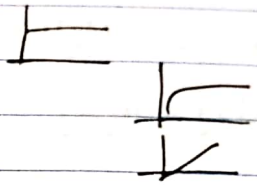
20 Aug

Analysis of Algorithms

Runtime → As input ↑ how rapidly time ↑

Best case, Average case, Worst case



constant	1
log	$O(\log n)$
linear	$O(n)$
N logN	$O(n \log n)$
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Exponential	$O(2^n)$

Recursion

Factorial $f(n) = \begin{cases} 1 & n=0 \\ n \cdot f(n-1) & \text{else} \end{cases}$

Look for Base case
Find recursive call
Its moving towards base case

```
int fun (int &n) {
 if (n == 0)
      return 1;
 else return n x fun (n-1)
}
```

Runtime → No. of times you call recursive function

LinearSum (A, n) Array, size

Base n=1 return A[0]

else return A[n-1] + LinearSum (A, n-1)

rs(A,5)

A[0]+A[1]+A[2]+A[3]+A[4] ↳ return rs(A,4) + A[4]

A[0]+A[1]+A[2]+A[3] ↳ return rs(A,3) + A[3]

A[0]+A[1]+A[2] ↳ return rs(A,2) + A[2]

A[0]+A[1] ↳ return rs(A,1) + A[1]

stack memory is used for recursive functions

ReverseArray (A, i, j)

if i < j swap A[i], A[i]

else reverseArray (A, i+1, j-1)

* No Return Here.

Power (x, n)

if n = 0 return 1

if n>0 return x · power(x, n-1)

$$P(x,n) \rightarrow \begin{cases} 1 & \text{if } n=0 \\ x\ P(x,(n-1)/2) & \end{cases}$$

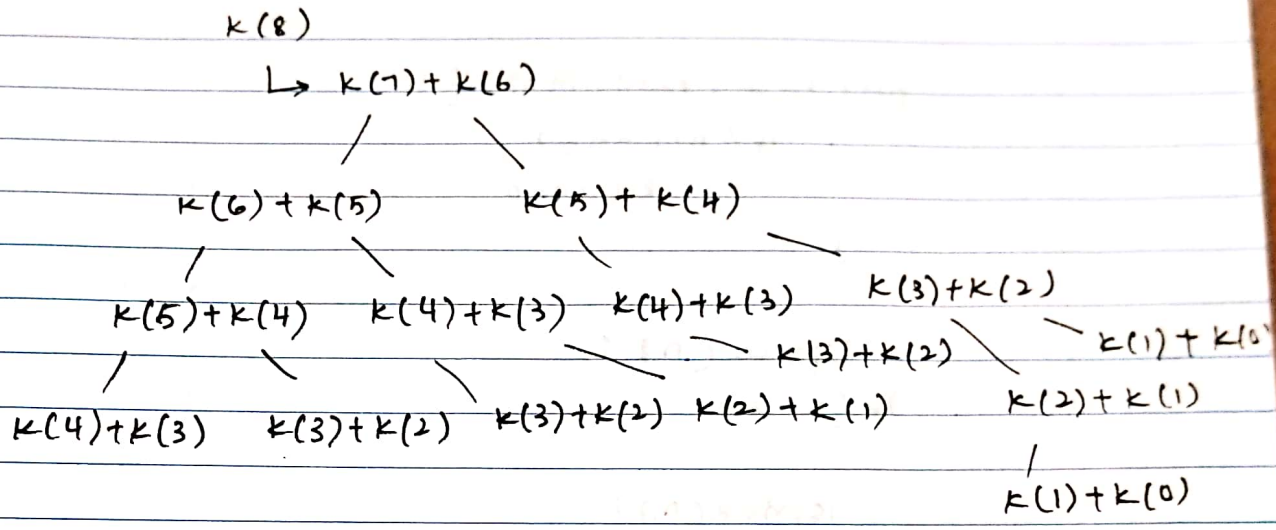
Binary Sum

Fibonacci (K)

K = 0 return 0

K = 1 return 1

else return F(K-1) + F(K-2)



log n → Everytime ↑ half the calls

Exponential → Everytime ↑ double the calls

22 Aug Greatest Common Divisor

$$gcd(x,y) = x \quad y = 0$$

$$= gcd(y, reminder(x,y)) \quad \text{if } y > 0$$

gcd(25,5) → gcd(5,0) → 5

gcd(16,8) → gcd(8,0) → 8

↑ Recursive function keeps going forever
→ It crashes as stacks memory runs out.

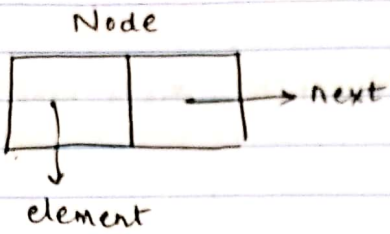
→ Recursive is prime

```
isprime(n)                    n=2,3,5,7,11 prime.

for(i=2; i<n/2; i++){
    if(n/i==0)
        Return 0
}

isprime(n,i)
if(n=2||

isprime(n,p)
if(n==p)
        return true;
if(n%p==0)
        return false;
return isprime(n,i+1)
```

Palindrome

```
palindrome(str, i, j){
if(str[i]==str[j] && i<j)
        return palindrome(str,i+1,j-1)
else    return false
else return true.
```

Linked List



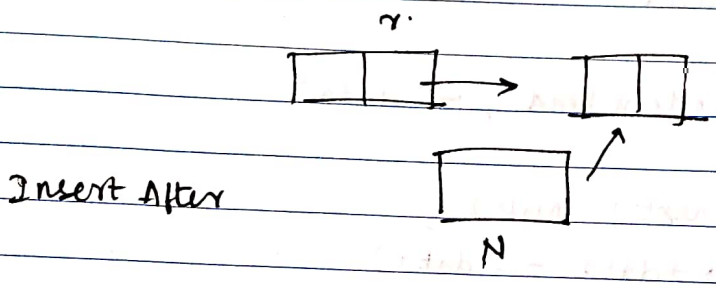
Each node stores element
link to next node

First node → head

```
traverse (node * head , int data)

while ( head → next != null) {
    if (head → data == data)

node {
 int val ;
 node * next;
}

Traverse    while (head != null) {
                print head → value ;
                head = head → next ;
            }

if (head == null)
        return ;
else {  print head.value ;
        head = head → next ; return recursive (head → next) ;
}
```

Insert At Head

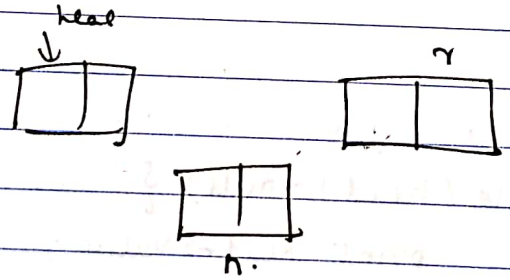
At Head
Insert Before (Node n , Node head)



Insert After

n → Next = r → Next

r → Next = n




```
InsertB (n, r, h)
  if (r == h)
    n→Next = head;
    head = n;
  else (
while ( head→next→data == r→data)
```

27 Aug

```
delete (head, data)
  if (tmp.value == data)    head = tmp.next;
while (temp->next != null && temp->next->data != data)
        temp = temp->next;

tmp->next = tmp->next->next
```

```
Recursiveadd (head)
if (head == null)
        return zero
if (head->next == null)
        return head->data;
else return head->data + Recursiveadd(head->next)
```

largest value in single linkedlist atleast one node

```
max (head)
 if (head->next == null)
        return head->data
else return max (head->data, max(hedd->next));
```

Recursive count no of nodes

```
Count (head)
 if (head == null)
        return 0;
else return 1+Count (head->next);
```

```
if (head→next == value)
        return 1;
else return  nodes (head→next, value, count++)

Nodes ( Head, value)
  if (head == null)
        return 0;     if(head→next == value)
  else return  nodes ( head→next, value

if (tmp == null)  return 0;
if (tmp→next == value)  return 1+ count (tmp→next, val)
else return  count (tmp→next, val);
```

Double linked list :

when no element Head, tail null

when 1 element Head = tail

```
if (n = head)
        head = head → next ;
        if (head == null)   tail = null ;
        else  head.prev = null;
else if (n = tail)
        tail = tail.prev;
        tail.next = null;
else    n.prev.next = n.next ;
        n.next.prev = n.prev;
```

Double LinkedList , write recursive to middle.
Assume odd no.

```
Middle (Head, Tail)
if (head == tail)
        return head.data;
else return middle ( head → next , tail → prev);
```

Single linked list

```
Middle (head &, length)
  if (len == 1)  return head;
```

↓

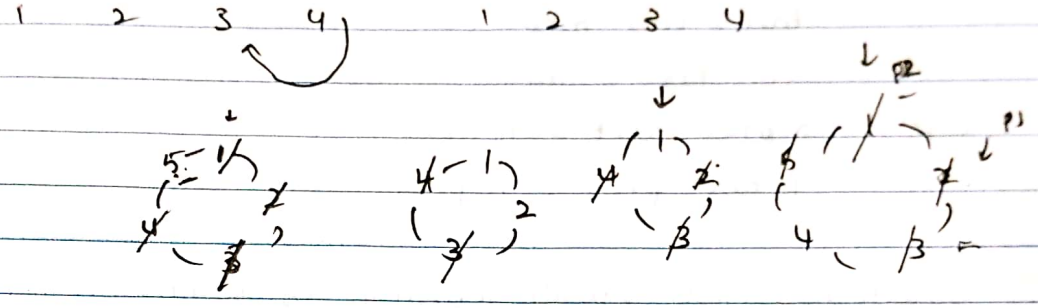
1 3 5 7 9

```
Middle ( tmp1, tmp2 → next)
  if (tmp2 == null)
        return tmp1
  else middle ( tmp1 → next, tmp2 → next → next )
```

Detect loop

```
Mersedup ( tmp1, tmp2)
  if (tmp1 == tmp2)
        return true;
  else return mersedup (tmp1->Next, tmp2->Next->Next)
```

if (tmp2 next or tmp2 == null)
return false




```
while (ptr1.next != ptr1)
    count = 1
    while( count != m )
        ptr2 = ptr1
        ptr1 = ptr1.Next
        c++;

p2.Next = p1.n        p1 = p2.Next
```

Stacks Last In First out

Queue First In First out

Sorting Algorithm

Q1) Swap the Nodes

```
for (i=0 ; i<size ; i++)
    for (j=i+1; i<size ; j++)
      if (a[i] < a[j])
```

```
Binarysearch (A, n, v)
    int l=0, int h = n-1;
    while (l<=h)
        mid = (l+h)/2;
        if A[mid] == v   return mid;
        elseif A[mid]>v    h = mid-1;
        else    l = mid+1
```

```
BS (A, l, h, v)
   mid = (l+h)/2.
If A[mid]==v  return mid;      if (l>h) return -1
elseif (l<=h) A[mid]>v  BS (A, l, mid-1, v)
else   return BS (A, mid+1, h, v)
```

0	1	2	3	4	5	6	
7	11	15	19	21	32	56	57.

(4,6) 5

(0,2)
(0,1)
(0,0)
(0,-1)

(6,6)
(7,6)

Sept

Root . Node

Internal Node / External Node / Leaf Node

Depth → No. of ancestors

Height → Max depth of any node

Binary Tree – Each internal node has atmost 2 children

Full Binary Tree – Every node has 2 children

Perfect Binary Tree – Every node has 2 children, All leaf nodes present

complete Binary Tree – Same as perfect, but leaf nodes can be missing, Present leaf nodes must be to the right.

Total No. of nodes in smallest complete BTree of Height h

$2^h - 1 + 1 = 2^h$

Traversal DFS BFS → Level order

DFS → Inorder LDR, Preorder, Postorder

Preorder DLR

Postorder LRD

```
preorder (root)                    DLR
    cout << (root->data)           if (tmp==NULL) return;
    preorder (root->left);
    preorder (root->right)
```

Postorder (LRD)

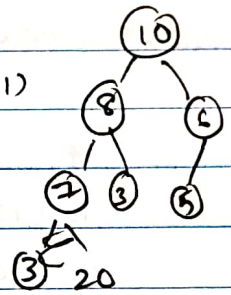
```
if (tmp == null) return;
postorder ( tmp→left);
postorder ( tmp→right);
cout << tmp→data
```

Inorder LDR

```
if (tmp == null) return;
Inorder ( tmp→left);
print tmp→data;
Inorder ( tmp→right);
```

Given a Binary Tree count no. of Nodes

```
count ( root *, count)
if (root != null)  count++;
count ( root→left);
count (root→right)
if (root == null) return;
```




```
if (tmp == null) return 0;
else return  left + right + 1;
```

Count leaf nodes

```
if ( tmp.left == null && tmp.right == null)  return 1;
                (if tmp = null)
else return 0;
else   count(tmp.left) + count(tmp.right)
```

c(10)

/

c(8)

Given node n find Sibling ,

```
Sibling (root, v)
  if ( tmp.left ==

if (tmp.left.val == v)
        return tmp.right.val
elsif (tmp.right.val == v)
        return tmp.left.val
else
else  Sibling (tmp.left);
      Sibling
else  return s( tmp.left )
```

```
if (r == null) return null;
if (r == v)    ret null
(r.left == v)      r.right
(r.right == v)     r.left
S = Sib(r.left, v)
(10)  if (S != null)
(12)  ret Sib(r.right, v)
      else return S
```

S(10)

S(8) S(12)

Given node find parent

```
if   Parent (root, node)
if (root == null)  return null;
if (root == node)  return null;
if (root->left = node || root->right = node)  return root;
else return
    S = parent (root->left, node)
    if (S != null)
        ret parent (root->right, node)
    else ret S;
```

Node with largest value

```
if (tmp == null) ret 0;
if (Node temp.left == null && temp.right == null)
        return temp.val
else ret Max(temp->data, Max(temp.left), Max(temp.right))
```

Find mirror

```
Mirror (temp)
If (temp == null) ; return temp ;
if (temp->left == null && temp->right) return temp;
else
        temp1 = temp
        temp1.left = Mirror (temp.right)
        temp1.right = Mirror (temp.left)
        return temp1;
```

```
else swap (left, right)
     Mirror (left)
     Mirror (right)
```

(5)

Array Based representation

Index 0 stores No. of elements

Null Node -1

children of index i 2i, 2i+1

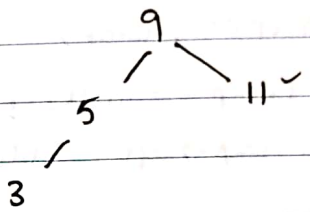
Parent of index i i/2

Error re Array representation of complete Binary tree will have -1 to the end.

Insert / Delete only in Binary search Tree

```
InsertBST (node, val)
 if (node == null)
        return new node (val)
 if (node.val < val) && node.right == null)
        return InsertBST (node.right, val)      node -> right =
 else if (node.val > val)
        return  InsertBST (node.left, val)
```

12




```
if (node.left = null && right = null)

if (node == null)
        return new node (val)
if (node.val < val)
      if if (  .left = null && .right = nul

if (.left = null && .right. null)
        return new node (.val)
if  (  < val)  InsertBST

if (node.val < val)
      if (node.right == null)
                insert & return
        rinsert (node.right, val);
if (node.val > val)
        if (node.left == null)
                insert & return
        rinsert (node.left, val);
```

⇒ Do Delete (node, val)

Pred → Largest on left
Succ → Smallest on right

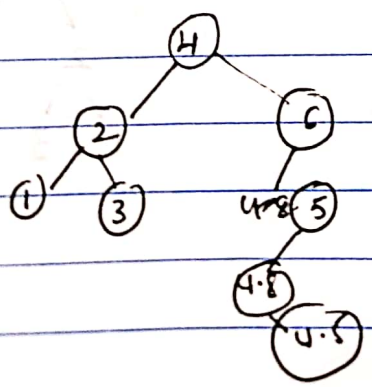
BFS → Level order Traversal
Use Queue Iterative

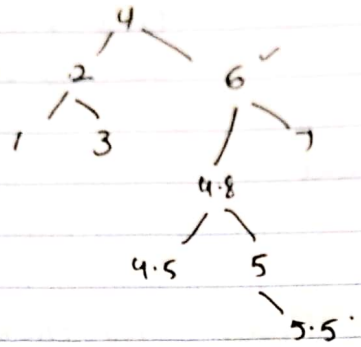
** In a tree which is not Binary No Inorder possible

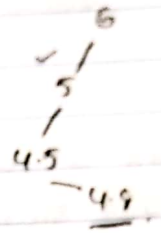
Bubble sort

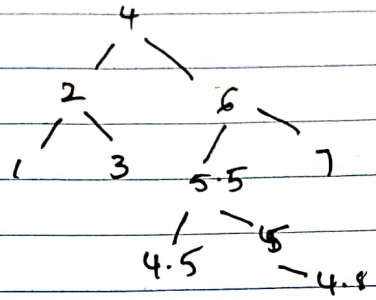
```
for (i=0 ; i<n ; i++)
    for(j=i+1 ; j<n-i ; j++)
        if a[i] > a[j]
            swap a[i], a[i]
while (swap = true)
    swap = false
    for (i=0; i<n-1; i++)
        if a[i] > a[i+1]
            swap(a[i], a[i+1]
            swap = true
```

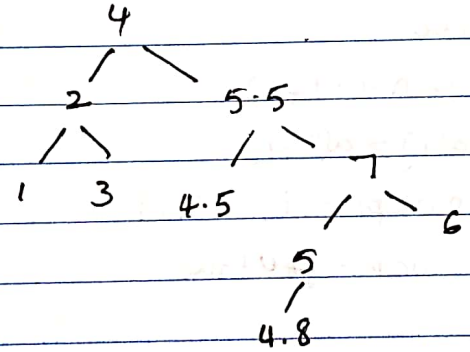
AVL Trees (Balancing Algorithm)
An AVL Tree is BST such that height of left and right differs by 1

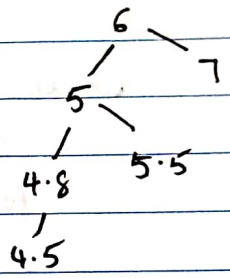


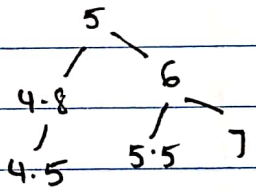


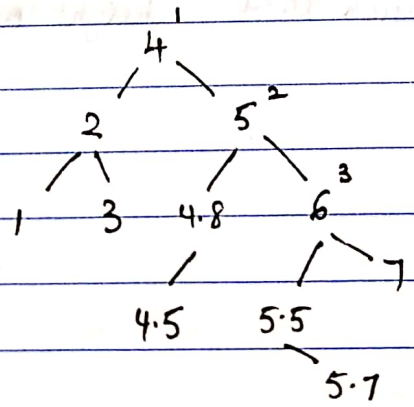


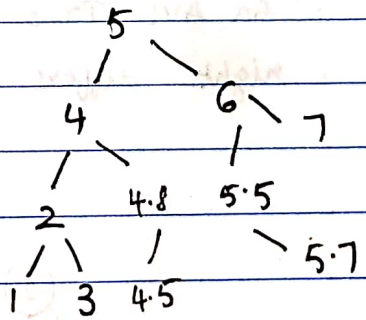


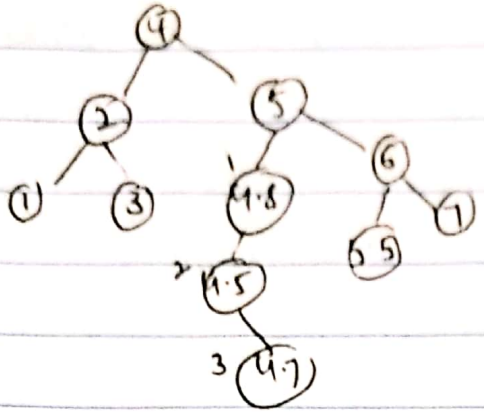


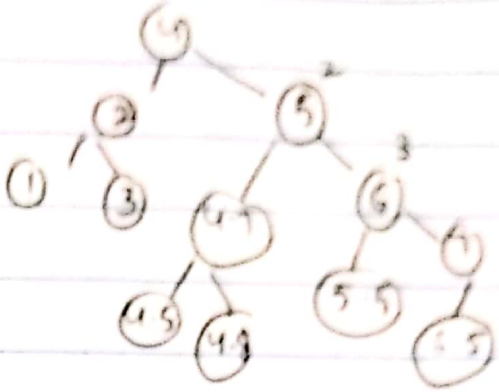


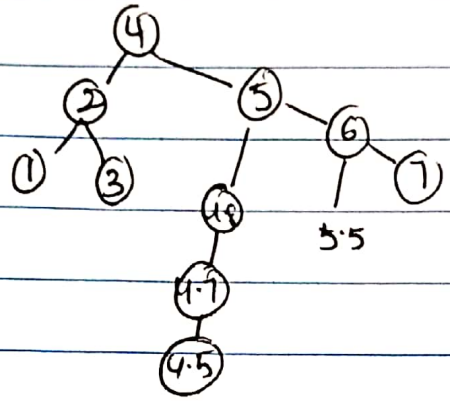


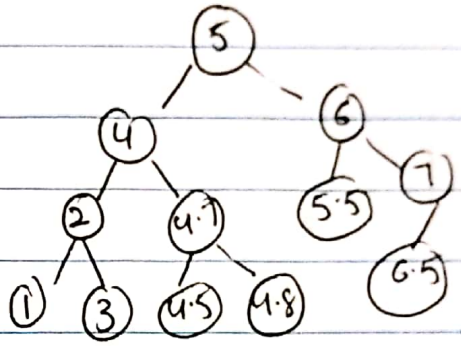


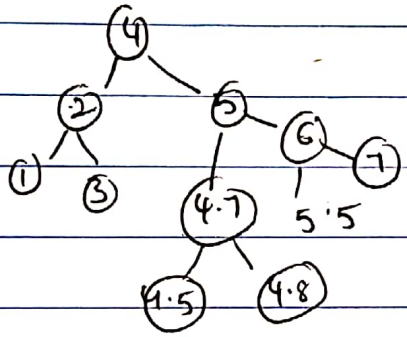


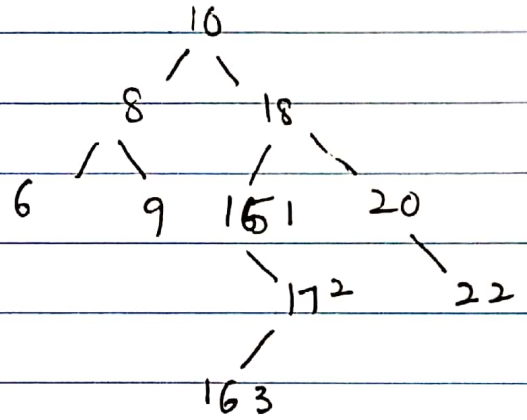


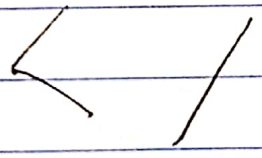


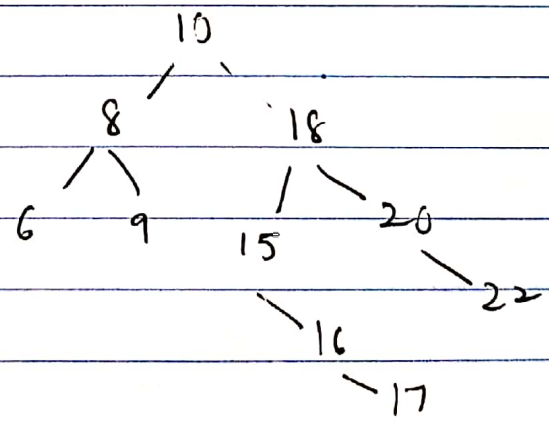


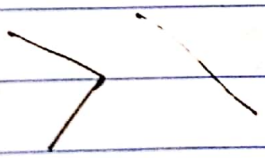


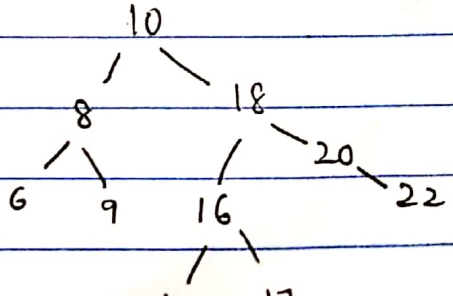


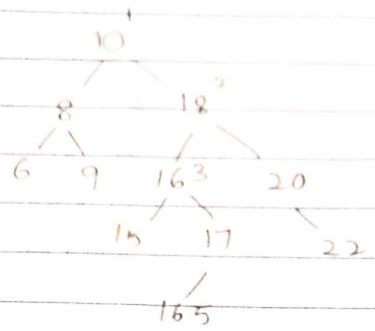






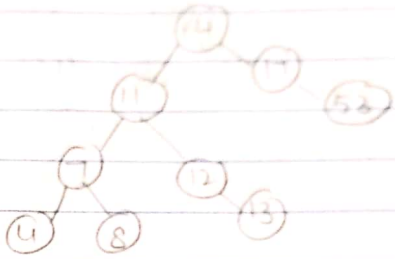




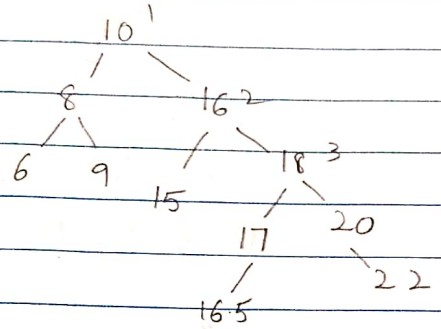


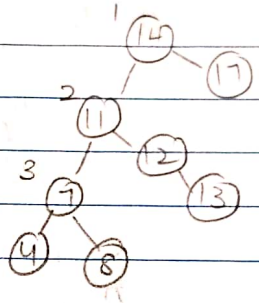
Delete

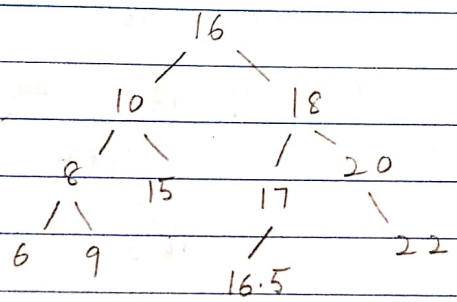
First step what BST says



Delete 53

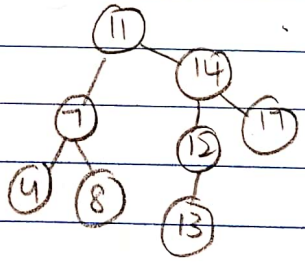


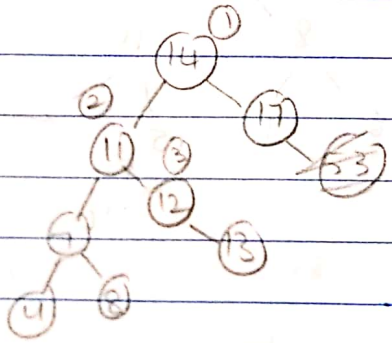


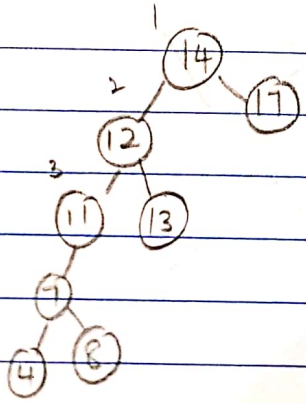


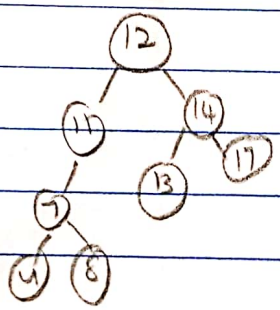
1,2,3 from longest

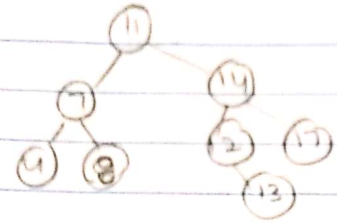
Both 7,12 can be 3









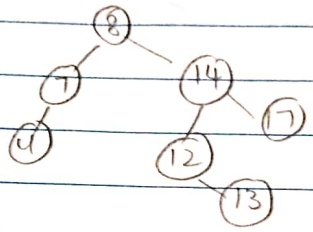


Delete 11

predecessor 8

physically 8 is deleted

start numbering from 8th parent

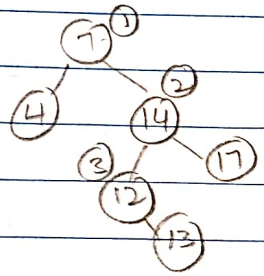


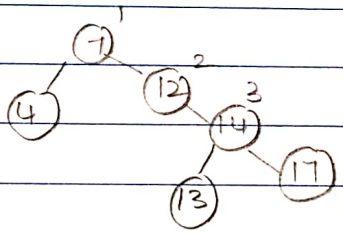
Delete 8

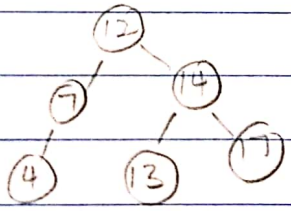
predecessor 7 Replace 7

succ 12

Replace with 12





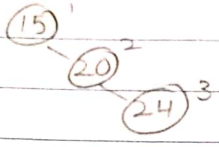


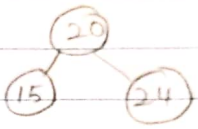
Build 15, 20, 24, 10, 13, 7, 30, 36, 25

15

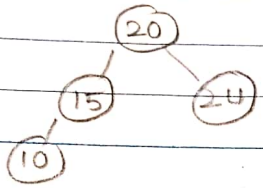
20

24

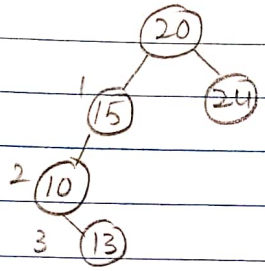


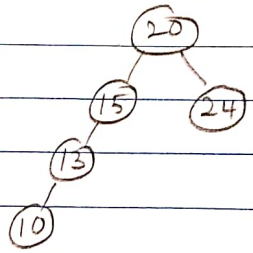


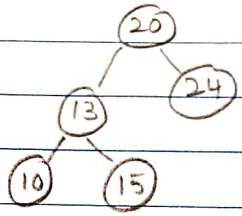
10



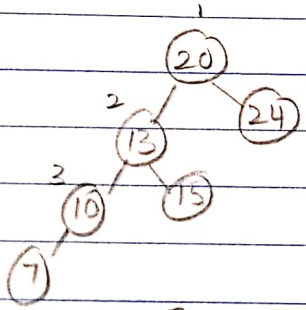
13

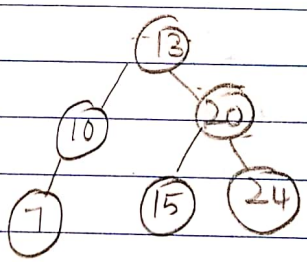




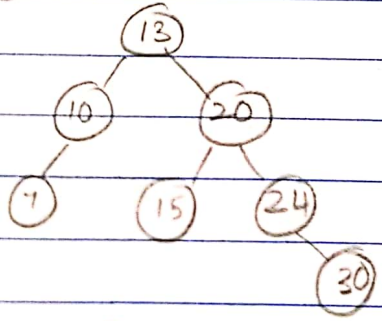


7

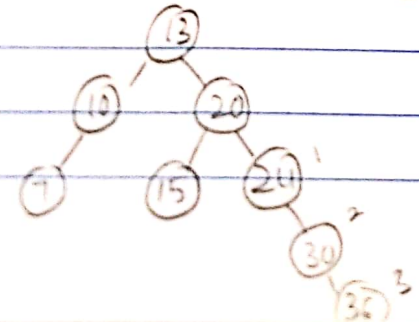


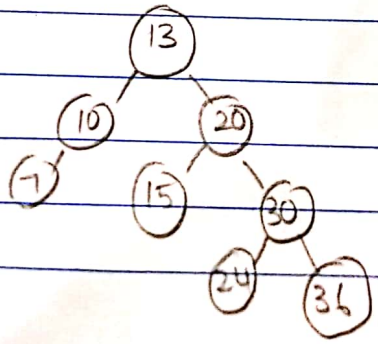


30

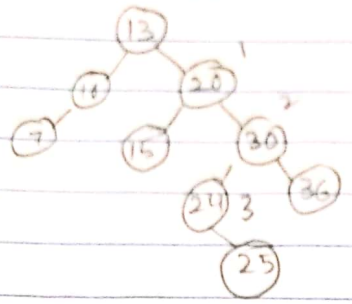


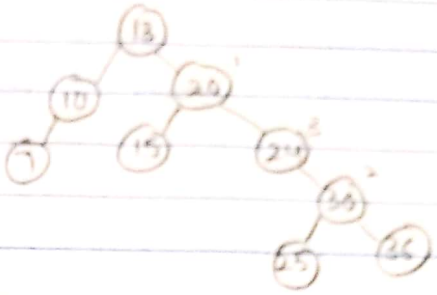
36

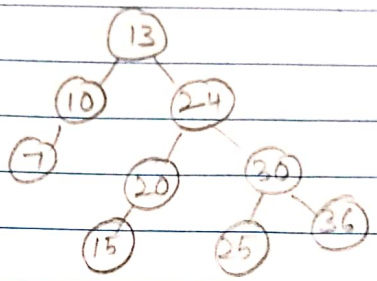




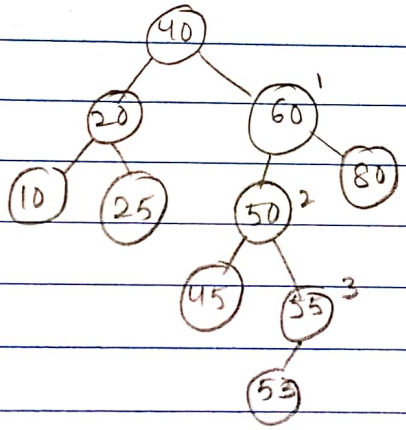
25

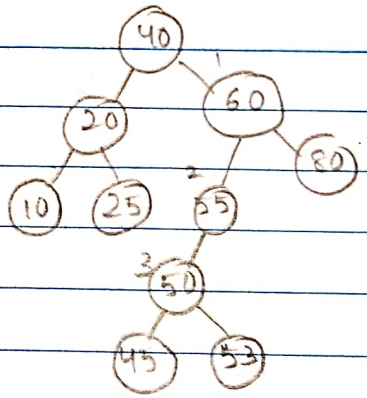


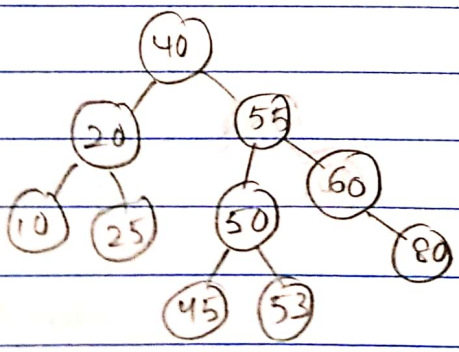




When choice - Always take straight line.

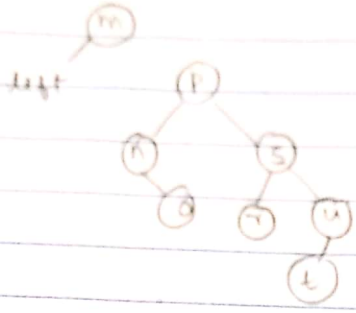


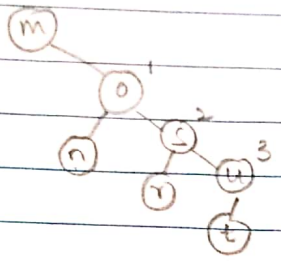




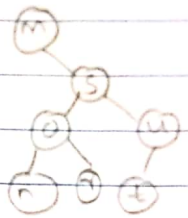
** Double Balancing

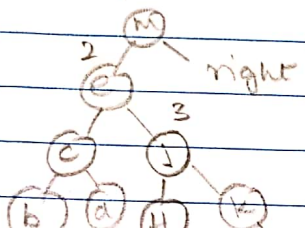
Delete 0

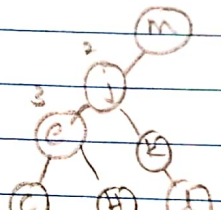




After balance







```
AvlNode {
    AvlNode lchild;
    AvlNode rchild
    AvlNode parent
    int ht;
```

HeapTree

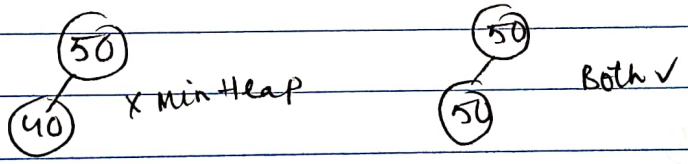
Min Heap , Max Heap

Parent smaller/Equal than both children — Parent larger/Equal than both its children

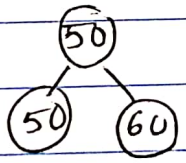
* Must be Complete Tree

We look at Binary HeapTree

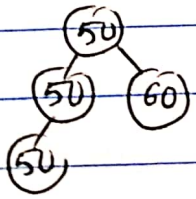
Given 2 nodes MinHeap, BST both



Given 3 nodes MinHeap, BST both



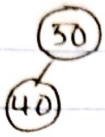
Given 4 nodes MinHeap, BST both



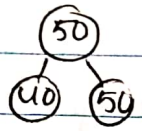
Given 5 nodes

Not possible

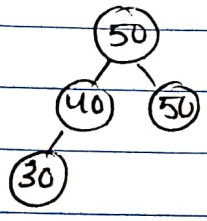
Given 2 Nodes MaxHeap, BST



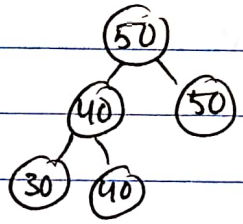
Given 3 nodes MaxHeap, BST



Given 4 nodes

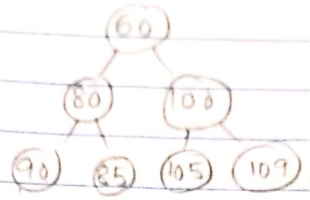


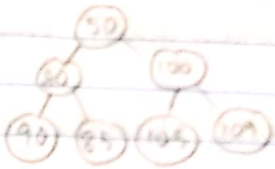
Given 5 nodes



Given 6 nodes

✗ Not possible





Insert ⑤ first structure and keep swapping

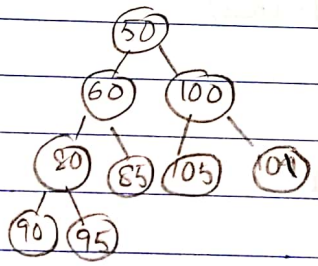
Percolate up

Here Delete is always the root.
First maintain structure and last leaf becomes root
Compare it with two children and min of 3 goes up
This is percolate down.

~~queue~~

60 80 100 90 85 105 109

Representation → Array



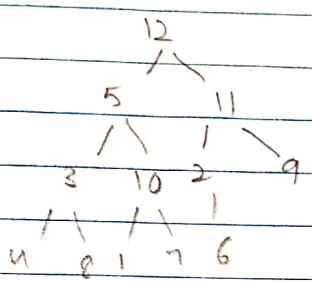
a	9	50	60	100	80	85	105	109	90	95
	0	1	2	3	4	5	6	7	8	9

Delete is always a[1]
Replace with a[9]
Reduce size a[0]--
Compare 2i, 2i+1 replace with smaller

Build Heap Floyd's Method

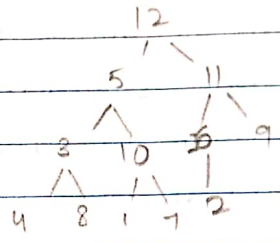
① Add elements to form a complete tree
② Pretend it's a heap and fix heap order
   start from last parent lastIndex/2
   percolate down

12 5 11 3 10 2 9 4 8 1 7 6



Next 10

Start 2

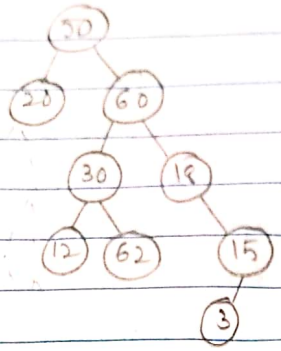


Next 3

Heapify

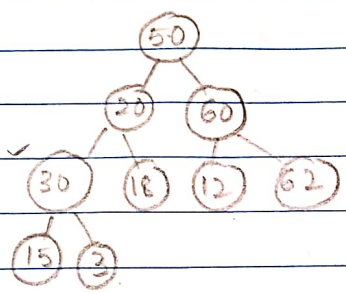
Max swaps level 0 – 0 (leaf nodes)
1 – 1

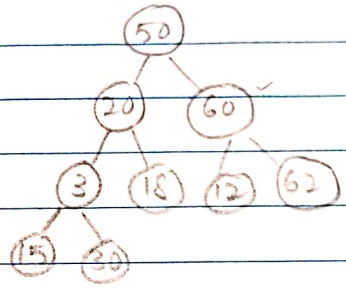
Build a Min Heap

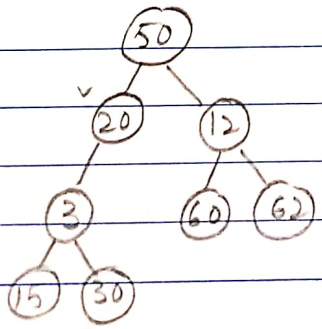


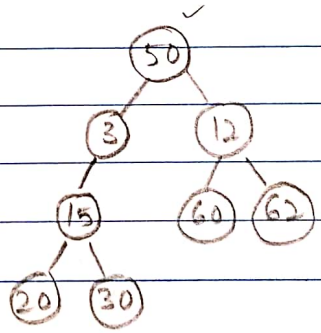
50 20 60 ~~A~~ ~~G~~ 30 18 ~~d~~ ~~D~~ ~~J~~ ~~K~~ 12 62 ~~P~~ 15

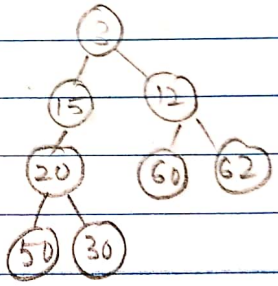
3











Build using insert method takes O(nlogn)

Floyd's method is less O(n)

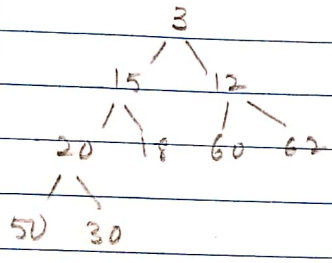
Heapsort Make heap O(n)

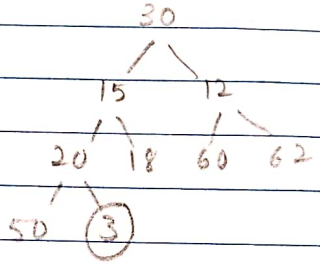
percolate down O(logn)

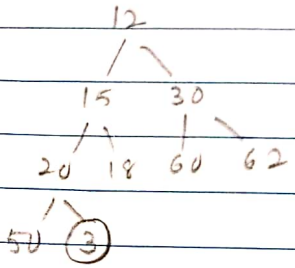
Heap sort O(nlogn)

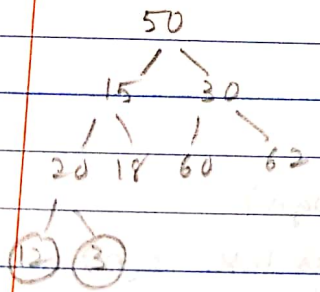
Heap sort

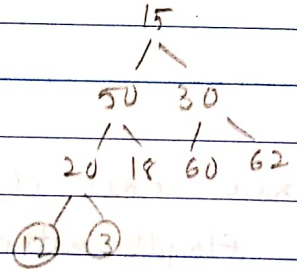
0	1	2	3	4	5	6	7	8	9
9	3	15	12	20	18	60	62	50	30
6	30	15	12	20	18	60	62	50	3
8	12	15	30	20	18	60	62	50	3
7	50	15	30	20	18	60	62	12	3
7	20	15	30	50	18	60	62	12	3
6	62	15	30	50	18	60	20	12	3
6	30	15	62	50	18	60	20	12	3
6	30	15	60	50	18	62	20	12	3
5	62	15	60	50	18	30	20	12	3
5	50	15	60	62	18	30	20	12	3
4									

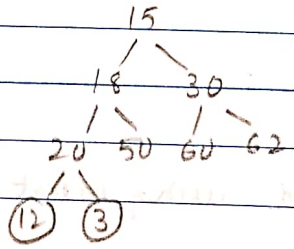


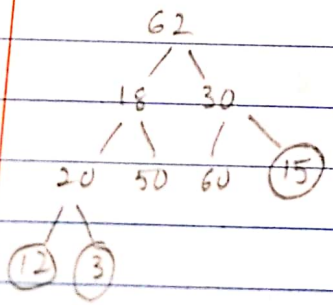


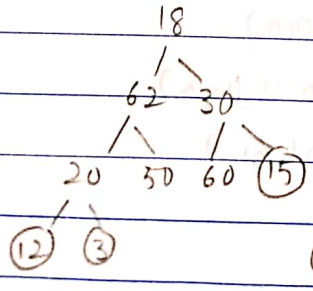


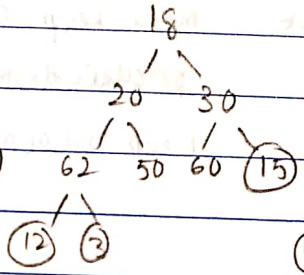


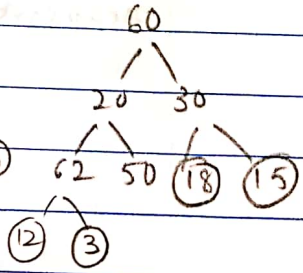


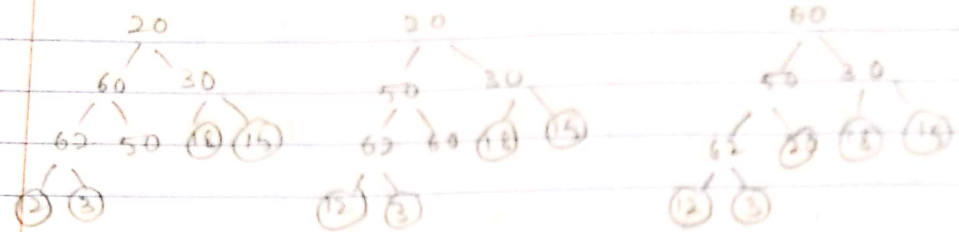












```
Heapify (A,n)
    parent = A[0]/2
    while (parent != 0)
        percolate down (A, Parent)
        Parent--;
```

Heap Sort
1. Make a Heap out of given NOS
2. place decliner element at end
3. Decrease A[0]
4. percolate down A[1]

Merge Sort

O(n logn)

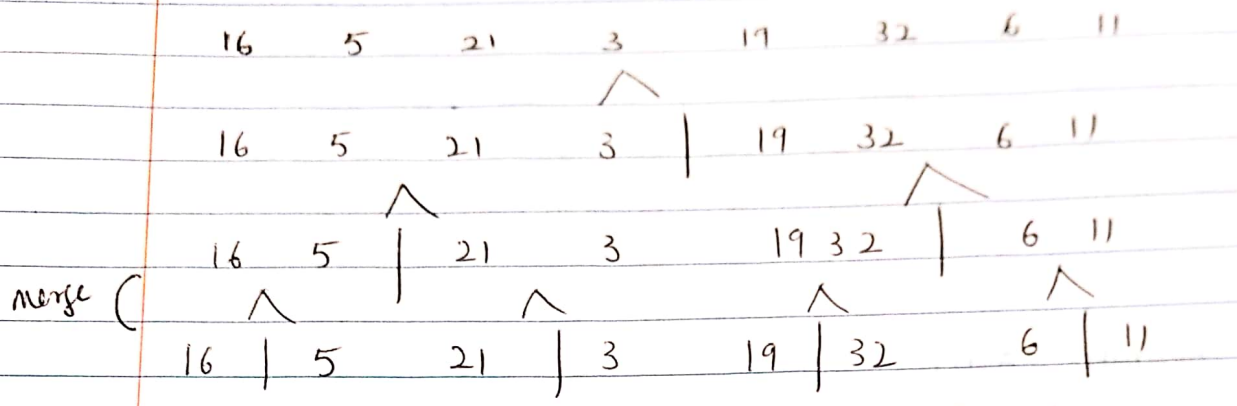
Total logn levels

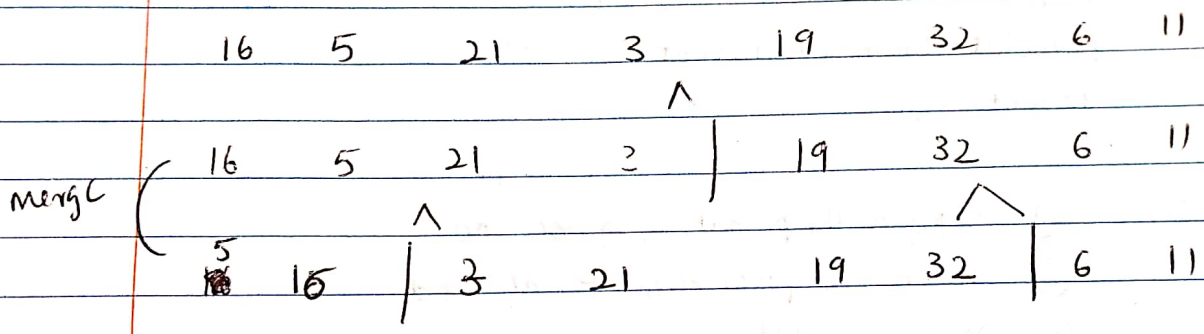
At each level n comp ∴ n logn

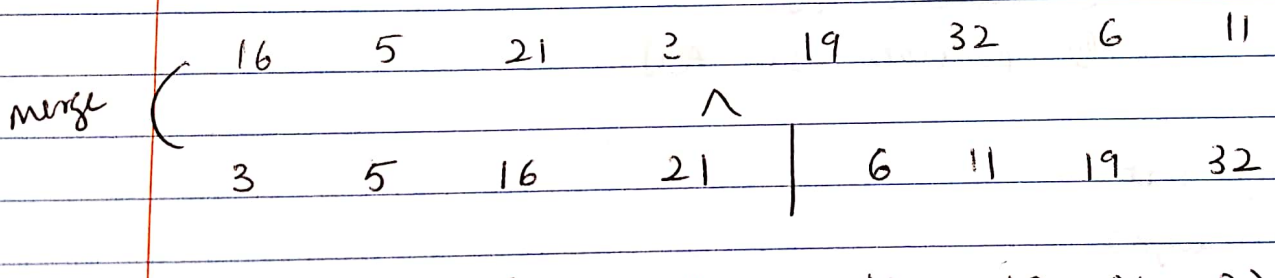
```
MergeSort ( a[], left, right ) {
    mid = left+right/2
    if (right<= left)
        return;
    Mergesort (a, left, mid)
    Mergesort (a, mid+1, right)
    merge (a, left, mid, right)
```

Oct 8

Exam







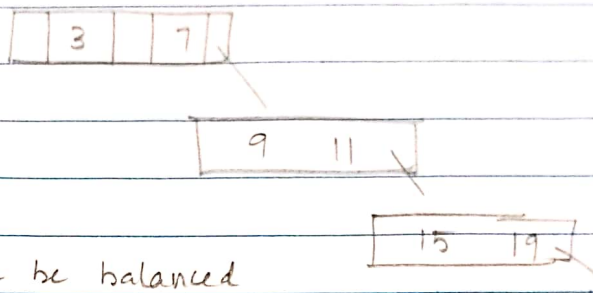
Final 3 5 6 11 16 19 21 22

B Tree

m-1 keys

m children

-> 3 children 3 7 9 11 15 19 21



This also should be balanced

then search will be $\log_m n$ → no of children

Balance condition left height = right ~~thi~~ height

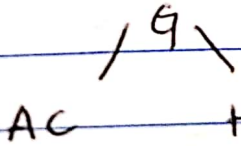
	Root Node	Non Root Node
Min key	1	$\lceil m/2 \rceil - 1$
Min Subtree	2	$\lceil m/2 \rceil$

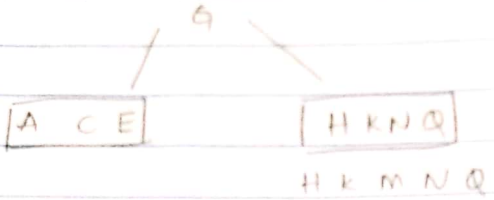
Array root — 1 key, 2 children non root — 2 key, 3 children max → 4

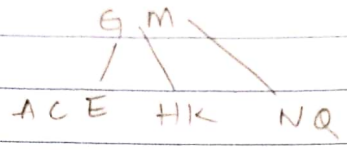
C N G A H E K Q M F W L T Z D P R X Y S

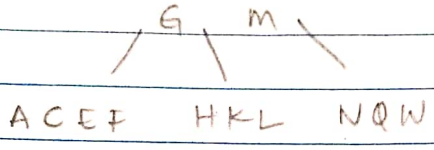
A C G N

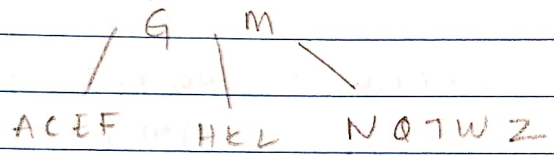
A C G H N

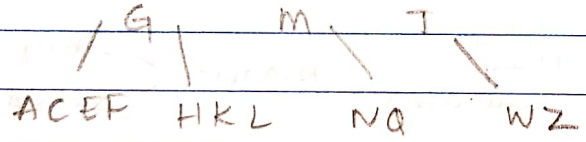


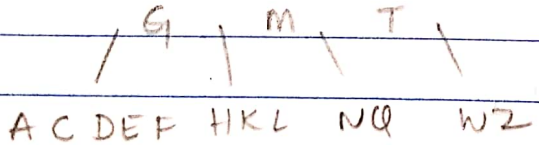


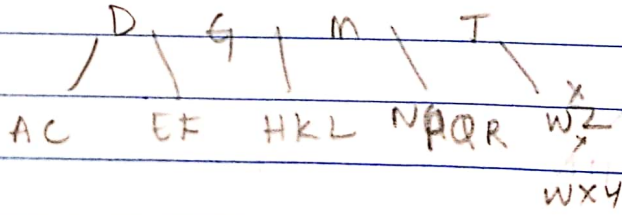








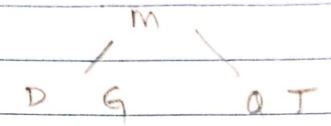




D G M T

N P Q R S

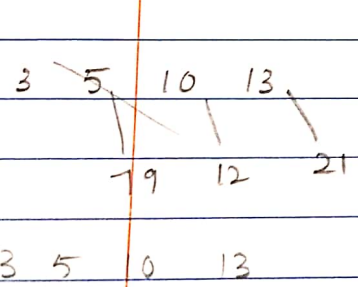
D G M Q T

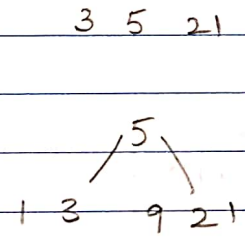


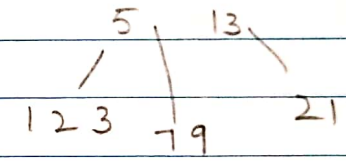
even keys → decide how to decide and be consistent

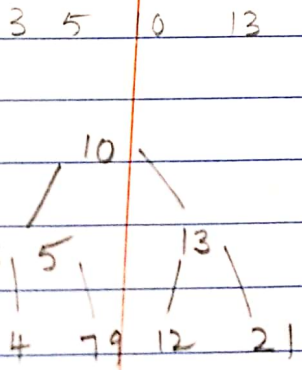
order 4 root < 1 key 2 children non root < 1 key 2 children max → 3

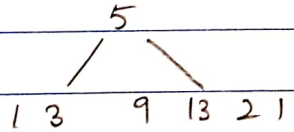
5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8

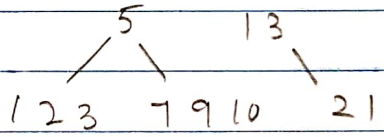


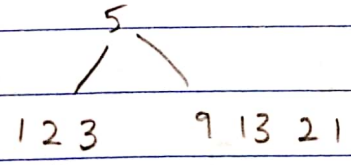


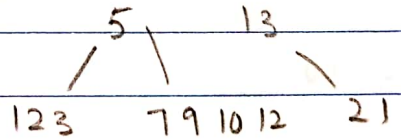


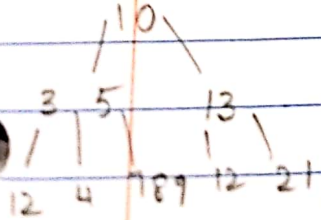


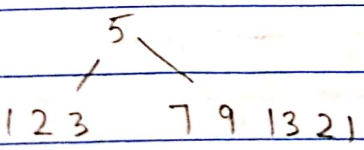


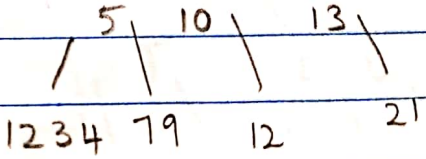




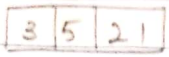






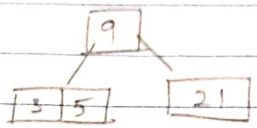


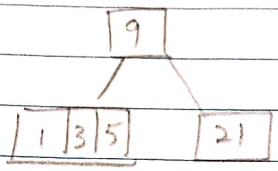
5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8

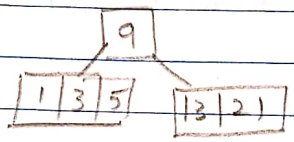


3 5 9 21

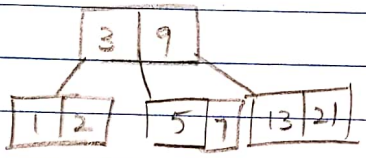
3 5

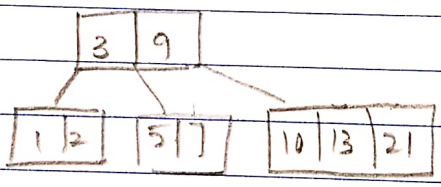




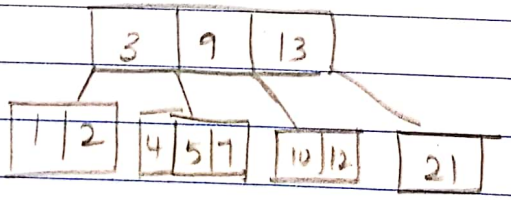


1 2 3 5





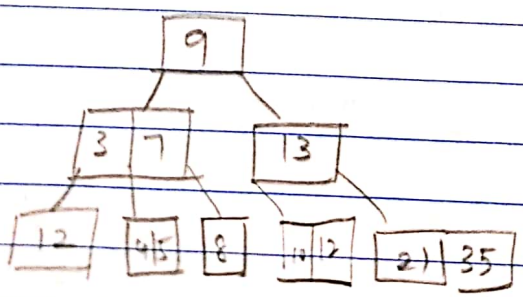
10 12 13 21



3 7 9 13

4 5 7 8

35, 40

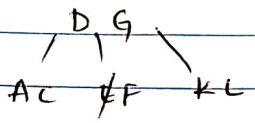


Delete

→ leaf node — Just delete it

→ Non leaf node — Replace with Succ or Pred
if underflow doesn't take place

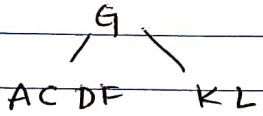
If underflow → look for siblings having spare key
if any has extra borrow
accordingly



Delete E

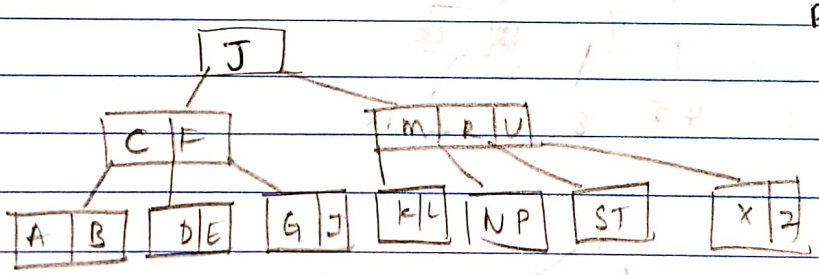
Underflow

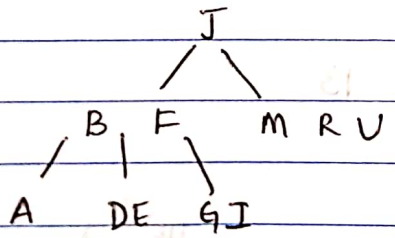
No spare → Merge

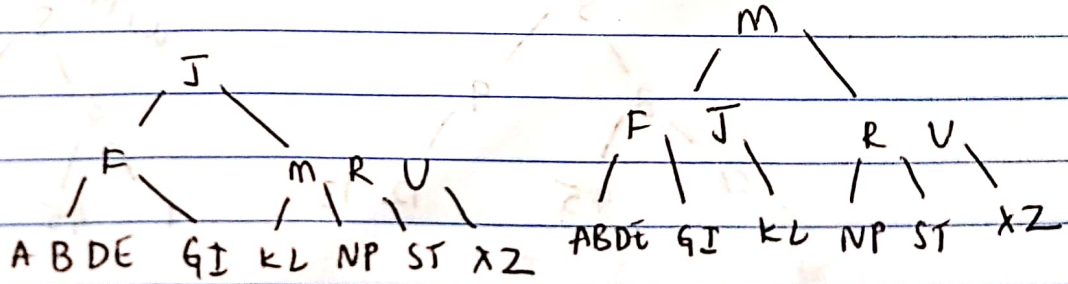


Order – 5 Min 2

pred

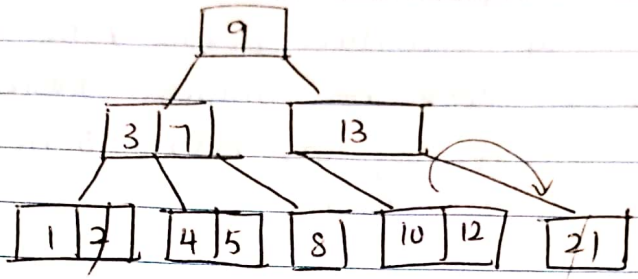




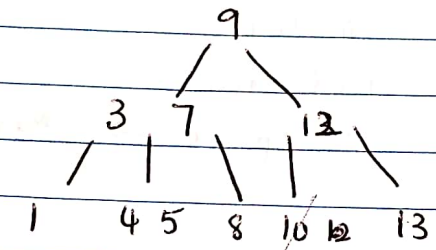


Order 4 3keys max
1 min

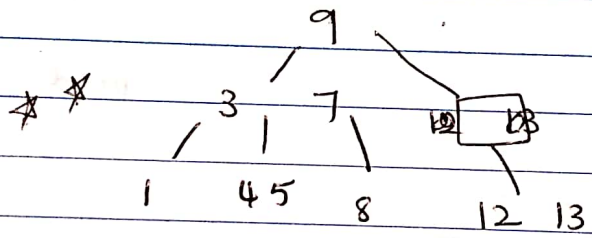
Delete 2



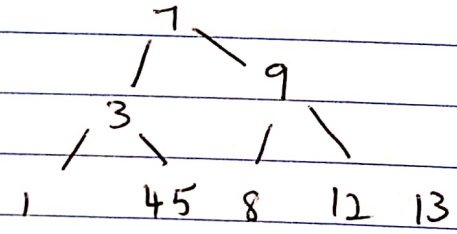
Delete 21

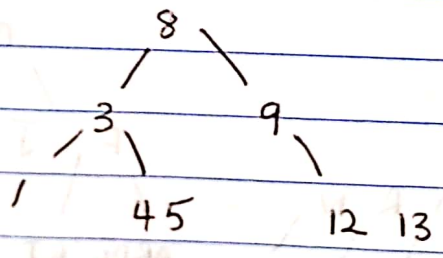


Delete 10

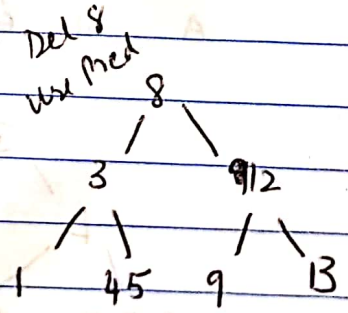


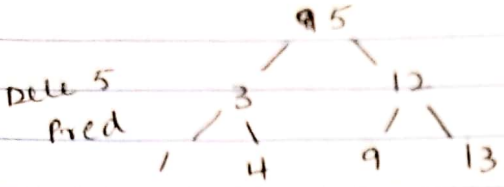
Delete 7 succ

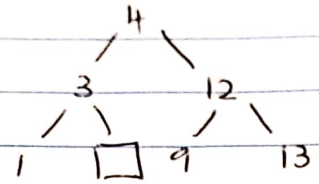


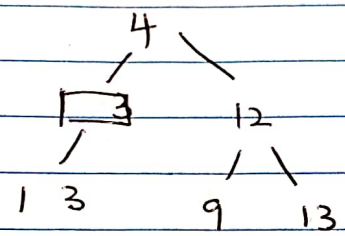


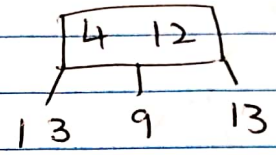
Del 8 use pred



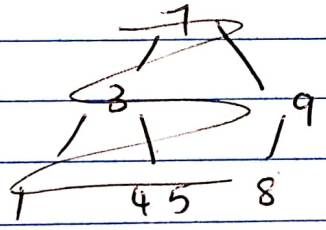


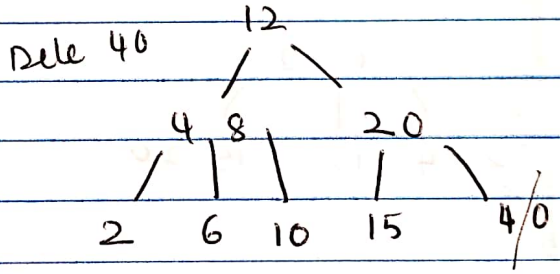


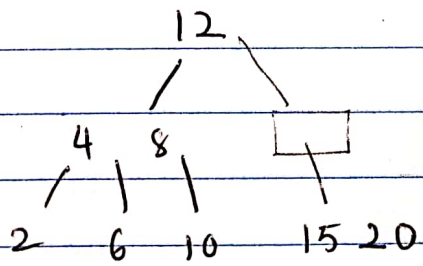


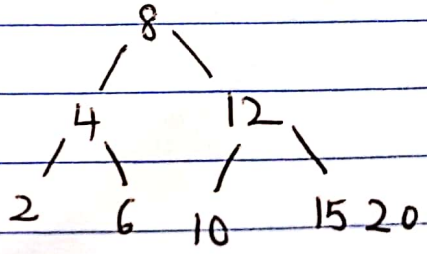


order 3 max-2 min 1

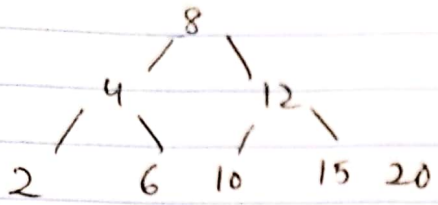


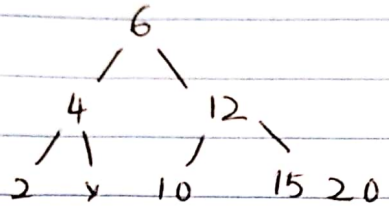


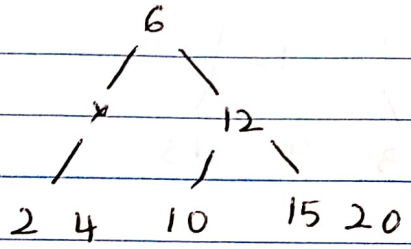


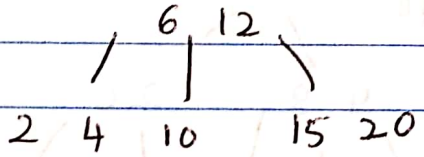


Del 8
use pred





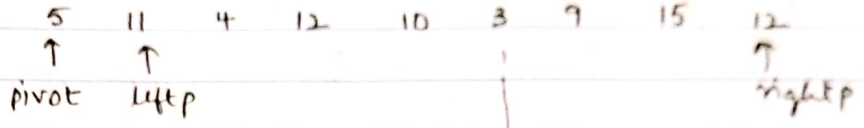




Quick Sort

Worst case $O(n^2)$

Average $O(n \log n)$



look to no smaller than pivot

5 < 12

5 < 15

9 < 15

5 < 11

swap (11, 3) move left pointer

5 3 4 12 10 11 9 15 12

left = right

swap 4 & 5

4 3 5 12 10 11 9 15 12

First pass pivot = 5 left of 5 is smaller
right of 5 is larger

QS (A, 0, 8)

QS (A, 0, 1) QS (A, 3, 8

4 3

pivot = 4 ↑ right

↑ left

swap 3 and 4

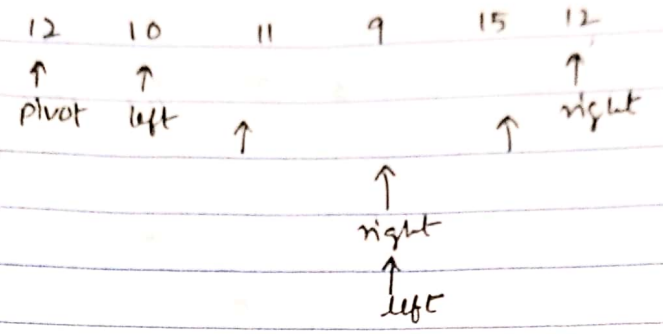
3 4

QS (A, 0, 1) pivot+1 pivot-1

QS (A, 0, 0) returns left = right

QS (A, 2, 1) returns left > right

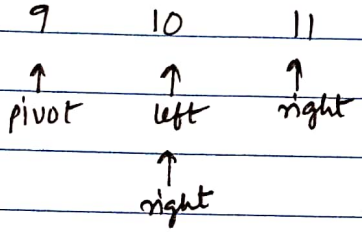
QS (A, 3, 8)



swap with pivot ∴ (9, 12)

9 10 11 12 15 12

QS (A, 3, 5) QS (A, 7, 8)



Swap with pivot ~~9, 10~~ left = right but not small than pivot

Swap 9 with left of 10

to Swap 9 with 9

QS (A, 3, 2) QS (A, 4, 5)

10 11 Swap 10 with 10

p

QS (A, 4, 3) QS (A, 5, 5)

QS (A, 7, 8)

15 12 Swap 15 & 12 12 15

p

QS (A, 7, 7) QS (A, 9, 8)

0 1 2 3 4 5 6 7 8 9 10 QS(A,0,8)

8 3 19 6 1 15 9 2 10

pivot, left, left, right, right

Swap 19, 2

8 3 2 6 1 15 9 19 10

pivot, left, right, left, right

Swap 8 & 1

0 1 2 3 4 5 6 7 8 9 10 QS(A,0,3) QS(A,5,8)

1 3 2 6 8 15 9 19 10

0 1 2 3

1 3 2 6

pivot, left, right, right

Swap 1 with 1 QS(A,0,-1) QS(A,1,3)

return

3 2 6

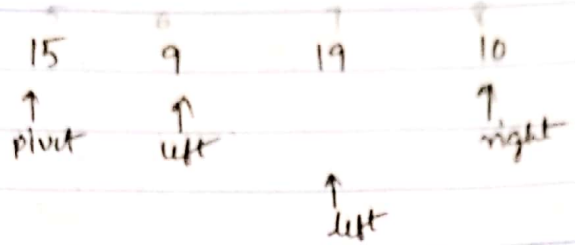
pivot, left, right, right

1 2 3

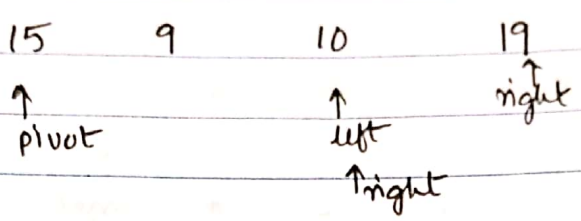
2 3 6 QS(A,1,1) QS(A,3,3)

∴ 1 2 3 6 8

QS(A,5,8)



Swap 10, 19



Swap 10 and 15

5	6	7	8
10	9	15	19

QS(A,5,6) QS(A,8,8)

5	6
10	9

pivot l r swap 9 and 10

5	6
9	10

QS(A,5,5) QS(A,7,6)

∴ 9 10 15 19

1 2 3 6 8 9 10 15 19

pivot lr

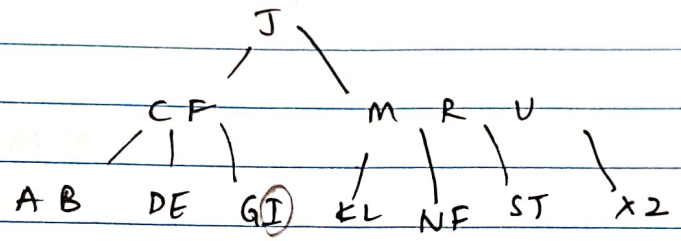
If lr is less than pivot swap (pivot, lr)

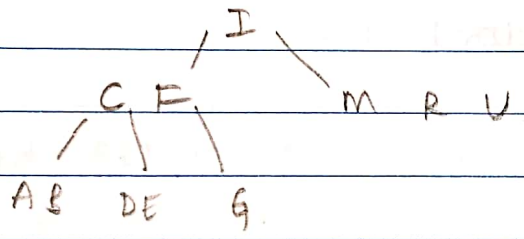
If lr is greater than pivot swap (pivot, left of lr)

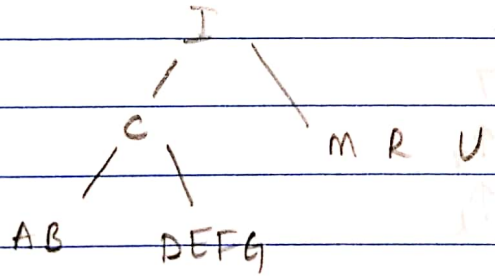
```
BTNode {
        int val[m-1]        int count
        BTnode child[m]
        BTnode parent
};
```

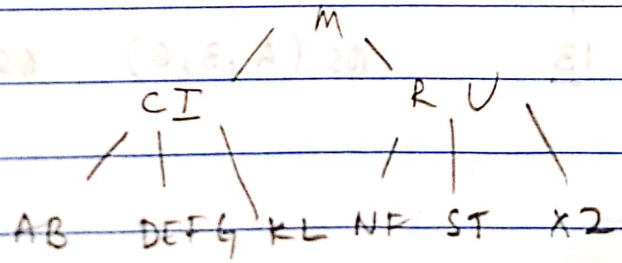
Order 5 max − 4 min 2.

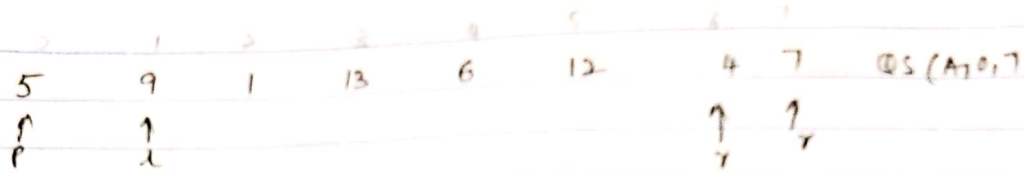
Del J
pred.





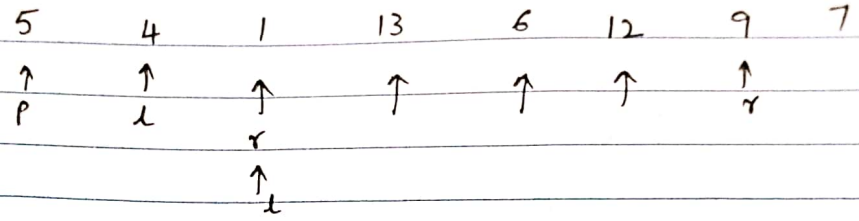






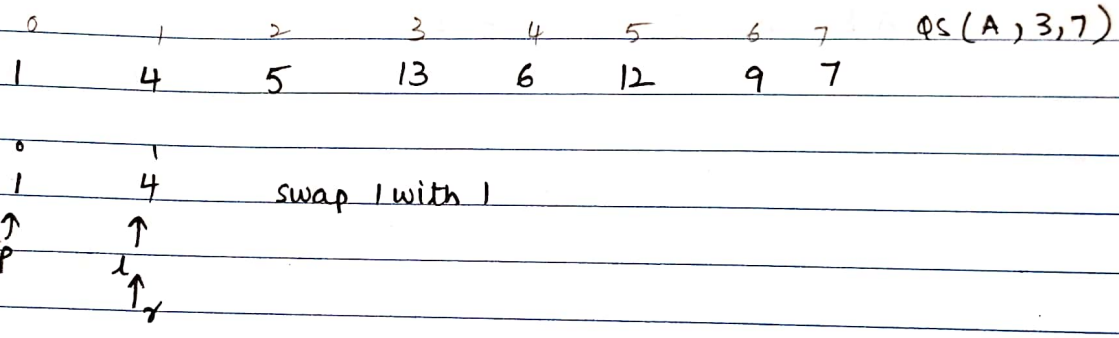
swap 9 & 4

5 4 9 1

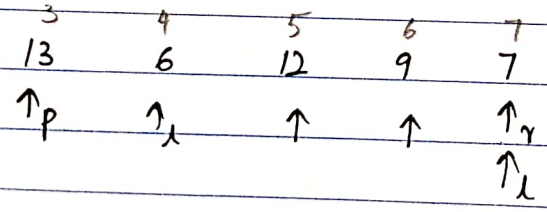


swap 1 and 5 QS(A,0,1)

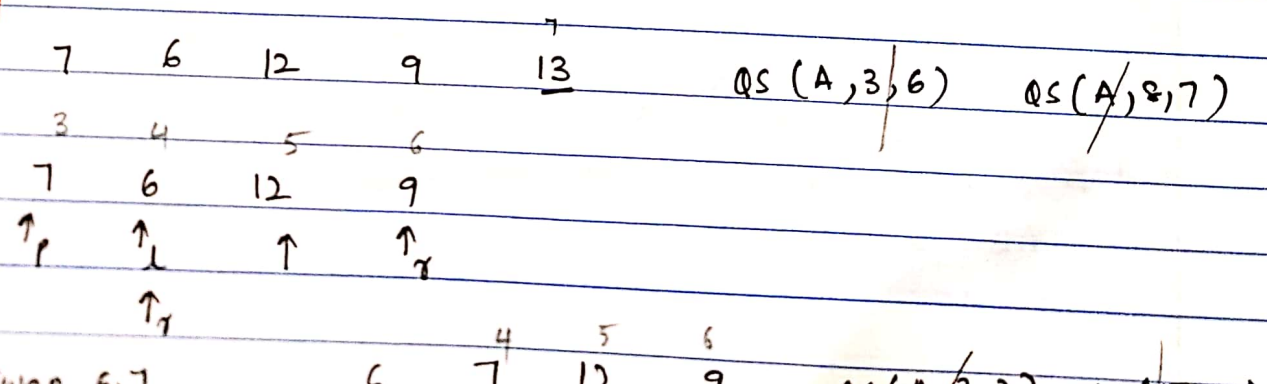
QS(A,3,7)



1 4 QS(A,0,-1) QS(A,1,1)



swap 7, 13



12 9 Swap 9,12 9 12 AS(A,5,5) AS(A,7,6)

↑p ↑l

↑r

1 4 5 6 7 9 12 13

Each time we move 2 no.s at a time so, their nlogn is better than other nlogn

Analysis

Recursion

Linked Lists - Single, Double

Stacks Queues

Binary Trees - Definition Complete, Perfect

Count nodes, leaf, sibling, parent, height

traversals - trees with more than 2 children

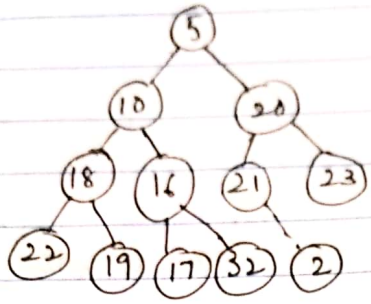
BST

Balanced BST is AVL - Insert | Delete

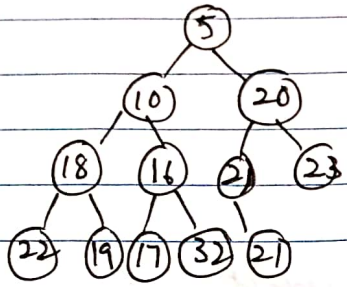
Heap Tree - given random no.s → make heap

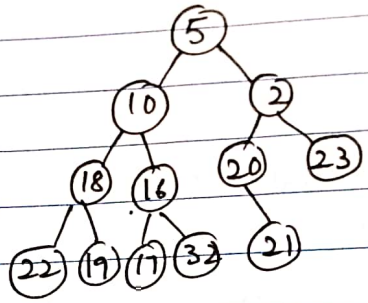
BTree - Insert, Delete

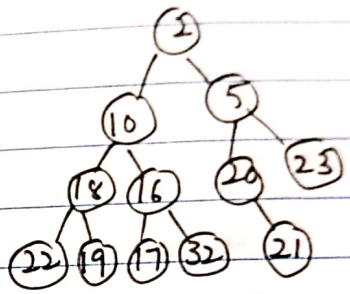
Min Heap



Insert 2







Max Heap

	1	2	3	4	5	6	7	8	9
	9	5	13	2	7	19	21	15	23	12

no. of

$9/2 = 4$

			3				6	7		
9	5	13	2	23	19	21	15	7	12

		2		4	5				
9	5	13	21	23	19	2	15	7	12

				4					
9	5	23	21	13	19	2	15	7	12

		2		4	5				
9	23	5	21	13	19	2	15	7	12

	1	2	3	4	5	6	7	8	9
9	23	19	21	13	5	2	15	7	12

Heap sort

	9	23	19	21	13	5	2	15	7	12
swap 23,12, red size										
	8	12	19	21	13	5	2	15	7	23
down										
	8	21	19	12	13	5	2	15	7	23
	8	21	19	15	13	5	2	12	7	23
	7	7	19	15	13	5	2	12	21	23
	7	19	7	15	13	5	2	12	21	23
	7	19	13	15	7	5	2	12	21	23
	6	12	13	15	7	5	2	19	21	23
	6	15	13	12	7	5	2			
	5	2	13	12	7	5	15	19	21	23
	5	13	2	12	7	5				
	5	13	7	12	2	5				
	4	5	7	12	2	13	15	19	21	23
	4	12	7	5	2					
	3	2	7	5	12	13	15	19	21	23
	3	7	2	5	12					
	2	5	2	7						
	1	2	5	7						

stop at 1.

Quick sort

0	1	2	3	4
7	1	9	18	6
↑p	↑l	↑l		↑r

QS(A,0,4)

QS(A,0,1) QS(A,3,4)

Swap (9,6)

7	1	6	18	9
↑p		↑l ↑r	↑	↑r

QS(A,0,0) QS(A,2,1) QS(A,3,3) QS(A,5,4)

Swap 7,6

0	1	2	3	4
6	1	7	18	9

pivot index = 2

QS(A,0,1)

0	1
6	1
↑p	↑l ↑r

Swap 6,1

0	1
1	6

pivot index = 1

QS(A,3,4)

3	4
18	69
↑p	↑l ↑r

swap 18,6

3	4
69	18

pivot index = 4

∴ 1 6 7 9 18

Bubble sort

P1 13 9 2 14 5
9 13 2 14 5
9 2 13 14 5
9 2 13 14 5
9 2 13 5 | 14

P2 9 2 13 5 | 14
2 9 13 5 | 14
2 9 13 5 | 14
2 9 3 | 13 14

P3 2 9 3 | 13 14
2 9 3 | 13 14
2 3 | 9 13 14

P3. 2 3 | 9 13 14
2 | 3 9 13 14

order - 5 max - 4 min 2

Insert

Directed edge — undirected edge

$(u,v) \neq (v,u)$ — $(u,v) = (v,u)$

u = source

v = destination

Directed graph — Undirected graph

Edge or vertex not repeated in a path → simple path

Origin and destination are same → cycle.

That is the only vertex repeated.

Length of path is no. of edges.

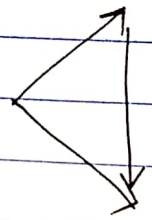
connected – a path exists b/n them

adjacent – one edge between them.

A graph is connected if there is path for every vertex to every other vertex

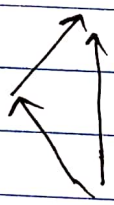
Acyclic graph → graph with no cycles.

A tree is acyclic connected graph



Directed

~~a~~ weakly connected



Directed

strongly connected.

Even with arrows → connected

with arrows, atleast one vertex

Sum total of all degrees of all vertices = 2 (no. of ~~vertices~~ edges)

undirected graph — No. of edges $\leq \frac{n(n-1)}{2}$

Prove by MI — $P(k)$ true — $P(k+1) = P(k) + n$

Directed graph $m \leq n(n-1)$

Graph Representation — Adjacency matrix

n vertices

Matrix $n \times n$

edge adj present i,j — $A[i,j] = 1$

not edge — $A[i,j] = 0$

For undirected → matrix is symmetric

Directed graph → 2 matrix are possible — i | (i) ; i | (o)

edge from 0 to 1

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	1	1	0	0	1	1
3	1	1	0	0	1	1
4	1	1	0	0	0	0
5	1	1	0	0	0	0

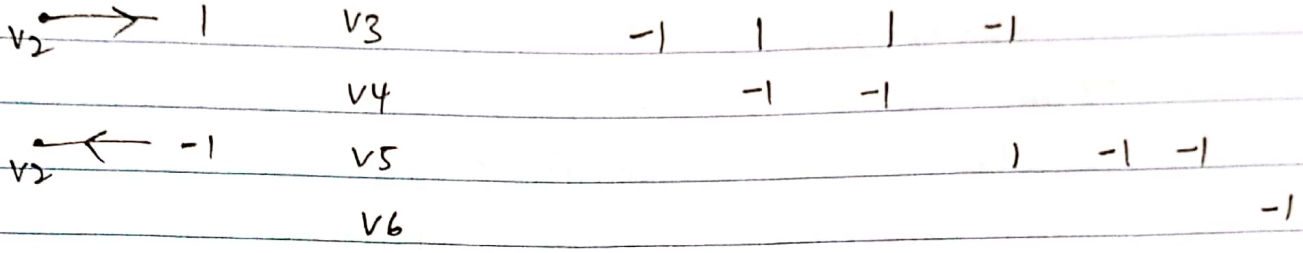
Incidence matrix $m \times n$

~~edges x vertices~~ vertices x edges

$m[i,j] = 1$ if edge e_j is incident on v_i

4×6

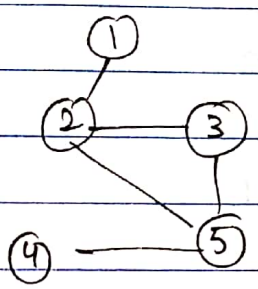
vertex	1	2	3	4	5	6
a	1	1			1	
b	1		1			
c				1	1	1
d		1	1	1		

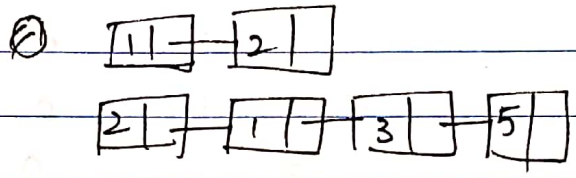


If self loop clockwise - 1

anticlockwise - -1

Adjacency list





Adjacency matrix $O(n^2)$

list $O(n)$, saves space

L(1) : 2, 6, 5

L(2) : 1, 7, 3

L(3) : 2, 4

L(4) : 3, 7, 5

L(5) : 7, 4, 1, 6

L(6) : 1, 5

L(7) : 2, 4, 5

Degree of a – 6

For directed

in degree, outdegree of a, 3, 3

Graph traversals

Depth First Search

Breadth first search

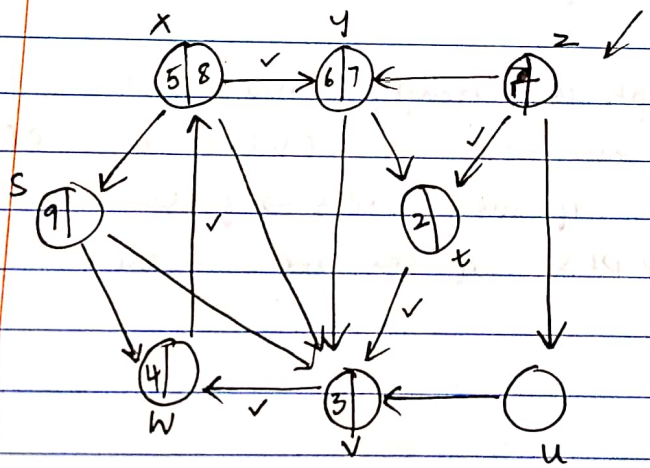
Depth First

go deep

- Trees

If strongly connected ~~graph~~ tree

weakly connected multiple trees



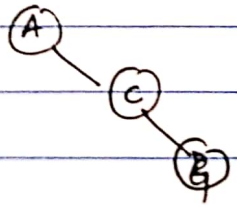
~~z t v w x s~~

z → t → v → w → x → y

u

s

when choice alphabetical order



A — B — D — E

F

C

G

int visited[7]

```
DFS[A]
    V[A] = 1
    for each adj
      an = getadj[A]
      if v[an] == 0
          DFS (an)
```

DF search First go to all adjacents
using Queue
O(V+E)
For dense graph in worst case e is n^2 then $O(n^2)$
Space complexity of DFS is less than BFS

15 oct For a digraph No. of edges $\leq n(n-1)$
Reachability
How to know if a graph is strongly connected ?
DO DFS from every node → $O(V(V+E)) = O(V^2)$
Pick one node DO DFS if more trees → false
Reverse all edges DO DFS if one tree → true

→ Pick a vertex
Run DFS
Ans 1 tree → Reverse edges
Pick same vertex O(V+E)
Run DFS
Ans 1 tree → True
else false
else false

How many components of a graph are strongly connected ?

Transitive closure -

If there is a path → create a direct edge

~~Institute A to D it~~

Floyd Marshall Algorithm

k is middle

k = V1 $i \to k \to j$

i = V2

j = V3

No edge from V_2 skip all

k = V1 V_3 to V1 yes

i = V3

j = V2

Start with k = V1 i = V2 j = V3

check $i \to k \to j$

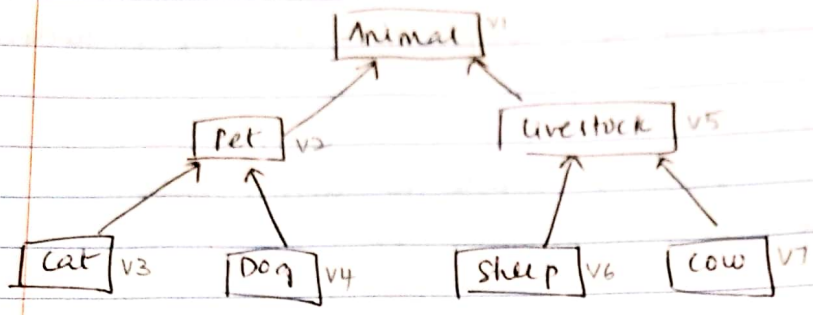
$i \to k$ fails

$i \not\to k \to j$

~~out of i~~ Increase j

Increase i

then increase k



i k j

I 1 i k = V_1 i = V_2 j = V_3

$V_2 \to V_1 \to V_3$ ✗

i k j

I k = V_1 i = V_2 j = V_3

$V_2 \to V_1 \to V_3$ ✗

$V_3 \to V_1$ ✗

1 k = Animal, Pet, LV, Cat,

Cat → Animal

Dog → Animal

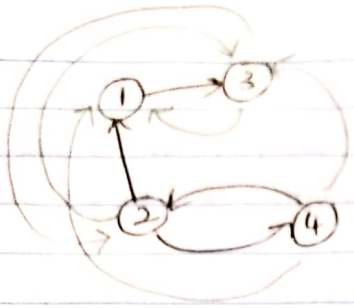
Sheep → Animal

k = Animal i = Pet

~~i → k~~ Pet → Animal ✓ → LV ✗

Pet → ~~LV~~

k =	Animal,	~~Pet~~,	Livestock,	Cat,	Dog, Sheep, Cow
i =	~~Pet~~	Animal	Animal		
~~j~~ =	~~LV~~	LV, ~~Cat~~ Dog, Sheep, ~~Cow~~	~~Pet~~ Cat Sheep		
j =	~~LV~~	Animal ~~Animal~~ Animal,	Animal		



k = 1 2 3 4

i = 2 3 4 | 1 3 4 | 1 2 4 | 1 2 3

2 → 1

j = 3 | 1 3 | 2 4 | 1 2

If there is an edge from i → k and k → j and from i → j if a ~~path~~ edge doesn't exist then transitive closure creates a ~~path~~ edge from i → j

Don't use ~~edge~~ Path word

Shortest path - D

weights on the edge

A subpath of shortest path is itself a shortest path

=> A → B → D is shortest

implies B → D is also shortest

There is a tree ~~from~~ of shortest paths from a vertex to all other vertexes

Choose vertex A
Adjacent vertex B C D put weights on verts
Add same smaller distance adjacent verts to tree

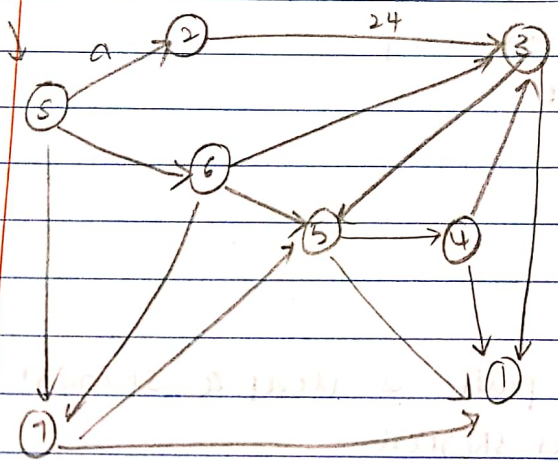
Edge Relaxation

Making priority Queue for each vertex $O(V)$
while Q not empty then percolate up/down $O(\log(edge))$
$\rightarrow O(edge \cdot \log V)$

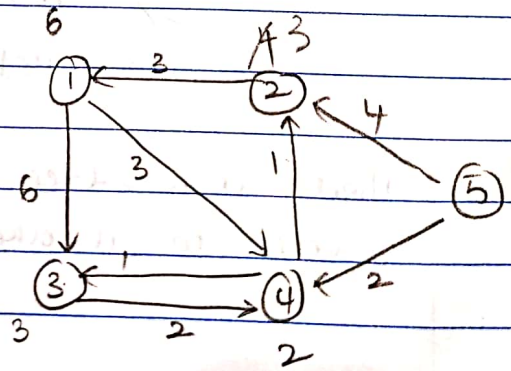
every v → percolate down
e → percolate up

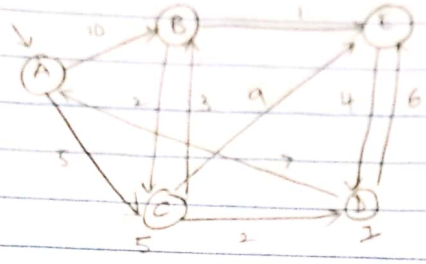
Total $O((V+e)\log v)$

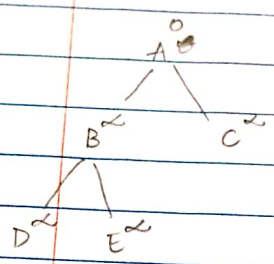
If graph is dense $n(n-1)$ edges
$\therefore O(n^2)$

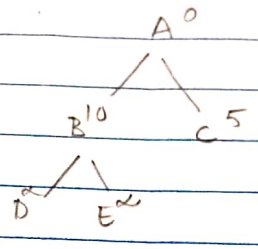


S
Adjacent → 2, 7
dequeue S





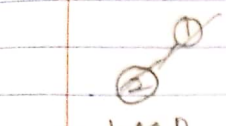




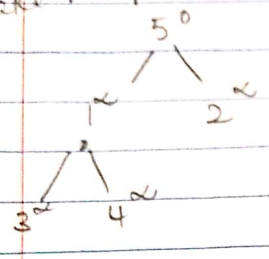
A C D B E

vertex selected	A	B	C	D	E
→	(0)	∝	∝	∝	∝
	0	10	(5)	∝	∝
	(0)	8	(5)	(7)	14
	(0)	(8)	(5)	(7)	13
	(0)	(8)	(5)	(7)	(9)

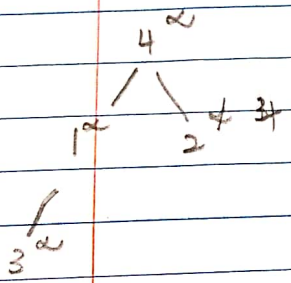
1	2	3	4	5 ↓
∝	∝	∝	∝	(0)
∝	4	∝	(2)	0
∝	3	(3)	2	0
∝	(3)	3	2	0
(6)	3	3	2	0



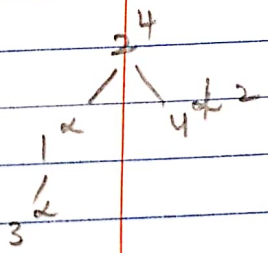
Make heap

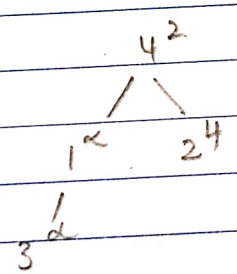


Deque 5^0

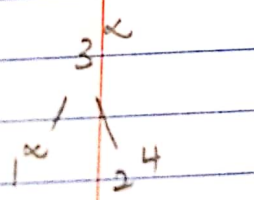


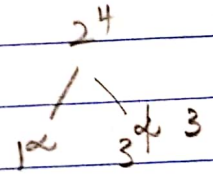
All edges add to 5



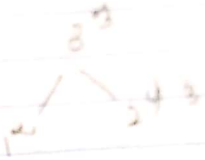


Deque 4^2

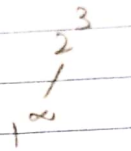




* Incident on 4^2



Deque 3^3



edges of 3^3

Nothing

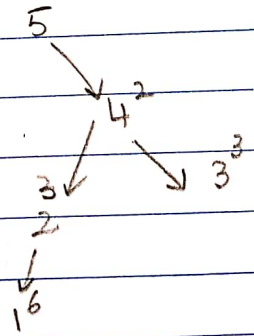
Deque 2^3

1^6

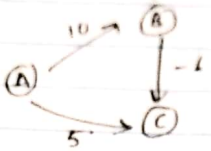
edges of 2^3

1^6

Deque 1^6



Dijkstra doesn't work for edges with negative weight



A → C = 5
A → B → C = 4 ... wrong

∴ Bellman ford Algorithm

distance →	d					
parent →	π					

Max Iterations = n-1
In one iteration if nothing changes, don't try others.

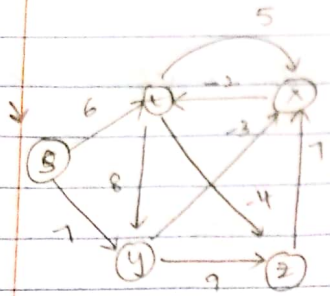
(1,3)
(1,4)

(5,2) ⇐ 5 is 0 2 is 0 to 4
(5,4) 5 is 0 4 is 2

① 6 ∝ 4 ∝ 2 0
/ 5 / 5 /

② 7 3 3 2 0
2 4 4 5 /

③



(s,t)	6		
(s,y)	7		
(t,x)	5		
(t,y)	8	(t,z)	-4
(x,t)	-2		
~~(y,z)~~	~~9~~		
(y,x)	-3	(y,z)	9
(z,x)	7		

①

s	t	x	y	z
0	∞	∞	∞	∞
/	/	/	/	/

②

s	t	x	y	z
0	6	∞	7	∞
/	s	/	s	/

③

s	t	x	y	z
0	6	11	7	2
/	s	t	s	t

④

s	t	x	y	z
0	6	4	7	2
/	s	y	s	t

⑤

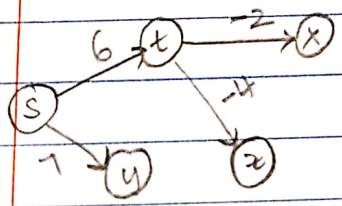
s	t	x	y	z

I1

0	2	4	7	2
/	x	y	s	t

I2

0	2	4	7	-2
/	x	y	s	t



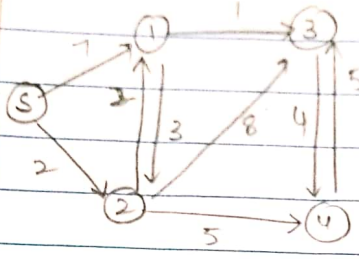
What if there is change in (n-1) iteration

goto nth iteration

if again change is present

then there is negative cycle.

⇒ No shortest path



(S,1)	7
(S,2)	2
(1,3)	1
(1,2)	3
(2,3)	2
(2,3)	8
(2,4)	5
(3,4)	4
(4,3)	5

I①

S	1	2	3	4
0	∞	∞	∞	∞
/	/	/	/	/

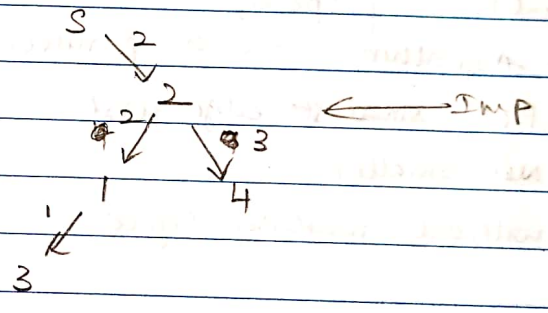
I1

0	≠4	2	8	7
/	⊘②	S	①	②

I2

0	4	2	5	7
/	②	S	①	②

I3 No change

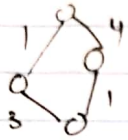


Negative edges make sense only in directed graph

Minimum spanning Tree

Just want connectivity

No need origin



No shortest distance

MST

Both are different in this example.

cycle Property

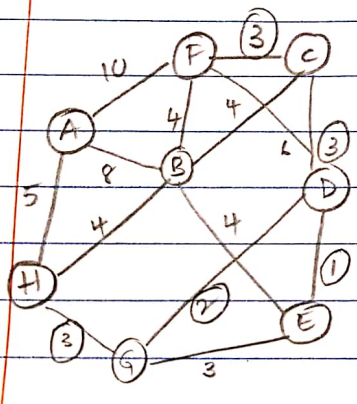
Partition property

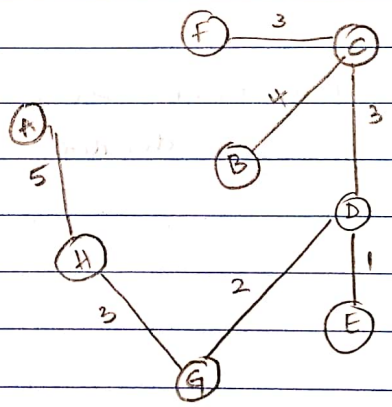
Kruskals Algorithm — uses partition property

Pick smallest edge first

Next smaller

without making cycle





Sort

(D,E)	1	✓
(D,G)	2	✓
(E,G)	3	✓
(C,D)	3	✗

Making PQ O(E)

pick 1 & percolate down O(log E)

worst case all edges gets picked

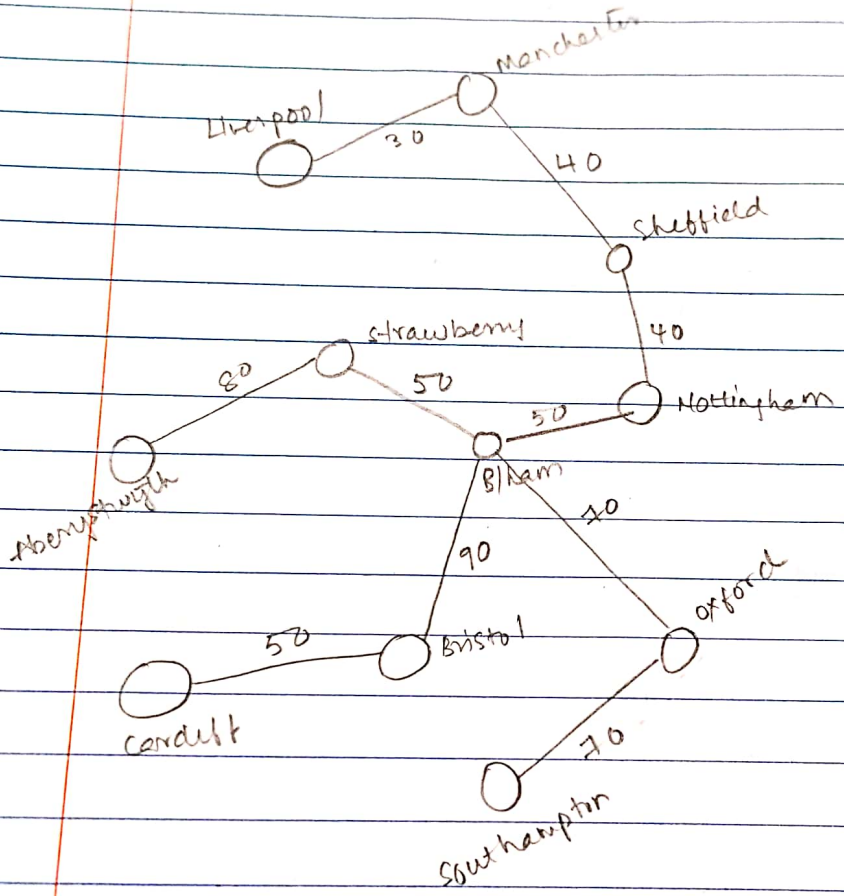
∴ O(E + E log E)

Algo –

Sort edges by increasing edge weight – $O(E \log n)$

Select first $|V|-1$ edges that don't form cycle

Two diff → Making Heap, Heap Sort

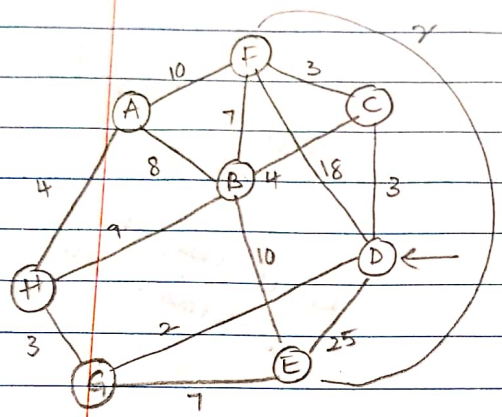


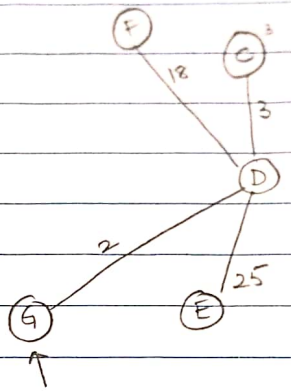
Pick an Edge | Reject

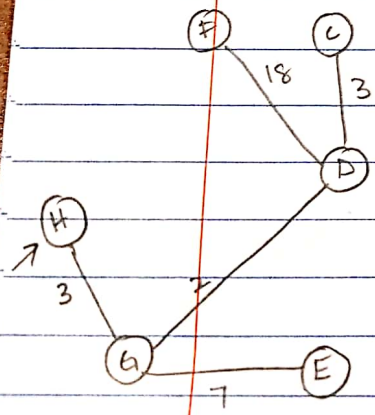
Reject only if it forms a cycle

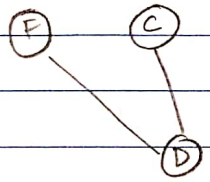
Prims Algorithm - uses cycle property
Here we have a start point

for this pd of vertices
If less ~~more~~ vertices → dense → Prims
less edges → sparse → kruskals is better





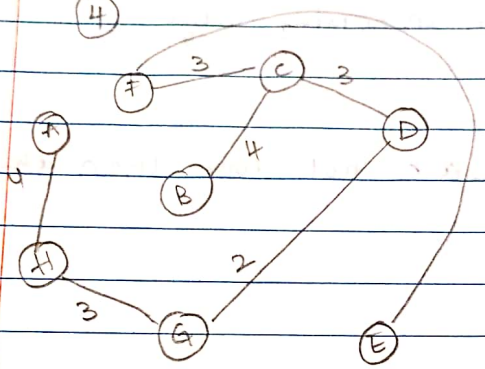




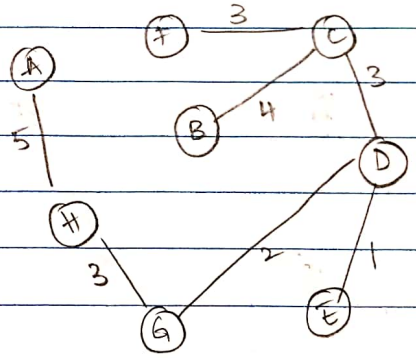
5 (4) - - -

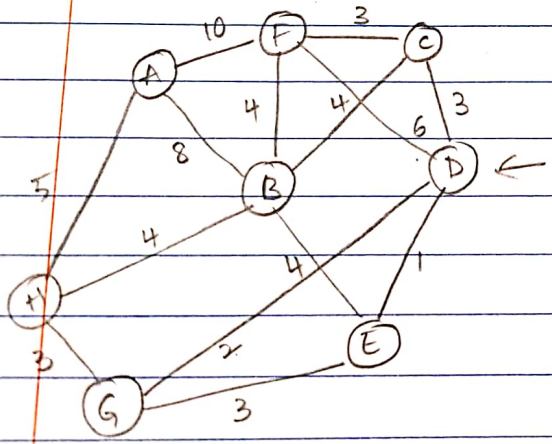
(5)

A	B	C	D	E	F	G	H
∞	∞	∞	(0)	∞	∞	∞	∞
∞	∞	3	0	23	18	(2)	∞
∞	∞	(3)		7	18		3
∞	4			7	(3)		3
10	4			(2)			3
10	4						(3)
4	(4)						
(4)							



Draw final tree always





A	B	C	D	E	F	G	H
∞	∞	∞	0	∞	∞	∞	∞
∞	∞	3	-	(1)	6	2	∞
∞	4	3	-	-	6	(2)	∞
∞	4	3	-	-	6	-	(3)
5	4	(3)	-	-	6	-	-
5	4	-	-	-	(3)	-	-

Constraint MST

DAG and topological ordering

Topological can be done only one DAG

Directed acyclic graph → directed graph with no cycles.

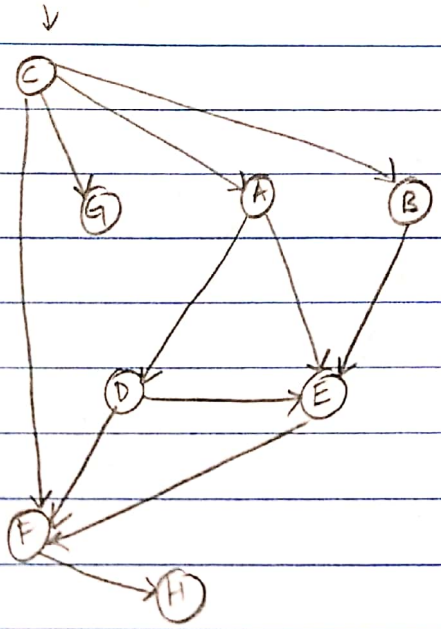
Atleast one vertex with no incoming edges

Starting point wont be given

Pick vertex that donot have incoming edge

Two flags visited done

If a vertex is visited & not done then there is a cycle.



Dfs(C)

Dfs(A)

DFS traversal | DFS topological sorting

Diff | This can be done only on Directed Acyclic graph

Visited	Done
A	
E	
D	
F	
L	L
K	
J	
G	

DFS(A) visited[A] = true

for each adj vertex to A → v

if visited[v] is false

DFS[v]

DFS(A)

DFS(E)

DFS[D]

DFS(L) / DFS(F)

DFS(K) DFS(L)

DFS(J) No adj to L

DFS(G) Done ▲ L

Modification

if visited[v] == true && done[v] == false

then a cycle.

BFS  this and this and this

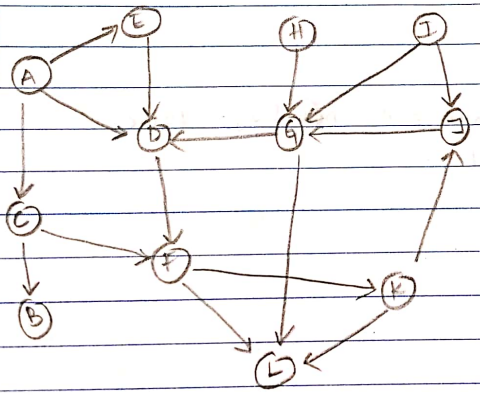
DFS  this or this or this

BFS Topological Ordering

Have an array of indegrees

Add to queue the vertices with 0 in indegree

When we queue the indegree of other vertices may change

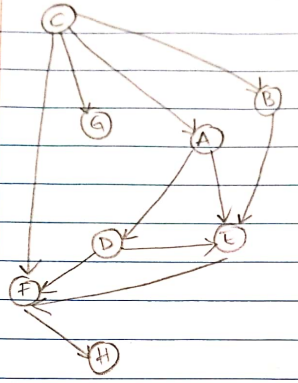


A cycle

Incoming	A	B	C	D	E	F	G	H	I	J	K	L
	0	~~1~~ 0	~~1~~ 0	~~3~~ ~~2~~ ~~1~~ 0	~~1~~ 0	~~2~~ ~~1~~ 1	~~3~~ ~~2~~ ~~1~~ 0	0	0	~~2~~ ~~1~~ 1	1	3

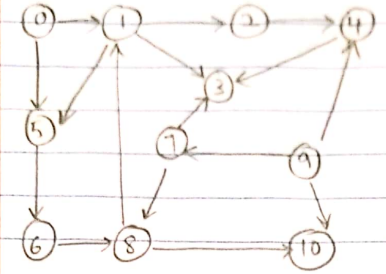
Out ordering A H I C E B

Queue ~~A~~ ~~H~~ ~~I~~ ~~C~~ ~~E~~ ~~B~~



	A	B	C	D	E	F	G	H
Incoming	~~1~~ 0	~~1~~ 0	0	~~1~~ 0	~~2~~ ~~1~~ 0	~~2~~ ~~1~~ 0	~~1~~ 0	~~1~~ 0
TO	C	A	B	G	D	E	F	H

Queue ~~C~~ ~~A~~ ~~B~~ ~~G~~ ~~D~~ ~~E~~ ~~F~~ ~~H~~



Indegree	10	⓪	1	2	3	4	5	6	⑦	8	⑨
	✗	0	✗	1	✗	✗	✗	1	✗	✗	0
	1		1		2	1	1		0	1	

Queue ϕ 9 7

cycle

TO 0 9 7

Use adjacency matrix

looking at columns gives incoming edges

rows gives outgoing edges.

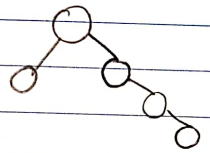
In BST → worst case search O(n)
To get $\log n$ we did balancing AVL trees
One more Algo is Red Black tree

Balancing condition for red black is more relaxed
Ratio 1:2
More balances are required in AVL

Red → imbalance
Every node is red / black
root is always black
We need null leaf nodes too
null leaf nodes are always black.
If a node is red, then both its children are black
No two consecutive nodes can be red.
No. of black nodes along any path must be same

worst case

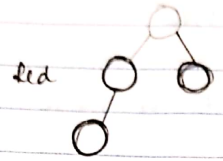
left → all black
right → alternate red black then 1:2



2 4

Insertion

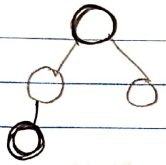
① Find place through BST Algo



Look at uncle

Make parent, uncle → black

And grand parent red



Case-1 → Inserted node always red

Look at parent if black nothing

If parent is red

Look at uncle → red

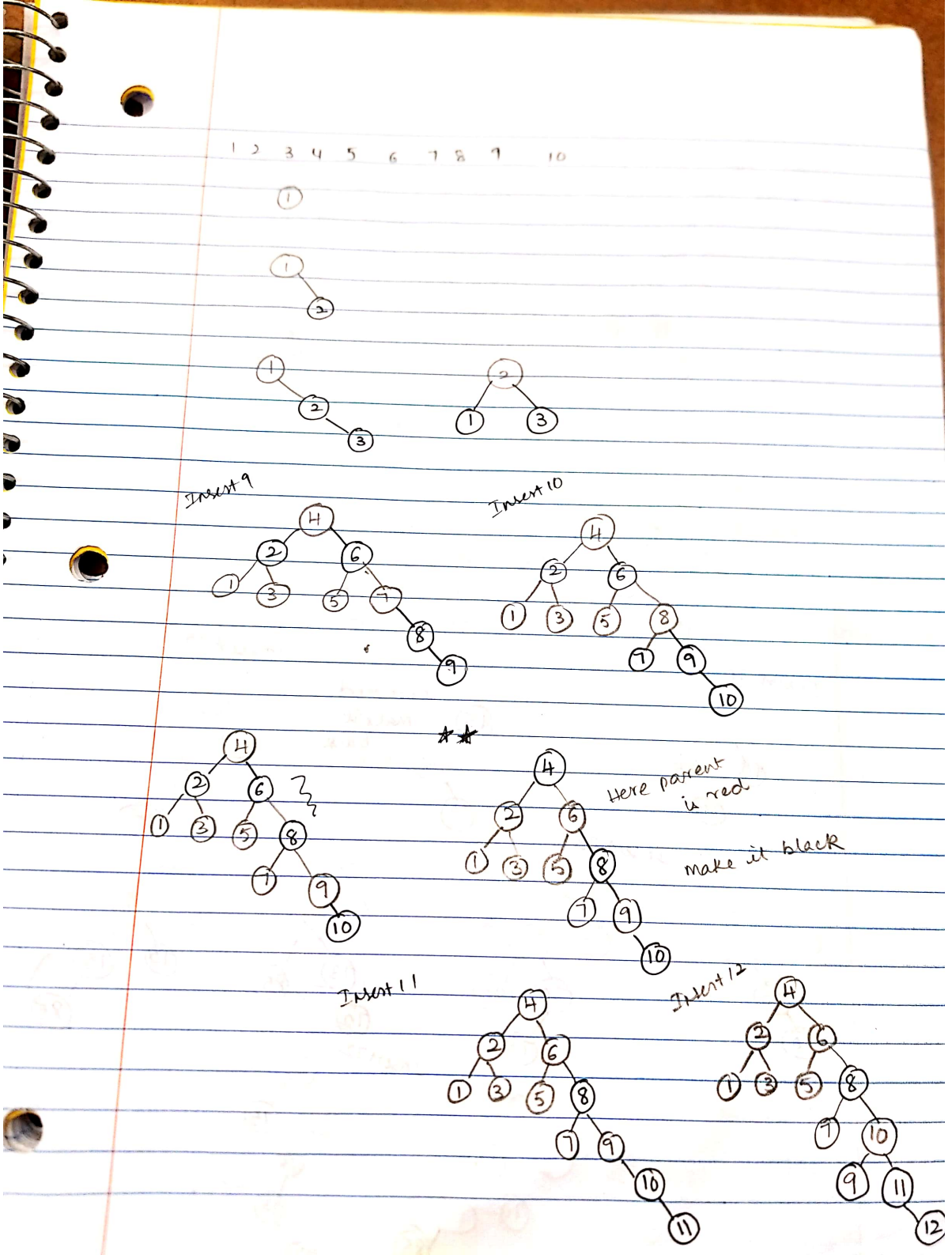
Make parent, uncle black and grandparent red.

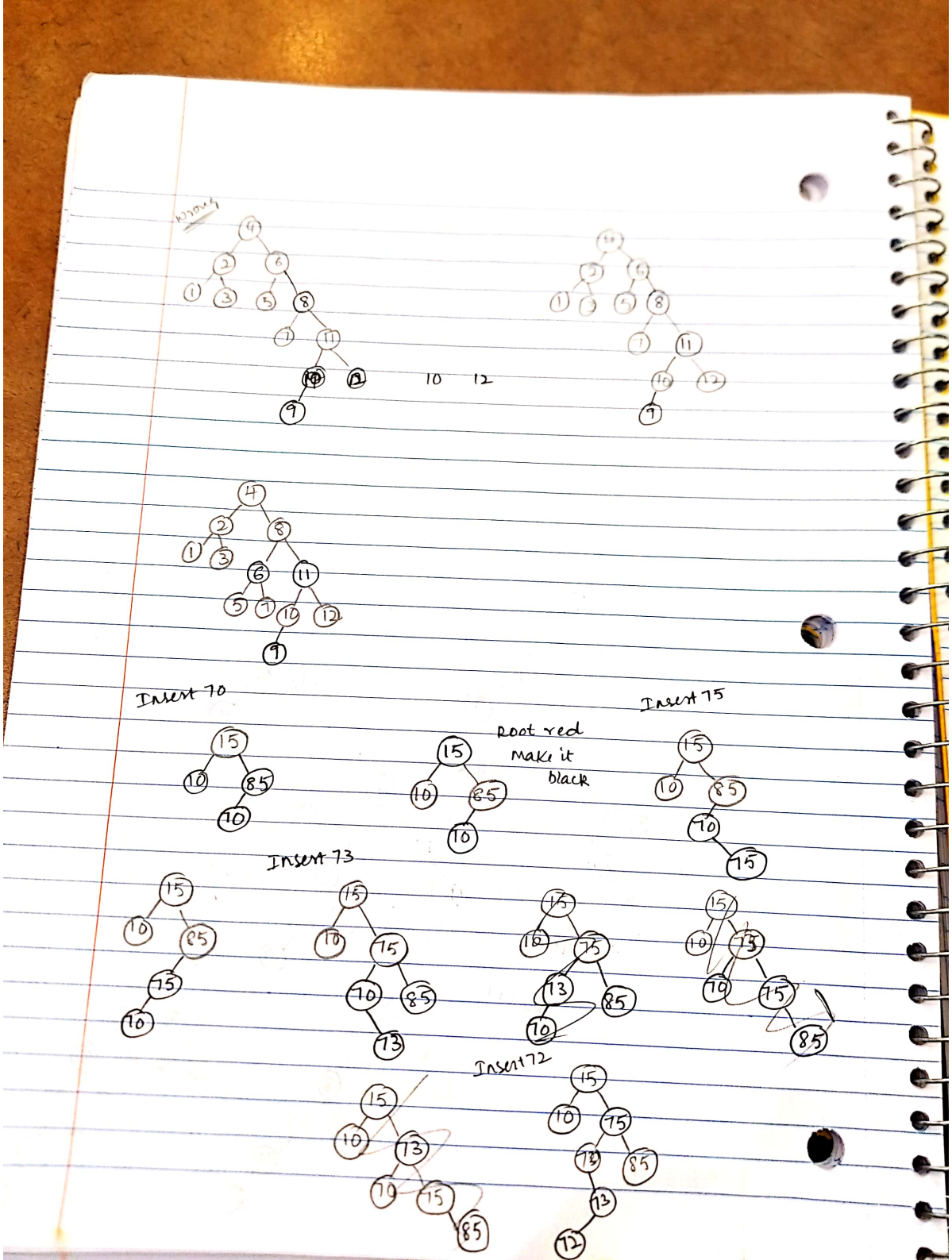
uncle → black

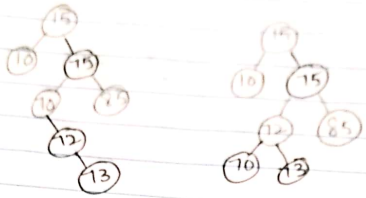
Do rotation

grand parent has to be black

push a red to other side







Delete

Do as in BST

Color of node that physically went away is red. No prob

If physically black node went away.

Find red node on other side and make it black here.

Look at nephew atleast one is red

If red do rotation

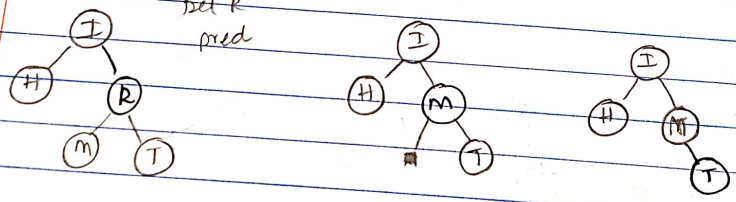
If nephew is black and sibling also black => no red on other side

Make sibling red, parent black. move dot to parent

If sibling is red

Do rotation & reapply the rules.

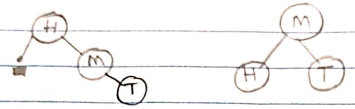
Keep moving up and check



Del 1 use pred

Del 2 use pred

Del leaf with pred



Delete do BST

physically deleted node black put dot.

Look at siblings and nephew

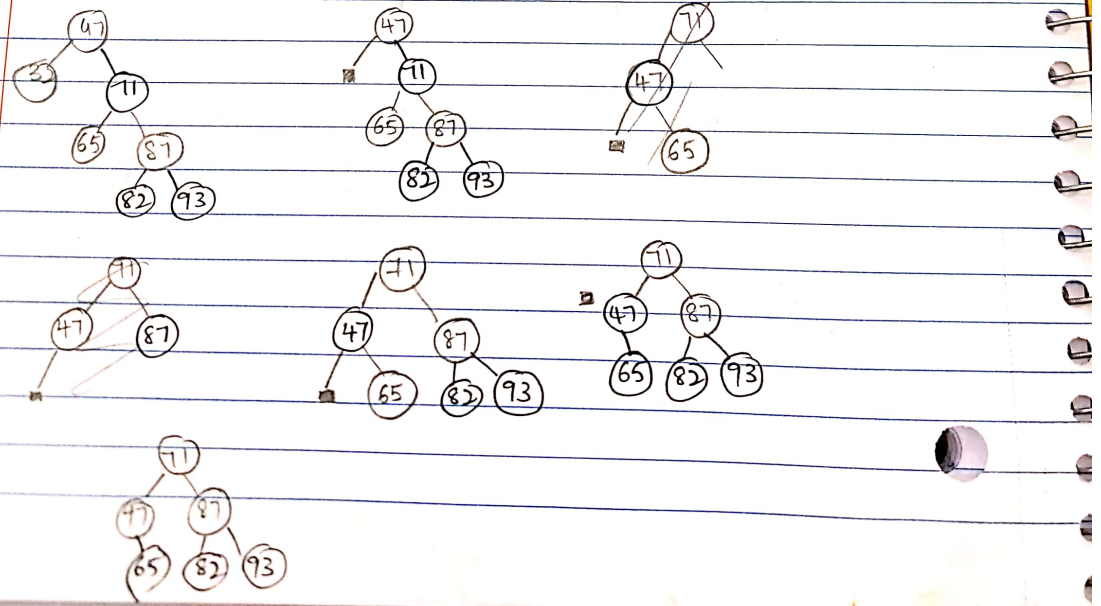
If no red make sibling red and move dot up.

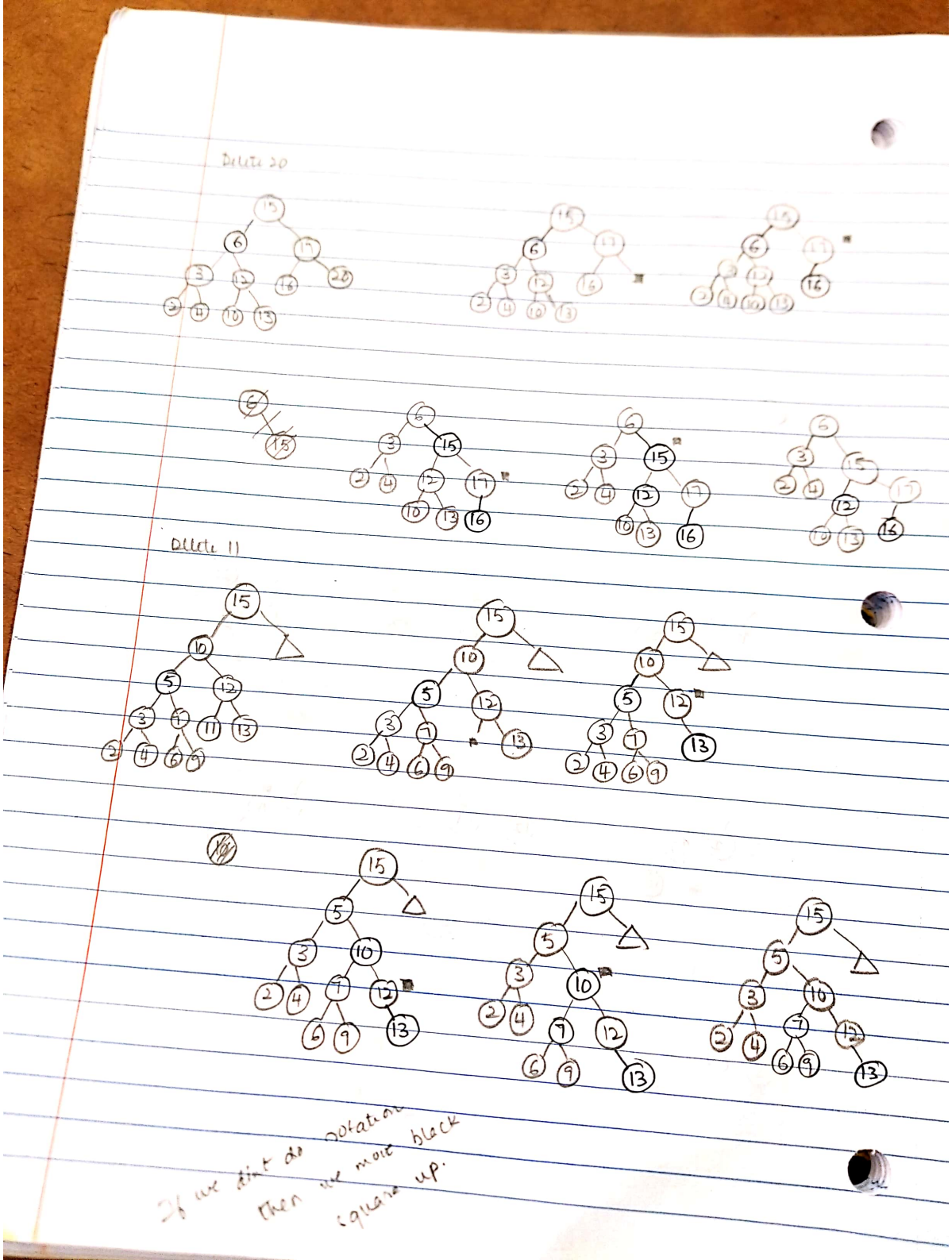
If nephews are red → rotate (normal)

If sibling is red → nephews are black when rotate one nephew goes to other side

Delete 32

Rotate





Hashing

Store and get in O(1) Insert, delete, find

Items are stored in Hash tables.

* Hash function gives values → that value needs to be done modulus with array size

Table size preferred to be prime number.

Collision Resolution

Separate chaining

open Addressing – Linear probing
Quadratic probing
Double Hashing.

Separate chaining → lists of linked lists → Easy to delete

Load factor λ = $\frac{\text{No. of elements stored in Hash table}}{\text{Table size}}$.

Linear probing → If collision occurs, store it in next available index.

Here deletion is done using flag. Flag say whether the value is deleted or is present. This is called lazy deletion.

Quadratic probing – $h+1^2$, $h+2^2$, $h+3^2$

Find next square from previous square

$H_i = H_{i-1} + 2i - 1$

* → load factor becomes high → Always double to next prime the values need to be hashed from beginning again.

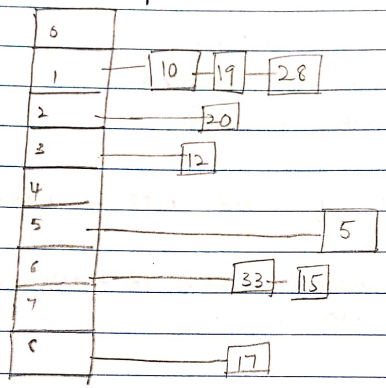
Double hashing → on collision use a second hash function.
If second hash function gives collision
then use $i * hash_2(x)$

→ Table size = 9

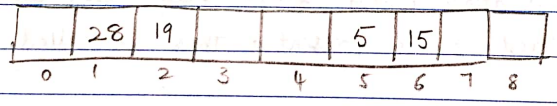
5, 28, 19, 15, 20, 33, 12, 17, 10

h(Key) = Key

Separate chaining



Linear probing



$\frac{5}{9} > 0.5$

Double size → 19 (doubles to next prime)

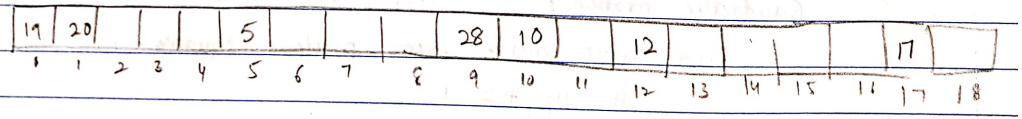


Table size = 15

Insert 16, 7, 28, 31, 67, 38, 29, 73, 99, 43, 218

Quadratic Probe

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
29	16	31					7	67	38				28	

i^2

Load factor > 0.5

Next size 31

mod 31

0	31
1	
5	67 — 6 99
7	7
8	38 — 11 73
16	16 — 12 43
28	28
29	29
30	

Double Hash %13+1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	16		67			31	7	38	73			99	28	29

31%13+1

After double Hash also collision

do i * hashkey

hash(2) second collision value

99%15 → 9

99%13+1 → 9

2×9%15 → 3 filled

3×9%15 → 12

Radix sort

O(n)

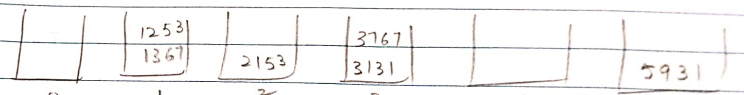
Radix → no. of unique keys present in the items that we wanted to sort.

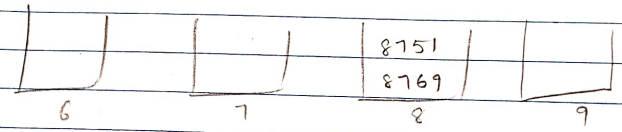
Two types MSD Radix sort
LSD Radix sort

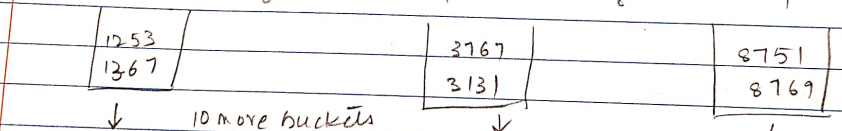
sorted
1253
1367
2153
3131
3767
5931
8751
8769

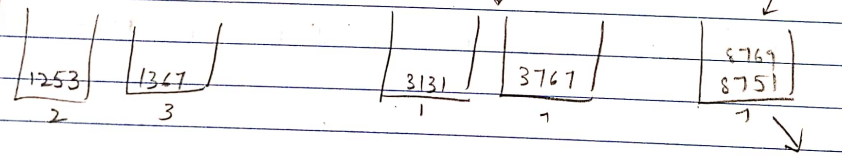
5931
1367
2153
8769
3131
1253
8731
3767

All digits Radix → 10
MSD is 1st digit









For 4 digit nos max buckets used → 40.

Also (No. of digits × n) O(4×8)
Space (No. of digits × Radix) O(4×10)

10,000 nos or more Radix sort helps

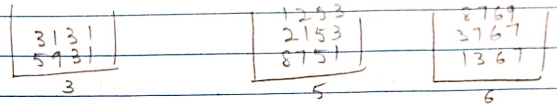
LSD Radix sort

Pass 1

FIFO (order map)

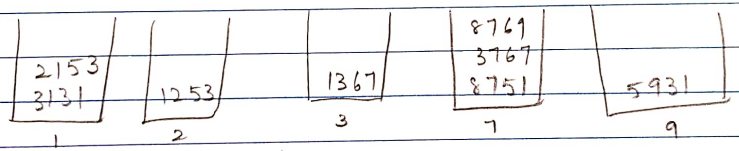
5931 3131 8751 2153 1253 1367 3767 8769

take same bucket look 2nd

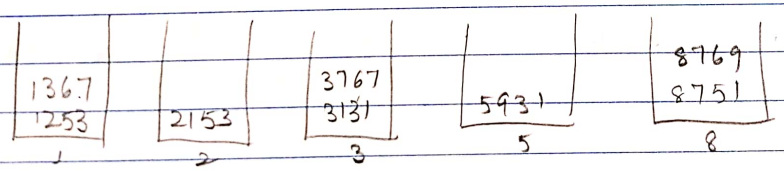


After each pass we have result → Therefore it is stable sort.

FIFO 5931 3131 8751 2153 1253 1367 3767 8769

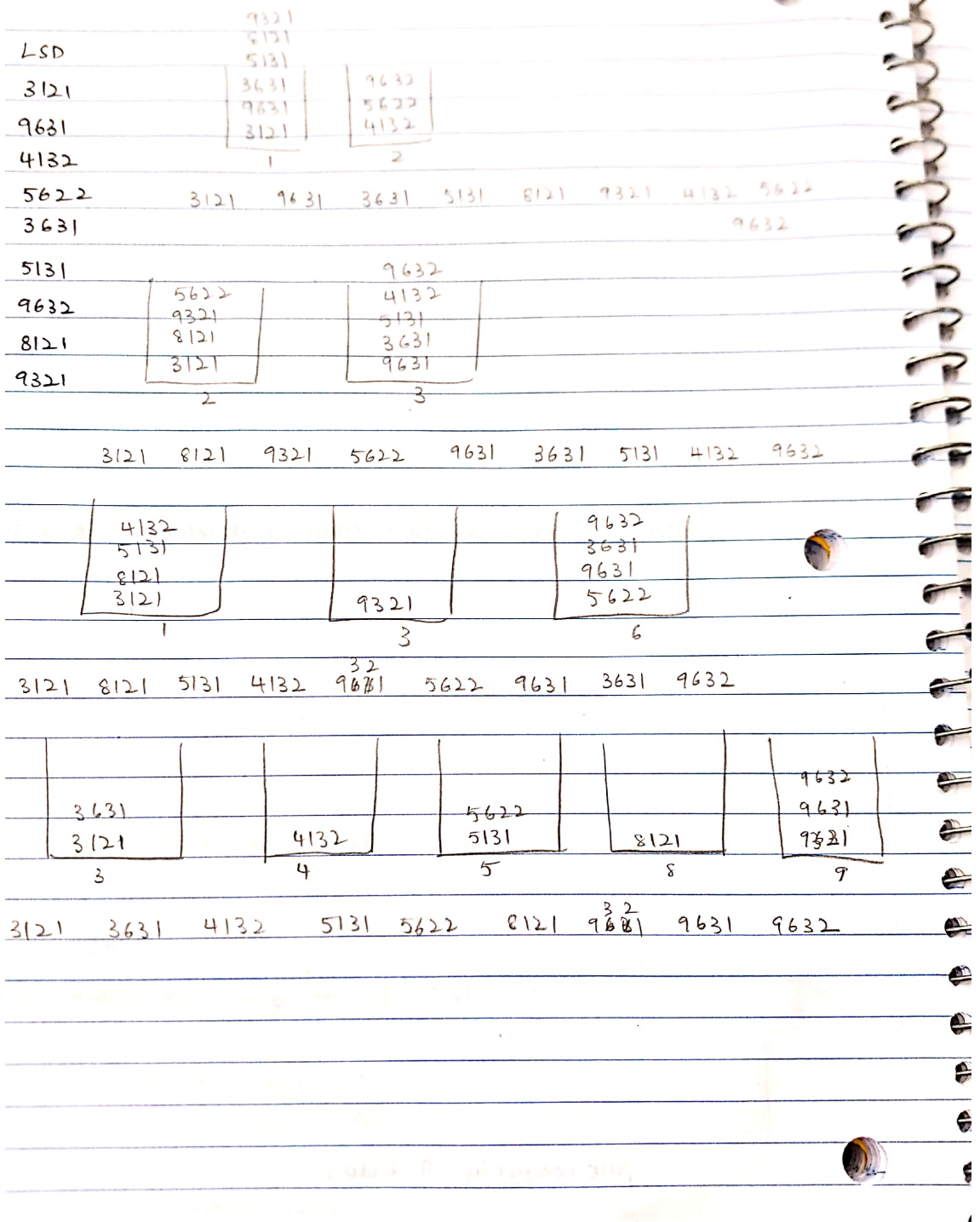


FIFO 3131 2153 1253 1367 8751 3767 8769 5931



1253 1367 2153 3131 3767 5931 8751 8769

space complexity O(Radix)



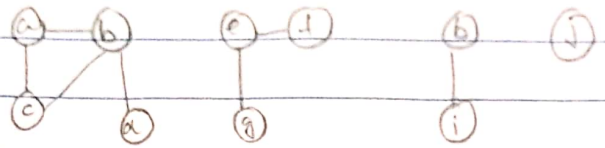
Set operations

Union

Intersection

partition → Union of all subsets is the set itself

check no of connected components of a graph



{a} {b} {c} {d} {e} {f} {g} {h} {i} {j}

{b,d}

{e,g}

{a,c}

{h,i}

{a,b,c,d}

{e,f,g}

{a,b,c,d} {e,f,g} {h,i} {j}

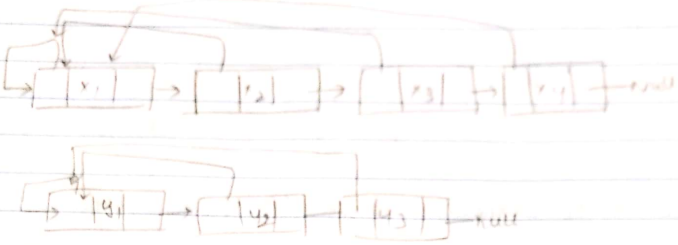
4 components

Similar to Kruskal's Algorithm

Each set has an id

Linked list representation

data



way ① union change x_1.id to y_1 — O(1)
Find O(n)

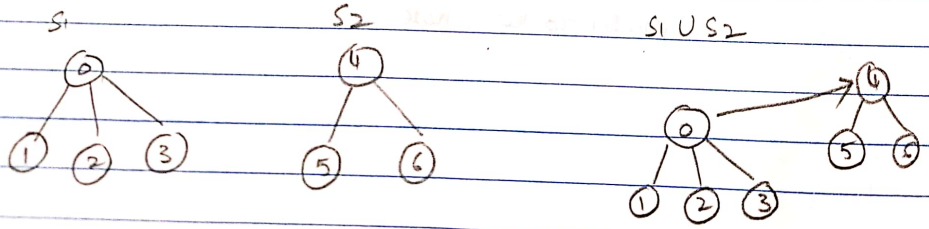
way ② change x_1.id to y_1
x_2.id to y_1
x_3.id to y_1
x_4.id to y_1 Union takes O(n)
Find O(1)

Selection depends on the application.

Weighted union → let smaller set join the larger set. using way (2)
O(n log n)

Tree Representation of set :

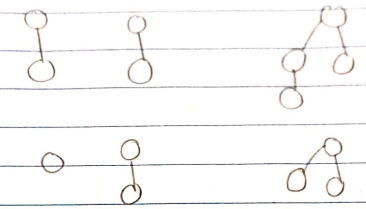
Root of a tree is setId.



Here height matters when union

General union → O(n log n)

Height increases only when we do union of equal height



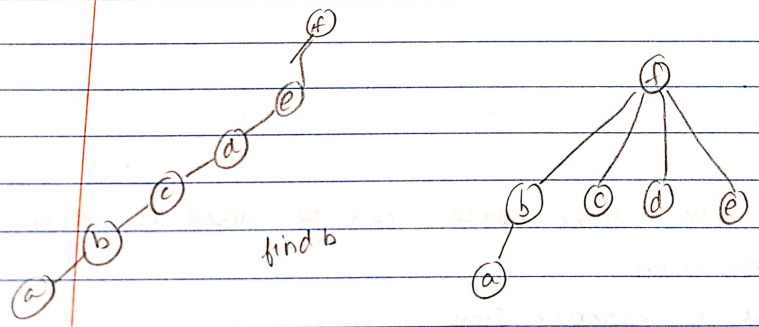
Union by weight (nodes count) → Find O(1)

Union by rank (height) → Find O(log n)

Path compression

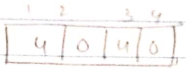
First find(5) traverse and change pointers

For next find traverse is not required.



Array representation of set (from trees)

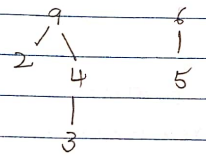
0 is an index Index & element both same → root



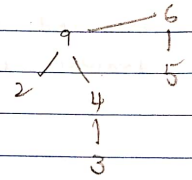
No 0 index value 0 represents root

way 1 Union check if different components
change 1 to other

way 1 Make one root point to other



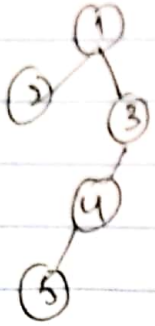
Union of 4 and 5



Weighted union

Instead of zero same space can be used to store height, nodes count.
Put in negative sign

Union → change root of smaller set
increasment the negative count of longer set



1	2	3	4	5
-3	1	1	3	4

Find 5 using PC

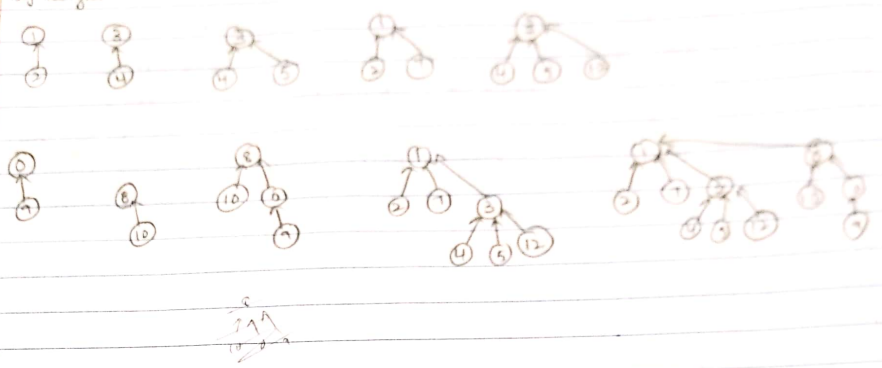
1	2	3	4	5
-5	1	1	1	1

→ u(1,2) u(3,4) u(3,5) u(1,7) u(3,12) u(0,9) u(8,10)
u(8,9) u(7,4) u(2,9)

a) by size

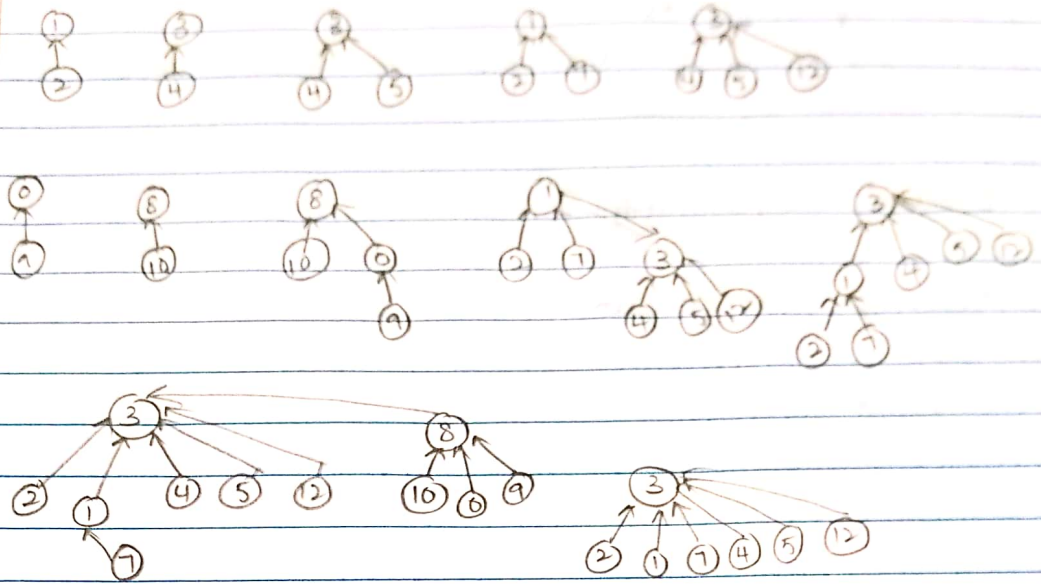
0	1	2	3	4	5	6	7	8	9	10	12
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	
-1	-2	1	-1	-1	-1	-1	-1	-1	-1	-1	
-1	-2	1	-2	3	-1	-1	-1	-1	-1	-1	
-1	-2	1	-3	3	3	-1	-1	-1	-1	-1	
-1	-3	1	-3	3	3	-1	1	-1	-1	-1	
-1	-3	1	-4	3	3	-1	1	-1	-1	-1	3
-2	-3	1	-4	3	3	-1	1	-1	0	-1	3
-2	-3	1	-4	3	3	-1	1	-2	0	8	3

By height



	0	1	2	3	4	5	6	7	8	9	10	12
	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
u(1,2)	-1	-2	1	-1	-1	-1	-1	-1	-1	-1	-1	-1
u(3,4)	-1	-2	1	-2	3	-1	-1	-1	-1	-1	-1	-1
u(3,5)	-1	-2	1	-2	3	3	-1	-1	-1	-1	-1	-1
u(1,7)	-1	-2	1	-2	3	3	-1	1	-1	-1	-1	-1
u(3,12)	-1	-2	1	-2	3	3	-1	1	-1	-1	-1	3
u(0,9)	-2	-2	1	-2	3	3	-1	1	-1	0	-1	3
u(8,10)	-2	-2	1	-2	3	3	-1	1	-2	0	8	3
u(8,9)	8	-2	1	-2	3	3	-1	1	-3	0	8	3
u(7,4)	8	-3	1	1	3	3	-1	1	-3	0	8	3
u(2,9)	8	-4	1	1	3	3	-1	1	1	0	8	3
u(2,12)	Same set No union											

PC By size



	0	1	2	3	4	5	6	7	8	9	10	12
	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
u(1,2)	-1	-2	1	-1	-1	-1	-1	-1	-1	-1	-1	-1
u(3,4)	-1	-2	1	-2	3	-1	-1	-1	-1	-1	-1	-1
u(3,5)	-1	-2	1	-3	3	3	-1	-1	-1	-1	-1	-1
u(1,7)	-1	-3	1	-3	3	3	-1	1	-1	-1	-1	-1
u(3,12)	0 -1	-3	1	-4	3	3	-1	1	-1	-1	-1	3
u(0,9)	-2	-3	1	-4	3	3	-1	1	-1	0	-1	3
u(8,10)	-2	-3	1	-4	3	3	-1	1	-2	0	8	3
u(8,9)	8	-3	1	-4	3	3	-1	1	-4	0	8	3
u(7,4)	8	3	1	-7	3	3	-1	1	-4	0	8	3
u(2,9) PC	8	3	3	-7	3	3	-1	1	-4	8	8	3
	8	3	3	-11	3	3	-1	1	3	8	8	3
u(7,12) PC								3				

no union same set