School of Electrical & Electronic Engineering

EE2008 Data Structure & Algorithms

Academic Year 2016-2017

L2008A

Abstract Data Types & Their Implementations

Software Engineering Laboratory (S2.2-B4-04)

Laboratory Manual

1. Objectives

This experiment aims to provide students with hands-on training in order that they are able to gain a better understanding on the design and implementation of abstract data types. Students will be required to write programs to

(i) implement ADTs using appropriate data structures
(ii) process and/or manipulate data elements associated with ADTs

2. Introduction to Abstract Data Types

An abstract data type (ADT) consists of data together with functions that operate on the data. An abstract data type is "abstract" in the sense that it does not specify how data are represented and how functions are implemented. Only the *behaviour* of the functions is specified. Examples of ADTs include that of Stacks and Queues.

2.1 The Stack ADT

A stack is a list of items with the restriction that insertion and deletion can only be performed in only one position, namely the end of the list called the *top.* The fundamental operations of a stack are *push*, which is equivalent to insert, and *pop*, which deletes the most recently added item. Stacks are sometimes known as *LIFO* (last in, first out) lists and a model depicting this ADT is shown in Figure 1.

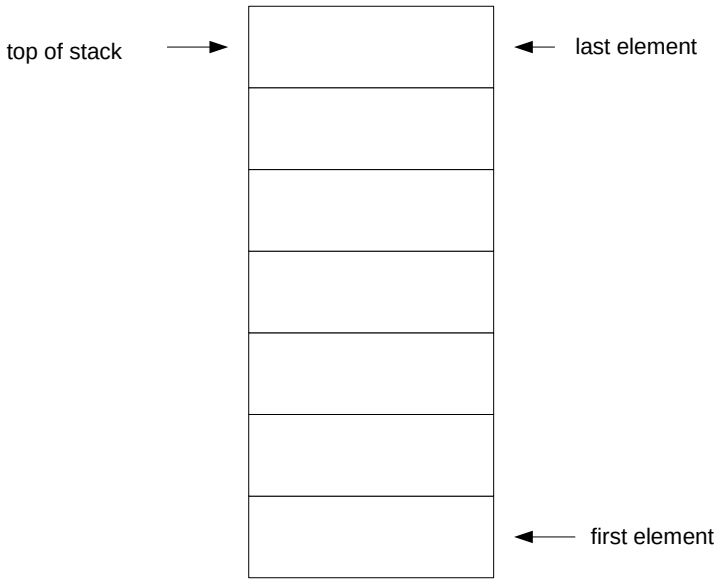

Figure 1: Model of a Stack

2.2 The Queue ADT

A queue is a list of items where insertion is done at one end and deletion is performed at the other end of the list. The basic operations on a queue are *enqueue*, which inserts an element at the end of the list (called the *rear*), and *dequeue*, which deletes (and returns) the element at the start of the list (known as the *front*). Figure 2 shows a model of a queue. Queues are sometimes known as *FIFO* (first in, first out) lists.

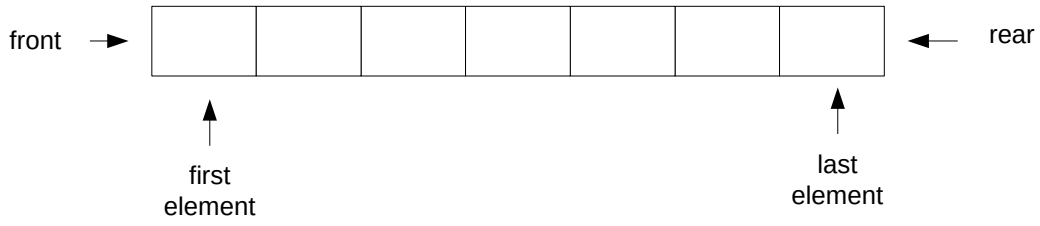

Figure 2: Model of a Queue

3. Implementation of a stack

A stack may be implemented using an array. We use a variable *t* to index the top of the stack. When an item is pushed onto the stack, we increment *t* and put the item in the cell at index *t*. When an item is popped off the stack, we decrement *t*.

3.1 Algorithms for Stack Functions

The following are the pseudocodes which outline the various operations that may be performed on a stack.

Algorithm 3.1 Initializing a Stack
This algorithm initializes the stack to empty. An empty stack has *t*=-1.

Input: None
Output: None

```
stack_init() {
   t = -1
}
```

Algorithm 3.2 Testing for an Empty Stack
This algorithm returns true if the stack is empty or false otherwise. An empty stack has *t*=-1.

Input: None
Output: a boolean value

```
empty() {
   return t == -1
}
```

Algorithm 3.3 Adding an Element to a Stack
This algorithm adds the value *val* to a stack. The stack is represented using an array *data*. The algorithm assumes that the array is not full. The most recently added item is at index *t* unless the stack is empty, in which case, *t* = -1.

Input: *val*
Output: None

```
push(val) {
   t = t + 1
   data[t] = val
}
```

Algorithm 3.4 Removing an Element from a Stack
This algorithm removes the most recently added item from a stack. The algorithm assumes that the stack is not empty. The most recently added item is at index *t*.

```
Input:    None
Output:   None
pop() {
   t = t – 1
}
```

Algorithm 3.5 Returning the Top Element in a Stack
This algorithm returns, but does not remove, the most recently added item in a stack. The algorithm assumes that the stack is not empty. The stack is represented using an array *data*. The most recently added item is at index *t*.

```
Input:    None
Output:   value from top of stack
top() {
   return data[t]
}
```

An illustration of the stack operations is given in Figure 3.

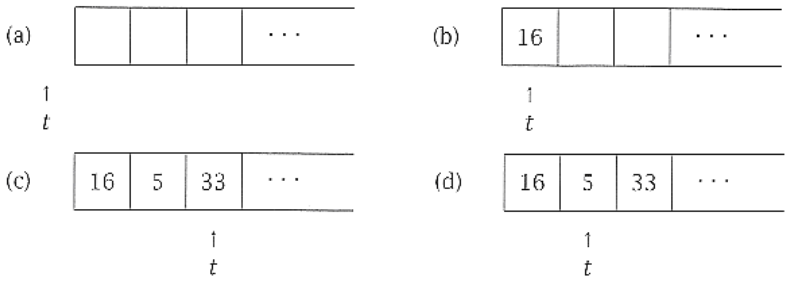

Figure 3: Implementing a stack *s* using an array, whose first index is 0. The top of the stack is at index *t*. An empty stack, shown in (a), has $t = -1$. After *s.push*(16), we have the situation shown in (b). After *s.push*(5) and *s.push*(33), we have the situation shown in (c). After *s.pop*(), we have the situation shown in (d).

3.2 Implementation Requirements

Write a C program to

(i) implement the following functions of a Stack using an integer array:
- stack_init(): Make the stack empty
- empty(): return true if the stack is empty. Return false if the stack is not empty
- push(*val*): add the item *val* to the stack
- pop(): remove the item most recently added to the stack
- top(): return the item most recently added to the stack, but do not remove it.

(ii) the program reads in integer values as input:

- for a positive integer input value greater than 0, the program will *push* the value into the stack
- for any negative integer input value, the program will return the most recently added value from the stack if it is non-empty, i.e. *top*
- for the input value 0, the program will remove the most recently added value from the stack if it is non-empty, i.e. *pop*

(iii) test your program with the following set of input:

Input	Expected Output
0	“Error - Stack is empty”
78	
456	
-1	456
0	
-1	78
60	
33	
-1	33
0	
-1	60
0	
0	
-1	“Error - Stack is empty”
0	“Error - Stack is empty”

4. Implementation of a Queue

Like a stack, a queue may be implemented using an array. In a queue, items are added at rear and deleted from the front; thus, we use two variables, *r* and *f*, to track the indexes of the rear and front of the queue. An empty queue has both *r* and *f* equal to -1. When an item is added (at the rear) to an empty queue, we set *r* and *f* to 0 and put the item in the cell at index 0. When an item is added to a nonempty queue, we increment *r* and put the item in the cell index *r*. If *r* is the index of the last cell in the array, we "wrap around" by setting *r* to 0. When an item is deleted (at the front) from the queue, we increment *f*. If *f* is the index of the last cell in the array, we "wrap around" by setting *f* to 0. When an item is deleted from the queue and the queue becomes empty, we set *r* and *f* to -1.

4.1 Algorithms for Queue Functions

The following are the pseudocodes which outline the various operations that may be performed on a queue.

Algorithm 4.1 Initializing a Queue
This algorithm initializes a queue to empty. An empty queue has *r* = *f* = -1.

```
Input:      None
Output:     None
queue_init() {
   r = f = - 1
}
```

Algorithm 4.2 Testing for an Empty Queue
This algorithm returns true if the queue is empty or false if the queue is not empty. An empty queue has *r* = *f* = -1.

```
Input:      None
Output:     a Boolean value
empty() {
   return r == - 1
}
```

Algorithm 4.3 Adding an Element to a Queue
This algorithm adds the value *val* to a queue. The queue is represented using an array *data* of size *SIZE*. The algorithm assumes that the queue is not full. The most recently added item is at index *r* (rear), and the least recently added item is at index *f* (front). If the queue is empty, *r* = *f* = -1.

```
Input:      val
Output:     None
enqueue(val) {
                    if (empty())
                      r = f = 0
                    else {
                      r = r + 1
                      if (r == SIZE)
                      r = 0
                    }
                    data[r] = val
}
```

Algorithm 4.4 Removing an Element From a Queue

This algorithm removes the least recently added item from a queue. The queue is represented using an array of size *SIZE*. The algorithm assumes that the queue is not empty. The most recently added item is at index *r* (rear), and the least recently added item is at index *f* (front). If the queue is empty, *r* = *f* = -1.

```
Input:      None
Output:     None
dequeue() {
                    // does queue contain one item?
                    if (r = = f)
                      r = f = -1
                    else {
                      f = f + 1
                      if (f == SIZE)
                      f = 0
                    }
}
```

Algorithm 4.5 Returning the Front Element in a Queue

This algorithm returns, but does not remove, the least recently added item in a queue. The algorithm assumes that the queue is not empty. The queue is represented using an array *data*. The least recently added item is at index *f* (front).

```
Input:      None
Output:     the value of the least recently added item
front() {
   return data[f]
}
```

An illustration of the queue operations is given in Figure 4.

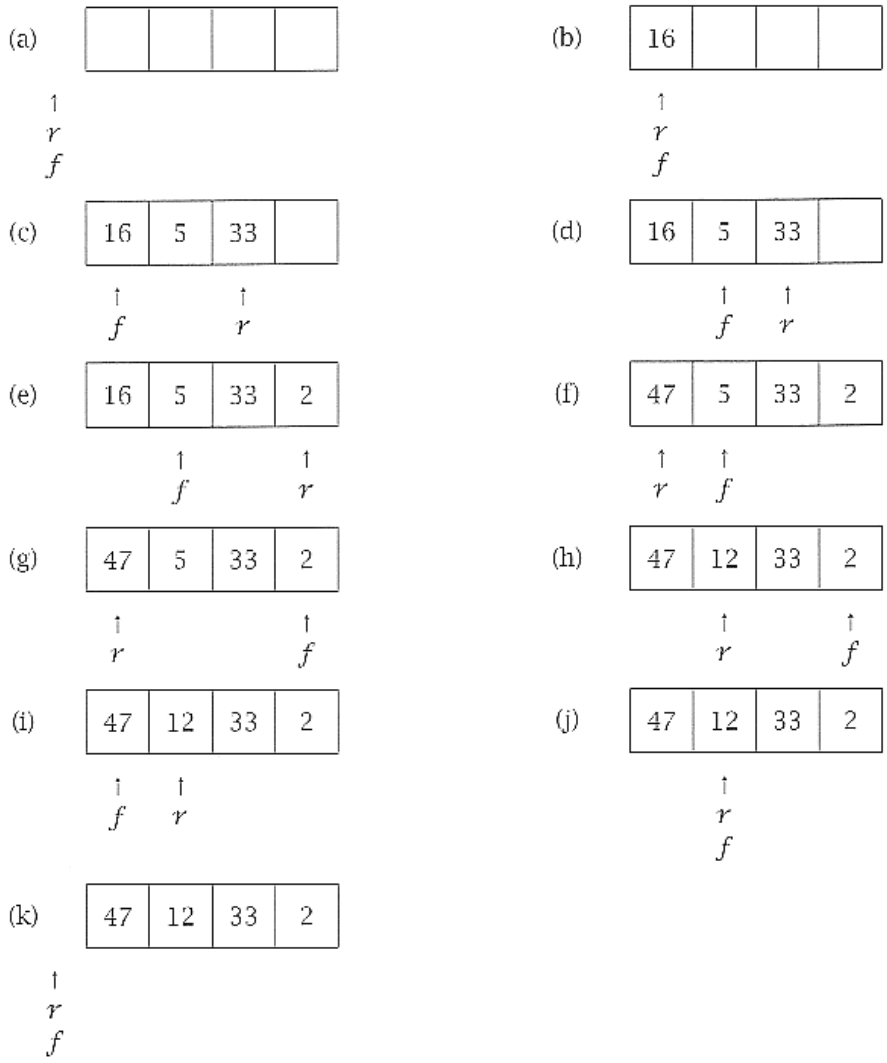

Figure 4: Implementing a queue *q* using an array of size 4, whose first index is 0. The rear of the queue is at index *r*, and the front of the queue is at index *f*. An empty queue, shown in (a), has $r = f = -1$. After *q.enqueue*(16), we have the situation shown in (b). After *q.enqueue*(5) and *q.enqueue*(33), we have the situation shown in (c). After *q.dequeue*(), we have the situation shown in (d). After *q.enqueue*(2), we have the situation shown in (e). After *q.enqueue*(47), we have the situation shown in (f). After *q.dequeue*() and *q.dequeue*(), we have the situation shown in (g). After *q.enqueue*(12), we have the situation shown in (h). After *q.dequeue*(), we have the situation shown in (i). After *q.dequeue*(), we have the situation shown in (j). After *q.dequeue*(), we have the situation shown in (k).

4.2 Implementation Requirements

Write a C program to

(i) implement the following functions of a Queue using an integer array:

- queue_init(): Make the queue empty
- empty(): return true if the queue is empty. Return otherwise
- enqueue(*val*): add the item *val* to the queue
- dequeue(): remove the item least recently added to the queue
- front(): return the item least recently added to the queue, but do not remove it.

(ii) the program reads in integer values as input:

- for a positive integer input value greater than 0, the program will insert the value into the queue, i.e. *enqueue*
- for any negative integer input value, the program will return the least recently added value from the queue if it is non-empty, i.e. *front*
- for the input value 0, the program will remove the least recently added value from the queue if it is non-empty, i.e. *dequeue*

(iii) test your program with the following set of input:

Input	Expected Output
0	"Error - Queue is empty"
78	
456	
-1	78
0	
-1	456
60	
33	
0	
-1	60
0	
-1	33
0	
-1	"Error - Queue is empty"
0	"Error - Queue is empty"

5. Additional Implementation Requirements

Some of the following parts will be selected for the student to implement during their lab session.

a) Write a function *is_full* that returns true or false to indicate whether the array that implements a stack is full. Assume that SIZE specifies the size of the array.

b) Write a version of *push* that checks for a full array. If the array is full, the function simply returns false. If the array is not full, the behaviour is the same as the original *push*, except that the function also returns true. Assume that *SIZE* specifies the size of the array.

c) Write a version of *enqueue* that checks for a full array. If the array is full, the function simply returns false. If the array is not full, the behaviour is the same as the original *enqueue*, except that the function also returns true.

d) Write a function *rear* that returns, but do not remove, the most recently added item in a queue.

A Sample Program

```
#include <stdio.h>                    /* Include header file for printf & scanf */
#define max 5                         /* Define a constant */

int t, var, i;                        /* Declare variables */
int marks[max+1];                     /* Declare a one-dimensional array */

void marks_array_init(void);          /* Function prototype */
void insert_marks(int);               /* Function prototype */


void main()                           /* Main program. */
{
   marks_array_init();                /* call function to initialise array  */
   for (i=1; i<=5; i++)
   {
      printf("\nInput marks for Student %d =>", i);
      scanf("%d", &var);
      insert_marks(var);              /* call function to insert marks into array*/
   }

   for (i=1; i<=5; i++)
       if (marks[i] > 50)
          printf("Student %d  : Pass\n", i);
       else
          printf("Student %d  : Fail\n", i);
}


void marks_array_init( void )         /* Function to initialise array */
{
  t = -1;
}


void insert_marks( int x )            /* Function to insert marks into array */
  {
    t = t+1;
    marks[t]=x;
  }
```

Some Explanatory Notes on the Sample Program

- Commands intended for the C preprocessor, instead of the C compiler itself, start with a "#" and are known as "preprocessor directives" or "metacommands". The sample program has two such metacommands:

```
#include <stdio.h>
#define max 5
```

  The first statement, "#include <stdio.h>", simply merges the contents of the file "stdio.h" into the current program file before compilation. The "stdio.h" file contains declarations required for use of the standard-I/O library, which provides the "printf()" and scanf() functions.

- Any part of the program that starts with /* and ends with */ is called a *comment*. The purpose of using comments is to provide supplementary information to make it easier to understand the program.

- A C program is built up of one or more functions. The program above contains three user-defined functions, "main()", "marks_array_init()" and “insert_marks()”.

- The "main()" function is mandatory when writing a self-contained program. It defines the function that is automatically executed when the program is run. All other functions will be directly or indirectly called by "main()".

- You call a C function simply by specifying its name, with any arguments enclosed in following parentheses, with commas separating the arguments. For example the statement “insert_marks(var);” in the sample program calls the function “insert_marks()” with argument “var”.

- All variables in a C program must be "declared" by specifying their name and type. The sample program declares three variables for the "main()” program:

```
int  t, var, i;
```

  The above statement declares “t” , “var” and “i” as integer variables.

- You'll notice that besides the variable declarations, there is also a function declaration, or "function prototype", that allows the C compiler to perform checks on calls to the function in the program and flag an error if they are not correct. The sample program declares the following two function prototypes:

```
void marks_array_init(void);
void insert_marks(int);
```

  The function prototypes declare the type of value the function returns (the type will be "void" if it does not return a value), and the arguments that are to be provided with the function.

- The "printf()" library function provides text output capabilities for the program. For example, assuming the value of variable *i* is 1, the statement

  printf("\nInput marks for Student %d =>", *i*);

  will display the text : Input marks for Student 1 =>

  The printf() statement doesn't automatically add a "newline" to allow the following printf() statement to print on the next display line. You must add a newline character ("\n") to force a newline.

- You can also include "format codes" in the string and then follow the string with one or more variables to print the values they contain. For example, in the above statement, "%d" is the format code that tells "printf" to print a decimal integer (i.e. we assume that *i* have been declared as an integer variable).

- The scanf() function reads data from the keyboard according to a specified format and assigns the input data to one or more program variables. Like printf(), scanf() use a format string to describe the format of the input. For example, the statement

  scanf("%d", &var);

  reads a decimal integer from the keyboard and assigns it to the integer variable *var*.