Intermediate Data Structures

Saketh Are

March 4, 2012

1 Introduction

Simply put, a data structure is any particular way of storing and organizing data in a computer's memory. When writing programs that deal with significant amounts of data, the programmer's choice of data structures determines how efficiently various operations on the data can be performed. As we will see, these variations in operational efficiency mean that different types of data structures are suitable for different types of applications. Expanding your repertoire of data structures will allow you to address a wider range of problems in a computationally efficient manner.

We will be taking a look at a few data structures often useful for USACO Silver Division problems.

2 Heaps

Heaps are a type of data structure that can be thought of as a tree with additional restrictions. We will be dealing with two types of heaps: *min heaps* and *max heaps*.

2.1 Min Heaps

In a *min heap*, every child node is required to have a greater value than that of its parent. This implies that the minimum value in the data set must be stored at the root node, which doesn't have a parent.

2.2 Max Heaps

In a *max heap*, every child node is required to have a lesser value than that of its parent. This implies that the maximum value in the data set must be stored at the root node, which doesn't have a parent.

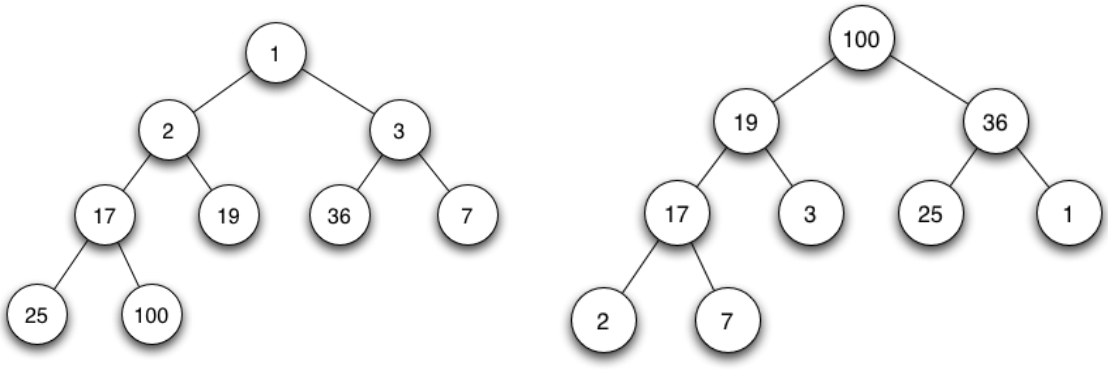

Figure 1: Examples of a *min heap* and a *max heap*, respectively

2.3 Implementation

While describing the structure of min and max heaps is fairly simple, efficient implementation can be a bit trickier. Here, we will describe how max heaps are implemented; the slight modifications then needed to implement min heaps are straightforward.

The underlying stucture is provided by a standard binary tree. We will want to implement two operations: insertion of a new element, and deletion of the root. Any operations must be made in a way that maintains the heap's structural properties.

2.3.1 Insertion

Insertion into a max heap is begun by attaching the new node to the left-most empty slot on the bottom row of the heap. We then perform what is known as an "up-heap" operation; as long as our new value is located in a node whose parent has a smaller value, we swap it with its parent.

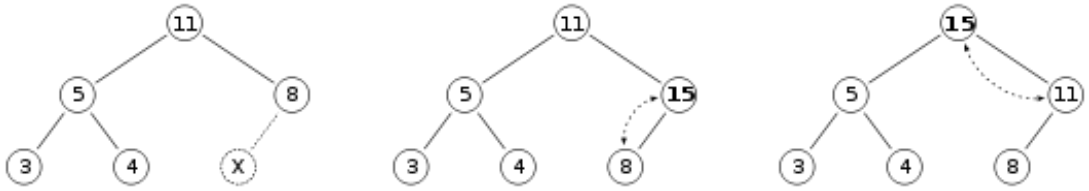

Figure 2: Inserting the value 15 into a max heap

Notice that as we go up the tree, we will only ever be swapping old values with our new, larger value. This means that the heap's ordering property property will never be distubred for any modified node's subtrees (transitivity of ordering). Since we perform a swap at most once per layer of the tree, this operation will run in $O(\log N)$, where N is the number of nodes.

2.3.2 Deletion

Deletion of a max heap's root is begun by swapping the root node to the right-most node on the bottom row of the heap and deleting it. We then perform what is known as a "down-heap" operation; as long as the new value is located in a node one of whose children has a larger value, we swap it with its larger child.

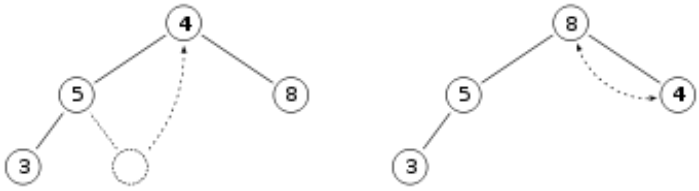

Figure 3: Deleting the root node from a max heap

Notice that as we go down the tree, we will only ever be swapping old values with our new, smaller value. This means that the heap property will be disturbed only within a modified node's subtrees, which we will process next. Since we perform a swap at most once per layer of the tree, this operation will run in $O(\log N)$, where N is the number of nodes.

2.4 Priority Queues

So why are heaps useful? Min and max heaps are used most commonly in implementation of *priority queues*. Priority queues can store data for you and serve it up in sorted order "on the fly." By taking advantage of the heap's ordering properties, we can find the min/max in constant time, remove it in logarithmic time,

and insert new values in logarithmic time. The implementation of Dijkstra's algorithm for sparse graphs is a good example of where priority queues can come in handy.

3 Binary Search Trees

Binary search trees are another example of tree-based data structures. Every node in a binary search tree has three basic properties:

1. The left subtree of a node contains only nodes with keys less than the node's key.
2. The right subtree of a node contains only nodes with keys greater than the node's key.
3. Both the left and right subtrees must also be binary search trees.

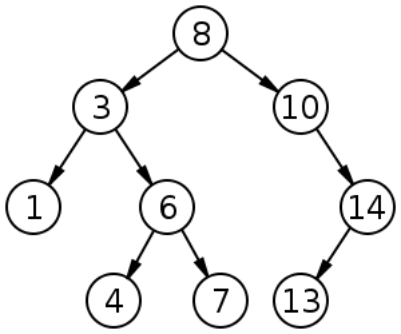

Figure 4: An example of a binary search tree

Binary search trees are good tools for performing operations on a data set that deal with its ordering, such as sorting or traversing it in order. They can also be used to implement a set.

To implement binary search trees, we will write an insertion function and a deletion function. Then, we will write functions to take advantage of the BST's properties to quickly search through the elements or traverse them in order.

3.1 Insertion

Inserting a value into a binary search tree is a recursive process. Our method will take two arguments: our current location in the tree, and the value we want to insert. Any time the function is called, if there is no value at our current location, we simply insert the new value.

Otherwise, we compare the new value to the value in the current location. If the new value is less than or equal to the stored value, we recursively call the insert function on the left subtree. If not, we recursively call it on the right subtree. In this way, a call to this method using the root node as the starting location will eventually put the new value in a location satisfying the properties of the binary search tree.

3.2 Deletion

When deleting a node, there are three cases to consider. If it does not have any children, we can simply delete it. If it has one child, we replace its value with that of its child, and then delete the child node. Finally, if our node has two children, we find the minimum value in its right subtree (this can be found by going to the right child and then going left as many times as we can), delete it from the tree, and place its value in our current node.