UNIT1

INTRODUCTION TO DATA STRUCTURES

Basic Concepts

- The term data structure is used to describe the way data is stored, and the term algorithm is used to describe the way data is processed. Data structures and algorithms are interrelated. Choosing a data structure affects the kind of algorithm you might use, and choosing an algorithm affects the data structures we use.
- An Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions.

Introduction to Data Structures:

- Data structure is a representation of logical relationship existing between individual elements of data. In other words, a data structure defines a way of organizing all data items that considers
- not only the elements stored but also their relationship to each other. The term data structure is used to describe the way data is stored.
- To develop a program of an algorithm we should select an appropriate data structure for that algorithm. Therefore, data structure is represented as:

Algorithm + Data structure = Program

- A data structure is said to be **linear** if its elements form a sequence or a linear list. The linear data structures like an array, stacks, queues, and linked lists organize data in linear order.
- A data structure is said to be **non-linear** if its elements form a hierarchical classification where, data items appear at various levels. Trees and Graphs are widely used non-linear data structures. Tree and graph structures represents hierarchical relationship between individual data elements. Graphs are nothing but trees with certain restrictions removed.
- Data structures are divided into two types:

- Primitive data structures.
- Non-primitive data structures.

Primitive Data Structures are the basic data structures that directly operate upon the machine instructions. They have different representations on different computers. Integers, floating point numbers, character constants, string constants and pointers come under this category.

Non-primitive data structures are more complicated data structures and are derived from primitive data structures. They emphasize on grouping same or different data items with relationship between each data item. Arrays, lists and files come under this category. Figure below shows the classification of data structures.

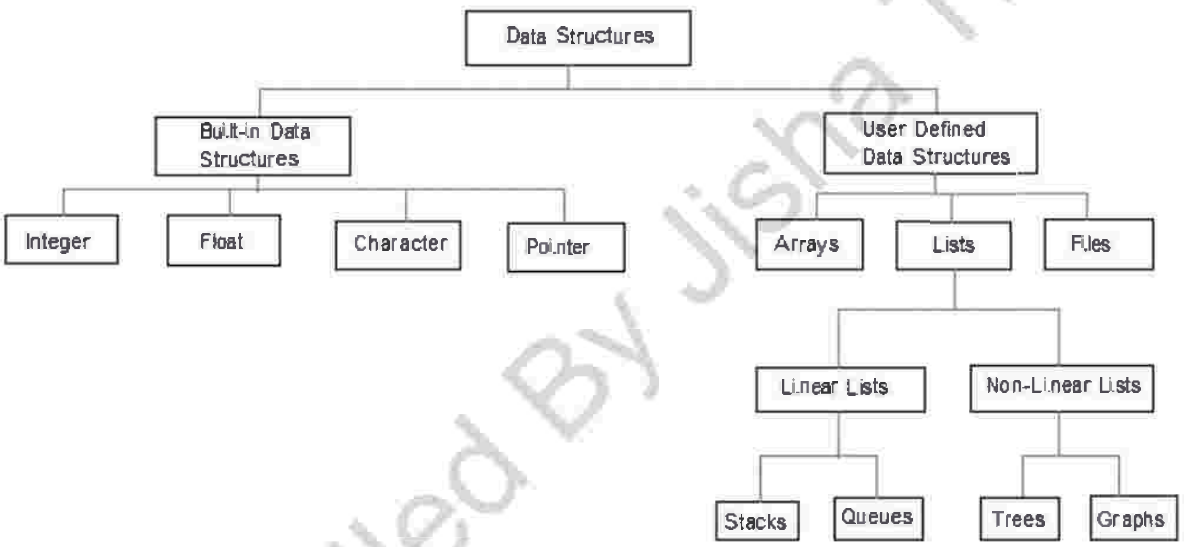


Figure 1. 1. Classification of Data Structures

Data structures: Organization of data

- The collection of data you work with in a program have some kind of structure or organization. No matter how complex your data structures are they can be broken down into two fundamental types:
    - Contiguous
    - Non-Contiguous.

- In contiguous structures, terms of data are kept together in memory (either RAM or in a file). An array is an example of a contiguous structure. Since each element in the array is located next to one or two other elements.
- In contrast, items in a non-contiguous structure and scattered in memory, but will be linked to each other in some way. A linked list is an example of a non-contiguous data structure. Here, the nodes of the list are linked together using pointers stored in each node. Figure below illustrates the difference between contiguous and noncontiguous structures.

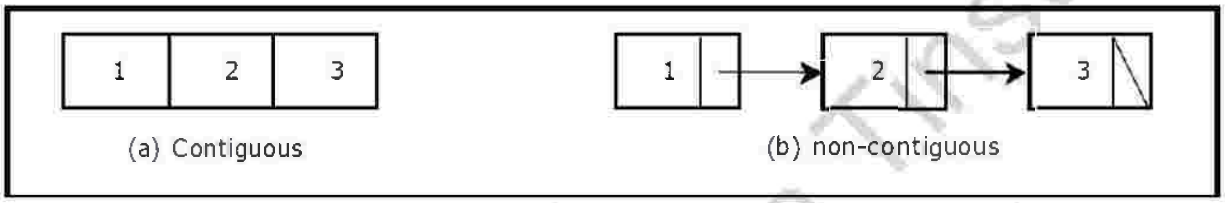


Figure 1.2 Contiguous and Non-contiguous structures compared

Need of data structure

- It gives different level of organization data.
- It tells how data can be stored and accessed in its elementary level.
- Provide operation on group of data, such as adding an item, looking up highest priority item.
- Provide a means to manage huge amount of data efficiently.
- Provide fast searching and sorting of data.

ABSTRACT DATA TYPE

- It can be defined as a collection of data items together with the operations on the data. The word "abstract" refers to the fact that the data and the basic operations defined on it are being studied independently of how they are implemented.
- An implementation of ADT consists of storage structures to store the data items and algorithms for basic operation.
- An abstract data type is a type with associated operations, but whose representation is hidden.
- The idea of an ADT is to separate the notions of specification (what kind of thing we're working with and what operations can be performed on it) and implementation (how the thing and its operations are actually implemented).

- There are two views of an abstract data type in a procedural language like C. One is the view that the rest of the program needs to see: the names of the routines for operations on the data structure, and of the instances of that data type. The other is the view of how the data type and its operations are implemented.
- The advantages of ADT are that the code becomes more readable, its implementation can be changed for better efficiency without affecting the rest of the code and they can be reused in different programs.
- Common ADTs are stacks, queues, binary trees, etc.
- For example, a stack uses the LIFO mechanism for storing and removing the data. The last element that is inserted is the first element to be removed from the stack. Common operations are pushing an element on top of the stack, popping the last element inserted, finding/seeking the current top of the stack, etc. These functions or operations can be defined first and then these operations can be used whenever required in the program.
- The implementation of a stack can be by either an array or a linked list. It doesn't affect the logic of the rest of the program.

Operations on data structure

1) **Traversing:**

- Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.
- Example: If we need to calculate the average of the marks obtained by a student in 6 different subject, we need to traverse the complete array of marks and calculate the total sum, then we will divide that sum by the number of subjects i.e. 6, in order to find the average.

2) **Insertion:**

- Insertion can be defined as the process of adding the elements to the data structure at any location.
- If the size of data structure is n then we can only insert n-1 data elements into it.

3) **Deletion:**

- The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.
- If we try to delete an element from an empty data structure, then underflow occurs.

4) Searching:

- The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search.

5) Sorting:

- The process of arranging the data structure in a specific order is known as Sorting.
- There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

6) Merging:

- When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging.

UNIT - II

PAGE NO.
DATE

INTRODUCTION TO STACK.

Eg: Stack of Plates, CD's. (Explain)

→ Addn and deletion of elements are allowed through only one end. (TOP)

* Basic operations-

1. createStack()
2. push(item)
3. pop()
4. peek()
5. isFull()
6. isEmpty()

} Refer defn of algorithm discussed in class.

- Stack is a data structure in which addition and removal of an element is allowed at the same end called the "top" of the stack.
- Stack stores elements in an ordered manner.
- Stack is called LIFO (Last-In-First-Out) data structure, as the element that was inserted last is the first one to be taken out.

(2)

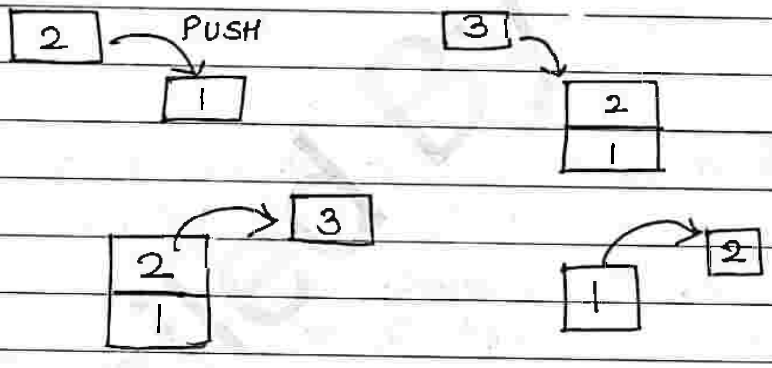
PAGE No
DATE / /

* ARRAY IMPLEMENTATION OF STACK:

Imp

Q. Explain the concept of representing stack through arrays.

- Stack is generally represented into two ways: static and dynamic
- The static implementation of stack is done using Array.
- Dynamic implementation is done using Linked List.
- Array is considered as a static data structure because the size of the array is fixed.



- Now, to implement stack using array:
  We know, stack is a data structure in which the topmost element is pointed by the integer variable "top".

Example:-

```
int a[5];
```

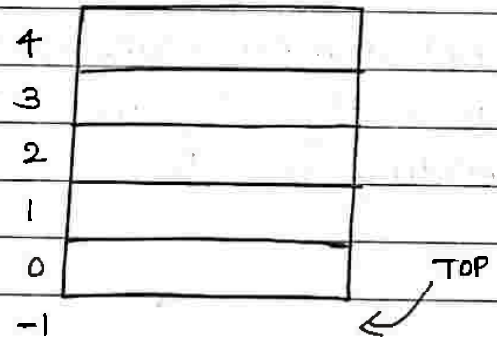
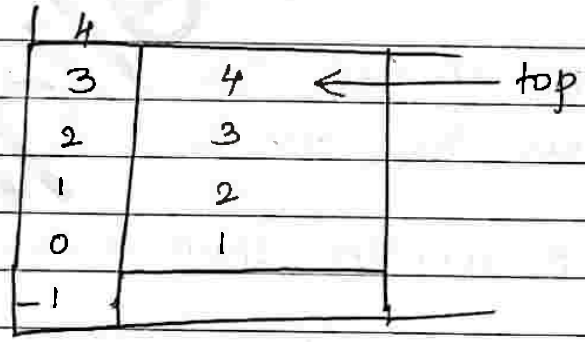


Fig. Empty Stack

→ For empty stack, top points to "-1". ie. we can consider that if top points to -1 position, means the stack is empty.

→ As elements are added in the stack, the top is incremented by one every time. Hence "top" always points to the upper (topmost) element.

→ Consider we have added 4 elements in stack, say 1, 2, 3, 4, then the structure would be —



— If the top is at size_of_array -1, then the stack is considered as full.

— Suppose we add one more element (ie. PUSH 5) Now top will be at position 4.

ie. size of the array - 1 ∴ Stack is now full.

- Similarly, we cannot remove elements from an stack if its empty. So, we will check if stack top is pointing to -1.

⇒ Initializing Stacks:-

- After stack is created, we can initialize it by setting the top at the position -1.

```
void initstack()
{
    stack [top] = -1;
}
```

➡ Inserting Element in the stack.

Q. Write a procedure to push an element onto the stack.

- The insertion of element in the stack is known as "push".
- Before inserting elements into the stack, we have to check whether the stack is full or not.

Function to check whether stack is full or not.

```
int isfull()
{
    if (top == SIZE-1)
        return (1);
    else
        return (0);
}
```

* Function to add (push) element:

```
void push()
{
  if (is_full()==1)
  {
    pf("\n\t STACK is overflow ");
  }
  else
  {
    pf(" Enter an element to add in the stack :");
    sf("%d", &ele);
    top++;
    stack[top] = ele;
  }
}
```

→ The function initially checks whether the stack is full by calling the is_full() function, and if it it full, it will print "STACK OVERFLOW".

→ In the other case, element to be inserted (pushed) will be read from the user and "top" will be incremented, to get the next location of stack and the element would be inserted at the position pointed by variable top.

PAGE NO
DATE / /

* Deleting Element from the Stack :-

Q. Explain POP operation on stack using array representation.

- The deletion of an element from the stack is known as pop.
- Before deleting an element from the stack, we have to check whether it is empty or not.
- When deleting an element, if the stack is found empty, then the situation is considered as "stack underflow."

Function to check whether stack is empty or not:

```
int is_empty ()
{
  if (top == -1)
      return (1);
  else
      return (0);
}
```

Function to delete (pop) an element:

```
void pop ()
{
  if ( is_empty () == 1)
  {
     printf ("\n \t STACK is underflow ");
  }
```

```
else
 {
   printf ("\n Element popped : %d", stack[top]);
   top--;
 }
}
```

* Displaying Element of Stack.

Q. Explain "Displaying stack elements" in detail.

- Now, to display elements from the stack, we use loop and begins from position top to position 0 in the array..
- Hence, elements are displayed in reverse order from which they are added and because of this stack is known as "LIFO" structure.
- Here also, we have to first check whether stack is empty or not.

```
void display()
 {
   int i;
   if ( is_empty() == 1)
    {
      printf("\n\t Stack is underflow");
    }
   else
    {
      printf ("\n Stack elements: ");
```

```
for ( i = top ; i >= 0 ; i--)
  {
    printf ("%d", stack [i]);
  }
  }
}
```

* Refer entire program of stack implementation discussed in class.

PAGE NO.
DATE / /

* APPLICATIONS OF STACKS–

There are various applications of stack.

1) Well for ~~mness~~ of parenthesis or matching parenthesis in an expression
2) Infix to Postfix conversion.
3) Postfix evaluation
4) Recursion.

I] Well form-ness of Parenthesis:

- Stack can be used to check whether a set of paranthesis is well-formed or not.
- What is matching parenthesis ? or well-formness of paranthesis

Eg. '((( ))) ' ⟶ ✓ matching

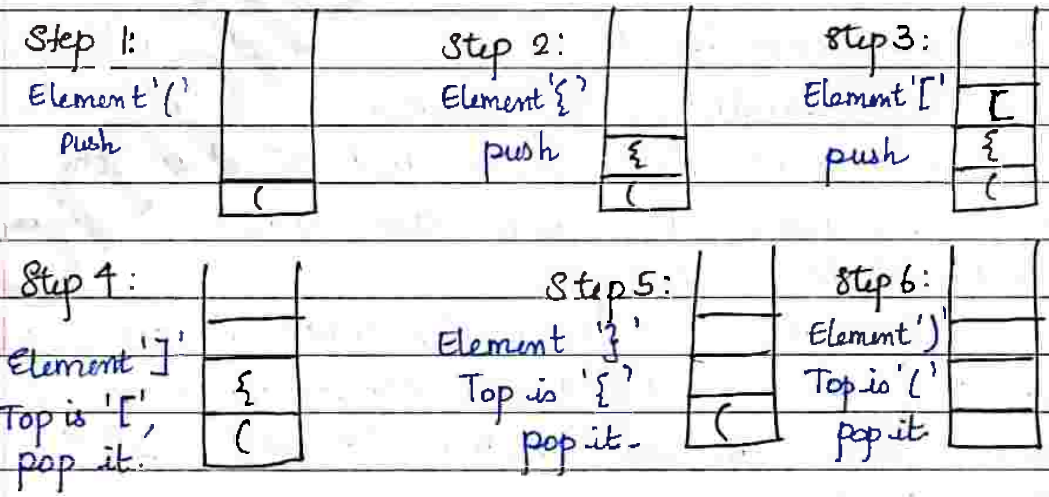
)(( )) ⟶ ✗ not matching

- For an expression to be well formed, a closing parenthesis symbol must match the last unmatched opening parenthesis symbol and all parenthesis symbols must be matched when the input is finished.
- Why it is needed?
  - Compiler needs it for loops, nested loops, conditional loops to denote block of code. If the pairs {, } do not match, it will lead to an error (compile time)..

Algorithm to check well-formness of an expression:-

1. Declare a stack.
2. Input Algebraic Expression from the user.
3. Traverse the Expression.
   For each character of input string,
4. Push the current character to stack if it is an opening paranthesis such as '(', '[' or '{'. (Ignore all other characters)
5. If character is closing paranthesis
   (a) check top of stack, if it is non-empty, and the current character matches the character on top of stack, remove the character from top of stack.
   (b) Else return unbalanced paranthesis.
6. After complete Traversal, if there is any remaining opening bracket left in the stack, then the paranthesis are not balanced.

Ex.① ({ [ ] })



11

Ex. ② ({])

Step 1 → Element '('
Push '(' into the stack

Step 2 →
Element '{'
Push onto the stack

Step 3 : →
Element ']'

∴ Top is not '['. Exit.

As the stack is not empty, the given string is not balanced.

* POLISH NOTATIONS:

Questions Asked in Exams.

1. Explain Infix, postfix and prefix expressions with an example.
2. Discuss the polish operations of arithmetic expression in application of stack

- Notation is a way of writing arithmetic expression.
- Polish Notations are used as a way of expressing arithmetic expressions, that avoids the use of brackets to define priorities for evaluation of operators.

- There are three ways to write any arithmetic expression. All these ways are different but equivalent notations ie. eventhough they are different, they do not change the essence or output of an expression.

- The polish notations are as follows:-
  1. Infix Notation
  2. Prefix (Polish) Notation
  3. Post-fix (Reverse-Polish) Notation.

Imp

- The way of using operators in the expression decides the name of notation.

I] Infix Notation:

- This is the basic and usual way of writing an expression.
- Here, the operators are placed in between the operands as per standards.

Eg: a + b

Here a, b are "operands" and '+' is an "operator".

- This is readable for humans, but sometimes it is not easy for computing devices, for which it has to be converted to other formats like prefix or postfix.

II) Prefix Notation (Polish Notation)

- This is another way of writing an arithmetic expression, in which operators are placed before operands.
  - Eg. + ab
    '+' → operator
    'a, b' → operands
- Prefix Notation is also called as "Polish Notation".

III) Postfix (Reverse-polish) Notation:-

- Here, it is a mathematical notation in which operators follow their operands, in contrast to polish Notations.
  - Eg. a b +
    '+' → operator
    'a', b'' → operands.

Postfix Notation is also known as "Reverse-Polish Notation".

Conversion of Infix to Postfix Expression:-

Imp

To convert Infix to Postfix expression, a stack would be used. In this conversion, the stack would be used as a mediator between source string (Infix) and target string (postfix). Stack will hold operators for some time.

* Algorithm:-

Step 1: The input string (Infix notation) is scanned from left to right.

Step 2: If the scanned character (ch) is a space or tab, it is skipped.

Step 3: If the scanned character (ch) is a digit, or alphabet, it is appended to postfix string.

Step 4: If the scanned character (ch) is opening parenthesis '(', it is pushed to stack.

Step 5: If the scanned character (ch) is an operator:
- All operators from the top of stack which has higher or same priority as ch are popped and appended to postfix string.
- When a less priority operator is found, then it will be pushed in the stack with ch.

Step 6: If the scanned character (ch) is a closing parenthesis ')', then all the operators above

the pending parenthesis are appended to postfix string.

Step 7 : After evaluation of all characters, the remaining operators from stack are appended to postfix string.

Example :- ((A+B) * (C-D)) / E

Character from Infix String	Stack	Postfix Expression.
(	(	
(	((	
A	((	A
+	((+	A
B	((+	A B
)	(	A B +
*	(*	A B +
(	(* (	A B +
C	(* (	A B + C
-	(* (-	A B + C
D	(* (-	A B + C D
)	(*	A B + C D -
)		A B + C D - *
/	/	A B + C D - *
E	/	A B + C D - * E
end of string		A B + C D - * E /

* EVALUATION OF A POSTFIX EXPRESSION:-

- The postfix notation is used to represent algebraic expressions.
- The expressions written in postfix form are evaluated faster as compared to infix notation as parenthesis are not required in postfix.
- As postfix expression can be evaluated as two operands and one operator at a time, it is easier for the compiler and the computer to handle this evaluation.

* Rules Regarding Evaluation of postfix expressions:-

Basic Rules:-

- Read the elements from left to right and push the element in the stack if it is an operand.
- If the element is found to be operator, then pop two elements from stack. Evaluate the expression which is formed by inserting the operator in operands.
- Push the results of the evaluation into the stack. Repeat it till the end of expression.
- Print the popped element from stack as a result.

* Algorithm to Evaluate Postfix Expression:-

Q. Write an algorithm to Evaluate a Postfix Expression with an example.

Q. How will you evaluate a Reverse-polish expression? Explain with an example.

ALGORITHM:-

Step 1: Read the postfix expression from Left to Right.

Step 2: If operand is encountered, push it in Stack.

Step 3: If operator is encountered, Pop two elements
* A → To Element
* B → Next Top Element
Evaluate B operator A.

Step 4: Push the result onto the Stack.

Step 5: Read next postfix element if not end of postfix string.

Step 6: Repeat from Step 2.

Step 7: Print the result popped from Stack.

* Example:-

1) Evaluate the foll. postfix expression and show stack after every step in tabular form.
Given A=5, B=6, C=2, D=12, E=4; ABC + * DE \-

⇒ Soln:-

Given Expression = ABC + * DE \-

This will become "5 6 2 + * 12 4 \ - " after the substitution.

5 6 2 + * 12 4 / -

Postfix string Elements	Stack	Evaluation.
5	5	
6	5 6	
2	5 6 2	
+	5 8	A=2 B=6 B+A 6+2=8
*	40	A=8 B=5 => B*A 5*8=40
12	40 12	
4	40 12 4	
/	40 3	A=4 B=12 12/4 =3
-	37.	A=3 B=40 40-3 =37

∴ Result = 37 //

2) " 6 2 3 + - 3 8 2 + + * 2 ^ 3 + "

⇒ Solutions

Postfix String Elements	Stack	Evaluate
6	6	
2	6 2	
3	6 2 3	
+	6 5	2+3=5
-	1	6-5=1
3	1 3	
8	1 3 8	

2	1 3 8 2	
+	1 3 10	8+2 =10
+	1 13	3+10 =13
*	13	1*13 =13
2	13 2	
^	169	13^2 = 169
3	169 3	
+	172.	169+3 =172

∴ Result = 172 //

3) " 4 5 6 * + "

Postfix String Elements	Stack	Evaluation
4	4	
5	4 5	
6	4 5 6	
*	4 30	5*6=30
+	34	4+30=34

∴ Result = 34 //

* RECURSION :-

Q. Explain the term recursion with an example.
Q. Explain recursion as an example of stack with an example.

⇒ Definition :-
Calling a function inside itself is called as recursion. Such a function is called as recursive function.

* How recursion works?

- Recursion is a technique of solving any problem by calling same function again and again until some breaking condition (base) condition where recursion stops and it starts calculating the solution from there on.
For eg. Calculating factorial of a given number.

- Thus, in recursion last function called needs to be completed first.
- Now, stack is a LIFO data structure ie. (Last in First Out) and hence it is used to implement recursion.
- The High Level Programming languages such as Pascal, C etc. that provides support for recursion use stack for book keeping.
- In each recursive call, there is need to save the

- 1) current values of parameters
- 2) local variables and
- 3) the return address (the address where the control has to return from the call)

– Also as a function calls to another function, first its arguments, then the return address and finally space for local variables is pushed onto the stack.

– Recursion is extremely useful and extensively used because many problems are elegantly specified or solved in a recursive way.

* Advantages of Recursion:

1. It helps to reduce the size of the code by minimizing the code.
2. It makes easy to maintain function calling related information.
3. Evaluation of stack can be implemented through recursion.
4. Also infix, prefix, postfix notation can be evaluated with the help of recursion.

Recursion:

* **Direct Recursion →** Main fn calling a function a() and fn. a() calling itself again.

```
main()
{
  a()
}

a()
{
  a()
}
```

ie. main()
↓
a()
↺

* **Indirect Recursion :-** Main fn calling function a, fn a calling fun. b and fn b again calling fn. a.

```
main()
{
  a()
}

a()
{
  b()
}

b()
{
  a()
}
```

ie. main()
↓
a() ←
↓
b();
↺

(Ex) $3! = 3 * 2!$

$\downarrow$

$2 * 1!$

$\downarrow$

$1 \times 0!$

$\downarrow$

1

Here, we observe two things

$n! = n * (n-1)!$ → Recursive condn.

$0! = 1$ → Base condn. or Stopping condn.

→ So, inorder to implement Recursion, we require two things —

1) a recursive condn
2) a base or stopping condn.

(Ex) Suppose we want to find out a^b.

$3^2 = 3 \times 3^1$

$\downarrow$

3×3^0

$\downarrow$

1

Here, we can write it as

$a^b = a * a^{b-1}$ → Recursive condn

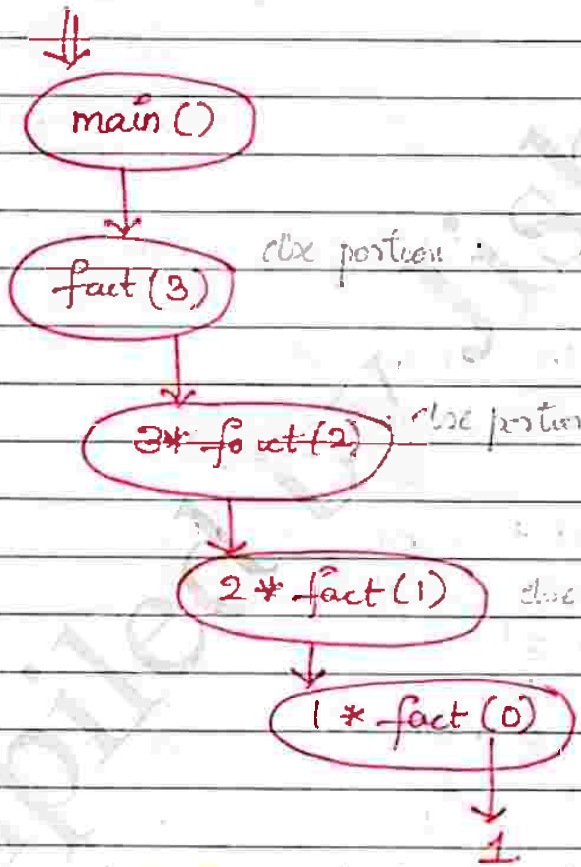
$a^0 = 1$ → Base condn.

Program (factorial):-

```
int fact (int n)
{
    if (n==0)                          ] → Base condn / stopping condn
        return 1;
    else
        return (n * fact (n-1));       ] → Recursive condn.
}
```

```
void main()
{
  int x, ans;
  scanf("%d", &x);
  ans = fact(x);
  printf("%d", ans);
}
```

→ function call



If you observe the last called function is executed first and this property is called LIFO and this is implemented by data structure called "STACK". Hence, we use stack to implement Recursion.

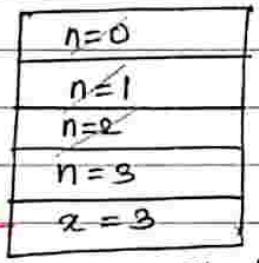
⟹ Now, lets understand how stack is used to implement Recursion.

Our main mly has four parts :

Heap	→ Dynamic mly allocation
Stack	→ variables or data of our progm
Static variables	→ Global variables are held here
Code section	→ Instructions / code part

How recursion uses stack:

First main function is called (lets say we give x=3)
value of x=3 is stored in stack, called "activation record of main-fn".
↳ value of variables in that fn at that point of time
↳ & when the function terminates, its activation record is deleted.



Then fact of 3 is called. Then n=3, so in stack n=3 is stored.
Then fact of 2 is called, Then n=2, so activation record of fact of 2 ⇒ n=2 is stored in stack.
Then fact of 1 is called, so activation record of fact of 1 is stored in stack.
Then fact of 0 is called, n=0 is stored in stack

Now, if n==0, then return 1 → once this stmt is executed control goes to main
so, fact of 0 has completed its execution.
& fact(0) = 1 and activation record of fact(0) can nbe removed from stack.
Then fact(1) = 1 × 1 = 1 → so activation record of fact(1) will be deleted
fact(2) = 2 * 1 = 2 → delete activation record of fact(2) will be deleted.
fact(3) = 3 * 2 = 6 → fact of (3) activation function will be deleted.
& value of fact(3) will be transferred and main will print 3, so activation fn of main will be deleted.
& execution completes.

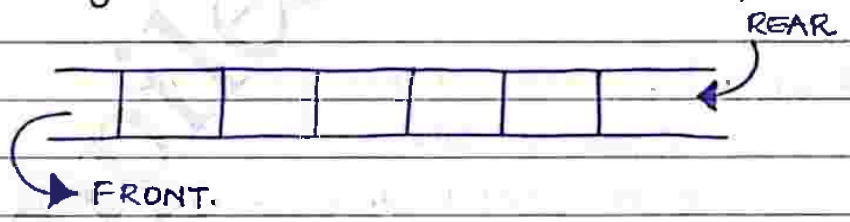
UNIT - II (CONTINUED)

INTRODUCTION : ADT OF QUEUE

* **Definition :-**

Queue is a data structure in which addition of an element is allowed at one end (called as rear) while removal of an element is allowed at another end (called as front).

- Queue is an Abstract Data Type.
- Queue is also known as FIFO (First In First Out).
- It indicates that the first inserted element will be removed first and the last inserted element will be removed last.
- Real life examples of queue : People waiting in a queue to get a train / movie ticket.
- The first person in the queue will be the first one to get a ticket and leave the queue.



- Some other Real Life Examples of Queue are :
  - OS processes
  - Queue of packets in data communication.

* OPERATIONS ON QUEUE :-

- The queue is an abstract data type which is defined by the following structure and operations :
- Queue is structured as an ordered collection of items which are added at one end, called the "rear" and from the other end, called the "front".
- The queue operations are as follows :-

1. Create
2. Enqueue
3. Dequeue
4. isEmpty
5. isFull
6. Size

(1) Create () : Creates and initializes new queue that is empty. It does not require any parameter an returns an empty queue.

(2) Enqueue (item) : Adds a new element to the rear of the queue. It requires the element to be added and returns nothing.

(3) Dequeue () : Removes the element from the front of the queue. It does not require any parameter and returns the deleted item.

(4) isEmpty () : Checks whether the queue is empty or not. It does not require any parameter and returns a Boolean value.

(5) Size() : Returns the total number of elements present in the queue. It does not require any parameter and returns an Integer.

* Example :-

Consider q, is a queue that has been created and starts out empty, then in the following table we can observe the results of a sequence of queue operations

Queue operation	Queue Content	Return Value.
q.isEmpty()	[]	True
q.enqueue("A")	[A]	
q.enqueue(5)	[A, 5]	
q.dequeue()	[5]	A
q.enqueue("B")	[5 B]	
q.size()	[5 B]	2

* Differentiate between Stack and Queue.

→ Parameter	stack	Queue
1. Operation end :	Elements are added and deleted from the same end.	Elements are added and deleted from different ends
2. Pointer :	Single pointer is used to point to top of the element.	Two pointers are used to point to front and rear ends.

3.	Order :	LIFO	FIFO
4.	Operation : Names	Insertion - PUSH Deletion - POP	Insertion - ENQUEUE Deletion - DEQUEUE
5.	Examples:	Books arranged in shelf, plates arranged one above another.	Queues at Booking counter.

* ARRAY IMPLEMENTATION OF QUEUE:-

Q. Explain Queue Full and Queue Empty condition with suitable example.

→ Queue is considered as a linear data structure, hence it can be represented using array.
- It is also known as static implementation of queue. It can also be implemented using Linked List.
- Here two pointers will work : front and rear.

- Front will point to the first element, while rear will point to the last element.
- Consider an int arr[5].
- Initially the queue is empty. Therefore, front & rear will both point to -1.
- Hence, this condition can be used to check the emptiness of queue.

i.e. If Front = Rear = -1, Then Queue is Empty.

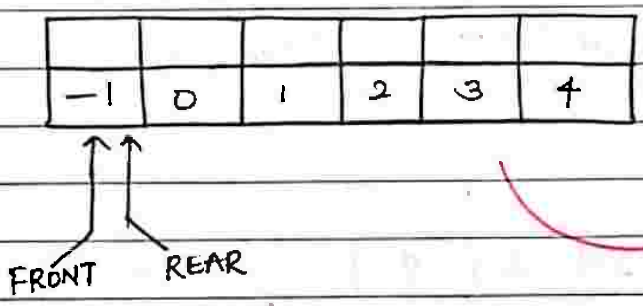


Fig: Represents a condition when the queue is empty.

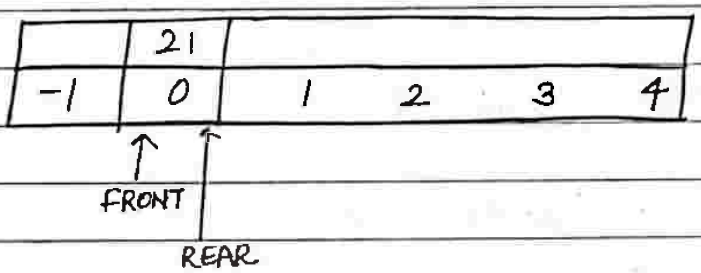
* ENQUEUE : INSERTING AN ELEMENT IN QUEUE.

Q. Describe the Enqueue operation in queue with the help of an algorithm.

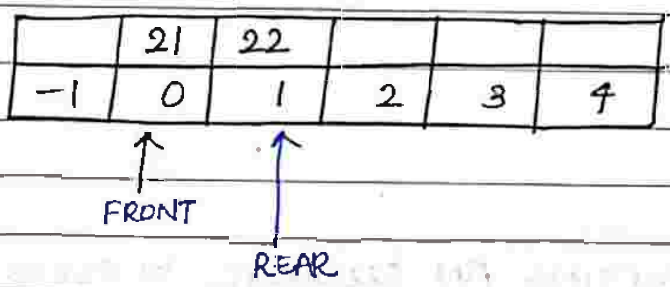
- Like in a normal queue, a new person entering the queue will stand in the queue at the back or enters the queue from back. Similarly, in Queue ADT also insertion happens at the rear end.

- While adding the first element, both FRONT and REAR are incremented by one and at rear position new element would be added.

Eg. Suppose, 21 is to be inserted (enqueued) into an empty queue.

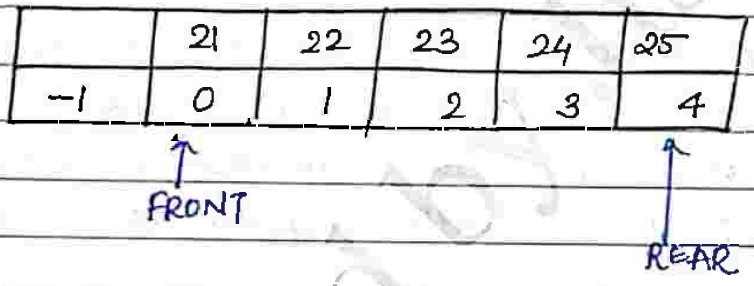


- From here after every insertion, the REAR will be incremented and at REAR position new element will be added.



Front & Rear pointers after inserting "22".

- When REAR reaches SIZE -1 position, the queue is considered as full.



REAR is at pos. MAX-1 or SIZE-1 ⇒ includes queue is Full.

* ALGORITHM TO INSERT AN ELEMENT IN QUEUE:

Step 1 : If REAR = SIZE-1, then print "Queue is Overflow"

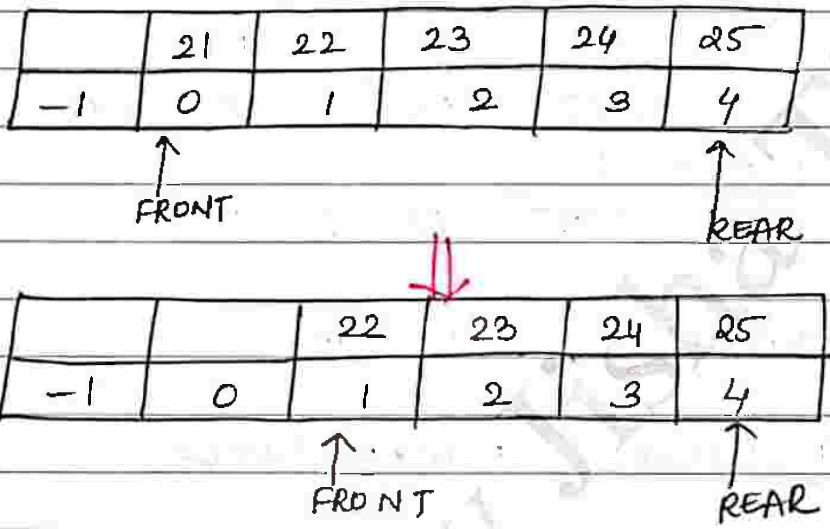
Step 2 : REAR = REAR + 1

Step 3 : QUEUE [REAR] = X

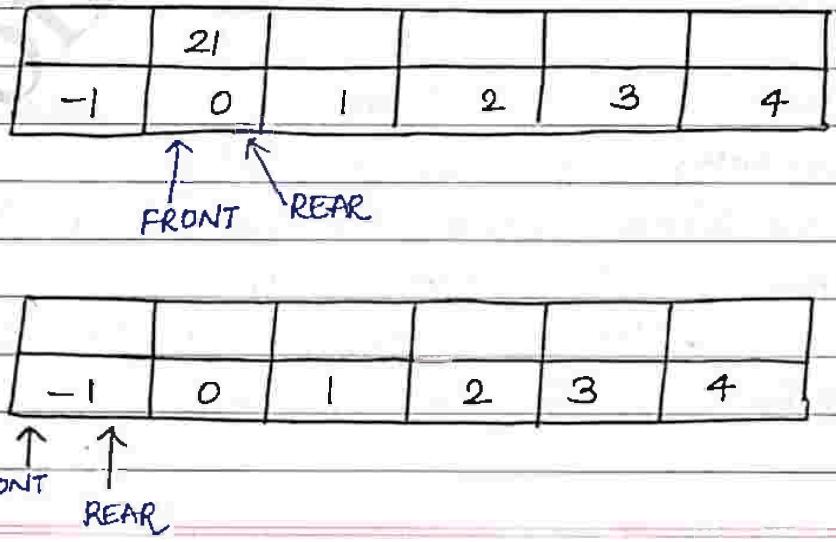
Step 4 : If FRONT = -1; then FRONT = 0.

* DEQUEUE : Deleting an element from Queue.

- Whenever a deleted occurs, element at front position is deleted an front is incremented by 1.



- If a single element is present in the queue at the time of deletion (FRONT and REAR both are at same position), then after deletion both FRONT and REAR are set at position -1, which now indicates that now queue is empty.



* ALGORITHM TO DELETE AN ELEMENT FROM QUEUE:-

Step 1: If FRONT = -1 then write "Queue is underflow"

Step 2: Return QUEUE[FRONT]

Step 3: If FRONT = REAR ; then FRONT = -1,
REAR = -1
Else FRONT = FRONT + 1.

* Implementation: Refer Classroom.

* TYPES OF QUEUE : CIRCULAR QUEUE.

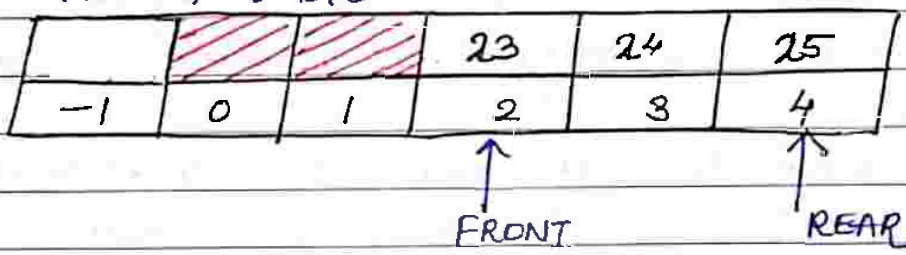
Q. What is a circular Queue ? Explain with an example.

- There are different types of Queue, like Linear Queue, Circular Queue, Priority Queue and Double Ended Queue.
- We have already seen what is a Linear Queue. (Check the implementation of Queue ADT using Array)

- To understand Circular Queue, let us understand the disadvantages of linear queue:
- In simple queue, when elements are deleted, the pointer FRONT is incremented.
- The spaces which get free after deletion cannot be reutilized in this type of queue.

SIZE = 5

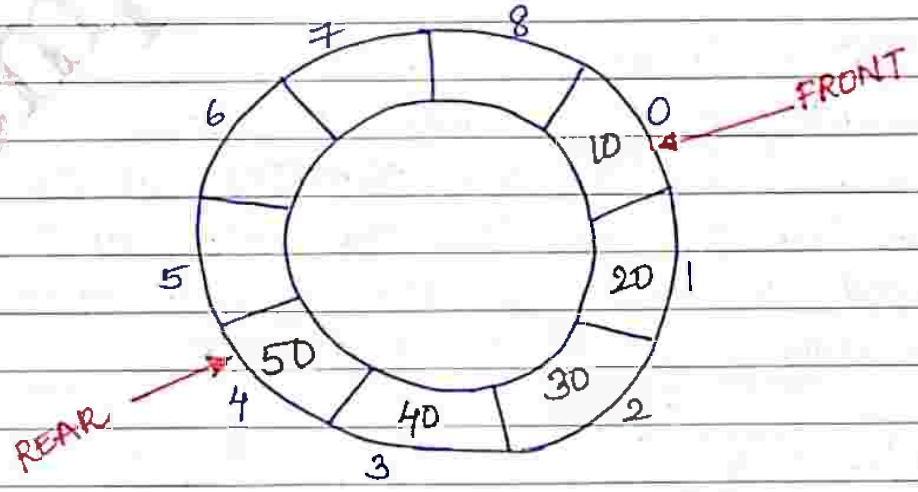
MEMORY WASTE



- In the above fig. the first two locations are free because of the deletion of first two elements, but as rear is at SIZE-1 position, the queue is considered as full and we cannot add new elements.
- It leads to the wastage of memory.
- So, to solve this problem, DS provides the concept of circular queue.

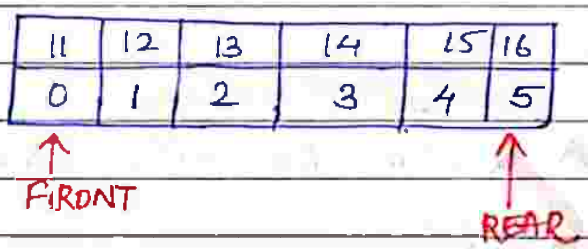
Definition :-

Circular Queue is a linear data structure in which the operations are carried out on the basis of FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.

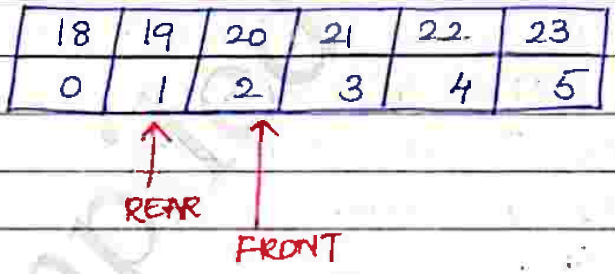


- Let us see the array implementation of circular queue.
- Here when the REAR reaches at SIZE-1 position, it is not considered that queue is full.
- Only in two situations circular queue is considered as full:

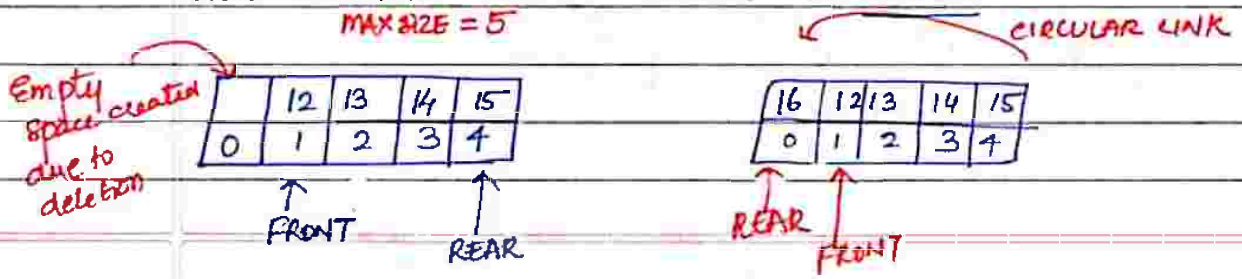
1. FRONT is at position 0 and REAR is at position SIZE-1.



2. FRONT is one place next to REAR.



- This is because in the process of adding elements, if REAR reaches to last position and if there are vacant positions at the beginning of array, then the REAR can be shifted to the first position of array and new element can be added at REAR.



ALGORITHM TO INSERT AN ELEMENT INTO CIRCULAR QUEUES

The insert operation for array implementation of circular queues involves the following tasks:

1. Checking whether the queue is already full.
2. Updating the rear pointer
   (a) If the queue is empty, set FRONT and REAR to point to the first location in the queue.
   (b) If the REAR is pointing at the last location of the queue, set rear to point to the first location in the queue.
   (c) If none of the above situations exist, simply increment the REAR pointer by 1.
3. Insert the new (I/P) element at REAR location.

Q. Write an algorithm to realize the insert operation for array implemented circular queues.

insert (queue [MAX], front, rear, element)

Step 1 : Start

Step 2 : if (FRONT = 0 and REAR = MAX-1) or FRONT = REAR + 1 goto Step 3 else goto Step 4.

Step 3 : Display message, "Queue is Full" and goto Step 10.

Step 4 : if FRONT = NULL go to Step 5 else goto Step 6

FRONT = (FRONT+1) mod n

REAR = (REAR+1) mod n

Step 5 : Set FRONT = REAR = 0

Step 6 : If REAR = MAX-1 goto Step 7 else goto Step 8.

Step 7 : Set REAR = 0

Step 8 : Set REAR = REAR + 1

Step 9 : Set queue [REAR] = element

Step 10 : Stop

* Applications of Circular Queue :-

1. Memory Management
2. CPU Scheduling (OS uses circular queue to insert the processes and then execute them)
3. Traffic Sim (Each light get ON one by one after every interval of time)

* ENQUEUE operation :-

- First we will check whether the Queue is full or not.
- Initially, the front and rear are set to -1. When we insert the first element in a Queue, front and rear are both set to 0.
- When we insert a new element, the rear gets incremented. i.e. rear = rear + 1.

Scenarios for inserting an element

→ There are two scenarios in which queue is not full:

1] If rear != max-1, then rear will be incremented to mod (max size) and the new value will be inserted at the rear end of the queue.

2] If front != 0 and rear = max-1, it means that queue is not full, then set the value of rear to 0 and insert new element there.

→ There are two cases in which element cannot be inserted:

1] When front == 0 && rear = max-1, which means that front is at the first position of the Queue and rear is at the last position of the Queue.

2] front == rear + 1.

PAGE No.
DATE / /

* Algorithm to Insert an element in a circular queue.

Step 1 : If (REAR+1) % MAX = FRONT
write "OVERFLOW"
Go to step 4.
[End 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) % MAX
[End If]

Step 3 : Set QUEUE [REAR] = VAL

Step 4 : Exit

* DEQUEUE OPERATION:-

The steps of dequeue operation are given below:

- First, we check if the Queue is empty or not. If the Queue is empty, we cannot perform the dequeue operation.
- When the element is deleted, the value of front gets decremented by -1.
- If there is only one element left which is to be

deleted, then the front and rear are reset to -1.

* Algorithm to delete an element from the circular queue.

Step 1 : If FRONT = -1
write "UNDERFLOW"
Goto Step 4
[End of if]

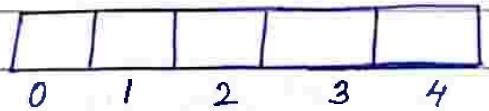
Step 2 : Set VAL = QUEUE [FRONT]

Step 3 : If FRONT = REAR // If there's only one elmt in the Queue.
Set FRONT = REAR = -1
Else
If FRONT = MAX-1 // Last location is front then
Set FRONT = 0
Else
Set FRONT = FRONT + 1 // normal deletion ; increment FRONT
[End of if]
[End of if]

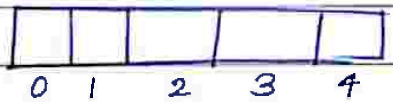
Step 4 : Exit.

* Example (Insertion & Deletion)

MAXSIZE =5

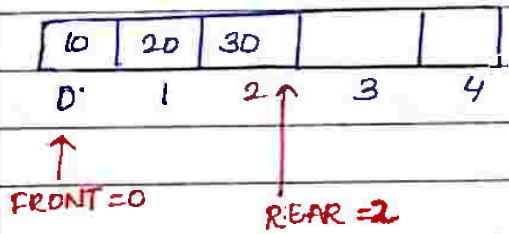


Front = -1
Rear = -1

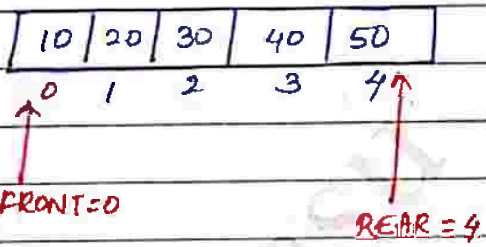


Front = 0
Rear = 0

Insert 20, 30

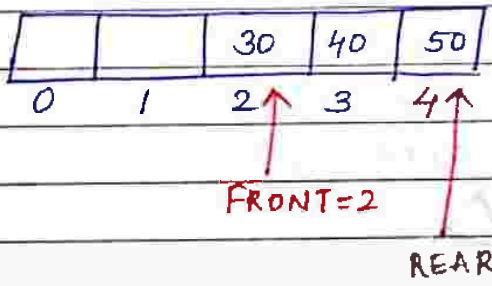


Insert 40, Insert 50



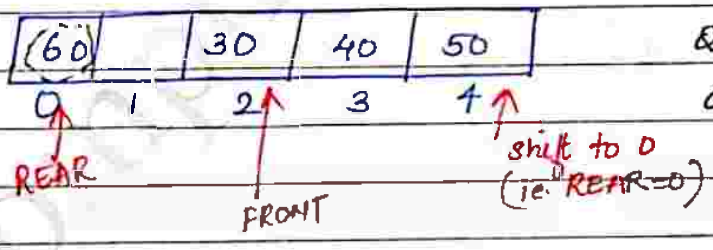
Queue Full
FRONT = 0 &/9 REAR = MAX-1

After 2 dequeue operations:



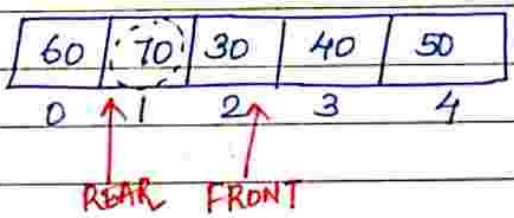
⟹ Queue not full as deletion has made the starting positions empty.

Insert 60



⟹ Insertion will be done at the front of the Queue where space is available. (∵ REAR=MAX-1 &/FRONT! =0)
REAR = 0.
QUEUE[REAR] = value

Insert 70



⇒ Here, FRONT ! =0
REAR = (REAR+1) % M
= (0+1) % 5
= 1 % 5 = 1
So, insert at QUEUE[1]
↑ REAR

* PRIORITY QUEUE :-

- The priority queue is considered as an extension of queue with the following given properties:-

1. Every element in the queue has a priority associated with it.
2. An element with high priority is dequed before an element which has low priority.
3. If two elements have the same priority, they are served according to their order in the queue.

→ ADVANTAGES:-

a) Simple priority
b) Reasonable support for priority
c) Most important tasks having higher priority are performed first.
d) Important tasks do not have to wait for completion of less priority tasks.
e) Suitable for applications with varying time & resource requirements.

→ TYPES OF PRIORITY QUEUE:-

A priority queue is of two types:

a) Ascending order Priority Queue
b) Descending Order Priority Queue

PAGE NO.
DATE / /

I] Ascending order Priority Queue :-

- An ascending order priority queue gives the highest priority to the lowest number in that queue.
- For example; you have six numbers in the priority queue that are 4, 8, 12, 45, 35, 20.
- Firstly, you will arrange these nos. in ascending order. The new list is as follows:

  4, 8, 12, 20, 35, 45.
- In this list 4 is the smallest number. Hence, the ascending order priority queue treats number 4 has highest priority.

4	8	12	20	35	45

4 - Highest
45 - Lowest priority.

II] Descending order Priority Queue :-

- A descending order priority queue gives the highest priority to the highest number in that queue.
- For example, you have six nos in the priority queue that are 4, 08, 12, 45, 35, 20.
- Firstly you will arrange them in descending order.

  45, 35, 20, 12, 08, 4.
- In this list 45 is the highest number. Hence, the descending order priority queue treats number 45 as highest priority.

45	35	20	12	8	4

45 - Highest
4 - Lowest.

Elements of Priority Queue:

- As an element in priority queue, there may be numbers, characters or various types of complex structures which can be searched on specific fields
- Example: Records of Telephone Directory
- The priority values need not be a part of the actual queue elements.

Implementation of Priority Queue:

- A priority queue can be implemented using data structures like arrays, linked lists, or heaps.
- We study the implementation of priority queue using an array.

```
struct data
{   int element;
    int priority;
}
```

- insert() operation → can be carried out by adding element at the location depending upon priority.
- delete() operation → can be carried out by deleting element from the front position.

⟹ Array Implementation of Priority Queue.

Rules :-

1) The lower priority number indicates the higher priority.

2) Add jobs in the queue and arrange them priority wise.

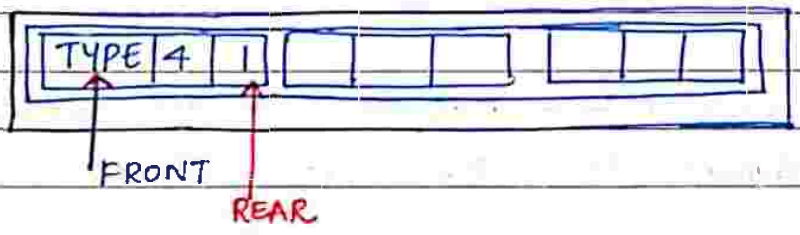
3) When two jobs have same priority, they will be added order-wise.

→ **Example :-**

Following jobs have to be added :

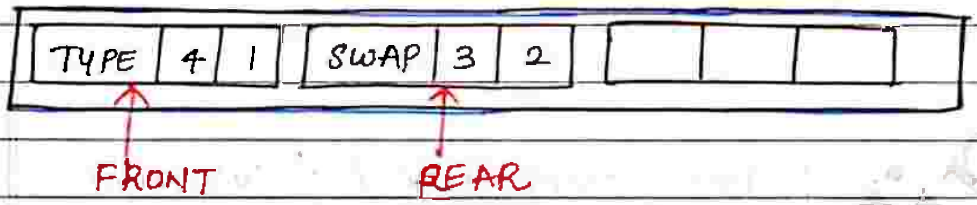
Name of job.				
	TYPE	4	1	← priority
	SWAP	3	2	
	COPY	5	3	← order

1] First job when added into an empty queue

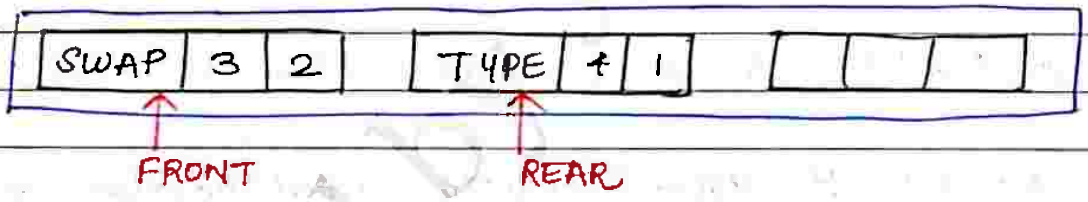


FRONT & REAR both will point to the job TYPE with priority 4.

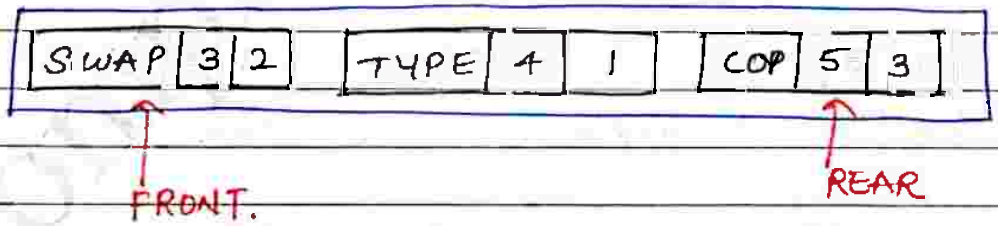
2] Next | SWAP | 3 | 2 | is to be added into the queue.



- Now, REAR would point to SWAP.
- Now, again the priority of SWAP is more as compared to TYPE. Therefore, SWAP should be placed before TYPE.



3] Next | COPY | 5 | 3 | is to be added into the queue.



Algorithm to implement Priority Queue:-

Step 1 : Start

Step 2 : Show options 1. Insert 2. Delete
3. Display 4. Exit

Step 3 : Accept choice

Step 4 : As per user's choice, call a function out of InsertPQ(), DeletePQ(), display(), exit().
Step 5 : Exit.

Algorithm to insert element in Priority Queue:

Step 1 : If queue is full, print the message of exit.
Step 2 : Accept element
Step 3 : Search position for element as per priority by moving existing elements.
Step 4 : Set new element at proper position.

Algorithm to delete element from priority queue.

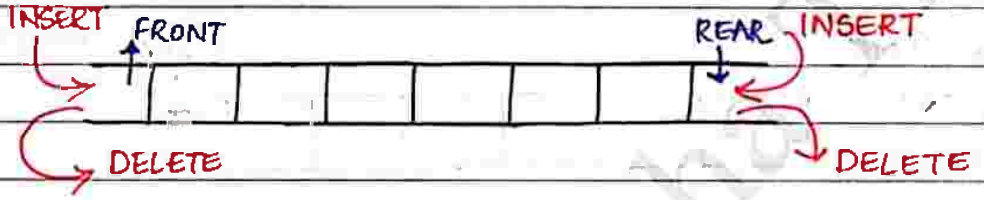
Step 1 : If queue is empty, print message and exit.
Step 2 : Print front element as deleted.
Step 3 : P = PQ [front]
Step 4 : Increment front by one
Step 5 : Return P.

Algorithm to display elements from Priority Queue:

Step 1 : If queue is empty, print the message and exit.
Step 2 : Print elements from Front to rear.

* INTRODUCTION OF DOUBLE ENDED QUEUE:-

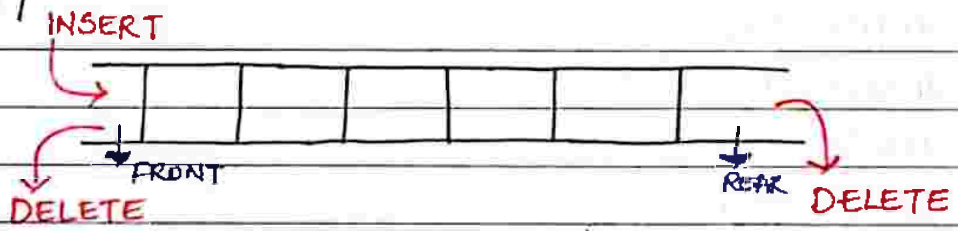
- Double Ended Queue is a data structure in which the insertion and deletion operations are allowed at both the ends (front & rear).
- That means, it has an unrestrictive nature of adding & deleting elements



- There are two ways to represent Double Ended Queue:
  1. Input Restricted double ended queue.
  2. Output Restricted double ended queue.

I] Input Restricted Double Ended Queue:

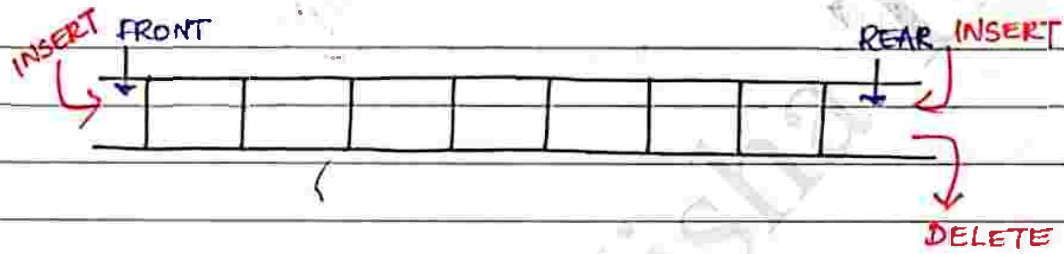
- The input restricted queue means some restriction is applied to the insertion.
- In this type of queue, the insertion operation is allowed at only one end, while deletion operation is allowed at both the ends.



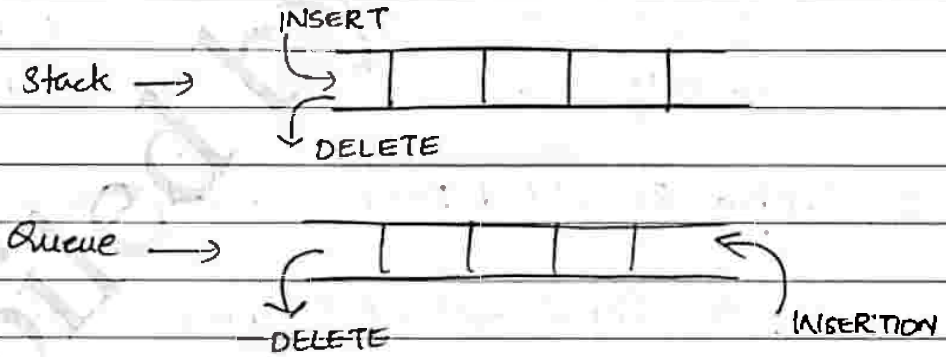
DEQUE

II] Output Restricted Double Ended Queue :- (Dqueue) (Deque)

- The output restricted Queue means some restrictions are applied to the deletion operation.
- In this type of queue, the deletion operation is allowed at only end while insertion operation is allowed at both the ends.



→ It has the properties of both Stack & Queue, so it can be used as both Stack and Queue.



* Operations on Deque :

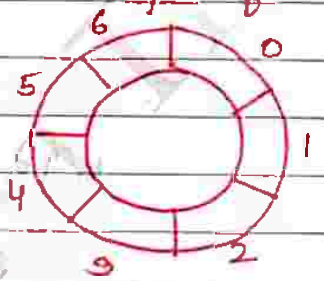
1) insert at front
2) delete from front
3) insert at rear
4) delete from rear

- ⊙ getfront ()
- ⊙ getrear ()
- ⊙ isFull ()
- ⊙ isEmpty ()

→ We can implement the queue using a Circular Array or a Doubly Linked List.

→ We will be checking the implementation using Circular Array.

→ Similar to the concept of Circular Queue



* Applns:

1) Redo and Undo operations.
2) Palindrome checker. Eg. RADAR
3) Multi-processor scheduling → (stores ready to execute processes)

Algorithm to Implement Double-Ended Queue:

Step 1 : Start

Step 2 : Show options 1. Insert at Front 2. Insert at rear 3. Delete from Front 4. Delete from Rear 5. Display

Step 3 : Accept choice

Step 4 : As per users choice call a fn out of enqueueFront(), enqueueRear(), dequeFront() dequeRear(), display()

Step 5 : Exit

Algorithm of EnqueueFront:- (ie. Insertion at the Front End of Deque)

Step 1 : If queue is full, print message and Exit.

Step 2 : Accept element

Step 3 : If queue is empty, set front = rear = 0
deque_arr [front] = element. // 1st element

else if (front == 0) // deque cannot be full until all places are filled (circular queue concept)
set front = MAX-1
deque_arr [front] = element

else
front = front - 1 // Insertion at front, always decrement front
deque_arr [front] = element

end if

Algorithm of EnqueueRear :- (ie. Insertion at the RearEnd of Deque)

Step 1 : If the queue is Full, print the message and Exit.

Step 2 : Accept the element

Step 3 : If the queue is empty, set front = rear = 0
deque_arr[0] = element

```
else if (rear = MAX-1)          //if rear is at the last position set it to point to the first positn (circular array concept)
    set rear = 0

else
    rear = rear + 1
                    // normal insertion
deque_arr [rear] = element
```

Algorithm for DequeueFront ie. Deletion of an element from the front in Deque.

Step 1 : If the queue is empty, print message and exit.

Step 2 : x = deque_arr [front]

```
if (front == rear)              // only one element
    set front = -1 and rear = -1
else if                         // if front points to last location, then set front to 0 (circular array concept)
    front = MAX-1
        set front = 0
else
    front = front + 1
                    // normal deletion
```

Step 3 : return (x)

Algorithm for DequeueRear ie. Deleting an element from RearEnd of Deque.

Step 1 : If the queue is empty, print message and Exit.

Step 2 : x = deque_arr[rear]

if (front = rear) // only one element
set front = rear = -1

else if
rear = 0, then set rear = MAX-1 // rear is at first position

else
rear = rear - 1 // normally when we delete from rear, we have to decrement rear.

Step 3 : return(x).

Algorithm for Display.

Step 1 : If queue is empty, print the message and exit.

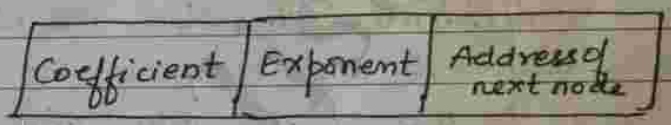
Step 2 : Print elements from front to rear.

* Applications of Linked List :-
* Polynomial Representation and Addition using Linked List:

A polynomial $p(x)$ is the expression in variable x, which is in the form $ax^n + bx^{n-1} + \dots + jx + k$ where a, b, c fall in the category of real nos and n is a non-negative integer which is called the degree of the polynomial.

Polynomial Expression → $4x^3 + 6x^2 + 10x + 6$

Node Structure



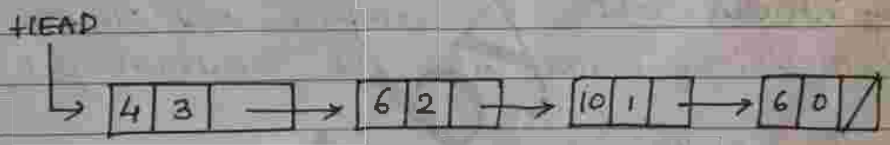


Fig: Linked List Representation of Polynomial Expression.

⇒ Polynomial Addition:

↳ Addition of two polynomials involve combining like terms present in the two polynomials.
↳ means adding terms having same variables of same exponents.

$3x^2 + 2x + 1$ —— (1)
$5x^2 - x + 2$ —— (2)

$3x^2$ & $5x^2$ are like terms
$2x$ and x are like terms.

$$\begin{array}{r} 3x^2 + 2x + 1 \\ 5x^2 - x + 2 \\ \hline 8x^2 + x + 3 \end{array}$$

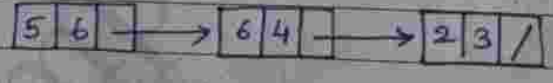
↳ Addition of two polynomials becomes easier if the terms are arranged in descending order of exponents.

↳ If exponents & variables are same, add their coefficients, otherwise the highest exponent to the resultant and compare the next term.

↳ If any of the polynomial terms are finished in between while we compare, add all the remaining terms into the resultant polynomial.

$$\begin{array}{rl} 5x^6 + 6x^4 + 2x^3 & \text{— (1)} \\ \underline{8x^6 + 3x^2 + 4x + 5} & \text{— (2)} \\ 13x^6 + 6x^4 + 2x^3 + 3x^2 + 4x + 5. & \end{array}$$

Linked List Representation



.... (Poly 1)

... (Poly 2)

* Algorithm:

1. Repeat the following until PTR1 or PTR 2 becomes NULL.

if (PTR1 → expo == PTR2 → expo)

Add the coeffs and insert the newly created node in the resultant linked list and make PTR1 and PTR2 point to the next nodes.

if (PTR1 → expo > PTR2 → expo)

Insert the node pointed by PTR1 in the resultant linked list and make PTR1 point to the next node.

if (PTR1 → expo < PTR2 → expo)

Insert the node pointed by PTR2 in the resultant linked and make PTR2 point to the next node.

repeat until PTR2 != NULL

Insert the remaining nodes.

repeat until PTR1 != NULL

Insert the remaining nodes.

UNIT - III

LINKED LIST

* Introduction :-

- In C we have studied the concept of arrays, which are a group of elements with same datatype.
- Array is considered as an example of static memory allocation i.e. the size of the array is fixed and while declaring the array, we have to compulsory mention the size.
- Thus, if there is a need at run time to store more data in the array than its actual capacity, then there is no way of doing this. This is where a linked list becomes more useful.
- It allows dynamic memory allocation of memory space at run time. Thus, there is no need to block memory space at compile time.

⇒ Importance of dynamic memory allocation :-

- No need to initially occupy large memory.
- Memory can be allocated and de-allocated as per need.
- It avoids memory shortage as well as memory wastage.

LINKED LIST:-

Q. Explain Linked list as an ADT.

Defn:-

Linked list is a linear data structure which consist's of a group of elements called as nodes. Every node contains two parts : data and next. Data contains the actual element while next contains address of the next node.

TERMINOLOGIES:-

1. Node :-
   - Every element in a linked list is called as a Node. A node consists of two parts, Data and Next.
   - Data contains actual element, while next contains the address of the next node.

2. Data (Information) :-
   - It is the actual element (record) present in the data part of the node.
   E.g. Roll No. of a student.

3. Next :-
   - It contains the address of the next node. It helps to go from one node to another node.

4) Address :-

- Next part of every node contains the memory address of next node. This is very useful for traversing purpose.
- The last node always contains NULL value in the next part as there is no next node.
- When there is a single node, then it contains NULL value in the next part.

5) Pointer :-

- A pointer always points to the next member of the list.
- The first node to which the pointer points is called as head or header node.
- Pointer is considered as a reference to the linked list.

6) NULL Pointer :-

- When the linked list is empty, ie. there is no need in it then NULL value is set to the pointer.

E.g. P = NULL.

7) Empty List :-

- When there is no node in the linked list, then it is called an empty linked list.

Advantages of Linked List:

1. Linked list allows dynamic memory allocation by allowing elements to be added or deleted at any time during program execution.
2. The use of linked list ensures efficient utilization of mly space as only that much amount of memory space is reserved as is required for storing the list elements.
3. It is easy to insert or delete elements in a linked list, unlike arrays, which requires shuffling of other elements with each insert and delete operations.

Disadvantages of Linked List:-

1. A linked list element requires more memory space in comparison to an array element because it has to also stores the address of the next element in the list.
2. Accessing an element is a little more difficult in linked lists than arrays because unlike arrays there is no index identifier associated with each list element.

Thus, to access a linked list element, it is mandatory to traverse all the preceeding elements.

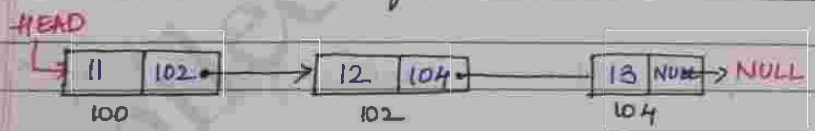
Q: Diff. blw Array & Linked list.

* TYPES OF LINKED LISTS:-

1) Singly Linked List
2) Doubly Linked List
3) Circular Linked List.

I) Singly Linked List:

- It is the basic form of linked list.
- In this linked list, every node is made up of two parts, data and next. Data part contains the actual element while next part contains address of next node.
- The next part of last node always contains null value.
- It is a one way traversing list, which means we can traverse in only the forward direction. We cannot traverse in the backward direction.
- The first node is called the head node, which is considered as reference to the linked list.



* Operations on Singly Linked List:-

1) Traversing
2) Searching
3) Inserting at end
4) Inserting at beginning
5) Inserting at middle.
6) Deleting First
7) Deleting Last
8) Deleting Middle

I] Traversing a Singly Linked List.

- Traversing means the process of visiting each node at least once.

⇒ Algorithm to visit a singly linked list :-

1. If head = NULL
   print ("List is empty")
   else
2. q = head
3. print (q → data)
4. Go to next node.
5. if q != NULL
   Repeat from step 3.
6. End.

II Counting the number of Nodes in a Singly Linked List.

Step 1 : If head is NULL then
print ('List is empty')
else

Step 2 : q = head and counter = 0

Step 3 : Counter = counter + 1

Step 4 : goto next node

Step 5 : if q != NULL
Repeat from step 3

Step 6 : return counter

II] SEARCHING A LINKED LIST:

- we can search for a node in the linked list. The data to be searched can be accepted from the user.
- This data part is compared with data parts of all the nodes.

Algorithm to search node in a singly linked list.

Step 1 : Read the data element to be searched.

Step 2 : If head is Null then,
print ('List is empty')
else

Step 3 : q = head

Step 4 : if data = q -> data
print (q -> data)
return

Step 5 : goto next node

Step 6 : if (q != NULL)
Repeat from step 4

{

III] INSERTING A NODE IN A SINGLY LINKED LIST:-

A node can be inserted at any place in the linked list:

i) At the end
ii) At beginning
iii) In the middle of linked list.

Before inserting a node into a linked list, in all the three cases, we check whether memory is available for new node. If the free memory has exhausted, then an OVERFLOW message is printed. Otherwise if the free memory is available, we allocate space for newNode.

Algorithm for adding a node at the end of a singly linked list:

Step 1 : q = head

Step 2 : Read data

Step 3 : alloc (newNode) . ie. create a new node
newNode → data = data
newNode → next = NULL

Step 4 : if q is NULL then
q = newNode and return

// If the head node is NULL ie. Empty Linked List make the new node as Head

Step 5 : temp = q

Step 6 : goto next node

temp = temp→next

Step 7 : if temp → next is not NULL then
Repeat Step 6.

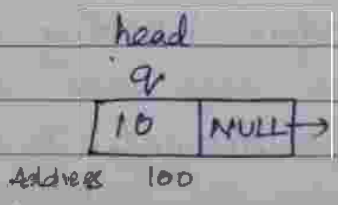
else (it means LL already has some element) find the last node

Step 8 : Set the address of new node in next of temp.
ie. temp→next = newNode

make the last node's next part pt to new node

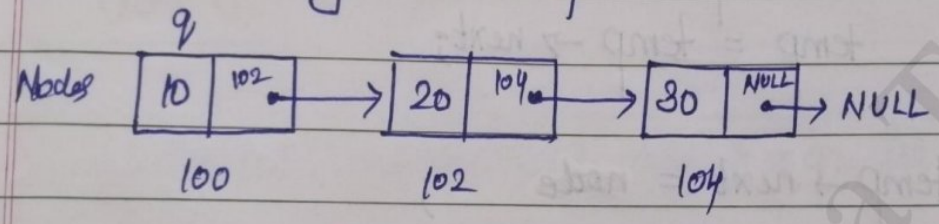
* Examples-

→ when first node is added.

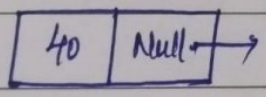


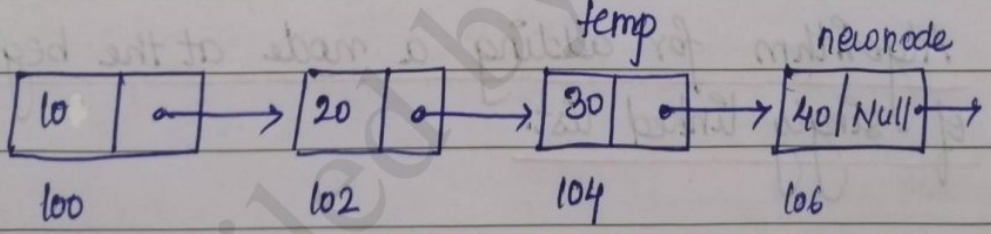
PAGE No.
DATE / /

→ When already nodes are present.



New Node:





Here, temp→next = newNode

}

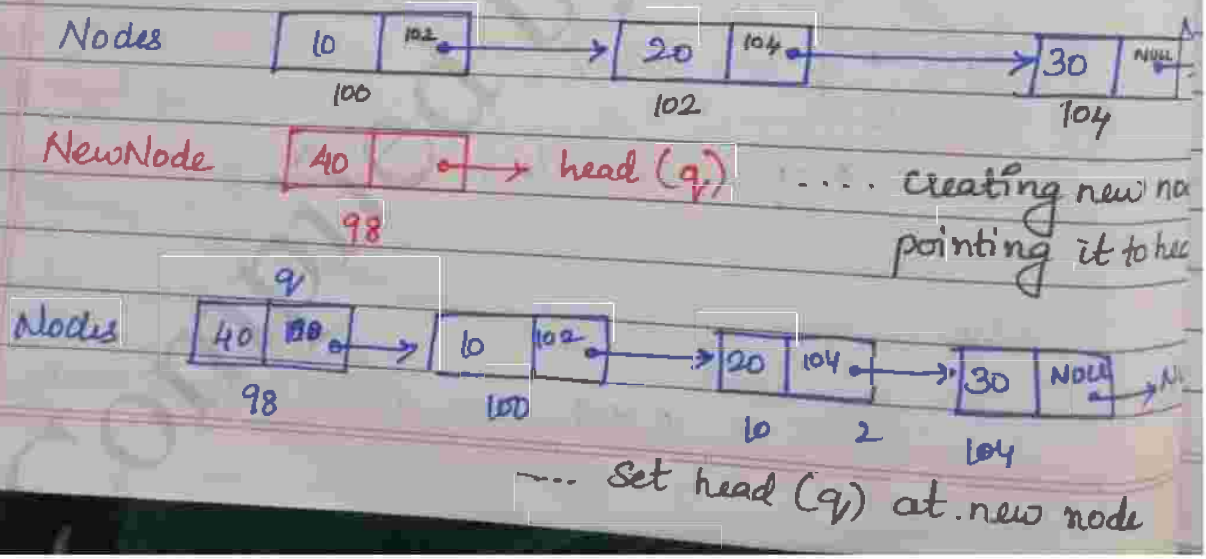
Algorithm for adding a node at the beginning of singly linked list.

Step 1 : set q at head // q = head
Step 2 : Read data

Step 3 : alloc (newNode)
newNode → data = data
Step 4 : newNode → next = q

Step 5 : Set q at newNode
// Set newNode as the new HEAD

⊙ Representation:-



* Algorithm to add a node at the middle of the Single Linked List.

Step 1 : temp = head.

Step 2 : read num to be inserted.

Step 3 : traverse temp to location (loc)

Step 4 : alloc (newNode)

Step 5 : newNode → data = num
Link newNode with next node. newNode→next = temp→next
Link newNode with previous node. temp→next = newNode

Here you can also use two pointers to traverse to the location as in the case of deletion.

* Representation:-

if you got to insert at second position, then new node next should point to current second (next) and address of first should point to newNode

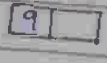
temp -> next = new node
and newNode -> next = temp -> next

new node will be created using malloc.

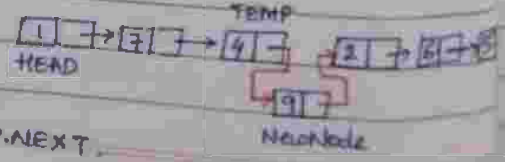
→ Here, we read the number to be inserted (num) and location to be inserted (loc) from the user.
→ Step 1 initializes a variable temp to the start of the linked list (pointed by head).
→ Then using a loop, we traverse to the location (loc) where the node has to be inserted.
→ Once we get that location, the the next pointers are changed in such a way that the new node is inserted after the desired node.

Another Example:-

⊙ Allocate mly for the newNode and initialize data part to 9.



⊙ Initialize temp = HEAD & traverse to location → 4



ie. TEMP.NEXT = NewNode
NewNode.NEXT = TEMP.NEXT.

Lab Name

Deleting a Node from the Singly Linked List:-

Here also we will discuss three cases:-

1) Deleting the first node
2) Deleting the last node
3) Deleting the node after a given node (i.e. from middle)

I) Deleting the first node from a Linked List :-

Algorithm:-

Step 1: If HEAD = NULL
Print "underflow"
Go to Step 5.

- PTR → here it is the reference given to HEAD, which we can used to traverse the entire list.

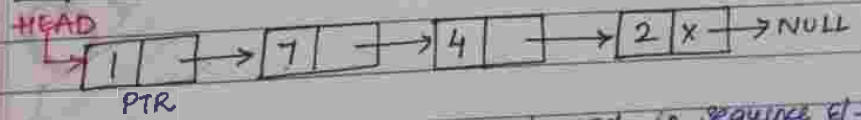
Step 2 : Set PTR = HEAD

Step 3 : Set HEAD = HEAD → NEXT

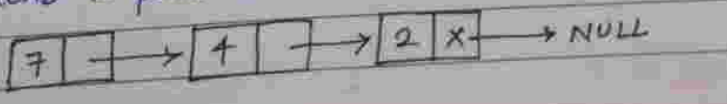
Step 4 : Delete PTR (i.e Free PTR)

Step 5 : Exit

Representation:-



- Make HEAD to point to the next node in sequence of free PTR.



- In the first step we check if the linked list exists or Not. If HEAD = NULL, then it implies 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 HEAD, that stores the address of the first node in the list.
- In Step 3, HEAD is made to point to the next node in the sequence and finally the memory occupied by the node pointed by PTR (initially the first node in the list) is freed and returned to the free pool.

* Deleting the last node from the singly linked list:-

Algorithm:

Step 1 : If HEAD = NULL,
Print "under-flow"
Go to step 8

Step 2 : Set PTR = HEAD

Step 3 : Repeat steps 4 and 5 until PTR → NEXT != NULL

Step 4 : Set PREPTR = PTR

Step 5 : PTR = PTR → NEXT

Step 6 : Set PREPTR → NEXT = NULL

Step 7 : Free PTR

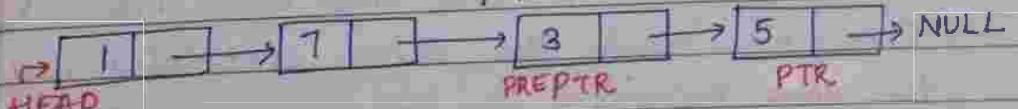
Step 8 : Exit.

* Example:-

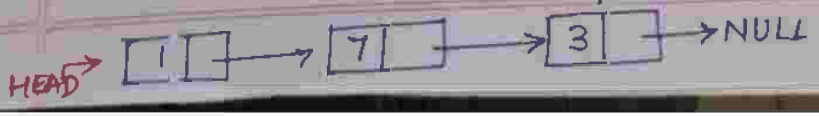
Let HEAD = PTR & PREPTR = PTR

Move PTR and PREPTR such that Next of PTR = NULL

PREPTR always points to the node just before PTR.



Set the next part of PREPTR to NULL.



* Deleting a node from the middle of the Linked List:-

*. Read the data to be deleted from user. (N

Step 1: If HEAD = NULL, then
P rn 't "Underflow"
Go to step 10.
[End of If]

Step 2: Set PTR = HEAD

Step 3: Set PREPTR = PTR

Step 4: Repeat steps 5 and 6 while PTR→ DATA ! = NUM

Step 5: Set PREPTR = PTR.

Step 6: Set PTR = PTR →.NEXT
[End of Loop]

Step 7: Set. TEMP = PTR

Step 8: Set PREPTR → NEXT = PTR→ NEXT

Step 9: Free TEMP

Step 10: Exit.

→ Here, we discuss about deleting a node after a given node from the singly linked list. In step 2, we take a pointer variable 'PTR' and initialize it to the starting node pointed by 'HEAD'.

→ we take another pointer variable PREPTR such that it always points to one node before PTR.

→ Once we reach the value containing node and the node succeeding it, we set the NEXT pointer

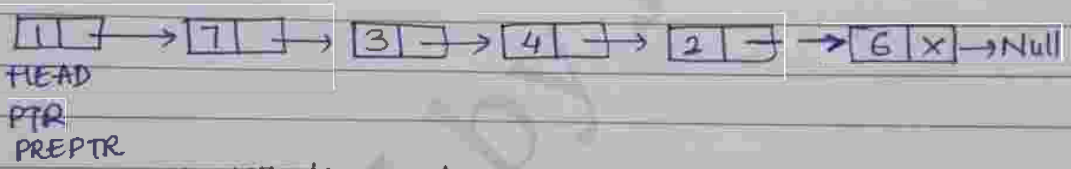
of the node containing value to be deleted, to the address contained in the next field of the node succeeding it.

→ The memory of the node succeeding the given node is freed and returned back to the free pool.

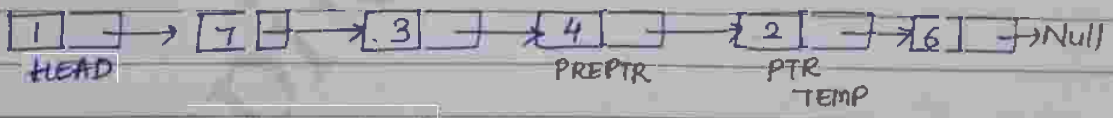
Representation:-

Here, node containing value 2 has to be deleted, so to traverse it to that node, we use two pointers PTR and PREPTR.

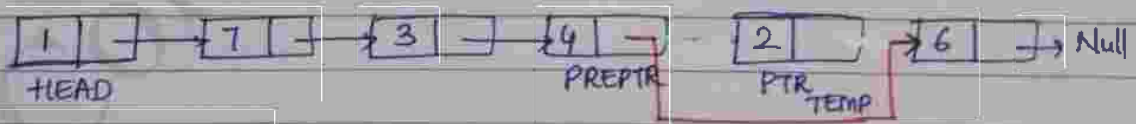
→ Firstly initialize PTR and PREPTR to HEAD.



→ Traverse it till the pointer PREPTR points to the node containing value = 2 and PTR points to its succeeding node.



Set the next part of PREPTR to the next part of PR



Free TEMP

UNIT - 5

GRAPHS

- A graph is a non-linear data structure
- A graph G is a collection of two sets V and E, where $V = \{V_1, V_2, V_3, \ldots V_n\}$ is a finite non-empty set of vertices and $E = \{e_1, e_2, e_3, \ldots e_n\}$ is a finite non-empty set of edges.
- Graph G is denoted by $G = \{V, E\}$

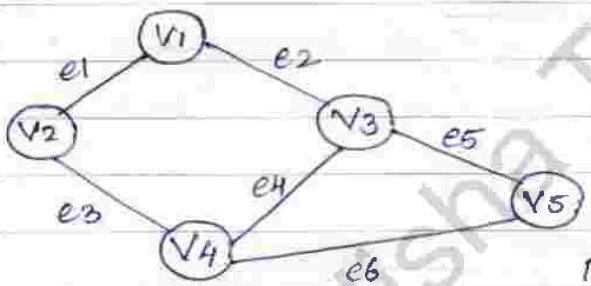


Fig: Graph G

$G = \{\{V_1, V_2, V_3, V_4, V_5\}, \{e_1, e_2, e_3, e_4, e_5, e_6\}\}$

* Graph Terminology:

1. Directed Graphs:

In the directed graph, the directions are shown on the edges. Sometimes to show direction like if its from Mumbai to Pune or Pune to Mumbai.

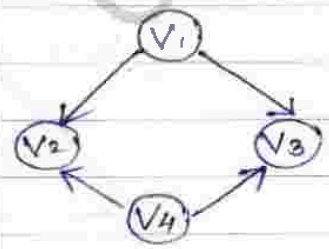


fig: directed Graph

Here, pair (V_1, V_3)
V_1 is called start vertex
and V_2 is end vertex.

(2) Undirected Graph:-

The graph in which edges do not show any direction is called as Undirected Graph.

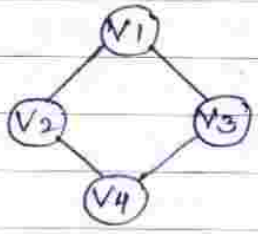
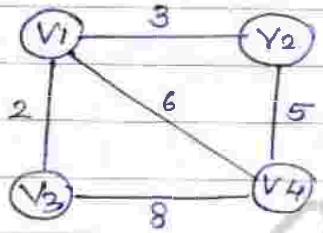


fig. undirected graph.

(3) Weighted Graph:-

When weight is assigned to each and every edge of a graph, then such graph is called weighted Graph.



(4) Complete Graph:-

- It is an undirected graph in which there is a direct edge between each pair of nodes.
- A complete graph with 'n' vertices has $\frac{n(n-1)}{2}$ number of edges.

In this, every vertex has an edge to all other vertices.

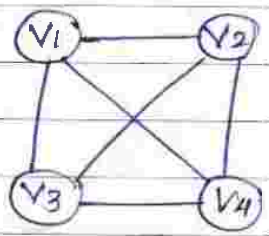
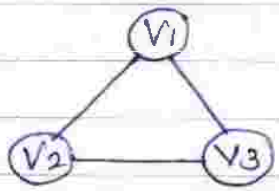


fig. complete graph.

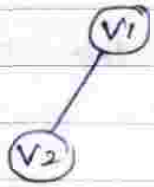
(5) Subgraph:-

. A subgraph G' of graph G is a graph, such that the set of vertices and set of edges of G' are proper subset of the set of edges of G.

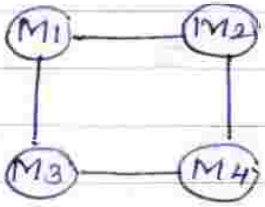
. ie A subgraph of G is another graph, formed from a subset of the vertices and edges of G.

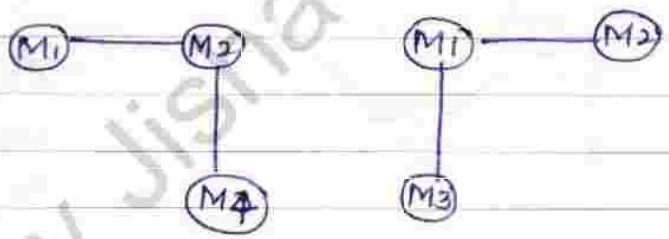


Graph G



Graph G'

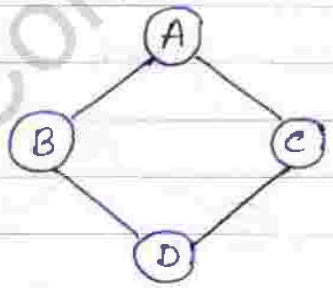




Subgraphs.

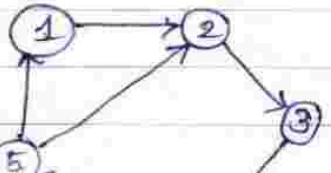
(6) Connected Graph :-

It is an undirected graph in which there is a path between each pair of nodes/vertices.



(7) Cycle :-

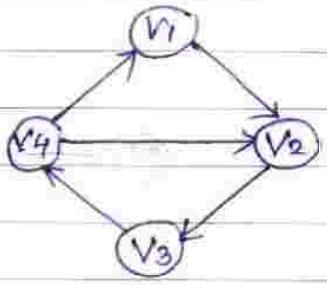
It is a path that starts and ends at the same vertex.



Cycle - 2-3-4-5-2-1-2-3-4-5-)

(8) Degree of vertices :-

- The degree of a vertex is the number of edges associated with it.
- In-degree of a vertex is the number of edges that incident to that vertex.
- Out-degree of a vertex is the number of edges going away from the vertex.



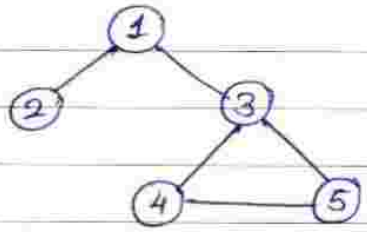
Vertices	In-degree	Out-degree
V_1	1	1
V_2	2	1
V_3	1	1
V_4	1	2

(9) Loop :-

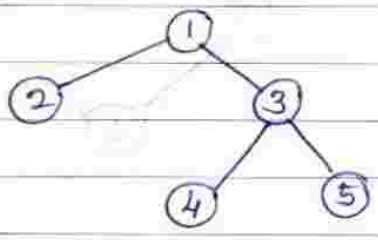
It is an edge whose end points are same, that is, $e = (4,4)$.

(10) Tree :-

A tree is a connected graph without any cycle.



a) Graphs.



b) Tree (Graph without cycle)

(*) Graph Representation :-

(1) Adjacency matrix :-

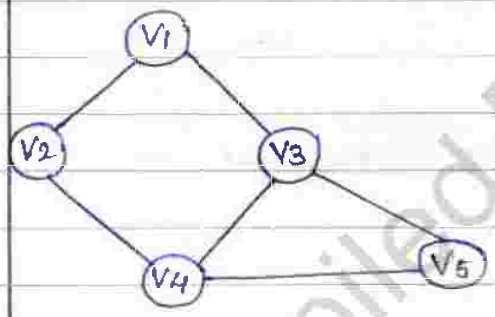
The graph represented using arrays is called adjacency matrix.

(2) Adjacency List :-

The graph represented using Linked List is called adjacency list.

I] Adjacency matrix :-

Consider a graph G of n vertices and the matrix M. If there is an edge present between vertices Vi and Vj then M[i][j] = 1 else M[i][j] = 0



v	1	2	3	4	5
1	0	1	1	0	0
2	1	0	0	1	0
3	1	0	0	1	1
4	0	1	1	0	1
5	0	0	1	1	0

Fig. An adjacency matrix representation.

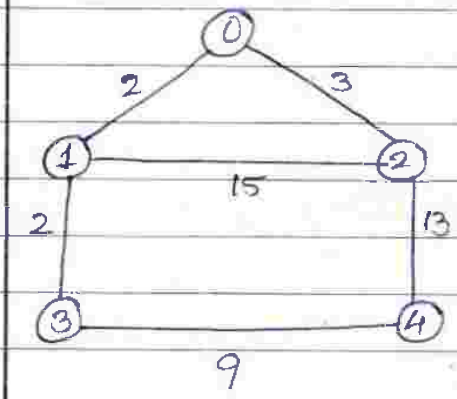
Advantage :-

1. Representation is easier to implement.
2. Removing an edge takes O(1) time.

Disadvantage :

1. Consumes more space. ie $O(n^2)$. If even the graph is sparse (contains less no. of edges)
2. Inefficient representation of graphs that have large no. of vertices.

Adjacency matrix representation of weighted graph:-

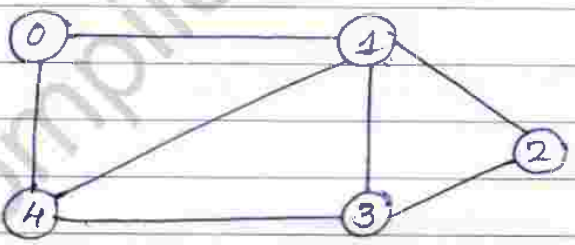


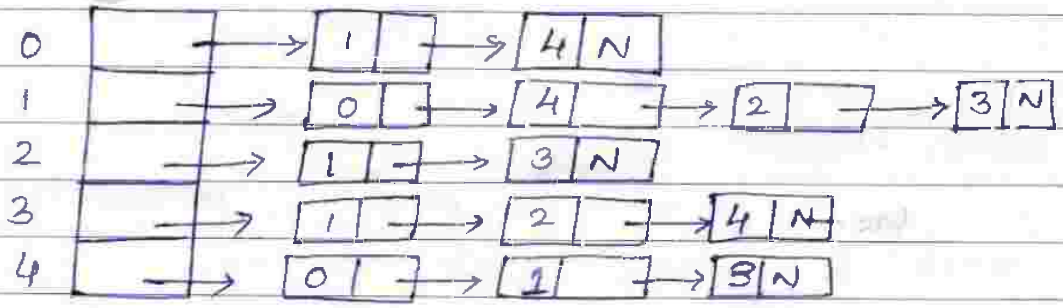
	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

fig: weighted graph and its adjacency matrix.

II) Adjacency List :-

- Adjacency list is a linked representation of a graph. It consists of a list of graph nodes with each node itself consisting of a linked list of its neighbouring nodes.
- All the advantages of a linked list can be obtained in this type of graph.





(*) Traversal of a graph :-

Imp → • Traversal of a graph means visiting each node of graph, exactly once.

• Two techniques are:
1. Breadth first search (BFS)
2. Depth first search (DFS)

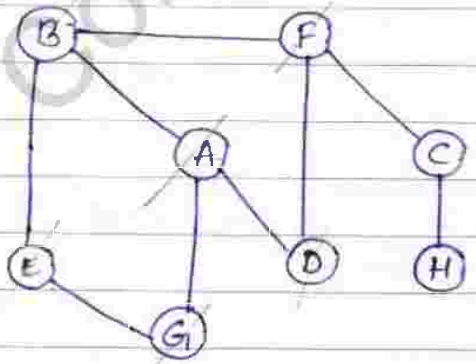
I] Breadth first search (BFS):-

* You can start traversal from any node as root, unless specified.

- Also called as level order traversal.
- This method starts from a given vertex v_0.
- v_0 is marked visited.
- All adjacent vertices to v_0 are visited next.

** In BFS the datastructure used is Queue (First in First out).
- The vertices which have been visited but not processed for adjacent vertices can be stored in queue.
  1. Initially the queue contains the starting vertex.
  2. In every iteration, a vertex is removed from the queue and its adjacent vertices which are not yet visited are added to queue.
  3. The algorithm terminates when the queue becomes empty.

Breadth first search (BFS):-



Result: A B

↳ Mark A as visited.

↳ Adjacent of A: B D G
↓
Add to Queue.

↳ Delete B from Queue & print it.

↳ Check adjacent unvisited of B: A, F, E
↓
Add to Queue

Queue:

A	B	D	G	F	E	C	H

RESULT: A B D G F E C H

Next, delete D from Queue and print it.
↳ Check adj of unvisited of D: F, A → Already
(Do not add anything)

Next is G, delete G and print it
↳ Check adj of unvisited vertex of G: A, F → Already added

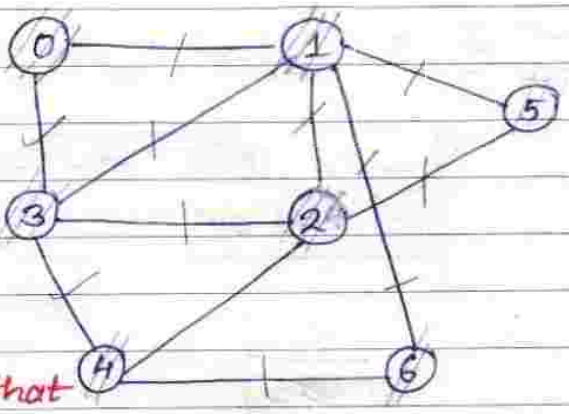
Next is F, check adjacent of unvisited: B, D, C → Add C to Queue.

Next is vertex E, delete and print E.
Adj of unvisited of E: B, G → already added.
Next is vertex C, delete and print C.
Check Adj and unvisited of C: F, H. → Add H to Queue.

Next is vertex H, check adj of H: E → Already visited
↓
Remove & print it.

∴ the BFS Traversal is: ABDGFECH.

Ex.(2) Traverse the given Graph using BFS.



Here, no vertex is specified as starting point, so we can start from any vertex.
Suppose if 0 is taken as root, add that to queue.

Queue

	0	1	2	3	4	5	6
0	0	1	0	1	0	0	0
1	1	0	1	1	0	1	1
2	0	1	0	1	1	1	0
3	1	1	1	0	1	0	0
4	0	0	1	1	0	0	1
5	0	1	1	0	0	0	0
6	0	1	0	0	1	0	0

(Deleted 0 and printed 0 in result)

RESULT: 0132 564

As soon as 0 is printed, adjacent vertices of 0 ie. 1 and 3 would be inserted in queue. (Order does not matter)

Now, 1 is deleted and printed.
Now adjacent and unvisited vertices of 1 will be entered in queue. 0 3 2 5 6

Now, 3 is deleted from queue and printed.
Adjacent vertices of 3: 0 1 4 2 → Add 4 to queue

Now 2 is deleted from queue and printed.
Adjacent vertices of 2: 1, 3, 5, 1 → Nothing is added.

Now 5 is deleted from queue and printed.
Adjacent vertices of 5: 1, 2 → Nothing is added

Now 6 is deleted from queue and printed.
Adjacent vertices of 6: 1, 4 → Already visited

Now 4 is deleted from queue and printed.
Adjacent vertices of 4: 2 3 1 → Already

II ⊛ **Depth First Traversal (DFS):**

— In DFS, the datastructure used in STACK (Last in First Out principle).

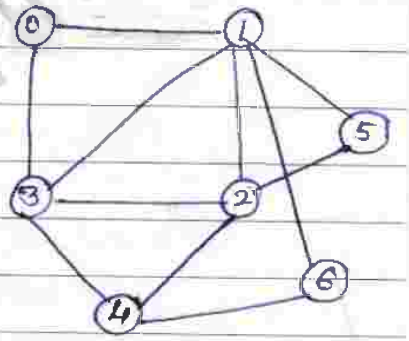
~~6~~
~~5~~
2
3
1
~~0~~

STACK

— Same like BFS, if root is not specified, then any node can be taken as root.

— Depth First means deeper & deeper until you reach dead-end.

— Unlike in BFS, where all adjacent vertices were added into queue, here only one vertex will be added into stack.



** RESULT : 0, 1, 3, 2, 4, 6, 5.

Now, Adjacent vertices of 0 : 1, 3. Add any 1 to stack. Suppose we add 1.

Now 1, Adjacent vertices : ~~3~~, ~~2~~, 6, 5. Take any one and add to stack.

Take 3, Adjacent vertices : ~~0~~, ~~1~~, (2) 4. Push 2 to stack

Take 2, Adjacent vertices : ~~3~~, 5, (4) Push 4 to stack

Take 4, Adjacent vertices : 3, 2, (6) Push 6 to stack

Now after 6, there are no more unvisited adjacent vertices of 6. So start Back.

In such situation:

Top most element is popped out. ie. 6

After 6,

Check 4 if there are any adjacent unvisited vertex of 4. No. so pop 4 out.

After 4, check 2.

Adjacent unvisited of 2: yes → 5.

5 would be printed and pushed into the stack.

Now 5 does not have any unvisited elements.

∴ Pop 5.

After 5, 2 is the new top.

Check 2 for any unvisited adjacent → NO → Pop 2

Next top is 3, check unvisited adjacent → NO → POP 3

Nex top is 1, Check —"— —"— → NO → POP 1

Next top is 0, —"— —"— → NO → POP 0.

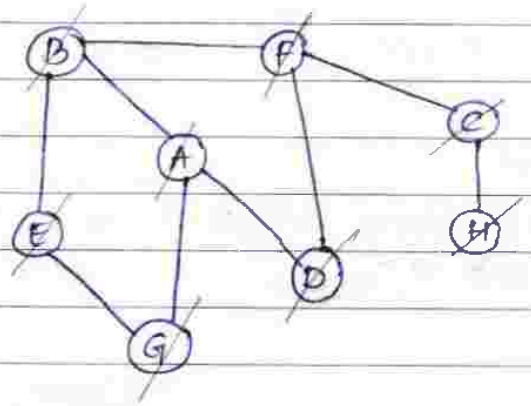
Now stack is Empty, indicating where to stop the algo.

And the RESULT: 0, 1, 3, 2, 4, 6, 5 shows the DFS traversal of our graph.

(* But not the only one, there can be numerous)

* try to explore deeper.

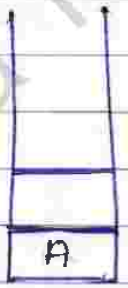
E18(2)



Solution:-

As no vertex is specified, lets start with A.

Push A onto stack, mark A as visited.



STACK

RESULT: A

Now, check vertex at the top of stack.

ie A → check adjacent of A : (B), G. → Add B to stack & mark B as visited.

ie.

Stack

RESULT: A B

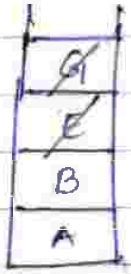
↳ Now check top of stack → B.

check adjacent of B: (E), A, F → Add E to stack and mark E as visited.

Result : A B E

↳ Now, again check the top of stack, ie. E

Adjacent of E : G, B ~~B~~ → Add G to stack and mark G as visited



RESULT : A B E G

↳ Top of stack is G.

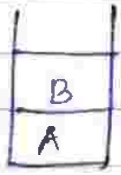
Adjacent of G : A, E → already visited.
(Now back-track)

pop G from stack.

↳ Now E is the top.

Adjacent of E : B, G → already visited.

Pop E.



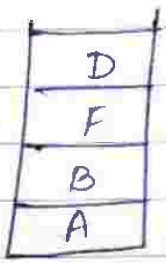
↳ Now B is the new top: Adjacent of B = F ~~A~~ ~~E~~

Add F to stack and mark F as visited.

RESULT : A B E G F

↳ Now F is the new top : Adjacent of F = ~~B~~, (D), C

Add D to stack & mark D as visited.

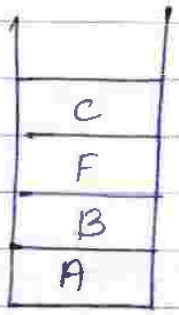


RESULT : A B E G F D

↳ Now D is the top : Adj(D) ⇒ A, F → Already visited.

Pop D from stack

Now F is top → Adj(F) ⇒ C, D, B ⇒ Add C to stack & mark visited.



... Result : A B E G F D C

Now top = C

Unvisited / Adj(C) ⇒ H, F

Add H to stack and mark as visited.

Result : A B E G F D C H

Now, top = H

Adj(H) ⇒ C → Pop H.

top = C, Adj(C) ⇒ F, H → already visited → Pop C

top = F, Adj(F) = B, D, C → already visited → Pop F

⋮

top = A, Adj(A) = A, G, D → already visited → Pop A.

Now Stack is Empty. which indicates that the traversal can be terminated.

∴ DFS = A B E G I F D C H J.

AVL Trees

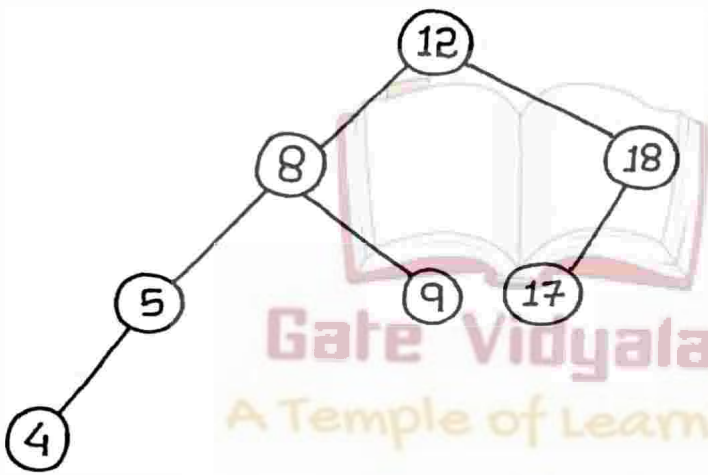
Definitions-

AVL trees are self-balancing Binary Search Trees where the difference between heights of left and right subtrees cannot be more than one for all nodes.

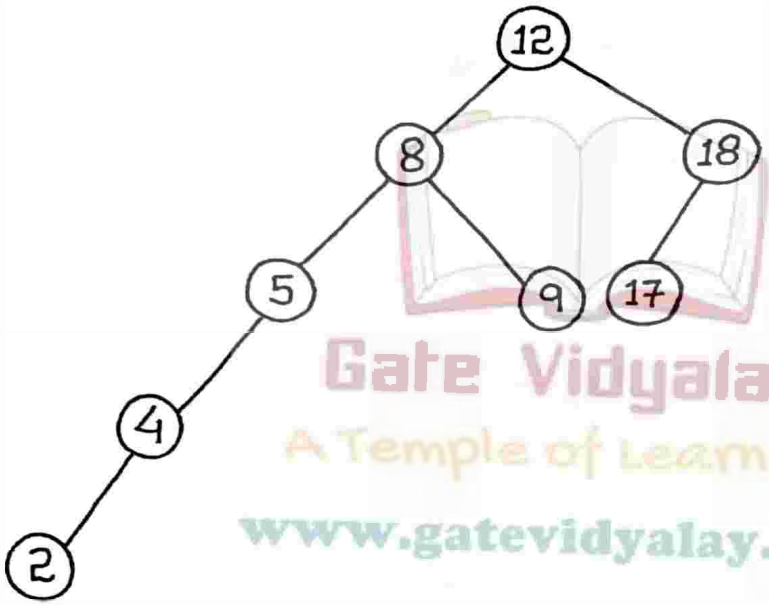
OR

AVL trees are Binary Search Trees in which the heights of the left and right subtrees of every node differs by at most one.

Example of AVL Tree -



Example of not an AVL Tree-



Balance Factor-

Balance factor for any node is defined as-

> Balance factor = Height of left subtree − Height of right subtree

For a node to be balanced, the value of balance factor has to be either 0 or 1 or -1.

How balancing is done in AVL Trees?

In AVL Tree, after performing every operation like insertion and deletion, we check the balance factor of every node in the tree.

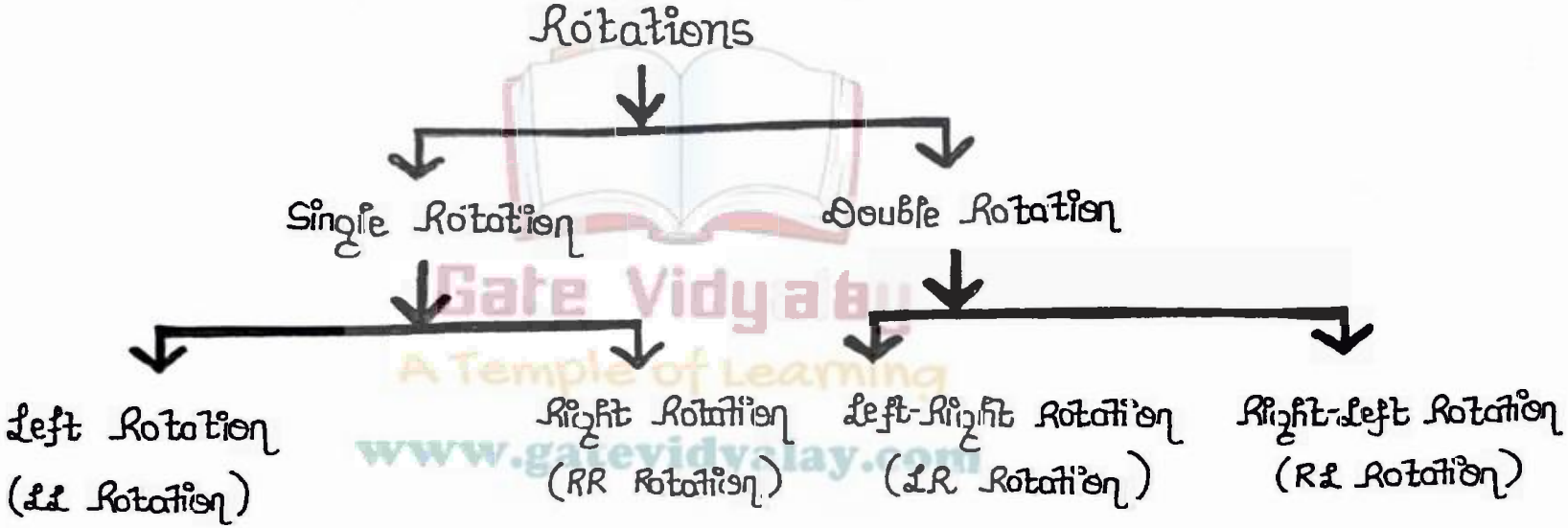
If every node satisfies the balance factor condition, then the operation is concluded otherwise we must make it balanced.

We use rotation operations to make the tree balanced when the tree becomes unbalanced due to any operation.

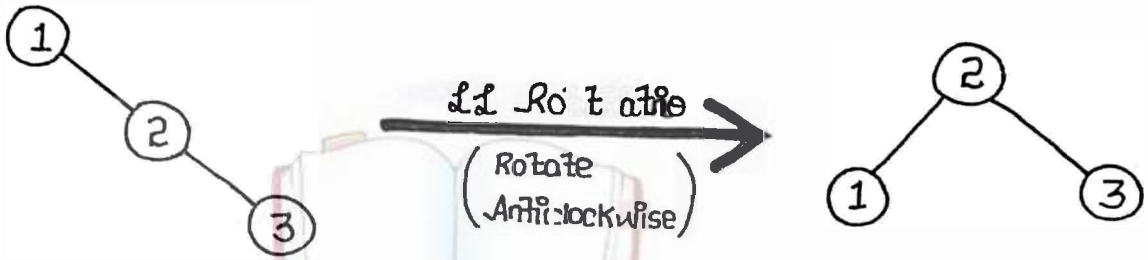
Thus,

> Rotation is the process of moving the nodes to make tree balanced.

Kinds of Rotations-



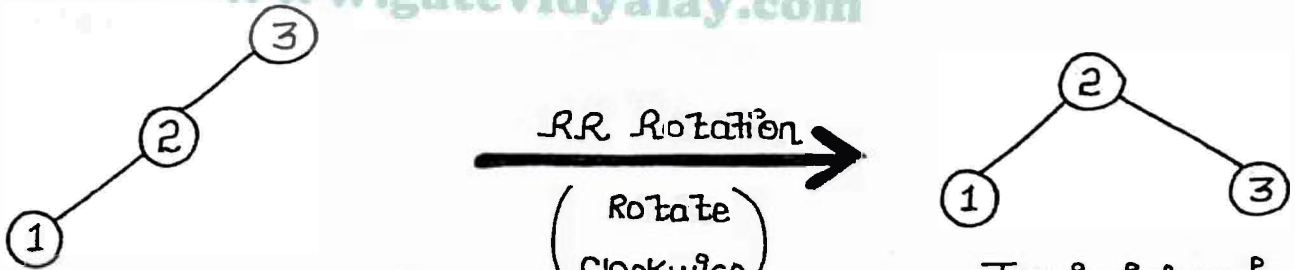
Case-I:



(Insertion order: 1,2,3)
Tree is Imbalanced

Tree is balanced

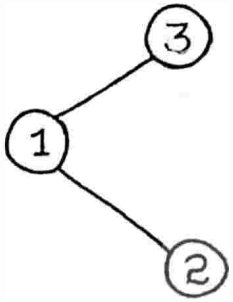
Case-II:



(Insertion order: 3,2,1)
Tree is Imbalanced

Tree is balanced

Case-III:

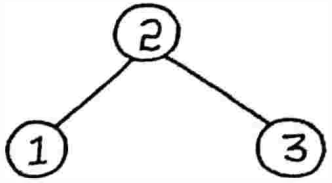


(Insertion Order : 3,1,2)

Tree is Imbalanced

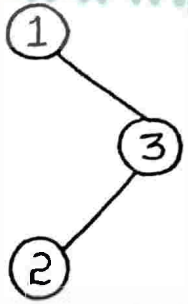
LL Rotation + RR Rotation

(LR Rotation)



Tree is balanced

Case-IV:

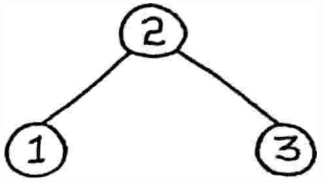


(Insertion Order : 1,3,2)

Tree is Imbalanced

RR Rotation + LL Rotation

(RL Rotation)



Tree is balanced

Operations on AVL Trees-

The following operations are performed on AVL trees-

i) Search

ii) Insertion

iii) Deletion