CSS 343 – Data Structures, Algorithms, and Discrete Mathematics II
Professor Clark F. Olson
Lecture Notes 0 – Introduction/Review

This is an informal compilation of the topics that we discussed in the first lecture.

Preconditions and postconditions

The use of preconditions and postconditions is one standard way to define the interface for functions and methods. The preconditions say what must be true (assumptions) when the function is called (with respect to instance variables, parameters, files, and any other **run-time** information available to the function). The postconditions say what side effects the function has that can be observed outside of the function (output, changes to variables and files, etc.) Note that they do not say the process by which the postconditions are met. These specify a starting point and an end point, not a process for getting from one to the other.

Here is an example of documentation using preconditions and postconditions for a sorting method. See also the array class example code on the course assignments page.

```
// sort:  function to sort an array of integers
// preconditions:  n is the number of values to sort and is no less than zero.
//                 array has been allocated and holds at least n integers.
// postconditions: the first n values of array are sorted into non-decreasing order.
//                 specifically array[i] <= array[i+1] for 0 <= i < n-1

template <typename Comparable>
void sort(Comparable array[], int n) { … }
```

Linked lists review

Let's review linked lists with the following simple node struct:

```
struct Node {
        int data;
        Node *next;
        };
```

First, let's write a function to insert an element into a sorted linked list (could be empty or not).

```
        // Insert node in sorted order.
        // Precondition: list must be sorted into non-descending order.
        // Postcondition: data is inserted into list in non-descending order.
        void insert(Node *&head, int data) {
                // note the importance of passing the head pointer by reference

                // create a node
                Node *nextNode = new Node;
                nextNode->data = data;
                nextNode->next = NULL;          // not actually necessary

                // handle case where the head must be changed.
                if (head == NULL || nextNode->data <= head->data) {
                        nextNode->next = head;
                        head = nextNode;
                }
```

```
        // handle general case where new node is inserted in middle (or end) of list
        else {
            Node *cur = head;
            while (cur->next != NULL && cur->next->data < nextNode->data) {
                cur = cur->next;
            }

            nextNode->next = cur->next;
            cur->next = nextNode;
        }
    }
```

Can we do this recursively? Of course!

```
    // Insert node in sorted order.
    // Precondition: list must be sorted into non-descending order.
    // Postcondition: data is inserted into list in non-descending order.
    void insert(Node *&head, int data) {

        if (head == NULL || data <= head->data) {
            Node *nextNode = new Node;
            nextNode->data = data;
            nextNode->next = head;
            head = nextNode;
        }
        else insert(head->next, data);
    }
```

Here is a copy constructor for a class containing a linked list of Nodes (this assumes that the head pointer is the only data member). *Note: Not covered in lecture for Winter 2015.*

```
LinkedList::LinkedList(const LinkedList &rhs) {

    // handle special case
    if (rhs.head == NULL) {
        head = NULL;
        return;
    }

    // copy head
    Node *orig = rhs.head;
    head = new Node;
    // must also copy over data from head here (undefined in this case)

    // copy rest of list
    Node *cur = head;
    orig = orig->next;
    while (orig != NULL) {
        cur->next = new Node;
        cur = cur->next;
        // must also copy over data from orig here (undefined in this case)
        orig = orig->next;
    }
    cur->next = NULL;
}
```

A recursive solution would use a helper method, such as copyList:

```
Node * LinkedList::copyList(Node *head) const {
        if (head == NULL) return NULL;

        Node *cur = new Node;
        // must also copy over data from head here (undefined in this case)
        cur-> next = copyList(head->next);
        return cur;
}
```

Next, let's write a function to remove all nodes from a linked list that hold a particular integer. For this problem, we will not assume that the list is sorted.

```
        void removeAll(Node *&head, int toRemove) {
                if (head == NULL) return;          // unnecessary

                // remove head (maybe more than once!)
                while (head != NULL && head->data == toRemove) {
                        Node *tmp = head;
                        head = head -> next;
                        delete tmp;
                }

                if (head == NULL) return;

                // remove nodes after head
                for (Node *cur = head; cur->next != NULL; cur = cur->next) {

                        // being careful to loop here, so that we don't skip over any!
                        while (cur->next != NULL && cur->next->data == toRemove) {
                                Node *tmp = cur->next;
                                cur->next = tmp->next;
                                delete tmp;
                        }
                }
        }
```

OR:

```
        void removeAll(Node *&head, int toRemove)
        {
                if (head == NULL) return;

                removeAll(head->next, toRemove);

                if (head->data == toRemove) {
                        Node *tmp = head;
                        head = head->next;
                        delete tmp;
                }
        }
```

What is the running time of the code for a list with *n* elements in it? O(n) for either function. Unfortunately, the recursive version doesn't use tail recursion, so it does use more space than the iterative version (but only by a constant multiplier, since we must use O(n) space to store the list)

Finally, let's take two sorted lists and merge them. *Note: Not covered in lecture for Winter 2015.*

A simple, but inefficient method, would use insert (either one).

```
// Merge two sorted lists.
// Precondition: none (the insert method will work even with non-sorted inputs)
// Postcondition: the lists are merged into a new list in non-decreasing order.
// (Original lists are not changed.)
Node * merge (Node *list1, Node *list2) {
        Node *head;

        while (list1 != NULL) {
                insert(head, list1->data);
                list1 = list1->next;
        }

        while (list2 != NULL) {
                insert(head, list2->data);
                list2 = list2->next;
        }
}
```

What is the running time of this algorithm in Big-O notation if the initial lists both have n different elements in them? O(n^2),which is quite inefficient.

How can we improve this? We walk through the lists one element at a time in sorted order and keep track of the end of the new list, which is where we will always insert.

Solutions:
For both solutions, I use a helper function that determines which list has the smallest next element:

```
// Precondition: at least one of head1 and head2 is not NULL.
bool firstIsSmaller(Node *head1, Node *head2) {
        if (head1 == NULL) return false;
        if (head2 == NULL) return true;
        return (head1->data < head2->data);
}
```

Recursive solution:

```
Node *mergeLists(Node *head1, Node *head2) {
        Node *head3 = NULL;
        // Make sure there is at least one list to merge.
        if (head1 == NULL && head2 == NULL) return head3;

        // Take care of head first.
        head3 = new Node;
        if (firstIsSmaller(head1, head2)) {
                head3->data = head1->data;
                head3->next = mergeLists(head1->next, head2);
        } else {
                head3->data = head2->data;
                head3->next = mergeLists(head1, head2->next);
        }
        return head3;
}
```

Iterative solution:

```
Node *mergeLists(Node *head1, Node *head2) {
        Node *head3 = NULL, *cur = NULL;

        // Make sure there is at least one list to merge.
        if (head1 == NULL && head2 == NULL) return head3;

        // Take care of head first.
        head3 = new Node;
        cur = head3;
        while (head1 != NULL || head2 != NULL) {
                if (firstIsSmaller(head1, head2)) {
                        cur ->data = head1->data;
                        head1 = head1->next;
                }
                else {
                        cur ->data = head2->data;
                        head2 = head2->next;
                }

                if (head1 != NULL || head2 != NULL) {
                        cur->next = new Node;
                        cur = cur->next;
                }
        }
        cur->next = NULL;
        return head3;
}
```

Both of these run in time O(n) for two lists of size n.