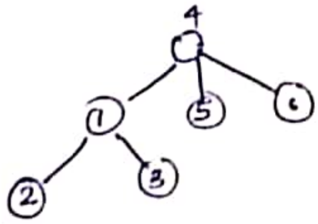
Rooted tree

For tree in which one vertex is designated as "root".

Suppose "4" is root



→ There is unique path from root to all nodes.

→ leaf nodes & internal nodes.

→ leaf node – not on path from root to any other node.

x is a child of y if y is immediately preceding node in path from root to x.

y is a parent of x if x is a child of y.

x is a descendant of y if y belongs to the path from root to x.

y is ancestor of x, if x is descendant of y.

Scanned by CamScanner

Degree in rooted tree → no. of child nodes.

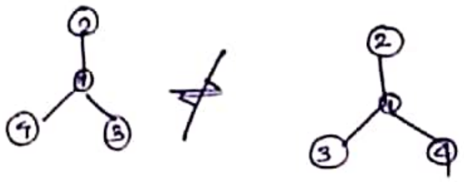
Depth of a node/vertex
↳ distance from the root.

height of ~~of~~ a vertex/node – length of longest path to a descendant leaf child.

height of the root – longest path to a leaf.

height of the tree – height of root.

Ordered tree – sequencing or "ordering" on the child nodes of every node.

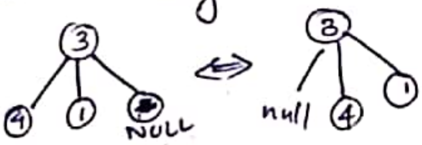


Positional Tree

a "position" for every child node.

- NULL tree
- "ariness" – number of child nodes for each non null node;
- recursive of a k-ary positional tree.

Positional 3-ary tree



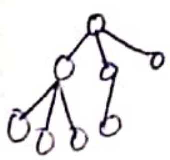
→ In a positional tree

k-ary tree
↓
no. of non null nodes & height.

Full tree : Every node has degree k or 0.
↓
leaf node.

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

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



No. of nodes in complete k-ary tree

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

h - height of the tree

No. of internal nodes $= \frac{k^h - 1}{k-1}$

No. of leaf nodes $= k^h$

Binary → 2-ary positional tree

left, right } name for 2 child slots.

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

leaf nodes $= 2^h$

Internal $= 2^h - 1$

BST: key → Unique : for every node x

$x.key > x.left.key$

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

Worst case height : $n-1$

Search O(height)

→ O(n)

h is $O(\log_2 n)$

⇕

Balance binary search tree.

Rotation

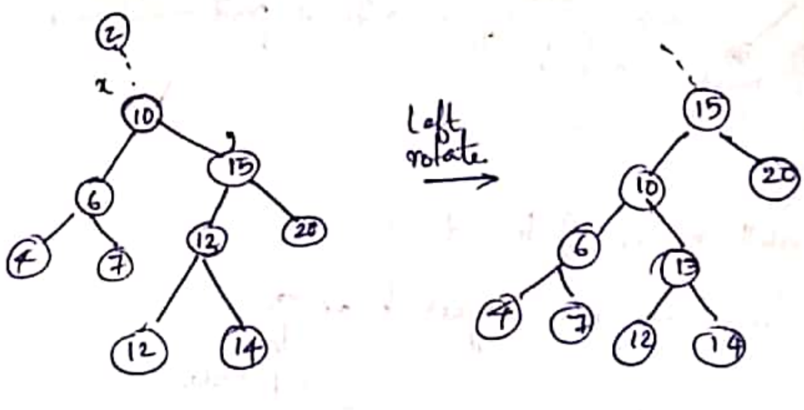
left rotate(x)

y = x.right

x.right = y.left

y.left = x

return y.



left-rotate(x) returns a ptr to a rotated subtree of nodes contained in subtree of x.

left-rotate(x)

x.right = left rotate (z.right)

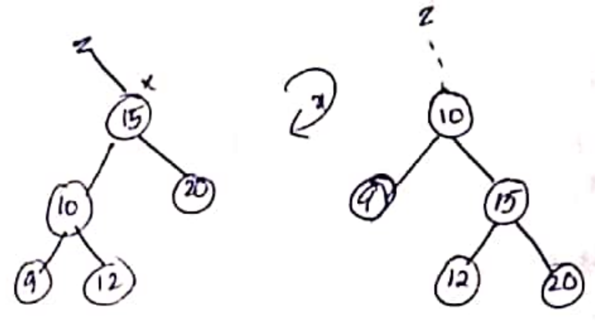
right rotate (x)

$y = x.left$

$x.left = y.right$

$y.right = x$

return(y)



Binary Search Tree - expected height is $O(\log n)$

Input Sequence

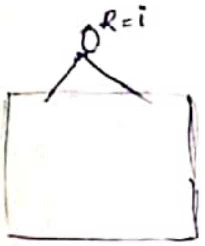
random variable $R_j \rightarrow$ root key of the BST made with j nodes (keys)

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

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

X_n be the height of the tree made with n nodes

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



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

Randomly Built BST

Insert / Search $\rightarrow$ height?

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

R_i = value of root in a r.b bst of i nodes.

X_n - height of a randomly built bst of i nodes.

$Y_i = 2^{X_i}$

$$Z_{j,i} = \begin{cases} 1 & i \text{ the root node of a r.b bst of } j \text{ node.} \\ 0 & \text{otherwise} \end{cases}$$

$R_n = i \rightarrow$ just key

RB BST   RB BST

$\{ i, i-1, \ldots n \quad , n-1$

$i-1$ node $i+1$ $(n-i)$ node

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

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

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

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

$$= 2 \cdot \max\left(Y_{i-1}, Y_{n-i}\right)$$

$$Y_n = 2\max(Y_{i-1}, Y_{n-i})$$

$$E(Y_n) = 2 \cdot E(\max(Y_{i-1}, Y_{n-i}))$$

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

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

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

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

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

~~find $E(Y_i)$ for each of the i's~~

Find $E(Y_i)$ for each of the i's

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

Base case $n = 1$

$$E(Y_1) = 1$$

Scanned by CamScanner

We'll prove that

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

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

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

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

$$= {}^6C_4 + \cdots + {}^{n+2}C_3$$

$$= {}^{n+1}C_4$$

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

$$\le \frac{4}{n} \sum_{i=0}^{n-1} \frac{1}{4} {}^{i+3}C_3$$

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

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

$$\le \frac{1}{4n} {}^{n+3}C_4 \le {}^{n+3}C_3 \quad \underline{\text{Proved}}$$

$$\frac{1}{4} \frac{1}{x} \frac{(n+3)(n+2)(n+1)/n}{2 \cdot 3 \cdot 4}$$

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

$$\frac{1}{4x} \frac{(n+3)(n+2)(n+1)x}{2 \cdot 3 \cdot 4}$$

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

$$E(Y_n) \le \frac{1}{4n} {}^{n+3}C_3$$

↓

bound X_n

$$E(X_n) \le \log_2 (E(2^{X_n}))$$

$$\le c\left[{}^{n+3}C_3\right] \le \log(c_1 n^3 + c_2 n^2 + c_3 n^1 + \ldots)$$

$$\boxed{E(X_n) \text{ is } O(\log n)}$$

↓

height Y_n – exp ht

<u>Worst Case</u> $n-1$

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 and right subtree should differ by atmost

always calls for a re-adjustment to satisfy AVL Property.

– self adjusting data structure.

height : h

$N(h)$ – no. of nodes in AVL Tree of height h

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

$N(0) = 1$

$N(1) \geq 2$

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

$$N(h) \approx C_1\left(\frac{\sqrt{5}+1}{2}\right)^h - C_2\left(\frac{\sqrt{5}-1}{2}\right)^h$$

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

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

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

$h < \log(n)$

Balance factor = |height of x.left – height of x.right|

≤ 1 (self adjusting tree)

Node structure

AVL tree

key	
l	r
height	

root_left(x)

```
y = x.right
if (y != nil)
    x.right = y
    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)
```

T.root = AVL_Tree_Insert (T.root, x)   // x is address of node to be inserted.

AVL_Tree_Insert (r, x)   // return val. an AVL Tree root formed with nodes in the subtree rooted at r (which is AVL) and x

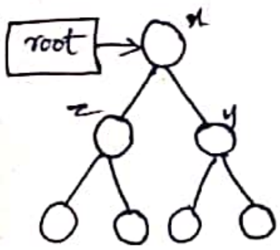
```
if (r == NIL)
    return (x)
else
if (x.key < r.key)
    r.left = AVL_Tree_Insert (r.left, x)
else
    if (x.key > r.key)   r.right = AVL_Tree_Insert (r.right, x)
if ((r.left - r.right.height) > 1)   r = right rotate (r)
else if ((r.left.height - r.right.height) < -1)   r = left_rotate (r)
    r.height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

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

$\boxed{T(h) = O(h) = O(\log n)}$ → Insert

↓

Always (in all cases)



A.root = AVL Tree Insert (A.root, x)
↓
NIL

```
left_rotate (x)
y = x.right
if (y == nil)
    return x
else
    x.right = y.left        x.height = 1 + max (x.left, y.left)
    y.left = x              y.height = 1 + max (x, y.right)
    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 the ptr to the modified tree

AVL_Tree_Delete (r, x)   # delete x from subtree rooted at r.

```
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 = leftrotate (r)
                       return y.
              else if (x.key > r.key)
                       r.right = AVLtree delete
                       if r.right - r.left < -1
                               y = right rotate (r)
                               return y.
  if (x.left == NIL) AND (x.right == NULL)
        return NIL
  else
     if (r.left == NIL)
            return (r.right)
  else if (r.right == NIL)
```

```
minimum (y)            // find and returns minimum key node in
  if (y.left == nil )     subtree rooted at y.
      return (y)
```

Scanned by CamScanner

else
return (minimum (y.left))

y = minimum (x.right)
x = AVL-Tree-Delete (x.right, y)
y.right = x.
y.left = x.left
y.h = 1+ max (y.r.h, y.l.h)
// restore AVL Property at y.
return y.

⇒ DISJOINT SETS

1) Union
2) Findset (u): return the representative element (for a set there should be only one representative)
   ↳ $\boxed{O(1)}$
3) Makeset (u): makes a set out of a given key (element) & returns the representative itself.
   ↳ $\boxed{O(1)}$

{a} {b} → findset(a) ≠ findset(b)
{a} ∪ {b} → findset (a) = findset (b)

Connected Components

Build up the sets, each connected component is a different set.

Connected components (G) // building connected components.
  for each v ∈ G.V
    makeset (v)
  for each edge (u, v) ∈ G.E
    union (u, v)

⇒ Is x connected to (x, y) ?

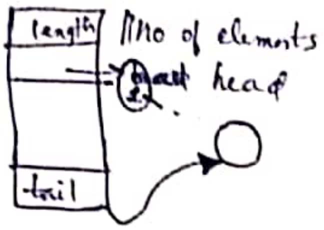
IsConnected (x, y)
  return (findset (x) == findset (y))

n elements in all

What is the worst case union scenario here ? i.e. the total cost of all unions (n - 1 operations)

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

⇒



makeset (u)

O(1) { allocate a header and put u as first and tail element, next = null
put length = 1 rep ptr to u itself
return (pointer to u)

findset (u)

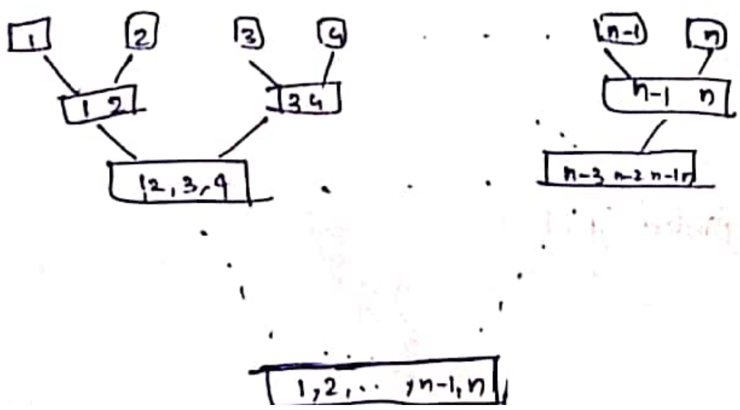
return rep(u)

Union (u, v) (weighted union heuristic).

x = findset (u)
y = findset (y)
if (x = = y)
return (u.rep)

Linked list - Weighted union heurestic
↓
What is its worst case complexity

Assume n makesets and then union resulting in 1 set of n element



No. of unions	Cost of union
n	1
n/2	2
n/4	⋮
1	n/2

Scanned by CamScanner

Total number of operation $= \frac{n}{2}\cdot 1 + \frac{n}{4}\cdot 2 + \frac{n}{8}\cdot 4 + \frac{n}{16}\cdot 8 + \cdots + 1\left(\frac{n}{2}\right)^2$

$$= (\log_2 n)\, n/2 = O(n\log n)$$

The avg cost is $O(\log n)$

Disjoint Forest

Trees

- Every set is a tree

```
Link(u,v)        Findset(u)              Union(u,v)
  u.p = v          while (u.p ≠ u)         ~~Link(u,v)~~
                       u = u.p             link(findset(u), findset(v))
                   return u
```

Disjoint Set with Disjoint Forest Implementation

Ranked Union Heuristic

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

parent
data
rank

```
Makeset(u)       findset(u)               Link(u,v)
  u.p = u          while(u.p ! = u)         if u.rank < v.rank
  u.rank = 0           u = u.p                  u.p = v
                   return(u)                else
                                                v.p = u
                                                if (v.rank == u.rank)
                                                    u.rank = u.rank + 1
```

Union - with ranked union heuristic

```
Union(u,v)              Findset(x)
  p = findset(u)          if (x.p == x)
  q = findset(v)              return x
  link(p,q)               else
                              x.p = findset(x.p)
```

Sum up union costs

no. of union	cost of single union
$n/2$	2×1
$n/4$	2×2
$n/8$	2×3
$n/16$	2×4
⋮	⋮

Σ costs $= O(n\log n)$

→ Path Compression Heuristic

→ Analysis

First study about very fast growing funct^n

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

$(k \geq 0, j \geq 1)$

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

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

$$A_k^0(j) = j$$

$A_0(1) = 2$

$A_1(1) = 3$

How fast does $A_k(j)$ grow?

Theorem 1

$$\boxed{A_1(j) = 2j+1}$$

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

Prove it by induction

Base $i=1$

$$A_1(j) = 2j+1 \quad \text{(Theorem 1)}$$

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

Inductive step

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

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

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

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

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

$$\boxed{A_2(j) = 2^{j+1}(j+1) - 1}$$

Scanned by CamScanner

$A_0(1) = 2$

$A_1(1) = 3$

$A_2(1) = 7$

$A_3(1) = 2047$

$A_4(1) = A_3\left(A_3^{2047}(2047)\right) > 10^{90}$

A very slow growing function :

$\alpha(n) = \cancel{\max k \text{ such that } A_k(1) \leq n = \max(k;\ A_k(1) \leq n)}$

$\alpha(n)$: slowly growing funct^n

$\alpha(n) : \min\{k : A_k(1) \geq n\}$

$\alpha(n)$	n
0	1
0	2
1	3
2	4, 5, 6, 7
3	8, 9, ..., 2047
4	2048, .

Our analysis uses $\alpha(n)$ and proves that $\hat{c} = O(\alpha(n))$ which is practically a const.

m : operations n are makesets.

Makeset

Union
Findset $\}(m-n)$

link, Findset

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

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

1 Union = 1 link + 2 findsets.

m : n makesets + (m-n) links and findsets.

$\phi(F) = \sum_{x \in F} \phi(x)$, where x is a node.

$\phi_q(x)$: potential of x after q^{th} operat^n

node is a leaf

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

node is a root

① $\phi(x) = \alpha(n) \cdot x.rank$

x.rank

↳ monotonically increases till it ceases to be a root and then remains constant

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

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

level (x)

what is the level (k) of A_k that can be applied to it & still let it ~~remain~~ 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 and still remain $\leq$ x.p.

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

$$A_{\alpha(n)}(x.rank) \geq A_{\alpha(n)}(1) \geq n > x.p.rank$$

E
proof
very
pp

④ Max rank is n-1 even without heuristics.

iter(x) :

$$iter(x) = \max\{i \mid A^{i}_{level(x)}(x.rank)$$

Then $A^{0}_{level(x)}(x.rank) < x.p.rank$

$$x.rank < x.p.rank.$$

$$A^{1}_{level(x)}(x.rank) \leq x.p.rank.$$

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

(By def of level)

LHS $= A_{level(x)}^{x.rank+1}(x.rank) \geq x.p.rank$

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

Potential Method

$\alpha(n) = \min(k \mid A_k(1) \geq n)$

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

root leaf $\phi(x) = \alpha(n) \cdot x.rank$

$level(x): \max(k \mid A_k(x.rank) \leq x.p.rank)$ $\quad 0 \leq level(x) \leq \alpha(n) - 1$

$iter(x): \max(i \mid A_k^i(x.rank) \leq x.p.rank)$ $\quad 1 \leq iter(x) \leq x.rank$

$\cancel{0 \leq level(x) \leq \alpha(n) - 1}$

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

① increase monotonically till becomes a non-root node

② rank strictly increase in a path

③ rank of parent monotonically increases; potential method.

$\boxed{\phi(x) = (\alpha(n) - level(x))\, x.rank - iter(x)}$

$\phi_0(F) = 0$ $\quad$ m-operations

n - makesets

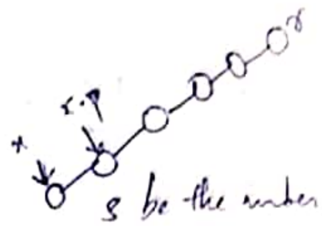
After n makesets, $\phi(F) = 0$

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

leaf nodes, root nodes } +ve potential $\quad$ } non root non leaf

	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: either 0, or, if y's rank changes (① level does not change, iter changes).	

Findset	c	$\phi_q - \phi_{q-1}$
	s	$-(s-2-\alpha(n))$
		$O(\alpha(n))$



s be the number of nodes from on the path from x to p.

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

Suppose it was tree.

Let x be a node such that it has level as ~~half~~ level(x) & somewhere higher in the path there is a node y such that level (x) = level (y)

Then x has drop by atleast 2.

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

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

iter(x)

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

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

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

$$level(x) = level(y)$$

Mergeable Heaps

- Binomial heap
- Fibanocci heap.

Operations

Makeheap (H)

Insert (H, x)

Minimum (H) – return the node with min. value.

Extract min (H) – return x

$H = H - \{x\}$ // x is the min. element.

- Union $(H_1, H_2) \rightarrow$ return (H) where H is $H_1 \cup H_2$

Decrease key $(H, x, k) \rightarrow$ if$(k < x.key)$ $x.key = k$

Delete$(H, x) \rightarrow H = H - \{x\}$

10 20 30 40 50

Binary Heaps

Makeheap	$O(1)$
Insert	$O(\lg n)$
Minimum	$O(1)$
Extract min	$O(\log n)$
Decrease key	$O(\log n)$
Delete	$O(\lg n)$
Union	$O(n)$

Total no. of steps $= \frac{n}{4} \cdot 1 + \frac{n}{8} \cdot 2 + \frac{n}{16} \cdot 3 + \cdots + \frac{n}{2^{\log_2 n}} (\log_2 n - 1)$

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

$$+ \frac{n}{16}\left[1 + \frac{1}{2} + \cdots + \frac{1}{2^{\log_2 n - 2}}\right)$$

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

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

Binomial Heap

Union is $O(\log n)$

Minimum is also $O(\log n)$ and everything else is also $O(\log n)$

(Content of heap)

Binomial Tree (basis for a binomial heap)

↳ ordered tree

recursively defined.

Binomial tree of degree k : B_k

$B_0 = \bullet$ (single node)

B_i is one B_{i-1} added as the leftmost child of another B_{i-1}

1) The number of nodes in B_k is 2^k - inductive 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 : ht is i

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

3) The no. of nodes at level i in a B_k tree is ${}^kC_i$

$k = 0$ ${}^0C_0 = 1$

Assume true for k : ${}^kC_i$ no. of nodes in level i of B_k

4) Max. degree in B_k tree is k, if the root node has it, the child nodes are ordered as $k-1, k-2, \ldots, 0$, 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 : let us assume it is true — prove true for B_{k+1} → $k-1$ to 0, satisfy

call leftmost child "k". by assumpt^n

→ B_{k+1} tree - satisfies

$$\boxed{\begin{aligned} 2^k &= n \\ k &= \log_2 n \end{aligned}}$$

↓

height

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 nodes' keys

key
degree
sibling → next sibling
child → left most child
parent

"ordering"

Heap field → nil or points to first root node.

root-node list → sorted by degree.

Makeheap(H, x) → fields of x

$x.s = nil$

$x.c = nil$

$x.p = nil$

$x.degree = 0$

$H.head = x$

Minimum (H) #return the mini element

↓ //search in the singly linked root list

O(log n) (sibling pointer)

and return min. element.

Binomial Link (x, y) ⟶ O(1)

// called with node pointers x and y, where x.degree = y.degree & they are roots of binomial heaps of degree (x.degree) return a binomial tree of degree (x.degree + 1) by linking these 2 trees, and heap property retained.

if (x.key > y.key)

$x.s = y.c$

$y.c = x$

$x.p = y$

$x.degree = y.degree + 1$

~~return y~~

return y.

else

//Symmetric code.

$y.s = x.c$

$x.c = y$

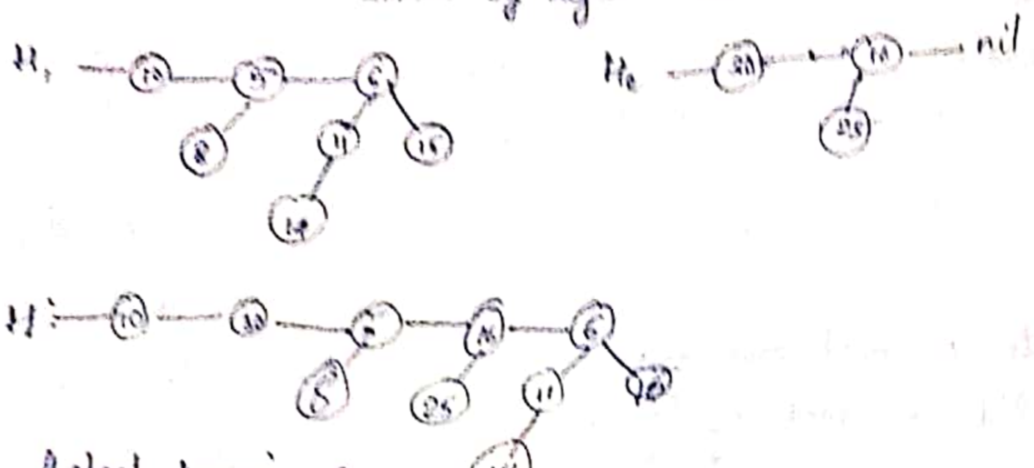
$y.p = x$

$y.degree = x.degree + 1$

return x.

Scanned by CamScanner

Union (H_1, H_2) // Initialize heap H. Merge the root lists H_1 and 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 : All roots upto x are maintained in correct Binomial heap order and have degree less than the next tree. y is the next node and the root list from y is in the "merged" order only.

Union (H_1, H_2)

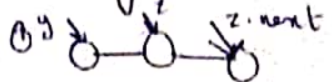
H - Merged lists of H_1 and H_2 in degree sorted order.

x = nil
y = H.head
if (y = = nil) return (H)
else
: z = y.next
while (z != nil)
if ((y.degree < z.degree)
(// advance to right) ~~// advance to the right~~
|| ((y.degree = z.degree) &&
(z.next != null) && (z.degree = = z.next.degree))
x = y
y = z
z = z.next

Loop Invariant

1. upto node x, heap H is in proper binomial heap order and covers all elements upto node x's tree.
2. y to the end of the root list are the remaining elements in the merged order
3. x.degree ≤ y.degree

if x.degree = = y degree then there can be only two roots in y's list possible, y and z with same degree as x.



all same degree

2) if y's degree = z.degree & z.next is higher

3) y.degree < z.degree degree/null

Scanned by CamScanner

else

q = y
r = x
t = x.s
t = binomial_link (y, x)

y.s = h
h.s = t

if (x = = nil) H.head = h
else x.s = h

h.s = t
y = h
x = h.s

Extract_Min (H)

1. m = minimum (H)
2. // Remove m from the linked list of H.head (root list)
3. q = m.c
4. // Initialize heap H_1 and put elements in q into H_1 in reverse order.
5. H = Union (H, H_1)
6. m.c = nil , m.s = nil
7. return (m)

Decrease-key (H, x, k)

// give a 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 and x.

Delete(H, x)

x.key = −∞

Decrease key (H, x, −∞)

Extractmin (H)

~~Dhe~~

Fibonacci Heaps

Fibonacci Heaps ——— O(1) for almost all operations except extract" & delete : O(log n)

- Mergeable Heap → Union is efficient
- → relax some of the very strict requirements of the binomial tree/heap.

1) Each tree in the Fibonacci tree is a rooted tree, not ordered (like un unordered binomial tree)

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

Root List, Child list of a node } (Doubly linked circular list)

key
child
left (sibling)
right (sibling)
parent
degree
mark ← "false" status.

H

min	→ points to min. element in root list)
n	→ no. of elements.

Makeheap (H) $\boxed{O(1)}$

H.min = nil

H.n = 0

Minimum (H) $\boxed{O(1)}$

return(H.min)

Insert (H, x) // Lazy operation

x.p = nil

x.c = nil

// Consider nil case too

Splice H.min's list and insert x into it

Set H.min to min of H.min and x (l.g, key value)

H.n = H.n + 1

Union (H_1, H_2) $\boxed{O(1)}$

Slice the 2 root lists & join them

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

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

Extract_min (H)

remove H.min and consolidate the heap
like a binomial heap.

Potential function

$\phi(H) = t(H) + 2m(H)$ m - No. of marked nodes.

t - No. of trees in the root list.

Initially all nodes are unmarked.

Insert (H, x)
$\hookrightarrow$ x.mark = 0.

Consolidate (H)

Array $A[0 \ldots D_n(H)]$ → where $D_n(H)$ is an upper bound on the degree of a node in an n node heap.

$\downarrow$
$\lceil \log_2 n \rceil$

~~for every element in root list let k be its degree~~

1) Initialize array to NIL
$O(D_n(H)+1)$

for every element x in root list let k be its degree

if $A[k] == nil$ $A[k] = x$

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.
```

// consolidate the root list

m = consolidate (z)

// return the min element

H.min = m

H.n = H.n − 1

return (y)

↳ Upper bound on the degree of a node when there are n nodes.

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[d]

if (x.key < q.key)

  fiblink (q, x)

else

  exchange (x, q)

  fiblink (q, x)

d = d + 1

connect all the roots in A in circular term.

$\boxed{O(1)}$ = fib_link (a, b)

a.p = b

b.degree = b.degree + 1

all other adjustments

Time complexity

$O(D_n(H) + n)$

Amortized analysis

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

$$\phi_i \leq D_n(H) + 2m(H)$$

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

Actual cost :

↳ for loop : Actual cost = $t_{i-1}(H) + D_n(H)$

↓

in each while iteration t fall by 1 & it cannot be less

c_i

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

$= O(D_n(H))$

Decrease key (H, x, k)

When x is a part of root list just decrease & reset H. min if required.

When x is a non-root node -

$O(1)$
- Decrease the key & make the subtree rooted at x as a part of the root list
- Set parent to nil
- Update H.min

if x.p is "unmarked"

mark x.p. set x.p.mark = 1.
if x was marked, remove x's mark.

→ if x.p was a "marked node"
cut x.p & follow it with a "cascading cut"

Cut the node & put in root lists if parent is "marked"
do the same with parent.

Till an unmarked node or "root" is reached, keep putting the nodes in root list and unmarking them.

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

Decrease key:
Actual key: "c" nodes were cut & made a part of root list

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

$\phi_i(H) = t(H) + c + 2(m(H) - (c-1))$

Mele Mele
Mele
vannamayy
Mele Mele.
Mele
Vannavilli
Kunnaran
onnachevan
kotGopuran

Diff in ϕ. $-c + 2$

Actual cost $= c$

$$\boxed{\hat{c} = 2}$$

Delete (H, x)

Decrease key to $-\infty$

$D_i(H)$ { Extract Min

→ RED-BLACK TREES

1) Every node is either red or black
2) Root is always black.
3) Leaves always black
4) A red node can have only black children.
5) Every path from the root to a leaf has same no. of black nodes.

$bh(x)$ //black height of node x is the no. of black nodes (not counting itself) on any path from x to a leaf.

How is height related to no. of nodes?

Claim : $h \leq 2 \log_2 (n+1)$

Prove :- for every node x.

$$size(x) \geq 2^{bh(x)} - 1$$

Suppose  $bh(x) = 1$

$$size(x) = 1 \geq 2^1 - 1$$

Assume true upto (black hts) 1, 2, 3, . . . , b−1 for ht b.

→ Let r be the root node of RB tree

Let n be the no. of nodes in T

$$n \geq 2^{bh(r)} - 1$$

$$n \geq 2^{\frac{height(r)}{2}} - 1$$

$$n+1 \geq 2^{\frac{height(r)}{2}}$$

$$\log(n+1) \geq \frac{height(r)}{2}$$

$$ht \leq 2\log_2(n+1) = O(\log n)$$

rbinsertleft

⇒ RBTreeInsert (T, x)

```
if (T.root == NIL)
    x.colour = black
    T.root = x
else
    x.colour = red
    T.root = rbinsert (T.root, x)
```

rbinsert (r, x)

// The subtree with nodes in subtree of r with x included in it.

// called with a red x

```
if (r == NIL)
    return (x)
if (x.key < r.key)
    r.left = rbinsert (r.left, x)
else
    r.right = rbinsert (r.right, x)
if (r.left.left.colour == red
    || r.left.right.colour == red)
{   if (r.right.colour == RED)
        // case 1.
        r.left.colour = BLACK
        r.right.colour = BLACK
        r.colour = RED
}
// r.right was black
    y = r
    r = rightrotate(y)
    r.right.colour = red
    r.colour = black
    return (r)
```

```
if (r.right.colour == RED)
    if (r.right.left.colour == RED
        OR
        r.right.right.colour == RED)
        if (r.left.colour == RED)
            r.right.colour = BLACK
            r.left.colour = BLACK
            r.colour = RED
else
    y = r
    r = rightrotate (y)
    r.right.colour = black
    r.left.colour = black
```