BYJU'S EXAM PREP

Computer Science & IT

C Programming & Data Structures

SHORT NOTES

Short Notes — C PROGRAMMING & DATA STRUCTURES

Chapter 1: C programming

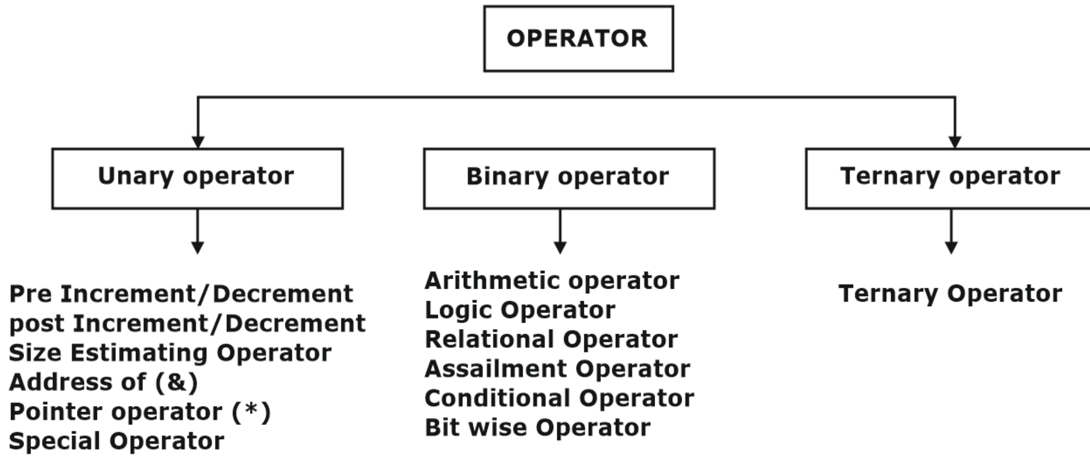
- **C TOKENS:**
  The smallest individual units recognizable by the lexical analyser are known as C tokens. C has six types of tokens: Keywords, Identifiers, Constants, Operators, Delimiters / Separators and Special symbols.
  E.g.: int, _name, +, - , ++, -- etc.

- **Data types:**

Types	Data Types
Basic data types	Int, char, float, double
Enumeration data type	enum
Derived data type	Pointer, array, structure, union
Void data type	void

Data Types	Size (in bytes)
short int	1
int	2
long int	4
char	1
float	4
double	8

- **TYPES OF OPERATORS:**



Operators	Symbols
Arithmetic operators	+, -, *, /, %, ++, --
Assignment operator	=, +=, -=, *=, etc
Relational operators	<, <=, >, >=, !=, ==
Logical operators	&&, ||, !
Bitwise operators	&, |, ~, ^, <<, >>
Special operators	sizeof(), comma
Pointer operators	* - Value at address (indirection), & - Address Operator

- **Operator Precedence table:**

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through a pointer	
	(type){list }	Compound literal(C99)	

<table>
<tr><th>Precedence</th><th>Operator</th><th>Description</th><th>Associativity</th></tr>
<tr><td rowspan="8">2</td><td>++ --</td><td>Prefix increment and decrement</td><td rowspan="8">Right-to-left</td></tr>
<tr><td>+ -</td><td>Unary plus and minus</td></tr>
<tr><td>! ~</td><td>Logical NOT and bitwise NOT</td></tr>
<tr><td>(type)</td><td>Type cast</td></tr>
<tr><td>*</td><td>Indirection (dereference)</td></tr>
<tr><td>&</td><td>Address-of</td></tr>
<tr><td>Sizeof</td><td>Size-of</td></tr>
<tr><td>_Alignof</td><td>Alignment requirement(C11)</td></tr>
<tr><td>3</td><td>* / %</td><td>Multiplication, division, and the remainder</td><td rowspan="11">Left-to-right</td></tr>
<tr><td>4</td><td>+ -</td><td>Addition and subtraction</td></tr>
<tr><td>5</td><td><< >></td><td>Bitwise left shift and right shift</td></tr>
<tr><td rowspan="2">6</td><td>< <=</td><td>For relational operators < and ≤ respectively</td></tr>
<tr><td>> >=</td><td>For relational operators > and ≥ respectively</td></tr>
<tr><td>7</td><td>== !=</td><td>For relational = and ≠ respectively</td></tr>
<tr><td>8</td><td>&</td><td>Bitwise AND</td></tr>
<tr><td>9</td><td>^</td><td>Bitwise XOR (exclusive or)</td></tr>
<tr><td>10</td><td>|</td><td>Bitwise OR (inclusive or)</td></tr>
<tr><td>11</td><td>&&</td><td>Logical AND</td></tr>
<tr><td>12</td><td>||</td><td>Logical OR</td></tr>
<tr><td>13</td><td>?:</td><td>Ternary conditional</td><td rowspan="6">Right-to-Left</td></tr>
<tr><td rowspan="5">14</td><td>=</td><td>Simple assignment</td></tr>
<tr><td>+= -=</td><td>Assignment by sum and difference</td></tr>
<tr><td>*= /= %=</td><td>Assignment by product, quotient, and remainder</td></tr>
<tr><td><<= >>=</td><td>Assignment by bitwise left shift and right shift</td></tr>
<tr><td>&= ^= |=</td><td>Assignment by bitwise AND, XOR, and OR</td></tr>
<tr><td>15</td><td>,</td><td>Comma</td><td>Left-to-right</td></tr>
</table>

- **C FLOW CONTROL STATEMENTS:**

1. **If Statement:**

```
if (condition)
{
statement1
statement2
:
}
```

2. **If else statement:**

```
if (condition)
    {statements}
else
    {statements}
```

3. **The switch Statement:**

```
switch (control variable)
{
case constant-1: statement(s);
break;
case constant-2: statement(s);
break;
   :
case constant-n: statement(s);
break;
default: statement(s);
    }
```

Note: Here constant can be either character or integer

4. **Loop Control Structure:**

4.1. While Loop:

```
while (testExpression)
{
   // statements inside the body of the loop
}
```

4.2. do while Loop:

```
do
{
  // statements inside the body of the loop
```

```
}
while (testExpression);
```

4.3. for Loop:

```
for (initializationStatement; testExpression; updateStatement)
{
   // statements inside the body of loop
}
```

5. Unconditional Flow Control Statements:

5.1. The break Statement: The break statement is used to instantly jump out of a loop, without waiting to get back to the conditional test.

5.2. The continue Statement: The 'continue' statement is used to take the control to the beginning of the loop, by passing the statement inside the loop, which has not yet been executed.

5.3. goto Statement: C supports an unconditional control statement, goto, to transfer the control from one point to another in a C program.

Chapter 2 : Recursion

- **FUNCTION:**

  There are three aspects of a C function:

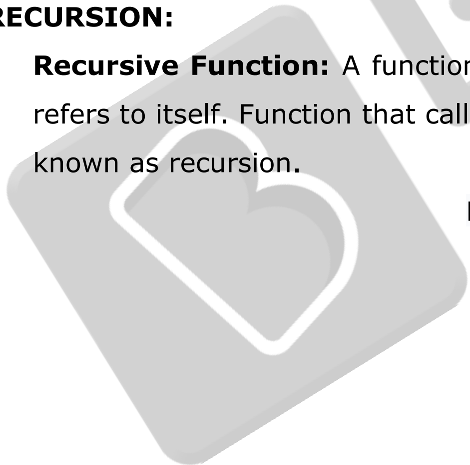
  1. **Function declaration** A function must be declared globally in a c program to tell the compiler about the function name, function parameters, and return type.
  2. **Function call** Function can be called from anywhere in the program. The parameter list must not differ in function calling and function declaration. We must pass the same number of functions as it is declared in the function declaration.
  3. **Function definition** It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is called. Here, we must notice that only one value can be returned from the function.

- **TYPES OF FUNCTION CALL:**

  1. **Call by value in C :** This method copies the actual value of an argument into the function's formal parameter. In this case, changes made to the parameter inside the function do not affect the argument.
  2. **Call by reference in C** : This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

- **RECURSION:**

  - **Recursive Function:** A function is called recursively defined if the definition of the function refers to itself. Function that calls itself is known as a recursive function. And this technique is known as recursion.



How does recursion work?

```
void recurse ( )
{
    ...  ...  ...
    recurse ( );      // recursive call
    ...  ...  ...
}
int main ( )
{
    ...  ...  ...
    recurse ( );
    ...  ...  ...
}
```

For this the definition of the function should satisfy the following conditions :

- There must be certain arguments for which the function does not refer to itself, these arguments are called base values or base conditions.
- Each time the function does not refer to itself, its argument must be closer to a base value or base condition.
- The base condition(s) must be defined before the recursive call in the function definition.

Example :
```
#include<stdio.h>
#include<math.h>
int rec_fun(int n)
{
    Static int count = 0;
    if(n ≤ 0)                        //Base condition
      return 0;
    if(n%2 == 0)
      count++;
    rec_fun(n/2);                //Recursive call
    return count;
}
int main()
{
    int m, i, k;
    printf("\nEnter any number >0 : ");
    scanf("%d", &m);
    i = rec_fun(m);                  //Function call by value
    k = ceil(log2(m)) - i;
    printf("\nNumber of zeros in binary representation = %d\nNumber of ones in binary representation = %d", i, k);
    return 0:
}
```

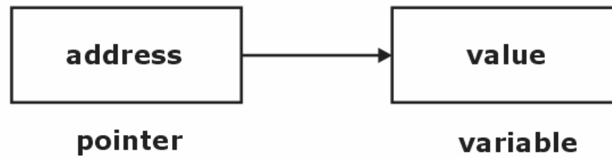
INPUT : Enter any number >0 : 78

OUTPUT : Number of zeros in binary representation = 3

Number of ones in binary representation = 4

Chapter 3 : Arrays & Linked List

- **Pointers:** It is a variable which stores the address of another variable.



1. **Declaring a pointer:**

e.g:
```
int *p;
char *c;
```

2. **Usage of Pointers:**

2.1. Pointer to array:

e.g:
```
int arr[10];
int *p[10]=&arr;            //pointer to array
```

2.2. Pointer to a function:

e.g:
```
void show (int);            //function declaration
void(*p)(int) = &show;      //pointer to function
```

2.3. Pointer to structure:

e.g:
```
struct emp                  //structure
{
int id;
char name[20];
}e1;
struct emp *p;      //pointer to structure
p = &e1;
```

2.4. NULL Pointer: A pointer that is not assigned any value but NULL is known as the NULL pointer.

e.g:
```
int *p = NULL;
```

3. **Advantage of pointer:**

3.1. Pointer reduces the code and improves the performance.

3.2. We can return multiple values from a function using the pointer.

3.3. It makes easy to access any memory location in the computer's memory.

BYJU'S EXAM PREP

3.4. It makes Dynamic memory allocation possible.

4. STORAGE CLASSES IN C:

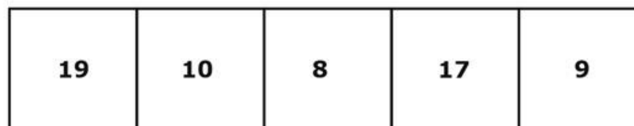
Storage Classes	Storage Place	Default Value	Scope	Lifetime
auto	RAM	Garbage Value	Local	Within function
extern	RAM	Zero	Global	Till the end of the main program May Be declared anywhere in the program
static	RAM	Zero	Local	Till the end of the main program, Retains value between multiple functions call
register	Register	Garbage Value	Local	Within the function

- **Arrays:** It is a linear data structure. It is a collection of similar elements having same data type.

1. Declaration and Initialization:

e.g: int mark[5]; //declaration

mark[5] = {19, 10, 8, 17, 9}; //initialization



2. TYPES OF ARRAYS:

2.1. One dimensional (1-D) arrays or Linear arrays:

Consider a single dimensional array as **A[lb------ub]**

The base address of array = BA

Size of each element in array = c

Total elements in array is given by (ub-lb+1)

Then address of any random element A[i] is given by = BA + (i-lb+1)*c

2.2. Two dimensional (2-D) arrays or Matrix arrays: 2-D arrays can be stored in the system in two ways: Row Major order and Column Major order.

E.g: **$A[lb_1$-----$ub_1]$ $[lb_2$------$ub_2]$**

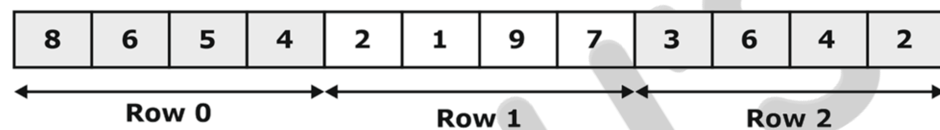
The base address of array = BA

Size of each element in array = c

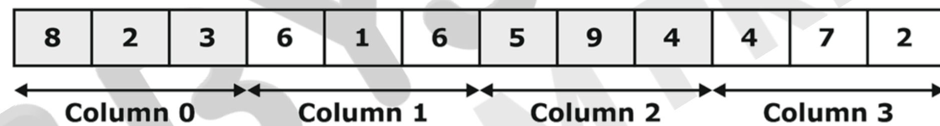
Number of rows = $ub_1 - lb_1 + 1$

Number of columns = $ub_2 - lb_2 + 1$

Row-Major (Row Wise Arrangement)



Column-Major (Column Wise Arrangement)



2.2.1. Row Major order:

Address of A[i][j] = $BA + [(i - lb_1)\times(ub_2 - lb_2 + 1) + (j - lb_2)] \times c$

2.2.2. Column Major order:

Address of A[i][j] = $BA + [(j - lb_2) \times(ub_1 - lb_1 + 1) + (i - lb_1)] \times c$

3. POINTERS & ARRAYS: An array variable is just a pointer to the first element in the array.

A[i] ≡ *(A+i)

A[i][j] ≡ *(*(A+i) + j)

A[i][j] ≡ *(A[i] + j)

&A[i][j] ≡ *(A+i) + j

4. STRINGS: Strings are defined as an array of characters. The difference between a character array and a string is that the string is terminated with a special character '\0'(Null character).

Syntax: char str_name[size];

e.g: char c[] = "abcd";

c[0]	c[1]	c[2]	c[3]	c[4]
a	b	c	d	\0

- **STRUCTURE:** Structure is a user-defined datatype in C language which allows us to combine data of different types together.

1. DEFINING A STRUCTURE:

```
struct [structure_tag]
{
    //member variable 1
    //member variable 2
    //member variable 3
    ...
}[structure_variables];
```

1.1. DECLARING STRUCTURE VARIABLES:

```
struct student
{
    //member variables
}s1,s2;              //declaration
Struct student s3,s4;    //declaration
```

1.2. ACCESSING STRUCTURE MEMBERS: In order to assign a value to any structure member, the member name must be linked with the **structure** variable using a dot (.) operator also called **period** or **member access** operator.

Syntax: s1.member_variable_name

2. SELF REFERENTIAL STRUCTURES: Structures pointing to the same type of structures are self-referential in nature.

e.g:
```
struct node {
        int data1;
        char data2;
        struct node* link;
        };
```

2.1. Types of Self Referential Structures:

2.1.1. Self Referential Structure with Single Link.

2.1.2. Self Referential Structure with Multiple Links.

3. **APPLICATIONS**: Self referential structures are very useful in creation of other complex data structures like:

   3.1. Linked Lists
   3.2. Stacks
   3.3. Queues
   3.4. Trees
   3.5. Graphs etc.

- **UNIONS:** A union is a special data type available in C that allows to stores different data types in the same memory location.

  1. **Defining a Union:**

```
union [union tag] {
        member definition;
        member definition;
        ...
        member definition;
        } [one or more union variables];
```

e.g:

```
union Data {
        int i;
        float f;
        char str[20];
        } data;
```

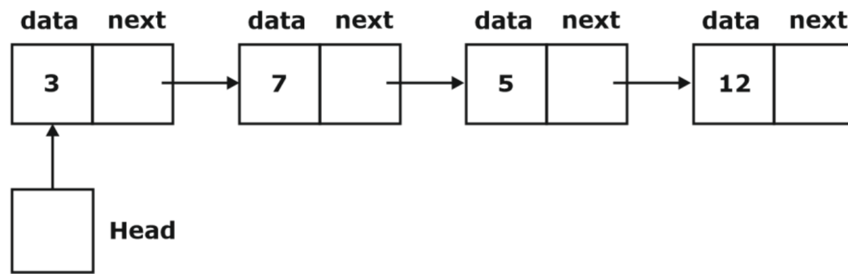
  2. **Accessing Union Members:** To access any member of a union, we use the member access operator (.).

     e.g: data.i or data.f or data.str[5]

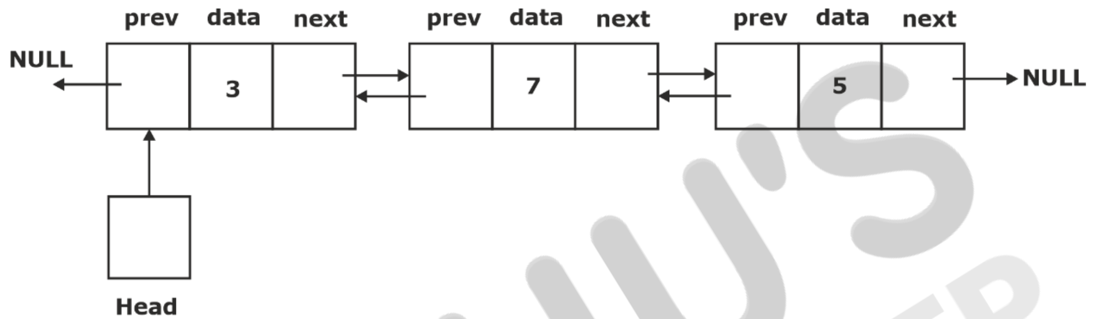
- **LINKED LIST:** Linked List is a linear data structure which consists of group of nodes in a sequence.

  1. **Operations:**
     1.1. Insertion.
     1.2. Deletion.
     1.3. Traversing.

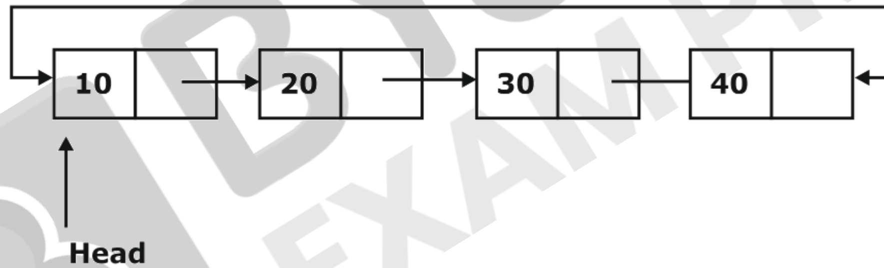
  2. **TYPES OF LINKED LIST:**
     2.1. **Singly Linked list:**



2.2. Doubly linked list:



2.3. Circular Linked List:



3. TIME COMPLEXITY:

Operations →	Insertion			Deletion		
Types of Linked list ↓	At Starting	At Middle	At End	At Starting	At Middle	At End
Singly Linked List	0(1)	0(n)	0(1)	0(n)	0(n)	0(n)
Doubly Linked List	0(1)	0(n)	0(1)	0(1)	0(n)	0(n)
Circular Linked List	0(1)	0(n)	0(1)	0(n)	0(n)	0(n)

BYJU'S EXAM PREP

Chapter 4 : Stacks & Queues

- **STACKS:** A stack is a last in first out (LIFO) abstract data type and data structure.
  1. **Stack Operations:**
     1.1. PUSH: Inserting an item into a stack.
     1.2. POP: Deleting an item from the stack.
     1.3. PEEK: Displaying the contents of the stack.
- **EXPRESSION NOTATION:**
  1. **Infix Expression:** Here, the binary operator comes between the operands. It is of the form <left><data><right> or <operand><operator><operand>.
     e.g: (a+b)/c
  2. **Postfix Expression:** Here, the binary operator comes after both the operands. It is of the form <left><right><data> or <operand><operand><operator>. A postfix expression is a parenthesis-free expression. For evaluation, we evaluate it from left-to-right. It is also known as Polish Notation.
     e.g: ab+c/
  3. **Prefix Expression:** Here, the binary operator comes before both the operands. It is of the form <data><left><right> or < operator > < operand > < operand >**.** It is also known as Reverse Polish Notation.
     e.g: /+abc
- **APPLICATIONS OF STACK:**
  1. **Infix to Postfix:** Operator stack is used for infix to postfix conversion.

e.g: Infix : (A+B)/C to Postfix : AB+C/

Stack	Input	Output
Empty	(A+B)/C	-
(	A+B)/C	-
(	+B)/C	A
(+	B)/C	A
(+	)/C	AB
Empty	/C	AB+
/	C	AB+
/	END	AB+C
Empty	END	AB+C/

2. **Postfix Evaluation:** Operand stack is used for evaluation. Scan the postfix expression from left to right.

e.g: 75+6/

Step	Input Symbol	Operation	Stack	Calculation
1.	7	Push	7	
2.	5	Push	7,5	
3.	+	Pop (2 elements & evaluate)	Empty	7+5 = 12
4.		Push result (30)	12	
5.	6	Push	12, 6	
6.	/	Pop (2 elements & evaluate)	Empty	12/6 = 2
7.		Push result (2)	2	
8.		No-more elements (pop)	Empty	2 (Result)

3. **Prefix Evaluation:** Operand stack is used for evaluation. Scan the prefix expression from right to left.

e.g: /+756

Symbol	Opnd1	Opnd2	Value	Opndstack
6				6
5				6, 5
7				6, 5, 7
+	7	5	12	6
				6, 12
/	12	6	2	Empty
				2

4. **Prefix to Postfix:**

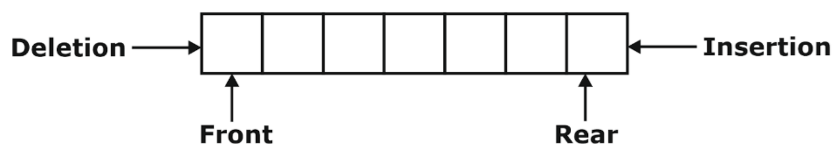
e.g: Prefix : +-435 to Postfix : 43-5+

Symbol	Opnd1	Opnd2	Value	Opndstack
5				5
3				5, 3
4				5, 3, 4
-	4	3	43-	5
				5, 43-
+	43-	5	43-5+	
				43-5+

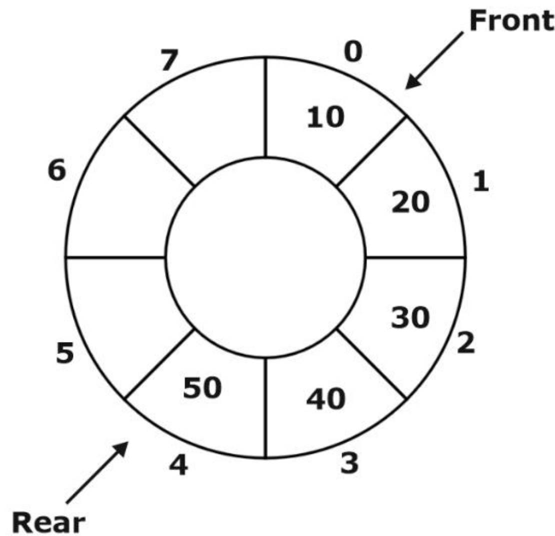
5. **Recursion using Stacks:** In recursion the last function called needs to be completed first. As Stack is a LIFO data structure i.e. (Last In First Out) and hence it is used to implement recursion.

- **QUEUES:** Queues follow the **First In First Out (FIFO)** i.e. the first element that is added to the queue is the first one to be removed. Elements are always added to the back and removed from the front.
  1. **Queue Operations :**
     1.1. Enqueue : Inserting an item into a queue.
     1.2. Dequeue : Deleting an item from the queue.

- **TYPES OF QUEUES:**
  1. **Simple Queue:** Insertion occurs at the rear (end) of the queue and deletions are performed at the front (beginning) of the queue list.
  All nodes are connected to each other in a sequential manner.



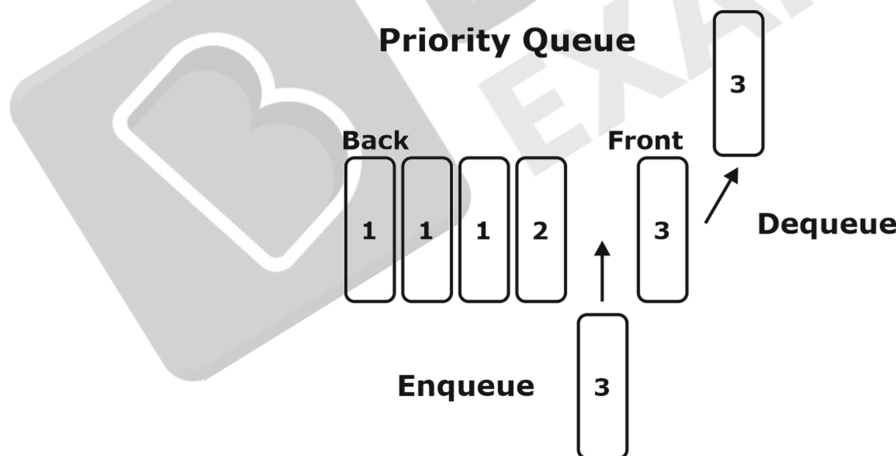
2. Circular Queue: A circular queue overcomes the problem of un-utilized space in linear queues implemented as arrays.



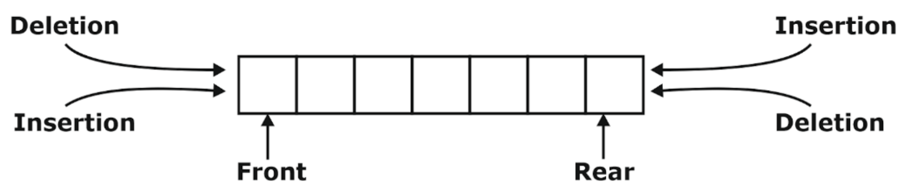
2.1. If : (Rear+1) % n == Front, then queue is Full

2.2. If Front = Rear, the queue will be empty.

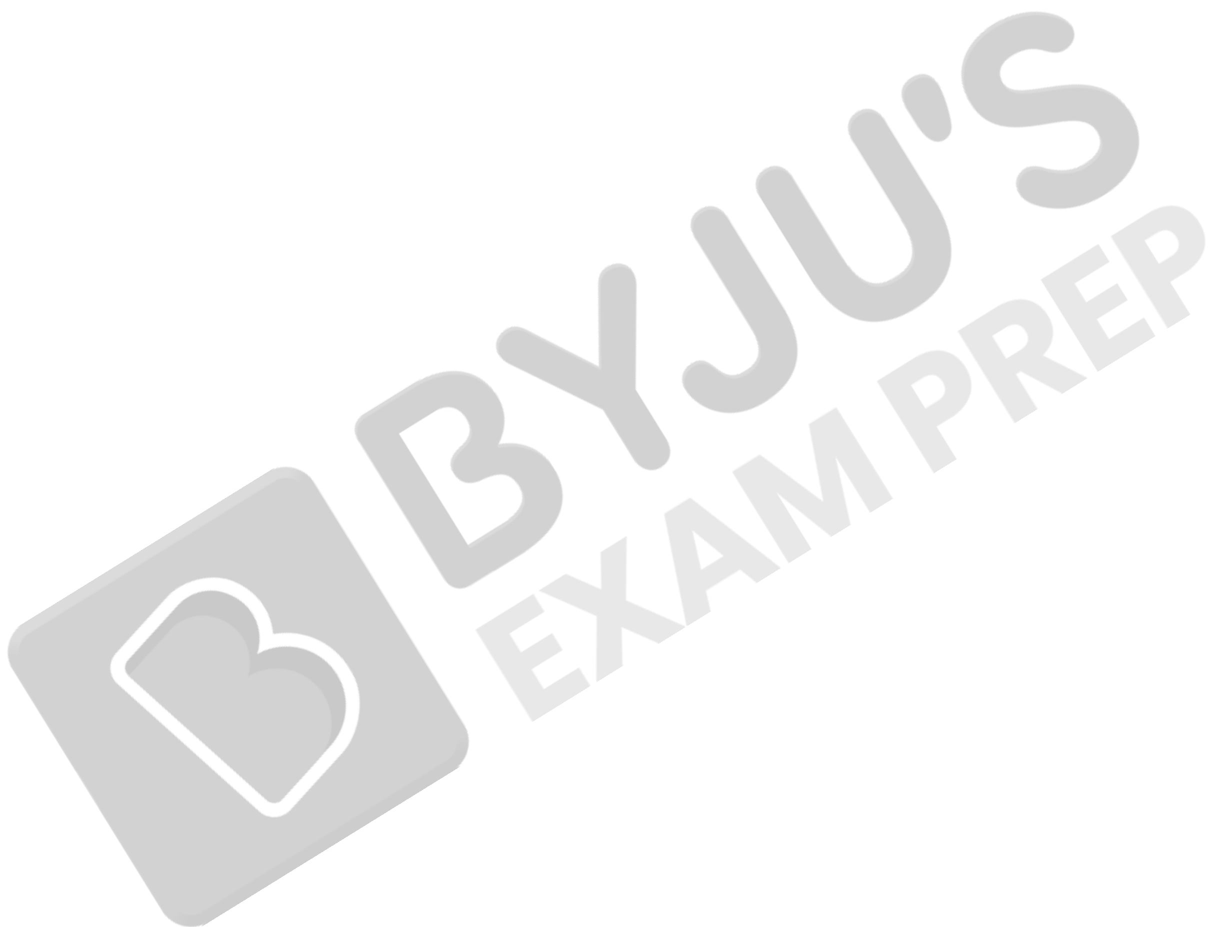
3. Priority Queue: While the deletion is performed in accordance with priority number (the data item with highest priority is removed first), insertion is performed only in the order.



4. Doubly Ended Queue (Deque): The doubly ended queue or deque allows the insert and delete operations from both ends (front and rear) of the queue.



- **APPLICATIONS OF QUEUE:**
  1. Breadth first Search can be implemented.
  2. CPU Scheduling.
  3. Handling of interrupts in real-time systems.
  4. Routing Algorithms.
  5. Computation of shortest paths.
  6. Computation a cycle in the graph.



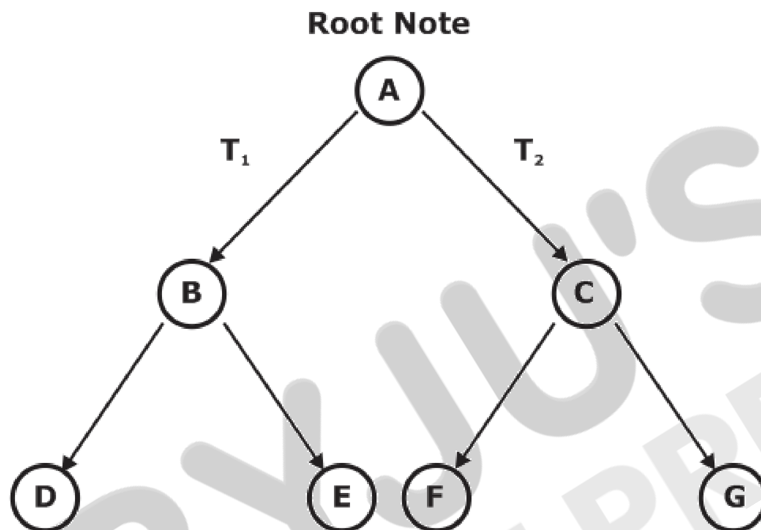
BYJU'S EXAM PREP

Chapter 5 : Trees

- **TREES:** A tree is a non-linear data structure designated at a special node called the root and elements are arranged in levels without containing cycles.

1. TYPES OF TREES:

1.1. Binary Tree: It is a special type of tree where each node of tree contains either 0 or 1 or 2 children.



Time Complexities:

Insertion: O(n) {Every Case}

Deletion: O(n) {Every Case}

1.2. Types of Binary Trees:

1.2.1. Full Binary Tree: If each node of binary tree has either two children or no child at all, is said to be a **Full Binary Tree**.

1.2.2. Complete Binary Tree: If all levels of tree are completely filled except the last level and the last level has all keys as left as possible, is said to be a **Complete Binary Tree**.

1.2.3. Skewed Binary Tree: If a tree which is dominated by left child node or right child node, is said to be a **Skewed Binary Tree**.

1.2.4. Strict Binary Tree: If every non-leaf node in a binary tree has nonempty left and right subtrees, the tree is called a strict binary tree. A strict binary tree with n leaves always contains 2n -1 nodes.

1.2.5. Binary Search Tree: A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties –

left subtree (keys) ≤ node (key) ≤ right subtree (keys)

Time Complexities:

Insertion: O(n) {worst Case}
O(1) { Best Case}
O(logn) { Average Case}

Deletion: O(n) {worst Case}
O(1) { Best Case}
O(logn) { Average Case}

2. TREE TRAVERSAL:

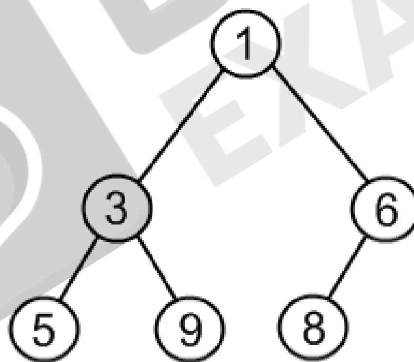
2.1. Pre-order : (D, L, R)

2.2. In-order : (L, D, R)

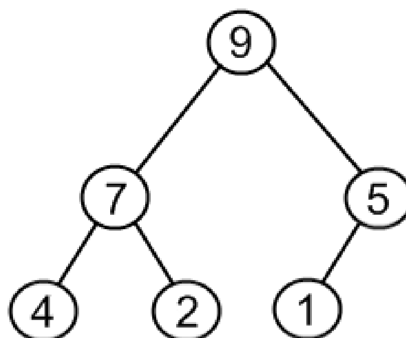
2.3. Post-order : (L, R, D)

- **BINARY HEAPS:** Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly.

1. Min-Heap – Where the value of the root node is less than or equal to either of its children. The same property must be true for all subtrees.



2. Max-Heap – Where the value of the root node is greater than or equal to either of its children. The same property must be true for all subtrees.



- **AVL TREE:** AVL tree is a height balanced tree. It is a self-balancing binary search tree.

Balance Factor = Height of left subtree – Height of right subtree

Domain of Balance Factor = {-1, 0, +1}

Time Complexities: Insertion: O(logn)

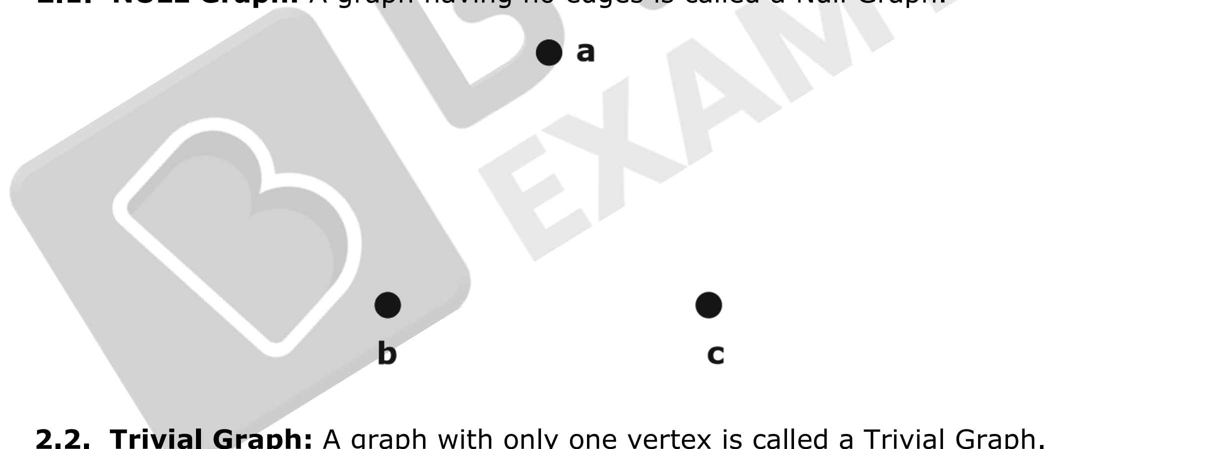
Deletion: O(logn)

1. **Types of Rotations:**
   1.1. **Left Rotation (LL-Rotation)**
   1.2. **Right Rotation (RR-Rotation)**
   1.3. **Left-Right Rotation (LR-Rotation)**
   1.4. **Right-Left Rotation (RL-Rotation)**

- **Graphs:** It is a collection of vertices and edges connecting two such vertices. It is represented as **G(V, E)** where **V** is the set of vertices and **E** is the set of edges.

1. **Graph Operations:**
   1.1. **Add Vertex**
   1.2. **Add Edge**
   1.3. **Display Vertex**
2. **Types of Graphs**:

2.1. NULL Graph: A graph having no edges is called a Null Graph.

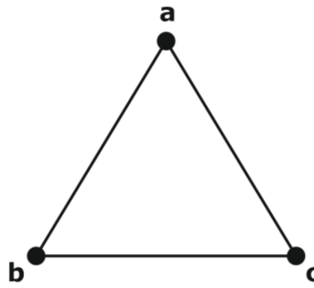


2.2. Trivial Graph: A graph with only one vertex is called a Trivial Graph.

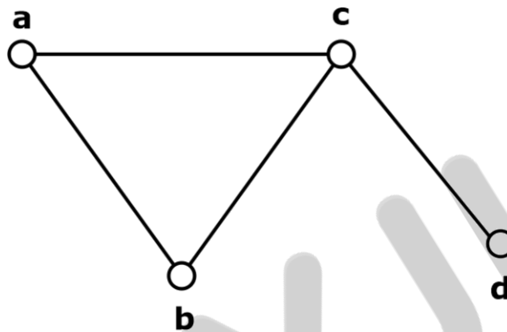
2.3. Non-Directed Graph: A non-directed graph contains edges, but the edges are not directed ones.

2.4. Directed Graph: In a directed graph, each edge has a direction.

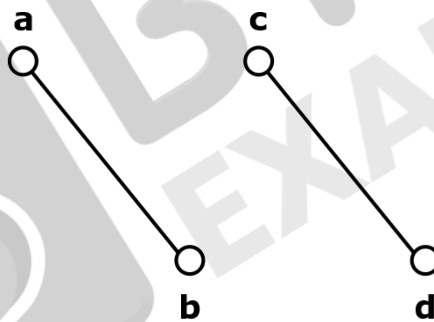
2.5. Simple Graph: A graph with no loops and no parallel edges is called a simple graph. The number of simple graphs possible with 'n' vertices = $2^{n(n-1)/2}$.



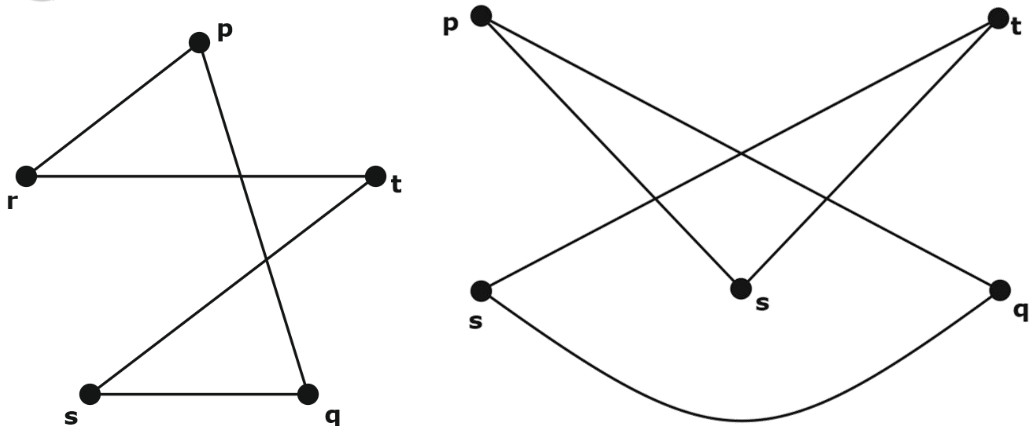
2.6. Connected Graph: A graph G is said to be connected if there exists a path between every pair of vertices.



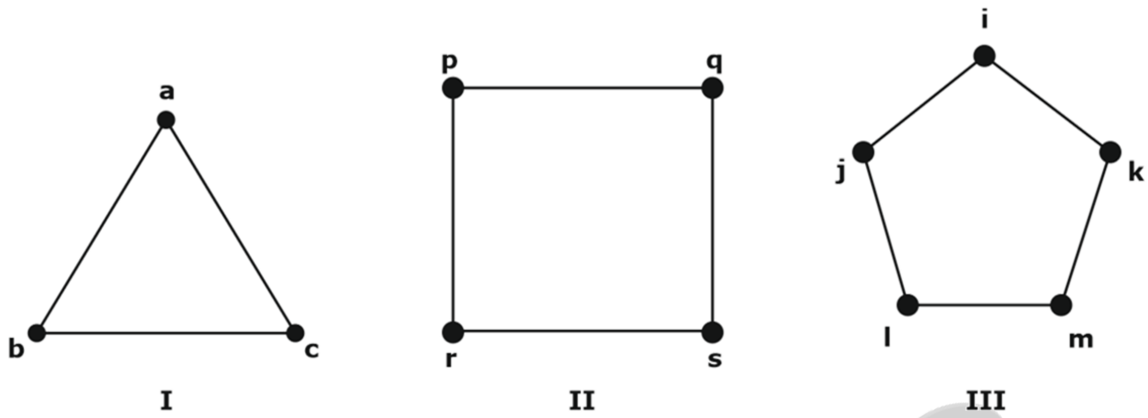
2.7. Disconnected Graph: A graph G is disconnected if it does not contain at least two connected vertices.



2.8. Regular Graph: A graph G is said to be regular, if all its vertices have the same degree. In a graph, if the degree of each vertex is 'k', then the graph is called a 'k-regular graph'.



2.9. Cycle Graph: A simple graph with 'n' vertices (n >= 3) and 'n' edges is called a cycle graph if all its edges form a cycle of length 'n'.



- **Graph Traversal:**

1. DFS (Depth First Search): The result of a DFS traversal of a graph is a spanning tree. Spanning Tree is a graph without loops. We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal.

Time complexity: $O(V + E)$ when Adjacency List is used
$O(V^2)$ when Adjacency Matrix is used

2. BFS (Breadth First Search): Breadth First Search (BFS) algorithm traverses a graph in a breadth-ward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

Time complexity: $O(V + E)$ when Adjacency List is used
$O(V^2)$ when Adjacency Matrix is used

****