

9.7 INTERVAL HEAPS

9.7.1 Double-ended Priority Queues

The priority queues we have studied so far in this chapter may more accurately be called **single-ended priority queues** because they permit deletion from only one end—either the min end or the max end. A **double-ended priority queue**, on the other hand, permits you to delete from either end. ADT 9.2 gives the abstract data type specification for a double-ended priority queue.

```
AbstractDataType DoubleEndedPriorityQueue {  
  instances  
    finite collection of elements, each has a priority  
  operations  
    Create(): create an empty double-ended priority queue  
    Size(): return number of elements in the queue  
    Min(): return element with minimum priority  
    Max(): return element with maximum priority  
    Insert(x): insert x into the queue  
    DeleteMin(x): delete the element with minimum priority from the queue;  
                   return this element in x;  
    DeleteMax(x): delete the element with maximum priority from the queue;  
                   return this element in x;  
}
```

ADT 9.2 Abstract data type specification of a max priority queue

Like single-ended priority queues, double-ended priority queues can be used in simulations. Suppose we are running simulations under the constraint that the queue at any station has a maximum permissible size. When an attempt is made to insert a new job that would increase the queue size beyond its maximum permissible size, the job with minimum priority is deleted. When determining this minimum priority job, the new job is also included. When the station is ready to service a new job, the job with maximum priority is deleted from the queue.

9.7.2 Definition of an Interval Heap

An interval heap is an elegant extension of a min heap and a max heap that permits us to insert and delete elements in $O(\log n)$ time, where n is the number of elements in the double-ended priority queue.

Definition An interval heap is a complete binary tree that satisfies the following properties:

- When the total number n of elements is even, each node of this complete binary tree has two elements a and b , $a \leq b$. We say that the node represents the interval $[a, b]$.
- When n is odd, each node other than the last one has two elements a and b , $a \leq b$. The last node has a single element a . The interval represented by the last node is $[a, a]$.
- Let $[a_c, b_c]$ be the interval represented by any nonroot node in the interval heap. Let $[a_p, b_p]$ be the interval represented by the parent of this node. Then, $[a_c, b_c] \subseteq [a_p, b_p]$ (equivalently, $a_p \leq a_c \leq b_c \leq b_p$). ■

A 19 element interval heap is shown in Figure 9.12. The shaded node is the last node. Since each node other than the last has two elements, the total number of nodes is 10. Notice that the left ends of the node intervals define a min heap and that the right ends define a max heap. Using the definition of an interval heap, you can verify that this is the case for all interval heaps. Further, you can verify that when $n > 1$, the minimum element is the left end of the root interval and the maximum element is the right end of the root interval; when $n = 1$, the single element in the root is both the min and the max element; and when $n = 0$, there is neither a min nor a max element.

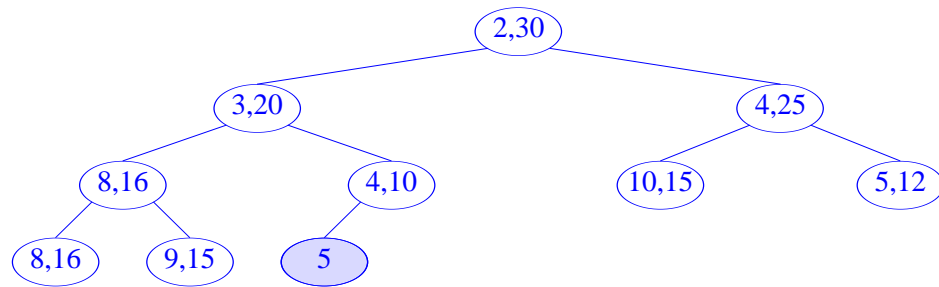


Figure 9.12 A 19 element interval heap

Theorem 9.4 A complete binary tree in which each node represents an interval is an interval heap iff the left ends define a min heap and the right ends define a max heap.

Proof From the definition of an interval heap, it follows that for every pair (c,p) of nodes in an interval heap such that c is a child of p , $a_p \leq a_c$ and $b_p \geq b_c$. Therefore, the interval left ends define a min heap and the right ends define a max heap.

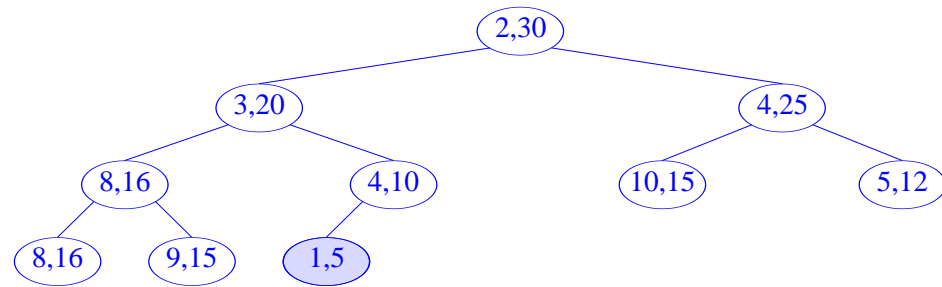
Next suppose we have a complete binary tree in which every node represents an interval and the interval left ends define a min heap and the right ends a max heap. It follows that for every pair (c,p) of nodes in the binary tree such that c is a child of p , $a_p \leq a_c$ and $b_p \geq b_c$. Further, since the elements in c define an interval whose left end is a_c and whose right end is b_c , it follows that $a_p \leq a_c \leq b_c \leq b_p$. Therefore, the binary tree is an interval heap. ■

Since an interval heap is a complete binary tree, it is efficiently represented in a one-dimensional array using the formula-based scheme described in Section 8.4. Recall that the same scheme is used to represent a min heap and a max heap. However, in the case of an interval heap, two elements are stored in each array position. An n element interval heap uses array positions $1, 2, \dots, \lceil n/2 \rceil$. The last node is in array position $\lceil n/2 \rceil$.

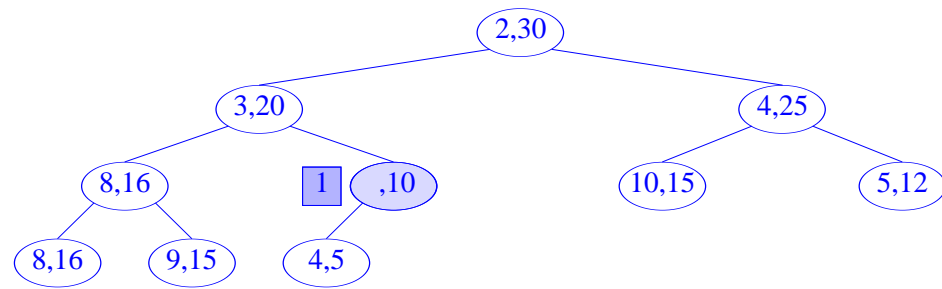
Notice also that the height of an interval heap is $O(\log n)$ because it is a complete binary tree.

9.7.3 Insertion into an Interval Heap

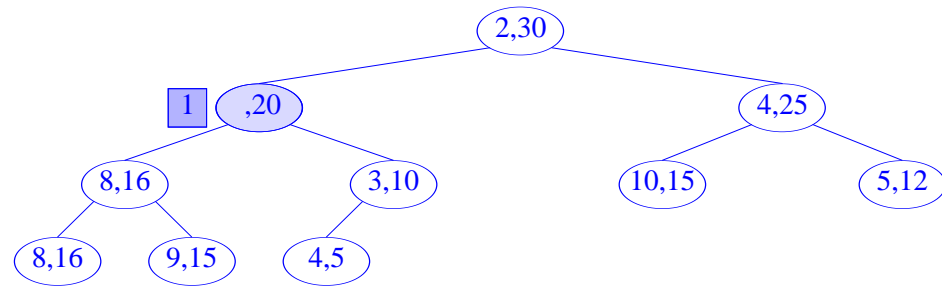
Suppose we wish to insert the element 1 into the interval heap of Figure 9.12. Since the last node has only one element, this node can accommodate the new element. If we put the new element into this node, the interval represented by the node becomes $[1,5]$ (see Figure 9.13(a)). Notice that the interval right ends still define a max heap. However, the interval left ends do not define a min heap. To remedy this, we proceed as if we are inserting 1 into the min heap defined by the left ends. Recall that when inserting into a min heap, we traverse the path from the new node to the root. In the case of an interval heap, this corresponds to traversing the path from the last node to the root. Since $1 < 4$, the 4 is moved down to the last node to get the configuration of Figure 9.13(b). Then 1 is compared with the left end 3 of the parent of the shaded node of Figure 9.13(b). Since $1 < 3$, the 3 is moved to the shaded node and the configuration of Figure 9.13(c) is obtained. Finally, 1 is compared with the left end of the root. Since $1 < 2$, the 2 is moved down and we obtain the configuration of Figure 9.13(d). Notice that this moving down of left ends replaces old left ends with smaller ones. Consequently, the pair in each node defines an interval and every child interval is contained in its parent interval. The result is an interval heap.



(a)



(b)

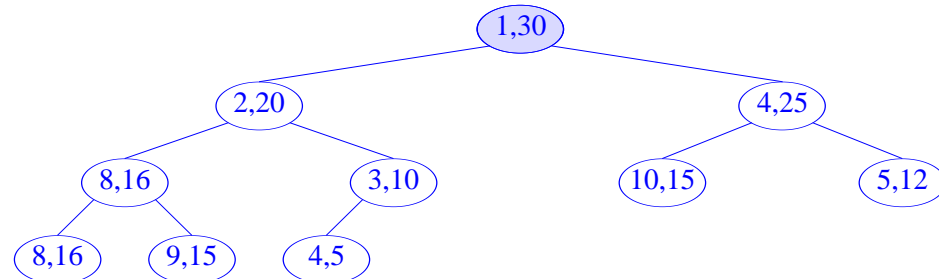


(c)

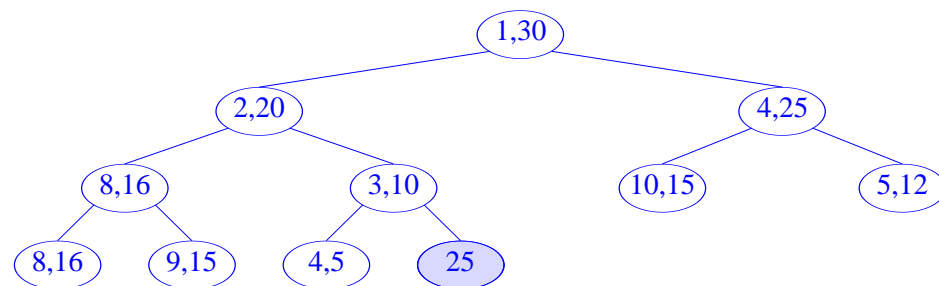
Figure 9.13 Insertion into an interval heap (continues)

As another example, consider the insertion of 25 into the interval heap of Figure 9.13(d). Since this interval heap has an even number of elements, we must add a new last node. The new last node will contain only one element. Tentatively, the element in the new last node is 25 (see Figure 9.13(e)). The

interval right ends do not define a max heap. To fix this problem, we use the strategy used to insert into a max heap. That is, we follow the path from the new node to the root. The interval right ends of the nodes on this path are examined and a right end is moved down if it is less than the newly inserted element.



(d)



(e)

Figure 9.13 Insertion into an interval heap (continues)

In our example, 25 is first compared with 10. Since $10 < 25$, the 10 is moved down to the last node to get the configuration of Figure 9.13(f). Next 25 is compared with 20 and the 20 is moved down. Since $25 < 30$, the 30 is not moved down and the 25 is inserted into the left child of the root. The resulting interval heap is shown in Figure 9.13(g).

Whenever we insert into an interval heap, three cases are possible—(1) the new element lies within the interval defined by the parent (if any) of the last node, (2) the new element lies to the left of the parent interval, and (3) the new element lies to the right of the parent interval. When case (1) happens, the new element remains in the last node and no element moving is required. In case (2) the interval left ends do not define a min heap and we proceed to fix this problem

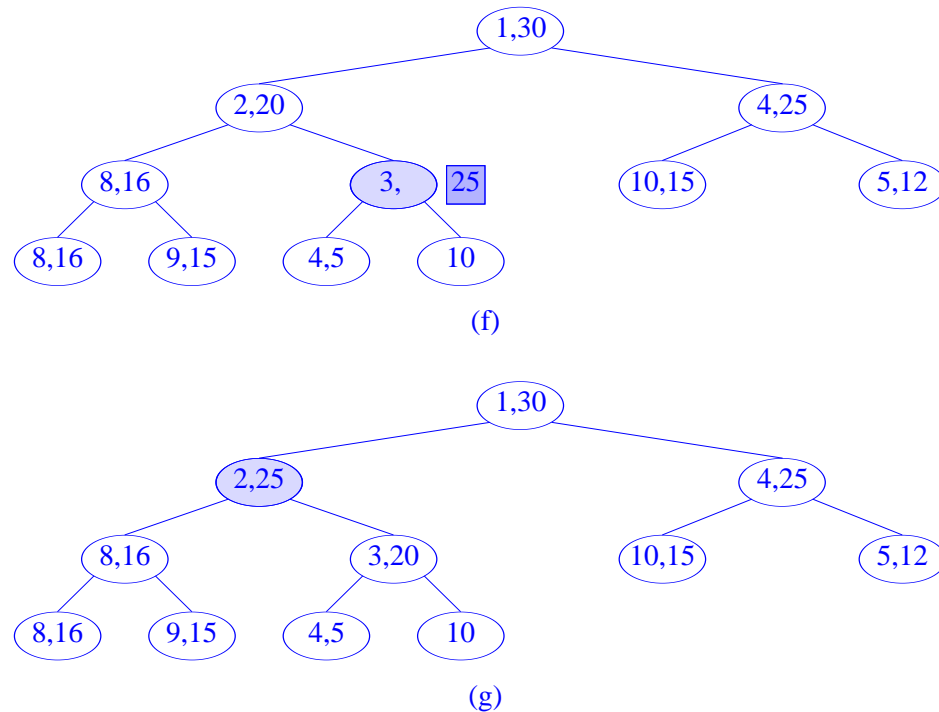


Figure 9.13 Insertion into an interval heap (concluded)

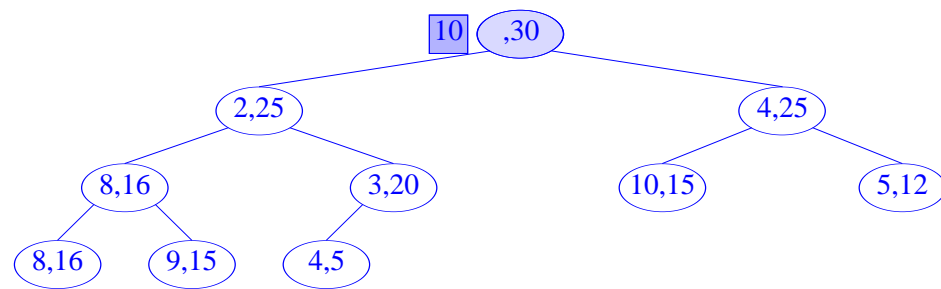
using the strategy used to insert into a min heap. When case (3) arises, the interval right ends do not form a max heap and we fix this problem using the strategy used to insert into a max heap.

Since we can insert an element into an interval heap making a leaf to root pass doing constant work per node encountered, the time needed for the insert operation is $O(\log n)$.

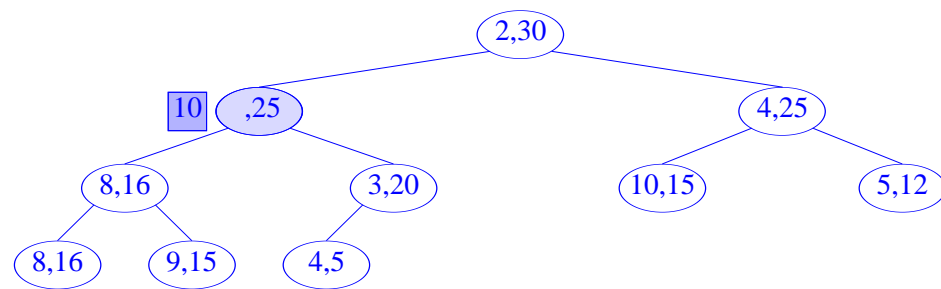
9.7.4 Deletion from an Interval Heap

Suppose we are to delete the min element from the interval heap of Figure 9.13(g). This element is the left end of the root interval, that is element 1. Following the deletion of the element 1, the interval heap contains only 20 elements. Therefore, the current last node is to be removed. The single element 10 that is in this last node must be reinserted. Figure 9.14(a) shows the interval heap configuration after the min element and the last node are deleted. Notice that the

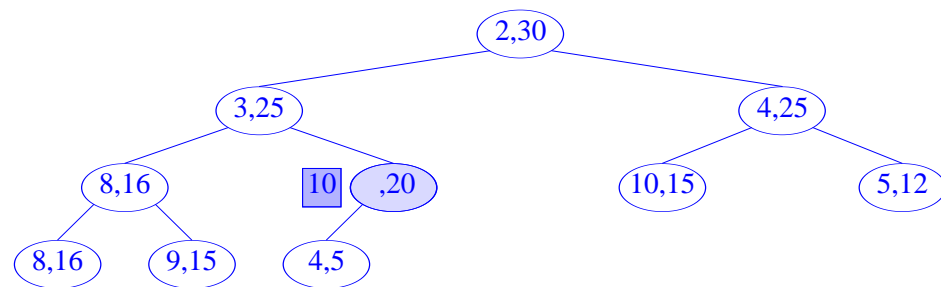
element removed from the last node is guaranteed to be \leq the right end of the root interval because this right end in the max element.



(a)



(b)



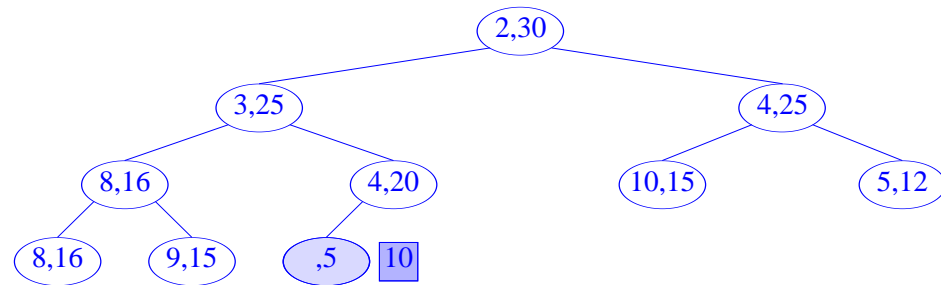
(c)

Figure 9.14 Deletion from an interval heap (continues)

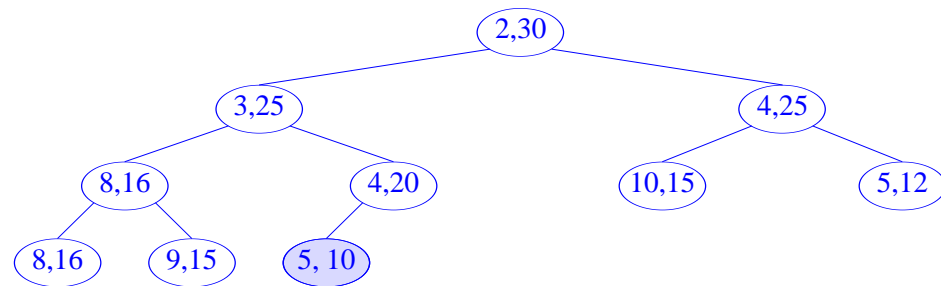
Since the element 10 that is to be reinserted is less than the remaining element in the root, it is a candidate for the left end of the root interval. We check for a possible left end violation by comparing against the left ends of the children of the root. The smaller of the children left ends is 2 and $2 < 10$. Therefore, the 2 is moved to the root and becomes the root's left end. Figure 9.14(b) shows the new configuration. Next we attempt to place 10 into the node that formerly housed 2. Since $10 < 25$, we attempt to put it in as the left end of this node's interval. The smaller of the left ends of the children of the shaded node of Figure 9.14(b) is determined and compared with 10. Since $3 < 10$, 3 is moved up and the configuration of Figure 9.14(c) is obtained. Since $10 < 20$, we attempt to make 10 the left end of the interval of the shaded node of Figure 9.14(c). So we check the min heap property at this node. Since $10 > 4$, 4 is moved up and the configuration of Figure 9.14(d) is obtained. Now since $10 > 5$, we swap the 10 and 5 and attempt to make 5 the left end of the interval of the shaded node of Figure 9.14(d). Figure 9.14(e) shows the result. Even though we have swapped the 10 and 5, a right end violation is not possible because we have replaced the old right end by a larger value that is known to be \leq the right end of the parent. Notice the similarity between this deletion process and that used to delete from a heap. The only difference is that each time we move one node down, we may need to swap the left end candidate with the right end of the node we have moved to.

Suppose we wish to delete the max element from the interval heap of Figure 9.14(e). The right end 30 of the root interval is to be deleted. Following the deletion, the interval heap will have 19 elements. So the last node should have a single element. We remove one of the two elements from the last node. If we remove the right end 10 of the last node, the configuration of Figure 9.14(f) results. Notice that the element removed from the last node is guaranteed to be \geq the left end of the root because the left end of the root is the min element. Before we can make 10 the right end of the root interval, we must verify the max heap property at this node. Since $10 < 25$, one of the two 25s is to be moved up. If we move the 25 from the left child, the new configuration is as in Figure 9.14(g). Since $10 > 3$, we attempt to make 10 the right end of the shaded node's interval. But since $10 < \max\{16, 20\}$, the 20 is moved up and the configuration of Figure 9.14(h) obtained. Since $10 > 4$, we attempt to make 10 the right end of the interval of the shaded node. Since 10 is at least as large as the right end of the child's interval, 10 is inserted into the shaded node of Figure 9.14(h) and the configuration of Figure 9.14(i) obtained.

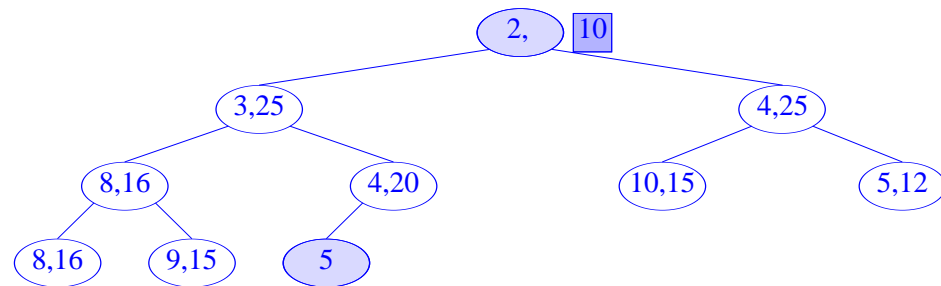
To summarize, we can delete either the min or the max element following a procedure similar to that used to delete from a heap. To delete the min element, we delete the left end of the root as well as the last element in the interval heap. We then proceed to reinsert the deleted last element into the min heap of the interval heap. This reinsertion proceeds like the reinsert step following the deletion of the min element from a min heap. However, each time we move to a node at the next level, we swap the element to be reinserted with the right end of



(d)



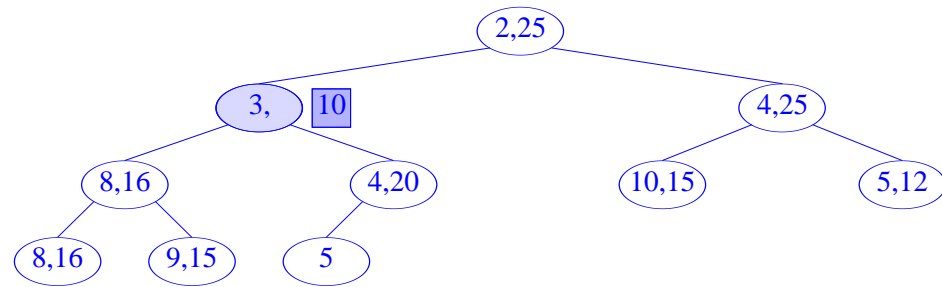
(e)



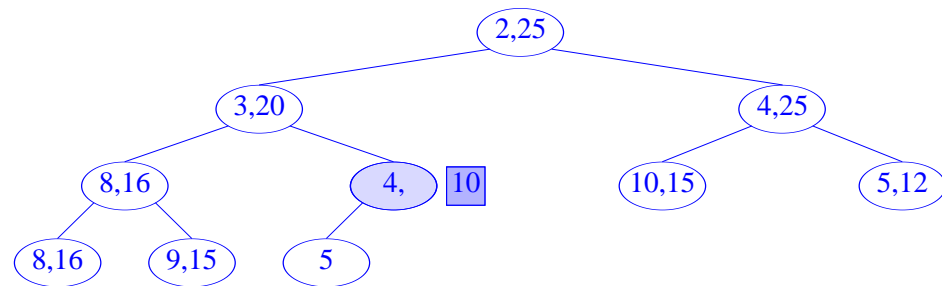
(f)

Figure 9.14 Deletion from an interval heap (continues)

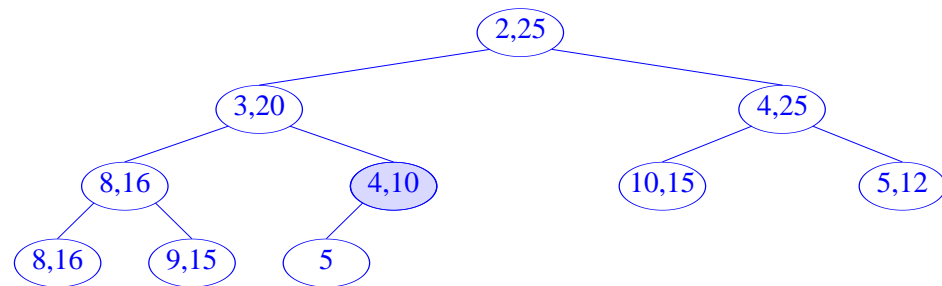
the node moved to in case the element to be reinserted is greater than the right end. To delete the max element, we delete the right end of the root as well as the last element of the interval heap. The deleted last element is reinserted using a process similar to that used to reinsert into a max heap. However, each time we move to a node at the next level, we swap the element to be reinserted with the



(g)



(h)



(i)

Figure 9.14 Deletion from an interval heap (concluded)

left end of the node moved to in case the element to be reinserted is less than the left end.

Since we can delete either the min or max element from an interval heap making a root to leaf pass doing constant work per node encountered, the time needed for the delete is $O(\log n)$.

9.7.5 Initializing an Interval Heap

An n element array with two elements in each array position except possibly the last position which may have one element can be restructured, in linear time, so as to represent an n element interval heap. The strategy is very similar to that used for the linear time initialization of a min or max heap.

To initialize an interval heap, the array positions are examined in the order $\lceil n/2 \rceil, \dots, 1$. When we examine array position i , we first order the up to two elements in position i so that the left one is \leq the right one. Next we ensure that the right element in position i is the largest one in the subtree with root i . This is done by using the procedure used to reinsert the element removed from the last node following the deletion of the max element from the root. Finally we ensure that the left element in i is the smallest one in the subtree with root i . This is done by using the procedure used to reinsert the element removed from the last node following the deletion of the min element from the root. The time spent at each array position i is $O(\text{height of the subtree with root } i)$. This is the same as the time spent at each nonleaf node by the heap initialization method. The time spent at each leaf node is $\Theta(1)$ and the number of leaf nodes is $\Theta(n)$. Therefore, the total time taken to initialize an interval heap is $\Theta(n)$.

9.7.6 The Class `IntervalHeap`

An interval heap can be represented as an array of type `TwoElement`. Program 9.16 gives the definition of the class `TwoElement`.

```
template <class T>
class TwoElement {
    friend IntervalHeap<T>;
public:
    T left,    // left element
      right;  // right element
};
```

Program 9.16 The class `TwoElement`

To simplify the code we duplicate the single element that resides in the last node whenever the total number of elements is odd. When this is done, all nodes have two elements in them. We shall use this strategy for our implementation. Program 9.17 gives the specification of the class `IntervalHeap`. The codes for the methods `Min` and `Max` are also given here. Notice that because of the duplication strategy used, the code for `Max` does not handle the case when the current size is 1 any different from the case when the current size is > 1 .

```

template<class T>
class IntervalHeap {
public:
    IntervalHeap(int IntervalHeapSize = 10);
    ~IntervalHeap() {delete [] heap;}
    int Size() const {return CurrentSize;}
    T Min() {if (CurrentSize == 0)
            throw OutOfBounds();
           return heap[1].left;}
    T Max() {if (CurrentSize == 0)
            throw OutOfBounds();
           return heap[1].right;}
    IntervalHeap<T>& Insert(const T& x);
    IntervalHeap<T>& DeleteMin(T& x);
    IntervalHeap<T>& DeleteMax(T& x);
private:
    int CurrentSize, // number of elements in heap
        MaxSize;    // max elements permitted
    TwoElement<T> *heap; // element array
};

```

Program 9.17 The class IntervalHeap

Program 9.18 gives the code for the interval heap constructor. Although the code permits you to have an odd maximum size, there will generally be no advantage to specifying an odd maximum size.

```

template<class T>
IntervalHeap<T>::IntervalHeap(int IntervalHeapSize)
{ // Interval heap constructor.
    MaxSize = IntervalHeapSize;

    // determine number of array positions needed
    // array will be heap[0:n-1]
    int n = MaxSize / 2 + MaxSize % 2 + 1;

    heap = new TwoElement<T> [n];
    CurrentSize = 0;
}

```

Program 9.18 Constructor for IntervalHeap

Program 9.19 gives the code for the insert method and Programs 9.20 and 9.21 give the codes for the delete methods.

```

template<class T>
IntervalHeap<T>& IntervalHeap<T>::Insert(const T& x)
{ // Insert x into the interval heap.
  if (CurrentSize == MaxSize)
    throw NoMem(); // no space

  // handle CurrentSize < 2 as a special case
  if (CurrentSize < 2) {
    if (CurrentSize) // CurrentSize is 1
      if (x < heap[1].left)
        heap[1].left = x;
      else heap[1].right = x;
    else { // CurrentSize is 0
      heap[1].left = x;
      heap[1].right = x;
    }
    CurrentSize++;
    return *this;
  }

  // CurrentSize >= 2
  int LastNode = CurrentSize / 2 + CurrentSize % 2;
  bool minHeap; // true iff x is to be
                // inserted in the min heap part
                // of the interval heap
  if (CurrentSize % 2)
    // odd number of elements
    if (x < heap[LastNode].left)
      // x will be an interval left end
      minHeap = true;
    else minHeap = false;
  else { // even number of elements
    LastNode++;
    if (x <= heap[LastNode / 2].left)
      minHeap = true;
    else minHeap = false;
  }
}

```

Program 9.19 Inserting into an interval heap (continues)

```

if (minHeap) { // fix min heap of interval heap
    // find place for x
    // i starts at LastNode and moves up tree
    int i = LastNode;
    while (i != 1 && x < heap[i / 2].left) {
        // cannot put x in heap[i]
        // move left element down
        heap[i].left = heap[i / 2].left;
        i /= 2; // move to parent
    }
    heap[i].left = x;
    CurrentSize++;
    if (CurrentSize % 2)
        // new size is odd, put dummy in LastNode
        heap[LastNode].right = heap[LastNode].left;
}
else { // fix max heap of interval heap
    // find place for x
    // i starts at LastNode and moves up tree
    int i = LastNode;
    while (i != 1 && x > heap[i / 2].right) {
        // cannot put x in heap[i]
        // move right element down
        heap[i].right = heap[i / 2].right;
        i /= 2; // move to parent
    }
    heap[i].right = x;
    CurrentSize++;
    if (CurrentSize % 2)
        // new size is odd, put dummy in LastNode
        heap[LastNode].left = heap[LastNode].right;
}

return *this;
}

```

Program 9.19 Inserting into an interval heap (concluded)

The insert and delete codes closely follow our earlier discussion. Our strategy of duplicating the last element when the current size is odd has simplified these codes.

```

template<class T>
IntervalHeap<T>& IntervalHeap<T>::DeleteMin(T& x)
{ // Set x to min element and delete
  // min element from interval heap.
  // check if interval heap is empty
  if (CurrentSize == 0)
    throw OutOfBounds(); // empty

  x = heap[1].left; // min element

  // restructure min heap part
  int LastNode = CurrentSize / 2 + CurrentSize % 2;
  T y; // element removed from last node
  if (CurrentSize % 2) { // size is odd
    y = heap[LastNode].left;
    LastNode--;
  }
  else { // size is even
    y = heap[LastNode].right;
    heap[LastNode].right = heap[LastNode].left;
  }
  CurrentSize--;
}

```

Program 9.20 Deleting the min element (continues)

9.7.7 An Application—The Complementary Interval Problem

The dynamic complementary interval problem is a problem from computational geometry that involves a dynamic collection of points that lie on a straight line. Each point is described by its coordinate which is just the distance of the point from some reference point on the line. The following operations are performed on the collection of points:

- insert a point (i.e., a coordinate)
- delete the point with min coordinate
- delete the point with max coordinate
- report all points that lie outside any given interval $[x,y]$, $x \leq y$ (the point with coordinate z is outside the interval iff either $z < x$ or $z > y$)

It is clear that by using an interval heap to represent the point coordinates, the insert and delete operations can be performed in $O(\log n)$ time per operation, where n is the number of points at the time the operation is performed. The

```

// find place for y starting at root
int i = 1, // current node of heap
    ci = 2; // child of i
while (ci <= LastNode) { // find place to put y
    // heap[ci].left should be smaller child of i
    if (ci < LastNode &&
        heap[ci].left > heap[ci+1].left) ci++;

    // can we put y in heap[i]?
    if (y <= heap[ci].left) break; // yes

    // no
    heap[i].left = heap[ci].left; // move child up
    if (y > heap[ci].right)
        Swap(y, heap[ci].right);
    i = ci; // move down a level
    ci *= 2;
}
if (i == LastNode && CurrentSize % 2)
    heap[LastNode].left = heap[LastNode].right;
else heap[i].left = y;

return *this;
}

```

Program 9.20 Deleting the min element (concluded)

points that lie outside a given interval can be reported in $\Theta(K)$ time, where K is the number of points to be reported. That is, the interval heap structure is optimal to within a constant factor for this operation.

Suppose we have the 19 points that are in the interval heap of Figure 9.14(i) and we are to report all points outside the interval $I = [4,30]$. We start at the root and see if either of the points stored here are outside I . If neither point is outside I then we terminate because the intervals below the root are contained in the root interval and so cannot contain points outside I . In our example, only point 2 is outside I and it is output. Next we repeat this examination process at the children of the root. Examining the left child, we see that it contains a point outside I . This point, 3, is output and we must examine the children of the left child of the root. The first node with interval $[8,16]$ is examined. Since it contains no point that is outside I , its subtrees are not examined. When the node with interval $[4,10]$ is examined, no point is reported and so its subtrees are not to be examined. Only the right subtree of the root remains to be examined.

```

template<class T>
IntervalHeap<T>& IntervalHeap<T>::DeleteMax(T& x)
{ // Set x to max element and delete
  // max element from interval heap.
  if (CurrentSize == 0)
    throw OutOfBounds(); // empty

  x = heap[1].right; // max element

  // restructure max heap part
  int LastNode = CurrentSize / 2 + CurrentSize % 2;
  T y; // element removed from last node
  if (CurrentSize % 2) { // size is odd
    y = heap[LastNode].left;
    LastNode--;
  }
  else { // size is even
    y = heap[LastNode].right;
    heap[LastNode].right = heap[LastNode].left;
  }
  CurrentSize--;
}

```

Program 9.21 Deleting the max element (continues)

Since the root of this subtree reports no point, its subtrees are not examined. Figure 9.15 shows the nodes that are examined by the above procedure.

We observe that each examined node fits into one of the following categories:

- One or two points are reported from this node.
- No point is reported from this node. However, if this node is not the root, its parent reports at least one point. Consequently, the number of nodes that fit into this category is at most twice the number of points reported overall (an exception is when zero points are reported, in this case the number of nodes examined is at most 1).

The total number of nodes examined is $\Theta(K)$, where K is the number of points that lie outside the given interval. Each node takes $\Theta(1)$ time to examine. Therefore, the time need to report all points outside an interval is $\Theta(K)$

```
// find place for y starting at root
int i = 1, // current node of heap
    ci = 2; // child of i
while (ci <= LastNode) { // find place to put y
    // heap[ci].right should be larger child of i
    if (ci < LastNode &&
        heap[ci].right < heap[ci+1].right) ci++;

    // can we put y in heap[i]?
    if (y >= heap[ci].right) break; // yes

    // no
    heap[i].right = heap[ci].right; // move child up
    if (y < heap[ci].left)
        Swap(y, heap[ci].left);
    i = ci; // move down a level
    ci *= 2;
}
heap[i].right = y;

return *this;
}
```

Program 9.21 Deleting the max element (concluded)

9.7.8 Reference

Interval heaps were invented by Leeuwen and Wood. Their original paper “Interval Heaps,” *The Computer Journal*, 36, 3, 1993, 209–216 also describes extensions to the basic interval heap structure described here.

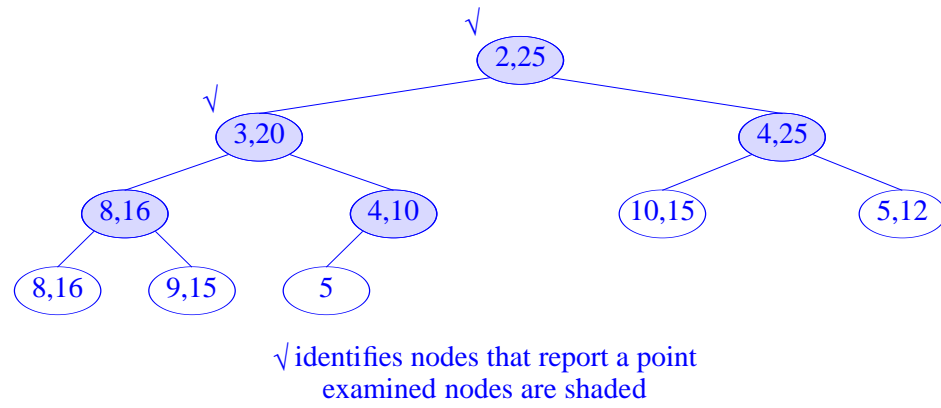


Figure 9.15 Nodes examined when reporting points outside $[4,30]$