Data Structures Final Review (Workshop)

Topics Covered:

- Binary Search Trees
- AVL Trees
- Graphs
- B-Trees

Binary Search Trees

BST #3

Complete the function `int getSize(node *root)` which returns the size (number of nodes) in a BST.

```
struct node{
  int value;
  node *left;
  node *right;
};

int getSize(node *root){


}
```

recursive

```
int getSize(node *root){

if(root == nullptr){

return 0;

}

return 1 + getSize(root->left) + getSize(root->right);

}
```

Iteratively

```cpp
int getSizeIterative(node *root) {
    if (root == nullptr) {
        return 0;
    }

    int size = 0;
    std::stack<node *> s;
    s.push(root);

    while (!s.empty()) {
        node *current = s.top();
        s.pop();
        size++;

        // Push right child first so that it is processed later (LIFO
        if (current->right) {
            s.push(current->right);
        }
        if (current->left) {
            s.push(current->left);
        }
    }

    return size;
}
```

RECURSION

BST #4

Complete the function `node *getNode(node *root, int value)` which returns the node containing `value`. Your function must use recursion (no iteration).

```
struct node{
  int value;
  node *left;
  node *right;
};

node *getNode(node *root, int value){


}
```

```
node *getNode(node *root, int value){
  if(root == nullptr){
    return nullptr;
  }
  else if(value < root->value){
    return getNode(root->left, value);
  }
  else if(value > root->value){
    return getNode(root->right, value);
  }
  else{
    return root;
  }
}
```

```
node *getNode(node *root, int value) {
    if (root == nullptr) {
        return nullptr;
    } else if (root->value == value) {
        return root;
    } else if (value < root->value) {
        return getNode(root->left, value);
    } else {
        return getNode(root->right, value);
    }
}
```

ITER

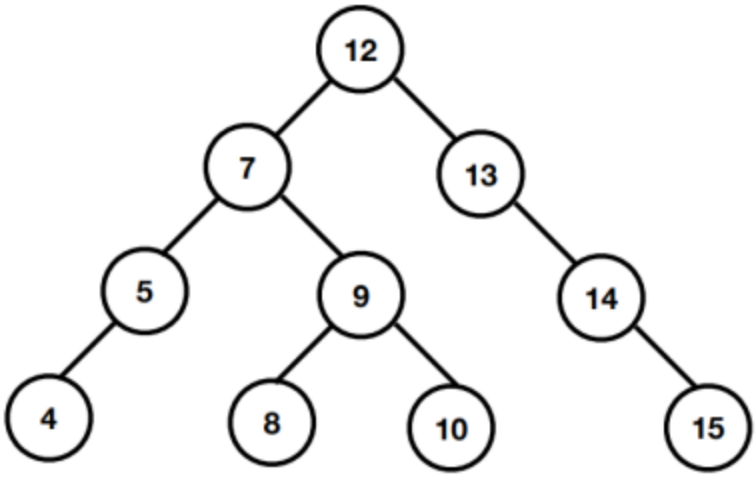
BST #5

Complete the function `node *getNode(node *root, int value)` which returns the node containing `value`. Your function must use iteration (no recursion).

```
struct node{
  int value;
  node *left;
  node *right;
};

node *getNode(node *root, int value){


}
```

```
node *getNode(node *root, int value) {
    while (root != nullptr) {
        if (root->value == value) {
            return root;
        } else if (value < root->value) {
            root = root->left;
        } else if (value > root->value) {
            root = root->right;
        }
    }
    return nullptr; // If the value is not found, return nul
}
```

. Write the function which produced the output below



Output: 4, 5, 8, 10, 9, 7, 15, 14, 13, 12

2. Write a function minNode() that takes the Node* root of a BST and returns the smallest value in the tree.
3. Write a function countNodes() that takes the Node* root of a BST and returns the total number of nodes in a BST.
4. Write a function countFullNodes() that takes the Node* root of a BST and returns the number of nodes that have both left and right children.

```
int minNode(TreeNode* root) {
    while (root->left != nullptr) {
        root = root->left;
    }

    return root->val;
}
```

Count complete nodes

```
int countNodes(TreeNode* root) {
        if(root == nullptr)
        {
            return 0;
        }
        return 1 + countNodes(root->left) + countNodes(root->right);
}
```

```
int countFullNodes(node* root){
    if(root == nullptr)
        return 0;
    else if(root->left != nullptr && root->right != nullptr)
        return 1 + countFullNodes(root->left) + countFullNodes(root->right);
    else
        return countFullNodes(root->left) + countFullNodes(root->right);
}
```

1. Explain why a Binary Search Tree's average time complexities for operations are O(log(n)), what attributes allow this?

2. Implement an **insertToBST(node* root, int value)** function. Can you do this iteratively and **recursively**?

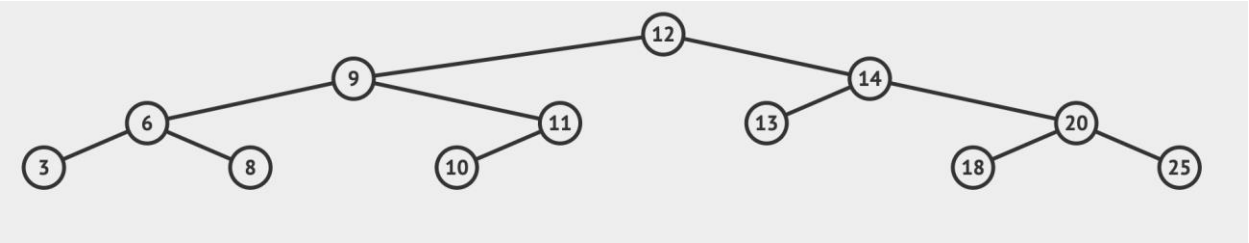
```
treeNode* BST::insertToBST(treeNode* curr, int data)
{
    // If we have reached the end of the tree, insert the new node
    if(curr == nullptr)
    {
        // If the tree is empty, set the root to the new node
        if(root == nullptr)
        {
            root = new treeNode(data);
            return root;
        }

        // Otherwise, return the new node
        else
            return new treeNode(data);
    }

    else if(data < curr->data)
    {
        curr->leftChild = insertToBST(curr->leftChild, data);
    }

    else if(data > curr->data)
    {
        curr->rightChild = insertToBST(curr->rightChild, data);
    }

    return curr;
}
```

Iteratively

```
treeNode* BST::insertToBSTIterative(treeNode* curr, int data)
{
    // If the tree is empty, set the root to the new node
    if(root == nullptr)
    {
        root = new treeNode(data);
        return root;
    }

    // Otherwise, traverse the tree until we find the correct spot
    else
    {
        treeNode* temp = root;

        while(temp != nullptr)
        {
            // If the data is less than the current node, go left
            if(data < temp->data)
            {
                if(temp->leftChild == nullptr)
                {
                    temp->leftChild = new treeNode(data);
                    return temp->leftChild;
                }

                else
                    temp = temp->leftChild;
            }
```

```
            // If the data is greater than the current node, go right
            else if(data > temp->data)
            {
                if(temp->rightChild == nullptr)
                {
                    temp->rightChild = new treeNode(data);
                    return temp->rightChild;
                }

                else
                    temp = temp->rightChild;
            }
        }
    }
}
```

3. Given the following BST, perform: **Preorder**, **Inorder**, and **Postorder** Traversal



Can you implement the code for all of the following traversals?

Preorder: **12, 9, 6, 3, 8, 11, 10, 14, 13, 20, 18, 25**

Inorder: **3, 6, 8, 9, 10, 11, 12, 13, 14, 18, 20, 25**

Postorder: **3, 8, 6, 10, 11, 9, 13, 18, 25, 20, 14, 12**

```
void BST::printPreOrder(treeNode* curr)
{
    if(curr == nullptr)
        return;

    cout << curr->data << " ";
    printPreOrder(curr->leftChild);
    printPreOrder(curr->rightChild);
}
```

```
void BST::printInOrder(treeNode* curr)
{
    if(curr == nullptr)
        return;

    printInOrder(curr->leftChild);
    cout << curr->data << " ";
    printInOrder(curr->rightChild);
}
```

```
void BST::printPostOrder(treeNode* curr)
{
    if(curr == nullptr)
        return;

    printPostOrder(curr->leftChild);
    printPostOrder(curr->rightChild);
    cout << curr->data << " ";
}
```

4. Implement a function that returns the **inorderSuccessor(node* root, node* givenNode)** of a given node.

5. Implement a **search(node *root, int value)** function that checks whether a value is present in a BST or not.

```
bool BST::search(treeNode* curr, int data)
{
    // If we have reached the end of the tree, return false, since we didn't find the value
    if(curr == nullptr)
        return false;

    // If we found the value, return true
    else if(curr->data == data)
        return true;

    else if(data < curr->data)
        return search(curr->leftChild, data);

    else
        return search(curr->rightChild, data);
}
```

```
bool search(node *root, int value) {
    if (root == nullptr) {
        return false;
    } else if (value < root->val) {
        return search(root->left, value);
    } else if (value > root->val) {
        return search(root->right, value);
    } else { // value == root->val
        return true;
    }
}
```

```
bool search(node* root, int value) {
    if (root == nullptr)
        return false;
    else if (root->val == value)
        return true;
    else if (value < root->val)
        return search(root->left, value);
    else // value > root->val
        return search(root->right, value);
}
```

6. Implement a function that **printsLeaves(node* root)** of a given BST.

```
12 void printLeaves(TreeNode* root){
13   if (root == nullptr) {
14     return;
15   }
16
17   if (root->left == nullptr && root->right == nullptr) {
18     // Node is a leaf when no child exist
19     cout << root->val << " ";
20     return;
21   }
22
23   // Recursively process left and right subtrees
24   printLeaves(root->left);
25   printLeaves(root->right);
26 }
```

7. Implement a function that checks if a given BST **isBST(node * root)**.

```
bool BST::isBST(treeNode* curr)
{
    if(curr == nullptr)
        return true;

    else if(curr->leftChild != nullptr && curr->leftChild->data > curr->data)
        return false;

    else if(curr->rightChild != nullptr && curr->rightChild->data < curr->data)
        return false;

    else
        return (isBST(curr->leftChild) && isBST(curr->rightChild));
}
```

8. Implement a function that adds the values in a BST into an array in descending order. **sortedValues(node* root, vector<int> &arr)**.

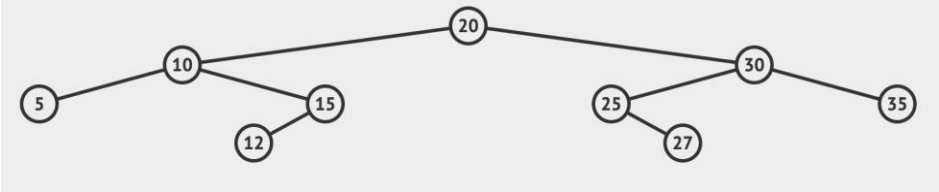
```
void BST::sortedValues(treeNode* curr, vector<int>& values)
{
    if(curr != nullptr)
    {
        sortedValues(curr->rightChild, values);
        values.push_back(curr->data);
        sortedValues(curr->leftChild, values);
    }
}
```

9. Implement a function that checks if a BST's structure **isMirror(node* root)** of itself (check if the root's left subtree structure is a mirror of its right subtree's structure)

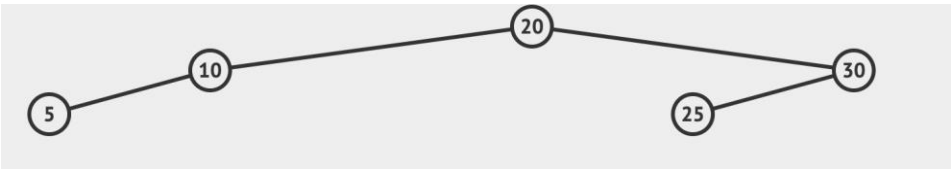
10. Implement a function that returns the depth of a given node (distance from the root node) starting at 0. Return -1 if the node doesn't exist in the tree → **getDepth(node* root, int targetNode, int depth)**

```
int BST::getDepth(treeNode* curr, int targetNode, int depth)
{
    // If we have reached the end of the tree, return -1
    if(curr == nullptr)
        return -1;

    // If we found the target node, return the current depth
    else if(curr->data == targetNode)
        return depth;

    // Otherwise, go left or right and increment the depth
    else if(targetNode < curr->data)
        return getDepth(curr->leftChild, targetNode, depth + 1);

    else
        return getDepth(curr->rightChild, targetNode, depth + 1);
}
```

isMirror() = TRUE



isMirror() = FALSE



Leetcode Problems w/ "Binary Search Tree" tags (easy/mediums) are good practice for the exam.

AVL Trees

1. Insert the following values into an AVL tree, show the tree after every insertion:

[21, 17, 15, 19, 18, 10, 8, 13]

2. Name the different sorts of rotations you use when performing the self-balancing in an AVL tree. Explain how they work and when you perform them.

3. Implement all the AVL rotation methods.

```
treeNode* AVLTree::singleLeftRotation(treeNode* curr)
{
  treeNode* temp = curr -> right;
  curr -> right = temp -> left;
  temp -> left = curr;

  curr -> height = max(height(curr -> left), height(curr -> right)) + 1;
  temp -> height = max(height(temp -> left), height(temp -> right)) + 1;
  return temp;
}
```

```
35 node *singleLeftRotation(node *root){
36     node *newRoot = root->right;
37     node *newRight = newRoot->left;
38     newRoot->left = root;
39     root->right = newRight;
40     return newRoot;
41   }
```

```
treeNode* AVLTree::singleRightRotation(treeNode* curr)
{
  treeNode* temp = curr -> left;
  curr -> left = temp -> right;
  temp -> right = curr;

  curr -> height = max(height(curr -> left), height(curr -> right)) + 1;
  temp -> height = max(height(temp -> left), height(temp -> right)) + 1;
  return temp;
}
```

```
43 node *singleRightRotation(node *root){
44     node *newRoot = root->left;
45     node *newLeft = newRoot->right;
46     newRoot->right = root;
47     root->left = newLeft;
48     return newRoot;
49 }
```

```
node *singleRightRotation(node *root){
  node *newRoot = root->left;
  root->left = newRoot->right;
  newRoot->right = root;
  return newRoot;
}
```

BETTER ONE

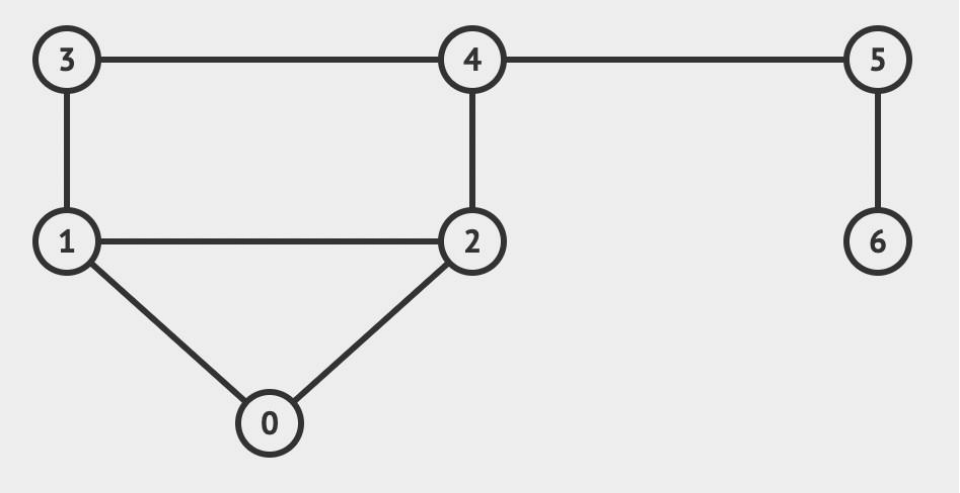
```
treeNode* AVLTree::leftRightRotation(treeNode* curr)
{
  curr -> left = singleLeftRotation(curr -> left);
  return singleRightRotation(curr);
}

treeNode* AVLTree::rightLeftRotation(treeNode* curr)
{
  curr -> right = singleRightRotation(curr -> right);
  return singleLeftRotation(curr);
}
```

4. Implement a function that returns the **balanceFactor(node *givenNode)** of a given node.

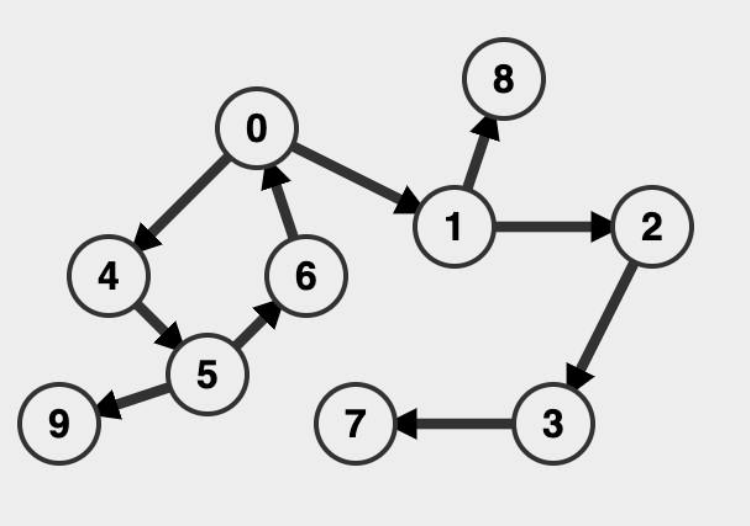
```
int AVLTree::bFactor(treeNode* curr)
{
  return (height(curr -> left) - height(curr -> right));
}
```

Graphs

1. Given the graph, write out its representation as an adjacency matrix and adjacency list.

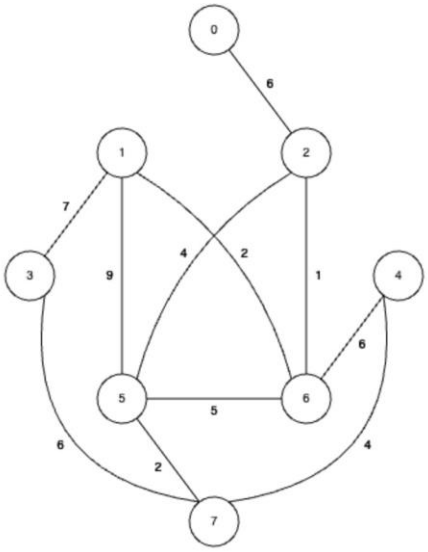


2. Perform the DFS algorithm on the following graph and show the order in which the nodes were visited (assuming 0 is the source node)

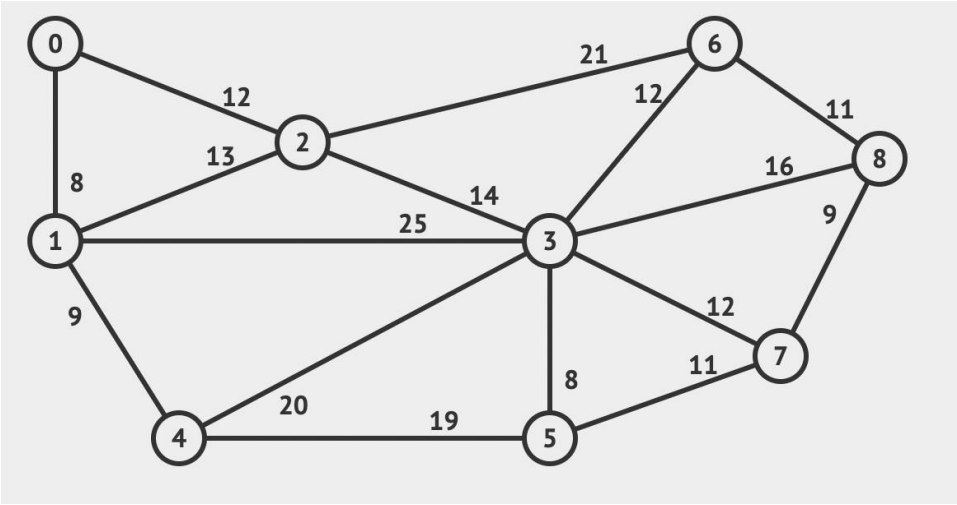


3. Implement the **BFS(int graph[][], int sourceNode, int numVertices)** algorithm represented as an adjacency matrix. It should as well print the order in which the vertices are visited.

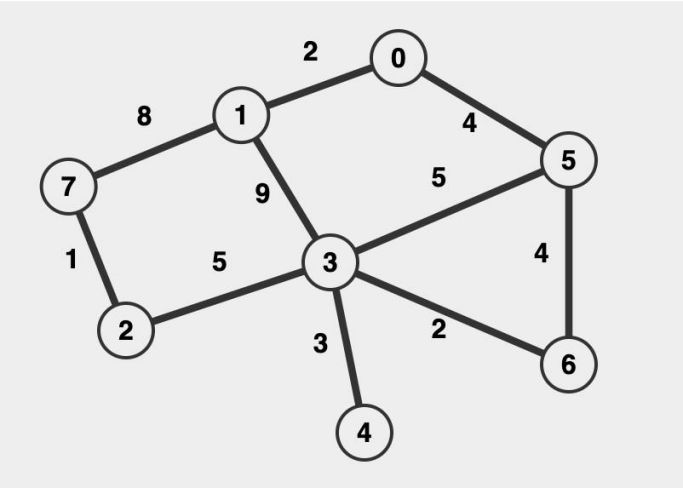
4. Perform Kruskal's Algorithm on the following graph and draw the minimum spanning tree.



5. Perform Prim's Algorithm on the following graph and draw the minimum spanning tree. Start from Vertex 5.



6. Given the graph, draw the shortest path tree using Dijkstra's Algorithm starting from Vertex 1.



0	1	2	3	4	5	6	7

B Trees

1. Go over past insertion into B-Tree problems covered in past Workshop sessions.

Additional stuff

BST

1. Write the function getSum( ) which returns the sum of all the

values in a BST.

```
struct node{
   int value;
   node *right;
   node *left;
};

int getSum(node *n){



}
```

```
int getSum(node *n){
  if(n == nullptr)
    return 0;
  return (n->value + getSum(n->right) + getSum(n->left));
}
```

2. Write the function leafCount( ) which returns the number of leafs in a BST.

```
struct node{
     int value;
     node *left;
     node *right;
};

int leafCount(node *root){



}
```

```
int leafCount(node *n){
  if(n == nullptr)
    return 0;
  else if(n->left == nullptr && n->right == nullptr)
    return 1;
  else
    return leafCount(n->left) + leafCount(n->right);
}
```

AVL

AVL Insertion Process

Insertion in an AVL tree is similar to insertion in a binary search tree. But after inserting and element, you need to fix the AVL properties using left or right rotations:

- If there is an imbalance in the *left child's right sub-tree*, perform a **left-right rotation**
- If there is an imbalance in the *left child's left sub-tree*, perform a **right rotation**
- If there is an imbalance in the *right child's right sub-tree*, perform a **left rotation**
- If there is an imbalance in the *right child's left sub-tree*, perform a **right-left rotation**

LEETCODE (BINARY SEARCH TREE EASY)

Given the root of a binary tree, return *its maximum depth*.

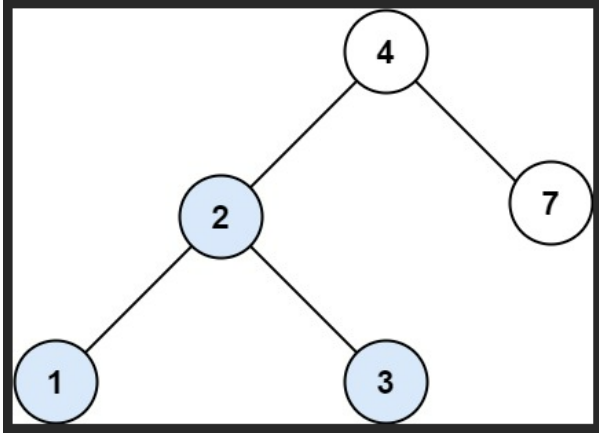
A binary tree's **maximum depth** is the number of nodes along the longest path from the root node down to the farthest leaf node.

```cpp
int maxDepth(TreeNode* root) {
    if (!root)
        return 0;

    return 1 + max(maxDepth(root->left), maxDepth(root->right));
}
```

You are given the `root` of a binary search tree (BST) and an integer `val`.

Find the node in the BST that the node's value equals `val` and return the subtree rooted with that node. If such a node does not exist, return `null`.

Example 1:



Input: root = [4,2,7,1,3], val = 2
Output: [2,1,3]

```
TreeNode* searchBST(TreeNode* root, int val) {
    if(!root || root->val==val)
        return root;

    if(val < root->val)
        return searchBST(root->left, val);

    else
        return searchBST(root->right, val);
}
```