

자료구조 및 실습 보고서

[제08주] DS_08_201502273_김현종

2018.05.13.

201502273/김현종

1.내용

```
package tree;

import Queue.LinkedList;

public class BinaryTree {
    private BinaryNode root;
    private String order;

    /*
     * BinaryTree의 Constructor
     * 저장해야하는 value와 저장할 order를 나타내는
     * 두 파라미터를 받는다.
     * order를 저장하고 저장할 value를 parse한다.
     */
    public BinaryTree(String value, String order) {
        // Don't save the value.
        this.order = order;
        parse(value);
    }

    /*
     * 저장할 String을 parse하는 메소드
     * this.order의 값에 따라
     * parseToPreOrder, parseToPostOrder,
     * parseToLevelOrder, parseToInorder를 호출하여
     * 값을 넘긴다.
     */
    private void parse(String value) {
        if ("preOrder".equals(this.order)) {
            parseToPreOrder(value);
        } else if ("postOrder".equals(this.order)) {
            parseToPostOrder(value);
        } else if ("levelOrder".equals(this.order)) {
            parseToLevelOrder(value);
        } else {
            parseToInorder(value);
        }
    }

    /*
     * 입력값을 inorder로 parse하는 메소드
     * 입력 값이 비었을 경우 값 할당 없이 root를
     * BinaryNode의 객체로 할당한다.
     * 입력 값이 있을 경우 makeInorderTree에 입력값을 넘기고
     * 반환값을 root로 취한다.
     */
    private void parseToInorder(String value) {
        if ("".equals(value)) {
            this.root = new BinaryNode();
        } else {
            this.root = makeInorderTree(value);
        }
    }

    /*
     * 입력된 값을 inorder Tree로 만들어 반환하는 메소드
     * 입력된 값이 비었으면 null을 반환하고
     * 값이 있으면 String의 중간에 있는 값을 찾아서
     * 처음 반을 재귀적으로 돌리고 얻은 결과를 왼쪽 트리,
     * String중간에 있는 값을 Node의 값으로 갖는 BinaryNode를 root로
     * 뒤쪽 반을 재귀적으로 돌리고 얻은 결과를 오른쪽 트리으로 하는 큰 트리를 반환한다.
     */
    private BinaryNode makeInorderTree(String value) {
        // TODO : Fill it
        if ("".equals(value))
            return null;

        int length = value.length();
        int middle = length / 2;
```

```

        BinaryNode left = makeInorderTree(value.substring(0, middle));
        BinaryNode root = new BinaryNode(value.substring(middle, middle + 1));
        BinaryNode right = makeInorderTree(value.substring(middle + 1, length));

        root.setLeft(left);
        root.setRight(right);

        return root;
    }

    /*
     * 입력값을 preorder로 parse하는 메소드
     * 입력 값이 비었을 경우 값 할당 없이 root를
     * BinaryNode의 객체로 할당한다.
     * 입력값이 있을 경우 makePreOrderTree에 입력값을 넘기고
     * 반환값을 root로 취한다.
     */
    private void parseToPreOrder(String value) {
        if ("".equals(value)) {
            this.root = new BinaryNode();
        } else {
            this.root = makePreOrderTree(value);
        }
    }

    /*
     * 입력된 값을 preorder Tree로 만들어 반환하는 메소드
     * 입력된 값이 비었으면 null을 반환하고
     * 값이 있으면 String의 처음에 있는 값을 찾아서
     * 처음 값을 값으로 갖는 Node를 root로
     * 남은 부분 중 처음 반을 재귀적으로 돌리고 얻은 결과를 왼쪽 트리,
     * 뒤쪽 반을 재귀적으로 돌리고 얻은 결과를 오른쪽 트리로 하는 큰 트리를 반환한다.
     */
    private BinaryNode makePreOrderTree(String value) {
        // TODO : Fill it
        if ("".equals(value))
            return null;

        int length = value.length();
        int middle = length / 2;
        BinaryNode root = new BinaryNode(value.substring(0, 1));
        BinaryNode left = makePreOrderTree(value.substring(1, middle + 1));
        BinaryNode right = makePreOrderTree(value.substring(middle + 1,
length));

        root.setLeft(left);
        root.setRight(right);

        return root;
    }

    /*
     * 입력값을 postorder로 parse하는 메소드
     * 입력 값이 비었을 경우 값 할당 없이 root를
     * BinaryNode의 객체로 할당한다.
     * 입력값이 있을 경우 makePostOrderTree에 입력값을 넘기고
     * 반환값을 root로 취한다.
     */
    private void parseToPostOrder(String value) {
        if ("".equals(value)) {
            this.root = new BinaryNode();
        } else {
            this.root = makePostOrderTree(value);
        }
    }

    /*
     * 입력된 값을 preorder Tree로 만들어 반환하는 메소드
     * 입력된 값이 비었으면 null을 반환하고
     * 값이 있으면 String의 중간에 있는 값을 찾아서
     * 처음 반을 재귀적으로 돌리고 얻은 결과를 왼쪽 트리,

```

```

    * 마지막 하나를 뺀 뒤쪽 반을 재귀적으로 돌리고 얻은 결과를 오른쪽 트리,
    * 마지막 값을 값으로 갖는 Node를 root로 하는 큰 트리를 반환한다.
    */
private BinaryNode makePostOrderTree(String value) {
    // TODO : Fill it
    if ("".equals(value))
        return null;

    int length = value.length();
    int middle = length / 2;
    BinaryNode left = makePostOrderTree(value.substring(0, middle));
    BinaryNode right = makePostOrderTree(value.substring(middle, length -
1));

    BinaryNode root = new BinaryNode(value.substring(length - 1, length));
    root.setLeft(left);
    root.setRight(right);

    return root;
}

/*
 * 입력값을 levelorder로 parse하는 메소드
 * 입력 값이 비었을 경우 값 할당 없이 root를
 * BinaryNode의 객체로 할당한다.
 * 입력값이 있을 경우 makeLevelOrderTree에 입력값을 넘기고
 * 반환값을 root로 취한다.
 */
private void parseToLevelOrder(String value) {
    if ("".equals(value)) {
        this.root = new BinaryNode();
    } else {
        this.root = makeLevelOrderTree(value);
    }
}

/*
 * 입력된 값을 levelorder Tree로 만들어 반환하는 메소드
 * Queue를 하나 선언하고 입력값의 첫번째 값을 값으로 갖는 Node를 하나 선언한다.
 * 그리고 q에 해당 node를 넣고 q가 빌때까지
 * Queue에서 값을 빼서 root로 정의하고 해당 번째 값을 갖는 BinaryNode를 선언하여
 * Queue에 넣어주고 root에 왼쪽 오른쪽에 붙인다.
 * 반복문이 끝난 후 node를 반환한다.
 */
private BinaryNode makeLevelOrderTree(String value) {
    // TODO : Fill it
    LinkedList q = new LinkedList();
    BinaryNode node = new BinaryNode(value.substring(0, 1));
    q.add(node);
    for (int i = 1; !q.isEmpty(); i+=2) {
        BinaryNode root = q.remove();
        if (i + 1 <= value.length()) {
            root.setLeft(new BinaryNode(value.substring(i, i + 1)));
            q.add(root.getLeft());
        }
        if (i + 2 <= value.length()) {
            root.setRight(new BinaryNode(value.substring(i + 1, i +
2)));
            q.add(root.getRight());
        }
    }

    return node;
}

/*
 * inorder로 Tree를 검색하는 메소드이다.
 * BinaryNode를 받아오고 StringBuilder를 선언하여
 * 왼쪽 자식노드가 있으면 해당 노드를 재귀적으로 돌리며 얻은 값을 append하고
 * 그리고 받은 Node가 가지고 있는 값을 append하고
 * 오른쪽 자식노드가 있으면 해당노드를 재귀적으로 돌리며 얻은 값을 append한다.

```

```

    * 그리고 만든 String을 반환한다.
    */
private String inorder(BinaryNode node) {
    // TODO : Fill it
    StringBuilder sb = new StringBuilder();
    if (node.hasLeft())
        sb.append(inorder(node.getLeft()));
    sb.append(node.getValue());
    if (node.hasRight()) sb.append(inorder(node.getRight()));

    return sb.toString();
}

/*
 * preorder로 Tree를 검색하는 메소드이다.
 * BinaryNode를 받아오고 StringBuilder를 선언하여
 * 받아온 Node가 가지고 있는 값을 append하고
 * 그리고 왼쪽 자식노드가 있으면 해당 노드를 재귀적으로 돌리며 얻은 값을
append하고
 * 오른쪽 자식노드가 있으면 해당 노드를 재귀적으로 돌리며 얻은 값을 append한다.
 * 그리고 만든 String을 반환한다.
 */
private String preOrder(BinaryNode node) {
    // TODO : Fill it
    StringBuilder sb = new StringBuilder();
    sb.append(node.getValue());
    // System.out.print(node.getValue());
    if (node.hasLeft())
        sb.append(preOrder(node.getLeft()));
    if (node.hasRight())
        sb.append(preOrder(node.getRight()));

    return sb.toString();
}

/*
 * postorder로 Tree를 검색하는 메소드이다.
 * BinaryNode를 받아오고 StringBuilder를 선언하여
 * 왼쪽 자식노드가 있으면 해당 노드를 재귀적으로 돌리며 얻은 값을 append하고
 * 오른쪽 자식노드가 있으면 해당 노드를 재귀적으로 돌리며 얻은 값을 append하고
 * 받아온 Node가 가지고 있는 값을 append한다.
 * 그리고 만든 String을 반환한다.
 */
private String postOrder(BinaryNode node) {
    // TODO : Fill it
    StringBuilder sb = new StringBuilder();
    if (node.hasLeft())
        sb.append(postOrder(node.getLeft()));
    if (node.hasRight())
        sb.append(postOrder(node.getRight()));
    sb.append(node.getValue());

    return sb.toString();
}

/*
 * levelOrder로 Tree를 검색하는 메소드이다.
 * BinaryNode를 받아오고 StringBuilder를 선언하고
 * Queue를 하나 선언한다.
 * Queue에 받아온 Node를 넣고 Queue가 빌때까지 반복문을 돌리며
 * Queue에서 값을 하나 빼고 빼온 값을 append하고 왼쪽 자식이 있으면 해당 노드를
Queue에 넣고
 * 오른쪽 자식이 있으면 해당 노드를 Queue에 넣어준다.
 * 위를 반복하고 반복문이 끝난 뒤 만든 String을 반환한다.
 */
private String levelOrder(BinaryNode node) {
    // TODO : Fill it
    LinkedList q = new LinkedList();
    StringBuilder sb = new StringBuilder();

    q.add(node);

```

```

        while (!q.isEmpty()) {
            BinaryNode root = q.remove();
            sb.append(root.getValue());
            if (root.hasLeft())
                q.add(root.getLeft());
            if (root.hasRight())
                q.add(root.getRight());
        }
        return sb.toString();
    }

    /*
     * 이 Tree에 있는 값을 반환하는 메소드이다.
     * 현재 Tree의 order에 따라 preOrder, postOrder,
     * levelOrder, inorder 함수를 호출하여 이 Tree의 root 노드를 넘겨주고
     * 해당 메소드에서 얻은 String을 반환한다.
     */
    public String getValue() {
        String value;
        if ("preOrder".equals(this.order)) {
            value = preOrder(this.root);
        } else if ("postOrder".equals(this.order)) {
            value = postOrder(this.root);
        } else if ("levelOrder".equals(this.order)) {
            value = levelOrder(this.root);
        } else {
            value = inorder(this.root);
        }
        return value;
    }

    /*
     * 이 Tree가 가지고 있는 value를 levelorder의 형태로 검색하며
     * 트리의 모습으로 터미널에 출력해주는 함수이다.
     */
    public void print() {
        LinkedQueue queue = new LinkedQueue(); //Queue 선언
        queue.add(this.root); //Queue에 root를 넣는다.
        int level = this.root.level(); //root의 level을 얻어온다.
        String interval = "%" + (int) Math.pow(2, this.root.height()) + "s";
        //2^(root.height)만큼 띄우는 형식
        StringBuilder tree = new StringBuilder(); //StringBuilder 선언
        while (!queue.isEmpty()) { //Queue가 빌때까지 반복
            BinaryNode currentNode = queue.remove(); //Queue에서
            Node를 하나 가지고와서
            if (currentNode != null) { //가져온 Node가 null이 아니면
                if (level < currentNode.level()) { //해당 Node의
                    level이 기존의 level보다 커졌으면
                    level = currentNode.level(); //level
                    갱신
                    interval = "%" + (int) Math.pow(2,
                        currentNode.height()) + "s"; //interval 재 계산
                    tree.append("\n"); //next line
                }
                queue.add(currentNode.getLeft()); //왼쪽 자식을
                queue.add(currentNode.getRight()); //오른쪽 자식을
                //interval만큼씩 띄워서 현재 Node의 값 append
                tree.append(String.format(interval,
                    currentNode.getValue()));
                } else { //가져온 Node가 null이면
                    //interval만큼씩 띄워서 "" append
                    tree.append(String.format(interval,
                        ""));
                }
            }
            System.out.println(tree.toString()); //반복문이 끝나고 만들어진 String 출력
        }
    }
}

```

BinaryTree.java

```

package tree;

public class BinaryNode {
    private String value;
    private BinaryNode parent;
    private BinaryNode left;
    private BinaryNode right;

    /*
     * BinaryNode의 Constructor
     */
    BinaryNode() {
    }

    /*
     * BinaryNode의 Constructor
     * 받은 value를 this.value에 할당
     */
    BinaryNode(String value) {
        this.value = value;
    }

    /*
     * 현재 노드의 level을 반환하는 메소드
     * 부모 Node가 없으면 1을 반환하고 그렇지 않으면 부모 노드의 level + 1을 반환한다.
     */
    public int level() {
        if (parent == null) {
            return 1;
        }
        return parent.level() + 1;
    }

    /*
     * 해당 Node의 height를 반환하는 메소드
     * 해당 Node가 Leaf이면 1을 반환한다.
     * 왼쪽 자식이 있으면 해당 노드의 height를 받아오고
     * 오른쪽 자식이 있으면 해당 노드의 height를 받아와
     * 둘 중 더 큰 값 + 1을 반환한다.
     */
    public int height() {
        // TODO : Fill it
        if(this.isLeaf()) return 1;
        else{
            int leftHeigth = 0;
            if(this.hasLeft()){
                leftHeigth = this.left.height();
            }
            int rightHeigth = 0;
            if(this.hasRight()){
                rightHeigth = this.right.height();
            }
            return leftHeigth > rightHeigth ? leftHeigth + 1 : rightHeigth + 1;
        }
    }

    /*
     * 자식이 둘 있으면 두 자식에게 재귀적으로 노드 수를 받아와 +1하여 반환하고
     * 왼쪽 자식이 있으면 왼쪽 자식이 갖고있는 노드수를 재귀적으로 받아와 +1하여 반환하고
     * 오른쪽 자식이 있으면 오른쪽 자식이 갖고있는 노드수를 재귀적으로 받아와 +1하여 반환하고
     * 자식이 없으면 1을 반환한다.
     */
    public int numberOfNodes() {
        // TODO : Fill it
        if(this.hasLeft() && this.hasRight()){
            return this.left.numberOfNodes() + this.right.numberOfNodes() + 1;
        }else if(this.hasLeft()){
            return this.left.numberOfNodes() + 1;
        }else if(this.hasRight()){
            return this.right.numberOfNodes() + 1;
        }else{
            return 1;
        }
    }
}

```

```

    }

    /**
     * value의 getter
     */
    public String getValue() {
        return value;
    }

    /**
     * 왼쪽 자식의 setter
     * 왼쪽 자식의 parent를 자신으로 지정한다.
     */
    public void setLeft(BinaryNode left) {
        if (left != null) {
            this.left = left;
            this.left.parent = this;
        }
    }

    /**
     * 오른쪽 자식의 setter
     * 오른쪽 자식의 parent를 자신으로 지정한다.
     */
    public void setRight(BinaryNode right) {
        if (right != null) {
            this.right = right;
            right.parent = this;
        }
    }

    /**
     * 왼쪽 자식의 getter
     */
    public BinaryNode getLeft() {
        return this.left;
    }

    /**
     * 오른쪽 자식의 getter
     */
    public BinaryNode getRight() {
        return this.right;
    }

    /**
     * 왼쪽 자식이 있는지 확인하는 메소드
     */
    public boolean hasLeft() {
        // TODO : Fill it
        return this.left != null;
    }

    /**
     * 오른쪽 자식이 있는지 확인하는 메소드
     */
    public boolean hasRight() {
        // TODO : Fill it
        return this.right != null;
    }

    /**
     * 이 노드가 Leaf인지 확인하는 메소드
     */
    public boolean isLeaf() {
        // TODO : Fill it
        return !(this.hasLeft() || this.hasRight());
    }
}

```

BinaryNode.java


```

package Queue;

import tree.BinaryNode;

public class Node {
    private Node next;
    private Node prev;
    private BinaryNode value;

    /*
     * Node의 Constructor
     */
    Node() {
        next = this;
        prev = this;
    }

    /*
     * Node의 Constructor
     * 입력받은 값을 this.value의 값으로 초기화
     */
    private Node(BinaryNode value) {
        this.value = value;
    }

    /*
     * Queue의 마지막에 값을 넣는 함수
     * 받은 값을 값으로 갖는 새 Node를 생성하여
     * prev Node의 next로 설정한다.
     * 그리고 새 Node의 prev Node는 현재 Node의 prev Node로 하고
     * 새 Node의 next Node는 현재 Node로 한다.
     * 그리고 현재 Node의 prev Node는 새로운 Node로 한다.
     */
    public void addLast(BinaryNode value) {
        Node newNode = new Node(value);

        this.prev.next = newNode;
        newNode.prev = this.prev;
        newNode.next = this;
        this.prev = newNode;
    }

    /*
     * 첫번째 Node를 삭제하고 반환하는 메소드이다.
     * 현재 Node의 next Node를 삭제할 Node의 next Node로 바꾸고
     * next Node의 prev Node를 현재 Node로 설정하고 삭제한 Node를 반환한다.
     */
    public Node removeFirst() {
        Node node = this.next;
        this.next = node.next;
        this.next.prev = this;
        return node;
    }

    /*
     * next의 getter
     */
    public Node getNext() {
        return this.next;
    }

    /*
     * value의 getter
     */
    public BinaryNode getValue() {
        return this.value;
    }
}

```

Node.java

```

package Queue;

import tree.BinaryNode;

public class LinkedQueue {
    private Node head;

    /*
     * LinkedQueue의 Constructor
     * 값이 없는 Node를 head로 지정한다.
     */
    public LinkedQueue() {
        this.head = new Node();
    }

    /*
     * 해당 Queue가 비었는지 확인하는 메소드
     */
    public boolean isEmpty() {
        return head.getNext() == head;
    }

    /*
     * Queue에 새로운 값을 넣는 메소드
     * 받아온 값을 head의 addLast를 호출하여 넘긴다.
     */
    public void add(BinaryNode binaryNode) {
        head.addLast(binaryNode);
    }

    /*
     * 해당 Queue에서 값을 하나 제거하고 반환하는 메소드
     * Queue가 비었으면 null을 반환하고
     * 그렇지 않으면 first를 없애며 값을 반환한다.
     */
    public BinaryNode remove() {
        if (isEmpty()) {
            return null;
        }
        return head.removeFirst().getValue();
    }
}

```

LinkedQueue.java

```

import tree.BinaryNode;
import tree.BinaryTree;

import java.util.Scanner;

public class MainClass_08_201502273 {
    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.println("=== Start Tree Practice ===");
        System.out.print("> Insert String : ");
        String string = scanner.nextLine();
        System.out.println();
        System.out.print("> Select Sorting Method.\n" + "Default is inorder.\n"+
            "1: inorder 2:pre order, 3: post order 4: level order :");

        int order = scanner.nextInt();
        System.out.println();
        BinaryTree binaryTree = makeTree(string, order);

        while (true) {
            System.out.println("1: Input String 2: Change Order 3: Check String");
            System.out.print("4: View Tree 9: Exit : ");

```

```

        int num = scanner.nextInt();
        scanner.nextLine();
        System.out.println();

        switch (num) {
            case 1:
                System.out.print("> Input String. : ");
                string = scanner.nextLine();
                binaryTree = makeTree(string, order);
                System.out.println();
                break;
            case 2:
                System.out.print("> Select Sorting Method.\n" + "Default is
inorder.\n"+
                                "1: inorder 2:pre order, 3: post order 4: level order :");
                order = scanner.nextInt();
                binaryTree = makeTree(string, order);
                System.out.println();
                break;
            case 3:
                System.out.print("Check String. : ");
                System.out.println(binaryTree.getValue());
                System.out.println();
                break;
            case 4:
                System.out.println("View Tree Shape. : ");
                binaryTree.print();
                System.out.println();
                break;
            case 9:
                System.out.println("=== End. ===");
                return ;
        }
    }
}

private static BinaryTree makeTree(String string, int order) {
    switch (order) {
        case 2:
            return new BinaryTree(string, "preOrder");
        case 3:
            return new BinaryTree(string, "postOrder");
        case 4:
            return new BinaryTree(string, "levelOrder");
        default:
            return new BinaryTree(string, "inorder");
    }
}
}

```

MainClass_08_201502273.java

2.결과

```
=== Start Tree Practice ===
> Insert String. : abcdefghijklmnopqrstuvwxyz

> Select Sorting Method.
Default is inorder.
1: inorder 2:pre order, 3: post order 4: level order :1

1: Input String 2: Change Order 3: Check String
4: View Tree 9: Exit : 3

Check String. : abcdefghijklmnopqrstuvwxyz

1: Input String 2: Change Order 3: Check String
4: View Tree 9: Exit : 4

View Tree Shape. :
              n
            g
      d      k      u      x
    b    f    i    m    p    r    t    w
  a  c  e  h  j  l  o  q  s  v  y  z

1: Input String 2: Change Order 3: Check String
4: View Tree 9: Exit : 2

> Select Sorting Method.
Default is inorder.
1: inorder 2:pre order, 3: post order 4: level order :2

1: Input String 2: Change Order 3: Check String
4: View Tree 9: Exit : 3

Check String. : abcdefghijklmnopqrstuvwxyz

1: Input String 2: Change Order 3: Check String
4: View Tree 9: Exit : 4

View Tree Shape. :
              a
            b
      c      i      o      v
    d    g    j    m    p    t    w
  e  f  h  k  l  n  r  q  s  u  x  z  y
```

```

1: Input String 2: Change Order 3: Check String
4: View Tree 9: Exit : 2

> Select Sorting Method.
Default is inorder.
1: inorder 2:pre order, 3: post order 4: level order :3

1: Input String 2: Change Order 3: Check String
4: View Tree 9: Exit : 3

Check String. : abcdefghijklmnopqrstuvwxyz

1: Input String 2: Change Order 3: Check String
4: View Tree 9: Exit : 4

View Tree Shape. :
          z
        m
      f
    c   e   l   s   y
  a b d g h j k n o q r t u v w

1: Input String 2: Change Order 3: Check String
4: View Tree 9: Exit : 2

> Select Sorting Method.
Default is inorder.
1: inorder 2:pre order, 3: post order 4: level order :4

1: Input String 2: Change Order 3: Check String
4: View Tree 9: Exit : 3

Check String. : abcdefghijklmnopqrstuvwxyz

1: Input String 2: Change Order 3: Check String
4: View Tree 9: Exit : 4

View Tree Shape. :
          a
        b
      d   e   f   c
    h   i   j   k   l   m   n   g   o
  p q r s t u v w x y z

1: Input String 2: Change Order 3: Check String
4: View Tree 9: Exit : 9

=== End. ===

```

“abcdefghijklmnopqrstuvwxyz”를 문자열로 입력을 해주고 각각 Inorder, Preorder, Postorder, Levelorder로 Tree에 저장을 시키고 각각의 Tree를 각각의 검색방법으로 돌며 출력을 한 결과 처음에 입력했던 “abcdefghijklmnopqrstuvwxyz”가 그대로 나오는 것을 볼 수 있었고 해당 Tree들을 Levelorder방식으로 터미널에 Tree의 포맷으로 출력한 결과를 본 결과 각각 방식에 맞게 깊이가 최소가 되도록 들어가 있는 것을 볼 수 있다.

깨달은 점 및 결론

Preorder로 데이터를 저장할 때 첫 번째 데이터를 root로 하고 두 번째 요소에서 중간에 있는 요소까지를 왼쪽 자식으로 넘기고 중간부터 끝까지 있는 요소를 오른쪽 자식으로 넘기는 방식으로 구현을 해봤는데 preorder방식으로 검색을 하며 출력을 하는 것은 잘 되었지만 Tree의 모습으로 출력을 하는 경우 한쪽으로 쏠리며 모양이 깨지는 현상이 있었는데 두 번째 요소에서 중간까지가 아니라 중간 + 1까지를 왼쪽 자식에게 넘기고 중간 + 1부터 끝까지를 오른쪽 자식에게 넘기는 방식으로 구현을 하니 Tree모습으로 출력하는 것 또한 잘 출력되었다. 기존 코드와 맞지않는 부분이 있어서 그렇게 출력되었던 것으로 생각된다. String을 substring을 사용해 잘라 사용하는 방식으로 구현을 하다 보니 구현 중에 OutOfBoundException이 자주 발생했다. Leaf에 다다라서 입력 String의 길이가 1 혹은 0까지 내려가다 보니 난 오류였으며 디버깅을 통해 오류를 제거하였다.