Chapter 13: Sorting

Exercises 13.1

1. After first pass, elements of x are: 10 50 70 30 40 60.
   After second pass: 10 30 70 50 40 60.

2. (a) 60 70 50 40 20 10
   70 60 50 40 20 10

   (b) 3, since no interchanges occur on the third pass.

   (c) Worst case is when elements are in reverse order.

3. (a) After x [4] is positioned: 20 30 40 60 10 50
   After x [5] is positioned: 10 20 30 40 60 50

   (b) If the list is in increasing order; no interchanges are required.

4. (a) After one pass: 10 50 60 30 40 70
   After two passes: 10 30 40 50 60 70

   (b) Function to perform double-ended selection sort:

```
template <typename ElementType>
void doubleEndedSelectionSort(ElementType x[], int n)
/*-----------------------------------------------------------------------
  Sort a list into ascending order using double-ended selection sort.

  Precondition:  array x contains n elements.
  Postcondition: The n elements of array have been sorted into
      ascending order.
 ----------------------------------------------------------------------*/
{
  int minValue, maxValue, minPos, maxPos;

  for (int i = 1; i <= n/2; i++)
  {
    minValue = maxValue = x[i];
    minPos = maxPos = i;

    // find min and max values among x[i], . . ., x[n]
    for (int j = i+1; j <= n-i+1; j++)
    {
      if (x[j] < minValue)
      {
        minValue = x[j];
        minPos = j;
      }

      if (x[j] > maxValue)
      {
        maxValue = x[j];
        maxPos = j;
      }
    }

    // make sure that positioning min value doesn't overwrite max
    if (i == maxPos)
      maxPos = minPos;

    x[minPos] = x[i];
    x[i] = minValue;
    x[maxPos] = x[n-i+1];
    x[n-i+1] = maxValue;
  }
}
```

(c) Computing time is O(n^2).

5. (a) After step 1 executes, x[i]'s are: 30 50 90 10 60 70 20 100 80 40

There are 5 passes for step 2. x[i]'s are:

After first pass:	10	50	40	20	60	70	30	90	80	100
After second pass:	10	20	40	30	60	70	50	80	90	100
After third pass:	10	20	30	40	60	70	50	80	90	100
After fourth pass:	10	20	30	40	50	60	70	80	90	100
After fifth pass:	10	20	30	40	50	60	70	80	90	100

(b) The functions below implement Min-Max Sort.

```
template <typename ElementType>
void swap(ElementType x[], int a, int b)
{
  int temp = x[a];
  x[a] = x[b];
  x[b] = temp;
}
```

```
template <typename ElementType>
void minMaxSort(ElementType x[], int n)
{
  // Create rainbow pattern
  for (int i = 1; i <= n/2; i++)
    if (x[i] > x[n + 1 - i])
      swap(x, i, n + 1 - i);


  // Find smallest in first half of list
  for (int i = 1; i <= n/2; i++)
  {
    int minPos = i;
    int minValue = x[i];

    for (int j = i + 1; j <= n/2; j++)
      if (x[j] < minValue)
      {
        minPos = j;
        minValue = x[j];
      }
    // Swap smallest with first element
    x[minPos] = x[i];
    x[i] = minValue;

    // Find largest in last half of list
    int maxPos = n + 1 - i;
    int maxValue = x[maxPos];

    for (int j = n/2 + 1; j <= n + 1 - i; j++)
      if (x[j] > maxValue)
      {
        maxPos = j;
        maxValue = x[j];
      }
    // Swap largest with last element
    x[maxPos] = x[n + 1 - i];
    x[n + 1 - i] = maxValue;

    // Recreate rainbow pattern
    if (x[minPos] > x[n + 1 - minPos])
      swap(x, minPos, n + 1 - minPos);
    if (x[maxPos] < x[n + 1 - maxPos])
      swap(x, maxPos, n + 1 - maxPos);
  }
}
```

6.

```
// Recursive helper function for recSelectionSort() so
// it has same signature as other sorting functions.

template <typename ElementType>
void recSelectionSortAux(ElementType x[], int n, int first)
{
  if (first < n)
  {
    int minPos = first;
    int minValue = x[first];

    for (int j = first + 1; j <= n; j++)
      if (x[j] < minValue)
      {
        minPos = j;
        minValue = x[j];
      }

    x[minPos] = x[first];
    x[first] = minValue;

    recSelectionSortAux(x, n, first + 1);
  }
}

/* recursive SelectionSort */
template <typename ElementType>
void recSelectionSort(ElementType x[], int size)
{ recSelectionSortAux(x, size, 0); }
```

7.

```
// Recursive helper function for recBubbleSort() so
// it has same signature as other sorting functions.
template <typename ElementType>
void recBubbleSortAux(ElementType x[], int numCompares)
{
  if (numCompares > 0)
  {
    int last = 1;
    for (int i = 1; i <= numCompares; i++)
      if (x[i] > x[i + 1])
      {
        int temp = x[i];
        x[i] = x[i + 1];
        x[i + 1] = temp;
        last = i;
      }
    recBubbleSortAux(x, last - 1);
  }
}

/* recursive BubbleSort */
template <typename ElementType>
void recBubbleSort(ElementType x[], int n)
{ recBubbleSortAux(x, n); }
```

8. Bubblesort algorithm for a linked list with head node:

```
1. If first->next == 0      // empty list
      terminate this algorithm.
   Else continue with the following:
2. Initialize lastPtr = 0, lastSwap = first.
3. While (lastSwap->next != 0)
      lastSwap = lastSwap->next.    // Initially, put lastSwap at next-to-last node
4. While (lastSwap != first->next)
      a. ptr = first->next
      b. while (ptr != lastSwap )
              i. if (ptr->data > ptr->next->data)
                      (1) swap(ptr->data, ptr->next->data)
                      (2) lastPtr = ptr
             ii. ptr = ptr->next
      c. lastSwap = lastPtr
```

9. (a) Elements of x after each pass:

left to right: 30 80 20 60 70 10 90 50 40 100
right to left: 10 30 80 20 60 70 40 90 50 100
left to right: 10 30 20 60 70 40 80 50 90 100
right to left: 10 20 30 40 60 70 50 80 90 100
left to right: 10 20 30 40 60 50 70 80 90 100
right to left: 10 20 30 40 50 60 70 80 90 100
left to right: 10 20 30 40 50 60 70 80 90 100
right to left: 10 20 30 40 50 60 70 80 90 100

(b) Algorithm for two way bubble sort: $O(n^2)$.

```
Do the following:
     1. Set interchanges1 and interchanges2 to false.

     2. For i = 1 to n – 1 do the following:
          If x[i] > x[i+1] then
               a. Interchange x[i] and x[i+1]
               b. Set interchanges1 to true.
        End for.

     3. If noInterchanges1 is false then
               For i = n – 1 downto 1 do the following:
                     If x[i+1] < x[i] then
                          a. Interchange x[i] and x[i+1]
                          b. Set interchanges2 to true.
               End for.

While interchanges1 or interchanges2 are true.
```

10. Algorithm to insertion sort a linked list with head node pointed to by first.

```
If the list has fewer than 2 elements,
     Terminate this algorithm,
Else do the following:
1. ptr = first                    // points to the predecessor of the place to insert the new node
   predPtr = ptr->next            // points to the predecessor of the next node to be inserted
   nextElPtr = predPtr->next // points to the node of the next element to be inserted

2. While (nextElPtr != 0)
      a. While (ptr->next != nextElPtr && ptr->next->data < nextElPtr->data)
              ptr = ptr->next
         End while
      b. If ( ptr->next != nextElPtr )
                i. predPtr->next = nextElPtr->next
               ii. nextElPtr->next = ptr->next
              iii. ptr->next = nextElPtr.
               iv. nextElPtr = predPtr->next
         Else
               i. predPtr = nextElPtr
              ii. nextElPtr = nextElPtr->next
      c. ptr = first
   End while
```

11. (a) Algorithm for binary insertion sort.

```
For index = 1 to end of array, do
     1. If x[index - 1] > x[index]                    // Adjustment needed
        a. first = 0, last = index, found = false
        b. item = x[index]
        c. while (!found && last - first > 1)         //binary search
             i. loc = (first + last) /2;
            ii. if x[index] == x[loc]
                   found = true
                else if x[index] < x[loc]
                   last = loc;
                else
                   first = loc;
        d. if found
              i. shift array elements x[loc – 1] .. x[index – 1] one position to the right
             ii. set x[loc] = item
           else if item > x[first]
              i. shift array elements x[last] .. x[index – 1] one position to the right
             ii. set x[last] = item
           else
              i. shift array elements x[first] .. x[index – 1] one position to the right
             ii. set x[first] = item
```

(b)	Insert 90:	90	100	60	70	40	20	50	30	80	10
	Insert 60:	60	90	100	70	40	20	50	30	80	10
	Insert 70:	60	70	90	100	40	20	50	30	80	10
	Insert 40:	40	60	70	90	100	20	50	30	80	10
	Insert 20:	20	40	60	70	90	100	50	30	80	10
	Insert 50:	20	40	50	60	70	90	100	30	80	10
	Insert 30:	20	30	40	50	60	70	90	100	80	10
	Insert 80:	20	30	40	50	60	70	80	90	100	10
	Insert 10:	10	20	30	40	50	60	70	80	90	100

There were 27 comparisons of list elements.

(b) There would be 35 comparisons of list elements.

12. (a)

40 10 50 30 80 20 60 70 100 90
10 20 30 40 50 60 70 80 90 100

(b)

```
/*---Incremented Insertion Sort---*/
template <typename ElementType>
void incrInsertSort(ElementType x[], int numElements,
                    int start, int gap)
{
  ElementType nextElement;
  int j;

  for (int i = start + gap; i <= numElements; i += gap)
  {
    // Insert x[i] into its proper position among
    // x[start], x[start + gap], . . .

    nextElement = x[i];
    j = i;

    while (j - gap >= start && nextElement < x[j - gap])
    {
      // Shift element gap positions to right to open a spot
      x[j] = x[j - gap];
      j -= gap;
    }
    // Now drop next Element into the open spot
    x[j] = nextElement;
  }
}
```

```
/*--- ShellSort x[1], x[2], ...., x[numElements] ---*/
template <typename ElementType>
void shellSort(ElementType x[], int numElements)
{
  int gap = 1;
  // Find largest value in incr. seq. <= numElements

  while (gap < numElements)
    gap = 3 * gap + 1;

  gap /= 3;
  while (gap >= 1)
  {
    for (int i = 1; i <= gap ; i++)
      incrInsertSort(x, numElements, i, gap);
    gap /= 3;
  }
}
```

13-14. The following are array-based treesort and supporting routines. Array and linked list implementations are similar. The primary difference is one of traversal: index (for loop) is used for an array. A temporary pointer, continually updated to the contents of the next field (while loop), is used for a linked list.

```
template <typename ElementType>
void treeSort(ElementType x[], int size)
{
  BST<ElementType> tree;
  for (int index = 0; index < size; index++)
    tree.insert(x[index]);

  tree.inOrder(x);
}
```

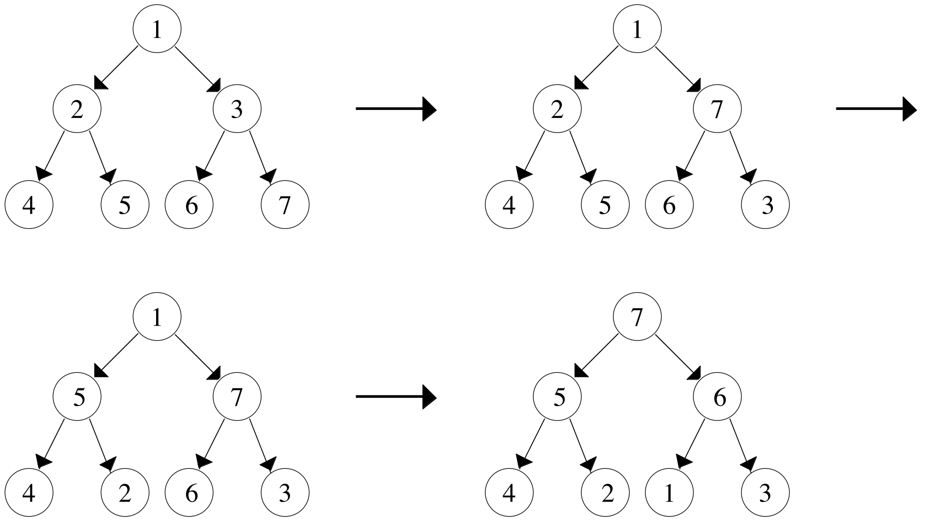
where `inOrder()` is a function member of `BST` defined as follows:

```
template <typename ElementType>
inline void BST<ElementType>::inOrder(ElementType x[])
{ int index = 0; inOrderAux(root, x, index); }

template <typename ElementType>
void BST<ElementType>::inOrderAux(BST<ElementType>::BinNodePointer root,
                                  ElementType x[], int & index)
{
  if (root != 0)
  {
    inOrderAux(root->left, x, index);
    x[index] = root->data;
    index++;
    inOrderAux(root->right, x, index);
  }
}
```

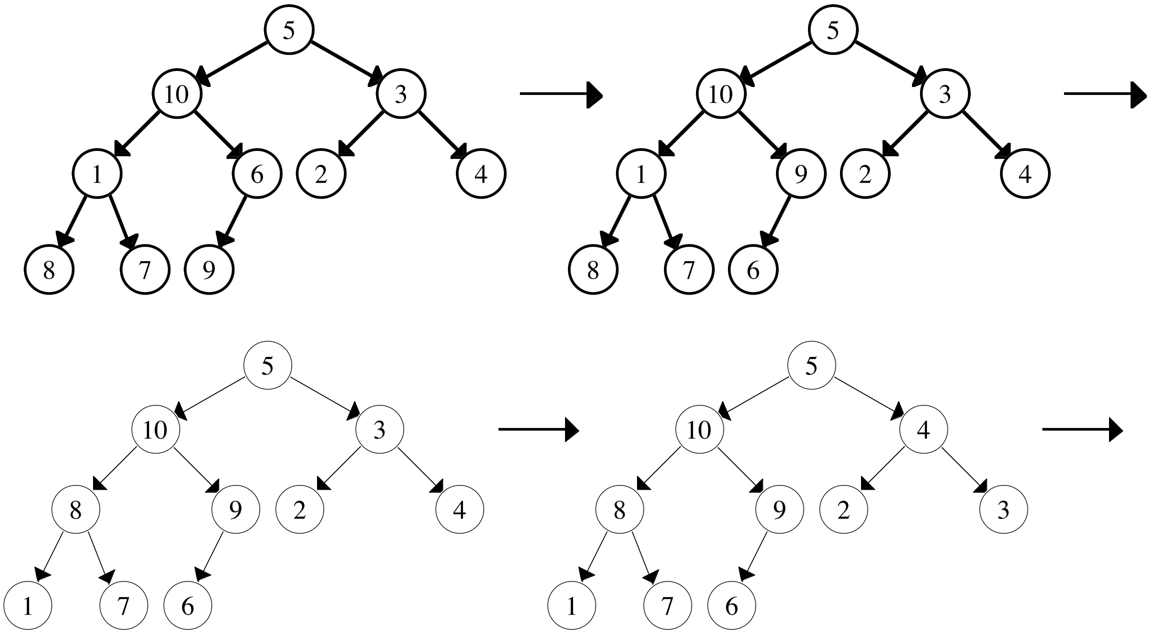
Exercises 13.2

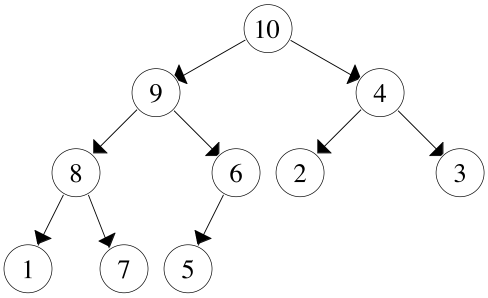
1. The tree is not complete because the next-to-bottom level is not completely full.

2.

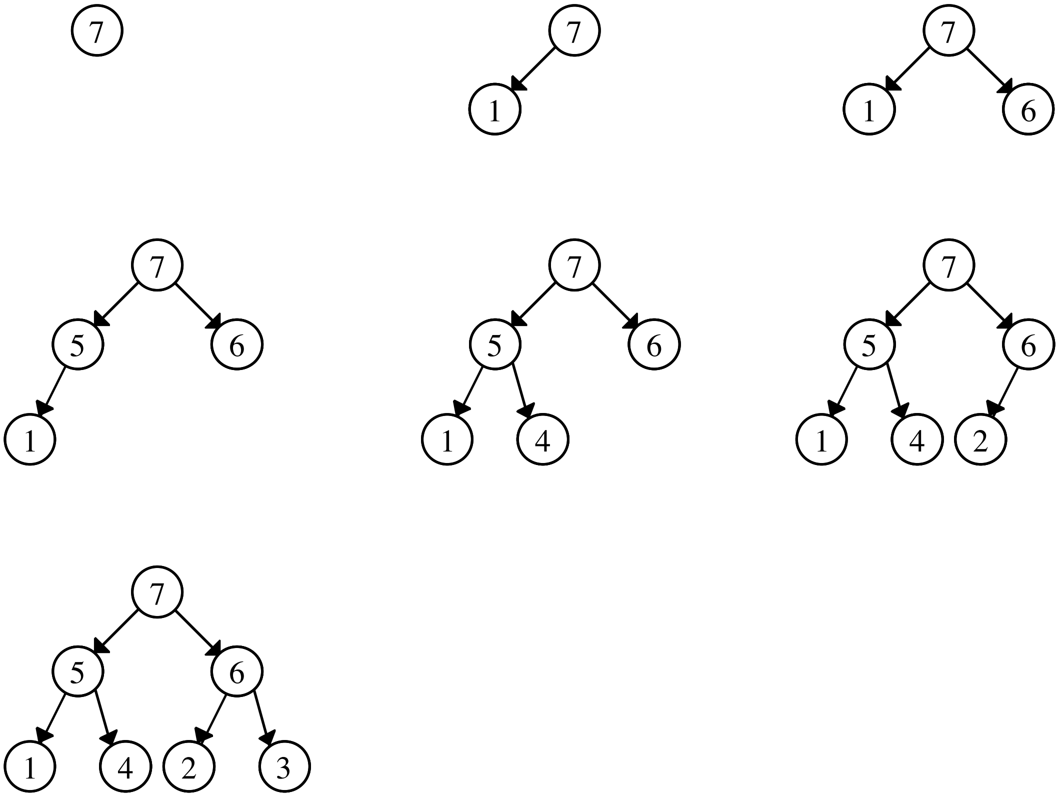


3.

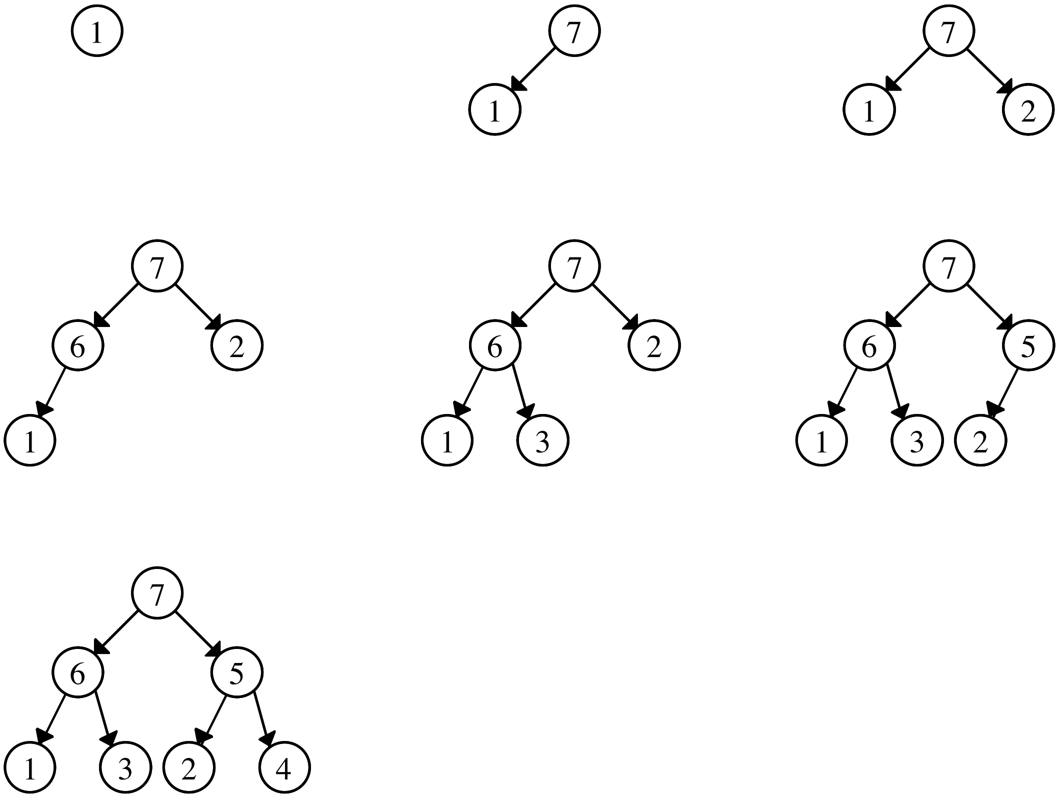




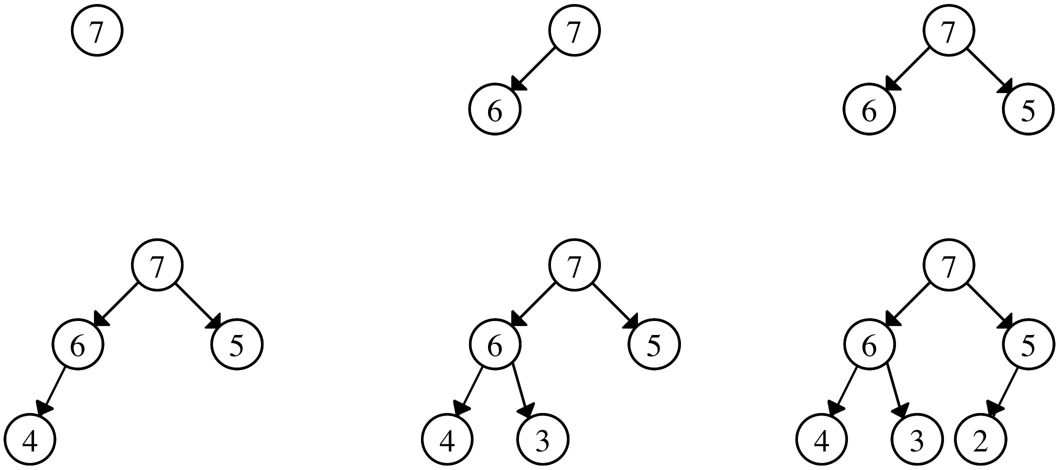
4.

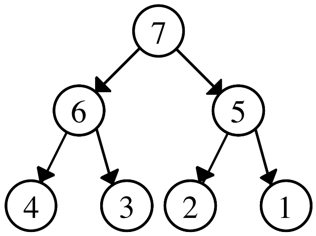


5.

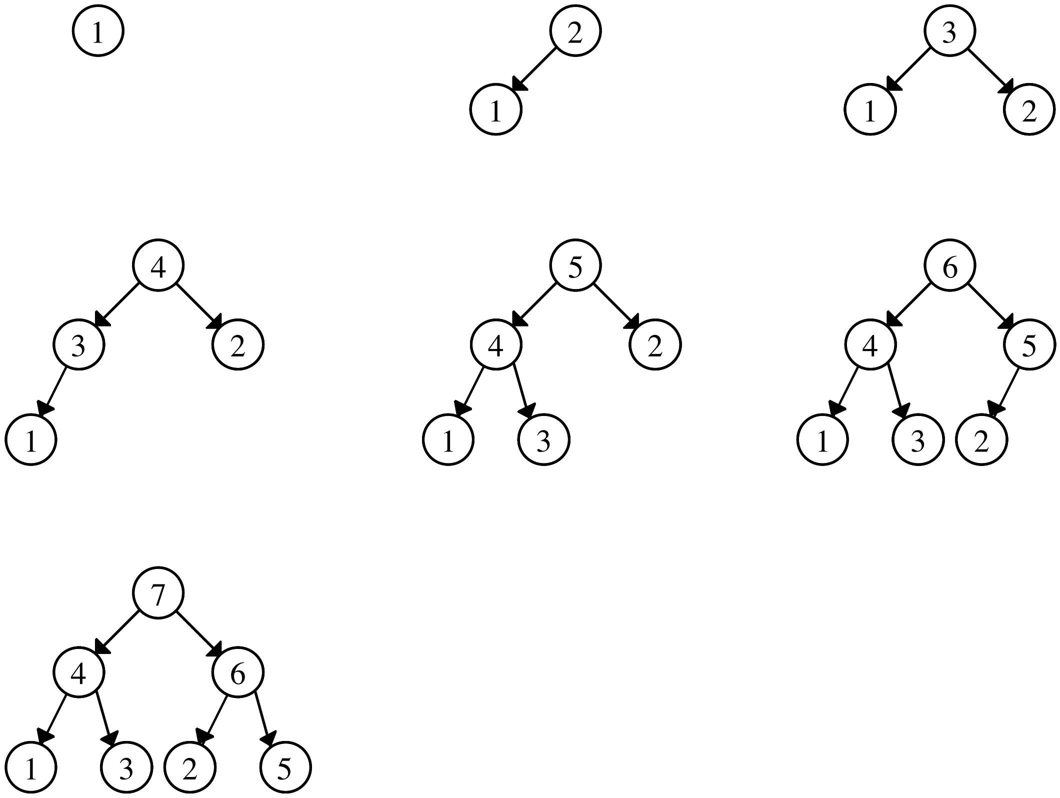


6.

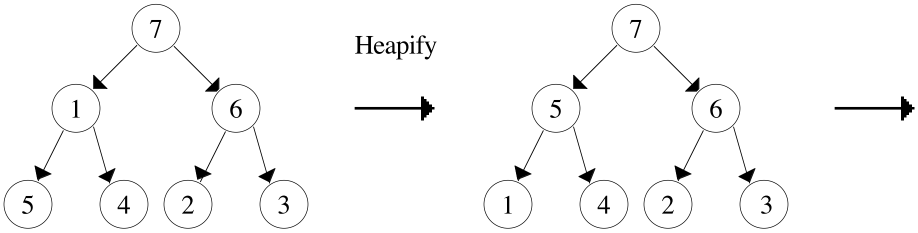


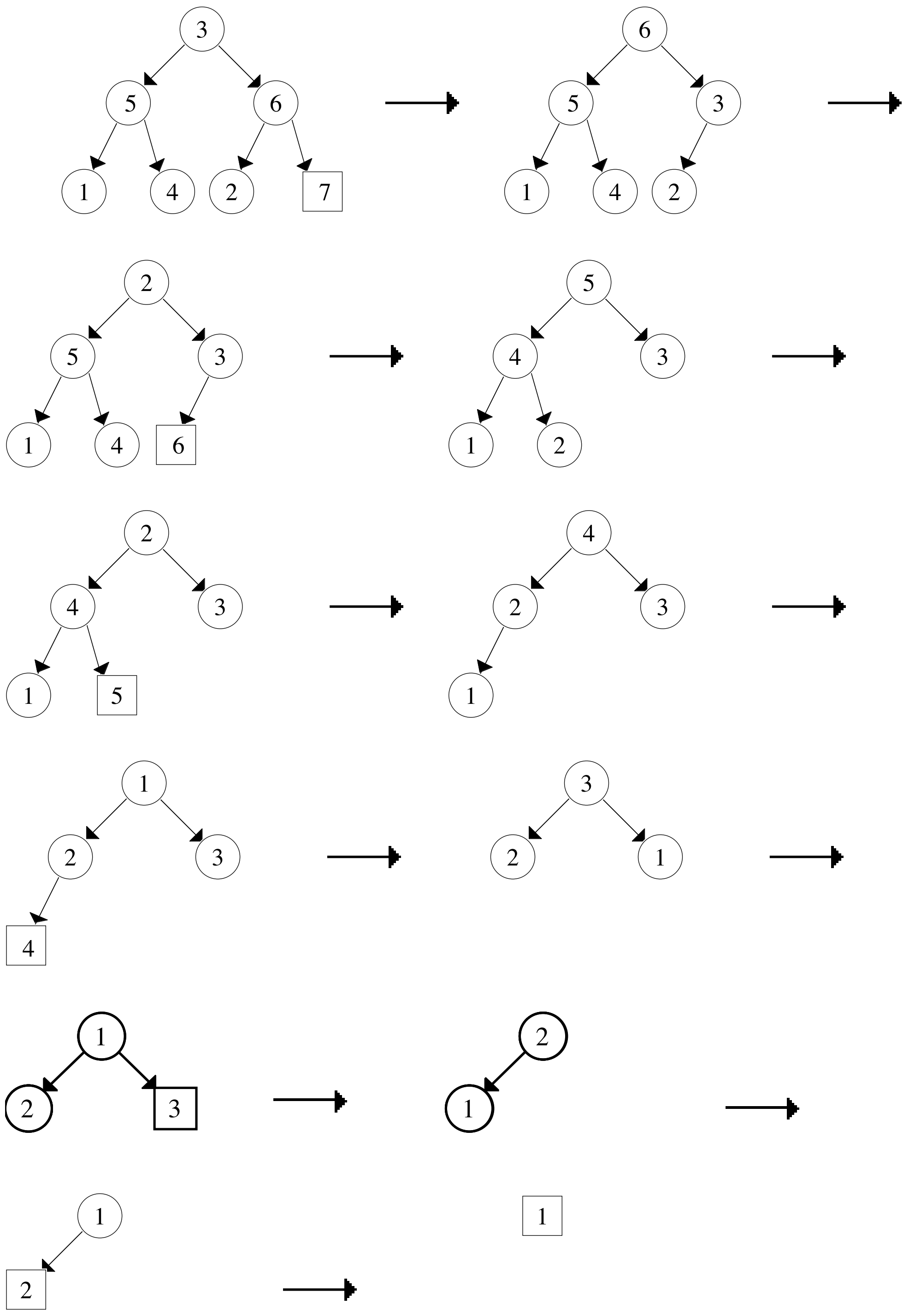

7
6
5
4
3
2
1

7.

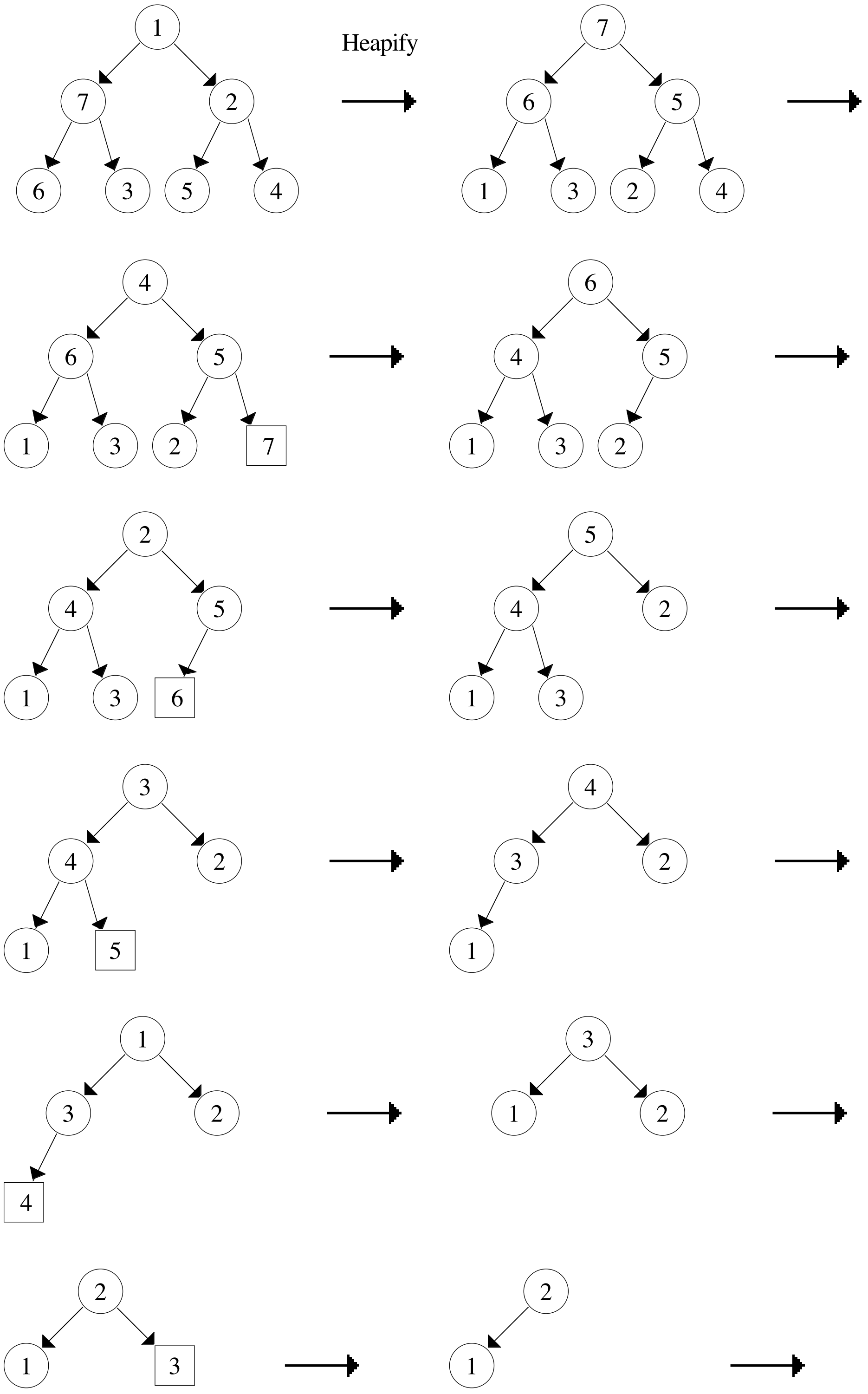

1
2
1
3
1
2
4
3
2
1
5
4
2
1
3
6
4
5
1
3
2
7
4
6
1
3
2
5

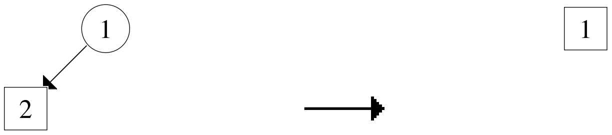
8.


7
1
6
5
4
2
3
Heapify
7
5
6
1
4
2
3

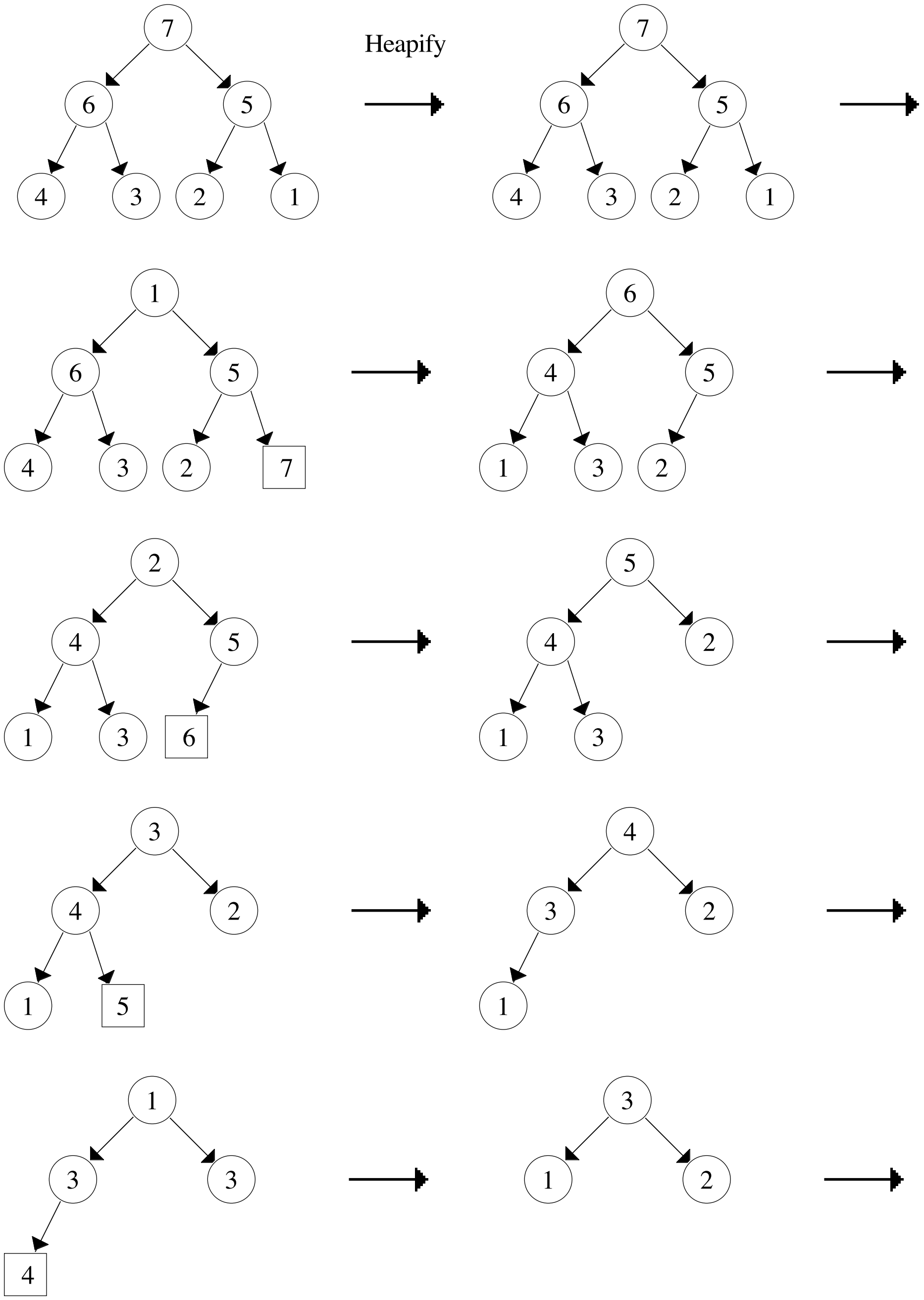

3
5
6
1
4
2
7
6
5
3
1
4
2
2
5
3
1
4
6
5
4
3
1
2
2
4
3
1
5
4
2
3
1
1
2
3
4
3
2
1
1
2
3
2
1
1
2
1

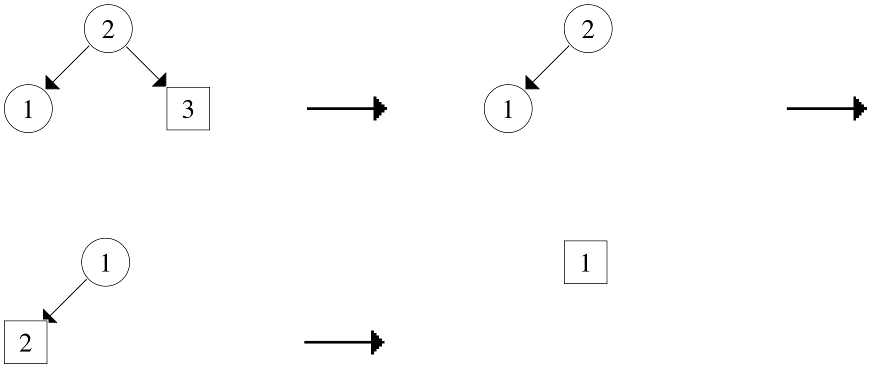
9.


1
7
2
6
3
5
4
Heapify
7
6
5
1
3
2
4
4
6
5
1
3
2
7
6
4
5
1
3
2
2
4
5
1
3
6
5
4
2
1
3
3
4
2
1
5
4
3
2
1
1
3
2
4
3
1
2
2
1
3
2
1

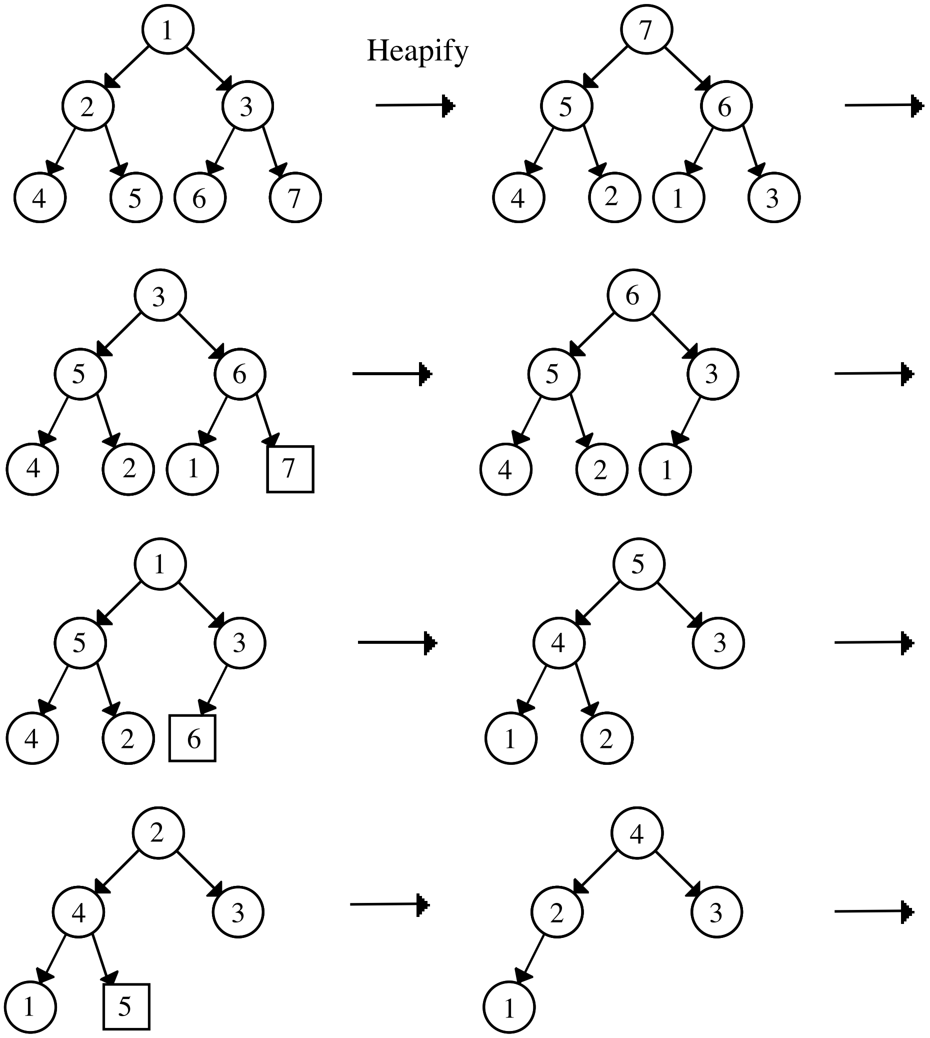

1
2
1

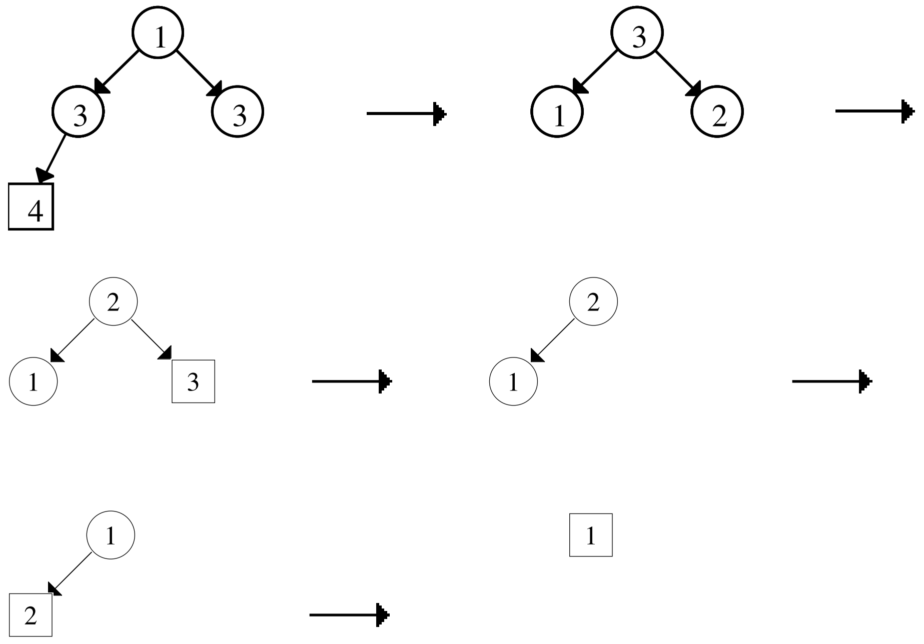
10.


7
6
5
4
3
2
1
Heapify
7
6
5
4
3
2
1
1
6
5
4
3
2
7
6
4
5
1
3
2
2
4
5
1
3
6
5
4
2
1
3
3
4
2
1
5
4
3
2
1
1
3
3
4
3
1
2


2
1
3
2
1
1
2
1

11.


1
2
3
4
5
6
7
Heapify
7
5
6
4
2
1
3
3
5
6
4
2
1
7
6
5
3
4
2
1
1
5
3
4
2
6
5
4
3
1
2
2
4
3
1
5
4
2
3
1



12. Contents of array x after each of the calls:

After first call: 20 15 31 49 67 50   3 10 26
After second call: 20 15 50 49 67 31   3 10 26

(The remaining calls produce:
After third call: 20 67 50 49 15 31   3 10 26
After fourth call: 67 49 50 26 15 31   3 10 26)

13. Contents of array x after each of the calls:

After first call: 88 77 55 66 22 33 44 99
After second call: 77 66 55 44 22 33 88 99

(The remaining calls produce:
After third call: 66 44 55 33 22 77 88 99
After fourth call: 55 44 22 33 66 77 88 99
After fifth call: 44 33 22 55 66 77 88 99
After sixth call: 33 22 44 55 66 77 88 99
After seventh call: 22 33 44 55 66 77 88 99

14-15.

```
#include <iostream>
using namespace std;

const int HEAP_CAPACITY = 127;
template <typename ElementType>
class Heap
{
 public:
  //----- PUBLIC FUNCTION MEMBERS -----
  Heap();
  /*-------------------------------------------------------------------------
   Constructor

   Precondition:  None.
   Postcondition: An empty heap that can store HEAP_CAPACITY elements has
       been constructed.
   ------------------------------------------------------------------------*/

  bool empty() const;
  /*-------------------------------------------------------------------------
   Check if heap is empty.

   Precondition:  None.
   Postcondition: True is returned if heap is empty, false if not.
   ------------------------------------------------------------------------*/

  int getSize() const;
  /*-------------------------------------------------------------------------
   Return number of elements in heap.

   Precondition:  None.
   Postcondition: mySize is returned.
   ------------------------------------------------------------------------*/

  ElementType * getArray();
  /*-------------------------------------------------------------------------
   Return array used to store elements of heap.

   Precondition:  None.
   Postcondition: myArray is returned.
   ------------------------------------------------------------------------*/

  void insert(ElementType item);
  /*-------------------------------------------------------------------------
   Insert operation

   Precondition:  mySize < HEAP_CAPACITY.
   Postcondition: item has been inserted into the heap so the result is
       still a heap, provided there is room in myArray; otherwise, a
       heap-full message is displayed and execution is terminated.
   ------------------------------------------------------------------------*/
  ElementType getMax() const;
  /*-------------------------------------------------------------------------
   Retrieve the largest element in the heap.

   Precondition:  Heap is nonempty.
   Postcondition: Largest element is returned if heap is nonempty,
       otherwise a heap-empty message is displayed and a garbage value
       is returned.
   ------------------------------------------------------------------------*/

  void removeMax();
  /*-------------------------------------------------------------------------
   Remove the largest element in the heap.

   Precondition:  Heap is nonempty.
   Postcondition: Largest element is removed if heap is nonempty and result
       is still a heap; Otherwise a heap-empty message is displayed.
   ------------------------------------------------------------------------*/

  void remove(int loc);
  /*-------------------------------------------------------------------------
   Remove the element in location loc.

   Precondition:  1 <= loc <= mySize.
   Postcondition: Element at location loc is removed and result is still a
       heap; otherwise a bad-location message is displayed.
   ------------------------------------------------------------------------*/

 //-- Extra Functions to help visualize heaps
 private:
  //----- DATA MEMBERS -----
  int mySize;
  ElementType myArray[HEAP_CAPACITY];

  //----- PRIVATE FUNCTION MEMBERS -----
  void percolateDown(int r, int n);
  /*-------------------------------------------------------------------------
   Percolate-down operation
   Precondition:  myArray[r], ..., myArray[n] stores a semiheap.
   Postcondition: The semiheap has been converted into a heap.
   ------------------------------------------------------------------------*/

  void heapify();
  /*-------------------------------------------------------------------------
   Heapify operation
   Precondition:  myArray[1], ..., myArray[mySize] stores a complete binary
       tree.
   Postcondition: The complete binary tree has been converted into a heap.
   ------------------------------------------------------------------------*/
};


//--- Definition of constructor
template <typename ElementType>
inline Heap<ElementType>::Heap()
: mySize(0)
{ }
```

```
//--- Definition of empty()
template <typename ElementType>
inline bool Heap<ElementType>::empty() const
{ return mySize == 0; }

//--- Definition of getSize()
template <typename ElementType>
inline int Heap<ElementType>::getSize() const
{ return mySize; }

//--- Definition of getArray()
template <typename ElementType>
inline ElementType * Heap<ElementType>::getArray()
{ return myArray; }

//--- Definition of insert()
template <typename ElementType>
void Heap<ElementType>::insert(ElementType item)
{
   if (mySize >= HEAP_CAPACITY)
   {
      cerr << "No more room in heap -- increase its capacity\n";
      exit(1);
   }
   //else
   mySize++;
   myArray[mySize] = item;
   int loc = mySize,
       parent = loc / 2;

   while (parent >= 1 && myArray[loc] > myArray[parent])
   {
      //-- Swap elements at positions loc and parent
      ElementType temp = myArray[loc];
      myArray[loc] = myArray[parent];
      myArray[parent] = temp;
      loc = parent;
      parent = loc/ 2;
   }
}

//--- Definition of getMax()
template <typename ElementType>
ElementType Heap<ElementType>::getMax() const
{
   if (!empty())
      return myArray[1];
   //else
   cerr << "Heap is empty -- garbage value returned\n";
   ElementType garbage;
   return garbage;
}
```

```
//--- Definition of removeMax()
template <typename ElementType>
void Heap<ElementType>::removeMax()
{
   if (!empty())
      remove(1);
   else
     cerr << "Heap is empty -- no element removed";
}

//--- Definition of remove()
template <typename ElementType>
void Heap<ElementType>::remove(int loc)
{
   if (1 <= loc and loc <= mySize)
   {
      myArray[loc] = myArray[mySize];
      mySize--;
      percolateDown(loc, mySize);
   }
   else
      cerr << "Illegal location in heap: " << loc << endl;
}

//--- Definition of percolateDown()
template <typename ElementType>
void Heap<ElementType>::percolateDown(int r, int n)
{
  int c;
  for (c = 2*r; c <= n; )
  {
    if (c < n && myArray[c] < myArray[c+1] )
      c++;
    // Interchange node and largest child, if necessary
    //  move down to the next subtree.
    if (myArray[r] < myArray[c])
    {
       ElementType temp = myArray[r];
       myArray[r] = myArray[c];
       myArray[c] = temp;
       r = c;
       c *= 2;
    }
    else
      break;
  }
}

//--- Definition of heapify()
template <typename ElementType>
void Heap<ElementType>::heapify()
{
  for (int r = mySize/2; r > 0; r--)
    percolateDown(r, mySize);
}
```

16.

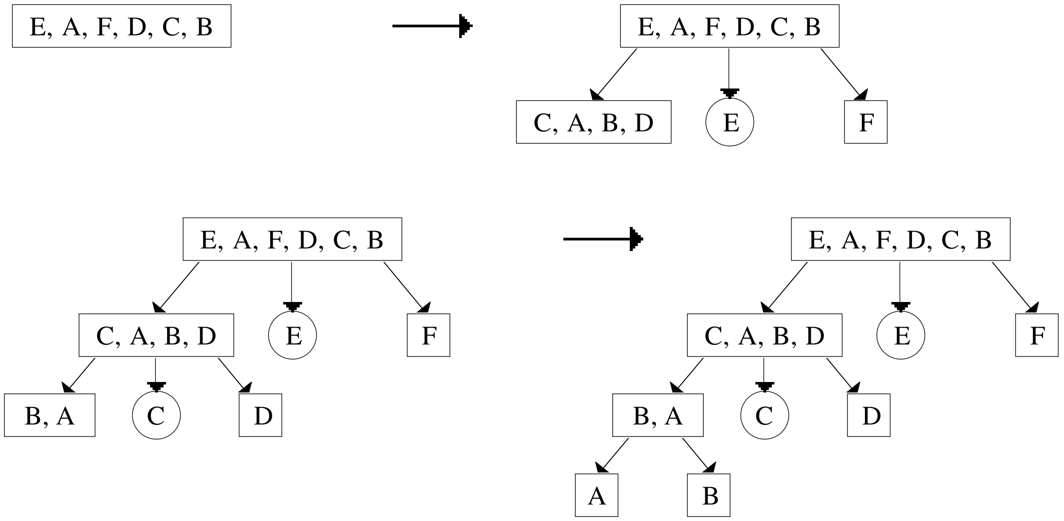
```
#include <iostream>
using namespace std;
#include "Heap.h"     // file containing Heap class template

const int PQ_CAPACITY = HEAP_CAPACITY;
template <typename ElementType>
/*  < is assumed to be defined for type ElementType so that
    x < y if x's priority < y's priority.  */

class PriorityQueue
{
 public:

  PriorityQueue();
  /*--------------------------------------------------------------------------
   Constructor

   Precondition:  None.
   Postcondition: An empty priority queue that can store PQ_CAPACITY elements
       has been constructed.
   -------------------------------------------------------------------------*/

  bool empty();
  /*--------------------------------------------------------------------------
   Check if priority queue is empty.

   Precondition:  None.
   Postcondition: True is returned if priority queue is empty, false if not.
   -------------------------------------------------------------------------*/

  void insert(ElementType item);
  /*--------------------------------------------------------------------------
   Insert operation

   Precondition:  mySize < PQ_CAPACITY.
   Postcondition: item has been inserted into the priority queue so the
       result is still a priority queue, provided there is room in myHeap;
       otherwise, a priority-queue-full message is displayed and execution
       is terminated.
   -------------------------------------------------------------------------*/

  ElementType getMax();
  /*--------------------------------------------------------------------------
   Retrieve the largest (i.e., with highest priority) element in the
       priority queue.

   Precondition:  Priority queue is nonempty.
   Postcondition: Largest element is returned if priority queue is nonempty,
       otherwise a priority-queue-empty message is displayed and a garbage
       value is returned.
   -------------------------------------------------------------------------*/
  void removeMax();
  /*-------------------------------------------------------------------------
   Remove the largest (i.e., with highest priority) element in the
       priority queue.

   Precondition:  Priority queue is nonempty.
   Postcondition: Largest element is removed if priority queue is nonempty
       and result is still a priority queue; Otherwise a priority-queue-
       empty message is displayed.
   ------------------------------------------------------------------------*/

 void display(ostream & out);
 /*-------------------------------------------------------------------------
   Display elements of priority queue.

   Precondition:  ostream out is open.
   Postcondition: Elements of priority queue have been displayed (from
       front to back) to out.
  ------------------------------------------------------------------------*/

 private:
  Heap<ElementType> myHeap;
};

//--- Definition of constructor
template <typename ElementType>
inline PriorityQueue<ElementType>::PriorityQueue()
{
  // Let Heap constructor do the work
}

//--- Definition of empty()
template <typename ElementType>
inline bool PriorityQueue<ElementType>::empty()
{
  myHeap.empty();
}


//--- Definition of insert()
template <typename ElementType>
inline void PriorityQueue<ElementType>::insert(ElementType item)
{
   if(myHeap.getSize() < PQ_CAPACITY)
      myHeap.insert(item);
   else
   {
      cerr << "No more room in priority queue -- increase its capacity\n";
      exit(1);
   }
}
```

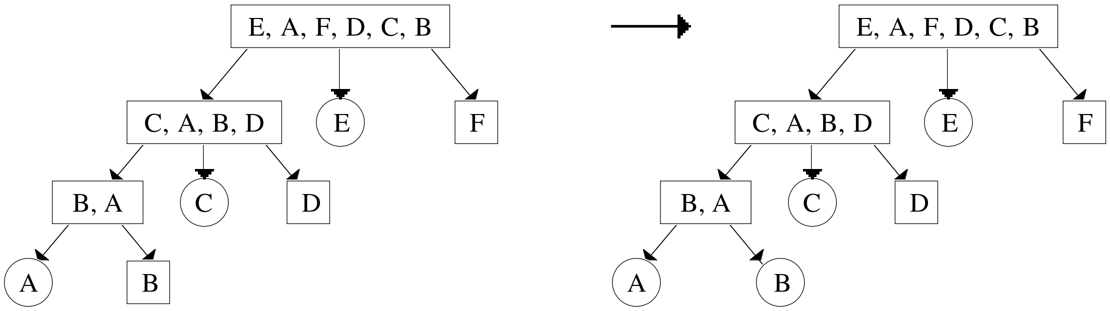
```
//--- Definition of getMax()
template <typename ElementType>
ElementType PriorityQueue<ElementType>::getMax()
{
   return myHeap.getMax();
}

//--- Definition of removeMax()
template <typename ElementType>
void PriorityQueue<ElementType>::removeMax()
{
   myHeap.removeMax();
}


//--- Definition of display()
template <typename ElementType>
void PriorityQueue<ElementType>::display(ostream & out)
{
  for (int i = 1; i <= myHeap.getSize(); i++)
    out << myHeap.getArray()[i] << "  ";
  out << endl;
}
```

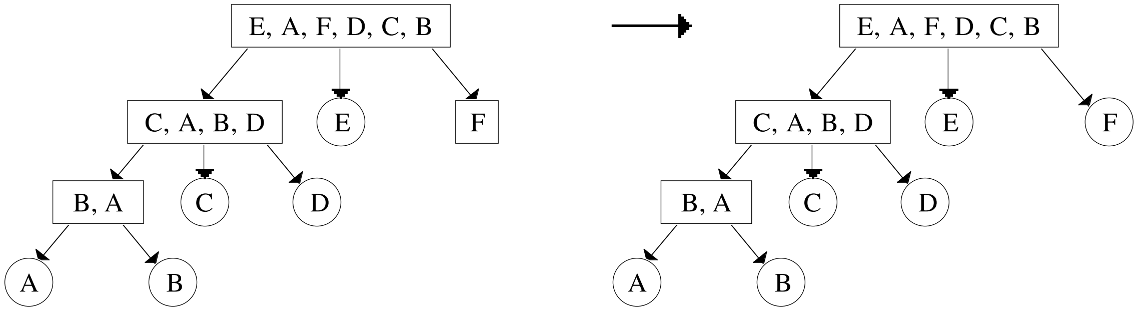
Exercises 13.3

1. The array elements are 10 20 40 30 45 80 60 70 50 90.

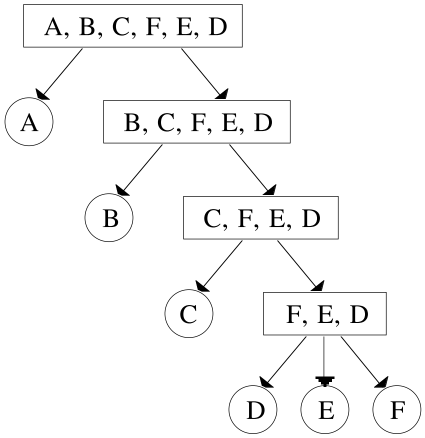
2. The following diagram shows the sequence for trees for Exercise 2. Only the final trees for Exercises 3, 4 and 5 are given.



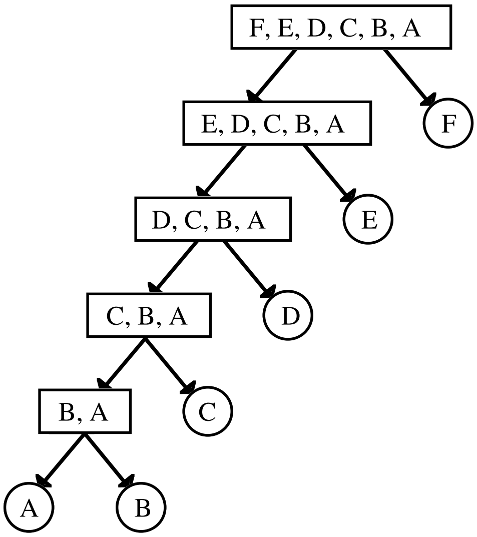

E, A, F, D, C, B
C, A, B, D
E
F
B, A
C
D
A
B
E, A, F, D, C, B
C, A, B, D
E
F
B, A
C
D
A
B


E, A, F, D, C, B
C, A, B, D
E
F
B, A
C
D
A
B
E, A, F, D, C, B
C, A, B, D
E
F
B, A
C
D
A
B

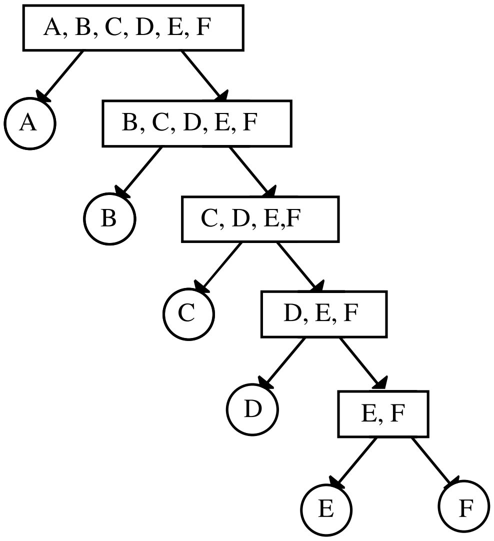
3.


A, B, C, F, E, D
A
B, C, F, E, D
B
C, F, E, D
C
F, E, D
D
E
F

4.



5.



6. In #4, the compound boolean expression prevents the left pointer from going off the right end of the list.

7.

```
template <typename ElementType>
void quicksort(ElementType x[], int first, int last)
/*-----------------------------------------------------------------
  Modified quicksort of array elements x[first], ..., x[last] so
  they are in ascending order.  Small lists (of size < LOWER_BOUND
  are sorted using insertion sort.

  Precondition: < and == are defined for ElementType.
      Note: Client programs call quicksort with first = 1
      and last = n, where n is the list size.
  Postcondition: x[first], ..., x[last] is sorted.
------------------------------------------------------------------*/
{
  const int LOWER_BOUND = 20;
  if (last - first < LOWER_BOUND)   // Small list
    insertionSort(x, first, last);
  else
  {
    int mid = split(x, first, last);
    quicksort(x, first, mid-1);
    quicksort(x, mid+1, last);
  }
}
```

8.

```
template <typename ElementType>
void quicksortAux(ElementType x[], int first, int last)
/*------------------------------------------------------------------
  Auxiliary function that does the actual quicksorting.
-------------------------------------------------------------------*/
{
  const int LOWER_BOUND = 20;
  if (last - first >= LOWER_BOUND)
  {
    int mid = split(x, first, last);
    quicksort(x, first, mid-1);
    quicksort(x, mid+1, last);
  }
}

template <typename ElementType>
void quicksort(ElementType x[], int first, int last)
/*------------------------------------------------------------------
  Modified quicksort of array elements x[first], ..., x[last] so
  they are in ascending order.  Small lists (of size < LOWER_BOUND
  are left unsorted, and a final insertion sort used at the end.

  Precondition: < and == are defined for ElementType.
      Note: Client programs call quicksort with first = 1
      and last = n, where n is the list size.
  Postcondition: x[first], ..., x[last] is sorted.
-------------------------------------------------------------------*/
{
  quicksortAux(x, first, last);
  insertionSort(x, first, last);
}
```

9.

```
template <typename ElementType>
int split(ElementType x[], int first, int last)
/*------------------------------------------------------------------
  Rearrange x[first], ... , x[last] to position pivot.

  Precondition: < and == are defined for ElementType;
      first <= last. Note that this version of split()
      uses the median-of-three rule to select the pivot
  Postcondition: Elements of sublist are rearranged and  pos
      returned so x[first],..., x[pos-1] <= pivot and
      pivot < x[pos+1],..., x[last].
------------------------------------------------------------------*/
{
  int mid = (first + last) / 2;
  ElementType item1 = x[first],
              item2 = x[mid],
              item3 = x[last],
              pivot;
  if ( ( item2 < item1 && item1 < item3 )
    || ( item3 < item1 && item1 < item3 ) )
  {
    pivot = item1;
    mid = first;
  }
  else if ( ( item1 < item2 &&  item2 < item3 )
         || ( item3 < item2 && item2 < item1 ) )
    pivot = item2;
  else
  {
    pivot = item3;
    mid = last;
  }

  // Put pivot in position first
  x[mid] = x[first];
  x[first] = pivot;

  int left = first;
  int right = last;

  while (left < right)
  {
    while (x[right] > pivot)
      right--;
    while (left < right && x[left] <= pivot)
      left++;
    if (left < right)
    // swap elements at positions left and right
    {
      ElementType temp = x[left];
      x[left] = x[right];
      x[right] = temp;
    }
  }

  mid = right;
  x[first] = x[mid];
  x[mid] = pivot;
  return mid;
}
```

10.

```
template <typename ElementType>
void quicksort(ElementType x[], int first, int last)
/*------------------------------------------------------------------
   Nonrecursive version of quicksort to sort array elements
   x[first], ..., x[last] so they are in aascending order.
   Uses a stack to store "recursive" calls.

   Precondition: < and == are defined for ElementType.
       Note: Client programs call quicksort with first = 1
       and last = n, where n is the list size.
   Postcondition: x[first], ..., x[last] is sorted.
  -------------------------------------------------------------------*/
```

```
{
  int mid;
  stack<int> s;

  s.push(first);
  s.push(last);

  while(!s.empty() )
  {
    last = s.top();
    s.pop();
    first = s.top();
    s.pop();
    if ( first < last)
    {
      mid = split(x, first, last);
      s.push(first);
      s.push(mid-1);
      s.push(mid+1);
      s.push(last);
    }
  }
}
```

11.

```
template <typename ElementType>
int median(ElementType x[], int first, int last, int mid)
/*----------------------------------------------------------------
  Find the median of a list using a quicksort scheme.

   Precondition: < and == are defined for ElementType.
       Note: Client programs call median() with first = 1
       last = n, mid = (n + 1)/2, where n is the list size.
   Postcondition: Index of median element is returned.
  -----------------------------------------------------------------*/
{
  int pos = split(x, first, last);
  if (pos > mid)
    return median(x, first, pos - 1, mid);
  else if (pos < mid)
    return median(x, pos + 1, last, mid);
  else
    return pos;
}
```

12. This is a simple modification of #11.
Call `median()` with `median(array, 1, n, k)`.

Exercises 13.4

1.

F	13 57 39 85 70 22 64 48				
F1	13	39	70	64	
F2	57	85	22	48	
F	13 57 39 85 22 70 48 64				
F1	13 57	22 70			
F2	39 85	48 64			
F	13 39 57 85 22 48 64 70				
F1	13 39 57 85				
F2	22 48 64 70				
F	13 22 39 48 57 64 70 85				

2.

F	13 57 39 85 99 70 22 48 64				
F1	13	39	99	22	64
F2	57	85	70	48	
F	13 57 39 85 70 99 22 48 64				
F1	13 57	70 99	64		
F2	39 85	22 48			
F	13 39 57 85 22 48 70 99 64				
F1	13 39 57 85	64			
F2	22 48 70 99				
F	13 22 39 48 57 70 85 99 64				
F1	13 22 39 48 57 70 85 99				
F2	64				
F	13 22 39 48 57 64 70 85 99				

3.

F	13 22 57 99 39 64 57 48 70				
F1	13	57	39	57	70
F2	22	99	64	48	
F	13 22 57 99 39 64 48 57 70				
F1	13 22	39 64	70		
F2	57 99	48 57			
F	13 22 57 99 39 48 57 64 70				
F1	13 22 57 99	70			
F2	39 48 57 64				
F	13 22 39 48 57 57 64 99 70				
F1	13 22 39 48 57 57 64 99				
F2	70				
F	13 22 39 48 57 57 64 70 99				

4.

F	13 22 39 48 57 64 70 85				
F1	13	39	57	70	
F2	22	48	64	85	
F	13 22 39 48 57 64 70 85				
F1	13 22	57 64			
F2	39 48	70 85			
F	13 22 39 48 57 64 70 85				
F1	13 22 39 48				
F2	57 64 70 85				
F	13 22 39 48 57 64 70 85				

5.

F	85 70 64 57 48 39 22 13			
F1	85	64	48	22
F2	70	57	39	13
F	70 85 57 64 39 48 13 22			
F1	70 85	39 48		
F2	57 64	13 22		
F	57 64 70 85 13 22 39 48			
F1	57 64 70 85			
F2	13 22 39 48			
F	13 22 39 48 57 64 70 85			

6.

F	13 57 39 85 70 22 64 48		
F1	13 57	70	48
F2	39 85	22 64	
F1	13 57 70	48	
F2	39 85	22 64	
F	13 39 57 70 85 22 48 64		
F1	13 39 57 70 85		
F2	22 48 64		
F	13 22 39 48 57 64 70 85		

7.

F	13 57 39 85 99 70 22 48 64	
F1	13 57	70
F2	39 85 99	22 48 64
F1	13 57 70	
F2	39 85 99	22 48 64
F	13 39 57 70 85 99 22 48 64	
F1	13 39 57 70 85 99	
F2	22 48 64	
F	13 22 39 48 57 64 70 85 99	

8.

F	13 22 57 99 39 64 57 48 70	
F1	13 22 57 99	57
F2	39 64	48 70
F	13 22 39 57 64 99 48 57 70	
F1	13 22 39 57 64 99	
F2	48 57 70	
F	13 22 39 48 57 57 64 70 99	

9.

F	13 22 39 48 57 64 70 85
F1	13 22 39 48 57 64 70 85
F2	
F	13 22 39 48 57 64 70 85

10. This is the same as Exercise 5.

11.

```
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

int merge(string & outName, string inName1, string inName2)
/*-------------------------------------------------------------------------
  Merge sorted subfiles in two different files.

  Precondition:  Files named inName1 and inName2 contain sorted subfiles.
  Postcondition: File named outName contains the result of merging
      these sorted subfiles.
 -------------------------------------------------------------------------*/
{

  ofstream f(outName.data());
  ifstream  f1(inName1.data()),
            f2(inName2.data());;

  int in1;
  int in2;
  bool inSub1,
       inSub2;

  int  numSubfiles = 0;
  int  oldone1, oldone2;

  f1 >> in1;
  f2 >> in2;

  while ( !f1.eof()  && !f2.eof() )
  {
    inSub1 = inSub2 = true;

    while (inSub1 && inSub2)
    {
      if (in1 < in2)
      {
        f << in1 << endl;
        oldone1 = in1;
        f1 >> in1;
        inSub1 = !f1.eof() && (oldone1 <= in1);
      }
      else
      {
        f << in2 << endl;
        oldone2 = in2;
        f2 >> in2;
        inSub2 = !f2.eof() && (oldone2 <= in2);
      }
    }
    if (inSub2)
      while (inSub2)
      {
        f << in2 << endl;
        oldone2 = in2;
        f2 >> in2;
        inSub2 = !f2.eof() && (oldone2 <= in2);
      }
    else
      while (inSub1)
      {
        f << in1 << endl;
        oldone1 = in1;
        f1 >> in1;
        inSub1 = !f1.eof() && (oldone1 <= in1);
      }
    numSubfiles++;
  }

  while ( !f1.eof() )
  {
    f << in1 << endl;
    oldone1 = in1;
    f1 >> in1;
    if ( !f1.eof() )
      if (oldone1 > in1)
        numSubfiles++;
  }

  while ( !f2.eof() )
  {
    f << in2 << endl;
    oldone2 = in2;
    f2 >> in2;
    if ( !f2.eof() )
      if ( oldone2 > in2)
        numSubfiles++;
  }
  numSubfiles++;
  return numSubfiles;
}

void split(ifstream & f, string outName1, string outName2)
/*-------------------------------------------------------------------------
  Split file f by writing sorted subfiles alternately to the files
  named outName1 and outName2.

  Precondition:  f is open for input.
  Postcondition: Files named outName1 and outName2 contain the result of
      splitting f.
 -------------------------------------------------------------------------*/
{
  ofstream f1(outName1.data()),
           f2(outName2.data());
  bool inSub;
  int oldone,
      value;

  f >> value;
  while ( !f.eof() )
  {
    inSub = true;
    while (inSub)
    {
      f1 << value << endl;
      oldone = value;
      f >> value;
      inSub = ( !f.eof() ) && (oldone <= value);
    }

    if ( !f.eof() )
    {
      inSub = true;
      while (inSub)
      {
        f2 << value << endl;
        oldone = value;
        f >> value;
        inSub = ( !f.eof() ) && (oldone <= value);
      }
    }
  }
}

void mergesort(string filename)
/*-------------------------------------------------------------------------
  Mergesort.

  Precondition:  None.
  Postcondition: File named filename has been sorted into ascending order.
 -------------------------------------------------------------------------*/
{
  int subfiles = 2;  // to prime the while loop
  while (subfiles > 1)
  {
    ifstream infile(filename.data());

    string outfilename1 = "hold1",
           outfilename2 = "hold2";;

    split(infile, outfilename1, outfilename2);

    subfiles = merge(filename, outfilename1, outfilename2);
  }
}
```

12. See #11; the process is essentially the same. Now, the *array limits* control the iteration rather than the *end-of-file*.

13. See #11, the process is essentially the same. Now the *end of the linked list* controls the iteration rather than the *end-of-file*.

14. Change `mergesort()` as follows:

```
void mergesort(string filename)
/*-----------------------------------------------------------------------
  Mergesort.

  Precondition:  None.
  Postcondition: File named filename has been sorted into ascending order.
 -----------------------------------------------------------------------*/
{
  bool firstTime = true;
  int subfiles = 2;  // to prime the while loop
  while (subfiles > 1)
  {
    ifstream infile(filename.data());

    string outfilename1 = "hold1",
           outfilename2 = "hold2";;

    if (firstTime)
    {
      firstSplit(infile, outfilename1, outfilename2);
      firstTime = false;
    }
    else
      split(infile, outfilename1, outfilename2);

    subfiles = merge(filename, outfilename1, outfilename2);
  }
}
```

where `firstSplit()` is:

```
void firstSplit(ifstream & f, string outName1, string outName2)
/*-----------------------------------------------------------------------
  Split file f by copying fixed-size subfiles into an array, quicksorting
  these subfiles, and then writing these sorted arrays alternately to the
  files named outName1 and outName2.

  Precondition:  f is open for input.
  Postcondition: Files named outName1 and outName2 contain the result of
      splitting f.
 -----------------------------------------------------------------------*/
{
  const int SUBFILE_SIZE = 8;
  ElementType internalStore[SUBFILE_SIZE + 1];


  ofstream f1(outName1.data()),
           f2(outName2.data());

  ElementType value;
  int filenum = 1;
  do
  {
    int count;
    for (count = 0; count < SUBFILE_SIZE; count++)
    {
      f >> value;
      if ( f.eof() ) break;
      internalStore[count + 1] = value;
    }

    quicksort(internalStore, 1, count);
    switch (filenum)
    {
      case 1:  for (int i = 1; i <= count; i++)
                  f1 << internalStore[i] << endl;
               break;
      case 2:  for (int i = 1; i <= count; i++)
                  f2 << internalStore[i] << endl;
    }
    filenum = 3 - filenum;
  }
  while ( !f.eof() );
}
```

15. A three-way merge is similar to a two-way merge, except now the comparison is between three files and the smallest element of the three starts the copying of that subfile to the output file.

Exercises 13.5

1. 029, 778, 11, 352, 233, 710, 783, 812, 165, 106

Distribute:

710	011	812 352	783 233		165	106		778	029
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Collect together from left to right, bottom to top:
710, 011, 352, 812, 233, 783, 165, 106, 778, 029

Distribute:

106	812 011 710	029	233		352	165	778	783	
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Collect together from left to right, bottom to top:
106, 710, 011, 812, 029, 233, 352, 165, 778, 783

Distribute:

029 011	165 106	233	352				783 778 710	812	
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Collect together from left to right, bottom to top:
011, 029, 106, 165, 233, 352. 710, 778, 783, 812

2. 038, 399, 892, 389, 683, 400, 937, 406, 316, 005
Distribute:

400		892	683		005	316 406	937	038	389 399
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Collect together from left to right, bottom to top:
400, 892, 683, 005, 406, 316, 937, 038, 399, 389

Distribute:

406 005 400	316		038 937					389 683	399 892
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Collect together from left to right, bottom to top:
400, 005, 406, 316, 937, 038, 683, 389, 892, 399

Distribute:

038 005			399 389 316	406 499		683		892	937
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Collect together from left to right, bottom to top:
005, 038, 316, 389, 399, 400, 406, 683, 892, 937

3. 353, 6, 295, 44, 989, 442, 11, 544, 209, 46
Distribute:

	011	442	353	544 044	295	046 006			209 989
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Collect together from left to right, bottom to top:
011, 442, 353, 044, 544, 295, 006, 046, 989, 209

Distribute:

209 006	011			046 544 044 442	353			989	295
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Collect together from left to right, bottom to top:
006, 209, 011, 442, 044, 544, 046, 353, 989, 295

Distribute:

046 044 011 006		295 209	353	442	544				989
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Collect together from left to right, bottom to top:
006, 011, 044, 046, 209, 295, 353, 442, 544, 989

4. 8745, 7438, 15, 12, 8501, 3642, 8219, 6152, 369, 6166, 8583, 7508, 8717, 8114, 630
Distribute:

0630	8501	6152 3642 0012	3583	8114	0015 8745	6166	8717	7508 7438	0369 8219
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Collect together from left to right, bottom to top:
0630, 8501, 0012, 3642, 6152, 8583, 8114, 8745, 0015, 6166, 8717, 7438, 7508, 8219, 0369

Distribute:

7508 8501	8219 8717 0015 8114 0012		7438 0630	8745 3642	6152	0369 6166		8583	
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Collect together from left to right, bottom to top:
8501, 7508, 0012, 8114, 0015, 8717, 8219, 0630, 7438, 3642, 8745, 6152, 6166, 0369, 8583

Distribute:

	6166				8583				
0015	6152				7508	3642	8745		
0012	8114	8219	0369	7438	8501	0630	8717		
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Collect together from left to right, bottom to top:
0012, 0015, 8114, 6152, 6166, 8219, 0369, 7438, 8501, 7508, 8583, 0630, 3642, 8717, 8745

Distribute:

								8745	
								8717	
0630								8583	
0369								8501	
0015						6166	7508	8219	
0012			3642			6152	7438	8114	
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Collect together from left to right, bottom to top:
0012, 0015, 0369, 0630, 3642, 6152, 6166, 7438, 7508, 8114, 8219, 8501, 8583, 8717, 8745

5. 9001, 78, 8639, 252, 9685, 3754, 4971, 888, 6225, 9686, 6967, 6884, 2, 4370, 131
Distribute:

	0131								
	4971	0002		6884	6225			0888	
4730	9001	0252		3754	9685	9686	3937	0078	8639
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Collect together from left to right, bottom to top:
4370, 9001, 4971, 0131, 0252, 0002, 3754, 6884, 9685, 6225, 9686, 6967, 0078, 0888, 8639

Distribute:

							0078	0888	
0002			8639		3754		4971	9686	
9001		6225	0131		0252	6967	4370	9685	
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Collect together from left to right, bottom to top:
9001, 0002, 6225, 0131, 8639, 0252, 3754, 6967, 4370, 4971, 0078, 6884, 9685, 9686, 0888

Distribute:

0002 9001	0131	0252 6225	4370			9686 9685 8639	3754	0888 6884	4971 6967
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Collect together from left to right, bottom to top:
9001, 0002, 0078, 0131, 6225, 0252, 4370, 8639, 9685, 9686, 3754, 6884, 0888, 6967, 4971

Distribute:

0888 0252 0131 0078 0002			3754	4971 4370		6967 6884 6225		8639	9686 9685 9001
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Collect together from left to right, bottom to top:
0002, 0078, 0131, 0252, 0888, 3754, 4370, 4971, 6225, 6884, 6967, 8639, 9001, 9685, 9686

6. for#, if##, do##, else, case, int#, main (# denotes a blank)
Distribute:

	case
	else
			main		for#
					if##
					do#
					int#

...	**e**	**...**	**n**	**...**	**blank**

Collect together from left to right, bottom to top:
else, case, if##, main, do##, for#, int#

Distribute:

	main		for#	case
				else
					int#		do##
							if##

...	**i**	**...**	**r**	**s**	**t**	**...**	**blank**

Collect together from left to right, bottom to top:
main, for#, else, case, int#, if##, do##

Distribute:

case
main
		if##		else		int#	do##
							for#	

a	**...**	**i**	**...**	**l**	**...**	**n**	**o**	**...**

Collect together from left to right, bottom to top:
main, case, if##, else, int#, for#, do##

Distribute:

	case	do##	else	for#		int# if##		main
...	**c**	**d**	**e**	**f**	**...**	**i**	**...**	**m**

Collect together from left to right, bottom to top:
case, do##, else, for#, if##, int#, main

7. while, if###, for##, break, float, bool#   (# denotes a blank)
Distribute:

	while		break		float		bool#
							if###
							for##
							do###

...	**e**	**...**	**k**	**...**	**t**	**...**	**blank**

Collect together from left to right, bottom to top:
while, break, float, do###, for##, if###, bool#

Distribute:

float
break
		while		bool#		if###
						for##
						do###

a	**...**	**l**	**...**	**o**	**...**	**blank**

Collect together from left to right, bottom to top:
break, float, while, bool#, do###, for##, if###

Distribute:

break		while		bool#
				float
						for##		if###
								do###

a	**...**	**i**	**...**	**o**	**...**	**r**	**...**	**blank**

Collect together from left to right, bottom to top:
break, while, float, bool#, for##, do###, if###

Distribute:

	if###		while		float		do###
							for##
							bool#
									break	

...	**f**	**...**	**h**	**...**	**l**	**...**	**o**	**...**	**r**	**...**

Collect together from left to right, bottom to top:
if###, while, float, bool#, for##, do###, break

Distribute:

	break bool#		do###		for## float		if###		while	
...	**b**	**...**	**d**	**...**	**f**	**...**	**i**	**...**	**w**	**...**

Collect together from left to right, bottom to top:
bool#, break, do###, float, for##, if###, while

8. Selection sort is not stable. Consider the following list of records consisting of an integer and a character:

   [2, A], [2, B], [1, C]

   Sorting so integers are in ascending order gives:

   [1, C], [2, B], [2, A]

   The relative order of the 2’s has changed.

9. Bubble sort is stable.

10. Insertion sort is stable.

11. Heapsort is not stable. See the example from #8.

12. Quicksort is not stable. See the example from #8.

13. Binary Mergesort is not stable. Consider this list : [2, A], [2, B], [1, C], [3, C]

    Again, sorting so integers are in ascending order gives: [1, C], [2, B], [2, A], [3, C].

    The relative order of the 2's has changed.

14. Natural Mergesort is not stable. See example from #13.

15. Radix sort is stable.

16-17.

```
#include <iostream>
#include <list>
#include <iomanip>
using namespace std;

typedef int ElementType;
void radixSort(list<ElementType> & x, int numDigits, int base)
{

  list<ElementType> * bucket = new list<ElementType>[base];
  int basePower = 1;
  ElementType value;

  for (int pass = 1; pass <= numDigits; pass++)
  {
    while (!x.empty())
    {
      value = x.front();
      x.pop_front();
      int digit = value % (base * basePower) / basePower;
      bucket[digit].push_back(value);
    }

     for (int i = 0; i < base; i++)
       while ( !bucket[i].empty() )
       {
         value = bucket[i].front();
         x.push_back(value);
         bucket[i].pop_front();
       }

    basePower *= base;

// UNCOMMENT THE FOLLOWING LINES TO TRACE RADIX SORT
/*
#include <iomanip>
cout << pass << ": ";
for (list<ElementType>::iterator it = x.begin(); it != x.end(); it++)
   cout << setfill('0') << setw(numDigits) << *it << ", ";
cout << endl;
*/

  }
}
```

18. The function in the preceding exercise can be easily modified for this.

```
#include <iostream>
#include <list>
#include <string>
#include <cctype>
using namespace std;
```

```
typedef string ElementType;
void radixSort(list<ElementType> & x, int maxLength)
{
  list<ElementType> * bucket = new list<ElementType>[27];
  ElementType value;

  for (int pass = maxLength - 1; pass >= 0;  pass--)
  {
    while (!x.empty())
    {
      value = x.front();
      x.pop_front();

      int charPos;
      if (value[pass] != ' ')
        charPos = int(value[pass]) - int('a');
      else
        charPos = 26;
      bucket[charPos].push_back(value);
    }

     for (int i = 0; i <= 26; i++)
       while ( !bucket[i].empty() )
       {
         value = bucket[i].front();
         x.push_back(value);
         bucket[i].pop_front();
       }

     // UNCOMMENT THE FOLLOWING LINES TO TRACE RADIX SORT
     /*
     cout << pass << ": ";
     for (list<ElementType>::iterator it = x.begin(); it != x.end(); it++)
       cout << *it << ", ";
     cout << endl;
     */
  }
}
```