CSC 325 Adv Data Structures

Lecture 10

Splay Trees

Splay trees are somewhat similar to Red Black trees (which will be discussed later). They are a type of BST and follow all rules coinciding with the BST's search property. However, there is no direct attempt to keep the tree balanced. The efficiency of a Splay tree comes in the form of quick accesses to values that have been recently accessed. When a value is accessed, the node containing that value moves up to the root of the tree (via rotations). This means that if that value is accessed again, it will be retrieved a lot faster. If you know you will be searching for the same values over and over again, a splay tree will put those values at the top to make accessing them quicker. This does still mean, though, that the first time a value is searched for, we could experience a linear time cost. Here are the details for a Splay tree:

Abstract Data Type: Splay Tree

`insert(value)`	Inserts the given value into the tree using normal BST rules. [optional] Calls splay function on the newly inserted node
`search(value)`	Searches for the given value in the tree. Returns true if found, false otherwise. [mandatory] If found, call the splay function on the found node
`delete(value)`	Deletes the node that contains the given value using normal BST rules. [optional] if node containing given value is found, call splay function on the parent of the deleted node
`min()`	Returns the minimum value in the tree, same as BST. [optional] if a minimum node is found, call splay function on it
`max()`	Returns the maximum value in the tree, same as BST [optional] if a maximum node is found, call splay function on it
`splay(node)`	Internal function that raises the given node up to root position by following a series of rotations

The splay function is responsible for moving a node up to the root. We call this function on the found node after a successful search of a value. Some versions of the Splay tree will also splay a newly created node after an insert call, whereas others only splay after a search. This is a design decision that forms a tradeoff. If you think you might be searching for a value shortly after inserting it, you probably should splay on insertion. Otherwise, perhaps the added cost of calling the splay function isn't worth it after an insert. Either way, splay is definitely called on a found node after a successful search.

Note that splay is an internal function (i.e. a "private" function). There is no need to expose it to any outside classes and this function should be called automatically by the Splay class at the end of a search or, optionally, at the end of an insert, delete, min or max call.

Delete operation

Splay trees can delete nodes and, just like insert and search, will follow all the normal rules of a BST. When a node is found and deleted, its parent can be splayed (an optional addition to the delete function). If no node was found matching the given value, then no node is deleted and splay isn't called. In case you are unfamiliar or just plain don't remember how to delete a node from a BST, allow me to duplicate my lecture notes from 220 here as a reference:

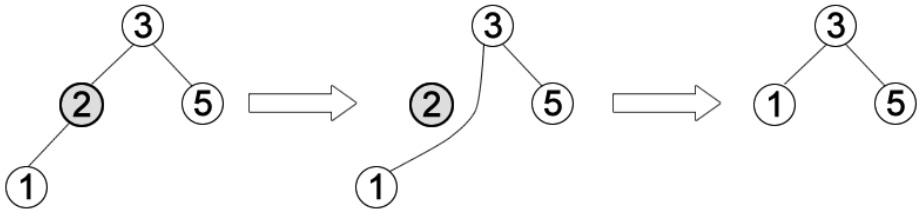
First, we are going to need an extra operation to be added to our binary search tree. In an actual implementation, this may be a private function or may just be incorporated into the logic of the delete function (in my own implementation in the demo, I just incorporated the logic into my delete function rather than make it its own function). Here is the operation we must add:

`successor(node)` Required for the delete operation. This operation finds the successor for the given node. This is the node containing the value that would be printed next in an inorder traversal. This is found by visiting the right child, and then continually going down the left child path until we don't have a left child (i.e. the left child link is null).

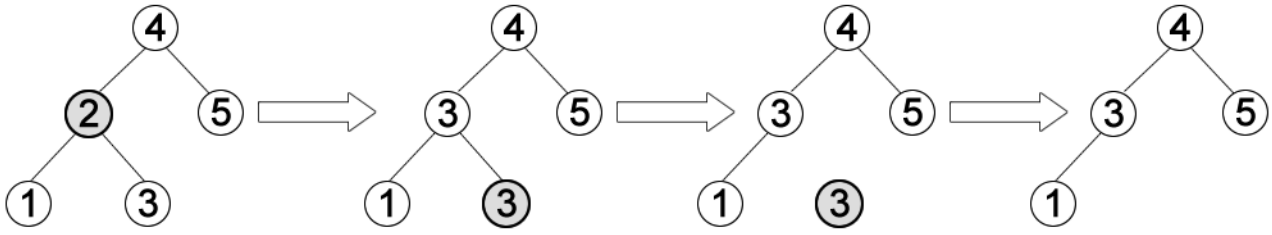
To delete a node, we need to consider the different cases we can be in. This is similar to the cases we covered for linked list. Here are the results from calling `delete(2)` on different trees.

CASE 0. The node to be deleted has 0 children (i.e. it is a leaf). This is easy: just modify the parent's link to this node by making it null.

CASE 1. The node to be deleted has 1 child. This is also easy: just connect the parent to the child. This is similar to deleting a node in a linked list, we simply redirect the pointer to "hop over" the deleted node.

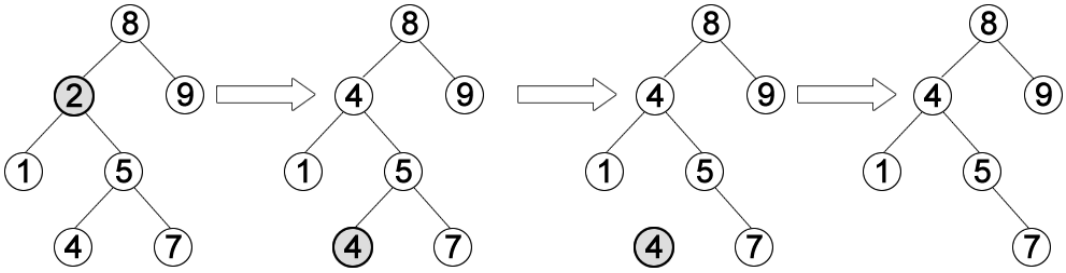


CASE 2. The node to be deleted has 2 children. This is our more difficult case. This is where our `successor` operation comes in. First, we find the node's successor and overwrite the node's value with the successor node's value. Now perform the deletion operation on the successor node. This requires us to revisit either case 0 or case 1 on the successor node.



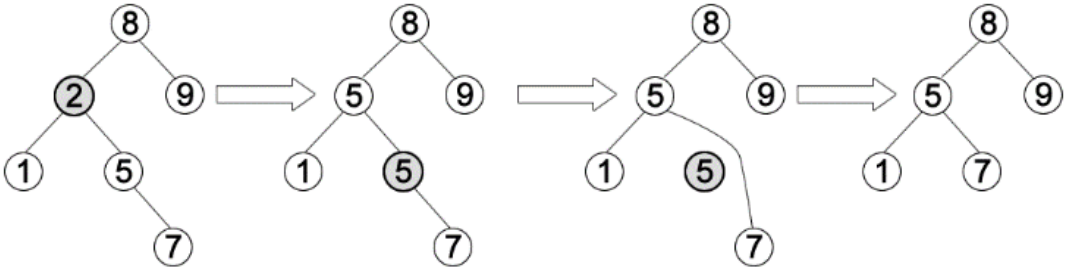
First, we found the successor of 2 by visiting the right child (which is guaranteed to be greater) and then continuously visiting the left child (going down the tree) to find the minimum value that is greater than 2. In the case above, the node containing 3 had no left child, so it was the successor. We copy the value of the successor, 3 over the node-to-delete's value of 2. After replacing the value, we turn our attention to deleting the successor node (now shown in gray). To delete this node, we refer to case 0 since this node doesn't have any children.

Here is another case 2 situation.



In this case our successor wasn't the right child. Since the right child has a left child, we look to that for our successor value. We then replace the value of 2 (our node-to-delete's value) with our successor value of 4. Now we run the delete operation on the successor node. This is in case 0, so we simply remove the link to it from the parent node.

One final example, this time the successor node will be in case 1:



After replacing 2 with 5, we need to delete 5. Since we are in case 1, we jump the parent pointer over the node and onto the child node (i.e. 7).

Another Way to Delete

Note that there is another way to delete from a binary search tree. Case 0 and case 1 are pretty straightforward but case 2 is the tricky one. We chose the successor value to replace the deleted value, but is this the only choice? The logic is that the successor is guaranteed to be larger than all values in the left subtree and smaller than all values in the right subtree, thereby maintaining the search property of the tree. However, the *predecessor* satisfies this as well. For the predecessor we simply go to the left child and then recursively go right until we can't anymore. Both successor and predecessor approaches are equally as good and you have an equal chance of finding either one if you look online for deletion in a binary search tree. To make grading easier, however, I want everyone to use the successor method described in the images above. After this class, feel free to use whichever method (successor or predecessor) you like, unless explicitly told otherwise.

Splay operation

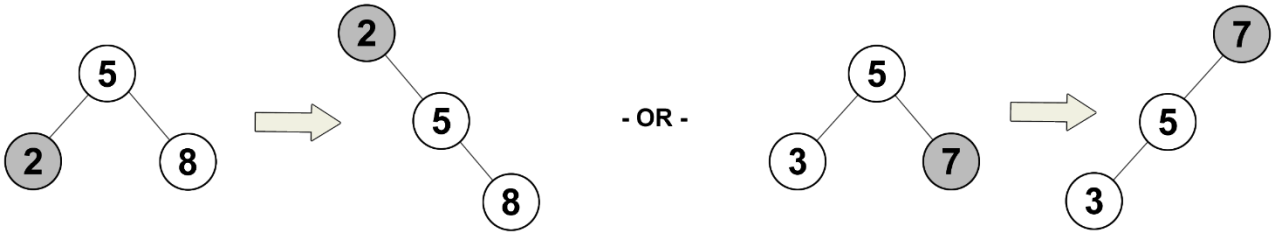
Since the only difference between a normal BST and a Splay tree is the splay operation, let's focus on that. I won't revisit how to insert or search for values since this works EXACTLY like it does for a BST with the only addition being the splay function call at the end on either the newly inserted node or the found node. Also, min and max work exactly the same as well. Whether you splay the min or max node after finding it, is up to you. You could technically consider this to be a search and therefore fair game for splaying. Your call.

There are really just 5 cases we can be in when we splay a node. Here is a quick reference:

Left Case (no sibling) **Right Case (no sibling)**

The shaded node is the node that was just inserted (or searched for). The images show before and after splaying the shaded node. If the splayed node is a left child of the root, the splayed node becomes the root and root becomes right child of splayed node. If the node to be splayed is a right child of root, splayed node becomes root and root becomes left child of splayed node.

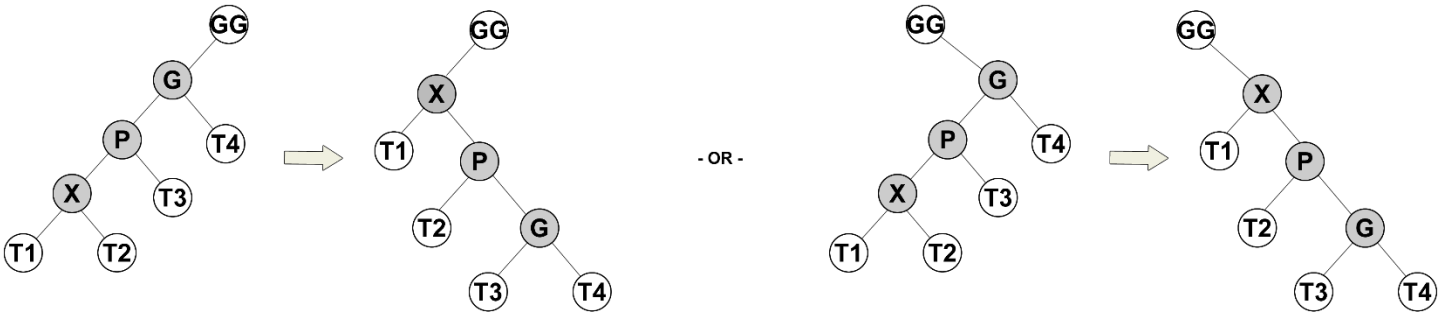
Left Case (with sibling) **Right Case (with sibling)**



These show you what to do if the newly inserted (or searched) node has a sibling. Obviously, you will need to think about if that sibling has children, but the code should work its way out as long as you move the sibling node to the right place (i.e. the child nodes will follow).

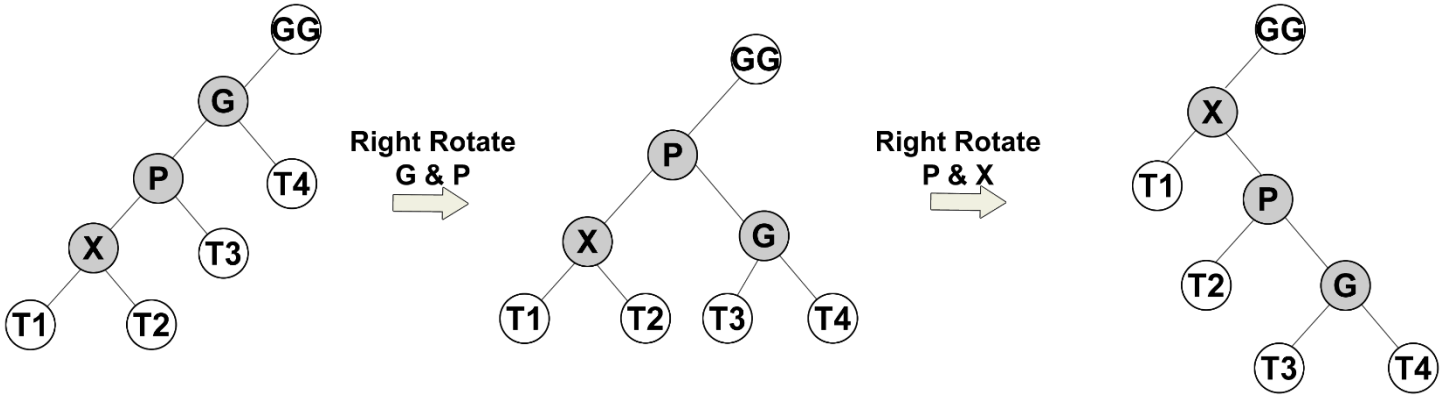
Left/Right case only happens when the node we are splaying is a direct child of the root, i.e. it does not have a grandparent. However, if the node does have a grandparent, then we need to move onto one of the next cases.

Left Left Case

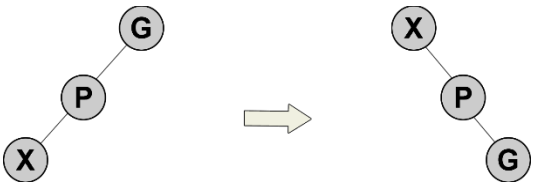


I'm using abbreviations for the nodes instead of showing their real values. P is parent, G is grandparent, and X is the node we are splaying. I added a GG node for great grandparent because you really do have to consider this during implementation, so it helps to visualize it. The Left Left case happens when you go left from the grandparent and left from the parent to get to your splay node X. With this there can be two cases depending on whether the grandparent is left or right of the great grandparent. There may not even be a great grandparent if the grandparent is the root.

This case could also be broken down into parts. We could perform a right rotation on G, then a right rotation on P. This would give us the same result:

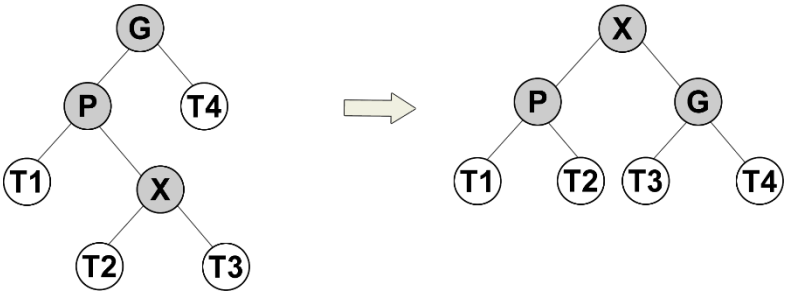


A, B, C and D could be part of bigger subtrees, or they may not even be there (they may be null). In fact, all you really need is the following to be in the Left Left case:



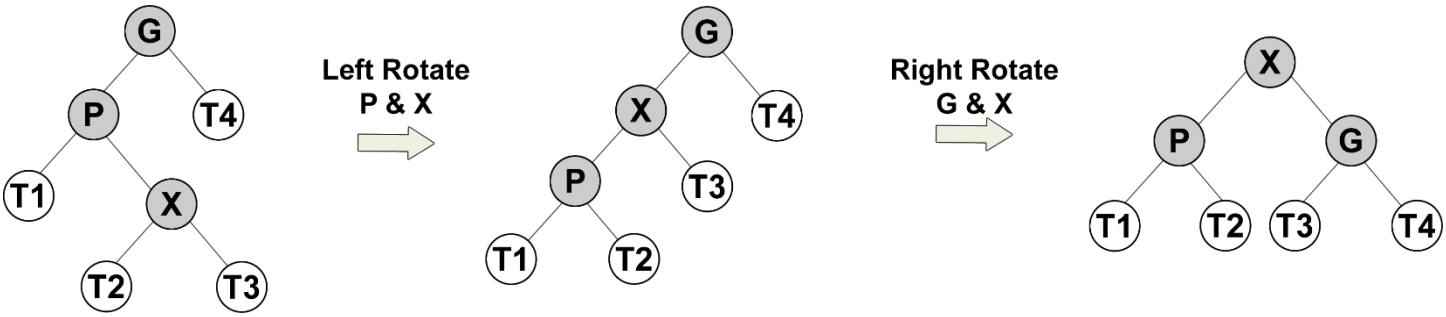
However, I find looking at the full potential tree state helps me know where to put everything in case there are more nodes than the simple case shows.

Left Right Case

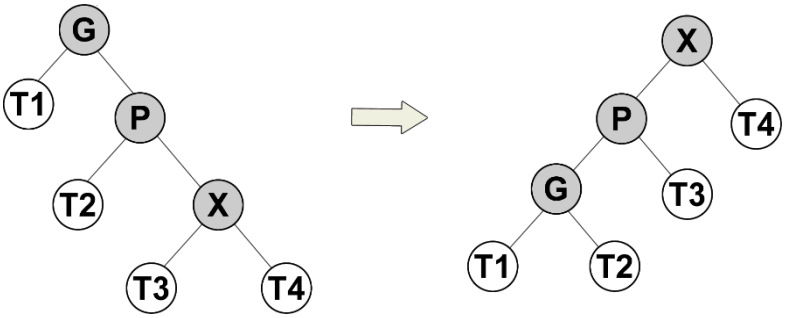


The Left part means going left from G to P and the Right part is going right from P to X. It's interesting that even though being balanced is not a direct goal of the splay operation, sometimes this happens as a result of moving the splay node up.

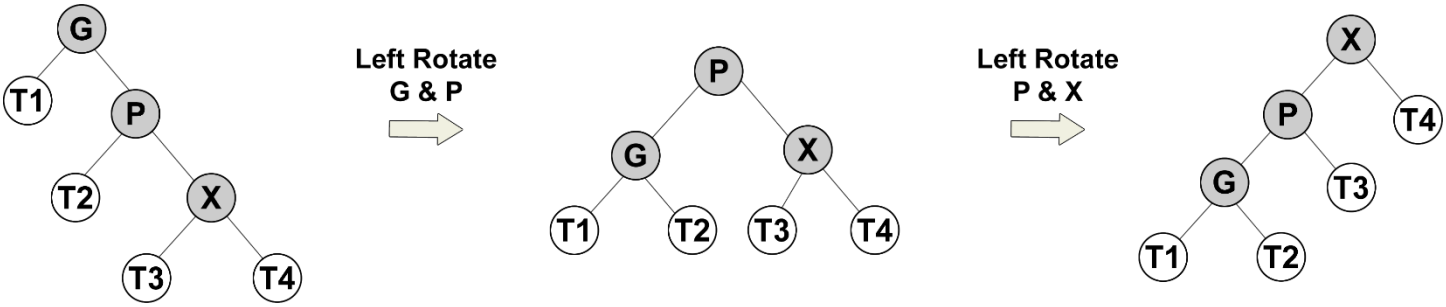
Here is the same Left Right case broken down by rotations:



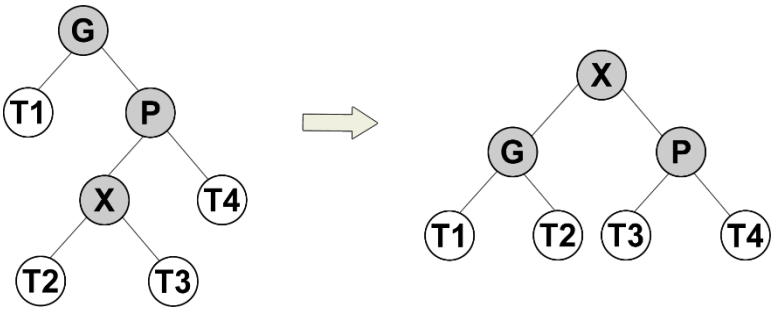
Right Right Case



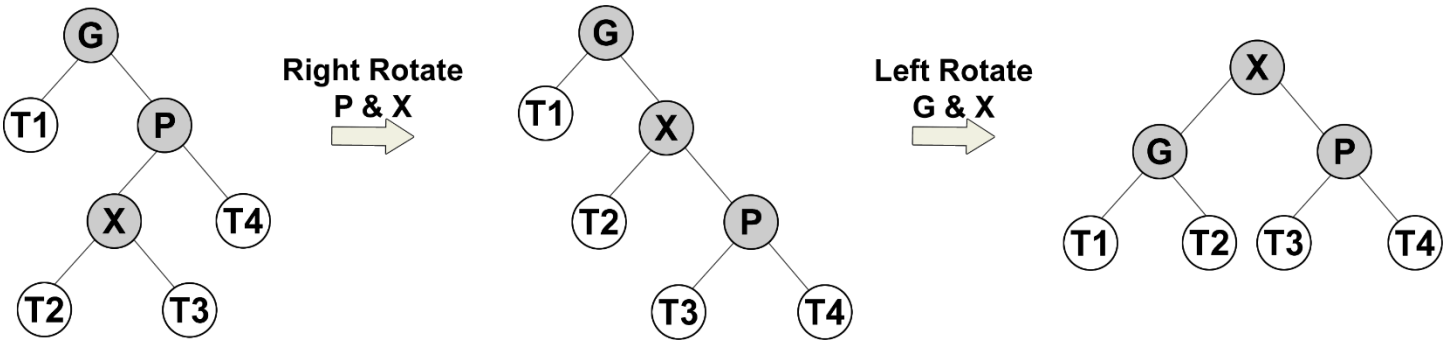
This is the same as Left Left but in reverse. X is right of P and P is right of G. We just move things back to a Left Left case (with G and X in different spots). If you prefer to view it as a series of two rotations, we can view it like this:



Right Left Case



And with the rotations broken down:



Final Remarks

There are tons of different trees out there! Some remain balanced (such as 2-3 trees), some don't. Some optimize in other ways. Some have various properties that go into their optimization strategy. Splays are very common and are a simple tweak to the standard binary search tree. Later we will learn about more ways we can alter the basic BST data structure.

References:

- https://en.wikipedia.org/wiki/Splay_tree
- https://www.geeksforgeeks.org/splay-tree-set-1-insert/