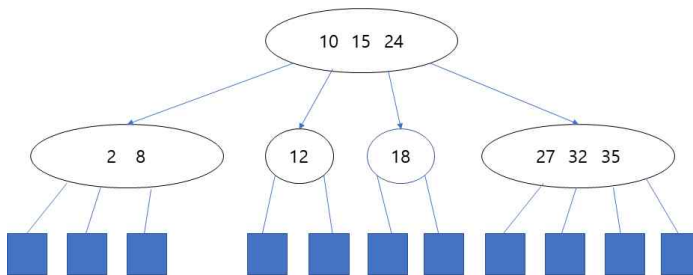


● Windows 10, Dev-C++ 5.11, TDM-GCC 4.9.2 64-bit Release 환경에서 작성하였다.

문제 1 : 2-4 Tree 구현

1. 2-4 Tree의 구현 방법 및 코드

1-1 2-4 Tree의 구조



2-4 Tree는 다음과 같은 성질을 만족해야 한다.

- 모든 internal 노드는 최대 4개의 children 까지만 가질 수 있다.
- 모든 external 노드는 같은 depth를 가진다.

1-1-1 2-4 tree 의 기본 골격 코드

2-4 tree는 한 노드에 최대 3개의 key, 최소 1개의 key, 최대 4개의 Leaf를 가지고 있어야 하므로, $\text{maxLeaf} = 4$ 로 지정하고, 최소로 가지고 있어야 하는 Entry 개수는 $(\text{maxLeaf}+1)/2 - 1$ 로 계산하여 minimum에 저장한다. ENTRY 노드는 데이터와 왼쪽, 오른쪽 노드를 가리키는 포인터를 가지고 있고, NODE 구조체는 entry 배열과 entry 개수를 저장한다. 그리고 최대 저장할 수 있는 entry의 개수는 $\text{maxLeaf}-1$ 이다. 2-4 Tree의 구조체는 root 노드를 포함하고, 전체 원소 개수를 저장하는 변수를 가진다.

```
const int maxLeaf = 4;
const int minimum = (((maxLeaf + 1)/2) - 1 );

struct node;

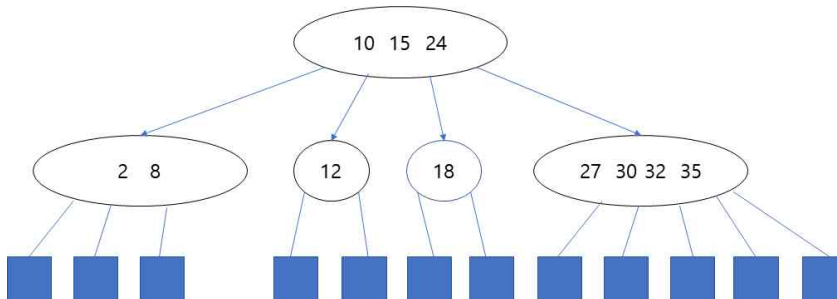
typedef struct{
    void* dataP;
    struct node* rightPtr;
    struct node* leftPtr;
}ENTRY;

typedef struct node{
    struct node* first;
    int numEntries;
    ENTRY entries[maxLeaf-1];
}NODE;

typedef struct{
    int count;
    NODE* root;
    int (*compare) (void* a1, void* a2);
}TFTREE;
```

1-2 Insertion

2-4 tree에 요소를 집어넣는 방법은 binary tree에서 요소를 집어 넣는 방법과 근본적으로 동일하다. 기존에 tree에 담겨 있는 요소들과 크기를 비교한 다음, 집어 넣을 위치를 결정하는 방식이다. 30을 아래의 트리에 삽입한다고 하자.

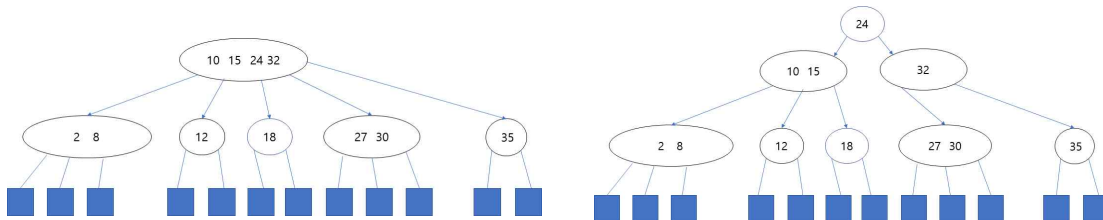


- 1) 30은 24보다 크기 때문에 가장 오른쪽 화살표를 따라서 이동한다.
- 2) 30은 27보다는 크고 32보다는 작기 때문에 27과 32 사이에 삽입한다.

이렇게 삽입하면 external 노드의 depth는 유지되지만, internal 노드는 최대 4개의 자식만 가질 수 있다 (한 노드는 최대 3개의 요소만 가질 수 있다.)를 위반하게 된다. 이 경우를 overflow라고 하며, overflow를 해결하는 방법은 overflow가 발생한 노드를 2개의 노드로 나누는 것이다. (split) split을 하는 방법은 다음과 같다.

- 1) overflow가 발생한 노드의 key 중 하나를 부모 노드로 보낸다. k3
- 2) overflow가 발생했던 노드를 3node와 2node로 분할한다. (k1,k2/k4) (1)

위 과정을 따르면 split의 결과는 왼쪽 그림과 같다. 이 경우, root 노드에서 overflow가 발생하게 된다. 이 경우에도 split을 똑같이 적용해 주면 오른쪽 그림처럼 되고, 2-4 tree의 property를 만족하게 된다.



1-2-1 Insertion 및 split을 수행하는 함수의 코드 - TFTree_Insert, insert, SplitNode

- 1) TFTree_Insert : Interface 함수
- 2) insert 함수 : Tree에 데이터를 삽입

TFTree_Insert 함수를 통해 TFTree 구조체에 데이터를 넣는데, TFTree_Insert 함수는 insert 함수를 이용하여 tree에 새 노드를 삽입한다. insert 함수는 우선 노드를 tree에 삽입한 후, 알맞은 위치에 들어갈 수 있도록 위치를 조절한다. 한도를 초과하면 split을 실행한다.

```
void TFTree_Insert (TFTree* tree, void* dataInP)
{
    bool taller;
    NODE* newPtr;
    ENTRY upEntry;

    if(tree->root == NULL)
    {
        if(newPtr = (NODE*)malloc(sizeof(NODE)))
        {
            newPtr->first = NULL;
            newPtr->numEntries = 1;
            newPtr->entries[0].dataP = dataInP;
            newPtr->entries[0].rightPtr = NULL;
            tree->root = newPtr;
            (tree->count)++;

            for(int i=1; i<maxleaf-1; i++)
            {
                newPtr->entries[i].dataP = NULL;
                newPtr->entries[i].rightPtr = NULL;
            }
            return;
        }
        else
            exit(100);
    }

    taller = insert(tree, tree->root, dataInP, &upEntry);
    if(taller)
    {
        newPtr = (NODE*)malloc(sizeof(NODE));
        if(newPtr)
        {
            newPtr->entries[0] = upEntry;
            newPtr->first = tree->root;
            newPtr->numEntries = 1;
            tree->root = newPtr;
        }
        else
            exit(100);
    }

    (tree->count)++;
    return;
}

//2. internal insert function
bool insert(TFTree* tree, NODE* root, void* dataInP, ENTRY* upEntry)
{
    int compResult;
    int entryN;
    bool taller;

    NODE* subtreePtr;

    if(!root)
    {
        (*upEntry).dataP = dataInP;
        (*upEntry).rightPtr = NULL;
        return true;
    }

    entryN = searchNode(tree, root, dataInP);
    compResult = tree->compare(dataInP, root->entries[entryN].dataP);

    if(entryN <= 0 && compResult < 0)
    {
        subtreePtr = root->first;
    }
    else
    {
        subtreePtr = root->entries[entryN].rightPtr;
        taller = insert(tree, subtreePtr, dataInP, upEntry);
    }

    if(taller)
    {
        if(root->numEntries >= maxleaf-1)
        {
            splitNode(root, entryN, compResult, upEntry);
            taller = true;
        }
        else
        {
            if(compResult >= 0)
                InsertKey(root, entryN+1, *upEntry);
            else
                InsertKey(root, entryN, *upEntry);
            (root->numEntries)++;
            taller = false;
        }
    }

    return taller;
}
```

3) SplitNode : Split을 수행하는 함수로, 노드가 짝 차면 노드를 2개의 노드로 나눈다.

```
void splitNode (NODE* node, int entryN, int compResult, ENTRY* upEntry)
{
    int fromNdx;
    int toNdx;
    NODE* rightPtr;

    rightPtr = (NODE*)malloc(sizeof(NODE));
    if(!rightPtr)
        exit(100);

    if(entryN < minimum)
        fromNdx = minimum;
    else
        fromNdx = minimum + 1;
    toNdx = 0;
    rightPtr->numEntries = node->numEntries - fromNdx;
    while(fromNdx < node->numEntries)
        rightPtr->entries[toNdx++] = node->entries[fromNdx++];
    node->numEntries = node->numEntries - rightPtr->numEntries;

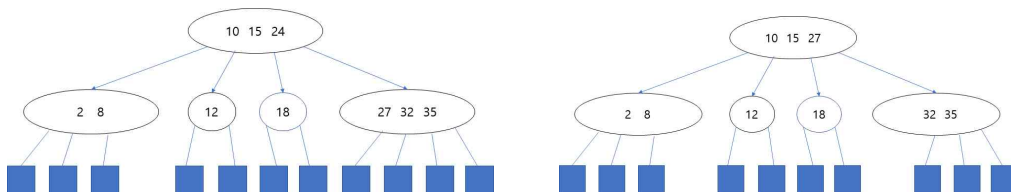
    if(entryN < minimum)
    {
        if(compResult < 0)
            insertKey(node, entryN, *upEntry);
        else
            insertKey(node, entryN+1, *upEntry);
    }
    else
    {
        insertKey(rightPtr, entryN - minimum, *upEntry);
        (rightPtr->numEntries)++;
        (node->numEntries)--;
    }

    upEntry->dataP = node->entries[minimum].dataP;
    upEntry->rightPtr = rightPtr;
    rightPtr->first = node->entries[minimum].rightPtr;

    return;
}
```

1-3 Deletion

2-4 Tree의 Deletion 방법은 Binary search tree에서의 삭제 방법과 근본적으로 같은 방법이다. 어떤 key를 찾는다고 하고, key와 tree 내 요소의 대소 비교를 통해서 그 key에 도달하면, 그 key를 지우는 방식이다. key를 지우게 되면, 현재 노드의 key 개수와 children 개수가 맞지 않게 된다. 이 경우, 삭제한 key의 inorder successor를 그 위치에 불러 온다. 왼쪽 tree에서 24를 삭제한다고 하자. 24를 삭제하면 root 노드의 key 개수는 2개인데 반해, children의 개수는 4개인 상태이다. 이 경우, 24의 다음 key인 27을 root 노드에 불러온 다음, 가장 오른쪽 child 노드를 3node로 변경해 주면 2-4 tree의 property를 만족하게 된다.

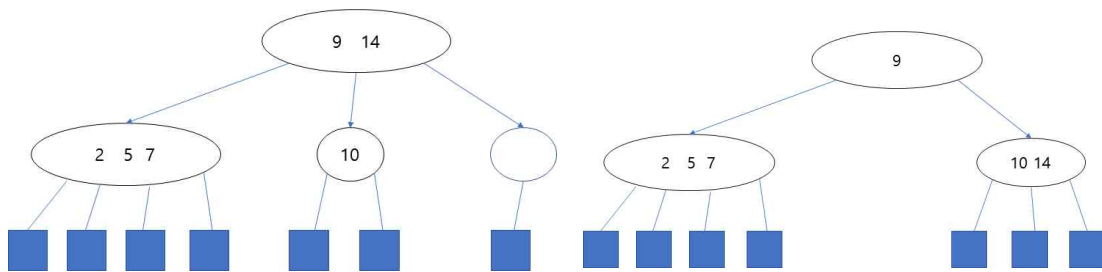


그러나 만약 2-node의 key를 삭제했을 경우, key는 없고, child는 1개밖에 없는 노드가 발생할 수 있다. 2-4 tree의 노드는 최소 2개의 노드를 children으로 가져야 하므로, 이 경우는 2-4 tree의 property를 위반하는 것이 된다. (**underflow**) underflow를 해결하는 방식은 underflow가 발생한 노드의 형제 노드가 어떤지에 따라 달라진다.

1) 1-node의 형제 노드가 2-node인 경우 => Fusion

이 경우, 1-node와 2-node를 합친다. 두 노드를 합치면 key는 1개, child는 3개인 상태가 된다. 또한, 부모 노드에서도 key와 children 수가 맞지 않는 문제가 발생한다. 이를 해결하기 위해서는 부모 노드에서 key를 1개 가지고 오면 된다.

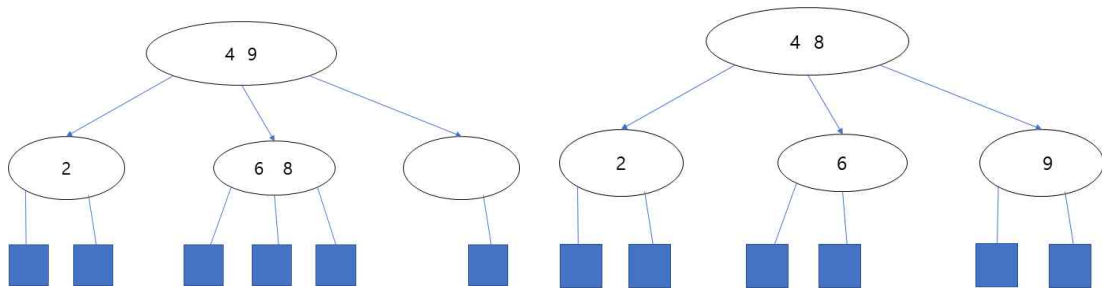
아래 그림에서는 1node가 발생한 경우, 10을 가진 노드와 1node를 합치고, 부모 노드로부터 14를 받아왔다. Fusion을 완료한 결과는 오른쪽 그림과 같다.



2) 1-node의 형제 노드가 3-node 이상인 경우 => Transfer

이 경우, 3,4-node로부터 leaf 하나를 1-node에 붙인다. 그 다음, 1-node의 부모로부터 key를 하나 받아온다. 이렇게 되면 1-node는 2-node가 되어 성질을 위반하지 않지만, 형제 노드는 leaf가 부족하게 되고, 부모 노드는 key가 부족하게 된다. 이를 해결하기 위해서 형제 노드의 key 하나를 부모 노드로 옮긴다.

아래 그림에서는 9를 1-node로 보내고 8을 부모 노드로 보냈다. Transfer를 완료한 결과는 오른쪽 그림과 같다.



1-3-1 Deletion 및 Fusion/Transfer를 수행하는 함수의 코드

TFTree_Delete, _delete, deleteMidKey, reFlow, transferLeft, transferRight, Fusion

1) TFTree_Delete : Interface 함수

2) _delete : Tree의 데이터를 leaf를 삭제하는 방식으로 삭제

3) deleteMidKey : internal 노드의 데이터를 삭제

TFTree_Delete 함수를 통해 TFTree 구조체에서 데이터를 삭제하는데, TFTree_Delete 함수는 _delete 함수를 이용하여 tree에서 노드를 삭제한다. _delete 함수는 삭제하는 노드에 따라 삭제 방법을 다르게 선택한다. 노드를 삭제하고, underflow가 생기지 않도록 key의 위치를 조절한다.

```
bool TFTree_Delete(TFTREE* tree, void* deletedKey)
{
    bool success;
    NODE* dltPtr;

    if(!tree->root)
        return false;

    _delete (tree, tree->root, deletedKey, &success);

    if(success)
    {
        (tree->count)--;
        if(tree->root->numEntries == 0)
        {
            dltPtr = tree->root;
            tree->root = tree->root->first;
            free(dltPtr);
        }
    }

    return success;
}
```

```

bool _delete(TFTREE* tree, NODE* root, void* deletedKeyP, bool* success)
{
    NODE* leftPtr;
    NODE* subTreePtr;
    int entryN;
    int underflow;

    if(!root)
    {
        *success = false;
        return false;
    }

    entryN = searchNode(tree, root, deletedKeyP);
    if(tree->compare(deletedKeyP, root->entries[entryN].dataP) == 0)
    {
        *success = true;
        if(root->entries[entryN].rightPtr == NULL)
            underflow = deleteKey(root, entryN);
        else
        {
            if(entryN > 0)
                root->entries[entryN - 1].rightPtr;
            else
                leftPtr = root->first;
            underflow = deleteMidKey(root, entryN, leftPtr);
            if(underflow)
                underflow = reFlow(root, entryN);
        }
    }
    else
    {
        if(tree->compare(deletedKeyP, root->entries[0].dataP) < 0)
            subTreePtr = root->first;
        else
            subTreePtr = root->entries[entryN].rightPtr;

        underflow = _delete(tree, subTreePtr, deletedKeyP, success);

        if(underflow)
            underflow = reFlow(root, entryN);
    }
    return underflow;
}

bool deleteMidKey (NODE* root, int entryN, NODE* subtreePtr)
{
    int dltNdx;
    int rightNdx;
    bool underflow;

    if(subtreePtr->first == NULL)
    {
        dltNdx = subtreePtr->numEntries - 1;
        root->entries[entryN].dataP = subtreePtr->entries[dltNdx].dataP;
        --subtreePtr->numEntries;
        underflow = subtreePtr->numEntries < minimum;
    }
    else
    {
        rightNdx = subtreePtr->numEntries - 1;
        underflow = deleteMidKey(root, entryN, subtreePtr->entries[rightNdx].rightPtr);

        if(underflow)
            underflow = reflow(subtreePtr, rightNdx);
    }
    return underflow;
}

```

underflow를 처리해주는 함수에는 4)reFlow, 5)transferLeft, 6)transferRight, 7)fusion이 있다. reFlow 함수는 나머지 3개의 함수를 포함하며, _delete 함수에서 reFlow가 실행될 때, 상황에 맞는 underflow 처리를 실행한다. transfer 함수는 이웃 노드가 3,4 노드인 경우, fusion함수는 이웃 노드가 2 노드인 경우 실행한다.

```

void transferLeft(NODE* root, int entryN, NODE* leftTreePtr, NODE* rightTreePtr)
{
    int toNdx;
    int shifter;

    toNdx = rightTreePtr->numEntries;
    rightTreePtr->entries[toNdx].dataP = root->entries[entryN].dataP;
    rightTreePtr->entries[toNdx].leftPtr = leftTreePtr->first;
    ++leftTreePtr->numEntries;

    root->entries[entryN].dataP = leftTreePtr->entries[0].dataP = leftTreePtr->entries[0].dataP;
    leftTreePtr->first = leftTreePtr->entries[0].leftPtr;
    shifter = 0;

    while(shifter < leftTreePtr->numEntries - 1)
    {
        leftTreePtr->entries[shifter] = leftTreePtr->entries[shifter + 1];
        ++shifter;
    }
    --leftTreePtr->numEntries;
    return;
}

void transferRight(NODE* root, int entryN, NODE* leftTreePtr, NODE* rightTreePtr)
{
    int toNdx;
    int shifter;

    toNdx = leftTreePtr->numEntries;
    leftTreePtr->entries[toNdx].dataP = root->entries[entryN].dataP;
    leftTreePtr->entries[toNdx].rightPtr = rightTreePtr->first;
    ++leftTreePtr->numEntries;

    root->entries[entryN].dataP = rightTreePtr->entries[0].dataP = rightTreePtr->entries[0].dataP;
    rightTreePtr->first = rightTreePtr->entries[0].rightPtr;
    shifter = 0;

    while(shifter < rightTreePtr->numEntries - 1)
    {
        rightTreePtr->entries[shifter] = rightTreePtr->entries[shifter + 1];
        ++shifter;
    }
    --rightTreePtr->numEntries;
    return;
}

```


- Right Rotate를 할 때는 왼쪽 자식 노드의 오른쪽 자식 노드를 부모 노드의 왼쪽 자식으로 연결한다.
- Left Rotate를 할 때는 오른쪽 자식 노드의 왼쪽 자식 노드를 부모 노드의 오른쪽 자식으로 연결한다.

1-2-1 회전을 구현하는 함수의 코드 - RBT_RotateRight, RBT_RotateLeft

오른쪽 회전의 코드에 대한 설명만 간단히 하자면, 왼쪽 자식 노드의 오른쪽 자식 노드를 부모 노드의 왼쪽 자식 노드로 등록하고, 1) 만약 부모가 NULL이면 이 노드는 Root이고, 이 경우는 왼쪽 자식을 Root로 만드는 회전을 한다. 2) 부모가 NULL이 아닌 경우는 왼쪽 자식 노드를 부모 노드가 있던 곳(조부모의 자식 노드)에 위치시킨다. 왼쪽, 오른쪽 회전의 구현은 근본적으로 같은 방법이다.

```
void RBT_RotateRight( RBTNode** Root, RBTNode* Parent )
{
    RBTNode* LeftChild = Parent->Left;
    Parent->Left = LeftChild->Right;

    if ( LeftChild->Right != Nil )
        LeftChild->Right->Parent = Parent;

    LeftChild->Parent = Parent->Parent;

    if ( Parent->Parent == NULL )
        (*Root) = LeftChild;
    else
    {
        if ( Parent == Parent->Parent->Left )
            Parent->Parent->Left = LeftChild;
        else
            Parent->Parent->Right = LeftChild;
    }

    LeftChild->Right = Parent;
    Parent->Parent = LeftChild;
}

void RBT_RotateLeft( RBTNode** Root, RBTNode* Parent )
{
    RBTNode* RightChild = Parent->Right;
    Parent->Right = RightChild->Left;

    if ( RightChild->Left != Nil )
        RightChild->Left->Parent = Parent;

    RightChild->Parent = Parent->Parent;

    if ( Parent->Parent == NULL )
        (*Root) = RightChild;
    else
    {
        if ( Parent == Parent->Parent->Left )
            Parent->Parent->Left = RightChild;
        else
            Parent->Parent->Right = RightChild;
    }

    RightChild->Left = Parent;
    Parent->Parent = RightChild;
}
```

1-3 Red-Black Tree의 연산 - 삽입

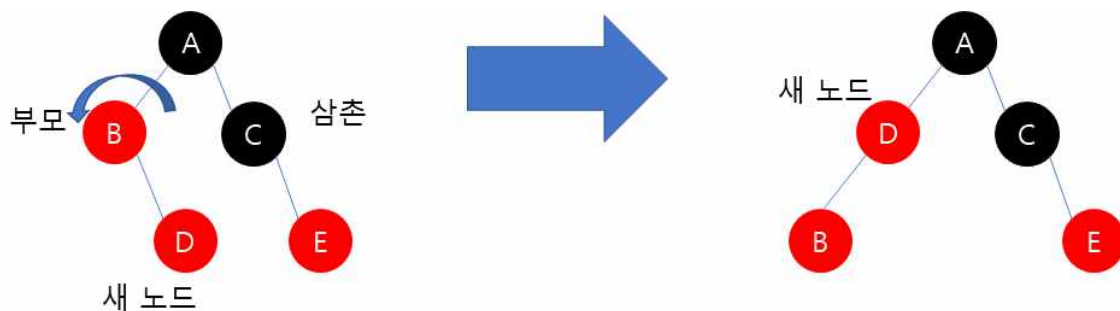
Red-Black Tree에 새 노드가 삽입되고 나면, 일단 새로 삽입된 노드를 Red로 칠한다. 그리고 양쪽 자식에 Black인 빈 노드를 연결해 준다.

새 노드를 Tree에 넣고 나서 'Root 노드는 Black이어야 한다' 와 'Red 노드의 자식들은 모두 Black이다' 가 위배되는지 확인해야 한다.

- Root 노드의 성질은 무조건 Root 노드를 Black으로 칠하는 방법을 통해 유지할 수 있다.
- 'Red 노드의 자식들은 모두 Black'이어야 한다는 성질은 부모 노드가 Red이고, 새로 삽입된 노드가 Red인 경우 위배될 수 있다. (Double Red) 이런 경우는 3가지가 있다.

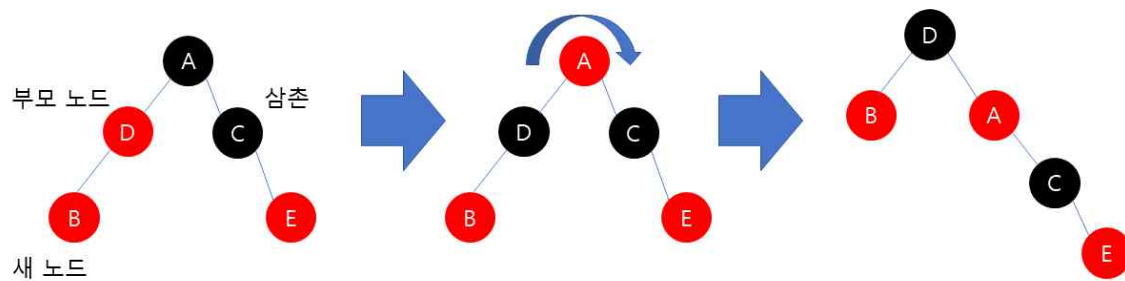
① 부모 노드의 형제 노드가 Black이고, 자식 노드가 오른쪽인 경우 - Reconstructing

: 부모 노드를 왼쪽으로 회전시킨 다음, ②번 문제의 경우로 생각한다.



② 부모 노드의 형제 노드가 Black이고, 자식 노드가 왼쪽인 경우 - Reconstructing

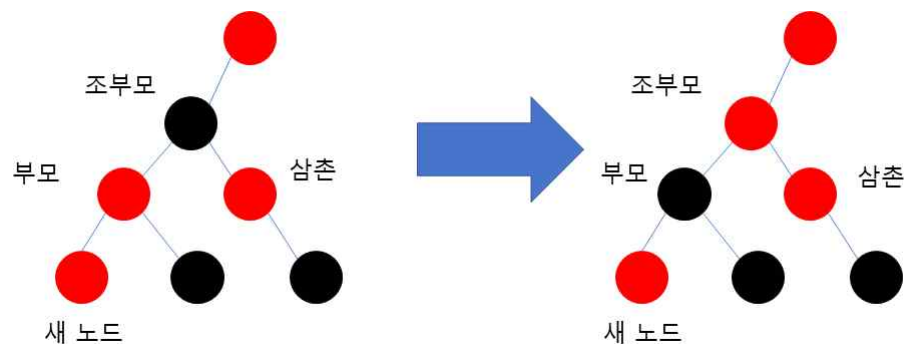
: 부모 노드를 Black, 조부모 노드를 Red로 칠한 다음, 조부모 노드를 오른쪽으로 회전시킨다.



③ 부모 노드의 형제 노드도 Red인 경우 - Recoloring

: 부모 노드와 삼촌 노드를 Black으로 칠하고, 조부모 노드를 Red로 칠한다.

Recoloring을 수행한 후, 'Red 노드의 자식은 모두 Black이어야 한다는 규칙이 위배될 수 있다. 이 경우, 조부모 노드를 새로 삽입한 노드로 간주하고, 앞의 ①, ②, ③을 위배하는지 다시 살펴보아야 한다. 이 확인 과정은 계속 위로 올라가며 부모 노드가 Black이거나, 새로 삽입한 노드가 Root여야만 끝난다.



1-3-1 노드를 삽입하는 함수의 코드 - RBT_InsertNode, RBT_InsertNodeHelper

1) RBT_InsertNode 함수는 만약 이미 키가 존재하면, key가 이미 존재한다는 문구를 출력한다. 삽입하려고 하는 key가 tree에 존재하지 않는다면, RBT_InsertNodeHelper 함수를 이용하여 노드를 Tree에 삽입한다.

2) RBT_InsertNodeHelper는 재귀적으로 작동하여 key가 기준 노드보다 작으면 왼쪽으로, 기준 노드보다 크면 오른쪽에 연결한다. 연결한 후, 새 노드를 Red로 칠하고, 자식을 NULL로 지정한다.

```
string RBT_InsertNode( RBTNode** Tree, RBTNode* NewNode)
{
    string output;
    for(int i=0; i<v.size(); i++){
        if(v[i] == NewNode->Data){
            output += IntToString(NewNode->Data);
            output += " ";
            output += "ALREADY EXIST";
            output += "\n";
            return output;
        }
    }
    RBT_InsertNodeHelper( Tree, NewNode ); //key가 표준형
    v.push_back(NewNode->Data);

    NewNode->Color = 1;
    NewNode->Left = Nil;
    NewNode->Right = Nil;
    RBT_RebuildAfterInsert( Tree, NewNode );
    return "";
}
```

```

void RBT_InsertNodeHelper( RBTNode** Tree, RBTNode* NewNode)
{
    if ( (*Tree) == NULL )
        (*Tree) = NewNode;

    if ( (*Tree)->Data < NewNode->Data ) //부모 노드 값이 작을 경우
    {
        if ( (*Tree)->Right == Nil )
        {
            (*Tree)->Right = NewNode;
            NewNode->Parent = (*Tree);
        }
        else
            RBT_InsertNodeHelper(&(*Tree)->Right, NewNode); //공의 자식인 채 균형적으로 삽입
    }
    else if ( (*Tree)->Data > NewNode->Data ) //부모 노드 값이 큰 경우
    {
        if ( (*Tree)->Left == Nil )
        {
            (*Tree)->Left = NewNode;
            NewNode->Parent = (*Tree);
        }
        else
            RBT_InsertNodeHelper(&(*Tree)->Left, NewNode); //공의 자식인 채 균형적으로 삽입
    }
}

```

1-3-2 Insert 후 Tree의 구조를 바꾸는 함수의 코드 - RBT_RebuildAfterInsert

4번 규칙을 위반하고 있는 동안에는 이 함수를 계속 반복한다. 부모 노드가 조부모 노드의 왼쪽 자식인 경우에, 1) 삼촌이 RED인 경우-Recoloring, 2) 삼촌이 BLACK이고 현재 노드가 오른쪽 자식인 경우-Reconstructing 2가지로 나누어서 처리한다. 만약 부모 노드가 조부모 노드의 오른쪽 자식인 경우에는 위의 경우에서 왼쪽/오른쪽만 바꾸면 된다.

```

void RBT_RebuildAfterInsert( RBTNode** Root, RBTNode* X )
{
    while ( X != (*Root) && X->Parent->Color == 1 )
    {
        if ( X->Parent == X->Parent->Parent->Left ) //부모 노드가 조부모 노드의 왼쪽 자식인 경우
        {
            RBTNode* Uncle = X->Parent->Parent->Right;
            if ( Uncle->Color == 1 ) //Recoloring
            {
                X->Parent->Color = 0;
                Uncle->Color = 0;
                X->Parent->Parent->Color = 1;

                X = X->Parent->Parent;
            }
            else //reconstructing
            {
                if ( X == X->Parent->Right )
                {
                    X = X->Parent;
                    RBT_RotateLeft( Root, X );
                }

                X->Parent->Color = 0;
                X->Parent->Parent->Color = 1;

                RBT_RotateRight( Root, X->Parent->Parent );
            }
        }
        else // 부모 노드가 조부모 노드의 오른쪽 자식인 경우
        {
            RBTNode* Uncle = X->Parent->Parent->Left;
            if ( Uncle->Color == 1 ) //Recoloring
            {
                X->Parent->Color = 0;
                Uncle->Color = 0;
                X->Parent->Parent->Color = 1;

                X = X->Parent->Parent;
            }
            else //Reconstructing
            {
                if ( X == X->Parent->Left )
                {
                    X = X->Parent;
                    RBT_RotateRight( Root, X );
                }

                X->Parent->Color = 0;
                X->Parent->Parent->Color = 1;
                RBT_RotateLeft( Root, X->Parent->Parent );
            }
        }
    }

    (*Root)->Color = 0;
}

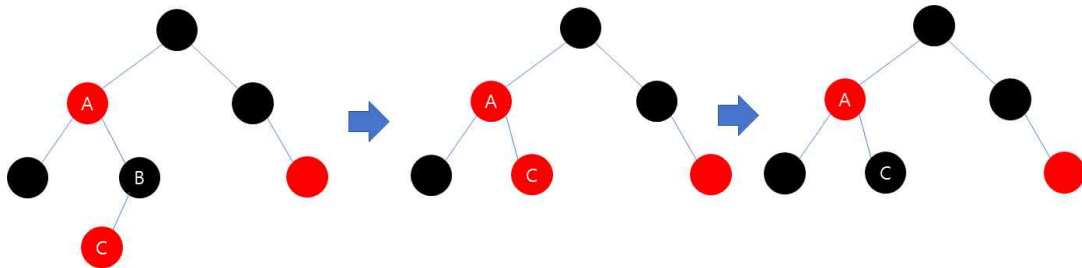
```

1-4 Red-Black Tree의 연산 - 삭제

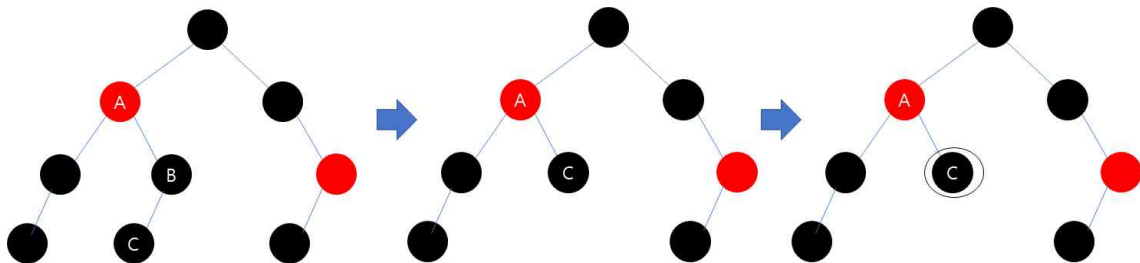
Red-Black Tree에서는 Red 또는 Black 노드가 삭제된다. Red가 삭제되는 경우는 추가적인 작업이 필요 없다. Red-Black tree의 성질을 해치는 경우가 없기 때문이다. 1. Red 노드를 삭제한다고 해서 다른 노드의 색이 바뀌지 않고, 2.3. Root 노드와 Leaf 노드의 색은 항상 Black으로 유지되고, 4.Red 노드의 자식들은 원래 Black이었고, 5. Red 노드를 삭제해도 Black depth는 유지된다.

따라서 삭제 연산에서는 Black 노드를 삭제하는 경우에만 뒤풀이를 해 주면 된다.

Black 노드를 삭제하는 경우에, 5. Black depth가 가장 먼저 위배된다. 그리고 삭제한 노드의 부모 노드와 자식 노드가 Red인 경우에, 4. Red 노드의 자식은 Black이라는 규칙도 위배된다. 위배된 4,5 번 규칙을 한 번에 고치기 위해서는 삭제된 노드를 대체하는 노드를 Black으로 칠하면 된다.



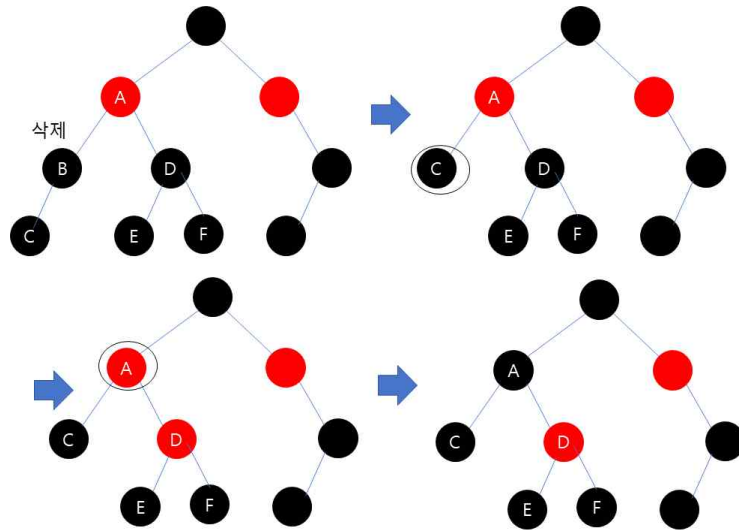
만약 C노드가 원래 Black인 경우에는, 4번 규칙은 위반하지 않고, Black depth만 위반하게 된다. 이 경우에도 마찬가지로, 삭제된 노드 C를 Black으로 한번 더 칠한다. (Double Black) 이렇게 해서 5번 규칙을 복원할 수 있다.



이렇게 하면 다시 1번 규칙인 '모든 노드는 Red 또는 Black'이 위배된다. Black이 두 번 칠해진 노드가 생겼기 때문이다. (실제로는 Black Depth가 고쳐지지 않은 상태이다. 1번 규칙이 위배되었다고 생각하고 문제를 해결하는 것이 더 편리하다.)

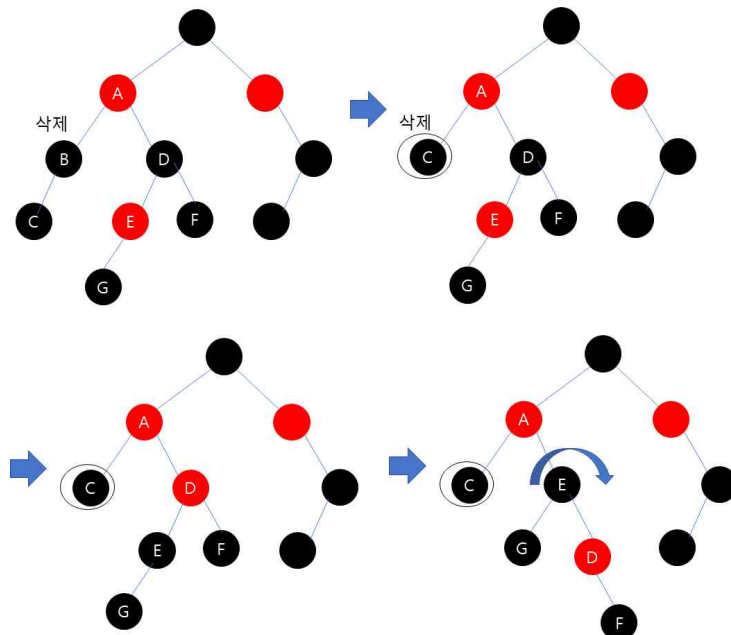
Double Black을 해결하는 방법은 Double Black 노드의 형제와 형제의 자식의 상태에 따라 4가지로 구분된다. 아래 그림의 설명은 Double Black 노드가 부모 노드의 왼쪽 자식인 경우에 대한 것이다. 만약 오른쪽에 Double Black 노드가 달려 있다면 아래 설명의 좌우를 바꾸기만 하면 된다. (2)

① 형제 노드가 Black이고 형제 노드의 자식이 모두 Black인 경우



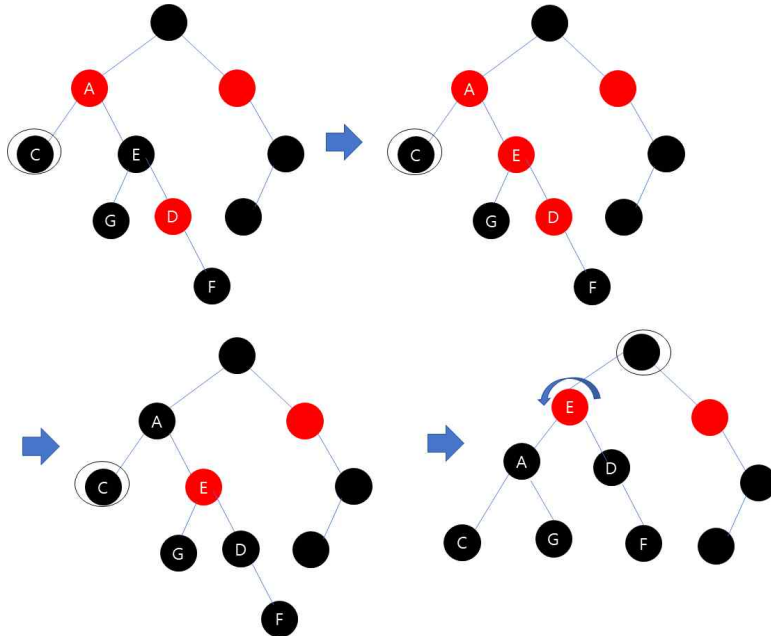
=> Double Black 노드의 형제가 검은색이고 형제의 양쪽 자식이 모두 Black이면, 형제 노드만 Red로 칠한 다음 Double Black 노드의 자식 중 하나를 부모 노드에게 주면 된다.

② 형제 노드가 Black이고 형제의 왼쪽 자식이 Red, 오른쪽 자식은 Black인 경우



=> 이 경우에는 형제 노드를 Red로 칠하고, Red였던 자식을 Black으로 칠한 다음, 색이 변한 자식이 위로 가도록 회전을 수행한다. 좌우가 바뀐 경우는 왼쪽/오른쪽만 바꿔주면 성립한다.

③ 형제 노드가 Black이고 형제의 오른쪽 자식이 Red인 경우

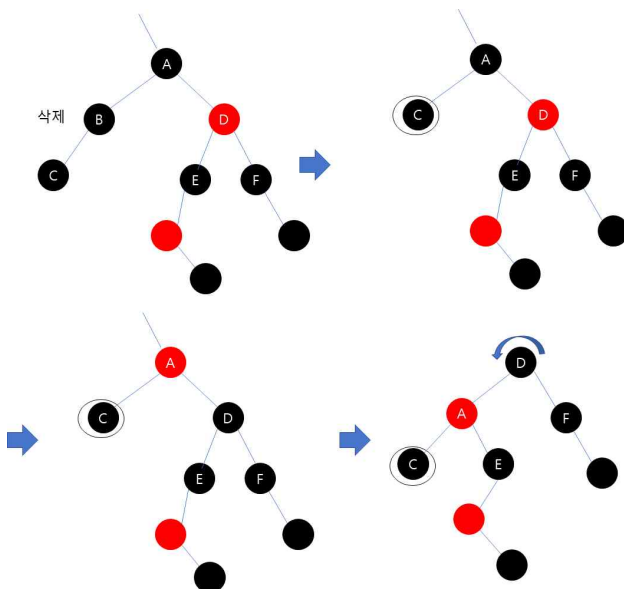


=> Double Black의 부모 노드의 색을 형제 노드에 칠하고, 부모 노드와 형제 노드의 오른쪽 자식 노드를 Black으로 칠한 다음, 부모 노드를 기준으로 좌회전한다. 좌우가 바뀐 경우에는 왼쪽/오른쪽만 바꿔주면 성립한다. 또한 이 방법은 다른 하나의 자식의 색과 관계 없이 적용 가능하다. 즉, 자식이 둘 다 Red인 경우에도 사용 가능하다. (2)

④ 형제 노드가 Red인 경우

형제 노드를 Black, 부모 노드를 Red로 칠한 다음, 부모 노드를 기준으로 좌회전한다. (Double Black이 왼쪽 child인 경우이다. 만약 Double Black이 오른쪽 child이면, 우회전을 하면 된다. (3))

이렇게 하면 Double Black이 여전히 남아 있게 되지만, 문제의 경우가 ①,②,③ 의 경우로 바뀐다. 좌우가 바뀌면 연산의 방향만 반대로 해 주면 된다.



1-4-1 노드를 삭제하는 함수의 코드 - RBT_RemoveNode

삭제하려는 노드가 없으면, NULL을 반환한다. 삭제하려는 노드의 왼쪽 또는 오른쪽이 LEAF 이면, LEAF와 함께 그 노드를 지운다. 아닌 경우에는, SearchMinNode 함수를 이용하여 그 노드의 inorder successor의 key를 지우려고 하는 노드에 넣어서 key를 교체한다. 그 다음 inorder successor를 원래 가지고 있던 노드를 삭제한다. (1)

```
RBTNode* RBT_RemoveNode( RBTNode** Root, ElementType Data )
{
    RBTNode* Removed = NULL;
    RBTNode* Successor = NULL;
    RBTNode* Target = RBT_SearchNode( *Root, Data );

    if ( Target == NULL ) //삭제하려는 노드가 없을 때
        return NULL;

    if ( Target->Left == Nil || Target->Right == Nil ) //삭제하려는 노드가 LEAF를 가지는 경우
    {
        Removed = Target;
    }
    else //LEAF가 없는 경우, inorder successor를 불러 온다.
    {
        Removed = RBT_SearchMinNode( Target->Right );
        Target->Data = Removed->Data;
    }

    //inorder successor를 원래 가지고 있던 노드를 처리한다.

    if ( Removed->Left != Nil )
        Successor = Removed->Left;
    else
        Successor = Removed->Right;

    Successor->Parent = Removed->Parent;

    if ( Removed->Parent == NULL )
        (*Root) = Successor;
    else
    {
        if ( Removed == Removed->Parent->Left )
            Removed->Parent->Left = Successor;
        else
            Removed->Parent->Right = Successor;
    }

    if ( Removed->Color == 0 )
        RBT_RebuildAfterRemove( Root, Successor );

    for(int i=0; i<v.size(); i++){
        if(v[i] == Data){
            v.erase(v.begin()+i);
        }
    }

    return Removed;
}
```


1-4-2 노드 삭제 후 Tree의 구조를 바꾸는 함수의 코드 - RBT_RebuildAfterRemove

현재 노드가 Root 노드이거나 Red 노드에게 Black 하나가 넘어가면 함수가 종료된다. 함수는 1) 형제가 빨간색인 경우, 2-1) 형제가 검은색인 경우 양쪽 자식이 모두 검은색인 경우 2-2) 형제가 Black인 경우 왼쪽이 Red인, 오른쪽이 Black인 경우 2-3) 형제가 Black인 경우 오른쪽 자식이 Red인 경우(둘 다 Red인 경우도 포함) 의 4가지로 나누어 실행한다.

```
void RBT_RebuildAfterRemove( RBTNode** Root, RBTNode* Successor )
{
    RBTNode* Sibling = NULL;

    while ( Successor->Parent != NULL && Successor->Color == 0 )
    {
        // 현재 노드가 root가 아니거나 black이 넘어가기 전 까지 함수를 반복
        if ( Successor == Successor->Parent->Left ) // Double Black이 왼쪽 노드인 경우
        {
            Sibling = Successor->Parent->Right;

            if ( Sibling->Color == 1 ) // 1) 형제가 Red인 경우
            {
                Sibling->Color = 0;
                Successor->Parent->Color = 1;
                RBT_RotateLeft( Root, Successor->Parent );
            }
            else // 형제가 Black인 경우
            {
                if ( Sibling->Left->Color == 0 &&
                     Sibling->Right->Color == 0 ) // 2-1) 자식이 모두 Black인 경우
                {
                    Sibling->Color = 1;
                    Successor = Successor->Parent;
                }
                else // 자식 중 Red가 있는 경우
                {
                    if ( Sibling->Left->Color == 1 ) // 2-2) 왼쪽 자식이 Red인 경우
                    {
                        Sibling->Left->Color = 0;
                        Sibling->Color = 1;

                        RBT_RotateRight( Root, Sibling );
                        Sibling = Successor->Parent->Right;
                    }
                    // 2-3) 오른쪽 자식이 Red인 경우
                    Sibling->Color = Successor->Parent->Color;
                    Successor->Parent->Color = 0;
                    Sibling->Right->Color = 0;
                    RBT_RotateLeft( Root, Successor->Parent );
                    Successor = (*Root);
                }
            }
        }
        else // Double Black이 오른쪽 노드인 경우
        {
            Sibling = Successor->Parent->Left;

            if ( Sibling->Color == 1 ) // 1) 형제가 Red인 경우
            {
                Sibling->Color = 0;
                Successor->Parent->Color = 1;
                RBT_RotateRight( Root, Successor->Parent );
            }
            else // 형제가 Black인 경우
            {
                if ( Sibling->Right->Color == 0 &&
                     Sibling->Left->Color == 0 ) // 2-1) 자식이 모두 Black인 경우
                {
                    Sibling->Color = 1;
                    Successor = Successor->Parent;
                }
                else
                {
                    if ( Sibling->Right->Color == 1 ) // 2-2) 형제의 오른쪽 자식이 Red인 경우
                    {
                        Sibling->Right->Color = 0;
                        Sibling->Color = 1;

                        RBT_RotateLeft( Root, Sibling );
                        Sibling = Successor->Parent->Left;
                    }
                    // 2-3) 왼쪽 자식이 Red인 경우
                    Sibling->Color = Successor->Parent->Color;
                    Successor->Parent->Color = 0;
                    Sibling->Left->Color = 0;
                    RBT_RotateRight( Root, Successor->Parent );
                    Successor = (*Root);
                }
            }
        }
    }

    Successor->Color = 0;
}
```

1-5 Prob-2.cpp 의 나머지 주요 함수/변수 목록

typedef int ElementType : key의 type int를 ElementType으로 새로 정의한다.

vector<int> v : 넣은 키들을 저장하는 벡터

typedef struct tagRBTNode {} RBTNode : 노드를 나타내는 구조체. 구조체의 멤버로는 부모, 왼쪽, 오른쪽 자식을 가리키는 포인터 변수와 색깔 정보, key가 포함된다.

void RBT_DestroyTree(RBTNode* Tree) : 프로그램이 종료된 후 Tree의 메모리 공간 반환

RBTNode* RBT_CreateNode(ElementType NewData) : 새 노드를 생성

void RBT_DestroyNode(RBTNode* Node) : 노드에 할당된 메모리 공간 반환

RBTNode* RBT_SearchNode(RBTNode* Tree, ElementType Target) : Tree의 key 탐색.
Search와 관련된 함수는 Binary Tree의 탐색 방식을 사용한다.

```
RBTNode* RBT_SearchNode( RBTNode* Tree, ElementType Target )
{
    if ( Tree == Nil )
        return NULL;

    if ( Tree->Data > Target )
        return RBT_SearchNode ( Tree->Left, Target );
    else if ( Tree->Data < Target )
        return RBT_SearchNode ( Tree->Right, Target );
    else //같은 값을 찾으면
        return Tree;
}
```

RBTNode* RBT_SearchMinNode(RBTNode* Tree) : Tree의 최소 key 탐색. Inorder successor를 찾는 데 사용한다.

Search와 관련된 함수는 Binary Tree의 탐색 방식을 사용한다.

```
RBTNode* RBT_SearchMinNode( RBTNode* Tree )
{
    if ( Tree == Nil )
        return Nil;

    if ( Tree->Left == Nil )
        return Tree;
    else
        return RBT_SearchMinNode( Tree->Left );
}
```

string RBT_PrintTree(RBTNode* Node, int Depth, int BlackCount) : Tree의 내용을 출력하는 함수

- Tree를 읽으며 노드에 들어 있는 key를 출력할 때, indent를 depth만큼 출력한 후, key를 출력하고, (색깔)을 출력한다. 현재 노드를 출력한 다음, 이 함수를 재귀적으로 실행해서 이 노드의 자식 노드도 출력한다.
- 현재 출력하는 자식 노드가 LEAF 노드인 경우가 있다. 이 경우는 3가지 경우로 처리한다.
 - 1) 자식이 둘 다 LEAF인 경우는 depth에 맞게 indent를 준 후, LEAF (BLACK)를 2줄 출력한다.
 - 2) 왼쪽 자식만 LEAF인 경우, LEAF (BLACK)을 먼저 출력한 후, 오른쪽 자식에 대해 이 함수를 재귀적으로 실행한다.
 - 3) 오른쪽 자식만 LEAF인 경우, 왼쪽 자식에 대해 이 함수를 재귀적으로 실행한 후, LEAF (BLACK)을 출력한다.

```

string RBT_PrintTree( RBTNode* Node, int Depth, int BlackCount )
{
    if(v.empty())
        return "NO KEYS\n";

    string output;
    int i = 0;
    string c = "X";
    int v = -1;
    string cnt;

    if ( Node == NULL || Node == Nil )
        return "";

    if ( Node->Color == 0 )
        BlackCount++;

    if ( Node->Parent != NULL )
    {
        v = Node->Parent->Data;

        if ( Node->Parent->Left == Node )
            c = 'L';
        else
            c = 'R';
    }

    if ( Node->Left == Nil && Node->Right == Nil ){
        for ( i=0; i<Depth+1; i++){
            cnt += " ";
            cnt += "\n";
        }
        cnt += "LEAF (BLACK)";
        cnt += "\n";
        for ( i=0; i<Depth+1; i++){
            cnt += " ";
            cnt += "\n";
        }
        cnt += "LEAF (BLACK)";
        cnt += "\n";
    }

    if ((Node->Left == Nil || Node->Right == Nil) && !(Node->Left == Nil && Node->Right == Nil)){
        for ( i=0; i<Depth+1; i++){
            cnt += " ";
            cnt += "\n";
        }
        cnt += "LEAF (BLACK)";
        cnt += "\n";
    }
    else
        cnt += "\n";

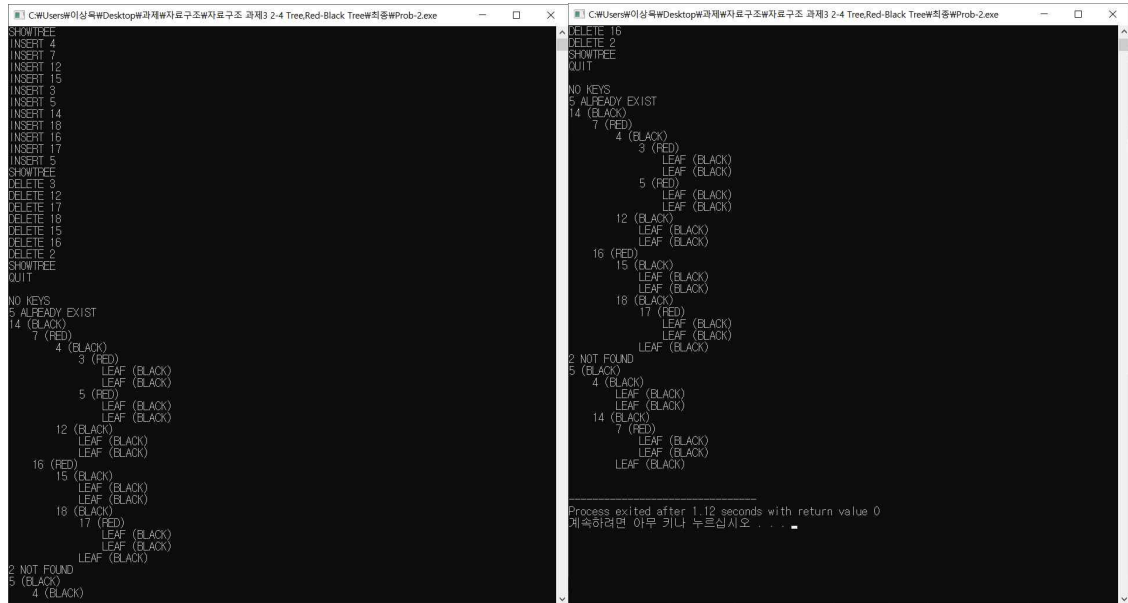
    for ( i=0; i<Depth; i++){
        output += " ";
        output += "\n";
    }

    if (Node->Left == Nil && Node->Right == Nil){
        output += intToString(Node->Data);
        output += "\n";
        output += "(";
        output += (Node->Color == 1)?"RED":"BLACK";
        output += ") ";
        output += "\n";
        output += cnt;
    }
    else if ((Node->Left == Nil) && !(Node->Left==Nil && Node->Right== Nil)){
        output += intToString(Node->Data);
        output += "\n";
        output += "(";
        output += (Node->Color == 1)?"RED":"BLACK";
        output += ") ";
        output += "\n";
        output += cnt;
        output += RBT_PrintTree(Node->Right, Depth+1, BlackCount);
    }
    else if ((Node->Right == Nil) && !(Node->Left==Nil && Node->Right==Nil)){
        output += intToString(Node->Data);
        output += "\n";
        output += "(";
        output += (Node->Color == 1)?"RED":"BLACK";
        output += ") ";
        output += "\n";
        output += cnt;
        output += RBT_PrintTree(Node->Left, Depth+1, BlackCount);
    }
    else{
        output += intToString(Node->Data);
        output += "\n";
        output += "(";
        output += (Node->Color == 1)?"RED":"BLACK";
        output += ") ";
        output += "\n";
        output += cnt;
        output += RBT_PrintTree( Node->Left, Depth+1, BlackCount);
        output += RBT_PrintTree( Node->Right, Depth+1, BlackCount );
    }
    return output;
}

```

2. 실행 결과 캡처

과제 수행자의 컴퓨터에서는 오류 없이 정상 작동 하였다. 왼쪽은 입력 부문, 오른쪽은 입력에 따른 출력 결과이다.



```

C:\Users\이상욱\Desktop\과제\자료구조\자료구조 과제3 2-4 Tree\Red-Black Tree\최종wProb-2.exe
SHOWTREE
INSERT 4
INSERT 7
INSERT 12
INSERT 15
INSERT 9
INSERT 5
INSERT 14
INSERT 18
INSERT 16
INSERT 17
INSERT 5
SHOWTREE
DELETE 9
DELETE 12
DELETE 17
DELETE 18
DELETE 15
DELETE 16
DELETE 2
SHOWTREE
QUIT
NO KEYS
5 ALREADY EXIST
14 (BLACK)
7 (RED)
4 (BLACK)
3 (RED)
LEAF (BLACK)
LEAF (BLACK)
5 (RED)
LEAF (BLACK)
LEAF (BLACK)
12 (BLACK)
LEAF (BLACK)
LEAF (BLACK)
16 (RED)
15 (BLACK)
LEAF (BLACK)
LEAF (BLACK)
18 (BLACK)
17 (RED)
LEAF (BLACK)
LEAF (BLACK)
2 NOT FOUND
5 (BLACK)
4 (BLACK)

C:\Users\이상욱\Desktop\과제\자료구조\자료구조 과제3 2-4 Tree\Red-Black Tree\최종wProb-2.exe
DELETE 16
DELETE 2
SHOWTREE
QUIT
NO KEYS
5 ALREADY EXIST
14 (BLACK)
7 (RED)
4 (BLACK)
3 (RED)
LEAF (BLACK)
LEAF (BLACK)
5 (RED)
LEAF (BLACK)
LEAF (BLACK)
12 (BLACK)
LEAF (BLACK)
LEAF (BLACK)
16 (RED)
15 (BLACK)
LEAF (BLACK)
LEAF (BLACK)
18 (BLACK)
17 (RED)
LEAF (BLACK)
LEAF (BLACK)
2 NOT FOUND
5 (BLACK)
4 (BLACK)
LEAF (BLACK)
LEAF (BLACK)
14 (BLACK)
7 (RED)
LEAF (BLACK)
LEAF (BLACK)
LEAF (BLACK)

Process exited after 1.12 seconds with return value 0
계속하려면 아무 키나 누르십시오 . . .
```