Vinod Kumar Rajput 34/750

Monday

Page No.
Date

* Algorithm

It is a finite step by step procedure to solve a problem.
It is a well defined computational procedure that takes some values as input and produ-ces some values as output.

* Design of Algorithm

→ Techniques

(i) Dynamic Programming Algorithm — substructure — Table — Bottom-up computation
(ii) Greedy Algorithm
(iii) Randomized Algorithm
(iv) Branch & Bound
(v) Backtracking
(vi) Divide and conquer

* Characteristics of Algorithm

1. Correctness
2. Simplicity
3. Non-Ambiguity
4. Generalization
5. Efficiency

* Analysis of an algorithm

Analysis of algorithm is required to check the correctness and measure the quantitative efficiency of an algorithm.

* There are three cases for analysis of algorithm

1. Best case - min. no. of steps in an instance.
2. Worst case - max. no. of steps in an inst.
3. Average case - average no. of steps

Best → atleast Worst → atmost
average - average

1. Worst case - The worst case efficiency of an algorithm is its efficiency for the worst case input of size n which is an input (or inputs) of size n for which an algorithm runs the longest among all possible inputs of that size.

2. Best case - The best case efficiency of an algorithm is its efficiency for the best case input of size n which is an input (or inputs) of size m for which an algorithm runs the fastest among all possible inputs of that size.

Average case time - complexity is the function defined by the average no. of steps for any instance of size.

* Asymptotic Notations

It is used to measure the time and space complexity.

* Types of Asymptotic Notations

1. Big Theta
2. Big oh
3. Big Omega
4. Small oh
5. Small Ω

1. Big Theta

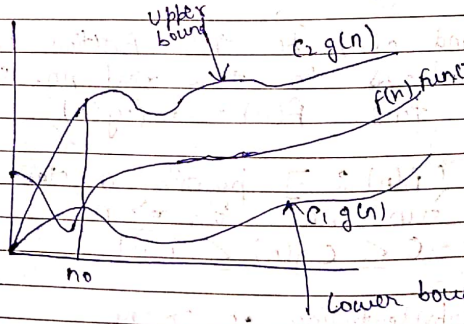
Let $f(n)$ and $g(n)$ be the function that maps +ve integers to the +ve real no.

$f(n) = \Theta(g(n))$

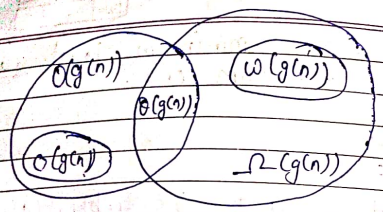
$\Theta(g(n))$ as the set

$\Theta(g(n)) = \{ f(n) \}$ ∃ positive constants c_1, c_2 and n_0 such that $\forall n \geq n_0$, we have

$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$



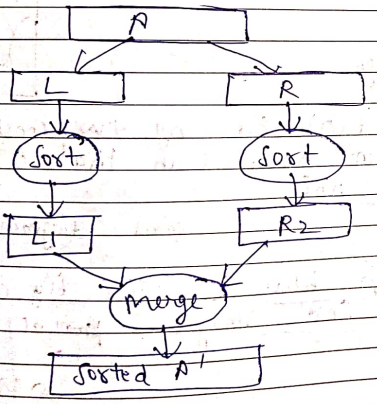
$f(n) = \Theta(g(n))$



* Merge sort

Merge sort is basically divide and conquer paradigm.

(i) Divide step
(ii) Conquer step
(iii) Combine step



MERGE(A, p, q, r)
1. $n_1 \leftarrow q - p + 1$
2. $n_2 \leftarrow r - q$
3. Create arrays L[1, ... n_1+1] and R[1 ... n_2+1]

4. for $i \leftarrow 1$ to n_1
5. do $L[i] \leftarrow A[p+i-1]$
6. for $j \leftarrow 1$ to n_2
7. do $R[j] \leftarrow A[q+j]$
8. $L[n_1+1] \leftarrow \infty$
9. $R[n_2+1] \leftarrow \infty$
10. $i \leftarrow 1$ $j \leftarrow 1$
    for $k \leftarrow p$ to r.
11. do if $L[i] < R[j]$
12. then $A[k] \leftarrow L[i]$
13. $i \leftarrow i+1$
14. else $A[k] \leftarrow R[j]$
15. $j \leftarrow j+1$

MERGESORT(A, p, r)
if (p < r)
then $q \rightarrow \lfloor (p+r)/2 \rfloor$
MERGESORT(A, p, q)
MERGESORT(A, q+1, r)
MEGRE(A, p, q, r)

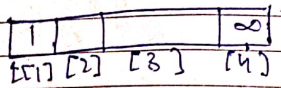
p 1, 6, 5, 4 q | q+1 8, 9, 12, 13 r

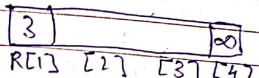
* Illustrate the operation of MERGE SORT on the array A = {1, 2, 4, 3, 5, 7}
p = 1, r = 6, $q = \lfloor (1+6)/2 \rfloor = 3.5 = 3$
$n_1 = 3$, $n_2 = 3$
L[1 ... n_1+1] = L[1 ... 3+1] = L[1 ... 4]
R[1 ... n_2+1] = R[1 ... 4]
4. for $i \leftarrow 1$ to n_1 $i \leftarrow 1$ to 3
   L[1] ← A[1+1-1]
   L[2] ← A[2]
   L[3] ← A[3]

j ← 1 to 2 1 to 3

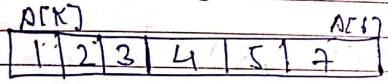
R[1] ← A[4]
R[2] ← A[5]
R[3] ← A[6]

$n_1 + 1 \rightarrow \infty$
$n_2 + 1 \rightarrow \infty$





for k ← p to r k ← 1 to 6
do if (L[i] < R[j])
L[1] < R[1]
1 < 3



Analyzing Merge sort

Divide step (D)
$q = (p+q)/2$

The run time
$D(n) = \theta(1)$
In this step run time is always constant.

Conquer step
In this array are divided into n/2
So total run time $= 2T(n/2)$

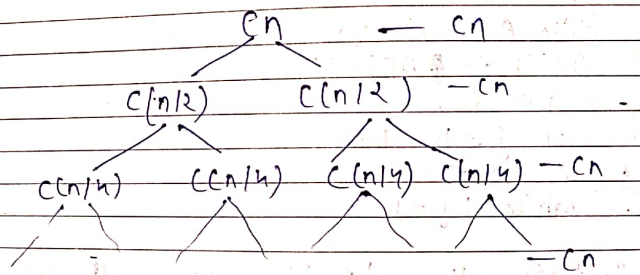
3. Combine step
Run time $C(n) = \theta(n)$

Complete Recurrence Variation Time →

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 2T(n/2) + \theta(n) & \text{if } n \geq 1 \end{cases}$$

For standard purposes

$$T(n) = \begin{cases} & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n \geq 1 \end{cases}$$



$T(n) = cn(\log n + 1)$
$= cn \log n + cn$
$T(n) = \theta(n \log n)$

* Shell SORT

```
Shell_Sort(A,n)
1.  h ← 1
2.  while h ≤ n
3.  h ← 3*h + 1
4.  do
5.  h ← (h-1)/3
6.  for i → h+1 to N
7.  V ← A[i]
8.  j ← i
9.  while A[j-h] > V
10. A[j] ← A[j-h]
11.   j ← j-h
12. if j ≤ h then
13. A[j] ← V
14. while h != 1
```

eg Illustrate the operation of shell sort on the following array

A[] = | A | S | O | R | T | I | N | G | E | X | A | M | P | L | E |

n = 15

Initially $h = 1$

$h \leftarrow 3 \times h + 1$

$h = 4$

$h \leq n$

$4 \leq 15$

$h \leftarrow 3*4+1$

$h = 13$

$13 \leq 15$

$h = 40$

condition false

Steps $h \leftarrow (h-1)/3$

$h \leftarrow (40-1)/3$

$h = 12$

again $h \leftarrow (13-1)/3$

$h = 4$

again $h = 1$

$(h \leq n)$

$h = 1, (4), 13, 40$

1 (5)

| A | S | O | R | T | I | N | G | E | X | A | M | P | L | E |

A, L are in order so we don't change

S, E are not in order so we interchange

now we start increament of 4

1st element is compared with 5th & similarly.

* Binary Search

```
Binary_Search(A, k)
1.  low ← 1
2.  high ← n
3.  while low ≤ high
4.  do m ← [(low + high)/2]
5.     IF k = A[m]
6.     return m
7.     else if k ≤ A[m]
8.        high = m-1
9.     else
10.       low ← m+1
11.    return -1
```

* Applying the binary searching for searching k = 70 in the following array

3, 4, 27, 37, 39, 43, 53, 70, 73, 79, 83, 90, 95

low ← 1 high ← 13

while low ≤ high 1 ≤ 13

m = (1+13)/2 = 14/2 = 7

If k = A[m] 70 = 58

else if k ≤ A[m] 70 ≤ 58

else

low ← m+1 low ← 7+1 = 8

* STRASSEN'S Algorithm for Matrix Multiplication

(i) Express each matrix A, B and C as a 2×2 block matrix of n/2 × n/2 matrices and So find a, b, c, d, e, f, g, h

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad B = \begin{bmatrix} e & f \\ g & h \end{bmatrix} \quad C = \begin{bmatrix} r & s \\ t & u \end{bmatrix}$$

(ii) Compute

$A_1 := a$		$B_1 := f - h$
$A_2 := a + b$		$B_2 := h$
$A_3 := c + d$		$B_3 := e$
$A_4 := d$		$B_4 := g - e$
$A_5 := a + d$		$B_5 := e + h$
$A_6 := b - d$		$B_6 := g + h$
$A_7 := a - c$		$B_7 := e + f$

(iii) Compute the matrix product

$P_i := A_i B_i$ $\quad i := 1, 2, 3, \ldots, 7$

(iv) Then we have

$r := P_4 + P_5 + P_6 - P_2$

$s := P_1 + P_2$

$t := P_3 + P_4$

$u := P_1 + P_5 - P_3 - P_7$

(v) The require maxtrix product = $C = AB$ is :-

$$C = \begin{bmatrix} r & s \\ t & u \end{bmatrix}$$

Q Use the STRASSEN's algorithm to compute the following matrix product $\begin{bmatrix} 1 & 3 \\ 5 & 7 \end{bmatrix} \begin{bmatrix} 8 & 4 \\ 6 & 2 \end{bmatrix}$

$A_1 = 1$
$A_2 = 1+3 = 4$
$A_3 = 5+7 = 12$
$A_4 = 7$
$A_5 = 8$
$A_6 = -4$
$A_7 = -4$

$B_1 = 2$
$B_2 = 2$
$B_3 = 8$
$B_4 = -2$
$B_5 = 10$
$B_6 = 8$
$B_7 = 12$

$P_1 = 2$
$P_2 = 8$
$P_3 = 96$
$P_4 = -14$
$P_5 = 80$
$P_6 = -32$
$P_7 = -48$

or, $-14 + 80 - 32 - 8 = 26$
$s = 2 + 8 = 10$
$t = 96 - 14 = 82$
$u = 2 + 80 - 96 + 48 = 34$

$$C = \begin{bmatrix} 26 & 10 \\ 82 & 34 \end{bmatrix}$$

* Convex Hull problems

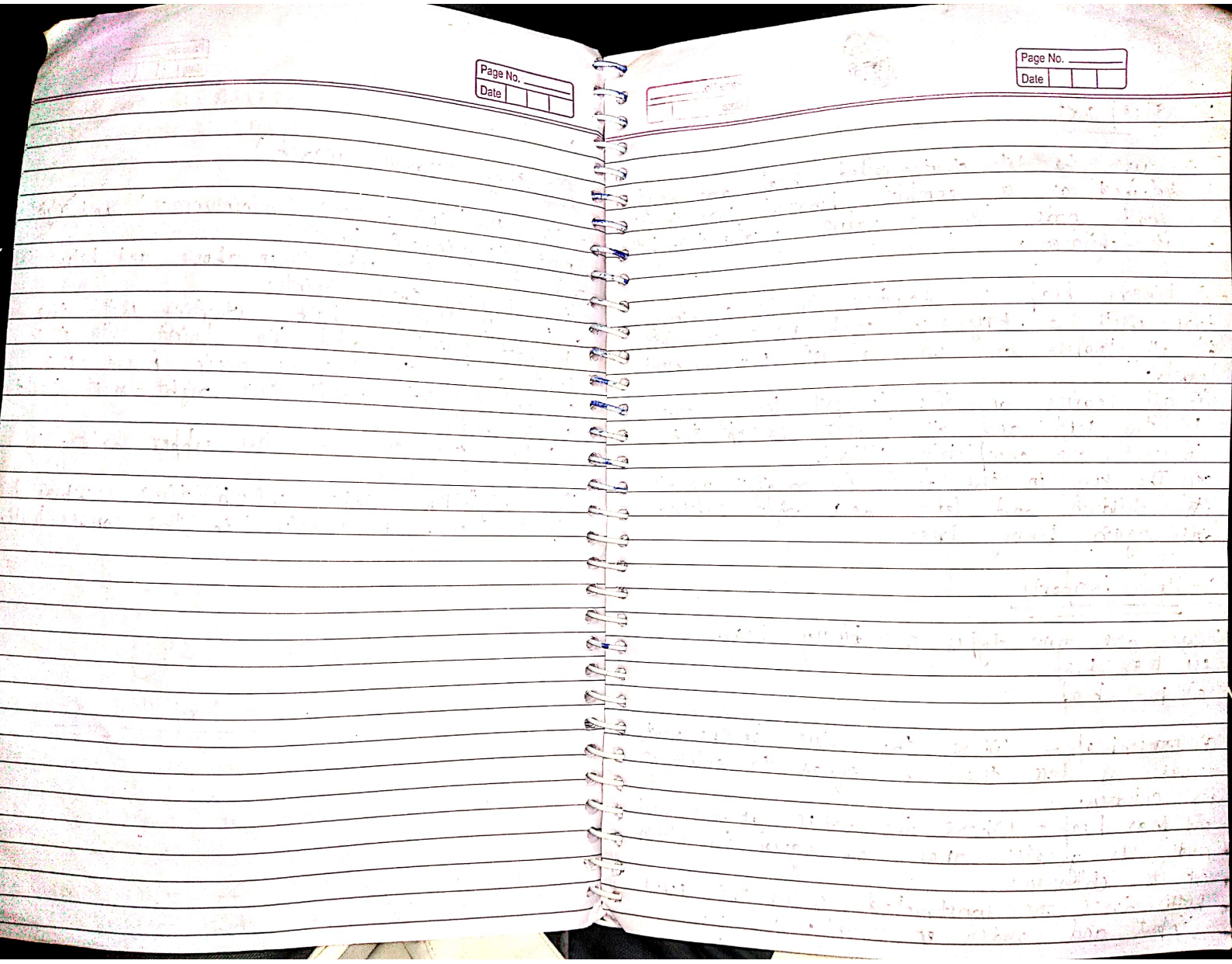
This was given a set of S with n points $P_1 = (x_1, y_1)$ ... $P_n(x_n, y_n)$ and we want to find out the convex hull of S.

The convex Hull of S is the smallest convex polygon that contains all the points in S.

Algorithm (Jarvis march)

- Find the left-most (minimum x) and right most (maximum x) points.
- Divide points into those above and below the line joining these points.
- In the bottom half starting with the leftmost point, add the point with the least angle to the -y axis from the current point until the right-most point is reached.
- Repeat the scan in the upper half.

This algorithm takes $O(nh)$ time, where h is the no. of vertices in the convex hull.



Heapsort

It uses the data structure called heap and heap is defined as a complete binary tree in which each node has a value greater than both of its children.

- A binary heap is defined to be a binary tree with a key in each node such that
(i) all leaves are on, atmost two adjacent levels.
(ii) all leaves on the lowest level occupy to the left and all levels except the lowest are completely filled.
(iii) The key in the root is greatest of all its children and left and right subtree are again binary tree.

Heap property

There are two types of binary heap
(i) Max heap
(ii) Min heap

(i) minheap – Where the value of the root node is less than or equal to either of its children.
(ii) Max heap – Where the value of the root node is greater than or equal to either of its children.
Both trees are constructed using the same input and order of arrival.

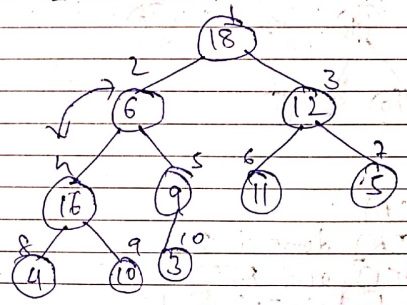
Maintaining the heap property

For max heap

```
Max_heapify(A, i)
l ← left(i)
r ← right(i)
if l ≤ heapsize(A) and A[l] > A[i]
then largest ← l
else largest ← i
if r ≤ heapsize(A) and A[r] > A[largest]
then largest ← r
if largest ≠ i
then exchange A[i] <-> A[largest]
Max_heapify(A, largest)
```

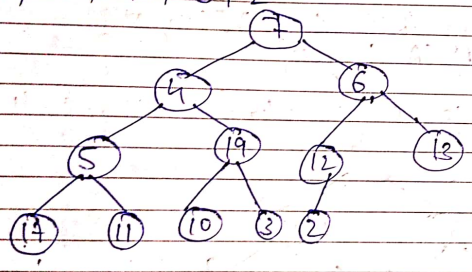
<-> means with

Q A[10] = 18, 6, 12, 16, 9, 11, 5, 4, 10, 3



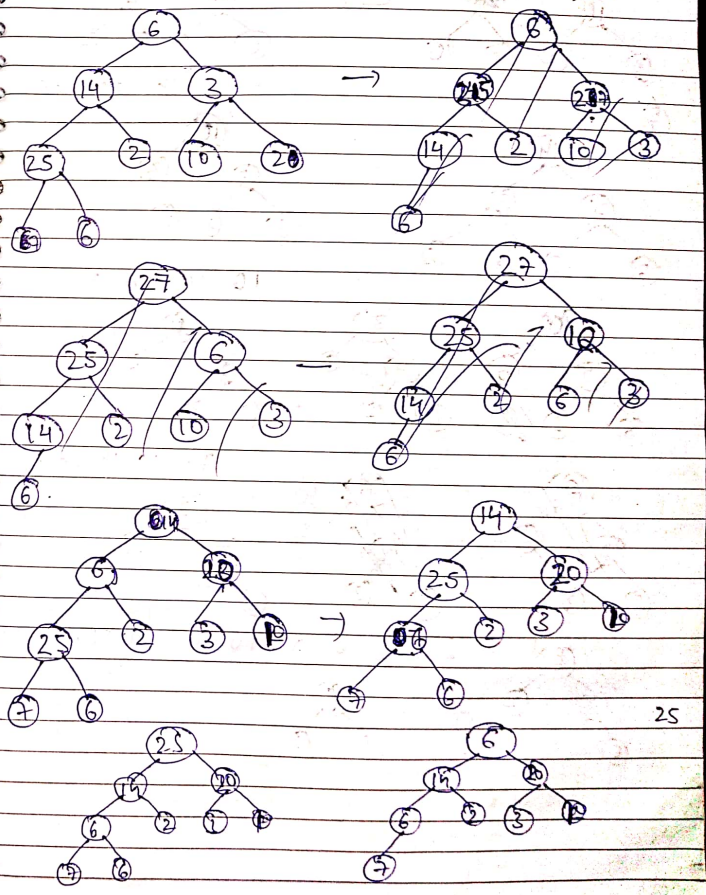
- Build max heap(A)
heap size(A) ← length(A)
for i ← (length[A]/2) down to 1
do max_heapify(A,i)

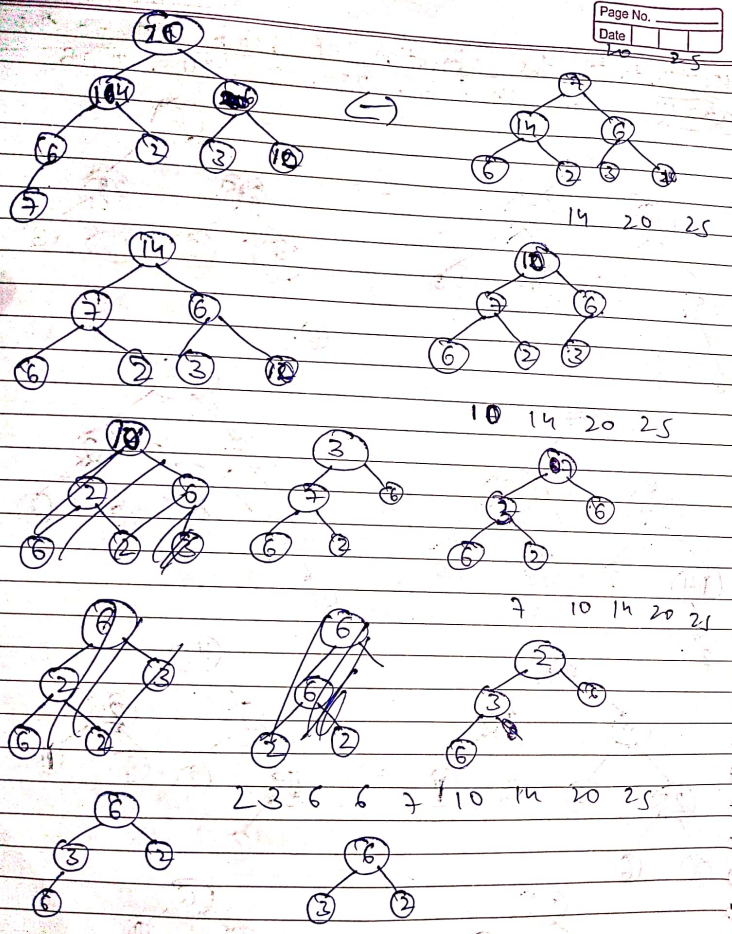
In a straight term operation of build max heap on the array A= 7, 4, 6, 5, 19, 12, 13, 17, 11, 10, 3, 2



Heapsort(A)
Build_max_heap(A)
for i ← length[A] down to 2
do exchange A[1] and A[i]
heapsize[A] ← heapsize[A] - 1
max_heapify(A,1)

Illustrate the operation of heap sort on the array 6, 14, 3, 25, 2, 10, 20, 7, 6





14 20 25

10 14 20 25

7 10 14 20 25

2 3 6 6 7 10 14 20 25

* Quick sort

```
QUICKSORT(A, p, r)
if p<r
then q ← Partition(A, p, r)
QUICKSORT(A, p, q-1)
QUICKSORT(A, q+1, r)

Partition(A, p, r)
  x ← A[r]
  i ← p-1
for j ← p to r-1
do if A[j] ≤ x
  then i ← i+1
    exchange A[i] ⇔ A[j]  (loop end)
    exchange A[i+1] ⇔ A[r]
    return i+1
```

Illustrate the operation of partition on the array A =

1	2	3	4	5	6	7	8
9	2	7	5	1	4	10	6

$n = 6$ $\quad i \leftarrow 1-1 = 0$

$j \leftarrow 1$ to 7

$9 \leq 6$ $\quad 2 \leq 6$ $\quad i = 0+1 = 1$

$9 \leftrightarrow 2$ $\quad i = 1+1 = 2$

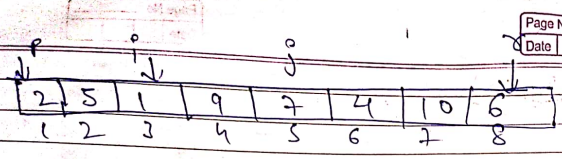
$i = 2+1 = 3$

$j = 5$

2	9	7	5	1	4	10	6

$5 \leq 6$

2	5	7	9	1	4	10	6



* Randomized Quicksort Algorithm

```
Randomized Partition (A, p, r)
    i ← Random(p, r)
    return Partition (A, p, r)

Randomized Quicksort (A, p, r)
    if p < r
    then q ← Randomized_Partition(A, p, r)
         Randomized Quicksort(A, p, q-1)
         Randomized_Quicksort(A, q+1, r)
```

* Sorting in Linear Space (Radix sort, Bucket sort)

Sorting of no by radix sort by sorting the least significant to the most significant digit.

```
Radix_sort(A, d)
for i ← 1 to d
do use a stable sort to sort
array A on digit i
```

326	690
453	751
608	453
825	704
751	825
435	435
704	326
690	608

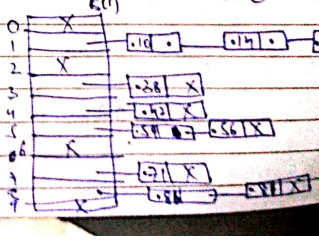
704	326
608	435
326	453
825	608
435	690
751	704
453	751
690	825

* Bucket Sort (Floating Point Nos)

```
Bucket(A)
n ← length(A)
for i ← 1 to n
do insert A[i] into list B[n A[i]]
for i ← 0 to n-1
do sort list B[i] with insertion sort
concatenate the list B[0], B[1] ... B[n-1] together
                                  -er in order.
```

Q Sort the following list using the bucket sort. 0.42, 0.71, 0.10, 0.14, 0.86, 0.38, 0.59, 0.17, 0.81, 0.56.

(4.2), (7.1), (1.0), (1.4), (8.6), (3.8), (5.9), (1.7), (8.1), (5.6)



Unit-2

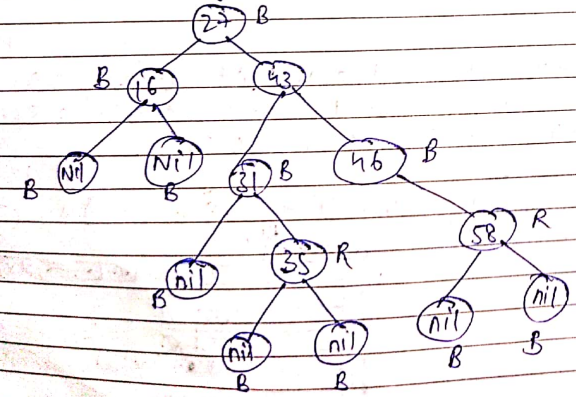
Red Black Tree

It is a binary search tree with extra bit of storage.

Property

Red black tree is a binary search tree if it satisfies the following red black property -

(i) Every node is either red or black.
(ii) The root is black.
(iii) Every leaf is black.
(iv) If a node is red then both its children are black.
(v) For each node every path from the node to the leaf contains the same no. of black nodes.
(vi) all path from the node to the leaf have the same black height



Black height = 3

There is one relationship height and black height of a node and that relation is

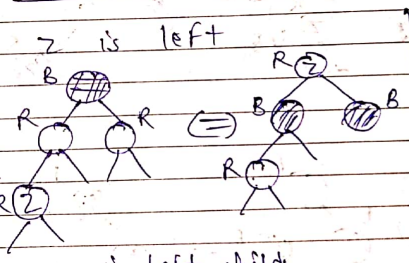
$bh(n) \leq h(n) \leq 2bh(n)$

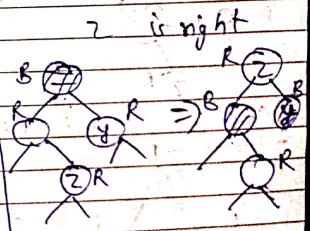
Rotation

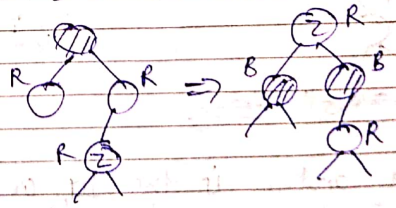
It is a fundamental operation. This operation is used to operate the red black tree

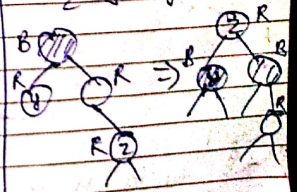
Insertion

Case I :- Insertion into Red black tree

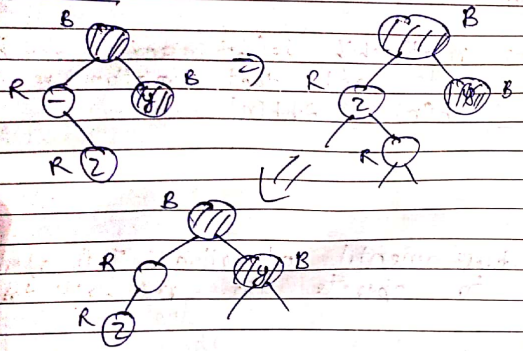




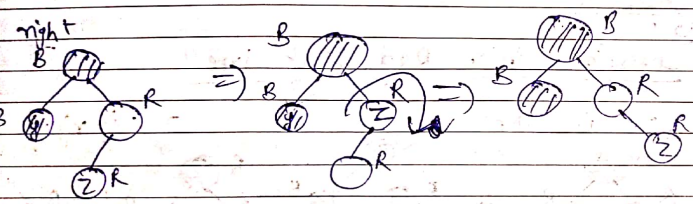




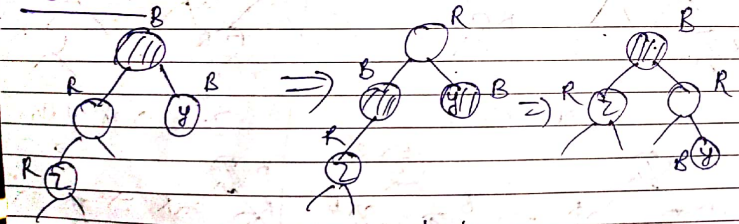
Case 2°: Left



right



Case 3: If is left



Case 1°:
If z's uncle y is red and z is the left or right child make z parent and uncle black. (ii) Make z grandparent red.

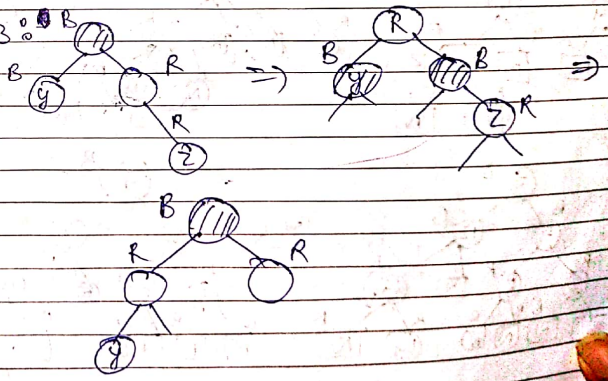
(iii) Set z grandparent as new z.
If the z is on right subtree.
→ Make z parent and uncle y black.

Case 2°:
(i) If z uncle y is black and z is rightchild set z parent as new z then left rotate around new z. Take us to case 3.

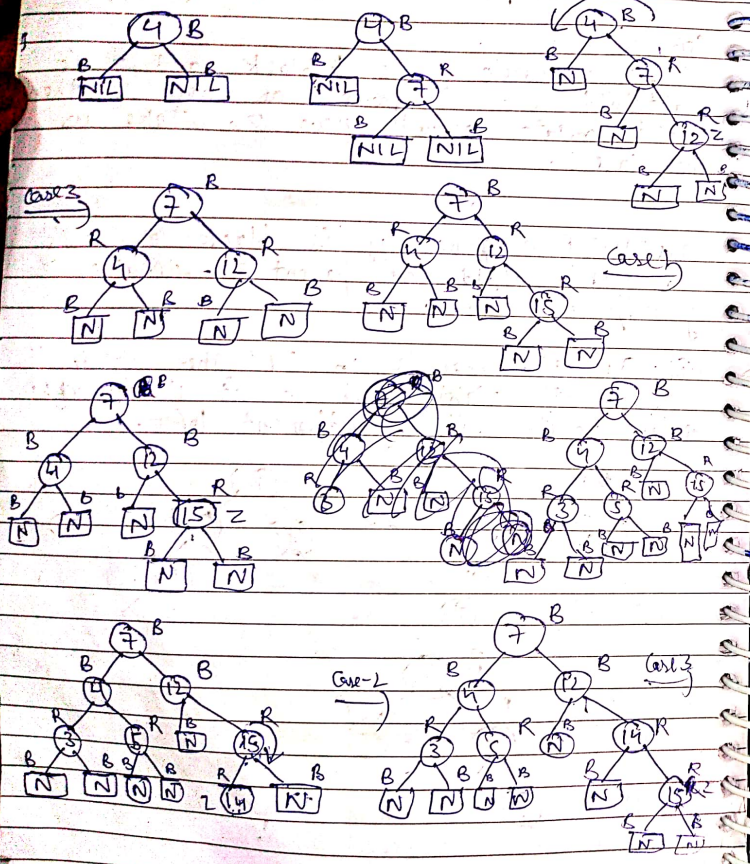
Case 3: (i) If z uncle y is black or z is the left child of parent R. (ii) Make z grandparent red.

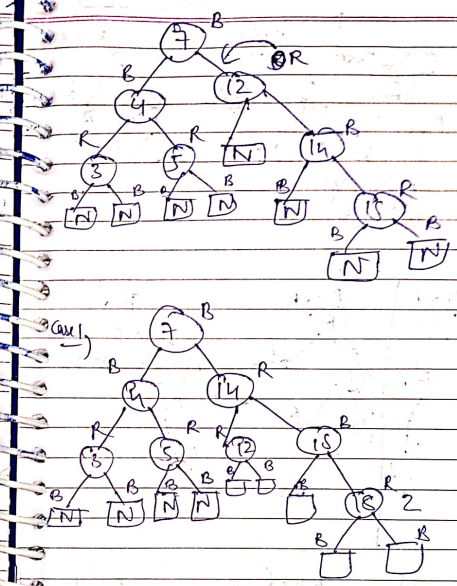
(iii) If child node is in left then rotate it granparent right and if it is in right then rotate its grandparent left.

Case 3°:

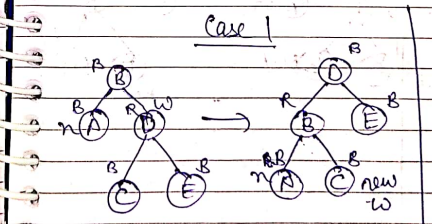


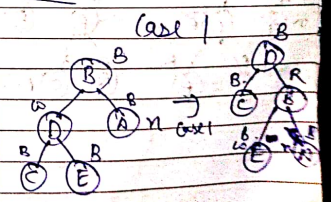
* Insert the following into an initially empty red black tree 4, 7, 12, 15, 3, 5, 14, 18, 16,



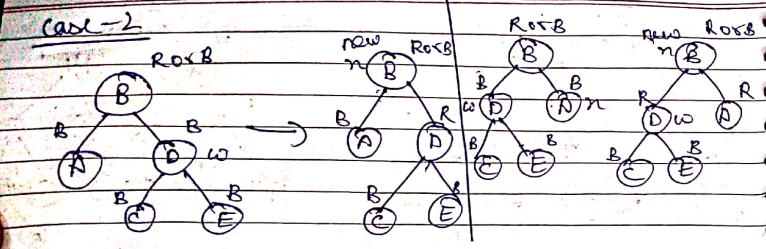


* Red - black tree deletion

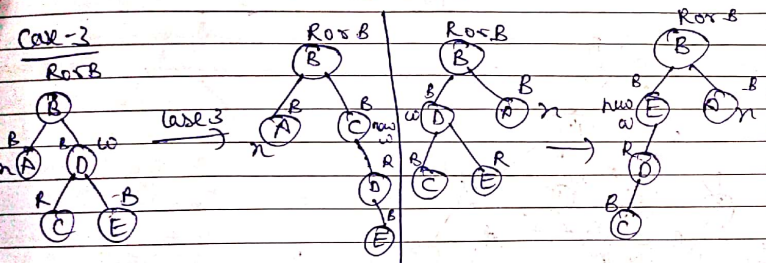




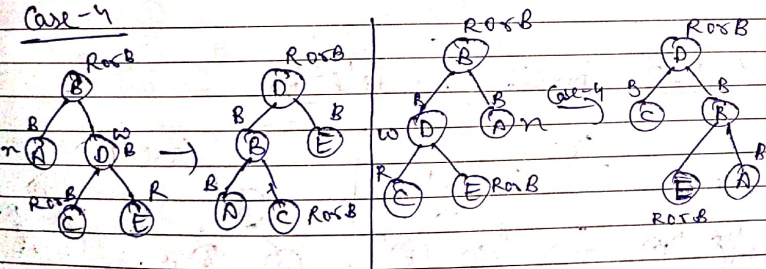
Case-2



Case-3



Case-4



Case 1°: If n's sibling w is red, make w black and

(ii) make n parent red.

(iii) Left rotate around n parent.

(iv) Set the right child of n parent as new w.

n - Let n be a node which take the position of dlleted node. A new node w is taken as a sibling of n

In case of right rotate right.

Case-2°: n's sibling w is black and both of w's children are black

(i) Make w red.

(ii) Set n parent as a new n.

Case-3°: n sibling w is black, w's left child is red and w's right child is black

(i) Make w left child black.

(ii) Make w red.

(iii) Right rotate around w.

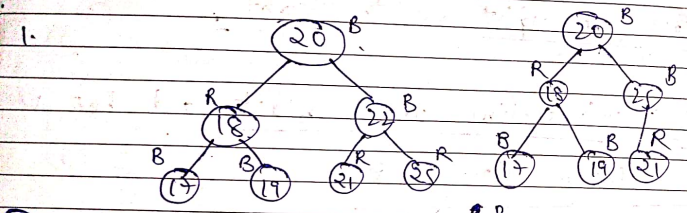
(iv) Set right child of n parent as a new w.

Case-4: n sibling is is black, ii right child is red

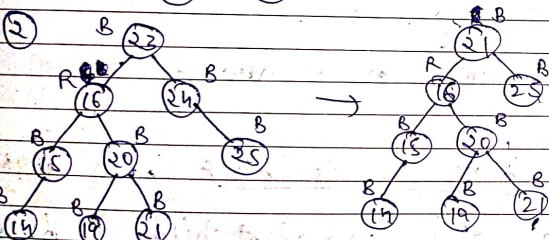
(i) Set n parent black.
(ii) Make n's right child black.
(iii) Left rotate around n parent

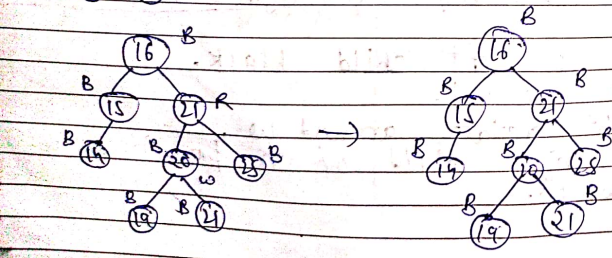
example

1.



2.

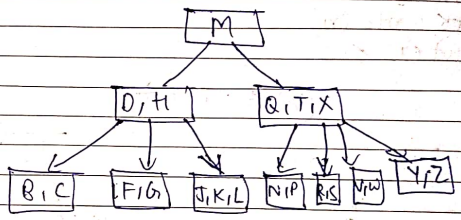




* B-Tree

B-Trees is a balanced tree structure designed to work on magnetic disk or other direct access secondary storage devices. B-Tree has many children.

$n = [n(n) + 1]$ children



* Basic operation on B-Tree

(i) Search

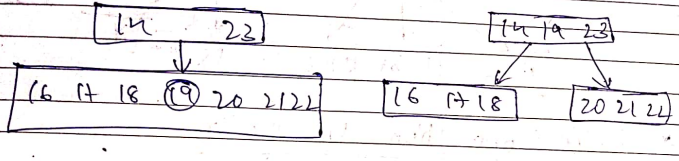
Searching for a key in a B-Tree -

```
B_Tree_search (n, K)
  i ← 1
  while i ≤ n[n] and K > key_i[n]
      do i ← i+1
  if i ≤ n[n] and K = key_i[n]
      then return (n, i)
  if leaf[n]
  then return NIL
  else DISK_READ (C_i[n])
  return B_Tree_search (C_i[n], K)
```

(ii) Creating an empty B-Tree

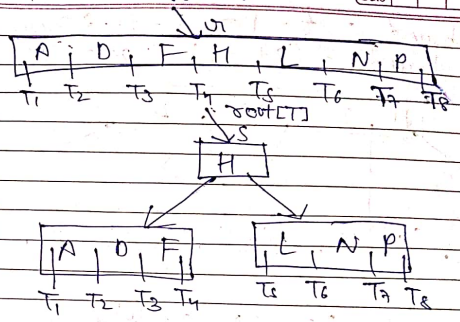
B-Tree-Created (T)
x ← Allocated-Node()
leaf[x] ← True
n[x] ← 0
Disk-write (x)
root(T) ← x

(x) ← Root



(iii) B-Tree-Insert (T, K)
r ← root[T]
if n[r] = 2t - 1
then S ← Allocate-node()
root[T] ← S
leaf[S] ← False
n[S] ← 0
c[S] ← r
B-Tree-Split-child (S, 1, r)
B-Tree-Insert-Nonfull (S, K)
else B-Tree-Insert-Nonfull (r, K)

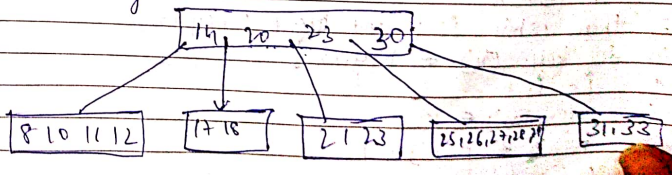
root[T]



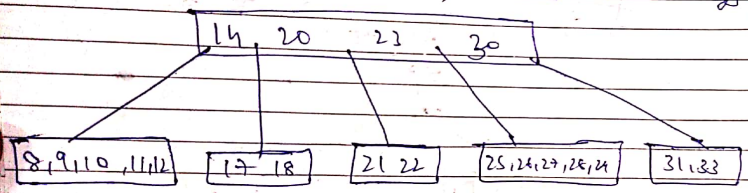
for lower bound
no. of
There are upper bound and lower bound on no. of keys in a node.

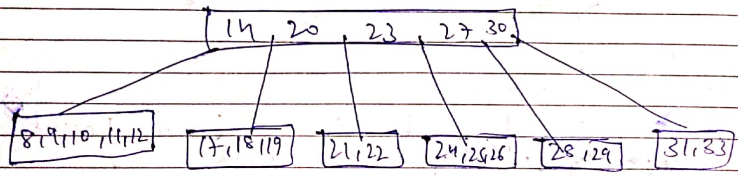
Lower bound (t-1) keys
Atleast t
Upper bound (2t-1) keys
atmost 2t

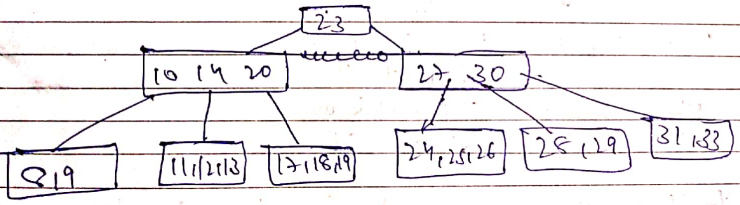
* Show the result of inserting the keys 9, 24, 19, 13, in order in the following order into the following B-Tree with Min degree t = 3.



Lower bound = $t-1 = 3-1 = 2$ keys
Upper bound = $(2t-1) = 2\times3-1 = 5$ keys.







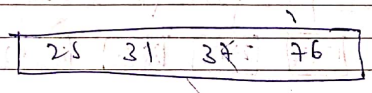
B_Tree_Insert_nonfull (n, K)
1. $i \leftarrow n[n]$
2. if leaf[n]
3. then while $i \geq 1$ and $K < key_i[n]$
4. do $key_{i+1}[n] \leftarrow key_i[n]$
5. $i \leftarrow i-1$
6. $key_{i+1}[n] \leftarrow K$
7. $n[n] \leftarrow n[n]+1$
8. Disk_write(n)

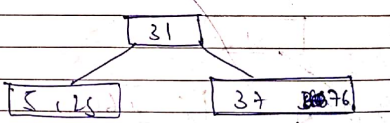
degree = min no. of children
order = max no. of children

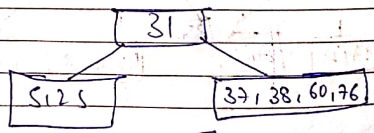
9. else while $i \geq 1$ and $K < key_i[n]$
10. do $i \leftarrow i-1$
11. $i \leftarrow i+1$
12. DISK-READ [$C_i[n]$]
13. if $n[C_i(n)] = 2t-1$
14. then B_Tree_Split_child(n, i, $C_i[n]$)
15. if $K > key_i[n]$
16. then $i \leftarrow i+1$
17. B_Tree_Insert_nonfull ($C_i[n]$, K)

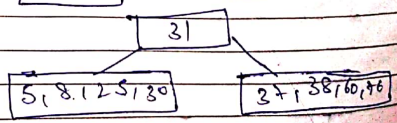
Q Show the result of inserting the following keys in an initially empty B-Tree of order 5: 25, 31, 37, 76, 5, 60, 38, 8, 30, 15, 35, 17, 23, 53, 27, 43, 65, 48.

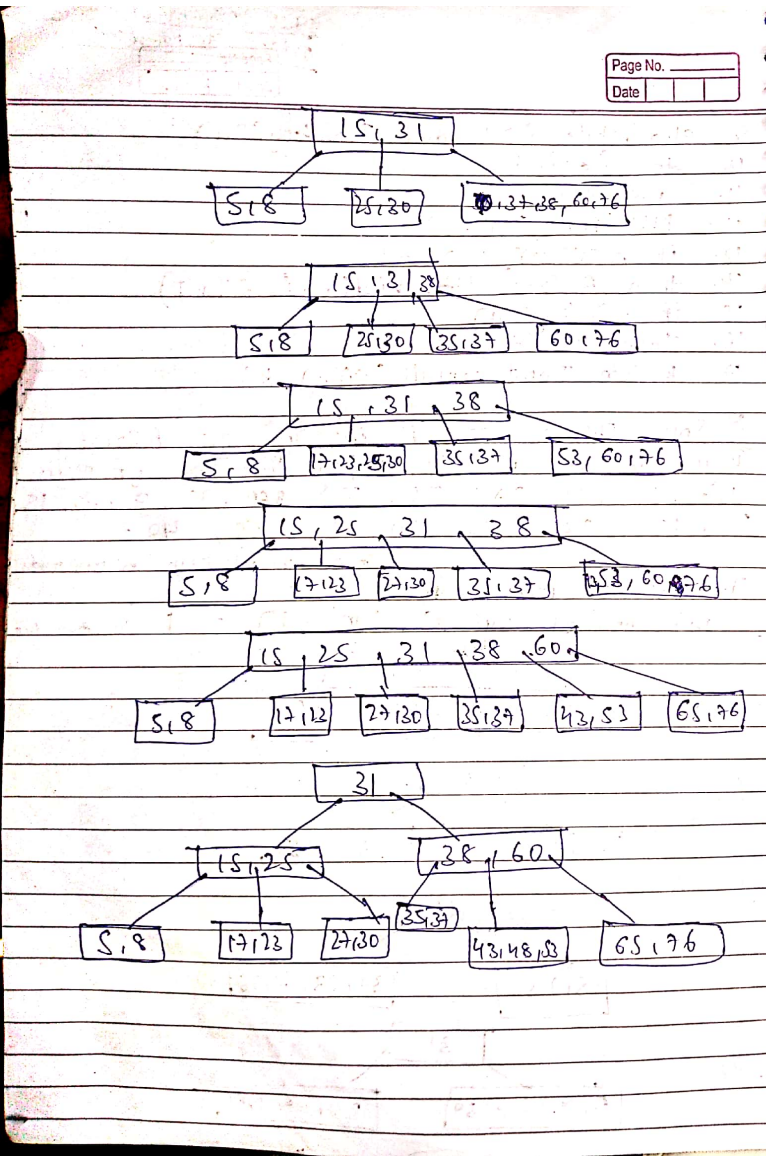
for maximum no. of keys in one node
In case of order, $(m-1) = 4$.
min no. of keys = $\left[\frac{m}{2}-1\right] = 1$ Minimum No



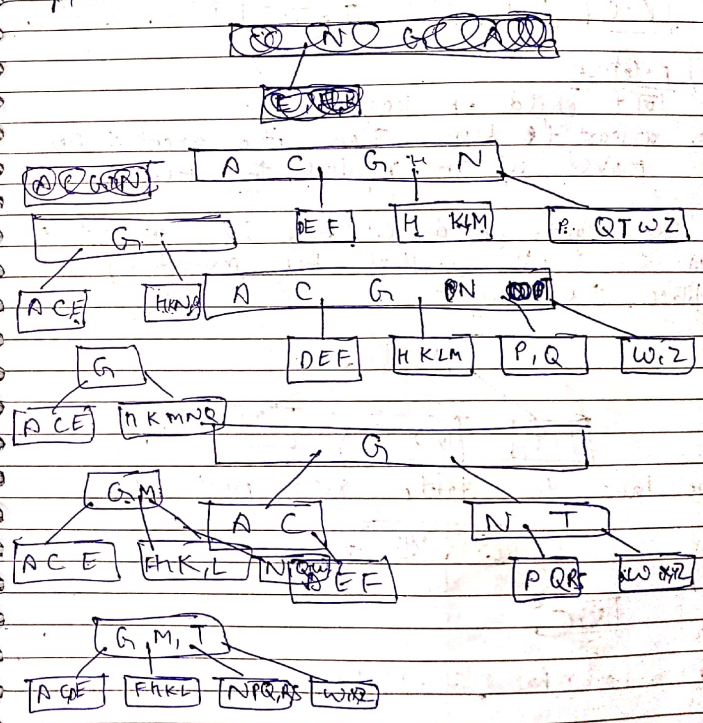




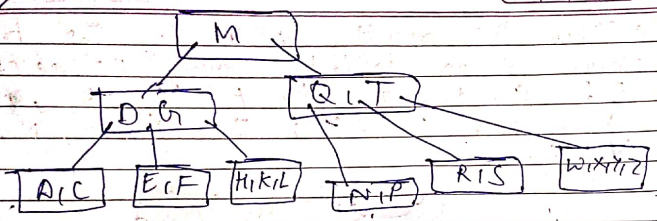




Q Show the result of inserting the following keys in an initially empty B-Tree of order 5: C, N, G, A, H, E, K, Q, M, F, W, L, T, Z, D, P, R, X, Y, S.



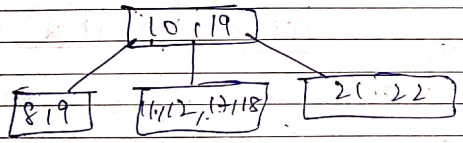
Ans =



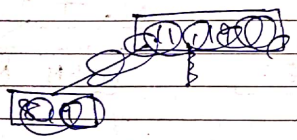
(iv) Deletion

Case 1 If left child of key has atleast t keys Then it's greatest key is moved up to replace the key K.

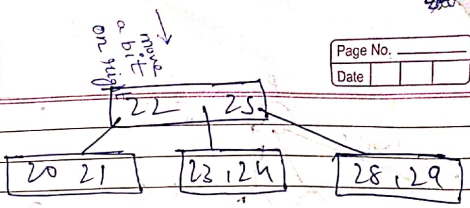
Case 2 If the right child of key has atleast t' key then it's smallest key moved up to replace the key K.



we have to delete root node 10.



Case 3 If a node n has (t-1) keys.



Q Write an algorithm to perform insertion into a B-Tree

* Binomial Heap

Binomial Tree

It is the building block of binomial heap and binomial tree B_K is an ordered tree defined recursively.

B_0 (single) O

B_K B_{K-1} — B_{K-1}

* Properties of Binomial Tree

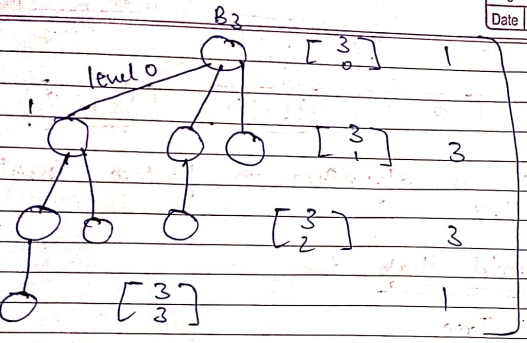
→ There are 2^K node. $n = 2^K$

→ Height of a binomial tree is k. and the the k means degree. Degree means no. of children of root node. $H = \log n$.

→ The no. of nodes at depth i for $i = 0, 1, 2, \ldots$

$$K_i = \binom{K}{i} = \frac{K!}{i!\,(K-i)!}$$

Ans

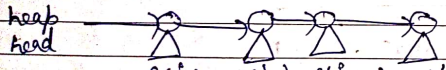


$\binom{3}{0} = \frac{3!}{0!(3-0)!} = \frac{3!}{3!} = 1$

$\binom{3}{1} = \frac{3!}{1!\,2!} = \frac{3 \times 2!}{2!} = 3$

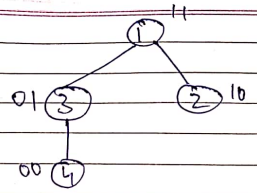
* Binomial Heap

It is represented by H. This is a collection of binomial trees.



$0 < i_1 < i_2 < i_3 < \dots < i_k$

B_0 : 0-bit X
B_1 : One bit
B_2 : Two bit
B_3 : Three bit



* Representing binomial heap