HONR 278J: Lecture
Heaps and Sorting
Adapted from David Mount's notes

Sorting: The reasons for studying sorting algorithms in details are twofold. First, sorting is a very important algorithmic problem. Procedures for sorting are parts of many large software systems, either explicitly or implicitly. Thus the design of efficient sorting algorithms is important for the overall efficiency of these systems. The other reason is more pedagogical. There are many sorting algorithms, some slow and some fast. Some possess certain desirable properties, and others do not. Finally sorting is one of the few problems where there provable lower bounds on how fast you can sort. Thus, sorting forms an interesting case study in algorithm theory.

In the sorting problem we are given an array $A[1..n]$ of n numbers, and are asked to reorder these elements into increasing order. (More generally, A is of an array of records, and we choose one of these records as the *key value* on which the elements will be sorted. The key value need not be a number. It can be any object from a *totally ordered* domain. Totally ordered means that for any two elements of the domain, x, and y, either $x < y$, $x =$, or $x > y$.)

Slow Sorting Algorithms: There are a number of well-known slow sorting algorithms. These include the following:

Bubblesort: Scan the array. Whenever two consecutive items are found that are out of order, swap them. Repeat until all consecutive items are in order.

Insertion sort: Assume that $A[1..i-1]$ have already been sorted. Insert $A[i]$ into its proper position in this subarray, by shifting all larger elements to the right by one to make space for the new item.

Selection sort: Assume that $A[1..i-1]$ contain the $i-1$ smallest elements in sorted order. Find the smallest element in $A[i..n]$, and then swap it with $A[i]$.

These algorithms are all easy to implement, but they run in $\Theta(n^2)$ time in the worst case. However, MergeSort sorts an array of numbers in $\Theta(n \log n)$ time. We will study two others, HeapSort and QuickSort.

Priority Queues: The heapsort algorithm is based on a very nice data structure, called a *heap*. A heap can be used to represent a priority queue, which stores elements, each of which is associated with a numeric key value, called its *priority*.

A simple priority queue supports three basic operations:

Create: Create an empty queue.

Insert: Insert an element into a queue.

ExtractMax: Return the element with maximum key value from the queue. (Actually it is more common to extract the minimum. It is easy to modify the implementation (by reversing $<$ and $>$ to do this.)

Empty: Test whether the queue is empty.

Adjust Priority: Change the priority of an item in the queue.

It is common to support a number of additional operations as well, such as building a priority queue from an initial set of elements, returning the largest element without deleting it, and changing the priority of an element that is already in the queueu (either decreasing or increasing it).

Heaps: A heap is a data structure that supports the main priority queue operations (insert and delete max) in $\Theta(\log n)$ time. For now we will describe the heap in terms of a binary tree implementation, but we will see later that heaps can be stored in arrays. The heap described here is a max-heap with the maximum element at the root.

By a *binary tree* we mean a data structure which is either empty or else it consists of three things: a root node, a left subtree and a right subtree. The left subtree and right subtrees are each binary trees. They are called the *left child* and *right child* of the root node. If both the left and right children of a node are empty, then this node is called a *leaf node*. A nonleaf node is called an *internal* node.

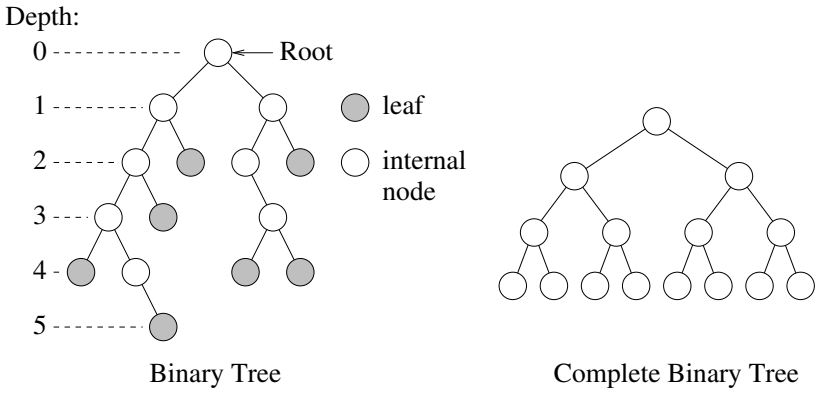

Figure 1: Binary trees.

The *depth* of a node in a binary tree is its distance from the root. The root is at depth 0, its children at depth 1, its grandchildren at depth 2, and so on. The *height* of a binary tree is its maximum depth. Binary tree is said to be *complete* if all internal nodes have two (nonempty) children, and all leaves have the same depth. An important fact about a complete binary trees is that a complete binary tree of height h has

$$n = 1 + 2 + \ldots + 2^h = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

nodes altogether. If we solve for h in terms of n, we see that the the height of a complete binary tree with n nodes is $h = (\lg(n+1)) - 1 \approx \lg n \in \Theta(\log n)$.

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.

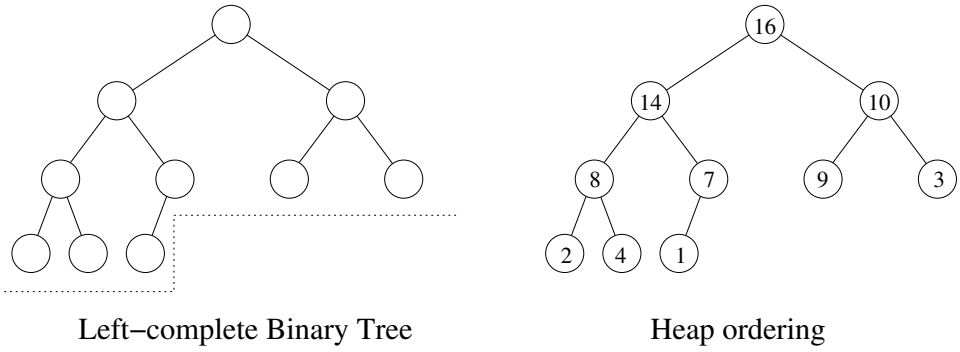

Figure 2: Heap.

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

Array Storage: One of the clever aspects of heaps is that they can be stored conveniently in an array (table). 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.

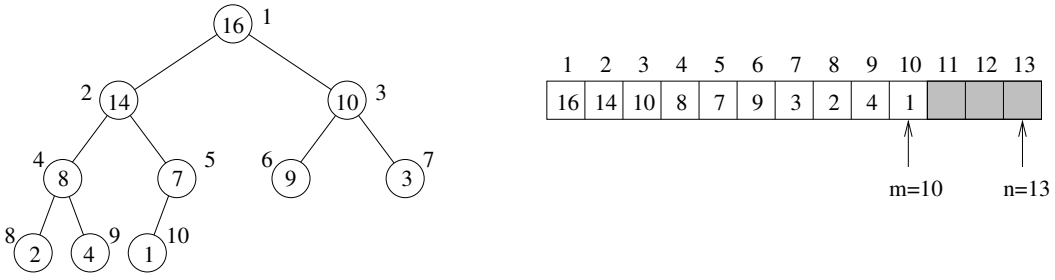

Figure 3: Storing a heap in an array.

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

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*$(i)] \geq A[i]$".

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 until the element is either larger than both its children or until its falls all the way to the leaf level.

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?)

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.

Here is a detailed description of the code:

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
    }
}
```

We make $n-1$ calls to Heapify, each of which takes $O(\log n)$ time. So the total running time is $O((n-1)\log n) = O(n \log n)$.

HeapSort Analysis: 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.