Assignment - 12
Binary search trees

1. Algorithm to insert in a Binary search tree.

```
Algorithm Insert_BST( root, key)
{
    1. if root == NULL :
           create a newNode and return.

    2. Else find the correct position of the node to
       be inserted in the B.S.T. using traverse().

       position = traverse( root, key).

    3. if position → data > item.
           position → left = newNode.

    4. if position → data < item.
           position → right = newNode

}

Algorithm traverse (root, item)
{
    1. if root → data == item.
           return root.

    2. if root → data < item.
           if root → right exists
               then recursively call traverse() for right
                                    subtree of root
```

i.e.

```
if ( root → data < item )
    if (root → right)
        return traverse (root → right, item)
    else
        return root.
```

3. if root → data > item

```
if (root → left)
    return traverse( root → left, item)
else
    return root.
```

2. Algorithm to delete from B.S.T.

i. Find position to delete using traverse().

ii. If position == NULL print message saying Item not found in B.S.T.

iii. If node at position has no child node.
→ free node memory & return.

iv. If node at position has one child. then make the new-position pointing to the child node & delete the original node.

1. Node *child = position->left ? position->left : position->right;
2. position = child free(position)
3. free child position = child.

4. if node to be deleted has 2 children.

1. Find inorder successor(minimum in right subtree)
2. Copy inorder successor data into root.
3. Delete the inorder successor

Algorithm to print leaf & non-leaf nodes.

i. Declare 2 arrays of node pointers.
Node* terminals[20];
Node * non-terminals[20];

ii. Traverse through the tree.

iii. If a node has no child :
Add it to 'terminals' array.

Else.
Add it to 'non-terminals' array.

4. Algorithm to print nodes at each level.

I. Initialise queue of node pointers.

```
Node * queue[100];
front = rear = 0;
```

II. Add root to the queue.

```
queue[rear++] = root.
```

III. while (front < rear)

1. dequeue first element

```
current = queue[front++]
```

2. If 'current' has child nodes enqueue them.

5. Conclusion:

B.S.T. provide efficient searching, insertion and deletion operations while maintaining sorted order. The implement algorithms successfully demonstrate hierarchial data organisation and recursive tree processing.

Assignment - 13
Heap trees.

Algorithm to insert element into heap and move to appropriate position.

Algorithm insert_heap (heap, size, element)

i. If heap is full, return error.
ii. Place new element at the end of heap array.
iii. Set current_index = last position.
iv. while (current_index > 0):
    Find parent_index = (current_index - 1) / 2
    if heap[current_index] > heap[parent_index]
    Swap elements at current_index & parent_index
    curr_index = parent_index.
Else.
    break loop.

Algorithm to delete Root element & Re-heap the tree.

Algorithm delete_root (heap, size)
i. If heap is empty, return error.
ii. Store root_element = heap[0].
iii. Replace root with last element
    heap[0] = heap[size - 1].

iv. Set curr_index = 0.

v. While curr-index has children:
find left-child = 2 * curr-index + 1
find right-child = 2* curr-index +2.
Set largest = curr-index.

if (left-child exists and is greater than curr-index)
→ largest = left-child.
if (right-child exists & is greater than curr-
→ largest = right-child.

if (largest ≠ curr-index)
Swap heap[curr-index] & heap[largest]
curr-index = largest.

else
break loop

3. CONCLUSION:
Heap operations efficiently manage/maintain the heap property using bubble-up & trickle-down techniques. The algorithms ensure optimal performance for priority queue operations.