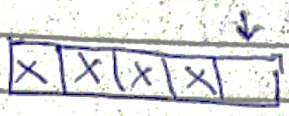
DATA STRUCTURE

Array



- Static
- Dynamic

ptr, the array pointer

cap: physical size of the array
↓
number of cells that have been allocated

// size: logical size
↓
number of cells that are currently being used

~~Class~~

GROW

```
template<typename T>
class vector{

    T *aptr;
    int cap, size;

Public:

  vector(){
    cap=size=0;
    aptr=nullptr;
  }

void push_back(const T &obj){

  if(size<cap){
    aptr[size++]=obj;

  }else if(size==0){
    size=cap=1
```

```
        aptr = new T[size];
        aptr[size] = obj;
    }
    else if (size == cap){
        // grow the array
        T *temp = new T[2 * cap];
        for(int i=0; i<cap; i++)
        {
            temp[i] = aptr[i];
        }
        temp[size++] = obj;
        cap *= 2;
        delete [] aptr;
        aptr = temp;
    }
}
```

#include <vector> → function have push_back

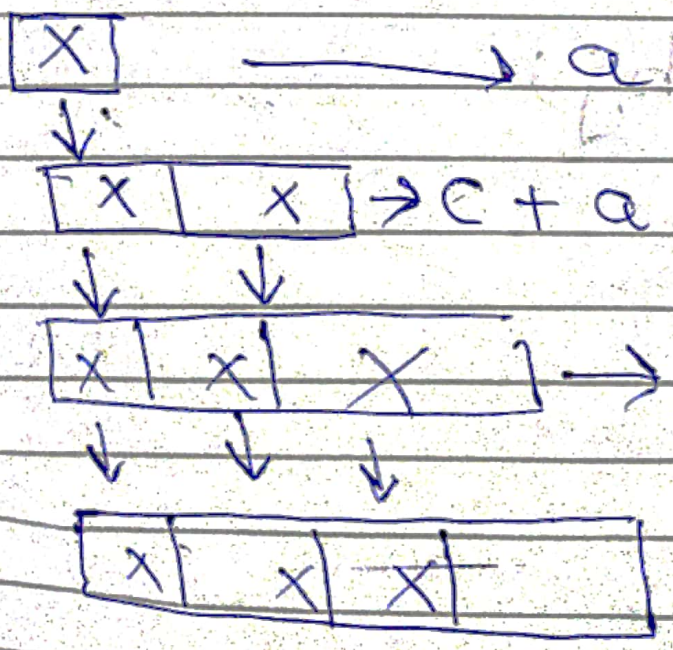
new method try for heap question

Analysis: When we grow the vector by 1 slot

c: cost of copying one element from one array to another

a: cost of writing a new element into an array

Scenario:
We have n consecutive push_back operations.



(n - c) + a

$$T(n) = a + (c+a) + (2c+a) + \cdots (n-1)c + a)$$

$$= an + [c(1+2+3+\cdots(n-1))]$$

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

$$= an + \frac{cn^2}{2} - \frac{cn}{2}$$

~~$= \frac{cn^2}{2} + (a \quad c$~~

$$= \frac{cn^2}{2} + \left(a - \frac{c}{2}\right)n + 0$$

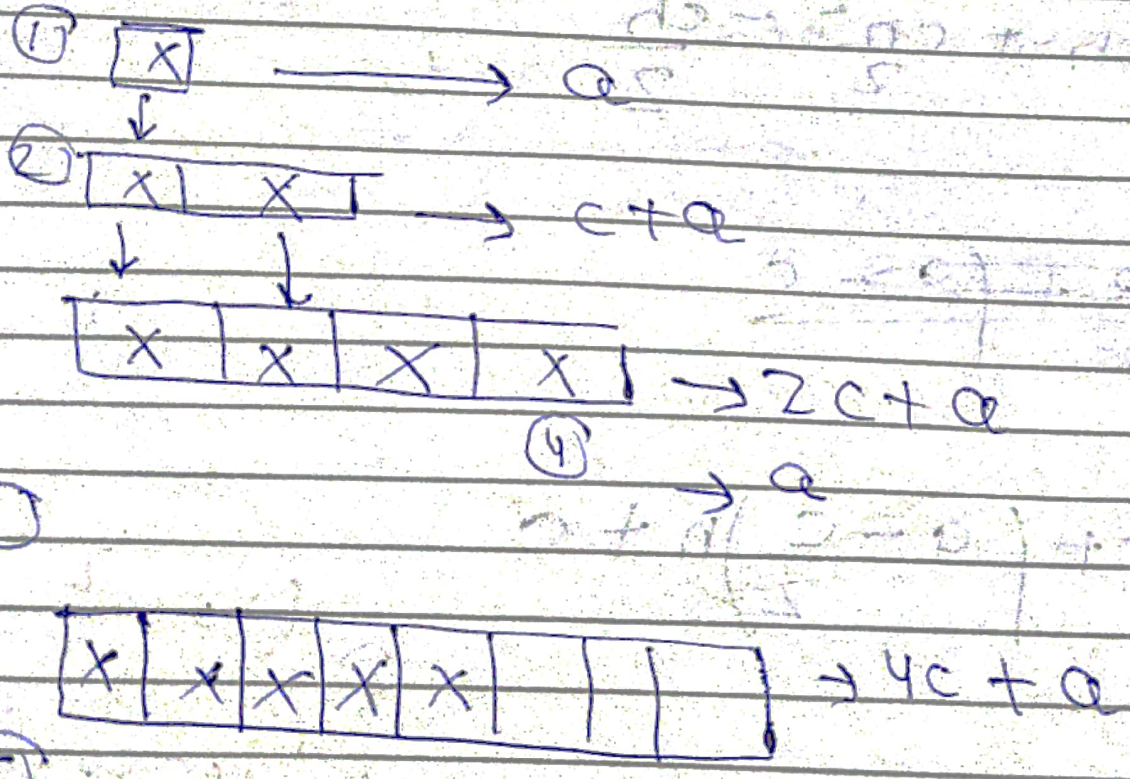
$$= an^2 + bn + c$$

$\therefore \simeq (1000)^2$

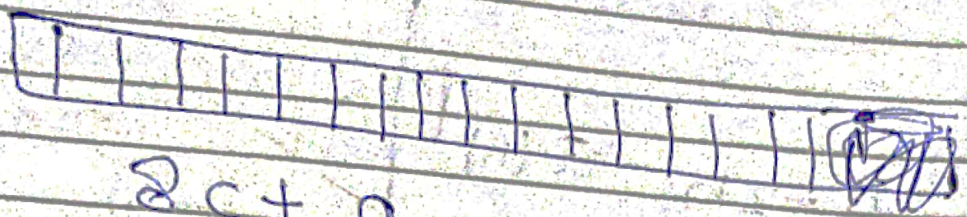
$\simeq (10000000)^2$

an^2

analysis #2 when we grow the array by doubling its size



(a)



$8c + a$

$\rightarrow a$

$\rightarrow a$

$\rightarrow a$

$\vdots$

$\rightarrow 16c + a$

a

a

$\vdots$

$\rightarrow 32c + a$

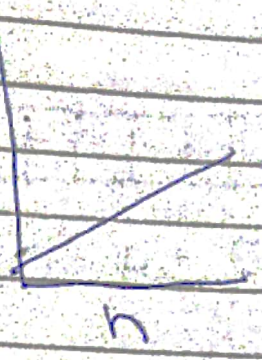
$$T(n) = a + (c+a) + (2c+a) + a + 4c + a + a + a + a + \ldots$$

$$= na + c[1 + 2 + 4 + 8 + 16 + \ldots + \frac{n}{2}]$$

$$= (na - n) + c[1 + 2 + 4 + \ldots + n]$$

$= nq^{-n} + 2n$

$= qn + n$



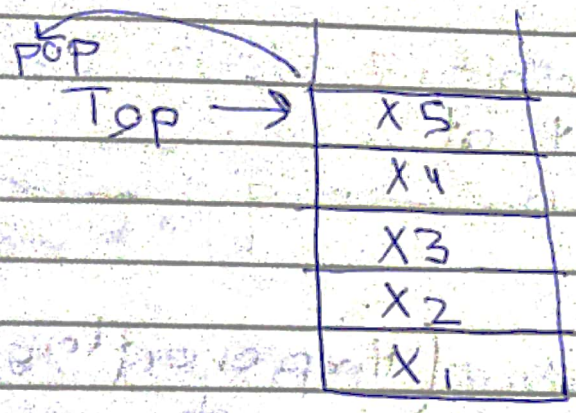
$n + \frac{n}{2} + \frac{n}{4} + \cdots + \leq 2n$

$$S_n = \frac{q(1-q^n)}{1-q^n}$$

Stacks and Queue

Stack

On which order you enter out and in which order it is leaving.



LIFO
Last in first out

retrieve the stack in opposite order.

operations
push
pop
Empty()
peak(نلقي)

empty()

tube
Peek("look a

operations

push: add an element at top of stack
pop: Remove the element from the top

Empty: true if it is empty
false: otherwise

peek: read the element at the top of stack

Dynamic stack (Heap overflow → exception)

Stack applications:

system stack

```
loo A(){

    B();
    loo →
}
```

Assembly calling

200

Reserved data structure → stack

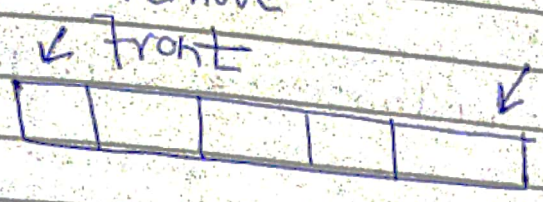
1. undo / Redo
2. Back button on Browser
3. expression evaluation: 3+4+2/1
4. Cards game

~~QUEUE:~~

~~FIFO~~

Queue → Horizontal (just like bus)

FIFO Dequeue



Enqueue: add an element at the rear

Dequeue: Removes an element from the front.

Peek: Read the front element

Empty: returns true if the queue is empty

Size: How many types we greq

empty queue
size = 0

Applications of queue.

i) waiting is required
Network resources
routers on the network

ii) MCDONALDS DRIVE

iv) operating system

~~Source code.~~

top=-1  top+1=size

```
template<typename T>
class Stack{
    T *arr;
    int size,cap;
    int top;
public:
    Stack(int isize=0);
    if(isize>0){

    arr=new T[isize];
    size=0;
    top=-1;

    cap=isize;

}
else{
    arr=nullptr;
    top=-1;
    size=0;
    capacity=0;
```

* Writer
* User of vector already available in stl

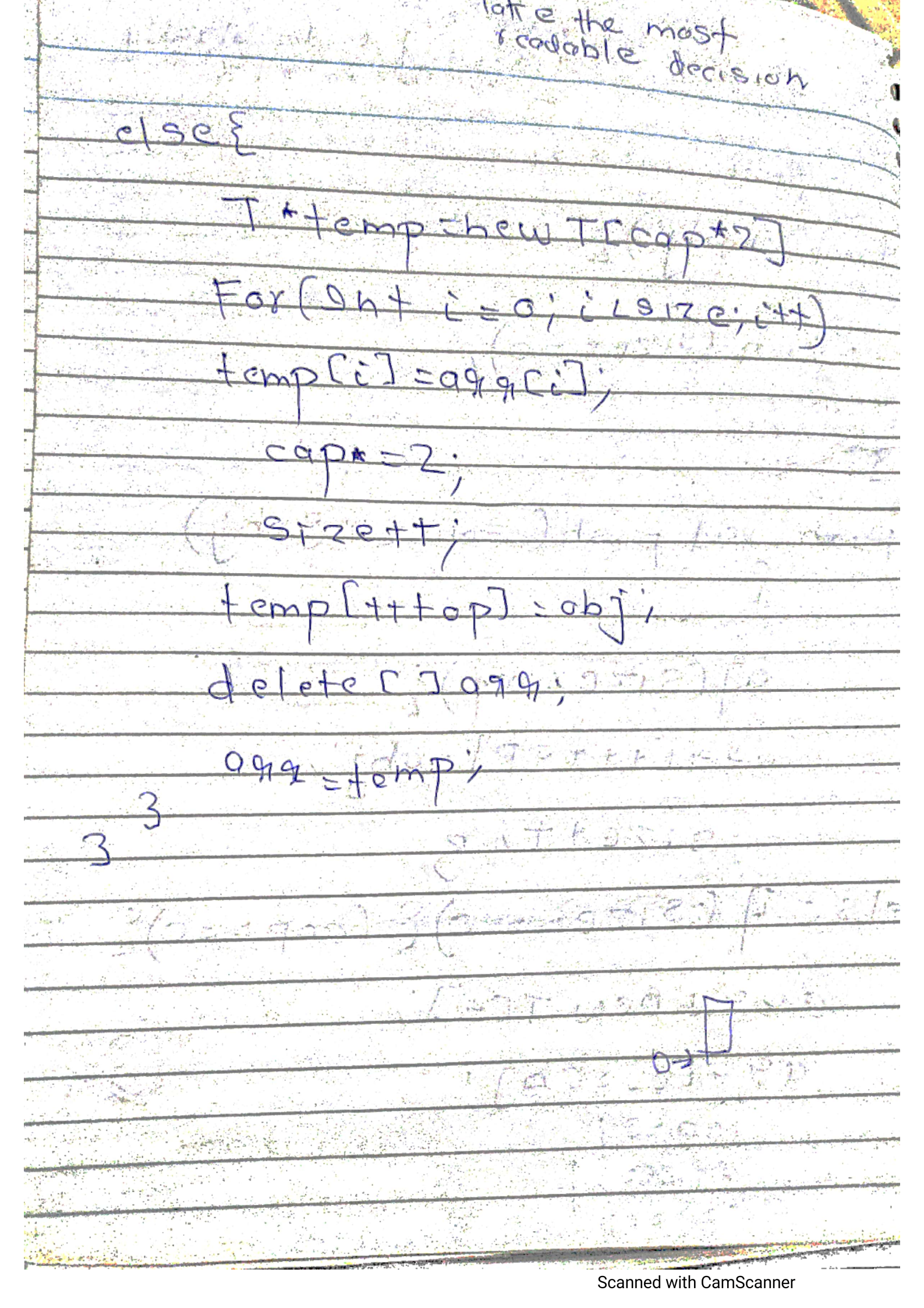
```
Stack<int> s;
bool empty(){
return (size==0);
}
void push(const T& obj)
{
if(size<cap){
arr[++top]=obj;
size++;}
else if(size==0){(cap==0){
arr=new T[1];
arr[0]=obj;
top=0;
size=1;
cap=1;
}
```

late the most
readable decision

```
else{
    T *temp=new T[cap*2]
    For(int i=0;i<size;i++)
    temp[i]=arr[i];
      cap*=2;
      size++;
    temp[++top]=obj;
    delete[] arr;
      arr=temp;
```

3 3

3



| 1 | 2 | 3 | 4 |

Pop:

size=4;

top →

9
6
8
5

size--;

const T &peek()

return a[top]

← top=-1

stack underflow

only pop if stack is empty

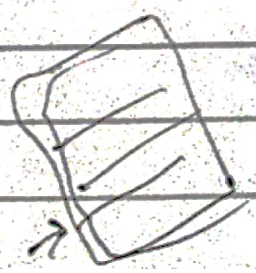
pop

↓

does not return

pop return type void

```
void pop(){
    if(size
    if(empty())
        return
    if(size>cap/2){
        top--;
        size--;
    }
    else{
        T *temp = new T[cap/2];
```



QUEUE

```
T *arr;
cap = 5;+1
size = 0;
```

} index values.

operations::

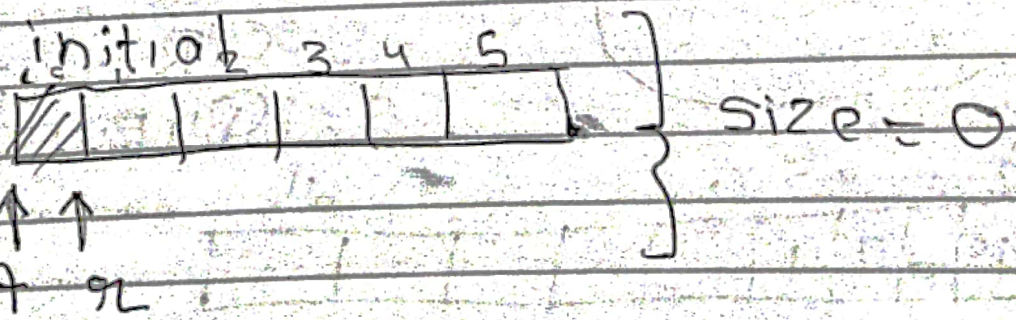
Enqueue :→ Add at rear

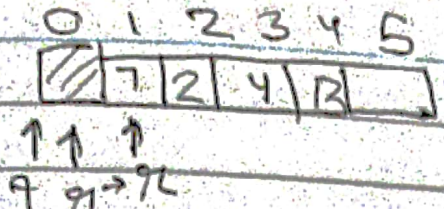
Dequeue → Remove at front.

peek - read from front

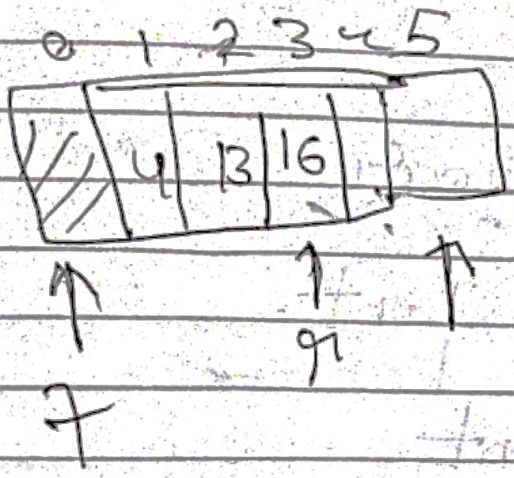
empty() ~~true if empty~~

return(size == 0)



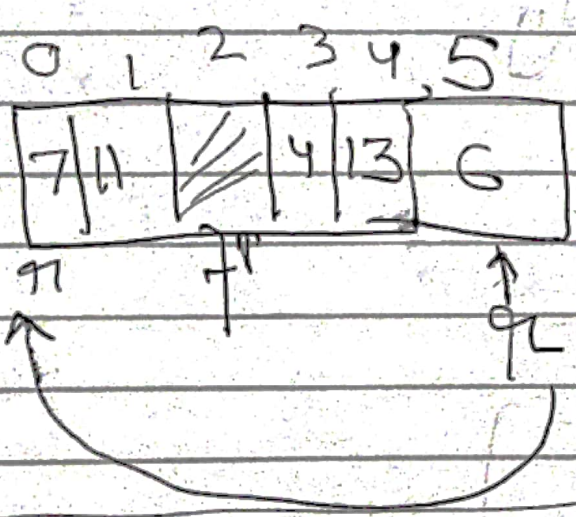


Bad Sol



n steps. Non-shifting circular

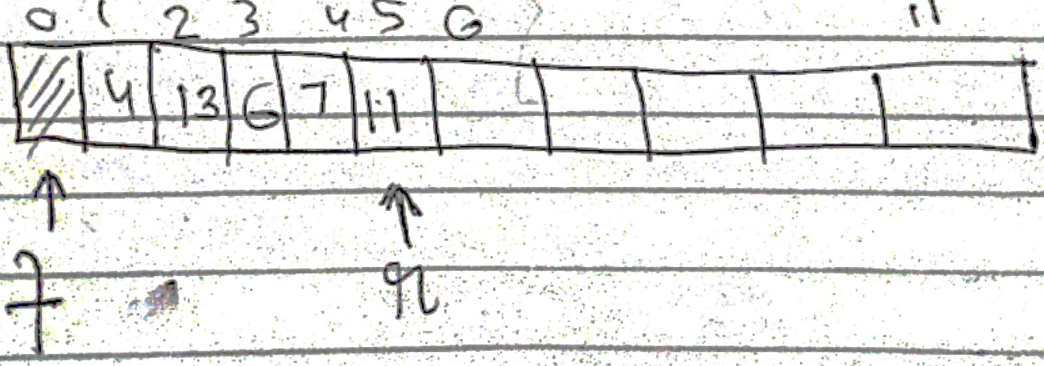
First deq

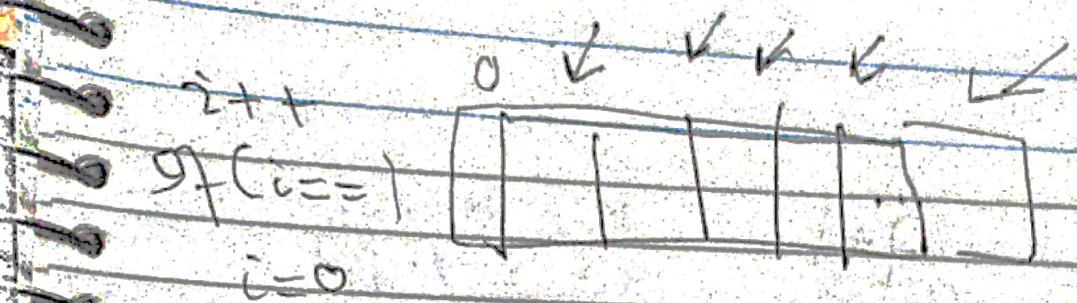


Size=4

Size=5

CIRCULAR QUEUE

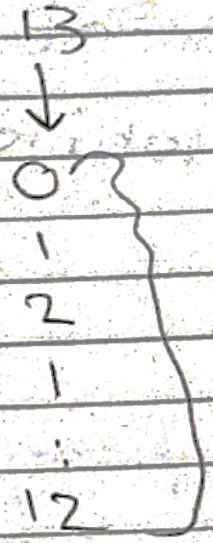




i = (i+1) % 10

0 → 1 → 2 → 3

9 → 0




```
int next(int i){

  return (i+1) % cap;
}
```

i = next(i);

Print Queue

What is wrong?

```
For(int i=0; i<size; i++)
    cout<<arr[i];
```

correct way

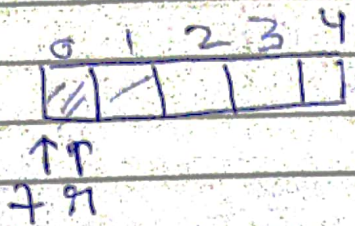
front+1

~~front~~

```
int i= next(front)
int count=0;
while(count<size)
{
    cout<<arr[i];
    i=next(i);
    count++;
}
```

front is not always be zero.

size is number of elements enqueued.



cap = 5
f = 0
r = 0
size = 0

Goal:: To perform performance analysis of own DS

→ How shall we do it?

↳ option: do it for specific tech

↳ analyze the DS not for a specific machine but for a general machine (Modern computer)

How to describe it

Constant
Addage

machine to machine
↳ vary fraction of
seconds.

2 steps
a = a + b

c = a / b
↳ 2 steps:

A[i] are also step

① We wish to count the
no of steps taken by
algo

② → We wish to express this
step count as a function
of size of data

a=a+b | # of steps depends of size of input n

We estimate

$T(n)$ = no of steps taken by an algo on an input of size n

units are steps

$T(n) = 2n^2 + 5$ /

$T(n) = 3n + 5$ /

Efficient DS: To perform eh operation it takes

$T(n)$ steps and $T(n)$ grows slowly with n

and perform well for large n

$10^1, 10^2$

Analysis where we are interested in large n: Asymptotic Analysis

→ step count analysis

→ Asymptotic

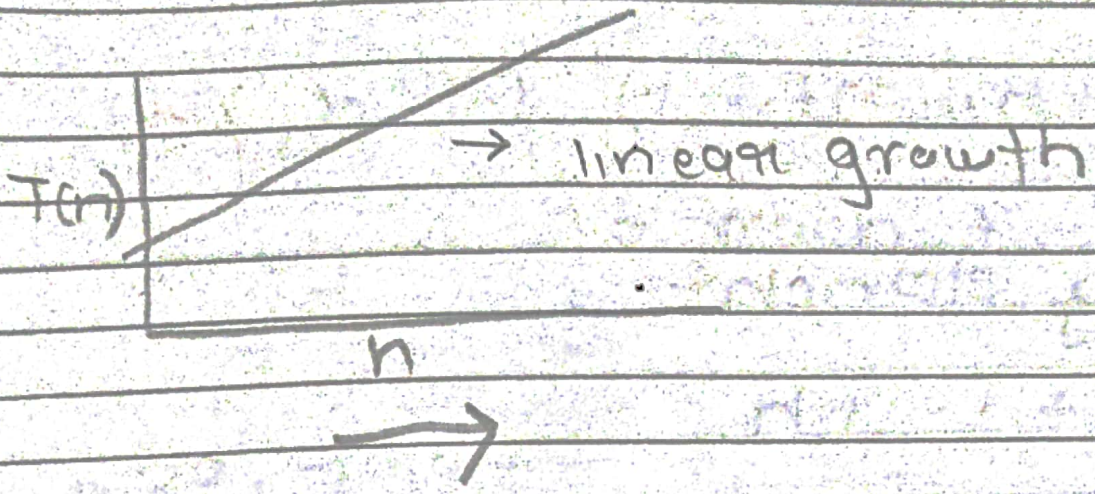
```
int i=0, sum=0;          → c1
For(; i<n; i++)          → n c2
    sum = sum+i          → n c3
```

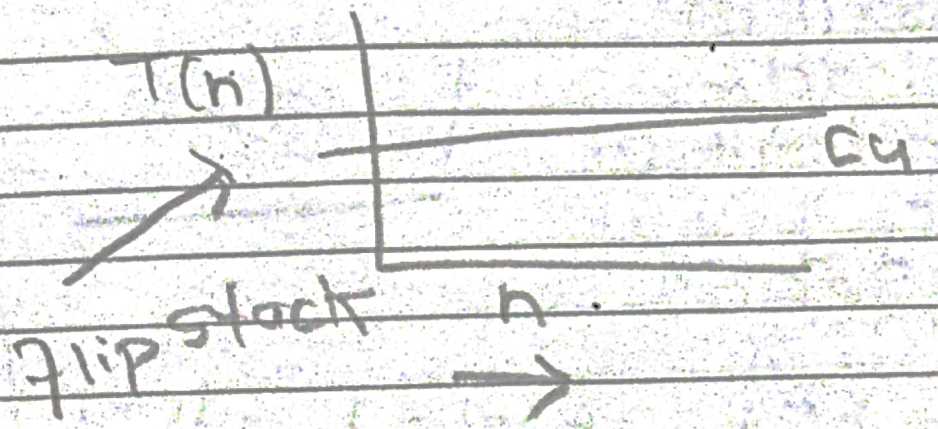
step count

$(c_2 + c_3)n + c_1$

$an + b$

a=2
b=3





→ Linear time
→ constant

Math Review

Simplest form

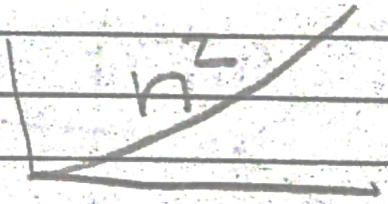
$1+2+\cdots+n = \frac{n(n+1)}{2}$

Traversing element

$1+2+3+\cdots+n$

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

$= an^2+bn+c$



Geometric Series

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

Binary search

$\longleftarrow n \longrightarrow$

2	11	16	25	32

step 1 → n

step 2 → n/2

step 3 n/4

$n = 2^k$

$k = \log \log_2 n$

$\log_2 n$ # of times you divide n by 2 to reach 1

$2^{50}, 2^{100}$

estimated number of H atoms in the 2^{2^0} universe

Fast

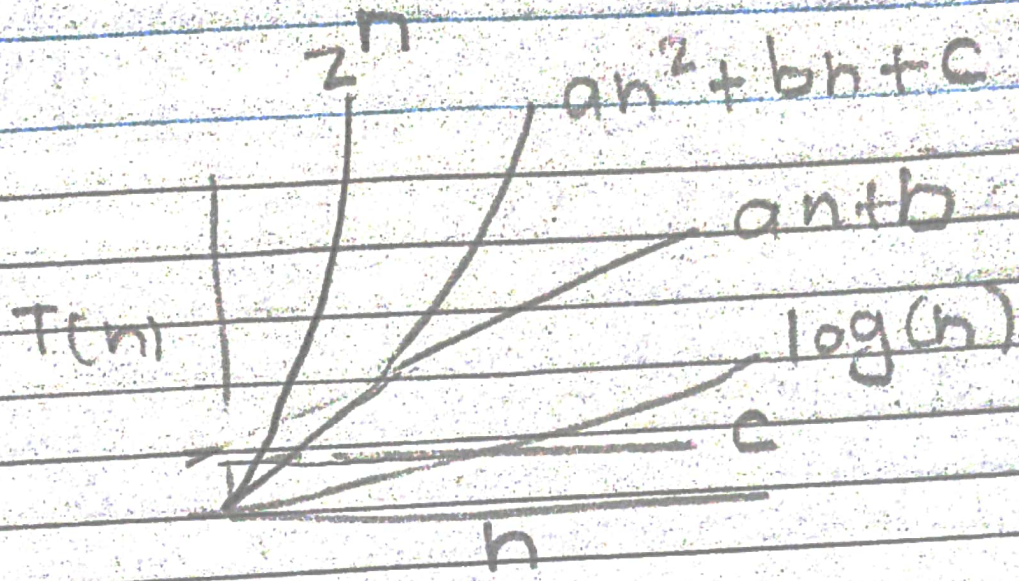
$\log n$

$an + b$

$an^2 + bn + c$

2^n

Slow



int i=1, sum=0; → c1

For(; i<n i=i×2) c2 log n

sum = sum + i;

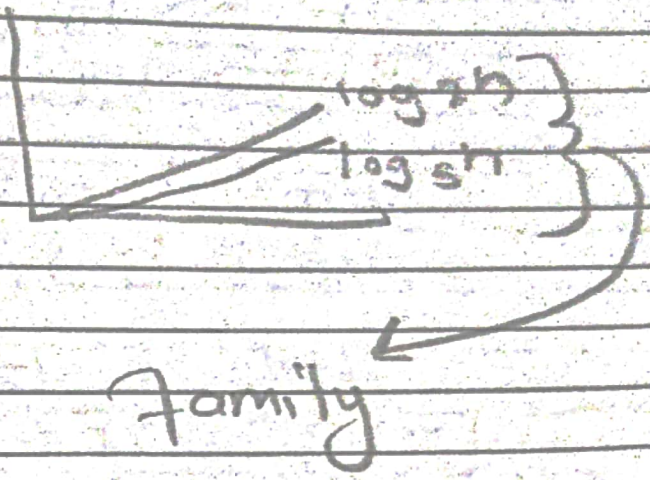
$T(n) = a\log n + b$

1, 2, 4, 8, 16, 32 .... n/8, n/4, n/2, n

1 2 3 1 2 3

3 2 1

$\log_5 n \leq \log_2 n$



Careful analysis
Careless analysis.

Worst-case analysis:

Input example

Farmalies of T(n)

(a class of sols of T(n)

$T(n)$ $\begin{matrix} 2n+3 \\ 4n+5 \end{matrix}$ } lines

same family
when n is large

$T(n) = 4n^2 + 5n + 6$

$T(n) = 2n^2 - 6n + 2$

same family

Math Review

→ Sum of Arithmetic series:

$$S_n = \frac{n}{2}(2a + (n-1)d)$$

$$1 + 2 + \dots + n = \frac{n(n+1)}{2}$$

→ Sum of geometric series:

$$1 + r + r^2 + \dots + r^{n-1} = \frac{1 - r^{n+1}}{1 - r}$$

More generally

$$a + ar + ar^2 + \dots + ar^{n-1} = \frac{a(1 - r^{n+1})}{1 - r}$$

$$|r| < 1$$

Infinite geometric series

$$1 + r + r^2 + \dots + r^{k-1} = \frac{1}{1 - r} \quad |r| < 1$$

1) Linked vs contiguous structures.

→ array

0 1 2 ...

Direct access

arr[2]

↳ [base+i*2]

not contiguous

↓

→ No direct access. (con)

0 1 2

O(1)

copy (Pro)

data during any resize (con)

→ resize can be done quickly (Pro)

(as fast as O(1))

2) Singly Linked list

head

120

Notes do not have to be consecutive.

tail

700

Node (Dynamically created in the heap)

(we must always maintain correct head pointer)

3	35	20

0	700	35

1h	NULL	700

Linked in single direction (does not know the previous node address)

next pointer

List is made up of nodes.

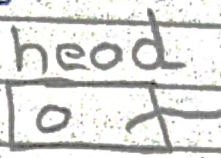
Note structure

T data

Node* next

Scanned with CamScanner

Empty List:



List is made up of nodes

(3) Linked list operations:

- insert
  - insert at head
  - insert at tail
- erase
  - erase at head
  - erase at tail ??

worst case → O(1) operations

> O(n)

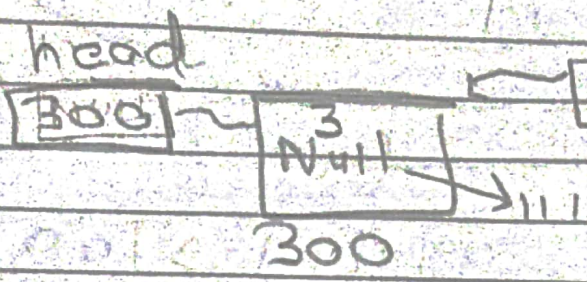
- insert → insert in order
- erase → erase key

O(n) worst case

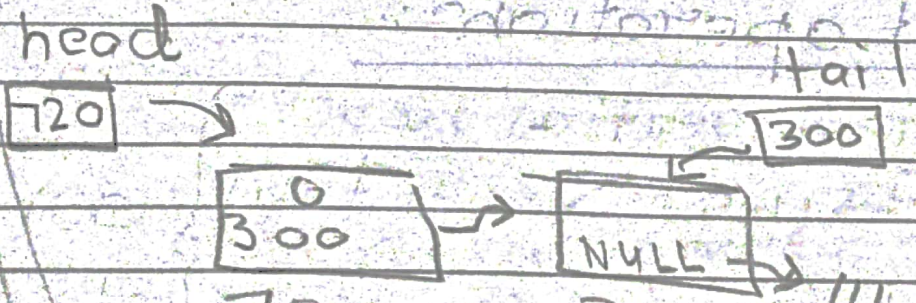
Find → ???

3.1 insert at head, insert at tail, insert in order

insert at head

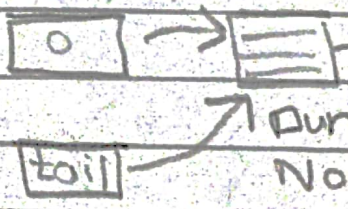


f. insertatHead(3)
f. insertathead(6)



→ create new node at 720 + add data
→ set head = 720

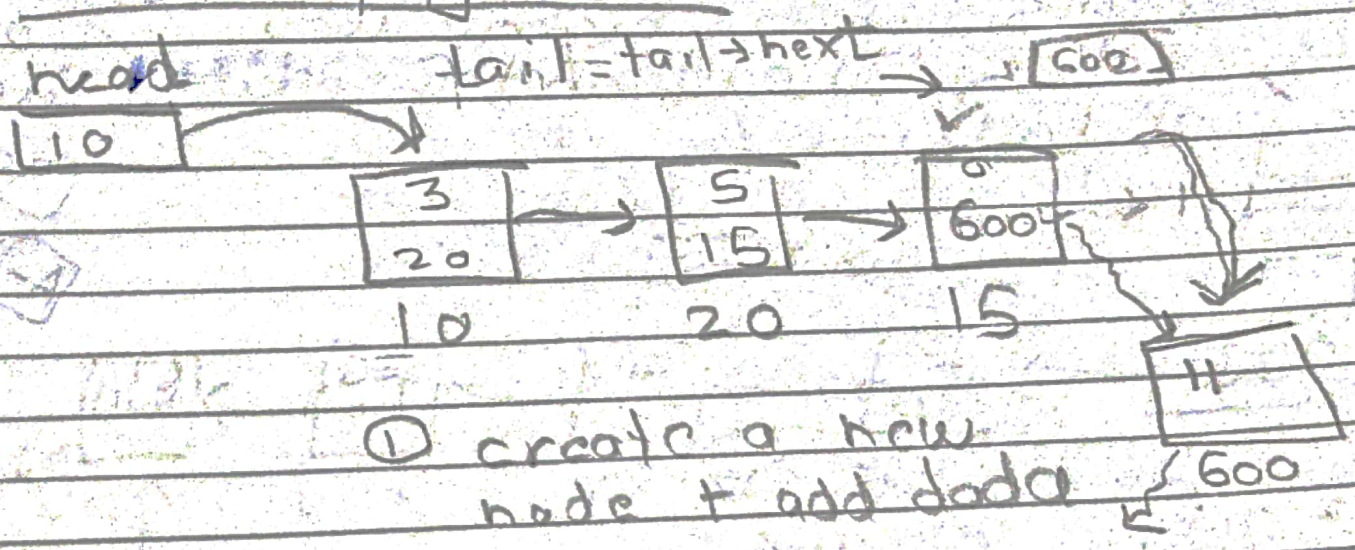
For coding head



→ set next of new node to old head
→ set head to new node address.

Insert at Tail

Non-empty list



① create a new node & add data

② Set tail's next to the new node's address

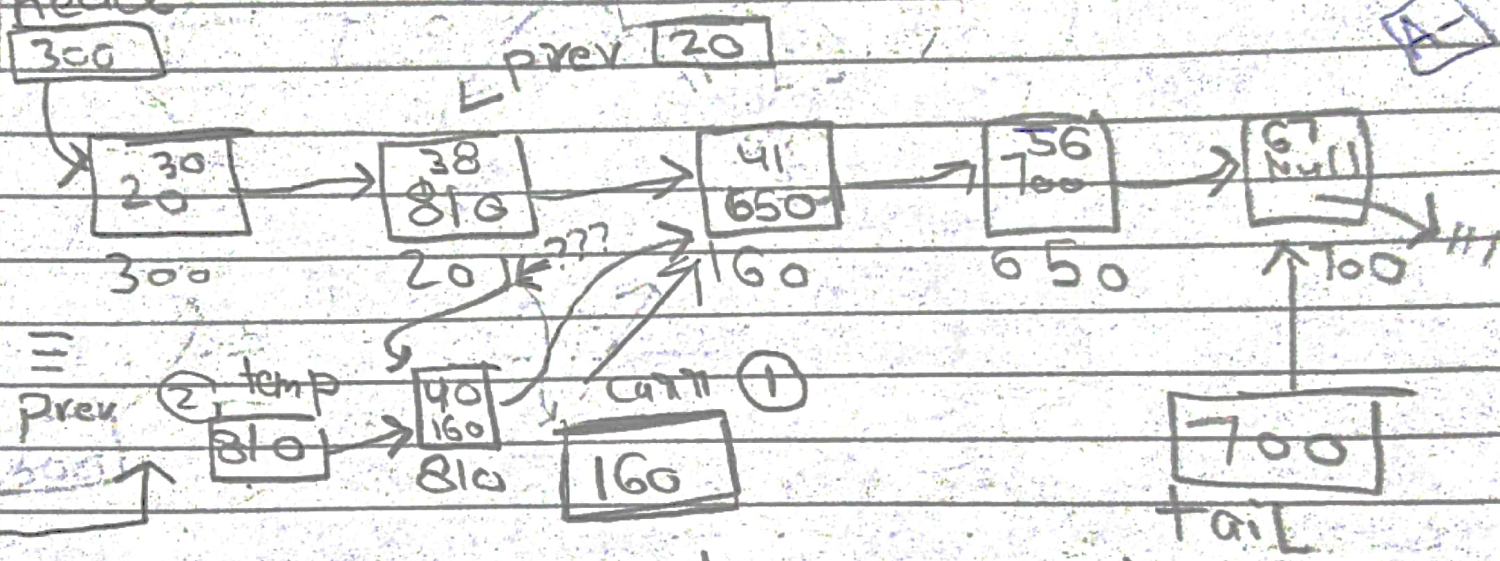
③ Next of new node set to null

④ move tail to the next node

insert in order:

→ If the list is empty? at

works in the same way insert at head and insert at tail.



l.insertinOrder(15) → same actions as insert at Head

l.insertinorder(75) → same actions as insert at tail

→ Interesting case

l.insertinorder(40)

↓ Steps

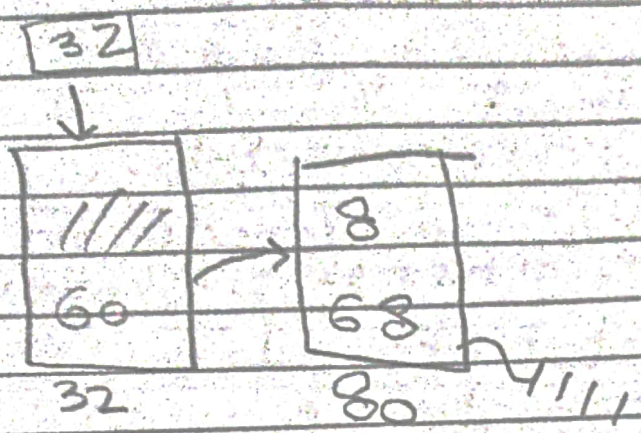
1. move current to the node just after the new node
   (Prev is one behind)
2. crete new node and use pointer temp to point to it
3. Set next of new node to current
4. Put temp in the next of the previous node
   (using previous pointer)

↓
no access

Example of linked storage:-

Test Editor :-
{ Should I use an array
[or a linklist?

head



dummy node

Erase at head:- cases: empty list → do nothing
→ size = 1 → remove node and move tail to dummy

Algo
→ size > 0 head → next = head → next → next

Erase at tail:- O(n)

→ First Find previous of next
→ Delete tail node
→ set ptail → next = nullptr
→ set tail to ptail

*Stack method for deletion.

Question:-
We wish to implement stack and Queue using a singly link list.

Stack:- LIFO
- InsertATTail → O(1) (push)
- EraseALTail → O(n) (pop)

} better choice
- Insert at head → O(1)
- Erase at head → O(1)

Queue:-
FIFO
- InsertatTail() → O(1)
- EraseAtHead() → O(1)

} good choice

Destructor of Singly linked list:-

Eraseatkey:

Step#1 Find the node which contains the key i.e find pointer current which points to it

RECURSION, RECURSIVE ALGORITHMS

① Factorial

$n! = n(n-1)(n-2)\ldots 1$

$1! = 1$
$0! = 1$

$n! = n(n-1)!$
Recursive

Expressing a function in terms of a smaller instance of itself is called recursion.

```
int fac(int n)
{
  if(n==0||n==1)
    return 1
  else{
    int P=1;
    for(int i=2; i<=n; i++)
    {
      P*=i;
    return P;
```

→ Recursion: when a problem reduces to itself

Example of reduction

6, 2, 8, 1, 5
→ min find

1, 2, 8, 6, 5
1, 2, 5, 6, 8

selection sort not reduces to recursion
→
find min

recursion is special kind of reduction

```
int fact(int n)
{
base call  if(n==0)
               return 1;
recursive call  else
               return n*fact(n-1);
}
```

fact(5)
↓ 5*fact(4)
recursive depth of call: 4*fact(3)
3*fact(2)
2*fact(1)
1*fact(0)
↓
hot recursive call = ?? = basecase
otherwise infinit recurrsionX

General Structure of a recursive function:

```
fun(____)
```

Base call → directly return an answer.

recursive call → make some combination of recursive call + some work

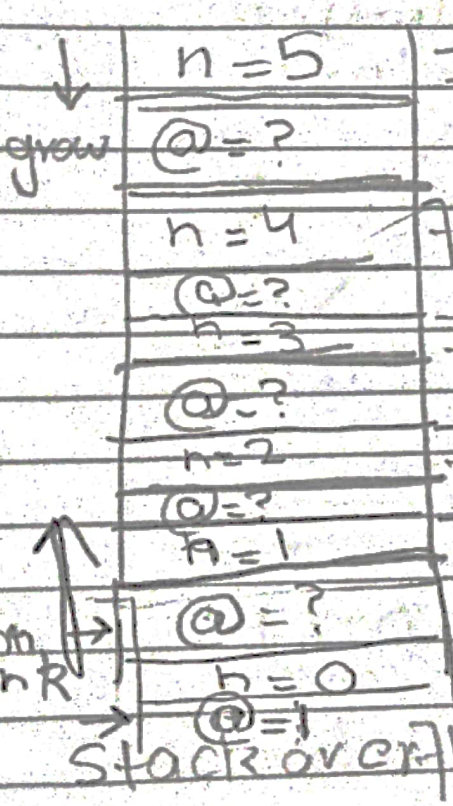
* function call → system stack

How is fact(5) computed

main

int x = fact(5);

* Recall: Every function call get a fixed size area on a system stack (Activition Record



* A recursive call grows the stack and a return shrinks the stack

→ base case.

Stack overflow } result of infinite recursion

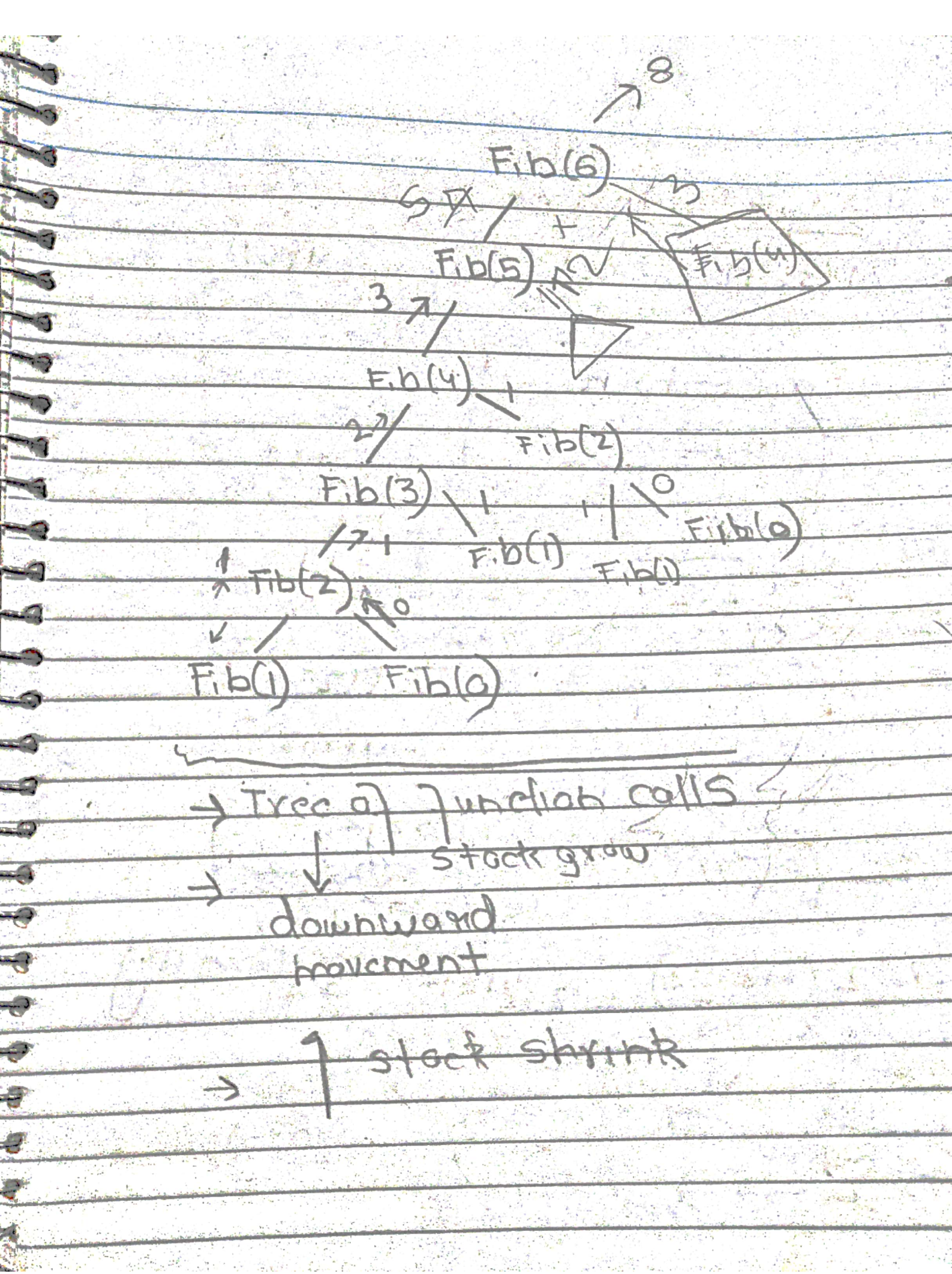
(7) Fibonacci Numbers

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
F_0 F_1 F_2 F_3 F_4 F_5 F_6

$F_n = F_{n-1} + F_{n-2}$ → The F_n reduces to $F(n-1)$ and $F(n-2)$

$F_0 = 0$, $F_1 = 1$

```
int Fib(int n){
base call   if (n <= 1)
                return n
recursive call  else {
                return Fib(n-1) + Fib(n-2)
            }

main;
    cout << Fib(6);
```



→ Tree of function calls

→ stack grow downward movement

→ stack shrink

(3) FAST POWER

P(x, n) → x^n

P = 1

For i ← to n

P ← P * x

return P

T(n) = O(n) simple power

$2^{32} = 2^1 \cdot 2^{31}$ recursive

Both are recursive

$2^{32} = 2^{16} \cdot 2^{16} = (2^{16})^2 = ((2^8)^2)^2 =$

$(((2^4)^2)^2)^2$

$((((2^2)^2)^2)^2)^2$

$(((2)^2)^2)^2)^2)^2$

$2^{27} = 2(2^{13} \cdot 2^{13}) = 2(2^{13})^2$ Squaring

a^n → n odd → $a \cdot (a^{n/2})^2$

→ $(a^{n/2})^2$ n even

$a^0 = 1$

```
int FastPower(int a, int n)
{
    base
    if (n == 0)
        return 1
    else
    recursive call
    if (n%2 == 0)
        return FastPower(a, n/2) * FastPower(a, n/2)
    else
        return a * FastPower(a, b/2) * FastPower(a, b/2)
}
```

Right way

```
int ans = FastPower(a, n/2)
if (n%2 == 0)
    return ans * ans
else
    return a * ans * ans
```

$T(n) = O(?)$ answer the question

n

n
n/2
n/4
.
.
1

lg(n)

$n = \lg n$

$2^{\lg n} = n$

exponentially smaller than n

Example::

compute a*n

a = 5
n = 9

Constraint: cannot use * or / operators

Prod = 0

For (int i=0; i<n; i++)

Prod += a;

return prod;

T(n) = O(n)

*operator → O(1)

can we do this in O(log)

Alternate Method to compute a*n without using the * operator

a=5 n=9

Prod=5	times	
Prod += Prod = 10	2	
Prod += prod = 20	4	n/2 + recursive call (5, n-times)
prod += prod = 40	8	
→ 80	(16)	

$\lg(n/2)+\lg(n/4)+\lg(n/8)+\cdots \{O(\lg^2 n)$

$+=\lg^2(n)$

int ProdwithoutM(int a, int b) {n≥1

```
if(n==0)
    return 0;
else if(n==1)
    return a;
```

base case

```
else {
    int prod=0;
    int times=1; oldt=0;
    while(times<=n){
        prod+=prod; → oldt=times
        times+=times; becomes 16
    }
    return prod+prodwithoutM(a,n-oldtimes)
}
```

40 + prodwithoutM(5,1)

ProdwithM(5,6) →

Prod without(5,6)

prod = 5 | times = 1

prod += prod
10

prod += prod
= 10

times
2
4
8

40+20+10
=70
=5*14=70

prodwithoutM(5,2)

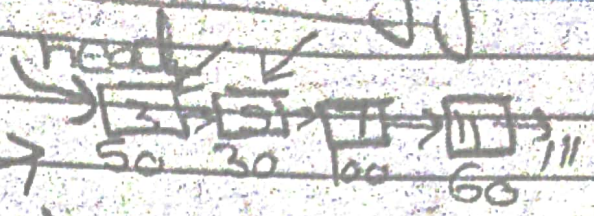
prod = 5
10

times
1
2

prodwithoutM(5,0)

Use recursion to print a singly linklist

Pvirate:



void print(Node *curr) {
Bounce-hit

if(curr != nullptr)
{

print(curr->next);

cout << curr->data

}

print(head)

print(50)

print(30)

print(100)

print(60)

print(null

forward

backtracking

round-trip

backward

to reverse the things

wrapper Method

public:

void print() {

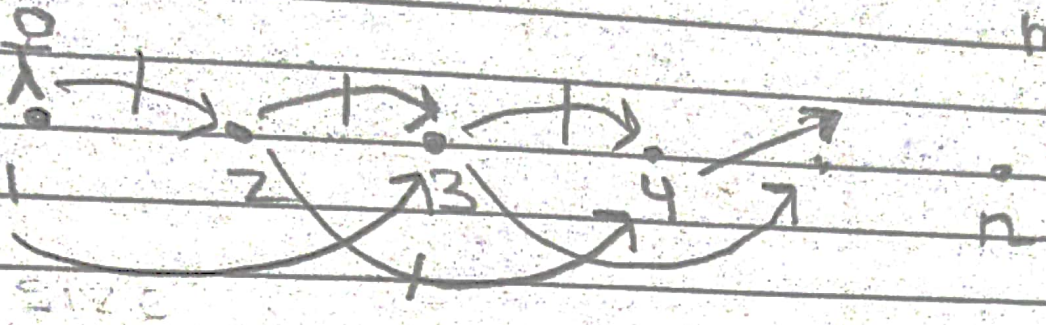
print(head)

}

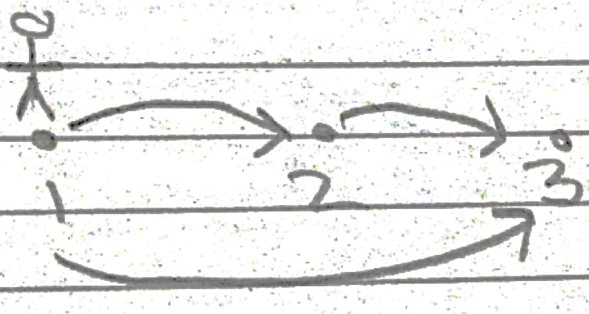
fl.print()

Parameterless or few parameters

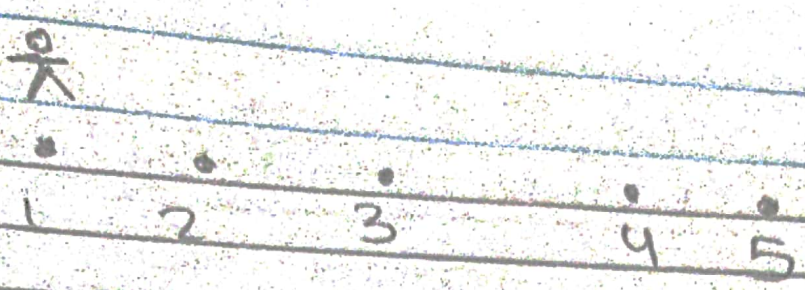
wrapper method



ways = 1



ways = 2



Fways:?

```
int numWay(int n)
{
    if (n == 2)
        return 1
    else{
        ??
    }
}
```

Graphs

Exercises (recursive).

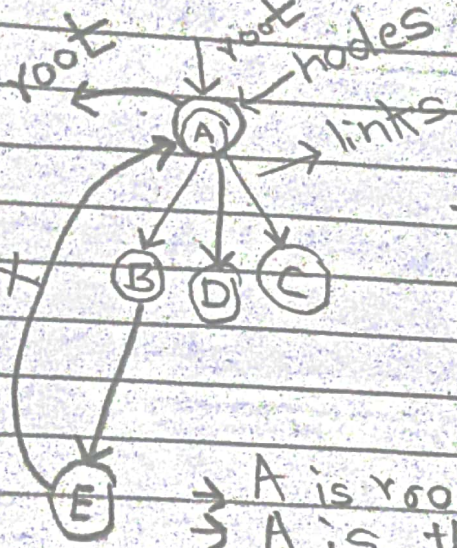
① Break a number into digits and print them line by line

n = 1234

1
2
3
4

STRUCTURES.

① Trees: Collection of (vertices) nodes and links (edges, loops, ptrs)
(directed)
properties
with certain properties
→ tree has no cycle



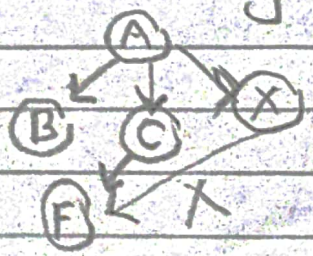
- Our trees will be rooted
→ There will be a unique root node
→ Because of root we get a hierarchy
(in trees we use the language of family hierarchy)

→ A is root
→ A is the parent of B, D and C
→ B, D and C are children of A
→ E is a grand child of A

- B, D and C are siblings

Another property of a rooted tree:

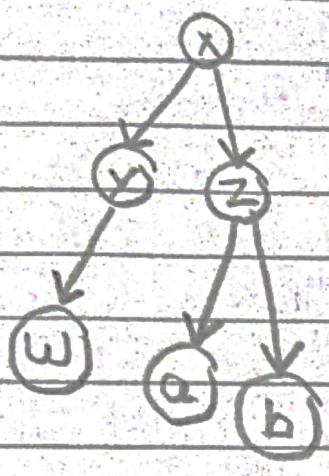
There is a unique path from the root to every node in the tree
(in other words each node has a unique parent)



Path: a sequence of loops and nodes blw A and B

② Nodes has two categories

① Internal nodes: have one or more children
↳ special internal node: root has no parent

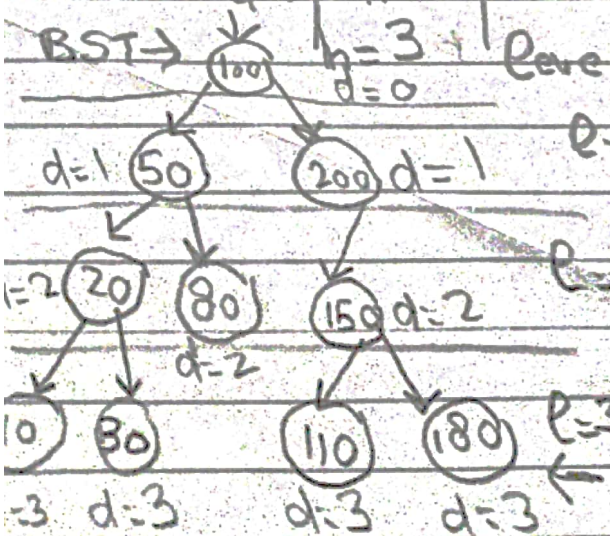


② Leaf Node: have no children

③ Some special matrices for trees

iii) Levels of the tree
i) - height of a node
ii) - depth of a node

height(h): is the length of the longest path from the node to a leaf node.



height(50) = 2
height(200) = 2

length of the path is the number of links on that path.

110, 150, 180, 200

height(110) = 0

→ leaves have height 0

height(100) = 3

→ height of the root node is also the height of the tree

ii) Depth: of a node is the length of the path from root to that node (Distance from node)

iii) Level: All nodes with same depth are at the same level

generation

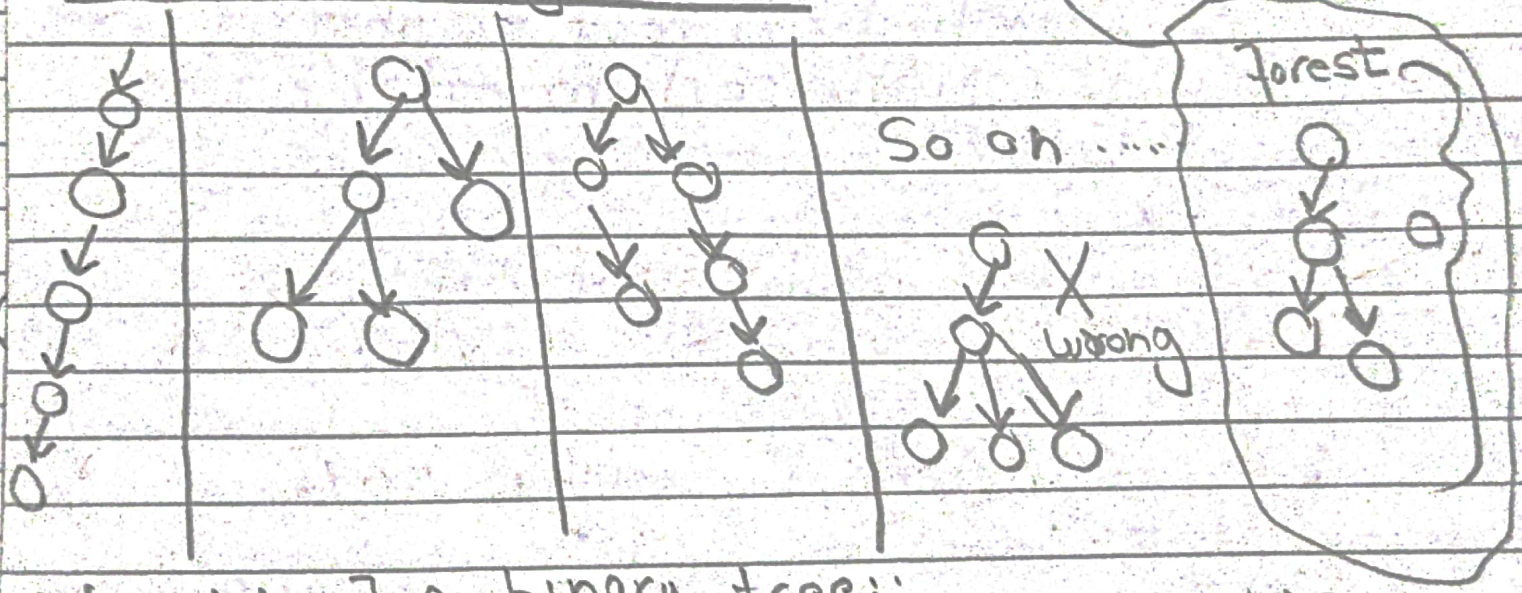
(3) Structural variations of a tree:

→ We can bound the the number of childrens.

→ e.g A tree where any node has 0,1 or 2 children is called a binary tree

→ We focus on the structure of a binary tree.

5 node Binary tree



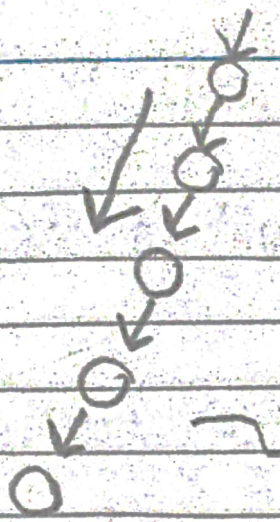
→ height of a binary tree:: $h \geq 0$

Question: Consider a binary tree over n nodes

① What could be its maximum height?

② What could be its minimum height?

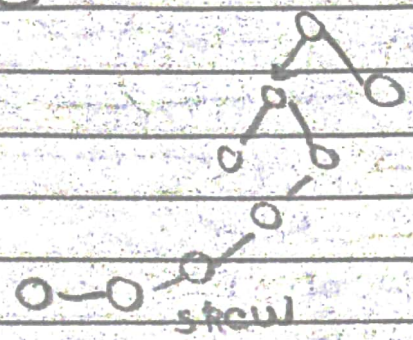
→ Ans:



→ h nodes can have max height h-1
i.e O(h)

→ So: a tree of a million nodes has height of million (10,00,000)
(≈ 2^{20})

→ Completely skewed tree



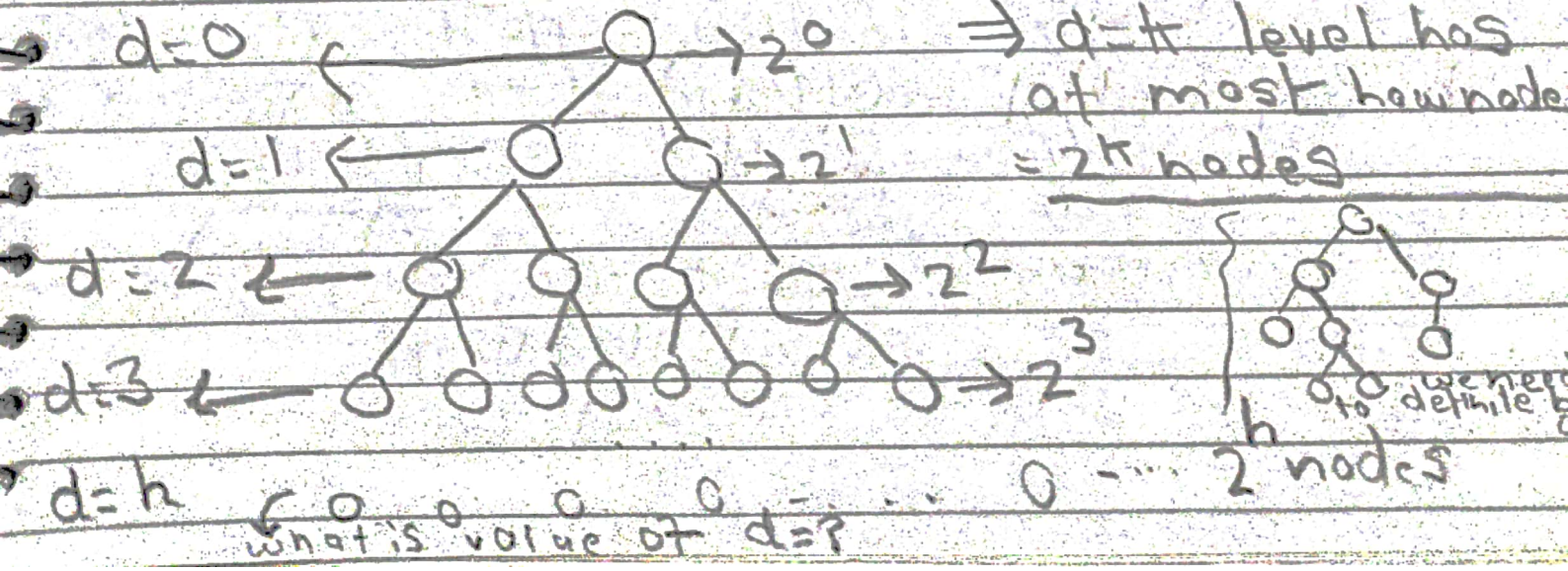
⇒ Skew increases height

⇒ For a skewed tree h=O(n)

Ans 2) Min height

→ Every level can "eat up" a lot of nodes

→ if every level is packed → Perfect Binary tree



⇒ d=k level has at most how nodes
= 2^k nodes

total no of nodes = n = (no of nodes on level 0)
+
(no of nodes on level 1)
+
⋮
(no of nodes on level h)

$n = 2^0 + 2^1 + 2^2 + \cdots + 2^h$ → geometric series

$S_n = \frac{a(r^n - 1)}{r - 1}$

$r = 2, a = 1$

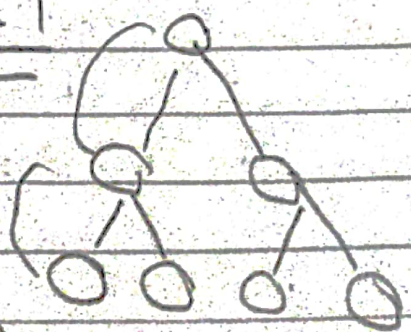
$\Rightarrow n = 2^{h+1} - 1$

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

$h + 1 = \lg(n+1)$

$h = \log(n+1) - 1 = O(\lg n)$

h = 7



$\log(7+1) - 1$

$= 3 - 1 = 2$

1. Million nodes $\approx 2^{20}$

For a perfect tree of $2^{20} = h$

what is the height? Just 20

lesson:-

$\log n \leq h \leq n$

Balanced Binary tree

skewed.

Perfect Binary tree is the most balanced

c) A search tree will also use to search, insert and erase data based on a tree

we will discuss a binary search tree.

Hint:: Time for search will depend on height

$h = 2^{20}$ $\log(n) = 20$ | $2 \lg n$

Lecture 14

Height Balanced Tree: is such a tree whose height $h = O(\lg n)$ ⇐ good

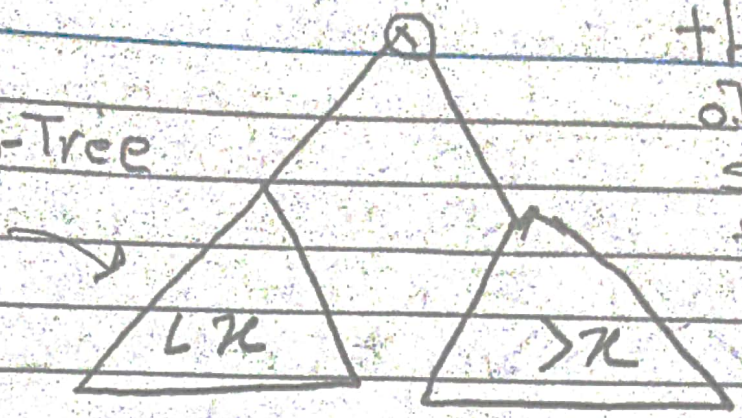
Binary Search Tree:

It has two properties:

① Structure property: It should be a binary tree.

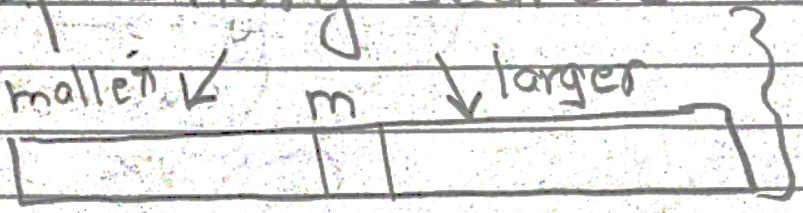
② Search Property:

For any node x,



the left subtree of x contains smaller keys than x and the right subtree contains larger subtree than x

- Designed to mimic the behaviour of Binary Search



Binary Search
O(lgn)

Operations on a BST (as a Dynamic Set)

↳ Size can change at run time

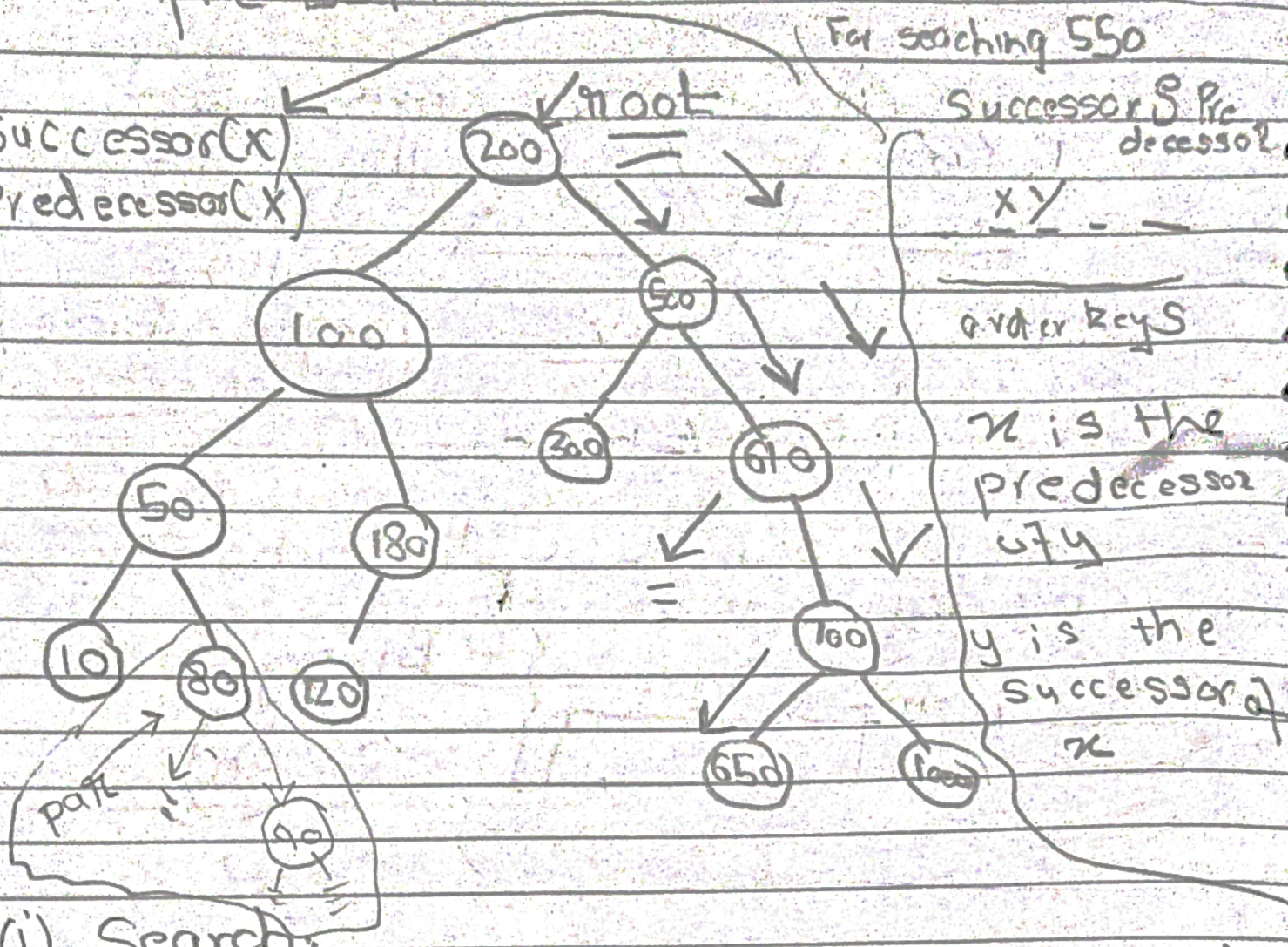
- → Insert
- → Erase
- → Search

} v.v.i Imp
*We must maintain both the structure and search properties while executing these operations

- → Housekeeping: copy const, Dest, ...

- print → print the data in accessending order of keys

Example BST:



(i) Search:

- Start from root as current node
- If the current node is X, return Success
- else if curr node is null return failure
- else if x is smaller than the curr node data, move to the left subtree

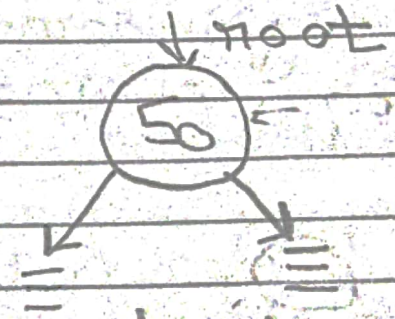
else if x is larger than the curr node data, move to

the right subtree.

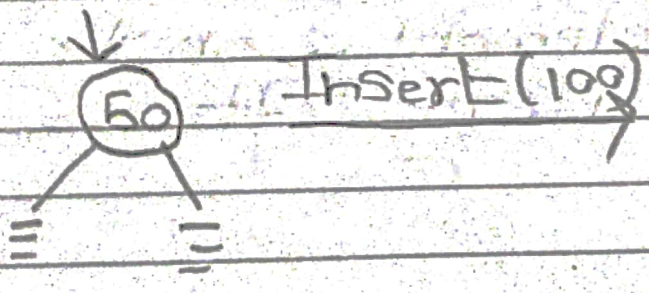
$T(h)$ = length of the longest path from root to a leaf.

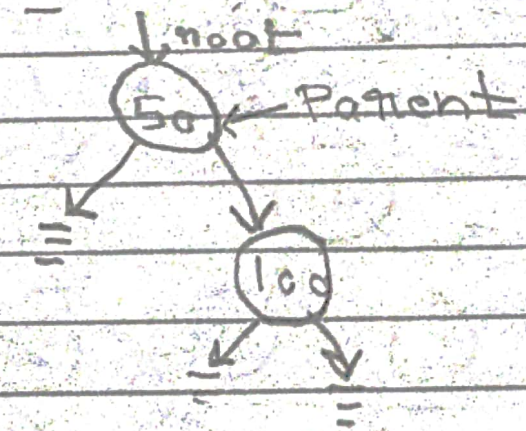
$T(h) = O(h)$ { remember $c\lg h \le h \le dh$

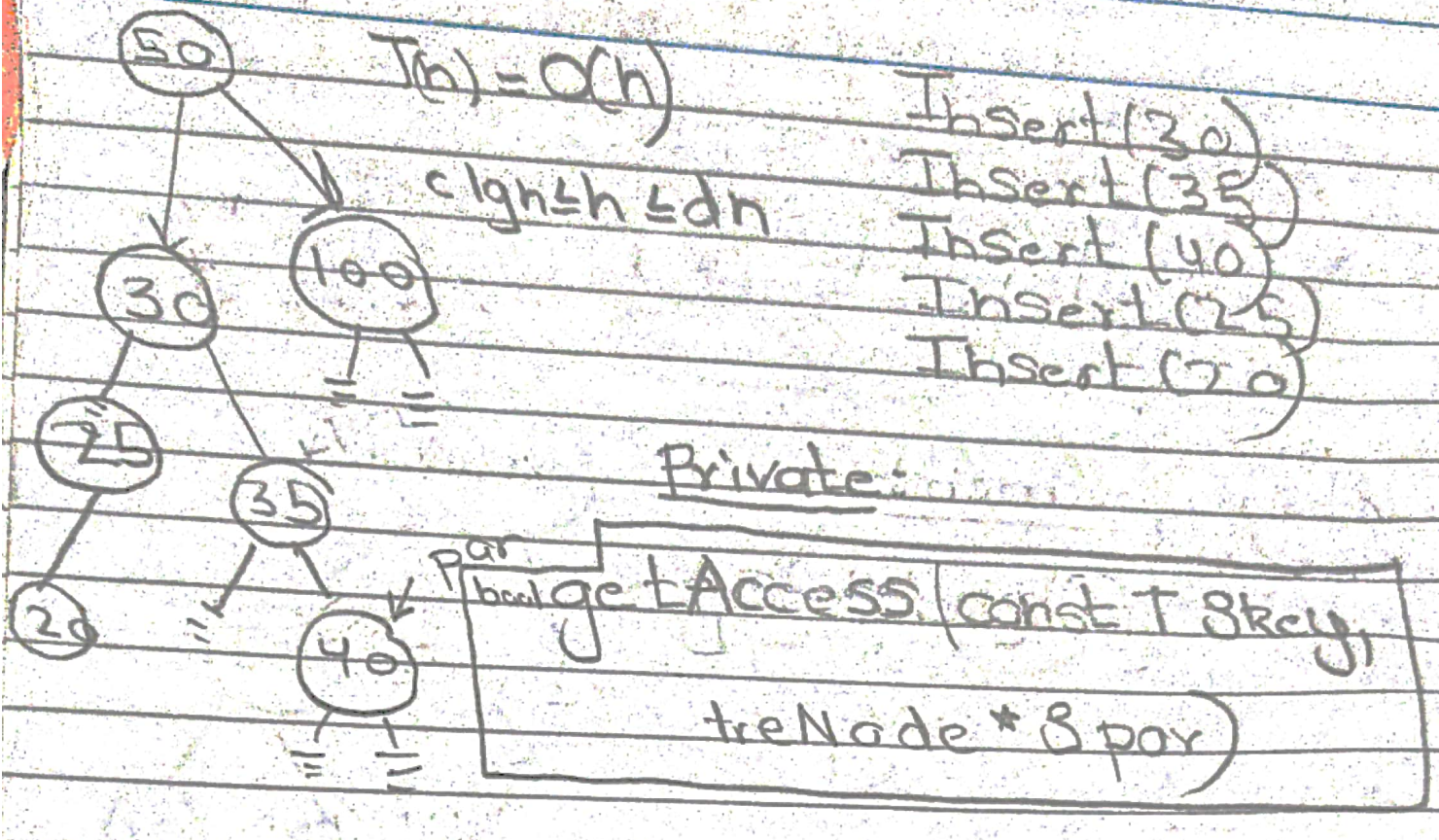
ii) Insert(x):



empty tree

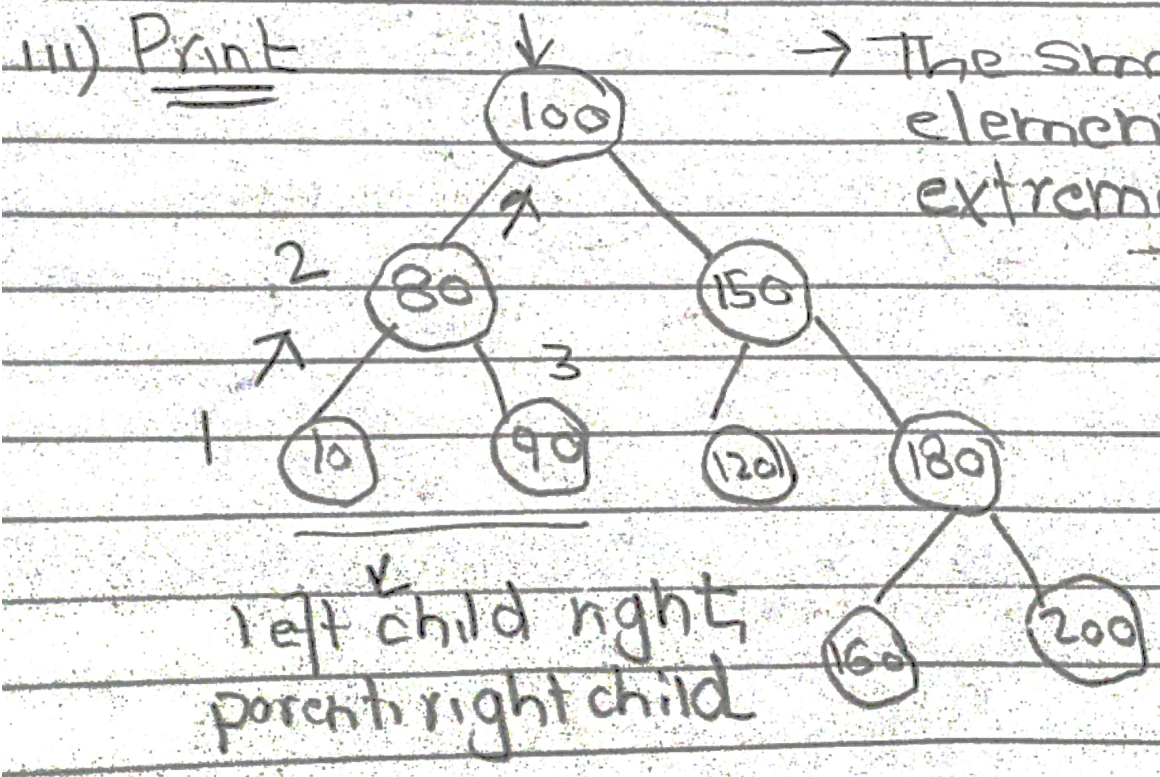




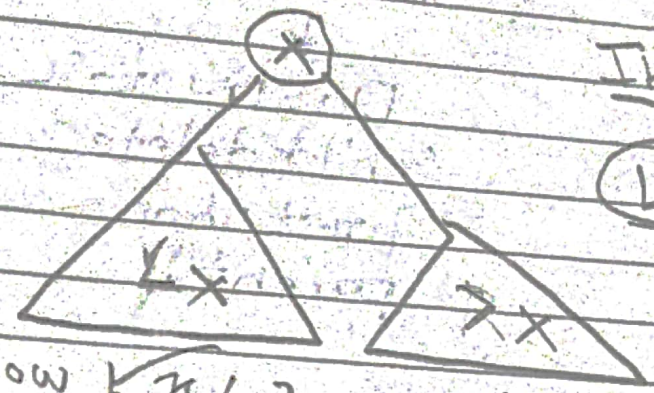


Note: Every incomming node in a bst becomes a leaf node

iii) Print → The smallest element is on the extreme left



10, 80, 90, 100 → Recursion



Allow L<x>L<x> repetation.

Inorder Traversal

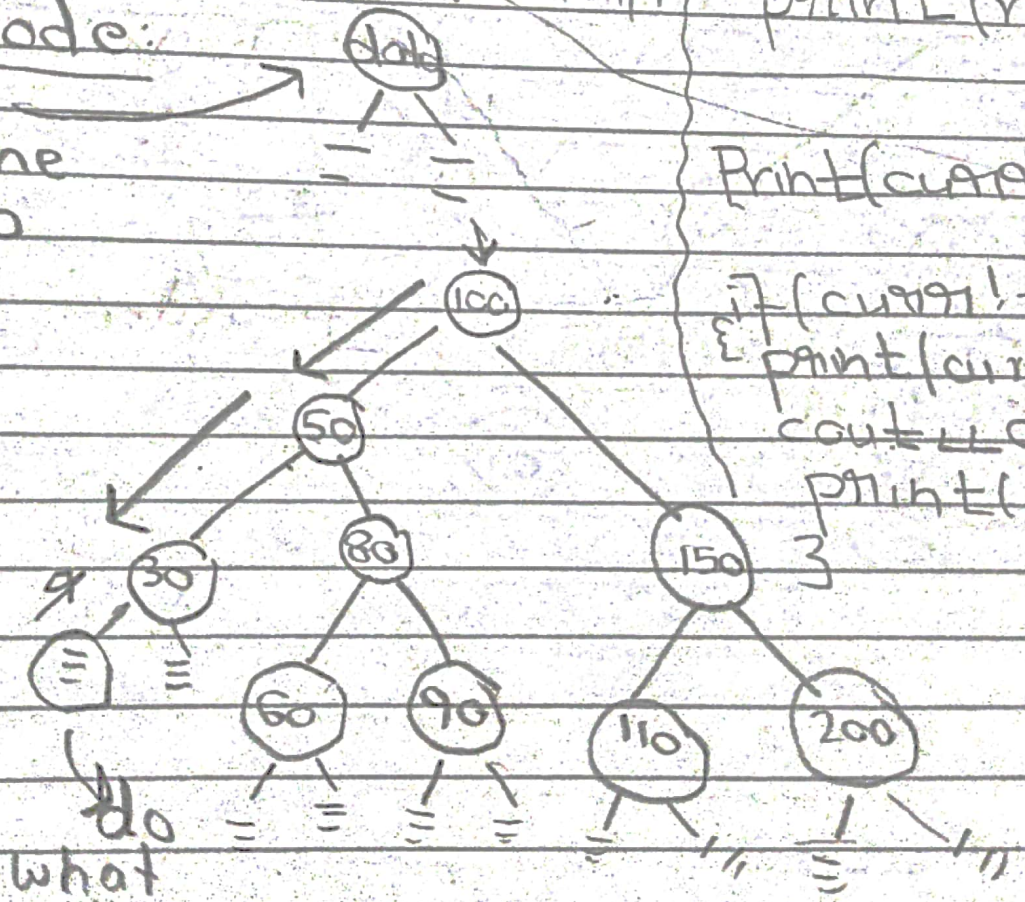
LNR → v.v.v Imp

left Node right

```
Print(30)
print(nullptr)
cout<<30
print<<nullptr
```

```
t1.print()
print(root)
Print(curr)
if(curr!=nullptr)
{print(curr->child)
cout<<curr->data
print(curr->child
}
```

treeNode:



on the heap

do what nothing?

LRN V.V.V Imp

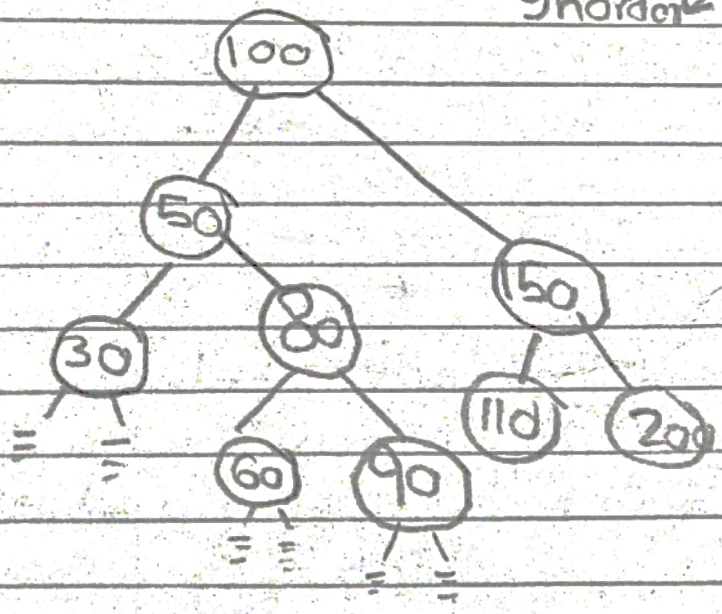
Postorder
30, 60, 90, 80, 50, 110, 200, 150, 100

Print(80)

Print(60)
Print(90)
cout << 80

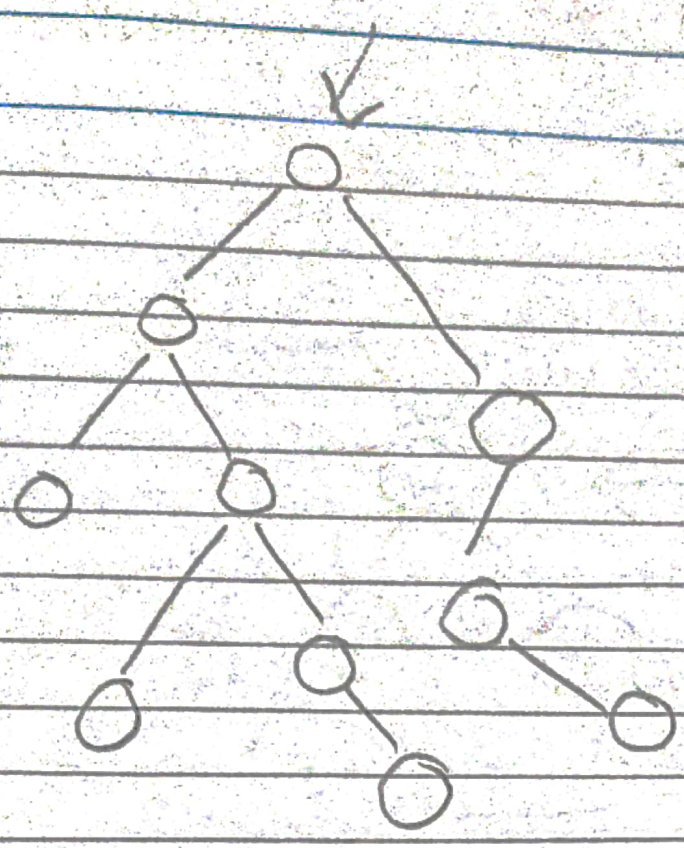
Post order
↓
Destructor
"child delete
first than
parent"

copy constructor:



preorder → LNRX
LRN ✓ → postorder
NLR ✓

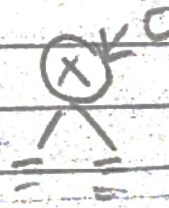
preorder



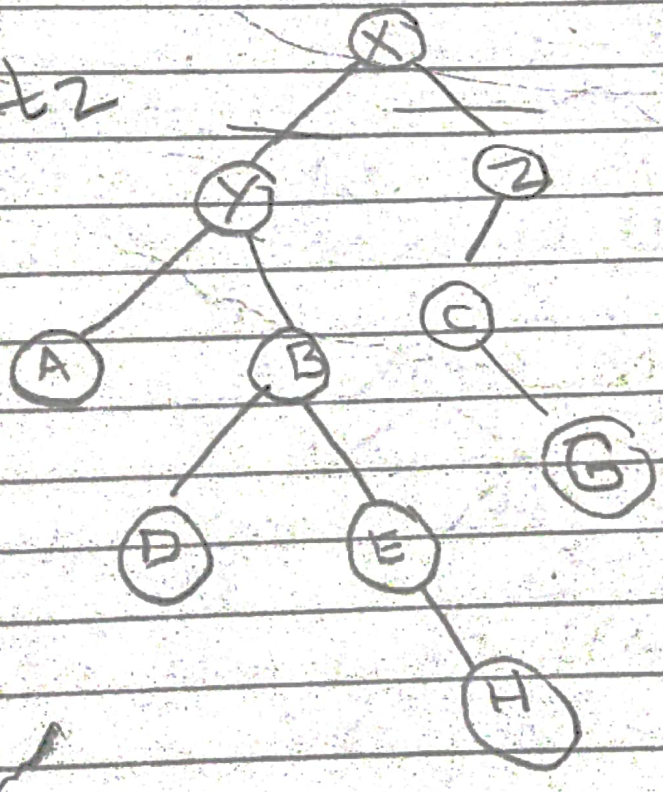
$O(n^2)$

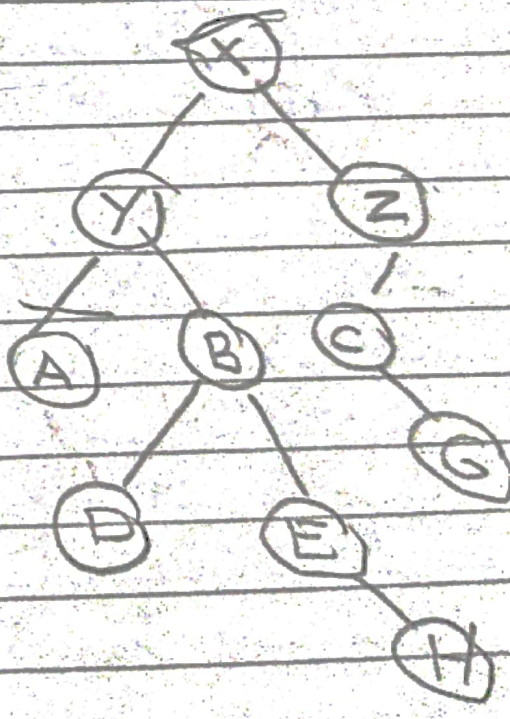
$O(h)$

To make deep copy



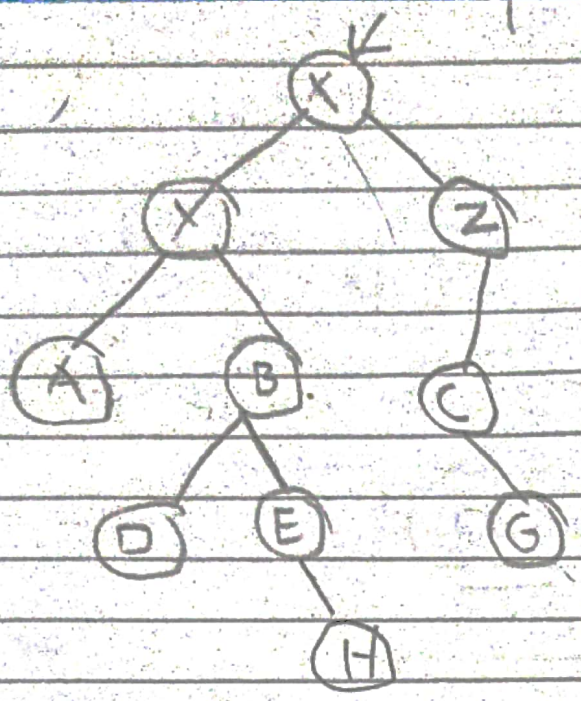
t2

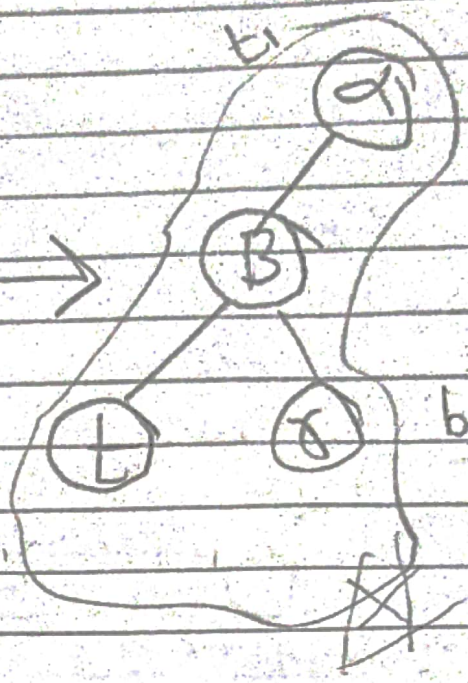




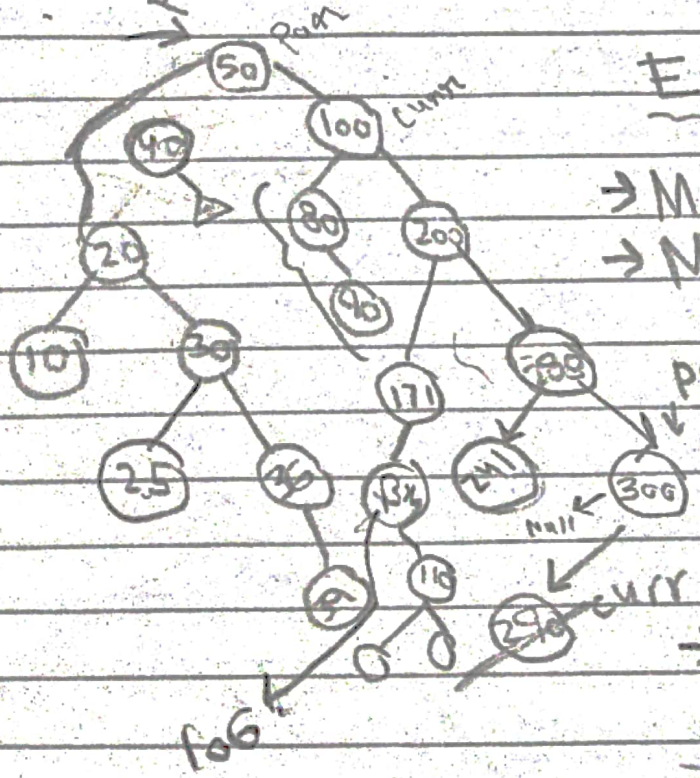
Difference b/w assignment and copy:-

t2 (Why we do delete in assignment operator?)



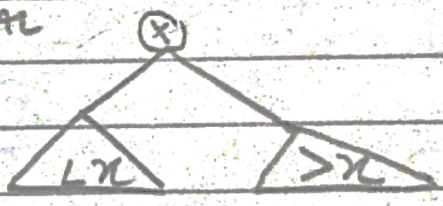


Erase:



Erase must do following

→ Maintain structure property
→ Maintain search property



→ Node should be erases

case1 → erase(290)
case2 → erase(40)
case3 → erase(100)

(i) erase case 0: erase a leaf Node { erase(x) x has 0 children

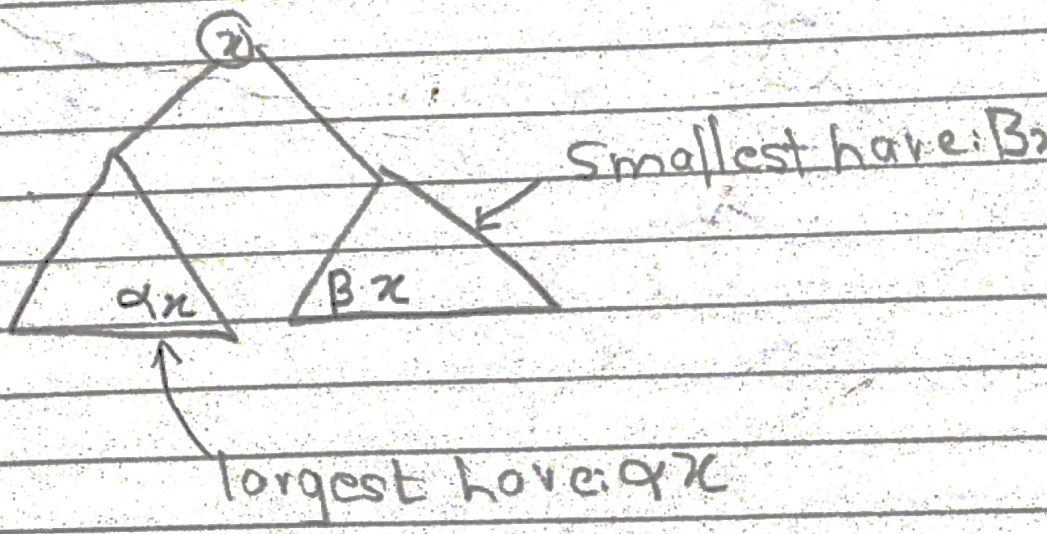
ii) erase case 1: erase a node with 1 child { erase(x) x has 1 children

iii) erase case 2: erase a node with 2 child { erase(x) x has 2 children

Scheme

Convert it into either case 0 or case 1

case III



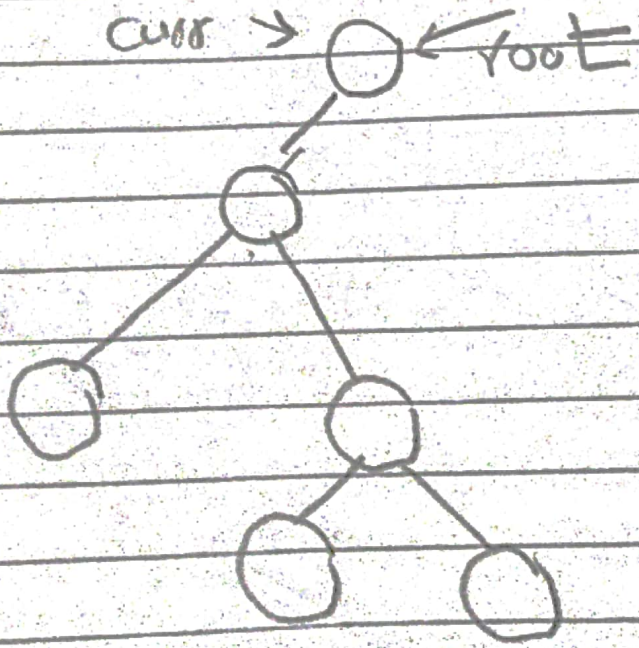
Conversion:

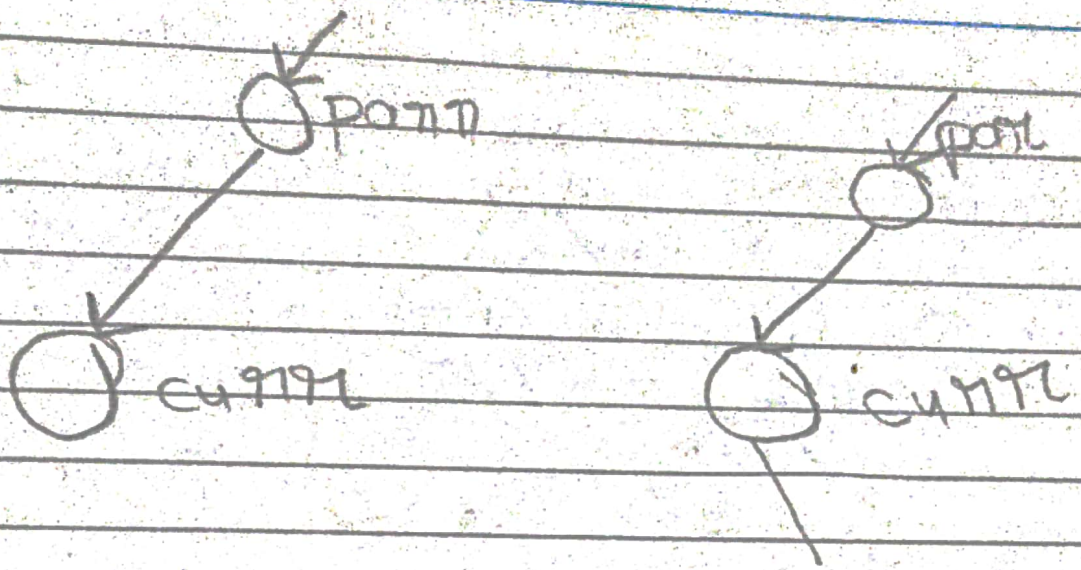
(i) Find Bx the smallest key node in the right subtree of x

(ii) Move Bx into the place of X

(iii) Delete/Erase the 'hole' created at Bx using either case0 or case1 (case2 not possible as Bx never has left child)

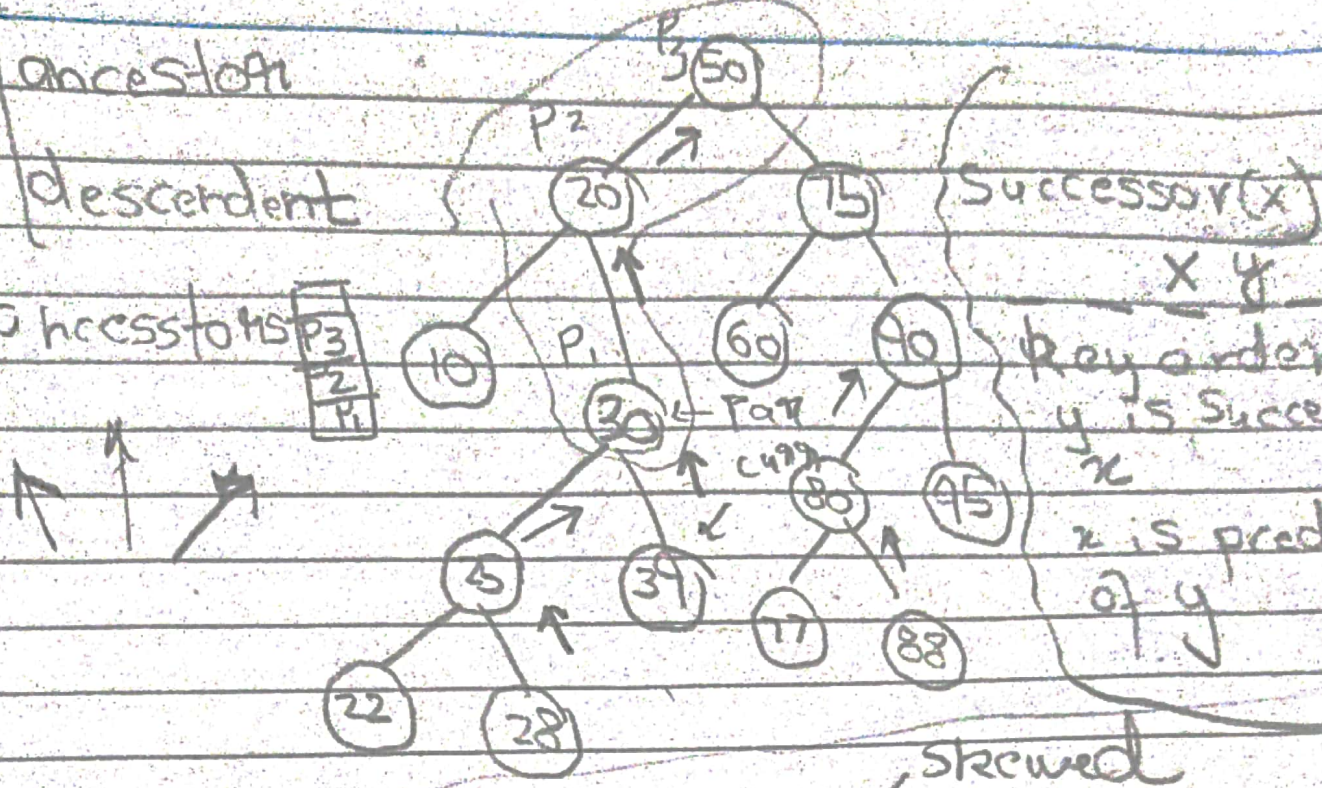
Erase case 1 (delete root):-





ancestor

descendent



successor(x)

x y

key order

y is successor of x

x is predecessor of y

skewed

Final Good

height balanced tree

insert
search
erase
} O(h)

clgh ≤ h ≤ ch

height balanced

$h = 2^{20}$

$n/2 = 2^{19}$

$n/4 = 2^{18}$

- Successor of x, may be decendent of x or an ancestor of x

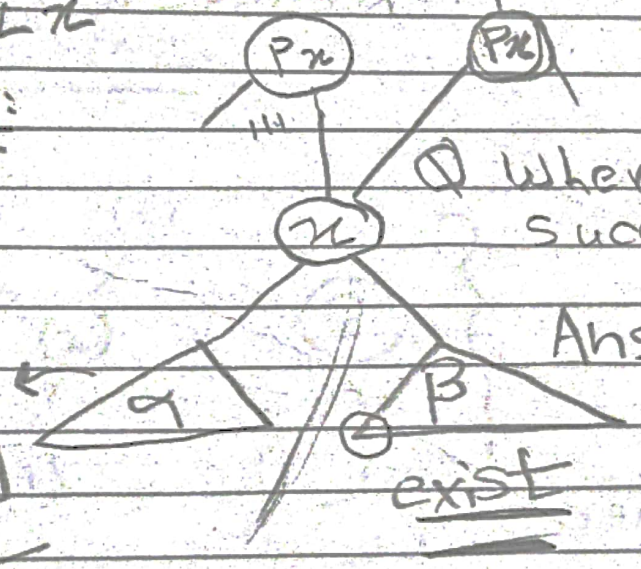
⇒ Using inorder traversal to find successor of x, takes O(n)

→ If will be better to find it in $O(h)$ so that when we have made a height balanced bst, it behaves $O(\lg n)$

Lets focus on a node x, when successor is required

$P_x L x$ $\quad$ $x L \beta L P_x$

Case:



Q Where will be the successor of x?

Ans: $succ(x) = y$ should be the left most node of β

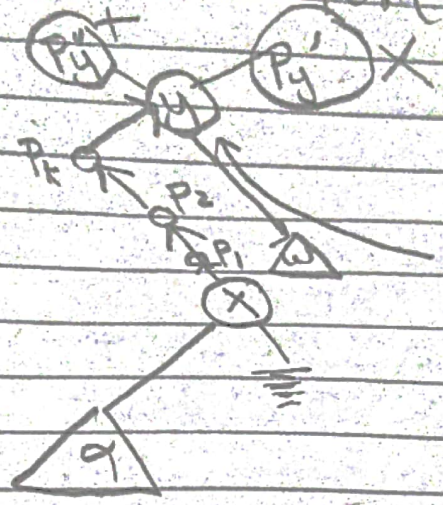
may or may not exist ← exist

(A)

- If x has a right subtree, then its successor must be the left most node of the right subtree

(B). If x has no right subtree, then its successor is the first rightward ancestor (moving upward)

Case B:

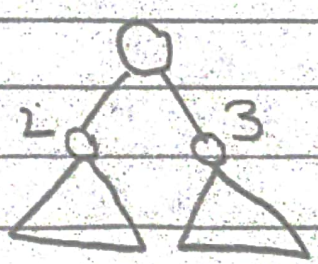


~~B to a~~
~~Balan~~

Assignment 4

Balanced factor:-

$bf \in \{-1, 0, 1\}$

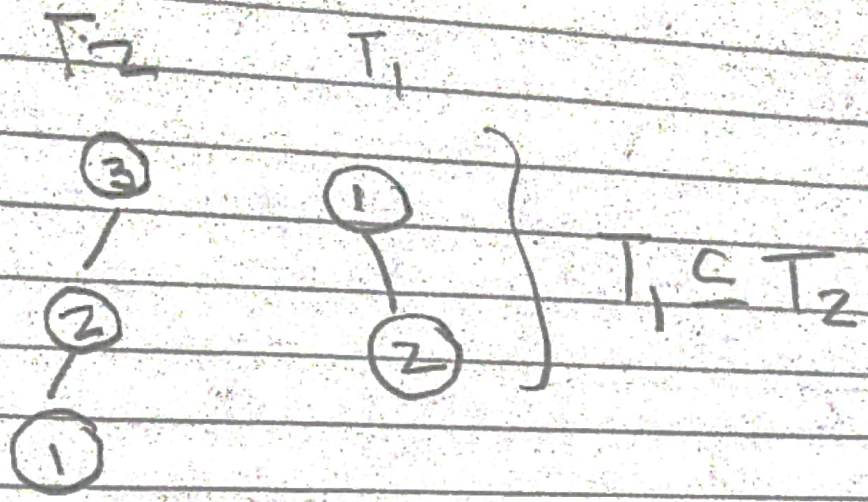


$bf \in \{-1, 0, 1\}$

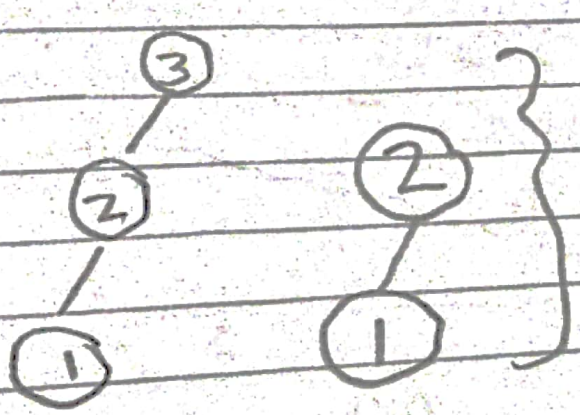
1 → more height to the right

0 = -1 → left heav

Subset



Subtree



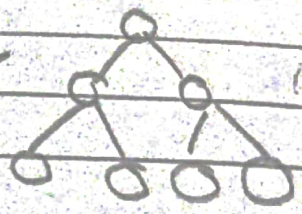
① BST & height (h)

Search
Insert
Erase
} O(h)

$\lg(n+1)-1 \le h \le n-1$

Approach:

i) compute height
ii) store height in each node
} later



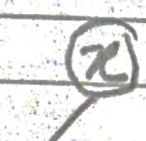
$\lg(n+1)-1$ $\lg n$)

How to compute height

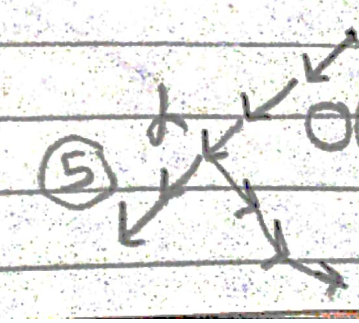
h(x)=0

h(nullptr)=-1

h(nullptr)=-1



longest path



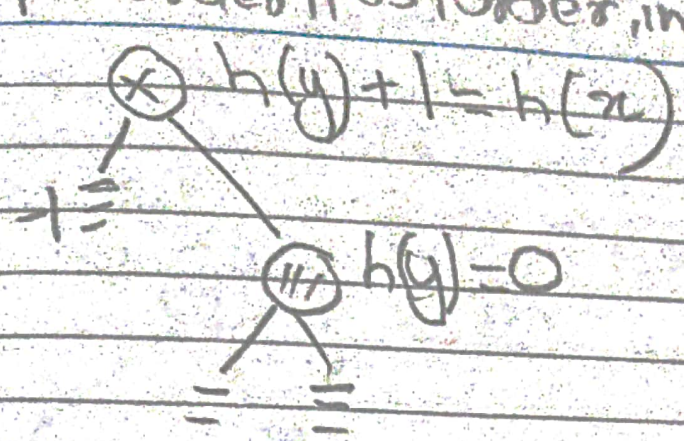
h

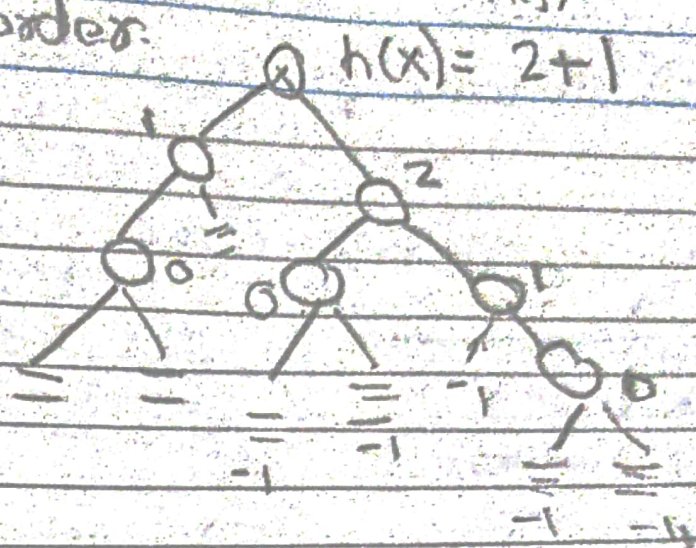
O(h)

⑤

h(x) length of longest path from x to any leaf

preorder, postorder, inorder.





We can compute height in postorder

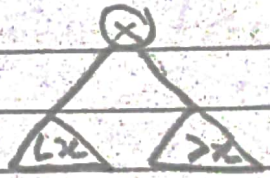
```
int height(treenode *ptr)

if(ptr == nullptr)
    return -1;
else{
    return(1+max(height(ptr->lchild),
                 height(ptr->rchild
}
```

② A way to restructure the tree

→ Rotations are used to re-structure a tree.

→ ~~Search~~

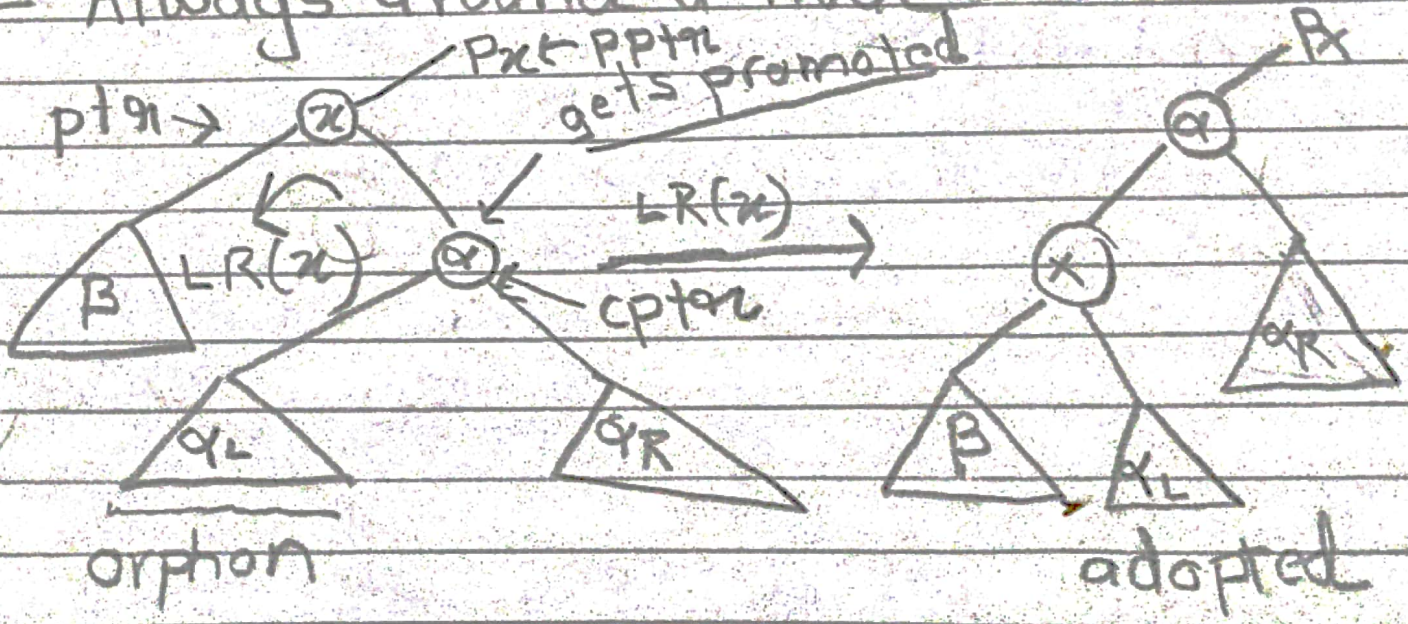
→ We must the structural and search properties of BST



(i) Left Rotation (towards left)

(anti-clockwise) material from right to left

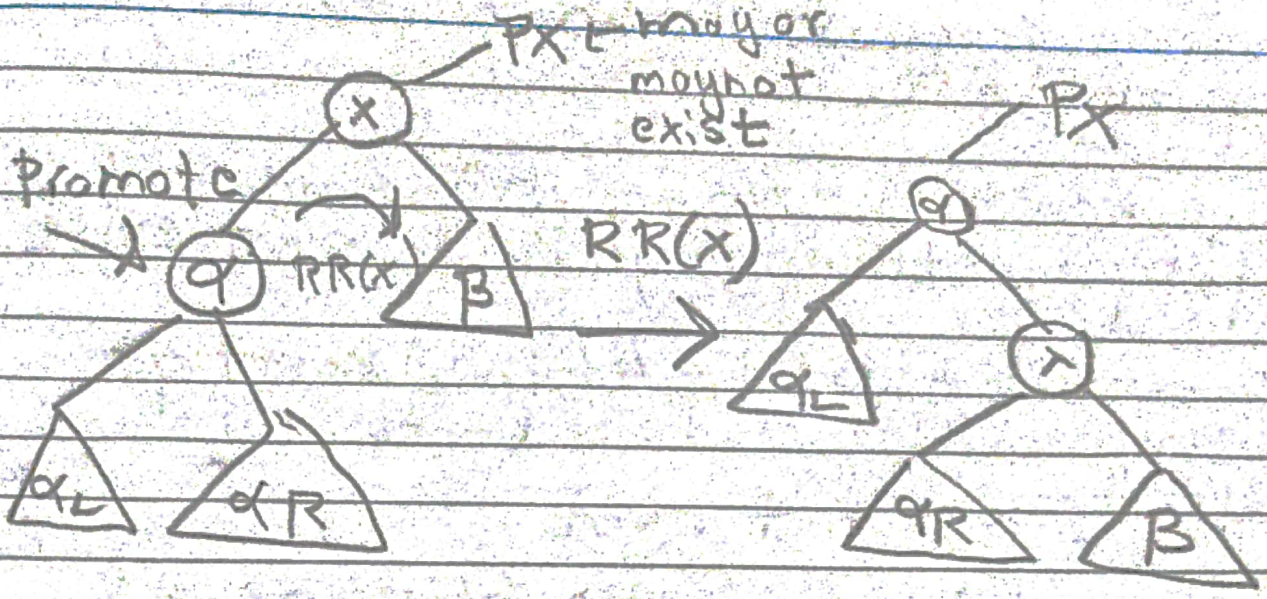
- Always around a node



$\angle x \angle \alpha_L \angle \alpha \angle \alpha_R$ $\beta \angle x \angle \alpha_L \angle \alpha \angle \alpha_R$

material from left to right

ii) Right Rotation (toward right)



αL ∠ q ∠ αR ∠ x ∠ β αL ∠ q ∠ αR ∠ x ∠ β

Class Bst

Private: O(1)

```
BoolRl(treeNode*ptq, treeNode*ppt
{
    treeNode * cptq = ptr→rchild; //
    if(cptr == nullptr) //no
        return false;
```

// promote α

if(pptr == nullptr) // Px does not exist

root = cptr; // x is root

// α become root

else { // x is not root

// α must become the appropriate

// child of Px

α become left child of Px

if(ptr == pptr → lchild)

pptr → lchild = cptr;

else

pptr → rchild = cptr;

α become right child of Px

}

// x must adop qL

treeNode* orphan = cptqx->lchild;

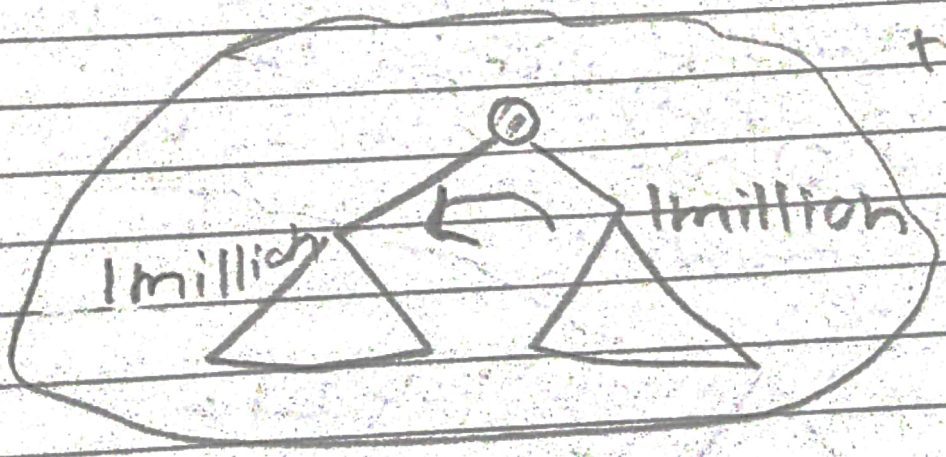
cptqx->lchid = ptqx; // x becomes child of q

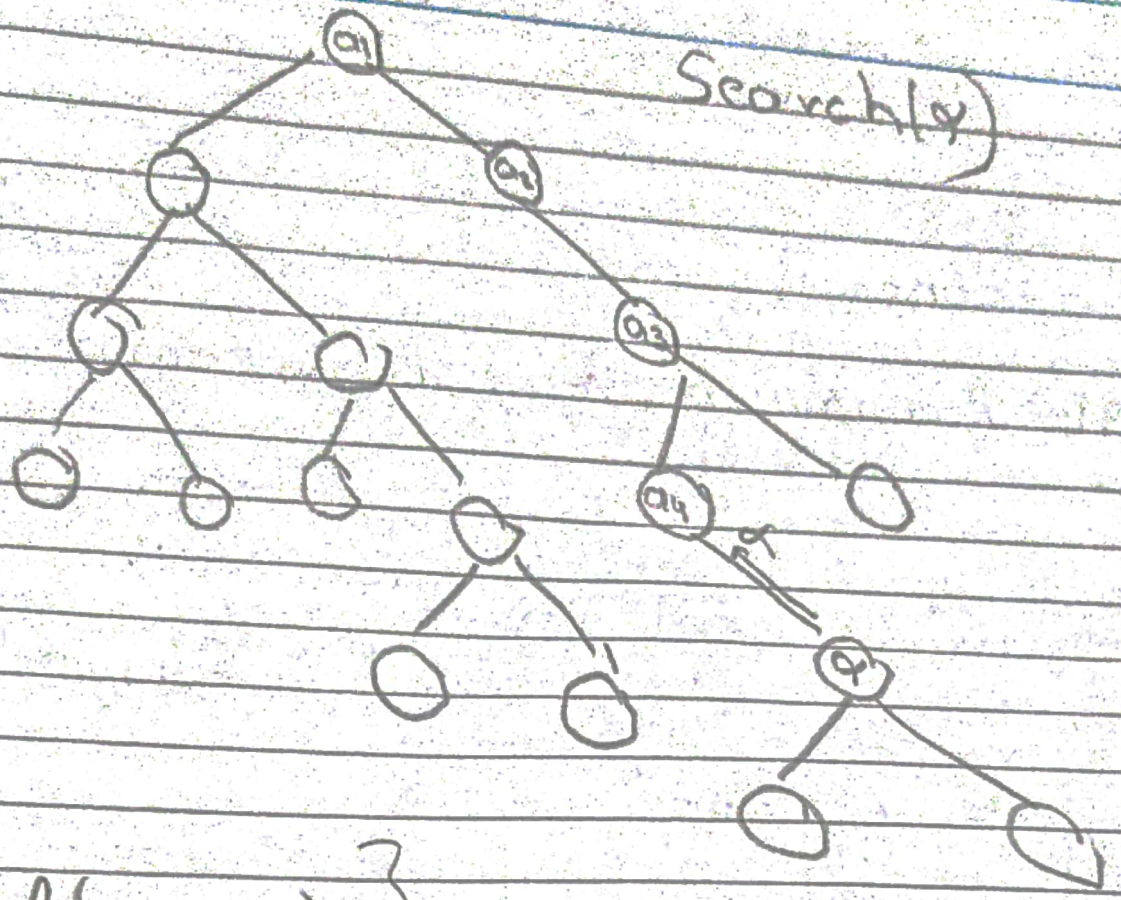
ptqx->qxchild = orphan; // qL become righchild of x

return true;

} // end of solution

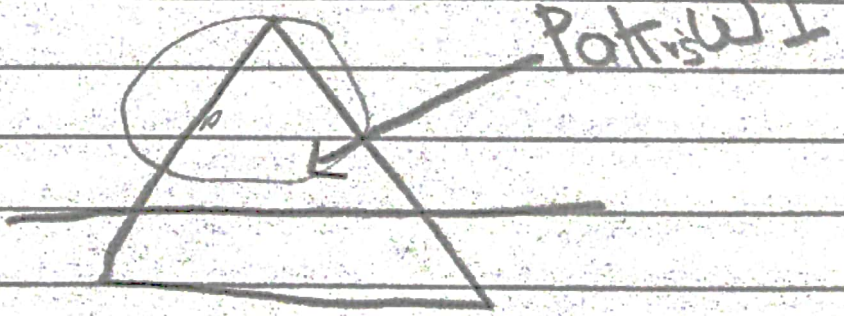
boolqx(treeNode* pt
treeNode* pptqx)





rl(a4 ...)
rl(a3 ...)
rl(a2 ...)
rl(a1 ...)

searchandPromote
O(h)



Splaytrees

Self reorganizing Data Structure:

Average case is good

Topics:

→ The AVL Property of Height Balanced BSTs

→ How to maintain the AVL Property

(A) our BST now must have 3 properties:

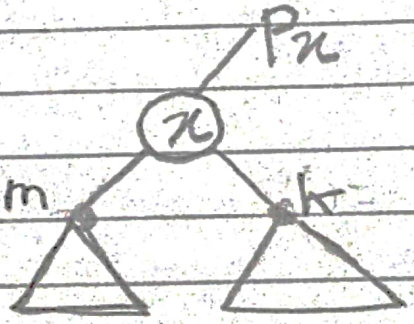
i) Structure Property

ii) Search property

iii) AVL property: (We will store every nodes height inside the node, to compute balance factor)

$\forall$ nodes x: bf(x) $\in \{-1, 0, 1\}$

where bf(x) = height(x→rchild) − height(x→lchild)



bf(x) = k − m $\in \{-1, 0, 1\}$

Types of nodes:

bf(x) = 0 → equal balanced

bf(x) = -1 → balanced but left heavy

bf(x) = 1 → balanced but right heavy

If inbalance is created (temporarily), then it may be of the following types:

bf(x) = -2 → left inbalance

bf(x) = 2 → right inbalance

} we will remove the inbalance