# Final Review Lab!

Congrats!!! You are almost done with 104!

***disclaimer, these slides only cover content not on past review labs*

# TIPS

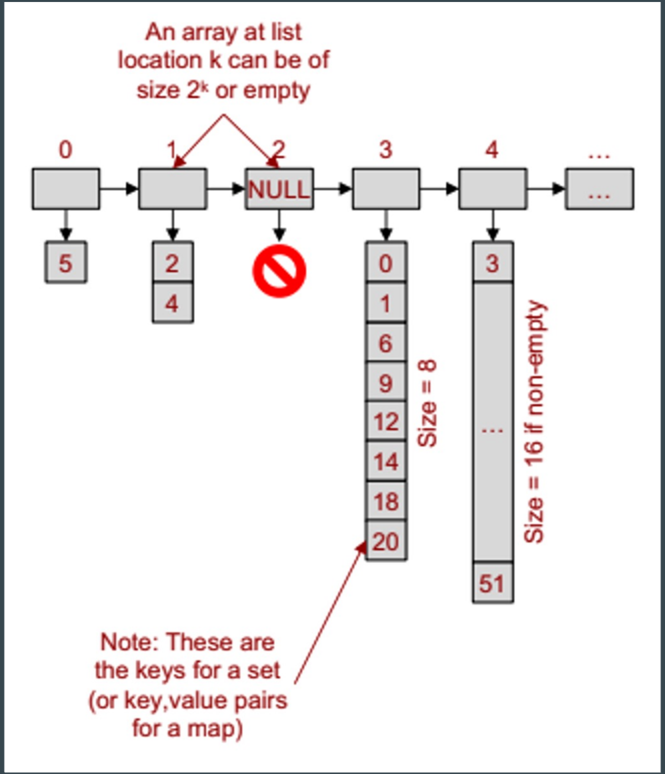
## EXAM TIME: **SATURDAY, DEC. 14th at 11:00AM**

- **Start your cheat sheet early! Potential things to include:**
  - Counting/Probability Formulas
  - ADTs and their runtimes based on implementation
  - Recursion/backtracking steps (ex. Base case, recursive step, “undoing” step)
  - Anything you have been struggling with!
- **Go through notes, labs, programming assignments, midterms, and practice exams**
- **Try to come up with your own practice problems**
- **Get enough sleep!**

# Merge Trees

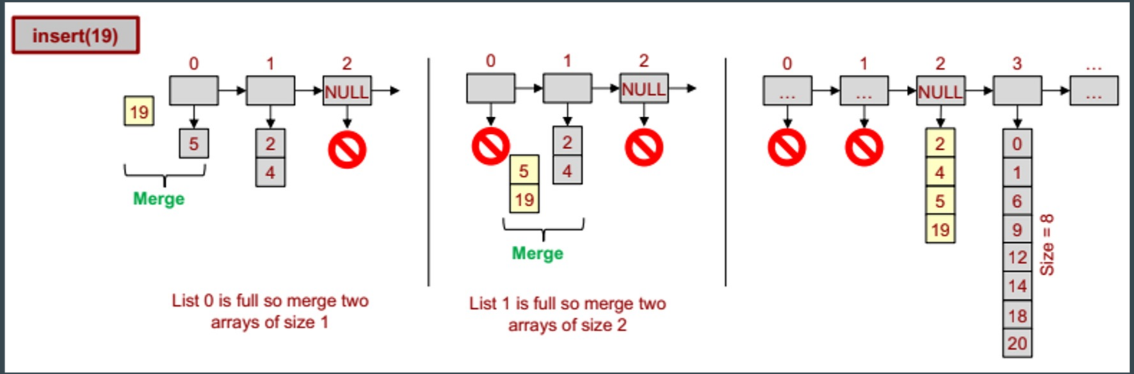
- Linked list of arrays of integers
- Array at index i of linked list (0-indexed) is exactly of size 2^i or empty
  - Each array is sorted

- Finding an element
  - Iterate through each array in the linked list
  - For each array, perform binary search
  - For k nodes in the linked list, the worst-case time of find() is:
    - T(n) = log(1) + log(2) + ... + log(2^(k-1))
    - T(n) = 0 + 1 + 2 + ... + k-1 = O(k^2)
    - k = log(n+1) for n nodes in our tree
    - Runtime = **O(log(n)^2)**



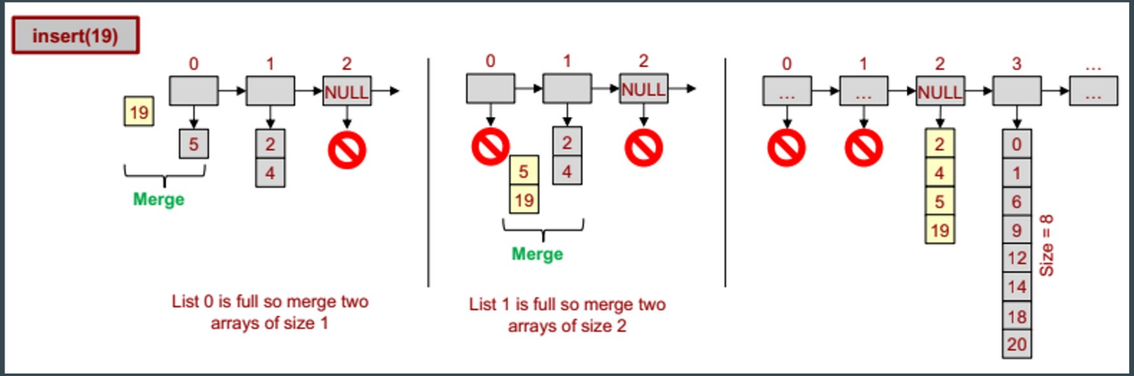
# Merge Trees

- Inserting an element
  - Find the first empty array slot in the linked list
  - That slot will become completely filled and all arrays in previous slots will become empty
    - Starting at array 0, merge the element to insert with the contents in array 0
    - Merge the current array with the next array
    - Continue merging until you reach an empty array



# Merge Trees

- Insertion Runtime
  - Worst Case:
    - All k arrays are full so we merge at each location
    - Merging two sorted arrays of size m/2 to create an array of size m is O(m)
    - We will end up with an array of size n=2^k at position k of the linked list
    - Total cost = 1 + 2 + 4 + 8 + ... + 2^k = O(2^(k+1)) = **O(n)**

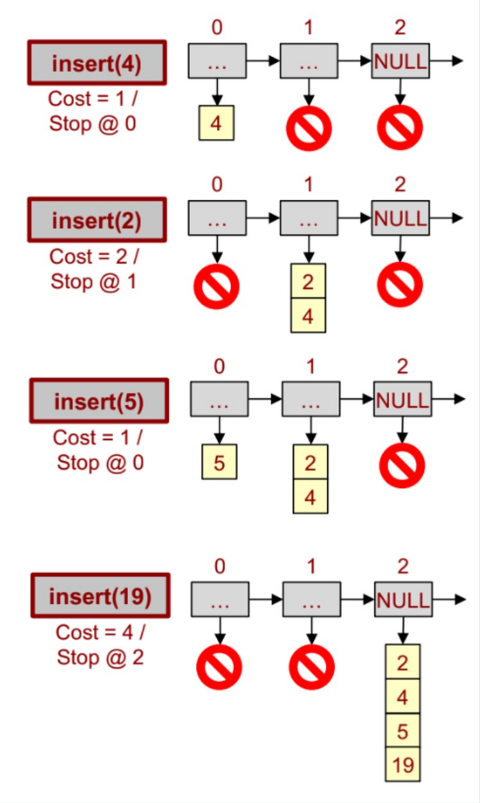


# Merge Trees

- Insertion Runtime
  - Amortized Case:
    - Which array will be empty first and with what probability?
      - Array 0 is empty -> O(2)  (P = 1/2^1 = 0.5)
      - Array 1 is empty -> O(4) (P = 1/2^2 = 0.25)
      - Array 2 is empty -> O(8) (P = 1/2^3)
      - Array k has probability 1/2^(k+1) of doing 2^(k+1) work
    - For k levels, where k = log(n):

$$\sum_{k=0}^{\log(n)} 2^{k+1}2^{-(k+1)} = \sum_{k=0}^{\log(n)} 1 = \log(n)$$

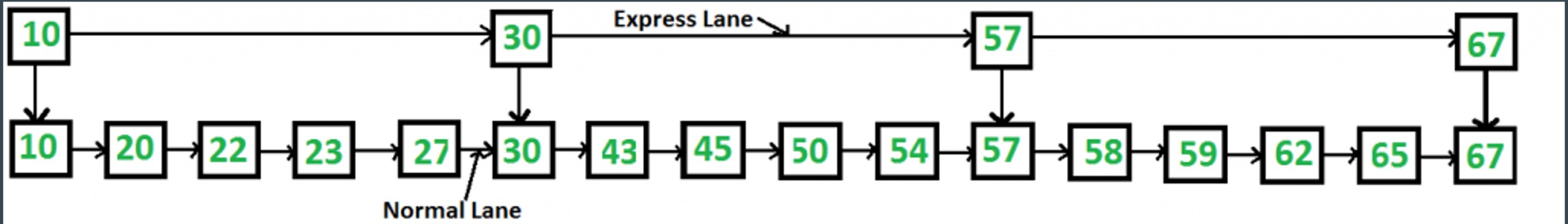
      - Runtime = **log(n)**



# Skip Lists

- Think of it like a layered, more efficient sorted linked list
- The amortized time complexity for search, insertion, and deletion for a skip list is O(log n); compare to linked list amortized time complexity of O(n)

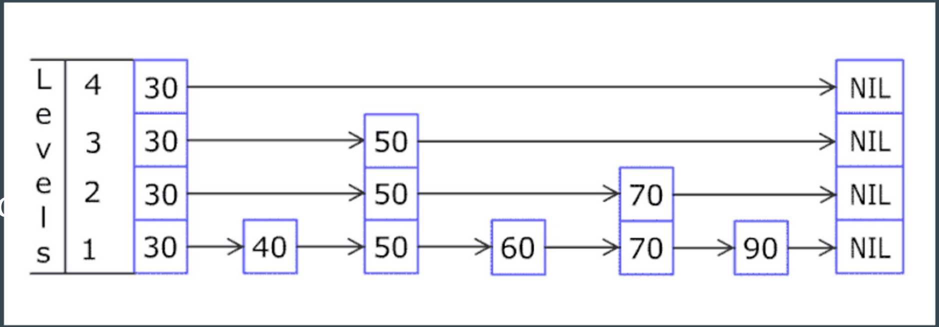
Example with just two layers



# Skip Lists

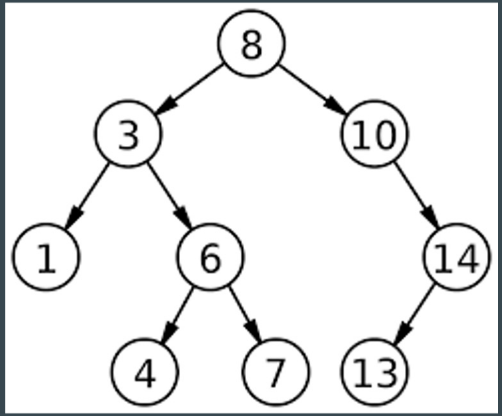
- There can be more than 2 layers; bottom layer is always just regular LL, top is typically just head node
- An element that lives in a lower layer (say, layer *i* ) has probability *p* (commonly 25% or 50%) of being in the layer directly above (layer *i+1* )
- Whether or not element is in a layer is usually determined randomly, ex. Pick threshold probability *p*, choose a random number, and if random number is < *p*, put it into the higher level list

Insertion int



# Binary Search Trees (BSTs)

- Refer to last lab's slides
- Basic principle: **recursive structure of nodes and edges**
  - every node is *greater than* its left subtree
  - every node is *less than* its right subtree
  - Nodes with no children are called "leaves"
  - Node with no parent is "root"

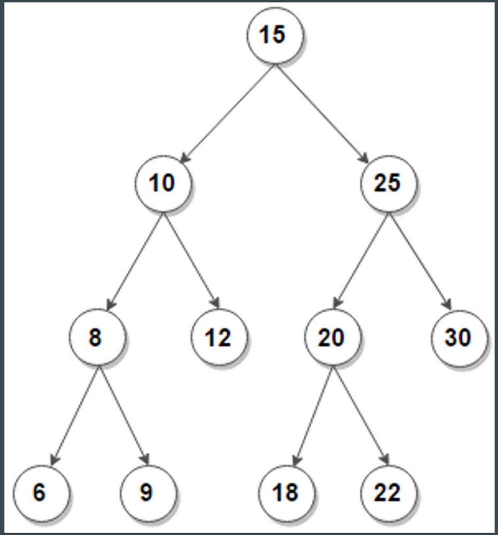


# BST Coding Problem: Find Number of Subtrees within Range

- Given the root of a binary search tree and a range (ex. [1-10]), return the number of subtrees that values are within this range
- The range is inclusive
- A leaf node that is within the range counts as a subtree

Example: range [5 - 21] = 6 subtrees

```
6         9          8              10          12          18
                   /   \           /   \
                  6     9         8     12
                                 / \
                                6   9
```



# BST Coding Problem: Find Number of Subtrees within Range

- Any ideas?
- Hint: should use recursion

- What info do we need to know at each subtree?
- Base case, recursive case?

```
struct Node {
    int val;
    Node* left;
    Node* right;
}
```

```
int numSubtrees(Node* root, int low, int high) { }
```

# BST Coding Problem: Find Number of Subtrees within Range

- **Things we want to know:**
  - Whether or not our children subtrees are within the range
    - Think: if both left and right are within range, then current node is guaranteed to be in range too, forming another subtree in range...
  - How pass this along?
  - Can't rely on int return value to figure out if child node is valid since we don't know how many subtrees there are underneath it...
  - **Solution: create a helper function that returns a bool, keep track of the number of subtrees with an extra parameter** (passed by reference to get for final return)

# BST Coding Problem: Find Number of Subtrees within Range

- Helper Function signature:

```
bool isValidSubtree(Node* root, int low, int high, int& count) { }
```

- Great! Now what?

- Figure out base case + recursive case!

# BST Coding Problem: Find Number of Subtrees within Range

- Base case: node is null (standard stuff)
  - What happens? Return **true**, since if we said false then technically no tree would be in range!
  - We do NOT adjust count though, since a null node isn't actually a node... just empty placeholder!

```
bool isValidSubtree(Node* root, int low, int high, int count) {
    // base case
    if(!root) {
        return true;
    }

    // now what...?

}
```

# BST Coding Problem: Find Number of Subtrees within Range

- Recursive case: decide whether or not current tree is valid; if yes, return true and bump count up, if no, return false
  - **Need to know if left and right subtrees are valid first**
    - Post-order traversal!

```
bool isValidSubtree(Node* root, int low, int high, int count) {
    // base case
    if(!root) {
        return true;
    }

    // figure out if left and right are valid
    bool left = isValidSubtree(root->left, low, high, count);
    bool right = isValidSubtree(root->right, low,high, count);

    // now use this info...
}
```

# BST Coding Problem: Find Number of Subtrees within Range

- Fitting the last parts together...

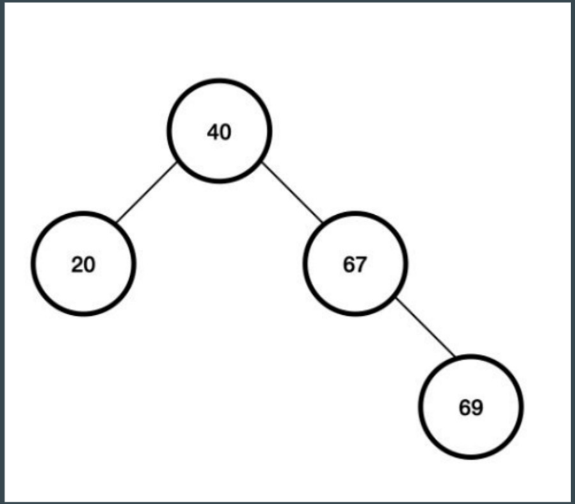
```cpp
bool isValidSubtree(Node* root, int low, int high, int& count) {
    // base case
    if(!root) {
        return true;
    }

    // figure out if left and right are valid
    bool left = isValidSubtree(root->left, low, high, count);
    bool right = isValidSubtree(root->right, low,high, count);

    // if current tree valid, increase count and return true
    // recursive trust fall
    if(left && right && root->val >= low & root->val <= high) {
        count++;
        return true;
    }

    // if not valid, will hit here and return false
    return false;
}
```

# BST Coding Problem: Find Number of Subtrees within Range

- Final usage within our other function:

```
int numSubtrees(Node* root, int low, int high) {
    int count = 0;
    isValidSubtree(root, low, high, count);
    // count modified by isValidSubtree because pass by ref!
    return count;
}
```

# AVL Trees



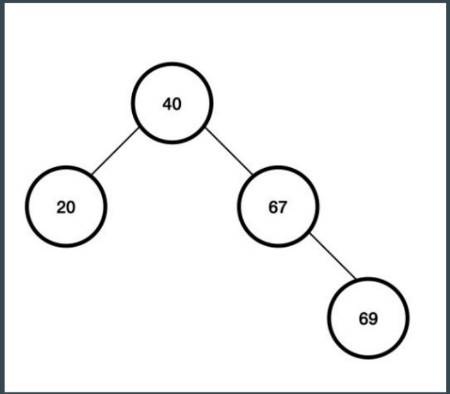

- Also refer to last lab's slides!
- Self-balancing binary search tree

- What is the height of the tree?

- Provide a value that, when inserted, would cause a zig-zig (single) rotation

- Provide a value that, when inserted, would cause a zig-zag (double) rotation

# AVL Trees




- What is the height of the tree?
- **A: 3**

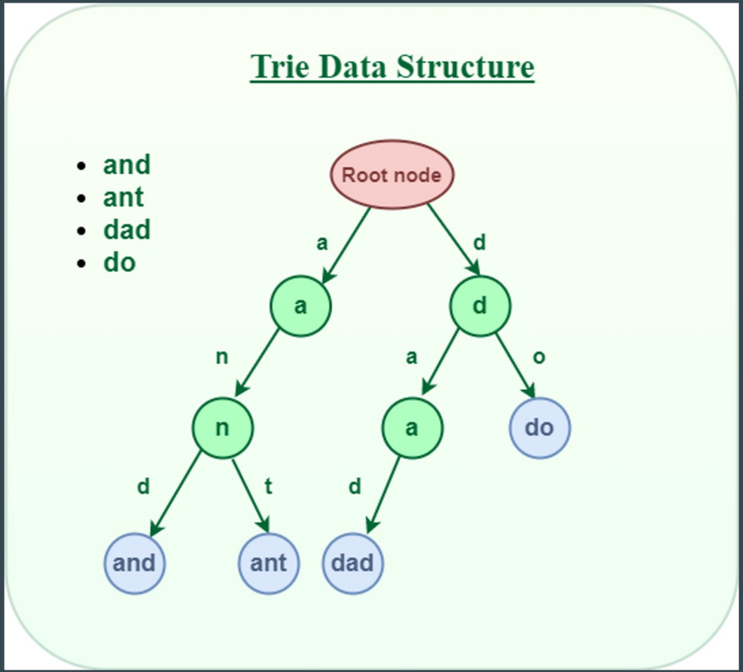
- Provide a value that, when inserted, would cause a zig-zig (single) rotation
- **A: anything greater than 69**

- Provide a value that, when inserted, would cause a zig-zag (double) rotation
- **A: 68**

# Tries

- A.K.A. a prefix tree, most commonly used for strings to determine prefixes!

- Flexible in terms of alphabet you use; ex. If use English alphabet, a node could have up to 27 children! (26+1 because of end character, ex. a ‘$’)



# Trie Coding Problem: Return Longest Prefix

- Given a string, return the longest prefix of the string that exists in the trie; if the word has no prefix, then return an empty string; if the string is in the trie, then you will just return the string! A string in the trie is a valid word if it has a '$' in the children array

```
You, 4 seconds ago | 1 author (You)
struct Node {
    char val;
    Node* children[27];
    // returns proper index in children array corresponding to c
    int getIndex(char c);
}
```

```
string findPrefix(Node* root, string word) { }
```

# Trie Coding Problem: Return Longest Prefix

- Take a couple minutes to think about how you would implement findPrefix

```
string findPrefix(Node* root, string word) { }
```

# Trie Coding Problem: Return Longest Prefix

- Disclaimer, there are many ways to do it, this is just one way!

```
string findPrefix(Node* root, string word) {
    // step 1: setup
    string prefix; // to store the result to be returned
    string workingStr; // to store string we're traversing through

    // step 2: loop through the word using root, prefix, and workingStr
    for(int i = 0; i < word.length(); ++i) {
        // what goes in here?
    }

    // step 3: return!
    return prefix;
}
```

# Trie Coding Problem: Return Longest Prefix

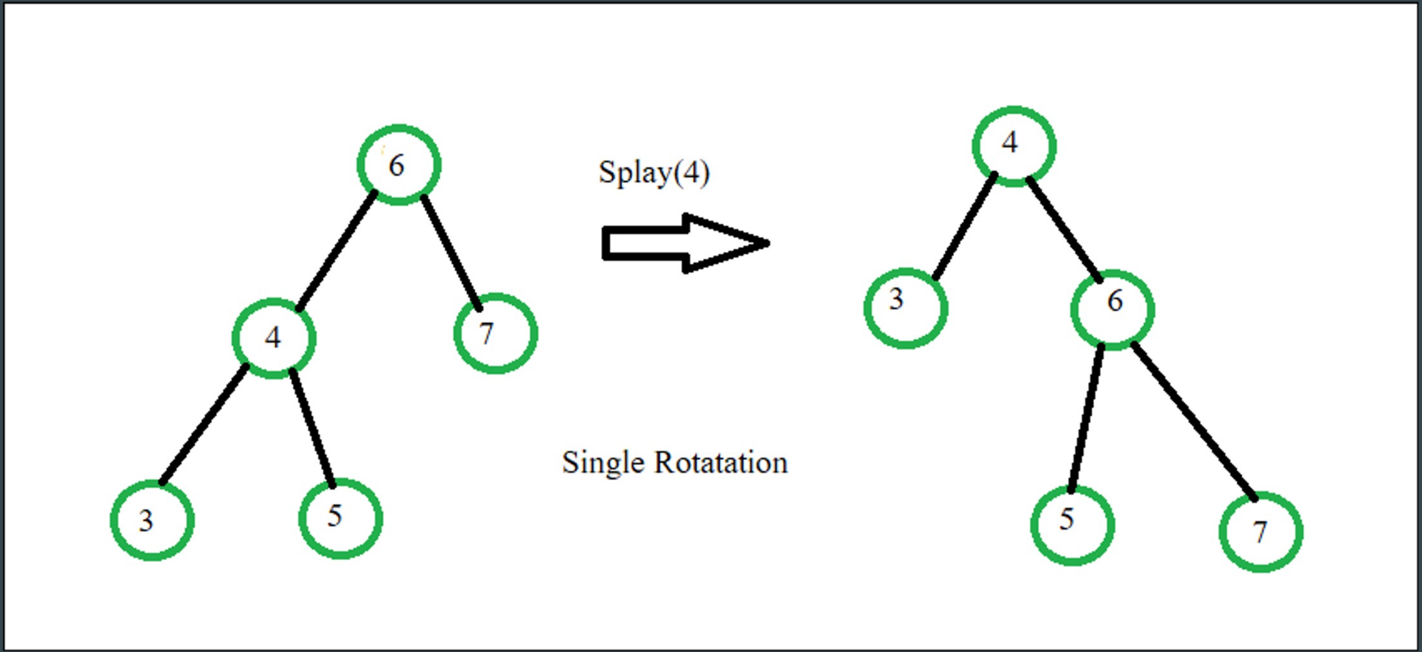
```cpp
string findPrefix(Node* root, string word) {
    // step 1: setup
    string prefix; // to store the result to be returned
    string workingStr; // to store string we're traversing through

    // step 2: loop through the word using root, prefix, and workingStr
    for(int i = 0; i < word.length(); ++i) {
        // setup
        char c = word[i];
        int cIndex = root->getIndex(c);
        root = root->children[cIndex];

        // the node corresponding to c does not exist;
        // break and return current prefix
        if(!root) {
            break;
        }

        // otherwise, update our working string to include this char
        workingStr += c;
        // if c has a '$' child, then we can update our prefix!
        if(root->children[root->getIndex('$')]) {
            prefix += workingStr;
            workingStr = "";
        }

        // another way to do lines 100-106 is to just take the substring
        // of word up to the current index and set equal to prefix
        // therefore no need for workingStr, but more copying necessary
    }
```

# Splay Trees

- Like AVL trees, also self-adjusting, but not height balanced
- Goal is to have most-recently accessed nodes near the top of the tree, so less traversal
- Search, insert, and delete all have O(log n) amortized runtimes
- Has rotation mechanisms very similar to AVL trees, just a couple extra

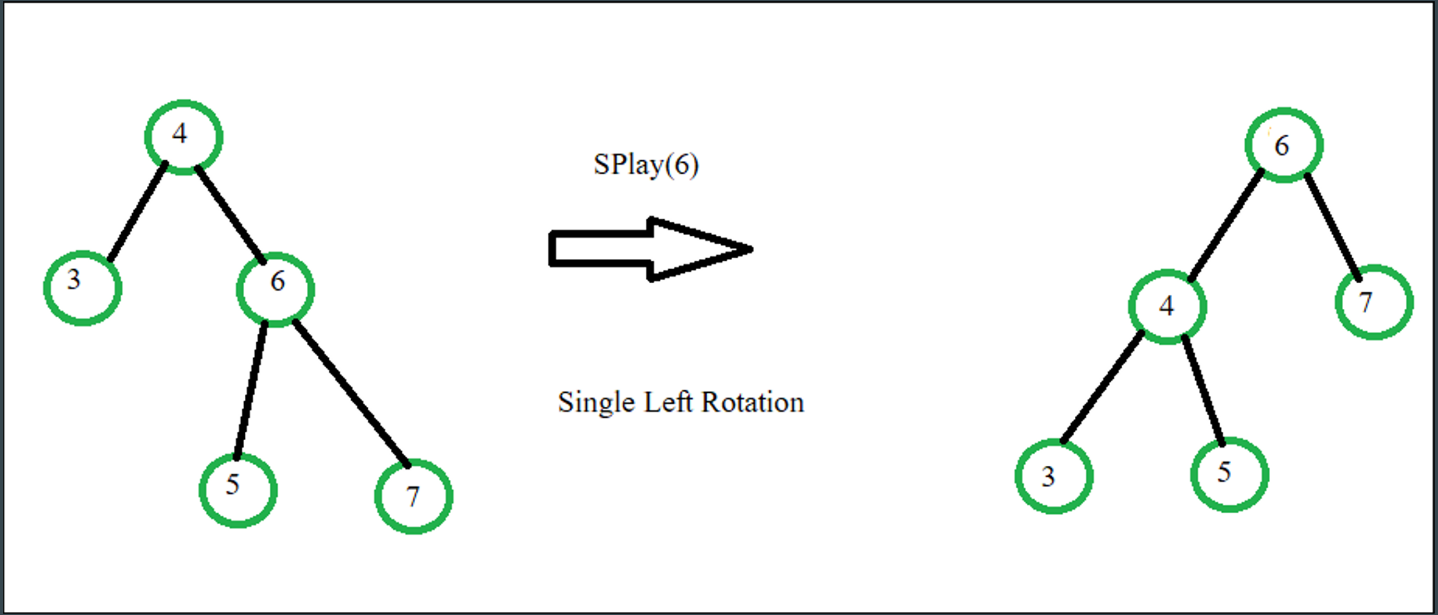
# Splay Trees - Single Right Rotation

- Same as AVL right rotation



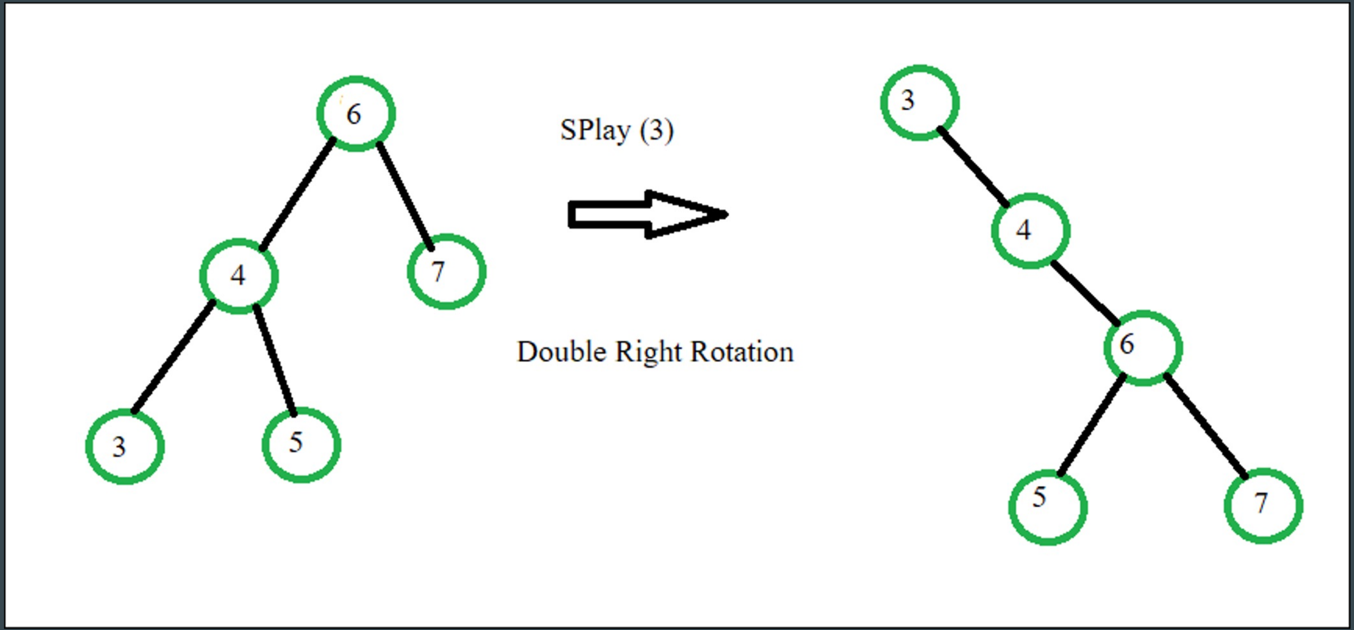
# Splay Trees - Single Left Rotation

- Same as AVL left rotation



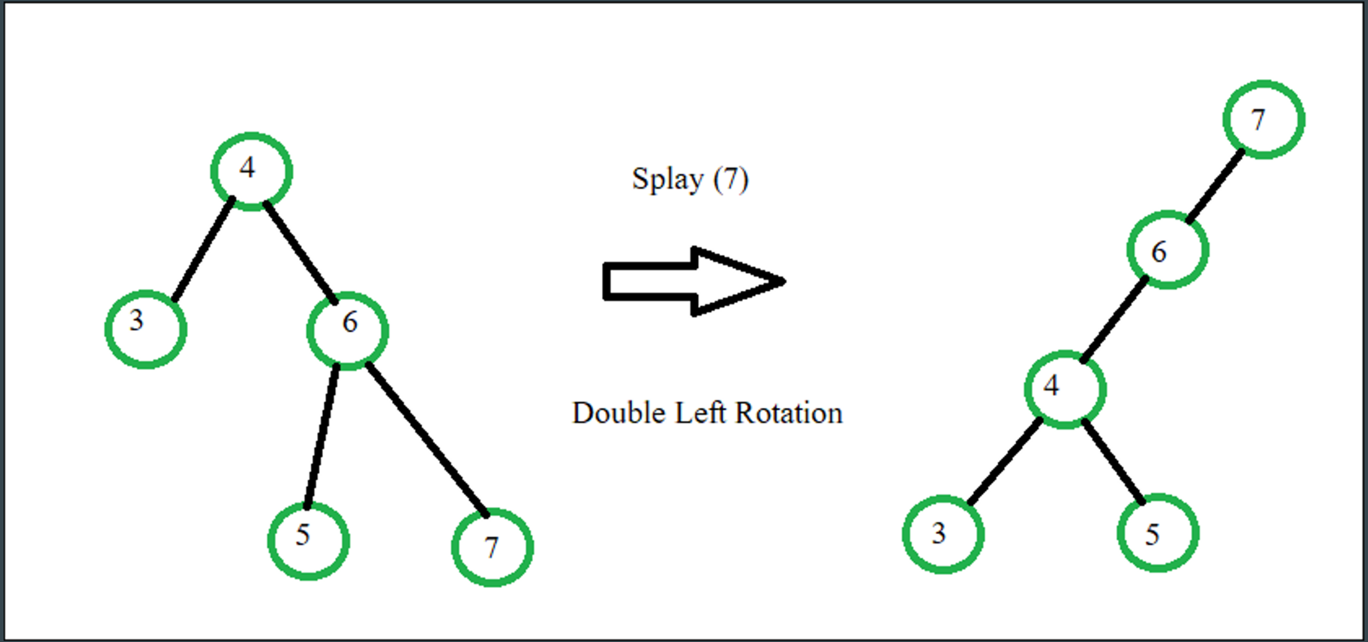
# Splay Trees - Double Right Rotation

- Two right rotations in a row to get a node two levels down up to root



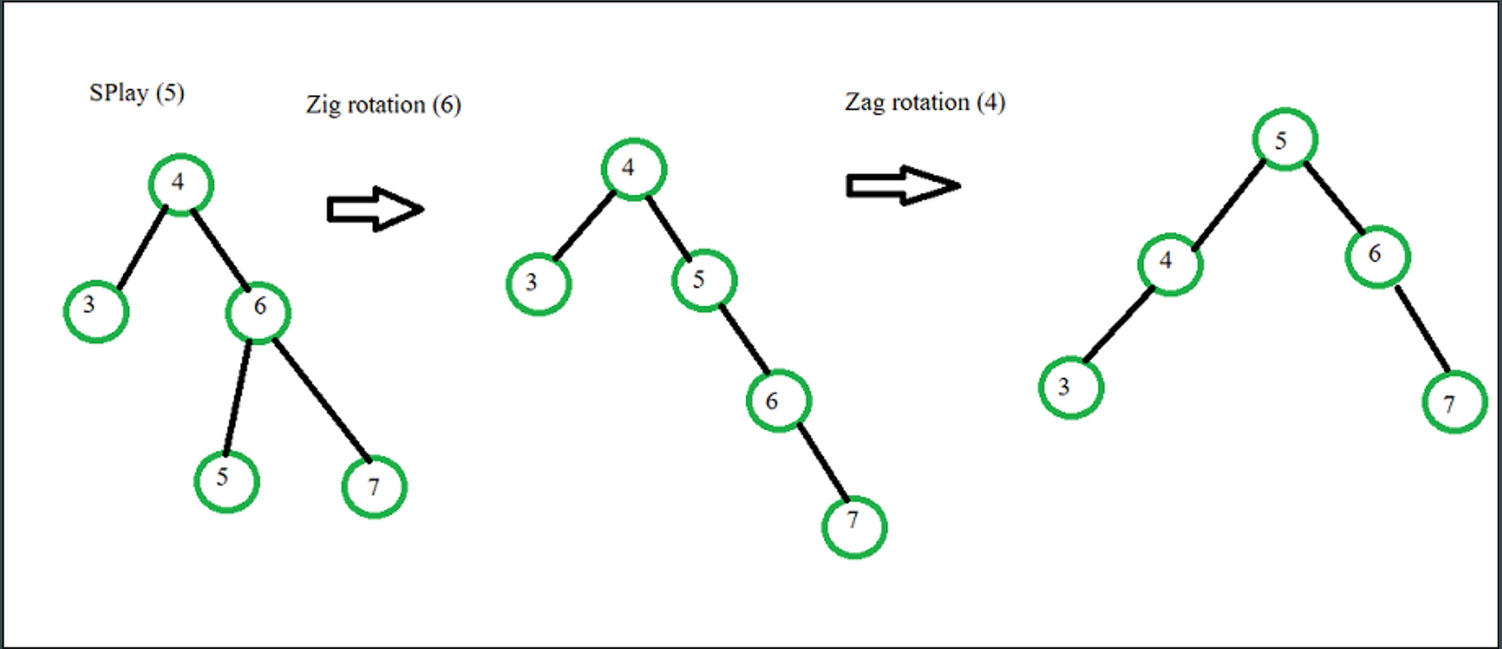
# Splay Trees - Double Left Rotation

- Two left rotations in a row to get a node two levels down up to root



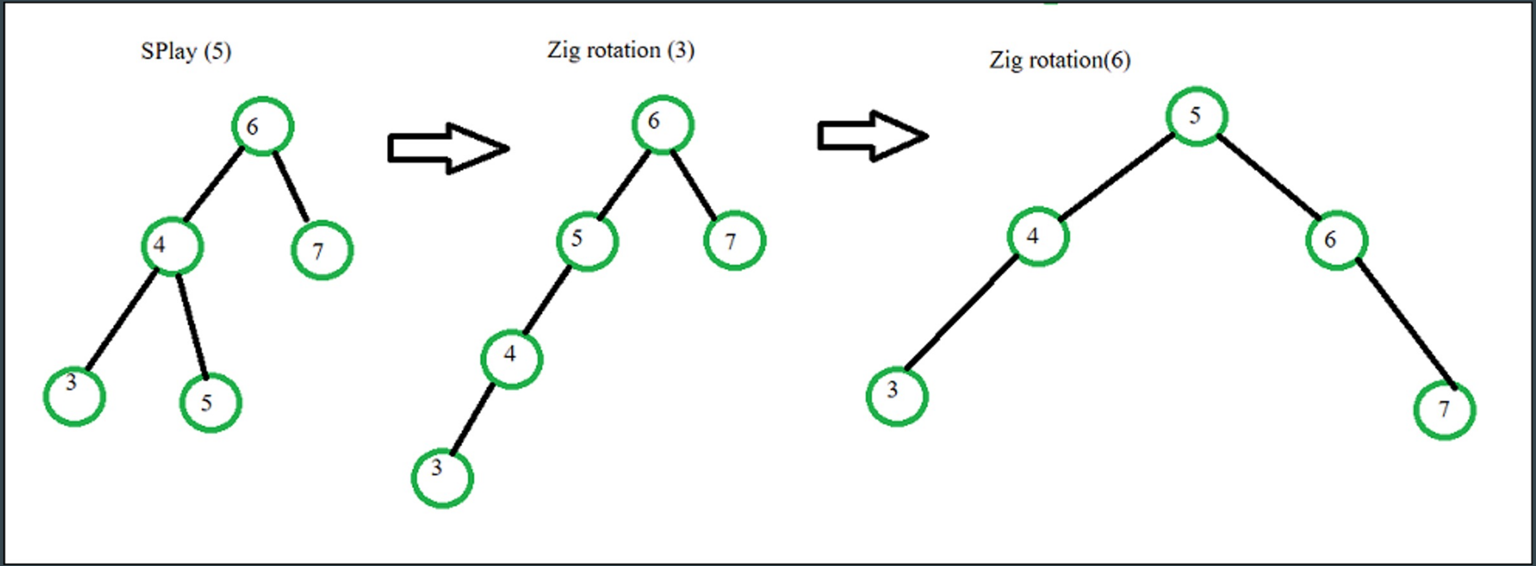
# Splay Trees - Right Left Rotation

- A right rotation followed by a left rotation



# Splay Trees - Left Right Rotation

- A left rotation followed by a right rotation



# Splay Trees Usage

Q: When would you want to use a splay tree?

Q: When would you NOT want to use a splay tree?

# Splay Trees Usage

Q: When would you want to use a splay tree?

**A: When key locality matters, i.e. you're more likely to access a recently used key rather than a random/older key.** An example of this is a network router with sending packets (likely to send packets received closely together to same connection)

Q: When would you NOT want to use a splay tree?

**A: When you *need* to guarantee worst-case performance such as a security system.** Also, splay trees aren't great if the use case doesn't care about key locality.