Midterm 2 Review

CSCI 104

Heaps

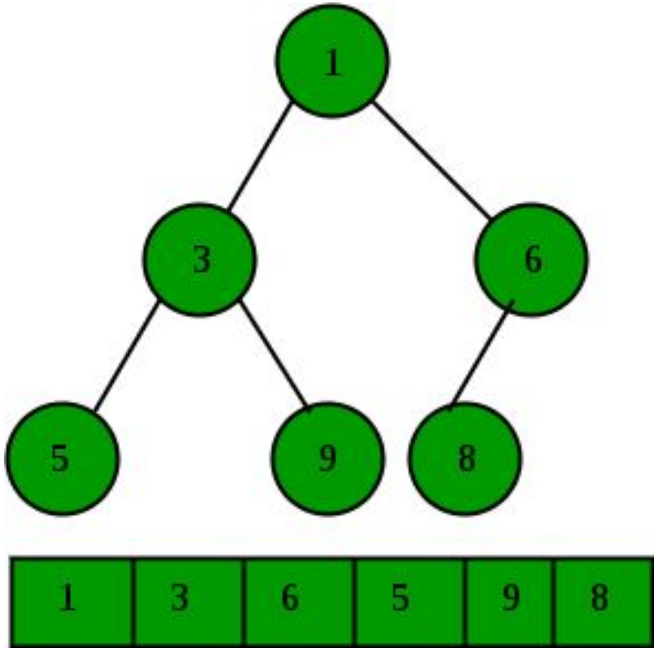
Review: Store a heap in an array

Array starting at index 0, given location i:

Parent Location: (i - 1) / 2

Left Child: 2i + 1

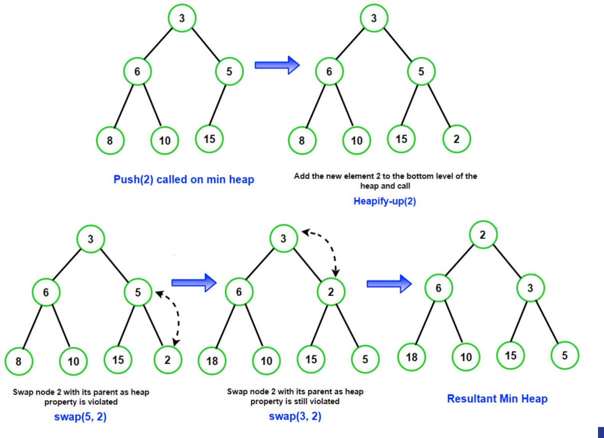
Right Child: 2i + 2



Pushing + Popping a Heap

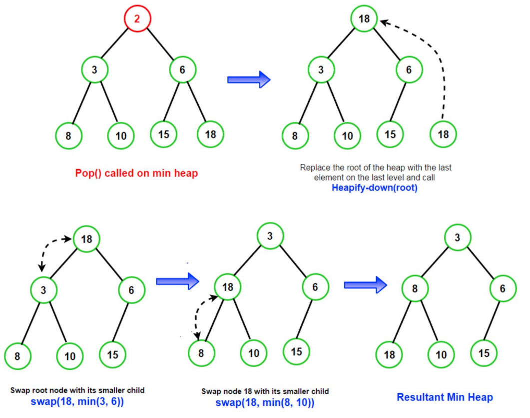
Push:

1. Insert to next index (bottom of tree)
2. Swap with parent until it’s not “better than” its parent, or is now the root



Pop:

1. Swap 0th element (thing to be popped) with last element, delete
2. Swap root element down til in correct spot

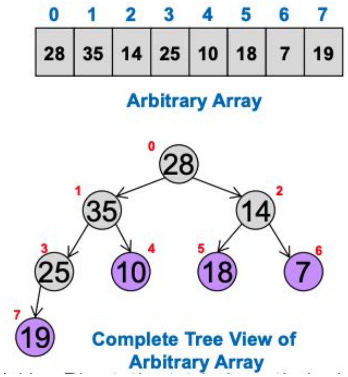


Heap Sort

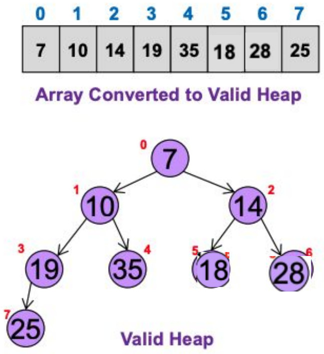
- **Steps:**
  - Convert array into valid heap
    - Min-heap for descending order
    - Max-heap for ascending order
  - Call top() and pop() *n* times to get data in sorted order

- Runtime Analysis
  - Converting array into heap (make-heap): **O(n)**
  - Top() *n* times: O(1) * n -> **O(n)**
  - Pop() *n* times: O(log n) * n -> **O(n log n)**
  - Total runtime: **O(n log n)**

Heap Sort



heapify(3)
heapify(2)
heapify(1)
heapify(0)



Repeat for each node:
top() and pop()

0	1	2	3	4	5	6	7
35	28	25	19	18	14	10	7

Hashtables

Consider a hash table of size 7 with a loading factor of 0.5, the resize function is 2n + 3, where n is the size of the hash table. *(an insertion may end with the loading factor being >= 0.5; the next insertion would cause the resize).*

When resizing, keys are inserted in the order they appear index-wise in the old hashtable.

Hashtables

The hash function is (3k + 4) % n, using quadratic probing. Insert 3, 11, 10, 6, 8, 23.

Key	HashFunc(key)	Loading Factor Before Insert	Probe Sequence
3	(9 + 4) % 7 = 6	0/7 = 0	6

Hashtables

The hash function is (3k + 4) % n, using quadratic probing. Insert 3, 11, 10, 6, 8, 23.

Key	**HashFunc(key)**	**Loading Factor Before Insert**	**Probe Sequence**
3	(9 + 4) % 7 = 6	0/7 = 0	6
11	(33 + 4) % 7 = 2	1/7 = 0.14	2

Hashtables

The hash function is (3k + 4) % n, using quadratic probing. Insert 3, 11, 10, 6, 8, 23.

Key	**HashFunc(key)**	**Loading Factor Before Insert**	**Probe Sequence**
3	(9 + 4) % 7 = 6	0/7 = 0	6
11	(33 + 4) % 7 = 2	1/7 = 0.14	2
10	(30 + 4) % 7 = 6	2/7 = 0.28	6 → 0

Hashtables

The hash function is (3k + 4) % n, using quadratic probing. Insert 3, 11, 10, 6, 8, 23.

Key	**HashFunc(key)**	**Loading Factor Before Insert**	**Probe Sequence**
3	(9 + 4) % 7 = 6	0/7 = 0	6
11	(33 + 4) % 7 = 2	1/7 = 0.14	2
10	(30 + 4) % 7 = 6	2/7 = 0.28	6 → 0
6	(18 + 4) % 7 = 1	3/7 = 0.42	1

Hashtables

The hash function is (3k + 4) % n, using quadratic probing. Insert 3, 11, 10, 6, 8, 23.

Key	HashFunc(key)	Loading Factor Before Insert	Probe Sequence
3	(9 + 4) % 7 = 6	0/7 = 0	6
11	(33 + 4) % 7 = 2	1/7 = 0.14	2
10	(30 + 4) % 7 = 6	2/7 = 0.28	6 → 0
6	(18 + 4) % 7 = 1	3/7 = 0.42	1
8		**4/7 = 0.57**	

Hashtables

The hash function is (3k + 4) % n, using quadratic probing. Insert 3, 11, 10, 6, 8, 23.

Key	HashFunc(key)	LF	Probe
3	6	0	6
11	2	0.14	2
10	6	0.28	6 → 0
6	1	0.42	1
8		**0.57**	

Move to new table →

New size is 2n + 3 = 17

Key	HashFunc(key)	LF	Probe
10	34 % 17 = 0	0	0
6	22 % 17 = 5	1/17	5
11	37 % 17 = 3	2/17	3
3	13 % 17=13	3/17	13
8	28 % 17 = 11	4/17	11

Hashtables

The hash function is (3k + 4) % n, using quadratic probing. Insert 3, 11, 10, 6, 8, 23.

Key	**HashFunc(key)**	**LF**	**Probe**
10	34 % 17 = 0	0	0
6	22 % 17 = 5	1/17	5
11	37 % 17 = 3	2/17	3
3	13 % 17=13	3/17	13
8	28 % 17 = 11	4/17	11
23	73 % 17 = 5	5/17	5 → 6

Hashtables

Final hashtable:

Hashtable index	Key
0	10
3	11
5	6
6	23
11	8
13	3

Hashtables

More questions to consider:

1. What are the benefits of double hashing over things like linear or quadratic probing?
2. No examples of a double collision came up. If there was a double collision, what index do we go to next?
3. Can you explain the benefits of resizing?
4. Are probes ever guaranteed to go to distinct locations? If yes, what are the conditions for this to happen?

Graphs & Graph Representations

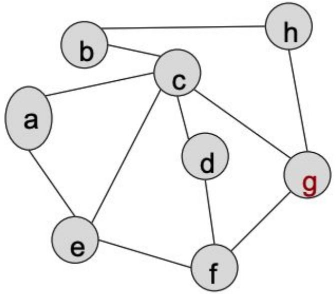
A **graph** is a collection of vertices (or nodes) and edges that connect vertices.

Adjacency List

- List of vertices each having their own list of adjacent vertices

List of Vertices

a	c,e
b	c,h
c	a,b,d,e,g
d	c,f
e	a,c,f
f	d,e,g
g	c,f,h
h	b,g



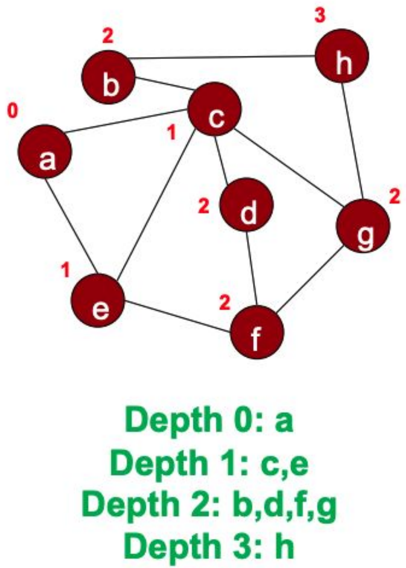
Adjacency Matrix

- Entry at (i,j) = 1 if there is an edge between vertex i and j, 0 otherwise

	a	b	c	d	e	f	g	h
a	0	0	1	0	1	0	0	0
b	0	0	1	0	0	0	0	1
c	1	1	0	1	1	0	1	0
d	0	0	1	0	0	1	0	0
e	1	0	1	0	0	1	0	0
f	0	0	0	1	1	0	1	0
g	0	0	1	0	0	1	0	1
h	0	1	0	0	0	0	1	0

Graph Algorithms

Breadth-First Search (BFS)

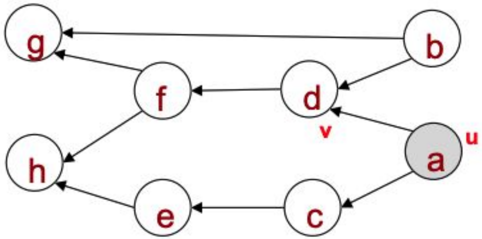


- Explores all nearer neighbors before exploring further away neighbors
- Explores vertices in First-In/First-Out order via a Queue
- "Mark" a vertex as visited on the first time we encounter it
- Can maintain a "predecessor" structure or value per vertex that indicates which prior vertex found this vertex
  - From depth 0, we know that pred(c) and pred(e) = a
- Useful to find the shortest path from a start vertex to any other vertex

Graph Algorithms

Depth-First Search (DFS)

- Explores ALL children before completing a parent
- Explores vertices in Last-In/First-Out order via a Stack or Recursion
- Process:
  - Visit a node
  - Mark as visited (started)
  - For each visited neighbor, visit it and perform DFS on all of their children
  - Only then, mark as finished

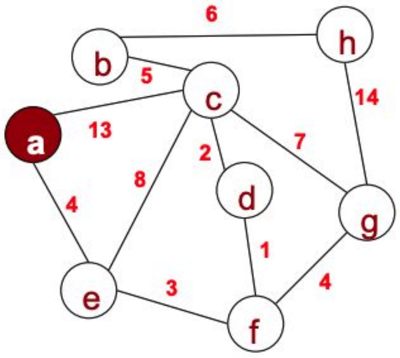


1. Visit(a)
2. Visit(d)
3. Visit(f)
4. Visit(h)
    a. Mark h as finished
5. Visit(g)
6. …

Graph Algorithms

Dijkstra's Algorithm

- Finds the shortest path from source node to all other nodes
- Chooses the closest node (based on smaller distance)
- Uses priority queue to store distance to all vertices from source node
  - All nodes (except for source node) initially start at infinite distance

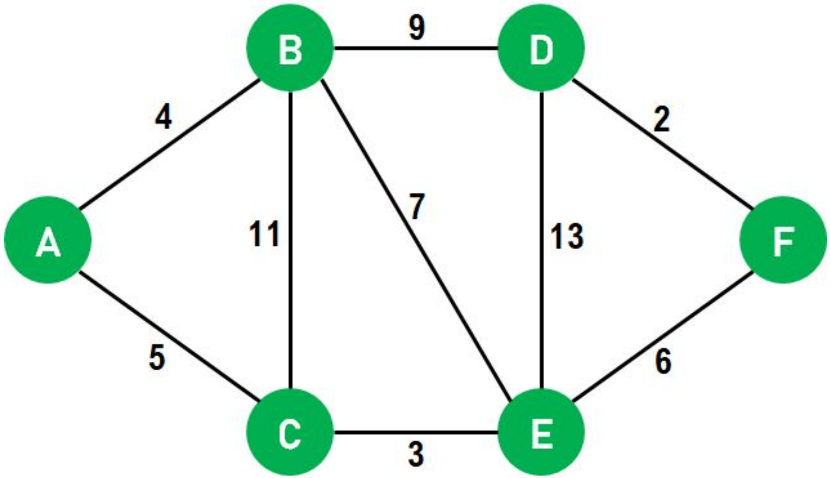


List of Vertices

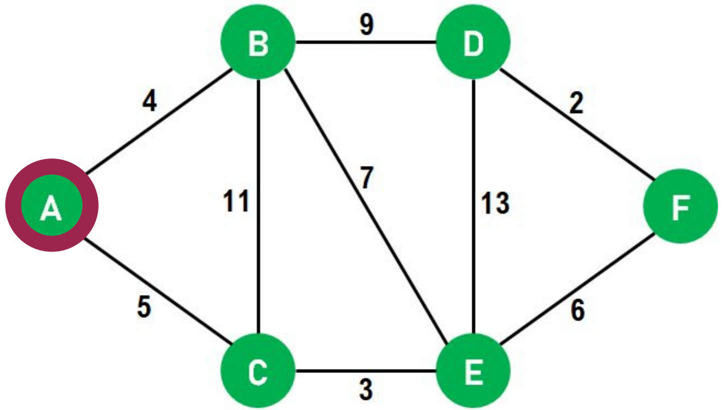
Vert	Dist
a	0
b	inf
c	inf
d	inf
e	inf
f	inf
g	inf
h	inf

Graph Algorithms

Run Dijkstra’s Algorithm on this graph, starting at A, and return the final updated table.



Graph Algorithms



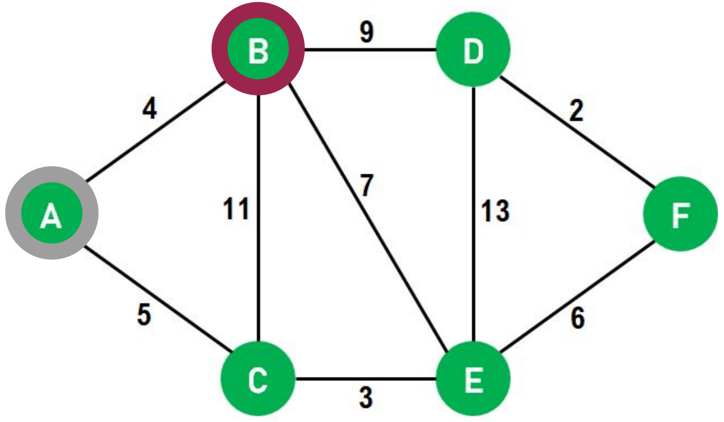
Iteration 1

- Pop from PQ: A
- Explore A's neighbors:
  - A->B: 4
  - A->C: 5
- Updated table (and PQ)
  - A: 0
  - B: 4
  - C: 5
  - D: inf
  - E: inf
  - F: inf

Graph Algorithms

Iteration 2

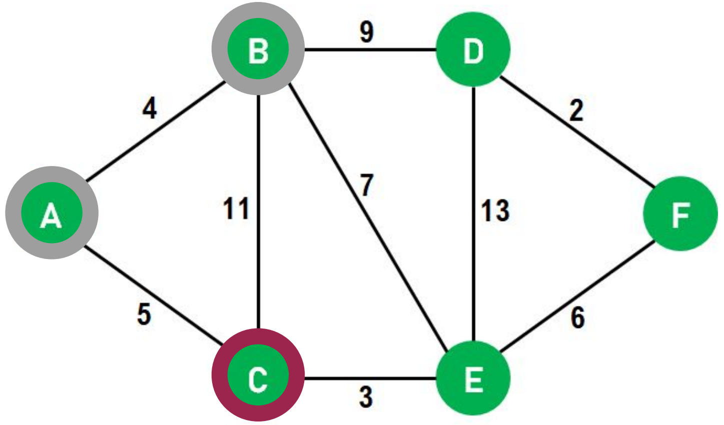
- Pop from PQ: B (4)
- Explore B's neighbors:
    - B->A: 4+0 = 4
    - B->C: 4+11 = 15
    - B->D: 4+9 = 13
    - B->E: 4+7 = 11
- Updated table (and PQ)
    - A: 0
    - B: 4
    - C: 5
    - D: 13
    - E: 11
    - F: inf



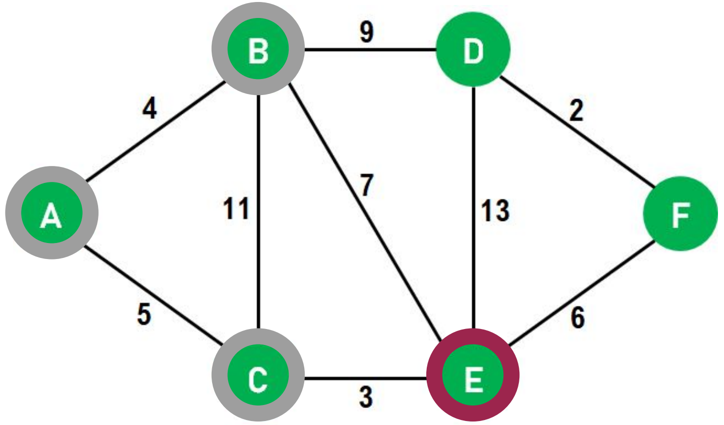
Graph Algorithms

Iteration 3

- Pop from PQ: C (5)
- Explore C's neighbors:
  - C->A: 5+5 = 10
  - C->B: 5+11 = 16
  - C->E: 5+3 = 8
- Updated table (and PQ)
  - A: 0
  - B: 4
  - C: 5
  - D: 13
  - E: 8
  - F: inf



Graph Algorithms



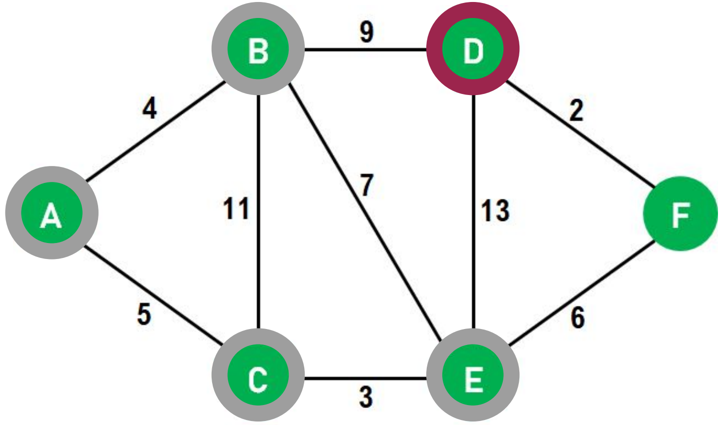
Iteration 4

- Pop from PQ: E (8)
- Explore E's neighbors:
    - E->C: 8+3 = 11
    - E->B: 8+7 = 15
    - E->D: 8+13 = 21
    - E->F: 8+6 = 14
- Updated table (and PQ)
    - A: 0
    - B: 4
    - C: 5
    - D: 13
    - E: 8
    - F: 14

Graph Algorithms

Iteration 5

- Pop from PQ: D (13)
- Explore D's neighbors:
  - D->B: 13+9 = 22
  - D->E: 13+13 = 26
  - D->F: 13+2 = 15
- Updated table (and PQ)
  - A: 0
  - B: 4
  - C: 5
  - D: 13
  - E: 8
  - F: 14

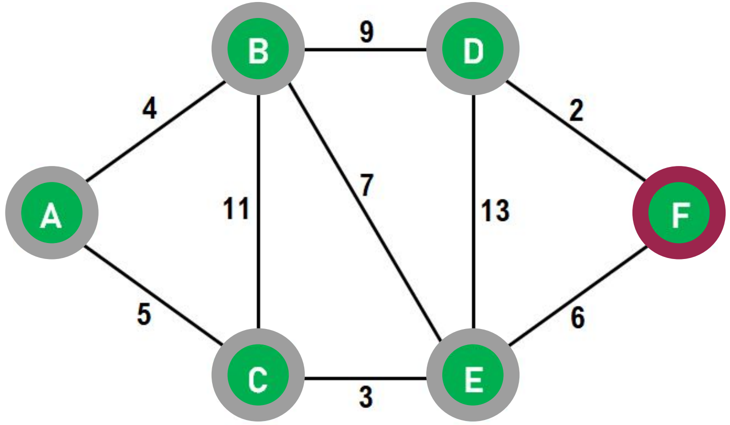


Graph Algorithms

Iteration 6

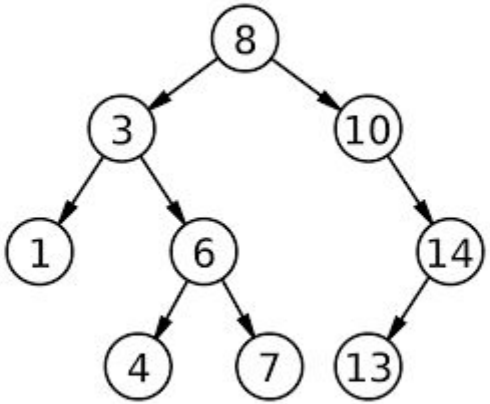
- Pop from PQ: F (14)

- Final table
    - A: 0
    - B: 4
    - C: 5
    - D: 13
    - E: 8
    - F: 14



Binary Search Trees (BSTs)

- Refer to old 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 Traversals: Pre-Order, In-Order, Post-Order

- All traverals operate on EVERY node eventually–just in different orders
  - "Pre" : visit the parent "pre-" (before) visiting left and right sub-trees.
  - "In" : visit the parent "in"-between visiting left and right sub-trees.
  - "Post": visit the parent "post-" (after) visiting left and right sub-trees.

Pre-Order Traversal

```
// Operate on current node
// Recurse left
// Recurse right
// return
```

In-Order Traversal

```
// Recurse left
// Operate on current node
// Recurse right
// return
```

Post-Order Traversal

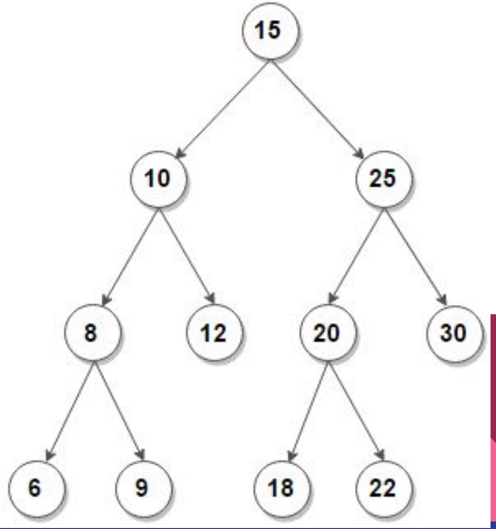
```
// Recurse left
// Recurse right
// Operate on current node
// return
```

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…

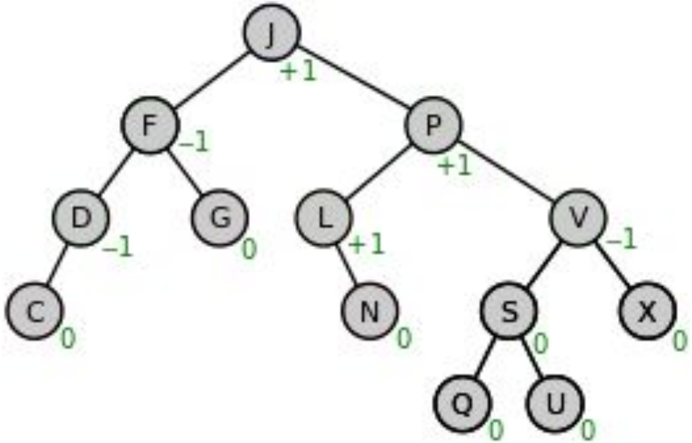
```cpp
bool isValidSubtree(Node* root, int low, int high, int& count) {
    // base case
    if(!root) {
        return true;
    }

    // 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

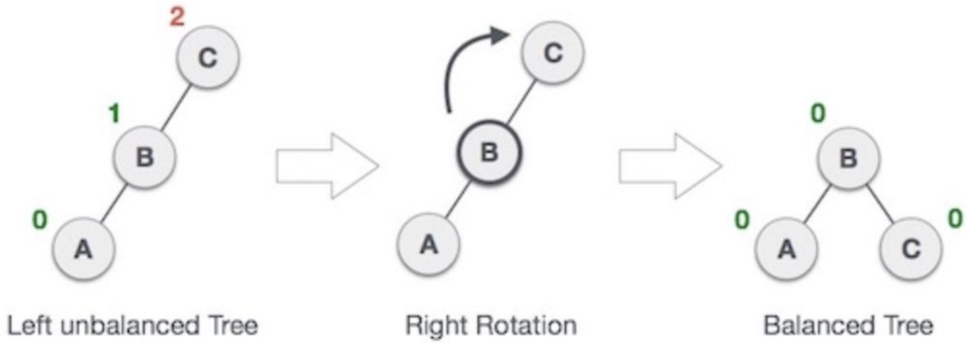


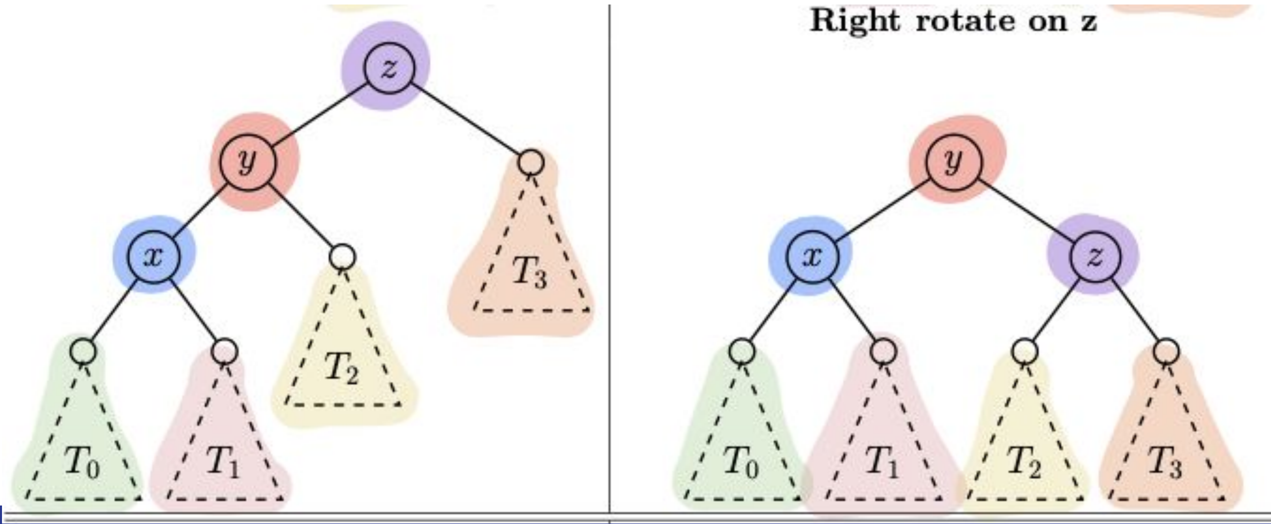
- Self-balancing binary trees
- ALWAYS maintains:
  - BST property
  - Worst case log(n) access time, due to there never being a height difference >= 2

AVL Insert and Remove

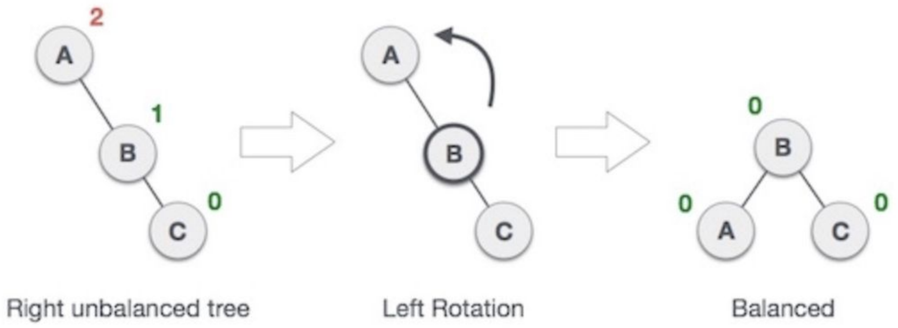
- Insert
  - Insert as you would in a BST
  - Fix the tree if it is unbalanced after inserting the node (ROTATION)
    - Need at most 1 rotation (either a single or double rotation)
- Remove
  - Remove as you would in a BST
  - Keep traversing up the tree and fixing tree if unbalanced (ROTATIONS)
    - You may need multiple rotations to fully fix the tree

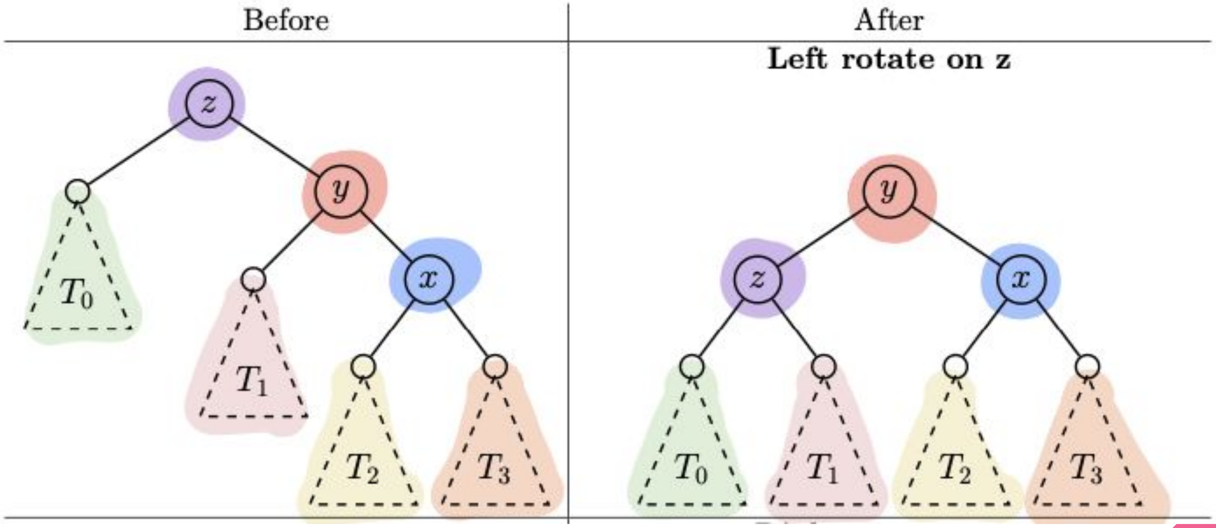
Single Rotations
RIGHT



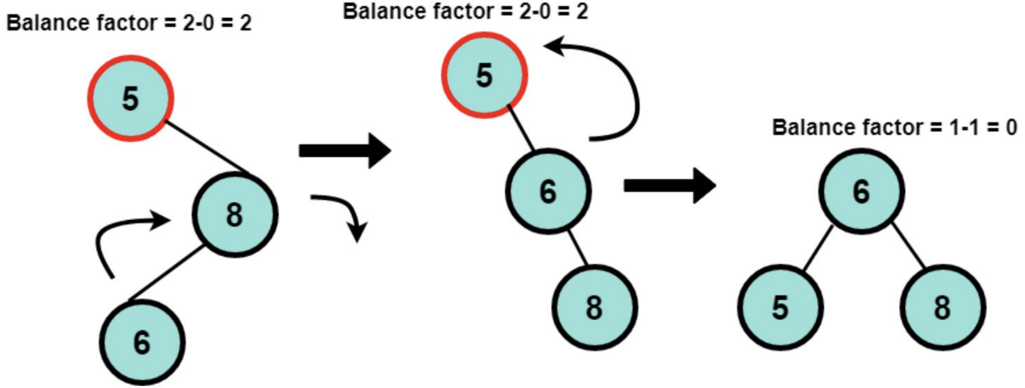


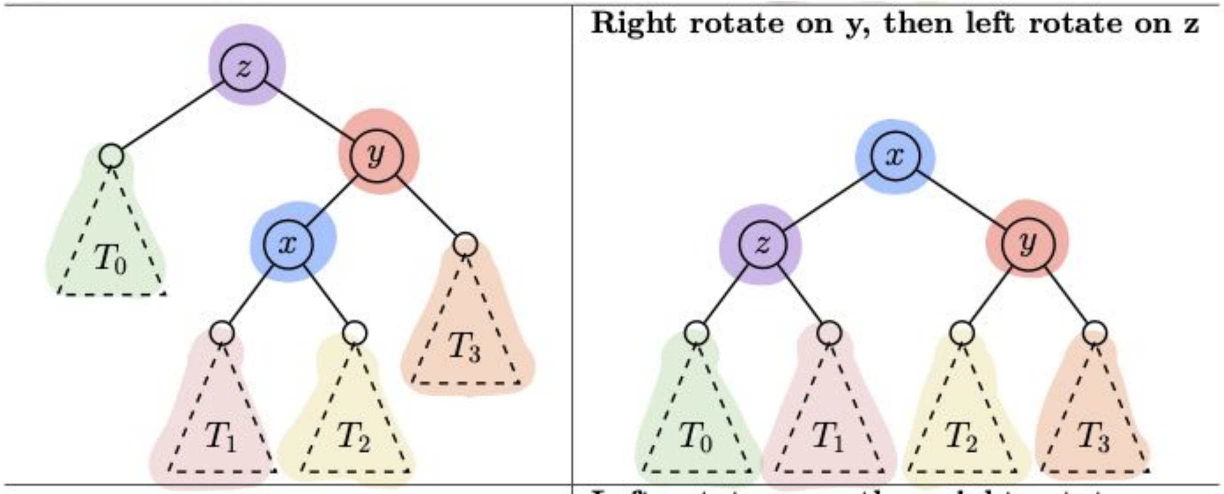
Single Rotations
LEFT



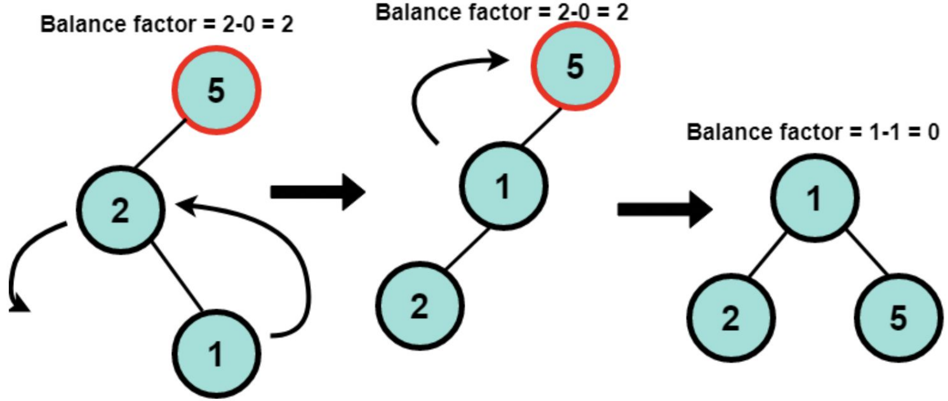


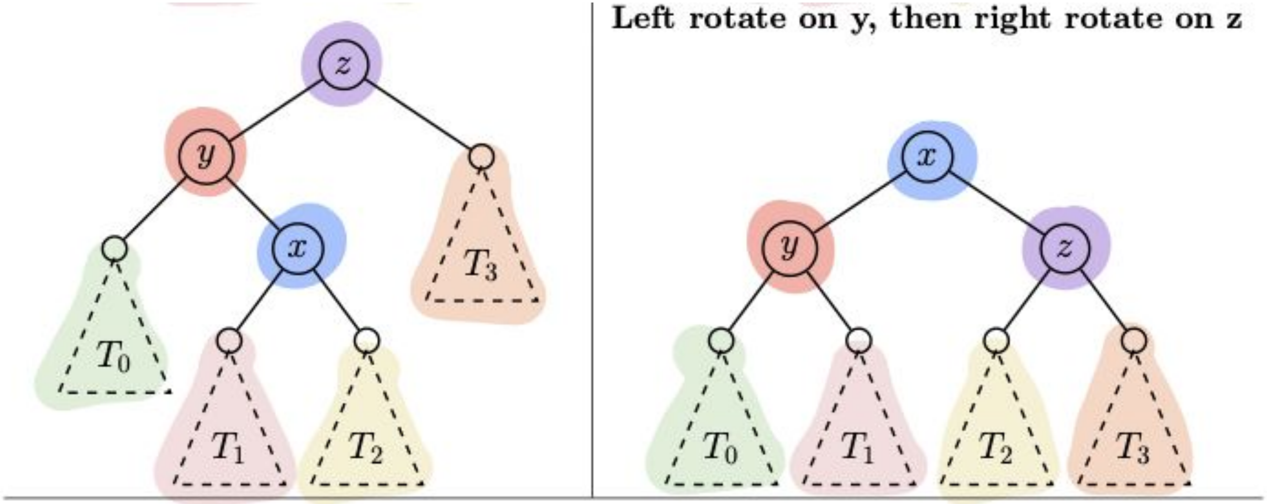
Double Rotations
RIGHT LEFT





Double Rotations LEFT RIGHT

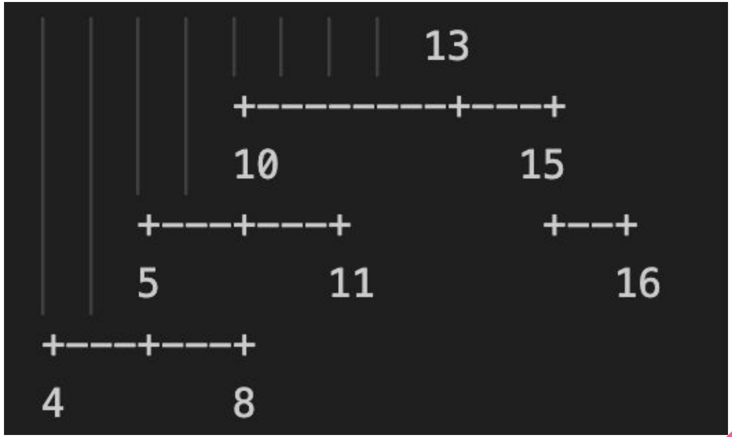




AVL Practice

Given this AVL tree, draw the tree after each of these operations (they build on each other)

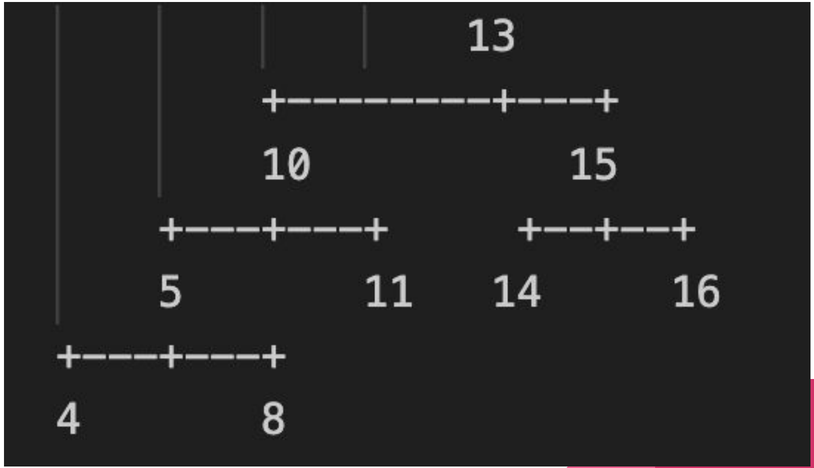
- Insert 14
- Insert 3
- Remove 3
- Remove 4



AVL Practice

Given this AVL tree, draw the tree after each of these operations (they build on each other)

- **Insert 14**
- Insert 3
- Remove 3
- Remove 4



AVL Practice

Given this AVL tree, draw the tree after each of these operations (they build on each other)

- Insert 14
- **Insert 3**
- Remove 3
- Remove 4

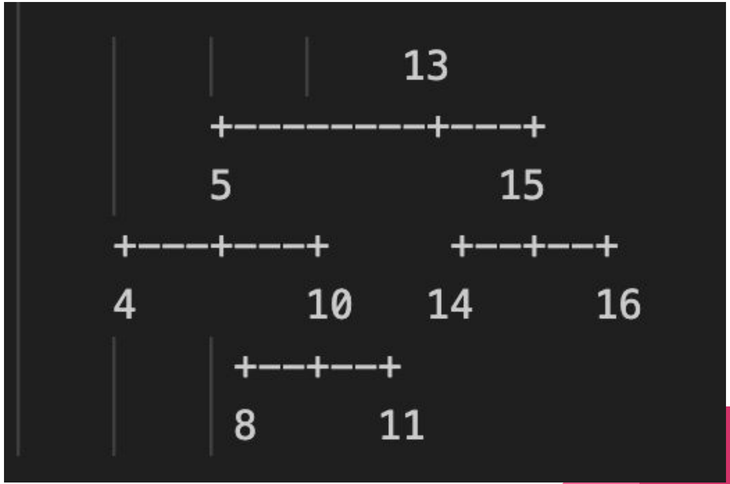
```
                    13
            +---------+----+
            10             15
       +----+----+    +---+---+
       5         11   14      16
  +----+----+
  4         8
+---+
3
```

```
    (Right rotates 10)

                13
          +---------+----+
          5              15
     +----+----+     +---+---+
     4         10   14      16
+----+     +---+---+
3          8       11
```

AVL Practice

Given this AVL tree, draw the tree after each of these operations (they build on each other)

- Insert 14
- Insert 3
- **Remove 3**
- Remove 4



AVL Practice

Given this AVL tree, draw the tree after each of these operations (they build on each other)

- Insert 14
- Insert 3
- Remove 3
- **Remove 4**

```
            13
+---------+----+
5              15
+----+     +---+---+
    10    14       16
 +---+---+
 8       11
```

```
(Left rotate 5)

          13
   +---------+----+
   10             15
+---+----+    +---+---+
5       11   14       16
+---+
   8
```

Splay Trees

- Like AVL trees, also self-adjusting, but not height balanced

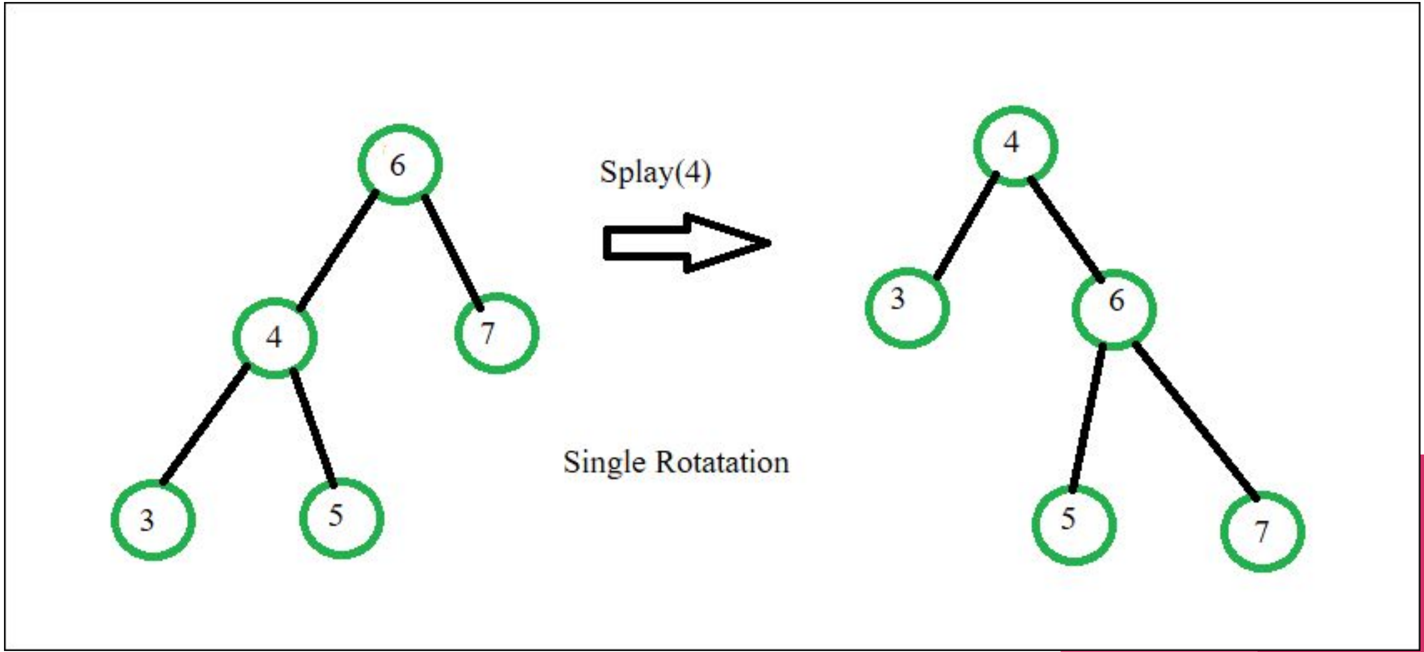
- Goal is to have most-recently accessed nodes near the top of the tree, so less traversal

- Search, insert, and delete all have O(log n) amortized runtimes

- Has rotation mechanisms very similar to AVL trees, just a couple extra

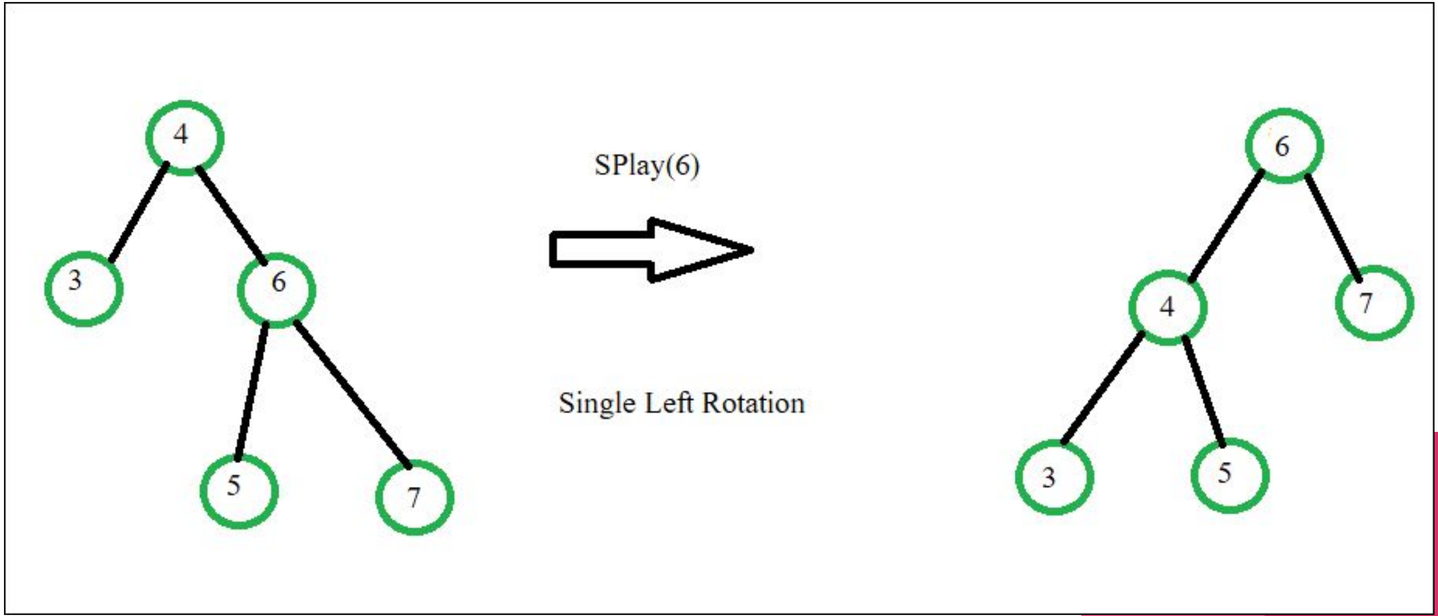
Splay Trees - Single Right Rotation

- Same as AVL right rotation



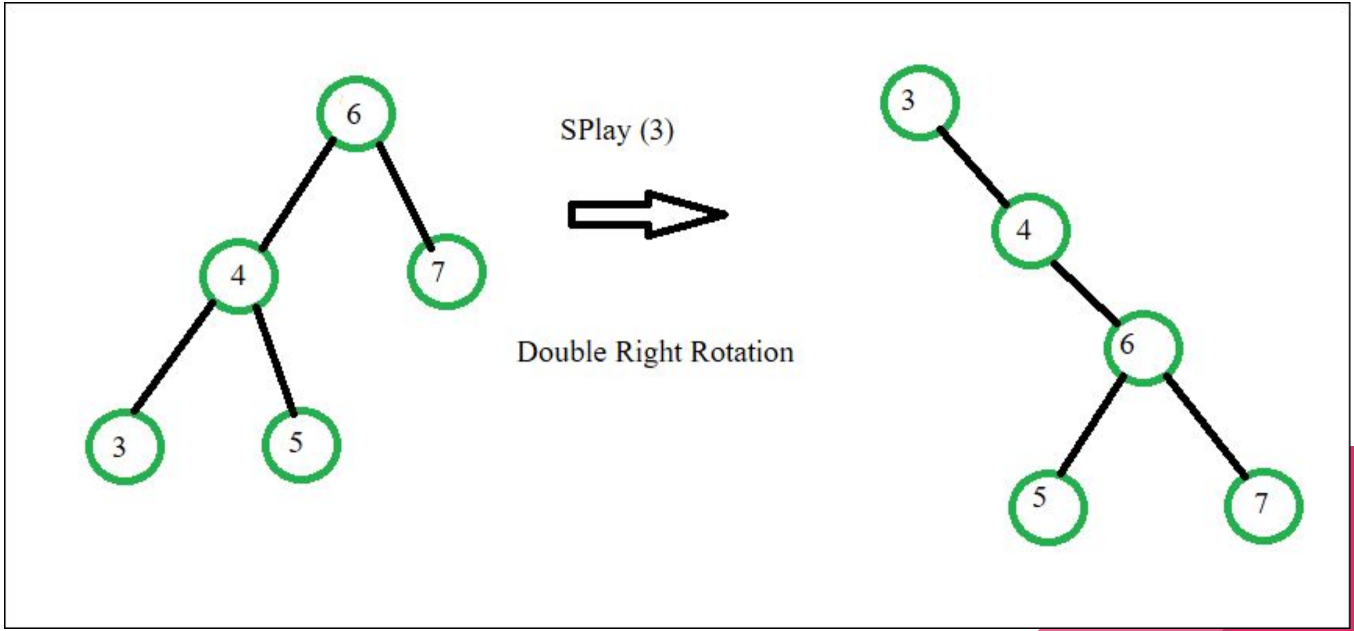
Splay Trees - Single Left Rotation

- Same as AVL left rotation



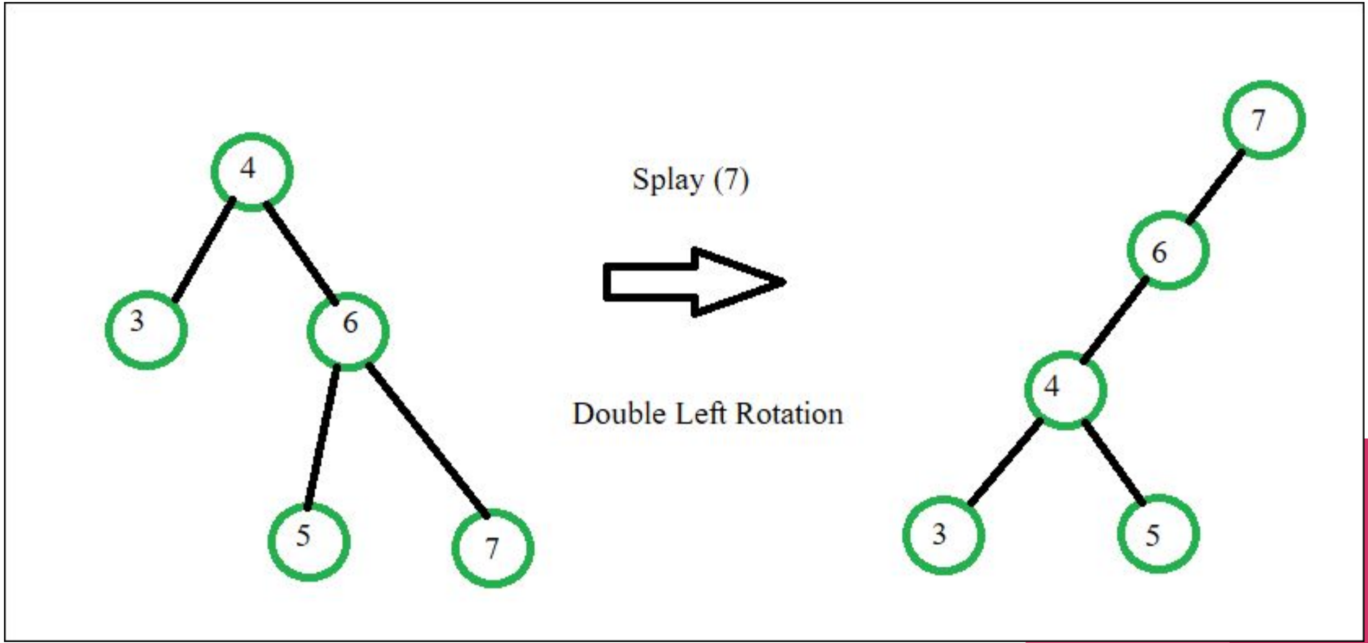
Splay Trees - Double Right Rotation

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



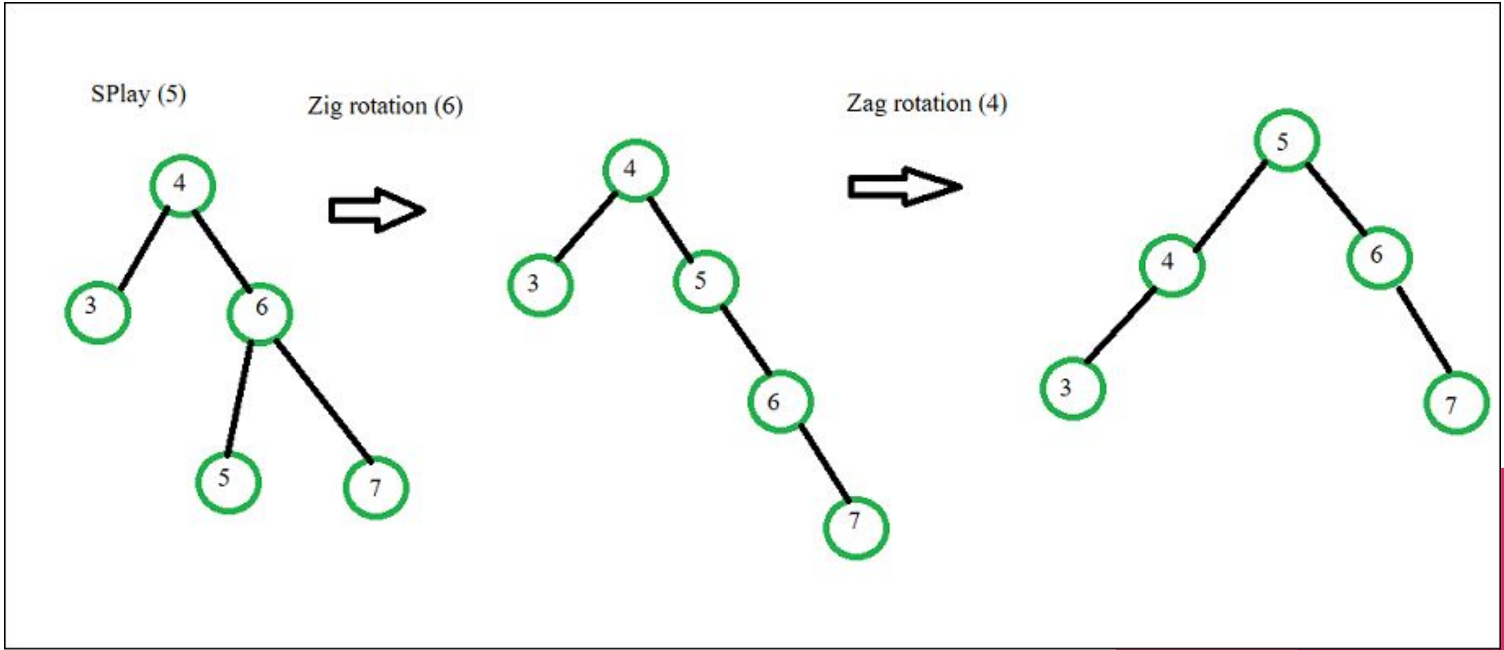
Splay Trees - Double Left Rotation

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



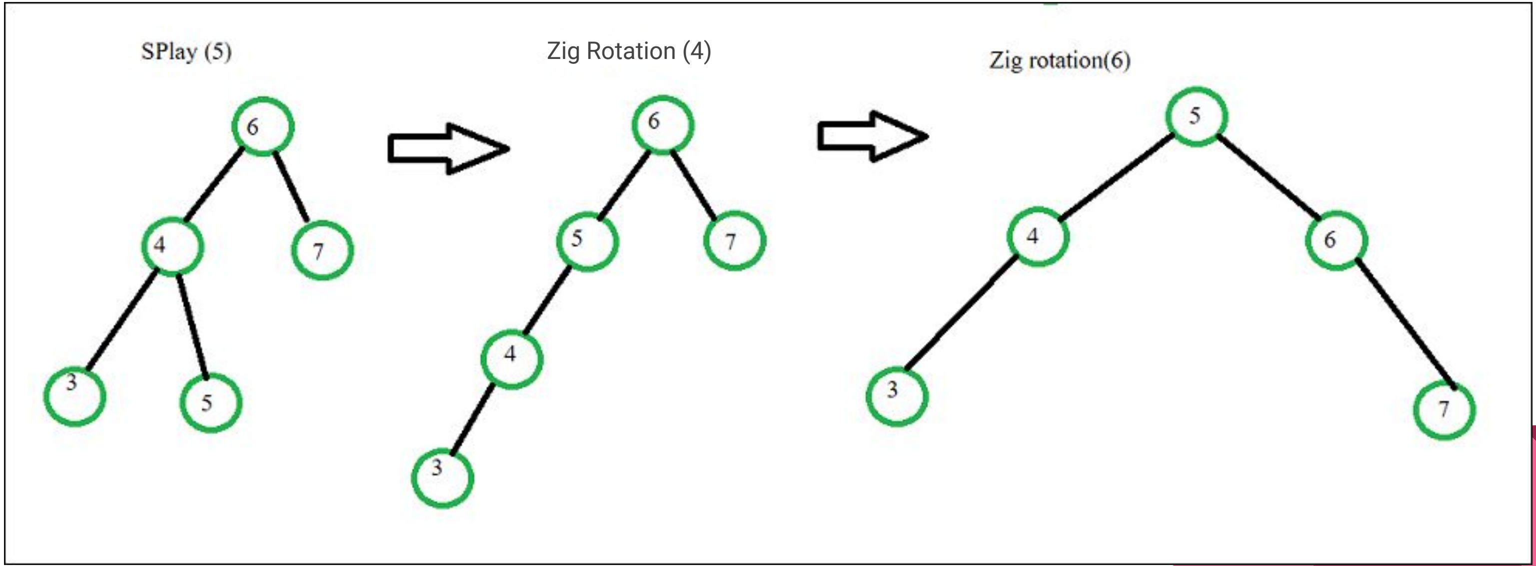
Splay Trees - Right Left Rotation

- 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.

Recursion: Backtracking Refresher

- The general backtracking algorithm is a modification of **depth first search**
- Apply a change to our current state
- if it's valid, then we continue down that path, adding new changes (depth first search!!!)
    - If we find a solution, great! Return and all done
    - If there are no more possible valid options and we haven't hit an end state yet, then we "undo" our last change, return false, and are brought back up to the previous state where we can try a different option

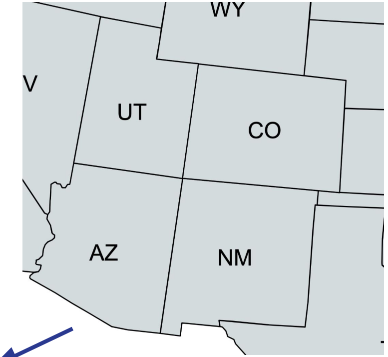
Pseudocode

- Not **ALL** backtracking algos will return a bool
- Sometimes you check state/validity at multiple points in the function
- Etc.

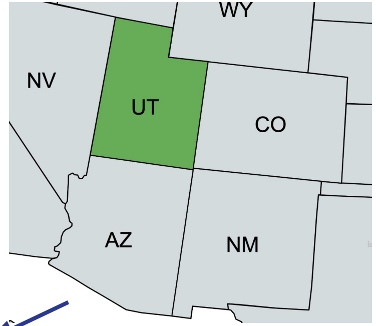
```
void solve():
  recursive_helper(params)

bool is_valid():
  # returns whether state is valid

bool recursive_helper(params):
  if finished state and valid:
    save solution
    return true

  for each next possible state choice:
    apply choice to state
    if choice is valid:
      recursive call with current state
    remove choice (Backtrack)

  return false (no viable solution found)
```

Time Complexity?

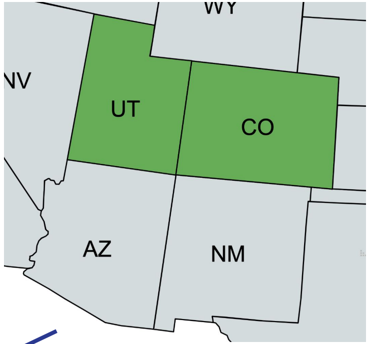
- DFS map coloring



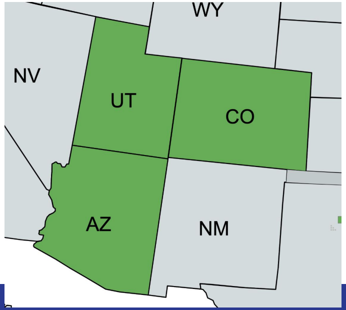
Initial state



adds color; map
not done



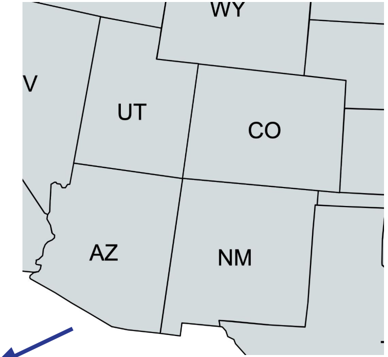
adds color; map
not done



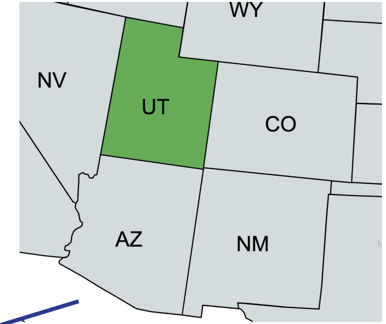
DFS keeps coloring green until all states colored; only
when **done** it realizes it's invalid!

Map Coloring Example

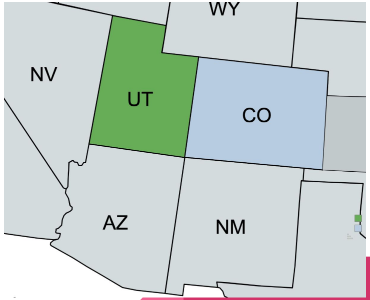
- Backtracking map coloring



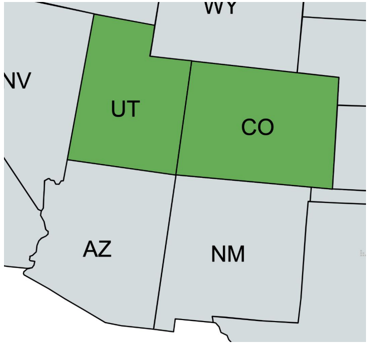
Initial state



adds color; valid, but map not done



adds next color option; valid, but map not done



adds color; **INVALID!!! backtrack**

Backtracking will **immediately stop** when it realizes something is invalid

Recursion: Backtracking & Combinations

Suppose you are given an integer array *nums* of unique elements. **Return all possible subsets** of the array (in other words, the power set). The solution can be in any order, and you must not include duplicate subsets.

- Example:
  - Input: nums = [1,2,3]
  - Output: [[ ], [1], [2], [1,2], [3], [1,3], [2,3], [1,2,3]]

```
// Returns all subsets of nums
vector<vector<int>> subsets(vector<int>& nums) {

}
```

- How many subsets are possible?
  - Each element can be included or not included in the subset
  - For an array of n elements, this would be **2^n total subsets**

Recursion: Backtracking & Combinations

We need to explore all possible combinations of the array's elements -> backtracking!

- We will start building a subset that is initially empty
- We will iterate through the array and add the current number to our subset
  - Recursively build the subset without the letters we have already used (adjust the range of our loop)
  - Backtrack by removing the number we added and proceed to the next iteration of the loop
- In each recursive call, we will have a new subset that will be a part of our solution

Recursion: Backtracking & Combinations

Solution:

```
// helper function for subsets
void dfs(vector<int>& nums, int start, vector<int>& curr, vector<vector<int>>& result) {
    result.push_back(curr);
    for (int i = start; i < nums.size(); i++) {
        curr.push_back(nums[i]);
        dfs(nums, i + 1, curr, result);
        curr.pop_back();
    }
}

// Returns all subsets of nums
vector<vector<int>> subsets(vector<int>& nums) {
    vector<int> curr;
    vector<vector<int>> result;
    dfs(nums, 0, curr, result);
    return result;
}
```