LAB - 11

Name : Vraj S. Shah
Roll_No : CE115
ID : 21CEUOF071

1. HEAP SORT

```
#include<iostream>

#include<bits/stdc++.h>

using namespace std;

void heapify(int arr[], int n, int i)
{ int largest = i;
   int l = 2*i + 1;
   int r = 2*i + 2;

   //If left child is larger than root
   if (l < n && arr[l] > arr[largest])
      largest = l;

   //If right child largest if (r < n
   && arr[r] > arr[largest])
      largest = r;

   //If root is nor largest
   if (largest != i)
   { swap(arr[i], arr[largest]);

      //Recursively heapifying the sub-
      tree
      heapify(arr, n, largest);
   }
}
```

```
void heapSort(int arr[], int n)
{

   for (int i = n / 2 - 1; i >= 0; i--)
       heapify(arr, n, i);

   //One by one extract an element from heap
   for (int i=n-1; i>=0; i--)
   {
       //Moving current root to end
       swap(arr[0], arr[i]);

       //Calling max heapify on the reduced heap
       heapify(arr, i, 0);
   }
}

//Function to print array
void display(int arr[], int n)
{ for (int i = 0; i < n; i++)
   { cout << arr[i] << "\t”;
   } cout <<
   "\n";
}


int main()
{ int arr[] = {1, 14, 3, 7, 0}; int n =
   sizeof(arr)/sizeof(arr[0]); cout
   << "Unsorted array  \n";
   display(arr, n);
   heapSort(arr, n);

   cout << "Sorted array  \n";
   display(arr, n);
}
```

2. BST

```
#include<bits/stdc++.h>
Using namespace std;

class Node{
public: int
data; Node*
left;
   Node* right;
   Node(int d){
      data = d; left
      = NULL; right
      = NULL;
   }
}

Node* search(Node* root, int key) {
      if(root == NULL || root->data == key)
         return root;

      // Key is greater than root's data
      if(root->data < key)
         return search(root->right,key);

      // Key is smaller than root's data
      return search(root->left,key);
   }
   Node* insert(Node* root, int data) {
      if(root == NULL){
         return new Node(data);
      }
   else{ Node* cur; if(data <=
        root->data) {
            cur = insert(root->left, data);
            root->left = cur;
        }
        else
        {
            cur = insert(root->right, data);
            root->right = cur;
        } return
    root;
}
}

Node* deletenode(Node* root, int k)
{
    // Base case if
    (root == NULL)
        return root;
    //If root->data is greater than k then we delete the root's
    subtree if(root->data > k){
        root->left = deletenode(root->left, k);
        return root;
    } else if(root->data <
    k){
        root->right = deletenode(root->right, k);
        return root;
    }


    // If one of the children is
    empty if (root->left == NULL) {
    Node* temp = root->right;
    delete root;
        return temp;
    }
    else if (root->right == NULL)
        { Node* temp = root->left;
        delete root; return temp;
    }

else {
         Node* Parent = root;
```

```
      // Find successor of the
      node Node *succ = root-
      >right; while (succ->left !=
      NULL) { Parent = succ; succ
      = succ->left;
      }

      if (Parent != root)
        Parent->left = succ->right;
      else
        Parent->right = succ->right;

      // Copy Successor Data root-
      >data = succ->data;

      // Delete Successor and return root
      delete succ; return root;
    }
  }
```

3. AVL

```
#include<iostream>
#include<cstdio>
#include<sstream>
#include<algorithm>
#define pow2(n) (1 << (n))
using namespace std;
struct avl { int d; struct avl
*l; struct avl *r;
}*r; class
avl_tree {
  public:
    int height(avl *); int
    difference(avl *); avl
    *rr_rotat(avl *); avl
    *ll_rotat(avl *); avl
    *lr_rotat(avl*); avl
    *rl_rotat(avl *); avl *
    balance(avl *); avl *
    insert(avl*, int); void
    show(avl*, int); void
    inorder(avl *); void
    preorder(avl *); void
    postorder(avl*);
    avl_tree() {
      r = NULL;
    }
};
int avl_tree::height(avl *t) {
  int h = 0; if (t
  != NULL) {
    int l_height = height(t->l); int r_height =
    height(t->r); int max_height =
    max(l_height, r_height); h = max_height
    + 1;
  } return h; } int
avl_tree::difference(avl *t) { int
l_height = height(t->l); int r_height
= height(t->r); int b_factor =
l_height - r_height; return
b_factor;
} avl *avl_tree::rr_rotat(avl
*parent) {
  avl *t; t = parent->r; parent->r =
t->l; t->l = parent; cout<<"Right-
Right Rotation"; return t; } avl
*avl_tree::ll_rotat(avl *parent) {
  avl *t; t = parent->l; parent->l =
t->r; t->r = parent; cout<<"Left-Left
Rotation"; return t; } avl
*avl_tree::lr_rotat(avl *parent) {
  avl *t; t = parent->l; parent-
  >l = rr_rotat(t); cout<<"Left-
  Right Rotation"; return
  ll_rotat(parent);
} avl *avl_tree::rl_rotat(avl *parent)
{
  avl *t; t = parent->r; parent-
  >r = ll_rotat(t);
  cout<<"Right-Left
  Rotation"; return
  rr_rotat(parent);
} avl *avl_tree::balance(avl *t)
{
  int bal_factor =
  difference(t); if (bal_factor >
  1) { if (difference(t->l) > 0)
      t = ll_rotat(t);
    else
      t = lr_rotat(t);
  } else if (bal_factor < -1) {
    if (difference(t->r) > 0)
      t = rl_rotat(t);
    else
      t = rr_rotat(t); } return t; }
avl *avl_tree::insert(avl *r, int v) {
  if (r == NULL)
    { r = new avl;
    r->d = v; r->l
    = NULL; r->r
    = NULL;
    return r;
  } else if (v< r->d) { r-
    >l = insert(r->l, v); r
    = balance(r);
  } else if (v >= r->d) { r-
    >r = insert(r->r, v); r
    = balance(r);
  } return r; } void
avl_tree::show(avl *p, int l) { int i;
  if (p != NULL) {
    show(p->r, l+
    1); cout<<" "; if
    (p == r)
      cout << "Root -> ";
    for (i = 0; i < l && p != r; i++)
      cout << " "; cout
      << p->d;
      show(p->l, l +
      1);
  }
} void avl_tree::inorder(avl *t)
{
  if (t == NULL) return;
    inorder(t->l); cout
    << t->d << " ";
    inorder(t->r);
} void avl_tree::preorder(avl *t)
{
  if (t == NULL)
    return;
    cout << t->d << " ";
    preorder(t->l); preorder(t-
    >r);
} void avl_tree::postorder(avl *t)
{
  if (t == NULL)
    return; postorder(t
    ->l); postorder(t -
    >r); cout << t->d
    << " ";
} int main()
{ int c, i;
  avl_tree avl;
  while (1) {
    cout << "1.Insert Element into the tree" <<
    endl; cout << "2.show Balanced AVL Tree" <<
    endl; cout << "3.InOrder traversal" << endl;
    cout << "4.PreOrder traversal" << endl; cout <<
    "5.PostOrder traversal" << endl; cout <<
    "6.Exit" << endl; cout << "Enter your Choice: ";
    cin >> c; switch (c) {
      case 1: cout << "Enter value to be
      inserted: "; cin >> i; r = avl.insert(r, i);
      break; case 2:
        if (r == NULL) {
          cout << "Tree is Empty" << endl;
          continue;
        }
        cout << "Balanced AVL Tree:" <<
        endl; avl.show(r, 1); cout<<endl;
      break;
      case 3:
        cout << "Inorder Traversal:" <<
        endl; avl.inorder(r); cout << endl;
      break;
      case 4:
        cout << "Preorder Traversal:" <<
        endl; avl.preorder(r); cout << endl;
      break;
      case 5:
        cout << "Postorder Traversal:" <<
        endl; avl.postorder(r); cout << endl;
      break;
      case 6:
      exit(1);
      break;
      default:
        cout << "Wrong Choice" << endl;
    }
  } return
  0;
}
```