COMP20003: Algorithms and Data Structures
Week 3 Workshop: Stacks and Queues

Objectives

- To implement the LIFO stack ADT as a dynamically allocated array.
- To implement the FIFO queue ADT as a doubly linked list.
- To revise dynamic memory allocation in C.

Preparation

- Read this sheet
- Read Chapter 10 Chapter 10 of Programming, Problem Solving, and Abstraction with C by A Moffat
- Revise the functions in the standard library that are concerned with dynamic memory allocation: malloc, free, realloc and calloc
- Familiarise yourself with the source code for stack and queue programs that will be worked on in this lab

Source Code

Download the source code from subject's LMS pages (workshop).

Part 1: Implementing a stack as a dynamically allocated array

The program stack reads its input from stdin (standard input, typically your terminal). The input for the stack program is a whitespace separated sequence of integers and asterixes, '*'.

Sample input:

```
10 20 30 * 54 * 3 * * *
2 3 4 *
* *
```

The program uses a single stack. When it reads an integer, it pushes that integer onto the stack. When it reads a '*' it pops the stack and writes out the integer that was on top of the stack to stdout (standard output, typically the screen attached to your terminal). The program prints a warning message if an attempt is made to pop an empty stack.

Stack.c currently contains code that implements all the required functionality except the code required for the stack. In this part of the exercise, you will add the code that is necessary to implement the stack.

We are going to implement the stack using a dynamically allocated array. A dynamically allocated array is one whose size is determined at runtime. (a.k.a. runtime arrays)

We wrap the array up in a structure called Stack. The structure Stack will contain the afro-mentioned runtime array as well as two integer fields: `size` and `top`. `size`
Is the current size of the array and `top` is the element of the array that represents the top of the stack.

You will need to add code to implement the following functions:

- stack_init
- stack_is_empty
- stack_push
- stack_pop

Their interfaces are described in comments in the source code. There must be no limit on the number of integers that can be pushed onto your stack. The stack begins with an initial size that is set by the argument to the stack_init function. If, while running the program, this initial size is exceeded then you will need to resize the array in order to increase the size of the stack. You should do this using the realloc function from C's standard library.

When testing the program, rather typing the input in at the terminal (which tends to be both tedious and error-prone), we suggest that you redirect standard input in order to provide the program with its input. Typing the following on the command line and pressing enter, will cause stack to read its input from the file data:

```
$ stack < data
```

You can use '>' to redirect standard output. The following command line will cause stack to read its input from the file 'data' and write its output to the file 'output':

```
$ stack < data > output
```

We have supplied some test cases that you may use to help test if your stack code is working correctly. These test cases are in the same directory as the source code. The files with the extension .inp are sample input files. The files with the extension .exp contain the expected output for each corresponding .inp file. If you redirect the output of each test case to a file, the you can compare your output against the expected output using unix diff command. This is particularly useful for large test cases where it is difficult to tell if there are any differences by eye.

If you are unfamiliar with the 'diff' command, you should consult diff's man page. You can access the man page for diff visa the following command:

```
$ man diff
```

Part 2: Implementing a queue as a doubly linked list

The program queue operates in a similar fashion to the stack program, except that when it reads an integer it puts (enqueues) the integer onto the rear of queue. When it reads a '*' it gets (dequeues) the integer that is in front of the queue and prints it out on stdout.

For this part of the exercise you must implement the queue as a doubly linked list. Recall that a singly linked list (usually referred to as just linked list) can be implemented as a structure that contains some data and a pointer to another structure of the same type. For example:

```
typedef struct node Node;

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

We create instances of these structures by allocating memory for them via the malloc function and "link" them together via the pointers, in this case the next field.

A doubly linked list is a variation on this basic idea. Instead of a pointer to the next element, a node in a doubly linked list also contains a second pointer, the is used to point back to the previous element:

```
typedef struct dnode DNode;

struct dnode {
     int data;
     DNode *next;
     DNode *prev;
}
```

Using doubly-linked lists simplifies the implementation of some list operations, such as insertion into an ordered list, because we always have access to the previous node in the list, via the prev pointer. The trade-off, of course, is that doubly-linked lists require extra space, because there is an additional pointer in each ode.

We are going to implement a FIFO (first-in-first-out) queue using a doubly linked list. In the source code, the structure Queue represents a queue. (It is the "header" node that the textbook discusses.) It contains two pointer field, front and rear. The field front points to the head of the list that we are going to use to implement the Queue. The field rear points to the back of the list.

You will need to implement the following functions:

- queue_init

- queue_is_empty
- queue_enqueue
- queue_dequeue

As with the stack program we have provided several test cases and their expected outputs. Again, you should show your lab demonstrator your completed program.

You should keep a copy of your work for this exercise because we will be using it in future laboratory exercises.

Part 3: Postfix evaluation function.

Use the program written in Part 1 to solve postfix evaluation discussed in Week 2 lectures.

You will need to modify postfix_main.c to implement this function.