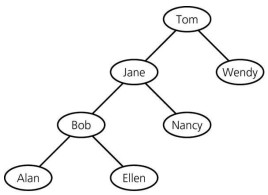
자료구조 중간고사

4/27, 2016. Open book, 75분, 100점 만점

1. (점수주기 문제. 20점) 아래 binary search tree에 "Donald"를 insert하려 한다. 이를 위해 아래 알고리즘 insert( )를 수행할 때
   (a) setLeft( )와 setRight( )가 호출되는 횟수는 각각 몇회인가?
   (b) setLeft( )나 setRight( )가 원래의 값을 변하게 하는 경우는 총 몇회인가?




```
insert(Comparable newItem) {
                root = insertItem(root, newItem);
}

TreeNode insertItem(TreeNode tNode, Comparable newItem) {
        if (tNode == null) {
                tNode = new TreeNode(newItem, null, null);
        } else if (newItem < tNode's item) {
                tNode.setLeft( insertItem(tNode.getLeft( ), newItem) );
        } else {
                tNode.setRight( insertItem(tNode.getRight( ), newItem) );
        }
        return tNode;
} // end insertItem
```

2. (15점) 아래는 수업 시간에 배운 ListReferenceBased를 이용한 Stack의 구현이다.
   (a) list의 첫 번째 item을 stack top으로 간주하지 않고, list의 마지막 item을 stack top으로 간주하도록 이 코-드를 바꾸어보라. 다 쓸 필요없고 바뀌는 부분만 쓰면 된다.
   (b) 이렇게 바꾸면 작업의 효율성은 원래의 코-드에 비해 어떤가? "더좋다," "동일하다," "더나쁘다" 중에 선택하고 더 좋거나 더 나쁘면 이유도 간단히 설명하라.

```
public class StackListBased implements StackInterface {
        private ListInterface list;

        public StackListBased( ) {
                list = new ListReferenceBased( );
        }

        public boolean isEmpty( ) {
                return list.isEmpty( );
        }
        public void push(Object newItem) {
                list.add(1, newItem);
        }
        public Object pop( ) {
                if (!list.isEmpty( )) {
                        Object temp = list.get(1);
                        list.remove(1);
                        return temp;
```

```
                } else {exception 처리; }
        }
        public void popAll ( ) {
                list.removeAll( );
        }
        public Object peek( ) {
                if (!isEmpty( )) return list.get(1);
                else { exception 처리; }
        }
} // end class StackListBased
```

3. (15점) 아래는 dummy head node가 있는 linked list에서 한 원소를 삭제하는 Java code다. List의 맨 마지막 노드의 next는 **null** 값을 갖는 non-circular list이다. 아래 method remove(index)는 linked list에서 index번째 원소를 삭제하는 것이다. 이제 여기에 변수를 하나 더 주어 remove(index, k)와 같이 부르도록 하고, index번째부터 연속된 k개의 원소(index번째 원소 포함하여 k개)를 삭제하도록 하고자 한다. remove(index, 1)을 부르면 원래의 remove(index)과 같은 일을 하게 될 것이다. 만일 index번째 노드부터 시작해서 남은 노드가 k개가 안되면 지울 수 있는 최대한도인 마지막 노드까지만 지운다. 아래 코-드를 이에 맞게 변형해 보라.

```
public class ListReferenceBased implements ListInterface {
        private Node head;
        private int numItems;
        ...
        public void remove(int index) {
                if (index >= 1 && index <= numItems) {
                        Node prev = find(index - 1);
                        Node curr = prev.getNext( );
                        prev.setNext(curr.getNext( ));
                        numItems--;
                } else { exception 처리; }
        }

}

* 아래 ??? 부분을 채워 넣을 것
public class ListReferenceBased implements ListInterface {
        private Node head;
        private int numItems;

        public void remove(int index, int k) {
                if (index >= 1 && index <= numItems) {

                        ???

                } else {Exception handling; }
        }

}
```

4. (20점) 아래는 d-자리 정수로 구성된 A[1...n]을 Radix sorting 하는 sample algorithm이다. 이를 recursive version으로 바꾸어 보아라.

```
radixSort(A[ ], n, d)
// Sort n d-digit integers in the array A[1...n]
// 1st digit: most significant digit
{
```

```
        for (j = d downto 1) {
                Do a stable sort on A[1..n] by j^th digit;
        }
}
```

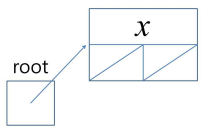
5. (15점) 아래는 수업 시간에 배운 mergesort 알고리즘에 counter 하나를 삽입한 것이다. Array A[1...2^k]로 mergesort가 수행되는 과정에서 가질 수 있는 cnt의 최대값은 얼마인가?

```
// 아래 키워드 sequence는 의미상으로만 이해하고 받아들이면 됨
// 최초의 mergeSort( )가 호출될 때 cnt 값은 0으로 시작함

int mergeSort(sequence S, int cnt)
// Input: sequence S
// Output: sorted sequence S
{
        cnt++;
        if (S.size( ) > 1) {
                Let S1, S2 be the 1st half and 2nd half of S, respectively;
                cnt <- mergeSort(S1, cnt);
                cnt <- mergeSort(S2, cnt);
                S <- merge(S1, S2);
        }
        cnt--;
        return cnt;
}

sequence merge(sequence S1, sequence S2)
{       sorting된 두 sequence S1, S2 를 합쳐
        sorting 된 하나의 sequence S를 return한다
}
```

6. (15점) 아래는 binary search tree에서 search key x를 가진 노드를 삭제하기 위해 수업 시간에 배운 알고리즘이다. 이 알고리즘만으로는 tree가 아래 그림과 같이 단 한 개의 노드만으로 이루어져 있을 때는 잠재적인 문제가 있다. 어떤 문제인가? 그리고 이 문제를 해결할 수 있도록 보완해보라. 변수 root는 root node를 가리키는 reference variable이다. deleteItem( )내에서 해결하거나 추가로 함수를 하나 더 만들거나 상관없다.




```
TreeNode deleteItem (TreeNode tNode, Comparable x) {

        if (tNode == null) { exception 처리 }; // item not found!
        else {
                if (x == tNode's key) { // item found!
                        tNode = deleteNode(tNode);
                } else if (x < tNode's key) {
                        tNode.setLeft(deleteItem(tNode.getLeft( ), x));
                } else {
                        tNode.setRight(deleteItem(tNode.getRight( ), x));
                }
        }
        return tNode;
}
```