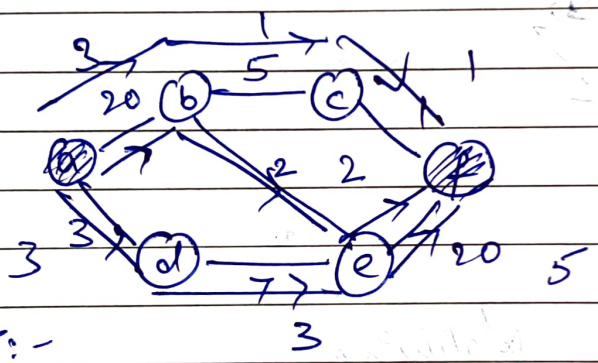
Network flows

1) An algorithm for network flows (maximum)
2) Network flow as a design paradigm

Input :- A directed weighted graph $G = (V, E, c)$

1) A capacitated N/w (graph with edges having +ve input)
2) Source, destination pair (Two vertices in +ve graph)



Output :-

1) It should be a flow
2) It should be a maximum flow

Def (Flow)

A flow is a function from ($E \Rightarrow$ set of directed edges)

$f : E \rightarrow R^{\oplus}$ st

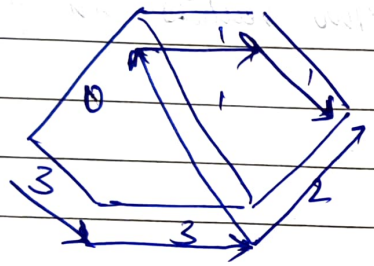
1) Capacity constraints (not more than value
2) conservation constraints (incoming = outgoing)

1) Capacity constraints

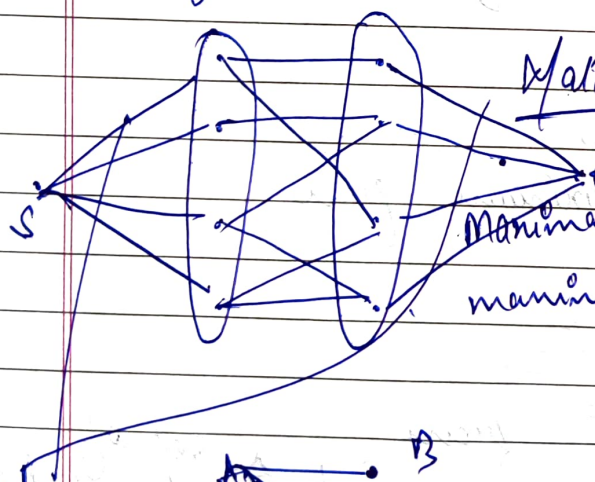
$\forall e \quad f(e) \leq c(e)$

2) Conservation criterion

$\forall v \in V / \{s, t\} \quad \sum_{e \in \text{incoming edge at } v} f(e) = \sum_{e \in \text{outgoing edge at } v} f(e)$.



Bipartite graph

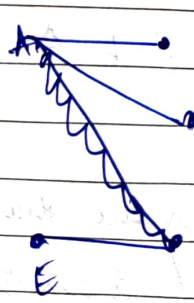


Matching

Maximal (can't add)

maximum (largest size)

add capacity 1



AD (Maximal)

AB, ED (Maximum)

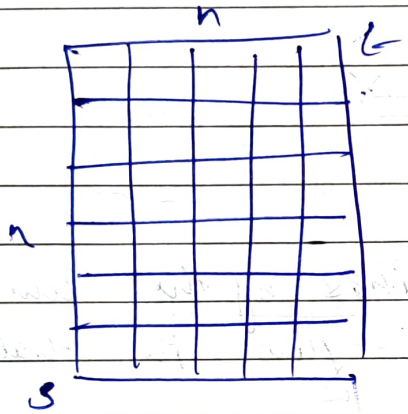
Last class

1. Introduced network flows
2. Solving Bipartite matching using network flows

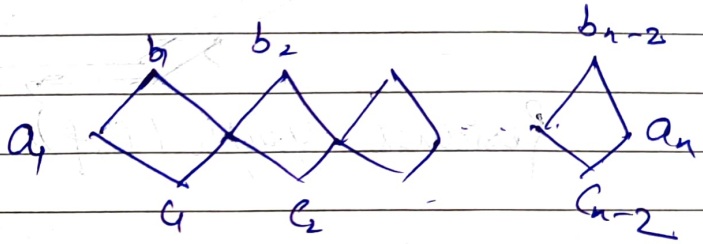
Algorithm for calculating the maximum flow in a ~~capic~~ capacitated n/w

(1) Find an s-t path determine its bottle neck capacity say c_1, route a flow of value c_1 through the path.

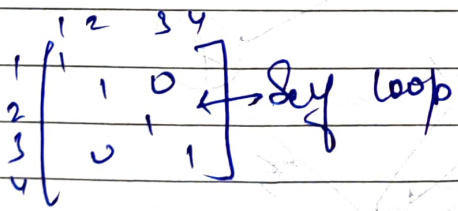
(2)

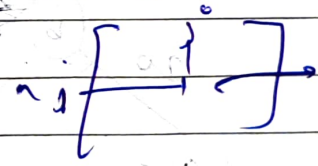


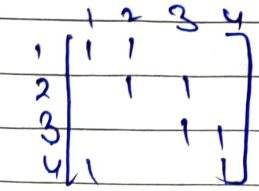
exponentially large



(Refresh adj list & adj matrix)





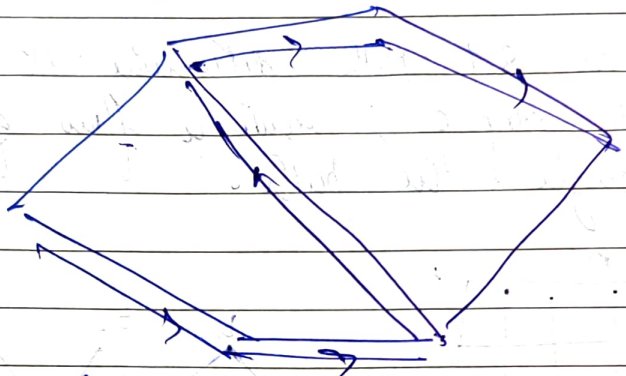


~~calculate~~ the Transitive closure

Remove the used capacities from each edge and then repeat.

(3) Repeat for the reduced graph.

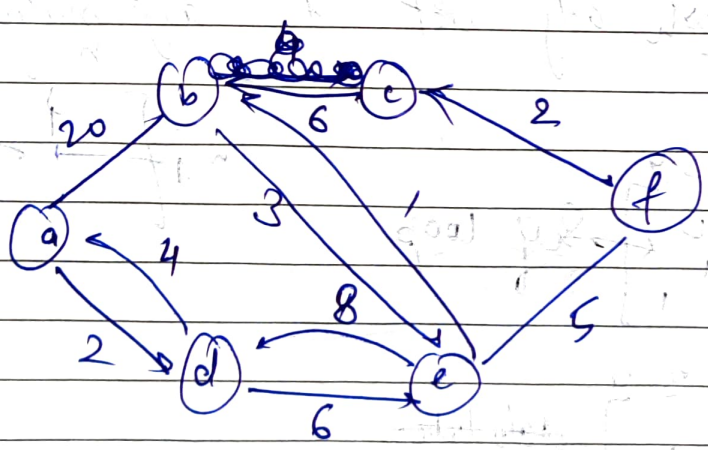
Introduce back edge.



Residual Graph (G_f)

1, Reduce all capacities by the amount corresponding to the flow in that edge

2, Introduce "back edge" equal to the capacity of the flow



Modified Algo

1. f : Initial flow (zero on each edge)

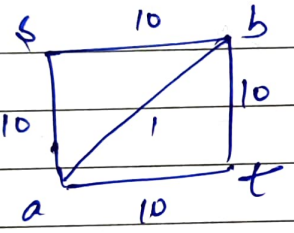
2. & compute an s-t path, determine its bottleneck capacity c

   f = f + c (f + c is the flow obt by inc. f by c on all edges on s-t path)

3. compute the residual graph R(G, f)

4. Repeat

Why it will terminate?



As at each step flow value increases and max value is 20.

Input: A directed weighted graph $G = (V, E, c)$ (capacity)
(Think of c is the weight func $c: E \to R^+$)
$\{s, t\}$ → destination
Source

Output: flow network having max. flow.

Flow network
is a func from the ~~edges~~

$$f: E \to R^+$$

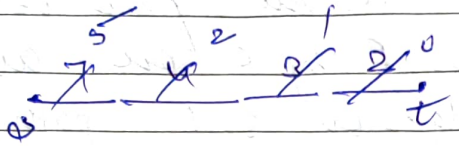
1. Capacity constraints $\forall e \quad f(e) \leq c(e)$
2. conservation constraints $\forall v \in V / \{s,t\} \quad \sum f(e) = \sum f(e)$ (incoming at v = outgoing at v)

Value of a flow

$$T(v) = \sum_{e \mid} f(e) - \sum_{e \mid} f(e)$$

e is an outgoing edge at s.

e is an incoming edge at s



bound on edge weight

bound on edge.

Residual flow network

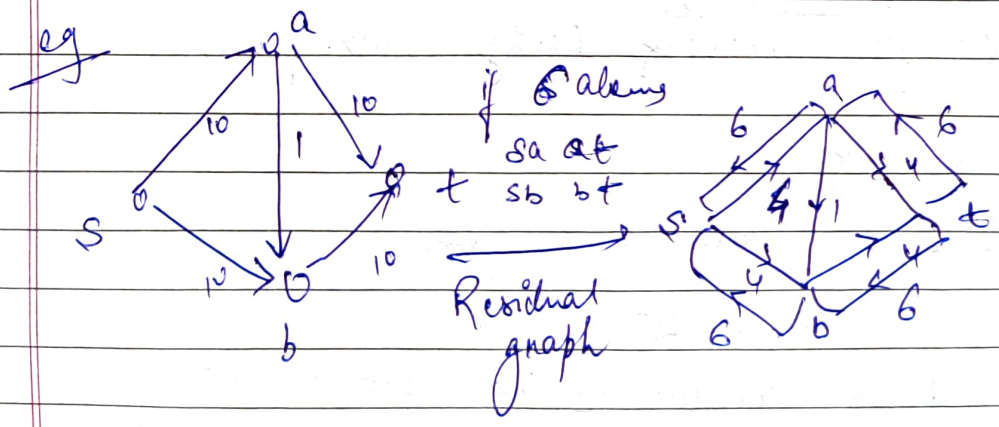
Given $G(V, E, c)$ & a flow f the residual flow n/w G_f has

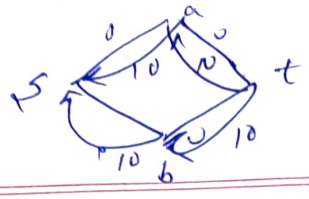
$V_f = V$

If e in an edge in G with capacity c(e) then add e' with cap c'

e'' with cap c''

eg





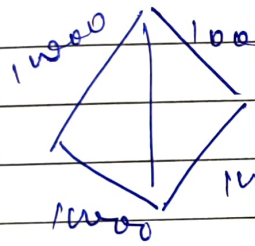
1) Find an s-t path P & compute its bottleneck capacity call f

2) Compute G_f

3) find an s-t path in G_f compute its bn capacity f'

$f = f + f'$

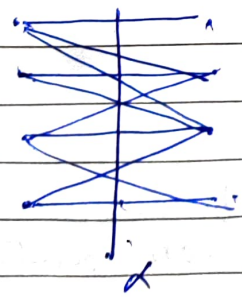
↳ flow in original



exponential time algo

- Maximum flow algorithm
- Bipartite Matching

Max cut algorithm



if we have a general graph m find how large α can be.

Why $2m/3$?

$2m/3$

$k(n-k)$ → max at $n/2$

if fully connected graph

A B

at least there are $m/2$ edge in the cut

Probabilistic proof → as for each edge it can be on cannot be a cut thus on an avg $m/2$ and it also guarantee that there is also value above $m/2$.

→ Any partition of vertex set into two is called a cut

→ Total cuts possible $= \frac{2^n - 2}{2}$

→ cut edge → one side to other

- Algorithm

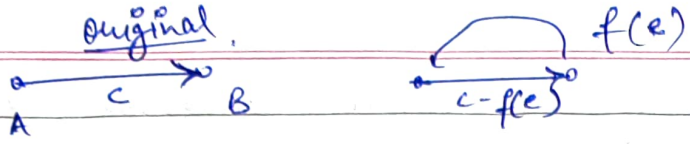
0) $f = 0$
1) Find an s-t path in G
2) Compute the bottleneck capacity of f'
3) Augment the flow f using f'
4) Compute the residual graph

Repeat

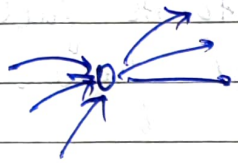
- Let f be the flow in G let G_f be the residual of and f' be the flow corresponding to a bottle neck path in G_f

$f \oplus f'$ is a flow in G.

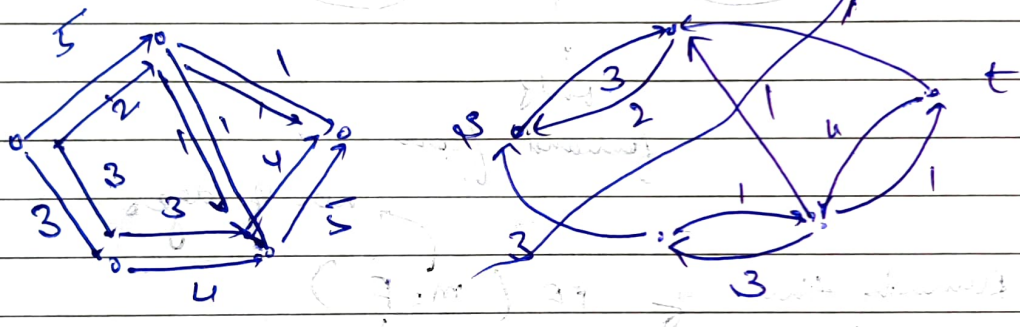
original.



if backward $f' \leq f(e)$ thus $f(e) - f' \geq 0$
and and thus. forward flow ~~do~~ ↓.
Similarly forward



why gcd? ~~so~~ so?



Max flow min cut theorem

~~min~~

s and t on opp side and min

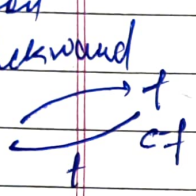
Let say at point algorithm stop and ~~it~~ partition ~~and~~ S,t if we compute the sum of cut value and it matches the flow then that flow is maximum.

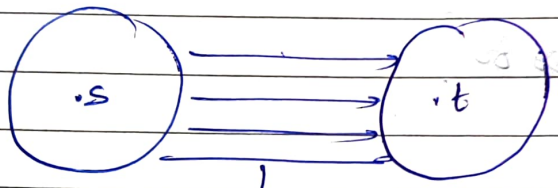
S is a set of all vertices reachable from s

$T = V/S$

all forward edge are running to its full capacity at residual flow.

for backward





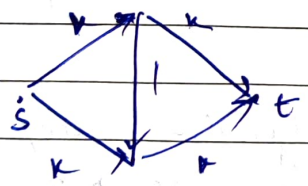
only forward flow

Running time of FF (m . F)

m → no. of edges

F → value of flow

$F \leq \Sigma$ capacities at source



for real also max flow min cut algorithm holds,

100 digit number

$\sqrt{10^{99}} = 10^{48}$

1 sec 10^{10} 10^{12} → 10^{22} comp

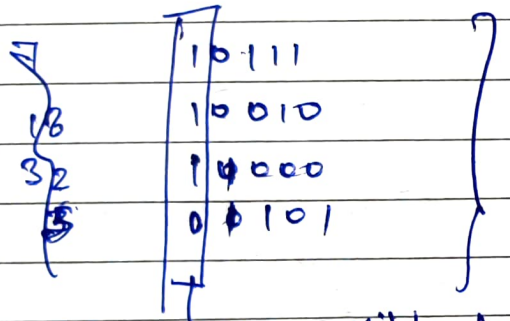
10^{38} sec

10^{31}

10^{24}

$O(\sqrt{n})$ → $O(\sqrt{10^k})$ → $O(10^{k/2})$ → exponential

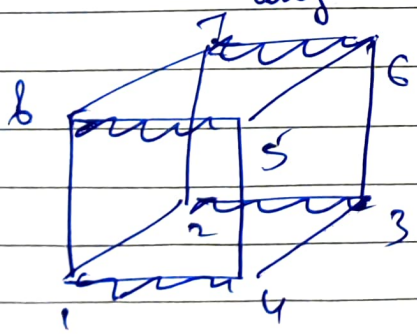
- Current → $m^1 + \dots$ → almost linear



~~$m^2 \log m$~~ ~~$m \cdot (m \log m)$~~

$(\log F) \cdot m(m)$

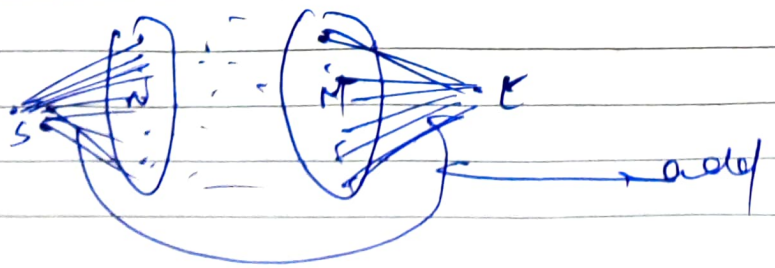
Matching → Coll of edges which do not have any vertices

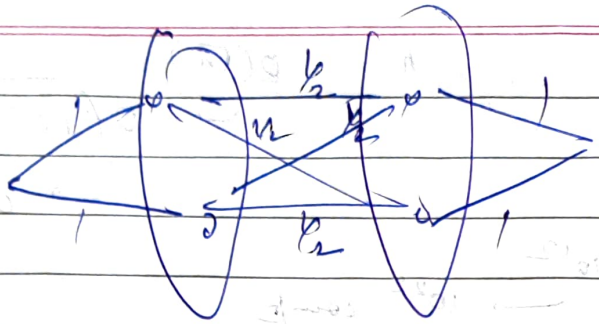


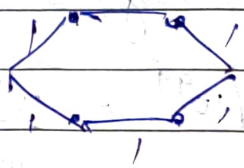
1 → 2
3 → 4
5 → 6
7 → 8

Halls Theorem:

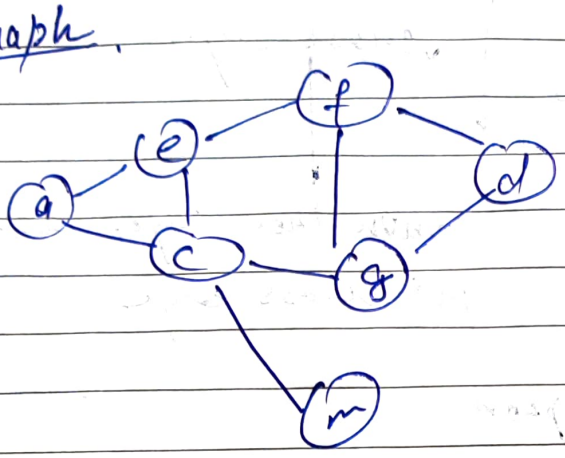
$\min(N, M)$

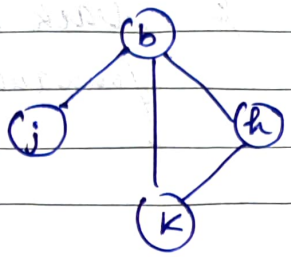



1/2
1/2
1/2
1/2



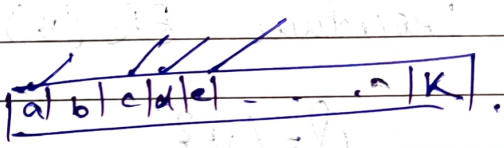
Graph.





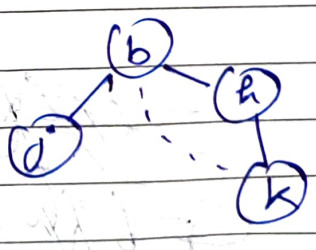
* Searching of Graph → DFS, BFS

G(V,E)

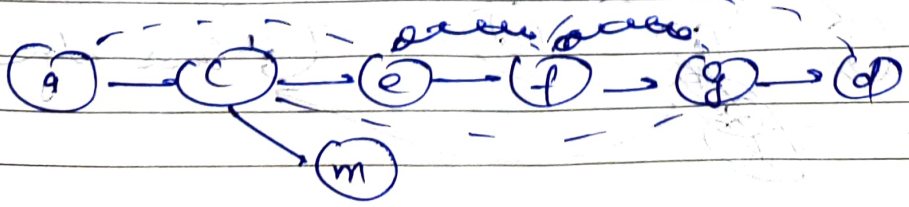


Search (G)
T = ∅
for v ∈ V
mark v "unvisited"
while (∃ v ∈ V marked unvisited)
DFS(v) / BFS(v)

Procedure DFS(v)
mark v as visited
for w ∈ L[v]
if w is "unvisited"
add (v,w) to T
DFS(w)



DFS tree / forest



& back edges
forward edges

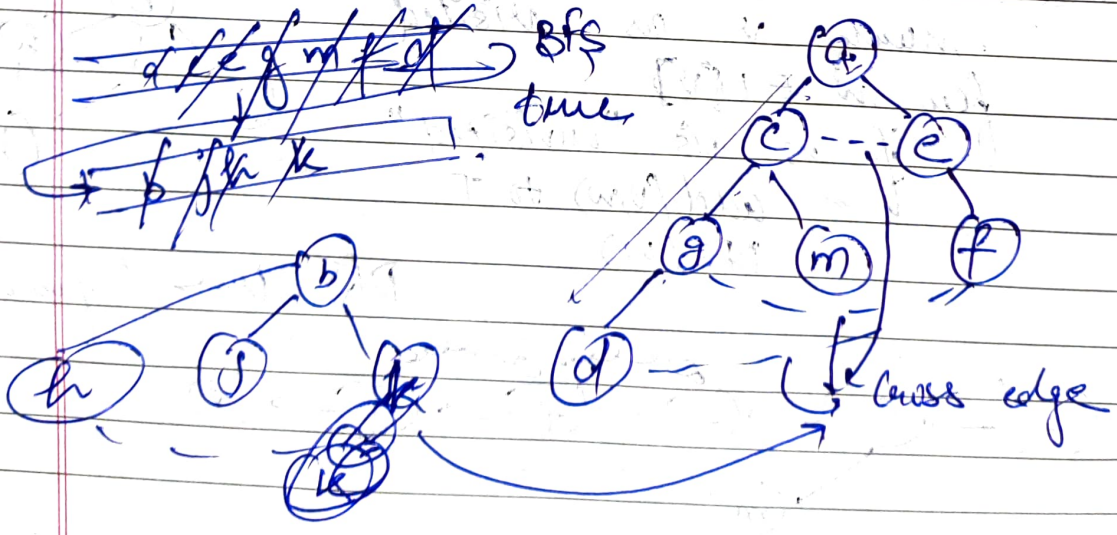
cross edge

non tree edges are not cross edge

$3n + 2m$ operation

Thus $O(n+m)$

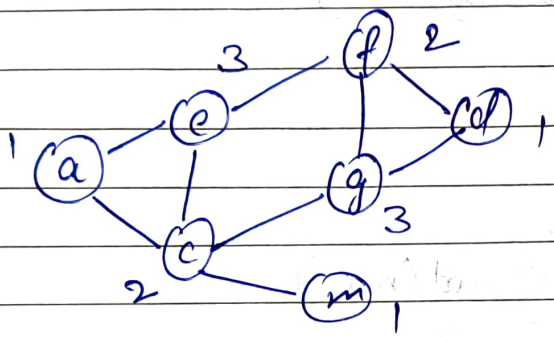
```
Procedure BFS(v)
    Q = {v}
    while (Q ≠ ∅)
        w = dequeue(Q)
        mark w as "visited"
        for v ∈ L[w]
            if (v ∉ Q and v is unvisited)
                enqueue(v)
                Add (w,v) to T
```

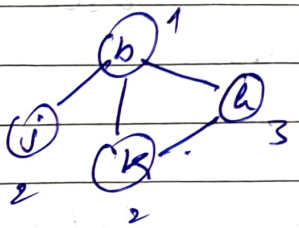


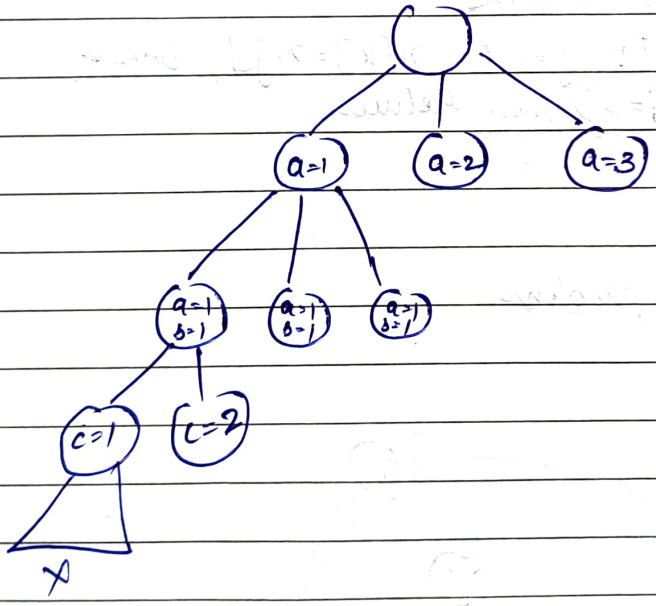
Shortest path in unweighted graph
↳ BFS can be used.

Vertex Colouring

Adjacent vertices to get diff colour





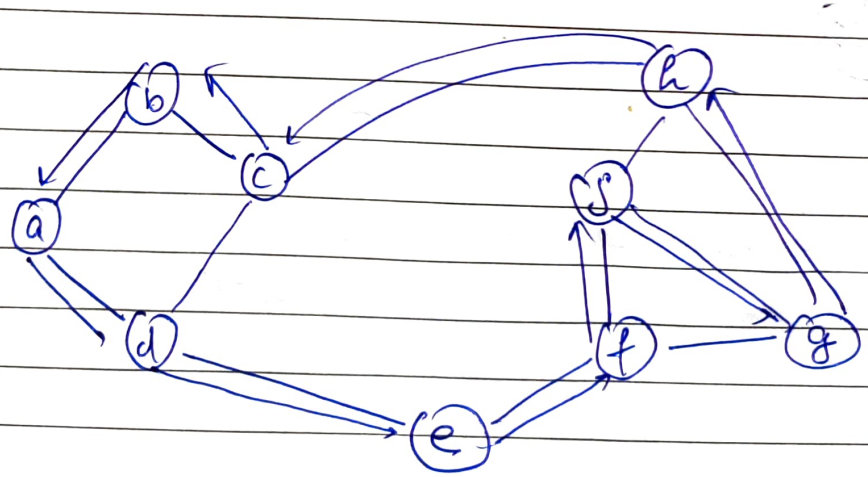


- Vertex Colouring — Vcolouring (1)
  G(V, E) integer m → colour possible

Algorithm Vcolouring(k)
repeat NextColour(k)
if (x[k] = 0) then return
if (k = n) write x[1] .. x[n]
else Vcolouring(k+1)
until false

Algorithm NextColour(k)
repeat x[k] = (x[k] + 1) mod (m+1)
if (x[k] = 0) then return
for j = 1 to k
if (G[k, j] = 1 ∧ x[k] = x[j]) break
if (j = k) then return
until (false)

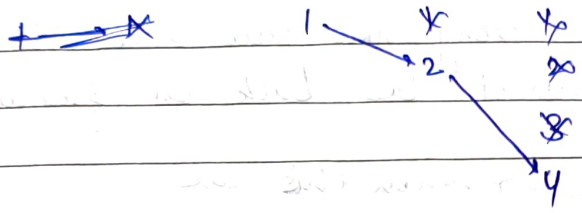
- Hamiltonian cycle problem



n length

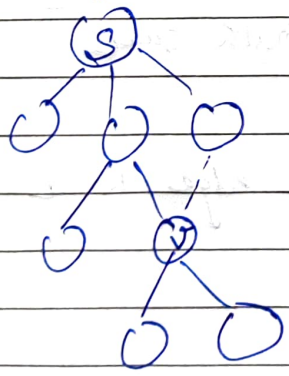
every cycle → a permut of V

Start at Vertex 1



Properties:

1. A Bfs tree is a Shortest-path tree

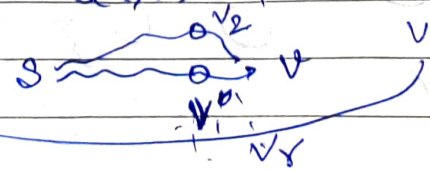


$dist(s,v) = 2$

Basis $dist(s,s) = 0$
s is the root $d[s] = 0$

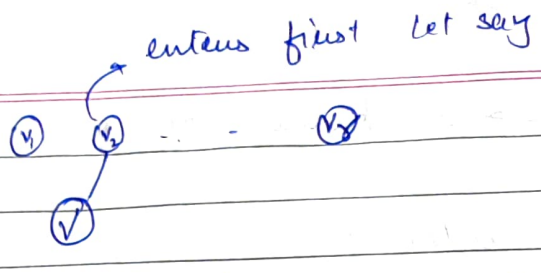
IH Statement true for dist $< d$

IS $\delta(s,v) = d$



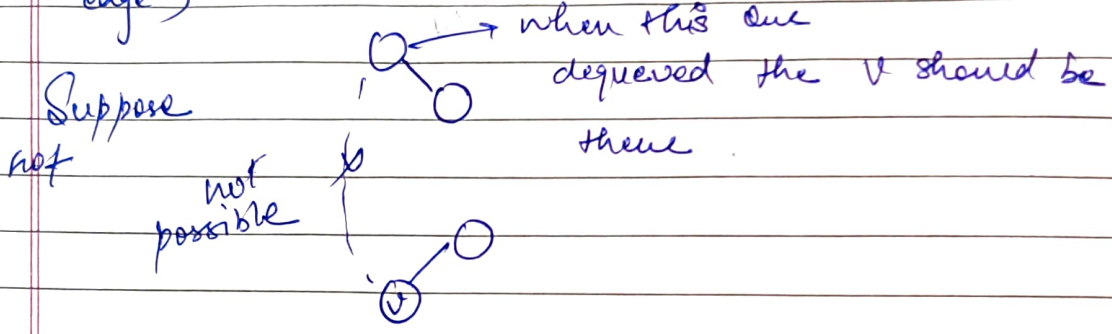
$v_1, v_2 \ldots v_r$

in bfs tree at dist $d-1$

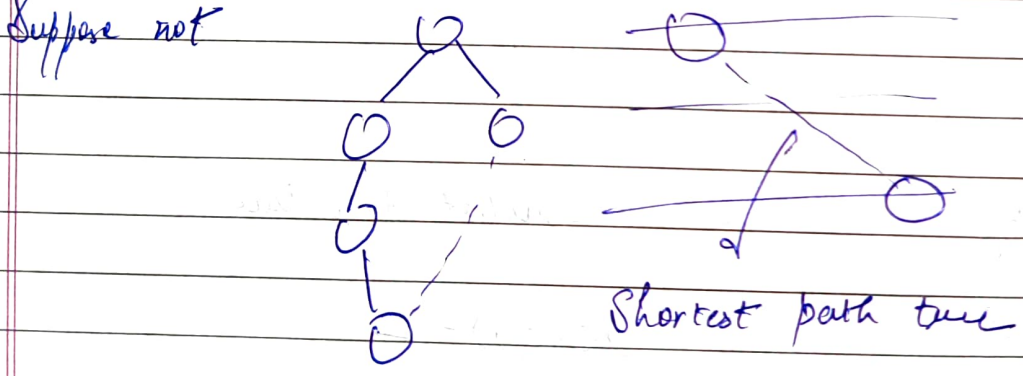


thus a shortest path tree.

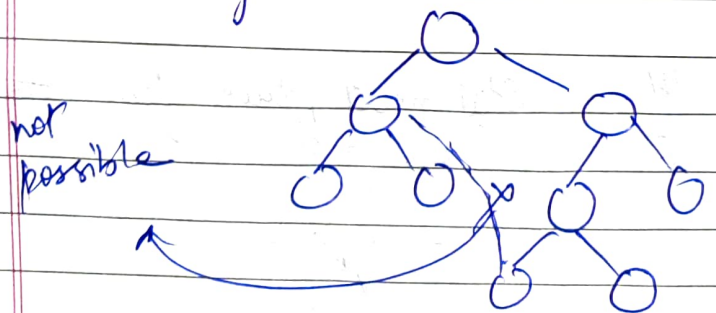
- In a bfs tree every non tree edge is a cross edge (cannot be back or forward edge)



- Every non tree edge is b/w a level or bet adjacent level



- In a DFS tree no non tree edge is a cross edge



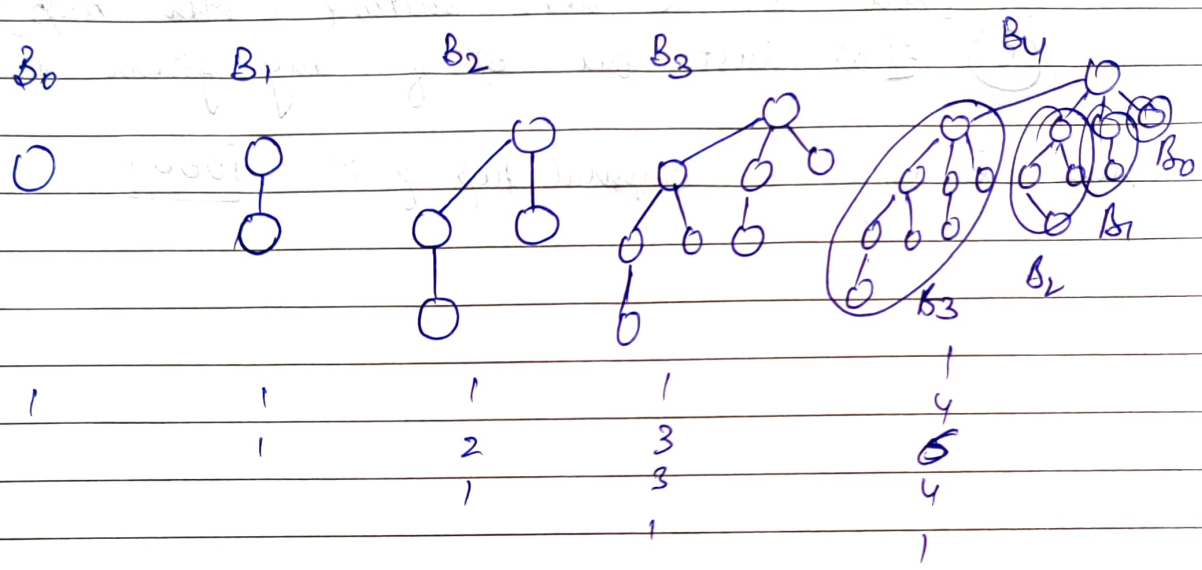
- **Binomial heaps**

Binomial tree → B_0 B_1 B_2

B_0: 1, order 0

B_1: order 1

B_2: order 2



(1) 2^k nodes in B_k

(2) height of B_k is k

(3) at level i of B_k there are $\binom{k}{i}$ nodes

$$\binom{k-1}{i-1} + \binom{k-1}{i} = \binom{k}{i}$$

↓ push down

(4) Root of B_k has k children

$$1 + (k-1) = k$$

(5) If we number the child of B_k we have

B_0 B_1 B_2 - . B_{k-1}

Binary rep of natural no.

5 = 101

(6) Root has the highest degree of k

- A Binomial heap is a set BT s.t
  (1) Each BT is heap ordered (Min heap)
  (2) ∃ at most one BT of any given order

there is only 1 Binomial rep of 17 10001

Binary rep of natural no.

$5 = 101$

(6) Root has the highest degree of K

- A Binomial heap is a set BT s.t
  (1) Each BT is heap ordered (Min heap)
  (2) ∃ at most one BT of any given order

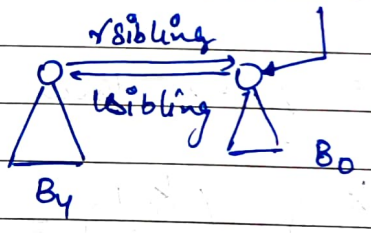
there is only 1 Binomial rep of 17 10001

- height of a BH of n nodes $\lfloor \log n \rfloor + 1$

p: parent pointer
key: key value
degree: # children
child: the child with the largest degree
lsibling: the sibling with the next higher degree
rsibling: ——→ —— lower

—

eg 17

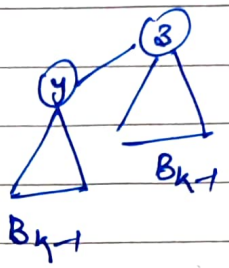


Make Heap()
[Allocate an object H
head[H] = NIL return

O(log(n)) Fin Min(H)
Access the roots using head[H]
return the smallest key among those in the roots

- y and z are roots of B_{k-1}

  link them so that z became the root



link (y, z)

P[y] = z

rsibling [y] = rchild [z]

lsibling [child [z]] = y

child [z] = y

degree [z] ++

B_5^3 B_4^3 B_3^3

B_5' B_4' B_3' B_2' B_0'

B_5^2 B_3^2 B_2^2

B_6^3 B_5' — B_3' — — B_0'

```
  1 1 1 1 0 1
+ 1 0 1 1 0 0
-------------
1 1 0 1 0 0 1
-------------
```

Merge of two heaps in O(log n)

↳ Union

- Insert (H, x)

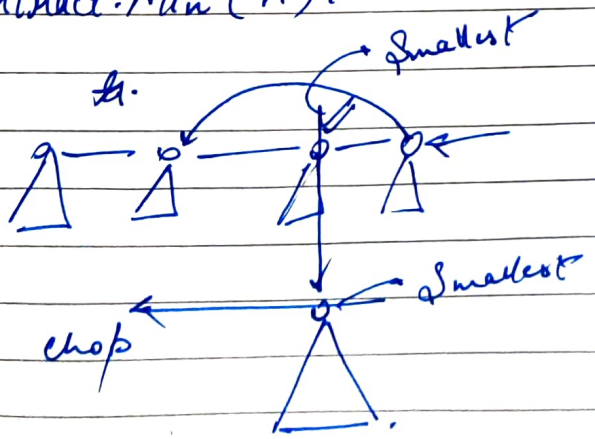
O(log n)

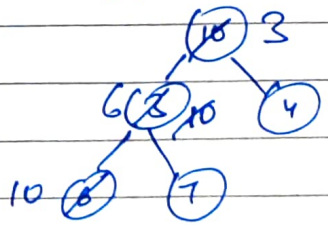
Make heap (H_1)

Insert x in H_1 → "B_0"

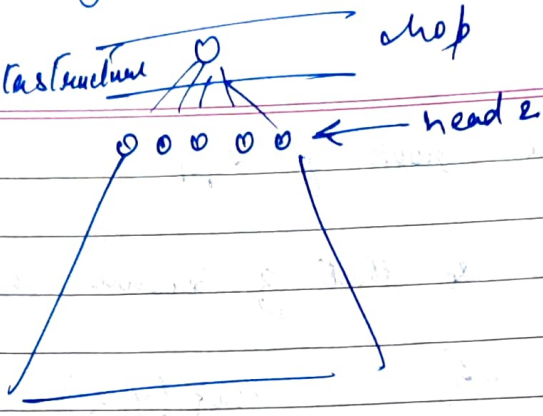
Union (H, H_1)

- Extract-Min (H) — replace with leaves in original heap and then apply heapify





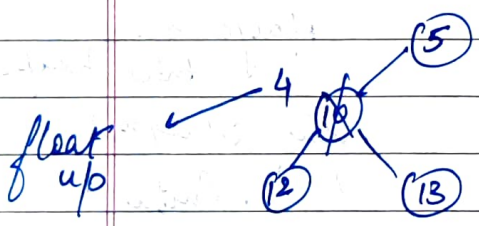
This is a priority queue
↳ not a
Searchable data structure



Take Union

Decrease key (H, x, k)

key[x] = k O(log n)



Delete (H, x)

O(log n)

Decrease key (H, x, -∞)
Extract-Min (H)

Amortised Analysis

K bits A[k-1] .... A[0]

0 - - - 0

0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0

n steps

- Agg Aggregation

LSB flips every time

2 LSB — 2 steps

3 — 4 steps

n

$+\lfloor n/2 \rfloor$

$+\lfloor n/4 \rfloor$

$+\lfloor n/2^k \rfloor$

$< 2n$

$n + \lfloor n/2 \rfloor + \lfloor n/4 \rfloor \cdots \lfloor n/2^k \rfloor < n + n/2 + n/4 \cdots + n/2^k$

$< 2n$

$O(n+k)$

$O(n+k)$

- **Accounting Method**

Charge 2 for each increment
1 for the actual set

$$0 \quad 1^{①} \; 1^{①} \; 1^{①} \; 1^{①}$$
$$1^{①} \quad 0 \; 0 \; 0 \; 0$$

By taking 2 basically bit is always ahead thus linear

$O(n+k)$

- **Potential Method**

Potential $\phi(D_i)$

D_i DS at time t
D_{i+1} DS at time $t+1$

$\phi(D_i)$
$\phi(D_{i+1})$

acutual cost + Potential diff

$$\hat{C}_i = C_i + \phi(D_{i+1}) - \phi(D_i)$$

$$\Sigma \hat{C}_i = \Sigma C_i + \phi(D_n) - \phi(D_0)$$

ϕ_i # 1's in the counter

$\phi(10110) = 3$

i^{th} op

t_i the # bits reset

actual cost $t_i + 1$

$$\phi \text{ diff} = -t_i + 1$$

$$\frac{}{2}$$

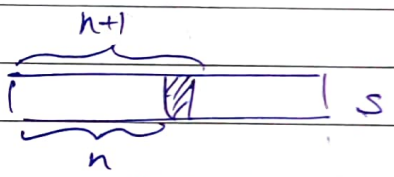
thus $= 0$

$\hat{c}_i = 2$

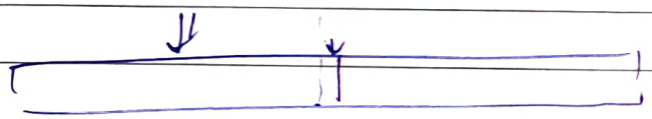
$\sum \hat{c}_i = 2n$

Dynamic table

Insert



When the space becomes full double



$\phi = 2N - S$

where n actual elements

S the space

when $n = S$ $\phi = 2S - S = S = n$

when $n = S/2$ $\phi = 0$

i^{th} insert does not triggers an expansion

$C_i^o = 1$

ϕ_i^o ~~potten diff~~ = ~~2~~ $2N_i^o - S_i^o$

ϕ_{i-1} = $2N_{i-1} - S_i^o$

$2(N_i^o - N_i^o - 1) = 2$

C_i + pot diff = $1 + 2 = 3$

$\hat{C}_i = 3$

Triggers

$C_i^o = N_i^o$

potten diff $-\phi_i^o$ = $2N_i^o - S_{i-1}^o$

ϕ_{i-1} = $2N_{i-1} - S_{i-1}$

$2(N_i^o - N_{i-1}) - S_{i-1}$

$-N_{i-1}^o$

$2 - N_{i-1}$

So C_i^o + pot diff = $2 - N_{i-1} + N_i^o$

$= 3$

$\hat{C}_i^o = 3$

Thus $\sum \hat{C}_i^o = 3n$

$O(n)$

delete (T, x)

load factor = N/S

continue contracting until the lf becomes 1/4

$[1/4, 1/2)$

$$\phi = \begin{cases} \frac{S}{2} - N & \text{if } \alpha < 1/2 \\ 2N - S & \text{if } \alpha \geq 1/2 \end{cases}$$

Fibonacci heap

node : p : parent
child : to any one child
lsibling :
rsibling :
key :
degree : no of children
mark :

mark[x] = true iff x has lost a child since the last time x was made the child of another node.

~~Fib~~. These can have many trees of same order

here basically we delay some operation

$t(H)$ # trees in the ~~heap~~ H

$m(H)$ # marked nodes in H

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

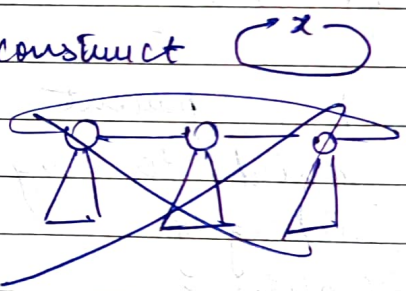
$D(n)$ maximum degree of any node in an n node FH.

$$D(n) = \log(n)$$

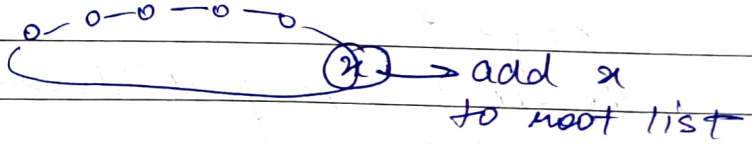
	Actual	Amortised
• Make heap	$O(1)$	$O(1)$

• Insert (H, x)

if H is empty construct

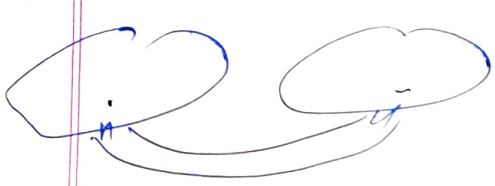


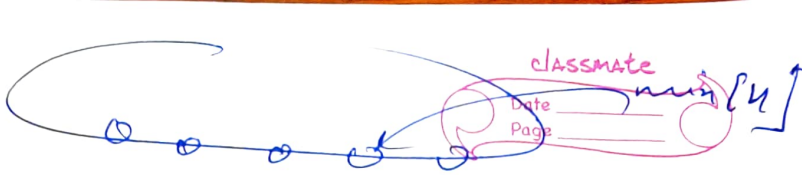
otherwise



Actual $O(1)$

Amortised $O(1)$





min[H]
points the root with the least key

DL list: splicing the list O(1)
delete a node O(1)

make Heap()

creates H
n[H] = 0
min[H] = nil

AC = O(1)
Pdiff = 0
am cost = O(1)

Insert (H, x)

degree[x] = 0
p[x] = nil, child[x] = nil
left[x] = right[x] = x
fuse H's root list with (H)

AC = O(1)
Pdiff = 1
am cost = O(1)

if (min[H] = nil) or (key[x] < key[min[H]])
min[H] = x

n[H]++;

no of nodes in H

FindMin(H)

O(1) return key[min[H]]

Union (H_1, H_2)

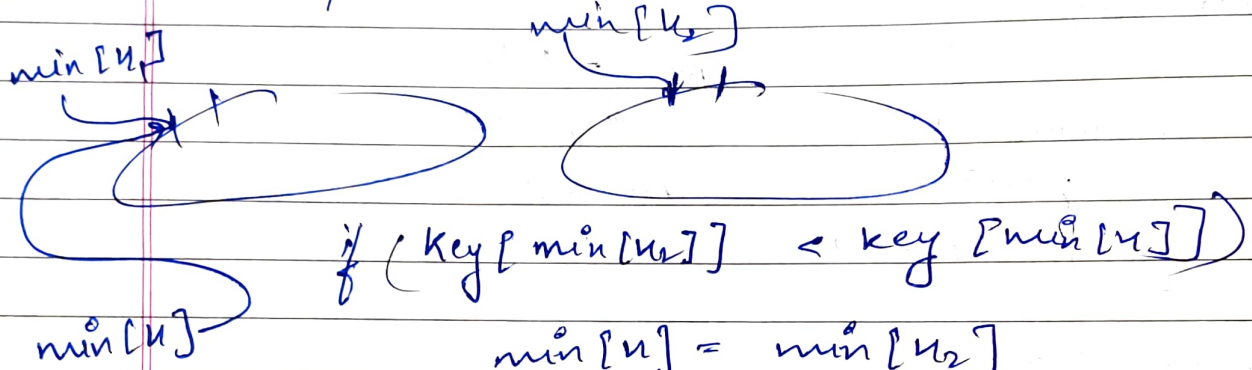
H : ~~Create H~~ makeHeap ( )

~~root list of H = root list (H_1)~~

$min[H] = min[H_1]$

Join H_1;

Splice the root list of H_2 into H



if (key [min [H_2]] < key [min [H]])

$min[H] = min[H_2]$

Join H_2

return H ;

$\alpha_i = \alpha(1)$

$\phi diff = 0$

$\alpha c = O(1)$

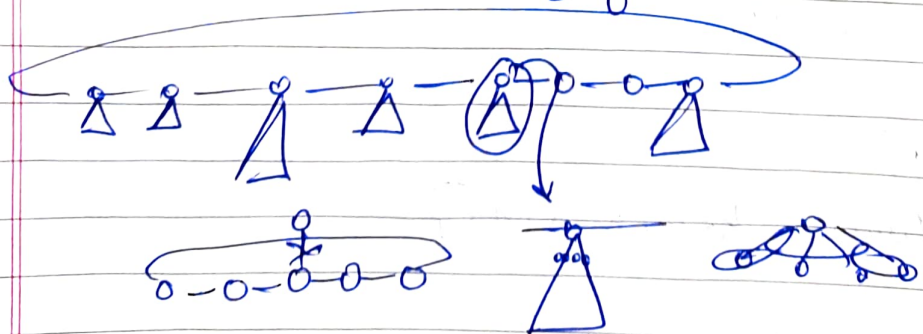
Extract Min (H)

z = min [H]

if (z ≠ nil)

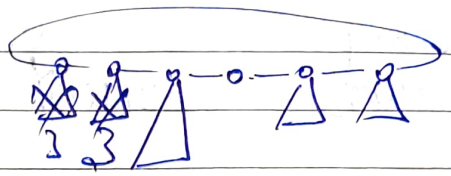
Splice the child list of z into the root list of H.

Remove z from the root list of H;



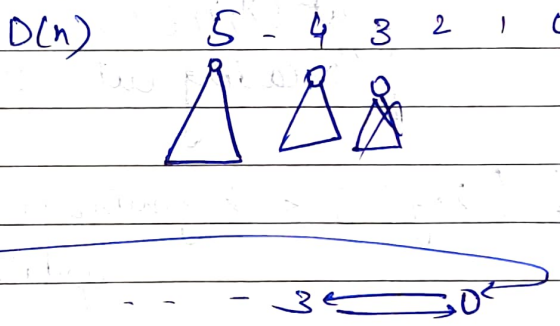
```
if (z = right[z])  then  min[H] = child[z]
else                     min[H] = right[z]
consolidate (H).
n[H]--;
```

Consolidate (H)



D(n) the largest degree possible in FibHeap of n nodes.

D(n)+1 pegs



ac $O(D(n) + t(H))$ time

~~Potential~~ initial $t(H) + 2m(H)$

final $D(H)+1 + 2m(H)$

$D(H)+1 - t(H)$ → we can scale this as much as we want thus it will cancel $t(H)$ of ac

amortised $O(D(n))$

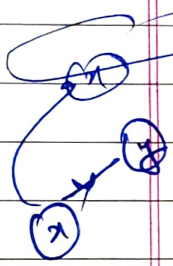
$O(\log n)$

$O(a+b+c+e+(d+f)\log n)$

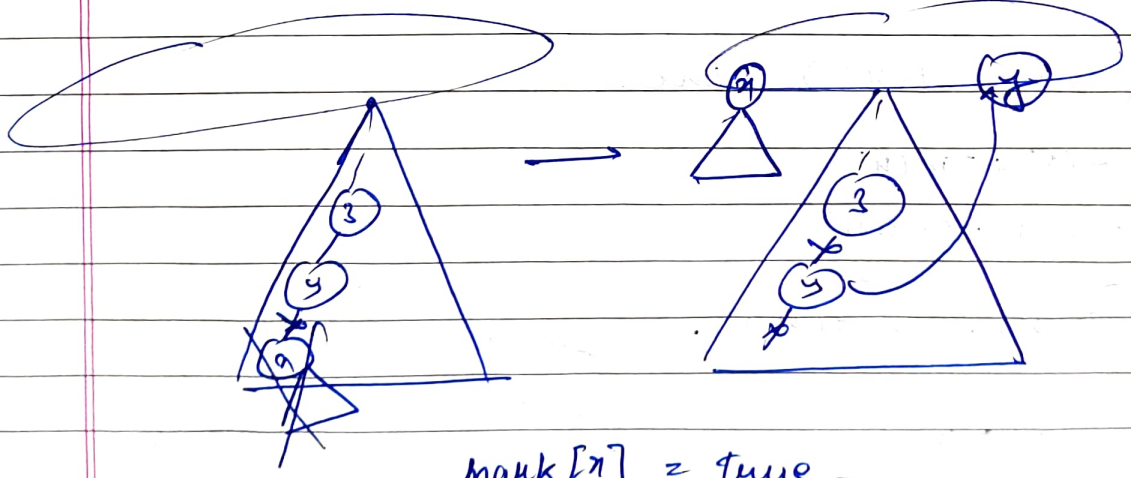
$(t-1) + D(n)$
old new
$O(t-1+D(n))$

	am. cost	ϕ diff	actual	
Make heap	$O(1)$	0	$O(1)$	
Insert	$O(1)$	1	$O(1)$	a
Find min	$O(1)$	0	$O(1)$	b
Union	$O(1)$	0	$O(1)$	c
Extract Min	$O(D(n))$ ~~$O(\log n)$~~	$D(n)+1-t$	$O(t+D(n))$	d
↓ key	$O(1)$	$4-c$	$O(c)$	e
Delete	$O(D(n))$ ~~$O(\log n)$~~			f

Decrease key (H, x, k) /* $k < key[x]$ */




```
key[x] = k
if ( y ≠ nil && key[x] < key[y] )
    [ Cut (H, x, y);
      Cascading Cut (H, y);
if ( key[x] < key[min[H]] )
        update min[H]
```



mark[x] = true

if x has lost a child since last becoming the child of another node

if y was marked earlier then move y also to root list node and perform same on y.

```
Cut (H, x, y)
    Remove x from y's childList
    degree[y]--;
    Add x to the root list;
    p[x] = nil;
    mark[x] = false;
```

```
Cascading Cut (H, y)
    z = p[y]
    if (z ≠ nil)
        if (mark[y] = false)  mark[y] = true
        else { Cut (H, y, z)
               Cascading Cut (H, z)
```

c = #cascading cut

		actual	diff
initial	$t + 2m$		$c-1+1=c$.
final	~~$t+(c-1)+1=c$~~	$c-1$ unmark	
	$c+t+2m-2(c-2)$	1 mark	
	~~$= c+2c-4$~~		
	$4-c$		

Delete(H, x)

[DecreaseKey(H, x, -∞)
ExtractMin(H)

Lemma :

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i$$

0, 1, 1, 2, 3, 5

$$F_i = F_{i-1} + F_{i-2}$$ for

Basis $F_2 = 1 + F_0 = 1$

IH $F_{k+2} = F_k + F_{k+1} = F_k + 1 + \sum_{i=0}^{k-1} F_i$

$$= 1 + \sum_{i=0}^{k} F_i$$

$$F_{k+2} \geq \phi^k \qquad \phi = \frac{\sqrt{5}+1}{2}$$

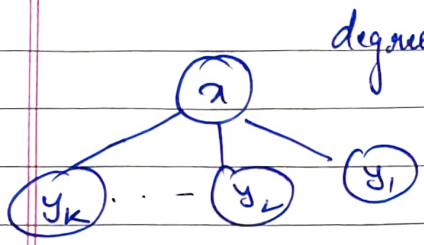
$k=0$ $\quad 1 = F_2 \geq \phi^0 = 1$

$k=1$ $\quad 2 = F_3 \geq \phi^1 = 1.619$

$$F_{k+2} = F_k + F_{k+1} \geq \phi^{k-2} + \phi^{k-1} = \phi^{k-2}(1+\phi)$$

$$F_{k+1} \geq \phi^{k-2}(1+\phi) = \phi^{k-2}(\phi^2) = \phi^k$$

- Size(x) is the no. of nodes in the subtree rooted at x.



y_i degree $\geq i-2$ $i \geq 2$

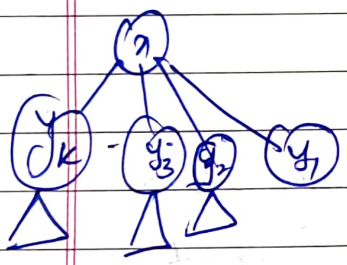
y_1 degree ≥ 0

at the time of linking y_1 degree $\geq i-1$

now y_i degree $\geq i-2$

Let S_k denote the min size a node of degree k can have in any Fibheap.

let x be a node of degree k such that size of $x = S_k$



$$S_k \geq 1 + 1 + \sum_{i=2}^{k} S_{i-2}$$

claim: $S_k \geq F_{k+2}$

Basis $S_0 = 1 = F_2$

$S_1 = 2 = F_3$

IS

$$S_k \geq 2 + \sum_{i=2}^{k} S_{i-2}$$

$$\geq 2 + \sum_{i=2}^{k} F_i$$

$$\geq 1 + \sum_{i=0}^{k} F_i = F_{k+2}$$

thus $\boxed{S_k \geq F_{k+2}}$.

any node x degree k

$$Size(x) \geq S_k \geq F_{k+2} \geq \phi^k$$

Take a heap of n nodes

$$n \geq Size(x) \geq \phi^k$$

$$\log_\phi n \geq k \log_\phi$$

thus $k \leq \log_\phi n = \frac{\log n}{\log \phi}$

thus $k = O(\log n)$

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

	actual	ϕ diff	amortized	Binary heap
make heap	$O(1)$	0	$O(1)$	$O(n)$
Insert	$O(1)$	1	$O(1)$	$O(\log n)$
find min	$O(1)$	0	$O(1)$	$O(1)$
Union	$O(1)$	0	$O(1)$	$O(n)$
Extract Min	$O(t+D(n))$	$D(n)+1-t$	$O(D(n))$	$O(\log n)$
Decrease key	$O(c)$	$4-c$	$O(1)$	$O(\log n)$
Delete			$O(D(n))$	$O(\log n)$

Single Source shortest path

Dijkstras algorithm

- directed graph
- Source vertex s
- edge weights $E \to R^+ \cup \{0\}$