자료구조 중간고사, 2014.4.30.

Open book, 60분

* 그런 일이 없다고 생각하지만 문제에 오타나 실수가 있다고 생각하면 출제자의 의도를 생각하여 고쳐서 풀도록 하라. 자신이 고친 이유를 반드시 명시하도록 하라. 합리적인 수정은 가산점 부여함.

1. (20점. 이해도 테스트) 맞으면 O, 틀리면 X 표시하고, 틀린 경우에는 이유를 밝히라. $O(\ )$ notation 문제의 경우 $O(\ )$ notation의 정확한 정의를 생각하면서 풀 것.
   1.1 quicksort의 asymptotic running time은 $O(n \log n)$이다.
   1.2 mergesort의 최악의 경우 running time은 $O(n \log n)$이다.
   1.3 radix sort의 running time은 $O(n \log n)$이다.
   1.4 radix sort에서 least significant digit부터 sorting 하는 대신 most significant digit부터 sorting해도 제대로 sorting이 된다.

2. (10점. 점수주기) 비어있지 않은 queue q에 아래 코드를 수행하면 queue의 내용은 어떻게 되는가?

```
s = new StackReferenceBased( );
while (!q.isEmpty( ))
        s.push(q.dequeue( ));
while (!s.isEmpty( ))
        q.enqueue(s.pop( ));
```

3. (15점. 이해도 테스트) 아래는 array A[first...last]를 quicksort 하는 서두부의 partition 알고리즘이다. 이 partition 알고리즘은 A[first]를 pivot으로 삼고 있다. 이것을 A[last]를 pivot으로 삼는 알고리즘으로 바꾸어 보아라. 아래 알고리즘을 제대로 이해하는 지를 같이 묻는 문제이므로, 최대한 이 알고리즘의 틀에서 벗어나지 않아야 한다. 이 틀을 벗어나 다른 방식의 partition을 시도하면 이 알고리즘을 이해 못하고 다른 참고자료에서 복사한 것으로 간주하여 점수를 못받음.

```
int partition(A[ ], first, last)
// partition array A[first...last]
{
        pivot = A[first];
        lastS1 = first;

        for (firstUnknown = first+1; firstUnknown <= last; ++firstUnknown) {
                if (A[firstUnknown] < pivot) {
                        ++lastS1;
                        A[firstUnknown] ↔ A[lastS1]; // swap
                } //end if
        } // end for
        A[first] ↔ A[lastS1]; // swap
        return lastS1;
} //end partition
```

4. (15점) Queue에 원소를 더할 때는 back 뒤에만 삽입할 수 있고, 삭제할 때는 front에 있는 원소만 삭제할 수 있다. 이 제한을 풀어 front와 back 쪽 모두에서 삽입과 삭제가 가능하도록 한 것을 double-ended queue(또는 줄여서 dequeue)라고 한다. 아래 queue 구현의 틀을 유지하면서 double-ended queue를 위한 두 개의 method insertFront( ), removeBack( )를 구현해 보아라.

```
public class QueueArrayBased implements QueueInterface {
        final int MAX_QUEUE = 50;
        private Object items[ ];
        private int front, back, numItems;

        public QueueArrayBased( ) {
                items = new Object[MAX_QUEUE];
                front = 0;
                back = MAX_QUEUE - 1;
                numItems = 0;
        }

        public void enqueue(Object newItem) {
                if (!isFull( )) {
                        back = (back+1) % MAX_QUEUE;
                        items[back] = newItem;
                        ++numItems;
                } else {exception 처리;}
```

```
        } // enqueue
        public Object dequeue( ) {
                if (!isEmpty( )) {
                        Object queueFront = items[front];
                        front = (front+1) % MAX_QUEUE;
                        --numItems;
                        return queueFront;
                } else {exception 처리;}
        } // dequeue
} //QueueArrayBased
```

5. (20점) 아래는 binary tree에서 삭제할 노드(tNode)가 정해진 상태에서 해당 노드를 삭제하는 알고리즘이다. 여기서 children이 2개 있을 경우 오른쪽 subtree를 손대지 않고 대신 왼쪽 subtree를 손대는 알고리즘으로 바꾸어 보아라. 아래 알고리즘을 제대로 이해하는 지를 같이 묻는 문제이므로, 알고리즘 전체를 쓰되 아래에 주어진 알고리즘을 기반으로 해야 하고, 아래 알고리즘과 달라진 부분은 밑줄을 그어 채점자가 쉽게 볼 수 있게 하라. (밑줄 없으면 감점됨. 아래 알고리즘의 틀을 벗어나면 점수 없음.)

```
TreeNode deleteNode (TreeNode tNode) {

        if ( (tNode.getLeft( ) == null)  && (tNode.getRight( ) == null)) {
                        return null
        } else if (tNode.getLeft( ) == null ) {
                        return tNode.getRight( );
        } else if (tNode.getRight( ) == null) {
                        return tNode.getLeft( );
        } else {
                        tNode.setItem(minimum item of tNode's right subtree);
                        tNode.setRight(deleteMin(tNode.getRight( ));
                        return tNode;
        } //else
} //deleteNode

TreeNode deleteMin (TreeNode tNode) {
        if (tNode.getLeft( ) == null) {
                return tNode.getRight( );
        } else {
                tNode.setLeft(deleteMin(tNode.getLeft( ));
                return tNode
        } // else
} // deleteMin
```

6. (20점. 거의 기출 문제) 아래는 Hanoi Tower 알고리즘이다.

```
move(n, A, B, C)
{
        if (n=1) then move the disk from A to B
        else {
                move(n-1, A, C, B);
                move(1, A, B, C);
                move(n-1, C, B, A);
        }
}
```

이제 규칙을 조금 확장해보려 한다. 한 번에 원반 1개씩을 옮기는 대신 한 번에 3개까지 옮길 수 있다고 하자. 즉, 한 번에 1개, 2개, 또는 3개를 옮길 수 있다. 이런 것은 한번의 이동으로 간주한다. 단, 큰 원반이 작은 원반 위에 놓이는 경우가 없다는 조건은 여전히 지켜야 한다.

6.1 (10점) 한번에 3개까지 옮길 수 있는 버전으로 위 알고리즘을 변형하라. 아래 "???" 부분을 채우라.

```
move(n, A, B, C)
{
        ???
}
```

6.2 (10점) 이렇게 하면, n개의 원반을 다 옮기려면 총 몇 번의 이동이 필요한가? 당신의 답을 제시하고 이를 증명하라. 당신의 답은 모든 자연수 n에 대해 성립해야 한다.