well as the application of F-heaps to the assignment problem and to the problem of finding a minimum-cost spanning tree. Their result is that using F-heaps, minimum-cost spanning trees can be found in $O(e\beta(e,n))$ time, where $\beta(e,n) \leq \log^* n$ when $e \geq n$. $\log^* n = \min\{i \mid \log^{(i)} n \leq 1\}$, $\log^{(0)} n = n$, and $\log^{(i)} n =$

$\log(\log^{(i-1)} n)$. The complexity of finding minimum-cost spanning trees has been further reduced to $O(e\log\beta(e,n))$. The reference for this is "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs," by H. Gabow, Z. Galil, T. Spencer, and R. Tarjan, *Combinatorica*, 6:2, 1986. pp. 109-122.

Pairing heaps were developed in the paper "The pairing heap: A new form of self-adjusting heap", by M. Fredman, R. Sedgewick, R. Sleator, and R. Tarjan, *Algorithmica*, 1, 1986, pp. 111-129. This paper together with "New upper bounds for pairing heaps," by J. Iacono, *Scandinavian Workshop on Algorithm Theory*, LNCS 1851, 2000, pp. 35-42 establishes the amortized complexity of the pairing heap operations. The paper "On the efficiency of pairing heaps and related data structures," by M. Fredman, *Jr. of the ACM*, 46, 1999, pp. 473-501 provides an information theoretic proof that $\Omega(\log \log n)$ is a lower bound on the amortized complexity of the decrease key operation for pairing heaps.

Experimental studies conducted by Stasko and Vitter reported in their paper "Pairing heaps: Experiments and analysis," *Communications of the ACM*, 30, 3, 1987, 234-249 establish the superiority of two pass pairing heaps over multipass pairing heaps. This paper also proposes a variant of pairing heaps (called *auxiliary two pass pairing heaps*) that performs better than two pass pairing heaps. Moret and Shapiro establish the superiority of pairing heaps over Fibonacci heaps, when implementing Prim's minimum spanning tree algorithm, in their paper "An empirical analysis of algorithms for for constructing a minimum cost spanning tree," *Second Workshop on Algorithms and Data Structures*, 1991, pp. 400-411.

A large number of data structures, inspired by the fundamental heap structure of Section 5.6, have been developed for the representation of a DEPQ. The symmetric min-max heap was developed in "Symmetric min-max heap: A simpler data structure for double-ended priority queue," by A. Arvind and C. Pandu Rangan, *Information Processing Letters*, 69, 1999, 197-199.

The twin heaps of Williams, the min-max pair heapsof Olariu et al., the interval heaps of Ding and Weiss and van Leeuwen et al., and the diamond deques of Chang and Du are virtually identical data structures. The relevant papers are: "Diamond deque: A simple data structure for priority deques," by S. Chang and M. Du, *Information Processing Letters*, 46, 231-237, 1993; "On the Complexity of Building an Interval Heap," by Y. Ding and M. Weiss, *Information Processing Letters*, 50, 143-144, 1994; "Interval heaps," by J. van Leeuwen and D. Wood, *The Computer Journal*, 36, 3, 209-216, 1993; "A mergeable double-ended priority queue," by S. Olariu, C. Overstreet, and Z. Wen, *The Computer Journal*, 34, 5, 423-427, 1991; and "Algorithm 232," by J. Williams, *Communications of the ACM*, 7, 347-348, 1964.

The min-max heap and deap are additional heap-inspired stuctures for DEPQs. These data structures were developed in "Min-max heaps and generalized priority queues," by M. Atkinson, J. Sack, N. Santoro, and T. Strothotte, *Communications of the ACM*, 29:10, 1986, pp. 996-1000 and "The deap: A double-ended heap to implement double-ended priority queues," by S. Carlsson. *Information Processing Letters*, 26,

1987, pp. 33-36, respectively.

Data structures for meldable DEPQs are developed in "The relaxed min-max heap: A mergeable double-ended priority queue," by Y. Ding and M. Weiss, *Acta Informatica*, 30, 215-231, 1993; "Fast meldable priority queues," by G. Brodal, *Workshop on Algorithms and Data Structures*, 1995 and "Mergeable double ended priority queue," by S. Cho and S. Sahni, *International Journal on Foundation of Computer Sciences*, 10, 1, 1999, 1-18.

General techniques to arrive at a data structure for a DEPQ from one for a single-ended priority queue are developed in "Correspondence based data structures for double ended priority queues," by K. Chong and S. Sahni, *ACM Jr. on Experimental Algorithmics*, Volume 5, 2000, Article 2.

For more on priority queues, see Chapters 5 through 8 of "Handbook of data structures and applications," edited by D. Mehta and S. Sahni, Chapman & Hall/CRC, Boca Raton, 2005.