Searching

→ refers to the operation of finding the location of a given item in a collection of items.

→ Some useful terms, a table or file is a group of elements each of which is called a record. Associated with each record is called a Key; which is used to differentiate among different records.

→ for every file, there is at least one set of keys that is unique, is called a primary key

Eg if the file is stored as an array, the index within the array of an element is a unique external key for that element

Sequential Searching

→ this search is applicable to a table organized either as an array or as a linked list.

Algo: LINEAR (DATA, N, ITEM, LOC)

1. Set DATA [N+1] = ITEM
2. Set LOC = 1
3. Repeat while DATA [LOC] ≠ ITEM
   Set. LOC = LOC + 1
4. if LOC = N+1, then set LOC = 0
5. EXIT

Complexity:

→ Worst case: $f(n) = n+1$
i.e. running time is proportional to n.

→ Average case:
n:

Binary Search:

→ most efficient method of searching a sequential table without the use of auxiliary indices or tables

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

1. Set Beg = LB, End = UB and Mid = int((Beg + End)/2)
2. Repeat steps 3 and 4 while Beg ≤ End and Data[mid] ≠ Item
3. if ITEM < Data[mid], then
   Set End = Mid - 1
   else
   set Beg = Mid + 1
4. set Mid = int((Beg + End)/2)
5. if Data[mid] = Item then
   set LOC = Mid
   else
   set Loc = Null
6. Exit

Complexity:

→ measured by number of comparisons
→ each comparison reduces the sample size in half.

$$2^{f(n)} > n \quad \text{or} \quad f(n) = \lfloor \log_2 n \rfloor + 1$$

Worst case: $\log_2 n$

Average Case: $\log_2 n$

Comparison: $n/2 + n/4 + n/8 + \cdots + n/2^i$

Lecture-28

Sorting

→ it refers to the operation of arranging data in some given order, such as increasing or decreasing with numerical data or alphabetically with character data

→ There are two broad categories of sorting methods

Internal sorting takes place in the main memory, where we can take advantage of the random access nature of main memory

External sorting is necessary when the number and size of objects are prohibitive to be accommodated in the main memory.

Insertion sort:

→ Suppose an Array A with n elements A[1], A[2], ... , A[N] is in memory.

→ This algorithm scans A from A[1] to A[N], inserting each element A[K] into its proper position in the previously sorted subarray A[1], A[2] - —— , A[K-1], that is

Pass 1: A[1] by itself is trivially sorted

Pass 2. A[2] is inserted either before or after A[1], so that: A[1], A[2] is sorted

Pass 3. A[3] is inserted into its proper place in A[1], A[2], so that A[1], A[2], A[3] is sorted.

Pass N. A[N] is inserted into its proper place in A[1], A[2], ... A[N-1], so that A[1], A[2], A[3], ... A[N] is sorted.

Eg:

77, 33, 44, 11, 88, 22, 66, 55

Pass	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
K=1	$-\infty$	(77)	33	44	11	88	22	66	55
2	$-\infty$	77	(33)	44	11	88	22	66	55
3	$-\infty$	33	77	(44)	11	88	22	66	55
4	$-\infty$	33	44	77	(11)	88	22	66	55
5	$-\infty$	11	33	44	77	(88)	22	66	55
6	$-\infty$	11	33	44	77	88	(22)	66	55
7	$-\infty$	11	22	33	44	77	88	(66)	55
8	$-\infty$	11	22	33	44	66	77	88	(55)
Sorted	$-\infty$	11	22	33	44	55	66	77	88

Algorithm: INSERTION (A, N)

1. Set A[0] := $-\infty$ [initializes sentinel element]
2. Repeat Steps 3 to 5 for K = 2, 3, ---, N
3. Set Temp = A[K] and ptr = K-1
4. Repeat while Temp < A[ptr]:
   (a) set A[ptr+1] = A[ptr]
   (b) set ptr = ptr-1
5. Set A[ptr+1] = Temp
6. Return

Complexity

→ Worst case: array is in reverse order, so K-1 comparisons

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

→ Average case: approximately (K-1)/2 comparisons. so, $f(n) = n(n-1)/4 = O(n^2)$

Lecture-29

Selection Sort:

→ Suppose an array A with n elements A[1], A[2], ... , A[N] is in memory.

→ First find the smallest element in the list and put it in the first position. Then find the second smallest element in the list and put it in the second position & so on.

E.g

Pass	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
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
Sorted:	11	22	33	44	55	66	77	88

Algo: SELECTION (A, N)

1. Repeat Steps 2 and 3 for K=1, 2, ... N-1
2. Call MIN (A, K, N, LOC)
3. Set, Temp = A[K], A[K] = A[LOC] and A[LOC] = Temp
4. Exit

MIN (A, K, N, LOC)

1. Set, Min = A[K] and LOC = K
2. Repeat for J = K+1, K+2, ... N
3. if Min > A[J], then, Set Min = A[J] and LOC = A[J] and LOC = J
3. Return

Complexity

→ no. of comparisons $f(n)$ is independent of original order of elements.

→ Min (A, K, N, LOC) requires $n-K$ comparisons.

i.e. $(n-1), (n-2), \cdots (n-2)$ and so, on no. of comparisons

So, $f(n) = (n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n-1)}{2} = O(n^2)$

→ Worst case : $O(n^2)$

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

Lecture-30

Bubble Sort:

→ First compare A[1] and A[2] & arrange them in desired order, so that A[1] < A[2], then compare A[2] and A[3] & arrange them so that A[2] < A[3]. Continue until we compare A[N-1] with A[N] and arrange them so that A[N-1] < A[N]

it involves n-1 comparisons.

→ Repeat above step with one less comparisons i.e. Now we stop after we compare and possibly rearrange A[N-2] and A[N-1], it involves n-2 comparisons.

→ Repeat above step with two fewer comparisons i.e. stop after we compare and possibly rearrange A[N-3] and A[N-2]

⋮

→ After n-1 steps, the list will be sorted in increasing order

E.g (32), (51), 27, 85, 66, 23, 13, 57

Pass 1

32, (51), (27), 85, 66, 23, 13, 57

32, 27, (51), (85), 66, 23, 13, 57

32, 27, 51, (85), (66), 23, 13, 57

32, 27, 51, 66, (85), (23), 13, 57

32, 27, 51, 66, 23, (85), (13), 57

32, 27, 51, 66, 23, 13, (85), (57)

85, largest no. has moved to the last position

Pass 2

(32), (27), 51, 66, 23, 13, 57, 85

27, (32), (51), (66), (23), 13, 57, 85

27, 32, 51, 23, (66), (13), 57, 85

27, 32, 51, 23, 13, (66), (57), 85

27, 32, 51, 23, 13, 57, (66), (85)

27, 32, 51, 23, 13, 57, 66, 85

66, second largest no. has moved to its way down to the next-to-last position.

Pass 3: 27, 32, (23) (51), 13, 57, 66, 85
27, 32, 23, (13) (51), 57, 66, 85

Pass 4: 27 (23) (32), 13, 51, 57, 66, 85
27, 23, (13) (32), 51, 57, 66, 85

Pass 5: (23) (27), 13, 32, 51, 57, 66, 85
23, (13) (27), 32, 51, 57, 66, 85

Pass 6: (13), (23), 27, 33, 51, 57, 66, 85 . A1 & A2 | A2 & A3

Pass 7: 13, 23, 27, 33, 51, 57, 66, 85 . A1 & A2.

Algorithm: BUBBLE (DATA, N)

1. Repeat steps 2 and 3 for K=1 to N-1
2. Set Ptr = 1
3. Repeat while Ptr ≤ N-K
   (a) if Data [ptr] > Data [ptr+1], then
   Interchange Data [ptr] and Data [ptr+1]
   (b) Set ptr = ptr+1
4. Exit

Complexity: no. of comparisons, f(n)

$f(n) = (n-1) + (n-2) + \cdots + 2 + 1 = n(n-1)/2$

$= n^2/2 + O(n) = O(n^2)$

Lecture - 3

Quick Sort

→ It uses divide and conquer technique, i.e. the problem of sorting a set is reduced to the problem of sorting two smaller sets.

Eg 54, 26, 93, 17, 77, 31, 44, 55, 20

54, will serve as our first pivot value.

→ first select a value, which is called the pivot value. (many different ways to choose the pivot value)

→ the actual position where the pivot value belongs in the final sorted list, commonly called split point, will be used to divide the list for subsequent calls to quick sort.

→ Partitioning begins by locating two position markers - leftmark and rightmark, at the beginning and end of the remaining items in the list.

Greater

54, 26, 93, 17, 77, 31, 44, 55, 20 — less

↑ leftmark ↑ rightmark

54, 26, 93, 17, 77, 31, 44, 55, 20 — 26 < 54, 93 ≮ 54, stop move to right

↑ leftmark ↑ rightmark — 20 < 54, stop

Now, continue moving left & right mark — exchange 20 & 93

54, 26, 20, 17, 77, 31, 44, 55, 93 — 77 > 54

↑ left ↑ right — 44 < 54 stop

exchange 44 & 77

54, 26, 20, 17, 44, 31, 77, 55, 93 — 77 > 54

↑ left ↑ right — 31 < 54 stop

split point found as right < left

exchange 54 and 31

31, 26, 20, 17, 44, 54, 77, 55, 93

Quicksort left half — right half

Algorithm:

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

```
quicksort (left, right)
    if (left < right):
        splitpoint = partition (array, left, right)
        quicksort (left, splitpoint - 1)
        quicksort (splitpoint + 1, right)

partition (array, left, right)
    pivot = array [left]
    leftmark = left + 1
    rightmark = right
    done = false
    while not done:
        while leftmark <= rightmark and array [leftmark] <= pivot
            leftmark = leftmark + 1
        while array [rightmark] >= pivot and rightmark >= leftmark
            rightmark = rightmark - 1
        if rightmark < leftmark
            done = True
        else
            temp = array [leftmark]
            array [leftmark] = array [rightmark]
            array [rightmark] = temp
```

temp = array [left]

array [left] = array [rightmark]

array [rightmark] = temp

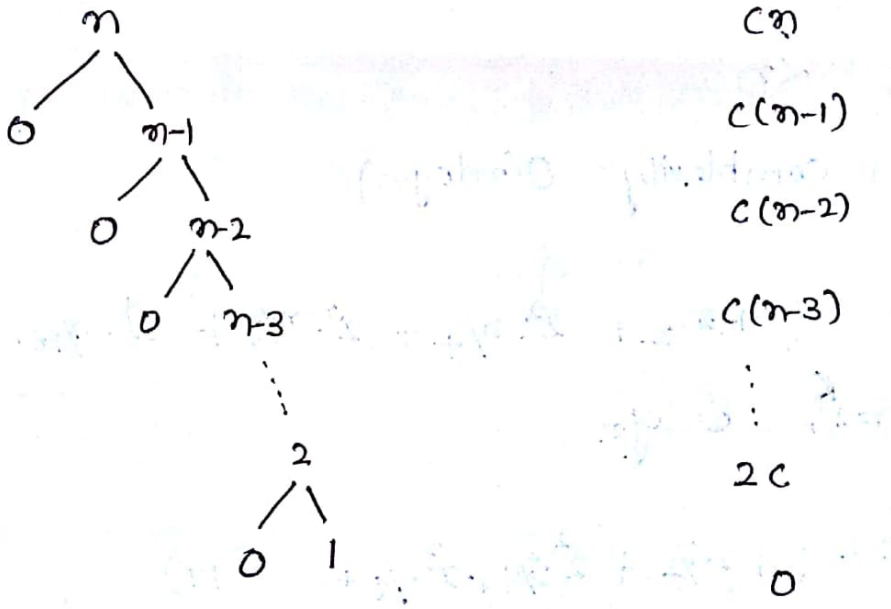
return rightmark

Complexity

Worst Case:

Subproblem size

Total partitioning time for all subproblems of this size



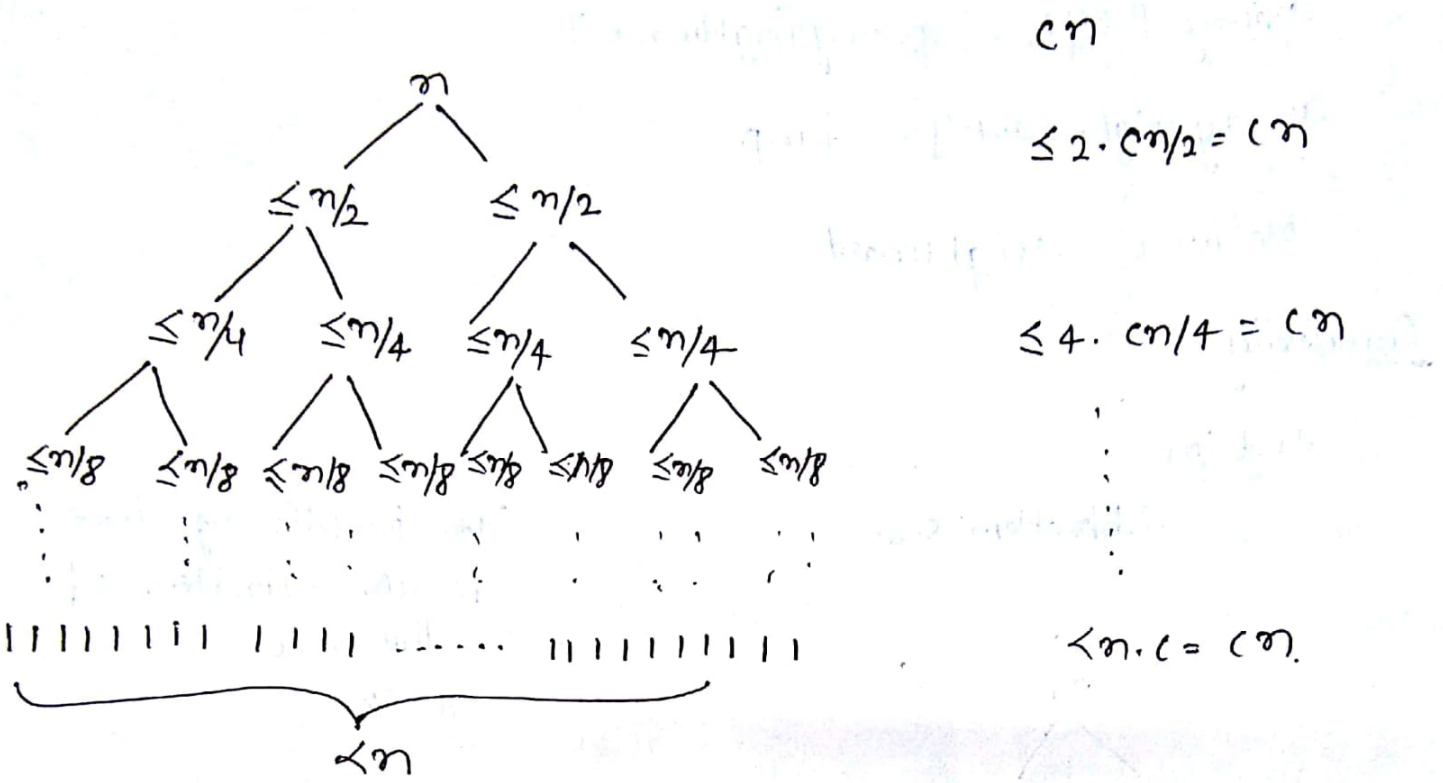
$cn + c(n-1) + c(n-2) + \cdots + 2c = c(n + (n-1) + (n-2) + \cdots + 2)$

$= c((n+1)(n/2) - 1)$

i.e.

worst case running time $= \Theta(n^2)$

Best Case & Average Case



So, time complexity = $O(n \log n)$

$c(n + 2n/2 + 2^2 \cdot n/4 + 2^3 \cdot n/8 + 2^4 \cdot n/16 + \cdots + 2^K \cdot n/2^K)$

$n = 2^K, \quad K = \log n$

$n \cdot c\,(1 + 2 \cdot 1/2 + 2^2 \cdot 1/4 + 2^3 \cdot 1/8 + \cdots + 1)$

$n \cdot c \cdot 1 = O(n \cdot \log n)$

Lecture - 32

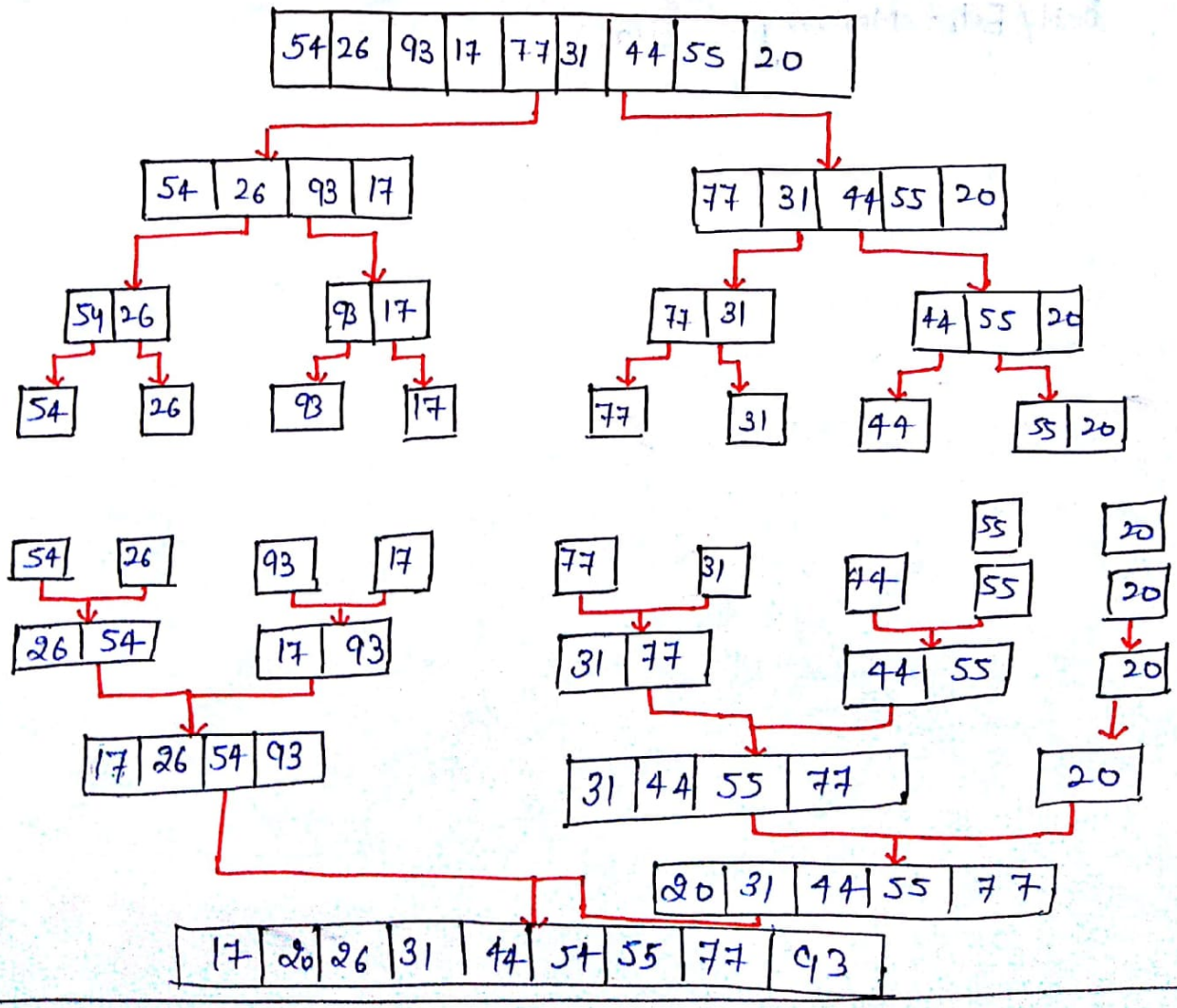
Two-way Merge Sort

→ Also, based on divide and conquer strategy.

→ it is a recursive algorithm that continually splits a list in half. If the list is empty or has one item, it is sorted by definition and If the list has more than one item, we split the list and recursively invoke a merge sort on both halves.

→ A K-way merge sort that sorts that sorts a data stream using repeated merges. It distributes the input into two streams by repeatedly reading a block of input that fits in memory, then writing it to the next stream.

→ repeatedly distributes the runs to two streams & merges them until there is a single output.



Algorithm:

Mergesort (A, N)

1. Set L=1 (Initialize no. of elements in subarray)
2. Repeat steps 3 to 5 while L < N
3. Call Merge pass (A, N, L, B) B → auxiliary array
4. Call Merge Pass (B, N, 2*L, A)
5. Set L= 4*L
6. Exit.

Complexity

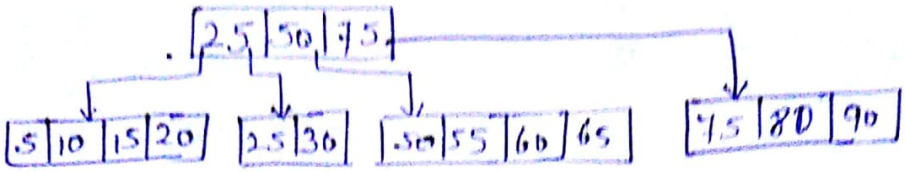
$f(n) \leq n \log n$.

Worst/Average case : $O(n \log n)$

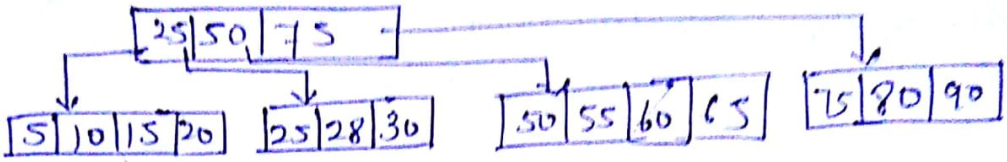
Best/Extra Memory : $O(n)$

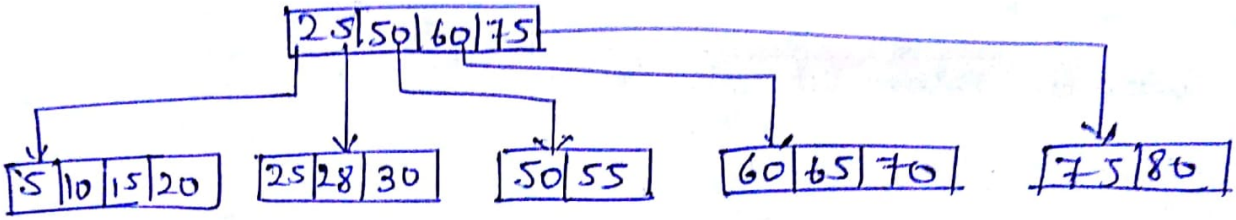
B+ Tree

4- nodes.



Insert 28, 70





35, 45, 25, 11, 6, 85, 17, 35 sort using heap sort.

Heap Sort:

→ it is a comparison based sorting technique based on Binary heap data structure

→ Suppose, H is a complete binary tree with n elements then H is called a heap or 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

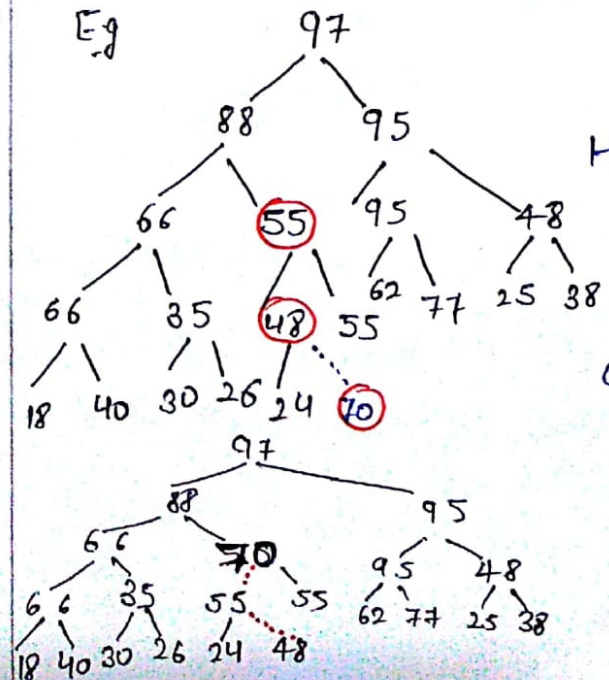
→ A minheap is defined an analogously: the value at N is less than or equal to the value at any of the children of N

Inserting into a Heap

Suppose H is a heap with N elements.

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.

Eg



Add, 70.

H

97	88	95	66	55	95	48	66	35	48	55	62	77	25	38	18	40	30	26	24

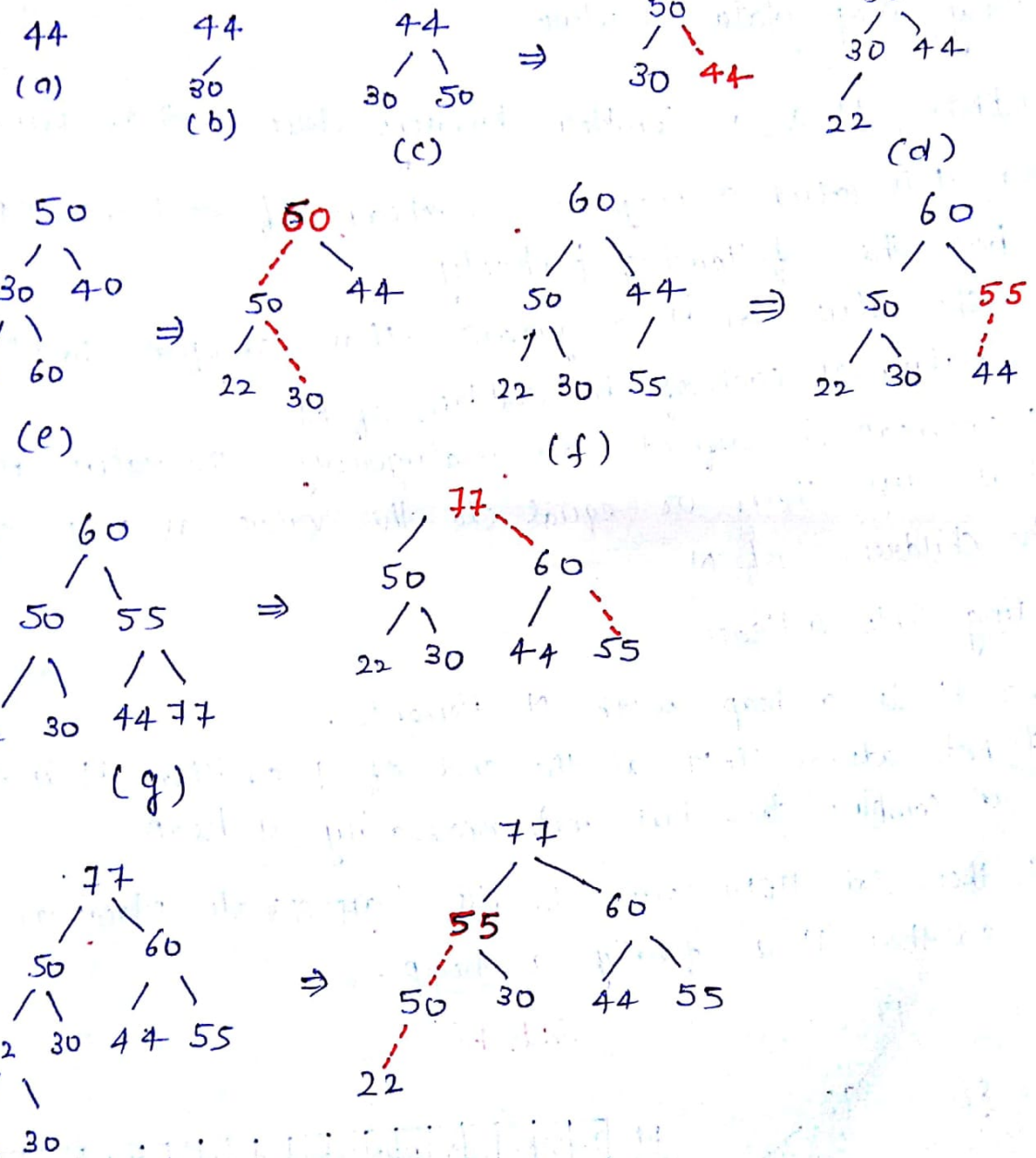
Sequential representation

Set H[21] = 70, then 70 is the right child of H[10] = 48.

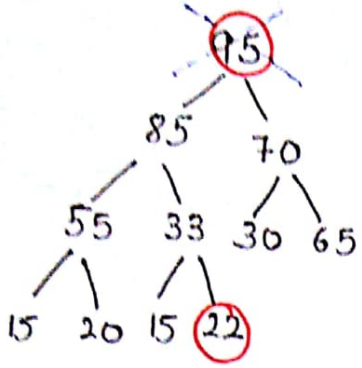
Now, find the appropriate place of 70.

Build a heap H

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

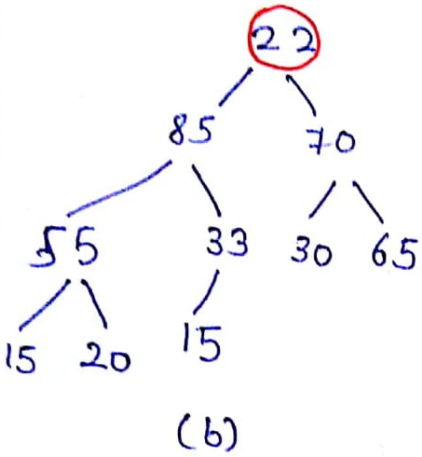


Deletion in Heapsort



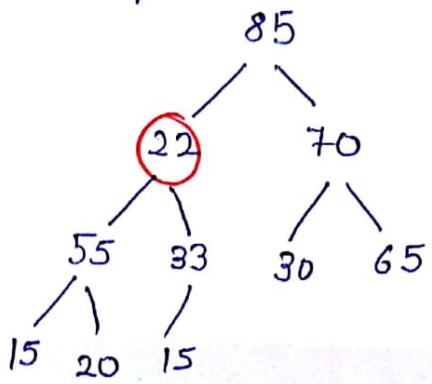
Suppose, H is heap & we want to delete the root R of H.

1. Assign the root R to 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.

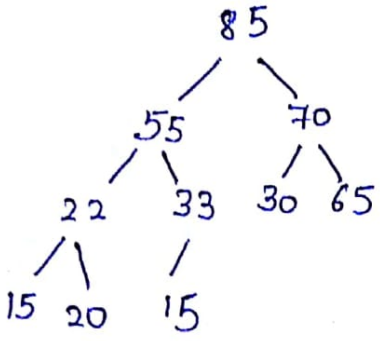


(b)

⇒



(c)



(d)

Complexity:

a) Insert a single node in an empty heap $O(1)$

b) Insert a single node in an n node heap $O(\log n)$

c) Insert n element in a n node heap $O(n \log n)$

d) To create a heap, time complexity $(n \log n)$

Radix Sort:

→ is a clever and intuitive little sorting algo.

→ It is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value.

→ Two classifications of radix sort
  (i) Least significant digit (LSD)
  (ii) Most significant digit (MSD)

E.g 493, 812, 715, 710, 195, 437, 582, 340, 385

We should start by comparing & ordering the one's digit

digit	Sublist
0	340, 710
1	
2	812, 582
3	493
4	
5	715, 195, 385
6	
7	437
8	
9	

340, 710, 812, 582, 493, 715, 195, 385, 437

Now, based on ten's digit

710, 812, 715, 437, 340, 582, 385, 493, 195

Finally, based on hundred's digit

195, 340, 385, 437, 493, 582, 710, 715, 812

Complexity: $O(n \log n)$

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 following property:

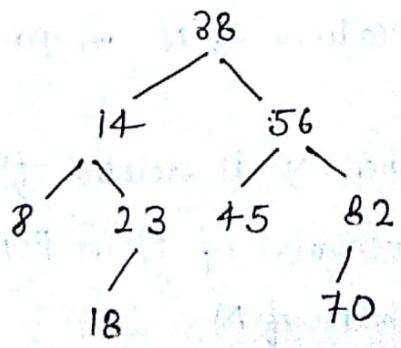
⇒ The value at N is greater than every value in the left subtree of N and is less than or equal to every value in the right subtree of N. (i.e. inorder traversal of T will yield a sorted listing of the elements of T)

→ This structure contrasts with the following structures:

(a) Sorted linear array: one can search for and find an element with a running time $f(n) = O(\log_2 n)$, but it is expensive to insert and delete elements.

(b) Linked list: one can easily insert and delete elements, but it is expensive to search for and find an element, one must use a linear search with running time $f(n) = O(n)$.

Searching in BST



Search, 20

20 < 38 → proceed to left child

20 > 14 → proceed to right of 14

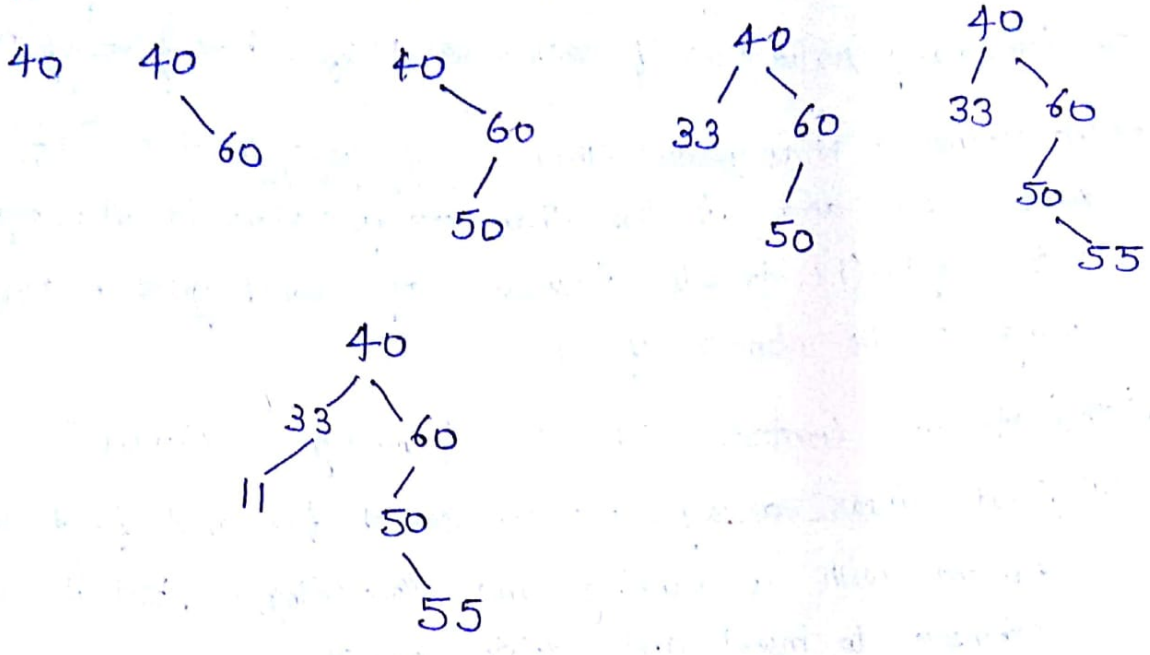
20 < 23 → proceed to left of 23

20 > 18 and 18 does not have right child

So, 20 is not in tree.

Inserting into BST

40, 60, 50, 33, 55, 11



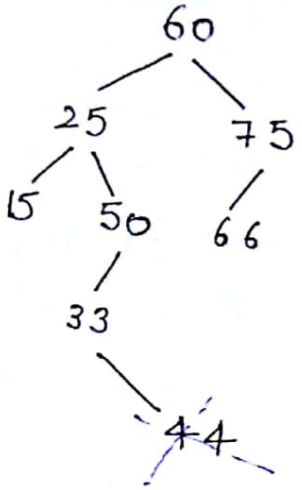
Deleting in BST

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

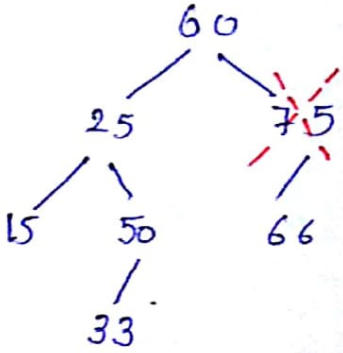
Case 1: N has no children, then N is deleted from T by simply replacing the location of N in parent node by the null pointer.

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 only child of N

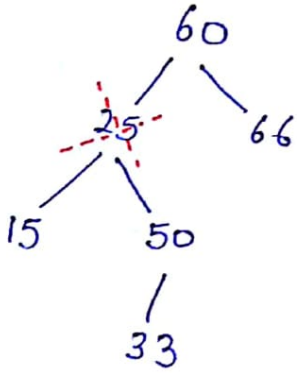
Case 3: N has two children. Let S(N) denote the inorder successor of N (S(N) does not have left child). then N is deleted from T by first deleting S(N) from T and then replacing node N in T by node S(N)



Delete 44

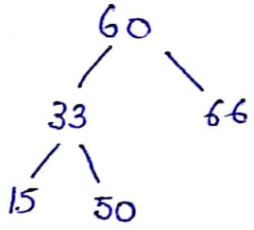


Delete 75



Delete 25

Inorder successor 33



Complexity :

For searching : $\log n$

FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)

1. if ROOT = NULL
2. then LOC = NULL and PAR = NULL and return
3. if ITEM = INFO[ROOT]
4. then LOC = ROOT and PAR = NULL and return
5. if ITEM < INFO[ROOT]
6. then PTR = LEFT[ROOT] and SAVE = ROOT
7. else PTR = RIGHT[ROOT] and SAVE = ROOT
8. Repeat step 9 and 11 while PTR ≠ NULL
9. if ITEM = INFO[PTR]
10. then LOC = PTR and PAR = SAVE and return
11. if ITEM < INFO[PTR]
12. then SAVE = PTR and PTR = LEFT[PTR]
13. else SAVE = PTR and PTR = RIGHT[PTR]
14. LOC = NULL and PAR = SAVE
15. Exit

Algorithm for Insertion in Binary Search Tree

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

1. Call FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC PAR)
2. if LOC ≠ NULL
3. then exit
4. if AVAIL = NULL
5. then write OVERFLOW and exit
6. NEW = AVALL ; AVAIL = LEFT [AVAIL], INFO[NEW] = ITEM
7. LOC = NEW, LEFT [NEW] = NULL and RIGHT [NEW] = NULL
8. if PAR = NULL
9. then ROOT = NEW
10. Else if ITEM < INFO[PAR]
11. then LEFT [PAR] = NEW
12. else RIGHT[PAR] = NEW
13. Exit.

Deletion From Binary Search Tree

```
DEL(INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM)
1. call FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)
2. if LOC = NULL
3.     then write "ITEM not in Tree" and exit.
4. if RIGHT[LOC] ≠ NULL and LEFT[LOC] ≠ NULL
5.     then call CASEB(INFO, LEFT, RIGHT, ROOT, LOC, PAR)
6.     else call CASEA(INFO, LEFT, RIGHT, ROOT, LOC, PAR)
7. LEFT[LOC] = AVAIL and AVAIL = LOC
8. Exit.

CASEA(INFO, LEFT, RIGHT, ROOT, LOC, PAR)
1. if LEFT[LOC] = NULL and RIGHT[LOC] = NULL
2.     then CHILD = NULL
3.     else if LEFT[LOC] ≠ NULL
4.             then CHILD = LEFT[LOC]
5.             else CHILD = RIGHT[LOC]
6. if PAR ≠ NULL
7.     then if LOC = LEFT[PAR]
8.             then LEFT[PAR] = CHILD
9.             else RIGHT[PAR] = CHILD
10.    else ROOT = CHILD
```

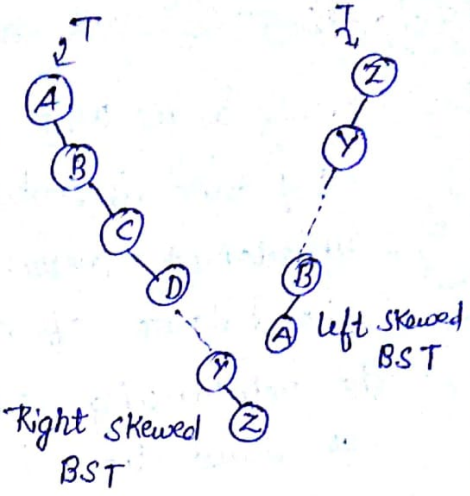
```
CASEB (INFO, LEFT, RIGHT, ROOT, LOC, PAR)

1.  PTR = RIGHT [LOC] and SAVE = LOC
2.  Repeat while LEFT[PTR] ≠ NULL
3.          SAVE = PTR and PTR = LEFT [PTR]
4.  SUC = PTR and PARSUC = SAVE
5.  Call CASEA (INFO, LEFT, RIGHT, ROOT, SUC, PARSUC)
6.  If PAR ≠ NULL.
7.      then if LOC = LEFT [PAR]
8.          then LEFT [PAR] = SUC
9.          else RIGHT [PAR] = SUC
10.     else ROOT = SUC
11. LEFT [SUC] = LEFT [LOC]
12.  RIGHT [SUC] = RIGHT [LOC]
13. Return.
```

Lecture -36

AVL search tree:

→ Insert A, B, C, D, ... Z into BST.

→ BST turns out to be right skewed and left skewed for Z, Y, ... B, A



→ The disadvantage of skewed BST is that the worst case time complexity of search is $O(n)$

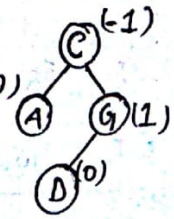
⇒ therefore, there arises the need to maintain BST to be of balanced height.

→ AVL, introduced in 1962 by Adelson-Velskii and Landis

→ An empty binary tree is an AVL tree.

→ A non empty binary tree T is an AVL tree iff given T^L and T^R to be the left & 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 balance factor (BF) & for AVL tree the balance factor of a node can be either 0, 1 or -1.



→ An AVL search tree is BST which is an AVL tree.

Insertion in AVL Search tree

→ same as in BST, if after insertion, the balance factor of any node is ~~unbalanced~~ affected so as to render the BST unbalanced, resort to techniques called **Rotations** to restore the balance of search tree.

→ the rebalancing rotations are classified as LL, LR, RR and RL as illustrated below:

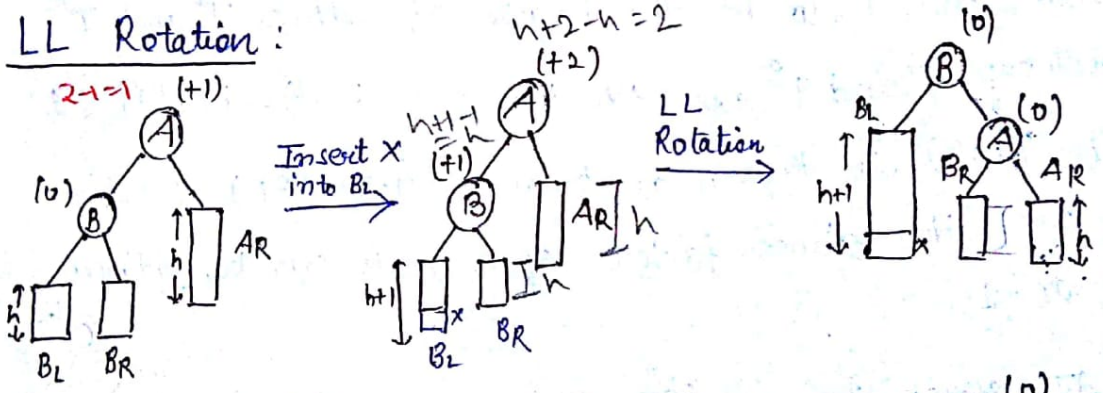
LL rotation: Inserted node is in the left subtree of left subtree of node A

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

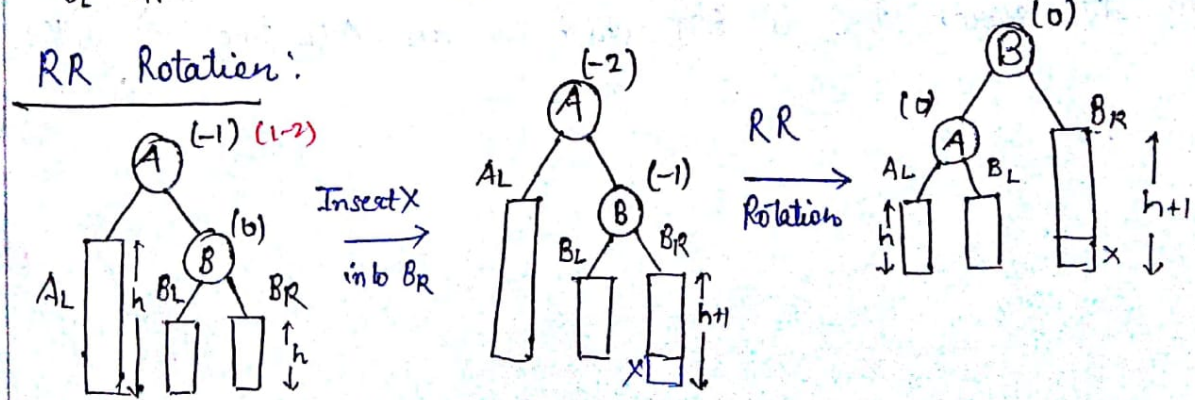
LR rotation: Inserted node is in the right subtree of left subtree of node A

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

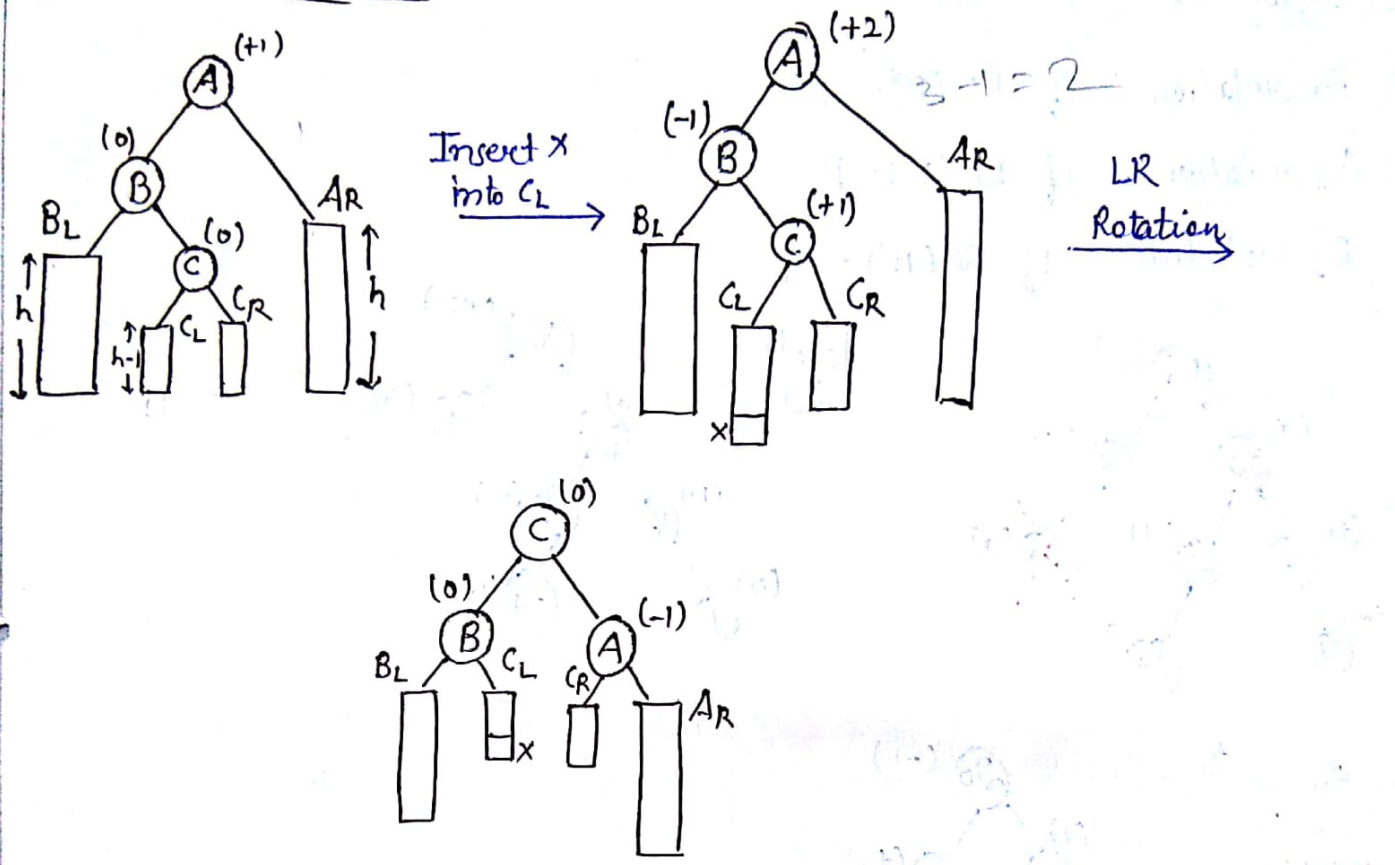
LL Rotation:



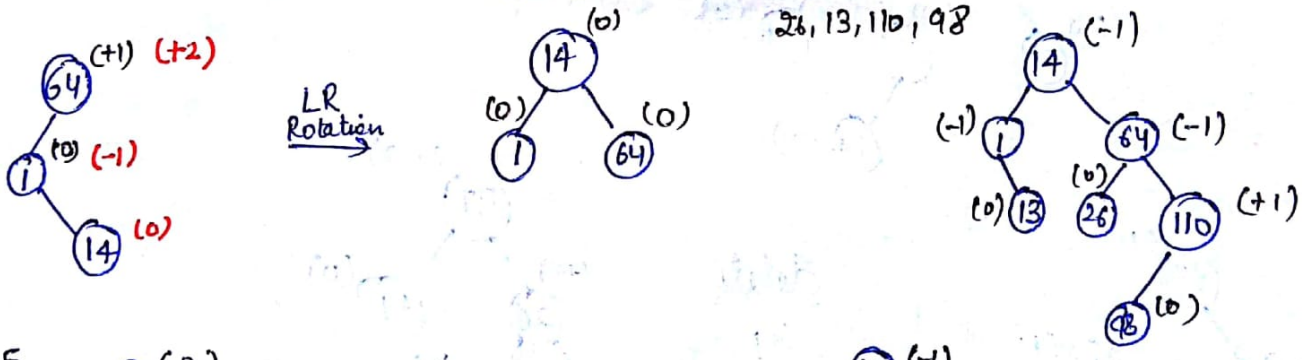
RR Rotation:



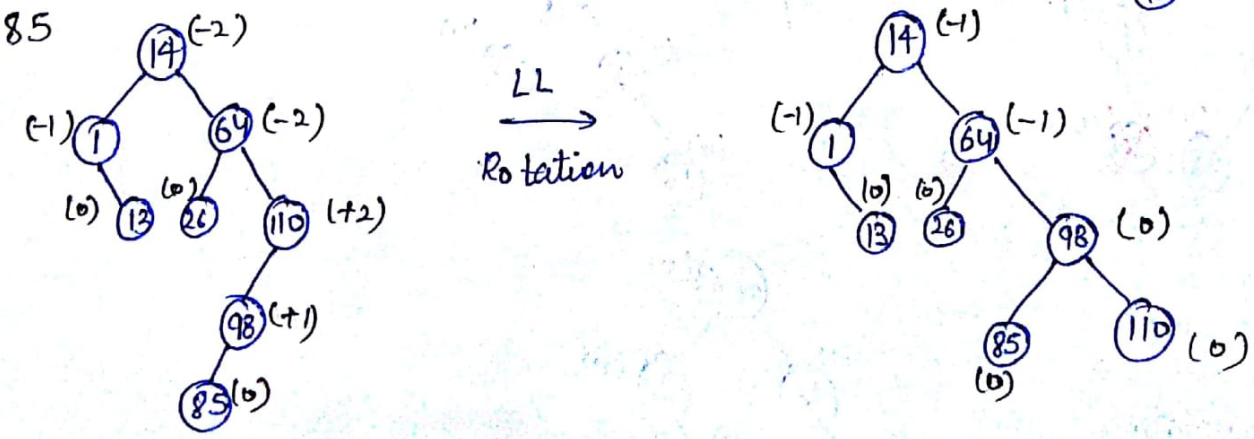
LR and RL rotation



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



85

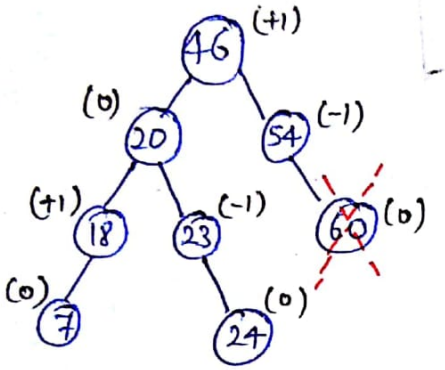


Deletion in AVL Search Tree

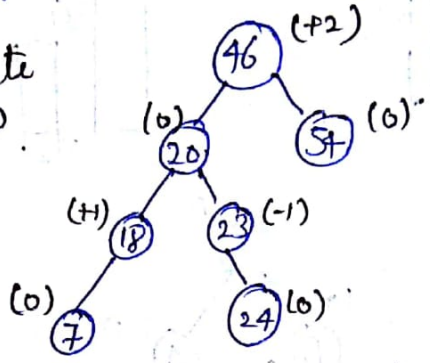
R_0 rotation : if $BF(B) = 0$,

R_1 rotation : if $BF(B) = 1$

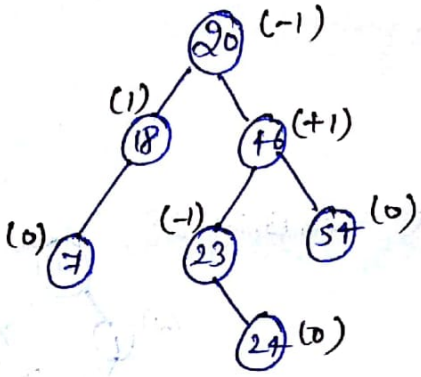
R_{-1} rotation : if $BF(B) = -1$

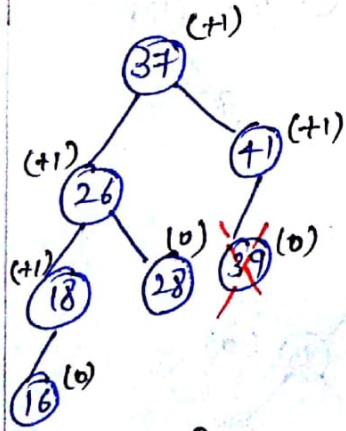


delete 60

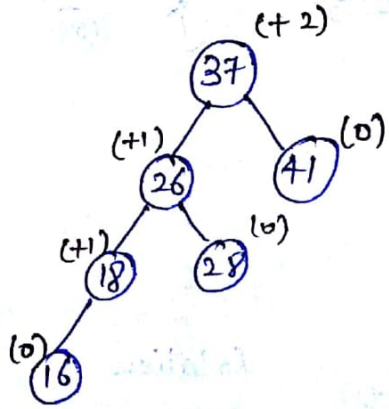


R_0 rotation

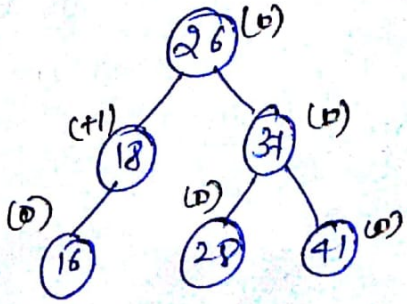




delete 39

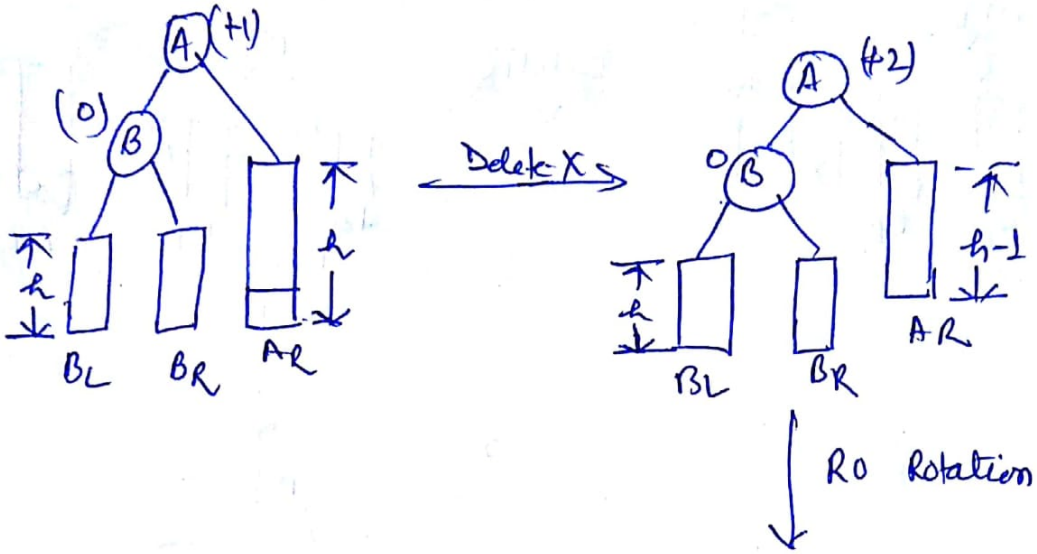


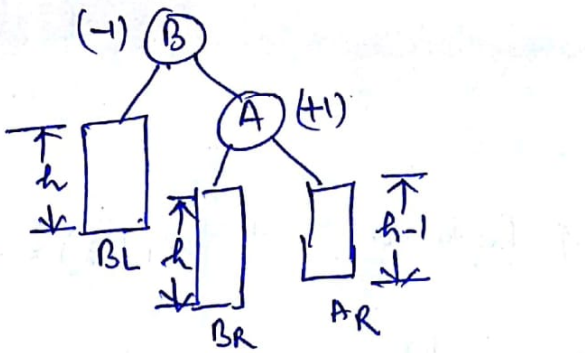
R_1 rotation



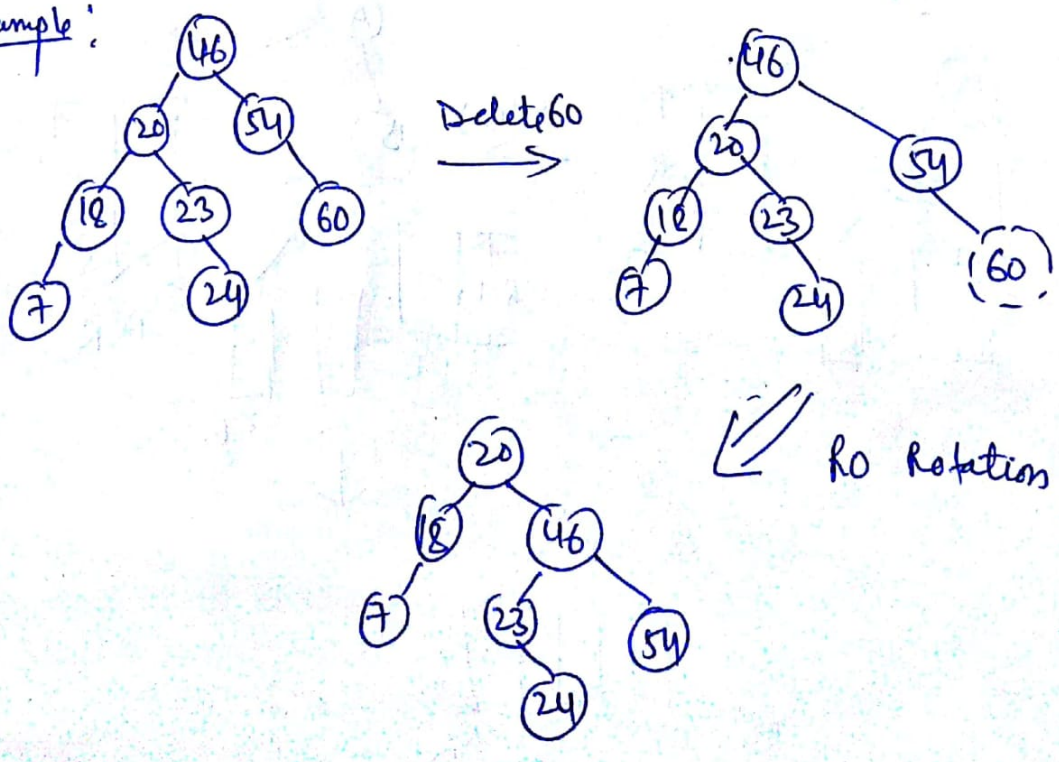
Deletion in an AVL Search Tree

RO Rotation – if BF(B) = 0 (LL anomaly)

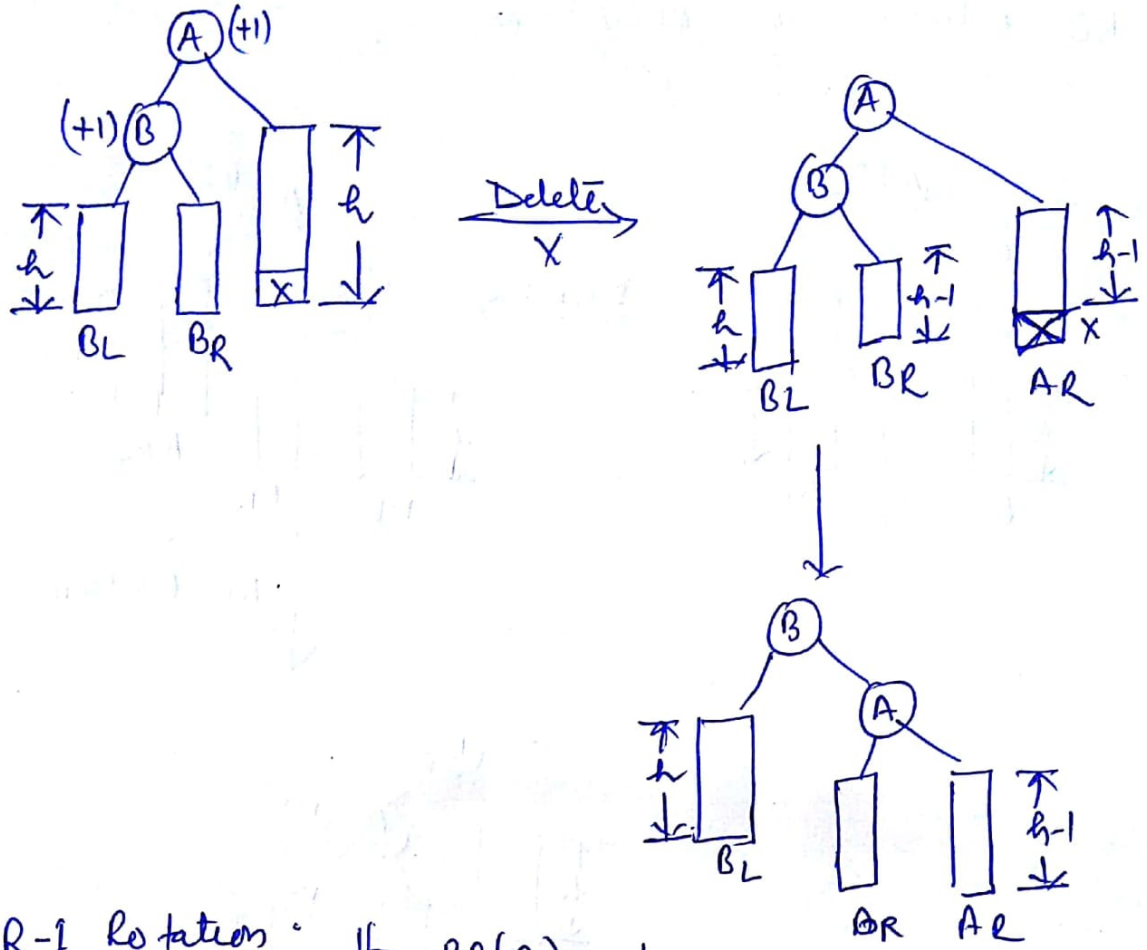




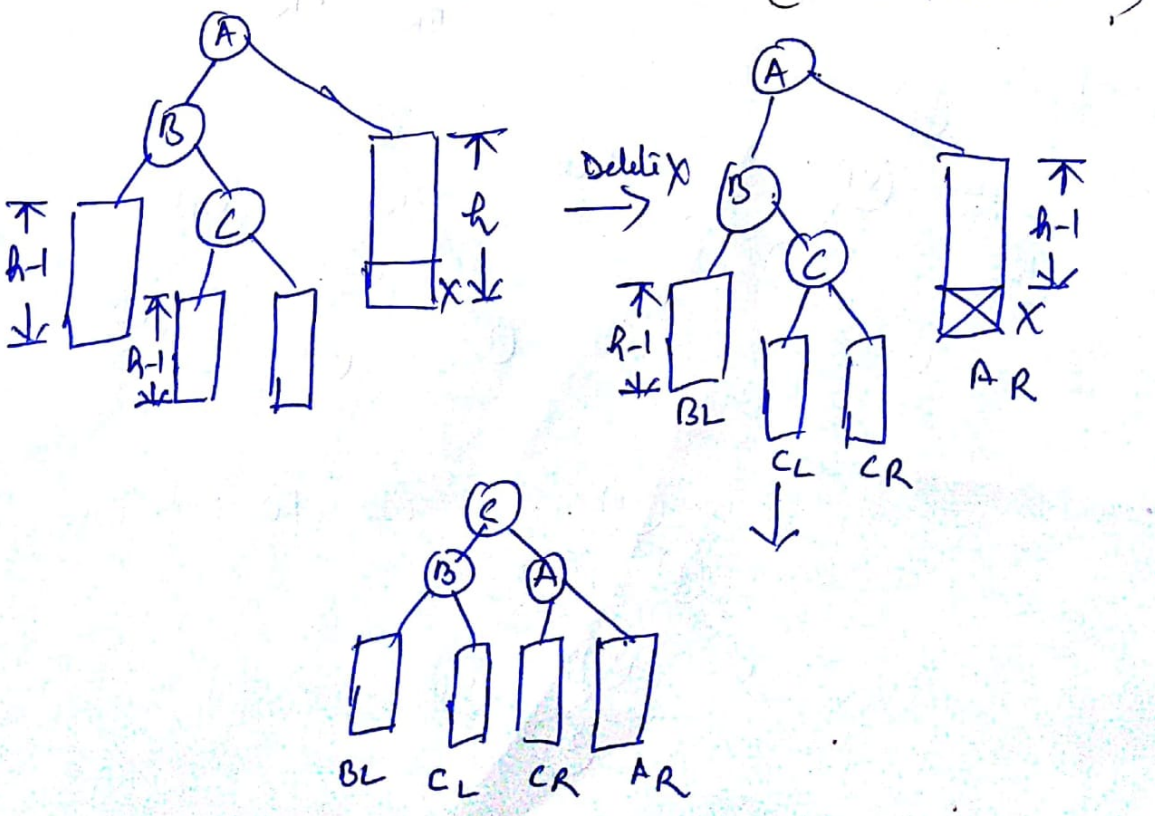
Example:

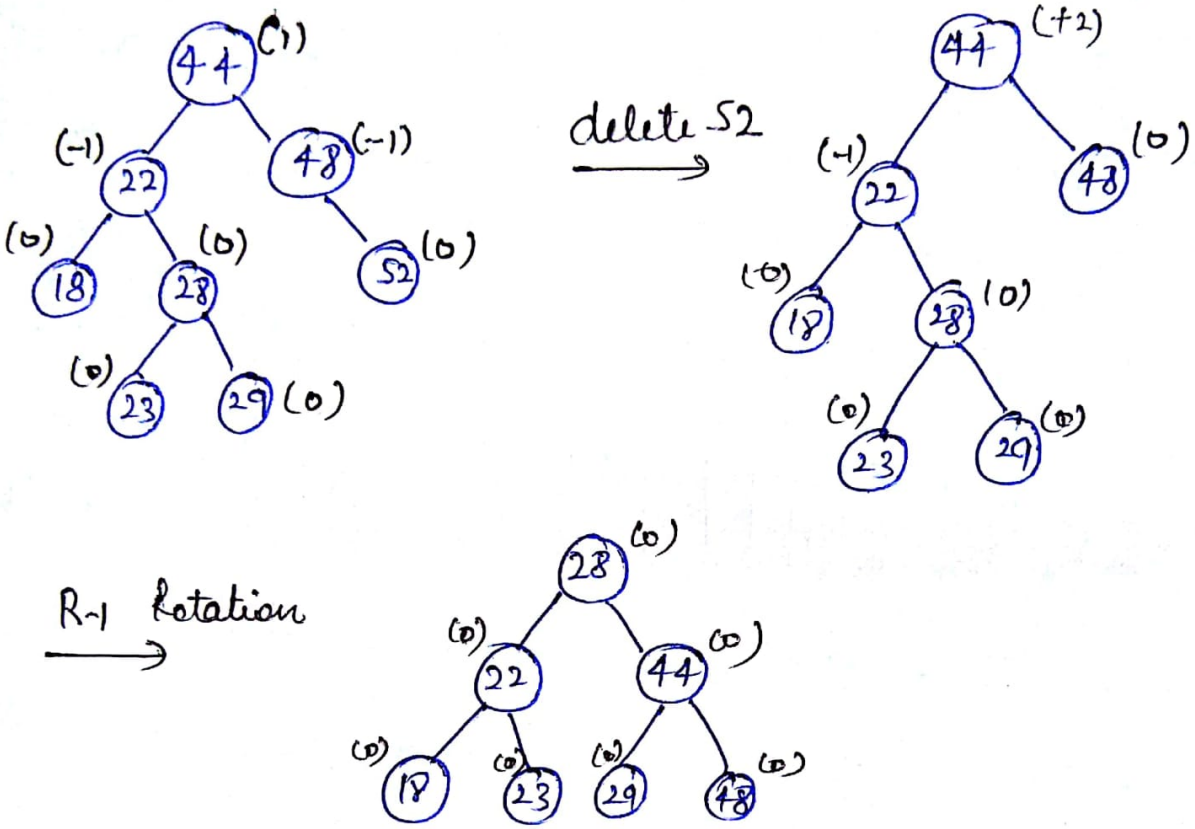


RL Rotation : If BF(B) = 1 (LL Rotation)



R-1 Rotation : If BF(B) = -1 (LR Rotation)





Lecture-38

B-Tree (Perfectly height balanced m-way search Tree)

→ m-way search trees have the advantage of minimizing file accesses due to their restricted height.

→ need to maintain balancing of m-way search tree.

→ A B-tree of order m, if non-empty, is an m-way ST in which

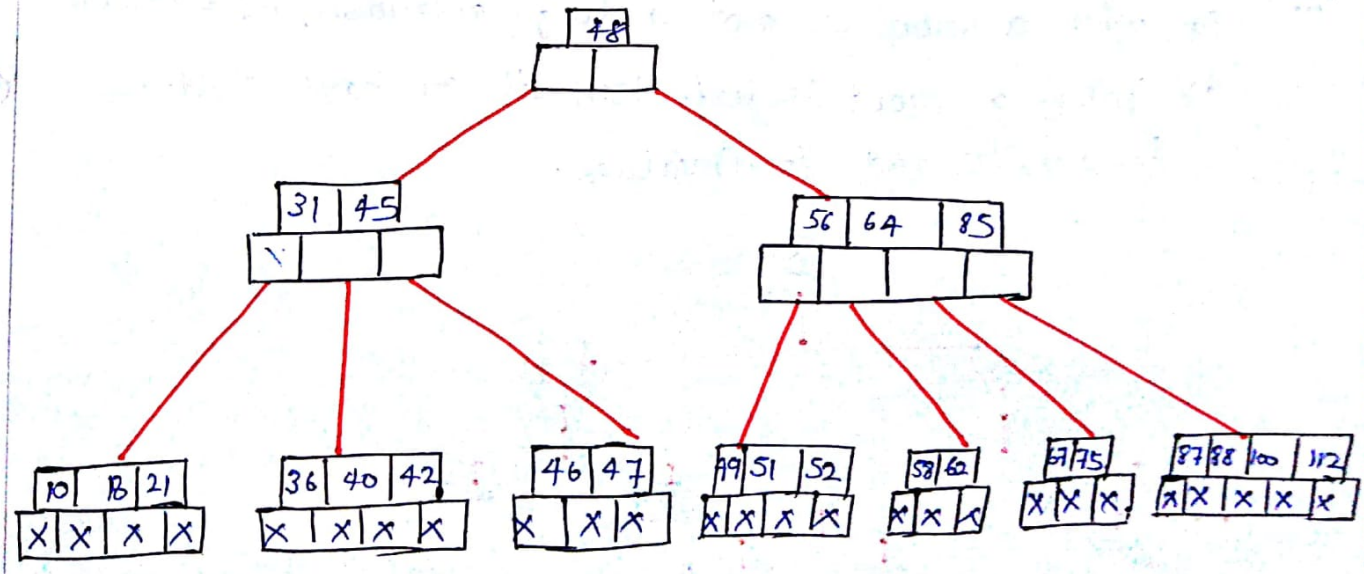
(i) the root has atleast two child nodes & at most m.

(ii) the internal nodes except the root have at least half-full $\lceil m/2 \rceil$ child nodes & at most m child nodes.

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

(iv) all leaf nodes are on same level.

B-tree of order 5



Lecture -39

B+ Tree

→ data is processed randomly by using B-tree by almost all popular file Organization methods.

→ In Conditions where the data to be processed is sequential, the processing time is high due to moving up & down in the tree. B+ tree introduced.

→ also a form of balanced tree where all the paths from the root till the leaf of tree are of same length.

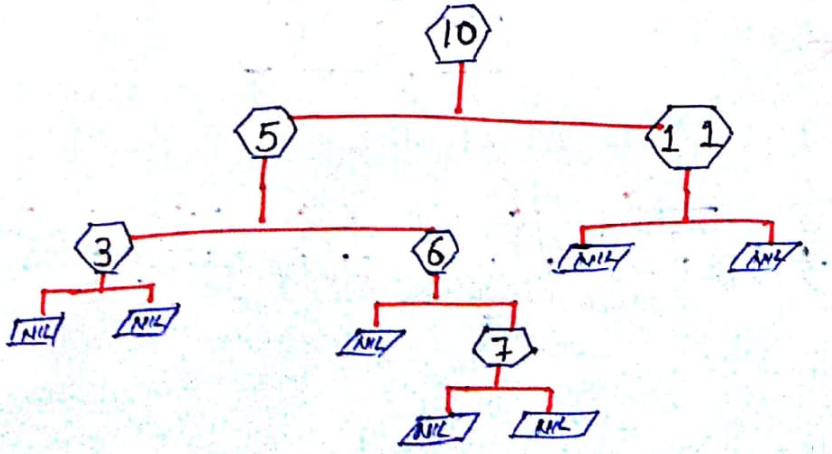
→ B+ tree as a rooted tree which satisfies the following Conditions:

(i) from the root to leaf all paths should be of same length

(ii) When a node is not either a root or leaf then it has children between $[n/2]$ and n.

(iii) (a) when a root is not a leaf, minimum of 2 children

(b) when a root is leaf, then it can have children between 0 and $(n-1)$ values.



B-Tree

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

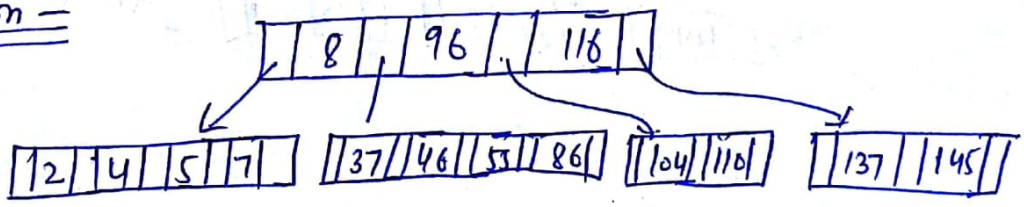
i) the root has at least two child nodes at most m child nodes

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

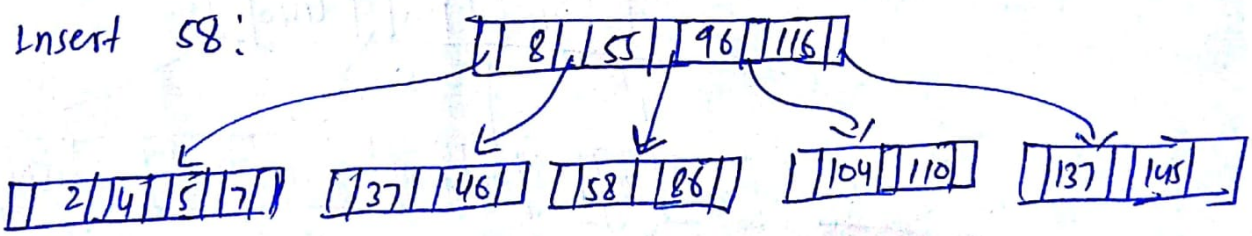
iii) the number of keys in each internal node is one less than the number of child nodes and these keys partition the keys in the subtrees of the node in a manner similar to that of m-way search trees.

iv) all leaf nodes are on the same level.

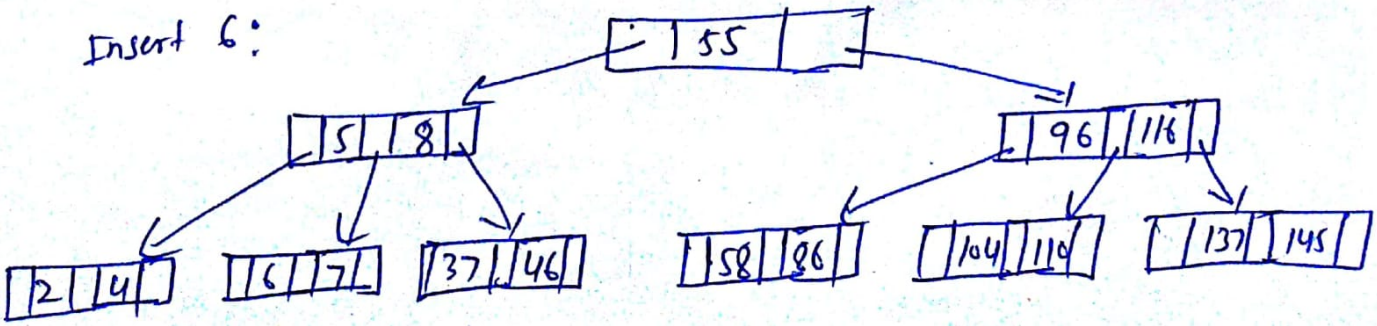
Insertion –



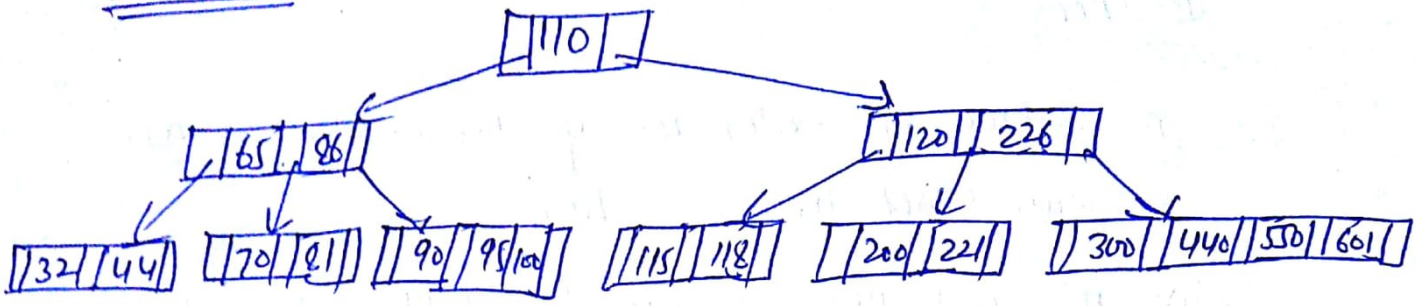
Insert 58:



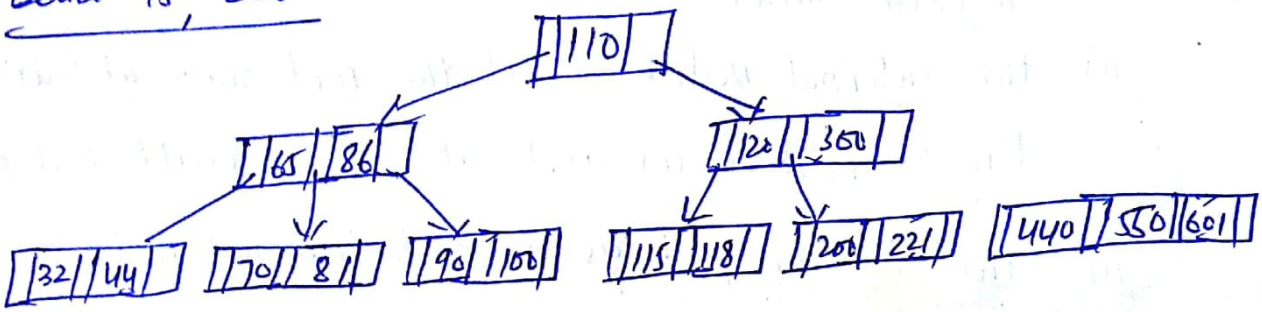
Insert 6:



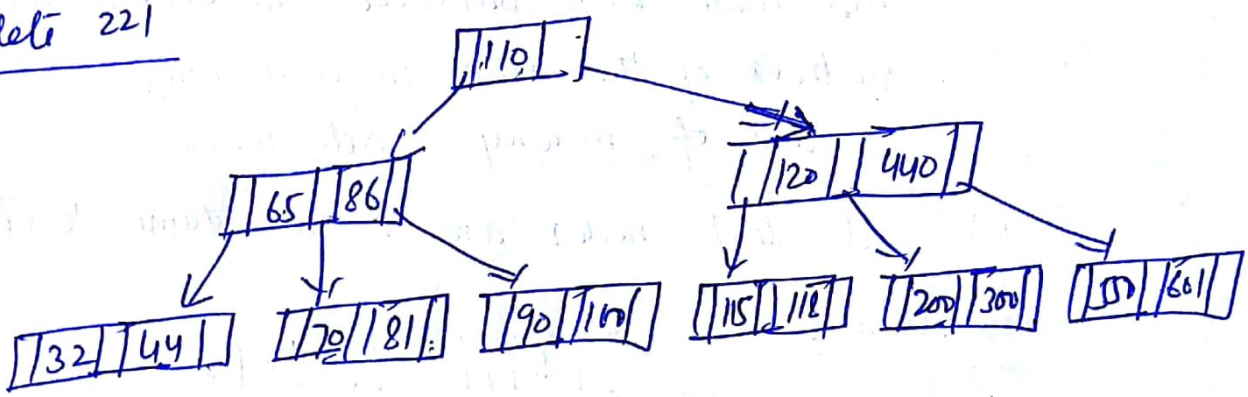
Deletion



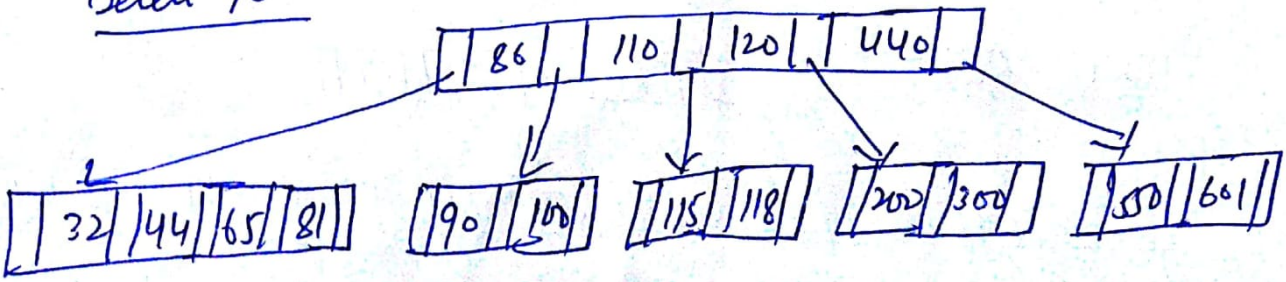
Delete 95, 226



Delete 221



Delete 70



Lecture - 40

Hashing:

→ is the transforming of a string of characters into a usually shorter fixed length value or key that represents the original string

→ used to index & retrieve items in a db because it is faster to find the item using shorter hashed key than to find it using original value.

→ Hash function is used to index the original value or key & then used later each time the data associated with the value or key is to be retrieved

→ hashing is always a one-way operation

→ some simple hash functions are:

(1) Division method: choose a number m larger than the no. n of keys in K. (m is usually choosen to be prime)

$$H(K) = K \pmod m \quad \text{or} \quad H(K) = K \pmod m + 1$$

(2) Midsquare method: 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

(3) Folding method: Key K is partitioned into a no. of parts $K_1, K_2, \ldots K_r$ where each part except possibly the last, has the same no. of digit as the required address. then the parts are added together, ignoring the last carry

$$H(K) = K_1 + K_2 + \cdots + K_r$$

(4) Subtraction method: In some cases, keys are consecutive but they don't start from 1, then subtract a number from the key to determine the address.

(5) Digit extraction method: selected keys are extracted from key and made use as its address using a method called digit extraction

Collision Resolution:

→ if hash function produce the same hash value from two different inputs, this is known as collision.

→ 2 broad ~~set~~ ways

(i) open Addressing, where an array based implementation

(ii) Separate chaining, where an array of LL based.

→ open Addressing includes:

(i) Linear probing (linear search)

(ii) Quadratic probing (non-linear search)

(iii) Double hashing (uses 2 hash functions)

Storage Management

Garbage Collection

→ also known as automatic m/m management is the automatic recycling of heap memory

→ performed by a garbage collector which recycles memory that it can prove will never be used again.

Compaction

→ automatic removal of expired data from storage area to condense the existing archive & make room for new data.