1. Implement a disjoint set forest, with union-by-rank and path compression, following section 21.3 in your textbook. To get you started, here is a skeleton of the header file:

```cpp
template <class T>
class Node
{
    // you write this
};

template <class T>
class DisjointSets
{
public:
    DisjointSets(); // construct a disjoint set forest with default capacity
    DisjointSets(int size); // construct a disjoint set forest with given capacity
    void MakeSet(T* x); // make a new singleton set containing data element x
    void Union(T* x, T* y); // unite the disjoint sets containing data elements x and y
    T* FindSet(T* x); // return the representative of the set containing element x
    void Print(); // print the contents of the disjoint set forest

private:
    void Link(Node<T>* x, Node<T>* y); // other private methods here to connect data items to nodes
    Node<T> **elements; // array of nodes in the forest
    int capacity; // size of elements array
    int length; // number of elements in the forest
};

class Full { }; // full exception
class NotFound { }; // element not found exception
```

The pseudocode in your textbook assumes that callers have access to pointers to the nodes in the forest. However, we will want to hide those pointers in our implementation in order to separate the ADT from its implementation. Therefore, like your previous classes, methods will take and return pointers to data items instead of nodes. This will require that we maintain an array of nodes in our implementation that we can search through in order to find the node corresponding to a data pointer. (How is this going to affect running time?) It will also require that we utilize some private methods to perform this search and implement the algorithms in your textbook.
The `Print()` method should print the contents of the disjoint set forest by iterating over the array of nodes and printing ranks and parent relationships. For example, if the disjoint set forest looked like the following

```
4
0 1
6
3
7
2 5
```

then your `Print()` method should display

```
0:0 -> 4:2
1:1 -> 4:2
2:0
3:1
4:2
5:0
6:0 -> 1:1 -> 4:2
7:0 -> 3:1
```

where each `i:j` represents a data element `i` and its rank `j`. As in previous assignments, we assume that type `T` has the stream insertion operator defined for it. (In this example, I am assuming that the key value for each data element happens to be the index at which the node for that element is stored in the array. Otherwise, the elements won’t necessary print in this order.)

Write a unit test for each of your methods *as you write it*, include pre- and post-conditions for each method, and throw exceptions where appropriate.

2. For this particular implementation, what is the total worst case running time for a sequence of `n` `MakeSet()`’s and `n − 1` `Union()`’s? Explain your answer.