알고리즘

- 04. BST (Binary Search Tree) -

제 출 일	2018.10.29
학 과	컴퓨터공학과
학 번	201402391
이 름	이 병 만

※ 과제 목표 및 해결 방법

- 배열로 구현.
- Insert, Median Insert, Recursive/Iterative Search, Successor, Predecessor, Delete를 구현하여 입출력 파일 방식으로 결과를 도출하여라.

※ 주요 부분 코드 설명 (알고리즘 부분 코드 캡쳐)

1. BST

```
public class Node {
    int data;
    Node left;
    Node right;

    public Node(){
        this.left = null;
        this.right = null;
    }

    public Node(int data){
        this.data = data;
        this.left = null;
        this.right = null;
    }
}
```

[그림1. Node Class]

➔ Node 클래스로 data와 left, right를 가지고 노드를 생성한다. 생성한 노드는 데이터의 값을 가지고 노드의 left, right는 null의 값을 가진다.

```
public BST() {
    this.root = null;
}
```

[그림2. 생성자]

➔ 생성자로서 root을 생성한다.

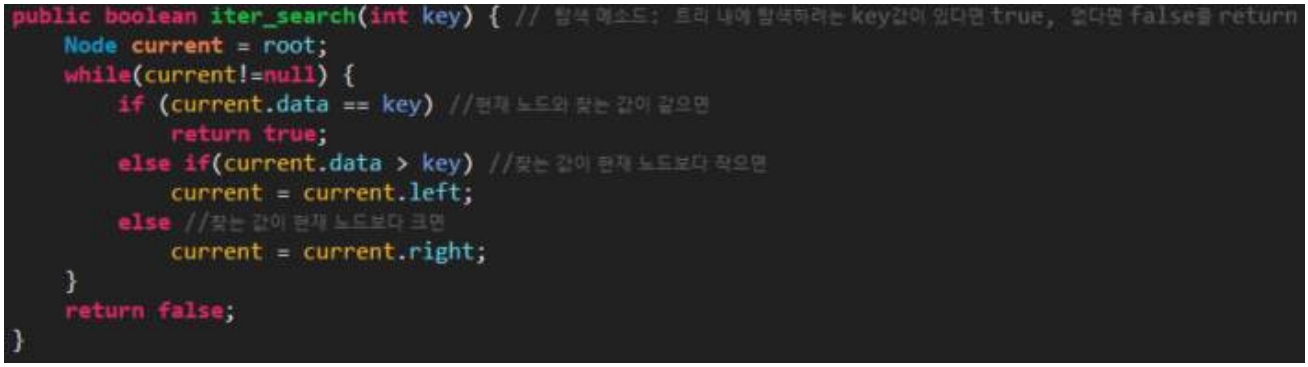
```
public void insertBST(int key) {
    root = insert(root, key);
}

public Node insert(Node root, int key) {
    Node p = root;
    Node newNode = new Node(key);

    if (p == null)
        return newNode;
    else if (p.data > newNode.data) {
        p.left = insert(p.left, key);
        return p;
    }
    else if (p.data < newNode.data) {
        p.right = insert(p.right, key);
        return p;
    }
    else
        return p;
}
```

[그림3. insertBST]

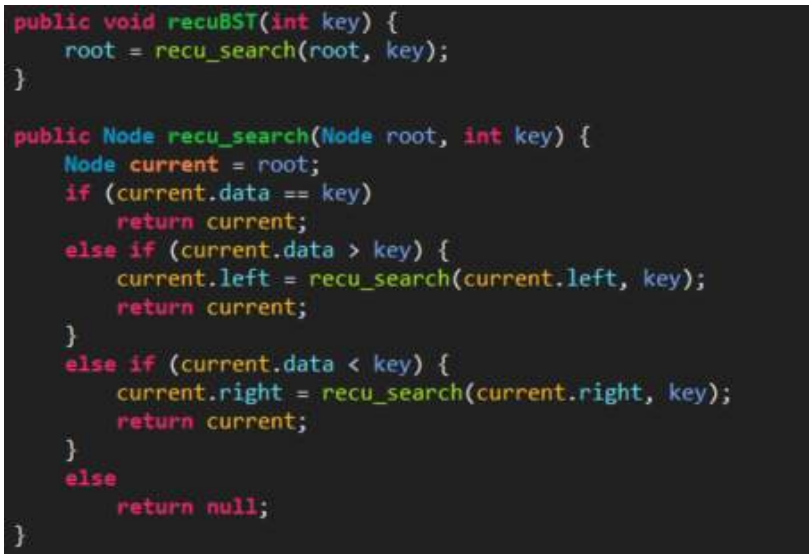
➔ 트리를 생성하는 함수로서 insertBST에서 root와 추가 할 key의 값을 받는다. 루트가 null이면 새로운 노드를 반환한다. 받은 key의 값을 비교하면서 p보다 작으면 왼쪽으로 크면 오른쪽으로 값을 삽입하고 반환한다.



```
public boolean iter_search(int key) { // 탐색 메소드: 트리 내에 탐색하려는 key값이 있다면 true, 없다면 false를 return
    Node current = root;
    while(current!=null) {
        if (current.data == key) //현재 노드와 찾는 값이 같으면
            return true;
        else if(current.data > key) //찾는 값이 현재 노드보다 작으면
            current = current.left;
        else //찾는 값이 현재 노드보다 크면
            current = current.right;
    }
    return false;
}
```

[그림4 iter_search]

➔ iterative하게 tree를 search하는 함수이다. 인자로는 검색하고 싶은 값을 넣어준다. 처음에 root를 현재로 저장하고 현재 노드의 값과 찾으려는 값이 같을 때까지 반복한다. 작으면 현재 노드의 왼쪽 크면 오른쪽으로 이동해서 값을 찾는다. 찾으면 true를 반환해주고 찾지 못하면 false를 반환한다.



```
public void recuBST(int key) {
    root = recu_search(root, key);
}

public Node recu_search(Node root, int key) {
    Node current = root;
    if (current.data == key)
        return current;
    else if (current.data > key) {
        current.left = recu_search(current.left, key);
        return current;
    }
    else if (current.data < key) {
        current.right = recu_search(current.right, key);
        return current;
    }
    else
        return null;
}
```

[그림5. recu_search]

➔ recursive하게 tree를 search하는 함수이다. 인자로는 동일하게 검색하고 싶은 값을 넣어준다. 처음에 root를 현재로 저장하고 현재의 데이터와 검색하고자하는 키의 값이 같은지 비교한다. 작으면 루트의 값을 현재의 왼쪽으로 재귀를 실행하고, 크면 현재의 오른쪽으로 재귀를 실행한다.

```
public void medianBST(ArrayList<Integer> list) {
    root = median_insert(list, 0, list.size() - 1);
}

public Node median_insert(ArrayList<Integer> list, int start, int end) {
    if (start > end )
        return null;
    int median_value = (start + end) / 2;
    Node node = new Node(list.get(median_value));
    node.left = median_insert(list, start, median_value - 1);
    node.right = median_insert(list, median_value + 1, end);
    return node;
}
```

[그림6. median_insert]

➔ median_insert함수는 정렬된 배열로부터 중간 값부터 순서대로 트리에 추가해서 생성하는 것이다. 인자로는 정렬된 데이터와 배열의 처음 인덱스와 마지막 인덱스를 받는다. 중간 값을 결정하고 그 기준으로 노드의 왼쪽, 오른쪽에 삽입해준다. start의 값이 end보다 커지면 null을 반환한다.

```java
public ArrayList<Integer> inorder(Node root, ArrayList<Integer> list){
    if(root!=null){
        inorder(root.left, list);
        System.out.print(root.data + " ");
        list.add(root.data);
        inorder(root.right, list);
    }
    return list;
}

public ArrayList<Integer> printBST(ArrayList<Integer> list) {
    inorder(root, list);
    System.out.println();
    System.out.println();
    return list;
}
```

[그림7. inorder]

➔ 중위순회를 하는 함수이다. 중위순회를 통해서 출력하면 오름차순으로 트리에서 값이 출력되는 것을 알 수 있다. 여기서는 파일입출력을 위해서 리스트를 반환받는 과정을 추가하여 구현하였다.

```java
public Node successor(Node data) {
    Node successsor = null;
    Node successsorParent = null;
    Node current = data.right;

    while (current != null) {
        successsorParent = successsor;
        successsor = current;
        current = current.left;
    }
    if (successsor != data.right) {
        successsorParent.left = successsor.right;
        successsor.right = data.right;
    }
    return successsor;
}
```

[그림8. successor]

➔ successor는 current의 부모 node를 pointer로 사용한다. successorParent는 current의 조부모 node를 pointer로 사용한다. 삭제 할 노드의 우측 자식 노드 주소를 current pointer에 저장한다. 반복문을 통해서 current 값이 없을 때까지 반복한다. 좌측의 node 주소를 계속 찾음. successsor가 삭제 노드의 우측 자식 노드가 아닐 경우 successsor의 우측 자식 노드를 sucessor 의 자리를 대체한다. successsor의 우측 자식 노드 pointer는 삭제 될 node 의 우측 자식노드의 주소를 가르킨다.

```java
public Node predecessor(Node data) {
    Node predecessor = null;
    Node predecessorParent = null;
    Node current = data.left;

    while (current != null) {
        predecessorParent = predecessor;
        predecessor = current;
        current = current.right;
    }
    if (predecessor != data.left) {
        predecessorParent.right = predecessor.left;
        predecessor.left = data.left;
    }
    return predecessor;
}
```

[그림8. successor]

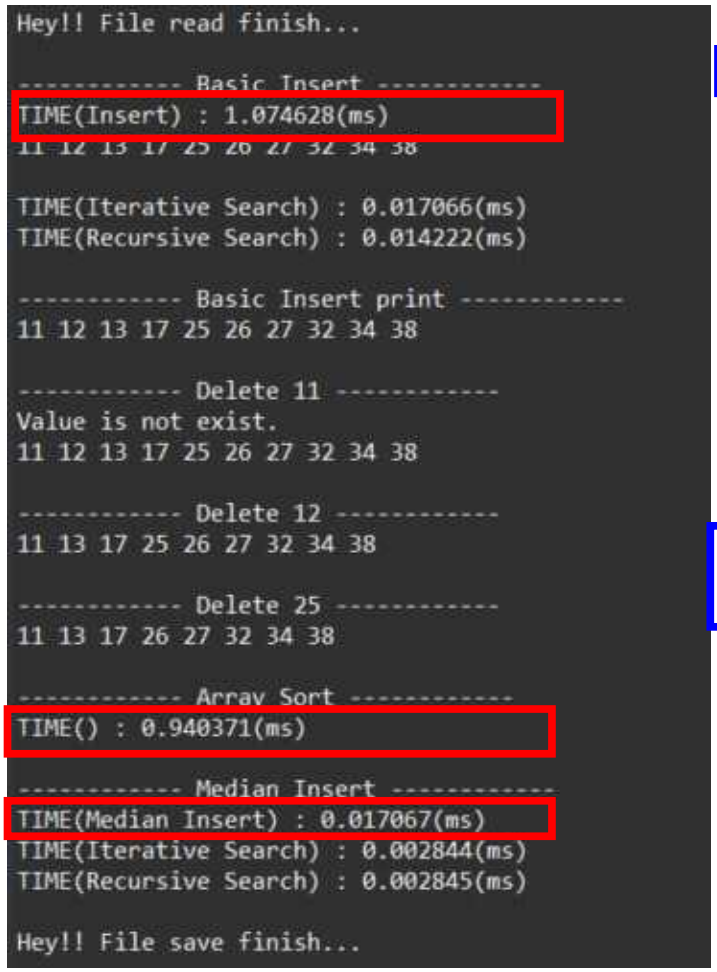
➔ predecessor는 current의 부모 node를 pointer로 사용한다. predecessorParent는 current의 조부모 node를 pointer로 사용한다. 삭제 할 노드의 좌측 자식 노드 주소를 current pointer에 저장한다. 반복문을 통해서 current의 값이 없을 때까지 반복한다. predecessor가 data의 좌측 자식이 아닐 경우 predecessorParent의 우측노드를 predecessor의 좌측노드로 저장하고 data의 좌측을 predecessor의 왼쪽에 저장한다.

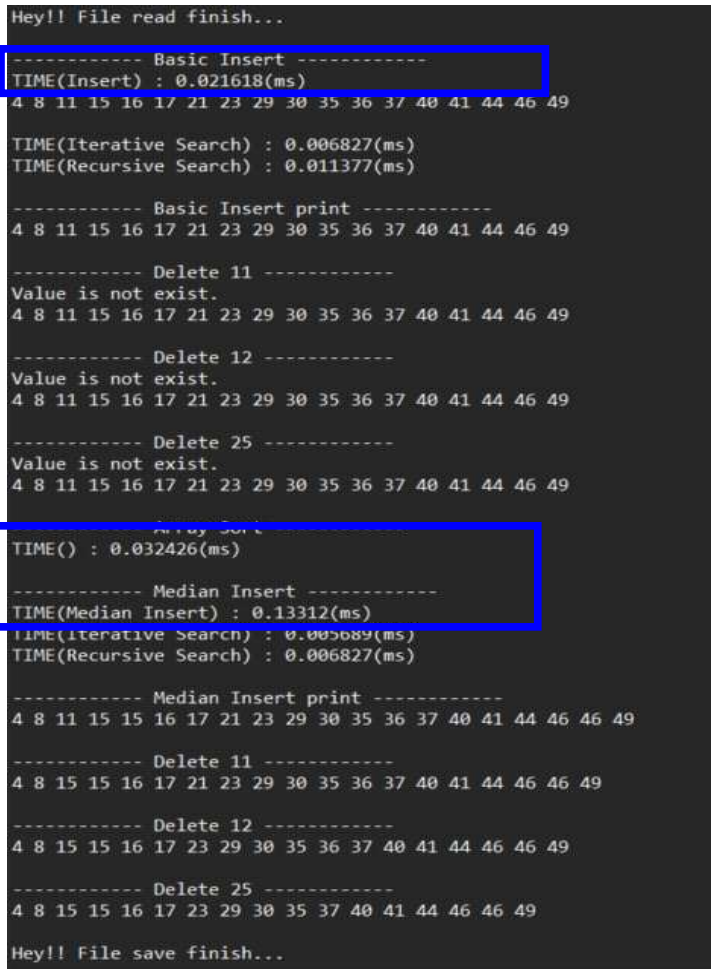
```java
public boolean delete(int key){
    Node parent = root;
    Node current = root;
    boolean isRightchild = false;
    if (!iter_search(key)) {
        System.out.println("Value is not exist.");
        return false;
    }
    while (current.data != key) {
        parent = current;
        if (current.data > key) {
            current = current.left;
            isRightchild = false;
        }
        else if (current.data < key){
            current = current.right;
            isRightchild = true;
        }
        else
            return false;
    }
    if (current.left == null & current.right == null) {
        if (current == root)
            root = null;
        if (isRightchild)
            parent.right = null;
        else
            parent.left = null;
    }
    else if (current.left == null) {
        if (current == root)
            root = current.right;
        else if (isRightchild)
            parent.right = current.right;
        else
            parent.left = current.right;
    }
    else if (current.right == null) {
        if (current == root)
            root = current.left;
        else if (isRightchild)
            parent.right = current.left;
        else
            parent.left = current.left;
    }
    else if (current.left != null && current.right != null) {
        Node successor = successor(current);
        if (current == root)
            root = successor;
        else if (isRightchild)
            parent.right = successor;
        else
            parent.left = successor;
        successor.left = current.left;
    }
    return true;
}
```

[그림9. delete]

➔ delete함수를 구현하는 것이다. 삭제할 키의 값을 인자로 받는다. search를 통해서 삭제할 키가 있는지 없는지 확인한다. 없다면 함수를 종료하고 있다면 삭제할 키의 위치를 찾는다. 반복문을 통해서 키의 위치가 있는 노드까지 이동한다. 키를 찾았으면 그 다음 3가지의 경우가 있다. 삭제할 키의 자식이 없는 경우, 1개만 있는 경우, 2개 있는 경우를 다 고려해서 삭제해줘야 한다. 자식이 없는 경우에는 그냥 삭제를 해주면 되고 만약 1개만 있는 경우에서는 왼쪽만 있는 경우 오른쪽만 있는 경우를 나눠서 삭제한다.
2개가 있는 경우에는 successor를 구해서 가장 큰 데이터를 찾아서 그 값으로 대체해서 삭제해준다.

※ 실행 결과(시간 복잡도 포함)





실행한 결과 Insert보다도 Median_insert의 실행시간이 적다는 것을 알 수 있다. 정렬하는 것까지 포함하더라도 insert의 실행시간이 확실하게 느리다는 것을 알 수 있다.