Introduction :-

Data are assemble when used for ~~several~~ set up values of a data item refers to a singal unit of values. Data items that are divided into sub items are called ~~book~~ group item. and those are not divided into sub items are called elementry item.

Data structure ⇒

Data may be organised in many different way the logical ~~and~~ or mathematical mode of a perticular organization of data is called Data Structure. The choice of a perticular data model depends on two considerations.

(i) It must be rich enough in structure to mirror the actual relationships of the data in the world.

(ii) The structure should be simple that one can effectively process the data when ~~nessary~~ necessary.

Categorisation of Data Structure :-

(i) Linear
- Array
- Linked List
- Queue
- Stacks

(ii) Non-linear
- Trees
- Graph
- Sets

Date / /
Page

100 on 100

Type of DS:-

(i) primitive DS :- It is primitive data types.
The int, char, float, double---

(ii) Non primitive DS:-

(i) Linear DS (sequential manner)
↳ Arrays, Linked List, Stacks, Queues

(ii) Non linear DS:- When one element is connected to the 'n' number of elements known as a non-linear DS.
Tree, Graph, sets.

Data structure operations ⇒ The data appearing in Data structure are processed by mean of certain operations.

(i) Insertion :- Adding a new record to the structure.

(ii) Deletion :- Removing a record from the structure.

(iii) Traversing :- Accessing each record exactly once so that certain items in the record may be processed.

(iv) Searching :- Finding the location of a record with a given key value.

With following 2 ~~four~~ operations are used in special situations, these are

(i) Sorting :- Arranging the ~~elements~~ record in some logical order.

(ii) Merging :- Combining the record in two different sorted file into a singal sorted file.

Algorithom complexity ⇒

When ever we have a problem statement and we have to identify the solution for it then every solution / algorithom has its time and space complexity that denotes time and space are consumed by that perticular algorithom to solve the problem statement.

Searching Algorithom :—

There are two ways by which we can perform such operations these are :—

(i) Linear Search
(ii) Binary Search

(i) Linear search :— For linear search, it does not mandantary that array elements should be in sorted forms.

(ii) Binary Search :— for binary search, it is mandantary that array elements should be in sorted form.

Mathematical notation :—

Floor and Ceiling :-

Let x be any real number then x lies b/w two integers called the floor and ceiling of x.

→ $\lfloor x \rfloor$ called the floor of x, denotes the greatest integer that does not exceed x.

→ $\lceil x \rceil$ called the ceiling of x, denotes the least integer that is not less than x.

If x is itself an integer, then

$$\lfloor x \rfloor = \lceil x \rceil ;$$

otherwise

$$\lfloor x \rfloor + 1 = \lceil x \rceil ,$$

eg- $\lceil 3.14 \rceil = 4$

$\lfloor 3.14 \rfloor = 3$

$\lfloor \sqrt{5} \rfloor = 2$

$\lceil \sqrt{5} \rceil = 3$

$\lfloor \sqrt{3} \rfloor = 1$

$\lceil \sqrt{3} \rceil = 2$

$\lfloor -8.5 \rfloor = -9$

$\lceil -8.5 \rceil = -8$

$\lfloor 7 \rfloor = 7$

$\lceil 7 \rceil = 7$

Modular Arithmetic ⇒

Let k be any integer and let M be a positive integer than

$k \pmod M$

will denote the integer reminder when k is divided by M. More exactly it can be represented as

$$k = Mq + r \quad \text{where} \quad 0 \le r < M$$

eg- $25 \pmod 7 = 4$

Integer and Absolute value functions :—

→ Let x be any real number. The integer value of x, written INT(x), converts x into a integer by deleting (truncating) the fractional part of the number.

eg- INT(3.14) = 3 , INT(7) = 7
INT($\sqrt{5}$) = 2
INT(−8.5) = −8

Observe that INT(x) = $\lfloor x \rfloor$ or INT(x) = $\lceil x \rceil$ according to whether x is positive or negative.

→ The absolute value of the real number x, written ABS(x) or |x| is defined as the greater of x or −x. Hence

ABS(0) = 0 and for ABS(x) = x or ABS(-x) = -x depending on wheather x is positive or negative.

eg- |-15| = 15 , |-3.33| = 3.33
|7| = 7

~~Exactio~~ Factorial function ⇒ The product of positive integer from 1 to n, inclusive, is denoted by n!

~~ALGORITHM~~ ASYMPTOTIC NOTATIONS FOR COMPLEXITY OF ALGORITHM (Ω, θ, O)
⇒

Big O Notation defines an upper bound function g(n) for f(n) which represents the time/space complexity of the algorithm ~~of~~ on an input characteristic.
There are other asymptotic notation such as Ω, θ, o which also serve to provide bounds for the function f(n).

$f(n) \leq (\geq g(n))$

Omega Notation (Ω) ⇒ The Ω notation is used when the function g(n) defines a lower bound for the function f(n)

$f(n) \geq (\leq g(n))$

Big O → $f(n)$ is run time algorithm and $g(n)$ is an arbitrary time complexity that we are trying to relate with our algorithum. $f(n) = O(g(n))$ for some real cons. c where $c>0$ and

Definition:-

$f(n)$ = omega of $g(n) = \Omega(g(n))$, if there exists a positive integer n_0 and a positive no. M such that

$|f(n)| \geq M|g(n)|$, for all $n \geq n_0$

eg- For $f(n) = 18n + 9$

$f(n) > 18n$ for all n

hence $f(n) = \Omega(n)$

eg- $f(n) = 90n^2 + 18n + 6$

$f(n) > 90n^2$ for $n \geq 0$

therefore $f(n) = \Omega(n^2)$

Theta notation (θ) ⇒

The theta notation is used ~~used~~ when the function is bounded in both from above and below by the function $g(n)$.

$f(n) = \theta(g(n))$, if there exists two positive constants c_1 and c_2 and a positive integer n_0 such that

$c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$ for all $n \geq n_0$.

eg- $f(n) = 18n + 9$

since $f(n) > 18n$ and $f(n) \leq 27n$ for $n \geq 1$

$f(n) = \Omega(n)$ and $f(n) = O(n)$ respectively for $n \geq 1$

Big O :— $f(n) \rightarrow$ run time algorithm
$g(n) \rightarrow$ arbitrary time complexity
$f(n) = O(g(n))$
for some real con. C a where $C > 0$ and no.

$$f(n) \leq c(g(n))$$

for some input size n where $n > n_0$

Q. $f(n)$ $O(g(n)) = ?$
where $f(n) = 3\log n + 100$
$g(n) = \log n$

Ans-

$$f(n) \leq c(g(n))$$
$$3\log_e n + 100 \leq c\log_e n$$

According to question
$n \geq 2$

On putting $c = 150$, $n = 2$

$$3\log_e 2 + 100 \leq 150\log_e 2$$
$$102.4 \leq 104.7$$

$\boxed{C = 150,\ n \geq 2}$

8

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \cdots < 2^n < 3^n < \cdots < n^n$

lower | avg | upper

9

Q. $f(n) = 3 * n^2$, $g(n) = n$, is $f(n) O(g(n)) = ?$

Ans -

$$f(n) \leq c(g(n))$$
$$3 * n^2 \leq cn$$
$$n \geq 1$$

let $n = 1$, $c \geq 3$
let $n = 2$, $c \geq 6$
let $n = 3$, $c \geq 9$

∴ n is possible but c is not possible. because c is varing with n.

Analysing Algo Control Structures ⇒

(i) The analysing of an algorithm is calculated by considering its individual instructions. The individual instructions are calculated and then according to the control structures we combined these times. some contrl structures algo. are as:-

(i) Sequencing
(ii) if then else
(iii) for - loop
(iv) while - loop
(v) Recursion

(i) Sequencing ⇒ Suppose our algorithm consist of two part A & B. A takes time t_A and B takes time t_B.

The total computation $t_A + t_B$ is according to a sequencing rule, according to the max. rule this max. computation time will be

$$\max(t_A, t_B)$$

(ii) if - then - else :-

The total computation time is according to a conditional rule if then else. According to a max. rule this computation time is

$$\max(t_A, t_B)$$

Q. 1 For the following program big O analysis of the running time.

(iii) For-loop ⇒ (i)
```
for( i=0 ; i<n ; i++ )      ⇒ n+1
    A[i] = + ;              ⇒ n
```

Ans-

$$f(n) = n+1+n$$
$$f(n) = 2n+1$$

$$f(n) = O(n)$$
$$\text{complexity} = n$$

(ii)
```
for( i=0 ; i<n ; i++ )          ⇒ (n+1)
    for( j=i ; j<n ; j++ )      ⇒ n*(n+1)
        for( k=j ; k<n ; k++ )  ⇒ n*n*(n+1)
```

Ans-

$$f(n) = (n+1) + n^2+n + n^3+n^2$$
$$f(n) = n^3 + 2n^2 + 2n + 1$$

(iv) While loop :⇒ In this there is no obvious method which determines we shall have to repeat the loop the simple technique for analysis the loop to firstly determine the function of variable involve whose value decreases each time around. Secondly for terminating the loop it is necessary that this value must be +ve integer by keeping the track of how many times the value of function decreases one can obtain a no. of repeatation for the loop. The another approach for analysis while loop to treat them as a recursive algorithum.

Q. The running time of alg. array max for computing the max. element in an array of n integer is big O(n).

Ans -

```
array max(A, n)
1. current max ← A[0]
2. for (i=1 ; i≤n ; i++)
3. do if current max < A[i].
4. then current max ← A[i]
5. return current max
```

$f(n) = 2n+1$

complexity $= n$

Array

Algorithum for traversing :—

(i) Algo (Traversing of an array)
Traverse (A, N, LB, UB)
Here A is array variable , N = no. of element
LB is low bound of array
UB is upper bound of array
(i) Repeat for k = LB to UB
(ii) process A[K]
[End of step 1 loop]
(iii) Exit

or

(i) K = LB
(ii) while (K <= UB)
(iii) Process A[K]
[End of step 2 loop]
(iv) Exit

(ii) Algo (deletion of an array)
Deletion (A, N, k, ITEM)
Here A is linear array with ~~variable~~ N elements
N = no. of element
k is a positive integer such that k<N
This algorithum delets the kth element from A

1. Set ITEM = A[K]
2. Repeat for J = k to N-1
[Shifting the element left side]
A[J] = A[J+1]

[end of step 2 loop]

2. [Reset the no. N in array A]

3. Set N = N-1

4. Exit

(iii) Algo (insertion of an array)

INSERT (A, N, K, ITEM)

1. [initialize] set J = N
2. Repeat 3 & 4 while J >= K
3. [move J^th element downward] Set A[J+1] = A[J]
4. [Decrease counter] Set J = J-1
   [End the step 2 loop]
5. [insert element] Set A[K] = ITEM
6. [Reset N] Set N = N+1
7. Exit.

(iv) (Linear search) LINEAR (DATA, N, ITEM, LOC)

1. [insert item at the end of DATA] Set DATA[N+1] = ITEM
2. [initialize] Set LOC = 1
3. [search for ITEM]
   Repeat while DATA[LOC] ≠ ITEM
   Set LOC = LOC + 1
   [End the loop]
4. [Successful?] if LOC = N+1, then Set LOC = 0
5. Exit

Binary Search

Algo (Binary Search)

BINARY (DATA, LB, UB, ITEM, LOC)

1. Set BEG = LB, END = UB
   MID = INT ((BEG + END)/2)
2. Repeat 3 & 4 while (BEG ≤ END) and DATA[MID] ≠ ITEM
3. If ITEM < DATA[MID] then
   set END = MID - 1
   Else BEG = MID + 1
   [end of if structure]
4. Set MID = INT(BEG + END)/2
   [end of step 2 loop]
5. if DATA[MID] = ITEM then set LOC = MID
   else set LOC = NULL
   [end of if structure]
6. Exit

Note- when ever ITEM does not appear in data then eventually arrive at the stage BEG = END = MID Then next step yields END < BEG, and control transfers to step 5 of the algorithm.

eg- We have a following sorted array DATA

DATA = 11, 22, 30, 33, 40, 44, 50, 60, 66, 77, 80, 88, 99

(i) then give an array to binary search to find 40 in given array

Ans -

1. initially BEG = 1, END = 13, MID = 7
2. Since 40 < 50, and has changed its value END = 7 - 1 = 6

Hence MID = INT((1 + 6)/2)

MID = 3

So DATA[MID] = 30

3. Since 40 > 30

BEG has changed its value BEG = 3+1 = 4

hence MID = INT((4+6)/2) = 5

So DATA[MID] = 40

We have find an element 40 at location MID = 5 so search is successful.

(ii) find 88 in given array.

1. initially BEG = 1, END = 13, MID = 7
2. Since 88 > 50 and has changed its value BEG = 7+1 = 8

MID = INT((8+13)/2)

MID = 10

So DATA[MID] = 77

Since 88 > 77

and BEG has changed its value:

BEG = 10 + 1 = 11

MID = INT((11+13)/2)

MID = 12

So DATA[MID] = 88

we have found an element 88 at location MID = 12 so search is successful.

Unit - I
Stack

Stack :- Stack is a linear Data structure. It works on LIFO principal. LIFO means the element inserted at last will be removed first.

→ The end from which we perform insertion & deletion is known as Top of the stack.

→ Means to perform insertion and deletion.

→ We only have a single end.

→ For insertion and deletion we use terminology PUSH and POP respectively.

→ Whenever we have to perform insertion & deletion in a stack we always check whether the stack is empty or not. If the stack is full and we want to perform insertion in stack the situation is known as OVER FLOW.

→ And whenever we want to perform deletion in the stack and stack is already empty, this situation is known as UNDER FLOW. Means to perform insertion we will check overflow and to perform deletion we will check underflow.

ARRAY representation of STACK ⇒

Stack may be represented in a CS in various ways usually by means of

a one way list or linear array.
Unless until its stated or implied, each & every stack will be maintained by STACK, a pointer variable TOP which contains the location of the TOP elements of the STACK and a variable MAXSTK which gives the max. no. of element that can be held by a STACK.

→ The condition TOP = 0 or TOP = NULL will indicate that stack is empty.

→ The operation of adding an ITEM onto the stack and operation of removing an ITEM from stack implimented by the procedure PUSH and POP respectively.

→ At the time of, insertion first we will check OVERFLOW and at the time of deletion, first we will check UNDERFLOW.

Procedure ⇒

Push (STACK, MAXSTK, TOP, ITEM)

In this procedure, pushes an item on to the stack

1. [Stack already filled?] If TOP = MAXSTK then print OVER FLOW and Return
2. Set TOP = TOP + 1 [increment TOP by 1]
3. Set STACK [TOP] = ITEM [insert ITEM to the TOP position of STACK]

4. Return

Procedure :—

POP(STACK, [redacted], TOP, ITEM)

This procedure deletes the top element of stack and assigns it to variable ITEM

1. [Stack already empty?] If TOP = NULL then print UNDERFLOW and return
2. Set ITEM = STACK[TOP] [initialize the element of TOP to item]
3. Set TOP = TOP - 1 [Decrement the TOP by 1]
4. Return

Application of STACK ⇒

1.) For conversion of mathematical expression.
2.) To solve any recursive problem.
3.) For memory management.
4.) Reversing a list.

Minimising ~~Over flow~~

Minimising overflow OR multi stack implementation using an array ⇒

Whenever we talk about STACK we can perform insertion & deletion on it. Whenever we will talk about insertion (PUSH) the programmer has to keep in mind how efficiently space can be used.

Overflow of STACK depends upon the memory space reserved for the stack. The particular choice of the amount of the memory for ~~the~~ a given STACK involves Time - space tread off Reserving a great deal of space minimises the overflow. However this may be expensive use of space if most of the space is used ~~a~~ rarely.
On the other hand, reserving a small amount of space for each stack may increase the no. of time OVERFLOW occurs.
To solves this problem and approach multi stack implementation using an array was introduced. Let suppose we have an stack A & B with size n_1 & n_2 respectively.
Now be defined a single stack C ~~of~~ & d of size is $n_1 + n_2$. We ~~said~~ defined stack C[i] as the bottom element of stack A and it ~~grows~~ goes towards right upon n. And we defined stack C[n] as the bottom element of stack B and it goes towards Left to stack A.
In this case, OVERFLOW will occur only ~~the~~ when A & B together have more than $n_1 + n_2$.
This technique will repeat the no. of overflow withaut increasing a memory space.

Linked representation of STACK ⟹

The linked representation of stack commonly termed linked stack that is implemented using a singly Linked List. The info field of the nodes ~~for~~hold the element of stack and the linked field hold the pointers to the ~~reverse~~ neighbouring element in the stack.
The start pointer of the linked list behaves as the top pointer variable of the stack and the NULL pointer of the last nodes in the list singles the bottom of the stack.

Procedure:—

LINK STACK (INFO, LINK, ITEM, AVAIL, TOP)

This procedure push an element into a linked stack.

1. [available space?] If AVAIL = NULL then print OVERFLOW and exit
2. [Remove first node from avail list] NEW = AVAIL
   AVAIL = LINK[AVAIL]
3. Set INFO[NEW] = ITEM [Copies item into new node
4. LINK[NEW] = TOP [new node points to the original top node]
5. TOP = NEW
   (Reset top to point to the new node
6. Exit.

Procedure :-

LINK STACK(INFO, LINK, TOP, AVAIL, ITEM)

This procedure deletes the top element of a linked stack and assigns it to the ~~element~~ of available item.

1. (Stack already empty ?) If TOP = NULL then print UNDERFLOW and return
2. Set ITEM = INFO[TOP] [Copies the top of element in ITEM]
3. Set TEMP = TOP and TOP = LINK[TOP] [remember the old value of TOP pointer in TEMP and reset TOP to point to the next element in the stack]
4. [Return deleted node to the AVAIL list] Set LINK[TEMP] = AVAIL
   AVAIL = TEMP
5. Exit

Application of STACK ⇒

(i) Reversing a List ⇒

STACK Reverse (STACK, ~~MAXSTK~~ N, TOP)

This procedure reverse a stack.

1. Repeat from top=N to top=1 (having N element)
   process STACK[TOP]
2. Exit

For reversing the list of stack first we assumed that we have stack of size N in memory and we have to process this stack

Top of stack
from ~~STACK[TOP]~~ to while TOP = 1.

2.] Stack is also used to solve recursive problems like factorial, tower of Hanoi.

Recursion ⇒

Recursion is an important concept in computer science. Many algorithm can be based described in term of recursion. A recursive procedure must have following 2 properties:-

(i) There must be certain criteria ~~area~~ called base criteria, for which the procedure does not call itself

(ii) Each time the ~~prod~~ procedure does call itself (directly & indirectly), it must be closer to the base criteria.

Procedure to evaluate the factorial of a no. ⇒

(a) loop procedure →

FACTORIAL (FACT, N)

This procedure calculate factorial N and returns the value in the FACT.

1. If N=0 then set FACT = 1 and Return.
2. Set FACT = 1 [initialises the FACT for loop]
3. Repeat for k=1 to N
   Set FACT = FACT * K
   [end of loop]
4. Return

(b) Recursion procedure →

FACTORIAL (FACT, N)

This procedure can calculates factorial N and returns the value in variable FACT.

1. If N = 0 then set FACT = 1 and return
2. Call FACTORIAL (FACT, N-1)
3. Set FACT = FACT * factorial(fact, n-1)
4. Return

Tower of Hanoi ⇒

TOH is a puzzle game. In this we have 3 towers a, b, c and we have to shift all the disk contained in tower a to tower c. All disk are arranged in tower c in Descending order of their size from bottom to top. When we start shifting of disk from tower a to tower c, we have to follow some primary conditions :—

(i) At a time only a single disk can be moved from tower a and from any other tower.

(ii) Largest disk cannot be come over a smallest disk. And a no. of moves for moving disk from tower a to tower c is equal to $2^N - 1$ where N = No. of disk.

Procedure :-

TOWER (N, BEG, AUX, END)

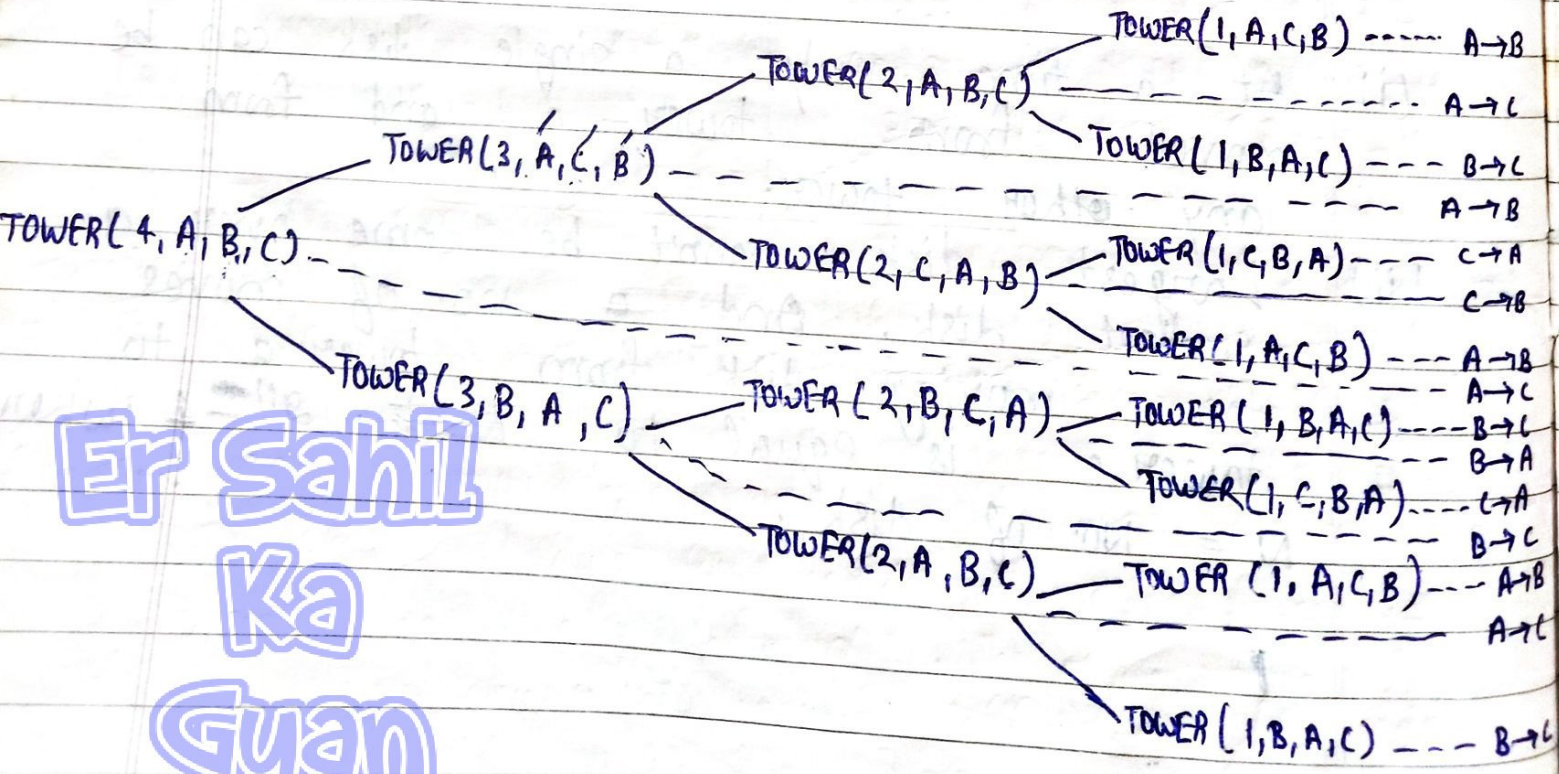
This procedure gives a recursive solution to the TOH problems for N no. of disk.

1. If N=1 then
   (a) write BEG → END
   (b) return
2. [move N-1 Disk from tower BEG to tower AUX]
   call TOWER (N-1, BEG, END, AUX)
3. write BEG → END
4. [move N-1 Disk from tower AUX to END]
   call TOWER(N-1, AUX, BEG, END)
5. Return.

Q. Solve the TOH problem for N=4.

Ans -

TOWER(4, BEG, AUX, END)



3./ For solving any mathematical expression: ⇒

An IF expression can be represented in 3 ways :-

(i) Infix :- (Polish notation)
Whenever operators comes b/w operands.

(ii) Prefix :-
Whenever operator comes before operands.

(iii) Postfix :- (Reverse Polish notation)
Whenever operator falls after operands.
A+B+C , +AB , AB+

Evaluation of a postfix expression ⇒

Suppose P is an arithmetic expression return in postfix notation the algorithm to evaluate with the help of stack will be as —

Algo
This algorithm find the value of an arthmetic expression P return in postfix notation.

1. Add right parenthesis ")" at the end of P.
2. Scan P from left to right & repeat step 3 & 4 for each element of P until the right (")" is encontered.
3. If an operand is encounteded, put it on STACK
4. If an operator ⊗ is encountered, then:
   a) Remove the two top elements of STACK, where A is the top element & B is the next-to-top element.
   (b) Evaluate B ⊗ A.
   (c) Place the result of (b) back on STACK.

[End of if structure]
[End of step 2 loop]

5. Set VALUE equal to the top element of the stack.
6. Exit.

Q. Consider the following expression in postfix notation.
P: 5, 6, 2, +, *, 12, 4, /, -

Soln

1. Add Right parenthesis ")" at the end of expression.
P: 5, 6, 2, +, *, 12, 4, /, -,)

Symbol scanned	STACK
(i) 5	5
(ii) 6	5, 6
(iii) 2	5, 6, 2
(iv) +	5, 8
(v) *	40
(vi) 12	40, 12
(vii) 4	40, 12, 4
(viii) /	40, 3
(ix) -	37
(x))	

IMP

Infix Expressions into Postfix Expression ⟹

Algorithm: POLISH (Q, P)

Suppose Q is an arithmetic expression ~~return in~~ written in infix notation. This algorithm finds the ~~evin~~ equivalent postfix expression P.

1. Push "(" onto STACK and add ")" at the end of Q.
2. Scan Q from left to right & repeat step 3 to 6 for each element of Q until the stack is empty.
3. If an operand is encountered add it to P.
4. If a left parenthesis "(" is encountered push it onto stack.
5. If an operator ⊗ is encountered, then
   (a) Repeatedly pop from STACK and add to P each operator which has same precedence as or higher precedence than ⊗
   (b) Add ⊗ to STACK.
   [End of if structure]
6. If a right parenthesis is encountered, then:
   (a) Repeatedly pop from STACK and add to P each operator until a left parenthesis is encountered.
   (b) Remove the left parenthesis.
   [Do not add the left parenthesis to P]
   [End of if structure]
   [End of step 2 loop]
7. Exit

Q. Consider the following arithmetic infix expression Q: A + (B*C - (D/E↑F) * G) * H

Soln:-

(ABC * DEF ↑ / G * - H * +)

Symbol Scanned		STACK	Expression P
(i)	A	(	A
(ii)	+	(+	A
(iii)	(	(+ (	A
(iv)	B	(+ (	AB
(v)	*	(+ (*	AB
(vi)	C	(+ (*	ABC
(vii)	-	(+ (-	ABC*
(viii)	(	(+ (- (	ABC*
(ix)	D	(+ (- (	ABC*D
(x)	/	(+ (- (/	ABC*D
(xi)	E	(+ (- (/	ABC*DE
(xii)	↑	(+ (- (/ ↑	ABC*DE
(xiii)	F	(+ (- (/ ↑	ABC*DEF
(xiv)	)	(+ (-	ABC*DEF↑/
(xv)	*	(+ (- *	ABC*DEF↑/
(xvi)	G	(+ (- *	ABC*DEF↑/G
(xvii)	)	(+	ABC*DEF↑/G*-
(xviii)	*	(+ *	ABC*DEF↑/G*-
(xix)	H	(+ *	ABC*DEF↑/G*-H
(xx)	)		ABC*DEF↑/G*-H*+

Transformation infix to prefix expressions ⇒

1. PUSH ")" onto stack and add "(" at the end of the expression A.
2. Scan A from right to left and repeat step 3 to 6 for each element of A until the stack is empty.
3. If an operand is encountered, add it to B.

4. If an ")" is encountered push it onto STACK.

5. If an operator ⊗ is encountered, then:
   (a) Repeatedly pop from STACK and add to B each operator which has higher precedence than the operator ⊗.
   (b) Add operator ⊗ to the stack.

6. If ")" is encountered then (a) repeatedly pop from the stack and add to B each operator.
   (b) Remove the left parenthesis "("

7. Exit.

Q. Q: A + (B * C - (D / E ↑ F) * G) * H

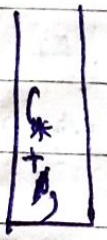
Ans -

Symbol Scanned	STACK	Expression P
H	)	H
*	)*	H
)	)*)	H
G	)*)	H G
*	)*)*	* G H
)	)*)*)	* G H
F	)*)*)	* F G H
↑	)*)*)↑	F G H
E	)*)*)↑	E F G H
/	)*)*)/	↑ E F G H
D	)*)*)/	D ↑ E F G H
(	)*)*	/ D ↑ E F G H
−	)*)-	* / D ↑ E F G H
C	)*)-	C * / D ↑ E F G H
*	)*)-*	C * / D ↑ E F G H
B	)*)-*	B C * / D ↑ E F G H
(	)*	− * B C * / D ↑ E F G H

+	)+	*-*BC/D↑FFGH
A	)+	A*-*BC/D↑EFGH
C		+A*-*BC/D↑EFGH

Q. Q:- A * B + C / D prefix

Ans-

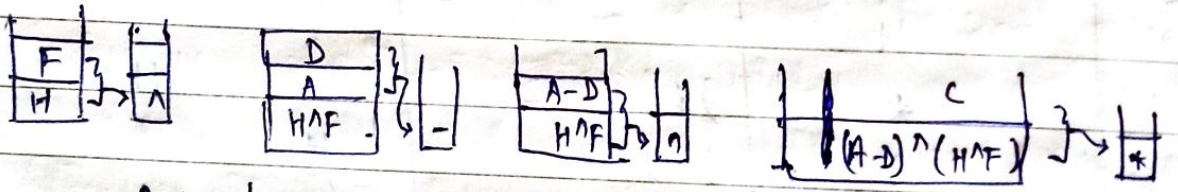


+ * AB / CD

Prefix to Infix :-

Q. + A | B * C ^ - D A ^ F H

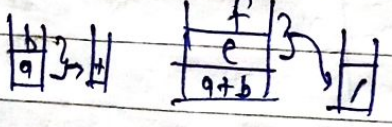
Reverse :- H F ^ A D - ^ C * B | A +



A + B | C * (A-D) ^ (H ^ F) Ans

Postfix to Infix :-

Q. a b + e f / *



⇒ (f/e) * (a+b) Ans

Unit-2

Queue & Linked List

Queue ⇒ Quene is a linear data structure. It works on the principle of FIFO means the elements inserted first will be removed first from the Quene. Queue will be maintained by linear array unless until it stated & implied. We will consider queue implementation to linear array.

Queue has two different ends FRONT & REAR. The condition FRONT = NULL indicate that the queue is empty.

→ Whenever an element is deleted from the queue, FRONT will be incremented by 1 and it can be shown with the help of assignment

FRONT = FRONT + 1

→ Similarly Whenever an element is added to the queue the value of REAR will be incremented by 1 and it can be shown by assignment

REAR = REAR + 1

This means after N insertion the rear element of the queue will occupy

QUEUE[N]

or in other word eventually the queue is occupied the last part of array. This occurs even the queue itself may not contain many elements.

let suppose we want to insert element in queue and that time REAR = N. One way to do this simply move the entire queue to the begining of the array changing FRONT & REAR accordingaly and the inserting item in Queue This procedure is expensive. Another way to solve this we consider our queue. ~~Another way~~, as the ~~circle~~ circular queue, means queue of 1 comes after queue[N] in array. With this assumption we insert item into queue by assigning item to queue[1].

Instead of increasing REAR to N+1, we reset REAR = 1 and then assign Queue [REAR] = ITEM

→ Similar, if FRONT = N and element is deleted from the queue we reset FRONT = 1 instead of increasing FRONT = N+1.

→ If Queue contains only a one element in that case FRONT = REAR ≠ NULL.

~~Supp~~ we have a single element in a queue and that element has been deleted and that case FRONT = NULL & REAR = NULL will be assigned.

enqueue

Procedure: QINSERT (QUEUE, N, FRONT, REAR, ITEM)
This procedure inserts an ITEM into QUEUE

1. [Queue already filled?] If FRONT = 1 and REAR = N OR FRONT = REAR + 1 then

write OVERFLOW and Return

2. [Find new value of REAR]
If FRONT = NULL then Set FRONT = REAR = 1
else if REAR = N then Set REAR = 1
else REAR = REAR + 1
[End of if structure]

3. QUEUE [REAR] = ITEM

4. Return

dequeue

Procedure :-

QDELETE (QUEUE, N, FRONT, REAR, ITEM)

This procedure deletes an element from the Queue and assign it to the variable ITEM.

1. [Queue already empty?] If FRONT = NULL then write UNDERFLOW and Return

2. Set ITEM = QUEUE [FRONT]

3. [Find new value of FRONT]
(when we have one element)
If FRONT = REAR then set FRONT = NULL and REAR = NULL
else if FRONT = N then set FRONT = 1
else FRONT = FRONT + 1
[End of if structure]

4. Return

Linked Representation of Queue :-

A linked queue is a Queue implemented as a linked list with two pointer variable REAR & FRONT pointing to the node which is in FRONT & REAR of the queue the info field of the list ~~co~~hold the

elements of queue and the link field holds pointers to the neighbouring element of the queue. The array representation of queue suffers from the drawback queue capacity. This is turn calls for the checking of OVERFLOW condition everytime and insertion is made into the queue.

Procedure :-

(a) LINKQ_INSERT (INFO, LINK, FRONT, REAR, ITEM, AVAIL)
This procedure inserts an ITEM into a linked queue

1. [Available space?] If AVAIL = NULL then write OVERFLOW and ~~Return~~ Exit.
2. [Remove first node from avail list]
   Set NEW = AVAIL
   AVAIL = LINK[AVAIL]
3. Set INFO[NEW] = ITEM and LINK[NEW] = NULL
4. If FRONT = NULL then Set FRONT = REAR = NEW
   else set LINK[REAR] = NEW and ~~REAR = NEW~~ REAR = NEW (Rear update)
5. Exit

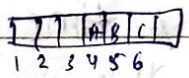
(b) LINKQ_DELETE (INFO, LINK, REAR, FRONT, ITEM, AVAIL)
This procedure deletes the FRONT element of linked queue and stores it in ITEM.

1. [Linked Queue empty?] If FRONT = NULL then write UNDERFLOW and Exit.
2. Set TEMP = FRONT [if linked queue is non-empty remember FRONT in a tempery ~~valu~~ variable TEMP]
3. ITEM = INFO[TEMP]
4. FRONT = LINK[TEMP] [Reset FRONT to point to the next element in the queue]

5. Set LINK[TEMP] = AVAIL and AVAIL = TEMP
[return the deleted node TEMP to the AVAIL List]

6. Exit.

Dequeue ⇒ A dequeue is pronounced either deck / dequeue. In this type of queue, element can be added or removed at either end but not in the middle. The term dequeue is contraction of the name double-ended queue. Dequeue can be either input restricted or output restricted. In input restricted Dequeue input can take place from a single end but output can be performed from both the ends. In output restricted dequeue output can take place from a single end but input can performed from both the ends. Unless Until it is stated or implied, we will assume that dequeue is maintained by a circular array with pointers left & right which points to the two ends of the queue



Left: 4
Right: 6

Priority Queues ⇒ A priority Queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed come from the following rules:

(i) An element of higher priority is processed before any element of lower priority

(2.) Two elements with the same priority are processed according to the order in which they were added in Queue.

Circular ~~Application of~~ Queue ⇒ In this case, we will consider queue as circular means after queue of N queue[1] falls. In this case when we perform insertion after reaching queue[N] REAR comes to 1.
Similarly if we perform deletion when the FRONT reaches N then after deleting the N^{th} element front will come to 1.
The circular queue will utilize the space efficiently without increase the space required.

** Round Robin Algo ⇒ Round Robin algorithm is an application of queue. CPU is assigned to the process on the bases on FCFS (First come first serve) for a fixed amount of time is called Time Quatum / Time Slices. After the time quatum expires the running process is ~~primited~~ preempted and sent to the ready Queue then the processor is assigned to the next arrived process.
It is also preemptive in nature.
Advantage →
(i) It gives the best performance in term of average of rest time.

(iii) It is best suited for time sharing system, client server architechturly

Disadvantage :-

(i) It leads to started processes with larger burst time as they have to repeat cycle many time.

(ii) Its performance heavily depends upon time quantum.

(iii) Priority can't be set for the process.

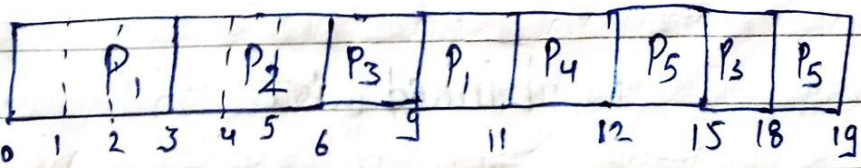
Note- (a) Smaller value of time quantum is better in terms of response time.

(b) Higher value of time quantum is better in term of no. of context switches.

Q. Consider the following solve this by using round robin algo by considering TQ = 3

	AT	BT	RT
P_1	0	5	2
P_2	1	3	
P_3	3	6	3
P_4	5	1	
P_5	6	4	1

Ans- Rough :- P_1 | P_2 | P_3 | P_1 | P_4 | P_5 | P_3 | P_5

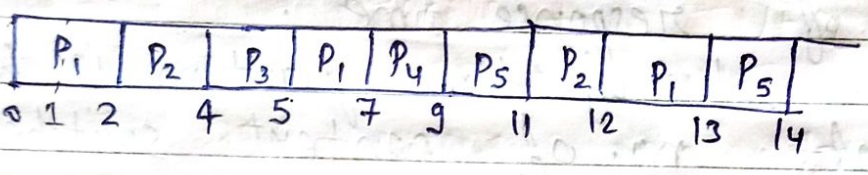


Q.2 Consider the set of 5 processes whose AT & BT are given below:

P.Id	AT	BT	RT
P_1	0	5	3=1
P_2	1	3	1
P_3	2	1	
P_4	3	2	0
P_5	4	3	1

If the CPU scheduling is Round Robin with TQ = 2 Unit then calculate the average waiting time and average turned around time.

Ans -



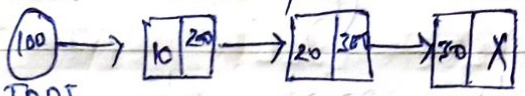
Now we have turned around time
= Exit time - arrival time
And waiting time = turn around time - Burst time

P. id	Exit time	Turnaround time	waiting time
P_1	13	13	8
P_2	12	11	8
P_3	5	3	2
P_4	9	6	4
P_5	14	10	7

Linked List

Linked list is a linear data structure which is collection of nodes. Each node has two parts - INFO, LINK. LINK part will hold the address of next node. First node's address will be

held by the special pointer i.e. known as start. In Last node's link part will have NULL in it. means if we traverse linked list from the begining we will identify the end of list by NULL in last node's link part.

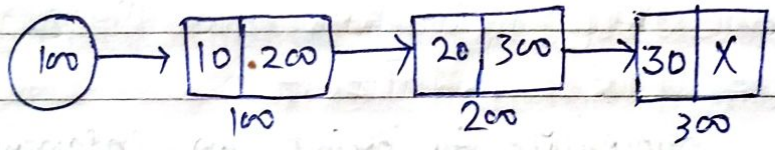
eg- 

Linked list can be sorted or unsorted.

Type of linked list

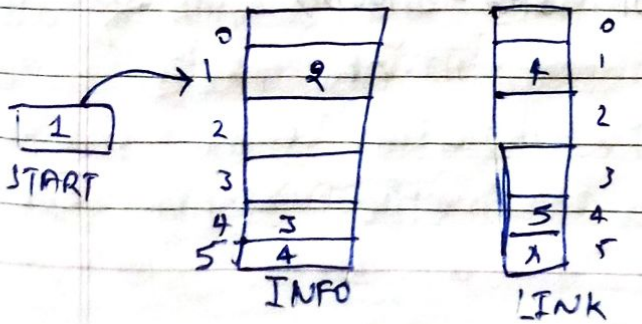
(i) Singly linked list (ii) Doubly linked list
(iii) Hedder linked list (iv) Circular linked list

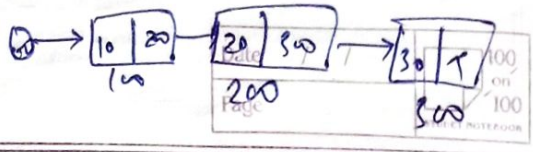
(i) Singly linked list :— A linked list is called singly linked list if we can traverse from begining to end but we cann't traverse link in backward direction.



Memory representation of linked list :—

Let list be a linked list in a memory. Unless until its stated or implied, List will be represented by two linear array. Linear array Info[K] and LINK[K]. List also requires a variable that will contains the begining location of list and it is known as start.





Traverse operation of linked list:—

(A) Traversing ⇒ Let ~~links~~ list be a linked list memory the traversing ~~by~~

The variable ptr points to the node currently being processed:

~~Step~~ 1. Set PTR = START [initialises pointer PTR]
2. Repeat step 3 & 4 while PTR ≠ NULL
3. Apply process to INFO[PTR]
4. set PTR = LINK[PTR]
[PTR now points to the next node]
[end of step-2 loop]
5. Return

(B) ~~WAP~~ Write a procedure to print an element

PRINT (INFO, LINK, START)
[This procedure is used to print an element in linked list]
(i) Set PTR = START [initialises pointer PTR]
2. Repeat step 3 & 4 while PTR ≠ NULL
3. WRITE INFO[PTR]
4. Set PTR = LINK[PTR]
(end of step-2 loop)
5. Return

(iii) Write a procedure to count the no. NUM of elements in a linked list.

Ans – COUNT (INFO, LINK, START, NUM)

1. Set NUM = 0 [Initialises COUNTER]
2. Set PTR = START [initialises PTR]
3. Repeat step 4 & 5 while PTR ≠ NULL
4. NUM = NUM + 1
5. PTR = LINK[PTR]
6. Return

(B) Searching ⇒ There are two type of linked list in case of search (i) sorted linked list
(ii) Unsorted linked list

(i) Sorted linked List ⇒ Let suppose we have a LIST in memory and the elements in this LIST are in sorted form. We travers the Linked LIST by using a pointer PTR and compare ITEM with the contain of INFO[PTR] of each node one by one. However we can stop once ITEM exceeds INFO[PTR]

Algo :—

SRCHSL (INFO, LINK, START, ITEM, LOC)

This algorithm finds the location LOC of the node where ITEM first appears in the list or sets LOC = NULL. We will consider LIST as a sorted list in memory.

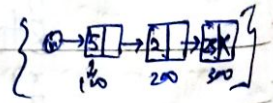
1. Set PTR = START [initialises pointer]
2. Repeat step 3 while PTR ≠ NULL

3. If ITEM > INFO[PTR]
then set PTR = LINK[PTR]
else if ITEM = INFO[PTR]
then set LOC = PTR and exit
else
Set LOC = NULL and exit
[end of if structure]

(ii) Unsorted linked list :—

SEARCH(INFO, LINK, START, ITEM, LOC)

1. Set PTR = START
2. Repeat 3 while PTR ≠ NULL
3. If ITEM = INFO[PTR] then
set LOC = PTR and exit
else
set PTR = LINK[PTR]
[end of if structure]
[end of step 2 loop]
4. when search is unsuccessful
Set LOC = NULL
5. Exit



Memory allocation Garbage Collection ⇒
The maintenance of linked list in memory assumes the possibility of inserting new nodes into the list and hence required some machnism which provides unused memory space for new nodes for which a special list is

maintained which consists of unused memory cells. This list has its own pointer AVAIL and its called as list of available space/ free storage list/ free pool.

Garbage collection:— The operating system of a computer may periodically collect all the deleted space onto the free list. Any technique which does this collection is called garbage collection. Garbage collection usually takes place in 2-steps (A) The computer runs through all list taging those cells which are currently used and then the computer runs through the memory collecting all untaged space onto the free storage list. (B) The Garbage collection may take place when there is only some minimum amount of space or no-space at all left in free storage list. or when the CPU is ideal and has time to do the collection.

Insertion into a linked list:— Whenever we will talk about insertion in a list we have 3 cases to perform insertion

(i) Insertion at the begining of the list.
(ii) Insertion after a giving location.
(iii) Insertion when the list is sorted.

(i) Insertion at the begining of the list ⇒ Whenever we have to insert an elemant at the begining of a list

we will consider list is unsorted until unless its stated or implied. Insertion ~~at the~~ of a node at the beginning is easiest task.

Algo :-

INFIRST (INFO, LINK, START, AVAIL, ITEM)

This algorithm inserts item as the first node in the list.

1. [Overflow ?] If AVAIL = NULL then write OVERFLOW and exit
2. [Remove the first node from AVAIL list] Set NEW = AVAIL and AVAIL = LINK[AVAIL]
3. Set INFO[NEW] = ITEM [copy value to new node]
4. Set LINK[NEW] = START [new node points to the original first node]
5. Set START = NEW [changes start so it points to the new node]
6. Exit.

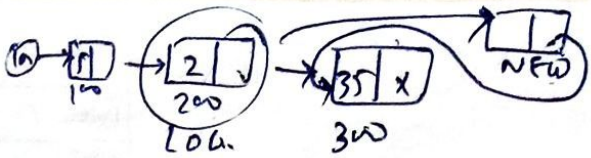
(ii) Insertion after a giving location ⇒

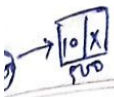
Algo :-

INSLOC (INFO, LINK, START, AVAIL, ITEM, LOC)

This algorithm inserts item so that item follows the node with location LOC or inserts item as the first node when LOC = NULL

1. [Overflow ?] If AVAIL = NULL then write OVERFLOW and exit
2. [Remove the first node from the AVAIL list] Set NEW = AVAIL and AVAIL = LINK[AVAIL]
3. Set INFO[NEW] = ITEM [copy value to new node]
4. If LOC = NULL then [insert new node as first node of your linked list





LINK [NEW] = START and START = NEW
else [insert the new node after the location LOC]
Set LINK [NEW] = LINK[LOC]
LINK[LOC] = NEW
[end of if structure]

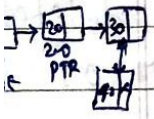
5. Exit

(iii) Insertion when the list is sorted ⇒ Let suppose we have a sorted list in memory and we want to insert an in item then the item must be inserted b/w node A & B

INFO(A) < ITEM ≤ INFO(B)

Procedure :-

FINDA (INFO, LINK, START, ITEM, LOC)



This procedure finds the location LOC of the last node in the sorted list such that INFO[LOC] < ITEM or sets LOC = NULL

1. [List empty?] If START = NULL then set LOC = NULL and return
2. [Special case?] If ITEM < INFO[START] then set LOC = NULL and return.
3. Set SAVE = START and PTR = LINK[START]
[initialises pointer]
4. Repeat step 5 while PTR ≠ NULL
5. If ITEM <= INFO[PTR] then
set LOC = SAVE and Return
[end of if structure]

6. else Set SAVE = PTR and PTR = LINK[PTR]
[update pointers]
[end of step 4 loop]
7. Set LOC = SAVE and
8. Return

Algo:-

INSERT (INFO, LINK, START, AVAIL, ITEM)
This algorithm inserts item into a sorted list.

1. Call FINDA (INFO, LINK, START, ITEM, LOC)
2. Call INSLOC (INFO, LINK, START, ITEM, AVAIL, LOC)
3. Exit

Deletion in linked list ⇒

Let list be a linked list in memory with node N ~~is deleted~~ b/w node A & B. Let suppose N is deleted form list. In that case pointer of node A will points to the node B means A will points to the location that was pointed by ~~poin~~ node N before deletion.

→ There are two special cases in deletion:-
(i) When N is the first node of the linked list and when N is the last node of link list.
(i) If N is the last node of the link list in that case link A will contain NULL in its link part and (ii) if the deleted node from the list is first node of the linked list then START will point to node B.

Deletion Algorithm:—

Whenever we have to perform deletion on linked list there are two cases for it-

(i) Deleting the node following in given node.

Algo:

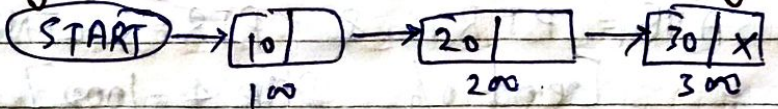
DELETE(INFO, LINK, START, LOC, LOCP, AVAIL)

This algorithm deletes the node N with location LOC, LOCP is the location of node which precides N or when N is the first node of the list. In that case LOCP = NULL

Step-

1. If LOCP = NULL then set START = LINK[START]
   [deletes first node from the list]
   else set LINK[LOCP] = LINK[LOC]
   [Deletes node N]
   [end of if structure]
2. [Return the deleted node to AVAIL list]
   Set LINK[LOC] = AVAIL and
   AVAIL = LOC
3. Exit

(ii) Deleting the node with a giving item of information:-



Let list be a linked list in memory and supposed we have, giving a item of information and we want to delete from list the first node N which contains items. Means first of all we have to find the location of first appearence of item in the list and after that we have to update the pointers of the

link accordingly.

Procedure :—

FIND B (INFO, LINK, START, ITEM, LOC, LOCP)

This procedure finds the location LOC of the first node which contains item & the location LOCP of the node preciding N. If item does not appear in the list. Then the procedure sets LOC = NULL and if item appear in the first node LOCP = NULL.

Step - 1. [list empty ?] If start = NULL then set LOC = NULL & LOCP = NULL and return (end of if structure)

2. [item appear in first node ?] if INFO[START] = ITEM then set LOC = START & LOCP = NULL [end of if]

3. Set SAVE = START & PTR = LINK[START] (initialises pointer)

4. Repeat step 5 & 6 while PTR ≠ NULL.

5. If INFO[PTR] = ITEM then set LOC = PTR & LOCP = SAVE and return (end of if structure).

6. Else SAVE = PTR & PTR = LINK[PTR] [end of step 4 loop]

7. Set LOC = NULL

8. Return

Algorithm's

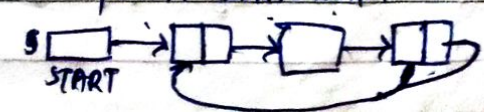
Algorithm :- DELETE (INFO, LINK, START, ITEM, AVAIL)
This algo deletes from a link list the first node N which contains the given item of information

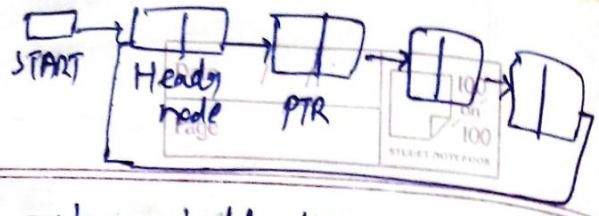
Up- 1. Call FIND B(INFO, LINK, START, ITEM, LOC, LOCP)
2. If LOC = NULL then write item doesn't exist in list & exit.
else set LINK[LOCP] = LINK[LOC]
[end of if structure]
3. Return the deleted node to AVAIL list
set LINK[LOC] = AVAIL
AVAIL = LOC
4. Exit.

Header linked list ⇒ A header linked list is a linked list that contains a special node known as header node in the list. This header node will contain some special information about the list like how many nodes we have in list?, what kind of information we have stored in linked list?
There are two types of Header linked list
(i) Grounded header
(ii) Circular header

(i) Grounded header :- In grounded header, linked list the last node of linked list will contain null.

(ii) Circular Header :- Last node of the linked list will point backtoward the header node of the list.





If LINK[START] = NULL, it indicates that the grounded header is empty. And
If LINK[START] = START then it indicates that circular header list is empty.

Algorithm for traversing a circular list :-

Let list be a circular list in memory this algorithm traverses list, appling an operation process to each node of the list.

1. Set PTR = LINK[START]
2. Repeat step 3 & 4 while PTR ≠ START
3. Apply process to INFO[PTR]
4. Set PTR = LINK[PTR] [update pointer]
   ~~Exit~~ [end of loop 2]
5. Exit

Q. Write algorithm to perform deletion in circular header from (i) a specific location
(ii) where an item of information given

(ii) FINDBHL (INFO, LINK, START, ITEM, LOC, LOCP)

1. Set SAVE = START & PTR = LINK[START]
2. Repeat while INFO[PTR] ≠ ITEM and PTR ≠ START
   Set SAVE = PTR & PTR = LINK[PTR]
   [end of loop]
3. If INFO[PTR] = ITEM
   Set LOC = PTR & LOCP = SAVE
   else
   LOC = NULL & LOCP = SAVE
   [end of if structure]
4. Exit

(ii) DELLOCHL (INFO, LINK, START, AVAIL, ITEM)

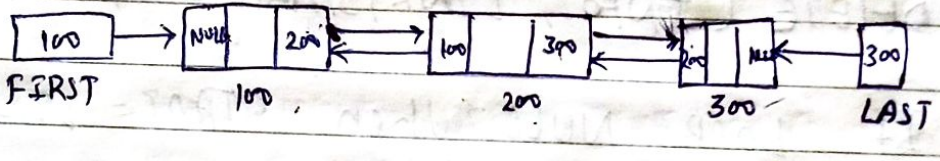
1. Call FINDBHL(INFO, LINK, START, ITEM, LOC, LOCP)
2. If LOC = NULL then write ITEM not in list & Exit
3. Set LINK[LOCP] = LINK[LOC]
   [Return deleted node to AVAIL list]
4. Set LINK[LOC] = AVAIL
   AVAIL = LOC
5. Exit.

(i) DELETE (INFO, LINK, START, LOC, LOCP, AVAIL)

1. If LOCP = NULL then START = LINK[START]
   else
   LINK[LOCP] = LINK[LOC]
   [delete node N] [end of if structure]
2. [Return the deleted node to AVAIL list]
   Set LINK[LOC] = AVAIL and
   AVAIL = LOC
3. Exit.

(Double linked list)

2-way list ⇒ In 2-way list, we have two pointers for the nodes that will be used. These pointers are known as previous and next. We can travers the two-way list in forward & backward direction The first node's address from the left is held by first / LEFT ~~next~~ and last node's address from the left ~~and~~ or the first node's address from the right will be held by special pointer that is known as LAST / RIGHT



Operation of 2-way list:-

The following operation can be performed in 2-way list

(i) Insertion (ii) Deletion (iii) Traversing (iv) Searching

(i) Deletion in 2-way list:- Delete node at Particular location LOC.

Algo- DELETWL (INFO, START, FORW, BACKW, LOC, AVAIL)

Step-1 [Delete the node]

Set FORW[BACKW[LOC]] = FORW[LOC]

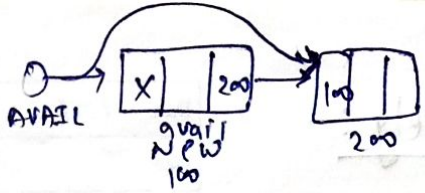
and BACKW[FORW[LOC]] = BACKW[LOC]

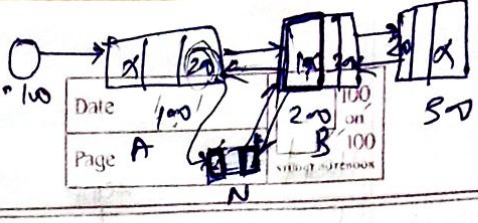
Step-2 [Return the deleted node to AVAIL List]

FORW[LOC] = AVAIL

and AVAIL = LOC

Step-3 Exit.





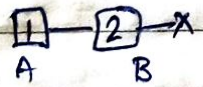
(ii) Algorithm to perform insertion in 2-way list:-

INSTWL (INFO, START, FORW, BACKW, LOCA, LOCB, AVAIL, ITEM)

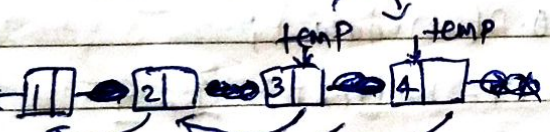
1. [Overflow?] if AVAIL = NULL then write overflow and exit.
2. [Remove the node from AVAIL list and copy data into new node]
   Set NEW = AVAIL and
   AVAIL = FORW[AVAIL],
   INFO[NEW] = ITEM
3. Set FORW[LOCA] = NEW,
   FORW[NEW] = LOCB
4. Set BACKW[LOCB] = NEW,
   BACKW[NEW] = LOCA
5. Exit

Q. (A) Is it possible to reverse a Singly list and if yes then how?
(B) What are the advantages & disadvantages of singly & doubly linked list?

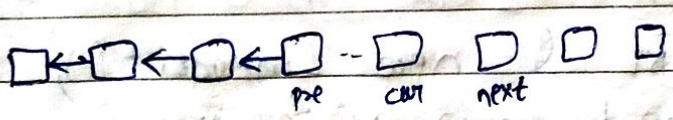
(A)



Link[B] = A Reversed.
Link[A] = Null

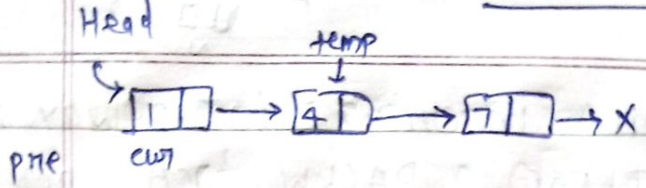


Link[temp] = Prev
temp++;



Link[cur] = pre

Iteratively :-




```
class LinkedList {
Node reverse(Node head) {
    Node cur = head
    Node pre = null
    while (cur != null)
    { Node temp = cur.next;
      cur.next = pre;
      pre = cur;
      cur = temp;
    }
    return pre; }
```

~~Recursively~~ :-

```
static Node head;
static class Node { int data;
                    Node next;
    Node(int d) { data = d;  next = null; }
                }

Void print(Node n) {
    while( n != null)
    { s.o.p( n.data );
      n = n.next; } }

psvm( )
{   LinkedList l = new LinkedList( );
    l.head = new Node (1);
    l.head.next = new Node (4);
    l.head.next.next = new Node (7);
    head = l.reverse(head);
    l.print(head); } }
```

Unit - 3
Searching & Sorting

Searching & Sorting operation can be performed on array, which is a linear data structure.

→ **Searching :-** As the name implies we have to search an element in data structure (array) The searching can be of two type

(i) Linear search
(ii) Binary Search

→ **Sorting :-** This operation is used to sort the elements of data structure (array). Their are various types of sorting techniques like insertion sort, selection sort, radix sort, quick sort, merg sort etc.

Searching :-

(i) Linear Search (element by element) ⇒ (Sequencial Search)

In A linear Search, array elements will be in unsorted form. It is also known as element by element or sequencial search.

Algo (Linear Search)

LINEAR(DATA, N, ITEM, LOC)

Here DATA is linear array with N elements and ITEM is a given item of information. This algorithm find the LOC location of item in data or sets LOC = 0 if the search is unsuccessful

1. [insert item at the end of data]
   Set DATA[N+1] = ITEM
2. [initializes counter] Set LOC = 1
3. [Search for item]
   Repeat while DATA[LOC] ≠ ITEM
   Set LOC = LOC + 1
   [end of loop]
4. [Successful?] If LOC = N+1
   then set LOC = 0
5. Exit

Complexity of linear Search:—

→ In the worst case, we have to search the entire array and if the element does not appear in DATA then the algorithm require $f(n) = n+1$ comparisons thus in worst case the running time is proportional to ~~worst time~~ n.

→ In average case, we use the probabilistic notation. Let suppose P_K is the probability that the element appear to DATA[K] and suppose q is the probability when Item does not appear in DATA. Since the algorithm uses k comparison when ITEM appears in DATA[k] the average no of comparison is given by

$$f(n) = 1 \cdot P_1 + 2 \cdot P_2 + n \cdot P_K + (n+1)q$$

In particular suppose, q is very small and item appear with equal probability in each element of DATA then $q = 0$ & $p_i = \frac{1}{n}$

then $$f(n) = 1 \cdot \frac{1}{n} + 2 \cdot \frac{1}{n} + \cdots \frac{n \cdot 1}{n}$$

$$f(n) = \frac{1}{n}(1 + 2 + \cdots n)$$

$$f(n) = \frac{n(n+1)}{2} \times \frac{1}{n}$$

$$f(n) = \frac{n+1}{2}$$

i.e. in this special case, the average no of comparison required to find the location of ITEM is approximately is equal to half the no. of elements in the array.

Binary Search ⇒

In Binary Search, its mandatory that the array elements should be in sorted form.

In this we will take 3 variable BEG, END & MID we have to evaluate $MID = \frac{BEG + END}{2}$

This process of evaluating MID we have to repeat. As we have comparing the element to be find out with MID element of array accordingly the position of BEG & END will be changed. If the element > MID element then BEG will be changed to MID + 1 and if the element less than MID element

then END will be changed to MID - 1.

Limitation of Binary Search

(i) The list must be sorted.
(ii) One must have direct access to the middle element in the list.
(iii) The cost of storing sorted element in array/memory / Data structure is high.

Algorithm :—

BINARY (DATA, LB, UB, ITEM, LOC)
Here data is sorted array with Lower bound LB & upper bound UB & item is an given ~~element~~ item of information.
The variable BEG, END, MID denotes respectively Begining, end, middle position of the segment of elements of data.
This algorithm finds the location LOC of ~~ele~~ item in data or set LOC = NULL

1. [Initialise] segment variable]
   Set BEG = LB , END = UB
   MID = (BEG + END) / 2

2. Repeat Step 3 & 4 while (BEG <= END) and DATA[MID] ≠ ITEM

3. If ITEM < DATA[MID] then
   Set END = MID - 1
   else
   Set BEG = MID + 1
   [end of if structure]

4. Set MID = INT((BEG + END)/2)
[end of step 2].

5. If DATA[MID] = ITEM
then set LOC = MID
else
LOC = NULL
[end of if structure]

6. Exit

SORTING ⇒

Sorting means to ~~arrange~~ arrange the elements either in ascending or in descending order.

(i) **Selection Sort:—**

Let suppose we have an array A with n elements in memory. The selection sort algorithm for sorting will work as follows:
First, we find the smallest element in the list and put it in the first position then find the second smallest element in the list and put it in second position and so on.

Algorithm ⇒

Selection (A, n)

This algorithm sorts the array A with n elements.

1. Repeat step 2 & 3 for K=1, 2, 3 upto n-1
2. Call MIN (A, K, n, LOC)
3. [Interchange the value of A[K] and A[LOC]]
Set TEMP = A[K]

A[k] = A[LOC]
A[LOC] = TEMP

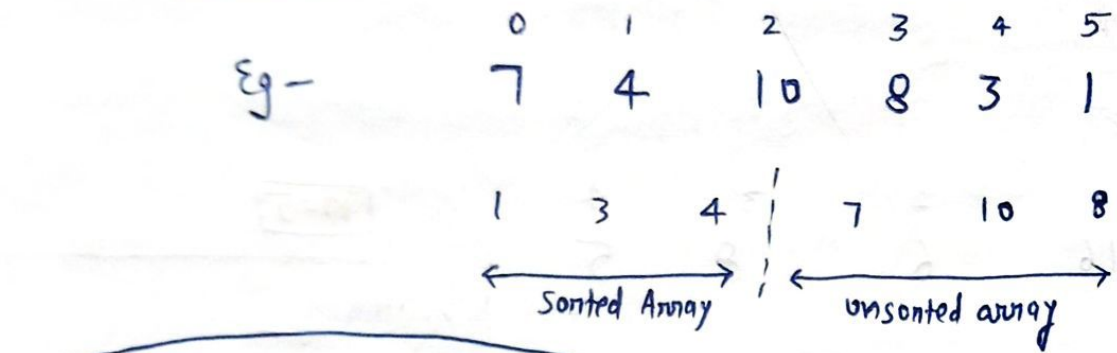
4. Exit

Procedure MIN(A, K, n, LOC)

An array A is in memory this procedure find the location LOC of the smallest element among A[K] to A[n]

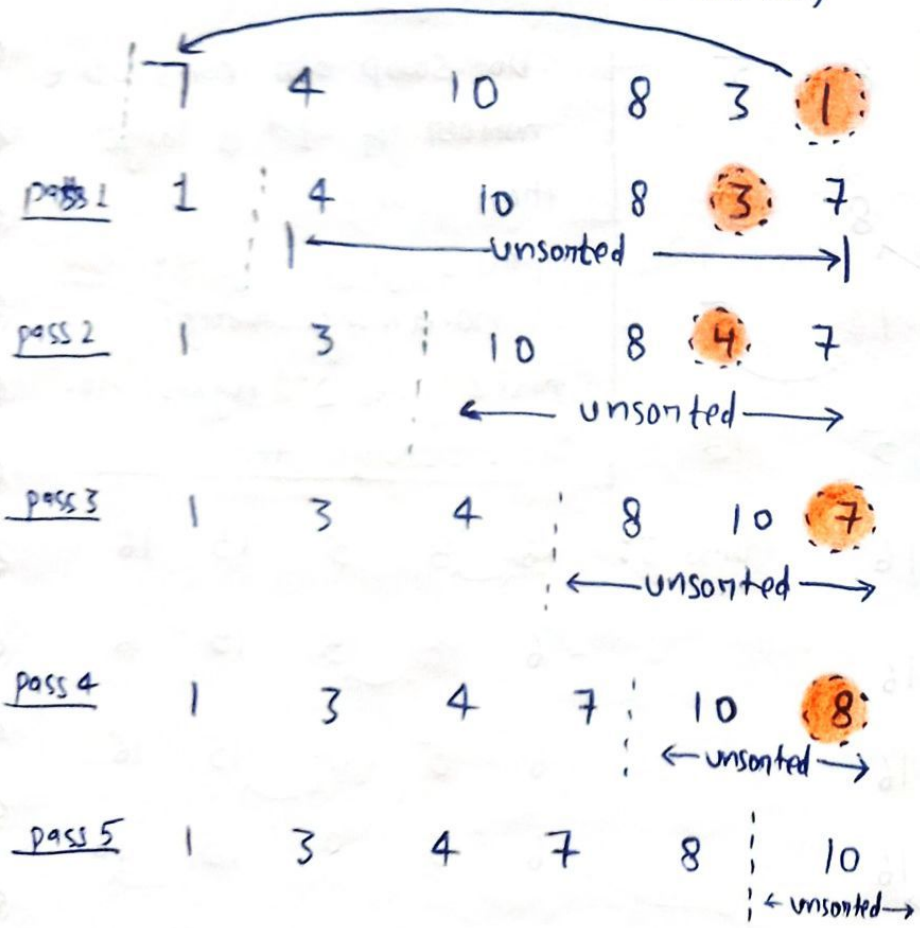
1. Set MIN = A[K] and LOC = K
   [initialise pointer]
2. Repeat for J = K+1, K+2 --- n
   if MIN > A[J] then
   Set MIN = A[J] and LOC = J
   [end of loop]
3. Return

(i) Selection Sort:-



n = 6

pass = n-1



find min element of array

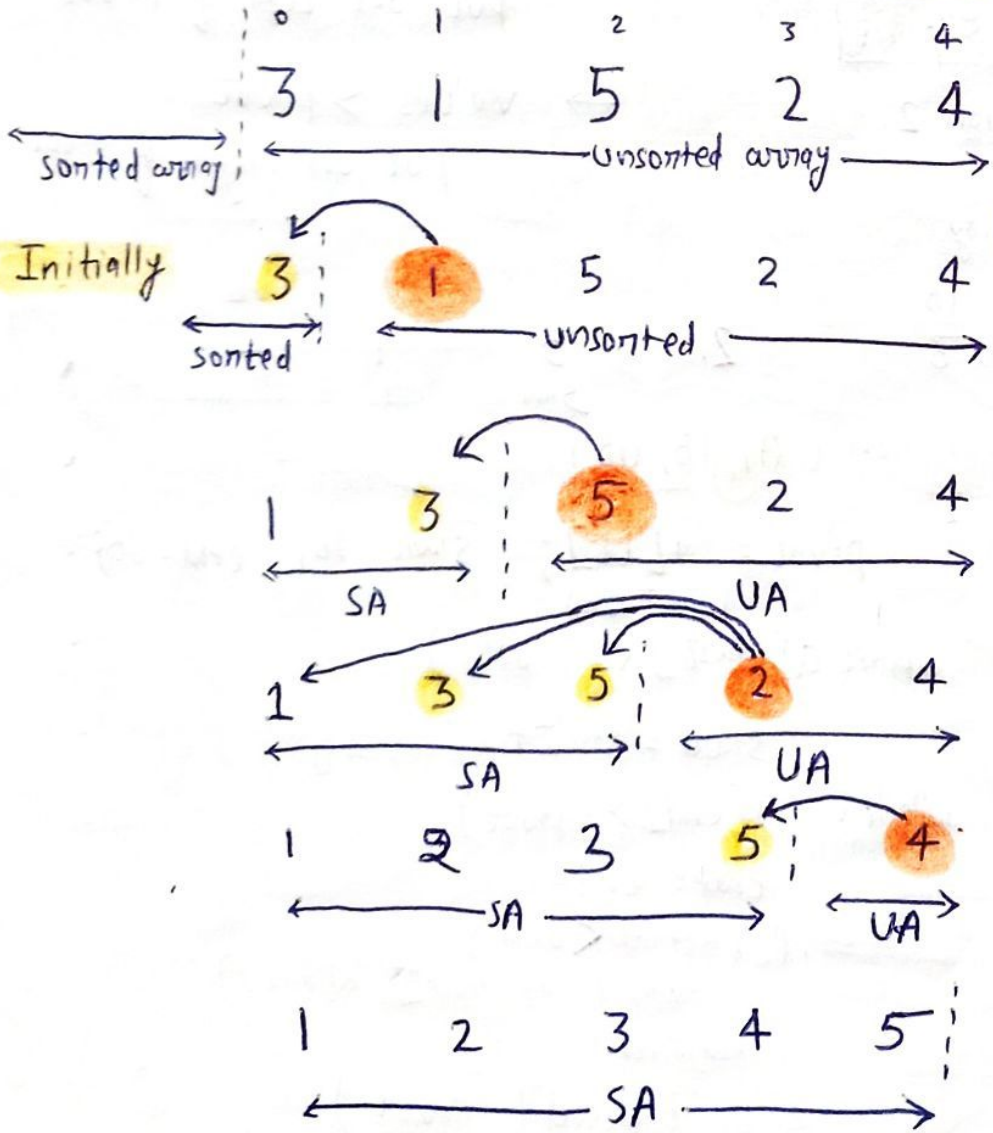
Working:-

Find out the min element from the unsorted array and place it at the 1st position of unsorted array.

Best case & Worst case:- Already sorted (But 2 loop) $O(n^2)$

(3) Insertion Sort ⇒

$n = 5$



working:-

→ It considers first element in sorted array at initially.

→ At a time, pick 1st element from UA and compare with SA. If it is min just do swap operation. and set on connect position.

→ Compare that element with each element of the SA.

Best case:- $O(n)$

Worst case:- (If array is in descending order)

i	Comparison
1	1
2	2
3	3
⋮	⋮
(n-1)	(n-1)

$1+2+3 \cdots (n-1)$

$$\frac{(n-1)(n-1+1)}{2} = \frac{n^2-n}{2}$$

$O(n^2)$

(ii) Insertion sort ⇒

Insertion sort works in the similar ways as the sort cards in our hand in a card game. We ~~can~~ assume that the card is already sorted then we selects a unsorted card if the unsorted card is greater than the card in hand it is placed on the right otherwise to the left. In the same way, other unsorted cards are taken & put at there right place a similar approach is used by insertion sort. Insertion sort is a sorting algorithm that places an unsorted element at its suitable position / place at each iteration.

Algo

InsertionSort (Array)

1. Mark first element as sorted
2. For each unsorted element X.
3. Extract the element X
4. For j ← last sorted Index down to zero
5. If current element j > X
6. Move sorted element to right by 1.
7. Break loop and insert X here and
8. InsertionSort (Array)

3. Quick Sort ⇒

Quick sort is an application of stack. Quick sort basically works on divide and conquer. It divides the array into sub array and we will apply the procedure on these sub-array and this sub arrays further divided into sub array. For these division of subarray will done by we the help of by pivot element.

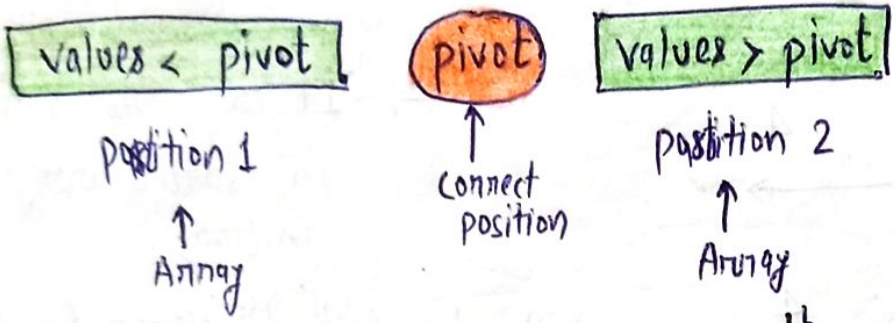
All the element to this pivot elements left will be smaller from it and all the elements from greater than pivot will decide towards right of pivot element. we also have to use left and right pointer other than pivot element.

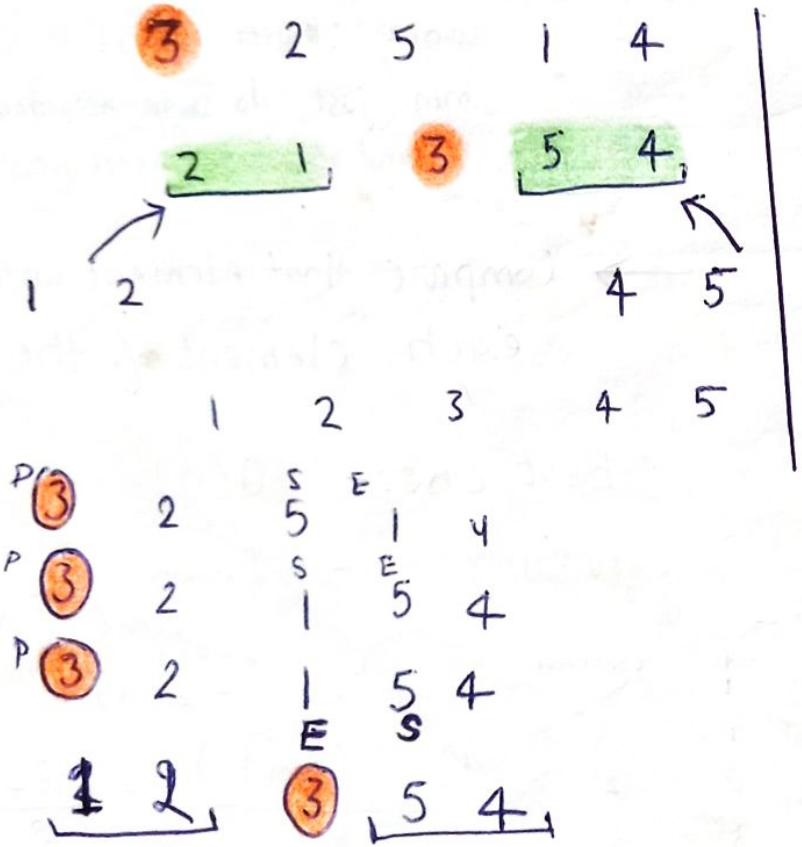
pivot

3 2 5 1 4

→ values < pivot
put on Left of pivot

→ values > pivot
put on right of pivot





lb ub

3 2 5 1 4

start end

```
partition ( A, lb, ub)
{
    pivot = a[lb];   start = lb;   end = ub;
    while ( start < end)
    while( a[start] <= pivot; )
        start = start + 1;
    while( a[end] >= pivot)
        end = end - 1;
    if ( start < end )
        swap ( a[start], a[end])
    end while
    swap ( a[end], pivot )
    return end
```

Code:-

```
partition (A, lb, ub)
{
    pivot = a[lb];   start = lb;   end = ub;
while ( start < end)
    {
        while (a[start] <= pivot)    start++;
        while (a[end] > pivot)       end++;
        if ( start < end)
            { swap(a[start], a[end]) }
    }
    swap( a[lb] , a[end])
}

Quicksort (A, lb, ub)
{
    if ( lb < ub )
        {
            loc = partition (A, lb, ub)
            Quicksort (A, lb, loc - 1);
            Quicksort (A, loc + 1, ub);
        }
}
```

WORST CASE:-
(Array is already sorted)

① ② ③ ④ ⑤
n n-1 n-2 ----

$n + n-1 + n-2 + \dots$

$\frac{n(n+1)}{2}$

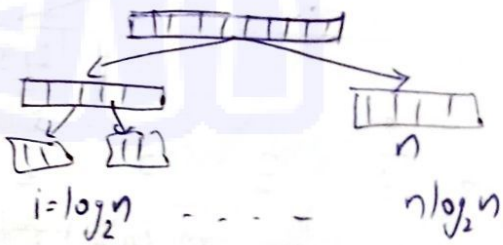
$O(n^2)$

Best Case:-
(If Array breaks from middle element)

$\frac{n}{2^i} = 1 \Rightarrow n = 2^i$

$i = \log_2 n$

$O(n \log n)$



We can optimize the Quicksort
In worst case, we can chose pivot to middle element & can be got $O(n \log n)$. So the worst case can be best case.

eg - consider the following set of element :—
44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66
and perform quick sort.

sol -

(44), 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66

22, 33, 11, 55, 77, 90, 40, 60, 99, (44), 88, 66

22, 33, 11, (44), 77, 90, 40, 60, 99, 55, 80, 66

22, 33, 11, 40, 77, 90, (44), 60, 99, 55, 80, 66

22, 33, 11, 40, (44), 90, 77, 60, 99, 55, 80, 66
(1) (2)

① → (22), 33, 11, 40
11, 33, (22), 40
11, (22), 33, 40

② → (90), 77, 60, 99, 55, 80, 66
66, 77, 60, 99, 55, 80, (90)
66, 77, 60, (90), 55, 80, 99
66, 77, 60, 80, 55, (90), 99
(3)

③ → (66), 77, 60, 80, 55
55, 77, 60, 80, (66)
55, (66), 60, 80, 77
55, 60, (66), 80, 77
(4)

④ → (80), 77 ⇒ 77, (80)

(4.) **Bubble Sort ⇒**

Bubble sort works same as a bubble in water.

Algo

BUBBLE (A, N)

Here A is an array with N elements. This algorithm sorts the element in A.

1. Repeat step 2 & 3 for k = 1 to N-1
2. Set ptr = 1
3. Repeat while PTR ≤ N-k [executes pass]
   (a) if A[PTR] > A[PTR+1] then interchange A[PTR] & A[PTR+1]
   [end of if structure]
   (b) set PTR = PTR+1
   [end of inter loop]
   [end of step 1 loop]
4. Exit

n=5

Eg- 0 1 2 3 4

15 16 6 8 5

Working:-

Do swap b/w consecutive number if 1st is large then swap.

pass 1 gives 1st highest no. among array.

pass 2 gives 2nd highest no.

---

pass-1 :-

15 16 6 8 5

15 6 16 8 5

15 6 8 16 5

15 6 8 5 16

pass-2 :-

15 6 8 5 16

6 15 8 5 16

6 8 15 5 16

6 8 5 15 16

6 8 5 15 16

pass-3:

6 8 5 15 16

6 8 5 15 16

6 5 8 15 16

6 5 8 15 16

6 5 8 15 16

pass 4:

6 5 8 15 16

5 6 8 15 16

5 6 8 15 16

5 6 8 15 16

5 6 8 15 16

j = 5-3

j = n-i

5 6 8 15 16

This is a sorted array.

(5) Radix Sort ⇒

Radix sort is the method used by many people to arrange list according to alphabetically There are 26 alphabets so we will create 26 classes to arrange ~~that~~ then this method can also be used to arrange no. we will create 10 classes to arrange 0 to 9.

eg. ~~Let suppose we have following no.~~ for performing radix sort we will use a radix number. By using the radix

no. we will evaluate remainder from the given series & arranged them accordingly.

eg - Let suppose we have following numbers 348, 143, 361, 423, 538, 128, 321, 543, 366 and the radix no. is 10. sort the series using radix sort.

Ans -

Input	0	1	2	3	4	5	6	7	8	9
348									348	
143				143						
361		361								
423				423						
538									538	
128									128	
321		321								
543				543						
366							366			

New series is

361, 321, 143, 423, 543, 366, 348, 538, 128

Input	0	1	2	3	4	5	6	7	8	9
361							361			
321			321							
143					143					
423			423							
543					543					
366							366			
348					348					
538				538						
128			128							

Now, new series will be :—

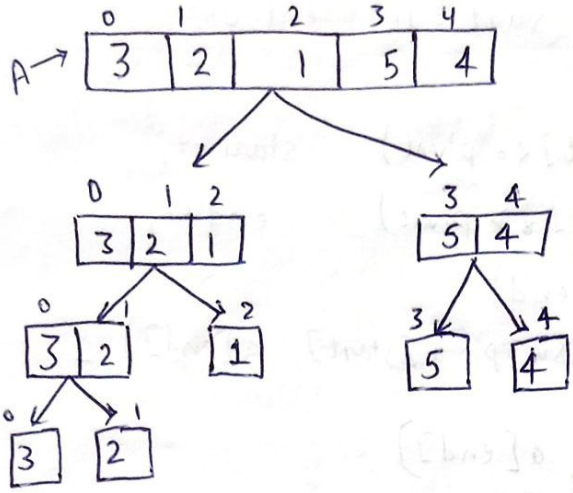
321, 423, 128, 538, 143, 543, 348, 361, 366

Input	0	1	2	3	4	5	6	7	8	9
321				321						
423					423					
128		128								
538						538				
143		143								
543						543				
348				348						
361				361						
366				366						

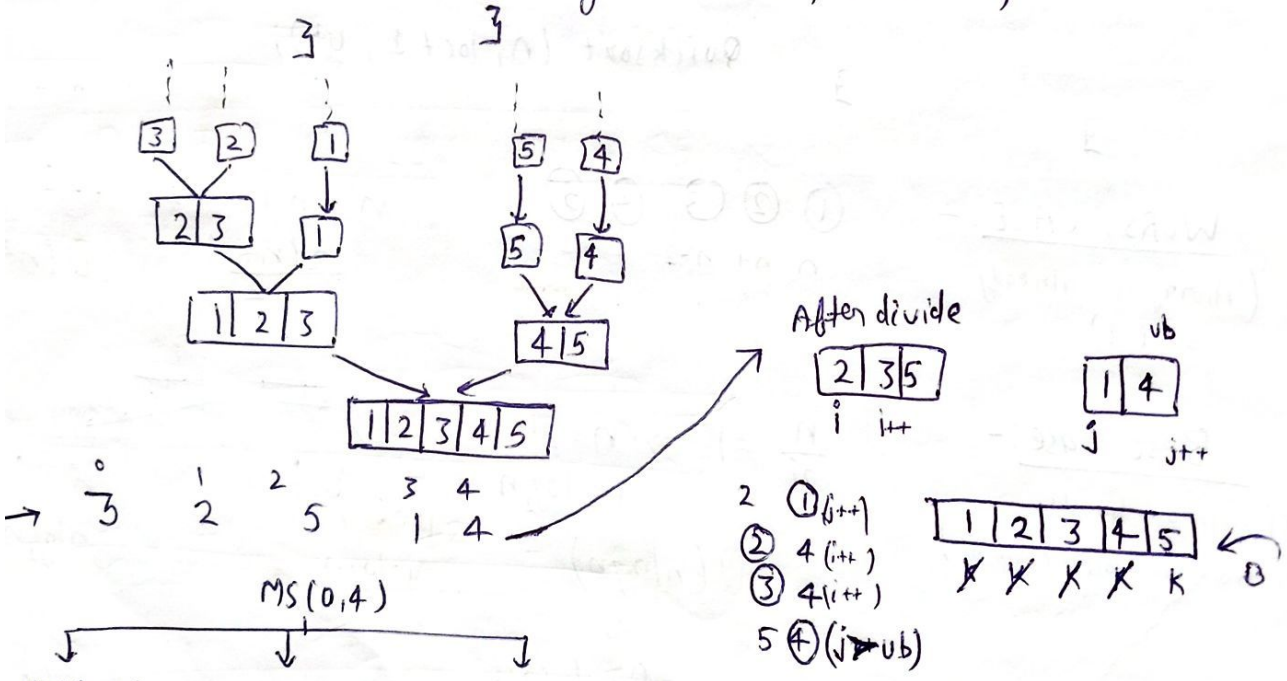
The result is

128, 143, 321, 348, 361, 366, 423, 538, 543

) **Merge sort** ⇒ → Divide & conquer method




```
MergeSort (A, lb, ub)
{   if (lb < ub)
    {   mid = (lb+ub)/2;
        mergesort (A, lb, mid);
        Mergesort (A, mid+1, ub);
        merge ( A, lb, mid, ub);
    }
}
```




```
merge (A, lb, mid, ub)
{   i = lb;   j = mid+1;   k = lb;
    while (i <= mid && j <= ub)
        if (a[i] <= a[j])
        {   b[k] = a[i];
            k++;
            i++;
        }
```

```
else {    b[k] = a[j];
          k++;
          j++;
    }}
    if (i > mid)
    {     while( j <= ub)
                { b[k] = a[j]; k++; j++; }}
    if ( j > ub)
    {     while( i <= mid)
                { b[k] = a[i]; k++; i++; }}
```

PAGE NO.:
DATE: / /

Merge Sort ⇒ Merge sort works on divide & conquer approach.

Let suppose we have the array with following elements

66, 33, 40, 22, 55, 88, 60, 11, 80, 20, 50, 44, 70, 30

Each pass of merge sort algorithm will start at the begining of array and merge pairs of sorted sub-arrays as follows.

Pass-1 : Merge each pair of elements to obtain the following list of sorted pairs. As following:—

33, 66 , 22, 40 , 55, 88 , 11, 60 , 20, 80 , 44, 50 , 30, 70

Pass-2 : Merge each pair of pairs to obtain the following list of sorted quarturals.

22, 33, 40, 66 , 11, 55, 60, 88 , 20, 44, 50, 80 , 30, 70

Pass-3 : Merge each pair of sorted quadruplus to obtain the following to sorted sub-array.

11, 22, 33, 40, 55, 60, 66, 88 , 20, 30, 44, 50, 70, 80

Pass-4 : Merge the two sorted sub array to obtain the single sorted array.

11, 20, 22, 30, 33, 40, 44, 50, 55, 60, 66, 70, 80, 88

Algorithm ⇒

Mergesort (A, N)

This algorithm sorts the N element of array A using an auxillary array B

1. Set L = 1 (initialises the no. of elements in sub array)

Teacher's Signature.....

2. Repeat step 3 to 6 while ($L \leq N$)
3. call MERGEPASS (A, N, L, B)
4. Call MERGEPASS (B, N, 2*L, A)
5. Set L = 4*L
[end of step 2]
6. Exit

Procedure :—
MERGEPASS (A, N, L, B)
The N element array A is composed of sorted sub array where each sub array of L elements accept possibly the last sub-array which may have fewer than L elements. The procedure merges the pairs of sub-arrays of A and assigned them to the array B.

1. Set Q = INT (N/(2*L)), S = 2*L*Q, and R = N-S
2. [procedure merge to merge the Q pairs of sub array]
Repeat for j = 1, 2 upto Q
(a) Set LB = 1 + (2*J - 2) * L [find LB of first array]
(b) call MERGE (A, L, LB, A, L, LB+L, B, LB)
[end of loop]
3. [only one sub array left ?]
if R ≤ L then repeat for J = 1, 2 --- R
set B(S+J) = A(S+J)
[end of loop]
else
call MERGE (A, L, S+1, A, R, L+S+1, B, S+1)

Teacher's Signature.....

[end of if structure]

4. Return

Procedure :—

MERGE(A, R, LBA, S, LBB, C, LBC)

This procedure merges the sorted array A & B into the array C.

Set NA = LBA , NB = LBB , PTR = LBC ,
UBA = LBA + R - 1

Same as algorithm merging accept R replaced by UBA & S by UBB

Same as algorithm merging accept R replaced by UBA & S by UBB

~~Exit~~ Return

(7) Counting Sort ⇒

Counting sort is a stable sorting technique which is used to sort object according to the keys that are small numbers. It counts the no. of keys whose values are same this sorting technique is effective when the difference b/w different keys are not so big otherwise it can increase the space complexity.

(i) It is sorting algorithm in which we sort a collection of elements based on numeric keys.

(ii) In this algorithm, we don't compare elements while sorting.

(iii) It is often used as a sub-routine in other sorting algorithms.

Teacher's Signature

There are two cases in count sort :-

(i) When we have unique no. / elements in array.

(ii) When we have repeated numbers in array

Case-1

When we have unique no./elements in array

eg- I/p → 10 7 12 4 9 13

First we will find the minimum & maximum value from the given element by scaning array.

min = 4 max = 13

So our range is 4 to 13. Create index from 4 to 13.

Index	4	5	6	7	8	9	10	11	12	13

Pass-1

Create an array that will hold the count of each number.

COUNT	1	0	0	1	0	1	1	0	1	1

Pass-2

Now that we have counted all the numbers in the input we can fill the empty space in count array with zero.

Now create ~~some~~ sumcount array that will hold the sum ~~some~~ of count for given index

1	0	0	1	0	1	1	0	1	1

SUM COUNT	1	1	1	2	2	3	4	4	5	6

Now that we have filled the sumcount array its time to sort the input

Teacher's Signature

using sumcount.

There are 6 numbers in input so we will create 6 positions and filled those positions with the given no. as per the sumcount

Position	1	2	3	4	5	6

First we will scan the array's first element and its 10 w.r.t. index 10, we check sumcount and its value is 4 so we will insert 10 at position 4 and after the reduce the sumcount at that place by 1. And repeat the same process for the element of array.

eg- I/P → 10 7 12 4 9 15

Index	4	5	6	7	8	9	10	11	12	13
sumcout	1	1	1	2	2	3	4	4	5	6

Pass-1

	1	2	3	4	5	6
				10		

Pass-2

Index	4	5	6	7	8	9	10	11	12	13
sumcout	1	1	1	2	2	3	3	4	5	6

pass-3 sumcout

1	1	1	1	2	3	3	4	5	6

		7				10			
1		2		3		4		5	6

pass-4 sumcount

1	1	1	1	2	3	3	4	4	6

1	2	3	4	5	6
7		10		12	

pass-5 sumcount

0	1	1	1	2	3	3	4	5	6

1	2	3	4	5	6
4	7		10	12	

pass-6 Sum count

0	1	1	1	2	2	2	3	4	5	6

1	2	3	4	5	6
4	7	9	10	12	

4	7	9	10	12	15
1	2	3	4	5	6

Case-2 When we have repeated no. in array ⟹

I/p → 9, 4, 10, 8, 2, 4

Firstly we will find min. & max. from the element of array by scanning it.

min = 2 , max = 10

so our range is 2 to 10

create index from 2 to 10

Index	2	3	4	5	6	7	8	9	10
count	1	0	2	0	0	0	1	1	1
sumcount	1	1	3	3	3	3	4	5	6
Position	1	2	3	4	5	6			

Pass - 1 i/p → 9, 4, 10, 8, 2, 4

index	2	3	4	5	6	7	8	9	10
sumcount	1	1	3	3	3	4	5	6	

1	2	3	4	5	6
					9

Pass - 2

sumcount	1	1	3	3	3	4	4	6

position	1	2	3	4	5	6
			4		9	

sumcount

pass - 3

1	1	2	3	3	3	4	4	6

1	2	3	4	5	6
4			9	10	

pass - 4

sumcount

1	1	2	3	3	3	4	4	5

1	2	3	4	5	6
4	8	9	10		

pass - 5

sumcount

1	1	2	3	3	3	3	4	5

1	2	3	4	5	6
2	4	8	9	10	

pass - 6

sumcount

0	1	2	3	3	3	3	4	5

1	2	3	4	5	6
2	4	4	8	9	10

pass - 7

sumcount

0	1	1	3	3	3	3	4	5

position

1	2	3	4	5	6
2	4	4	8	9	10

Algorithm	Best	Average	Worst
Quick Sort	$\Omega(n \log n)$	$\theta(n \log n)$	$O(n^2)$
Bubble Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$
Merge Sort	$\Omega(n \log n)$	$\theta(n \log n)$	$O(n \log n)$
Insertion Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$
Selection Sort	$\Omega(n^2)$	$\theta(n^2)$	$O(n^2)$
Heap Sort	$\Omega(n \log n)$	$\theta(n \log n)$	$O(n \log n)$
Radix Sort	$\Omega(nk)$	$\theta(nk)$	$O(nk)$
Bucket Sort	$\Omega(n+k)$	$\theta(n+k)$	$O(n^2)$

- Quick Sort is better than other sorting Algorithm. Quicksort is very fast if you need a comparison based sort.

→ Quicksort doesn't have to allocate any extra space but Mergesort does.

→ Locality of reference for Arrays

→ Tail recursive Algorithm [optimization]

16-September

→ Binary tree is a tree data structure in which each node has at most two children which are referred to as left child and the right child.

Unit-4
Tree

Tree is a non-linear data structure and it is used to represent data in Hierarchical manner.

(i) Binary Tree ⇒

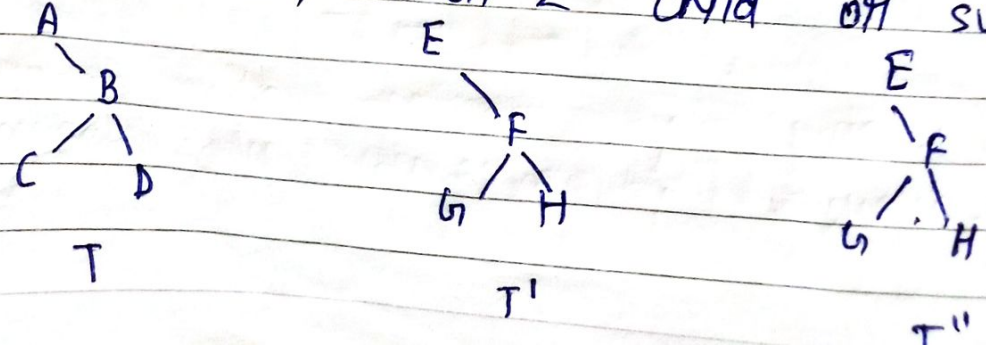
Binary Tree T is defined as a finite set of elements called nodes as;

(a) T is empty (called NULL tree or empty tree)
or
(b) T contains distinguish node R called the root of T and the remaining nodes of T form N ordered pair of disjoint binary tree T1 & T2.

Binary tree T & T' are similar if they have the same structure or in other words if they have the same shape. The trees are said to be copies if they are similar and if they have the same contents at corresponding nodes.

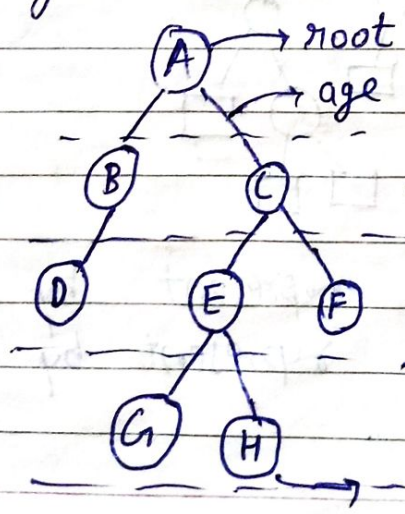
→ In binary tree a node N can have either zero, 1 or 2 child or successors

eg-



{ T' & T'' are copies tree
T & T' are similar tree }

Terminologies :- Terminology describes family relationship and it is used to describe relationship between the different nodes of the tree T. N is a node in T with left successor S_1 and right successor S_2 then N is called the parent of S_1 & S_2. S_1 is called the left child of N or son of N or S_2 is called right child or son of N. S_1, S_2 are said to be siblings. Every node N in binary tree except the root has a unique parent called the predicar of N. The line drawn from N of T to a successor is called an age and the sequence of consequetive eages is called a path. A terminal node is called a leaf and a path ending in a leaf is called a branch. Those nodes with the same level no. are said to belong to the same generation.

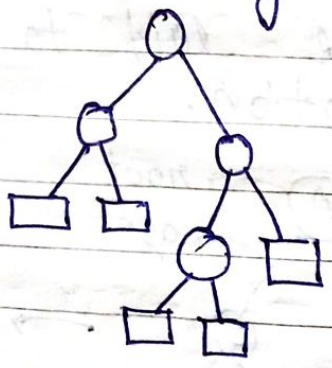


Complete binary Tree ⇒ A tree T is said to be complete if all its level except possibly the last have the maximum no. of possible node and if all the nodes at the last level appear as far as possible thus there is a unique complete tree Tn with exactly n nodes. All nodes are as far left as possible. The depth of complete binary tree with small n no. of nodes is given by

$$Dn = \lfloor \log_2 n + 1 \rfloor$$

Extended binary tree ⇒ A binary tree is said to be a 2-Tree or extended binary tree if each node N has either 0 or 2 children. In such a case, the nodes with 2 children are called <u>internal nodes</u> & the nodes with 0 children are called <u>external nodes</u>.

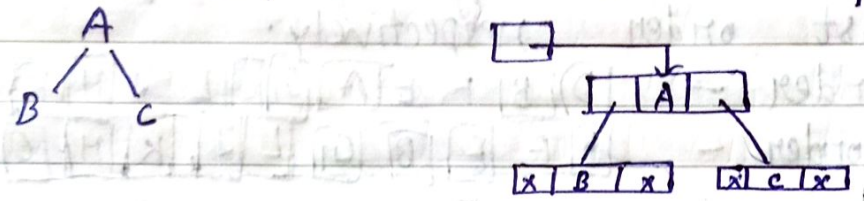
Consider the following tree



→ Internal nodes represent by circle.
→ External nodes represent by square.

Representing Binary tree in Memory ⇒ There are two ways by which we can represent binary tree in memory. The first & useful way is called linked representation. The second way by using a single array called the sequencial representation or array representation.

Linked representation of binary tree ⇒



Each node in liked representation of binary tree consists of 3 parts info, left pointer & right pointer. Left pointer will point toward left sub tree / node or right pointer will point toward right sub tree. If a node doesn't have left & right child then NULL will be entered w.r.t. that pertiodar node in its left & right pointer field.

Array representation of binary tree ⇒

For representing the binary tree then we requires 3 array info, left, right respectively. But to represent tree using in sequencial manner we consider an array with name tree and enter values in it.

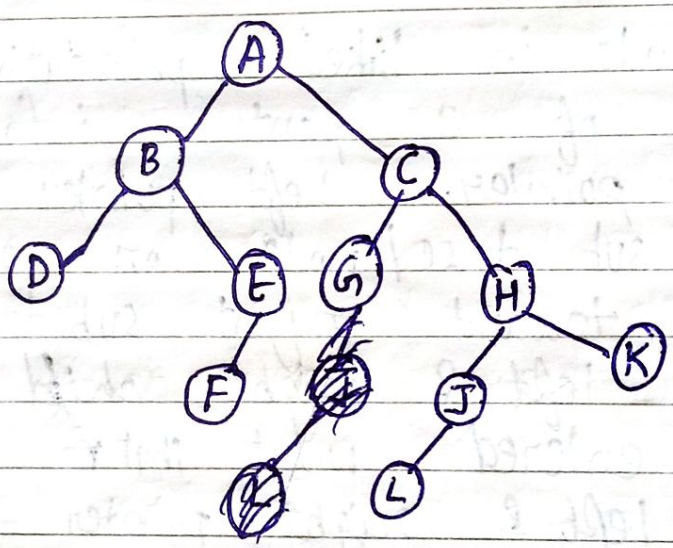
Traversing in Tree ⇒

We can travers a binary tree in three ways.

(i) In order traversion. (L Rt R)
(ii) Pre order traversion. (Rt L R)
(iii) Post order traversion (L R Rt)

Q. Consider the following sequence of In order & post order respectively.

In order – D B F E A G C L J H K
Post order – D F E B G L J K H C A



pre order :–

A B D E F C G H J L K

Traversal Algorithm using stack ⇒

(i) Pre order traversal :–

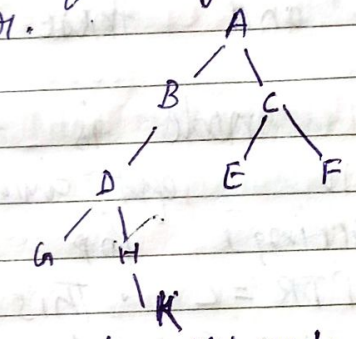
Initially push NULL onto stack and then set PTR = ROOT then repeat the following step until PTR = NULL or while PTR ≠ NULL

Note:- We will insert NULL in stack and that node denotes empty ~~node~~ stack end of traversal

A. Proceed down the left most part rooted at PTR processing each node N on the path and pushing each right child R(N) if any onto stack. The traversing after an all N with no left child L(N) is processed.

B. [Back tracking] POP and assign to PTR the top element of the STACK if PTR≠NULL then return to STACK A otherwise exit. ~~Note: we will in~~

eg- Consider the following tree and find its tree order.



Ans- Step-1 Initially push NULL onto stack then set PTR = A [The root of given tree]

Step-2 Proceed down the left most depth/path at PTR = A as follows.

(i) Process A and push its right child on to stack.

(ii) Process B (there is no right child of it)

(iii) Process D and push it right child on to stack.

(iv) Process G there is no right child of it.

Step-3 [Back tracking] pop the top element H from the stack and set PTR = H this leaves stack: ∅, C. Since PTR ≠ NULL return the stat step 4 A of the algorithm.

Step-4 Proceed down the left most path rooted at PTR = H as follows:-

(i) Process H and push its right child K onto stack

(ii) No other node is processed since H has no left child.

Step-5 [Back tracking] pop the top element K from the stack and set PTR = K. This lives stack : ϕ, C. Since PTR ≠ NULL return the step A of algorithm.

Step-6 Proceed down the left most part rooted at PTR = K as follows :

(i) Process K and there is no right child of it.

(ii) No other node is processed since K has no left child.

Step-7 [Back tracking] Pop C from the stack and set PTR = C. This lives stack : ϕ. Since PTR ≠ NULL & return to step A (algo).

Step-8 Proceed down the left most part rooted at PTR = C as follows.

(i) Process C and push its right child F onto stack. stack : ϕ, F.

(ii) Process E and there is no right child of it.

Step-9 [Back tracking] pop the element F from the stack & set PTR = F. This lives stack : ϕ. Since PTR ≠ NULL return to step A (algo.)

Step-10 Proceed down the left most path rooted at PTR = F as follows :-

(i) Process F & there is no right child of it.

(ii) No other node is processed since F has no left child.

Step-11 [Back tracking] Pop the top element of from the stack & set PTR = NULL. Since PTR = NULL the algo. is completed. ABDGHKCEF

(a) Pre order

Algo - 6 PREORDER (INFO, LEFT, RIGHT, ROOT)
A binary tree T is in memory this algorithm does a preorder traversal of T, appling an operator process to each of its nodes and array stack is used to ~~temporay~~ hold the address of nodes temporarily

1. [Initially push NULL onto stack and initialise PTR]
   Set TOP = 1 , STACK[1] = NULL , PTR = ROOT
2. Repeat step 3 to 5 while PTR ≠ NULL
3. Apply Process to INFO[PTR]
4. [Right child ?] If RIGHT[PTR] ≠ NULL then
   Set TOP = TOP + 1
   STACK[TOP] = RIGHT[PTR]
   [end of if structure]
5. [Left child ?] If LEFT[PTR] ≠ NULL then
   PTR = LEFT[PTR]
   else [Pop element from stack]
   PTR = STACK[TOP] and
   set TOP = TOP - 1
   [end of if structure]
   [end of step 2 loop]
6. Exit

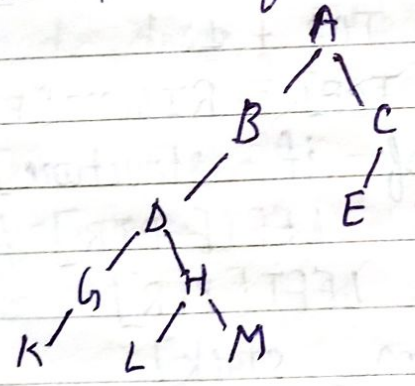
(ii) In order traversal :—

Algo - Initially push null onto stack then set PTR = ROOT then repeat the following step until NULL is poped from the stack.

(A) Proceed down the left most path rooted at PTR, pushing each node N onto stack and stop when a node N with no left child is pushed onto stack.

(B) [Back tracking] Pop and process the node on the stack if NULL is poped then exit. If a node N with a right child R(N) is proceeded, set PTR = R(N) and return to step A.

Consider the following binary tree & find its INORDER



Ans -
1. Initially push NULL on to STACK and set PTR = A
2. Proceed down the left most path rooted at PTR = A, Pushing the node A, B, D, G, K onto stack. stack: ∅, A, B, D, G, K
3. [Back tracking] ~~Process~~ The nodes K, G & D are poped and processed leaving the stack.
   stack : ∅, A, B

[We stop the processing at D since D has right child]

then set PTR = H , The right child of D

4. Proceed down the left most path rooted at PTR = H, pushing the node H, L onto stack. stack ∅, A, B, H, L

5. [Back tracking] The node L, H are poped & processed leaving the stack. so stack : ∅, A, B
[we stop the processing at H since H has right child]
then set PTR = M , the right child of H

6. Proceed down the left most path rooted at PTR = M pushing the node M onto stack: ∅, A, B, M.

7. [Back tracking] The nodes M, B, A are poped and processed . Leaving the stack ∅.
[No other element is poped since A ~~does not~~ have a righ child so PTR = C]

8. proceed down the left most path rooted at PTR = C pushing the nodes C & E onto stack.
so now stack is ∅, C, E.

9. [Back tracking] the node E is poped and proceed since E has no right child. node C is poped & proceed since C has no right child. Next element is NULL so poped the NULL

Now the algorithm finished it work & inorder of tree is

K, G, D, L, H, M, B, A, E, C

Inorder Algo —

Inorder (INFO, LEFT, RIGHT, ROOT)

A binary tree is in memory . This algorithm ~~the~~ does an inorder traversal of T by appling an

operation process to each of its nodes - an array stack is used to temporarily hold the addresses of nodes

1. [Push NULL onto stack and initialise PTR]
   Set TOP = 1 , STACK[TOP] = NULL
   PTR = ROOT
2. Repeat while PTR ≠ NULL [pushes left most path onto stack]
   (a) Set TOP = TOP + 1
   and STACK[TOP] = PTR
   (b) Set PTR = LEFT[PTR]
   [end of loop]
3. Set PTR = STACK[TOP] and TOP = TOP - 1
4. [Back tracking]
   Repeat step 5 to 7 while PTR ≠ NULL
5. Appling process to INFO[PTR]
6. [Right child ?] If RIGHT[PTR] ≠ NULL
   then
   (a) Set PTR = RIGHT[PTR]
   (b) go to step 2
   [end of if structure]
7. Set PTR = STACK[TOP]
   & TOP = TOP - 1
   [end of step 4 loop]
8. Exit.

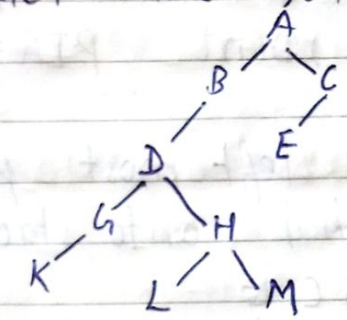
(3) Post order traversal :- Algo - Initially push NULL on to stack then set PTR = ROOT and after that repeat the following step until null is poped

from the stack.

Step-(A) Proceed down the left most path rooted PTR at each node N of the path push N onto stack and if N has right child R(N) then push -R(N) onto stack.

Step(B) [Back tracking] Pop & process positive nodes on stack if NULL is poped then exit. If -ve node is poped i.e. if PTR = -N for some node N set PTR = N and return to step-A.

→ let consider the following tree & find its POSTORDER



Ans- 1. Initially push NULL onto stack & set PTR = ROOT = A. so the stack is NULL.

2. Proceed down the left most path rooted at PTR = A, pushing the nodes A, B, D, G, K onto stack. Furthermore since A has a right child C push -C onto stack after A but before B. Since D has a right child H so -H push onto stack after D but before G.

So stack : A, -C, B, D, -H, G, K

3. [Back tracking] Pop & process K, process G. since -H is a negative, pop -H this leaves stack as A, -C, B, D : Now PTR = -H so reset PTR = H and return to step-A.

4. Proceed down the left most path rooted at PTR=H First push H onto stack since H has a right child M. Push -M onto stack after H

push L onto stack so now the stack is
Ø, A, -C, B, D, H, -M, L

5. [Back tracking] Pop & process L. Now pop -M
Now stack : Ø, A, -C, B, D, H now PTR = -M
so reset PTR = M and return to step-A

6. Process down the left most path rooted at PTR=M
only M is pushed onto stack. so now
stack Ø, A, -C, B, D, H, M

7. [Back tracking] Pop & process M, H, D & B. Now
pop -C this leaves stack : Ø, A.
Now PTR = -C so resent PTR = C & return to
step-A

8. Process down the left most path rooted at PTR=C
C, E are pushed onto stack. so now
stack : Ø, A, C, E

9. [Back tracking] Pop & process E, C, A.
The NULL is poped the stack is empty
and algorithm has finished its work.
So the POSTORDER of the given tree

K, G, L, M, H, D, B, E, C, A

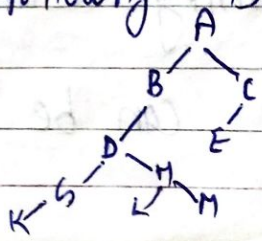
Post order Algo :—

POSTORDER (INFO, LEFT, RIGHT, ROOT)
A binary tree T is in memory This algorithm does a post order traversal T appling an operation process to each of it node. An array stack is used to temporary hold the

address of node.

1. [Push NULL onto stack & initialise pointer]
   Set TOP=1
   STACK[1] = NULL & PTR=ROOT
2. [Push left most path on to stack]
   Repeat step 3 to 5 while PTR ≠ NULL
3. Set TOP = TOP + 1 and
   STACK[TOP] = PTR
4. If RIGHT[PTR] ≠ NULL then
   [Push on stack] Set TOP = TOP + 1
   and STACK[TOP] = - RIGHT[PTR]
   [End of if structure]
5. Set PTR = LEFT[PTR] [end of step 2 loop]
6. Set PTR = STACK[TOP] and
   TOP = TOP-1 [Pop node from the stack]
7. Repeat while PTR > 0
   (a) Apply PROCESS to INFO[PTR]
   (b) set PTR = STACK[TOP] and
   TOP = TOP-1
   [Pops node from STACK]
   [end of loop]
8. If PTR < 0 then
   (a) PTR = -PTR
   (b) Go to step 2
   [End of if structure]
9. Exit

Q. Consider the following BT.



Set TOP=1
Ex- 1. STACK[1] = NULL & PTR = A
2. Repeat 3 to 5 while PTR ≠ NULL
3. Set TOP = TOP+1
STACK[2] = PTR = A
4. If RIGHT[A] ≠ NULL then
set TOP = TOP+1 and
STACK[3] = - RIGHT[A] = - C
[end of if structure]
5. set PTR = LEFT[PTR] = B
6. set PTR = STACK[3]
TOP = TOP-1
7. Repeat while PTR > 0
(a) Apply PROCESS to INFO[PTR]
(b) PTR = STACK[TOP]
TOP = TOP-1 [end of loop]
8. If PTR < 0 then
(a) PTR = -PTR
(b) Go to step 2
9. Exit

Binary Search Tree ⇒ A tree is said to be BST if its left child / sub-tree contains smaller ~~greater~~ value from root and its Right sub-tree or child are greater than root / parent node. This structure enables one to search for and find an element with an average running time $f(n) = \theta(\log_2 n)$.

This structure can be constructed by

(i) Sorted linear array
(ii) linked representation

Special case ⇒ When we have repeatation of number :- In this case, each node N has the following property: The value of N is greater than every value in the left subtree of N and is less than or equal to in the right subtree of N.

* Searching and inserting in BST ⇒

Procedure - FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)
In BST, A binary search tree is in memory and an ITEM of information is given. This procedure finds the location of LOC of item in T and also the location PAR of the parent of item. There are 3 special cases

(i) If LOC = NULL & PAR = NULL, will indicate that tree is empty.
(ii) If LOC ≠ NULL & PAR = NULL, will indicate that the item is the root of T.
(iii) If LOC = NULL & PAR ≠ NULL, will indicate that item is not in T and can be added to T as a child of the node N with location PAR.

Step - 1 [Tree empty ?] If ROOT = NULL then set LOC = NULL & PAR = NULL & return

Step - 2 [item at Root ?] If ITEM = INFO[ROOT] then set LOC = ROOT & PAR = NULL & return

Step-3 [initialises pointer PTR & SAVE]
If ITEM < INFO[ROOT]
then set PTR = LEFT[ROOT] &
SAVE = ROOT

else
set PTR = RIGHT[ROOT] &
SAVE = ROOT
[end of if structure]

Step-4 Repeat step 5 & 6 while PTR ≠ NULL
Step-5 [item count ?]
If ITEM = INFO[PTR] then
set LOC = PTR & PAR = SAVE
and return

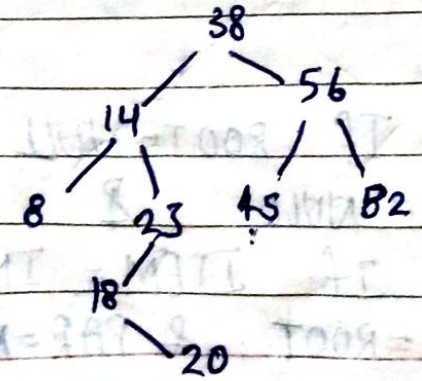
Step-6 If ITEM < INFO[PTR] then
set SAVE = PTR and
PTR = LEFT[PTR]

else
set SAVE = PTR and
PTR = RIGHT[PTR]
[end of if structure]
[end of step 4 loop]

Step-7 [search unsuccessful]
Set LOC = NULL
PAR = SAVE

Step-8 Exit

Q.



Date / /
Page 9

INSERTION IN BST ⇒

Algo —

INSBBT (INFO, LEFT, RIGHT, AVAIL, ITEM, LOC)

In binary tree T is in memory and an item of information is given this algorithm find the location LOC of item in T or adds item as a new node in T at location LOC.

Step-1 Call FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)

Step-2 If LOC = NULL then exit

Step-3 [copy item into NEW node in AVAIL list]

(a) If AVAIL = NULL then write OVERFLOW and exit

(b) Set NEW = AVAIL
AVAIL = LINK[AVAIL]
& INFO[NEW] = ITEM

(c) Set LOC = NEW , LEFT[NEW] = NULL and RIGHT[NEW] = NULL

Step-4 [Add item to tree]

If PAR = NULL then set ROOT = NEW
else if ITEM < INFO[PAR] then
set LEFT[PAR] = NEW
else
set RIGHT[PAR] = NEW
[end of if structure]

Step-5 Exit

DELETION IN BST ⇒

Suppose T is a binary search tree and an item of information is given. There are three case

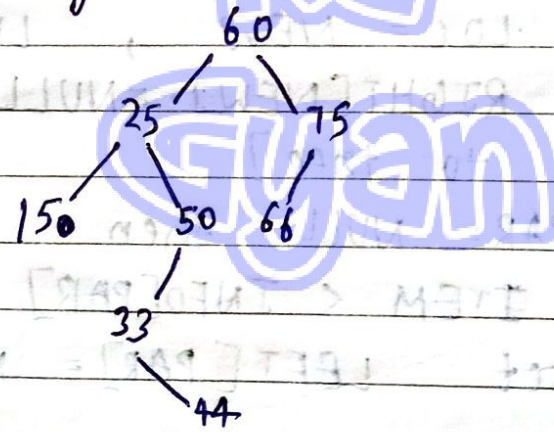
to perform deletion in BST.

Case-1 N has no child then N is deleted from T by simply replacing the location of N in the parent node P[N] by NULL pointer.

Case-2 N has exactly one child then N is deleted from T by simply replacing the location of N in P(N) by the location of only child of N.

Case-3 N has 2 children Let S(N) denote the inorder successer of N. Then N is deleted from T by first deleting S(N) from T (by using case 1 or case 2) and then replacing node N in T by the node S(N)

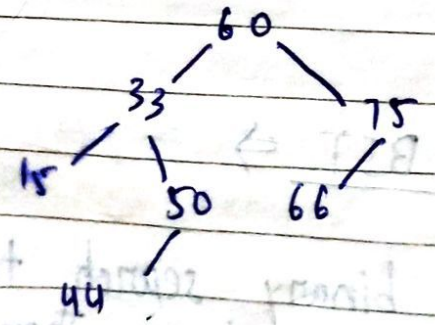
Q. The following ~~list~~ Tree



Now suppose to delete 25 node in given tree.

Ay —

As we observe node 25 has 2 children inorder of 25 = 15, 25, 33, 44, 50



(almost one)

AVL tree ⇒ A height balance tree is 1 in which the difference in the height of 2 sub-trees for any node is less or equal to some specified amount. One type of height balance tree is the AVL tree named after its originators ('Adel-son' velksi' and landis')

Balance factor:- To implement an AVL tree each node must contain a balance factor which indicate its state of balance related to its sub-tree. If a node has its B.F. 0, 1, -1 then that node is in balance condition and if node has its value other than this that node is in unbalance condition.

We can evaluate a B.F. for a particular node by height of left sub-tree − height of Right sub tree.

B.F. = H of left sub-tree − H of right sub-tree

Rotation:-

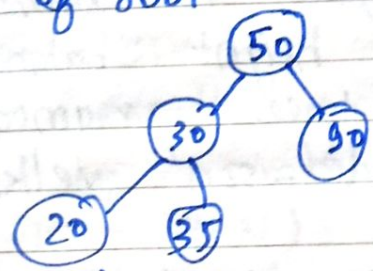
If a tree becomes unbalance then we have to use rotations to make it in balance condition. There are four rotation we have

(i) LL Rotation
(ii) RR Rotation
(iii) LR Rotation → double rotation
(iv) RL rotation

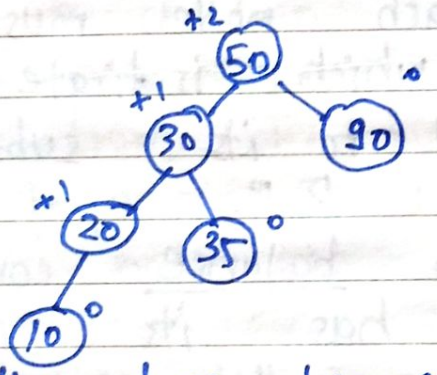
] single rotation

(i) Left-Left Rotation :- In this rotation new node is inserted at the left subtree of the ~~root~~ left subtree of root

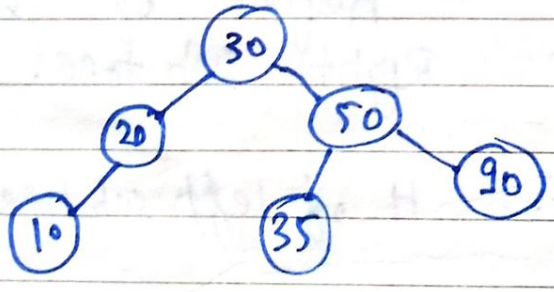
eg-



Now insert 10 above tree

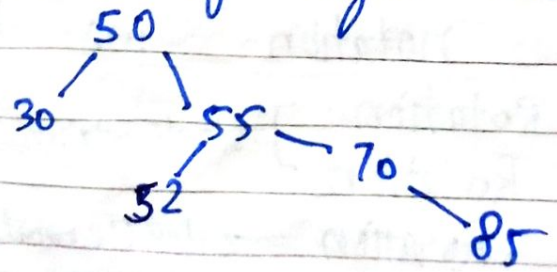


- Now the tree becomes unbalance and by inspection we identified that we have to apply LL Rotation.

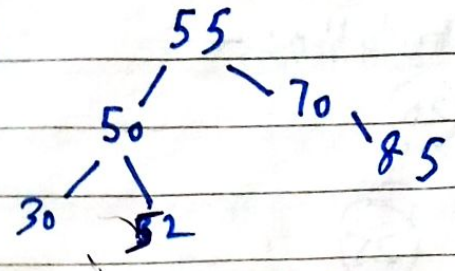


(ii) Right -Right Rotation:- In this rotation, new node is inserted at the right subtree of right subtree of root node.

eg-

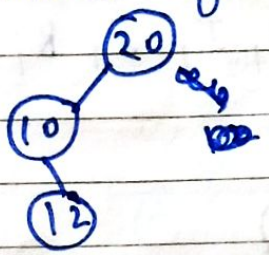


apply RR rotation

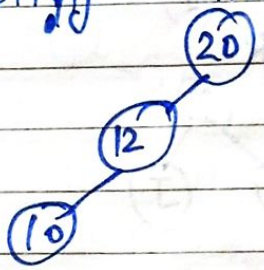


(iii) Left to right rotation :— In this rotation, new node is inserted at the Right subtree of the left sub tree.

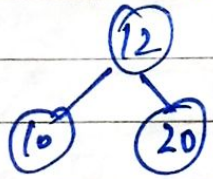
eg-



Now appling RR rotation :—

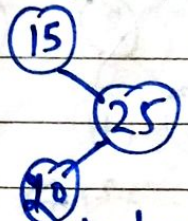


Now appling LL rotation :—



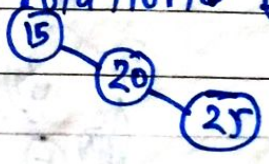
(iv) Right to left rotation :— The new node is inserted at the left subtree of the right sub tree.

eg-



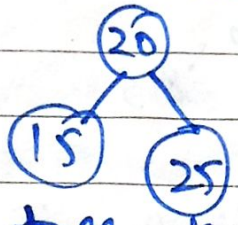
This tree is unbalance then.

Appling ~~RR~~ LL rotation (By the rule of LL forwarded by RR)



Date / /
Page 14

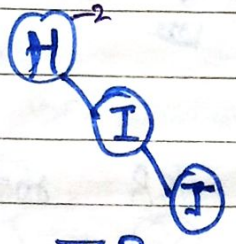
Appling RR rotation :—



Now this tree is balance.

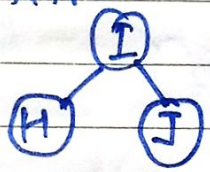
eg- Create an AVL tree from the given set of values. H, I, J, B, A, E, C, F, D, G, K, L

Ans- (1)

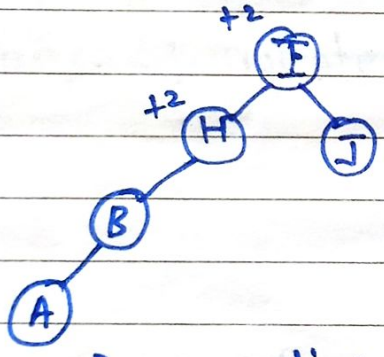


By appling RR

(2)

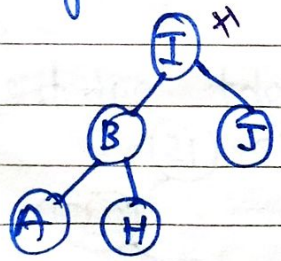


(3)

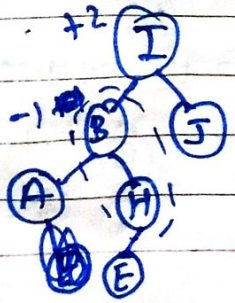


By appling LL

(4)



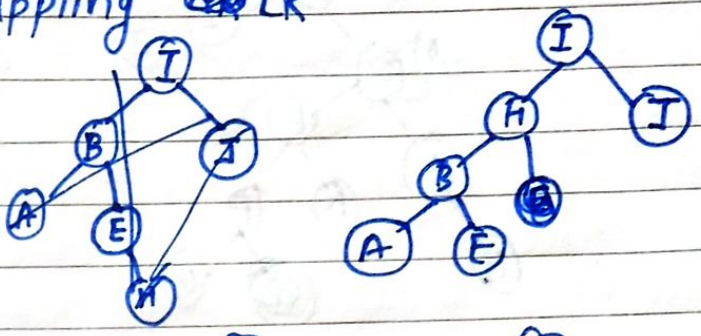
(5)



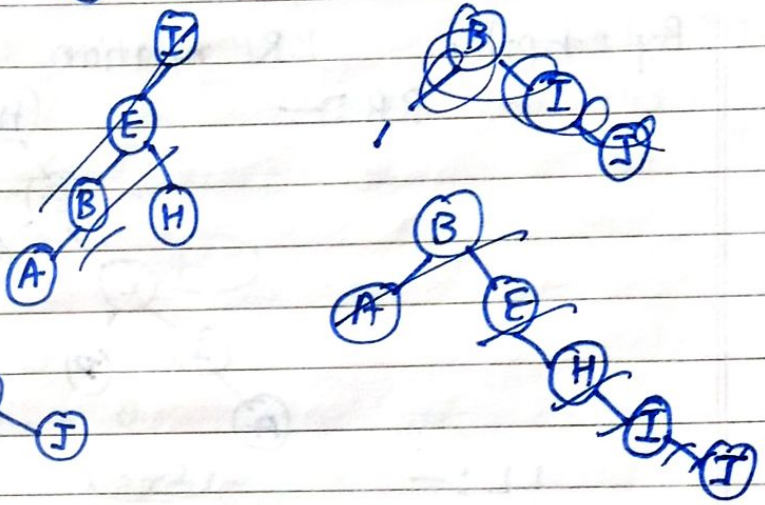
Date / /
Page 15

By appling LR

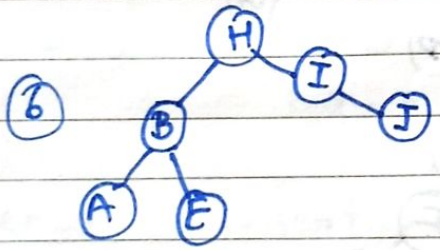
RR rotation:-



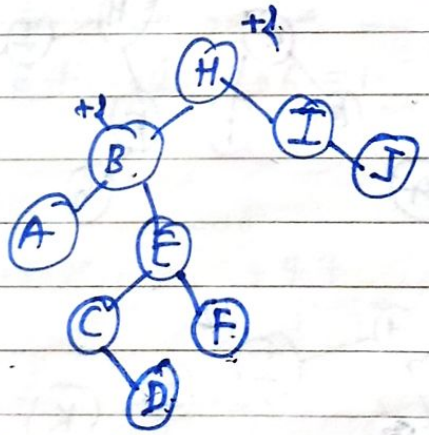
LL rotation:-



(6)

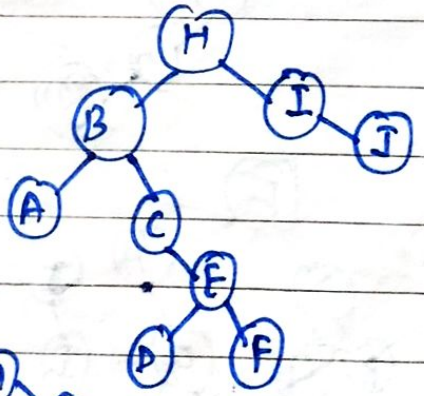


(7)

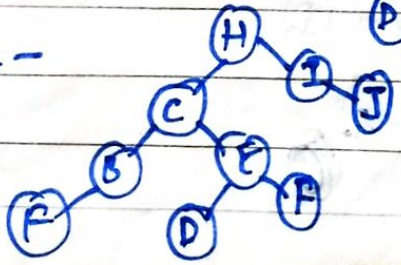


By appling RL

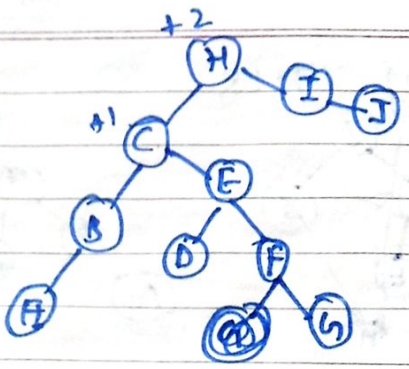
LL rotation:-



RR rotation:-

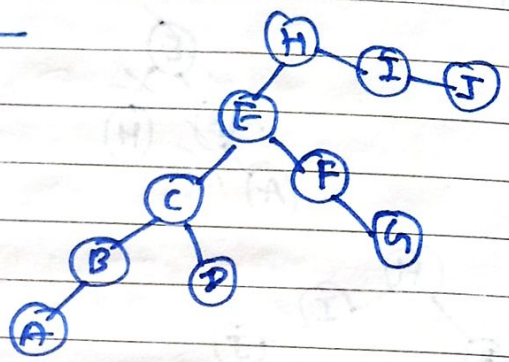


8

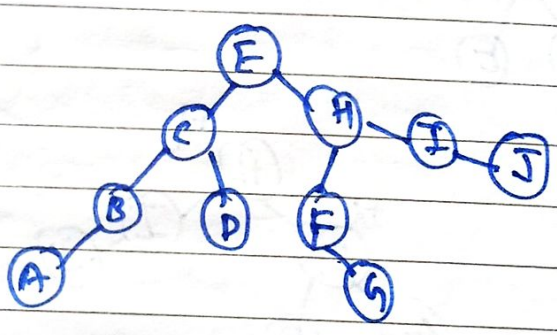


By Apply LR rotation

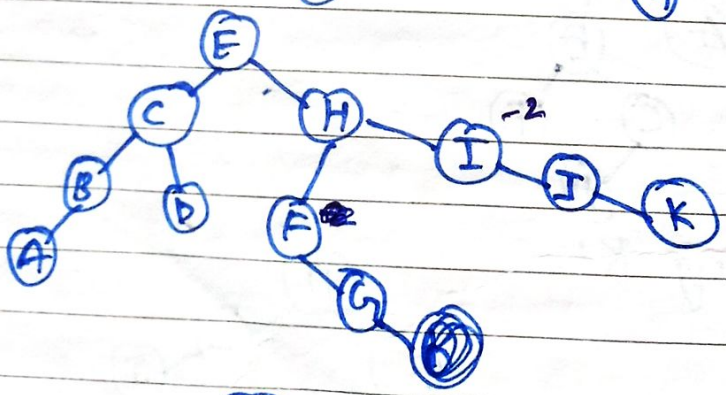
RR :—



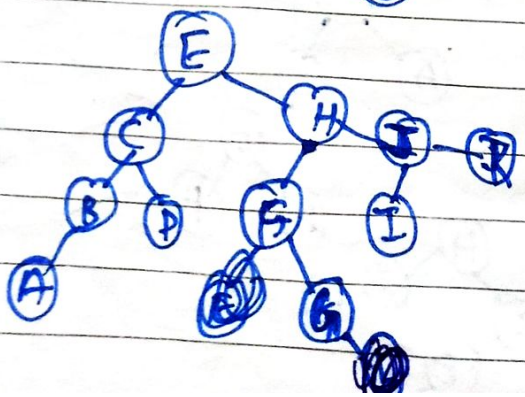
LL :—

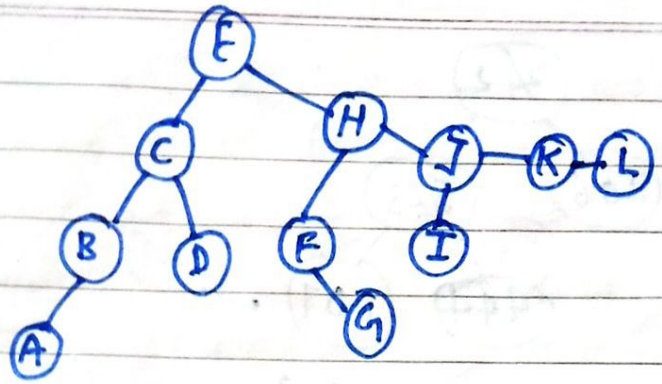


9



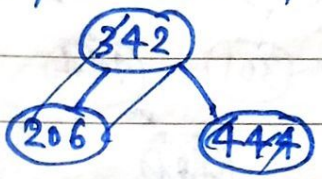
10





Q. construct following 342, 206, 444, 523, 607, 301, 142, 183, 102, 157 and 149. AVL tree

Ay



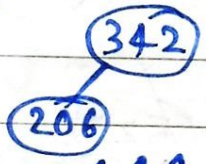
step-1 insert 342

B.F. = 0

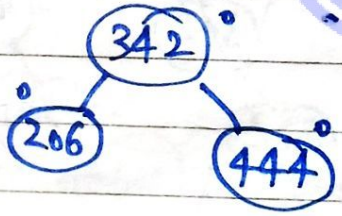
step-2 insert 206

B.F. of 342 = 1

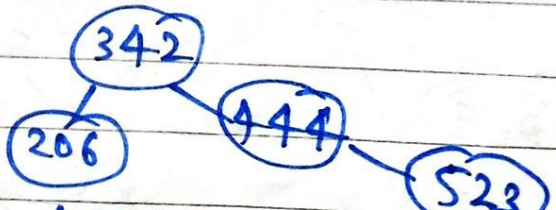
B.F. of 206 = 0



step-3 insert 444

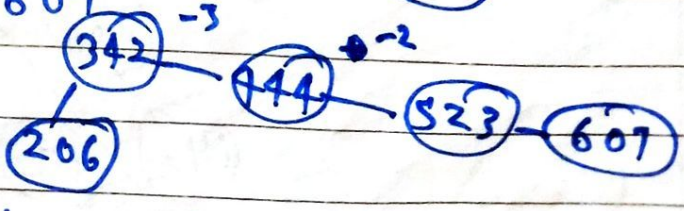


step-4 insert 523



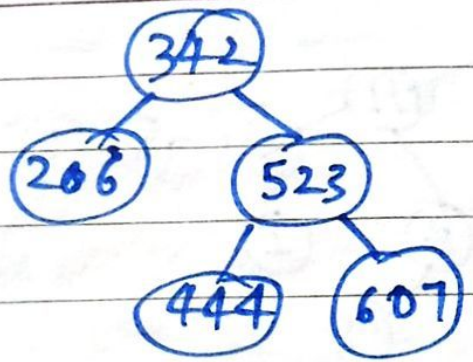
step-5 insert 607

(a)

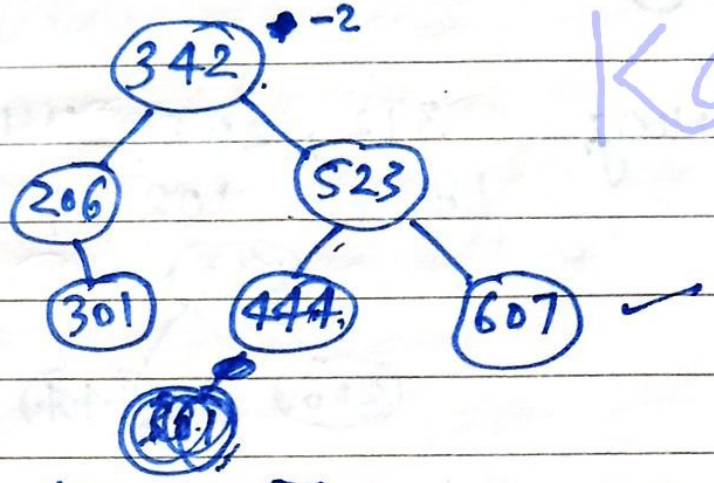


By appling RR

(b)

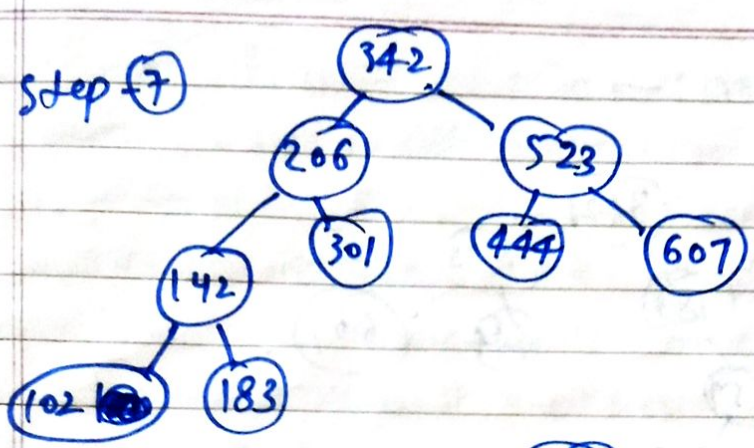


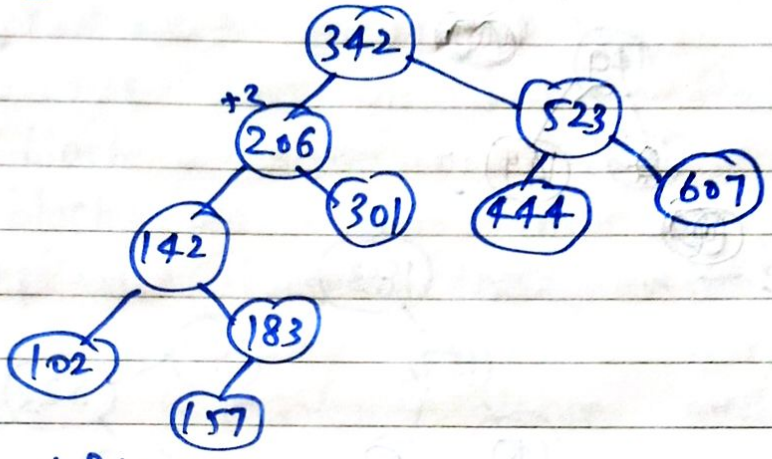
(c)



By appling [redacted] :-

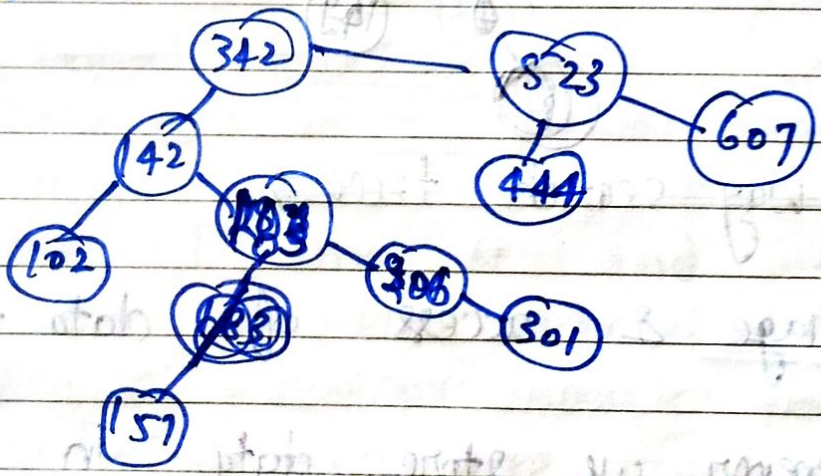
Step-7



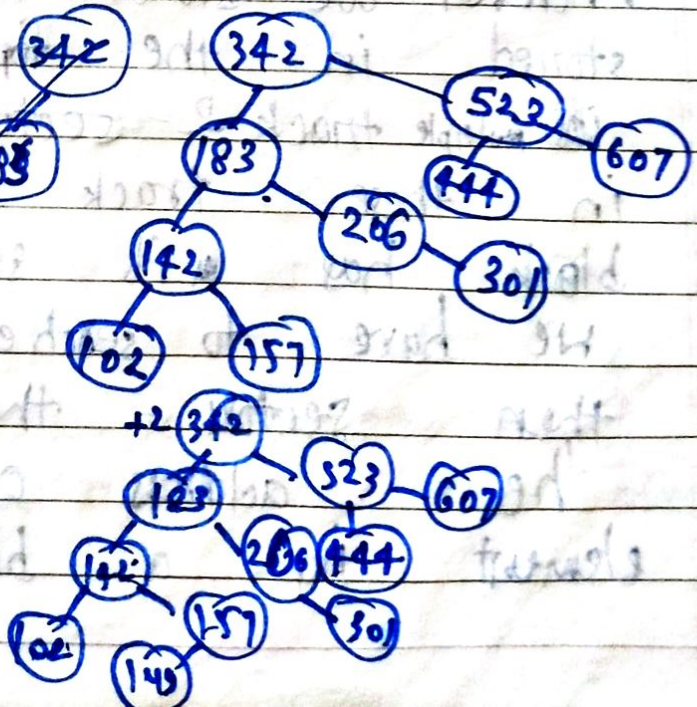


LR:-

RR:-



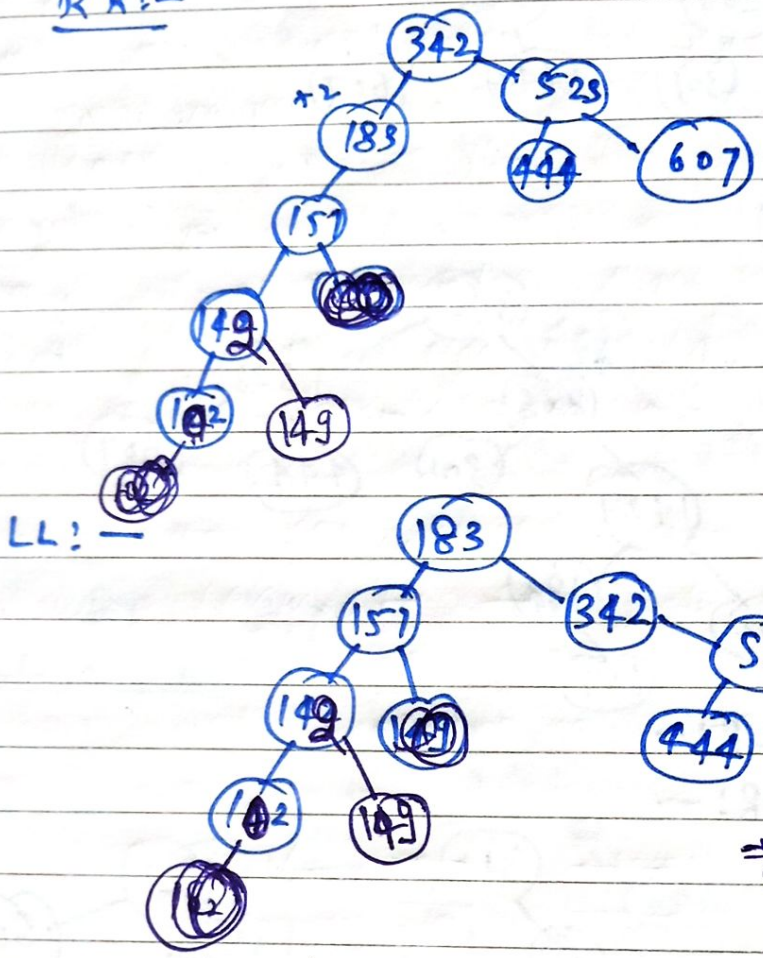
LL:-



Step-8

LR rotation :-

RR :-



~~M-way search tree~~ ⇒

Storage & access of data from Harddisk :-

Whenever we store data in harddisk, it's stored in the form of bits. HD has multiple track & sector and our data stored in these track & form of block. Each block has its specify size. Whenever we have to search element then we require track then sector then offset. Offset is the address of a particular value/ element in a block.

But whenever we are storing large amount of data and after storing the data we want to access each track then sector then block sequencially more time consuming and inefficiency to make such process efficient we use the concept of index.

Index has pointer w.r.t data store in the block. These index are also stored in block so to access the contain of block, we access the index. Sometime we have to use multiple indexing to make the pointer structure dense and to make such operation more efficient.

M-way search tree ⇒

M-way search tree are generalised version of BST. M-way search tree T may be and empty tree or it can be a non empty tree. The goal of M-way search tree is to minimised the access which retriving a key from a file. If tree T is non-empty then it should satisfy following properties :—

(i) For some integer n known as order of tree. Each node of degree which can reach a maximum of n, in other words each node has atmost m child nodes.

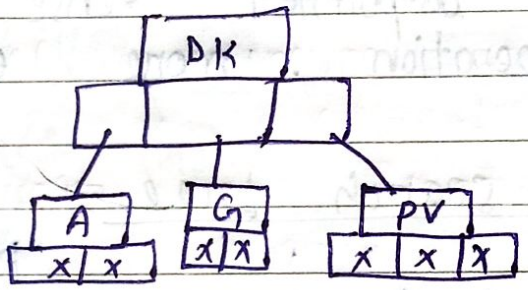
(ii) If a node has k child nodes where $k \leq m$ then the node can have only k-1 keys. For a node A_0, (K_1, A_1), (K_2, A_2) -- (K_{m-1}, A_{m-1})

(iii) All key's values in the subtree pointed to by A_i are less than the key K_{i+1} and all key values in the subtree pointed to by A_{m-1} are greater than K_{m-1}.

(iv) Each of subtree A_i, $0 \leq i \leq m-1$ are all M-way search tree.

Q. Construct a 3-way search tree considering the following elements :- D, K, P, V, A, G

Ans -



B-Tree :- B Tree is an example of M-way search tree. A tree of order m, if non empty is an M-way ~~empty~~ tree in which

(i) The root has at least 2 child nodes and atmost m child nodes (ii) The internal nodes accept the root have atleast $\lceil m/2 \rceil$ and atmost m child nodes.

(iii) The no. of keys in each internal node one less than the number of child nodes and these keys partition the key in the sub tree of the node

B tree can be widely used for disk access. A B-tree of order m can have at most m-1 keys and m children. One of the main reason of using B tree is its capability to store no. of keys in a single node & large key values by keeping height of tree relatively small. in an manner similar to M-way search tree.

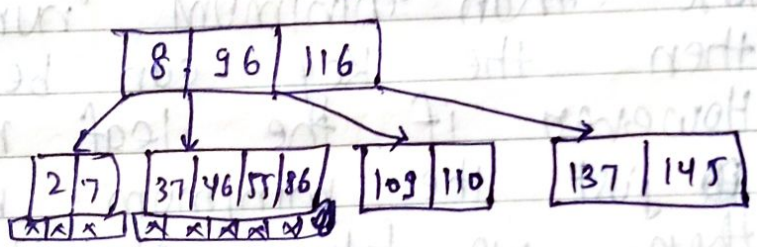
(iv) All leaf nodes are on the same level.

A B-tree of order 3 is refered to as 2-3 since the internal nodes are of degree 2 & 3 only.

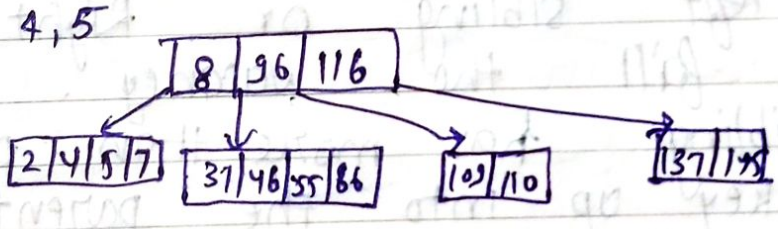
Insertion in a B-Tree :—

Consider the following B-Tree of order-5

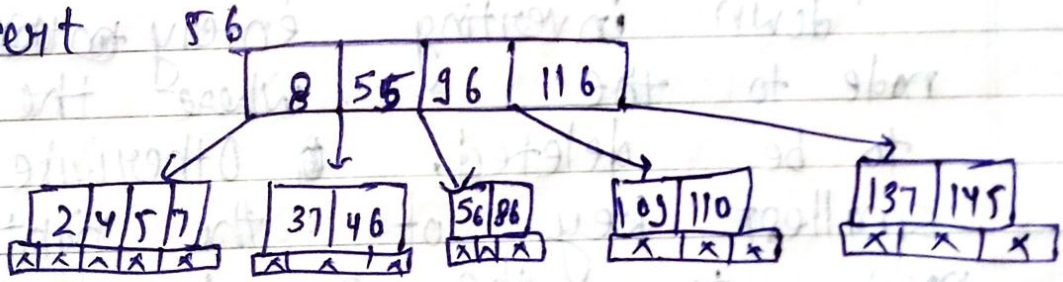
Q.



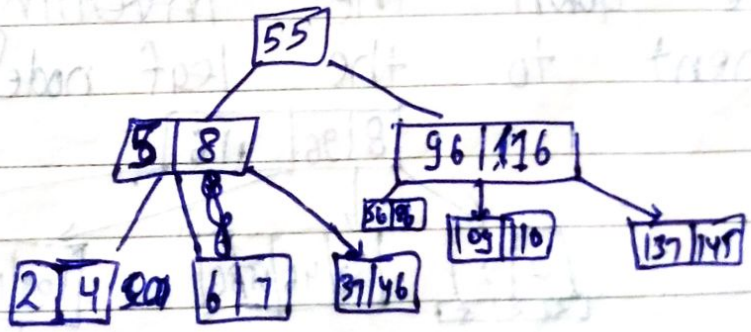
insert 4, 5



Insert 56



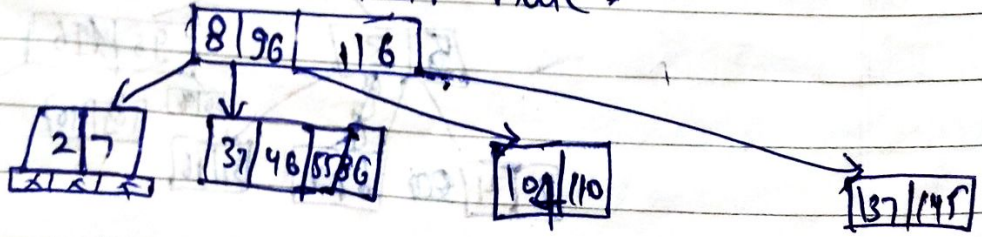
Insert 6



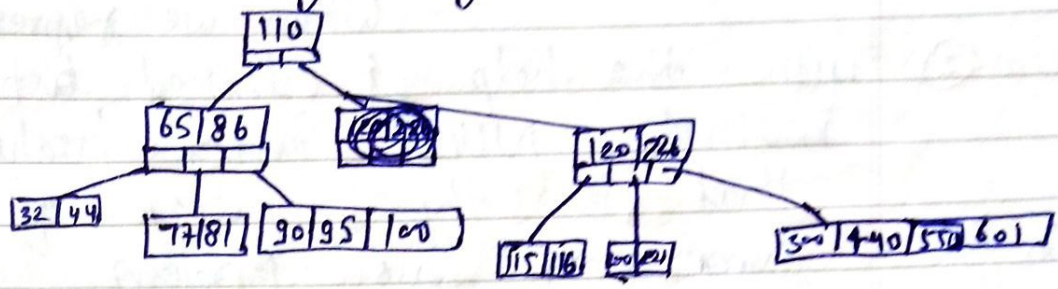
Deletion in B-Tree ⟹

while deleting a tree, it is desirable that the keys in leaf node are removed. However when a case arises for the keys that is in an internal node to be deleted then we promote a successor or producer of a key to be deleted to occupy the position of the deleted key.
While removing the key, if the node contains more than minimum numbers of elements then the key can be easily removed. However if the leaf node contain just the minimum no. of element then we take element either from left sibling or Right sibling to fill the vacancy. If the left sibling has more than - pull the largest key up into the parent node & move down inverting entery parent node to the where the key is to be deleted. Otherwise pull the smallest key of the right sibling node to the parent node and move down the inverting parent element to the leaf node.

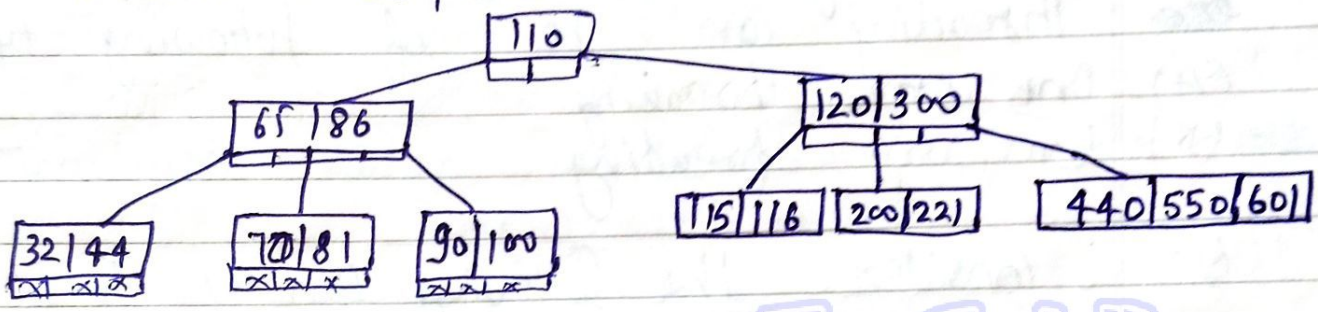
Q.



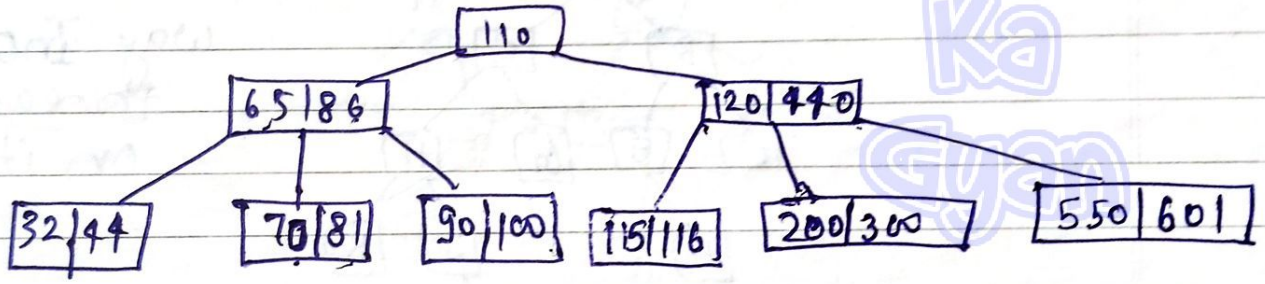
Q. Delete the following key 95, 226, 221 ← order - 5



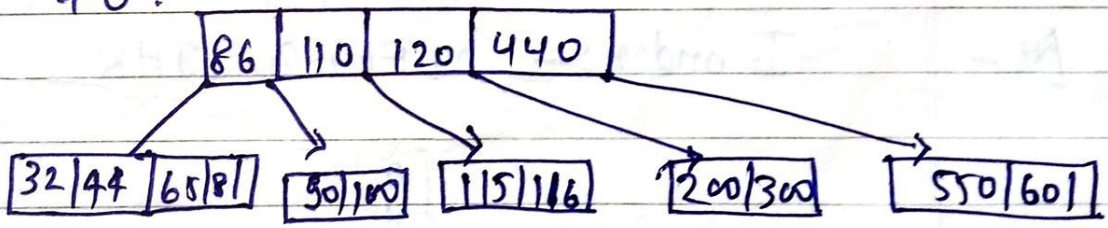
Ans - Delete 95, 226 :—



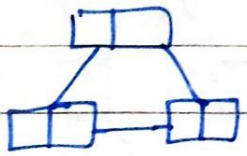
Delete 221 :—



Delete 70 :—



B+ Tree ⇒



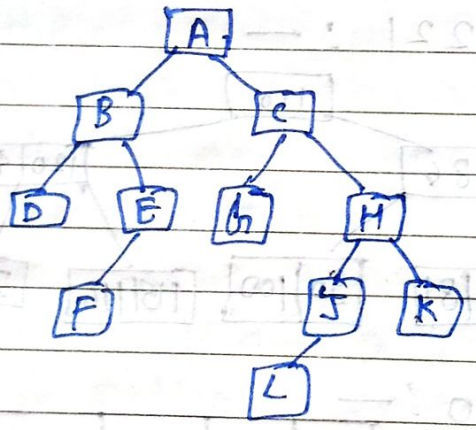
Threading :—

When we represent a binary tree with the help of linked list there will be multiple NULL entries / values stored in this. that is the wastage of memory. To come over this problem the concept of Threading has been introduced.

Threading can be of following type :—

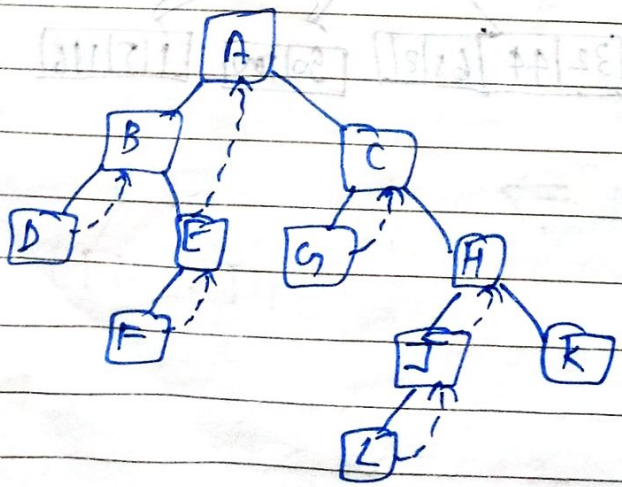
(A) One way threading
(B) two way threading

Q. Consider the following binary tree :—



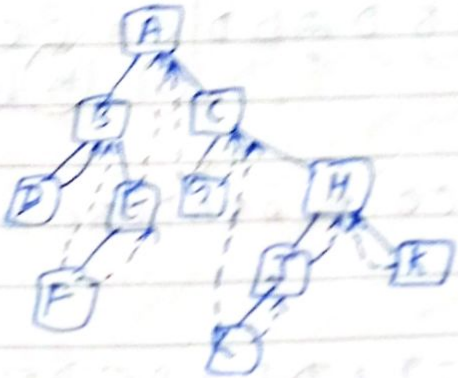
perform one way Inorder threading on it.

Ans - Inorder :— D B F E A G C L J H K

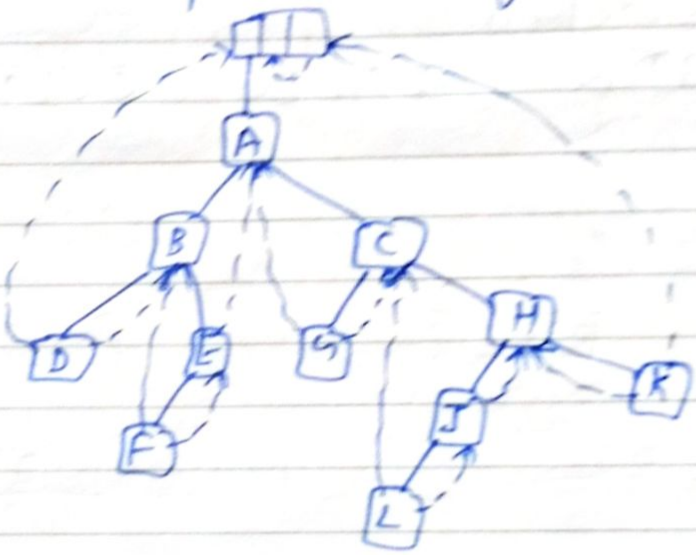


Q. Construct 2 way Inorder threading for the same tree.

Ans -



Q. Construct 2 way threading with header node



Heap is a tree based data structure in which all the nodes of the trees are in a specific order and tree is a complete binary tree.

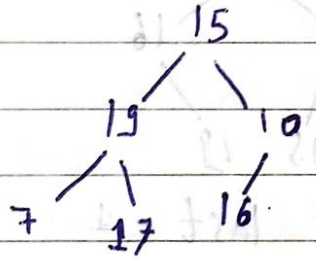
Heap Sort ⇒ Set of elements → tree → Heap

Heap → max / min → process to apply this operation is known as **heapify**

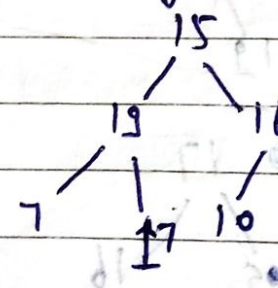
Tree [Parent value > children → max → Heap
Parent value < children → min → Heap

Q. Given an array of 6 element:—
15, 19, 10, 7, 17, 16 · sort it in ascending order using heap sort.

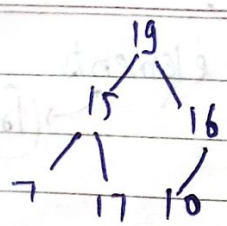
Ans- (1.) First we create heap — max heap



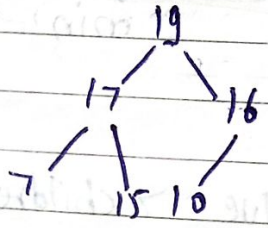
(2.) start processing from the right most node. i.e. 16 which is greater than its parent i.e. 10 so interchange the value



(3.) Now 19 which is greater than parent i.e. 15 so interchange the value.



(4.)

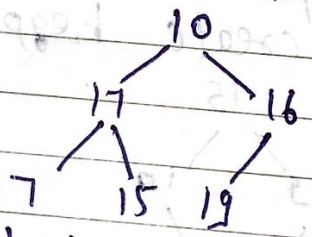


interchanged 15 & 17.

Part-B Sorting

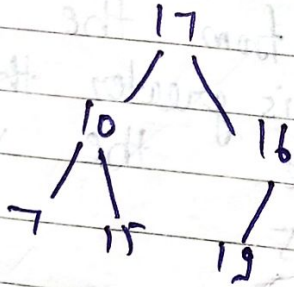
(i) Right most element 10 & interchange it with maximum element 19 of tree. and decrease last.

(a)

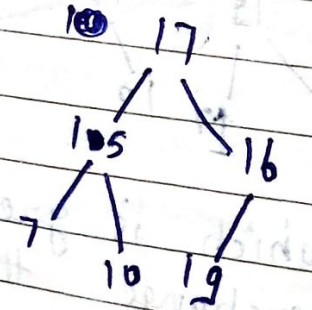


last = last - 1

(b)



(c)

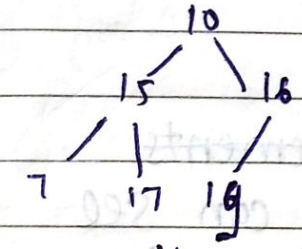


10<15 so interchanged

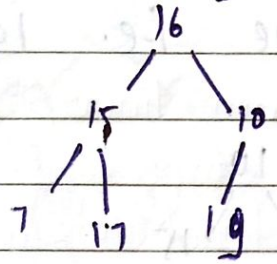
(2.) Right most element 10 & interchange it with max. element 17 of tree and decrease last.

last = last - 1

(a)



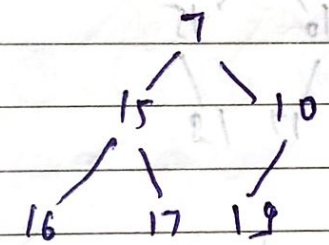
(b)



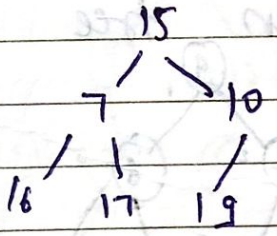
(3.) Right most element 7 & interchange it with max. element 16 of tree & decrease last.

last = last - 1

(a)



(b)

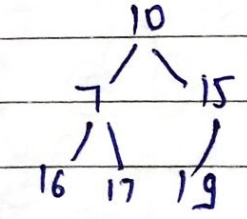


interchanged 7 & 15

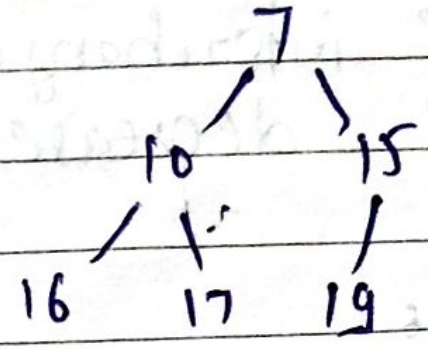
(4.) Right most element 10 & max. 15 interchange

last = last - 1

(a)



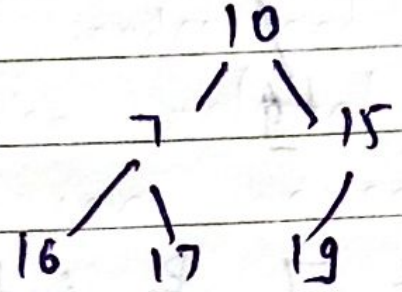
(a)



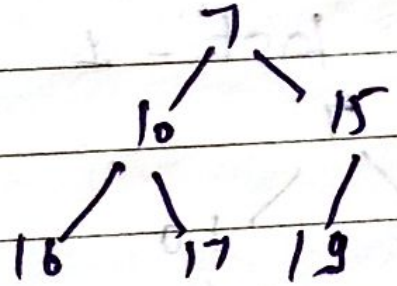
~~Now heap elements~~

(5.) Now we can see first node contain maximum value i.e. 10 so exchanged with 7

(a)



(b)



interchanged 7 & 10

GRAPH

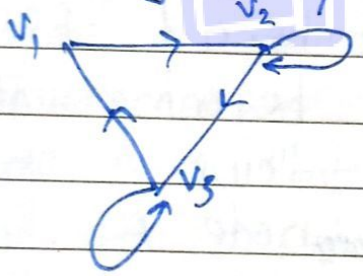
Type of Graph:—
(i) Simple (ii) Multi (iii) directed (iv) Undirected
(v) Weighted (vi) Unweighted

Representation of Graph:—

There are two ways to represent a graph.

(i) Adjacency matrix or Adjacency list
(ii) linked representation

(i) Adjacency:—

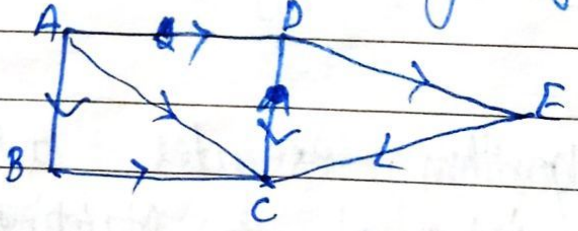


	V_1	V_2	V_3
V_1	0	1	0
V_2	0	1	1
V_3	1	0	1

(ii) linked representation:— There are two type of list, we will use in this representation

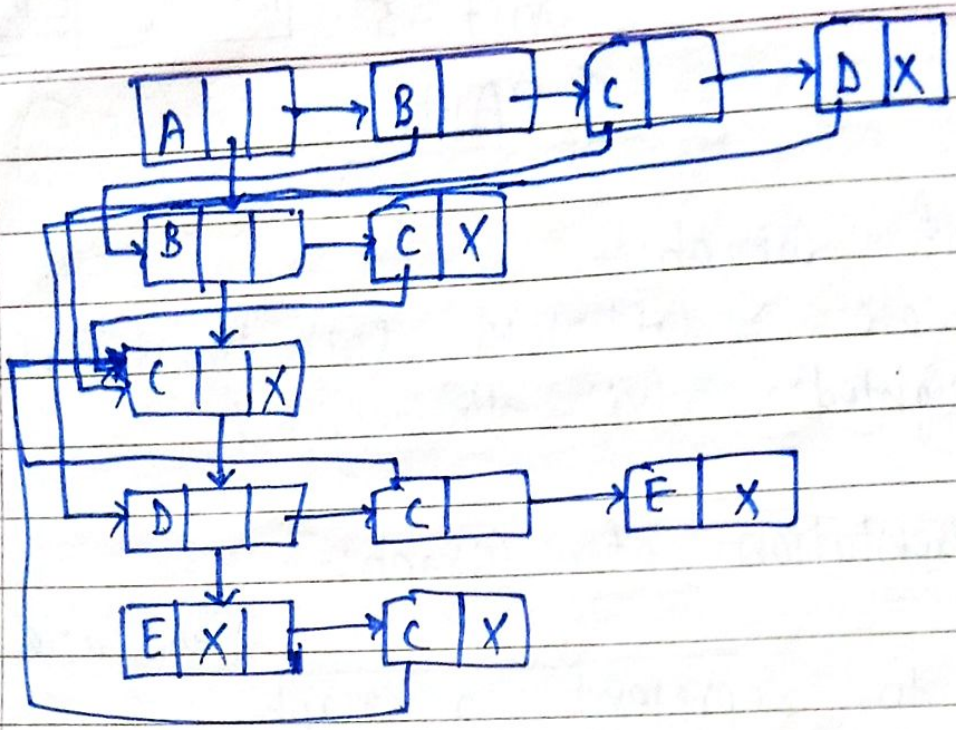
(A) Node list
(B) EDGE List

Consider a following graph:—



Ans-

Node	Adj. node
A	B C D
B	C
C	—
D	C, E
E	C



Operation of Graph:-

(i) Traversing :-

BFS ⇒ The general idea behind BFS (Breadth first search) search begining at the starting node A as follows

(a) First we examin the starting node A. Then we examin all the neighbours of A. Then we examin all the neighbours of neighbours of A & so on.

Queue

Means we need to track of the neighbours of the node A & we need to garantee that no node is presented more than one. This is done with the help of queue and by using the 9 field status.

Algo:-

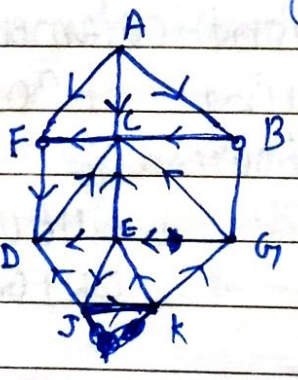
This algorithm executes a BFS on a graph G beginning at a starting node A

1. [Initialise] all the node to ready state [Status = 1]

2. Put the starting node A in queue and change its status to waiting state {status = 2}
3. Repeat step 4 & 5 until queue is empty
4. Remove the front node N of queue, process N and change the status of N to the processed state {status = 3}
5. Add to the rear of queue all the neighbour of N that are in the ~~study~~ ready state {status = 1} and change their states to the waiting state {states 2}
(end of step 3)
6. Exit

Note → This algorithm will process only those nodes which are reachable from the starting node A. If any one wants to examine ~~of~~ all the node in the graph then this algorithm much modify so the its begins with another node.

Eg- Consider the following graph:—



Suppose the graph G represents the trains B/w different cities and we want more from city A to city J with minimum no. of stops in other words we want minimum path P from A to J

where each edge length 1.
The minimum path P can be found by using BFS beginning at city A & ending at J is encountered. During the execution of search we will also keep track of the origin of each edge by using an array ORIG together with the array QUEUE. The steps of BFS searches are as

Ans -

Step-1 Initially add A to the queue and NULL to the origin ORIG as follow

QUEUE = A , FRONT = 1
ORIG = ∅ , REAR = 1

Step-2 Remove the FRONT element A from the queue by setting the front = front + 1 and add the neighbour of A ~~Queue~~ in queue and

FRONT = 2 , QUEUE = A F C B
REAR = 4 ORIG = ∅ A A A

Step-3 Remove the front element F from the queue by setting FRONT ++ and add the neighbour of F in queue

FRONT = 3 , QUEUE = A F C B D
REAR = 5 ORIG = ∅ A A A F

Step-4 Remove the front element C & add neighbour of C in queue

FRONT = 4 QUEUE = A F C B D
REAR = 5 ORIG = ∅ A A A F

(Because we didn't add element already elem)

Step-5 Remove element B & add neighbour of B in queue

FRONT = 5 QUEUE = A F C B D G
REAR = 6 ORIG = ∅ A A A F B

Step-6 Remove element D and add neighbour of D in queue

FRONT = 6 QUEUE = A F C B D G
REAR = 6 ORIG = ∅ A A A F B

Step-7 Remove element G and add neighbour of G in queue

FRONT = 7 QUEUE = A F C B D G E
REAR = 7 ORIG = ∅ A A A F B G

Step-8 Remove element E and add neighbour of E in queue

FRONT = 8 QUEUE = A F C B D G E J
REAR = 8 ORIG = ∅ A A A F B G E

As we encountered J which is a final destination so we stop here. We now back track from J using the array ORIG to find the path P.

J ← E ← G ← B ← A

DFS (Depth first search):—

This search works as follow it begins with a starting node then we examin each node N along a path P which begins at A, that is we process a neighbour of A & then neighbour of neighbour of A and so on. After coming to a "deid end" i.e. to a end of path P, we backtrack on P until we can continuous along another path P' and so on. This algorithm is very similar to BFS but now we use stack instead of QUEUE.

Algorithm:— This algorithm executes a DFS on a graph beginning at starting node A

tep-1 Initialise all node to ready state {status = 1}

ep-2 Push the starting node onto stack and change its state to the waiting state {status = 2}

ep-3 Repeat step 4&5 until stack is empty

tep-4 Pop the top node N of stack process N and change its status to the processed state {status = 3}

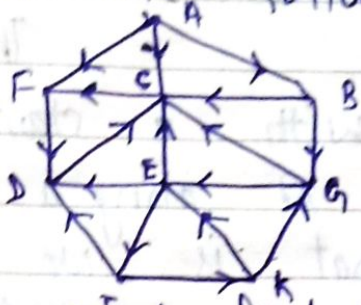
tep-5 Push onto STACK all the neighbours of N that are still in the ready state. {status = 1} and change their status to the waiting state {status = 2}

(End of step-3 loop)

tep-6 Exit

Teacher's Signature.....

Q. Let consider the following given graph :—



Suppose we want to find & print all the node reachable from the node J. The steps of our search as follow -

Ans - (a) initially push J onto the stack as follows:

STACK = J

(b) Pop & print top element J & then push onto stack all the neighbour of J as follow:

PRINT = J STACK = D K

(c) Pop & print top element K & then push onto stack all the neighbours of K as follows:

PRINT = K STACK = D E G

(d) Pop & print top element G & then push onto stack all neighbours of G as follow:

PRINT = G STACK = D E C

(e) Pop & print top element C & then push onto stack all neighbours of C as follow:

PRINT = C STACK = D E F

(f) Pop & print top element F & then push onto stack all neighbour of F as follow:

PRINT = F STACK = D E

Teacher's Signature

(g) Pop & print top element E & push onto stack ~~onto stack~~ all neighbours of E as follow:
PRINT = E STACK = D

(h) Pop & print top element D & push onto stack all neighbours of D as follow:
PRINT = D STACK = NULL
The stack is now empty, so the DFS of Graph G stating at J is now complete

J, K, G, C, F, E, D those are reachable from J

* Minimum Spanning Tree ⇒ A spanning tree of a graph is minimizes length of edges of tree.
just a subgraph that contains all the vertexes and it is a tree. A graph may have many spanning tree we can also assign a weight to each edge which ~~edo~~ is no. representing how unfavourable it is and use this to assign a weight to spanning tree by computing the some of weight of the edges in the spanning tree.
A minimum spanning tree or minimum weight spanning tree is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally any undirected graph has a minimum spanning forest.

Application of Minimum spanning tree ⇒

(i) Application of MST would be finding the best airline routes.

(ii) one practical application would be design of a network.

KRUSKAL'S Algorithm : ⇒

kruskal algorithm is an algorithm that finds a MST for a connected weighted graph. It finds a safe edge to add to the growing forest by finding of all the edges that connect any two trees in the forest, an edge (u, v) of least weight. This means it finds a subset of the edges that forms the tree that includes any vertexes, where the total weight of the edge in the tree is minimised.

If the graph is not connected then it finds a minimum spanning forest (a minimum spanning tree for each connected component)

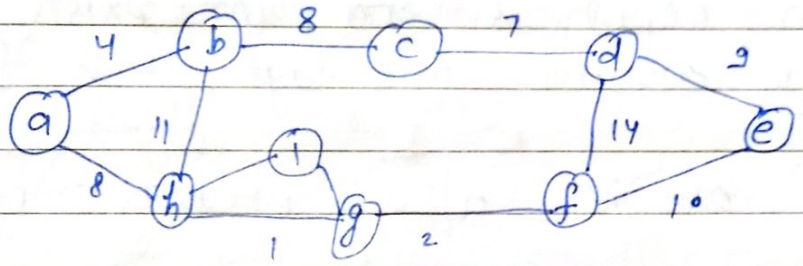
Algo :—

```
MST - kruskal (G, w)
1. A ← ∅
2. For each vertex v ∈ V[G]
3. do MAKE_SET(v)
4. Short the edges E into non decreasing order by weight (w)
```

5. For each edge $(u, v) \in E$, taken in non decreasing order by weight
6. do if FIND-SET(u) ≠ FIND-SET(v)
7. then $A \leftarrow A \cup \{(u, v)\}$
8. UNION (u, v)
9. Return A

Complexity:— kruskal algorithm can be shown to run in $O(E \log E)$ or equivalenty $O(E \log V)$

Q.

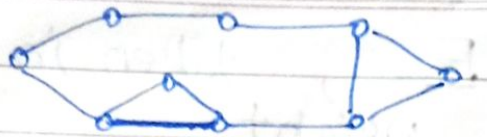


A-

Sort the edge order by that is

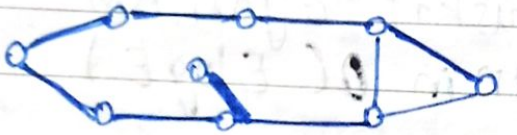
(h, g) = 1	(b, c) = 8
(g, f) = 2	(a, h) = 8
(i, g) = 6	(d, e) = 9
(h, i) = 7	(e, f) = 10
(c, d) = 7	(b, h) = 11
	(d, f) = 14

Now check for each edge (u, v) where the end points u & v belongs to the same tree. If they do then the edge (u, v) can't be added. Otherwise the two vertexes belong to the different tree at the edge is added to A. and the vertexes in the two ~~trees~~ are merged in by union -(u, v)
So first take (h, g)



Now take (g, f), (a, b), (i, g)

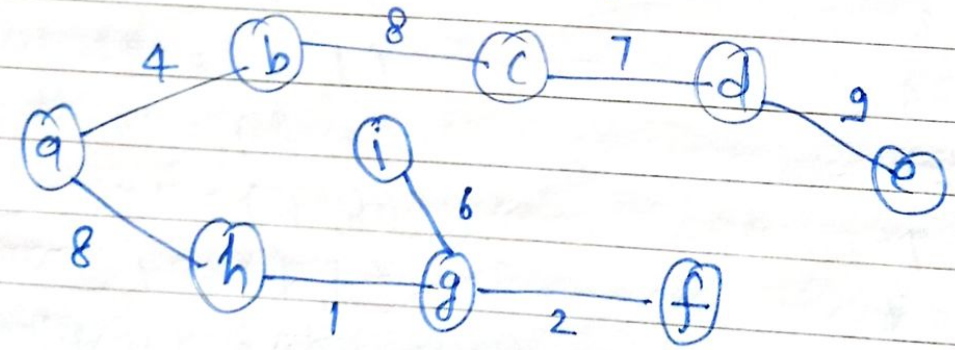
Now edge (h, i) vertex in same set thus it creates cycle so this edge is discarded.



Now take (c, d), (b, c), (a, h), (d, e)

Now in (e, f) is end point exists in same tree so discard this edge then discard (b, h) as it also creates cycle.

So finally the graph has been changed into a Minimum spanning tree. by Kruskal's algorithm.



PRIM's Algorithm ⇒ Prim's algorithm is also a greedy algorithm. It starts with a empty spanning tree the idea is to maintain two set of vertexes. The first set contains the vertexes already included in the MST, other set contains vertexes not yet included. At every set is considered all the edges that connect the two sets and pick the minimum weight from this edge. After picking the edge is move other end point of the edge to the set containing MST. A group of edges that connect two set of vertexes in a graph is called cut in graph theory. So at every step of prim's algorithm we find a cut, pick the minimum weight edge from the cut and include to vertexes to MST set.

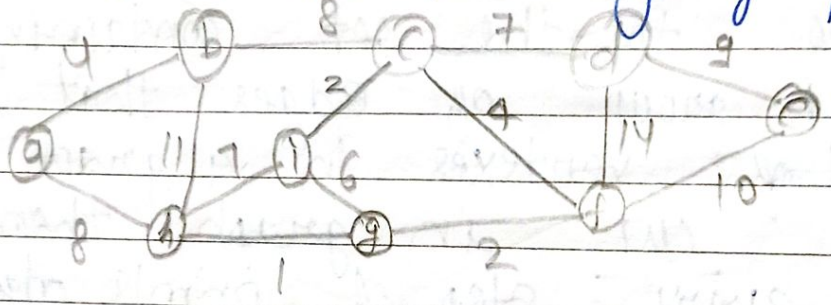
Algo

Step-1 Create a set mstset that keeps track of vertexes already included in MST.

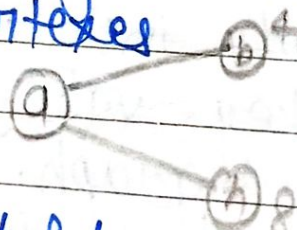
2. Assign a key value to all vertexes in the input graph. Initialises all key values as infinite. Assign key value 0 for the first vertex. so that it is picked first.

3. While Mstset doesn't include all vertexes
(a) Pick a vertex u which is not there in mstset & has minimum key value
(b) Include u to mstset.
(c) Update key value of all adjacent vertexes
To update the key value iterate through all adjacent vertexes. For every adjacent v, if weight of edge u-v is less than the previous key value of v, update the key value as weight of u-v

Q. Consider the following graph:—

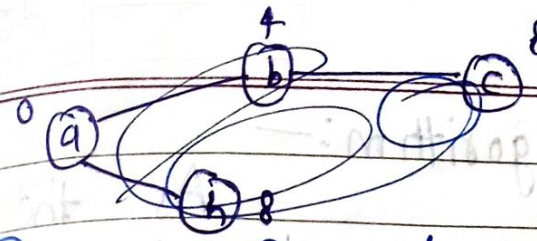


Ans—
→ Initially The mstset is empty & key is assigned to vertexes are [0, INF, INF ---] where INF = infinite.
→ Now pick the vertex with minimum key value
→ The vertex a is picked, include it in mstset. Now mstset becomes {a}
→ After including mstset update key value of adjacent vertexes

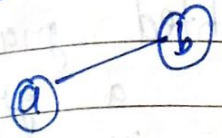


The key value of b & h are updated as 4 & 8 respectively. Pick the vertex key value i.e. already not in mstset. So vertex b is picked & added to mstset. Now mstset became {a, b}

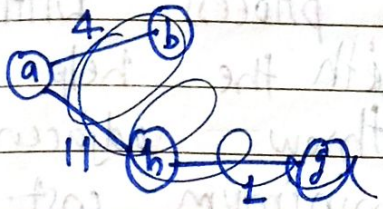
Teacher's Signature.....

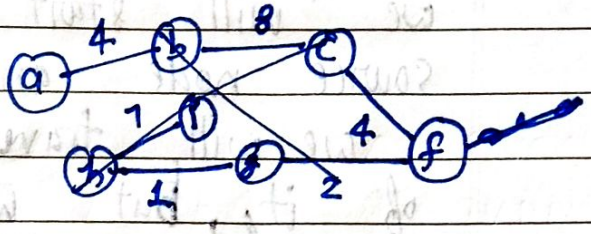


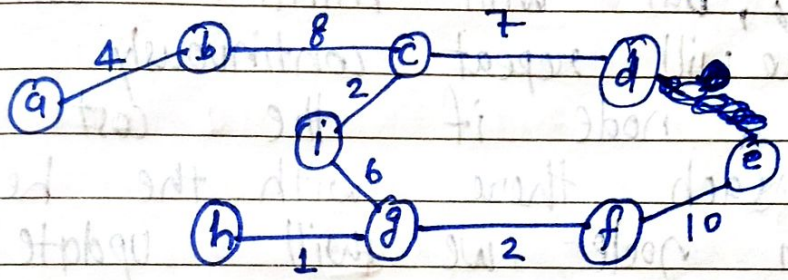
(a) $S = \{a\}$ $V|S = \{b, c, d, e, f, g, h, i\}$



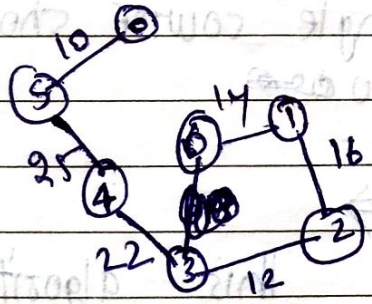
$S = \{a, b\}$, $V|S = \{c, d, e, f, g, h, i\}$







$1+2+2+4+8+6+7+10 = 40$



Shortest path algorithm :—

As the name implies we have to find shortest path in a given directed weighted graph. To find a shortest path in a graph we will start to process with a source node and with the help of it we will traverse throw adjacency vertexes of it, But with minimum cost. This process we will repeat continuously and for a node if the cost comes low to reach there with the help of another node we will update that perticular node.

There are various algorithms to find shortest path

(i) Dijkstra's Algo (Single source shortest path)
(ii) Bellman - Ford's Algo

Dijkstra's Algorithm ⇒

This algorithm works on greedy approach. It solves single source shortest path problem for a directed graph with non negative edge weight.

Algo DIJKSTRA (G, w, s)

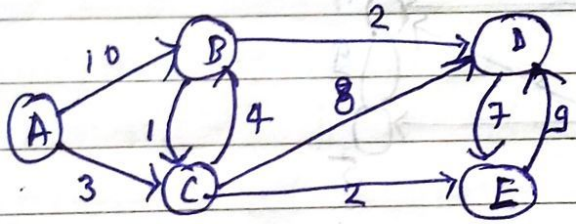
1. Initialise single source (G, s).
2. $S \leftarrow \phi$
3. $Q \leftarrow V[G]$

4. While $Q \neq \phi$
5. Do $u \leftarrow$ Extract_MIN(Q)
6. $S \leftarrow S \cup \{u\}$
7. For each vertex $v \in Adj[u]$
8. do RELAX (u, v, w)

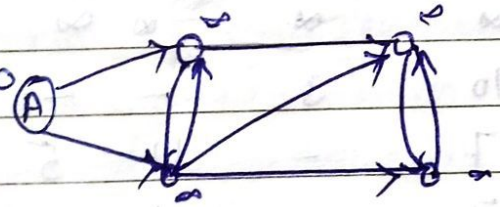
RELAX (u, v, w)
1. if $d[v] > d[u] + w(u, v)$
2. do $d[v] \leftarrow d[u] + w[u, v]$
3. $\pi[v] \leftarrow u$ (π = Predicessor)

[predicessor i.e. either another vertex or NIL]

Q. Consider the following graph & the shortest path using dijkstra's algorithms.

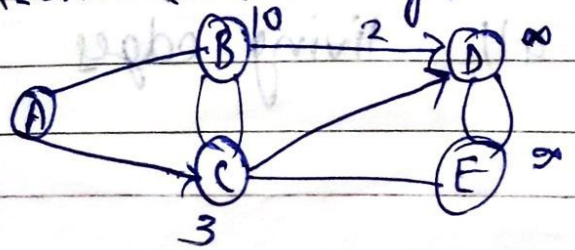


Ans → Initialise source



Q : A B C D E
0 ∞ ∞ ∞ ∞

→ A ← Extract_min(Q)
S ← {A}
RELAX all edges living A



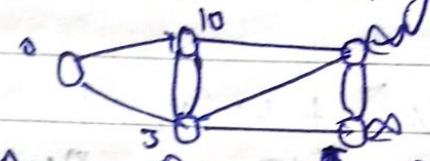
Q: A B C D E

D 10 3 ∞ ∞

C ← Extract_MIN(Q)

S ← {A, C}

RELAX all living edges



Q: A B C D E

0 ∞ ∞ ∞ ∞

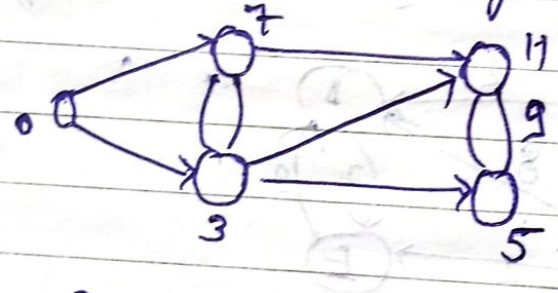
0 10 3

7 11 5 (3rd)

E ← Extract_MIN(Q)

S ← {A, C, E}

RELAX all living edges



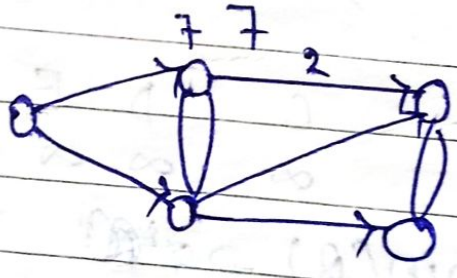
Q: A B C D E

0 ∞ ∞ ∞ ∞

10 3 — —

7 — 11 5

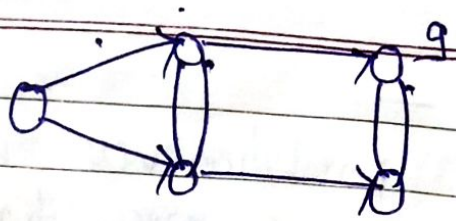
14



B ← Extract_MIN(Q)

S ← {A, C, E, B}

RELAX all living edges



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3		
		7		11	5
		7		11(2+9)	
				9	

D ← Extract-MIN(Q)

S ← {A, C, E, B, D}

RELAX all living edges

HASHING ⟹

The search time of an algorithm depends upon the no. of elements in it.

Hashing or Hash addressing is a searching technique which is independent of no. of elements. In the hashing, we apply hash function on key values to find memory address (L) and this notation is denoted by.

$H: K \longrightarrow L$

~~Hash function~~ If we get same value on applying hash function to different value set this situation is known as collision.

Hash function :—

Whenever we will select hash function there are two principle criteria for selecting it :—

(i) The function H should be very easy & quick to compute.

(ii) The function H as far as possible uniformly distribute the Hash addresses through out the set L so that there are a minimum no. of collisions.

Naturally there is no garantee the second condition can be completely fulfilled without actually going before hand of the keys or the addresses.

Method to evaluate the Hash function :-

i) Division Method ⇒

Choose the no. N larger than no. n of keys in K. (The no. M is usually chosen to be prime no. or no. without small division since this frequently minimises the no. of collisions) the Hash function H is defined by

$$H(k) = k \pmod M$$

$$H(k) = k \pmod M + 1$$

$$H : K \longrightarrow L$$

Here $k \pmod M$

(ii) Mid square method ⇒

then the hash function H is defined by $H(k) = l$ where l is obtained by deleting digits from both end of k^2.

(iii) Folding method ⇒

The key k is partition into a no. of part $k_1, k_2 \dots k_n$ where each part accept ~~positi~~ possibly the last has the same no. of digits as the required address then the parts are added together ignoring the last carry.

$$H(k) = k_1 + k_2 + \dots + k_n$$

where the leading digits carry if any are ignored.

Example –

Q. Let suppose we have 68 employees and unique four digit employee number is assigned to them. suppose L consists of two digit addresses 00, 01 -- 99.

Let consider the following 3 employees no. 3205, 7148, 2345

(i) division Method :–

Choose a prime no. n close to 99. Such as $n = 97$ then

H(3205) = 4
H(7148) = 67
H(2345) = 17

i.e. dividing 3205 by 97 gives a remainder 4, 7148 by 97 gives a remainder 67 & 2345 by 97 gives a remainder 17

In the case, if memory address begins with 01 rather than 00, we choose hash function

H(k) = k (mod m) + 1

H(3205) = 5, H(7148) = 68 & H(2345) = 18

(ii) Mid square method:—

k:	3205	7148	2345
k^2:	10272025	51093904	5499025
H(k):-	72	93	09

(iii) folding method:—

Choping the key k into 2 part and adding them gives the following hash addresses

K:	3205	7148	2345
	32+05	71+48	23+45
	37	119	68
H(K):-	37	19	68

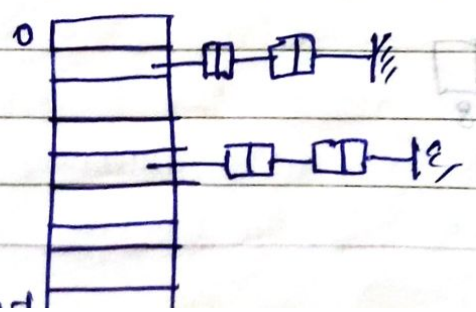
Alternatively 1st may want to reverse before adding.

3250 7184 2354
82 155 77
82, 55, 77

Hashing with chaining ⇒

The element in S are stored in hashable of size m where m is somewhat larger than n, the size of S. The hashtable is said to have n slots. If more than one key in S hashes into the same slot then we have a collision. In such a case, all keys that hash into the same slot are placed in the linked list associated with that slot this linked list is called the chain at that slot. The load factor of hashtable is defined to be alpha(α) = $\frac{n}{m}$ . . .

It represents the average no. of keys per slot usually $\alpha < 1$. If a large no. of insertion or deletion destroys its property a more suitable value of m is chosen & the keys are rehashed into a new table with a new hash function.

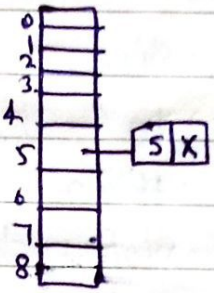


Q- Let us consider the element 5, 28, 19, 15, 20, 33, 12, 17, 10 into a chained hashtable. let us suppose the hash table T has 9 slot and hash function $h(k) = k \bmod 9$

Sol^n- The initial state of chain is empty.

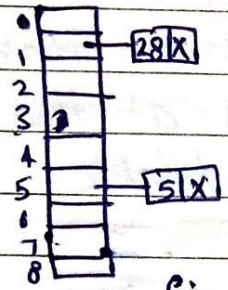
$h(5) = 5 \bmod 9 = 5$

Create a linked list for T(5) and insert value 5 in it.

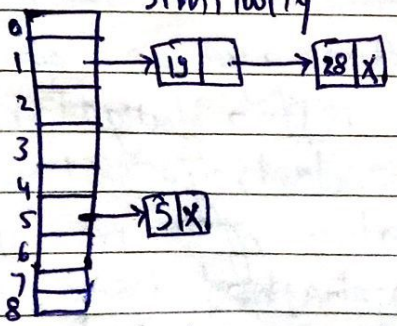


Similarly,

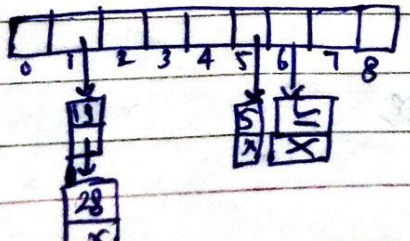
$h(28) = 28 \bmod 9 = 1$



Similarly $h(19) = 19 \bmod 9 = 1$



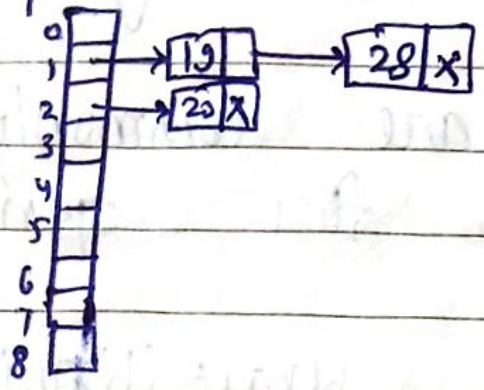
Similarly $h(15) = 15 \bmod 9 = 6$



Date / /

Page

Similarly $h(20) = 20 \bmod 9 = 2$



Hashing with open addressing :—

There are 3 technique are commonly used probe sequences required for open addressing

(i) linear probing $h(k,i) = (h'(k)+i) \bmod m$
(ii) Quardtic probing $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
(iii) Double hashing $h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m$

(i) linear probing :—

An ordinary hash function $h' : U[0, 1, \ldots m-1]$. The method of linear probing uses the hash function

$$h(k,i) = (h'(k) + i) \bmod m$$

where m is the size of hash table and $h'(k)$ is $k \bmod m$ (basic hash function)

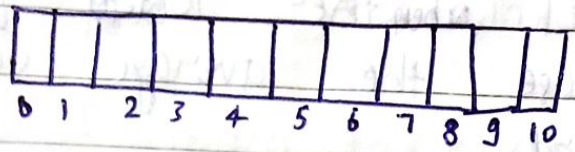
For $i = 0, 1, \ldots m-1$ given key k, the first slot is probed $T(h'(k))$. We next probed slot $T(h'(k)+i)$ and so on upto slot $T(m-1)$.

Then we wrapped around to slot $T[0]$, $T[1]$ until we finally probed slot $T[h'(k)-1]$. Since the initial probe position determine the entire probe sequence only m distinguished probe sequence are used with linear probing.

eg- Consider the keys 26, 37, 59, 76, 65, 86, to be inserted in hash table. There the size of hash table is 11 using the linear

probing. Consider the primary hash function
$h'(k) = k \bmod m$.

Solution ⇒ Initial state of hash table



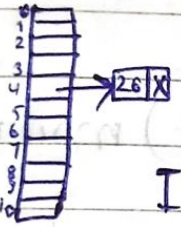
Insert 26, Now

$h(26,0) = (h'(26) + 0) \bmod 11$

$h(26,0) = (26 \bmod 11 + 0) \bmod 11$

$h(26,0) = (4) \bmod 11$

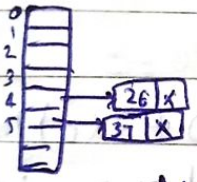
$h(26,0) = 4$



Insert 37, Now

$h(37,1) = (h'(37) + 1) \bmod 11$

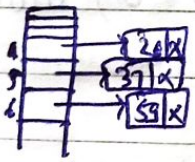
$h(37,1) = 5 \bmod 11 = 5$



Now, insert 59

$h(59,2) = (h'(59) + 2) \bmod 11$

$h(59,2) = 6 \bmod 11 = 6$



insert 76 $h(76,0) = 10$

insert 65 ⇒

$h(65,0) = 10$ T[10] is occupied

$h(65,1) = 0$

insert 86 ⇒ $h(86,0) = 9$

The main disadvantage of linear probing is that records tends to cluster i.e. appears next to one another, when the load factor > 50%. that will increase the average search time for a record.

To technique to minimise clustering are as :
(i) Quardric probing
(ii) Double hashing

(i) Quardric probing :—

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

eg - 76, 26, 37, 59, 21, 65 , $m = 11$, $c_1 = 1$, $c_2 = 3$

Sol :—

Insert 76 :—

$$h(76,0) = (76 \bmod 11 + 0 + 3 \times 0) \bmod 11$$
$$= (10 + 0 + 0) \bmod 11 = 10$$

Insert 26

$$h(26,0) = (26 \bmod 11 + 0 + 0) \bmod 11 = 4$$

Insert 37

$$h(37,0) = (37 \bmod 11 + 0 + 0) \bmod 11$$
$$= (4 + 0 + 0) \bmod 11 = 4$$

T(4) is not free, so next probe sequence is computed as

$$h(37,1) = (37 \bmod 11 + 1 + 3) \bmod 11$$
$$= (4 + 1 + 3) \bmod 11 = 8 \bmod 11 = 8$$

Insert 59

$h(59,0) = 4$, T[4] is not free

$h(59,1) = 8$, T[8] is also not free

$h(59,2) = (59 \bmod 11 + 2 + 3 \times 2^2) \bmod 11$

$= (4 + 2 + 12) \bmod 11 = 18 \bmod 11$

$h(59,2) = 7$

Insert 21

$h(21,0) = 10$, T[10] is not free

$h(21,1) = (21 \bmod 11 + 1 + 3) \bmod 11$

$= (10 + 1 + 3) \bmod 11 = 3$

T[3] is free, so insert key 21 at this place

Insert 65

$h(65,0) = 10$, T[10] is not free

$h(65,1) = 3$, T[3] is not free

$h(65,2) = 2$ T[2] is free, so insert key 65

Thus, after inserting all keys, the hash table is:—

0	1	2	3	4	5	6	7	8	9	10
/	/	65	21	26	/	/	59	37	/	76

(ii) Double hashing ⇒

$$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m$$

$h_1(k) = k \bmod m$

$h_2(k) = k \bmod m'$

eg - keys 76, 26, 37, 59, 21, 65 $m = 11$, $m' = 9$

Insert 76 :—

$h_1(76) = 76 \bmod 11 = 10$

$h_2(76) = 76 \bmod 9 = 4$

$h(76, 0) = (10 + 0 \times 4) \bmod 11 = 10$, T[10] is free

Insert 26 :—

$h(26, 0) = (h_1(76) + i\, h_2(76)) \bmod 11$

$h(26, 0) = (4 + 0 \times 8) \bmod 11 = 4$, T[4] is free

Insert 37

$h(37, 0) = (4 + 0 \times 1) \bmod 11 = 4$

T[4] is not free

So $h(37, 1) = (4 + 1 \times 1) \bmod 11 = 5$, T[5] is free

Insert 59

$h(59, 0) = 4$, T[4] is not free

$h(59, 1) = (4 + 1 \times 5) \bmod 11 = 9 \bmod 11 = 9$

T[9] is free, so insert key 59.

Insert 65

$h(65, 0) = 10$, T[10] is not free

$h(65, 1) = (10 + 1 \times 2) \bmod 11 = 1$

T[1] is free, so insert key 65 at this place.

Thus after insertion of all keys the final hash table is

0	1	2	3	4	5	6	7	8	9	10
/	65	21	/	26	37	/	/	/	59	76

	Graph	Tree
1.	It is a non linear data structure.	Tree is non-linear data structure.
2.	It is a collection of vertices & edges	collection of nodes & edges
3.	There is no unique node called root.	There is unique node called root.
4.	A cycle can be formed	There will not be any cycle.
5.	finding shorest path in networking graph is used.	Eg - for game trees, decision trees.

	BFS	DFS
1.	Breadth first search uses Queue data structure.	Depth first Search uses Stack data structure.
2.	BFS is more suitable for searching vertices which are closer to given source.	DFS is more suitable where there are solutions away from source.
3.	Siblings are visited before the children.	Children are visited before the siblings.
4.	It is a vertex based technique.	It is an edge based technique.
5.	BFS does not use backtracking concept.	DFS uses backtracking to traverse all unvisited nodes.
6.	BFS is slower than DFS.	DFS is faster than BFS.