ECE-250 – Algorithms and Data Structures (Winter 2012)

Assignment 3

Due on Friday, February 10, in class.

All questions will have equal weight for the assignment grade (except for the bonus marks questions).

1 - Provide the definition of a function `swap_nodes` that swaps two elements in a doubly-linked list by readjusting links only (that is, the actual values stored in the two nodes are not touched or in any way moved). You are allowed to assume a circular list with dummy element implementation (or in any case, you are allowed to assume that neither of the two nodes is the first or last element in the list). You may also assume that the two nodes are not consecutive elements in the list, or the same element. Assume also that the function is declared `friend`, so it has access to the data member of class Node. The function declaration is:

```
template <typename Type>
void swap_nodes (Node<Type> * node1, Node<Type> * node2);
```

2 - Given a hash table of size 16, with hash function $\mathrm{h}(x) = x \bmod 16$, we want to insert prime numbers in sequence starting at 11 (i.e., 11, 13, 17, 19, 23, 29, $\cdots$) until two collisions occur — that is, include the number that causes the second collision.[1]

Show the above procedure (insertion by insertion, showing the contents of the array after each insertion) with collisions handled by:

(a) Linear probing.
(b) Double hashing, with the hash function for the jump size being $\mathrm{h_J}(x) = (x \bmod 10)$ OR 1 (that is, we take the number modulo 10, and if it is even, we add 1, so that we can only obtain values 1, 3, 5, 7, or 9).

3 - Based on the code for class template `Node<Type>` from the course slides for 2012-02-03 (Tree implementations and traversal), sketch a C++ function (a *standalone function*, as opposed to a method of class `Node`) that receives two nodes (as in, two pointers to `Node<Type>` objects) and returns the nearest common ancestor (i.e., the node with largest depth that is an ancestor of both nodes). Notice that in this case, the function is *not* a friend function of class Node. The function declaration should be:

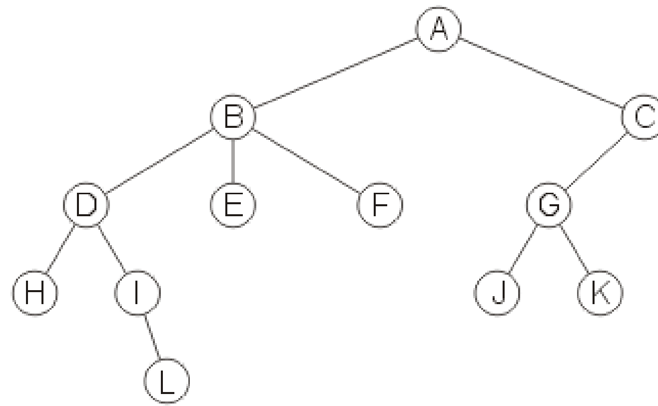
```
template <typename Type>
Node<Type> * nearest_common_ancestor (const Node<Type> * n1, const Node<Type> * n2);
```

The function must run in $O(n)$, where n is the number of nodes in the tree, and must use $O(1)$ storage (i.e., you're not allowed to use a list or in any case a sequential container to store any sequence of tested or matching nodes, or even use recursion — remember that recursion uses the stack of the processor, and so it uses memory; in a way that does not show explicitly in the program, but still — if your function makes $\boldsymbol{\omega}(1)$ recursive calls, then it uses $\boldsymbol{\omega}(1)$ storage, which is disallowed).

Hint: Start by determining the depth of each of the two nodes.

[1] To avoid unnecessary mistakes/oversights, you can check the list of prime numbers on Wikipedia, `http://en.wikipedia.org/wiki/List_of_prime_numbers`

4 - Given the tree shown below:



(a) Identify all the leaf nodes and all the internal nodes.
(b) List the nodes with depth 0, 1, 2, and 3.
(c) What is the height of the tree? What is the height of the subtrees with root B and C?
(d) For the nodes B, C, and D: list the parent, the children, the siblings, all the ancestors, and all the descendants.
(e) What is the longest path, and what is its length? (explain why) If there is more than one, list all of them (explaining why).

In all cases, you must explain/justify your answer (any answer without a justification will be considered invalid)

10% Bonus Marks:

We want to implement an iterator class for a singly linked list with dummy or sentinel element. The linked list and node classes are declared as follows,[2] and the default constructor for List is shown below:

```
template <typename Type> class Node;

template <typename Type>
class List
{
public:
    class Iterator
    {
    public:
        Iterator (Node<Type> * node, const Node<Type> * sentinel)
            : d_node(node), d_sentinel(sentinel)
        {}

        void advance();
        Type retrieve() const;
        bool at_end() const;

    private:
        Node<Type> * d_node;
        Node<Type> * d_sentinel;
    };

    List()
        : d_head (new Node<Type>)
    {
        d_head->d_next = d_head;
    }

    Iterator begin() const;
    void insert (const Type & value, Iterator at);

private:
    Node<Type> * d_head;
};

template <typename Type>
class Node
{
public:
    Node (const Type & value = Type());

    Node<Type> * next() const;
    Type retrieve() const;

private:
    Type d_value;
    Node<Type> * d_next;
    friend class List<Type>;
};
```

Given this information (the "documentation" implied by the code sample), provide definitions for the three Iterator methods (`advance`, to move to the next element in the list; `retrieve` to get the value of the element "pointed at" by the iterator; and `at_end`, to check if we are outside the sequence of elements in the list).

Also, provide definitions for the two methods in class List: `begin`, to return an iterator pointing at the first element in the list; and `insert`, to insert a new element after the node pointed at by the

[2] The first line is a *forward declaration*, to break the mutual (circular) dependency — List needs to know about Node, and Node needs to know about List

given iterator. If the list is empty, `List::begin()` should return an iterator for which the method `at_end()` returns true.

A typical loop going through the elements in the list could look like this:

```
List<int> values;
// ...
for (List<int>::Iterator i = values.begin(); !i.at_end(); i.advance())
{
    // ...
}
```