Question 1:

**Approach 1 -> Binary Search Tree**

Let HeapNode be an object which holds item and priority value. Let Heap be an binary heap, an array which holds objects in the order of their priority.

Since binary search doesn’t involves ordering items based on their priority, it takes linear time to search for items in binary heap. In order to reduce the time complexity involved in searching an item in binary heap we can use balanced binary search tree to keep track of index of the item in the BinaryHeap. This reduces the time complexity of search operation to O(logn) from O(n).

In AVL balanced BinarySearchTree each node is associated with an item, index, leftNode , rightNode and height of the node in the tree.

In BinaryHeap each node is associated with the priority and item

In AVL BST implementation we are recursive approach so we don’t need to have pointers to their as we can access parent node while traversing back using the recursive stack frame.

**Balanced BinarySearchTree methods:**

Steps involved in insertNodeBST(item, index)

1. Add item into the tree using normal binary search tree operation i.e. start from root node, move right if item to be greater than the current node item value else move left.

This is done recursively until we reach the leaf node

1. On encountering a leaf node we create a new node using the passed index, item and setting its left and right child as null. This node is attached to the current node as appropriate child to satisfy binary search tree property left(item) <item < right(item).
2. After adding item we need to check If the tree is balanced at the current node i.e. difference between the height of the left node and the height of the right node is not more than one.
3. If the node is unbalanced we employ one of the techniques mentioned below to convert unbalanced BST to balanced BST:

Case i: Right branch is longer than the left and item inserted is greater than right node item value. (Right- Right case)

In this case in-order to preserve the balanced tree property we perform left rotation on the current node

Case ii : Right branch is longer than the left and item inserted is greater than right node item value. (Right left case)

In this case in-order to preserve the balanced tree property we perform right rotation on the right child of the node, followed by left rotate on the current node

Case iii : Left branch is longer than the right and item inserted is greater than the left node (Left-right case)

In this case in-order to preserve the balanced tree property we perform left rotation on the left-child of the current node followed by right rotation on the current node

Case iv: Left branch is longer than the right and item inserted is lesser than the left node

(Left-left case)

In this case in-order to preserve the balanced BST property we perform right rotation on the current node

During each step in back traversal of the recursive stack frame, we perform the above mentioned steps to balance the tree.

Steps involved in rightRotation(BSTNode node):

1. Let leftNode be node.left
2. Assign leftNode.right to node.left
3. Assign node to leftNode.right
4. Update height of node
5. Update height of leftNode
6. Return leftNode

Steps involved in leftRotation(BSTNode node):

1. Let rightNode be node.right
2. Assign rightNode.left to node.right
3. Assign node to rightNode.left
4. Update height of node
5. Update height of rightNode
6. Return rightNode

Steps involved In fetchIndexOfItem (int item)

1. Start from the root node of the binary search tree. Recursively traverse through the nodes of the tree.
2. Check if the item value of the current node is greater than or lesser than the item to be searched.
3. If the value is greater then call the same function on right node.
4. Else if the value is lesser then call the same function on left node.
5. Else if its equal then return the index value stored in that node.
6. If the item being searched is not found return -1

Steps involved In setIndexOfItem(int node,int item, int newIndex)

1. Start from the root node of the binary search tree. Recursively traverse through the nodes of the tree.
2. Check if the item value of the current node is greater than or lesser than the item to be searched.
3. If the value is greater then traverse right child of the node
4. Else if the value is lesser then traverse left child of the node
5. Else if its equal then set the index of that node to new index.

**BinaryHeap Methods:**

Steps Involved in add(item, priority):

1. Create a new HeapNode using the given item and priority.
2. Add this new HeapNode to the end of binaryHeap .
3. Add the new item along with it’s index to the AVL Balanced binary search tree. Each node in AVL tree consists of height of the node, the item it holds and the index position of this node in Binary Heap
4. Perform bubble up operation on the new added node in binary heap by passing index of the node. This ensures that the max binary heap property is maintained. i.e. all the nodes have priority greater than the priority of their children.

Steps involved in BubbleUp(index)

1. Read the node at the given index.

2. Perform the subsequent operations until we reach first node.

3. If the priority of the node at the given index is greater than the priority of

the parent node then make call to swap function to swap the node with its parent and

make appropriate changes in AVL balanced binary search tree

4. After every swap we set the current node as parent and repeat step 3.

1. We continue process until we have reached the first node or until we come across a

parent node whose priority is greater than the current node.

Steps involved in BubbleDown(index)

1. Read the node at the given index.

2. Perform the subsequent operations until we reach last parent index in binary heap.

3. Compare the priority of the node with the priority of their children. If priority of

both the children are of higher than the parent then select the node with higher

priority then make call to swap function to swap the node with its parent and

make appropriate changes in AVL balanced binary search tree

4. Now make the child node as parent and continue the process until you reach the last

parent node which is at the index position (binaryHeap.length -2)/2 or until you reach

a point where priority of both the children are lesser than the current parent node

selected.

Steps involved in swap (int node1,int index, int node2, int index2)

1. Find the nodes in the balanced binary search tree with value node1.item and node2.item. Let them be oldNode1 and oldNode2 respectively.
2. Now change the index value in oldNode1 to index2 and change the index value in oldNode2 to index1
3. We swap the nodes in the binaryheap, we set node1 at index2 and node2 at index1 .

Steps Involved in getMaxPriorityItem():

1. We return the item corresponding to the first node in the binaryHeap array since the binary heap we are building is a max heap and the first element in the binaryHeap corresponds to the node with highest priority in the entire binary heap array.

Steps Involved in changePriorityOfItem(item, priority):

1. Fetch the index of the item whose priority needs to be changed from the

Balanced BinarySearch tree by traversing the tree.

1. If item is not found then we do nothing to the existing binary heap or to the AVL balanced binary search tree.
2. Change the priority of the node in the binaryHeap, at the index position fetched in the previous step
3. If the new priority is lesser than the old priority then we will have to check if the priority of any of it’s children are greater than the new priority. So we need to perform bubbleDown Operation starting from the index of the node whose priority is changed.
4. Else if the new priority is greater than the old priority, then chances are that the parent node priority might be lesser than the new priority.
5. So we need to perform bubbleUp operation starting from the index of the node whose priority is changed.
6. BubbleUp and Bubbledown operations will ensure that the max binary heap principle is maintained

Class HeapNode

{

int item;

int priority;

HeapNode(int item, int priority)

{

this.item = item;

this.priority = priority;  
}

}

Class BSTNode

{  
 int item;

int height;

int index;

BSTNode left , right;

BSTNode(int item, int index)

{

this.item = item;

this.index = index;

this.height = 1;

this.left = null;

this.right = null;

}

}

Class BalancedBST

{

BSTNode root;

int heightOfNode(BSTNode node)

{

If(node = null)

return 0;

return node.height;

}

Node rightRotate(BSTNode node)

{

BSTNode leftNode = node.left;

BSTNode tempRight = leftNode.right;

// rotating right

node.left = tempRight;

leftNode.right = node;

// updating the heights after right rotation

node.height = Math.max(heightOfNode(node.left),heightOfNode(node.right))

leftNode.height = Math.max( heightOfNode( leftNode.left ),

heightOfNode( leftNode.right ))

return leftNode;

}

Node leftRotate(BSTNode node)

{

BSTNode rightNode = node.right;

BSTNode tempLeft = rightNode.left;

// rotating left

node.right = tempLeft;

rightNode.left = node

// updating the heights after left rotation

node.height = Math.max(heightOfNode(node.left),heightOfNode(node.right))

rightNode.height = Math.max( heightOfNode( rightNode.left ),

heightOfNode( rightNode.right ))

return rightNode;

}

int searchItem(int item)

{

return fetchIndexOfItem(this.root, item);

}

int fetchIndexOfItem (BSTNode root, int item)

{

If(root == null)

{

return -1;

}

else

{  
 if(root.item == item)

{  
 return this.index

}

else if(root.item > item)

{

return fetchIndexOfItem (root.left, item);

}

else

{

return fetchIndexOfItem (root.right, item);

}

}

}

void setItemIndex(int item, int newIndex)

{  
 setIndex(this.root, item, newIndex);

}

void setIndexOfItem (BSTNode root, int item, int newIndex)

{

if(root.item == item)

{  
 this.index = newIndex;

}

else if(root.item > item)

{

setIndexOfItem (root.left, item, newIndex);

}

else

{

setIndexOfItem (root.right, item, newIndex);

}

}

void insertNodeBST(int item, int index)

{

insertNodeAVLBST(this.root, item, index);

}

void insertNodeAVLBST(BSTNode node, int item, int index)

{

if ( node == null)

{

return new BSTNode(item,index);

}

If( key < node.key)

{

node.left = insertNodeAVLBST (node.left, item, index);

}

else if( key > node.key )

{

node.right = insertNodeAVLBST (node.right, item, index);

}

else

{

return node;

}

node.height = Math.max( heightOfNode( node.left),

heightOfNode( node.right)) +1;

int balanceHeight;

if(node == null)

balanceHeight =0;

else

balanceHeight = (node.left – node.right);

// Left Left case

if(balanceHeight > 1 && item < node.left.item)

{

return rightRotate(node);

}

// Left Right case

else if( balanceHeight > 1 && item > node.left.item)

{

node.left= leftRotate(node.left);

return rightRotate(node);

}

// Right Left case

else if( balanceHeight > -1 && item < node.right.item)

{

node.right = rightRotate(node.right);

return leftRotate(node);

}

// Right Right case

else if( balanceHeight > -1 && item > node.right.item)

{

return leftRotate(node);

}

// when there is no imbalance in tree

return node;

}

}

Class BinaryHeap

{

// AVL Balanced Binary Search tree to keep track of index of the item in BinaryHeap

BalancedBST bst;

// Max Binary Heap Array List to hold Node in the order of their priority

// Node with the max priority is the first element in the list

Array<HeapNode> binaryHeap = new ArrayList<Node>();

add(int item, int priority)

{

// adding item and it’s index to the map

bst.insertNodeBST(item, binaryHeap.length);

// append the new item to the end of the array list

BinaryHeap.add(new HeapNode(item, priority);

// Let n be the length of the binary heap

// Now perform Bubble up operation to make sure heap property is

// maintained i.e. all the parent nodes have higher priority than its children

// node

// initially consider the parent of the last node which is inserted

Let index= n-1

// bubble up operation is performed to maintain the heap property after //adding new item

bubbleUp(index)

}

bubbleUp(int index)

{

While( index >=0 )

{

HeapNode parent = BinaryHeap.get((index-2)/2)

HeapNode node = BinaryHeap.get(index)

if( parent.priority > node.priority)

{

break;

}

//if the parent nodes priority is lesser than the child nodes priority

// then we swap parent and the child node and move up and continue // this until we reach first node

swap(index ,node , (index-2)/2, parent)

// parent index becomes currIndex

Index = (index-2)/2;

}

}

swap(int Index1,HeapNode node1, int index2, HeapNode node2)

{

// swapping HeapNodes in BinaryHeap  
 binaryHeap.set(index1,node2);

binaryHeap.set(ndex2, node1);

// swapping the indexes in the BST

setItemIndex (node1.item, index2)

setItemIndex (node2.item, index1)

}

bubbleDown(int index)

{

Int lastParent = (binaryHeap.length-2)/2

while(index <= lastParent)

{

Int maxIndx = index;

Int leftChildIndx = (2\*index )+1;

// if left child priority is greater than parent node then swap

// parent with left child

If( leftChildIndx < lastParent )

{

HeapNode currMaxNode = binaryHeap.get(maxIndx)

Int leftChild = binaryHeap.get(leftChildIndx)

If( leftChild.priority > currMaxNode.priority)

{

maxIndx = leftChildIndx

maxNode = leftChild;

}

}

// if right child priority is greater than parent node then swap

// parent with right child

Int rightChildIndx = (2\*index )+2;

If( rightChildIndx < lastParent)  
 {

HeapNode currMaxNode = binaryHeap.get(maxIndx)   
Int rightChild = binaryHeap.get(rightChildIndx);

If( rightChild.priority > currMaxNode.priority)

{  
 maxIndx = rightChildIndx;

maxNode = rightChild;

}

}

If( maxIndx == index) break;

swap(index, currMaxNode, maxIndx, maxNode);

Index = maxIndx

}

}

getItemWithHighestPriority()

{

// item with highest priority is the first element in the BinaryHeap

HeapNode highPriorityNode=binaryHeap.get(0);

// return item corresponding to the first node

Return highPriorityNode.item;

}

changePriority(int item, int priority)

{

// fetch the index of the item in BST

int nodeIndx = fetchIndexOfItem(item);

HeapNode node = binaryHeap.get(nodeIndx);

// change the priority of the old node to new priority

node.priority = priority;

// if the new priority is greater than the older priority then

// perform bubble up operation else bubble down operation

// This ensures that the max binary heap property is maintained

// i.e. priority of a node should be greater than the priority

// of it’s children

If(node.priority > priority)

{

bubbleUp(nodeIndx);

}

else

{

bubbleDown(nodeIndx);

}

}

**Time Efficiency:**

Time complexity of **add(item,priority)** functionality:

1. This involves adding item to binaryheap at the end. Time taken = C1
2. After adding it to binary heap, we need to perform bubble up operation. Max time taken for this = h, where h is height of BST

Since this a balanced binary search tree, h = log n (n is the no of nodes in binaryHeap)

Therefore, time taken = log n.

1. This also involved adding item to the Balanced BinarySearchTree. This operation takes= h, as it involves traversing from root node one of the leaf nodes whose length is atmost height of the tree. .

Therefore time taken = log n

1. All the other steps take constant amount of time = c2

By add adding all the above mentioned values, We have

Time complexity, T(n) = logn +logn + c1 + c2

= 2logn + C

* **T(n) = O(logn)**

Time complexity of **getItemWithHighestPriority()**  functionality:

Since we are creating a max binary heap, the first item in the heap is always item with highest priority. This takes some constant amount of time c.

Therefore Time Complexity T(n) = O(1)

Time Complexity of changePriority(item, newPriority)

This involves fetching the item index from BST which holds item along with its index.

Since BST holds the item in a particular order, the time taken to fetch the index of the item

will at-most be height of the tree = log n

Therefore, time taken to fetch the item index = log n

Next it involves changing the priority of the heapNode at that particular index to the newPriority provided.

This involves bubble up operation if the newPriority > oldPriority of the item or bubble down operation is newPriority < oldPriority.

Time taken by bubbleUp operation and bubbleDown operation is at-most height of the tree as they traverse from the current node either till root node or till leaf node respectively.

Therefore time taken = log n

All the other steps take constant amount of time = C

Total time taken = ( log n )2+ C

After ignoring all the small terms, we have

* T(n) = O(log n)2

We can make use of a reference to the node in BST in BinaryHeap and to improve the complexity to work in (log n).

**Approach -2 using HashMap :**

Let HeapNode be an object which holds item and priority value. Let Heap be an binary heap, an array which holds objects in the order of their priority.

Since binary search doesn’t involves ordering items based on their priority, it takes linear time to search for items in binary heap. In order to reduce the time complexity involved in searching an item in binary heap we can use HashMap to map item with its index in the BinaryHeap. This reduces the time complexity of search operation to O(1).

Steps Involved in add (item, priority):

1. Create a new HeapNode using the given item and priority.
2. Add this new HeapNode to the end of binaryHeap .
3. Add the new item to the HashMap “mapNode” with its key as item and value as length of the binary heap.
4. Perform bubble up operation by passing index of the new added node. This ensures that the max binary heap property is maintained. i.e all the nodes have priority greater than the priority of their children.

Steps involved in BubbleUp(index)

1. Read the node at the given index position.

2. Check if the index of the node is greater than zero.

3. Next, If the priority of the node at the given index is greater than the priority of

the parent node then we swap the selected node with the parent node.

4.After every swap we set the current node as parent and repeat steps 2 and 3.

1. We continue process until we have reached the first node or until we come across a

parent node whose priority is greater than the current node.

Steps involved in BubbleDown(index)

1. Read the node at the given position

2. Check if the index of the node is within the BinaryHeap array range.

3. Now, compare the priority of the node with the priority of their children. If priority of

both the children are of higher than the parent then select the node with higher

priority and swap it with the parent node

4. Now make the child node as parent and continue the process until you reach the last

parent node which is at the index position (binary Heap.length -2)/2 or until you reach

a point where priority of both the children are lesser than the current parent node

selected.

Steps Involved in getMaxPriorityItem ():

1. We return the item corresponding to the first node in the binaryHeap array since the binary heap we are building is a max heap and the first element in the binaryHeap corresponds to the node with highest priority in the entire binary heap array.

Steps Involved in changePriorityOfItem(item, priority):

1. Fetch the index of the item whose priority needs to be changed from the hashmap “mapNode”.
2. Change the priority of the node in the binaryHeap, at the index position fetched in the previous step
3. If the new priority is lesser than the old priority then we will have to check if the priority of any of it’s children are greater than the new priority. So we need to perform bubbleDown Operation starting from the index of the node whose priority is changed.
4. Else if the new priority is greater than the old priority, the chances are that the parent node priority might be lesser than the new priority.
5. So we need to perform bubbleUp operation starting from the index of the node whose priority is changed.
6. BubbleUp and Bubbledown operations will ensure that the max binary heap principle is maintained

Class HeapNode

{

int item;

int priority;

HeapNode(int item, int priority)

{

this.item = item;

this.priority = priority;  
}

}

Class BinaryHeap

{

// Hashmap to map the item with its index in the binary heap

HashMap<int, int> mapNode = new HashMap<int, int>();

// Max Binary Heap Array List to hold Node in the order of their priority

// Node with the max priority is the first element in the list

Array<HeapNode> binaryHeap = new ArrayList<Node>();

Add(int item, int priority)

{

// adding item and it’s index to the map

mapNode.add(item, binaryHeap.length);

// append the new item to the end of the array list

BinaryHeap.add(new HeapNode(item, priority);

// Let n be the length of the binary heap

// Now perform Bubble up operation to make sure heap property is

// maintained i.e. all the parent nodes have higher priority than its children

// node

// initially consider the parent of the last node which is inserted

Let index= n-1

// bubble up operation is performed to maintain the heap property after //adding new item

bubbleUp(index)

}

bubbleUp(int index)

{

While( index >=0 )

{

HeapNode parent = BinaryHeap.get((index-2)/2)

HeapNode node = BinaryHeap.get(index)

if( parent.priority > node.priority)

{

break;

}

//if the parent nodes priority is lesser than the child nodes priority

// then we swap parent and the child node and move up and continue // this until we reach first node

swap(index ,node , (index-2)/2, parent)

// parent index becomes currIndex

Index = (index-2)/2;

}

}

swap(int Index1,HeapNode node1, int index2, HeapNode node2)

{

binaryHeap.set(node2.item, index1)

binaryHeap.set( node1.item, index2)

// swapping the indexes in the hashmap

mapNode[node1.item] = index2

mapNode[node2.item] = index1

}

bubbleDown(int index)

{

Int lastParent = (binaryHeap.length-2)/2

while(index <= lastParent)

{

Int maxIndx = index;

Int leftChildIndx = (2\*index )+1;

// if left child priority is greater than parent node then swap

// parent with left child

If( leftChildIndx < lastParent )

{

HeapNode currMaxNode = binaryHeap.get(maxIndx)

Int leftChild = binaryHeap.get(leftChildIndx)

If( leftChild.priority > currMaxNode.priority)

{

maxIndx = leftChildIndx

maxNode = leftChild;

}

}

// if right child priority is greater than parent node then swap

// parent with right child

Int rightChildIndx = (2\*index )+2;

If( rightChildIndx < lastParent)  
 {

HeapNode currMaxNode = binaryHeap.get(maxIndx)   
Int rightChild = binaryHeap.get(rightChildIndx);

If( rightChild.priority > currMaxNode.priority)

{  
 maxIndx = rightChildIndx;

maxNode = rightChild;

}

}

If( maxIndx == index) break;

swap(index, currMaxNode, maxIndx, maxNode);

Index = maxIndx

}

}

getItemWithHighestPriority()

{

// item with highest priority is the first element in the BinaryHeap

HeapNode highPriorityNode=binaryHeap.get(0);

// return item corresponding to the first node

Return highPriorityNode.item;

}

changePriority(int item, int priority)

{

// fetch the index of the item using hashmap

int nodeIndx = mapNode[item];

HeapNode node = binaryHeap.get(nodeIndx);

// change the priority of the old node to new priority

node.priority = priority;

// if the new priority is greater than the older priority then

// perform bubble up operation else bubble down operation

// This ensures that the max binary heap property is maintained

// i.e. priority of a node should be greater than the priority

// of it’s children

If(node.priority > priority)

{

bubbleUp(nodeIndx);

}

else

{

bubbleDown(nodeIndx);

}

}

**Time Efficiency:**

Time complexity of **add(item, priority)** functionality:

1. This involves adding item to binaryheap at the end. Time taken = log n as it involves re-heapify operation.
2. After adding it to binary heap, we need to add the item with its corresponding index into hash mapwhich takes constant time

Therefore time taken =c1

1. All the other steps take constant amount of time = c2

By add adding all the above mentioned values, We have

Time complexity, T(n) = logn + c1 + c2

= logn + C

* **T(n) = O(logn)**

Time complexity of **getItemWithHighestPriority()**  functionality:

Since we are creating a max binary heap, the first item in the heap is always item with highest priority. This takes some constant amount of time c.

Therefore Time Complexity T(n) = O(1)

Time Complexity of changePriority(item, newPriority)

This involves fetching the item index from HashMap which involves constant amount of time. = O(1) =C

After we get the node in the binaryheap in the index got in previous step.

Therefore, time taken to fetch the item index = constant time = c2

After fetching the item we need to change the priority of the item in the node we fetched and

Change of priority triggers bubble up or bubble down operation

This involves bubble up operation if the newPriority > oldPriority of the item or bubble down operation is newPriority < oldPriority.

Time taken by bubbleUp operation and bubbleDown operation is at-most height of the tree as they traverse from the current node either till root node or till leaf node respectively.

Therefore time taken = log n

All the other steps take constant amount of time = C1

Total time taken = ( log n )+ C + C1

After ignoring all the small terms, we have

**T(n) = O(log n)**

**Question 2 :**

Inorder to create a tree which allows us to search operation in log n time we need to create a balance binary search tree using all the nodes of the given unbalanced tree, whose height will be log n

Algorithm :

1. Sort the elements of the tree in the increasing order by doing inorder traversal on the binary search tree.
2. Once the elements are sorted we can create a balanced binary search tree by

Inserting middle element into the tree first.

1. We recursively call the same function on the left and right part part of the sorted array. This allows to insert in the order which creates an balanced binary search tree.

Time Complexity :

Sorting elements which involves inorder traversal takes linear time as it involves visiting all nodes in the tree recursively. As mentioned in class, we can think of it as edge traversals. Each edge in the binary search tree is traversed exactly twice.

Number of edges in the binary search tree with n nodes = (n -1)

Therefore maximum time taken = 2(n -1) = 2n- 2

Time taken by the recursive call to insert the mid element of the array to create a balanced tree = 2T(n/2) + C

T(n) = T(n/4) +2C

= T(n/8) +3C

:

:

* T(n) = T(1) + i\*C -> Equation 1

By using above equations this can be generalized as

T(n) = T(n/2^i) + i\*C -> Equation 2

By using Equation 1 and 2, we have:

n = 1 => i = log2 n -> Equation 3

2i

Substituting Equation 3 in 1, we have:

T(n) = T(1) +C ( log2 n)

Since T(1) is running time when n=1 , it is considered as some constant Q

* T(n) = Q+ C ( log2 n)

We can ignore constants as they are insignificant when n is large

Total time complexity T(n) = log2 n + 2n -2

By ignoring smaller term we have ,

Time complexity **T(n) = O(n)**

**Space Complexity :**

Since this involves first sorting the algorithm using inorder traversal, the space complexity of this traversal is = h (height of the tree) as max size of the recursive stack can be “h” at any point of time

Space taken to create another balanced binary search tree using the existing nodes = n

Total space complexity = h + n

Ignoring the smaller terms, we have

**Space complexity = O(n)**

**Question 3 :**

**3 a> Algorithm to check if two binary search trees are same:**

**Steps:**

1. Check if the root node of the two given trees have same data
2. If they are then recursively call the same function on left child of the root node. Compare left child node data of the trees
3. Now do the same on the right child of the root node. Compare their data. This is done until we reach leaf nodes.
4. If all of them return true then the two trees are same else they are not same

**Time Complexity :**

This algorithm involves visiting each node in the tree to compare their data through recursive calls. In order to check if two trees are equal, we are traversing the tree in preorder. We can think of it as edge traversals, during preorder traversal each node is visited exactly twice.

Number of edges in the binary search tree with n nodes = (n -1)

maximum time taken = 2(n -1) = 2n- 2

* T(n) = 2n-2

By ignoring small terms, we have

**Time complexity T(n) = O(n)**

Since this involves pre-order traversal, the space complexity = h (height of the tree) as max size of the recursion stack can be h at any point of time

**Space Complexity = O(n)**

**3 b> Algorithm to create a copy of binary search tree:**

**Steps:**

1. Create an empty binary search tree to create a copy
2. Traverse the nodes of the given tree in pre-order. i.e first insert the root node to the new BST .
3. Then recursively call the same function on left node and then on right node.
4. This ensures the cloned copy created has the nodes in the same order and structure as the given BST.

**Time Complexity :**

This algorithm involves visiting each node in the tree to add the node to the new BST through recursive calls. we are traversing the tree in preorder. We can think of it as edge traversals, during preorder traversal each node is visited exactly twice.

Number of edges in the binary search tree with n nodes = (n -1)

maximum time taken = 2(n -1) = 2n- 2

* T(n) = 2n-2

By ignoring small terms, we have

**Time complexity T(n) = O(n)**

Since this involves pre-order traversal, the space complexity = h (height of the tree) as max size of the recursion stack can be h at any point of time

**Q4>** We are having two implementations of the binary search tree. In the Composite implementation, we have a class for empty node i.e. EmptyNode and a different class for non-empty nodes i.e. ElementNode. The methods which are shared across these two implementations are defined in an interface called Node, which allows to have uniformity. This approach allows us to take the advantage of Java’s automatic dynamic dispatching to identify the type of node and call the respective method.

This automation is one of the advantages we can obtain using object oriented programming approach.

In the second approach, i.e. in single node approach we have just one type of node and in-order to differentiate between the empty and non-empty node, we write an extra piece of code which involves an if condition to check if node is null/empty and perform necessary action.

This approach is more like the one used in functional programming languages. This results in cluttering of code within the same function. The impact is more when we have many methods being defined.

The other difference between composite and single node implantation is that, we can traverse backwards using parent reference in case of composite, where-as in single node implementation the only option to traverse back is recursive stack frame.

In case of composite implementation, the numbers of methods increase with time whereas in case of single node implementation too many if cases may cause code cluttering.