ECE-250 – Algorithms and Data Structures (Winter 2012)
Assignment 3 – Suggested solutions

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);
```

Solution:

It suffices in principle to just swap the appropriate "next" and "previous" pointers for the two nodes, their previous and their next; however, one detail that we have to be careful about is the fact that when we reassign the next or previous pointers for the two nodes, then the nodes in the sequence (that require reassigning to point to the other node) are no longer the same.

One simple solution could be to just save the previous and next for both nodes (that's four local vaiables; prev1, next1, prev2, next2, corresponding the node1 and node2) before reassigning anything. This is probably the most "economical" and safe way to implement it (and it runs in $\Theta(1)$):

```
template <typename Type>
void swap_nodes (Node<Type> * node1, Node<Type> * node2)
{
    Node<Type> * prev1 = node1->previous();
    Node<Type> * next1 = node1->next();
    Node<Type> * prev2 = node2->previous();
    Node<Type> * next2 = node2->next();

    prev1->d_next = node2;
    next1->d_prev = node2;
    prev2->d_next = node1;
    next2->d_prev = node1;

    node1->d_next = next2;
    node1->d_prev = prev2;
    node2->d_next = next1;
    node2->d_prev = prev1;
}
```

2 - Given a hash table of size 16, with hash function $h(x) = x \bmod 16$, 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.

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 $h_J(x) = (x \bmod 10)$ 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).

Solution:

(a) For each value inserted we compute its hash, which determines the bin where they're placed— 11 and 13 hash to the same values, so they go in bins 11 and 13, respectively; 17 goes in bin 1 (hash(17) = 17 mod 16 is 1), 19 in bin 3, 23 in bin 7, then 29 collides (hash(29) = 29 mod 16 is 13, and bin 13 is already taken), so we probe the following bins, to find bin 14 available; so 29 goes in bin 14. Then, 31 goes in bin 15, 37 in bin 5, 41 in bin 9, and 43 causes the second collision at bin 11; so, 43 is placed in bin 12.

(b) Similar procedure, but for collisions, we compute the jump size for the probing, with the secondary hash function:

Thus, 11, 13, 17, 19 and 23 go in the exact same positions (no collisions so far). 29 collides at bin 13, and the jump size is 9. So, we probe bin 6 (13 + 9 mod 16 = 6), which is available; so, 29 goes in bin 6. 31 goes in bin 15, then 37 in bin 5, 41 in bin 9, and 43 causes the second collision, at bin 11. The jump size is 3, so we probe bin 11+3 = 14, which is available; so, 43 goes in bin 14.

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.

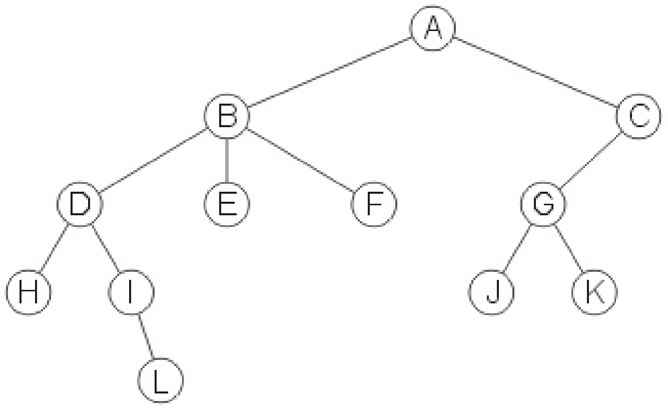
Solution:

The key idea is that if both nodes are at the same depth, then we advance (advance meaning move one level up; i.e., to the parent) for both simultaneously, and both will hit the nearest common ancestor at some point (specifically, both at the same iteration).

So, we first determine the depths, and make the deeper one advance until getting to the same depth as the other one; then proceed as per the above description.

```
template <typename Type>
Node<Type> * nearest_common_ancestor (const Node<Type> * n1, const Node<Type> * n2)
{
    int depth1 = 0;
    const Node<Type> * n = n1;
    while (!n->is_root())
    {
        ++depth1;
        n = n->parent();
    }

    n = n2;
    while (!n->is_root())
    {
        ++depth2;
        n = n->parent();
    }

    for (int i = 0; i < std::abs(depth1-depth2); i++)
    {
        if (depth1 > depth2)
        {
            n1 = n1->parent();
        }
        else if (depth2 > depth1)
        {
            n2 = n2->parent();
        }
    }

    while (n1 != n2)
    {
        n1 = n1->parent();
        n2 = n2->parent();
    }

    return n1;
}
```

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).

Solution:

(a) Internal nodes are the ones that have some children (one or more); leaf nodes are those that have no children — A,B,C,D,G,I; and E,F,H,J,K,L, respectively.

(b) Depth 0 is the root (always the case, by definition). Depth 1 are the root's children, B, and C; depth 2 are D, E, F, and G; and depth 3 are H, I, J, K.

(c) The deepest node is L, with depth 4 (path from root is A,B,D,I,L — four "links", telling us that its depth is 4), and thus the height of the tree is 4. Since this height is reached through B, the height of the subtree with B as root has to be one less than the previous one (and by visual inspection we confirm that it is 3 — path being B,D,I,L). For the subtree below C, the height is 2.

(d) B: parent is A, children are D,E,F, sibling is C (since it's the only one with parent A). Ancestors include itself (as do descendants!); so ancestors of B and A,B; descendants are B,D,E,F,H,I,L. Notice that ancestors are basically the nodes in the path from root to the node, and descendants are all the nodes in the subtree with root being the given node. I won't list the answers for C, and D — same details as here.

(e) We recall that a path is a sequence of nodes where each pair of consecutive nodes have the relationship parent/child — so, we're not talking about the "path" going from L to J or K (passing through A and continuing down to the C branch).

With this in mind, the longest path is the one defining the height of the tree, which is A,B,D,I,L.

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,[1] 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

[1] 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.

Solution:

Couple of key details: the sentinel element defines the end of the list; when we reach it, we're outside the range, and so at_end() should return true then.

begin() should return an iterator pointing at the *first element* in the list — this of course excludes the sentinel element; so, we should point to the sentinel's next; with the nice side-effect that if the list is empty, sentinel's next is itself, and so we return an iterator for which at_end() will return true!

```
template <typename Type>
void List<Type>::Iterator::advance()
{
    d_node = d_node->next();
}

template <typename Type>
Type List<Type>::Iterator::retrieve() const
{
    return d_node->retrieve();
}

template <typename Type>
bool List<Type>::Iterator::at_end() const
{
    return d_node == d_sentinel;
}

// List methods
template <typename Type>
List<Type>::Iterator List<Type>::begin() const
{
    return Iterator (d_head->next(), d_head);
        // d_head points to the dummy/sentinel node
}

template <typename Type>
void List<Type>::insert (const Type & value, List<Type>::Iterator at)
{
    Node<Type> * added = new Node<Type>(value);
    added->d_next = at.d_node->next();
    at.d_node->d_next = added;
}
```