CS 218 : Design and Analysis of Algorithms

Lecture 1: Introduction

Lecturer: *Sundar Vishwanathan*
Computer Science & Engineering Indian Institute of Technology, Bombay

1 Review: Data Structures

A data structure is a way of storing data in a computer so that it can be used efficiently. If you are reading this you should have already done a course on Data Structures. You should be familiar with common data structures like arrays, lists, queues, stacks, heaps and balanced binary search trees. Anything else is bonus. Data Structures are usually viewed as a set with operations. Usually we will deal with sets that are ordered, like the integers. Data structures are usually evaluated by analysing the time taken in the worst case for each operation.

Consider an example which requires the following operations on an ordered set:

1. Insert: Insert a new element.
2. Find: Determine if an element is currently present

There are many ways of doing this. If the maximum number of elements to be inserted is known, an array could be used. A linked list is another possibility. In either case we can insert in $O(1)$ time, but find will take $O(n)$ time if n items have been inserted. Do you know more efficient ways of implementing the above? Depending on the data structure used, the cost of each operation may be different.

2 Heaps: A quick review

We begin with heaps. You should read a standard text for a more leisurely read. These notes give a summary.

The heap is a data structure on an ordered set with three operations.

1. Insert
2. Findmin: Which returns the minimum value among the elements currently in the set.
3. Deletemin: Which deletes and returns the minimum among the elements in the set.

This is implemented in the form of an almost complete binary tree. The data will be stored in nodes in a binary tree. The tree will have the following properties:

- The value at a node is smaller or equal to the value of any of its children.
- All levels except the last will be full. That is, level i which contains elements at distance i from the root will have 2^i elements, if i is not the last level.
- The last level may not be full. However, the elements will be left justified.

The first constraint is an order relationship to be maintained. The last two are structural constraints that help in the implementation. The last two conditions ensure that the length of the longest root-leaf path is $O(\log n)$. This will upper bound on time taken per operation.

2.1 Implemeting the operations on Heap

Prove that if there are n elements in the heap, the height of the binary tree is at most $\lceil \lg n \rceil$.

Implementing Insert

Add the element to the leftmost empty spot in the bottom level of the heap. This way the structure of the heap is preserved. However, elements may not satisfy the order relationships. We fix this as follows.

Compare the added element with its parent; if they are in the correct order, stop. If not, swap the new element with the value at the parent and repeat this procedure with the parent. That is, compare the value now at the parent with its parent etc. This way, the newly added element may bubble up the path to the root till it finds its correct place. Clearly, the time is at most the height of the tree which is $O(\log n)$.

Implementing Extract Min

Replace the value at the root by the rightmost element of the last level in the heap. Again, the structure of the heap is maintained. We need to move values to restore the order relationship.

Compare the value at the root with those of its children. If the value is smaller than those of its children we are done. Otherwise swap this with the value at the child containing the smaller value. Now repeat this process with this child. This way, the value at the root will bubble down the tree till all order relations are satisfied.

Exercise. How do you find the rightmost full slot or the leftmost empty slot in the last level? Hint: One way is to keep track of k, the number of elements in the heap. How do you get to the rightmost occupied slot starting from the root? It has something to do with writing k in binary. The other way is to keep moving up to the parent as long as you go left. As soon as you move right, come down to the other sibling and then keep going left till you hit the empty spot.

Binary Search Trees

Suppose we need to implement *Insert, Delete* and *Find*.

The data structure of choice here are (balanced) binary search trees. We will discuss balance in the next lecture. A *binary search tree* is a binary tree which stores values at nodes with the following order relation. The value at a node is larger than all the values at the nodes in its left subtree and is smaller than all the values present in its right subtree. By a left (right) subtree we mean the subtree rooted at the left (right) child.

Exercise. Write a procedure to output the values of a binary search tree in sorted order. It should visit each node at most twice.

It is instructive to see how to do these operations without maintaining balance before introducing *rotations*. To find a value in a binary search tree, make comparisons with the root and move in an appropriate direction down the tree till you find an element or reach an empty spot. If you reach an empty spot you know that the element is not present. For insert, you perform find and insert the value at the appropriate empty spot.

Here is an exercise for you to solve before you read ahead.

The *rank* of an element in the binary search tree is one plus the number of elements less than it.

Exercise. Given a node in a binary search tree write a procedure to find the next element in the sorted order. That is, find the element whose rank is one more (less). How much time does your procedure take?

How do you delete an element? See if you can solve (recall) this before reading ahead.

One way to do this is as follows: if the element e were a leaf then remove it. Otherwise, if e has a right child, we determine the element f whose rank is one more than the one we are deleting and replace it with that element. If e does not have a right child, we determine an element whose rank is one less than the one we are deleting and replace e with f. Now recurse on the node which contained the element f, till you reach a leaf. What is the time needed to

do this in terms of the height of the tree? What happens to the ordering relationship? What happens to the structure of the tree?

Exercise. Write down formal procedures that achieves the above.