Birla Institute of Technology & Science, Pilani
2nd Semester 2016-17 - CS F211 – Data Structures and Algorithms

Lab 2 - 28th Jan 2017
Topics – Abstract Data Types and Performance Measurements

1. Implementing Abstract Data Types in C

An Abstract Data Type is realized by separating the user concerns (types and interface) from provider concerns (implementation). In C,

- a header file (with suffix ".h") is used to define the data type and declare the functions / procedures operating on it
- a code file (with suffix ".c") is used to define the implementation (I.e. define the functions / procedures).

For instance, consider the ADT Linked List, and an insert operation over it.

We use linkedlist.h and linkedlistImpl.c to realize this ADT:

```
/* linkedlist.h */
typedef … *linkedList;
void insert(linkedList);
…

/* linkedlistImpl.c */
void insert(linkedList list) {  return … ; }

…
```

Then, a (test) program that uses linkedList, say driver, can be implemented as a code file, say driver.c:

```
#include "linkedlist.h"
int main(int argc, char **argv)
{
  linkedList list;
  …
  insert(list, element);
   …
   return 0;
}
```

Problem 1: Use this approach to realize ADT List using a linked list implementation with following operations:

0	insertInFront	Should read an integer from stdin, insert it at the beginnning and call traverse function
1	insertAtEnd	Should read an integer from stdin, insert it at the end and call traverse function
2	deletFromFront	Should delete the element at the beginning of list and print on stdin, should print -2 in case of failure
3	deleteAtEnd	Should delete the element at the end of list and print on stdin, should print -2 in case of failure
4	traverse	Should print the linked list with a tab in between each element

A sample input file:

```
0
12
0
5
1
6
2
1
7
3
4
2
3
-1
```

Corresponding output should be:

```
12
5	12
5	12	6
5
12	6	7
7
12	6
12
6
```

INPUT

```
2
1
6
1
8
4
2
0
3
3
3
```

```
2
-1
```

OUTPUT:

```
-2
6
6	8
6	8
6
3	8
8
3
-2
```

INPUT

```
0
9
0
6
1
8
2
1
6
2
1
9
2
1
8
3
0
8
3
0
9
3
0
6
2
2
2
3
-1
```

OUTPUT

```
9
6	9
6	9	8
6
9	8	6
```

```
9
8	6	9
8
6	9	8
8
8	6	9
9
9	8	6
6
6	9	8
6
9
8
-2
--------------------------------------------

INPUT
0
8
0
4
1
16
2
4
3
4
3
-1

OUTPUT
8
4	8
4	8	16
4
8	16
16
8
8
------------------------------------------------------
INPUT
0
6
0
7
3
1
2
0
5
2
4
```

```
2
2
2
3
-1

OUTPUT
6
7	6
6
7	2
5	7	2
5
7	2
7
2
-2
-2
```

Problem 2: Suppose that the linked list may have a cycle. Then implement the “hare and tortoise” algorithm for detecting a cycle in the linked list: Set two pointers – “hare” and “tortoise” to the first node in the list. Then in lockstep, we move the hare pointer forward by two nodes and the tortoise pointer by one node. In their traversal, if the hare crosses the tortoise, then there is a cycle in the linked list. [For testing purposes, create a cycle in the linked list by setting the last node to point to one of the nodes of the linked list.]

2. Measuring (Dynamic) Memory Used and Time Taken

a. **Measuring Memory:** The amount of dynamic memory allocated can be kept track by programming. Define the following:

- a global array, heapAllocs, records <pointer, size>, keeping track of (heap) space allocated for each pointer
- a global variable, curHeapSize, keeping track of total (heap) space currently allocated
- a global variable, maxHeapSize, keeping track of the maximum total (heap) space allocated in this run of this program.

Define procedures *myAlloc* and *myFree* such that:

myAlloc calls *malloc* but also adds an element to heapAllocs, increments curHeapSize by the amount requested for allocation, and updates maxHeapSize if curHeapSize exceeds the previous maximum.

myFree calls *free* but also decrements curHeapSize, and deletes the record corresponding to the freed pointer from heapAllocs

In the end, you shall need to print heap memory left (i.e. not freed) and maximum heap memory allocated during the entire execution. You shall need to print the values in bytes.

b. The amount of running time can be measured using the *time* function. Use #include <time.h> this function. To measure the execution time of a particular function, you need to measure the clock time using *time* function before and after the function call and then take difference of the two. Refer to man pages on how to use this function.

Exercise:

(a) Given a file containing a list of N integers in increasing order, create a sorted list in decreasing order, Ls, implemented as a dynamically allocated array:
   i. Measure the time taken for deletion of elements at positions K, 2*K, ... (N/K)*K from Ls for a given K. [Note that deletions would result in left-shifting of elements.]
   ii. Repeat this for several values of K.
   iii. Also, measure the total heap space allocated.

(b) Repeat the exercise where the sorted list is implemented as a linked list.