Date / /
Page No.
Shivalal

Unit-1

Analysis of Algorithms:- There are various steps involved to analysis of Algo.

1- Implement the algo completely.
2- Determine the time required for each basic operation.
- Identify unknown quantities that can be used to describe the frequency of execution of the basic operations.
- Develop a realistic model for the input to the Program
- Analyse the unknown quantities, assuming the modelled input.
Calculate the total running time by multiplying the time by the frequency for each operation, then adding all the Products. $t1 \times f1 + t2 \times f2 \cdots tn \times fn = f(n)$ (running time)

Why Analysis of Algo is important →

1- To Predict the behaviour of an algo without implementing it on a specific computer.
2- The analysis is thus only an approximation, it is not Perfect.
- By analyzing different algorithms, we can compare them to determine the best one for our purpose.

Date / /
Page No.
Shivalal

Types of Algorithm Analysis:

(i) **Best case** → Define the i/p for which algo takes minimum time. Ex In linear search when search data is present at the I^st location of large data then the best case occurs.

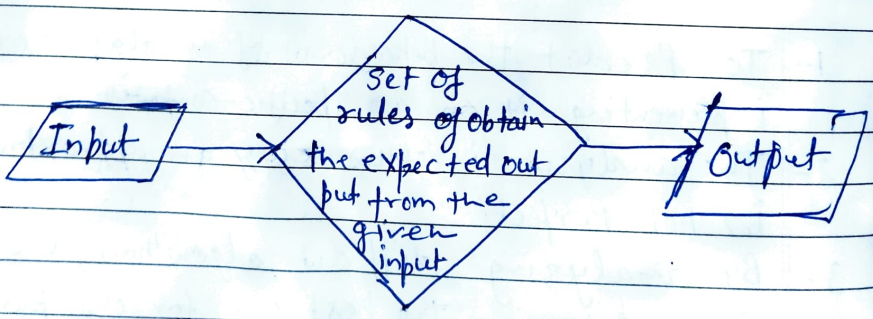
(ii) **Worst case** → Define the i/p for which algo takes a long time or maximum time. Ex In linear search when search data is not present at all then the worst case occurs.

(iii) **Average case** → In the average case take all random i/p and calculate the computation time for all inputs. And then we divide it by the total numbers of inputs.

Average case = all random case time / total no of case

$$\frac{t_1 + t_2 + t_3 \ldots t_n}{n \text{ (Total no of inputs)}}$$

Algorithm



Algorithm.

↓

is defined as a step by step Process that will be designed for a Problem.

Date / /
Page No.
Shivalal

O(f) notation represents the complexity of an algorithm, which is also termed as an Asymptotic notation or "Big O" notation.

Typical complexities of an Algorithm →

1- Constant complexity → It imposes a complexity of O(1). It undergoes an execution of a constant number of steps like 1, 5, 10 etc. for solving a given Problem.

2- Logarithmic complexity: O(log(N)). execution of the order of log(N) steps.

3- Linear complexity: (i) O(N) same number of steps as that total no of elements to implement an operation on N elements.

(ii) O(n * log(n)) → execution of the order N * log(N) on N number of elements to solve the given Problem. Ex → for a given 1000 elements the linear complexity will execute 10,000 steps for solving Problem.

$$N * \log(N)$$

$$1000 * \log_2(1000)$$

$$1000 \times \log_2(10^3)$$

$$1000 \times 3 \log_2(10)$$

$$1000 \times 3(3.32) = 1000 \times 9.96$$

$$= 10,000$$

1000 × log(1000)
1000 × 3 log 10
3000 × 3 × 3.32

$\log_2 10 = 3.32$

Date / /
Page No.
Shivalal

4- Quadratic complexity → $O(n^2)$ → it undergoes the order of N^2 count of operations on N elements for solving a given problems.
if N=100 it will endure 10,000 steps.

5- Cubic complexity → $O(n^3)$
for 100 elements, it is going to execute 1000000 steps

6- Exponential complexity → It imposes a complexity of $O(2^n)$, $O(N!)$, $O(n^k)$.

Types of Algorithm → Two types of Algorithms

(i) Iterative Algorithm → The function repeatedly runs until the condition is met or it fails. It involves the looping construct.

(ii) Recursive Algorithm → In this, the function calls itself until the condition is met. It integrates the branching structure.

But to analyze the iterative program, we have to count the number of times the loop is going to execute, whereas in recursive program, we write recursive equations i.e we write a function of F(n) in terms of F(n/2).

If the Prog is neither iterative nor recursive In that case there is no dependency of the running time on the i/p data size, the running time is going

to be a constant value. For such Programs, the complexity will be O(1).

(1) Iterative Programes →

```
A()
{
int i;
for (i=1 to n)
printf ("ABC");
}
```

Since i equals 1 to n. So this Prog will print ABC, n number of times. Complexity O(

```
A()
{
int i, j;
for(i=1 to n)
for(j=1 to n)
printf("ABC")
```

The outer loop run n time such that for each time, th inner loop will also run n times. Complexity $O(n^2)$.

2) Recursive Program →

```
A(n)
{
if (n>1)
return (A(n-1))
}
```

$T(n) = 1 + T(n-1)$ —

Step1: $T(n-1) = 1 + T(n-2)$ —

Step2: $T(n-2) = 1 + T(n-3)$ —

⋮

Basically n will start fr a very large number, and will decrease gradually. S

Date / /
Page No.
Shivalal

When $T(n) = 1$, the algo eventually stops, and such a terminating condition is called base condition or stopping condition.

Asymptotic Analysis of Algorithms (Growth of function)

To study function growth efficiently, we reduce the function down to the important part.

Let $f(n) = an^2 + bn + c$

In this function, the n^2 term dominates the function that is when n gets sufficiently large. Dominate means reducing a function. In this we ignore all constants and coefficient & look at the highest order term concerning n.

Asymptotic notation :- The word asymptotic means approaching a value or curve arbitrarily closely. Asymptotic analysis is a technique for representing limiting behavior. It can be used to analyze the performance of an algo for some large data set. Asymptotic notations are used to write fastest & slowest possible running time for an algorithm.

Importance → (i) They give simple characteristics of an algorithm's efficiency.

(ii) They allow the comparisions of the performances of various algorithms.

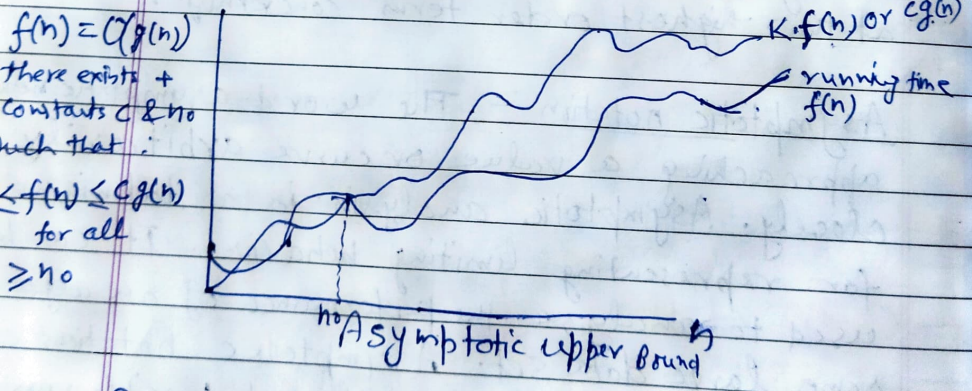
Date / /
Page No.
Shivalal

Asymptotic Notations: Three notations used to calculate the running time complexity of an algorithm.

1- **Big-Oh notation :→** It is a formal method of representing the upper bound of an algorithm running time. It is the measure of the longest amount of time. The function $f(n) = O(g(n))$ ["f of n is big-oh of g of n"]

$f(n) \leq K \cdot g(n)$ for all $n \geq n_0$

Hence function $g(n)$ is an upper bound for function $f(n)$, as $g(n)$ grows faster than $f(n)$.



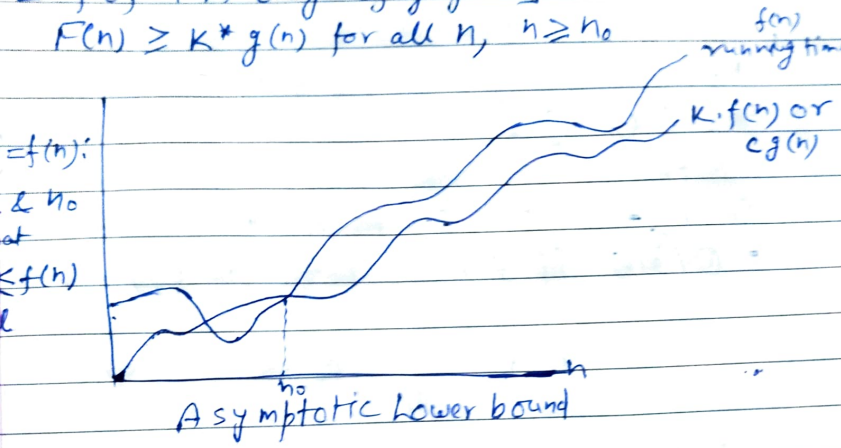
Ex $3n+2 = O(n)$ as $3n+2 \leq 4n$ for all $n \geq$

$3n+3 = O(n)$ as $3n+3 \leq 4n$ for all $n \geq 3$

Hence the complexity of $f(n)$ can be represented as $O(g(n))$.

Omega() Notation: The function $f(n) = \Omega(g(n))$
["f of n is omega of g of n"]

$F(n) \geq K * g(n)$ for all n, $n \geq n_0$



Asymptotic Lower bound

$f(n) = 8n^2 + 2n - 3 \geq 8n^2 - 3$

$= 7n^2 + (n^2 - 3) \geq 7n^2$ (g(n))

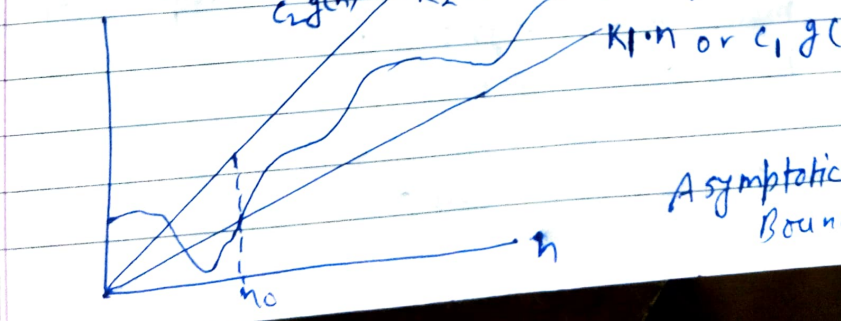
Thus, $K_1 = 7$

Hence, the complexity of f(n) can be represented as $\Omega(g(n))$

Theta(Θ): The function $f(n) = \Theta(g(n))$.

["f is the theta of g of n"]

$K_1 * g(n) \leq f(n) \leq K_2 g(n)$ for all n, $n \geq n_0$



Asymptotic Tight Bound.

Date / /
Page No.
Shivalal

Ex :- $3n+2 = \Theta(n)$ as $3n+2 \geq 3n$ and $3n+2 \leq 4n$, for n

$k_1 = 3$, $k_2 = 4$, & $n_0 = 2$

Hence the complexity of $f(n)$ can be represented as $\Theta(g(n))$.

Theta notation is more precise than both the big-oh and Omega notation. The function $f(n) = \Theta(g(n))$ if $g(n)$ is both an upper & lower bound.

Performance of an Algorithm

means predicting the resources which are required to an algorithm to perform its task. There are 3 main measurements of an algorithm performance.

1- Time efficiency.
2- Space efficiency.
3- Asymptotic dominance → comparision of cost functions when n is large.

There are basically 3 examples of performance measurements systems.

1- Balanced Scorecards.
2- ISO standards
3- Industry dashboards.

Space complexity has Two parts :-

i) A fixed part → includes instruction space, space for constants, simple variables & fixed size structure variable.

ii) A variable part → includes structured variables whose size depends on the particular problem being solved.

"Sorting & order statistics"

The sorting problem consists in the following:
Input: a sequence of n elements $(a_1, a_2 \cdots a_n)$
Output: a permutation $(a_1', a_2', \cdots, a_n')$ of the initial sequence, sorted given an ordering relation $\leq$: $a_1' \leq a_2' \leq \cdots \leq a_n'$

Example: (8,1,6,3,6,4) → Sorting Algo → (1,3,4,6,6,8)

Shell sort :- is mainly variation of insertion sort. In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. In this case, shell sort is used to allow exchange of far items. This algo avoids large shifts as in case of insertion sort. In shell sort, we make the array h-sorted

for a large value of h. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sublists of every h'th element is sorted.

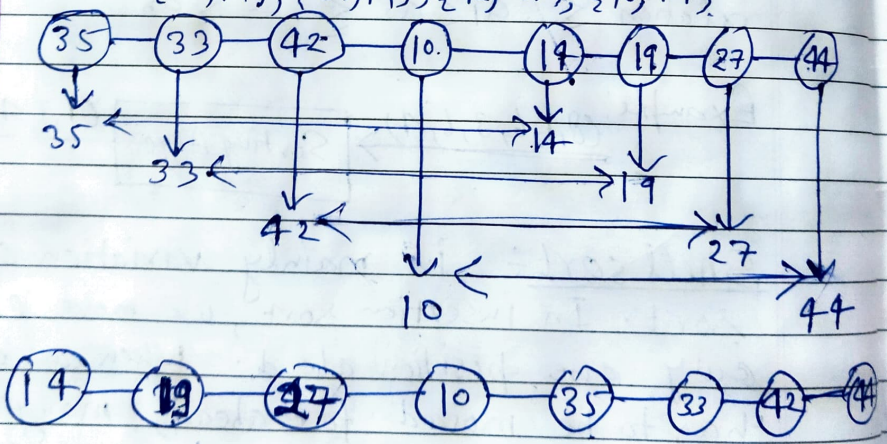
This Algo uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as interval. This interval is calculated based on "Knuth's" formula as →

Knuth's formula →

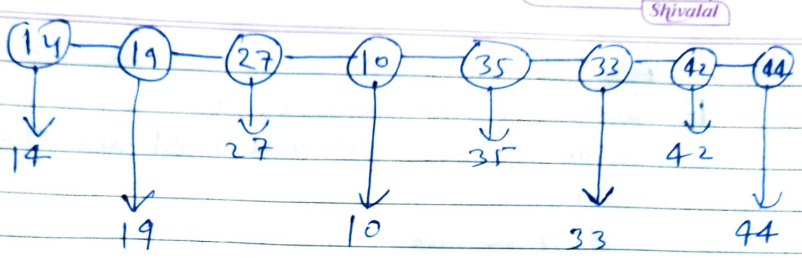
$h = h * 3 + 1$

where h - is interval with initial value 1.

{35, 14}, {33, 19}, {42, 27}, {10, 44}



After we take interval of 1 & this gap generates two sub lists — {14, 27, 35, 42}, {19, 10, 33, 44}



we compare & swap →

finally, we sort the rest of array using interval of '1'.

step-by-step →

14	19	27	10	35	33	42	44
14	19	27	10	35	33	42	44
14	19	27	10	35	33	42	44
14	19	27	10	35	33	42	44
14	19	10	27	35	33	42	44
14	10	19	27	35	33	42	44
10	14	19	27	35	33	42	44
10	14	19	27	33	35	42	44

Date / /
Page No.
Shivalal

Algo →

1- Initialize the value of h.
2- Divide the list into smaller sub-list of equal interval h.
3- Sort these sub-lists using insertion sort.
4- Repeat until complete list is sorted.

Quick sort Algo

↓

It is an algo of "Divide & conquer type"

Divide → split arrays into 2 sub-arrays. Each element in left sub array is less than or equal to the average element & each element in the right sub-array is larger than the middle element.

Conquer → Recursively, sort two sub arrays.

Combine → combine the already sorted array.

/* low -> Starting index, high -> Ending index */

Algorithm →

```
QUICKSORT(array A, int m, int n)
1- if (n>m)
2- then
3- i ← a random index from [m, n]
4- swap A[i] with A[m]
5- o ← PARTITION (A, m, n)
6- QUICKSORT(A, m, o-1)
7- QUICKSORT(A, o+1, n)
```

```
uicksort(arr[], low, high)
if (low < high)
Pi is partitioning index,
rr[Pi] is now at right place */
i = partitioning(arr, low, high);
ickSort(arr, low, Pi-1) //Before Pi
ickSort(arr, Pi+1, high) //After Pi
```

Date / /
Page No.
Shivalal

Partition Algo → rearrange the sub arrays in a place

PARTITION(array A, int m, int n)
arr[] low high

1- $X \leftarrow A[m]$
2- $O \leftarrow m$
3- for $P \leftarrow m+1$ to n
4- do if $(A[P] < x)$
5- then $O \leftarrow O+1$
6- Swap $A[O]$ with $A[P]$
7- Swap $A[m]$ with $A[O]$
8- return O

Pivot = arr[high]
i = (low - 1) // index of smaller elements and indicates the right Position of Pivot found so far
for (j = low; j <= high - 1; j++)
if (arr[j] < pivot)
i++
swap arr[i] and arr[j]
swap arr[i+1] and arr[high]
return (i+1)

Ex 44 33 11 55 77 90 40 60 99 22 88

Let 44 be the Pivot (center role, imp) element & scanning from right to Left

Compare 44 to the right side element if right elemeht are smaller than 44 then swap it.

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

Now compare 44 to the left side element it should be greater than 44 then swap them.

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

Recursively, repeating step (1) & (2) untill we get two lists one left from Pivot element (44) & one right from Pivot element.

Date / /
Page No.
Shivalal

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

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

Now we get two sorted lists:

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

Sublist 1 Sublist 2

After that these sublists are sorted under the same process as done previously.

Pivot

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

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

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

88 77 60 (55) (90) 99

first sorted list.

Sublist 3 ~~sublist 4~~

Pivot

(88) 77 60 55 90 99

(55) 77 60 (88) 90 99

Pivot Sublist 5 Sublist 6

(55) 77 60

55 60 77 88 90 99

Date / /
Page No.
Shivalal

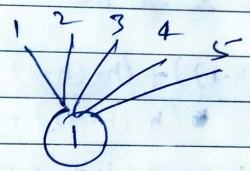
Merging sublists →

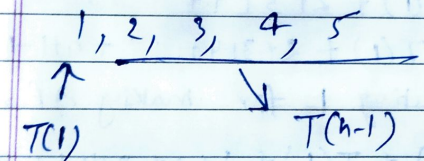
11 22 33 40 44 55 60 7.7 88 90 99

Worst case Analysis → if item are already in sorted form & we try to sort them again. This will take lots of time & space.

$T(n) = T(1) + T(n-1) + n$

- $T(1)$: time taken by Pivot element
- $T(n-1)$: remaining element
- n: no of comparisions to identify exact position of itself





Rational formula of worst case →

$T(n) = T(1) + T(n-1) + n$ ——— (eq)

$T(n-1) = T(1) + T(n-1-1) + (n-1)$ (Put $(n-1)$ in place of n)

Put $T(n-1)$ in eq (1)

$T(n) = T(1) + T(1) + (T(n-2) + (n-1) + n$ ---(ii)

$T(n) = 2T(1) + T(n-2) + (n-1) + n$

$T(n-2) = T(1) + T(n-3) + (n-2)$

worst case will happen when array is sorted.

Date / /
Page No.
Shivalal

Put $T(n-2)$ in equation (ii)

$T(n) = 2T(1) + T(1) + T(n-3) + (n-2) + (n-1) + n$

$T(n) = 3T(1) + T(n-3) + (n-2) + (n-1) + n$

$T(n-3) = T(1) + T(n-4) + n-3$ (By put $(n-3)$ in place of n in eq(i).

$T(n) = 3T(1) + T(1) + T(n-4) + (n-3) + (n-2) + (n-1) + n$

$= 4T(1) + T(n-4) + (n-3) + (n-2) + (n-1) + n$ ---(iii)

for making $T(n-4)$ as $T(1)$ we will put $(n-1)$ in place of '4' & if we put $(n-1)$ in place of 4 then we have to put $(n-2)$ in place of 3 & $(n-3)$ in place of 2 & so on.

$T(n) = (n-1)T(1) + T(n-(n-1)) + (n-(n-2)) + (n-(n-3)) + (n-(n-4)) + n$

$T(n) = (n-1)T(1) + T(1) + 2 + 3 + 4 + \cdots n$

$T(n) = (n-1)T(1) + T(1) + 2 + 3 + 4 + \cdots + n + 1 - 1$

Adding 1 & subtracting 1 for making AP series

$T(n) = (n-1)T(1) + T(1) + 1 + 2 + 3 + 4 + \cdots + n - 1$

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

Stopping condition: $T(1) = 0$

Because at last there is only one element left & no comparision is required.

$T(n) = (n-1)(0) + 0 + \frac{n(n+1)}{2} - 1$

$T(n) = \frac{n^2+n-2}{2} = O(n^2)$ so worst case complexity $T(n) = O(n^2)$

Date / /
Page No.
Shivalal

Randomized Quick Sort [Average case] → In average case the number of chances to get a pivot element is equal to the no of items.

Let total time taken = $T(n)$

Ex $P_1, P_2, P_3, P_4 \dots P_n$

if P_1 is the pivot list then we have 2 lists. i.e $T(0)$ & $T(n-1)$

if P_2 is the pivot list then we have 2 lists i.e $T(1)$ & $T(n-2)$

$P_1, P_2, P_3, P_4 \dots P_n$

if P_3 is the Pivot element then we have 2 lists i.e $T(2)$ & $T(n-3)$

$P_1, P_2, P_3, P_4 \dots P_n$

So, in general if we take the K^{th} element to be the pivot element.

Then,

$$T(n) = \sum_{K=1}^{n} T(K-1) + T(n-K)$$

Pivot element will do n comparision & we are doing average case so,

$$T(n) = n+1 + \frac{1}{n}\left(\sum_{K=1}^{n} T(K-1) + T(n-K)\right).$$

↑ N comparision

Average of n elements

Date / /
Page No.
Shivalal

So, Relational formula for Randomized Quick sort is:

$$T(n) = n+1 + \frac{1}{n}\left(\sum_{k=1}^{n} T(k-1) + T(n-k)\right)$$

$$= n+1 + \frac{1}{n}\left(T(0)+T(1)+T(2)+\cdots T(n-1) + T(n-2) + T(n-3) + \cdots T(0)\right)$$

$$= n+1 + \frac{1}{n} \times 2\left(T(0)+T(1)+T(2)+\cdots T(n-2)+T(n-1)\right)$$

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

$$n\,T(n) = n(n+1) + 2\left(T(0)+T(1)+T(2)+\cdots+T(n-1)\right) \cdots (1)$$

Put $(n-1)$ in eq (1)

$$(n-1)\,T(n-1) = (n-1)\,n + 2\left(T(0)+T(1)+T(2)+\cdots T(n-2)\right) \cdots (2)$$

from (1) & (2) subtraction

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

Average case :

$$O(n) = O(n \log n)$$

Best case : $O(n \log n)$

(3) Best case → we dont make any comparision b/w elements that is only done when we have only one element to sort.

Date / /
Page No.
Shivalal

Merge Sort

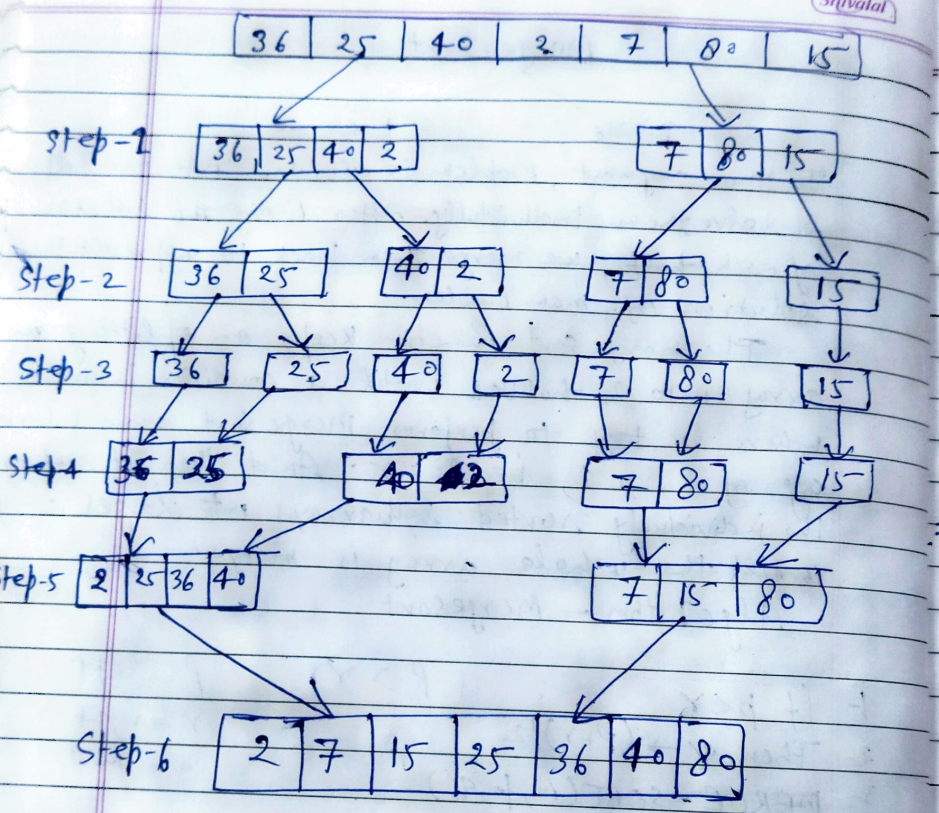
Divide. Conquer

In this segment, Problem is divided into two halfs & solve them individually. After finding the solution of each half, we merge them back to represent the solution of main Problem.

The mergesort function keeps on splitting an array into 2 halves until a condition is met where we try to perform MergeSort on a subarray of size 1, i.e $p == r$. And then it combines the individually sorted subarrays into larger arrays until the whole array is merged.

Algorithm – Mergesort

1- if $p < r$ if $P > r$. p = left
2- Then $q \rightarrow (p+r)/2$ r = right
3- MERGE-SORT(A, p, q) q = mid.
4- MERGE-SORT(A, q+1, r)
5- MERGE(A, p, q, r)



Hence the array is sorted.

Step-1 The merge sort Algo divides an array into equal halves until we achieve an atomic value. If there are odd no of elements in array then one of the halves will have more elements than the other half.

Date / /
Page No.
Shivalal

Hurdle (interfering)

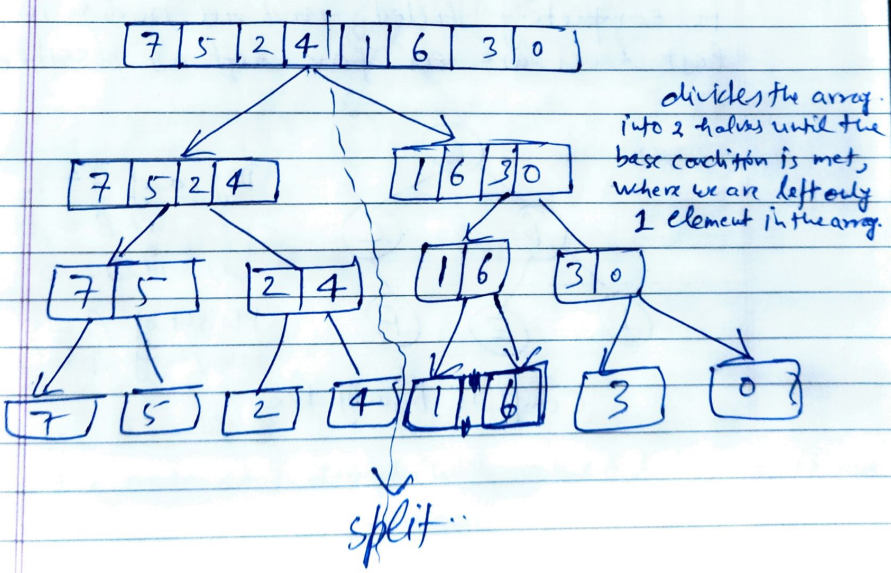
tep-2 After dividing 2 subarrays, if it did not hamper the order of elements as they were in the original array, then we will further divide into 2 halves

ep-3 Again, we will divide these arrays until we achieve an atomic value, i.e a value that can not be further divided.

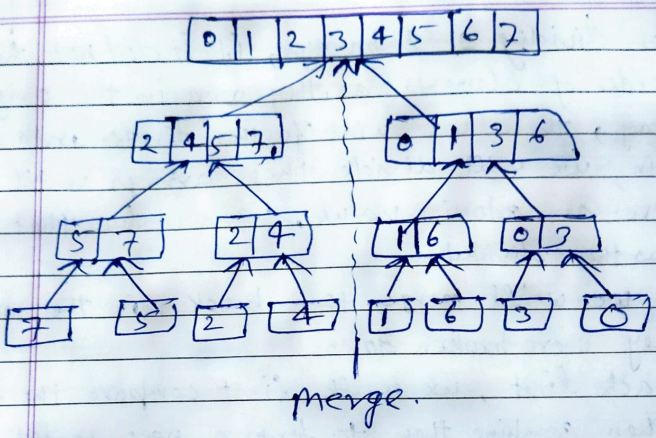
tep-4 Next, we will merge them back in the same way as they were broken down.

tep-5 for each list, we will first compare the element and then combine them to form a new sorted list.

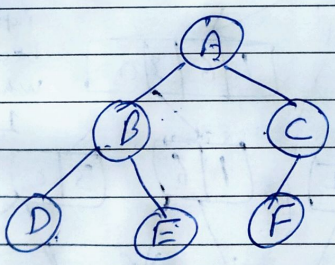
tep-6 In the next iteration, we will compare the lists of two data values & merge them back in a sorted manner.



divides the array into 2 halves until the base condition is met, where we are left only 1 element in the array.



Complete binary tree → is a binary tree in which every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.



Degree of A = 2
" " B = 1
" " D = 0

Height of A = 0
Height of D = 2
} no of edges

Complete binary tree

1-m

61, 2 → M (Saturday)

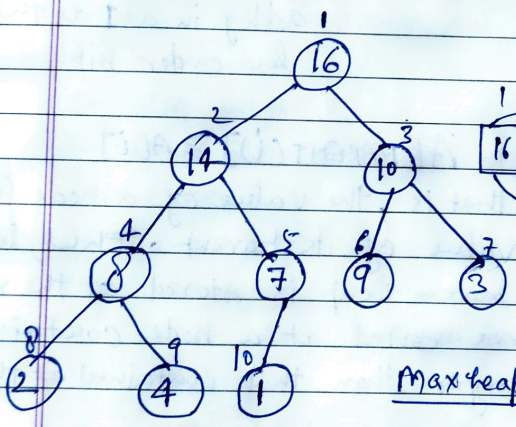
Date / /
Page No.
Shivalal

Heap sort

The (binary) heap data structure is an array object that we can view as a nearly complete binary tree. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. An array A that represents a heap is an object with two attributes

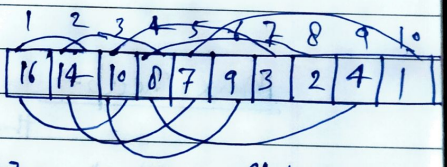
1- A.length → gives the number of elements in the array

2- A.heap-size → represents how many elements in the heap are stored within array A.



(a)

Binary tree



(b)

An array

Max heap -

height → 3

The node at index 4 (with value 8) has height one.

Date / /
Page No.
Shivalal

```
PARENT (i)
1  return ⌊i/2⌋

Left (i)
1  return 2i

Right (i)
1  return 2i+1
```

Left Procedure → Can compute $2i$ in one instruction by simply shifting the binary representation of i left by one bit position.

Right Procedure → $2i+1$ by shifting the binary representation of i left by one bit position & then adding in a 1 as the low order bit.

Max heap → $A[PARENT(i)] \geq A[i]$

that is, the value of a node is at most the value of its parent. Thus, largest element in a max-heap is stored at the root and the subtree rooted at a node contains values no larger than that contained at the node itself.

Min-heap → $A[PARENT(i)] \leq A[i]$

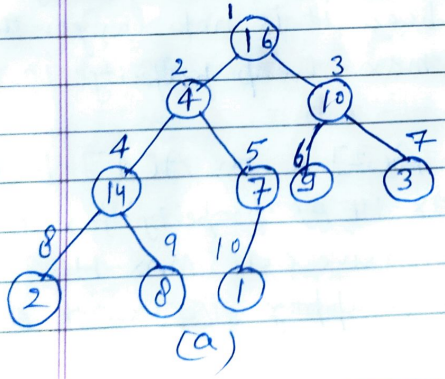
The smallest element in a min-heap is at the root.

Date / /
Page No.
Shivalal

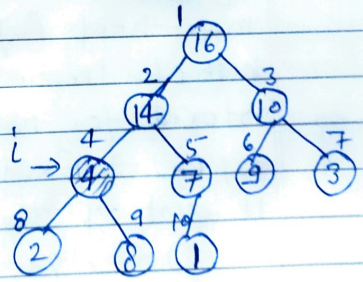
Heap Property → (Max heap) Procedure MAX-HEAP(A,i)

MAX-HEAP assumes that the binary tree rooted at LEFT(i) and RIGHT(i) are max heap, but that A[i] might be smaller than its children, thus violating the max heap property.

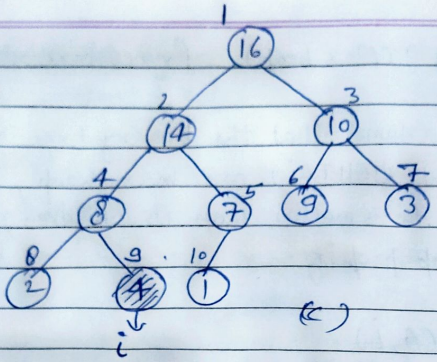
```
MAX-HEAP(A, i)
1-   l = LEFT(i)
2-   r = RIGHT(i)
3- if l ≤ A.heap-size and A[l] > A[i]
4-   largest = l
5- else largest = i
6- if r ≤ A.heap-size and A[r] > A[largest]
7-   largest = r
8- if largest ≠ i
9-   exchange A[i] with A[largest]
10-  MAX-HEAP(A, largest)
```



(a)



(b)



(c)

In this figure the largest of the elements A[i], A[LEFT(i)] and A[RIGHT(i)] is determined and its index is stored in largest. If A[i] is largest, then the subtree rooted at node i is already a max heap and the procedure terminates. Otherwise, one of two children has the largest element. In this case A[i] is swapped with A[largest]

fig(a) → A[2] at node i=2 violating the max-heap property bcoz it is not larger than both children. The max-heap property is restored for node 2.

fig(b) → Exchange A[2] with A[4] but it destroy the MAX-HEAP property for node 4.

fig(c) → Again recursive call MAX-HEAP(A,4) now has i=4, After swapping A[4] with A[9] Now node 4 is fixed up, and the recursive call

MAX-HEAP(A,9) give no further change to t
data structure.

RED-BLACK TREE

A Red-black tree is a BST with one extr
bit of storage per node; its color which ca
be either RED or BLACK. Red-black tree en
that no such path is more than twice as long
any other, so that the tree is approximatel
balanced.

Each node of the tree now contains the attri
color, key, left, right and p.

Properties →

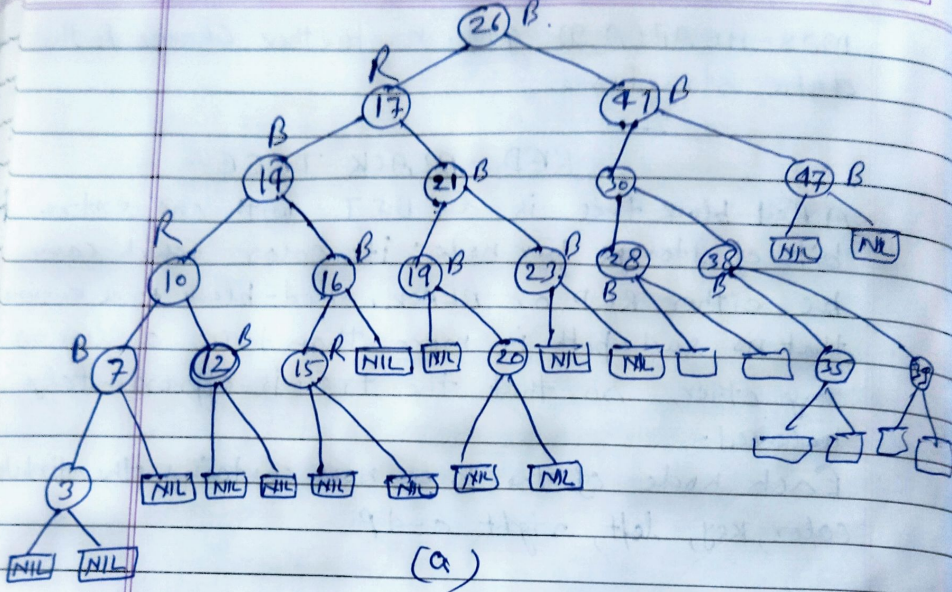
1- Every node is either red or black.
2- The root is always black.
3- Every leaf (NIL) is black.
4- If a node is red, the both its children are bla
5- for each node, all simple paths from the nod
to descendant leaves contain the same numbe
of black nodes

→ Note: can not have 2 consecutive reds on
path

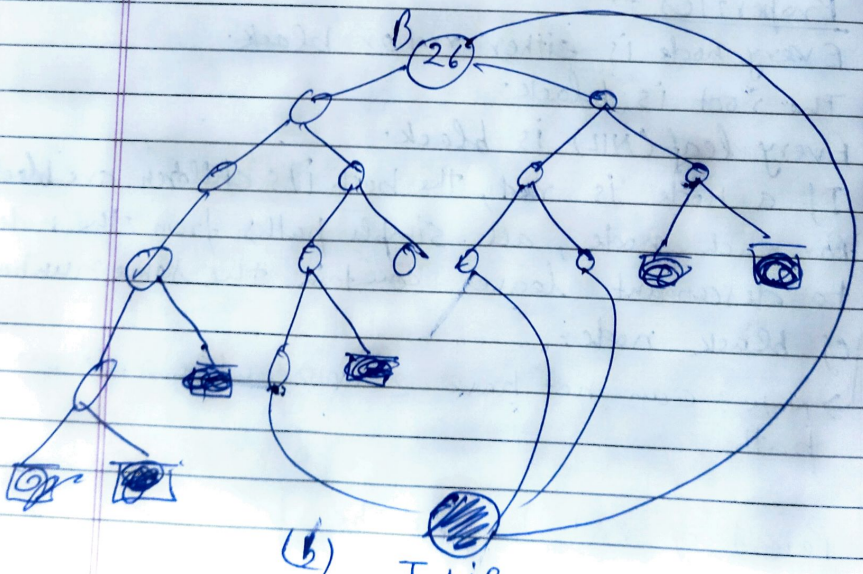
Parent of red should be black.

7D178585 39

Date / /
Page No.
Shivalal



(a)



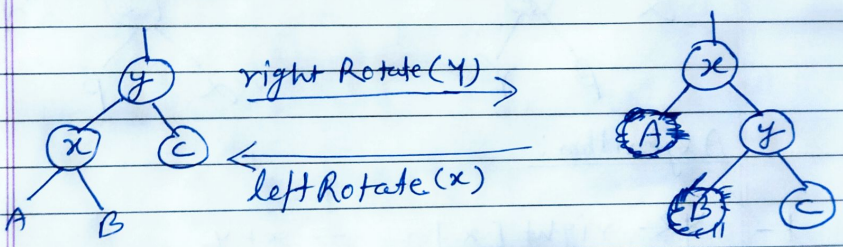
(b) T. nil.

Date / /
Page No.
Shivalal

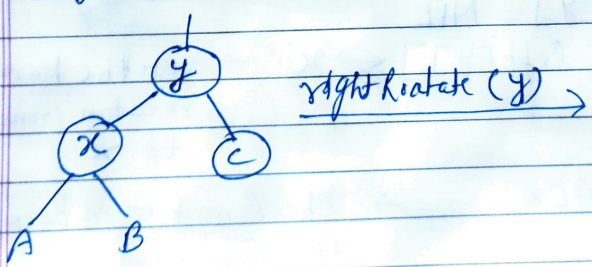
(a) Every leaf shown as a nil is black.
(b) same red black tree but with each NIL replaced by the single sentinel T.nil, which is always black. The root's parent is also the sentinel.
(c) same Red Black tree but with leaves and the roots parent omitted entirely.

Rotation

Basic operation that is used for changing tree structure is called rotation.



Right Rotation →

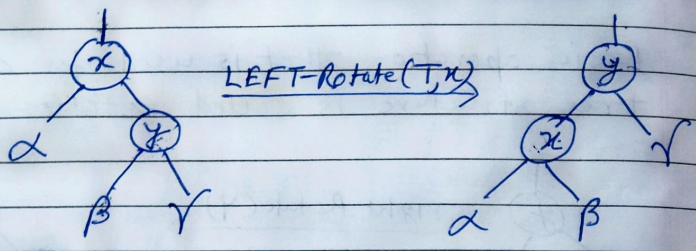


Date / /
Page No.
Shivalal

Algorithm: A lot of pointer manipulation

1- x keeps its left child
2- y keeps its right child
3- x's right child becomes y's left child.
4- x's and y's parents change.

Left Rotation:



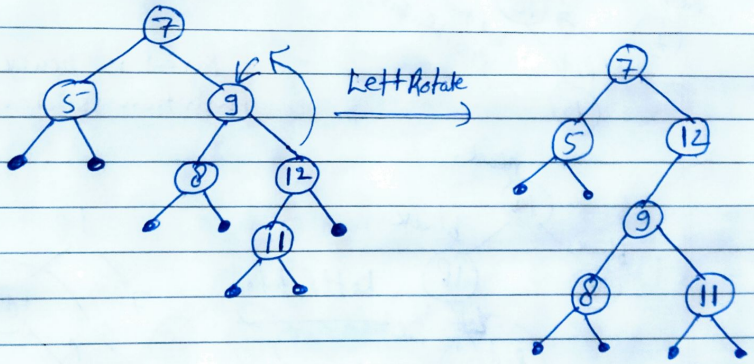
Algorithm:

1- $y \leftarrow right[x]$ → set y
2- $right[x] \leftarrow left[y]$ → y's left subtree becomes x's right subtree
3- if $left[y] = NIL$
4- then $p[left[y]] \leftarrow x$ → set the parent relation from left[y] to x.
5- $p[y] \leftarrow p[x]$ → The Parent of x becomes the Parent of y.

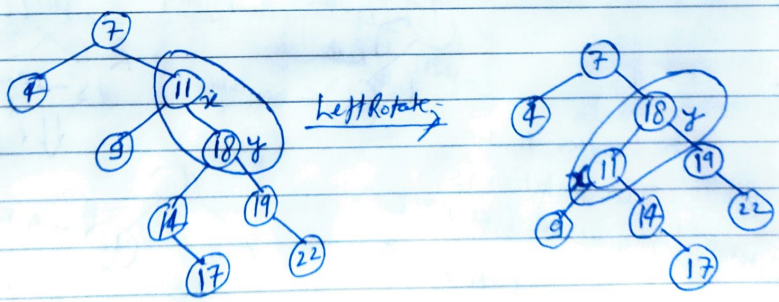
Date / /
Page No.
Shivalal

6- if P[x] = NIL
7- then root[T] ← y
8- else if x = left[P[x]]
9- then left[P[x]] ← y
10- else right[P[x]] ← y
11- left[y] ← x ⟶ Put x on y's left
12- P[x] ← y ⟶ y becomes x's parent

Ex 1 Rotate left about 9.



Ex 2

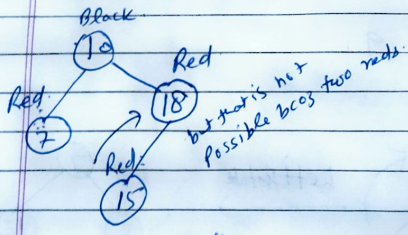


Date / /
Page No.
Shivalal

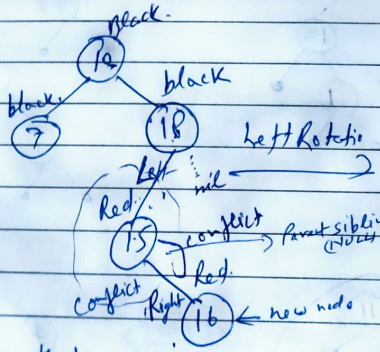
Insertion

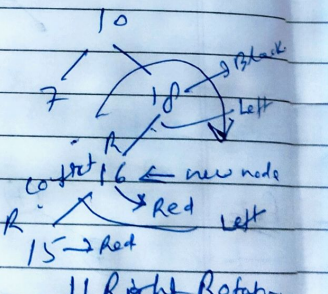
3 Rules to implement:
- root = Black.
- no two adjacent red nodes.
- count no of black nodes in each Path.

10, 18, 7, 15, 16, 30, 25, 40, 60, 2, 1, 70



if Parent is black the insertion is done Rexit

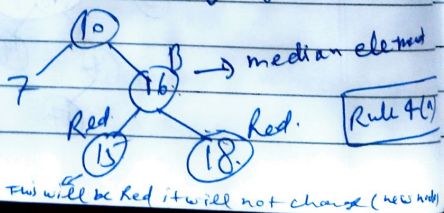




Right Rotation

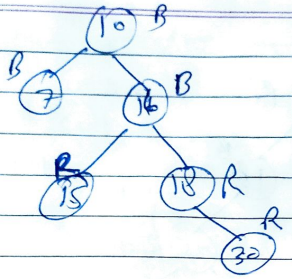
LR Rotation

After that Two Rotation is required like AVL Tree
(i). first Left Rotation
(ii) second Right Rotation

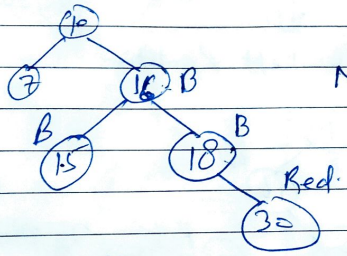


Two will be Red it will not change (new node)

16 → black
18 → Red
} After recolor

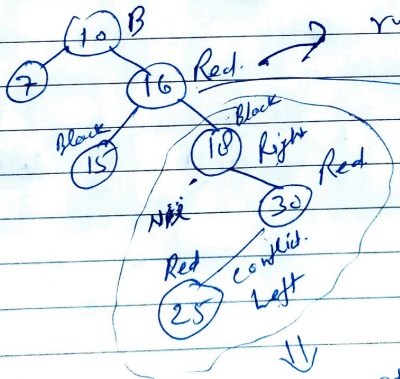


No Rotation is required because Parent's sibling is same color. do recolor.

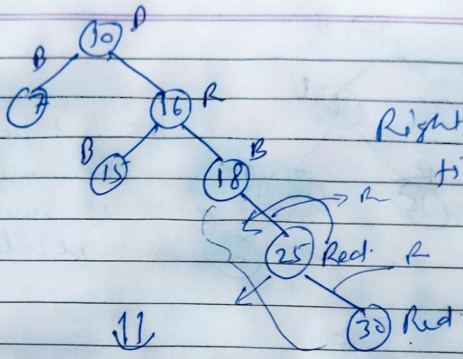


No rotation According to rule 4(b) recolor only

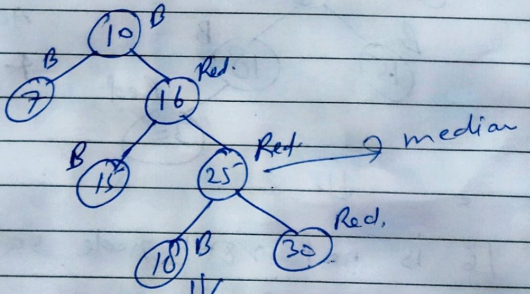
16 is not root node so it becomes Red



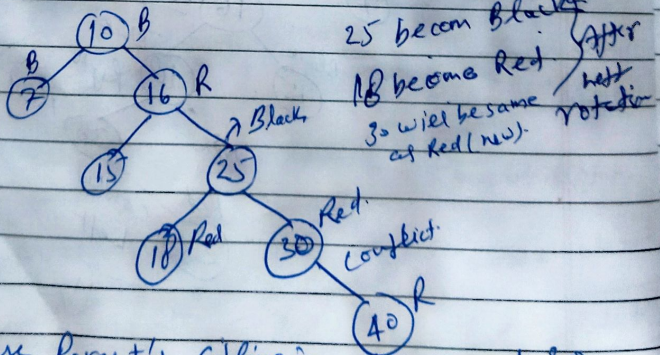
Two rotation will be there (R L)



Now Left Rotation



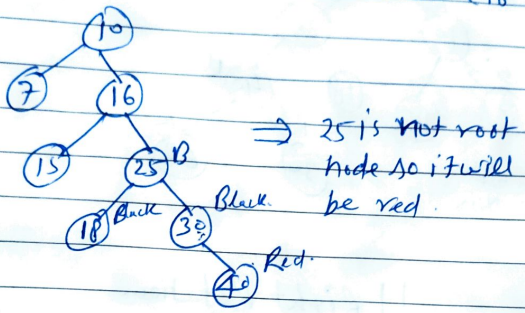
After recoloring.

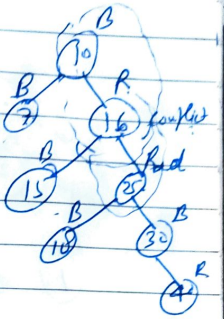


in this case Parent's Sibling is also red so no rotation is required so just recoloring.

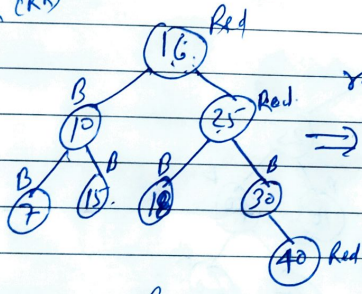
Date / /
Page No.
Shivalal

So we recolor both node (18 & 30).

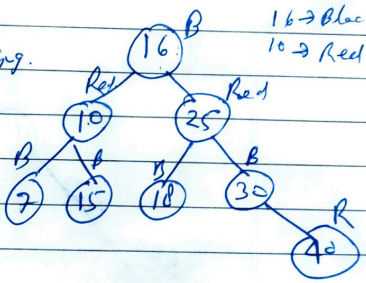




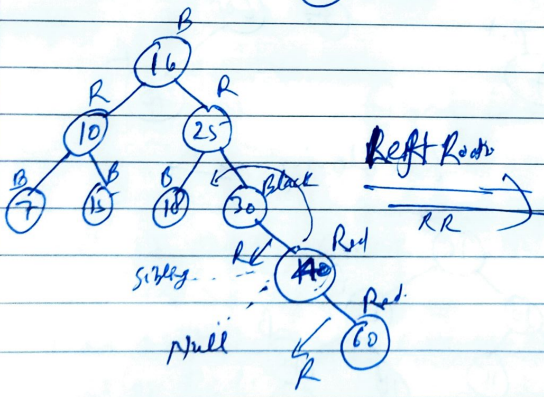
Rotation (RR)

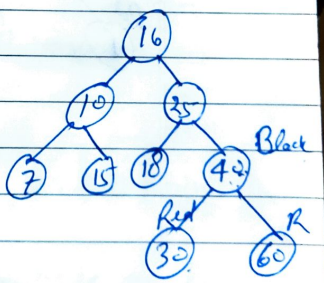


recoloring ⇒



16 → Black
10 → Red.

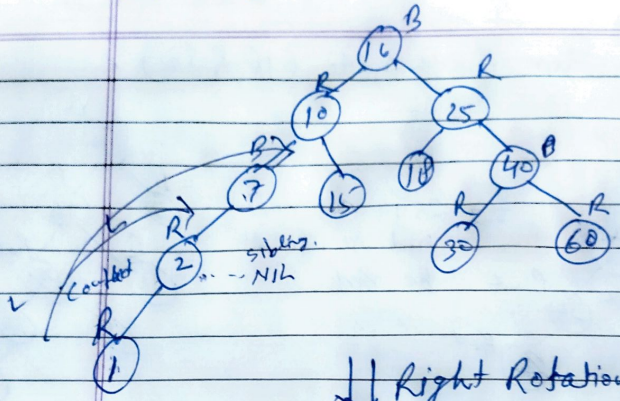




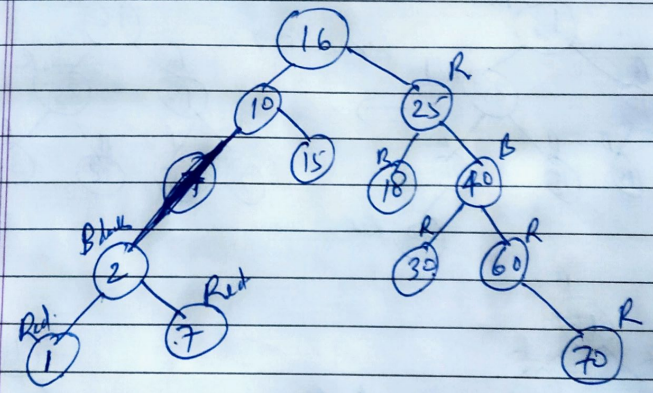
Change { 40 → Black
30 → Red

RBT → BST

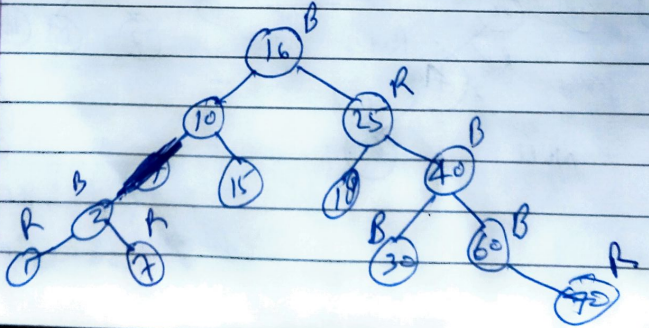
but every BST ≠ RBT



↓↓ Right Rotation.

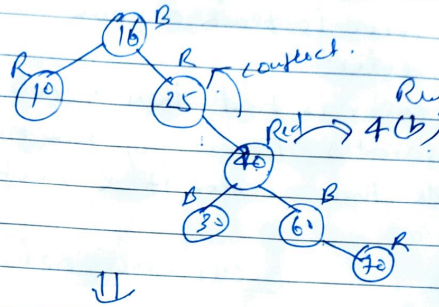


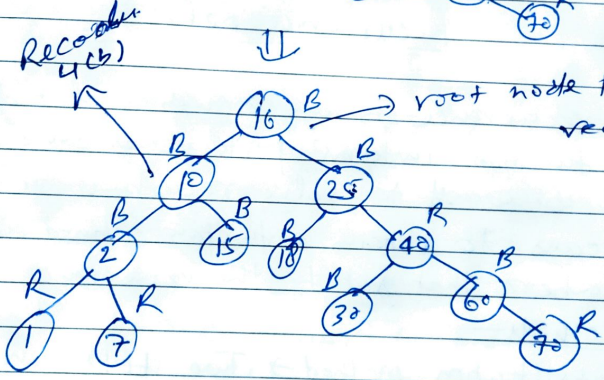
Recoloring only.



Application → RB tree is used for Linux Kernel in the form of scheduling Algorithm. It is used for faster insertion, retrivals.

But Parent's Parent (40) is B so recolor 40 as Red.





final Black tree.

Rules →

1- If tree is empty, create new node as root node with color black.

2- If tree is not empty, create new node as leaf node with color Red.

3- If Parent of new node is black then exit.

4- If Parent of new node is Red, then check the color of Parent's sibling of new node:

(a) if color is black or null then suitable rotation & recolor

(b) if color is Red then recolor & also check if Parent's Parent of new node is not root node then recolor it & recheck.

"Recurrence Relation"

A recurrence is an equation that describes a function in terms of its values on smaller i/p. If we want to solve recurrence relation it means to obtain a function defined on natural numbers that satisfy the recurrence.

Methods:-

1- Substitution method → Two steps are there.

(a) Guess the solution (b) Use the mathematical induction to find boundary condition & shows that guess is correct.

Ex-1 Solve the equation by substitution method:

$$T(n) = T\left(\frac{n}{2}\right) + n$$

(we have to show that it is asymptotically bound by $O(\log n)$)

Recurrence eq.

Solution → for $T(n) = O(\log n)$

$T(n) \le c \log n$ (we have to show that for some constant c)

Put this in eqⁿ)

$T(n) \le c \log(\frac{n}{2}) + 1 = c \log n - c \log_2 2 + 1$

$\le c \log n$ for $c \ge 1$

Thus $T(n) = O \log n$

Ex-2 Consider the Recurrence

$T(n) = 2T(\frac{n}{2}) + n \quad n > 1$ Find an Asymptotic bound on T.

Soln - We guess the solution is $O(n(\log n))$. Thus for constant 'c'.

$T(n) \le c\, n \log n$

Put this in given Recurrence Equation.

Now, $T(n) \le 2c(\frac{n}{2}) \log(\frac{n}{2}) + n$

$\le cn \log n - cn \log 2 + n$

$= cn \log n - n(c \log 2 - 1)$ $\quad \log_2 2 = 1$

$\le cn \log n$ for $(c \ge 1)$

Thus $T(n) = O(n \log n)$.

(2-) Iteration Methods → It means to expand the recurrence and express it as a summation of terms of n & initial condition.

Ex-1 $T(n) = 1$ if $n = 1$

$= 2T(n-1)$ if $n > 1$

Soln → $T(n) = 2T(n-1)$

$= 2[2T(n-2)] = 2^2 T(n-2)$

$= 4[2T(n-3)] = 2^3 T(n-3)$

$= 8[2T(n-4)] = 2^4 T(n-4)$ ——— (1)

Repeat the procedure for i times.

$T(n) = 2^i T(n-i)$

Put $n-i=1$ or $i=n-1$ in ——— (1)

$T(n) = 2^{n-1} T(1)$

$= 2^{n-1} \cdot 1$ {$T(1)=1$ ——— given}

$= 2^{n-1}$

Ex-2 Consider the Recurrence

$T(n) = T(n-1)+1$ and $T(1) = \Theta(1)$

Soln $T(n) = T(n-1)+1$

$= (T(n-2)+1)+1 = (T(n-3)+1)+1+1$

$= T(n-4)+4 = T(n-5)+1+4$

$= T(n-5)+5 = T(n-k)+k$

Where $k = n-1$

$T(n-k) = T(1) = \Theta(1)$

$T(n) = \Theta(1) + (n-1) = 1+n-1 = n = \Theta(n)$

(3) Recursion Tree Method →

(a) It is useful when divide & conquer algo is used.

(b) In this, each root & child represents the cost of a single subproblem.

(c) Sum all per-level costs to determine the total cost of all levels of the recursion.

(d) Recursion tree is best to generate good guess, which can be verified by Substitution method.

Delete 'D' →

Delete 'B' →

$1 + \frac{1}{2} + \frac{1}{4}$

$\frac{1(1-(\frac{1}{2})^3)}{1-\frac{1}{2}}$

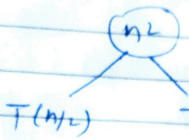
Date / / Page No.

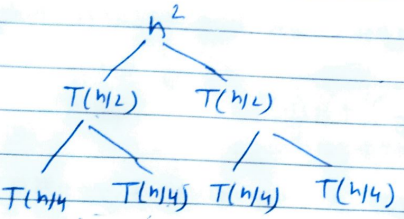
Shivalal

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

we have to obtain the asymptotic bound using recurrence tree method.

Soln



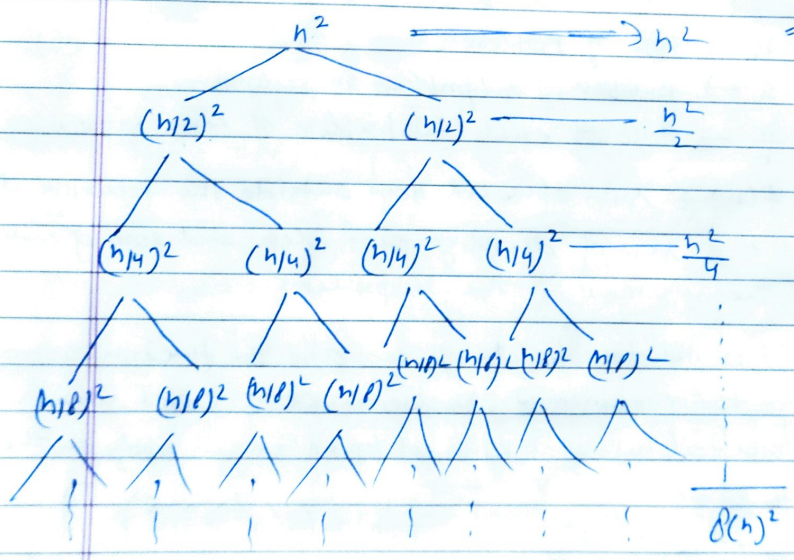


G.P summation

$S = \frac{a(r^n - 1)}{r-1}$

$2^0 + 2^1 + 2^2 + 2^3 + \cdots$

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



$T(n) = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \cdots$ logn times.

$\leq n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2^i}\right) \leq n^2 \left(\frac{1}{1-\frac{1}{2}}\right) \leq 2n^2$

$T(n) = \Theta n^2$ (G.P)

Date / /
Page No.
Shivalal

Master Method :- is used to solve the following Types of recurrence.

$T(n) = aT(\frac{n}{b}) + f(n)$ with $a \geq 1$ & $b \geq 1$ be constant & $f(n)$ be a function and $\frac{n}{b}$ can be interpreted as.

Let $T(n)$ is defined on non-negative integers by the recurrence

$T(n) = a\,T\left(\left\lfloor \frac{n}{b} \right\rfloor\right) + f(n)$

In the analysis of a recursive algorithm, the constants and function will be on the following sequence

(i) $n \rightarrow$ size of Problem

(ii) $a \rightarrow$ number of subproblems in recursion

(iii) $\frac{n}{b} \rightarrow$ size of each subproblem of same size.

(iv) $f(n) \rightarrow$ sum of work done outside the recursive call means sum of dividing the problem and sum of combining the solutions to the subproblems.

(v) It is not possible always to bound the function according to the requirement, so we make 3 cases which will tell us what kind of bound we can apply on the function.

$T(n) = \Rightarrow$ then.

$\Theta(n^{\log_b a})$	$f(n) = O(n^{\log_b a - \varepsilon})$
$\Theta(n^{\log_b a} \log n)$	$f(n) = \Theta(n^{\log_b a})$
$\Theta(f(n))$	$f(n) = \Omega(n^{\log_b a + \varepsilon})$ AND $af(n/b) < cf(n)$ for large n

$\varepsilon > 0$
ε is very small

Date / /
Page No.
Shivalal

Case-1 If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$ then it follows that:
$T(n) = \theta(n^{\log_b a})$

Ex $T(n) = 8T(\frac{n}{2}) + 1000n^2$ apply master theorem on it.

Compare $T(n) = 8T(\frac{n}{2}) + 1000n^2$ with

$T(n) = aT(\frac{n}{b}) + f(n)$ with $a \geq 1$ and $b \geq 1$

$a = 8$, $b = 2$, $f(n) = 1000n^2$, $\log_b a = \log_2 8 = 3$

Put all values in $f(n) = O(n^{\log_b a - \varepsilon})$

$1000n^2 = O(n^{3-\varepsilon})$ if we choose $\varepsilon = 1$

$f(n) = 1000n^2 = O(n^2)$

Then, $T(n) = \theta(n^3)$ bcoz $\log_b a = 3$ from above.

Case-2 $T(n) = 2T(\frac{n}{2}) + 10n$

Compare with $T(n) = aT(\frac{n}{b}) + f(n)$ with $a \geq 1$, $b > 1$

$a = 2, b = 2$, $k = 0$, $f(n) = 10n$, $\log_b a = \log_2 2 = 1$

Put all values in

$f(n) = \theta(n^{\log_b a} \log^k n)$, we will get

$10n = \theta(n^1) = \theta(n)$ which is true

Therefore $T(n) = \theta(n^{\log_b a} \log^{k+1} n)$

$= \theta(n \log n)$

if $f(n) = \theta(n^{\log_b a} \log^k n)$

⇓

it follows

$T(n) = \theta(n^{\log_b a} \log^{k+1} n)$

d = number of ~~element~~ digits in the largest element (maxim).

Date
Page No.
Shivalal

Radix Sort (A, d)

① // Each key in A[1 - - - n] is a d-digit integer.

② // Digits are numbered 1 to d from Right to Left. d = max number of digits that the ~~largest~~ biggest element contain.

③ for i = 1 to d do. ← least significant digit from condition ②

④ use a stable sort Algo to sort A on digit 'i'

[3 2 1]

Here d = 3

Ex

4 2 0		4 2 0
5		0 0 5
2 6	⟹	0 2 6
9 0		0 9 0
2		0 0 2
2 2 3		2 2 3

⟹ After making 3 digit integer should be there like 420

Always sorting Algo will be performed on '1' digit..

1st iteration on '1' digit

4 2 0		4 2 0
0 0 5	digit 1	0 9 0
0 2 6	⟹	0 0 2
0 9 0		2 2 3
0 0 2		0 0 5
2 2 3		0 2 6

1 digit no has been sorted

one digit no has been sorted

relative position

digit 2 ⟹

0 0 2
0 0 5
4 2 0
2 2 3
0 2 6
0 9 0

Two digit no has been sorted

Application → 1- It is used for storing string.
2- Sorting faster if runs on parallel machine.
3- is used for construction of suffix array.

digit 3 →

0 0 2
0 0 5
0 2 6
0 9 0
2 2 3
4 2 0

finally sort Array → 2, 5, 26, 90, 223, 420

Radix Sort → In this sort, we sort the elements by processing them in multiple passes, digit by digit. Each pass sorts elements according to the digits in a particular place value, using a stable sorting algo. We can start sorting from least significant digit (LSD). Radix sort uses counting sort as an intermediate sorting Algorithm.

Suffix Array → An array that contains "all the possible suffixes" of a string in sorted order.

String → "sort"

Suffix Array SA[] will be: SA[0] = "ort"
SA[1] = "rt"
SA[2] = "sort"
SA[3] = "t"

Radix, Bucket, Counting
Non comparision-based

Date / /
Page No.
Shivalal

Bucket sort (Non-comparision based Algo)

1- Works on floating-point numbers b/w range 0.0 to 1.0.
2- Inputs should be uniformly and independently distributed across [0,1] to get a running time of O(n).

Best case & Average case → O(n)

Algorithm → BUCKET-SORT(A)

1- let B[0 ... n-1] be a new array.
2- n = A.length
3- for i = 0 to n-1
4- make B[i] an empty list
→ 5- for i = 1 to n
6- insert A[i] into list B[⌊nA[i]⌋]
7- for i = 0 to n-1
8- sort list B[i] with insertion sort
9- concatenate the lists together
B[0], B[1] -- B[n-1] (in order)

Advantage:
1- it reduces the no of comparision
2- it is fast bcoz uniformly distributed

Disadvantage:
1- it is not useful if large Array increase cost.

first number after dot(.)

Ex 0.79, 0.13, 0.64, 0.39, 0.20, 0.89, 0.53, 0.42, 0.06, 0.94

1 2 3 4 5 6 7 8 9 10

In this 0 & 1 range

0.79×10 ← floor value

$\lfloor 7.9 \rfloor = 7$

$.13 \times 10$

$\lfloor 1.3 \rfloor = 1$

sort properly

→ uniformaly in one bucket

So this is best case or Avarage case (step 8 will not apply)

for insertion of each element → O(1)

n number of time so

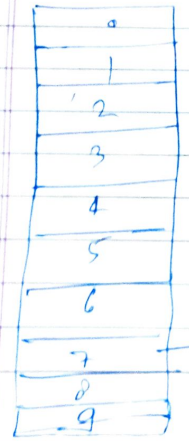
complexity will be → O(n)

So, no sorting is needed in I case

→ Worst case → if it is not uniformaly distributed

if the element like this:

0.79, 0.78, 0.795, 0.74



$O(n) \times n$

Complexity → $O(n^2)$

Here we will apply (8) step according to Algo.

.78 .79

.79 .78 .79

Insertion sort

.74 .76 .74 .74

Application → sorting a list of numbers or string in ascending or descending order.
Bubble, insertion & quick sort.

Counting sort :- It is a non comparision based algorithm.

1- Given I/P size is 'n'
2- Given Range is 'K'

2 1 2 3 1 2 4 → keys.
→ Range = 5

1	2
2	3
3	1
4	1
5	0

Size = K+1 = 4+1 = 5
Array = 5

⇒ Traverse

1 1 2 2 2 3 4 (After sorting)

O(n+k) → Time complexity.
input size, Range

O(k) → Space complexity

Range is already defined in counting sort.
it is working in linear time.

Range is constraint in counting sort.

1 0 2 1 0 1 1 5 6 7 5 4 2 2 0 0 1
n = 17
max

Range → 0 - 7
K = 7

Counting sort is a sorting technique which is based on the range of I/P value.

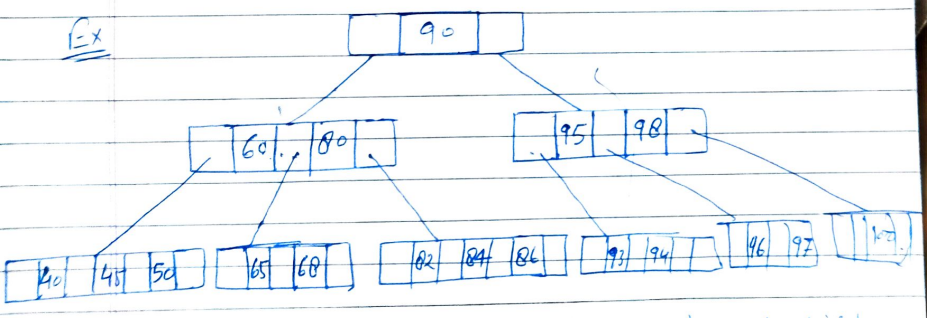
B-Tree

B-Tree is a special tree that is used for disk access. It is also called m-way tree. If the order of B-Tree is 'm' then it will contain at most 'm-1' keys & m children. The main reason of using B-tree is to store large number of keys in a single node and large key values by keeping the height of B tree small.

Properties →

1- Every node in a B-tree contains at most 'm' children
2- Every node except the root & leaf node contain at least 'm/2' children
3- The root nodes must have at least 2 nodes
4- All leaf nodes must be at the same level

Ex



Degree → The degree of a node is the number of children it has. So every node of a B tree has a degree greater than or equal to 'zero' and less than or equal to the order of the B-Tree.

Date / /
Page No.
Shivalal

Degree → The number of subtree of a node is called its degree.

Order	Degree
Represents the upper bound on the number of children i.e max number possible.	represents the lower bound on the number of children except for root. i.e min number of children possible.

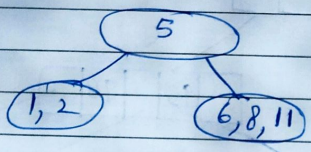
Construct B-Tree using following elements having degree $t = 2$:

1, 5, 6, 3, 8, 11, 13, 18, 20, 7, 9

formula → Max number of element/key a node can have $= 2t - 1 = 2 \times 2 - 1 = 3$ (max number of key)

Insert 1, 5, 6 — (1, 5, 6) — max keys are "3".

Insert 2, 8, 11 this node will be split from middle bcoz we can not insert 2 bcoz it will be 4 keys.

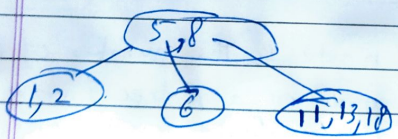


key	$\lceil m/2 \rceil - 1 =$	$m-1 = 4-1 = 3$
ptr	$\lceil m/2 \rceil = 2$	$m = 4$

↓ degree = min key

After we can not insert 13. So middle element will go up

I → B



In b-tree you can store both keys & data in the internal as well as leaf nodes, but in b+ tree we can store data in leaf node only.

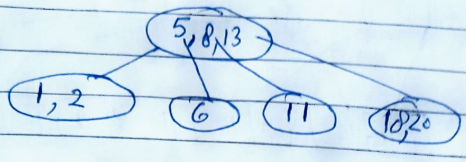
using order → order = max no of child.
max no of keys = 0-1 = 5-1 = 4
order order

This is not in course

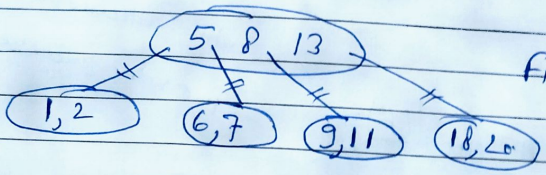
Date
Page No.
Shivalal

After we can not insert 20. So middle element '13' will go up.

I → 20



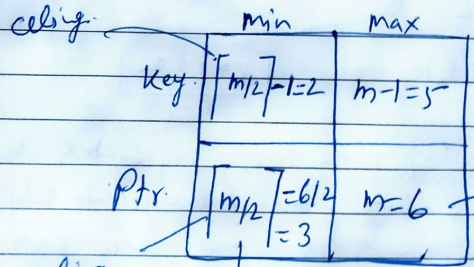
I → 7



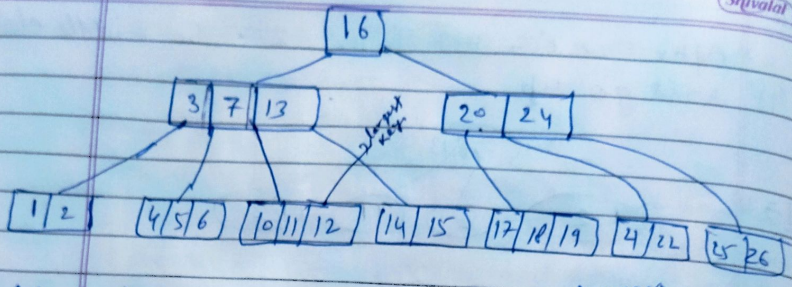
Final B-Tree of degree t = 2

In this tree all the leaf node at the same level. There are '3' keys so number of child nodes are → K+1 = 3+1 = 4. And all of them stored in increasing order.

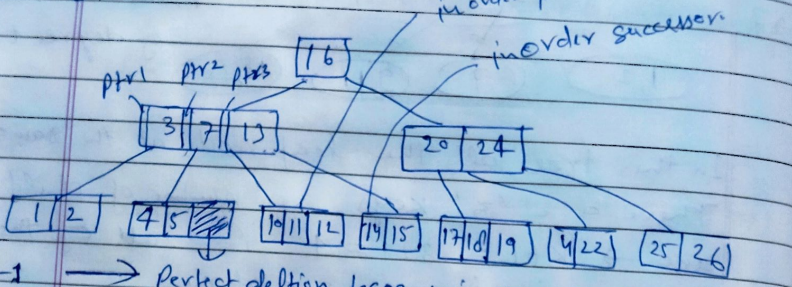
B-Tree (Deletion using Degree)



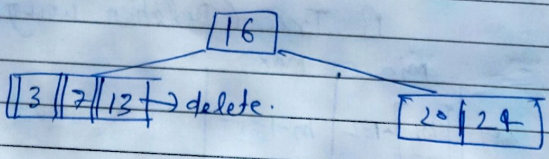
Q.1 Delete the following keys: 6, 13, 7, 4, 2
Minimum degree for this B-Tree is 3



Delete 6 →



Case-1 → Perfect deletion bcoz min keys can be '2' according to rule.



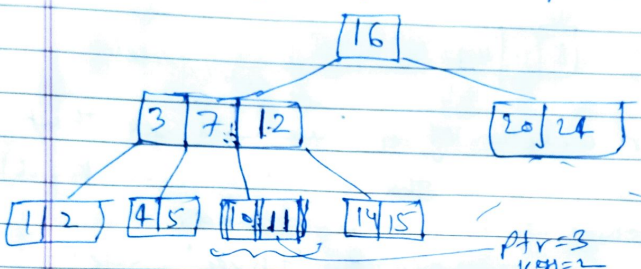
Case-2

In case delete '13', but there is 3 pointers so these pointer can link 3 child, but there is 4 child in starting. After deletion of '13' only three nodes will be associated but in starting it was four node. In this case the predecessor of 13 is '12'. (inorder)

In-order Predecessor → The largest key on the left child of a node is called its inorder predecessor.
⇓ 12

In-order Successor → The smallest key on the right child of a node is called its inorder successor.
⇓ (10)

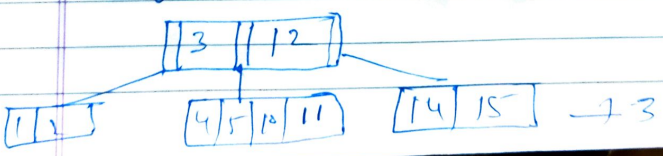
It will go "up side" inplace of 13.



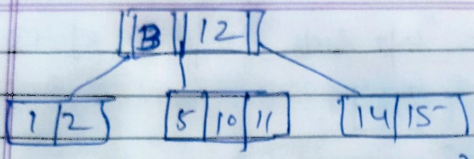
Case-2 After shifting '12' upper side we will check the rules, if the rules is not destroyed than it is correct. In case of True we check ptr values and min number of keys that in ptr=3, key=2 that is correct.

Next if we delete '7' then check inorder predecessor of '7' is '5' but we can not shift '5' up side bcoz no of keys will become '1' that destroy the rule.

Case-3 Suppose we delete '7' then there are '3' pointer will remain but this pointer hold '4' child node. that is not possible. In this case we will merge. This will be correct.



→ 3 node & 3 pointer

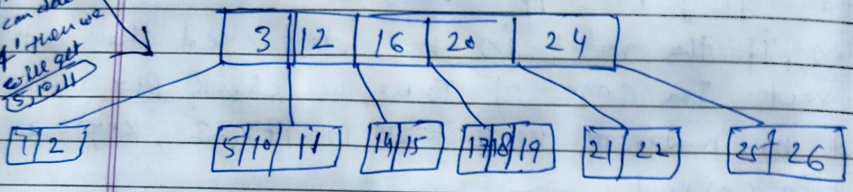


After deletion of '4' → easily delete.
we will see that can we reduce the height of tree. By default searching time will be reduced.

Case-1

After merging we can delete '4' then we will get 5, 10, 11

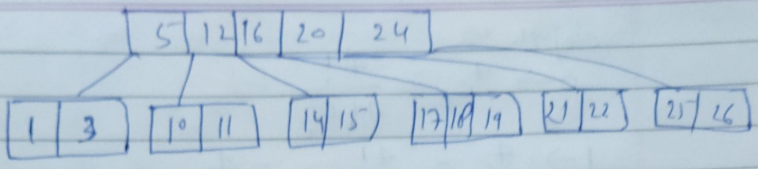
After merging all the 'three' nodes.



delete 2 → we can not delete '2' directly so we can borrow '3' but there will be problem. After shifting '3' below inplace of '2'. In this case only '4' keys will be remain upper side and '6' nodes will be there below so we can shift '5' upper side of node = 5, key = 4 but there are '6' nodes so we shift '5' upper side.

Case-2

Date / /
Page No.
Shivalal



Binomial Heap

The main application of Binary heap is as implement a priority queue. Binomial Heap is an extension of Binary Heaps, that provides faster union & merge operation with other operation ~~Hum~~ Provided by binary heap.
Binomial Heap is a set of Binomial trees, where each binomial tree follows min Heap property.

Properties →

1- B_K tree has 2^K node.
2- Degree of root is K.
3- Height = K
4- There are exactly ^KC_i node at depth i

$$^KC_i = \frac{K!}{(k-i)!\,i!}$$

Time complexity of insertion & Deletion = $O(\log n)$