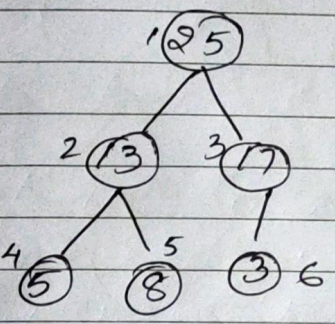
Heap Data Structure : Heap ds is a complete binary tree that satisfy heap property. It is also called binary heap.

- Binary heap :- A data structure in which each node of the binary tree corresponds to an element of the array. The array is completely filled on all levels except possibly lowest.

We represent heaps in level order, going from left to right.



1	2	3	4	5	6	
25	13	17	5	8	3	X

= A

heap size = length [A]
= No. of elements in [A]

- Root of tree A[1] . i = indices of node ~~parent(i)~~)

↳ Parent (i)
return ($i/2$)

↳ left child ie left(i)
return (2i)

↳ Right child ie: right(i)
return (2i+1)

eg. index $i = 2$

$p = 2/2 = 1$

left $= 2i$

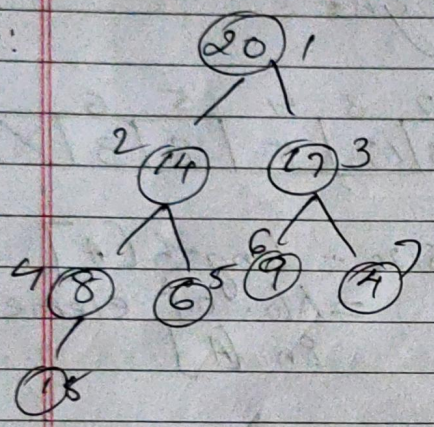
$= 2 \times 2 = 4$

right $2i + 1$

$= 2 \times 2 + 1$

$= 5$

eg:



Here index 20

$i = 1$

left $= 2i$

$= 2 \times 1 = 2$

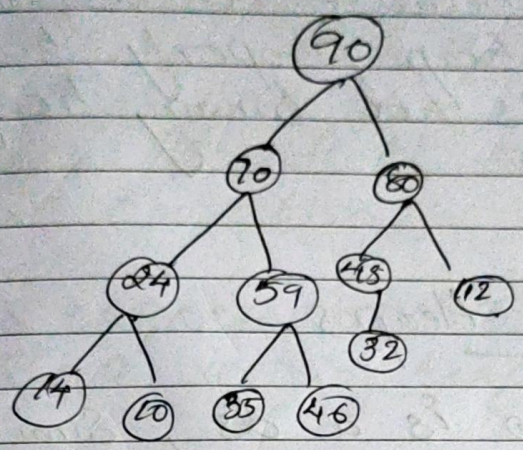
Right $= 2i + 1$

$= 2 \times 1 + 1$

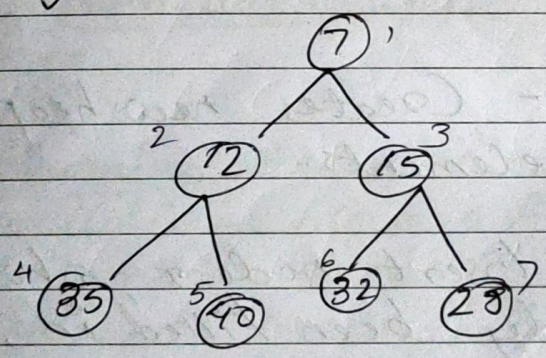
$= 3$

Heap property

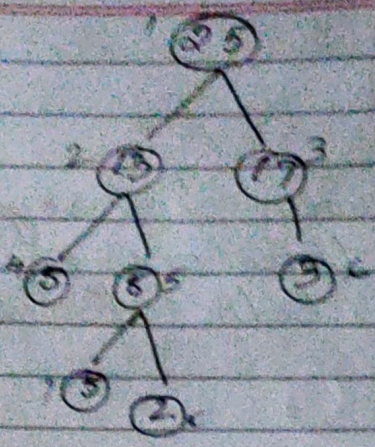
1. MAXHEAP - The key value in any node N is greater than or equal to the keyvalues in both of its children.



2. MIN HEAP - The key value in any node N is less than or equal to key value in both of its children



Ex:



This is not a heap because it satisfies heap property but it is not binary heap

Binomial Heaps

A binomial heap is a ds, similar to binary heap but also supporting the operation of merging two heaps quickly.

* Mergeable heap is a ds that supports five operations

1. MAKE-HEAP() - Creates new heap containing no elements.

2. INSERT (H, x) - Insert node x whose key field has already been filled in, into heap H.

3. MINIMUM (H) - Returns a pointer to the node in heap H whose key value

is minimum.

4. EXTRACT - MIN (H) → deletes the node from heap H, whose key is minimum, returning a pointer to node.

5. UNION (H_1, H_2) → Creates and returns a new heap that contains all the nodes of heaps H_1 & H_2.

Heaps H_1 and H_2 are destroyed by this operation.

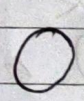
6. DECREASE-KEY (H, x, k) → assigns to node x within heap H, the new key value k,

7. Delete (H, x) → Delete node x from the heap H.

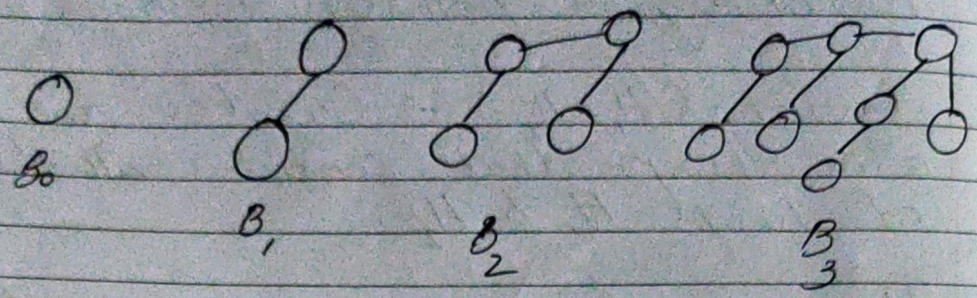
* Binomial heap : The Collection of binomial tree.

* Binomial tree : Binomial tree is the building block of binomial heap H. A Binomial tree B_k is an ordered tree defined recursively

* B_0 is binomial tree with single node



* Binomial tree B_k consists of two copies of B_{k-1} that linked together in such a way that root of the one is the leftmost child of the root of other.



* B_3 is based on combination of 2 B_2's
* B_2 is based on combination of 2 B_1's
* $B_0 \rightarrow$ has $2^0 = 1$ node — depth 0

  $B_1 \Rightarrow$ has $2^1 = 2$ node — depth 1

  $B_3 \rightarrow$ has $2^3 = 8$ node — depth 3

  $B_4 \rightarrow$ has $2^4 = 16$ node — depth 4

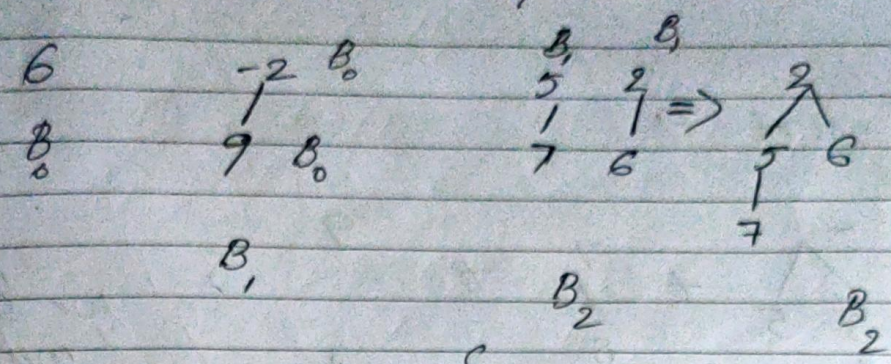
Properties of Binomial tree

1. There are 2^k nodes
2. Height ~~depth~~ ~~height~~ of the tree is k
3. There are exactly $({}^{k}C_i)$ nodes at depth

$i = 0, 1, 2 \cdots k$

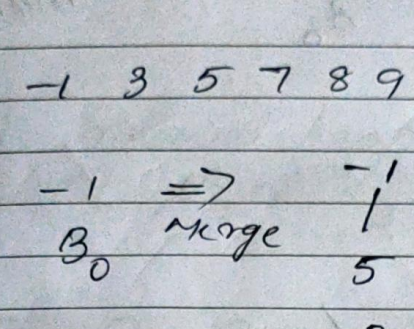
4. The root has degree k, which is greater than that of any other node, moreover if the children of the root are numbered from left to right by $k-1, k-2, \ldots 0$ child i is the root of subtree B_i

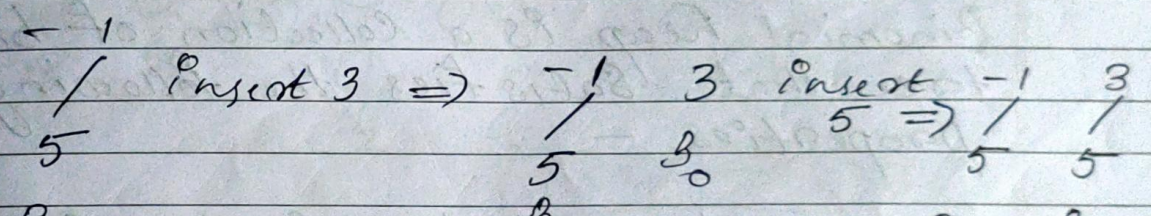
Eg:

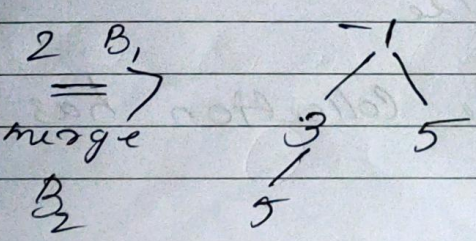


ie; 2 B_1 together to form B_2 with lower root
Here 2^k nodes
ie; $2^2 = 4$ nodes

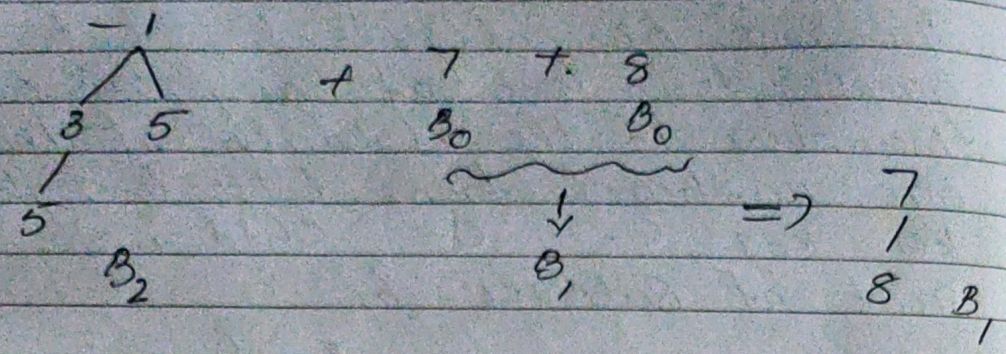
Ex: 5 -1 3 5 7 8 9



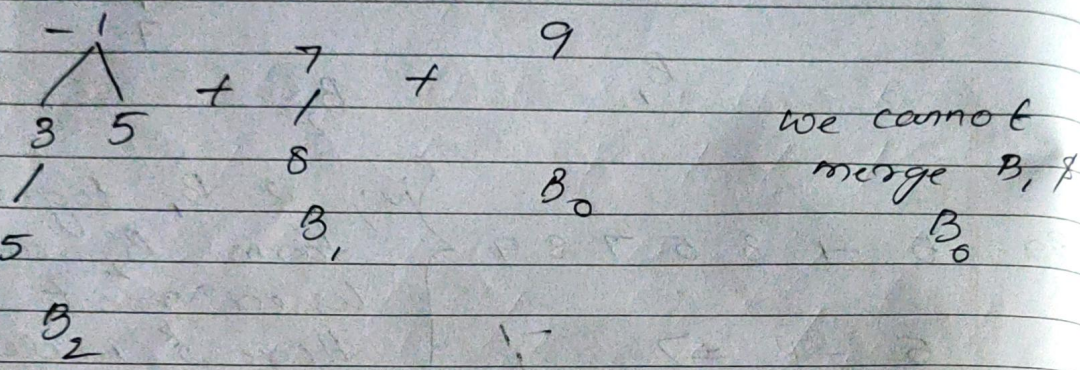




insert 7 insert 8



insert 9



Binomial heap

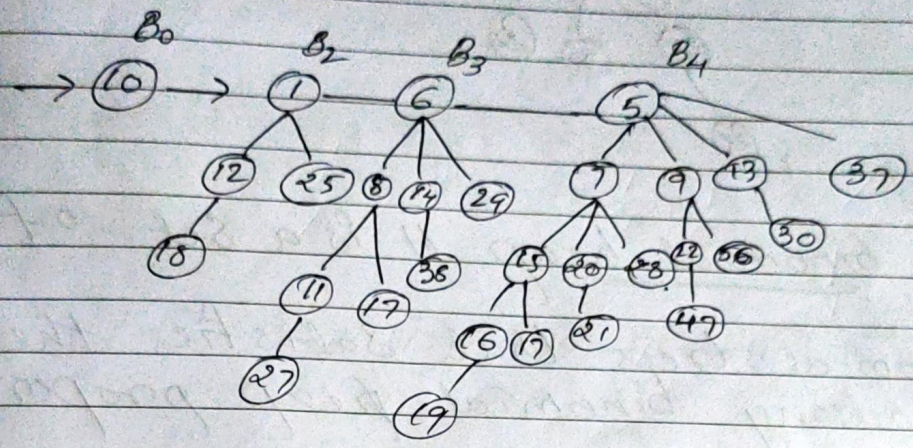
Binomial heap is a collection of binomial tree that satisfies the following properties: -

1. No two binomial tree in the collection have the same size

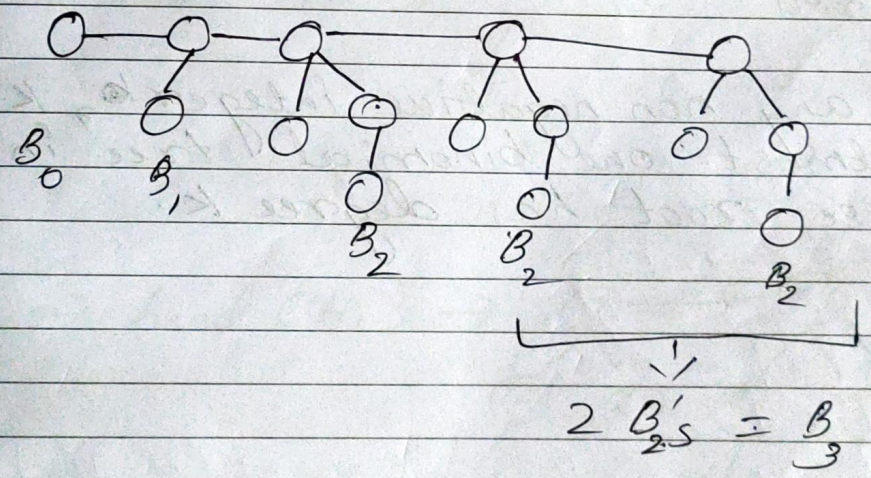
2. Each node in the collection has a key

* Each binomial tree in the collection satisfies the heap order property (max or min heap)

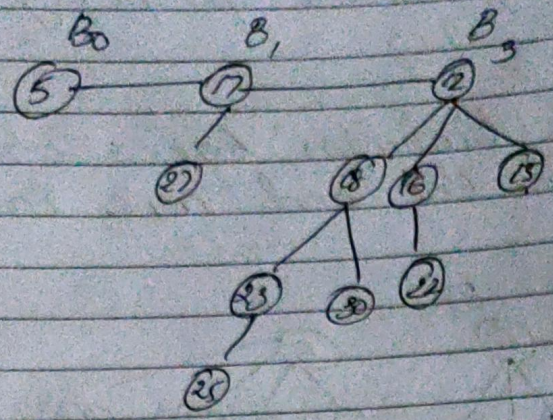
* Roots of the binomial trees are ~~Consi-dered~~ connected and are increasing order



* A binomial heap $n = 15$ nodes containing B_0, B_1, B_2 & B_3 binomial tree



Binomial Heap Example



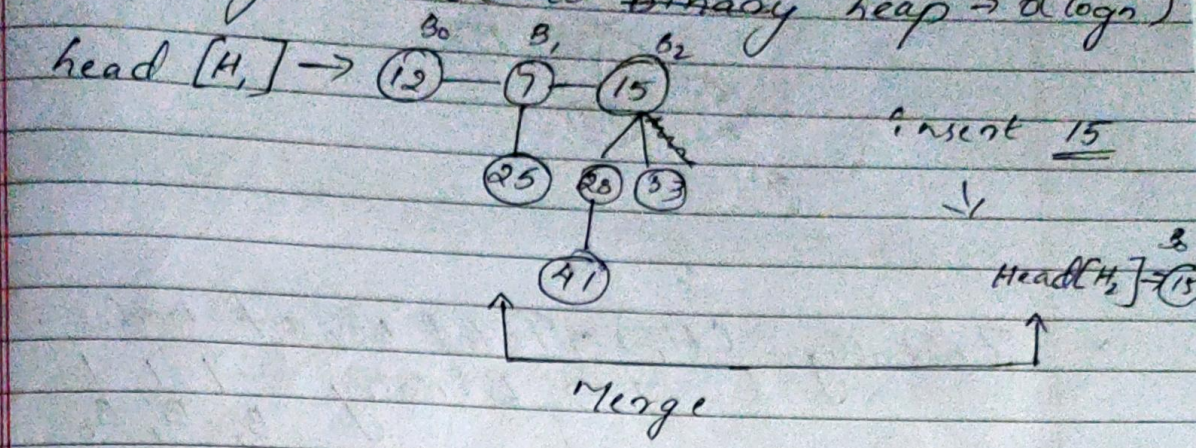
A <u>binomial heap</u> H is a set of binomial trees that satisfies the following binomial heap properties

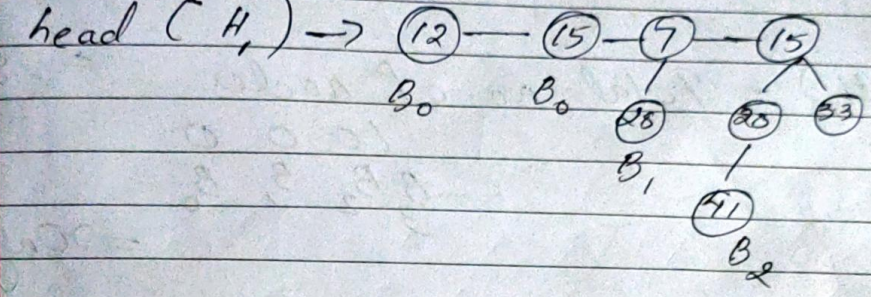
1. Each binomial tree in H obeys the MINHEAP property, ie, key of a node is greater than or equal to the key of its parent. ie, MINHEAP ordered.

2. For any non negative integer k, there is atmost one binomial tree in H whose root has degree k.

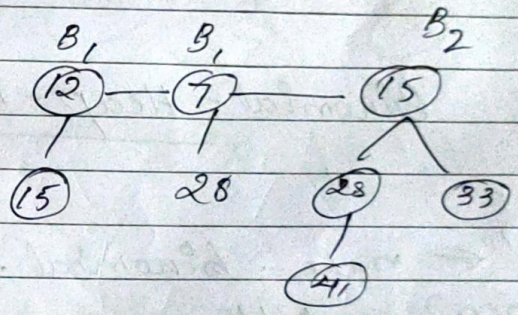
Binomial Operation

1. Inserting a node to ~~binary~~ binomial heap → $O(\log n)$

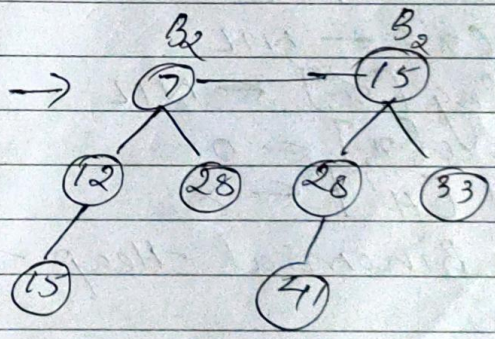




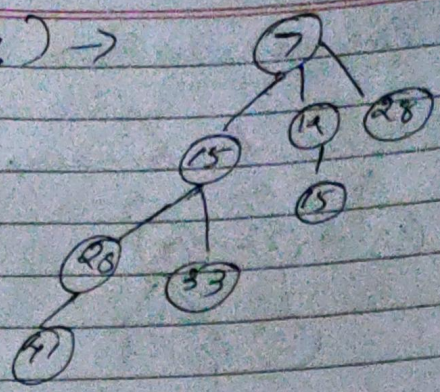
⇒ head (H1) →



⇒ head (H1) →



head $(H_1) \rightarrow$



* Initially (H_1) = Total no. of nodes = 7

$7 \rightarrow$ Binary $\underset{B_2}{1}\ \underset{B_1}{1}\ \underset{B_0}{1}$

head(H_2) = Total no. of nodes = 8

$\underset{B_3}{1}\ \underset{B_2}{0}\ \underset{B_1}{0}\ \underset{B_0}{0}$

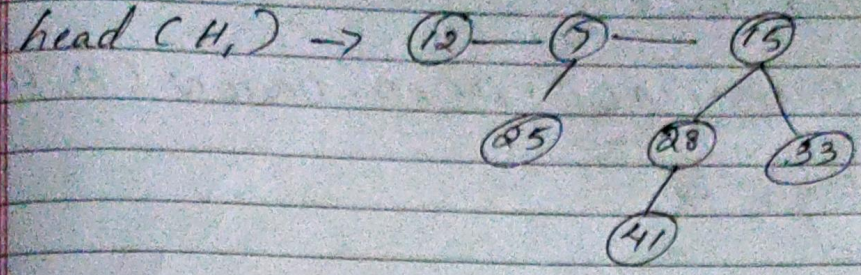
$\Rightarrow$ Only B_3

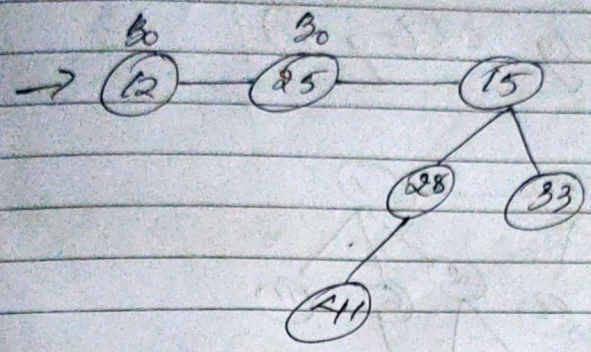
Algorithm

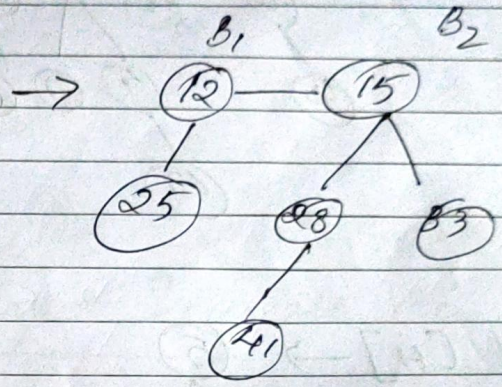
Binomial-Heap-insert (H, x)

1. $H' \leftarrow$ make-binomial-heap ()
2. $p(x) \leftarrow$ NIL
3. child$(x) \leftarrow$ NIL
4. Sibling $[x] \leftarrow$ NIL
5. degree$[x] \leftarrow 0$
6. head $[H'] \leftarrow x$
7. $H \leftarrow$ Binomial-Heap-UNION (H, H')

1) Extracting minimum key - O(log n)







Minimum key reside at root...

1. Find the root x with the minimum key in the rootlist of H and remove x from the rootlist of H. FIND-MIN-KEY()

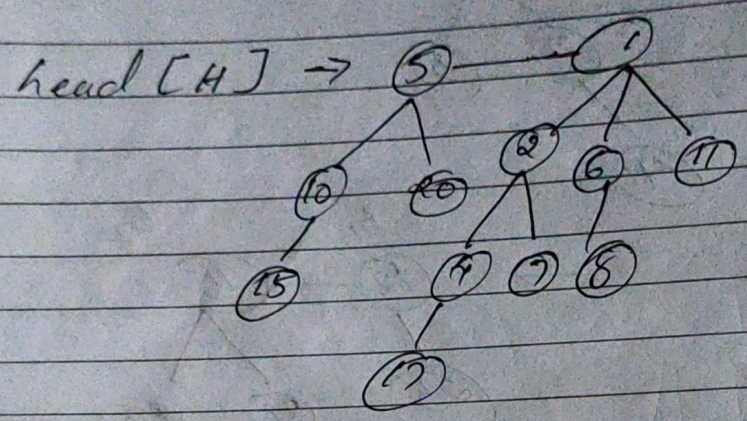
2. H' ← MAKE-BINOMIAL-HEAP()

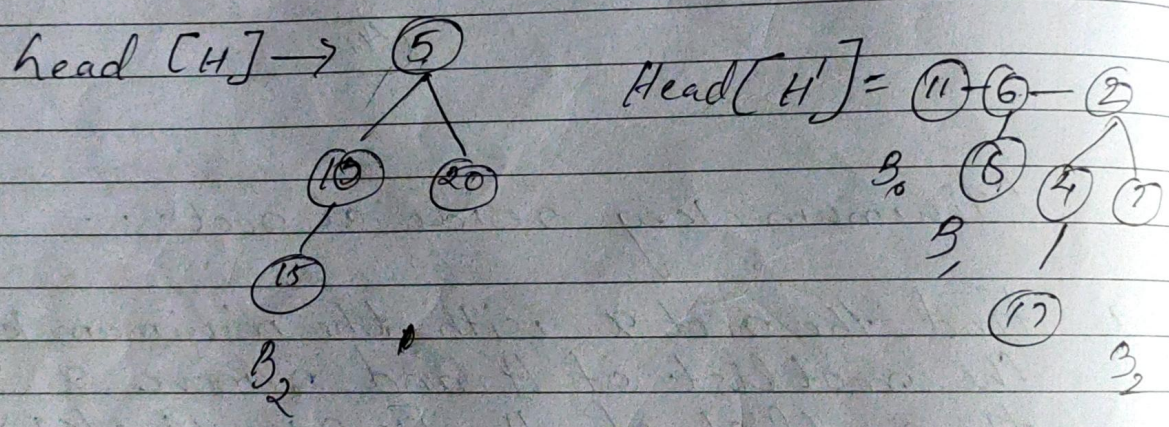
3. Reverse the order of linked list of x' children and set head [H'] ← head of the resulting list.

4. H ← BINOMIAL · HEAP · UNION (H, H')

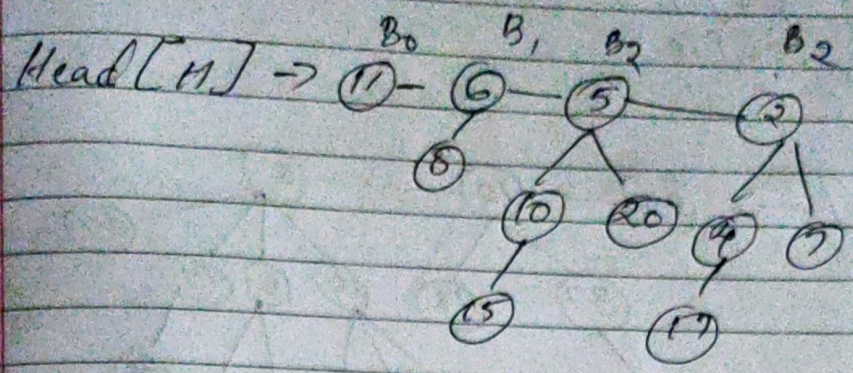
return x
end

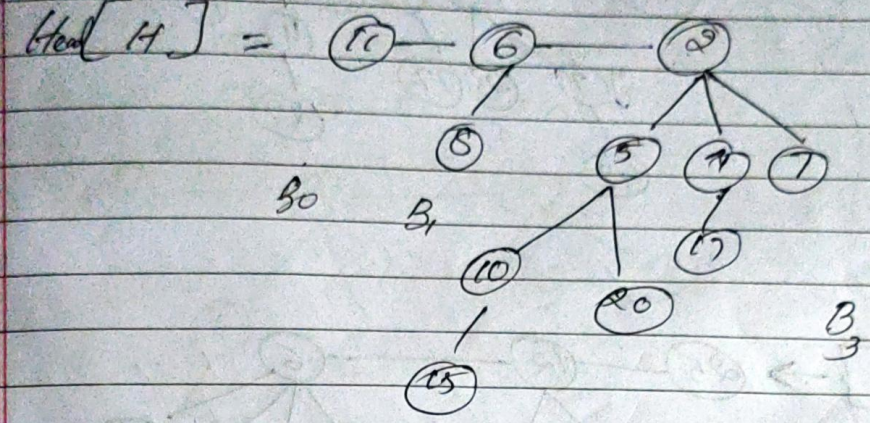
↳ Time complexity O(log n)





→ union H & H'





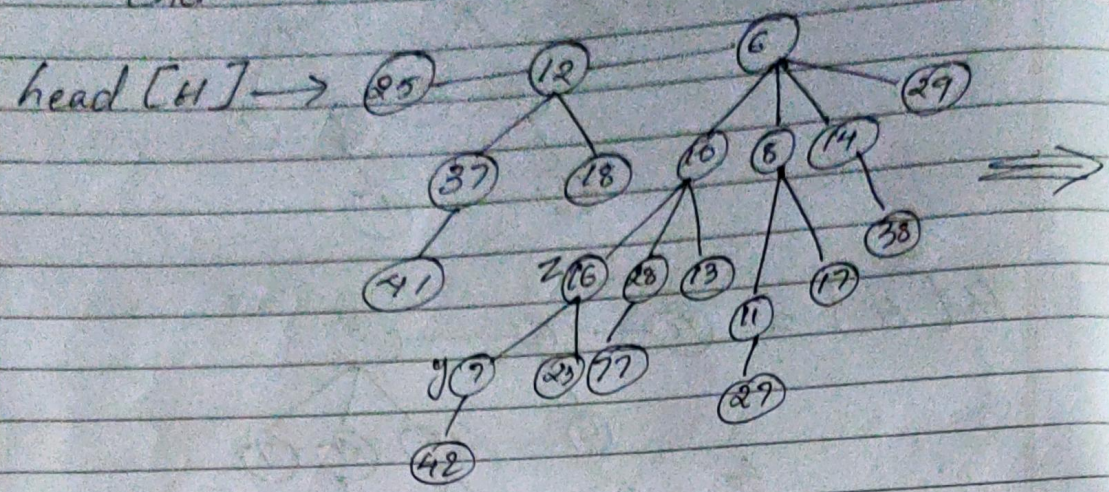
III Binomial Heap – Decrease key [O(log n)]

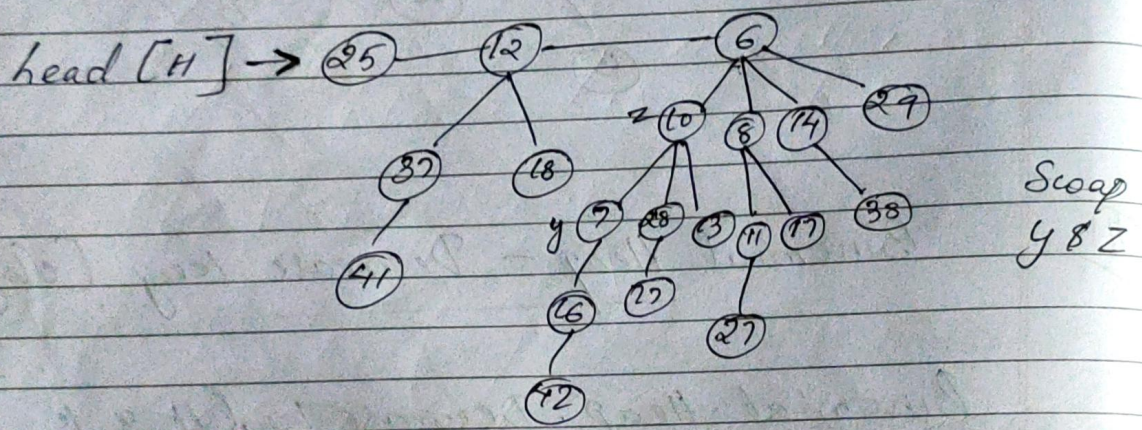
Binomial-Heap-Decrease key[H, x, k)

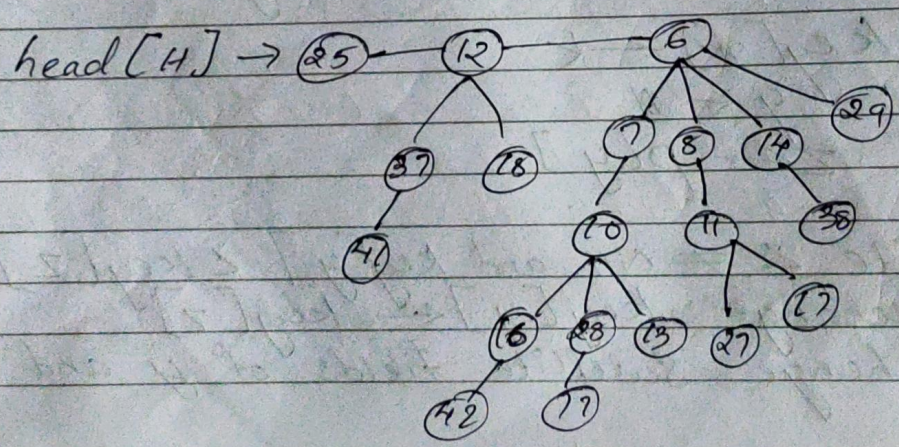
key[x] ← k
y ← x
z ← p[y]

while z ≠ NIL and key[y] < key[z] do
exchange key[y] ← key[z]
exchange satellite fields of y and z

$y \leftarrow z$
$z \leftarrow p[y]$
end while
End





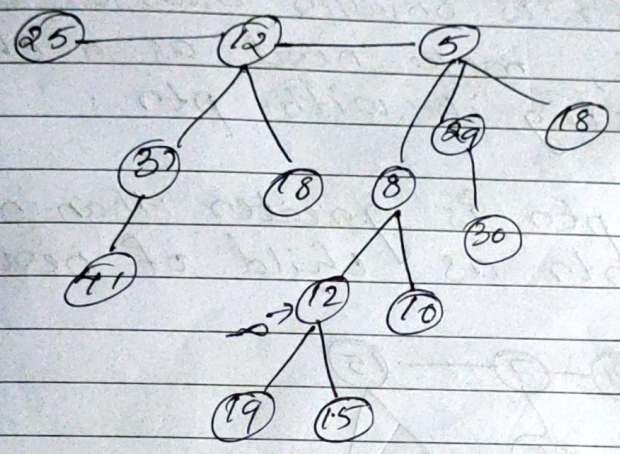


IV. Binomial Heap - Delete key [O(logn)]

Binomial Heap - Delete (H, x)

1. Binomial Heap Decrease key (H, x, $-\infty$)
2. Binomial Heap Extract MIN (H)
3. End.

First Decrease value to $-\infty$ then it comes at root and then do Binomial heap extract MIN (H).



Delete 12: Change 12th place to $-\infty$

V. Binomial Heap Union

The procedure of uniting two binomial heaps into one binomial heap.

Algorithm: Given binomial heaps H_1 & H_2

Step 1: Merge H_1 & H_2 ie, links the roots of H_1 & H_2 in non decreasing order.

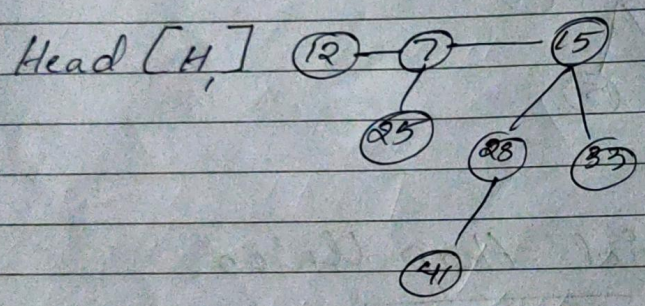
Step 2: Restoring binomial heap by linking binomial trees of the same degree together; traverse the linked list, keep track of three pointers, prev, 'ptr' and 'next'.

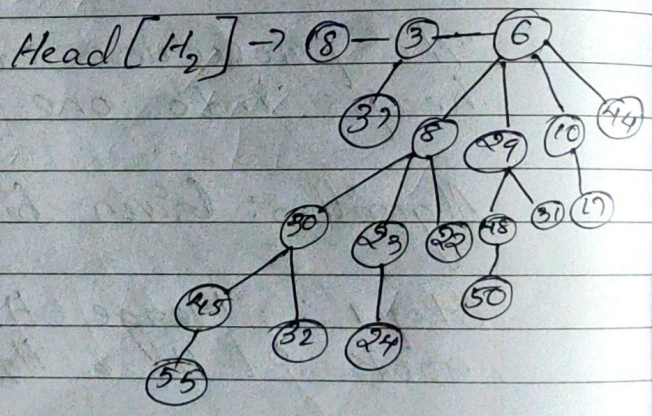
Case 1: degree of ptr and next are not same move ahead

Case 2: Degree of next → next is also same; move ahead.

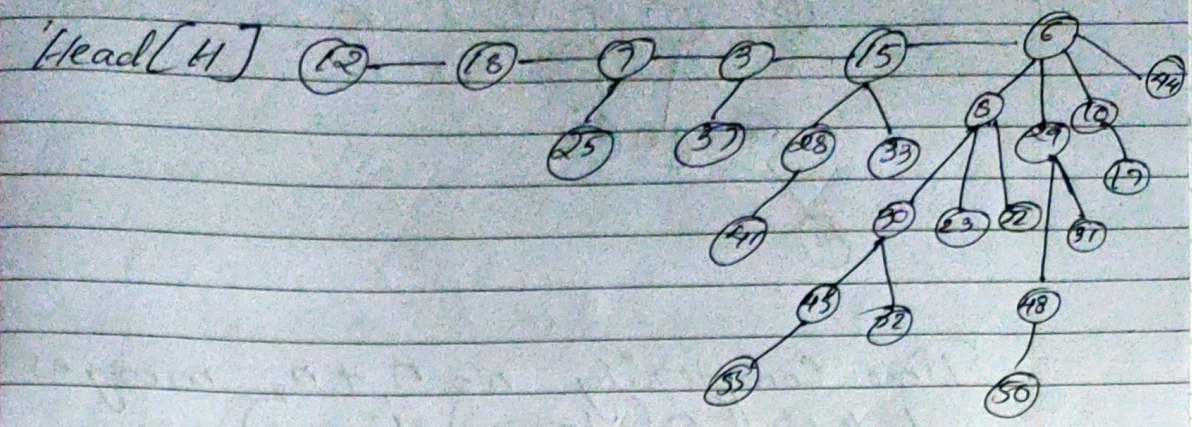
Case 3 If key of ptr is smaller than or equal to key of next, make next as a child of ptr by linking it with ptr.

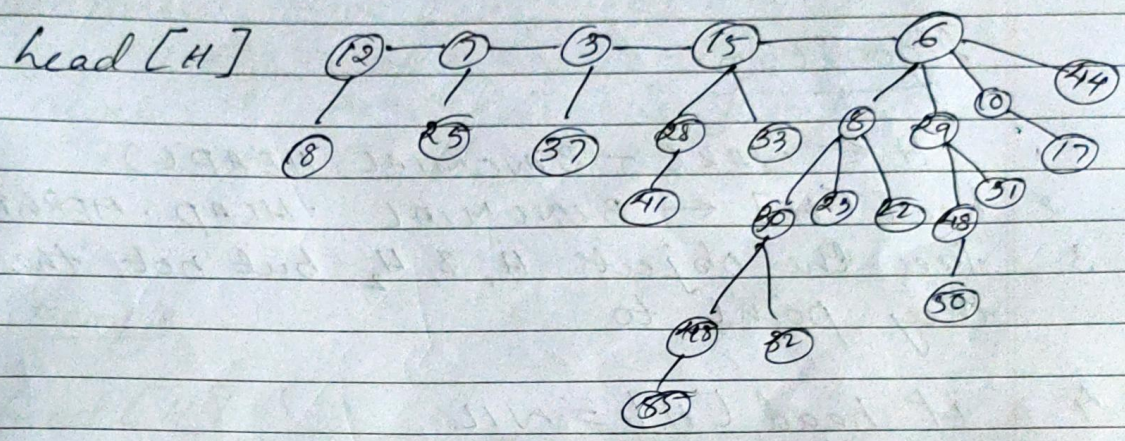
Case 4 If key of ptr is greater than next, then make ptr as child of next.

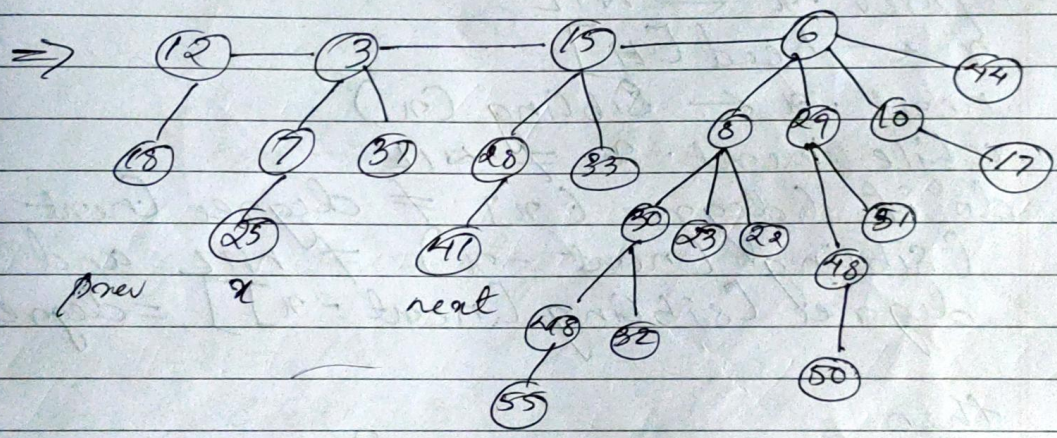


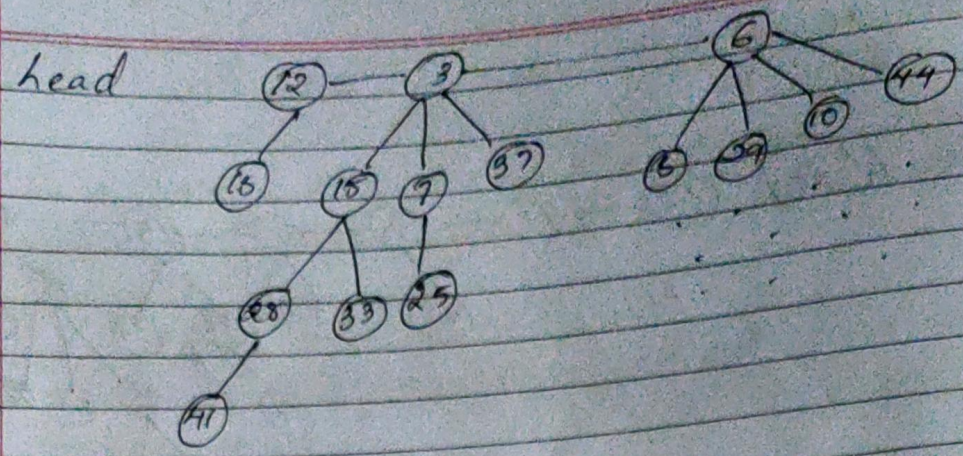


Step 1: Merge









Time Complexity $n = n_1 + n_2$ merges takes $(O(\log n)$ time$)$

Algorithm

```
H ← MAKE - BINOMIAL HEAP()
head[H] ← BINOMIAL · HEAP · MERGE(H, H2)
free the objects H, & H2 but not the list
they point to

if head[H] = NIL
  return H
prev · x ← NIL
x ← head[H]
next · x ← Sibling (x)
while next · x ≠ NIL
do if (degree [x] ≠ degree [next - x]) or
   (sibling [next - x] ≠ NIL and
   degree [sibling [next - x]] = degree[x]

then
        prev · x ← x        } case 1 & 2
        x ← next · x
```

13 else if key(x) ≤ key(next(x))
14 then
Sibling[x] ← Sibling[next · x]
15 Binomial heap Link (next · x, x)

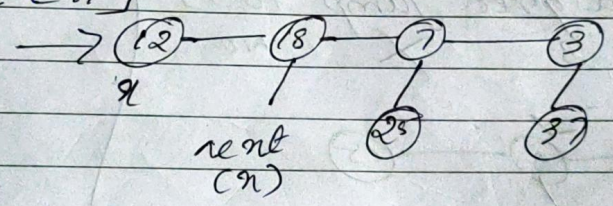
Case 3

16 else if prev · x = NIL
17 then head[H] ← next · x
else
18 Sibling[prev · x] ← next · x
19 Binomial Link (x, next · x)
20 x ← next · x
21 next · x ← sibling [x]
22 return [H]

Case 4

Example

Head [H]



next → x = Sibling (x)

- If degree (x) = degree (next(x))

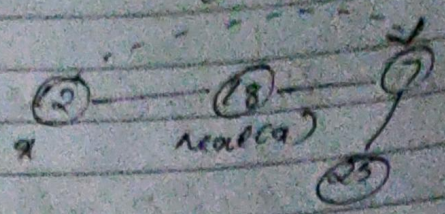
12 = 18 ✓ Yes

- go to Step 13

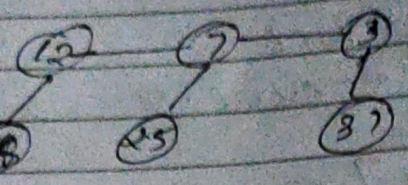
12 < 18

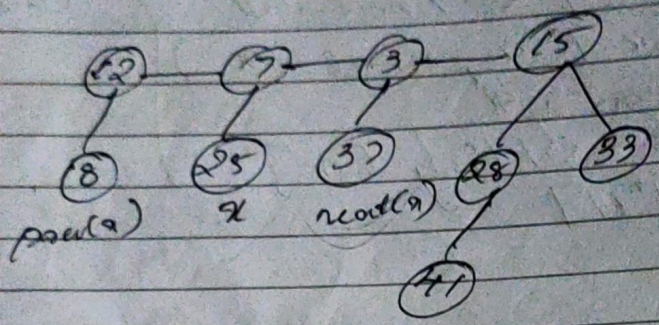
then now sibling (x) became or link

ptr to sibling of next(x)



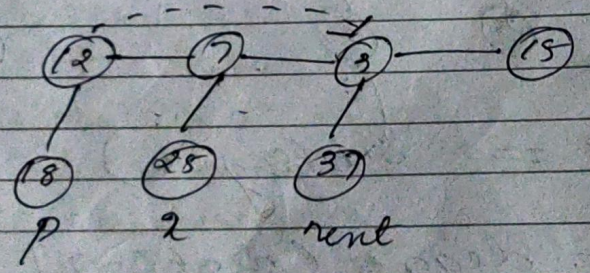
then



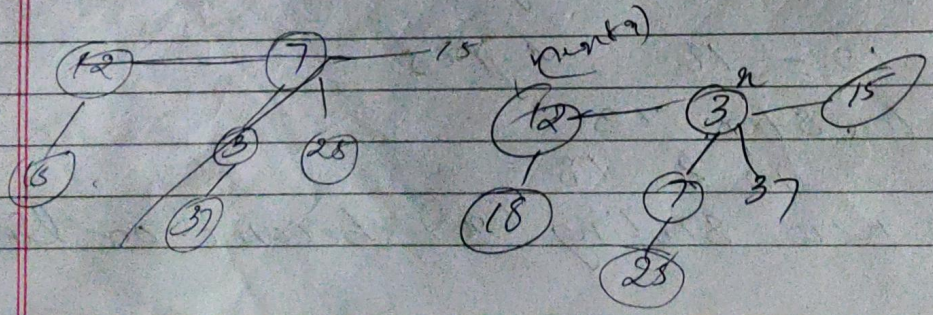


Case 4

Condition n check
Same degree jump to 13
7 < 3 ✗ NO



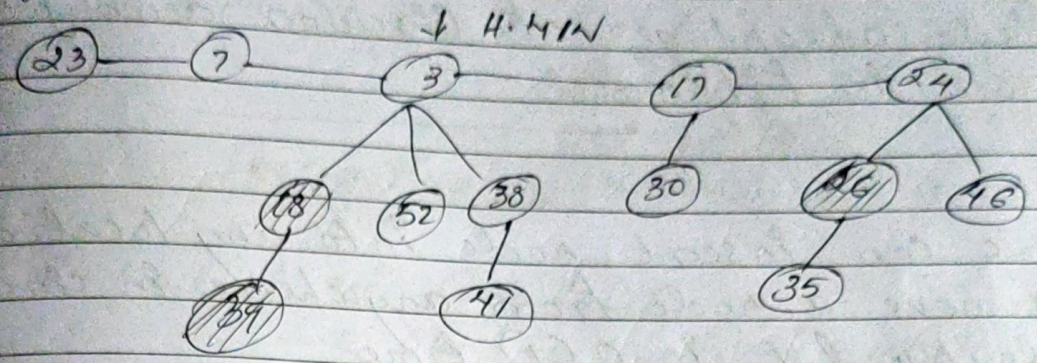
7 < 3
7 7 3



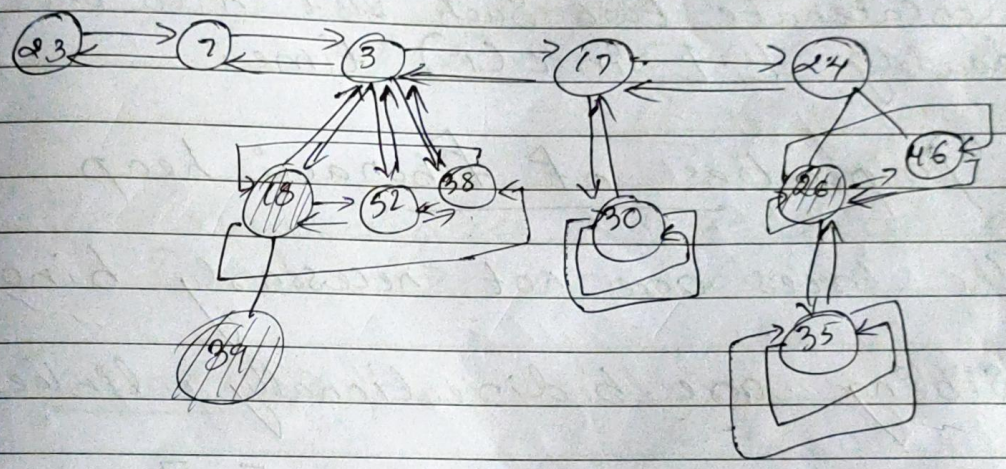
Fibonacci heap

A Fibonacci heap is a collection of rooted tree that are min heap ordered ie; MINHEAP property - The key of node is greater than or equal to the key of its parent.

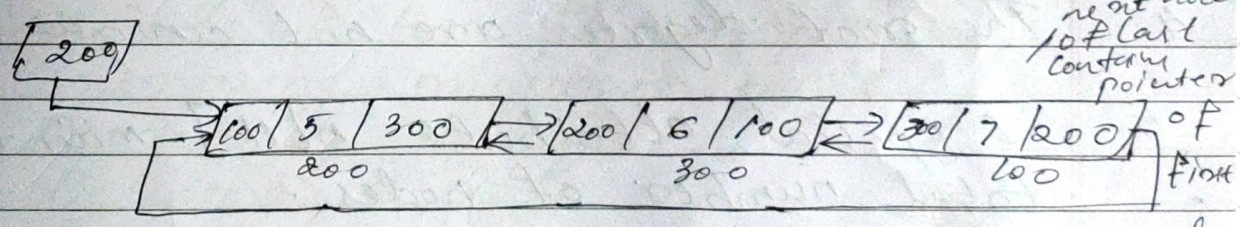
a.



Fibonacci heap



- Fibonacci heap follows circular, doubly linked list structure.



Each node x contains a pointer x.p to its parent and a pointer x.child to any one of its children. The children of x are linked together in a circular doubly L.L.

Two advantages of circular doubly L.L used in fibonacci

1. We can insert node into any location or remove a node from anywhere in circular doubly L.L in O(1) time

2. Concatenate two such list into one circular doubly L.L in O(1) time.

Properties of fibonacci heap

1. The trees are not necessarly binomial

2. Siblings are bidirectionally linked

3. There is a pointer MIN[H] to the root with minimum key.

4. The root degrees are not unique

5. A special attribute n[H] maintains total number of nodes.

Each node has additional label "Mark"

Potential function

To analyse the performance of fibonacci heap operations we use potential method. For a given fibonacci heap H, we indicate by

$t(H)$ = no. of trees in root list of H

$m(H)$ = No. of marked node in H

Potential of fibonacci Heap =

$$\phi(H) = t(H) + 2m(H)$$

from previous example

$$\phi(H) = 5 + 2 \times 3$$
$$= \underline{\underline{11}} \quad \text{(constant)}$$

Maximum degree represented as $D(n) \rightarrow$ upper bound

Operations

1. Creating a new fibonacci heap
2. Inserting a node
3. finding minimum node
4. union of two fibo heap
5. Extracting minimum node
6. Decreasing a key
7. Deleting a node

1. Creating

To make an empty fibonacci heap the MAKE-FIB-HEAP procedure allocates and returns a fibonacci heap object H. where $H.n = 0$, $H.MIN = NIL$; there are no trees in H. So $t(H) = 0$, $M(H) = 0$.
So potential of empty tree

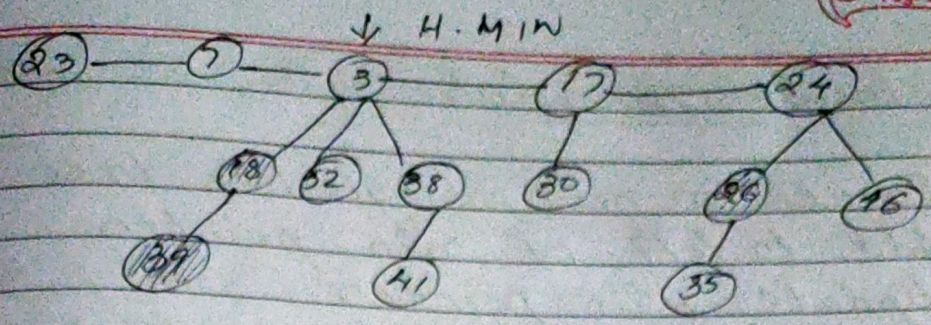
$$\phi(H) = 0$$

So amortized cost of MAKE-FIB-HEAP
$= O(1) =$ actual cost.

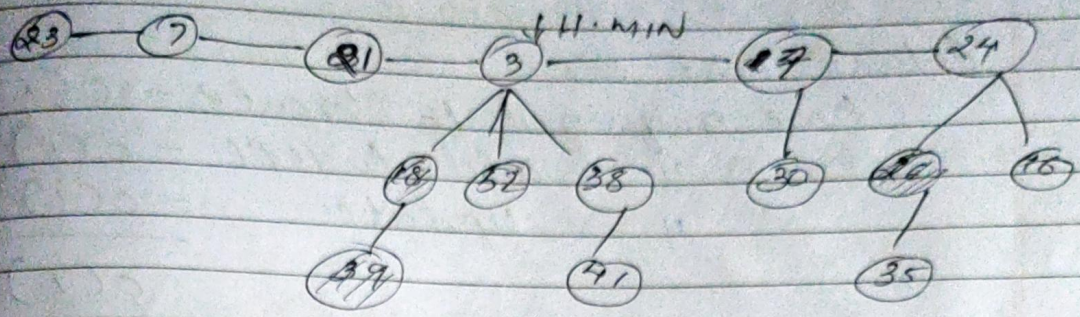
2. Inserting a node

```
FIB-HEAP-INSERT (H, x)
x.degree = 0
x.p = NIL
x.child = NIL
x.MARK = FALSE
If H.MIN == NIL
Create a rootlist containing just x
H.MIN = x
else insert x into H's root list
If x.key < H.MIN.key
    H.MIN = x
H.n = H.n + 1
```

Q9.



Insert 21



* To determine the amortized cost of FIB-HEAP-INSERT,

let H be input Fibo Heap
H' be resulting heap
then

$$t(H') = t(H) + 1$$
$$m(H) = m(H)$$

* Increase in potential (or change in potential)

$$= (t(H) + 1) + 2m(H) - (t(H) + 2m(H))$$

$$= \cancel{t(H)} + 1 + \cancel{2m(H)} - \cancel{t(H)} - \cancel{2m(H)}$$

$$= \underline{\underline{1}}$$

Insert procedure

1. Create a new single tone tree
2. Add to left of min pointer
3. update min pointer

Actual cost ⇒
time required to create = $O(1)$
" add to left = $O(1)$
" update = $O(1)$
$O(1)$

* Amortized
cost = Actual cost + change in potential
$= O(1) + 1$
$= O(1)$

3. finding minimum node

The minimum node of a fibonacci heap H is given by the pointer H.MIN. So find minimum node in $O(1)$ time. Potential of H does not change.

∴ Amortized cost = $O(1)$ = actual cost

4 Union two fibonacci heap

* Concatenates rootlist of H_1 & H_2

and then determines new minimum node.

FIB · HEAP · UNION (H_1, H_2)

1. H = MAKE · FIB · HEAP()
2. $H \cdot MIN = H_1 \cdot MIN$
3. Concatenate the rootlist of H_2 with the rootlist of H.
4. If ($H_1 \cdot MIN == NIL$) or ($H_2 \cdot MIN \neq NIL$ and $H_2 \cdot MIN \cdot key < H_1 \cdot MIN \cdot key$)
5. $H \cdot MIN = H_2 \cdot MIN$
6. $H \cdot N = H_1 \cdot N + H_2 \cdot N$
7. return H

Potential (Change in Potential)

$$\phi(H) - (\phi(H_1) + \phi(H_2))$$

$$= (t(H) + 2m(H)) - [t(H_1) + 2m(H_1) + t(H_2) + 2m(H_2)]$$

$$= t(H) + 2m(H) - t(H_1) - 2m(H_1) - t(H_2) - 2m(H_2)$$

Since $t(H) = t(H_1) + t(H_2)$
$m(H) = m(H_1) + m(H_2)$

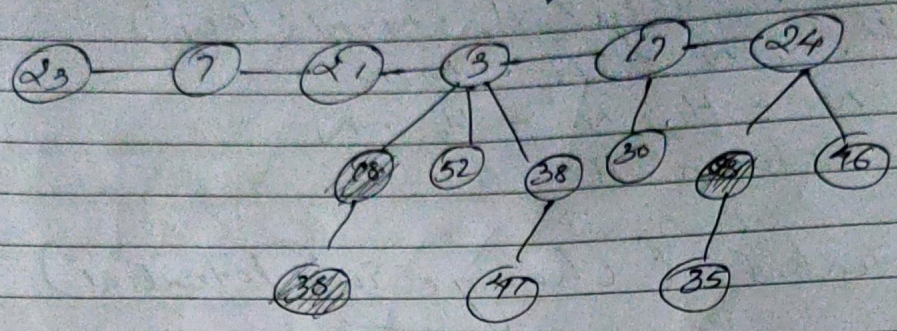
$$\therefore \cancel{t(H_1)} + \cancel{t(H_2)} + 2m(H_1) + 2m(H_2) - \cancel{t(H_1)} - 2m(H_1) - \cancel{t(H_2)} - 2m(H_2)$$

$=$

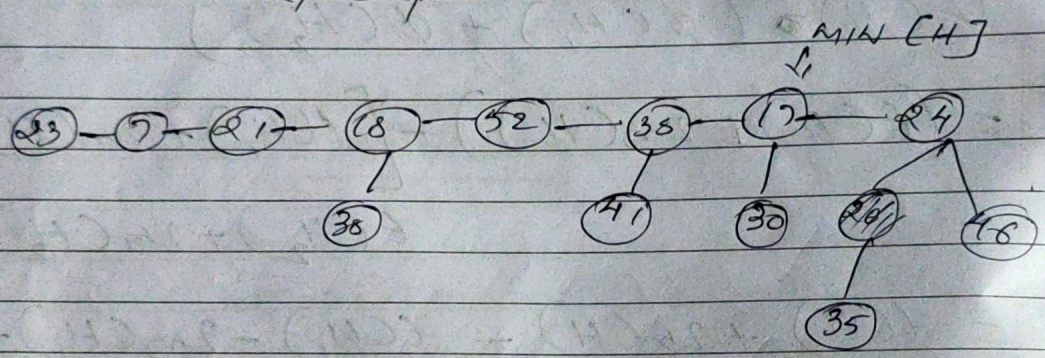
Amortized Cost = 0

5. Extracting Minimum (H)

The operation is accomplished by first deleting the minimum key node and moves all its children to rootlist. It uses procedure 'CONSOLIDATE' to merge the tree with same degree

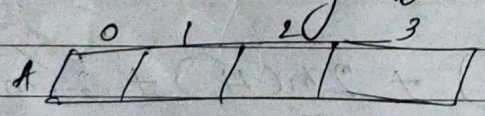


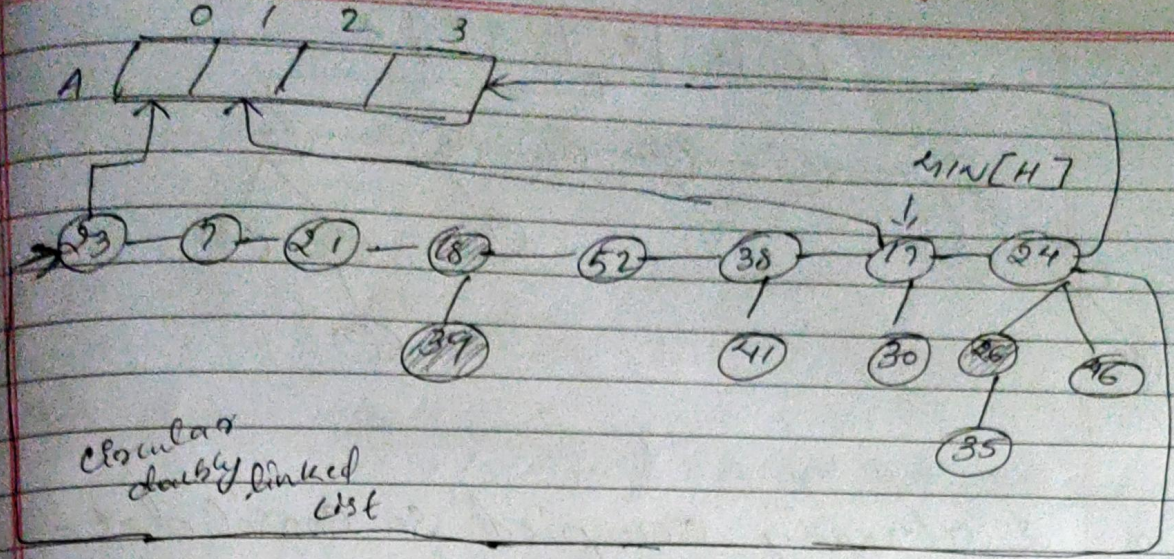
// Step 1



* ie, update minimum pointer to the right of minimum node.

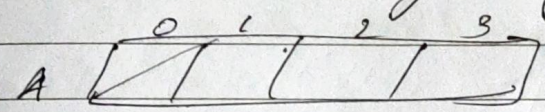
* Create an array based on degree

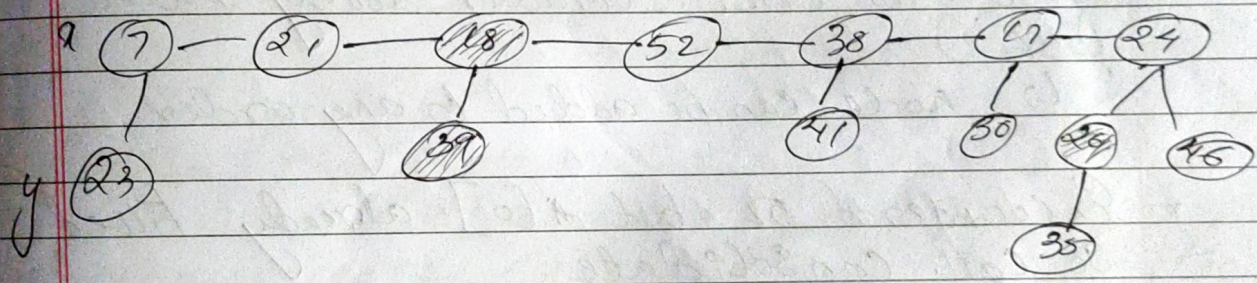




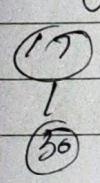
Then node 7 tries to allocate into array with degree 0, but collision occurs then call CONSOLIDATE procedure to merge the tree with same degree.

Compare the key and make smaller key (x) in the root & larger key as the child (y)

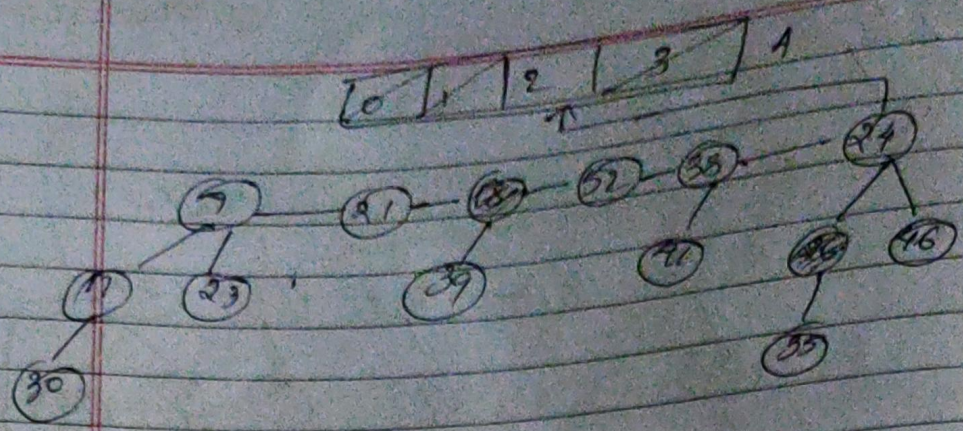




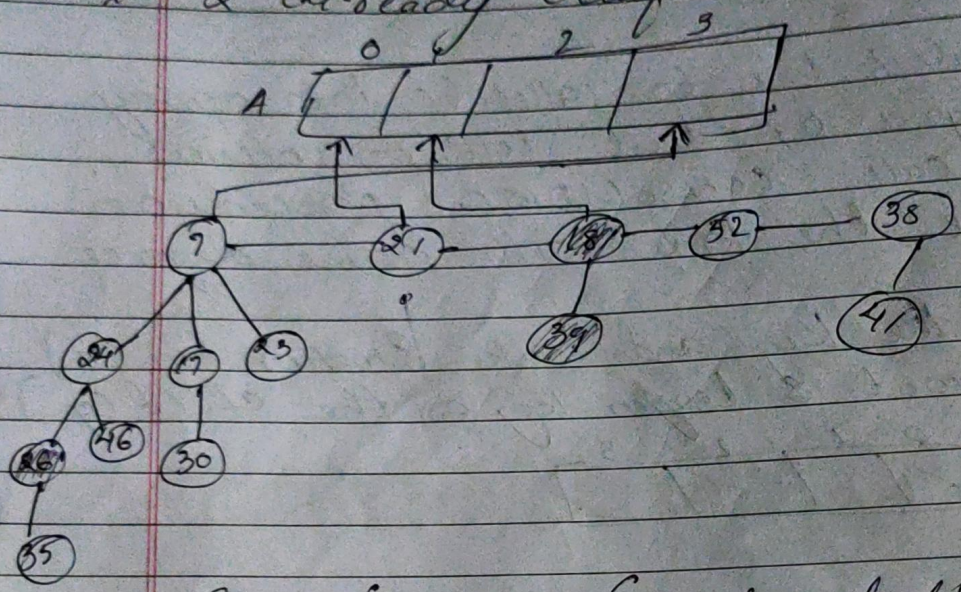
Degree 1 is already allocated with



so consolidate.

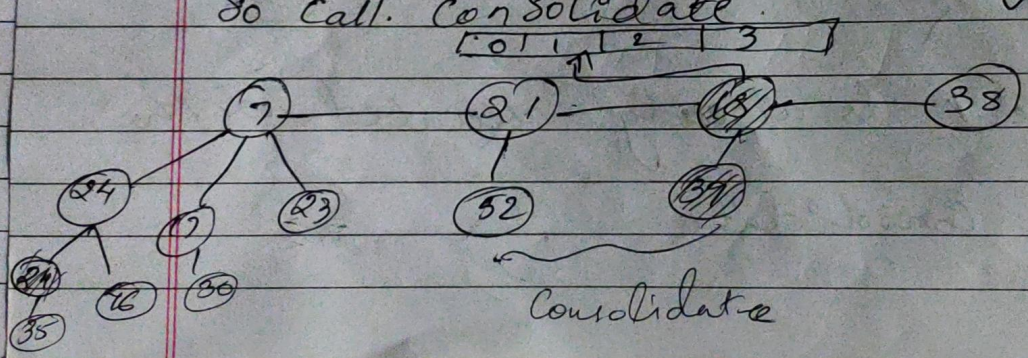


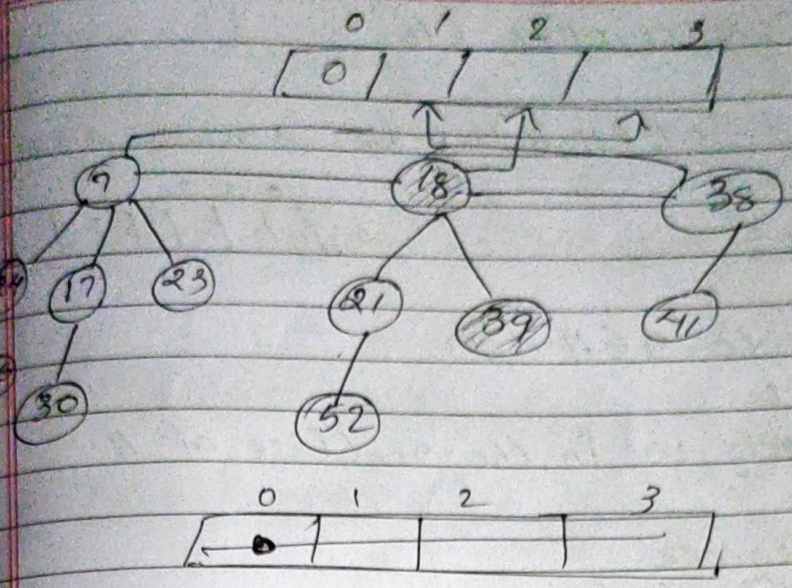
* 2 already occupied so consolidate



* Since its CDLL (circular doubly linked list)
  ↳ node can be added to any order

* Encountered 52 but A[0] already filled so call. consolidate





FIB-HEAP-EXTRACT-MIN(H)

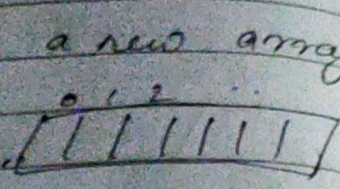
```
Z ← MIN(H)
If Z ≠ NIL
then for each node x of Z
do add x to the rootlist of H
   p(x) ← NIL
remove Z from the rootlist of H
If Z = right(Z) // Z was the only node in
                   rootlist with no child

then min[H] ← NIL
else min[H] ← right[Z]
CONSOLIDATE (H) // every root in the
                   rootlist has a distinct
                   degree value.

n[H] ← n[H] - 1
Return Z.
```

CONSOLIDATE (H)

```
let A[0 ... D(H.n)] be a new array
```




```
for i = 0 to D(H.n)
A[i] = NIL
for each node w in the rootlist of H
    x = w
    d = x.degree
while A[d] ≠ NIL
do y ← A[d]
If key[x] > key[y]
then exchange x, y
FIB-HEAP-LINK (H, y, x)
    A[d] ← NIL
    d ← d + 1
A[d] ← x   // After consolidate update
                MIN pointer of modified
                tree.
for (i ← 0 to D(H))
do if A[i] ← NIL
then add A[i] to the rootlist of H
If MIN(H) = NIL or
    key[A[i]] < key [MIN(H)])
then MIN[H] ← A[i]
```

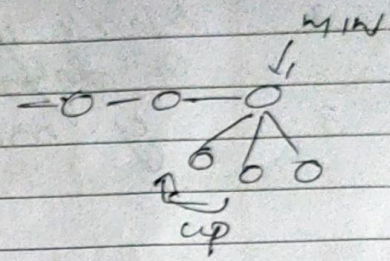
Potential

$$\phi(H) = t(H) + 2 \cdot m(H)$$

Actual cost $= O(\text{rank}(H) + O(\text{trees}(H))$

* The size of rootlist upon calling 'Consolidate' is almost

$$D(n) + t(H) - 1$$ — negated node



* Total amount of consolidate is $O(D(n)) + t(H)$

* Potential before extract-min $= t(H) + 2m(H)$

* Potential after Extract-min is almost

Mathematically proven // $(D(n) + 1) + 2m(H)$ // Since almost $D(n)+1$ root remains

Amortized cost

$$\underbrace{O(D(n) + t(H))}_{\text{Consolidate potential}} + \overbrace{(D(n)+1) + 2m(H)}^{\text{extract min}} - \underbrace{(t(H) + 2m(H))}_{\text{original}} \leftarrow \text{Actual cost}$$

$= O(D(n)) + O(t(H)) - t(H)$ (Amortized Cost)

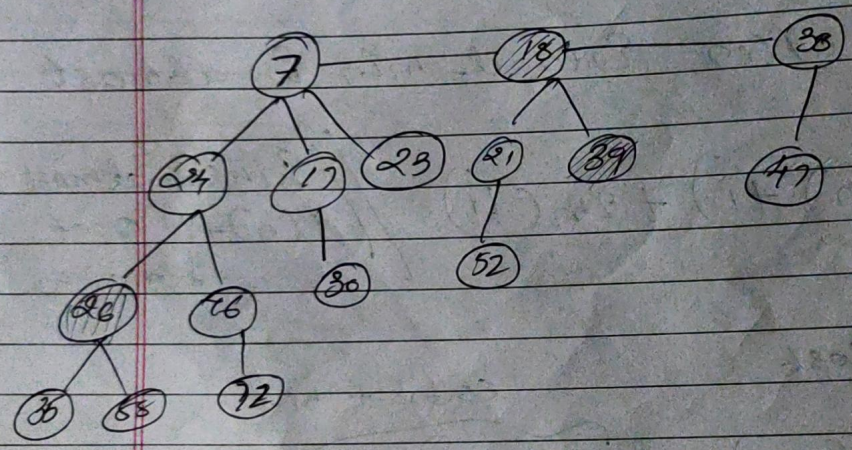
$= O(D(n))$ $= O(D(H))$

$D(n) \le \log n$ $O(\log n)$

FIB-HEAP-LINK(H, y, x)

1. remove y from rootlist of H
2. Mark y a child of x incrementing x.degree
3. y.mark = FALSE

Decreasing a key and deleting a node



Marked.

Case 1: heap order property not violated

1. Decrease key of x
2. Change heap min pointer (if necessary)

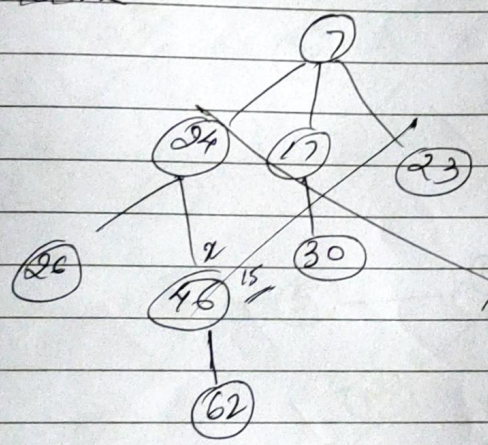
eg: 46 change to 35 ?

easily we can decrease

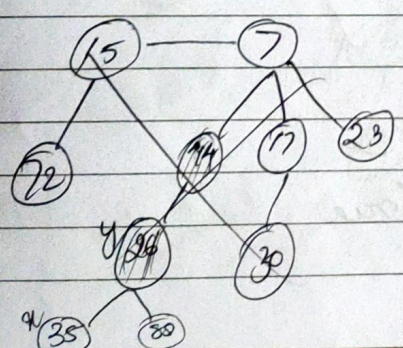
Case 2

[heap order violated]

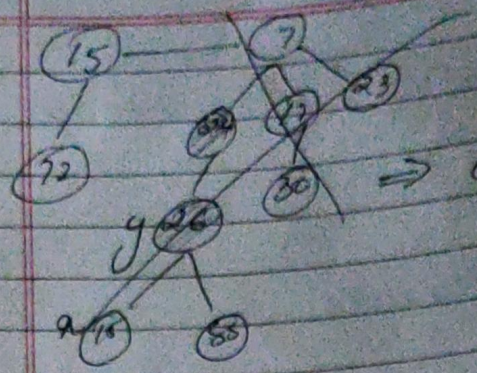
- Decrease key of x [eg 46 decrease to 15]
- Cut tree rooted at x, meld into rootlist and unmark.
- If parent p of x is unmarked then mark it.
- otherwise cut p & meld into rootlist and unmark



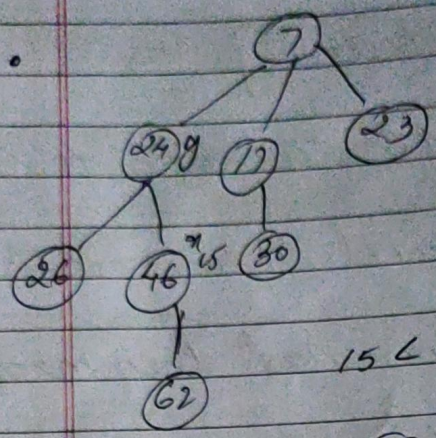
15 < 24 apply cut



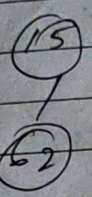
Decrease 35 to 5

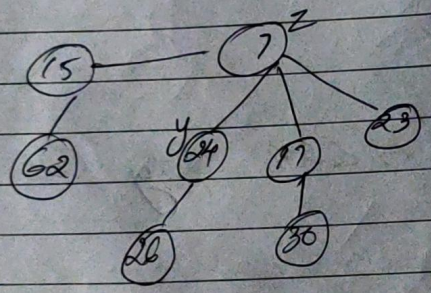


⇒ call cut procedure

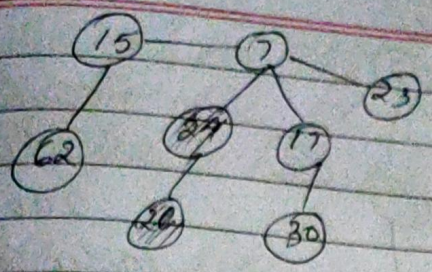


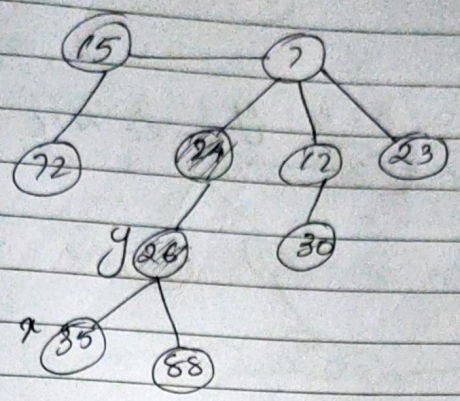
15 < 24 apply cut



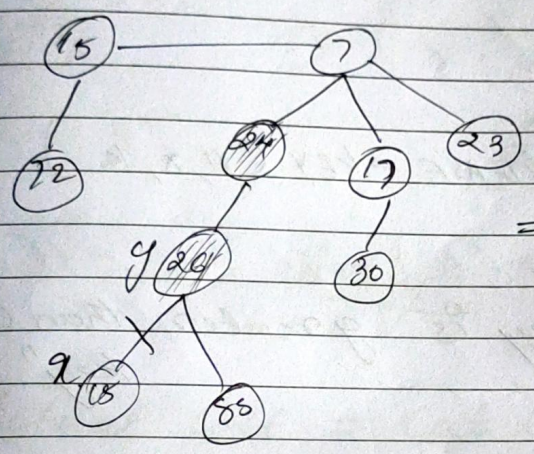


The call cascade cut
if $z \neq$ NULL
Mark(y) = false
then Mark(y) = true

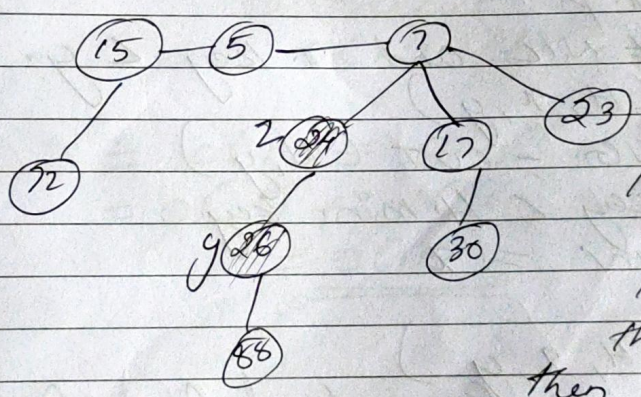




Decrease 35 to 5

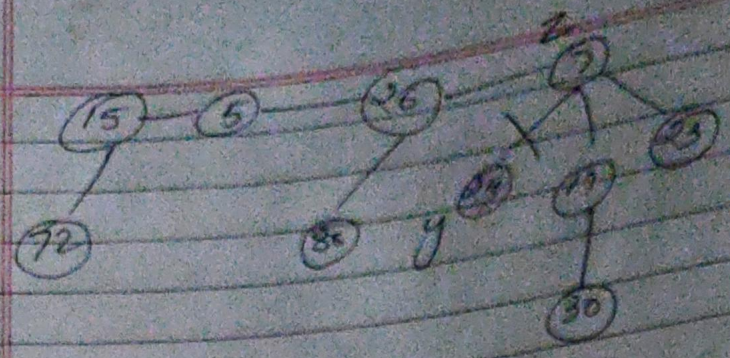


=> call cut produce

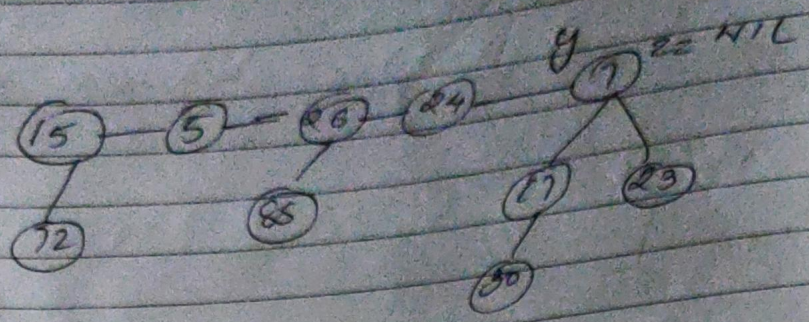


Apply cascade cut.

In cascade cut
$z = p(y)$
if $z \neq NIL$
then if mark(y) = f
then mark(y) = T
but here True so call
cut(H, y, z)



Cascade (H, z)
So 26 cut to root & apply cascade cut
z = p(y)



Algorithm

FIB-HEAP-DECREASE-KEY (H, x, k)

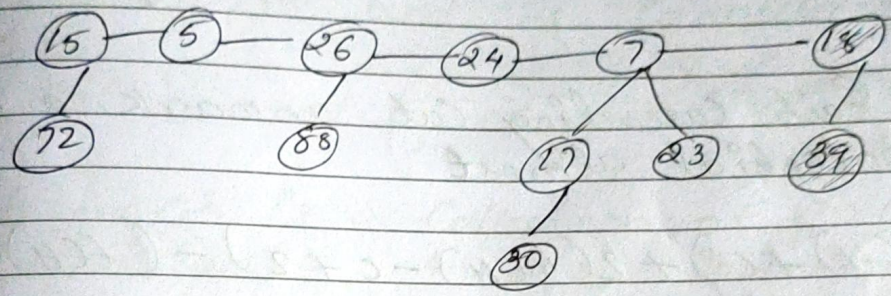
1. If $k > x.key$
2. error "new key is greater than current key"
3. $x.key = k$
4. $y = x.p$
5. If $y \neq NIL$ and $x.key < y.key$
6. CUT (H, x, y)
7. CASCADING-CUT (H, y)
8. If $x.key < H.min.key$
9. $H.min = x$

CUT (H, x, y)

1. remove x from child list of y, decrementing y.degree

add x to the rootlist of H
$x.p = NIL$
$x.mark = FALSE$

CASCADING-CUT (H, y)
$z = y.p$
If $z \neq NIL$
If $y.mark == FALSE$
$y.mark == TRUE$
else CUT (H, y, z)
CASCADING CUT $(H\ z)$



Analysis

Actual cost = cut $(O(1))$ + recursive cascadecut c (times then $O(1)$ time needed).

$$= O(1) + c \cdot O(1)$$
$$= O(1) + O(c)$$
$$= \underline{\underline{O(c)}}$$

Amortize

Change in potential

- Initially → $t(H) + 2m(H)$
- After apply cuts → $t(H) + c$ trees
  $= t(H) + (c-1)$ trees (cascade) + 1 tree at root
- Marked nodes at most $\underline{m(H) - c + 2}$

ie; Each cascading cut unmark a node potential almost

$(t(H) + c) + 2(m(H) - c + 2) - (t(H) + 2m(H))$

$= t(H) + c + 2m(H) - 2c + 4 - t(H) - 2m(H)$

$= c - 2c + 4$
$= (4 - c)$

Amortized cost = Actual + P. Diff
$O(c) + 4 - c = O(1)$

Deleting a node

Amortized cost at most $O(D(n))$

Amortized cost at most $O(D(n))$

FIB · HEAP - DELETE (H, x)

1. FIB HEAP DECREASE KEY (H, x, $-\infty$)
2. FIB HEAP - EXTRACT MIN (H)

Module - 4

Graph algorithms

Many problems are formulated in terms of objects and the connection between them

eg: Airline routes, electrical circuit.

* A graph is a collection of vertices V and edges E; An edge connects two vertices.

* A simple path is a path with no vertex repeated and a simple cycle is a simple path except the first & last vertex is repeated

* A tree is a graph with no cycle.

* A Complete graph is a graph in which all edges are present.

* A Sparse graph is a graph with relatively few edges.

* A dense graph is a graph with many edges.
* An undirected graph has no specific direction between the vertices.
* A directed graph has edges that are 'one way'.
* A weighted graph has weights associated with each edge.

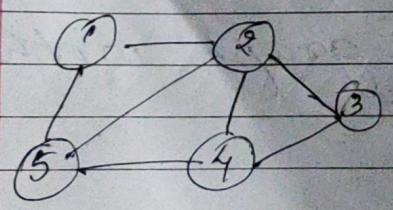
Graph Representation

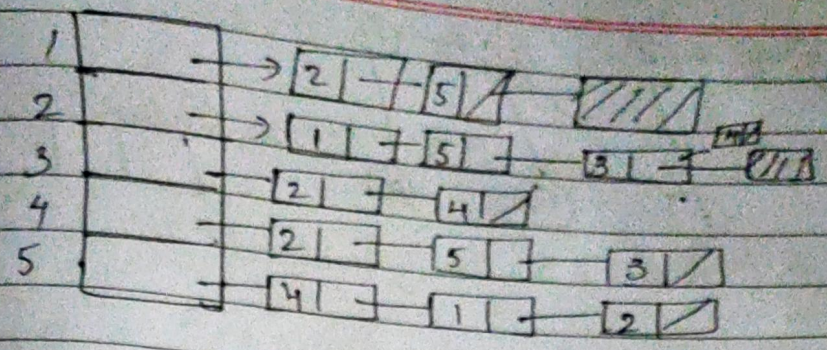
2 ways

1. Adjacency list
2. Adjacency matrix

An adjacency list is an array which contain a list for each vertex. The list for a given vertex contains all other vertices that are connected to the first vertex by a single edge

An adjacency matrix is a matrix with rows and column for each vertex. Matrix entry is one if there is an edge





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

Tree Searching algorithm (Searching algorithm for directed or undirected graph)

1. Breadth First Search (BFS)
2. Depth first Search (DFS)

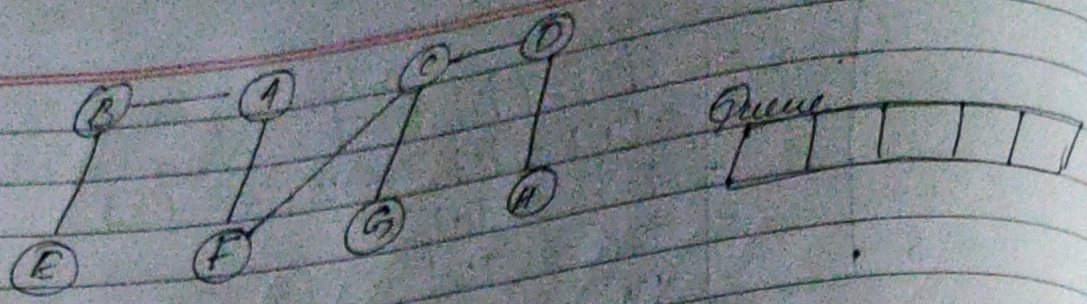
BFS

Process of visiting a nodes

* It starts at the root of a graph and explores all of the neighbour nodes at the present depth prior to moving on to the nodes at the next depth level.

* DS used to BFS is queue.

eg:



Array visited

A	B	C	D	E	F	G	H
0	0	0	0	0	0	0	0

Queue

A

* If graph has unvisited vertex, visit and enqueue it.

A	B	C	D	E	F	G
1	0	0	0	0	0	0

Queue

A

* While queue is not empty, dequeue vertex and visit its adjacent unvisited nodes and enqueue them.

A	B	C	D	E	F	G	H
1	1	0	0	0	0	0	0

Queue

A	B	F

A	B	C	D	E	F	G	H
1	1	1	1	1	1	1	

A	B	F	E	C	G

A	B	F	E	C	G	D

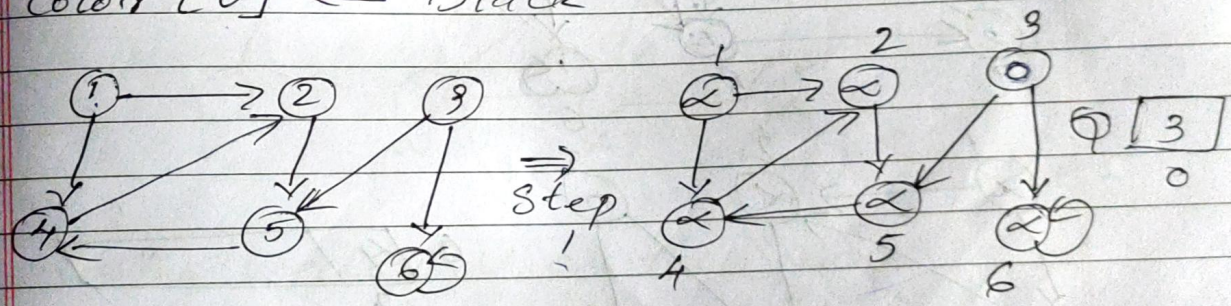
A	B	C	D	E	F	G	H
1	1	1	1	1	1	1	1

A	B	F	E	C	G	D	H

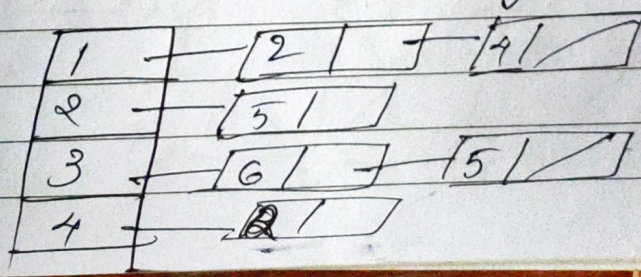
Algorithm BFS (G, s)

```
for each vertex u ∈ V(G) - {s}
do color[u] ← white
d[u] ← ∞        // adjacent node initialise to ∞
π[u] ← NIL      // parent NILL
color[s] ← Gray
d[s] ← 0
π[s] ← NIL
Q ← ∅
Enqueue (Q, s)
while Q ≠ ∅
do u ← DEQUEUE (Q)
for each v ∈ Adj[u]
do if color[v] = WHITE
  then color[v] ← Gray
  d[v] ← d[u] + 1
  π[v] ← u
  Enqueue (Q, v)
color[u] ← Black
```

eg:



Create adjacency list



5	4
6	6

So adj[3] = {6, 5}

So color[6] = Gray

color[5] = Gray

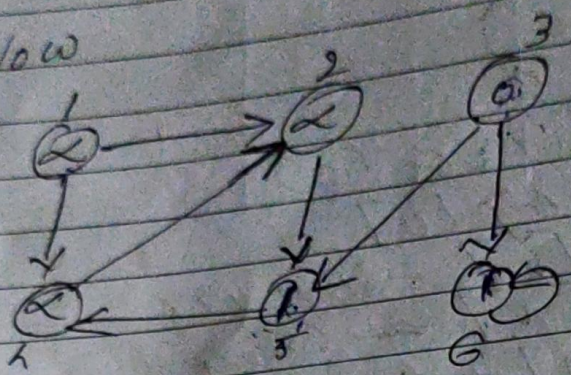
d[6] = 1

π[6] = 3

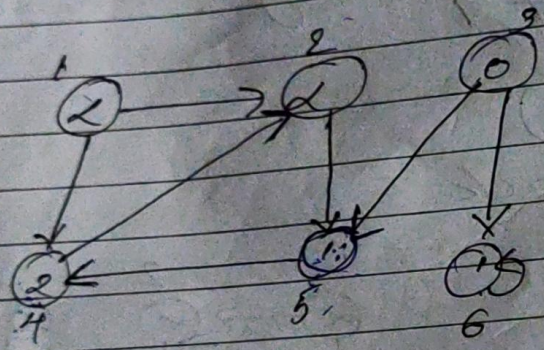
d[5] = 1

π[5] ← 3

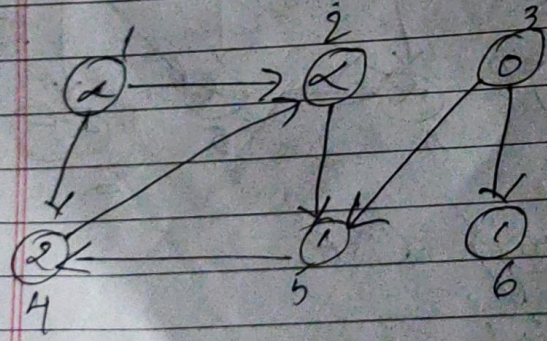
Now



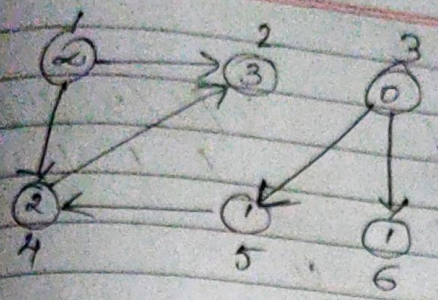
Q [5 | 6]
1 1



Q [6 | 4]
1 2



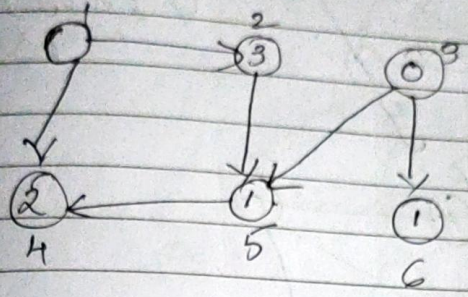
Q [4 |
2



Q [2] 3

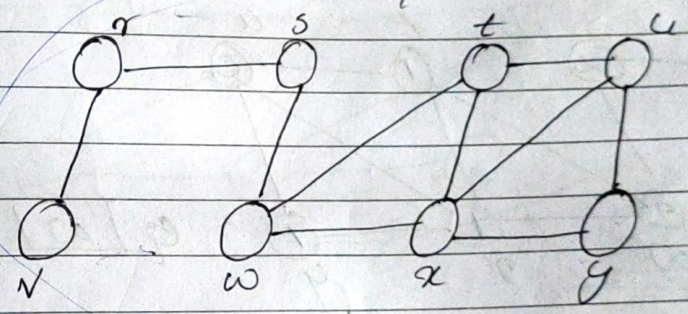
Q [∅]

No adjacent from 3



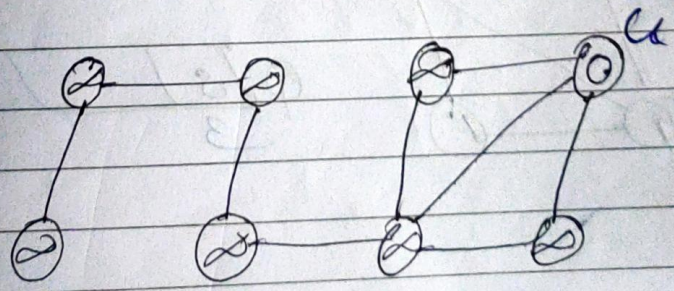
Vertex 1 cannot be reachable from source

Q.10 Consider the graph G shown in the fig Describe the whole process of BFS



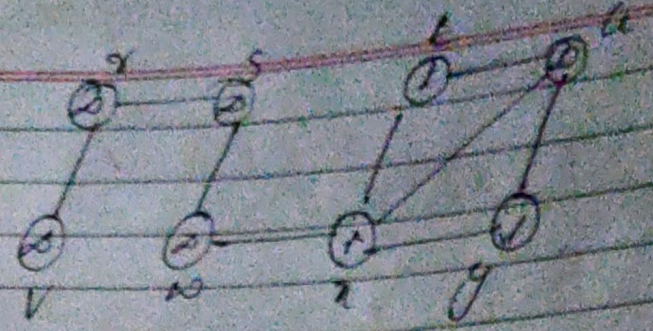
Using vertex v as the source

a.



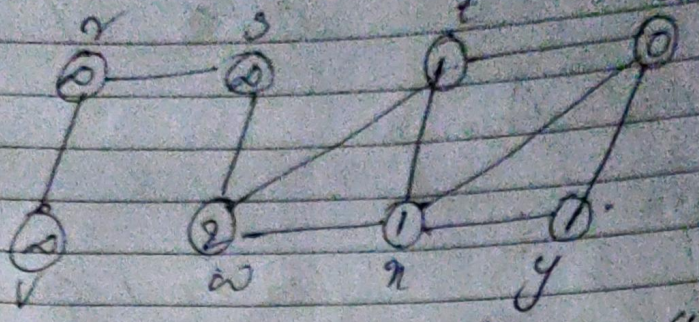
Q [u]

b.



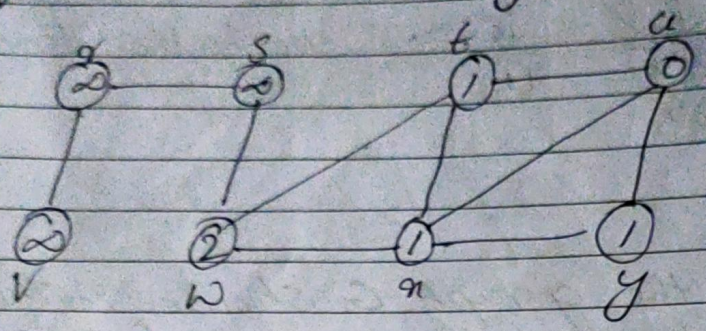
Q [t | x | y]
1 1 1

c



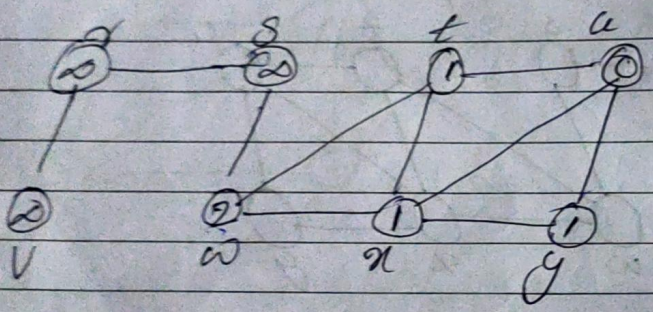
Q [x | y | w]
1 1 2

d.



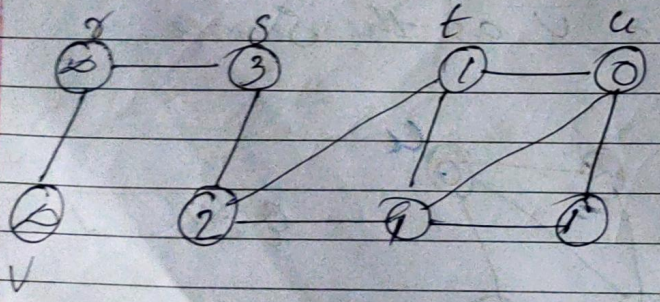
Q [y | w]
1 2

e

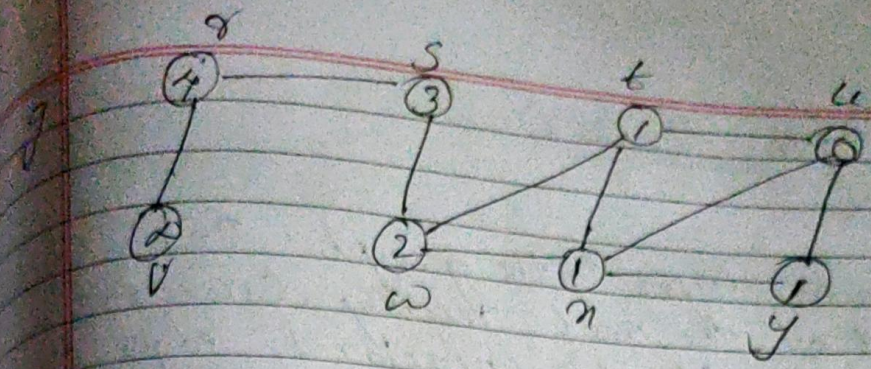


Q [w]

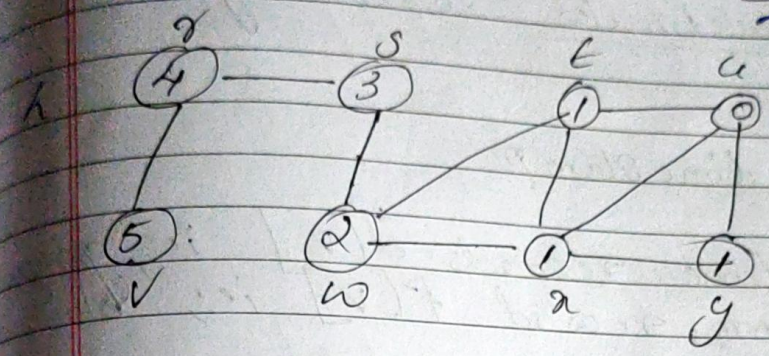
f



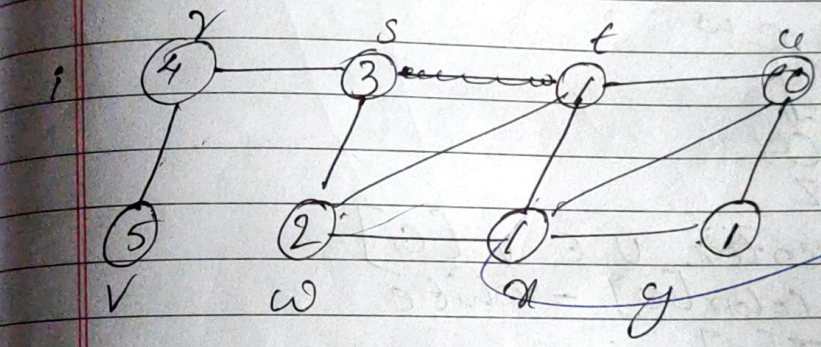
Q [s]
3



Q [r]



Q [v]



$Q = \phi$

Depth first Search

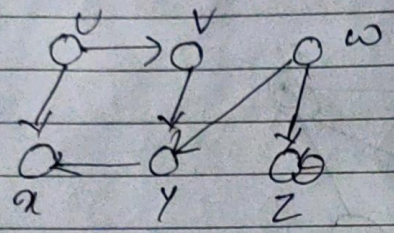
To Search deeper in the graph. DFS explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. Once all of v's edges have been explored, the search backtracks to the original source vertex.

In DFS 2 time stamp

1. Discovering time records $d[v]$ ie, graged
2. finishing time record $f[v]$ ie, blacked

* DFS is implemented used stack ds

eg:



for each vertex $u \in V[G]$
do color[u] = white
$\pi[u]$ = NIL

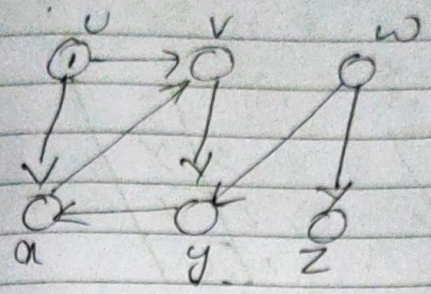
ie, assign color white to all vertices in the graph.

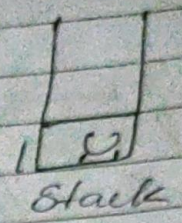
// for each vertex $u \in V[G]$
if color[u] = white
DFS-VISIT[u]

Now apply DFS-VISIT [u]

color[u] = gray
time = 0 + 1 = 1
d[u] = 1

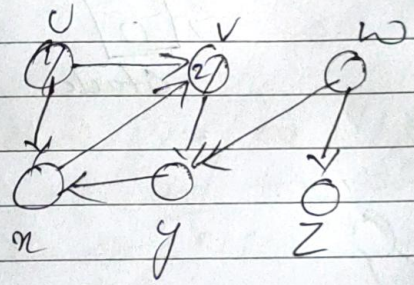
ie;

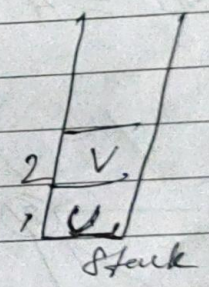




// for each $v \in adj[u]$
ie; $adj[u] = \{v, x\}$

if color[v] = white [true]
$\pi[v] = u$ // make u as parent of v
Now, apply DFS VISIT [v]
Make color[v] = gray
time = 1 + 1 = 2
d[v] = 2





// Now find adjacent of vertex v
adj[v] = y

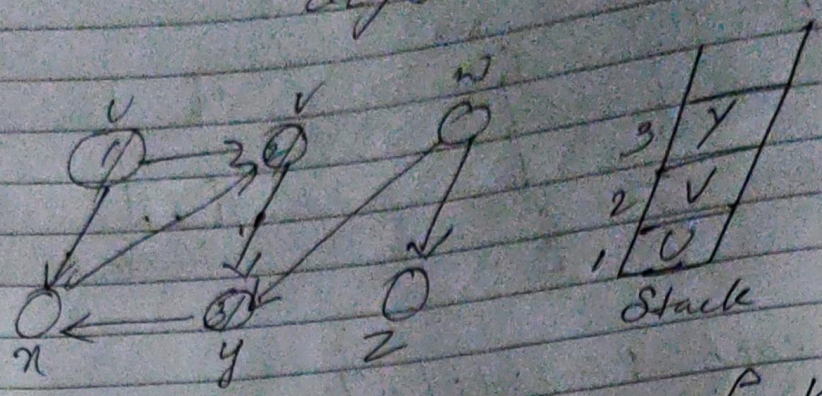
Check color of y if color[y] = white
// make parent of y as v
$\pi[v] = y$

Again apply DFS-VISIT for Vertex y

// Color[y] = gray

time = 2+1 = 3

d[y] = 3



// Now find adjacent of Vertex y

adj[y] = x

Color[x] = white

Check Color of x

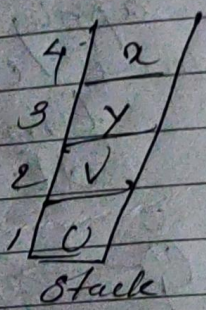
Make vertex y as parent of x

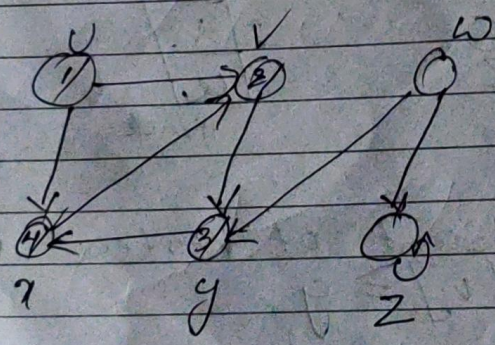
DFS-VISIT[x]

Color[x] = gray

time = 3+1 = 4

d[x] = 4





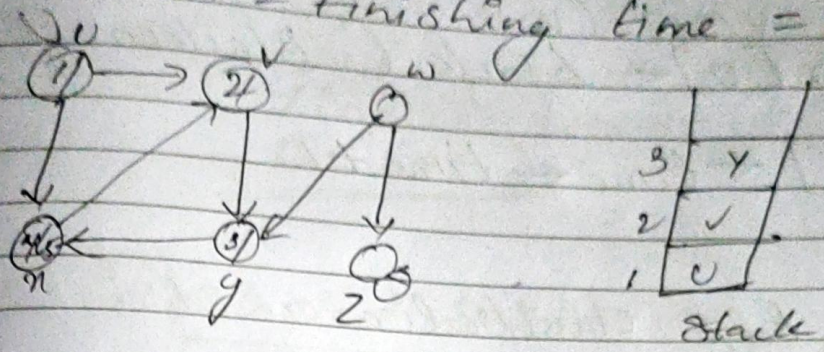
// Now find adjacent of vertex x

adj[x] = v

check color of v

color[v] = white ✗ No
grey

So color[x] = black
time = 4+1 = 5
f(x) = finishing time = 5



Similarly apply above steps to explore all the vertices present in the graph.

Algorithm

DFS (G)

1. for each vertex $U \in V[G]$
2. do color[U] ← white
3. $\pi[U]$ ← NIL
4. time ← 0
5. for each vertex $U \in V[G]$
6. do if color[U] = white
7. then DFS - VISIT (U)

DFS-VISIT (U)

1. Color[U] ← Gray // white vertex U ha just been discovered

```
time ← time + 1
d[u] ← time
for each v ∈ Adj[u] // explore edge (u, v)
    do if color[v] = white
        then π[v] ← u
            DFS-VISIT (v)
color[u] ← black ( Blacken u, it is finished )
f[u] ← time ← time + 1
```

Edge classification on DFS

1. True edge
2. forward edge
3. Back edge
4. Cross edge

- True edge: Edge in depth first forest

Back edge: (u, v) in which v is an ancestor of u in the depth first forest. Self loop are also considered as back edge.

- forward edge : edge (u, v), in which v is v appears after u and path from u to v

Cross edge - (u, v) where there is no path from v to u.

$U \to V$

$U \to V$

$V \to U$

BFS Time Complexity

$T = O(|V| + |E|)$ — set of edges [worst case]

set of vertices

DFS

Each DFS call $\Theta(V)$

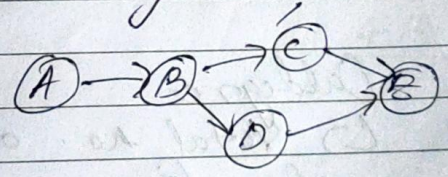
After DFS, check adjacent edges $\Theta(E)$

$\therefore$ Total time in worst case

$\Theta(V+E)$

Topological Sort

- Graph should be acyclic and directed
- It is linear ordering of its vertices such that for every directed edge (u, v) for vertex u to v, u comes before vertex v in the ordering.
- Graph should be DAG
- Every DAG will have atleast one topological ordering.
- If G is cyclic, no ordering exists.



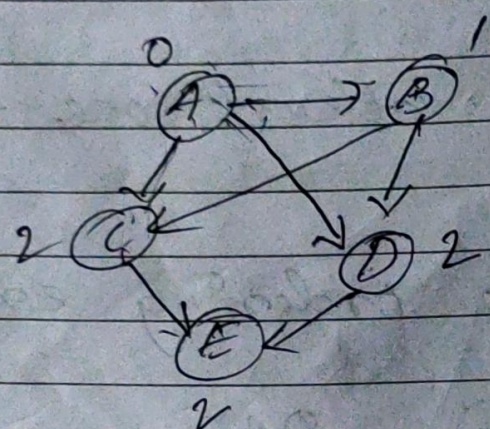
DAG

Topological Sort
- → DFS based
- → Source Removal algorithm

DFS based

1. Call DFS (G) to compute finishing time f(v) for each vertex. v.
2. As each vertex is finished insert into the front of linked list.
3. Return the linked list of vertices.

Source Removal Algo



Indegree
↳ Total no. of incoming ~~ver~~ edges to a vertex

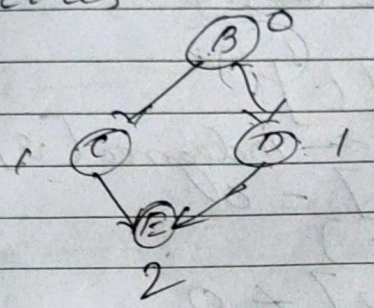
Outdegree
↳ Total no. of outgoing edges from a vertex

Step 1: Write indegree of each vertex

2: If there is any vertex have not indegree 0, then there is a cycle and donot discover topological ordering.

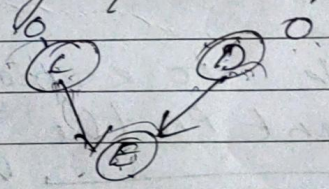
3: Check vertex have indegree = 0 for topological ordering.

4: Since vertex A has indegree 0, remove vertex A and its associated edges and update indegree of other vertices.



Keep [A]

5 look at graph & process continuous



[A B]

[A B C]

0 E

A B D C E or

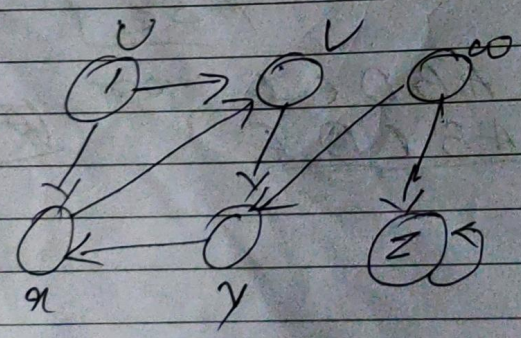
A B C D E

[A B C D] E

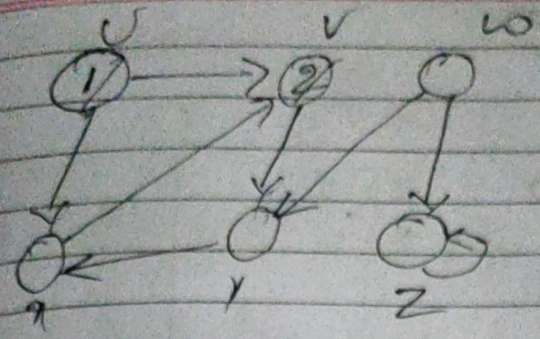
Topological Sort (G)

```
for each vertex U ∈ V
do indegree[U] ← 0
for each vertex U ∈ V
do for each V ∈ Adj[U]
do indegree[V] ← indegree[V] + 1
Q ← ∅
for each vertex U ∈ V
do if indegree[U] = 0
then Enqueue (Q, U)
while Q ≠ ∅
do U ← DEQUEUE (Q)
Output U
for each V ∈ Adj[U]
do in degree[V] ← indegree[V] - 1
If in degree[V] = 0
then Enqueue (Q, V)
do if in degree[U] ≠ 0
Report there is a cycle
```

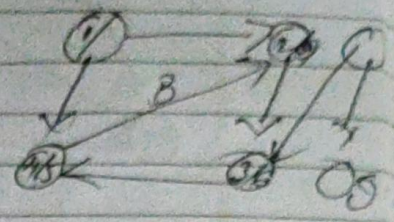
DFS Example to explore all the vertices present in the graph



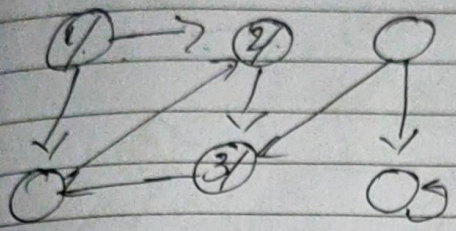
b


u
v
w
1
2
x
y
z

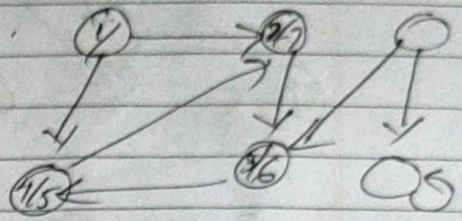
g.


1
2
B
4/8
3/6

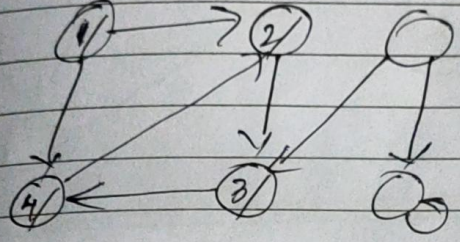
c


1
2
3

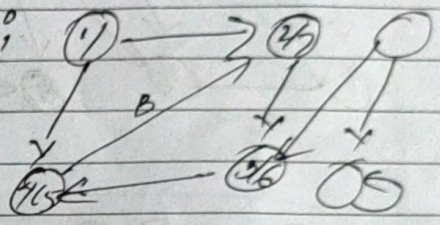
h.


1
2/7
4/5
4/6

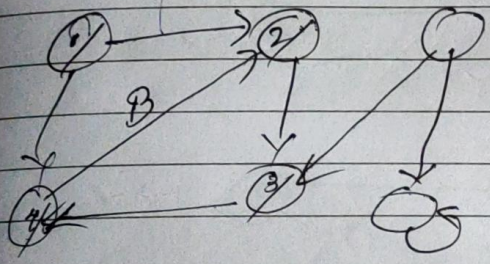
d


1
2
4
3

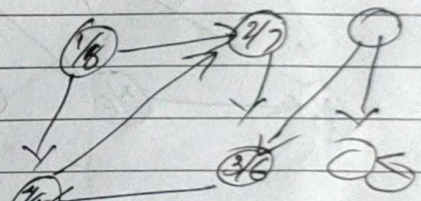
i


1
2/7
B
4/5
4/6

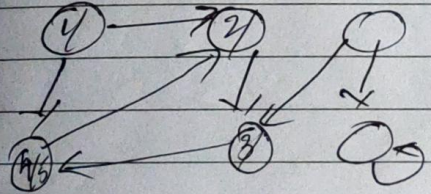
e


1
2
B
4/6
3

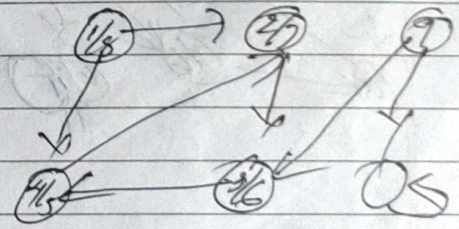
j


1/8
2/7
3/6
4/5

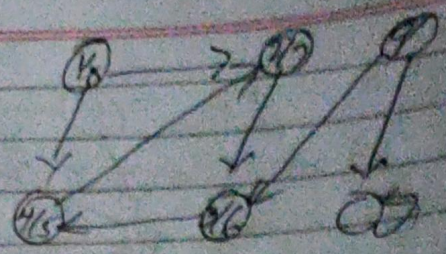
f


1
2
4/5
3

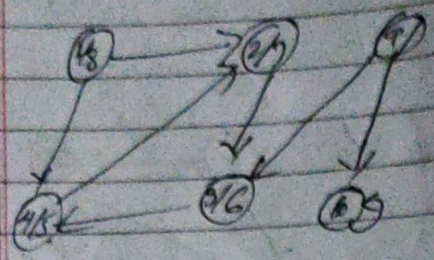
k


1/8
2/7
4/5
4/6

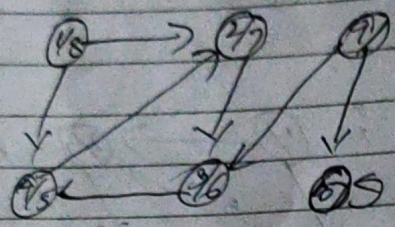
l



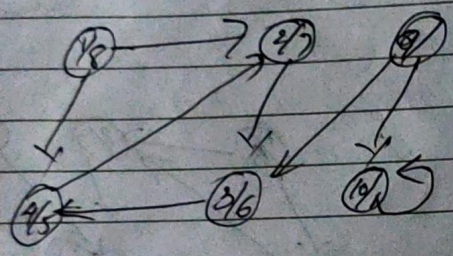
m



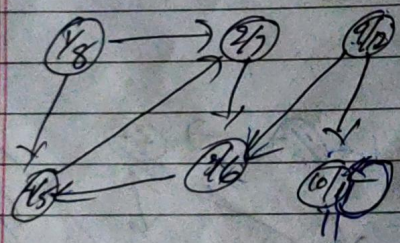
n



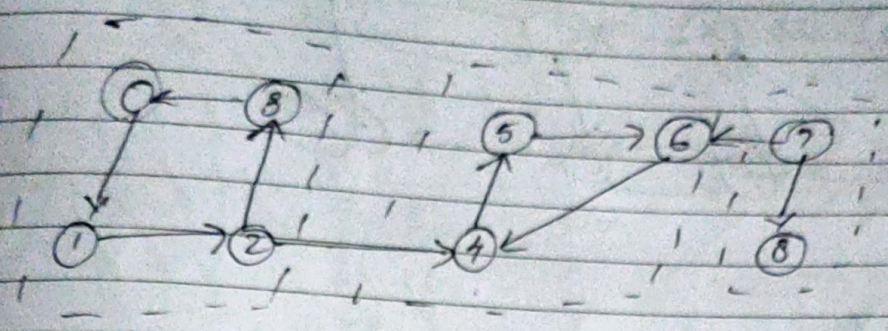
o



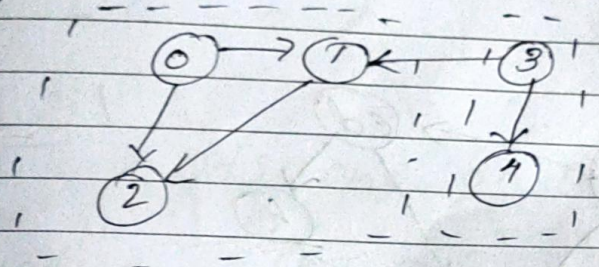
p



Strongly Connected Components



* Strongly connected component of a directed graph is a maximal strongly connected subgraph.



2 Subgraph

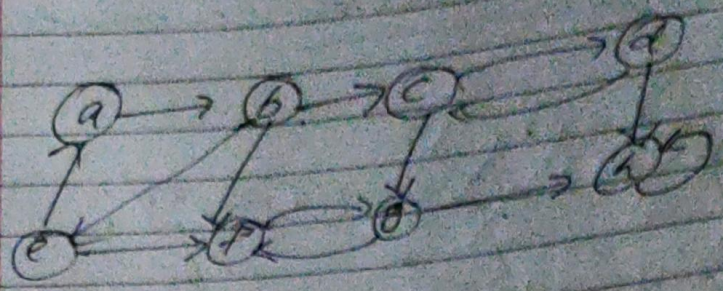
Kosaraju's Algorithm

Strongly - Connected Component (G)

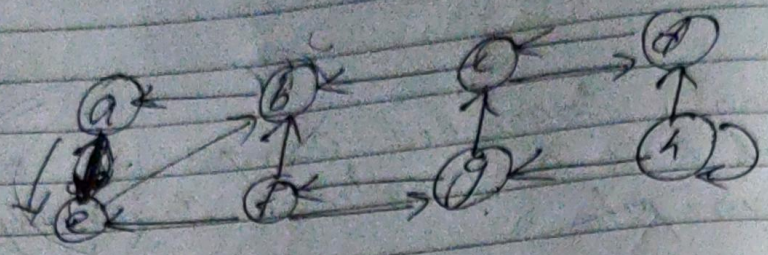
1. Call DFS (G) to compute finishing time f(u) for each vertex u.
2. Compute G^T i.e. Transpose
3. Call DFS (G^T)
4. Produce output of DFS forest as separate SCC

ex:

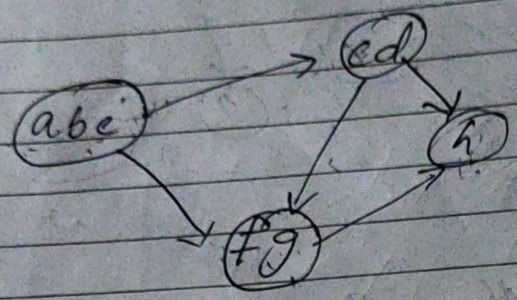
a.



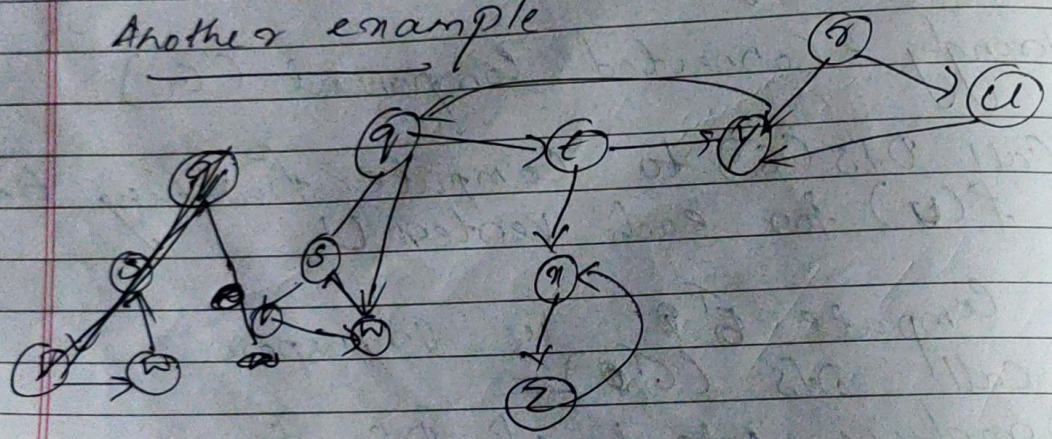
b. G^T



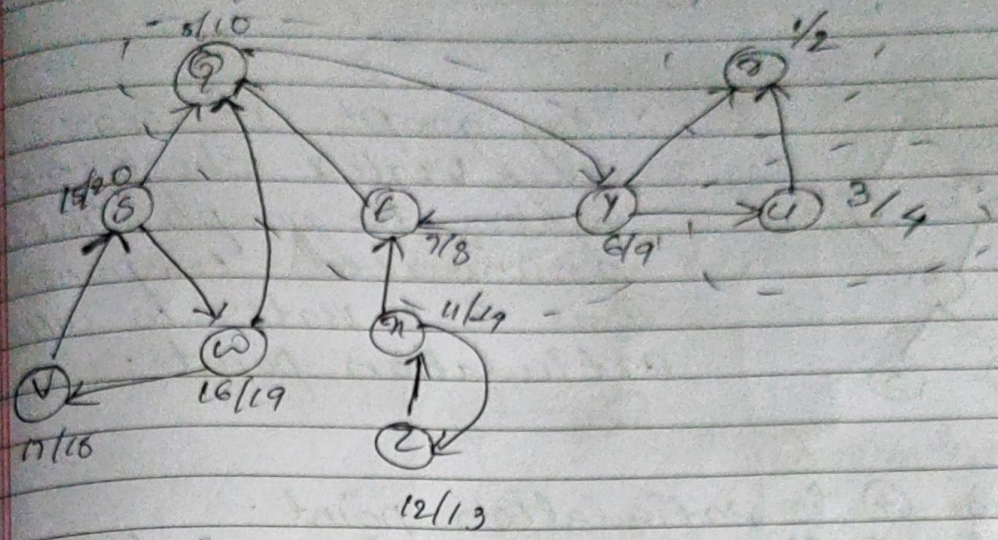
c. G^{SC}



Another example



Transpose



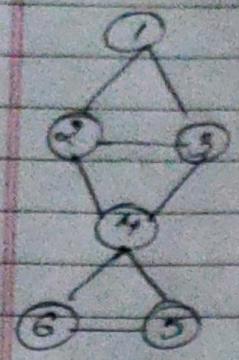
Call DFS (G^T)

Analysis

2 DFS; one on G and other on G^T

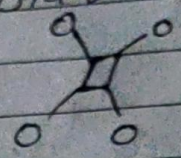
adjacency list $\Theta(V + E)$
Matrix $O(V^2)$

Biconnected Component



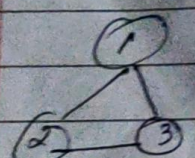
* Any graph, if there is any vertex whose removal makes the graph into disconnected component. Thus that vertex is called articulation point.

* Here ④ is articulation point. Articulation point is not good for a network. eg: Star N/W



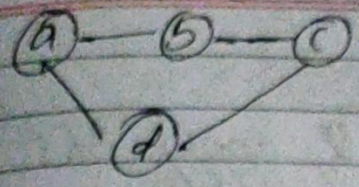
* So a vertex V of graph G is called an articulation point if the deletion of V together with all edges incident on it disconnects the graph into two or more component.

Biconnected graph



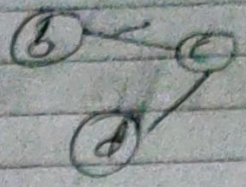
A biconnected graph is a connected graph that has no ~~at~~ articulation point.

⑥ - ⑤



"a deleted"

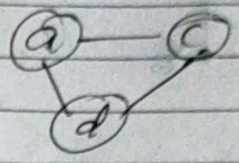
↳



* Graph remains connected

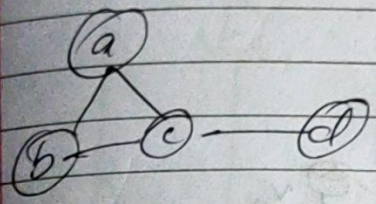
"delete b"

↳

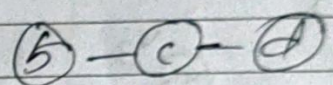


Graph remains connected

eg:

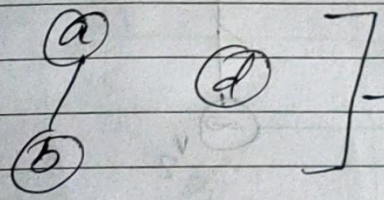


$\xrightarrow{\text{delete } a}$



Graph remains connected.

* delete c



Two separate graphs so 'c' is an articulation point.

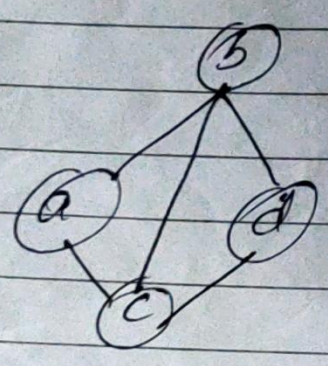
Spanning Tree

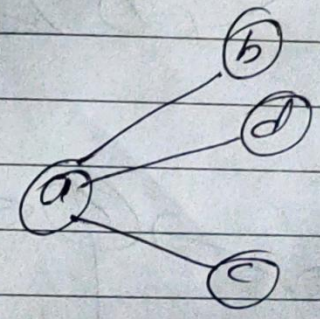
Given $G = \{V, E\}$ where $V = \{V_1, V_2, V_3 \cdots V_n\}$
$E = \{E_1, E_2, \cdots E_n\}$

$E_{ij} = (V_i, V_j)$ where V_i and V_j are adjacent vertex

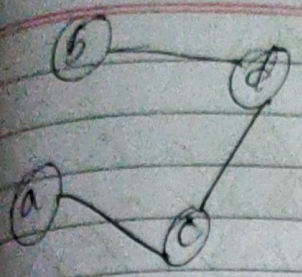
Let $G = (V, E)$ be an undirected connected graph. A subgraph $T = \{V', E'\}$ of G is a spanning tree of G, if and only if T is a tree. We can construct any no. of spanning tree from a graph.

A spanning tree should have n vertex and some $(n-1)$ edges without any cycle.

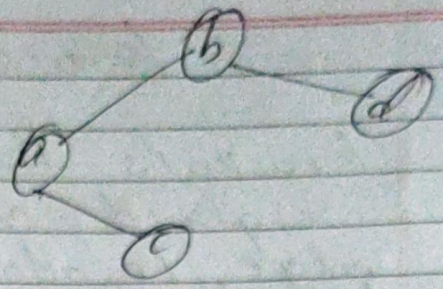




Sp1



SP2



SP3

* If the graph is weighted then a spanning tree with minimum cost is called minimum cost spanning tree

Conditions of Spanning tree

i $V' = V$, ie. no. of vertices is equal in main and sub graph

ii $E' \subseteq E$ ie. E' is the subset of E if n edges in E, then $(n-1)$ edges in E'

iii without any cycle or closed path.
S is the subset of G where $S = \{V', E'\}$
$S \subseteq G$.

* In MSP, the two algorithm commonly used are:
  - ↳ Kruskal's Algorithm
  - ↳ Prims Algorithm

(1) Kruskal's Algorithm

Kruskal's find a MST for a connected weighted graph, it finds a sub forest by finding all the edges that connect any two trees in the forest. edge to add the growing forest

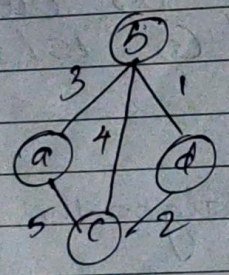
3 Operation in kruskal

i MAKESET: The MAKE-SET Operation determine or arrange each tree in non-decreasing order.

ii FIND-SET: Find-SET Operation determine two vertices u and v belong to same tree by testing whether

Find-set u = Find-Set v

iii To combine trees kruskal's call a function known as UNION

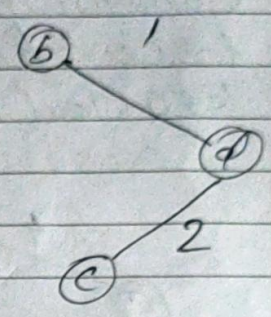


Step 1: Use MAKE-SET operation and write in non-decreasing order

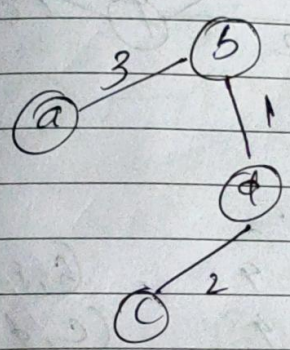
(b,d) = 1
(c,d) = 2
(a,b) = 3
(b,c) = 4
(a,c) = 5

Steps : (i)

Step (ii)



Step (iii)



Minimum cost = 1+2+3 = 6

Algorithm

MST - kruskal (G, w)
Step1 : A = ϕ
Step2 : for each Vertex V ∈ V[G]
Step3 : MAKE-SET (V)
Step4 : Sort the edges into non-decreasing order by weight

Step5: for each edge $(u,v) \in G$, taken in non-decreasing order

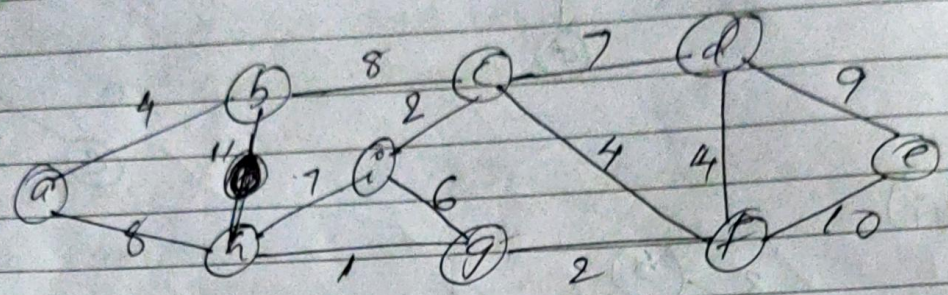
Step6: If FIND-SET $(u) \neq$ FIND-SET (v)

Step7: $A = A \cup \{u,v\}$

Step8: union (u,v)

Step9: return A

2. Construct minimum cost spanning tree for the following graph using kruskal's method.



MAKE-SET

(h, g) = 1
(i, f) = 2
(i, c) = 2

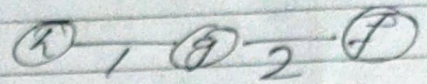
(a, b) = 4
(c, f) = 4
(i, g) = 6

(b, i) = 7
(c, d) = 7
(a, h) = 8
(b, c) = 8
(d, e) = 9

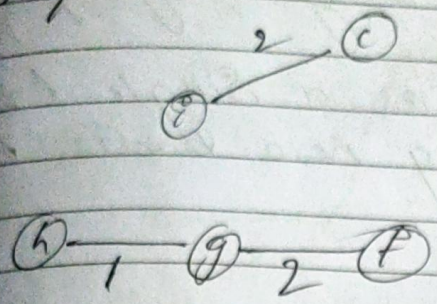
(f, e) = 10
(b, h) = 11
(f, d) = 14

Step 1

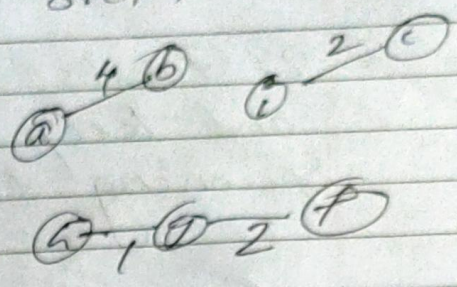
Step 2



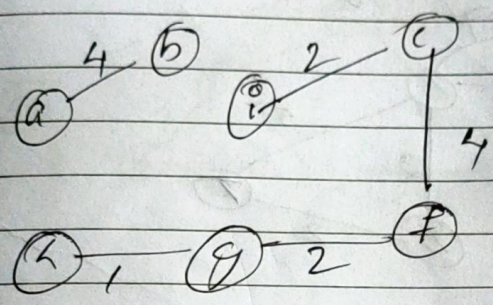
Step 3



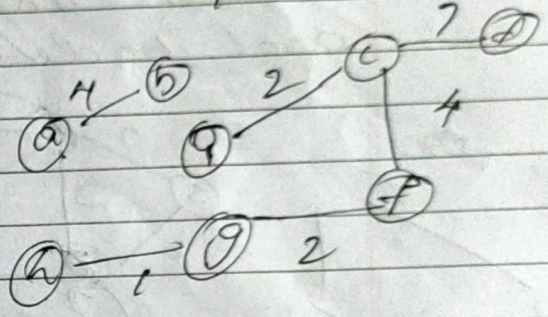
Step 4



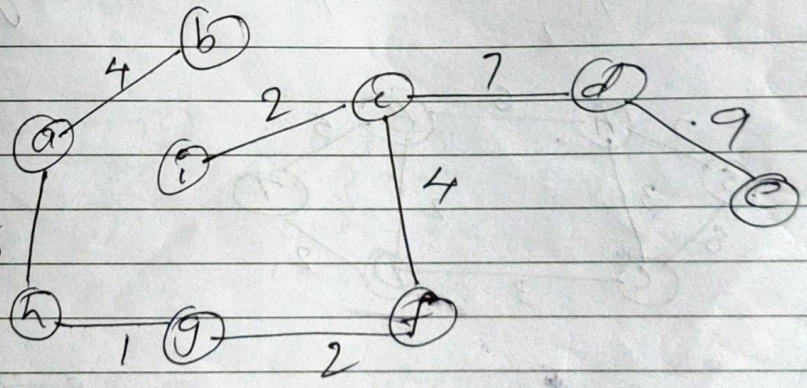
Step 5



Step 6



Step 7

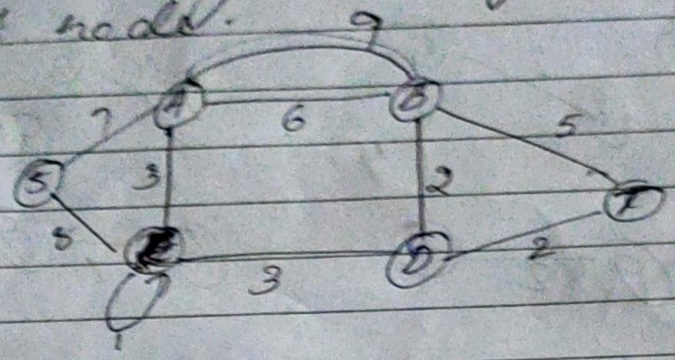


Total Cost = 37

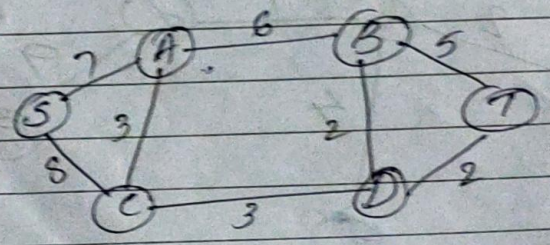
Prims Algorithm

- Prims Algorithm is used to finding shortest path in a given graph.
- It starts the node as a single tree and keep on adding new nodes to spanning tree.
- Remove loops and parallel edges
- Choose any arbitary node as root node.

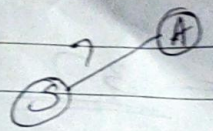
eg:



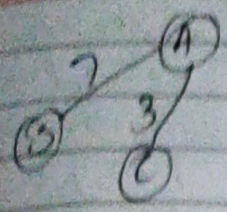
Step 1. Remove loops and parallel edges



$S \to A = 7$ ✓
$S \to C = 8$

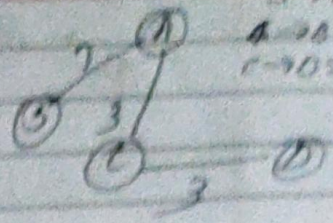


iii



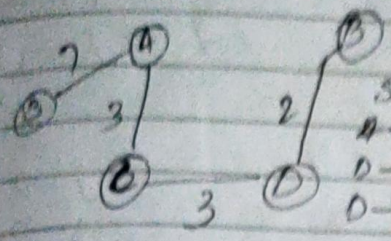
$S \to C = 8$
$A \to B = 6$
$A \to C = 3$ ✓

iv



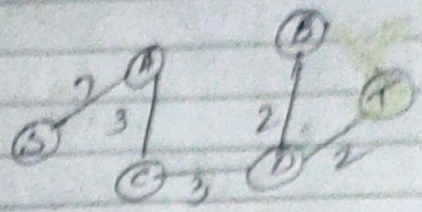
$S \to C = 8$
$A \to B = 6$
$C \to D = 3$ ✓

v



$S \to C = 8$
$A \to B = 6$
$D \to B = 2$ ✓
$D \to T = 2$

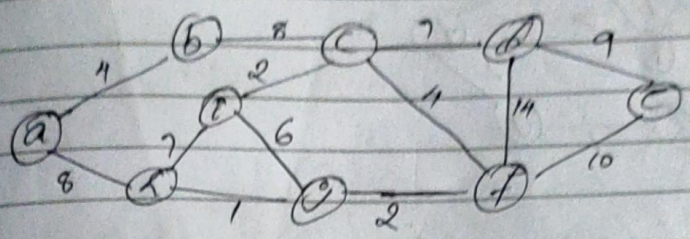
vi



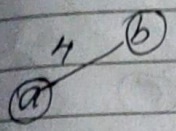
$S \to C = 8$
$A \to B = 6$
$D \to T = 2$ ✓
$B \to T = 5$

Minimum Cost = 17

Q.2

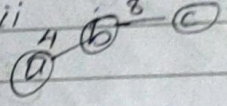


i



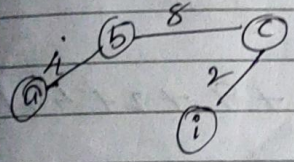
$a \to b = 4$ ✓
$a \to h = 8$

ii



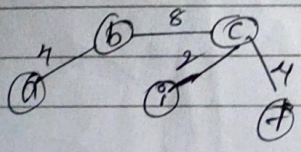
$a \to h = 8$
$b \to c = 8$ ✓

iii

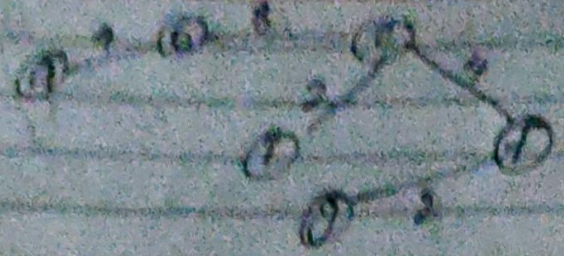


$a \to h = 8$
$c \to d = 7$
$c \to i = 2$ ✓
$c \to f = 4$

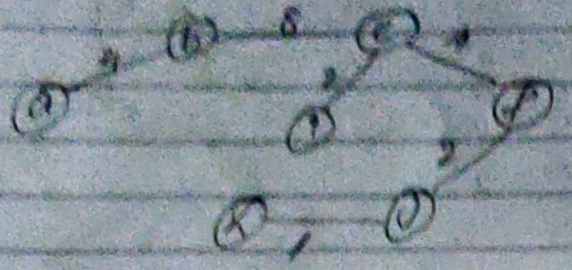
iv



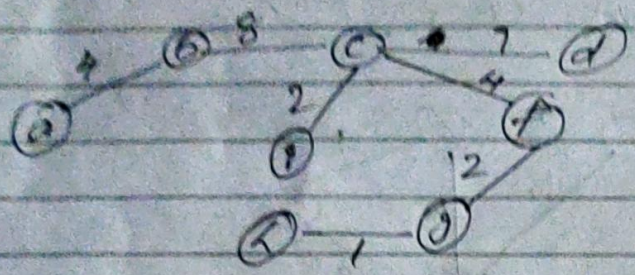
$a \to h = 8$
$c \to d = 7$
$c \to f = 4$ ✓
$i \to h = 7$
$i \to g = 6$

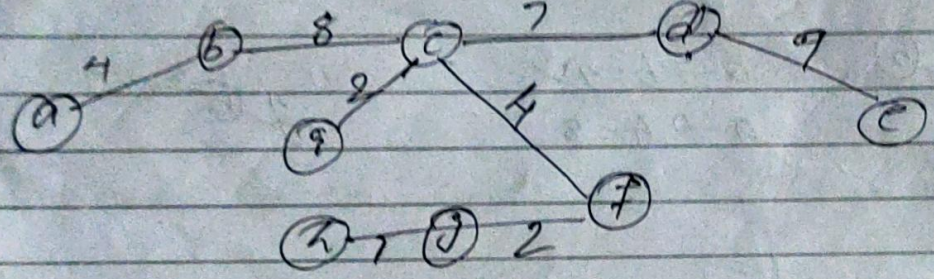


a → h = 8
c → d = 7
f → h = 7
i → j = 6
f → d = 14
f → e = 10
f → j = 2 ✓



j → k = 10
a → h = 8
c → d = 7
g → h = 7
f → d = 14
f → e = 10



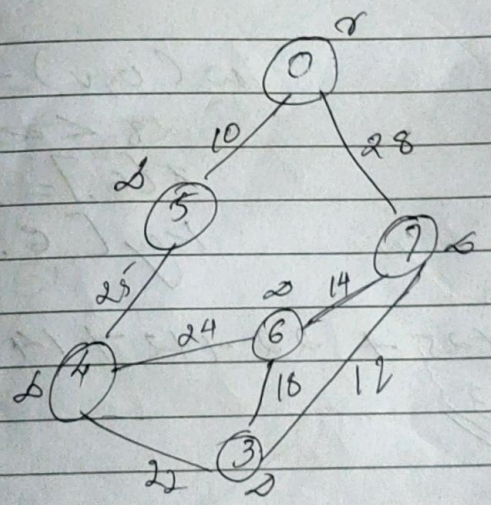


Minimum Cost = 4 + 8 + 7 + 9 + 2 + 4 + 1 + 2
= 37

Algorithm

MST Prims (G, w, r)

```
for each u ∈ V[G]
  do key[u] ← ∞
π[u] ← NIL
key[r] ← 0
    Q ← V[G]
  While Q ≠ ∅
    do u ← extract min (Q)
       for each v ∈ adj [u]
           do if v ∈ Q and w[u,v] < key[v]
                then π[v] ← u
                key[v] ← w(u,v]
```



Q

Q1 $w(u,v) = 10$

$10 < \infty$

$\pi[5] = 0$ ✓

$key[5] = \underline{\underline{10}}$

$w(u,v) = 28$

$28 < \infty$

$\pi[1] = 0$

$key[1] = \underline{\underline{28}}$

1) $w(u,v) = 25$
$25 < \infty$
$\pi[4] = 5$
$key[4] = 25$

✓ $w(u,v) = 24$
$24 < \infty$
$\pi[6] = 4$
$key[6] = 24$

$w(u,v) = 22$
$22 < \infty$
$\pi[3] = 4$
$key[3] = 22$ ✓

✗ $w(u,v) = 18$
$18 < \infty$
$\pi[6] = 3$
$key[6] = 18$

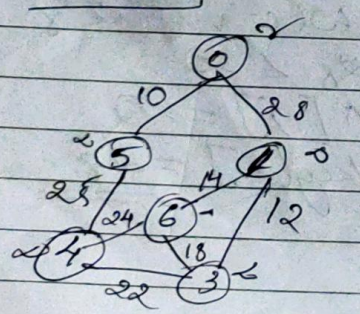
$w(u,v) = 12$
$12 < \infty$
$\pi[1] = 3$
$key[1] = 12$ ✓

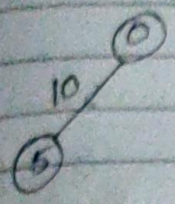
✗ $w(u,v) = 14$
$14 < \infty$
$\pi[0] = 1$
$key[0] = 14$

$w(u,v) = 28$
$28 < \infty$
$\pi[6] = 1$
$key[6] = 28$

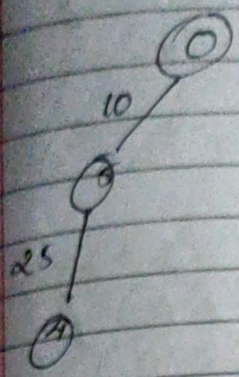
$M.C = 10 + 25 + 22 + 12 + 14$
$= 83$

Otherway

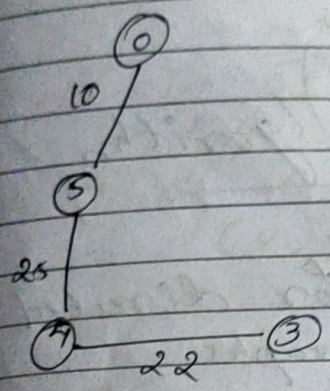




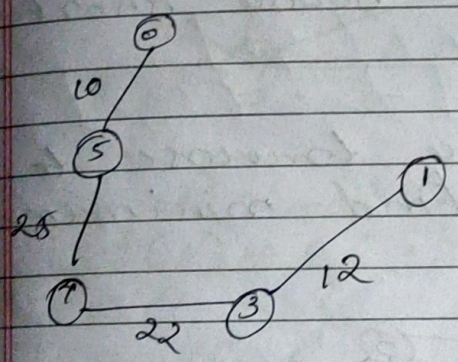
0 — 1 = 28
0 — 5 = 10 ✓



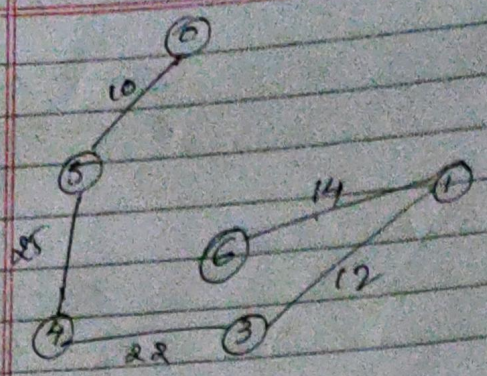
0 — 1 = 28
5 — 4 = 25 ✓



0 — 1 = 28
4 — 6 = 24
4 — 3 = 22 ✓



0 — 1 = 28
4 — 6 = 24
3 — 6 = 18
3 — 1 = 12 ✓



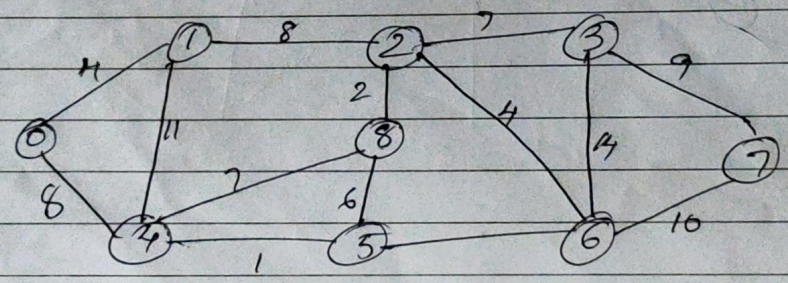
0 – 1 = 28
4 – 6 = 24
8 – 6 = 18
1 – 6 = 14
"

MC = 10 + 25 + 22 + 14 + 12
= 83

Dijkstra's Algorithm

- This algorithm used for directed as well as undirected graphs
- Also known as Single-Source Shortest path algorithm
- From a single source traverse to other vertices to find minimum cost.

ea:



Make all the distance be infinity (∞)

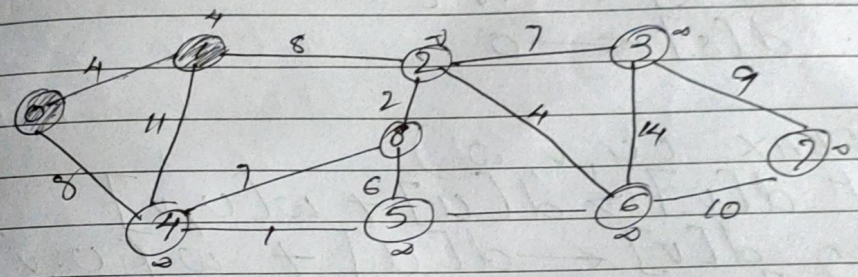
Distance Calculation

if $d(u) + c(u,v) < d(v)$, then update $d(v)$

ie; $d(v) = d(u) + c(u,v)$

$u = 0$
$d(u) = 0$
$c(u,v) = c(0,1)$
$= 4$

$d(u) + c(u,v)$
$= 0 + 4$
$= 4 < \infty$
$\therefore d(v) = d(1) = 4$



Out of remaining node
$d(1) + c(1,2)$
$= 4 + 8$
$= 12$

$d(1) + c(1,4)$
$4 + 11$
$= 15$

like so on...

Algorithm

Dijkstra (G, w, s)

```
Initialize Single-Source (G, s)
    S ← ∅
Q ← V[G]
while Q ≠ ∅
    do u ← extractmin(Q)
        S ← S ∪ {u}
        for each vertex v ∈ adj[u]
            do RELAX (u, v, w)
```

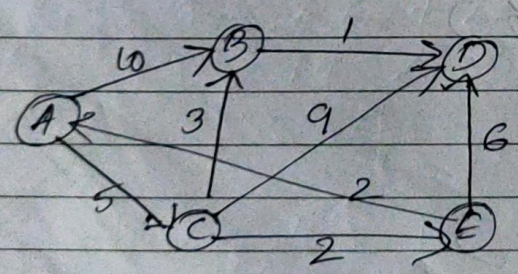
Initialize Single-Source (G, s)

```
for each vertex v ∈ V[G]
    do d[v] ← ∞
    π[v] ← NIL
d[s] ← 0
```

RELAX (u, v, w)

```
if d[v] > d[u] + w[u, v]
then d[v] ← d[u] + w[u, v]
π[v] ← u
```

ex: 2



	A	B	C	D	E
A	$\boxed{0}$	∞	∞	∞	∞
C		10	$\boxed{5}$	∞	∞
E		8		14	$\boxed{7}$
B		$\boxed{8}$		13	
D				$\boxed{9}$	

Shortest distance

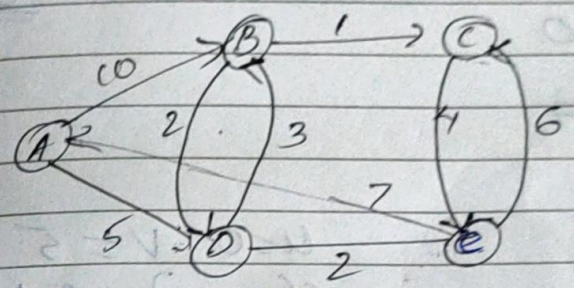
A → B = 8

A → C = 5

A → D = 9

A → E = 7

Ex:2



	A	B	C	D	E
A	$\boxed{0}$	∞	∞	∞	∞
D		10	∞	$\boxed{5}$	∞
E		8	∞		$\boxed{7}$
B		$\boxed{8}$	13		
C			$\boxed{9}$		

A → B = 8

A → C = 9

A → D = 5

A → E = 7