Page No.
Date: / /

TREES

Tree is a non-linear data structure and is mainly used to represent data containing a hierarchical relationship between elements for example: records, family, trees and table of contents.

BINARY TREES

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

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

(b) T contains a distinguished node R, called the root of T and the remaining nodes of T form an ordered pair of disjoint binary trees T_1 and T_2.

T_1 and T_2 are called the left and right subtrees of R. If T_1 is nonempty then its root is called the left successor of R. Illy if T_2 is nonempty, then its root is called the right successor of R.

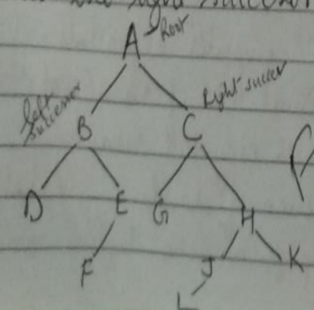


Figure-1

Page No.
Date: / /

NOTE

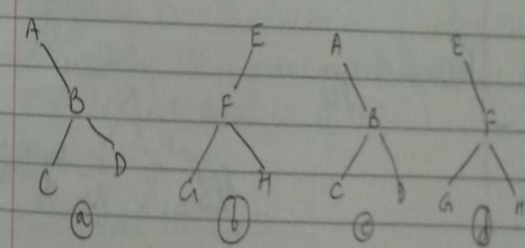
→ Any node N in a binary tree has either 0, 1 or 2 successors.

→ The nodes with no successors are called terminal nodes.

→ Binary trees T and T' are said to be similar if they have same shape.

→ The trees are said to be copies if they have the same contents at corresponding nodes.

for eg:



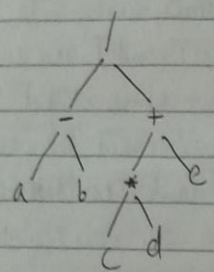
(a), (b), (d) are similar
(a), (c) are copies

Example: Consider any algebric expression E involving only binary operations, such as

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

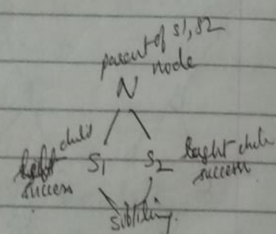
E can be represented by means of the binary Tree T.

Each variable or constant in E appears as an internal node in T whose left and right subtrees corresponds to the operands of the operation

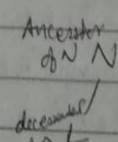


Terminology

–



- Every node N in a binary tree T, except the root has a unique parent called the predecessor of N.
- For graph Theory



→ the line drawn from node N of T to a successor is called an edge

→ A seq. of consecutive edges is called path.

- A terminal node is called a leaf
- A path ending a leaf is called a branch.
- ~~Th~~Each node in a binary tree T is assigned a level no.

→ The root R of the tree T is assigned the level no 0 & every other node is assigned a level no which is 1 more than the level no. of its parent.

- Those nodes with same level no. are said to belong to same generation
- Depth of T : The max no. of nodes in a branch. This turns out to be 1 more than the largest level no. of T.

Complete Binary tree

Consider a binary tree T. Each node of T can have at most two children

∴ level r of T can have at most 2^r nodes

The tree T is said to be complete if all its levels except the last have the max. no. of possible nodes.

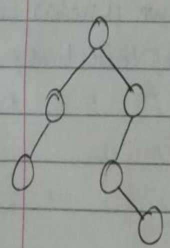
Extended Binary Trees : 2-Trees

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.
In such case, the nodes with 2 children are called internal nodes and nodes with 0 children are called external nodes.

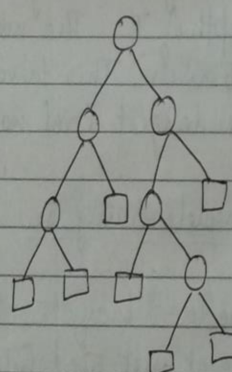
□ → external nodes

○ → internal nodes

Example



(a) Binary Tree T



(b) Extended 2-Tree

Tree corresponding to any Algebric expression E which uses only binary operat is an example of 2-Tree

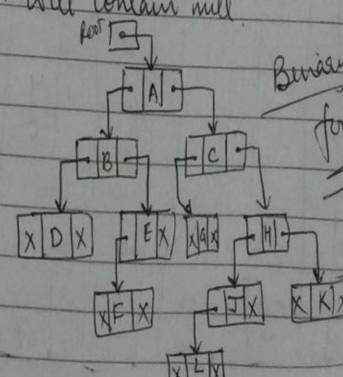
REPRESENTING BINARY TREES IN MEMORY

(1) Linked Representation of Binary Trees

Consider a binary tree T and will be maintained by means of a linked representation which uses three parallel array INFO, LEFT and RIGHT and a pointer variable ROOT.
Each node N of T will correspond to a location K such that :

(i) INFO[K] contains the data at the Node N.
(ii) LEFT[K] " " location of the left child
(iii) RIGHT[K] " " " " right " " node
(iv) ROOT contains the location of the root R of T.
If ROOT = NULL Tree is empty.
If any subtree is empty then corresponding ptr will contain null.

Exam



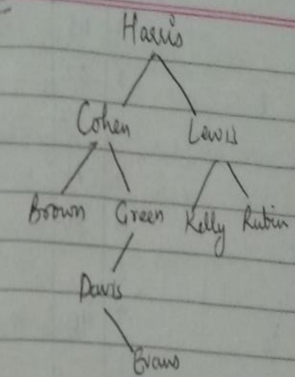
Binary Tree T for figure 1

Example
Suppose the personal file of a small company contains the following data on its nine employees
Name, SSN, Salary
Fig shows how the file may be maintained in memory as a binary tree. draw the tree diagram for the above data by taking only Key value name for nodes

ROOT: 14
AVAIL: 8

	NAME	SSN	Salary	LEFT	RIGHT
1				0	
2	Davis	✓	-	0	12
3	Kelly	✓	-	0	0
4	Green	✓	-	2	0
5			-	1	
6	Brown	✓	-	0	0
7	Lewis	✓	-	3	10
8				11	
9	Cohen	✓	-	6	4
10	Rubin	✓	-	0	0
11				13	
12	Evans	✓	-	0	0
13				5	
14	Harris	✓	-	9	7

Output



(2) Sequential Representation of Binary Trees

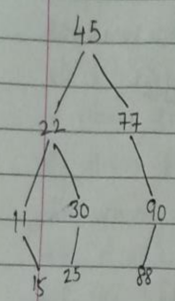
Suppose T is a binary tree & i.e. complete or nearly complete then there is an efficient way of maintaining T in memory called the sequential representation of T. This representation uses only a single array TREE as follows;

(a) The root R of T is stored in TREE[1]
(b) If a node N occupies TREE[K] then its left child is stored in TREE[2*K] and its right child is stored in TREE [2*K+1]

In general, a seq. representation of a tree with depth d will require an array with approx. 2^{d+1} elements

Page No. Date: / /

Soln



(a)

TREE

1	45
2	22
3	77
4	11
5	30
6	
7	90
8	
9	15
10	25
11	
12	
13	
14	88
15	
16	
:	
29	

NOTE:- Observe we require 14 locations in the array TREE even though T has only 9 nodes

Page No. Date: / /

TRAVERSING BINARY TREES

These are three standard ways of traversing a binary tree T with root R. These three algorithms called
- preorder
- inorder
- postorder

PREORDER (NLR traversal) Node-left-Right

1 Process the root R
2 Traverse the left subtree of R in preorder
3 Traverse the right subtree of R in preorder

INORDER (LNR traversal)

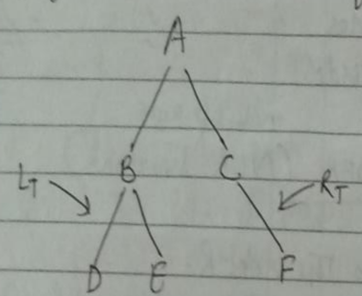
1 Traverse the left subtree of R in inorder
2 Process the root R
3 Traverse the right subtree of R in inorder

POSTORDER (LRN traversal)

1 Traverse the left subtree of R in postorder
2 " " right " " " "
3 Process the root R.

Example :

Consider the binary tree T. A is the root, its left subtree L_T consists of B,D,E its right subtree L_R consists of C,F.



Sol:

① Preorder (NLR)

ABDECF is the preorder traversal of T

② Inorder (LNR)

DBEACF is the inorder traversal of T.

③ Postorder (LRN)

DEBFCA is the postorder traversal of T.

Example :

The binary tree T has 11 nodes. The inorder and ~~postorder~~ preorder Traversals yield the following sequences of nodes

LNR (inorder) D B F E A G C L J H K

NLR (preorder) A B D E F C G H J L K

Draw the tree T.

① Preorder : Root node A

② Inorder : Left child of A → DBFE B is the left child of A

③ Right subtree has GCLJHK C is the right child of A

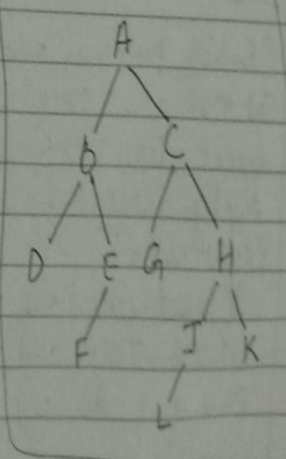
Preorder ABDEFCG~~HJK~~

Inorder (LNR)

DBFEAGCLJHK

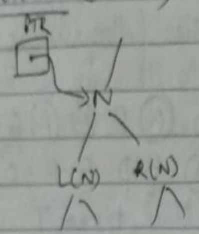
Postorder (LRN)

DFEBGLJKHCA



TRAVERSAL ALGORITHMS USING STACKS

PREORDER TRAVERSAL (NLR)



PTR contains the location of the Node N currently being scanned.
STACK → an array to hold the add. of nodes for future processing

Algo :- PREORD (INFO, LEFT, RIGHT, ROOT)

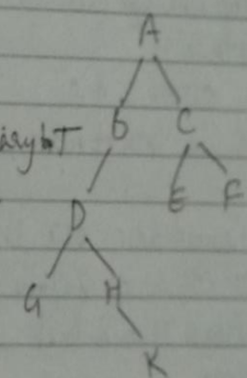
1 [Initially push NULL onto STACK and initialize PTR]
Set TOP := 1, STACK[1] := NULL and PTR := ROOT
2 Repeat steps 3 to 5 while PTR ≠ NULL
3. Apply PROCESS to INFO[PTR]
4. [Right child?]
If RIGHT[PTR] ≠ NULL then [Push on STACK]
set TOP := TOP+1 and STACK[TOP] := RIGHT[PTR]
[End of if structure]

5. [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 2 loop]

6 Exit

Example

Consider the binary T & apply the algo.



1 Initially push NULL onto STACK:
STACK: ∅
Then set PTR := A, the root of T
2 Proceed down the left-most path rooted at PTR = A as follows:
(i) Process A and push its right child C onto STACK

STACK: ϕ, C

(ii) Process B (There is no right child)

(iii) Process D & push its right child H on STACK

STACK: ϕ, C, H

(iv) Process G. (There is no right child)

No other node is processed, since G has no left child.

3. [Backtracking]

Pop the top element H from stack & set PTR := H

STACK: ϕ, C

Since PTR ≠ NULL return to step (a) of algo

4. Proceed down the left most path rooted at PTR = H as follows.

(v) Process H and push its right child K onto STACK.

STACK: ϕ, C, K

No other node is processed, since H has no left child.

5. [Backtracking] Pop K from STACK and set PTR := K. This leaves STACK: ϕ, C

Since PTR ≠ NULL, return to step (a) of algo

6. Proceed down the left most path rooted at PTR = K as follows.

(vi) Process K

No other node is processed, since K has no left child.

7. [Backtracking] Pop C from STACK & set PTR := C

STACK: ϕ

Since PTR ≠ NULL return to step (a) of the algo

8. Proceed down the left most path rooted at PTR = C as follows.

(vii) Process C & push right child F onto STACK

STACK: ϕ, F

(viii) Process E

9. [Backtracking] Pop F from STACK & set PTR := F

STACK: ϕ

Since PTR ≠ NULL return to step (a) of the algo

10. Proceed down the left most path rooted at PTR = F

(ix) Process F

11. [Backtracking] Pop the top element NULL from STACK and set PTR := NULL. Since PTR = NULL the algo is completed

INORDER TRAVERSAL (LNR)

Algo: Initially push NULL onto STACK & then set PTR := ROOT. Then repeat the following steps until NULL is popped from STACK.

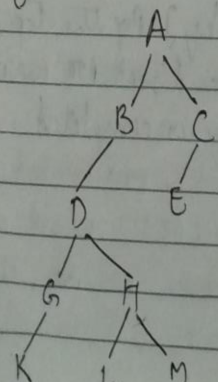
(a) Proceed down the left most path rooted at PTR pushing each node N onto STACK and stopping when a node N with no left child is pushed onto STACK.

(b) [Backtracking] Pop and process the nodes on STACK. If NULL is popped, then Exit. If a node N with a right child R(N) is processed set PTR = R(N) & return to step (a)

Example: Consider a binary tree T. Simulate the inorder algo with T, showing the contents of STACK.

Inorder

KGDLHMBAEC



Solution

1. Initially push NULL onto STACK:
   STACK: ϕ
   Then set PTR := A, the root of T.
2. Proceed down the left-most path rooted at PTR = A, pushing the nodes A, B, D, G, K onto STACK
   STACK: ϕ, A, B, D, G, K
3. [Backtracking] The nodes K, G, D are popped and processed
   STACK: ϕ, A, B
   (Stop the processing at D as it has a right child)
   The set PTR := H
4. Proceed down the left most path rooted at PTR = H, pushing the nodes H and L onto STACK
   STACK: ϕ, A, B, H, L
5. [Backtracking] The nodes L & H are popped & processed
   STACK: ϕ, A, B
   Then Set PTR := M
6. Proceed down leftmost path rooted at PTR = M pushing M onto STACK
   STACK: ϕ, A, B, M
7. [Backtracking] The nodes M, B, A popped & processed
   STACK: ϕ
   Set PTR := C

8. Proceed down the left-most path rooted at PTR = C, pushing C & E.
STACK: ϕ, C, E
9. [Backtracking] Node E is popped & processed. Since E has no right child, node C is popped and processed. Since C has no right child the next element NULL is popped from STACK

Formal presentation of inorder traversal algo

Algo INORD (INFO, LEFT, RIGHT, ROOT)

1. [Push NULL onto STACK and initialize PTR]
Set TOP := 1, STACK[1] := NULL & PTR := ROOT
2. Repeat while PTR ≠ NULL [Pushes left-most path onto STACK]
(a) Set TOP := TOP+1 & STACK[TOP] := PTR
(b) Set PTR := LEFT[PTR]
[End of loop]
3. Set PTR := STACK[TOP] & TOP := TOP-1
4. Repeat steps 5 to 7 while PTR ≠ NULL. [Backtrack]
5. Apply PROCESS to INFO[PTR]
6. [Right child ?] If RIGHT[PTR] ≠ NULL then
(a) Set PTR := RIGHT[PTR]
(b) Goto step 3.
[End of If structure]
7. Set PTR := STACK[TOP] & TOP := TOP-1
[End of step 4 loop]
8. Exit

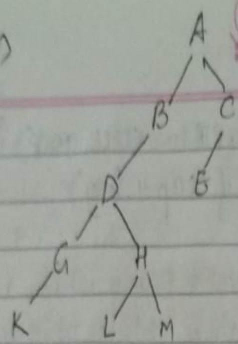
POSTORDER TRAVERSAL

Algorithm :- Initially push NULL onto STACK & then set PTR := ROOT Then repeat the following steps until NULL is popped from STACK.

(a) Proceed down the leftmost path rooted at PTR. At each node N of the path, push N onto STACK and if N has a right child R(N), push -R(N) onto STACK

(b) [Backtracking] Pop and process positive nodes on STACK. If NULL is popped then Exit. If a negative node is popped i.e if PTR = -N for some node N, set PTR = N & return to step (a)

Example (LRN)



KGLMHDBECA

1 Initially, push NULL onto STACK & set PTR:=A
~~to~~ STACK: ϕ

2. Proceed down the left-most path rooted at PTR=A, pushing the node B,D,G,K onto STACK
Since A has a right child C push -C onto stack after A but before B & since D has a right child H push -H onto stack after D but before G. This yields
STACK: ϕ, A, -C, B, D, -H, G, K

3. [BackTracking] Pop and process K then G. Since -H is -ve only pop -H
STACK: ϕ, A, -C, B, D
Now PTR:=-H, Reset PTR=H & return to step(a)

4. Proceed down the left-most path rooted at PTR=H. Push H. Since H has a right child M, push -M onto the STACK. Then push L.
STACK: ϕ, A, -C, B, D, H, -M, L

(5) [Backtracking] Pop and process L but only pop -M
STACK: ϕ, A, -C, B, D, H
Now PTR:=-M, Reset PTR:=M & return to step(a)

(6) Proceed down the left most path rooted at PTR=M. Push M
STACK: ϕ, A, -C, B, D, H, M

(7) [Backtracking] Pop & process M, H, D, B but only pop -C
STACK: ϕ, A
Now PTR=-C, Set PTR:=C & return to Step(a)

(8) Process down the left most path rooted at PTR=C & push C. & then E
STACK: ϕ, A, C, E

(9) [Backtracking] Pop & process E, C, A. When NULL is popped STACK is empty & algo is completed.

Formal presentation of POSTORDER traversal algo

Algo: POSTORD (INFO, LEFT, RIGHT, ROOT)

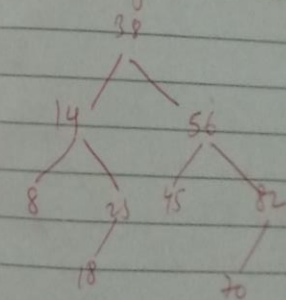
1. Set TOP := 1, STACK[1] := NULL & PTR := ROOT
2. [Push left-most path onto STACK]
   Repeat Steps 3 to 5 while PTR ≠ NULL
3. Set TOP := TOP+1 & STACK[TOP] := PTR
   [Pushes PTR on STACK]
4. If RIGHT[PTR] ≠ NULL then
   Set TOP := TOP+1 & STACK[TOP] := -RIGHT[PTR]
   [End of If structure]
5. Set PTR := LEFT[PTR]
   [End of step 2 loop]
6. Set PTR := STACK[TOP] & TOP := TOP-1
7. Repeat while PTR>0
   (a) Apply PROCESS to INFO[PTR]
   (b) Set PTR := STACK[TOP] & TOP := TOP-1
   [End of loop]
8. If PTR<0 then
   (a) Set PTR := -PTR
   (b) Goto step 2
   [End of if structure]
9. Exit

BINARY SEARCH TREES

The average running time $f(n) = O(\log_2 n)$ for binary search tree

Suppose T is a binary tree. Then T is called a binary search tree or binary sorted tree if each node N of T has the following property: The value of N is greater than every value in the left subtree of N and is less than every value in the right subtree of N.

Eg



Searching and Inserting in binary search trees

Suppose an ITEM of information is given. The following algo. finds the location of ITEM in the binary search tree T or inserts ITEM as a new node in its appropriate place in the tree.

(a) Compare ITEM with the root node N of the tree

(i) If ITEM < N, proceed to the left child of N

(ii) If ITEM > N, proceed to the right child of N.

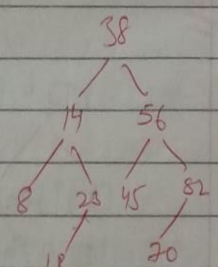
(b) Repeat step (a) until one of the following occurs:

(i) We meet a node N such that ITEM = N. In this case the search is successful

(ii) We meet an empty subtree, which indicates that the search is unsuccessful, and we insert ITEM in place of the empty subtree

Example

Consider the binary tree T given. Suppose ITEM = 20 is given ie to be searched.



Sol (1) Compare ITEM = 20 with root 38 since 20 < 38 proceed to the left child of 38 which is 14.

(2) Compare since 20 > 14, proceed to the right child of ~~38~~14 ie 23.

(3) Compare 20 < 23 proceed to the left child of 23 i.e 18

(4) Compare since 20 > 18 & 18 does not have a right child, insert 20 as a right child of 18.

Procedure 1: FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PTR)

LOC → location of ITEM in T

PAR → location of the parent of ITEM

Special cases

(i) LOC = NULL & PAR = NULL → tree is empty

(ii) LOC ≠ NULL & PAR = NULL → ITEM is the root of T

(iii) LOC = NULL & PAR ≠ NULL → ITEM is not in T & can be added to T as a child of node N with location PAR

1. [Tree empty?]
   If ROOT = NULL then set LOC := NULL & PAR := NULL
   Return
2. [ITEM at root?]
   ~~If ROOT ≠ NULL then~~ If ITEM = INFO[ROOT] then
   Set LOC := ROOT & PAR := NULL & Return
3. [Initialize pointers PTR & SAVE]
   If ITEM < INFO[ROOT] then
   Set PTR := LEFT[ROOT] & SAVE := ROOT
   Else
   Set PTR := RIGHT[ROOT] & SAVE := ROOT
   [End of If structure]
4. Repeat Steps 5 & 6 while PTR ≠ NULL
5. [ITEM FOUND?]
   If ITEM = INFO[PTR] then set
   LOC := PTR & PAR := SAVE & Return

6. If ITEM < INFO[PTR] then
SAVE := PTR & PTR := LEFT[PTR]
Else
Set SAVE := PTR & PTR := RIGHT[PTR]
[End of If structure]
[End of loop]
7. If LOC = NULL & PAR := SAVE WRITE "Unsucc"
8. Exit

Algorithm: INSBST (INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM, LOC)

1. Call FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)
2. If LOC ≠ NULL then Exit
3. [Copy ITEM into new node in AVAIL list]
(a) If AVAIL = NULL then Write: OVERFLOW & Exit
(b) Set NEW := AVAIL, AVAIL := LEFT[AVAIL] and INFO[NEW] := ITEM
(c) Set LOC := NEW, LEFT[NEW] := NULL & RIGHT[NEW] := NULL.
4. [Add ITEM to tree]
If PAR = NULL then
Set ROOT := NEW
Else if ITEM < INFO(PAR) then
Set LEFT[PAR] := NEW
Else
Set RIGHT[PAR] := NEW
[End of If structure]
5. Exit

Complexity of the searching algo
$f(n) = O(\log_2 n)$

Applications of Binary Search tree

Consider a collection of n data items, $A_1, A_2, \ldots A_N$. Suppose we want to find & delete all duplicates in the collection.

AlgoA: Scan the elements from A_1 to A_N (from left to right)
(a) For each element A_K compare A_K with $A_1, A_2, \ldots, A_{K-1}$ i.e. compare A_K with those elements which precede A_K.
(b) If A_K does occur among $A_1, A_2, \ldots A_{K-1}$ then delete A_K.

No. of comparisons $f(n) = 0+1+2+3+\cdots+(n-2)+(n-1) = \frac{(n-1)n}{2} = O(n^2)$

Algo B :- Build a binary search tree T using the elements $A_1, A_2, \ldots A_N$. In building the tree, delete A_K from the list whenever the value of A_K already appears in the tree.

Advantage of Algo B :- Each element A_K is compared only with the elements in a single branch of tree

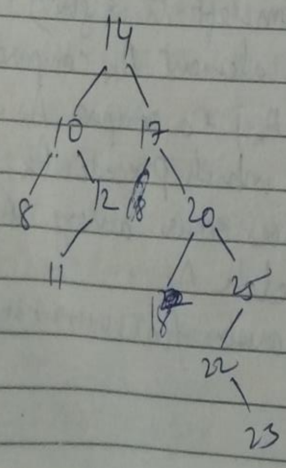
$\therefore f(n) = O(n \log_2 n)$

Example :-

Consider again the following list of 15 no.

14, 10, 17, 12, 10, 11, 20, 12, 18, 25, 20, 8, 22, 11, 23.

Apply Algo B & we obtain the tree



Exact no. of comp

$0 + 1 + 1 + 2 + 2 + 3 + 2 + 3 + 3 + 3 + 3 + 2 + 4 + 4 + 4 + 5 = 38$

On the other hand algo A requires

$= 72$

DELETING IN A BINARY SEARCH TREE

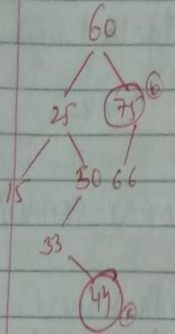
Deletion algo first uses Procedure 1 to find the location of node N which contains ITEM and also the location of the parent node P(N). The way N is deleted from the tree depends primarily on the no. of children of node N.

Case 1 N has no children. Then N is deleted from T by simply replacing the location of N in the parent node P(N) by the 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) denotes the inorder successor of N. Then N is deleted from T by first deleting S(N) from T (by using Case 1 or 2) & then replacing node N in T by the node S(N).

Example Consider binary search tree T & suppose T appears in memory as



ROOT $\boxed{3}$

AVAIL $\boxed{5}$

	INFO	LEFT	RIGHT
1	33	0	9
2	25	8	10
3	60	2	7
4	66	0	0
5		6	
6		0	
7	75	4	0
8	15	0	0
9	44	0	0
10	50	1	0

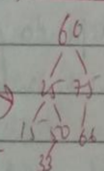
ⓐ ROOT $\boxed{3}$ AVAIL $\boxed{9}$

1	33	0	0
2	25	8	10
3	60	2	7
4	66	0	0
5		0	
6		0	
7	75	4	0
8	15	0	0
9		5	
10	50	1	0

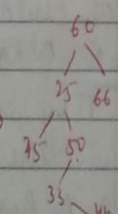
ⓑ ROOT $\boxed{3}$ AVAIL $\boxed{7}$

1	33	0	9
2	25	8	10
3	60	2	4
4	66	0	0
5		6	
6		0	
7		5	
8	15	0	0
9	44	0	0
10	50	1	0

ⓐ Delete node 44 from T

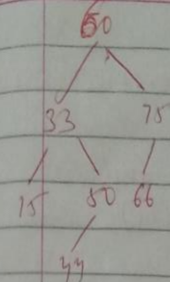


ⓑ Delete node 75 from T



ⓒ Delete node 25 from T

— Delete 33 from tree & then replacing node 25 by node 33. Only change the pointers not to move the contents of the node.



ROOT $\boxed{3}$

AVAIL $\boxed{2}$

1	33	8	10
2		5	
3	60	1	7
4	66	0	0
5		6	
6		0	
7	75	4	0
8	15	0	0
9	44	0	0
10	50	9	0

Procedure 2: CASEA (INFO, LEFT, RIGHT, ROOT, LOC, PAR)

This procedure deletes the node N at the location with Case 1 & 2 when Node N either has no children or only one child

PAR → gives location of the parent of N or else PAR = NULL indicates that N is the root node.

CHILD → gives the location of the only child of N or else CHILD = NULL indicates N has no children.

Page No.
Date: / /

1 [Initialize CHILD]
If LEFT[LOC] = NULL & RIGHT[LOC] = NULL then
Set CHILD := NULL
Else if LEFT[LOC] ≠ NULL then
Set CHILD := LEFT[LOC]
Else
Set CHILD := RIGHT[LOC]
[End of If structure]

2 If PAR ≠ NULL then
If LOC = LEFT[PAR] then
Set LEFT[PAR] := CHILD
Else
Set RIGHT[PAR] := CHILD
[End of If structure]
Else
Set ROOT := CHILD
[End of If structure]
3 Return

Page No.
Date: / /

Procedure 3 : CASEB(INFO, LEFT, RIGHT, ROOT, LOC, PAR)
This procedure deletes the node N at the location LOC where N has two children
PAR → location of parent of N
SUC → location of the inorder successor of N
PARSUC → location of the parent of the inorder successor

1 [Find SUC and PARSUC]
(a) Set PTR := RIGHT[LOC] & SAVE := LOC
(b) Repeat while LEFT[PTR] ≠ NULL
Set SAVE := PTR & PTR := LEFT[PTR]
[End of loop]
(c) Set SUC := PTR & PARSUC := SAVE
2 [Delete inorder successor using Procedure 2]
Call CASEA (INFO, LEFT, RIGHT, ROOT, SUC, PARSUC)
3. [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]

3. Set LEFT[SUC] := LEFT[LOC] &
RIGHT[SUC] := RIGHT[LOC]
4. Return.

Algo : DEL(INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM)

ITEM → info i.e. to be deleted.

1 [Find the locations of ITEM & its parent using procedure 1]
Call FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)

2 [ITEM in tree?]
If LOC = NULL then: Write: 'ITEM not in tree & Exit

3 [Delete node containing ITEM]
If RIGHT[LOC] ≠ NULL & 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]

4. [Return deleted node to the AVAIL list]
Set LEFT[LOC] := AVAIL & AVAIL := LOC

5 Exit

AVL SEARCH TREES

In many applications insertion & deletions occur continually, with no predictable order. In some app. it is imp. to optimize search times by keeping the tree very nearly balanced at all times. This goal is achieved by using AVL trees described by G.M. ADEL-SON-VEL-SKII & E.M. LANDS

Def: An AVL tree is a binary search tree in which heights of the left & right subtrees of the root differ by at most 1 & in which the left & right subtrees are again AVL trees

Def:- An empty binary tree is an AVL tree. A non empty binary tree T is an AVL tree iff given T^L & T^R to be the left & right subtrees of T & $h(T^L)$ & $h(T^R)$ are the heights of left & right subtrees & $|h(T^L) - h(T^R)| \leq 1$

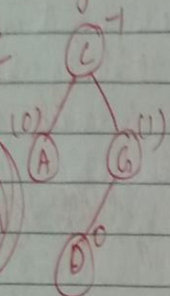
$h(T^L) - h(T^R) \rightarrow$ balance factor (BF)

& for an AVL tree the balance factor of a node can be either 0, 1 or -1.

Representation of an AVL Search Tree

AVL search trees like binary search trees are represented using a linked representation. Every node registers its balance factor

Examples of AVL trees

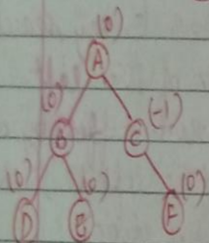


C $1-2=-1$

A equal $=0$

G $1-0=1$

(D) equal $=0$

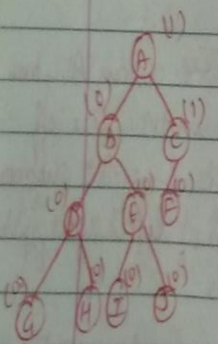


A $2-2=0$

B 0

C $0-1=-1$

(D), (E), (F) $=0$

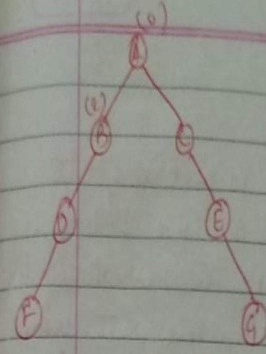


A $3-2=1$

B 0

C $1-0=1$

D, E, F, G, H, I, J $=0$



A $=0$

B $2-0=2$

Ex: NON AVL Tree

Insertion in An AVL Search Tree

1. Inserting an element into an AVL search tree in its first phase is similar to that of the one used in binary search tree Procedure 1

2. However, if after insertion of the element the BF of any node in the tree is affected so as to render the binary search tree unbalanced, a technique called Rotation is used to restore the balance of the search tree.

fig 1 For eg Insertion of E after D makes the the tree unbalanced since BF(C) $1-3=2$

Rotations

→ To perform rotation, it is necessary to identify a specific node A whose BF(A) is ≠ -1,0,1 & which is the nearest ancestor to the inserted node on the path from the inserted node to the root.

→ This implies that all nodes on the path from the inserted node to A will have their balance factor to be either 0,1,-1

The rebalancing rotations are classified as LL, LR, RR, RL

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

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

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

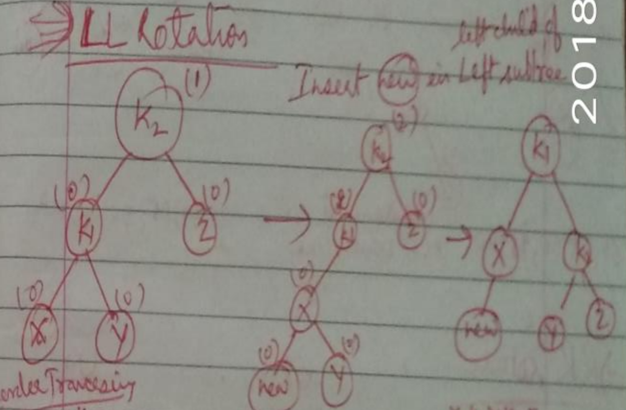
RL rotation : Inserted node is in the left subtree of right subtree of node

CS-study-blogspot.in/2012/11/cases-of-rotation-of-avl-tree.html

In case of LL & RR, single rotation can fix the balance

In case of LR & RL, single rotation can not fix the balance

⇒ LL Rotation



Inorder Traversing
XK1YK2Z

XK1YK2Z

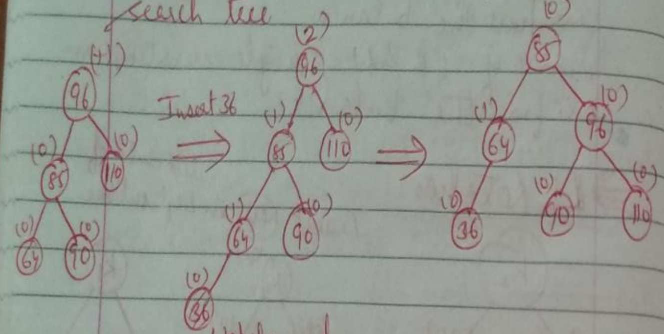
Rotation

→ K_2 node has to rotated single time towards right to become the right child of K_1 & Y has become the ~~right~~ left child of K_2

Note If we traverse the tree in the inorder fashion it will remain same

Example

Insert 36 into the following AVL search tree

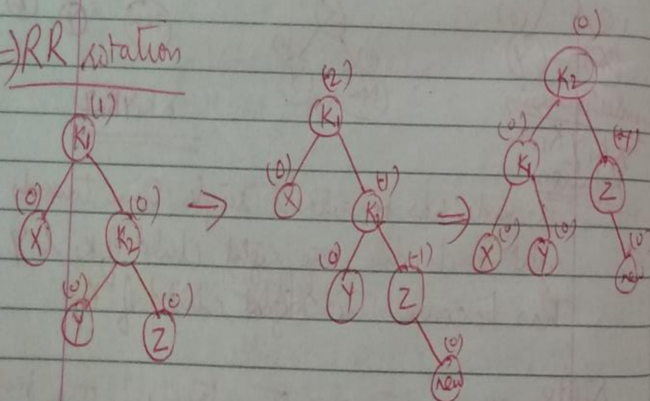


64, 85, 90, 96, 110

Unbalanced AVL search tree

36, 64, 85, 90, 96, 110

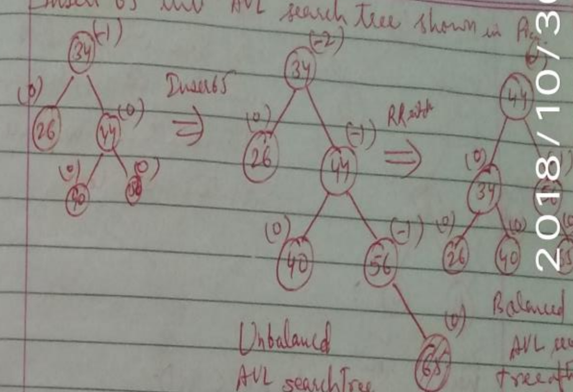
⇒ RR rotation



K_1 is rotated once towards left. Node Y becomes the right child of node of node K_1

Example

Insert 65 into AVL search tree shown in Fig.

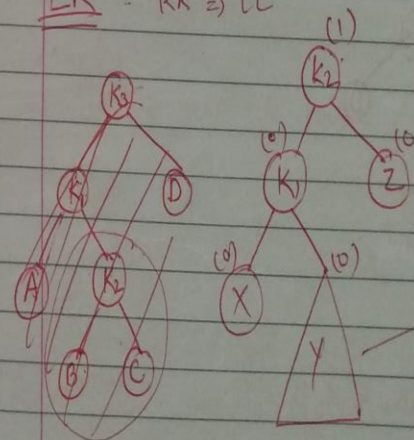


Unbalanced AVL search tree

Balanced AVL search tree after RR rotation

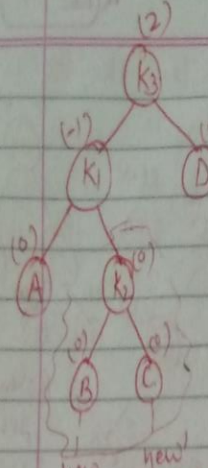
⇒ LR and RL Rotations

LR - RR ⇒ LL



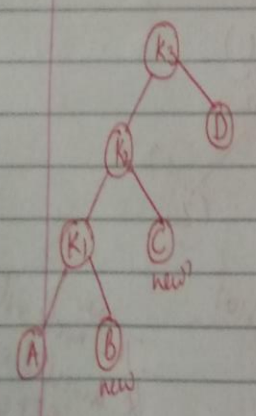
The new node will be inserted either in the left of Y or right of Y

If we turn k_1 into root, the tree will be still unbalanced. The only alternative is to place K_2 as the new root.



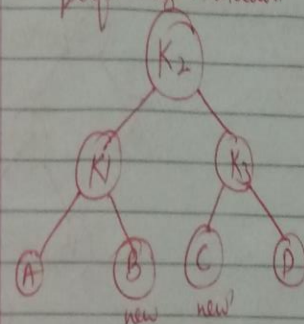
Inorder $AK_1BK_2CK_3D$

Step 1. Do left rotation b/w K_1 & K_2



Inorder AK_1BK_2,C,K_3,D

Step 2 - To make K_2 as root perform right rotation

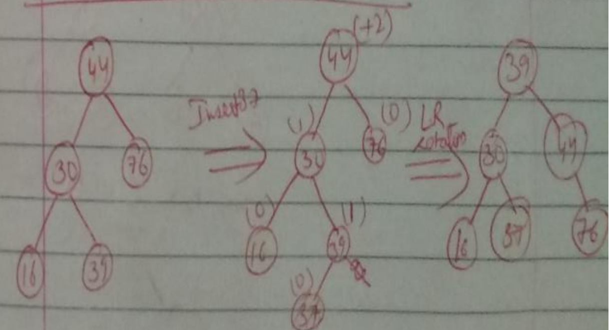


Inorder $AK_1BK_2CK_3D$

RL double rotation

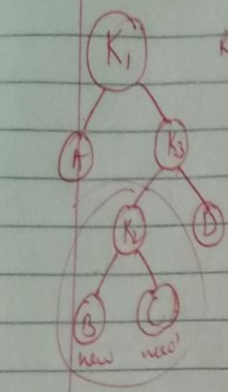
Example

Insert 37 use LR double rotation.

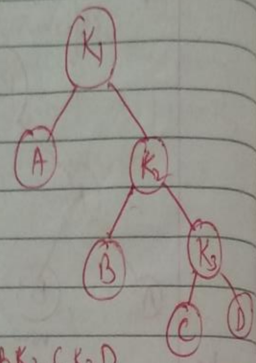


2018/10/30 09:28

⇒ RL double rotation



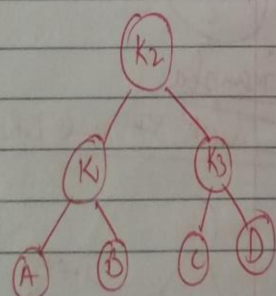
Rotate Right ⇒



AK₁ BK₂ C K₃ D

AK, BK2 C K3 D

Left rotation

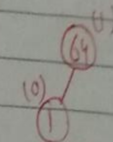


AK, B K2 C K3 D.

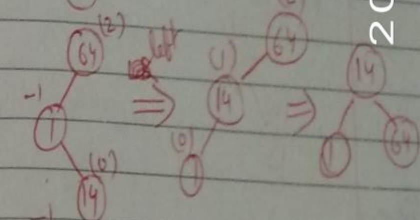
Example Construct an AVL search tree by inserting the following elements in the order of their occurrence

64, 1, 14, 26, 13, 110, 98, 85

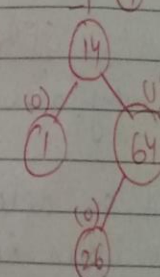
(1) Insert 64, 1



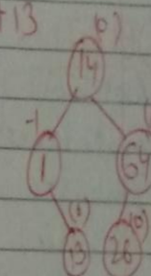
(2) Insert 14 LR



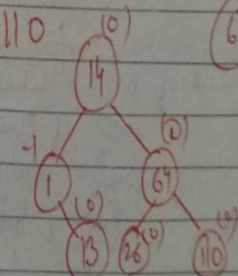
(3) Insert 26



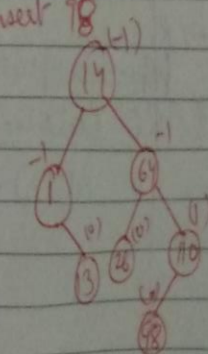
(4) Insert 13



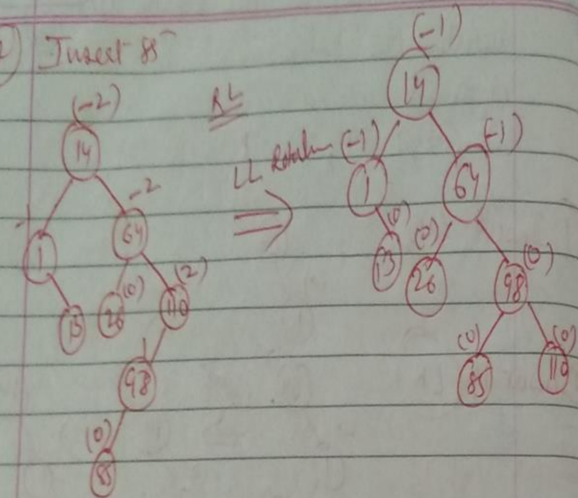
(5) Insert 110



(6) Insert 98



7) Insert 85



DELETION IN AN AVL SEARCH TREE

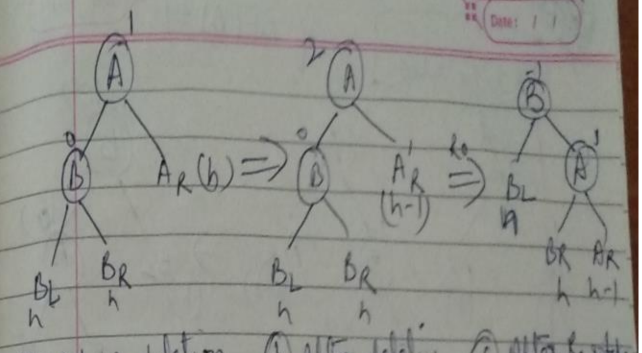
→ Deletion of a node may also produce an imbalance

→ Imbalance incurred by deletion is classified into the types R_0, R_1, R_{-1}, L_0, L_1, L_{-1}

→ Rotation is also needed for rebalancing. Depending upon the value of BF(B) where B is the root of the left or right subtree of A, R & L imbalance is further classified

R0 Rotation

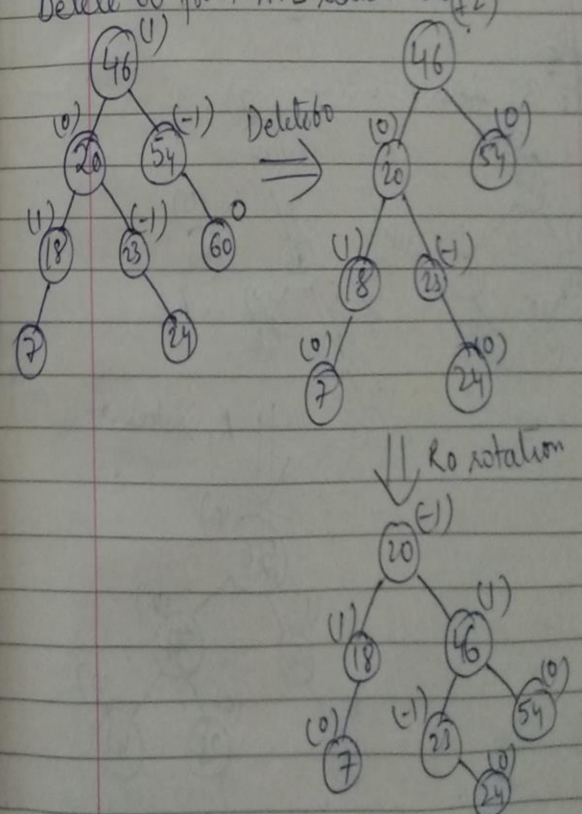
If $BF(B) = 0$



(a) Before deletion (b) After deleting from A_R (c) After R_0 rotation

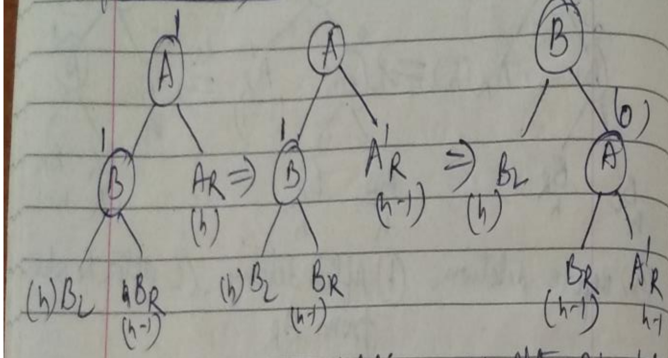
Example

Delete 60 from AVL search tree



R1 Rotation

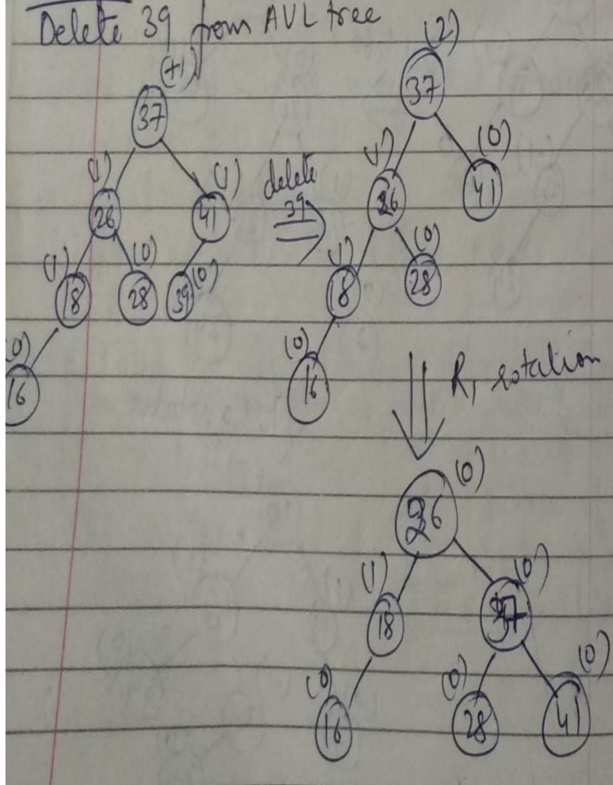
If $BF(B) = 1$



(a) Before deletion (b) After deleting from A_R (c) After R1 rotation

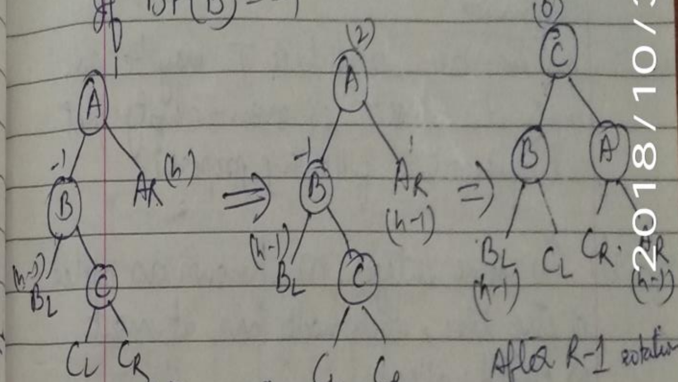
Example

Delete 39 from AVL tree



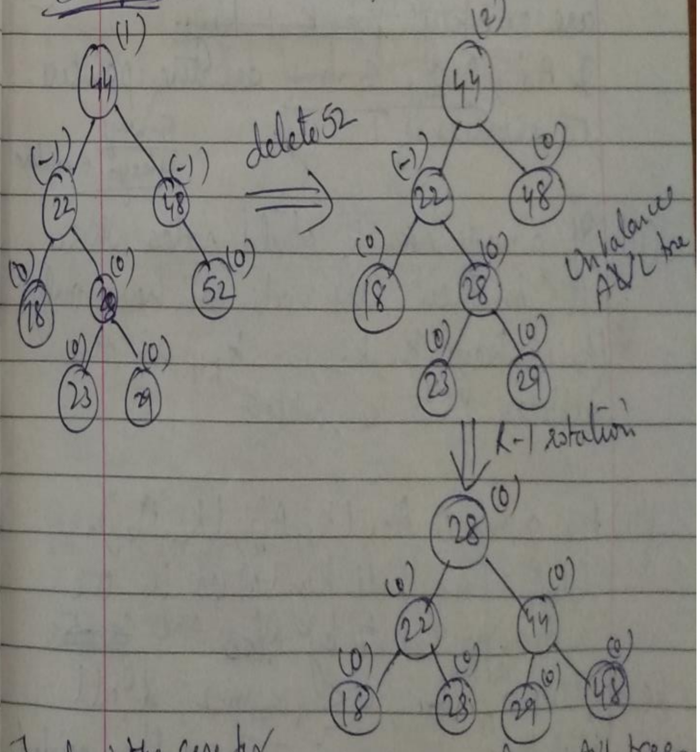
R-1 rotation

If $BF(B) = -1$



(a) Before deletion (b) After deleting from A_R After R-1 rotation

Example Delete 52 from AVL search tree



Similar is the case for L0, L1, L-1

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 has at most m child nodes
A node may be represented as $A_0, (K_1, A_1), (K_2, A_2) \ldots (K_{m-1}, A_{m-1})$ where $K_i, 1 \le i \le m-1$ are the keys ~~i.e K-1 keys~~
& $A_i, 0 \le i \le m-1$ are the pointers to subtrees of T.

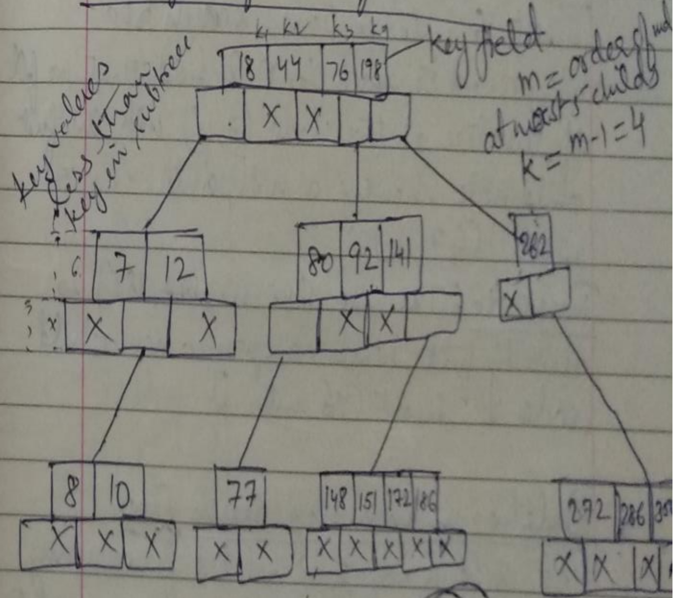
5-way
4 keys 5 pointers

(ii) If a node has k child nodes where $k \le m$ then the node can have only $(k-1)$ keys $k_1, k_2, \ldots k_{k-1}$ in the subtrees into k subsets

(iii) For a node $A_0, (K_1, A_1), (K_2, A_2) \ldots (K_{m-1}, A_{m-1})$, all key values in the subtree pointed to by A_i are less than the key K_{i+1}, $0 \le i \le m-2$, & all the key values in the subtree pointed to by A_{m-1} are greater than K_{m-1}.

(iv) Each of the subtrees $A_i, 0 \le i \le m-1$ are also m-way search trees

Example of 5-way search Tree



Searching

It is same as of binary search trees

Eg search for 77
- Begin at root $77 > 76 > 44 > 18$ move to 4^{th} subtree $77 < 80$ move to 1^{st} subtree
Search succesful

Search for 13
$13 < 18$ move to 1^{st} subtree
$13 > 12$ move down the third subtree but it is null
Search unsuccessful

Insertion in an m-way search tree

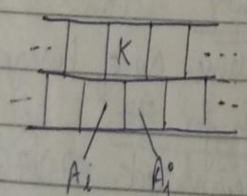
- To insert a new element into an m-way search tree we proceed in the same way one would in order to search for the element

Insert 6 into a 5-way search tree.
→ We proceed to search for 6 & find that we fall of the tree at the node [7,12] with the first child node showing a null pointer ∴ we insert [6,7,12]

Insert 146 : The node [148, 151, 172, 186] is already full, hence we open a new child node & insert 146 into it.

Deletion in an m-way search tree

Let K be a key to be deleted from m-way search tree. To delete the key, first search for the key.



K: Key
Ai, Aj: pointers to subtrees

If ($A_i = A_j$ = NULL) then delete K

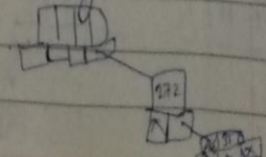
If ($A_i \neq$ NULL, A_j = NULL) then choose the largest key element K' in child node pointed to by A_i, delete the key K' & replace K by K'.

If (A_i = NULL, $A_j \neq$ NULL) then choose the smallest key elements K'' from the subtree pointed to by A_j, delete K'' & replace K by K''

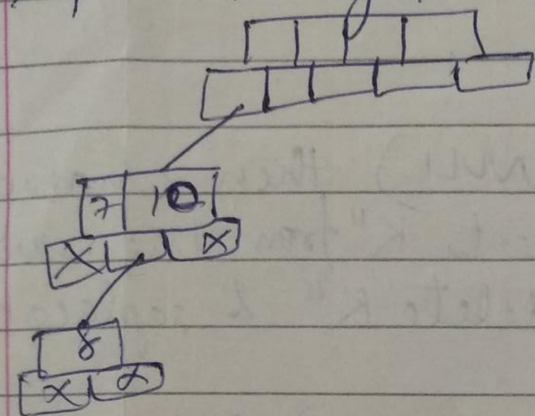
If ($A_i \neq$ NULL, $A_j \neq$ NULL) then choose either the largest of the key elements K' in the subtree pointed to by A_i or the smallest of the key elements K'' from the subtree pointed to by A_j to replace K.

Example ①: To delete 151, we observe that it is present in leaf node [148, 151, 172, 186] with left & right subtree pointers = NULL. Delete 151 & node becomes [148, 172, 186]. Deletion of 92 is similar.

② To delete 262 : (A_i = NULL & $A_j \neq$ NULL)
Choose smallest element 272 from [272, 286, 350]
delete 272, Replace 262 by 272

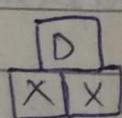


To delete 12 : from [7,12] $A_j \neq$ NULL
find largest element i.e 10 & delete 10 &
replace 12 by 10

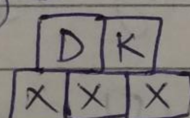


Example Construct a 3-way search tree out of empty search tree with the following keys
D, K, P, V, A, G
1 2 3

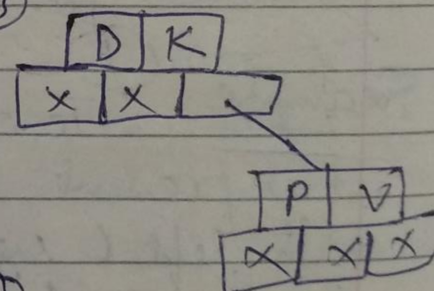
1



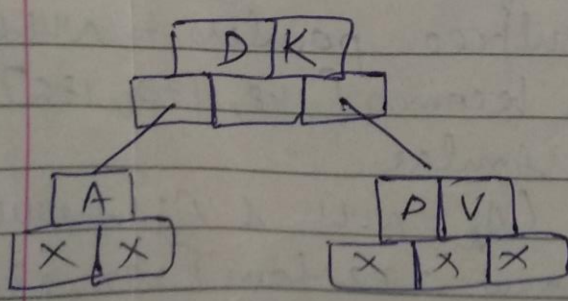
2



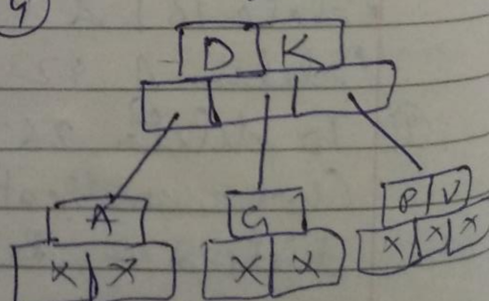
3



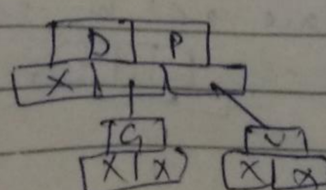
3



4



Delete A & K



Date ....
Page ....

B TREES

m-way search trees have the advantage of minimizing file accesses due to their restricted height. However it is essential that the height of the tree be kept as low as possible & so there arises the need to maintain balanced m-way search tree. Such a balanced m-way search tree is known as B-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.

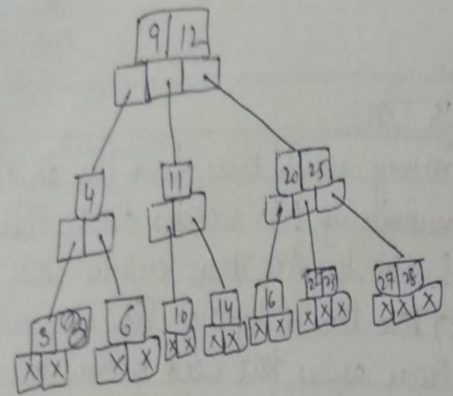
(ii) the internal nodes except the root have at least $\lceil \frac{m}{2} \rceil$ child nodes & at most m child nodes.

(iii) The no. of keys in each internal node is one less than the no. of child nodes and these keys partition the keys in the subtree of the node in a manner similar to that of m-way search trees.

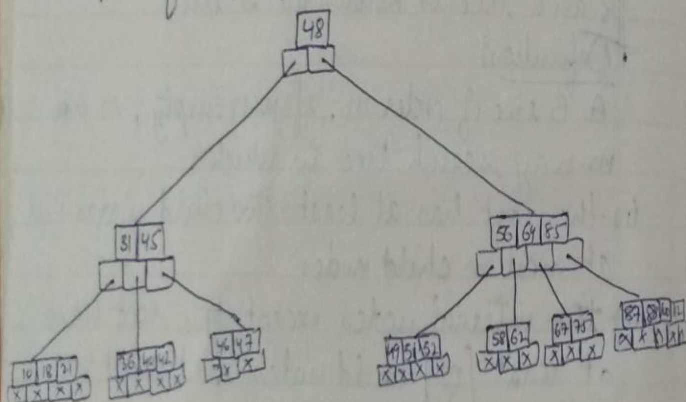
(iv) All leaf nodes are on the same level.

NOTE: A B tree of order 3 is referred to as 2-3 tree since the internal nodes are of degree 2 or 3 only.

Example of 2-3 tree



Example of BTree of order 5



Searching a B-tree

→ It is similar to the m-way search tree.

Insertion in B-tree

→ The insertion proceeds as if one were searching for the key in the tree. When the search terminates in a failure at a leaf node & tends to fall off the tree, the key is inserted acc. to the following procedure.

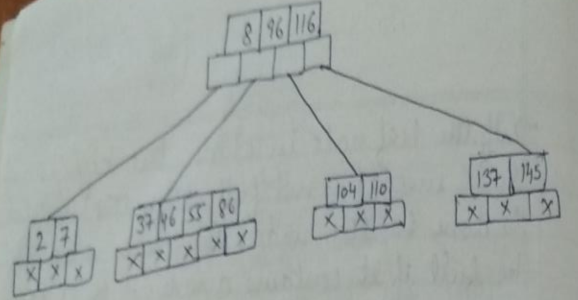
Date ....
Page ....

→ If the leaf node in which the key is to be inserted is not full, then the insertion is done in the node. A node is said to be full if it contains a max of (m-1) keys where m is the order of B-tree

→ If the node were to be full, then insert the key in order into the existing set of keys in the node, split the node at its median into two nodes at the same level pushing the median element up by one level. Note that the split nodes are only half full. Accomodate the median element in the parent node if it is not full. Otherwise repeat the same procedure & this may even call for rearrangement of the keys in the root node or the formation of a new root itself.

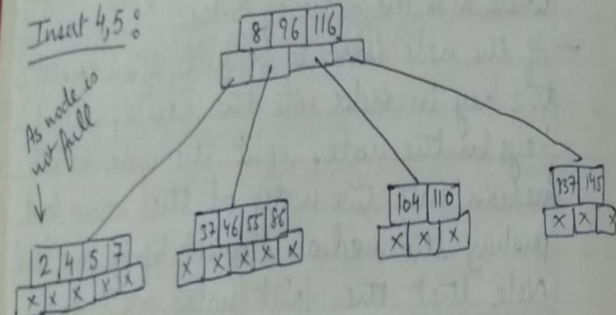
NOTE: B-Trees grows upwards.

Example: Consider the B-tree of order 5 given. Insert 4, 5, 58 in the order given.



Insert 4,5 :

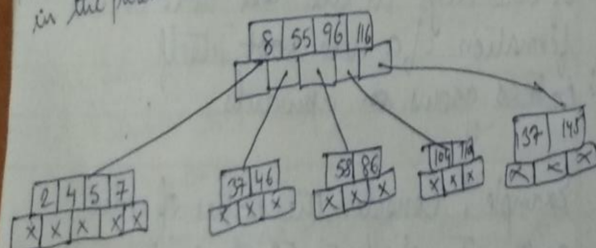
As node is not full



Insert 58 :

The second child node of the root is full

Now we insert the key into the list of keys existing in the node in the sorted order & split the node into two nodes at the same level at its median 55. Push the key 55 up to accomodate it in the parent node which is not full. Insert 58



~~[illegible]~~

cse.ohio-state.edu/~gurari/course/cis680/cis680Ch14-

Date
Page

Deletion in a B-Tree

While deleting a key it is desirable that the keys in the leaf node are removed.

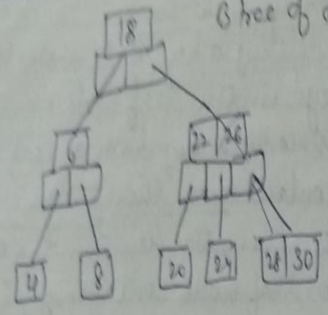
* While removing a key from a leaf node, if the node contains more than the minimum no. of elements, then the key can be easily removed.
→ If the leaf node contains just the minimum no. of elements, then element from ~~the~~ either the left sibling node or right sibling node to fill the vacancy.
* If the left sibling node has more than the minimum no. of keys, pull the largest key up into the parent node and move down the intervening entry from the parent node to the leaf node
* Otherwise, pull the smallest key of the right sibling node to the parent node and move down the intervening parent element to the leaf node.

→ If both the sibling nodes have only min. no. of entries, then create a new leaf node out of two leaf nodes & the intervening

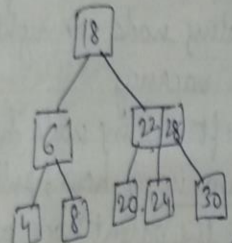
→ Key i.e to be removed from a node with non-empty subtree is being replaced with the largest key of the left subtree or the smallest key in the right subtree

B tree of order 3

1



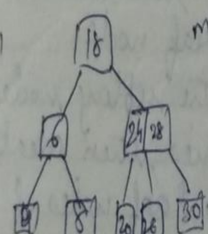
Remove 24



→ If a node becomes under staffed, it looks for a sibling with an extra key. If such a sibling exist, the node takes a key from the parent and the parents gets the extra key from the sibling

Remove 22 from Fig 1

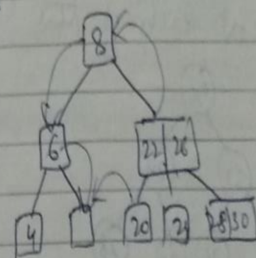
m/2 nodes

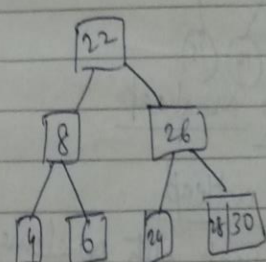


Date ....
Page ....

→ If a node becomes under staffed & it can't receive a key from the sibling, the node is merged with a sibling & a key from the parent is moved down to node

Remove 18





HEAP

A binary heap is a simple data structure that efficiently supports the priority queue operations.

Suppose H is a complete binary tree with n elements. Then H is called a heap or a maxheap if each node N of H has the following property
→ The value at N is greater than or equal to the value at each of the children of N

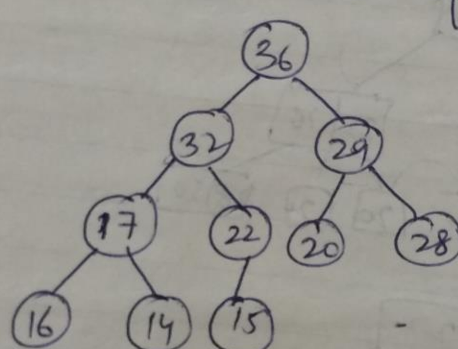


Fig: Maxheap

Inserting into a Heap

Suppose H is a heap with N elements & suppose an ITEM of info. is given. We insert ITEM in 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 heap.

2) Then let ITEM rise to its appropriate place in H so that H is finally a heap

2018/10/30 10:43

Inserting into Heap

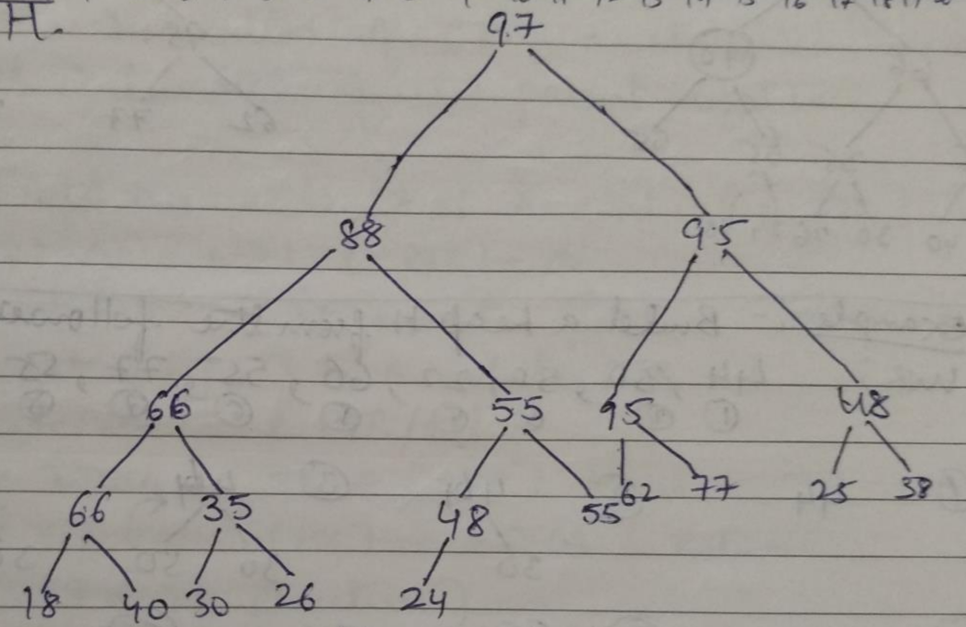
~~Efficiency of B-trees~~

Example

TREE

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
97	88	95	66	55	95	48	66	35	48	55	62	77	25	38	18	40	30	26	24

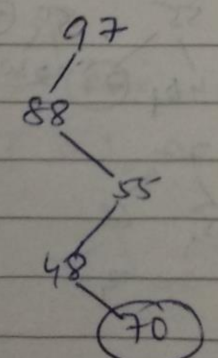
Given H.



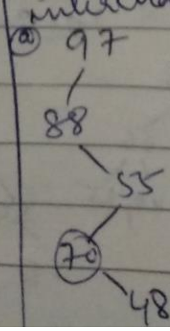
Heap

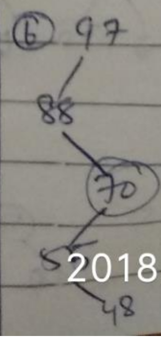
Add 70 to H

Adjoin 70 as the next element in the complete tree i.e TREE[21] = 70 i.e. right child of TREE[10] = 48



(a) Compare 70 with its parent 48. Since 70 > 48 interchange





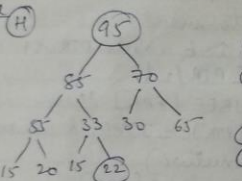
2018/10/30 10:43

Deleting the Root of a Heap

Suppose H is a heap with N elements & suppose we want to delete the root R of H. This 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.

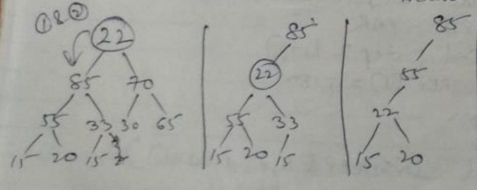
Example
Given H



~~Return~~ Delete 95

1. Delete 95
2. Replace 95 by last node i.e 22.
3. Make a heap
4. Compare 22 with 85 & 70
22 < 85 (larger child)
Interchange

1 & 2

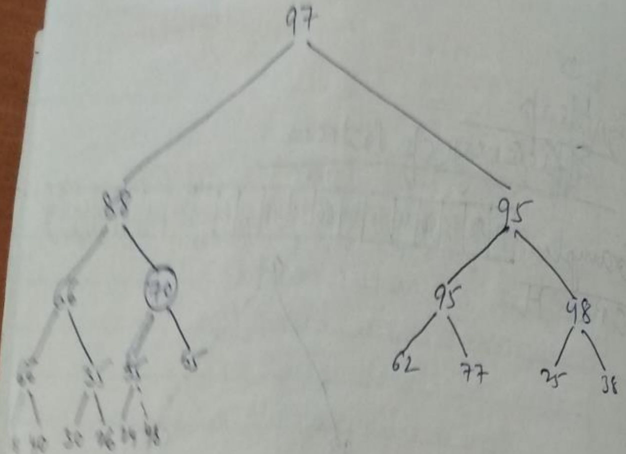


8. 22 < 55
6. 22 > 20, 15

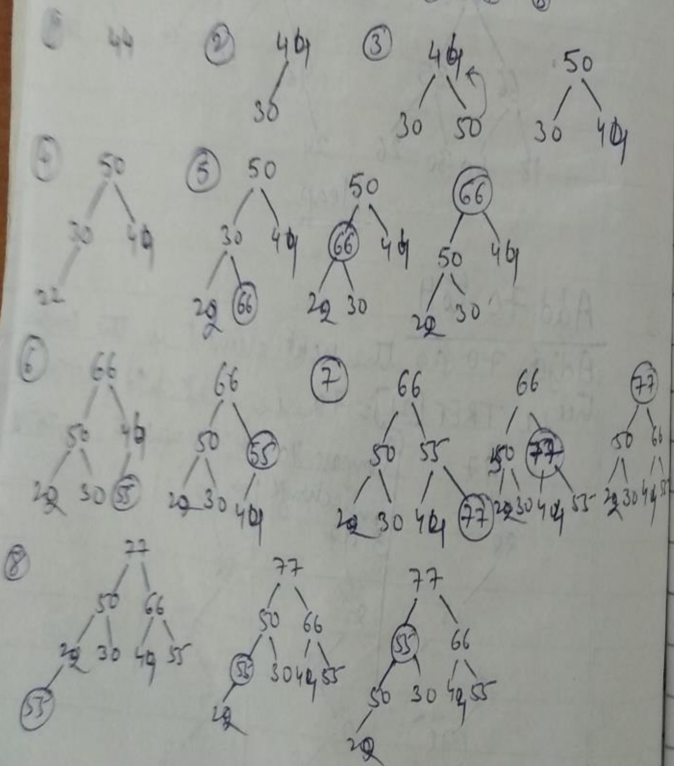
Procedure 2 : DELHEAP (TREE, N, ITEM)

TREE → array of size N
LAST → saves last node

1. Set ITEM := TREE[1]
2. Set LAST := TREE[N] & N := N-1
3. Set PTR := 1, LEFT := 2 & RIGHT := 3
4. Repeat Steps 5 to 7 while RIGHT ≤ N
5. If LAST ≥ TREE [LEFT] & LAST ≥ TREE [RIGHT] then
   Set TREE [PTR] := LAST & Return
   [End of If structure]
6. If TREE [RIGHT] ≤ TREE [LEFT] then:
   Set TREE [PTR] := TREE [LEFT] & PTR := LEFT
   Else
   Set TREE [PTR] := TREE [RIGHT] & PTR := RIGHT
   [End of If structure]
7. Set LEFT := 2 * PTR & RIGHT := LEFT + 1
   [End of Step 4 loop]
8. If LEFT = N & if LAST < TREE [LEFT] then
   Set PTR := LEFT
9. Set TREE [PTR] := LAST
10. Return



Example :- Build a heap H from the following list of numbers 44, 30, 50, 22, 66, 55, 77, 55



Date ....
Page ....

Procedure1: INSHEAP (TREE, N, ITEM)

N → no. of elements in array.
TREE → array
PTR → location of ITEM as it rises in tree.
PAR → location of the parent of ITEM

1 [Add new node to H & initialize PTR]
Set N := N+1 & PTR := N
2 [Find location to insert
Repeat steps 3 to 6 while PTR < 1
3. Set PAR := $\lfloor PTR/2 \rfloor$
4. If ITEM ≤ TREE [PAR] then
Set TREE [PTR] := ITEM & Return
[End of if structure]
5. Set TREE [PTR] := TREE [PAR]
6. Set PTR := PAR
[End of step 2 loop]
7. Set TREE [I] := ITEM
8 Return

Call INSHEAP [A, J, A[J+1])
for J = 1, 2, --, N-1

Heap Sort

Phase A : Build a heap H out of the elements of A

Phase B : Repeatedly delete the root element of H.

Since the root H always contains the largest node H, Phase B deletes the elements of A in decreasing order

Algo HEAPSORT (A, N)

1. [Build a heap H]
   Repeat for J = 1 to N-1
   Call INSHEAP(A, J, A[J+1])
   [End of loop]

2. Repeat While N > 1
   (a) Call DELHEAP (A, N, ITEM)
   (b) Set A[N+1] := ITEM
   [End of loop]

3. Exit