알고리즘

- 05. RBT (Red Black Tree) -

제 출 일	2018.11.05
학 과	컴퓨터공학과
학 번	201402391
이 름	이 병 만

※ 과제 목표 및 해결 방법

- 배열로 구현.
- Red Black Tree를 구현하여 BST와 Red Black Tree의 Search 시간을 비교하여라.

※ 주요 부분 코드 설명 (알고리즘 부분 코드 캡쳐)

1. RedBlackNode

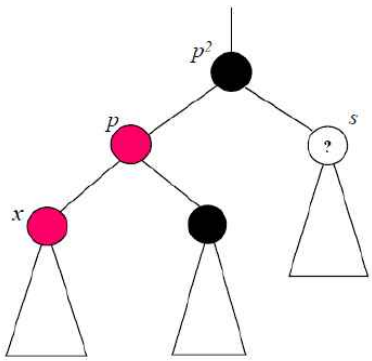
```
public class RedBlackNode {
    RedBlackNode left, right;
    int element;
    boolean color;

    public RedBlackNode(int theElement) {
        this(theElement, null, null);
    }

    public RedBlackNode(int theElement, RedBlackNode lt, RedBlackNode rt) {
        left = lt;
        right = rt;
        element = theElement;
        color = true;
    }
}
```

[그림1. RedBlackNode Class]

➔ RedBlackNode 클래스로 입력받은 키의 값을 저장하는 element와 자식노드인 left, right를 변수로 가진다. 생성자를 통해서 RedBlackNode에 필요한 값을 지정한다. RedBlackTree는 색깔 구분도 해주기 위해서 생성하였고, 처음에는 블랙인 true로 선언하였다.

2. RedBlackTree

```
public class RedBlackTree {
    public RedBlackNode current;
    public RedBlackNode parent;
    public RedBlackNode grand_parent;
    public RedBlackNode great_grand_parent;
    public RedBlackNode root;
    public static RedBlackNode NIL;

    // Red - false  Black - true
    static final boolean Red = false;
    static final boolean Black = true;

    static {
        NIL = new RedBlackNode(0);
        NIL.left = NIL;
        NIL.right = NIL;
    }

    public RedBlackTree(int key) {
        root = new RedBlackNode(key);
        root.left = NIL;
        root.right = NIL;
    }
```



[그림2. RedBlackTree Class]

➔ RedBlackTree에서 필요한 변수들(x = current, p = parent, p^2 = grand_parent, p^3 = great_grand_parent)로 선언하였다.
색 구분을 위한 Red, Black은 각각 false, true로 선언하였다.
static을 통해서 초기값을 세팅한다. RedBlackTree 생성자는 입력받은 key의 값을 루트로 하고 자식은 NIL로 한다.

```
public void insert(int key) {
    current = parent = grand_parent = root;
    NIL.element = key;

    while (current.element != key) {     // 현재노드의 값이 key와 같지 않으면
        great_grand_parent = grand_parent;  // 할아버지를 증조로
        grand_parent = parent;  // 부모를 할아버지로
        parent = current;       // 현재를 부모로
        if (key < current.element)  // 현재노드의 값이 key보다 작으면
            current = current.left; // 현재노드의 왼쪽으로
        else
            current = current.right;     // 현재노드의 오른쪽으로
        if (current.left.color == Red && current.right.color == Red) // 현재노드의 왼/오른쪽의 색이 Red이면
            rebalance(key); // 재배치
    }
    if (current != NIL) // 현재노드가 NIL이 아니라면
        return; // 함수 종료
    current = new RedBlackNode(key, NIL, NIL); // 삽입할 노드 생성
    if (key < parent.element) // 부모의 값이 key보다 크면
        parent.left = current; // 현재노드를 부모의 왼쪽에
    else     // 작으면
        parent.right = current; // 현재노드를 부모의 오른쪽에
    rebalance(key); // 재배치
}
```

[그림3. insert]

➔ 다음 함수는 인자로 받은 키의 값을 삽입하는 함수이다. 모든 노드를 같은 값으로 할당해준다. 키의 값을 NIL의 값으로 할당한다. 반복문을 현재 노드의 값이 key와 같을 때까지 반복한다. 다르면 grand_parent를 uncle, parent를 grand_parent, current를 parent로 할당해준다. 만약 키의 값이 현재 노드의 값보다 작으면 현재 노드의 왼쪽으로 이동하고 크면 오른쪽으로 이동한다.
현재 노드의 왼쪽/오른쪽 자식의 색이 레드이면 재배치 함수를 실행한다.
만약 현재 노드가 NIL이 아니라면 함수를 종료하고 그렇지 않으면 노드를 생성해서 현재 노드에 반환해준다.
생성한 노드를 부모 노드에게 붙여주는 과정을 진행한다. 키의 값이 부모 노드의 값보다 작으면 왼쪽에다 붙여주고 크면 오른쪽에 붙여준다. 노드를 붙여준 뒤 다시 재배치 함수를 실행한다.

```
private void rebalance(int key) {
    current.color = Red;
    current.left.color = Black; // 리프는 블랙
    current.right.color = Black; // 리프는 블랙

    if (parent.color == Red) {
        grand_parent.color = Red;
        if (key < grand_parent.element != key < parent.element)
            parent = rotate(key, grand_parent);
        current = rotate(key, great_grand_parent);
        current.color = Black;
    }
    root.right.color = Black;
}
```

[그림4. rebalance]

➔ 다음 함수는 트리를 재배치해주는 함수이다. 현재 노드의 색을 레드, 자식을 블랙으로 설정한다.
만약 부모의 색이 레드이면 사촌의 색을 레드로 변경한다. 키의 값이 사촌의 값과 비교하고 부모의 값과 비교해서 두 결과가 다르면 사촌의 노드를 회전시켜 부모로 반환해준다. 그렇지 않으면 현재의 노드로 반환하고 현재 노드의 색을 블랙으로 변경해준다. 만약 부모의 색이 블랙이면 루트의 오른쪽 자식의 색을 블랙으로 변경한다.

```
private RedBlackNode rotate(int key, RedBlackNode parent) {
    if (key < parent.element)
        if (key < parent.left.element)
            return parent.left = rotateLeftChild(parent.left);
        else
            return parent.left = rotateRightChild(parent.left);
    else
        if (key < parent.right.element)
            return parent.right = rotateLeftChild(parent.right);
        else
            return parent.right = rotateRightChild(parent.right);
}
```

[그림5. rotate]

➔ 다음 함수는 트리의 균형이 깨지면 트리를 회전시키기 위한 함수이다. 인자로 받은 키와 부모의 값을 비교한다. 부모의 값이 더 크다면 부모의 왼쪽 자식의 값과 키와 비교해서 키가 작으면 왼쪽으로 회전시켜서 부모의 왼쪽 자식으로 반환해준다. 크다면 오른쪽으로 회전시켜 부모의 왼쪽 자식으로 반환해준다.
부모의 값이 작다면 부보의 오른쪽 자식의 값과 키와 비교해서 키가 작으면 왼쪽으로 회전시켜서 부모의 오른쪽 자식으로 반환해준다. 크다면 오른쪽으로 회전시켜 부모의 오른쪽 자식으로 반환해준다.

```
/* Rotate tree left child */
private RedBlackNode rotateLeftChild(RedBlackNode key_1) {
    RedBlackNode key_2 = key_1.left;
    key_1.left = key_2.right;
    key_2.right = key_1;
    return key_2;
}

/* Rotate tree right child */
private RedBlackNode rotateRightChild(RedBlackNode key_2) {
    RedBlackNode key_1 = key_2.right;
    key_2.right = key_1.left;
    key_1.left = key_2;
    return key_1;
}
```

[그림6. rotateLeft/RightChild]

➔ 다음 함수는 왼쪽/오른쪽 자식을 회전시키는 함수이다. 입력받은 노드의 왼쪽/오른쪽을 key_2/key_1로 저장한다. 저장한 노드의 오른쪽/왼쪽을 입력받은 노드의 왼쪽/오른쪽으로 변경한다. 입력받은 노드를 저장한 노드의 오른쪽/왼쪽으로 변경하고 저장한 노드를 반환한다.

```
public boolean search(int key) {
    return search(root.right, key);
}
private boolean search(RedBlackNode root, int key) {
    boolean find = false;
    while ((root != NIL) && !find) {
        int rval = root.element;
        if (key < rval)
            root = root.left;
        else if (key > rval)
            root = root.right;
        else {
            find = true;
            break;
        }
        find = search(root, key);
    }
    return find;
}
```

[그림7. search]

➔ 다음 함수는 인자로 받은 키의 값을 트리에서 찾는 함수이다. 인자로 검색하고 싶은 키를 입력받는다.
처음에 find 변수를 false로 한 다음 반복문을 수행하면서 key의 값과 root의 값을 비교하면서 둘의 값이 같을 때까지 재귀를 수행한다. 찾으면 find = true로 변경해주고 함수를 빠져나온다.

```java
public void inorder() {
    inorder(root.right);
}
private void inorder(RedBlackNode root) {
    if (root != NIL) {
        inorder(root.left);
        char color = 'B';
        if (root.color == false)
            color = 'R';
        System.out.print(root.element +""+color+" ");
        inorder(root.right);
    }
}
```

[그림8. inorder]

➔ 다음 함수는 함수는 트리를 오름차순으로 출력해주는 함수이다. 루트가 NIL이 아닌 경우 왼쪽부터 오른쪽까지 수행하면서 각 노드의 값에 해당하는 색깔을 붙여서 출력해주었다.

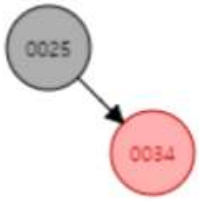
※ 실행 결과(시간 복잡도 포함)

```
Hey!! File read finish...

------------ Red_Black_Tree TEST ------------
TIME(RBT_Insert) : 0.062009(ms)
TIME(RBT_Search) : 0.007396(ms)

1R 2B 3B 6B 7B 8R 9B 10R 11B 12R 13B 17R 25B 26B 27R 32B 33R 34B 36B 38R 41R 43B 45R 47B

------------ Binary_Search_Tree TEST ------------
TIME(BST_Insert) : 1.334613(ms)
TIME(BST_Search) : 0.008533(ms)

1 2 3 6 7 8 9 10 11 12 13 17 25 26 27 32 33 34 36 38 41 43 45 47
```

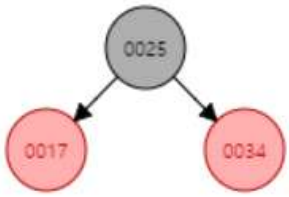
실행한 결과 삽입 시간은 RBT가 BST보다 압도적으로 빠르다는 것을 알 수 있었다. 그 이유는 이미 균형이 이루어진 트리에서 값을 삽입하는 것이기 때문에 삽입하려는 위치를 찾기에도 BST보다 빠르기 때문이라고 생각한다.
이번 과제에서는 두 트리의 search시간을 비교하는 것이기 때문에 그 결과 RBT가 BST보다 빨랐다. 하지만 확연하게 차이가 날 정도는 아니였다. 그 이유를 생각해보면 이 트리는 BST조차 한쪽으로 치우쳐진 트리가 아닌거 같기 때문에 이러한 결과가 나온거 같다. 만약 루트를 기준으로 한쪽으로 무거운 트리가 만들어졌다면 search의 시간의 차이가 컸을 것이다. RBT는 B-트리와 AVL트리와 비슷한 균형을 맞추는 트리이기 때문에 BST와 같이 들어오는 값에 대해 비교 후 바로 삽입하는 것과 다르게 균형이 깨지면 트리를 회전함으로써 트리의 전체적인 균형을 유지한다. 이러한 과정을 거친 트리는 search를 하는 과정에서도 빛을 볼 수 있다. 균형을 이룬 트리이기 때문에 한쪽이 무거운 트리보다는 찾는 시간이 절약된다. 이 과제의 목적도 이와 같다고 볼 수 있다.

※ 그래프 생성 과정

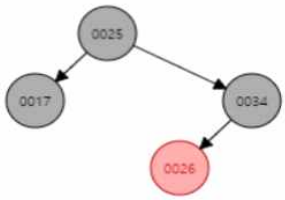
[25삽입]



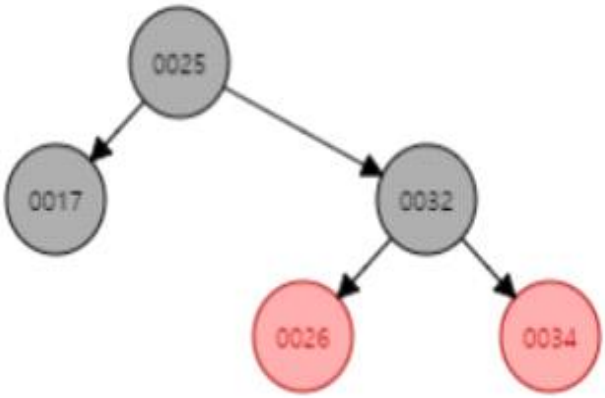
[34삽입]



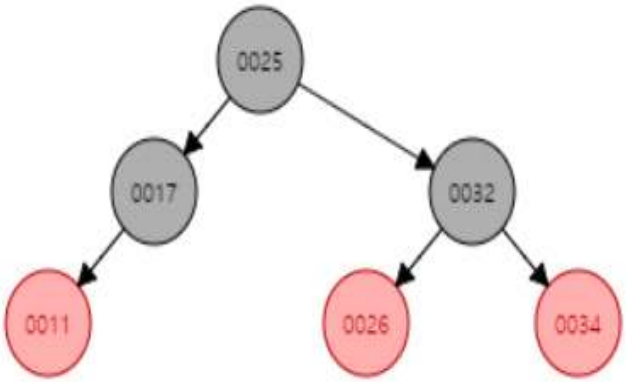
[17삽입]



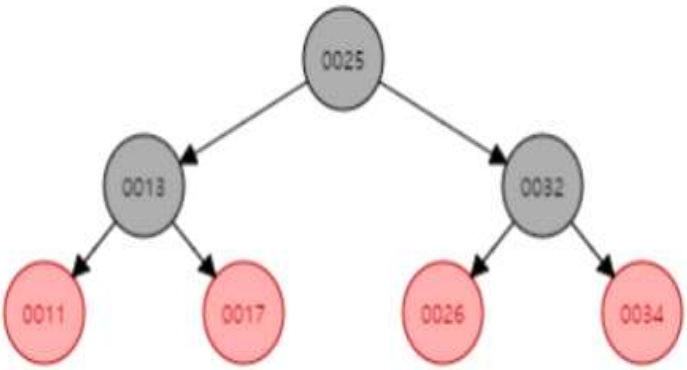
[26삽입]



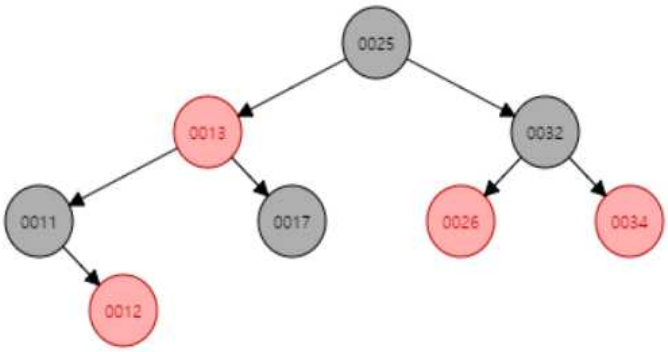
[32삽입]



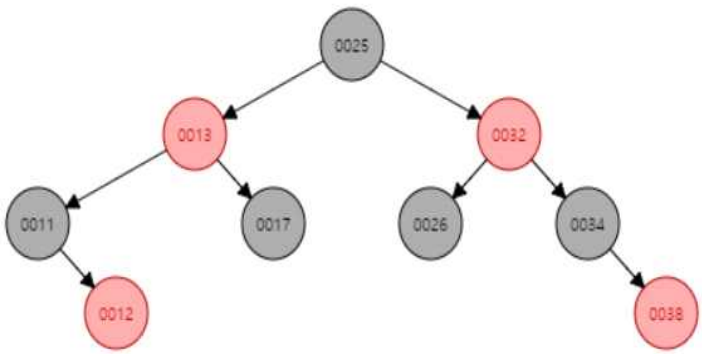
[11삽입]



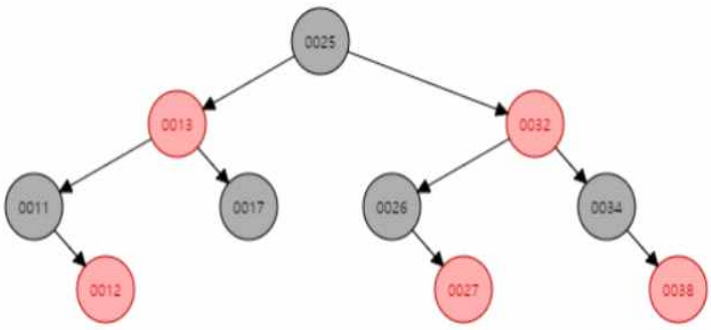
[13삽입]



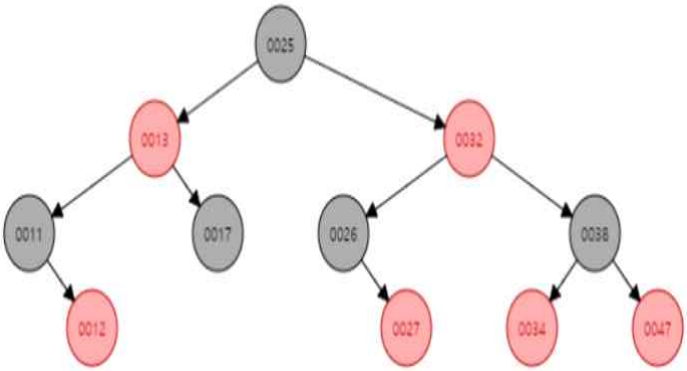
[12삽입]



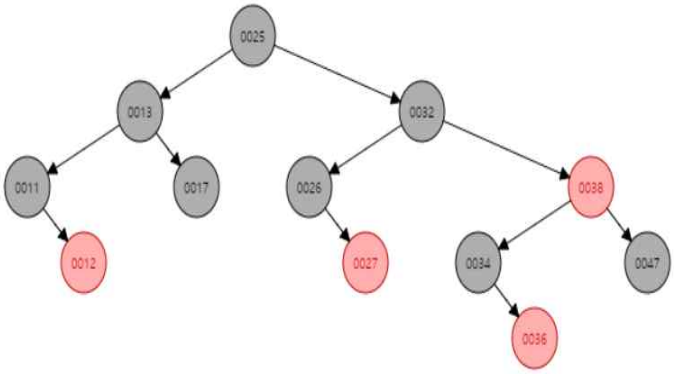
[38삽입]



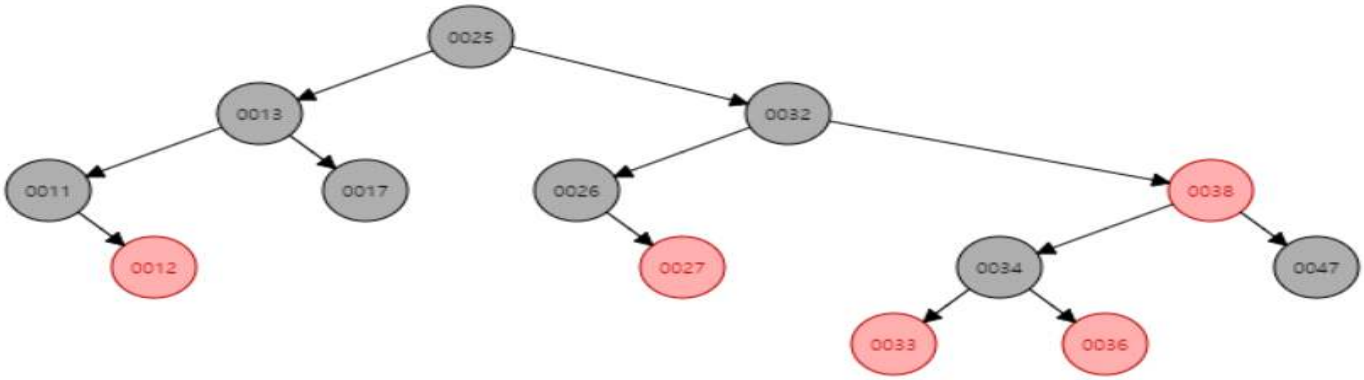
[27삽입]



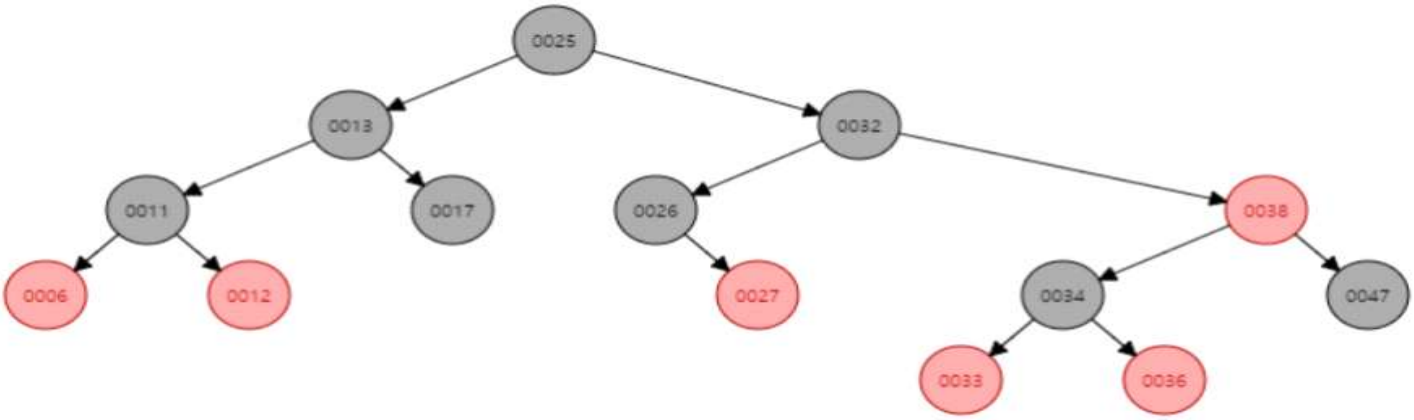
[47삽입]



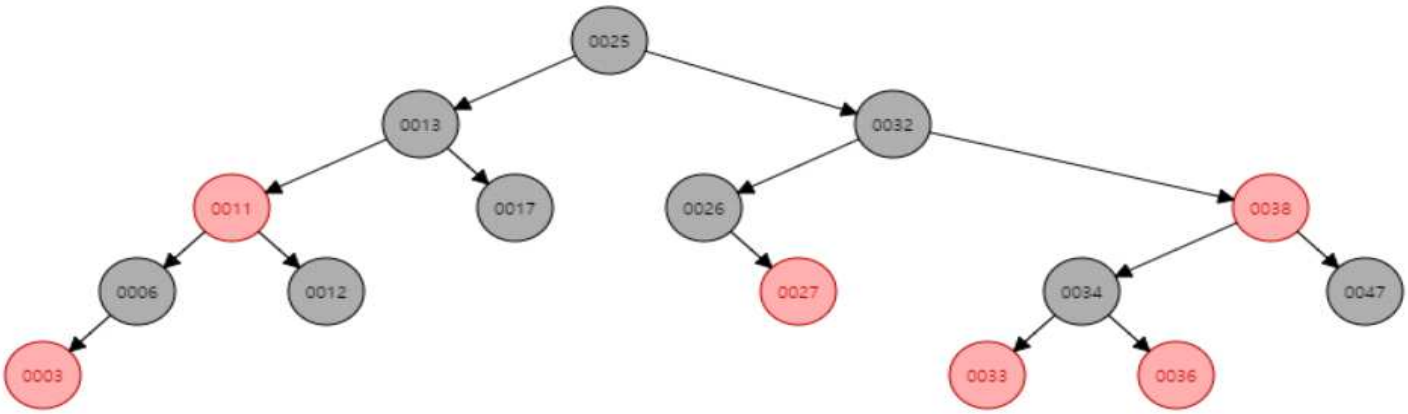
[36삽입]



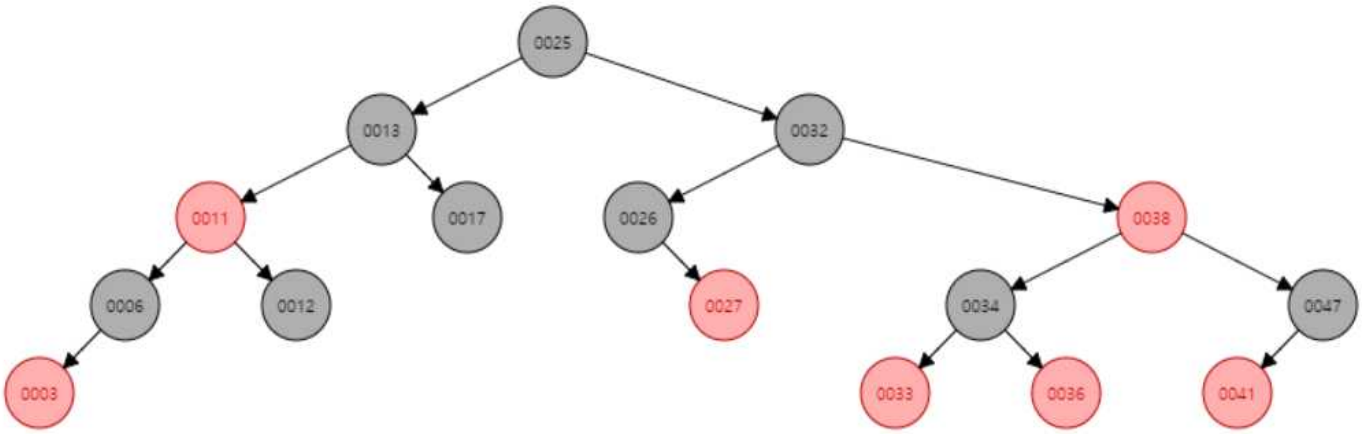
[33삽입]



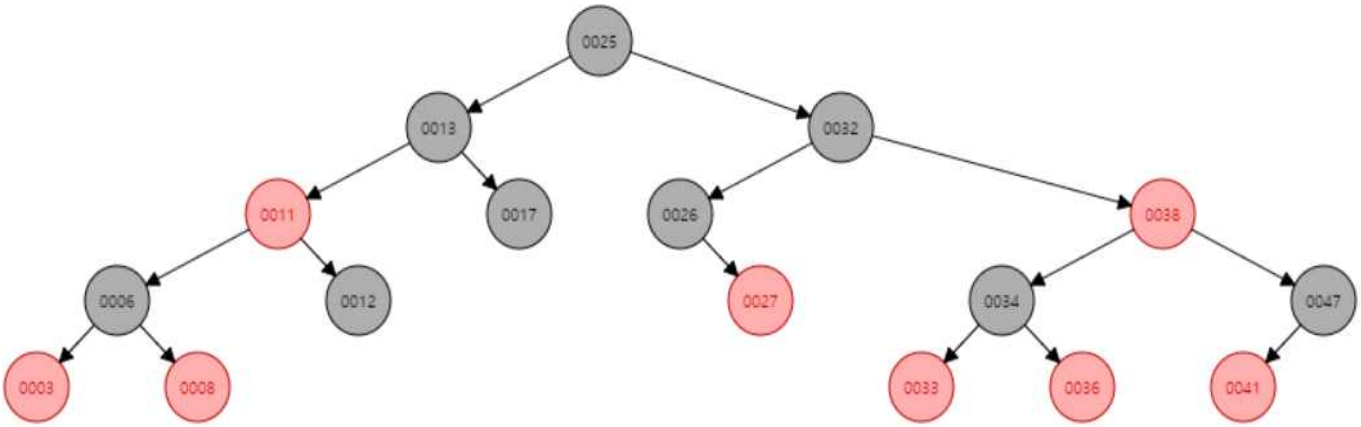
[6삽입]



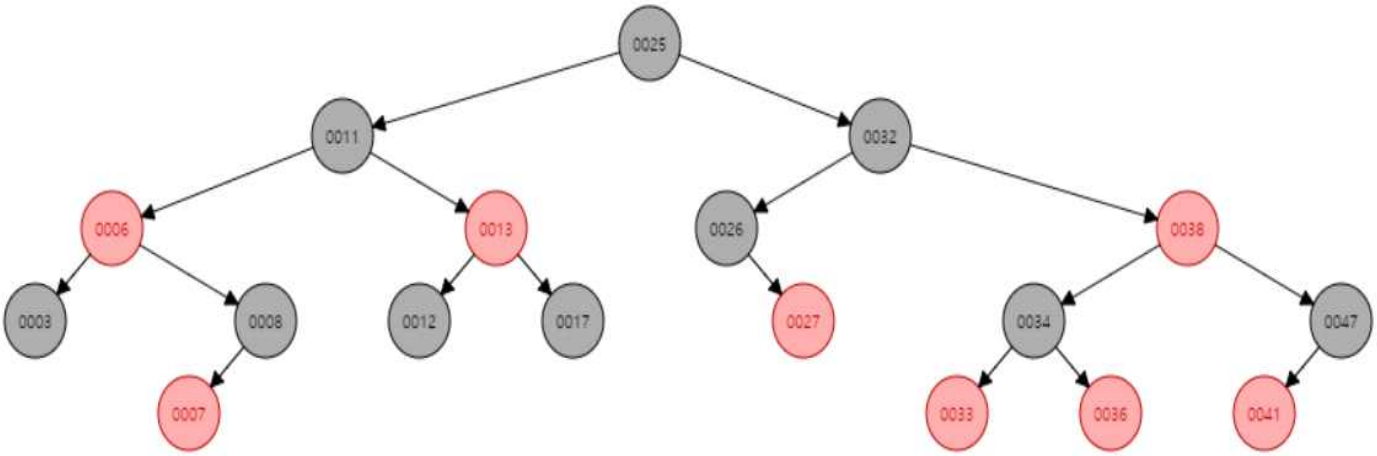
[3삽입]


0025
0013
0032
0011
0017
0026
0038
0006
0012
0027
0034
0047
0003
0033
0036
0041

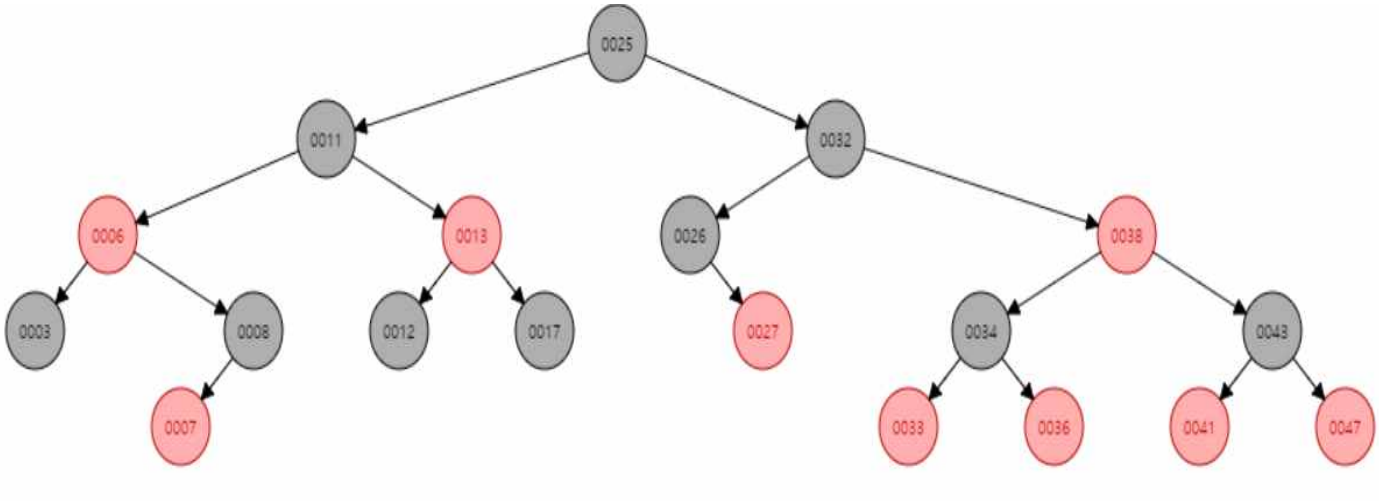
[41삽입]


0025
0013
0032
0011
0017
0026
0038
0006
0012
0027
0034
0047
0003
0008
0033
0036
0041

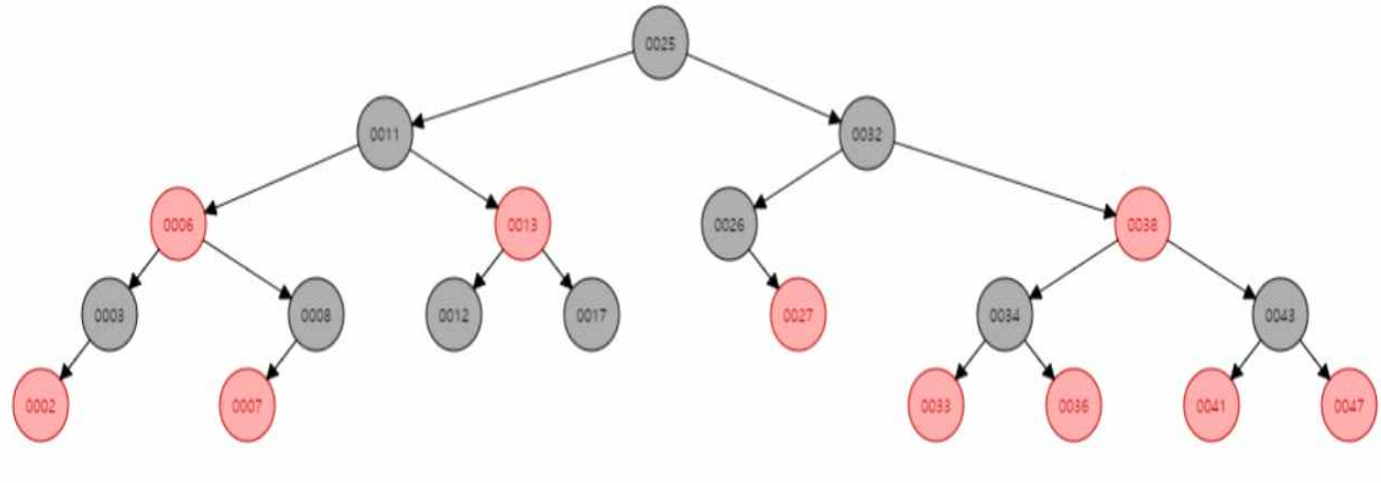
[8삽입]


0025
0011
0032
0006
0013
0026
0038
0003
0008
0012
0017
0027
0034
0047
0007
0033
0036
0041

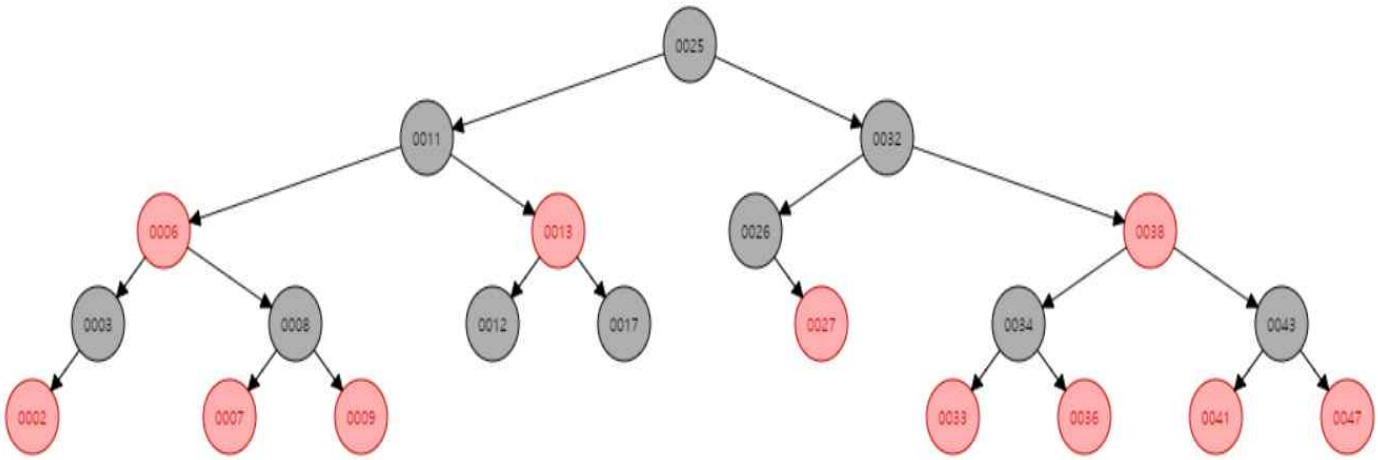
[7삽입]


0025
0011
0032
0006
0013
0026
0038
0003
0008
0012
0017
0027
0034
0043
0007
0033
0036
0041
0047

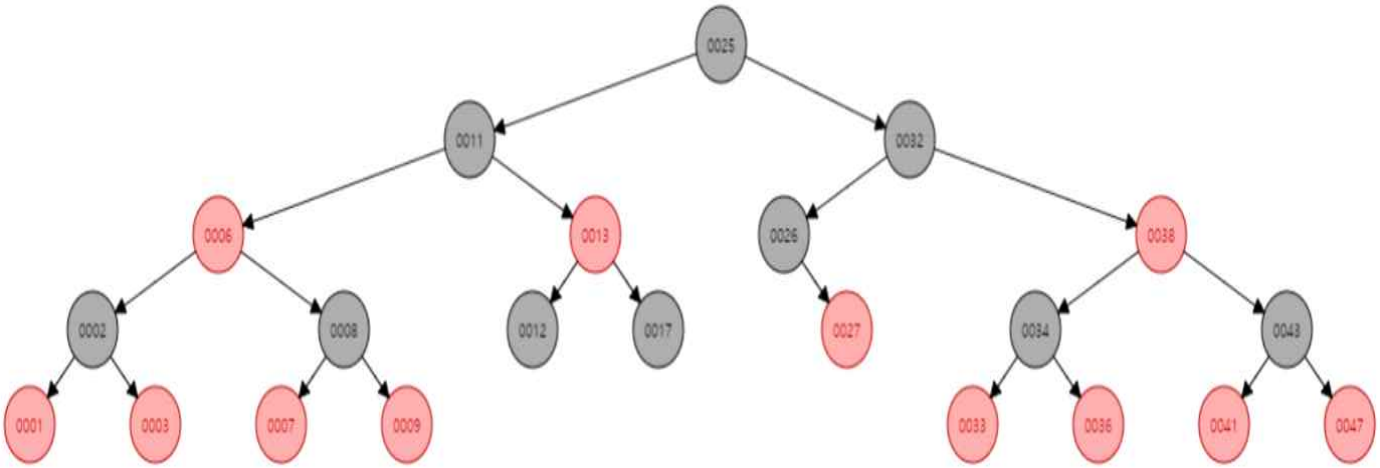
[43삽입]


0025
0011
0032
0006
0013
0026
0038
0003
0008
0012
0017
0027
0034
0043
0002
0007
0033
0036
0041
0047

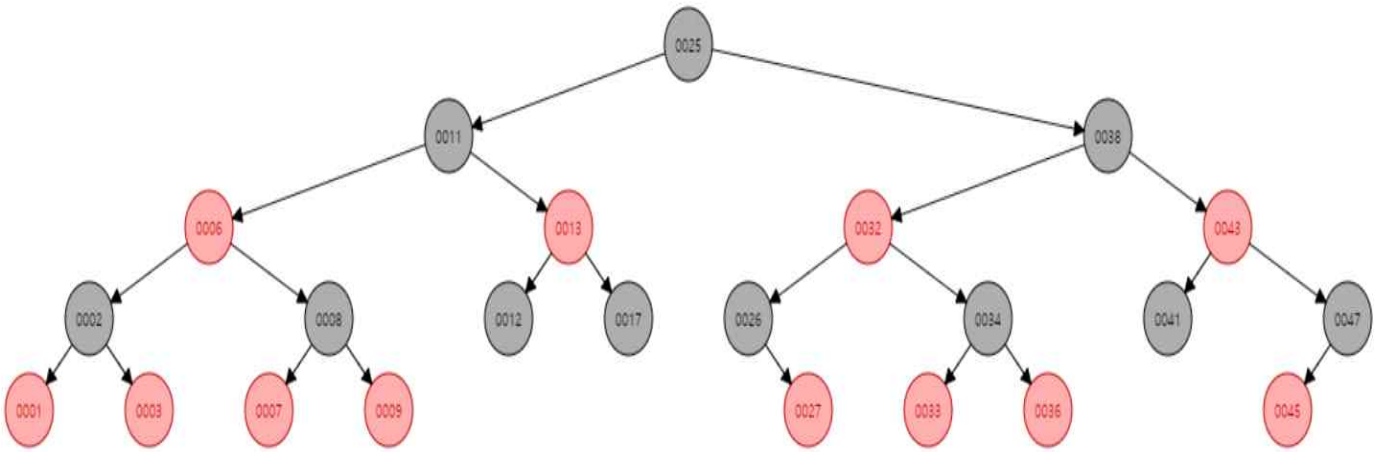
[2삽입]


0025
0011
0032
0006
0013
0026
0038
0003
0008
0012
0017
0027
0034
0043
0002
0007
0009
0033
0036
0041
0047

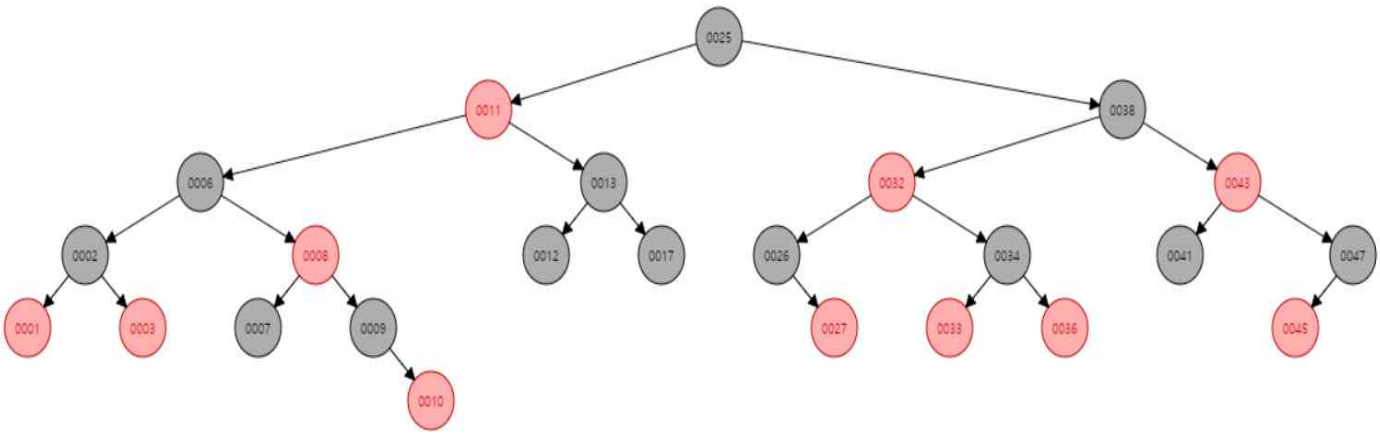
[9삽입]



[1삽입]



[45삽입]



[10삽입]