DATE 28-1-2019 Success
PAGE

T-2

Lecture-20

Graph.

Tree - Free

- Ordered
- Positional.

Directed ... V

E = Set of relations

Undirected

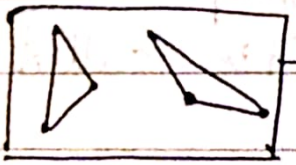
→ no. self loops.

Path, cycle, reachability,

Degree, Connected components

Simple path

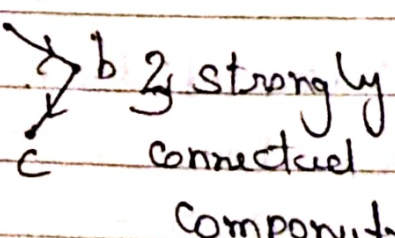
- length of path → # edges.



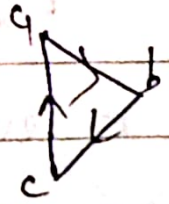
→ No. of connected component = 2.

Directed

- strongly connected → a ⇄ b is 2.



2 strongly connected components



1 strongly connected component.

Undirected Graph:

If there is a path from vertex v to u.

⇒ There is a simple path from v to u

Suppose:

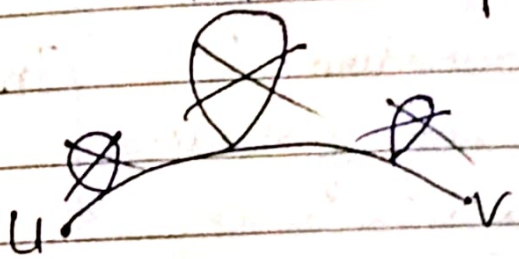
$v = v_1 \cdots v_i \cdots v_j \cdots v_k = u$

$\Rightarrow v = v_1 \cdots v_{i-1}\, v_i\, v_{j+1} \cdots v_k$ is also a path

Similarly,

repeat

Till no repetations ⇒ Simple path.



* Subgraph:

multigraph } → multiple edges between same pair of vertices.

hypergraph } instead pairs of nodes → edges

↓

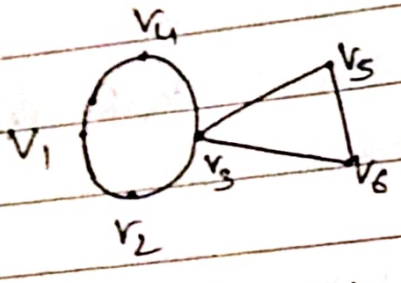
a longer tuple of nodes.

Cycle:

path $v = (v_1 \dots\dots v_k) = u$

Same vertex.

simple cycle:
↳ all vertices except v_1 & v_k are distinct.

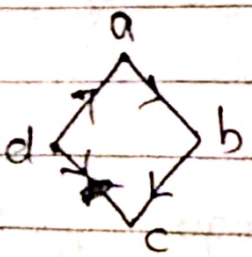


$(v_1\ v_2\ v_3\ v_4)$
$(v_2\ v_3\ v_4\ v_1)$
$(v_3\ v_4\ v_1\ v_2)$

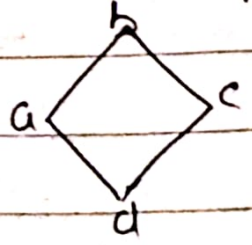
Any cycle obtained as a result of reordering/rotating the same set of vertices is the same cycle.

} In undirected graph.

Directed graph:
↓
Acyclic as well then it is called Directed Acyclic Graph (DAG).



DAG



Similar undirected graph is not acyclic.

Undirected Graph:-

G → Undirected Graph

G is a "free tree" if, G is connected & Acyclic.

The following statements are equivalent.

1. G is a free tree.
2. ⇒ If G is a free tree then there is a unique simple path between every pair of vertices.

Proof

Take any (u, v)

free tree ⇒ connected

⇒ there is path u & v.

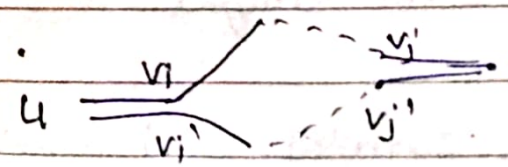
Prove that the path is unique.

Suppose there are two paths.

u : $v_1\ v_2 \dots v_k, v$

v $v_1'\ v_2' \dots v_k', v$

Assume v_i is the point at which they diverge & v_j is the point at which they converge.



path ($v_i\ v_{i+} \dots v_j, v_{j-1} \dots v_i'$) → cycle.

cycle ⇒ G is not a tree true

↓

contradiction.

↓

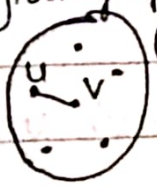
So, no 2 paths possible

↓

⇒ path is unique.

G is an undirected graph, such that there is a unique simple path between every two pair of vertices, then if any edge is removed the graph becomes disconnected

given graph



If ~~e~~ (u,v) is removed.

Graph is disconnect in G (u,v).

path between u & v.

Suppose it does not get disconnected

⇓

There was another path between (u,v)

⇓

path is not unique.

③ G is connected but

If an edge is G is removed G becomes disconnected

(4) G is an undirected graph such that it is connected and if any edge in G is removed it becomes disconnected then

$$|E| = |V| - 1$$

If G is connected then

$$|E| \geq |V| - 1$$

2 vertices.

extend add 1 or more edge per vertex.

$$|E| \geq |V| - 1.$$

Next

prove

$$|E| \leq |V| - 1.$$

Edges	Connected components	Influence (No of edges)
$\|E\|$	1	$\|E\| = m$
$\|E\| - 1$	2	$\|E\| = m-1$
$\|E\| - 2$	3	$\|E\| = m-2$
$\|E\| - (\|V\| - 1)$	$\|V\|$	0

Hence,

$$|E| - (|V| - 1) = 0 \Rightarrow |E| = |V| - 1.$$

DATE: 29-01-2019
PAGE:

1.

(4) ~~G is connected &~~

Lecture-21

Free tree

- Theorem

Rooted tree & related terms

Positional & Ordered tree

-Binary tree - randomly built.

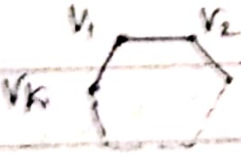
Theorem:

G is an undirected graph. Then the following statements about it are equivalent

1. G is a free tree.
2. Every vertex pair in G has a unique simple path connecting the 2 vertices
3. There is a unique simple path between any vertex pair but removing an edge disconnects the graph
4. G is connected & $|E| = |V| - 1$.
5. G is acyclic & $|E| = |V| - 1$.

Proof $4 \Rightarrow 5$

Assume G has a cycle of k vertices.



$\longrightarrow$ V_1 to V_k = k vertices, k edges.

Remaining $|V| - k$.

To add V_{K+1} to this graph component atleast 1 edge is needed.

vertices	edges (at least)
K+1	K+1
K+2	K+2
⋮	⋮
K+|V|-K	K+|V|-K).
|V|	|V| → Contradiction
	→ assumption wrong
	⇒ Acyclic.

6. G is acyclic but adding any edge creates a cycle in G.

proof 5 ⇒ 6

Given undirected acyclic G with $|E| = V-1$

prove that adding any edge (u,v) creates a cycle.

Suppose adding an edge, does not created a cycle then it must have been that ... u & v belong to different connected component their must have been more than 1 connected component.

Let k be the no. of connected component in G →

→ each connected component is a free tree, by definition

1 ⇒ 2 ⇒ 3 ⇒ 4 ⇒ 5.

for each of them. $|E_i| = |V_i| - 1$

DATE: 29-01-2019
PAGE:

~~G is connected &~~

Lecture-21

Free tree

- Theorem

Rooted tree & related terms

Positional & Ordered trees

- Binary trees - randomly built.

Theorem:

G is an undirected graph. Then the following statements about it are equivalent

G is a free tree.

Every vertex pair in G has a unique simple path connecting the 2 vertices.

There is a unique simple path between any vertex pair but removing an edge disconnects the graph.

G is connected & $|E| = |V| - 1$.

G is acyclic & $|E| = |V| - 1$.

Proof $4 \Rightarrow 5$

To add V_{K+1} to this graph component atleast 1 edge is needed.

vertices	edges (at least)
K+1	K+1
K+2	K+2
⋮	⋮
K+|V|-K	K+|V|-K).
|V|	|V| → Contradiction
	→ assumption wrong
	⇒ Acyclic.

6. G is acyclic but adding any edge creates a cycle in G

proof 5 ⇒ 6

Given undirected acyclic G with $|E| = V-1$

prove that adding any edge creates a cycle.

Suppose adding an edge (u,v) does not created a cycle then it must have been that ... u & v belong to different connected component their must have been more than 1 connected component.

Let K be the no. of connected component in G →

→ each connected component is a free tree, by definition

1 ⇒ 2 ⇒ 3 ⇒ 4 ⇒ 5.

for each of them. $|E_i| = |V_i| - 1$

Success
DATE: .....
PAGE: .....

sum both sides over all compnts.

$|E| = |V| - K$.

given

$|K| = |V| - 1$

$\Rightarrow K = 1$.

6 ⇒ 1:

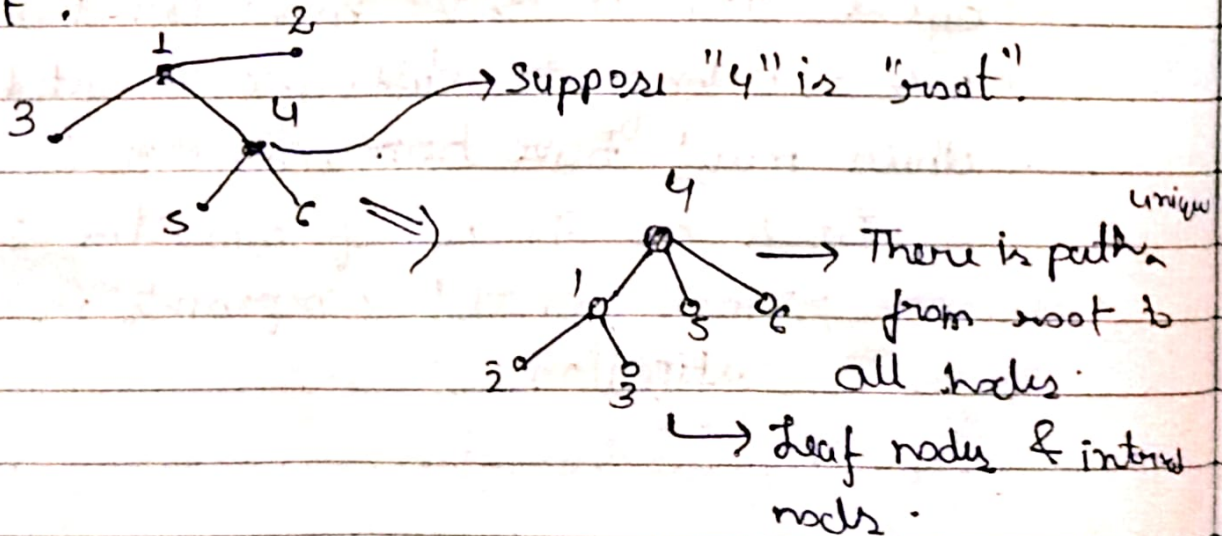
If we add any edge it makes a cycle it mens all the compnts are already connected.

Free Tree:

Degree: No. of adjacent edges do the vertex.

Rooted Tree:

Free tree in which one vertex is 'disignated' as "root".



↳ Leaf nodes: Not on path from root any other node.

Degree in rooted tree:

Child → X is a child of y if y is the immediately preceding node in path from root to X.

Parent → Y is a parent of X if X is is a child of Y.

Descendant: X is descendant of y if y belongs to the path from root to X.

Ancestor: Y is ancestor of X, if X is descendant of y.

Siblings: X & Y have same parents.

Degree: No. of child nodes.

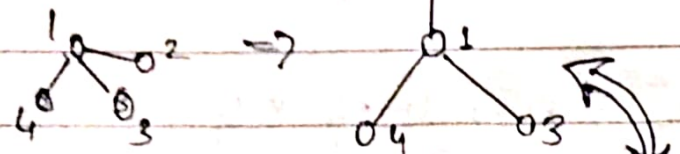
Height of a vertex/Node: X is the length of longest path to a descendant leaf.

Height of the root:
- Longest path to a leaf

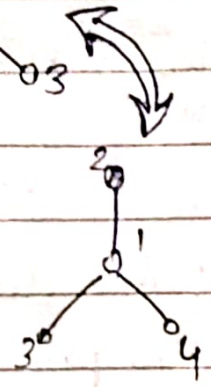
Height of the Tree:
Height of the root.

Depth of a node: Distance from the root.

Ex: Free tree:

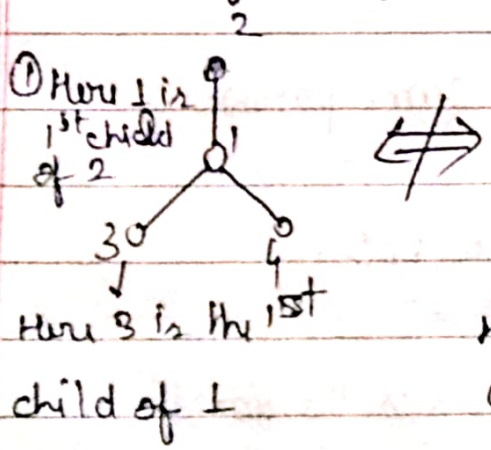


→



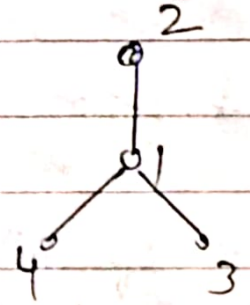
* Ordered tree:

Sequencing or "ordering" on the child nodes of every node.



① Here 1 is 1st child of 2

Here 3 is the 1st child of 1



Here 4 is the 1st child of 1.

* Positional tree:

A "position" for every child node.

DATE: 30-01-2019

Lecture: 22

Ordered & Positional trees

Binary trees, BSTs

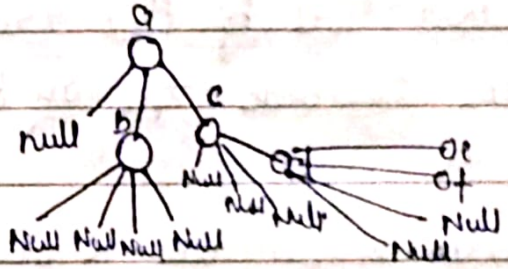
Balance

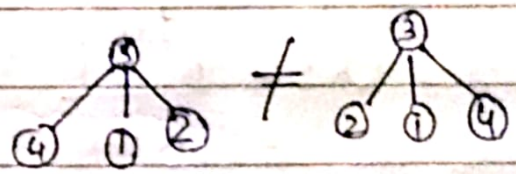
Randomly built binary search trees.

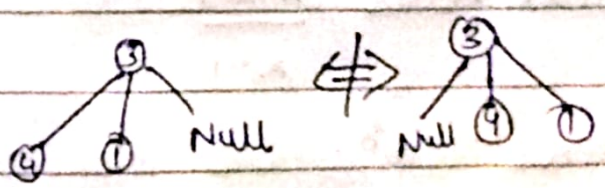
Positional tree:

- Null tree
- "ariness": Number of child nodes for each non null node.

→ recursive of a k-ary positional tree.



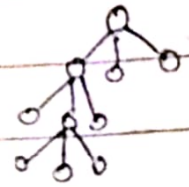




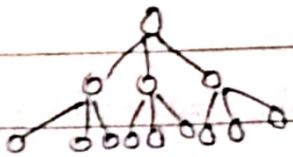
→ In a positional tree

Full Tree: Every node has degree k or 0

↑ Leaf Node



Complete k-ary tree: If a full tree in which all the leaf nodes are at the last level.



Nearly Complete: The last level of the tree could have no nodes towards the right part.

Height – Number of nodes in complete k-ary tree.

$$n = 1 + k + k^2 + \cdots + k^h = \frac{k^{h+1} - 1}{k - 1}$$

#nodes

$$\#\text{internal nodes} = \frac{k^h - 1}{k - 1}$$

$$\#\text{leaf nodes} = k^h.$$

Binary: 2-ary positional tree.

left } names for 2 child slots.
right

$n = 2^{h+1} - 1$

leaf nodes $= 2^h$

internal $= 2^h - 1$.

Dynamic Set: Binary Search Trees.

Key: Unique → BST: for every node x

$x.key > x.left.key$

$x.key < x.right.key$.

Worst case height $= n-1$

Search $O(height)$

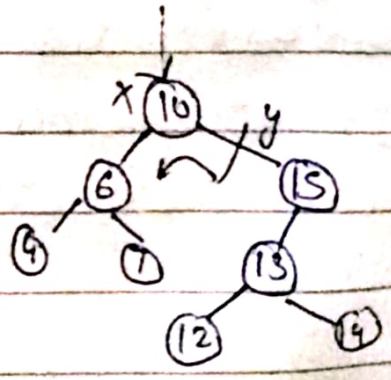
$\rightarrow O(n)$.

Best case height is $O(\log_2 n)$.

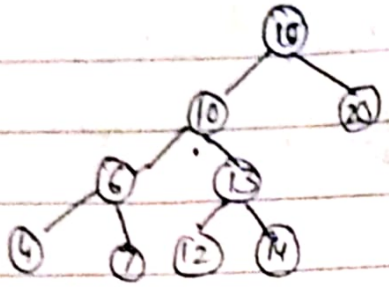
⇕

Balanced binary search tree.

Rotation:



left rotate on x



Left rotate (x)

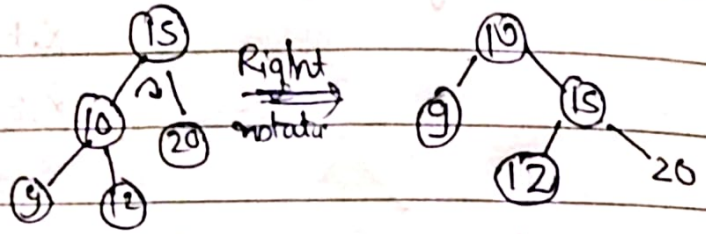
```
y = x.right
x.right = y.left
y.left = x
return y
```

} O(1)

Right rotate (x)

```
y = x.left
x.left = y.right
y.right = x
return y
```

} O(1)



left rotate (x) # returns a pointer to a rotated subtree of nodes originally contained in sub tree of x.

Binary Search Tree: → Expected height is O(log n).

- Inserts
- No deletes

| Input sequence - random

Input Sequence:

random variable:

$R_j \rightarrow$ root key of the bst made with j nodes (keys).

$R_n = i, \forall i = 1$ to n with equal probability

$$Z_{n,i} = \begin{cases} 1, & i \text{ is the root of tree made with } n \text{ nodes} \\ 0, & \text{otherwise} \end{cases}$$

X_n be the height of the tree made with n nodes

Y_n exponential height $= 2^{X_n}$.

$$X_n = 1 + \max(X_{i-1}, X_{n-i}).$$

31-01-19 Lecture-23

Randomly built bst → Insert → height?

Binary trees — Search

- AVL trees.

$\{1 \ldots n\} \rightarrow n!$ permutations are equally likely.

R_i = value of root in a randomly built bst of i nodes.

X_i = height of a randomly built bst of i nodes.

$Y_i = 2^{X_i}$

$Z_{j,i} = 1$ i the root node of a r.b. bst of j nodes

0 otherwise.

$R_n = i$ ⟶ first key

R.B. bst of 1 to $i-1$ ($i-1$ nodes)

R.B. bst of $n, n-1, \ldots, i+1$ ($n-i$ nodes)

$x_n = 1 + \max(x_{i-1}, x_{n-i})$

$y_n = 2^{1+\max(x_{i-1}, x_{n-i})}$

$= 2 \cdot 2^{\max(x_{i-1}, x_{n-i})}$

$= 2 \cdot \max(2^{x_{i-1}}, 2^{y_{n-i}})$

$= 2 \cdot \max(y_{i-1}, y_{n-i})$.

$y_n = 2 \cdot \max(y_{i-1}, y_{n-i})$

$E(y_n) = 2 \cdot E(\max(y_{i-1}, y_{n-i}))$.

$= \sum_{i=1}^{n} P_r(R_n = i) \, 2 \max(y_{i-1}, y_{n-i})$.

$$= \frac{2}{n} \sum_{i=1}^{n} E(\max(Y_{i-1}, Y_{n-i}))$$

$$\leq \frac{2}{n} \sum_{i=1}^{n} E(Y_{i-1}, Y_{n-i})$$

$$\leq \frac{2}{n} E\left(Y_0 + Y_{n-1} + Y_1 + Y_{n-2} + \cdots + Y_{n/2} + Y_{n/2+1} + \cdots + Y_{n-1} + Y_0 \right)$$

$$\leq \frac{4}{n} \sum_{i=0}^{n-1} E(Y_i)$$

$$E(Y_n) \leq \frac{4}{n} \sum_{i=0}^{n-1} E(Y_i)$$

Find $E(Y_i)$ for each the $i+j$.

Substitution method :

(guess) $E(Y_n) \leq \frac{1}{4}\, {}^{n+3}C_3$.

base case $n = 1$.

$E(Y_1) = 1$. $\rightarrow \leq 1/4\, {}^{1+3}C_3 \leq 1/4 \cdot 4$.

Assume it is true for $r = 1, 2, 3 \cdots n-1$

Show true for $r = n$.

$${}^{n+3}C_4 = \sum_{i=0}^{n-1} {}^{i+3}C_3$$

RHS $= {}^{3}C_3 + {}^{4}C_3 + {}^{5}C_3 + \cdots + {}^{n+2}C_3$

$= \boxed{{}^{4}C_4 + {}^{4}C_3} + {}^{5}C_3 + \cdots + {}^{n+2}C_3$

$= {}^{5}C_4 + {}^{5}C_3 + \cdots + {}^{n+2}C_3$

$\vdots$

$= {}^{n+2}C_4 + {}^{n+2}C_3$

$= {}^{n+3}C_4.$

$$E(Y_n) \leq \frac{4}{n} \sum_{i=0}^{n-1} E(Y_i)$$

$$\leq \frac{1}{n} \sum_{i=0}^{n-1} {}^{i+3}C_3$$

$$\leq \frac{1}{n} {}^{n+3}C_4$$

$$\leq \frac{1}{4} \cdot {}^{n+3}C_3$$

$$E(Y_n) \leq {}^{n+3}C_3.$$

$E(Y_n) = E(2^{X_n})$.

$E(2^{X_n}) \leq c^{n+3} C_3$.

$E(X_n) \leq \log(E(2^{X_n}))$

$\leq \log(c \cdot {}^{n+3}C_3)$

$\leq \log(c_1 n^3 + c_2 n^2 + c_3 n^1 + c_4)$

$E(X_n)$ is $O(\log n)$.

↓

height.

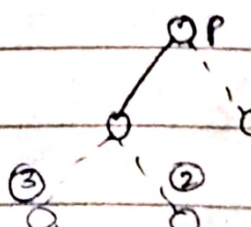
Worst case : $n-1$

---

* BST can be improved : to have height $O(\log n)$ ie. "balanced" BSTs always.

* AVL : Adelson, Velski, Landis.

↳ for every node, the heights of left & right subtrees should differ by atmost 1.



→ In case ③ we will apply right rotate at p.

→ Readjustment to satisfy AVL property.

Self adjusting data structure

height : h.

N(h) : number of nodes in AVL tree of height h.

$N(h) \geq 1 + N(h-1) + N(h-2)$

$N(0) = 1$

$N(1) \geq 2$

$N(2) \geq 4$

$N(3) \geq 7$

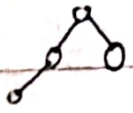
$$N(h) \approx c_1 \left(\frac{\sqrt{5}+1}{2}\right)^h - c_2 \left(\frac{\sqrt{5}-1}{2}\right)^h.$$

$$n \geq c_1 a_1^h + c_2 a_2^h b$$

Hence $h \leq c \log_2 n$.

1-02-19 Lecture-24

AVL Trees : Heights of child node differ by atmost 1.

	Min. nodes	
ht = 0	1	o
ht = 1	2	
ht = 2	4	

N(h): min. nodes in ht AVT tree.

$N(h) = 1 + N(h-1) + N(h-2)$

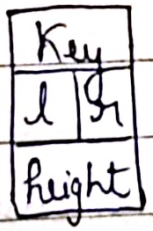
$N(h) \geq c_1^h + c_2^h$

$h < \log(N(h))$

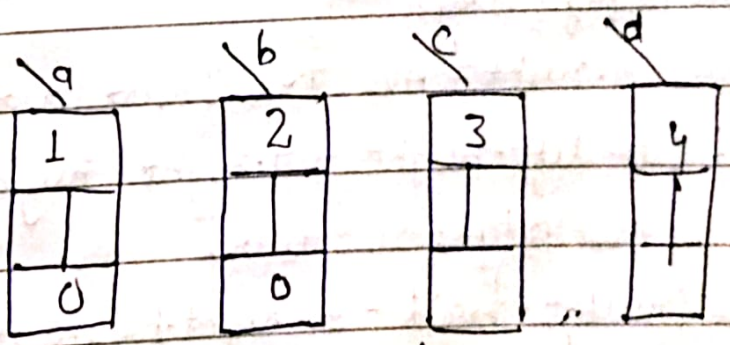
$h < \log n$.

* Balance factor: |Height of x.left - height of x.Right| ≤ 1.
(Self adjusting tree).

Node Structure:



Example:




```
rotate_left(x)
    y = x.right
    if (y != nil)
        x.right = y.left
        y.left = x
    return(y)
    else return(x)
```

```
rotate_right(x)
    y = x.left
    if (y != nil)
        x.left = y.right
        y.right = x
    return(y)
    else return(x)
```

Call from : AVL_Tree_Insert (T.root, x)
Application # x is address of node to be inserted.

AVL_Tree_Insert (r, x)
return an AVL Tree root formed with node in the subtree rooted at r (which is AVL) on x.

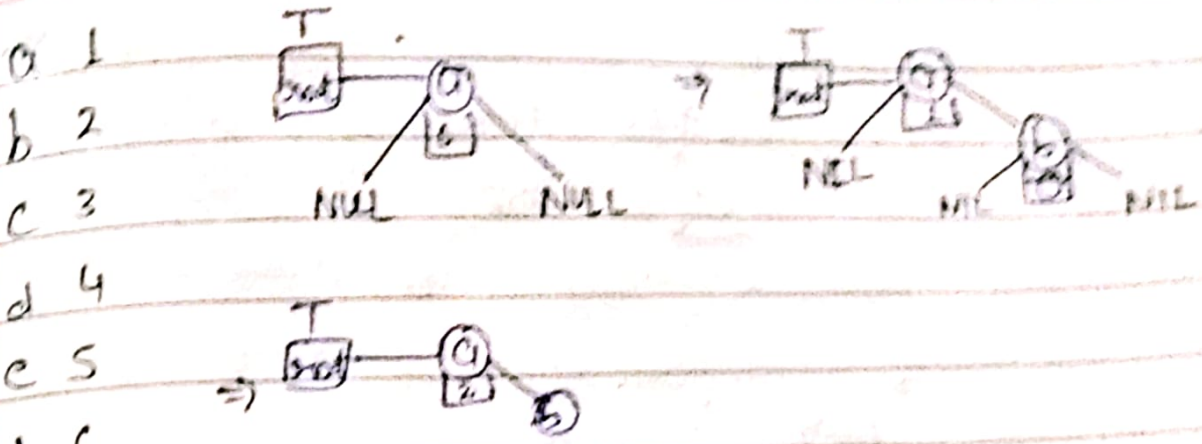
* Distinct keys
* keys are positive integers

```
AVL_Tree_Insert (r, x)
    if (r == NIL) return (x)
    else
    if (x.key < r.key)
        r.left = AVL_Tree_Insert (r.left, x)
    else if (x.key > r.key)
        r.right = AVL_Tree Insert (r.right, x).
    if (r.left.height - r.right.height) > 1
        r = right rotate (r)
    if (r.left.height - r.right.height) < -1
        r = right rotate (r)
    r.height = 1 + max (r.right.high, r.right.height)
    return r
```

$T(h) \leq T(h-1) + c$

$T(h) = O(h)$

$= O(\log n)$

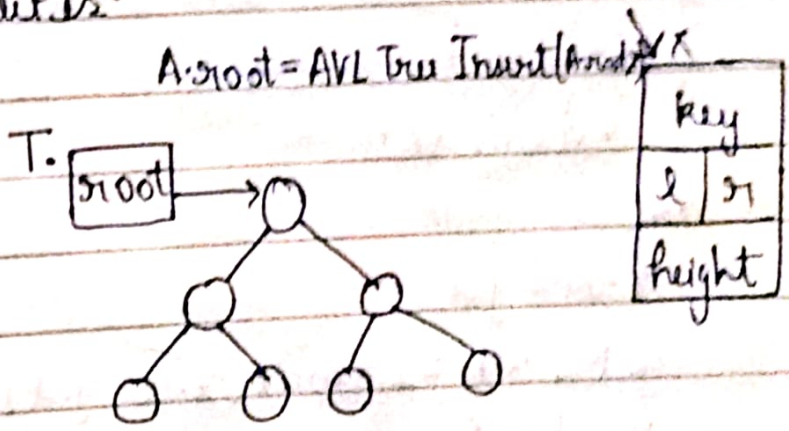


4-2-18 Lecture-25

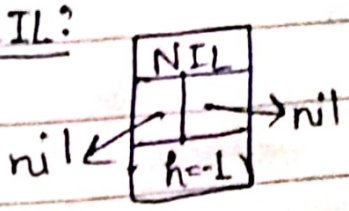
AVL - Trees:

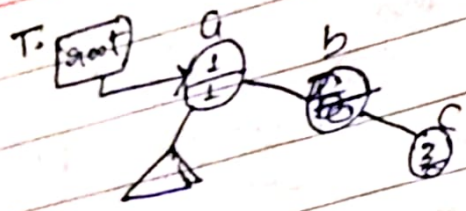
1. Inserts
2. Deletes

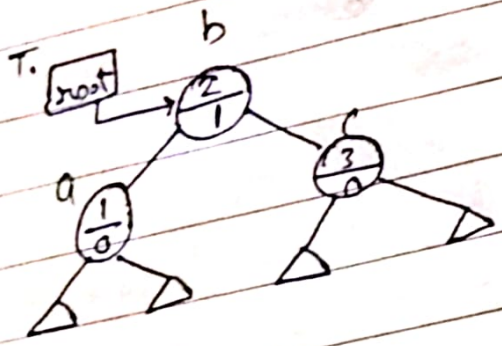
A.root = AVL Tree Insert(root, k)



NIL:








```
left.rotate(x)
    y = x.r
    if (y == nil)
        return (x)
    else
        x.r = y.l
        y.l = x.h = 1 + max(x.l, y.l)
        y.l = x
        y.h = 1 + max(x, y.r)
    return (y).

right.rotate'(x)
    z = x.l
    if (z == nil) return(x)
```

else

x.l = z.r

x.h = 1 + max (x.r, z.r)

z.r = x

z.h = 1 + max (x.h, z.l).

return (z).

AVL_Tree_Delete (r, x)

delete x from subtree rooted at r.
and return pointer to modified tree

if (r == NIL) return (r).

else

if (x != r)

if (x.key < r.key)

r.left = AVL Tree Delete (r.left, x)

if r.left - r.right < -1.

y = left rotate (r).

return y

else if (x.key > r.key)

r.right = AVL Tree Delete (r.right, x)

if r.right - r.left < -1

y = right rotate (r)

return y.

```
else
    if (x.left == NIL) AND (x.right == NIL)
        return NIL.
    else if (x.left == NIL)
        return (x.right)
    else if (x.right == NIL)
        return (x.left).
    y = minimum (x.right).
    z = AVL Tree delete (x.right, y)
    y.right = z.
    y.left = x.left
    y.height = 1 + (y.x.h, y.l.h).
    (restore AVL property at y).
    return y
```

```
minimum(y)
returns finds
minimum # key
node in subtree
rooted at y.
if (y.left == nil)
    return (y)
else return
    (minimum(y.left))
```

Lecture 26

Disjoint sets

- Application
- Operations
- Implementation.

n: number of nodes → n.log(log n) bits

↓

extra space

Use to:

BST → dynamic data

: AVL Trees.

One approach to improve the complexity:

Red-Black Tree:

---

* Disjoint sets: {1}, {2} {3} {4} {5} different set.

Union: operation {1, 2} {3} {4}.

Findset(u): return the representative element.

(for a set there should be only one representative)

Makeset(u): make a set out of a given key (element) & returns the representative i.e. itself.

{a} {b} ⟶ findset(a) ≠ findset(b)

↳ {a} ∪ {b}

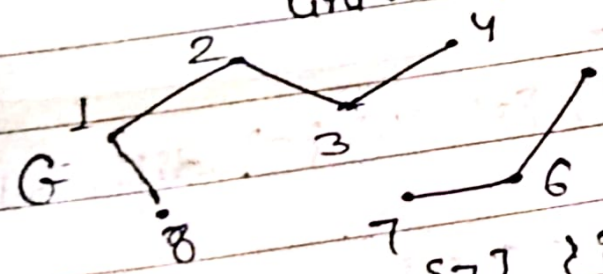
↳ findset(a) = findset(b).

Connected_components:

Built up the sets each connected component is a different set.

building:

```
Connected_components(G)
    for each ~~vert~~ v ∈ G, V
        makeset(v)
    for each edge (u, v) ∈ G, E.
        union(u, v).
```



{1}, {2} .... {7} {8}.

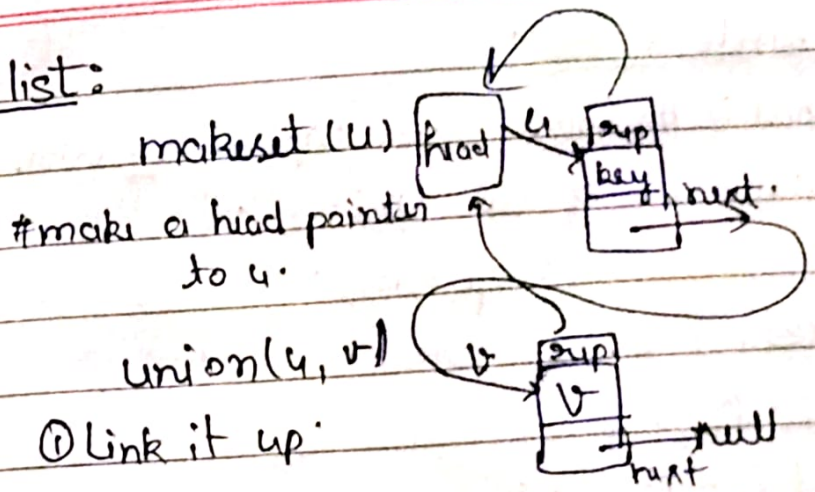
{1, 8}, {2} .... {7}

{1, 2, 8} .... {7}

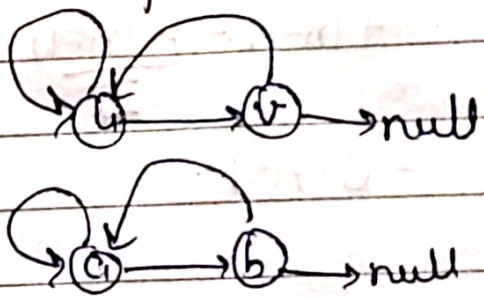
{1, 1, 3, 8, 4} {5, 6, 7}.

```
Isconnected(x, y)
    return | findset(x) = findset(y)}.
```

Linked list:



findset(v)
findset(u) → return (rep-value).



union(u,u) # currently our convention merge 2nd to 1st function.

1. Traverse to last node of u set.
2. Change its next to 2nd set.
3. Update all the pointers in 2nd list to head of fi

makeset: O(1)
findset: O(1)

n elements in all:

What is the worst case union sequence here? i.e. the total cost of all unions.

(n-1): operations.

[2] [2] : 1

[3] [1]–[2] : 2

[4] [3]–[1]–[2] : 3

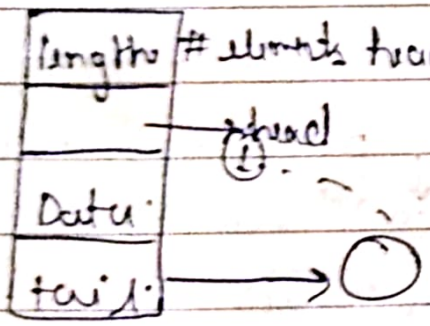
. . . . . .

[n] [n-1]–[n-2]–···–[1]–[2] : n-1

Total cost = $1+2+3+\cdots+(n-1) = \frac{n(n-1)}{2}$

$= O(n^2)$.

Head.



makeset(u):

O(1) { allocate a header & put u as first & tail elements, next=null
put length = 1.
return (pointer to u).

rep pointer to u header.

findset(u)

return(u.rep)

union(u, v) · weighted union heuristic

```
x = findset(u)
y = findset(v)
if (x == y)                    {did not d.
        return(u.rep)
```

7-2-19 Lecture: 27

Disjoint sets

Operations

Linked list implementation: Weighted union heuristic

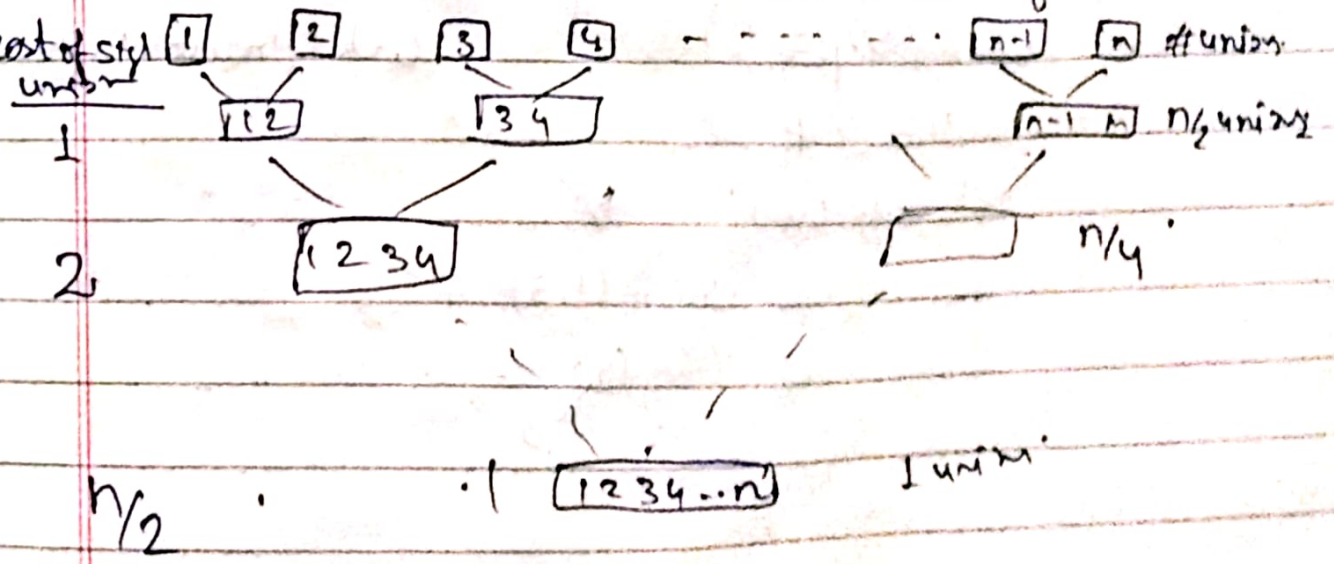
- Analysis.
- other methods.

Weighted union heuristic:

What is its worst case complexity.

Assume n makesets.

& then unions results in 1 set of n elmds.



Total no. of operations:

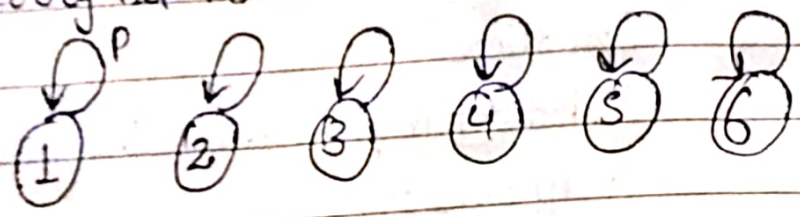
$$= \frac{n}{2} \times 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 4 + \cdots + 1 \cdot \left(\frac{n}{2}\right)$$

$$= \log_2 n \times \frac{n}{2} = O(n \log n)$$

The average cost $O(\log n)$.

Trees: DISJOINT FOREST

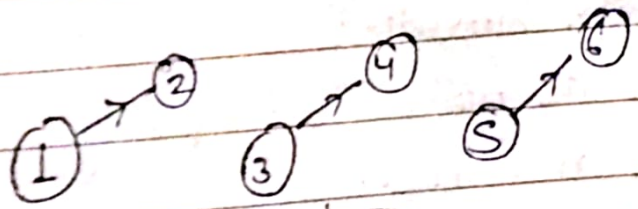
↳ Every set is a tree.



Findset(u)

union(u,v).
Link(u,v)

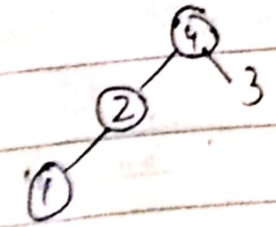
Link(u,v)
u.p = v.



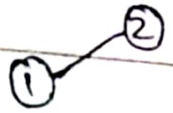
Findset(u)
while (u.p ≠ u)
u = u.p
return (u).

union(u,v)
Link(findset(u), findset(v))

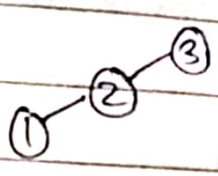
Union(1,3)



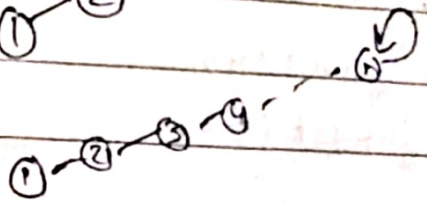
1 union (1, 2)



2 union (1, 3)



n-1 union (1, n)



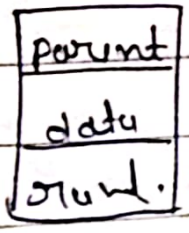
$$1+2+3+\cdots+(n-1)$$

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

$$= O(n^2).$$

* Disjoint set with Disjoint forest implementation +
* Ranked Union Heuristic:

A rank is an upper bound on a node's height




```
makeset(u)
    u.p = u
    u.rank = 0

findset(u)
    while (u.p != u)
        u = u.p
    return (u)

Link (u, v)
    if u.rank < v.rank
        u.p = v
    else v.p = u
    if (v.rank == u.rank) u.rank = u.rank + 1
```

Union(u, v)

Union with ranked union heuristic.

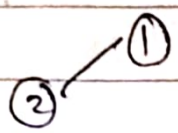
Union (u, v)

p = find(u)

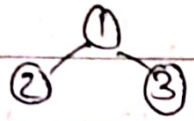
q = find(v)

link(p, q)

1. Union (1, 2)

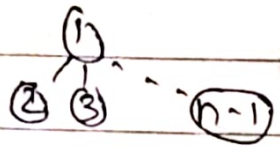


2. Union (1, 3)

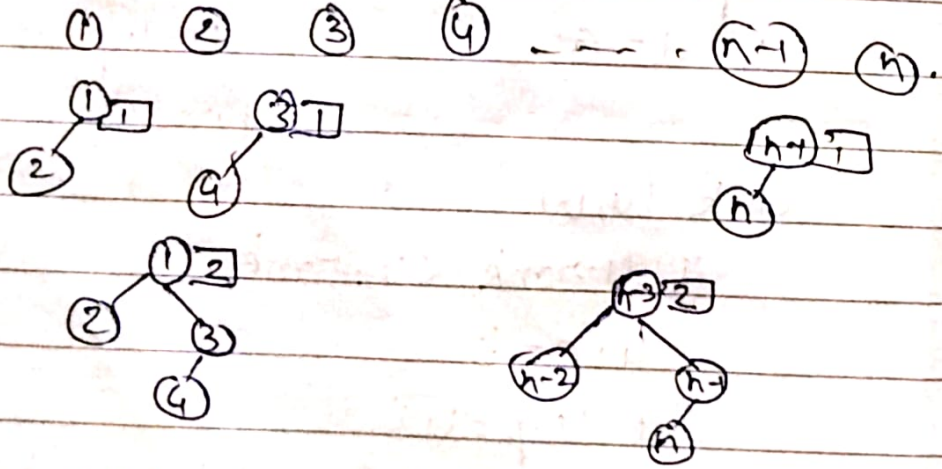


3. Union (1, 4)

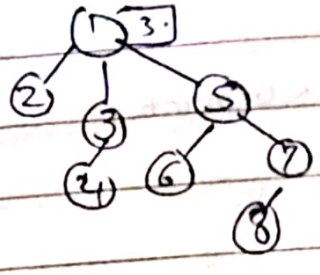
n-1 Union (1, n)



Worst case:



union(4,8)



Sum-up union costs:

# unions	costs of single union
$n/2$	$2 \wedge 1$
$n/4$	2×2
$n/8$	2×3
$n/16$	$n \times 4$.

Σ costs $= O(n \log n)$.

* Path compression Heuristic:

9-2-19 Lecture: 28

- Path compression & Ranked union.
- Analysis.

makeset(x)

x.rank = 0

x.p = x.

```
Link(x, y)
  if x.rank < y.rank
     x.p = y
  else
     y.p = x
     if (x.rank == y.rank)
        x.rank = x.rank + 1.

Union (a, b)
  Link (findset (a), findset (b))

Findset (x)
  if (x.p == x)
    return (x).
  else
    x.p = findset (x.p).
    return (x.p)
```

* Analysis: (Potential method):

First study about very fast growing function.

$$A_K(j) = \begin{cases} j+1, & \text{if } K = 0 \\ A_{K-1}^{j+1}(j), & \text{if } K > 0 \end{cases}$$

$K \geq 0 \quad j \geq 1$.

$A_k^i(j) = A_k(A_k^{i-1}(j))$

$A_k^1(j) = A_k(j)$

$A_k^i(j) = j$

$A_0(1) = 2$

$A_1(1) = A_0^2(1) = A_0($

How fast does $A_k(j$

Theorem 1:

$A_1(j) = A_0^{j+1}$

$= A_0($

$A_0^i(j) = j+$

$i = 1$

$A_0(j) = j + 1$

~~Similarly~~ Suppos

$A_0^{j+1}(j) = A_0$

Theorem 2:

$A_2(j) = A_2^j$

$A_1^i(j) = 2^i($

Proof by

Base $i = 1$

$A_1(j) = = 2j + 1$ (Theorem 1)

$= 2^1(j+1) - 1$.

for i $A_1^i(j) = 2^i(j+1) - 1$

$A_1^{i+1}(j) = A_1(A_1^i(j)) = 2(A_1^i(j)) + 1$.

$= 2(2^i(j+1) - 1) + 1$

$= 2^{i+1}(j+1) - 1$.

proved for $(i+1)$.

$\Rightarrow A_2(j) = A_1^{j+1}(j) = 2^{j+1}(j+1) - 1$.

$A_0(1) = 2$

$A_1(1) = 3$

$A_2(1) = 7$.

$A_3(1) = A_2^2(1) = A_2(7) = 2^8(8) - 1 = 2047$

$A_4(1) = A_3^2(1) = A_3(A_3(1)) = A_3(2047)$

$= A_2^{2048}(2047)$

$= A_2^{2047}(A_2(2047))$

$= A_2^{2047}(2^{2048}(2048+1))$

$> 10^{700}$.

& $10^9 \approx$ no. of atoms in the universe

* A very slow growing functions:

$\alpha(n) = \max K$, such that
$A_K(1) \leq n$
$= \max(K, A_K(1) \leq n)$

$\alpha(n)$	Θn
0	
1	

18-02-19: $\alpha(n)$ · Slowly growing function

$\alpha(n) = \min\{K : A_K(1) \geq n\}$.

$n = 1$ $\alpha(n) = 0$
$n = 2$ $\alpha(n) = 0$
$n = 3$ $\alpha(n) = 1$
$n = 4, 5, 6, 7$ $\alpha(n) = 2$
$n = 8, \ldots, 2047$ $\alpha(n) = 3$.
$n = 2048, \ldots$ $\alpha(n) = 4$

Our analysis uses $\alpha(n)$ and proves that

$\hat{c} = O(\alpha(n))$

which is practically a constant.

m: Operations, n are makesets.

Makeset
Union } → (m-n)
Findset }

→ Link, findset.

Thus $m = n$ link + 2 findset.

m' operations such that they are independent i.e. makeset, link, findset.

$$m' \leq 3m = O(m)$$

m: n makesets + m-n links & findsets.

$$\phi(F) = \sum_{x \in F} \phi(x)$$

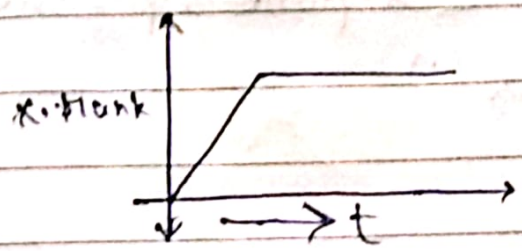
$\phi_q(x)$: potential of x after q^{th} operation.

node is a leaf :

$\phi(x) = d(n) \cdot x.rank$

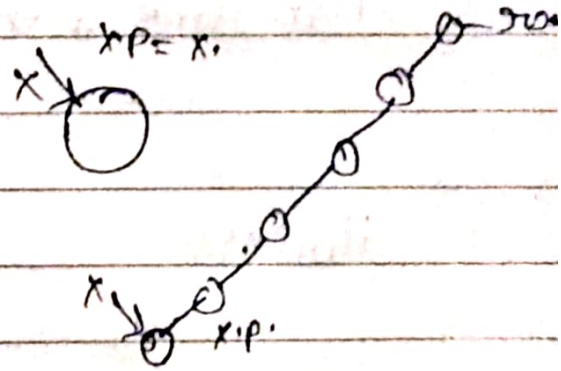
node is a root:

$\phi(x) = d(n) \cdot x.rank$

① 

monotonically increases till it ceases to be a root & then remains constant.

② As we go on a path from x to the root, ranks strictly increases atleast by 1 at a time



③ x.p.rank monotonically increases over time & then remains constant.

* ~~level~~ level(x):

What is the level (k) of Attr A_k that can be applied to it & still let it remain $\leq$ rank of its parent.

$level(x) = max(k : A_k(x.rank) \leq x.p.rank)$

* iter(x): How many times can $A_{level}(x)$ be applied to x.rank & still remain $\leq x.p.rank$.

no leaf nodes

level(x): $\boxed{0 \leq level(x) < d(n)}$

$A_0(x.rank) = x.rank + 1$

$\leq$ ~~x.rank~~ x.p.rank

$A_{iter}(x.rank) \geq A_{iter}(1)$

$\geq n$

④ Max. rank is(n) even without heuristics.

iter(x): $\boxed{1 \leq iter(x) \leq x.rank}$

$\boxed{iter(x) = \min\{j : A^j_{level(x)}(x.rank) > x.p.rank\}}$

$A^{iter(x)}_{level(x)}(x.rank) \leq x.p.rank$.

&

$A^{iter(x)-1}_{level(x)}(x.rank) > x.p.rank$

(by def. of level)

L.H.S = $A_{2^{rank(x)}}$ (x.rank) $\geq$ x.p.rank,

14-02-19 - Analysis of D.S. Forests with Ranked union & path compression
- Mergable heaps
- Binomial heaps:

non-root with leaf nodes:

$\phi(x) = (\alpha(n) - level(x))\,x.rank - rank(x)$.

$\phi_0(F) = 0$ n-

n-makeuts

after n makeuts

$\phi(F) = 0$.

Show that after all subsequent operations $\phi(x) \geq 0$

leaf nodes, root nodes } → have potential } non root nodes, non-leaf.

As in the above case $level(x) = \alpha(n) - 1$.

$\Rightarrow \phi(x) = (\alpha(n) - (\alpha(n) - 1))\,x.rank - rank(x)$

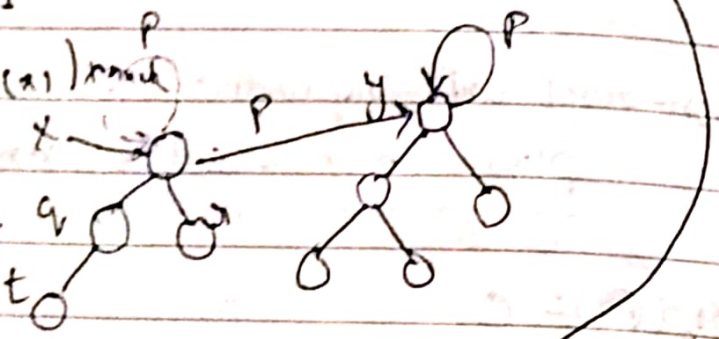
$= 1.\,x.rank - rank(x)$ → since new value is root

$\Rightarrow \phi(x) \geq 0$ (Always).

	c	$\Phi_q - \Phi_{q-1}$	$\hat{c}$
Makeset	1	0	1
Link	1	x: fall in Φ by at least 1. y: either 0, or $\alpha(n)$. z, w:	$1 + (-1) + \alpha(n)$ $= \alpha(n)$.

for $x : \Phi_q - \Phi_{q-1}$

$= (\alpha(n) - level(x)) \cdot x.rank$
$- iter(x)$
$- \alpha(n) \cdot x.rank$



$= - level(x) \cdot x.rank - iter(x)$.

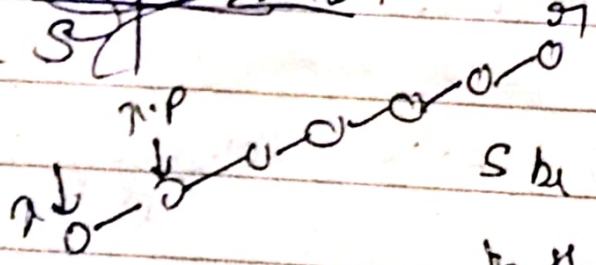
either 0, or.

if y's rank changes.

① level does not change iter
- iter changes so -1 or less.

② If level changes
$z.rank$ or $w.rank$.

Findset: S



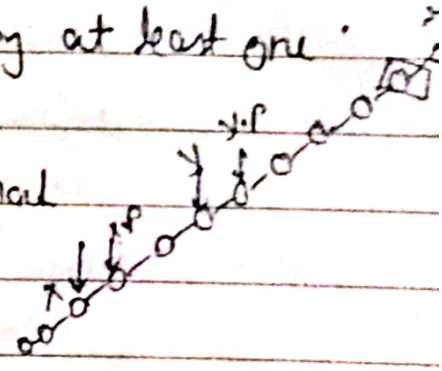
S is the number of nodes in the path from x to r

Claim: At least $s - 2 - \alpha(n)$ nodes have a drop in potential by atleast 1.

Suppose it was true:

Findset $\{ s \mid -(s-2-\alpha(n)) \mid 2+\alpha(n) \cdot \alpha(\alpha(n)) \}$

Let x be a node such that it has level as level(x) and somewhere higher in the path their is a node y such that level 'x' is equal to level 'y' i.e. (level(x) = level(y)) then x has drop by at least one.



$x.rank < x.p.rank \leq y.rank < y.p.rank \leq r.rank$

$A_{level(x)}^{iter(x)} (x.rank) \leq x.p.rank \leq y.rank < y.p.rank \leq r.rank.$

$A_{level}^{iter(x)+1} > x.p.rank.$

$A_{level(x)} (x.p.rank) < A_{level(x)}^{y.rank} \leq y.p.rank \leq r.rank$

DATE: 15-02-2019
PAGE:

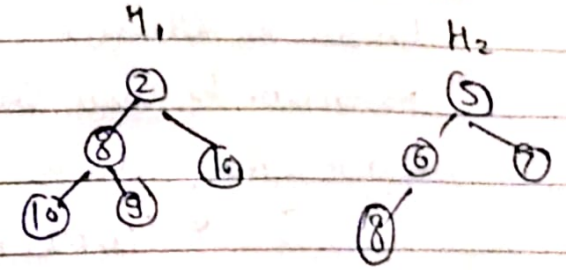
Mergeable Heap:
- Binomial heaps
- ~~B~~ Fibonacci heaps

Heaps: Priority Queues
Scheduling

Mergeable Heaps:
Makeheap (H)
Insert (H, x)
Minimum (H) # return the node with minimum value
Extract min (H) return x, H = H - {x} (x is the min element)
Union (H_1, H_2) → return (H), where H is $H_1 \cup H_2$
Decrease Key (H, x, k)
if k < x.key
x.key = k
Delete (H, x)
H = H - {x}
return (H)

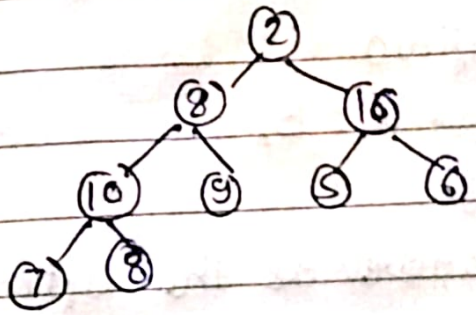
Binary heap:

Make	O(1)
Insert	O(logn)
Minimum	O(1)
Extract min	O(logn)
Decrease key	O(logn)
Delete	O(logn)



H

O(n) | 2 8 16 10 9 5 6 7 8 |



$\frac{n}{2} \rightarrow 0$

$\frac{n}{4} \rightarrow 1$

$\frac{n}{8} \rightarrow 2$

$\frac{n}{2^{\log_2 n}} (\log n - 1)$

Total # steps:

$$\frac{n}{4} \cdot 1 + \frac{n}{8} \times 2 + \cdots + \frac{n}{2^i}(i-1) + \cdots \frac{n}{2^{\log_2 n}}(\log_2 n - 1)$$

$$= \frac{n}{4}\left[1 + \frac{1}{2} + \frac{1}{2^2} + \cdots \frac{1}{2^{\log_2 n}}\right] + \frac{n}{8}\left[1 + \frac{1}{2} + \frac{1}{2^2} + \cdots \frac{1}{2^{\log_2 n - 1}}\right]$$

$$+ \frac{n}{16}\left[1 + \frac{1}{2} + \cdots + \frac{1}{2^{\log n - 2}}\right]$$

$$\leq \frac{n}{4}[2] + \frac{n}{8}[2] + \frac{n}{16}[2] + \cdots$$

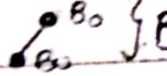
$$\leq 2n\left[\frac{1}{2} + \frac{1}{4} + \frac{1}{8} \cdots\right] \leq 2n \cdot \frac{1}{2}$$

Binomial Heap:

Union is O(log n)
Minimum is also O(lg n)
& everything else is also O(lg n)
(context of heap).

* Binomial Tree (basis for a binomial heap)
  ↳ Ordered tree
  recursively defined
  Binomial tree of degree k: B_k
  B_0: • (single node).
  B_i: One B_{i-1} is added as the leftmost child of another B_{i-1}.

B_0 : •

B_1 : 

B_2 : 

B_k : 

① The number of nodes in B_k is 2^k
- Induction proof
  B_0 : $2^0 = 1$ base
  Let B_i : 2^i
  Then B_{i+1} : $2^i + 2^i = 2^{i+1}$.

(2) Height of $B_K = K$

B_0 : ht is 0.

Let B_i : height is i

B_{i+1} : because one B_i is connected as child

so i+1.

(3) The number of nodes at level i in a B_K tree is ${}^{K}C_i$.

$K=0 \quad {}^{0}C_0 = 1$

Assume, true for K

${}^{K}C_i$ # of nodes in level i of B_K.

Then number of nodes in level i of B_{K+1} is:

$${}^{K}C_i + {}^{K}C_{i-1} = {}^{K+1}C_i$$

(4) Max degree in B_K tree is K & root node has it; the child nodes are ordered as K-1, K-2, - - - , 0 from left to right and each of those is a root of a B_i tree, where i is its order number.

B_0 : trivially true.

B_{K+1} : let us assume it to true. → Prove it for B_{K+1}

↳ K-1 to 0 satisfy by our assumption.

call left most child 'K' (has degree K).

→ B_{K+1} tree satisfies.

$2^k = n$

$k = \log_2 n$

↓

height.

19-02-19 Binomial Heaps: ⟶ Mergable Heaps.

↓

Union operation

- Binomial Tree
- A binomial heap
- Operation.

B_k : Binomial tree of degree k.

↓

2^k ⟶ height : $\log_2 n$

↓

k

# of nodes at level i in $B_k = {}^kC_i$.

root : max degree k.

child nodes : k-1 to 0 ordered

degrees : also k-1 to 0 respectively.

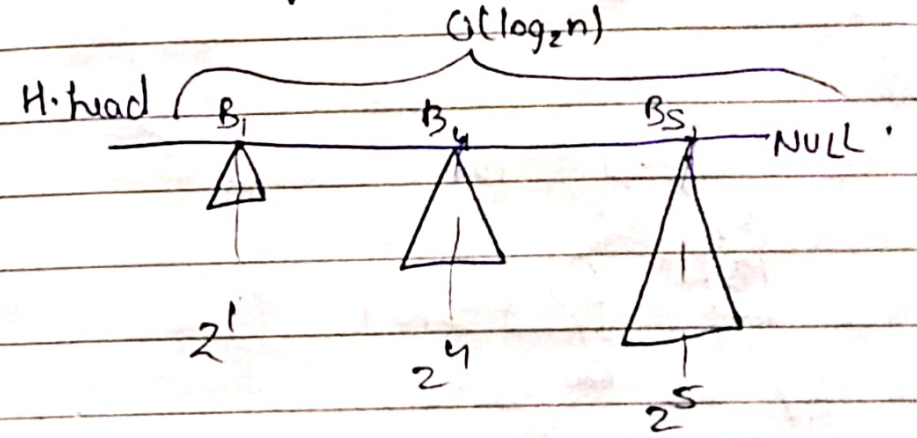
Heap property: (Min)

Every node in the B_k tree satisfies the property that its key is less than or equal to all its child nods keys.

$50 = (1\,1\,0\,0.1\,0)_2$ → length: $\lceil \log_2 n \rceil$.

B_5 & B_4 & B_1.

linked list of these three binomial trees.

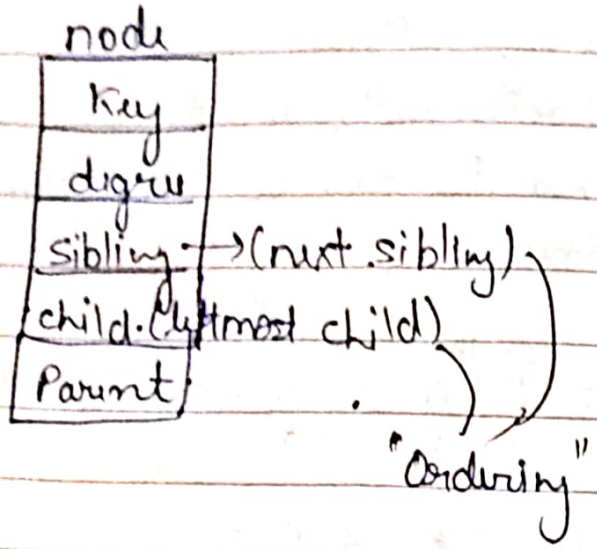


Binomial Heaps:

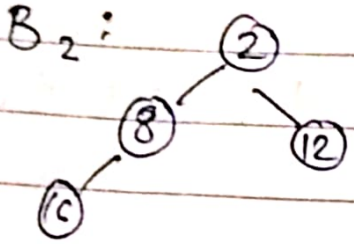
Linked list of Binomial tree corresponding to 1's in the binary rep. of n.

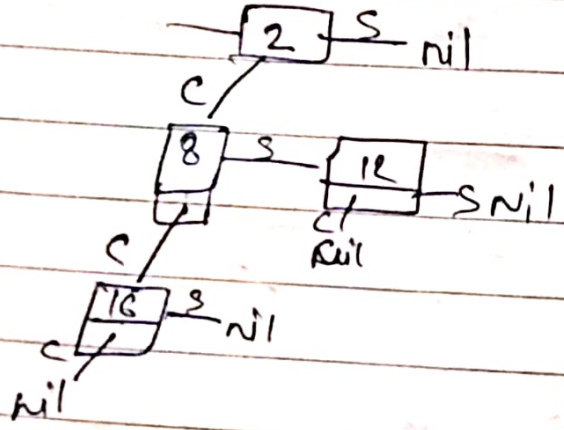
Each of the satisfies the heap property.

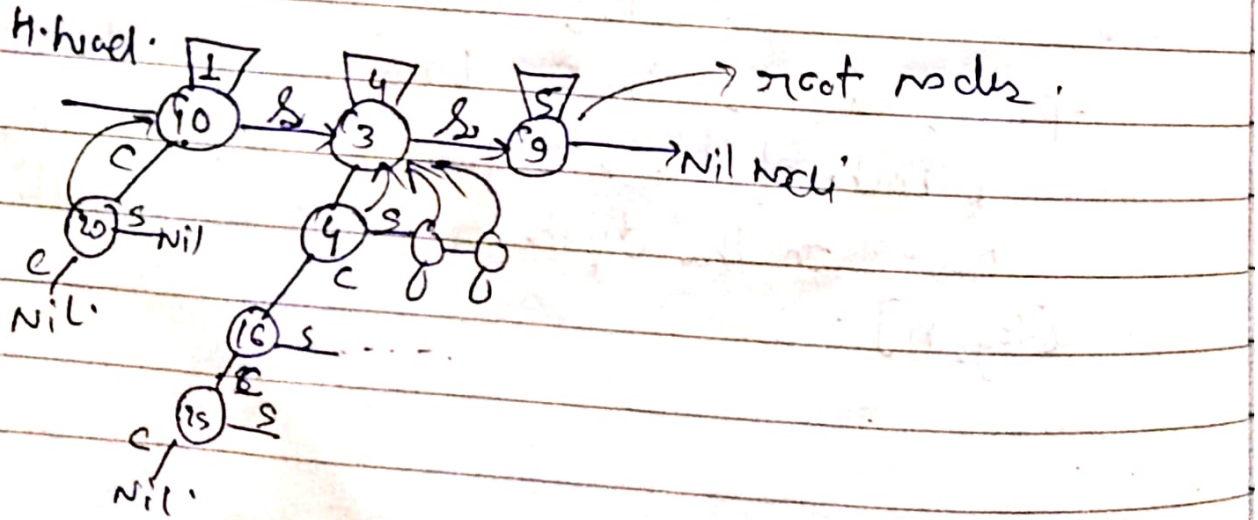
Bound on the depth of any tree in this list is $\lceil \log_2 n \rceil$.



B_2 :







Heap H : head field → nil or points to first root node

root node list : sorted by degree

MakeHeap (H)

H.head = NULL

MakeHeap (H, x) → fields of x

x.S = nil

x.C = nil

x.p = nil

x.degree = 0

H.head = x

#rigid structures always preserved.

Minimum (H) #returns the min element

search in the sibling root list (sibling pointer)

return minimum element.

} O(log n)

Binomial link (x, y) # called with node pointers x & y, where roots of binomial heaps of binomial heaps of degree (x.degree)

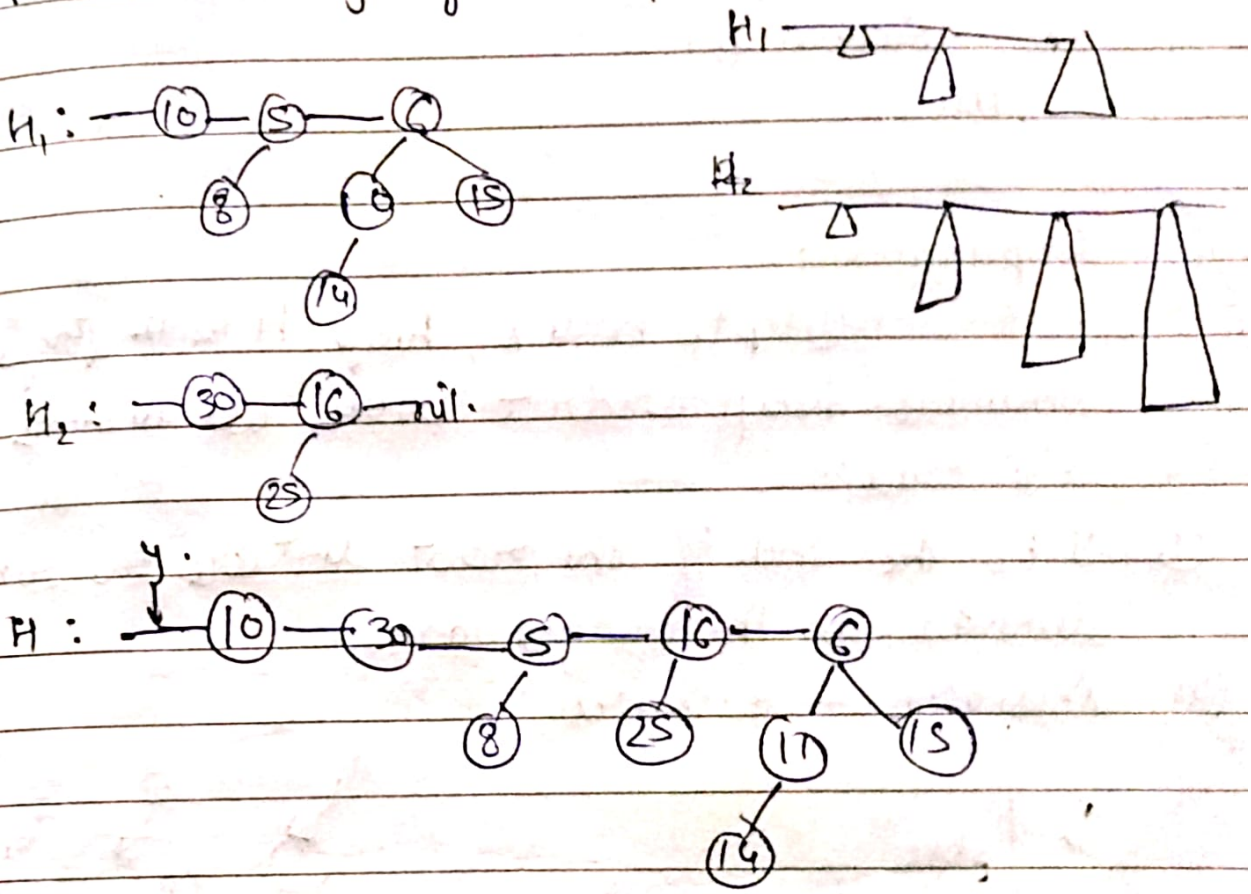
return a binomial tree of degree (x.degree+1) by linking these 2 trees & heap property retained.

```
Binomial link (x,y)
    if (x.key > y.key)
        x.s = y.c
        y.c = x
        x.p = y
        y.degree = y.degree+1
        return y.
    else
        y.s = x.c
        x.c = y
        y.p = x
        x.degree = y.degree+1
        return y.
```

Binomial link is going to have a complexity of O(1).

Union (H_1, H_2)

Initialize heap H. Merge the root lists H_1 & H_2 into H. In sorted by degree order.



start traversing from the left & merging 2 trees of equal degree into one of higher degree.

loop invariant: (1) All roots upto x are maintained and ~~have degree less than the next tree~~ in correct binomial heap order and have degree less than next tree (2) y is the next node & the root list from y is in the 'merged' order only.

(3) x.degree ≤ y.degree.

n(H_1, H_2)

x = nil

= H.head

(y == nil)

return (H)

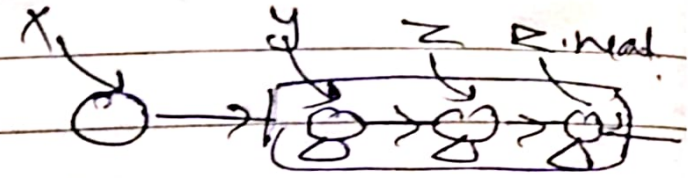
se

invariant:

① Upto node x, heap H is in proper
nial heap order & covers all elements upto
tree.

the end of the root list are the remaing
nts, in the merged order.

gree ≤ y.degree

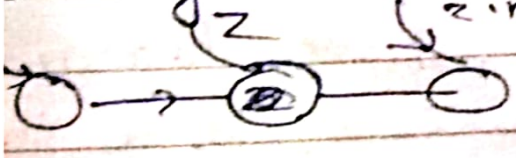


else

z = y.next

while (z != nil).

x.degree = y.degree, then there can be only
ts in y's list possibly, y & z, with
degree as x.



all same degree H.

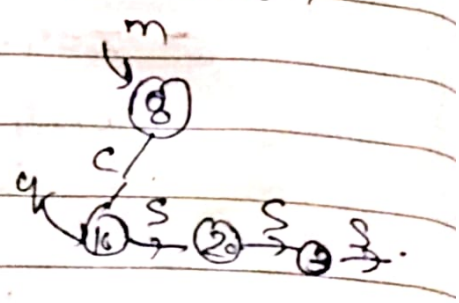
s degree = z degree & z.next is higher
degree/null.

③ y.degree < z.degree
↳ x.degree must have been < y.degree)

else
- z = y.next
while (z != null)
if (y.degree < z.degree)
# advance to the right
|| (y.degree = z.degree) && (z.next != null) &&
(z.degree == z.next.degree)
x = y
y = z
z = z.next
else
p = y
r = z
t = z.next
b = binomial_link(y, z)
x.p = b
b.s = t
if (x == nil)
H.head = b
else
r.s = b
y = b
z = b.s

Extract_Min (H)

1. m = minimum (H)
2. Remove m from the link list of H.head. (root list).
3. q = mc
4. Initialize heap H1
5. put childs in q into H1 in reverse order
6. H = Union (H, H1)
   return (m).
   m.c=nil, m.s=nil
   return (m).



→ and set parent to nil. i.e. m.c=nil

Decrease_Key (H, x, K)

give x

decrease its key to K.

only if x.key > K.

if (x.key > K)

x.key = K.

y = x.p

while ((y != nil) && (y.key > k))

exchange y & x.

Delete (H, x)

x.key = −∞ (Decrease key(H, x, −∞))

Decrease key (H, x, −∞) ⟶ log n

Extract min (H) ⟶ log n.

20-02-19

Fibonacci Heaps:

Mergeable Heap ⟷ Union is efficient.

↳ O(1) for almost all operations except extraction & delete : O(log n)

Like a binomial heap

↳ made of binomial trees in heap order.

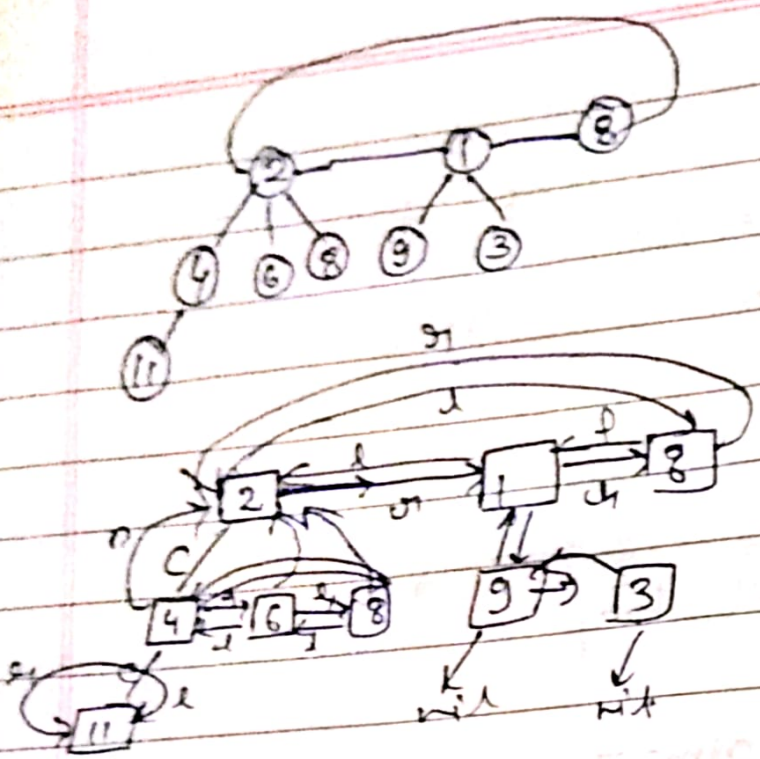
↳ Fib. Heap: (relax some of the very strict requirements of the binomial tree/heap).

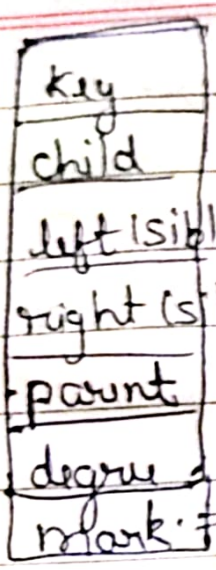
1) Each tree in the Fib. heap is a rooted tree, not ordered (like a "unordered binomial tree").

2) Each tree is "allowed" to be deficient in some nodes (in a "controlled" manner)

Root list – } Doubly linked circular list

Child list of a node – }





H
min (element in root list)
n
elements

Makeheap (H)
H.min = nil
H.n = 0
} O(1)

Minimum (H)
return (H.min)
} O(1)

"Lazy" operation.

Insert (H, x)
x.p = nil
x.c = nil
Slice H.min, list & insert x into it.
Set H.min to min of H.min and x (by key value)
H.n = H.n + 1.

Union (H_1, H_2)

slice the 2 root lists & join them } O(1)

$H_1.min = min(H_1.min, H_2.min)$

$H_1.n = H_1.n + H_2.n$

Extract_min (H)

remove H.min and

consolidate the heap

↳ (like a binomial heap).

Potential function:

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

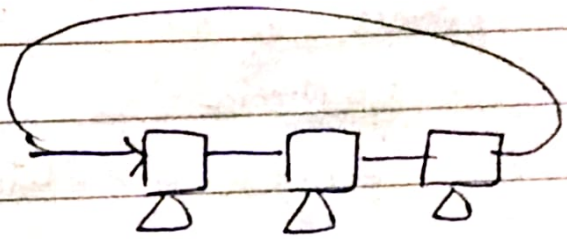
- $t(h)$: number of trees in the root list.
- $m(H)$: No. of "marked" nodes.

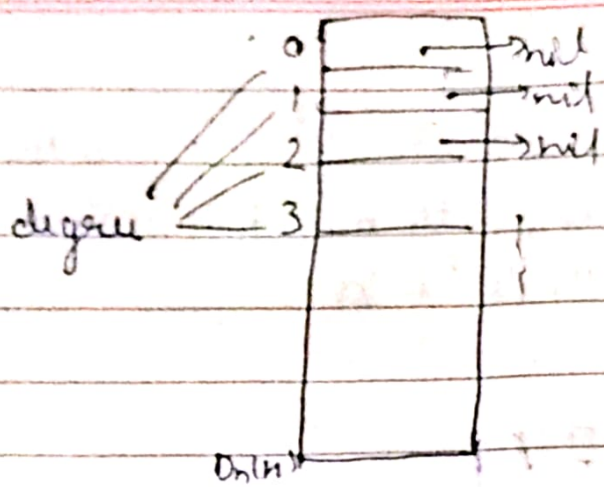
Mark	
1	→ ~~Not~~ marked
0	→ Not marked

Consolidate (H)

Array A[0 .... $D_n(H)$]

↳ where $D_n(H)$ is an upper bound on the degree of a node in an n node heap.





for every element in root list H if K is its degree
if A[x] == nil
A[K] = x
else

12-19 Fibonacci Heaps ⟶ lazy paradigm for consolidation.
Operations & analysis
more space (const. per node)

Extract_min (H)
y = H.min
if (y == nil) return y
else
z = y.right
if (z == y) H.min = nil
H.n = 0
return (z)

```
else
    remove y from root list
    m = consolidate (z)
        # return the new min element.
    H.min = m
    H.n = H.n - 1
    return (y)

Consolidate (z)
    min = z
    for each element x in the list of z
        if (x.key < min.key)
            min = x.
    d = x.degree
    while (A[d] != nil)
        q = A[z]
        if (x.key < q.key)
            fiblink (q, x)
        else
            exchange (x, q)
            fiblink (q, x)
        d = d + 1
    A[d] = X
    connect all the roots in A in circular form.
    Return (min)
```

fib_link (a, b)
a.p = b, b.degree = b.degree + 1
all other adjustments.

Time complexity: (of consolidate (z)).
$O(D_n(H) + n)$

Amortized Analysis:
potential difference:

$$\phi_{i-1} = t_{i-1}(H) + 2m(H)$$
$$\phi_i \leq D_n(H) + 2m(H)$$

$$\phi_i - \phi_{i-1} \leq O_n(H) - t_{i-1}(H)$$

Actual cost:
for loop: Actual cost = $t_{i-1}(H) + D_n(H)$
: in each while iteration
t falls by 1 & it cannot be less than 1.

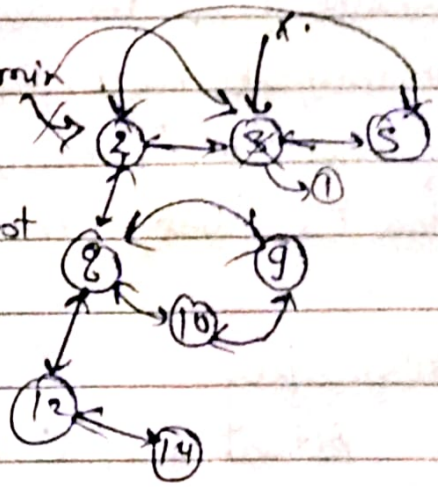
$$\hat{c} = t_{i-1}(H) + D_n(H) + D_n(H) - t_{i-1}(H)$$
$$= O(D_n(H))$$

DATE 22-02-19 Success
PAGE

Isolate y (Statement in the Extract_min)

Add the child list of y to the root list setting their parent pointer to nil.

Decrease_key (H, x, k)



when x is a part of root list just decrease & read the H.min if required.

when x is a non root node

Decrease the key & make the subtree rooted at x a part of the root list.

set parent to nil

update the H.min'

mark (x.p) # set x.p mark = 1

if x was marked, remove x's mark.

if x.p was a mark node

cut x.p & follow it up with a "cascady cut"

Cascading cut : Cut the node & put in root loop.

if parent is "marked" do the same with parent.

Till an unmarked node or "root" is reached, keep putting the nodes in rootlist & unmark them.

So, a cascading cut, would result in nodes along "p" pointers from the node x (and subtrees) to either an unmarked node, or root node to become part of rootlist (subtrees). The roots are unmarked & the last "unmarked" node is marked.

Decreasekey

Actual key: 'c' nodes were cut & made a part of root list.

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

$$\Phi_i(H) = t(H) + c + 2[m(H) - 2(c-1)]$$

Diff. in $\Phi = -c+2$

Acutual cost $= c$

$\hat{c} = 2$

Delete (H, x)

Success
DATE: .....
PAGE: .....