Divide-and-conquer: Understand how to design algorithms by divide-and-conquer. Understand the divide-and-conquer algorithm for MergeSort, and be able to work an example by hand. Also understand how the sieve technique works, and how it was used in the selection problem. (Chapt 10 on Medians; skip the randomized analysis. The material on the 2-d maxima and long integer multiplication is not discussed in CLR.)

Lecture 11: First Midterm Exam

(Tuesday, March 3, 1998)

First midterm exam today. No lecture.

Lecture 12: Heaps and HeapSort

(Thursday, Mar 5, 1998)

Read: Chapt 7 in CLR.

Sorting: For the next series of lectures we will focus on sorting algorithms. 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 $x > y$.

There are some domains that can be partially ordered, but not totally ordered. For example, sets can be partially ordered under the subset relation, $\subset$, but this is not a total order, it is not true that for any two sets either $x \subset y$, $x = y$ or $x \supset y$. There is an algorithm called *topological sorting* which can be applied to "sort" partially ordered sets. We may discuss this later.

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. We have already seen that 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 is a concrete implementation of an abstract data type called a *priority queue*. A priority queue 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.

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.

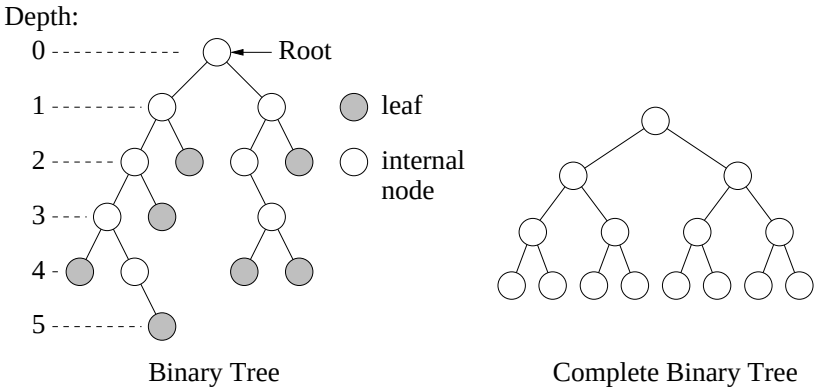

Figure 10: Binary trees.

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.

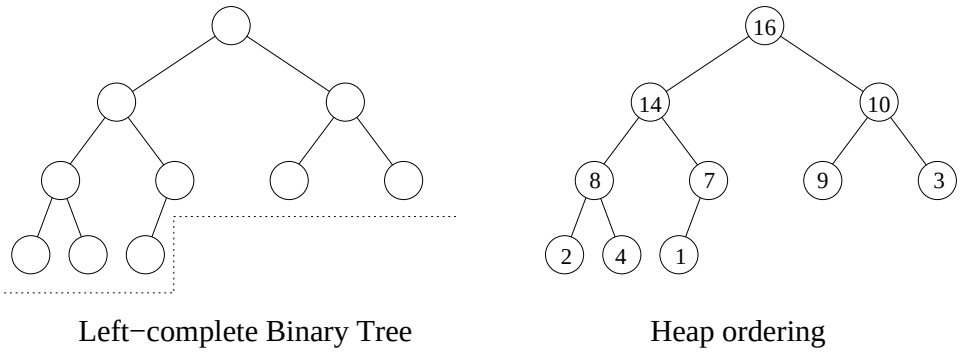

Figure 11: Heap.

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.