Recursion

11/9/23

Fact(F, N)

1. If N=0 , Set F=1 & return
2. Call Fact(F, N-1)
3. Set F=N*F
4. Return.

Fact (N)

1. If N=0 Set F=1 & return
2. If (N>1)

   return N*(Fact(N-1))

Fibonacci Series 0 1 1 2 3 5 8 13 21 .....

Fibo (N)

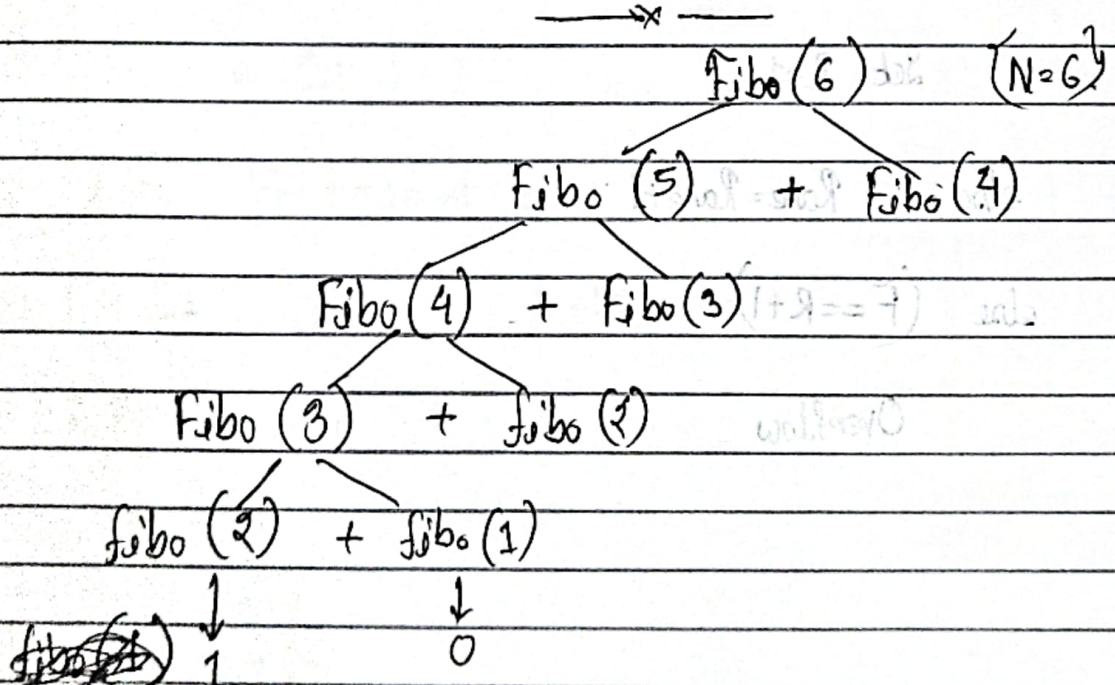
1. Set N

2. If N=1 , return 0

3. If N=2 , return 1

4. Else (N ≥ 3)

return Fibo(N-1) + Fibo(N-2)



Queue

18/9/23

~~First in first out~~

Front (delete) ⟶ Enqueue.

Rear (Insert) ⟶ Dequeue.

Enqueue. (Queue Insert)

IF (Front = 1 , Rare = 5) (F ≤ R)

Overflow

else Front ≠ ~~Front~~ + 1 , R = 1

Set R = 1

else Rare = Rare + 1

else (F == R + 1)

Overflow

Dequeue (Queue)

1. If Front = NULL

   print overflow.

2. Set item = Queue[Front]

3. If Front = Rare

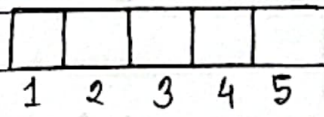
   Then Front = Null , Rare = Null.

   Else if Front = N

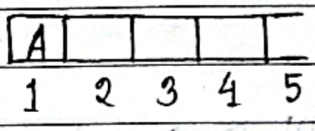
   Then Set F = 1

   Else Front = Front + 1

F = 0 , R = 0



(a) Initially Empty. N = 5



(b) A, B, C inserted.

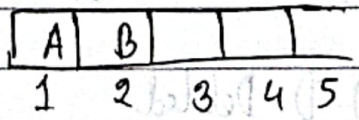
(i) F = R = 1

Q[F] = A

Q[R] = A

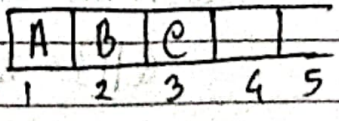
(ii) R = 1 + 1 = 2

Q[2] = B



(iii) R = 2 + 1 = 3

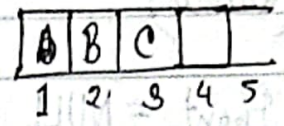
Q[3] = C



(c) A deleted.

(i) Front = 1

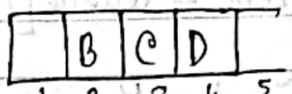
.Front = Front + 1 = 2



(d) D, E Inserted

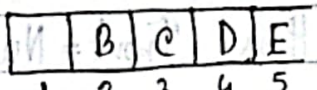
(i) R = 3 + 1

Q[R] = Q[4] = D



(ii) R = 4 + 1

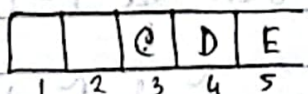
Q[R] = Q[5] = E



(e) B, C deleted

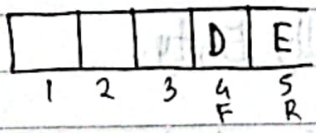
(i) Front = Front + 1

= 2 + 1 = 3



(ii) Front = Front + 1

= 3 + 1 = 4

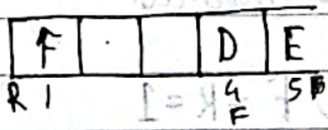


(f) F Inserted

(i) R = 5 + 1 = N no,

R = 1

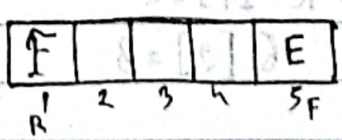
Q[R] = Q[1] = F



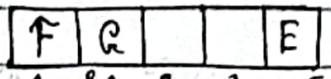
(g) D Deleted

(i) Front = Front + 1 = 5

Q[F] = Q[5] = F

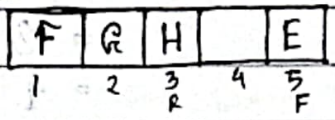


(h) G, H Inserted.

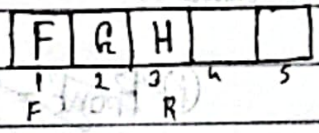


(i) $R = R+1$
$= 1+1$
$= 2$
$Q[R] = Q[2] = G$

(ii) $R = R+1$
$= 2+1$
$= 3$
$Q[R] = Q[3] = H$

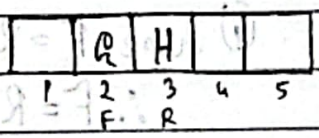


(i) E deleted.



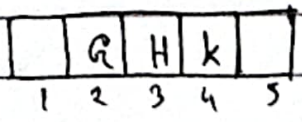
(i) $Front = Front + 1 == N$, So,
$Front = 1$
$Q[Front] = Q[1] = F$

(j) F deleted



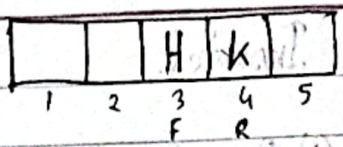
(i) $Front = Front + 1$
$= 1+1$
$= 2$
$Q[Front] = Q[2] = G$

(k) k inserted



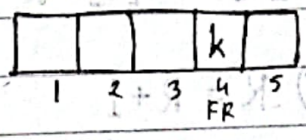
(i) $R = R+1$
$= 3+1$
$= 4$
$Q[R] = Q[4] = k$

(l) G, H deleted.

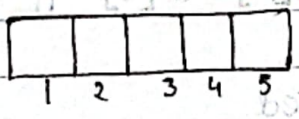


(i) Front = Front + 1
= 2 + 1
= 3

(ii) Front = Front + 1
= 3 + 1
= 4
Q[Front] = Q[4] = k



(m) k deleted,



(i) ~~Front = Front + 1~~
~~= 4 + 1~~
~~= 5~~
~~Q[Front] = Q[5] =~~

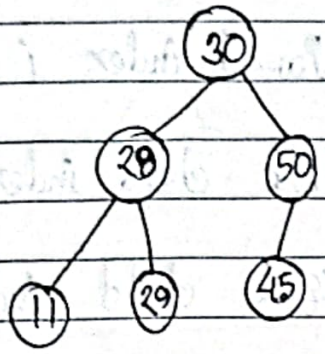
(i) Here F = R = 4
∴ F = R = 0

Tree

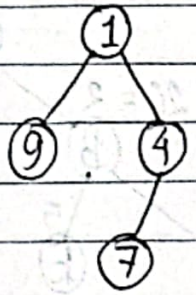
25/9/23

Binary tree → 2 children at most.

A<B<C
A<a

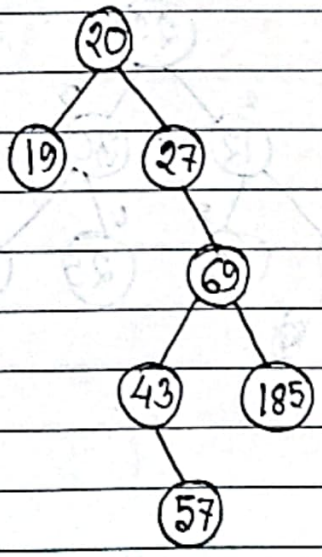


Binary Search tree



Normal Binary tree.

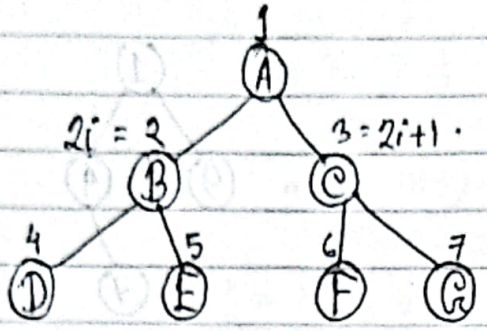
Ex: 20, 19, 27, 69, 185, 43, 57; Construct a Binary Search tree.



1/12/23

Tree

- In order, pre order, post order of a binary tree. (normal)



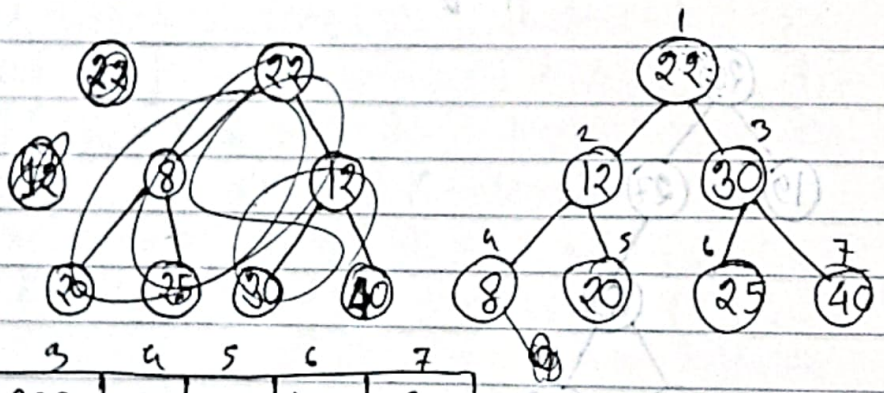
Parent index i

left child index 2*i

Right child index 2i+1

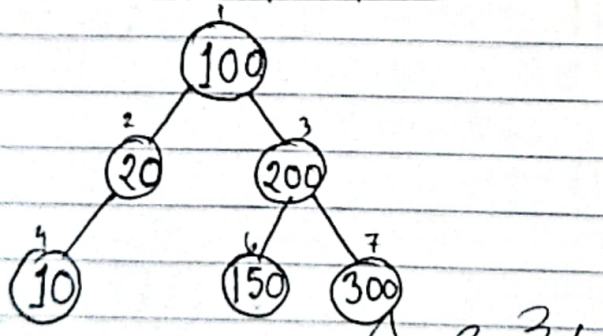
A	B	C	D	E	F	G
1	2	3	4	5	6	7

- Binary search tree: 22, 12, 8, 30, 40, 20, 25



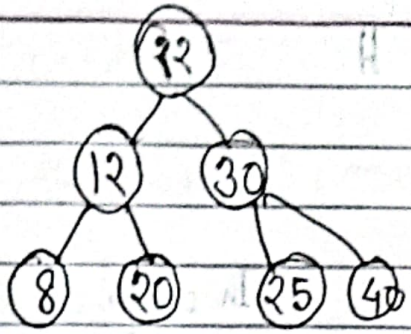
-

1	2	3	4	5	6	7
100	20	200	10		150	300



pre-order R T L R →

1	2	3	4	5	6	7
100	20	10	200	150	300	



L R$_T$ R → system of f calling & Value Returning.

8	12	20	22	25	30	40

L R$_T$ R → This is In order of Binary search tree.

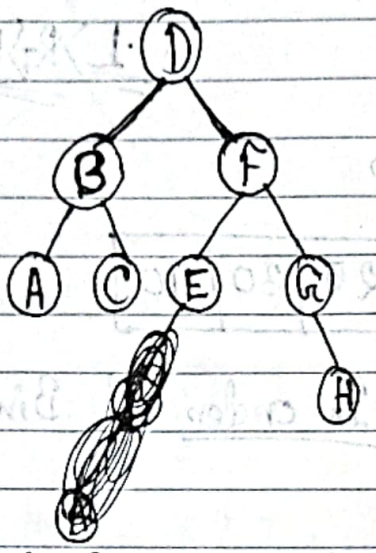
22	12	8	20	30	25	40

R$_T$ L R → This is Pre order of Binary Search tree.

~~20~~	8	20	12	~~22~~	25	40	30	22

L R R$_T$ → Post order of Binary ~~tree~~ Search tree.

BST: D, F, E, B, G, A, C, H



In order, pre order, post order.

In Order: A, B, C, D, E, F, G, H

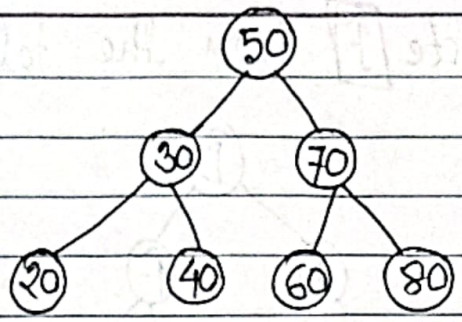
Pre Order: D, B, A, C, F, E, G, H

Post Order: A, C, B, E, H, G, F, D.

5/10/23

Deletion on BST:

Delete 20 from BST –



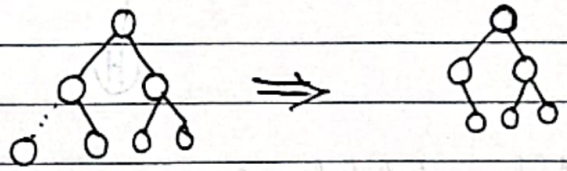
there is 3 cases:-

(i) leaf node

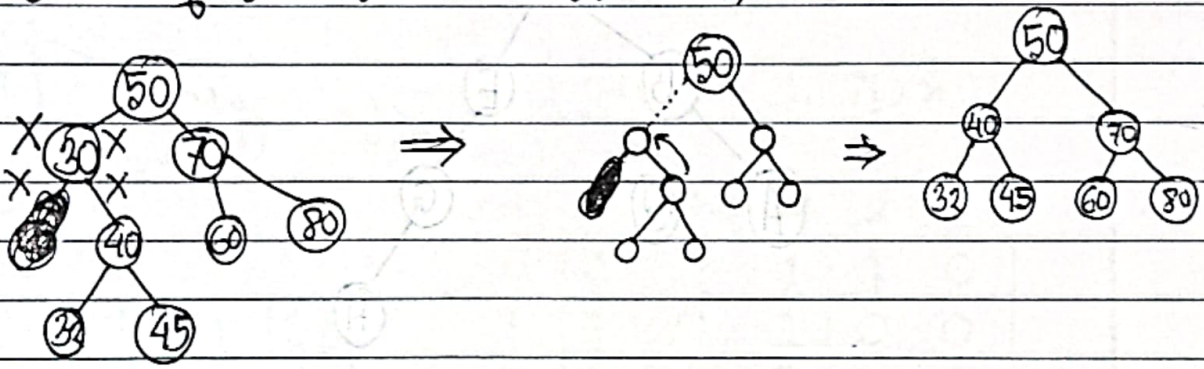
(ii) has 1 child

(iii) has 2 child.

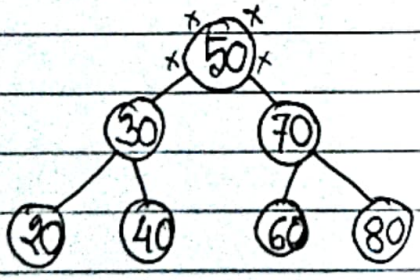
In case of leaf node -



If the deleting index has 1 child then;



In case of that index which has 2 child, then –

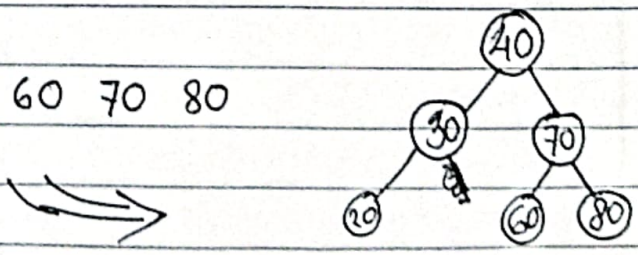


(i) Step 1: have transform to In order.

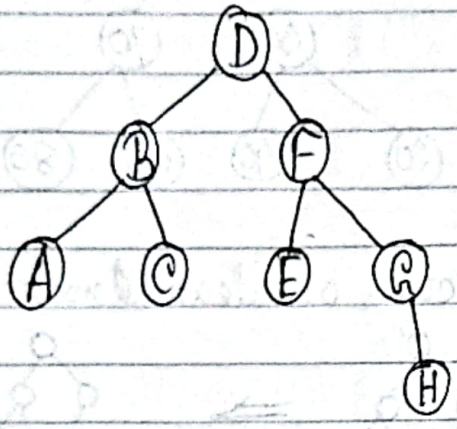
(ii) Step 2: Del & set predicitor.

In Order: 20 30 40 50 60 70 80

L R, R.

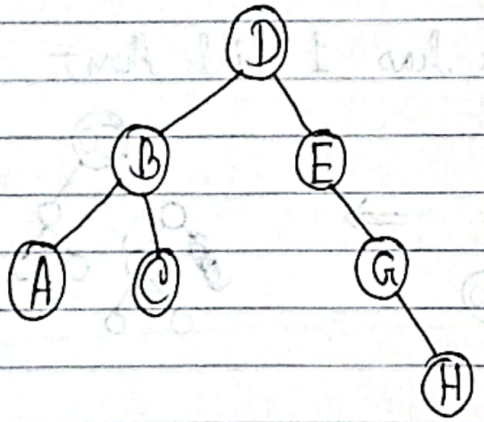


Delete [F] from the following tree:



In Order: A, B, C, D, E, F, G, H,

After deletation:



int N=15

Char T={'\0', 'D', 'B', 'F', 'A', 'C', 'E', 'G', '\0' ..... 'H'}

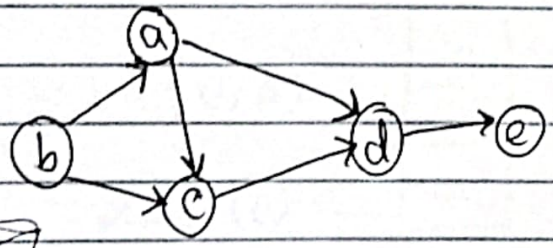
1	2	3	4	5	6	7	.....	15
D	B	F	A	C	E	G		15

```
int main(){
int a;
pf("Enter root");
scan("%d",&a);
Postorder(a);
```

Void PostOrder (int index)

{

(Graph)



n=5 | n x n

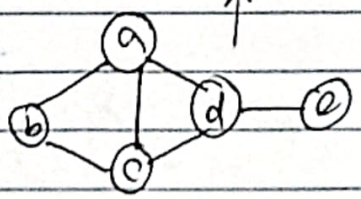
	a	b	c	d	e
a	0	1	1	1	0
b	1	0	1	0	0
c	1	1	0	1	0
d	1	0	1	0	1
e	0	0	0	1	0

$a_{ij}=1$ if $(u,v) \in E$

E={(a,b),(a,c),(a,d),(b,c),(c,d),

(d,e),(b,a),(c,a),(d,a),

(c,b),(d,c),(e,d)}

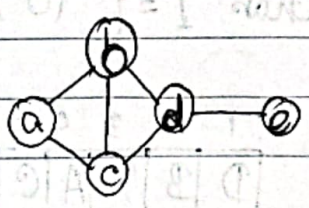


	a	b	c	d	e
a	0	0	1	1	0
b	1	0	1	0	0
c	0	0	0	1	0
d	0	0	0	0	1
e	0	0	0	0	0

DFS & BFS

15/10/23

- DFS = Depth first Search (Stack) (Completed Graph)
- BFS = Breath first Search



Algorithm of DFS

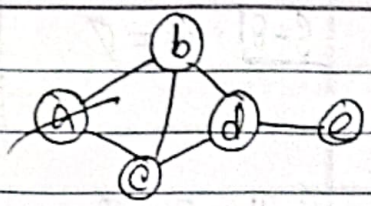
1. For each vertex v in V do mark "new"
2. While there exists a vertex v in V marked "new" do -
   (a) Search (v)

Procedure Search (v):

1. mark v "old"
2. Print (v)
3. For each vertex w adjacent to v do -
   (a) If w is marked "new" then -
      (i) Search (w)

যখন কোনো Adj. vertex ও Adj. circle থাকবে না তখন Stack থেকে বের হতে হবে

Source a



S-1 Initially

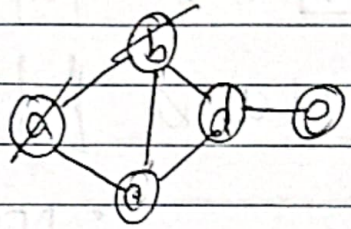
S-2 | a |

S-3 Print (a)

a = {b, c}

Search (b)

| b |
| a |

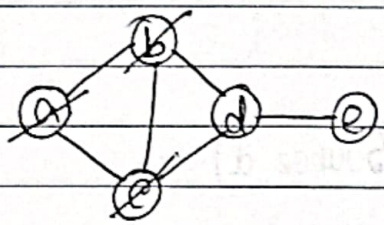


S-4 Print (b)

b = {c, d}

Search (c)

| c |
| b |
| a |

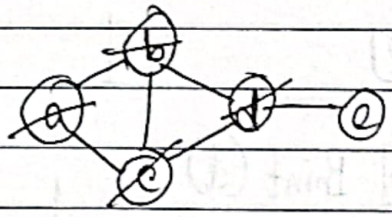


S-5 Print (c)

c = {d}

Search (d)

| d |
| c |
| b |
| a |

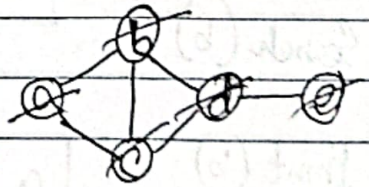


S-6 Print (d)

d = {e}

Search (e)

| e |
| d |
| c |
| b |
| a |



S-7 Print (e)

e = ∅

| d |
| c |
| b |
| a |

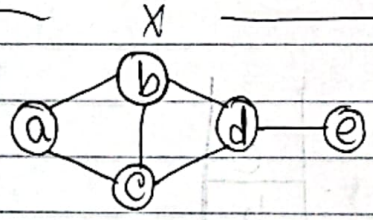
S-8] $d = \emptyset$ | e |
| b |
| a |

S-9] $c = \emptyset$ | b |
| a |

S-10] $b = \emptyset$ | a |

S-11] $a = \emptyset$ | |

∴ DFS: a, b, c, d, e.



Source d

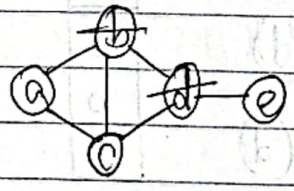
S-1] Initially | |

S-2] | d |

S-3] Print (d)

$d = \{b, c, e\}$ | b |
| d |

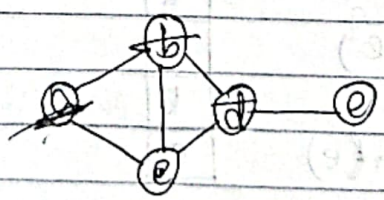
Search (b)



S-4] Print (b)

$b = \{a, c\}$ | a |
| b |
| d |

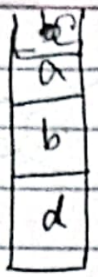
Search (a)

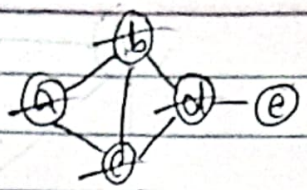


S-5 Print(a)

a = {c}

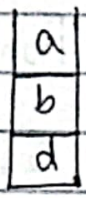
Search (c)



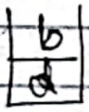


S-6 Print (c)

c = { }

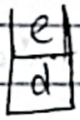


S-7 a = { }



S-8 b = ∅

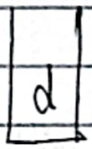
S-9 d = {e}



Search (e)

S-10 Print (e)

e = { }

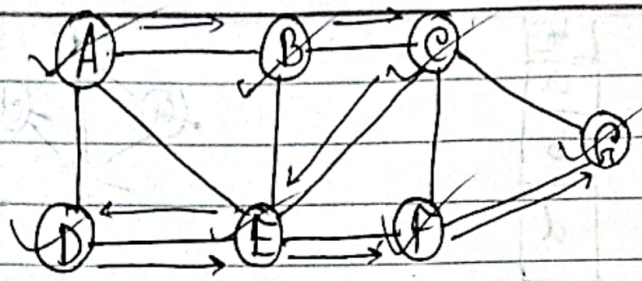


S-11 d = { }

∴ DFS : d, b, a, c, e.

16/01/23

DFS



Source - A

S-1] Initially ☐

S-2] A

S-3] Print (A)

A = {B, D, E} | A | B |

Search (B)

S-4] Print (B)

B = {C, E} | A | B | C |

Search (C)

S-5] Print (C)

C = {E, F, G} | A | B | C | E |

Search (E)

S-6] Print (E)

E = {D, F} | A | B | C | E | D |

Search (D)

S-7] Print (D)

D = { } | A | B | C | E |

S-8] E = {F} | A | B | C | E | F |

Search (F)

~~S-9] C = {F, G}~~

~~Search (F)~~ ~~A B C F~~

S-10] F = {G}

Search (G) | A | B | C | F | G |

S-11] G = { } | A | B | C | F |

S-12] F = { } | A | B | C |

S-13] B = { } | A | B |

S-14] B = { } | A |

S-15] A = { } ☐

∴ DFS : A, B, C, E, D, F, G.

Example 7.22

Insert 65 into the AVL search tree shown in Fig. 7.39(a). Figure 7.39(b) shows the unbalanced tree after insertion and Fig. 7.39(c) the rebalanced search tree after RR rotation.

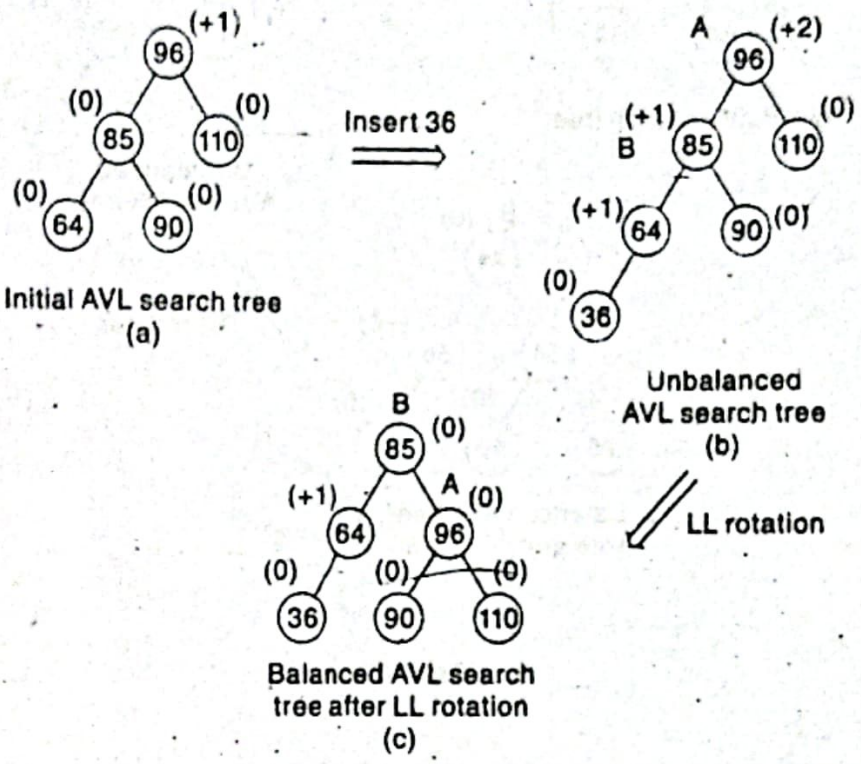


Fig. 7.37

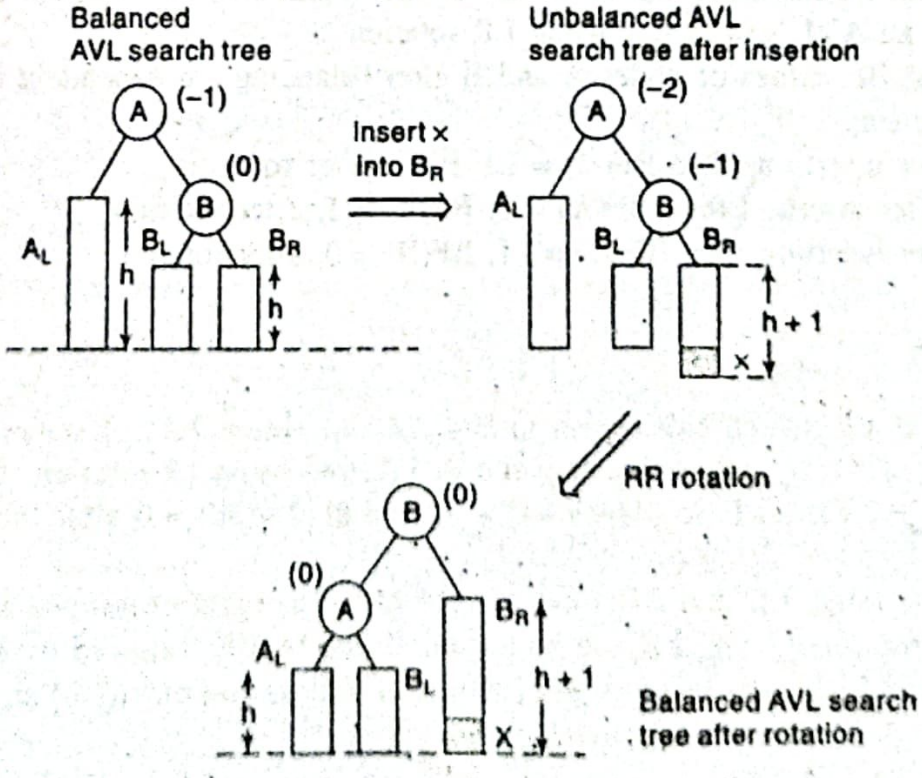


Fig. 7.38

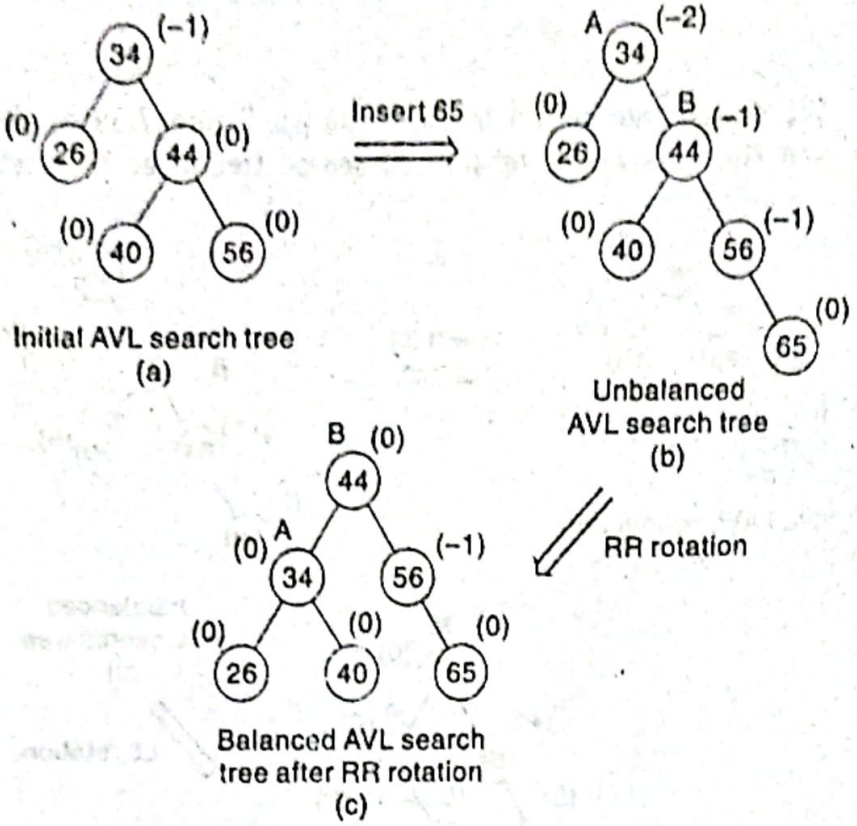


Fig. 7.39

LR and RL Rotations

The balancing methodology of LR and RL rotations are similar in nature but are mirror images of one another. Hence we illustrate one of them viz., LR rotation in this section. Fig. 7.40 illustrates the balancing of an AVL search tree using LR rotation.

In this case, the BF values of nodes A and B after balancing are dependent on the BF value of node C after insertion.

If BF (C) = 0 after insertion then BF(A) = BF(B)=0, after rotation
If BF (C) = –1 after insertion then BF(A) = 0, BF(B) = 1, after rotation
If BF (C) = 1 after insertion then BF(A) = –1, BF(B) = 0, after rotation

Example 7.23

Insert 37 into the AVL search tree shown in Fig. 7.41(a). Figure 7.41(b) shows the imbalance in the tree and Fig. 7.41(c) the rebalancing of the AVL tree using LR rotation. Observe how after insertion BF(C = 39) = 1 leads to BF(A = 44) = –1 and BF(B = 30) = 0 after the rotation.

Amongst the rotations, LL and RR rotations are called as *single rotations* and LR and RL are known as *double rotations* since, LR can be accomplished by RR followed by LL rotation and RL can be accomplished by LL followed by RR rotation. The time complexity of an insertion operation in an AVL tree is given by $O(height)=O(\log n)$.

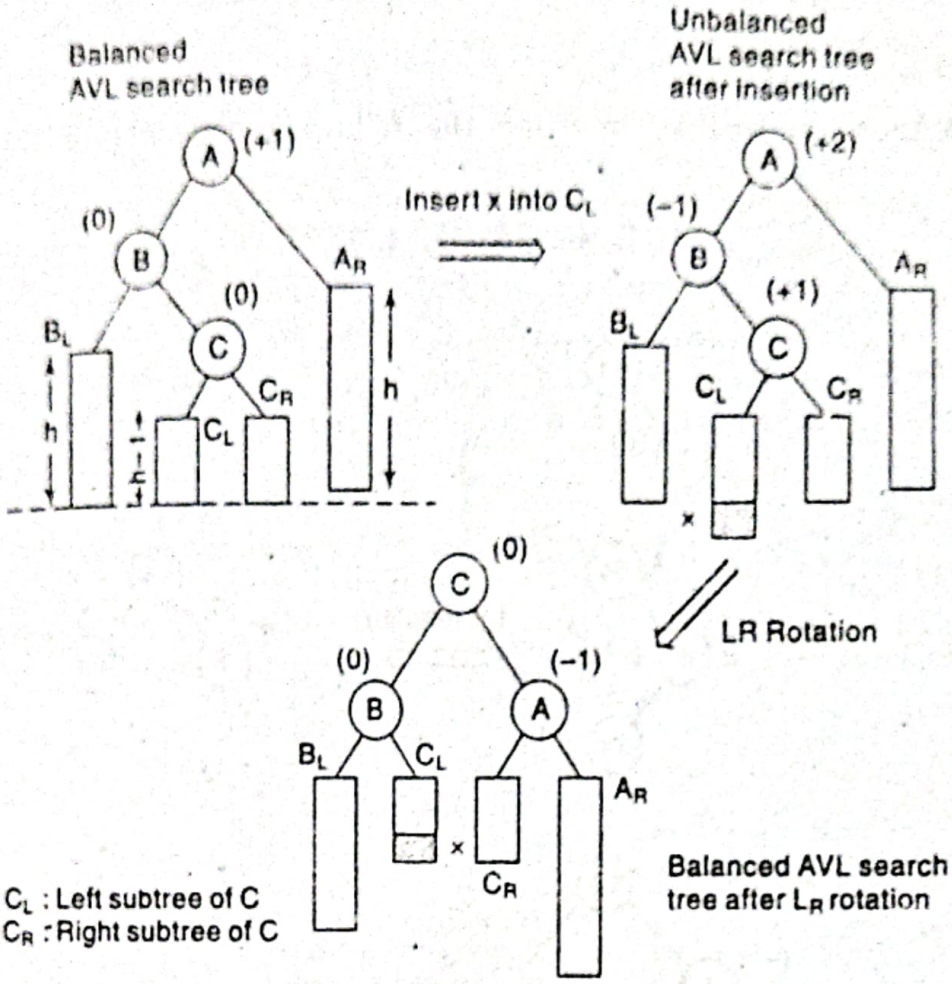


Fig. 7.40

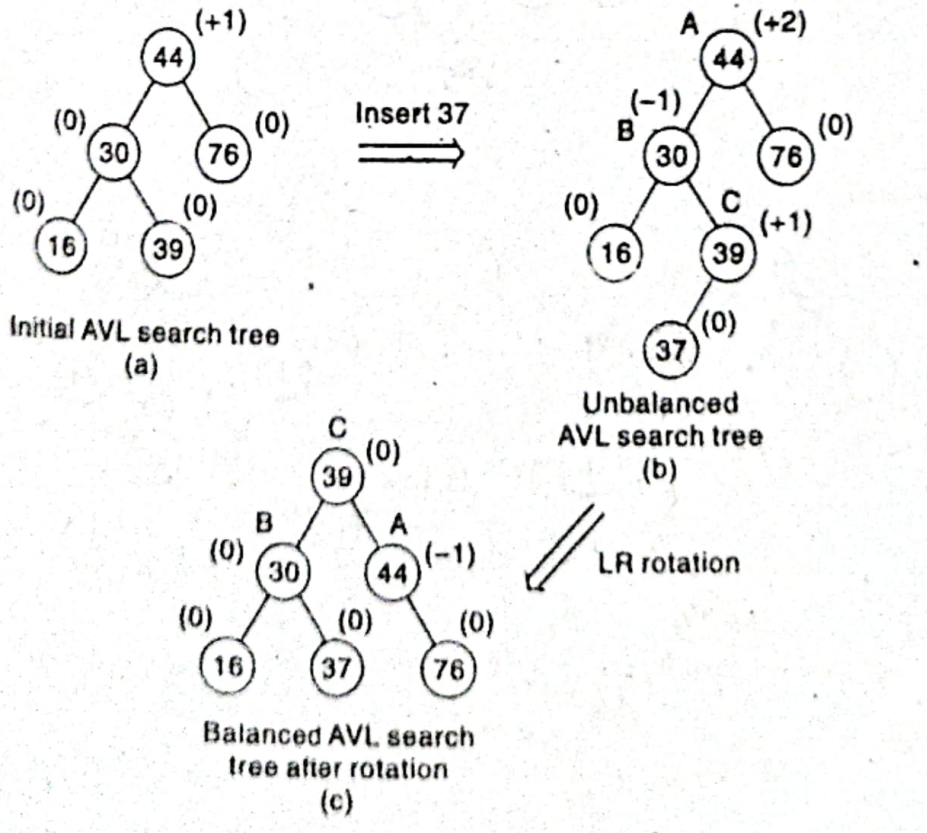


Fig. 7.41

Example 7.24

Construct an AVL search tree by inserting the following elements in the order of thei occurrence.

64, 1, 44, 26, 13, 110, 98, 85

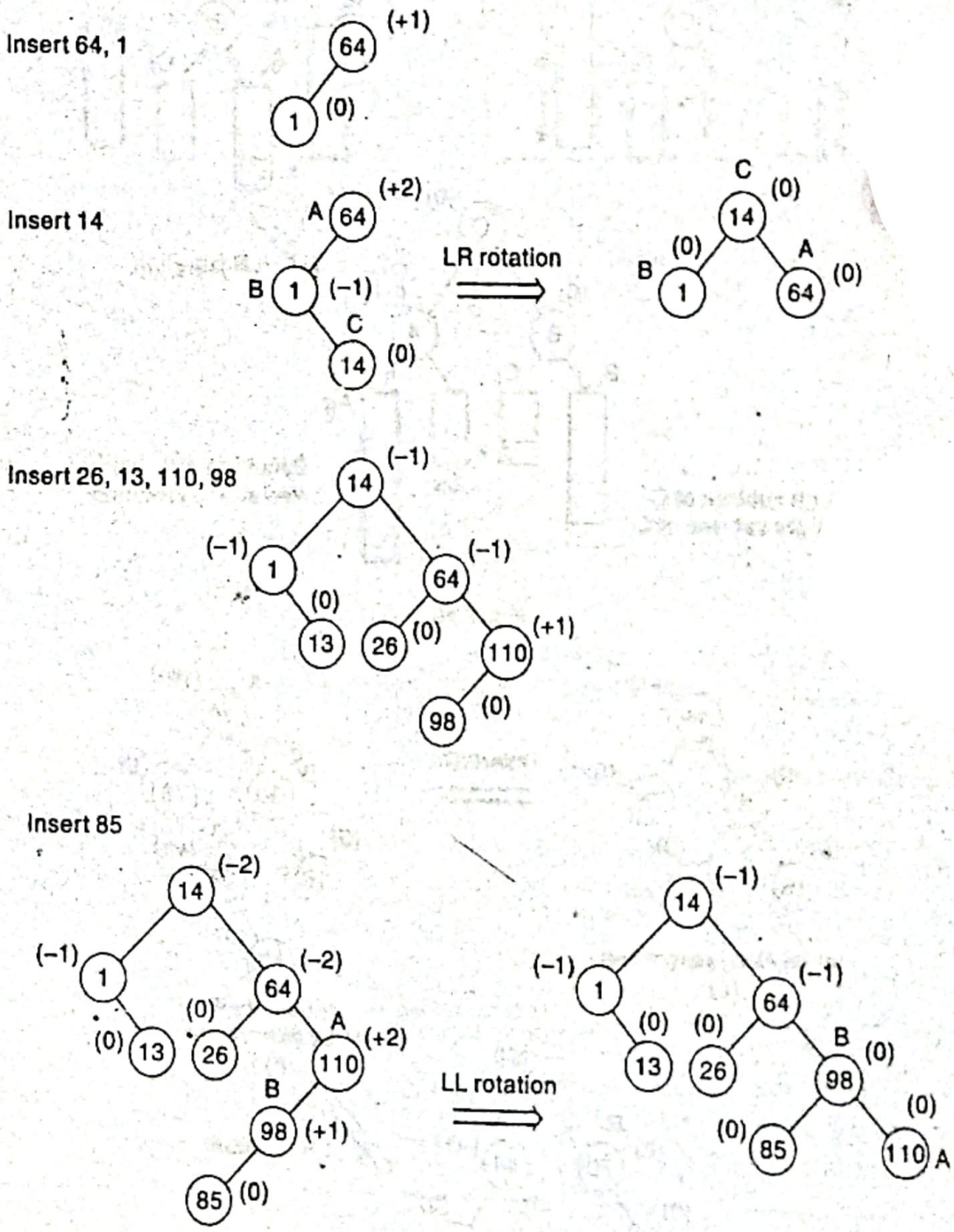


Fig. 7.42

DELETION IN AN AVL SEARCH TREE

deletion of an element in an AVL search tree proceeds as illustrated in procedures 7.6, 7.7 and rithm 7.8 for deletion of an element in a binary search tree. However, in the event of an lance due to deletion, one or more rotations need to be applied to balance the AVL tree.
deletion of a node X from the AVL tree, let A be the closest ancestor node on the path X to the root node, with a balance factor of +2 or –2. To restore balance the rotation is first sified as L or R depending on whether the deletion occurred on the left or right subtree of A. ow depending on the value of BF(B) where B is the root of the left or right subtree of A, the R imbalance is further classified as R0, R1 and R-1 or L0, L1 and L-1. The three kinds of R ons are illustrated with examples. The L rotations are but mirror images of these rotations.

Rotation

F(B)=0, the R0 rotation is executed as illustrated in Fig. 7.43.

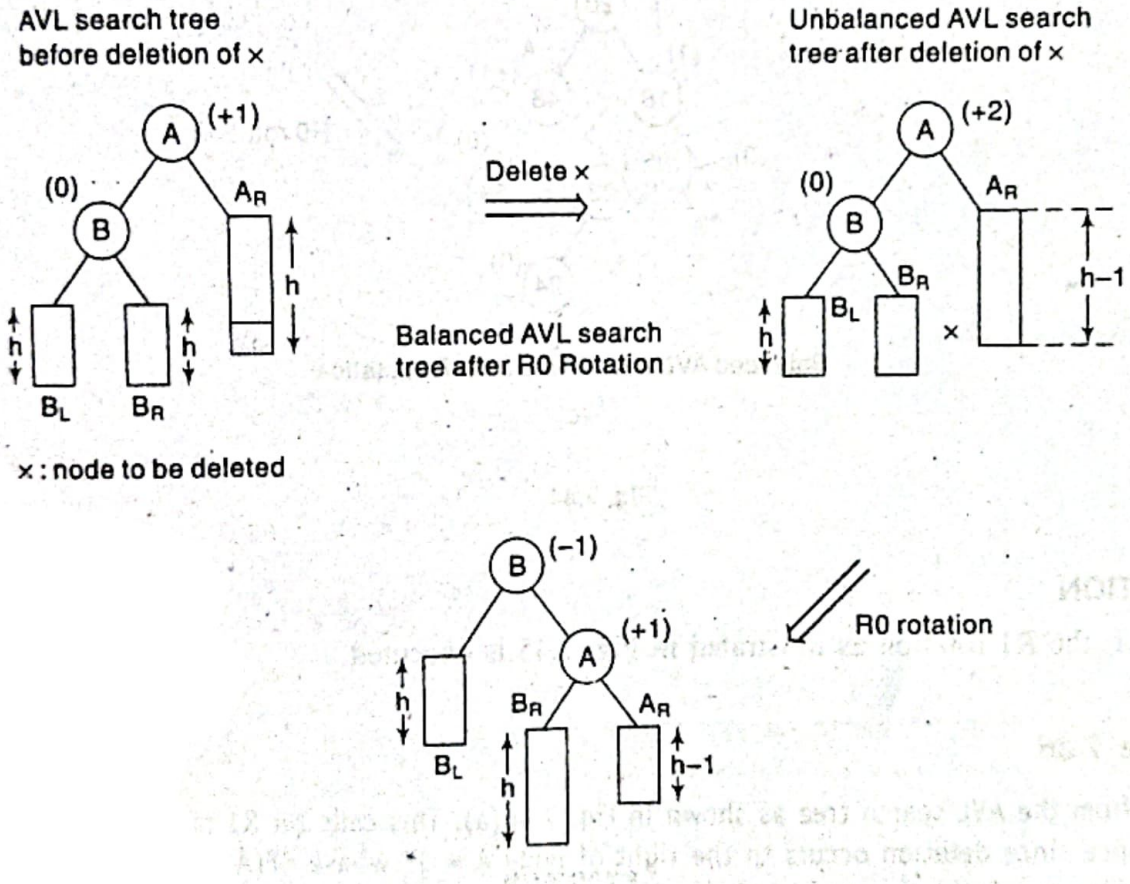


Fig. 7.43

... 7.25

elete 60 from the AVL search tree shown in Fig: 7.44(a). This calls for R0 rotation to set right e imbalance since deletion occurs to the right of node A = 46 whose BF(A = 46) = +2 and BF(B 20) = 0. Figure 7.44(b) shows the imbalance after deletion and Fig. 7.44(c) the balancing of the ee using R0 rotation.

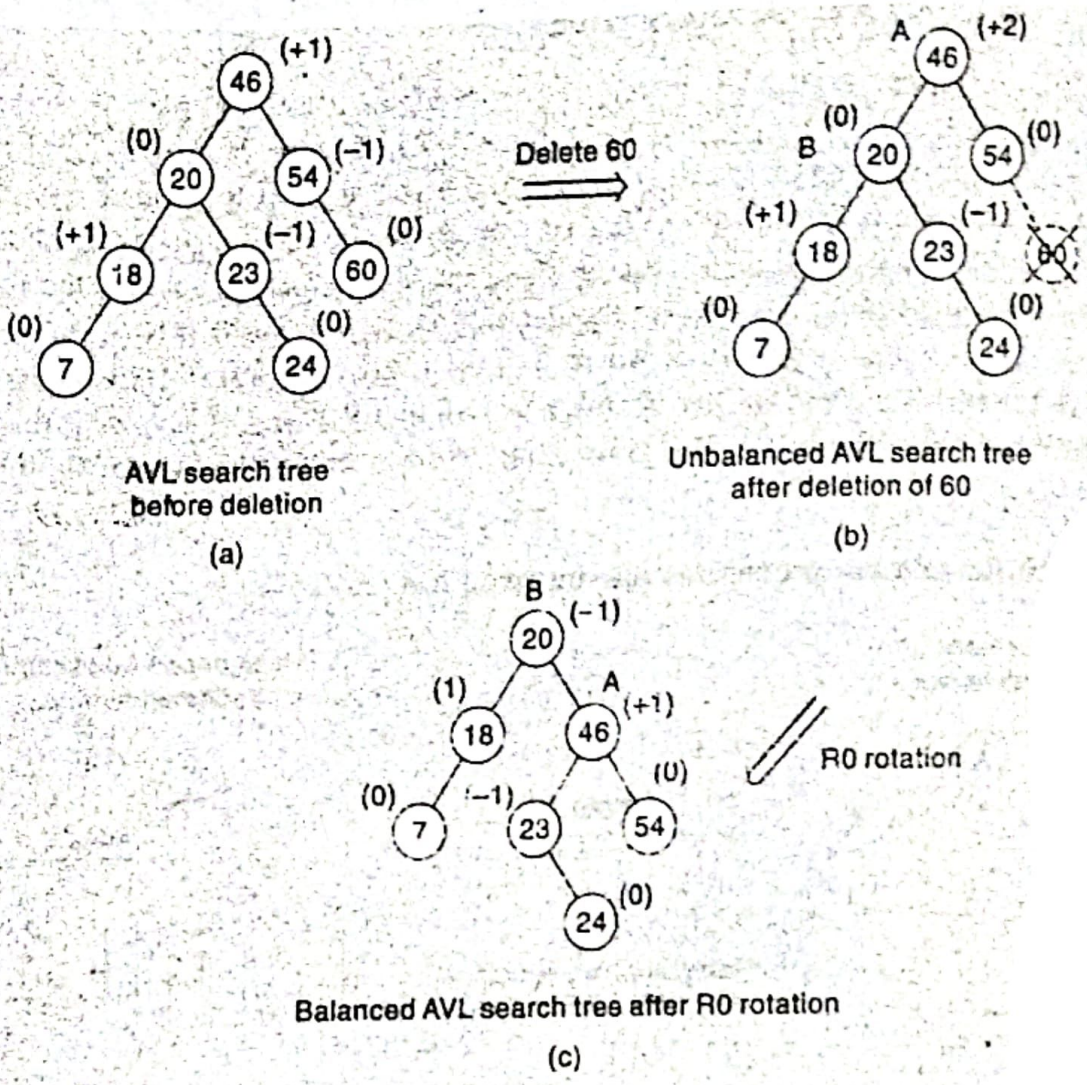


Fig. 7.44

R1 ROTATION

If BF(B) = 1, the R1 rotation as illustrated in Fig. 7.45 is executed.

Example 7.26

Delete 39 from the AVL search tree as shown in Fig. 7.46(a). This calls for R1 rotation to set right the imbalance since deletion occurs to the right of node A = 37 whose BF(A = 37) = +2 and BF(B = 26) = 1. Figure 7.46(b) shows the imbalance after deletion and Fig. 7.46(c) the balancing of the tree using R1 rotation.

R-1 Rotation

If BF(B) = −1, the R−1 rotation as illustrated in Fig. 7.48 is executed.

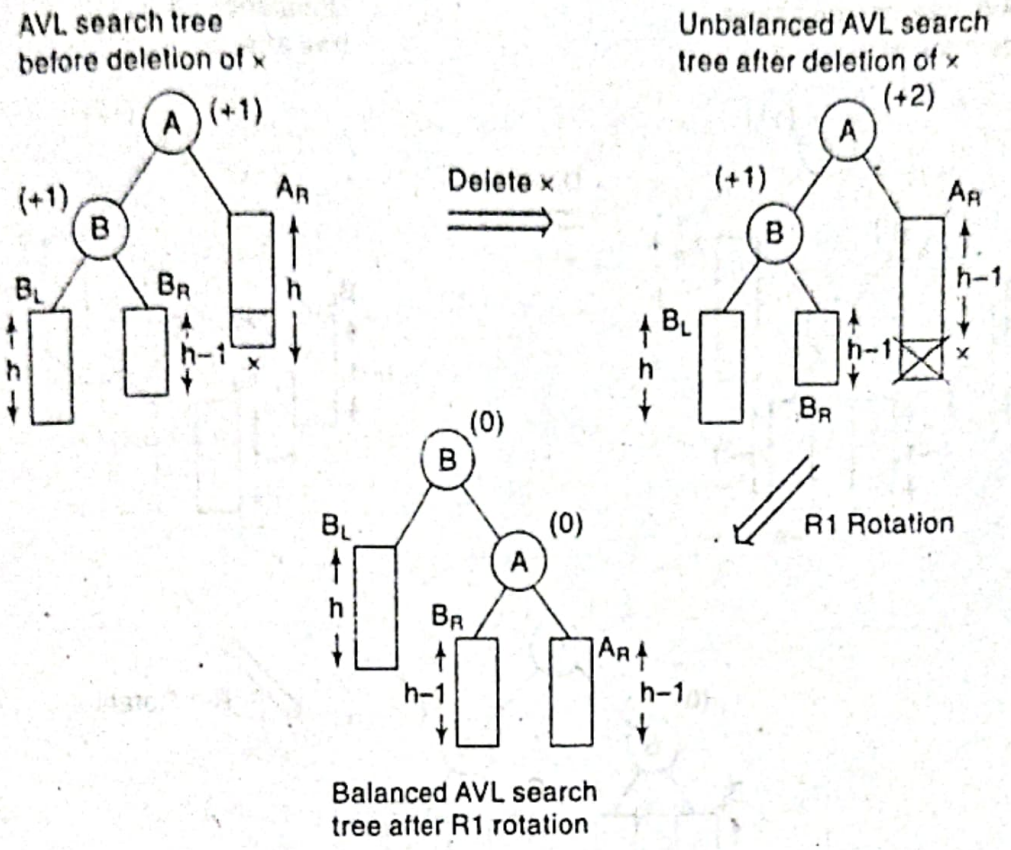


Fig. 7.45

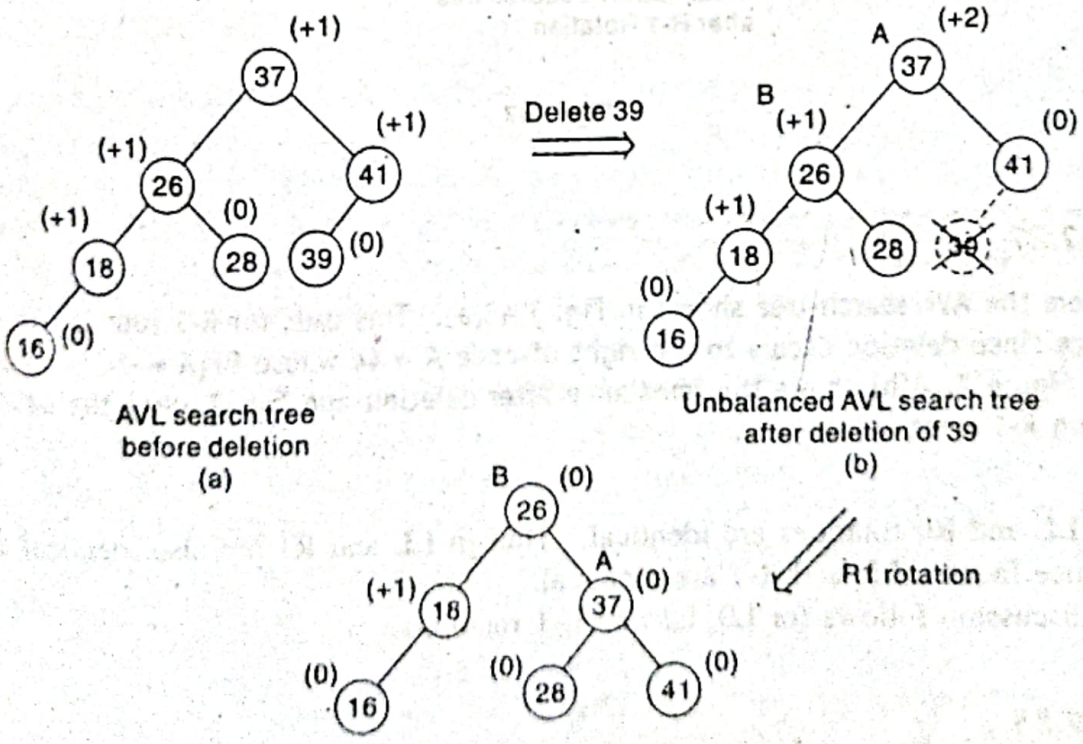


Fig. 7.46

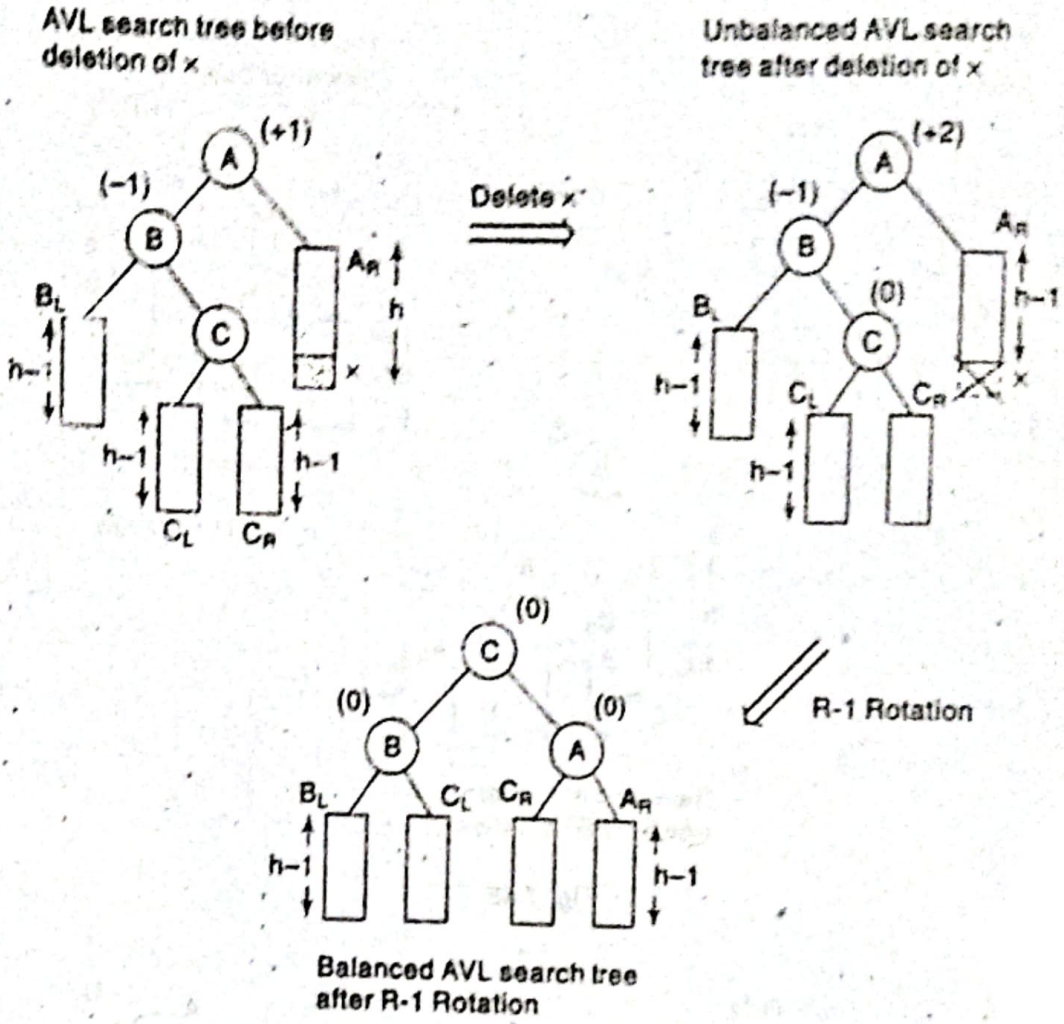


Fig. 7.47

Example 7.27

Delete 52 from the AVL search tree shown in Fig. 7.48(a). This calls for R-1 rotation to set right
the imbalance since deletion occurs to the right of node A = 44 whose BF(A = 44) = +2 and BF(B
= 22) = −1. Figure 7.48(b) shows the imbalance after deletion and Fig. 7.48(c) the balancing of
the tree using R-1 rotation.

Observe that LL and R0 rotations are identical. Though LL and R1 are also identical they yield different balance factors. LR and R-1 are identical.

A similar discussion follows for L0, L1 and L-1 rotations.

Example 7.28

Given the AVL search tree shown in Fig. 7.49(a) delete the listed elements:

120, 64, 130

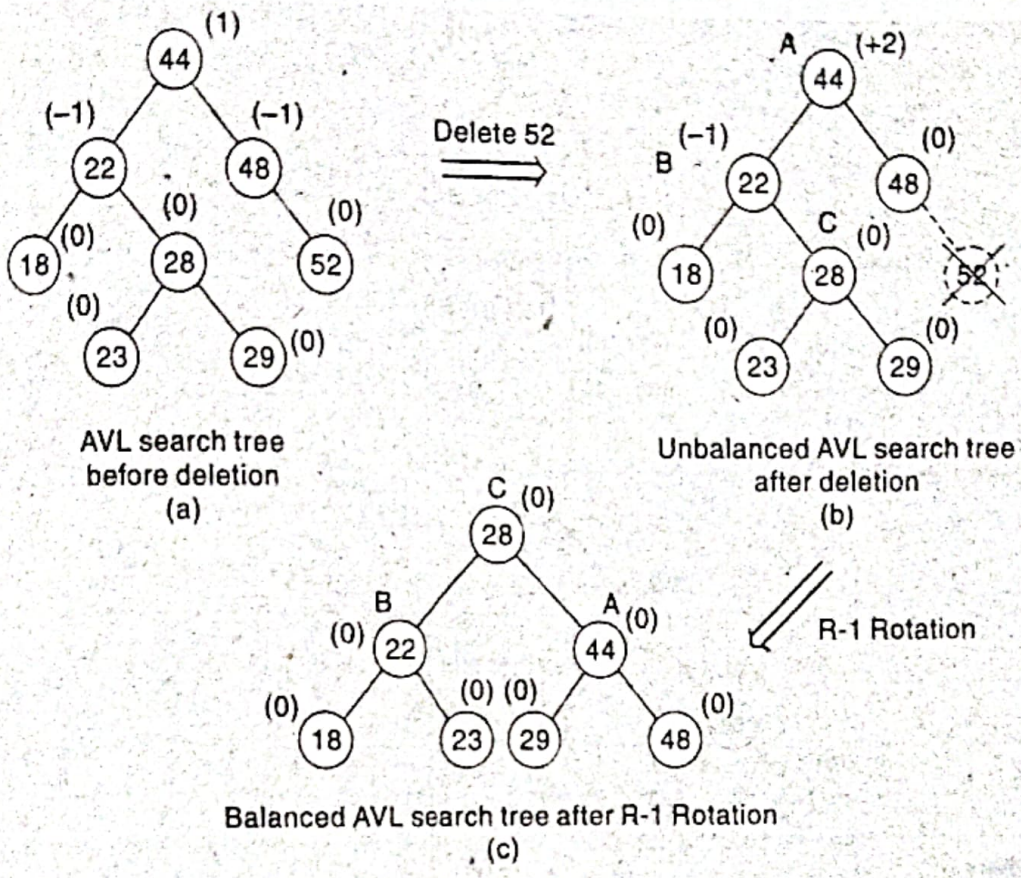


Fig. 7.48

7.15 *m*-WAY SEARCH TREES

All the data structures discussed so far favor data stored in the internal memory and hence support internal information retrieval. However, to favor retrieval and manipulation of data stored in external memory, viz., storage devices such as disks etc., there is a need for some special data structures. *m*-way search trees, B trees and B^+ trees are examples of such data structures which find application in problems such as file indexing.

m-way search trees are generalized versions of binary search trees. The goal of *m*-way search tree is to minimize the accesses while retrieving a key from a file. However, an *m*-way search tree of height h calls for $O(h)$ number of accesses for an insert/delete/retrieval operation. Hence it pays to ensure that the height h is close to $\log_m(n+1)$, because the number of elements in an *m*-way search tree of height h ranges from a minimum of h to a maximum of $m^h - 1$. This implies that an *m*-way search tree of n elements ranges from a minimum height of $\log_m(n+1)$ to a maximum height of n. Therefore there arises the need to maintain balanced *m*-way search trees. B trees are balanced *m*-way search trees.

Definition

An *m*-way search tree T may be an empty tree. If T is non empty, it satisfies the following properties:

(i) For some integer m, known as *order of the tree*, each node is of degree which can reach a maximum of m, in other words, each node has, at most m child nodes. A node may be