Solutions to Exercises in Chapter 13

13.2 The effect of successive additions to a priority queue represented by a sorted SLL, an unsorted SLL, and a heap are shown in Figures S13.1, S13.2, and S13.3, respectively.

13.3 The effect of successive additions to a priority queue represented by a sorted SLL, an unsorted SLL, and a heap are shown in Figures S13.1, S13.2, and S13.3, respectively.

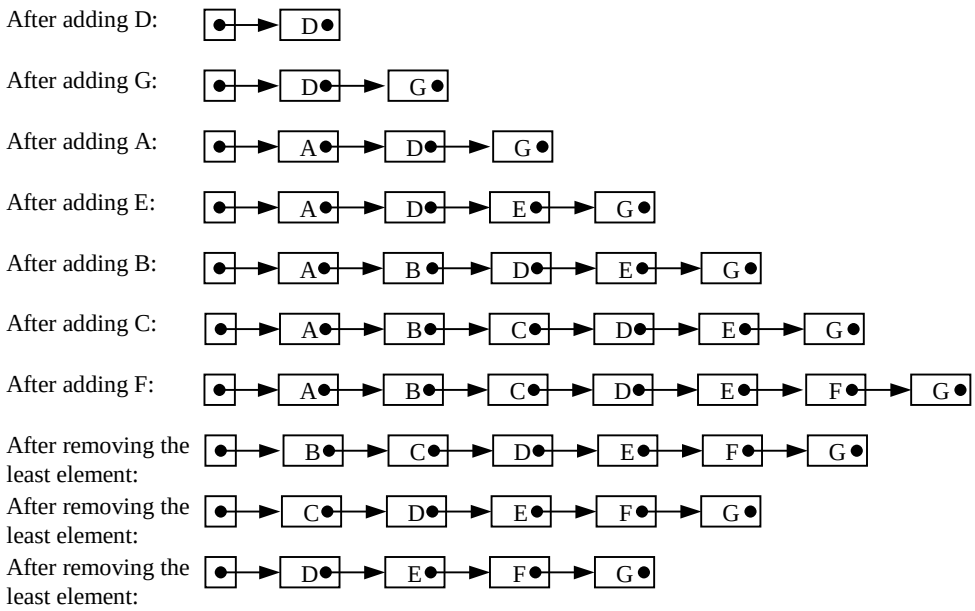

Figure S13.1 Effect of successive additions and removals on a priority queue represented by a sorted SLL.

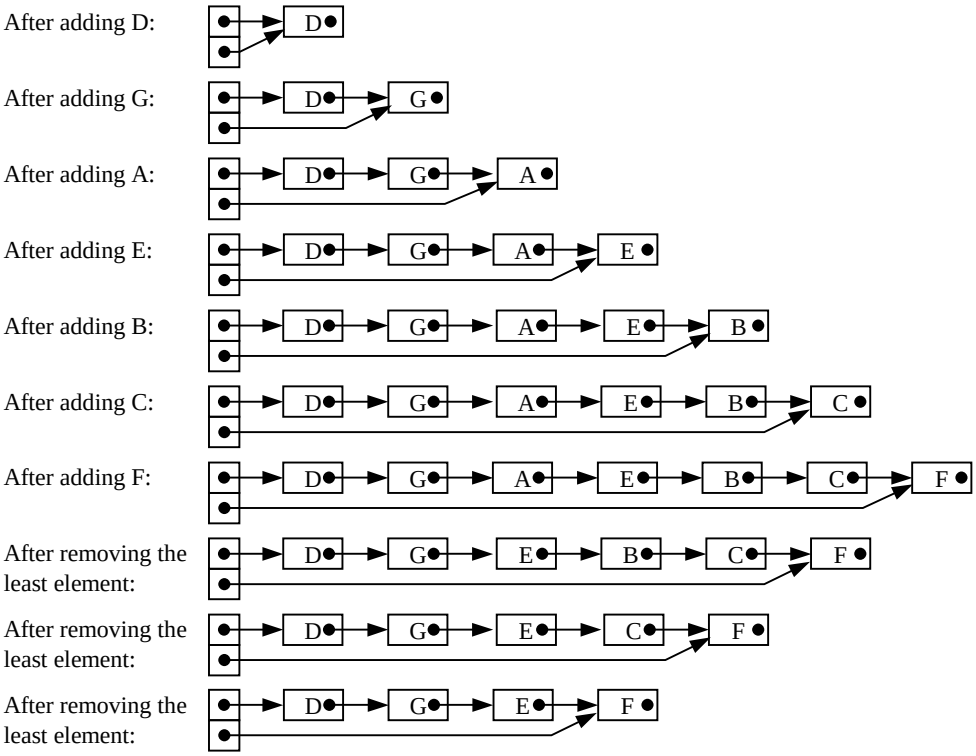

Figure S13.2 Effect of successive additions and removals on a priority queue represented by an unsorted SLL.

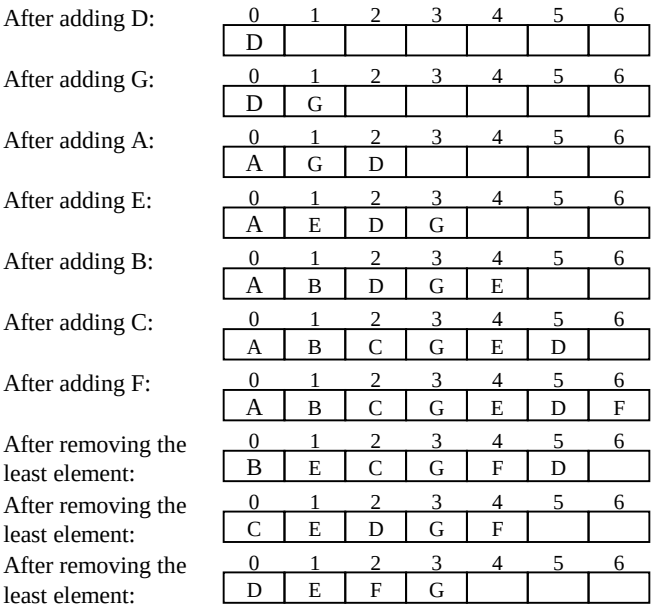

Figure S13.3 Effect of successive additions and removals on a priority queue represented by a heap (with *maxlength* = 7).

13.4 An implementation of priority queue using unsorted SLLs is shown in Program S13.4. This implementation does achieve the time complexities shown in Table 13.7.

```
public class UnsortedLinkedPriorityQueue
       implements PriorityQueue {

  // Each UnsortedLinkedPriorityQueue object is a priority queue
  // whose elements are Comparable objects.

  // This priority queue is represented as follows: length is the number of
  // elements; first and last are links to the first and last nodes of an
  // SLL containing the elements in order of addition.
  private SLLNode first, last;
  private int length;

  /////////////  Constructor  /////////////

  public UnsortedLinkedPriorityQueue () {
  // Construct a priority queue, initially empty.
    first = last = null;
    length = 0;
  }

  /////////////  Accessors  /////////////

  public boolean isEmpty () {
  // Return true if and only if this priority queue is empty.
    return (length == 0);
  }

  public int size () {
  // Return this priority queue's length.
    return length;
  }

  public Comparable getLeast () {
  // Return the least element in this priority queue, or throw a
  // NoSuchElementException if this priority queue is empty.
  // (If there are several equal least elements, return any one of them.)
    if (first == null)
      throw new NoSuchElementException();
    Comparable leastElem = (Comparable)first.element;
    for (SLLNode curr = first.succ; curr != null;
         curr = curr.succ) {
      Comparable currElem = (Comparable)curr.element;
      int comp = currElem.compareTo(leastElem);
      if (comp < 0)  leastElem = currElem;
    }
    return leastElem;
  }

  /////////////  Transformers  /////////////

  public void clear () {
  // Make this priority queue empty.
    first = last = null;
    length = 0;
  }
```

Program S13.4 Implementation of priority queues using unsorted SLLs *(continued on next page)*.

```
  public void add (Comparable elem) {
  // Add elem to this priority queue.
     SLLNode newNode = new SLLNode(elem, null);
     if (first == null)
       first = newNode;
     else
       last.succ = newNode;
     last = newNode;
     length++;
  }

  public Comparable removeLeast () {
  // Remove and return the least element in this priority queue, or throw
  // a NoSuchElementException if this priority queue is empty.
  // (If there are several equal least elements, remove the same element
  // that would be returned by getLeast.)
     if (first == null)
       throw new NoSuchElementException();
     Comparable leastElem = (Comparable)first.element;
     SLLNode leastPred = null;
     for (SLLNode curr = first.succ, pred = first;
          curr != null;
          pred = curr, curr = curr.succ) {
       Comparable currElem = (Comparable)curr.element;
       int comp = currElem.compareTo(leastElem);
       if (comp < 0) {
          leastElem = currElem;
          leastPred = pred;
       }
     }
     if (leastPred == null) {
       first = first.succ;
       if (first == null)  last == null;
     } else {
       leastPred.succ = leastPred.succ.succ;
       if (leastPred.succ == null)  last = leastPred;
     }
     length--;
     return leastElem;
  }

}
```

Program S13.4 Implementation of priority queues using unsorted SLLs *(continued)*.

13.6 Modified interface for extended priority queues:

```
public interface ExtendedPriorityQueue {

   // Each ExtendedPriorityQueue object is a priority queue
   // whose elements are Comparable objects, with the property that
   // both least and greatest elements can be accessed..

   /////////////  Accessors  ////////////

   …

   public Comparable getLeast ();
   // Return the least element in this priority queue, or throw a
   // NoSuchElementException if this priority queue is empty.
   // (If there are several equal least elements, return any one of them.)
```

```
    public Comparable getGreatest ();
    // Return the greatest element in this priority queue, or throw a
    // NoSuchElementException if this priority queue is empty.
    // (If there are several equal greatest elements, return any one of
    // them.)

    //////////// Transformers ////////////

    …

}
```

In an implementation using sorted SLLs, the `getGreatest` operation must return the last element in the SLL. It is convenient to maintain links to both first and last nodes.

In an implementation using unsorted SLLs, the `getGreatest` operation must visit all nodes of the SLL, just like the `getLeast` operation.

In an implementation using heaps, the `getGreatest` operation need visit only the leaf nodes. If the heap has size n, the leaf nodes are those in positions $n/2$ through $n–1$. The number of leaf nodes is at most $(n+1)/2$. Therefore `getGreatest` has time complexity $O(n)$.

The time complexities of `getLeast` and `getGreatest` are summarized in Table S13.5.

Table S13.5 Implementation of extended priority queues using sorted SLLs, unsorted SLLs, and heaps: summary of time complexities (where n is the length of the priority queue).

Operation	Time complexity (sorted SLL)	Time complexity (unsorted SLL)	Time complexity (heap)
`getLeast`	$O(1)$	$O(n)$	$O(1)$
`getGreatest`	$O(1)$	$O(n)$	$O(n)$

13.7 A priority queue could be used (inefficiently!) to implement a last-in-first-out sequence as follows. Attach a timestamp to each element as it is added. Use the *negated* timestamp as the element's priority. Thus the element with the latest timestamp will be the first to be removed.

13.8 A priority queue could be used (inefficiently!) to implement a first-in-first-out sequence as follows. Attach a timestamp to each element as it is added. Use the timestamp itself as the element's priority. Thus the element with the earliest timestamp will be the first to be removed.

13.14 Modified interface for dynamic priority queues:

```
public interface DynamicPriorityQueue {

    // Each DynamicPriorityQueue object is a priority queue
    // whose elements are Comparable objects, with the property
    // that elements can be changed dynamically.

    //////////// Accessors ////////////

    …

    //////////// Transformers ////////////

    …
```

```
    public void setElement (Comparable oldElem,
                            Comparable newElem);
    // Replace the element in this priority queue equal to oldElem by
    // newElem. Throw a NoSuchElementException if this
    // priority queue contains no such element.

}
```

An implementation using heaps is outlined in Program S13.7. The `setElement` operation first searches the heap for `oldElem`, and then finds a suitable position for `newElem`.

To search the heap for `oldElem`, the simplest algorithm is a linear search of the array representing the heap. A better algorithm traverses the heap in pre-order, terminating when it finds an element either equal to `oldElem` (successful search) or greater than `oldElem` (unsuccessful search). In the latter case, there is no point in searching the subtrees, since they must contain only elements that are also greater than `oldElem`.

To find a suitable position for `newElem`, proceed as follows. If `newElem` is less than `oldElem`, repeatedly swap `newElem` with its parent element as long as the latter is greater than `newElem`. (This can be implemented by code taken from the `add` operation.) If `newElem` is greater than `oldElem`, repeatedly swap `newElem` with its child element (or the lesser of the two child elements) as long as the child element is less than `newElem` and there is at least one child to compare. (This can be implemented by code taken from the `removeLeast` operation.) If `newElem` is equal to `oldElem`, there is nothing to be done.

The `setElement` operation's two stages have time complexity $O(n)$ and $O(\log n)$, respectively, so the operation as a whole has time complexity $O(n)$.

The other priority queue operations are unaffected. The time complexities are summarized in Table S13.6.

Table S13.6 Implementation of dynamic priority queues using sorted SLLs, unsorted SLLs, and heaps: summary of time complexities (where n is the length of the priority queue).

Operation	Time complexity (sorted SLL)	Time complexity (unsorted SLL)	Time complexity (heap)
`add`	$O(n)$	$O(1)$	$O(\log n)$
`removeLeast`	$O(1)$	$O(n)$	$O(\log n)$
`setElement`	$O(n)$	$O(n)$	$O(n)$
`getLeast`	$O(1)$	$O(n)$	$O(1)$

```
public class HeapDynamicPriorityQueue
             implements DynamicPriorityQueue {

  // Each HeapDynamicPriorityQueue object is a priority queue
  // whose elements are Comparable objects, with the property that
  // elements can be changed dynamically.

  // This priority queue is represented as follows: the subarray
  // elems[0...length–1] contains the priority queue's elements,
  // arranged in such a way that elems[(p–1)/2] is less than or equal
  // to elems[p] for every p > 0.
  private Comparable[] elems;
  private int length;

  //////////// Constructor ////////////

  ...

  //////////// Accessors ////////////

  ...

  //////////// Transformers ////////////

  ...

  public void add (Comparable elem) {
  // Add elem to this priority queue.
    if (length == elems.length)  expand();
    length++;
    locateAtOrAbove(length-1, elem);
  }

  public Comparable removeLeast () {
  // Remove and return the least element in this priority queue, or throw
  // a NoSuchElementException if this priority queue is empty.
  // (If there are several equal least elements, remove the same element
  // that would be returned by getLeast.)
    if (length == 0)
      throw new NoSuchElementException();
    Comparable least = elems[0];
    Comparable last = elems[--length];
    locateAtOrBelow(0, last);
    return least;
  }

  public void setElement (Comparable oldElem,
                Comparable newElem) {
  // Replace the element in this priority queue equal to oldElem by
  // newElem. Throw a NoSuchElementException if this
  // priority queue contains no such element.
    int hole = search(oldElem, 0);
    if (hole < 0)
      throw new NoSuchElementException();
    int comp = newElem.compareTo(oldElem);
    if (comp < 0)
      locateAtOrAbove(hole, newElem);
    } else if (comp > 0)
      locateAtOrBelow(hole, newElem);
    }
  }
```

Program S13.7 Outline implementation of dynamic priority queues using heaps *(continued on next page)*.

```
//////////// Auxiliary methods ////////////

…

private void locateAtOrAbove (int hole,
                Comparable newElem) {
// Find a position for newElem, in position hole or in its parent or in a
// grandparent or ….
     for (;;) {
       if (hole == 0) {
         elems[hole] = newElem;
         return;
       }
       int parent = (hole-1)/2;
       if (elems[parent].compareTo(newElem) <= 0) {
         elems[hole] = newElem;
         return;
       } else {
         elems[hole] = elems[parent];
         hole = parent;
       }
     }
}

private void locateAtOrBelow (int hole,
                Comparable newElem) {
// Find a position for newElem, in position hole or in a child or in a
// grandchild or ….
     for (;;) {
       int left = 2*hole + 1, right = 2*hole + 2;
       int child;
       if (left > length) {     // hole has no child
         elems[hole] = newElem;
         return;
       }
       else if (right > length) // hole has no right child
         child = left;
       else                     // hole has two children
         child =
              (elems[left].compareTo(elems[right])
                <= 0 ? left : right);
       if (newElem.compareTo(elems[child]) <= 0) {
         elems[hole] = newElem;
         return;
       } else {
         elems[hole] = elems[child];
         hole = child;
       }
     }
}
```

Program S13.7 Outline implementation of dynamic priority queues using heaps *(continued on next page)*.

```
    private int search (Comparable target, int top) {
    // Return the position of an element that is equal to target in the subtree
    // of this heap whose topmost node is at position top. Return –1 if there is
    // no such element.
      int comp = target.compareTo(elems[top]);
      if (comp == 0)
        return top;
      else if (comp < 0)
        return -1;
      else { // comp > 0
        int left = 2*hole + 1, right = 2*hole + 2;
        int pos = search(target, left);
        if (pos < 0)  pos = search(target, right);
        return pos;
    }
  }
```

Program S13.7 Outline implementation of dynamic priority queues using heaps *(continued)*.