OOP

- class:

basically a blueprint (functions and variables) (Data)

Note:

'this'

↓

pointer to

Eg: Matrix

If we overload + operator:

~~mat~~ operator + (matrix m_2) {

:

}

m_1.plus(m_2)

$m_1 + m_2$ → how it called

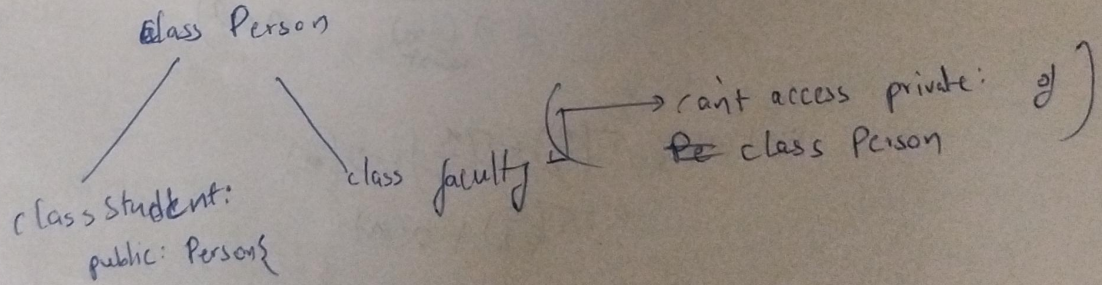
Note:

Friend functions can access put variables

- Inheritance:

- one class that is derived from another

Eg:



}

Eg: Person:

name;

DOB; } → if these are put, student won't be able to access it unless there's a public func" that allows to access it

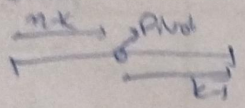
~~sta~~

- so we can use 'protected' instead of 'private'
  - basically private but derivatives can access

- virtual f()
  - Syntax to show that derivative may override the func^n
  - if virtual f() = 0 → function has to be overriden by the derivative.

* Quick sort Time cost

n-1 (excluding pivot)



$T(n) = T(n-k) + T(k-1) + O(n)$

worst case:

$T(n) = T(n-1) + O(n)$

$= T(n-1) + cn$

$= T(n-2) + c(n-1) + cn$

$\vdots$

$\therefore T(n)_{worst} = O(n^2)$

Best case: $T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + O(n)$

$= 2T\left(\frac{n}{2}\right) + O(n)$

$T_{Best}(n) = O(n \log n)$

Average case:

$k=1 \quad T(n) = T(n-1) + T(0) + O(n)$

$k=2 \quad T(n) = T(n-2) + T(1) + O(n)$

$\vdots$

$k=n \quad T(n) = T(0) + T(n) + O(n)$

$$T_{Avg}(n) = \frac{1}{n}\sum_{j=1}^{n-1} T_{Avg}(j) + cn + \frac{1}{n}\left(\sum_{j=1}^{n} T_{Avg}(n-j) + \sum_{j=1}^{n} T_{Avg}(j-1)\right)$$

$$= cn + \frac{2}{n}\left(\sum_{j=0}^{n-1} T(j)\right)$$

~~Best case: nlogn~~

base: $n=0, n=1$

$$T(n) = O(n) + \frac{2}{n}\sum_{j=0}^{n-1} T_{Avg}(j)$$

prove that $T(n) \le cn\log n$

$T(j) = \text{const.}$

and $T(1) = \text{const.}$

$$T(n-1) = O(n-1) + \frac{2}{n}\sum_{j=2}^{n-1} j\log j$$

$$\approx \int_{x=2}^{n-1} n\log n\, dn$$

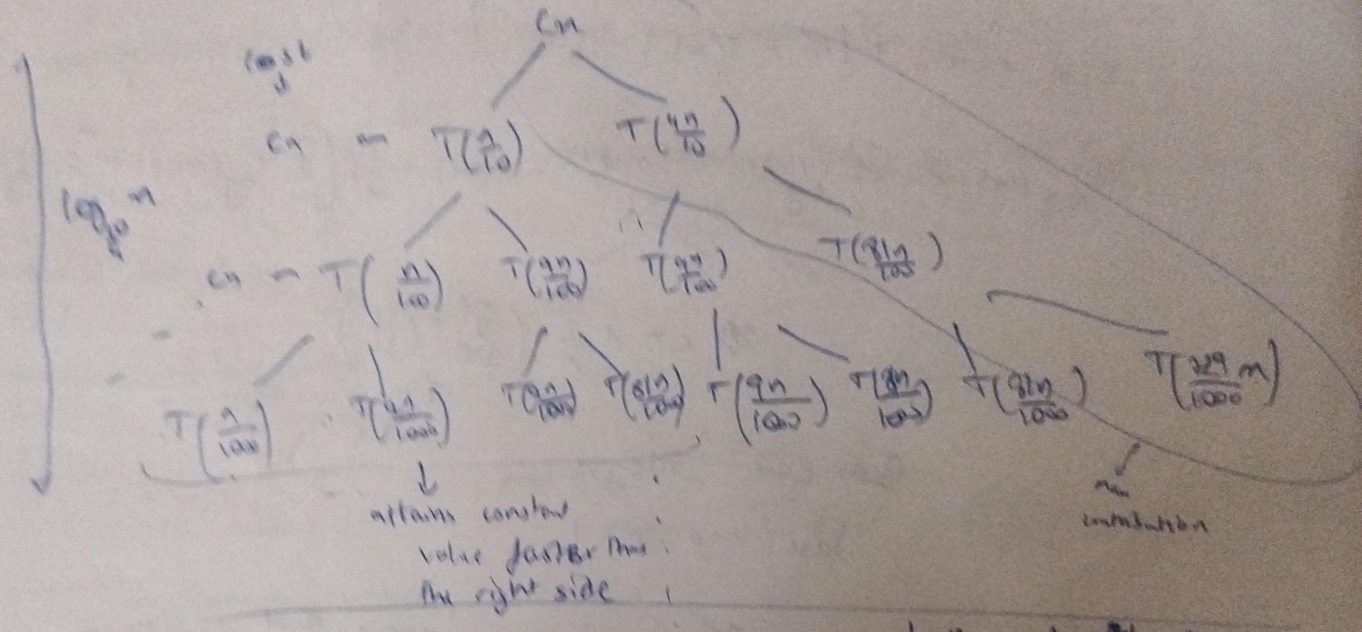
$\int n\log n\, dn = \int \frac{1}{2}\log x\, dx$

$n^2\log n - \frac{n^2}{4}$ | $_2^{n-1}$

eg $T(n) \leq O(n) + T(n - \frac{n}{2}) + T(\frac{n}{2})$.

If $n = 10$

$T(n) \leq T(\frac{9n}{10}) + T(\frac{n}{10}) + O(n)$



$T(n) = cn \log_{10/9} n = c \cdot n^2 \log_2 n$

$T(n) = n \log n$

* Master Method

- If $T(n) = aT(\frac{n}{b}) + f(n)$

  $T(n)$ has the asymptotic bounds:

(i) If $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$,

$T(n) = \Theta(n^{\log_b a})$

(ii) If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \log n)$

(iii) If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$,

and if $af(\frac{n}{b}) \leq cf(n)$ for some constant $c > 0$,

and for all sufficiently large n then $T(n) = \Theta(f(n))$

Ans (i) $T(n) = 9T\left(\frac{n}{3}\right) + n$

$a = 9 \quad b = 3$

$f(n) = n$

$f(n) = n = O(n^{\log_3^9 - \varepsilon})$

$= O(n^{2-\varepsilon})$ ✓

for $\varepsilon = 1$

$\therefore T(n) = \theta(\log n^{\log_3^9}) = O(n^2)$

(ii) $T(n) = T\left(\frac{2n}{3}\right) + 1$

$a = 1 \quad b = 3/2 \quad f(n) = 1$

$\log_b^a = \log_{3/2}^1 = 0$

$f(n) = 1 = \theta(\log n^{\log_b^a})$

$\therefore T(n) = \theta(\log n)$

(iii) $T(n) = 3T\left(\frac{n}{4}\right) + n\log n$

$a = 3 \quad b = 4 \quad f(n) = n\log n$

$n\log n = f(n) \geq \Omega(n^{\log_4^3 + \varepsilon})$ ✓

$af\left(\frac{n}{b}\right) \leq cf(n)$

$3f\left(\frac{n}{4}\right) = \frac{3n}{4}\left(\log \frac{n}{4}\right) \leq cn\log n$

(iv) $T(n) = 2T(\frac{n}{2}) + n\lg n$

$a_2^2 = 1$

$n \lg n = f(n)$

Note:

(i) if $f(n) = \Theta(g(n)) \Rightarrow$
$g(n) = \Theta(f(n))$

$f(n) = O(g(n)) \Rightarrow g(n) = \Omega(f(n))$

(ii) if $f(n) = O(g(n))$ and
$g(n) = O(h(n))$
$\Rightarrow f(n) = O(h(n))$
same with Θ and Ω also

(iii) ~~$f(n) = \Omega(f(n))$~~
$f(n) = \Theta(f(n))$
same with Θ and Ω

* $o(g(n)) = \{f(n) \mid$ for any constant c (>0) ~~$f(n) < cg(n)$~~ $\exists n_0 > 0$, such that $0 \le f(n) < cg(n) \quad \forall n \ge n_0\}$

(small o)

Ex: (i) $2n^2 \overset{?}{=} O(n^2)$ ✓ (ii) $n = o(n^2)$

$2n^2 \overset{?}{=} o(n^2)$ ×

(small)

- another way to define: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

* $\omega(g(n)) = \{f(n) \mid$ for any constant $c > 0$, $\exists n_0 > 0$ such that $0 \le cg(n) < f(n) \quad \forall n > n_0\}$

* ~~$\omega(g(n)) = \{f(n) \mid$ for any const. $c > 0$, $\exists n_0$ s.t. $0 \le f(n) \le cg(n) \quad \forall n \ge n_0\}$~~

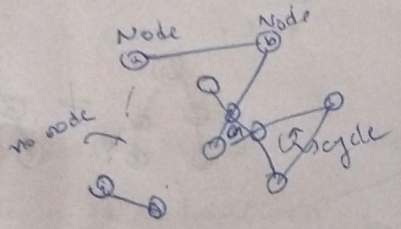
(small theta)

Reading: 25/25 Thee ree

- ~~Data structures~~ are for: organizing information in structured way

Post Midsems

* Graphs:
  - Relations b/w elements of sets in visual way



Graphs:

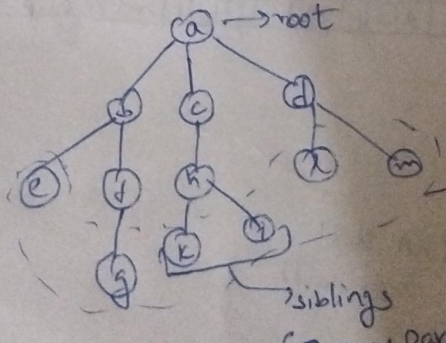
V → set of nodes

E → set of edges

$E \subseteq V \times V$

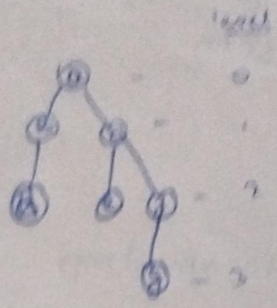
- can have constraints (like noCycle, connected, etc.)
  - Tree

- Tree:



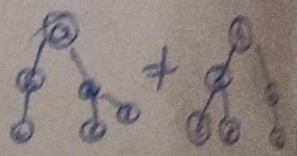
b c d f h → internal nodes (not root nor leaf)

- Binary Tree:
  - two children per parent at most



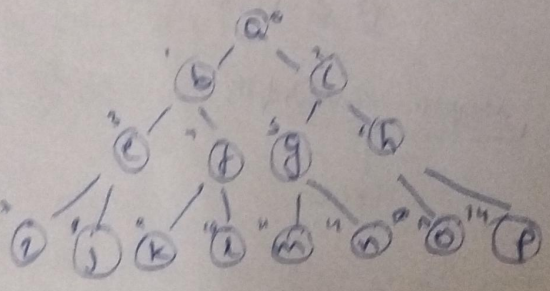
height = 3

- note that we cannot say branch to the left or right, it will then be a different tree.

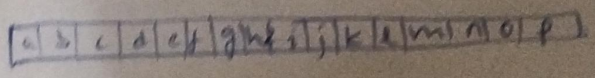


- Complete Tree: every internal node has two children.

Ex:



representing in array:



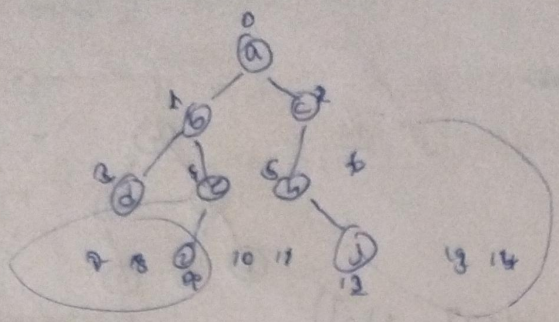
say n = internal

left(n) = 2n+1 (left child)

right(n) = 2n+2 (right child)

$parent(n) = \left\lfloor \frac{n-1}{2} \right\rfloor$

Eg: for non-complete



too many empty spaces, ~~waste of~~ inefficient to use array same way.

- Almost complete tree:

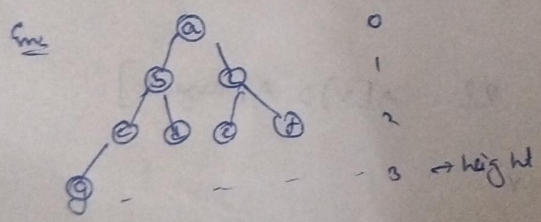
  If height = h,

  upto h-2, all nodes ~~at~~ have two children

  at level h-1, children are added from the left

  (can use array to represent)

Eg:



Note:

Satellite data:

extra data associated with an element which is to be sorted

Eg: name and DOB associated with rollno.

stable sorting:

If algorithm can keep relative positions of elements with same key values (different satellite data)

P.T.O

* Heap:

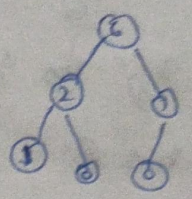
↳ Almost complete tree.

↳ value of parent ≥ values of its children

A[parent(i)] ≥ A[i]

(Max Heat)

Eg



(Min Heap: A[parent(i)] ≤ A[i])

~~81~~ 5 ~~23, 17, 14, 13, 10, 1, 5, 7, 12~~

- MaxHeapify (A, i)

```
l = Left(i);
r = Right(i)
if(l< heapsize  &&  A[l] > A[i])
        largest = l
else
        largest = i
if (r< heapsize  &&  A[r] > A[largest])
        largest = r
if (largest ≠ i)
        swap(A[i], A[largest])
        MaxHeapify(A, largest);
```

complexity:

recursion:

$T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1)$

using master's theorem

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

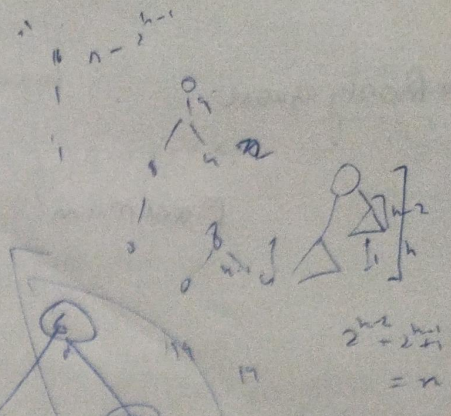
- Call heapify from bottom non-leaf nodes and go up layer by layer

- Build Heap(A)

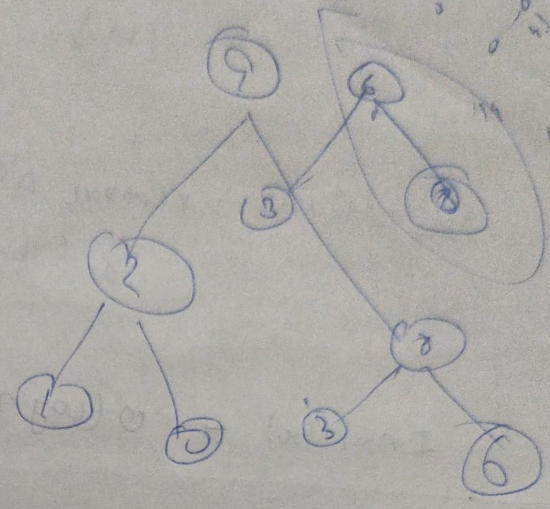
heapsize = A.size

for $i = \lfloor \frac{heapsize}{2} \rfloor - 1$ to 0

MaxHeapify(A, i)



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



- HeapSort(A)

BuildHeap(A);

~~swap (A[0], A[last])~~

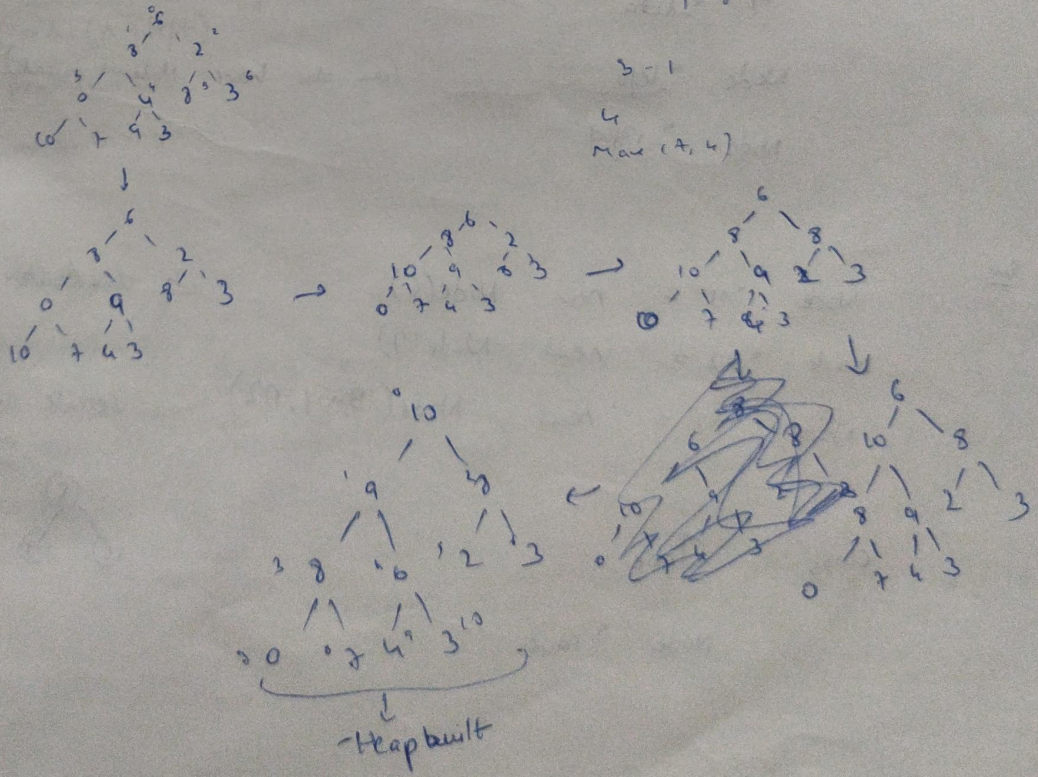
for i = A.Size - 1 to 1

swap (A[0], A[i])

heapsize--

MaxHeapify(A, 0)

Ex 6, 8, 2, 0, 4, 8, 3, 10, 7, 9, 3



n = heapsize

3 - 1

4

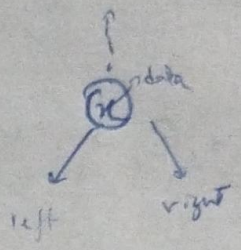
Max (A, 4)

- Heap built

Insert(x) - ~~O(log n)~~ O(n)

(add at end and move all values)

* General Binary Tree



- class Node

  T *_data

  Node *_left (can also have Node *_parent)

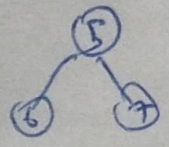
  Node *_right

Ex:

Node *n1 = new Node(6) //create constructor for 6 this

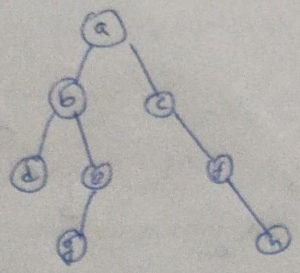
Node *n2 = new Node(7)

Node *n3 = new Node(5, n1, n2) //create another (constructor to make 5 parent



class Tree

Node *root



(i) Preorder Traversal

a → b → d → e → g → c → f → h

root

(go left first then right)

```
preorder(Node *n)
    print n→data
    preorder(n→left)
    preorder(n→right)
```

check for NULL and return (base case)

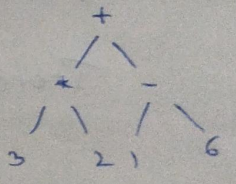
(ii) Inorder traversal

(go left and then print)

d → b → g → e → a → c → f → h

```
inorder(Node *n)          (if n==NULL return)
    inorder(n→left)
    print n→data
    inorder(n→right)
```

Eg:



inorder traversal:
(3*2) + (1-6)

(iii) Postorder Traversal

~~postorder(Node *n)~~

~~postorder~~

d → g → e → b → h → f → c → a

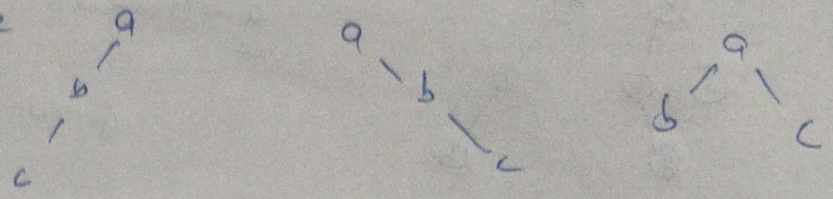
```
postorder(Node *n)
    post(n→left)
    post(n→right)
    print(n→data)
```

Eg:

3 2 * 1 6 - +

Eg: If preorder trav = a b c

can be



more than one tree.

∴ can't construct one tree from an order

- Two ~~tree~~ necessary for constructing unique tree

  (i) Preorder and Inorder

  (ii) Post and In

Note:
Pre and Post
won't work for all trees

Eg:

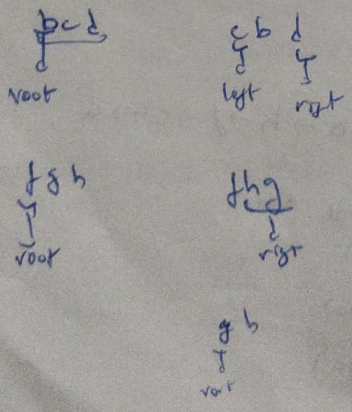
preorder: a bcd efgh

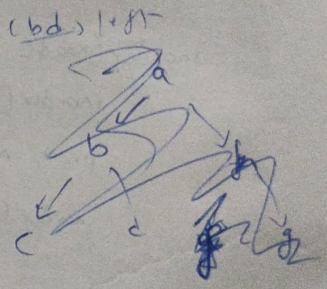
Inorder: cbd a hg

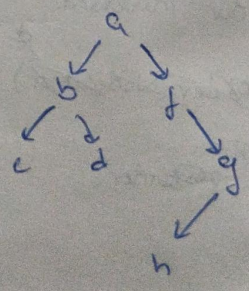
root

left of root

right of root







Trees

$n \rightarrow$ nodes

$e \rightarrow$ edges

$e = n - 1$

Proof:

(removing an edge from a tree will result in two trees)

(gonna use this)

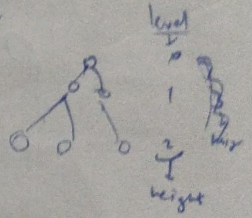
~~Proof~~ state#

Trivial for $n = m$

take $m+1$ and remove one edge

Back to rooted binary tree

- The maximum number of nodes at level i is 2^i



- The maximum number of nodes in a tree of height k is $2^{k+1} - 1$
- For any non-empty binary tree T, if number of leaf nodes is n_0 and n_2 be the number of nodes of degree 2 (2 children) then $n_0 = n_2 + 1$

Proof: total number of nodes $n = n_0 + n_1 + n_2$

total no. of edges $= 2 \cdot n_2 + n_1$

$= n - 1$

$= n_0 + n_1 + n_2 - 1$

$\Rightarrow n_0 = n_2 + 1$

* Time complexity of build heap

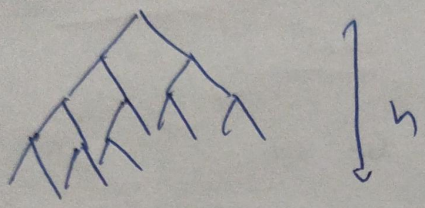
Manheapify: $O(\log(n))$

↓

called on $O(n)$ nodes ~~Each node~~ ⇒ $O(n \log n)$

*not tight bound

bedtw:



No. of times Manheapify called:

$$\sum_{k=1}^{h} k \cdot (\text{no. of trees of height } k)$$

$$= \sum_{k=1}^{h} k \cdot 2^{h-k}$$

$$= 2^h \sum_{k=1}^{h} \frac{k}{2^k} \leq 2^{h+1}$$

∴ Time complexity of Buildheap = $O(2^{h+1})$

$= O(2n)$

$= O(n)$

- Heapsort:

  Buildheap → $O(n)$

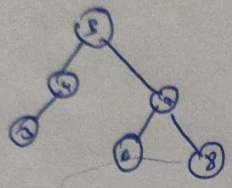
  Manheapify is run on top of heap → $O(n \log n)$

∴ $O(n \log n)$

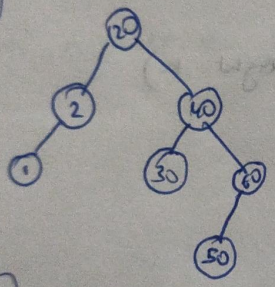
* Binary search Tree:

- ng be empty
- Nodes store data and associated key
- key values are distinct
- key values ~~on~~ in left subtree of a node are smaller than the node value.

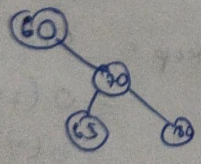
Ex: (i)



(ii)



(iii)



* Search:

check root first, go right if greater or lyt if smaller

~~Time to~~

Time complexity: O(h) → height

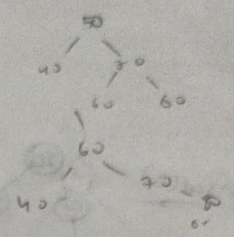
* Insert(n)

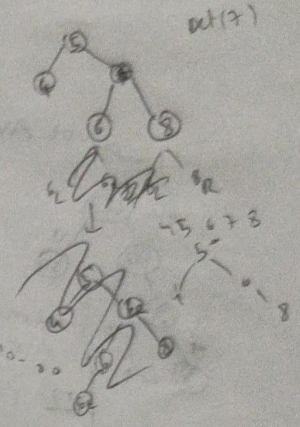
search for 'n', insert at place where we get NULL value.

time complexity: O(h) (also ensures no duplicate)

* Sorted sequence:

Inorder traversal

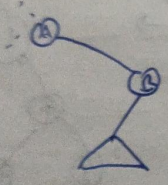




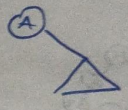
* Deleting:

(i) If Leaf Node, set it to NULL

(ii) Only one child

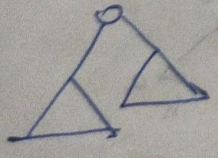


remove 'B' and set the child as child of 'A' →



(iii) Has children on both sides:

swap ~~entries~~ with greatest of left subtree and delete that node



BF

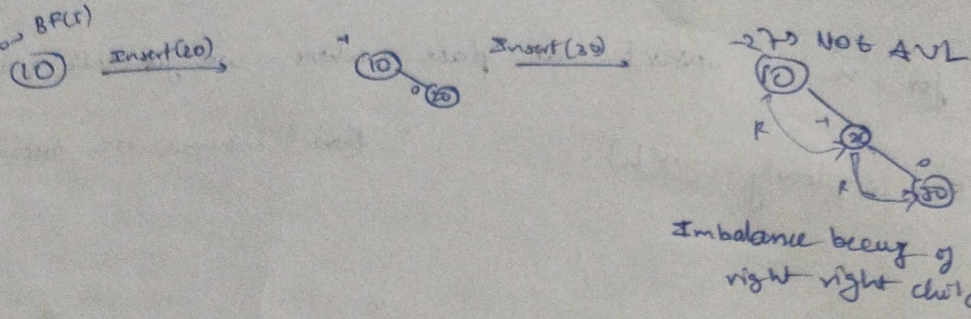
Balance Factor: difference between height of left and right subtrees

$$BF(T) \Rightarrow h_L - h_R$$

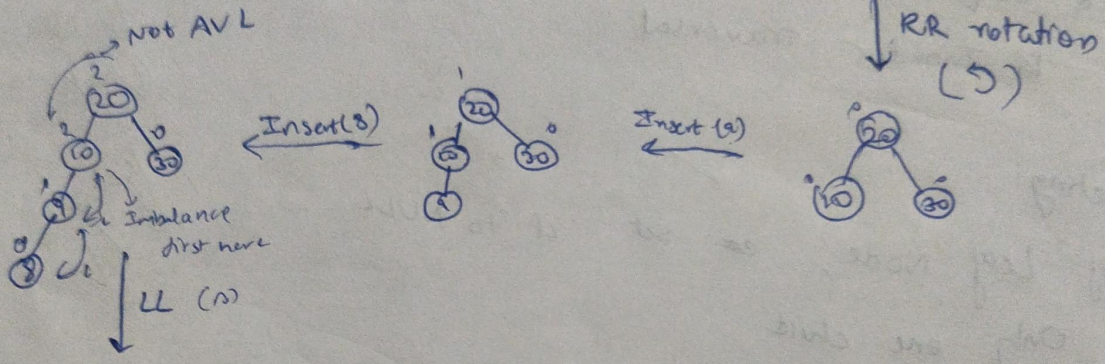
T: Node; h_L: height of left subtree; h_R: height of right subtree

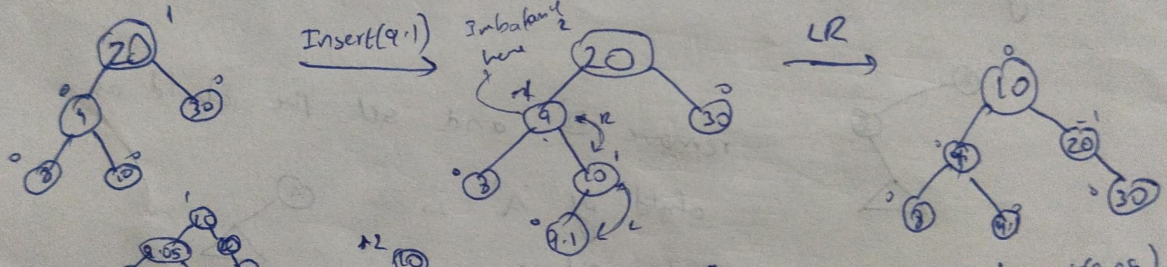
* AVL Tree: BF(T) = -1 or 0 or 1 for any node T.

Ex: → BF(r)

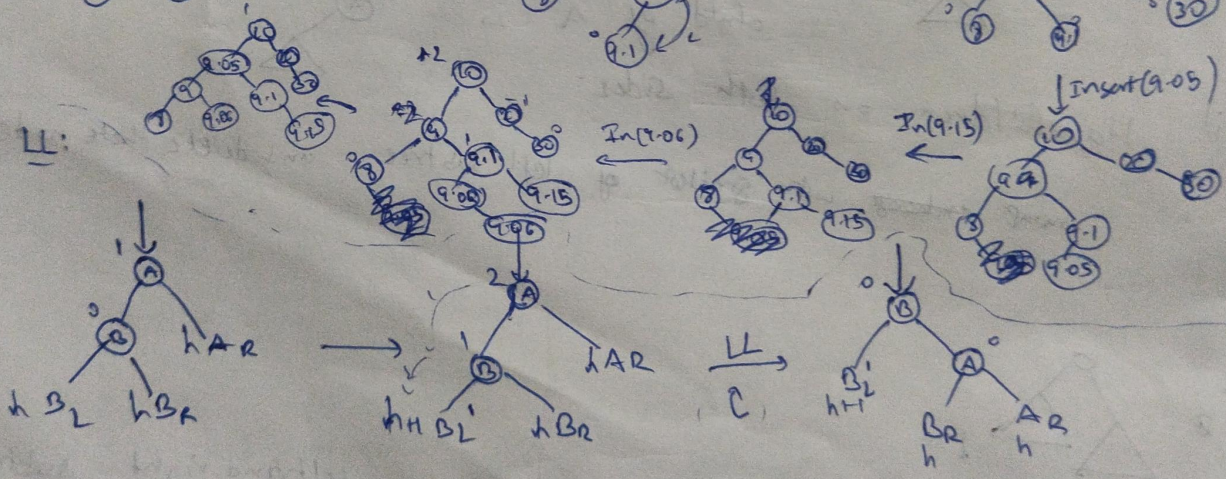


RR rotation (5)

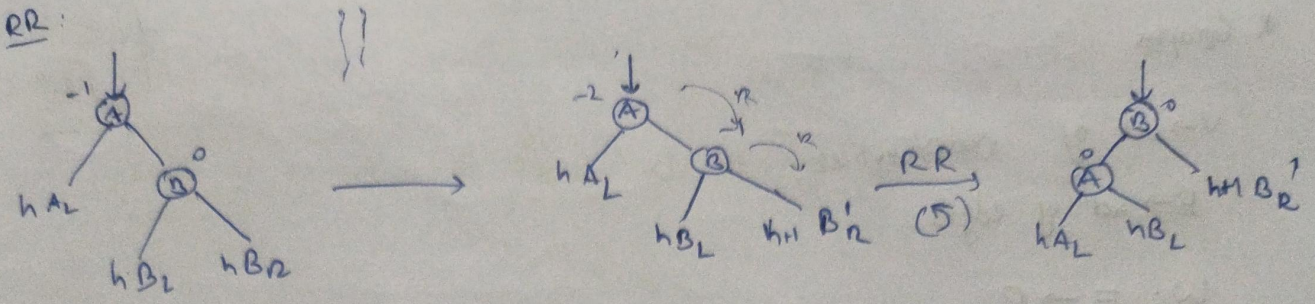




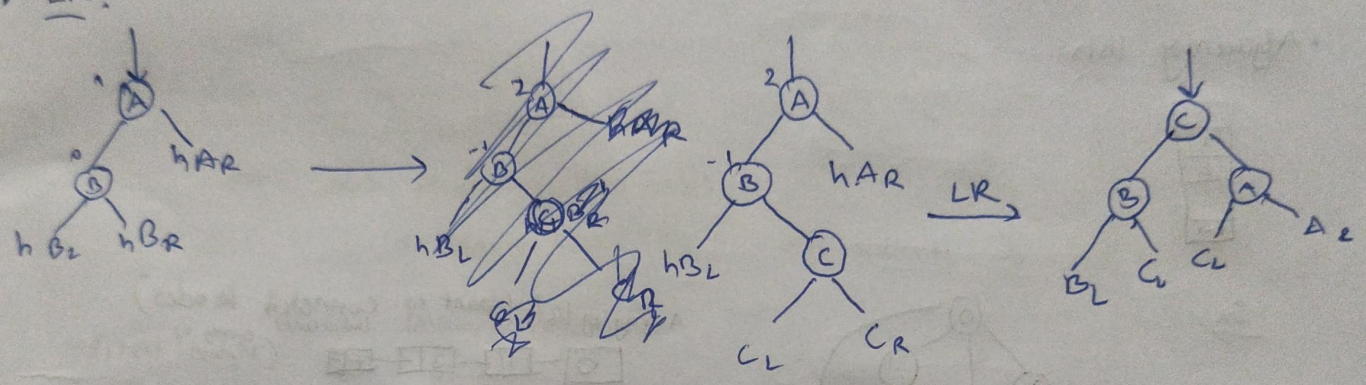
- LL:



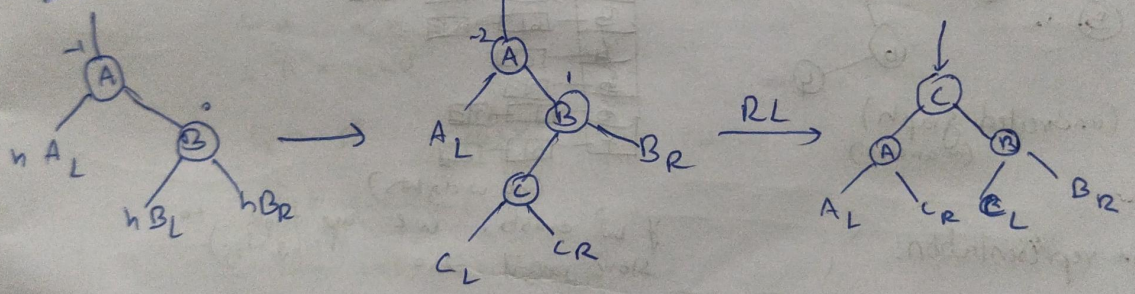
- **RR:**

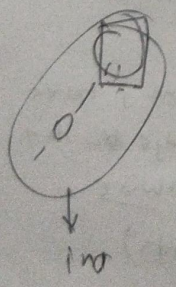


- **LR:**



- **RL:**





* Graphs

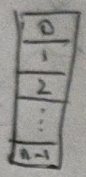
V → set of nodes/vertices

E → set of edges

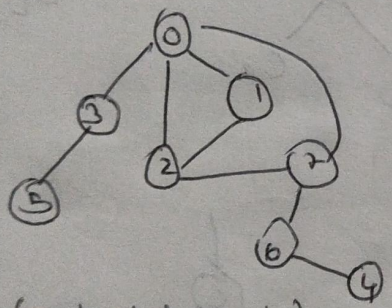
$W : E \to \mathbb{R}$

(weight, (edges), (real nos.))

* Adjacency lists:



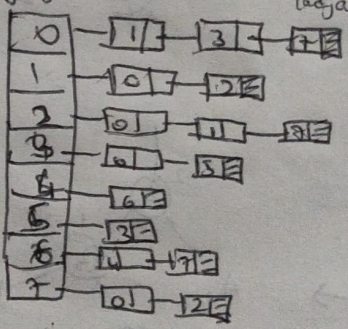
Eg:1



(undirected graph)
(no arrows)

Array of lists (point to connected (adjacent) nodes)

(array of linked lists)



(no weights)

If 'w' exists, we use tuple to store weight also i: (j, w_{ij})

* Adjacency matrix representation:

M

	0	1	2	3	4	5	6	7
0	0	1	0	1	0	0	0	1
1	1	0	1	0	0	0	0	0
2	1	1	0	0	0	0	0	0
3	1	0	0	0	0	1	0	0
4	0	0	0	0	0	0	1	0
5	0	0	0	1	0	0	0	0
6	0	0	0	0	1	0	0	1
7	1	0	1	0	0	0	0	0

$M_{ij} = 1 \Rightarrow$ i and j have edge b/w them

0 otherwise

(if no weights)

If 'w' exists (weight func^n), $M_{ij} = w_{ij}$ (weight)

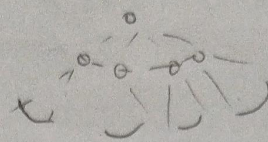
- BFS

(Breadth First Search)

vector<> array

array[i]

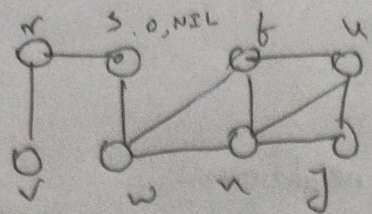
look at a node and then all its neighbours (print/mark)

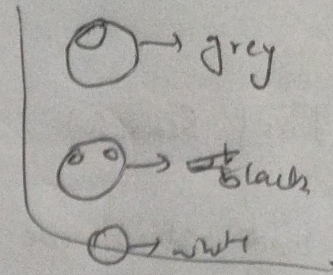


```
BFS ( G (graph) , s (source node) )
    for each vertex u ∈ G - {s}:                                        cost ↓
            u.color = WHITE          // to indicate its not visited yet   ] → θ(V)
            u.d = ∞  (distance from source)
            u.π = null  (parent)
    s.color = GREY      // to indicate root is visited      → 1
    s.d = 0                                                 → 1
    s.π = null                                              → 1
    Q = ∅    // creating empty queue                        → 1
    Q.Enqueue(s)                                            → 1
    while not Q.isEmpty(){
        u = Q.Dequeue()
        for each 'v' Adjacent to 'u'{
                if v.colour = WHITE
                    v.colour = GREY                          → Σ dg(u)
                    v.d = u.d + 1                              (no. of edges connected)
                    v.π = u                                  = θ(2E)
                    Queue.Enqueue(v);
        }
        u.color = Black
```

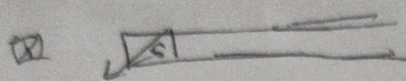
∴ Complexity = θ(V+E)

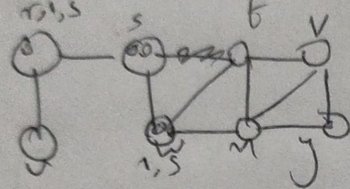
Ex.

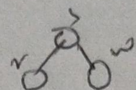


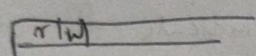


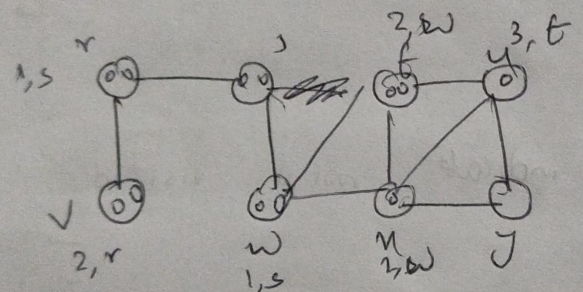
s.d=0
s.π =NIL

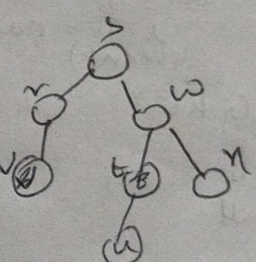


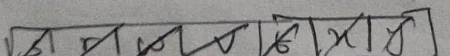










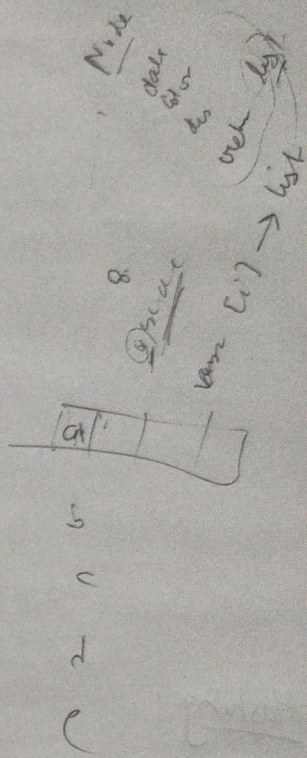


vector<int> state

state[i]=0 = white

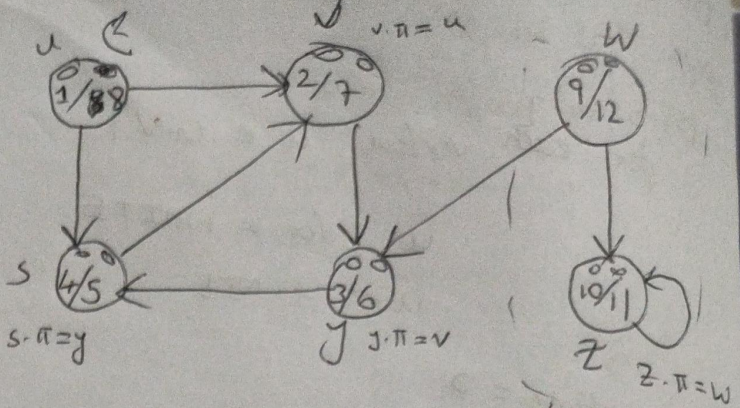
* DFS
(Depth First Search)

```
DFS(G){
  for each vertex u ∈ G.V{
        u.color = WHITE
        u.π = NIL
  }
  time = 0
  for each vertex u ∈ G.V{
        if u.color == WHITE
              DFSVisit(G, u)
  }
}
DFSVisit(G, u){
      time = time + 1
      u.d = time
      u.color = GRAY
      for each v ∈ G.Adj(u){
            if v.color = WHITE
                  v.π = u
                  DFSVisit(G, v)
      }
      u.color = BLACK
      time = time + 1
      u.f = time
}
```

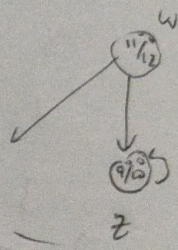


Eg.

Ex 2.



here we went to 'w' after we done with 'u'
if we went to 'z'.



*Hashing:

Set U
(universe set)

Object → key —d→ {0, ..., n-1} (→ I)

d → converts the key to an integer

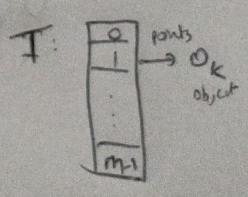
(indexing) (→ In attempt to store the data as an array)

Ex if Object → student
key → roll no.

size of universe may not be same as size of 'I'

If they are same → direct addressing

$|U| = n \quad |I| = n \quad U \to I$



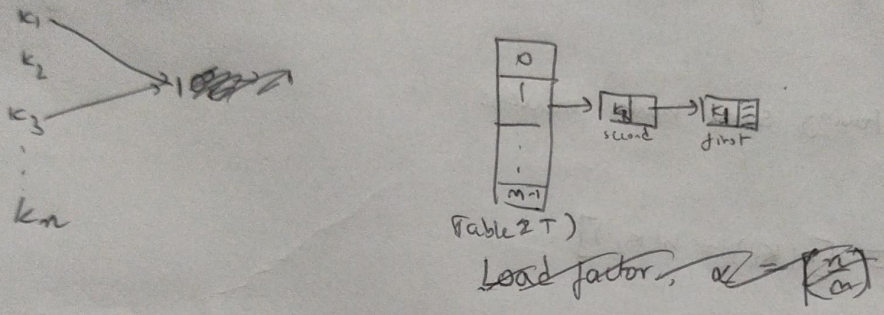
- Direct addressing not very good if universe is large and only small amount needs to be stored.

- If $n = m$, f be a one-one function

  If $n > m$, f will be many-one → collision

- Collision Handling

  Index points to list of objects instead of just object

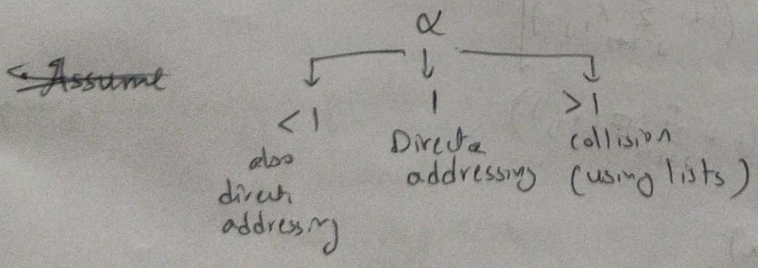


- Load factor 'α' for table 'T' is defined as $\frac{n}{m}$.

  'n' elements in the universe

  'm' no. of slots in table

  Avg. no. of elements stored in a chain



* Assume Simple uniform Hashing (SUH): Any given element is equally likely to hash to any of the m slots independantly of where other elements has hashed.

Theorem:

- In a hash table where collisions are handled by chaining an unsuccessful search takes average case time: $\Theta(1+\alpha)$ under the assumption of ~~SHA~~ SUH.

Find slot | Scan list

Theorem:

- In a hash table where collision is handled by chaining, a successful search takes average case time: $\Theta(1+\alpha)$ under the assumption of SUH.

Find slot

To find chance of having same hash values:

Indicator RV

$$X_{ij} = I\left[h(k_i) = h(k_j)\right]$$

$$\text{Prob}\{h(k_i) = h(k_j)\} = \frac{1}{m}$$

$$\therefore E(X_{ij}) = \frac{1}{m}$$

$1 + \sum_{j=2}^{n} X_{ij}$ &

$1 + \sum_{j=3}^{n} X_{2j}$

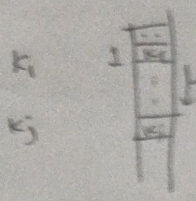
$$\therefore E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} X_{ij}\right)\right]$$

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} E(X_{ij})\right)$$

$$= \Theta(1+\alpha)$$

*Open Addressing:

No chaining pointers. keys stored in the Table itself.



for chaining $h: U \rightarrow \{0, \ldots m-1\}$

("hash function")

for open add. ~~$h(k,0)$~~ $h(k,i)$

~~check for empty~~ gives an index value

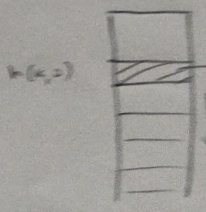
if not empty

call $h(k, i+1)$

$\underbrace{\langle h(k,0),\ h(k,1) \ldots h(k,m-1)\rangle}_{\text{can be called "m" times}} \rightarrow$ permutation symbols

- Linear Probing

$\langle h(k,0),\ h(k,0)+1, \ldots \rangle$



primary clustering problem:

all elements with same key values will cluster together pushing others away

- Quadratic Probing:

$\langle h(k,0),\ h(k,0)+c_1+c_2,\ h(k,0)+c_1 2 + c_2 4, \ldots \rangle$

chose c_1, c_2

clustered at a different place (avoiding clusters)