Trees

1 Trees and Binary Search Trees

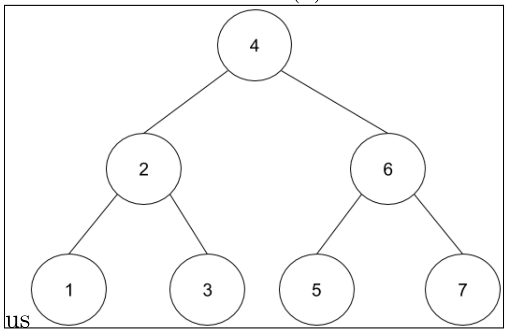
Trees, in the most basic sense, are a set of nodes connected to each other by a set of edges. Nodes can have 0 children, in which case they are called a *leaf*, or more children. A property of a tree is that every child of a tree is also a tree. The only constraint is that, from any one node, there is only one path to any other node. A node that is the child of no other node is called the *root*
A specific type of tree that is useful for ordering data is the ***Binary Search Tree***. The Binary Search Tree,

or BST for short, is a way of organizing *comparable* nodes. Nodes in BST's have between $0-2$ children. Children to the left of a node will are always "less than" the root and items to the right of a node are always "greater than" the root, we will not really worry about handling duplicates at this point of time. The API for a basic Binary Search Tree is as follows.

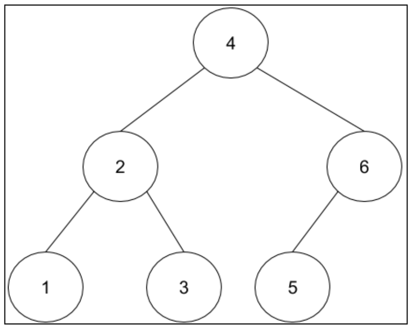
```
public BST get(BST T, Key k); //gets the node that has the given key k
public BST insert (BST T, Key k;//inserts a key k, if it does not exist already.
```

Inserting and getting from a BST are very similar processes. In both cases, you start at the root of the tree, if your key is less than the current node, you recursively go to the left node, if your key is greater than the current you go to the right and if your key is equal, you return your current node. Inserting and getting from a BST would both operations, $\Theta(log(n))$, as that tends to be the height of the tree.
The process for adding to a Binary Search Tree is relatively trivial; however, the process for removing a node

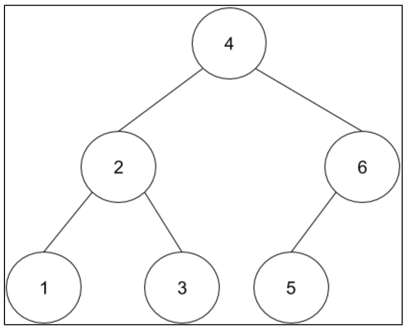
can be more complicated. In the case that a node has no children, all you do is remove the parent's link to it, then Java garbage collects it. In the case that a node has 1 child, you just make the parent of that node point have its pointer (whichever was pointing at the initial node) and have it point to the child. This works because the child of the initial node will always have the same relation to the parent that the initial node did (either less than or greater than). Below are examples of removing nodes. The first example is removing a node with no children (7).



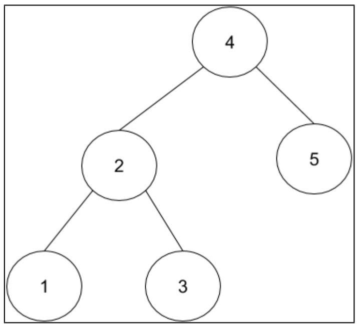
$\longrightarrow$



Below is an example of how to remove a node with only one child, in this case 6.

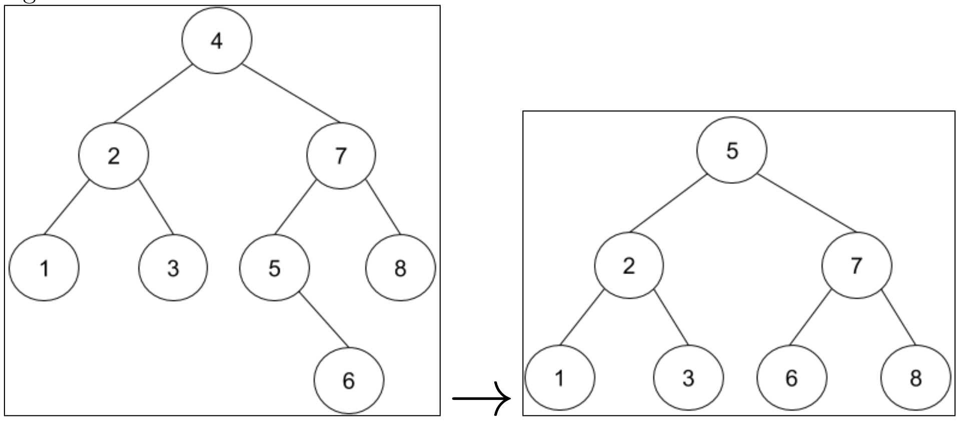


$\longrightarrow$



When a node has 2 children, things get a bit trickier. You cannot always pick the immediate child otherwise

the BST property may not always be maintained. Look at the following example.
To solve this problem, we can replace the node that is being deleted with either the node that is greatest on its left side or least on its right side. The node chosen will always have 1 child maximum, as if it had two children, it is the case that, on the right side, something would be less than it, and on the left side, something would be greater than it. Once we replace the node, we do the process for the deletion of a node with 1 child. That was a lot of words, so let's show an example, in it, we will take the smallest item on the right side of the node 4.



As we stated, runtimes of Binary Search Trees can vary based off the height of the tree. The height of the

tree is affected by the order in which keys are inserted. For example, if the first item in the tree is "2" and the second item is "1", it will look different than a tree than a tree with "1" inserted first and then "2". This means that the order in which keys are inserted can actually affect the runtime. If the keys are inserted in a good way, we will get a "bushy" tree, one that looks like a shrub and if the items are inserted in a poor way, we may get a "spindly" tree, or one that looks like a linked list.
We state the runtime of a Binary Search Tree operation like insert to be $\Omega(lg(n))$ in the worst case. This is

because each item in a bushy tree will have 2 children. This means that at each layer, we divide the problem in $\frac{1}{2}$. As we go through the layers, the problem's size becomes $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}$.... If we were to, for some reason, increase the number of children each node could have to 3 for example, the runtime for insert would be, in best case, $\Theta(log_3(N)$. We don't really care much about the base change for logarithms, but this is important so you understand how we come across these runtimes.