알고리즘 과제(05)

201204025 김대래

과목 명 : 알고리즘
담당 교수 : 김동일
분반 : 02 분반

Index

1. 실습 환경

OS : MacOs Mojave 10.14
Language : C
Tool : Visual Studio Code, Neovim, gcc

2. 과제 개요

임의의 나열된 수가 저장된 파일(Data1.txt)을 입력 받아 레드-블랙 트리를 구성하고 결과를 파일로 출력한다.

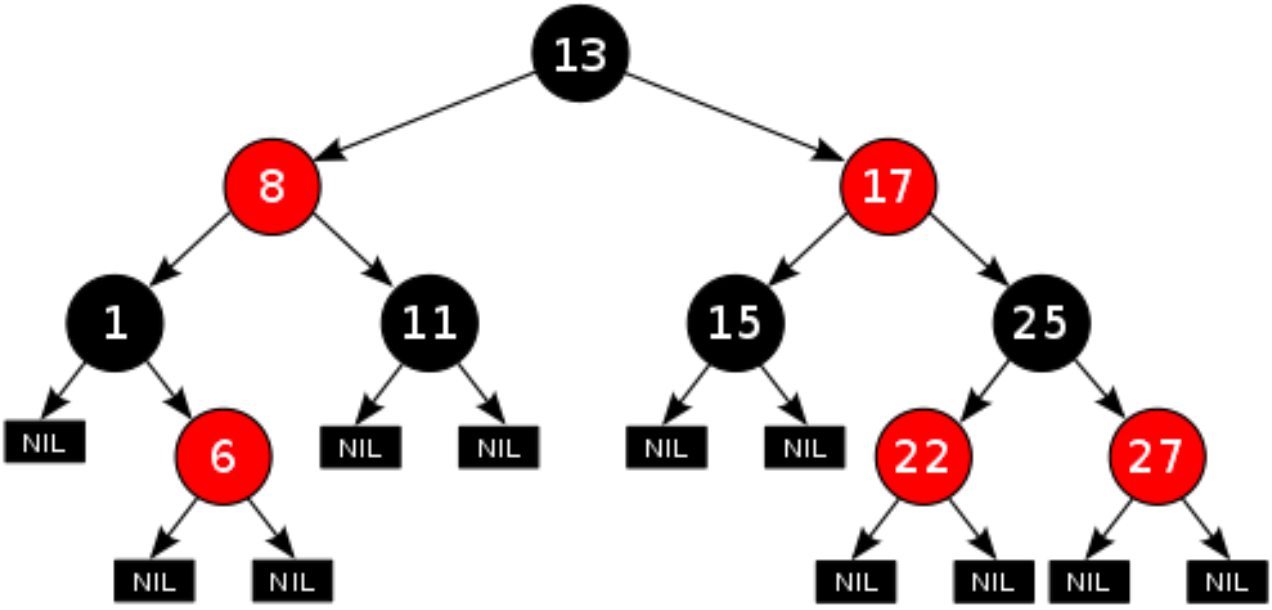
입력 예시 : Data1.txt
>> Input Test File Name : Data1.txt

출력 예시 : inorderTree.txt, levelOrderTree.txt

3. 주요 개념

3.1. 레드-블랙 트리(Red-Black Tree)

3.1.1. 개념



레드-블랙 트리는 자가 균형 이진 탐색 트리(self-balancing binary search tree)로써, 최악의 경우에도 상당히 우수한 실행 시간을 보인다.

트리에 n개의 원소가 있을 때 $O(logn)$ 의 시간 복잡도로 삽입, 삭제, 검색을 할 수 있다.

3.1.2. 조건

I. 루트는 블랙이다.
II. 모든 리프(NIL)는 블랙이다.
III. 노드가 레드이면 그 노드의 자식은 반드시 블랙이다.
IV. 루트 노드에서 임의의 리프 노드에 이르는 경로에서 만나는 블랙 노드의 수는 모두 같다.

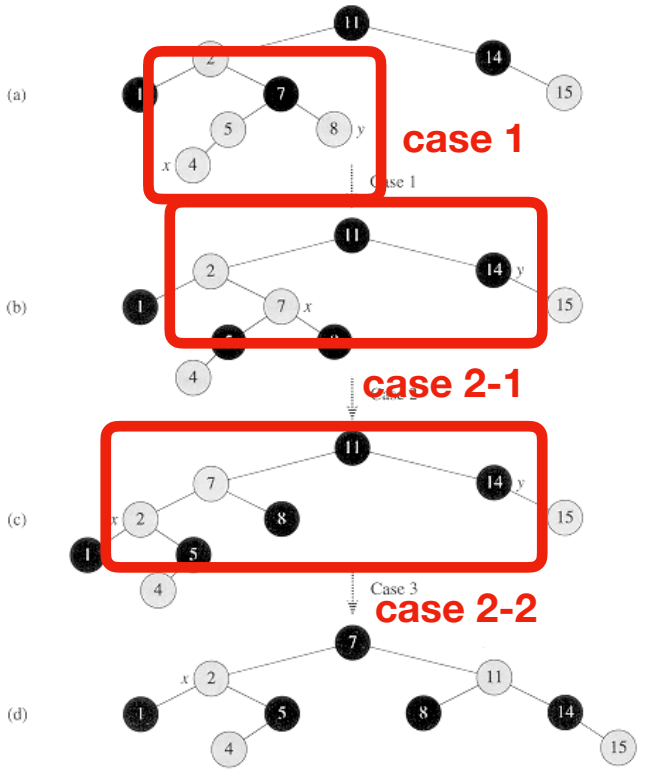
3.1.3. 삽입

레드-블랙 트리의 삽입은 이진탐색 트리의 삽입과정을 따르며 삽입 노드의 색상을 레드로 하여 조건을 만족 시키도록 색 변환(case 1) 및 트리 회전(case 2)을 거친다.

삽입은 아래 3가지의 경우를 고려하여 작성한다.

삽입하려는 노드의 부모 색상이 블랙이면 그냥 삽입하면 되며 부모 노드의 색상이 레드일 때 3가지 경우를 고려할 수 있다.

case 1 : 삼촌 노드의 색상이 빨강
case 2-1 : 삼촌 노드의 색상이 블랙이며, 삽입 노드가 부모의 오른 자식
case 2-2 : 삼촌 노드의 색상이 블랙이며, 삽입 노드가 부모의 왼 자식



3.1.4. 탐색

레드-블랙 트리의 경우 root노드에서부터 해당 값을 찾기 위해 자식을 탐색해 나가는 방식으로 최악의 경우 높이 h만큼 탐색해야하기 때문에 복잡도가 $O(h)$이며 이진 트리(Binary Tree)의 특성에 따라 n개의 원소의 최대 높이인 $O(logn)$의 복잡도를 가진다.

하지만 레드-블랙 트리는 자가 균형 이진 탐색 트리로써 검색시 최악의 경우(worst-case)에서의 시간복잡도가 트리의 높이(또는 깊이)에 따라 결정되기 때문에 보통의 이진 탐색 트리에 비해 효율적이라고 할 수 있다.

4. 과제 5 Red-Black Tree

4.1. main() - input data

```
int main(){
    int i = 0;
    char input[256];
    char *input_data;
    char inputFileName[256];
    printf(">> Input Test File Name : ");
    scanf("%s", inputFileName);

    FILE *input_FD;
    FILE *output_FD;
    clock_t start, end;
    float time;

    input_FD = fopen(inputFileName, "r");
    if (input_FD == NULL){
        printf("Could not open file %s",inputFileName);
        return 1;
    }
    NIL = NILNode();
    root = NIL;
    start = clock();
    while( !feof( input_FD ) ){
        input_data = fgets(input, sizeof(input), input_FD);
        if(input_data == NULL) break;
        int data = atoi(input_data);
        RBNode *z = newNode(data);
        rb_tree_insert(root, z);
    }
    end = clock();
    time = (float)(end - start) / CLOCKS_PER_SEC;
    printf("Time : %f\n", time);
```

- 입력 받을 파일을 받아 해당 크기 만큼 반복하며 데이터를 저장
- 해당 데이터를 노드로 만들어 트리에 삽입
- 삽입하는 과정의 시간을 측정

4.2. rb_tree_insert(T, z)

4.2.1. 의사 코드

```
RB-INSERT(T, z)
 1  y = T.nil
 2  x = T.root
 3  while x ≠ T.nil
 4      y = x
 5      if z.key < x.key
 6          x = x.left
 7      else x = x.right
 8  z.p = y
 9  if y == T.nil
10      T.root = z
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
14  z.left = T.nil
15  z.right = T.nil
16  z.color = RED
17  RB-INSERT-FIXUP(T, z)
```

- 루트 노드를 기준으로 리프 노드가 될 때까지 반복하며 입력할 z 노드의 값에 맞는 위치를 탐색

- 삽입 시 자식으로 NIL노드를 넣고 색상은 RED로

- RB-INSERT-FIXUP함수를 호출하여 레드블랙트리 조건에 맞춤

4.2.2. 소스 코드

```
RBNode* newNode(int data){
    RBNode *new_node = (RBNode *)malloc(sizeof(RBNode));
    if(new_node == NULL){
        fprintf(stderr, "error memory! (create node)\n");
        exit(1);
    }
    new_node->data = data;
    new_node->left = NIL;
    new_node->right = NIL;
    new_node->parent = NIL;

    return new_node;
}

void rb_tree_insert(RBNode *node, RBNode *z){
    RBNode *y = NIL;
    RBNode *x = node;
    while(x != NIL){
        y = x;
        if(z->data < x->data)
            x = x->left;
        else
            x = x->right;
    }
    z->parent = y;
    if(y == NIL)
        root = z;
    else if( z->data < y->data)
        y->left = z;
    else
        y->right = z;
    z->left = NIL;
    z->right = NIL;
    z->color = RED;
    rb_tree_fixup(node, z);
}
```

- 입력 받은 데이터 노드 생성

- 의사코드를 바탕으로 작성

- 삽입할 노드 z 노드를 root 노드 x 부터 차례대로 탐색하며 삽입 (이진 탐색 트리와 동일)
- 리프 노드 NIL을 삽입 노드 z의 자식으로
- 삽입 노드는 RED로 삽입 후
- 레드-블랙 트리의 조건에 만족하도록 해당 노드를 기준으로 fixup 함수를 호출

4.3. rb_tree_fixup(T,z)

4.3.1. 의사 코드

```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left
 3          y = z.p.p.right
 4          if y.color == RED
 5              z.p.color = BLACK                    // case 1
 6              y.color = BLACK                      // case 1
 7              z.p.p.color = RED                    // case 1
 8              z = z.p.p                            // case 1
 9          else if z == z.p.right
10                  z = z.p                          // case 2
11                  LEFT-ROTATE(T, z)                // case 2
12              z.p.color = BLACK                    // case 3
13              z.p.p.color = RED                    // case 3
14              RIGHT-ROTATE(T, z.p.p)               // case 3
15      else (same as then clause
                with "right" and "left" exchanged)
16  T.root.color = BLACK
```

- 삽입 시 3가지 경우에 따라 레드-블랙 트리 조건에 맞도록 색 변환 또는 트리 회전을 실시(3.1.3 삽입 참고)
- case 1의 경우 삼촌(y)의 색상이 레드일 때 색 변환
  (할아버지를 다시 검사하기 위하여 z = z.p.p)
- case 2의 경우 삼촌(y)의 색상이 블랙이며 오른 자식일 때 왼쪽 회전을 통해 case 3의 경우와 동일하도록 변환
- case 3의 경우 삼촌(y)의 색상이 블랙이며 왼 자식일 때 할아버지(부모의 부모)를 기준으로 오른쪽 회전을 통해 레드-블랙 트리 조건을 충족

4.3.2. 소스 코드

```
void rb_tree_fixup(RBNode *node, RBNode *z){
    RBNode *y = (RBNode *)malloc(sizeof(RBNode));

    while(z->parent->color == RED){
        if(z->parent == z->parent->parent->left){
            y = z->parent->parent->right;
            if(y->color == RED){
                z->parent->color = BLACK;
                y->color = BLACK;
                z->parent->parent->color = RED;
                z = z->parent->parent;
            }
            else{
                if(z == z->parent->right){
                    z = z->parent;
                    left_rotate(node, z);
                }
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                right_rotate(node, z->parent->parent);
            }
        }
```

case 1

case 2

case 3

- 의사 코드를 바탕으로 작성
- 해당 부분은 노드가 왼쪽에 있을 때를 기준으로 작성한 부분

- case 1, 2, 3에 따라 색 변환 및 트리 회전 실시

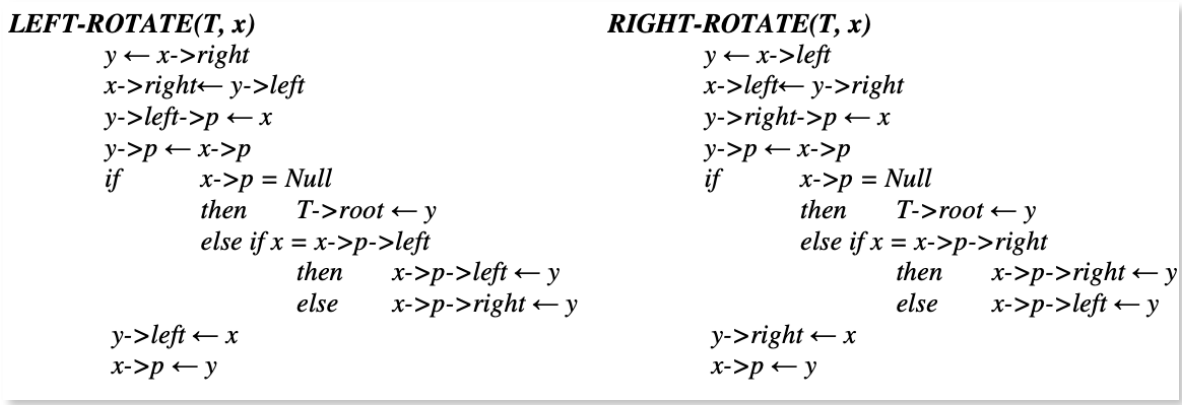
```
        else{
            y = z->parent->parent->left;
            if(y->color == RED){
                z->parent->color = BLACK;
                y->color = BLACK;
                z->parent->parent->color = RED;
                z = z->parent->parent;
            }
            else{
                if(z == z->parent->left){
                    z = z->parent;
                    right_rotate(node, z);
                }
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                left_rotate(node, z->parent->parent);
            }
        }
    }
    root->color = BLACK;
}
```

• 노드가 오른쪽에 있을 경우로 위와 대칭되는 구조

• 마찬가지로 case 1, 2, 3에 따라 색 변환 및 트리 회전 실시

4.4. left_rotate / right_rotate

4.4.1 의사 코드



```
LEFT-ROTATE(T, x)
    y ← x->right
    x->right← y->left
    y->left->p ← x
    y->p ← x->p
    if      x->p = Null
            then    T->root ← y
            else if x = x->p->left
                    then    x->p->left ← y
                    else    x->p->right ← y
    y->left ← x
    x->p ← y
```

```
RIGHT-ROTATE(T, x)
    y ← x->left
    x->left← y->right
    y->right->p ← x
    y->p ← x->p
    if      x->p = Null
            then    T->root ← y
            else if x = x->p->right
                    then    x->p->right ← y
                    else    x->p->left ← y
    y->right ← x
    x->p ← y
```

- 기준이 되는 노드 x의 왼쪽/오른쪽 회전에 따라 자식을 부모로 두고
- 자식의 자식을 x의 자식으로 연결
- 왼쪽 회전의 경우 x의 오른 자식이 x의 부모
- 오른 회전의 경우 x의 왼 자식이 x의 부모

4.4.2. 소스 코드

```c
void left_rotate(RBNode *node, RBNode *z){
    RBNode *p = z->parent;
    RBNode *y = z->right;

    z->right = y->left;
    if( y->left != NIL)
        y->left->parent = z;
    y->parent = z->parent;
    if(z->parent == NIL)
        root = y;
    else if(z == z->parent->left)
        z->parent->left = y;
    else
        z->parent->right = y;

    y->left = z;
    z->parent = y;
}
```

```c
void right_rotate(RBNode *node, RBNode *z){
    RBNode *p = z->parent;
    RBNode *y = z->left;

    z->left = y->right;
    if( y->right != NIL)
        y->right->parent = z;
    y->parent = z->parent;
    if(z->parent == NIL)
        root = y;
    else if(z == z->parent->left)
        z->parent->left = y;
    else
        z->parent->right = y;

    y->right = z;
    z->parent = y;
}
```

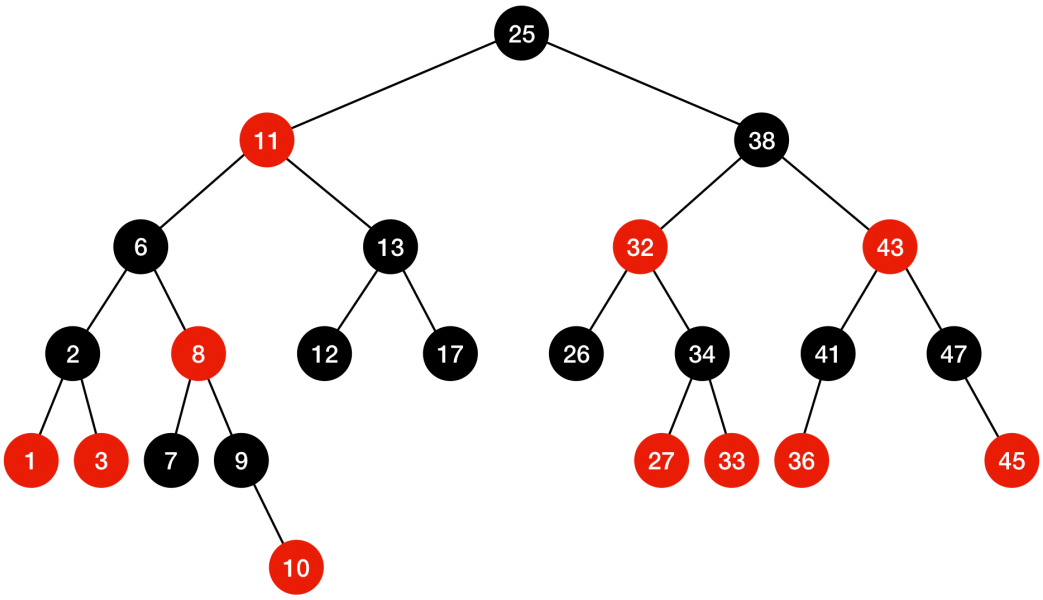
- 의사 코드를 바탕으로 작성
- 왼쪽 회전의 경우 오른 자식의 왼 자식을 자신의 오른 자식으로 두고 오른 자식을 부모로 하여 왼쪽 회전
- 오른 회전의 경우 왼 자식의 오른 자식을 자신의 왼 자식으로 두고 왼 자식을 부모로 하여 오른 회전

5. 실행

5.1. Red-Black Tree insert

```
~/Desktop/Algorithm/실 습 /hw06
➜  hw06 ./rbtree
>> Input Test File Name : Data1.txt
Time : 0.000306
Level Order :
(25, B)
(11, R) (38, B)
(6, B) (13, B) (32, R) (43, R)
(2, B) (8, R) (12, B) (17, B) (26, B) (34, B) (41, B) (47, B)
(1, R) (3, R) (7, B) (9, B) (NIL) (NIL) (NIL) (NIL) (NIL) (27, R) (33, R) (36, R) (NIL) (NIL) (45,
R) (NIL)
(NIL) (NIL) (NIL) (NIL) (NIL) (NIL) (NIL) (10, R) - - - - - - - - - - (NIL) (NIL) (NIL) (NIL) (NIL)
 (NIL) - - - - (NIL) (NIL) - -
- - - - - - - - - - - - - - - (NIL) (NIL) - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 - -

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



- 위 Level Order 결과를 보기 좋게 표현하면 다음과 같은 그림이 나온다.
- In Order(중위 순회) 결과 정렬이 되는 것을 확인
- 각 Leaf 노드 까지 거치는 블랙의 수가 동일하며 레드-블랙 트리의 조건을 모두 충족함을 확인

6. 결과

7.1. 삽입 시 트리 구조 비교

```
~/Desktop/Algorithm/실습/hw06
➜  hw06 ./rbtree
>> Input Test File Name : data1.txt
Time : 0.000239
Level Order :
(25, B)
(11, R) (38, B)
(6, B) (13, B) (32, R) (43, R)
(2, B) (8, R) (12, B) (17, B) (26, B) (34, B) (41, B) (47, B)
(1, R) (3, R) (7, B) (9, B) (NIL) (NIL) (NIL) (NIL) (NIL) (27, R) (33, R) (36, R) (NIL) (NIL) (45, R) (NIL)
(NIL) (NIL) (NIL) (NIL) (NIL) (NIL) (NIL) (10, R) - - - - - - - - - - (NIL) (NIL) (NIL) (NIL) (NIL) (NIL) - - - - (NIL) (NIL) - -
- - - - - - - - - - - - - - - - (NIL) (NIL) - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

~/Desktop/Algorithm/실습/hw06
➜  hw06 ./bst
>> Input Test File Name : data1.txt
Time : 0.000126
Level Order :
25
17 34
11 - 26 38
6 13 - - 32 36 47
3 8 12 - - - 27 33 - - 41 -
2 - 7 9 - - - - - - - - - - - - 43 -
1 - - - - - 10 - - - - - - - - - - - - - - 45 -
```

- 동일한 파일(Data1.txt)로 삽입
- 삽입 시 해당 데이터의 자리를 찾기위한 탐색의 복잡도가 필요
- 탐색은 $O(h)$의 복잡도
- 이진 탐색 트리는 최악의 경우 한 쪽으로 치우치게 되어 $O(n)$의 복잡도
- 이진 탐색 트리는 높이를 기준으로 복잡도의 차이가 커지기 때문에 balance를 맞추는게 중요
- Red-Black 트리는 균형을 맞추어 삽입하여 최악의 경우에도 $O(logn)$을 보장
- Binary Search Tree 삽입의 경우 height가 6
- Red-Black Tree 삽입의 경우 height 가 5
- balance를 보장하는 레드-블랙 트리가 높이가 더 작게 나옴을 확인
- 삽입 시 색 변환 및 회전에 따른 시간 소모로 인하여 red-black트리가 더 오래 걸렸지만
- **탐색 속도는 높이에 따라 좌우 되기 때문에 더 큰 값들을 입력 받았다고 가정한다면 Red-black 트리가 더 효율적이다.**

```
~/Desktop/Algorithm/실습/[Al]실습5주차
➜  [Al]실습5주차  ./test
>> Input test number [1 : Data1.txt, 2: Data2.txt] : 1
>> Input insert mode [1 : insert, 2 : median inser, 3 : balanced median] : 1
Time : 0.000048
25
17 34
11 - 26 38
- 13 - - 32 - -
- 12 - - - 27 - - -
delete data : [exit : -1] -1

~/Desktop/Algorithm/실습/[Al]실습5주차
➜  [Al]실습5주차  cat inorderTree.txt
11
12
13
17
25
26
27
32
34
38
```

```
~/Desktop/Algorithm/실습/[Al]실습5주차
➜  [Al]실습5주차  ./test
>> Input test number [1 : Data1.txt, 2: Data2.txt] : 1
>> Input insert mode [1 : insert, 2 : median inser, 3 : balanced median] : 2
Time : 0.000011
25
17 26
13 - - 27
12 - - - - 32
11 - - - - - - 34
- - - - - - - - - 38
delete data : [exit : -1] -1

~/Desktop/Algorithm/실습/[Al]실습5주차
➜  [Al]실습5주차  cat inorderTree.txt
11
12
13
17
25
26
27
32
34
38
```