## Module 1:

# Introduction to Stacks, Queues and Linked Lists
# CO1 (8 hours)

- Introduction to Data Structures: Linear and Non Linear Data Structures, Static and Dynamic Data Structures.
- Concept of Stack and Queue.
  - ❑ Array Implementation of Stack and Queue,
  - ❑ Circular Queue,
  - ❑ Double Ended Queue,
  - ❑ Priority Queue.
- Concept of Linked Lists.
  - ❑ Singly linked lists,
  - ❑ doubly linked lists and
  - ❑ circular linked lists.
- Insertion, deletion, update and copying operations with Singly linked lists, doubly linked lists and circular linked lists.
- Reversing a singly linked list.

**Self-learning Topics:** Linked List Implementation of Stack, Linked List implementation of Queue, Circular Queue, Double Ended Queue, Priority Queue.]

| Sr. No. | Course Outcomes | Cognitive levels of attainment as per Bloom's Taxonomy | PO Attainment | PSO attainment |
|---|---|---|---|---|
| | On successful completion, of course, learner/student will be able to: | | | |
| **1** | **Understand the Basic concept of Abstract Data types and Classify and Apply the concepts of stacks, queues and linked list in real life problem solving.** | **L1, L2, L3** | **PO1,PO2** | **PSO1** |
| **2** | **Classify, apply and analyze** the concepts trees in real life problem solving. | L2, L3,L4 | PO1,PO2,PO3 | PSO1 |
| **3** | **Illustrate and justify** the concepts of graphs in real life problem solving | L2, L3, L5 | PO2,PO3 | PSO1 |
| **4** | **List, explain and examine** the concepts of sorting, searching techniques in real life problem solving | L1, L2, L4 | PO2,PO3 | PSO1 |
| **5** | **Use and identify** the concepts of recursion, hashing in real life problem solving. | L2, L3 | PO2,PO3 | PSO1 |
| **6** | **Examine and justify** different methods of stacks, queues, linked list, trees and graphs to various applications. | L4, L5 | PO4,PO5 | PSO1 |

# Which is more organized ?

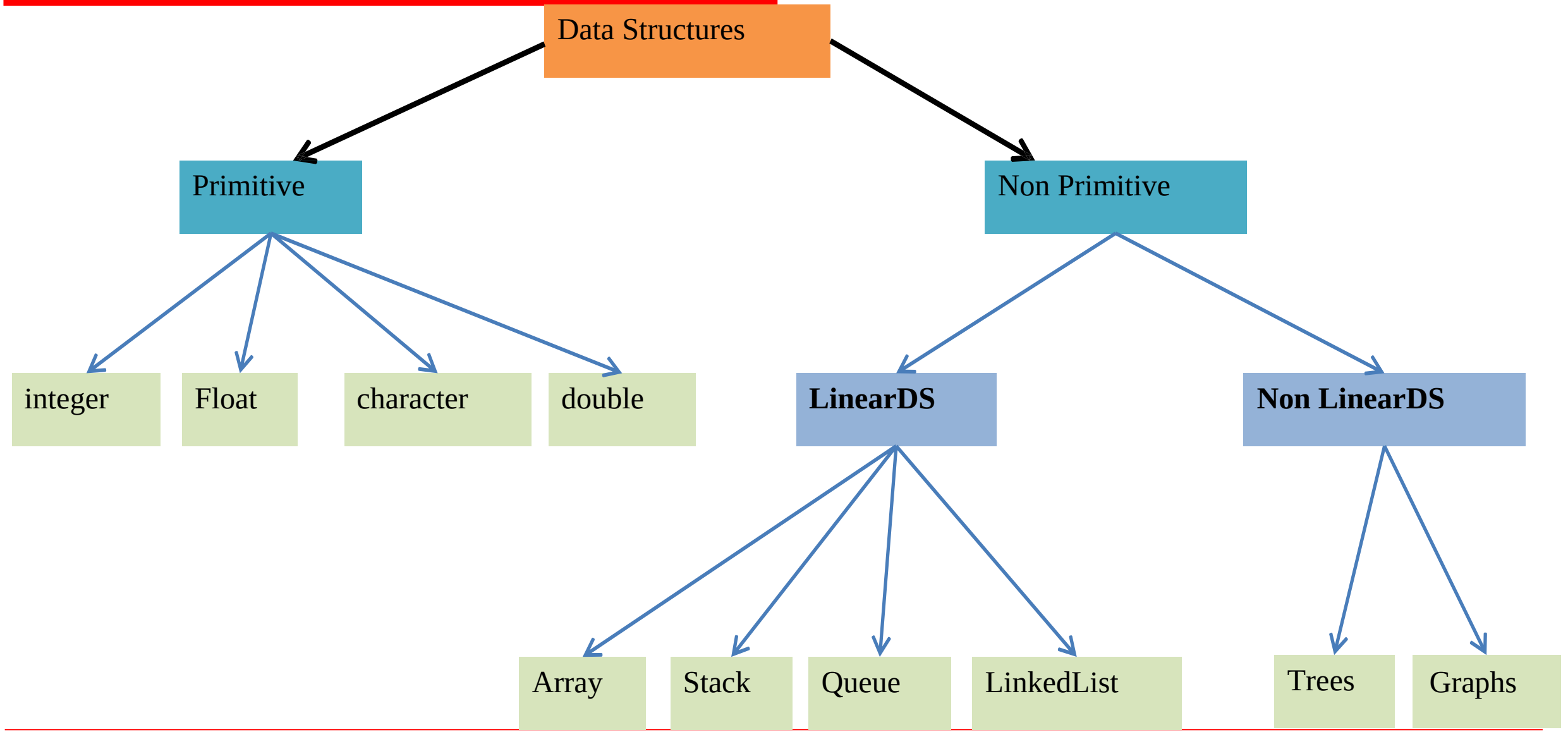
Figure 1

Figure 2

What is a Data structure?

Data Structure is a set of algorithms which are used to structure the information while storing.

- Formal definition is data structure is a way of storing and organizing data in a computer so that it can be used and accessed efficiently.



## 1.Linear data structure

All the elements are accessed in sequential order or linear order means that all the elements are adjacent to each other. Each element has exactly two neighbor-. Predecessor and successor.

- the first element does not have predecessor
- The last element does not have successor.

## 2. Non linear data structure

-no such sequence in elements, if one element is connected to a more than two adjacent element then it is a non linear data structure.

| Linear data Structures | Non Linear Data Structures |
|---|---|
| 1. Data items arranged in sequence | 1. Not in sequence |
| 2. All elements are adjacent to each other | 2. One element adjacent to more than one element |
| 3. Easy implementation | 3. Difficult to implement |
| 4. All elements can access in single run | 4. Not possible with single run |
| 5. array, stack, queues, linked list | 5. Trees and graphs |

# Static Data structures and dynamic data structures

| Static Data Structures | Dynamic Data Structures |
|---|---|
| Size is fixed | Size can vary during runtime |
| Possible to change contents of data structure without changing memory space | Possible to change contents of data structure and also changing memory space |
| Faster access to elements | Slower access to elements |
| Add, remove and modify is not directly possible | Ability to add, remove and delete elements efficiently |
| Resource allocation at creation of data structures though the data structure may be empty | Effective use of resources as they are allocated during runtime |

# Operations on Data structures

**Insertion** — It is used to add new data items to the given list of data items. For example, to add the details of a new student who has recently joined the course

**Deletion** — It means to remove (delete) a particular data item from the given collection of data items. For example, to delete the name of a student who has left the course

**Searching** — It is used to find the location of one or more data items that satisfy the given constraint.

**Traversal** — It means to access each data item exactly once so that it can be processed. For example, to print the names of all the students in a class

**Sorting** — Data items can be arranged in some order like ascending order or descending order depending on the type of application. For example, arranging the names of students in a class in an alphabetical order, or calculating the top three winners by arranging the participants' scores in descending order and then extracting the top three.

**Merging** — Lists of two sorted data items can be combined to form a single list of sorted data items.

- Introduction to Data Structures: Linear and Non Linear Data Structures, Static and Dynamic Data Structures.
- **Concept of Stack and Queue.**

**Array Implementation of Stack and Queue,**

**Circular Queue,**

**Double Ended Queue,**

**Priority Queue.**

# ❑ Concept of Linked Lists.

- Singly linked lists,
- doubly linked lists and
- circular linked lists.
- Insertion, deletion, update and copying operations with Singly linked lists, doubly linked lists and circular linked lists.
- Reversing a singly linked list.

## ❑ Self-learning Topics:

❑ Linked List Implementation of Stack, Linked List implementation of Queue, Circular Queue, Double Ended Queue, Priority Queue.

# Stacks

It is a linear, abstract data structure

**Definition - A stack is an ordered collection of homogeneous data elements, where the insertion and deletion takes place at one end, known as TOP.**

- The stack is also called LAST IN FIRST OUT(LIFO)
- It means: the element which is inserted last must be deleted first
- Real world Examples

1.pile of plates in cafeteria

2.stack of coins

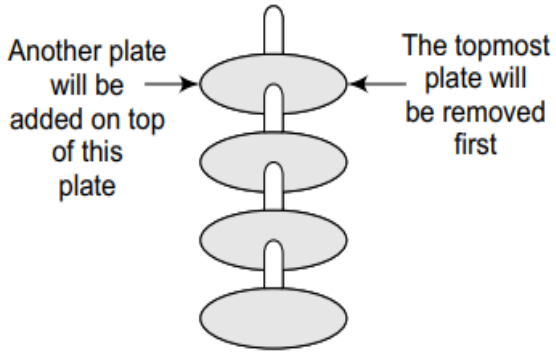
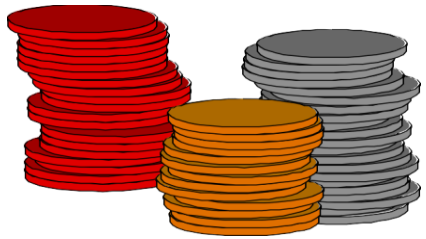
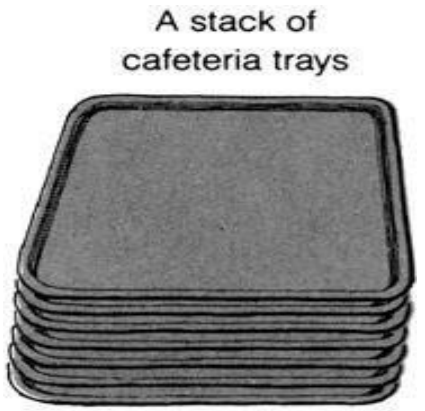



Figure 7.1 Stack of plates





- Stack maintains a pointer called top, which keeps track of the top most element in the stack.
- Any insertions or deletions should be based upon the value of top.
- It works on the basis of LIFO (Last in First out).
- According to the definition, new elements are inserted from top and the elements are deleted from same end i.e again top.
- This suggests that the element inserted most recently can only be deleted.
- In the stack, the elements are removed in the reverse order of that in which they were added to the stack i.e last element inserted is to be deleted first.
- Stack has structured with two operations

1. push- insertion-adding elements on to a stack
2. Pop- deletion -removing element from the stack

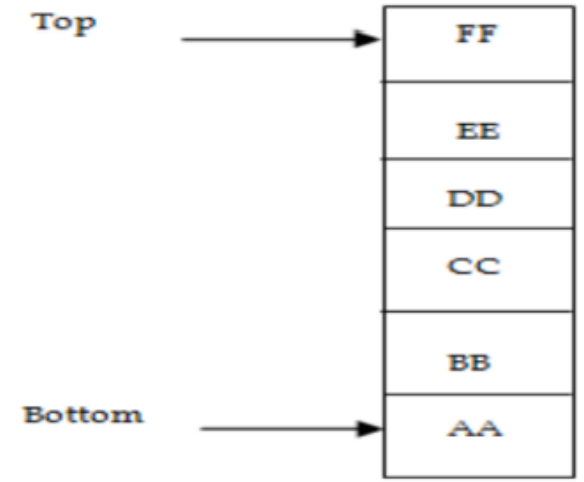


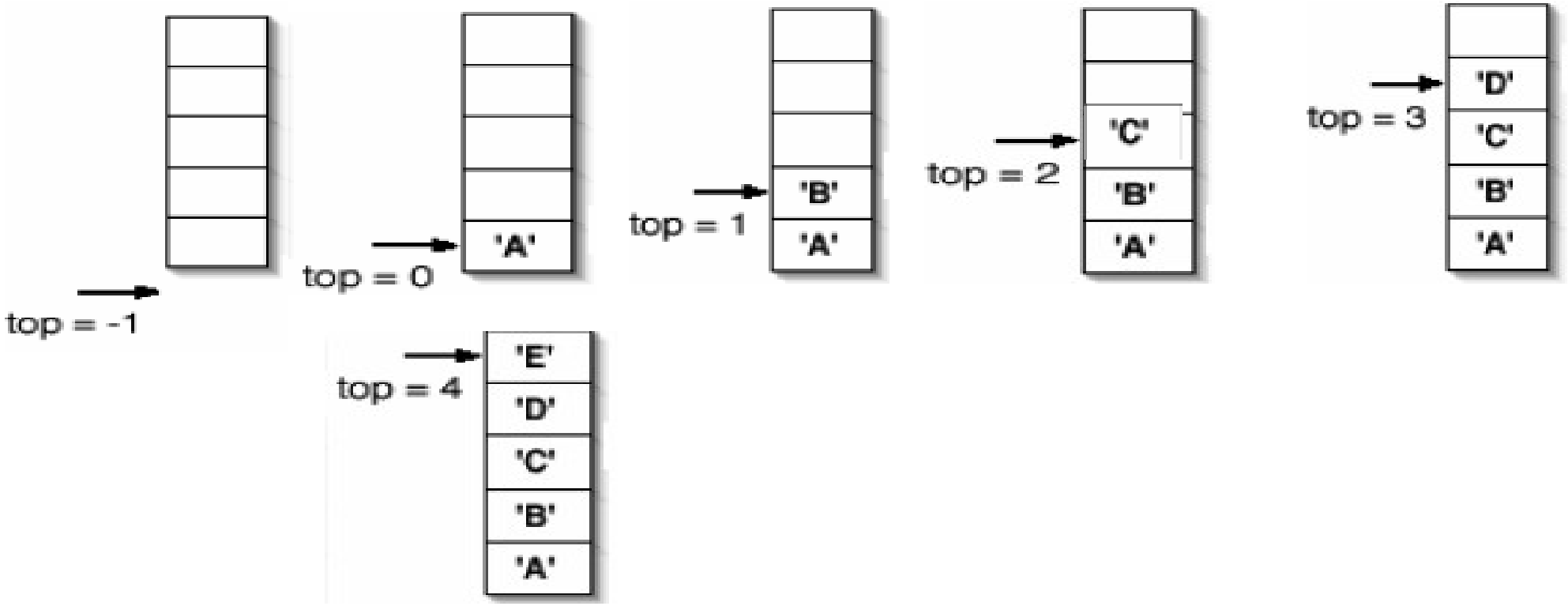

**Figure 1.4: A Stack**

# Basic operations:

- The basic or primary operations are insertion, deletion ,display.
- In stacks, special terms are given for insert and delete. i.e push for insert and pop is for delete.
- Push: inserting or adding element into the stack is called push.
- Pop: deleting or removing element from the stack is called pop.
- Peek : Returns the topmost element of the stack without deleting

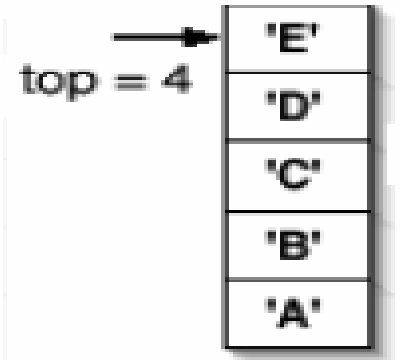
## PUSH Operation

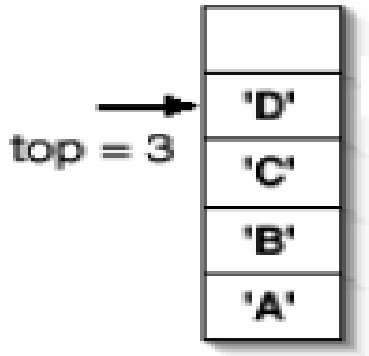
- Elements are inserted in the order as A,B,C,D,E
- It represents the stack of 5 elements.

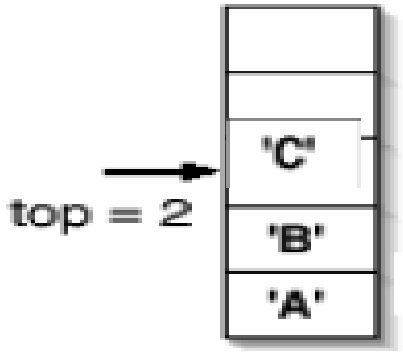


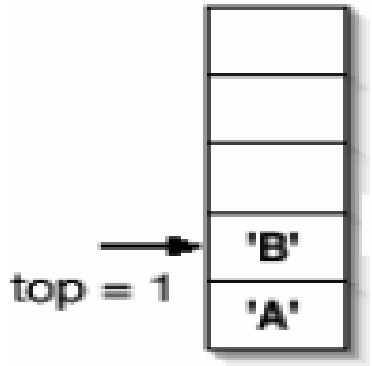

- The top most element in the stack is E
- If we want to delete element E has to be deleted first

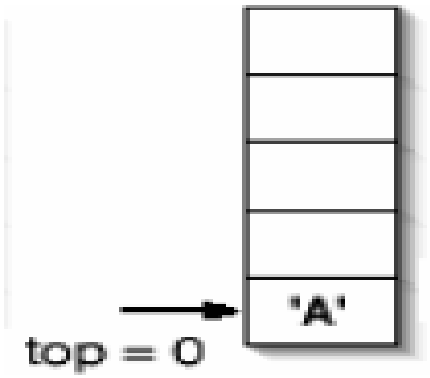
## ❖ Pop operation- delete element from the stack

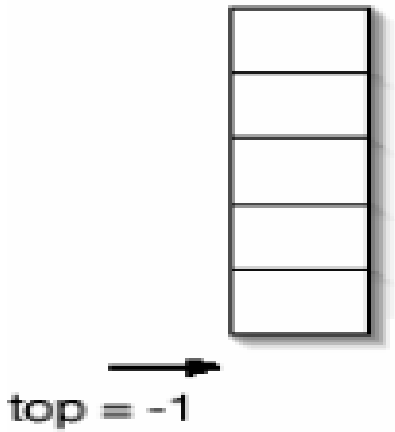


# Example :-

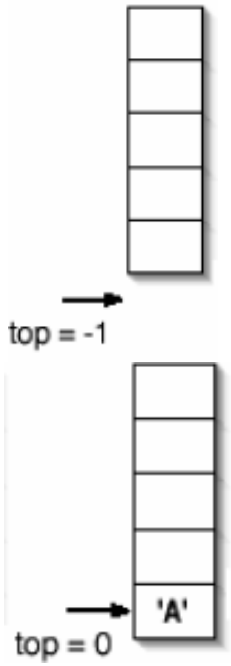
```
...

push('A');
push('B');
push('D');
pop();
push('C');
push('D');
push('E');
pop();
pop();

...
```

1. **int stack[5]**; Fixed size stack of size 5
2. **Top = -1**; pointer points to latest element in stack,initially -1 when stack is empty
3. **void push();** Inserting elements into stack after incrementing top by 1
4. **void pop();** Deleting element from stack and then decrementing top by 1
5. **Void peek();** Returing topmost element with the help of TOP
6. **void display();** show elements
7. **void isEmpty();** check for underflow
8. **void isFull();** Check for Overflow

# Arrays Implementation of Stack

Algorithm for inserting element into the stack:

Algorithm push()
1.if top=(SIZE-1)
then write ('stack overflow')
else
2.read item or data
3.top←top+1
4.stack[top]← item
5.stop




- The stack is of size max. This procedure inserts an element item on to the top of a stack which is represented by an array stack.
- The first step of this algorithm checks for an overflow condition.
- Overflow means inserting element into a stack which is full.
- If the top value reaches to maximum capacity of the stack then elements cannot be inserted into the stack i.e. stack is full.
- Otherwise top is incremented by one and element is inserted into the stack.

Algorithm to delete elements from the stack:

Algorithm pop()

1. if top=-1
   then write ('stack underflow')
   else
2. item ← stack[top]
3. top ← top-1

Explanation:

- This procedure deletes an element from the stack.
- The first step of this algorithm checks for underflow condition.
- If the top value is -1 then stack is empty.
- Empty stack is known as underflow.
- Takeout the element from the location where, the top is pointing and then decrement top by one.

# Display of stack:

- Printing the contents of stack after push and pop operations.

```
Algorithm print()
1.if top=-1
        then write ('stack empty')
  1.Repeat for i ← top to 0
        print(stack[i])
1.stop
```

- Limitation in stack using array is that maximum size of the stack must be pre defined and it cannot be changed(fixed).
- When trying to add elements in the stack when the stack is full will rise the exception.

# Applications of stacks

- Implementing function calls(Recursion)
- Parenthesis matching.
- Evaluation of postfix expressions.
- Infix to prefix conversions.
- Infix to postfix conversions.
- Web browser history
- Undo operations in text editors
- Matching tags in HTML and XML
- Quick sort.
- Balancing symbols

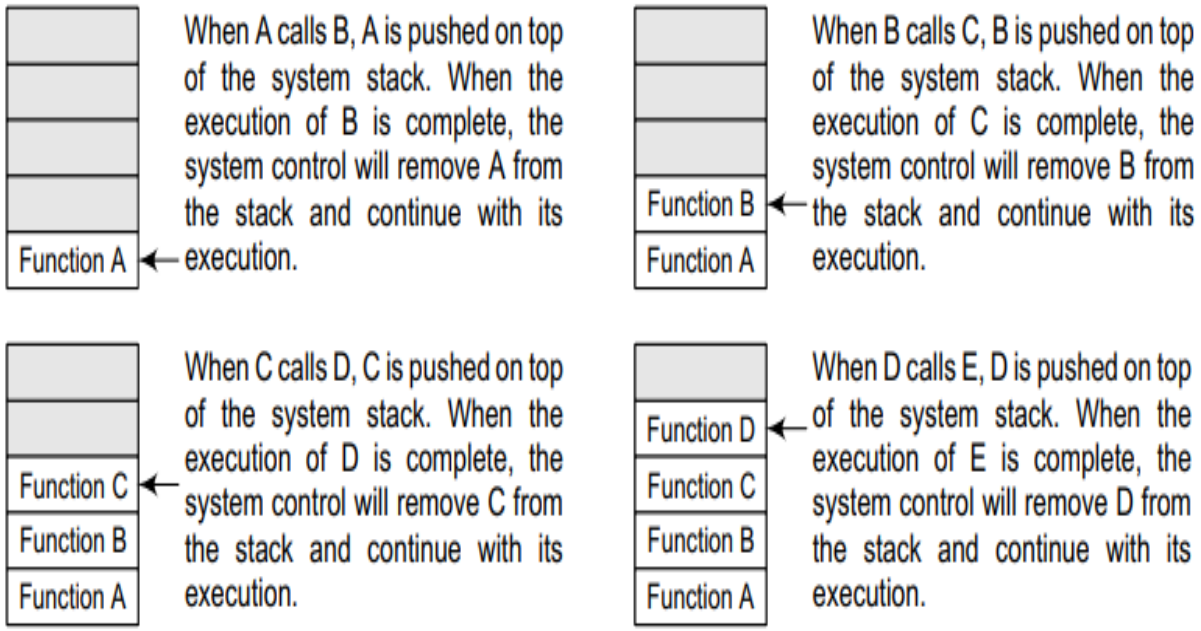



**Figure 7.2** System stack in the case of function calls

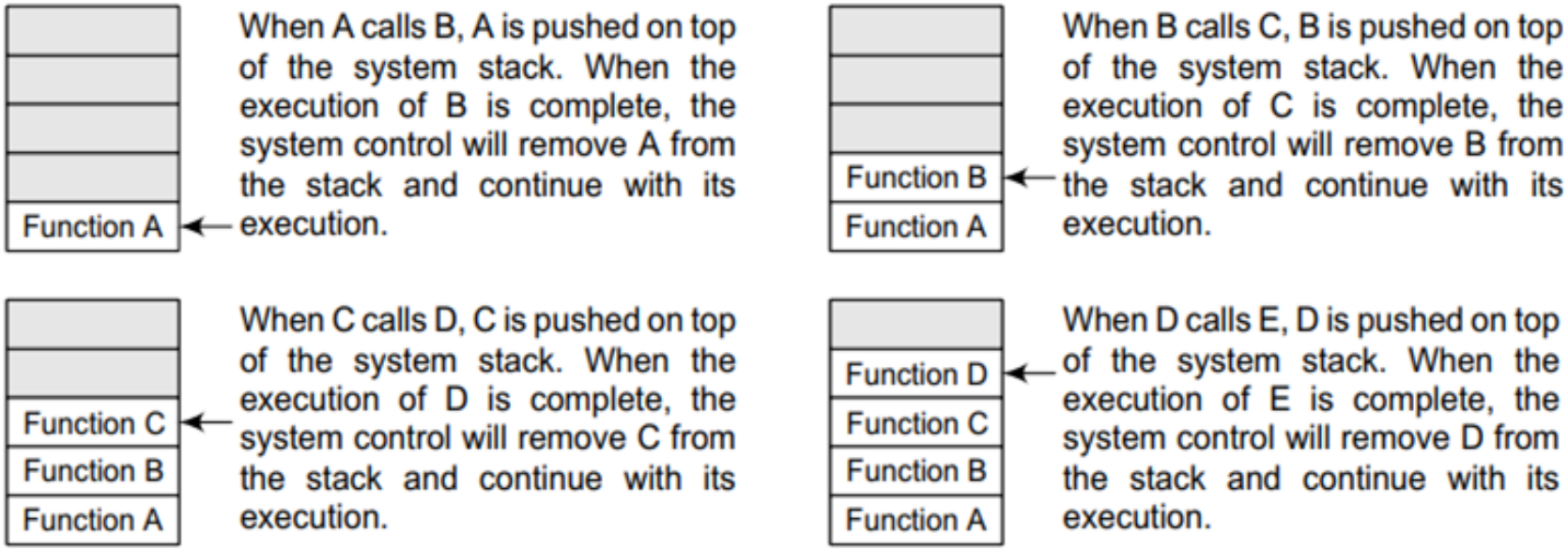



**Figure 7.2** System stack in the case of function calls

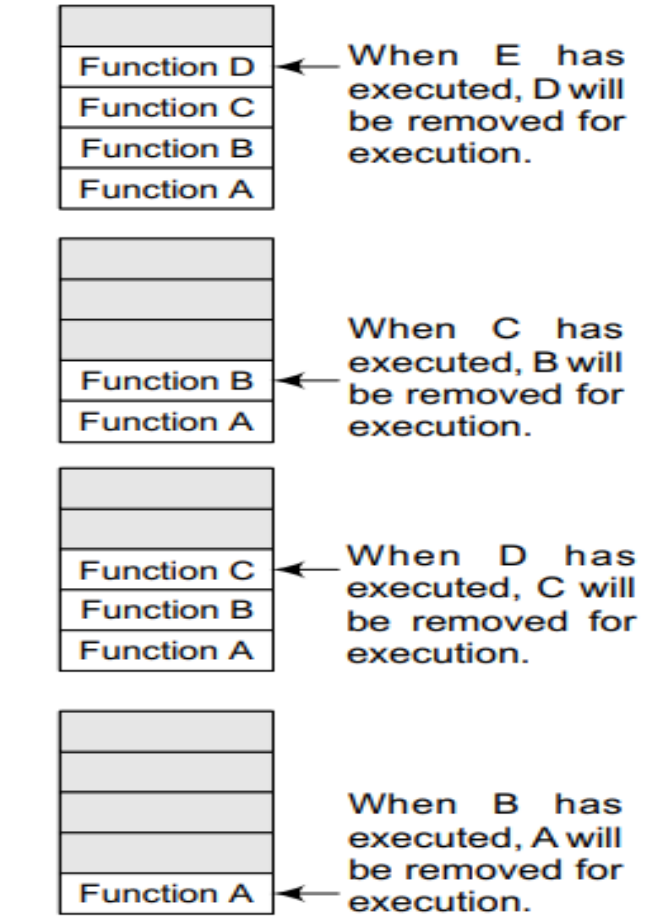



**Figure 7.3** System stack when a called function returns to the calling function

# Applications of stacks-Parenthesis matching

- The objective of this function is to check the matching of parenthesis in an expression
- i.e In an expression the number of left parenthesis must be equal to number of right parenthesis.

- Ex: ((A+B)*C)
- This is a valid expression,because in this number of left parenthesis (2) = number of right parenthesis (2).

**Review Question on Data structure Stacks:**

1. **What do you understand by stack overflow and underflow?**
2. **Stack is a (a) LIFO (b) FIFO**
3. **Write a function that accepts two stacks. Copy the contents of first stack in the second stack. Note that the order of elements must be preserved. (Hint: use a temporary stack)**
4. **Which function places an element on the stack? (a) Pop() (b) Push() (c) Peek() (d) isEmpty()**
5. **When an Element is Placed on to the stack , the top is incremented/decremented/remains unchanged**

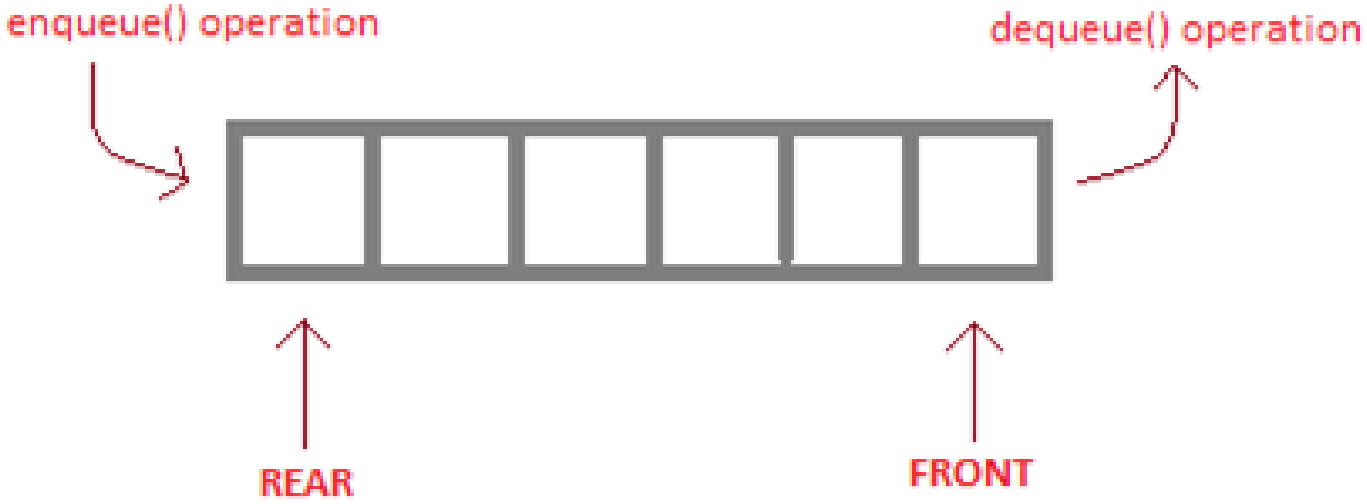
**University Questions :**

Explain the working of stack with its operations: push, pop, peek, display, empty , full. Proper diagrammatic representations of operations as mentioned above, are also expected. Also, write two applications (algorithms) where stack data structure is used.

# Concept of Queue

- A linear and Abstract Data types
- A queue is a first-in, first-out **(FIFO) data structure** in which the element that is inserted first is the first one to be taken out.
- The elements in a queue are added at one end called the **rear** and removed from the other end called the **front**.
- Like stacks, queues can be implemented by using either arrays or linked lists.

The process to add an element into queue is called **Enqueue**
and the process of removal of an element from queue is called **Dequeue**




**enqueue( )** is the operation for adding an element into Queue.

**dequeue( )** is the operation for removing an element from Queue .

**QUEUE DATA STRUCTURE**

Real life Scenarios:

- Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
- In real life scenario, Call Centre phone systems uses Queues to hold people calling them in an order, until a service representative is free.
- Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.

# Implementation of Queue Data Structure

- Queue can be implemented using an Array or Linked List.
- A queue is a first-in, first-out (FIFO) data structure in which the element that is inserted first is the first one to be taken out.
- The easiest way of implementing a queue is by using an Array.
- Initially the head(FRONT) and the tail(REAR) of the queue points at the first index of the array (starting the index of array from 0).
- As we add elements to the queue, the tail keeps on moving ahead, always pointing to the position where the next element will be inserted, while the head remains at the first index.

# Queue as an ADT

- Create() : creates and initializes new queue that is empty. Does not require any parameter and return empty queue
- Enqueue(item) : Adds new element to the rear of the queue. It requires the element to be added and return nothing
- Dequeue() : removes element from front end of the queue, does not require any parameter and return the deleted items
- isEmpty() : checks whether the queue is empty or not. Return a Boolean value
- isFull() : checks whether the queue is full or not, return Boolean value
- Size(): return total number of elements present in the queue, returns an integer

# Difference between stack and queue?

| Parameter | Stack | Queue |
|---|---|---|
| Operation end | Elements are added and deleted from same end | Elements are added and deleted from different ends |
| Pointer | Single pointer to point top of stack | Two pointers to point two ends of the queue |
| order | LIFO | FIFO |
| Operation names | PUSH() , POP() | Enqueue(), Dequeue() |
| visualization | Vertical collection | Horizontal collection |
| examples | Stack of books, tower of hanoi | Queue at ticket counter, jobs at scheduler |

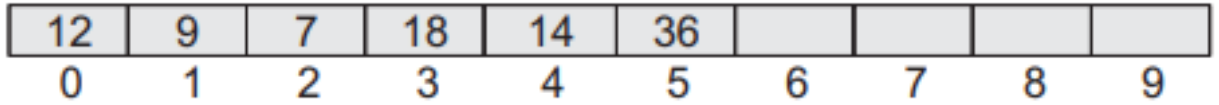



**Figure 8.1** Queue

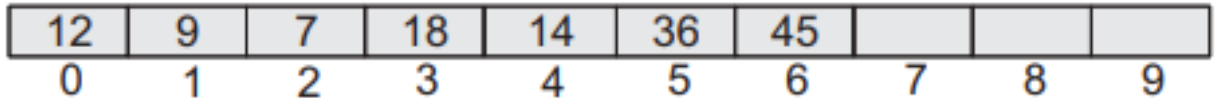



**Figure 8.2** Queue after insertion of a new element

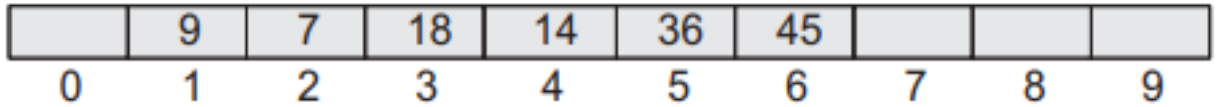



**Figure 8.3** Queue after deletion of an element

## *Operations on Queues*

In Fig. 8.1, FRONT = 0 and REAR = 5. Suppose we want to add another element with value 45, then REAR would be incremented by 1 and the value would be stored at the position pointed by REAR. The queue after addition would be as shown in Fig. 8.2. Here, FRONT = 0 and REAR = 6. Every time a new element has to be added, we repeat the same procedure.

If we want to delete an element from the queue, then the value of FRONT will be incremented. Deletions are done from only this end of the queue. The queue after deletion will be as shown in Fig. 8.3.

Here, FRONT = 1 and REAR = 6.

# Enqueue

```
Step 1: IF REAR = MAX-1
            Write OVERFLOW
            Goto step 4
        [END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
            SET FRONT = REAR = 0
        ELSE
            SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: EXIT
```

**Figure 8.4** Algorithm to insert an element in a queue

## Algorithm for ENQUEUE operation

- Check if the queue is full or not.
- If the queue is full, then print overflow error and exit the program.
- If the queue is not full, then check if front and rear both are -1 means queue is empty, thus set front and rear = 0 and add element at position pointed by rear
- Else if both front or rear are not 0 then increment rear to 1 and add element to position pointed by rear

# Dequeue

## Algorithm for DEQUEUE operation:

- Check if the queue is empty or not.
- If the queue is empty, then print underflow error and exit the program.
- If the queue is not empty, then print the element at the head position and increment the head.

```
Step 1: IF FRONT = -1 OR FRONT > REAR
            Write UNDERFLOW
        ELSE
            SET VAL = QUEUE[FRONT]
            SET FRONT = FRONT + 1
        [END OF IF]
Step 2: EXIT
```

**Figure 8.5** Algorithm to delete an element from a queue

# Types of Queues :

- Circular Queue,
- Double Ended Queue,
- Priority Queue.

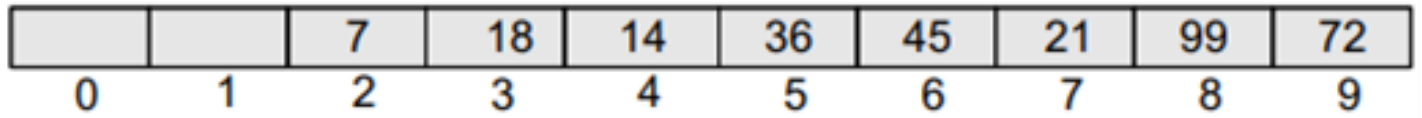



**Figure 8.14** Queue after two successive deletions

Suppose we want to insert a new element in the queue shown in Fig. 8.14.

Even though there is space available, the overflow condition still exists because the condition rear = MAX – 1 still holds true.

**This is a major drawback of a linear queue**.

# ❑ Circular Queue

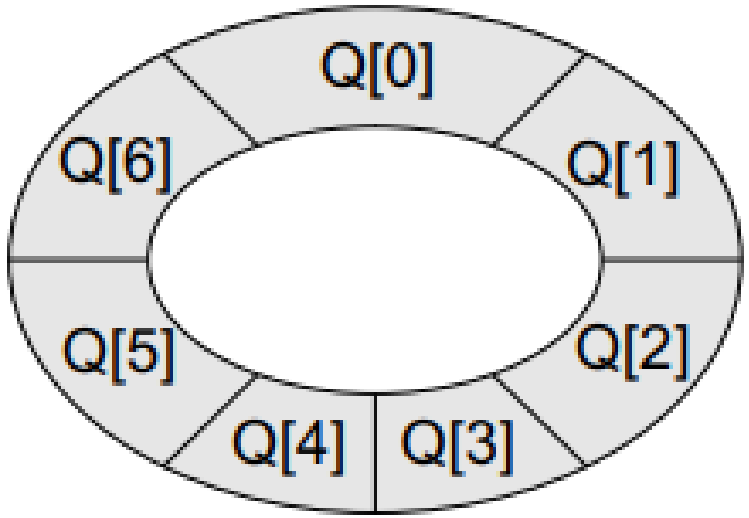



**Figure 8.15** Circular queue

To resolve the problem of linear queues, we have two solutions.

First, shift the elements to the left so that the vacant space can be occupied and utilized efficiently.
But this can be very time-consuming, especially when the queue is quite large.

The second option is to use a circular queue.

In the circular queue, the first index comes right after the last index. Conceptually, you can think of a circular queue as shown in Fig. 8.15

# ❑ Circular Queue

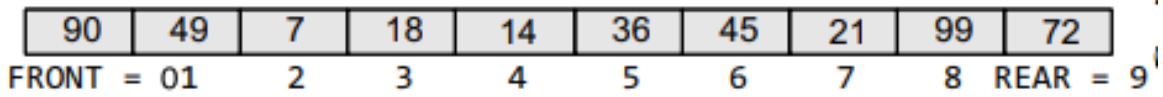



**Figure 8.16** Full queue

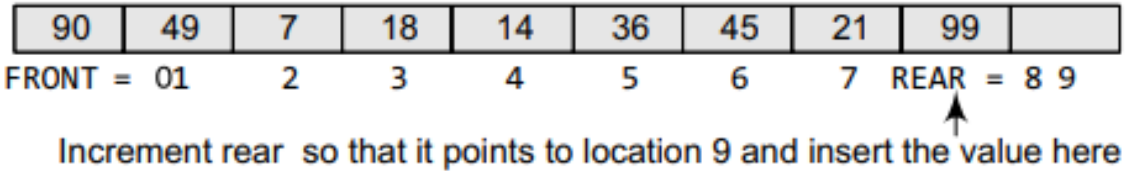



**Figure 8.17** Queue with vacant locations

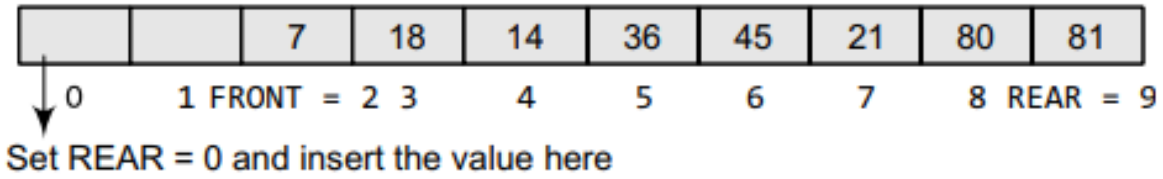



**Figure 8.18** Inserting an element in a circular queue

- If front = 0 and rear = MAX – 1, then the circular queue is full. Look at the queue given in Fig. 8.16 which illustrates this point.
- If rear != MAX – 1, then rear will be incremented and the value will be inserted as illustrated in Fig. 8.17.
- If front != 0 and rear = MAX – 1, then it means that the queue is not full. So, set rear = 0 and insert the new element there, as,shown in Fig. 8.18

# ❑ Circular Queue (enqueue)

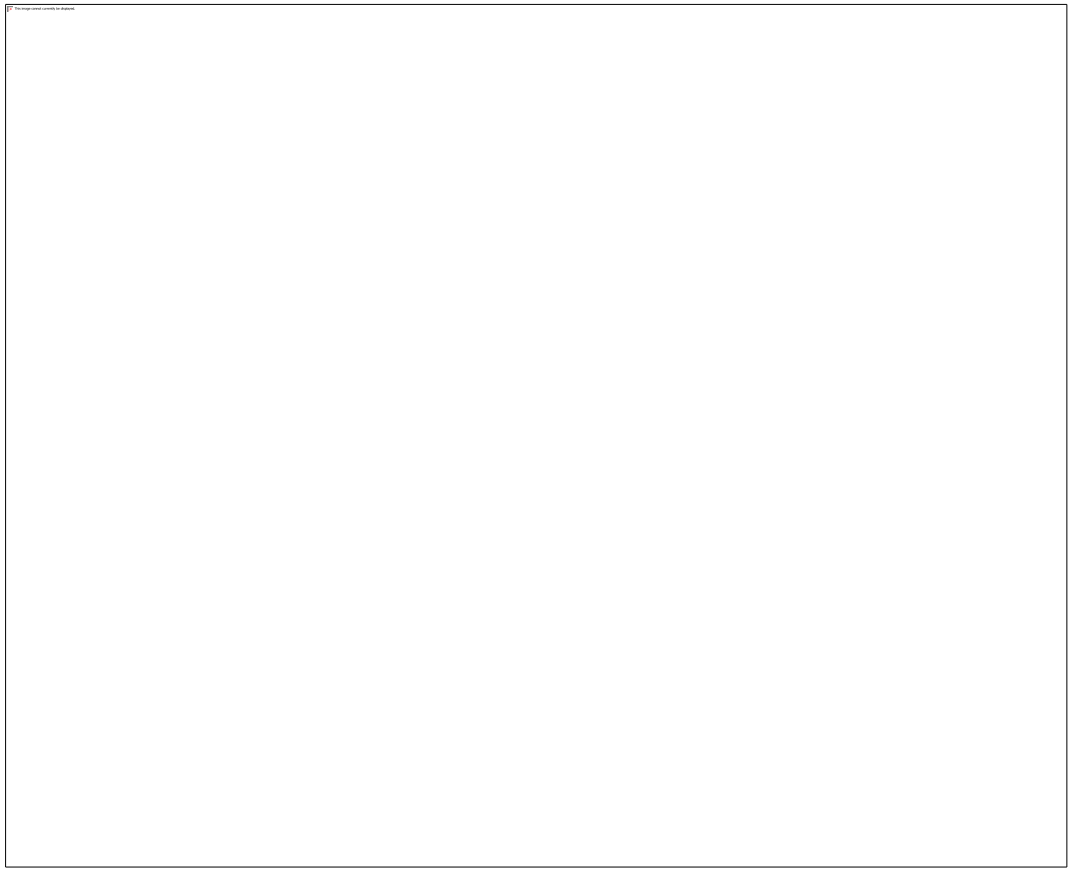
Limitation Of Circular queue

```
Step 1: IF FRONT = 0 and Rear = MAX - 1
            Write "OVERFLOW"
            Goto step 4
        [End OF IF]
Step 2: IF FRONT = -1 and REAR = -1
            SET FRONT = REAR = 0
        ELSE IF REAR = MAX - 1 and FRONT != 0
            SET REAR = 0
        ELSE
            SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: EXIT
```

**Figure 8.19** Algorithm to insert an element in a circular queue

- If front = 0 and rear = MAX – 1, then the circular queue is full. Look at the queue given in Fig. 8.16 which illustrates this point.
- If rear != MAX – 1, then rear will be incremented and the value will be inserted as illustrated in Fig. 8.17.
- If front != 0 and rear = MAX – 1, then it means that the queue is not full. So, set rear = 0 and insert the new element there, as shown in Fig. 8.18

# ❑ Circular Queue(dequeue)



- Look at Fig. 8.20. If FRONT = -1, then there are no elements in the queue. So, an underflow condition will be reported.
- If the queue is not empty and FRONT = REAR, then after deleting the element at the front the queue becomes empty and so FRONT and REAR are set to -1. This is illustrated in Fig. 8.21.
- If the queue is not empty and FRONT = MAX-1, then after deleting the element at the front, FRONT is set to 0. This is shown in Fig. 8.22.

# ❑ Circular Queue

```
Step 1: IF FRONT = -1
            Write "UNDERFLOW"
            Goto Step 4
        [END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
            SET FRONT = REAR = -1
        ELSE
            IF FRONT = MAX -1
                  SET FRONT = 0
            ELSE
                  SET FRONT = FRONT + 1
            [END of IF]
        [END OF IF]
Step 4: EXIT
```

**Figure 8.23** Algorithm to delete an element from a circular queue

- Look at Fig. 8.20. If FRONT = -1, then there are no elements in the queue. So, an underflow condition will be reported.
- If the queue is not empty and FRONT = REAR, then after deleting the element at the front the queue becomes empty and so FRONT and REAR are set to -1. This is illustrated in Fig. 8.21.
- If the queue is not empty and FRONT = MAX-1, then after deleting the element at the front, FRONT is set to 0. This is shown in Fig. 8.22.

# ❑ Double Ended Queue (DE-Queue)

is a list in which the elements can be inserted or deleted at either end. It is also known as a head-tail linked list because elements can be added to or removed from either the front (head) or the back (tail) end

However, no element can be added and deleted from the middle.

# ❑ Double Ended Queue (DE-Queue)

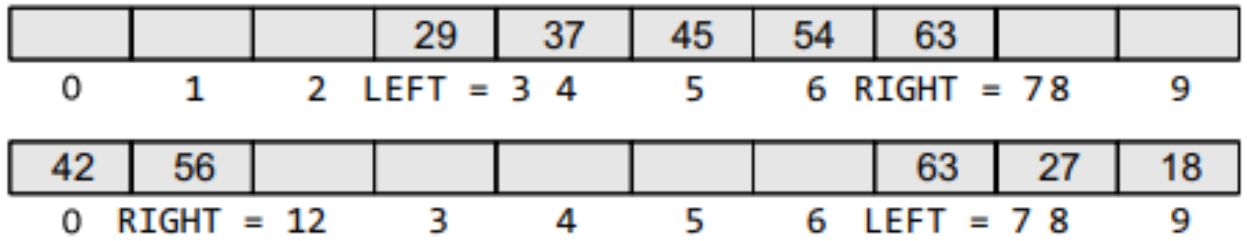



**Figure 8.24** Double-ended queues

In a deque, two pointers are maintained, LEFT and RIGHT, which point to either end of the deque. The elements in a deque extend from the LEFT end to the RIGHT end and since it is circular, Dequeue[N–1] is followed by Dequeue[0].

# ❑ Double Ended Queue (DE-Queue)

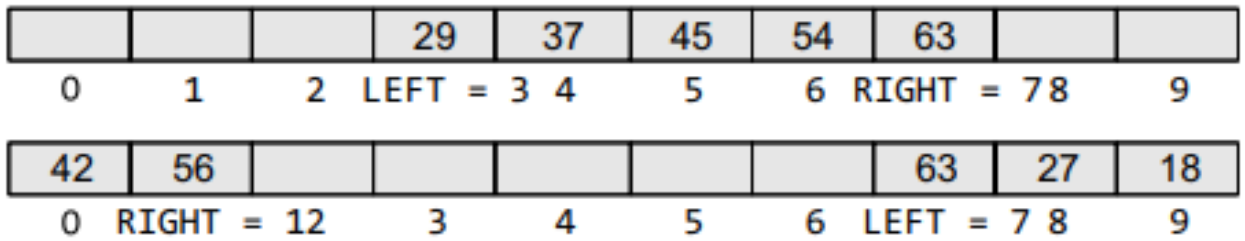



**Figure 8.24** Double-ended queues

Input restricted deque: In this dequeue, insertions can be done only at one of the ends, while deletions can be done from both ends.

Output restricted deque: In this dequeue, deletions can be done only at one of the ends, while insertions can be done on both ends.

https://www.youtube.com/watch?v=pSSXpYd2QxE

# ❑ Double Ended Queue (DE-Queue)

Step1 : start

Step 2: show options 1. create, 2. Insertrear , 3. Insertfront,
4.DeleteRear 5. Deletefront

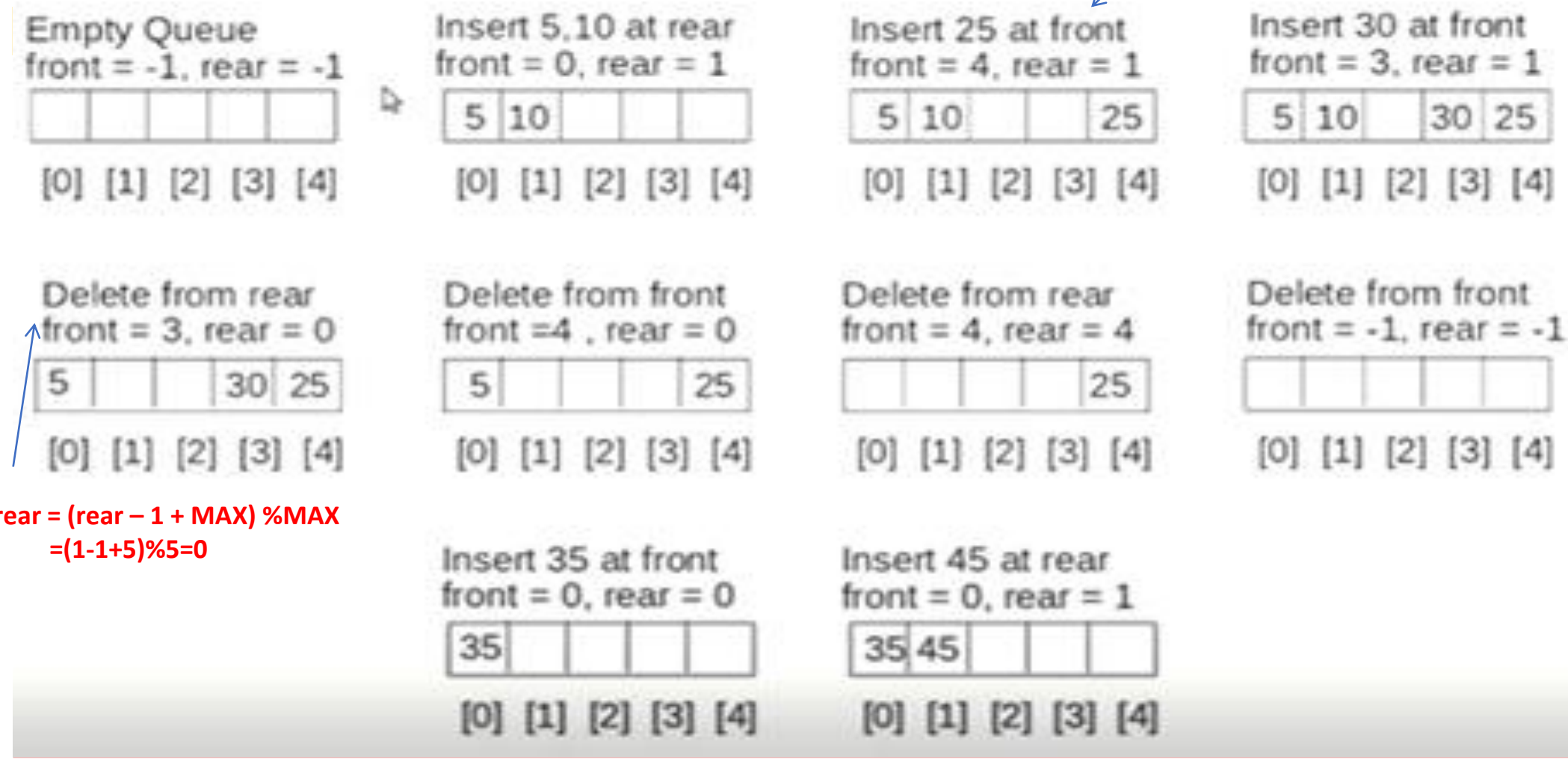
Step 3: accept choice

Step 4: As per users choice call a function

Step 5: exit

# ❑ Double Ended Queue (DE-Queue)

front = (front -1 +MAX) % MAX
=(0-1+5)%5=4




rear = (rear – 1 + MAX) %MAX
=(1-1+5)%5=0

## Algorithm for EnqueueFront

Step 1: if queue is full print message overflow and exit
Step 2: accept element
Step3: if queue is empty, set front=rear=0
data[0] = element
else
front = (front -1 +MAX) % MAX
data [ front] = element

## Algorithm for DequeueFront

Step 1: if queue is empty print message underflow and exit
Step 2: x = data[front]
Step3: if rear = front
initialize() // front,rear=-1
else
front = (front + 1) %MAX
Step 4: Return (x)

## Algorithm for EnqueueRear

Step 1: if queue is full print message overflow and exit
Step 2: accept element
Step3: if queue is empty, set front=rear=0
data[0] = element
else
rear = (rear + 1) % MAX
data [ rear] = element

## Algorithm for DeQueueRear

Step 1: if queue is empty print message underflow and exit
Step 2: x = data[rear]
Step3: if rear = front
initialize() //front,rear=-1
else
rear = (rear – 1 + MAX) %MAX
Step 4: Return (x)

# ❑ Priority Queue:

- Each element is assigned a priority.
- Priority determines the order in which the elements will be processed

- An element with higher priority is processed before an element with a lower priority.
- Two elements with the same priority are processed on a first-come-first-served (FCFS) basis
- widely used in operating systems to execute the highest priority process first.

# ❑ Priority Queue:

For example, if there are three processes, where the first process needs 5 ns to complete, the second process needs 4 ns, and the third process needs 7 ns, then the second process will have the highest priority and will thus be the first to be executed.

However, CPU time is not the only factor that determines the priority, rather it is just one among several factors.

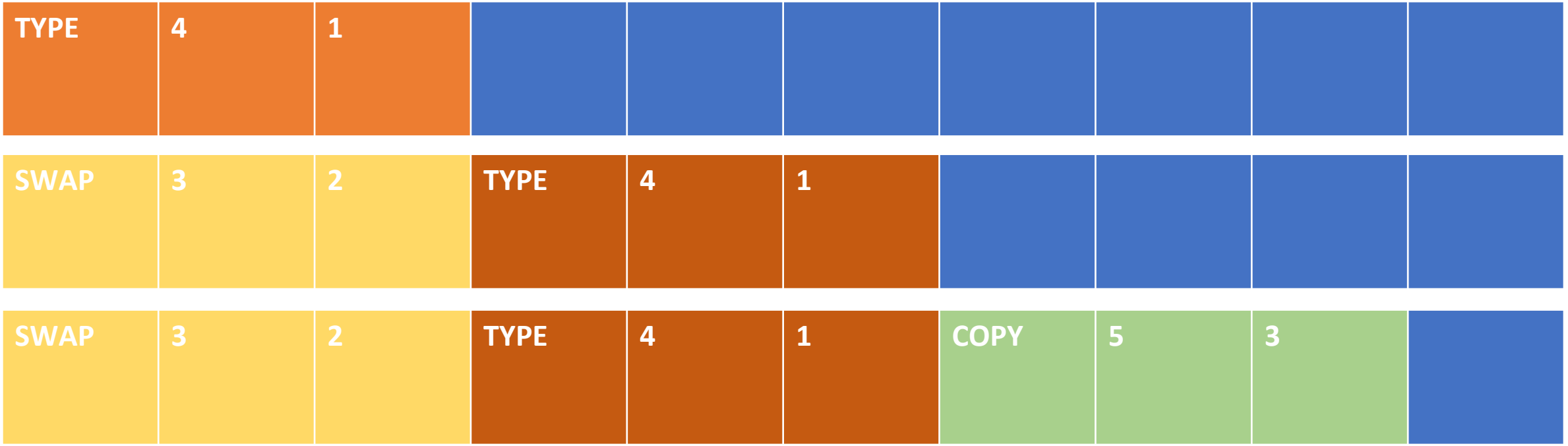
# ❑ Priority Queue:

Using arrays

```
Struct data{

        int element;
        int priority ;
}
```

Insert operation – adding element at location depending on priority

| Ex. Jobs | priority (Execution Time) | sequence |
|---|---|---|
| TYPE | 4 | 1 |
| SWAP | 3 | 2 |
| COPY | 5 | 3 |



Inserting element in priority queue:

Step 1: if rear = MAX-1, then print message overflow and exit

Step 2: Accept element

Step 3: Search position of element as per the priority by moving existing elements

Step 4: set new element at proper position

Deleting element in priority queue:

Step 1: if front = rear = -1, then print message underflow and exit

Step 2: print front element as deleted

Step 3: P = PQ[front]

Step 4: increment front by one

Step 5: return P

Inserting element in priority queue:

Step 1: if rear = MAX-1, then print message overflow and exit

Step 2: Accept element

Step 3: Search position of element as per the priority by moving existing elements

Step 4: set new element at proper position

Deleting element in priority queue:

Step 1: if front = rear = -1, then print message underflow and exit

Step 2: print front element as deleted

Step 3: P = PQ[front]

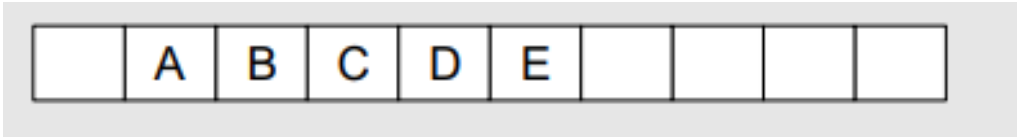
Step 4: increment front by one

Step 5: return P

Que: Draw the queue structure in each case when the following operations are performed on an empty queue.

(a) Add A, B, C, D, E, F
(b) Delete two letters
(c) Add G
(d) Add H
(e) Delete four letters
(f) Add I

Que: Consider the queue given below which has FRONT = 1 and REAR = 5.




Now perform the following operations on the queue: (a) Add F (b) Delete two letters (c) Add G (d) Add H (e) Delete four letters (f) Add I

Identify which statements are incorrect

1. The elements are inserted from Rear in Queue
2. Dequeue operation will delete an element from front
3. isEmpty function will check is Rear = 0
4. The circular queue will be full only when FRONT = MAX –1 and REAR = Max –1
5. Priority Queue is useful in handling interrupts.
6. Output-restricted deque allows deletions to be done only at one end of the dequeue, while insertions can be done at both the ends.
7. Output restricted Deque allows insertion of elements at either ends but not in the middle.
8. In queue, IsFull operation checks for Front = Rear = 0

Identify which statements are incorrect

1. The elements are inserted from Rear in Queue : Correct
2. Dequeue operation will delete an element from front : correct
3. isEmpty function will check if Rear = 0 : incorrect checks for front = rear = -1
4. The circular queue will be full only when FRONT = MAX –1 and REAR = Max –1 : incorrect front = 0 and rear = max-1
5. Priority Queue is useful in handling interrupts: correct
6. Output-restricted deque allows deletions to be done only at one end of the dequeue, while insertions can be done at both the ends.: correct
7. In queue, IsFull operation checks for Front = Rear = 0: incorrect it check for rear = max-1

- Introduction to Data Structures: Linear and Non Linear Data Structures, Static and Dynamic Data Structures.
- Concept of Stack and Queue.
  - ❑ Array Implementation of Stack and Queue,
  - ❑ Circular Queue,
  - ❑ Double Ended Queue,
  - ❑ Priority Queue.
- **Concept of Linked Lists.**
  - ❑ **Singly linked lists,**
  - ❑ **doubly linked lists and**
  - ❑ **circular linked lists.**
- Insertion, deletion, update and copying operations with Singly linked lists, doubly linked lists and circular linked lists.
- Reversing a singly linked list.

**Self-learning Topics:** Linked List Implementation of Stack, Linked List implementation of Queue, Circular Queue, Double Ended Queue, Priority Queue.]

1. Can we use arrays if we are not sure of the number of elements in advance?

1. Does array make efficient use of memory?

1. What if elements can be stored randomly at any location rather than in consecutive locations, is it possible?

https://www.youtube.com/watch?v=pSSXpYd2QxE

# Using arrays for storing values

## 1. Drawbacks:

## Memory wastage : int arr[5]

| **Element** | **11** | **12** | **13** | **Mempry wastage** | |
|---|---|---|---|---|---|
| Index | 0 | 1 | 2 | 3 | 4 |

## Memory shortage

| **Element** | **11** | **12** | **13** | **14** | **15** |
|---|---|---|---|---|---|
| Index | 0 | 1 | 2 | 3 | 4 |

What if 6th element is given by user

# ❑ Concept of Linked Lists

- is a linear collection of data elements.
- user can add any number of elements to it
- data elements are called nodes
- acts as a building block to implement data structures such as stacks, queues, and their variations
- each node contains data field and a pointer to the next node.

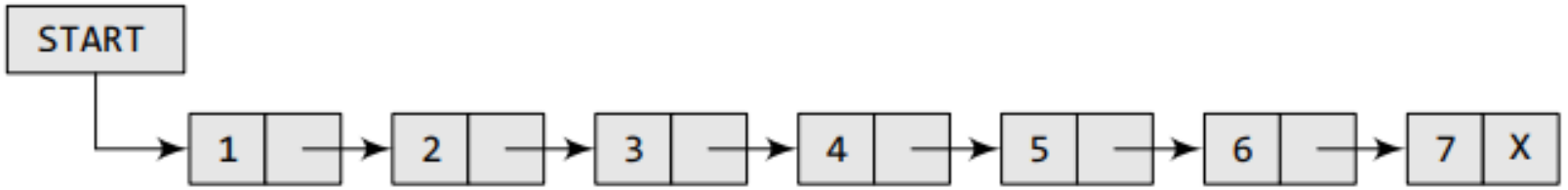



**Figure 6.1** Simple linked list

# ❑ Concept of Linked Lists

START is used to store the address of the first node.

In C, we can implement a linked list using the following code:

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

**Node:** every elements in linked list is called node

Data: information or actual value

**Next** is a second part of a node, it contains address of next node. It helps us to go from one node to other node

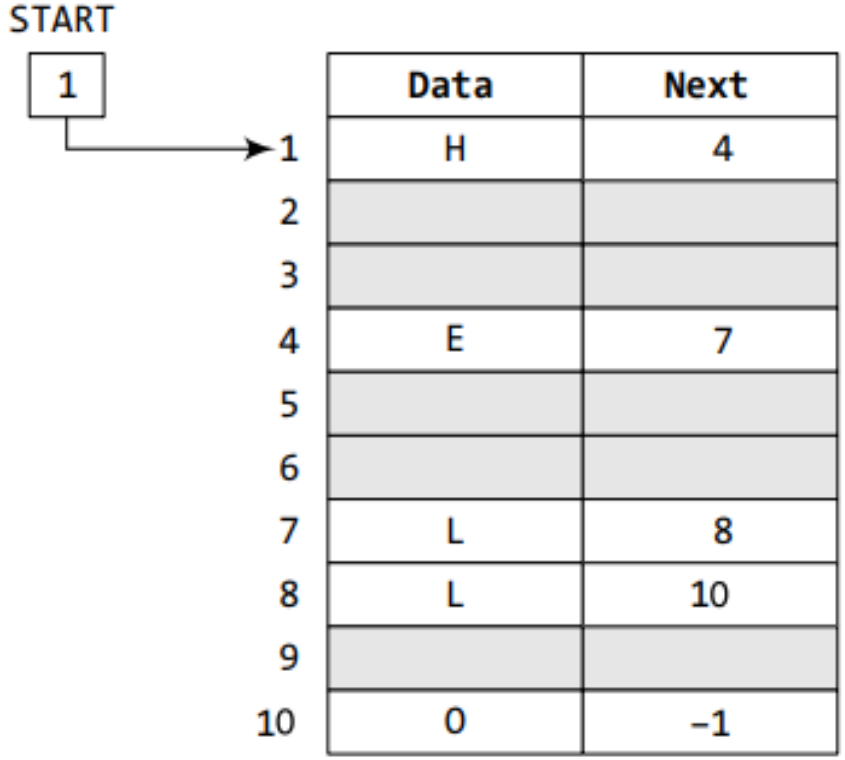


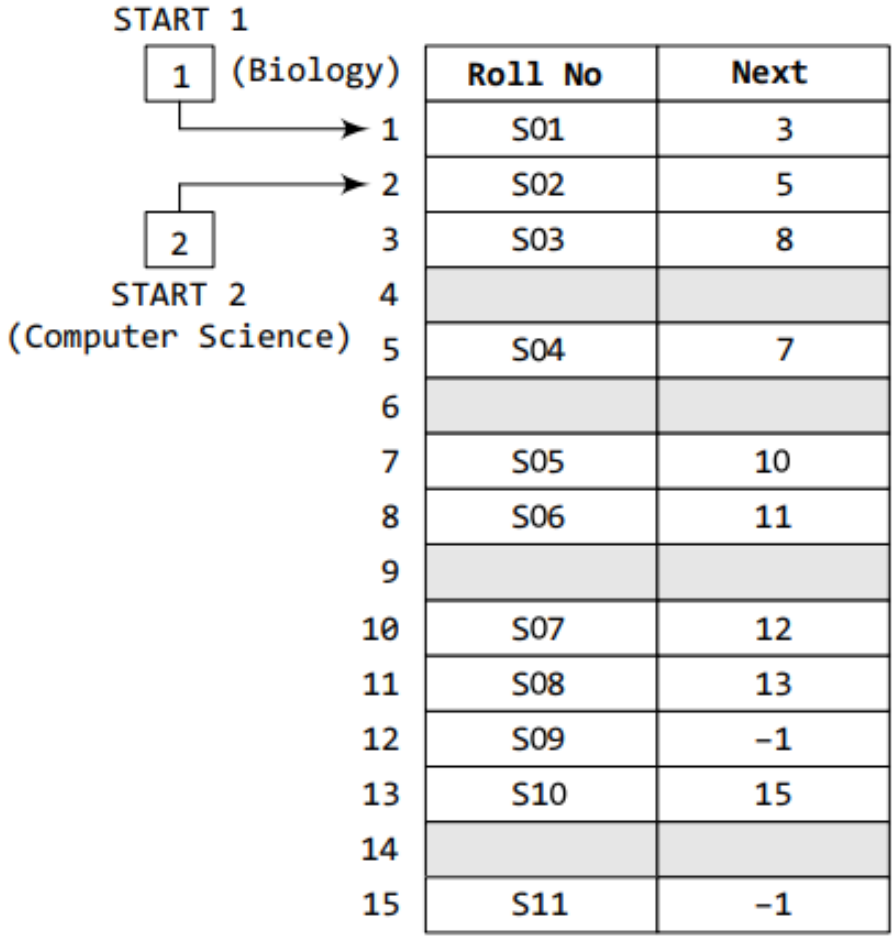

**Figure 6.2** START pointing to the first element of the linked list in the memory

a position where the NEXT entry contains –1 or NULL is end of the linked list

# ❑ Concept of Linked Lists

Header node:

- A pointer that always points to the first node of the linked list
- It is considered as a reference to the link list
- You can also call it Start node
- If start node or header value is NULL means the link list is empty




| | Roll No | Next |
|---|---|---|
| 1 | S01 | 3 |
| 2 | S02 | 5 |
| 3 | S03 | 8 |
| 4 | | |
| 5 | S04 | 7 |
| 6 | | |
| 7 | S05 | 10 |
| 8 | S06 | 11 |
| 9 | | |
| 10 | S07 | 12 |
| 11 | S08 | 13 |
| 12 | S09 | -1 |
| 13 | S10 | 15 |
| 14 | | |
| 15 | S11 | -1 |

**Figure 6.3** Two linked lists which are simultaneously maintained in the memory

# ❑ Memory Allocation and deallocation in Linked list

- Dynamic Memory allocation
- As per the necessity allocation and de allocation occurs
- Avoids memory shortage and memory wastage
- Use of malloc()
- Deallocating by using free function()

# ❑ Concept of Linked Lists

START
18

| | Roll No | Name | Aggregate | Grade | Next |
|---|---|---|---|---|---|
| 1 | S01 | Ram | 78 | Distinction | 6 |
| 2 | S02 | Shyam | 64 | First division | 14 |
| 3 | | | | | |
| 4 | S03 | Mohit | 89 | Outstanding | 17 |
| 5 | | | | | |
| 6 | S04 | Rohit | 77 | Distinction | 2 |
| 7 | S05 | Varun | 86 | Outstanding | 10 |
| 8 | S06 | Karan | 65 | First division | 12 |
| 9 | | | | | |
| 10 | S07 | Veena | 54 | Second division | -1 |
| 11 | S08 | Meera | 67 | First division | 4 |
| 12 | S09 | Krish | 45 | Third division | 13 |
| 13 | S10 | Kusum | 91 | Outstanding | 11 |
| 14 | S11 | Silky | 72 | First division | 7 |
| 15 | | | | | |
| 16 | | | | | |
| 17 | S12 | Monica | 75 | Distinction | 1 |
| 18 | S13 | Ashish | 63 | First division | 19 |
| 19 | S14 | Gaurav | 61 | First division | 8 |

**Figure 6.4** Students' linked list

# ❑ Linked Lists Vs Arrays

| Linked List | Arrays |
|---|---|
| Linear data collection | Linear data collection |
| Data may not stored in consecutive memory locations | Data stored in consecutive memory locations |
| No random access of data | Random access of data allowed |
| Nodes in a linked list can be accessed only in a sequential manner. | Data elements can be accessed randomly |
| insertions and deletions can be done at any point in the list in a constant time. | Insertion deletions can not be done at any point in array |
| Flexible in Size | Fixed in size |

# ❑ Types of Linked Lists

- **Single link list**
- **Double**
- **Circular**

Pointer:
-holds the address of the entire structure.
-Members of the structure can be accessed using (->) operator

# ❑ Singly Linked Lists

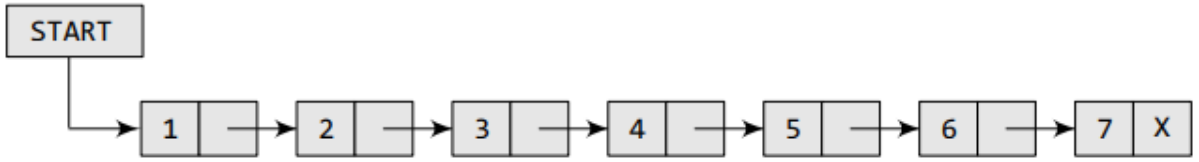
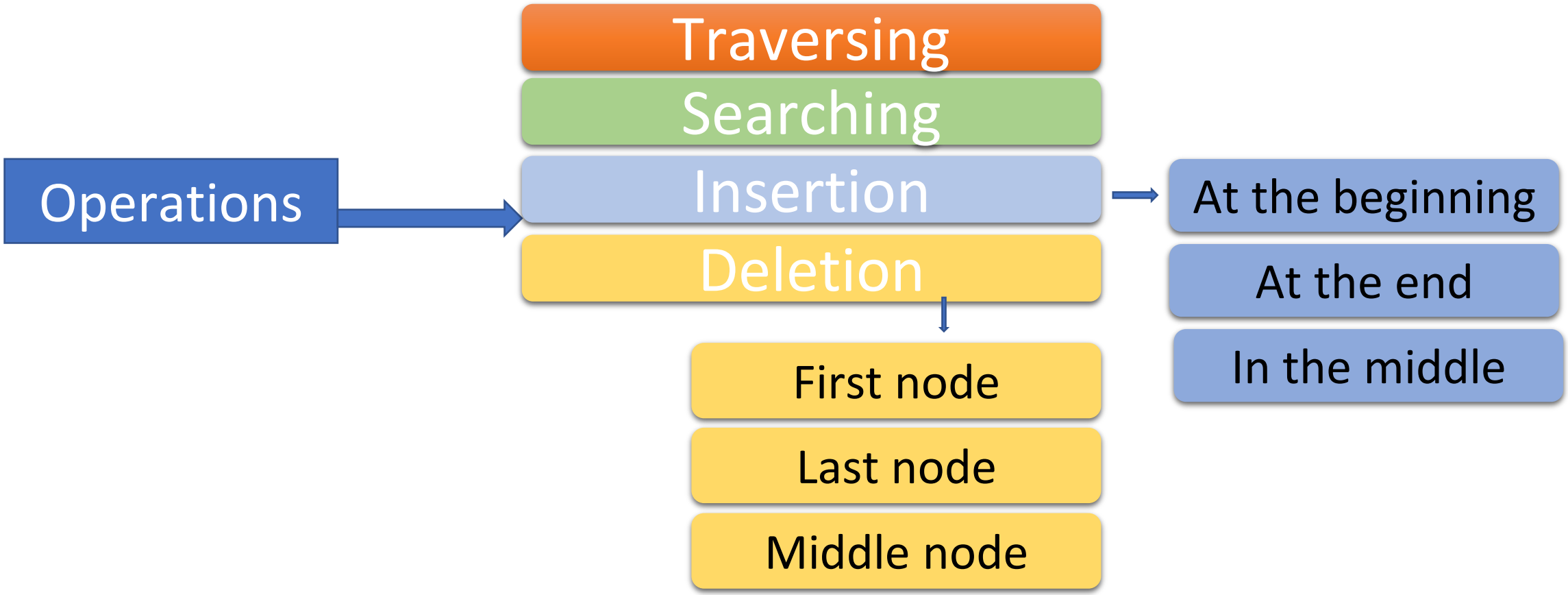



**Figure 6.7** Singly linked list

- every node contains some data and a pointer to the next node of the same data type.
- the node stores the address of the next node in sequence.
- A singly linked list allows traversal of data only in one way.

Note: Since in a linked list, every node contains a pointer to another node which is of the same type, it is also called a **self-referential data type**

# ❑ Operations on **Singly Linked Lists**



# ❑ Searchng &Traversing Singly Linke

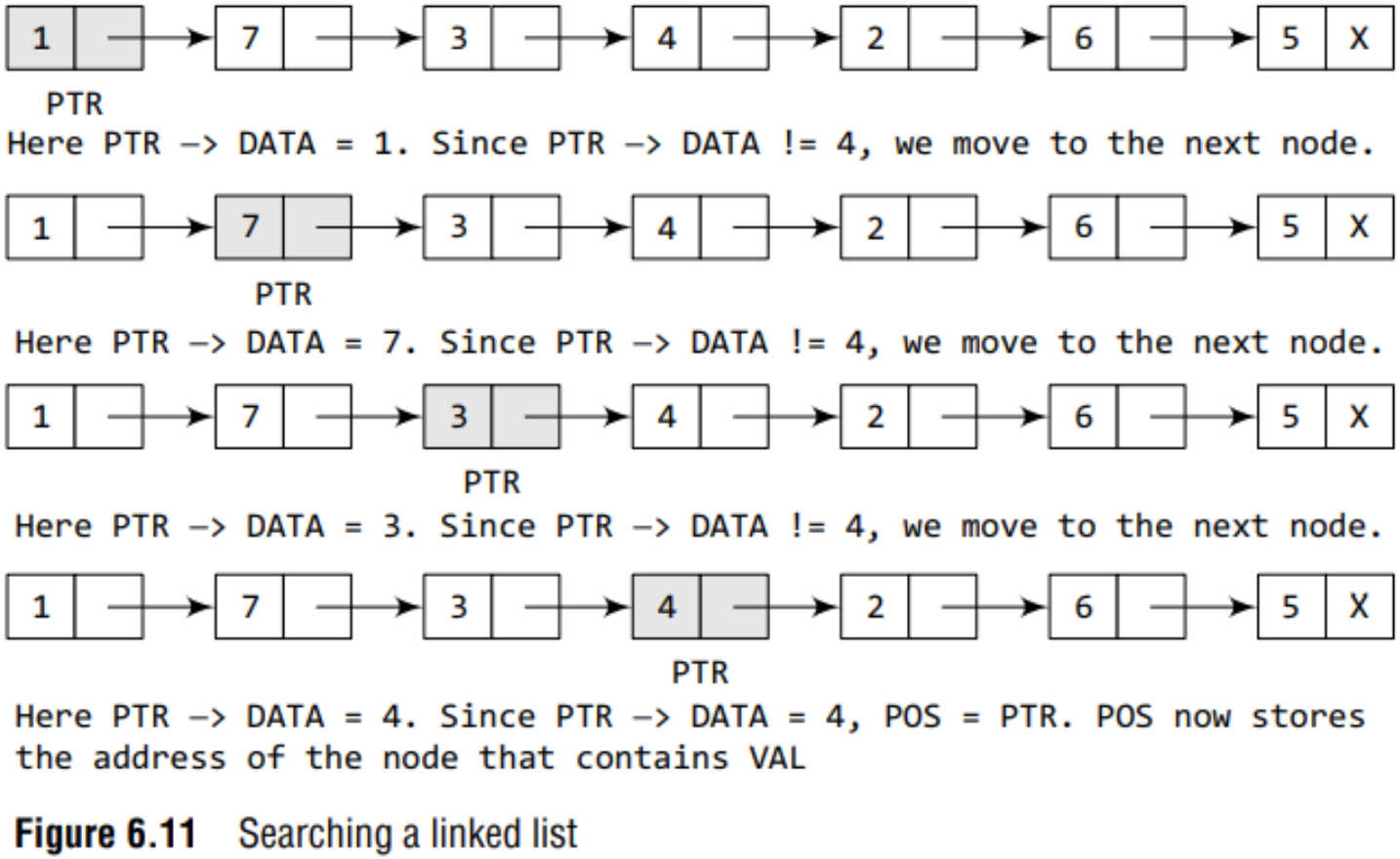



**Figure 6.11** Searching a linked list

# ❑ Traversing Singly Linked Lists

```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3:            Apply Process to PTR -> DATA
Step 4:            SET PTR = PTR -> NEXT
       [END OF LOOP]
Step 5: EXIT
```

**Figure 6.8** Algorithm for traversing a linked list

```
Step 1: [INITIALIZE] SET COUNT = 0
Step 2: [INITIALIZE] SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR != NULL
Step 4:            SET COUNT = COUNT + 1
Step 5:            SET PTR = PTR -> NEXT
       [END OF LOOP]
Step 6: Write COUNT
Step 7: EXIT
```

**Figure 6.9** Algorithm to print the number of nodes in a linked list

# ❑ Traversing Singly Linked Lists

```
Void display (struct node *q)
{

        /* address of head is passed to q*/
        while(q!=NULL)
        {
                printf("%d", q-> data);
                q = q->next;
        }
}
```

# ❑ Searching in Singly Linked Lists

```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Step 3 while PTR != NULL
Step 3:     IF VAL = PTR -> DATA
                SET POS = PTR
                Go To Step 5
            ELSE
                SET PTR = PTR -> NEXT
            [END OF IF]
        [END OF LOOP]
Step 4: SET POS = NULL
Step 5: EXIT
```

**Figure 6.10** Algorithm to search a linked list

```
Void search (struct node *q, int data)
{
        while(q!=NULL)
        {
                if(data == q->data)
                {printf("Element found");
                return;
                }
        q = q->next;
        }
        printf("Element not found");
}
```

# ❑ Inserting nodes in Linked Lists

four Insertion cases:

Case 1: The new node is inserted at the beginning.

Case 2: The new node is inserted at the end.

Case 3: The new node is inserted after a given node.

Case 4: The new node is inserted before a given node.

# ❑ Inserting nodes in Linked Lists

Case 1: The new node is inserted at the beginning.

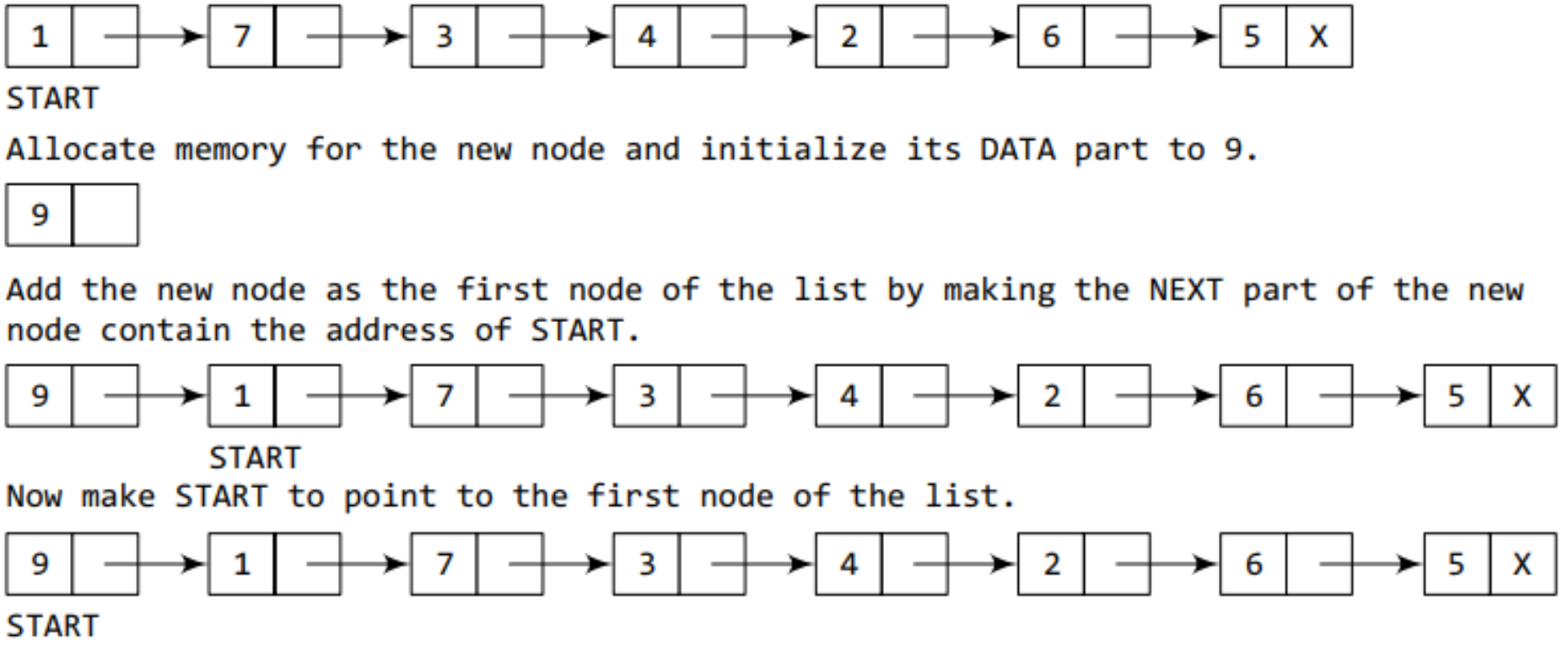



**Figure 6.12** Inserting an element at the beginning of a linked list

# ❑ Inserting nodes in Linked Lists

## Case 1: The new node is inserted at the beginning.

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 7
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET START = NEW_NODE
Step 7: EXIT
```

**Figure 6.13** Algorithm to insert a new node at the beginning

- **we first check whether memory is available for the new node.**
- **If the free memory has exhausted, then an OVERFLOW message is printed.**
- **Otherwise, if a free memory cell is available, then we allocate space for the new node.**
- **Set its DATA part with the given VAL and the next part is initialized with the address of the first node of the list, which is stored in START.**
- **Now, since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of the NEW_NODE**

# ❑ Inserting nodes in Linked Lists

## Case 1: The new node is inserted at the beginning.

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 7
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET START = NEW_NODE
Step 7: EXIT
```

**Figure 6.13** Algorithm to insert a new node at the beginning

Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT

These steps allocate memory for the new node.

In C, there are functions like malloc(), alloc, and calloc() which automatically do the memory allocation on behalf of the user.

# ❑ Inserting nodes in Linked Lists

## Case2: Inserting a Node at the End of a Linked List

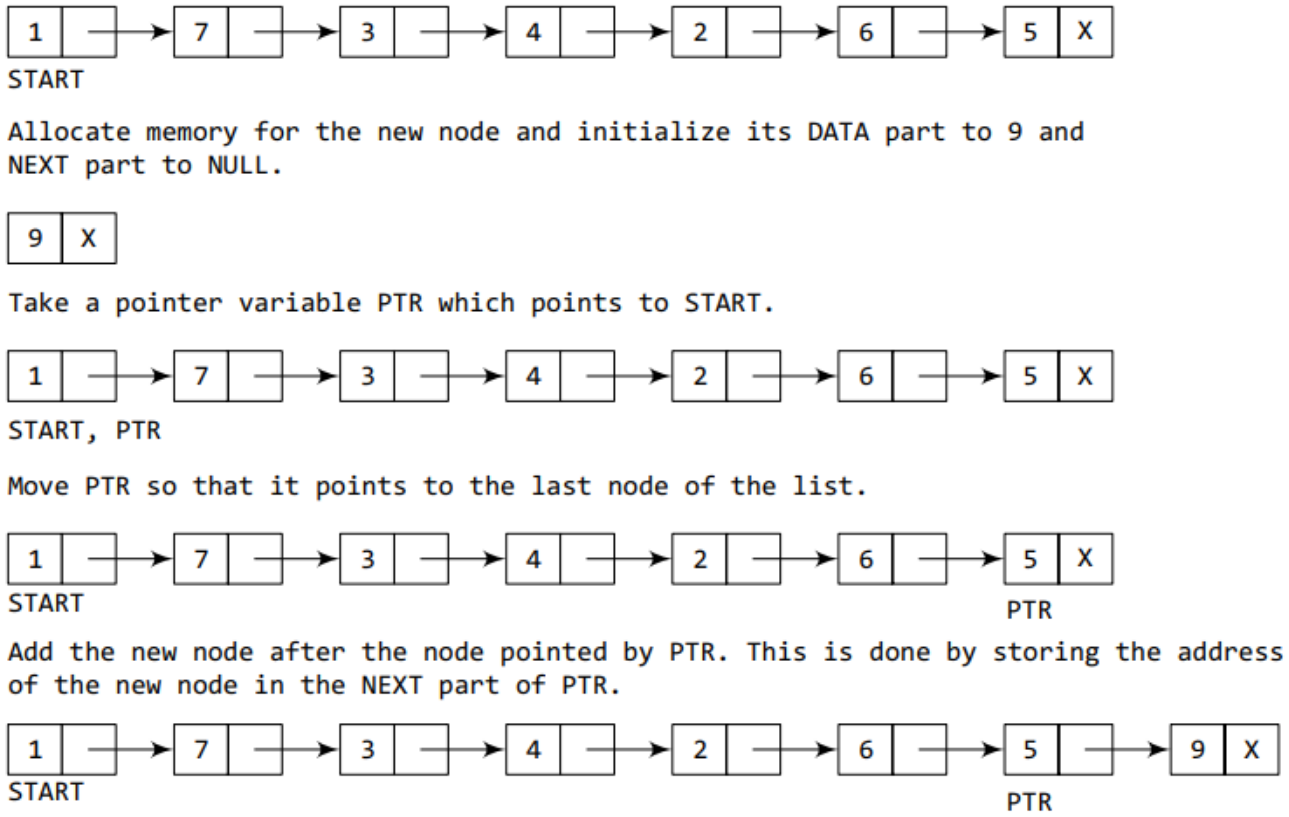



**Figure 6.14** Inserting an element at the end of a linked list

- we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list.
- In the while loop, we traverse through the linked list to reach the last node.
- Once we reach the last node, we change the NEXT pointer of the last node to store the address of the new node.
- Remember that the NEXT field of the new node contains NULL, which signifies the end of the linked list.

# ❑ Inserting nodes in Linked Lists

## Case2: Inserting a Node at the End of a Linked List

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 10
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL->NEXT
Step 4: SET NEW_NODE->DATA = VAL
Step 5: SET NEW_NODE->NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR->NEXT != NULL
Step 8:     SET PTR = PTR->NEXT
        [END OF LOOP]
Step 9: SET PTR->NEXT = NEW_NODE
Step 10: EXIT
```

**Figure 6.15** Algorithm to insert a new node at the end

- we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list.
- In the while loop, we traverse through the linked list to reach the last node.
- Once we reach the last node, we change the NEXT pointer of the last node to store the address of the new node.
- Remember that the NEXT field of the new node contains NULL, which signifies the end of the linked list.

# ❑ Inserting nodes in Linked Lists

Case 3: Inserting a Node After a Given Node in a Linked List

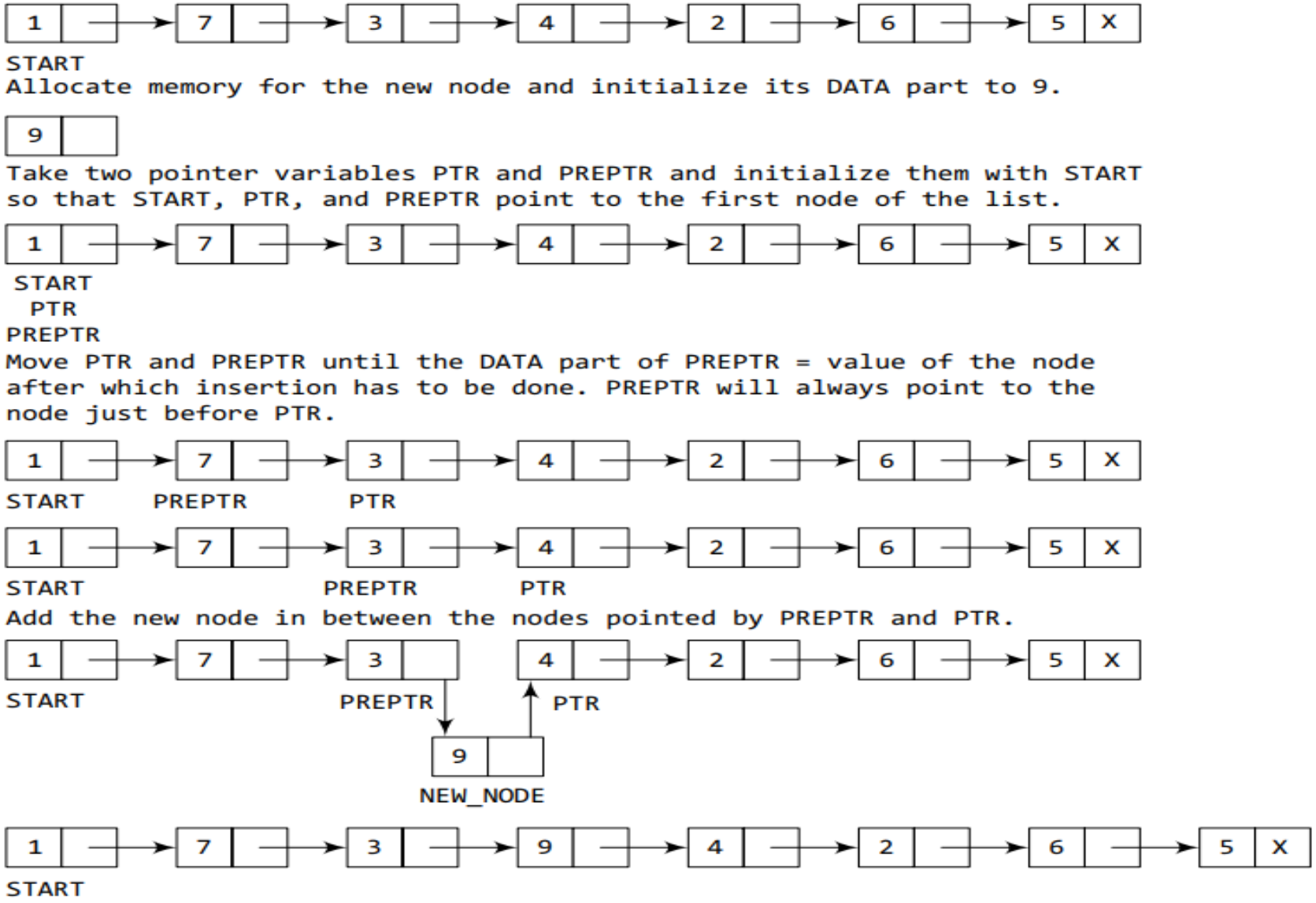



**Figure 6.17** Inserting an element after a given node in a linked list

# ❑ Concept of Linked Lists

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PREPTR -> DATA
        != NUM
Step 8:     SET PREPTR = PTR
Step 9:     SET PTR = PTR -> NEXT
         [END OF LOOP]
Step 10: PREPTR -> NEXT = NEW_NODE
Step 11: SET NEW_NODE -> NEXT = PTR
Step 12: EXIT
```

**Figure 6.16** Algorithm to insert a new node after a node that has value NUM

- PREPTR which will be used to store the address of the node preceding PTR. Initially, PREPTR is initialized to PTR.
- So now, PTR, PREPTR, and START are all pointing to the first node of the linked list.
- In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM. We need to reach this node because the new node will be inserted after this node.
- Once we reach this node, in Steps 10 and 11, we change the NEXT pointers in such a way that new node is inserted after the desired node.

# ❑ Deleting nodes in Linked Lists

Three cases:

Case 1: The first node is deleted.

Case 2: The last node is deleted.

Case 3: The node after a given node is deleted.

Note: Underflow is a condition that occurs when we try to delete a node from a linked list that is empty.

This happens when START = NULL or when there are no more nodes to delete.

# ❑ Deleting nodes in Linked Lists

## Case 1: The first node is deleted.

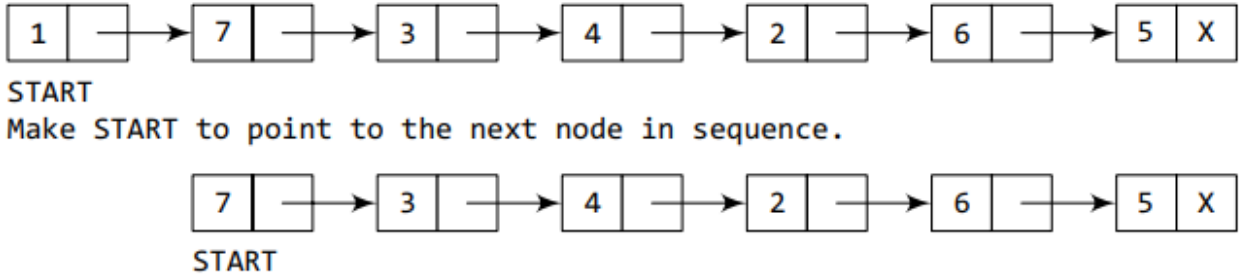



**Figure 6.20** Deleting the first node of a linked list

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 5
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: FREE PTR
Step 5: EXIT
```

**Figure 6.21** Algorithm to delete the first node

- In Step 1, we check if the linked list exists or not.
- If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm.
- However, if there are nodes in the linked list, then we use a pointer variable PTR that is set to point to the first node of the list.
- For this, we initialize PTR with START that stores the address of the first node of the list.
- In Step 3, START is made to point to the next node in sequence and finally the memory occupied by the node pointed by PTR (initially the first node of the list) is freed and returned to the free pool.

# ❑ Deleting nodes in Linked Lists

## Case 1: Deleting the Last Node from a Linked List.

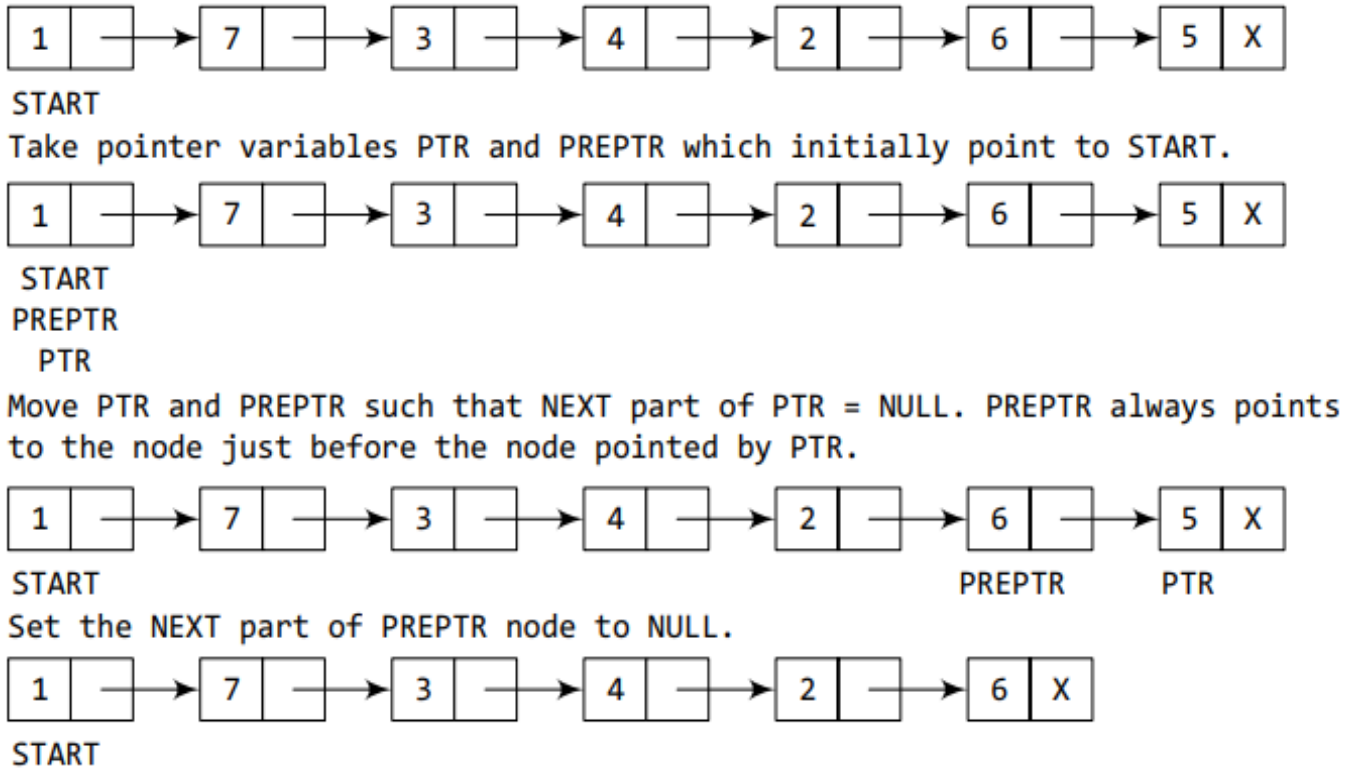



**Figure 6.22** Deleting the last node of a linked list

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 8
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != NULL
Step 4:     SET PREPTR = PTR
Step 5:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 6: SET PREPTR -> NEXT = NULL
Step 7: FREE PTR
Step 8: EXIT
```

**Figure 6.23** Algorithm to delete the last node

# ❑ Types of Linked Lists

- Singly
- Doubly
- Circular link list

# ❑ Types of Linked Lists

## □ Doubly Linked List-

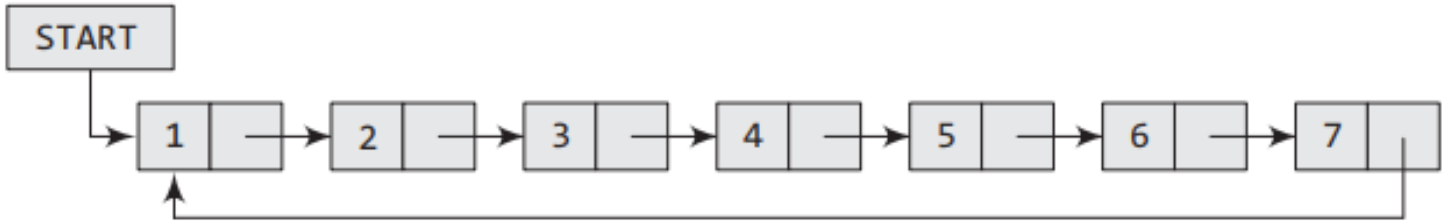
# ❑ Types of Linked Lists

## ▯ Doubly Linked List

# ❑ Circular link list

we can begin at any node and traverse the list in any direction, forward or backward, until we reach the same node where we started. Thus, a circular linked list has no beginning and no ending.

# ❑ Circular link list




we can begin at any node and traverse the list in any direction, forward or backward, until we reach the same node where we started.

Thus, a circular linked list has no beginning and no ending.

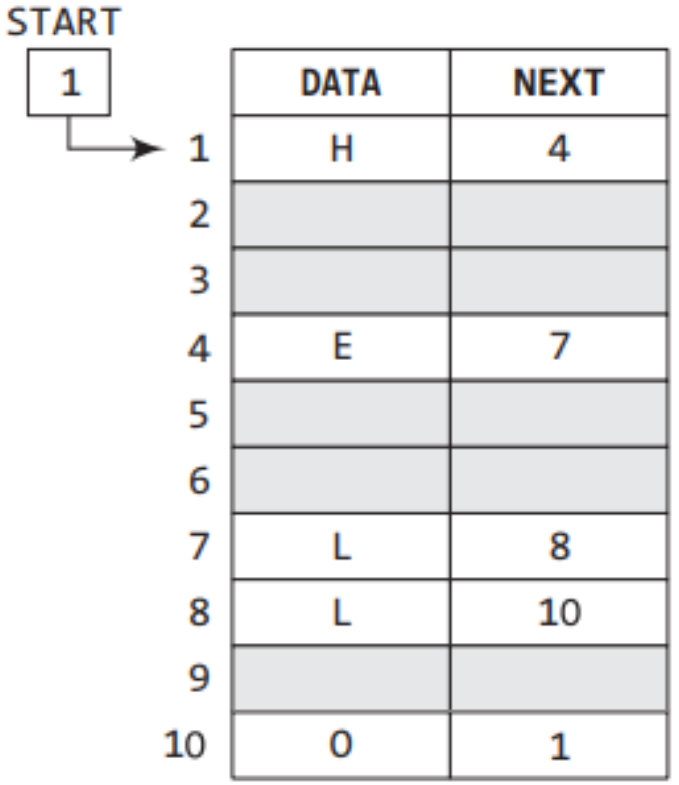
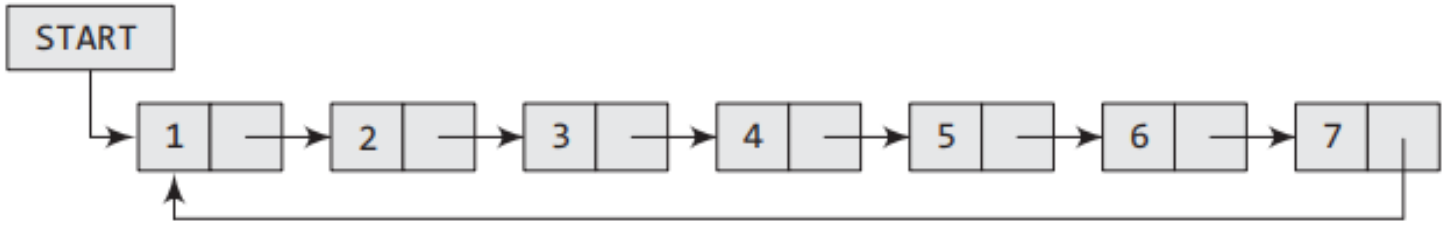



**Figure 6.27** Memory representation of a circular linked list

# ❑ Circular link list




Operations :
Insertion
Deletion
Displaying data

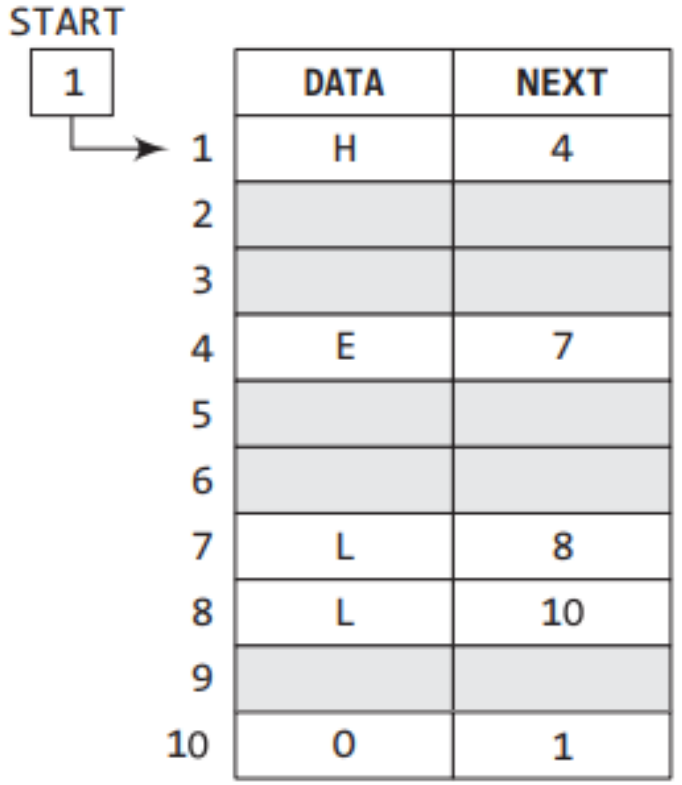



**Figure 6.27** Memory representation of a circular linked list

# ❑ Inserting nodes in Circular Linked Lists

Algorithm:

1. Alloc(NewNode)
2. Read data for the node
3. If rear is NULL
4. set both front and rear to newnode
5. Else
6. assign address of newnode to rear->next
7. Set rear to newnode
8. Set address of front in rear->next

# ❑ Inserting nodes in Circular Linked Lists

Algorithm:

1. Alloc(NewNode)
2. Read data for the node
3. If rear is NULL
4. set both front and rear to newnode
5. Else
6. assign address of newnode to rear next
7. Set rear to newnode
8. Set address of front in rear next
9.

# References:

1.Reema Thareja; Data Structures using C; Oxford.
2. Ellis Horowitz, Sartaj Sahni; Fundamentals of Data Structures; Galgotia Publications; 2010.

https://www.digitalocean.com/community/tutorials/stack-in-c