알고리즘 과제(05)

201404377 진승언

1. 과제 목표 및 해결 방법

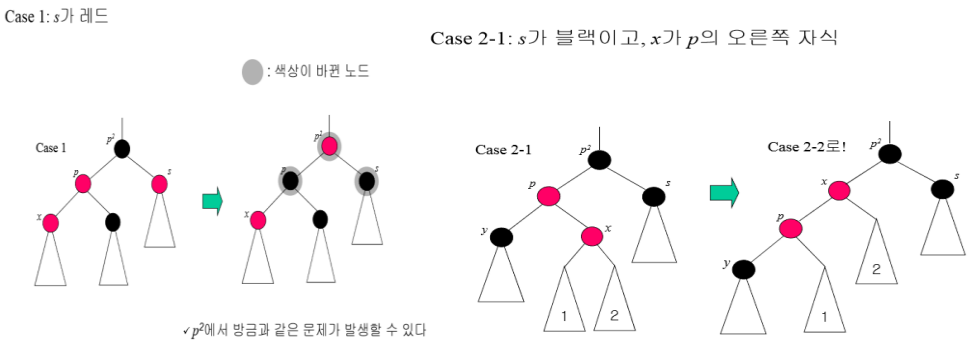
⇨ 이번 과제의 목표는 레드블랙트리를 구현하는 것이 목표였다. 그 중에서도 insert부분을 구현하고 이전에 구현했던 BST(Binary Search Tree)의 search하는데 걸리는 시간을 비교하는 것이 주된 목표였다. 레드블랙트리의 insert를 구현하는것이 가장 먼저 해야 할 일이었는데 이전에 했던 BST에서 nil 노드라는 리프노드를 노드 클래스에 추가해주었다. 그리고 insert는 레드블랙트리의 특성에 만족하게 구현하였다. 레드블랙크리의 특성은 루트는 블랙이다, 모든 리프(NIL)는 블랙이다, 노드가 레드이면 그 노드의 자식은 반드시 블랙이다, 루트 노드에서 임의의 리프 노드까지 이르는 경로에는 같은 수의 블랙 노드가 있다라는 것이다. 이 점을 유의하여 insert와 fix 그리고 rotate를 구현하였다. 특히 roatate는 이진탐색트리의 특성을 유지하면서 왼쪽 혹은 오른쪽으로 회전하는 방법이다. 레드블랙트리에서는 삽입이나 삭제 후 레드블랙특성을 유지하기 위해 사용된다. search함수는 루트 노드에서부터 비교하면서 왼쪽 오른쪽으로 이동하면서 찾게 구현하였다.

2. 주요 부분 코드 설명(알고리즘 부분 코드 캡쳐)

=>BinarySearchTree 부분은 저번과제와 겹치므로 생략하였다.

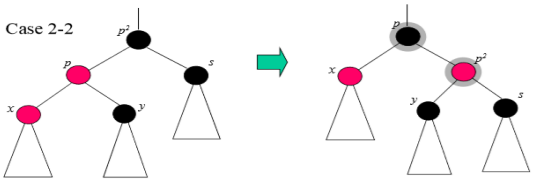
```
29      // 삽입
30      public void insert(int data) {
31          Node newNode = new Node(data); // 삽입할 노드 생성
32          insert(newNode);
33      }
34
35      private void insert(Node newNode) {
36          Node parent = root; // 루트노드값을 복사해논다(삽입할려는 노드의 부모노드 가리키는 용도로 쓴다)
37          if (root == nil) { // 루트노드가 nil(리프노드)인 경우
38              root = newNode; // 삽입한 노드가 루트노드가됨
39              newNode.color = BLACK; // 루트노드의 색은 BLACK이여야함
40              newNode.parent = nil; // 루트노드의 부모는 nil 이여야한다
41          } else { // 처음 삽입하는 노드가 아닐 경우
42              newNode.color = RED; // 일단 삽입할 때의 색은 RED이여야함
43              while (true) { // 노드 삽입완료 될 때까지 반복
44                  if (newNode.data < parent.data) { // 삽입할려는 노드 값이 parent(항상 newNode의 부모노드) 값보다 작을 경우
45                      if (parent.left == nil) { // 부모노드의 왼쪽 자식이 nil인 경우 그 자리에 newNode삽입
46                          parent.left = newNode;
47                          newNode.parent = parent;
48                          break;
49                      } else { // nil이 아니라면 부모노드를 왼쪽 자식노드로 하고 다시 반복문을 돌린다
50                          parent = parent.left;
51                      }
52                  } else if (newNode.data >= parent.data) { // 삽입할려는 노드 값이 parent(항상 newNode의 부모노드) 값보다 클 경우
53                      if (parent.right == nil) { // 부모노드의 왼쪽 자식이 nil인 경우 그 자리에 newNode삽입
54                          parent.right = newNode;
55                          newNode.parent = parent;
56                          break;
57                      } else { // nil이 아니라면 부모노드를 오른쪽 자식노드로 하고 다시 반복문을 돌린다
58                          parent = parent.right;
59                      }
60                  }
61              }
62              // 삽입을 한 후 레드블랙트리 특성에 맞게 고쳐준다
63              fixTree(newNode);
64          }
65      }
```

=>insert 함수이다. 알고리즘에 대한 설명은 주석에 자세히 적어놓았다. 크게 세가지 케이스로 나누어서 코드를 구현하였다. 그 3가지 case는 이론시간에 배웠던 다음과 같은 경우이다.



Case 2-2: s가 블랙이고, x가 p의 왼쪽 자식

: 색상이 바뀐 노드



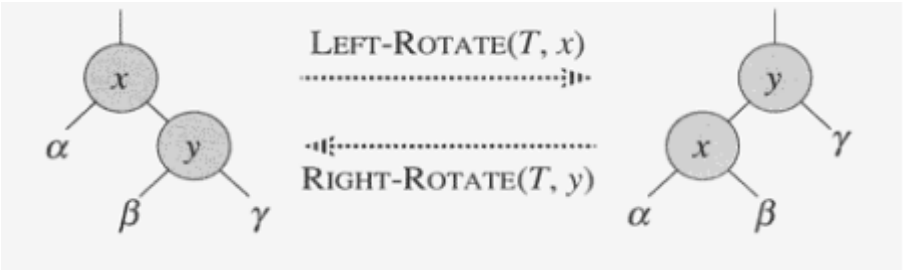
✓ 삽입 완료!

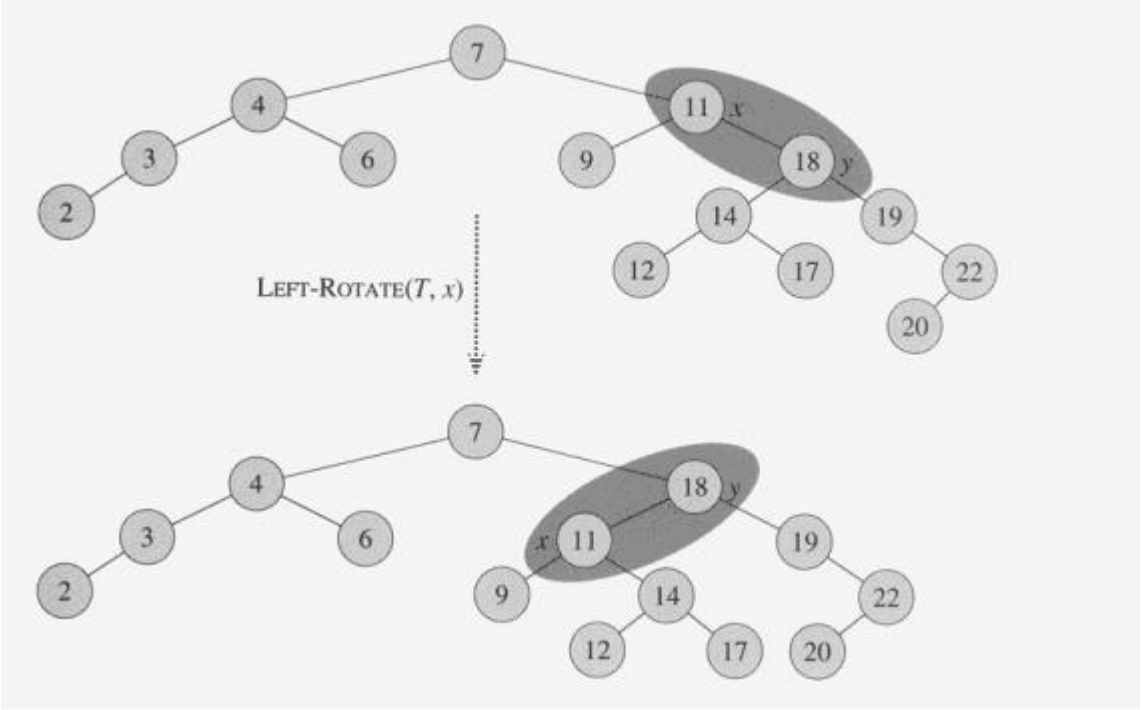
먼저 삽입할 때 root노드가 nil인지 확인하고 nil인 경우 삽입한 노드가 처음으로 삽입되는 루트노드이므로 레드블랙트리의 특성에 맞게 색은 black으로 하고 루트노드의 부모를 nil로 해준다. 만약 루트노드가 존재한다면 루트노드와 그 서브트리 노드들과 삽입할 노드의 키값을 비교하면서 알맞은 위치에 노드를 삽입한다. 이 때 레드블랙트리에서 삽입하는 노드의 색은 일단 Red이다. 삽입을 마치고 레드블랙트리의 특성에 맞게 트리를 balancing해준다. 그것을 fixTree함수에 구현하였다. 주석과 위에 그림과 마찬가지로 위와 같은 3가지 경우로 나눠서 구현을 하였고 else문은 루트노드의 오른쪽 서브트리일 경우인데 왼쪽에서 구현한것과 left와 right만 잘바꿔주고 똑같이 구현하면 된다.

```
// 왼쪽회전
void rotateLeft(Node node) {
    if (node.parent != nil) { // 노드의 부모가 nil노드일 때(즉 node.parent가 루트노드일 경우)
        if (node == node.parent.left) { // 노드가 부모의 왼쪽 자식일 경우
            node.parent.left = node.right; // 부모노드의 왼쪽자식에 현재노드의 오른쪽 자식을 이어준다.
        } else { // 오른쪽 자식인 경우는 부모노드의 오른쪽 자식노드에 현재노드의 오른쪽자식을 이어준다.
            node.parent.right = node.right;
        }
        node.right.parent = node.parent; // 부모노드의 오른쪽자식의 부모노드에 부모노드를 넣는다.
        node.parent = node.right; // 부모노드를 현재노드의 오른쪽자식노드로함
        if (node.right.left != nil) { // 오른쪽자식의 왼쪽자식 노드가 nil인 경우
            node.right.left.parent = node; // 오른쪽 자식의 왼쪽자식의 부모노드에 현재노드를 넣는다.
        }
        node.right = node.right.left; // 노드의 오른쪽 자식노드에 오른쪽자식의 왼쪽자식 노드를 넣는다.
        node.parent.left = node; // 부모노드의 왼쪽 자식에 현재노드를 넣는다.
    } else {
        Node right = root.right; // 임시 오른쪽 자식노드 right 저장
        root.right = right.left; // 루트의 오른쪽 자식노드에 임시로 저장한 right노드의 왼쪽 자식노드를 넣는다
        right.left.parent = root; // 오른쪽 자식의 왼쪽 자식의 부모노드에 루트노드를 넣는다
        root.parent = right; // 루트노드의 부모노드에 임시저장했던 right 노드를 넣는다.
        right.left = root; // rigth노드의 왼쪽 자식에 루트노드를 넣는다
        right.parent = nil; // 루트노드의 부모를 nil로 해준다.
        root = right; // 루트 노드를 right로 해준다.
    }
}
```

=>왼쪽회전함수이다. rotateRight는 왼쪽오른쪽만 바뀌고 똑같으므로 보고서에

생략하였다.) 코드 주석에 설명을 써놓았고 간략히 동작구조를 보이자면 다음 사진과 같다. 삽입할려는 노드 x를 기준으로 회전방향에 맞게 회전하고 y의 왼쪽 서브트리는 왼쪽에 붙이고 오른쪽 서브트리는 y의 오른쪽에 그대로 붙이면 된다. 즉 삽입노드를 알맞은 방향으로 회전시키고 x와 y의 서브트리를 알맞게 연결해주면 된다.





```
163     public Node search(int data) {
164         Node searchNode = new Node(data);
165         Node result = search2(searchNode, root);
166         System.out.println("Find Node(RBT) :  " + result.data);
167         return result;
168     }
169     public Node search2(Node searchNode, Node root) {
170         if (root == nil) { //루트노드가 nil이면 null반환
171             return null;
172         }
173 
174         if (searchNode.data < root.data) {
175             if (root.left != nil) {
176                 return search2(searchNode, root.left);
177             }
178         } else if (searchNode.data > root.data) {
179             if (root.right != nil) {
180                 return search2(searchNode, root.right);
181             }
182         } else if (searchNode.data == root.data) {
183             return root;
184         }
185         return null;
186     }
```

=>search 함수이다. BST에서 구현한 search와 마찬가지로 루트 노드에서부터 찾는 노드의 키값을 비교하면서 내려오면서 찾고 찾으면 그 노드를 리턴해준다.

```
public int[] inorder() {
    inorder(root);
    return inordered_data;
}

// 중위순회 (오름차순 노드 키값 출력에 사용한다)
public void inorder(Node root) {
    if (root != nil) { // nil도 노드로 설정하고 -1이란 값으로 설정하였는데 nil노드는 출력안되게 하기위해
                       // 다음 if문 조건을 넣어 nil노드는 순회하지 않게하였다.(nil노드값이 떠도 상관은 없겠지만)
        inorder(root.left);
        System.out.println(root.data);
        inordered_data[num++] = root.data;
        inorder(root.right);
    }
}
```

=>중위순회함수이다. nil노드를 -1을 가진 노드로 해놨었기 때문에 안해도 상관은 없지만 if문으로 nil인 노드는 탐색하지 않게 하였다.

3. 결과(시간 복잡도 포함)

```
================================================
This is find time of RED BLACK TREE
Find Node(RBT) :  47
RBT : 251802
This is fine time of BINARY SEARCH TREE
Find Node(BST) : 47
BST : 267540
================================================
START Write File
```

먼저 레드블랙트리와 이진탐색트리의 47 키값을 예로 search(탐색) 시간을 비교해보았다. 데이터 표본이 작아서 큰 차이는 안나는것도 있겠지만 레드블랙트리가 벨런싱이 더 잘 되있어서 이진탐색트리보다 수행시간이 짧다는 것을 볼 수 있었다.

Below is BST intsert result:	Below is RBT insert result:
1	1
2	2
3	3
6	6
7	7
8	8
9	9
10	10
11	11
12	12
13	13
17	17
25	25
26	26
27	27
32	32
33	33
34	34
36	36
38	38
41	41
43	43
45	45
47	47

BST와 RBT의 결과 텍스트파일이다. 중위순회한 결과 모두 알맞게 들어감을 볼 수 있었다.

레드블랙 트리는 깊이가 최악의 경우에도 O(nlogn)이 되는 것을 보장한다. 단 하나의 노드로 된 트리의 깊이는 1이라 정의한다. 키의 총 수가 n이라는 것은 레드 블랙 트리의 내부 노드의 수가

n임을 뜻한다. 레드나 블랙의 색상을 고려하지 않을 때 가장 이상적으로 꽉 채워진 트리의 깊이는 [logn] + 1 이다. 그러므로 레드 블랙 트리가 아무리 잘 만들어져도 루트에서 임의의 리프에 이르는 경로에 존재하는 블랙 노드의 개수는 [logn] + 1 을 넘을 수 없다. 레드 블랙 특성에 따라 레드 노드는 두 개가 연속해서 존재할 수 없으므로 루트에서 임의의 리프에 이르는 경로에서 블랙 노드의 개수보다 많을 수 없다. 그러므로 루트에서 임의의 리프에 이르는 경로의 길이는 2([logn]+1)을 넘을 수 없다. 이 것은 O(logn)이다. 따라서 키의 총수가 n인 레드 블랙 트리의 가능한 최대 깊이는 O(logn)이다.

그에 반해 이진 탐색 트리는 최악의 경우에는 O(n)의 시간복잡도를 가질 수 도 있다. 평균적으로는 O(nlogn)의 시간복잡도를 가지진 말이다.

결론은 그러므로 레드 블랙 트리가 성능이 더 좋다고 볼 수 있다는 점이다. 특히 탐색 관해서는 시간복잡도가 더 좋다.