DATA STRUCTURES AND ALGORITHMS

Concepts, Techniques and Applications

GAV PAI

Scanned by CamScanner

CORE PAPER VI.

DATA STRUCTURES.

AND ALGORITHMS

Concepts, Techniques, operations and Applications.

[For Semester 3]

Scanned by CamScanner

B.C.A.-DATA STRUCTURES

UNIT I: Data Structures – Definition and Classification - Arrays – Array Operations – Representation of Arrays – Applications of Arrays. Stack – Operations on Stacks -Stack applications – Infix to postfix notation and Evaluation of Postfix notation. Queues - Operations on the Queues - Circular queue – Dequeue - Priority queue - Applications of queue.

UNIT II: Introduction to the Linked List - Basic operations on linked list – Singly Linked Lists – Doubly Linked Lists – Circularly Linked Lists – Applications of Linked List – Addition of Polynomials – Linked Stacks – Linked Queues.

UNIT III: Trees - Basic Terminology - Binary Trees - Representation of Trees and Binary trees – Binary Tree Traversals – Threaded Binary Trees – Binary Search Tree - Insertion and Deletion operations at Binary Search Trees - Applications of Trees.

UNIT IV: Graphs – Basic Terminology – Representation of Graphs - Graph Traversals – BFS, DFS - Applications of Graphs – Minimum Cost Spanning tree and Dijkstra's Shortest Path. Searching Techniques – Linear Search – Binary Search – Fibonacci Search.

UNIT V: Sorting techniques – Bubble Sort, Selection Sort, Quick Sort, Merge Sort, and Insertion Sort – Introduction to Hashing – Hash table structure – Hash Functions – Linear Open Addressing – Chaining.

TEXT BOOKS:

1. **S. Sahni & E. Horowitz,** "*Fundamentals of Data Structure*", Second Edition, 1999, Galgotia Publications.
2. **Richard F Gilberg Behrouz A. Forouzan, Thomson,** "*Data Structure A Pseudocode Approach with C*", First Edition, 2005, University of California.
3. **Seymour Lipschutz,** "*Schaum's Outline of Theory and Problems of Data structure*", Second Edition, Tata McGraw-Hill.
4. **Alfred V.Aho, John E.Hopcraft, Jeffrey D.Ullman,** "*Data Structures and Algorithms*", 2009, Addison-Wesley.

REFERENCE BOOKS:

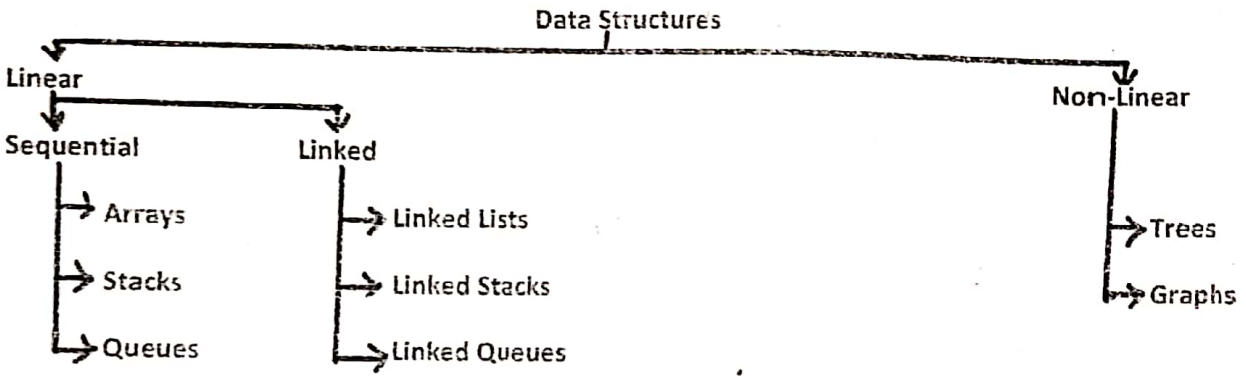
1. **G.A. Vijayalakshmi Pai,** "*Data structures and Algorithms- Concepts, Techniques and Applications*", First Edition, 2011, Tata McGraw-Hill.
2. **Dr. A. Chitra,** "*Data Structures*", 2009, Vijay Nicole Imprints Private Limited.

E- REFERENCES:

- http://nptel.iitm.ac.in/courses/Webcourse-contents/IIT-%20Guwahati/data_str_algo/frameset.htm
- http://www.personal.kent.edu/~rmuhamma/Algorithms/algorithm.html
- en.wikibooks.org/wiki/Data_structures

Scanned by CamScanner

The major tasks done in any computer system are based on 3 activities storing, accessing, manipulations of data. People need to have an abstract thinking to understand the principles and techniques to solve the vast array of unfamiliar problems that arise in a rapidly changing field. Even a good programmer may unable t o write new algorithms for the given problem if he has no knowledge in data structures. Hence "**Data structure is a collection of data elements organized in a specified manner and accessing functions are defined to store and retrieve individual data elements**". The classification of data structure is as given below:



A linear data structure is one in which the elements have a linear ordering between them. An element can be marked as first, second, third etc. A nonlinear data structure is one in which such an ordering is not present.

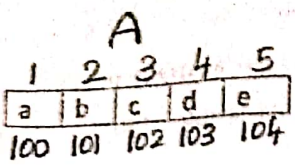
ARRAYS

An array refers to the homogeneous collection of elements stored contiguously in memory. Arrays follow static allocation of memory (ie) once the size of the array is declared it cannot be expanded nor contracted. The general features of an array are:

- A finite collection of elements.
- All elements of an array are of same data type.
- It is an ordered collection since we can label the elements as first, second, third etc.
- Each element of an array is stored adjacent to each other (ie) follows contiguous memory allocation.
- All elements are referred by same and single name.
- It is a random access data structure (ie) any element can be picked up and manipulated without sequential traversal.
- Individual element of an array is addressed by an integer called as index or subscript. It denotes the relative position of the element within the array from the starting position of the array.
- The first index of the array is called as Lower Bound while the last index of the array is called as Upper Bound. Bounds depend upon the programming language.

Scanned by CamScanner

Eg: char A[5]



Types of Arrays and its Representation in Memory

According to the number of index, the arrays can be broadly classified into three types as One-Dimensional Array, Two-Dimensional Array and Multi-Dimensional Array.

- One-Dimensional Array: It is the simplest form of array. It indicates a linear set of elements. It is declared as

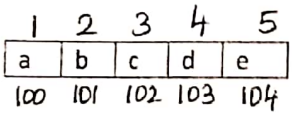
  datatype arrayname[size];

Eg:int a[100]; defines an integer array of size 100. The array declaration involves both the type of element that will be contained in the array as well as the maximum number of elements that will be stored inside the array. It is also called as Vectors. a[100] can be written as a[1:100].

Number of elements in the array=u-l+1
Address of any ith element in the array=α+(i-l)
where u=upper bound, l=lower bound and α=base or starting address of the array.

Eg:-The following figure shows the representation of one-dimensional array in memory.

char A[5];



Here u=5,l=1,α=100

Number of elements in A=5-1+1=5

Address of A[3]=100+(3-1)=100+2=102.

- Two-Dimensional Array: They are popularly called as matrices. As the name implies, this type of array is having two indices-one for row and other for column. It is usually denoted as

Arrayname[l1:u1,l2:u2] where l1=lower bound of row, u1=upper bound of row, l2=lower bound of column, u2=upper bound of column.

Eg: Representation of 3 subject marks for 4 students may be given as the following matrix.

Scanned by CamScanner

	Col 1	Col 2	Col 3
Row 1	80	95	92
Row 2	85	76	89
Row 3	91	90	98
Row 4	77	65	35

2D arrays are logical representation while the memory storage is single dimension. Hence there should be a mapping between 2D representation and single dimension. This mapping should suggest a systematic way of storing the elements of a two-dimensional array into a one-dimensional array. This can be done in two ways as row-major representation in which the elements of the row gains priority first for storing while in column-major representation, the elements of the column gains priority first for storing in memory.

Eg: Row Major Representation of the above matrix

(1,1)	(1,2)	(1,3)	(2,1)	(2,2)	(2,3)	(3,1)	(3,2)	(3,3)	(4,1)	(4,2)	(4,3)
80	95	92	85	76	89	91	90	98	77	65	35

Column-Major Representation of the above matrix

(1,1)	(2,1)	(3,1)	(4,1)	(2,1)	(2,2)	(2,3)	(2,4)	(3,1)	(3,2)	(3,3)	(3,4)
80	85	91	77	95	76	90	65	92	89	98	35

Number of elements in the array=(u1-l1+1)(u2-l2++1)

Address of any (i,j)th element in the array=α+(I-l1) (u2-l2+1)+(j-l2)

where α=base or starting address of the array.

Eg: int A[4,3] is a matrix with 4 rows and 3 columns. Consider the values at the matrix as,

(1,1)	(1,2)	(1,3)	(2,1)	(2,2)	(2,3)	(3,1)	(3,2)	(3,3)	(4,1)	(4,2)	(4,3)
80	95	92	85	76	89	91	90	98	77	65	35
100	101	102	103	104	105	106	107	108	109	110	111

With l1=1,u1=4,l2=1,u2=3, α=100

Number of elements in the above array=(4-1+1)(3-1+1)=12

Address of [2,3]rd element in the given array=100+(2-1)(3-1+1)+(3-1)=100+(1)(3)+2=105

- Multi-Dimensional Array: If the index of the array goes beyond 2, it is normally referred as multi-dimensional array.

Eg: To denote the x,y,z axis in image processing, the array can be defined as A[x.,y,z]

Scanned by CamScanner

Primitive Operations on Array

The basic operations on an array are extraction (retrieval) and storing values.

- Extraction refers to getting the value of an element stored in an array as X=Data[i], where we are reading the value present at ith position in the array Data and storing it in Variable X.
- The operation of storing is just opposite to retrieval. Here a value is stored in an element of the array as Data[i]=X, where the value X is stored at ith position in the given array Data.

Applications of array

The following are the two applications of array.

1. SPARSE MATRIX: A matrix is an arrangement of m.n elements in m rows and n columns. The sparse matrix is a matrix with 0's as dominating elements. Eg: In a university of 20000 students say there are 5 electives. If all students are needed to select only 1 elective, that matrix will become sparse.

Usually a matrix consumes a lot of space in memory (ie) a 50x50 matrix needs 2500 locations-If that matrix is sparse, most of the locations will have 0's. So the representation of sparse matrix is developed as a triplet (ie) [i,j,value] used to represent the non-zero value in the sparse matrix.

If A is the given sparse matrix, then B will be the sparse matrix representing A with one row extra than that of A. Usually the first row of B will give some statistics like the total number of rows, total number of columns and total number of non-zero elements in A. The remaining row will tell about the exact row and column position of the non-zero element along with its value at A.

Eg: Consider the following sparse matrix

$$A=\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 3 & 0 & 0 \end{bmatrix}$$

The following is the representation of the above sparse matrix

$$B=\begin{bmatrix} 4 & 4 & 3 \\ 1 & 1 & 2 \\ 3 & 3 & 1 \\ 4 & 2 & 3 \end{bmatrix}$$

2. ORDERED LISTS: One of the simplest and useful data objects in computer science is an ordered list or linear list. It may be empty or non-empty. The elements of the list are called as atoms. Variety of

Scanned by CamScanner

operations like retrieval, insertion, deletion, [illegible] can be done here. The most common [illegible] represent ordered list is one dimensional array. Sequential mapping is established here.

Eg. i) (sun,mon,tue,wed,thu,fri,sat)

ii) (a1,a2,a3,.....an)

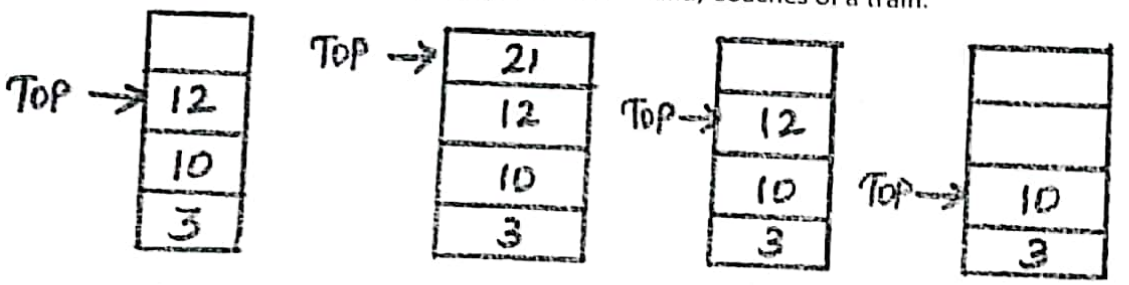
iii) (UNIX, Windows,Linux)

OPERATION	ORIGINAL LIST	RESULTANT LIST
insert a6	a1,a2,a4,a7	a1,a2,a4,a6,a7
delete a4	a1,a2,a4,a6,a7	a1,a2,a6,a7
update a6 to a9	a1,a2,a6,a7	a1,a2,a7,a9

STACKS

Linear Lists allow the insertion and deletion of elements at any position. If we put a restriction on the lists such that the insertion and deletion can occur only at one end, we get an important subclass of lists called as Stack. In stack, the user can add or delete the element only from one end called as Top of the stack. Manipulation is done only at the top item.

Eg: Pile of coins, Pile of plates, Bangles in Ladies hand, Coaches of a train.



A new element 21 is added at the top of the stack. Similarly the element to be deleted is also 21 the top of the stack. During deletion, the element 12 is removed.

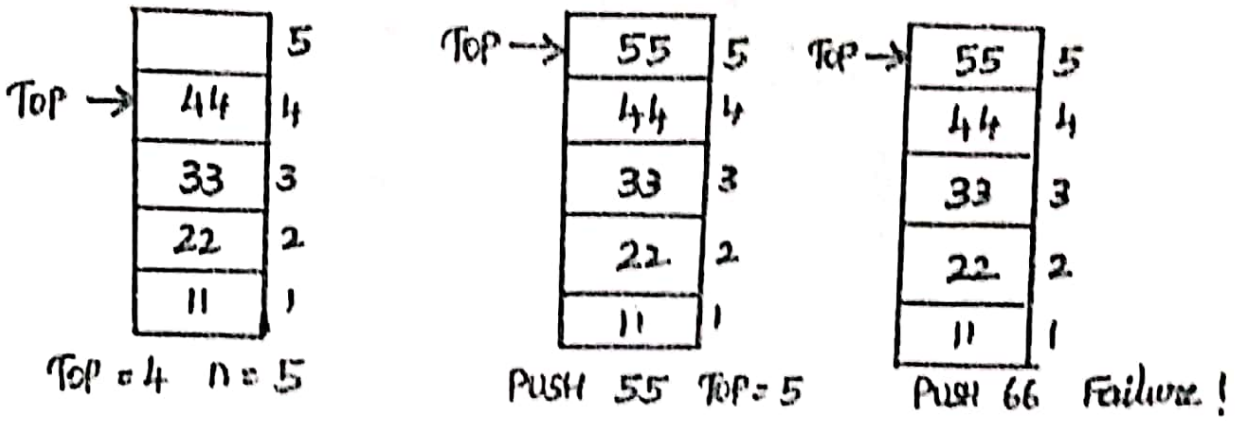
The insertion of the element is called as PUSH and the deletion of the element is called as POP. It needs to be observed that during insertion of elements into the stack it is essential that their identities are specified, whereas for removal no identity need be specified since by virtue of its functionality, the element which occupies the top of stack position is automatically removed. Since the elements inserted into the stack join last and those added last are the first to be removed, the stack belongs to LIFO (Last In First Out) data structure.

NOTE: The stack can be represented as an array STACK[n] i.e., it can store "n" elements. It is therefore becomes essential to issue a warning called STACK-FULL when we try to insert more elements (PUSH) than "n". Similarly when we try to delete an element (POP) from the stack which is already empty, a warning should be issued to indicate STACK-EMPTY. Therefore overflow and underflow should be properly indicated to the users.

Scanned by CamScanner

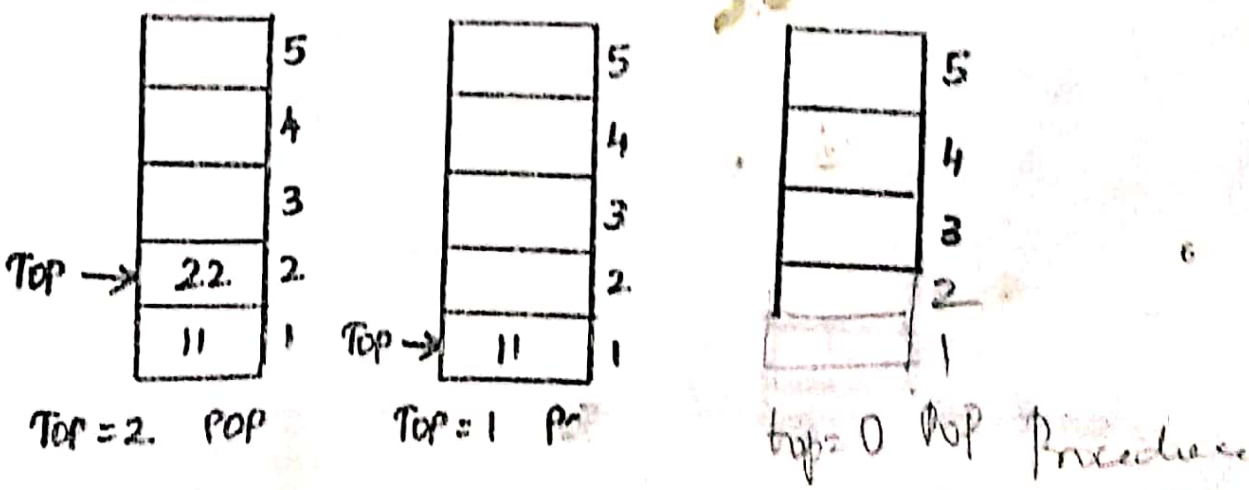
PUSH OPERATION: Let STACK [1:n] be an array representing stack and top be the pointer indicating the top element of the stack. Let top is initialized to 0. "item" is the element to be pushed into the stack and 'n' is the maximum capacity of the stack. STACK FULL is the procedure used to take remedial action for overflow of stack and print respective error message.)

```
Procedure PUSH (STACK,n,top,item)
If (top=n) then
Call STACK FULL;
else
{
top=top+1;
STACK[top]=item;
}
End PUSH
```



POP OPERATION: In case of POP, no element identity need be specified since by default the element occupying the top of stack position is deleted. (STACK EMPTY is the procedure used to do alternative measure to tackle underflow of stack by printing respective error message.

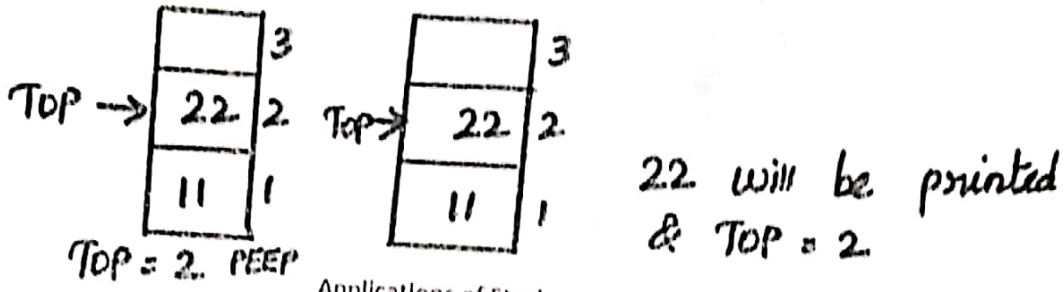
```
Procedure POP (STACK,top,item)
If (top=0) then
call STACK-EMPTY
else
{
item=STACK[top];
top=top-1;
}
End POP
```



Scanned by CamScanner

PEEP OPERATION: Peep is the operation used to display the top element of the stack without removing the element from the stack.

```
Procedure PEEP(STACK,top,item)
If (top=0) then
call STACK EMPTY
else
Print "The top element=" ,STACK[top]
End PEEP
```



Applications of Stack

Stacks have been used in various computer science applications like recursion, towers of Hanoi, syntax checking, checking of well formedness of parenthesis, conversion of infix to postfix expression, evaluation of postfix expression etc.

a) Infix to Postfix Expression (Notation): An expression can be expressed in three forms as Infix, prefix and postfix. They differ from each other in the position of operators. We already know the expression is the combination of operators and operands.

- If the operator is placed in between operands, it is called as Infix expression. Eg: a+b
- If the operator is placed in before operands, it is called as Prefix expression. Eg: + ab
- If the operator is placed in after operands, it is called as Postfix expression. Eg:ab+

To convert an infix to postfix notation, we need to find out the order of execution of the operators based on their hierarchy. The precedence of operators is

1. () Parenthesis
2. ^ Exponential operator
3. * / Multiplication, Division
4. + - Addition, Subtraction

Eg: Conversion of Infix notation A+ (B-C/D)*E to postfix notation

A + (B - C/D) * E

A + (B - CD/) * E

A + BCD/- * E

BCD/-E*

Scanned by CamScanner

Procedure INFIX-POSTFIX-CONV(E)
x=getnextchar(E);
while x≠"$" do
    Case x:
        operand: Print "x";
        ')': while (top element of stack≠"(") do
                Print top element of stack and Pop;
             End while
             Pop "(" from stack;
        Else: while ICP(x)<=ISP(top element of stack) do
                Print top element of stack and Pop;
              End while
              Push x onto stack;
    End case
    X=getnextchar(E);
End while
While stack is not empty do
    Print the top element of stack and pop;
End while
End INFIX-POSTFIX-CONV.

The above algorithm is governed by two priority factors-**ISP (In Stack Priority) and ICP (In Coming Priority)**. Apply ISP for the elements that are already residing in stack and ICP for the elements that are to be pushed into stack.

The following table will give the ISP and ICP values for operators and operands in the expression.

OPERATOR	ISP	ICP
)	NIL	NIL
^	3	4
*,/	2	2
+,-	1	1
(	0	4

NOTE: The infix expression is padded with a "$" to signal the end of input. The bottom of the stack is also with inserted with "$" with ISP($)=-1. Pop operators out of the workspace as long as the ICP of the incoming operator is <=to the ISP of the operator already existing in the stack.

Eg: Convert A*(B+C)-G into postfix.

Initialize stack as "$" and attach "$" at the end of the given expression as A*(B+C)-G $.

X	STACK	POSTFIX	REMARKS
A	$	A	Since A is an operand
*	$*	A	Since ICP(*)>ISP($)
(	$*(	A	Since ICP(()>ISP(*)
B	$*(	AB	Since B is an operand
+	$*(+	AB	Since ICP(+)>ISP(()
C	$*(+	ABC	Since C is an operand
)	$*	ABC+	Pop till "(" comes

Scanned by CamScanner

-	$-	ABC+*	Since ICP(-)<ISP(*) pop "*" and push "-"
G	$-	ABC+*G	Since G is an operand
$	$	ABC+*G-	Since $ is encountered at expression, pop all elements from stack until "$"

Therefore the equivalent postfix notation is ABC+*G-

b) Evaluation of Postfix expression: A postfix expression has the operator immediately following its operands. When an operator is encountered reading the input from left to right, its corresponding operators are those that precede them. This matches the LIFO property of stack. Hence a stack is useful to evaluate a postfix expression.

```
Procedure POSTFIX(E)
Clear the stack
symb=getnextchar(E)
While not end of expression E
        If symb is an operand
                Push the symb into the stack
        Else
        If symb is an operator
        {
                Pop the top element and store it as operand 2;
                Pop the top element and store it as operand 1;
                Compute Result=operand1 operator operand2;
                Push the Result into the stack;
        }
        symb=getnextchar(E)
End while
End POSTFIX
```

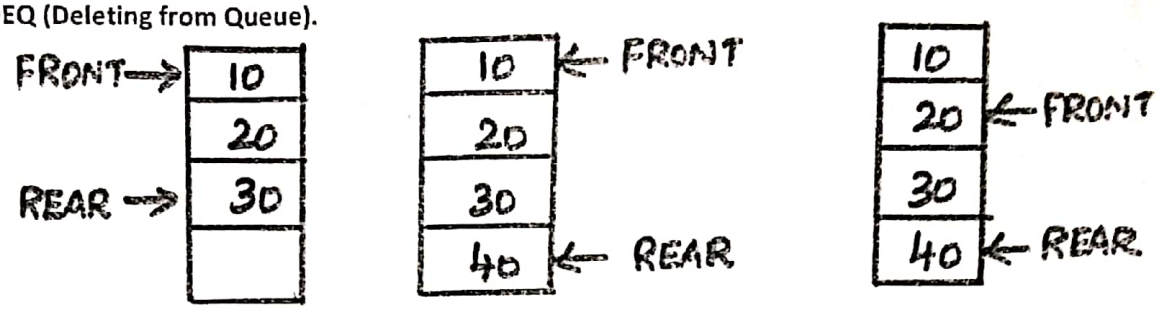
EG: Evaluate ABC+*G- where A=2, B=5,C=12,D=18,G=10.

SYMB	OPERAND 1	OPERAND 2	RESULT	STACK
A				A
B				AB
C				ABC
+	B	C	B+C=17	A 17
*	A	17	A*17=34	34
G				34 G
-	34	G	34-10=24	24

Hence the result of the above expression is 24.

Scanned by CamScanner

A queue is an ordered group of elements in which elements are added at one end called as rear end and elements are removed from the other end called as front end. Queue is often used to simulate the real world situations. Eg: Line of students in a fees counter, Line of customers in Bank. A person who stands first leaves the queue first and if any wants to add in the queue, they have to insert themselves at the last. So queue is called as FIFO (First In First Out) data structure. The process of adding a new element is called as **ENQ (Entering into Queue)** while the process of deleting an existing element is called as **DEQ (Deleting from Queue).**



Note: When we perform a DEQ operation, we must ensure that the queue is non-empty. If we try to delete an element from an empty queue, it will be an illegal attempt indicating overflow (QUEUE-FULL). Similarly while performing ENQ operation, we must ensure that the queue is not full. If we try to add an element to an already full queue, it will be illegal attempt indicating underflow (QUEUE-EMPTY).

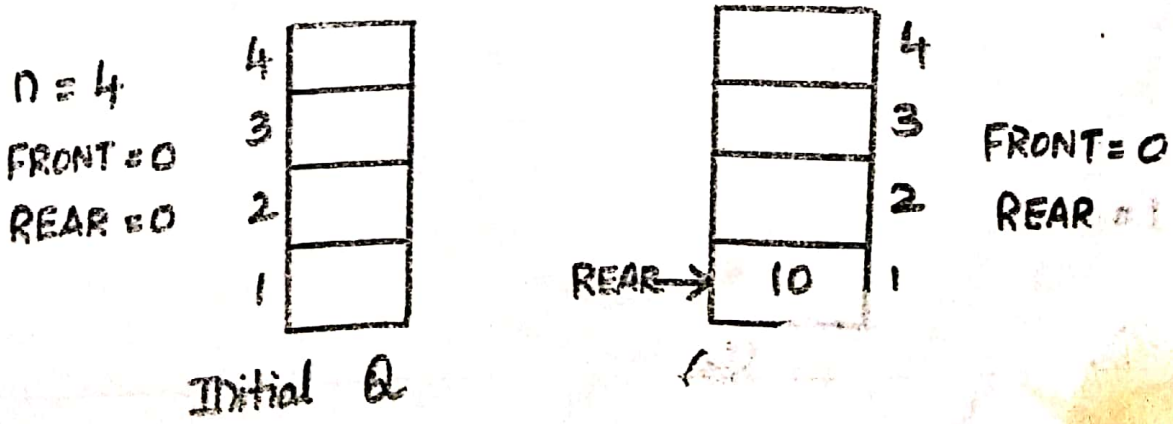
Primitive operations on Queue

Queue is also implemented by an array as Q[1:n] with a maximum capacity of n elements in the queue. Two variables front and rear are used to indicate the first and last positions in the queue respectively.

1.Insert Operation: Let item be the element inserted into the queue Q and the variables front and rear are initialized to zero.

```
Procedure ENQ(Q,n,item,REAR)
if (REAR=n) then
        Call QUEUE FULL;
else
{
        REAR=REAR+1;
        Q[REAR]=item;
}
End ENQ
```

The addition of every new element into the queue increments the REAR variable and before every addition the queue is checked for its full condition in order to avoid overflow of elements.

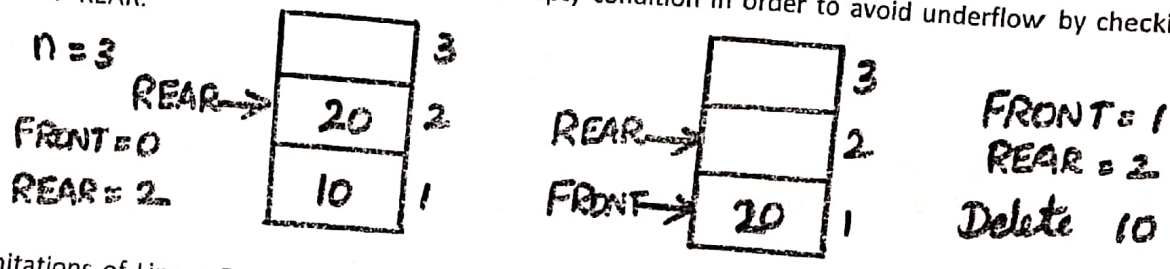


Scanned by CamScanner

2.Delete Operation: Though a deletion operation automatically deletes the front element of the queue, the variable item is used as an output variable to store and display the value deleted from queue.

```
Procedure DEQ(Q,FRONT,REAR,item)
if (FRONT=REAR) the
        Call QUEUE-EMPTY;
else
{
        FRONT=FRONT+1;
        Item=Q[FRONT];
}
End DEQ
```

The deletion of every new element into the queue increments the FRONT variable and before every deletion the queue is checked for its empty condition in order to avoid underflow by checking FRONT=REAR.



Limitations of Linear Queue: There is a gross limitation of a linear queue since QUEUE-FULL condition checks only REAR=n condition and it does not check whether the queue is physically full or not. But in real time situations, while deletion at rear takes place, automatically the remaining elements in the queue will be moved forward towards front leaving no gap and thereby providing new rooms for new insertions. But moving the data forward will be computationally expensive and may render the process very complex and inefficient.

Eg:Q[1:3]

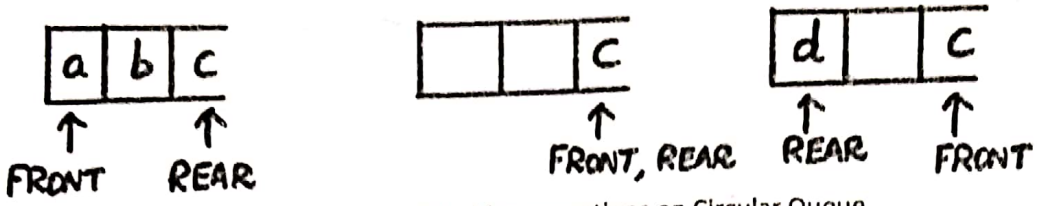
Operation	Queue	Values of FRONT and REAR	Remarks
------	Empty queue	FRONT=REAR=0	-----
Insert a	a	FRONT=0 REAR=1	Inserted successfully
Insert b	a b	FRONT=0 REAR=2	Inserted successfully
Insert c	a b c	FRONT=0 REAR=3	Inserted successfully
Insert d	a b c	FRONT=0 REAR=3	Insertion failure
Delete	b c	FRONT=1 REAR=3	Deleted successfully
Delete	c	FRONT=2 REAR=3	Deleted successfully
Insert d	c	FRONT=2 REAR=3	Insertion failure though there are free space at queue

Scanned by CamScanner

Circular Queue

In order to overcome the limitations of linear queue, we are going on for circular queue. As the name implies it is circular in nature. Here the FRONT and REAR variables are displayed in the circular movement rather than from left to right. A circular queue is also implemented as array and is conceptually viewed as a circular one. The overflow occurs only when all the locations are filled. When the rear pointer reaches the end, it is made to check the first location.

Eg: Let CIRC-Q be a circular queue with a capacity of three elements.

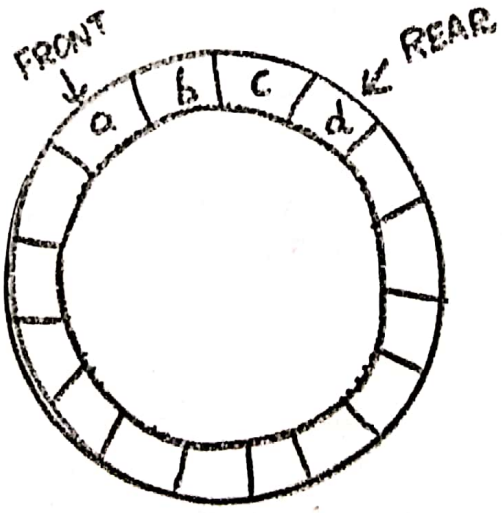


Primitive operations on Circular Queue

The circular movement of FRONT and REAR variables is implemented using mod function which is cyclical in nature. Also the array data structure CQ to implement the queue is declared to be CQ[0:n-1] to facilitate the circular operation of FRONT and REAR variables. As in linear queues, FRONT points to a position which is one less than the actual front of the circular queue. Both FRONT and REAR are initialized to 0. Note that (n-1) is the actual physical capacity of the queue in spite of the declaration 0 to n-1. The following are the procedure to add and delete element from a circular queue respectively.

```
Procedure:INSERT-CIRCQ(CQ,FRONT,REAR,n,item)
REAR=(REAR+1)mod n;
IF(FRONT=REAR) then
      {
            Call CIRCQ-FULL;
            Retract REAR Pointer to the old value;
      }
CQ[REAR] = item;
END INSERT - CIRCQ
```

```
Procedure DELETE-CIRCQ (CQ, FRONT, REAR, n, item)
IF FRONT = REAR) then
      Call CIRCQ - EMPTY;
FRONT = (FRONT+1) mod n;
item = CQ [FRONT];
End (INSERT) - CIRCQ
    DELETE
```



Scanned by CamScanner

Eg: CQ[0...3] with n=4

Operation	Circular Queue	Values of FRONT and REAR	Remarks
------	Empty queue	FRONT=REAR=0	-----
Insert a	[ ][a][ ][ ]	FRONT=0 REAR=1	Inserted successfully
Insert b	[ ][a][b][ ]	FRONT=0 REAR=2	Inserted successfully
Insert c	[ ][a][b][c]	FRONT=0 REAR=3	Inserted successfully
Insert d	[ ][a][b][c]	FRONT=0 REAR=3	Insertion failure
Delete	[ ][ ][b][c]	FRONT=1 REAR=3	Deleted successfully
Delete	[ ][ ][ ][c]	FRONT=2 REAR=3	Deleted successfully
Insert d	[d][ ][ ][c]	FRONT=2 REAR=0	Inserted successfully
Insert e	[d][e][ ][c]	FRONT=2 REAR=1	Inserted successfully

Priority Queues

A priority queue is a queue in which insertion or deletion of items from any position in the queue is done based on some property called as priority of task. When elements in the queue have the same priority, then the priority queue will behave as an ordinary queue following the principle of FIFO.

Eg: Set of patients waiting for the doctor.

Representation of priority queue

Let x be the element to be inserted/deleted with priority y. The element may be represented as xy.

$a^2 b^1 c^1$

$d^4 a^2 b^1 c^1$

$a^2 b^1 c^1$

There are two types of implementation of priority queue.

1. Open as many queues as per priority. A low priority queue will be operated if and only if all high priority predecessors are empty. Here the space required to build the queue is large. The time required to add a new element into the queue is O(1). But deletion requires more time since more time is taken for checking all the queues.

Scanned by CamScanner

2. Open only 1 queue for all the elements to be manipulated. . Here the space required to build the queue is small. The time required to add a new element into the queue is O(nlogn). But deletion requires only less time since the first element at the head of the queue is deleted easily and so the time complexity is O(1).

Dequeue

A dequeue (Double Ended Queue) is a linear list of elements in which all insertions and deletions are made at the either end of the list. A Dequeue is therefore more general than stack or linear queue and is a sort of **FLIFLO (First In Last In First Out Last Out).**

There are two variants of Dequeue as,

Dequeue
- Input-Restricted Dequeue
- Output Restricted Dequeue

Input-Restricted Dequeue: Here the input/insertion is restricted to the rear end and the deletion has the option of either ends (ie) the element at the front or the rear end of the queue can be deleted.

Output-Restricted Dequeue: Here the output/deletion is restricted to the front end and the insertion has the option of either ends (ie) the element at the front or the rear end of the queue can be deleted.

Dequeue is manipulated by two variables LEFT, RIGHT and their behavior during insertions and deletions are given below. During deletion, if LEFT=RIGHT, it implies that there is only one element in Dequeue and so after deletion, Dequeue becomes empty (ie) LEFT=RIGHT=NIL.

	LEFT	RIGHT
INSERTION	Decremented by 1 mod n	Incremented by 1 mod n
DELETION	Incremented by 1 mod n	Decremented by 1 mod n

Eg: DQ [1:6] be the Dequeue as a circular array.

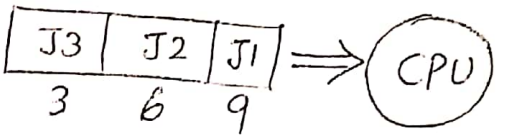
OPERATION	DEQUEUE (1 2 3 4 5 6)	REMARKS
Initial dequeue	R T S	LEFT=3 RIGHT=5
Insert X at left and Y at right	X R T S Y	LEFT=2 RIGHT=6
Delete twice from right	X R T	LEFT=2 RIGHT=4
Insert G,Q,M at left	G X R T M Q	LEFT=5 RIGHT=4
Insert J at right	G X R T M Q	Dequeue is full and so no insertion
Delete twice from left	G X R T	LEFT=1 RIGHT=4

Scanned by CamScanner

Application of queue

The queue is used in the scheduling of jobs by a processor in a time sharing system. A CPU (processor) associated with memory resources are to be shared by "n" number of users. The sharing of processor and memory resources is done by allotting a definite time slice of the processor's attention on the users and in a round-robin fashion. In such a system, the users are unaware of the presence of other users and are led to believe that their job receives the undivided attention of the CPU. However, to keep track of the jobs initiated by the users, the processor relies on a queue data structure recording the active user.

Eg:-



Consider the above setup of jobs whose burst times are 9, 6, 3 respectively. The time needed by the job to spent at CPU is called as burst time. The time for which CPU should at a particular job is called as time slice / time quanta. The following is the working procedure of round-robin CPU-scheduling algorithm.

Jobs	T1	T2	T3	T4
J1	9	6	3	0
J2	6	3	0	0
J3	3	0	0	0

↓ J3 quits ↓ J2 quits ↓ J1 quits

CPU spends a time quanta of 3 at each job equally.

Scanned by CamScanner

LINKED LISTS

The size of the array is declared and finalized at the compilation itself. We anticipate a higher bound and preserve the memory-but most of the time, we don't use all. This problem of memory under-utilization can be solved if we allocated memory only when there is a need. This is called as dynamic memory allocation. Here the elements are stored in non-contiguous fashion. So to maintain the order of elements, we need to know the address of each element. So a node will store the address of its successive node and the last node will have NULL. These types of data structure is called linked list. They are examples of linear data structures as they permit linear access of element one after other; they are also dynamic data structures as their size can grow or shrink during execution.

Limitations of arrays

1. Inefficient implementation of insertion and deletion.
2. Inefficient use of storage memory.

"Linked list is the natural data abstraction that arises in problems where a data structure maintains a collection of elements without knowing in advance about the number of elements"

Operations on Linked List

The primitive operations performed on the linked list are as follows

1. **Creation operation** is used to create a linked list. Once a linked list is created with one node, insertion operation can be used to add more elements in a node.
2. **Insertion operation** is used to insert a new node at any specified location in the linked list. A new node may be inserted.
   (a) At the beginning of the linked list
   (b) At the end of the linked list
   (c) At any specified position in between in a linked list
3. **Deletion operation** is used to delete an item (or node) from the linked list. A node may be deleted from the
   (a) Beginning of a linked list
   (b) End of a linked list
   (c) Specified location of the linked list
4. **Traversing** is the process of going through all the nodes from one end to another end of a linked list. In a singly linked list we can visit from left to right, forward traversing, nodes only. But in doubly linked list forward and backward traversing is possible.
5. **Searching** is the process of picking up desired element from the list.
6. **Concatenation** is the process of appending the second list to the end of the first list. Consider a list A having n nodes and B with m nodes. Then the operation concatenation will

Scanned by CamScanner

Place the 1st node of B in the (n+1)th node in A. After concatenation A will contain (n+m) nodes.

Advantages of Linked list

1. Linked list are dynamic data structure. That is, they can grow or shrink during the execution of a program.
2. Efficient memory utilization: In linked list (or dynamic) representation, memory is not pre-allocated. Memory is allocated whenever it is required and it is de-allocated (or removed) when it is not needed.
3. Insertion and deletion are easier and efficient. Linked list provides flexibility in inserting a data item at a specified position and deletion of a data item from the given position.
4. Many complex applications can be easily carried out with linked list.

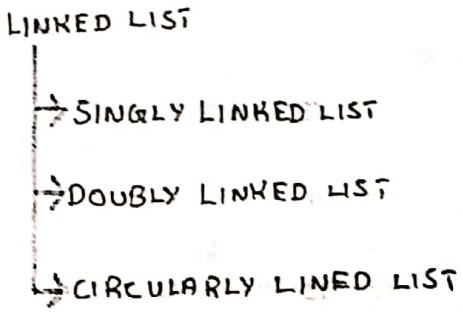
Limitations of Linked List

1. More memory: More memory space is needed even for storing a single data.
2. Access to an arbitrary data item is little bit cumbersome and also time consuming because it purely supports sequential traversal.

Mechanisms needed for linked lists

1. A mechanism to frame chunks of memory into nodes with required number of data items and fields.
2. Determine which nodes are free and which have been allotted for use.
3. Obtain nodes from free storage area [GETNODE(X)].
4. Return or dispose of nodes again to free pool area [RETURN(X)].

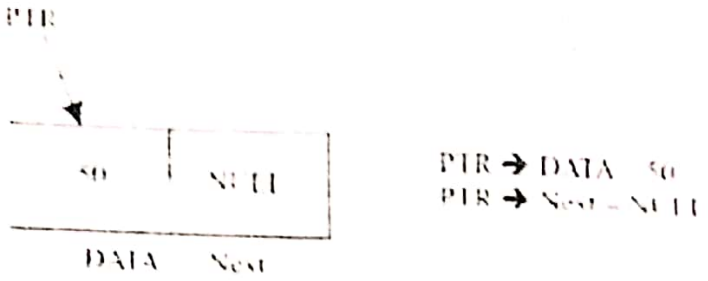
Hence dynamic allocation of memory solves the problem of static allocation of arrays. The classification of linked list done on the basis of the arrangement of nodes is as follows:

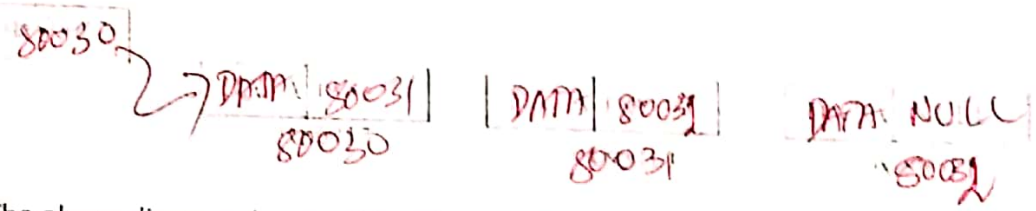


Scanned by CamScanner

SINGLY LINKED LIST

A linked list is a linear collection of specially designed data elements, called nodes linked to another by means of pointers. Each node is divided into two parts: the first part contains information of the element and the second part contains the address of the next node in the linked list. Address part of the node is also called **linked or next field**. As the name implies only one pointer arises from each node.





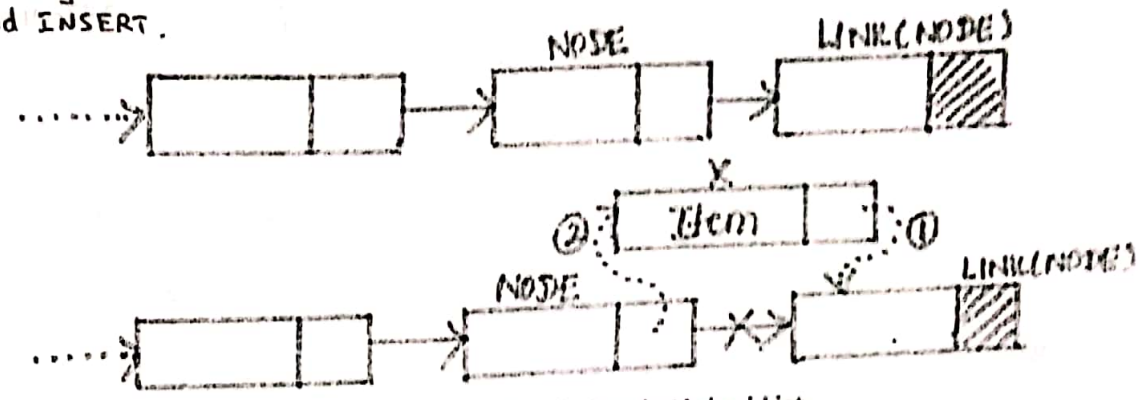
The above diagram shows a linked list with 3 nodes. Each node is divided into two parts. The left part of each node contains the data items and the right part represents the address of the next node; there is an arrow drawn from it to the next node. The next pointer of the last node contains a special value, called the NULL pointer, which does not point to any address of the node (ie) NULL pointer indicates the end of the linked list. START pointer will hold the address of the 1st node in the list START =NULL if there is no list (i.e.; NULL list or empty list). The data field can be integer, character, a string or even a large record.

Insert Operation in Singly Linked List

The below procedure is used to insert a new node ITEM to the right of an existing node NODE in singly linked list whose first node is pointed by START pointer. It is mandatory to tell the position after or before the insertion has to be performed.

Scanned by CamScanner

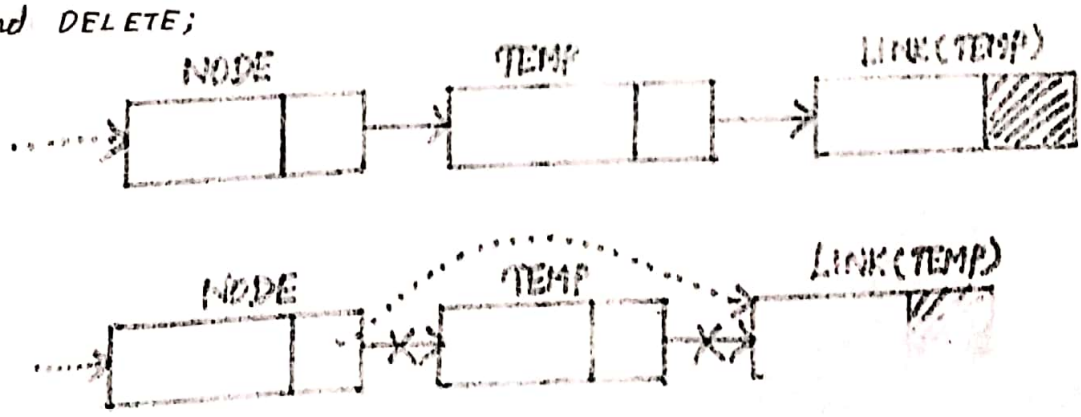
```
Procedure INSERT(START,NODE, ITEM)
call GETNODE(X);
DATA(X) = ITEM;
If (START = NULL) then
    {   LINK(X) = NULL;
        START = X;
    }
else
    {
        LINK(X) = LINK(NODE);
        LINK(NODE) = X;
    }
End INSERT.
```



Delete Operation in Singly Linked List

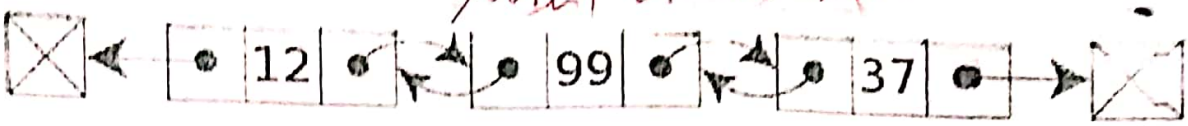
The below procedure is used to delete a node which is right of NODE in the existing singly linked list. It is possible to delete a node if and only if its previous node is known.

```
Procedure: DELETE (START, NODE)
   IF (START = NULL) then
      Call ABONDON - DELETE
   else
      {
          TEMP = LINK(NODE);
          LINK(NODE) = LINK(TEMP);
          Call RETURN (TEMP);
      }
End DELETE;
```



Scanned by CamScanner

DOUBLY LINKED LIST



A doubly-linked list is a linked data structure consists of a set of sequentially linked records called nodes. As the name implies, each node contains two fields, called **links (LLINK, RLINK)**, which are references to the previous and to the next node in the sequence of nodes. The beginning and ending nodes' previous and next links, respectively, point to some kind of terminator, typically a sentinel node or null, to facilitate traversal of the list. It can be conceptualized as two singly linked lists formed from the same data items, but in opposite sequential orders. The two node links allow traversal of the list in either direction.

Advantages:

1. The availability of 2 links LLINK and RLINK permit forward and backward movement during the processing of list.
2. The deletion of a node X from the list only for the value X to be known.

Disadvantages:

1. Obviously, we are in need of more space to construct doubly linked list as a minimum of 3 parts are needed to construct a node.

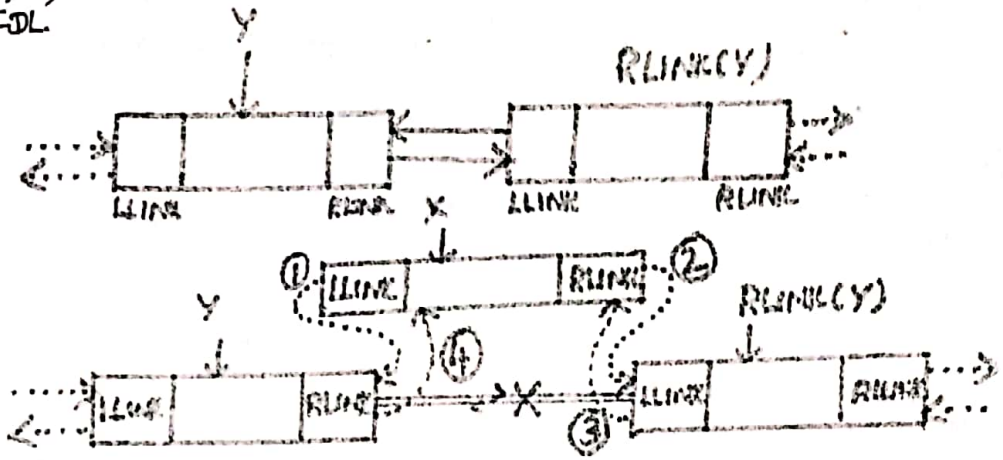
While adding or removing a node in a doubly-linked list requires changing more links than the same operations on a singly linked list, the operations are simpler and potentially more efficient (for nodes other than first nodes) because there is no need to keep track of the previous node during traversal or no need to traverse the list to find the previous node, so that its link can be modified. The availability of two links LLINK and RLINK permit forward and backward movement during the processing of list.

Insert Operation in Doubly Linked List

The below procedure is used to insert a new node X whose data part is ITEM to the right of an existing node Y in doubly linked list.

Scanned by CamScanner

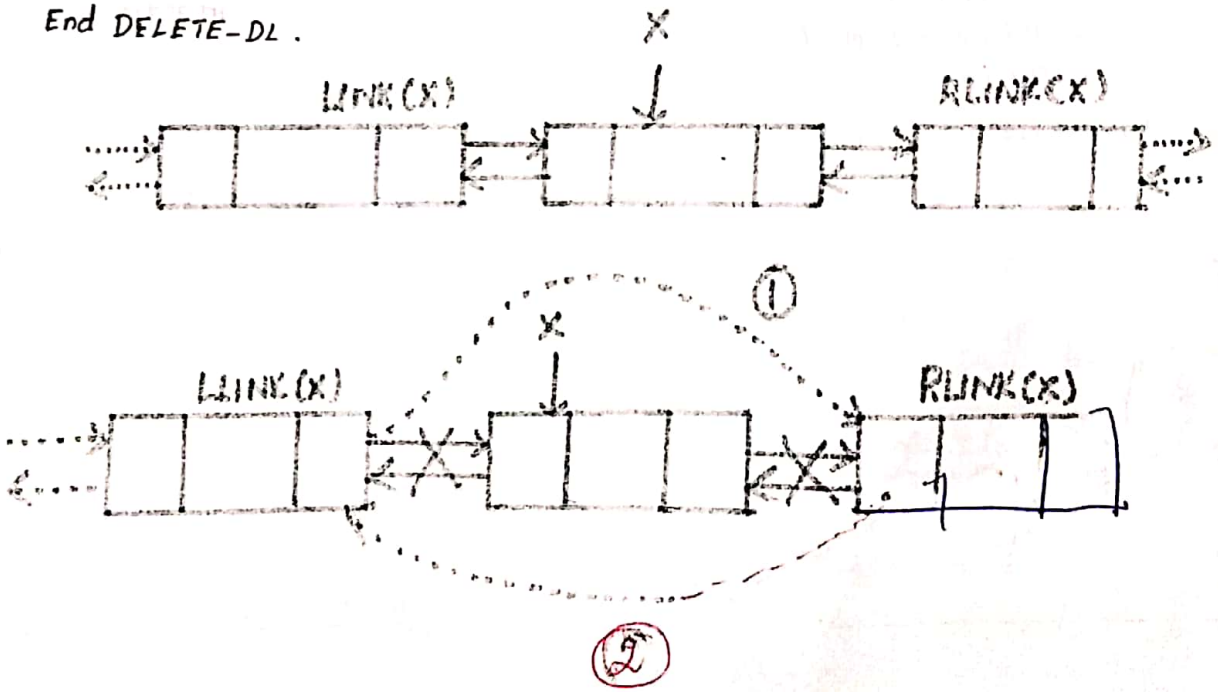
```
Procedure INSERT-DL(X,Y,ITEM)
    Call GETNODE(X);
    DATA(X)=ITEM;
    LLINK(X)=Y;
    RLINK(X)=RLINK(Y);
    LLINK(RLINK(Y))=X;
    RLINK(Y)=X;
End INSERT-DL.
```



Delete Operation in Doubly Linked List

The below procedure is used to delete a node from a doubly linked list P.

```
Procedure DELETE-DL (P, X)
    IF (P=NULL) then
      Call ABONDON-DELETE;
    else
      {
        RLINK(LLINK(X)) = RLINK(X);
        LLINK(RLINK(X)) = LLINK(X);
        Call RETURN(X);
      }
End DELETE-DL.
```



Scanned by CamScanner

CIRCULARLY LINKED LIST

A variant of the singly linked list is circular linked list in which the last node's link part points to the first node.

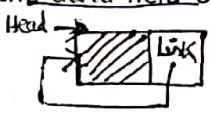
Advantages:

- Accessibility of a node- one can access any node from any given node due to the circular movement permitted by links.
- Deletion of a node – one need not specify the predecessor.
- Relative efficiency in the implementation of list based operations like merging of 2 lists, splitting of a list into two, erasing a list etc.

Disadvantages:

- During processing one has to make sure that one does not get into an infinite loop owing to the circular nature of the queue.

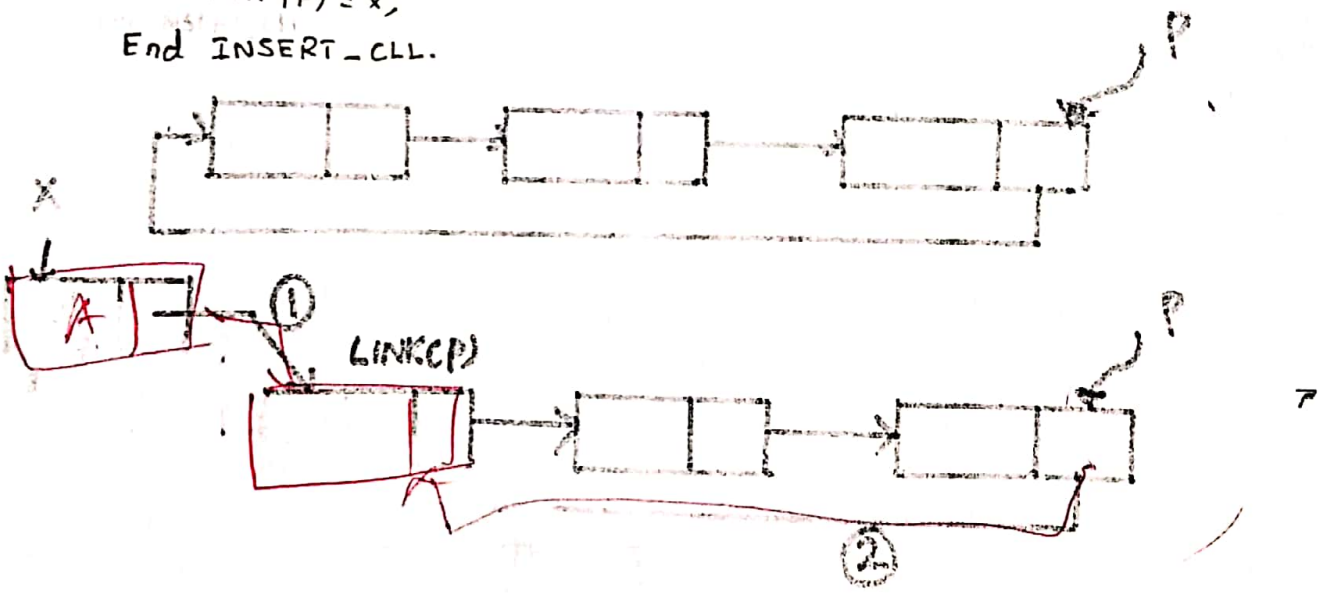
A solution to the infinite loop in circular linked list is having a special node called header nodes. So the list may be called a Headed circularly linked list or circular linked list with head node. The list can never be empty and will have a empty header node. If LINK(HEAD)=HEAD, it implies that the circular list is empty. Usually, the data field of the header node is empty, represented by a shaded portion in diagram.



Insert Operation in Circularly Linked List

The following algorithm will insert a new node X as the left-most element in a list represented by P.

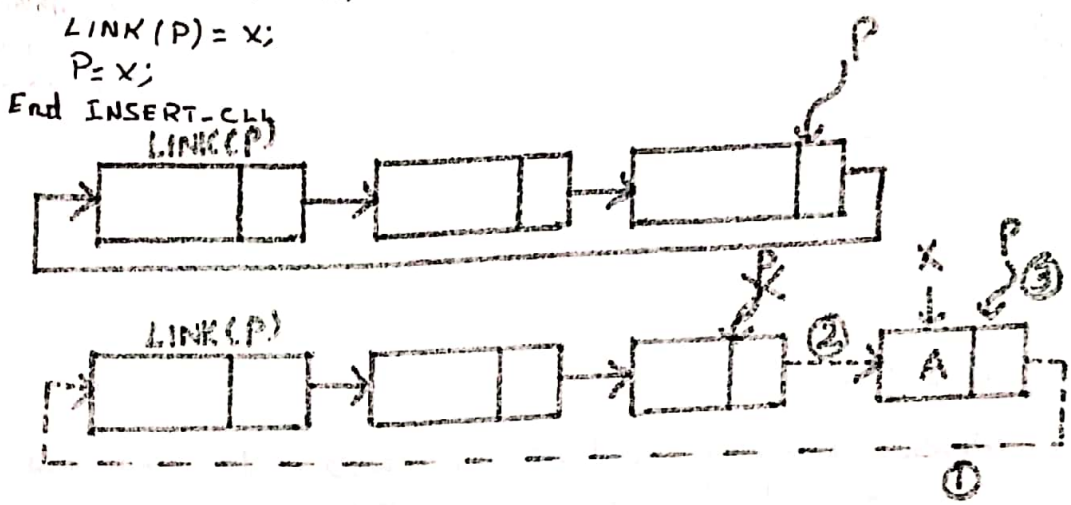
```
Procedure INSERT_CLL (P,x,A)
    Call GETNODE (x);
    DATA (x) = A;
    LINK (x) = LINK (P);
    LINK (P) = x;
End INSERT_CLL.
```



Scanned by CamScanner

The following algorithm will insert a new node X as the right-most element in a list represented by P.

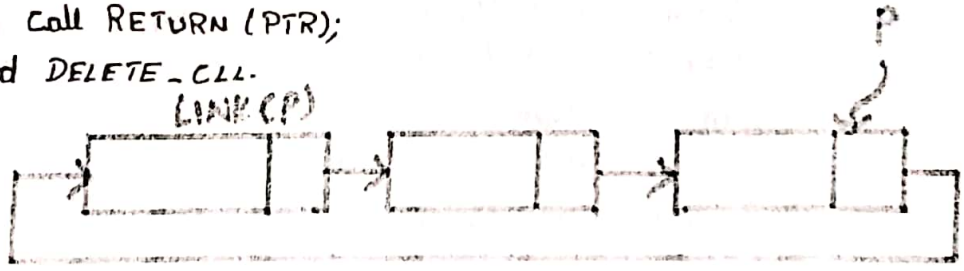
```
Procedure INSERT CLL (P, x, A)
    Call GETNODE (x);
    DATA (x) = A;
    LINK (x) = LINK (P);
    LINK (P) = x;
    P = x;
End INSERT_CLL
```

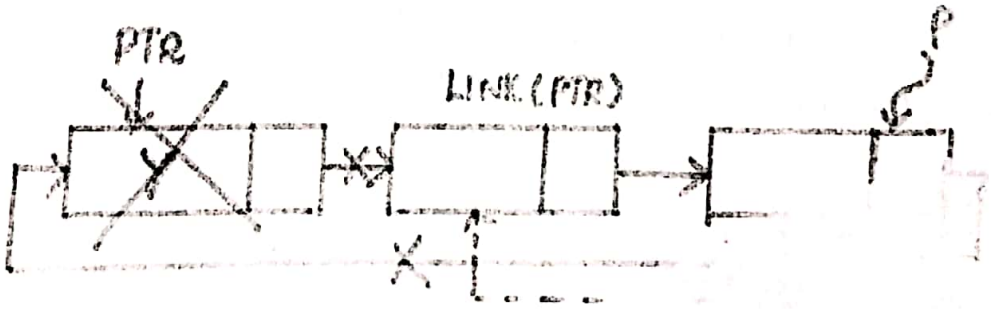


Delete Operation in Circularly Linked List

The following procedure will delete the left-most node from a circular linked list pointed by P.

```
Procedure DELETE CLL(P)
    PTR = LINK (P);
    Y = DATA (PTR);
    LINK (P) = LINK (PTR);
    Call RETURN (PTR);
end DELETE_CLL.
```

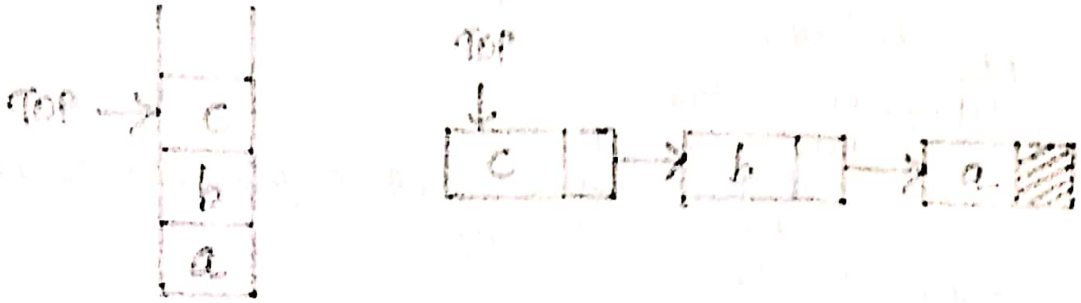




Scanned by CamScanner

LINKED STACK

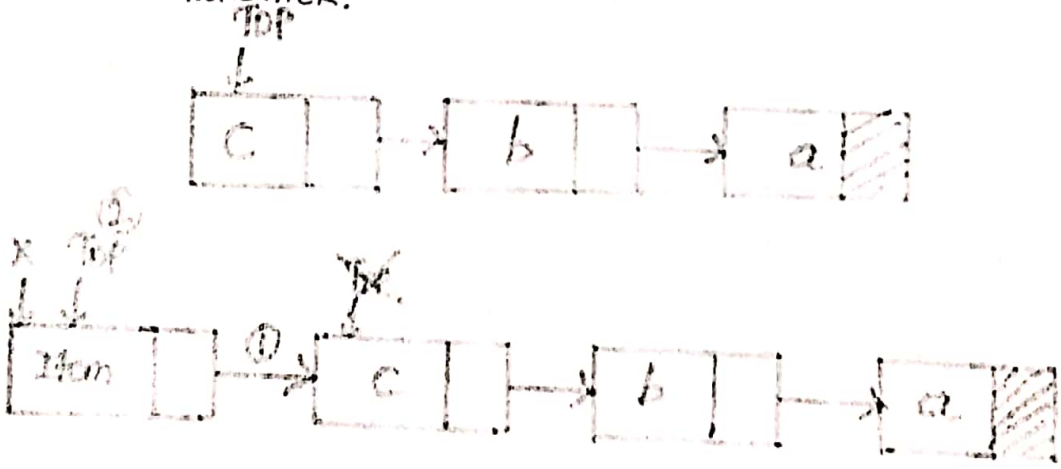
Linked Stack is a linear list of elements commonly implemented as a singly linked list whose start pointer performs the role of the top pointer.



OPERATIONS ON LINKED STACK

To push an element into the linked stack, we insert the node representing the element as the first element. The top pointer which points to the first element in the singly linked list is automatically updated to point to the new element.

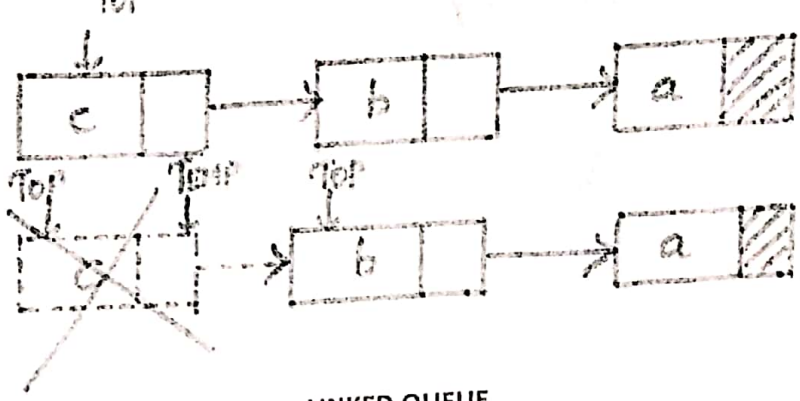
```
Procedure PUSH-LINKSTACK (TOP, ITEM)
    Call GETNODE (x);
    DATA (x) = ITEM;
    LINK (x) = TOP;
    TOP = x;
End PUSH-LINKSTACK.
```



The following procedure is used to delete an element at the TOP pointer from a linked stack.

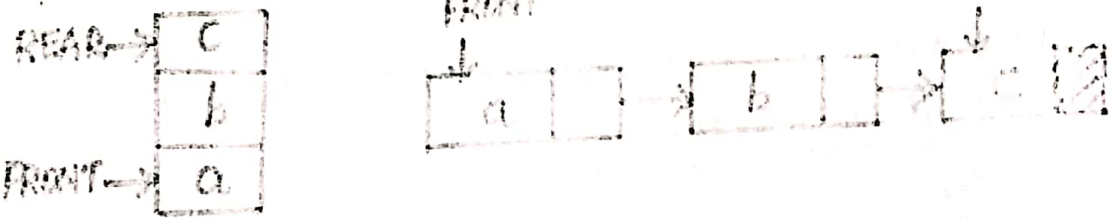
Scanned by CamScanner

```
Procedure POP-LINKSTACK ( TOP, ITEM)
    IF (TOP = 0) then
       Call LINKEDSTACK-EMPTY;
    else
       {
        TEMP = TOP;
        ITEM = DATA (TEMP) or DATA (TOP);
        TOP = LINK (TOP)
       }
Call RETURN(TEMP);
End POP-LINKSTACK
```



LINKED QUEUE

Linked Queue is a linear list of elements commonly implemented as a singly linked list with two pointers – FRONT and REAR. FRONT pointer points the start pointer of the singly linked list while REAR pointer points the last node of the list.

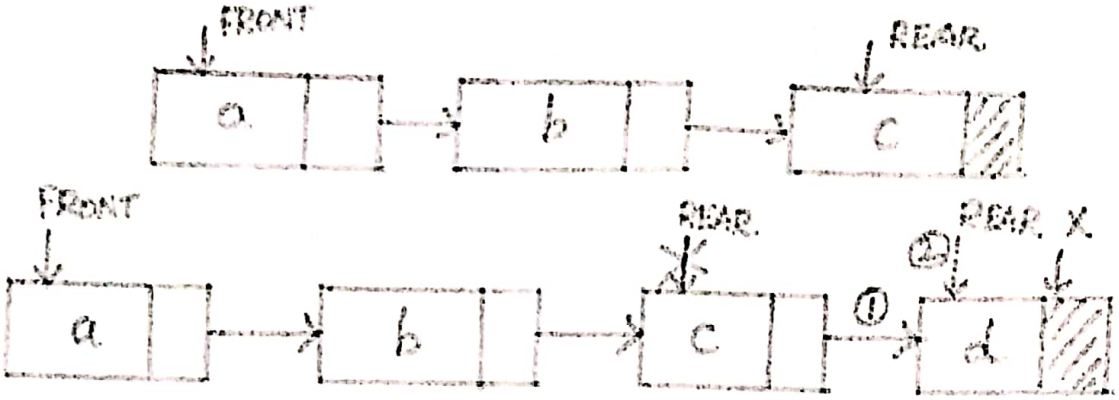


OPERATIONS ON LINKED QUEUE

To insert an element into the queue, insert the new node as the last node and reset REAR pointer to the new node.

Scanned by CamScanner

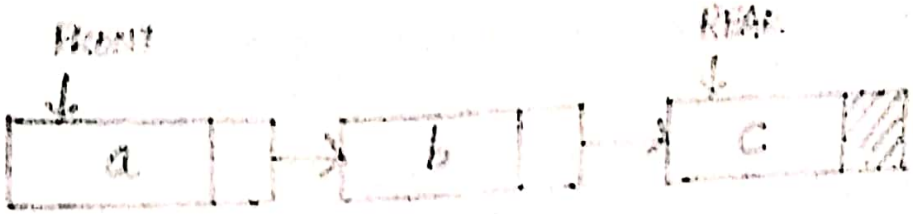
```
Procedure INSERT-LINKQUEUE(FRONT, REAR, ITEM)
    Call GETNODE (x);
    DATA(x) = ITEM;
    LINK(x) = NULL;
    IF (FRONT=0) then
        FRONT = REAR = x ; (insert as first element)
    else
      {
        LINK (REAR) = x;
        REAR = x;
      }
End INSERT_LINKQUEUE.
```

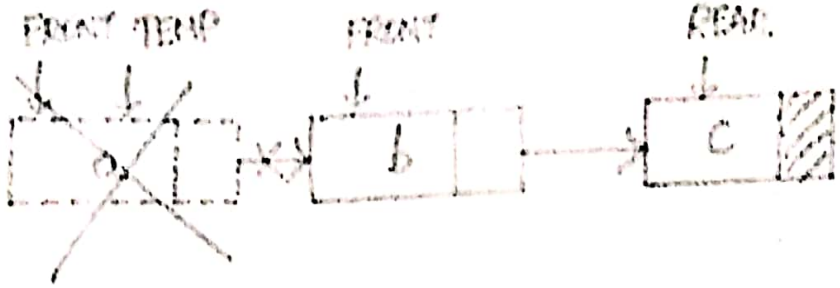


To delete an element from queue, we remove the first node of the list pointer by FRONT and it resets to the next node of the list.

```
Procedure DELETE-LINKQUEUE (FRONT, REAR, ITEM)
    IF (FRONT = 0) then
      Call LINKQUEUE_EMPTY;
    else
      {
        TEMP = FRONT;
        ITEM = DATA(TEMP);
        FRONT = LINK(TEMP);
      }
    Call Return (TEMP);
End DELETE_LINKQUEUE;
```


Scanned by CamScanner





Scanned by CamScanner

APPLICATIONS OF LINKED LISTS

Addition of polynomials: The objective of this application is to perform a symbolic addition of two polynomials p1 and p2. Make use of a singly linked list to represent each polynomial. To add the two polynomials, we have the nodes arrange in the decreasing order of the exponents of the variable X.

Structure of a node:

COEFFICIENT	POWER	NEXT
Coeff	Power	Next

Let p1=2x6+x3+5x+4 and p2=7x6+8x5-9x3+10x2+14. The objective is to create a new list of nodes represent the sum of p1+p2=9x6+8x5-8x3+10x2+5x+18. This is achieved by adding the coefficients of nodes of like powers of variable x in p1 and p2 into a new node at p1+p2. If any one of the lists during p1+p2 has exhausted before the other, then the node in the non-empty list will be simply appended at the end of list p1+p2.

```
Procedure POLYADD (P1,P2)
1. Initialize temp Pointers temp1 and temp2 as temp1=P1 and temp2 = P2.
2. Initialize Result to NULL as RESULT = NULL.
3. While (temp1 ≠ NULL and temp2 ≠ NULL)
        A1 = Power (temp1)
        A2 = Power (temp2)
        B1 = Coeff (temp1)
        B2 = Coeff (temp2)
        if (A1 = A2) then
            if (B1+B2 ≠ 0) then
                Result = addlast (Result, B1+B2, A1)
                temp1 = next (temp1)
                temp2 = next (temp2)
        else if (A1 > A2) then
                Result = addlast (Result, B1, A1)
                temp1 = next (temp1)
        else
                Result = addlast (Result, B2, A2)
                temp2 = next (temp2)
end while.
```


Scanned by CamScanner

```
4. if (temp1 ≠ NULL) then
        While (temp1 ≠ NULL)
            Result = addlast ((Result, Coeff (temp1), Power (temp1))
            temp1 = next (temp1)
        End while
    Else
    if (temp2 ≠ NULL) then
        while (temp2 ≠ NULL)
            Result = addlast ((Result, Coeff (temp2), Power (temp2))
            temp2 = next (temp2)
        End while
    End if
Return (Result)
End POLYADD.
```

The two polynomials are expected to be arranged in descending order in terms of power of X. Compare the power of X in P1 and P2 (first term). If both are same, add the coefficient of the selected terms and create a new node in the list whose data part will be the sum obtained, power of X considered and the pointer to the next node. Then move the pointers to the second term at both P1 and P2. If the first term's power of X of P1 is greater first term's power of X of P2, create a new node whose data part will be coefficient of X at P1 and power of the term considered at P1. Move the pointer to the second term at P1 and not at P2. If the first term's power of X of P2 is greater first term's power of X of P1, create a new node whose data part will be coefficient of X at P2 and power of the term considered at P2. Move the pointer to the second term at P2 and not at P1. Likewise continue the operation of adding two polynomials. If one polynomial expires before the other, just move the left terms in the long polynomial to the result.

Above algorithm shows the addition of polynomial over one variable X In case of polynomials of 2 or 3 variables, the node structure will be:

COEFF	EXPX	EXPY	NEXT

COEFF	EXPX	EXPY	EXPZ	LINK

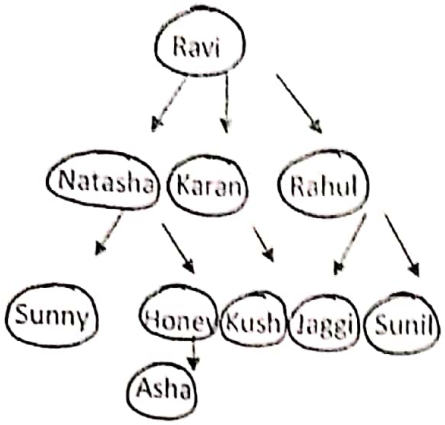
Drawbacks of the sequential representation of a stack/queue data structure are:

1. Finite capacity of a stack/queue.
2. Checking whether the stack/queue is full for every insert operation.

Scanned by CamScanner

TREES

Family Tree:



Basic Terminology:

1. Tree is a non-linear data structure. A tree consists of a collection of nodes which are connected by directed arcs.
2. A tree contains a unique first element called as root (i.e) node at the top of the tree. Eg: Ravi in a family tree
3. A node which points to other nodes is said to be the parent node of the node to which it is pointing and these nodes in the tree are called children of that node. Eg: Natasha is the parent for Sunny and Honey. Sunny and Honey are children of Natasha.
4. Root is the only node in the tree that has no parent. All nodes in the tree have exactly only one parent.
5. The nodes which have no child are designated as leaf. Eg: Sunny, Kush, Jaggi, Sunil, Asha.
6. The nodes which posses children are called interior nodes. Eg: Karan, Rahul. The nodes which have same parent are called Siblings. Eg: Jaggi, Sunil.
7. A node is an ancestor of another node if it is a parent of that node or parent of some other ancestor node of that node. The root is an ancestor of every node in that tree. Eg: Natasha is ancestor of Asha.
8. A node is said to be descendant of another node if it is the child of the node or the child of some other descendant of that node. Eg: Sunny, Honey and Asha are descendants of Natasha. Note: All the nodes in the tree are descendants of root.
9. A tree is a collection of one or more nodes with a distinct node called root, while the remaining are positioned as T1,T2... ....Tk,K>=0 each of which is called subtrees. The number of subtrees of a node is called degree. The degree of the maximum degree of node of the tree is the degree of a tree. Eg: Degree of Rahul=2.

Scanned by CamScanner

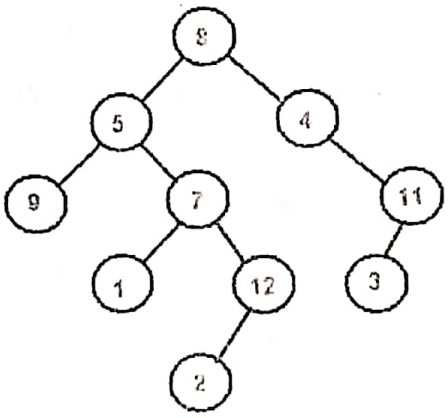
10. There is single unique path along arcs from the root to any particular node. The length of the longest path from root to any node is known as height or depth of tree. The root is at level 0 and the level of any other node in the tree is more than the level of its parent. Eg: Sunny is at level 2 , Asha at level 3.
11. Forest is a set of zero or more disjoint trees (removal of root will result in forests).

Advantages of trees

Trees are so useful and frequently used, because they have some very serious advantages:

- Trees reflect structural relationships in the data
- Trees are used to represent hierarchies
- Trees provide an efficient insertion and searching
- Trees are very flexible data, allowing to move subtrees around with minumum effort

BINARY TREES

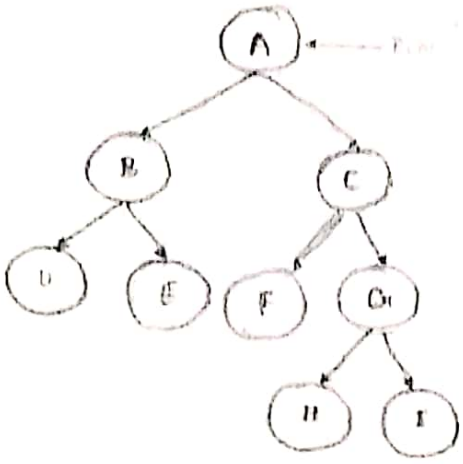


1. If we make a restriction that each node can have maximum 2 children, it is called binary tree.

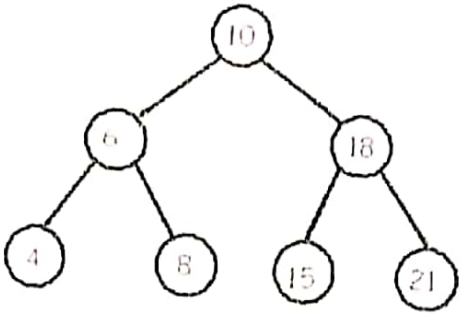
2. It is a tree which is either empty or consists of root node together with 2 binary trees called as left subtrees and right subtrees.

3. A binary tree is called a **strictly binary tree** if every non leaf node in the binary tree has non-empty left and right subtree (i.e) each node in a binary tree has either 0 or 2 children. A strictly binary tree with n leaves always contains exactly 2n-1 nodes.

Scanned by CamScanner



4. A **complete binary tree** can be defined as a binary tree whose non-leaf nodes have non empty left and right subtree and all leaves are at the same level. Eg: A complete binary tree of depth 3.



5. If a binary tree has only left child and totally the right child are null, it is called left skewed binary tree (Left child dominates).
6. If a binary tree has only a series of right child and totally the left child are null, it is called right skewed binary tree (Right child dominates).

LEFT-SKEWED BINARY TREE RIGHT-SKEWED BINARY TREE

Essential difference between a binary tree and tree:

1. Each element in a binary tree has exactly 2 subtrees (one or both of these subtrees may be empty).But each element in a tree can have any numbers of subtrees.

Scanned by CamScanner

2. Subtrees of each element in a binary tree are ordered and distinguished as left and right subtrees. The subtrees in a tree are unordered.

Properties of a binary tree:

1. Every binary tree with n elements (n>0) has exactly n-1 edges.

2. A Full binary tree of height h (h≥0) has at least h and at most 2^h-1 elements in it.

3. The height of a binary tree that contains n elements (n≥0) is at least $\log_2 (n+1)$.

4. Let i, 1≤ i≤ n, the number assigned to an element of a complete binary tree. The following are true:

4.1. If i=1, then this element is the root of the binary tree. If i>1, then the parent of this element has been assigned the number $\lfloor i/2 \rfloor$ (ie floor (i/2).

4.2. If 2i>n, then this element has no left child. Otherwise, its left child has been assigned the number 2i.

4.3. if 2i+1>n, then this element has no right child. Otherwise its right child has been assigned the number 2i+1.

BINARY TREE REPRESENTATION

Tree nodes may be implemented as array elements or as dynamically allocated variables. Let us consider each node of a tree contains info,left,right fields.

Array:

```
#define MAXNUM 100
struct nodetree
{int info,left,right;}
node[MAXNUM];
```

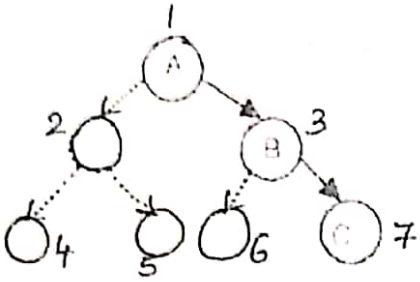
Linked list:

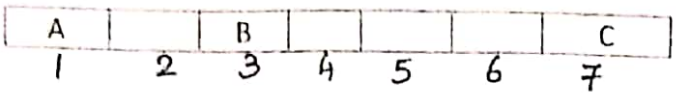
```
struct nodetree
{int info;
struct nodetree *leftchild,*rightchild;
}tree *p;
```

In an array representation, the binary tree is represented in an array by string-each element at the array position corresponding to the number assigned to it.

Eg:

Scanned by CamScanner



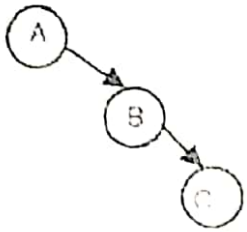


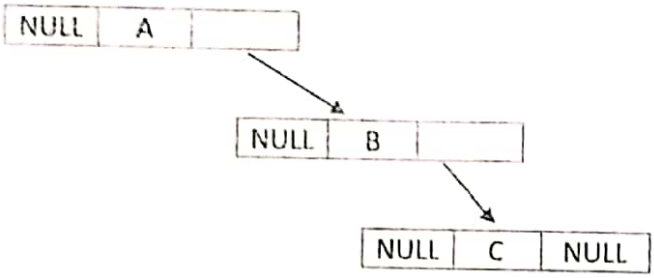
This representation is quite wasteful of space when many elements are missing. This representation is useful when the number of missing elements is small (i.e.) almost the tree is complete.

The most popular way to represent a binary tree is by using links or pointers. Let us call these pointer fields as leftchild and rightchild. Also each node has a field named info.

Observations:

1. If a binary tree has n nodes, then the number of pointers used is 2n.
2. The number of null pointers = n+1 for linked representation of n nodes.





Operations on Binary Trees:

1. Determine its height.
2. Determine the number of elements in it.
3. Make a copy.
4. Display the binary tree.

Scanned by CamScanner

5. Test whether 2 binary trees are identical.
6. Make the tree empty
7. Creation of binary tree
8. Insertion of a node into a binary tree
9. Deletion of a node from a binary tree
10. Searching of a node

BINARY TREE TRAVERSAL

All the operations above mentioned can be done only by traversing through the binary trees. Traversal means **"visiting all nodes at once"** (i.e.) processing the node of tree once and only once. Traversal can be used to find the information available in the binary tree in a linear order.

There are basically 3 standard traversal orders which differ by the position of the root visit. They are:

TRAVERSAL NAME	STYLE OF TRAVERAL		
Preorder	**Root**	**Left**	**Right**
Inorder	**Left**	**Root**	**Right**
Postorder	**Left**	**Right**	**Root**

If the root is visited before traversing its subtrees it is called preorder traversal. If the root is visited after traversing subtrees it is called postorder traversal. If the root is visited in between the trees, it is called inorder traversal.

There are also other tree traversals like,

TRAVERSAL NAME	STYLE OF TRAVERAL
Reverse Preorder	Right Left Root
Reverse Inorder	Right Root Left
Reverse Postorder	Root Right ~~Root~~ Left
Level Order	Visit by level from top to bottom and within a level from left to right

* In a postorder traversal, root occurs as last element
* In a preorder traversal, root occurs as first element.
* We can't locate root in an inorder traversal.

Scanned by CamScanner

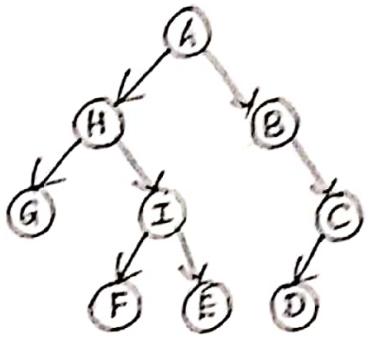
INORDER TRAVERSAL

```
Procedure INORDER_TRAVERSAL(NODE)
    IF NODE ≠ NIL THEN
    {
    Call INORDER_TRAVERSAL(LCHILD(NODE));
    Print DATA(NODE);
    Call INORDER_TRAVERSAL(RCHILD(NODE));
    }
End INORDER_TRAVERSAL
```

```
Pseudo Preorder(node)
if(node)
{
Cout<<node->data;
Preorder(node->left);
Preorder(node->right);
}
```

The traversal keeps moving left in the binary tree until one can't move no further, processes the node and moves to the right to continue its traversal again. In the absence of any node to the right, it retracts backwards by a node and continues the traversal. The above is the algorithm to perform Inorder traversal.

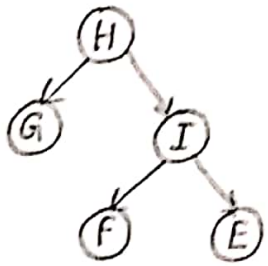
Eg:



Inorder = (left) Root (Right)

= (left) A (Right) ⟶ (1)

left:-



Since left is again a tree,

left = (left) Root (Right)

= (left) H (Right)

= G H [(left) Root (Right)]

= G H F I E ⟶ (2)

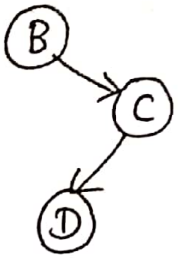
Sub (2) in (1):-

Hence (1) will become,

Inorder = G H F I E A (Right) ⟶ (1)

Scanned by CamScanner

Right:-



Since Right is again a binary tree,

Right = (Left) Root (Right)

= (Left) B (Right)

= (NULL) B (Right) (∵ there is no left subtree for B)

= B (Right)

= B [(Left) Root (Right)]

= BDC ⟶ ③ (∵ there is no right child for C)

Sub ③ in ①:-

Hence the inorder traversal for the given binary tree is, **GHFIEABDC**.

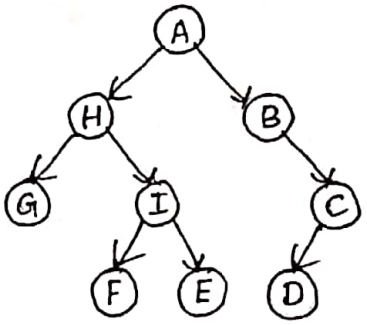
Scanned by CamScanner

POSTORDER TRAVERSAL

```
Procedure POSTORDER_TRAVERSAL (NODE)

IF NODE ≠ NIL THEN
{
Call POSTORDER_TRAVERSAL(LCHILD(NODE));
Call POSTORDER_TRAVERSAL (RCHILD (NODE));
Print DATA (NODE)
}
End POSTORDER_TRAVERSAL
```

The traversal proceeds by keeping to the left until it is no further possible, turns right to begin again or if there is not node to the right, processes the node and retraces its direction by one node to continue its traversal. The above is the algorithm to perform Postorder traversal.

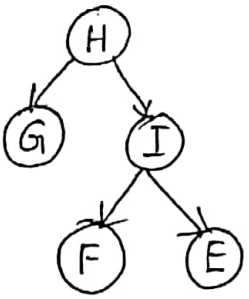
Eg:



Postorder traversal = (left) (Right) Root

= (left) (Right) A ⟶ (1)

Left:-



Since left child of A is a subtree, it has to undergo postorder traversal.

∴ left = (left) (Right) Root

= (left) (Right) H

= (G) (Right) H

Scanned by CamScanner

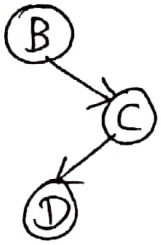
$= G\ [(Left)\ (Right)\ Root]\ H$

$= GFEIH \rightarrow (2)$

Sub (2) in (1):-

Here Postorder $= GFEIH\ (Right)\ A \rightarrow (1)$

Right:-



Since Right child of A is a subtree, traverse it by postorder traversal.

$Right = (Left)\ (Right)\ Root$

$= (Left)\ (Right)\ B$

$= (NULL)\ [(Left)\ (Right)\ (Root)]\ B$

$= D\ (NULL)\ C\ B$

$= DCB \rightarrow (3)$

Sub (3) in (1):-

$\therefore$ Postorder $=$ **GFEIHDCBA**

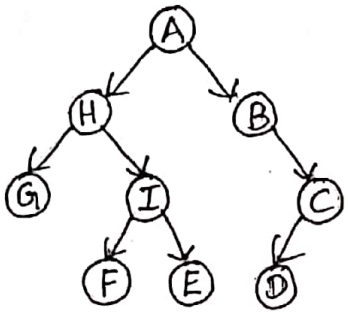
Scanned by CamScanner

PREORDER TRAVERSAL

```
Procedure PREORDER_TRAVERSAL (NODE);
   IF NODE ≠ NIL then
   {
   Print DATA (NODE);
   Call PREORDER_TRAVERSAL(LCHILD(NODE));
   Call PREORDER_TRAVERSAL (RCHILD(NODE));
   }
End PREORDER_TRAVERSAL
```

The traversal processes every node as it moves left until it can move no further. Now it turns right to begin again of if there is no node in the right, retracts until it can move right to continue its traversal. The above algorithm performs Preorder Traversal recursively.

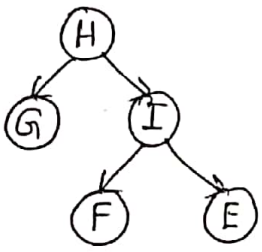
Eg:



Preorder Traversal = Root (Left) (Right)

= A (Left) (Right) ⟶ (1)

Left:-



Since the left child of A is a tree, traverse it by preorder method.

Left = Root (Left) (Right)

= H (Left) (Right)

= H (G) (Right)

Scanned by CamScanner

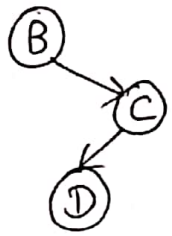
$= HG\ [(Root)\ (Left)\ (Right)]$

$= HGIFE \longrightarrow (2)$

Sub. (2) in (1):-

$\therefore$ Preorder $= AHGIFE\ (Right) \longrightarrow (1)$

Right:-



Since Right child of A is a tree, do preorder trave:

Right = Root (Left) (Right)

= B (NULL) (Right)

= B [Root (Left) (Right)]

= BCD (NULL) = BCD $\longrightarrow$ (3)

Sub (3) in (1):-

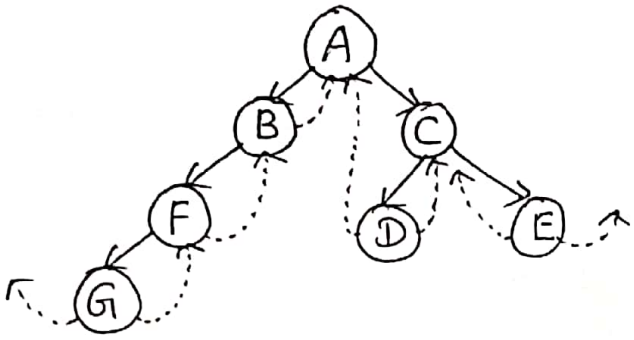
Hence Preorder = **AHGIFEBCD**

Scanned by CamScanner

THREADED BINARY TREES

The linked representation shows that to represent n nodes, we are in need of 2n pointers of which (n+1) are null. Two experts Perlis and Thornton provided a new way of utilizing those (n+1) null pointers called as threads. Threads are defined as links or pointers that replace the null pointers by pointing to some useful information in the tree. If R CHILD(NODE) is NULL, then the null pointer is replaced by a thread which points to the node which would occur after NODE when the binary tree is traversed as Inorder. Similarly if LCHILD(NODE) is NULL, then the null pointer is replaced by a thread which points to the node which would occur before NODE when the binary tree is traversed as Inorder.

Eg:



INORDER TRAVERSAL= **G F B A D C E**

Note that the left child of G and the right child of E have threads which are left dangling due to the absence of an inorder predecessor and successor respectively.

REPRESENTATION OF THREADING

One way threading Two way threading

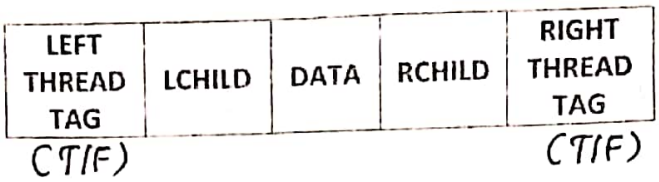
1) **One way threading** is where a thread appears only on the RCHILD field of a node, when it is null, pointing to the inorder successor of the node.

2)**Two way threading** has a thread appears in the LCHILD field also, if it is null, which points to the inorder predecessor of the node. However the first and last of the nodes in the inorder traversal will carry dangling threads.

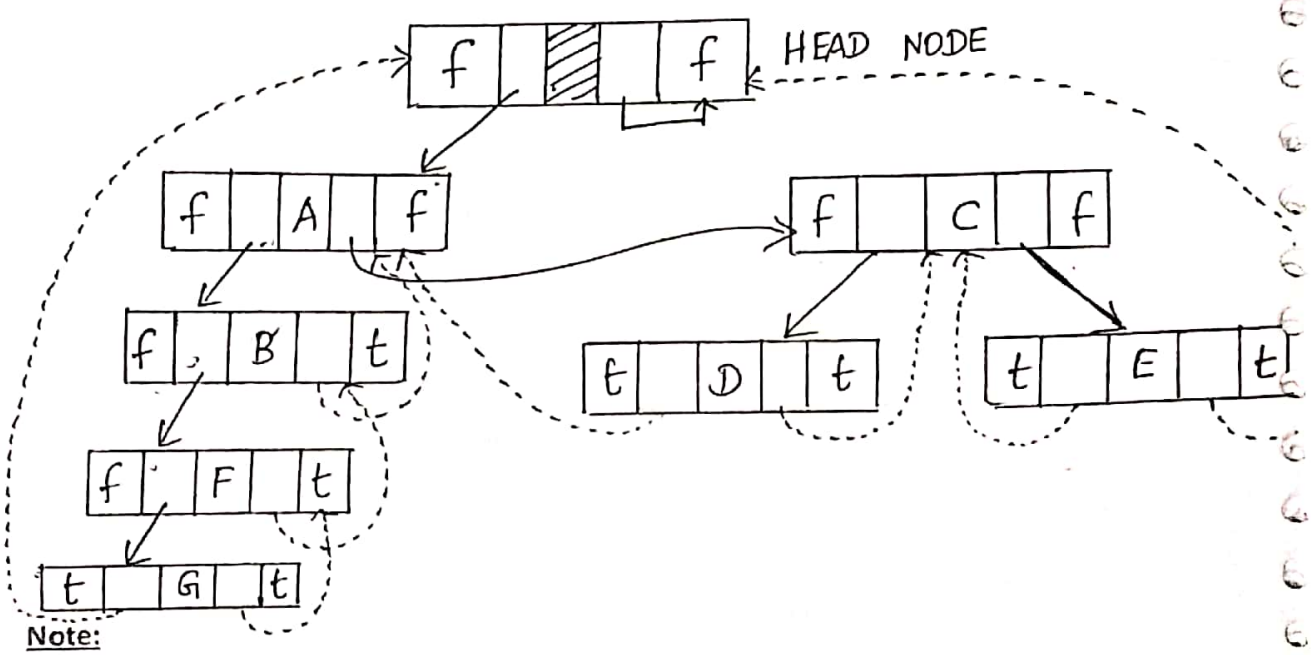
Scanned by CamScanner

Linked Representation of a Threaded Binary Tree:-

Since LCHILD, RCHILD fields are utilized to represent both links and threads, it becomes essential for the node structure for clear distinguish between threads and links-so employ two more flags which carries TRUE/FALSE. The following will be the linked representation of the threaded binary tree (two-way threading) of a node.



If the LEFT THREAD TAG or RIGHT THREAD TAG is marked as true then LCHILD and RCHILD fields represent threads otherwise they represent links. Also to avoid the dangling threads, a head node is employed whose LCHILD is the root of the tree and RCHILD is made to point itself. Hence the LEFT THREAD TAG AND RIGHT THREAD TAG of the head node is marked as false. The following will be the threaded binary tree representation for the above given binary tree.



Note:

IF (LEFT THREAD TAG(OR)RIGHT THREAD TAG =True)

LCHILD or RCHILD are threads

Else

LCHILD or RCHILD are links

Scanned by CamScanner

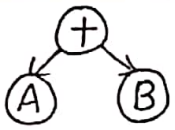
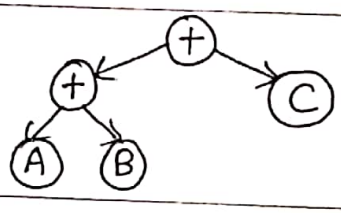
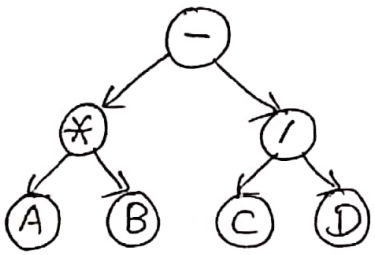
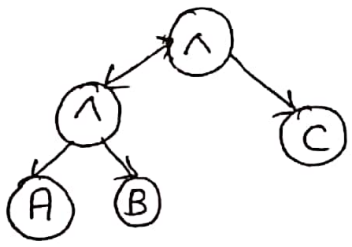
APPLICATION OF TREES

EXPRESSION TREES

Expressions are an inherent component of programming languages. It may be arithmetic or logical.

Eg :A+B*C, A^B$\vee$C, (A≤B)^(C≠ E)

Binary treea have found an application in the representation of expressions. An expression tree has the operand as its terminal or leaf nodes and the operators as its non-terminal nodes. The hierarchical precedence and associativity rules of the operators are to be followed while building the tree.

EXPRESSION	TREE
A+B	
A+B+C	
A*B-C/D	
A^B^C	

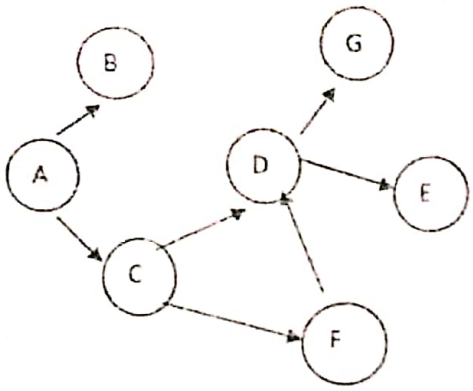
Scanned by CamScanner

GRAPHS

Graphs are non-linear data structures used to represent network structures.

Definition: A graph is defined as G(V, E) where V is the set of vertices (nodes) and E is the set of edges (arcs) connecting the vertices. An edge is represented as a pair of vertices (u,v).

Example:



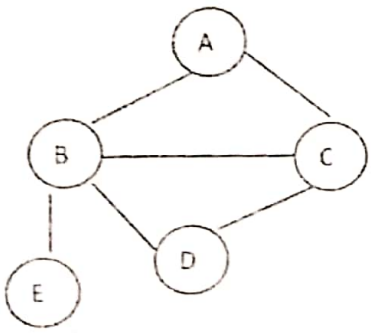
V(G) = {A,B,C,D,E,F,G}

E(G) = {(A,B), (A,C), (C,F), (C,D),(F,D), (D,E), (D,G)}

TYPES OF GRAPHS

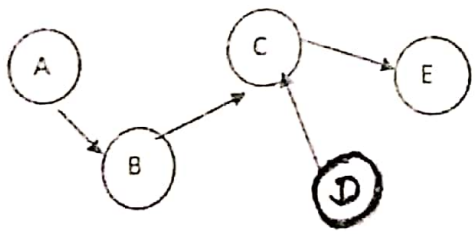
1. **Un-Directed Graph:** Graph that does not have any direction marks is called an undirected graph.

   Example:

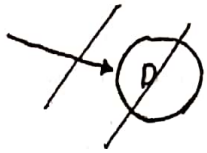


2. **Directed Graph:** Graph in which the edges are directed is called directed graph or digraph. A edge(u,v) starts from u and ends at v. u is the head and v is the tail.

   Example:

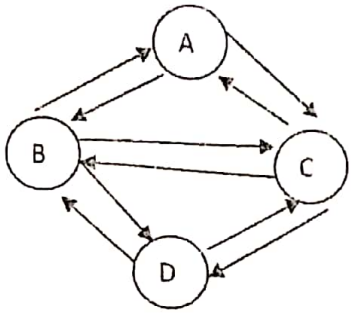


Scanned by CamScanner



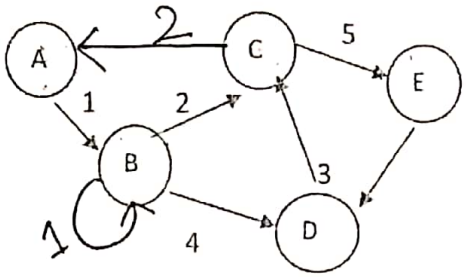
3. **Symmetric digraph:** Graph in which every edge has an edge in the reverse direction. For every edge (u,v) in the graph, there must be one edge with (v,u).

Example:



4. **Weighted Graph**: Graph in which the edges are associated with weights. These weights are used to mark the importance of the edges in the problem represented. For example in road map weight may indicate the traffic flow.

Example:



5. **Degree of a Vertex** is the number of edges incident to it.

   Indegree: Number of edges that are pointing on a vertex

   Outdegree: Number of edges leaves the vertex

6. **Adjacent vertices:** The vertices that are connected by an edge are said to adjacent to each other. In the above example, vertices A and B are adjacent vertices. A path is a sequence of vertices in which each vertex is adjacent to next one.

   Eg: In the above graph, (A,B,C,E) is a path.

7. **A cycle** is a path consisting of at least three vertices that starts and ends with the same vertex.

Scanned by CamScanner

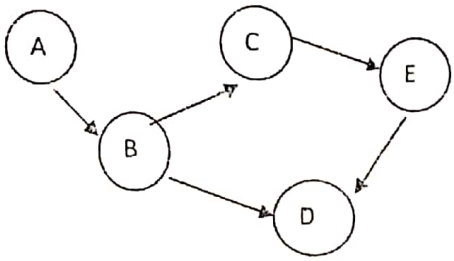
In the above graph ABCA is a cycle.

In a Cycle,

- No node appears more than once.
- First node of the path=Last node of the path.

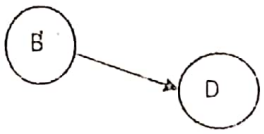
8. **A loop** is a special case of cyle in which a single arc begins and ends with the same vertex.
   Eg: Edge at node B is a loop.

9. A graph with cycles is called as **cyclic graph** while **acyclic graph** is the graph without cycles. directed graph without cycle is called as **DAG** (Directed Acyclic Graph).



10. **SubGraph:** A subgraph G' = (V', E') of a graph (G = (V,E) is such that V'∈ V and E'∈ E

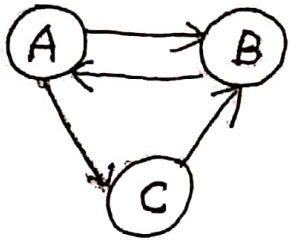
Example: Below is a sub graph of the graph available in the acyclic graph section.



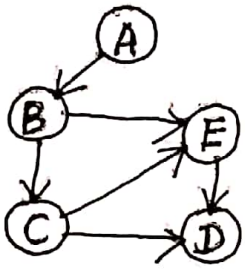
11. **Isomorphic graphs:** Two graphs are said to be isomorphic if,
    - They have the same number of vertices
    - They have the same number of edges
    - They have an equal number of vertices with given degree

Scanned by CamScanner

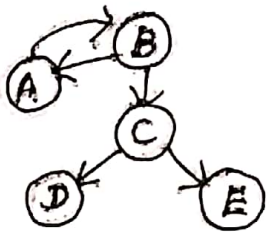
12. A directed graph is **strongly connected** if there is a path from each vertex to every other vertex.



13. A directed graph is **weakly connected** if at least two **vertices** are not connected.

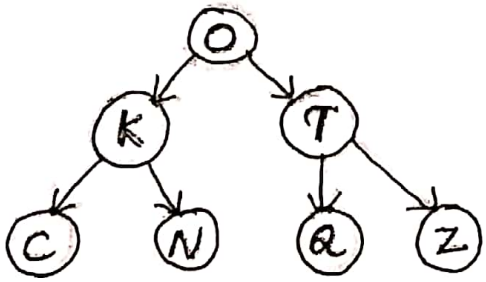


14. A vertex V is called as the **cut-vertex** of a graph if there exists a pair of vertices X and Y such th every path between x and y passes through V.



Here B is the cut-vertex for vertices A and C.

15. A tree is a special case of a directed graph. A graph need not be a tree but a tree must be a gra (Connected acyclic graph).



16. The walk which begins from a vertex and returns to it after traversing all edges in a graph is called **Eulerian Walk**.

Scanned by CamScanner

Graphs

Graphs can be represented in two ways using arrays :-

I. **Adjacency Matrix:**

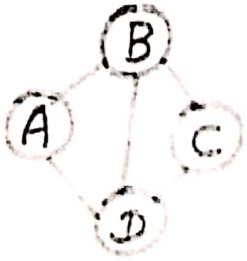
The adjacency matrix M of a graph G is a matrix of order V * V and the elements of M are defined as

$$M[i][j] = \begin{cases} 1, & \text{if } (v_i, v_j) \in E \\ 0, & \text{otherwise} \end{cases}$$

For a weighted graph the elements of M are defined as

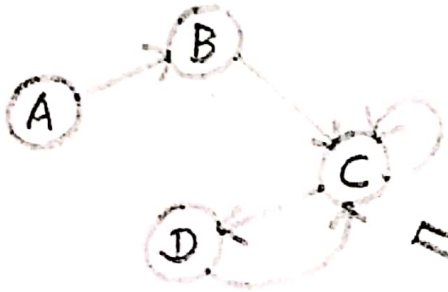
$$M[i][j] = \begin{cases} W_{ij}, & \text{if } (Vi, Vj) \in E \text{ and } W_{ij} \text{ is the weight of edge } (v_i, v_j) \\ 0, & \text{otherwise} \end{cases}$$

Undirected Graph:



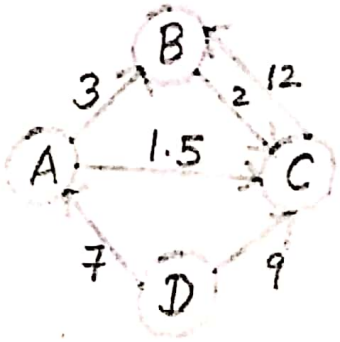
	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	1	0	1
D	1	1	1	0

Directed Graph:



	A	B	C	D
A	0	1	0	0
B	0	0	1	0
C	0	0	0	1
D	0	0	1	0

Weighted Graph:



	A	B	C	D
A	0	3	1.5	0
B	0	0	2	0
C	0	12	0	0
D	7	0	9	0

Scanned by CamScanner

Observations:

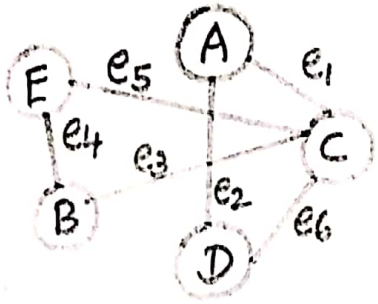
1. i^{th} row in the adjacency matrix is determined by the edges which start from vertex V_i.
2. The number of elements in the ith row whose value is 1 is the outdegree of the node V_i. (directed graph)
3. The number of elements in the jth column whose value is 1 is the indegree of the node V_j. (directed graph)
4. Null graph has adjacency matrix as null matrix.
5. Adjacency matrix for an undirected graph is symmetric.
6. Space required to represent a graph using adjacency matrix is n^2, where n=number of vertices.

II. Incidence matrix representation:

Incidence matrix I can also be represented as a 2D array of size nXe, where n and e correspon to the number of vertices and edges of the graph repectively.

$$I(Vi,ej)=\begin{cases}1, \text{ if ej is incident on Vi}\\ 0, \text{ otherwise}\end{cases}$$

Since each edge is incident on two vertices, every column of the incidence matrix must ha exactly two entries with values 1. Once again, the sum of the elements of a row denotes the degree the vertex corresponding to the row.



	e1	e2	e3	e4	e5	e6
A	1	1	0	0	0	0
B	0	0	1	1	0	0
C	1	0	1	0	1	0
D	0	1	0	0	0	1
E	0	0	0	1	1	0

Representing a digraph as an incidence matrix as above is not useful, because the direction edges cannot be detected. Let us name the starting and ending of vertices of an edge in a digraph tail and head respectively. In each column, the tail is assigned a value -1 and head the value +1.

$$\text{(ie) } I(Vi,ej)=\begin{cases}+1, \text{ if Vi is at the head of } e_j\\ -1, \text{ if Vi is at the tail of } e_j\\ 0, \text{ otherwise}\end{cases}$$

	e1	e2	e3	e4	e5	e6
A	-1	1	0	1	0	0
B	1	-1	-1	0	-1	-1
C	0	0	1	-1	0	0
D	0	0	0	0	0	1
E	0	0	0	0	1	0

Scanned by CamScanner

iii. Adjacency Lists:

The adjacency matrix/incidence matrix are static in nature (ie) to use this method, we m know the number of vertices in advance for storage. If a graph is updated dynamically during progra execution (add or delete a node), the dimension of the array has to be changed. The adjacen matrix/incidence matrixes require storage for each possible edge between nodes whether edge exi or not. The space must be reserved for every possible edge even if the adjacen cy matrix is sparse.

So the solution to the above problem is use a linked structure, which makes allocations a deallocations from a pool. This is called as adjacency lists which stores information only about t existing edges. Adjacency lists use an array of linked lists. The size of array (number of linked lists) same as the number of vertices in the graph. List i will have all the nodes that are adjacent to node i In a weighted graph, each node of the linked list will have the adjacent vertex number, weight of t edge and the pointer to the next node in the list.

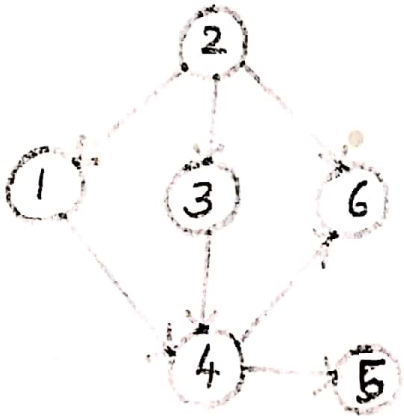
The adjacency list consists of a directory and set of linked lists, called as node directo representations. The directory has one entry for each node. Each entry in the directory points to t linked lists corresponding to the node it has edge. The node mainly contains two compomnets as

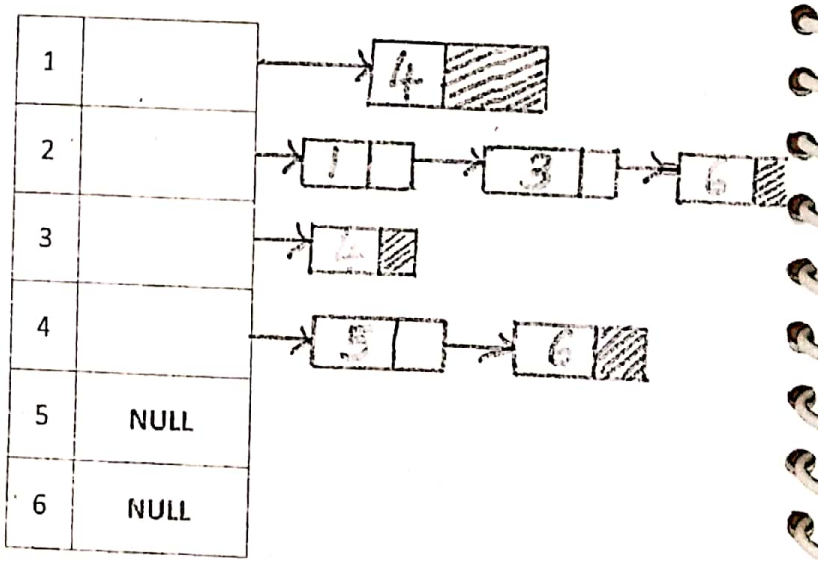
I. Node Identifier
II. Linked to the next field
III. Optional third field to represent weight (if any).

NODE ID	WEIGHT	NEXT

NODE ID	NEXT

Example:





Scanned by CamScanner

Let a graph has N nodes and E edges. Directed graph will have N entries in directory and E linked list entries. Undirected graph will have N entries in directory and 2E linked list entries sin each edge will be stored twice. Outdegree can be determined easily while not the indegree of a node

A traversal is a systematic walk which visits the nodes comprising the data structu (graphs in this case) in a specific order. We can traverse a graph in two ways.

i. Breadth first traversal
ii. Depth first traversal

Traversal in graphs is harder than trees because of multiple paths between the nodes

I. Breadth first traversal:

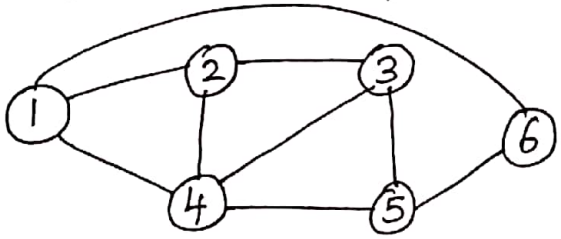
The traversal starts from a vertex u which is said to be visited. Now all nodes V_i adjacent to u visited. The unvisited vertices adjacent to each of V_i are visited next and so on. The traver terminates when there are no more nodes to visit. This procedure calls for the maintenance of a que to keep track of the order of nodes whose adjacent nodes are to be visited.

Procedure BFT(s)

```
/* s is the start vertex of the traversal in an undirected graph G
   Q is a queue which keeps track of the vertices whose adjacent nodes are to be visited.
   Visited flag of the visited vertices will be set as 1 (i.e. visited(vertex)).
   Initially visited(vertex) = 0 */

Initialize queue Q;
visited(s) = 1;
call ENQUEUE(Q,s);
while not EMPTY QUEUE(Q) do
    call DEQUEUE(Q,s);
    print(s)
    for all vertices v adjacent to s do
        if (visited(v)=0) then
        {
            call ENQUEUE(Q,v); // insert unvisited vertices into Q
            visited(v) = 1;
        }
    end
end while
end BFT
```

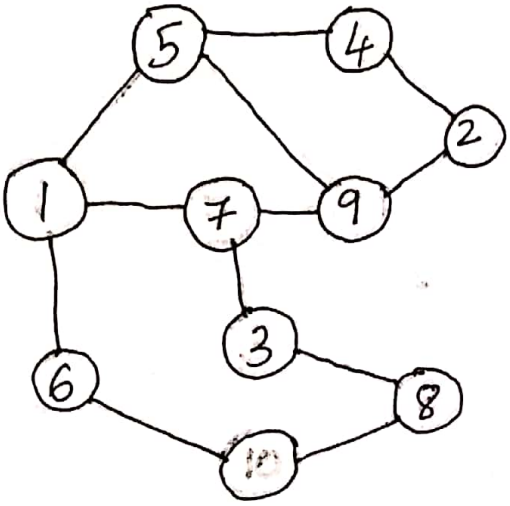
BFT traverses the successors of the start node, generation after generation i horizontal/linear fashion. Time complexity of the algorithm while using adjacency matrix is $O(n^2)$ a while using adjacency list is $O(V+E)$.

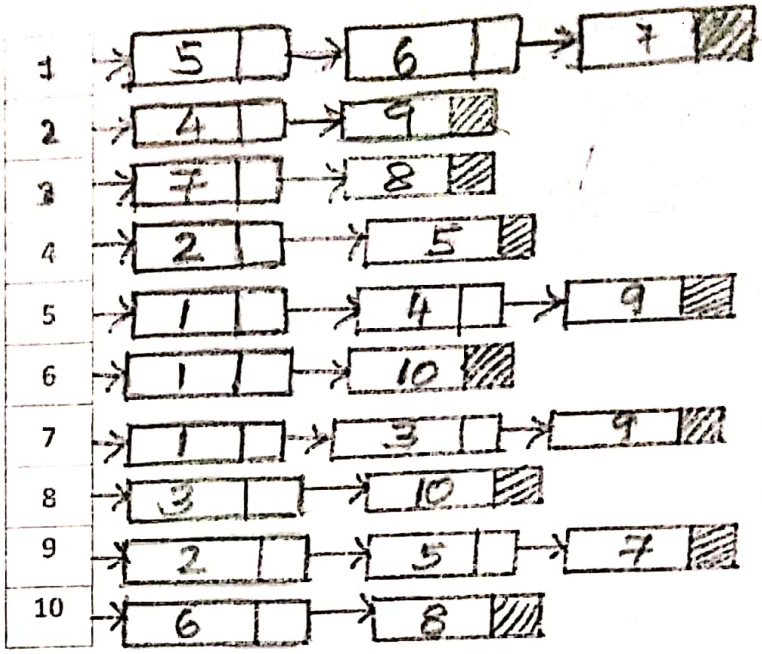
Eg:



Scanned by CamScanner

Example:





Trace of the BFT:

Current Vertex	Queue Q	Traversal Output	Status of visited flag of vertices (1 2 3 4 5 6 7 8 9 10)
------	Empty Queue	--------	0 0 0 0 0 0 0 0 0 0
1	5 6 7	1	1 0 0 0 1 1 1 0 0 0
5	6 7 4 9	1 5	1 0 0 0 1 1 1 0 1 0
6	7 4 9 10	1 5 6	1 0 0 0 1 1 1 0 1 1
7	4 9 10 3	1 5 6 7	1 0 1 0 1 1 1 0 1 1
4	9 10 3 2	1 5 6 7 4	1 1 1 1 1 1 1 0 1 1
9	10 3 2	1 5 6 7 4 9	1 1 1 1 1 1 1 0 1 1
10	3 2 8	1 5 6 7 4 9 10	1 1 1 1 1 1 1 1 1 1
3	2 8	1 5 6 7 4 9 10 3	1 1 1 1 1 1 1 1 1 1

Scanned by CamScanner

Current Vertex	Queue Q	Traversal Output	Status of visited flag of vertices
2	8	1 5 6 7 4 9 10 3 2	1 1 1 1 1 1 1 1 1 1
8		1 5 6 7 4 9 10 3 2 8	1 1 1 1 1 1 1 1 1 1

II. Depth first traversal:

The traversal starts from a vertex u which is said to be visited. Now, all the nodes v_i adjacent to ver u are collected and the first occurring vertex v_1 is visited, deferring the visits to other vertices. The noc adjacent to v_1 viz., w_{1k} are collected and the first occurring adjacent vertex viz., w_{11} is visited deferring visit to other adjacent nodes and so on. The traversal progresses until there are no more visits possible. T depth first traversal as its name indicates visits each node, that is, the first occurring among its adjace nodes and successively repeats the operation, thus moving 'deeper and deeper' into the graph.

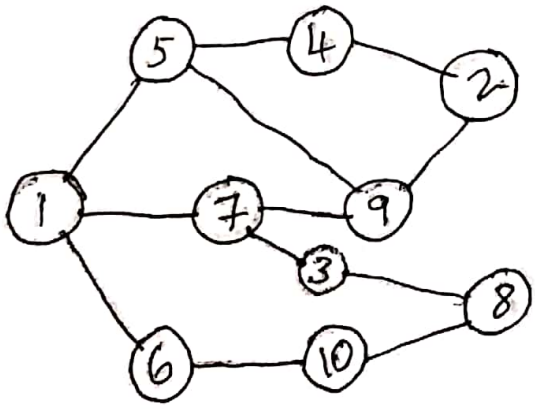
Procedure DFT(s)

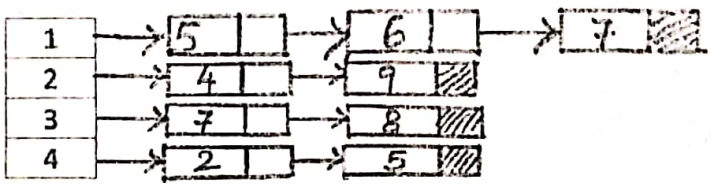
/* s is the start vertex of the traversal in an undirected graph G */

```
visited(s) = 1;
print(s);

for each vertex v adjacent to s do
    if(visited[v]=0) then // traverse unvisited adjacent vertex v of S.
    {
        call DFT(v);
    }

end DFT
```

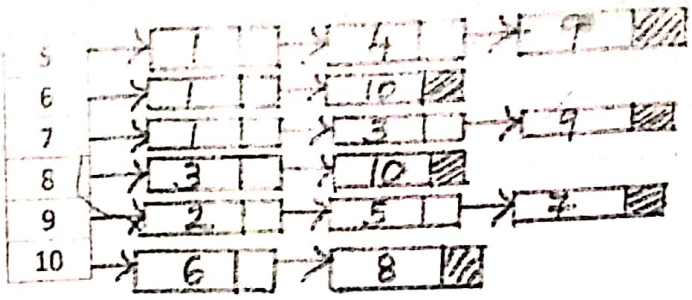
Time complexity of DFT using adjacency list is O(e) and using adjacency matrix is $O(n^2)$.

Example:

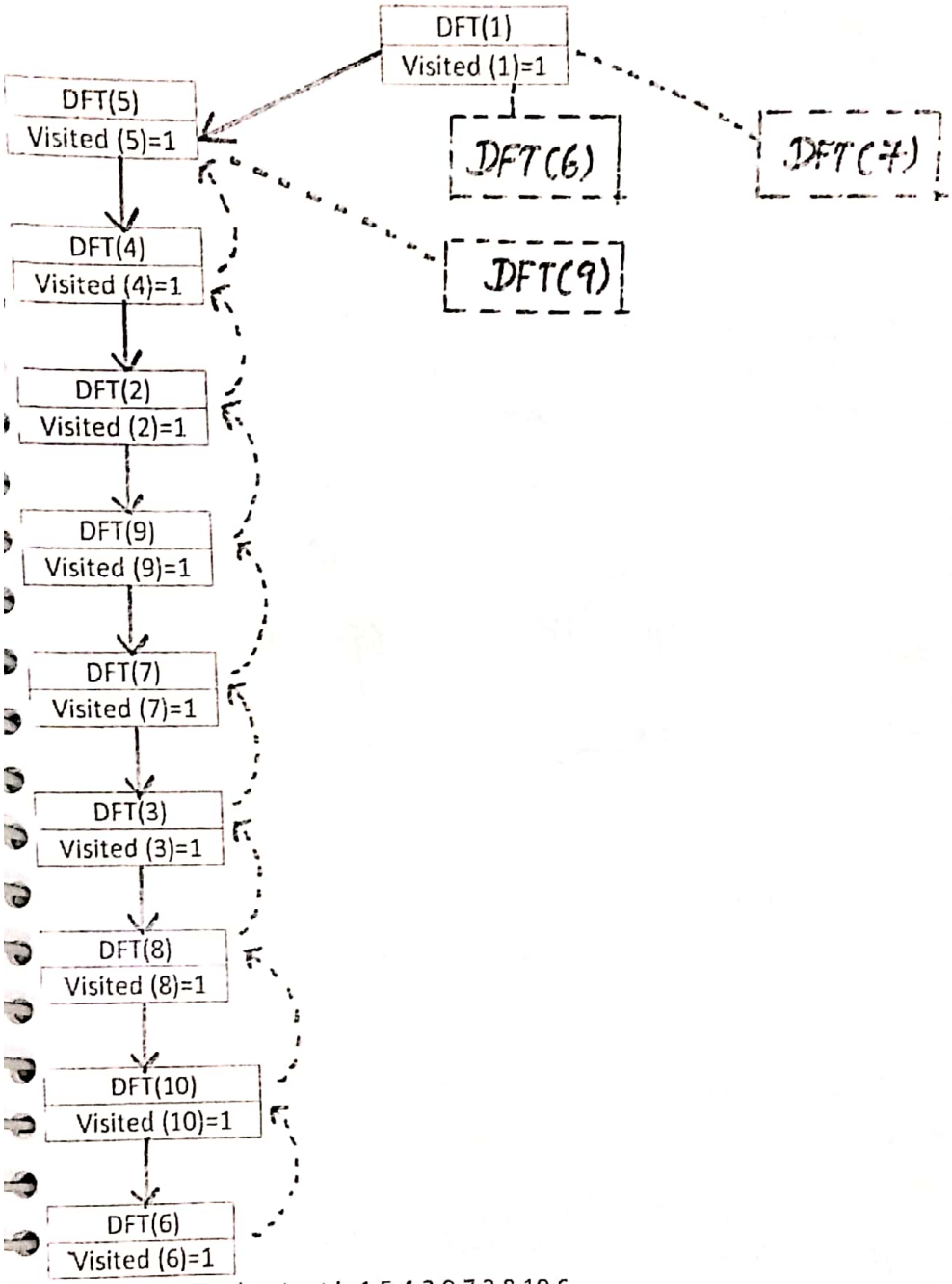




Scanned by CamScanner



Trace of the DFT:



Hence traversal output is 1 5 4 2 9 7 3 8 10 6.

NOTE: If the graph is connected, both traversals would visit all the vertices of the graph; if the graph is connected, both traversals will give only their connected components.

Scanned by CamScanner

APPLICATIONS OF GRAPHS

There are two main applications of graphs as discussed below:

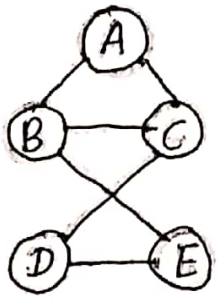
1. Constructing Minimum Cost Spanning Tree
2. Shortest Path Algorithm

MINIMUM COST SPANNING TREE

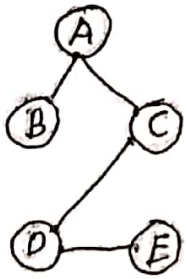
A Spanning Tree is a tree that contains all the nodes of graph and has no other nodes. When the graph is connected, a traversal staring at any vertex, visits all the vertices in G. The edges of G are partitioned into two sets: T (Tree edges) and B (Back edges or Non-tree edges). Spanning tree should not include cycles. G(V, E) is the graph & T(V', E') is the spanning tree such that V' = V and E' ∈ E.

AIM: To find a spanning tree T of minimum cost for a given graph G, which means that the total weights of the edges of the spanning tree must be minimum compared to all other spanning trees generated from G.

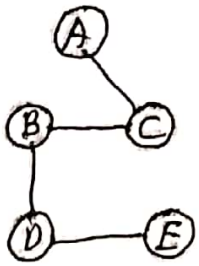
EG:



Given Graph



Spanning tree 1

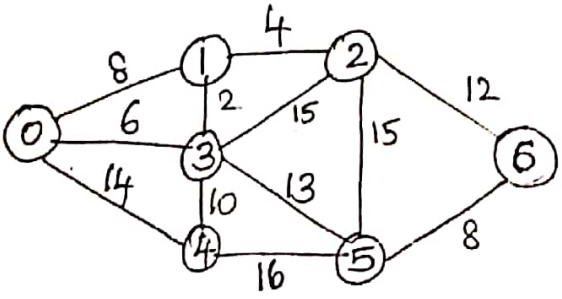


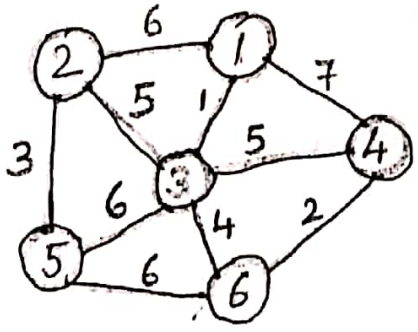
Spanning tree 2

Let G(V,E) is a weighted connected undirected graph, E' is the set of edges which are to be extracted.

```
PROCEDURE PRIM(G)
    E' = NULL; // Edge set of Spanning tree
    Select a minimum cost edge (u,v) from E;
    V' = {u}; // vertices of spanning tree
    while V' ≠ V do
        Let (u,v) be the lowest cost edge such that u in V-V' and v is in V'
        Add edge(u,v) to set E';
        Add v to set V';
    Endwhile
End PRIM
```

1. Kruskal's algorithm:- First include all vertices and then add edges.
2. Prim's algorithm:- Start with root & then add edge and vertex one by one.





Scanned by CamScanner

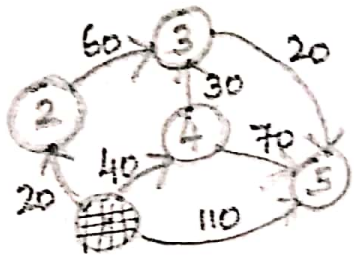
EG: Dijkstra's algorithm has a complexity of $O(N^2)$ where N=number of cities in weighted digraph.

Procedure DUKSTRA SSSP(N, Cost)

N=Number of vertices labeled {1,2,3,...N} of the weighted digraph. COST[i,j] is the cost matrix of the graph. If there is no edge then COST[i,j]=∞. The procedure computes the cost of the shortest path from vertex 1 the source, to every other vertex of the weighted digraph. Initially DISTANCE vector to the cost of the edges connecting vertex i with the source vertex

```
T={1};
for i=2 to N do
    DISTANCE[i] = COST[1,i];
for i=1 to N-1 do
    Choose a vertex u in V-T such that DISTANCE[u] is a minimum
    Add u to T;
    for each vertex w in V-T do
        DISTANCE[w] = minimum(DISTANCE[w], DISTANCE[u]+COST[u,w]);
    end for
end for
end DIJKSTRA_SSSP.
```

Consider the following network of cities.

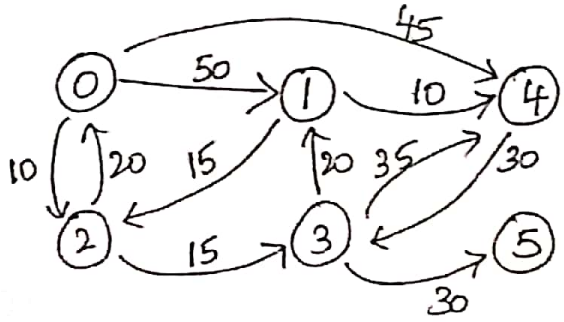


	1	2	3	4	5
1	0	20	∞	40	110
2	∞	0	60	∞	0
3	∞	∞	0	∞	20
4	∞	∞	30	0	70
5	∞	∞	∞	∞	0

Iteration	T	u	DISTANCE			
			2	3	4	5
Initial	{1}	-----	(20)	∞	40	110
1	{1,2}	2	(20)	80	40	110
2	{1,2,4}	4	(20)	70	(40)	110
3	{1,2,4,3}	3	(20)	(70)	(40)	90
4	{1,2,4,3,5}	5	(20)	(70)	(40)	(90)

Hence the shortest path to cities 2,3,4 and 5 from source city 1 has been calculated.

Eg:

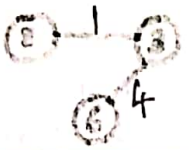
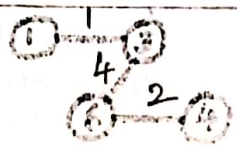
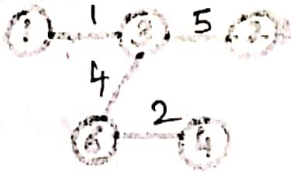
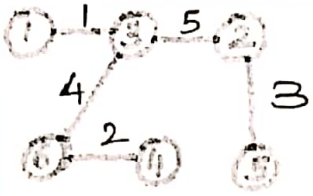


Source	Destination	Path	Distance
1	2	1-2	20
	3	1-4-3	70
	4	1-4	40
	5	1-4-3-5	90

Scanned by CamScanner

Make sure that addition of even one single edge results in the spanning tree losing its property of acyclicity and removal of one single edge results in losing the property of connectivity. Time complexity of PRIM algorithm is $O(n^2)$.

EG:

Edge	Cost of the Edge	V'	E'	Spanning tree
(1,3)	1	{1,3}	[(1,3)]	
.(3,6)	4	{1,3,6}	[(1,3),(3,6)]	
(6,4)	2	{1,3,6,4}	[(1,3),(3,6),(6,4)]	
(3,2)	5	{1,3,6,4,2}	[(1,3),(3,6),(6,4),(3,2)]	
(2,5)	3	{1,3,6,4,2,5}	[(1,3),(3,6),(6,4),(3,2),(2,5)]	

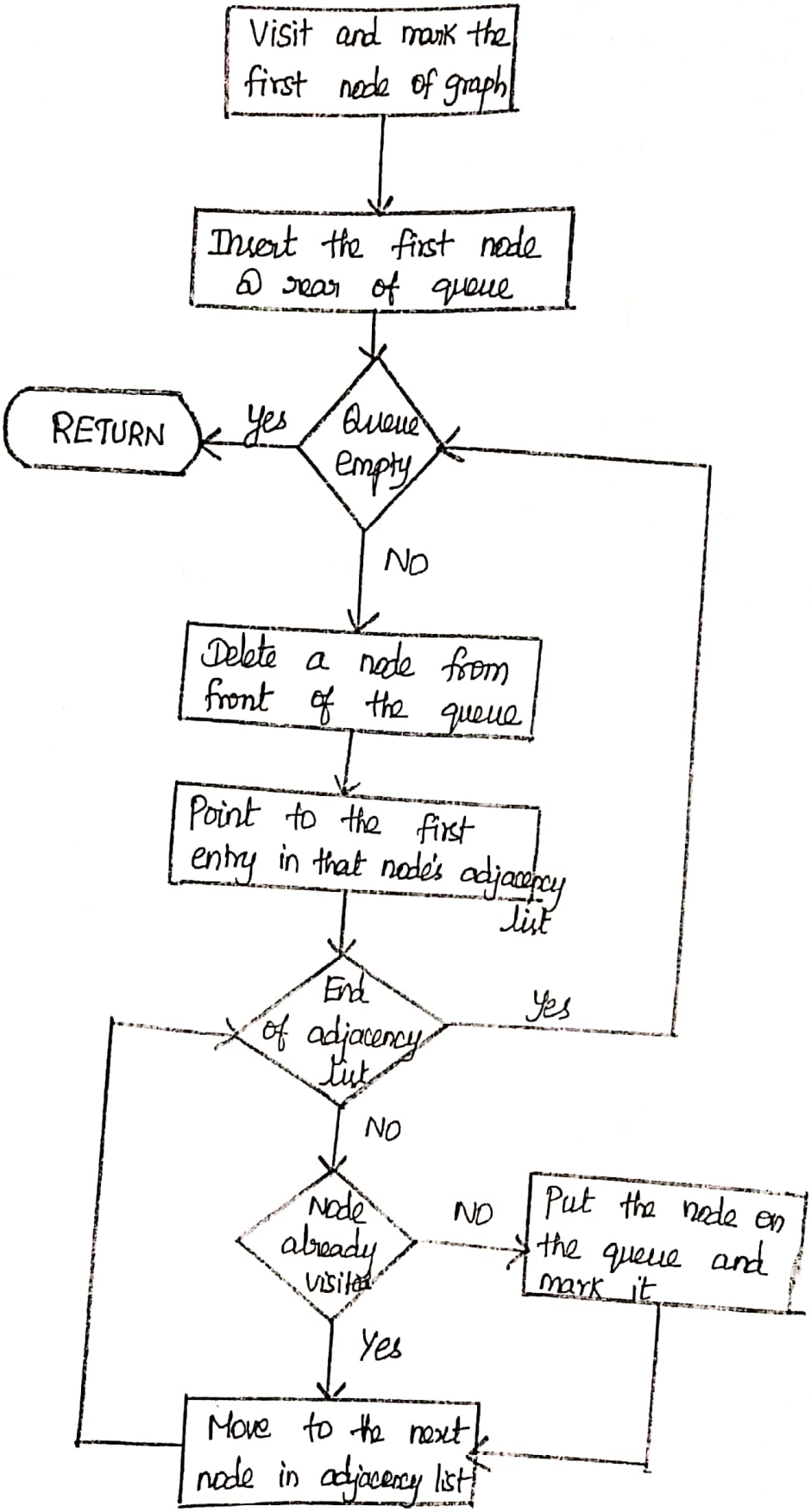
Total cost = (15)

SHORTEST PATH PROBLEM

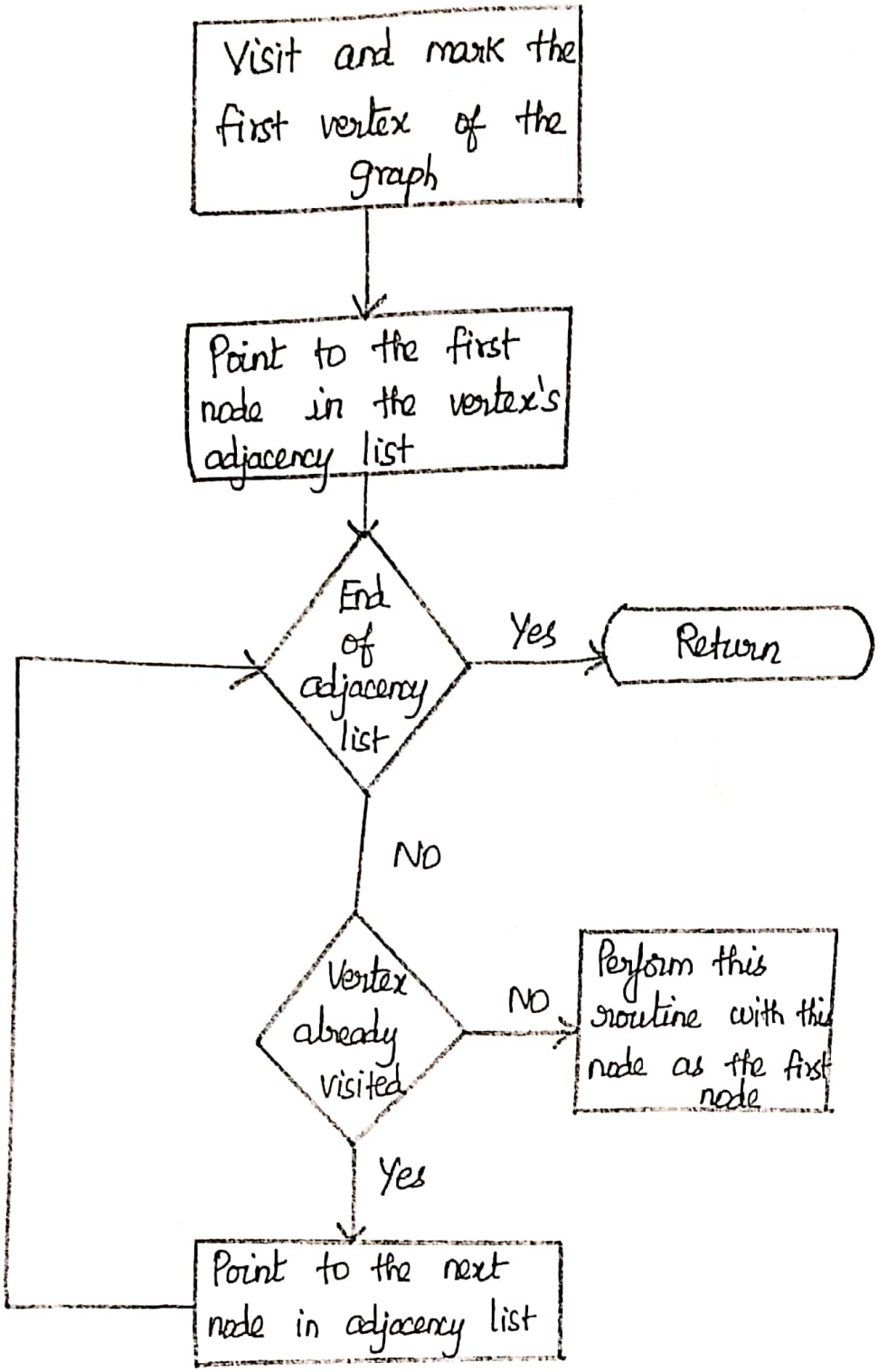
Given a network of cities and the distance between them, the objective of the single-source-shortest-path problem is to find the shortest path from a city to all other cities connected to it. The network of cities with their distances is represented as a weighted digraph. Let v be a set of N cities (vertices) of the digraph. Here the source is CITY1. The set T is initialized to CITY 1. The distance vector, DISTANCE [2:N] initially records the distances of cities from 2 to N connected to the source by an edge. If there is no edge directly connecting the city to the source, then we initialize its DISTANCE value to ∞ (infinity).

Once the algorithm completes its iterations, the DISTANCE vector holds the shortest distance of cities 2 to N from the source city 1. It is convenient to represent the weighted digraph using its cost matrix $COST_{NxN}$. The cost matrix records the distances between cities connected by an edge.

Scanned by CamScanner



Scanned by CamScanner



Scanned by CamScanner

SEARCHING TECHNIQUES

Searching is a day-to-day activity. Information retrieval is one of the most important applications of computers. A table or file is a collection of records; each record having one or more fields. Fields used to distinguish among records are known as Keys. There is at least one key which identifies a record uniquely called as **Primary Key.** If a key retrieves more than one record, then it is called as **Secondary Key**.

Eg: Telephone number in a directory – Primary key
Customer name in a directory – Secondary key

External searching is performed when most of the table is kept in secondary storage, whereas when the file or table is stored entirely in the main memory it is called as internal searching. Every searching algorithm accepts two arguments-the target key "x" and the list in which the search is to be done. The algorithm will search the key; if found return "success" else return "failure". An algorithm which adds the target "x" during the unsuccessful search in the list is called as search and insert algorithm.

Eg: Linear search, Transpose sequential search, Interpolation search, Binary search, Fibonacci search, Tree search, DFS, BFS, Indexed sequential search.

LINEAR SEARCH

The simplest search technique is the sequential search. We start at the beginning of a list and search for the desired record by examining each subsequent record until either the desired record is found or the list is exhausted. This technique is suitable for a table organized as an array or linked list. It can be applied to an unordered table but the efficiency of the search may be improved if the list is ordered.

Scanned by CamScanner

ORDERED LINEAR SEARCH Let $L=\{k_1,k_2,k_3,.....k_n\}$ where $k_1<k_2$, $k_2<k_3$,.....$k_{n-1}<k_n$ be the list of ordered elements. To search for a key, compare key k with each k_i. If $k>k_i$, progress the search; if $k=k_i$, search is successful else unsuccessful search.

```
Procedure ORDERED-LINEARSEARCH (L,n,K)
i=0;
while (i<n) and (K>L[i]) do
   i=i+1;
end while
if (K=L[i]) then
   {
        Print "Key Found";
        Return i;
   }
else
   Print "Key not Found";
End ORDERED_LINEAR SEARCH.
```

NOTE: Time complexity of ordered linear search can be given as
Worst and Average case=O(n) and O(n/2) respectively
Best case=O(1)

Eg: L[0..5]={16,18,56,78,90,100} and key is 78

i) L[0] = 78 is wrong (16 ≠ 78)

ii) L[1] = 78 is wrong (18 ≠ 78)

iii) L[2] = 78 is wrong (56 ≠ 78)

iv) L[3] = 78 is correct (78 = 78)

So the key is found after performing 4 comparisons.

Scanned by CamScanner

UNORDERED LINEAR SEARCH: In this search, a key k is looked for in an unordered linear list $L=\{k_1,k_2,k_3,.....k_n\}$. It follows "Brute-Force Method".

```
Procedure UNORDERED - LINEARSEARCH (L,n,K)
i=0;
While ((i<n) and (K≠L[i])) do
   i=i+1;
end While
if (K=L[i]) then
  {
       Print "Key Found";
       Return i;
  }
else
   Print "Key not Found";
End UNORDERED - LINEARSEARCH.
```

NOTE: Time complexity of unordered linear search can be given as

Worst case=O(n)

Average case=O(n+1/2) $\simeq O(n)$

Best case=O(1)

Eg: L[0..5]={23,14,98,45,67,53} and the key=53

i) L[0] = 53 is wrong (23 ≠ 53)

ii) L[1] = 53 is wrong (14 ≠ 53)

iii) L[2] = 98 is wrong (98 ≠ 53)

iv) L[3] = 453 is wrong (45 ≠ 53)

v) L[4] = 53 is wrong (67 ≠ 53)

vi) L[5] = 53 is correct (53 = 53)

Hence the key is found after doing 6 Comparisons.

Scanned by CamScanner

It is not always possible to predict the relative demand of the records. Moreover it is true that the probability of a retrieval of a given record may change over time. One method to accomplish the movement of records is successfully, the retrieved record is removed from its current position in the list and key at front of the list called as **move-to-front method or transposition** (group the most needed records at front). This is efficient for linked list. But there is no guarantee that the same record is requested more frequently. In due course, this record may be moved back by some other records. But sequential search is inefficient for larger lists i.e. to search the key in a list of N elements where the key is the last record (perform N comparisons).

BINARY SEARCH

Binary search or Logarithmic search or Bisection search is another efficient search technique that operates on ordered lists. It can be seen as an extension of ordered linear ~~list search~~ search.

To search a particular item with a certain value "x", the approximate middle entry of the table is located and its key value is examined. If its value is higher than the target, the key value can be searched in the first half of the list and the procedure is repeated until the required item is found. If the value is lower than the target, ~~the key~~ the key value can be searched in the second half of the list and the procedure is repeated until the required item is found. Continue the process until the required key is found or the search interval becomes empty.

This method is called as binary search because at each step, we reduce the table size by half. This algorithm can be defined as a recursive one. If the item is being searched for is not equal to the middle element, we must search a sub array using the same method i.e., Let ordered list L=$\{k_1,k_2,k_3,.....k_n\}$. It follows "Divide-Conquer Method". Let the element to be searched be k. Compare k and k_{mid} as

i) if ($k=k_{mid}$), binary search is done.
ii) if ($k<k_{mid}$), continue binary search from 1 to mid-1.
iii) if ($k>k_{mid}$), continue binary search from mid+1 to n.

```
Procedure BINARY-SEARCH (L, low, high, K)
if (low>high) then
   {
      BINARY-SEARCH =0;
      Print "Key not Found";
      Return;
   }
else
```


Scanned by CamScanner

```
{
    mid = (low+high) /2;
    case
        K=L[mid]: {  Print "Key Found";
                     BINARY_SEARCH = mid;
                     Return L[mid];
                  }
        K<L[mid]: BINARY-SEARCH (L,low, mid-1);
        K>L[mid]: BINARY-SEARCH (L, low, mid+1, high);
    End case
}
end BINARY-SEARCH.
```

NOTE: Worst Case=$O(\log_2 n)$

Let L={12,21,34,38,45,49,67,69,78,79,82,87,93,97,99}. Search k=21 in the given list L. low=1 and high=15. Hence mid=(1+15)/2=8 where k_{mid}=69

Iteration 1: 12 21 34 38 45 49 67 69 78 79 82 87 93 97 99

k_{mid}

Since k< k_{mid}, search k=21 in the first half of the table. So low=1 and high=mid-1=7.

Iteration 2: mid=(1+7)/2=4 and hence $k_{mid=38}$

12 21 34 38 45 49 67

k_{mid}

Since k< k_{mid}, search k=21 in the first half of the table. So low=1 and high=mid-1=3.

Iteration 3: mid=(1+3)/2=2 and hence k_{mid}=21

12 21 34

k_{mid}

Since k= k_{mid}, the search ends in success.

If n is small, sequential search is enough, else go for binary search.

Scanned by CamScanner

s_n=number of buckets searched during successful search

α=loading factor=n/b.

Smaller the loading factor better will be the average case performance and hence the performance of list is better than open addressing.

SORTING TECHNIQUES

Sorting refers to ordering of elements in the given list either in increasing or decreasing order. If the elements is alphanumeric and the sequence of ordering is either numerical or alphabetical is called as **Collating Sequence**. There are many sorting algorithms so far defined with its own set of advantages and disadvantages like sorting by exchange, sorting by insertion, sorting by distribution, sorting by selection etc. A sorting is said to be stable if the keys that are equal retain their relative order of occurrence even after sorting (ie) if k_1,k_2 are two keys such that $k_1=k_2$ and $p(k_1)<p(k_2)$ where $p(k_i)$ is the position index of keys in the unsorted list, then after sorting $p'(k_1)<p'(k_2)$ where $p'(k_i)$ is the index positions of the keys in the sorted list.

If the list of data or records to be sorted is small enough to be accommodated in the internal memory of the computer, then it is called as **internal sorting**. If the records are voluminous and they are at external storage devices like tapes, drums, disks, it is called as **external sorting**.

BUBBLE SORT

Working:

Bubble sort belongs to the category of sorting by exchange or transposition, where the process applies on the pair of elements that are out of order are interchanged until the whole list is sorted. At the end of first of comparisons, the largest element in the list L moves to the last position in the list. Once again the pair wise comparison of elements in the sublist results in the next largest element floating to the last position of the sublist. Then in (n-1) passes where n=number of elements in the list, the list L will be sorted. The sorting is called "Bubble sort" since with each pass the next largest element in the list floats or bubbles to its appropriate position in the sorted list.

```
PROCEDURE BUBBLE-SORT (L,n)
    for (i=1 to (n-1)) do
        for j=1 to n-i do
            if (L[j]>[j+1])
                swap (L[j] and L[j+1])
        end for
    end for
end BUBBLE-SORT
```


Scanned by CamScanner

Eg: L={92,78,34,23,56,90,17,52,67,81,18}

Pass (i)	j	List at the end of Pass i
1	1..10	78,34,23,56,90,17,52,67,81,18,92
2	1..9	34,23,56,78,17,52,67,81,18,90,92
3	1..8	23,34,56,17,52,67,78,18,81,90,92
4	1..7	23,34,17,52,56,67,18,78,81,90,92
5	1..6	23,17,34,52,56,18,67,78,81,90,92
6	1..5	17,23,34,52,18,56,67,78,81,90,92
7	1..4	17,23,34,18,52,56,67,78,81,90,92
8	1..3	17,23,18,34,52,56,67,78,81,90,92
9	1..2	17,18,23,34,52,56,67,78,81,90,92
10	1..1	17,18,23,34,52,56,67,78,81,90,92

Here i keeps the count of the passes and j keeps track of the pair wise element comparisons within a pass.

Performance analysis:

The time complexity of bubble sort in terms of key comparisons is given by $O(n^2)$ since the procedure involves two loops.

Stability analysis:

Bubble sort is a stable sort since equal keys do not undergo swapping.

Eg: L={8^1,8^2,8^3,6}. The repeating keys have been distinguished using their orders of occurrence as superscripts. The partially sorted lists at the end of each pass of the bubble sort algorithm are shown below:

Pass 1={8^1,8^2,6,8^3}
Pass 2={8^1,6,8^2,8^3}
Pass 3={6,8^1,8^2,8^3}

The equal keys 8^1,8^2,8^3 maintain their relative orders of occurrence in the sorted list as well, verifying the stability of bubble sort.

Scanned by CamScanner

INSERTION SORT

Working:

Insertion sort belongs to the family of sorting by insertion. The idea is a new key k is inserted at its appropriate position in an already sorted sublist.

First Pass: k2 is compared with its predecessor k1 and k2 inserts itself at the appropriate position to obtain the sorted list $\{k_1,k_2\}$.
Second Pass: k3 compares itself with its predecessor k1 and k2 and inserts itself at the appropriate position to form the sorted sublist $\{k_1,k_2,k_3\}$.
$(n-1)^{th}$ Pass: k_n compares itself with its predecessors to form the whole sorted list $\{k_1,k_2,....,k_n\}$ with $k_1<=k_2<=k_3<=.....k_n$.

Since each key finds its own position it is called as sinking or sifting technique. As long as the key is lesser than its predecessor elements, the swapping continues. If the preceding element in the sorted sublist is less than or equal to key, then the key is left at its current position and that current pass terminates.

```
PROCEDURE INSERTION-SORT(L,n)
    for i=2 to n do
        key=L[i]
        position=i
        while((position>1) and (L[position-1]>key)) do
            {
                L[position]=L[position-1]
                position=position-1
                L[position]=key
            }
        endwhile
    endfor
end INSERTION SORT
```

Eg: Let the list L={16,36,4,22,100,1,54}

Pass	List L
1	16
2	16,36
3	4,16,36
4	4,16,22,36
5	4,16,22,36,100
6	1,4,16,22,36,100
7	1,4,16,22,36,54,100

Scanned by CamScanner

Performance analysis:

Efficiency Class	Time Complexity
Best Case	O(n) when the list is already sorted in ascending order
Average Case and Worst Case	$O(n^2)$ when the list is sorted in descending order

Stability analysis:

Insertion sort is a stable sort since equal keys do not undergo swapping.

Eg: $L=\{3^1,1,2^1,3^2,3^3,2^2\}$. The repeating keys have been distinguished using their orders of occurrence as superscripts. The partially sorted lists at the end of each pass of the bubble sort algorithm are shown below:

Pass 1=$\{3^1\}$
Pass 2=$\{1,3^1\}$
Pass 3=$\{1,2^1,3^1\}$
Pass 4=$\{1,2^1,3^1,3^2\}$
Pass 5=$\{1,2^1,3^1,3^2,3^3\}$
Pass 6=$\{1,2^1,3^1,3^2,3^3,2^2\}$

The equal keys $2^1,2^2,3^1,3^2,3^3$ maintain their relative orders of occurrence in the sorted list as well, verifying the stability of ~~bubble~~ insertion sort. If the records are small and keys are difficult to compare, insertion sort is recommended.

SELECTION SORT

Working:

Selection sort is based on the principle of repeated selection of elements. The steps involved are:

i) Given an unordered list $L=\{k_1,k_2,...k_n\}$. Select the minimum key k.
ii) Swap k with the first element present in the list (ie) k_1. Now the first smallest element attains its position.
iii) Exclude k_1. Select the next smallest element and put it at second position.
iv) Continue the process and the elements will be sorted in (n-1) passes, in ascending process.

Selection sort can also be done by selecting the maximum number instead of smallest one and swap it with the last number in the list. Result will be in ~~descending~~ ascending order.

```
PROCEDURE SELECTION-SORT(L,n)
        for i=1 to n-1 do
            minimum-index = FIND-MINIMUM(L,i,n)
            Swap(L[I], L[minimum-index])
        end
end SELECTION-SORT.
```


Scanned by CamScanner

```
PROCEDURE FIND-MINIMUM(L, i, n)
        min = i
        for j = i+1 to n do
            if (L[j] < L[min])
        end for
        return (min)
end FIND-MINIMUM
```

Eg: L={71,17,86,100,54,27}

Pass	During Pass	After Pass
1	71,17,86,100,54,27	17,71,86,100,54,27
2	17,71,86,100,54,27	17,27,86,100,54,71
3	17,27,86,100,54,71	17,27,54,100,86,71
4	17,27,54,100,86,71	17,27,54,71,86,100
5	17,27,54,71,86,100	17,27,54,71,86,100 (List is sorted)

Performance analysis:

Time complexity of FIND-MINIMUM=O(n)
Time complexity of SELECTION-SORT=$O(n^2)$

Stability analysis:

Selection sort is not a stable sort since it does not maintain the order of occurrence of equal keys in sorted list.

Eg: L={6^1,6^2,2}

Pass	During Pass	After Pass
1	6^1,6^2,2	2,6^2,6^1
2	2,6^2,6^1	2,6^2,6^1

Scanned by CamScanner

MERGE SORT

Working:

Merge sort belongs to the family of sorting by merge in which two ordered lists of elements are combined or merged into a single ordered list. It was invented by John Von Newmann.

If the list is of length 0 or 1, then it is already sorted else divide the unsorted list into two sublists of about half the size. Sort each sublist recursively by reapplying merge sort and merge the two sublists back into one single sorted list. Merge sort incorporates two main ideas to improve its runtime.

a) A small list will take fewer steps to sort than a large list.
b) Fewer steps are required to construct a sorted list from two sorted lists.

1) Two way merging: Two-way merging deals with the merging of two ordered lists. Let $L1=\{a_1,a_2....a_n\}$ with $a_1<=a_2<=...<=a_n$ and $L2=\{b_1,b_2,...b_m\}$ with $b_1<=b_2<=....b_m$. The two lists L1 and L2 are combined into L by using the following cases:

A1: if ($a_i<b_j$) then drop a_i into L.
A2: if ($a_i>b_j$) then drop b_j into L.
A3:if ($a_i=b_j$) then drop both a_i and b_j into L.

The dropping of elements is continued until the following one of the cases occur:

B1: L1 is smaller than L2. Then drop the remaining elements of L2 into L.
B2: L2 is smaller than L1. Then drop the remaining elements of L1 into L.
B3: Both L1 and L2 are of equal length and both become empty.

After merging the list L will have (n+m) elements.

Eg: L1={4,6,7,8} and L2={3,5,6}.

L	Action done	L1	L2
Empty		4 6 7 8	3 5 6
3	4>3 (A2)	4 6 7 8	5 6
3 4	4<5 (A1)	6 7 8	5 6
3 4 5	6>5 (A2)	6 7 8	6
3 4 5 6 6	6=6 (A3)	7 8	Empty
3 4 5 6 6 7 8	Drop elements of L1 (B2)	Empty	Empty

Performance analysis: The time complexity of Merge sort is O(n)

Scanned by CamScanner

2) **k-way Merging:** Two-way merging can be extended to k ordered lists (ie) L1+L2+ ... +Lk=L with now L will contain (n1+n2+...nk) elements.

3) **Non-recursive merge sort:** The non-recursive merge sort treats the list L of n elements as n independent ordered lists of one element each. During passes, if any element is unable to find a pair, then it will simply carried forward to the next pass.

Pass 1: The n singleton lists are pair wise merged. At the end of pass 1, the merged lists would have a size of 2 elements each.

Pass 2: The lists of size 2 are pair wise merged to obtain an ordered list of size 4.

Pass i: The lists of size $2^{(i-1)}$ are merged to obtain ordered lists of size 2^i.

Eg: Let L={12,56,1,34,89,78,43,10}.

List L	12 56 1 34 89 78 43 10
PASS 1	[12,56] [1,34] [78,89] [10,43]
PASS 2	[1,12,34,56] [10,43,78,89]
PASS 3	[1,10,12,34,43,56,78,89]

Performance analysis: The time complexity of merge sort will be $O(n\log_2 n)$.

Stability analysis: Merge sort is a stable sort as the original order of occurrence of keys is maintained in sorted list also.

Eg: L={2,6^1,6^2,1}

List L	2 6^1 6^2 1
PASS 1	[2,6^1] [1 6^2]
PASS 2	[1 2 6^1 6^2] (list becomes sorted)

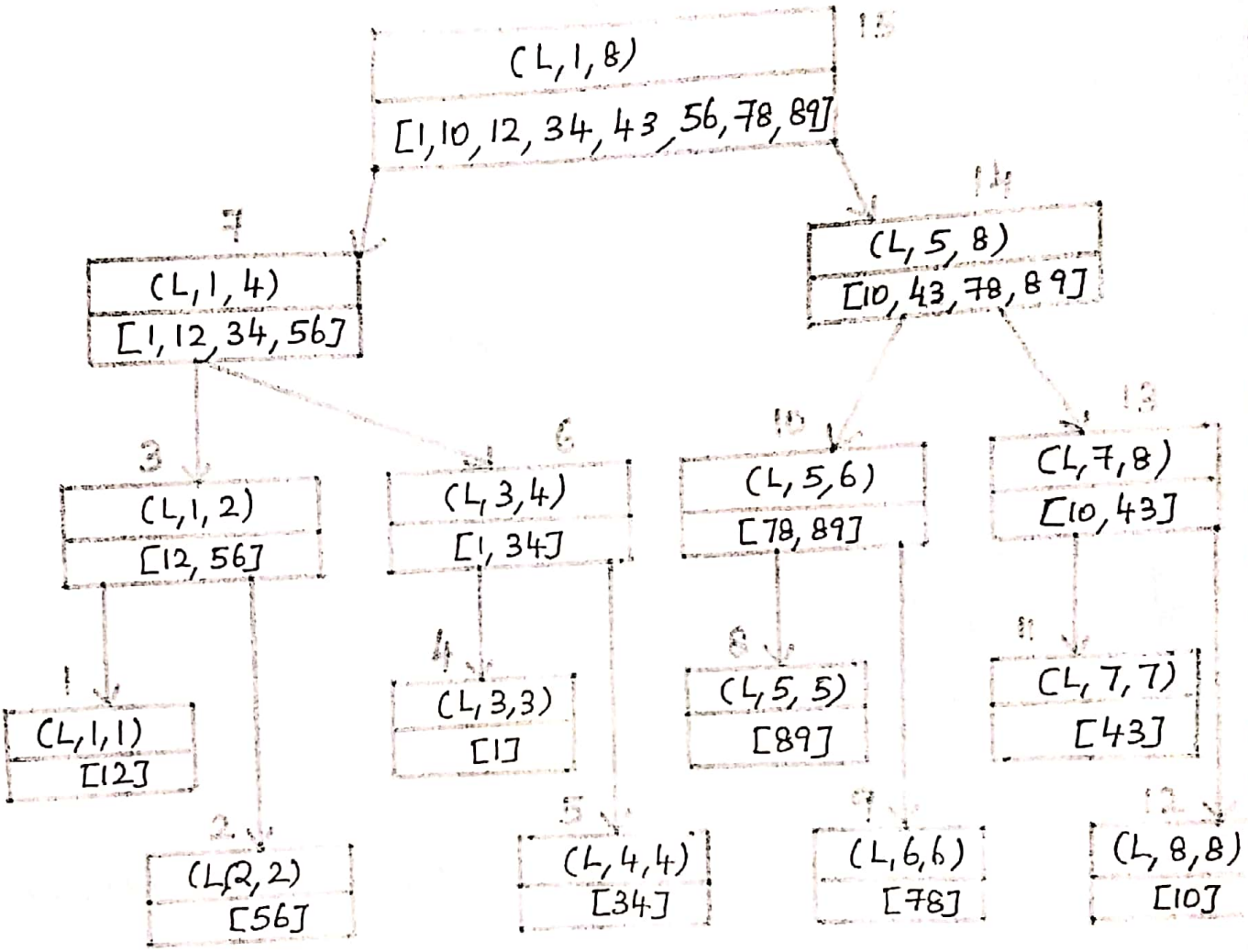
4) **Recursive Merge Sort:** It works on the principle of Divide and Conquer. Here the original unordered list of elements is recursively divided into two sublists until the sublists are small enough where a merge operation is done before they are combined to give the single ordered list.

```
PROCEDURE MERGE-SORT(L, first, last)
    if (first < last) then
        mid = (first + last)/2;
MERGE-SORT(L, first, mid);
```


Scanned by CamScanner

MERGE-SORT(L mid+1, last))
MERGE(L first— last);
end MERGE-SORT.

Eg: Let us apply merge sort for the list L={12,56,1,34,89,78,43,10}.



Note that the MERGE-SORT(L,1,8) generates two other calls MERGE-SORT(L,1,4) and MERGE-SORT(L,5,8) and so on leading to the construction of tree. With MERGE-SORT(L,1,4) giving the sublist [1,12,34,56] and MERGE-SORT(L,5,8) giving the sublist [10,43,78,89], the execution of MERGE(L,1,8) gives the sorted list [1,10,12,34,43,56,78,89].

~~Performance analysis:~~ The time complexity of recursive merge sort is $O(n\log_2 n)$

Scanned by CamScanner

QUICK SORT

Working:

(Quick Sort is a divide and conquer algorithm in which division is dynamically carried out as opposed to static division in Merge Sort.) Quick Sort is formulated by C.A.R. Hoare. It belongs to the family of sorting by exchange or transposition.

The procedure works on the principle of partitioning the unordered list into two sublists at every stage of the sorting process based on an element called as pivot element. The two sublists will occur as left and right sublist of pivot element. The position of pivot element is determined and it is freed from sorting process. Again each sublist will be sorted with their pivot element until no more partitioning is done. Here all elements will have their respective positions and the quick sort will come to an end. Two steps in quick sort are:

i) Divide: Rearrange the elements and split the array into two subarrays and an element in between such that each element in the left subarray is less than or equal to the middle element and each element in the right subarray is greater than the middle element.

ii) Conquer: Recursively sort the two subarrays.

```
PROCEDURE PARTITION (L, first, last, loc)
    left = first
    right = last+1;
    pivot = L[first];
    while (left < right) do
        repeat
            left = left+1; //pivot element moves left to right
        until L(left] >= pivot
        repeat
            right = right - 1; // pivot element moves right to left
        until L[right] <= pivot.
        if (left < right) then
            swap (L[left], L[right]);
    end while
    loc = right;                    // loc is the position where pivot settles down
    swap (L[first], L[right]); //arrows have crossed each other and so swap first element
    Insert Pivot @ loc with right element
```

Swap (Pivot, L[loc])

Scanned by CamScanner

```
PROCEDURE QUICK-SORT(L, first, last)
    if (first < last) then
        PARTITION (L, first, last, loc);
            QUICK-SORT (L, first, loc-1);
            QUICK-SORT (L, loc+1, last);
end QUICK-SORT
```

This is the in-place partition algorithm. It partitions the portion of the array between indexes left and right, inclusively, by moving all elements less than or equal to pivot to the beginning of the subarray, leaving all the greater elements following them. In the process it also finds the final position for the pivot element which it returns. It temporarily moves the pivot element to the end of subarray so that it doesn't get in the way.

Which can be pivot element?

a) Leftmost or rightmost element of the partition.
b) Middle index of the partition.
c) Median of first, middle and last element of the partition (Recommended by Sedgewick).

What we observe about the partitioned sublists and the pivot element?

a) The sublist occurring to the left of the pivot element k has all its elements less than or equal to k and the sublist occurring to the right of the pivot element has all its elements greater than or equal to k.
b) The pivot element has settled down at its appropriate position which would turn out to be its rank in the sorted list.

Summary:

a) Quick sort is a space-optimized version of binary tree sort.
b) Direct competitor of quick sort is heap sort and merge sort.
c) Three well-known variants of quick sort are balanced quick sort, External quick sort and three-way radix quick sort.

Eg:Let L={34,26,1,45,18,78,12,89,27}

Pivot element of L:34	
45,27 spotted	34 26 1 45 18 78 12 89 27 (→ ←)
Exchange 45 and 27	34 26 1 27 18 78 12 89 45
78,12 spotted	34 26 1 27 18 78 12 89 45 (→ ←)
Exchange 78 and 12	34 26 1 27 18 12 78 89 45
Swap 12 and pivot element	[12 26 1 27 18] 34 [78 89 45]

Since arrows cross

Scanned by CamScanner

Let L1={12,26,1,27,18} and L2={78,89,45}

Pivot element of L1:12	
26, 1 spotted	12 26 1 27 18
Exchange 26 and 1	12 1 26 27 18
Swap 1 and pivot element	[1] 12 [26 27 18]
Pivot element: 26	
27,18 spotted	26 27 18
Exchange 27 and 18	26 18 27
Swap 18 and pivot element	18 26 27
Pivot element of L2:78	
89, 45 spotted	78 89 45
Exchange 89 and 45	78 45 89
Swap 45 and pivot element	45 78 89

Hence the sorted list is {1,12,18,26,27,45,78,89}.

Performance analysis:

Efficiency class	Big-oh notation
Worst case	O(n2)
Best case and average case	O(nlogn)

Stability analysis:

Quick sort is not a stable sort since it does not maintain the order of occurrence of equal keys in sorted list as per given in unsorted list.

Eg: L={5^1,5^2,5^3}

Pivot element of L:5^1	
5^2,5^3 spotted	5^1 5^2 5^3
Exchange 5^2 and 5^3	5^1 5^3 5^2
Swap 5^3 and pivot element	5^3 5^1 5^2

Since the sorted list is {5^3 5^1 5^2}, quick sort is said to be unstable.

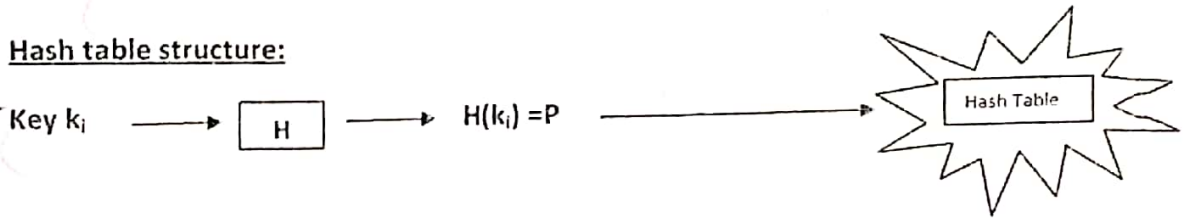
Scanned by CamScanner

HASHING

The search techniques discussed so far are based exclusively on comparing keys. The organization of the file and the order in which the keys are inserted affect the number of keys that must be examined before getting the desired one. If the record keys are integers, t he keys themselves can serve as the index of array. So the approach used to convert a key into an integer within a limited range is called **HASHING,** "the key-to-address transformation maps the key space (k) into an address space (A)". Given a key value k, a hash function h produces a table address where the records may be located. The address that h produces should cover the entire set of indices in the table.

Ideally no two keys should be converted into the same address. Unfortunately no ~~two keys~~ existing hashing function guarantee this. Since key space is larger than the address space, many keys are mapped to the same address. If more than one key try to occupy a same address, it is called as **COLLISION**. If $H(k_1)=H(k_2)=p$, then the keys k_1 and k_2 are called **SYNONYMS**.

Hash table structure:



A hash function H(x) is a mathematical function which is given a key x and maps it to the position P in the storage table called as hash table. The process of mapping the keys to their respective positions in the hash table is called as hashing.

HASHING FUNCTIONS

The main principal criteria for choosing a hash function is that

i) it should be easy and quick to compute.

ii) it should achieve an even distribution of keys over the possible array indices so that is should have a few collisions.

If we know about the keys in advance, a uniform distribution will occur-but this is not possible. If two keys are permutations of each other, they should also hash into different values. Eg: 269,926,692, 629 etc. Here are some of the popular hashing functions in practice.

i) The **Division Method** or **Modular Arithmetic:** The most commonly used hash function is the division method in which an integer key k is divided by the total size m and the remainder is taken as hash value.

Scanned by CamScanner

7 is hash value.

H(k)=k%m+1

Eg: H(45)=45%13+1=6+1=7. This method yields a hash value which belongs to the set {1,2,...m}. Let m=100 and that all the keys ends in the same two digits (may represent a branch code). In such a case, the hash value will be same for all the keys belonging to the same branch. If the length of the table m is a prime number, then the distribution of keys will be better.

ii) **Multiplicative method:** A real number c such that 0<c<1 is selected. For a non-negative integral key k, the hash function is defined as

H(k)=floor(m(ck%1))+1

ck%1=fraction part of c*k and floor yields the nearest integer which is less than or equal to its contents. Here multiply the key by a real number between 0 and 1, take the fractional part of the product to a random number between 0 and 1 which depends on every bit of key and multiply it by m to get an index between 0 to m-1. If c is chosen properly, it will produce good results.

iii) **Midsquare method:** A key is multiplied by itself and the hash value is obtained by selecting an appropriate number of digits from the middle of the square. The same positions in the square must be used for all keys.

Eg: Let the key is 12345 and its square is 152199025. If we are in need of 2 digit address, we can choose values at position 4 and 5 and so we get the address 19. This method can be used together with the division methods.

iv) **Folding method:** A key is broken into several parts-each part has the same length except the last part. All the parts are added together and ignore the final carry. If the keys are in binary form, all parts are exclusive ORed together to form a hash value.

Eg: Let key is 72963195. Divide the key into 3 parts as first two parts is having 3 digits and the last part with 2 digits (if we are in need of 3 digit address). Add those 3 parts as, 729+631+95=1455. Ignore the MSB 1 and declare "455" as the hash value. Since all parts of a key are used, folding often achieves a good spreading of indices.

v) **Digit Analysis:** It forms the address by selecting and shifting digits of the original key. Same rearrangement pattern should be made for all key sets. Digit positions are analyzed and the ones having the most uniform distributions are selected.

Eg: Let key k=7654321. Let the procedure be selecting digits in positions 1,2,4 and 7 and reverse the result. Applying the procedure in the given key k will yield the hash value as 1467.

Other Hashing Functions:

i) If a key consists of letters, each letter may be converted to digits by using the numbering 1 to 26 corresponding to letter A-Z.

ii) If the key is a real number, truncate the digits to have the key as integer.

Scanned by CamScanner

iii) We may combine two hashing functions one after another to get better results. Eg If the key is very large, first apply the folding method to reduce its size and then apply some other hashing technique.

Drawbacks:

All the above mentioned hashing functions do not preserve the order of key (ie) the hash value of the two keys are not necessarily in the same order as the keys themselves.

COLLISION-RESOLUTION TECHNIQUES

A Perfect Hash Function may be defined as a hash function which yields an address which is unique, (ie) no collision should occur. Unfortunately there is no such hash functions for a changing key set. Therefore we must go for techniques that resolve collisions which can be any one of the following two classes:

i) Open addressing

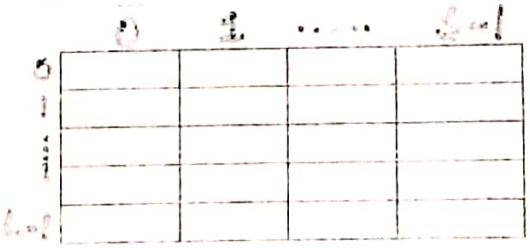
ii) Chaining

The aim of the collision resolution techniques is to find a place for colliding records somewhere else in the table.

LINEAR OPEN ADDRESSING

The simplest way to resolve collision is to start with the hash address and do the sequential search throughout the table for an empty location. The idea is to place the record in next empty location in the table. Let us suppose a group of keys are to be inserted into a hash table HT of size L, making use of the modulo arithmetic function H(K)=K mod L. Since the range of the hash table index is limited to lie between 0 to L-1, for N keys (N>I). Surely collisions occur. Therefore it is necessary to provide unique rooms for synonyms.

Let HT[0..L-1] be the given hash table. Here the L locations are called as buckets. In order to accommodate synonyms, the buckets are divided into partitions called as slots. There if a bucket has S slots means that the same bucket b can accommodate S synonyms. Thus the hash table can be redefined as HT[0..L-1,0..S-1], where the rows=buckets and columns=slots.



L = 11

S = 3

Eg: H(X)=X mod 11 and the keys are {45,98,12,55,46,89,65,88,36,21,77,34,43}.

Scanned by CamScanner

45 mod 11 (Remainder)

X	45	98	12	55	46	89	65	88	36	21	77	34	43
H(X)	1	10	1	0	2	1	10	0	3	10	0	1	10

Then the hash table will be,

	0	1	2
0	55	88	77
1	45	12	89
2	46	34	43
3	36		
4			
5			
6			
7			
8			
9			
10	98	65	21

Tackle - চেষ্টা করা

(The bucket to which the key is mapped by a hash function is known as the home bucket. To tackle overflows, we move further down, beginning from the home bucket and look for the closest slot that is empty and place the key in it. Such a method of handling overflows is known as **linear probing or linear open addressing or closed hashing.**)

Operations:

a) **Search:** Searching a key in home bucket requires a time complexity O(1).

Result of search	Reason
Successful Search	Key is found
Unsuccessful Search	Key is not found
	Search curl back to home bucket
	An empty slot is encounters in which case the search terminates

Scanned by CamScanner

```
PROCEDURE LOP-HASH-SEARCH (HT, b, S, x)
    h = h(x);
    i = h; //i is the bucket
    j = 0; //j is the slot
    while (HT[i,j] ≠ 0 and HT[i,j] ≠ x) do
        j = j+1;
        if (j > S-1) then
            j = 0
        if (i = 0) then
            {
                i = (i+1) mod b;
                if (i == h) then // Search end
                    print "Key not found";
                exit();
            }
    endwhile
    if (HT[i,j] = x) then print "key found"
    if (HT[i,j] = 0) then print "Key not found"
end LOP-HASH-SEARCH
```

i = bucket (Row)
j = slot (column)

0, 1, 2, 3

3 > S-1
3 > 2

i = (2) mod 11
i = 2

b) **Insert:** If the required key is not found, the algorithm may insert the key in the allocated slot. Therefore the statement if **(HT[i,j]=0) then print "Key not found" in the above algorithm** can be replaced as **if (HT[i,j]=0) then HT[i,j]=k.**

c) **Delete:** The delete operation on a hash table can be clumsy. When a key is deleted, the slot will become empty thereafter. Since the empty slot becomes the signal to end the search, it will bring out wrong result since the following elements in the same bucket may get unnoticed. To tackle this it is essential that the keys following the empty slot can be moved up. But moving elements is an expensive one. An alternative approach is to write a special element in the slot after deletion. This special element may act as a flag that there are some elements following it and also we can store any new ~~up-coming~~ element there. Anyhow, it is not encouraged to perform deletion unless necessary.

Limitations of Linear Open Addressing: Here the search for the synonym in the closest slot itself may degrade performance since every other slot has to be searched sequentially. Also since it can be viewed as a 2D array, the size of the hash table is reduced to bxs. If we try to increase the slots, the memory may be wasted.

clumsy → [illegible]

Scanned by CamScanner

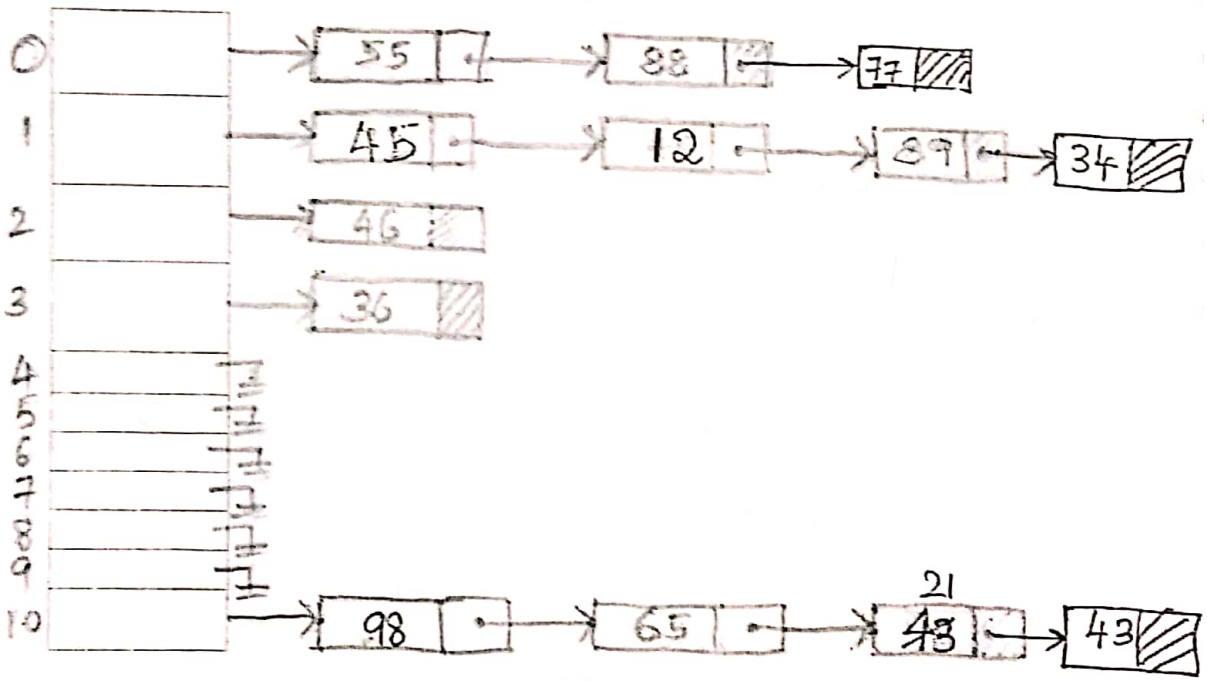
CHAINING

Here the colliding records are chained into a special overflow area which is separate from the prime area. Overflow area is implemented in the form of linked list. Along with the array, it is possible to keep only the header nodes in the array and the actual records are kept in the linked lists. The prime area contains the part of the array into which the records are initially hashed. Each record in the prime area contains a pointer field to maintain a separate linked list for each set of colliding records into the overflow area. If the array contains null at the hashed address, the linked list is empty and the required record is not there which may be inserted into the empty list. If the array is not null, we may search the list for the record. If the search succeeds, print the record else insert it as new record at the end. Even though additional area is required for pointers, the chaining is powerful than open addressing. This is also called as **open hashing or separate chaining.** The link field of the last synonym will carry the NULL value. There are no overflows since any number of synonyms can be handled at a particular index.

Eg: H(X)=X mod 11 and the keys are {45,98,12,55,46,89,65,88,36,21,77,34,43}.

X	45	98	12	55	46	89	65	88	36	21	77	34	43
H(X)	1	10	1	0	2	1	10	0	3	10	0	1	10

Then the hash table will be,



Scanned by CamScanner

Operations:

a) **Search:** The bucket corresponding to the value of H(x) is searched sequentially. IF the key is found, then the search is termed successful otherwise termed as unsuccessful. If the chain is too long, maintaining the keys in ascending or descending order will make the search somewhat easy and quick.

```
Procedure CHAIN-HASH-SEARCH (HT: b, x)
    h=H(x)
    temp=HT(h)
    While (DATA(temp)≠x and temp ≠ NULL) do
        temp=LINK (temp)
    endwhile
    if(DATA(temp)==x) then
        Print "Key found"
    if (temp==NULL) then
        Print "Key not found"
end CHAIN-HASH-SEARCH
```

b) **Insert:** To insert a key x into a hash table, we compute the hash function H(x) to determine the bucket. If the key is the first node to be added, mere call the procedure to insert a node into an empty singly linked list. But if the bucket already has some synonyms, then the new key may be added at the end of the list leaving th e list unordered. If we want to improve the performance, insert the key at its appropriate position making the list as an ordered one.

c) **Delete:** Unlike in the linear open addressing, here deletion is an effective one (ie) searches for the particular key, if found delete the node from the linked list.

PERFORMANCE ANALYSIS

Both for linear open addressing and chaining, the efficiency class analysis will be

	BEST CASE	WORST CASE	AVERAGE CASE
Linear Open Addressing	O(1)	O(n)	$u_n=1/2(1+1/(1-\alpha)^2)$ $s_n=1/2(1+1/(1-\alpha)^2)$
Chaining	O(1)	O(n)	$u_n=(1+\alpha)/2$ $s_n=1+\alpha/2$

where u_n=number of buckets searched during unsuccessful search

S_n = number of buckets searched during successful search

α = loading factor = n/b

Smaller the loading factor better will be the average case performance and hence the performance of list is better than opening address.

Scanned by CamScanner

APPENDIX

ALGORITHMS

An algorithm can be defined as a sequence of instructions each of which have a clear meaning and can be performed with a finite amount of effort in a finite length of time. It can also be defined as a collection of unambiguous instructions occurring in some specific sequence and such an algorithm should produce output for given set of input in finite amount of time.

Properties:

An algorithm has input step, assignment step, decision step, repetitive step and out put statement. The algorithm is endowed with the following properties:

1. ***Finiteness:*** Must terminate after a finite number of steps.
2. ***Definiteness:*** Steps must be precisely defined or unambiguously specified.
3. ***Generality:*** Must be generic enough t solve all problems of a particular class.
4. ***Effectiveness:*** Must be basic enough to be put down on pencil and paper (ie) should not be too complex.
5. ***Input/Output:*** Must have certain initial or precise input and output ste ps.

An algorithm does not enforce a language or mode for its expression but only demands adherence to its properties. The steps involved in the development of an algorithm are as follows:

1. Problem Statement
2. Model Formulation
3. Algorithm Design
4. Algorithm Correctness
5. Implementation
6. Algorithm Analysis
7. Program Testing
8. Documentation

EFFICIENCY OF ALGORITHMS

The performance of an algorithm can be measured on the scales of time and space.

SPACE COMPLEXITY: The space complexity can be defined as the amount of memory required by an algorithm (or) The space complexity of an algorithm is the function of space needed by the algorithm to run to complete. Consider the following three examples of algorithms to

Scanned by CamScanner

compute space complexity. Regarding space factor, the expectation is that the algorithm should assume limited memory space

EG 1:

```
Algorithm Add(a,b,c)
return (a+b+c)
```

If each a,b,c occupies one word size, the memory requirement is S(p)=3.

EG 2:

```
Algorithm Add(x,n)
sum=0.0
for i=1 to n do
        sum=sum+x[i]
return sum
```

The space requirement S(p) >=(n+3) where 3 is for n,i and sum.

EG 3:

```
Algorithm Add(x,n)
return (Add(x,n-1)+x[n])
```

The internal stack used for recursion includes space for formal parameters, actual parameters and return address. The space required by each call to function add requires 3 word (space for n values + space for return address + pointer to x[]). The length of recursion is (n+1) (ie) n times call to function and one return call. Hence S(p)=3(n+1) for the above algorithm.

TIME COMPLEXITY: Regarding the factor time, the expectation is the fastest algorithm in minimum possible time. The time complexity of an algorithm is a function of the running time of an algorithm. It is difficult to compute the time complexity in terms of physically clocked time. Some of the factors that have an impact over time are system load, number of other programs running, instruction set used, speed of underlying hardware etc.

Time complexity measurements
- Empirical or Posteriori Testing
- Theoretical or Apriori Testing

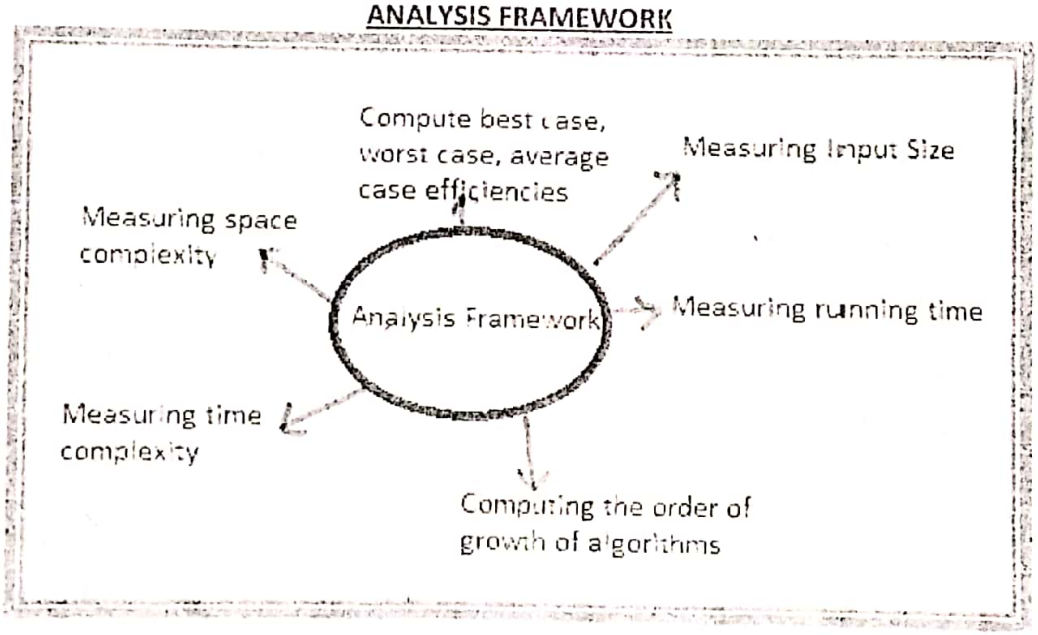
1. ***Empirical or Posteriori testing:*** This approach executes the algorithm for various instances of the problem and then compares those results; which result is yielded in least time is considered as the best. The demerit of this method is it is totally machine dependent, programming language and even on the programmer dependent.
2. ***Apriori or Theoretical testing:*** This approach calls for mathematically determining the resources (time, space) as a function of a parameter related to the instances of the

Scanned by CamScanner

problem considered. The advantage of this method is it is entirely machine, language and program independent. Time efficiency is analyzed by determining the number of repetitions of the basic operations as a function of input size. Basic operation is the operation that contributes most towards the running time of the algorithm. Hence $T(n)=C_{op}C(n)$ where T(n)=Running time, C_{op}=Execution of basic operation and C(n)=Number of times the basic operation is executed.

PROBLEM	INPUT SIZE MEASURE	BASIC OPERATION
Searching a value in a list	N elements	Key comparison
Matrix Multiplication	Two nxn matrices	Multiplication of two numbers
Graph Problems	Number of vertices and edges	Visiting a vertex

ANALYSIS FRAMEWORK



Operation Count:

The abstract operations count is used to analyze the algorithm. Following are a few examples of operations count. Consider only the major looping constructs and not the complete program.

1. Print the first element of the array. This statement is executed once irrespective of the size of the input. Hence the abstract operation count is a constant T(n)=1.
2. Print the elements of an array. The abstract operation count T(n)=n.

Scanned by CamScanner

3. Print the element of an nXn array. The abstract operation count $T(n)=n^2$. The inner loop runs for n iterations and this is repeated for every n iteration of the outer loop.
4. Multiplication of two matrices each of size nXn. Abstract operation count in innermost loop is n. Abstract operation count in middle loop is $n(1+n+1)=n^2+2n$. Abstract operation count in outer loop is $n*(n^2+2n) =n^3+2n^2$. Hence $T(n)=n^3+2n^2$.

```
for (i=0;i<n;i++)
    { for (j=0;j<n;j++)
            {
                sum=0;
                for(k=0;k<n;k++)
                    sum=sum+mat1[i][k]*mat2[k][j];
            }
        result[i][j]=sum;
        }
```

5.
```
for (i=0;i<n;i=i*2)
    for (j=0;j<n;j++)
        print data[i][j];
```

Abstract count in inner loop is n and in outer loop is log n. Hence T(n)=nlogn.

6. To add two square matrices, we add their corresponding elements (ie) we add the first of first and second matrix, second element of first and second matrix etc. The pseudo code to add 2 matrices is:

```
Algorithm addmatrix(matrix1,matrix2,matrix3,size)
{
    loop (not end of row)
        loop (not end of column)
            add matrix1 and matrix2 cells

        store sum in matrix3

        endloop
    endloop
}
end addmatrix.
```

For each element in a row, we add all of the elements in a column-classic quadratic loop. Therefore the efficiency of algorithm is $T(n)=O(size^2)$ or $O(n^2)$.

Hence the summary of analysis framework is:

a) Both time and space efficiencies are measured as functions of the algorithm's input size.

Scanned by CamScanner

b) Time efficiency is measured by counting the number of times the algorithms' basic operation is executed and space efficiency is measured by counting the number of extra memory units consumed by the algorithm.

c) The efficiencies of some algorithms may differ significantly for inputs of the same size. For such algorithms, we need to distinguish between worst-case, best-case and average-case efficiencies.

d) The framework's primary interest lies in the order of growth of the algorithms' running time (extra memory units consumed) as its input's size goes to infinity.

ASYMPTOTIC NOTATIONS

The time complexity of an algorithm indicates the runtime of the algorithm as a function of the size of the input. The exact running time of the algorithms depend on various factors like programming language used, configuration of computer etc. Asymptotic analysis is performed to measure the order of growth rates of the algorithm. This is to study how the execution time may increase or decrease according to the input size. In asymptotic analysis, only the largest term in the operations count is considered and the constant multipliers are ignored.

Eg: $2n^3+5n^2+1$ could behave asymptotically as n^3 since n^3 takes over n^2.

Reasons to determine step counts:

- To predict the growth in runtime as the instance characteristics increase.
- To compare the time complexities of two programs that performs the same task.

1. **Big-Oh Notation (O):** A function f(n), a non-negative functions for n<=0 is in O(g(n)), if f(n)<=C*g(n), for every n>N, where C and N are constants. Big-O gives an upper bound and not a tight bound. Hence Big-O notation cannot be used to generalize the algorithms since O(1) can be $O(n^2)$ or O(2n). We can give the longest time taken by the algorithm to complete using this Big-Oh notation. It is used to define the worst-case running time of an algorithm.

$$\lim_{n\to\alpha} \left[\frac{f(n)}{g(n)}\right] < \alpha$$

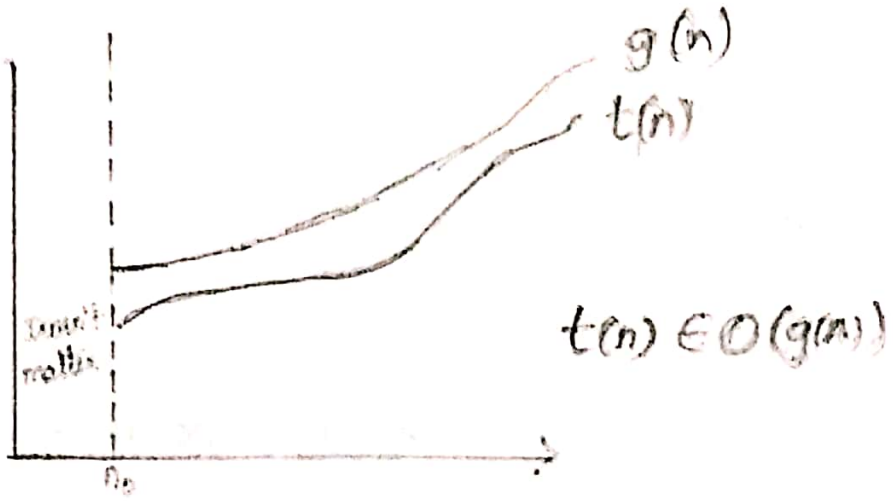
Eg: Consider f(n)=2n+2 and $g(n)=n^2$. We have to find some constant c so that f(n)<=c*g(n).

For n=1, f(n)=4 and g(n)=1. Hence f(n)>g(n)

For n=2, f(n)=6 and g(n)=4. Hence f(n)>g(n)

For n=3, f(n)=8 and g(n)=9. Hence f(n)<g(n). So we can conclude that for n>2, we obtain f(n)<g(n). Thus always upper bound of existing time is obtained by big-oh notation.

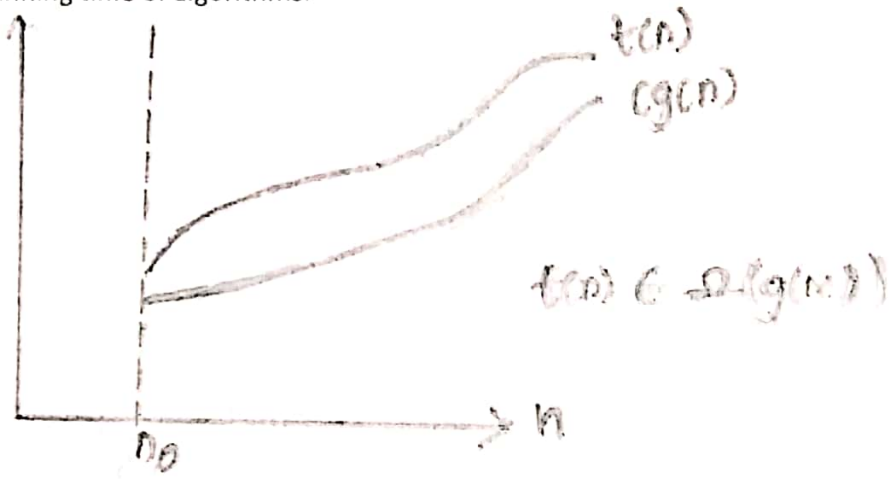
Scanned by CamScanner



2. **Omega notation (Ω):** The notation f(n), a non-negative function is in Ω(g(n)), if f(n)>=C*g(n), for every n>N and C and N are constants (ie)

$$\lim_{n\to\alpha}\left[\frac{f(n)}{g(n)}\right] > 0$$

This notation is used to represent the lower bound of algorithm's running time, so that we can denote the shortest amount of time taken by algorithm. The complexity of printing all the elements in the array is Ω(N). Ω Notation thus gives only a lower bound. Printing of all elements in an array can also be said to at least Ω(1). Hence a tight lower bound is required and it describes the best case running time of algorithms.



EG: $f(n)=2n^2+5$ and $g(n)=7n$

For n=0, f(n)=5 and g(n)=0. Hence f(n)>g(n).

For n=1, f(n)=7 and g(n)=7. Hence f(n)=g(n).

For n=2, f(n)=13 and g(n)=14. Hence f(n)<g(n).

For n=3, f(n)=23 and g(n)=21. Hence f(n)>g(n).

Thus for n>3 we get f(n)>c*g(n) and it can be represented as $2n^2+5\epsilon\ \Omega(n)$.

Scanned by CamScanner

3. **Theta notation (θ):** A function f(n) is in θ(g(n)), if there are constants C1, C2 and N such that,

$$C1*g(n)<=f(n)<=C2*g(n)$$

By this method we can find the running time between the upper bound and lower bound. Eg: Print all elements in an array is θ(n). Let f(n)=2n+8 and g(n)=5n; f(n)=2n+8 and g(n)=7n. (ie) 5n<2n+8<7n (for n>=2). Here c1=5 and c2=7 with n_0=2. The theta notation is more precise with both big-oh and omega notation.

4. **Little-oh notation (o):** The little-oh notation describes a strict upper bound on the asymptotic growth rate of the function f. Informally, f(n) is little-on of g(n) iff f(n) is asymptotically smaller than g(n).
(ie) f(n)=O(g(n)) and f(n) =Ω(g(n))

Basic Efficiency Classes:

Efficiency class	Order of growth	Description	Example
CONSTANT	1	As input size grows the we get larger running time	Scanning array elements
LOGARITHMIC	logn	Here the algorithm does not consider all its input rather the problem is divided into smaller parts on each iteration	Binary search
LINEAR	n	The running time of algorithm depends on the input size n	Sequential search
QUADRATIC	n^2	Here algorithm includes nested loops	Scanning matrix elements
CUBIC	n^3	Algorithm with three nested loops	Matrix multiplication
EXPONENTIAL	2^n	When the algorithm has very faster rate of growth then this types of efficiency occurs	Generating all subsets of n elements
FACTORIAL	n!	When an algorithm is executing all permutations	Generating all permutations

Scanned by CamScanner

We have learnt that by means of constant multiple we can get different order of growth. But the categorization of growth is not simply restricted within big-oh, theta or omega notations. We have different efficiency classes and each class possesses certain characteristics as listed above.

Analysis is what we do before coding. Analysis is important for the following reasons:

1. Analysis is more reliable than experimentation or testing (ie) analysis gives the performance of the algorithm for all cases of inputs, whereas testing is done only on specific cases of input.
2. Analysis helps to select better algorithms (ie) when there are different algorithms for the same tasks, analysis helps to pick out the most efficient algorithm.
3. Analysis predicts performance (ie) analysis can be used to predict the run time of the algorithms. If the algorithm is analyzed to be very slow, then it need not be implemented at all.
4. Analysis identifies scope of improvement of algorithm (ie) analysis of an algorithm can find out which portion of the algorithm is faster and which is slower. The slower part may then be modified, if possible, to increase execution time.

EG: To sequentially search for the first occurring even number in the given list of numbers.
INPUT 1: -1 3 5 7 -5 7 11 -13 17 71 21 9 3 1 5 -23 -29 33 35 37 40
INPUT 2: 6 17 71 12 9 3 1 5 -23 3 64 7 -5 7 11 33 35 37 -3 -7 11
INPUT 3: 71 21 9 3 1 5 -23 3 11 33 36 37 -3 -7 11 -5 7 11 -13 17 22

Let us determine the efficiency of the algorithm for the input instances in terms of number of comparisons done for displaying the first occurring even number.

INPUT 1: The first even number "40" is at the end of the list. Hence 21 comparisons are to be made which is approximately equal to the size of the list. So the input pattern is said to be worst since it takes the longest possible time to complete.
INPUT 2: The first even number "6" occurs as the first element in the list which requires only one comparison. So it is the best possible case as it produces the quickest execution of the program.
INPUT 3: Input 3 is an example of average case as the first even number "36" occurs almost in the middle of the list, nor the worst case nor the best case. Analyzing the average case of the algorithm is harder and mathematically involved when compared to worst and best case.

Scanned by CamScanner

Generalizing, the time complexity of the algorithm, O(1) is the best case and O(n) is the worst case where n=input size. Hence it is implicit that the running time of the algorithms is not just depending on the size of the input but also on the nature of input.

i) So the input instances for which the algorithm takes the maximum amount of time is called the **worst case** and the time complexity in such a case is referred to as the **worst case time complexity**.

ii) The input instances for which the algorithm takes the minimum possible time is called the **best case** and the time complexity in such a case is referred to as the **best case time complexity**.

iii) All other input instances which are neither of the two are categorized as the **average cases** and the time complexity of the algorithm in such cases is referred to as the **average case complexity**.

Scanned by CamScanner