Data Structures Final Review (Workshop)

Topics Covered:

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

Binary Search Trees

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

The BST property ensures that at each node, the left subtree contains elements smaller than the current node, and the right subtree contains elements greater than the current node. This allows for certain operations, like searching and insertion to on average be O(log(n)) time complexity. (Obviously, this is dependent on how balanced your BST is though).

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

Recursive:

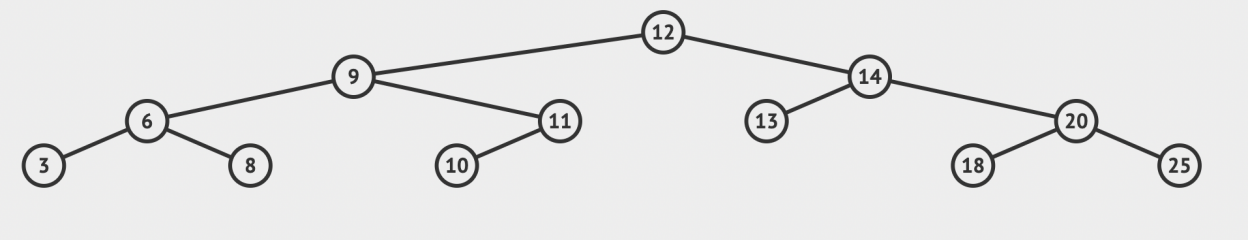
```
treeNode* BST::insertToBST(treeNode* curr, int data)
{
    // If we have reached the end of the tree, insert the new node
    if(curr == nullptr)
    {
        // If the tree is empty, set the root to the new node
        if(root == nullptr)
        {
            root = new treeNode(data);
     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

Iterative:

```
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: **Preoder**, **Inorder**, and **Postorder** Traversal



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

Preoder: 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)
    {
        cout << curr->data << " ";
        printPreOrder(curr->leftChild);
        printPreOrder(curr->rightChild);
    }
}

void BST::printInOrder(treeNode* curr)
{
    if(curr != nullptr)
    {
        printInOrder(curr->leftChild);
        cout << curr->data << " ";
        printInOrder(curr->rightChild);
    }
}

void BST::printPostOrder(treeNode* curr)
{
    if(curr != nullptr)
    {
        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.

```
treeNode* BST::getInorderSuccessor(treeNode* givenNode)
{
    // If the given node is null or the right child is null, return null
    if(givenNode == nullptr || givenNode -> rightChild == nullptr)
        return nullptr;

    // Otherwise, return the leftmost node in the right subtree
    treeNode* temp = givenNode -> rightChild;
    while(temp -> leftChild != nullptr)
        temp = temp -> leftChild;

    // Return the leftmost node in the right subtree
    return temp;
}
```

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

```
bool BST::searchBST(treeNode* curr, int data)
{
    if(curr == nullptr)
        return false;

    else if(curr->data == data)
        return true;

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

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

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

```
void BST::printLeaves(treeNode* root)
{
   if(root == nullptr)
      return;

    // If the node is a leaf, print it
    if(root -> leftChild == nullptr && root -> rightChild == nullptr)
        cout << root -> data << " ";

    // Otherwise, recursively call the function on the left and right subtrees
    printLeaves(root -> leftChild);
    printLeaves(root -> rightChild);
}
```

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

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

    // If the left child is greater than the current node, return false
    else if(curr->leftChild != nullptr && curr->leftChild->data > curr->data)
        return false;

    // If the right child is less than the current node, return false
    else if(curr->rightChild != nullptr && curr->rightChild->data < curr->data)
        return false;

    // Otherwise, recursively call the function on the left and right subtrees
    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)**.

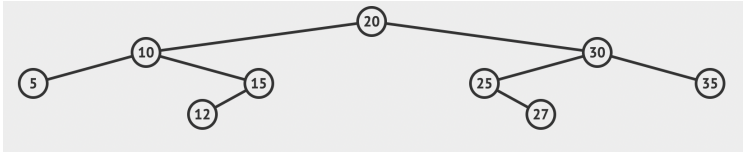
```cpp
void BST::sortedValues(treeNode* curr, vector<int>& values)
{
    if(curr != nullptr)
    {
        sortedValues(curr->rightChild, values);
        values.push_back(curr->data);
        sortedValues(curr->leftChild, values);
    }
}
```

9. Implement a function that 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)**

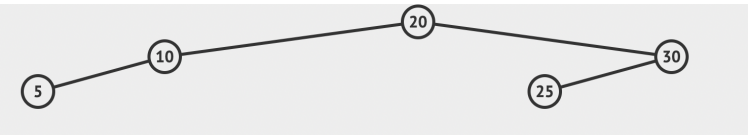
```cpp
int BST::getDepth(treeNode* root, int targetNode, int depth)
{
    // If the root is null, return -1
    if(root == nullptr)
        return -1;

    // If the root is the target node, return the current depth
    else if(root -> data == targetNode)
        return depth;

    // Otherwise, recursively call the function on the left and right subtrees
    else if(targetNode < root -> data)
        return getDepth(root -> leftChild, targetNode, depth + 1);

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

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

isMirror() = TRUE



isMirror() = FALSE



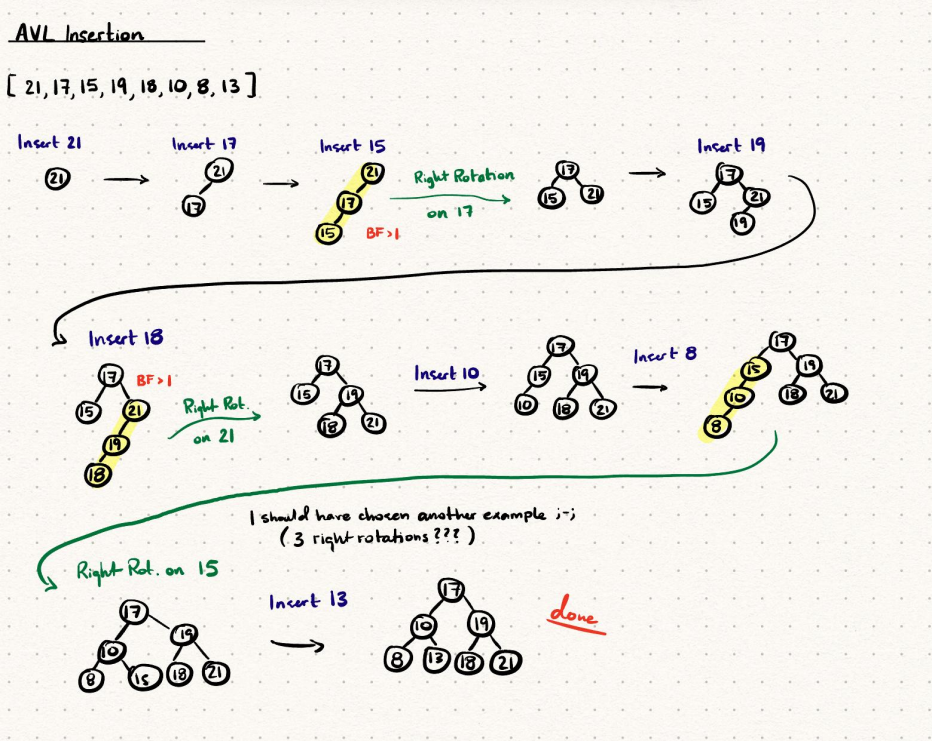
```cpp
bool BST::helper(treeNode* leftSubtreeRoot, treeNode* rightSubtreeRoot)
{
    // If both nodes are null, return true
    if(leftSubtreeRoot == nullptr && rightSubtreeRoot == nullptr)
        return true;

    // If one node is null and the other is not, return false
    else if(leftSubtreeRoot == nullptr || rightSubtreeRoot == nullptr)
        return false;

    return helper(leftSubtreeRoot -> leftChild, rightSubtreeRoot -> rightChild)
        && helper(leftSubtreeRoot -> rightChild, rightSubtreeRoot -> leftChild);
}

bool BST::isMirror(treeNode* root)
{
    if(root == nullptr)
        return true;

    return helper(root -> leftChild, root -> rightChild);
}
```

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.

   Single Left Rotation, Single Right Rotation, Left Right Rotation, Right Left Rotation
   - Look at workshop slides for more information

3. Implement all the AVL rotation methods.

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

  return temp;
}
```

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

  return temp;
}
```

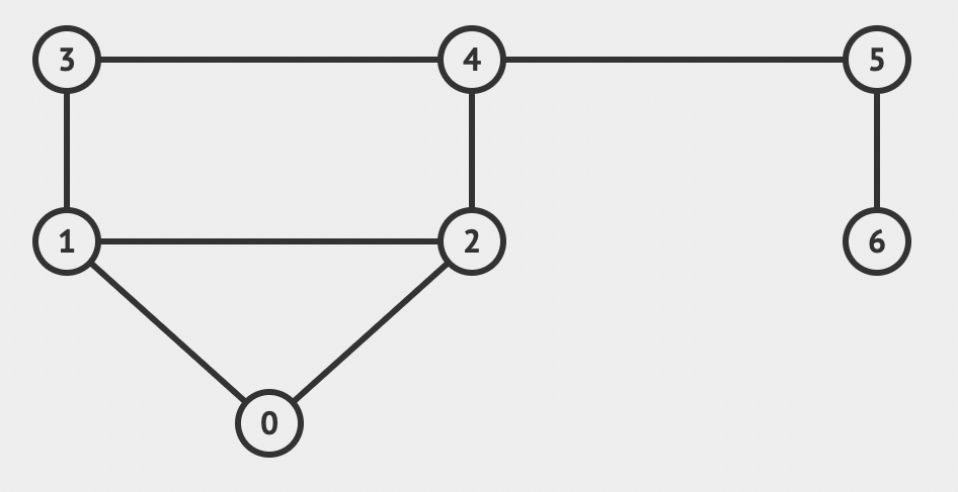
```
treeNode* AVLTree::leftRightRotation(treeNode* curr)
{
  curr -> left = singleLeftRotation(curr -> left);
  return singleRightRotation(curr);
}


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::balanceFactor(treeNode* curr)
{
  if(curr == nullptr)
    return 0;

  else
    return (height(curr -> left) - height(curr -> right));
}
```

Graphs

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



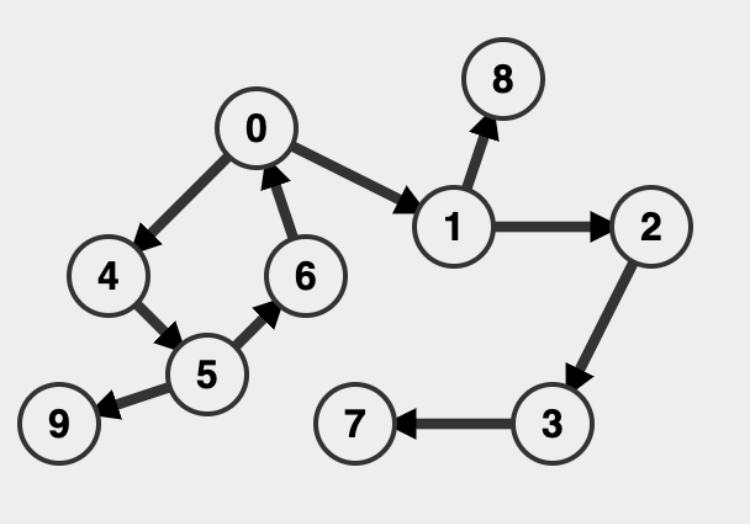
Adjacency Matrix

	0	**1**	**2**	**3**	**4**	**5**	**6**
0	0	1	1	0	0	0	0
1	1	0	1	1	0	0	0
2	1	1	0	0	1	0	0
3	0	1	0	0	1	0	0
4	0	0	1	1	0	1	0
5	0	0	0	0	1	0	1
6	0	0	0	0	0	1	0

Adjacency List

0 → 1, 2
1 → 0, 2, 3
2 → 0, 1, 4
3 → 1, 4
4 → 2, 3, 5
5 → 4, 6
6 → 5

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)



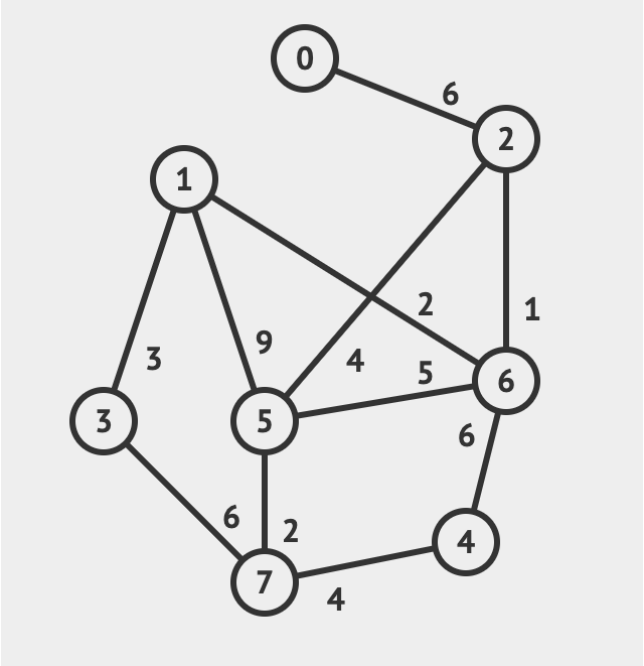
0, 1, 2, 3, 7, 8, 4, 5, 6, 9

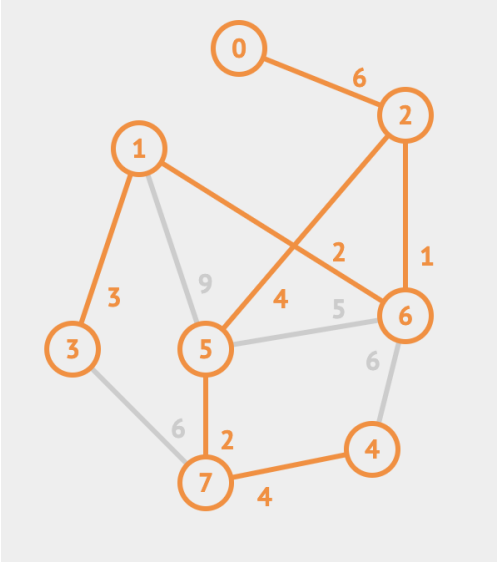
(refer to Teams recorded video for visualization)

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.

```
void BFS(int **adjacencyMatrix, int sourceNode, int numNodes)
{
    queue<int> q;
    bool *visited = new bool[numNodes];

    for (int i = 0; i < numNodes; i++)
        visited[i] = false;

    q.push(sourceNode);

    while (!q.empty())
    {
        int currentNode = q.front();
        q.pop();

        if(!visited[currentNode])
        {
            cout << currentNode << " ";
            visited[currentNode] = true;
        }

        for (int i = 0; i < numNodes; i++)
        {
            if (adjacencyMatrix[currentNode][i] == 1 && !visited[i])
                q.push(i);
        }
    }

    delete[] visited;
}
```

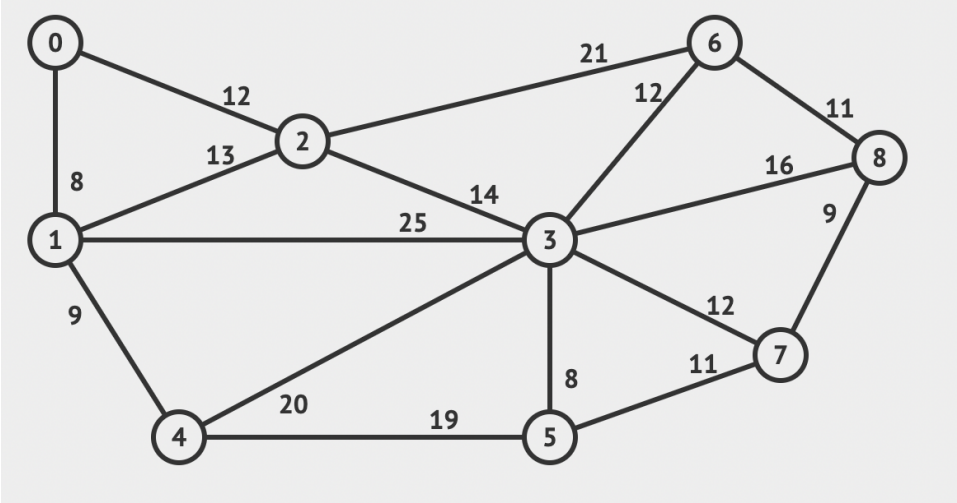
4. Perform Kruskal's Algorithm on the following graph and draw the minimum spanning tree.

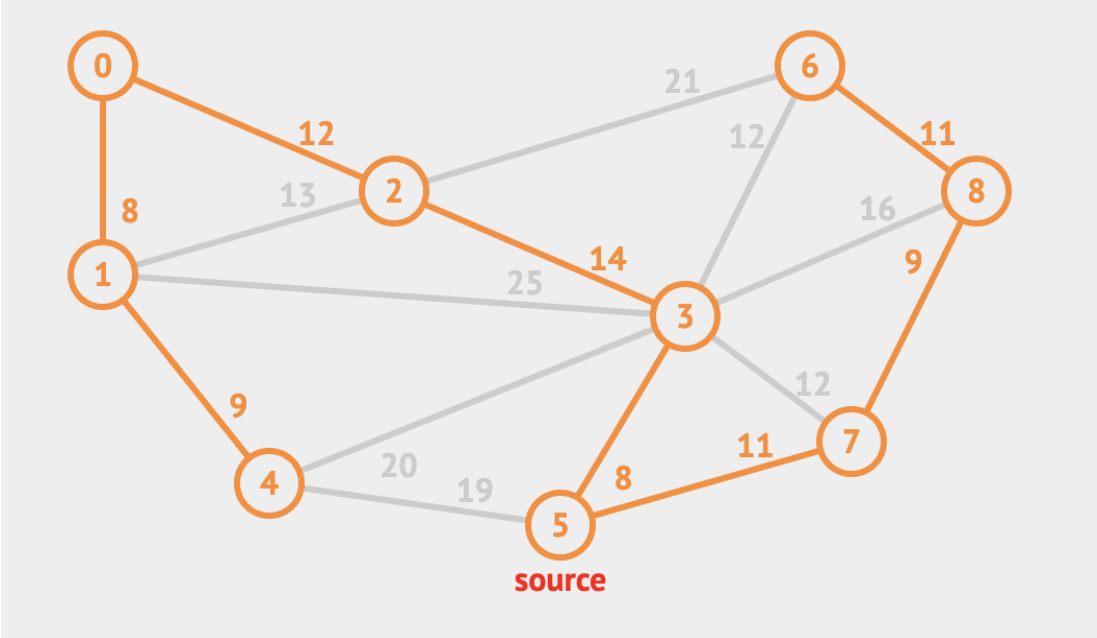




(refer to Teams recorded video for visualization)

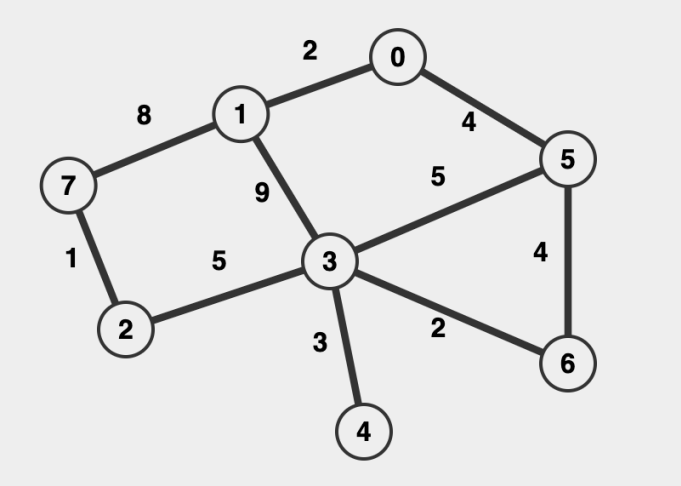
5. Perform Prim's Algorithm on the following graph and draw the minimum spanning tree. Start from Vertex 5.





(refer to Teams recorded video for visualization)

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



Underlining shows the next node we visit (always take shortest one that hasn't been visited yet)

0	1	2	3	4	5	6	7
INF	<u>0</u>	INF	INF	INF	INF	INF	INF
<u>2</u>		INF	9	INF	INF	INF	8
		INF	9	INF	<u>6</u>	INF	8
		INF	9	INF		10	<u>8</u>
		9	<u>9</u>	INF		10	
		<u>9</u>		12		10	
				12		<u>10</u>	
				<u>12</u>			

(We'll take 3 first because it was a neighboring node before 2)

<u>Order of Visited:</u> 1, 0, 5, 7, 3, 2, 6, 4

<u>(refer to Teams recorded video for visualization)</u>

B Trees

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