Topics: Trees

1 Trees

1.1 Introduction

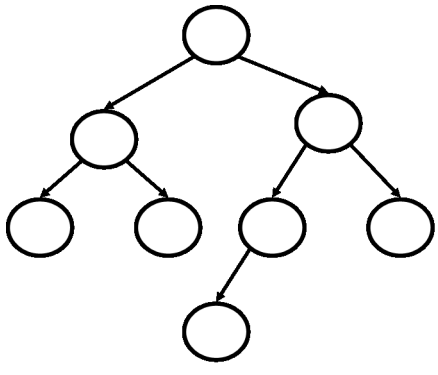
Recall our linked list definition:

```
class List {
  ListNode head;
  class ListNode {
    Object data;
    ListNode next;
  }
  // other fields and methods
}
```

Let's define some terminology. If a node points to a second node, we'll call the first node the *parent* of the second, and the second the *child* of the first. Then in our linked list, each node can only have one child. (This is also true for a doubly-linked list, but the child points back to its parent.) What if we extend our list definition to allow a node to have two children?

```
class NewList {
  NewListNode head;
  class NewListNode {
    Object data;
    NewListNode child1;
    NewListNode child2;
  }
  // other fields and methods
}
```

Let's draw an example of such a structure:



You may recall from CS61A that this structure is a tree. So a tree is just a linked list in which each node is no longer constrained to have a single child.

What if we extend our definition further to add another child. And another? For any $k > 0$, we can extend our definition such that each node can have as many as k children. If our tree allows k children, we

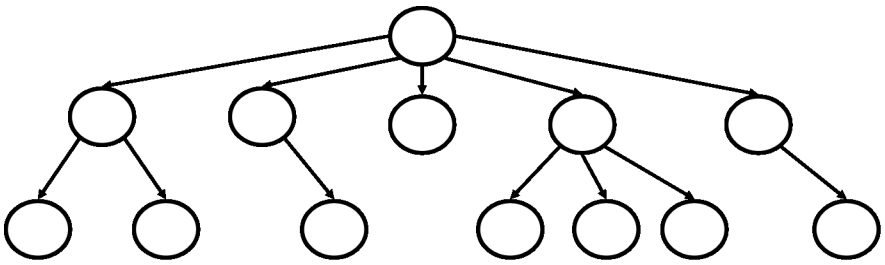


Figure 1: An example of a general tree.

call it a *k-ary tree*. Our definition above is a *binary tree* since each node may have up to two children. The fields in a binary tree have special names, and the tree is conventionally defined as:

```
class BinaryTree {
  BTNode root;
  class BTNode {
    Object data;
    BTNode left;
    BTNode right;
  }
  // other fields and methods
}
```

The naming of the two children make sense if you look at the above picture of the tree.

A linked list is just a unary tree. The most general case of a tree is one in which $k = \infty$, and is defined as follows:

```
class Tree {
  TreeNode root;
  class TreeNode {
    Object data;
    Vector children; // children nodes
  }
  // other fields and methods
}
```

Note that a constraint on linked lists carries over to trees. There cannot be any cycles in the tree. In addition, another constraint is introduced that was unnecessary for linked lists. Each node may only have a single parent. Without these constraints, tree traversals may fall into infinite loops.

1.2 Traversals

Our simple iterative traversal of a linked list will not work on trees, since multiple children of a node must be visited. Instead, we recursively traverse a tree. At each node, we must make one recursive call for each child.

Before we write our traversals, we need to figure out in what order we will traverse the tree. There are three ways to traverse the tree: *preorder*, *inorder*, and *postorder*. In a preorder traversal, the parent node is *visited* (i.e. whatever operation we are doing is done on the parent node) before any of its children. An inorder traversal only makes sense for a binary tree, and is one in which the left child is visited, then the parent, and finally the right child. In a postorder traversal, the parent is visited after its children. Since inorder traversals only apply to binary trees, we will use them as our medium for defining the traversals.

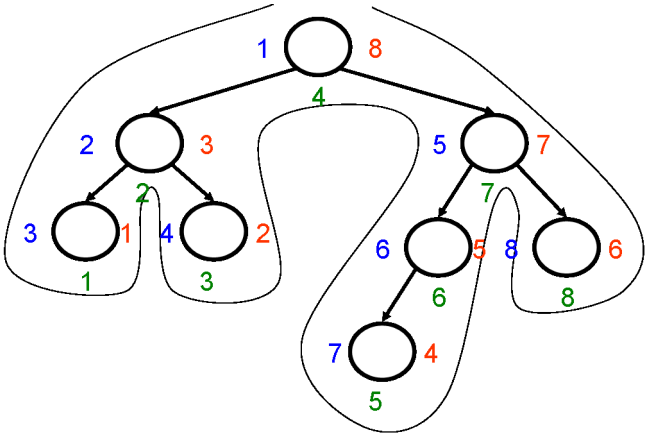


Figure 2: Preorder, inorder, and postorder traversals shown in blue, green, and red, respectively.

Pictorially we can execute the three traversals by drawing a line tightly around the tree, starting at the left of the root. For a preorder traversal, we visit a node when we pass to its left. In an inorder traversal, we visit a node when we pass underneath it. And for a postorder traversal, we visit a node when we pass to its right.

The actual definitions of the traversals are very simple. For simplicity, our traversals will just print out the values in the tree. A more advanced traversal could take in objects corresponding to operations that we could apply to each value.

```
class BinaryTree {
  BTNode root;
  void preorder() {
    System.out.print("The tree:");
    head.preorder();
    System.out.println();
  }
  void inorder() {
    System.out.print("The tree:");
    head.inorder();
    System.out.println();
  }
  void preorder() {
    System.out.print("The tree:");
    head.postorder();
    System.out.println();
  }
  class BTNode {
    Object data;
    BTNode left;
    BTNode right;
    void preorder() {
      System.out.print(" " + data);
      left != null ? left.preorder();
      right != null ? right.preorder();
    }
```

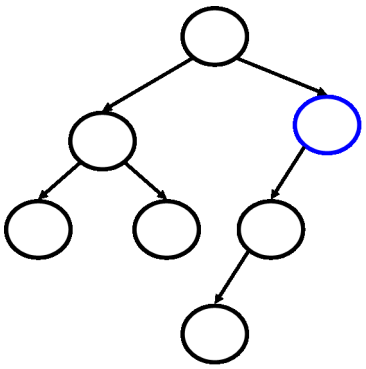


Figure 3: An unbalanced tree, with the unbalanced node in blue.

```
    void inorder() {
      left != null ? left.inorder();
      System.out.print(" " + data);
      right != null ? right.inorder();
    }
    void postorder() {
      left != null ? left.postorder();
      right != null ? right.postorder();
      System.out.print(" " + data);
    }
  }
  // other fields and methods
}
```

1.3 Specialized Trees

Now that we have defined a tree, what use are they to us? A general tree can be used to represent a heirarchy among items, such as the Java class tree we saw before. But besides that, a tree is not very useful unless we introduce more properties that must be satisfied. Specifically, we would like there to be some ordering among the values in the tree, and a fast and simple way to exploit that ordering. As such, the items in the tree must be of type `Comparable` so that an ordering actually exists among them. We will ignore the existence of equal items.

Some additional terminology is necessary in order to allow us to assess the usefulness of specialized trees. A binary tree is *balanced* if for each node in the tree, the heights of its subtree differ by at most one. A binary tree is *complete* if all but the last level in the tree are completely filled, and all nodes in the last level are all the way to the left. A complete tree is always balanced.

1.3.1 Binary Search Trees

If we are most interested in searching for individual items in a collection, or of obtaining a sorted list of all the items, then we use a *binary search tree* (BST). Such a tree introduces only a single constraint among the data items, that an inorder traversal on the tree visits the values in their natural order. This requires the tree to have two properties:

- The subtree to the left of a value may only contain values less than it.
- The subtree to the left of a value may only contain values greater than it.

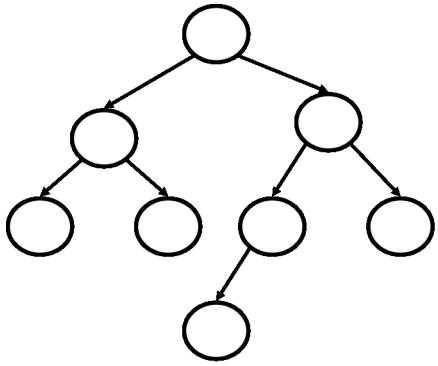


Figure 4: A balanced, but incomplete, tree.

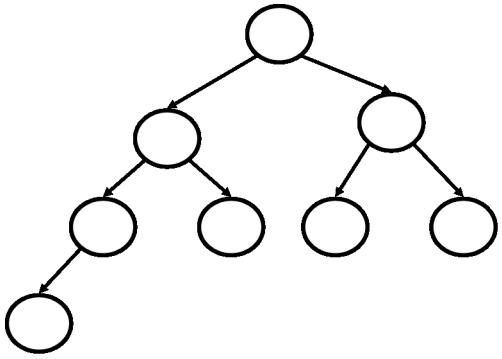


Figure 5: A complete tree.

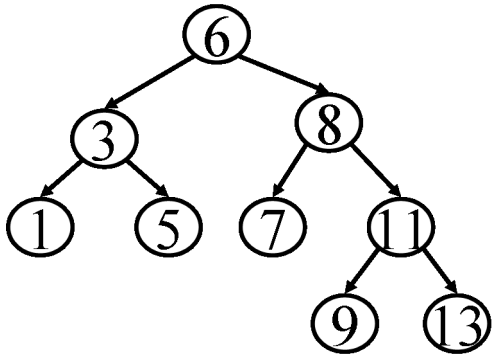


Figure 6: A binary search tree, with integer values.

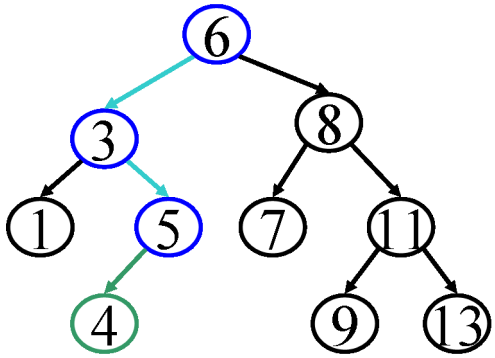


Figure 7: Insertion of the value 4 into the BST.

It is easy to see that an inorder traversal on a tree that obeys the two properties will visit values in order. Thus we can obtain an ordered list of elements by executing an inorder traversal on the tree.

Using the two BST properties, searching for an element is straightforward. Starting at the root, go left if the target is less than the value at the current node, right if it is greater. If it is equal, then of course you've found the element. To insert an item in a BST, use the search algorithm until you've hit a *leaf node* (a node with no children), and insert the item properly to its left or right.

Removal from a BST is a little trickier, since removing a node from a tree also removes its descendants. What if the value you wish to remove is not a leaf node? Then you must replace the node rather than just remove it. Inspection of a BST reveals that both the leftmost node in the right subtree or the rightmost node in the left subtree will always work as replacements, so either may be used.

Analysis of the above operations reveals that retrieving an ordered list of values is linear in the number of values in the tree, and the other three operations are linear in the height of the tree. But what is the height of the tree? A balanced tree has about $\lg n$ height were n is the number of elements, but a BST is not necessarily balanced. Consider a BST constructed by adding elements in order. Searching, insertion, and removal all take $O(n)$ time in this case! But in the average case, we would expect them to take $O(\lg n)$.

1.3.2 Heaps

Perhaps we can do better than the worst-case linear operations of a binary search tree if we require our tree to be complete. Unfortunately, nothing comes free, so most of those operations will become unavailable to us. However, if we are only interested in retrieving the minimum or maximum element in a set, we can use a *heap*. Depending on if we want the minimum or maximum, we use a *min-heap* or a *max-heap*. Here we

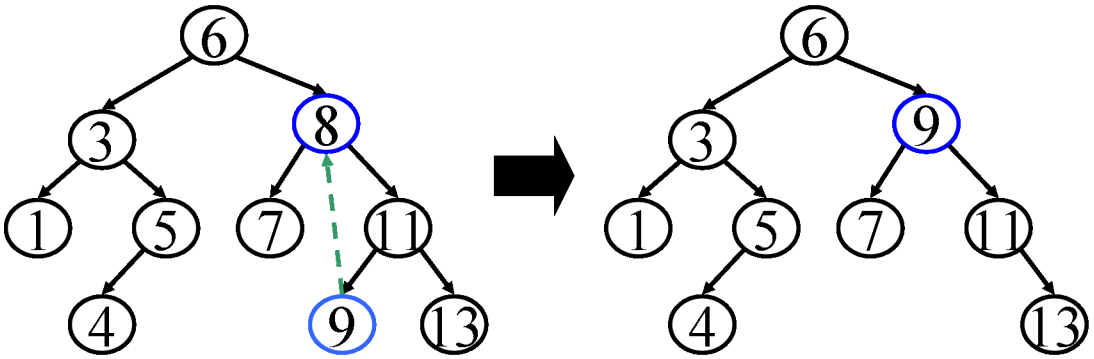


Figure 8: Removal of the value 8 from the BST.

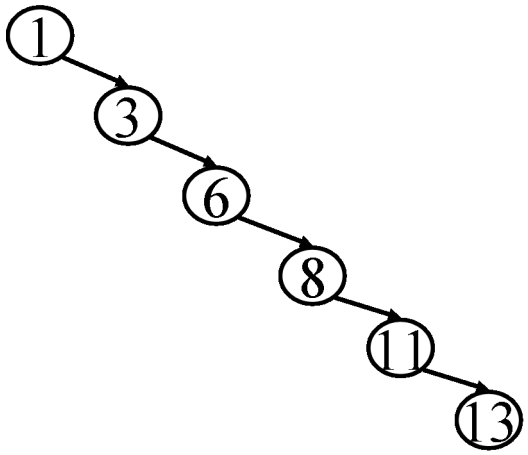


Figure 9: A binary search tree constructed in order.

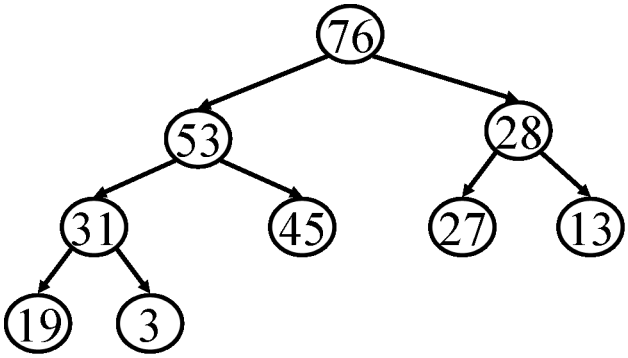


Figure 10: A max-heap.

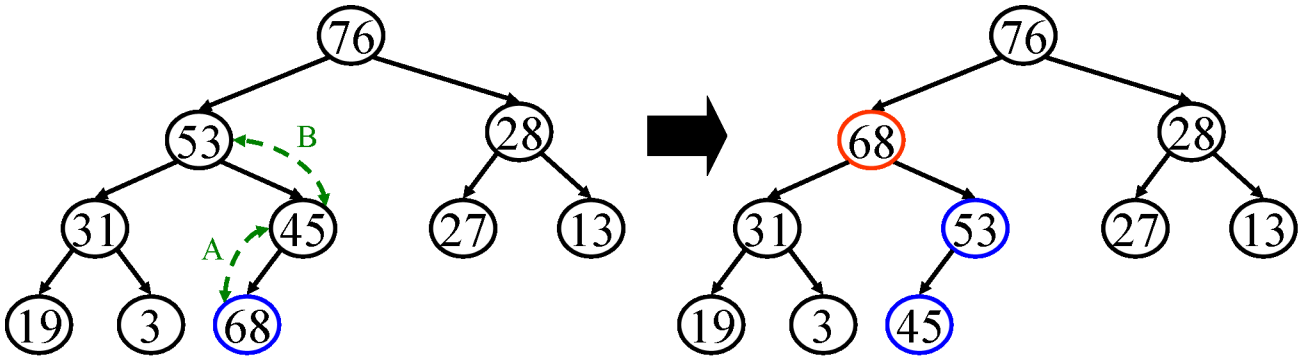


Figure 11: Insertion of the value 68 into the heap.

will only concern ourselves with max-heaps, but the max-heap representation and operations are analagous to min-heaps.

A heap is characterized by the heap property. For a max-heap, the value at a node is greater than any value contained in either of its subtrees. Therefore, the maximum value in the heap is always at the root and can be retrieved in constant time. In addition, a heap is required to be a complete binary tree.

Besides retrieving the maximum element from a heap, we want to insert arbitrary elements and remove the maximum element. In both cases, we do a simple insertion/removal that may result in a tree in violation of the heap property, but then do operations to restore the heap property.

To insert an element in a heap, we first insert it at the bottom in the proper position to preserve the completeness of the heap. Then we *reheapify up*. Starting at the inserted value, we compare the value to its parent. If it is greater than its parent, we swap the two, and continue with its new parent. Otherwise, we know the heap property is intact, so we stop. In the worst case, the new value is the greatest value in the heap, so we have to swap values until we reach the root. But since the heap is a complete tree, this is at most $\lg n$ swaps, so the insertion is logarithmic in the number of values in the tree.

Removing the maximum element is slightly more complicated. We first replace the maximum element with the last element in the heap (the rightmost element in the bottom level) and then *reheapify down*. Starting at the new root, we compare the value at the current node with each of its children. If either of the children are greater than the current value, we swap it with the greater child (greater of the two children if both are greater than the parent) and continue with its children. If neither child is greater, we stop. This can continue until we have reached the bottom of the heap, but this means at most $\lg n$ swaps again. Thus, removal of the maximum element is logarithmic as well.

Rather than representing the heap as an actual tree, we can use an array representation since the heap is

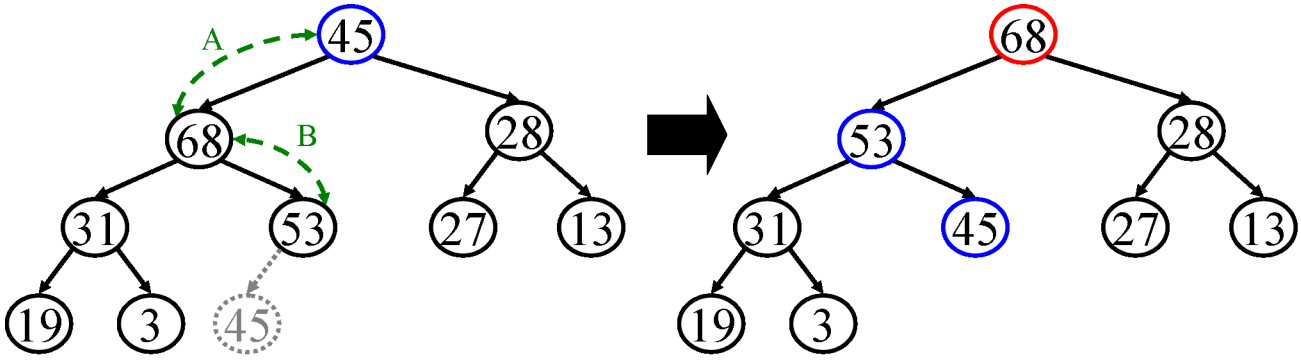


Figure 12: Removal of the maximum value 76 from the heap.

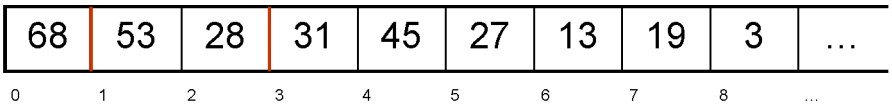


Figure 13: An array-based heap representation.

complete. We store the root value at position zero in the array. Then for a value at position k, its children are stored in position $2k + 1$ and $2k + 2$. The resulting array has no empty spots between values, and the array representation reduces the space and time requirements by constant factors.

While heaps only allow us to retrieve or remove the maximum element in a set, many times this is all that we require. Structures that only allow these operations are called *priority queues* and are useful in many ares, such as graph algorithms.