Name - Mukesh Singh
Class - B.Ed ICT Sec - 3rd Semester
Roll.No -
Subject - Data Structure and Algorithm
Subject Teacher -
College - Sukuna Multiple Campus

Data Structure & Algorithm Notes

ICT 3rd Semester

by

Mukesh Singh

ICT 5th batch, 2073

Sukuna Multiple Campus
Sundarharaincha, Morang

Unit-1

Introduction to Data Structures & Algorithm

Page No. 1
Date 07/11/2106

1.1. Data types, Data Structure and Abstract data type

Data types:

A data type is defined as a set of values that a variable can store along with the set of operations that can be performed on the variable.

Every variable and constant that we are using in our program, must be declared before it is used. There are two categories of data types:

ⓐ Fundamental or basic data type (int, char)
ⓑ Derived data type (array, structure, string, union)

Data Structure

In computer science, data structure is particular and in a computer so that it can be used efficiently. The word data means content and structure refers to the place where data of content are stored.

- Data structure is a way of organizing all data items and establishing relationship among those data items.
- Data structures are the building blocks of a program.

Two types of data structure:

ⓐ Static data structure ⓑ Dynamic data structure

@ Static data structure:
Whose capacity is fixed at creation.
for example: array.

ⓑ Dynamic data structure:
Whose capacity is variable, so it can expand or contract at any time. for example: linked list, binary tree, etc.

Abstract data types (ADTs)
An abstract data type is a specification of set of values and set of operations that can be performed on the data. It is a useful tool for specifying the logical properties of data type. The term ADT refers to the basic mathematical concept that defines data type.
When we use abstract data types, our program is divided into two pieces:
① Application
The part that uses the abstract data type.

② Implementation:
The part that ~~uses~~ implements the abstract data type.
These two pieces are completely independent. It should be possible to take the implementation developed for one application and use it for a completely different application with no changes.

1.2. Dynamic memory allocation in C

Consider the following array declaration:

```
int age[22];
```

It reserves memory space for 22 integer numbers. Sometimes problem arise with the declaration. If the users needs to work on more than 20 values, the allocated memory will not be sufficient. Besides, if the users need to work on less than 20 values, the allocated memory will be unused. This type of allocating memory at compile time is called static memory allocation.

To overcome these limitation, the memory for array is allocated at the run time. The first address of the allocated memory assign to the pointer variable which can subsequently used as array. The memory allocation at runtime is called dynamic memory allocation.

1.3 Introduction to Algorithms

Algorithm is one of the most basic programming tools used for solving problems. It is defined as finite sequence of instructions for solving a problem. It consists of stepwise list of English statements making sequential procedure. The number of instructions should be minimized to increase the speed of algorithm.

Properties of Algorithm:

→ The number of instructions should be finite.
→ The problem should be divided into small modules
→ The steps written in algorithm should be simple and unambiguous.
→ It should have an input, process and desire output after executing algorithm.
→ It should not depend on any particular language or Computer.

1.4 Asymptotic notations and ~~functions~~ Common functions:

Asymptotic Notations

It is the simplest and easiest way of describing the running time of an algorithm. It represents the efficiency and performance of an algorithm in a systematic and meaningful manner. It describes the time Complexity in terms of three common measures :- best case (fastest possible), worst case (slowest possible) and average case (average possible time).

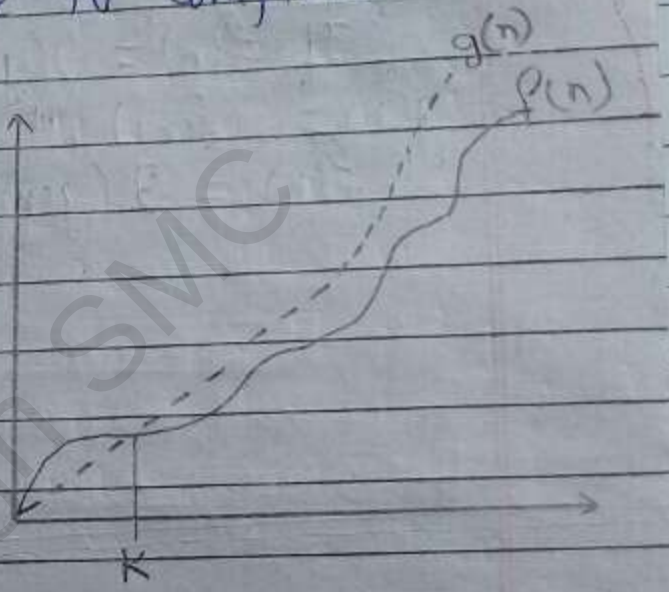
Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm:

① Big Oh Notation (O):

This notation, $O(n)$ is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time that an algorithm can possibly take to complete.

Definition

→ $f(n) = O(g(n))$ if there exists constants n_0 and c such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

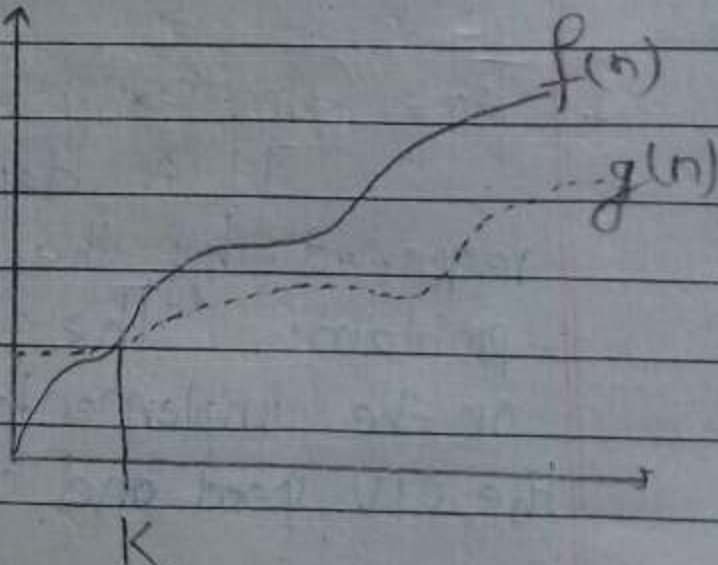


② Omega Notation (Ω)

This notation, $\Omega(n)$ is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

Definition

$f(n) = \Omega(g(n))$ if there exists constants n_0 & c such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.

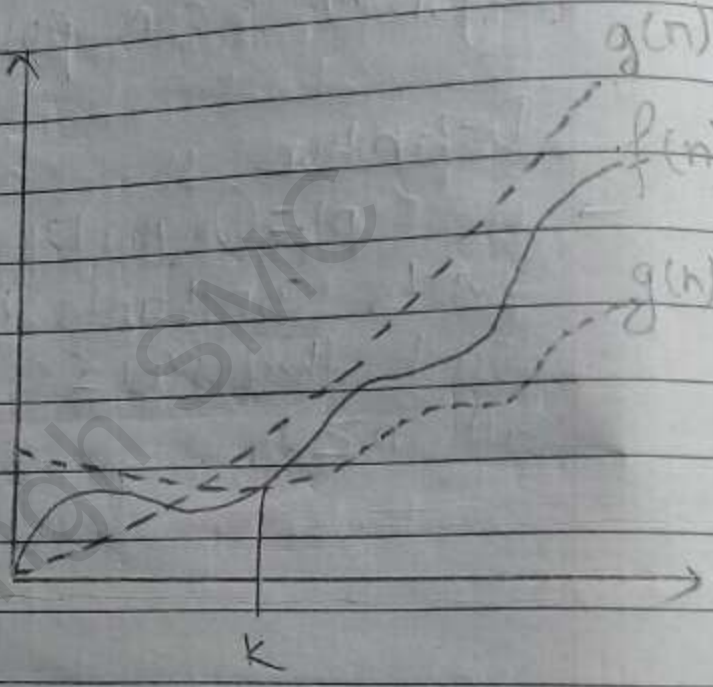


③ Theta Notation (Θ)

This notation, $\Theta(n)$ is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows:

Definition

If $f(n) = O(g(n))$ and $g(n) = O(f(n))$, then $f(n) = \Theta(g(n))$.



Algorithm Complexity:

The complexity of an algorithm $f(n)$ gives the running time and/or the storage space required by the algorithm in terms of 'n' as the size of input data.

1. Time Complexity:

It is defined as the amount of time required by the algorithm to solve a particular problem. The exact time will be depending on the implementation of algorithm, programming languages, the CPU speed and other hardware characteristics.

Some reasons for studying time complexity are:

i) We may be interested in knowing advance that whether an algorithm/program will provide a satisfactory real time response or not.

ii) There may be several possible solutions with different time requirements.

2. Space Complexity:

It is defined as the amount of memory space required by the algorithm to solve a particular problem. The space is needed by following components:

* Instruction Space
* Data Space
* Environment Stack Space

The Space required by an algorithm is equal to the sum of following two components:

(a) A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example- simple variables and constants used, program size etc.

(b) A variable part is a space required by variables, whose size depends on the size of problems. For example-

dynamic memory allocation, recursion stack space etc.

Thus, Space Complexity $S(P)$ of any algorithm P is: $S(P) = C + S(I)$.
Where C is fixed part and $S(I)$ is the variable part of the algorithm, which depends on instance characteristics I.

Some reasons for studying space complexity:

i) If a program is to run on multiuser system, it may be required to specify the amount of memory to be allocated to the problem.

ii) We may be interested to know in advance that whether sufficient memory is available to run the program or not.

iii) There may be several possible solution with different space requirement.

iv) It can be used to estimate the size of the largest problem that a program can solve.

Classification or types of data structure:

1. Primitive / Fundamental / Natural Data Structure:
The data structure which can be directly operated by machine level instructions are called primitive data structure.
Examples :- Integer, float, character, pointer etc
In C language, some primitive data structures are:
@ int (b) float (c) char

2. Non-primitive / Derived / Artificial Data Structure:
The data structure derived from the primitive data structure are called non-primitive data structures. The non-primitive data structure emphases on structure of a group of homogeneous (same type) or heterogeneous (different types) data items.
Example :- Arrays, linked list, files etc
Types of non-primitive data structure:

@ Linear data structure:
In linear data structure, the data is stored in the memory in linear or sequential order i.e one after the other. Example - Arrays, stacks, queues, linked lists etc

(b) Non-linear data structure:
In non-linear data structures, the data is

stored in different or non-sequential order i.e random order.
Examples - trees, graph etc

Description of some non-primitive data structure:

① Array:
An array is defined as a set of finite number of homogeneous element in which data items are stored in adjacent cells in memory.

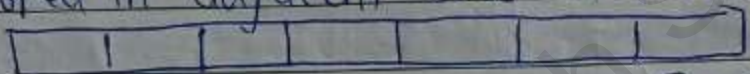


Fig: An array 'A' containing 'n' elements

② Linked list:
Linked list is a linear collection of data elements or nodes where each nodes are divided into two parts, the first part contains the information of elements and second part called linked filled contains the address of the next node in the list.

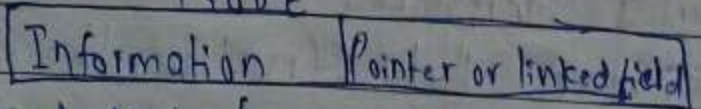


Fig: Linked list showing two parts of a node

③ Stack:
Stack is a linear data structure in which data and deleted at one end called the top of stack i.e. data is stored

and retrieved in Last in first Out (LIFO) order.

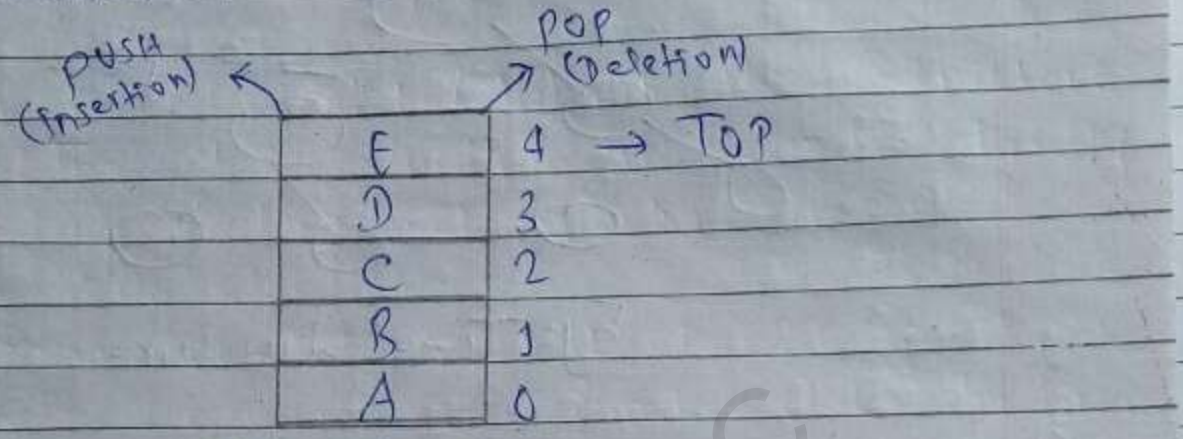
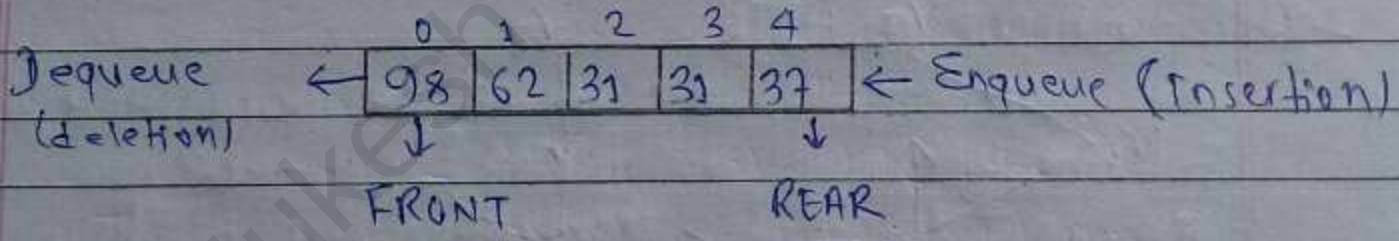


Fig: Stack

(4) Queues:

Queue is a linear data structure in which data is inserted from REAR End and deleted from FRONT End i.e. data is stored and retrieved in First in first Out (FIFO) order.



(5) Tree:

A tree can be defined as finite set of data items (nodes). Tree is a non-linear data structure in which data items are arranged or stored in a sorted order. It is a non-empty vertices and edges with some restrictions. A vertex is a node, edge is a connection between two nodes.

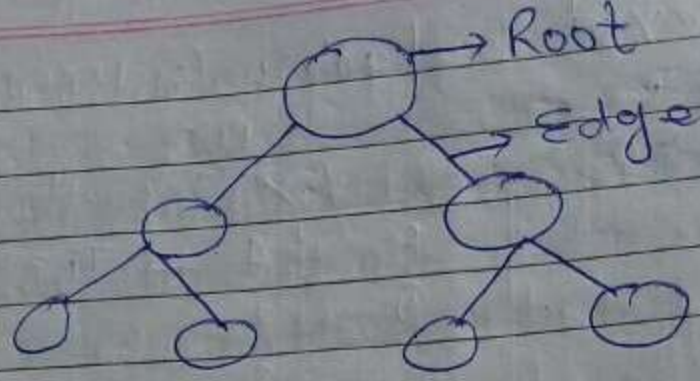


Fig: Tree structure

⑥ Graph:

A graph G(V, E) is a set of vertices 'V' and set of edges 'E'. An edge connects a pair of vertices. Vertices on the graph are shown as points or circles and edges are drawn as arcs or line segments.

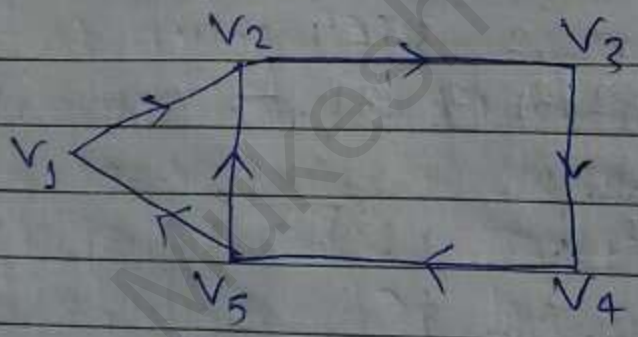


Fig 1: Directed Graph

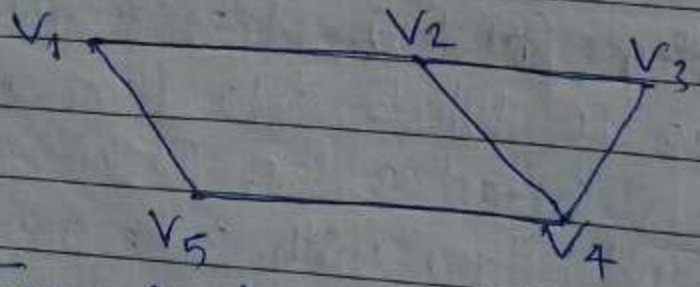


Figure 2: Undirected Graph

Unit-2
Stacks

Page No. 13
Date

2.1 Definition:

A stack is an abstract data type that is a collection of items into which items may be inserted and from which items may be deleted from one end, called the TOP of stack. The insertion and deletion in the stack is done from the TOP of the stack.

OR, A stack is a collection of elements, with two principle operations:

→ Push, which adds an element to the collection, and

→ Pop, which removes ~~an element~~ the most recently added element that was not yet removed.

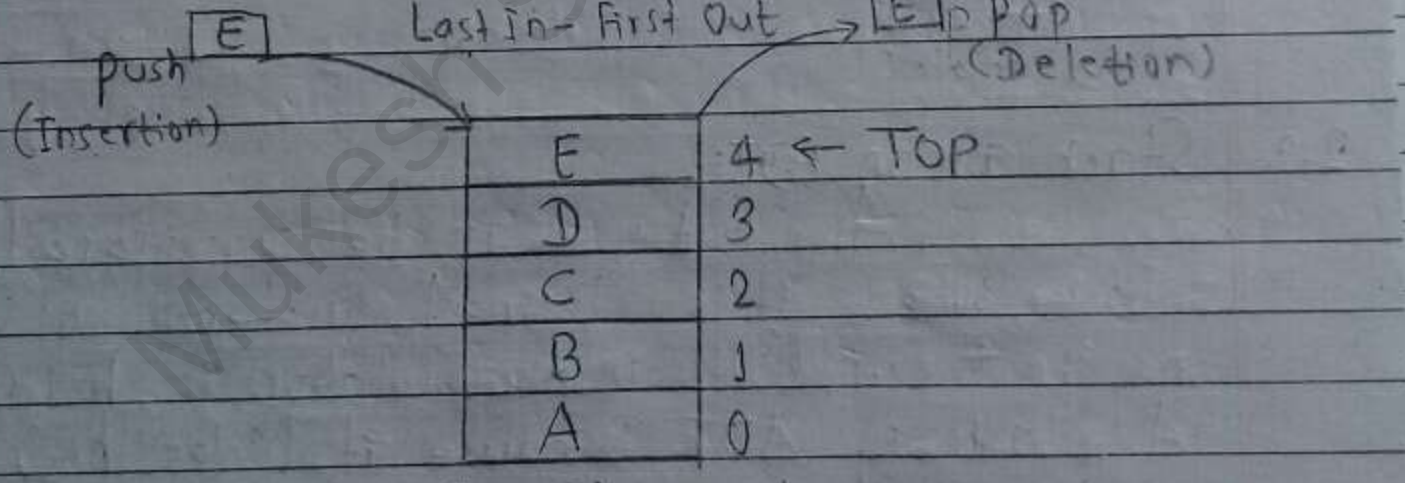


Fig: Stack

	Stacksize-1
	Stacksize-2
...	
...	
98	4
62	3
34	2
31	1
37	0

Fig(1)
Stack with 5 elements
Top=4, Count=5

12	Stacksize-1 → Top
34	Stacksize-2
...	
...	
98	4
62	3
37	2
61	1
87	0

Fig(2)
Full Stack
Top=Stacksize-1
Count=Stack size

	Stacksize-1
	Stacksize-2
...	
...	
98	4
15	3
35	2
51	1
17	0

Fig(3)
Empty Stack
Top=-1, Count=0

2.2 Stack as an ADT:

A data can be considered as stack when it is defined in terms of operations in it and its implementation is hidden. A stack is ADT because it hides how it is implemented like using array or linked list. However, if organized data for efficient management and retrieval, so it is a data structure also.

A stack of element of type T in a finite sequence of elements of T together with the following operation:

1. Create empty stack (S):
Create or make stack 'S' as an empty stack.

2. PUSH (S, x):
Insert 'x' at the one end of the stack 'S' i.e. TOP.

3. TOP (S):
If stack 'S' is not empty, then retrieve the element at its TOP.

4. POP (S):
If stack (S) is not empty, then delete the element at its TOP.

5. Is Full (S):
Determine if 'S' is full or not. Return TRUE if 'S' is full ~~Return FALSE if~~ otherwise, Return FALSE.

6. Is Empty (S):
Determine if 'S' is empty or not. Return TRUE if 'S' is empty otherwise, return FALSE.

2.3. Stack operation:

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations:

1. PUSH Operation
2. POP Operation

1 PUSH Operation:

The process of putting a new data element to the top of the stack is known as PUSH operation. After every PUSH operation, the Top is incremented by 1. When the array is full (if static implementation is applied), the status of the stack is full and this condition is called overflow. In such a case, no further element can be added.

If TOP = MAXSIZE-1, then the stack is overflow.

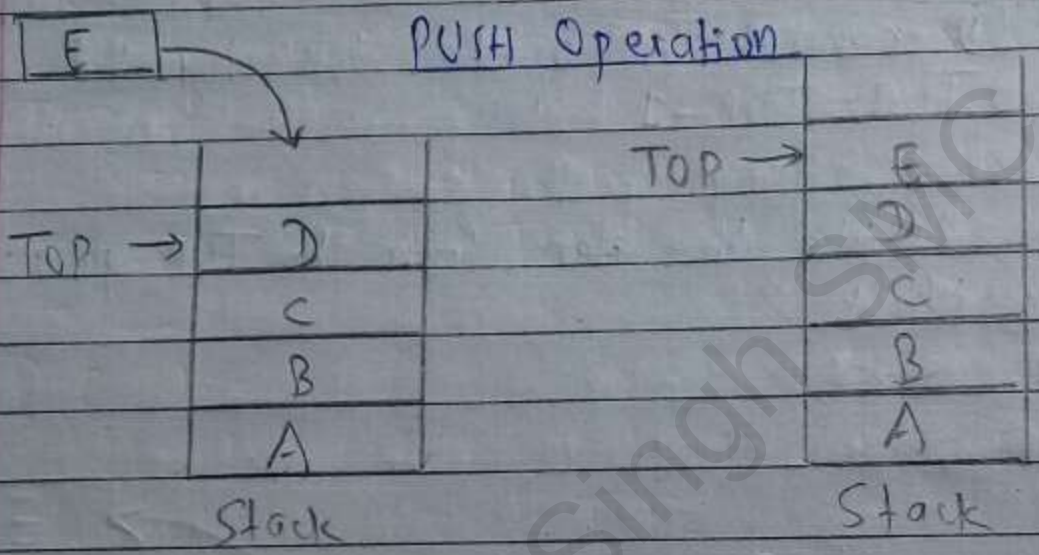
Algorithm for PUSH Operation:

Step 1: [Check that stack is full or not]
if TOP = MAXSIZE-1 then
print "stack is full"

Step 2: [Increment TOP of the stack by 1]
TOP = TOP + 1

Step 3: [Input the element to the stack]
STACK[TOP] = item

Step 4: Exit



2. POP Operation:

The process of removing an element from the TOP of the stack is called as POP operation. After every POP operation, the TOP of the stack is decremented by 1. If there is no element in the stack, it is called as empty stack or stack underflow. In such a case, POP operation cannot be applied. If TOP = -1, then stack is underflow.

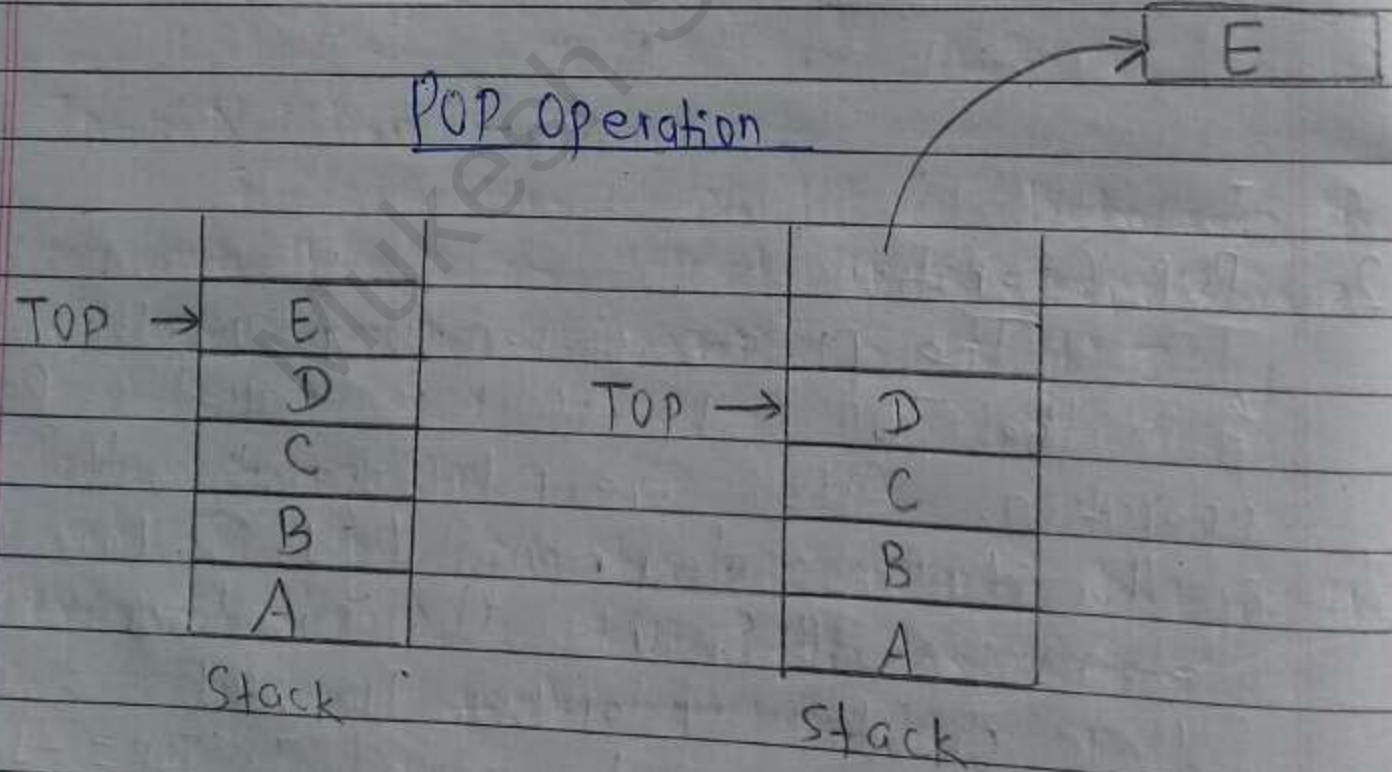
Algorithm for POP operation:

Step 1: [Check that stack is EMPTY or not]
if TOP = -1, then
print "stack is empty".

Step 2: [Decrement TOP of the stack by 1]
TOP = TOP - 1

Step 3: Delete the item from the TOP position of the stack.

Step 4: Exit



2.4. Stack Application : Conversion from infix to postfix / prefix expression, Evaluation of postfix / prefix expressions

① String Reversal:

We can reverse a string by pushing each character of the string on the stack. After the whole string is pushed on the stack, we can start popping the characters from the stack and get the reversal string.

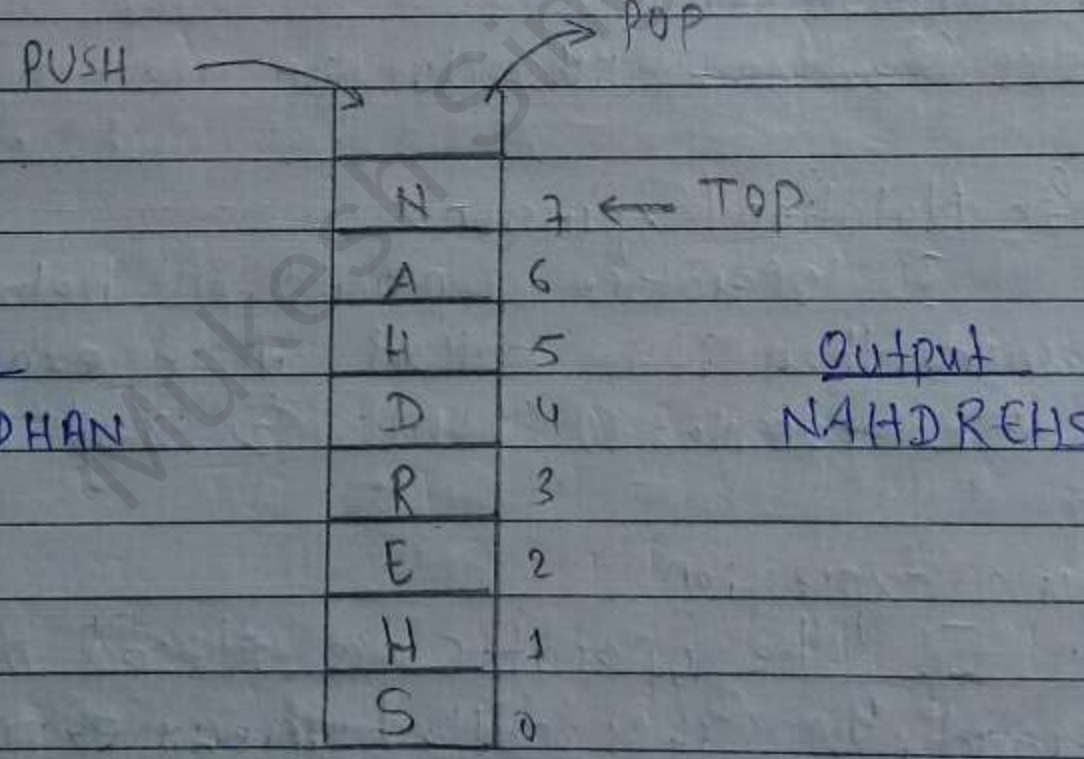


Fig: Reversing a string:

② Arithmetic Expression:

The way to write arithmetic expression is known as a notation. An arithmetic expression is a combination of various constants, symbols (variables) and operators.

Based on the placement of operators with the operands, an arithmetic expression can be written in three different but equivalent notations i.e without changing the output of an expression. These notations are:

ⓐ Infix Notation
ⓑ Prefix (Polish) Notation
ⓒ Postfix (Reverse-polish) Notation

a) Infix Notation / Expression

If operator is placed in between the operands, then it is called infix expression.

for example: ① (a+b) ② (x-y)*z.

b) Prefix Expression.

If the operator is placed before the operand, then it is called prefix expression.

for eg: +ab, +cd/e

c) Postfix Notation / Expression

If the operator is placed after the operand, it is called postfix expression or reverse polish notation.

for eg: ab+, cd+e-

Few examples of all three expression:

S.N	Infix	Prefix	Postfix
1.	a+b	+ab	ab+
2.	(a+b)*c	*+abc	ab+c*
3.	a*(b+c)	*a+bc	abc+*
4.	a/b+c/d	~~+/ab/cd~~	ab/cd/+
5.	(a+b)*(c+d)	*+ab+cd	ab+cd+*
6.	((a+b)*c)-d	-*+abcd	ab+c*d-

To parse an arithmetic operation, we need to take care of operator procedure and associativity also.

Precedence:

When an operand is in between two different operators, which operators will take the operand first, is decided by the precedence of an operator over others. For example:

$a+b*c \Rightarrow a+(b*c)$.

As multiplication operation has precedence over addition, b*c will be evaluated first. A table of operator precedence is provided later.

Associativity:

Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression a+b-c, both + and - have the same precedence then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both + and - are left associative so the expression will be evaluated as (a+b)-c.

Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest):

S.No	Operator	Precedence	Associativity
1.	Exponentiation ^	Highest	Right Associative
2.	Multiplication(*) & Division (/)	Second Highest	Left Associative
3.	Addition(+) & Subtraction(-)	Lowest	Left Associative

Rules for infix to postfix Conversion:

1. Parenthesis ( ), the expression starting from left to right.
2. During parenthesis, the expression, the operands associated with operator having higher precedence are first parenthesized.
3.

4. Once the expression is converted to postfix form, remove the parenthesis.

Convert the following infix operation to postfix operation.

1. a+b-c
   = a+b-c = infix
   = (ab+) -c
   = (ab+c)-
   = ab+c- postfix

2. a*b+c/d
   = a*b+c/d = infix
   = (ab*) + (cd/)
   = (ab*) + (cd/)
   = ab*cd/+

(3) a*b+c - infix
= ab*+c
= ab*c+ → postfix

(4) a+b/c-d
= a+(bc/)-d
= abc/+d
= (ab+)/(cd-)
= ab+cd-/ → postfix

(5) (a+b)*c/d
= (ab+)*c/d
= ab+c/d*
= ab+c/d*/ → postfix

(7) (a+b)*c/d+e↑f/g - infix
= (ab+)*c/d+e↑f/g
= (ab+)*c/d+(ef↑)/g
= (ab+c*)d/+(ef↑)/g
= (ab+c*d/)+(ef↑g/)
= (ab+c*d/)(ef↑g/)+
= ab+c*d/ef↑g/+ → postfix

(8) a+(((b+c)+(d+e)*f))/g
= a+(((bc+)+(de+)*f))/g
= a+(((bc+)+(de+)(f*)))/g
= a+((bc+)de+(f*)+)g/
= abc+de+f*+g/+ → postfix

(9) $((a+b)*c-(d-e)\wedge(f+g)$
$=(ab+)*c-(de-)\wedge(fg+)$ - infix
$=(ab+)*c-(de-fg+\wedge)$
$=(ab+c*)-(de-fg\wedge)$
$=ab+c*de-fg\wedge-$ → postfix

(10) $a+(b*c-(d/e\wedge f)*g)*h$
$=a+(b*c-(d/ef\wedge)*g/*h)$
$=a+(b*c-(def\wedge/)*g)*h)$
$=a+(b*c-(def\wedge/g*h*)$
$=a+((bc*)-(def\wedge/g*h*))$
$=abc*def\wedge/g*h*-+$
$=abc*def\wedge/g*-h*+$ postfix.

Rules for infix to prefix conversion:

① Parenthesize() the expression starting from left to right.

②

③ The sub-expression (part of expression) which has been converted into prefix is to be treated as single operand.

④ Once the expression is converted to prefix form,

remove the parenthesis '()'.

Convert the following infix expression to prefix expression:

① a*b+c
= a*b+c → Infix
= (*ab)+c
= (+*abc) → prefix

② a/b^c+d
= a/b^c+d → infix
= a/b(^bc)+d
= (/a^bc)+d
= (+/a^bcd)
= +/a^bcd → prefix

③ (a-b/c)*(d*e-f)
= (a-b/c)*(d*e-f) → Infix
= (a-(/bc))*((*de)-f)
= (-a/bc)*(-*def)
= (*-a/bc-*def)
= *-a/bc-*def → Prefix

④ (a*b+(c/d))-f
= (a*b+(c/d))-f → Infix
= (a*b+(/cd))-f
= ((*ab)+(/cd))-f

= (+*ab/cd)-f
= (-+*ab/cdf)
= -+*ab/cdf → Prefix

(5) a/b^c-d
= a/b^c-d → infix
= a/(^bc)-d
= (/a^bc)-d
= (-/a^bcd)
= -/a^bcd → prefix

(6) (a+b)+c
= (a+b)+c → infix
= (+ab)+c
= (++abc)
= ++abc → Prefix

(7) a+(b*c)
= a+(b*c) → infix
= a+(*bc)
= (+a*bc)
= +a*bc → prefix

(8) (a-b)/(c+d)
= (a-b)/(c+d) → infix
= (-ab)/(+cd)
= (/-ab+cd)
= /-ab+cd → Prefix

(9) $((a+b)*c-(d-e)\uparrow(f+g))$

$= ((a+b)*c-(d-e)\uparrow(f+g))$ → Infix

$= ((+ab)*c-(-de)\uparrow(+fg))$

$= ((+ab)*c-(\uparrow-de+fg))$

$= ((*+abc)-(\uparrow-de+fg))$

$= (-*+abc\uparrow-de+fg)$

$= -*+abc\uparrow-de+fg$ → Prefix

(10) $p+(q*r)/(s*t)$

$= p+(q*r)/(s*t)$ → Infix

$= p+(*qr)/(*st)$

$= p+/(*qr)(*st)$

$= +p/*qr*st$ → Prefix

Evaluation of Postfix Expression:

① Declare all necessary variables. Initialize an empty stack, 'RESULTSTACK'.

② Read a character in postfix expression. ~~If a character is an operand~~

If a character is an operand
PUSH it into a stack
else
Pop two operands from 'RESULT STACK'
and
Store in operand 1 'OP1' and operand 2 'OP2'
Evaluate applying the expression
OP1 operator OP2.

③ Repeat step 2 until the end of postfix expression.

④ Pop the element in 'RESULT STACK'.

⑤ Stop.

Q. Evaluate the following postfix expression

① 3 4 5 * +
= 3 20 +
= 23

② 6 2 + 5 9 * +
= 8 45 +
= 53

③ 6 2 3 + - 3 8 2 / + * 2 ^ 3 +
= 6 5 - 3 4 + * 2 ^ 3 +
= 1 7 * 2 ^ 3 +
= 7 2 ^ 3 +
= 49 3 +
= 52

④ 7 2 - 1 3 + /
= 5 4 /
= 5/4.

⑤ 7 5 3 2 ^ * 9 2 2 ^ - / + 6 4 * +
= 7 5 9 * 9 4 - / + 24 +
= 7 45 5 / + 24 +
= 7 9 + 24 +
= 16 24 +
= 40

Algorithm to evaluate a postfix expression by stack method.

01/09

Let, 'P' is an expression written in postfix notation, 'VSTACK' is a stack to hold operands.

① Scan 'P' from left to right.

② If scanned character is operand, PUSH it on 'VSTACK'.

③ If scanned character is operator, then
ⓐ POP the top two elements from 'VSTACK' where 'A' is the top element and 'B' is the next top element.

ⓑ Evaluate B <operator> A
ⓒ PUSH the result into 'VSTACK'.

④ Repeat step 1 to 3 until the end of 'P'.
⑤ Stop.

~~Q. Evaluat~~

Q. Evaluate the following postfix expression by stack method:

① 6,2,3,+,-,3,8,2,1,+,*,2,1,3,+
② 1,2,3,+,*,3,2,1,-,+,*
③ 6,5,3,+,*,12,3,/,-
④ 6,3,+,5,*,2,3,+,+

① 6, 2, 3, +, −, 3, 8, 2, /, +, *, 2, 1, 3, +

S.N.	Symbol Scanned	Operand 1	Operand 2	Value	Stack
1.	6				6
2.	2				6, 2
3.	3				6, 2, 3
4.	+	2	3	5	6, 5
5.	−	6	5	1	1
6.	3				1, 3
7.	8				1, 3, 8,
8.	2				1, 3, 8, 2
9.	/	8	2	4	1, 3, 4
10.	+	3	4	7	1, 7
11.	*	1	7	7	7
12.	2				7, 2
13	1	7	2	49	49
14.	3				49, 3
15	+	49	3	52	52

(2) 1,2,3,+,*,3,2,1,-,+,*

S.N.	Symbol Scanned	Operand 1	Operand 2	Value	Stack
1.	1				1
2.	2				1,2
3.	3				1,2,3
4.	+	2	3	5	1,5
5.	*	1	5	5	5
6.	3				5,3
7.	2				5,3,2
8.	1				5,3,2,1
9.	-	2	1	1	5,3,1
10.	+	3	1	4	5,4
11.	*	5	4	20	20

(3) 6,5,3,+,*,12,3,/,-

S.N.	Symbol Scanned	Operand 1	Operand 2	Value	Stack
1.	6				6
2.	5				6,5
3.	3				6,5,3
4.	+	5	3	8	6,8
5.	*	6	8	48	48
6.	12				48,12
7.	3				48,12,3
8.	/	12	3	4	48,4
9.	-	48	4	44	44

(9) 6, 3, +, 5, *, 2, 3, +, +

S.N.	Symbol Scanned	Operand 1	Operand 2	Value	Stack
1.	6				6
2.	3				6, 3
3.	+	6	3	9	9
4.	5				9, 5
5.	*	9	5	45	45
6.	2				45, 2
7.	3				45, 2, 3
8.	+	2	3	5	45, 5
9.	+	45	5	50	50

(5) 7, 5, 3, 2, 1, *, 9, 2, 2, 1, -, 1, +, 6, 4, *, +.

S.N.	Symbol Scanned	Operand 1	Operand 2	Value	Stack
1.	7				7
2.	5				7, 5
3.	3				7, 5, 3
4.	2				7, 5, 3, 2
5.	1	3	2	9	7, 5, 9
6.	*	5	9	45	7, 45
7.	9				7, 45, 9
8.	2				7, 45, 9, 2
9.	2				7, 45, 9, 2, 2
10.	1	2	2	4	7, 45, 9, 4
11.	-	9	4	5	7, 45, 5
12.	1	45	5	9	7, 9
13.	+	7	9	16	16
14.	6				16, 6
15.	4				16, 6, 4
16.	*	6	4	24	16, 24
17.	+	16	12	40	40

Algorithm to Evaluate prefix expression by stack method:

Let 'P' is a expression written in prefix notation 'RSTACK', RSTACK is a stack to hold operands.

1. Scan 'P' from right to left.
2. If scanned character is operand, PUSH it on 'RSTACK'.
3. If scanned character is operator, then
   a. POP the top two from 'RSTACK' where 'A' is the top element and 'B' is the next to top element.
   b. Evaluate 'A' <operator> B
   c. PUSH the result into RSTACK.
4. Repeat step 1 to 3 until the end of P.
5. Stop.

Q. Evaluate the following prefix expression by stack method.

1. -1*, +, 4, 3, 2, 5
2. -, +, *, 2, 3, +, 5, 4, 9
3. +, 5, *, 3, 2
4. /, +, 5, 3, -, 4, 2
5. +, -, *, +, 1, 2, /, 4, 2, 1, 1, 4, 2

① −, *, +, 4, 3, 2, 5

S.No.	Symbol Scanned	Operator 1	Operator 2	Value	Stack
1.	5				5
2.	2				5, 2
3.	3				5, 2, 3
4.	4				5, 2, 3, 4
5.	+	4	3	7	5, 2, 7
6.	*	7	2	14	5, 14
7.	−	14	5	9	9

② −, +, *, 2, 3, +, 5, 4, 9

S.No.	Symbol Scanned	Operator 1	Operator 2	Value	Stack
1.	9				9
2.	4				9, 4
3.	5				9, 4, 5
4.	+	5	4	9	9, 9
5.	3				9, 9, 3
6.	2				9, 9, 3, 2
7.	*	2	3	6	9, 9, 6
8.	+	6	9	15	9, 15
9	−	15	9	6	6

(3) +, 5, *, 3, 2

S.N.	Symbol Scanned	Operator 1	Operator 2	Value	Stack
1.	2				2
2.	3				2, 3
3.	*	3	2	6	6
4.	5				6, 5
5.	+	5	6	11	11

(4) /, +, 5, 3, −, 4, 2

S.N.	Symbol Scanned	Operator 1	Operator 2	Value	Stack
1.	2				2
2.	4				2, 4
3.	−	4	2	2	2
4.	3				2, 3
5.	5				2, 3, 5
6.	+	5	3	8	2, 8
7.	/	8	2	4	4

(5) +, −, *, +, 3, 2, /, 4, 2, 3, ^, 4, 2

S.N.	Symbol Scanned	Operator 1	Operator 2	Value	Stack
1.	2				2
2.	4				2, 4
3.	1	4	2	16	16
4.	3				16, 3
5.	2				16, 3, 2
6.	4				16, 3, 2, 4
7.	/	4	2	2	16, 3, 2
8.	2	2	3		16, 3, 2, 2
9.	3				16, 3, 2, 2, 3
10.	+	1	2	3	16, 3, 2, 3
11.	*	3	2	6	16, 3, 6
12.	−	6	3	5	16, 5
13.	+	5	16	21	21

Unit-3

Queues

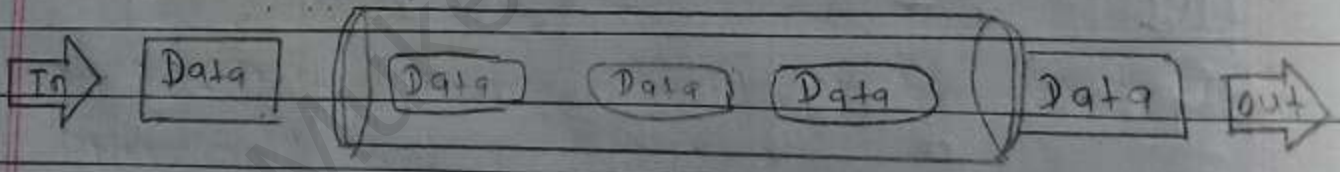
Page No. 40
Date 07/02/22

3.1. Definition

A queue is an ordered collection of items, from which items may be deleted at one end (FRONT) and into which item may be inserted at the other end (REAR). The first element inserted into the queue is the first element to be removed. So, it is also called FIFO (First in First Out).

Queue representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure:



Last in Last Out

First-in-First out

Queue

As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

3.2 Queue as an ADT (Abstract Data Type)

A Queue 'Q' of type 'T' is finite sequence of elements with the operations:

① Make Empty Queue(Q):
Make queue 'Q' as an empty queue.

② Is Empty (Q) :-
Check whether the queue 'Q' is empty or not.
Return 'True' if 'Q' is empty.
Return 'false otherwise

③ Is full (Q) :-
Check whether the queue 'Q' is full or not.
Return 'True' if Q is full.
Return 'false' otherwise

④ Enque (Q, x) :-
Insert the item 'x' at the REAR end of the queue 'Q' if and only if 'Q' is not full.

⑤ Dequeue (Q):-
Delete the item from the FRONT end of queue 'Q' if and only if the queue 'Q' is not empty.

⑥ Traverse(d)
Display the content of the entire queue.

3.3. Primitive Operations in queue: Enqueue and Dequeue

1. Enqueue Operation:
Queues maintain two data pointers FRONT and REAR. Therefore, its operations are comparatively difficult to implement than that of stacks.
The following steps should be taken to enqueue (Insert) data into a queue:-

→ Step 1:- Check if the queue is full.

- Step 2:- If the queue is full, produce overflow error and exit.

- Step 3:- If the queue is not full, increment REAR pointer to point the next empty space.

- Step 4: Add data elements to the queue location, where the REAR is pointing.

- Step 5: Return Success

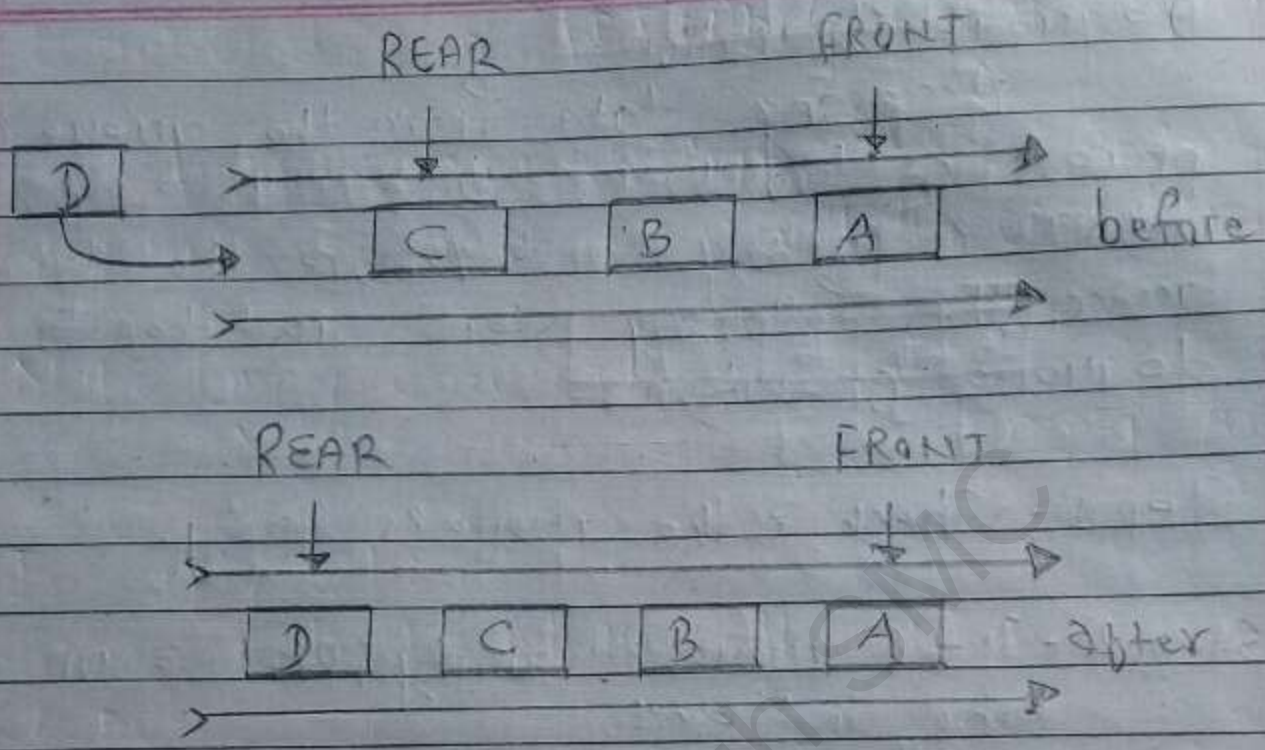


Fig: Enqueue Operation

Algorithm for enque operation:

```
procedure enqueue(data)
   if queue is full
   return overflow
   endif
 rear ← rear + 1
 queue[rear] ← data
   return true
end procedure
```

OR

```
Initialize R=-1 & F=-1
If R= MAXSIZE - 1
PRINT "Queue Overflow"
   and Exit
Else
  Set R = R+1
  Queue[R] = Item
Exit
```

2. Dequeue Operation:
Accessing data from the queue is a process of two tasks - access the data where FRONT is pointing and remove the data after access. The following steps are taken to perform dequeue operation:-

- Step 1 - Check if the queue is empty.
- Step 2 - If the queue is empty, produce underflow error and exit.
- Step 3 - If the queue is not empty, access the data where FRONT is pointing.
- Step 4 - Increment FRONT pointer to point the next available data element.
- Step 5 - Return success.

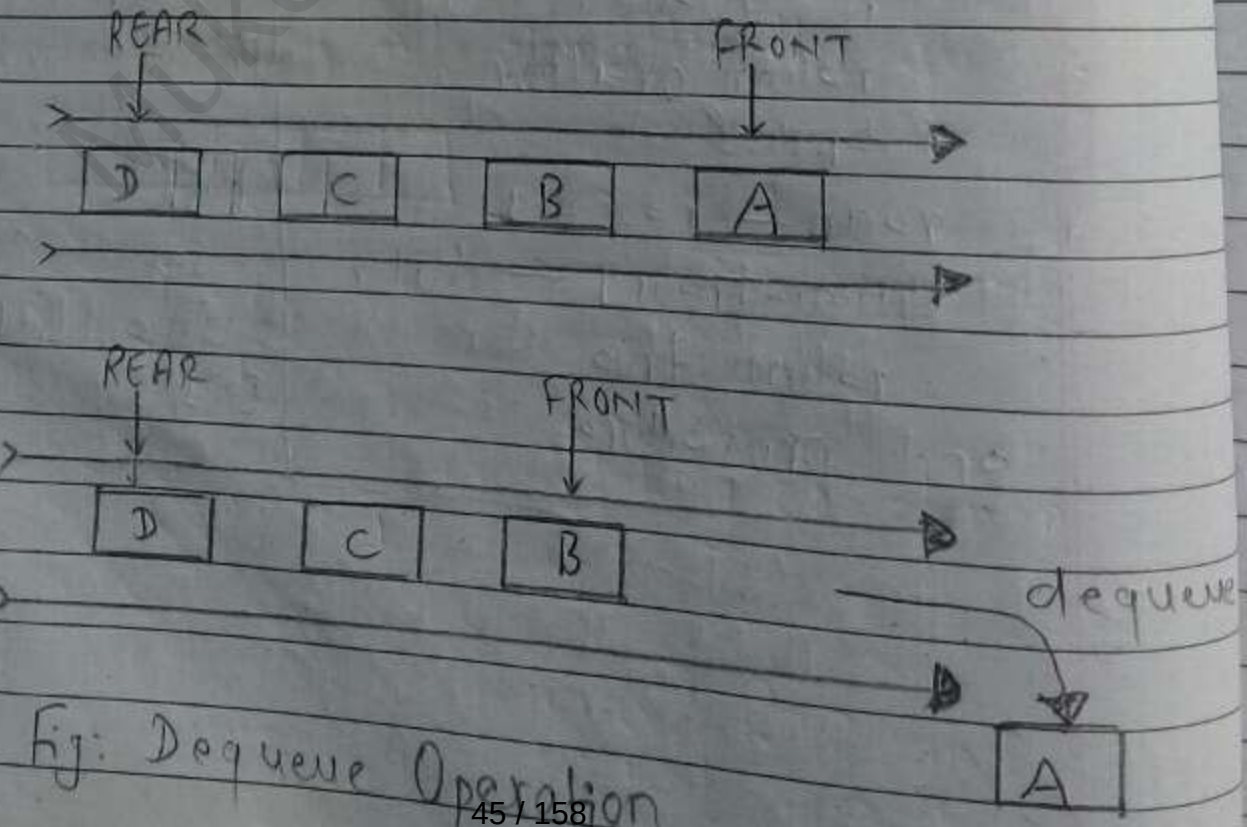


Fig: Dequeue Operation

Algorithm for dequeue operation:

OR

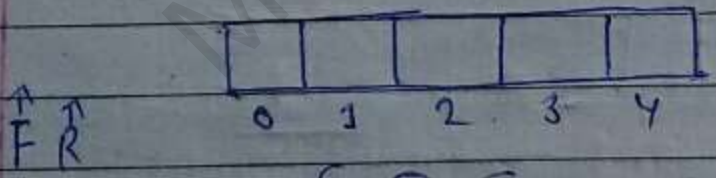
```
procedure dequeue
   if queue is empty
      return underflow
   end if
   data = queue[FRONT]
   FRONT ← FRONT + 1
      return true
   end procedure.
```

```
If R=-1 and F=-1
Print "Queue underflow
   and exit
Else
item = Queue[F]
   Set F= F+1
Exit
```

3.4. Linear Queue, Circular Queue, Priority Queue

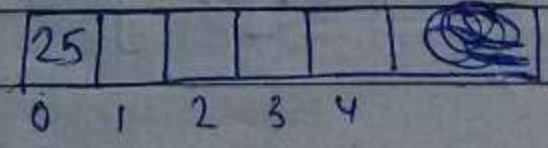
1. Linear Queue:

Linear queue is the queue where the value of FRONT and REAR can only be implemented.



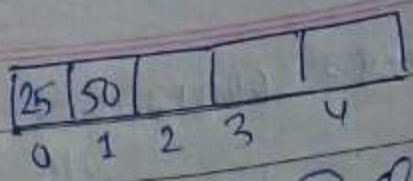
Fig① Empty Queue

F = -1, R = -1



Fig② Queue with one element

F = 0, R = 0



Fig③ Queue with two elements
F=0, R=1

Fig④ Queue with three elements
F=0, R=2

From the above figure, we can say that R=-1 and F=-1, where queue is empty. R=0 and F=0 when there is one element in a queue. After that only 'R' is implemented by 1 with insertion of each element in the queue. 'F' is only increased for one time during the insertion of first element in the queue.

Fig⑤ Queue after deleting one element.
R=2, F=1

~~Eaq~~ In the above figure⑤, R=2 and F=1, where 1 element is deleted from the queue.

then 'F' will get incremented by 1.

Problems with linear queue implementation:

- Both REAR and FRONT indices are increased but never decreased.
- As items are removed from the queue, the storage space at the beginning of the array is discarded and never used again.
- Wastage of space is the main problem with linear queue which is illustrated by the following examples:

Fig:- Linear queue with two elements deleted.

The queue is considered full, even though space at the beginning is vacant.

2. Circular Queue:

→ A circular queue is one in which the insertion of new element is done at very first location of the queue if the last location of the queue is full.

→ A circular queue overcomes the problem of unutilized space in linear queue implementation as array.

→ In circular queue we sacrifice ~~the~~ one element of the array thus to insert n elements in a circular queue. We need an array of size n+1 (or we can insert one less than the size of the array in circular ~~space~~ queue).

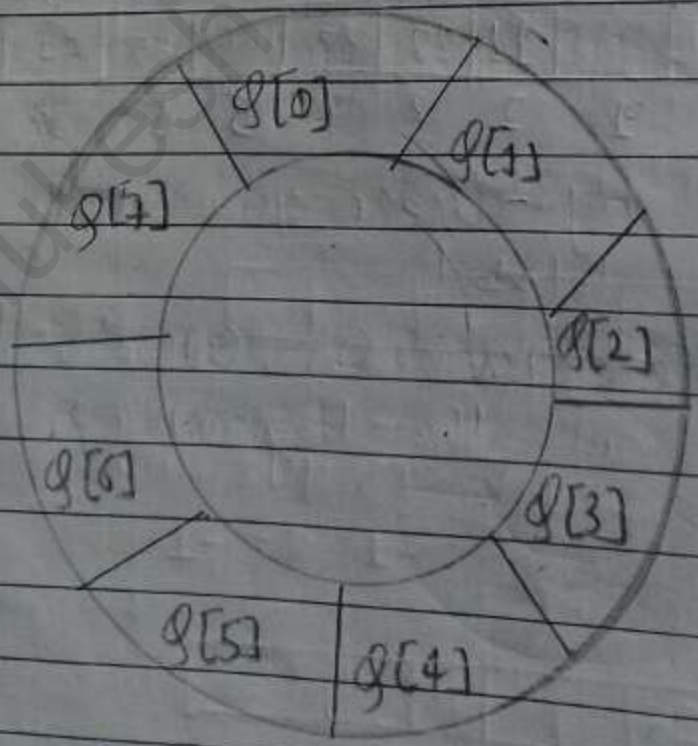


fig: Circular queue.

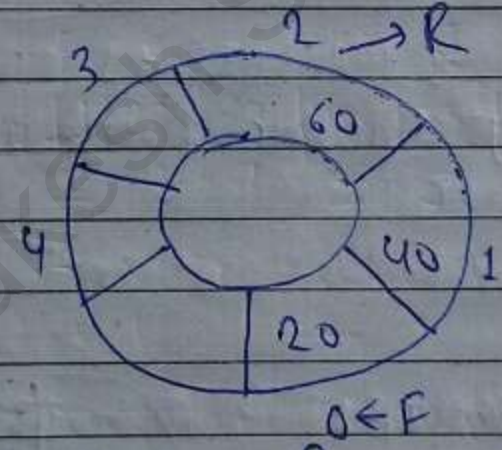
In circular queue, the following assumpti-ons are made:

(i) FRONT will always be pointing to the 1st element (as in the linear queue).

(ii) If FRONT = REAR then the queue will be empty.

(iii) Each time a new element is inserted into the queue, the rear is incremented by 1. i.e. ~~Rear = Rear+1~~ Rear = Rear+1.

(iv) Each time an element is deleted from the queue, the value of FRONT is incremented by 1. i.e. FRONT = FRONT+1



In the above figure, a circular queue (cq) with the capacity of five elements is shown. Initially, there are three elements, with F=0, R=2.

① Insert item 50

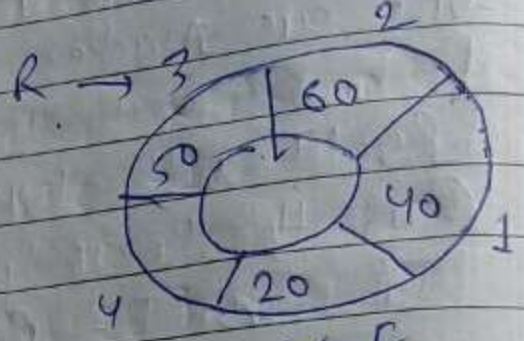
here

$R = (R+1) \% MAXSIZE$

$= (2+1) \% 5$

$= 3 \% 5$

$= 3$



② insert item 10.

Now

$R = (R+1) \% MAXSIZE$

$= (3+1) \% 5$

$= 4 \% 5$

$= 4$

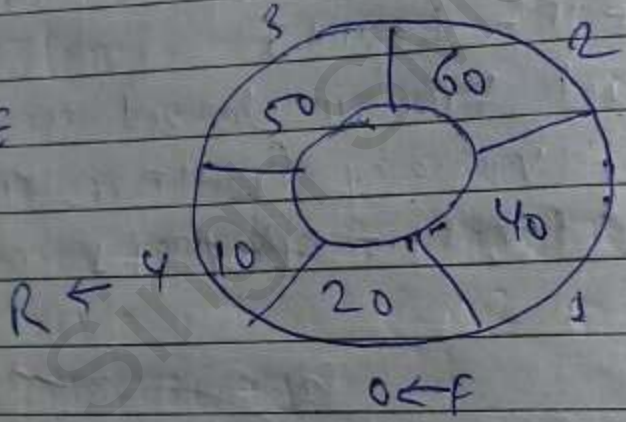


fig: Circular queue after inserting 10.

③ Insert item 30

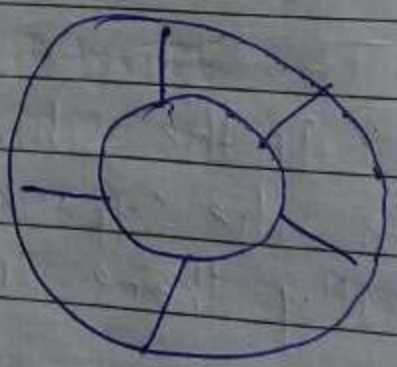
Now

$R = (R+1) \% MAXSIZE$

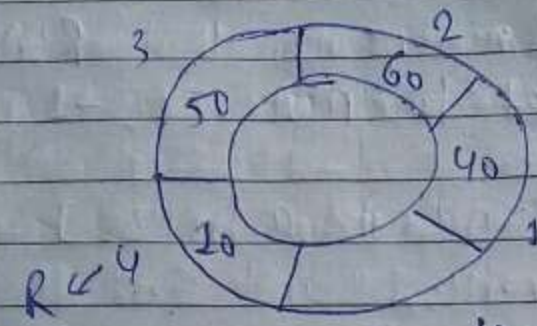
$= (4+1) \% 5$

$= 5 \% 5$

$= 0$



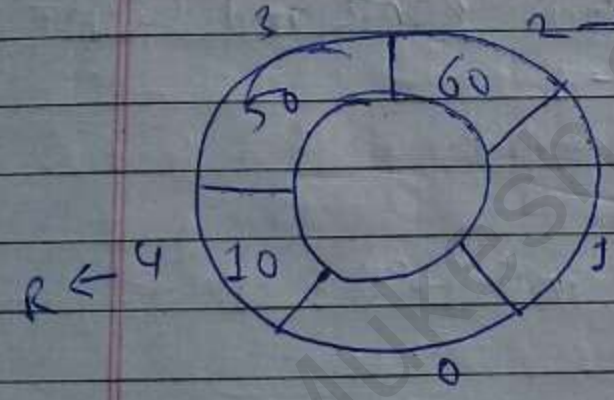
4. Delete one element:



Here, F = (F+1) % MAXSIZE
= (0+1) % 5
= 1 % 5 = 1

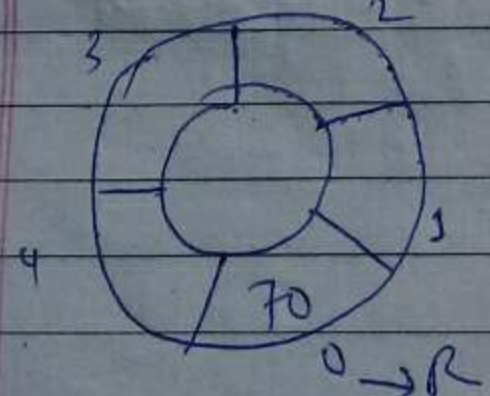
Fig: Circular queue after deleting 20.

(5) Delete another element:



Here
F = (F+1) % MAXSIZE
= (1+1) % 5
= 2 % 5
= 2

(6) insert item 70



Here
R = (R+1) % MAXSIZE
= (4+1) % 5
= 5 % 5
= 0.

Algorithm for enque operation in circular queue

This algorithm assumes that REAR and FRONT are initially set to MAXSIZE -1

If [FRONT = (REAR+1)% MAXSIZE]
Print "Queue overflow" and Exit
Else
REAR = (REAR+1) % MAXSIZE
Queue[REAR] = Item
Exit.

Algorithm for deque operation in circular queue

This algorithm assumes that REAR and FRONT are initially set to MAXSIZE - 1

If REAR = FRONT
PRINT "Queue Underflow" and Exit
Else
FRONT = (FRONT+1) % MAXSIZE
item = Queue [FRONT]
Return item
Exit

(3) Priority Queue:

A priority queue is a collection of elements such that each element has been assigned a priority and the order in which elements are deleted and process comes from the following rules:

@ An element of higher priority is processed before any element of lower priority

b Two elements with the same priority are processed according to the order in which they were added to the queue

Types of priority queue:

① Ascending Priority Queue:
A collection of items into which item can be inserted arbitrarily and from which only the smallest item can be removed, is called ascending priority queue.

② Descending Priority Queue:
A collection of items into which item can be inserted arbitrarily and from which only the largest item can be removed is called ~~Ascending~~ Descending priority queue.

Operations in priority queue:

(a) Insertion:
The insertion in priority queue is same as non-priority queue.

(b) Deletion:
Deletion requires a search for the element of highest priority and deletes the element of highest priority. The following methods can be used for deletion / removal from a given priority queue:

(i) An empty indicator replaces deleted items.

(ii) After each deletion, elements can be moved up in the array decrementing the REAR.

(iii) The array in the queue can be maintained as an ordered circular array.

Unit-4

Linked Lists

Page No. 71
Date

4.1. List as an ADT:

A list is an abstract data type that represents a countable number of ordered values, where the same value may occur more than once.

A list contains elements of same type arranged in sequential order and following operations can be performed on the list.

get() → Return an element ~~at any position of~~ from the list at any given position.

insert() → Insert an element at any position of the list.

remove() → Remove the first occurance of any element from a non-empty list.

removeAt() → Remove the element at a specified location from a non-empty list.

replace() → Replace an element at any position by another element.

Size() → Return the number of elements in the list.

IsEmpty() → Return true if the list is empty, otherwise return false.

4.3 Linked list

Linked list is a very commonly used linear data structure which consists of groups of nodes in a sequence.

OR

A linked list is a linear data structure where each element is a separate ~~element~~ object.

Each element (node) of a list consists of two items - the data and a reference to the next node. The last node has a reference to null. The entry point into a linked list is called the head of the list. It should be noted that head is not a separate node, but the reference to the first node. If the list is empty then the head is a null reference.

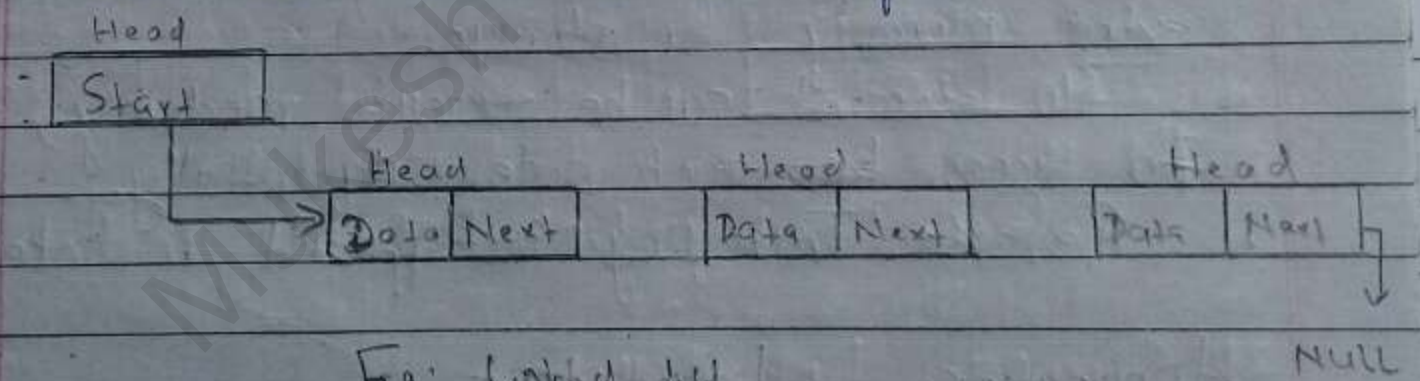


Fig: Linked list

Eg:

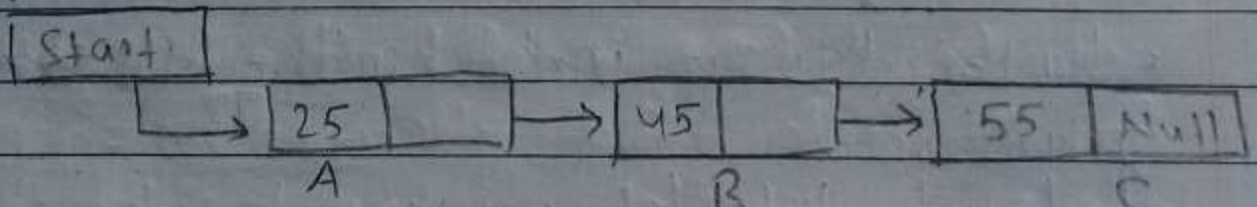


Fig: Linked list

Ref. Types of linked list: Doubly linked list, Circular linked list

Advantages of linked list:

- They are dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.
- Linked list reduces the access time.

Disadvantages of linked list:

- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each nodes sequentially.
- Reverse Traversing is difficult in linked list.

Applications of linked lists:

- Linked lists are used to implement stacks, queues graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.
- In linked list, we don't need to know the size in advance.

4.4. Types of linked list: Singly linked list, Doubly linked list, Circular linked list

There are 3 different implementation types of linked list available, they are:

1. Singly linked list:

A Singly linked list is a dynamic data structure which may grow or shrink, and growing and shrinking depends on the operation mode.

Singly linked list contains nodes which have a data part as well as an address part i.e. next, which ~~many~~ points to the next node in the sequence of nodes. The last node has a NULL pointer.

The operations we can perform on singly linked lists are insertion, deletion and traversal

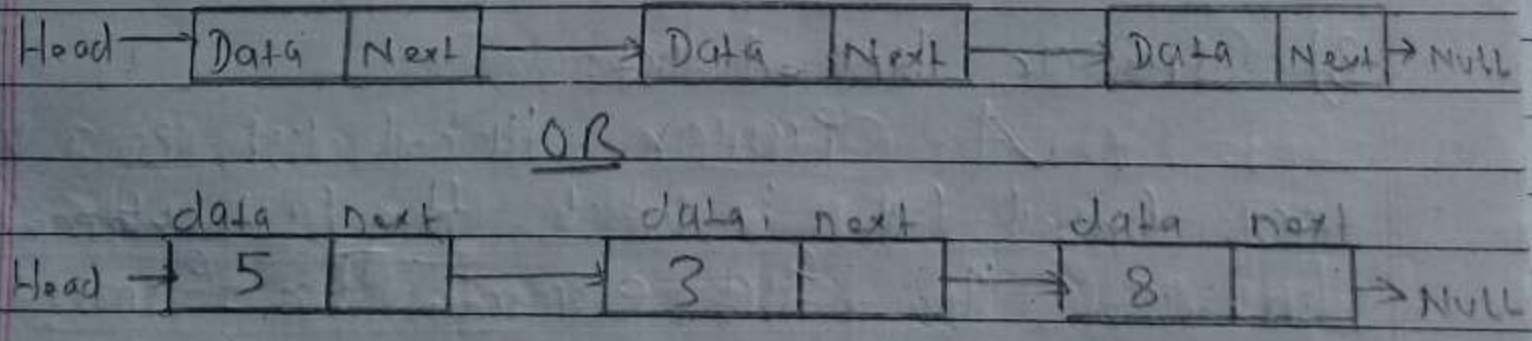


Fig: Singly linked list representation

2) Doubly linked list:

Doubly linked list is a sequence of elements in which every node has a link to its previous node and next node.

In doubly linked list, each node contains a data part and two addresses, one of the previous and one for the next node. Traversing can be done in both forward and backward directions and displays the contents in the whole list.

fig: Doubly linked list

3) Circular linked list:

A circular linked list is a variation of linked list in which the last element is linked to the first element. This forms a circular loop.

In circular linked list, the last node of the list holds the address of the first node hence forming a circular chain.

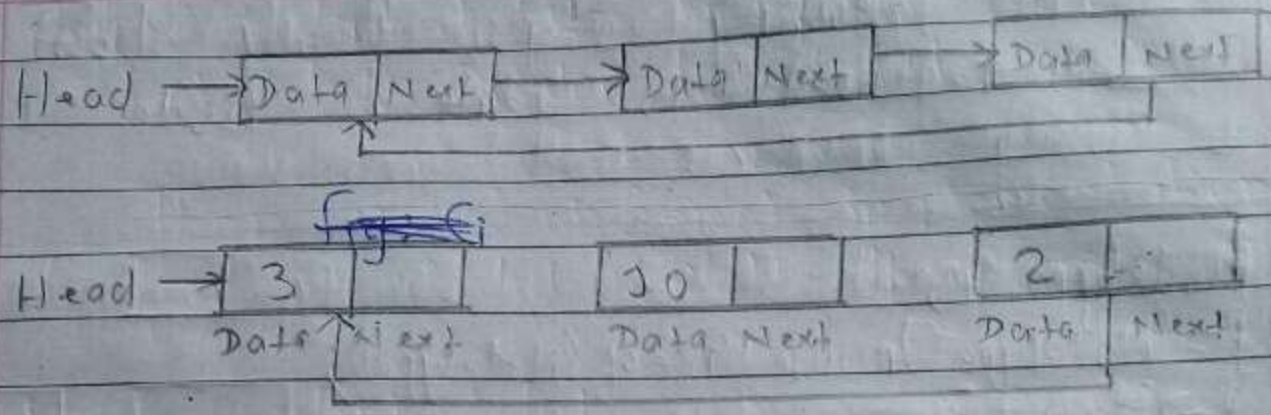


Fig: Circular linked list

A Circular linked list can be either singly linked or doubly linked.

4.5 Basic Operations in Linked list: creation, node insertion and deletion from beginning, end and specified position:

1) Creating a Node:

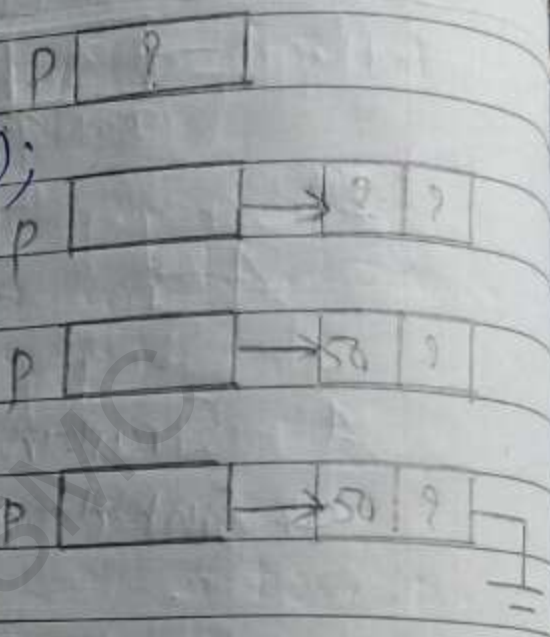
In this algorithm a linked list of nodes is created. The list is pointed by pointer first, the last node of the list to NULL, indicating the end of the list.

~~let us assume that a linked list of N number of nodes is to be created. The operator new will be used for dynamic allocation of node. A variable~~

The following steps clearly shows the steps required to create a node and storing an item:

```
Nodetype *p;
p=(Nodetype*)malloc(sizeof(Nodetype));
p->data=50;
p->next=NULL;
```

Note: P is pointer



2) Insertion of Nodes:

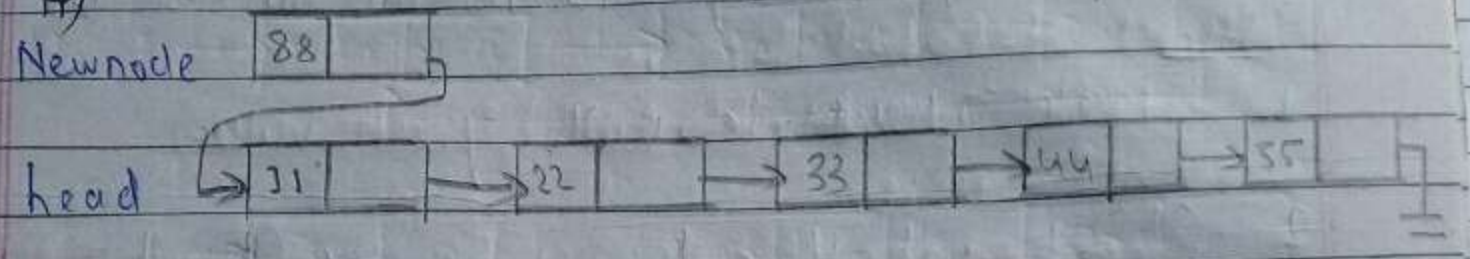
In a singly linked list, the insertion of nodes can be done in 3 ways:

i) At the beginning of the linked list

We can use following steps to insert a new node at the beginning of the linked list:

Step 1: Create a newNode with given value.
Step 2: Check whether list is Empty (head == NULL)
Step 3: If it is Empty then, set newNode = NULL and head = newNode
Step 4: If it is not Empty then, set newNode→next = head and head = newNode
Step 5: End

(ii)



(ii) At the end of the linked list

Steps

1. Create a newNode with given value and newNode → next as NULL

2. Check whether list is Empty (head == NULL)

3. If it is Empty then, set head = NewNode

4. If it is Notempty then, ~~define~~
   Set temp = head;

5. While (temp -> next != NULL)
   temp = temp -> next;

6. Set temp → next = newNode
7. End.

iii) At the specified position in the linked list

Steps:

1 Create a newNode with given value.

2 ~~Check whether list is Empty (head == NULL)~~

3. ~~If it is Empty then~~
~~Set newNode → next = NULL and head = newNode~~

4. ~~If it is Not Empty then~~
Set temp = head

2. Assign data to the data field of new node
newNode → data = newItem;

3. Enter the position of node at which you want to insert a new node let position be 'Pos'.

4. Set temp = head;

5. if (head == NULL) then
printf("void insertion");

6. for (i = 1; i < pos; i++)
temp = temp → next;

7. Set newNode → next = temp → next;
Set temp → next = newNode;

8. End.

3) Deletion of Nodes

In a singly linked list, deletion operation can be done in three ways:

i) from the beginning of the linked list

Steps

1. If (head == NULL) then
   print "Void deletion" and exit
2. Store the address of first node in a temporary variable temp.
   i.e temp = head;
3. Set head to next of head
   head = head -> next;
4. Free the memory reserved by temp variable
   free(temp)
5. End

ii) from the end of the linked list:

Steps

1. If (head == NULL) then
   print "Void deletion" and exit
2. else if (head -> next == NULL) then

Set temp = head;
print deleted item as,
printf("%d", head -> data);
head = NULL;
free(temp);

3. else
Set temp = head
while (temp -> next -> next != NULL)
Set temp = temp -> next;
End of while
free (temp -> next);
Set temp -> next = NULL;

4. End

iii) From the specified position in a linked list

Steps

1. Read position of node to be deleted. Let it be 'pos'.

2 if (head == NULL)
print "void deletion" and exit

3. Enter position of node at which you want to delete a new node.

4. Set temp = head.
   declare a pointer of structure, let it be *p
5. if (head == NULL) then
   print "void deletion" and exit.
   Otherwise
6. for (i=1; i<pos; i++)
   temp = temp → next → data

7. Print deleted item is temp → next → data

8. Set p = temp → next;

9. Set temp → next = temp → next → next

10. free(p);

11. End.

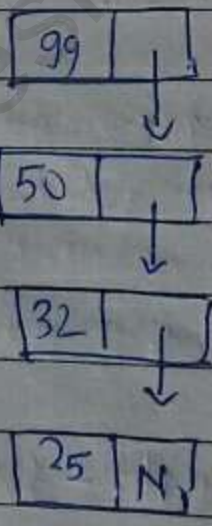
4.6 Stack and Queue as a linked list

A) Stack as a linked list:

A stack data structure can be implemented using ~~array~~ linked list data structure. The stack implemented using linked list can work for unlimited number of values.

In linked list implementation of stack, every new elemented is inserted as 'top' element. ~~That means~~ To remove an element from the stack, simply remove the node which is pointed by 'top' by moving 'top' to its next node in the list. The next field of the first element must be always NULL.

Example



In the above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32, 50 & 99.

Push(value) - Inserting an element into the Stack:

We can we following steps to insert a new node into the stack:

1) Create a newNode with given value.

2) Check whether stack is Empty (Top == NULL)

3) If it is Empty, then set newNode → next = NULL

4) If it is Not Empty, then set
newNode → next = top

5) Finally, set top = newNode.

Pop() - Deleting an element from Stak:

We can we the following steps to delete a node from the stack:

1) Check whether stack is Empty (top == NULL)

2) If it is Empty, then display "Stack is Empty!!! Deletion is not possible!!!" and exit.

3) If it is Not Empty, then set
temp = top

4) Then set top = top → next

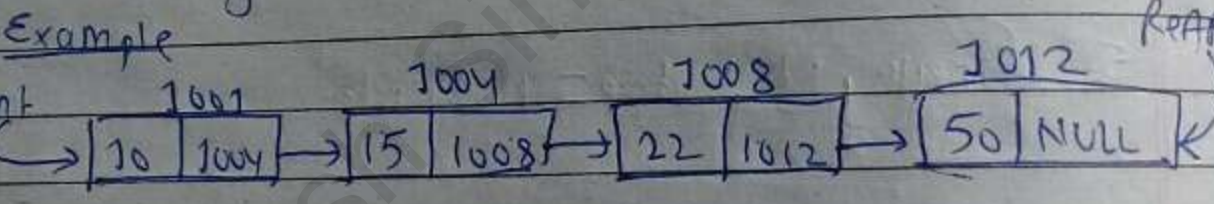
5) Finally, delete temp (free(temp))

3) Queue as a linked list:

A queue data structure ~~can be~~ implemented using a link list can work for unlimited number of values.

In linked list implementation of queue, the last inserted node is always pointed by 'rear' and the first node is always pointed by 'front'.

Example



In the above example, the last node is 50 and it is pointed by 'rear' and the first inserted node is 10 and it is pointed by 'front'. The order of elements inserted is 10, 15, 22 and 50.

i) Enqueue(value) - Inserting an element into the Queue

1) Create a newNode with given value and set 'newNode → next' to NULL.

2) Check whether queue is Empty (rear == NULL)

3) If it is Empty, set
front = newNode and rear = newNode

4) If it is not Empty then, set
rear → next = newNode and rear = newNode

5)

ii) DeQueue() – Deleting an element from Queue

1) Check whether queue is Empty (front == NULL)

2) If it is Empty, then display "Queue is Empty!!! Deletion is not possible!!" and exit.

3) If it is not Empty then, define a set
temp = front

4) Then set 'front = front → next' and
delete 'temp' (free(temp)).

Unit-5

Recursion

Page No. 55
Date

~~5.1. Principle of recursion:~~

Recursion is a process by which a function calls itself repeatedly until some specified condition has been satisfied. In order to solve a problem recursively, two conditions must be satisfied:

a) The problem must be written in recursive form.

b) The problem statement must include a stopping condition.

5.1. Principle of Recursion:

For designing a good recursive program, we must make certain assumption such as:

1) A recursion algorithm must have a base case

→ First, a base case is the condition that allows the algorithm to stop recursion. A base case is typically a problem that is small enough to solve directly. In the list sum algorithm the base case is a list of length 1.

2) A recursion algorithm must change its state and move toward the base case

To obey the second law, we must arrange for a change of state that moves the algorithm toward the base case. A change of state means that some data that

the algorithm is using is modified. In the list sum algorithm our primary data structure is a list, so we must focus our ~~effort~~ state changing efforts on the list.

3) A recursive algorithm must call itself
The final law is that the algorithm must call itself. This is the very definition recursion. Recursion is a confusing concept to many beginning programmers.

5.2. Comparison between recursion and iteration:

Recursion	Iteration
1. Recursion is the technique of defining anything in terms of itself.	1 Iteration allows to repeat the task till the given condition is satisfied.
2 All problems may not have recursive solution.	2 Any recursive problems can be solved iteratively.
3. Recursive programming technique conserves considerable execution time.	3 Iterative programming technique takes less execution time in comparison to recursion
4. Recursive problems are not efficient for memory	4 Iterative problems are effective in terms of

Utilization and execution speed of Problems.	memory consumption and execution speed.
5. Recursive technique is a top-down approach where the given program is divided into modules.	5. Iterative technique is a bottom up approach where it constructs the solution step by step.

Types of Recursion:

1. Direct Recursion:
The recursion in which the function calls itself is called direct recursion. In this type, only one function is invoked.

```
int xyz()
{
.....;
.....;
xyz();
}
```

2. Indirect Recursion:
If two function calls one another mutually then this type of recursion is called indirect recursion. The indirect recursion doesnot make any overhead as

direct recursion. When control exit from one function, the local variables of former function are destroyed. Hence, memory is not engaged.

```
int abc()
{
----;
----;
xyz();
}
int xyz()
{
abc();
---;
}
```

Advantages of Recursion:
i) Avoidance of unnecessary calling of function.
ii) Extremely useful when applying same solution.
iii) It does not use system stack.
iv) Fast at execution time.

Disadvantages of Recursion:
i) It occupies lot of memory space.
ii) A recursive function is often confusing.
iii) It is difficult to trace the logic of the function.

iv) The exit point must be explicitly code.

5.3 factorial, fibonacci sequence, GCD, Tower of Hanoi

Tower of Hanoi problem:

Tower of Hanoi is a game played with three poles and a number of different sized disks. Each disk has a hole in the center that makes easy to place into poles.

Initial State:

a) There are three poles named as origin, intermediate and destination.

b) 'n' number of different sized disks having hole at the center is stacked around the origin pole in decreasing order.

c) The disks are numbered as 1, 2, 3, 4... n.

Objective:

Transfer all disks from origin pole to destination pole using intermediate pole for temporary storage.

Rules:
a) Move only disk at a time.
b) Each disk must always be placed around one of the pole.
c) Never place larger disk on the top of smaller disk.

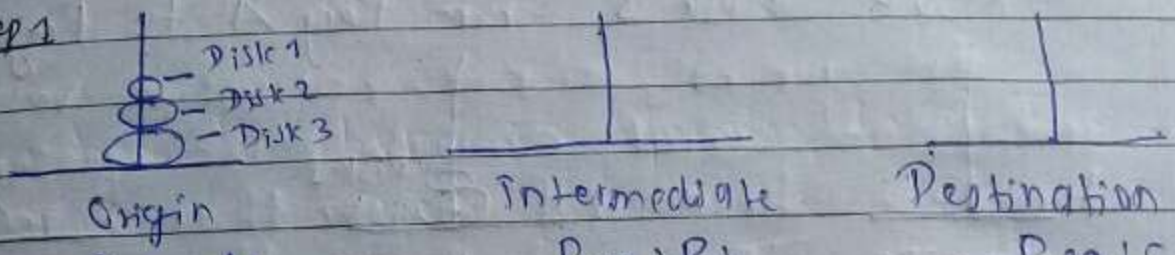
Algorithm:
To move a tower of 'n' disks from origin to destination pole (where 'n' is positive integer).
1) If $n = 1$, move a single disk from origin to destination pole
2) If $n > 1$,
a) Let 'intermediate' be the remaining pole other than origin and destination.
b) Move tower of ~~Hanoi~~ $(n-1)$ disks from origin to intermediate.
c) Move a single disk from origin to destination.
d) Move a tower of $(n-1)$ disks from intermediate to destination.

Solution to the Tower of Hanoi problem when number of disks 'n' $= 3$.

Let the origin, intermediate and destination pole or peg be denoted as Peg 'A', Peg 'B', and Peg 'C' respectively. Let the disks be

denoted as Disk 1, Disk 2 and Disk 3.

Step 1

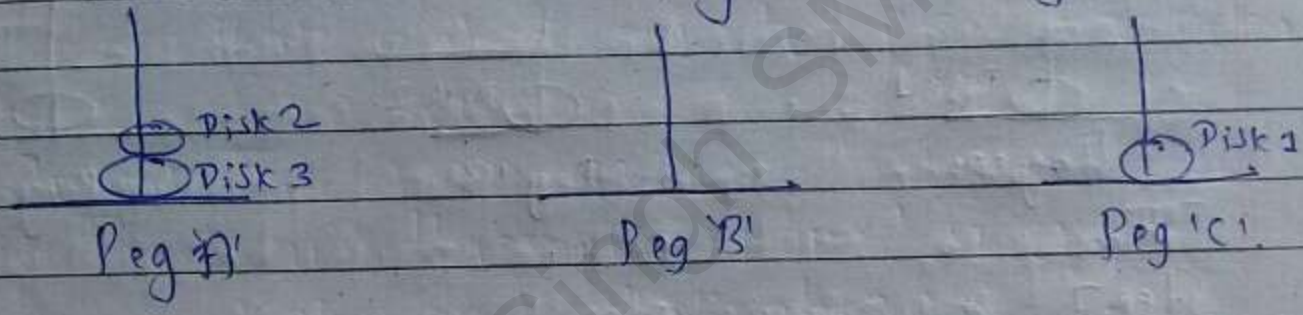


Peg → Peg 'A'

n = 3

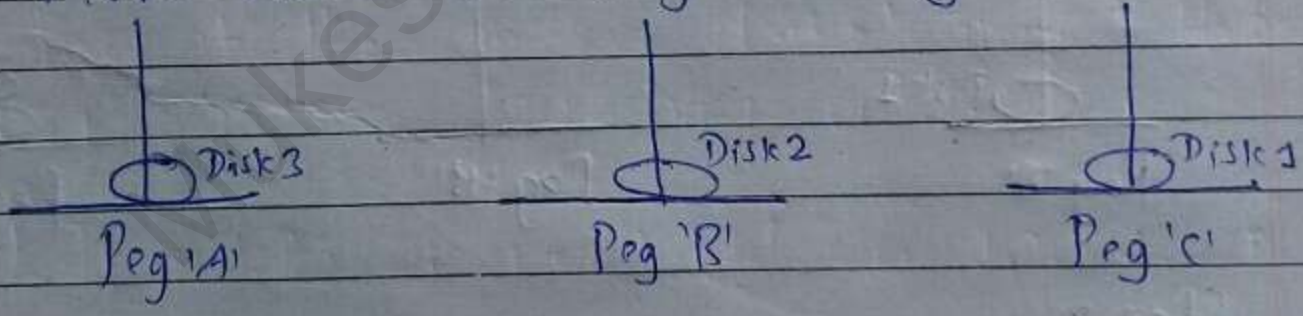
Step 2

Move Disk 1 from Peg 'A' to Peg 'C'



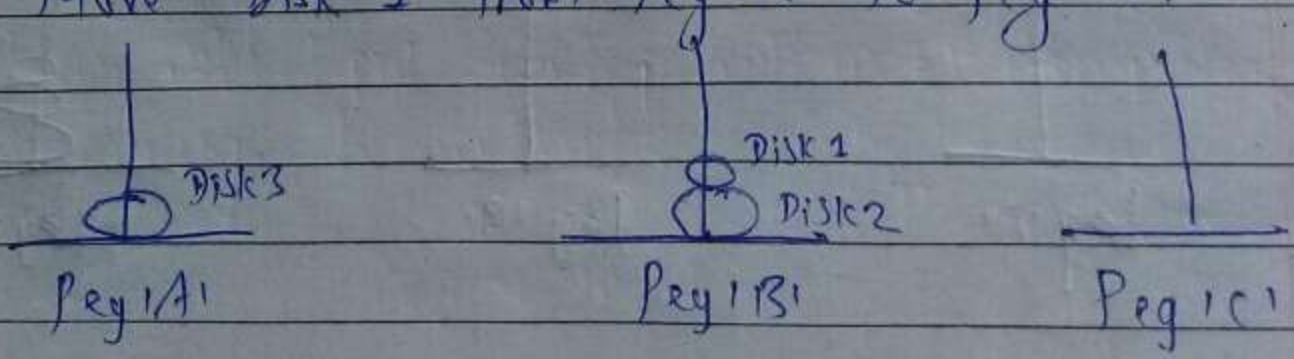
Step 3

Move Disk 2 from Peg 'A' to Peg 'B'



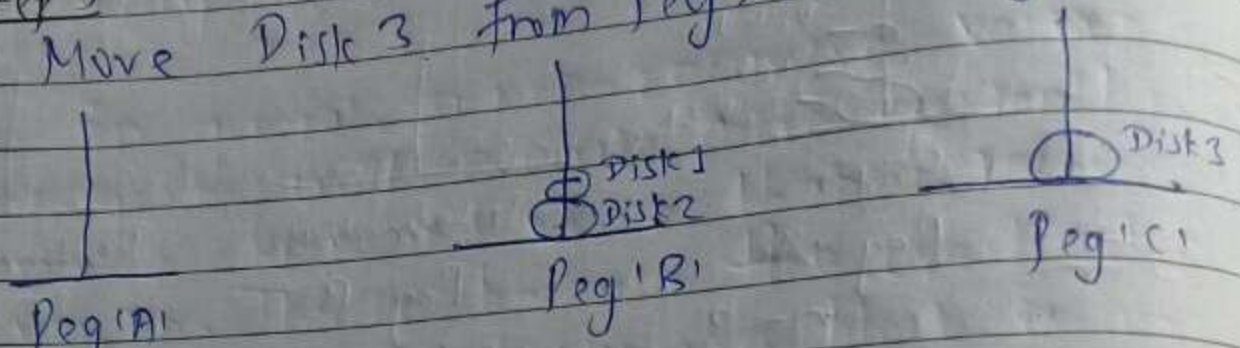
Step 4

Move Disk 1 from Peg 'C' to Peg 'B'.



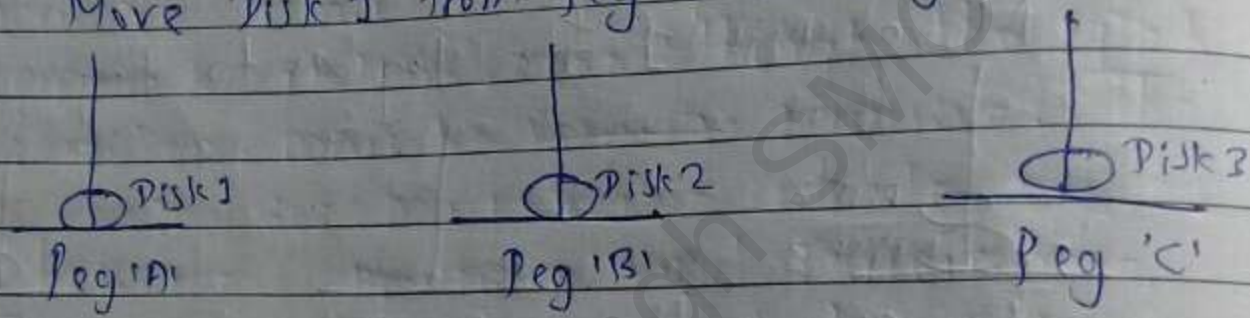
Step 5

Move Disk 3 from Peg 'A' to Peg 'C'



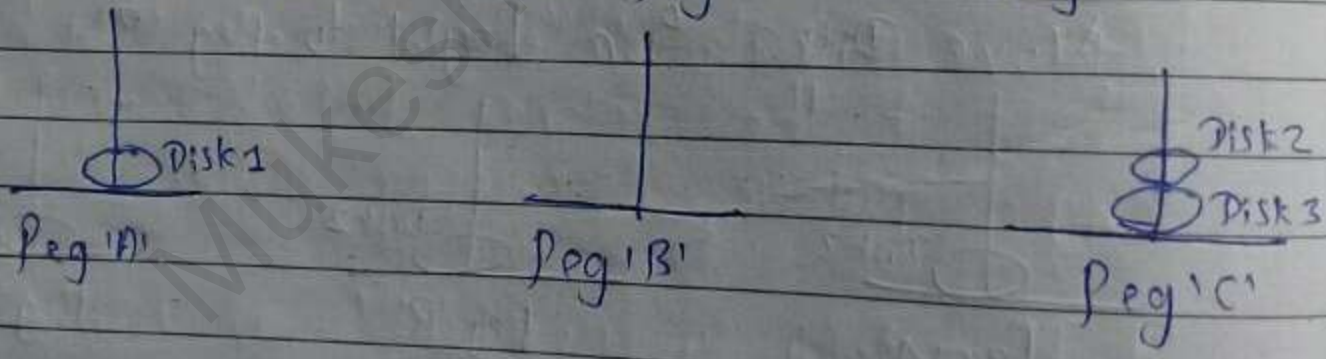
Step 6

Move Disk 1 from Peg 'B' to Peg 'A'



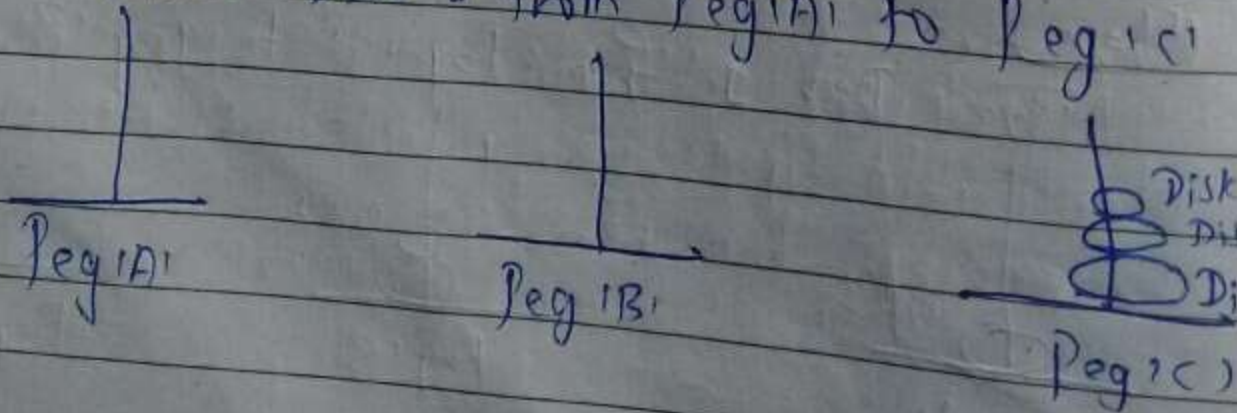
Step 7

Move Disk 2 from Peg 'B' to Peg 'C'



Step 8

Move Disk 1 from Peg 'A' to Peg 'C'



Tower of Hanoi Trick:

No. of moves required for transferring disks from origin to destination pole or peg is calculated as:

2^{n-1}, where 'n' is the no. of disks i.e.

for 1 disk, $2^1 - 1 = 2 - 1 = 1$ move

for 2 disks, $2^2 - 1 = 4 - 1 = 3$ moves

for 3 disks, $2^3 - 1 = 8 - 1 = 7$ moves

for 4 disks, $2^4 - 1 = 16 - 1 = 15$ moves.

5.9. Applications and efficiency of recursion

Applications of Recursion:

Recursion is applied to problems which can be broken into smaller parts, each part looking similar to the original problem. Some application areas of recursion are listed below:

i) Algorithms on trees and sorted lists can be implemented recursively.

ii) Recursion is also useful to guarantee the correctness of an algorithm.

iii) Recursion is used heavily in backtracking algorithms e.g. The Sudoku Solver.

iv) Recursion is used in several sorting algorithms like Quicksort, Merge Sort etc.

v) Parsers and Compilers also use recursion heavily.

Efficiency of Recursion:

In general, a non-recursive version of a program will execute more efficiently in terms of time and space than a recursive version. This is because the overhead involved in entering and exiting a block is avoided in the non-recursive version. Thus, the use of recursion is not necessarily the best way to approach problem, even though the problem definition may be ~~recursive~~ recursive. The use of recursion may involve a trade off in between simplicity and performance. Therefore, each problem must be judged on its own individuals merit.

Unit-6

Trees

6.1. Concept and definitions:

A tree is an abstract model of a hierarchial structure that consists of trees with a parent-child relationship.

- Tree is sequence of nodes.
- There is a starting node known as root node.
- Every node other than root has a parent node.
- Nodes may have any number of children.

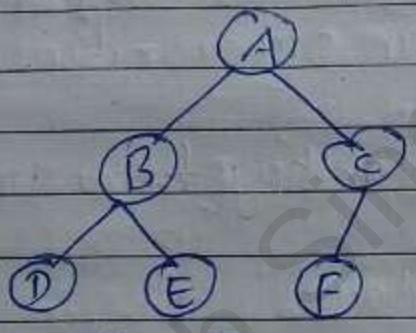


Fig: Tree

A has ~~two~~ 2 children B & C. and A is parent of B.

Recursive definition of tree:

Tree is a non-linear data structure which is collection of data (node) which is organized in hierarchial structure.

OR

A tree T of order 'n' is either empty or consists of a distinguished node 'r', called the root of T, together with at most n trees, $T_1, T_2 \ldots, T_n$ called the subtrees of T.

Characteristics of trees:
- Non-linear data structure
- Combines advantages of an ordered array.
- Searching as fast as in ordered array
- insertion and deletion as fast as in linked list.

Application:
- Directory structure of a file store.
- Structure of arithmetic expression
- Hierarchy of an organization.

Some terminology or key terms in trees:

i) Degree of a node:
The degree of node is the number of children of that node. In the above tree, the degree of node A is 3.

ii) Degree of a tree:
The degree of a tree is the maximum degree of node in a given tree. In the above tree the node A has maximum degree, thus the degree of the tree is 3.

iii) Path:

It is the sequence of consecutive edges from source node to destination node. There is a single unique path from the root to any node.

iv) Height of a node:

The height of a node is the maximum path length from that node to a leaf node. A leaf node has a height of 0.

v) Height of a tree:

The height of tree is the height of the root.

vi) Depth of a node:

Depth of the node is a path length from the root to that node. The root node has a depth of 0.

vii) Depth of a tree:

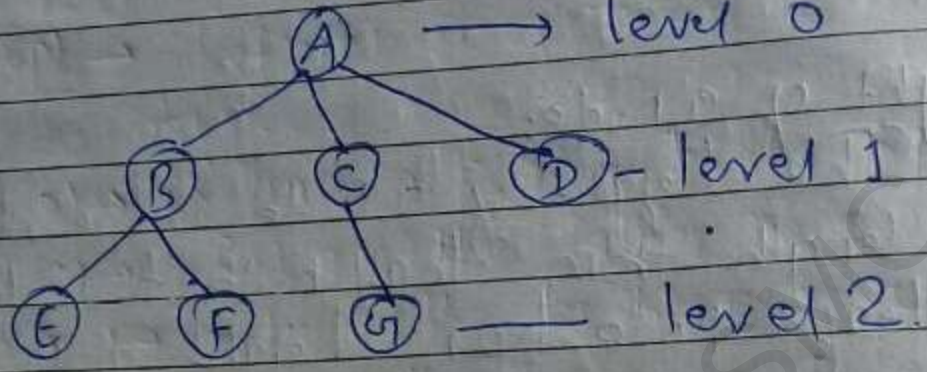
Depth of a tree is the maximum level of any leaf in the tree. This is equal to the longest path from the root of any leaf.

viii) Leaf of a node:

The level of a node is 0, if its

root, otherwise it is one more than its parent.

Illustration:



→ A is the root node.
→ B is the parent of E and F.
→ D is the sibling of B and C.
→ E and F are the children of B.
→ E, F, G, D are external nodes or leaves.
→ A, B, C are internal nodes.
→ Depth of F is 2.
→ the height of tree is 2.
→ the degree of node A is 3.
→ The degree of tree is 3.

6.2 Basic Operations in binary tree:

Binary Tree:

A binary tree is a finite set of elements that are either empty or partitioned into three disjoint subsets.

- The first ~~element~~ subset contains a single element called the root of the tree.
- The other two subsets are themselves binary trees called the left and right sub trees of the original tree. A left or right sub tree can be empty.

Each element of a binary tree is called a node of the tree. The following figure shows a binary tree with 9 nodes where A is root.

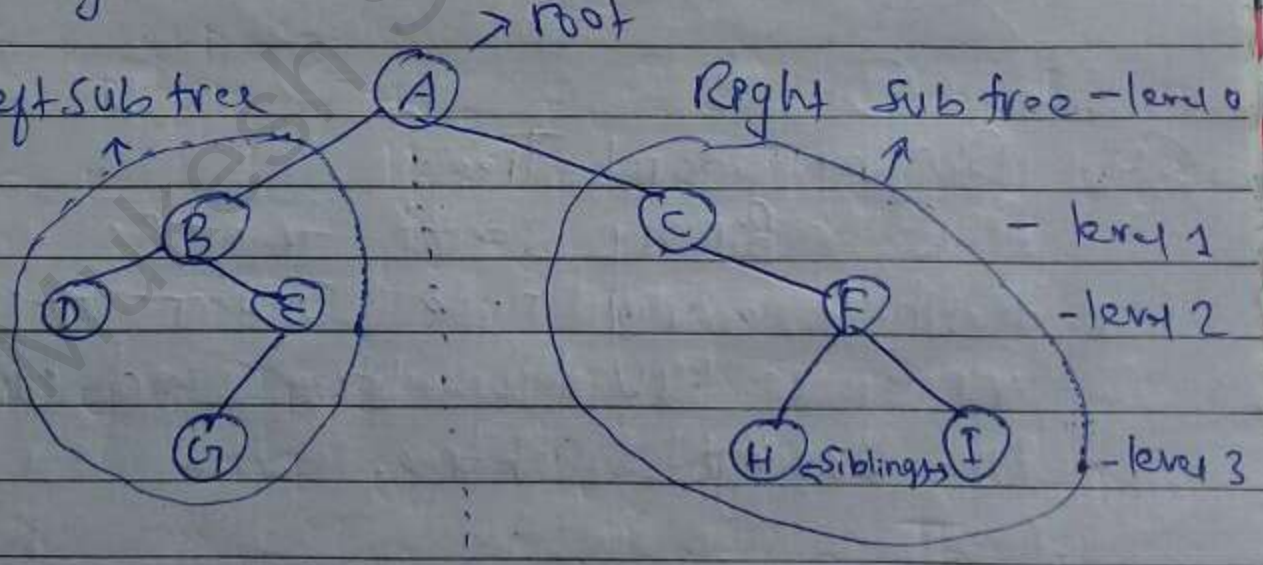


Fig: A binary tree

Properties of binary tree

→ if a binary tree contains m nodes at level 1, it contains at most 2m nodes at level 1+1.

→ Since a binary tree can contain at most 1 node

at level 0 (the root), it contains at most 2^1 nodes at level 1.

Operations on binary tree:

left(p) → Returns a pointer to the left sub-tree.
right(p) → Returns a pointer to the right sub-tree
parent(p) → Returns the father node of p.
brother(p) → Returns the brother node of p.
info(p) → Returns the contents of node p.
setLeft(p,x) → Creates the left child node of p and set the value x into it.
setRight(p,x) → Creates the right child node of p. the child node contains the info x.

6.4. Binary Search Tree:

Binary search tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

Example:

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.

Eg ①

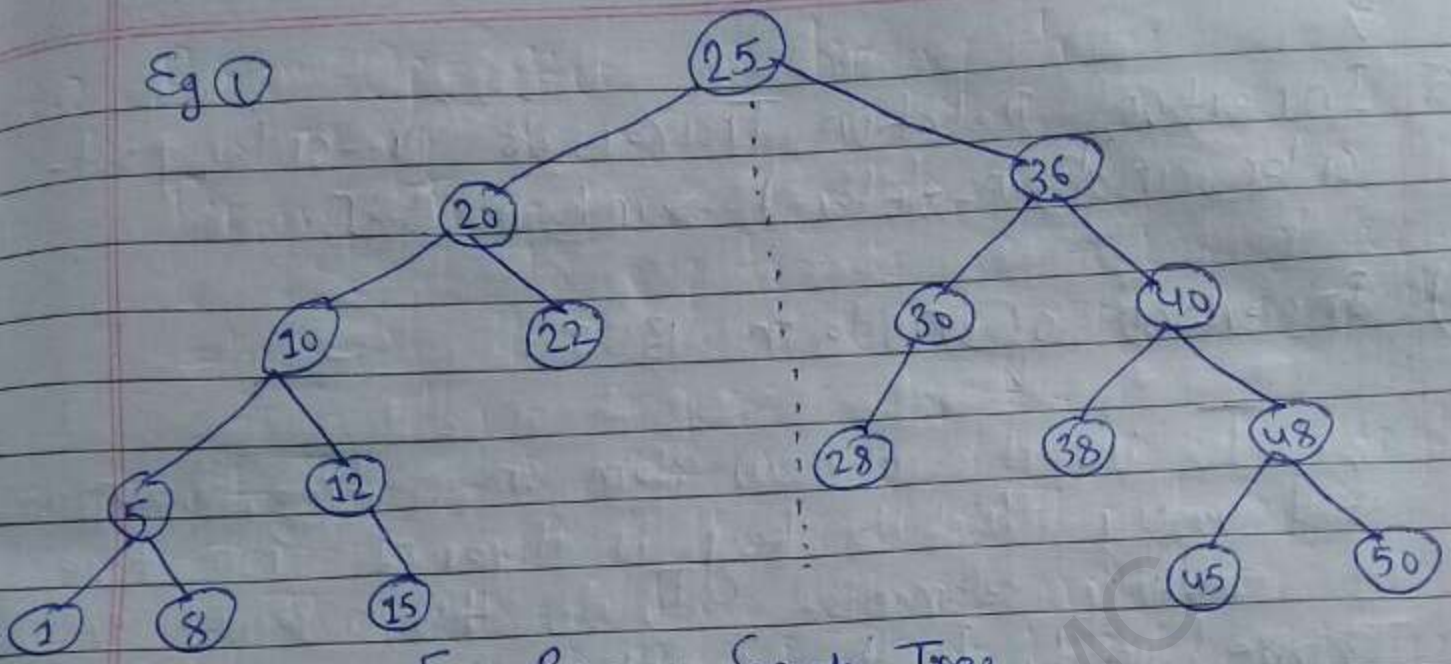


Fig: Binary Search Tree

Eg ② : 8, 16, 31, 6, 13, 24, 29, 5, 2, 11, 7

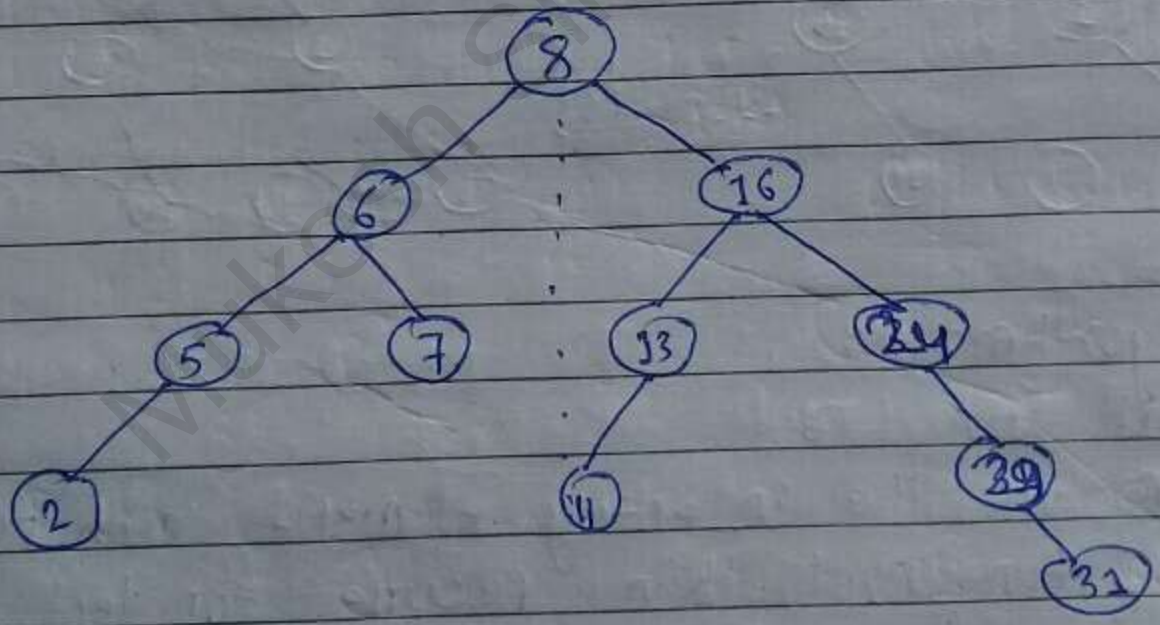
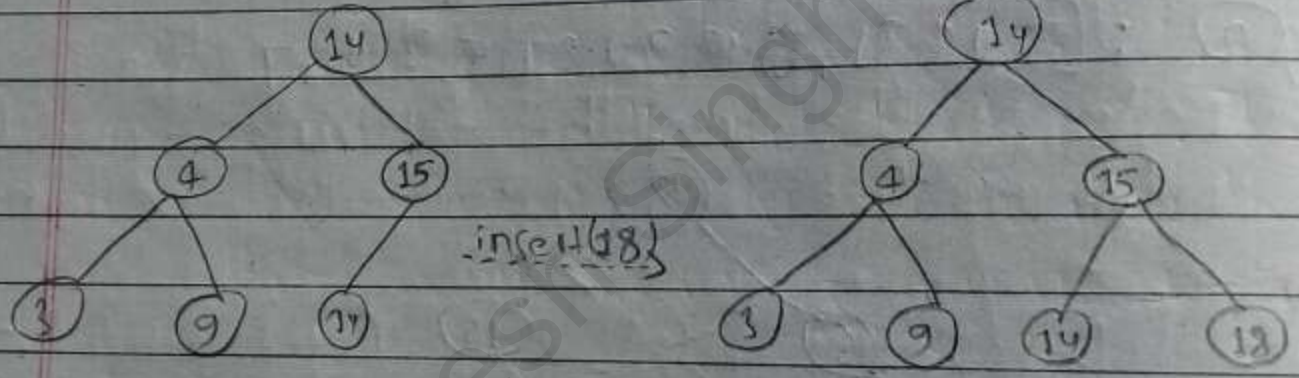


Fig: Binary Search Tree

6.5 Insertion, Deletion, Traversals (pre-order, post-order and in-order), Search in BST.

i) Insertion of node in BST:

To insert a new item in a tree, we must first verify that its key is different from those of existing elements. To do this, a search is carried out. If the search is unsuccessful, then new item is inserted.

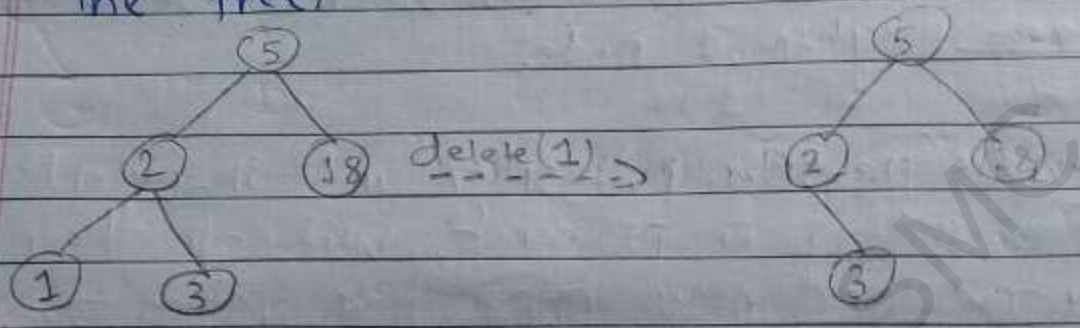


Algorithm:

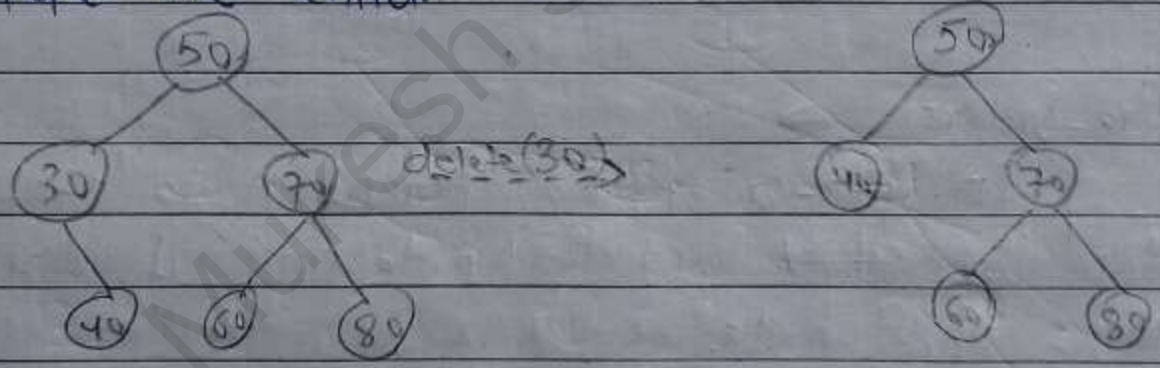
1. Start from root
2. Compare the inserting element with root, if less than root, then recurse for left, else recurse for right.
3. After reaching end, just insert that node at left (if less than current) else right.

ii) Deletion of node in BST:
While deleting a node from BST, there may be three cases:

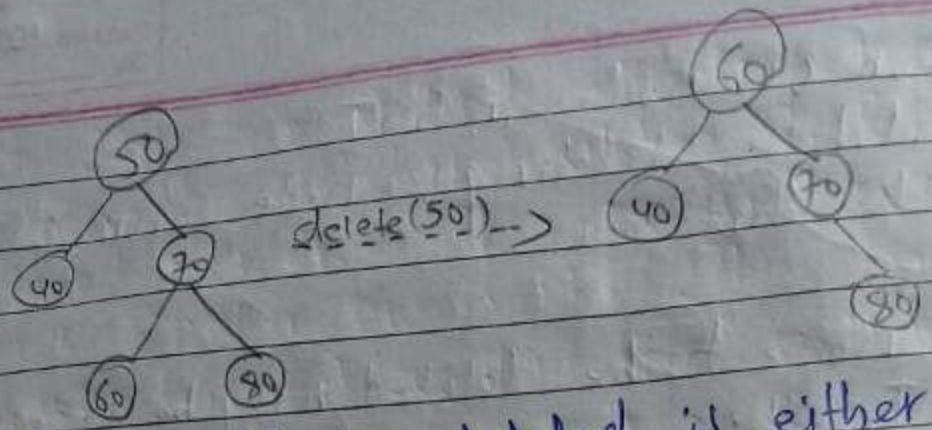
1) The node to be deleted may be a leaf node.
In this case, simply remove a node from the tree.



2) The node to be deleted ~~may be~~ has only one child:
In this case, copy the child to the node and delete the child.



3) Node to be deleted has two children:
Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.



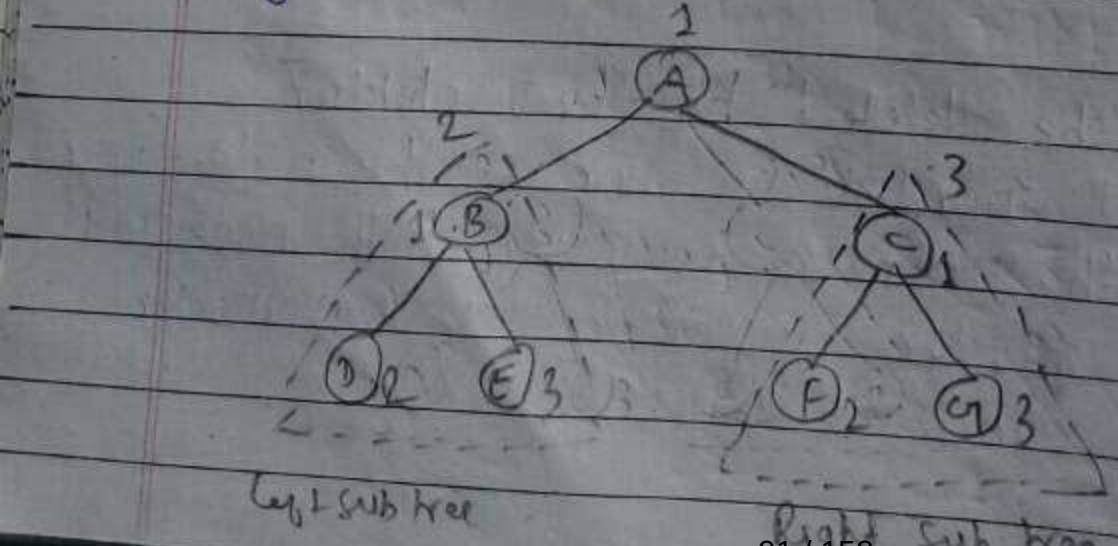
The node to be deleted is either replaced by its right sub-trees leftmost node or its left sub-trees rightmost node.

iii) Traversals (pre-order, post-order and in-order):

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways we use to traverse a tree -

a) Pre-Order:

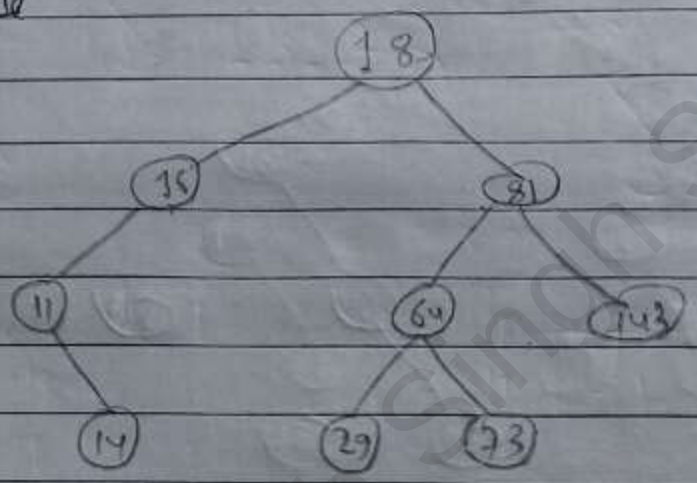
In pre-order, the parent node is visited first then the ~~child~~ left child node and at last the right child node.



We start from A, and following pre-order traversal, we first visit A itself and then move to its left subtree B. B is also also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be:

$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$

c) Example

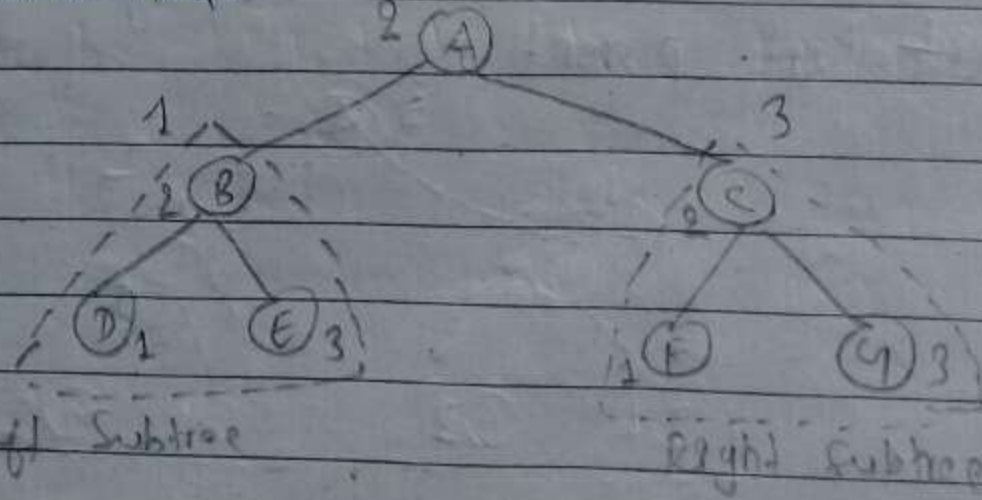


Pre-order: 18, 16, 11, 14, 81, 64, 29, 73, 143.

Binary Tree.

b) In-order Traversal:

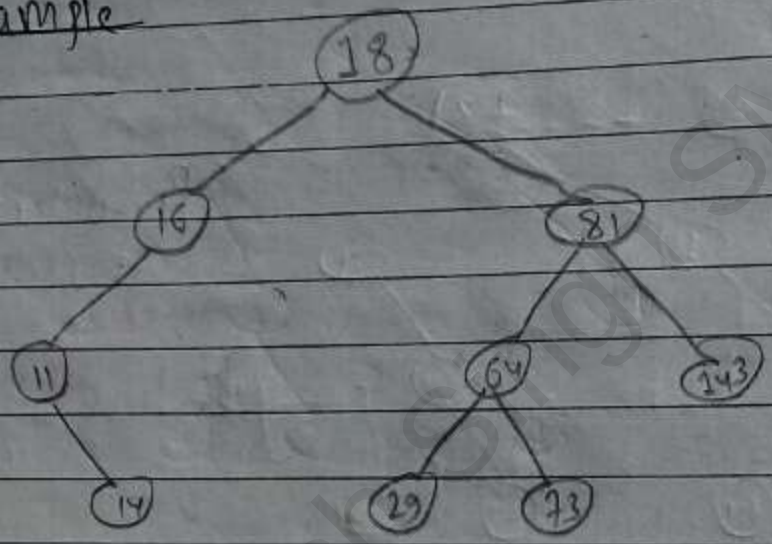
In in-order traversal, the left child node is visited first then parent node and at last the right child node.



We start from A, and following in-order traversal we move to its left subtree B. B is also traversed in-order. The process goes on until all the nodes are visited. The output of in-order traversal of this tree will be:

D → B → E → A → F → C → G.

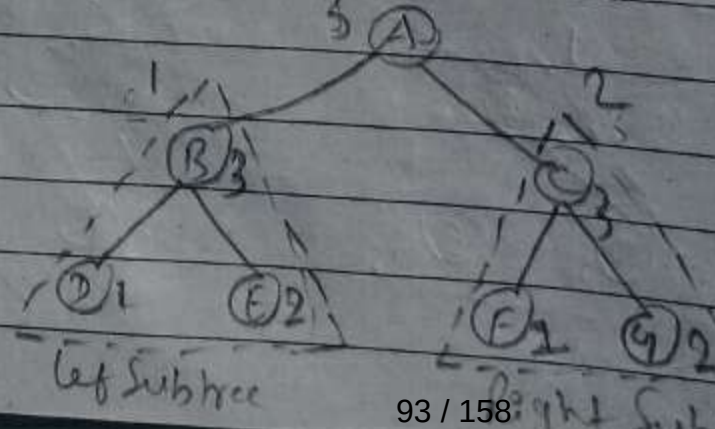
Example



In-order → 11, 14, 16, 18, 29, 64, 73, 81, 143

c) Post-order traversal:

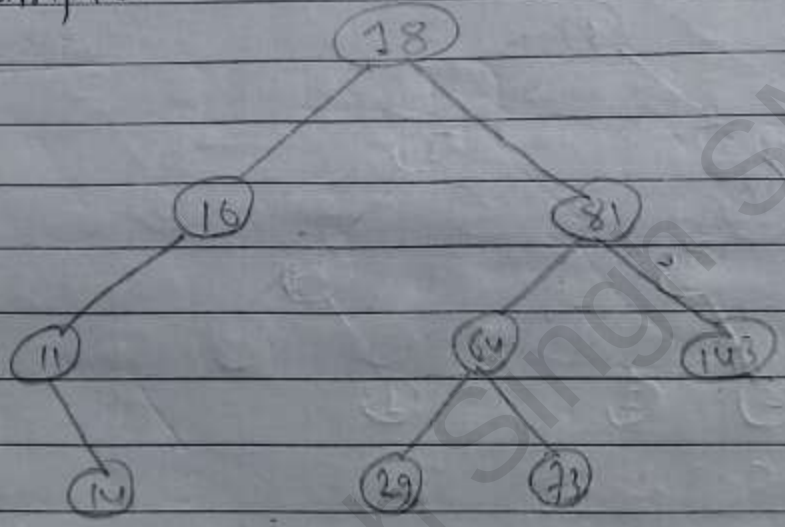
In post-order traversal, the left child node is visited first then the right child node and at last parent node.



a) We start from A, and following ~~the~~ pre-order traversal, we first visit the left subtree B. B is also traversed post-order. The process goes on until all nodes are visited. The output of post-order traversal of this tree will be:

D → E → B → F → G → C → A

Example:



Post order :→ 14, 11, 16, 29, 73, 64, 143, 81, 18.

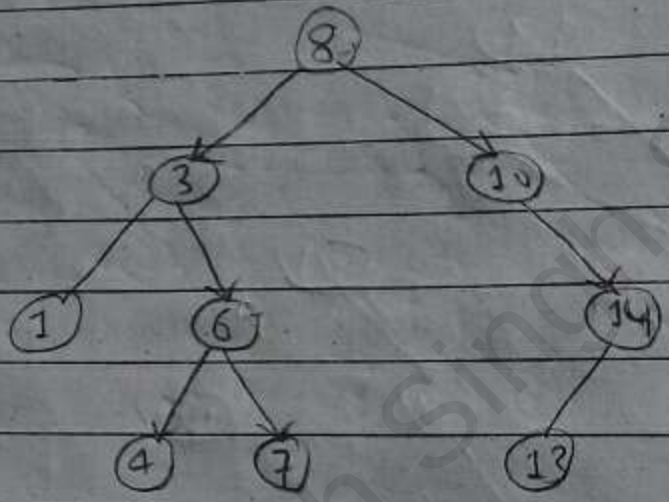
iv) Search in BST:

To search a given value in Binary Search Tree, we first compare it with root:

→ If the target value is equal, the search is successful.
→ If the target value is less, search the left subtree.
→ If the target value is greater, search the right subtree.
→ If the subtree is empty, the search is unsuccessful.

Illustration to search 6 in below tree:

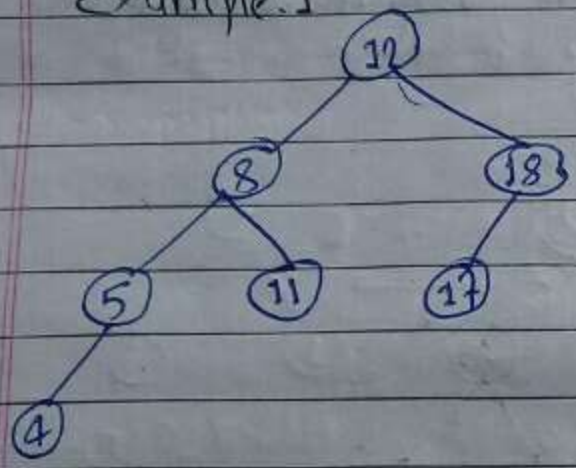
1. Start from the root.
2. Compare the inserting element with root, if less than root, then recurse for left, else recurse for right.
3. If element to search is found anywhere, return true, else return false.



6.6 AVL Tree and Balancing Algorithm:

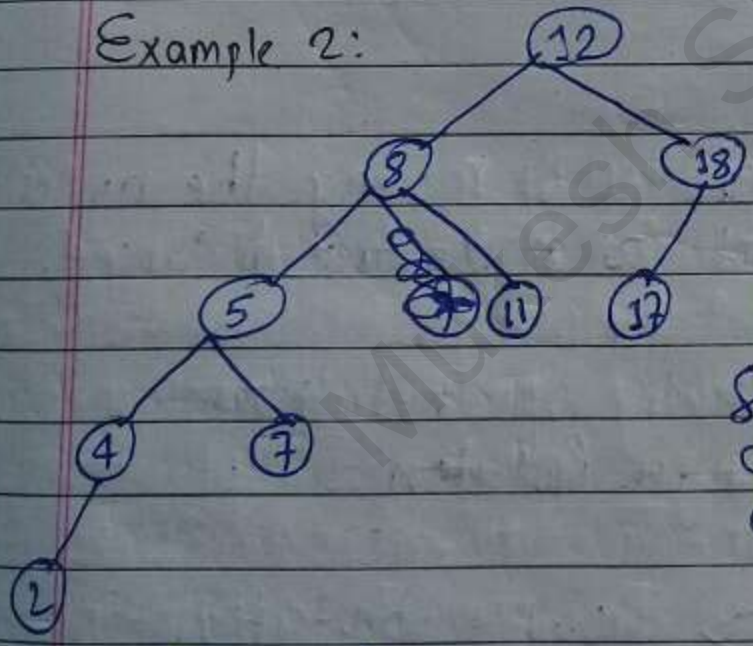
AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

Example: 1



~~It is a b~~
This tree is AVL because differences between heights of left and right subtrees for every node is less than or equal to 1.

Example 2:



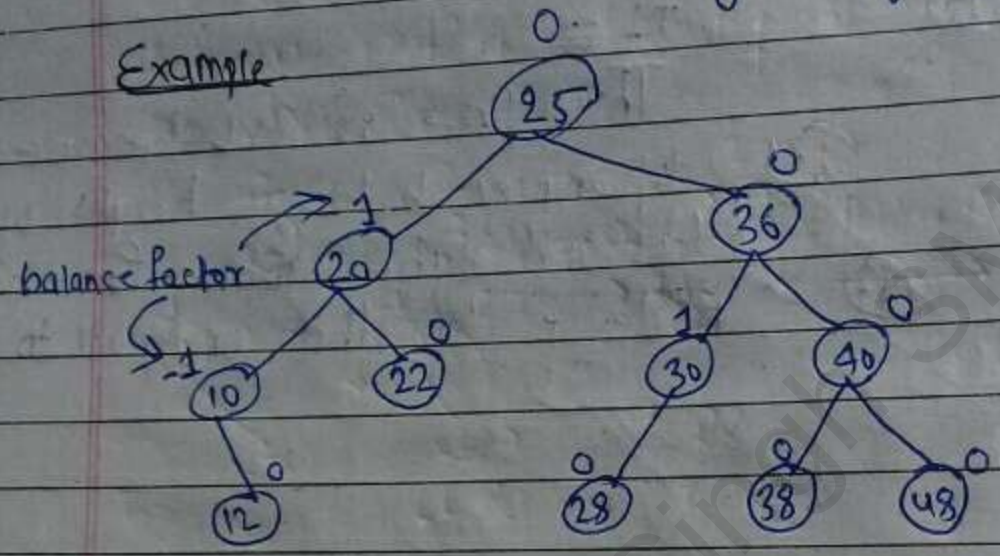
This tree is not AVL because difference between heights of left and right subtrees for 8 and 18 is greater than 1.

The AVL tree was introduced in the year of 1962 by G.M. Adelson-Velsky and E.M. Landis.

An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0, or +1.

Balance factor = height of left subtree − height of Right subtree

Example



AVL Tree Rotations

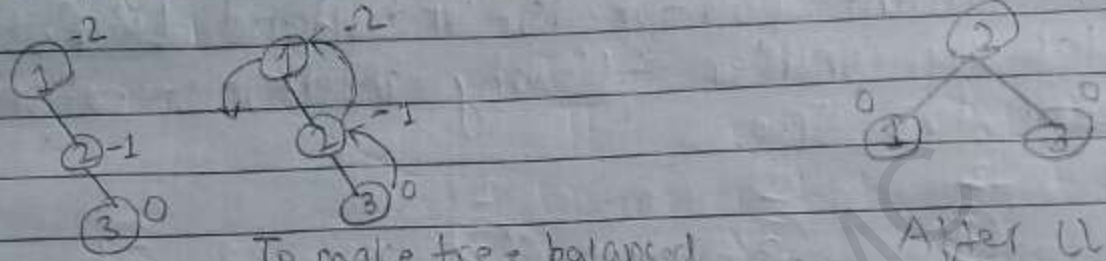
Rotation is the process of moving the nodes to either left or right to make tree balanced.

There are four kinds of rotations:

i) left rotation → Single Rotation
ii) Right rotation
iii) Left-Right rotation → Double Rotation
iv) Right-Left rotation

i) Single Left Rotation (LL Rotation):

In LL rotation, every node moves one position to left from the current position. To understand LL Rotation, let us consider following insertion operations into an AVL Tree:



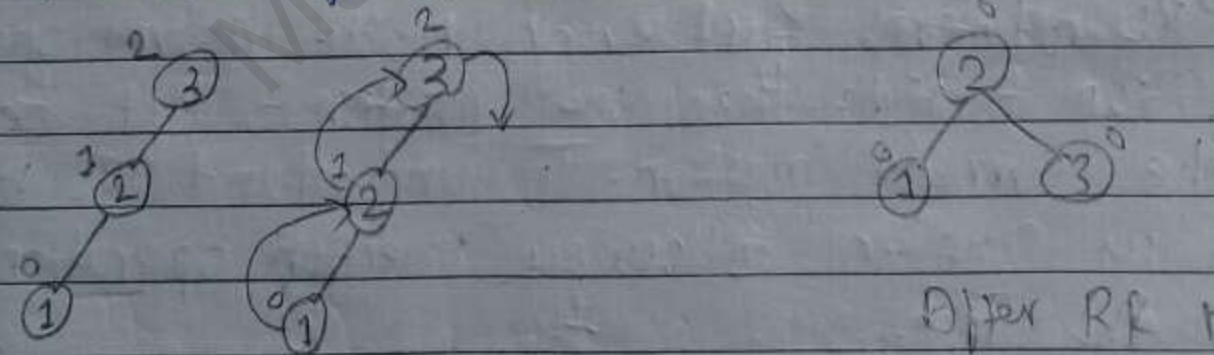
Tree is imbalanced

To make tree balanced we use LL rotation which moves nodes one position to left

After LL rotation, Tree is balanced

ii) Single Right Rotation (RR Rotation)

In RR rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider following insertion operations to an AVL Tree..



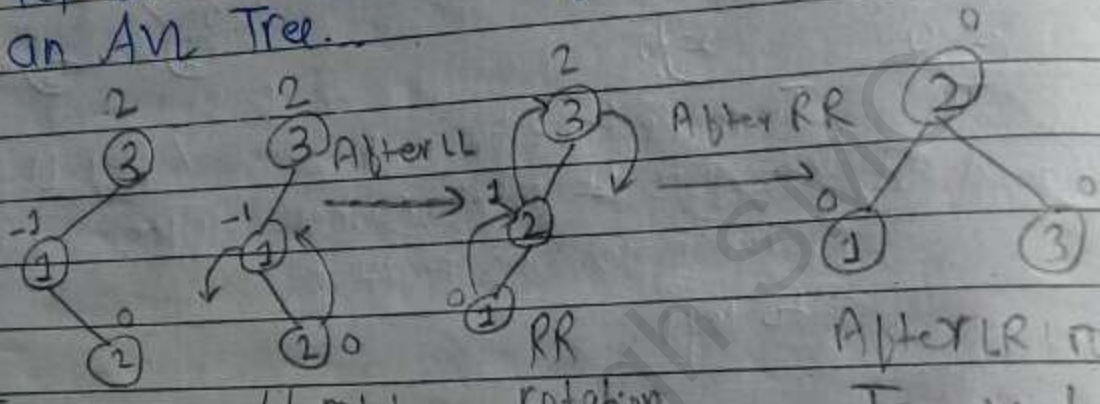
Tree is imbalanced

To make balanced we use RR rotation which moves nodes one position to right

After RR rotation Tree is balanced

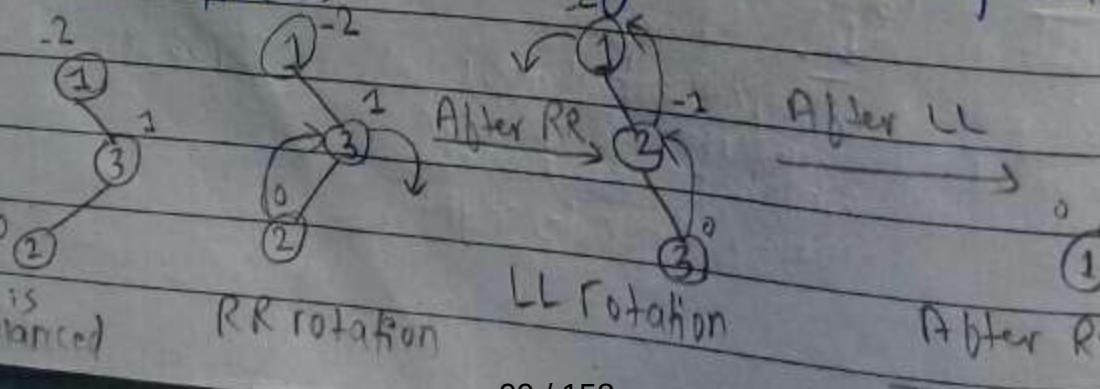
iii) Left Right Rotation (LR Rotation)

The LR rotation is combination of single left rotation followed by ~~LR rotation~~ single right rotation. In LR rotation, first every node moves one position to left then one position to right from the current position. To understand LR Rotation, let us consider following insertion operations into an AVL Tree...



iv) Right Left Rotation (RL Rotation):

The RL rotation is combination of single right rotation followed by single left rotation. In RL rotation, first every node moves one position to right then one position to left from the current position. To understand RL rotation let us consider following insertion operations into an AVL Tree...



6.7. Applications of Tree:

i) Binary search partion:
Used in almost 3D video games to determine what objects need to be rendered.

ii) Hash Trees:
Used in p2p programs and specialized image-signatures in which a hash needs to be verified, but the whole file is not available.

iii) Huffman Coding Tree
Used in comprenion algorithms, such as those used by the ~~for~~ .jpeg and .mp3 file-formats.

iv) Syntax tree
Constructed by compilers and calculators to parse expresions.

v) HTML Document Object Model (DOM):
All html text, attributes are stored in tree called DOM.

vi) Network Routing:
vii) Auto corrector and spell checker

Unit-7

Sorting

7.1 Introduction and Types of Sorting: Internal and External Sort:

Sorting is a process of arranging the items in a list in some order that is either ascending or descending order.

Let a[n] be an array of n elements $a_0, a_1, a_2, a_3 \ldots a_{n-1}$ in memory. The sorting of the array a[n] means arranging the content of a[n] in either increasing or decreasing order.

i.e $a_0 <= a_1 <= a_2 <= a_3 <= \ldots < a_{n-1}$

Consider a list of values: 2, 4, 6, 8, 9, 1, 22, 4, 77, 8, 9
After sorting the values: 1, 2, 4, 4, 6, 8, 8, 9, 9, 22, 77.

Internal Sort:

An internal sort is any data sorting process that takes place entirely within the main memory. This is possible whenever the data to be sorted is small enough to all be held in the main ~~mat~~ memory. For sorting larger datasets, it may be necessary to hold only a chunk of data in memory at a time and the rest of data is normally held on some larger medium like a hard-disk.

Defn- Internal Sorting is a way to first sort the small part of program then merge them all to get the desire result.

or types
Examples of internal Sorting are: Bubble sort, Insertion sort, quick sort, heap sort, radix sort, selection sort etc

External Sort:

External Sorting is a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory, usually a hard-disk. Thus, external memory algorithms are external memory algorithms and thus applicable in the external memory model of computation.

Defn- External sorting is a way to sort the directly our input to give the definite output.

Example or types of external Sorting are - ~~heap~~ merge sort, hybrid sort etc

7.2 Comparison Sorting Algorithms : Bubble ~~Sort~~, Selection and Insertion Sort

1. Bubble Sort:

Bubble sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Example:

First Pass:

(5 1 4 2 8) -> (1 5 4 2 8), Here, algorithm compares first two elements, and swaps since 5>1

(1 5 4 2 8) -> (1 4 5 2 8), Swap since 5 > 4

(1 4 5 2 8) -> (1 4 2 5 8), Swap since 5 > 2

(1 4 2 5 8) -> (1 4 2 5 8). Now, since there elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(1 4 2 5 8) -> (1 4 2 5 8)

(1 4 2 5 8) -> (1 2 4 5 8), Swap since 4 > 2

(1 2 4 5 8) -> (1 2 4 5 8)

(~~1 2~~

Now the array is already sorted, but our ~~order~~ algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

Third pass:
(1 2 4 5 8) -> (1 2 4 5 8)
(1 2 4 5 8) -> (1 2 4 5 8)
(1 2 4 5 8) -> (1 2 4 5 8)
(1 2 4 5 8) -> (1 2 4 5 8)

Algorithm:
begin bubbleSort (list)
for all elements of list
if (list[i] > list[i+1]
swap (list[i], list[i+1])
end if
end for
return list
end BubbleSort

2. Selection Sort:

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

i) The subarray which is already sorted.
ii) Remaining subarray which is unsorted.

Bubble Sort Example 2:

Initially

25	57	48	37	12	92	86	33
0	1	2	3	4	5	6	7

After Pass 1:

25	48	37	12	57	33	86	92
0	1	2	3	4	5	6	7

After pass 2

25	37	12	48	57	33	86	92
0	1	2	3	4	5	6	7

After pass 3:

25	12	37	48	33	57	86	92
0	1	2	3	4	5	6	7

After pass 4:

12	25	37	33	48	57	86	92
0	1	2	3	4	5	6	7

After Pass 5:

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

After Poss 6:

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

After Pass 7:

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

2. Selection Sort:

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The Algorithm maintains two subarrays in a given array.

i) The sub-array which is already sorted

ii) Remaining sub-array which is unsorted.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where n is the number of items.

Example:

Consider the following depicted array as an example:

14	33	27	10	35	19	42	44

Example

Assume that the array A = [7, 5, 4, 2] needs to be sorted in ascending order.

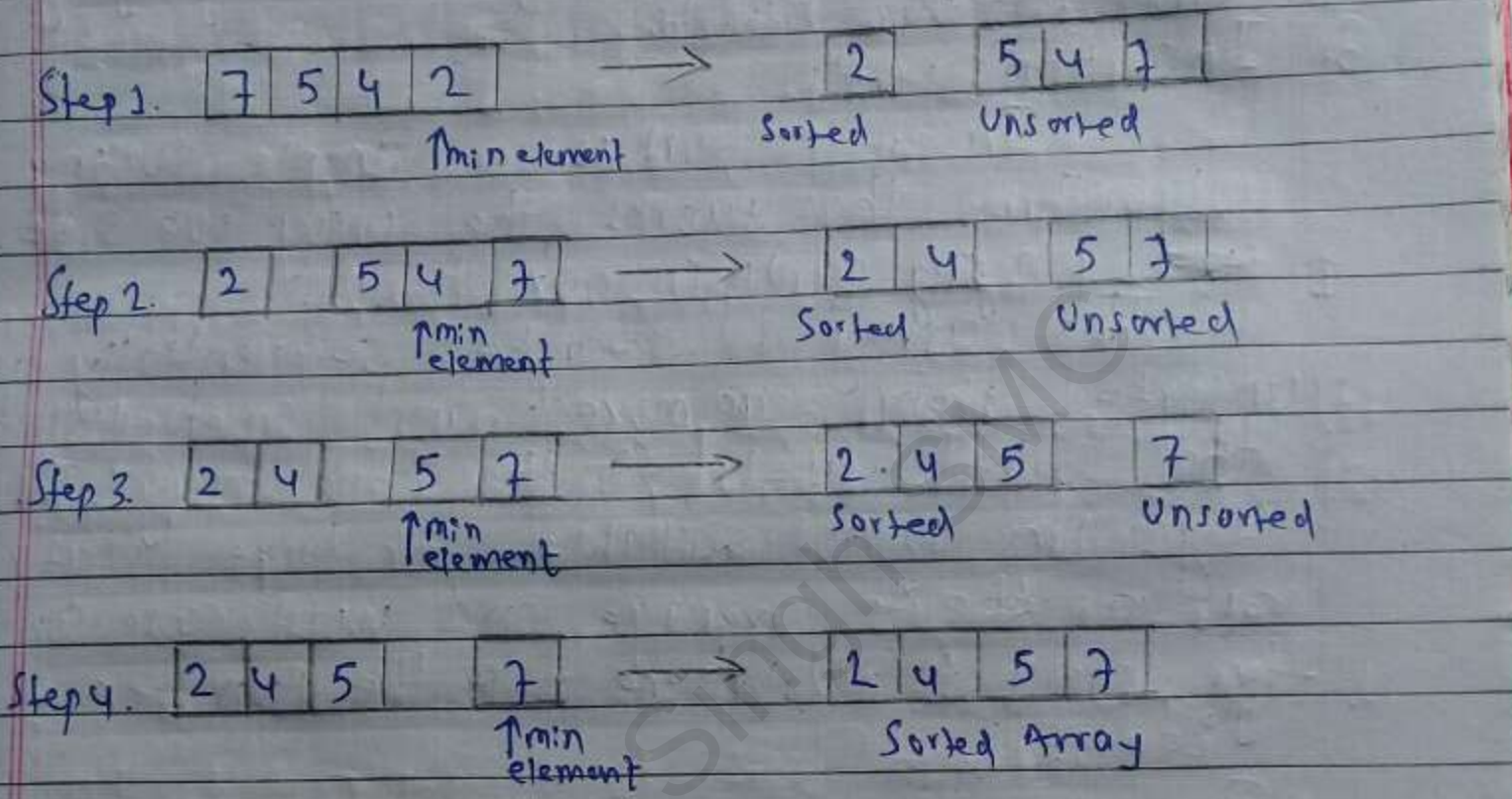


fig: Selection Sort

Algorithm

Step 1: Set MIN to location 0
Step 2: Search the minimum element in the list.
Step 3: Swap with value at location MIN.
Step 4: Increment MIN to point to next element.
Step 5: Repeat until list is Sorted.

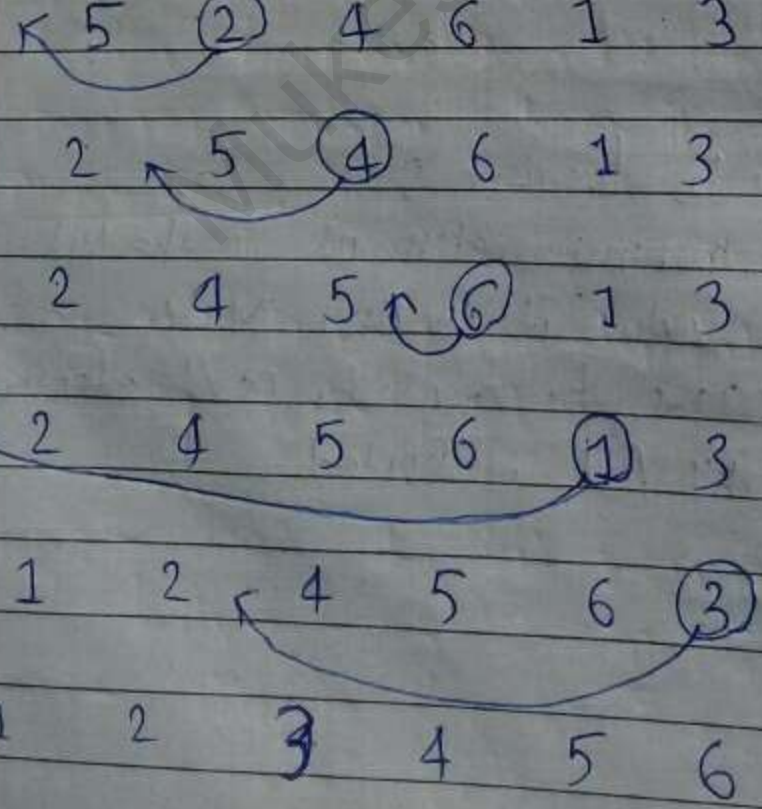
3. ~~Insertion Sort:~~
~~Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.~~

3. Insertion Sort:

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where n is the number of items.

Example:



Algorithm for insertion Sort.

Step 1 - If it is the first element, it is already sorted. return 1;

Step 2 - Pick the next

Step 3 - Compare with all elements in the sorted sub-list.

Step 4 - Shift all the elements in the sorted sub-list that is greater than the value to be sorted.

Step 5 - Insert the value.

Step 6 - Repeat until list is sorted.

7.3 Divide and Conquer Sorting: Merge, Quick and Heap Sort

Divide and Conquer is an algorithmic paradigm. A typical Divide and Conquer algorithm solves a problem using following three steps:

i) Divide: Break the given problem into subproblems of same type.
ii) Conquer: Recursively solve these subproblems.
iii) Combine: Appropriately combine the answers.

1. Merge Sort:

Merge sort is a sorting algorithm based on divide and conquer technique. The algorithm divides the array into two equal halves, recursively sorts them and finally merges/combines them in a sorted manner.

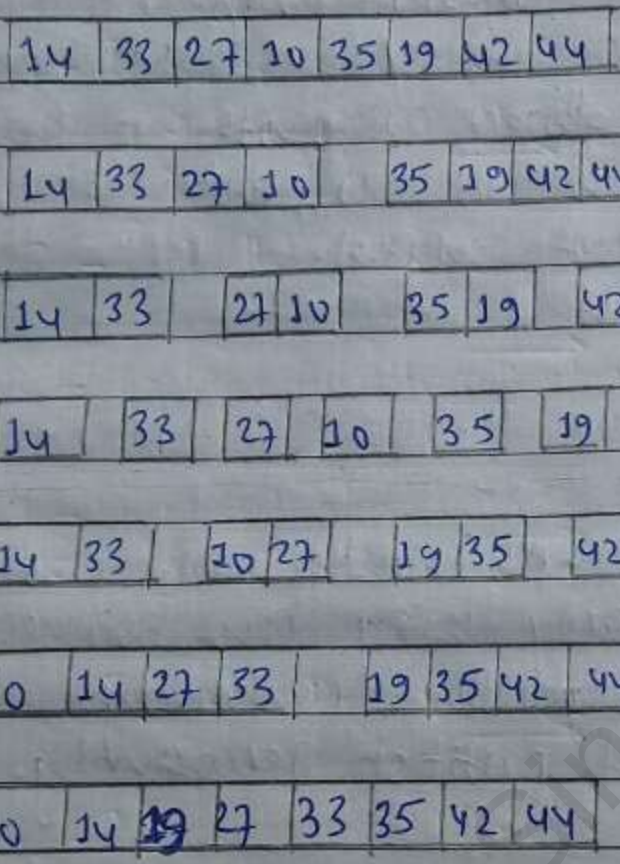
Algorithm

Step 1- If it is only one element in the list it is already sorted, return.

Step 2- divide the list recursively into two halves until it can no more be divided.

Step 3- Merge the smaller lists into new list in sorted order

Example:



2. Quick Sort:

Quick sort is based on the divide and conquer approach based on the idea of choosing one element as a pivot element and partioning the array around it such that: left side of pivot contains all the elements that are less than the pivot element. Right side contains all elements greater than the pivot.

It reduces the space complexity and removes the use of the auxiliary array that is used in merge sort. Selecting a random pivot in array results in improved time complexity in most of the cases.

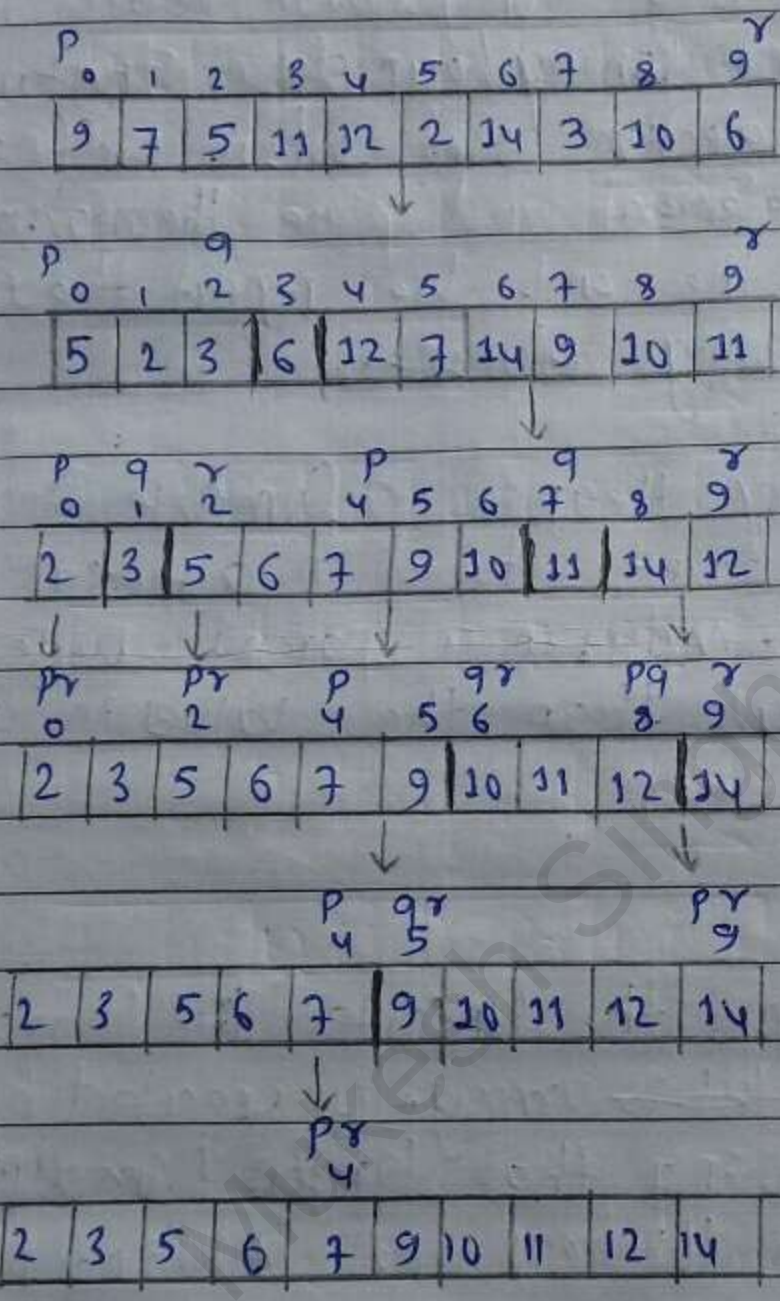
Example: There are different versions of quicksort that pick pivot in different ways:

1. Always pick first element as pivot.
2. Always pick last element as pivot.
3. Pick a random element as pivot.
4. Pick median as pivot.

Algorithm

Step 1 - Make the right-most index value pivot.
Step 2 - Partition the array using pivot value.
Step 3 - quicksort left partition recursively.
Step 4 - quicksort right partition recursively.

Example



3. Heap Sort:

Heap sort is a comparison based sorting technique based on Binary heap data structure. It is similar to selection sort where we find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

Binary Heap:

A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater (or smaller) than the values in two children nodes.

Algorithm:

1- Creating a Heap of the unsorted list / array.
2. - Then a sorted array is created by repeatedly removing the largest / smallest element from the heap and inserting it into the array.
3. Heap is reconstructed after each removal.

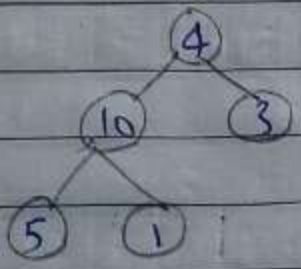
Example Index

	0	1	2	3	4
Input Data	4	10	3	5	1

Build Heap

Step 2

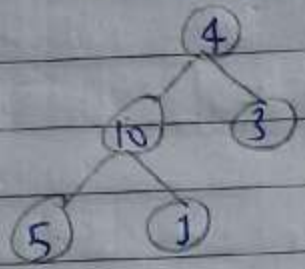
Index	0	1	2	3	4
Data	4	10	3	5	1



Step 3

Index	0	1	2	3	4
Data	4	10	3	5	1

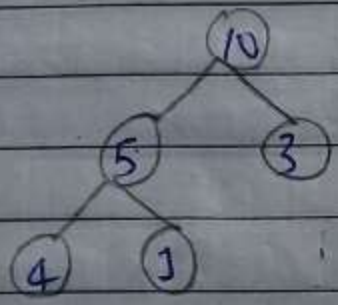
Create max heap



- 10 is greater than 4
- Swap 4 & 10.

Step 4

Index	0	1	2	3	4
Data	10	5	3	4	1

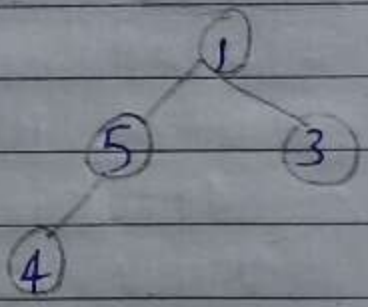


5 is greater than 4

Swap 5 and 4

Step 5

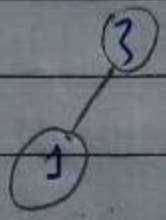
Index	0	1	2	3	4
Data	1	5	3	4	10



Step 6

Index	0	1	2	3	4
Data	1	3	4	5	10

Remove the node



Unit-8

Searching

8.1 Introduction to Searching:

Searching is a process of finding an element within the list of elements stored in any order or randomly.

8.2 Search Algorithms: Sequential Search, Binary Search

1 Sequential Search:

In linear / Sequential search, access each element of an array one by one sequentially and see whether it is desired element or not. A search will be unsuccessful if all the elements are accessed and desired element is not found.

In brief, simply search for the given element left to right and return the index of the element if found. Otherwise, return "Not found".

Algorithm

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with the first element in the list.

Step 3 - If both are matching, then display

"Given Element found!!!" and terminate the function.

Step 4 - If both are not matching, then compare search element with the next element in the list.

Step 5 - Repeat step 3 and 4 until the search element is compared with the last element in the list.

Step 6 - If the last element in the list also doesn't match, then display "Element not found!!!" and terminate the function.

Example:

Consider the following list of element and search element:

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

Search element 12

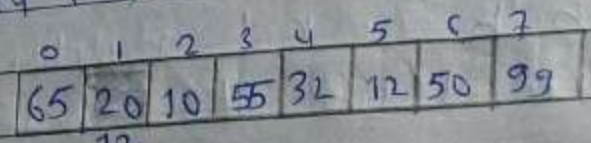
Step 1:

Search element (12) is compared with first element (65)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
	12							

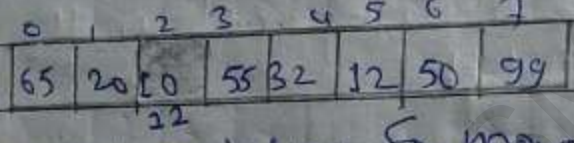
Both are not matching. So move to next elements.

Step 2:
Search element (12) is compared with next element (20)



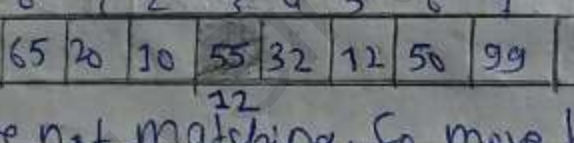
Both are not matching. So move to next element

Step 3: Search element (12) is compared with (10)



Both are not matching. So move to next element.

Step 4:
Search element (12) is compared with next element (55)

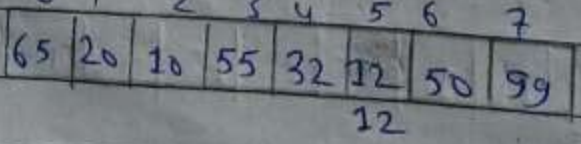


Both are not matching. So move to next element.

Step 5:
Search element (12) is compared with next element (32)

Both are not matching. So move to next element.

Step 6:
Search element (12) is compared with next element (12)



Both are matching. So we stop comparing and display element found at index 5.

2. Binary Search:

Binary search, also known as half-interval search or logarithmic search, is a search algorithm that finds the position of a target value within a sorted array.

Binary search is an efficient algorithm for finding an item from a sorted list of items. It works repeatedly dividing in half the portion of the list that could contain the item, until you have narrowed down the possible locations to just one.

Algorithm

1. Let min = 1 and max = n.
2. Guess the average of max and min, rounded down so that it is an integer.
3. If you guessed the number, stop. You found it!
4. If the guess was too low, set min to be one larger than the guess.
5. If the guess was too high, set max to be one smaller than the guess.
6. Go back to step 2.

Example 1.
If searching element 23 in the 10 element array.

	2	5	8	12	16	23	38	56	72	91
	L									H
23>16, take 2nd half	2	5	8	12	16	23	38	56	72	91
						L				H
23<56, take 1st half	2	5	8	12	16	23	38	56	72	91
						L	H			
Found 23. return 5.	2	5	8	12	16	23	38	56	72	91

Example 2
Consider the list of element and search element.

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

Search element 12

Step 1:
Search element (12) is Compared with middle element (50)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99
					12				

Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

Step 2:

Search element (12) is compared with middle element (12)

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99
	12							

Both are matching. So, the result is "Element found at index 1".

8.3 Efficiency of Searching Algorithms:

A. Efficiency of Linear Search:

i) In best case, Number of comparison = 1 = O(1)

ii) In worst case, Number of comparison = n
i.e. O(n)
Data is not present in the list i.e number of comparison will be n.

iii) For average case,

$$Avg = \frac{n+1}{2} = 0.5n + 3$$

$$= O(n/2)$$

Here, we assume that item does not appear, and that is equally likely to occur at any position in the array. According to the number of comparisons that can be any number 1, 2, 3 .... n and each number occurs with the probability $p = 1/n$. Then

$$T(n) = 1.(1/n) + 2(1/n) + 3(1/n) + \dots + n(1/n)$$
$$= (1 + 2 + 3 + \dots + n) \times (1/n)$$
$$= n(n+1)/2 \times (1/n)$$
$$= (n+1)/2$$
$$= O((n+1)/2)$$

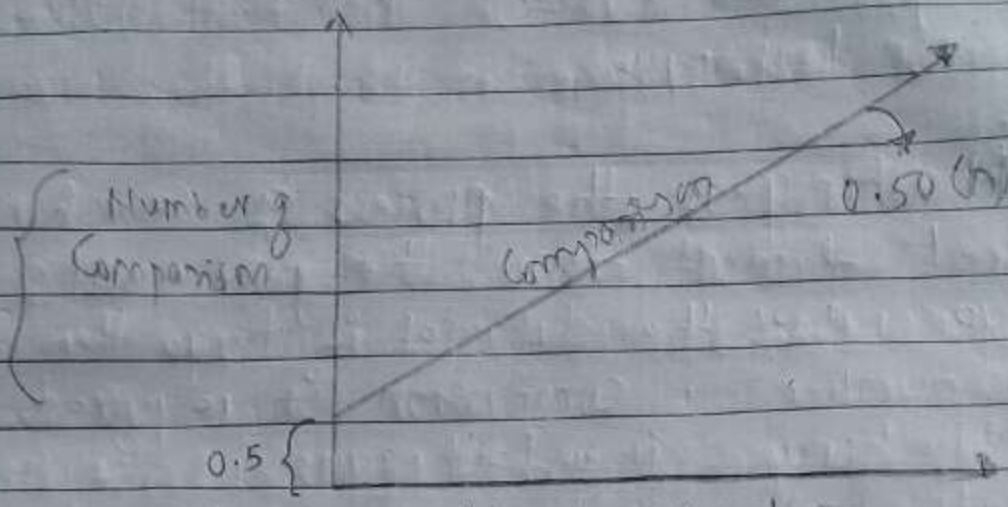


Fig: Example of linear Search

B. Efficiency of Binary Search:

If length of list is 8,
Maximum Comparison = 3+1
If length of list is 64,
Maximum comparison = 6+1
Similarly, for n size list
$n = 2^N$
Number of Comparison = N+1

$\therefore O(N) = \log_2 n$

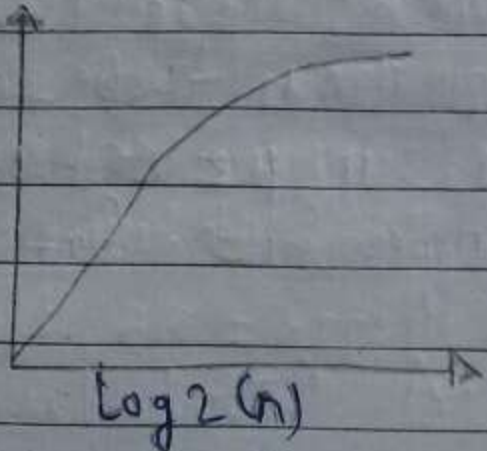


Fig: Binary searching

8.4. Hashing: Hash function and hash tables, Collision resolution technique

• Hashing is the process of searching the desired element from the given set of data. In general, if the data set is huge then the high number of comparison is required.

Hashing is a technique of finding a desired number from a given set of data with a minimum number of comparisons.

Hash function:

Hash function is a function which ~~maps~~ takes a piece of data (i.e key) as input and outputs an integer (i.e. hash value) which maps the data to a particular index in the hash table.

The desired property of hash function is those that compute and minimize the number of collisions.

• Perfect Hash function is a function which when applied to all the members of the set of items to be stored in a hash table, produces a unique set of integers within some suitable range. Such function produces no collisions.

• A Good Hash function minimizes collisions

by spreading the elements uniformly throughout the array.

Load factor (λ) = n/k

Where
n is the number of entries;
k is the number of buckets.

Characteristics of Good Hashing:
i) The hash value is fully determined by the data being hashed.

ii) The has function uses all the input data.
iii) The hash function uniformly distributes the data across the entire set of possible hash values.
iv) The hash function generates very different values for similar strings.

Types of Hash function:

1. Division Method:

The key is mapped by a key k into one of m slot by taking the remainder of k divided by m. The hash function is

$h(k) = k \bmod m$ or $h(k) = k \bmod m + 1$.

For example

If the hashtable has size m = 12 and the key is k = 100, then h(k) = 4.

2. Multiplication method:
In multiplication method, there are two steps:
First:
Multiply a key k by a constant A in the range $(0 < A < 1)$
Second:
Extract the fractional part of kA, multiply this value by m and take the floor of the result.
$h(k) = m(kA \bmod 1)$

3. Mid Square Method:
In mid square method the key value is square, after getting number. We take some digits from the middle of that number as an address.

4. Folding Method:
One of the ~~easiest~~ easiest way to compute the key is break the key into pieces, add them and get the hash address.
82394561, 87139465, 83567271, 85943228
Now chop them 3,2 and 3 digits
$82394561 = 823 + 94 + 561 = 1478$
$87139465 = 871 + 39 + 465 = 1375$
$83567271 = 835 + 67 + 271 = 1173$
$85943228 = 859 + 43 + 228 = 1130$

Now truncate them up to the digit based on the size of hash table. Suppose the table size is 1000, so hash address can be from 0 to 999. So we will truncate the higher digit of number.

h(82394561) = 478
h(87139465) = 375
h(83567271) = 173
h(85943228) = 130

Hash Tables

Hash table is a data structure which store data in associative manner. In hash table, data is stored in array format where each data value have its own unique index value. Access of data becomes very fast if we know the index of desired data.

- A hash table consists of an array in which data is accessed via a special index called a key.
- The primary idea behind the hash table is to establish a mapping between the set of all possible keys and position in the array using hash function. A hash function accepts a key and returns its hash coding, hash value. Keys ~~are~~ vary in type, but hash coding are always integers.
- When hash functions guarantee that no two

keys will generate the same hash coding, the resulting hash table is said to direct address.
• When two keys maps to the same position, they collide.
• A good hash function minimizes the collisions.

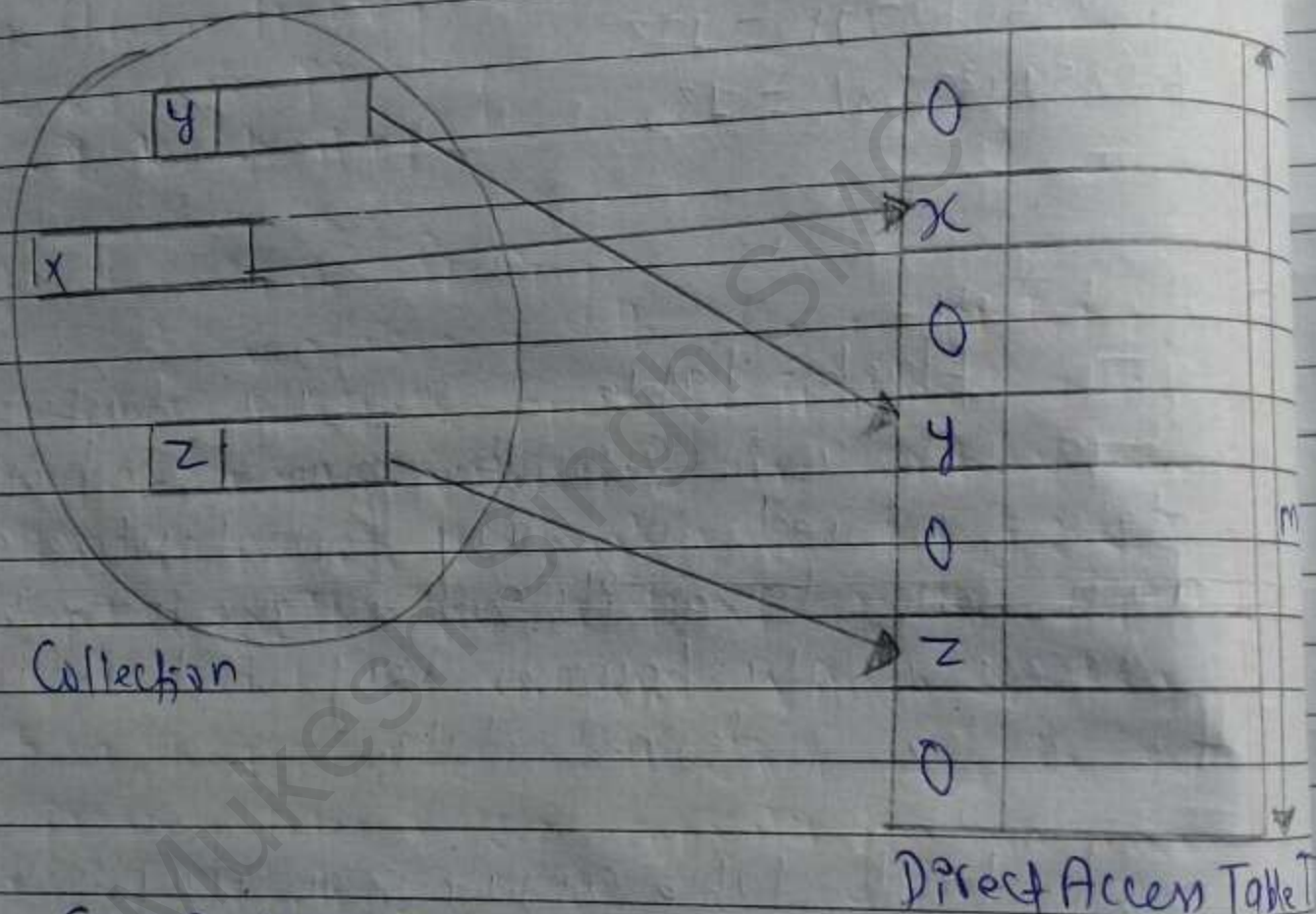


Fig: A Hash Table

Applications of Hash Tables:
i) Database System
ii) Symbol Tables
iii) Tagged Buffer
iv) Data Dictionaries

Hash Collision ~~Resolution Technique~~:

A hash collision is said to occur, when two or more than two identical identifiers are hashed into the same bucket, when the bucket size is 1. Collision and overflow occur simultaneously. Hash collision are practically unavoidable when hashing a random subset of a large set of possible keys. Therefore, almost all hash table implementation have some collision resolution strategy to handle such events.

Hash Collision Resolution Technique:

The collision may occur when more than one keys map to same hash value in the hash table. There are ~~two~~ types of hash collision resolution techniques:

1. Open Addressing (Closed Hashing)
2. Separate Chaining (Open Hashing)

1. ~~Open Addressing~~: When two ~~or more~~ items hash to the same slot, we must have a systematic method for placing the second item in the hash table. This process is called collision resolution.

1. Open Addressing (Closed Hashing):

In this technique, a hash table with pre-identified size is considered. All items are stored in the hash table itself. In addition to

the data, each hash bucket also maintains three states: EMPTY, OCCUPIED, DELETED. While inserting if a collision occurs, alternative cells are tried until an empty bucket is found. for which one of the following technique is adopted:

a. Linear Probing
b. Quadratic probing
c. Double hashing

b. Quadratic Probing:

A variation of the linear probing is called quadratic probing. instead of using a constant "skip" value, we use a rehash function that increments the hash value by 1, 3, 5, 7, 9 and so on. This means that if the first hash value is h, the successive values are h+1, h+4, h+9, h+16, and so on. In other words, quadratic probing uses a skip consisting of successive perfect squares.

The below figure shows example values after they are placed using this technique.

0	1	2	3	4	5	6	7	8	9	10
77	44	20	55	26	93	17	None	None	31	54

2 Separate Chaining (Open hashing)

An alternative method for handling the collision problem is to allow each slot to hold a reference to a collection (or chain) of items. Chaining allows many items to exist at the same location in the hash table. When collisions happen, the item is still placed in the proper slot of the hash table. As more and more items has to the same location, the difficulty of searching for the item in the ~~list~~ collection increases.

Below figure shows chaining to resolve collisions:

Unit-9

Graphs

9.1. Definition and Representation

Definition

Graph is a non linear data structure, which contains a set of points known as nodes (or vertices) and set of links known as edges (or Arcs) which connects the vertices. A graph is defined as follows.

- Graph is a collection of vertices and arcs which connects vertices in the graph.
- Graph is a collection of nodes and edges which connects nodes in the graph.

Generally, a graph (G) is represented as $G = (V, E)$, where 'V' is set of vertices and 'E' is set of edges.

Example

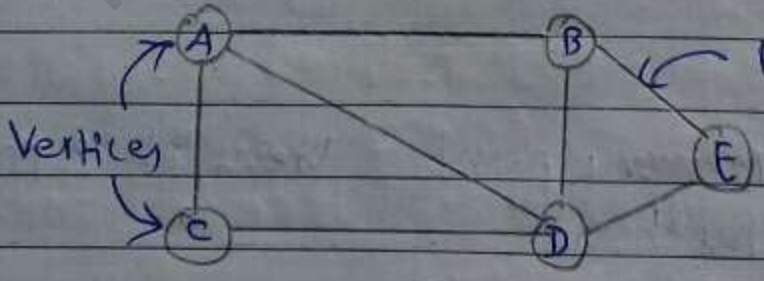


Fig: Graph

Representation of Graphs:

Usually graph can be represented in many ways. Some of these representations are:

1. Set Representation:

In this representation, two sets are maintained. They are:

i) Set of vertices, V and

ii) Set of edges, E, which is the subset of V x V.

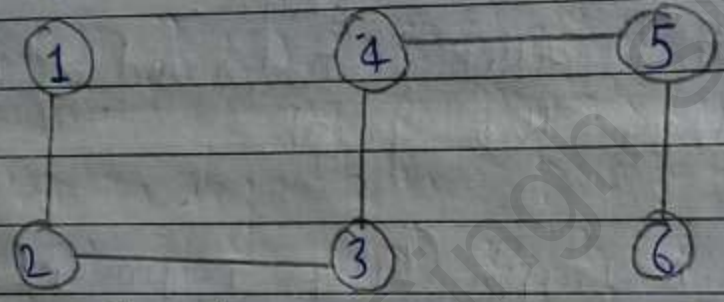


Fig: Representing a graph

The figure above has,

$V(G) = \{1, 2, 3, 4, 5, 6\}$

$E(G) = \{(1,2), (2,3), (3,4), (4,5), (5,6)\}$

Advantage:

* From the memory point of view it is most efficient method.

Disadvantage:

- This representation does not allow storing of parallel edges in a multigraph.
- Multiplication of the graph is difficult.

2. Sequential Representation of Graph:

The graphs can be represented a matrix in Sequential representation.

There are ~~two~~ 3 most common matrices:

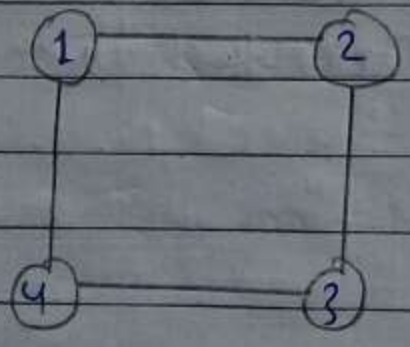
a. Adjacency matrix
b. Incidence Matrix
c. Adjacency list

a. Adjacency Matrix:

In this representation, graph can be represe-nted using a matrix of size total number of vertices by total number of vertices (i-e V×V). That means if a graph with 4 vertices can be represented using a matrix of 4×4 class. In this matrix, rows and Column represents vertices. This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

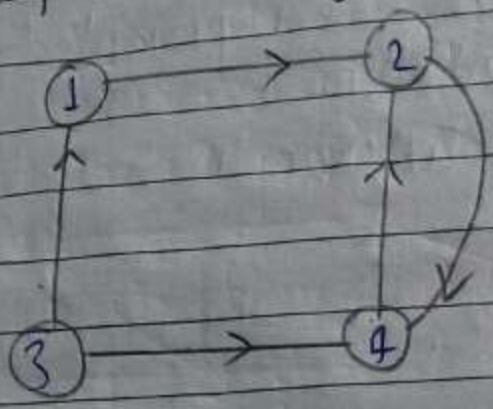
Example

1. Using undirect matrix



i/j	1	2	3	4
1	0	1	0	1
2	1	0	1	0
3	0	1	0	1
4	1	0	1	0

Example 2: Using directed graph:



i\j	1	2	3	4
1	0	1	0	0
2	0	0	0	1
3	1	0	0	1
4	0	1	0	0

b) Incidence matrix:

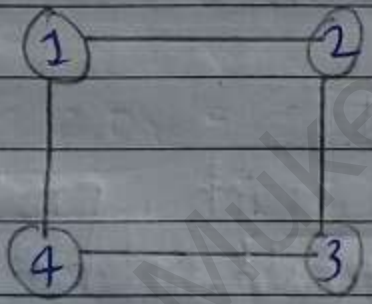
In this representation, graph can be represented using a matrix of size total number of vertices by total number of edges. That means if a graph with 4 vertices and 6 edges can be represented using a matrix of 4x6 class. In this matrix, rows represents vertices and columns represents edges. This matrix is filled with either 0 or 1 or -1. Here, 0 represents row edge is not connected to column vertex, 1 represents row edge is connected as outgoing edge to column vertex and -1 represents row edge is connected as incoming edge to column vertex.

Example

c) Adjacency list:

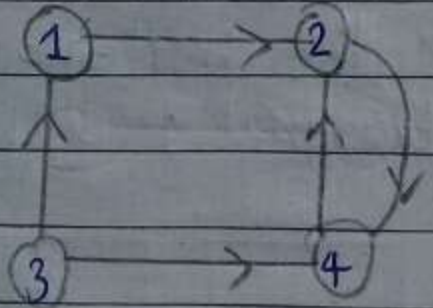
In this representation, every vertex of graph contains list of its adjacent vertices.

Example 1: Undirected graph



A1 → 2 → 4
A2 → 1 → 3
A3 → 2 → 4
A4 → 1 → 3

Example 2: Using directed graph



A1 → 2
A2 → 4
A3 → 1 → 4
A4 → 2

9.2 Graph Traversal: BFS and DFS

Traversing is the process of visiting each node in a graph in some systematic approach / order.

Graph traversal is technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices to be visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops that means using graph traversal we visit all vertices of graph without getting into looping path.

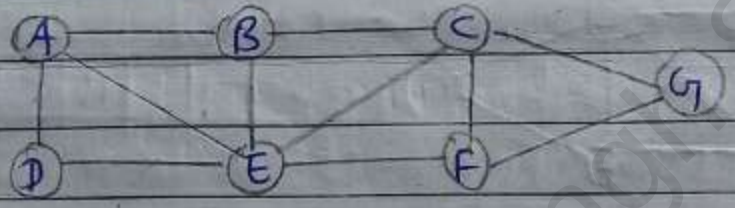
There are two graph traversal techniques and they are described below:

1. Breadth first search (BFS):

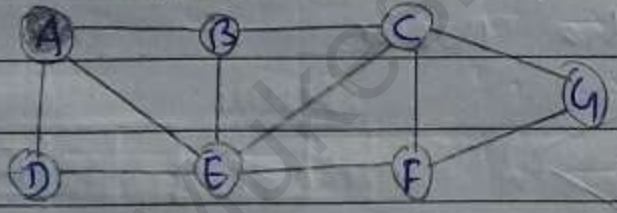
BFS traversal of a graph, produces a spanning tree as final result. Spanning tree is a graph without any loops. ~~We use~~ The aim of BFS algorithm is to traverse the graph as close as possible to the root node. Queue data structure is used in the implementation of the breadth first search traversal of a graph.

Algorithm:
Step 1: Push the root node in the queue.
Step 2: Loop until the queue is empty.
Step 3: Remove the node from the queue.
Step 4: If the removed node has unvisited child nodes, mark them as visited and insert the unvisited children in the queue.

Example
Consider the following example graph to perform BFS traversal.



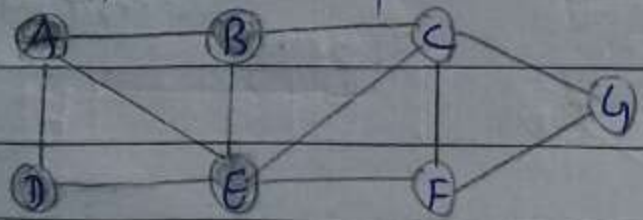
Step 1:
- Select the vertex A as starting point (visit A)
- Insert A into the queue.



Queue

A							

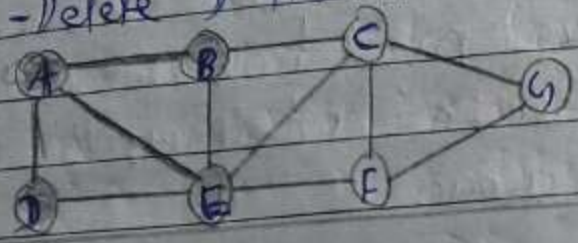
Step 2:
- Visit all adjacent vertices of A which are not visited (D, E, B).
- Insert newly visited vertices into the queue and delete A from the queue.



Queue

	D	E	B				

Step 3:
- Visit all adjacent vertices of D which are not visited (there is no vertex)
- Delete D from the queue.



Queue

	E	B				

Step 4:
- Visit all adjacent vertices of E which are not visited (C, F)
- Insert newly visited vertices into the queue and delete E from queue.

Queue

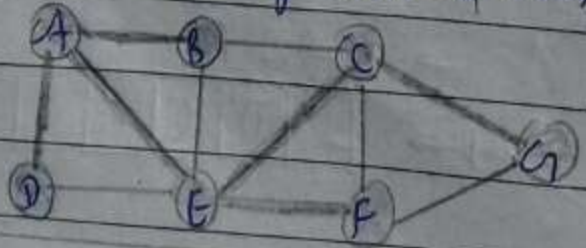
			B	C	F	

Step 5:
- Visit all adjacent vertices of B which are not visited (there is no vertex)
- Delete B from the queue.

Queue

				C	F	

Step 6:
- Visit all adjacent vertices of C which are not visited (G)
- Insert newly visited vertex into the queue and delete C from queue.

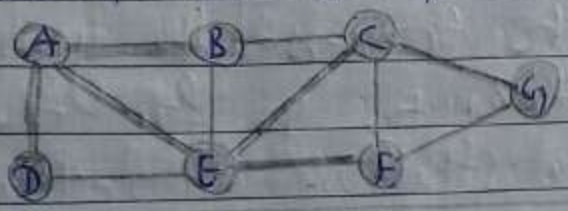


Queue

					F	G

Step 7:

- Visit all adjacent vertices of F which are not visited (There is no vertex).
- Delete F from the queue.



Queue

G						

Step 8:

- Visit all adjacent vertices of G which are not visited (there is no vertex).
- Delete G from the Queue.

Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is Spanning Tree as shown below.

2 Depth first Search (DFS)

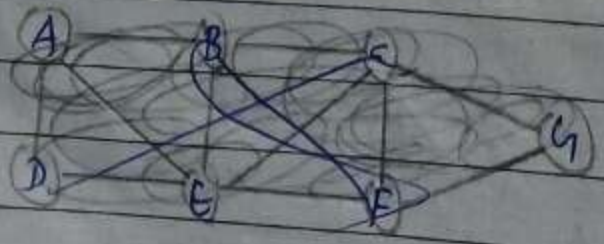
DFS traversal is a graph, produces a spanning tree as a final result. Spanning Tree is a graph without any loops. The aim of DFS algorithm is to traverse the graph in such a way that it tries to go far from the root node. Stack data structure is used in the implementation of DFS traversal.

Algorithm

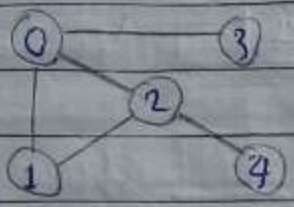
Step 1: Push the root node in the Stack.
Step 2: Loop until stack is empty.
Step 3: Peek the node of the stack.
Step 4: If the node has unvisited child nodes, get the unvisited child node, mark it as traversed and push it on stack.
Step 5: If the node does not have any unvisited child nodes, pop the node from the stack.

Example:

Consider the following example graph to perform DFS traversal.

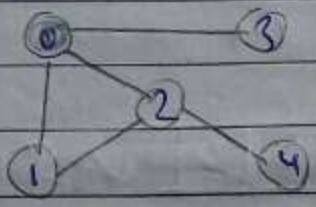


We use an ~~direct~~ unchrected graph with 5 vertices



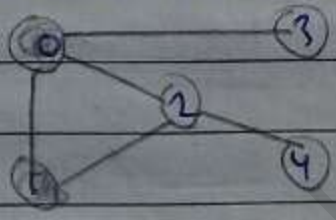
						visited
						stack

We start from vertex 0, the DFS algorithm starts by putting it in the visited list and putting putting all adjacent vertices in the stack.



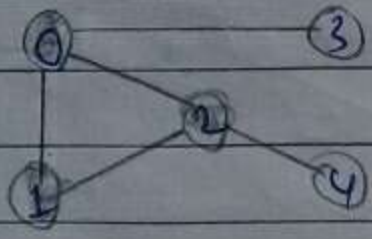
0					visited
1	2	3			stack

Next, we visit the element at the top of the stack i-e 1 and go its adjacent nodes.
Since 0 has already been visited, we visit 2 instead.

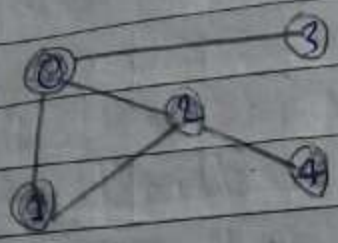


0	1				visited
2	3				stack

Vertex 2 has an unvisited adjacent vertex in 4, so we add that ~~the~~ to the top of the stack and visit it.



0	1	2			visited
4	3				stack



0	1	2	4			Visited

3					Stack

After we visit last element 3, it does not have any unvisited adjacent nodes, so we have completed the Depth first Search Traversal of graph

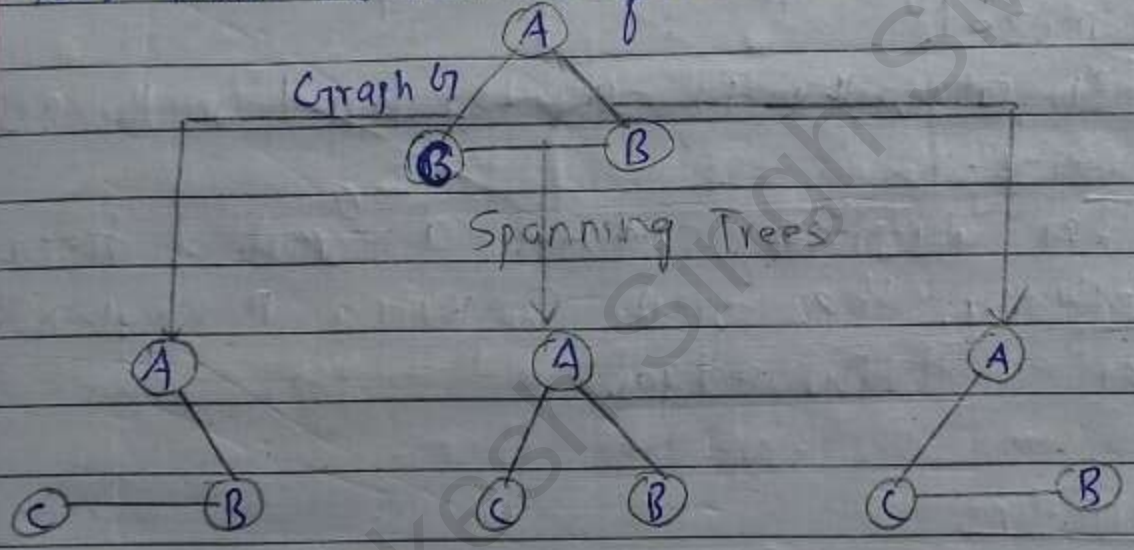
0	1	2	4	3	visited

					Stack

9.3 Minimum Spanning Trees: Kruskal and Prims Algorithm.

Spanning Tree

A spanning tree of a graph is just a sub-graph that contains all the vertices and is a tree. A spanning tree of a connected graph G contains all the vertices and has the edges which connect all the vertices. So, the number of edges will be one less than the number of nodes.



Minimum Spanning Trees:

A minimum spanning tree is a special kind of tree that minimizes the lengths (or weights) of the edges of the tree.

There are number of techniques for creating a minimum spanning tree for a weighted graph but the most famous methods are:

a. Kruskal's Algorithm
b. Prim's Algorithm

a) Kruskal's Algorithm:

Kruskal's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected weighted graph.

It finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

The algorithm is directly based on the MST Property.

- The algorithm creates a forest of trees.
- Initially, the forest consists of n single node trees and no edges.

Algorithm

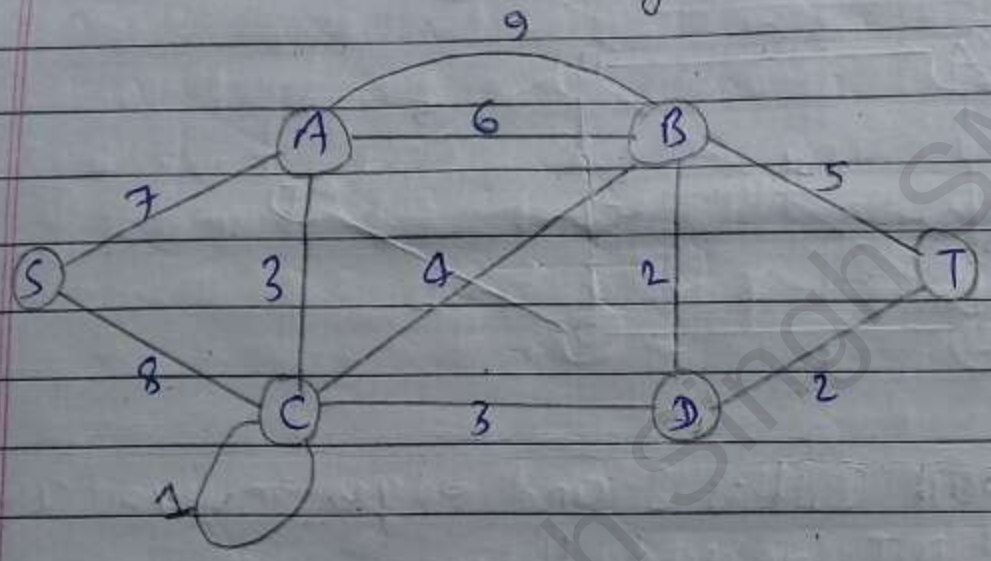
1. Sort all the edges in increasing order of their weight.
2. Take the edge with lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
3. Keep adding edges until we reach all vertices.

OR

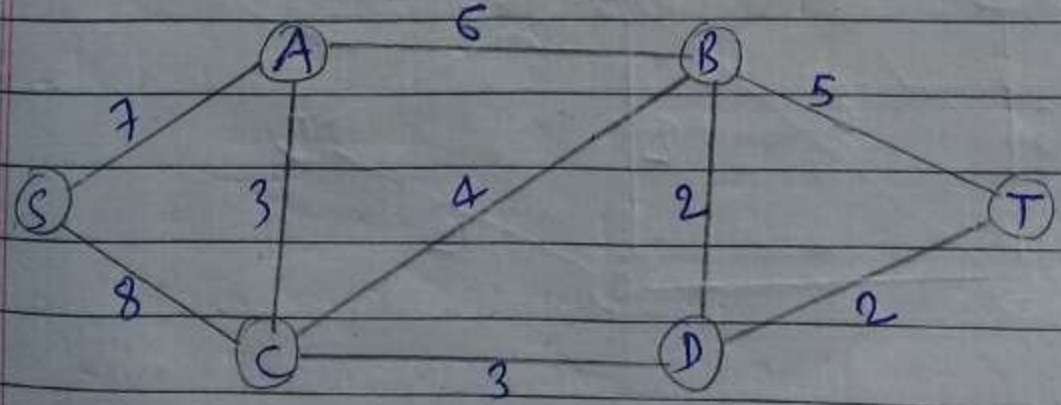
1. Remove all loops and parallel edges.
2. Arrange all the edges in ascending order of cost.
3. Add edges with least weight.

Example 1

To understand Kruskal's algorithm, let us consider the following example:



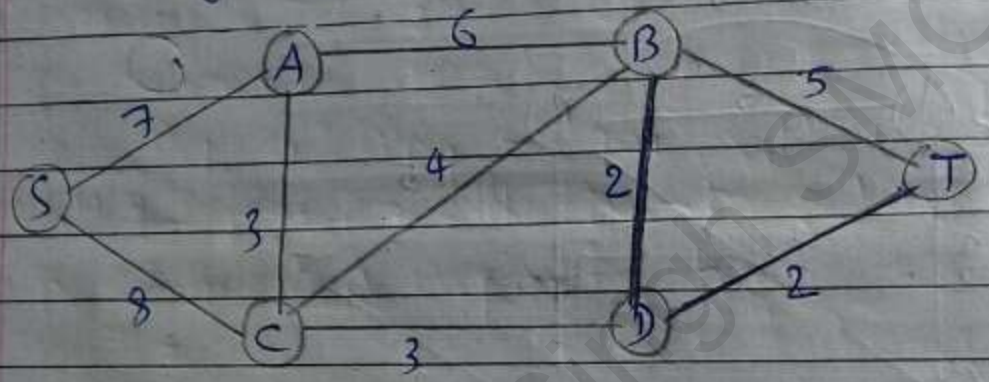
Step 1 Remove all loops and parallel edges. ~~Keep one which has the~~ in case of parallel edges keep one which has the least cost associated and remove all others.



Step 2- Arrange all edges in their increasing order of weight.

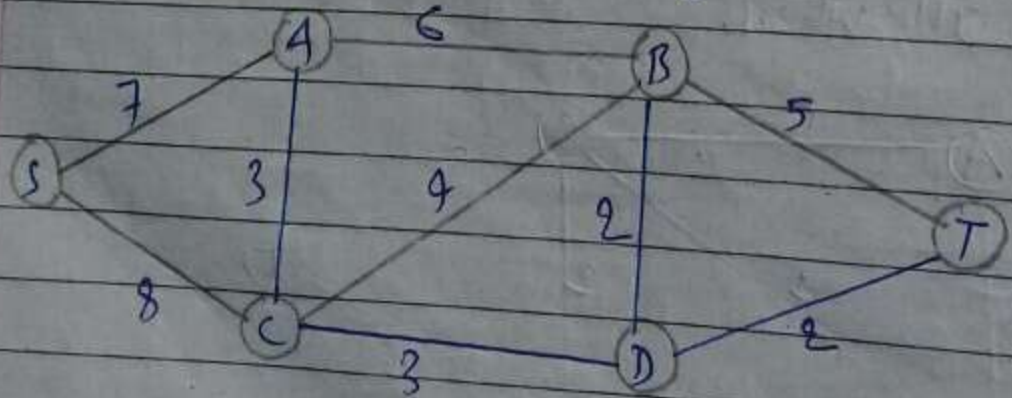
B,D	D,T	A,C	C,D	C,B	B,T	A,B	S,A	S,C
2	2	3	3	4	5	6	7	8

Step 3- Add the edge which has the least weightage.

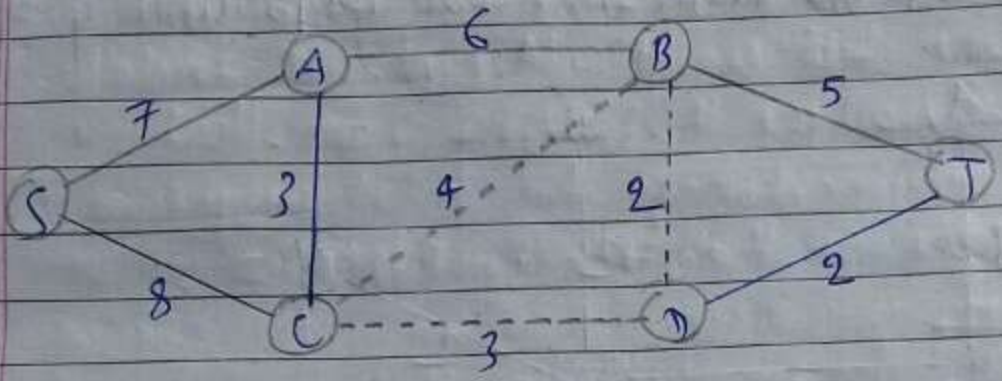


The least cost is 2 and edges involved are B,D and D,T. We add them.

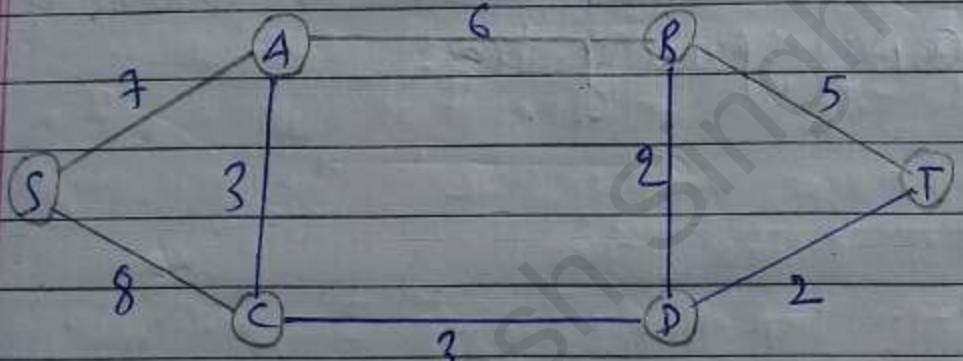
Next cost is 3, and associated edges are A,C and C,D. We add them again-



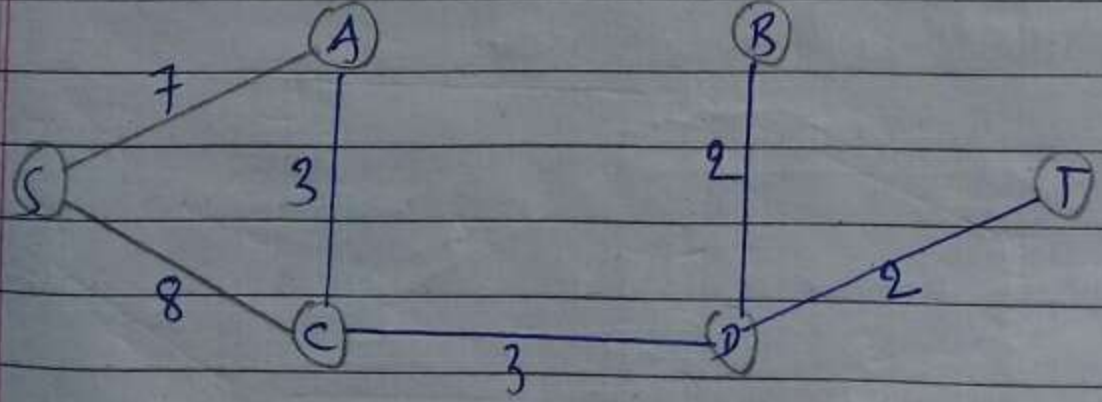
Next cost is 4, and we observe that adding will create a circuit in the graph.



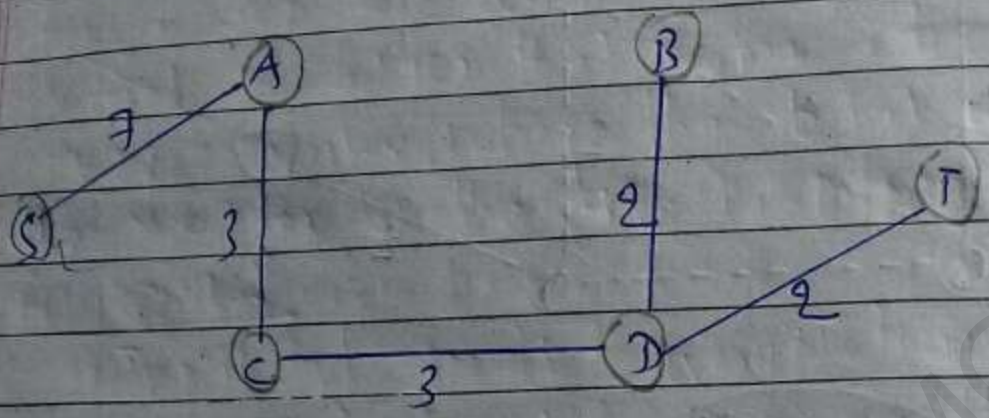
We ignore it. In this process, we shall ignore/avoid all edges that create a circuit.



We observe that edges ~~Cost~~ with cost 5 and 6 also create circuits. We ignore them and move on.



Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



By adding edges S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

b) Prim's Algorithm:

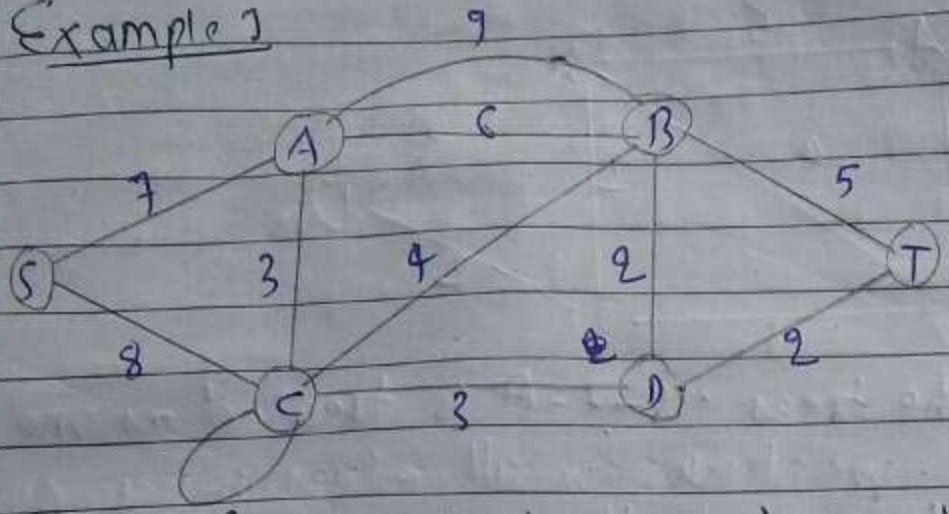
Prim's Algorithm is greedy algorithm in graph that finds a minimum spanning tree for a connected weighted undirected graph. It finds a subset of edges that forms a tree that includes every vertex, where the total weight of all the edges on the tree is minimized.

Algorithm

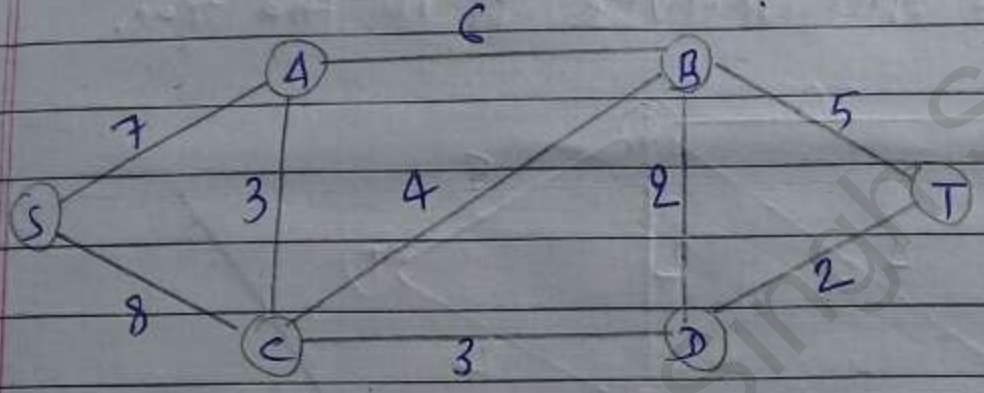
Input - A connected weighted graph.
Output - A connected spanning tree

Step 1 - Initialize the minimum spanning tree with a vertex chosen at random.
Step 2 - Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree.
Step 3 - Keep repeating step 2 until we get a minimum spanning tree.

Example 1

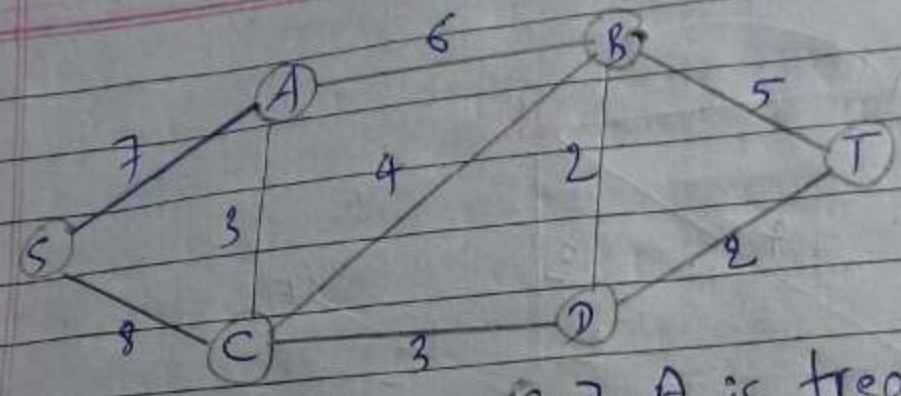


Step 1- Remove all loops and parallel edges.

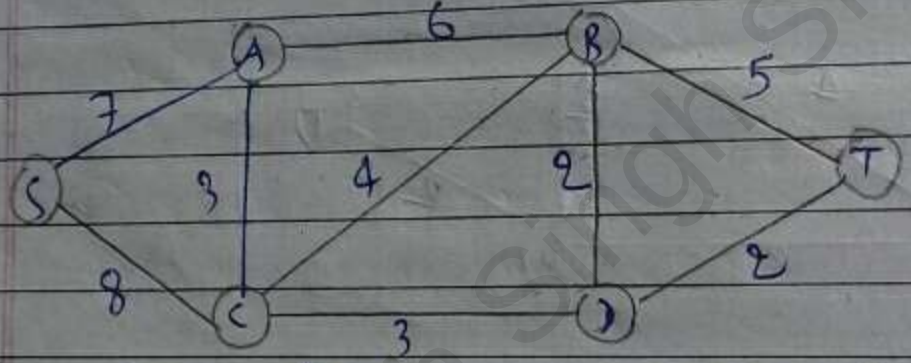


Step 2- Choose any arbitrary node as root node.
In this case choose S node as root node of Prim's spanning tree.

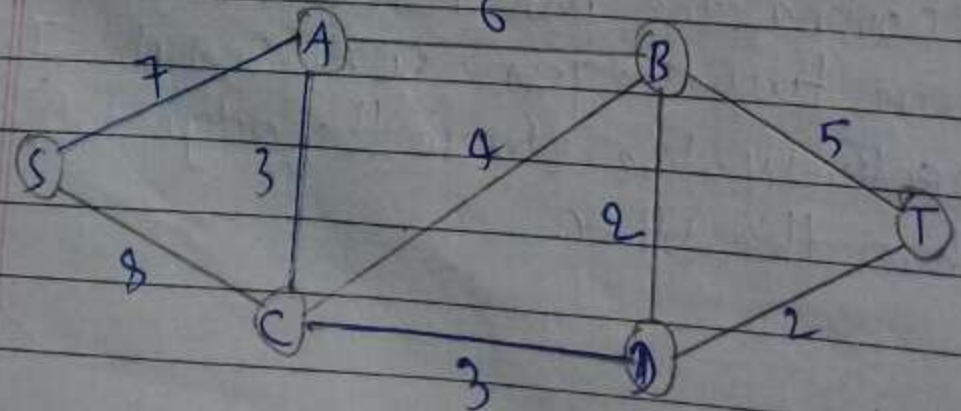
Step 3- Check out edges and select one with less cost.
After choosing the root node S, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



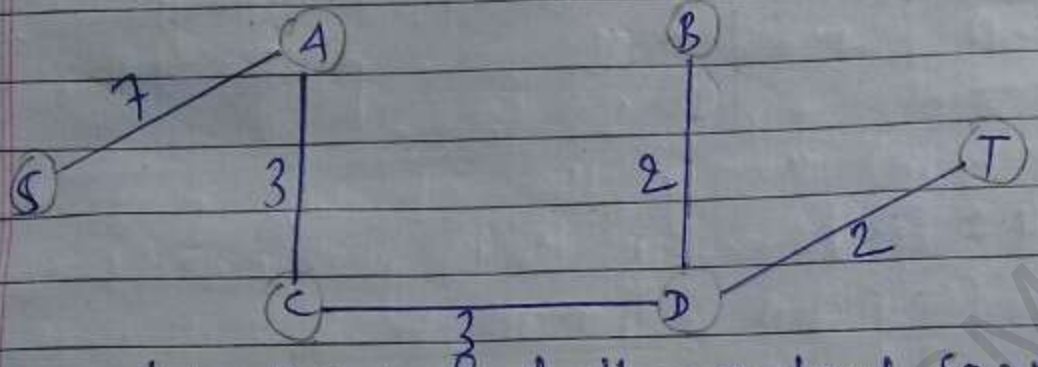
Now, the trees S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3 tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4 etc.



After adding node D to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one.



We may find the output spanning tree of the same graph using two different algorithms & same.

9.4. Shortest Path Algorithms: Dijkstra Algorithm

A path from source vertex s to t is shortest path from s to t if there is no path from s to t with lower weights. Shortest may be least number of edges, least total weight etc.

Shortest simple path is a path that does not visit any vertex twice.

In a graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.

Dijkstra Algorithm

Dijkstra's Algorithm can be used to determine the shortest path from one node in graph to every other node within the same graph data structure, provided that the nodes are reachable from the starting node.

~~Algorithm~~ It was conceived by computer science ~~engin~~ scientist Edsger W. Dijkstra in 1956 and published three years later.

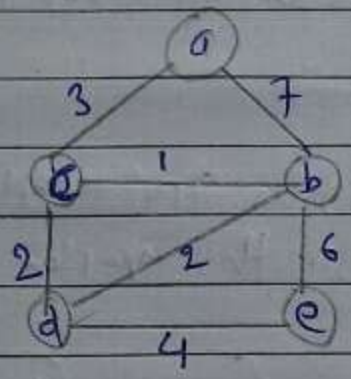
Algorithm

1. From the starting node, visit the vertex with the smallest known distance/cost.
2. Once we have moved to the smallest cost vertex, check each of its neighbouring nodes.

3. Calculate the distance/cost for the neighbouring nodes by summing the cost of the edges leading from the ~~start~~ start vertex.

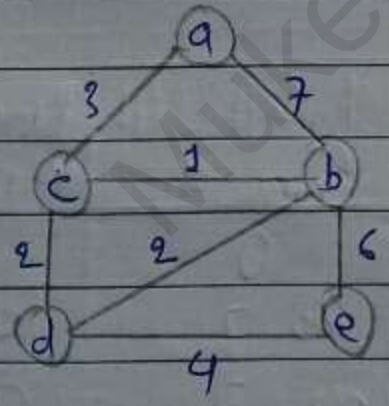
4. ~~If the dis~~ finally, if the distance/cost to a node is less than a known distance, update the shortest distance for that vertex.

Example



There are many possible paths that will allow us to reach node ⓔ from node ⓐ

Step 1

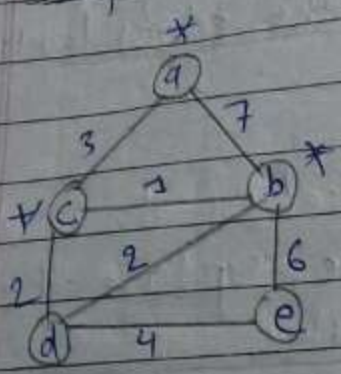


Vertex	Shortest distance from ⓐ	Previous Vertex
a	0	
b	∞	
c	∞	
d	∞	
e	∞	

Since, we are already at ⓐ, our start vertex, the shortest distance is 0.

c

Step 2



Vertex	Shortest dist from a	Previous Vertex
a	0	
b	∞	
c	∞	
d	∞	
e	∞	

Visited = [ ]

Unvisited = [a, b, c, d, e]

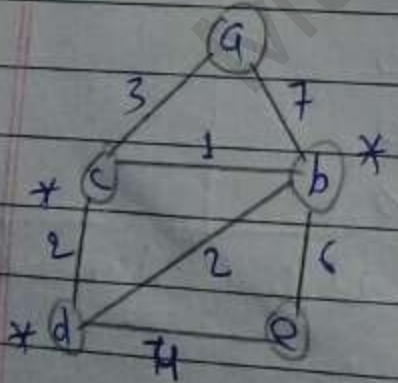
↑ Current Vertex

* Visit the vertex with the smallest known dist

* Examine its neighbouring nodes, and calculate the distance to them from the vertex we are visiting.

- distance to (b) : 0+7=7 | for node (b) : 7 < ∞
- distance to (c) : 0+3=3 | for node (c) : 3 < ∞

Step 3



Vertex	Shortest dist from a	Previous Vertex
a ✓	0	
b	~~∞~~ 7	a
c	~~∞~~ 3	a
d	∞	
e	∞	

Visited = [a]

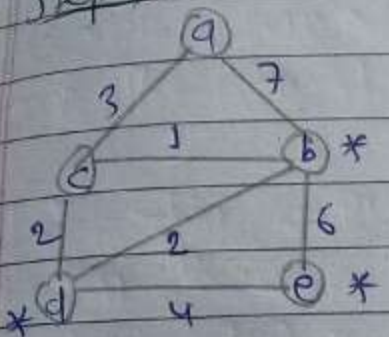
Unvisited = [b, c, d, e]

↑ current vertex

distance to (b) : 3+1=4

distance to (d) : 3+2=5

Step 4



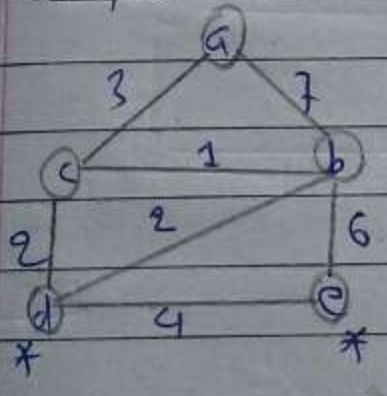
	Vertex	Shortest dist from a	Previous Vertex
✓	a	0	
	b	~~∞~~ 7 4	~~d~~ c
✓	c	∞ 3	a
	d	∞ 5	c
	e	∞	

Visited = [a, c]

Unvisited = [(b), d, e] distance to (e): 4 + 6 = 10

↑ current vertex

Step 5



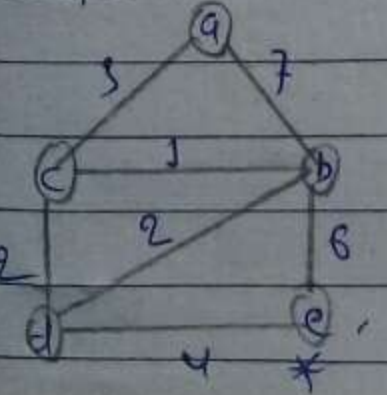
	Vertex	Shortest dist from a	Previous Vertex
✓	a	0	
✓	b	~~∞~~ 7 4	~~d~~ c
✓	c	∞ 3	a
	d	∞ 5	c
	e	∞ 10	b

Visited = [a, c, b]

Unvisited = [(d), e]

↑ current vertex

Step 6



	Vertex	Shortest dist from a	Previous vertex
✓	a	0	
✓	b	~~∞~~ 7 4	~~d~~ c
✓	c	∞ 3	a
✓	d	∞ 5	b
	e	∞ ~~10~~ 9	~~b~~ d

Visited = [a, c, b, d]

Unvisited = [(e)]

↑ current vertex

Finally we end up with just one left.

Vertex	Shortest dist from a	Previous Vertex
a	0	-
b	4	c
c	3	a
d	5	b
e	9	d

The final value from Dijkstra's algorithm

Example 2