Stack

→ A stack is a list of elements in which an element may be inserted or deleted only at one end, called the top of the stack

Two basic operations:

1. Push
2. Pop.

→ Stack is used for procesing of data when certain steps of the procesing must be postponed until other condutions are fulfilled.

→ Array representation of stack:-

1. maxstk = max number of elements in stack
2. Top = 0 empty stack (under flow)
3. Top = maxstk full stack (over flow)

Algo :- Push (Stack, top, maxstk, item)

Step I :- Initialization

Step II :- if Top = maxstk,
then print overflow and Return

Step III :- Set top = top + 1 [increase the top by 1]

Step IV :- Set stack[top] = Item
[Insert the item at new top position]

Step V :- exit

Algo :- Pop (stack, top, item)

Step I :- Initialization

Step II :- if Top = 0, then
print Underflow and return

Step III :- Set item = stack[Top]
[assign Top element to item]

Step IV :- Set top = top - 1 [decrease top by 1]

Step V :- exit

→ minimizing overflow.

→ Polish Notation :—

Arithmetic expressions :

constants and operations

Precedence : 1. Exponentiation (↑)
2. multiplication (*) and division (/)
3. Addition (+) and substraction (-)

polish notation :-

1. Infix (A+B)
2. Prefix +AB polish notation
3. postfix AB+ Reverse polish notation

Evaluation of Reverse polish notation :-

Evaluate postfix expression

Step I :- Initialization

Step II :- add a right parenthesis ")" at the end of P.

Step III :- Scan arithmetic expression P from left to right and Repeat Steps 4 and 5 for each element of P until the end ")" parenthesis is encountered.

Step IV :- If an operand is encountered, put it on stack

Step V : If an operator ⊗ is encountered, then

a) Remove the two top elements of stack, where A is the top element and B is the next to top element.

b.) evaluate B⊗A.

c.) place the result of(b) back on stack.

[End if]

[End of step 2 loop]

Step VI :- Set value equal to the top element on stack.

Step VII :- Exit

Example:- P = 5, 6, 2, +, *, 12, 4, /, -
ans = (37)

→ Infix to postfix.

A+(B*C-(D/E↑F)*G)*H.

procedure:-

1. add left and right parenthesis to the expression.
2. add operand to expression and operator to stack.
3. if an operator is encountered then
   * if top of the stack operator have higher or same precedence that present operator than pop the higher precedence operator and push lower precedence operator.
   * else push present operator to the stack.
4. if left parenthesis is encountered, push into the stack.
5. if right parenthesis is encountered, pop all the operator till left parenthesis in stack is encountered. pop left parenthesis also.

(A + (B * C - (D / E ↑ F) * G) * H)

	Stack	expression
A	(	A
+	(+	A
(	(+(	A
B	(+(	A B
*	(+(*	A B
C	(+(*	A B C
-	(+(-	A B C *
(	(+(-(	A B C *
D	(+(-(	A B C * D
/	(+(-(/	A B C * D
E	(+(-(/	A B C * D E
↑	(+(-(/↑	A B C * D E
F	(+(-(/↑	A B C * D E F
)	(+(-	A B C * D E F ↑ /
*	(+(-*	A B C * D E F ↑ /
G	(+(-*	A B C * D E F ↑ / G
)	(+	A B C * D E F ↑ / G * -
*	(+*	A B C * D E F ↑ / G * -
H	(+*	A B C * D E F ↑ / G * - H
)	Empty	A B C * D E F ↑ / G * - H * +

postfix expression.

Algorithm to polish notation :-

Polish (Q, P)

Q = infix notation of expression.

P = postfix notation of expression (equivalent

Step 1:- Initialization

Step 2:- Push "(" onto stack and add ")" to the end of Q.

Step 3:- Scan Q from left to right and Repeat Step 4 to 7 for each element of Q until the stack is empty.

Step 4:- If an operand is encountered, add it to P.

Step 5:- If a left parenthesis is encountered, push it onto stack

Step 6:- If an operator ⊗ is encountered, then:

a.) Repeatedly pop from stack and add to P each operator (on the top of the stack) which has the same or higher precedence than ⊗.

b.) add ⊗ to stack

[End of If structure]

Step 7:- If a right parenthesis is encountered then

a.) Repeatedly pop from stack and add to P each operator (on the top of stack) until a left parenthesis is encountered.

b.) Remove the left parenthesis. (do not add left or right parenthesis to P).

[End of if structure]

[End of step 3 loop]

Step 8:- Exit

→ Application of stack:-

1. Quick sort

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

pivot ↑ P ⟶ ⟵ ↑ Q.

1. Scan from left for larger element and right for smaller element.

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

Larger from left

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

Smaller from right

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

Larger from left

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

after first pass pivot = 44 get its correct position

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

first sublist — pivot — second sublist

→ algorithm of quick sort is divided into two sub algorithms.

1. Quick is for scan the elements from left and right and get the correct position for pivot element.
2. Quicksort is for divide the array in subarray to get the sorted subarrays using quick algo.

algorithm :- Quick (A, N, BEG, END, LOC)

A = given array.

N = number of elements in array.

BEG = ~~beginning~~ beguning of array.

END = ending of array.

LOC = location of pivot element

pivot = A[LOC]

Step 1:- Initialization

Set LEFT = BEG, RIGHT = END and LOC = BEG.

Step 2:- Scan from right to left.

a.) Repeat while A[LOC] ≤ A[RIGHT] and LOC ≠ RIGHT.

RIGHT = RIGHT - 1

[End of loop]

b.) if LOC = RIGHT, then: Return

c.) if A[LOC] > A[RIGHT], then:

i.) [Interchange A[LOC] and A[RIGHT]]

TEMP = A[LOC]

A[LOC] = A[RIGHT]

A[RIGHT] = TEMP.

ii) Set LOC = RIGHT

iii) Go to step 3.

[End of If structure].

Step 3:- Scan from left to right.

a.) Repeat while A[LEFT] ≤ A[LOC] and LEFT ≠ LOC

LEFT = LEFT + 1

[End of loop]

b.) if LOC = LEFT, then return.

c.) if A[LEFT] > A[LOC], then

i.) Interchange A[LEFT] and A[LOC]

TEMP = A[LOC]

A[LOC] = A[LEFT]

A[LEFT] = TEMP.

ii) Set LOC = LEFT

iii) Go to Step 2.

[End of if structure]

→ algorithm for quick sort:-

Step 1:- Initialization

TOP = NULL.

Step 2:- Push boundary conditions when N≥2

if N>1, then:

TOP = TOP + 1

LOWER[1] = 1

UPPER[1] = N

Step 3:- Repeat step 4 to 7 while TOP ≠ NULL.

Step 4:- Pop sublist from stack.

Set BEG = LOWER[TOP]

END = UPPER[TOP]

TOP = TOP - 1

Step 5:- Call Quick (A, N, BEG, END, LOC)

Step 6:- [Push left sublist onto stack when it has 2 or more elements]

if BEG < LOC-1, then:

TOP = TOP+1,

LOWER[TOP] = BEG

UPPER [TOP] = LOC-1

[End of if structure]

Step 7:- [Push right sublist onto stack when it has 2 or more elements]

if LOC+1 < END then:

TOP = TOP+1,

LOWER [TOP] = LOC+1,

UPPER [TOP] = END.

[End of If structure]

[End of step 3 loop]

Step 8 :- Exit.

Complexity of quick sort :-

Best and average case :- $O(n^2 \log n)$

Worst case :- $O(n^2)$

Recursion:- A module or function to call itself.

→ A function α either calls itself directly or calls a function β that in turn calls the original function α.

→ to make sure not continue to run indefinitely, a recursive procedure must have the following two properties:

1. There must be a base criteria, for which the procedure does not call itself.
2. Each time the procedure does call itself, it must be closer to the base criteria.

Factorial without recursion:-

Algo:- Factorial (Fact, N)

Fact = N!

Step 1:- Initialization

Step 2:- if N = 0; then Set Fact = 1 and Return

Step 3:- Set Fact = 1

Step 4:- Repeat for i = 1 to N.

set Fact = i * Fact.

[End of loop]

Step 5:- Return.

Factorial using Recursion :-

Algo:-

Factorial (Fact, N)

Step 1:- Initialization

Step 2:- if N=0, then Set Fact = 1 and Return

Step 3:- Call Factorial (Fact, N-1)

Step 4:- Set Fact = N * Fact.

Step 5:- Return

→ Fibonacci Series:-

a.) if n=0 or n=1, then $F_n = n$.

b.) if n>1, then Fib(n) = Fib(n-1) + Fib(n-2)

Algo:-

Fibonacci (n)

Step 1:- Initialization

Step 2:- if N=0 or N=1, set Fibonacci (N) = N and Return.

Step 3:- set Fibonacci (n) = Fibonacci (n-1) + Fibonacci (n-2)

Step 4:- Return

* Divide and conquer may be viewed as recursive procedure.

Ackermann Function:-

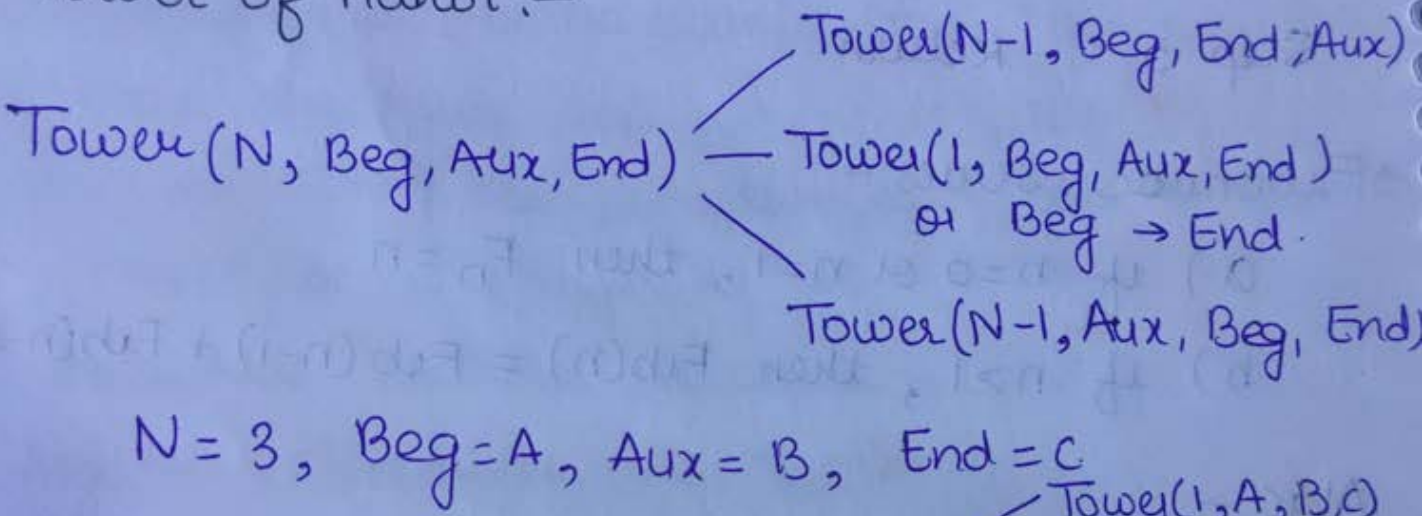
Classic recursive function and Divide & conquer approach.

Definition:- a.) if $m=0$, then $A(m,n) = n+1$

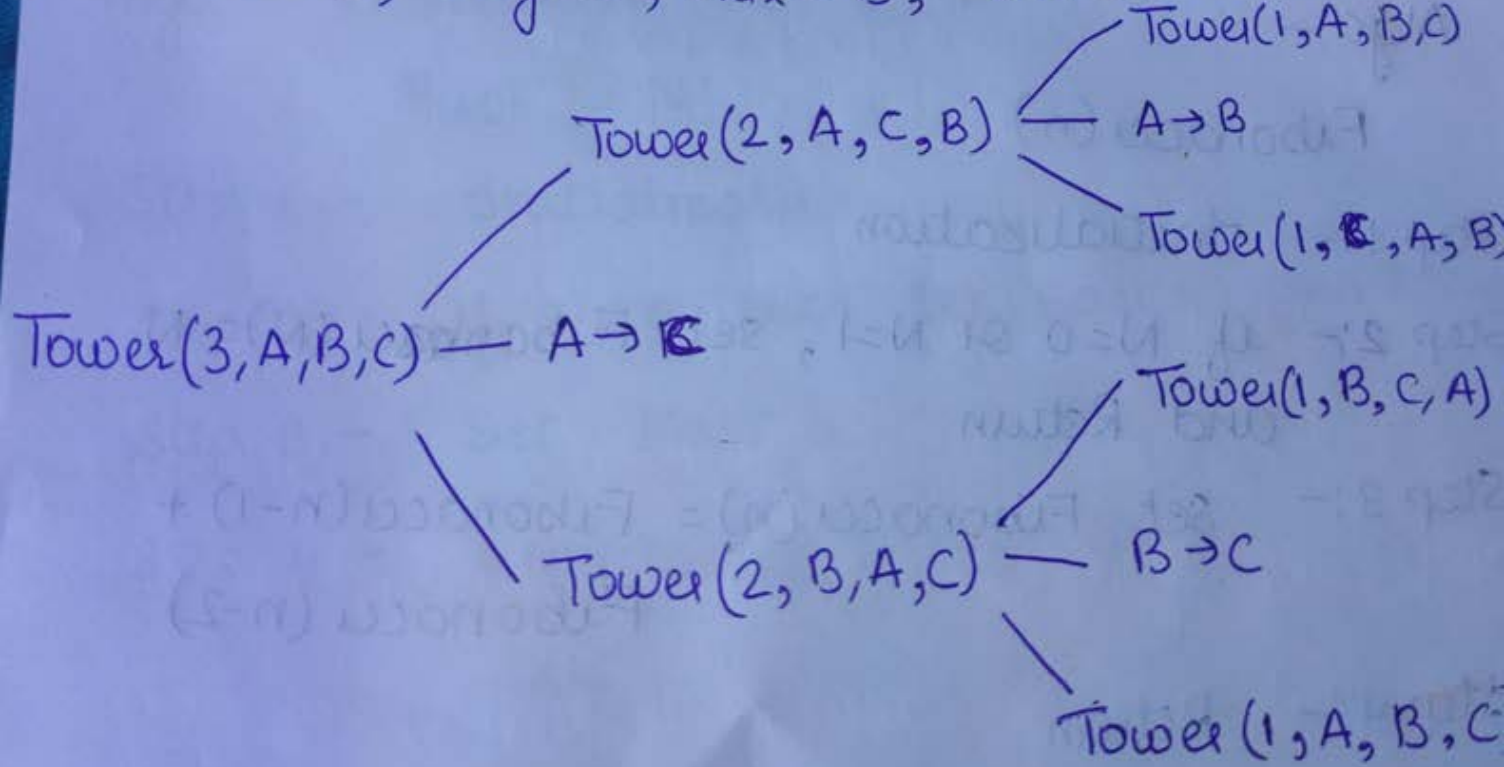
b.) if $m \neq 0$ but $n=0$, then $A(m,n) = A(m-1, 1)$.

c.) if $m \neq 0$ but $n \neq 0$, then $A(m,n) = A(m-1, A(m, n-1))$

Tower of Hanoi:-



N = 3, Beg = A, Aux = B, End = C



A → C, A → B, C → B, A → B, B → A, B → C, A → C.

Algo :- Tower (N, Beg, Aux, End)

N is number of disks

Step 1 :- Initialization

Step 2 :- if N=1, then:

a.) Write : Beg → End

b.) Return

[End of If structure]

Step 3 :- [Move N-1 disks from peg Beg to Beg Aux]

call Tower(N-1, Beg, End, Aux)

Step 4 :- Write : Beg → End.

Step 5 :- [Move N-1 disks from peg Aux to peg End]

call Tower(N-1, Aux, Beg, End).

Step 6 :- Return.

Queues

→ A queue is a linear list in which elements can be deleted only one end called front and can be inserted only one end called rear.

→ Queue is a FIFO structure

Representation of queue:-

→ Queue will be maintained using linear array with two pointers. Front and Rear.

→ Whenever an elements is added to the queue, the value of Rear is increased by 1.

Rear = Rear + 1

→ Whenever an element is removed from the queue, the value of Front is increased by 1

Front = Front + 1

Full.

Front

Rear.

→ Insert in a queue:-

Algo :- QInsert (Queue, N, Front, rear, item)

Step 1:- Initialization

Step 2:- [Check queue is already full]

If Front = 1 and Rear = N, or if Front = Rear + 1, then:

Write Overflow and Return

Step 3:- Find values of Rear.

If Front = Null, then [queue is empty]

Set Front = 1 and Rear = 1

Else if Rear = N, then:

Set Rear = 1

Else

Set Rear = Rear + 1

[End of If structure].

Step 4:- Set Queue [Rear] = Item

[Insert new element]

Step 5:- Return

→ Delete item from a queue:-

Algo:- QDelete (Queue, N, Front, Rear, Item)

Step 1:- Initialization

Step 2:- [Check queue is empty]

If Front = Null, then Write Underflow and Return.

Step 3:- Set Item = Queue [Front]

Step 4:- Find new value for Front.

If Front = Rear, then:

[only one item in queue]

Set Front = Null and

Rear = Null

Else if Front = N, then

Set Front = 1

Else:

Set Front = Front + 1

[End of if structure]

Step 5:- Return

Circular Queue:-

→ Shows queue is full, if the queue is partially free.

→ A circular queue, by definition, is a queue in which the element next to the last element is the first element.

Algorithm:- Insert in circular queue.

Step 1:- Initialization

Step 2:- If rear points to the end of the queue then go to step 3, else goto step 4.

Step 3:- Let Rear value = 0

Step 4:- Increase the rear value by 1.

Step 5:- If front and the rear pointers point at the same place in the queue and that place has a not null value, then queue is overflow, then go to step 7 else goto step 6.

Step 6:- Rear = Rear + 1

Queue [Rear] = Item

Step 7:- Exit.

Deques:- A deque or de queue or double ended queue, is a linear list in which elements can be added or deleted at either end but not in the middle.

→ left and Right are two pointers which point to the ends of deque.

→ Input-restricted deque is a deque, which allows insertions at only one end of the

list but allows deletions at both ends of the list.

→ Output - restricted deque is a deque which allows deletions at only one end of the list but allows insertions at both ends of the list.

→ complication may arise at overflow and underflow.

Priority Queues:- A priority queue is a collection of elements such that each element has ~~a~~ assigned priority and delete and process according to rules:

1. Higher priority element processed first.
2. Same priority elements are processed according to their order of insertion

→ A prototype of a priority queue is timesharing

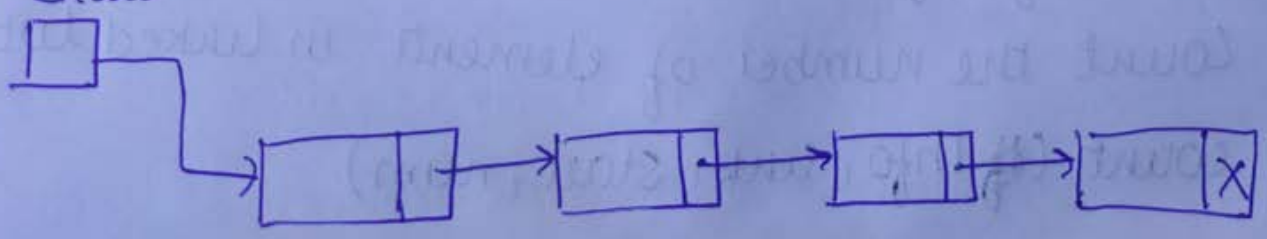
→ maintain a priority queue in memory is to use a separate queue for each level of priority

Linked list

→ In array, data are related by physical address of memory, not by their values.
→ Insert and delete are expensive.
→ No additional space facility provided.
→ A linked list, or one-way list, is a linear collection of data elements, called nodes, where the linear order is given by means of pointers.
→ Each node is divided into two parts:
→ first part → data field.
second part → link field.
→ A list pointer variable (start or name) contains the address of the first node in the list.
→ The pointer of the last node contains a special value, called the null pointer, which is any invalid address.

Traversing a linked list :-

→ A linked list in memory stored in linear array Info and link with start pointer pointing to first element.



Algo :- Traversing a linked list

List = linked list

process = operation to perform

ptr = pointer

Step 1 :- Initialization

Step 2 :- Set PTR = START

[Initialize pointer PTR]

Step 3 :- Repeat Step 4 and 5 while PTR ≠ NULL

Step 4 :- Apply Process to Info [PTR]

Step 5 :- Set PTR = LINK [PTR]

[PTR now points to the next node]

[End of Step 3 loop]

Step 6 :- Exit

→ Linear array and linked list traversing are similar from the fact that both are linear structures which contains a natural linear ordering of elements.

→ Count the number of elements in linked list

count (Info, link, start, num)

Step 1:- Initialization

Step 2:- Set Num = 0 [Initializes counter]

Step 3:- Set PTR = START [Initializes pointer]

Step 4:- Repeat Step 5 and 6 while PTR ≠ NULL

Step 5:- Set Num = Num + 1

Step 6:- Set PTR = Link [PTR]

[End of step 3 loop]

Step 7:- Return

→ Searching a linked list

Search item in linked list.

List is unsorted

Algo:- Search (Info, Link, Start, Item, Loc)

Step 1:- Initialization

Step 2:- Set Ptr = Start

Step 3:- Repeat step 4 while Ptr ≠ Null

Step 4:- If Item = Info [ptr], then:

Set Loc = Ptr, and Exit

else:

Set Ptr = Link [ptr]

[End of Step 3 loop]

Step 5:- [Search unsuccessful]

Set LOC = null.

Step 6:- Exit

complexity of algo:-

→ same as linear search.

worst case :- O(n)

Average case :- O(n/2)

List is sorted :-

Algo:- SRCHSL (Info, Link, start, item, loc)

Step 1:- Initialization

Step 2:- Set ptr = start

Step 3:- Repeat step 4 while ptr ≠ null

Step 4:- If item > Info[ptr], then:

Set ptr = Link[ptr]

[ptr now points to next node]

Else if item = Info[ptr], then:

Set LOC = ptr

[Search successful]

Else:

Set LOC = null, and Exit

[Item now exceeds Info[ptr]]

[End of If structure]

[End of step 3 loop]

Step 5:- loc = null

Step 6:- Exit

→ Binary search algo cannot be applied.

Memory allocation: Garbage collection

→ a special list is maintained which consists of unused memory cells.

→ This list contains its own pointer, is called as avail list, free storage list or free pool.

Garbage Collection:-

→ immediately reinsert the space into the free-storage list after deletion

→ periodically collect all the deleted space onto the free-storage list, known as garbage collection.

→ Garbage collection has two steps:

1. run through all lists, tagging currently using cells.
2. collect all untagged cells onto free space.

→ Overflow and underflow

Overflow ⇒ Avail = Null, underflow ⇒ start = Null.

Insertion into a linked list:

→ Insert a node between node A and node B.

→ new node to be inserted is node n.

→ Three pointer fields are changed as follows:

1. node A now points to node n, which is previously pointed by AVAIL.
2. AVAIL now points second node, which is previously pointed by node n.
3. node n now points to node B, which is previously pointed by node A.

→ insertion algo will include the following steps:

1. Checking the AVAIL list.
2. Remove first node from AVAIL list

   new = AVAIL , AVAIL = link[AVAIL]
3. Copy new info into the new node

   Info[new] = Item

→ Insert at the beginning of a list

Algo :- Insfirst (info, link, start, avail, item)

Step 1:- Initialization

Step 2:- [Check for overflow]

If AVAIL = NULL, then:
write: overflow, and exit

Step 3:- [Remove first node from AVAIL list]
Set new = AVAIL and AVAIL = LINK[AVAIL]

Step 4:- Set Info[new] = Item [copy new data in new nod

Step 5:- Set Link[new] = start
[new node points to original first node]

Step 6:- Set start = new
[changes start so it points to new node]

Step 7:- Exit

→ Insert after a given node or on location:

Algo:- Insertloc(Info, link, start, AVAIL, loc, item)

Step 1:- Initialization

Step 2:- [Check for overflow]
If AVAIL = null, then:
write: overflow, and exit

Step 3:- [Remove first node from AVAIL list]
Set new = AVAIL and AVAIL = Link[AVAIL]

Step 4:- Set Info[new] = Item [copy new data in new node]

Step 5:- If loc = null, then: [Insert as first node]
Set Link[new] = start and start = new.
Else: [Insert after given node]
Set Link[new] = link[loc]

and link [loc] = new

[End of if structure]

Step 6:- Exit

→ Insert into a sorted linked list:

→ first find the proper location to insert.

→ then insert on the location.

Algo :- FindA(Info, link, start, item, loc)

Step 1:- Initialization

Step 2:- [check empty list]

If start = null, then:

Set loc = null and return.

Step 3:- if Item < Info [start], then set loc = null and return.

Step 4:- Set save = start and ptr = link [start]

[Initializes pointers]

Step 5:- Repeat steps 6 and 7 while ptr ≠ null

Step 6:- if item < Info [ptr], then:

Set loc = save, and Return

[End of if structure]

Step 7:- Set save = ptr and ptr = link [ptr]

[Update pointers]

[End of step 5 loop]

Step 8:- Set loc = save

Step 9:- Return

Deletion from a linked list

Three pointer fields are changed after deletion:

1. The next node A now points to node B, where node N pointed previously.
2. node N now points to first node of avail list, where avail previously pointed.
3. Avail now points to the deleted node N.

Deleting the node following a given node

Algo:- Del (Info, link, start, avail, loc, locp)

Step 1:- Initialization

Step 2:- If locp = null, then:

Set start = link[start]

[delete first node]

Else:

Set link[locp] = link[loc]

[delete node N]

[End of If structure]

Step 3:- [Return deleted node to the avail list]

Set link[loc] = avail and avail = loc.

Step 4:- Exit

Deleting the node with a given item of information

Algo:- FindB(Info, link, start, item, loc, locp)

Step 1:- Initialization

Step 2:- [check empty list]
if start = null, then:
Set loc = null and locp = null
and Return
[End of if structure]

Step 3:- [check item is in first node]
if Info[start] = item, then:
Set loc = start and locp = null
and return
[End of if structure]

Step 4:- Set save = start and ptr = link[start]
[Initialize pointer]

Step 5:- Repeat step 5 and 6 while ptr ≠ null

Step 6:- If Info[ptr] = item, then:
Set loc = ptr and locp = save
and return

[End of if structure]

Step 7:- Set save = ptr and ptr = link[ptr]
[Update pointer]
[End of step 5 loop]

Step 8:- loc = null
[Search unsuccessful]

Step 9:- Return.

Delete the found item from linked list

Delete(info, link, start, avail, item)

Step 1:- Initialization

Step 2:- find the item from linked list using procedure findB(info, link, start, item, loc, locp)

Step 3:- if loc = null, then
write: item not found in list and exit

Step 4:- [Delete node]
if locp = null, then:
Set start = link[start] [delete first node]
Else
Set link[locp] = link[loc]
[End of if structure]

Step 5:- [Return deleted node to the avail list]

Set link[loc] = avail

and avail = loc

Step 6:- Exit

Header linked lists

A header linked list is a linked list which always contains a special node, called the header node, at the beginning of the list.

Two kinds of header lists:

1. grounded header list
2. Circular header list

→ In header list, first node is the node following the header node and location of the first node is link[start], not start, as with ordinary linked list.

→ Circular header lists are frequently used due to ease to state and implement using header list.

Two properties of circular header lists:

. The null pointer is not used, and hence all

pointers contain valid addresses.

2. Every (ordinary) node has a predecessor, so the first node may not require a special case.

Traversing a Circular Header list

Step 1:- Initialization

Step 2:- Set ptr = link[start]

[Initializes the pointer ptr]

Step 3:- Repeat steps 3 and 4 while ptr ≠ start

Step 4:- Apply Process to Info[ptr]

Step 5:- Set ptr = link[ptr]

[ptr now points to the next node.]

[End of step 2 loop]

Step 6:- Exit

Circular linked list

- In a circularly linked list, the link field of the last node points to the first node of the list.
- Last node does not contain a null pointer, but address of first node.
- Insertion and deletion same as single way list.

Two-way lists (or doubly linked lists)

A two-way lists is a linear collection of data elements, called nodes, where each node N is divided into three parts:

1. An information field INFO
2. Next pointer field NEXT
3. Previous pointer field PRE

Algo:- Insert(Info, Next, pre, start, avail, locA, locB, item)

Step 1:- Initialization

Step 2:- [Check overflow]
if avail = null, then
write: overflow and exit

Step 3:- [Remove node from avail list]
Set new = avail, avail = next[avail],
info[new] = item

Step 4:- [Insert node]
Set next[locA] = new, ~~pre~~ next[new] = locB
pre[locB] = new, pre[new] = locA

Step 5:- Exit

Algo :- Delete (Info, next, pre, start, avail, loc)

Step 1 :- Initialization

Step 2 :- [Delete node]

Set next[pre[loc]] = next[loc] and
pre[next[loc]] = pre[loc].

Step 3 :- [Return node to avail list]

Set ~~For~~ next[loc] = avail and
avail = loc.

Step 4 :- Exit

Sorting

Sorting refers to the operation of rearranging the contents of array A so that they are increasing in order such that

$$A_1 \leq A_2 \leq A_3 \leq \dots \leq A_n.$$

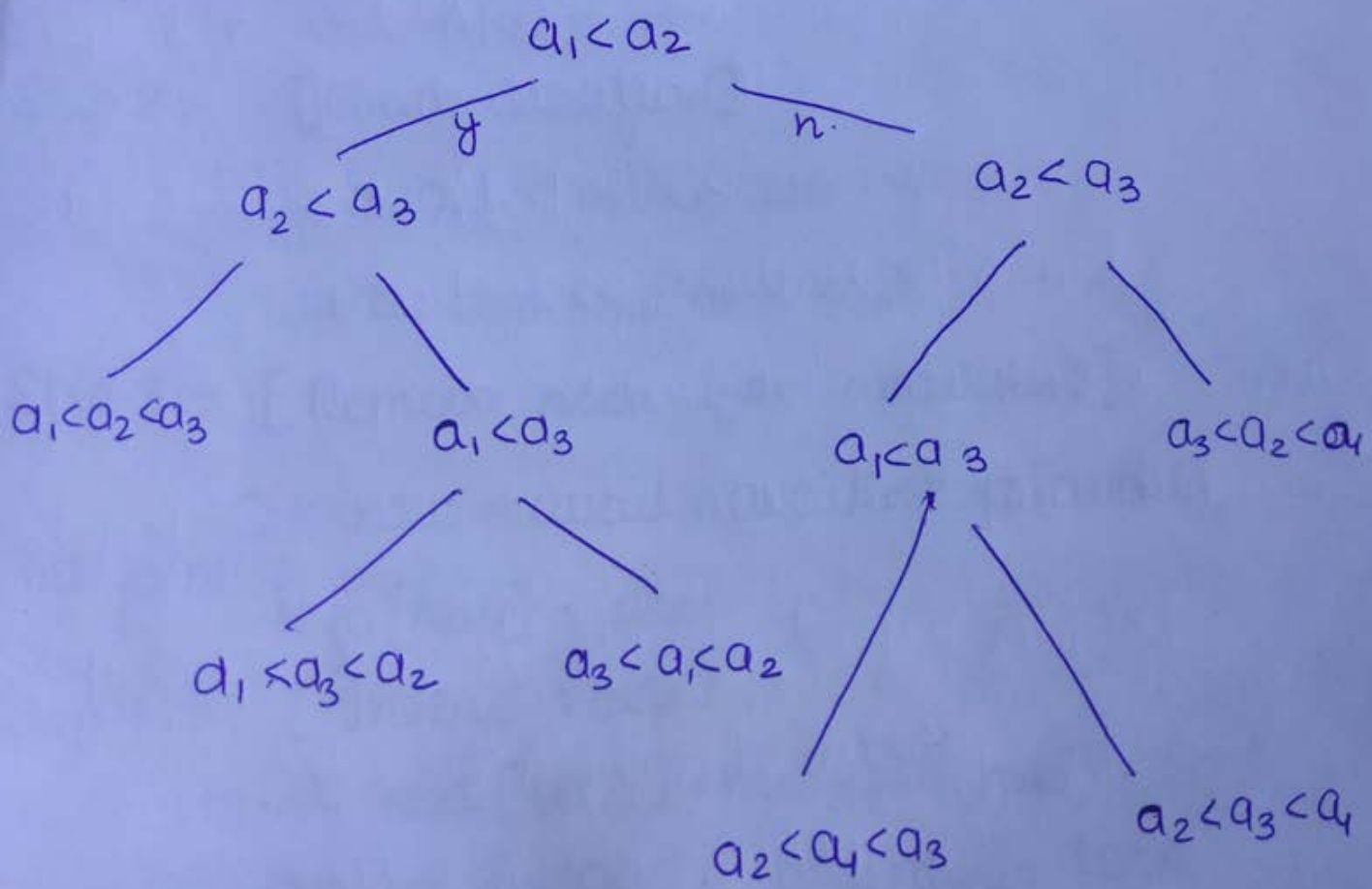
→ Sequence of operation

a. Comparison

b. Interchange

c. Assignment.

lower bounds:-



n = total number of element = 3

D = depth of tree

E = external path length

$D = 3$

$E = \frac{1}{6}(2+3+3+3+3+2) = 2.667$

$2^D \geq N$

N = external node.

$D \geq \log N$.

minimum external path length E(L).

$$E(L) = N \log N + O(n) \geq n \log N$$

→ N log N comes from N paths with length log N or log N-1.

→ O(n) - atmost N nodes on the deepest level.

→ Sorted order :- Ascending or descending

→ Sort stability is an attribute of a sort, indicating that data with equal keys maintain their relative input order in the output

Insertion Sort

Array A[1], A[2], A[3], ..., A[N]

Pass 1:- A[1] sorted

pass 2:- A[2] insert at proper location A[1], A[2]

pass 3:- A[3] inserted at proper location
A[1], A[2], A[3].

pass n:- A[N] A[1], A[2], A[3], ..., A[N]

Example:- 77, 33, 44, 11, 88, 22, 66, 55

Algo:- Insertion (A, N)

Step 1:- ~~Set A[0] = -∞~~
[Initialization]

Step 2:- Set A[0] = -∞
[Initialize sentinel element]

Step 3:- Repeat Step 4 to 6 for K=2, 3, ..., N:

Step 4:- Set Temp = A[K] and ptr = K-1

Step 5:- Repeat while temp < A[ptr]:

a.) Set A[ptr+1] = A[ptr].

b) Set ptr = ptr+1

[End of loop]

Step 6 :- Set A[ptr + I] = Temp

[Insert element in proper loc]

[End of Step 2 loop]

Step 7 :- Return

Complexity :-

Worst case = $\frac{n(n-1)}{2} = O(n^2)$

Average case = $\frac{n(n-1)}{4} = O(n^2)$.

Selection Sort

→ First find the smallest element in the list.

→ swap with first unsorted element.

Example :-

	77	33	44	11	88	22	66	55
K=1, loc = 4	(77)	33	44	(11)	88	22	66	55
K=2, loc = 6	11	(33)	44	77	88	(22)	66	55
K=3, loc = 6	11	22	(44)	77	88	(33)	66	55
K=4, loc = 6	11	22	33	(77)	88	(44)	66	55
K=5, loc = 8	11	22	33	44	(88)	77	66	(55)
K=6, loc = 7	11	22	33	44	55	(77)	(66)	88
K=7, loc = 7	11	22	33	44	55	66	77	88

Find minimum

Procedure : min(A, K, N, loc)

Step 1:- Initialization

Step 2:- Set min = A[K] and loc = K.
[Initialize pointer].

Step 3:- Repeat for J = K+1, K+2, ..., N:
if min > A[J], then:
Set min = A[J] and loc = J.
[End of loop]

Step 4:- Return.

Algo :- Selection Sort

Selection (A, N)

Step 1:- Initialization

Step 2:- Repeat steps 3 and 4 for K = 1 to N-1:

Step 3:- call min (A, K, N, loc)

Step 4:- [Interchange A[K] and A[loc]]
Set Temp = A[K], A[K] = A[loc]
A[loc] = Temp.
[End of step 2 loop]

Step 5:- Exit

Complexity:-

Worst case = $\frac{n(n-1)}{2} = O(n^2)$

Average case = $\frac{n(n-1)}{2} = O(n^2)$

Merging:-

Let A is a sorted list with a elements and B is a sorted list with b elements. Merging operation combines the elements of A and B into a single sorted list C with c = a+b elements.

Algo:- Merging(A, R, B, S, C)

Step 1:- Initialization

Step 2:- Set NA = 1, NB = 1, ptr = 1.

Step 3:- Repeat while NA ≤ R and NB ≤ S:

if A[NA] < B[NB]

a.) [assign element from A to C]

Set C[ptr] = A[NA]

b.) Set ptr = ptr + 1 and NA = NA + 1

else

a.) [assign element from B to C]

Set C[ptr] = B[NB]

b.) Set ptr = ptr + 1 and NB = NB + 1

[End of if structure]

[End of loop]

Step 4:- [Assign remaining elements to C]

if NA > R, then:

Repeat for K=0, to S-NB.

Set C[ptr+K] = B[NB+K]

Else [End of loop]

Repeat for K=0 to R-NA

Set C[ptr+K] = A[NA+K]

[End of loop]

[End of if structure]

Step 5:- Exit

Complexity of merging = O(n)

→ After Pass K, the array A will be partitioned into sorted sub arrays where each sub array, except possible the last, will contain exactly $L = 2^K$ elements.

Algo:- Merge pass (A, N, L, B)

Step 1:- Initialization

Step 2:- Set Q = int(N/(2*L)), S = 2*L*Q. and R = N-S

Step 3:- Repeat for J=1 to Q.

a.) Set LB = 1+(2*J-2)*L
[find lower bound of first array]
b.) call merge(A, L, LB, A, L, LB+L, B, LB)
[End of loop]

Step 4:- [only one subarray left].
if R≤L, then
Repeat for J=1 to 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)
[End of if structure]

Step 5:- Return

merge sort algo:-

algo (A, N)

Step 1:- Initialization

Step 2:- Set L=1 [Initialize the number of elements in subarray].

Step 3:- Repeat step 4 to 6 while L<N

Step 4:- call mergepass (A, N, L, B)

Step 5 :- Call mergepass (B, N, 2*L, A)

Step 6 :- Set L = 4*L

[End of step 3 loop]

Step 7 :- Exit

Procedure :- Merge (A, R, LBA, S, LBB, C, LBC)

Step 1 :- Initialization

Step 2 :- Set NA = LBA, NB = LBB, ptr = LBC,
UBA = LBA + R - 1, UBB = LBB + S - 1

Step 3 :- call merging and replace R by UBA and S by UBB.

Step 4 :- Call merging and replace R by UBA and S by UBB.

Step 5 :- Return

Complexity :-

worst case :- O(n log n)

average case :- O(n log n)

extra memory requried = O(n).

Shell Sort

→ algorithm is based on insertion sort.

→ also call as diminishing increment sort.

Example:-

13 17 19 10 15 11 16 18 14 12 10 16 11 15 17

13 14 19 18 12

complexity:-

$O(n^{3/2})$

Radix Sort:-

Example:- 348, 143, 361, 423, 538, 128, 321, 543, 366

complexity:-

Total comparison $C(n) \leq d * s * n$

d - digit

s - pass

n - elements

in worst case, $s = n$

So $C(n) = O(n^2)$

Best case $s = \log_d n$ Hence, $C(n) = O(n \log n)$.

Trees

→ The structure is mainly used to represent data containing a hierarchical relationship between elements.

Binary tree :- A binary tree T is defined as a finite set of elements, called nodes, such that

a.) T is empty (called null tree or empty tree)

or

b.) T contains a distinguished node R, called root of T, and remaining nodes of T form an ordered pair of disjoint binary trees T_1 and T_2, where T_1 is called left successor of R and T_2 as right successor of R.

→ Any node N in a binary tree T has either 0, 1 or 2 successors.

→ Trees T and T' are said to be similar if they have same structure or in other word, same shape.

→ Trees are used to represent algebraic Exp.

E = (a-b)/((c*d)+e)

Terminology:- 1. Parent.

2. left and Right child.

3. Siblings

4. predecessor

5. descendant and ancestor

6. leaf

7. branch

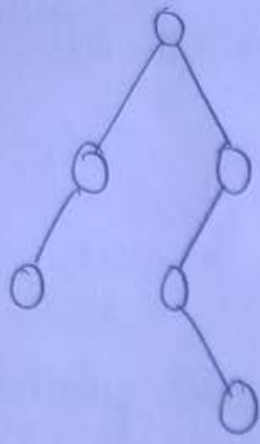
8. level number.

9. depth

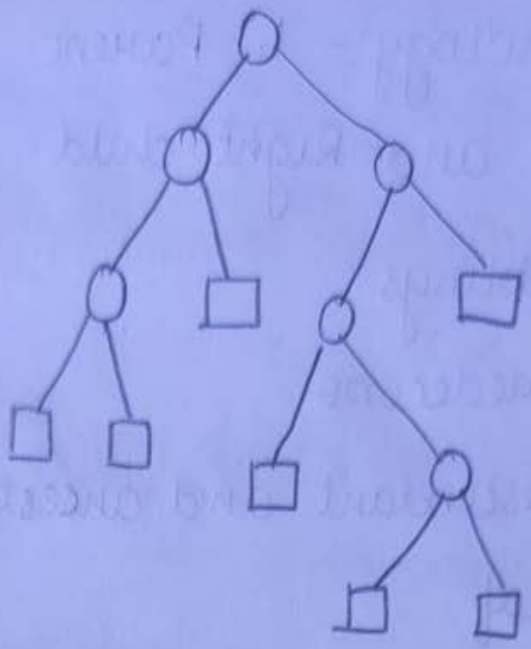
Complete Binary tree:- A tree T is said to be complete if all its levels, except possibly the last, have the maximum number of possible nodes, and if all the nodes at the last level appear as far left as possible.

$$\text{Depth } D = \lfloor \log_2 n + 1 \rfloor$$

Extended binary tree:- A binary tree T is said to be a 2-tree or an extended binary tree if each node N has either 0 or 2 children. the nodes with 2 children are internal nodes and the nodes with 0 child are external node



Binary tree



extended binary tree

Representation of Binary trees in memory.

→ using linked form

each node has three parts:

1. Info
2. left :- address of left child.
2. Right :- address of right child.

→ sequential representation :- This representation uses only single linear array T as follows.

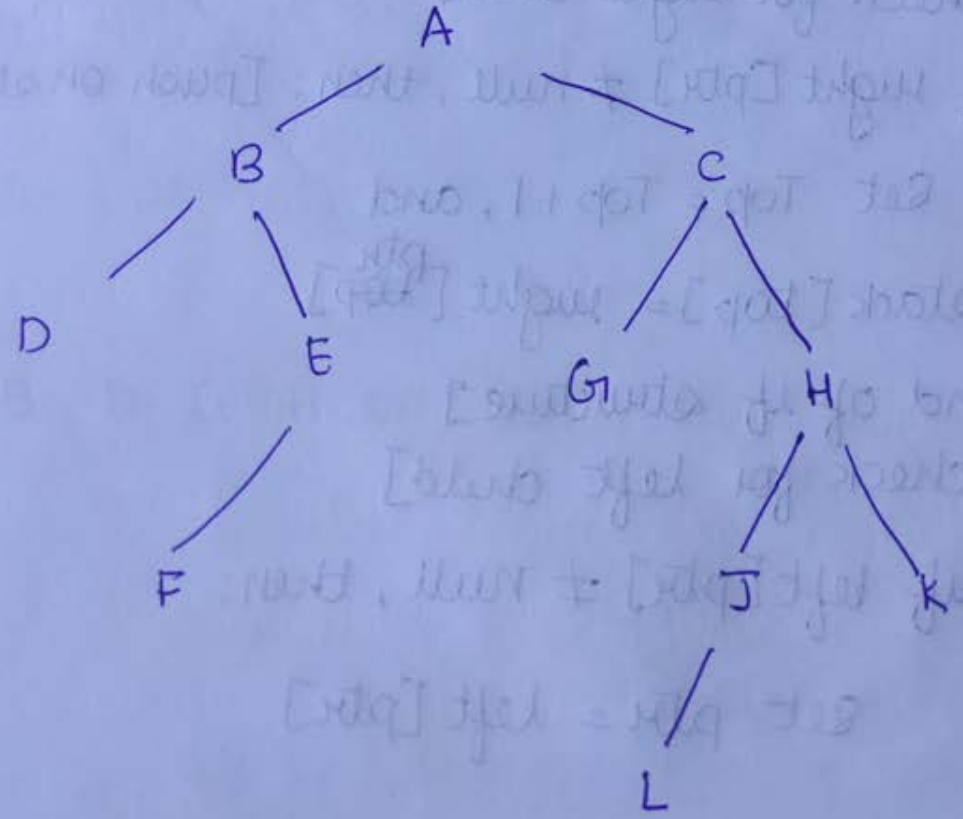
a.) The root R of T is stored in TREE[1].

b.) If node N occupies TREE[K], then left child is stored in TREE[2*K] and its right child is stored in TREE[2*K+1].

→ NULL is used to indicate empty subtree.

Traversing binary trees :-

1. Preorder (NLR)
   - → Process root R
   - → traverse left subtree
   - → traverse right subtree
2. Inorder (LNR)
   - → Traverse left subtree
   - → process root
   - → Traverse right subtree
3. Post order (LRN)
   - → Traverse left subtree
   - → Traverse right subtree
   - → Process root



Preorder: A B C D E F G H J K L

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

postorder :- D F E B G L J K H C A

Algo :- Preorder (Info, left, right, root)

Step 1:- Initialization

Step 2:- [Initially push NULL onto stack, and
initialize Ptr]

Set Top = 1, stack[1] = Null, and Ptr = root

Step 3:- Repeat step 4 to 6 while ptr ≠ null

Step 4:- Apply Process to Info[ptr]

Step 5:- [Check for right child]

if right[ptr] ≠ null, then: [push on stack]

Set Top = Top + 1, and

stack[top] = right[~~top~~ ptr]

[End of if structure]

Step 6:- [check for left child]

if left[ptr] ≠ null, then:

Set ptr = left[ptr]

Else: [Pop from stack]

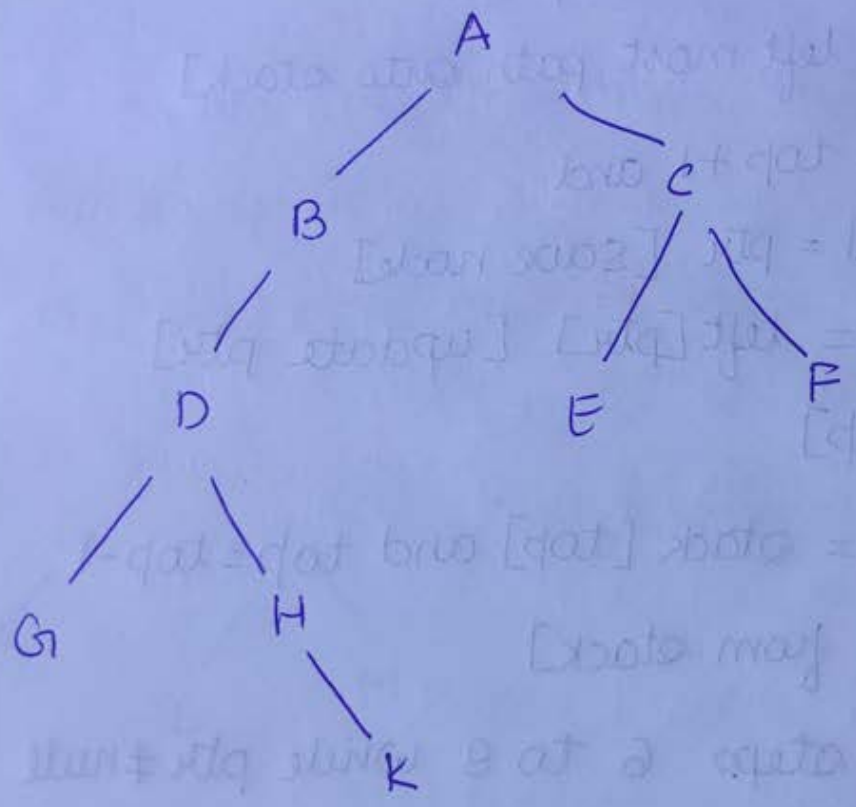
Set ptr = stack [top] and

top = top - 1

[End of if structure]

[End of step 3 loop]

Step 7:- Exit



A B D G H K C E F

Inorder Traversal :-

Algo :- Inorder (Info, left, Right, Root)

Step I :- Initialization

Step II :- [Push NULL onto stack and Initialize ptr]

Set Top = 1, stack[1] = null and ptr = Root

Step III :- Repeat while Ptr ≠ null

[pushes left most path onto stack]

a.) Set top = top + 1 and

Stack [top] = ptr [save node]

b.) Set ptr = left [ptr] [update ptr]

[End of loop]

Step IV :- Set ptr = stack [top] and top = top - 1

[Pops node from stack]

Step V :- Repeat steps 6 to 8 while ptr ≠ null

[Backtracking]

Step VI :- Apply Process to Info[ptr].

Step VII :- [Check for right child.]

if right [ptr] ≠ null, then

a.) Set ptr = right [ptr]

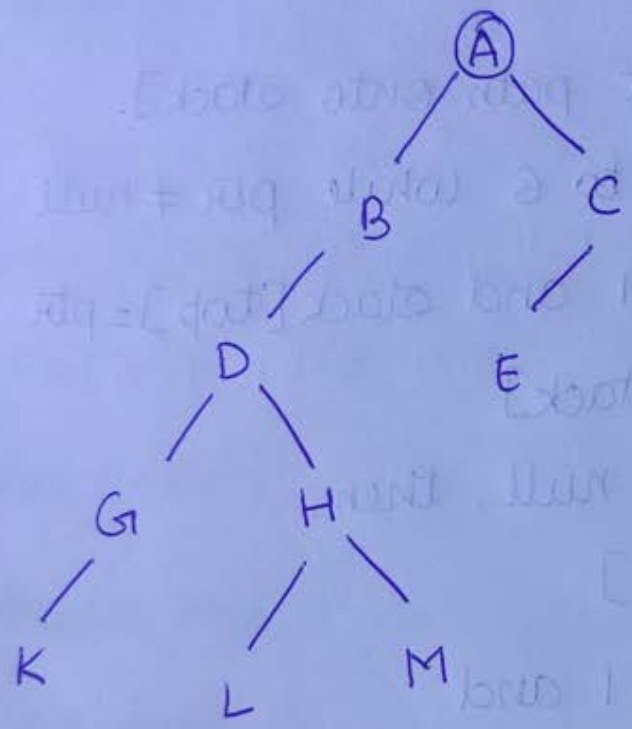
b.) go to step 4.

[End of if structure]

Step VIII :- Set ptr = stack [top] and
top = top - 1, [pops node]

[End of step 5 loop]

Step IX :- End.



Inorder : K G D L H M B A E C.

Post order traversal :-

Algo :- Postorder (Info, left, right, root)

Step I :- Initialization

Step II :- [push NULL onto STACK and initialize ptr]
Set top = 1, Stack[1] = null
and ptr = root

Step III :- [push left-most path onto stack].
Repeat steps 4 to 6 while ptr ≠ null.

Step IV :- Set top = top + 1 and stack[top] = ptr
[push ptr on stack]

Step V :- if right[ptr] ≠ null, then:
[push on stack]
Set top = top + 1 and
stack[top] = - right[ptr].
[End of if structure]

Step VI :- Set ptr = left[ptr]. [update ptr]
[End of loop, step 3]

Step VII :- Set ptr = stack[top]
and top = top - 1
[pops node from stack]

Step VIII:- 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].

Step IX:- if ptr < 0, then:

a.) set ptr = - ptr.

b.) Go to step 3

[End of if structure].

Step X :- Exit

postorder:- K G L M H D B E C A

Header node:- A header node is added to the beginning of T. The left pointer of the header node will point to the root of T and right pointer has a null pointer.

left [head] = null

it indicates an empty tree.

as null pointer contains any invalid address,

instead of that, the subtree will contain the header node's address. It means.

left [head] = head

it is also indicate an empty tree

Threads and inorder threading :- About half of the entries in the left and right pointer contain null. It will replace with certain null entries by special pointers which point to nodes higher in the tree.. These special pointers are called threads and binary trees with such pointers are called Threaded trees.

Binary Search Trees:-

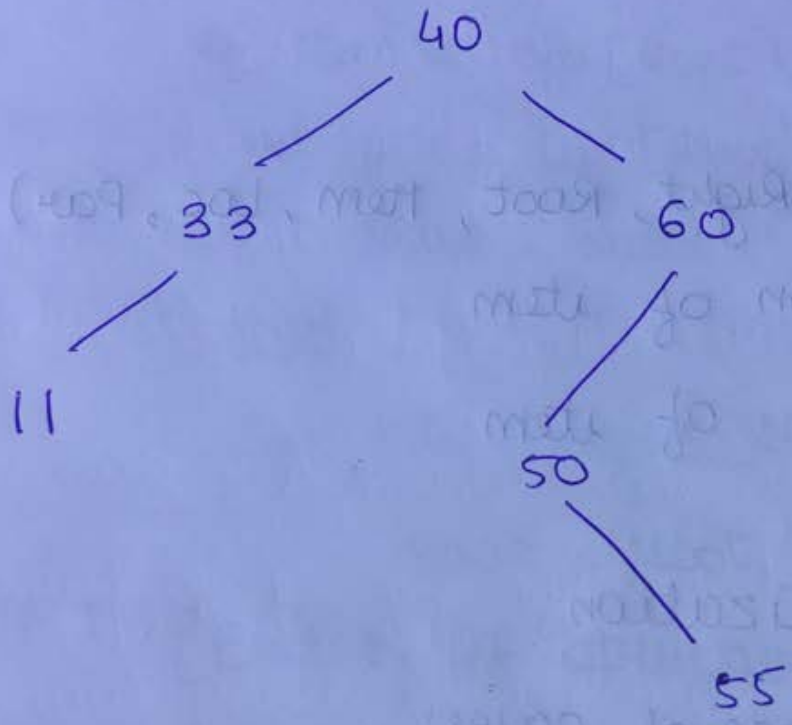
Suppose T is a binary tree and N be a node of this tree. Then each node must follow a property:

a.) Left subtree < N

b.) N < right subtree

Ex:- 40, 60, 50, 33, 55, 11

Ans.



Searching and inserting in BST:-

a.) Compare Item with the root node N of tree:

i.) if Item < N, proceed to left child of N.

ii) if Item > N, proceed to right child of N.

b.) Repeat step (a) until:

i) Item = N, the search is succesful

ii) if empty tree meets, the search is unsuccesful.

Algo:-

Find (Info, left, Right, Root, Item, Loc, Par)

Loc = location of item

Par = parent of item.

~~Special~~

Step I:- Initialization

three special cases:

i.) Loc = null and par = null ⇒ empty tree

ii.) Loc ≠ null and par = null ⇒ Item is at root

iii) Loc = null and par ≠ null ⇒ item is not in tree. and can be added as a child of node N.

Step II:- [Tree empty?]

if root = null, then:

Set Loc = null and Par = null and

Return

Step III:- [Item at root?]

if Item = Info[Root], then Set Loc = Root

and Par = null and Return.

Step IV:- [Initialize pointer Ptr and Save]

if Item < Info[Root], then ~~and save = root~~

Set ptr = left[Root] and

Save = root

else

Set ptr = right[Root] and

Save = root

[End of if structure]

Step V:- Repeat step 6 and 7 while ptr ≠ null:

Step VI:- [Item found]

if item = Info[ptr], then

Set loc = ptr and par = save

and return

Step 7:- 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 5 loop]

Step 8:- [Search unsuccessful]
Set loc = null and par = save

Step 9:- Exit

Insert in BST:-

Algo:- Insert BST(Info, left, Right, Root, Avail, Item, loc)

Step I:- Initialization

Step II:- Call find(info, left, right, root, Item, loc, par)

Step III:- if loc ≠ null, then exit

Step IV:- [Copy item into new node in Avail list]

a.) if avail = null, then write: Overflow and exit

b.) Set new = avail, avail = left [avail] and info [new] = item.

c.) Set loc = new, left [new] = null and right [new] = null.

Step 5 :- [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 6 :- Exit

Complexity :- $f(n) = O(\log_2 n)$

Deleting in a Binary Search Tree:-

Let N be the node which have to be deleted. The way N is deleted from the tree depends primarily on the number of children of node N. There are three cases:

Case 1: N has no child. Then N is deleted from tree 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 the only child of N.

Case 3: N has two children. Let S(N) denote the inorder successor of N (S(N) does not have a left child). Then N is deleted from T by first deleting S(N) from T and then replacing node N in T by the node S(N).

1. Procedure to delete node which ~~ties~~ belongs to case 1 and case 2

this procedure is adopted when node N has no child or only one child.

Algo:- CaseA(Info, Left, Right, Root, loc, Par)

Step I:- Initialization

Step II:- [Initialize Child]

if Left[loc] = null and Right[loc] = null, then : Child = null

Else if left[loc] ≠ null, then:

Set Child = left[loc].

Else

Set Child = Right[loc].

[End of if structure]

Step III:- if Par ≠ null, then:

if loc = left[par], then:

Set left[par] = Child

Else:

Set right[par] = Child.

[End of if structure]

Else:

Root = child

[End of if structure]

Step 4:- Return.

Algo

2. Procedure to delete node with Case 3:-

The node with 2 children.

Algo:- Case B (Info, left, right, Root, loc, Par)

Step 1:- Initialization

Step 2:- [Find Suc and ParSuc]

a.) Set Ptr = Right [loc] and Save = loc.

b.) Repeat while left [ptr] ≠ null

Set Save = ptr and

ptr = left [ptr].

[End of loop]

c.) Set Suc = ptr and parsuc = Save.

Step 3:- [Delete Inorder Successor]

call CaseA (Info, left, right, root, suc, parsuc)

Step 4:- [Replace node N by its inorder successor]

a.) if Par ≠ null, then:

if loc = left[par], then:

Set left[par] = suc.

Else:

Set right[par] = suc.

[End of if structure]

Else

Set Root = suc.

[End of if structure]

b.) Set left[suc] = left[loc] and

right[suc] = right[loc]

Step 5:- Return

3. Complete Algo to delete Node N from tree.

Algo:- Del (Info, left, right, root, avail, item)

Step 1:- Initialization

Step 2:- [Find the location of item and its parents]

Call Find(Info, left, right, root, item, loc, par)

Step 3:- [item not found]

if loc = null, then:

Write: Item is not in tree and exit.

Step 4: [Delete node containing info = Item]

if right[loc] ≠ null and left[loc] ≠ null, then:

call caseB(Info, left, right, root, loc, par)

Else

call caseA(Info, left, right, root, loc, par)

[End of if structure]

Step 5:- [Return deleted node to avail list]

Set left[loc] = avail and avail = loc.

Step 6:- Exit

Balanced Binary tree :-

Balanced binary tree is a tree with a balance factor, which is the difference in height ~~of~~ between the left and right subtrees. The value of balance factor is:

$$B = H_L - H_R$$

H_L = height of left subtree

H_R = height of right subtree

Height - balanced tree :- A height - balanced binary tree has left and right subtrees which differ in height by not more than one. In height balanced binary tree, the operation which run in time proportionate to the trees height are O(logn)

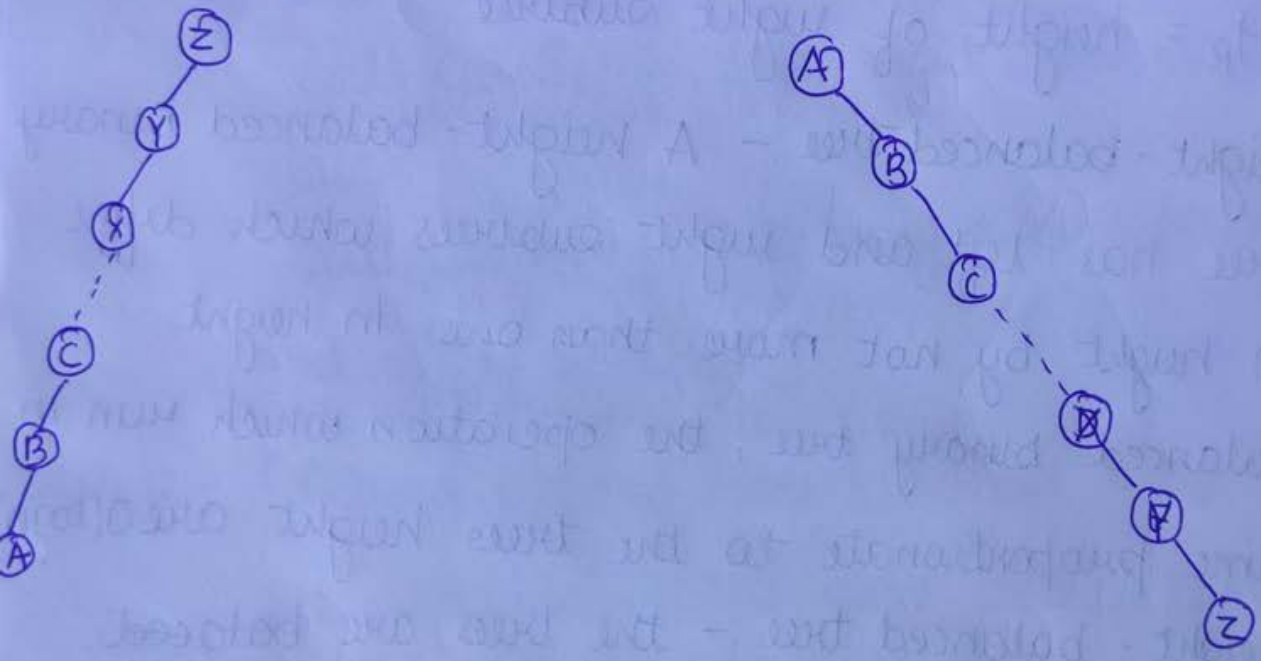
Weight - balanced tree :- the trees are ~~balaced~~ balanced in order to keep the sizes of the subtrees of all the nodes within a constant factor of all the present nodes. Discrete set or discrete maps are implemented using weight-balanced binary tree.

AVL tree :- Also known as AVL search tree.

→ Named after inventors Adelson - Velsky - Landis

→ AVL tree is a self- balancing binary search tree where the difference between heights of left and right subtrees cannot be more than one for all nodes.

left skewed and right skewed BST :-



A non empty binary tree T is an AVL tree iff given T^L and T^R to be the left and right subtrees of T and $h(T^L)$ and $h(T^R)$ to be the heights of subtrees T^L and T^R respectively, T^L and T^R are AVL trees and $|h(T^L) - h(T^R)| \leq 1$

→ $h(T^L) - h(T^R)$ is known as the balanced factor (BF) and for an AVL tree the balance factor of a node can be either 0, 1, or -1.

Insertion in an AVL search tree:-

→ first phase of inserting a node in AVL tree is similar to normal BST.

→ if after insertion of the element, the balance factor of any node in the tree is affected so as to render the binary search tree unbalanced, a techniques called Rotations are used to restore the balance of the search tree.

1. LL rotation
2. RR rotation
3. LR rotation
4. RL rotation

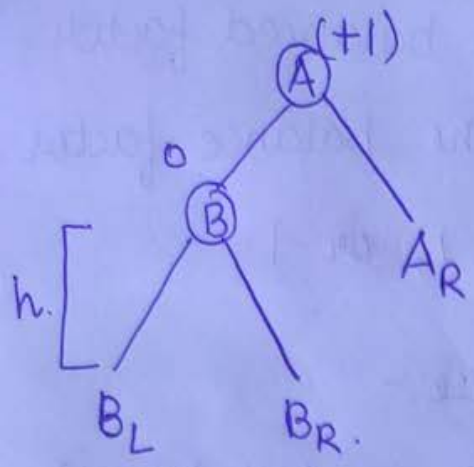
LL rotation:- Inserted node is in the left subtree of left subtree of node A.

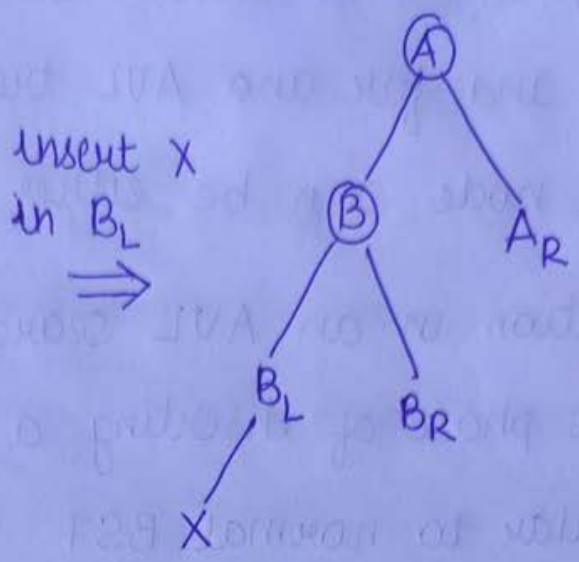
B_L: left subtree of B

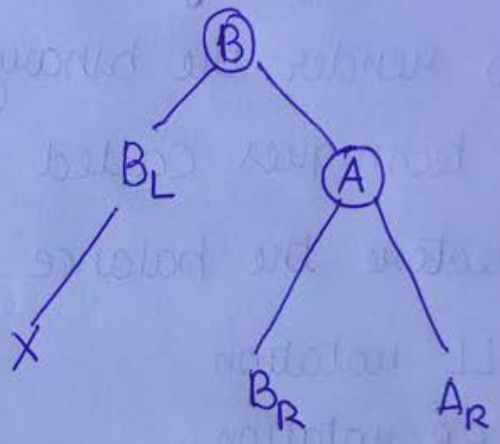
B_R: Right subtree of B

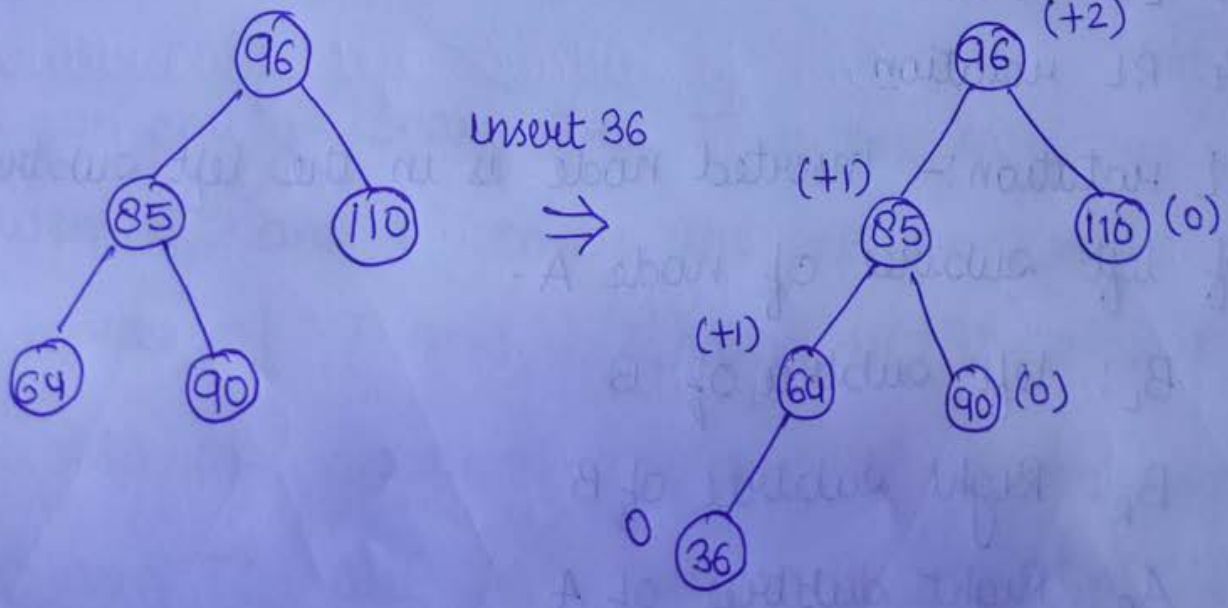
A_R: Right subtree of A

h : Height

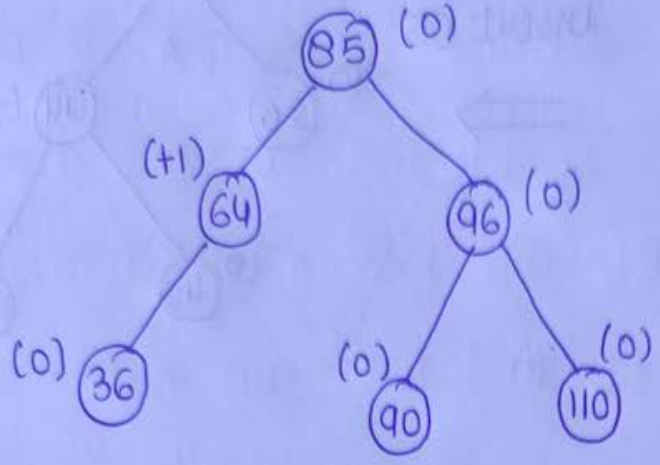

(+1)
A
0
B
h.
A_R
B_L
B_R.


insert X
in B_L
A
B
A_R
B_L
B_R
X

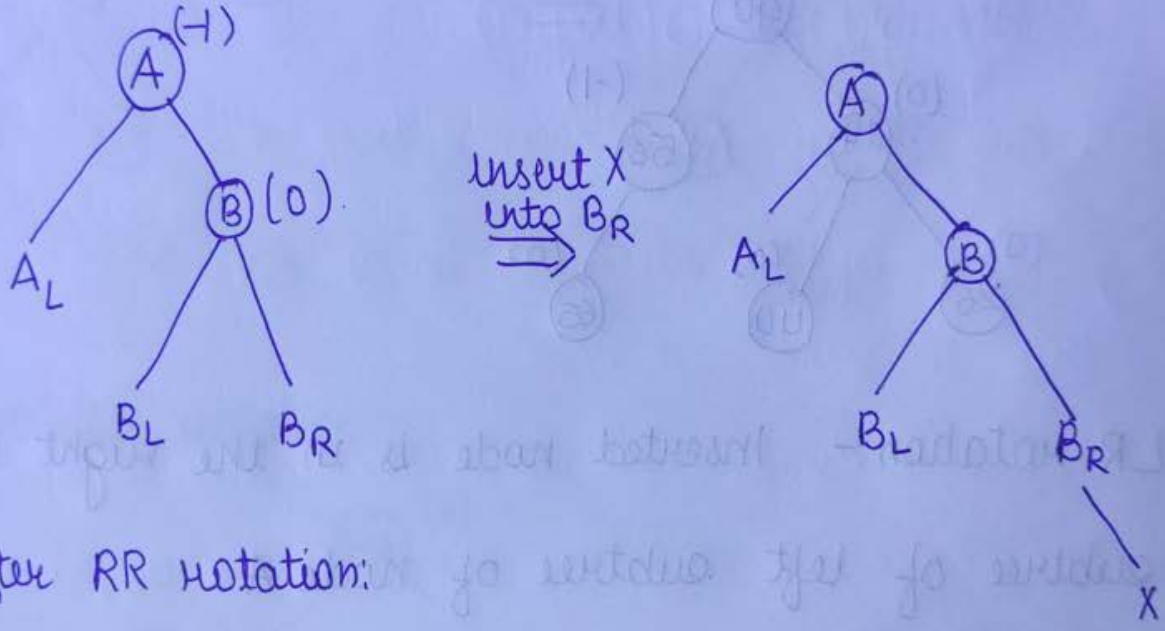

B
B_L
A
X
B_R
A_R


96
85
110
64
90
insert 36
(+2)
96
(+1)
85
110
(0)
(+1)
64
90
(0)
0
36

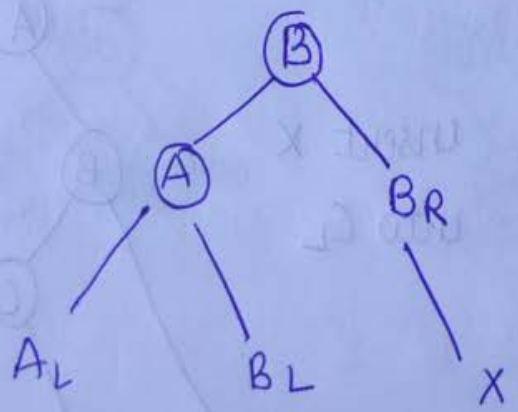
after LL rotation

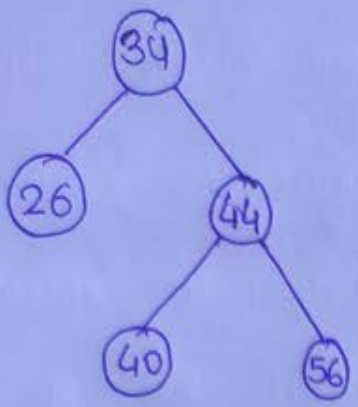


RR rotation:- Inserted node is in the right subtree of right subtree of node A.

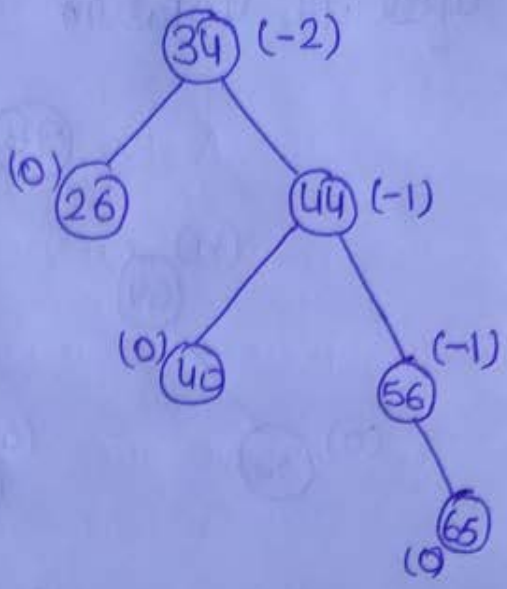


After RR rotation:

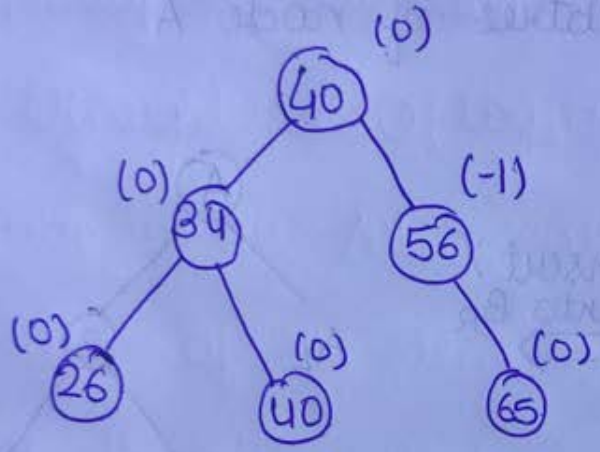




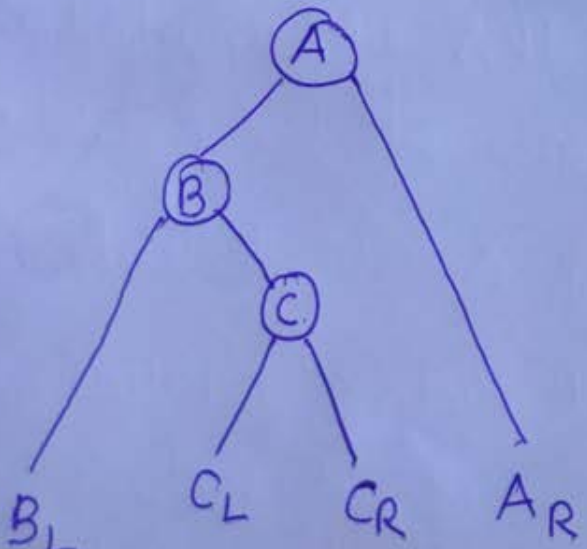
Insert 65 ⟹



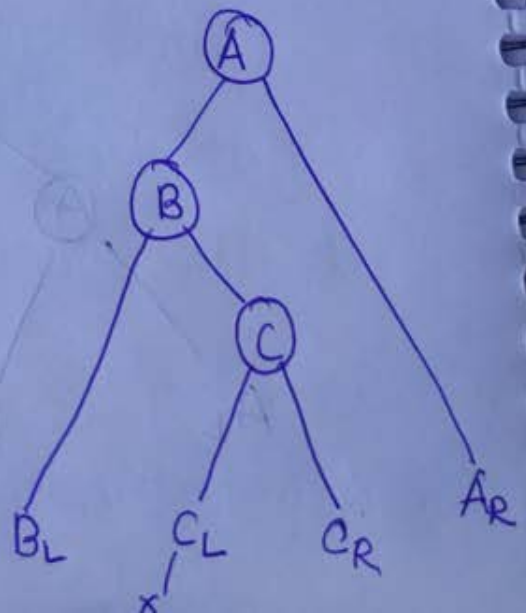
After RR rotation



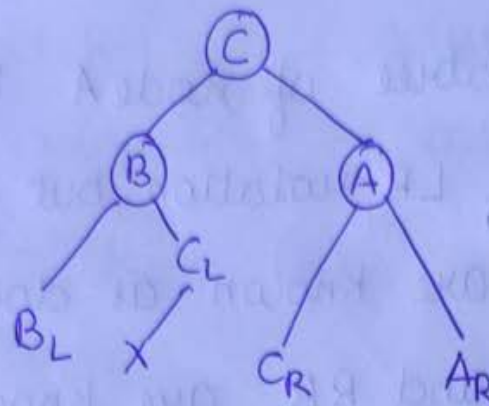
LR rotation:- Inserted node is in the right subtree of left subtree of node A.

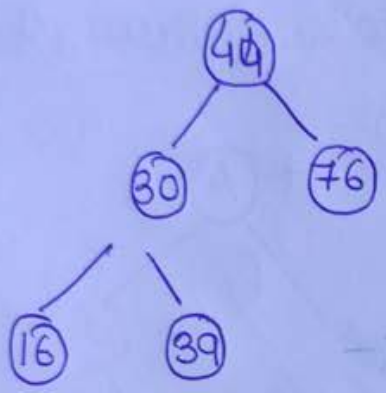


Insert X into C_L

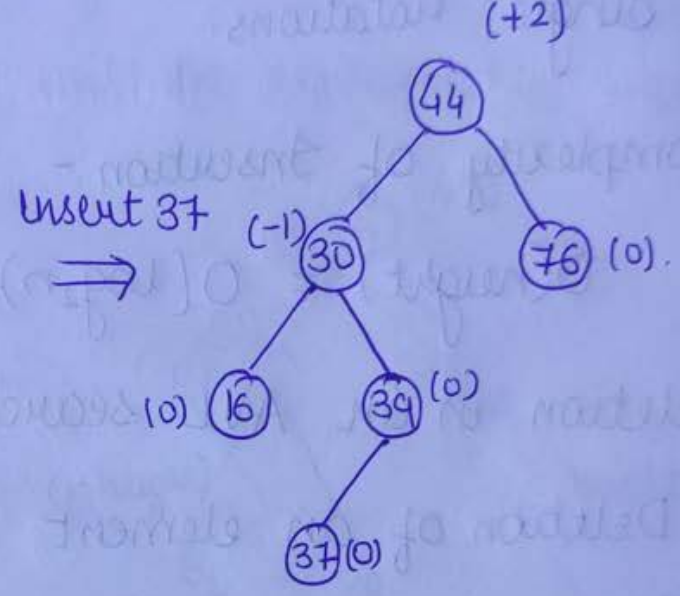


After LR rotation

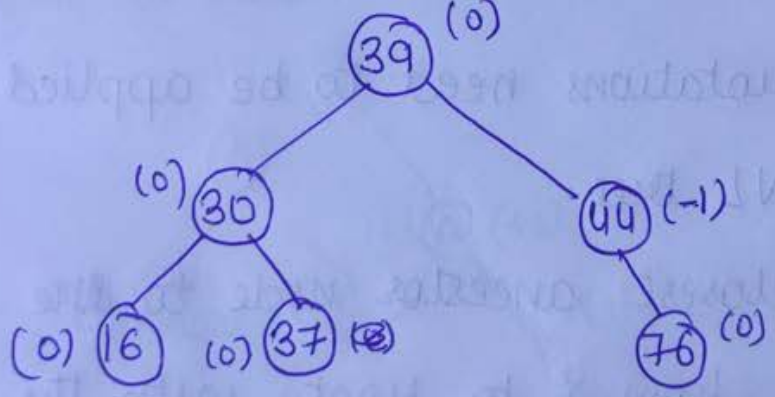




Insert 37 ⟹



After LR rotation:-



RL Rotation:- Inserted node is in the left subtree of right subtree of node A. It is similar in nature of LR rotation but a mirror image. LR and RL are known as double rotations, and LL and RR are known as single rotations.

Complexity of Insertion:-

$$O(\text{height}) = O(\log_2 n)$$

Deletion in an AVL search tree:-

→ Deletion of an element in an AVL search tree is same as BST.

→ An imbalance can be occured due to deletion, one or more rotations need to be applied to balance the AVL tree.

→ let A be the closest ancestor node to the deleted node X from X to root, with the balance factor +2 or -2.

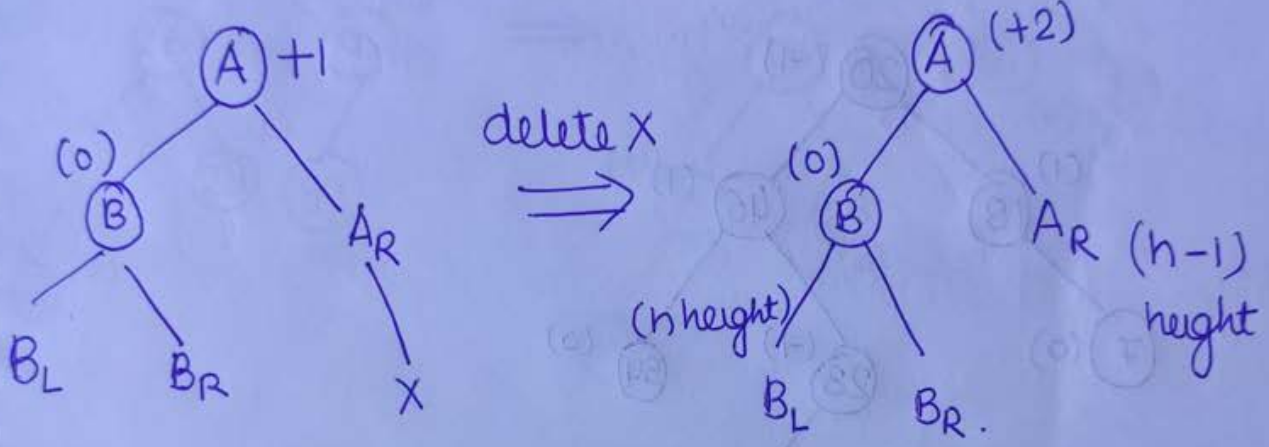
→ Restore the BF by rotation is classified as left or right subtree.

→ the R or L unbalance is further classified as R0, R1 and R-1 or L0, L1 and L-1.

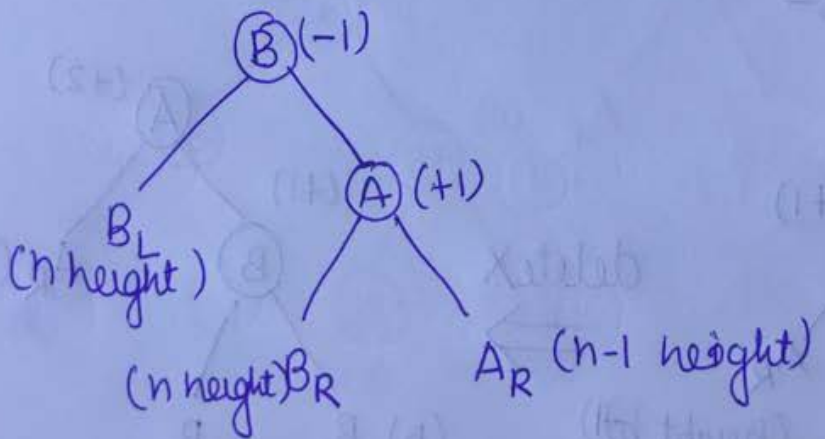
→ The L rotations are mirror images of these R rotation.

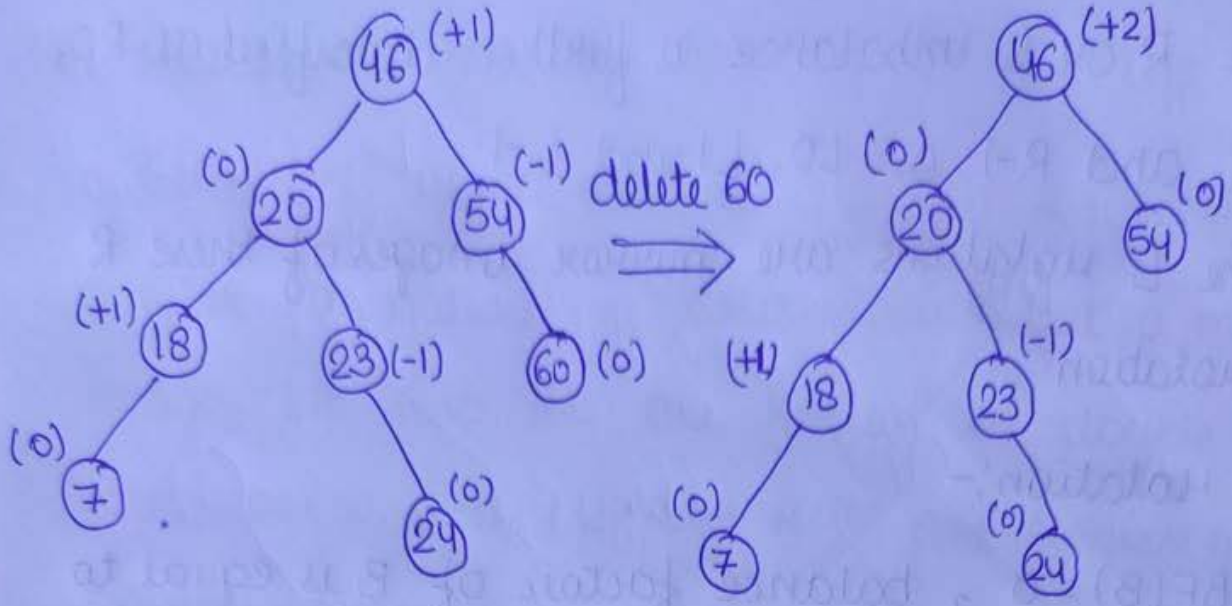
R0 Rotation:-

if BF(B)=0, balance factor of B is equal to 0 then rotation R0 will be performed:

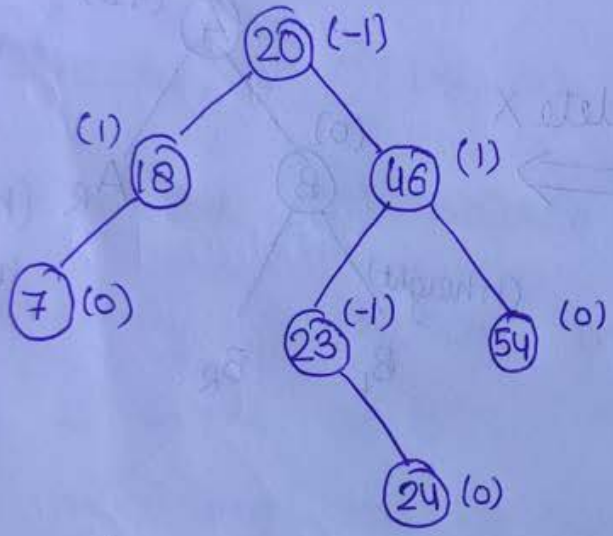


After R0 rotation



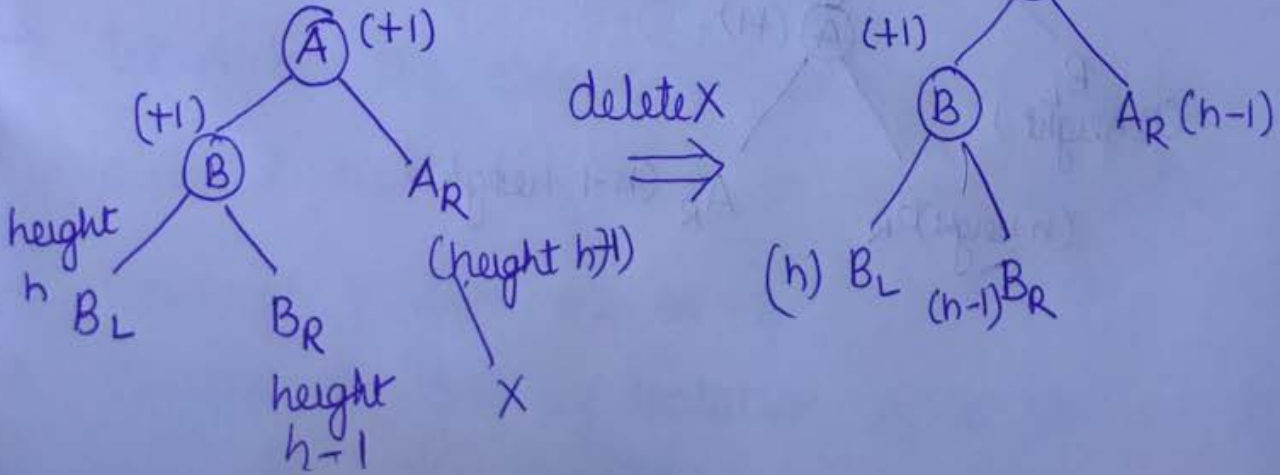


after ~~deletion~~ RO rotation

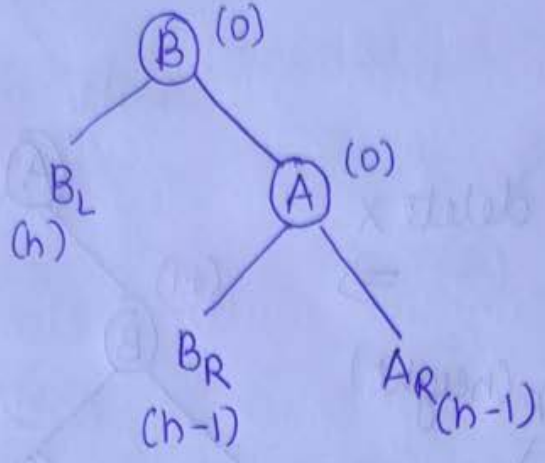


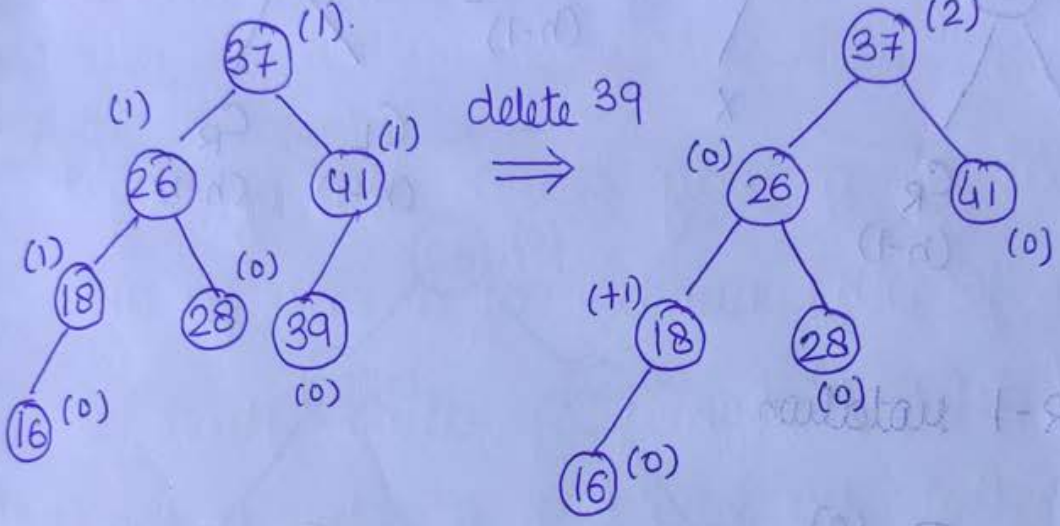
R1 rotation:-

If BF(B) = 1

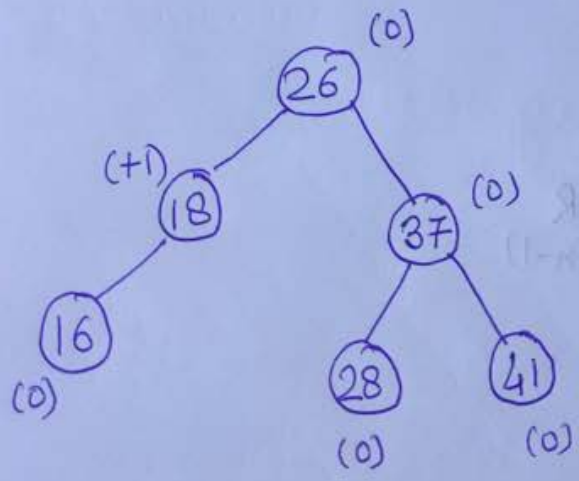


after R1 rotation



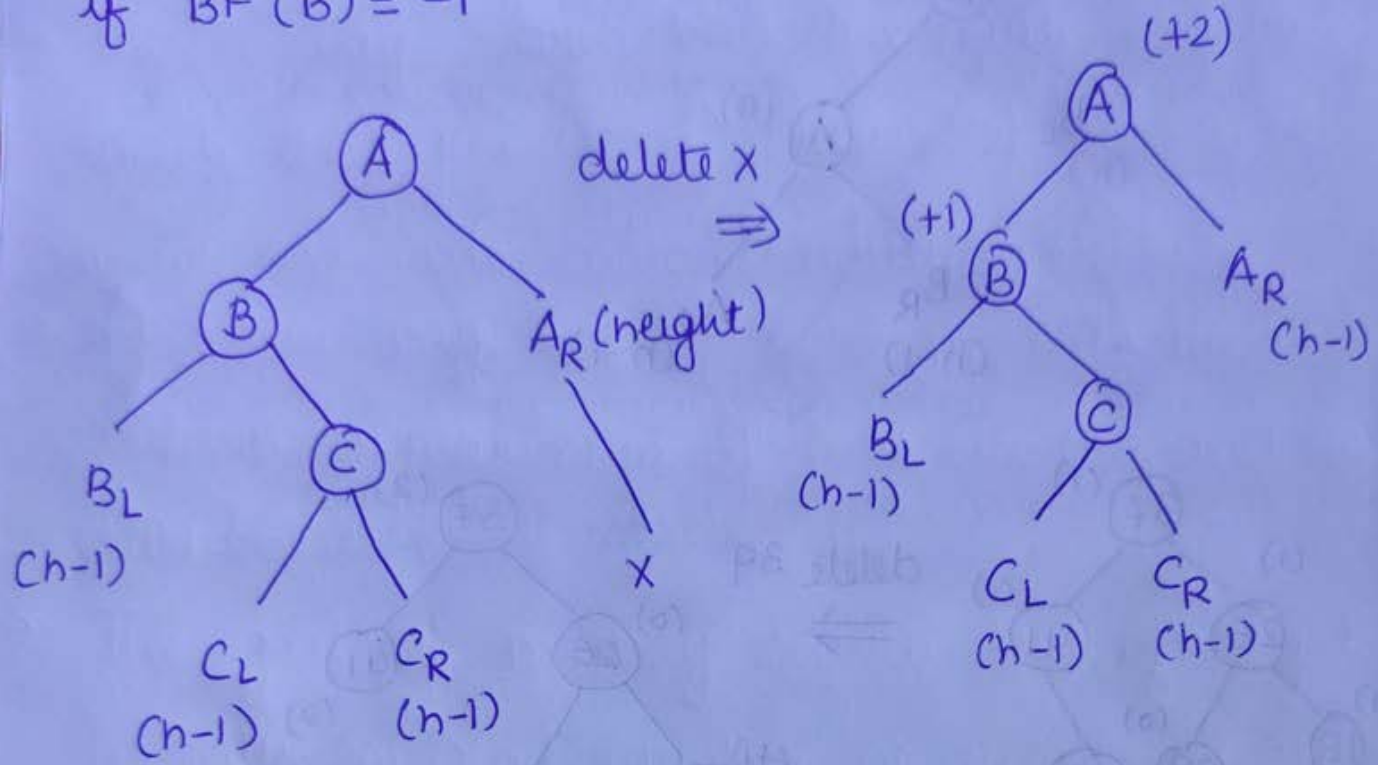


After R1 rotation :-

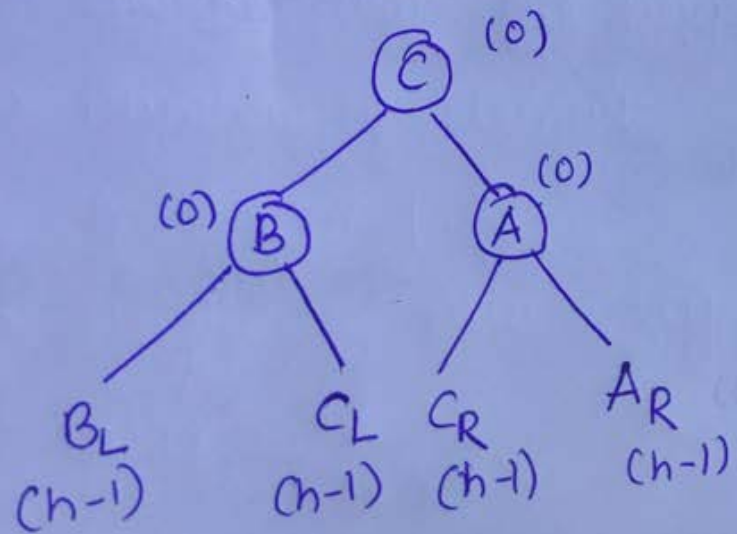


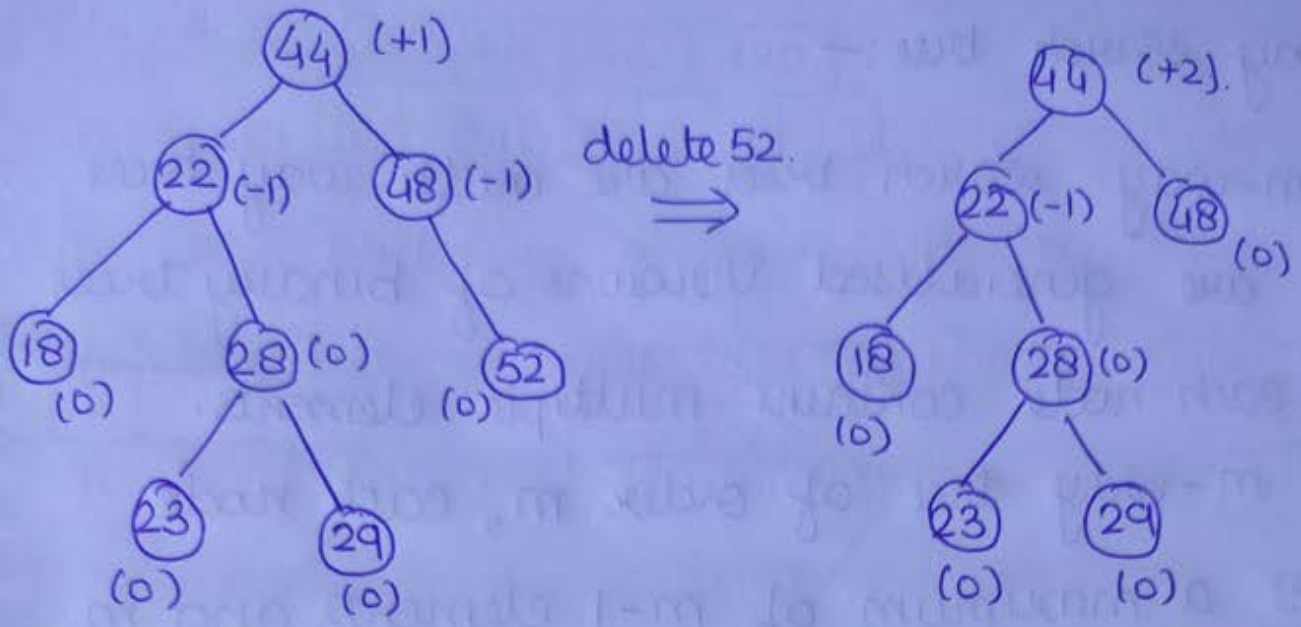
R-1 Rotation :-

if BF (B) = -1

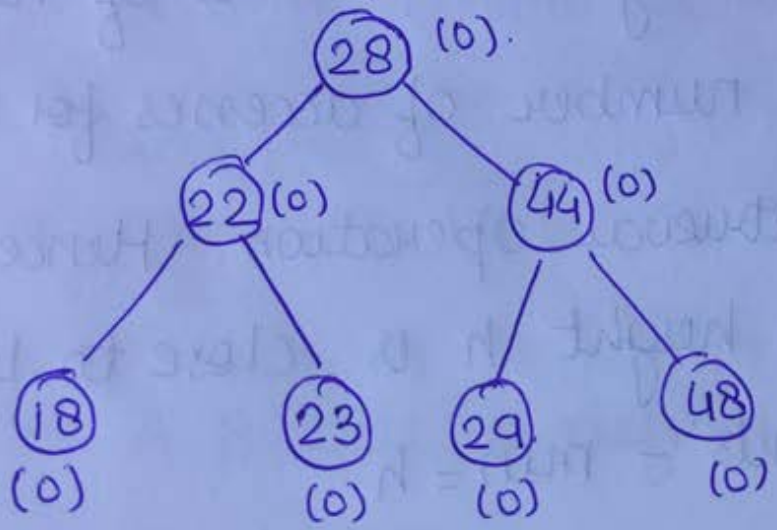


After R-1 rotation





After R-1 rotation :-



m- way search tree :-

→ The m-way search trees are multi-way trees which are generalised versions of binary trees where each node contains multiple elements.

→ In an m-way tree of order m, each node contains a maximum of m-1 elements and m children.

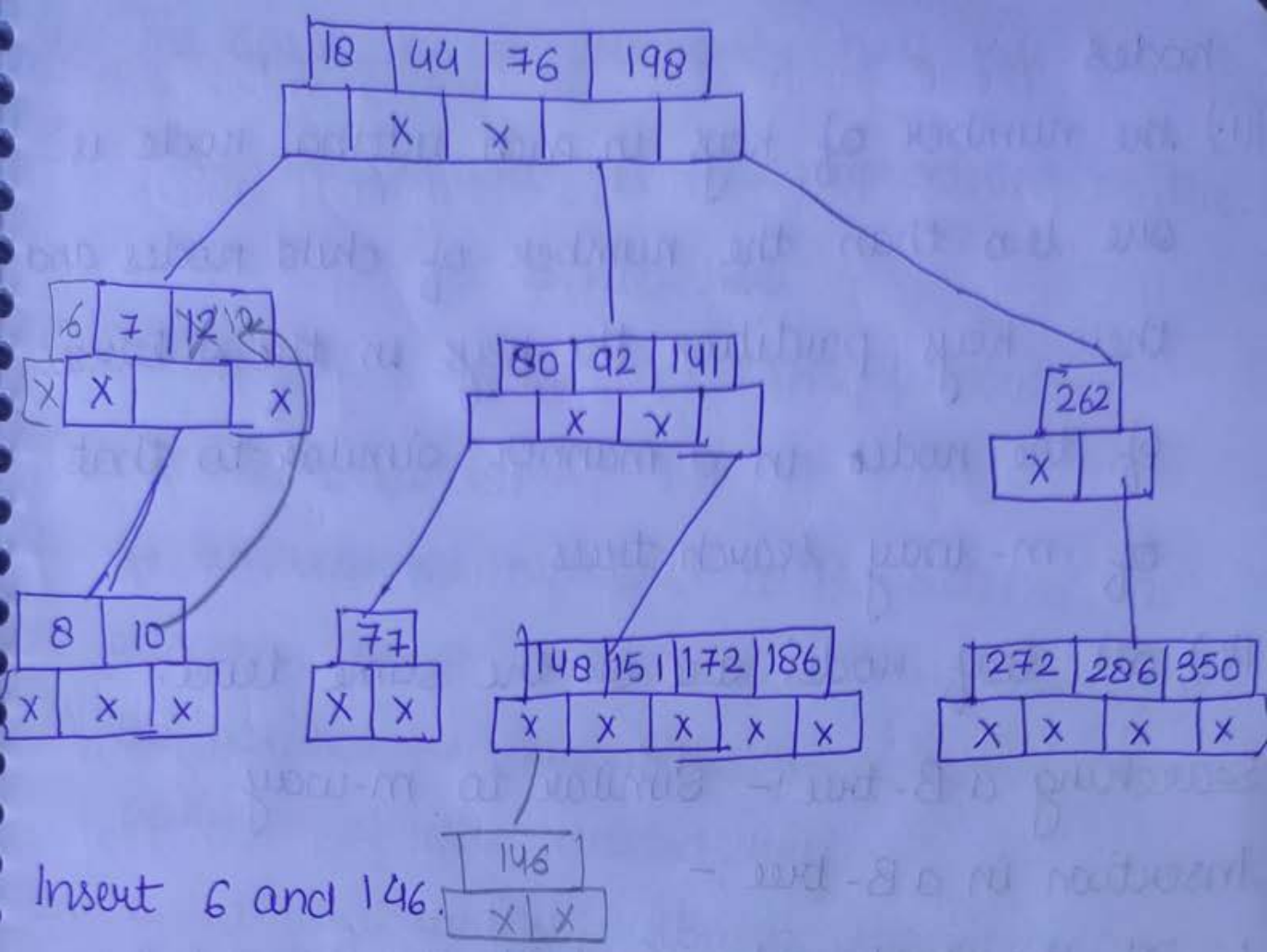
→ The goal of m-way search tree of height h calls for O(h) number of accesses for an insert / delete / retrieval operation. Hence, it ensures that the height h is close to $\log_m(n+1)$

→ number of elements :- min = h

max = $m^h - 1$.

→ Height :- min = $\log_m(n+1)$

max = n.



Insert 6 and 146.

delete 12

B-tree:- A B-tree is defined as a balanced m-way search tree:

Definition:- A B-tree of order m, if non empty, is an m-way search tree in which:

i.) The root has at least two child nodes and at most m child nodes.

i.) the internal nodes except the root have at least $\lceil \frac{m}{2} \rceil$ child nodes and at most m child

nodes.

iii.) the number of keys in each internal node is one less than the number of child nodes and these keys partition the keys in the subtrees of the nodes in a manner similar to that of m-way search trees.

iv.) all leaf nodes are on the same level.

Searching a B-tree :- Similar to m-way.

Insertion in a B-tree :-

1. If the leaf node in which the key is to be inserted is not full, then insertion is done in the node.

→ A node is said to be full if it contains (m-1) key or maximum keys.

2. If the node is already full, then the node is splited at its median into two nodes at the same level.

3. the median element is pushed up by one level.

4. The split nodes are only half full.
5. Median element is accommodated in the parent if it is not full.
6. Otherwise, repeat the same procedure.

Deletion in a B-tree:-

1. If an internal node is to be deleted, a successor or a predecessor of the key to be deleted is promoted to occupy the position of the deleted key.
2. If a key, which is from a leaf node and contains more than the minimum number of elements, then key can be easily deleted
3. If the leaf node contains just the minimum number of elements, then search for an element from either the left sibling node or right sibling node to fill the vacancy.
4. If the left sibling node has more than the minimum number of keys, pull the largest key up into the parent node.

5. Now, move down the intervening entry from the parent node to the leaf node where the key is to be deleted.
6. Otherwise, pull the smallest key of the right subling node to the parent node and move down the intervening parent element to the leaf node.
7. if both the subling nodes have only min number of entries, then create a new leaf node out of the two leaf nodes.
8. The intervening element of the parent node, ~~that~~ ensures that the total number does not exceed the maximum limit for a node.
9. the procedure propagates upwards ultimately reducing the height of B-tree.

B+ tree:- The difference from B-tree is that the nodes of a B+ tree can point to many children nodes instead of being limited to only two.

Definition:-

1. From the root to the leaf all paths should be of the same length.
2. When a node is not either a root or leaf then it has children between $\lceil n/2 \rceil$ and n.
3. Special condition:
   a.) When a root is not a leaf, it has a minimum of 2 children.
   b.) When a root is a leaf, then it can have children between 0 and n-1 values

Searching:- Same as B-tree

Insertion:- Same as B-tree and include the properties of B+ tree.

Deletion:- Same as B-tree and should follow the properties of B+ tree.

Heap:- Suppose H is a complete binary tree with n elements. H is called a heap (max heap), if each node N of H has the following properties:

1. The value at N is greater than or equal to the value at each of the children of N.

→ A min heap is defined as: the value at N is less than or equal to the value at any of the children of N.

Insertion into a Heap:- To insert an item into the heap H as follows:

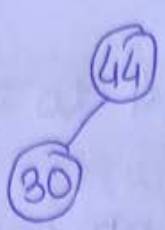
1. First adjoin item at the end of H so that H is still a complete tree, but not necessarily a heap.
2. Then let item rise to its appropriate place in H so that H is finally a heap.

Ex:- Build a heap H from the following elements:

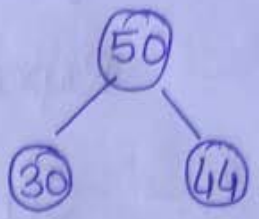
44, 30, 50, 22, 60, 55, 77, 55

a.) Item = 44

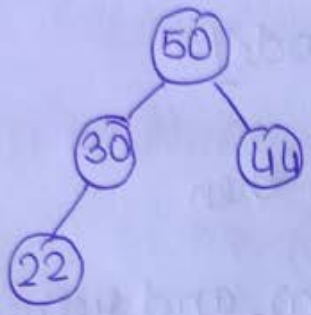
b.) Item = 30



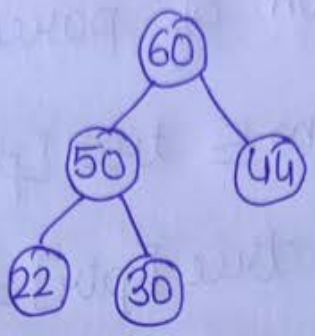
c.) Item = 50



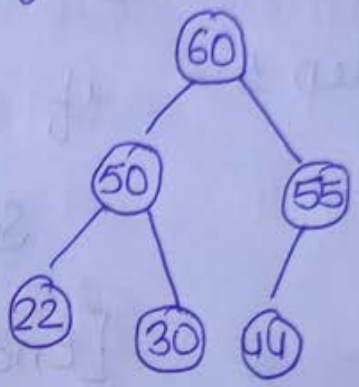
d.) Item = 22



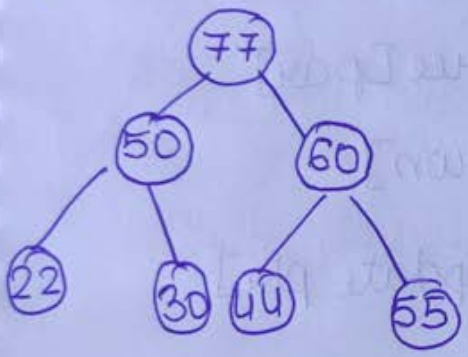
e.) Item = 60



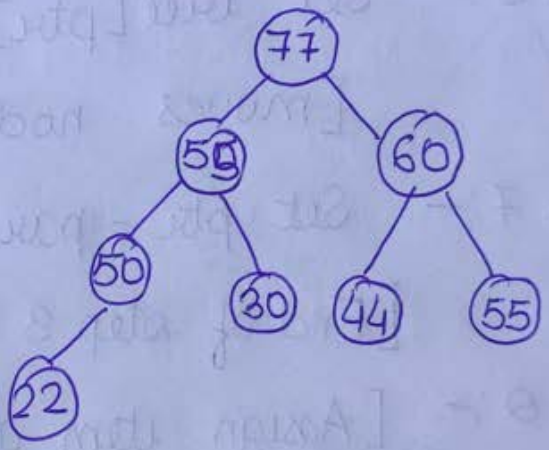
f.) Item = 55



g.) Item = 77



h.) Item = 55



Insertheap (tree, N, item)

Step 1 :- Initialization

Step 2 :- Add new node to H and initialize ptr

Set N = N+1 and ptr = N

Step 3:- [find location to insert item]
Repeat Step 4 to 7 while ptr > 1

Step 4:- Set par = L ptr/2].
[location of parent node]

Step 5:- if item ≤ tree [par], then
Set tree [ptr] = item, and return
[End of if structure]

Step 6:- Set tree [ptr] = tree [par]
[moves node down]

Step 7:- Set ptr = par. [update ptr]
[End of step 3 loop]

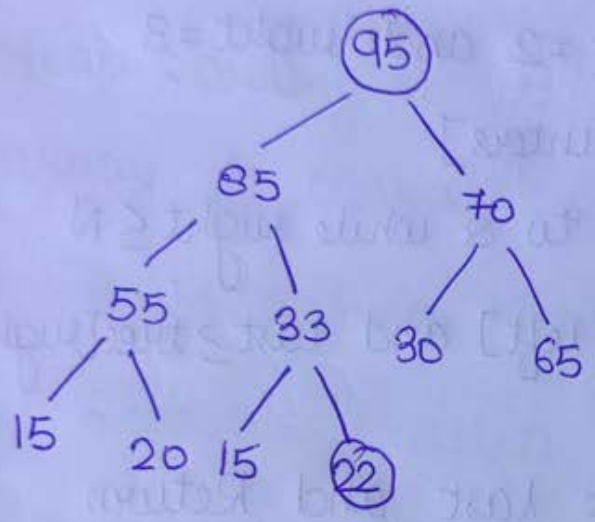
Step 8:- [Assign item as the root of H.]
Set tree [1] = item

Step 9:- Return

Delete the root of a heap:-

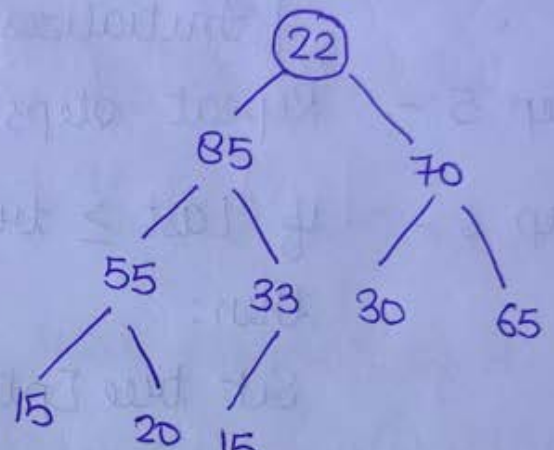
Delete the root of H, is accomplished as follows:

1. Assign the root R to some variable Item.
2. Replace the deleted node R by the last node L of H so that H is still a complete tree, but not necessarily a heap.
3. (Reheap) Let L sink to its appropriate place in H so that H is finally a heap.

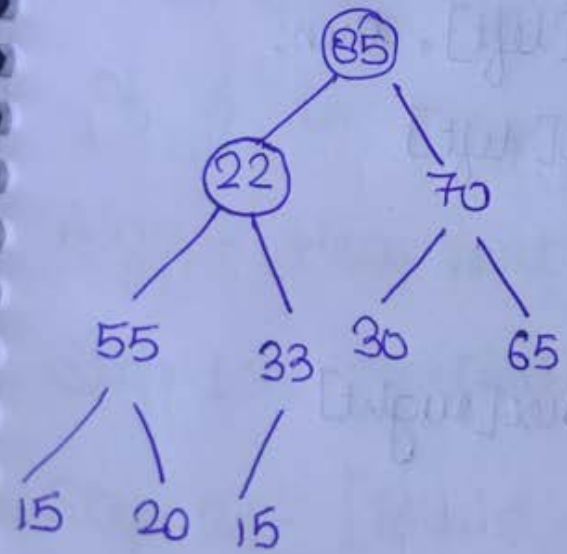


a.)

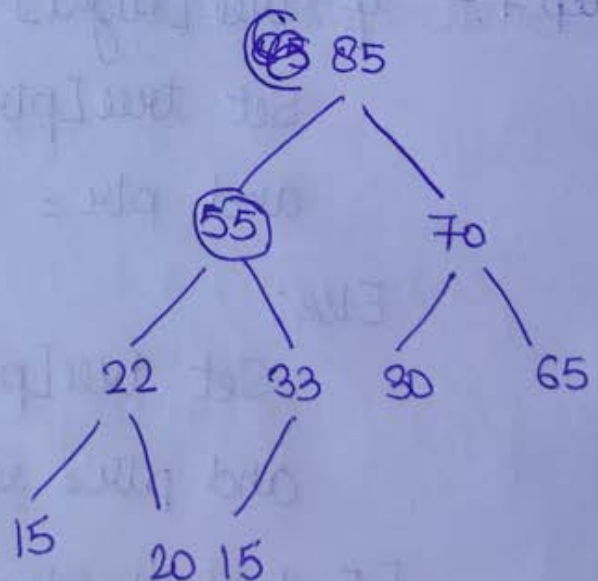
delete 95



(b.)



(c)



(d)

Deleteheap (tree, N, item)

Step 1:- Initialization

Step 2:- Set item = tree[1]
[Removes root of H]

Step 3:- Set last = tree[N]
and N = N-1 [Removes last node of H]

Step 4:- Set ptr = 1, left = 2 and right = 3
[Initializes pointers]

Step 5:- Repeat steps 6 to 8 while right ≤ N

Step 6:- if last ≥ tree[left] and last ≥ tree[right], then:
Set tree[ptr] = last and Return
[End of if structure]

Step 7:- if tree[right] ≤ tree[left], then:
Set tree[ptr] = tree[left]
and ptr = left
Else:
Set tree[ptr] = tree[right]
and ptr = right
[End of if structure]

Step 8:- Set left = 2* ptr and right = left + 1

[End of step 5 loop]

Step 9:- if left = N and if last < tree[left],

then: Set ptr = left

Step 10:- Set tree[ptr] = last

Step 11:- Return

Heap Sorting :- The heapsort algorithm to sort array A consists of the two steps:

1. Build a heap H out of the elements of A.
2. Repeatedly delete the root element of H.

→ Since, the root of H always contains the largest node in H, hence, step-2 deletes the elements of A in decreasing order.

Algo:- Heapsort (A, N)

Step 1:- Initialization

Step 2:- [Build a heap]

Repeat for J = 1 to N-1

Call Insertheap (A, J, A[J+1])

[End of loop]

Step 3:- [Sort A by repeatedly deleting the root of H, which is the largest number in heap]

Repeat while N>1:

a.) Call deleteheap (A, N, Item)

b.) Set A[N+1] = Item

[End of loop]

Step 4:- Exit

Complexity of Heap sort:-

Step 1:- $g(n) \leq n \log_2 n$

Step 2:- $h(n) \leq 4n \log_2 n$.

$f(n) = g(n) + h(n)$

$f(n) = O(n \log_2 n)$

Graph

→ Like tree, graph is also a non-linear data structure.

→ A graph G consists of two things:

1. A set V of elements called nodes (vertices)
2. A set E of edges such that each edge e in E is identified with a unique (unordered) pair [u,v] of nodes in V, denoted by e=[u,v].

neighbours:- For any edge e = [u,v], the nodes u and v are called the end points of e, and u and v are said to be adjacent nodes or neighbors.

Degree :- The degree of a node u, also written as deg(u), is the number of edges contains u.

Isolated node :- If deg(u)=0, that is, if u does not belong to any edge, then u is called as isolated node.

Path :- A path P of length n from a node u to a node v is defined as a sequence of n+1 nodes.

$$P = (v_0, v_1, v_2, \ldots, v_n)$$

such that v_0 is first node (u), v_n is last node (v) and v_{i-1} and v_i is adjacent.

→ The path P is said to be closed if $v_0 = v_n$.

→ The path P is said to be simple if all the nodes are distinct.

→ A cycle is closed simple path with length 3 or more. A cycle of length k is called a k-cycle.

→ A graph G is said to be connected if there is a path between any two of its nodes.

→ A graph G is connected if and only if there is a simple path between any two nodes in G.

Complete:- A graph G is said to be complete if every node u in G is adjacent to every other node v in G.

→ Also a connected graph.

→ for n vertices, $\frac{n(n-1)}{2}$ edges.

Tree:- A conneded graph T without any cycle is called a tree graph or free tree or simply a tree.

→ if nodes = n

edges = n-1

→ Labeled graph:- A graph is said to be labeled if its edges are assigned data.

Weighted graph:- A graph G is said to be weighted graph if each edge e in G is assigned a non negative numerical value w(e) called the weight.

→ Sum of all edges of a path is said to be weight of path.

→ if no other information given ~~with~~ about weights, then $w(e) = 1$ is assumed.

Multiple edges:- Distinct edges e and e' are called multiple edges if they connect the same endpoints, that is, if $e = [u, v]$ and $e' = [u, v]$

loops:- An edge e is called a loop if it has identical endpoints, that is, $e = [u, u]$

→ A graph, that contains multiple edges or loops is called as a ~~multiple~~ multigraph.

→ The graph usually does not allow either multiple edges or loops.

Finite graph:- A multigraph is said to be finite if it has a finite number of vertices and edges.

Directed graph:- A directed graph or digraph is a multigraph with a direction associated with each edge e.

→ each edge is identified as an ordered pair (u, v) rather than unordered pair $[u, v]$

→ A directed edge $e=(u,v)$ of directed graph G is also called as arc.

The terminology is used:

1. e begins at u and ends at v.
2. u is origin or initial point and v is destination or terminal point of e.
3. u is a predecessor of v and v is a successor of u.
4. u and v are adjacent to each other.

Outdegree:- The outdegree of a node u in G, is the number of edges beginning at u.

In degree:- The indegree of u in G, is the number of edges ending at u.

Sink:- u is called a sink, if it has a zero outdegree but a positive indegree.

Source:- A node u is called a source if it has positive outdegree but zero indegree.

Reachable node:- A vertex v is said to be reachable from a vertex u if there is a directed path from u to v.

Strongly connected :- A directed graph G is said to be connected or strongly connected, if for each pair u, v of nodes in G, there is a path from u to v and there is also a path from v to u.

Unilaterally connected :- A directed graph G is said to be unilaterally connected, if for any pair u, v of nodes in G, there is a path from u to v or a path from v to u.

parallel edge :- The edges are said to be parallel edges if their end points are same.

Rooted tree :- The non empty tree T with ~~this~~ a designated node R is called a rooted tree and R is called its roots.

→ There is a unique simple path from the root R to any other node in T.

Simple :- A directed graph G is said to be simple if G has no parallel edges.

→ A simple graph G may have loops, but it can't have more than one loop at a given node.

→ A non directed graph G may be viewed as a simple directed graph by assuming that each edge [u,v] in G represents two directed edges (u,v) and (v,u).

Directed Acyclic Graph:- These graphs are directed graphs with no cycles.

→ Each edge is directed from one vertex to another, such that following those directions will never form a closed loop.

→ A directed graph is a DAG if and only if it can be topologically ordered, by arranging the vertices as a linear ordering that is consistent with all edge directions.

Bi-connected Graphs:- A bi-connected graph is a connected graph which cannot be broken down into any further pieces by deletion of any single node.

Separation edge:- A separation edge is an edge whose removal disconnects G.

Separation vertex:- A separation vertex is a vertex which disconnects G when removed.

Representation of graphs:-

There are two standard ways to represent a graph G in the memory.

1. Sequential representation
2. Linked representation.

→ The sequential representation of G, is by means of its adjacency matrix A.

→ The linked representation of G, is by means of linked lists of neighbors.

→ Regardless of the representation, the graph G is normally input by using its formal definition: a collection of nodes and a collection of edges.

Adjacency Matrix:- Suppose G is a simple directed graph with m ordered nodes, called $v_1, v_2, \ldots, v_m$. Then, the adjacency matrix $A = (a_{ij})$ of the graph G is the m×m matrix defined as.

$$a_{ij} = \begin{cases} 1 & \text{if there is an edge between } v_i \text{ and } v_j \\ 0 & \text{otherwise.} \end{cases}$$

Such matrix with only 0 or 1 entries, is called a bit matrix or a Boolean matrix.

→ The adjacency matrix A of graph G does depend on the ordering of the nodes of G, that is, $a_{ij} \neq a_{ji}$.

→ it means if there is a directed edge from v_i to v_j, then it will not be counted as v_j to v_i because of direction associated with the edge.

→ let $a_k(i,j)$ = the ij entry in the matrix A^k.

K will be the length of path from vertex v_i to vertex v_j. for example, $a_2(i,j)$ gives the

number of paths of length 2 from v_i to v_j.

→ The ij entry of the matrix B_r gives the number of paths of length r or less, from v_i to v_j, where

$$B_r = A + A^2 + A^3 + \cdots\cdots + A^r.$$

Path matrix:- The path matrix or reachability matrix of G is the m- square matrix $P = (P_{ij})$ defined as follows:

$$P_{ij} = \begin{cases} 1 & \text{if there is a path from } v_i \text{ to } v_j \\ 0 & \text{otherwise.} \end{cases}$$

→ Let A be the adjacency matrix and let $P = (P_{ij})$ be the path matrix of a digraph G. Then, $P_{ij} = 1$ if and only if there is a nonzero number in the ij entry of the matrix

$$B_m = A + A^2 + A^3 + \cdots\cdots + A^m$$

→ strongly connected if and only if the path matrix P of G has no zero entries.

Transitive closure:- The transitive closure of a graph G is defined to be the graph G' such that G' has the same nodes as G and there is an edge (v_i, v_j) in G' whenever there is a path from v_i to v_j.

→ A graph G is strongly connected iff its transitive closure is a complete graph.

Warshall's Algorithm:- Warshall gave an algo for the purpose that is much more efficient than calculating the powers of the adjacency matrix A.

→ first define m-square Boolean matrices $P_0, P_1, \ldots, P_m$. define

$$P_k[i,j] = \begin{cases} 1 & \text{if there is a path from } v_i \text{ to } v_j, \text{ using only nodes } v_1, v_2, \ldots, v_k \\ 0 & \text{otherwise.} \end{cases}$$

$P_0 = A$

$P_m = P$.

Warshall analyzed that $P_K[i,j]=1$ can be occured only one of the following cases:

1. $P_{K-1}[i,j]=1$

   path from v_i to v_j using $v_1, v_2, \ldots, v_{K-1}$ nodes.

2. $P_{K-1}[i,k]=1$ and $P_{K-1}[k,j]$.

   a simple path from v_i to v_k and v_k to v_j using $v_1, v_2, \ldots, v_{K-1}$ nodes.

→ The elements of the matrix P_K can be obtained by

$$P_K[i,j] = P_{K-1}[i,j] \vee (P_{K-1}[i,k] \wedge P_{K-1}[k,j])$$

Warshall Algo :-

Step 1 :- Initialization

Step 2 :- Repeat for $I, J = 1, 2, \ldots, M$.

[Initializes P.]

if $A[i,j]=0$, then: Set $P[I,J]=0$;

Else: Set $P[I,J]=1$

[End of loop]

Step 3:- Repeat Step 4 and 5 for K = 1, 2, ..., M.

[Updates P]

Step 4:- Repeat Step 5 for I = 1, 2, ..., M:

Step 5:- Repeat for J = 1, 2, ..., M:

Set $P[I,J] = P[I,J] \vee (P[I,K] \wedge P[K,J])$

[End of loop]

[End of Step 4 loop]

[End of Step 3 loop].

Step 6:- Exit.

Shortest-Path algorithm :- Dijsktra's Algo

In a weighted graph G, each edge in G is assigned a non negative number w(e) called the weight of the edge. The weight matrix is defined as follows:

$$w_{ij} = \begin{cases} w(e) & \text{if there is an edge from } v_i \text{ to } v_j \\ 0 & \text{if there is no edge from } v_i \text{ to } v_j. \end{cases}$$

→ Define a matrix Q finds the lengths of the shortest paths between the nodes as:

q_{ij} = length of a shortest path from v_i to v_j

$Q_k[i,j]$ = Shortest path from v_i to v_j using $v_1, v_2, \ldots, v_k$ nodes.

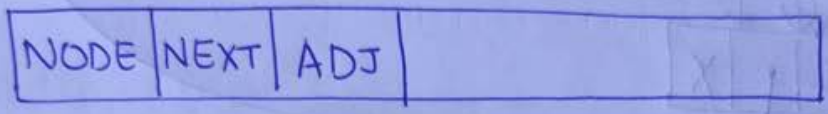
$$Q_k[i,j] = \min(Q_{k-1}[i,j], (Q_{k-1}[i,k] + Q_{k-1}[k,j])$$

Linked representation of a graph:-

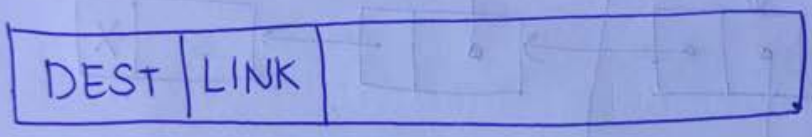
→ Graph is usually represented in memory by a linked representation, also called an adjacency structure.

→ The linked representation will contain two lists

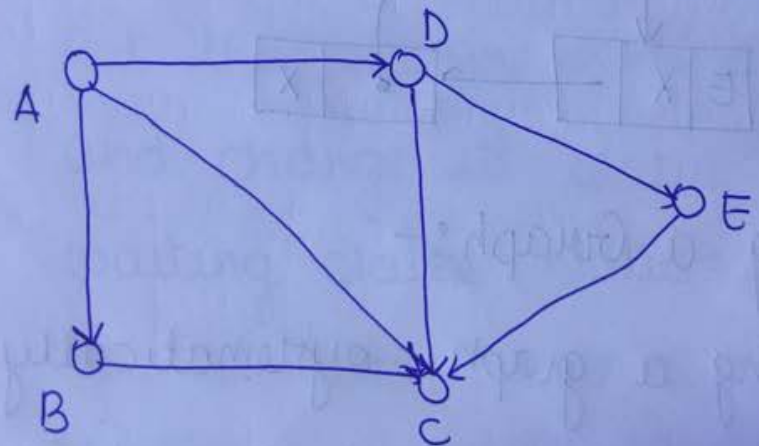
1. A node list NODE



2. Edge list EDGE

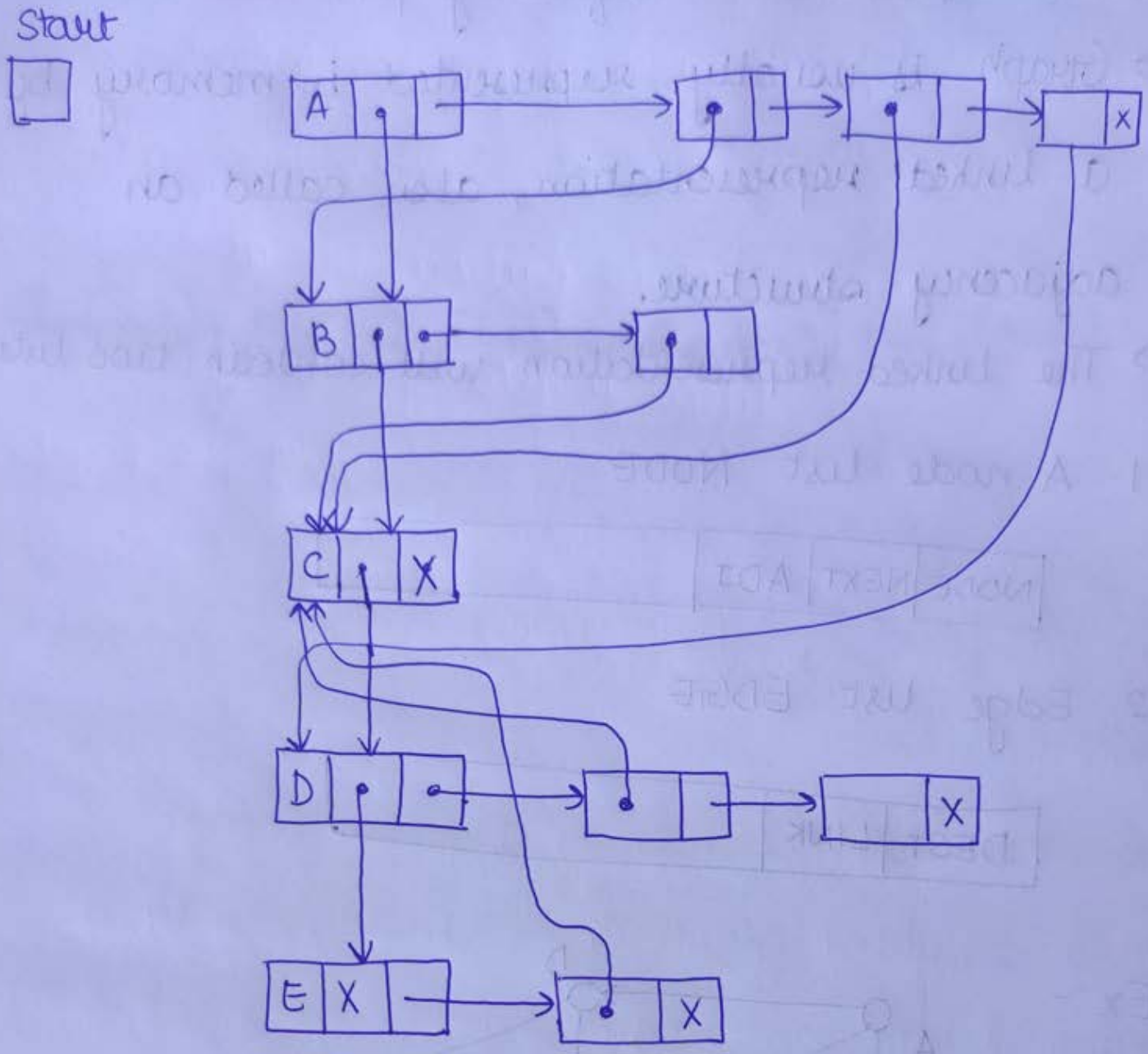


Ex:-



Adjacency list :-

node	Adjacent
A	B, C, D
B	C
C	
D	C, E
E	C.



Traversing a Graph:-

→ Traversing a graph, systematically examine the nodes and edges of a graph G.

→ There are two standard ways:

1. Breadth - first search
2. depth - first search.

→ During the execution of algo, each node N of will be in one of three states, called the status of N :

1. Status = 1 Ready state
2. Status = 2 Waiting state
3. Status = 3 Processed state

Breadth-First Search:-

Step 1:- Initialization

Step 2:- Initialize all nodes to the ready state (Status = 1)

Step 3:- Put the starting node A in Queue and change its status to the waiting states (Status = 2)

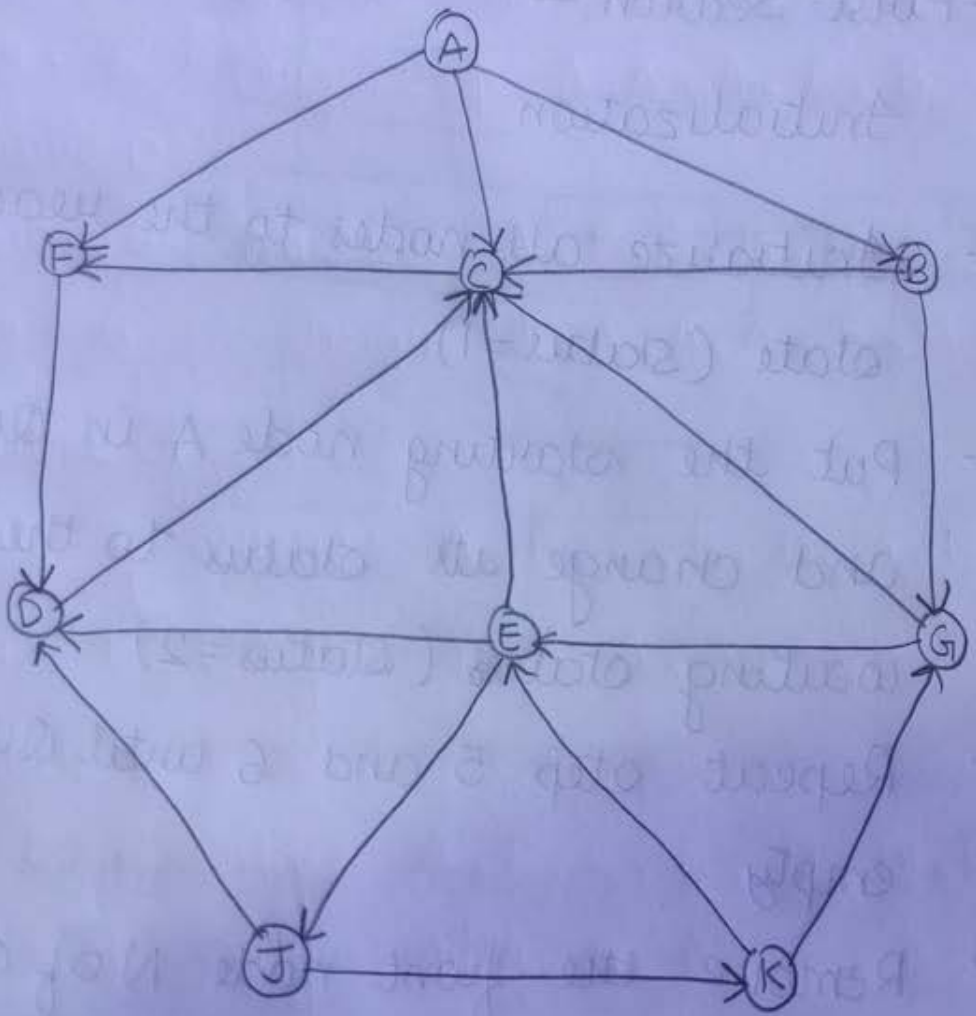
Step 4:- Repeat step 5 and 6 until Queue is empty

Step 5:- Remove the front node N of Queue. Proces N and change the status of N to the processed state. (Status = 3)

Step 6 :- Add to the rear of Queue all the neighbors of N that are in the ready state (Status = 1), and change their status to the waiting state (Status = 2)

[End of step 4 loop]

Step 7 :- Exit.



Adjancency lists:-

A : F, C, B

B : G, C

C : F

D : C

E : D, C, J

F : D

G : C, E

J : D, K

K : E, G.

→ Initially.

Fruotrt = 1 Queue = A

Rear = 1 Origin = ϕ

→ Remove A from front

Fruont = 2 Queue = A, F, C, B

Rear = 4 Origin = ϕ, A, A, A

→ Remove F

Front = 3 Queue: A, F, C, B, D

Rear = 5 Origin: ϕ, A, A, A, F

→ Remove C.

Front = 4 Queue: A, F, C, B, D.

Rear = 5 Origin: ϕ, A, A, A, F

→ Remove B

Front = 5 Queue = A, F, C, B, D, G

Rear = 6 Origin: ϕ, A, A, A, F, B

→ Remove D.

Front = 6 Queue: A, F, C, B, D, G

Rear = 6 Origin: ϕ, A, A, A, F, B

→ Remove G

Front = 7 Queue: A, F, C, B, D, G, E

Rear = 7 Origin: ϕ, A, A, A, F, B, G

→ Remove E

Front = 8 Queue: A, F, C, B, D, G, E, J

Rear = 8 Origin: ϕ, A, A, A, F, B, G, E

→ J is final destination

J ← E ← G ← B ← F ← A

is the required path.

Depth - First Search:-

Step 1:- Initialization

Step 2:- Initialize all nodes to the ready state (Status = 1)

Step 3:- Push the starting node A onto STACK and change its status to the waiting state (Status = 2).

Step 4:- Repeat steps 5 and 6 until stack is empty.

Step 5:- Pop the top node N of stack. Process N and change its status to the processed state (Status = 3)

Step 6:- Push onto stack all the neighbors 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]

Step 7:- Exit

→ Initially, push J

Stack : J

→ pop J, push D and K

Stack : D, K

print J

→ pop K, push E and G

Stack : D, E, G

print : K

→ pop G, push C

Stack : D, E, C

print : G

→ pop C, push F

Stack : D, E, F

print : C

→ pop F, push Nothing

Stack : D, E

print : F

→ pop E, push nothing

stack : D

print : E

→ pop D, push nothing

Stack : Empty

print : D.

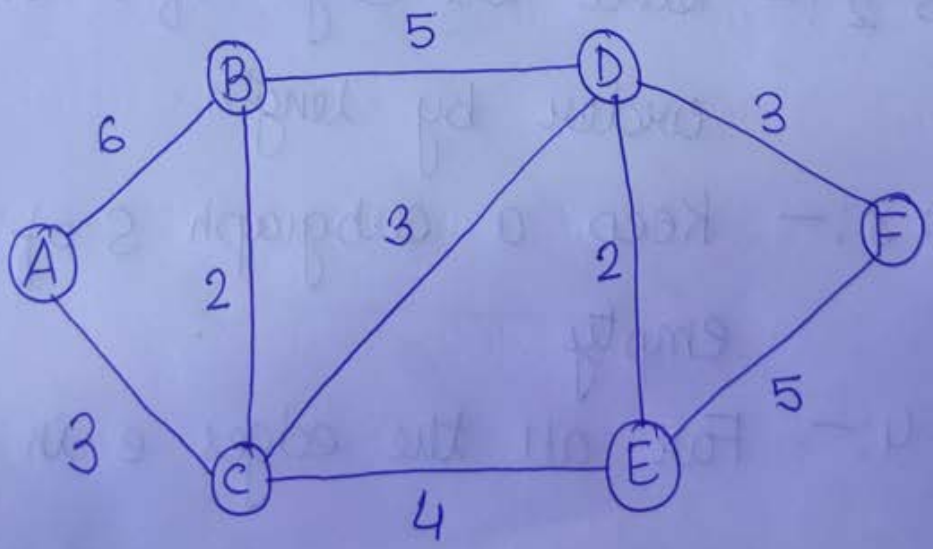
→ Nodes reachable from J

J, K, G, C, F, E, D

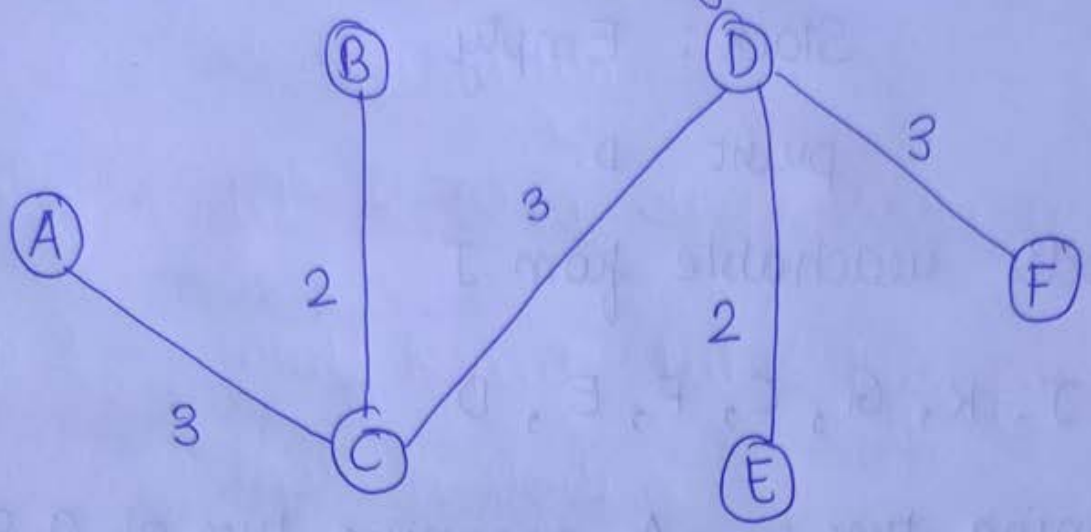
Spanning tree :- A spanning tree of a graph is simply a subgraph with all the vertices and is a tree.

Minimum Spanning tree :- A minimum spanning tree is a spanning tree in which the total weight of the ~~lines~~ edges is guaranteed to be the minimum of all possible trees in the graph.

Ex :-



minimum spanning tree of graph.



Finding minimum spanning tree:-

1. List all possible spanning trees and find minimum weight among all.
2. Build the MST using one edge at a time

Kruskal's Algo:-

Step 1:- Initialization

Step 2:- Sort the edges of H in increasing order by length.

Step 3:- Keep a subgraph S of H, initially empty

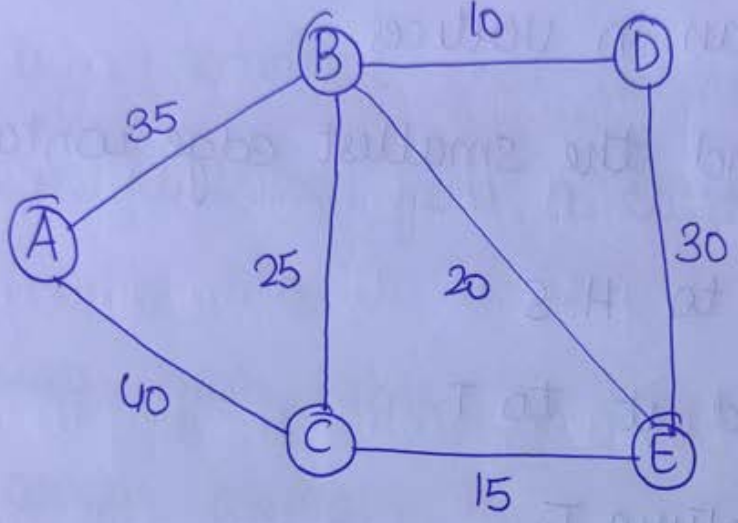
Step 4:- For all the edges e in sorted

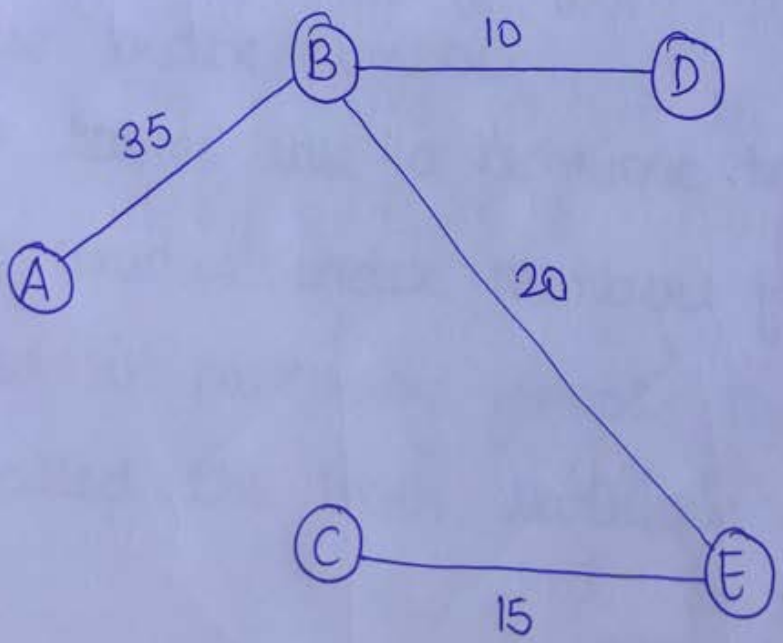
Order

a.) if the endpoints of e are disconnected in S.

b.) add e to S.

Step 5:- Return S





Prim's Algo :-

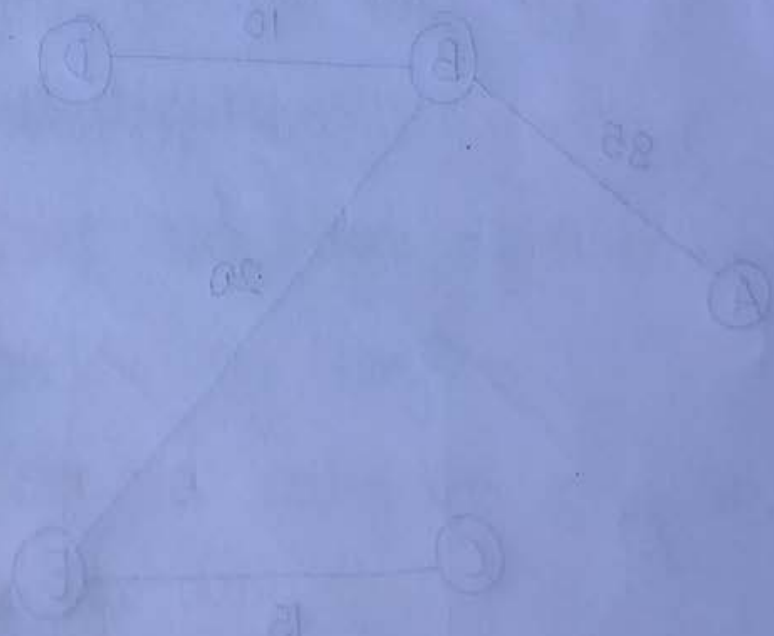
Step 1 :- Initialization

Step 2 :- Let S be a single vertex v

Step 3 :- ~~While~~ Repeat till S has fewer than n vertices

- find the smallest edge containing S to H·S.
- add it to T.

Step 4 :- Return T.



Hashing:- Hashing is a technique of mapping a large chunk of data into small tables using hash function.

→ It is also known as the message digest function.

→ It is a technique that uniquely identifies a specific item from a collection of similar item.

→ It uses hash tables to store the data in an array format.

→ Each value in the array has assigned a unique index number.

→ Hash tables use a technique to generate these unique index numbers for each value stored in an array format. This technique is called the hash technique.

Hashing in a data structure is a two-step process:-

1. The hash function converts the item into a small integer or hash value. This integer is used as an index to store the the original data.
2. It stores the data in a hash-table. You can use a hash key to locate data quickly.

→ The general idea of using the key to determine the addres of a record is an excellent idea, but it must be modified so that a great deal of space is not wasted.

→ The hash function or hashing function can be defined as

$$H : K \rightarrow L$$

Where H is function

K is set of keys.

L is set of memory location addres.

→ Hashing function H may not yield distinct values: two different keys k_1 and k_2 will yield the same hash address. This situation is called collision.

→ Hence, the hashing is divided into two parts:

1. hash functions
2. collision resolutions

→ methods to evaluate hash values:-

a.) Division method :- Choose a number m larger than the number n of keys in K.

→ The hash function H is defined by

$H(k) = k \pmod m$ or $H(k) = k \pmod m + 1$

→ Here, $k \pmod m$ denotes the remainder when k is divided by m.

→ Hence, hash addresses are range from 0 to m-1.

b.) Mid square method :-

→ The key k is squared. Then, the hash function

H is defined by

$$H(k) = l$$

where l is obtained by deleting digits from both ends of k^2.

c.) Folding method :- The key k is partitioned into a number of parts, $k_1, k_2, \ldots, k_r$.

→ Each part, except possibly the last, has the same number of digits as the required address.

→ Then, the parts are added together, ignoring the last carry. That is,

$$H(k) = k_1 + k_2 + \cdots + k_r$$

→ Sometimes, for extra "milling", the even-numbered parts, $k_2, k_4, \ldots$, are each reversed before the addition

Ex. 68 - employees assigned with a unique 4-digit employee number. L consist 100 - two-digit addres: 00, 01, 02, ..., 99.

Employee numbers : 3205, 7148, 2345

a. Division method :-

Choose a prime number m close to 99, m = 97

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

b.) Mid square method :-

k :	3205	7148	2345
k^2 :	10272025	51093904	5499025
H(k) :	72	93	9 /99

c.) Folding method :-

H(3205) = 32 + 05 = 37 , H(7148) = 71 + 48 = 19.

H(2345) = 23 + 45 = 68

Reverse the second part

H(3205) = 32 + 50 = 82 , H(7148) = 71 + 84 = 55

H(2345) = 23 + 54 = 77

Collision Resolution :- The efficiency of a hash fun" with a collision resolution procedure is measured by the average number of probes (key comparisons) needed to find the location of

the record with a given key k.

→ The efficiency depends mainly on the load factor λ.

S(λ) = average number of probes for a succesful search.

U(λ) = average number of probes for an unsuccesful search.

Collision Resolution Techniques:-

1. Open addressing :- array-base implementation
2. Separate Chaining:- array of linked list imp.

Open addressing :-

1. linear probing:-

→ New record R with key k has to add in memory table.

→ H(k) = h is already filled. now search T[h], T[h+1], T[h+2], ---- linearly to find the location to save R. R not meeting with empty location known as unsuccesful search.

→ this method of collision resolution is known as linear probing.

$S(\lambda) = \frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right)$ and $U(\lambda) = \frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$

where $\lambda = n/m$.

Ex:- Suppose table of 11 memory location Record with $H(k) = 5$ and $T[5]$ is already assigned.

The average number S

$$= \frac{1+1+1+1+2+2+2+3}{8} = \frac{13}{8} = 1.6$$

The average number U

$$= \frac{7+6+5+4+3+2+1+1+1+8}{11} = \frac{40}{11} = 3.6$$

2. Quadratic probing :- Instead of searching, $h, h+1, h+2 \cdots$, the searching location will be like.

$h, h+1, h+4, h+9, h+16, \cdots, h+i^2$.

3. double hashing :- using second hash function H'

$h, h+h', h+2h', h+3h', \cdots$

2. Chaining :- Chaining involves maintaining two tables in memory.

→ There is a table T in memory which contains the records in F.

→ Now T has an additional field LINK which is used so that all records in T with same hash address h may be linked together to form a linked list.

→ The average number of probes:

$$S(\lambda) \approx 1 + \frac{1}{2}\lambda$$

$$U(\lambda) \approx e^{-\lambda} + \lambda.$$