Advanced Computer Contest Preparation
Lecture 26

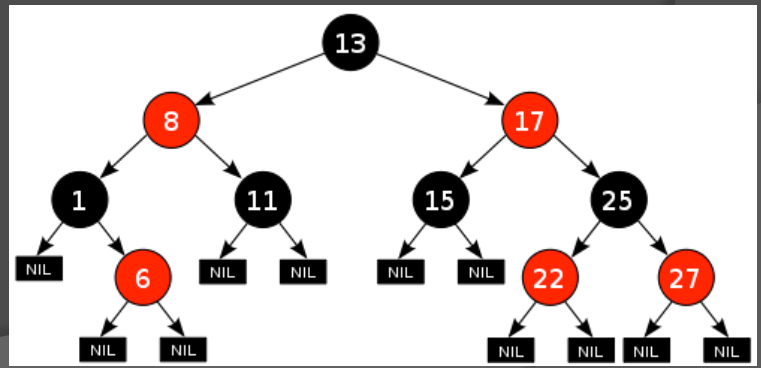
BINARY SEARCH TREES: SPLAY TREES

Problem

- You have an array
- We can perform the following operations:
  - Insert a value
  - Delete an instance of a value
  - Find a node with a given value
  - Get the x^{th} smallest value
  - If the array was sorted, get the index of a given value

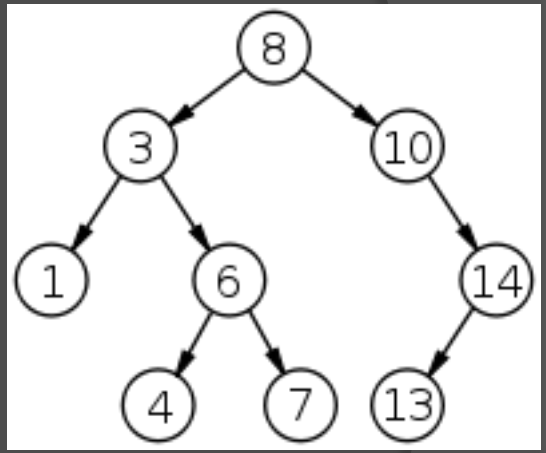
set

- C++ **set** implementation is a balanced binary search tree (red black tree)
- **set** can only support the first three operations
- For the other two operations, we implement our own binary search tree
  - Or, cheat and use **pb_ds**



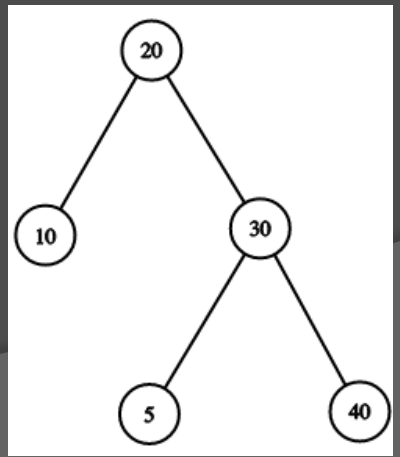
Binary Search Tree

- A binary tree that allows for quick insertion, deletion, and searching of elements
- BST property:
  - All descendants to the left of a node have a value less than that node's value
  - All descendants to the right of a node have a value greater than that node's value
  - Descendants with equal value can either be on the left or right (exclusive), or be compressed into one node



Incorrect BST Property

- The following property alone does not always follow BST property:
  - The left child node's value < current node value
  - The right child node's value > current node value



Operations

- BSTs should support 3 basic functions:
  - Insert
  - Delete
  - Find
- There can be multiple find functions with different purposes
- Find functions can be more complex, such as finding the x^{th} smallest element in the BST

Find

- BST property:
  - Left descendants < node < right descendants
- 3 cases:
  - Query value = node value
    - Return the node
  - Query value < node value
    - Go left
  - Query value > node value
    - Go right
- It is possible that no such value exists

Insert

- Follow the same steps as find until the current node is null
- Instantiate a new node
- Make sure that the node is properly linked to the tree!
- Common mistake:

```
Node *nd
//use nd to go down the tree until it is
null
nd = new Node();
```

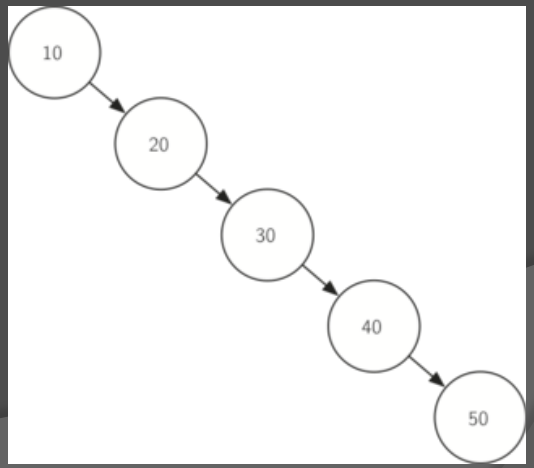
  - The last non-null node of the tree that `nd` pointed to will not point to the new node

Delete

- Locate the node that is to be deleted
- 3 cases:
  - No children
    - Simply delete the node
  - 1 child
    - Replace node with the child
  - 2 children
    - Find the smallest value node in the right subtree (the leftmost node) and replace the value of the node to be deleted
    - Delete the other node
    - Can also use the largest (rightmost) node in the left subtree

The problem with BSTs

- Suppose we insert elements in increasing order
- The BST looks like a linked list
- Any operation can take at most $O(N)$ time to complete



Animations

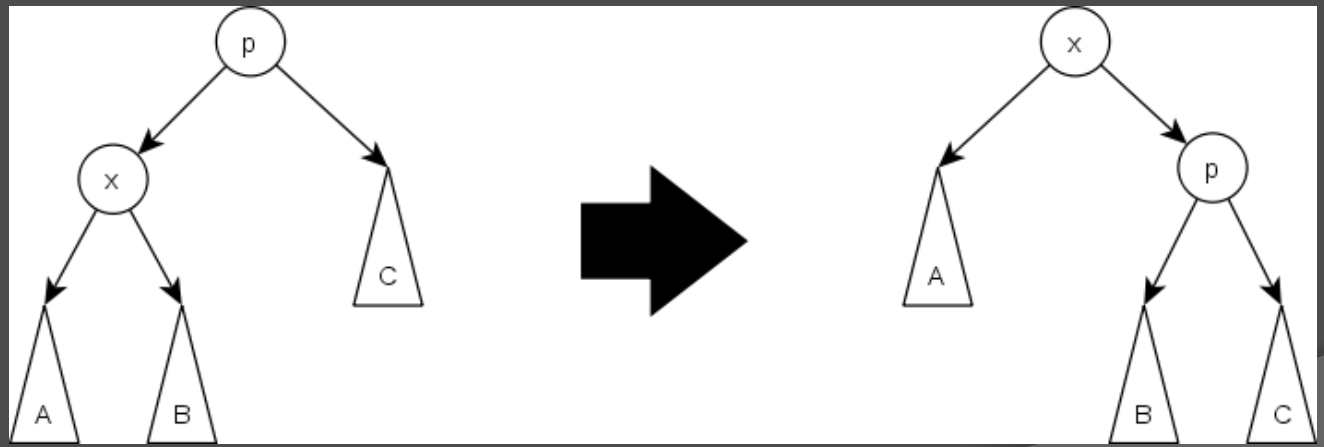
https://www.cs.usfca.edu/~galles/visualization/BST.html

Splay Tree

- A self-adjusting binary search tree
- Not technically a balanced binary search tree, but has similar worst-case runtime
- Key concept: whenever a node is accessed (inserted, queried, etc.), move it to the top of the tree
  - Principle of Locality

Rotate

- Given a node and its child, “rotate” the tree such that the node is now a descendant of the child and BST property still holds



Splay

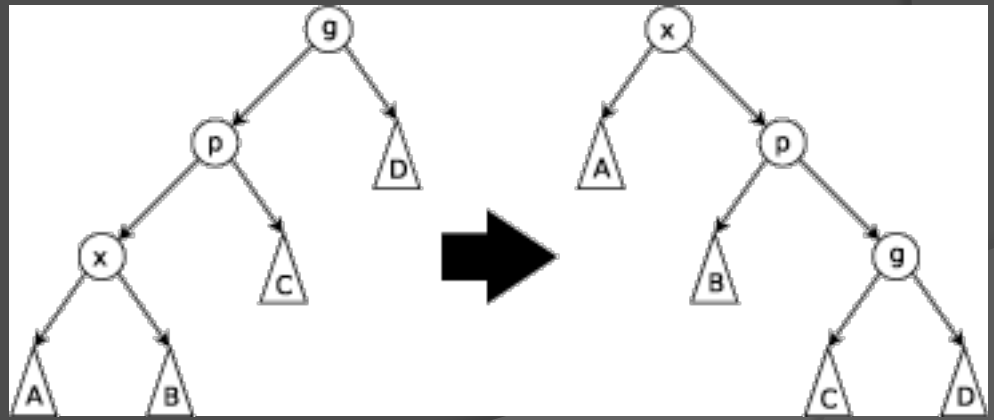
- A special method of moving a node to the top of the tree using rotations
- Pay attention: splaying is **not** the same as repeatedly applying single rotations to the node
- Splay should be used after accessing a node (insert, query, etc.)

Splay

- Case: Node is at the root
  - Stop rotating, the splay operation is complete
- Case: Node's parent is the root
  - Rotate the node once to make it the new root
  - Called "zig"

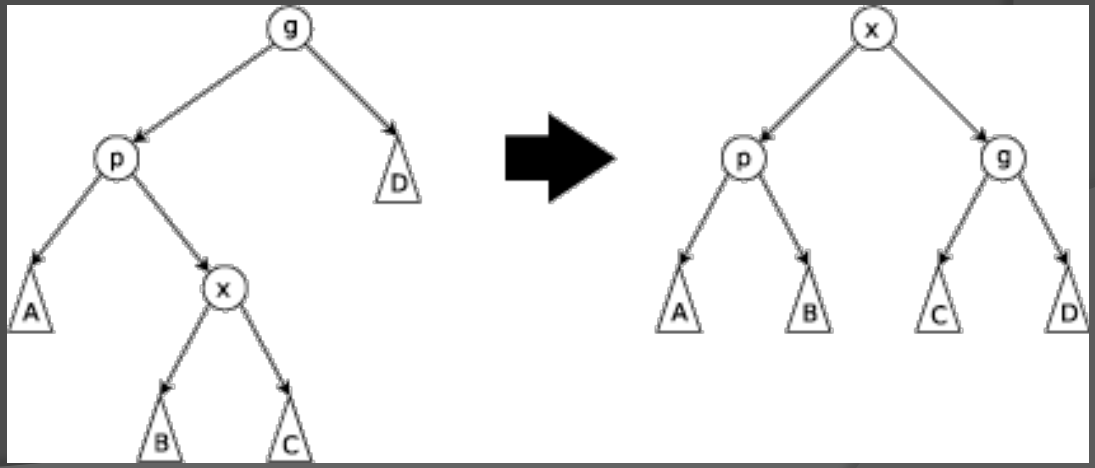
Splay

- If the node's parent has a parent:
- Zig-zig:
  - Node's direction is the same as parent's direction
  - Rotate parent node
  - Rotate node



Splay

- Zig-zag:
  - Node's direction is the opposite of its parent's direction
  - Rotate node twice



Insert

- Insert elements the same way as you would insert them into a normal BST
- Remember to splay the inserted node afterward

Delete

- Find the node that is to be deleted
- Splay it to the top
- Remove the node, splitting the splay tree into 2
- Find the largest node of the left tree and make it the new root
  - Rightmost node of the left tree
- Attach right tree to the new root
- If no left tree exists, make the right tree the new tree (no need to further change)

Find

- Find nodes using the same way that you would find a node in a BST
- Remember to splay the node after finding it
- Splay the last node accessed if we don't find the node we wanted
  - Same with delete, if the node we want to delete doesn't exist, splay the last accessed node

Animations

https://www.cs.usfca.edu/~galles/visualization/SplayTree.html

Analysis

- Time complexity of insert, delete, and find are all amortized $O(\log N)$ time
  - $O(N)$ time can happen, but the tree's structure will change so that it does not happen again in the near future
- Splay operation makes time complexity $O(\log N)$
- Naive rotations alone (only rotating the current node up) do not improve the worst case time complexity of $O(N)$

Tips

- Instead of using `left/right` for child node pointers, use an array of size 2 (`child[2]`)
- Functionalize rotate and splay code
  - Try only writing 1 rotate and splay function
  - Hint: use booleans that specify direction
- Be careful of what pointers are pointing to
  - Make sure nodes are pointing correctly to children
  - Make sure nodes are pointing correctly to parent
  - Make sure root is pointing to the top of the tree
  - Make sure root's parent is null

Get Index

- Get the index of a given value in the BST
- Count the number of nodes with a value less than the given value
- Recall BST property:
  - Left descendants < node < right descendants
- Algorithm:
  - While the current node's value is not equal to the given value:
    - If the current node's value is less, include the node and its left subtree; go right
    - If the current node's value is greater, do not include the node and its right subtree; go left
- We need a size variable in each node to accomplish this; size variable stores the size of the subtree rooted at the node

Get Value

- Operation: get the value of the element at index i in the BST
- Use a size variable in each node
- Suppose we are at the root
- Root's index is left size + 1
- If i = root's index, done
- If i < root's index, go to the left
- If i > root's index, $i = i$ – root's index, go right
  - Size variables only store size of the subtree, not anything above it, so index should also be in terms of the subtree
- Repeat on subtree until the node is found

THANK YOU!