HEAP SORT

In Chapter 18, we described the quick sort algorithm for contiguous lists, that is, array-based lists. We remarked that, on average, the quick sort is of the order $O(n\log_2 n)$. However, in the worst case, the quick sort is of the order $O(n^2)$. This section describes another algorithm, the heap sort, for array-based lists. This algorithm is of the order $O(n\log_2 n)$ even in the worst case, therefore, overcoming the worst case of the quick sort.

Definition: A **heap** is a list in which each element contains a key, such that the key in the element at position k in the list is at least as large as the key in the element at position $2k + 1$ (if it exists) and $2k + 2$ (if it exists).

Recall that in C++ the array index starts at `0`. Therefore, the element at position k is, in fact, the $k + 1$th element of the list.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
85	70	80	50	40	75	30	20	10	35	15	62	58

FIGURE H-1 A list that is a heap

Consider the list in Figure H-1.

It can be verified that the list in Figure H-1 is a heap.

Given a heap, we can construct a complete binary tree as follows: The root node of the tree is the first element of the list. The left child of the root is the second element of the list, and the right child of the root node is the third element of the list. Thus, in general, for the node k, which is the $k - 1$th element of the list, its left child is the $2k$th element of the list (if it exists), which is at position $2k - 1$ in the list, and the right child is the $(2k + 1)$th element of the list (if it exists), which is at position $2k$ in the list.

The diagram in Figure H-2 represents the complete binary tree corresponding to the list in Figure H-1.

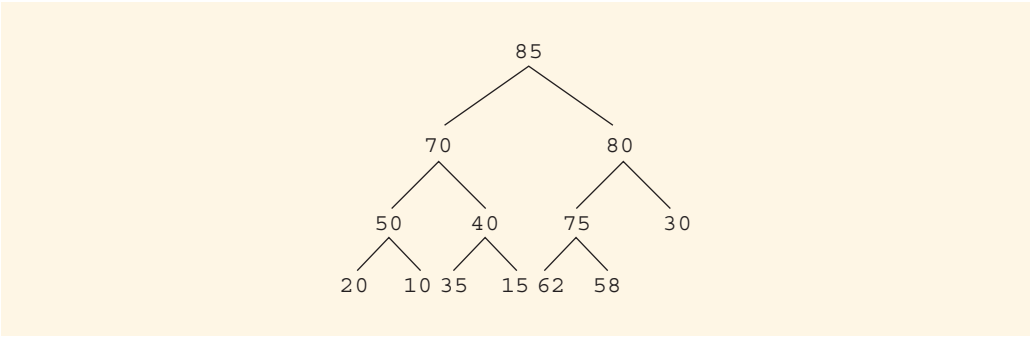

FIGURE H-2 Complete binary tree corresponding to the list in Figure H-1

Figure H-2 shows that the list in Figure H-1 is a heap. In fact, to demonstrate the heap sort algorithm, we will always draw the complete binary tree corresponding to a list.

We now describe the heap sort algorithm.

The first step in the heap sort algorithm is to convert the list into a heap, called `buildHeap`. After we convert the array into a heap, the sorting phase begins.

Build Heap

This section describes the build heap algorithm.

The general algorithm is as follows: suppose `length` denotes the length of the list. Let `index = length / 2 - 1`. Then, `list[index]` is the last element in the list that is not a leaf; that is, this element has at least one child. Thus, elements `list[index+1]...list[length-1]` are leaves.

First, we convert the subtree with the root node `list[index]` into a heap. Note that this subtree has, at most, three nodes. We then convert the subtree with the root node `list[index - 1]` into a heap, and so on.

To convert a subtree into a heap, we perform the following steps.

Suppose that `list[a]` is the root node of the subtree, `list[b]` is the left child of `list[a]`, and `list[c]`, if it exists, is the right child of `list[a]`.

Compare `list[b]` with `list[c]` to determine the larger child. If `list[c]` does not exist, then `list[b]` is the larger child. Suppose that `largerIndex` indicates the larger child. (Notice that `largerIndex` is either `b` or `c`.)

1. Compare `list[a]` with `list[largerIndex]`. If `list[a] < list[largerIndex]`, then swap `list[a]` with `list[largerIndex]`; otherwise, the subtree with the root node `list[a]` is already a heap.
2. Suppose that `list[a] < list[largerIndex]`, and we swap the elements `list[a]` and `list[largerIndex]`. After making this swap, the subtree with the root node `list[largerIndex]` might not be a heap. If this is the case, then

we repeat Steps 1 and 2 at the subtree with the root node `list[largerIndex]`, and continue this process until either the heaps in the subtrees are restored or we arrive at an empty subtree. This step is implemented using a loop, which is described when we write the algorithm.

Consider the list in Figure H-3. Let us call this `list`.

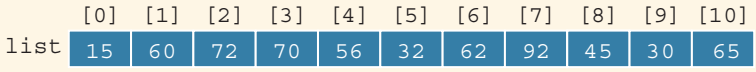

FIGURE H-3 Array `list`

Figure H-4 shows the complete binary tree corresponding to the list in Figure H-3.

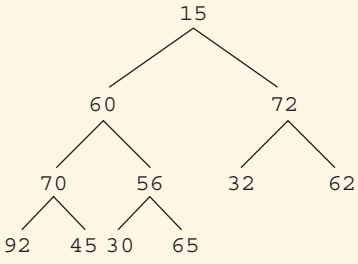

FIGURE H-4 Complete binary tree corresponding to the list in Figure H-3

To facilitate this discussion, when we say node `56`, we mean the node with info `56`.

This list has `11` elements, and so the length of the list is `11`. To convert the array into a heap, we start at the list element `n / 2 - 1 = 11 / 2 - 1 = 5 - 1 = 4`, which is the fifth element of the list.

Now `list[4] = 56`. The children of `list[4]` are `list[4 * 2 + 1]` and `list[4 * 2 + 2]`, that is, `list[9]` and `list[10]`. In the previous list, both `list[9]` and `list[10]` exist. To convert the tree with root node `list[4]`, we perform the following three steps:

1. Find the larger of `list[9]` and `list[10]`, that is, the largest child of `list[4]`. In this case, `list[10]` is larger than `list[9]`.
2. Compare the larger child with the parent node. If the larger child is larger than the parent, swap the larger child with the parent. Because `list[4] < list[10]`, we swap `list[4]` with `list[10]`.
3. Because `list[10]` does not have a subtree, Step 3 does not execute.

Figure H-5 shows the resulting binary tree.

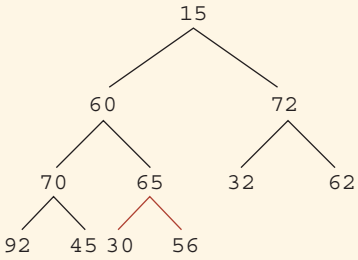

FIGURE H-5 Binary tree after swapping `list[4]` with `list[10]`

Next, we consider the subtree with root node **`list[3]`**, that is, `70`, and repeat the three steps given earlier to obtain the complete binary tree as given in Figure H-6. (Notice that, again, Step 3 does not execute.)

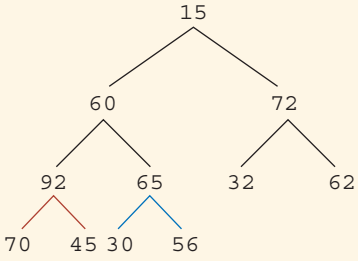

FIGURE H-6 Binary tree after repeating Steps 1, 2, and 3 at the root node `list[3]`

Now we consider the subtree with the root node **`list[2]`**, that is, `72`, and apply the three steps given earlier. Figure H-7 shows the resulting binary tree. (Note that in this case, because the parent is larger than both children, this subtree is already a heap.)

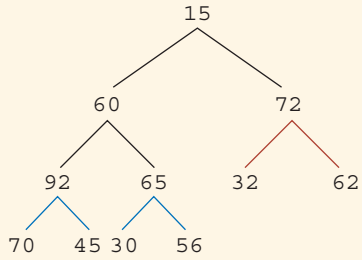

FIGURE H-7 Binary tree after repeating Steps 1 and 2 at the root node `list[2]`

Next, we consider the subtree with the root node `list[1]`, that is, `60`. First, we apply Steps 1 and 2. Because `list[1] = 60 < list[3] = 92` (the larger child), we swap `list[1]` with `list[3]` to obtain the tree as given in Figure H-8.

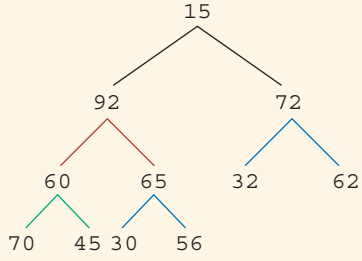

FIGURE H-8 Binary tree after swapping `list[1]` with `list[3]`

However, after swapping `list[1]` with `list[3]`, the subtree with the root node `list[3]`, that is, `60`, is no longer a heap. Thus, we must restore the heap in this subtree. To do this, we apply Step 3 and find the larger child of `60` and swap it with `60`. We then obtain the binary tree as given in Figure H-9.

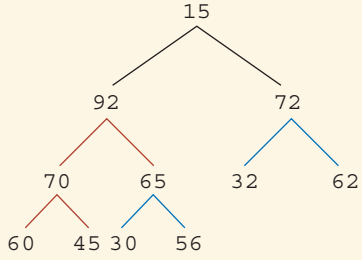

FIGURE H-9 Binary tree after restoring the heap at `list[3]`

Once again, the subtree with the root node `list[1]`, that is, `92`, is a heap (see Figure H-9).

Finally, we consider the tree with the root node `list[0]`, that is, `15`. We repeat the previous three steps to obtain the binary tree as given in Figure H-10.

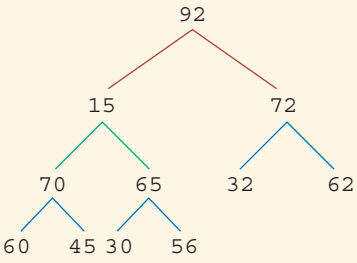

FIGURE H-10 Binary tree after applying Steps 1 and 2 at `list[0]`

We see that the subtree with the root node `list[1]`, that is, `15`, is no longer a heap. So, we must apply Step 3 to restore the heap in this subtree. (This requires us to repeat Steps 1 and 2 at the subtree with root node `list[1]`.) We swap `list[1]` with the larger child, `list[3]` (that is, `70`). We then get the binary tree as shown in Figure H-11.

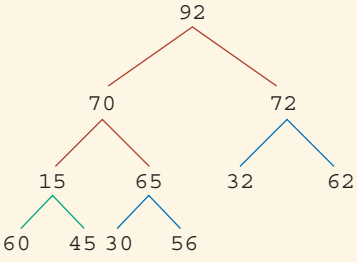

FIGURE H-11 Binary tree after applying Steps 1 and 2 at `list[1]`

The subtree with the root node `list[3] = 15` is not a heap, and so we must restore the heap in this subtree. To do so, we apply Steps 1 and 2 at the subtree with the root node `list[3]`. We swap `list[3]` with the larger child, `list[7]` (that is, `60`). Figure H-12 shows the resulting binary tree.

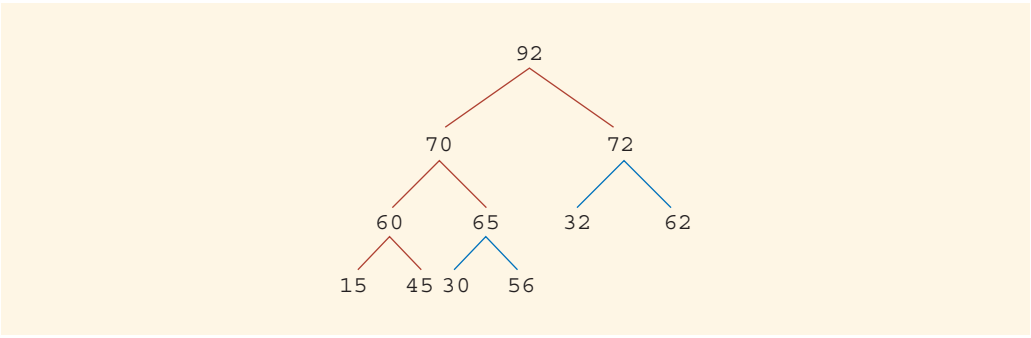

FIGURE H-12 Binary tree after restoring the heap at `list[3]`

The resulting binary tree (in Figure H-12) is a heap, and so the list corresponding to this complete binary tree is a heap.

Thus, in general, starting at the lowest level from right to left, we look at a subtree and convert the subtree into a heap as follows: If the root node of the subtree is smaller than the larger child, we swap the root node with the larger child; after swapping the root node with the larger child, we must restore the heap in the subtree whose root node was swapped.

Suppose `low` contains the index of the root node of the subtree, and `high` contains the index of the last item in the list. The heap is to be restored in the subtree rooted at `list[low]`. The preceding discussion translates into the following C++ algorithm:

```
int largeIndex = 2 * low + 1;    //the index of the left child

while (largeIndex <= high)
{
   if (largeIndex < high)
      if (list[largeIndex] < list[largeIndex + 1])
         largeIndex = largeIndex + 1; //the index of the larger
                                      //child
   if (list[low] > list[largeIndex])  //the subtree is already
                                      //a heap
      break;
   else
   {
      swap(list, list[low], list[largeIndex]);   //Line **
      low = largeIndex;  //go to the subtree to further
                         //restore the heap
      largeIndex = 2 * low + 1;
   } //end else
}//end while
```

The `swap` statement at the line marked `Line**` swaps the parent with the larger child. Because a `swap` statement makes three item assignments to swap the contents of two variables, each time through the loop, three item assignments are made. The `while` loop moves the parent node to a place in the tree, so that the resulting subtree with the root node `list[low]` is a heap. We can easily reduce the number of assignments each time through the loop from three to one by

first storing the root node in a temporary location, say, `temp`. Then, each time through the loop, the larger child is compared with `temp`. If the larger child is larger than `temp`, we move the larger child to the root node of the subtree under consideration.

Next, we describe the function `heapify`, which restores the heap in a subtree by making one item assignment each time through the loop. The index of the root node of the list and the index of the last element of the list are passed as parameters to this function:

```
template <class elemType>
void heapify(elemType list[], int low, int high)
{
    int largeIndex;

    elemType temp = list[low];  //copy the root node of
                                //the subtree

    largeIndex = 2 * low + 1;  //index of the left child

    while (largeIndex <= high)
    {
        if (largeIndex < high)
            if (list[largeIndex] < list[largeIndex + 1])
                largeIndex = largeIndex + 1; //index of the
                                             //largest child

        if (temp > list[largeIndex]) //subtree
                                     //is already a heap
            break;
        else
        {
            list[low] = list[largeIndex]; //move the larger
                                          //child to the root
            low = largeIndex;    //go to the subtree to
                                 //restore the heap
            largeIndex = 2 * low + 1;
        }
    }//end while

    list[low] = temp; //insert temp into the tree,
                      //that is, list
}//end heapify
```

Next, we use the function `heapify` to implement the `buildHeap` function to convert the list into a heap:

```
template <class elemType>
void buildHeap(elemType list[], int length)
{
    for (int index = length / 2 - 1; index >= 0; index--)
        heapify(list, index, length - 1);
}
```

We now describe the heap sort algorithm.

Suppose the list is a heap. Consider the complete binary tree representing the list as given in Figure H-13.

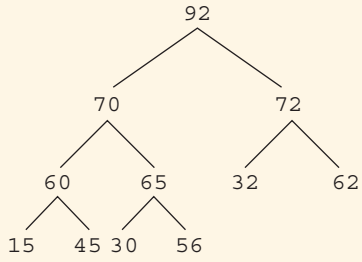

FIGURE H-13 A heap

Because this is a heap, the root node is the largest element of the tree (or the list). This means that it must be moved to the end of the list. We swap the root node of the tree (which is the first element of the list) with the last node in the tree (which is the last element of the list). We then obtain the binary tree as shown in Figure H-14.

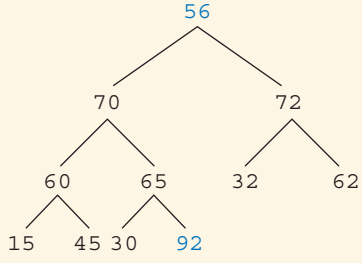

FIGURE H-14 Binary tree after moving the root node to the end

Because the largest element is now in its proper place, we consider the remaining elements of the list, that is, elements `list[0]...list[9]`. The complete binary tree representing this list is no longer a heap, and so we must restore the heap in this portion of the complete binary tree. We use the function `heapify` to restore the heap. A call to this function is:

```
heapify(list, 0, 9);
```

We thus obtain the binary tree as shown in Figure H-15.

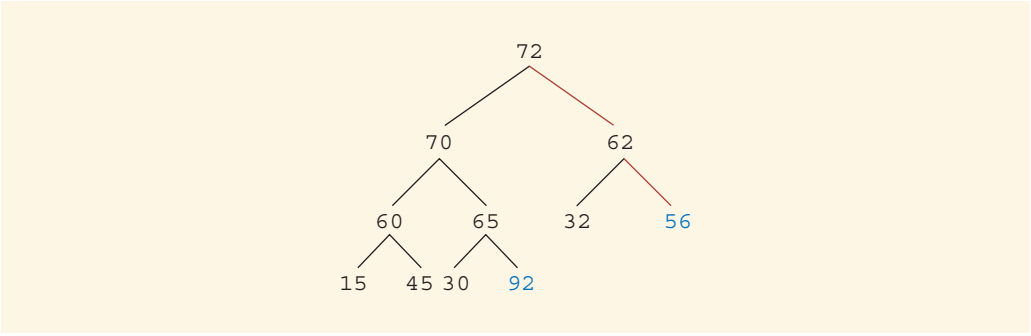

FIGURE H-15 Binary tree after the statement `heapify(list, 0, 9);` executes

We repeat this process for the complete binary tree corresponding to the list elements `list[0]...list[9]`. We swap `list[0]` with `list[9]` and then restore the heap in the complete binary tree corresponding to the list elements `list[0]...list[8]`. We continue this process.

The following C++ function describes this algorithm:

```
template <class elemType>
void heapSort(elemType list[], int length)
{
    buildHeap(list, length);
    for (int lastOutOfOrder = length - 1; lastOutOfOrder >= 0;
                                          lastOutOfOrder--)
    {
        elemType temp = list[lastOutOfOrder];
        list[lastOutOfOrder] = list[0];
        list[0] = temp;
        heapify(list, 0, lastOutOfOrder - 1);
    }//end for
}//end heapSort
```

We leave it as an exercise for you to write a program to test the heap sort algorithm. See Programming Exercise 1 at the end of this chapter.

Analysis: Heap Sort

Suppose that L is a list of n elements, where $n > 0$. In the worst case, the number of key comparisons in the heap sort algorithm to sort L (the number of comparisons in `heapSort` and the number of comparisons in `buildHeap`) is $2n\log_2 n + O(n) = O(n\log_2 n)$. Also, in the worst case, the number of item assignments in the heap sort algorithm to sort L is $n\log_2 n + O(n) = O(n\log_2 n)$. On average, the number of comparisons made by the heap sort algorithm to sort L is $O(n\log_2 n)$.

In the average case of the quick sort algorithm, the number of key comparisons is $1.39n\log_2 n + O(n)$ and the number of swaps is $0.69n\log_2 n + O(n)$. Because each swap is

three assignments, the number of item assignments in the average case of the quick sort algorithm is at least $1.39n\log_2 n + O(n)$. It now follows that for the key comparisons, the average case of the quick sort algorithm is somewhat better than the worst case of the heap sort algorithm. On the other hand, for the item assignments, the average case of the quick sort algorithm is somewhat poorer than the worst case of the heap sort algorithm. However, the worst case of the quick sort algorithm is $O(n^2)$. Empirical studies have shown that the heap sort algorithm usually takes twice as long as the quick sort algorithm but avoids the slight possibility of poor performance.

QUICK REVIEW

1. A heap is a list in which each element contains a key, such that the key in the element at position k in the list is at least as large as the key in the element at position $2k + 1$ (if it exists) and $2k + 2$ (if it exists).
2. The first step in the heap sort algorithm is to convert the list into a heap, called `buildHeap`. After we convert the array into a heap, the sorting phase begins.
3. Suppose that L is a list of n elements, where $n > 0$. In the worst case, the number of key comparisons in the heap sort algorithm to sort L is $2n\log_2 n + O(n) = O(n\log_2 n)$. Also, in the worst case, the number of item assignments in the heap sort algorithm to sort L is $n\log_2 n + O(n) = O(n\log_2 n)$.

EXERCISES

1. Use the function `buildHeap`, as given in this chapter, to convert the following array into a heap. Show the final form of the array.

   ```
   47, 78, 81, 52, 50, 82, 58, 42, 65, 80, 92, 53, 63, 87, 95, 59, 34,
   37, 7, 20
   ```

2. Suppose that the following list was created by the function `buildHeap` during the heap creation phase of the heap sort algorithm.

   ```
   100, 85, 94, 47, 72, 82, 76, 30, 20, 60, 65, 50, 45, 17, 35, 14,
   28, 5
   ```

   Show the resulting array after two passes of the heap sort algorithm. (Use the `heapify` procedure as given in this chapter.) Exactly how many key comparisons are executed during the first pass?

PROGRAMMING EXERCISE

1. Write a program to test the heap sort algorithm as given in this chapter.