HANYANG UNIVERSITY
DIVISION OF COMPUTER SCIENCE

Data Structures

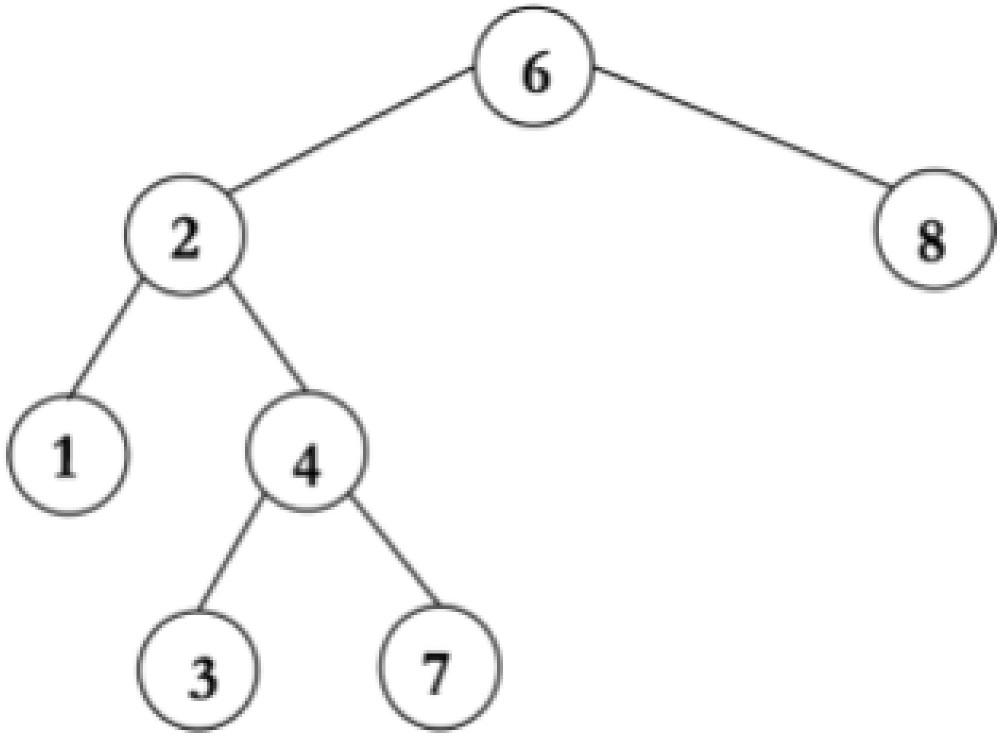
박영준 교수님

Lab5: BST

BST

- A rooted binary tree, whose internal nodes each store key(or value) and each have two distinguished sub-trees.
- The tree satisfies the binary search property, which states that the key in each node must be greater than or equal to any key stored in the left sub-tree, and less then or equal to any key stored in the right sub-tree.

BST



BST ADT

Binary Tree

- BTNode *MakeBTNode(void);
    - Create & initialize binary tree node
    - Return binary tree node
- void SaveData(BTNode *Node, DATATYPE Data);
    - Save Data into Node
- DATATYPE RetData(BTNode *Node);
    - Return data of Node

BST ADT

Binary Tree

- void MakeSubTreeLeft(BTNode *Parent, BTNode *Child);
    - Link Child with left edge of Parent
- void MakeSubTreeRight(BTNode *Parent, BTNode *Child);
    - Link Child with right edge of Parent

BST ADT

Binary Tree

- BTNode *RetSubTreeLeft(BTNode *Node);
    - Return left child of Node
- BTNode *RetSubTreeRight(BTNode *Node);
    - Return right child of Node

BST ADT

Binary Tree

- BTNode *RemoveSubTreeLeft(BTNode *Node);
  - Remove left child of Node
  - Link parent and child of target node
  - Return target node, but not free target
- BTNode *RemoveSubTreeRight(BTNode *Node);
  - Remove right child of Node
  - Link parent and child of target node
  - Return target node, but no free target

BST ADT

Binary Tree

- void ChangeSubTreeLeft(BTNode *Parent, BTNode *Child);
  - Change left child of Parent to Child
- void ChangeSubTreeRight(BTNode *Parent, BTNode *Child);
  - Change right child of Parent to Child

BST ADT

Binary Tree

- void PreorderTraversal(BTNode *Node);
- void InorderTraversal(BTNode *Node);
- void PostorderTraversal(BTNode *Node);
    - Traverse tree by given order

BST ADT

BST

- void MakeBST(BTNode **Node);
  - Initialize root node of BST
- void InsertBST(BTNode **Root, DATATYPE Data);
  - Insert new node with Data in Root BST
  - Compare each node and find location to inserted
    - If compared Node has bigger data, compare left child of Node
    - If compared Node has lower data, compare right child of Node
    - If there are no Node to compare, insert new node
  - Do not allow duplicate data

BST ADT

BST

- BTNode *SearchBST(BTNode *Node, DATATYPE Target);
    - Find node has Target
        - If compared Node has bigger data, compare left child of Node
        - If compared Node has lower data, compare right child of Node
    - If no node has Target, return NULL

BST ADT

BST

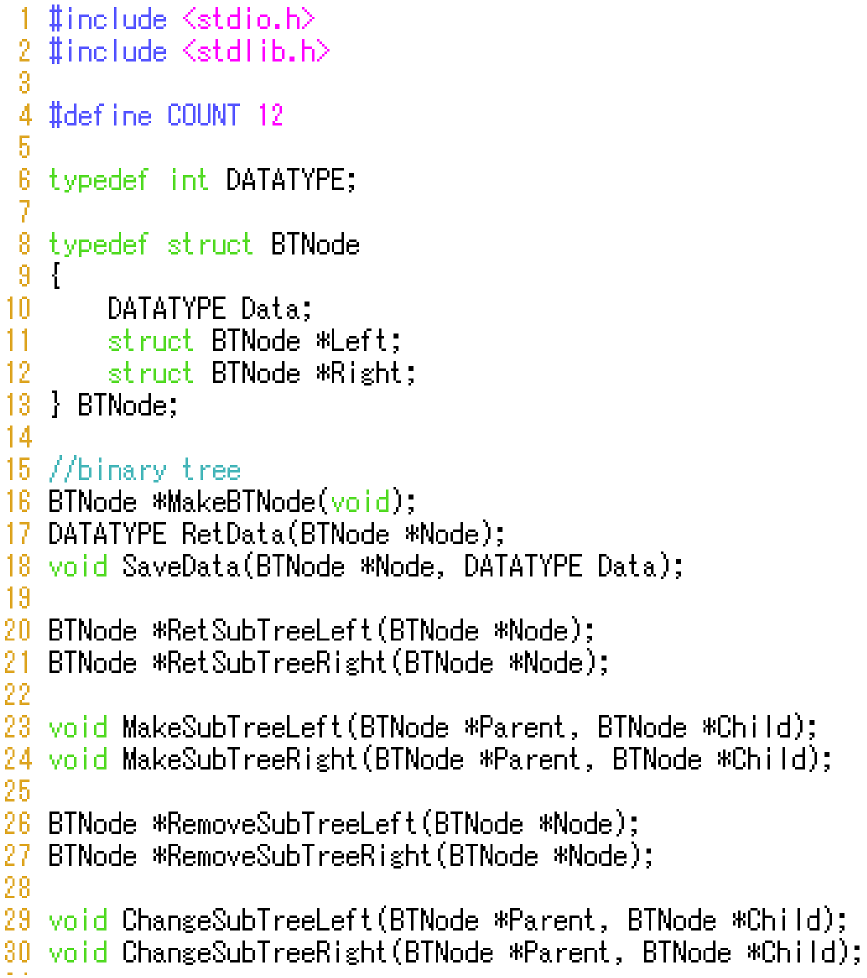
- BTNode *RemoveBST(BTNode **Root, DATATYPE Target);
  - Remove Node has Target in the BST
  - If Target is edge node, remove Target
  - If Target has single child, link parent and child of Target and remove Target
  - If Target has both child, replace Target with largest node in left sub-tree or smallest node in right sub-tree of Target
    And remove Target

BST ADT

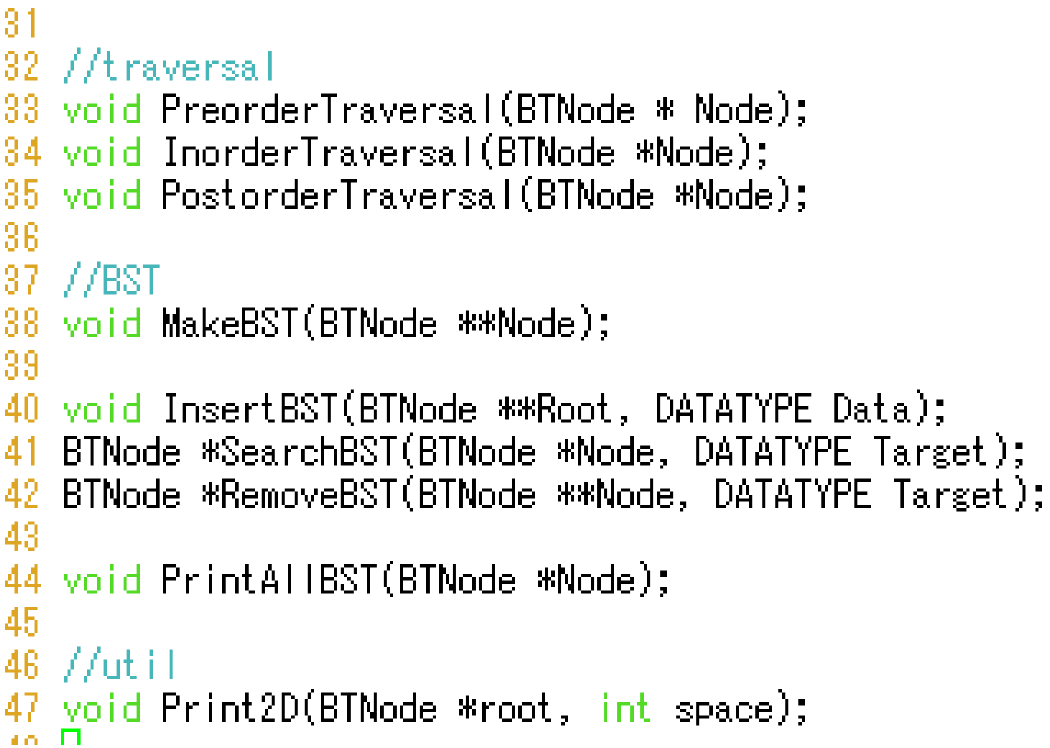
BST

- void PrintAllBST(BTNode *Node)
    - Print all nodes of BST in given order

BST ADT

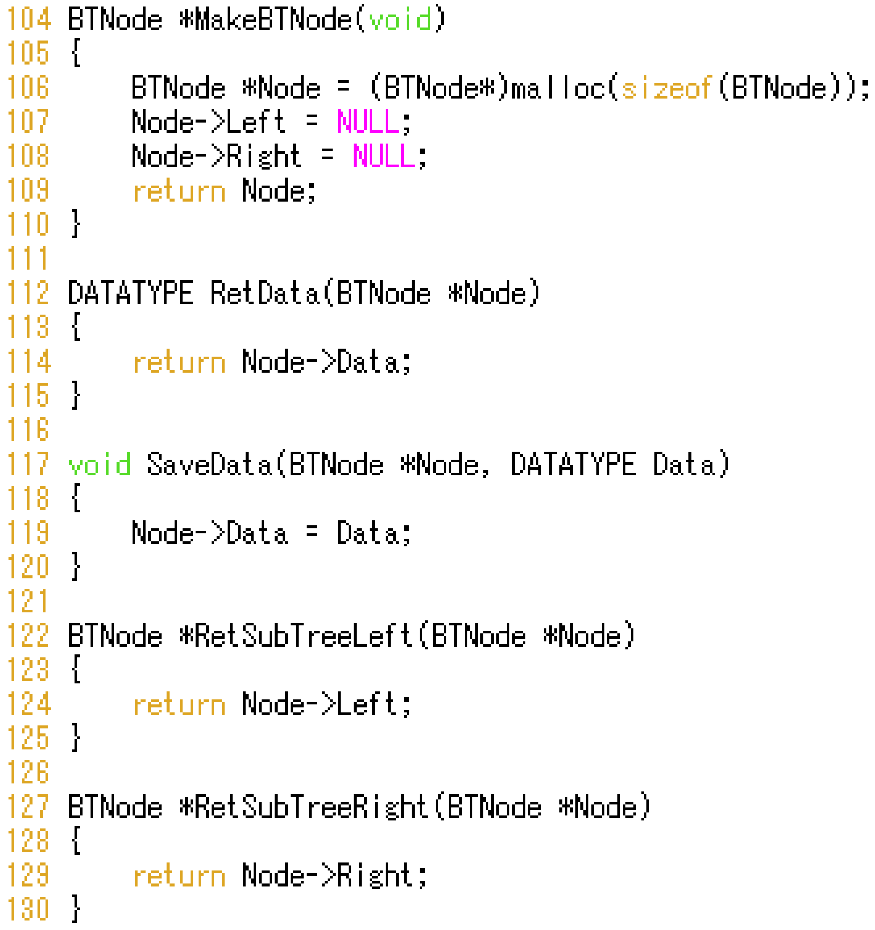


```c
#include <stdio.h>
#include <stdlib.h>

#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);
```

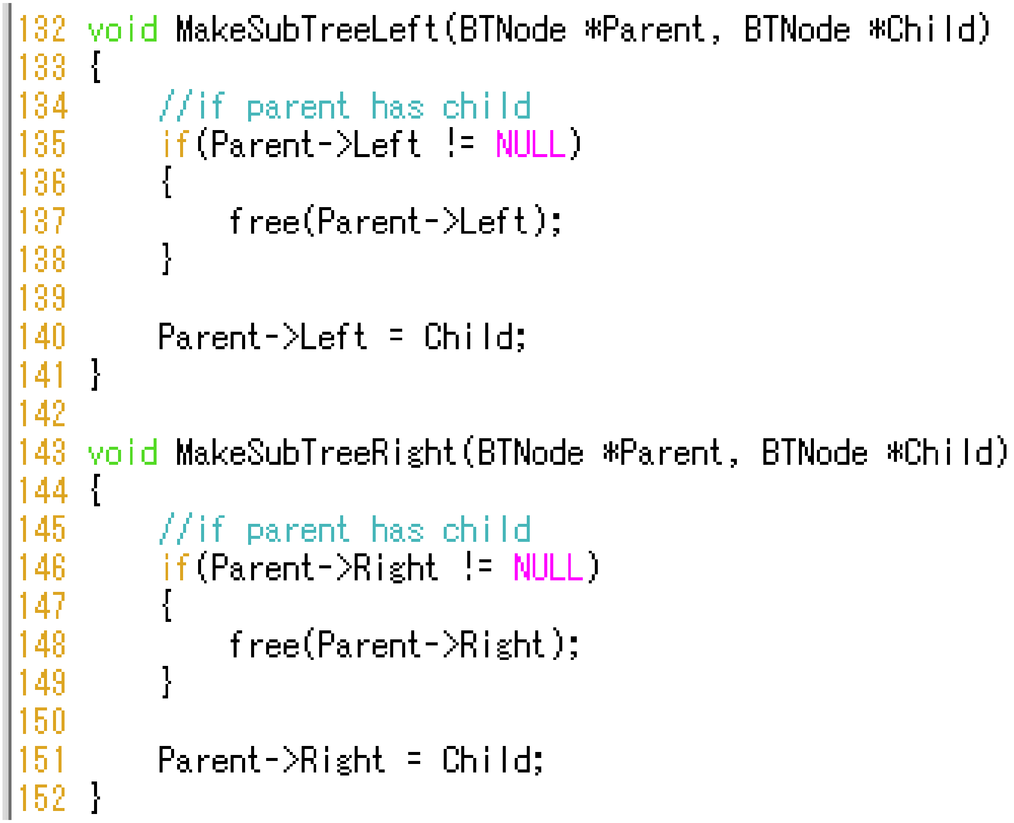


```c

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

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

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

void PrintAllBST(BTNode *Node);

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

BST ADT

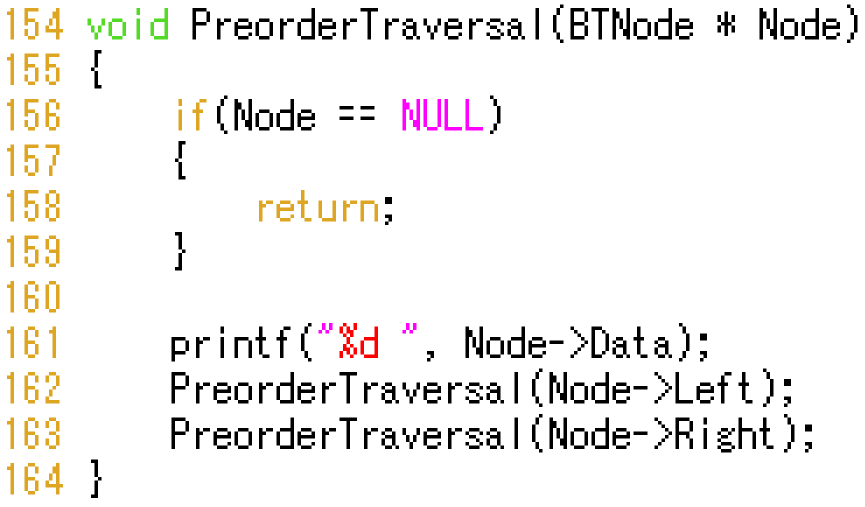


```c
BTNode *MakeBTNode(void)
{
    BTNode *Node = (BTNode*)malloc(sizeof(BTNode));
    Node->Left = NULL;
    Node->Right = NULL;
    return Node;
}

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;
}
```

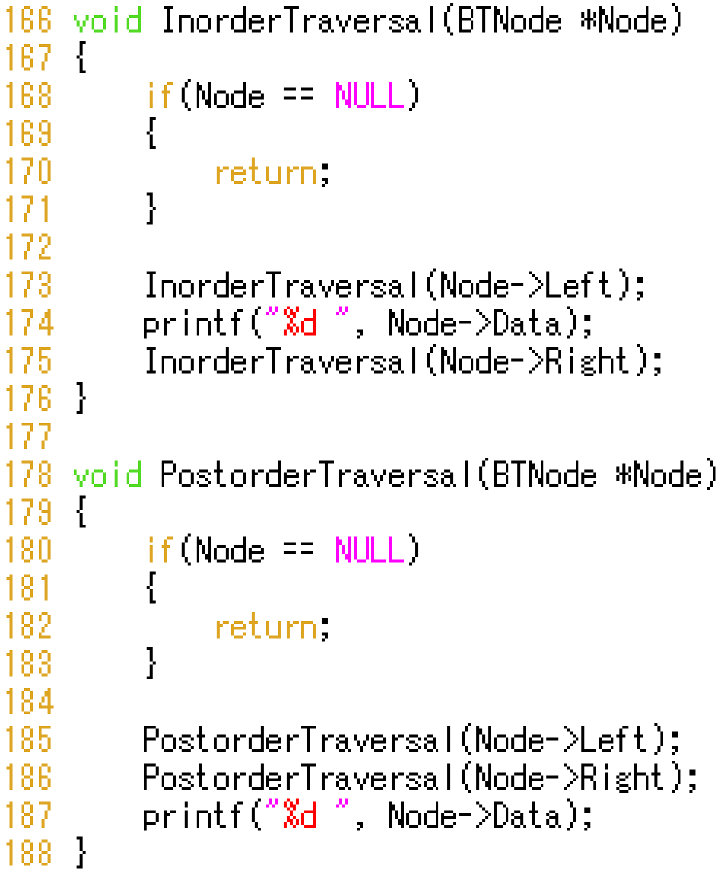


```c
void MakeSubTreeLeft(BTNode *Parent, BTNode *Child)
{
    //if parent has child
    if(Parent->Left != NULL)
    {
        free(Parent->Left);
    }

    Parent->Left = Child;
}

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

    Parent->Right = Child;
}
```

BST ADT

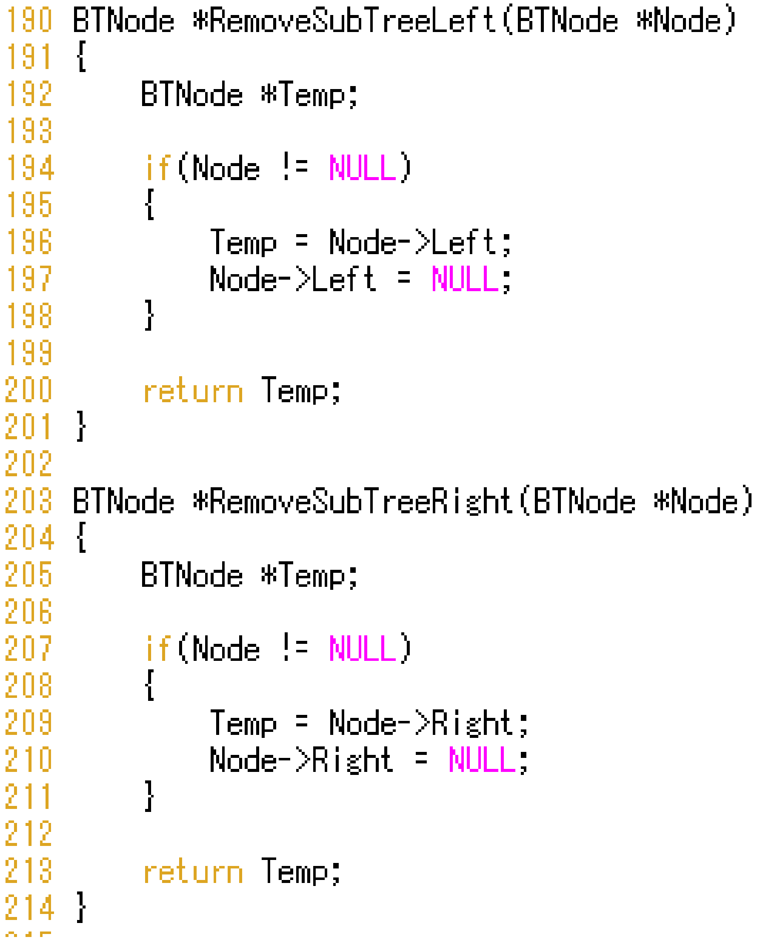


```c
void PreorderTraversal(BTNode * Node)
{
        if(Node == NULL)
        {
            return;
        }

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

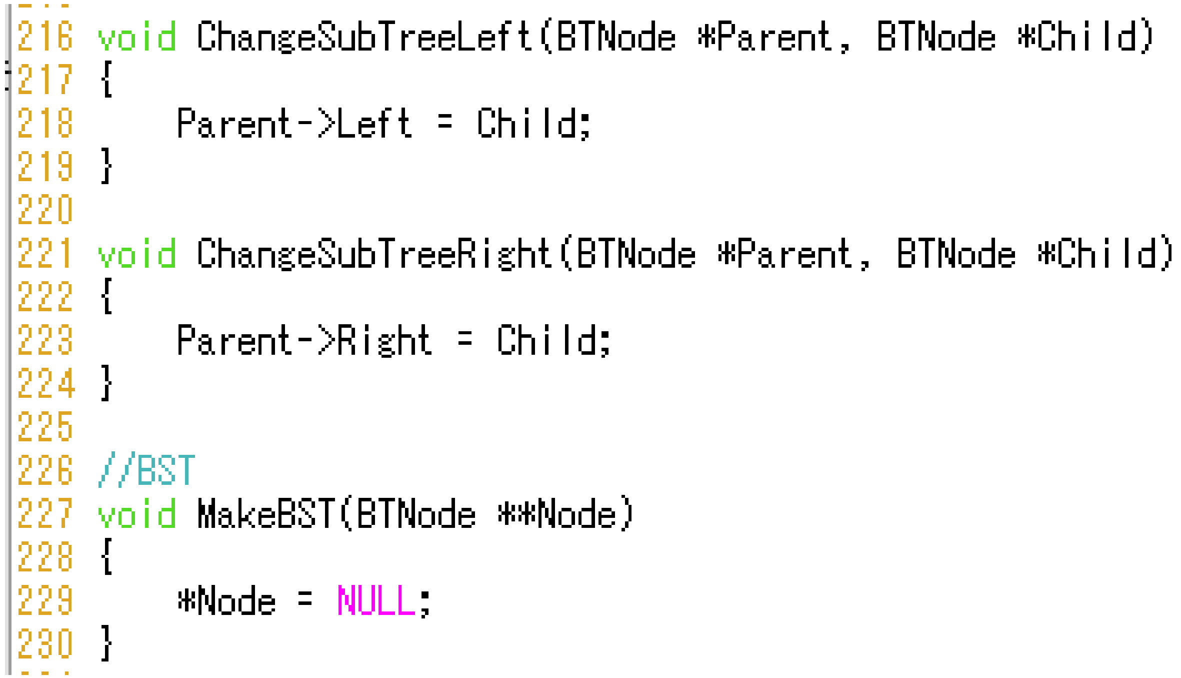


```c
void InorderTraversal(BTNode *Node)
{
        if(Node == NULL)
        {
            return;
        }

        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);
}
```

BST ADT

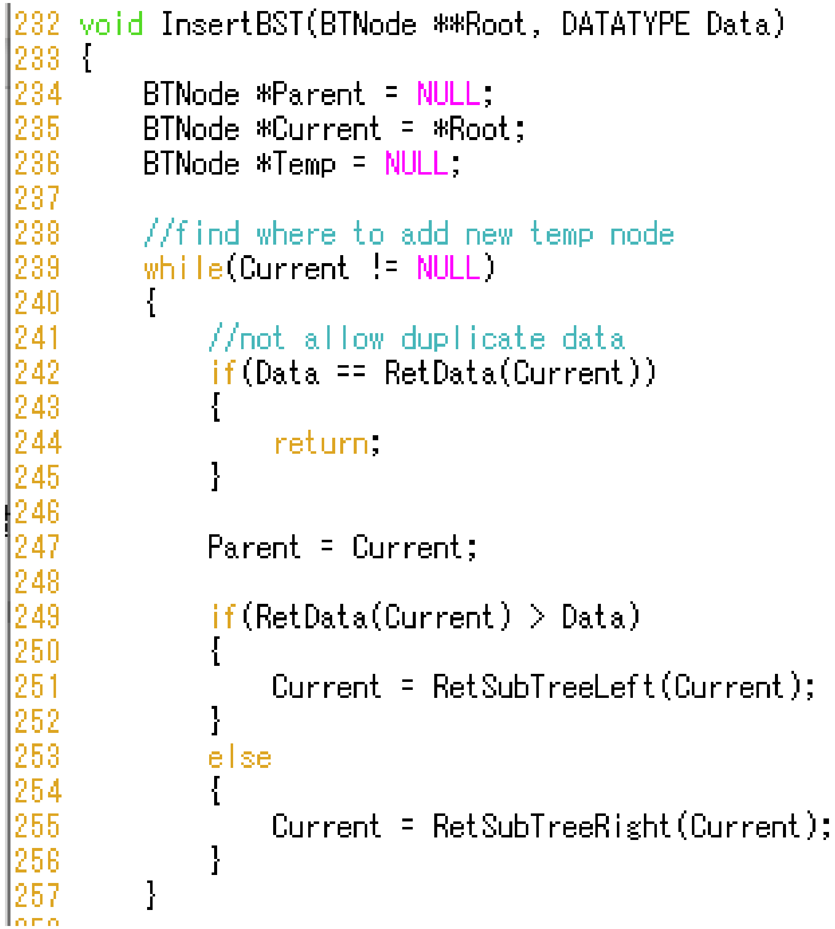


```
BTNode *RemoveSubTreeLeft(BTNode *Node)
{
    BTNode *Temp;

    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;
}
```

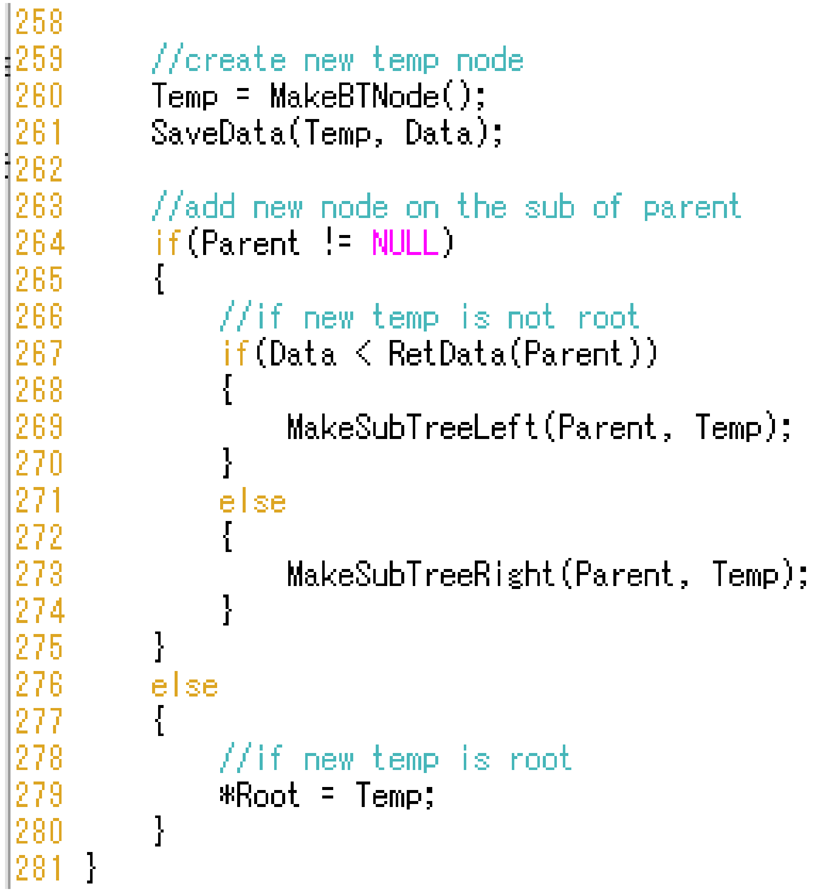


```
void ChangeSubTreeLeft(BTNode *Parent, BTNode *Child)
{
    Parent->Left = Child;
}

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

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

BST ADT

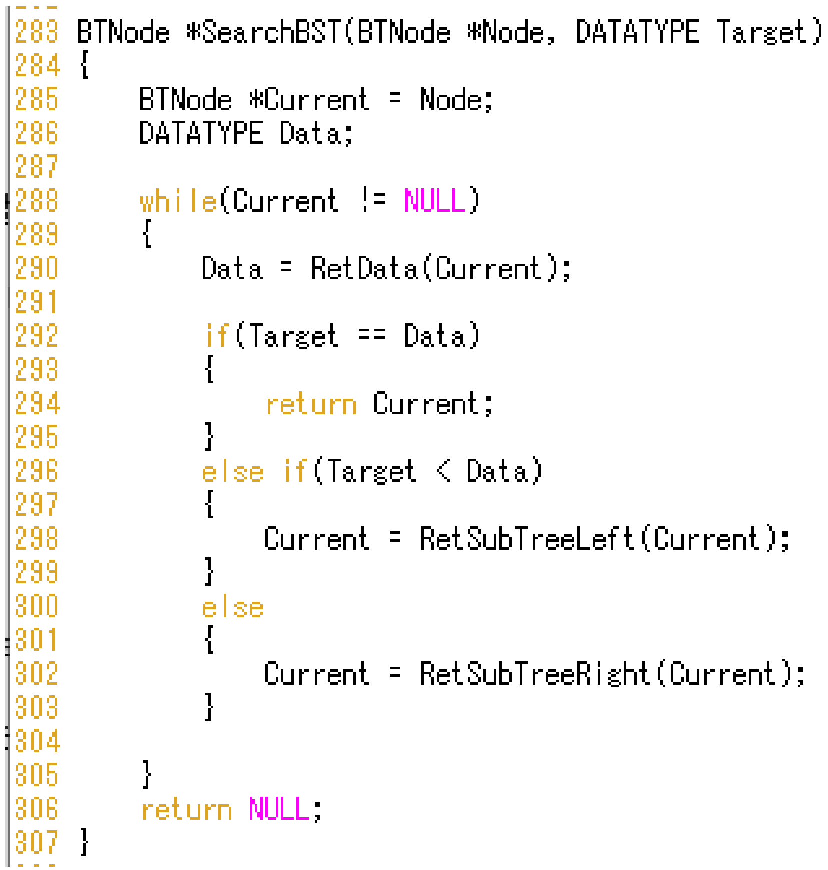


```c
void InsertBST(BTNode **Root, DATATYPE Data)
{
    BTNode *Parent = NULL;
    BTNode *Current = *Root;
    BTNode *Temp = NULL;

    //find where to add new temp node
    while(Current != NULL)
    {
        //not allow duplicate data
        if(Data == RetData(Current))
        {
            return;
        }

        Parent = Current;

        if(RetData(Current) > Data)
        {
            Current = RetSubTreeLeft(Current);
        }
        else
        {
            Current = RetSubTreeRight(Current);
        }
    }
```



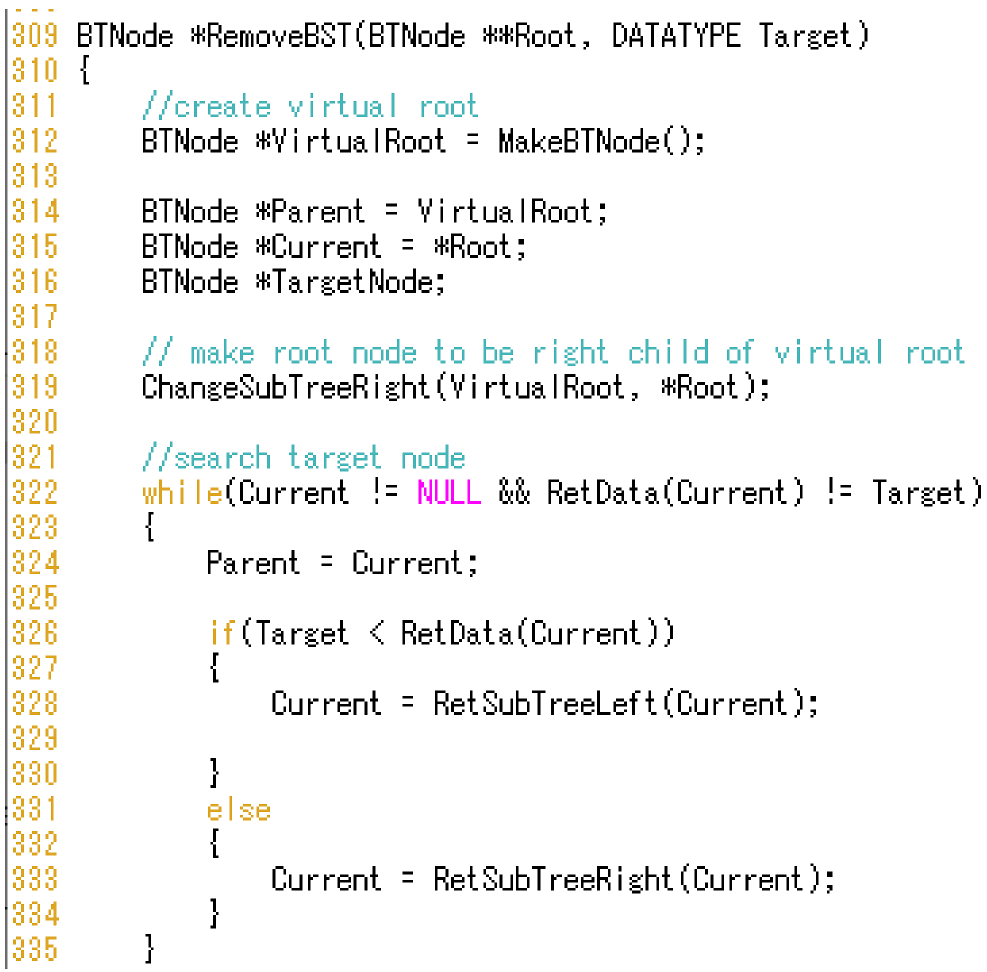
```c

    //create new temp node
    Temp = MakeBTNode();
    SaveData(Temp, Data);

    //add new node on the sub of parent
    if(Parent != NULL)
    {
        //if new temp is not root
        if(Data < RetData(Parent))
        {
            MakeSubTreeLeft(Parent, Temp);
        }
        else
        {
            MakeSubTreeRight(Parent, Temp);
        }
    }
    else
    {
        //if new temp is root
        *Root = Temp;
    }
}
```

BST ADT

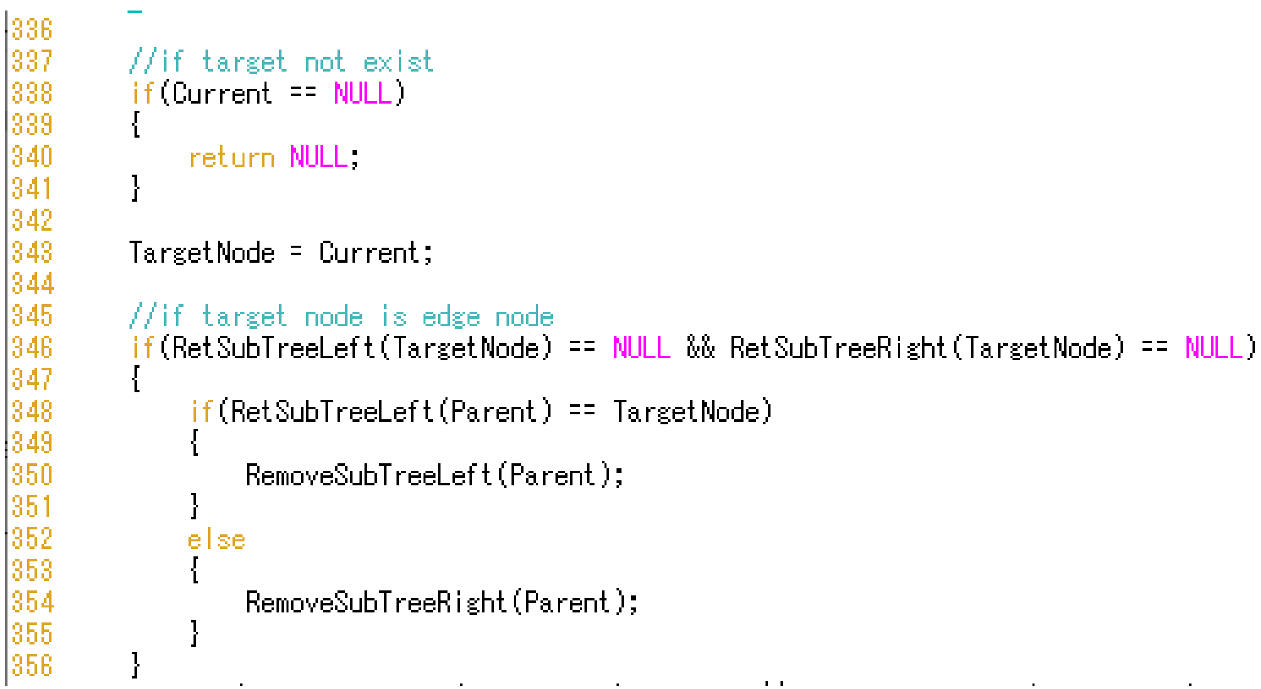


```
BTNode *SearchBST(BTNode *Node, DATATYPE Target)
{
    BTNode *Current = Node;
    DATATYPE Data;

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

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

    }
    return NULL;
}
```

BST ADT

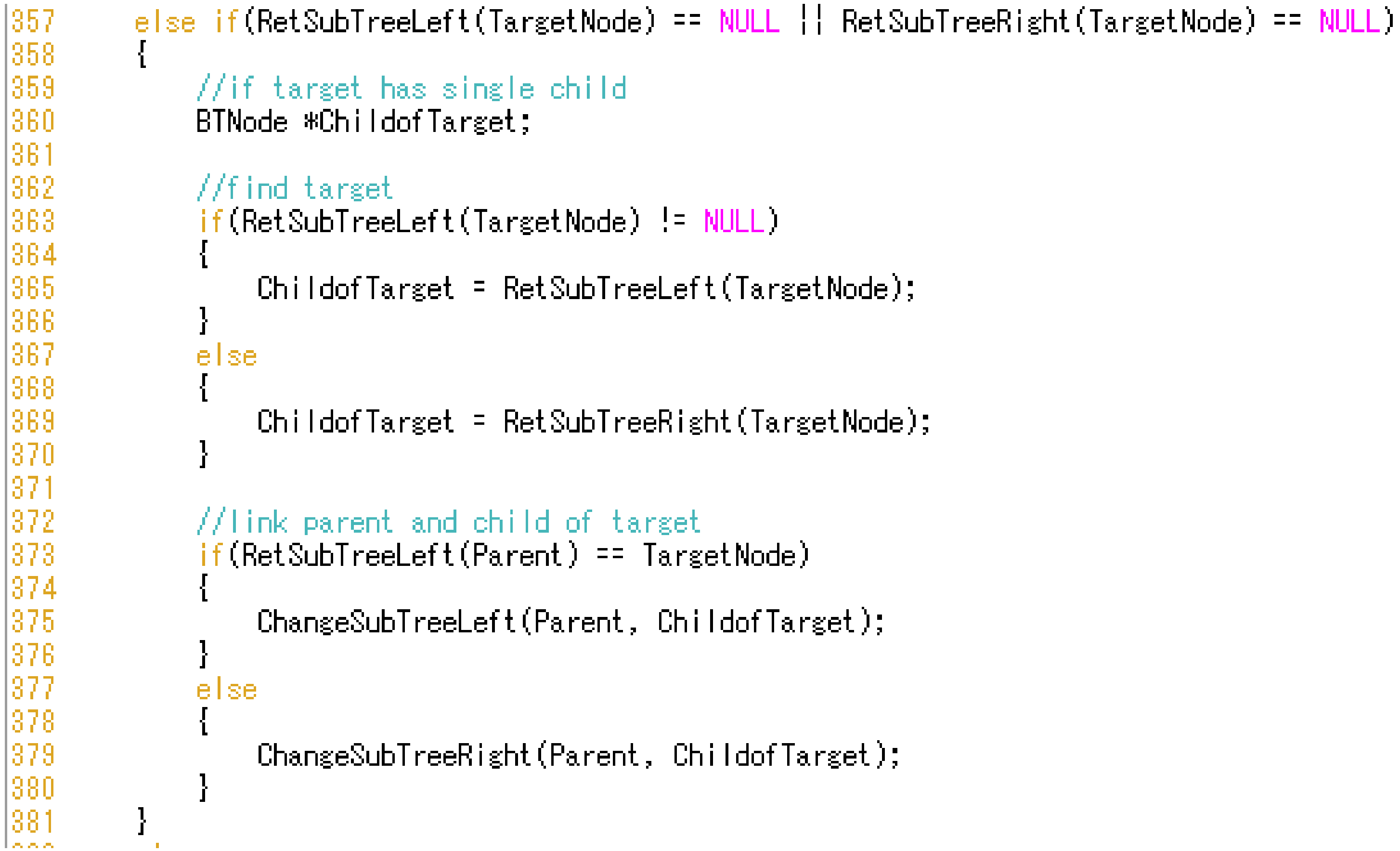


```
BTNode *RemoveBST(BTNode **Root, DATATYPE Target)
{
    //create virtual root
    BTNode *VirtualRoot = MakeBTNode();

    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);
        }
    }
```



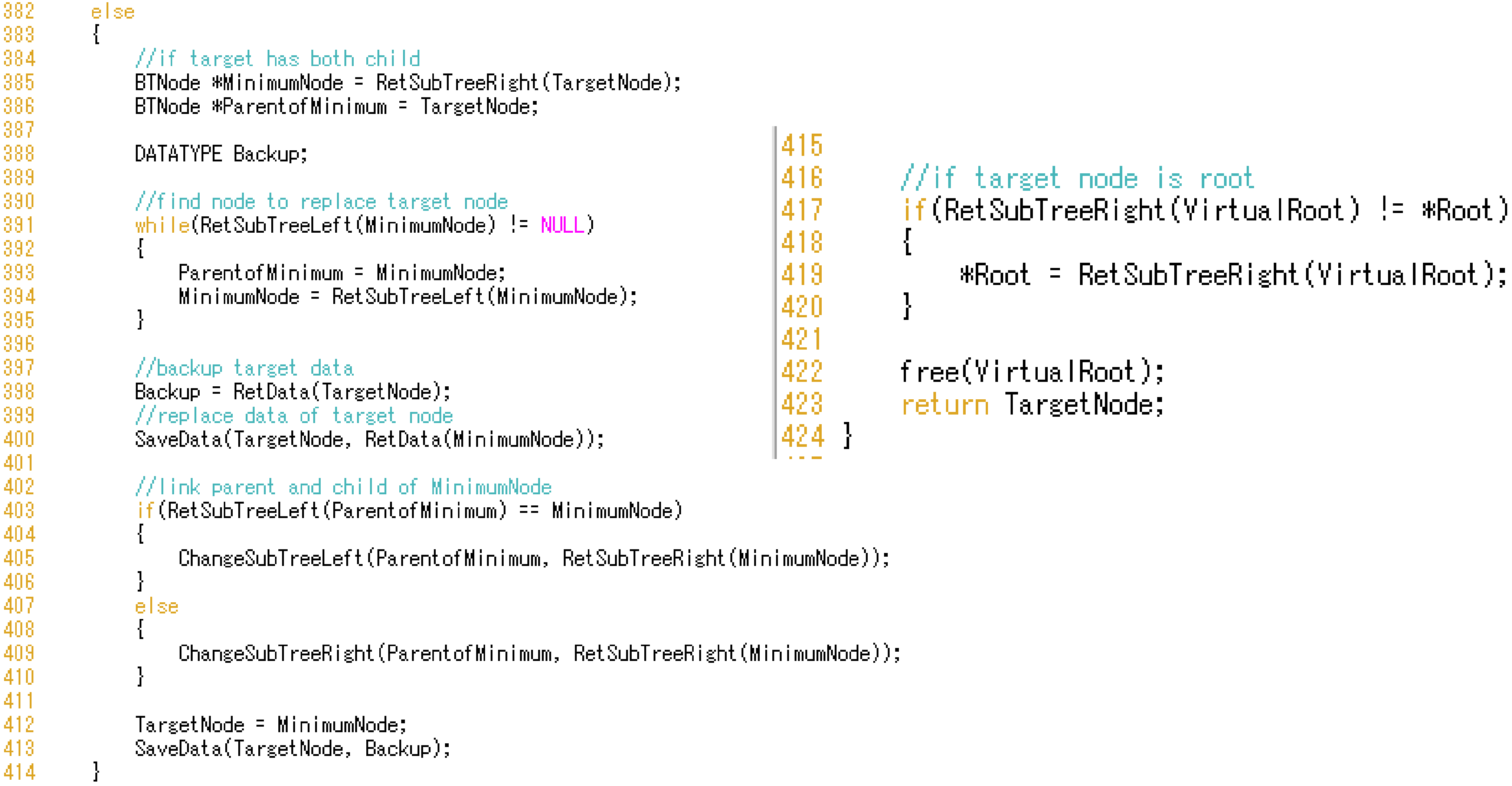
```

    //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);
        }
    }
```

BST ADT



```
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)
    {
        ChangeSubTreeLeft(Parent, ChildofTarget);
    }
    else
    {
        ChangeSubTreeRight(Parent, ChildofTarget);
    }
}
```

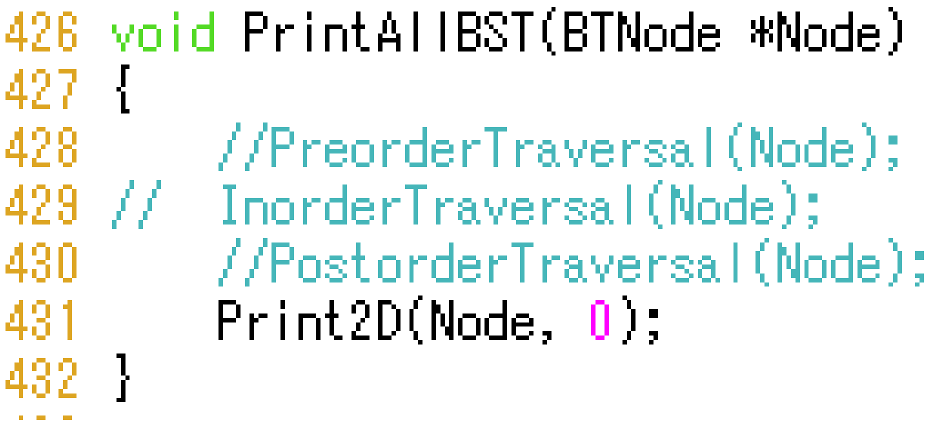
BST ADT



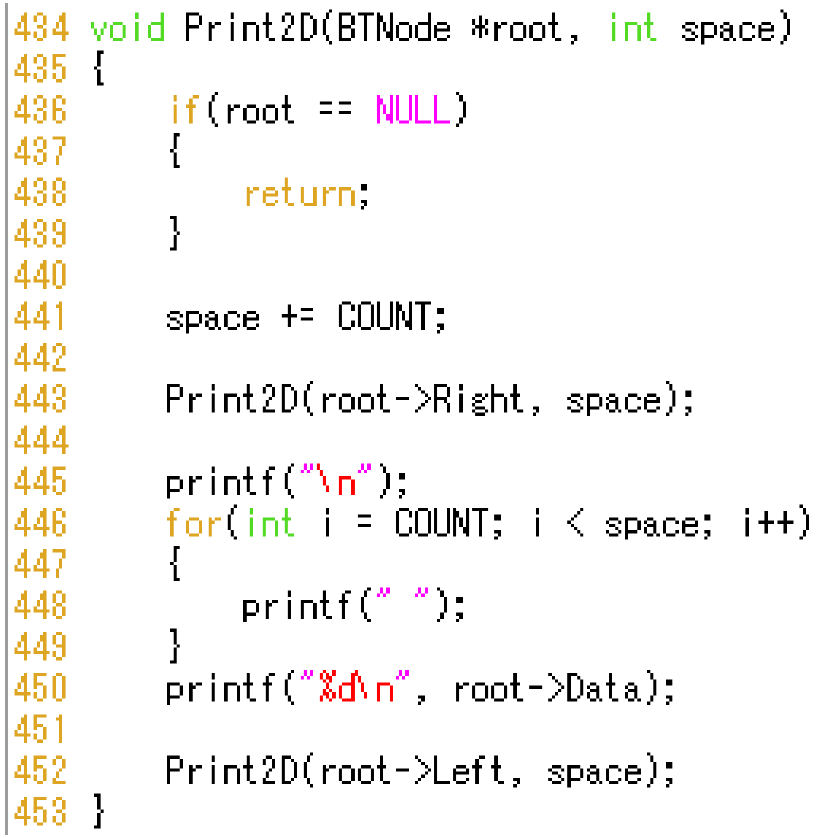
```c
    else
    {
        //if target has both child
        BTNode *MinimumNode = RetSubTreeRight(TargetNode);
        BTNode *ParentofMinimum = TargetNode;

        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);
    }
```

```c

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

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

BST ADT



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



```c
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);
}
```

Lab5: BST

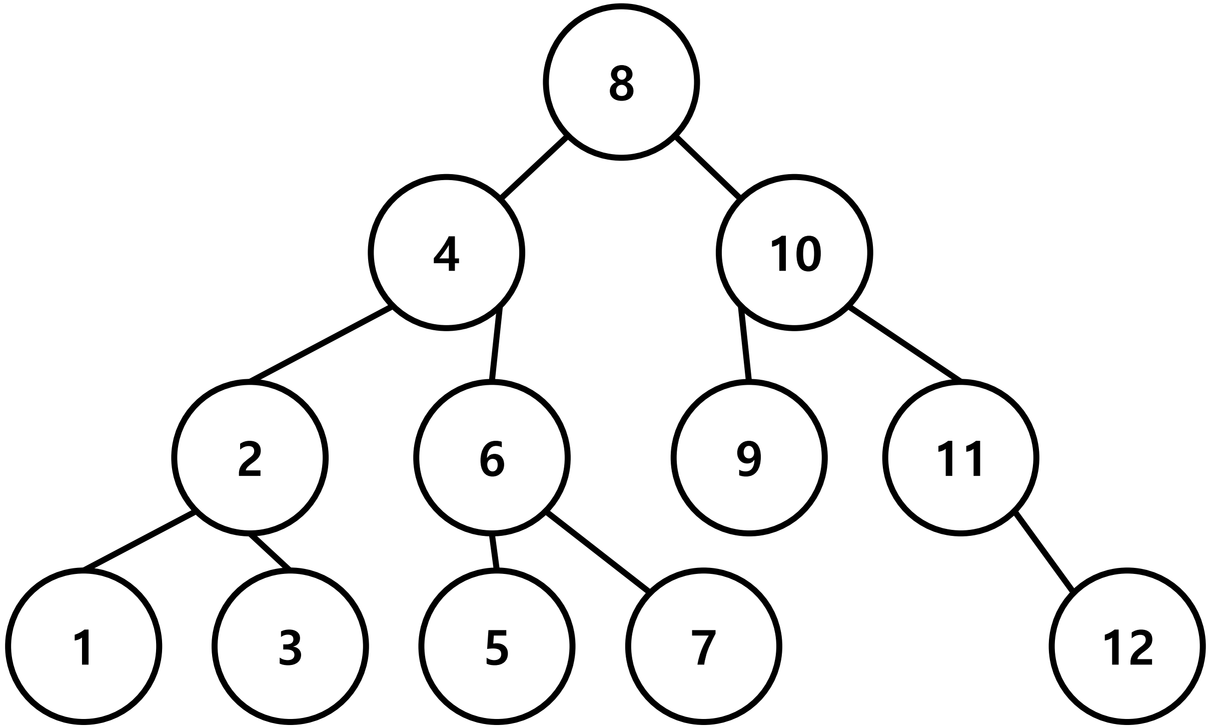
- Submit on GitLab
- BST replace with maximum value in remove operation
- Create Lab5 directory on your own GitLab project
- Submit file : source_code(c only, run on linux)
- Filename : StudentID_lab5.c
- Input file : no

Lab5: BST

- BST replace with maximum value in remove operation
- Change replace algorithm in RemoveBST()
- Node with the minimum value among subtree of the right child
- → Node with the maximum value among subtree of the left child

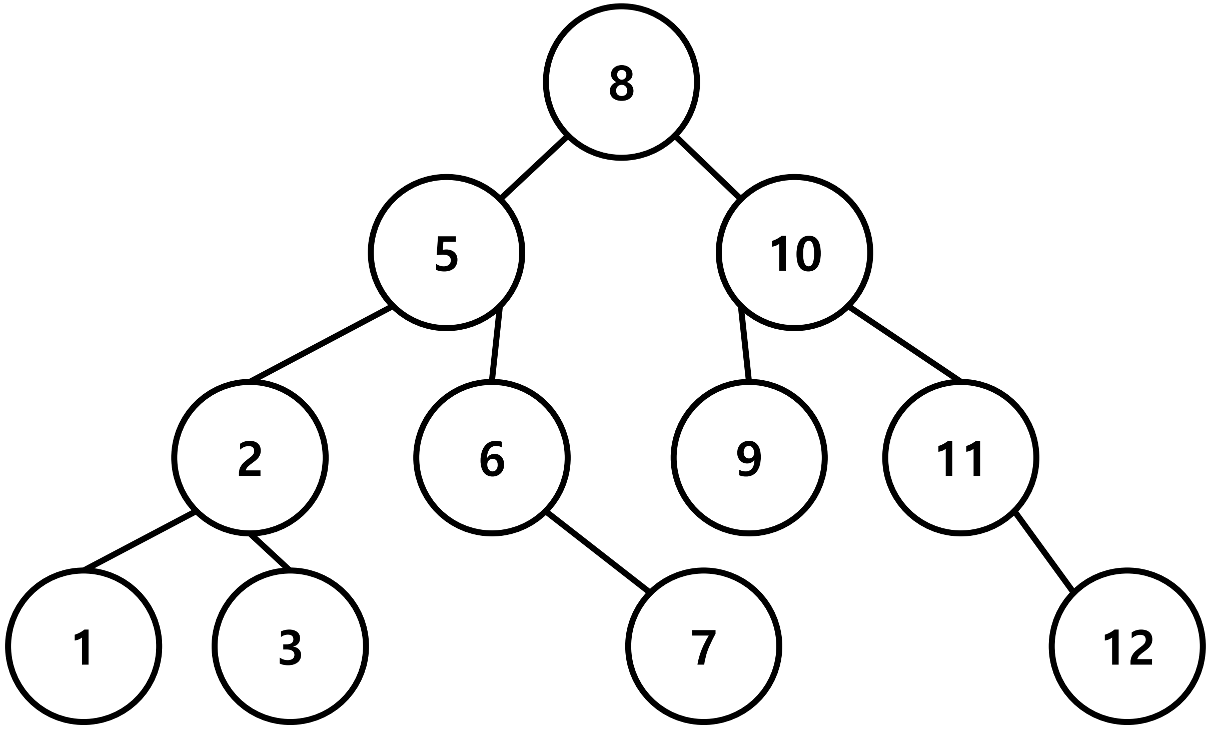
Lab5: BST

- Example
  - In original code, delete 4
  - Find minimum value in the right subtree
  - → 5
  - Replace 4 with 5
  - And delete 4



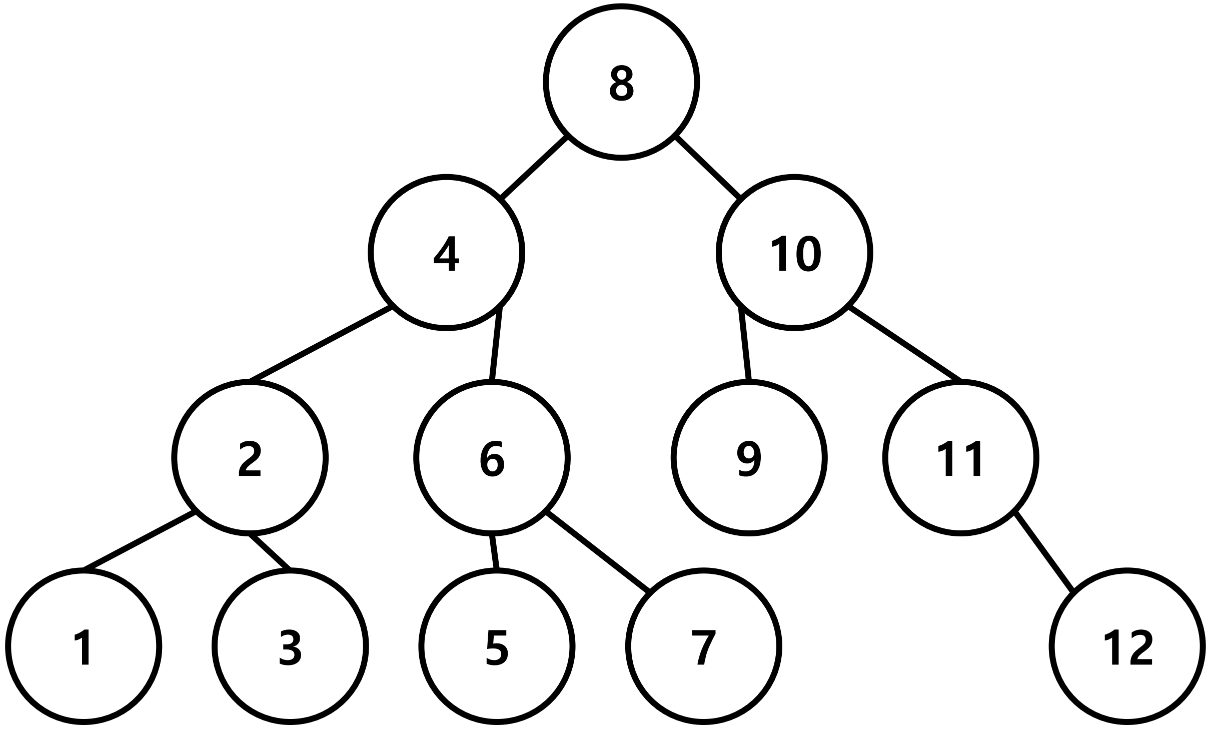
Lab5: BST

- Example
  - In original code, delete 4
  - Find minimum value in the right subtree
  - → 5
  - Replace 4 with 5
  - And delete 4



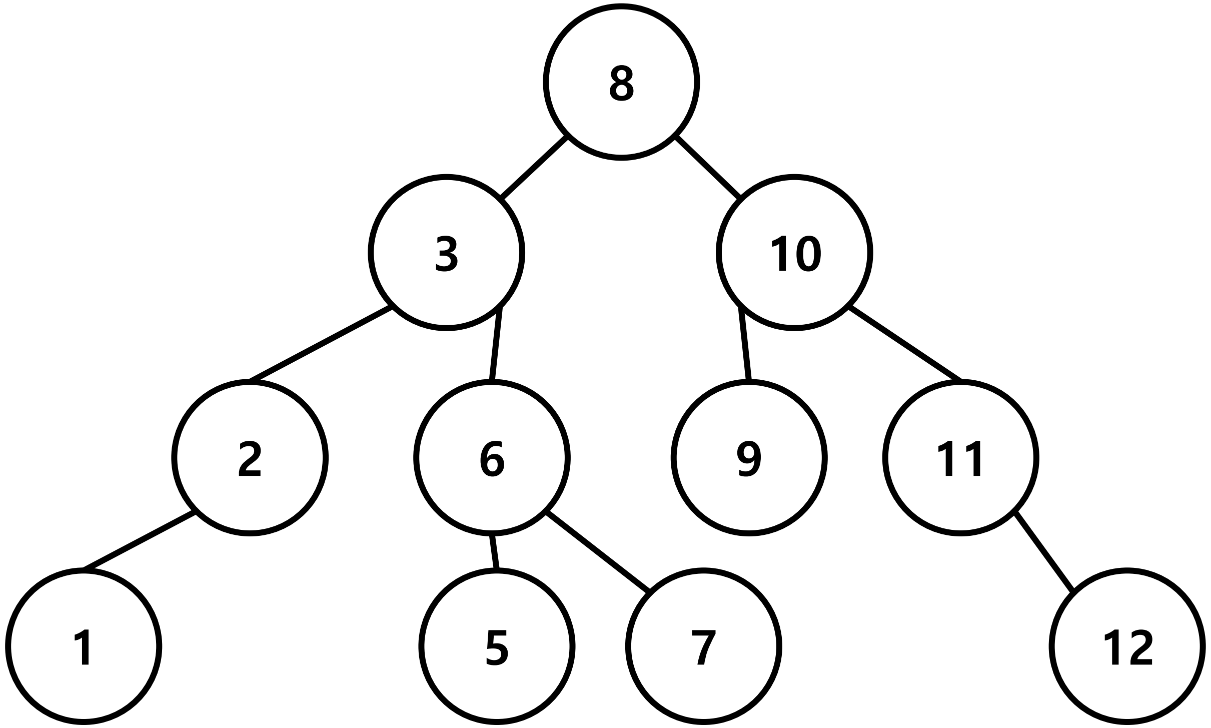
Lab5: BST

- Example
  - Change to
  - When delete 4
  - Find maximum value in the left subtree
  - → 3
  - Replace 4 with 3
  - And delete 4



Lab5: BST

- Example
  - Change to
  - When delete 4
  - Find maximum value in the left subtree
  - → 3
  - Replace 4 with 3
  - And delete 4


8
3
10
2
6
9
11
1
5
7
12