알고리즘 과제(04)

201404377 진승언

1. 과제 목표 및 해결 방법

=> 이진 탐색 트리란 특정 노드가 가지는 좌측 자식은 항상 해당 노드의 데이터보다 작은 데이터를 가 지고 우측 자식은 큰 데이터를 가지는 트리이다. 이번 과제는 이진 탐색 트리를 구현하고 그 중에서 보통의 insert와 삽입할데이터를 미리 정렬을 시키고 그 가운데 값부터 차례대로 삽입 시키는 median insert 비교하는 것이 가장 큰 목표였다. 먼저 구현해야 할 것은 insert, Median insert, Recursive Search, Iterative Search, Successor, Predecessor, Delete였고 파일입출력을 이용해서 텍스트파일을 입력 받고 출력시키는 것이었다 .

Insert는 주어진 순서대로 데이터를 이진 탐색 트리에 추가하게 구현 하였고 median insert는 주어진 데이터를 정렬 한 뒤 그 중간 값부터 순서대로 이진 탐색트리에 추가하였다. 또 이진 탐색트리의 주어진 데이터의 존재유무를 반환하는 함수인 search는 Recursive와 Iterative로 나눠서 구현하였다. 전자는 재귀의 형태로 구현하고 후자는 일반적인 반복문의 형태로 구현하였다. Successor(후임자)와 Predecessor(전임자)는 Predecessor(전임자)는 좌측 서브트리에서 가장 큰 데이터를 가진 노드를 반환하게 하였고 Successor(후임자)은 우측 서브트리에서 가장 작은 데이터를 가진 노드를 반환하게 하였다.(즉 후임자는 루트노드보다 큰 값 중 가장 작은 값이고, 전임자는 루트노드보다 작은 값 중 가장 큰 값이다.)

Delete는 이진 탐색 트리에서 구조를 깨지 않고 원소를 지우는 함수인데 3가지 유형으로 나눠서 지운는 방식을 달리하였다. 첫번 째는 지우고자 하는 노드가 자식이 없을 때와 두번 째는 지우고자 하는 노드가 하나의 자식을 가질 떄 세번쨰는 지우고자 하는 노드가 둘의 자식을 가질 때로 나눴다.

2. 주요 부분 코드 설명(알고리즘 부분 코드 캡쳐)

<주석에 자세하게 설명 적었습니다>

```
public void insert(int data) {
    if(root == null) { //루트노드가 null이면 루트노드에 해당값으로 생성해준다.
        root = new TreeNode(data);
    }
    else { //기존 루트노드가 비어있지 않으면 즉 처음넣는값이 아닌 경우
        TreeNode newNode = new TreeNode(data);
        insertKey(this.root, newNode);
    }
}


//트리에 데이터 삽입
public void insertKey(TreeNode root , TreeNode newNode) {
    TreeNode parent = null;  //x의 부모노드 역할함
    TreeNode x = root;  //root노드로 받아옴
    while(x != null) {
        parent = x;
        if(newNode.data < x.data) { //새로삽입할려는 노드가 현재 비교노드 보다 작은 경우(처음 비교노드는 루트노드)
            x = x.left;              //비교노드를 왼쪽자식노드로바꿈
        }
        else {
            x = x.right;    //비교노드를 오른쪽 자식노드로 바꿈
        }
    }
    newNode.parent = parent;  //삽입하는 노드의 부모를 parent노드로 해줌

    //마지막 비교까지 온 노드와(즉 리프노드까지 비교) 비교해서 해당비교노드의 왼쪽오른쪽 자식노드에 newNode 삽입
    if(newNode.data< parent.data) {
        parent.left = newNode;
    }
    else {
        parent.right = newNode;
    }

}
```

1) Insert : 먼저 insert에 데이터 값을 넣어준다. 그럼 insert에서 만약 이 노드가 처음 삽입하는 노드라 루트노드가 없는 상태인 경우(null) 삽입할려는 노드를 루트노드로 넣어주고 끝난다. 만약 루트노드가 존재하는 경우는 받은 insertKey노드를 호출한다. InsertKey에서는 주석에서 설명한거와 같이 삽입할려는 노드를 루트노드에서부터 서브트리의 노드들 까지 리프노드가 나올 때 까지 비교하고 맞는 자리에 삽입할려는 노드를 연결해주면 된다.

```
20      // median_insert는 기존데이터를 차례대로 삽입하지 않고 퀵정렬을 해줌(그 후 정렬한데이터의 중간값을 넣어주는
21      // median_insert2를 호출)
22      public void median_insert(int data[]) {
23          Quick_Sort quick = new Quick_Sort();
24          quick.sort(data, 0, data.length - 1); // 삽입전에 데이터 퀵정렬함
25          median_insert2(data, 0, data.length - 1);
26      }
27
28      // 중앙값 차례대로 삽입
29      public void median_insert2(int data[], int start, int end) { // 입력데이터배열을 받음
30
31          // 중앙값을 삽입해주는 알고리즘
32          if ((end - start + 1) < 1)
33              return;
34
35          int median = (start + end) / 2;
36
37          if (root == null) { // 루트노드가 null이면 루트노드에 해당값으로 생성해준다.
38              root = new TreeNode(data[median]);
39          } else { // 기존 루트노드가 비어있지 않으면 즉 처음넣는값이 아닌 경우
40              TreeNode newNode = new TreeNode(data[median]);
41              insertKey(this.root, newNode);
42          }
43          // 나머지 값들도 다 중간값 삽입이 되게 재귀를 돌려줌
44          median_insert2(data, start, median - 1);
45          median_insert2(data, median + 1, end);
46
47      }
```

2),median insert: 먼저 삽입하기전에 삽입정렬로 데이터들을 정렬해주었다. 그 후 median_insert2에서 중간값을 삽입해줄 수 있도록 재귀문을 사용하여 구현하였다.

```java
    // 재귀 탐색
    public TreeNode recursive_search(int k) {
        return recursive_search2(root, k);
    }

    public TreeNode recursive_search2(TreeNode x, int k) {
        if (x == null || k == x.data) { //해당 값을 가진 노드를 찾았을 때 해당노드 반환
            return x;
        }
        if (k < x.data) { //x가 비교하는 노드보다 작거나 클 때 알맞게 재귀를 돌린다.
            return recursive_search2(x.left, k);
        } else {
            return recursive_search2(x.right, k);
        }
    }

    // 반복문 탐색
    public TreeNode iterative_search(int k) {
        return iterative_search2(root, k);
    }

    public TreeNode iterative_search2(TreeNode x, int k) {
        //해당 값을 가진 노드가 나올 때 까지 반복문을 돌린다
        while (x != null && k != x.data) {
            if (k < x.data) { //기준노드 보다 찾으려는 값이 작으면 기준노드를 왼쪽자식으로 체인지
                x = x.left;
            } else {    //기준노드 보다 크면 기준노드를 왼쪽자식으로 체인지
                x = x.right;
            }
        }
        return x;
    }
```

3) Recursive search/ Iterative search : Recursive search는 해당 값을 가진 노드를 기준노드와 비교하면서 기준노드의 서브트리 노드들과도 끝까지 비교할 수 있게 재귀호출을하여 찾는다. Iterative search는 위와 똑 같은 목적으로 반복문을 이용하여 찾는다.

```java
    // 후임자(오른쪽 서브트리의 최솟값)
    public TreeNode sucessor(TreeNode x) {

        if (x.right != null) {  //먼저 해당노드의 오른쪽 자식이 있어야한다(루트의 오른쪽서브트리 중 가장 작은 값을 찾는것이기 때문)
            return treeMinimum(x.right);  //가장 작은 값을 반환해준다.
        }
        //오른쪽 자식이 null인 경우
        TreeNode y = x.parent;
        while (y != null && x == y.right) {
            x = y;
            y = y.parent;
        }
        return y;
    }

    // 자식 서브트리중 가장 작은 노드 반환
    public TreeNode treeMinimum(TreeNode x) {
        while (x.left != null) {
            x = x.left;
        }
        return x;
    }

    // 선임자(왼쪽 서브트리의 최댓값)
    public TreeNode prodecessor(TreeNode x) {
        if (x.left != null) { //왼쪽자식노드가 있는지 확인
            return treeMaximum(x.left);  //최댓값을 가진 노드를 반환함
        }
        //왼쪽 자식이 null인 경우
        TreeNode y = x.parent;
        while (y != null && x == y.left) {
            x = y;
            y = y.parent;
        }
        return y;
    }

    // 자식서브트리중 가장 큰 값 반환
    public TreeNode treeMaximum(TreeNode x) {
        while (x.right != null) {
            x = x.right;
        }
        return x;
    }

```

4) Successor(후임자)/ Processor(전임자) : 먼저 후임자는 루트의 오른쪽 서브트중 가장 작은 값을 반환하는 메소드이고 전임자는 루트의 왼쪽 서브트리중 가장 큰 값을 반환하는 메소드이다. 그래서 처음에 왼쪽/오른쪽 자식이 있는지 확인 후에 treeMinimum/treeMaximum 메소드를 호출하여 해당 역할에 맞는 노드를 반환시켜주었다.

```java
    public void delete(TreeNode root, TreeNode removed) {
        if (removed.left == null ) { //지우고자 하는 노드가 자식이 없을 때
            transplant(root, removed, removed.right); // 지우고자 하는 노드를 삭제한다
        } else if (removed.right == null) { // 지우고자 하는 노드가 하나의 자식을 가질 때
            transplant(root, removed, removed.left); // 지우고자 하는 노드의 자식을 지우고자 하는 노드의 위치로 이동한다.
        } else { // 지우고자 하는 노드가 둘의 자식을 가질 때
            TreeNode y = treeMinimum(removed.right);
            if (y.parent != removed) { //지우고자 하는 노드의 후임과 교체한다.
                transplant(root, y, y.right);
                y.right = removed.right;
                y.right.parent = y;
            }
            transplant(root, removed, y); //위의 다른 유형을 적용하여 지우고자 하는 노드를 지운다.
            y.left = removed.left;
            y.left.parent = y;
        }

    }

    //노드를 삭제하고 자리이동시켜준다.
    public void transplant(TreeNode root, TreeNode u, TreeNode v) {
        if (u.parent == null) {
            root = v;
        } else if (u == u.parent.left) {
            u.parent.left = v;
        } else {
            u.parent.right = v;
        }
        if (v != null) {
            v.parent = u.parent;
        }
    }
```

5) Delete : 삭제는 지우고자 하는 노드가 자식이 없을 때, 하나 있을 때, 둘 다 있을 때에 따른 경우의 수 3가지로 나눠서 구현하였다. 지우고자 하는 노드가 자식이 없을 때는 지우고자 하는 노드를 지워주었고, 하나의 자식을 가질 때는 지우고자 하는 노드의 자식을 지우고자 하는 노드의 위치로 이동시켰다. 마지막으로 자식 2개 모두 있을 경우는 지우고자 하는 노드의 후임과 교체시키고 위에 다른 1,2번 유형을 적용하여 지우고자 하는 노드를 지워주었다. Transplant가 이 역할을 하는 메소드이다.

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

public void inorder(TreeNode root) {
    if (root != null) {
        inorder(root.left);
        System.out.println(root.data);
        inordered_data[num++] = root.data;
        inorder(root.right);
    }
}
```

6) 중위순회 메소드 이다. 이진 탐색 트리를 출력시켜주는 역할을 담당했다.

3. 결과(시간 복잡도 포함)

```
================================================
this is recursive_search result EX)26(10개짜리텍스트파일),30(20개짜리텍스트파일)
26
30
================================================
this is iterative_search result EX)26(10개짜리텍스트파일),30(20개짜리텍스트파일)
26
30
================================================
this is sucessor result
sucessor of Data1.txt tree: 26
sucessor of Data2.txt tree: 37
================================================
this is prodecessor result
prodecessor of Data1.txt tree: 17
prodecessor of Data2.txt tree: 35
================================================
START Write File
```

=>콘솔로 몇 가지 메소드를 구현해 본 결과이다. 알맞게 됨을 볼 수 있었다.(손으로 직접 그려서 확인해보았다.)

```
Below is BST insert 10 result

11
12
13
17
25
26
27
32
34
38
-------------------------------------------------Below is BST median_insert 10 result

11
12
13
17
25
26
27
32
34
38
-------------------------------------------------
```

=>data10의 insert와 median insert한 결과를 중위순회로 출력한 결과이다.

```
Below is BST intsert 20 result

1
4
8
11
15
16
17
18
21
23
29
30
35
36
37
40
41
44
46
49
-----------------------------------------------Below is BST median_insert  20 result

1
4
8
11
11
15
16
17
18
21
23
29
30
35
36
37
40
41
44
46
49
-----------------------------------------------
```

=>data20의 insert와 median insert한 결과를 중위순회로 출력한 결과이다.

이진 탐색 트리에서의 탐색, 삽입, 삭제 연산의 시간 복잡도는 트리의 높이를 h라고 했을 때 O(h)가 된다. 따라서 n개의 노드를 가지는 이진 탐색 트리의 경우, 균형 잡힌 이진 트리의 높이는 log2n이므로 이진 탐색 트리 연산의 평균적인 경우의 시간 복잡도는 O(log2n)이다.

이번에 그냥 보통의 무작위로 데이터를 삽입하는 insert와 퀵정렬을 해서 정렬된 데이터의 중간값부터 차례대로 넣는 median insert의 시간복잡도를 비교해봤는데 무작위로 insert하는 것보다 median insert가 트리를 균형있게 만들어 주어 이진트리를 탐색하는 면에서는 시간복잡도 면에서 더 훌륭하였다. 그러나 median insert는 O(nlogn)의 시간복잡도를 가지는 퀵정렬을 사용하므로 insert하는 면에서는 최악이여도 삽입이 O(n)의 시간복잡도를 가지는 보통의 insert보다 안 좋은 것 같다.