HYU HANYANG UNIVERSITY DIVISION OF COMPUTER SCIENCE

Data Structures

박영준 교수님

Lab6: AVL

AVL

- Self-balancing binary search tree.
- For every node in the tree, the heights of its subtrees differ by at most 1

AVL ADT

AVL

- int RetHeight(BTNode *Node);
  - Return height of both sub tree
- int RetDiffInHeightOfSubTree(BTNode *Node);
  - Return difference of both sub tree

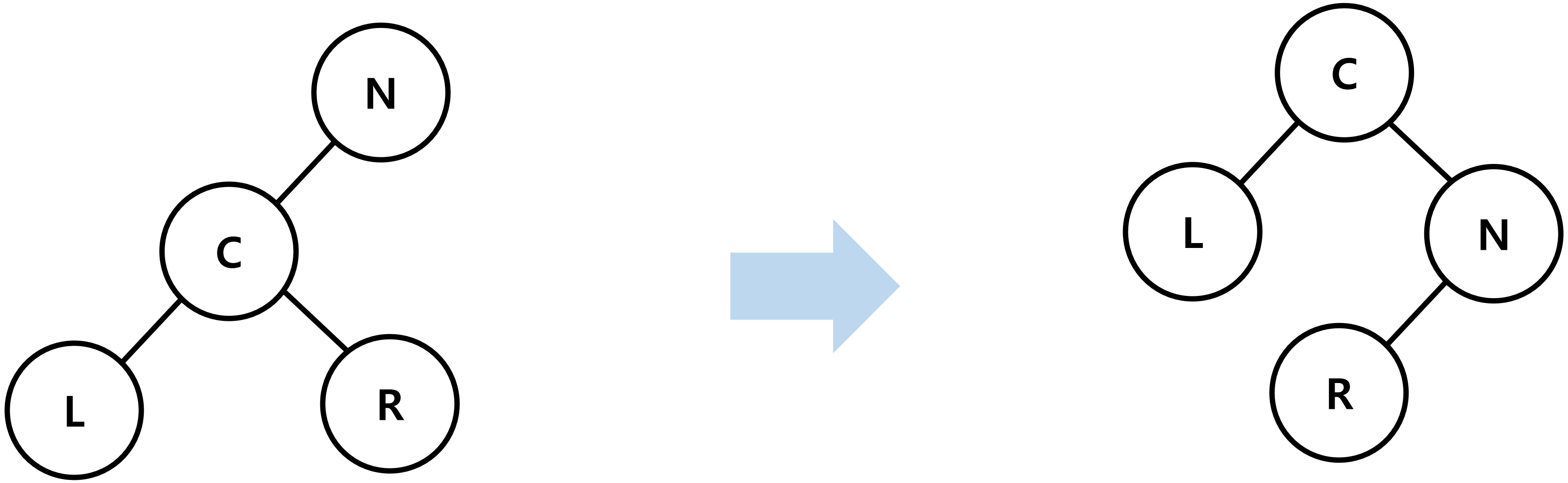
AVL ADT

AVL

- BTNode *RotateLL(BTNode *Node);
  - Balancing LL state of Node
  - Make left child(Child) of node to be parent of Node
  - And Node to be right child of Child
  - If Child had right child, make it to be left child of Node

AVL

LL Rotation



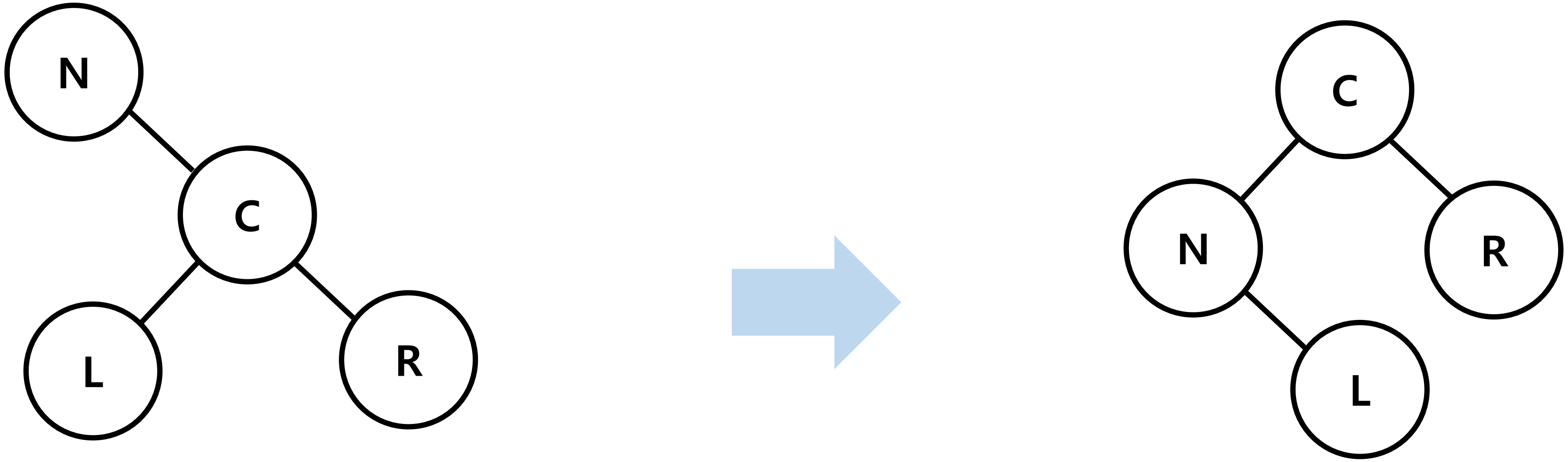
AVL ADT

AVL

- BTNode *RotateRR(BTNode *Node);
  - Balancing RR state of Node
  - Make right child(Child) of node to be parent of Node
  - And Node to be left child of Child
  - If Child had left child, make it to be right child of Node

AVL

RR Rotation



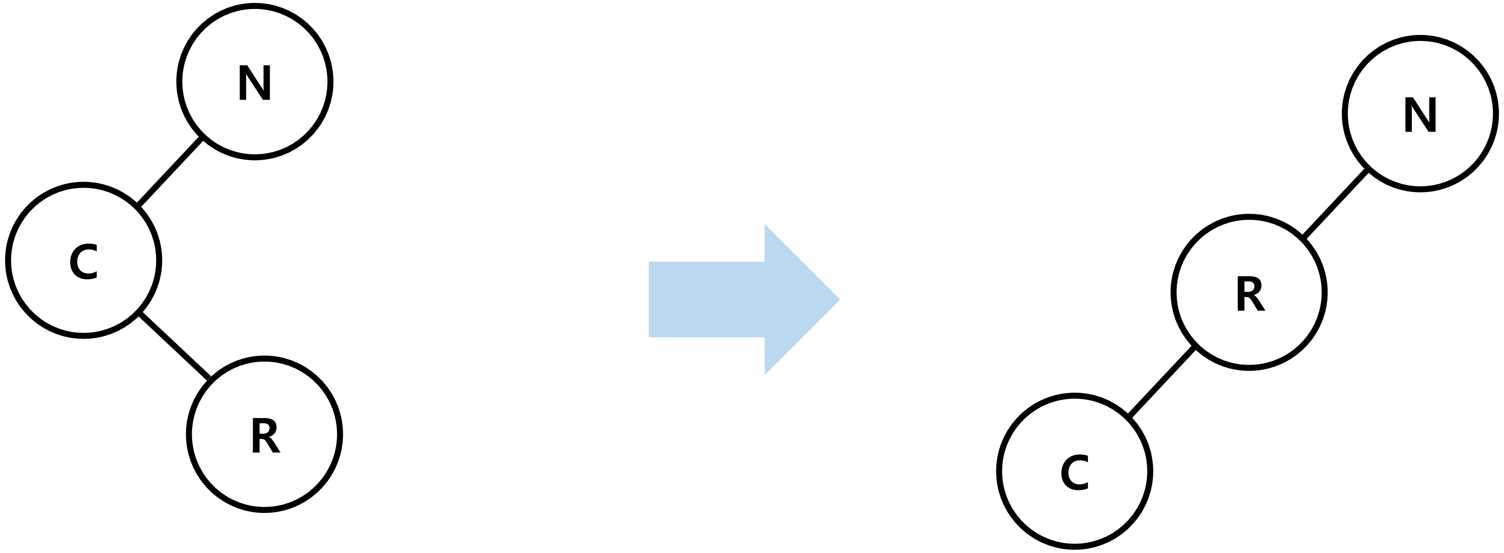
AVL ADT

AVL

- BTNode *RotateLR(BTNode *Node);
    - Balancing LR state of Node
    - First, RR rotate left child(Child) of Node with right child of Child
    - And LL rotate Node

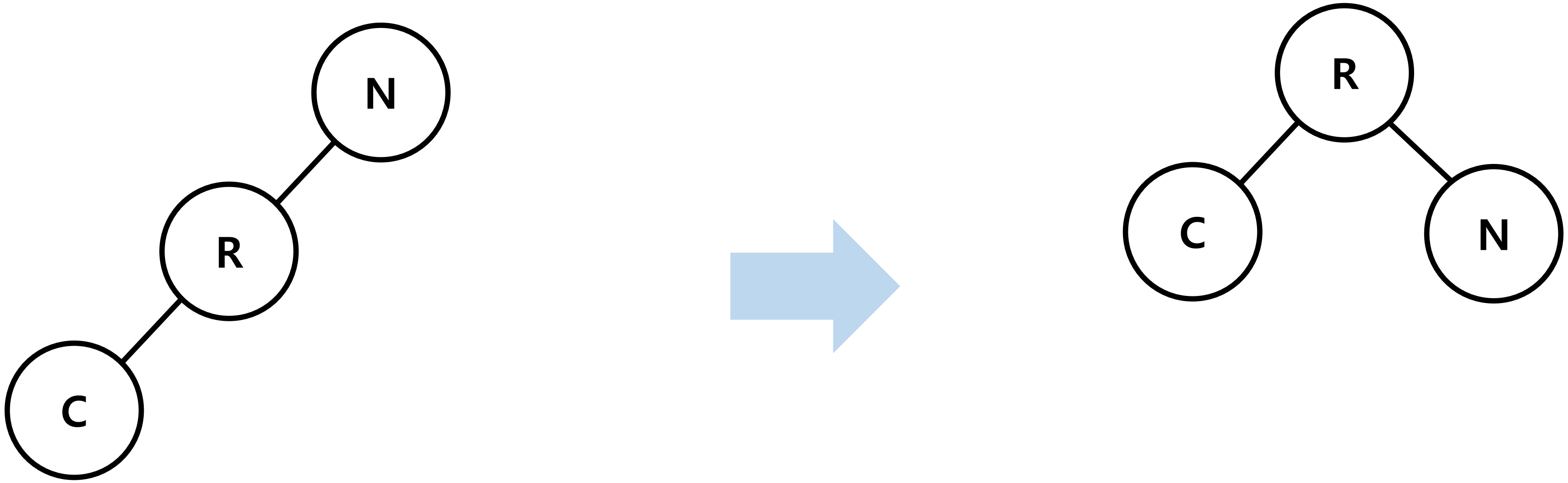
AVL

LR Rotation



AVL

LR Rotation



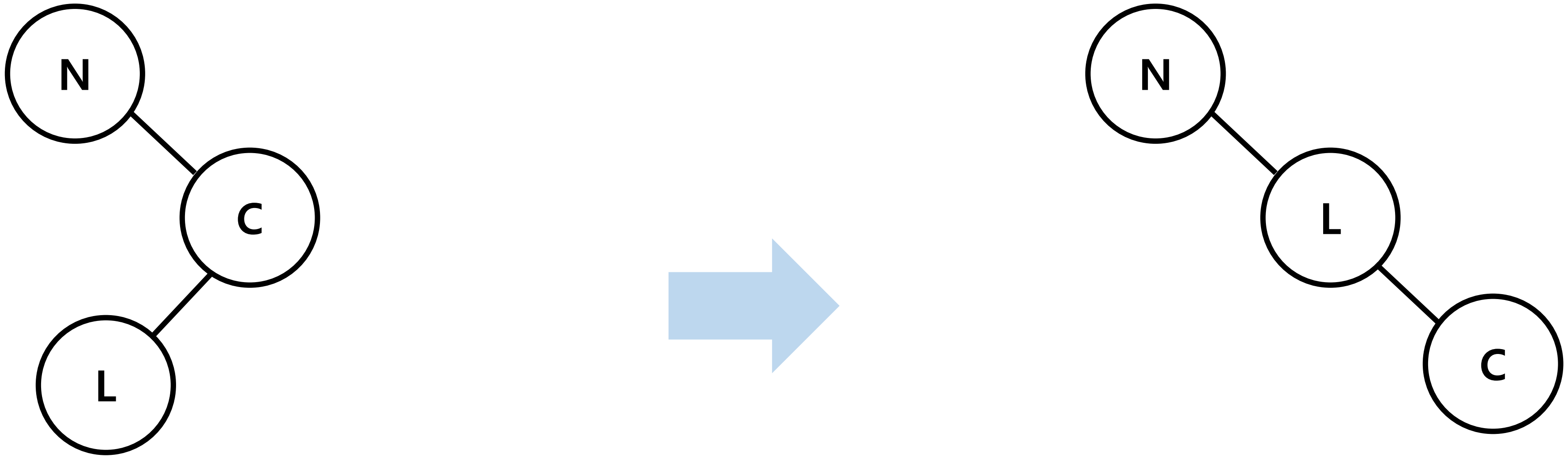
AVL ADT

AVL

- BTNode *RotateRL(BTNode *Node);
  - Balancing RL state of Node
  - First, LL rotate right child(Child) of Node with left child of Child
  - And RR rotate Node

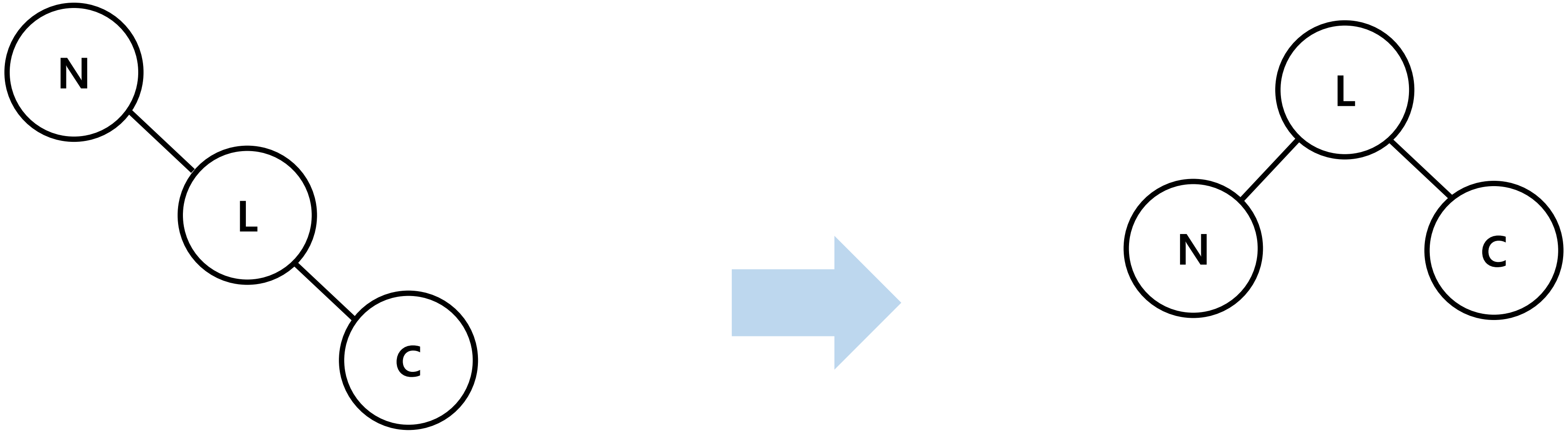
AVL

RL Rotation



AVL

RL Rotation

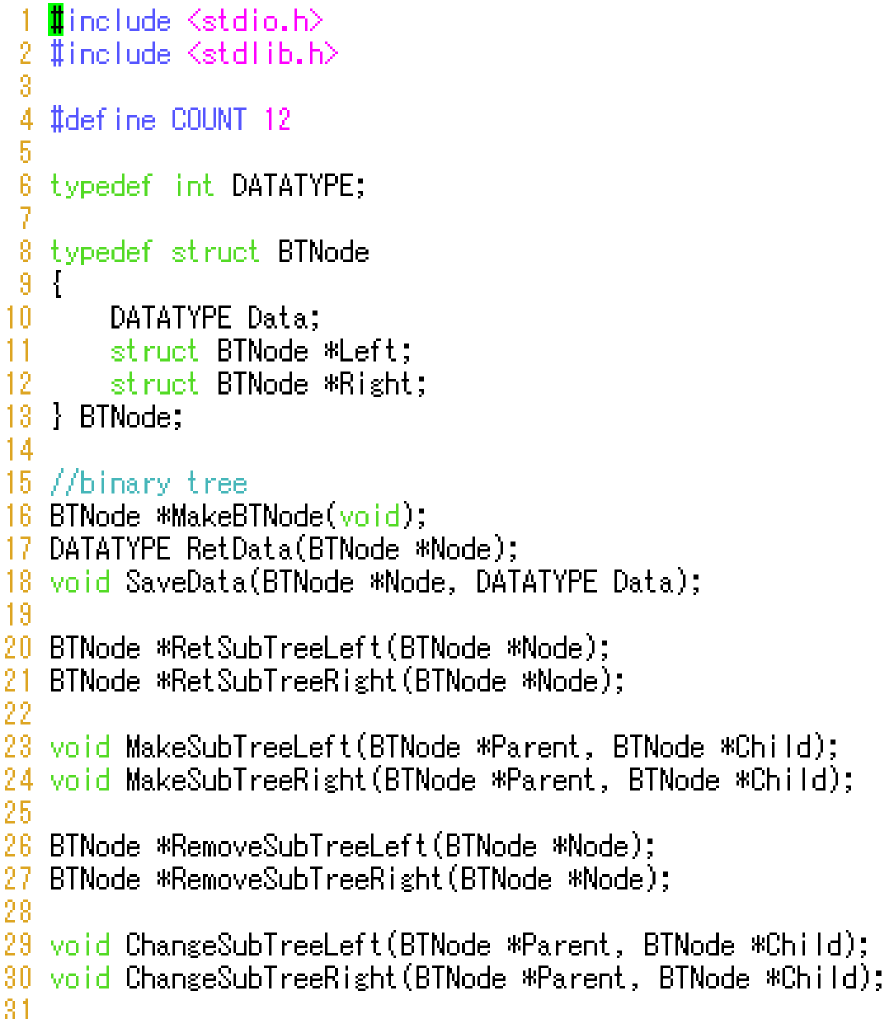


AVL ADT

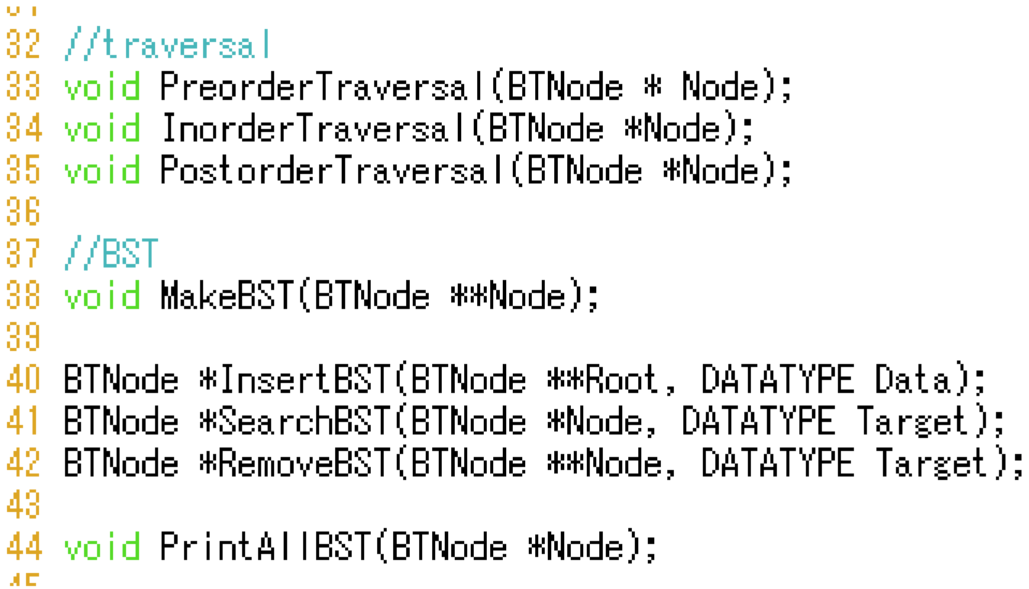
AVL

- BTNode *Rebalance(BTNode **Root);
  - Rotate BST if the difference between two subtree is higher than 2
- BTNode *InsertBST(BTNode **Root, DATATYPE Data);
  - Insert Data in Root
  - Rebalancing by comparing heights for each insert

AVL ADT

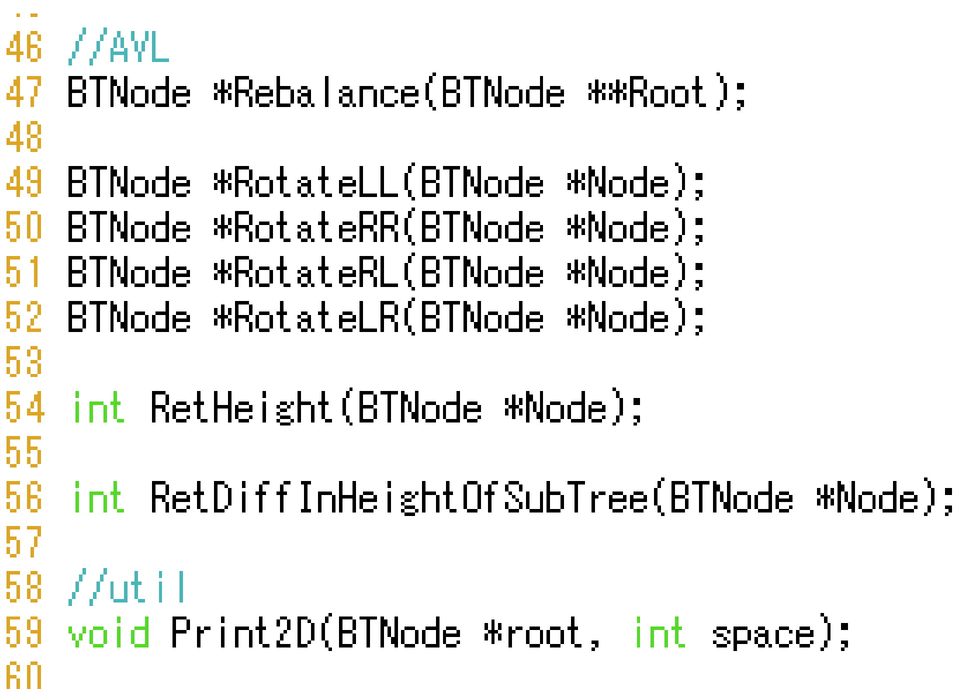


```
#include <stdio.h>
#include <stdlib.h>

#define COUNT 12

typedef int DATATYPE;

typedef struct BTNode
{
    DATATYPE Data;
    struct BTNode *Left;
    struct BTNode *Right;
} BTNode;

//binary tree
BTNode *MakeBTNode(void);
DATATYPE RetData(BTNode *Node);
void SaveData(BTNode *Node, DATATYPE Data);

BTNode *RetSubTreeLeft(BTNode *Node);
BTNode *RetSubTreeRight(BTNode *Node);

void MakeSubTreeLeft(BTNode *Parent, BTNode *Child);
void MakeSubTreeRight(BTNode *Parent, BTNode *Child);

BTNode *RemoveSubTreeLeft(BTNode *Node);
BTNode *RemoveSubTreeRight(BTNode *Node);

void ChangeSubTreeLeft(BTNode *Parent, BTNode *Child);
void ChangeSubTreeRight(BTNode *Parent, BTNode *Child);

```



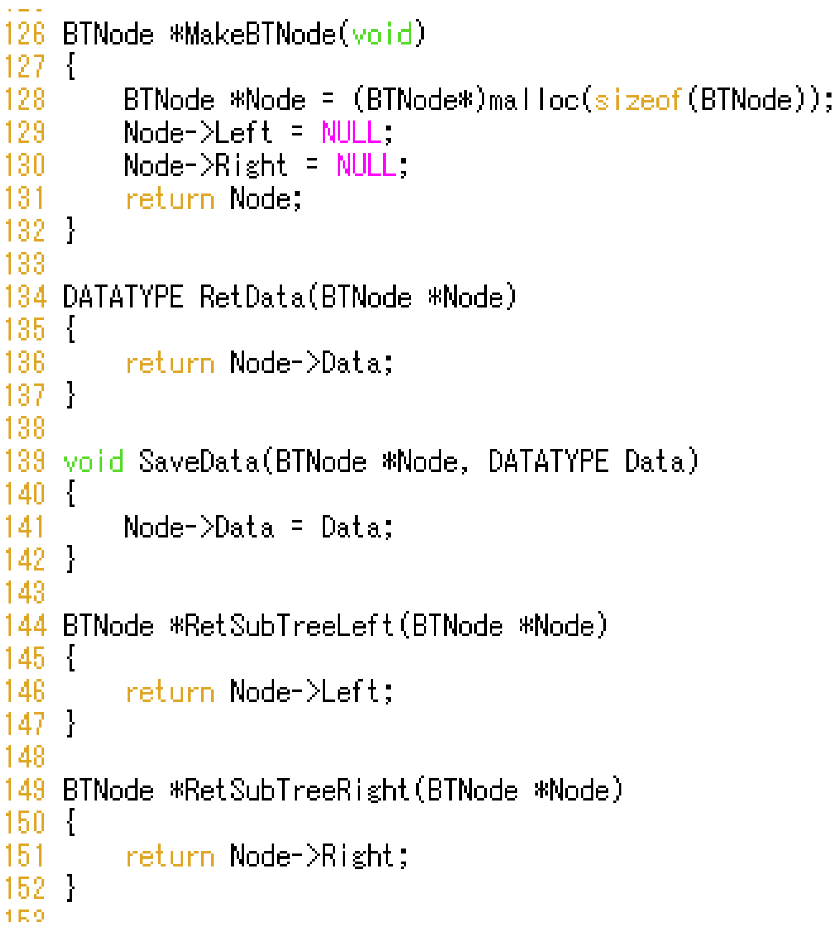
```
//traversal
void PreorderTraversal(BTNode * Node);
void InorderTraversal(BTNode *Node);
void PostorderTraversal(BTNode *Node);

//BST
void MakeBST(BTNode **Node);

BTNode *InsertBST(BTNode **Root, DATATYPE Data);
BTNode *SearchBST(BTNode *Node, DATATYPE Target);
BTNode *RemoveBST(BTNode **Node, DATATYPE Target);

void PrintAllBST(BTNode *Node);
```

AVL ADT

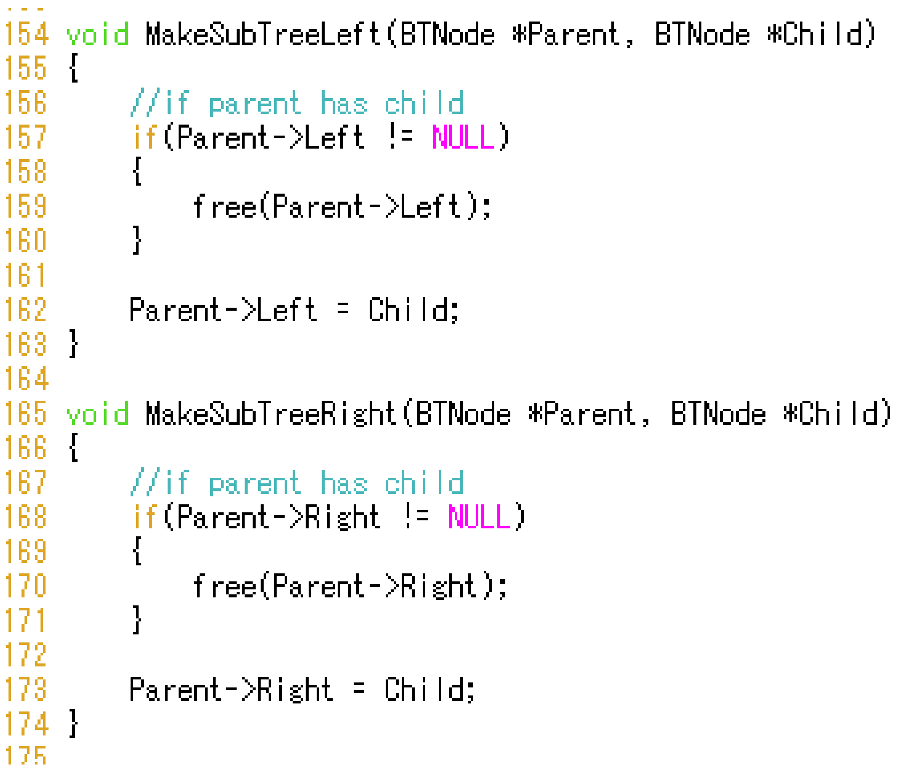


```c
//AVL
BTNode *Rebalance(BTNode **Root);

BTNode *RotateLL(BTNode *Node);
BTNode *RotateRR(BTNode *Node);
BTNode *RotateRL(BTNode *Node);
BTNode *RotateLR(BTNode *Node);

int RetHeight(BTNode *Node);

int RetDiffInHeightOfSubTree(BTNode *Node);

//util
void Print2D(BTNode *root, int space);
```

AVL ADT

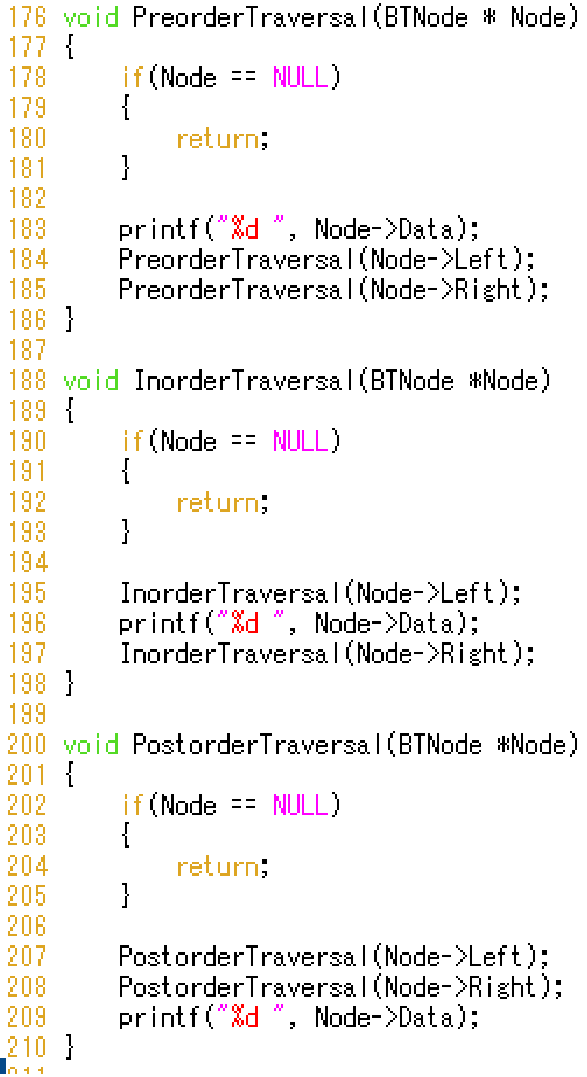


```c
BTNode *MakeBTNode(void)
{
    BTNode *Node = (BTNode*)malloc(sizeof(BTNode));
    Node->Left = NULL;
    Node->Right = NULL;
    return Node;
}

DATATYPE RetData(BTNode *Node)
{
    return Node->Data;
}

void SaveData(BTNode *Node, DATATYPE Data)
{
    Node->Data = Data;
}

BTNode *RetSubTreeLeft(BTNode *Node)
{
    return Node->Left;
}

BTNode *RetSubTreeRight(BTNode *Node)
{
    return Node->Right;
}
```

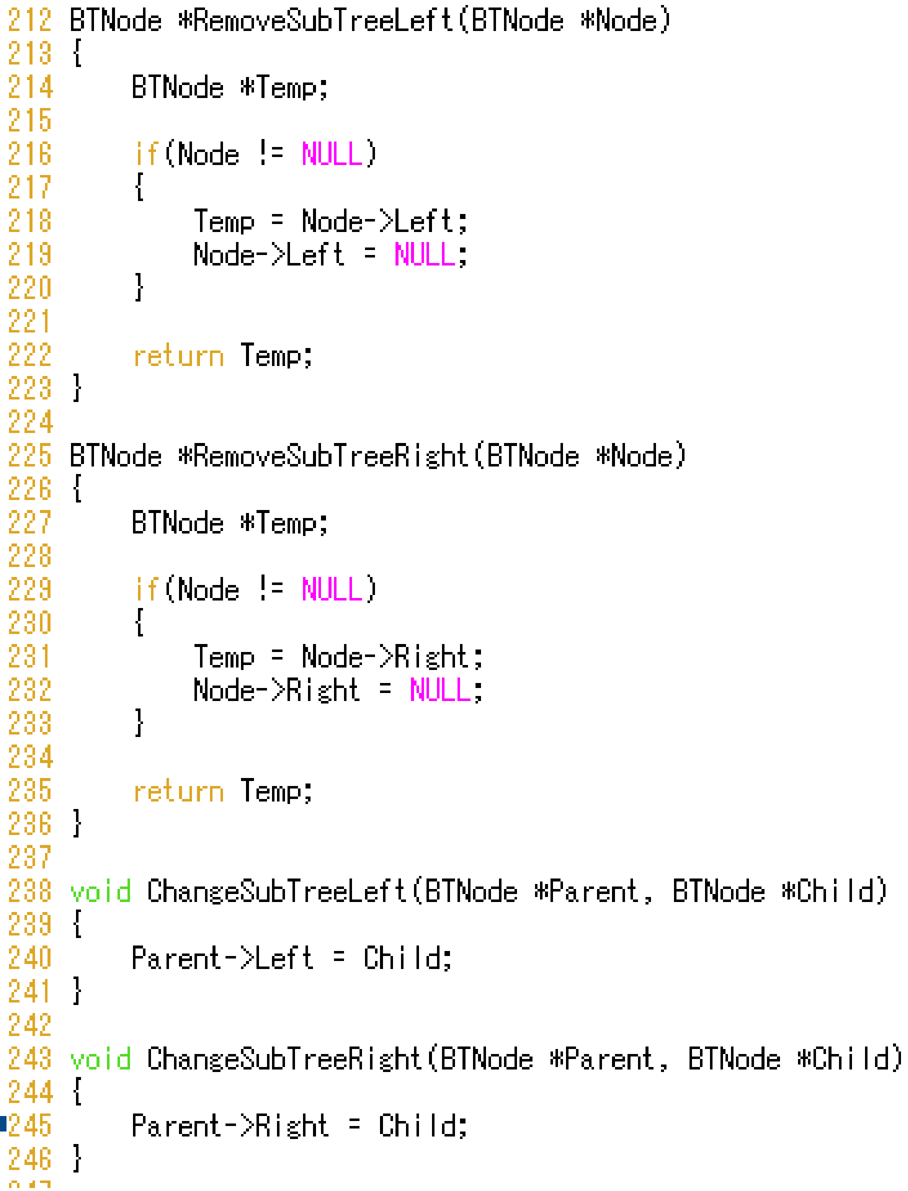


```c
void MakeSubTreeLeft(BTNode *Parent, BTNode *Child)
{
    //if parent has child
    if(Parent->Left != NULL)
    {
        free(Parent->Left);
    }

    Parent->Left = Child;
}

void MakeSubTreeRight(BTNode *Parent, BTNode *Child)
{
    //if parent has child
    if(Parent->Right != NULL)
    {
        free(Parent->Right);
    }

    Parent->Right = Child;
}
```

AVL ADT

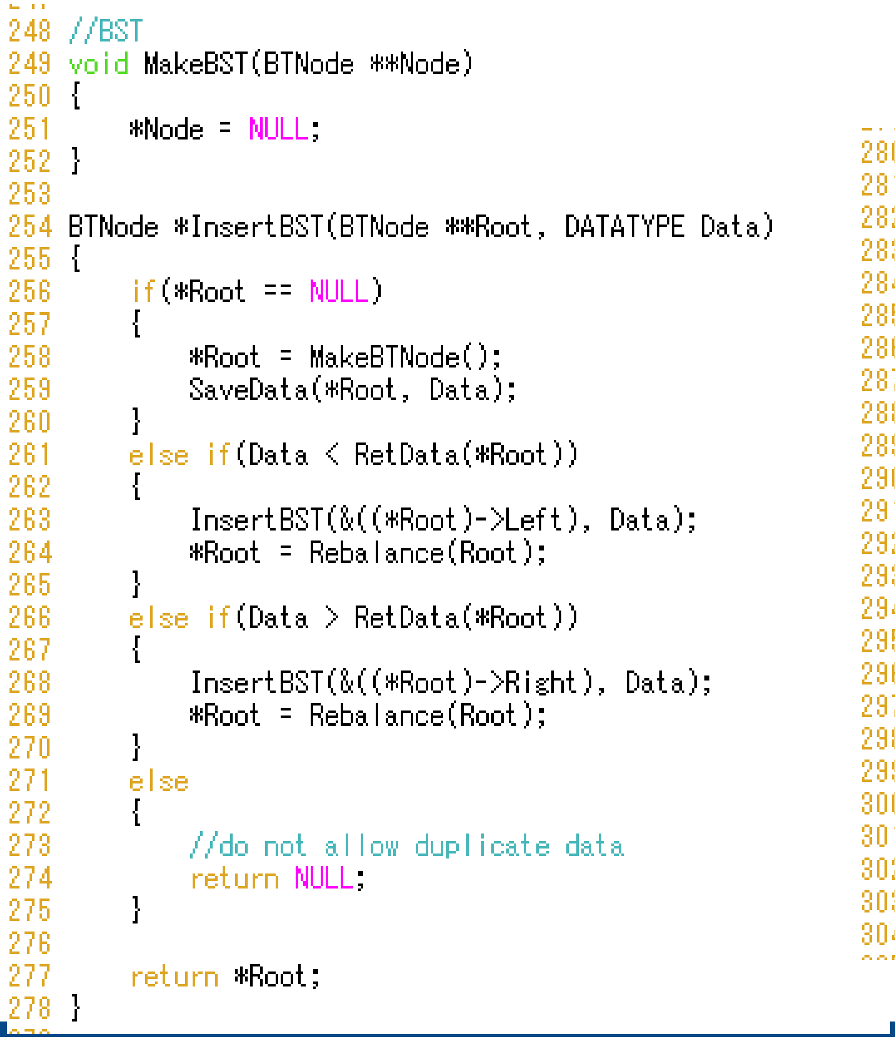


```c
void PreorderTraversal(BTNode * Node)
{
    if(Node == NULL)
    {
        return;
    }

    printf("%d ", Node->Data);
    PreorderTraversal(Node->Left);
    PreorderTraversal(Node->Right);
}

void InorderTraversal(BTNode *Node)
{
    if(Node == NULL)
    {
        return;
    }

    InorderTraversal(Node->Left);
    printf("%d ", Node->Data);
    InorderTraversal(Node->Right);
}

void PostorderTraversal(BTNode *Node)
{
    if(Node == NULL)
    {
        return;
    }

    PostorderTraversal(Node->Left);
    PostorderTraversal(Node->Right);
    printf("%d ", Node->Data);
}
```

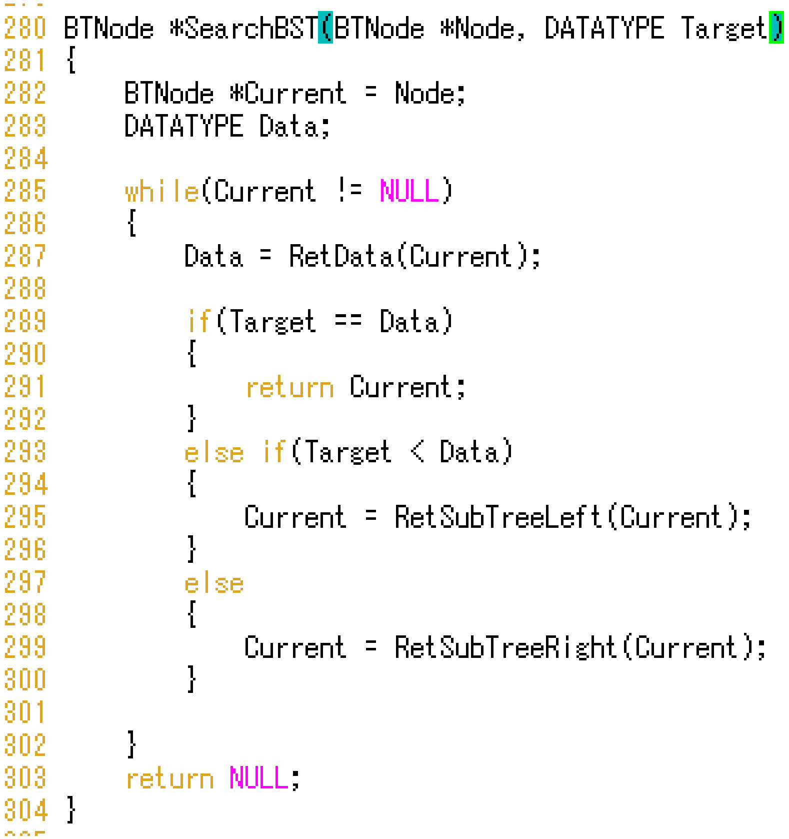


```c
BTNode *RemoveSubTreeLeft(BTNode *Node)
{
    BTNode *Temp;

    if(Node != NULL)
    {
        Temp = Node->Left;
        Node->Left = NULL;
    }

    return Temp;
}

BTNode *RemoveSubTreeRight(BTNode *Node)
{
    BTNode *Temp;

    if(Node != NULL)
    {
        Temp = Node->Right;
        Node->Right = NULL;
    }

    return Temp;
}

void ChangeSubTreeLeft(BTNode *Parent, BTNode *Child)
{
    Parent->Left = Child;
}

void ChangeSubTreeRight(BTNode *Parent, BTNode *Child)
{
    Parent->Right = Child;
}
```

AVL ADT

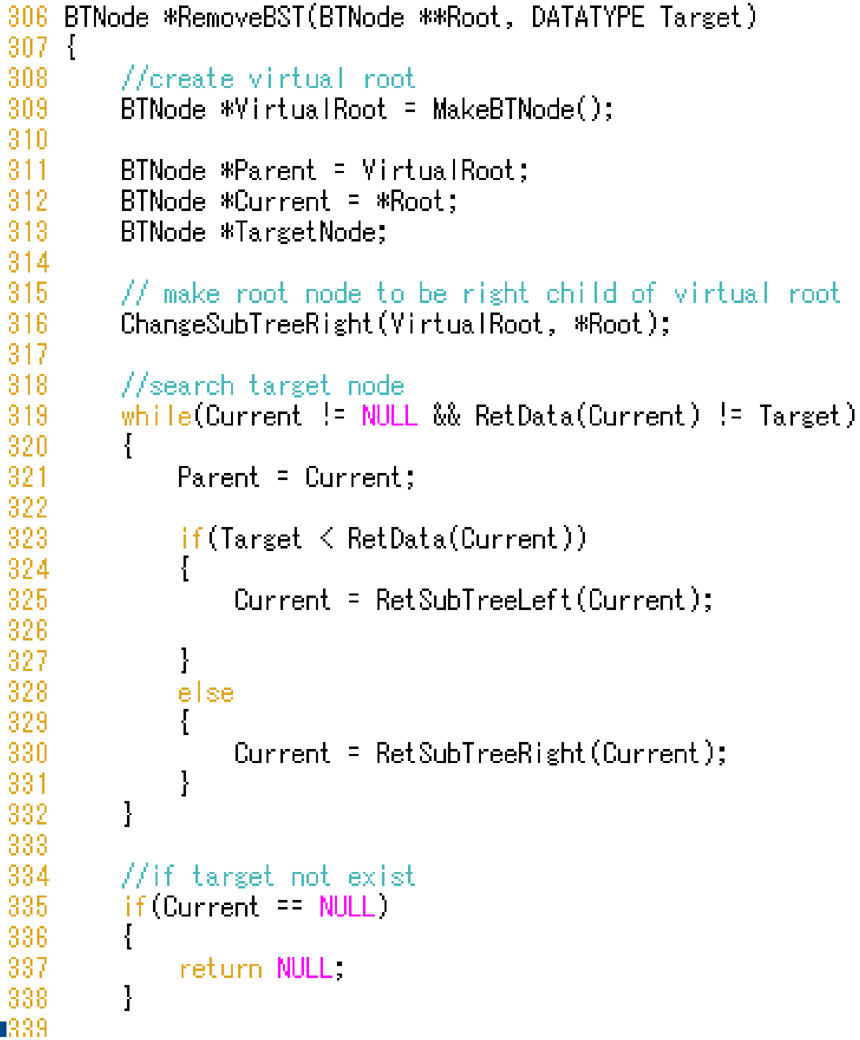


```c
//BST
void MakeBST(BTNode **Node)
{
    *Node = NULL;
}

BTNode *InsertBST(BTNode **Root, DATATYPE Data)
{
    if(*Root == NULL)
    {
        *Root = MakeBTNode();
        SaveData(*Root, Data);
    }
    else if(Data < RetData(*Root))
    {
        InsertBST(&((*Root)->Left), Data);
        *Root = Rebalance(Root);
    }
    else if(Data > RetData(*Root))
    {
        InsertBST(&((*Root)->Right), Data);
        *Root = Rebalance(Root);
    }
    else
    {
        //do not allow duplicate data
        return NULL;
    }

    return *Root;
}
```

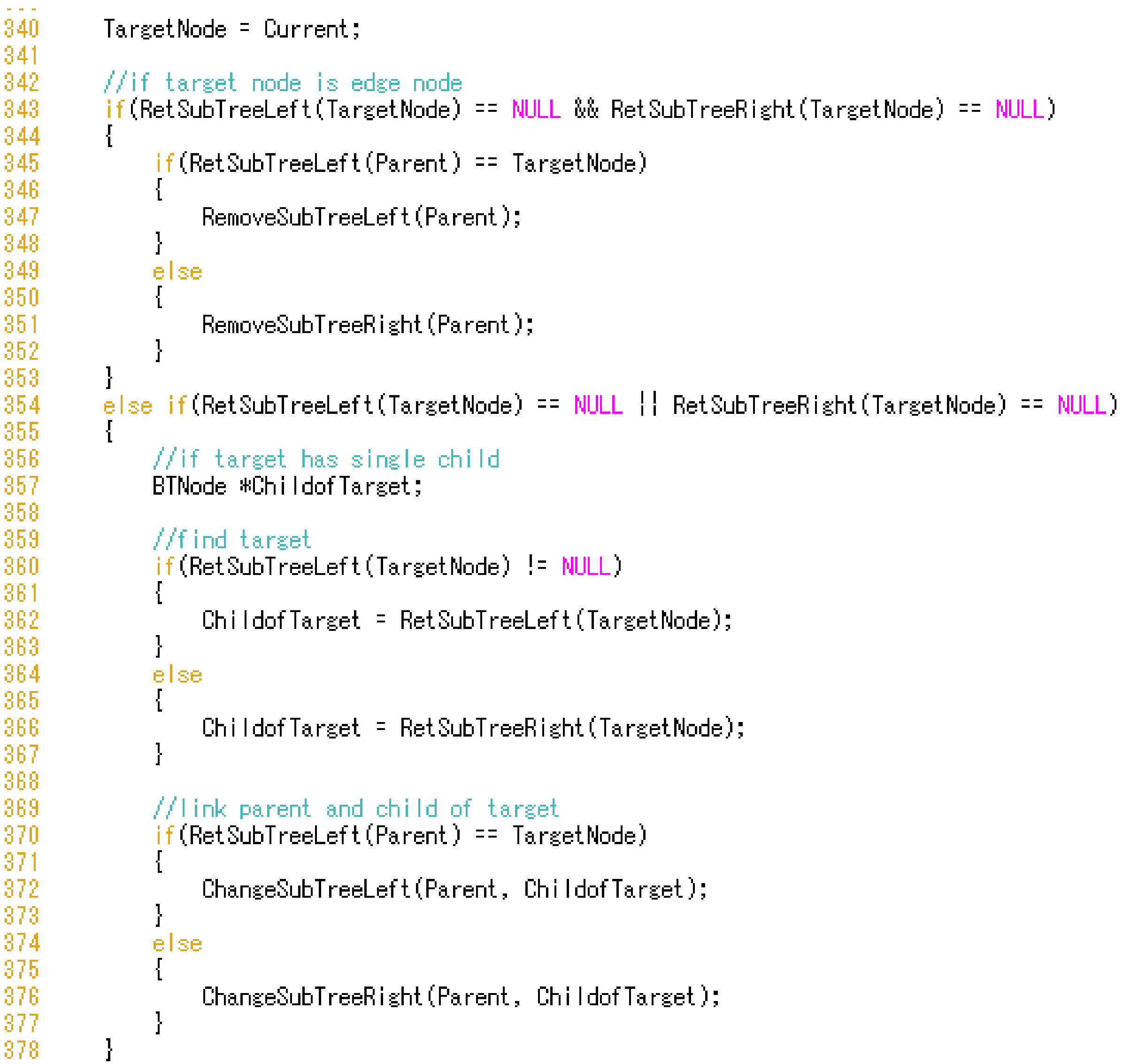


```c
BTNode *SearchBST(BTNode *Node, DATATYPE Target)
{
    BTNode *Current = Node;
    DATATYPE Data;

    while(Current != NULL)
    {
        Data = RetData(Current);

        if(Target == Data)
        {
            return Current;
        }
        else if(Target < Data)
        {
            Current = RetSubTreeLeft(Current);
        }
        else
        {
            Current = RetSubTreeRight(Current);
        }

    }
    return NULL;
}
```

AVL ADT

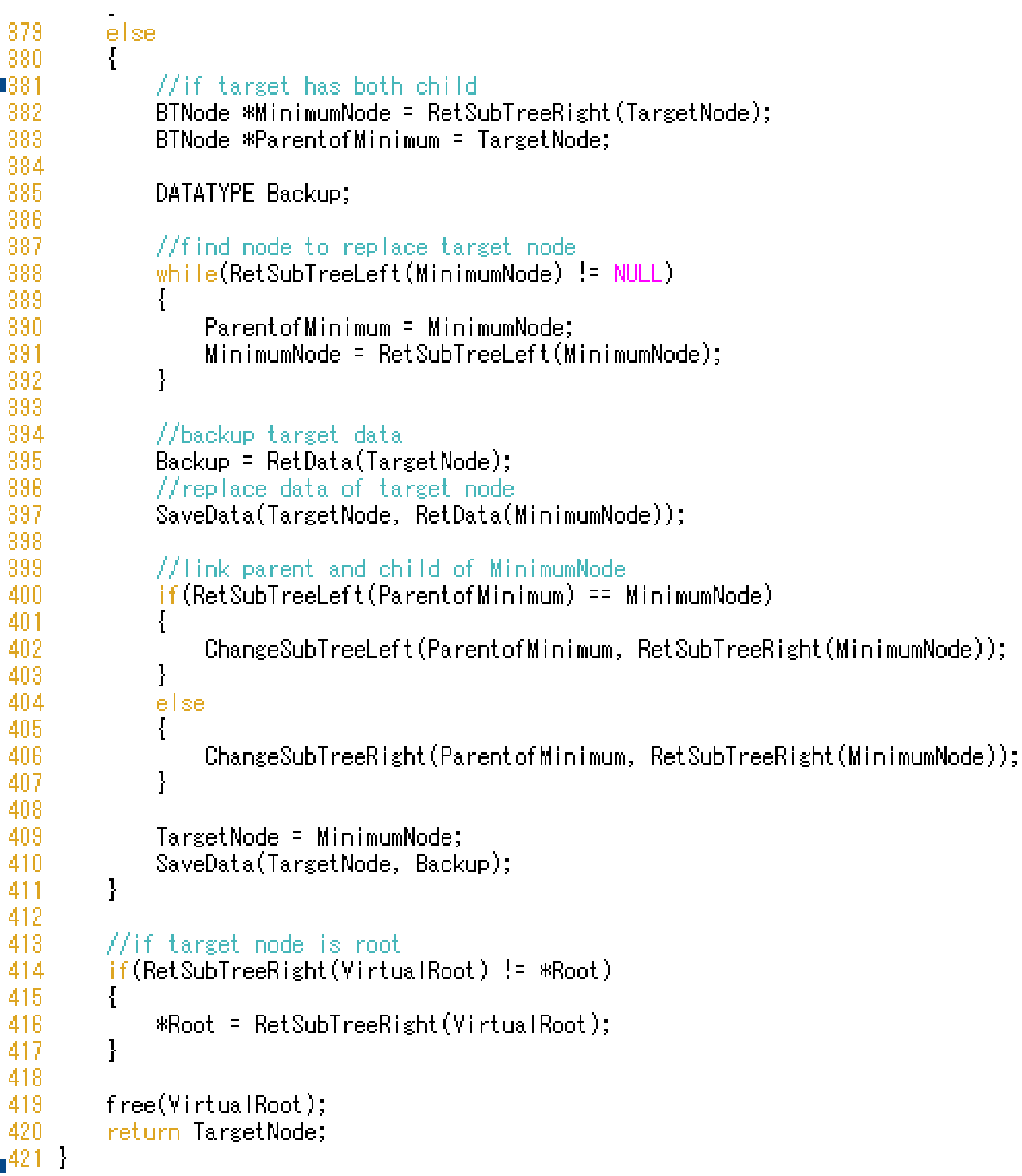


```
BTNode *RemoveBST(BTNode **Root, DATATYPE Target)
{
    //create virtual root
    BTNode *VirtualRoot = MakeBTNode();

    BTNode *Parent = VirtualRoot;
    BTNode *Current = *Root;
    BTNode *TargetNode;

    // make root node to be right child of virtual root
    ChangeSubTreeRight(VirtualRoot, *Root);

    //search target node
    while(Current != NULL && RetData(Current) != Target)
    {
        Parent = Current;

        if(Target < RetData(Current))
        {
            Current = RetSubTreeLeft(Current);

        }
        else
        {
            Current = RetSubTreeRight(Current);
        }
    }

    //if target not exist
    if(Current == NULL)
    {
        return NULL;
    }

```



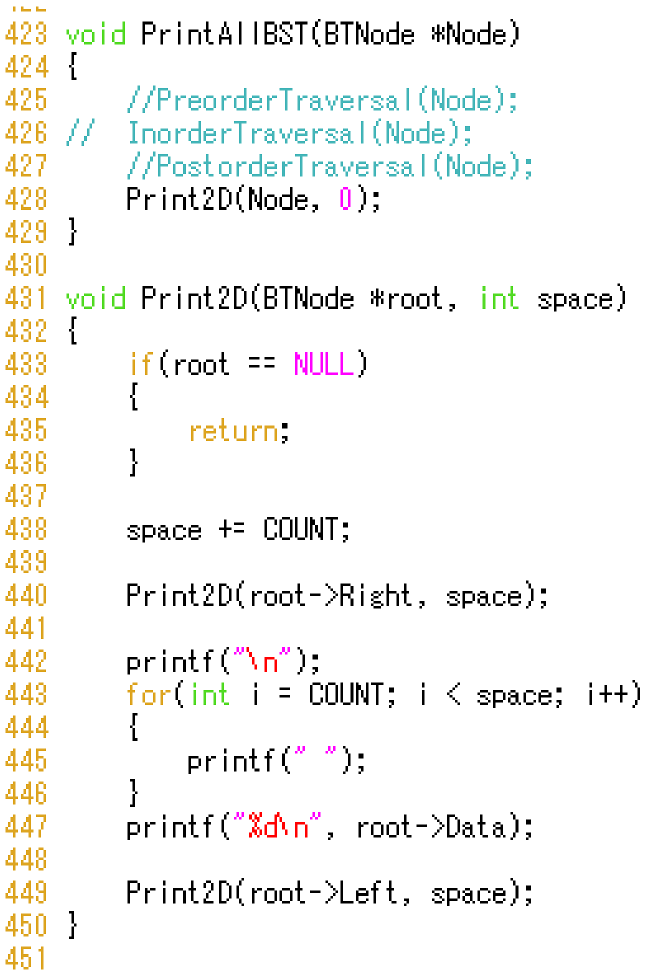
```
...
    TargetNode = Current;

    //if target node is edge node
    if(RetSubTreeLeft(TargetNode) == NULL && RetSubTreeRight(TargetNode) == NULL)
    {
        if(RetSubTreeLeft(Parent) == TargetNode)
        {
            RemoveSubTreeLeft(Parent);
        }
        else
        {
            RemoveSubTreeRight(Parent);
        }
    }
    else if(RetSubTreeLeft(TargetNode) == NULL || RetSubTreeRight(TargetNode) == NULL)
    {
        //if target has single child
        BTNode *ChildofTarget;

        //find target
        if(RetSubTreeLeft(TargetNode) != NULL)
        {
            ChildofTarget = RetSubTreeLeft(TargetNode);
        }
        else
        {
            ChildofTarget = RetSubTreeRight(TargetNode);
        }

        //link parent and child of target
        if(RetSubTreeLeft(Parent) == TargetNode)
        {
            ChildofTarget = RetSubTreeLeft(TargetNode);
            ChangeSubTreeLeft(Parent, ChildofTarget);
        }
        else
        {
            ChangeSubTreeRight(Parent, ChildofTarget);
        }
    }
```

AVL ADT



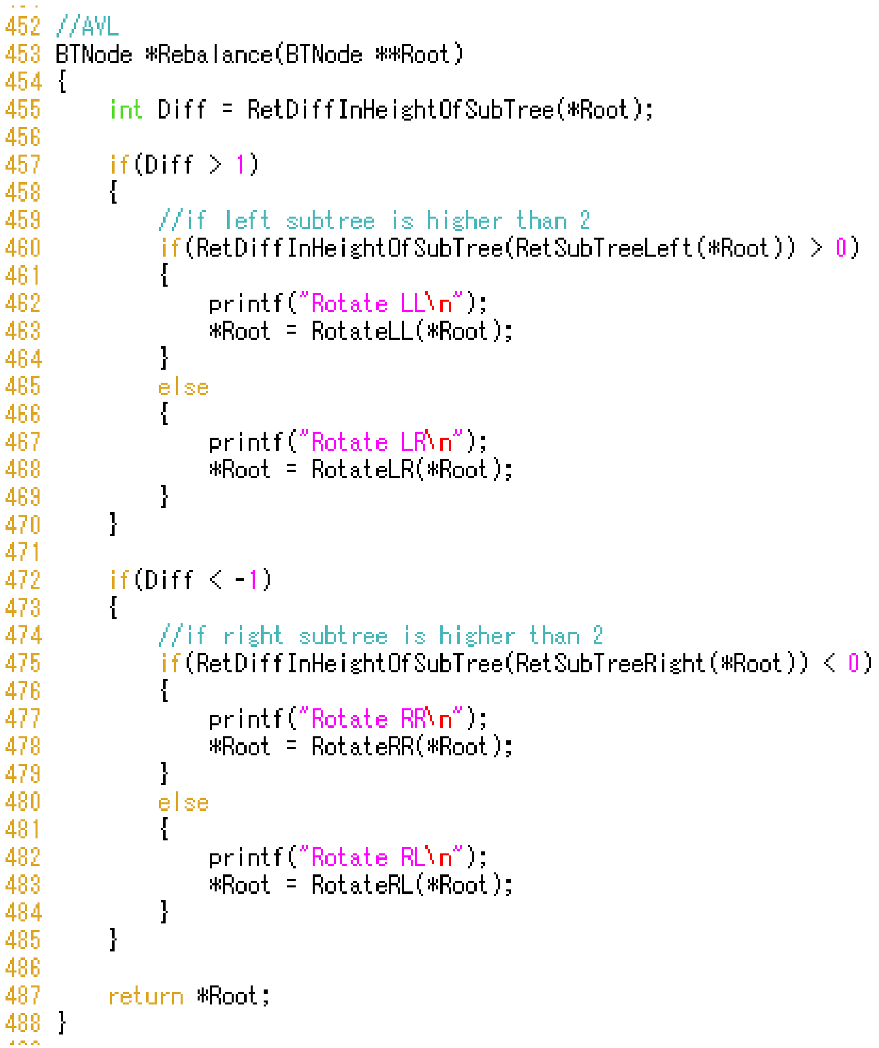
```
    else
    {
        //if target has both child
        BTNode *MinimumNode = RetSubTreeRight(TargetNode);
        BTNode *ParentofMinimum = TargetNode;

        DATATYPE Backup;

        //find node to replace target node
        while(RetSubTreeLeft(MinimumNode) != NULL)
        {
            ParentofMinimum = MinimumNode;
            MinimumNode = RetSubTreeLeft(MinimumNode);
        }

        //backup target data
        Backup = RetData(TargetNode);
        //replace data of target node
        SaveData(TargetNode, RetData(MinimumNode));

        //link parent and child of MinimumNode
        if(RetSubTreeLeft(ParentofMinimum) == MinimumNode)
        {
            ChangeSubTreeLeft(ParentofMinimum, RetSubTreeRight(MinimumNode));
        }
        else
        {
            ChangeSubTreeRight(ParentofMinimum, RetSubTreeRight(MinimumNode));
        }

        TargetNode = MinimumNode;
        SaveData(TargetNode, Backup);
    }

    //if target node is root
    if(RetSubTreeRight(VirtualRoot) != *Root)
    {
        *Root = RetSubTreeRight(VirtualRoot);
    }

    free(VirtualRoot);
    return TargetNode;
}
```

AVL ADT

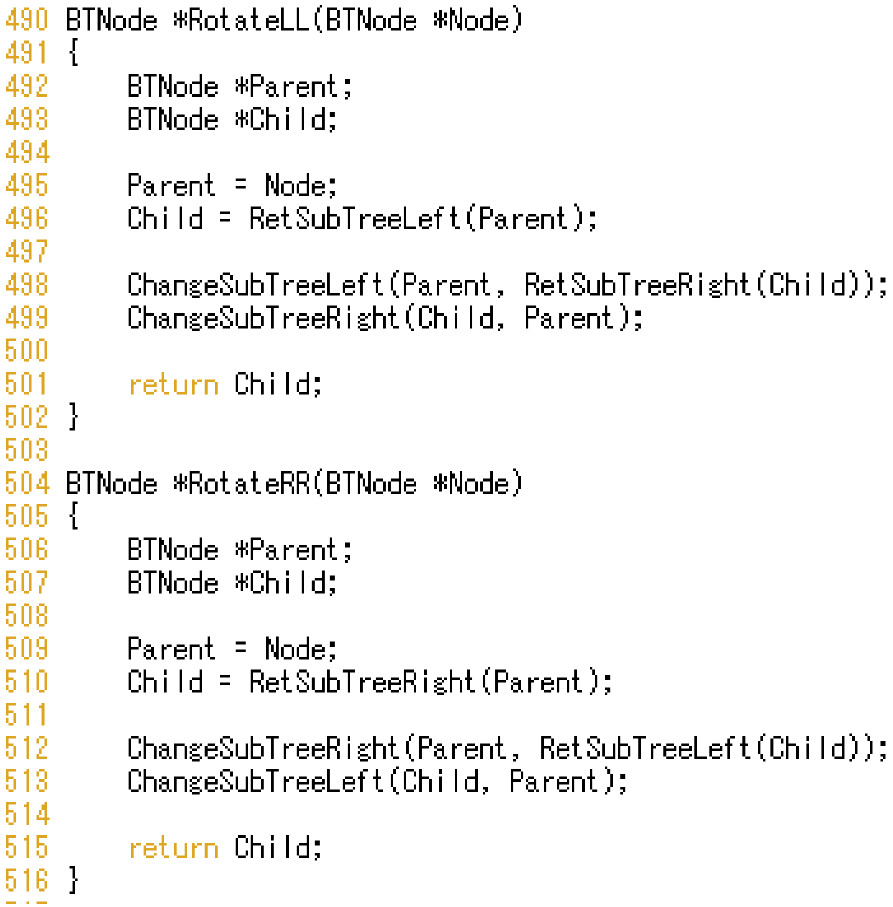


```
void PrintAllBST(BTNode *Node)
{
    //PreorderTraversal(Node);
//  InorderTraversal(Node);
    //PostorderTraversal(Node);
    Print2D(Node, 0);
}

void Print2D(BTNode *root, int space)
{
    if(root == NULL)
    {
        return;
    }

    space += COUNT;

    Print2D(root->Right, space);

    printf("\n");
    for(int i = COUNT; i < space; i++)
    {
        printf(" ");
    }
    printf("%d\n", root->Data);

    Print2D(root->Left, space);
}
```

AVL ADT

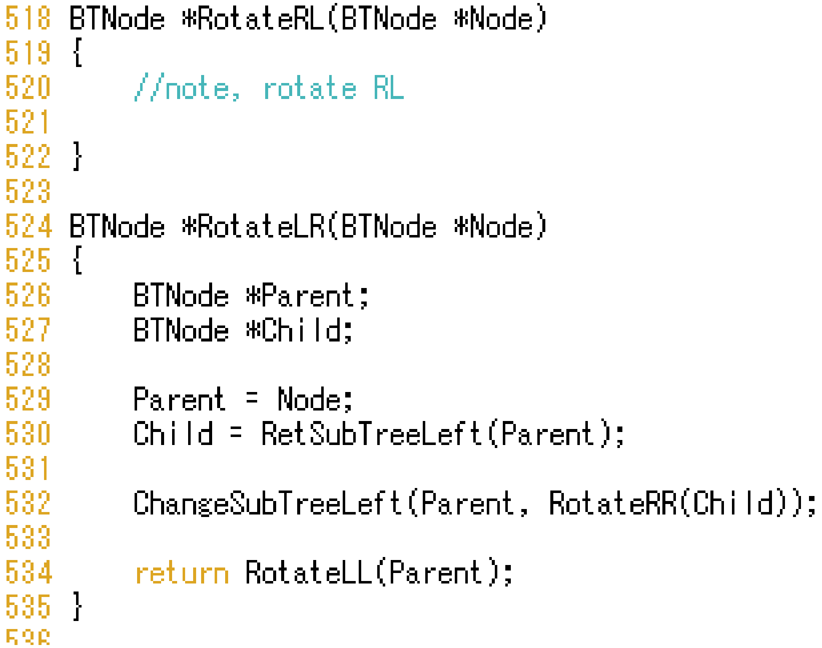


```
//AVL
BTNode *Rebalance(BTNode **Root)
{
    int Diff = RetDiffInHeightOfSubTree(*Root);

    if(Diff > 1)
    {
        //if left subtree is higher than 2
        if(RetDiffInHeightOfSubTree(RetSubTreeLeft(*Root)) > 0)
        {
            printf("Rotate LL\n");
            *Root = RotateLL(*Root);
        }
        else
        {
            printf("Rotate LR\n");
            *Root = RotateLR(*Root);
        }
    }

    if(Diff < -1)
    {
        //if right subtree is higher than 2
        if(RetDiffInHeightOfSubTree(RetSubTreeRight(*Root)) < 0)
        {
            printf("Rotate RR\n");
            *Root = RotateRR(*Root);
        }
        else
        {
            printf("Rotate RL\n");
            *Root = RotateRL(*Root);
        }
    }

    return *Root;
}
```

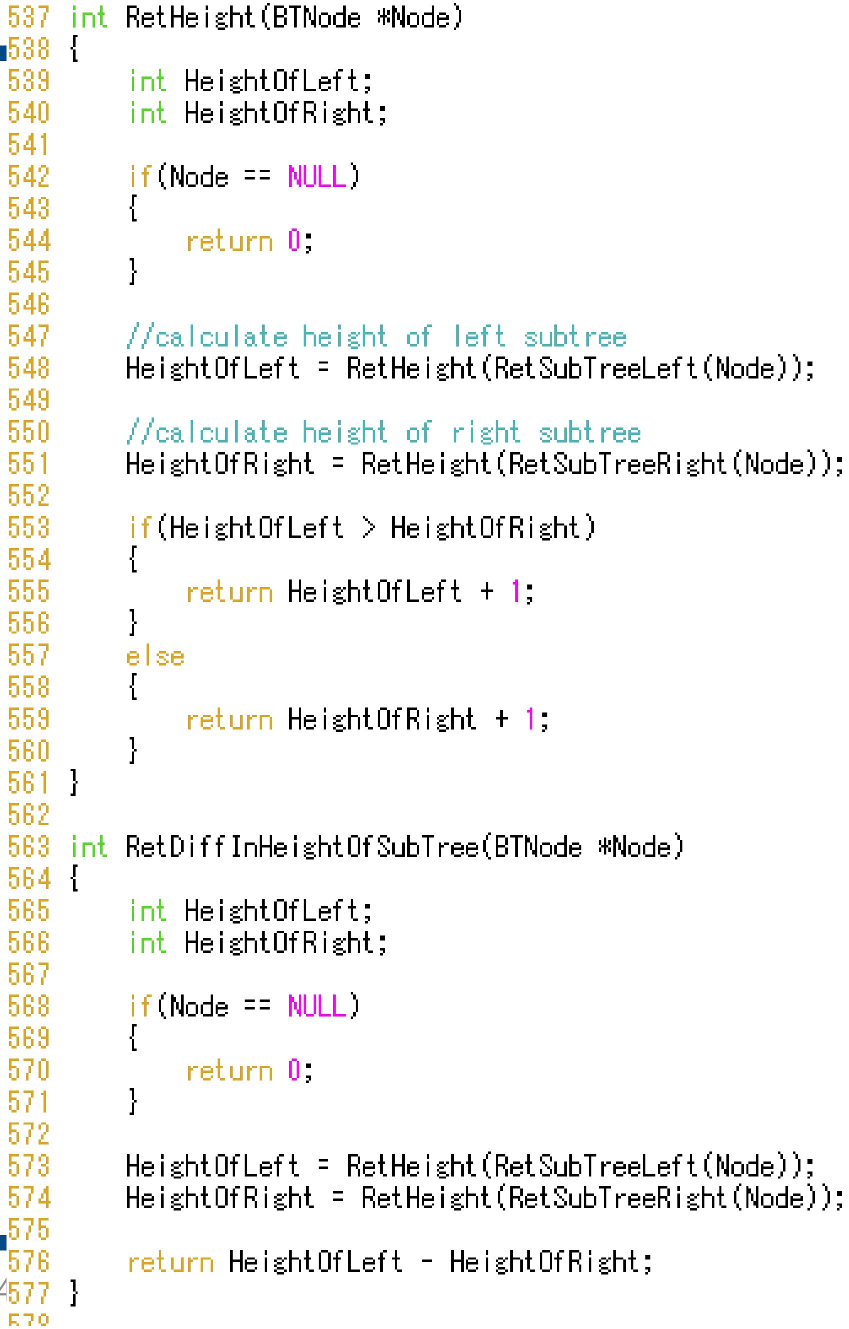


```
BTNode *RotateLL(BTNode *Node)
{
    BTNode *Parent;
    BTNode *Child;

    Parent = Node;
    Child = RetSubTreeLeft(Parent);

    ChangeSubTreeLeft(Parent, RetSubTreeRight(Child));
    ChangeSubTreeRight(Child, Parent);

    return Child;
}

BTNode *RotateRR(BTNode *Node)
{
    BTNode *Parent;
    BTNode *Child;

    Parent = Node;
    Child = RetSubTreeRight(Parent);

    ChangeSubTreeRight(Parent, RetSubTreeLeft(Child));
    ChangeSubTreeLeft(Child, Parent);

    return Child;
}
```

AVL ADT



```
BTNode *RotateRL(BTNode *Node)
{
    //note, rotate RL

}

BTNode *RotateLR(BTNode *Node)
{
    BTNode *Parent;
    BTNode *Child;

    Parent = Node;
    Child = RetSubTreeLeft(Parent);

    ChangeSubTreeLeft(Parent, RotateRR(Child));

    return RotateLL(Parent);
}
```



```
int RetHeight(BTNode *Node)
{
    int HeightOfLeft;
    int HeightOfRight;

    if(Node == NULL)
    {
        return 0;
    }

    //calculate height of left subtree
    HeightOfLeft = RetHeight(RetSubTreeLeft(Node));

    //calculate height of right subtree
    HeightOfRight = RetHeight(RetSubTreeRight(Node));

    if(HeightOfLeft > HeightOfRight)
    {
        return HeightOfLeft + 1;
    }
    else
    {
        return HeightOfRight + 1;
    }
}

int RetDiffInHeightOfSubTree(BTNode *Node)
{
    int HeightOfLeft;
    int HeightOfRight;

    if(Node == NULL)
    {
        return 0;
    }

    HeightOfLeft = RetHeight(RetSubTreeLeft(Node));
    HeightOfRight = RetHeight(RetSubTreeRight(Node));

    return HeightOfLeft - HeightOfRight;
}
```

Lab6:AVL

- Submit on GitLab
- Complete Rotate RL function
- Create Lab6 directory on your own GitLab project
- Submit file : source_code(c only, run on linux)
- Filename : StudentID_lab6.c
- Input file : no