MES College of Engineering, Pune-01

Faculty Orientation Workshop on Data Structures

Under the Aegis of BoS (E&TC), SPPU, Pune

SE E&TC/ Electronics) 2019 Course

(22nd to 26th June 2020)

Dr. Manisha P. Dale

Data Structures - Course Details

Programme: UG Programme in E&TC/ Electronics Engineering			Class: S.E (E&TC/ Electronics)			A.Y. 2020-21 Sem. I		
Course Code: 204184 (Theory)			Course : Data Structures					
Corresponding Lab Course Code : 204188			Lab Course Name: Data Structure Laboratory					
Teaching Scheme			Examination Scheme					
Theory	Practical	Tutorial	Theory				Lab	
(hrs/week)	(hrs/week)	(hrs/week)	~~Online~~/ Insem	Endsem	Sessional	Term Work	Practical	Oral
3 hrs	2 hrs	----	30	70	---	--	--	25

Abstract: On completion of this course, students will be familiar with C programming and should be able to implement sorting and searching algorithms and calculates it complexity. Students will be able to develop applications of stacks and queues using array and also demonstrate applicability of linear and non-linear data structures. Students will be able to design height balanced Binary Tree and apply the knowledge of graph for solving the problems of spanning tree and shortest path algorithm.

Unit III

UNIT III: Stack and Queue

Stack: Concept, Basic Stack operations, Array representation of stack, Stack as ADT, Stack

Applications: Reversing data, Arithmetic expressions conversion and evaluation.

Queue: Concept, Queue operations, Array representation of queue, Queue as ADT, Circular queue, Priority Queue,

Applications of queue: Categorizing data, Simulation of queue.

Data Structures: Books

TEXT BOOKS

T1: Ellis Horowitz, Sartaj Sahni, "Fundamentals of Data Structures", Galgotia Books Source.
T2: Richard. F. Gilberg, Behrouz A. Forouzan, "Data Structures: A Pseudocode Approach with C," Cengage Learning, second edition.

REFERENCE BOOKS

R1: Seymour Lipschutz, "Data Structure with C, Schaum's Outlines", Tata McGrawHill.
R2: E Balgurusamy, "Programming in ANSI C", Tata McGraw-Hill, Third Edition.
R3: Yedidyah Langsam, Moshe J Augenstein, Aaron M Tenenbaum "Data structures using C and C++" PHI Publications, 2nd Edition.
R4: Reema Thareja, "Data Structures using C", Second Edition, Oxford University Press, 2014

ADDITIONAL MATERIAL

MOOC / NPTEL:

1. NPTEL Course "Programming & Data Structure" https://nptel.ac.in/courses/106/105/106105085/
2. NPTEL Course "Data Structure & Algorithms" https://nptel.ac.in/courses/106/102/106102064/

Data Structures: Lab Experiments

Group A (Compulsory) Write a C program to:	
1	Perform following String operations with and without pointers to arrays (without using the library functions): a. substring b. palindrome c. compare d. copy e. reverse
2	Implement Database Management using array of structures with operations Create, Display, Modify, Append, Search and Sort. (For any database like Employee or Bank database with and without pointers to structures)
3	Implement Stack and Queue using arrays.
4	Create a singly linked list with options: a. Insert (at front, at end, in the middle), b. Delete (at front, at end, in the middle), c. Display, d. Display Reverse, e. Revert the SLL
5	Implement Binary search tree with operations Create, search, and recursive traversals.
6	Implement Graph using adjacency Matrix with BFS & DFS traversals.

Data Structures: Lab Experiments

Group B	
Write a C program to:	
1	Implement assignment 2 using files.
2	Add two polynomials using linked lists.
3	Evaluate postfix expression (input will be postfix expression)
4	Reverse and Sort stack using recursion.

Group C	
Write a C program to:	
1	Implement merge sort for doubly linked list.
2	Implement Dijkstra’s Algorithm.

VIRTUAL LAB: (A few experiments related to the course available on Virtual labs):

1. Data Structures-I: https://ds1-iiith.vlabs.ac.in/data-structures-1/

2. Data Structures -II: https://ds2-iiith.vlabs.ac.in/data-structures-2/

3. Data Structures Lab: http://cse01-iiith.vlabs.ac.in/

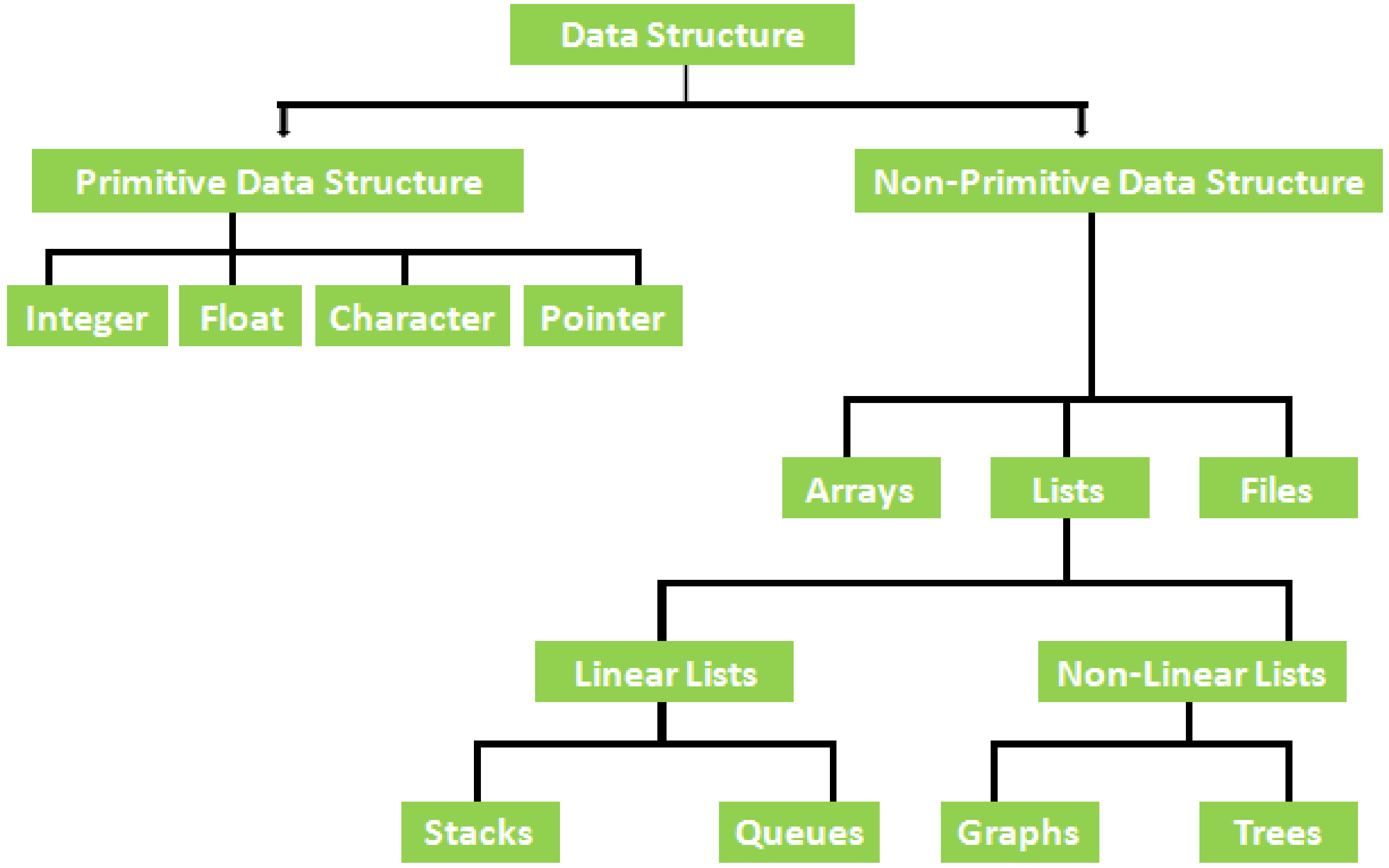
4. Computer Programming Lab: http://cse02-iiith.vlabs.ac.in/

Data Structures: Topic – Book – Pages Mapping

Sr. No.	Topic	Reference / text book with page no.
UNIT III: Stack and Queue		
3.1	• Concept • Basic Stack operations, • Array representation of stack • Stack as ADT • Stack Applications: Reversing data, Arithmetic expressions conversion and evaluation.	T2 (79-82) T2 (95-101) T2 (102-121)

3.2	• Concept
	• Queue operations
	• Array representation of queue
	• Queue as ADT
	• Circular queue & Priority Queue
	• Applications of queue: Categorizing data, Simulation of queue
		T2-(147-150)
		T2 (159-166)
		T1(119-120)
		T2 (168-178)

Classification of Data Structures



Introduction

--> An array is a *random access* data structure, where each element can be accessed directly and in constant time.

E.g.: A typical illustration of random access is a book - each page of the book can be open independently of others.

Random access is critical to many algorithms, for example binary search.

-> A linked list is a *sequential access* data structure, where each element can be accesed only in particular order.

E.g.: A typical illustration of sequential access is a roll of paper or tape - all prior material must be unrolled in order to get to data you want.

Stack and Queue

Are stacks and queues useful?

- YES. They come up all the time.

❑ **Stacks**

- Web browsers store the addresses of recently visited sites on a stack
- Each time the visits a new site ==> pushed on the stack. Browsers allow to "pop" back to

previously visited site.

- The undo-mechanism in an editor

→ The changes are kept in a stack. When the user presses "undo" the stack of changes is popped.

- The function-call mechanism

→ the active (called but not completed) functions are kept on a stack

→ each time a function is called, a new frame describing its context is pushed onto the stack the context of a method: its parameters, local variables, what needs to be returned, and

where to return (the instruction to be executed upon return)

→ when the function returns, its frame is popped, the context is reset to the previous method (now on top of the stack) and the program continues by executing the previously suspended method.

Stack and Queue

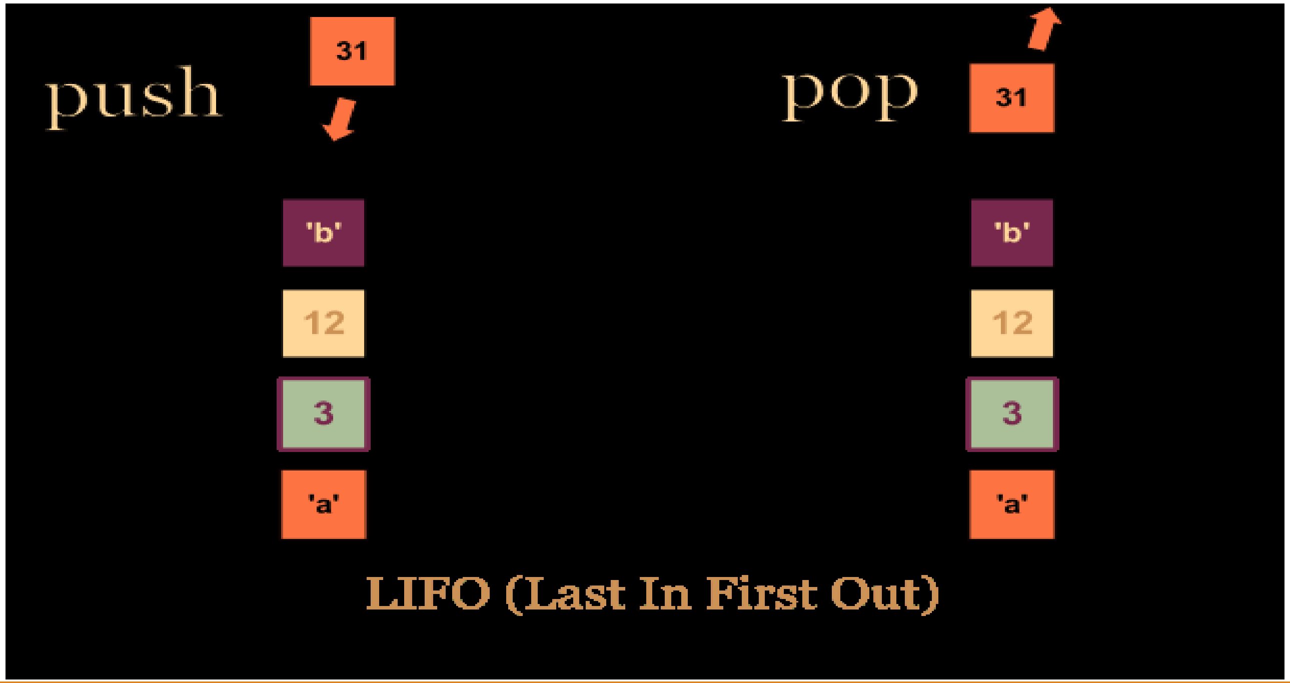
- **Queue**

• Queue of processes waiting to be processed for e.g. the queue of processes to be scheduled on the CPU. the process at front is dequeued and processed. New processes are added at the end of the queue.

• Round-robin scheduling: iterate through a set of processes in a circular manner and service each element: the process at front is dequeued, allowed to run for some CPU cycles, and then enqueued at the end of the queue

Stack

- *We know that the Stack is LIFO Structure i,e Last in First Out. It is very useful data structure in C Programming. Stack can be implemented using the Linked List or Array.*



Conditions of Stack

1. Stack is LIFO Structure ***[Last in First Out]***

2. Stack is Ordered List of ***Elements of Same Type****.*

3. Stack is Linear List

4. In Stack all Operations such as ***Insertion and Deletion*** *are permitted at only one end called* ***Top***

Building a Stack

- What data items do we need

 - An array/linked list to hold stack element

 - An integer/pointer to indicate the topof the stack

- What operations do we need

 - Constructor: build an empty stack

 - Empty: Check if the stack is empty

 - Push: Add an element on the top

 - Pop: Remove the top element

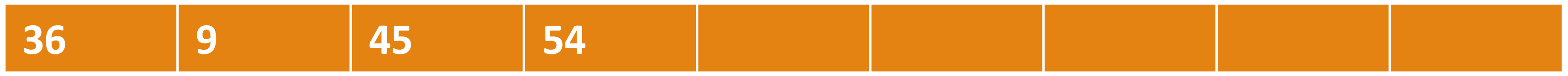
 - Top: return the top element

 - Display: show all the stack elements

Array implementation of Stack

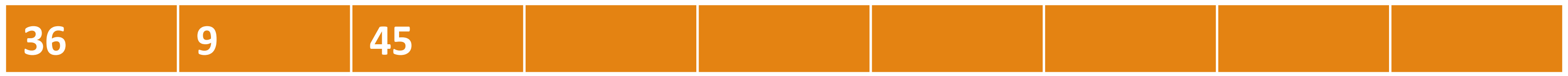
initial top = -1

Push



top=0 top=1 top=2 top=3

Pop



top=0 top=1 top=2

popped element: 54

Abstract Data Types (ADTs)

- An abstract data type (ADT) is a mathematically specified entity that defines a set of its instances.
- An ADT specifies:
- What data is stored
- Interface - Operations on the data
    - → manipulation
    - → access
- a set of axioms (preconditions and post conditions) that define what the operations do (not how)

Why ADTs

- ❑serve as specifications of requirements for the building blocks of an algorithm
- ❑encapsulate the data structure and the algorithms that works on the data structure
- ❑ separate issues of correctness and efficiency

Stack ADT

- Data Object
- Operations

new():ADT - creates a new stack

push(**S**:ADT, **o**:Object):ADT – Inserts object **o** onto the top of stack **S**

pop(**S**:ADT):ADT - removes the top object of stack **S**; if the stack is empty an error occurs

Stack ADT (Contd)

- Auxiliary operations

- **top**(**S**:ADT):Object - returns the top object of stack **S** without removing it; if the stack is empty, an error occurs
- **size**(**S**:ADT):integer returns the number of objects in stack **S**
- **isEmpty(S:**ADT):boolean - indicates if stack **S** is empty

- Axioms

- **pop**(**push**(S, **o**)) = **S**
- **top**(**push**(S, **o**)) = **o**

Stack Example

Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)
size( )	2	(5, 3)
pop( )	3	(5)
isEmpty( )	false	(5)
pop( )	5	( )
isEmpty( )	true	( )
pop( )	null	( )
push(7)	–	(7)
push(9)	–	(7, 9)
top( )	9	(7, 9)
push(4)	–	(7, 9, 4)
size( )	3	(7, 9, 4)
pop( )	4	(7, 9)
push(6)	–	(7, 9, 6)
push(8)	–	(7, 9, 6, 8)
pop( )	8	(7, 9, 6)

Example

Set: { 1, 2, 3 }

Stack: ||-><-

push(1)

Set: { 2, 3 }

Stack: | 1 |-><-

push(2)

Set: { 3 }

Stack: | 1 2 |-><-

push(3)

Set: {}

Stack: | 1 2 3 |-><-

pop()

returns 3

Stack: | 1 2 |-><-

pop()

returns 2

Stack: | 1 |-><-

pop()

returns 1

Stack: ||-><-

Stack Applications

- The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.
- Another application is an "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack.
- Arithmetic expression conversion and evaluation
- Language processing:
  - space for parameters and local variables is created internally using a stack.
  - compiler's syntax check for matching braces is implemented by using stack.
  - Support for recursion

Infix to Postfix Conversion

- We use a stack
- When an operand is read, output it
- When an operator is read
  - – Pop until the top of the stack has an element of lower precedence
  - – Then push it
- When) is found, pop until we find the matching (
- (has the lowest precedence when in the stack
- but has the highest precedence when in the input
- When we reach the end of input, pop until the stack is empt

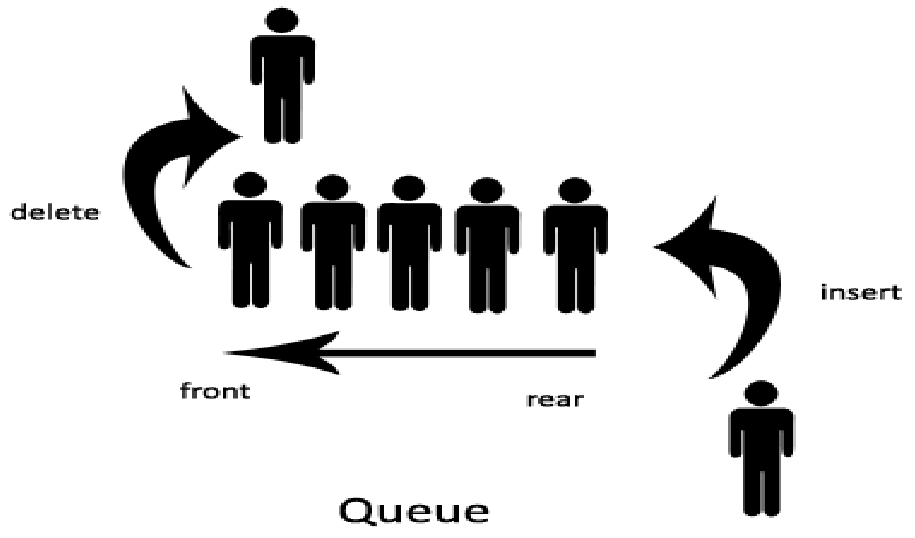
Example Infix to Postfix conversion

[Example]

[Evaluation of postfix expression]

Queue

A ***queue*** is a first in, first out (FIFO) structure or in the other sense, a last in, last out (LILO) structure. A queue is sometimes generalized as a structure where insertion (enqueue, pronounced N-Q) occur at one end and removal (dequeue, pronounced D-Q) occurs at the other end.



Queue

- The Queue is like the List but with “limited” insertion and deletion.
- Insertion can be done only at the end called as rear end
- Deletion can only be done in the front
- FIFO – first-in-first-out data structure
- Main Operations
  - enqueue
  - dequeue

Basic Operations on Queue

Primary queue operations: Enqueue and Dequeue

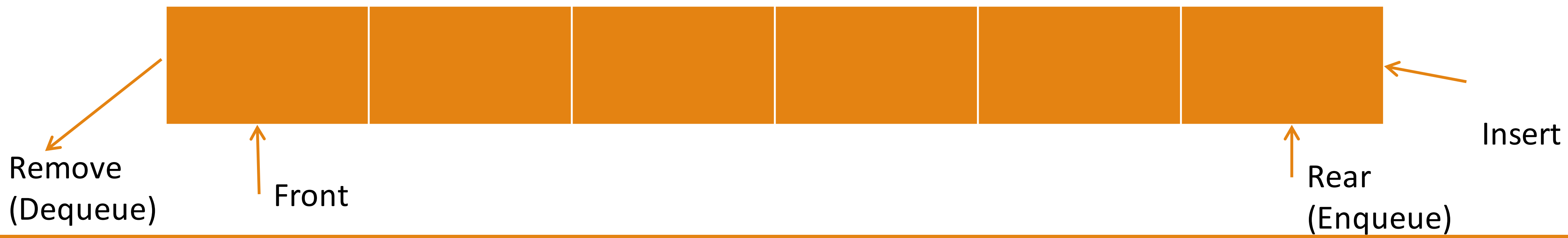
Like check-out lines in a store, a queue has a front and a rear.

Enqueue

- *Insert an element at the rear of the queue*

Dequeue

- *Remove an element from the front of the queue*



Queue ADT

The queue supports the following fundamental methods:

- ❑ **New**():ADT – creates an empty queue
- ❑ **Enqueue**(**Q**:ADT, **o**:element):ADT – inserts the object o at the rear of the queue
- ❑ **Dequeue**(**Q**:ADT):element – removes the object from the front of the queue; an error occurs if the queue is empty
- ❑ **First**(**Q**:ADT):element – returns, but does not remove the object at the front of the queue; an error occurs if the queue is empty.

Queue ADT (contd)

❑ Auxiliary methods

- **Size**(**Q**:QDT):integer – returns the number of elements in the queue
- **isEmpty**(**Q**:ADT):boolean – indicates whether the queue is empty

❑ Axioms

- Front(Enqueue(New(), v)) = v
- Dequeue(Enqueue(New(), v)) = New()
- Front(Enqueue(Enqueue(Q,w),v)) = Front(Enqueue(Q,w))
- Dequeue(Enqueue(Enqueue(Q,w), v)) = Enqueue(Dequeue(Enqueue(Q,w),v)

Example of Queue

Operation	***Output***	*Q*
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	*5*	(3)
enqueue(7)	–	(3, 7)
dequeue()	*3*	(7)
first()	*7*	(7)
dequeue()	*7*	()
dequeue()	*null*	()
isEmpty()	*true*	()
enqueue(9)	–	(9)
enqueue(7)	_	(9, 7)
size()	*2*	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()	*9*	(7, 3, 5)

Example of Queue

Set: { 1, 2, 3 }

Queue: <-||<-

enqueue(1)

Set: { 2, 3 }

Queue: <-| 1 |<-

enqueue(2)

Set: { 3 }

Queue: <-| 1 2 |<-

enqueue(3)

Set: { }

Queue: <-| 1 2 3 |<-

dequeue()

returns: 1

Queue: <-| 2 3 |<-

dequeue()

returns: 2

Queue: <-| 3 |<-

dequeue()

returns: 3

Queue: <-||<-

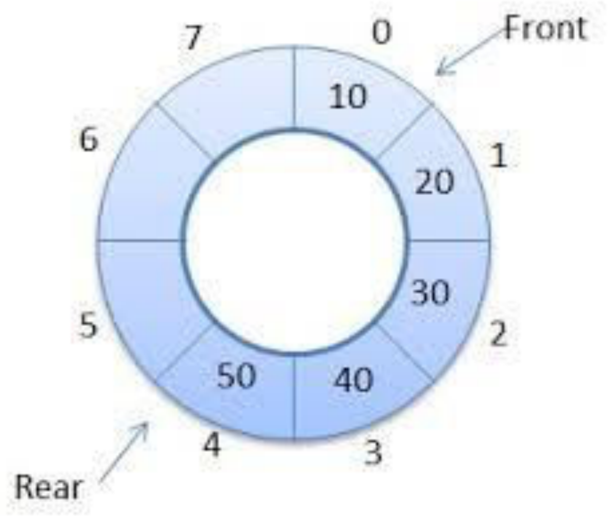
Practice problems

For the following sets, insert the elements into a queue and remove the elements again. Do the same for a stack. Be sure to indicate your arrows clearly.

1. Set = { 4, 7, 2, 8, 1, 5, 9 }
2. Set = {the, lazy, dog, jumped, over, the, fence }
3. Set = {aa, bb, cc, dd, ee}

Circular Queue

- The queue is reported as full (because the rear has reached the end of array (MAX limit)) even though there might be empty slots at the beginning of the queue
- To overcome above limitation, implement the queue as Circular Queue



Circular Queue

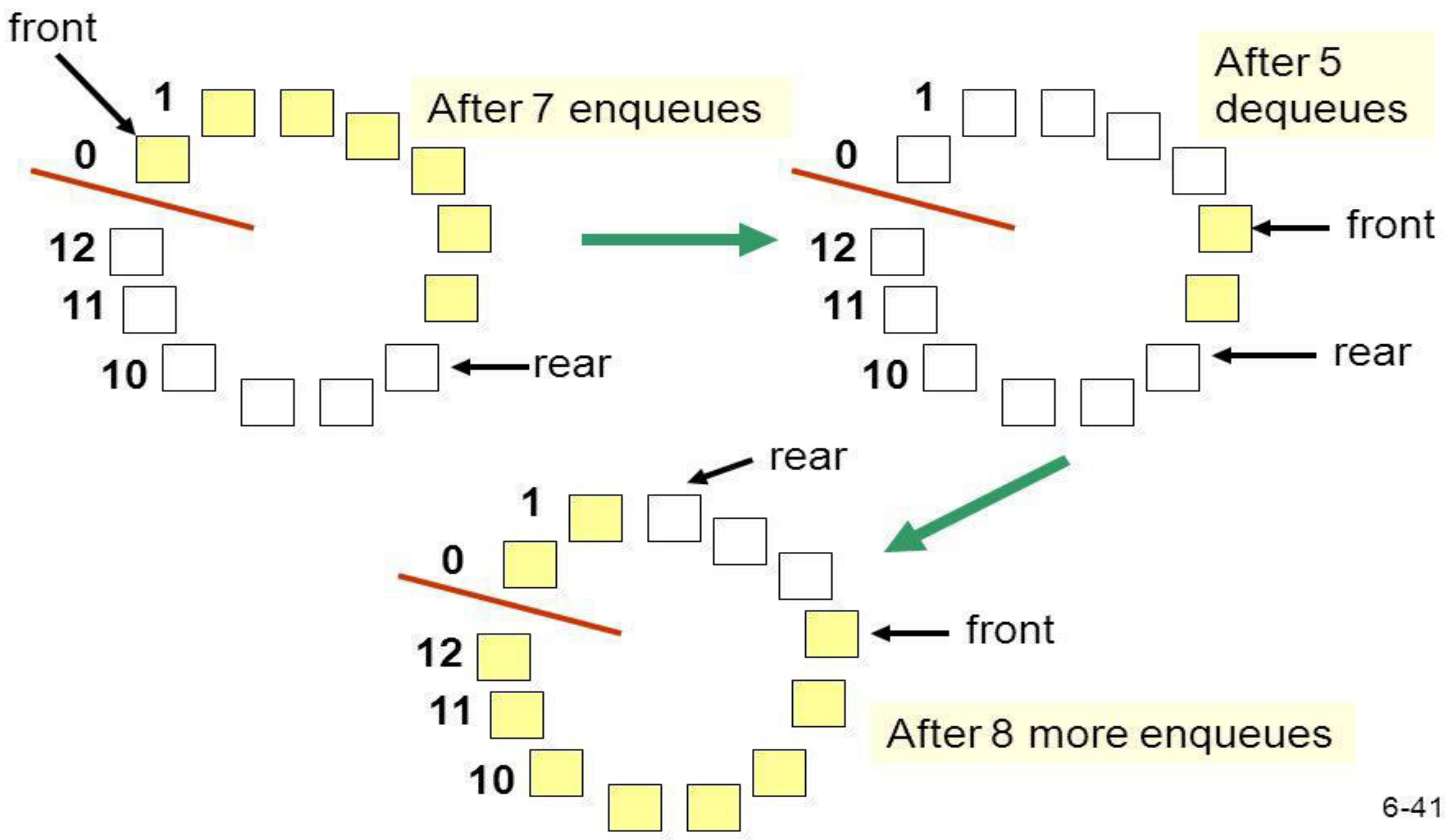
- Here as we go on adding elements to the queue and reach the end of the array, the next element is stored in the first slot of the array (provided it is free)
- simplest condition that can be used for checking Queue full is

```
If ((rear == MAX – 1 && front == 0) || (rear+1 = front)) then Queue is Full
else
if (rear == MAX -1)  rear = 0
 else rear++;
```

Circular Queue

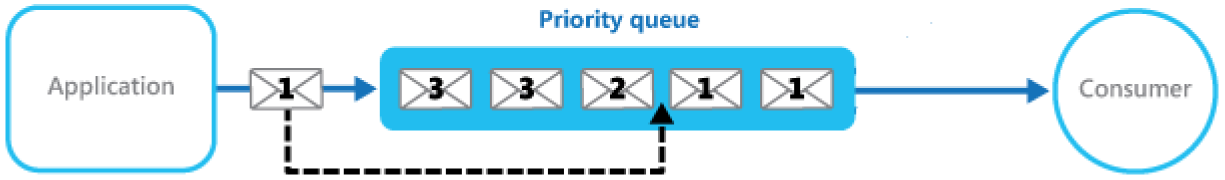
- ❑ Use an array of size ***N*** in a circular fashion
- ❑ Two variables keep track of the front and size
- ▪ ***f*** index of the front element
- ▪ ***sz*** number of stored elements
- ❑ When the queue has fewer than ***N*** elements, array location ***r* = (*f* + *sz*) *mod N*** is the first empty slot past the rear of the queue (**r – rear**)

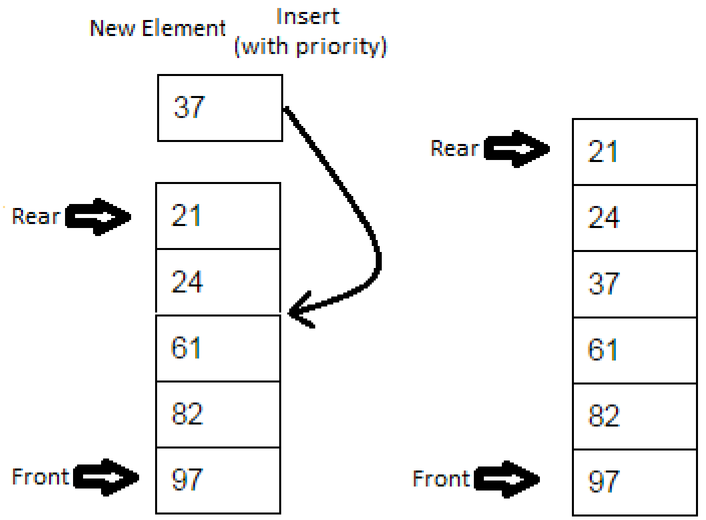
Conceptual Example of a Circular Queue



Priority Queue

- In computer science, a **priority queue** is an abstract data type similar to regular **queue** data structure in which each element additionally has a "**priority**" associated with it.
- In a **priority queue**, an element with high **priority** is served before an element with low **priority**





Priority Queue

- Priority Queue is an extension of queue with following properties.
- Every item has a priority associated with it.

1. An element with high priority is dequeued before an element with low priority.
2. If two elements have the same priority, they are served according to their order in the queue
3. In the below priority queue, element with maximum ASCII value will have the highest priority.

In the below priority queue, element with maximum ASCII value will have the highest priority.

Priority Queue Example

Initial Queue = {}

Operation	Return Value	Queue Content
Insert (C)		C
Insert (O)		C O
Insert(D)		C O D
removemax	O	C D
Insert(I)		C D I
Insert(N)		C D I N
removemax	N	C D I
Insert(G)		C D I G

Implementation

- A typical priority queue supports following operations.

insert(item, priority): Inserts an item with given priority.
getHighestPriority(): Returns the highest priority item.
deleteHighestPriority(): Removes the highest priority item.

How to implement priority queue?
Using Array: A simple implementation is to use array of following structure.

```
Struct item

{

int item;

int priority;

}
```

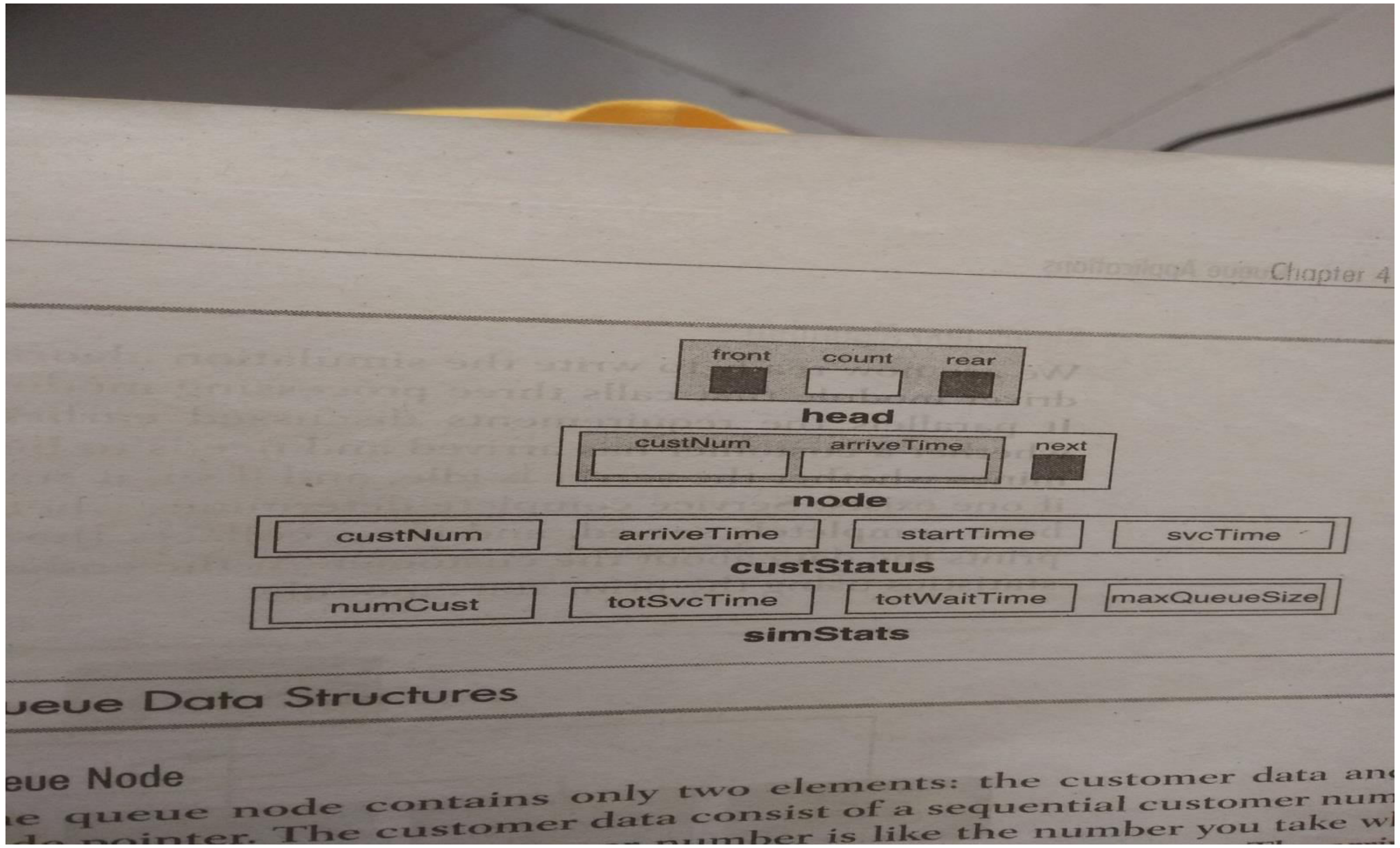
Applications of Queue

1. Categorizing data
2. Simulation of queue.

Queue Simulation

- Simulation enables the study of, and experimentation with, the interactions of a complex system (or a subsystem thereof).
- Informational, organisational, and environmental changes can be simulated and the changes to the model's behaviour can be observed.
- Knowledge gained by studying the model can be of great value, and may suggest improvement to the system under investigation.
- Changing simulation inputs can help understanding how certain components of the system interact. Simulation is, among others, used to experiment with new designs or policies prior to implementation, and may save huge amounts of money.
- Simulations are also used to verify analytical solutions.

Application(page 175 onwards from T2)



ueue Data Structures

eue Node

e queue node contains only two elements: the customer data an

de pointer. The customer data consist of a sequential customer num

Data Structures

Suggestions are Welcome!