**COMP 1917 Computing 1  
Session 2, 2014**

**Tutorial - Week 9**

**Linked Lists**

**This page was last updated: 09/17/2014 09:16:24**

**Presentation Topic for Week 9**

Explain Assignment 2. What parts of the assignment do you think will be most challenging? What are the "pitfalls" to watch out for?

1. Consider the following task, known as the "balanced bracket" problem:

A text file contains three different types of brackets: round brackets ( ) square brackets [ ] and curly brackets { }.  
The task is to read a file one character at a time and determine whether the brackets are "balanced" - in the sense that every opening bracket is matched by an appropriately-placed closing bracket of the same type.

The balanced bracket task can be achieved using a stack in the form of a linked list, as follows:

* + read characters one at a time from standard input
  + whenever you see an opening bracket (, [ or {, push it onto the list
  + whenever you see a closing bracket ), ] or }, check the first item in the list to see if it contains an opening bracket of the same type. If it does, pop that item from the list and continue searching; if the list is empty, or the brackets are not of the same type, print "No, not balanced" and exit.
  + after the entire text has been read, check to see if the list is empty; if it is, print "Yes, is balanced"; otherwise print "No, not balanced".
  + Trace through the above algorithm on the following examples, showing the content of the list at each step:
  + ([{}()]) // balanced
  + ({(()([])]) // unbalanced, because { is matched with ]
  + ((([]{})) // unbalanced, because initial ( is not matched
  + []}() // unbalanced, because } is not matched
  + Write a C program to perform the balanced bracket task using the above algorithm. (Note: you will be implementing this program for today's laboratory exercise; the functions makeNode(), push(), pop() and freeList() defined in lectures will be provided, so you won't need to re-write them; you just need to know what they do).

1. Write a function
2. Lnode \* min\_to\_front( Lnode \* head )

which takes a linked list, finds the minimum element in the list (the one for which data is smallest) and moves that element to the front of the list.

1. Write a function
2. Lnode \* sort\_list( Lnode \* head )

which takes an unsorted linked list, sorts the list items into order of increasing data by repeatedly calling min\_to\_front() and returns the head of the resulting sorted list.

1. (Bonus Challenge) Write a function
2. Lnode \* merge( Lnode \*L, Lnode \*M )

which takes two ordered linked lists L and M and rearranges their nodes to create a single ordered linked list containing all the nodes of the two original lists (in order). The function should return a pointer to the head of the new list. Note: the merge() function can be written either recursively or non-recursively. Try to do it both ways. Which one do you prefer?

**Presentation Topic for Week 11**

Any topic that was missed in earlier weeks, or a repeat of any topic that was previously covered, or any new topic of your own choosing (relevant to the course).