Let's next consider adding a key to a binary search tree. If the key is already there, then do nothing. Otherwise, make a new node containing that key, and insert that node into its *unique* correct position in the tree.

```
insert(root,key){
   if (root == null)
      root =  new BSTnode(key)    //  makes a new node and returns it
   else if (key < root.key){
      root.left = insert(root.left,key)
   else if (key > root.key){
      root.right = insert(root.right,key)
   return root
}
```

Here is an algorithm for removing a node from a binary search tree.

```
remove(root, key){
    if( root  == null )
        return null
    else  if ( key < root.key )                               (**)
        root.left = remove( root.left, key )
    else  if ( key > root.key )                               (**)
        root.right = remove( root.right, key)
    else  if root.left == null                                (*)
        root  =  root.right  // or just "return root.right"
    else  if root.right == null
        root  = root.left   // or just "return root.left"
    else{
        root.key = findMin( root.right ).key;                 (****)
        root.right = remove( root.right, root.key );          (***)
    }
    return root;
}
```

There are a few subtleties in the algorithm that need explaining. Let's consider different cases. First, suppose *the node to be removed is a leaf.* Go to line (*). If the algorithm reaches that line, then `root` is not null and `key` matches `key.root` and so we want to remove the `root`. If `root` is a leaf, then condition (*) will be true. Moreover, `root.right` also will be `null`, and so the algorithm will return `null`. Why does this remove the node from the tree? On the one hand, if the node to be removed is a leaf and the tree has just this one node, then `null` is just the modified tree, namely the tree with its one node removed. On the other hand, if the node to be removed is a leaf and the tree has more than one node, then the current `remove()` call must be a recursive call, and it must have been called from the parent, namely from one of the two lines (**). The `null` value will be the returned value of to parent's `remove()` call from one of the lines (**), and so `null` will be assigned to the parent's child. Thus, the desired node is correctly removed in this case.

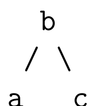
Second, suppose *the node to be removed is internal.* In this case, when we remove the node, we need to be sure that we maintain the binary search tree property. The "standard" technique[1] to remove the smallest node in right subtree, and to replace the current node with that smallest node. This is done recursively, since the smallest node in the right subtree might itself be an internal node.

We remove the smallest node in the right subtree, in two steps. We copy the key from the smallest node in the right subtree into the node to be removed, and then (recursively) remove the node containing the smallest element in the right subtree.

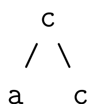
[ASIDE: In class, one student (CM) asked whether we could replace the line (***) by

```
remove( root.right, root.key );
```

This won't work, however. Consider the following example:



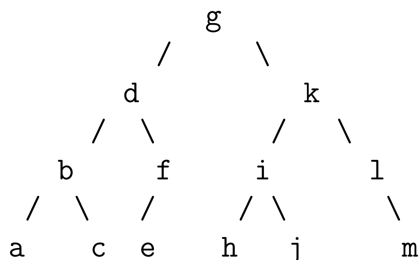
and suppose we call `remove( root, b)` where `root` is the node containing `b`. Since the keys match, the line (****) is first executed, and so `root.key` takes the value `c` and the tree becomes:



and then, as suggested, the `remove( root.right, root.key )` would be executed. This would cause (*) to be executed which would return `null`. However, the right child reference of the root node would not be reassigned to this `null` value, and so the tree would be stay as it is (above), namely with two copies of `c`. This is an error.]

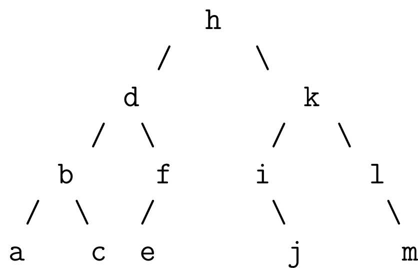
Example (remove)

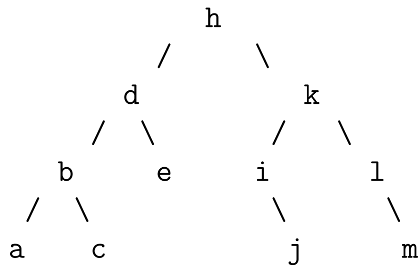
Take the following example with nodes `abcdefghijklm`:

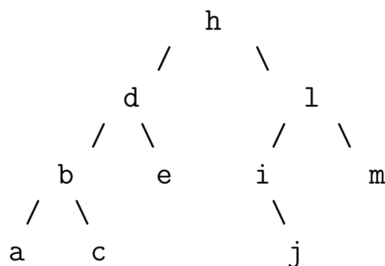


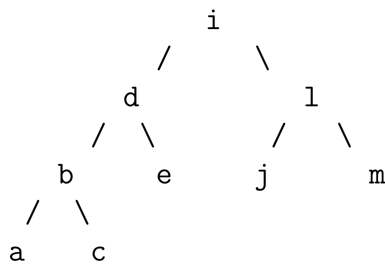
and then remove elements `g,f,k,h` in that order.

[1]The standard instead could just as easily have been to remove the largest element in the left subtree.









Discussion: binary search trees vs. binary search

As a final point of discussion on binary search trees, let's return to the `find` method that we saw last lecture. Finding a key in BST is reminiscent of a binary search, in that we compare a search key to given key in the tree and if the search key is less than the given key then we continue by considering the keys that are smaller that the given key. (Otherwise we continue by considering the keys larger than the given key).

There are several differences between the BST `find(key)` method and the binary search method that we saw earlier. First, the BST uses a tree, whereas binary search used an array.

Second, if we find the key at an internal node of the BST then the algorithm stops and returns the node, whereas in our binary search algorithm, we kept going until `low == high`).

Third, the binary search always reduced the number of items to be searched through by a factor 2 (or roughly 2, if n is odd) at each step. For a binary search tree, however, we are *not guarenteed*

to reduce the number of keys to be searched by a factor of two at each step. Indeed, the worst case that the BST is one long branch (with a single leaf), and so each recursive call only reduces the number of remaining nodes to be searched by 1 .

Advanced Discussion (definitely not on exam !)

In COMP 251, you will learn about data structures and algorithms for adding and removing keys to/from a BST such that the BST height is $O(\log n)$, rather than $O(n)$. This ensures a fast search. These trees are called "red-black" trees.

In more advanced courses, some of you will study *random trees*. You may carry out a probabilistic analysis of how binary search trees grow when random elements are added.[2] For example, here is a "back of the envelope" sketch of why the expected height[3] of a binary search tree is $O(\log n)$.

Let $t(n)$ be the expected height of a binary search tree with n nodes. Suppose we have a BST with n nodes, where n is big. If we add a $n+1^{st}$ node, the probability that this node will extend the height of the tree is about $\frac{2}{n}$. The reasoning is as follows. First, if h is the height of this tree, then it is very unlikely that there will be more than one path of length h. Thus, if adding a $n+1^{st}$ node extends the height by 1, then we only need to consider one leaf from which the height could be extended. In order for the height to be extended, the element that we are adding has to come either just before or just after this leaf. Since there are n nodes, the probability of it coming just before this leaf is about $\frac{1}{n}$ and the probability of it coming just after is about $\frac{1}{n}$. (If you want to know what its not exactly $\frac{1}{n}$, then email me or post the question on webct.)

Now, the expected height of the tree with $n+1$ nodes can be written in terms of the expected height with n nodes as follows:

$$
\begin{aligned}
t(n+1) &\approx \frac{2}{n}(t(n)+1) + (1-\frac{2}{n})t(n) \\
&\approx \frac{2}{n}t(n) + \frac{2}{n} + t(n) - \frac{2}{n}t(n) \\
&= \frac{2}{n} + t(n)
\end{aligned}
$$

The argument about the probability being $\frac{2}{n}$ only holds for large n, say $n \geq n_0$. So we back substitute to n_0:

$$
\begin{aligned}
t(n) &\approx 2(\frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \cdots \frac{1}{n_0}) + t(n_0) \\
&\approx 2\log_e n - 2\log_e n_0 + t(n_0)
\end{aligned}
$$

where we used the fact that

$$
\sum_{i}^{j} \frac{1}{n} \approx \int_{i}^{j} \frac{1}{x} dx = \log_e j - \log_e i,
$$

from Calculus. From here, you can easily see that $t(n)$ is $O(\log n)$.

[2]Two of our profs, namely Luc Devroye and Bruce Reed, have made many important research contributions on the topic of random trees.

[3]The "expected value" of a random variable is well-defined defined in probability theory. If you haven't seen it before, then you will see it in MATH 323 or equivalent.