A heap is represented as an *left-complete* binary tree. This means that all the levels of the tree are full except the bottommost level, which is filled from left to right. An example is shown below. The keys of a heap are stored in something called *heap order*. This means that for each node u, other than the root, $key(Parent(u)) \geq key(u)$. This implies that as you follow any path from a leaf to the root the keys appear in (nonstrict) increasing order. Notice that this implies that the root is necessarily the largest element.

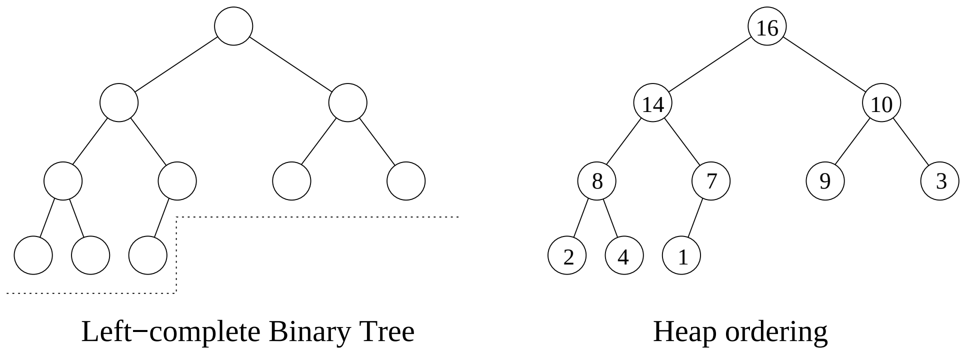


Figure 11: Heap.

Next time we will show how the priority queue operations are implemented for a heap.

Lecture 13: HeapSort

(Tuesday, Mar 10, 1998)

Read: Chapt 7 in CLR.

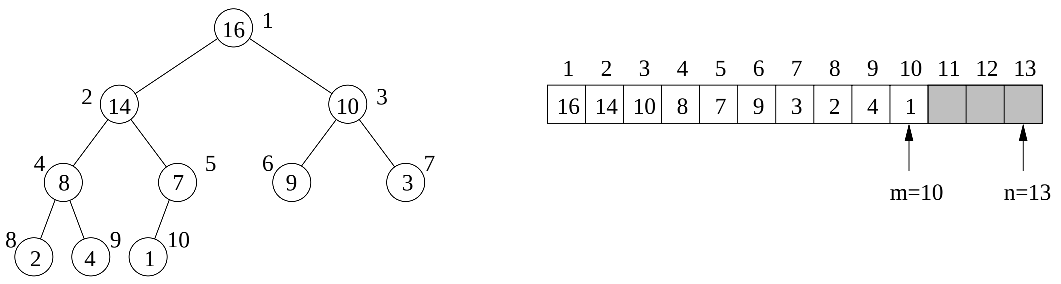
Heaps: Recall that a heap is a data structure that supports the main priority queue operations (insert and extract max) in $\Theta(\log n)$ time each. It consists of a left-complete binary tree (meaning that all levels of the tree except possibly the bottommost) are full, and the bottommost level is filled from left to right. As a consequence, it follows that the depth of the tree is $\Theta(\log n)$ where n is the number of elements stored in the tree. The keys of the heap are stored in the tree in what is called *heap order*. This means that for each (nonroot) node its parent's key is at least as large as its key. From this it follows that the largest key in the heap appears at the root.

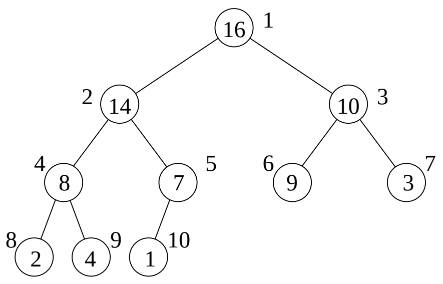
Array Storage: Last time we mentioned that one of the clever aspects of heaps is that they can be stored in arrays, without the need for using pointers (as would normally be needed for storing binary trees). The reason for this is the left-complete nature of the tree.

This is done by storing the heap in an array $A[1..n]$. Generally we will not be using all of the array, since only a portion of the keys may be part of the current heap. For this reason, we maintain a variable $m \leq n$ which keeps track of the current number of elements that are actually stored actively in the heap. Thus the heap will consist of the elements stored in elements $A[1..m]$.

We store the heap in the array by simply unraveling it level by level. Because the binary tree is left-complete, we know exactly how many elements each level will supply. The root level supplies 1 node, the next level 2, then 4, then 8, and so on. Only the bottommost level may supply fewer than the appropriate power of 2, but then we can use the value of m to determine where the last element is. This is illustrated below.

We should emphasize that this *only works* because the tree is left-complete. This cannot be used for general trees.





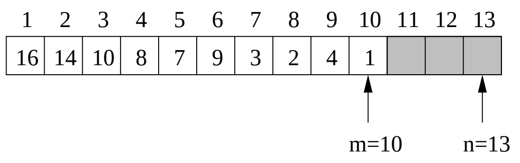


Figure 12: Storing a heap in an array.

We claim that to access elements of the heap involves simple arithmetic operations on the array indices. In particular it is easy to see the following.

$Left(i)$: return $2i$.

$Right(i)$: return $2i + 1$.

$Parent(i)$: return $\lfloor i/2 \rfloor$.

$IsLeaf(i)$: return $Left(i) > m$. (That is, if i's left child is not in the tree.)

$IsRoot(i)$: return $i == 1$.

For example, the heap ordering property can be stated as "for all i, $1 \leq i \leq n$, if (not $IsRoot(i)$) then $A[Parent(i)] \geq A[i]$".

So is a heap a binary tree or an array? The answer is that from a conceptual standpoint, it is a binary tree. However, it is implemented (typically) as an array for space efficiency.

Maintaining the Heap Property: There is one principal operation for maintaining the heap property. It is called `Heapify`. (In other books it is sometimes called *sifting down*.) The idea is that we are given an element of the heap which we suspect may not be in valid heap order, but we assume that all of other the elements in the subtree rooted at this element are in heap order. In particular this root element may be too small. To fix this we "sift" it down the tree by swapping it with one of its children. Which child? We should take the larger of the two children to satisfy the heap ordering property. This continues recursively until the element is either larger than both its children or until its falls all the way to the leaf level. Here is the pseudocode. It is given the heap in the array A, and the index i of the suspected element, and m the current active size of the heap. The element $A[max]$ is set to the maximum of $A[i]$ and it two children. If $max \neq i$ then we swap $A[i]$ and $A[max]$ and then recurse on $A[max]$.

Heapify

```
Heapify(array A, int i, int m) {                // sift down A[i] in A[1..m]
    l = Left(i)                                 // left child
    r = Right(i)                                // right child
    max = i
    if (l <= m and A[l] > A[max]) max = l       // left child exists and larger
    if (r <= m and A[r] > A[max]) max = r       // right child exists and larger
    if (max != i) {                             // if either child larger
        swap A[i] with A[max]                   // swap with larger child
        Heapify(A, max, m)                      // and recurse
    }
}
```

See Figure 7.2 on page 143 of CLR for an example of how Heapify works (in the case where $m = 10$). We show the execution on a tree, rather than on the array representation, since this is the most natural way to conceptualize the heap. You might try simulating this same algorithm on the array, to see how it works at a finer details.

Note that the recursive implementation of Heapify is not the most efficient. We have done so because many algorithms on trees are most naturally implemented using recursion, so it is nice to practice this here. It is possible to write the procedure iteratively. This is left as an exercise.

The HeapSort algorithm will consist of two major parts. First building a heap, and then extracting the maximum elements from the heap, one by one. We will see how to use Heapify to help us do both of these.

How long does Hepify take to run? Observe that we perform a constant amount of work at each level of the tree until we make a call to Heapify at the next lower level of the tree. Thus we do $O(1)$ work for each level of the tree which we visit. Since there are $\Theta(\log n)$ levels altogether in the tree, the total time for Heapify is $O(\log n)$. (It is not $\Theta(\log n)$ since, for example, if we call Heapify on a leaf, then it will terminate in $\Theta(1)$ time.)

Building a Heap: We can use Heapify to build a heap as follows. First we start with a heap in which the elements are not in heap order. They are just in the same order that they were given to us in the array A. We build the heap by starting at the leaf level and then invoke Heapify on each node. (Note: We cannot start at the top of the tree. Why not? Because the precondition which Heapify assumes is that the entire tree rooted at node i is already in heap order, except for i.) Actually, we can be a bit more efficient. Since we know that each leaf is already in heap order, we may as well skip the leaves and start with the first nonleaf node. This will be in position $\lfloor n/2 \rfloor$. (Can you see why?)

Here is the code. Since we will work with the entire array, the parameter m for Heapify, which indicates the current heap size will be equal to n, the size of array A, in all the calls.

BuildHeap

```
BuildHeap(int n, array A[1..n]) {              // build heap from A[1..n]
    for i = n/2 downto 1 {
        Heapify(A, i, n)
    }
}
```

An example of BuildHeap is shown in Figure 7.3 on page 146 of CLR. Since each call to Heapify takes $O(\log n)$ time, and we make roughly $n/2$ calls to it, the total running time is $O((n/2)\log n) = O(n \log n)$. Next time we will show that this actually runs faster, and in fact it runs in $\Theta(n)$ time.

HeapSort: We can now give the HeapSort algorithm. The idea is that we need to repeatedly extract the maximum item from the heap. As we mentioned earlier, this element is at the root of the heap. But once we remove it we are left with a hole in the tree. To fix this we will replace it with the last leaf in the tree (the one at position $A[m]$). But now the heap order will very likely be destroyed. So we will just apply Heapify to the root to fix everything back up.

HeapSort

```
HeapSort(int n, array A[1..n]) {              // sort A[1..n]
    BuildHeap(n, A)                           // build the heap
    m = n                                     // initially heap contains all
    while (m >= 2) {
        swap A[1] with A[m]                   // extract the m-th largest
        m = m-1                               // unlink A[m] from heap
```

```
            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