```
            Heapify(A, 1, m)                        // fix things up
        }
    }
```

An example of HeapSort is shown in Figure 7.4 on page 148 of CLR. We make $n-1$ calls to Heapify, each of which takes $O(\log n)$ time. So the total running time is $O((n-1)\log n) = O(n \log n)$.

Lecture 14: HeapSort Analysis and Partitioning

(Thursday, Mar 12, 1998)

Read: Chapt 7 and 8 in CLR. The algorithm we present for partitioning is different from the texts.

HeapSort Analysis: Last time we presented HeapSort. Recall that the algorithm operated by first building a heap in a bottom-up manner, and then repeatedly extracting the maximum element from the heap and moving it to the end of the array. One clever aspect of the data structure is that it resides inside the array to be sorted.

We argued that the basic heap operation of Heapify runs in $O(\log n)$ time, because the heap has $O(\log n)$ levels, and the element being sifted moves down one level of the tree after a constant amount of work.

Based on this we can see that (1) that it takes $O(n \log n)$ time to build a heap, because we need to apply Heapify roughly $n/2$ times (to each of the internal nodes), and (2) that it takes $O(n \log n)$ time to extract each of the maximum elements, since we need to extract roughly n elements and each extraction involves a constant amount of work and one Heapify. Therefore the total running time of HeapSort is $O(n \log n)$.

Is this tight? That is, is the running time $\Theta(n \log n)$? The answer is yes. In fact, later we will see that it is not possible to sort faster than $\Omega(n \log n)$ time, assuming that you use comparisons, which HeapSort does. However, it turns out that the first part of the analysis is not tight. In particular, the BuildHeap procedure that we presented actually runs in $\Theta(n)$ time. Although in the wider context of the HeapSort algorithm this is not significant (because the running time is dominated by the $\Theta(n \log n)$ extraction phase).

Nonetheless there are situations where you might not need to sort all of the elements. For example, it is common to extract some unknown number of the smallest elements until some criterion (depending on the particular application) is met. For this reason it is nice to be able to build the heap quickly since you may not need to extract all the elements.

BuildHeap Analysis: Let us consider the running time of BuildHeap more carefully. As usual, it will make our lives simple by making some assumptions about n. In this case the most convenient assumption is that n is of the form $n = 2^{h+1} - 1$, where h is the height of the tree. The reason is that a left-complete tree with this number of nodes is a complete tree, that is, its bottommost level is full. This assumption will save us from worrying about floors and ceilings.

With this assumption, level 0 of the tree has 1 node, level 1 has 2 nodes, and up to level h, which has 2^h nodes. All the leaves reside on level h.

Recall that when Heapify is called, the running time depends on how far an element might sift down before the process terminates. In the worst case the element might sift down all the way to the leaf level. Let us count the work done level by level.

At the bottommost level there are 2^h nodes, but we do not call Heapify on any of these so the work is 0. At the next to bottommost level there are 2^{h-1} nodes, and each might sift down 1 level. At the 3rd level from the bottom there are 2^{h-2} nodes, and each might sift down 2 levels. In general, at level j

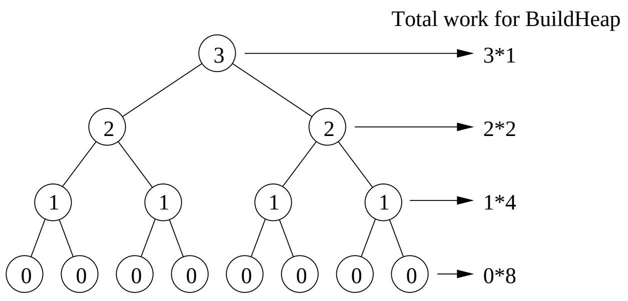

Figure 13: Analysis of BuildHeap.

from the bottom there are 2^{h-j} nodes, and each might sift down j levels. So, if we count from bottom to top, level-by-level, we see that the total time is proportional to

$$T(n) = \sum_{j=0}^{h} j2^{h-j} = \sum_{j=0}^{h} j\frac{2^h}{2^j}.$$

If we factor out the 2^h term, we have

$$T(n) = 2^h \sum_{j=0}^{h} \frac{j}{2^j}.$$

This is a sum that we have never seen before. We could try to approximate it by an integral, which would involve integration by parts, but it turns out that there is a very cute solution to this particular sum. We'll digress for a moment to work it out. First, write down the infinite general geometric series, for any constant $x < 1$.

$$\sum_{j=0}^{\infty} x^j = \frac{1}{1-x}.$$

Then take the derivative of both sides with respect to x, and multiply by x giving:

$$\sum_{j=0}^{\infty} jx^{j-1} = \frac{1}{(1-x)^2} \qquad \sum_{j=0}^{\infty} jx^j = \frac{x}{(1-x)^2},$$

and if we plug $x = 1/2$, then voila! we have the desired formula:

$$\sum_{j=0}^{\infty} \frac{j}{2^j} = \frac{1/2}{(1-(1/2))^2} = \frac{1/2}{1/4} = 2.$$

In our case we have a bounded sum, but since the infinite series is bounded, we can use it instead as an easy approximation.

Using this we have

$$T(n) = 2^h \sum_{j=0}^{h} \frac{j}{2^j} \le 2^h \sum_{j=0}^{\infty} \frac{j}{2^j} \le 2^h \cdot 2 = 2^{h+1}.$$

Now recall that $n = 2^{h+1} - 1$, so we have $T(n) \le n + 1 \in O(n)$. Clearly the algorithm takes at least $\Omega(n)$ time (since it must access every element of the array at least once) so the total running time for BuildHeap is $\Theta(n)$.

It is worthwhile pausing here a moment. This is the second time we have seen a relatively complex structured algorithm, with doubly nested loops, come out with a running time of $\Theta(n)$. (The other example was the median algorithm, based on the sieve technique. Actually if you think deeply about this, there is a sense in which a parallel version of BuildHeap can be viewed as operating like a sieve, but maybe this is getting too philosophical.) Perhaps a more intuitive way to describe what is happening here is to observe an important fact about binary trees. This is that the vast majority of nodes are at the lowest level of the tree. For example, in a complete binary tree of height h there is a total of $n \approx 2^{h+1}$ nodes in total, and the number of nodes in the bottom 3 levels alone is

$$2^h + 2^{h-1} + 2^{h-2} \;=\; \frac{n}{2} + \frac{n}{4} + \frac{n}{8} \;=\; \frac{7n}{8} = 0.875n.$$

That is, almost 90% of the nodes of a complete binary tree reside in the 3 lowest levels. Thus the lesson to be learned is that when designing algorithms that operate on trees, it is important to be most efficient on the bottommost levels of the tree (as BuildHeap is) since that is where most of the weight of the tree resides.

Partitioning: Our next sorting algorithm is QuickSort. QuickSort is interesting in a number of respects. First off, (as we will present it) it is a *randomized algorithm*, which means that it makes use of a random number generator. We will show that in the worst case its running time is $O(n^2)$, its expected case running time is $O(n \log n)$. Moreover, this expected case running time occurs with *high probability*, in that the probability that the algorithm takes significantly more than $O(n \log n)$ time is rapidly decreasing function of n. In addition, QuickSort has a better locality-of-reference behavior than either MergeSort or HeapSort, and thus it tends to run fastest of all three algorithms. This is how it got its name. QuickSort (and its variants) are considered the methods of choice for most standard library sorting algorithms.

Next time we will discuss QuickSort. Today we will discuss one aspect of QuickSort, namely the partitioning algorithm. This is the same partitioning algorithm which we discussed when we talked about the selection (median) problem. We are given an array $A[p..r]$, and a pivot element x chosen from the array. Recall that the partitioning algorithm is suppose to partition A into three subarrays: $A[p..q-1]$ whose elements are all less than or equal to x, $A[q] = x$, and $A[q+1..r]$ whose elements are greater than or equal to x. We will assume that x is the first element of the subarray, that is, $x = A[p]$. If a different rule is used for selecting x, this is easily handled by swapping this element with $A[p]$ before calling this procedure.

We will present a different algorithm from the one given in the text (in Section 8.1). This algorithm is a little easier to verify the correctness, and a little easier to analyze. (But I suspect that the one in the text is probably a bit for efficient for actual implementation.)

This algorithm works by maintaining the following *invariant condition*. The subarray is broken into four segments. The boundaries between these items are indicated by the indices p, q, s, and r.

(1) $A[p] = x$ is the pivot value,

(2) $A[p+1..q]$ contains items that are less than x,

(3) $A[q+1..s-1]$ contains items that are greater than or equal to x, and

(4) $A[s..r]$ contains items whose values are currently unknown.

This is illustrated below.

The algorithm begins by setting $q = p$ and $s = p + 1$. With each step through the algorithm we test the value of $A[s]$ against x. If $A[s] \geq x$, then we can simply increment s. Otherwise we increment q, swap $A[s]$ with $A[q]$, and then increment s. Notice that in either case, the invariant is still maintained. In the first case this is obvious. In the second case, $A[q]$ now holds a value that is less than x, and $A[s-1]$ now holds a value that is greater than or equal to x. The algorithm ends when $s = r$, meaning that

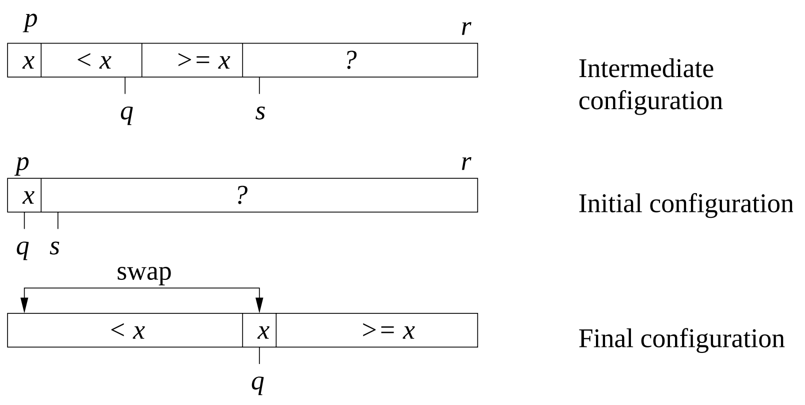

Figure 14: Partitioning intermediate structure.

all of the elements have been processed. To finish things off we swap $A[p]$ (the pivot) with $A[q]$, and return the value of q. Here is the complete code:

Partition

```
Partition(int p, int r, array A) {              // 3-way partition of A[p..r]
    x = A[p]                                    // pivot item in A[p]
    q = p
    for s = p+1 to r do {
        if (A[s] < x) {
            q = q+1
            swap A[q] with A[s]
        }
    }
    swap A[p] with A[q]                        // put the pivot into final position
    return q                                   // return location of pivot
}
```

An example is shown below.

Lecture 15: QuickSort

(Tuesday, Mar 17, 1998)

Revised: March 18. Fixed a bug in the analysis.

Read: Chapt 8 in CLR. My presentation and analysis are somewhat different than the text's.

QuickSort and Randomized Algorithms: Early in the semester we discussed the fact that we usually study the worst-case running times of algorithms, but sometimes average-case is a more meaningful measure. Today we will study QuickSort. It is a worst-case $\Theta(n^2)$ algorithm, whose expected-case running time is $\Theta(n \log n)$.

We will present QuickSort as a *randomized* algorithm, that is, an algorithm which makes random choices. There are two common types of randomized algorithms:

Monte Carlo algorithms: These algorithms may produce the wrong result, but the probability of this occurring can be made arbitrarily small by the user. Usually the lower you make this probability, the longer the algorithm takes to run.