BITS Pilani, Pilani Campus
2^{nd} Sem. 2021-22
CS F211 Data Structures & Algorithms
=================================
Lab III
===

Topics: Runtime Memory Layout and Heap Allocation, Linked Lists, Measuring Running Time and Space Usage

Programming: C on Linux

Memory Layout

- The typical runtime virtual memory (i.e. the collection of logical addresses) of a process (i.e. a program under execution) is divided into two parts: ***code area*** (which is typically *read-only*) and ***data area***.
- The data-area is usually divided into three logical parts:
    - ***global / static area***, meant for allocating *global variables* (referred to as *static variables* because they are allocated once at the start of program execution and stay allocated until execution ends)
    - **stack,** (a.k.a. *call stack* a.k.a. *activation stack*) where local variables of each function/procedure are allocated when that function is called and deallocated when it returns. (Local variables are referred to as *automatic variables* because they are allocated and deallocated automatically on call and return of functions/procedures).
    - **heap*,*** meant for dynamic allocation (i.e. allocation specified by programmer and executed at runtime) and deallocation. (In C, dynamic allocation and deallocation are explicitly specified by programmer e.g. by invoking *malloc* and *free* explicitly. In contrast, in Java, dynamic allocation happens when a new object is created, and deallocation is implicit – done by a ***garbage collector***.).

Exercise 1: **[Expected Time: 15 minutes.]**
Write a program with multiple procedures (say, *p*, *g*, *h*, and *d*) invoked from *main* – both one after the other and from each other. Let each of these procedures contain a local variable *pilani*, *goa*, *hyd*, and *dub* respectively. Let *bits* be a global variable.

(a) Print the addresses of the local variables pilani, goa, hyd, and dub under different call sequences. Use the unary prefix operator **&** to obtain the address and use *"%u"* format specifier (in *printf*) to print an address as *unsigned int*.
(b) Modify the procedure *p* such that it calls itself with a single updated argument e.g. p(n) { … p(n+1) … }. Print the address of *n* each time p is invoked. Modify the test for termination of the recursion and repeat the test until you get a runtime error.
(c) Summarize your understanding (for yourself) of the stack space allocated.

Exercise 2: [Expected Time: 25 minutes]

(a) Define your own allocation and deallocation procedures *myalloc* and *myfree* respectively such that:

- *myalloc* invokes *malloc* to allocate the space as requested and returns the starting address of the allocated block; in addition, *myalloc* updates a global variable that keeps track of total space allocated in the heap so far.
- *myfree* invokes *free* to free the space pointed to by the given argument, and in addition, updates the global variable that keeps track of total space allocated in the heap.

(b) Write a loop that repeatedly allocates and frees a dynamic array of size M using your *myalloc* and *myfree* procedures. In each iteration:
   i. choose a random number M in the range 10,000 to 25,000.
   ii. allocate an array A of M integers. Use *sizeof* to make it portable.
   iii. print the addresses of the first and the last location of A i.e. A and &(A[M-1])
   iv. free A

The loop should terminate when allocation fails. Test the return value of *malloc* for failure.

(c) Summarize your understanding (for yourself) of heap space used.

Linked Lists

A linked list is said to be:

- ***linear*** if traversing the list from the head ends in a “last node” which does not point anywhere i.e. there is a node whose next is set to NULL.
- ***cyclic*** if traversing the list from the head leads one to cycle the list i.e. one of the nodes points to another node in the list.
- ***circular*** if traversing the list from the head leads one to cycle the list through the head i.e. one of the nodes points to the head node. [Note that a circular list is a special case of a cyclic list.]

Exercise 3: [Expected Time: 45 minutes]

(a) Write a procedure *createList(N)* that generates N random numbers and stores them in a linear linked list *Ls* and returns *Ls*. All the allocation in this procedure must use *myalloc*. Output the total heap space allocated to a text file. N must be large (say 1 million or more.)

(b) Write a procedure *createCycle(Ls)* that tosses a coin – programmatically – to decide whether *Ls* must be linear or cyclic. If it must be cyclic, this procedure picks a random number, say *r*, and lets the last node in *Ls* point to (i.e. set its next to point to) the r^{th} node from the head node. If it must be linear this procedure returns *Ls* as is.

(c) Write a main program that creates a new linked list *Ls* using the *createList* and *createCycle* procedures. Note that such a list Ls may or may not be cyclic. The main program should classify Ls as linear or cyclic by invoking a procedure *testCyclic* that is declared in a header file **cycle.h**. Test the program with a dummy implementation of testCyclic that returns FALSE always (or TRUE always).

Testing for Cycles in Linked Lists:

There are two ways to test for a cycle in a linked list:

1. **Hare-and-Tortoise algorithm:**
   Maintain two pointers ***hare*** and ***tort*** such that hare jumps two nodes in the linked list when tort jumps one node. Repeat the jumps in lock-step until ***hare*** finds itself behind ***tort*** or in step with ***tort*.** Ensure that trivial/special cases are handled properly: empty list, singleton list, a list with two nodes, and a linear list (i.e. one that terminates.)
2. **Link-Reversal algorithm**
   Reverse the links of a linked list as you traverse the linked list starting from the head. What will happen if there is a cycle in this case?

Exercise 4: [Expected Time: 60 minutes]

a) Define two different implementations of *testCyclic* in cycle1.c and cycle2.c using the two different techniques mentioned above.
b) Compile these two files separately with the main program file (and other required files) to create two different executables.
c) Run the executables multiple times for different values of N, the size of the linked list created.
   - Measure the running time taken (programmatically) and output N and time taken along with the amount of heap space used.
   - Measure the running time of individual procedures using the gnu profiler for each run.
   - Plot the values and summarize your understanding of the running times of these procedures.

Exercise 5: [Expected Time: 30 minutes]

Write a procedure makeCircularList(Ls) which tests whether a given linked list is linear or cyclic:

- If it is linear it creates a circular list by setting the last node of Ls point to the first node of Ls.
- If it is cyclic it deletes all nodes from the head before the cycle i.e. after deletion, the result is a circular list and only nodes that were within the cycle remain.

makeCircularList should return the modified list. Use myFree to delete nodes. Output the total heap space used at different points.

Assignment task corresponding to Lab sheet 3:

Solve Exercise-4 with all other required functions. As an assignment, you need to only implement cycle1.c, i.e. hare and the tortoise algorithm. Create an excel file (observations.xlsx) with three columns: value of N, Time Taken, Heap Space Used. Put all the files (along with the excel file) in a folder, name the folder with your BITS ID, zip it, and upload it as a part of assignment.

Deadline for this assignment is Tuesday, Feb 8, 23:59. Please note that you should upload the files at least 4 hours before this deadline so that any issue of uploading can be handled by you. In any case, do not send a mail to me with your assignment.