

UNIT 3

Syllabus: List, Stack and Queue ADT Linked List: Array Vs Linked List - Singly Linked List, Doubly Linked Lists – Circular Linked Lists-implementation - application. Stack and Queue: Introduction – implementation (static and dynamic) application – Circular queues-application.

Array Vs Linked List

ARRAYS CONCEPTS

An array most commonly used **non-primitive data structures**. It is defined as a set of **finite number of homogeneous elements** or data items. Or we can say that an array is a **collection of similar elements**. These similar elements could be all **ints**, or all **floats**, or all **chars**, etc. Usually, the **array of characters is called a 'string'**, whereas an array of **ints** or **floats** is called simply an array. It means an array can contain one type of data only, either all integers, all floating-point numbers, or all characters. Declaration of arrays is as follows :

```
int a[10] ;  
int num[6] = { 2, 4, 12, 5, 45, 5 } ;  
int n[ ] = { 2, 4, 12, 5, 45, 5 } ;
```

Where **int** specifies the data type or type of elements array stores. "**a**" is the name of array, and the **number specified inside the square brackets** is the number of **elements an array** can store this is also **called size or length of array**.

Following are some of the concepts to be remembered about arrays:

1. The **individual element** of an array can be accessed by specifying **name of the array followed by index** or subscript inside square brackets. For. example to access fifth element of array a, we have to give the following statement : **a[4]**
2. The **first element** of the array has **index zero [0]**. It means the first element and last element will be specified as: **a[0] and a[9]** respectively.
3. The elements of array will always be **stored in consecutive memory locations**.
4. The number of elements that can be stored in an array i.e, the **size of an array** or its length is given by the following equation: - **(upper bound – lower bound) + 1**.
5. Arrays can always be **read or written through loop**. If we read-a one-dimensional array, it requires one loop for reading and other for writing (printing) the array, for example :

(a) For reading an array
for (i =0; i<=9; i++)
{ scanf ("%d", &a[i]) };

(b) For writing an array
for(i=0; i<=9; i++)
{ printf("%d", a[i]) ;

Some **common operations performed on arrays** are :

1. Creation of an array.
2. Traversing an array (accessing array elements)
3. Insertion of new elements.
4. Deletion of required elements.
5. Modification of an element.
6. Merging of arrays.

A program of array to find average marks.

```
main( )
{
    int avg, sum = 0 ; int i ;
    int marks[30] ;    /* array declaration */
    clrscr();
    for ( i = 0 ; i <= 29 ; i++ )
    {
        printf ( "\nEnter marks " ) ;
        scanf ( "%d", &marks[i] ) ; /* store data in array */
    }
    for ( i = 0 ; i <= 29 ; i++ )
        sum = sum + marks[i] ; /* read data from an array */
    avg = sum / 30 ;
    printf ( "\nAverage marks = %d", avg ) ;
    getch();
}
```

Array **elements can be passed to a function** by calling the function by value, or by reference. In the **call by value we pass values of array elements** to the function, whereas in the **call by reference we pass addresses of array elements** to the function. We can also use Pointer in array. Array elements are always stored in contiguous memory locations. A pointer when incremented always points to an immediately next location of its type.

Types of Array

1. One dimension Array
2. Two dimension Array
3. Three dimension Array

One dimension Array - Above details are of 1-D array. Declaration of one dimension arrays is as follows :

```
int a[10] ;
int num[6] = { 2, 4, 12, 5, 45, 5 } ;
int n[ ] = { 2, 4, 12, 5, 45, 5 } ;
```

Two dimension Array - It is also possible for arrays to have two or more dimensions. The **two-dimensional** array is also **called a matrix**. A sample program that stores roll number and marks obtained by a student side by side in a matrix.

```

main( )
{
int stud[4][2] ; int i, j ;
clrscr();
for ( i = 0 ; i <= 3 ; i++ )
{
printf ( "\n Enter roll no. and marks" ) ;
scanf ( "%d %d", &stud[i][0], &stud[i][1] ) ;
}
for ( i = 0 ; i <= 3 ; i++ )
printf ( "\n%d %d", stud[i][0], stud[i][1] ) ;
getch();
}

```

There are **two parts** to the program—in the first part through a **for** loop we read in the values of roll no. and marks, whereas, in second part through another **for** loop we print out these values.

Look at the **scanf()** statement used in the first **for** loop:
scanf ("%d %d", &stud[i][0], &stud[i][1]) ;

In **stud[i][0]** and **stud[i][1]** the first subscript of the variable **stud**, is row number which changes for every student. The second subscript tells which of the two columns are we talking about—the **zeroth** column which contains the roll no. or the first column which contains the marks. Remember the counting of rows and columns begin with zero. The complete array arrangement is shown below.

Thus, 1234 is stored in **stud[0][0]**, 56 is stored in **stud[0][1]** and so on. The above arrangement highlights the fact that a two-dimensional array is nothing but a collection of a number of one- dimensional arrays placed one below the other.

	col. no. 0	col. no. 1
row no. 0	1234	56
row no. 1	1212	33
row no. 2	1434	80
row no. 3	1312	78

Figure 1. 2-D array

In our sample program the array elements have been stored rowwise and accessed rowwise. However, you can access the array elements columnwise as well. Traditionally, the array elements are being stored and accessed rowwise; therefore we would also stick to the same strategy. Initialising a 2-Dimensional Array is done as

```
int stud[4][2] = { { 1234, 56 }, { 1212, 33 }, { 1434, 80 }, { 1312, 78 } } ;
```

or even this would work...

```
int stud[4][2] = { 1234, 56, 1212, 33, 1434, 80, 1312, 78 } ;
```

Three dimension Array - A three-dimensional array can be thought of as an **array of arrays of arrays**. The outer array has **three elements**, each of which is a two-dimensional array of four one-dimensional arrays, each of which contains two integers. In other words, a one-dimensional array of two elements is constructed first. Then four such one-dimensional arrays are placed one below the other to give a two-dimensional array containing four rows. Then, three such two-dimensional arrays are **placed one**

behind the other to yield a three-dimensional array containing three 2-dimensional arrays. In the array declaration note how the commas have been given. Figure 2 would possibly help you in visualising the situation better. In practice, 3-D array is rarely used. An example of initializing a three-dimensional array for understanding is shown:

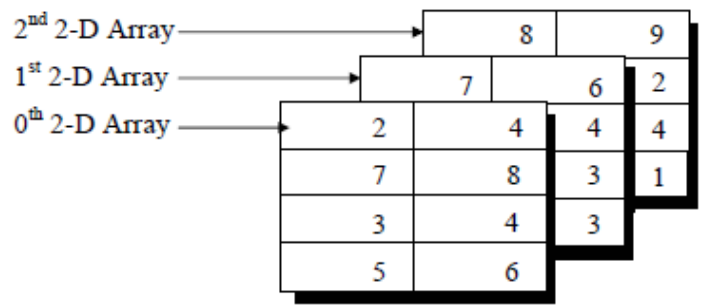


Figure 2. 3-D array

```
int arr[4][2][3] = { { { 2, 4 }, { 7, 8 }, { 3, 4 }, { 5, 6 } }, { { 7, 6 }, { 3, 4 }, { 5, 3 }, { 2, 3 } }, { { 8, 9 }, { 7, 2 }, { 3, 4 }, { 5, 1 } }, { } };
```

Advantages of Arrays:

1. Arrays store multiple data of similar types with the same name.
2. It allows random access to elements.
3. As the array is of fixed size and stored in contiguous memory locations there is no memory shortage or overflow.
4. It is helpful to store any type of data with a fixed size.
5. Since the elements in the array are stored at contiguous memory locations it is easy to iterate in this data structure and unit time is required to access an element if the index is known.

Disadvantages of Arrays:

1. The array is static in nature. Once the size of the array is declared then we can't modify it.
2. Insertion and deletion operations are difficult in an array as elements are stored in contiguous memory locations and the shifting operations are costly.
3. The number of elements that have to be stored in an array should be known in advance.
4. Wastage of memory is the main problem in the array. If the array size is big the less allocation of memory leads to wastage of memory.

Link List Concepts

If the **memory is allocated before the execution** of a program, it is **fixed and cannot be changed**. We have to adopt an alternative strategy to allocated memory only when it is required. There is a **special data structure called linked list** that provides a more **flexible storage system** and it does not require the use of arrays.

For **stacks and queues**, we employed **arrays to represent** them in computer memory. When we choose array representation (**also called sequential representation**) it is necessary to declare in advance the amount of memory to be utilized. We do this by declaring the maximum size of an array. When we really take up arrays, all the memory may not be unused even though allocated. This leads to **unnecessary wastage** of memory. The memory is very important resource. So, we should handle it efficiently.

Linked lists are special list of some data elements linked to one another. The logical ordering is represented by having each element pointing to the next element. Each

element is called a node, which has **two parts. INFO part** which stores the information and **POINTER** which points to the next element. Following Figure 1 shows both types of lists (singly linked list and doubly linked lists).

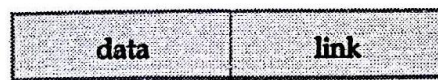


Figure 3. Node of Link List

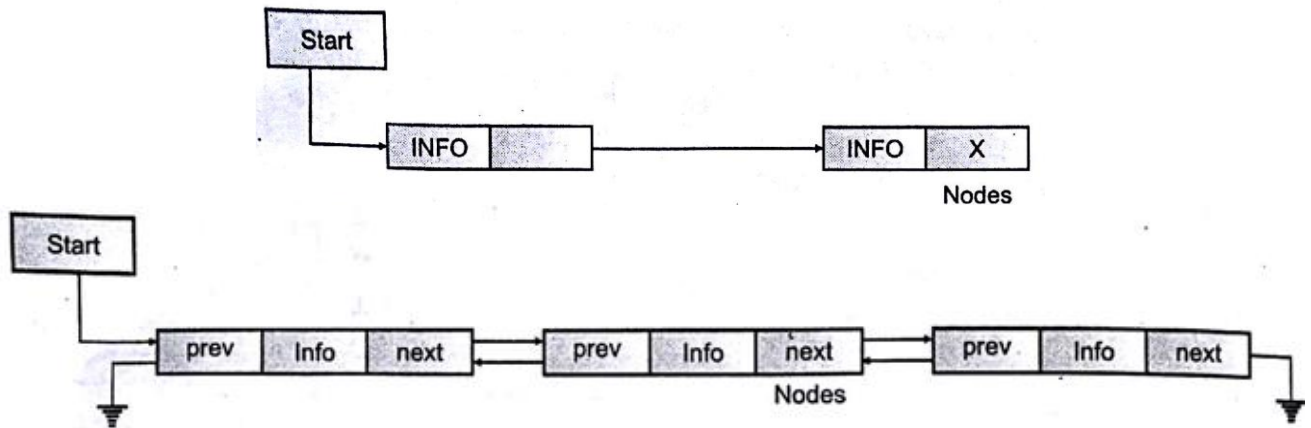


Figure 4. (a) Singly Linked Lists (b) Doubly Linked Lists

Singly linked list nodes have one pointer (Next) pointing to the next node, whereas nodes of doubly linked lists have two pointers (prev. and next). Prev. points to the previous node and next points to the next node in the list.

Why linked list data structure needed?

1. **Dynamic Data structure:** The size of memory can be allocated or de-allocated at run time based on the operation insertion or deletion.
2. **Ease of Insertion/Deletion:** The insertion and deletion of elements are simpler than arrays since no elements need to be shifted after insertion and deletion, Just the address needed to be updated.
3. **Efficient Memory Utilization:** As we know Linked List is a dynamic data structure the size increases or decreases as per the requirement so this avoids the wastage of memory.
4. **Implementation:** Various advanced data structures can be implemented using a linked list like a stack, queue, graph, hash maps, etc.

KEY TERMS

A linked list is a **non-sequential collection** of data items **called nodes**. These nodes in principle are structures containing fields. Each node in a linked list **has basically two fields**.

1. Data field
2. Link field

The **Data field contains an actual value** to be stored and processed. And, the **link field contains the address** of the next data item in the linked list. The address used to access a particular node is known as a pointer. Therefore, the elements in a linked list are ordered not by their physical placement in memory but their logical links stored as part of the data within the node itself.

Null Pointer - The link field of the last node contains NULL rather than a valid address. It is a null pointer and indicates the end of the list.

External Pointer - It is a pointer to the very first node in the linked list, it enables us to access the entire linked list.

Empty list - If the nodes are not present in a linked list, then it is called an empty linked list or simply empty list. It is also called the null list.

Notations - Let **P** be a pointer to node in a linked list. Then, we can form the following notations. To do something on linked lists.

Node (P) ; a node pointed to by the pointer P.

Data (P) ; data of the node pointed to by P.

Link (P) ; address of the next node that follows the node pointed to by the pointer p.

Suppose if the value of **P = 2000** then,

Node (P) = Node (2000) Is the second node, and

Data (P) = Data (2000) = 20

Link (P) = Link (2000) = 3000

Representation Of Linear Linked List

Suppose we want to store list of integer numbers, then the linear linked list can be represented in memory with the following declarations.

```
struct node
{
    int a ;
    struct node *p;
};
typedef struct node n1;
node *start;
```

The above declaration defines a new data type, whose each element is of type node_type and gives it a name n1.

Advantages: Linked lists have many advantages. Some of the very important advantages are:

1. Linked lists are **dynamic data structures**. That is, they can grow or shrink during the execution of a program.
2. **Efficient memory utilization**. Here, memory is not pre-allocated. Memory is allocated whenever it is required. And it is deallocated (removed) when it is no longer needed.
3. **Insertion and deletions are easier and efficient**. Linked lists provide flexibility in inserting a data item at a specified position and deletion of a data item from the given position.
4. Many **complex applications** can be **easily carried out** with linked lists.

Example: suppose we maintain a sorted list of IDs in an array `id[] = [1000, 1010, 1050, 2000, 2040,]`. And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

Deletion is also expensive with arrays unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

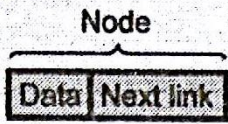
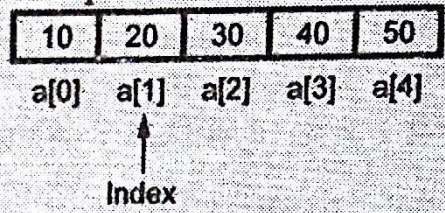
So Linked list provides the following two advantages over arrays:

1. Dynamic size
2. Ease of insertion/deletion

Disadvantages of Linked Lists:

1. **Random access** is not allowed. We have to access elements sequentially starting from the first node. So we cannot do a binary search with linked lists.
2. **Extra memory space** for a pointer is required for each element of the list.
3. Arrays have a better cache locality that can make a pretty big difference in **performance**.
4. It takes a **lot of time in traversing** and changing the pointers.
5. It will be **confusing when** we work with **pointers**.

The comparison between linked list and arrays is as shown below –

Sr. No.	Linked List	Array
1.	<p>The linked list is a collection of nodes and each node is having one data field and next link field. For example</p> <div style="text-align: center;">  </div>	<p>The array is a collection of similar types of data elements. In arrays the data is always stored at some index of the array. For example</p> <div style="text-align: center;">  </div>
2.	Any element can be accessed by sequential access only.	Any element can be accessed randomly i.e. with the help of index of the array.
3.	Physically the data can be deleted.	Only logical deletion of the data is possible.
4.	Insertions and deletion of data is easy.	Insertions and deletion of data is difficult.
5.	Memory allocation is dynamic. Hence developer can allocate as well as deallocate the memory. And so no wastage of memory is there.	The memory allocation is static. Hence once the fixed amount of size is declared then that much memory is allocated. Therefore there is a chance of either memory wastage or memory shortage.

Major differences between array and linked-list are listed below:

1. **Size:** Since data can only be stored in contiguous blocks of memory in an array, its size cannot be altered at runtime due to the risk of overwriting other data.

However, in a linked list, each node points to the next one such that data can exist at scattered (non-contiguous) addresses; this allows for a dynamic size that can change at runtime.

2. **Memory allocation:** For arrays at compile time and at runtime for linked lists. but, a dynamically allocated array also allocates memory at runtime.
3. **Memory efficiency:** For the same number of elements, linked lists use more memory as a reference to the next node is also stored along with the data. However, size flexibility in linked lists may make them use less memory overall; this is useful when there is uncertainty about size or there are large variations in the size of data elements; Memory equivalent to the upper limit on the size has to be allocated (even if not all of it is being used) while using arrays, whereas linked lists can increase their sizes step-by-step proportionately to the amount of data.
4. **Execution time:** Any element in an array can be directly accessed with its index. However, in the case of a linked list, all the previous elements must be traversed to reach any element. Also, better cache locality in arrays (due to contiguous memory allocation) can significantly improve performance. As a result, some operations (such as modifying a certain element) are faster in arrays, while others (such as inserting/deleting an element in the data) are faster in linked lists.
5. **Insertion:** In an array, insertion operation takes more time but in a linked list these operations are fast. For example, if we want to insert an element in the array at the end position in the array and the array is full then we copy the array into another array and then we can add an element whereas if the linked list is full then we find the last node and make it next to the new node
6. **Dependency:** In an array, values are independent of each other but In the case of linked list nodes are dependent on each other. one node is dependent on its previous node. If the previous node is lost then we can't find its next subsequent nodes.

Stack and Static Implementation & Application

A stack is a **non-primitive linear data structure**. It is an ordered list in which addition of new data item and deletion of already existing data item is done from only one end, **known as top of stack (TOS)**. As all the deletion and insertion in a stack is done from top of the stack, the last added element will be the first to be removed from the stack. That is the reason why stack is also **called Last-in-First-out (LIFO) type of list**. The most frequently accessible element in the stack is the top most elements, whereas the last accessible element is the bottom of the stack.

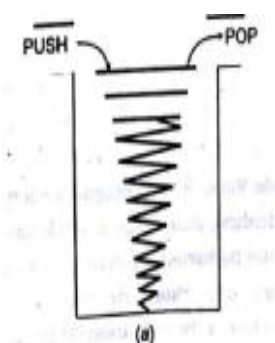
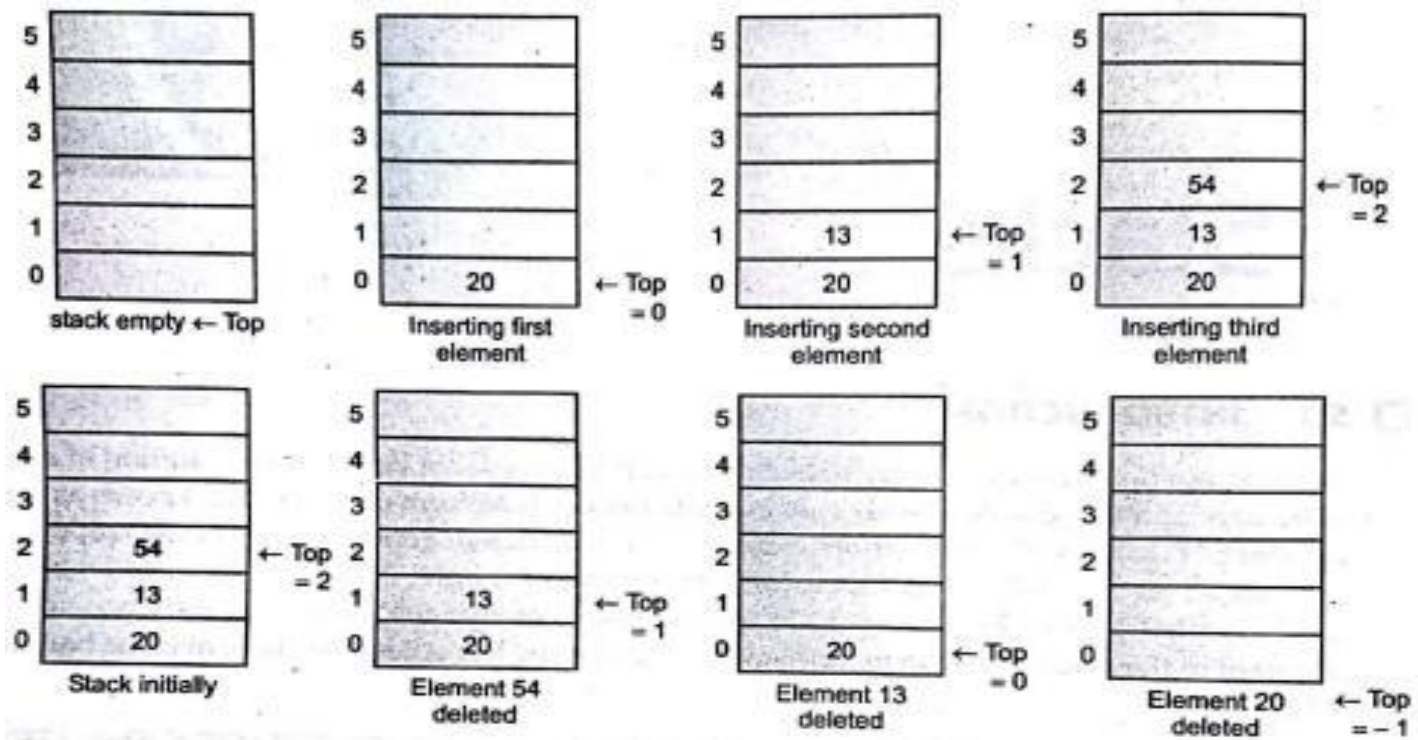


Figure 22 stack

Example 1: A common model of a stack is plates in a marriage party or coin stacker. Fresh plates are “pushed” onto to the top and “popped” from the top.

Example 2: Some of you may eat biscuits. If you assume only one side of the cover is torn and biscuits are taken out one by one. This is what is called popping and similarly,

if you want to preserve some biscuits for some time later, you will put them back into the pack through the same torn end called pushing.



(a) Stack top Increases during Insertion. (b) Stack top decreases during deletion.
 Fig. 23 shows various stages of stack top, during insertion and during deletion.

Whenever a stack is created, the stack base remains fixed, as a new element is added to the stack from the top, the top goes on increasing; conversely as the top most element of the stack is removed the stack top is decrementing. Shown above in example.

Stack static Representation

Stack can be implemented in two ways:

1. Static implementation.
2. Dynamic implementation.

Static implementation – Using Array & structure

Static implementation **uses arrays to create stack**. Static implementation though a very simple technique but is **not a flexible way** of creation, as the **size** of stack has to be **declared during program design**, after that the **size cannot be varied**. Moreover static implementation is **not too efficient** with respect to memory utilization. As the declaration of array (for implementing stack) is done before the start of the operation (at program design time), now if there are too **few elements to be stored** in the stack the statically allocated **memory will be wasted**, on the other hand if there are more number (if elements to be stored in the stack then we can't be able to change the size of array to increase its capacity, so that it can accommodate new elements.

Declaration 1:

```
# define size 100
int stack[size], top=-1;
```

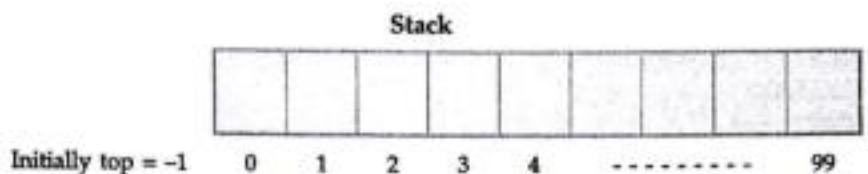


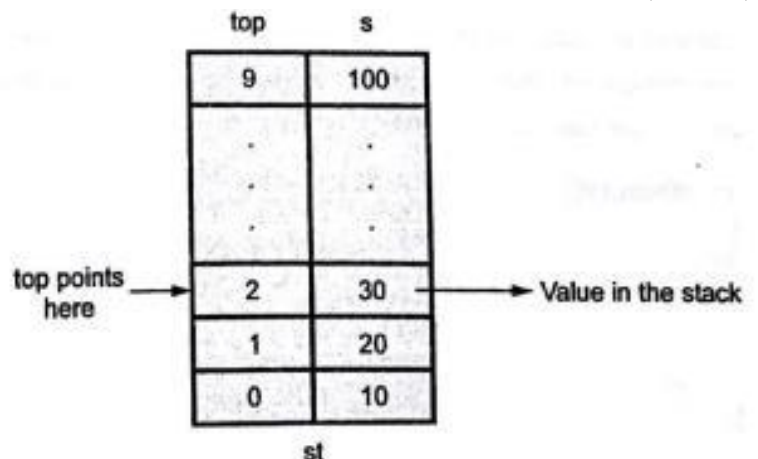
Figure 24. Stack using one dimensional array

The stack is nothing but an array of integers. And most recent index of that array will act as a top. The stack is of the size 100. As we insert the numbers, the top will get incremented. The elements will be placed from 0th position in the stack. At the most we can store 100 elements in the stack, so at the most last element can be at (size-1) position, i.e., at index 99.

Declaration 2:

```
# define size 100
Struct stack
{
int s[size];
int top;
} st1;
```

Figure 25. stack using structure



In the above declaration stack is declared as a structure. Now compare declaration 1 and 2. Both are for stack declaration only. But the second declaration will always prefer. Because in the second declaration we have used a structure for stack elements and, top. By this we are binding or co-relating top variable with stack elements. Thus top and stack are associated with each other by putting them together in a structure. The stack can be passed to the function by simply passing the structure variable.

ALGORITHMS FOR PUSH - Algorithm for inserting item into the Stack (PUSH)

PUSH (stack[Maxsize], Item): This algorithm inserts an item at the top of the Stack

Step 1: Initialize the **Maxsize** and set **top = - 1** to show stack is empty

Step 2: Insert the element in stack Repeat steps until **top <= Maxsize - 1**

2.1. Read, **Item** in stack

2.2. Increase top by 1, Set **top = top + 1**

2.3. Set top at current inserted item of the stack, Set **Stack [top] = Item**

Step 3: Insertion after condition get false, Print "**Stack overflow**"

The function for the stack push operation in C is as follows- Considering stack declared as **int Stack [5], top -1;**

```
void push( )
{
int item ;
if (top<4)
{
printf ("Enter the no") ;
scanf ("%d", & item) ;
top = top + 1 ;
stack [top] = item ;
}
else
printf ("Stack overflow") ;
}
```

ALGORITHMS FOR POP - Algorithm for deleting an item from the Stack (POP)

POP (Stack[Maxsize], Item). This algorithm delete item from the top of stack.

Step 1: Remove the element form stack and repeat steps until **top** ≥ -1

1.1. Remove element from top of stack, Set **item** = **Stack [top]**

1.2. After removing element decrement top by 1, Set **top** = **top - 1**

1.3. When element is removed Print, Number deleted is **Item**

Step 2: When all element is deleted and condition is false Print “**stack under flow**”.

This function of Pop operation is given as :

```
void pop( )
{
    int item ;
    if (top >= 0)
    {
        item = stack [top] ;
        top = top - 1;
        printf (“No deleted is = %d”, item) ;
    }
    else
    { printf ("stack is empty") ; }
}
```

Stack - Dynamic Implementation & Application

Dynamic Implementation – Using Structure and Pointer

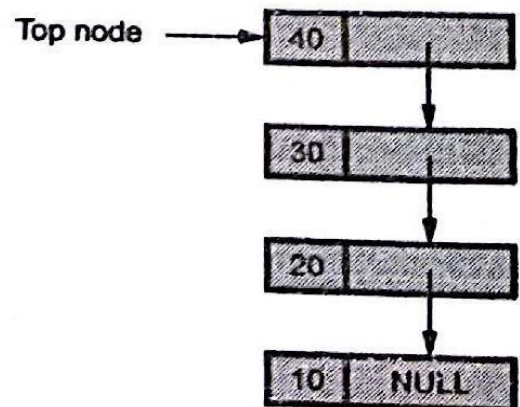
Dynamic implementation is also called **linked list representation** and **uses pointers** to implement the stack type of data structure. Stack is a **special case of list** and therefore we can **represent stack using arrays as well as using linked list**. The advantage of implementing stack using linked list is that we need **not have to worry about the size** of the stack. Since we are using linked list as many elements we want to inset those many nodes can be created. And the nodes are dynamically getting created so there **won't be any stack full condition**. The typical 'C' structure for linked stack can be

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

Figure 26. Representing linked stack

Each node consists of data and the next field. Such a node will be inserted in the stack. Fig. 26 represents stack using linked list. There are various operations that can be performed on the linked stack. These operations are:

1. Push
2. Pop
3. Stack empty
4. Stack full



OPERATIONS ON STACK - The basic operations that can be performed on stack are as follows:

1. **Create a stack** - The create function creates a stack in the memory. Creation of stack can be either arrays or linked list.
2. **Insert an element onto the stack – PUSH** - The insert function inserts a new element at the top of the stack.

Push - The process of adding a new element to the top of stack is called PUSH operation. Pushing an element in the stack invoke adding of element, as the new element will be inserted at the top after every push operation the top is incremented by one. In case the array is full and no new element can be accommodated, it is called STACK-FULL condition. This condition is called **STACK OVERFLOW**.

3. **Delete an element from the stack – POP** - The delete function removes that element which is at the top of the stack.

POP - The process of deleting an element from the top of stack is called POP operation. After every pop operation the stack is decremented by one. If there is no element on the stack and the pop is performed then this will result into **STACK UNDERFLOW** condition.

4. **Check which element is at the top of the stack** - Many times we need to test for the topmost element of the stack and to test whether the stack is empty or nonempty before we make subsequent operations. The function which test for the emptiness of the stack, say, '**stempty**' is a Boolean function which **returns true** if the **stack is empty** and **false otherwise**.

Initially stack is empty and the **top should be initialized** to **-1 or 0**. If we set top to **-1** initially then the stack will contain the elements from **0th position** and if we set top to **0 initially**, the elements will be stored from **1st position**, in the stack. Elements may be pushed onto the stack and there may be a case that all the elements are removed from the stack. Then the **stack becomes empty**. Thus whenever **top reaches to -1** we can say the stack is empty.

If **Top == -1** it means **Stack is empty**.

```
int stempty()
{
    if (sttop==-1)
        return 1;
    else
        return 0;
}
```

5. **Check whether a stack is empty or not** - A function, say, '**stfull**' is a Boolean function which tests whether the stack is full or not. It **returns true if it is full** and **false otherwise**. It is good to test whether stack is full or not before inserting a new element similarly before removing the element from stack one should check whether it is empty or not.

In the representation of stack using arrays, size of array means size of stack. As we go on inserting the elements the stack gets filled with the elements. So it is necessary

before inserting the elements to check whether the stack is full or not. Stack full condition is achieved when stack reaches to maximum size of array. If **top>=maxsize-1** or **top>=maxsize** it means stack is full.

```
int stfull()
{
    if (sttop>=maxsize-1)
        return 1;
    else
        return 0;
}
```

Stack terminology

1. **MAXSIZE** - This term is not a standard one; we use this term to refer the maximum size of stack.
2. **TOP** - This term refers to the top of stack (TOS). The stack top is used to check stack overflow or underflow conditions. Initially Top stores -1. Top is first incremented and then the item is inserted into the location currently indicated by the Top.
3. **Stack** - It is an array of size MAXSIZE.
4. **Stack empty or underflow** - This is the situation when the stack contains no element. At this point the top of stack is present at the bottom of the stack.
5. **Stack overflow** - This is the situation when the stack becomes full, and no more elements can be pushed onto the stack. At this point the stack top is present at the highest location of the stack.

ALGORITHMS FOR PUSH - Algorithm for inserting item into the Stack (PUSH)

PUSH : This algorithm inserts an item at the top of the Stack

Step 1: Using structure create node by defining data and pointer

Step 2: Insert the element in stack Repeat steps until you want to add element

- 1.1. Node is dynamically created
- 1.2. Read, **Item** in node data field
- 1.3. Set top at current inserted node

Step 2: Newly inserted node pointer address is initialized by null (node *top = NULL) and pass the address of newly created node to previous node address field.

The function for the stack push operation in C is as follows- Considering stack declared as

```
struct node
{
    int data;
    struct node *next;
};
struct node *top = NULL;
void push(int item)
{
    struct node *nptr = malloc(sizeof(struct node));
    nptr->data = item;
    nptr->next = top;
    top = nptr;
}
```

ALGORITHMS FOR POP - Algorithm for deleting an item from the Stock (POP)

Step 1: Check top is == Null, if yes stack is empty and stop program otherwise go to steps 2

Step 2: Create a new pointer *temp and assign top node to temp pointer.

- 1.1. Assign top to next node (node after deleted node) and print data deleted.
- 1.2. Free temp pointer
- 1.3. Repeat these steps till all data items get deleted

This function of Pop operation is given as :

```
void pop()
{
if (top == NULL)
{
printf("\n\nStack is empty ");
}
else
{
struct node *temp;
temp = top;
top = top->next;
printf("\n\n%d deleted", temp->data);
free(temp);
}
}
```

To display the new node and to check Inserted/deleted item use Display() function

```
void display()
{
struct node *temp;
temp = top;
while (temp != NULL)
{
printf("\n%d", temp->data);
temp = temp->next;
}
}
```

Applications Of Stacks

There are a number of applications of stacks, some of them are discussed briefly in this sections.

1. Reversing a String
2. Expression conversion
3. Expression evaluation
4. Parsing well-formed parenthesis
5. Decimal to binary conversion
6. Storing function calls

Evaluation Of Expression - postfix, Infix, prefix expression

There are basically **three types of notations** for an expression (mathematical expression; An expression is defined as a number of operand or data items combined using several operators.

1. **Infix notation**
2. **Prefix notation**
3. **Postfix notation**

Infix notation - The infix notation is what we come across in our general mathematics, where the operator is written in between the operands. For example - The expression to add two numbers A and B is written in infix notation as: **A + B**

Note that the operator '+' is written in-between the operands A and B. That's why this notation is called infix.

Prefix notation - The prefix notation is a notation in which the operator is written before the operands, it is also **called polish notation** in the honor of the mathematician Jan Lukasiewicz who developed this notation. The same expression when written in prefix notation looks like: **+AB**

As the operator '+' is written before the operands A and B, this notation is called prefix (pre means before).

Postfix notation - In the postfix notation the operators are written after the operands, so it is called the postfix notation (post means after), it is also known as **suffix notation** or **reverse polish notation**. The above expression if written in postfix expression looks like follows: **AB+**

Advantage of using postfix notation

Although human beings are quite used to work with mathematical expressions in infix notation, which is rather complex, as using this notation one has to remember a set of non-trivial rules. That set of rules must be applied to expressions in order to determine the final value. These rules **include precedence, BODMAS, and Associativity**.

Using **infix notation**, one **cannot tell the order** in which **operators should be applied** by only looking at expression. Whenever an infix expression consists of more than one operator, the precedence rules (BODMAS) should be applied to decide that which operator are evaluated first. As compared to postfix notation, which is much easier to work with or evaluate. In postfix expression operands appear before the operators, there is no need for operator precedence and other rules. As soon as an operator appears in the postfix expression during scanning of postfix expression the topmost operands are popped off and are calculated applying the encountered operator. Place the result back onto the stack, doing so the stack will contain finally a single value at the end of process.

Notation Conversions - Let an expression $A + B * C$ is given, which is in infix notation. To calculate this expression for values 4, 3, 7 for A, B, C respectively we must follow certain rule (called BODMAS in general mathematics) in order to have right result. For example

$$A + B * C = 4 + 3 * 7 = 4 + 21 = 25$$

Is this the right result? No, this is because the multiplication is to be done before addition, because it has higher precedence over addition. This means that for expression to be calculated we must have the knowledge of precedence of operators. The error in the above calculation occurs, as there are no braces to define the precedence of operators. Thus, expression $A + B * C$ is interpreted as, $A + (B * C)$. This is an alternative for us to convey the computer that multiplication has higher precedence over addition, as there is no other way to specify this.

Operator precedence

Exponential operator	$\wedge, \$, \uparrow$	Highest precedence
Bracket	$(), [], \{ \}$	Next precedence
Division / Multiplication	$/, *$	Next precedence
Addition / Subtraction	$+, -$	Least precedence

Converting infix expression to postfix form

$A + B * C$	Infix Form
$A + (B * C)$	Parenthesized expression,
$A + (BC^*)$	Convert the multiplication
$A(BC^*) +$	Convert the addition
$ABC^* +$	Postfix form

The **rules** to be remembered during **infix to postfix** conversion are:

1. Parenthesize the expression starting from **left to right**.
2. During parenthesizing the expression, the operands associated with operator having **higher precedence are first parenthesized**.
3. The **sub-expression** (part of expression) which has been converted into postfix is to be treated as **single operand**.
4. Once due expression is **converted to postfix** form **remove the parenthesis**.

Expression Conversion - Algorithm for Conversion of **INFIX TO POSTFIX** Expression

1. Read the infix expression for left to right one character at a time.
2. If input symbol read is "(" then push it on to the stack.
3. If the input symbol read is an operand (e.g. A,B,C) then place it in postfix expression.
4. If the input symbol read is operator (e.g. +, -, \wedge) then
 - 4.1 Check if priority of operator in stack is greater than the priority of incoming (or input read) operator then pop that operator from stack and place it in the postfix expression Repeat step 4.1 till we get the operator in the stack which has greater priority than the incoming operator.
 - 4.2 Otherwise push the operator being read, onto the stack.
 - 4.3 If we read input operator as ")" then pop all the operators until we get "(" and append popped operators to postfix expression. Finally just pop ")".
5. Final pop the remaining contents, from the stack until stack becomes empty append them to postfix expression.
6. Print the postfix expression as a result.

The priorities of different operators when they are in stack will be called as in stack priorities. And the priorities of different operator when then are from input will be called as incoming priorities.

Solved Problems - Convert: $(A + (B * C - (D / E - F) * G) * H)$ into postfix form showing stack status after every step in tabular form.

SOLUTION -

Symbol Scanned	Stack	Postfix Expression
((
A	(A
+	(+	A
((+(A
B	(+(AB
*	(+(*	AB
C	(+(*	ABC
-	(+(-	ABC*
((+(-(ABC*
D	(+(-(ABC * D
/	(+(-(/	ABC * D
E	(+(-(/	ABC * DE
-	(+(-(-	ABC * DE /
F	(+(-(-	ABC * DE / F
)	(+(-	ABC * DE / F -
*	(+(-*	ABC * DE / F -
G	(+(-*	ABC * DE / F - G
)	(+	ABC * DE / F - G * -
*	(+*	ABC * DE / F - G * -
H	(+*	ABC * DE / F - G * - H *
)		ABC * DE / F - G * - H * +

CONVERTING INFIX TO PREFIX EXPRESSION

This algorithm is **bit tricky**, in this we **first reverse the input expression** $a+b*c$ will become $c*b+a$ and then we do the conversion and then again we reverse to get the result. Doing this has an advantage that except for some minor modification algorithm for infix to prefix remains almost same as the one for infix to postfix.

Algorithm

1. Add bracket in starting and end of expression & Reverse the input string.
2. Examine the next element in the input.
3. If it is operand, add it to the output string.
4. If it is **closing parenthesis**, **push** it on stack.
5. If it is an operator, then
 - 5.1. if stack is empty, push operation on stack.
 - 5.2. if the top of stack is closing parenthesis push operator on stack.
 - 5.3. If it has same or higher priority than the top of stack, push operator on output string S.
 - 5.4. Else pop the operator from the stack and add it to output string S.
6. If it is a **opening parenthesis**, **pop** operator from stack and add them to S until a closing parenthesis is encountered. POP and discard the closing parenthesis.
7. If there is more input go to step 2. If there is no more input, unstack the remaining operators and add them.
8. Reverse the output string.

Convert $(A + B * C)$ into the prefix expression.

SOLUTION: Reverse the expression $A + B * C$ we get $)C * B + A($.

Symbol Scanned	Stack	Postfix Expression
))	~
C)	C
*)*	C
B)*	CB
+)+	CB*
A)+	CB*A
(-	CB * A +

So postfix expression is $CB*A+$. Now reverse it to get **final result** i.e. $+A*BC$

ALGORITHM TO **EVALUATE A POSTFIX EXPRESSION**

This algorithm finds the **VALUE** of an arithmetic expressions P written in postfix notation. The following algorithm, which uses a **STACK** to hold operands, evaluates P .

1. Add a parenthesis “(”, “)” at the start and end of P .
2. Scan P from left to right and repeat steps 3 and 4 for each element of P until the “)” is encountered.
3. If an operand is encountered, put it on **STACK**.
4. If an operator is encountered, then :
 - 4.1 Remove the two top elements of **STACK**, where A is the top element and B is the next-to-top element.
 - 4.2 Evaluate B and A .
 - 4.3 Place the result back on **STACK**.
 - 4.4 End of If statement & repeat Step 2.
5. Set **VALUE** equal to the top element on **STACK**.
6. Exit.

Queue and Static Implementation & Application

A queue is logically a **First in first out (FIFO) type** of list. In our day-to-day life we come across many situations where we ought to wait for the desired service, there we have to get into a waiting line for our turn to get serviced. This waiting queue can be thought of as a Queue. **Queue means a line**, at the railway reservation booth, we have to get into the reservation queue.

The important feature of the queue - new customers got into the queue from the rear end, whereas the customers who get their seats reserved leave the queue from the front end. It means the customers are serviced in the order in which they arrive at the service center i.e. first come first serve (FCFS) type of service.

A queue is a **non-primitive linear data structure**. It is an homogeneous collection of elements in which new elements are added at **one end called the Rear end** and the existing elements are deleted from **other end called the Front end**. The above figures show queue graphically during insertion operation. It is clear from the above figures that whenever we **insert an element in the queue**, the value of **Rear** is incremented by one i.e. **Rear = Rear + 1**

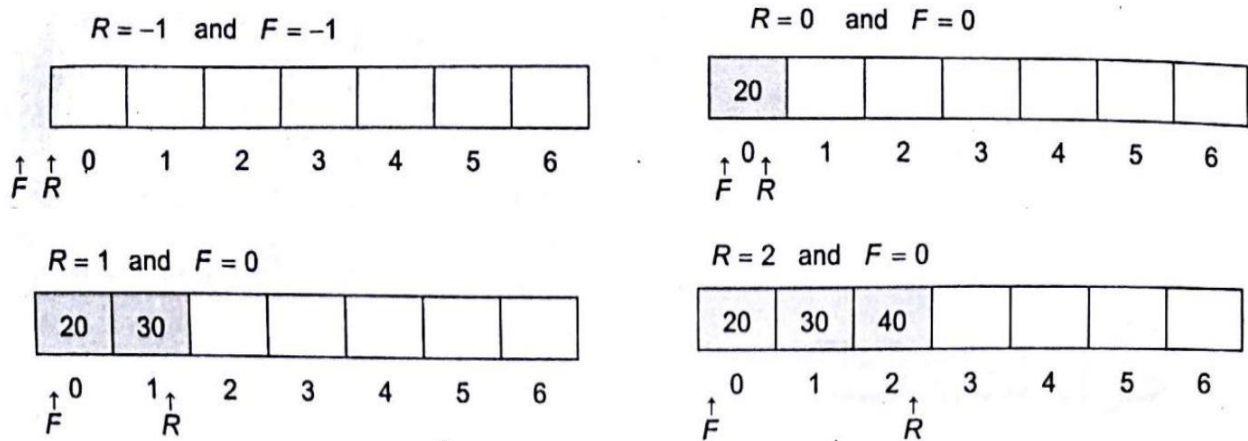


Figure 27. (a) Empty Queue (b) Queue with one element.
(c) Queue with two elements (d) Queue with three elements

Note that during the **insertion of the first element** in the queue we always increment the Front by one i.e.: **Front = Front + 1**. Afterwards the Front will not be changed during the entire addition operation. The following figure shows queue graphically during deletion operation:

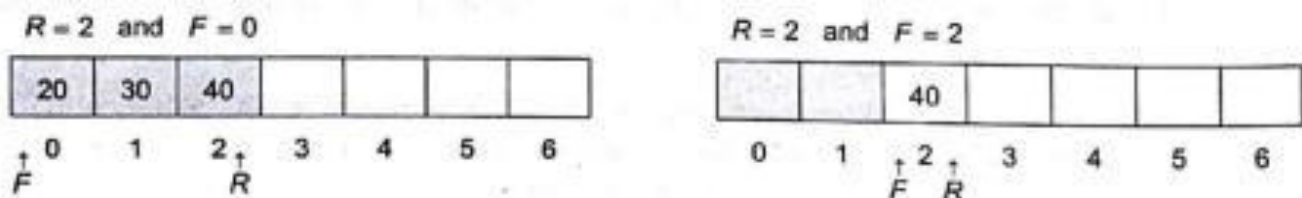


Figure 28. (a) Queue with three elements (b) Two elements deleted from front (20 & 30)

Now if we insert two elements in the queue, the queue will look like: $R = 4$ and $F = 2$ shown in Figure 29.

This is clear from the above figure 8 that whenever **an element is removed or deleted from the queue** the value of Front is incremented by one i. e. **Front = Front + 1**

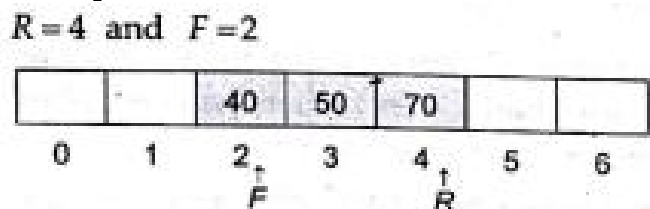


Figure 29

Advantages of Queue over Stack

1. The first and most important advantage of queue over stack is that there are two different pointers maintained for handling queues. That is front and rear.
2. In stack the element which is inserted first will be removed at the last. That means the first element has to stay in the stack for a long time. But in queue the element which is inserted first will get removed first.
3. Using queue we can maintain the priority of the elements, which is not possible using stack.
4. In certain type of queue such as deque both the ends can be utilized for insertion and deletion of elements.

As other lists queues can also be implemented in **two ways**:

1. **Static implementation** (using arrays & structure).
2. **Dynamic implementation** (using pointers & structure).

If Queue is Implemented using arrays, we must be sure about the **exact number of elements** we want to store In the queue, because we have to declare the size of the array at design time or before the processing starts in this case, the **beginning of the array** will become the **front** for the queue and the **last location** of the array will **act as rear** for the queue. The following relation gives the total number of elements present in the queue, when implemented using arrays: **front - rear + 1**

If **rear < front** then there will be **no element** in the **queue** or **queue** will always be **empty**.

ALGORITHMS FOR INSERTION AND DELETION IN QUEUE (USING ARRAYS) – Static Implementation

Let Queue be the array of some specified size say - MAXSIZE, then the insertion algorithm will be as given below. While implementing a queue using arrays we must check underflow and overflow conditions for queue.

Algorithm for insertion in a Queue

Queue is declared as int queue [5], **Front = -1 and rear = -1**. This algorithm inserts an Item at the rear of the queue (maxsize).

QINSERT (queue [maxsize], item)

Step 1: Initialization of

Set Front = -1

Set Rear = -1

Step 2: Repeat steps 3 to Until Rear <= maxsize - 1.

Step 3: Read item

Step 4: if Front == -1 then

Front = 0

Rear = 0

else

Rear = rear + 1

endif

Step 5: Set Queue [Rear] = Item

Step 6: Print. Queue overflow if step 2 condition failed

Function

```
void queue( )
{
    int item
    if (rear < maxsize -1)
    {
        printf ("Enter the number") ;
        scanf ("%d" & item) ;
        if (front == -1)
        { front = 0 ; rear = 0 ; }
        else
        { rear = rear+1; }
        queue [rear] = item ;
    }
    else
```



```
printf("queue is full");
}
```

Algorithm for Deletion From a Queue

This algorithm deletes an item at the front of the Queue [MaxSIZE]

QDELETE (queue, [maxsize], item)

Step 1: Repeat stages 2 to 4 until Front ≥ 0

Step 2: Set item = queue [Front]

Step 3: If Front == real

Set Front = -1

Set Rear = -1

else

Front = Front + 1

endif

Step 4: Print. No Deleted is. Item

Step 5: Print, "queue is Empty"

Function

```
void delete( )
{
int item ;
if (front != -1)
{
item = queue[front] ;
if (front == rear)
{ front = -1 ; rear = -1 ; }
else
front = front + 1 ;
printf ("No deleted is = %d", item) ;
}
else
printf ("Queue is empty") ;
}
```

LIMITATIONS OF SIMPLE QUEUES

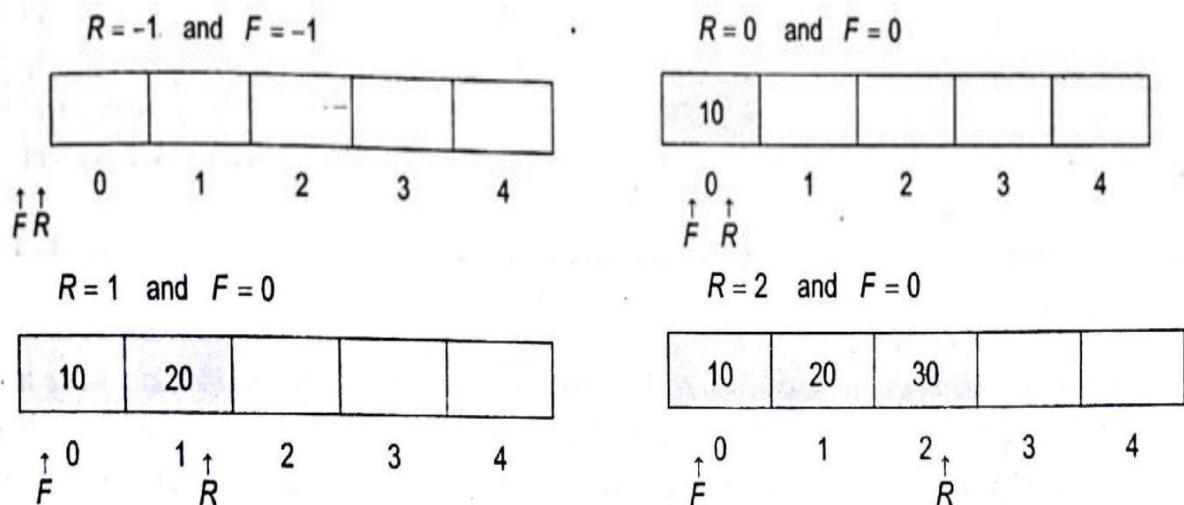


Figure 30. (a) Initial Queue (b) One Element Queue
(c) Two Element Queue (d) Three Element Queue

There are certain problems associated with a simple queue when queue is implemented using arrays as we have just studied. Consider an example of a simple queue $Q[5]$, which is initially empty. We analyze the problem with a series of insertion and deletion operations performed on the queue. The insertion and deletion processes are shown in the following Figures.

Up to now there is no problem; the problem arises when we delete an element from the queue.

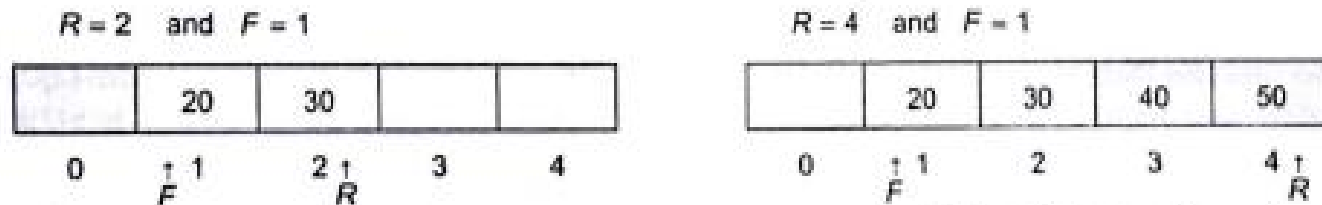


Figure 31. (a) After Deleting an Element (b) After Adding two Elements

If we further add new items to the queue, the queue looks like as given in figure below (figure 11). Now if we attempt to add more elements, **Problem arise** that the elements can't be inserted, because in a queue the new elements are always added from the rear end, and rear here indicates to last location of the array (location with subscript 4). Hence though the space is there in the queue we are not able to insert the elements (if we try to do so the program will show the message "Queue Is Full" though it is empty).

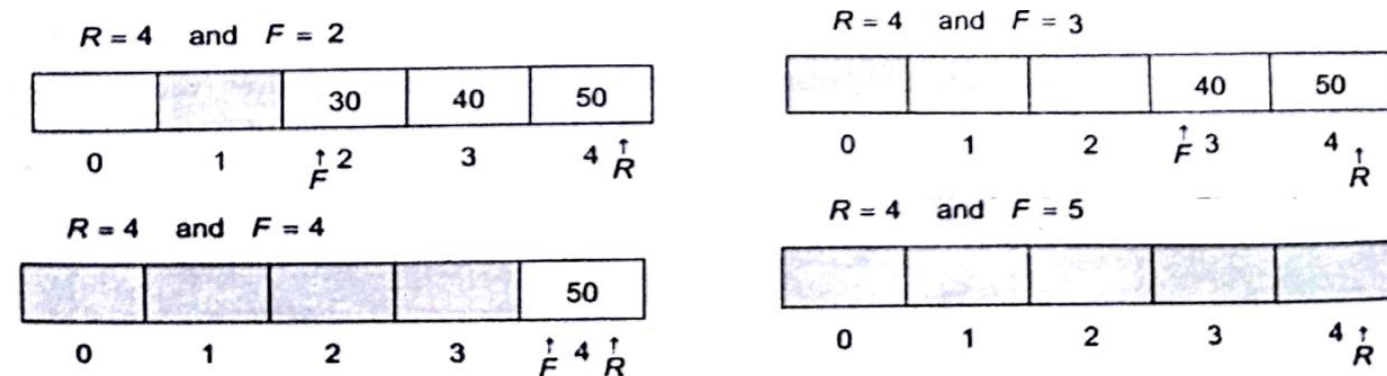


Figure 32. (a) After Deleting an Element (b) After Deleting an Element (c) Alter Deleting an Element in (d) After Deleting an Element

To **remove this problem**, the **first solution is**, which comes into our mind is whenever an element is deleted, shift all afterward elements to the left by one position. But if the queue is too large of say 5000 elements it will be a difficult job to do and time consuming too. To remove this problem we use circular queue. Now if $\text{Front} = 5$, this takes Front pointer out. To take this pointer again pointing to the right position in the queue we always reset (or check) both the pointers Front and Rear, this step is highlighted in the deletion algorithm given above.

Queue and Dynamic Implementation & Application

Implementing queues using pointers, the main **disadvantage** is that a node in a linked representation (using pointers) occupies more memory space than a corresponding element in the array representation, since there are at least two fields in each node one the data field and the other to store the address of next node, whereas in a array representation only data field is there. But note that the memory space occupied by a node in a linked representation is not exactly twice the space used by an element of array representation. **Advantage** of a linked representation makes an effective

utilization of memory. **Another advantage** of linked representation becomes apparent when it is needed to insert or delete an element in the middle of a group of other elements.

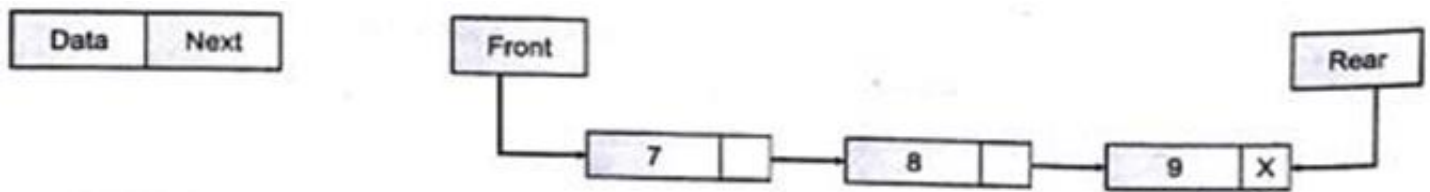


Figure 33. (a) A Node

(b) A Queue

Suppose we wish to insert an element between the second and third element in the array of size 10, which already contains eight elements ($a[0]$ to $a[7]$). This requires us to shift one place, all the elements from third onwards ($a[2]$ to $a[7]$) so that a slot should be made empty to insert new element. It means we have to shift five elements. If the array is of large size say 1000 or 2000 elements, it would be equivalent of doing a bulk of work and gaining no special results. Similarly when element is deleted between array, it requires all the elements beyond the deleted element to be shifted, so as to fill the gap created by deletion of element. On the other hand linked representation allows us to stress on our primary aim (i.e., addition or deletion) and not on the overheads related to these processes.

In other words the amount of work required is independent of the size of the list. Addition of a new node in between a queue in a linked representation requires creating a new node, **inserting** it in the **required** position by **adjusting two pointers**. Whereas to **delete** a node from the queue all we require is to **adjust a pointer** to point the node to be deleted, and then free that node. An individual node and queue in linked representation is shown in the figure 33.

Algorithm for adding and deleting in a Queue

Let Queue be a structure whose declaration looks as follows:

```
struct qnode
{
    int n;
    qnode *next;
} *start = NULL;
```

Algorithms for adding a node in a queue

```
Step 1: struct_queue NEWPTR, temp    [Declare variables of type struct queue]
Step 2: temp = start                  [Initialize temp]
Step 3: NEWPTR = new Node              [Allocate memory for new element]
Step 4: NEWPTR -> no = Value.          [Insert value into the data field of element]
Step 5: NEWPTR -> NEXT = NULL
Step 6: If (Rear = NULL)              [Queue is empty & element to be entered is 1st element]
        Set front = NEWPTR
        Set rear = NEWPTR
    else
        while (temp -> next != NULL)
```

```
temp = temp -> next
Step 7: temp -> next =- NEWPTR
Step 8: end
```

Algorithm for deleting a node from a queue

```
Step 1: if (front = NULL)
    Write Queue Is Empty and Exit
else
    temp = Start
    value = temp —> no
    Start = Start —> next
    free (temp)          [End else]
    return (value)       [End if]
Step 2: Exit
```

Note that we don't need to check overflow condition in dynamic implementation during addition operation. Though overflow can occur in this implementation but it occurs almost never. Only the underflow condition is to be checked during deletion operation.

Operation On Queue

The basic operations that can be performed on queue are:

1. To insert an element in a queue.
2. To delete an element from the queue.
3. Queue overflow.
4. Queue underflow.
5. Display of the queue.

Applications of Queue:

1. **Multi programming:** Multi programming means when multiple programs are running in the main memory. It is essential to organize these multiple programs and these multiple programs are organized as queues.
2. **Network:** In a network, a queue is used in devices such as a router or a switch. another application of a queue is a mail queue which is a directory that stores data and controls files for mail messages.
3. **Job Scheduling:** The computer has a task to execute a particular number of jobs that are scheduled to be executed one after another. These jobs are assigned to the processor one by one which is organized using a queue.
4. **Shared resources:** Queues are used as waiting lists for a single shared resource.

Real-time application of Queue:

1. ATM Booth Line
2. Ticket Counter Line
3. Key press sequence on the keyboard
4. CPU task scheduling
5. Waiting time of each customer at call centers.

Advantages of Queue:

1. A large amount of data can be managed efficiently with ease.
2. Operations such as insertion and deletion can be performed with ease as it follows the first in first out rule.
3. Queues are useful when a particular service is used by multiple consumers.
4. Queues are fast in speed for data inter-process communication.
5. Queues can be used in the implementation of other data structures.

Disadvantages of Queue:

1. The operations such as insertion and deletion of elements from the middle are time consuming.
2. Limited Space.
3. In a classical queue, a new element can only be inserted when the existing elements are deleted from the queue.
4. Searching an element takes $O(N)$ time.
5. Maximum size of a queue must be defined prior.

Circular Queues & Application

A circular queue is one in which the insertion of a new element is done at the very first location of the queue if the last location of the queue is full. In other words if we have a queue Q of say n elements, then after inserting an element last (i.e., in the $n-1$ th) location of the array the next element will be inserted at the very first location (i.e., location with subscript 0) of the array. It is possible to insert new elements, if and only if those locations (slots) are empty. We can say that a circular queue is one in which the first element comes just after the last element. It can be viewed as a mesh or loop of wire, in which the two ends of the wire are connected together. Fig 34 shows a empty circular queue $Q[5]$ which can accommodate five elements.

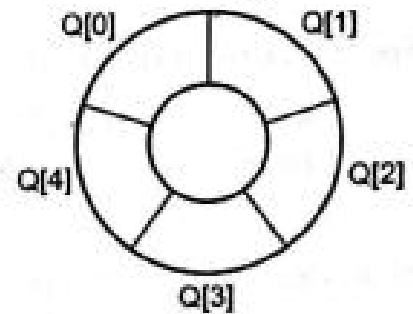


Figure 34. Circular Queue

A circular queue overcomes the problem of unutilized space in linear queues implemented as arrays. A circular queue also has a Front and Rear to keep the track of the elements to be deleted and inserted and therefore to maintain the unique characteristic of the queue. The below assumptions are made:

1. Front will always be pointing to the first element (as in the linear queue).
2. If $\text{Front} = \text{Rear}$ the queue will be empty.
3. Each time a new element is inserted into the queue the Rear is incremented by one.
 $\text{Rear} = \text{Rear} + 1$.
4. Each time an element is deleted from the queue the value of Front is incremented by one.
 $\text{Front} = \text{Front} + 1$

Insertion

Consider the circular queue shown above, the insertion in this queue will be same as with linear queue, we only have to keep track of Front and Rear with some extra logic.

A Circular Queue is shown in the figure 35(a). If more elements are added to the queue, it looks like shown in figure (b).

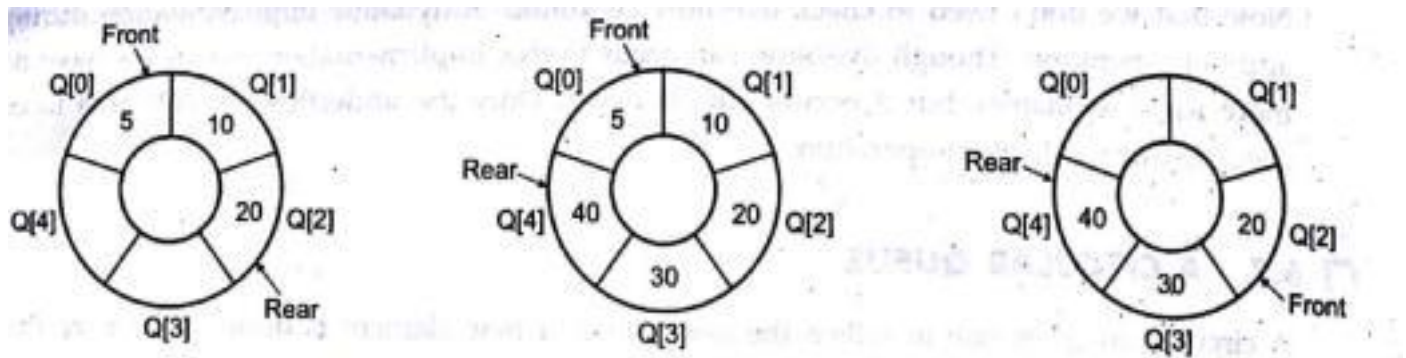


Figure 35. (a)(b)(c) Insertion in a queue

Now the queue will be full. If we now try to add another element to the queue, as the new element is inserted from the Rear end, the position of the element to be inserted will be calculated by the relation:

$$\text{Rear} = (\text{Rear} + 1) \% \text{MAXSIZE}$$

$$\text{Queue}[\text{Rear}] = \text{value}$$

For the current queue Fig 35(b) the value of Rear is 4, and value Of MAXSIZE is 5, hence

$$\text{Rear} = (\text{Rear} + 1) \% \text{MAXSIZE}$$

$$\text{Rear} = (4+1) \% 5 = 0$$

Note that Front is also pointing to Q[0] (Front = 0) and Rear also comes out to be 0 (i.e., Rear is also pointing to Q[0]). Since Rear = Front, the Queue Overflow condition is satisfied, and we try to add new element will flash the message “Queue Overflow” which avoids us to add new element.

Consider the Fig 35(c). Consider the situation when we add, an element to the queue. The Rear is calculated as

$$\text{Rear} = (\text{Rear} + 1) \% \text{MAXSIZE}$$

$$\text{Rear} = (5+1) \% 5 = 0$$

The new element will be added to Q[Rear] or Q[0] location of the array and Rear is increased by one (i.e., Rear = Rear + 1 = 0 + 1) The next element will be added to location Q[1] of the array.

Deletion

The deletion method for a circular queue also requires some modification as compared to linear queues. The deletion is explained in Fig 36. In the Fig 36 the queue is full. Now if we delete one element from the queue, it will be deleted from the **Front end**. After deleting the front element, the front should be modified according to position of Front, i.e., if Front indicates to the last element of the circular queue then after deleting that element the Front should be again reset to 0 (Front = 0). Otherwise after every deletion the new position which Front should indicate will be as:

$$\text{Front} = (\text{Front} + 1) \% \text{MAXSIZE}$$

Algorithms for Addition and Deletion in a Circular Queue (Using Arrays)

Algorithm for **adding an element** in a circular queue

Step 1: If $(\text{Front} == (\text{Rear} + 1) \% \text{MAXSIZE})$ write Queue Overflow and Exit.

Else: Take the value

If $(\text{Front} == -1)$

Set $\text{Front} = \text{Rear} = 0$

Else

$\text{Rear} = ((\text{Rear} + 1) \% \text{MAXSIZE})$ [Assign Value]

Queue $[\text{Rear}] = \text{value}$.

[End if]

Step 2: Exit.

Algorithm for **deleting an element** in a circular queue

Step 1: If $(\text{Front} == -1)$

write Queue underflow and Exit.

Else :

Item = Queue[Front]

if $(\text{Front} == \text{Rear})$ Set $\text{Front} = -1$ Set $\text{Rear} = -1$

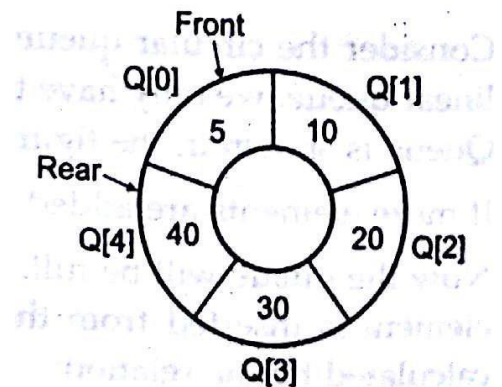
Else :

$\text{Front} = (\text{Front} + 1) \% \text{MAXSIZE}$

[End If]

Step 2: Exit.

Figure 36. Queue deletion



Applications Of a Circular Queue

1. Memory management: circular queue is used in memory management.
2. Process Scheduling: A CPU uses a queue to schedule processes.
3. Traffic Systems: Queues are also used in traffic systems.