11) MERGE TWO BST:

METHOD 1:
In this method, we know the inorder traversal of the two BST will be in the sorted order.So we first create the two arrays which are the inorder traversal of the two trees. Then we create the third array whose size is (n+m) and merge the two arrays.
Then we create the balanced BST from this sorted array like we did in the previous sum.
Space Complexity : O(n+m)
Time Complexity : O(n+m)

METHOD 2:
In this method, we just insert all the elements of the second tree into the first tree.
Time Complexity : O(N log M) or O(M log N)
Space Complexity : O(1)

METHOD 3:
USING DOUBLY LINKED LIST.
This method is very similar to the first method.Here first we convert both the BST into the Doubly linked list without extra space.
Then we merge both the linked list like we did in array to generate one sorted DLL and then we convert it into BST by using the same method we did for the sorted aray one. Here the mid is found using the slow and fast pointer approach.
Link of Explanation : Convert Sorted Doubly LinkedList to Balanced Binary Search Tree | Using…

CODE OF METHOD 1:(MAIN FUNCTIONS ARE HIGHLIGHTED IN RED BOLD)

```
#include<iostream>
#include<string>
#include<vector>
#include<queue>
using namespace std;

class node{
   public:
   int data;
   node *left=NULL;
   node *right=NULL;
};

class BinarySearchTree{
   public:
   node *head=NULL;
   public:
   BinarySearchTree(vector<string> &initialisation){
      queue<node*> q;
      node *root=new node;
      root->data=stoi(initialisation[0]);
      q.push(root);
```

```
        int k=1;
        this->head=root;
        while(q.size()!=0 && k+1<initialisation.size()){
            if(initialisation[k]=="N" && initialisation[k+1]=="N"){
                q.pop();
            }
            else if(initialisation[k]=="N" && initialisation[k+1]!="N"){
                node *temp=new node;
                temp->data=stoi(initialisation[k+1]);
                q.front()->right=temp;
                q.push(temp);
                q.pop();
            }
            else if(initialisation[k]!="N" && initialisation[k+1]=="N"){
                node *temp=new node;
                temp->data=stoi(initialisation[k]);
                q.front()->left=temp;
                q.push(temp);
                q.pop();
            }
            else{
                node *temp1=new node;
                node *temp2=new node;
                temp1->data=stoi(initialisation[k]);
                temp2->data=stoi(initialisation[k+1]);
                q.front()->left=temp1;
                q.front()->right=temp2;
                q.push(temp1);
                q.push(temp2);
                q.pop();
            }
            k=k+2;
        }
        if(k+1==initialisation.size() && q.size()!=0 && initialisation[k]!="N"){
            node *temp=new node;
            temp->data=stoi(initialisation[k]);
            q.front()->left=temp;
        }
    }
    void InOrderTraversal(node *root,vector<int> &v){
        if(root!=NULL){
            InOrderTraversal(root->left,v);
            v.push_back(root->data);
            InOrderTraversal(root->right,v);
        }
    }
    friend int main();
};

node *BalancedBST(vector<int> &v,int low,int high){
```

```
    if(low>high){
        return NULL;
    }
    else{
        int mid=(high+low)/2;
        node *t=new node;
        t->data=v[mid];
        t->left=BalancedBST(v,low,mid-1);
        t->right=BalancedBST(v,mid+1,high);
        return t;
    }
}

void peek(node *root){
    if(root!=NULL){
        peek(root->left);
        cout<<root->data<<" ";
        peek(root->right);
    }
}

node *function(vector<int> &v){
    int n=v.size();
    node *root=BalancedBST(v,0,n-1);
    return root;
}

vector<int> Merge(vector<int> &v1,vector<int> &v2){
    vector<int> v3(v1.size()+v2.size());
    int i=0,j=0,k=0;
    while(i<v1.size() && j<v2.size()){
        if(v1[i]<=v2[j]){
            v3[k]=v1[i];
            i++;
            k++;
        }
        else{
            v3[k]=v2[j];
            j++;
            k++;
        }
    }
    while(i<v1.size()){
        v3[k]=v1[i];
        i++;
        k++;
    }
    while(j<v2.size()){
        v3[k]=v2[j];
        j++;
```

```
        k++;
    }
    return v3;
}

int main(){
    int n,m;
    cout<<"\n Enter the number of nodes present in the first Binary Search Tree:";
    cin>>n;
    vector<string> initialisation1(n);
    cout<<"\n Enter the node values of the first Binary Search Tree:";
    for(int i=0;i<n;i++){
        cin>>initialisation1[i];
    }
    cout<<"\n Enter the number of nodes present in the second Binary Search Tree:";
    cin>>m;
    vector<string> initialisation2(m);
    cout<<"\n Enter the node values of the second Binary Search Tree:";
    for(int i=0;i<m;i++){
        cin>>initialisation2[i];
    }
    BinarySearchTree bst1(initialisation1);
    BinarySearchTree bst2(initialisation2);

    vector<int> v1,v2;
    bst1.InOrderTraversal(bst1.head,v1);
    bst2.InOrderTraversal(bst2.head,v2);
    vector<int> v3=Merge(v1,v2);
    node *root=function(v3);
    cout<<"\n The Inorder Traversal of the Merged BST's :";
    peek(root);
    return 0;
}
```

CODE OF METHOD 2:(IMPORTANT FUNCTIONS ARE HIGHLIGHTED IN RED BOLD)

```
#include<iostream>
#include<string>
#include<vector>
#include<queue>
using namespace std;

class node{
    public:
    int data;
    node *left=NULL;
    node *right=NULL;
```

```
};

class BinarySearchTree{
   public:
   node *head=NULL;
   public:
   BinarySearchTree(vector<string> &initialisation){
      queue<node*> q;
      node *root=new node;
      root->data=stoi(initialisation[0]);
      q.push(root);
      int k=1;
      this->head=root;
      while(q.size()!=0 && k+1<initialisation.size()){
         if(initialisation[k]=="N" && initialisation[k+1]=="N"){
            q.pop();
         }
         else if(initialisation[k]=="N" && initialisation[k+1]!="N"){
            node *temp=new node;
            temp->data=stoi(initialisation[k+1]);
            q.front()->right=temp;
            q.push(temp);
            q.pop();
         }
         else if(initialisation[k]!="N" && initialisation[k+1]=="N"){
            node *temp=new node;
            temp->data=stoi(initialisation[k]);
            q.front()->left=temp;
            q.push(temp);
            q.pop();
         }
         else{
            node *temp1=new node;
            node *temp2=new node;
            temp1->data=stoi(initialisation[k]);
            temp2->data=stoi(initialisation[k+1]);
            q.front()->left=temp1;
            q.front()->right=temp2;
            q.push(temp1);
            q.push(temp2);
            q.pop();
         }
         k=k+2;
      }
      if(k+1==initialisation.size() && q.size()!=0 && initialisation[k]!="N"){
         node *temp=new node;
         temp->data=stoi(initialisation[k]);
         q.front()->left=temp;
      }
   }
```

```
    void InOrderTraversal(node *root){
        if(root!=NULL){
            InOrderTraversal(root->left);
            cout<<root->data<<" ";
            InOrderTraversal(root->right);
        }
    }
    void Insert(node *head1,node *current){
        if(head1->data<current->data){
            if(head1->right==NULL){
                head1->right=current;
            }
            else{
                Insert(head1->right,current);
            }
        }
        else{
            if(head1->left==NULL){
                head1->left=current;
            }
            else{
                Insert(head1->left,current);
            }
        }
    }
    void Insertion(node *head1,node *head2){
        if(head2!=NULL){
            Insertion(head1,head2->left);
            head2->left=NULL;
            node *temp=head2->right;
            head2->right=NULL;
            Insert(head1,head2);
            Insertion(head1,temp);
        }
    }
    friend int main();
};

int main(){
    int n,m;
    cout<<"\n Enter the number of nodes present in the first Binary Search Tree:";
    cin>>n;
    vector<string> initialisation1(n);
    cout<<"\n Enter the node values of the first Binary Search Tree:";
    for(int i=0;i<n;i++){
        cin>>initialisation1[i];
    }
    cout<<"\n Enter the number of nodes present in the second Binary Search Tree:";
    cin>>m;
    vector<string> initialisation2(m);
```

```
    cout<<"\n Enter the node values of the second Binary Search Tree:";
    for(int i=0;i<m;i++){
        cin>>initialisation2[i];
    }
    BinarySearchTree bst1(initialisation1);
    BinarySearchTree bst2(initialisation2);

    bst1.Insertion(bst1.head,bst2.head);

    cout<<"\n The InOrder Traversal of the Merged BST's : ";
    bst1.InOrderTraversal(bst1.head);
    return 0;
}
```

METHOD 3:(IMPORTANT FUNCTIONS ARE HIGHLIGHTED IN RED BOLD)

(BELOW CODE HAS SOME SMALL ERROR)

```
#include<iostream>
#include<string>
#include<vector>
#include<queue>
using namespace std;

class node{
    public:
    int data;
    node *left=NULL;
    node *right=NULL;
};

class BinarySearchTree{
    public:
    node *head=NULL;
    node *previous=NULL;
    node *start=NULL;

    public:
    BinarySearchTree(vector<string> &initialisation){
        queue<node*> q;
        node *root=new node;
        root->data=stoi(initialisation[0]);
        q.push(root);
        int k=1;
        this->head=root;
        while(q.size()!=0 && k+1<initialisation.size()){
            if(initialisation[k]=="N" && initialisation[k+1]=="N"){
                q.pop();
            }
```

```
        else if(initialisation[k]=="N" && initialisation[k+1]!="N"){
            node *temp=new node;
            temp->data=stoi(initialisation[k+1]);
            q.front()->right=temp;
            q.push(temp);
            q.pop();
        }
        else if(initialisation[k]!="N" && initialisation[k+1]=="N"){
            node *temp=new node;
            temp->data=stoi(initialisation[k]);
            q.front()->left=temp;
            q.push(temp);
            q.pop();
        }
        else{
            node *temp1=new node;
            node *temp2=new node;
            temp1->data=stoi(initialisation[k]);
            temp2->data=stoi(initialisation[k+1]);
            q.front()->left=temp1;
            q.front()->right=temp2;
            q.push(temp1);
            q.push(temp2);
            q.pop();
        }
        k=k+2;
    }
    if(k+1==initialisation.size() && q.size()!=0 && initialisation[k]!="N"){
        node *temp=new node;
        temp->data=stoi(initialisation[k]);
        q.front()->left=temp;
    }
}
void convert(node *head){
    if(head!=NULL){
        convert(head->left);
        if(previous==NULL){
            head->left=previous;
            previous=head;
            start=head;
        }
        else{
            previous->right=head;
            head->left=previous;
            previous=head;
        }
        convert(head->right);
    }
}
void rightTraversal(){
```

```
        node *traverse=start;
        while(traverse!=NULL){
            cout<<traverse->data<<" ";
            traverse=traverse->right;
        }
    }
    void leftTraversal(){
        node *traverse=previous;
        while(traverse!=NULL){
            cout<<traverse->data<<" ";
            traverse=traverse->left;
        }
    }
    friend int main();
    friend node *Merge(node *start1,node *start2);
    friend void Traversal(node *start);
};

void Traversal(node *start){
    node *traverse=start;
    while(traverse!=NULL){
        cout<<traverse->data<<" ";
        traverse=traverse->right;
    }
}

void InOrderTraversal(node *root){
    if(root!=NULL){
        InOrderTraversal(root->left);
        cout<<root->data<<" ";
        InOrderTraversal(root->right);
    }
}

node* Merge(node *start1,node *start2){
    node *result;
    if(start1->data<start2->data){
        result=start1;
    }
    else{
        result=start2;
    }
    node *i=start1;
    node *j=start2;
    node *last;
    while(i!=NULL && j!=NULL){
        if(i->data<j->data){
            last=i;
            i=i->right;
        }
```

```
      else{
         node *temp=j->right;
         if(i->left!=NULL){
            i->left->right=j;
         }
         j->right=i;
         j->left=i->left;
         i->left=j;
         j=temp;
      }
   }
   if(j!=NULL){
      last->right=j;
      j->left=last;
   }
   return result;
}

node* convert(node *start){
   if(start==NULL){
      return NULL;
   }
   else if(start->right==NULL){
      return start;
   }
   else{
      node *slow=start;
      node *fast=start;
      while(fast->right!=NULL && fast->right->right!=NULL){
         slow=slow->right;
         fast=fast->right->right;
      }
      node *previous=slow->left;
      node *next=slow->right;
      if(previous!=NULL){
         previous->right=NULL;
      }
      if(next!=NULL){
         next->left==NULL;
      }
      slow->left=convert(start);
      slow->right=convert(next);
      return slow;
   }
}

int main(){
   int n,m;
   cout<<"\n Enter the number of nodes present in the first Binary Search Tree:";
   cin>>n;
```

```
    vector<string> initialisation1(n);
    cout<<"\n Enter the node values of the first Binary Search Tree:";
    for(int i=0;i<n;i++){
        cin>>initialisation1[i];
    }
    cout<<"\n Enter the number of nodes present in the second Binary Search Tree:";
    cin>>m;
    vector<string> initialisation2(m);
    cout<<"\n Enter the node values of the second Binary Search Tree:";
    for(int i=0;i<m;i++){
        cin>>initialisation2[i];
    }
    BinarySearchTree bst1(initialisation1);
    BinarySearchTree bst2(initialisation2);

    bst1.convert(bst1.head);
    bst2.convert(bst2.head);

    //At this point the binary tree is completely destroyed. So do not try to access the Binary
Search Trees.

    node *start=Merge(bst1.start,bst2.start);
    bst1.start={bst2.start=start};
    node *root=convert(start);
    cout<<"\n";
    InOrderTraversal(root);
    return 0;
}
```