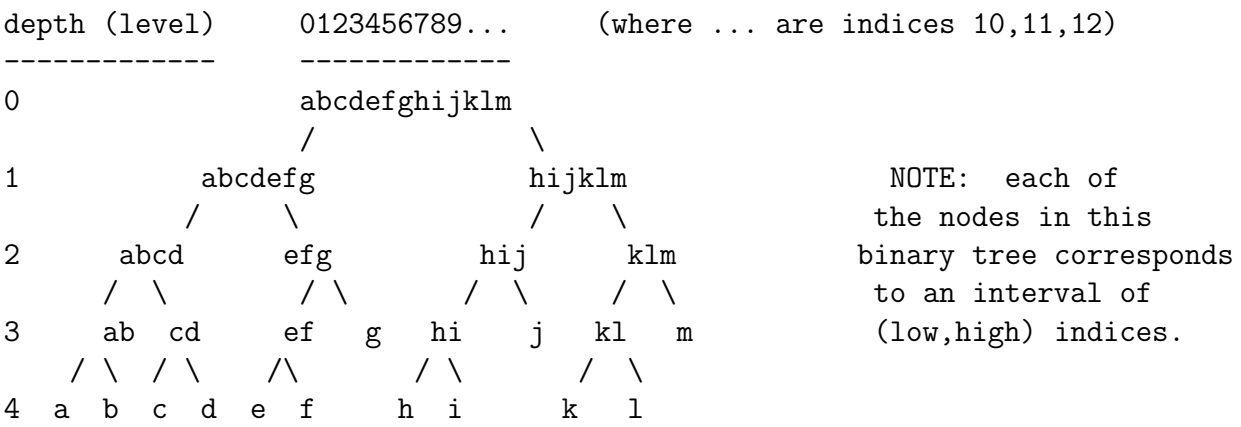
Recall lecture 11 where we considered how to search for an item in sorted array. The idea was to use recursion and to divide the search interval in half at each step. We kept track of indices `low` and `high`, computed their average at each step, and then checked if the value we were searching for was in the lower half or the upper half. This was a very efficient way to search. In particular, it required only $O(\log n)$ steps. (We say big O rather than Θ because sometimes it terminated very quickly e.g. if you were lucky and the value you were searching for happened to be the one with index at the average value.)

There is a problem with using arrays, however, namely it is relatively expensive to remove and add items since you need to perform a sequence of shifts. Today we will look at another way to do binary search, which uses a binary tree instead of an array.

Before we begin, notice that we can think of the array based binary seach algorithm we saw earlier as defining a binary tree where the root node represents the whole array, and the left and right children represent the two subarrays into which the whole array is partitioned (and so on, recursively). You can think of the choice of which of the subarrays to search through as being a choice of a left vs. right child in a binary tree. *The array based binary search does not explicitly construct this tree, of course.* Rather, I mention this binary tree structure so that you can get an intuition of how an array based binary search is related to other binary trees which we will be discussing shortly.



Binary Search Trees

Last lecture we defined a binary tree which stored some element of generic type `T` at each node. Let's now consider the case that we can define an ordering on the elements stored in the tree. In Java language, we would say that type `T` is/implements/extends `Comparable`. Thus, the elements of this type have a `compareTo()` method that defines an ordering. Let's assume for simplicity that `a`.compareTo(`b`) returns 0 only if `a` and `b` are the same object. That is, we are not going to allow for repeated elements. This gives us the following definition.

A *binary search tree* is a binary tree such that:

- each node has a distinct element
- all all nodes (if any exist) in the left subtree are less than the root[1]

[1]Slightly different from the condition I wrote on board in class, which was too weak.

- all nodes (if any exist) in the right subtree are greater than the root
- both the left and right subtrees are binary search trees

Note that the last condition makes this a recursive definition.

Also note that this definition loosely equates a comparison of two nodes with a comparison of the elements at the two nodes. Suppose I define a binary search tree node as follows:

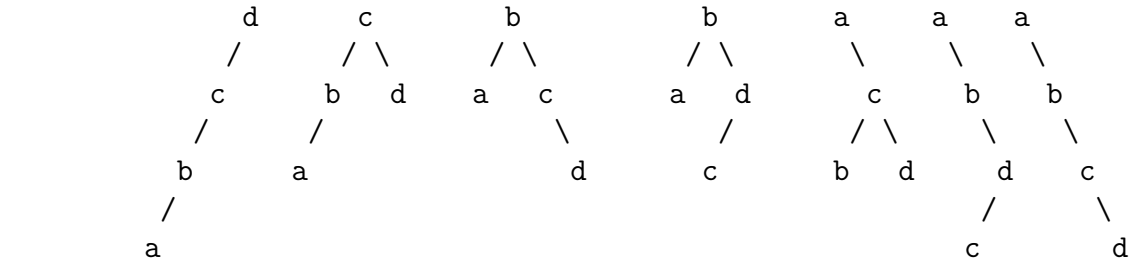
```
class  BSTNode<T>{
   T             e;
   BTNode<T>   left;
   BTNode<T>   right;
}
```

where objects of type **T** are `Comparable`, i.e. they have a `compareTo()` method. The definition loosely equates a comparison of two nodes with a comparison of the elements of type `T` referenced by these two nodes. Of course, when one is implementing a binary search tree in Java, one needs to be more precise (and, in particular, one compares the elements, not the nodes).

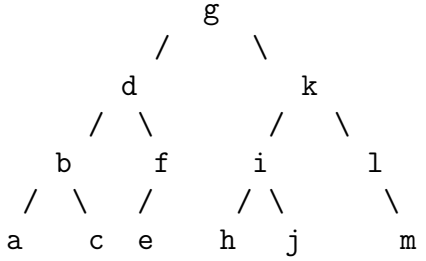
Examples

Given an ordered sequence of elements, there are many binary search trees that one can construct from these elements.

Here are several examples for four elements `a,b,c,d` which are in increasing order. (There are other examples.)



Here is an example with nodes `abcdefghijklm`:



Let's sketch out several operations that typically would like to perform with binary search trees.

find(e)

The first is to search for a particular element e. We first examine the root and compare e to the element at the root. If e is less than this element, we recursively proceed to the left child; if e is greater than the root, then we recursively proceed to the right child. The only remaining possibility is that e equals the element at the root, in which case we have found e in the tree.

findMinimum()

What if we wanted the find the minimum element in the tree (which happens to be `a` in the above examples). The minimum element has to be in the leftsubtree, if the left subtree is non-empty. So to search for the minimum element, we go from the root to the left child (if it exists) and then recursively find the node with the minimum element in the left subtree. We have found the minimum element when the node we are examining has no leftsubtree. Notice that this can occur if the element is not a leaf i.e. it can occur if the element has a right child but no left child. In the first example above, the last three elements have `a` at the root node !

findMaximum()

A similar algorithm is used to find the maximum element in the tree. Here we proceed to the right subtree (if it exists). Again, the maximum element need not occur at a leaf. In the first example above, `d` is the maximum element but it is not always at a leaf.

add(e)

Let's next consider adding an element to the tree. We proceed as with `find(e)`. If the element e equals an element already in the tree then we do nothing (rather than adding a duplicate). We proceed down the tree following the left or right child, until we either find a node whose element is e or there is a null reference where a node with element e should be (if it were already in the tree). In this case, we create a new node with e as the element, and then replace the null reference with a reference to this new node.

remove

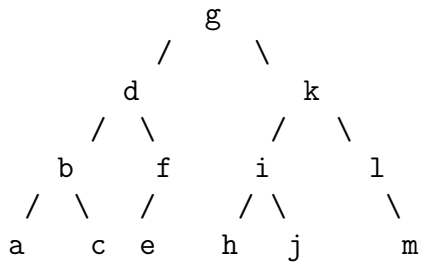
The most tricky operation on binary search trees is to remove a node. There are two cases to consider. If the node is a leaf, removal is easy. We just replace the reference to the node (from its parent) with a null reference. The tricky case is that the node is internal. The reason it is tricky is that we need to be sure we maintain the binary search tree property.

Rather than removing this internal node, we replace the element at the node with the smallest element in the node's right subtree[2] We then remove the subtree node whose element we have used.

[2] We could alternatively replace it with the largest element in the node's left subtree, but this is not the traditional approach.

Example

Take again the example with nodes `abcdefghijklm`:



and then remove elements `g,f,k,h` in that order.