City University, Bangladesh

Midterm Lab Report (Summer 2018)

Course Title: Data Structure

Course Code: CSE 213

**Submitted By**

**Rashed Talukder**

**Id: 171442521**

**Program: CSE (Day)**

**Batch: 44th**

**Submitted To**

**Richard Philip**

**Lecturer**

**Dept. of CSE**

**City University**

Submission Date: 10-10-2018

**Data & Structure**

**What is *Data*?**

Data means raw facts or information that can be processed to get results.

**What is *Structure*?**

Some elementary items constitute a unit and that unit may be considered as a structure.

A structure may be treated as a frame where we organize some elementary items in different ways.

**Data Structure**

**So, what is *Data Structure*?**

Data structure is a structure where we organize elementary data items in different ways and there exits structural relationship among the items so that it can be used efficiently.

In other words, a data structure is means of structural relationships of elementary data items for storing and retrieving data in computer’s memory.

**Data Structure…**

Usually elementary data items are the ***elements*** of a data structure.

However, a ***data structure may be an element of another data structure***. That means a data structure may contain another data structure.

**Time Complexity**

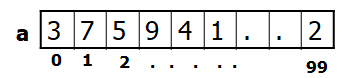
This complexity is related *to execution time* of the algorithm.

It depends on the number of element*comparisons* and number of element *movement* (movement of data from one place to another).

**The first Data Structure - An Array!**

An array in C / C++ is a collection of related data elements of the same type that are referenced by a common name and stored in a contiguous memory location.

The simplest form of an Array is a *one dimensional array* that may be defined as a finite ordered set of homogenous elements.

For Example 

int a[100];

**One Dimensional Array: Declaration**

* A single dimensional array declaration has the following form :

data\_type    array\_name[array\_size];

**Array Initialization**

⚫ In line initialization of an array may takes the following form: type   array\_name[size] = {element\_list};

**Two-Dimensional Array**

⚫ Some data fit better in a table with several rows and columns.

⚫ This can be constructed by using two-dimensional arrays using two subscripts/indices.  The first refers to the row, and the second, to the column.

**datatype  array\_name[1st dimn size][2nd dimn size];**

Example

int ROWS = 4, COLS = 3;  
 int exams[ROWS][COLS];

**Structure**

⚫The array takes simple data types like int, char or double and organizes them into a linear array of elements all of the same type.

⚫Now, consider a record card which records *name*, *age* and *salary*. The name would have to be stored as a *string*, the age could be int and salary could be float. As this record is about one person, it would be best if they are all stored under one variable.

⚫At the moment the only way we can work with this collection of data is as separate variables. This isn't as convenient as a single data structure using a single name and so the C language provides *structure*.

⚫A *structure* is an aggregate data type built using elements of other types.

**Defining Structure in C**

In general “structure” in C/C++ is defined as follows:

struct name{

list of component variables

};

For example

struct EmployeeRecord{

char name[5];

int age; float salary; } ;

**Stack**

⚫A stack or LIFO (last in, first out) is an abstract data type that serves as a collection of elements, with two principal operations:

push adds an element to the collection;

pop removes the last (top of the stack) element that was added.

⚫Bounded capacity

If the stack is full and does not contain enough space to accept an entity to be pushed, the stack is then considered to be in an overflow state.

A pop either reveals previously concealed items or results in an empty stack – which means no items are present in stack to be removed.

⚫Non-Bounded capacity

Dynamically allocate memory for stack. No overflow.

**Stack**

int Stack[100], Top=0, MaxSize=100; // Stack holds the elements;

// Top is the index of Stack always pointing to the first/top element of the stack.

bool IsEmpty(); // returns True if stack has no element

bool IsFull(); // returns True if stack full

void Push(int Element); // inserts Element at the top of the stack

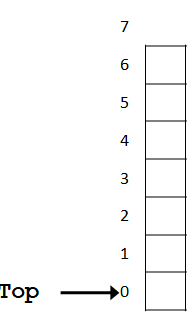
int Pop(); // deletes top element from stack into Element

int TopElement(); // gives the top element in Element

void Display(); // prints the whole stack

⚫**bool** IsEmpty(){

**// returns True if stack has no element**

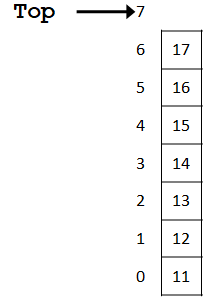
**return** (Top == 0); } Considering **MaxSize** = 7 

⚫**bool** IsFull(){

**// returns True if stack full**

**return** ( Top == MaxSize );

} Considering **MaxSize** = 7



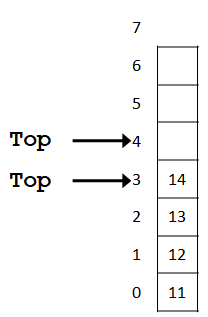
⚫void push(int data){

head++;

stack[head] = data;

} Considering **MaxSize** = 7

There are 3 elements inside Stack

So next element will be pushed at index 3 

⚫int pop(){

int data = stack[head];

head--;

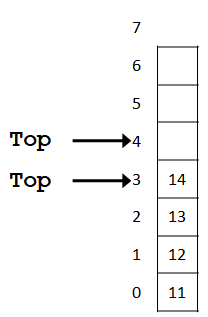
return data;

}

Considering **MaxSize** = 7

There are 4 elements inside Stack

So element will be popped from index 3



⚫**int** TopElement(){

**// gives the top element in Element**

**return** Stack[ Top - 1 ];

}Considering **MaxSize** = 7

There are 4 elements inside Stack

So top element will be at index 3

⚫void display(){

printf("Data in your stack\n");

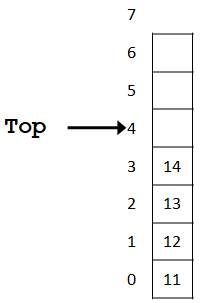
for(int i=0;i<=head;i++){

printf("%d",stack[i]); }}

Considering **MaxSize** = 7

There are 4 elements inside Stack

So element will be shown from index 3 down to index 0.



**Queue**

⚫A queue is a waiting line – seen in daily life

⚫A line of people waiting for a bank teller

⚫A line of cars at a toll both

⚫Queue data structure is like a container with both end opening.

An end called **rear**

Another end is **front**

⚫This mechanism is called First-In-First-Out (FIFO).

⚫Some of the applications are :

Device queue, printer queue, keystroke queue, etc.

**Queue in Computer Language**

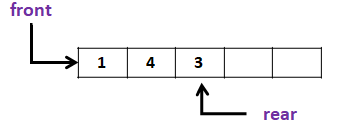
⚫A queue is a sequence of data elements

⚫In the sequence

* + Items can be added only at the **rear**
  + Items can be removed only at the other end, **front**

⚫Basic operations

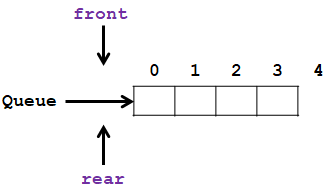
* + Check **IsEmpty**
  + Check **IsFull**
  + **EnQueue** (add element to back i.e. at the rear)
  + **DeQueue** (remove element from the front)
  + **FrontValue** (retrieve value of element from front)
  + **ShowQueue** (print all the values of queue from front to rear)



**Queue – Operation**

**Queue[4], MaxSize=4;**

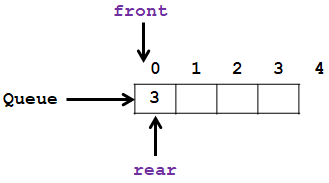
**Initialize( )**🡪**front=rear=-1**

****

**Queue[4],MaxSize=4;**

Initialize( ) 🡪 front=rear=-1

**EnQueue( 3 )** 🡪 **front=rear=0**

****

**Queue[4],MaxSize=4;**

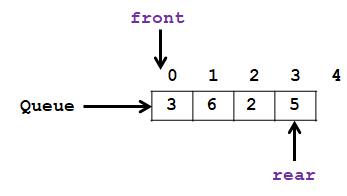
Initialize( ) 🡪 front=rear=-1

EnQueue( 3 ) 🡪 front=rear=0

EnQueue( 6 )

EnQueue( 2 )

**EnQueue( 5 )**



**Queue[4],MaxSize=4;**

Initialize( ) 🡪 front=rear=-1

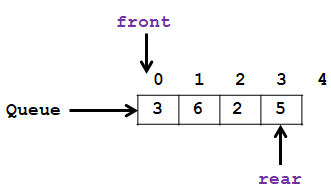
EnQueue( 3 ) 🡪 front=rear=0

EnQueue( 6 )

EnQueue( 2 )

EnQueue( 5 )

**EnQueue( 9 )** 🡪 **Queue Full, (rear==(MaxSize-1))**



**Queue[4],MaxSize=4;**

Initialize( ) 🡪 front=rear=-1

EnQueue( 3 ) 🡪 front=rear=0

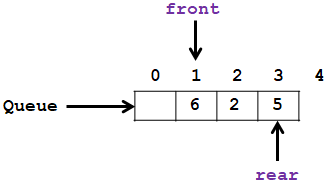
EnQueue( 6 )

EnQueue( 2 )

EnQueue( 5 )

EnQueue( 9 ) 🡪 Queue Full, (rear==(MaxSize-1))

**DeQueue()** 🡪 **3**

****

**Queue[4],MaxSize=4;**

Initialize( ) 🡪 front=rear=-1

EnQueue( 3 ) 🡪 front=rear=0

EnQueue( 6 )

EnQueue( 2 )

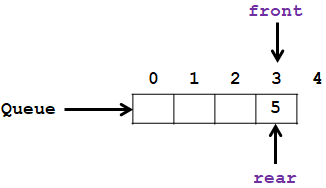
EnQueue( 5 )

EnQueue( 9 ) 🡪 Queue Full, (rear==(MaxSize-1))

DeQueue() 🡪 3

DeQueue() 🡪 6

**DeQueue()** 🡪 **2**



**Queue[4],MaxSize=4;**

Initialize( ) 🡪 front=rear=-1

EnQueue( 3 ) 🡪 front=rear=0

EnQueue( 6 )

EnQueue( 2 )

EnQueue( 5 )

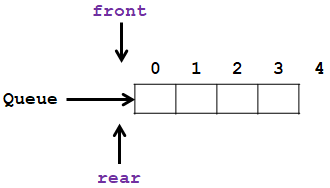
EnQueue( 9 ) 🡪 Queue Full, (rear==(MaxSize-1))

DeQueue() 🡪 3

DeQueue() 🡪 6

DeQueue() 🡪 2

**DeQueue()** 🡪 **5, front=rear=-1**



**Queue[4],MaxSize=4;**

Initialize( ) 🡪 front=rear=-1

EnQueue( 3 ) 🡪 front=rear=0

EnQueue( 6 )

EnQueue( 2 )

EnQueue( 5 )

EnQueue( 9 ) 🡪 Queue Full, (rear==(MaxSize-1))

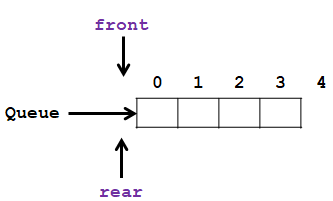
DeQueue() 🡪 3

DeQueue() 🡪 6

DeQueue() 🡪 2

DeQueue() 🡪 5, front=rear=-1

**DeQueue()** 🡪 **Queue Empty, (front==-1) && (rear==-1)**



**Linked List**

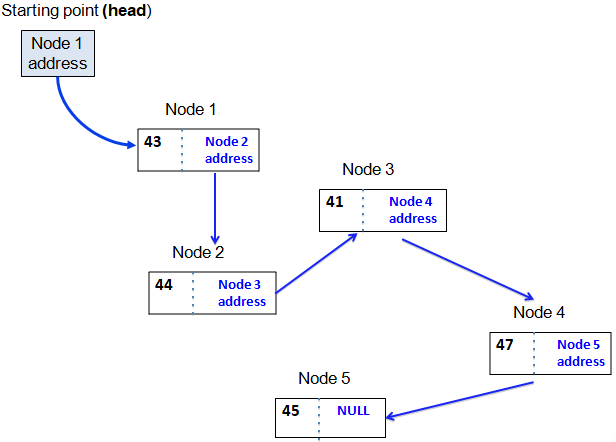
🞟Linked list is a data structure consisting of a group of memory space which together represent a list i.e. a sequence of data.

🞟A sequence of data can also be represented as an array. But in an array, data are stored consecutively in the memory

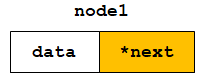
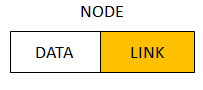
🞟A linked list is also a sequence of data. But in a linked list the data are not stored consecutively in the memory.

🞟Each data is stored in a separate memory space/block (called cell/node).

🞟Each memory block contains the data along with link/location/address to the memory location for the next data in the list.



**Representation – node**

**Representation of a NODE in C/C++**

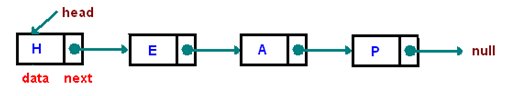
* **struct ListNode{ //Each NODE of the list contains- int data; The Data item**

**ListNode \*next; A LINK/POINTER to store the next**

**}; nodes address.**

**ListNode node1;**

**Basics of Linked Lists**



* A **group of nodes**.
* **Each node** represents by a block of memory and it has two fields/parts:
  1. **data**/value and
  2. a next **pointer** to point the **next** node.

**Type of Linked list**

1. Singly linked list (that has single references)
2. Doubly linked list (that has two references)
3. Multiply linked list (that has more than two references)
4. Circular list (tail element's next pointer points to the head element)

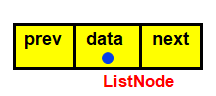
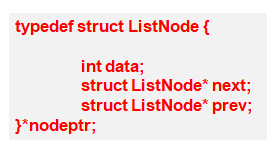
**Two-way linked lists**

* One that can be traversed in both directions – forward and backward
* Every node has two pointers:
  + One pointing to the next node (except the last node)
  + One pointing to the previous node (except the first node)
* Two external pointers - head and tail

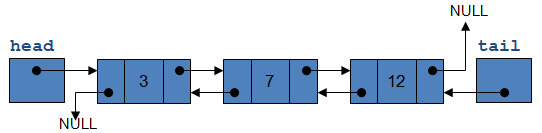
**Two way or Doubly Linked Lists**

Each node of the list contains

* + - the data item (an object)
    - a pointer to the next node
    - a pointer to the previous node

**Two-way linked list of integers**

****

The End