CHAPTER 9

PRIORITY QUEUES

9.1 PAIRING HEAPS

9.1.1 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 9.1 compares the actual and amortized complexities of the Fibonacci and pairing heap operations.

Although the amortized complexities given in Figure 9.1 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	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 9.1: Complexity of Fibonacci and pairing heap operations

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 9.2 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.

9.1.2 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 if Figures 9.2 (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 9.3 (a) is the resulting pairing heap. Figure 9.3 (b) shows the result of melding the pairing heaps of Figures 9.2 (c) and (d). When we meld the pairing

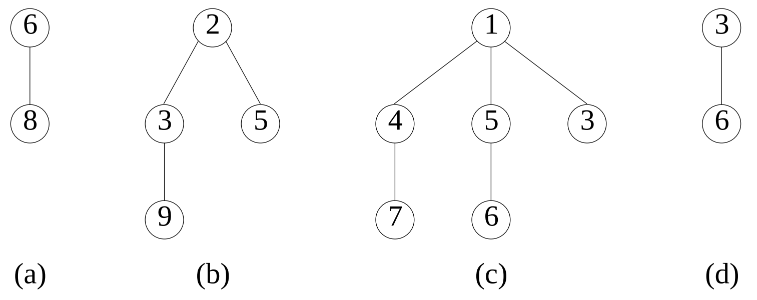
Figure 9.2: Example min pairing heaps

heaps of Figures 9.3 (a) and (b), the result is the pairing heap of Figure 9.4.

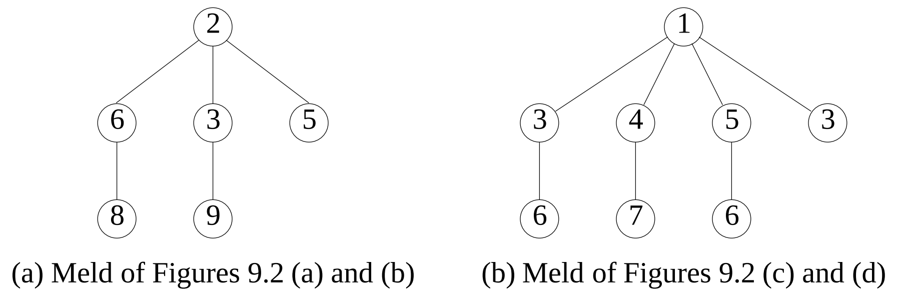
Figure 9.3: Melding pairing heaps

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*.

9.1.3 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.

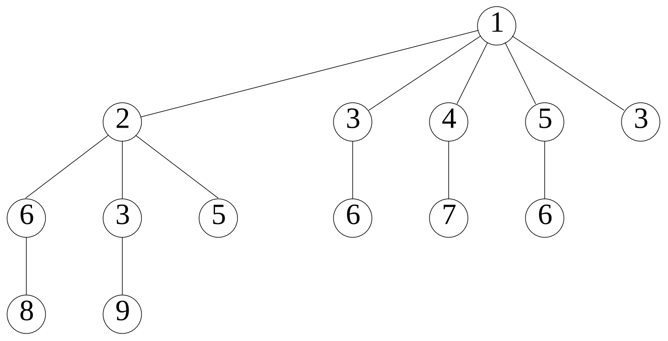

Figure 9.4: Meld of Figures 9.3 (a) and (b)

For example, if the key in the root of the tree of Figure 9.2 (c) is decreased from 1 to 0, or when the key in the leftmost child of the root of Figure 9.2 (c) is decreased from 4 to 2 no additional work is necessary. However, when the key in the leftmost child of the root of Figure 9.2 (c) is decreased from 4 to 0 the new value is less than that in the root (see Figure 9.5 (a)) and corrective action is needed.

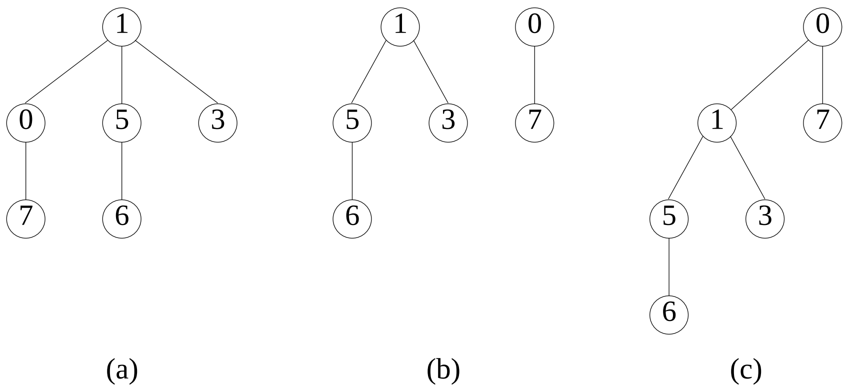

Figure 9.5: Decreasing a key

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 9.5 (b) shows the two min trees following Step 1, and Figure 9.5 (c) shows the result following Step 2.

9.1.4 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 9.6 (a). When the root is removed, we get the collection of 6 min trees shown in Figure 9.6 (b).

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 9.7 shows the resulting three min trees.

In Step 2 (which is a right to left pass), the two rightmost trees of Figure 9.7 are first melded to get the tree of Figure 9.8 (a).

Then the tree of Figure 9.7 with root 0 is melded with the tree of Figure 9.8 to get the final min tree, which is shown in Figure 9.9.

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

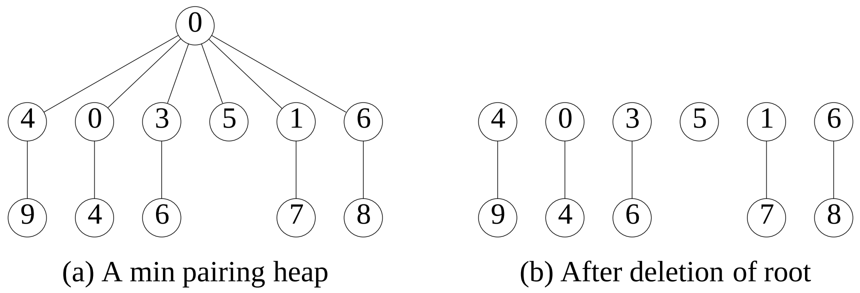

Figure 9.6: Deleting the min element

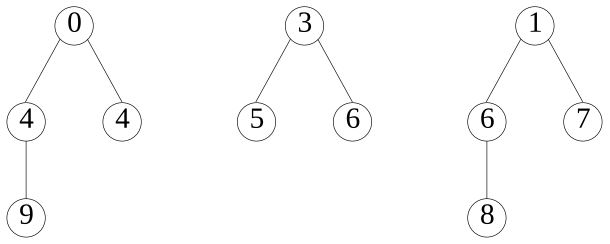

Figure 9.7: Trees following first pass

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 9.6 (b) that result when the root of Figure 9.6 (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

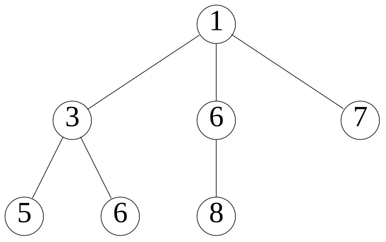

Figure 9.8: First stage of second pass

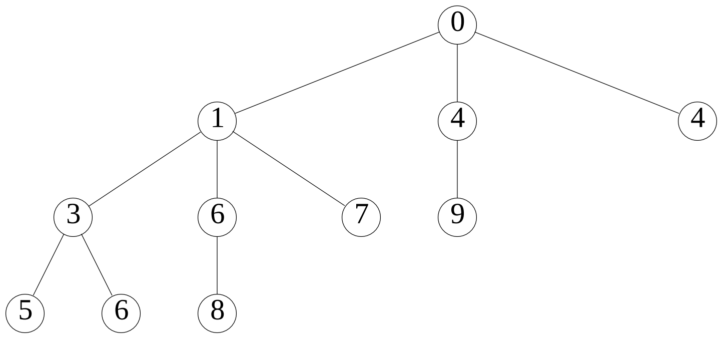

Figure 9.9: Final min pairing heaping following a delete min

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 9.7. 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 9.10.

Finally, the two min trees of Figure 9.10 are melded to get the min tree of Figure 9.11.

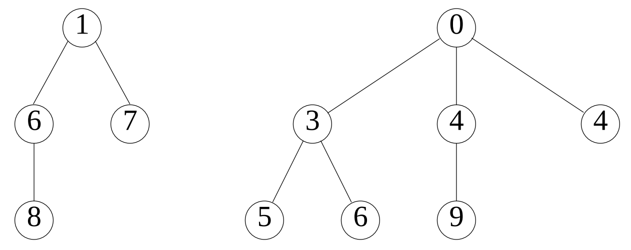

Figure 9.10: Next to last state in multi pass delete

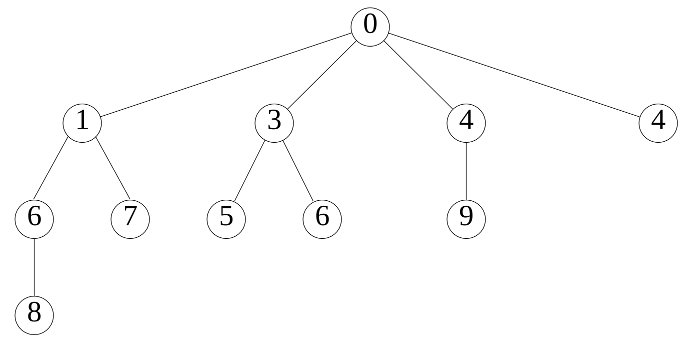

Figure 9.11: Result of multi pass delete min

9.1.5 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.

9.1.6 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 *DecreaseKey* operations) in O(1) time.

9.1.7 Complexity

You can verify that using the described binary tree representation, all pairing heap operations (other than *Delete* and *DeleteMin*) can be done in O(1) time. The complexity of the *Delete* and *DeleteMin* 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 $\Theta(n)$.

9.2 SYMMETRIC MIN-MAX HEAPS

9.2.1 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 9.12 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.

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:

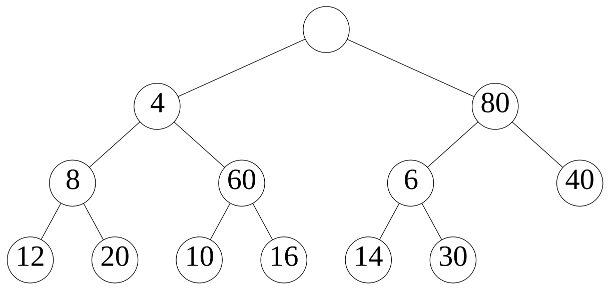

Figure 9.12: A symmetric min-max heap

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 of Program 9.1 can be done efficiently using an SMMH.

9.2.2 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 rightmost 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 *GetMin* and *GetMax* operations take $O(1)$ time each. Program 9.2 defines the class *SMMH*, which implements a symmetric min-max heap. Program 9.3 gives the constructor that creates an empty SMMH.

```
template <class T>
class SMMH : public DEPQ {
public:
    SMMH(int initialCapacity = 10);
    ˜SMMH() {delete [] h;}

    const T& GetMin() const
    {// return min element
        if (last == 1) throw QueueEmpty();
        return h[2];
    }

    const T& GetMax() const
    {// return max element
        if (last == 1) throw QueueEmpty();
        if (last == 2) return h[2];
        else return h[3];
    }

    void Insert(const T&);
    void DeleteMin();
    void DeleteMax();
private:
    int last;           // position of last element in queue
    int arrayLength;    // queue capacity + 2
    T *h;               // element array
};
```

Program 9.2: Class definition for symmetric min-max heap

```
template <class T>
SMMH<T>::SMMH(int initialCapacity)
{// Constructor.
        if (initialCapacity < 1)
        {
                ostringstream s;
                s << "Initial capacity = " << initialCapacity << " Must be > 0";
                throw IllegalParameterValue(s.str());
        }
        arrayLength = initialCapacity + 2;
        h = new T[arrayLength];
        last = 1;
}
```

Program 9.3 Constructor for *SMMH*

9.2.3 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 9.12. Since an SMMH is a complete binary tree, we must add a new node to the SMMH in the position shown in Figure 9.13; 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

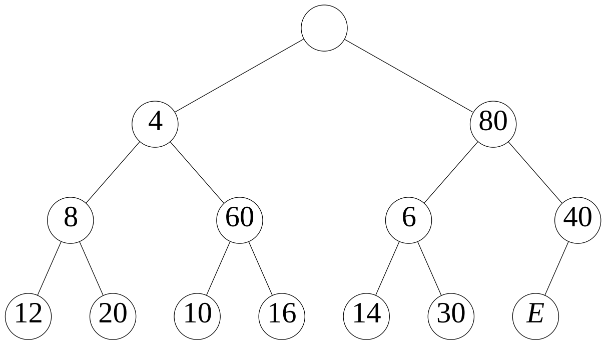

Figure 9.13: The SMMH of Figure 9.12 with a node added

one level to obtain the configuration of Figure 9.14.

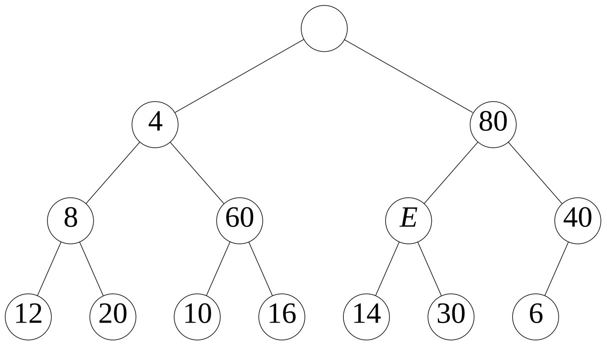

Figure 9.14: The SMMH of Figure 9.13 with 6 moved down

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 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 9.15 shows the resulting configuration.

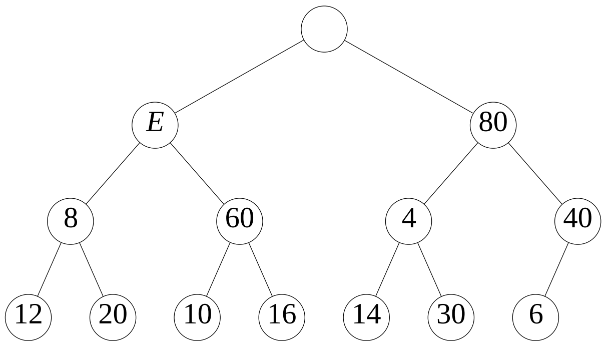

Figure 9.15: The SMMH of Figure 9.14 with 4 moved down

For the configuration of Figure 9.15 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 insert 2 into node *E* and obtain Figure 9.16.

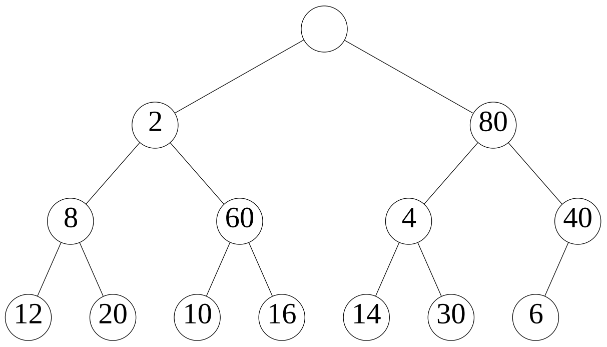

Figure 9.16: The SMMH of Figure 9.15 with 2 inserted

Let us now insert 50 into the SMMH of Figure 9.16. Since an SMMH is a complete binary tree, the new node must be positioned as in Figure 9.17.

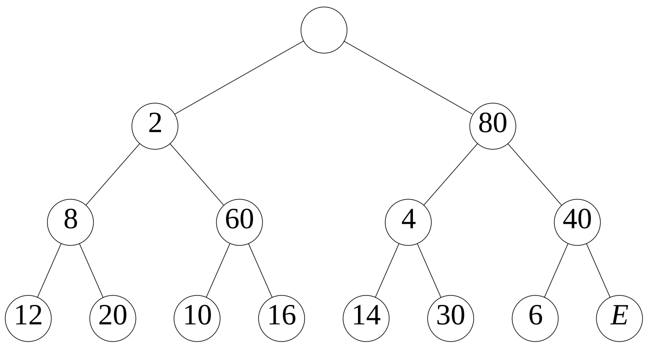

Figure 9.17: The SMMH of Figure 9.16 with a node added

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 child of the grandparent of *E* is moved down to node *E*. Figure 9.18 shows the resulting configuration. Placing 50 into node *E* of Figure 9.18 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*.

Program 9.4 gives the C++ code for the insert operation; the variable *currentNode* refers to the empty node *E* of our example. Since the height of a complete binary tree is $O(\log n)$ and Program 9.4 does $O(1)$ work at each level of the SMMH, the complexity of the insert function is $O(\log n)$.

9.2.4 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

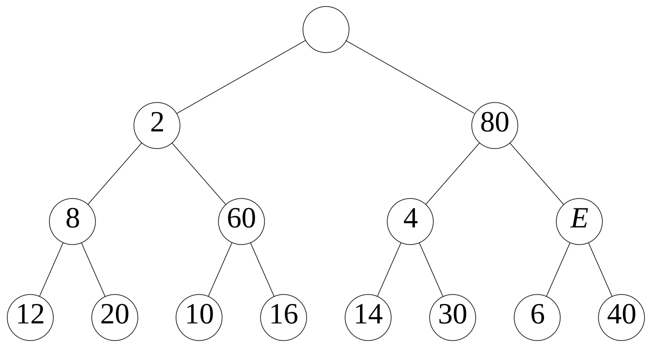

Figure 9.18: The SMMH of Figure 9.17 with 40 moved down

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 9.18 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 9.19. 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 9.20). 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.

```
template <class T>
void SMMH<T>::Insert(const T& x)
{// Insert x into the SMMH.
    // increase array length if necessary
    if (last == arrayLength – 1)
    {// double array length
        ChangeLength1D(h, arrayLength, 2 * arrayLength);
        arrayLength *= 2;
    }
    // find place for x
    // currentNode starts at new leaf and moves up tree
    int currentNode = ++last;
    if (last % 2 == 1 && x < h[last – 1])
    {// left sibling must be smaller, P1
        h[last] = h[last – 1];
        currentNode– –;
    }
    bool done = false;
    while (!done && currentNode >= 4)
    {// currentNode has a grandparent
        int gp = currentNode / 4;    // grandparent
        int lcgp = 2 * gp;           // left child of gp
        int 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 9.4: Insertion into a symmetric min-max heap

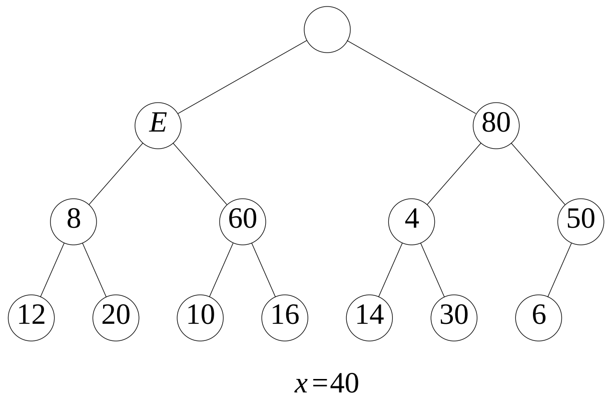

Figure 9.19: The SMMH of Figure 9.18 with 2 deleted

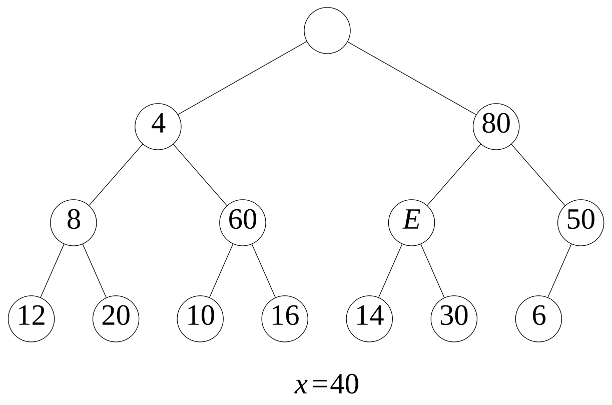

Figure 9.20: The SMMH of Figure 9.19 with *E* and 4 interchanged

So, x cannot be inserted into *E*. Rather, we swap *E* and 6 to get the configuration of Figure 9.21.

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

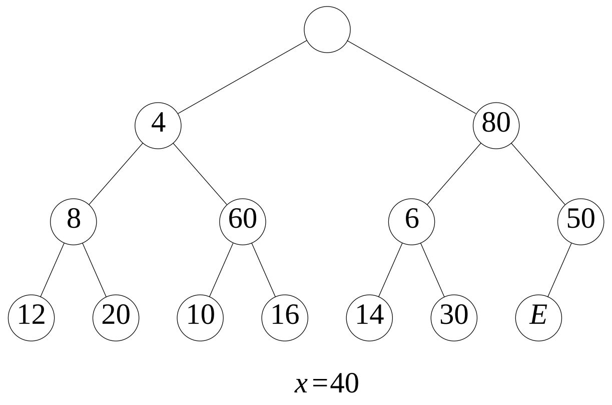

Figure 9.21: The SMMH of Figure 9.20 with *E* and 6 interchanged

consider another delete-min operation. This time, we delete the minimum element from the SMMH of Figure 9.21 (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 9.22 shows the resulting configuration.

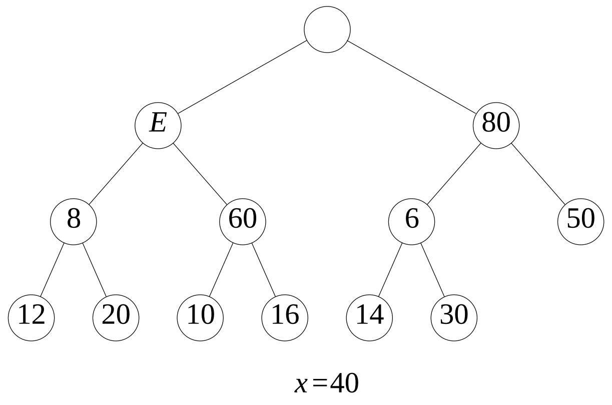

Figure 9.22: First step of another delete min

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 9.23.

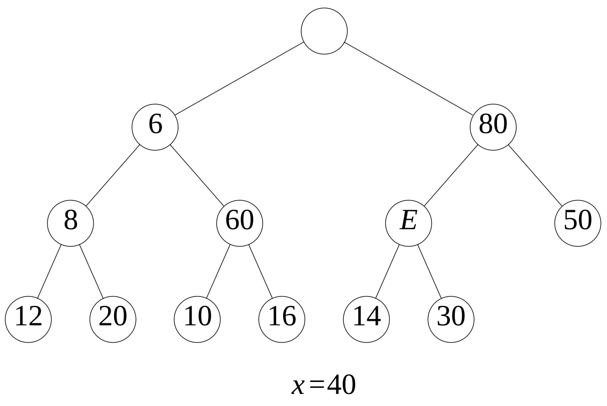

Figure 9.23: The SMMH of Figure 9.22 with *E* and 6 interchanged

Next, we check for a possible P1 violation at the new *E*. Since the sibling of *E* is 50 and $x=40 \le 50$, no P1 violation is detected. The smaller left child of *E* and its sibling 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 9.24.

Since there is a P1 violation at the new *E*, we swap *x* and 30 to get Figure 9.25 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*.

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 $O(\log n)$.

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 9.25 with 30 in the node *E*. Use the delete min strategy described in this section. Draw the SMMH following each delete min.

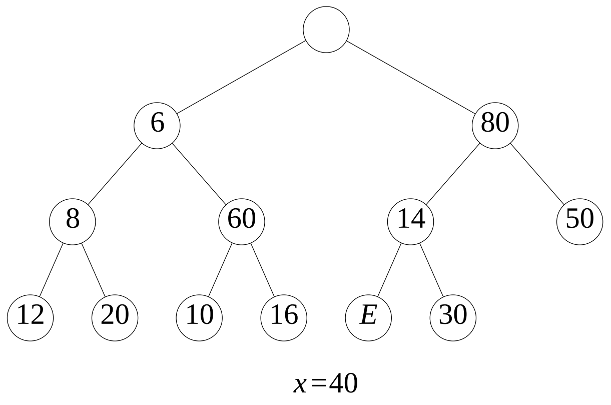

Figure 9.24: The SMMH of Figure 9.23 with *E* and 14 interchanged

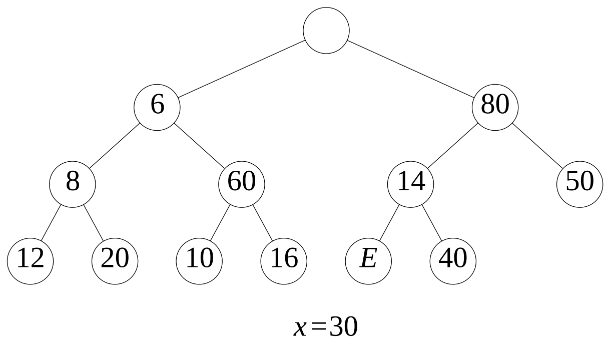

Figure 9.25: The SMMH of Figure 9.24 with *x* and 30 interchanged

4. Perform 4 delete max operations on the SMMH of Figure 9.25 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 functions of the class *SMMH* (Program 9.2). Test all functions using your own test data.

9.3 INTERVAL HEAPS

9.3.1 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 $a \le b$. 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 $a \le c \le d \le b$. 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 $a \le c \le b$, where $[a,b]$ is the interval of the parent (if any) of the last node.

Figure 9.26 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.

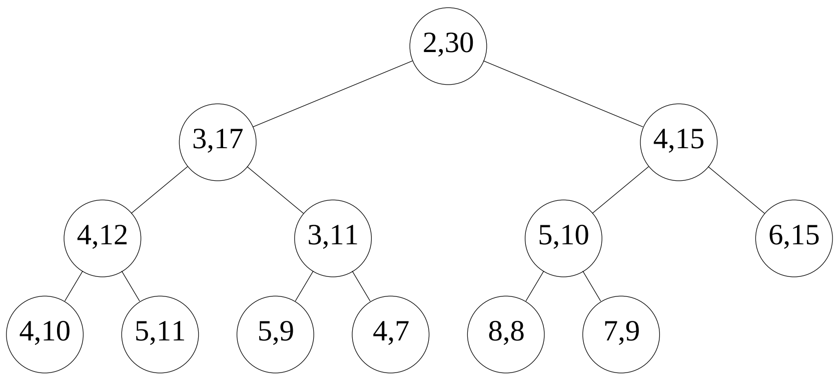

Figure 9.26: An interval heap

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 9.27 shows the min and max heaps defined by the interval heap of Figure 9.26.

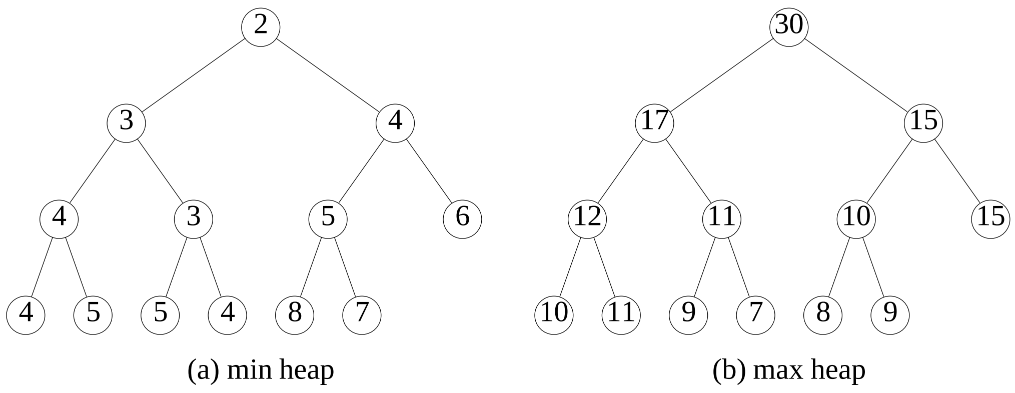

Figure 9.27: Min and max heaps embedded in Figure 9.26

(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.

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

9.3.2 Inserting into an Interval Heap

Suppose we are to insert an element into the interval heap of Figure 9.26. 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 9.28.

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

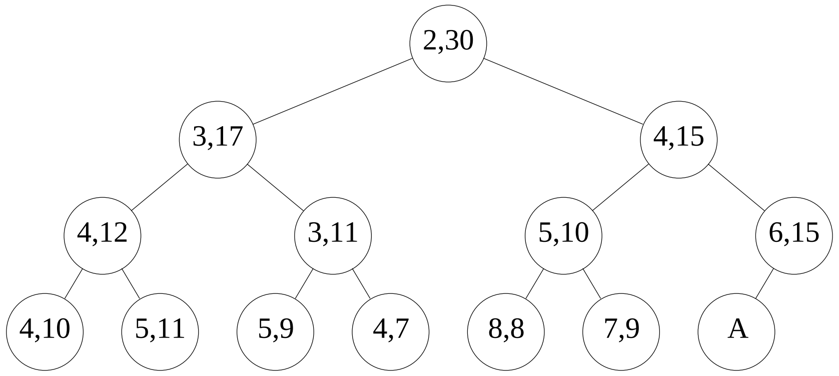

Figure 9.28: Interval heap of Figure 9.26 after one node is added

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 9.26, this element is put into the node *A* shown in Figure 9.28. 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 9.29 shows the resulting interval heap.

To insert the element 40 into the interval heap of Figure 9.26, we follow a path from node *A* (see Figure 9.28) towards the root, moving right end points down until we either pass the root or reach a node whose right end point is ≥ 40. The new element is inserted into the node that now has no right end point. Figure 9.30 shows the resulting interval heap.

Now, suppose we wish to insert an element into the interval heap of Figure 9.30. Since this interval heap has an odd number of elements, the insertion of the new element does not increase the number of nodes. The insertion procedure is the same as for the case when we initially have an even number of elements. Let *A* denote the last node in the heap. If the new element lies within the interval [6,15] of the parent of *A*, then the new element is inserted into node *A* (the new element becomes the left end point of *A* if it is less than the element currently in *A*). If the new element is less than the left end point 6 of the parent of *A*, then the

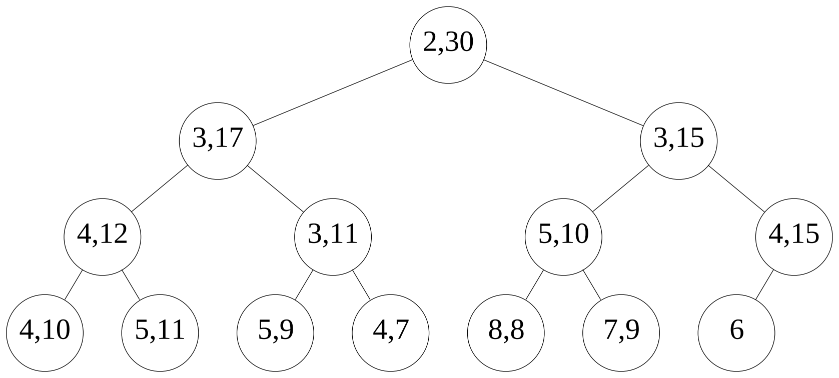

Figure 9.29: The interval heap of Figure 9.26 with 3 inserted

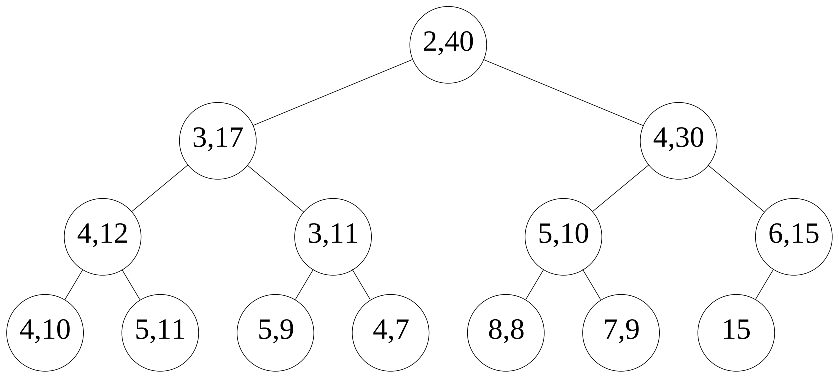

Figure 9.30: The interval heap of Figure 9.26 with 40 inserted

new element is inserted into the embedded min heap; otherwise, the new element is inserted into the embedded max heap. Figure 9.31 shows the result of inserting the element 32 into the interval heap of Figure 9.30.

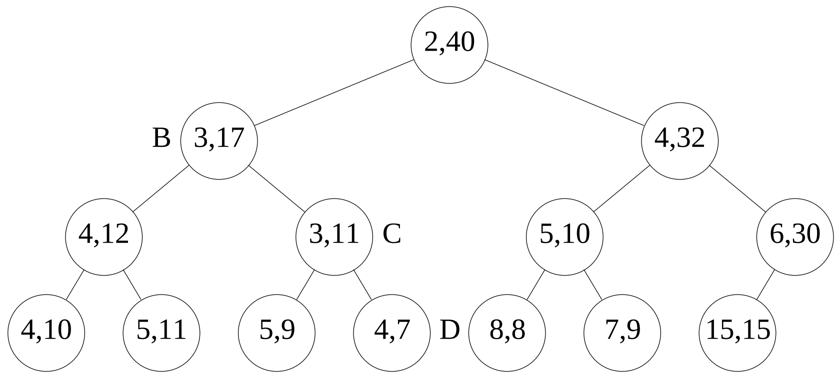

Figure 9.31: The interval heap of Figure 9.30 with 32 inserted

9.3.3 Deleting the Min Element

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

(1) When the interval heap is empty, the *DeleteMin* 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 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 $p \le r$. 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 9.31. 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 ≤ 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 *C*s 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 *D*s right end point. Now, since *D* has no children, the current *p*=7 is inserted into node *D* as *D*s left end point. Figure 9.32 shows the result.

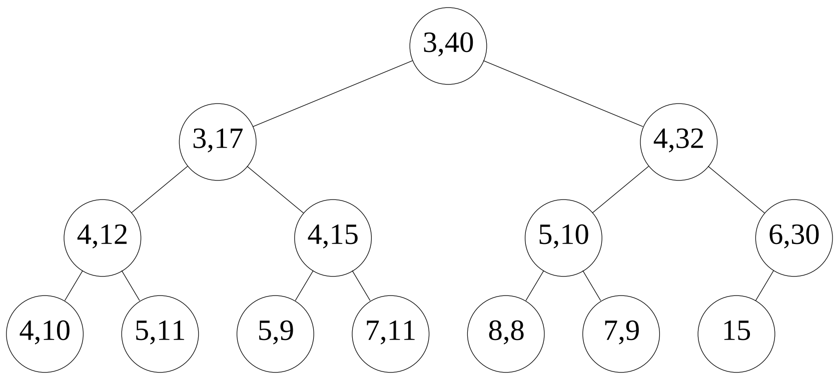

Figure 9.32: The interval heap of Figure 9.31 with minimum element removed

The max element may be removed using an analogous procedure.

9.3.4 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 *DeleteMax* operation.

9.3.5 Complexity of Interval Heap Operations

The operations *GetMin* () and *GetMax* () take O(1) time each; *Insert* (x), *DeleteMin* (), and *DeleteMax* () take O(log n) each; and initializing an n element interval heap takes $\Theta(n)$ time.

9.3.6 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 9.31. 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 9.32. 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 9.32. 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 functions of the class *IntervalHeap*, which implements the interval heap data structure and derives from the virtual class *DEPQ*. In addition to the functions specified in *DEPQ* 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 9.33 shows an example 12-elements min-max heap. We use shaded circles for max nodes and unshaded circles for min nodes.

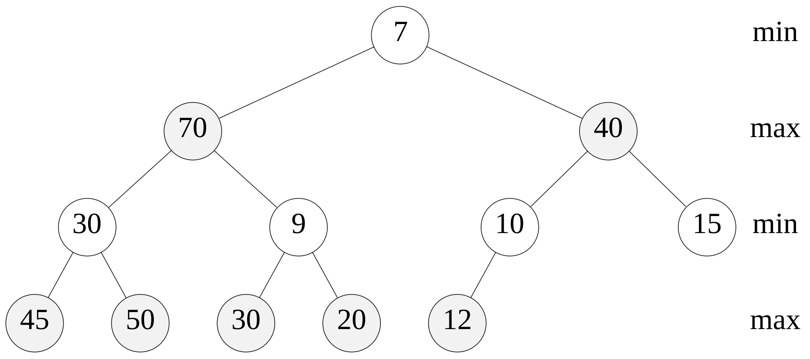

Figure 9.33: A 12-element min-max heap

Fully code and test the class *MinMaxHeap*, which impements a min-max heap using a one-dimensional array for the complete binary tree. Your class must provide an implementation for all DEPQ functions. The complexity of *GetMin* and *GetMax* should be O(1) and that for the remaining DEPQ functions should be O(log n).

9.4 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 $\beta(e,n) \le \log^* n$ when $e \ge n$. $\log^* n = \min\{i \mid \log^{(i)} n \le 1\}$, $\log^{(0)} n = n$, 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.