Camlin Page
Date / /

सरस्वति नमस्तुभ्यं वरदे कामरूपिणि ।
विद्यारम्भं करिष्यामि सिद्धिर्भवतु मे सदा ।

5

Algorithms

Objectives:
This course will covered some of the advanced data structures like Fibonacci Heaps, Treaps, AVL and red black trees. It covers the algorithms design techniques like Divide and Conquer, Greedy algorithms and Dynamic Programming. It also covers Graph algorithms including shortest path problem and Minimum Spanning tree and Network flows.

Prerequisite: Basic Data Structures like Arrays, stacks, queues, linked lists, trees, binary trees and travels methods, binary heaps, hashing and graph representation.

Contents: The course covers

- Algorithmic analysis : Revive of Asymptotic notations for algorithms, recurrence tree methods, complexity classes
- Abstract Data Structures: Binomial and Fibonacci Heaps, Balanced Binary Search Trees, AVL Trees and Red Black Trees and their applications
- Algorithmic paradigms: Divide and conquer, Dynamic Programming, greedy algorithms including metroid's:
- Graph Algorithms: Graph traversals: DFS and BFS, shortest path problem and the spanning tree problems. Network Flow and applications.
- Randomized Algorithms: Las Vegas and Monte Carlo paradigms, some example randomized algorithms

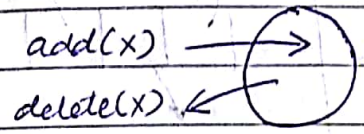
Text Book / References

- Introduction to Algorithms by Thomas H Cormen, Charles E Leiserson, Ronald L Rivest and Clifford Stein, MIT Press, 3rd Edition 2009.

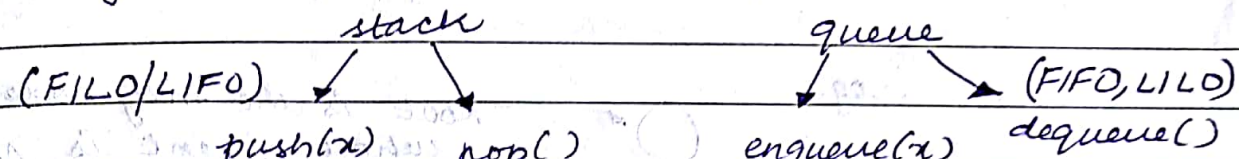
30

PREPARATORY TERM. (L-2).

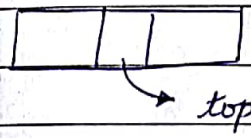
- **Dynamic Data Set** :- values of the elements in the set keeps on altering.



eg: DDS is implemented in form of



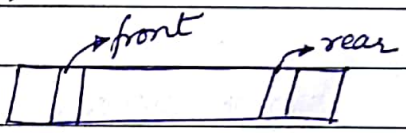
stack. (array based implementation)



push := D[top++] = x

pop := D[top--]

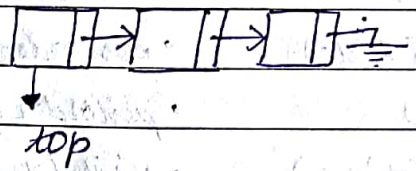
queue.

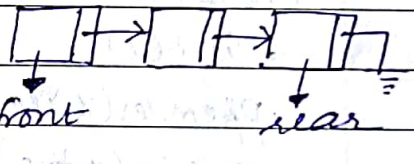


enqueue := Q[rear++] = x

dequeue := Q[front++]

(linked list based implementations)



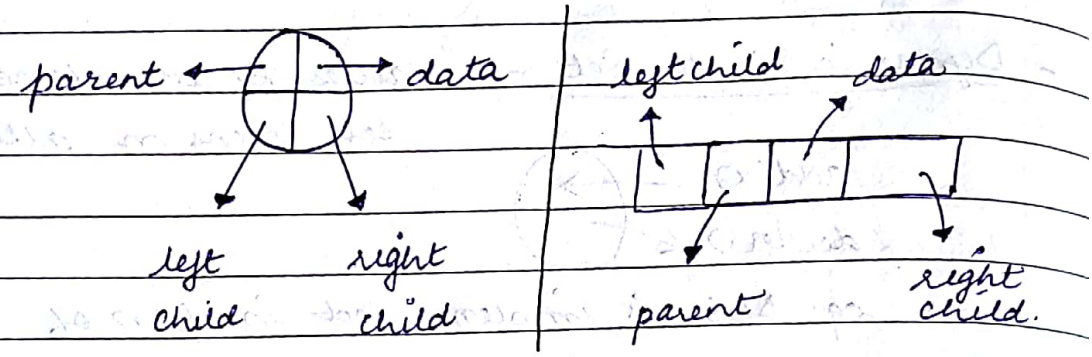


Constraint :-

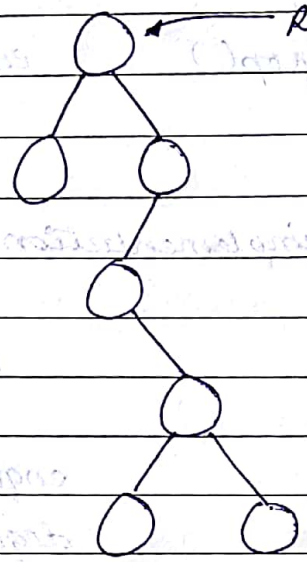
size of my dataset = 10^6 elements.
and I have 1000 stacks,

↳ In such a case, I cannot use array.
(array's size would be $10^3 \times 10^6 = 10^9$)
so, when you've multiple stacks, it is preferable to use linked list.
(don't Return values, use double pointers)

- **Binary Trees :-**



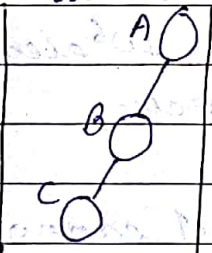
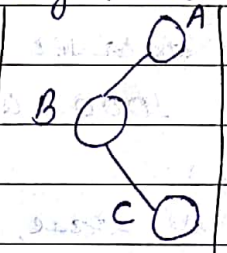
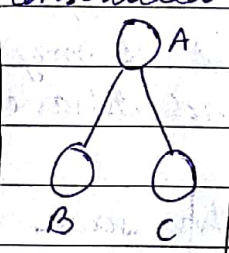
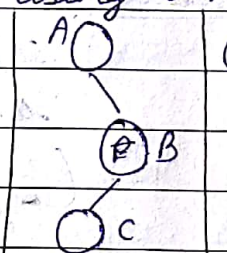
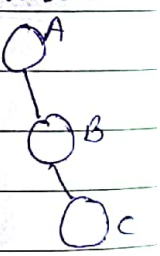
eg:



↳ **Traversals :-**

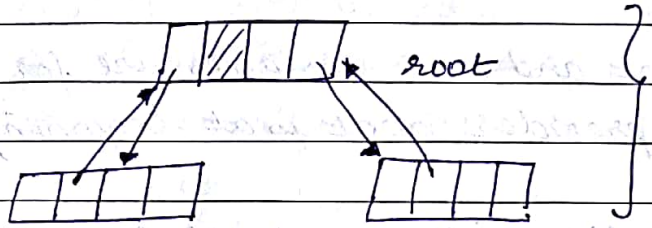
Preorder(x)	Inorder(x)	Postorder(x)
• visit(x)	• Inorder(x→lc)	• postorder(x→lc)
• preorder(x→lc)	• visit(x)	• postorder(x→rc)
• preorder(x→rc)	• Inorder(x→rc)	• visit(x)

↳ Consider all binary trees constructed using 3-nodes.

	(X)	(Y)	(Z)	(W)	(M)
					
preorder	ABC	ABC	ABC	ABC	ABC
Inorder	CBA	BCA	BAC	ACB	ABC
postorder	CBA	CBA	BCA	CBA	CBA

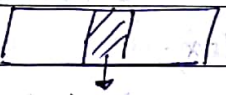
- From the above illustration, we can conclude that with only one traversal, we cannot find a unique binary tree; as there are two or more trees with same traversal values.

- Inorder + Preorder/Postorder = Unique tree.



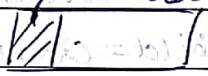
typical structure of a tree (binary)

now, Inorder



root of subtree

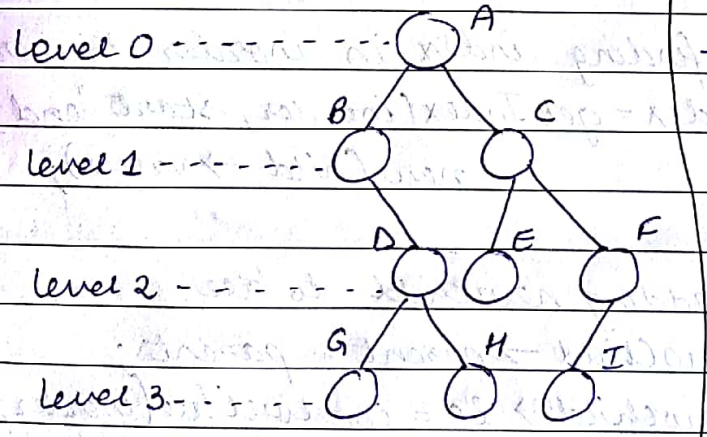
preorder



so, I know, when to exactly divide the tree.

- Similarly, Inorder + Postorder ≠ Unique tree. eg: X, Y.

↳ Level order traversal :- (BFS).



(Fig 1.1)

Levels.

- Level of root = 0
- Level of anyother node = 1 + level(parent)

preorder should be used as it visits node first, and then goes to its' children

↳ Height of a node/tree :-

height (leaf) = 0

height (parent) = max(height(both children)) + 1

↳ post order ensures that...

- Homework

Q1) Construct a binary tree with preorder and inorder. Print post order.

```
node *ConstructTree(int inorder[], int preorder[], int
                    start, int end, node * parent)
{
    // start and end variables are for inorder only
    // in preorder, root is at beginning.

    static int preorder_index = 0;
    int index;
    if (start > end) return NULL;
    node *newChild = createMemory(preorder[preorder
                    index++], parent);
    if (newChild != NULL)
    {
        if(start == end) // leaf node.
            return newChild;

        // finding index in inorder traversal
        index = getIndex(inorder, start, end,
                    newChild -> value);

        // adding newchild to tree
        newChild -> parent = parent;
        newchild -> lc = constructTree(inorder, preorder
                    start, index-1, newChild);
        newChild -> rc = constructTree(inorder, preorder
                    index+1, end, newChild)
    }
    return newChild;
}
```

```
node *createMemory(int val, node *parent)
{
node *newNode = new node;
if(newNode != NULL)
{
    newNode->lc = NULL;
    newNode->rc = NULL;
    newNode->parent = parent;
    newNode->value = val;
}
return newNode;
}

int getIndex(int inorder[], int start, int end, int key)
{
    while(start < end)
    {
        if(inorder[start] == key)
            return start;
        start++;
    }
}
```

Q1(ii) Construct binary tree with postorder and inorder. Print preorder.

→ same as previous problem.

Q2(i) Given a binary tree print its inorder, preorder and post order iteratively.

→

Preorder traversal :- (1-stack)

```
void preorder_traversal(node *root)
{
    stack <node*> s;
    node * temp=NULL;
    if (root!=NULL)
        s.push(root);
    while(!s.empty())
    {
        temp=(node*)s.top();
        cout<<temp->value<<" ";
        s.pop();
        if(temp->rc!=NULL)
            s.push(temp->rc);
        if(temp->lc!=NULL)
            s.push(temp->lc);
    }
}
```

Inorder Traversal :- (1-stack)

```
void inorder_traversal(node *root)
{
    stack <node*>s;
    node *t=root;
    while((t!=NULL) && (!s.empty()))
    {
        while(t!=NULL)
        {   s.push(t);
            t=t->lchild;
        }
```

// trying to find out the left most child

```
        t=s.top;
        s.pop();
        cout<<t->value<<"  ";
        t=t->rchild;
    }
}
```

Postorder Traversal :- (2-stacks)

```
void postorder_traversal(node *root)
{   node *temp;
    stack <node*>s1;
    stack <node*>s2;
    if(root!=NULL)
    {       s1.push(root);
            while(!s1.empty())
            {
                temp=s1.top();
                s1.pop();
                s2.push(temp);
                if(temp->lchild!=NULL)
                    s1.push(temp->lchild);
                if(temp->rchild!=NULL)
                    s1.push(temp->rchild);
            }
            while(!s2.empty())
            {
                temp=s2.top();
                cout<<temp->value<<"  ";
                s2.pop();
            }
    }
}
```

Q3. If level order traversal of a tree is given (levels only) is it possible that such a tree exist? Yes or no?
eg: from figure 1.1 → 0, 1, 1, 2, 2, 2, 3, 3, 3.

Yes, it is possible.
# of nodes in level (i+1) ≤ no. of nodes in level (i)

for all levels.
If it follows, such a tree exists.

Q4. I can also have height sequence given. Can you predict the existence of such a binary tree?

Yes, it can be predicted.
If # of nodes in level (i+1) ≤ # of nodes in level (i+1)

and

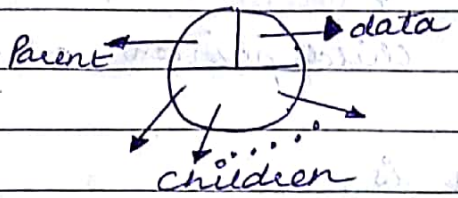
# of nodes in level (i) ≥ # of nodes in level (i+1) / 2

i.e.

$\frac{1}{2}$ nodes_level(i+1) ≤

PREPARATORY TERM (L-2)

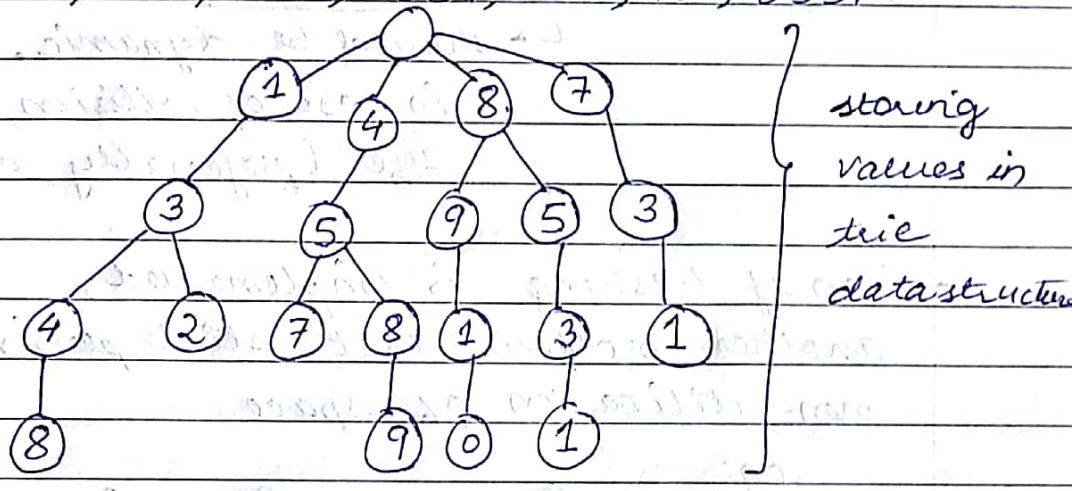
General Tree node :-



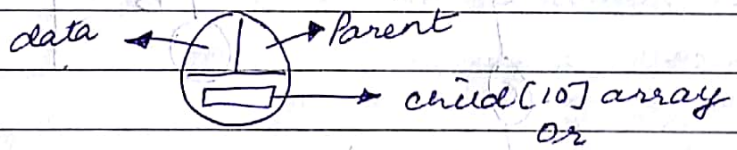
TRIE :- Also called digital tree, radesic tree, prefix tree

suppose, we want to store :-

1348, 132, 457, 4589, 8910, 731, 8531



so, for digits, the trie's "DS" would look like



eg: If I have 10^6 numbers, then maximum # of nodes?

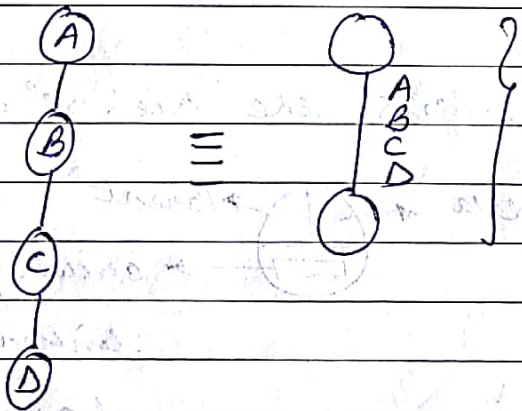
Sol: all the numbers are at the leaves, so, I should have atleast 10^6 leaf nodes
↓
exactly to be more precise.

# of non-leaf nodes can vary.
So, I require the memory for
$10^6 \times 10$ integers
↳ child pointer array/node.

- So, the problem is,
  structure itself takes too much memory.
  - → use linked list :- loss is random accessibility. sequential access is possible with Linked List.
  - → build a hash table :-
    - ↳ should be dynamic,
      in case of collision, change the size (preferably double).

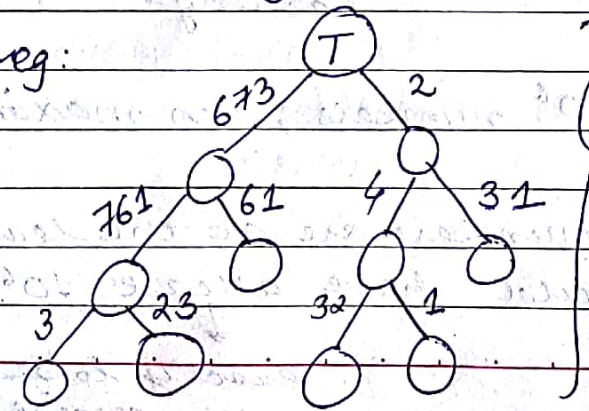
- Even if hashing is implemented, another problem that still persist is non-utilization of space.

eg:



This type of structure is found is compressed trie.

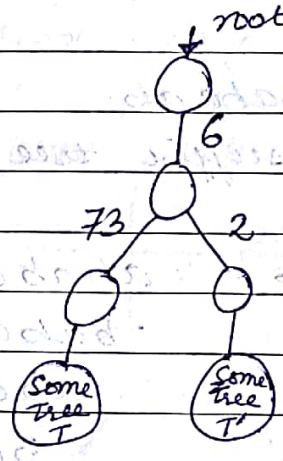
eg:



reduced # of nodes for storing all the data in compressed trie.

In the above diagram,
673$ is a string
67361 is a string.
241 is a string
231 is a string.

- Prefix free tree :-
eg: 62 and 673



eg:- root = 1 child.
every node = 2 children.
max # of nodes?
given :- # of leaves = atmost 'n'.

Sol:
⇒ # of internal nodes = atmost 'n-1'
∴ max # of nodes = n + n - 1
≤ 2n.

- SUFFIX TREES :- (PAT Tree, position tree) is a compressed trie containing all suffixes of a given text as their keys and positions in their text as their values.

→

consider a string of length 'n' :-

$X_1 X_2 \ldots\ldots X_n$

then, its suffixes are :-

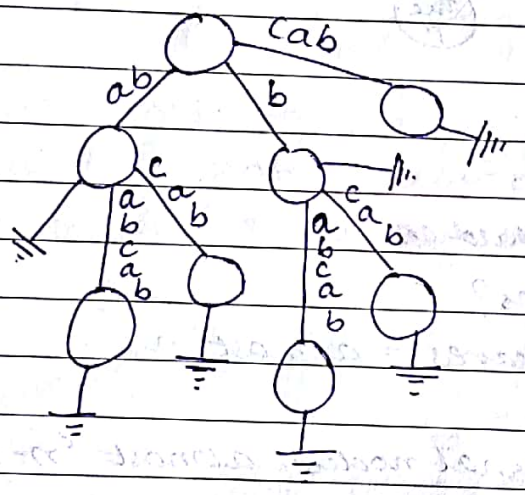
$X_1 X_2 \ldots\ldots X_n$
$X_2 \ldots\ldots X_n$
$X_3 \ldots\ldots X_n$
$\vdots$
X_n

} n- suffixes.

eg:- string ababcab.
construct suffix tree.

sol.

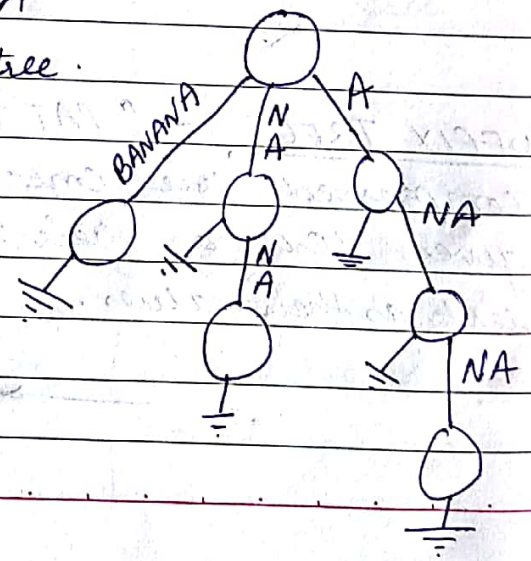
suffices => abababcab
babcab
abcab
bcab
cab
ab
b



eg: string: BANANA
construct suffix tree.

sol.

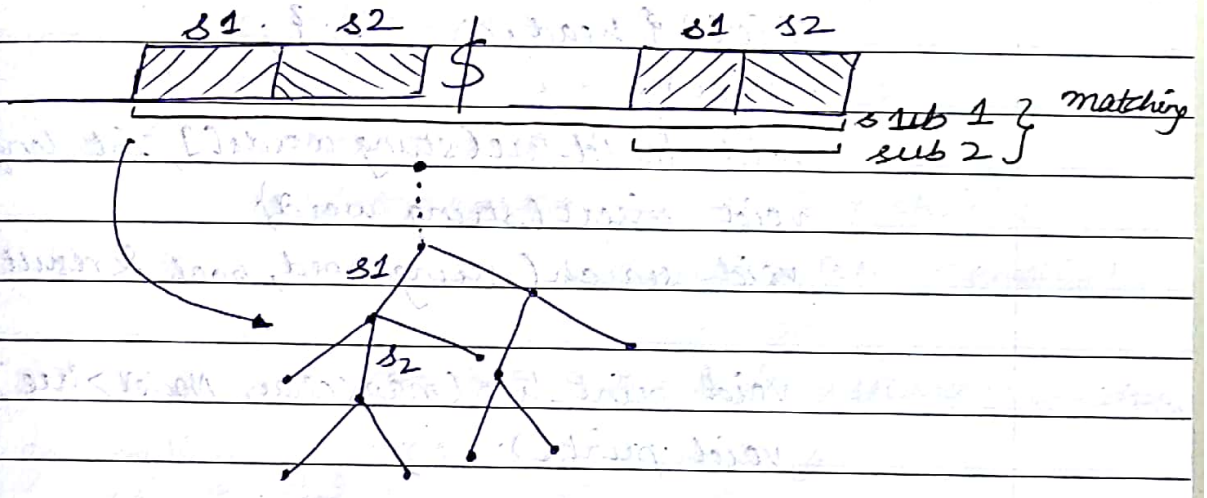
suffices:- BANANA
ANANA
NANA
ANA
NA
A



- suffix tree is a compressed trie of the given string's suffixes.

- There exist an algorithm that constructs a suffix tree in O(n).

eg:- (i) longest substrings of a given string
(ii) longest (common) subsequence → $O(n^2)$ through DP. Can be solved using suffix trees.



eg: Longest substring in a given string can also be done using suffix trees easily.

- Homework

Q1. Write a program to implement try.

Q2. Download the code for compressed tries and suffix tree construction (in O(n)) from internet, and see/analyse it's working.

```
// implementation of Trie.
#include<iostream>
#include<string>
#include<map>
```

```
#include<vector>
Using namespace std;

struct node{
    char ch;
    map<char, Node*> children;
};

class Trie{
  public:
    Trie(){head.ch = -1;};
    ~Trie();
    void build_trie(string words[], int length);
    void insert(string word);
    void search(string word, bool &result);

    void print_tree(map<char, Node*>tree);
    void print();

  protected:
    Node head;
    vector<Node*>children;
};

// memory release
Trie::~Trie(){
    for(int i=0; i<children.size(); i++)
        delete Children[i];
}
```

```
void Trie::build_Trie(string words[], int length)
{
    for(int i=0; i<length; i++)
    { insert(words[i]); }
}

void Trie::insert(string word) {
    map<char, Node*> *current_tree = &head.children;
    map<char, Node*>::iterator it;

    for(int i=0; i<word.length(); i++)
    { char ch=word[i];
      if((it=current_tree->find(ch))!=
                current_tree->end())
      {
        current_tree = &it->second->children;
        continue;
      }
      if(it == current_tree->end())
      {
        Node *new_node = new Node();
        new_node->ch = ch;
        (*current_tree)[ch] = new_node;
          // for continuous insertion
        current_tree = &new_node->children;
          // for easy m/m clean up
        children.push_back(new_node);
      }
    }
}
```

```
void Trie::search(string word, bool &result)
{
    map<char, Node*> current_tree = head.children;
    map<char, Node*>::iterator = it;

    for(int i=0; i<word.length(); i++)
    {
        if((it == current_tree.find(word[i])) ==
                current_tree.end())
        {   result = false;
            return;
        }
        current_tree = it->second->children;
    }
    result = true;
    return;
}

void Trie::print_tree(map<char, Node*> tree)
{
    if (tree.empty())
        return;
    for(map<char, Node*>::iterator it=tree.begin
        it != tree.end(); ++it) {
        cout<<it->first<<endl;
        printtree(it->second->children);
    }
}

void Trie::print() {
    map<char, Node*> current_tree = head.children;
    printtree(current_tree);
}
```

```
int main(){
    string words[] = {"mudit", "mittal", "dehradun",
                      "bengal", "bengaluru"};
    Trie trie;
    trie.build_trie(words, 5);
    cout<<"All nodes : "<<endl;
    trie.print();

    cout<<" Searching "<<endl;
    bool in_trie = false;
    trie.search("mudit", in_trie);
    cout<<" mudit : "<<in_trie;

    return 0;
}
```

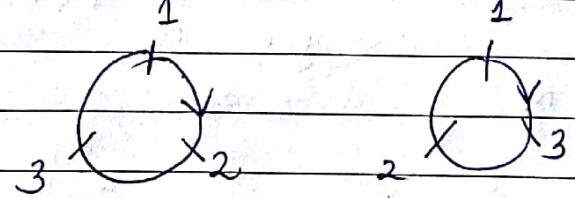
Introduction to Algorithms (L4)

Algorithm:- is a method to solve a problem by a finite sequence of instructions.
i.e., till the time, # of instr = finite, solution is fine.

- There are problems, for solution of which, no algorithm exists
  eg:- Halting Problem
  Hilbert's 10th problem.

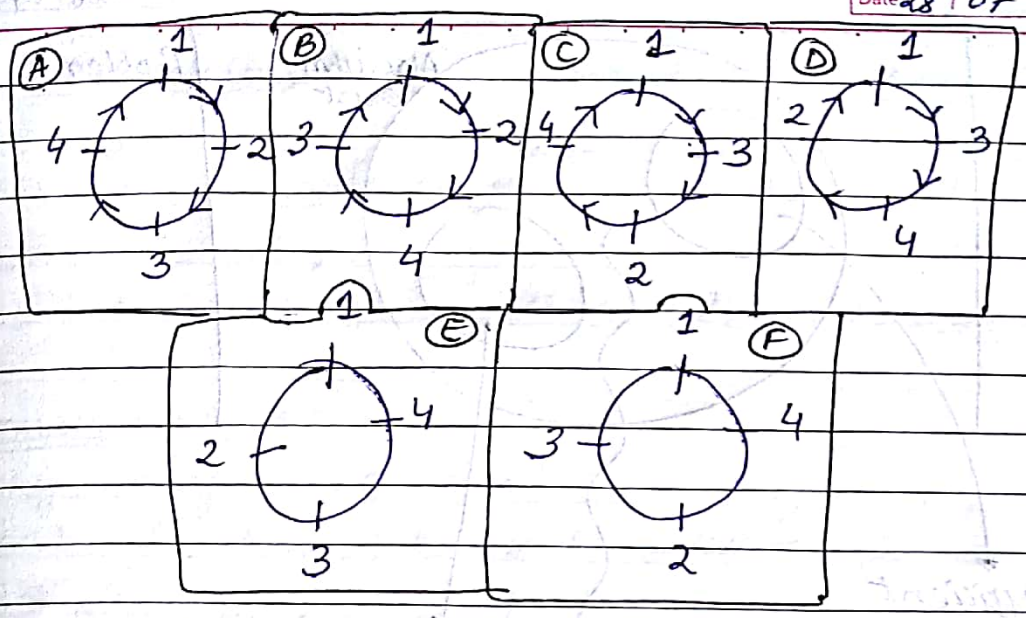
- For some solution, only complex time algorithms exists.
  eg: Travelling Salesman Problem
  or
  Chinese Postman Problem.
  ↳ we have to apply brute force method to find shortest path.
  ↓
  there will be $(n-1)!$ such enumerations.

eg: for $n=3$ and $\{1,2,3\}$
& '1' is source.



2 - possible ways.

eg: for $n=4$, and $\{1,2,3,4\}$
& '1' is source
then -



∴ for n-nodes, time complexity for best path = $O((n-1)!) = O(n!)$

eg: Hilbert's 10th problem :-

$$x^n + y^n - z^n = 0$$

for above equation, no root exist for $n > 2$.

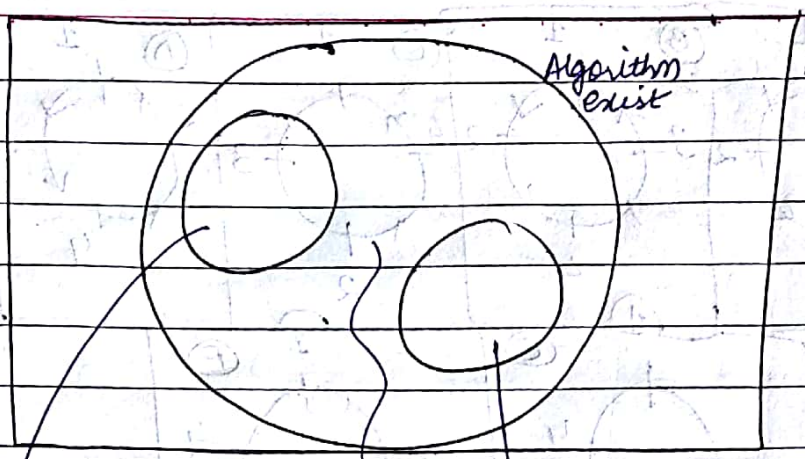
- Now, considering practical problem, a postman has to deliver 100 letters, then TSP for $n = 100$ is a practical problem.

now, for $n = 100$, bruteforce will give possible $\underline{99}$ solutions

and $\underline{99} > 1.2.3.\ldots.9.\underbrace{10.11.12.\ldots.99}_{10^{90}}$

$$\Rightarrow \underline{99} > 10^{90}$$

so, it is unsolvable!



efficient algorithms (polynomial time algorithms).

NP-Hard problems (eg: TSP)

There are some problems that are neither polynomial time solvable nor NPHard.

eg: Turn Piker's reconstruction.

$O(n^2 \log n)$ algorithm using Bruteforce / Backtracking.

- **Polynomial Reduction** :-

eg:

$p, q \in$ prime numbers.

$n = p.q$

now, find factors of n.

↳ # of steps $= \sqrt{n}$

which is a finite number given the value of n

so an algorithm exists for it.

so not NPH, but efficient sol^n.

But suddenly, $n = 2^{1000}$, then $\sqrt{n} = ?$

$$\sqrt{n} = 2^{500} \simeq 10^{150}!!!$$

suddenly this problem becomes inefficient!

- Homework

Q1. Why $n \log n$ solution in sorting is efficient, but $\sqrt{n}$ here, to find sqrt(n) is not?

If n is the input,

then size of input in bits = $\log_2 n$.

i.e. $n = 2^{\log_2 n}$,

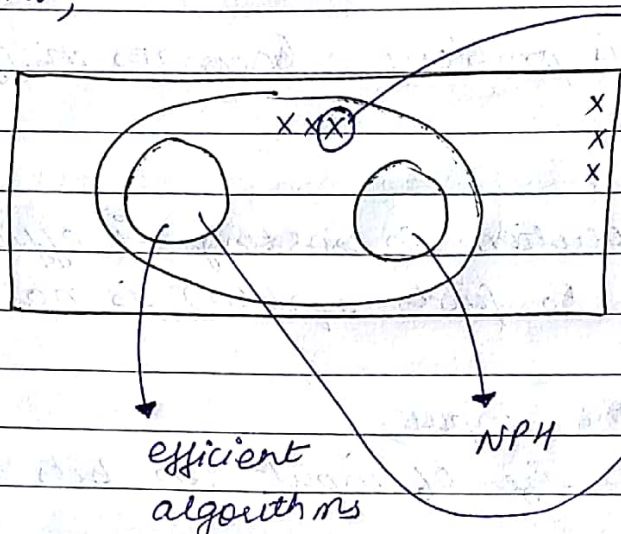
now, brute force here works in $\sqrt{n}$

$= 2^{\frac{1}{2}\log n}$!!

So, it's inefficient!

Lecture 5 (L-5)

So, consider the same graph drawn last time,



So, we can say that,
∃ algorithm, that checks primality in an efficient way, without finding a factor.

The guarantee of "efficiency" applies only if the number is non-prime, If it is prime, it takes same time as that is factorization.

- So, what metric should be used to judge an algorithm?
  - steps? (time complexity)
  - memory? (space complexity)

time complexity is dominant / standard
- Quantifiable & easy to compare
- often critical bottle neck.

eg:- find the speed of execution? (largest value of 'n' that gives output in ≤ 1 sec).

```
#include <stdio.h>
void main(){
    int i, j, k, n, s;
    n = 10000 ; s = 2;
    for(i=0; i<n; i++)
        for(j=0; j<n; j++)
            for(k=0; k<n ; k++)
                s = 2*s + 1;
    printf("%d\n", s);
}
```

after overflow, if once -1 is reached, then -2+1 = -1 always

(i) why do we get -1, after some point of time?

(ii) will this always happen?

eg:- what is the largest integer array you can use in your laptop?

```
#include <stdio.h>
int main(){
    int i, j, k, n, a[ _____ ];
    n = 1000;
    for(i=0; i<n; i++)
        for(j=0; j<n; j++)
            for(k=0; k<n; k++)
                a[i*n*n + j*n + k]
                    = 2*i*j*k + 1;
}
```

nearly 2.1×10^6

now, Benchmark for rest of the course is:-

processor speed = 10^9 instr/sec (approx)

max memory = 2.1×10^6 integer array

- Super computers are about 10^6 times faster.
  speed = 10^{15} } approx.
  memory = 10^{12} }

- Asymptotic Notations :-

(1) Big-Oh :- Bounded above by
$T(n) = O(f(n))$
for some c and N_0,
such that
$T(n) \leq c \cdot f(n)$
whenever $n > N_0$.

eg :- $T(n) = 4n^2 + 2n - 3$
$T(n) \leq 4n^2 + 2n$
$\forall n > 1$
$T(n) \leq 4n^2 + 2n^2$
$T(n) \leq 6n^2$
$\Rightarrow c = 6, N_0 = 1$
$T(n) = O(n^2)$.

(2) Big-Omega :- Bounded below by
$T(n) = \Omega(f(n))$
for some $c > 0$, N_0
such that
$T(n) \geq c \cdot f(n)$
whenever $n > N_0$.

eg :- $T(n) = 4n^2 + 2n - 3$
$T(n) \geq 4n^2$, such that $2n - 3 \geq 0$
$n \geq 3/2$
$\Rightarrow n \geq 2$
$\Rightarrow T(n) = \Omega(n^2)$
$n_0 = 2, \quad c = 4$.

now, $T(n) = \Omega(n^2)$

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

$$\boxed{4n^2 \leq T(n) \leq 6n^2, \quad \forall n > 2}$$

(3) Big Theta :-

$T(n) = \Theta(f(n))$

for above example

eg: Prove that; a polynomial of degree 'k' is $O(n^k)$ [as long as coefficient of highest degree is positive].

sol.

suppose, $f(n) = b_k n^k + b_{k-1} n^{k-1} + \dots + b_2 n^2 + b_1 n + b_0$.

Let $a_i = |b_i|$

$\therefore f(n) \leq a_k n^k + a_{k-1} n^{k-1} + \dots + a_2 n^2 + a_1 n + a_0$

$$f(n) \leq n^k \sum_{i=0}^{k} a_i \frac{n^i}{n^k} \leq n^k \sum_{i=0}^{k} a_i \leq c \cdot n^k$$

where $c = \sum_{i=0}^{k} |b_i|$.

$\Rightarrow f(n) \leq c \cdot n^k \Rightarrow \underline{f(n) = O(n^k)}$.

eg: $f(n) = 10n^3 + n^2 + 40n + 800$ for $n = 1000$.

error = 0.01% if we drop all but n^3 terms

↳ negligible.

Lecture-6

Q. Write an efficient algorithm for $f(n) = \begin{cases} 0 & n=0 \\ 1 & n=1 \\ (f(n-1)+f(n-2))\%100 & \text{otherwise} \end{cases}$

(a) what is the largest value of 'n' for which your laptop computes this in 1-sec?

(b) what is time & space complexity?

(i) **Recursion** :-

```
int f(int n)
{
    if(n<2)
        return n;
    return (f(n-1)+f(n-2))%100;
}
```

sol :- Largest value of $n = 38$

Time Complexity :-

I'm considering only additions

$$\begin{cases} T(n) = T(n-1)+T(n-2)+1 \\ T(2) = 1 \\ T(1) = 0 \\ T(0) = 0 \end{cases}$$

$T(n) \leq T(n-1) + T(n-1)$

$T(n) \leq 2T(n-1)$

so, $T(n-1) \leq 2T(n-2)$

$\Rightarrow T(n) \leq 2^2 . T(n-2)$

$T(n) \leq 2^3 . T(n-3)$

$\vdots$

$T(n) \leq 2^k T(n-k)$

$\vdots$

$T(n) \leq 2^{n-2} . T(n-2)$

$T(n) \leq 2^{n-2} . 1$

2^{n-2} is # of additions I did.

$\Rightarrow T(n) = O(2^{n-2}) = O\left(\frac{2^n}{4}\right)$

$= O(2^n)$

now, again,

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

$T(n) \geq T(n-1) + T(n-2)$

$T(n) \geq T(n-2) + T(n-2)$

since $T(n-1) = T(n-2) + \text{positive constant}$

$\therefore T(n-1) \geq T(n-2)$

$T(n) \geq 2 \cdot T(n-2)$

$T(n) \geq 2^2 \cdot T(n-4)$

$\vdots$

$T(n) \geq 2^k \cdot T(n-2k)$

$\vdots$

no segmentation - fault, bad algo. because of its time complexity

$T(2) = 1$

so, $n - 2k = 2$

$k = \frac{n-2}{2}$

$\therefore$

$T(n) \geq 2^{\frac{n-2}{2}} \cdot T(2) = 2^{\frac{n-2}{2}}$

$T(n) \geq 2 \cdot 2^{n/2}$

$\Rightarrow T(n) = \Omega(2^{n/2})$

so, $2 \cdot 2^{n/2} \leq T(n) \leq 2^n/4$

Space complexity:- Length of longest path in f^n call tree.

$S(n) = \max(S(n-1), S(n-2)) + 1$

f^n calls are not happening concurrently.

$S(n) = S(n-1) + 1$

$\Rightarrow S(n) = \Theta(n)$.

(ii) Dynamic Programming :-

```
int dp(int n)
{  int i;
   f[0]=0;
   f[1]=1;
   for(i=2; i<=n; i++)
      f[n]=(f[n-1]+f[n-2])%100;
}return f[n]
```

Sol. Largest value of 'n' = 2.1×10^6

→ Time Complexity :-

T(n) = # of additions
= n - 1
⇒ T(n) = O(n).

→ Space Complexity :-

S(n) = n-size array
= O(n).

There is a segmentation fault on execution of this code, due to non-contiguous availability of memory (for n-sized array).
This also is not upto the mark because of its space complexity !

(iii) Linear algorithm / iterative :-

```
int fibo(int n)
{  int a=0, b=1, c, i;
   for(i=2; i<=n; i++)
   {   c=(a+b)%100;
       a=b;
       b=c;
   }
   return c;
}
```

Sol. Largest value of $n = 3 \times 10^7$
(answer in ≤ 1 sec)

→ Time Complexity :- # of additions = $\theta(n)$

→ Space complexity: only for 3 variables a, b, c in all cases, $\therefore S(n) = \theta(1)$

Now, the problem arises, for the same problem if $N \simeq 10^{1000}$.

↳ possibly we have only $O(\log^3 N)$ time and $O(\log^2 N)$ space complexity algorithm.

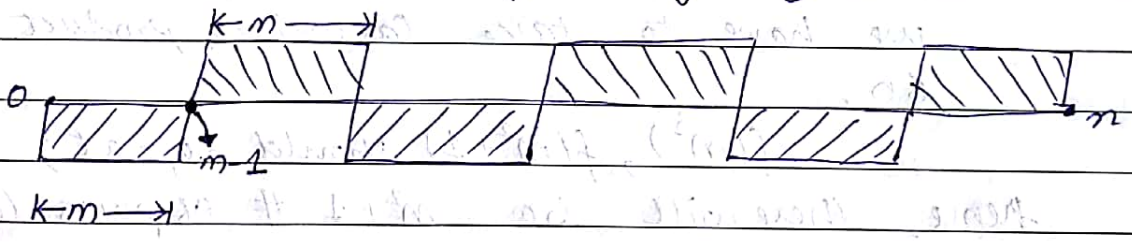
i.e. $\log 10^{1000} = 1000 \log 10 \simeq 1000$

so, $\log n$ or $\log^2 n$ algorithm is fine.

- Now, N^{th} term of fibonacci $= \left[\left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1-\sqrt{5}}{2}\right)^n\right]^{max}$

which is exponential algorithm.

- Since, $f(n)$ is a modulo function, thus, there exists a ~~a~~ periodicity say of 'm'.

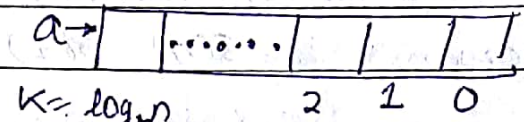


for the above example, $F(300) = 0$, $F(301) = 1$, thus there is a periodicity of 300

↓

$f(n)$ is a f" of previous two f's. and at '300', both 0 & 1 are repeating.

- storing $n = 10^{1000}$



$k = \log_{10} n$

$$\sum_{i=0}^{k} a_i \times 10^i$$

- Now, by piegonhole principle,

pigeons → A, holes → B

$f: A \longrightarrow B$

and $|A| > |B|$

so, atleast one hole (here value of 'f') shares a common hole (here storage value).

Thus, by Set theory,
if we want any Ordered pair $(f(0), f(1))$
we have to take Cartesian product
so,
$f(m^2), f(m^2+1)$ should repeat;
hence, there will be m^2+1 # of pairs (at max)

- generally repetition is at about $6m$ (in most of the cases).

$f: N \longrightarrow Z_n$

$NXN \longrightarrow Z_n X Z_n \quad (m^2)$

- Assignment

1. n = 10000 digit number.
Find 'm' and then find $f(n)$ where it is in modulo 'm'.

20) :-

```
#include<stdio.h>
#include<stdlib.h>
void input_m(int *m)
{   printf("Enter modulo value m:");
    scanf("%d",m);
}
void input_n(int *n, int k)
{   char *c;
    *n=0;
    FILE *f= fopen("input.txt","r");
    if(f!=NULL)
    {   while((c=fgetc(f))!=EOF)
            *n=((*n)*10+(c-'0'))%k;
        fclose(f);
    }
}
void fill_arr(int **arr, int m)
{
    int i;
    *arr=(int *)calloc(m*m+1, sizeof(int));
    (*arr)[0]=0;
    (*arr)[1]=1;
    for(i=2; i<=m*m; i++)
        (*arr)[i]=((*arr)[i-1]+(*arr)[i-2])%m;
}
void find_k(int m, int *arr, int *k)
{
    for(int i=2; i<m*m; i++)
        if(arr[i]==0 && arr[i+1]==1)
            break;
    *k=i;
}
```

```
int main()
{
    int m, n, k, *arr;
    input_M(&m);
    fillArr(&arr, m);
    find_k(m, arr, &k);
    input_n(&n, k);
    printf("%d\n", arr[n]);
    return 0;
}
```

Lecture - 7

- Considering previous solutions.

algorithm #	Time	Space	
1	$\Omega(2^{n/2})$	$\Theta(n)$	$\rightarrow 60$
2	$\Theta(n)$	$\Theta(n)$	$\rightarrow 10^6$
3	$\Theta(n)$	$\Theta(1)$	$\rightarrow 10^9$
4	$\Theta(\log n)$	$\Theta(m)$	$\rightarrow 10^{10^3}$

m is independent of 'n', we need to get rid of this.

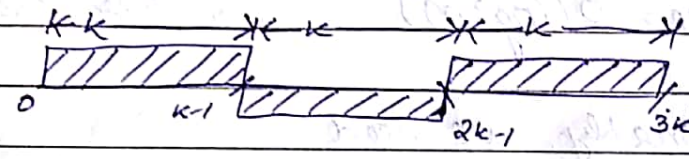
BigInteger class in Java

for algorithm 4,

↳ the sequence should be a periodic sequence only.

↳ Time = $O(\log^3 n)$ is fine (around 10^7 computations)

↳ Space = $O(\log^2 n)$ is fine.



periodic sequence

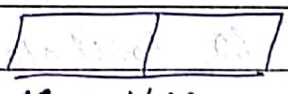
Here, we assume

- $O(n)$ space
- $O(k)$ time
- $F(n \% k) = O(\log n)$ time.

1. $F(0)$ & $F(1)$ repeats
2. Optimally, repetition occurs $< 6m$ (empirical result)

eg: $m > 10^9$ or $m > 10^{12}$, then how can you store 'm' to avoid integer overflow?

$m^2 = 10^{18}$



Consider digits on base 10^9.

But, here in this problem, we want to get rid of 'm'.

$k = 10^{10}$

$m = 10^{12}$

find $F(n\%k)$

- Solution 5 :- declare an array arr[10^6].
  - (a) copy 10^6 elements to array
  - (b) write them to an external file
  - (c) repeat until values repeat.

File handeling = I/O operation and is very costly. Copying 10^6 elements together enhances efficiency or not?

But, in this algorithm 5,

time = $O(n)$ $\Rightarrow 10^3$ seconds ✗

Space = $O(\log^3 m)$ (undesired!)

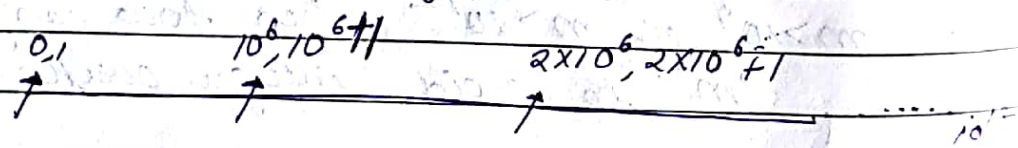
But optimally, we need

time $\leq O(\log^3 n)$

space $\leq O(\log^2 n)$

- another variant to it is :-

store any two numbers (consecutive) to a file, at some gap, say 10^6.



So, total elements to be stored

$= \frac{10^{12}}{10^6} \times 2 = 10^6 \times 2$ elements

but still, too much I/O and too much execution of this inefficient function!

- Algorithm 6 : (Optimal)

$$\begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F(n-1) \\ F(n-2) \end{bmatrix}$$

modulo 'm' multiplication

(m)

Homework :- Search, from where this came.

verify :- $\begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}}_{A} \begin{bmatrix} F(n-1) \\ F(n-2) \end{bmatrix}$

$$\begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix} = \begin{bmatrix} F(n-1) + F(n-2) \\ F(n-1) \end{bmatrix}$$

now, $\begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix} = A \cdot \begin{bmatrix} F(n-1) \\ F(n-2) \end{bmatrix}$

$$\begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix} = A^2 \cdot \begin{bmatrix} F(n-2) \\ F(n-3) \end{bmatrix}$$

$$\vdots$$

$$\begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix} = A^K \cdot \begin{bmatrix} F(n-K) \\ F(n-K-1) \end{bmatrix}$$

$$\vdots$$

$$\begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix} = A^{n-1} \begin{bmatrix} F(1) \\ F(0) \end{bmatrix}$$

Let $A^{n-1} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ then $\begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix} = \begin{bmatrix} a \\ c \end{bmatrix}$

Camlin Page 043
Date 09/08/18

Now, consider power function for an integer.
eg: x^n.

traditional algorithm :-

```
y = x
i = 2 to n
y = y*x
```

} O(n) algorithm!

more efficient way :-

```
Power(x, n)
{ if(n==0) return 1;
  if(n%2==0) Power(x^2, n/2);
  else Power(x^2, n/2)*x;
}
```

i^{th} bit

- even : if $n = 2k$, $n/2 = k$.
  $(x^2)^k = x^{2k} = x^n$
- odd : $n = 2k+1$; $k = n/2$
  $(x^2)^k . x = x^{2k} . x = x^n$

Considering multiplication operations :-

$$T(n) = \begin{cases} T(n/2) + 1 & n = \text{even} \\ T(n/2) + 2 & n = \text{odd} \end{cases}$$

Computing Time Complexity :-

$T(n) \leq T(n/2) + 2$ in worst case,
when $n = 2^k - 1$
(all 1's)

$$T(n) \leq 2 + 2 + T\left(\frac{n}{2.2}\right)$$

$$\Rightarrow T(n) \leq 2k + T\left(\frac{n}{2^k}\right)$$

now, $T(0) = 1$ (given)

so, $\frac{n}{2^k} < 1 \Rightarrow n < 2^k$

$\Rightarrow \log_2 n < k$

$\therefore T(n) \leq 2\log_2 n + T(1)$

$T(n) \leq 2\log_2 n + 1 \Rightarrow \underline{T(n) = O(\log_2 n)}$.

again,

$T(n) \geq T\left(\frac{n}{2}\right) + 1$ if $n = 2^k$ (best case)

$T(n) \geq 1 + 1 + T\left(\frac{n}{2.2}\right)$

$\Rightarrow T(n) \geq k + T\left(\frac{n}{2^k}\right)$ again $n < 2^k$

$\Rightarrow \log_2 n < k$.

$T(n) \geq \log_2 n + T(1)$

$T(n) \geq 1 + \log_2 n \Rightarrow \underline{T(n) = \Omega(\log_2 n)}$.

$\Rightarrow \underline{T(n) = \Theta(\log_2 n)}$.

But, recursive codes are slower in execution, and occupy more, so, converting it to iterative code :-

```
Power(x,n)
{ y=1;
  while(n>0)
  { if(n%2==1)
      y = y*x;
    x = x*x;
    n = n/2;
  }
}
```

check $n=13$

n	x	y
13	x	y
6	x	1
3	x^2	x
1	x^4	x
0	x^8	x^5
	x^{16}	x^{13}

so, $\log(n) \leq T(n) \leq 2\log(n)$

now, (given, n is bits)

$$n = \sum_{i=0}^{k} n_i \cdot 2^i \quad , k = \log_2 n \quad n_i = \{0, 1\}$$

$$x^n = x^{\sum_{i=0}^{k} n_i \cdot 2^i} \quad (\Sigma \text{ is summation})$$

$$= \prod_{i=0}^{k} x^{n_i \cdot 2^i} \quad (\Pi \text{ is product})$$

n_i → 0 → $x^{2^i \cdot n_i} = 0$

n_i → 1 → $x^{2^i \cdot n_i} = x^{2^i}$

Now, $A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$

N = [ ] Integer array of length 1000 digits.

N_b = [ ] 3100 binary digits

so, Power(A, N, k) → # of bits in N

{

i = 0;

$y = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

while (i < k) {

if (N[i]) y = MatrixMultiply(y, A)

X = MatrixMultiply(A, A)

i = i + 1;

}

return y; }

for a 2×2 matrix, # of multiplication
{ = 16 if that bit is odd.
{ = 8 if that bit is even.

matrix multiply has this type of structure

```
for(i)
  for(j)
    for(k)
      C[i,j] = (C[i,j] + A[i,k].B[j,i])%
```

inefficient way for A^{10}.

we can use binary mechanism, so, for A^{10}, can be done in atleast '4' function calls to multiply.

This, for binary representation,

time = O(log n)
space = O(log n)

Assignment

1. Implement code for algorithm 6, n is in binary
2. what if n is in decimal digits? Implement. for, n = 3200 bits for part 1, and 10^6 digits in decimals.

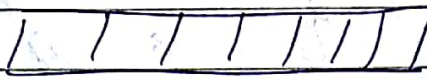
$m = 10^{12}$.

N digits → O(log²n)
N binary → O(log n)
solution

$T(n) = O(\log^2 n)$ since it is dominating.

if $n = 10^{10^6}$,

then,

$n =$  ← 10^6 element array.

Last bit is even/odd = O(1)

one division = O(log n)

for log n bits,

divisions = $\log^2 n$ ← 10^3 seconds!

can I do :-

$y = y \cdot A^{N[i]}$

$A = A^{10}$, 10 because decimal is the base

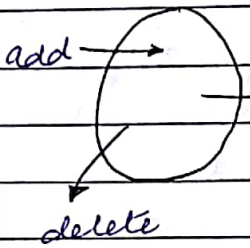
$n = \sum n_i \cdot 10^i \quad n_i \in \{0, 1, \ldots 8, 9\}$

$X^n = X^{\sum n_i 10^i} = \prod X^{n_i \cdot 10^i}$

Think over it and implement!

LECTURE-8

- Coming back again to dynamic dataset problem.

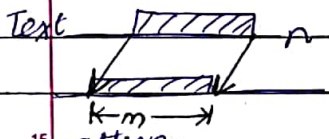


	Hashing		Balanced BST
add	O(1)	O(n)	O(logn)
delete	O(1)	O(n)	O(logn)
Seqach	O(1)	O(n)	O(logn)

average case (randomization)

worst case

- Recap:



→ O(n+m)

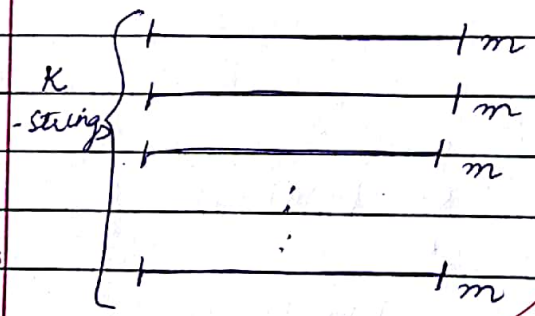
for input text

for hash value of pattern.

and worst case = O(nm)

(Hash value is same for all values)

- Now, consider, we have k such strings:-



then, we've to repeat previous algorithm k-times

T(n) = O(k(n+m))

⇒ $n = 10^6$,

$m = 10^4$

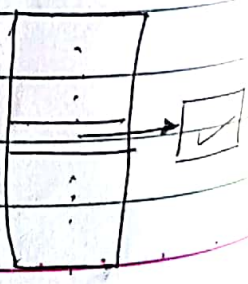
$k = 10^4$

substring length (m)

(Hash value)

Hash value for each pattern

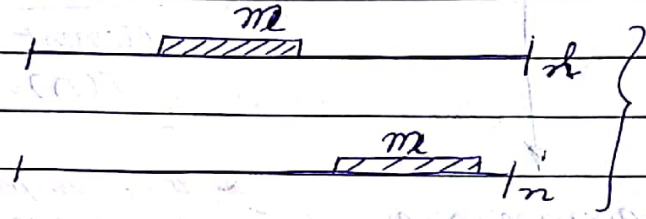
(✓) this one is better:



$\Rightarrow T(n) = \Theta(mK + n)$

K → total # of hash values.

Or, consider :-



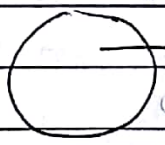
consider a fixed length (m) substring of both the strings

$\Rightarrow T(n) = \Theta(m) + \Theta(n) = \underline{\Theta(n)}$.

- The longest substring of length 'L' in both can be found in $\Theta(n \log n)$ (average case).

For above operations I'll use hashing. Except Hashing, I can also use Balanced Binary Search Trees (BBSTs)
eg: RedBlack Trees, AVL Trees.

eg: Data base query (range operation)
$= \Theta(\log n)$ using BBST
$\Theta(n)$ using hashing.

- Static Data Set :-  search.

Linear algorithm :-

n-comparisons ←

```
Traverse the list from i = 1, ——, n :
    if A_i = x return true
return false.
```

- Linear Search

$T(0 < i \leq n)$

False, i.e. key not present
$T(n) = \Theta(n)$

Worst case (last element in list)
$T(n) = \Theta(n)$

average case

$$T(n) = \sum_{i=1}^{n} \frac{1}{n} \cdot i$$

i → # of comparisons for a particular i

$\frac{1}{n}$ → probability of picking anyone # at random

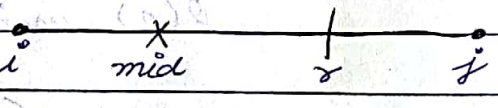
$$T(n) = \frac{1}{n} \cdot \frac{(n+1)n}{2}$$

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

$$T(n) = \Theta(n)$$

- Binary search : (default: consider sorted increasing sequence).

```
BinarySearch (A, i, j, x)
while (i<j)
{  mid = (i+j)/2;
   if (A[mid] == x)
      return true;
   if (A[mid] > x)
      j = mid-1
   else i = mid+1;
}
return false.
```



Comparisons (middle element and greater)

- Worst case analysis :- $T(n) = 2 + T\left(\frac{n}{2}\right)$

$T(n) \leq 2k + T\left(\frac{n}{2^k}\right)$

$\Rightarrow T(n) \leq 2\log n + T\left(\frac{n}{n}\right)$

$T(n) \leq 2\log n + 1$

such that

$\frac{n}{2^k} \leq 1$

$n \leq 2^k$

$\log n \leq k$

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

— Biased Binary Search :-

eg: Give 10^9 #s, i.e. 2^{30} numbers.

↳ Normal Binary Search will take 30 searches.

↳ However Biased Binary Search takes atmost "8" search (real life dictionary search)

Thus, sorting is a very important solution to many problems.

- sort $\theta(n\log n)$
Binary Search $\theta(\log n)$

} for search, this method should be used when we've to search K-times, and $K < \log_2 n$.

↳ sort + $\theta(n\log n + K\log n)$ (binary search)

if $K \not< \log n$, i.e. $K \geq \log n$,
linear search is better.

Ques If K is not known prehand, and we've n-numbers then what to do?

↳ first few iterations → linear search.

↳ start Binary search later.

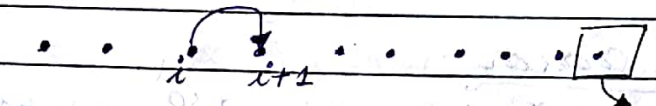
change data set. (but how?)

Hashing here would do this task in $O(n+k)$ (on average).

- Sorting Techniques:-

→ Bubble sort, → Selection sort, → Insertion sort } $O(n^2)$	→ merge sort, → Rand. Quick sort } $O(n \log n)$
	→ Deterministic Quick sort } $O(n \log n)$
	→ Heap sort

(1) Bubble sort :- $(a_i \leq a_{i+1})$



after i^{th} iteration i-elements are sorted. (invariant)

$a_i \leq a_{i+1}$

```
for j = 1 to n-1
    for i = 0 to n-j-1
        if(aj > aj+1)
            swap(aj, aj+1)
```

- $T(n)$ = # of comparisons

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

$j=1 \quad j=2 \quad \cdots\cdots$

$T(n) = \frac{n(n-1)}{2} \Rightarrow \underline{T(n) = O(n^2)}$.

- $T(n) = \frac{n \cdot (n-1)}{2} \leq \frac{n \cdot n}{2} \leq \frac{n^2}{2}$

$\Rightarrow \underline{T(n) = O(n^2)}$.

$[(n-1) \geq n/2]$

- $T(n) = \frac{n(n-1)}{2} \geq \frac{n}{2} \cdot \frac{n}{2} \geq \frac{n^2}{4} \Rightarrow \underline{T(n) = \Omega(n^2)}$

$\forall n \geq 2$.

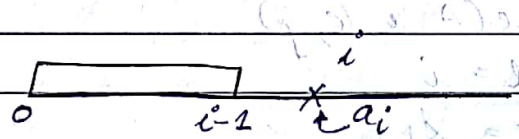
- # of swaps in worst case := $\Theta(n^2)$.
- Adding a swap variable "count" would impove best case, but we want to impove average and worst case.

(2) Selection Sort :-

```
for j = 1 to n-1                          → n-1-j is inclusive
    max = 0
    for i = 1 to (n-1-j)
        if a_i > a_max : max = i
    if a_{i+1} < a_max : swap(a_{i+1}, a_max)
```

- # of swaps in worst case = $\Theta(n)$
- $T(n) = \Theta(n^2)$

(3) Insertion Sort :-



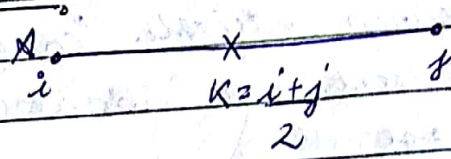

```
                                          → n-1 is inclusive
for i = 1 to (n-1)
    t = a_i , j = i-1
    while ( j >= 0 and a_j < t)
        a_{j+1} = a_j
        j--
    a_{j+1} = t
```

- $T(n) \leq T(n-1) + n-1$

$T(n) = \frac{n(n-1)}{2} := \Theta(n^2)$ # of comparisons

DIVIDE & CONQUER (2-9)

- eg:- Binary Search :-

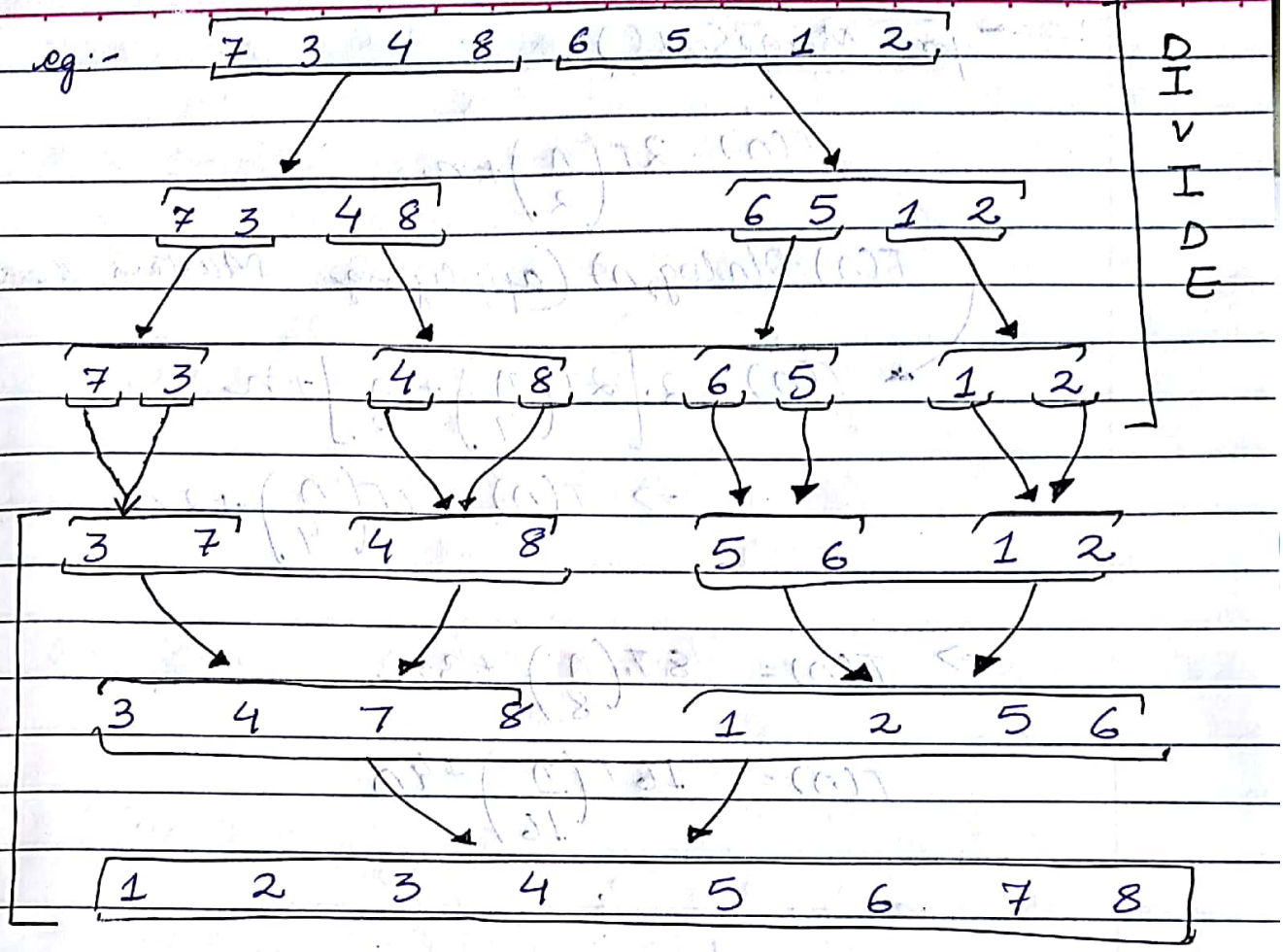


- MERGE SORT

```
Merge sort (A, i, j)
{
  if (i < j)
  { k = (i+j)/2
    Merge Sort (A, i, k)
    Merge sort (A, k+1, j)
    Merge (A, i, k, j)
  }
}

Merge (A, i, k, j)
{ l = i
  r = k+1
  a = 0
  T[0, ..., n]
  while (l ≤ k && r ≤ j)
  { if (A[l] ≤ A[r])
      T[a++] = A[l++]
    else T[a++] = A[r++];
  }
  while (l ≤ k)   T[a++] = A[l++]
  while (r ≤ j)   T[a++] = A[r++]
  l = i, a = 0;
  while (l < j)
    A[l++] = T[a++];
}
```

eg:-



- disadvantage :-

$S(n) = \theta(n)$, i.e. $\theta(n)$ space is required.

- Homework :-

Do we really need extra space?

↳ In case we don't do that, $T(n) = \theta(n^2)$?

↳ Time Complexity :-

$T(n)$ = # of comparisons that we do in worst case.

↳ $T(n) = \theta(n)$ for merge function as it performs two traversals for the given set of numbers.

→ For MergeSort();

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

$T(n) = \theta(n \log_2 n)$ (applying Master's Theorem)

$$T(n) = 2.\left[2T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n$$

$$\Rightarrow T(n) = 4T\left(\frac{n}{4}\right) + 2n$$

$$\Rightarrow T(n) = 8T\left(\frac{n}{8}\right) + 3n$$

$$T(n) = 16T\left(\frac{n}{16}\right) + 4n$$

$$\vdots$$

$$T(n) = 2^k . T\left(\frac{n}{2^k}\right) + kn$$

now, $\frac{n}{2^k} \simeq 1$

$\Rightarrow n \simeq 2^k$

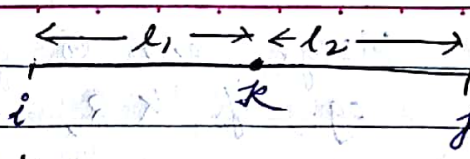
$\log_2 n \simeq k$

$\Rightarrow T(n) = n.1 + \log_2 n . n$

$T(n) = n \log_2 n + n$

$\Rightarrow \underline{T(n) = \theta(n \log_2 n)}$.

Now, in worst case,



# of comparisons
$= l_1 + l_2 - 1$
$= n - 1 \approx \underline{\theta(n)}$.

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n$$
$$\Rightarrow T(n) \leq \underline{O(n \log_2 n)}.$$

In, best case, # of comparisons (least # of comparisons) $= \frac{n}{2}$.

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

l_1 finished, l_2 is left

$$\Rightarrow T(n) = \underline{\Omega(n \log_2 n)}.$$

In many books, we'll find that

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

conquer happens in linear time.

- Homework.

Q. If we avoid temp. array, will we be still able to merge it in linear time.

- Now, suppose we've k-lists

S.no.	# of elements
1	n_1
2	n_2
3	n_3
:	:
n_k	n_k

$\sum_{i=1}^{k} n_i = n$.

all lists are sorted.

Merge them to get a sorted sequence.
eg: if $k=2$, then # of operations
$= n_1 + n_2 - 1 = \underline{O(n)}$.

Consider worst case :-

$(((n_1, n_2)\, n_3)\, n_4)\, n_5 \ldots$

n_1 : $n/2$ elements
n_2 : 1 element
n_3 : 1 element
n_4 : 1 element
n_5 : 1 element

$\therefore O[n.(k-1)] = \underline{O(nk)}$.

Otherwise also,

$(((n_1, n_2)\, n_3)\, n_4)\, n_5 \ldots$

$O(n)$

$O(n)$

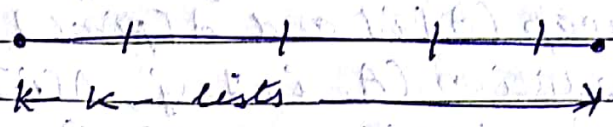
$O(n)$ $\Rightarrow \underline{O(nk)}$ overall.

Best way :- Consider two files / arrays at a time.

$n_1, n_2 \rightarrow n_{12}$
$n_3, n_4 \rightarrow n_{34}$
$n_{12}, n_{34} \rightarrow n_{1234}$
$n_5, n_6 \rightarrow n_{56}$
n_{5678}
n_7
$n_k \rightarrow n_{k+1/k}$

$\log_2 k$ $\Rightarrow n \log k$

So, for merge sort, if we divide the array into k-lists, instead of two, then recurrence relation is:-



$$T(n) = k \cdot T\left(\frac{n}{k}\right) + \theta(n \log k)$$

$$\vdots$$

after l iterations:

$$T(n) = k^l \cdot T\left(\frac{n}{k^l}\right) + \theta[l(n \log k)]$$

$$n \simeq k^l$$

$$\log_k n \simeq l$$

$$T(n) = n \cdot 1 + \theta\left[\log_k n \cdot n \cdot \log_k \overset{1}{\cancel{k}}\right]$$

$$T(n) = n + \theta(n \log_k n)$$

$$\Rightarrow T(n) = n + \theta\left(\frac{n \log n}{\log k}\right) \Rightarrow T(n) = \underline{\theta(n \log n)}.$$

But the value of constants will be very high.

QUICK SORT

(1) Randomized Quick Sort:-

↳ Pivot selected at random

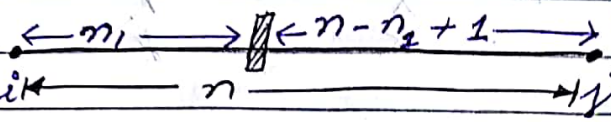
```
Randomized Quick Sort ( A, i, j)
{ if (i<j)
  {   pivot = random (i, j) ; // i + rand (i+i-1)
      swap [A[i] and A[pivot]
      K = partition (A, i+1, j, A[i])
      swap (A[i] and A[K])
      Randomized Quick Sort (A, i, k-1)
      Randomized Quick Sort (A, K+1, j)
  }
}

Partition (A, i, j, pivot)
{ l = i;
  r = j ;
  while ( l ≤ r)
  {
  while (A[l] ≤ pivot and l ≤ r)
          l++;
  while ( A[r] > pivot and l ≤ r)
          r--;
  if ( l < r)
    { swap [A[l] and A[r])
      l++; r--;
    }
  }
  return l-1;
}
```

MCQ.
return value
- l
- ✓ r } both are same
- ✓ l-1
- r+1

$T(n) = \Theta(n)$ (# of comparisons)

Time Complexity Analysis:- (randomized partition i/p)



whether $i < j$?

$$T(n) = T(n_1) + T(n - n_1 + 1) + n + 1$$

n ← partition.

In worst case, $n_1 = n - 1$

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

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

$$\Rightarrow T(n) = \Theta(n^2)$$

In average (best) case:- $n_1 = n/2$

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

$$\Rightarrow T(n) = \Theta(n \log n)$$

Also,

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

n ← partition

$\Theta(n)$ ← good pivot find

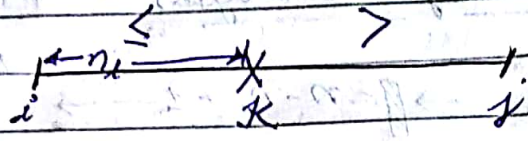
$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

$$\Rightarrow T(n) = \Theta(n \log n)$$

← average complexity of RQS.

LECTURE - 10

- Recap :-



RQS(i, K-1)
RQS(K+1, j) where $n = j - i + 1$

↳ Time Complexity analysis :-

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

where $0 \le n_l \le n-1$

now, since probability of selecting any one number out of 'n' numbers is 1/n, (randomized partitioning) so, on average,

$$T(n) = \sum_{n_l=0}^{n-1} \frac{1}{n} \left[T(n_l) + T(n-1+n_l) \right] + n + 1$$

Expected value — probability — n+1 such observations

n_l	$T(n_l)$	$T(n-1+n_l)$
0	T(0)	T(n-1)
1	T(1)	T(n-2)
2	T(2)	T(n-3)
⋮	⋮	⋮
n-1	T(n-1)	T(0)

So, $T(n) = (n+1) + \frac{2}{n} \sum_{n_\ell=0}^{n-1} T(n_\ell)$

(n+1) — independent of n_ℓ

n_ℓ is increasing.
$n-1-n_\ell$ is decreasing

$$\Rightarrow T(n) = (n+1) + \frac{2}{n} \sum_{n_\ell=1}^{n-1} T(n_\ell)$$

last term removed

$$T(n) = (n+1) + \frac{2}{n} \sum_{i=1}^{n-2} T(i) \quad \text{——— (1)}$$

$+ \frac{2}{n} T(n-1)$ — last term

multiply eqn (1) by n

$n - (2) \rightarrow$ last term removed

$$n.T(n) = n(n+1) + 2 \sum_{i=1}^{n-2} T(i) + 2T(n-1)$$

(2) last term

now, in eqn (2),

$$(n-1)T(n-1) = n(n-1) + 2 \sum_{i=0}^{n-2} T(i)$$

$$\therefore nT(n) = n(n+1) + (n-1).T(n-1) - n(n-1) + 2T(n-1)$$

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

$$nT(n) = 2n + (n+1).T(n-1)$$

divide by $n.(n+1)$

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

Or, $\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2}{n+1}$, $\forall n > 1$

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

$\vdots$

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

Now,

$\frac{T(n-2)}{n-1} = \frac{T(n-3)}{n-2} + \frac{2}{n-1}$

$\frac{T(n-3)}{n-2} = \frac{T(n-4)}{n-3} + \frac{2}{n-2}$

$\vdots$

$\frac{T(2)}{T(3)} = \frac{T(1)}{2} + \frac{2}{3}$

$\Rightarrow \frac{T(n)}{n+1} = \frac{1}{2} + 2\sum_{i=3}^{n+1} \frac{1}{i}$

$T(n) = \frac{n+1}{2} + 2(n+1)[\log(n+1) - \log(3)]$

$T(n) = n + 2n\log n$

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

Reference :-

$$\sum_{i=3}^{n+1} \frac{1}{i} \leq \int_{3}^{n+1} \frac{dx}{x} = \log(n+1) - \log 3$$

- Randomized Algorithms :-

Randomized is different from deterministic quick sort because if we run the same function 4 times, each time the amount of time it takes would be different; it solely depends on what choices the random function makes.

Las Vegas	Monte Carlo
It always gives correct answer.	May not give correct answer all the time.
No. of steps in each iteration or runs is different.	Fixed execution time.
It gives correct answer fast, considering "average" case; Its worst case is very bad in terms of complexity.	correct answer with "high probability".
eg:- Randomized QS, Hashing	eg:- Primality testing ↳ not a prime means not a prime ↳ prime means may or may not be prime

eg:- Given a sequence of numbers, $a_0, a_1, \ldots a_{n-1}$ and they are distinct. Given a # x from the sequence, find it's rank, i.e Rank(x).

↳ Rank(x) = 1 + # of nos. in sequence > x.

brute force:-

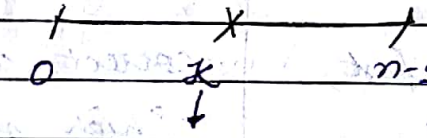
```
int c=1;
for i= 0 to n-1
    if(a[i] > x)
        ++c;
```

$\Theta(n)$.

↳ Now, If you've been given a rank and a sequence no., and you're required to find the number, corresponding to that rank?

approach :-

sort in ascending order. st $1 \le r \le n$



# of nos. between (k+1) and (n-1) are:-

$(n-1) - (k+1) + 1$

$= (n - k - 1)$

and, hence,

rank = no. of nos. (such) + 1

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

$= (n-k)$.

$\Rightarrow r(k) = (n-k)$.

∴ findRank(sorted sequence of size n, #k)

{ return n-k; }

eg:- Rank $n/2$ is linear time (Finding median is linear time).

→ In such a case, if we get it, then recurrence for quicksort would look like:-

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

- $2T\left(\frac{n}{2}\right)$: Since pivot is median always divides in 2 equal parts
- n : partition
- $O(n)$: finding median (Rank $n/2$)

Then, such a Quicksort always has a complexity $O(n \log_2 n)$.

eg:- (for understanding monte carlo)

Given a sequence

$a_0, a_1, a_2, \ldots\ldots, a_{n-1}$.

Find a_i such that rank$(a_i) > n/2$

↓

Soln: pick 'K' nos. uniformly at random

↓

Find minimum of them and return.

x_1
x_2
x_3
x_4
:
x_k

Probability of each of these numbers to have rank $> n/2$ is $\frac{1}{2}$. → we then take minimum of rank of all these #s.

P(mistake for x_i) = $1/2$,

So, P(mistake for K such entries) = $(1/2)^K$

∴ Probability that the given answer is correct is $\left(1 - \frac{1}{2^k}\right)$.

if $k = \log n$,
then, $P(n) = \left(1 - \frac{1}{n}\right)$ for correct answer,

and $T(n) = O(\log n)$.
as we consider $\log n$ #s at random

↳ for the same task, deterministic algorithm will make atleast $n/4$ comparisons
↓ between $n/2$ integers.
deterministic
$T(n) = \Omega(n)$.

So, if $n = 10^9 \simeq 2^{30}$,
Monte Carlo would make only 30 comparisons

- Given an unsorted sequence of n numbers, and a rank. we need to find the element at that rank (Las vegas implementation)

```
RFindRank(A, i, j, r)
{ pivot = rand()%(j-i+1) + i;
  k = partition(A, i, j, pivot);
  swap(a_i, a_k)
  Rank(k) = j-(k+1)+1+1
          = j-k+1
```

→

```
if (r == j-k+1)
        return k;
else if (r < j-k+1)
        AfindRank(A, k+1, j, r);
else
        RfindRank(A, i, k-1, r-(j-k+1));
}
```

as, smaller #s than this pivot, won't affect the rank.

↳ Complexity analysis :-

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

$\theta(n)$ → partition.

we'll only consider that half, which contains the rank.

$$T(n) = n + \frac{n}{2} + \frac{n}{4} + \dots\dots$$

$$\Rightarrow T(n) \leq 2n \Rightarrow T(n) = \theta(n)$$

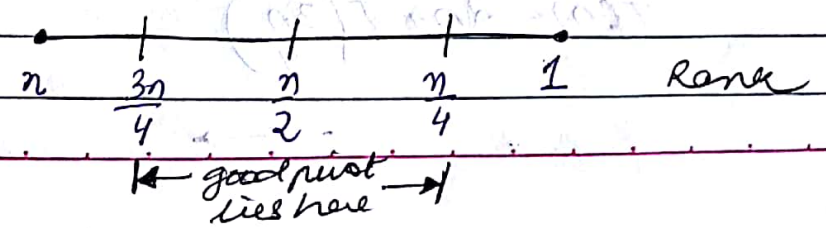
In the worst case :-

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

$$\Rightarrow T(n) = \theta(n^2)$$

- Good Pivot :-

For a pivot to be considered as a good pivot, it should lie between $\left[\frac{n}{4}, \frac{3n}{4}\right]$.



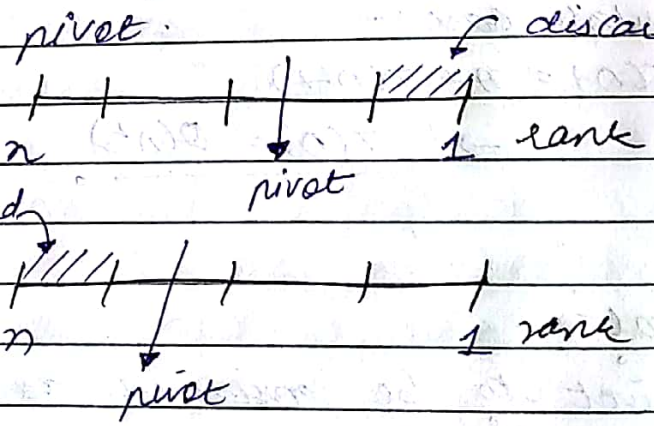
Thus, $R(\text{good pivot}) \in \left[\frac{n}{4}, \frac{3n}{4}\right]$

$$\frac{n}{4} \leq R(\text{good pivot}) \leq \frac{3n}{4}$$

- Probability that a randomly selected pivot is a good pivot is $1/2$.
- If a pivot is a good pivot, then worse case never arises.
- So, the expected # of trials / iterations needed to get a good pivot is $\frac{1}{2} \times 2 = 1$

$\therefore$ for finding good pivot,
$T(n) = \theta(2n)$.

alternative partitioning:
I'm believing one of them to be a good pivot.



So, In worst case,

$$T(n) = 2n + T\left(\frac{3n}{4}\right)$$

Camlin Page 074
Date 16/08/18

$$T(n) = 2n + 2 \cdot \frac{3n}{4} + 2 \cdot \frac{3}{4} \cdot \frac{3}{4} n + \cdots$$

$$T(n) = 2n \left[\frac{1}{1 - 3/4} \right] = 8n$$

$$\Rightarrow T(n) = \Theta(n)$$

Coverage:
Las vegas gives always correct answer

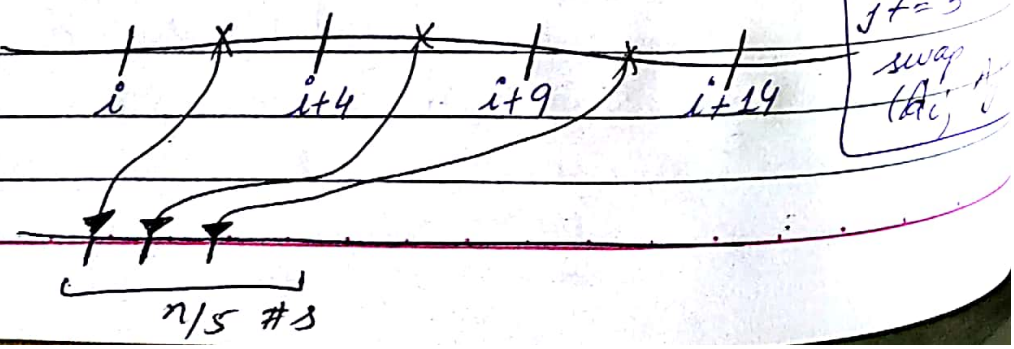
LECTURE-11

- Implementing this algorithm:-

```
FindRank(A,i,j,r)
{   if(j-i+1>5)
    {   p=goodPivot(A,i,j)
        swap(i,p)
        k=partition(A,i,j,A[i])
        swap(i,k)
        if(r==j-k+1) return k;
        else if(r<j-k+1) FindRank(A,k+1,j,r
        else FindRank(A,i,k-1,r-(j-k+1));
    }
}
```

- Now, consider this :-
  divide the given sequence of numbers in g. of 5 (natural grouping), so what happens is :-
  - ↳ first 5 numbers are sort to belong to one group only; then find their medians.
  - ↳ the median is called the leader, then swap the leader with the first element of the sequence.
  - ↳ repeat the process again & again.

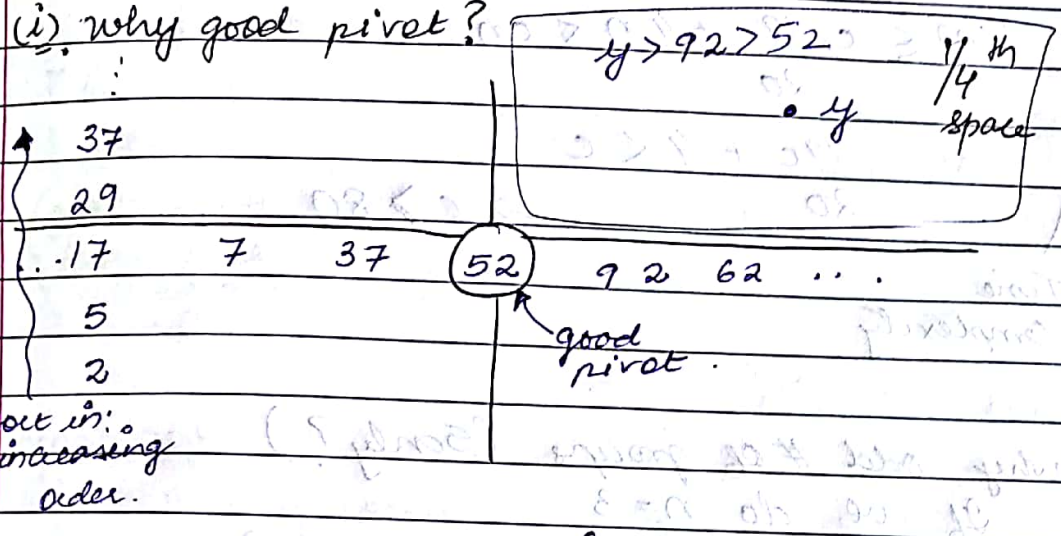
Thus, if we have 65 elements, we could have 13 medians at the first 13 indices. Now, find median of medians using FindRank.



so, goodPivot(A, i, j)
{
:
return findRanker(A, i, i + n/5 - 1, n/10)
}

Median of n/5 elements is at n/10th addr

(i) Why good pivot?



rank(goodPivot) ≤ 3n/4

- similarly, any element in the first quadrant would be smaller.
- Thus, consider 100 numbers, we can say that a good pivot is atleast bigger than 25 nos. and atleast smaller than 25 numbers.

T(n) = no. of comparisons I perform.

$$T(n) = \frac{4 \times 5}{2} \times \frac{n}{5} + \frac{3n}{5} + T\left(\frac{n}{5}\right) + n + T\left(\frac{3n}{4}\right)$$

- $\frac{4 \times 5}{2}$: bubble sort for 5 #s
- $\frac{3n}{5}$: swaps (1 swap = 3 operations)
- $T\left(\frac{n}{5}\right)$: finding median of the group found.
- n : partition
- $T\left(\frac{3n}{4}\right)$: In worst case we're left with 3n/4 elements.

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{3n}{4}\right) + 4n$$

$$T(n) \leq \frac{c \cdot n}{5} + \frac{c \cdot 3n}{4} + 4n$$

$T(n) \leq \frac{c.19n}{20} + 4n < cn$

$T(n) = O(n)$.

$T(n) \leq \frac{c.19}{20} + 4n < cn$

$\frac{19c}{20} + 4 < c$

$\Rightarrow c > 80$.

(2) Time Complexity

(3) Why odd # of groups (5 only?)

If we do $n = 3$

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

$= \frac{cn}{3} + \frac{c.3n}{4} \leq cn$

33% 75% >100%!

$n = 7$:-

$T(n) = T\left(\frac{n}{7}\right) + T\left(\frac{3n}{4}\right) + n + \frac{3.n}{7} + \frac{7.6}{2} \cdot \frac{n}{7}$

$T(n) \leq T\left(\frac{n}{7}\right) + T\left(\frac{3n}{4}\right) + 5n < cn$.

$c\left[\frac{1}{7} + \frac{3}{4}\right] + 5 < c$

Thus $c > \frac{5 \times 28}{3}$

- Thus odd numbers > 3 are okay.
- But we need to consider a tighter case for n/5

$$T(n) = \underbrace{8\frac{n}{5}}_{\text{7 for median, 1 for swap (given we use brute force instead of bubble)}} + n + T\left(\frac{n}{5}\right) + T\left(\frac{3n}{4}\right)$$

$$\frac{13n}{5} + \frac{cn}{5} + \frac{c3n}{4} < cn$$

$$\Rightarrow c > 52.$$

- problems:-

(a) Given a sequence of #s $a_0, a_1, \ldots a_{n-1}$, and a # 'x'. Find whether $a_i + a_j = x$?

Sol (i) for $i = 0$ to $n-1$
  for $j = i$ to $n-1$
    if $(a_i + a_j == x)$
      return true;
  return false;

$O(n^2)$

(ii) if we sort, then
sort() $\longrightarrow O(n \log n)$
So, in the worst case, we have to traverse through all the elements once, i.e. $O(n)$.

i.e. $O(n \log n)$

for $i = 0$ to $n-1$
  BinarySearch $(x - a_i$ in collection)

Camlin Page 074
Date 18/10/23

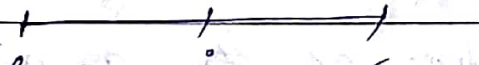
Or,

```
l = 0;
r = n-1;
while(l ≤ r)
{  if (a_l + a_r == x) return true;
   else if (a_l + a_r < x)  l++;
   else r++;
}
```

O(n)

total: $O(n \log n) + O(n)$
$= O(n \log n)$.

Now, consider that,



$a_j > a_l$

$a_j + a_r > a_l + a_r > x$

#s can be like:
-5, -3, -2
-3, 0, -3
2, 3, 5

(b) Given a sequence of #s a_0 to a_{n-1}. Find whether $a_i + a_j = a_k$?

Sol

```
for i = 0 to n-1
   for j = 0 to n-1
      for k = 0 to n-1
         if (a_i + a_j == a_k) return true;
return false;
```

$O(n^3)$.

→ alternative approach:-

```
sort()  → O(n log n)
for (k = 0 to n)
{  l = 0;  r = n-1;
```

```
while (l ≤ r) {
    if (a[l] + a[r] == a[k]) return true;
    else if (a[l] + a[r] < a[k]) l++;
    else r--;
}
```

O(n²)

This is an "open problem." Its answer is unknown, whether an faster algorithm exists or not?

Most of the above kind of problems come from computational geometry.

Where most of the problem's complexity depends on this problem.

LECTURE-12

- **Problems :-**

(c) Given a sequence $a_0, a_1, \ldots a_{n-1}$
Find $\max |a_i - a_j|$.

Sol. (Brute force) $max = a_0 - a_0$

$O(n^2)$.

```
for i = 1 to n
    for j = 1 to n
        if abs(a_i - a_j) > max
            max = abs(a_i - a_j)
```

Or, $\max(a_i - a_j) = \text{max. val} - \text{min. value}$
$= \underline{O(n)}$.

(d) Given a sequence $a_0, a_1, a_2 \ldots a_{n-1}$
find $\min |a_i - a_j|$?

Sol. sort $\rightarrow O(n \log n)$
scan L to R and track the difference between two consecutive nos.
$O(n)$

$T(n) = O(n \log n) + O(n)$
$= \underline{O(n \log n)}$.

This problem cannot be solved faster than $O(n \log n)$.

Element distinctness problem is a collection of numbers with repetition can be solved in $\underline{O(n \log n)}$.

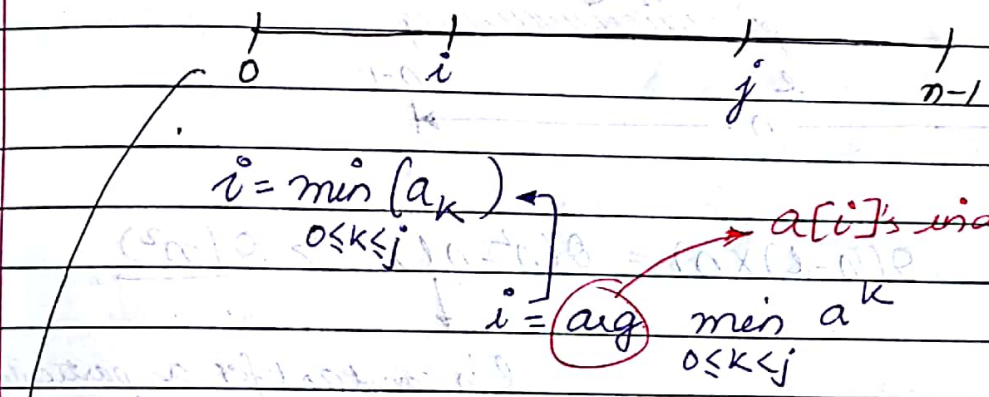
(e) Given a sequence $a_0, a_1, \ldots, a_{n-1}$, find $\max(a_j - a_i)$ such that $j > i$.
(Contrary :- Sell after you buy, where j = selling date, i = buying date)

Sol:

```
max = 0
for i = 0 to n
    for j = i+1 to n
        if (aj - ai) > max
            max = aj - ai
```

$\Theta(n^2)$

OR

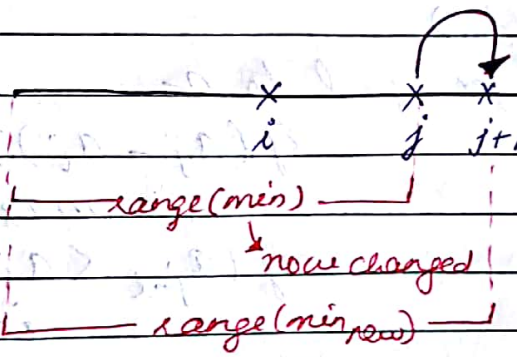


$i = \min_{0 \le k \le j} (a_k)$

$i = \arg\min_{0 \le k < j} a^k$ → a[i]'s index

```
for j = 1 to n
    if (aj - ai > max) {
        max = aj - ai ; }
    if (aj < ai)    i = j;
```

$\Theta(n)$ algorithm



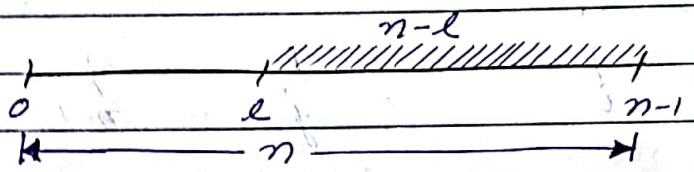
(f) Given a sequence $a_0, a_1, \ldots a_{n-1}$ find $\max(a_j - a_i)$ such that $j \geq i+l$, $l \in [0, n)$, i.e. $0 \leq l < n$.

Sol.

```
i = 0;
for j = l to n
    if (a_j - a_i) > max
        max = a_j - a_i
i++
```

$O(n)$

*



$O((n-l) \times n) = O(n^2 - nl) \Rightarrow O(n^2)$

l is constant for a partic problem,

(g) Given a sequence $a_0, a_1, a_2 \ldots a_{n-1}$ find $\max(a_j - a_i) = a_k$ such that $0 \leq k \leq j$

Sol.

$O(n)$.

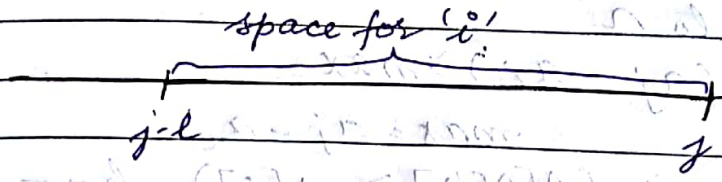
```
i = 0
for j = l to n
    if (a_j - a_i) > max
        max = a_j - a_i
    if (a_{j-l} < a_i)
        i = j - l + 1
```

(b) Given a sequence $a_0, a_1, a_2 \ldots a_{n-1}$

Imp. find max $a_j - a_i$ such that

$$j - l \leq i < j$$

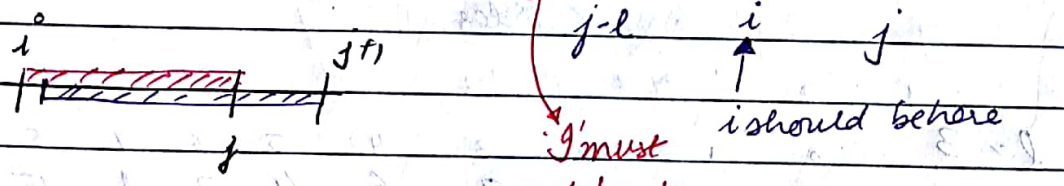
i.e. solution within l - days



Sol,

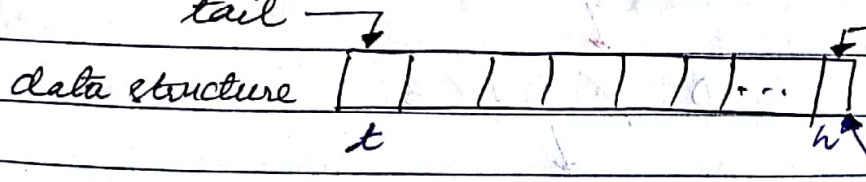
```
i = 0;
max = 0
for j = l to n
    if (aj - ai) > max
        max = aj - ai
    if (j - i < l)
        if (aj < ai)  i = j
    if (j - i == l)
```

then



i.e. I need the minimum value to find max. difference in the range $[j-l, j]$ and also, that in $O(1)$,

so I need an explicit Data Structure



and my Data Structure is stack.

at head and Queue at tail. It keeps track of indices:

```
i = 0
for j = 1 to n
    if (aj - ai) > max
        max = aj - ai
    while (A[D[h]] >= A[j]) h--
    D[++h] = j
    if (j+1-i > l) i = D[t++]
```

Why not while here? just try it out.

By such an implementation of a datastructure, my DS will have indices such that corresponding elements are always in increasing order.

eg:- l = 3

	0	1	2	3	4	5	6	7	8
A =	3	8	11	9	6	4	2	1	5
	↓			↓					
	i			j					

i = 0

j = 3+0 = 3 D = 0 1 2

max = 11 - 3 = 8

D = ∅ 1 3

D = 1 3 ... and so on

(i) Now, $j <= i+l$. → i.e. same as previous question

Sol,

```
h = 0
D[0] = 0
t = 0
i = 0
max = 0
for j = 1 to n
    if aj - ai > max
        max = aj - ai
    while (h > 0 && A[D[h]] > A[j]) h--
        D[++h] = A[j]
    while/if (j+1 - i > l)
        i = D[++t];
```

probably is constant time.

O(1) operation

while not required, as it is executed only once. So, it is fine to use if.

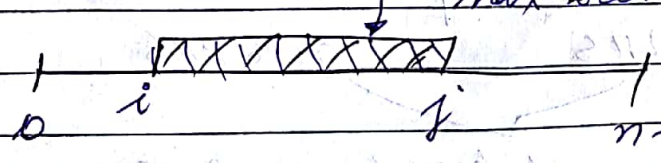
my stack can pop only as many elements as pushed, so, cannot be $O(n^2)$, it is $O(n)$.

$T(n) = O(n)$.

(j) Given a sequence $a_0, a_1, \ldots a_{n-1}$.
Find the subsequence a_i — a_j where sum of nos. is maximum.

max sum – subsequence.

Sol.



note: for i = 1 to n
mean i <= i <= n

```
max = 0
for i = 1 to n
  for j = 1 to n
      sum = 0;
      for k = i to j+1
          sum += A[k]
      if (sum > max)
          max = sum.
```

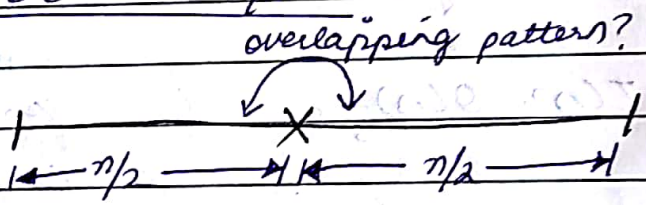
O(n^3)

OR,

```
for i = 0 to n
    sum = 0
    for j = i to n
        sum = sum + aj
        if sum > max
            max = sum.
```

O(n^2)

- divide & conquer

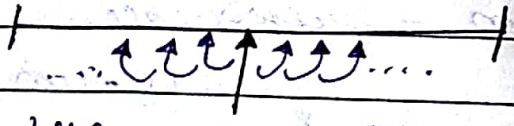


find max. subsequence entirely on LHS & max subsequence entirely on RHS.

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

max overlapping subsequence

$\rightarrow O(n \log n)$



go to LHS go to RHS

and keep track of max sum.

Wherever we get 'max', there we have to start & end.

OR

- Cumulative Sum :-

$a_0, a_1, \ldots\ldots\ldots\ldots a_{n-1}$

$P_0 = a_0$

$P_1 = P_0 + a_1 = a_0 + a_1$

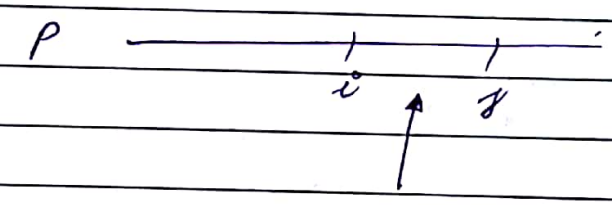
$P_2 = P_1 + a_2 = a_0 + a_1 + a_2$

$\vdots$

$P_{n-1} = a_0 + a_1 + \ldots a_{n-1}$

$\max(a_j - a_i)$, $j > i$ → linear time.

So, $\max(P_j - P_i)$ $\quad T(n) = \Theta(n)$



$\max \text{ sum} = P_j - P_i$

$= \sum_{k=i+1}^{j} a_k$

and sequence is $i+1$ to j

The problem with this method is that P_i is too large.
P requires additional time

- Homework :-

Try doing it without using P_i in $O(n)$

→ Length of subsequence

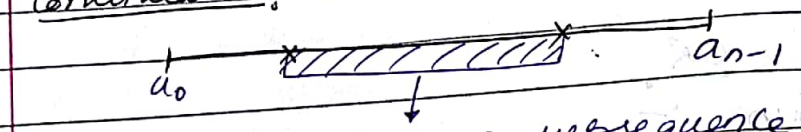
→ length of subsequence is atleast 'l'

→ length of subsequence is atmost 'l'

Camlin Page
Date 28.08.

LECTURE-13

- Continuation :-



$p_0 = 0$

$p_1 = a_0$

$p_2 = a_0 + a_1$

$p_i = a_0 + a_1 + a_2 + \cdots + a_{i-1}$

$$\boxed{p_i = p_{i-1} + a_{i-1}}$$

max $p_i - p_j$

(while $j > i$)

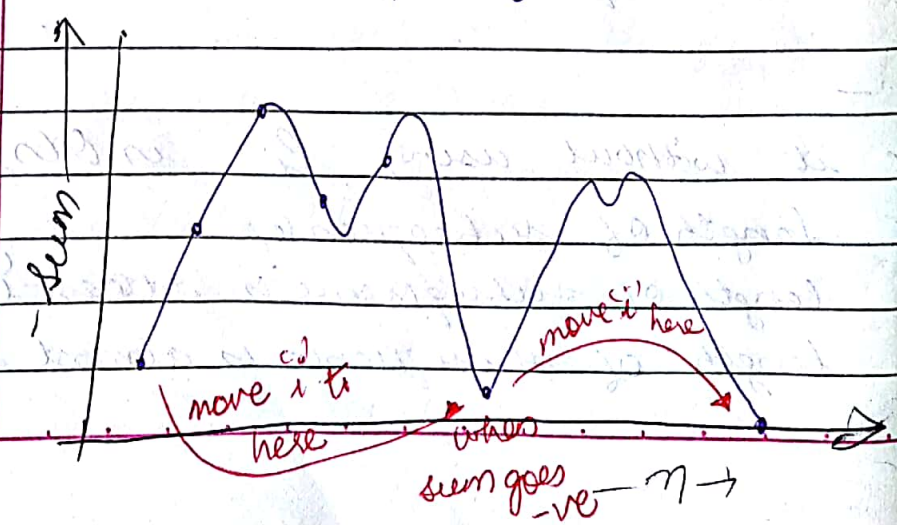
```
i=0;
max=0
for j = 1 to n+1
    if (pj - pi) > max
    { max = pj - pi
      l = i
      r = j-1
    }
if (Pj < Pi) i = j;
```

$T(n) = O(n)$

$S(n) = O(n)$

do we really need this?

avoid computing p's of i.

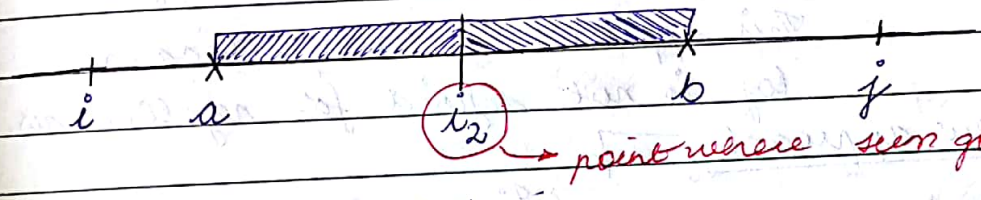


now, consider previous algorithm again:

```
sum = 0
max = 0
for j = 0 to n
    sum += a_j
    if (sum > max)
        max = sum // sum here mimics p_j
    if (sum < 0)
        sum = 0
```

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

$S(n) = \theta(1)$



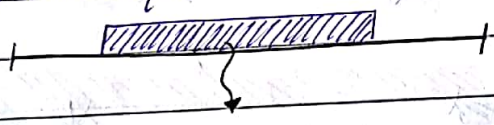
i_2 → point where sum goes < 0

at any point of time,

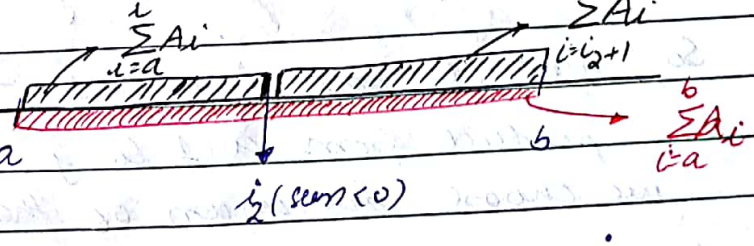
$sum(a, b) < sum(i_2, b)$ iff

$sum(a, i_2) < 0$

So, max subsequence cannot be like this



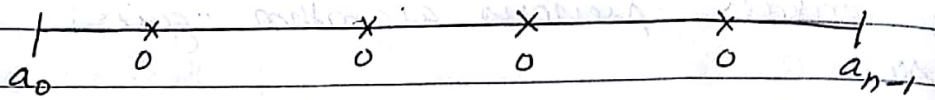
So, whenever sum goes < 0, then we've to start again, i.e. new subsequence



Therefore, $\sum_{i+1}^{b} A_i > \sum_{i+1}^{b} A_i + \sum_{a}^{i} A_i$ → negative

$\Rightarrow \sum_{i+1}^{b} A_i > \sum_{a}^{b} A_i$

eg:-



find maximum product.

↓

sequence cannot overlap a zero.

Sol. one of the possible solution is:-
use logarithms.

$\log xy = \log x + \log y$

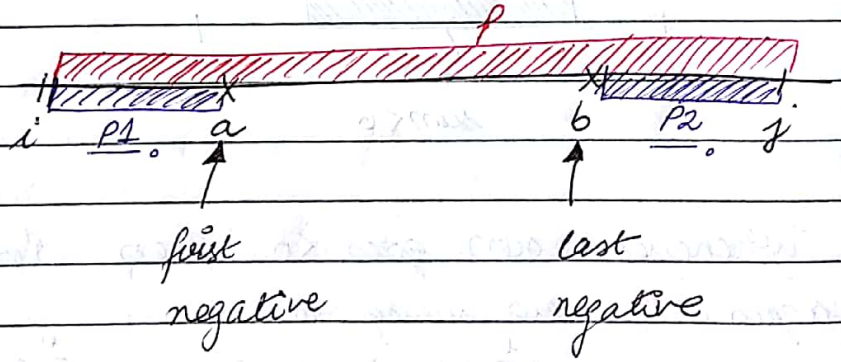
↓

This approach is wrong because log is not defined for negative nos.

new approach →

i can traverse from i to j guaranteed no zeroes in between

also, keep a track of no. of negative numbers. If they're even, still multiply



So, product from i to b-1 → P/P2
product from a+1 to j → P/P1
we choose maximum of them both.

if (P1 > P2) return P/P2.
else return P/P1

↓

$l = i$
$b = j-1$

→ $\{l = a+1$
$r = j$

eg. $a_0, a_1, \ldots\ldots a_n$. Count # of inversion pairs.

↳ if $i<j$, then $a_i > a_j$

sol:

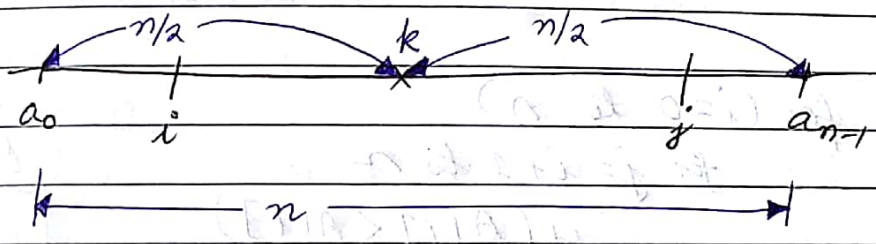
```
c=0;
for i = 0 to n
for j = i+1 to n
        if (a_i > a_j) c++;
print(c)
```

$O(n^2)$

$0 \le c \le \frac{n(n-1)}{2}$

all pairs are inversion pair

method 2 :-
divide and conquer.



Modified_merge_Sort (A, i, j)

$O(n \log_2 n)$.

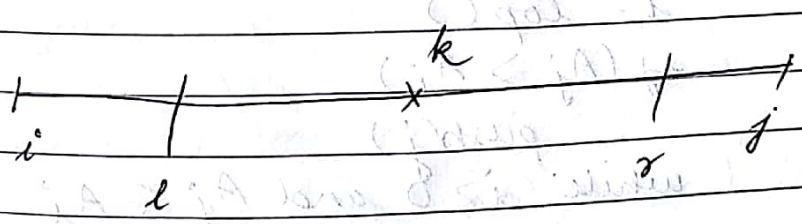
$k = (i+j)/2$

count = modified_merge_sort (A, i, k) + modified_merge_sort (A, k+1, j)

modified_merge(A, i, k, j)

# of inversions on LHS

# of inversions on RHS




```
modified_merge (A, i, k, j)
c=0;
if (A_l < A_r)   T[a++] = A[l++]
else {  T[a++] = A[r++]
c = c+k-l+1
}
return c;
}
```

eg:- a_0 ——— a_{n-1}

$a_i > a_j$: application of stack

first # on RHS smaller than 'i'

sol:-

0	1	2	3	4	5	6	7
4	3	8	10	9	7	12	11

1 0 5 4 5 0 7 0 ← 1st # on RHS smaller than A_i

no. at index 1 is smaller

min[0]=1

min[1]=0 → no. no. on RHS is smaller

```
for (i=0 to n)
    for j= i+1 to n
        if (A[j] < A[i])
            { min[i] = j;
              break; }
if j==n
    min[i]=0
```

$\theta(n^2)$ worst case is sorted array

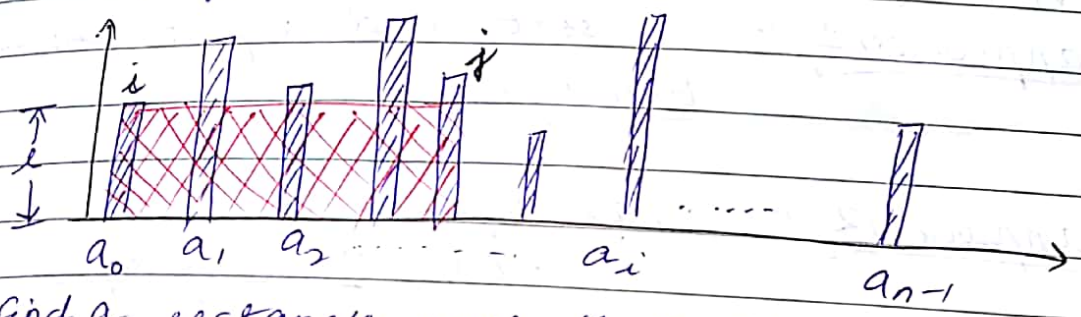
other approach ($O(n)$) using stack:-

index

```
push(0)
for j = 1 to n
    i = top()
    if (Aj > Ai)
        push(j)
    while i>=0 and Aj < Ai
        mini = j
        pop()
        i = top()
while i>=0
    mini = 0
    pop()
    i = top()
```

→ application.

a_0 ——————— a_{n-1} plotted on a histogram.



Find a rectangle such that its area is maximum.

area = $l(j - i + 1)$

```
for i = 0 to n
    for j = i to n
        l = A_i
        for K = i to j+1
            if A_K < l
                l = A_K
            if l*(j-i+1) > max
                max = l*(j-i+1)
```

$O(n^3)$

another approach :-

L_{min_i} → $O(n)$

R_{min_i} → $O(n)$

```
for i = 0 to n
    if ((R_min_i - 1) - (L_min_i + 1)) > max
        do something
```

$O(n)$

overall = $O(n) + O(n) + O(n) = O(n)$.

eg:- $a_0, a_1, a_2, \ldots\ldots, a_{n-1}$

Find K largest #s when —

(i) $n = 10^6, K = 10^4$

approach 1 :- sort and report last K #s.

$\Theta(n \log n)$.

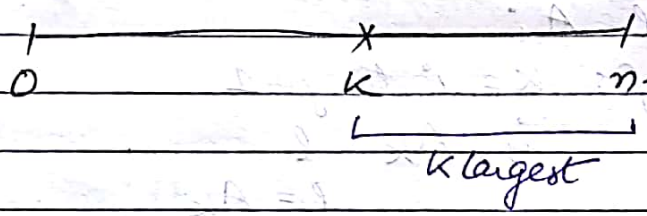
approach 2 :- maxheap.

$T(n) = \Theta(n + K \log n)$

but if $K \to \frac{n}{c} \Rightarrow T(n) = \Theta(n \log n)$ (c = constant).

approach 3:- Partition :-

Find k's rank.

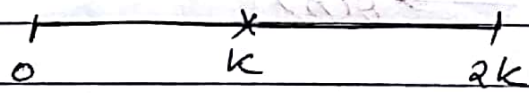


(ii) $n = 10^9, K = 10^6$

now, 'n' cannot be stored in an array

sol.

take an array of size 2K.



1. read first 2k #s, store them in an array
2. find its median
3. read next K #s from index 0 to k-1 in the new array of size 2K.

repeat

median = $\Theta(K)$

Repetition = n/k times.

$T(n) = \Theta\left(\frac{n}{k} \times k\right)$

$\Rightarrow T(n) = \Theta(n)$.

OR

using minheap algorithm.

first k #s ⟶ O(log k) for add & delete

Build heap ⟶ O(k)

for (n-k) # repeat add/delete

⇒ O(k) + (n-k). log k.

= O(n log k).

Assignment :-

1. $n = 10^9$, $k = 10^5$.
Using deterministic partition, (find rank), determine Largest 'k' elements. in O(n).
2. Using randomized algorithm. O(n).

- Note :- You will observe that deterministic is slower than randomized even though both have O(n) complexity. Explain why?

eg: a_0 ——— a_{n-1}

find k #s which are closest to the median.

eg:- 1 3 10 11 (15) 23 28 31 35 (Median)

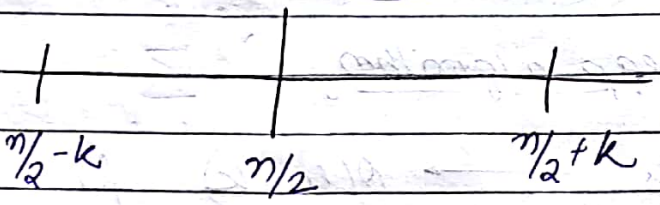
k=3 ⟶ 10, 11, 15

k=4 ⟶ 10, 11, 15, 23

k=5 ⟶ 3, 10, 11, 15, 23

Camlin Page 89
Date 28/08/2018

sol. find median $(n/2+1)$ $= O(n)$.



look at these 2k #s.

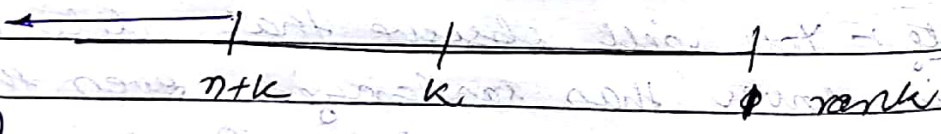
Substract median ~~elem~~ and then find the median of them; ie. median of the distance from the median.

$b_i = |a_i - \bar{x}|$ ⟶ $O(n)$.

a_i: distance from median

$\bar{x}$: median

now, rank 'n+k' including k, report them.

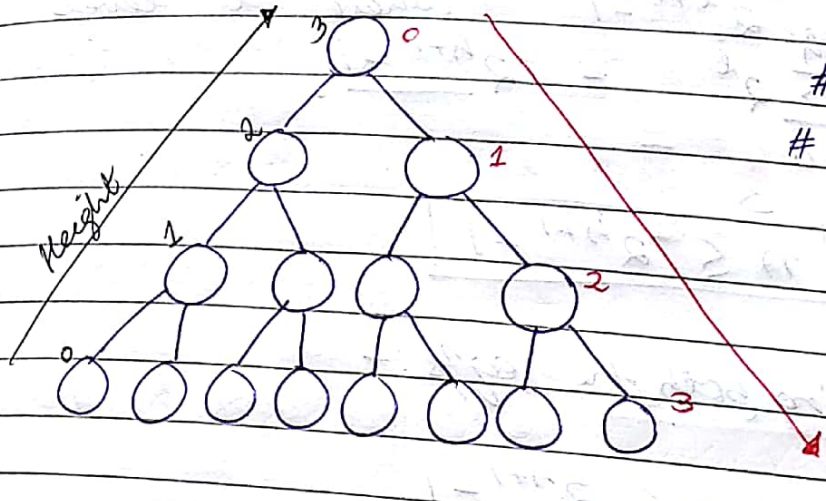


$O(n)$.

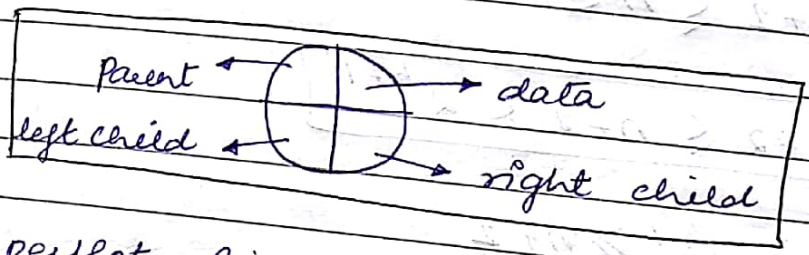
LECTURE- 14

Camlin Page 098
Date 30/08/2018

Perfect Binary Tree :-



# of leaf nodes $n_l = 8$
# of internal nodes $n_i = 7$



Full Binary Tree → Two children

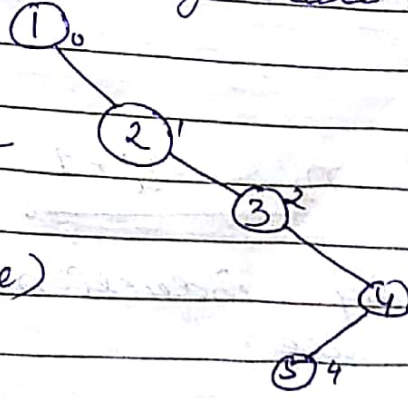
↳ perfect Binary tree = full + all leaves a same level.

maximum height is at root
max level is at leaf node
max height = max level.

→ a degenerate tree has 0, 1 children.

h = height of the tree
n = # of nodes in the tree.
then $h \leq n-1$

eg: Consider a degenerate tree :-



here, h is the maximum height we can get (degenerate tree)

$n = 5$
$h = 4$
so $h \leq n-1$

↳ now, if 'h' is fixed, the max # of nodes = $\sum_{l=0}^{h} 2^l = 2^{h+1} - 1$ where l = levels.

So, $\boxed{n \leq 2^{h+1} - 1}$

∴ considering both the eq^ns :-

$h+1 \leq n \leq 2^{h+1} - 1$

$\Rightarrow \boxed{h+2 \leq n+1 \leq 2^{h+1}}$

i.e. $2^{h+1} \geq n+1$

$\Rightarrow h+1 \geq \log(n+1)$

$\Rightarrow h = \Omega(\log n)$.

if $h = O(\log n)$, then it is a balanced binary tree.

↳ # of leaf nodes n_l
# of internal nodes n_i

then total nodes $n = n_l + n_i$

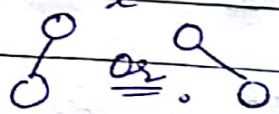
- Theorem :- $n_l \leq n_i + 1$.

Proof is by induction on # of nodes in the tree

if $n=1$, $n_l = 1$, $n_i = 0$

then $1 = 0+1$ (✓)

if $n=2$, $n_l = 1$, $n_i = 1$



$1 \leq 1+1$

$1 \leq 2$ (✓)

for a degenerate tree,

$n_i = n-1$

$n_l = 1$

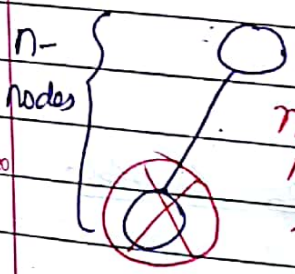
then, $1 \leq n-1+1$

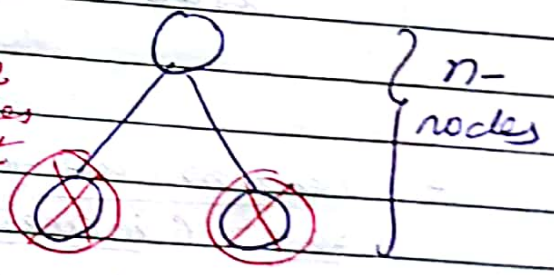
$1 \leq n$ (✓)

using

→ Strong Induction :-

tree with n # of nodes :





again I should be able to apply induction

$n_l' = n_l$

$n_i' = n_i - 1$

$n_l' \leq n_i' + 1$

$n_l \leq n_i - 1 + 1$

$\Rightarrow n_l \leq n_i \leq n_i + 1$

$\Rightarrow \boxed{n_l \leq n_i + 1}$

true

$n_l' = n_l - 1$

$n_i' = n_i - 1$

$n_l' \leq n_i' + 1$

$n_l - 1 \leq n_i - 1 + 1$

$n_l - 1 \leq n_i$

$\boxed{n_l \leq n_i + 1}$

true

↳ Height.

$n_l \leq n_i + 1$

add n_l

$2n_l \leq n_i + n_l + 1$

but $n_l \leq 2^h$

$\Rightarrow \boxed{h \geq \log n_l}$ → minimum height of a perfect binary tree $\geq \log n_l$.

- Sorting algorithm :-

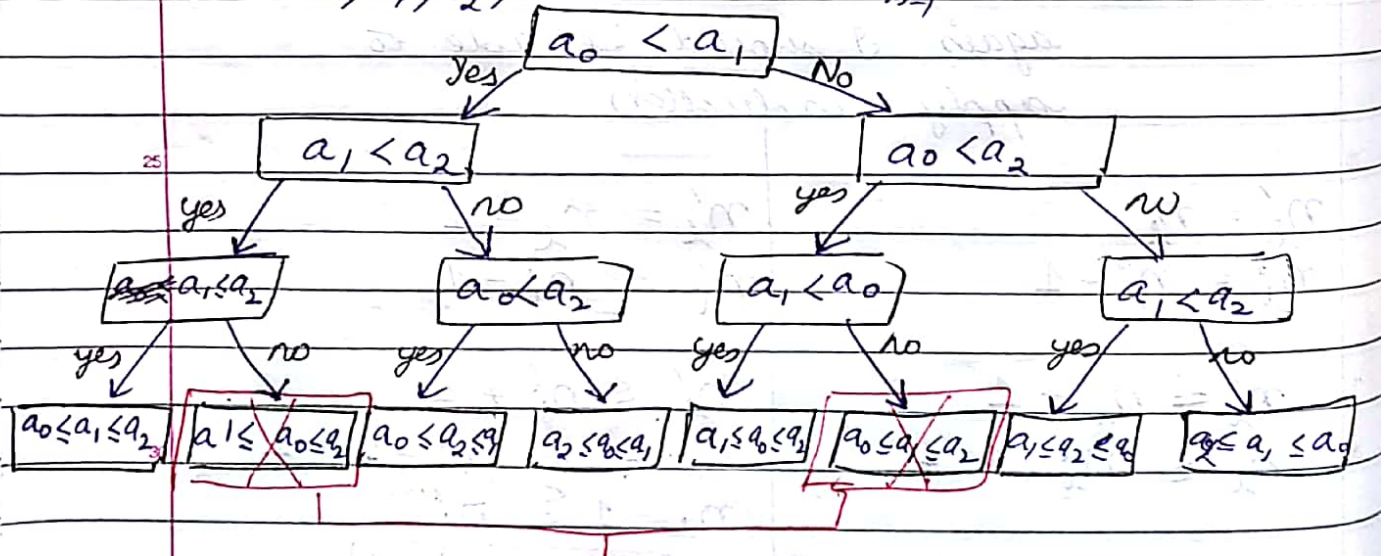
$a_i < a_j$ given that $i < j$

a_0 ——— a_{n-1}

is less than (a_i, a_j) → True / False

- Decision tree for sorting algorithm of bubble sort :- (increasing order)

$a_0, a_1, a_2, \ldots\ldots, a_{n-1}$



this will never happen (unreachable) though bubble sort does this again & again. There is no need to check whether this has happened before.

```
for i=0 to n
    for j= 0 to n
        if a_j > a_j+1
```

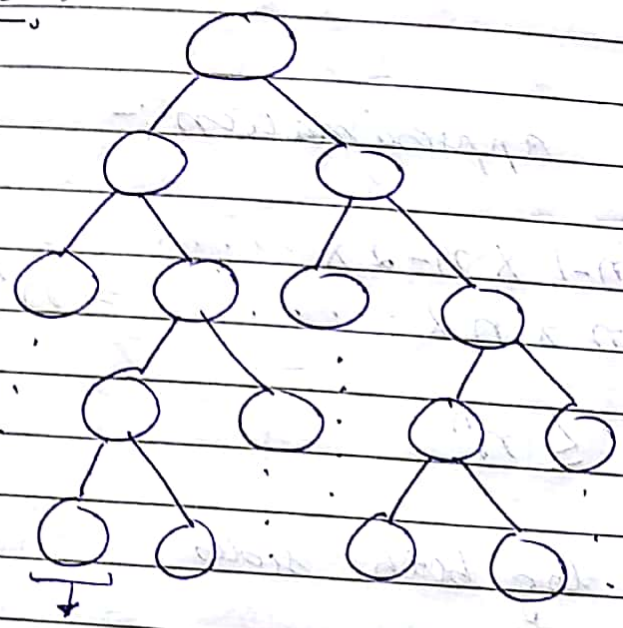
true

swap

true

$a_0 < a_1$ $a_1 < a_2$ a_0 a_1 $\boxed{a_2}$

- decision tree :-



conclusion
(sorted sequence)

- given an algorithm I can create a decision tree.
- given an input, it is easy for me to follow the path.
- worst case case of comparisons = # of internal nodes in the tree along a path.
- leaf node :- has sorted sequence.

- worst case of comparisons = height of tree (decision tree) = 'h'.
- Leaf nodes are permutations of sorted sequence.

# of leaf nodes $\geq$ # of permutations possible

$n_l \geq \boxed{n!}$

i.e. n_l can be much more than actual $n!$.

- to reach a leaf, # of comparisons = height / levels in the tree.

- so in worst case, # of comparisons

$$h \geq \log n_l \geq \log n!$$

$$\Rightarrow h \geq \log n! \quad \Rightarrow h = \Omega(n \log n)$$

(# of Comparisons)

By sterling's approximation :-

$$\rightarrow n! = n \times n-1 \times n-2 \times \dots$$

$$\rightarrow n! \leq n \times n \times n \times \dots = n^n$$

$$n! \leq n^n$$

taking log both sides

$$\log n! \leq n \log n$$

$$\Rightarrow \log n! = O(n \log n)$$

again,

$$n! \geq n \times n-1 \times \dots \times \frac{n}{2}$$

$$\rightarrow n! \geq \left(\frac{n}{2}\right)^{n/2}$$

$$\rightarrow \log n! \geq \frac{n}{2} \log \frac{n}{2}$$

$$\rightarrow \log n! = \Omega(n \log n)$$

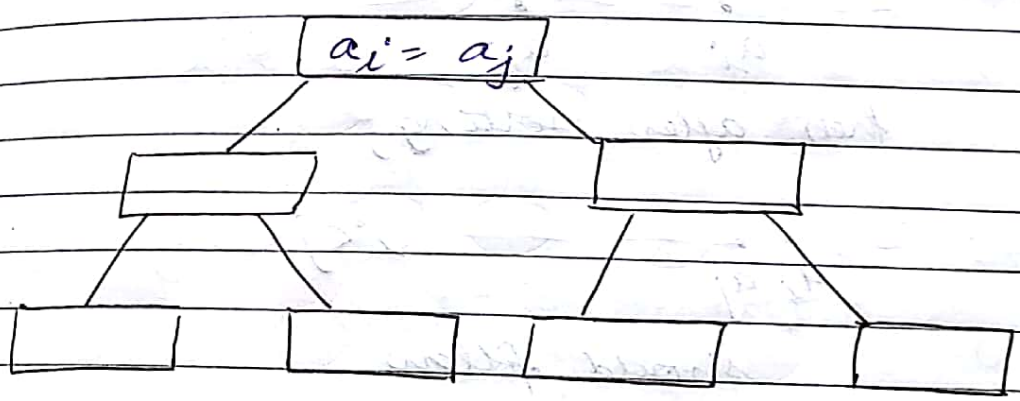
This implies $n \log n$ is the lower bound for comparison based sorting

so, $\frac{n}{2}\log\frac{n}{2} \leq \log \Omega \leq n\log n$.

$\rightarrow \underline{\log \Omega = \Theta(n\log n)}$.

↳ $a_0, a_1, \_\_\_\_\_, a_{n-1}$

$a_i = a_j = ?$ (checking distinct elements)



- in sorting algorithms, sorted sequence is always in leaf node of decision tree.
- If there are no distinct elements, algorithm terminates in leaf node, otherwise, internal nodes (one of the Internal Node in decision tree).
- worst case # of comparisons = $\Omega(n\log n)$ if allowed to do only comparisons.
- so for sorting → $\Omega(n\log n)$

  merge sort → additional array → $n\log n$

  Quick sort → deterministic → $n\log n$

  ↳ (but in QS constants involved are much higher)

In practice we use randomized Quick Sort.

- **Heap sort** :- $T(n) = \theta(n \log n)$

  $S(n) = \theta[n + (n-1)]$

  → space disadvantage same as merge sort, so we can't have whole array in memory.

- **Stable Sort** :-

  if before sorting for $\forall a_i = a_j$ in array a, such that $i < j$,

  $a_i \quad a_j$

  then, after sorting,

  $a_i \quad a_j \quad i < j$

  should follow.

↳ **Bubble** :- depends on the way of implement:
- $a_j > a_{j+1}$ → stable
- $a_j \geq a_{j+1}$ → unstable

↳ **Selection** :- depends on the implementation
- $a_j \geq a_{max}$ → stable
- $a_j > a_{max}$ → unstable.

↳ **Insertion** :- depends
- if I stop when I see smaller or equal elements → stable
- If I stop when I only see smaller elements → unstable

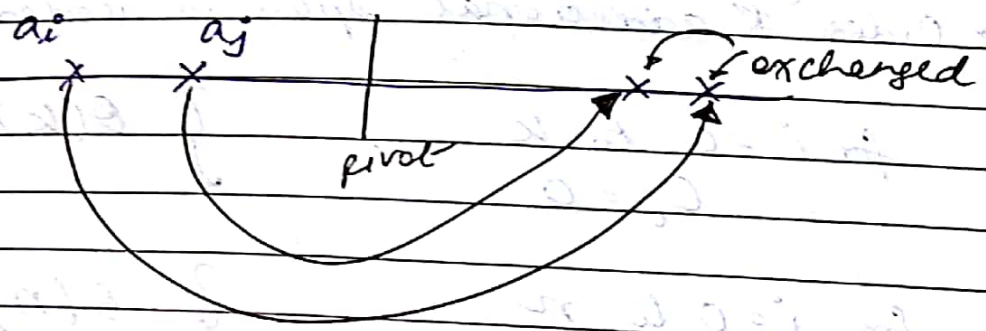
↳ merge sort :-

stable if ($A[L] \leq A[r]$)

$B[a++] = A[L++]$

unstable if ($A[L] < A[r]$) . . . .

↳ Quick Sort :- (unstable)



- it not only depends if pivot repeats, but also on this above fact. $a_i, a_j > pivot$

- so, if we require stability, we shouldn't use Quick Sort

- Sorting in linear time :-

Counting Sort

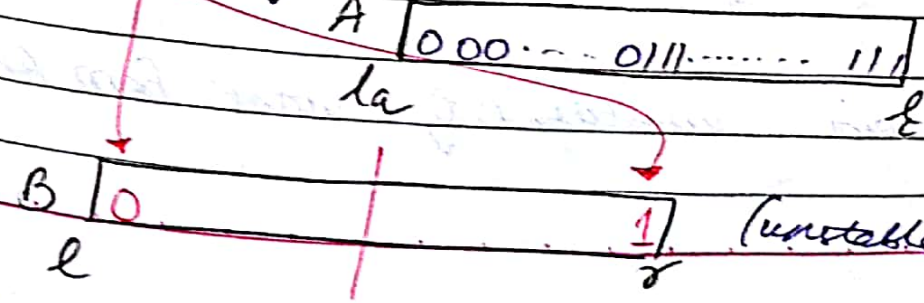
$a_i \in \mathbb{Z}^+$

$0 \leq a_i < K$ [K is a small #]

if $K = 1$, I have a binary sequence.

eg :- 1010110 1111110001001110110 . . . . .

sort it !



Count # of zeroes, then r. becomes 'c', so if we use it now, our sequence is stable.

↳ for general k,
$a_i \in Z^+$, $0 \le a_i \le K$
and $K_i = O(n)$

↳ Order 'K' additional space is needed (Count)

```
for i = 0 to k          } O(K)
    C_i = 0

for i = 0 to n          } O(n)
    C[A_i] ++
```

↳ Thus, C_i contains how many i's repeat. If I start directly, I dont know where to place each value in A_i form, so, find cumulative sum.

```
for i = 1 to n          } O(n)
    C_i = C_i + C_i-1
```

eg: A = | 0 0 2 2 3 3 3 5 5 5 |

C = | 2 2 4 7 7 10 |
0 1 2 3 4 5

↳ to avoid unstability start from RHS

for $i = n-1$ to 0
$B[--C[A_i]] = A_i$ } $O(n)$.

* stable unstable

for ($i = 0$; $i <= n$; $i++$)
$B[C[A_i]--] = A_i$

Here our basic assumption is that
$0 \leq a_0 \leq K$

$T(n) = O(3n+k)$.
$S(n) = O(2n+k)$.

↳ in case we have negative #s, just find their absolute value and add '1' to the count sequence

↳ Counting sort is not applicable for natural nos.

↳ $K = 10^9$ is senseless (cannot declare an array of such size)

- Radix sort :-

eg:

34	2
67	6
34	9
76	1
65	3
35	1

→

7	6	1
3	5	1
3	4	2
6	5	3
6	7	6
3	4	9

→

3	4	2	342
3	4	9	349
3	5	1	351
6	5	3	653
7	6	1	676
6	7	6	761.

↳ we get a sorted sequence in the end due to stability.

↳ $T(n) = O(n) * k$

sort

# of iterations of sorting (here # of digits)

$\Rightarrow T(n) = O(nk)$

↳ for a faster implementation, consider a bigger base say 1000000

so 9632 48342

iteration 1

iteration 2.

↳ in i^{th} iteration, how do I get i^{th} block of 1000000.

j^{th} iteration $\left(a_i / 10^{6j}\right) \% 10^6$ → Considers 5 digits at a time

for single bit,

j^{th} iteration $\left(a_i / 10^j\right) \% 10$

↳ here I traversed from right to left, i.e most significant bit to least significant bit.

↳ If I reverse the direction, i.e. I traverse from 1 to r, then it is bucket sort.

↳ 1 bucket may not have so many #s, and bucketing should be done effectively.

LECTURE - 15

- **Dynamic Data Set Problem:**

Hashing { • Add (x) • Delete (x) • Search (x)

- $O(1)$ on average
- $O(n)$ worst case

- However, Linear Search is most better, in worst case of hashing.

insert/add = $O(1)$ add at end.

delete = $O(n) + O(n) = O(n)$

- search: This can be achieved through Array or Linked list,
- delete & copy last element and decrease # of elements 'n' by 1.

- In practice, we use Hashing only.

- **Binary Search Tree:-**

- insert/add element
- delete element
- search element

↳ If we have operations like

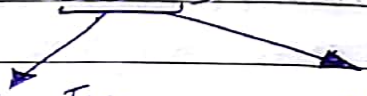
Rank(x) } BST does this in $O(\log n)$

FindRank(r)

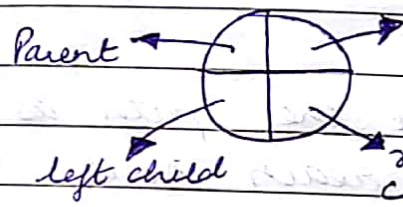
↳ given a #, how many # are > than it

↳ Hashing does findRank(r) in O(n); as it goes through all #s and search.
Thus, hashing does not efficiently does the task of Rank(x) and findRank(r).

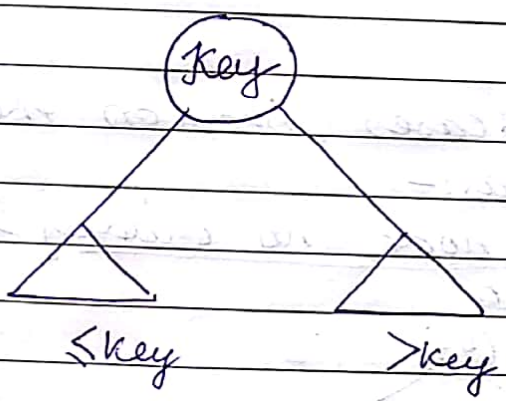
↳ These tasks take O(log n) in Balanced Binary Search Trees (BBST)



↳ Structure :-

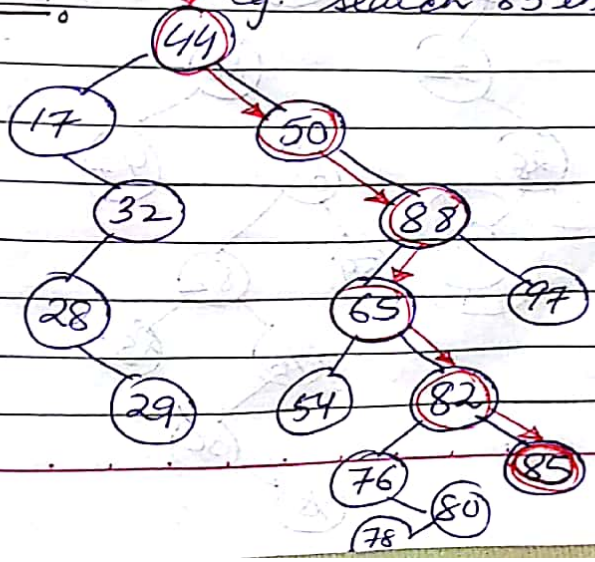


BST is defined on the basis of keys.



Inorder (BST) gives sorted sequence in O(n).

↳ Search in BST :- eg: search 85 in :-

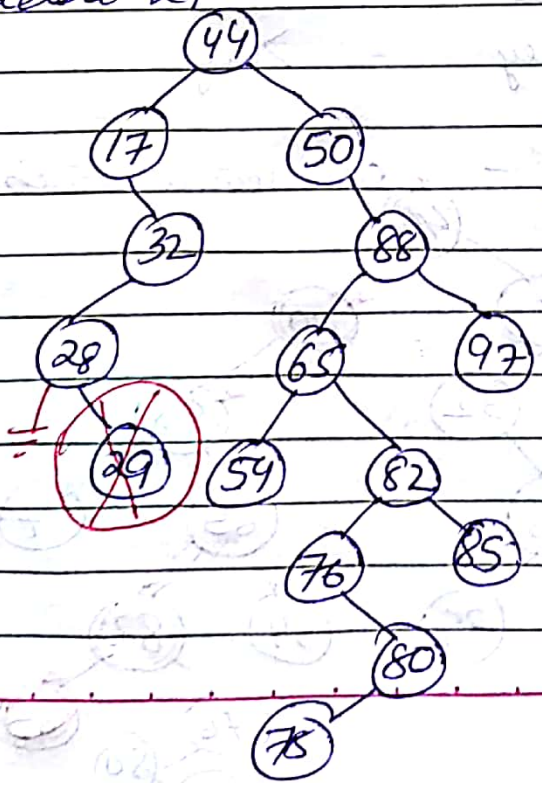


```
search (node, X)
{
  while(node)
  { if (node -> key == x)
        return true;
    else if node -> key > x
        node = node -> lc;
    else
        node = node -> rc;
  }
  return False;
}
```

- insertion :- traverse the path as done in searching. when reach null, then add the node to that place.

- delete :-
  delete has 3 cases, based on the # of children it has:-

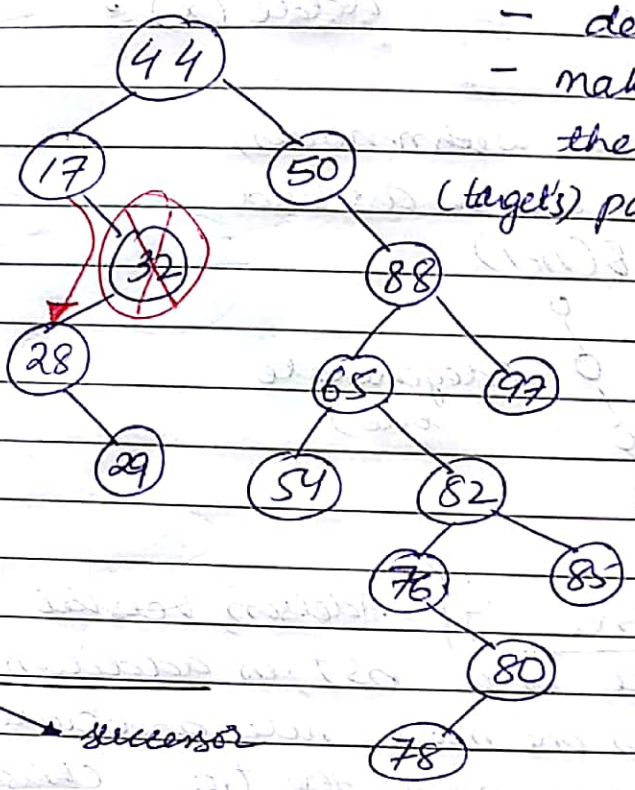
(i) case 0: (leaf node, no children)
eg: delete 29



- search the node to be deleted & keep track of its parent.
- when found, parents child (left/right) should now point NULL
- delete the target node.

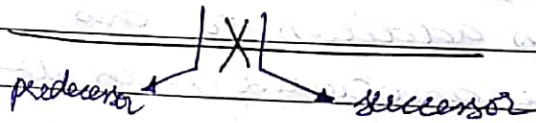
(ii) Case 1 :- (1 - child)

eg: delete 32



- delete the node
- make its child as the child of its (target's) parent

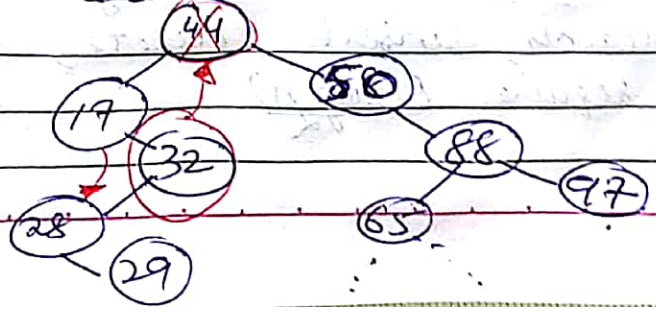
Inorder



↳ We did not used inorder predecessor as, I may have to delete many things, un-necessarily, until I descend down the tree's leaf, so more # of operations

(iii) Case 2 :- (2 - children) -

eg: delete 44



- replace with inorder predecessor or successor.
- recursively delete inorder predecessor.

↳ Complexity for these operations is O(height of the tree),
i.e. search(x)
insert(x)
delete(x)
} O(n)

↳ for a tree with n-nodes, height can be as bad as O(n-1)

in case tree is degenerate, it's no more than a linked list.

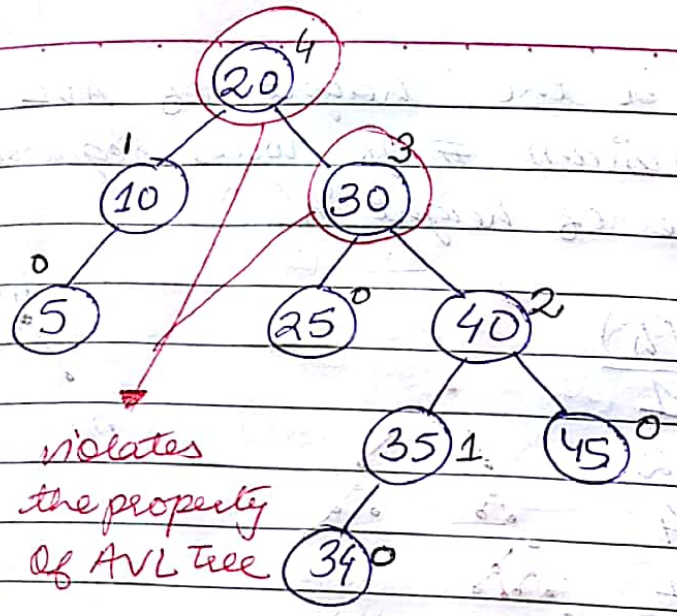
- AVL Tree :- (Adelson, Velskii & Landis)
AVL tree is a BST, in addition to this there is one more rule associated with it.
↳ for every node the tree, child's height can differ by atmost 1.

i.e. $|ht(lc) - ht(rc)| \leq 1$,

i.e. $ht(lc) - ht(rc) \in \{0, -1, +1\}$

↳ Hence, it is a height balanced tree & search, insert, delete, all operations require $O(\log_2 n)$

eg:

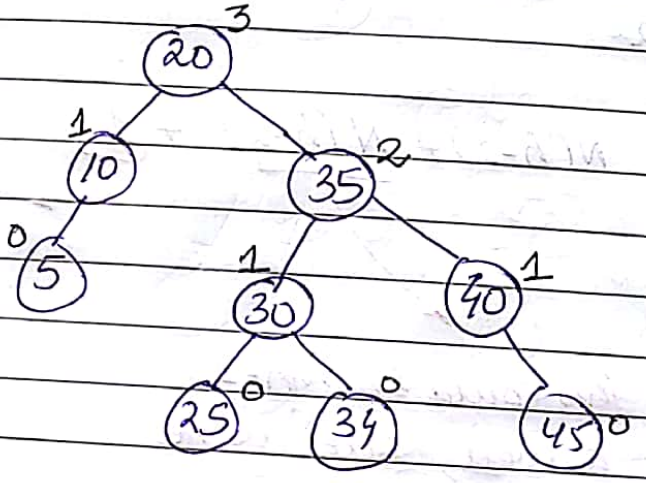


violates the property of AVL Tree

(height of NULL = -1)

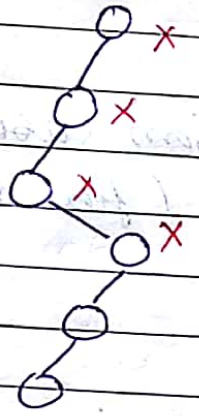
ht(node) = 1 + max(ht of childs)

eg:-



(AVL Tree) ✓

So,



is not an AVL tree as it is degenerated.

Height of an AVL Tree = O(log n) [Proof] :-

already proved :- any binary tree has height Ω(log n)

to show, AVL Tree's height = O(log n).

↳ Let h - be the height of AVL tree.
N(h) = minimum # of nodes required in AVL tree of height 'h'.

h	N(h)
0	1
1	2
2	4
3	7
4	12

↳ so, $N(h) = N(h-1) + N(h-2) + 1$

1 → the node itself

this could be N(h-1) as well, but we're considering minimum nodes

and we know that from above recurrence relation (fibonacci sequence)

$N(h) \geq 2^{h/2}$

↳ AVL has n-nodes & height h,
then $n \geq N(h) \geq 2^{h/2}$

$2^{h/2}$ → from fibonacci sequence

N(h) → min # of nodes for height 'h'.

n → atleast ≥ min # of nodes for AVL of ht 'h'.

$\Rightarrow 2^{h/2} \leq n$

$\rightarrow h \leq 2 \log n$

$\Rightarrow$ ht (AVL tree) = $O(\log n)$.

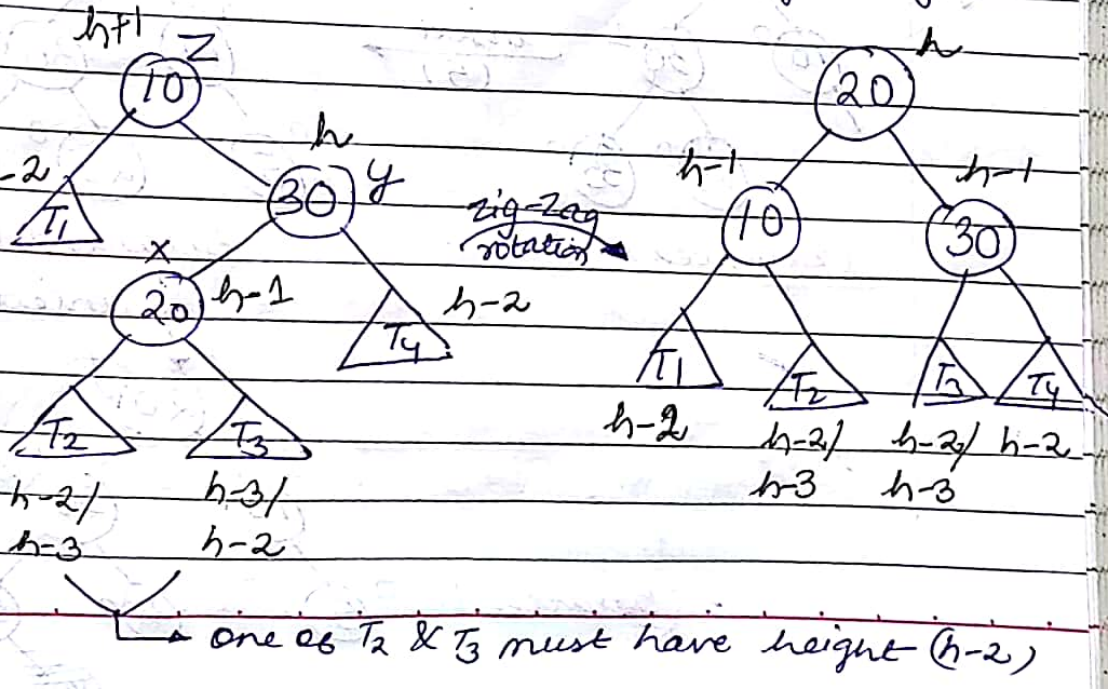
but, ht of any binary tree = $\Omega(\log n)$

$\Rightarrow$ ht of AVL tree = $\theta(\log n)$.

- Rotations :-
  - ↳ height of (parent) node = 1 + max(ht of children)
  - ↳ if during insertion/deletion, the property of AVL is violated for 1st time, stop there.
  - ↳ Perform rotations and stop if there is no change in height of a particular node.
  - ↳ for this, keep a trace of 2 previous nodes.
  - ↳ Rotation amongst z, y and x will be performed at any time when ht(z) ≥ 2.

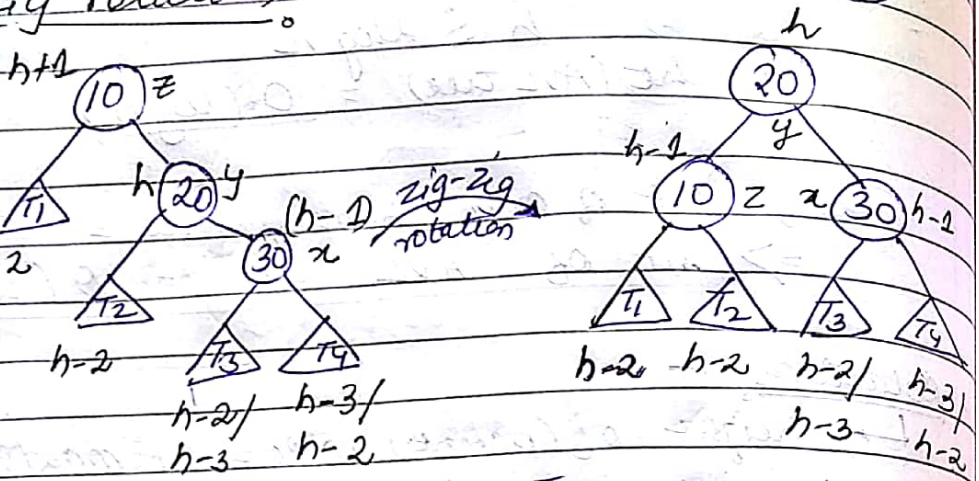
(i) Zig-Zag rotation :- (left-right and right-left)

eg:-



↳ one of T_2 & T_3 must have height (h-2)

(2) Zig-Zig rotation :- (LL or RR rotation)



↳ height of tree also decreases by 1

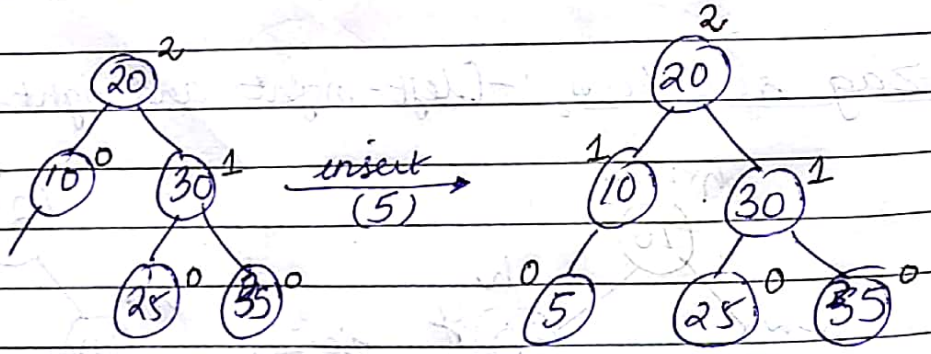
↳ Zig-Zag & Zig-Zig rotations are Binary Search Property preserving rotations.

↳ In both above problems,

$z \leq x \leq y$

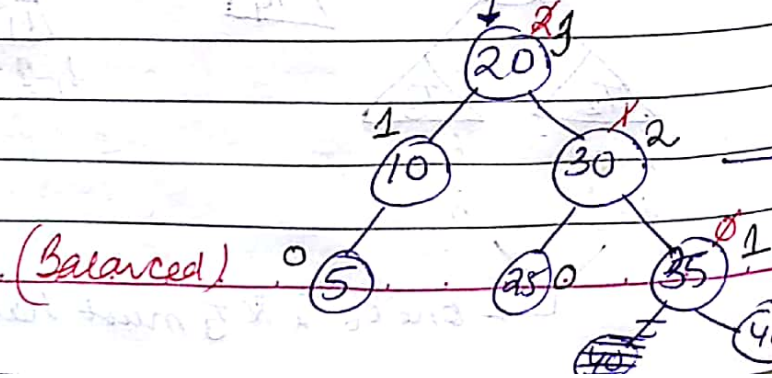
and violation of AVL property is because of 'z'.

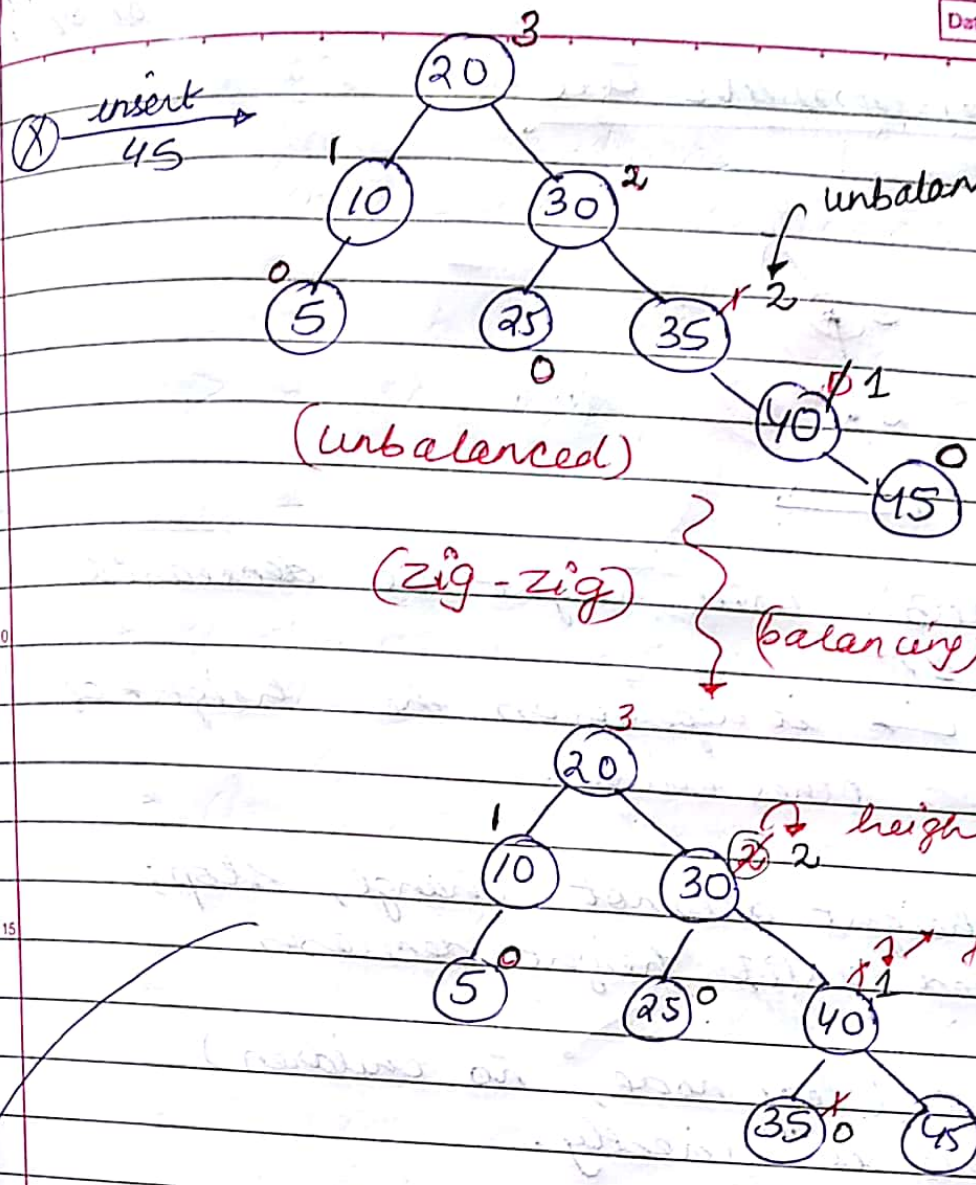
eg:-



(Balanced) (Balanced)

insert 40





↳ stop whenever parent's height stop changing.

↳ In zig-zig total pointer changes when 45 was inserted :-

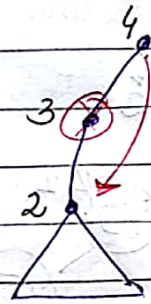
parent pointer of y
pointer of z
pointer of T_2
} 3 parent pointer changed.

children pointer 1+1 = 2 ⇒ 5 pointer changes.

So, total complexity of add operations = $\lambda \log n$.

Deletion from AVL Tree

eg:

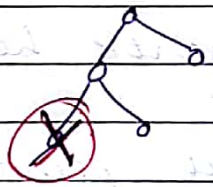


eg:- delete(3). will height be decreased by 1?

↳ it depends on the height of other child.

↳ If height does not change, stop; Proceed until height decreases.

(i) Case 0 :- (leaf node, no children)
delete it directly.



satisfy the property of AVL tree

↓

Not required!

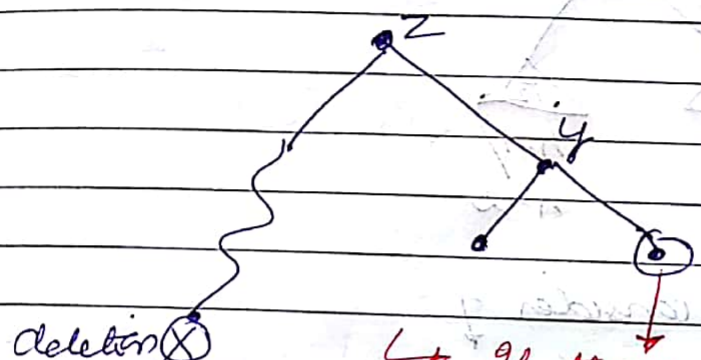
(ii) Case 1 :- (one child)
delete the node, make its parent point the child of target.

(iii) Case 3 :- (two children)
replace the node with its inorder predecessor / successor and recursively delete that node.

↳ here we also need to maintain the property of AVL tree & also should balance until the height decreases.

$H(z) = 1 + H(y)$

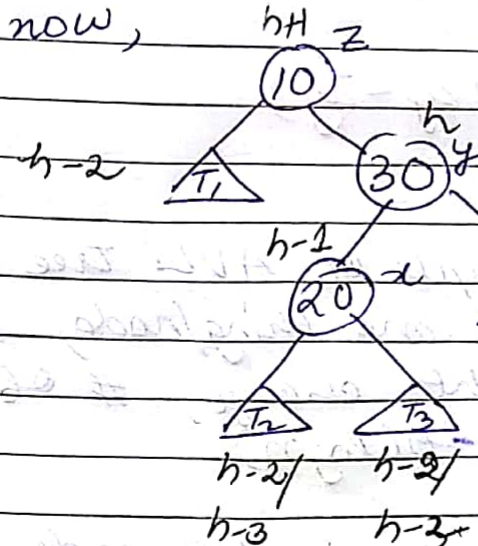
choose (X) such that $H(y) = 1 + H(x)$



↳ If there is a difference in height, choose one with more height.

↳ If there is same height choose one which causes less # of pointer changes (zig zig)

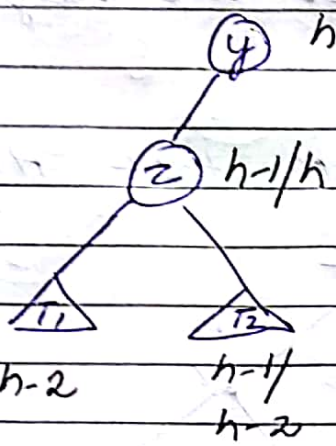
So now,



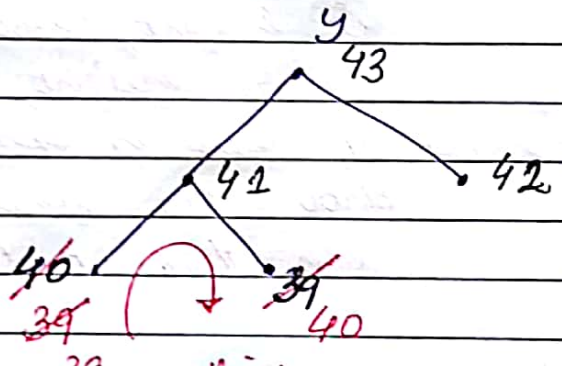
can be of height h-1 or h-2.

It can't guarantee height decreasing or changing. It may remain the same.

Now, in zig zig, if T2 = h-1/h-2 then



↳ So, consider ↓



- Homework :-

Construct an example of AVL tree where whenever I delete one thing/node, so every time height change, # of operations I do is O(log n)

↳ a tree with min # of nodes with height 'h' is an example.

↳ there are a lot of other trees with same problem.

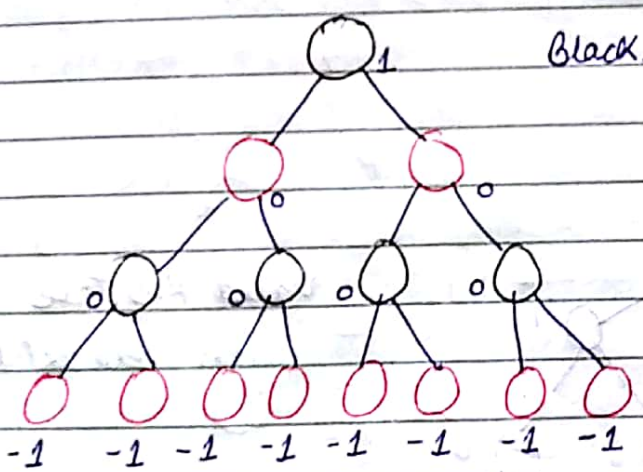
LECTURE -16

- **RED BLACK TREES :-**
  - ↳ every node is either red/black.
  - ↳ root = black (always)
  - ↳ If a node is red, its children are black.
  - ↳ Every path from a node (ROOT) to NULL contains same # of black nodes.
  - ↳ The black height of a node is a RB Tree is the # of black nodes on any path to a NULL node.

ensures ht = O(log n)

↓

does very few changes as compared to AVL

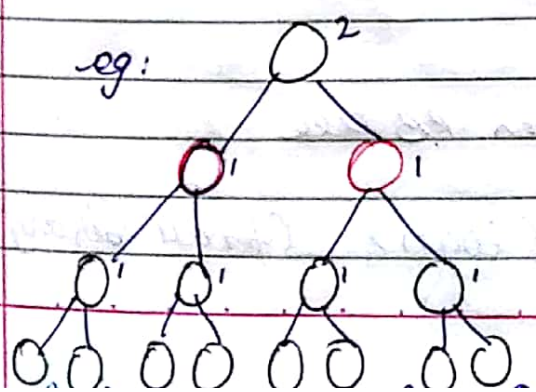


Blackht(n) = 1 + max(Blackht(n→lc), Blackht(n→rc))

↓ for above eg, Blackht = 1 valid

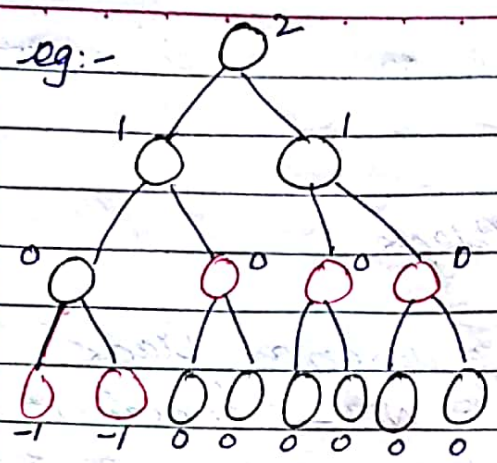
↑ A valid RB Tree

eg:



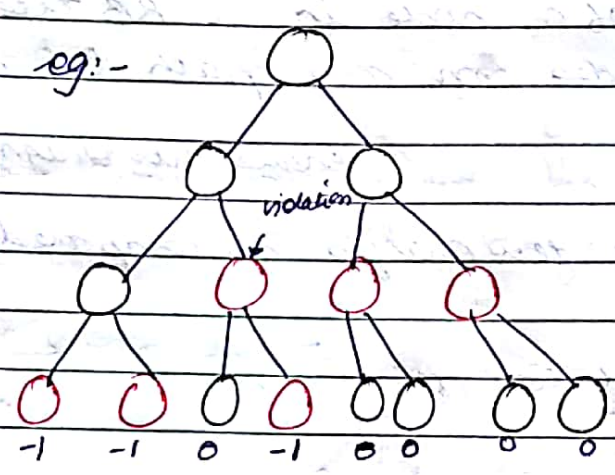
Black height = 2.

eg:-

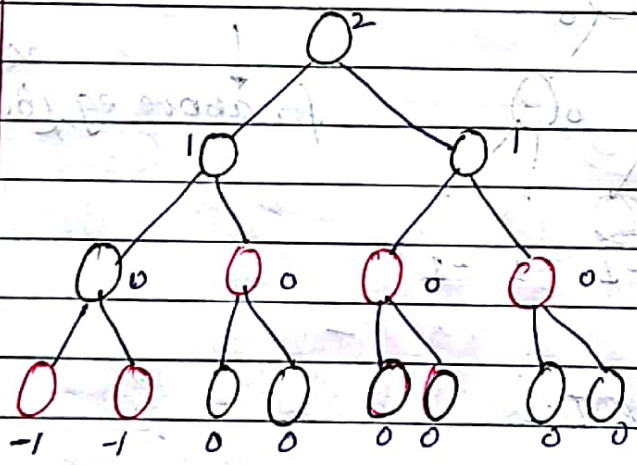


Valid RB Tree with black height = 2.

eg:-



✗ not a valid RB Tree.

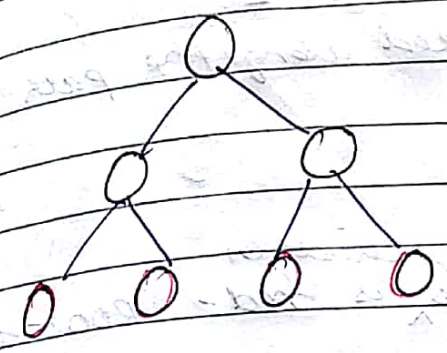


Valid RB Tree with Blackht = 2.

→ Let n_1 = # of nodes in RB Tree
Black height = bh
height = h $= \Omega(\log n)$ (for all binary trees)

↳ Red Black tree with min # of nodes whose black height is bh

$$n \geq \sum_{i=0}^{bh} 2^i = 2^{bh+1} - 1$$



↳ consider all black nodes,

then $n \geq 2^{bh+1} - 1$

now, $\frac{h}{2} \leq bh \leq h$

If I take any path from root to leaf, # of red nodes = atmost # of black nodes.

no. of red nodes ≤ no. of black nodes

now, if c = black − red ∵ >0 (along the path)

h = # of black + red.

#red ≤ # of black nodes

$ht \leq 2 *$ black nodes

black ht $\geq h/2$

$n \geq 2^{h/2+1} - 1$

$2^{h/2+1} \leq n+1$

$\log(n+1) \geq \frac{h}{2} + 1 \Rightarrow h \leq 2\log(n+1) - 2$

$h = O(\log n)$.

↳ 3rd & 4th properties are crucial to argue this.

4th property → $n \geq 2^{bh+1} - 1$

3rd property → b ≥ red along the path.

- Insert

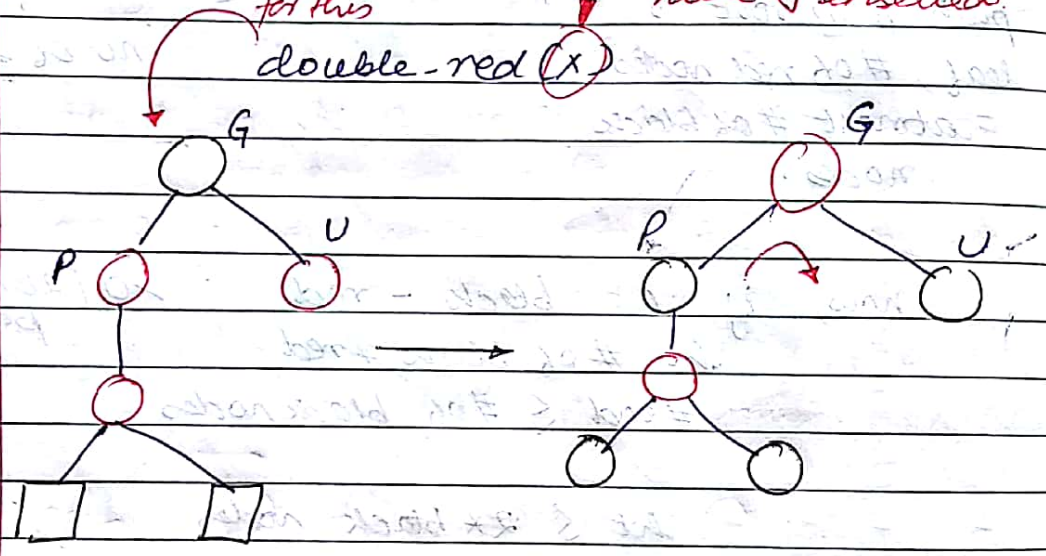
↳ as usual in BST

↳ insert in root node's children as red. Root should be black.

↳ If the node is red, its children should be black.

↳ Every path from node to descendant leaf must contain same # of black nodes.

↳ If parent is red → call double red.

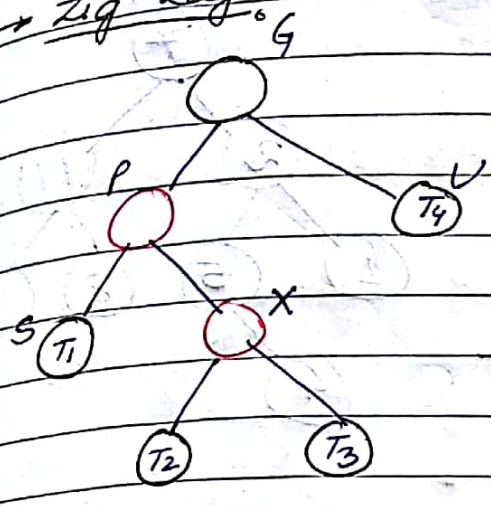


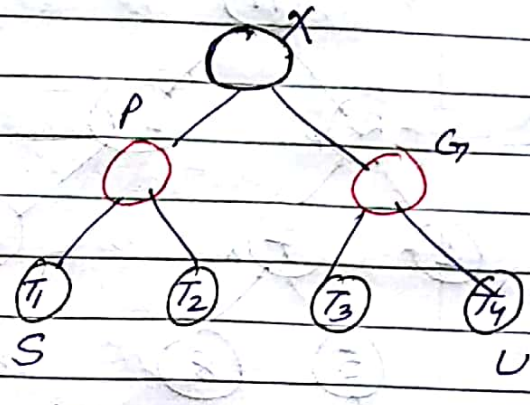
Case 1 :- color of uncle = 'Red'.

G → maybe if parent is red, then Grandparent is also red.

call double(G) with 'G' as 'x'

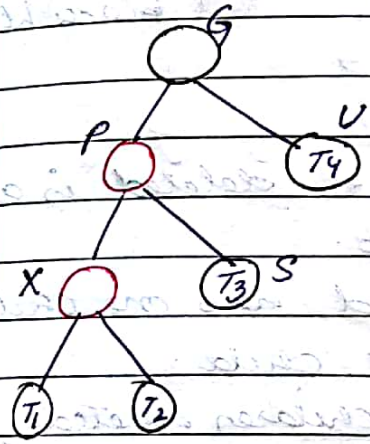
↳ zig zag :-

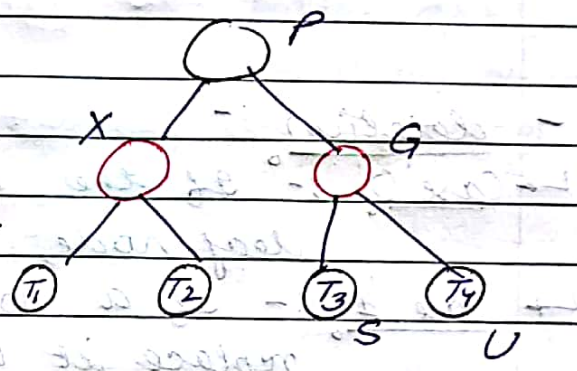




double red overcomes.

↳ zig zig :-

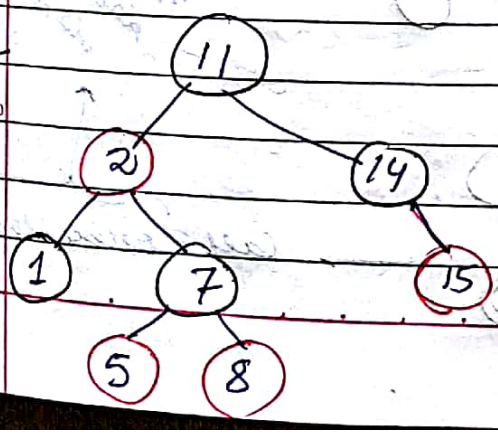




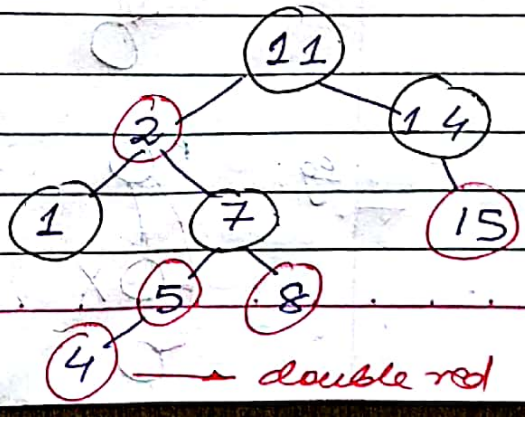
So, double Red takes X,

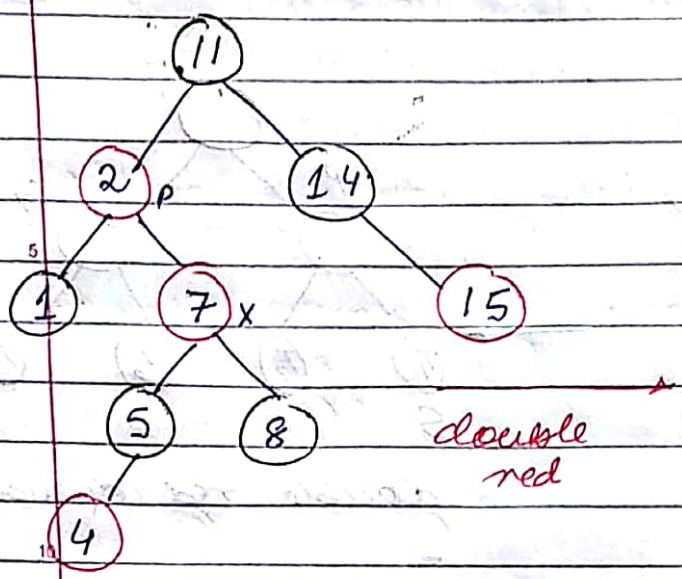
P = parent (X)
G = grandparent (X)
U = Uncle
S = Sibling

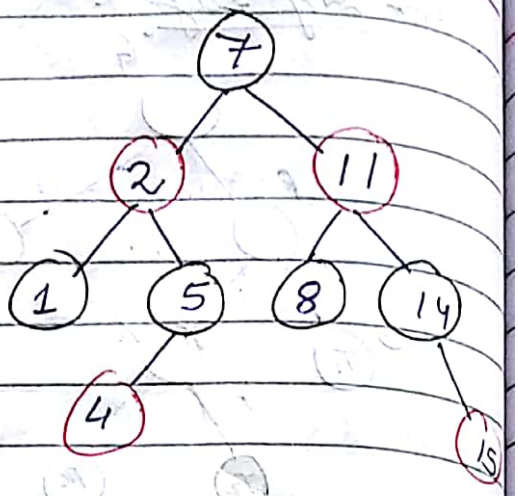
eg:-



insert 4







(parent always become black)

in above tree, on any path, 2 nodes are black
↓
black ht = 1

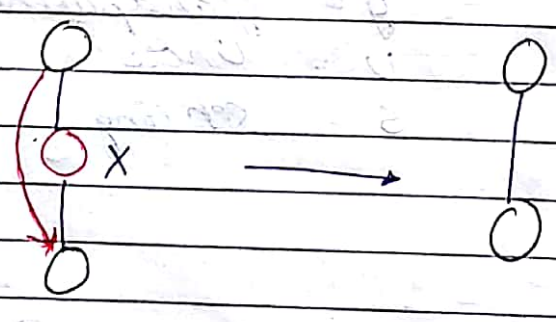
deletion :-

↳ Case '0' :- If the node to be deleted is a leaf node, delete it.

↳ Case '1' :- If a node deleted has one child, replace it with that child.

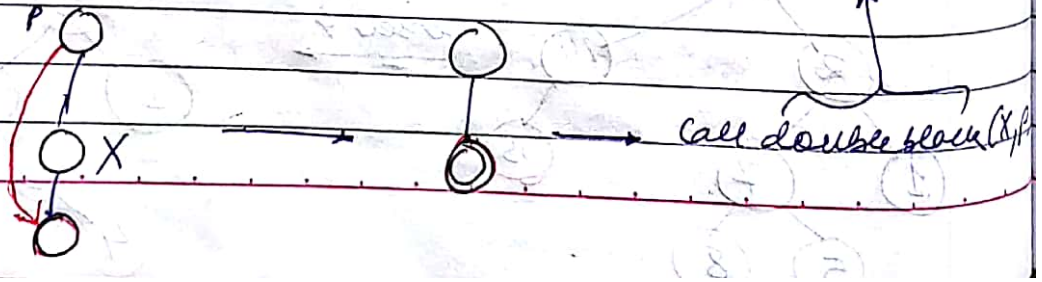
↳ Case '2' :- If vertex with 2 children deleted, replace the value by its' inorder predecessor then delete the inorder predecessor recursively.

eg:-



node was already black & assigned black color again.

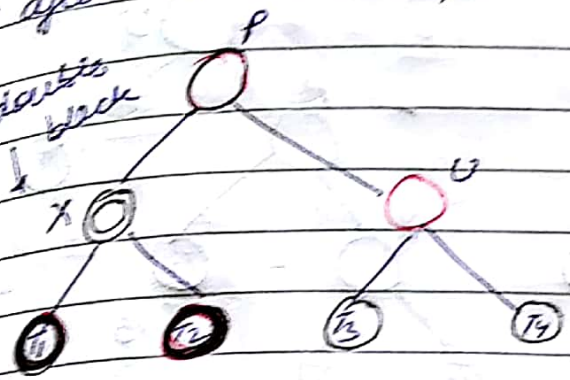
eg:



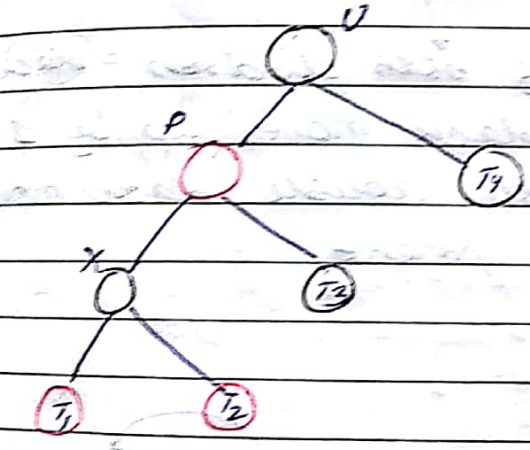
→ need to send pointer to 'p' also because 'x' can be a NULL pointer, then I get pointer to 'p'.

→ If I'm deleting a leaf node, to, I need x as well as the (parent) function.

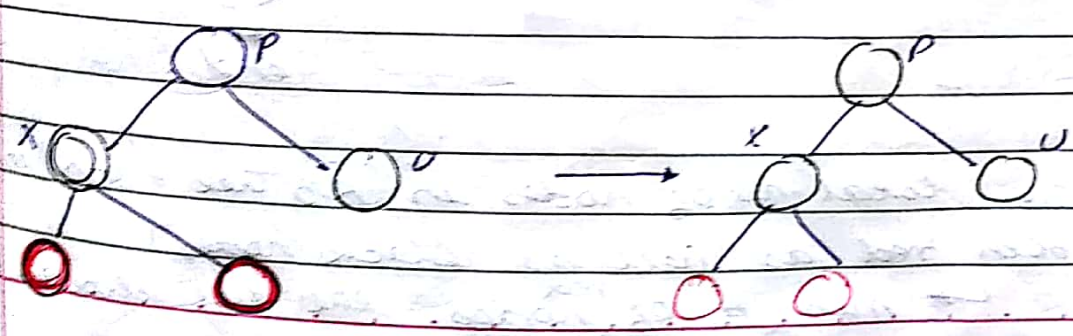
→ after deletion,



x = double black,
u = Red ;
then
change uncle to black applying reverse zig zig



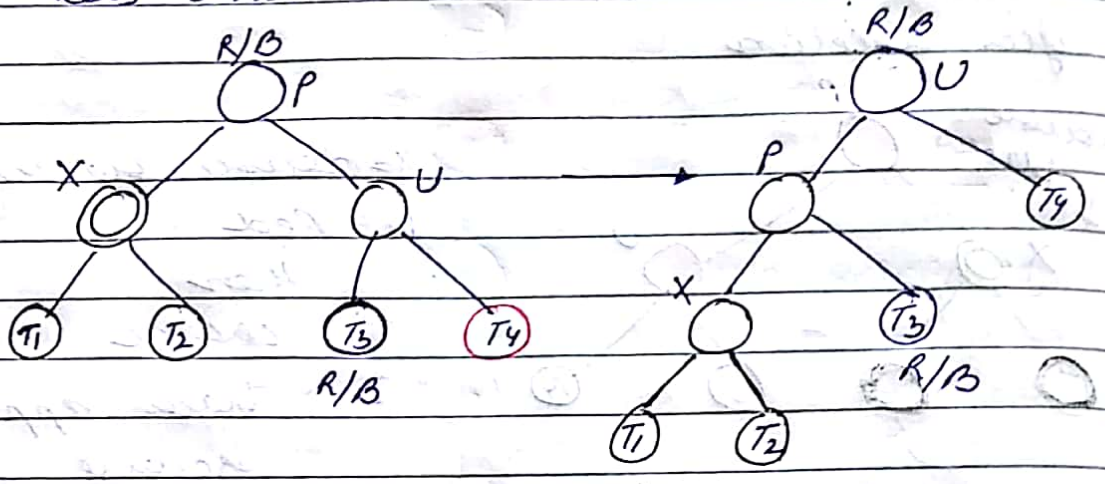
if, I don't no colour of 'p', black goes up.



↳ if, 'p' was already black, and again colouring black, it becomes double black again.

↳ If 'p' was red, & it becomes black, fine.

↳ Let's assume one is not black.



→ If uncle is red, inverse zig zig, uncle gets black.

If both children = black, black moves up by 1, so double black moves to parent.

Case 1 } call double
Case 2 } black again

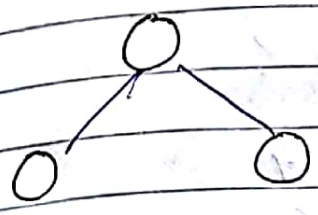
Case 3 : one zig zig rotation

Real Worst :- Case 3 → Case 2 → Case 1 → end.

Eg :- The total no of nodes in RB Tree = 100 (including both red as well as black nodes).

↳ min. # of red nodes the tree should have

$2^6 = 64$, $2^7 = 128$



→ 1 node

→ 2-node

→ 32 nodes

→ 37 nodes (all red)

↓ percolate up

$\lceil 37/2 \rceil$ = 19 nodes → 0

↓ percolate up

$\lceil 19/2 \rceil$ = 10 nodes

↓ percolate up

$\lceil 10/2 \rceil$ = 5 nodes

↓ percolate up

⇒ 3 nodes of red color required.

⋮

Found $2^k - 1 \leq n$

$n - 2^k + 1$

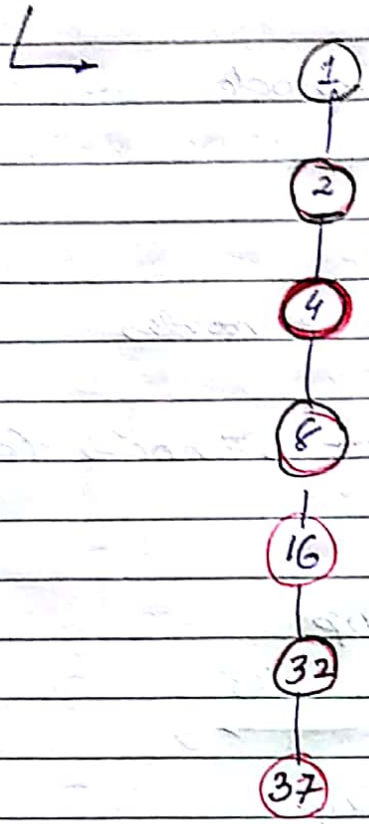
↓

# of 1's in this binary

⇒ # of 1's in binary representation of 37 (leaves).

So, maximum # of black nodes = 37.

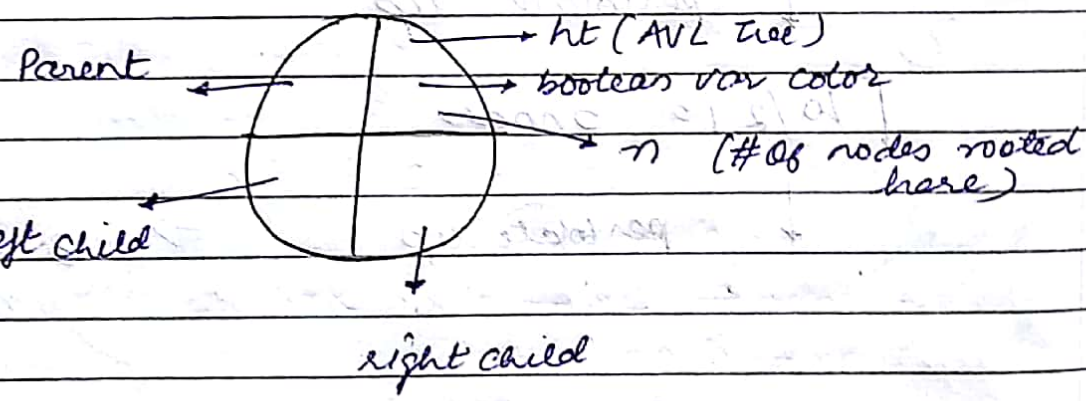
now, max # of red nodes ?



4
16
37
57 ← Max red nodes
= 57.

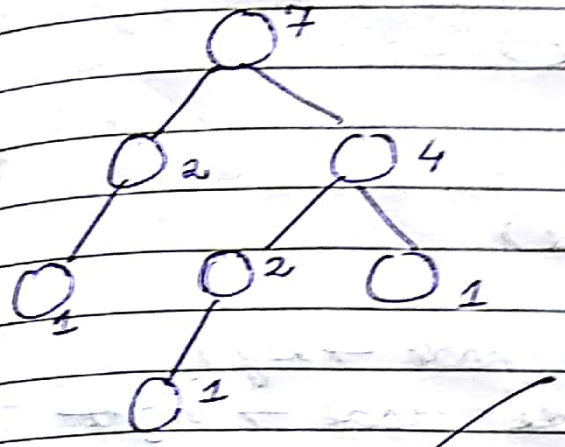
min black = 43.

- BBST's :-



AVL, RedBlack } search, insert, delete O(log n)
rank(x) = ?

eg: of 'n' → # of nodes rooted here



leaf = 1.

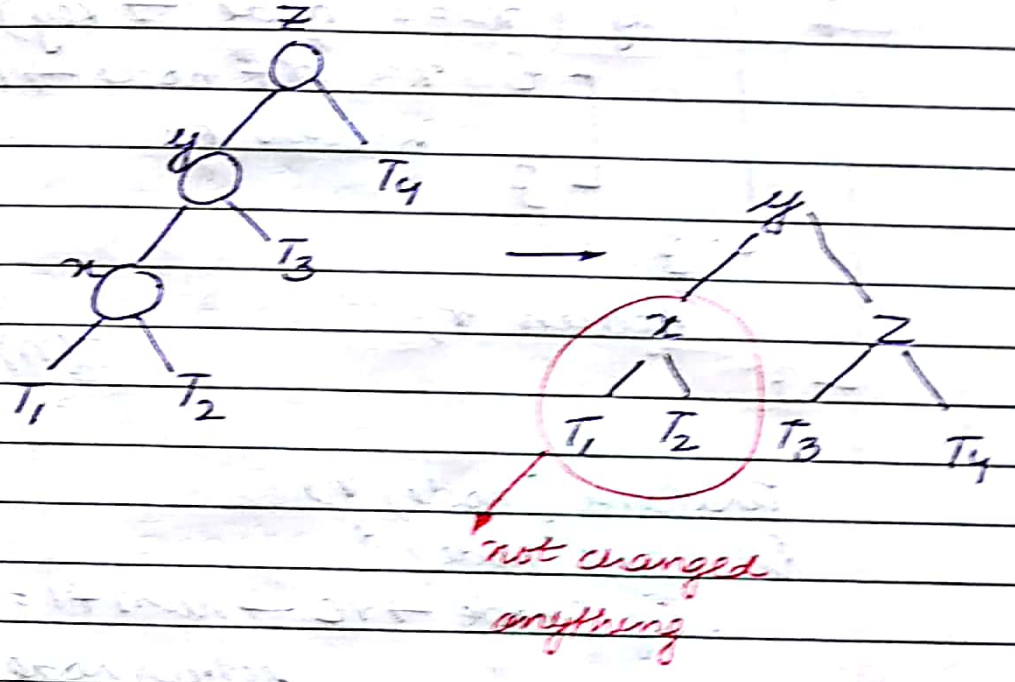
{ insert & delete properly →
# of nodes in LHS + # of nodes in RHS + 1

↓

change where rotation happens

If not, then O(n).

↳ zig zig :-



↳ zig zag :- There is change!

# of nodes in count in subtree when insertion happens

↓

follow the path, keep increasing

find Rank (x) = # of nos > x.

↳ Rank :-

```
Rank (X, node)
{
    r = 1.
    while(node)
    {
        if (x > node → key)
            node = node → right
        if (x < node → key)
        {   r = r + node → right → num + 1;
            node = node → left;
        }
        if (x == node → key)
        {   r = r + node → num;
            return r;
        }
    }
    return r;
}
```

T(n) = T(search) = O(log n)

```
FindRank (node, r)
{ while(node) {
    if (node → rc → num + 1 == rank)
                return node;
    else if (rank < node → rc → num + 1)
                node = node → rc;
    else { r = r → (node → rc → num + 1)
           node = node → lc
    }
}
```

T(n) = T(ht) = O(log n)

LECTURE-17:

Applications of BBST.

↳ Intervals :-

[1,6] [3,20] [7,18] [14,21] [19,22] [26 42] [24,32]

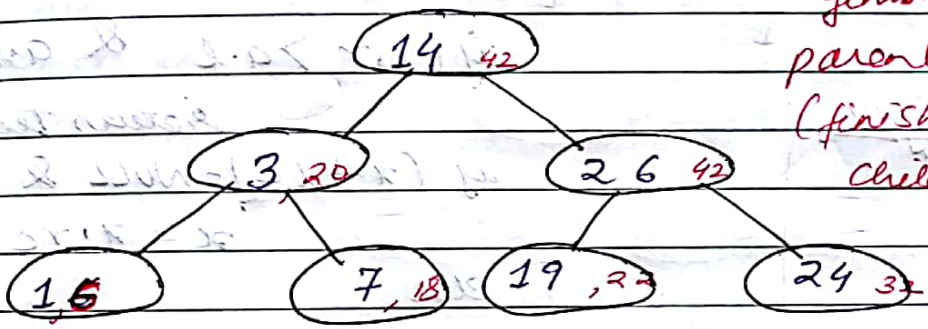
median

interval is an object — a (a.l, a.r)

↳ BBST (AVL or RB) with intervals is called an interval tree.

↳ max = max element amongst all right end points.

↳ don't worry about finish time, create tree with starting time



finish time of parent = max (finish time of children, itself)

↳ Law of Tricolomy :-

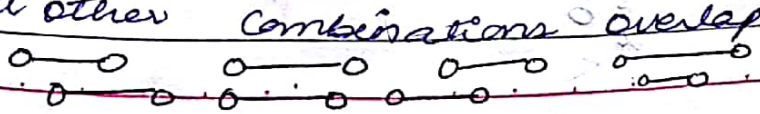
a.r < b.l — a — b

↳ 'a' should be to the left of 'b'.

b.r < a.l — b — a

↳ 'b' should be to the left of 'a'

all other combinations overlap.

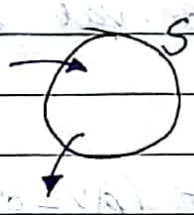


a.r ≥ b.l && b.r ≥ a.l

$x.max = max \begin{cases} x.r \\ x \to lc \to max \\ x \to rc \to max \end{cases}$

this can be accomodated by augmenting an AVL Tree.

- Dynamic Data Set :-



O(logn) ← Add(x)
O(logn) delete(X)
overlap(X) → true
→ False

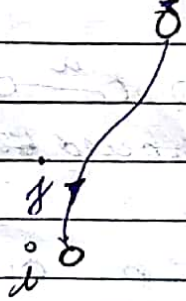
↳ pseudo code :-

node in a BBST
a.l, a.r (interval

```
overlap(x, a)
{   while(!X)
    {
        if(x.r > a.l & a.r > x.l)
            return true
        if (x→lc != NULL & x→lc→max < a.l)
            x = x.rc
        else
            x = x.lc
    }
}
```

O(logn)

↳ Consider the case :-



j is the one with max end point

$j.r = x \to lc \to max$

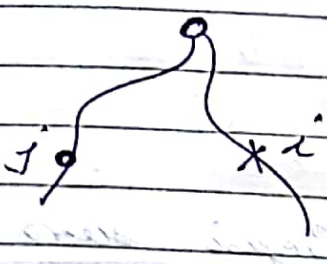
$x = x \to rc$ when $j.r < a.l$

$i.r \leq j.r < a.l$

$\hookrightarrow i.r < a.l$

so I cannot overlap with 'a'

↳ Thus, going RHS means no overlap on LHS and vice versa.



$x = x \to lc$

means $j.r \geq a.l$

'a' does not overlap anything on LHS so it should not overlap with 'j'.

↳ In that case, j.l and a.r should compare as

$a.r < j.l < i.l$

$\to a.r < i.l$

then 'a' cannot overlap with 'i'

↳ for 'i' & 'a' to overlap,

$i.r \geq a.l$ & $a.r \geq i.l$

↳ Consider dynamic dataset again:-

- add(x)
- delete(x)
- search(x)
- rank(x)
- findrank(r)

$\log_2(n)$

- sum of #s between l & r
- no. of #s between l & r

↳ # of nos. between l & r

$$\text{find}_{\text{rank}}(r) - \text{find}_{\text{rank}}(l) \pm 1$$

± 1 → depends on boundary condition you've used

$O(\log n)$ $O(\log n)$

$O(\log n)$.

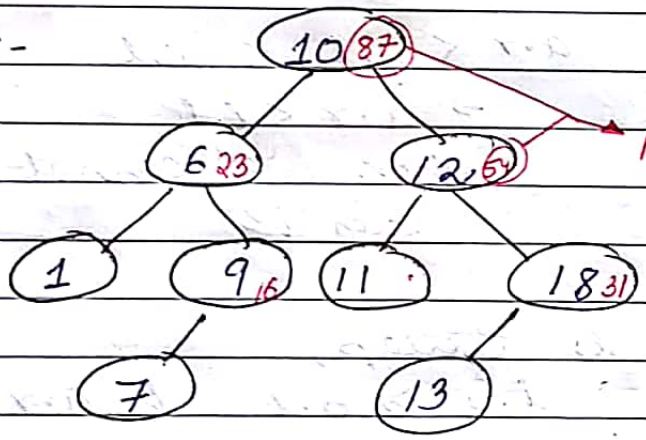
↳ sum of #s → Prefix sum (PS)

$PS(i)$ = sum of all the #s $< i$

if $O(\log_2 n)$, then $sum(l, r) = PS(r) - PS(l)$

$O(\log n)$

eg:-



Keep a track of sum of the key values of children + node itself.

→ key (node also stores sum)

```
PS(X, i)
{ sum = 0
  while (x)
  {
    if (x → key == i)
        return sum + x → lc → sum;
    if (x → key > i)
        x = x → lc
    else {
        x = x → rc;
        sum += x → lc → sum + x → key }
  }
} return sum;
```

$O(\log_2 n)$

↳ So, now, average of all #s b/w l & r

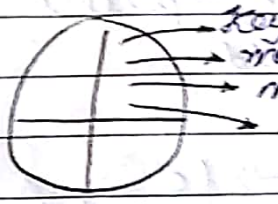
$$= \sum_{l}^{r} i \;/\; (r-l)$$

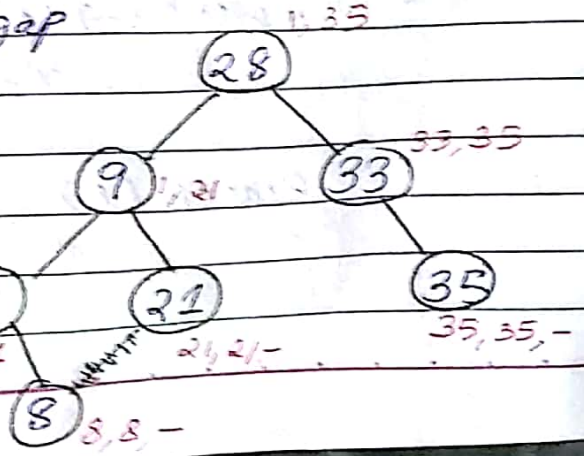
eg:- 1, 8, 9, 21, 28, 33, 35.

- add(x)
- delete(x)
- search(x)

$O(\log_2 n)$

- mingap(x) → $O(1)$ = ?? how?
- maxgap( ) → root.max − root.min, $O(1)$





$mingap(X) = \begin{cases} x \to key - x \to lc \to max \\ x \to rc \to max - x \to key \end{cases}$

minimum of 4

if not in the mingap?

$x \to mingap = min \begin{cases} x \to lc \to mingap \\ x \to rc \to mingap \end{cases}$

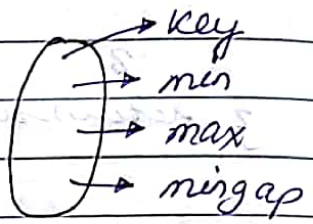
→ maintain/update mingap after add & delete

- add (x)
- delete (x)
- search (x)

} $O(\log_2 n)$

- mingap(l, r) } $O(\log_2 n)$
- maxgap (l, r)

→ very important.



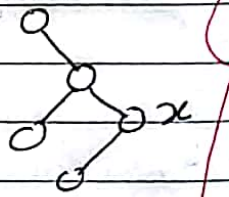
l r

→ [10 32] → min gap between

smallest # ≥ l

greatest # ≤ r

```
while(x -> parent -> rc != x)
        x = x -> parent.
x = x -> right child
while (x -> lc)
        x = x -> lc
```

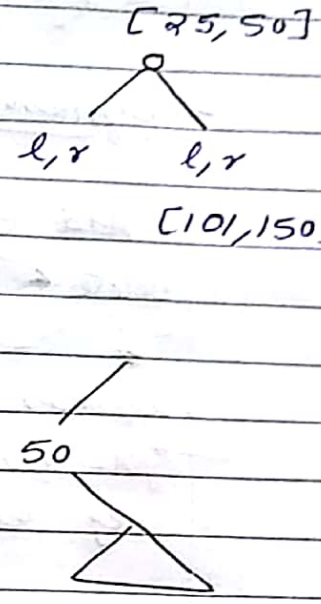


Successor for updation & deletion

search (l) , search (r)

PC 30

```
mingap(x, l, r)
{
  while(x) {
    if (x->key > r)
      x = x->lc
    else if (x->key < l)
      x = x->rc
    else {
      mingap = ∞
    }
  }
}
```



LECTURE-18

- Dynamic Dataset Problem :-

Add
Delete
depends on which one you delete.
eg: Queue, Stack

↳ Queue :- (FIFO) enqueue, dequeue
↳ Stack :- (FILO/LIFO) push, pop.
↳ Priority Queue :- → priority
→ enqueue (x, p)
→ dequeue () → delete the element with max/min priority.

Datastructure
↓
BBST
↓
key = priority → O(logn) for enqueue & dequeue.

↳ But BBST does lot more than add & delete, so using it is not right thing.

- Binary Trees They help us to do this in O(logn)
Union → priority queue requires this.
If we have BBST, this cannot be done efficiently.
O(n logn)

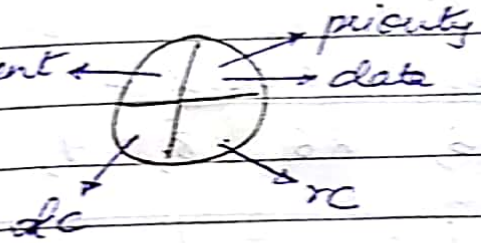
↳ We have 2 Priority Queues. find union of them
Binary Tree → O(n). XX for more than O(logn) we had earlier.

	Union
Binary Heap	O(n)
Binomial Heap	O(logn)
Fibonacci Heap	O(1)

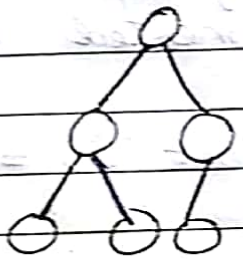
→ Queue & stack are special cases of priority Queue.

• Binary Heaps :- (min default)
Binary tree is associated with it; with two properties
i) Heap property.

x.priority ≥ (x → parent).priority
this should be be true ∀ x in the tree.



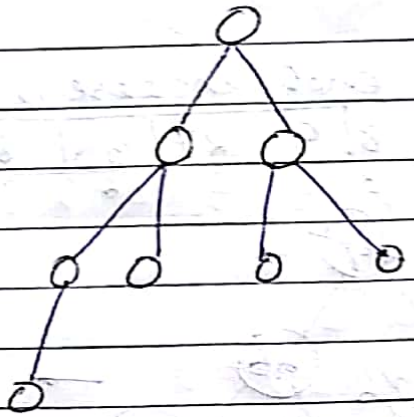
(ii) Complete Binary Tree :- Perfect tree at every level except last level. Last level filled from L to R.
eg:



$h = \Omega(\log n)$

→ Height of Complete Binary Tree (CBT) = $\Theta(\log_2 n)$

→ given, n (# of nodes), h (ht of tree)
(i) fixed 'h', min # of nodes?



till 'h^2-1' level,
nodes = 2^h
at last level, only
1

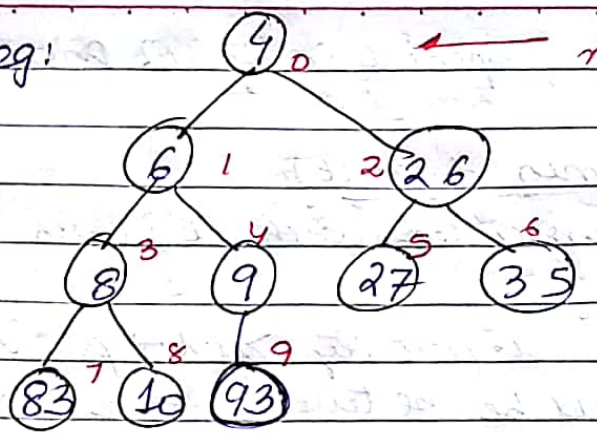
$$n \geq \sum_{i=0}^{h-1} 2^h + 1$$

$$n \geq 2^h - 1 + 1$$

$$n \geq 2^h$$

$$\Rightarrow h < \log(n).$$

eg:



← minimum element at root node.

maximum element has to be in one of the leaves.

↳ Level order traversal :-

4, 6, 26, 8, 9, 27, 35, 83, 10, 93

A	4	6	26	8	9	27	35	83	10	93
i	0	1	2	3	4	5	6	7	8	9

it means, Binary heap can be implemented through array

given, i

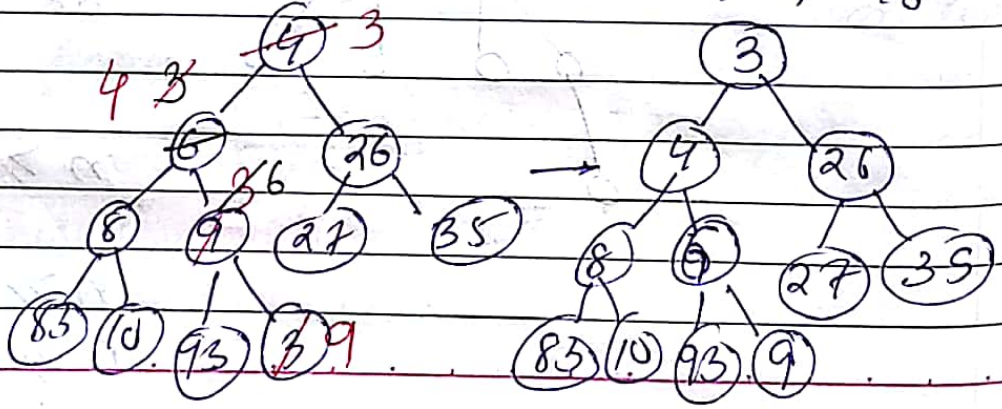
- parent $(i) = \lfloor i/2 \rfloor$
- left child $(i) = \lfloor 2i+1 \rfloor$
- right child $(i) = \lfloor 2i+2 \rfloor$

- Insertions :-

↳ Insert 3 :-

↳ insert at end, increase n by '1' $A[n+1] = x$

A	4	6	26	8	9	27	35	83	10	93	3
	0	1	2	3	4	5	6	7	8	9	10



A [3 | 4 | 26 | 8 | 6 | 27 | 35 | 83 | 10 | 93 | 9]

↳ Bottom up heapify :-

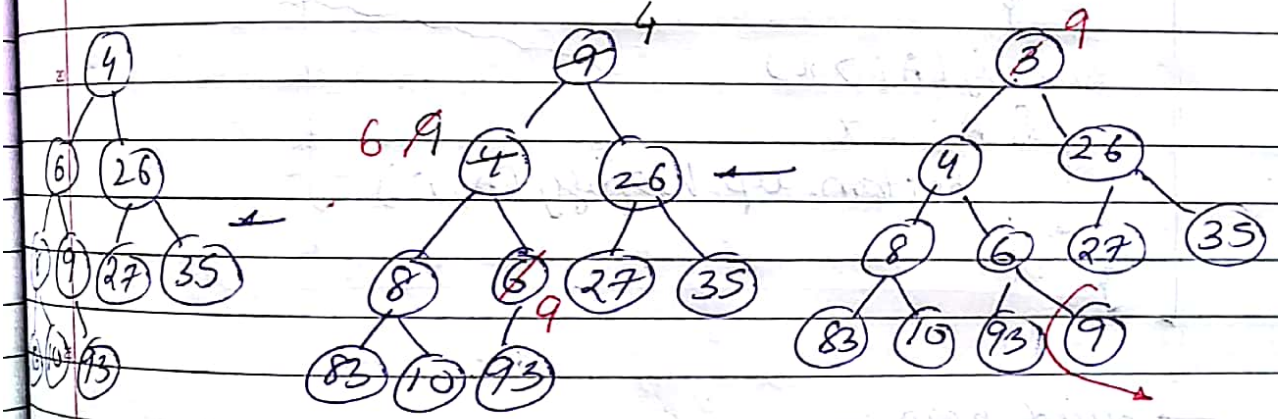
O(log n).

Add(A, x)
{ A[n++] = x
Bottom-up-heapify(A, n-1)

Bottom-up-heapify(A, i) → p = (i-1)/2
{ while(A[i] < A[p] & i > 0)
swap(A_i, A_p)
i = p;
p = i-1/2
}

↳ delete min :-

↳ replace last node with root node,
↳ delete last node, decrease n by 1.
↳ top down heapify.



top-down-heapify(A, i) ← O(log n). delete_min(A)
{ lc = 2i+1, rc = 2i+2
while (rc < n)
{ if (A_lc < A_rc) l = lc
else l = rc
if (A_i < A_l) return;
else { swap(A_i, A_l)
i = l; lc = 2i+1;
rc = 2i+2; } } if (lc < n) if (A_i > A_l) swap(A_i, A_l); }

{ A[0] = A[--n];
top_down_heapify(A, n)
}

→ Operations → Add (A, x) O(log n)
→ delete-min() O(log n)

• decrease_key (i, x) → $A_i > x$ (index)
{ A[i] = x // eg:- 26 → 16
bottom_up_heapify (A, i)
}

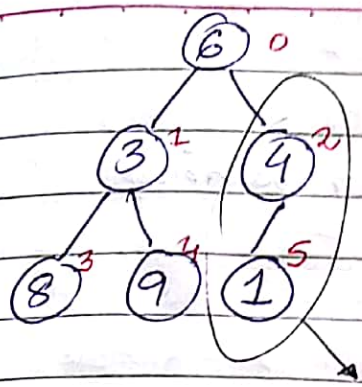
• increase_key (i, x)
{ A[i] = x
Top_down_heapify (A, i)
}

• change_key (i, x)
{
if (A_i < x)
{ A_i = x
Top_down_heapify (A, i)
}
else if (A_i > x)
{ A_i = x
Bottom_up_heapify (A, i)
}
}
O(log n)

→ Build heap :-
eg: B = 6 3 4 8 9 1

for i = 0 to n
Add (A[i])
} O(n log n)

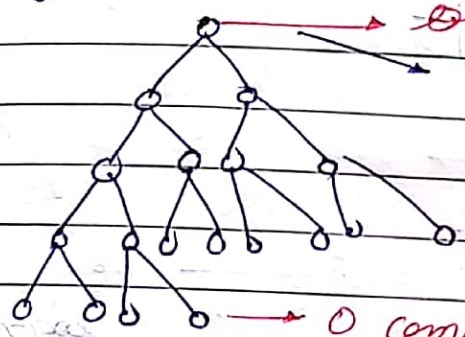
n add operations (log n each)



↳ store in 'A' → CBT is satisfied
↳ all leaf node satisfy heap property
↳ go to first non-leaf and start first subtree

for (i = n/2 to 0) → calling n/2 times
    topdown heapify (A, i) → log n

↳ Analysis :-



→ only this node do log n comparison
ht = h → Comparisons = 'h'
ht = 1 → Comparison = 1
→ 0 comparison ht = 0 → Comparisons = 0

at ht h → max # of nodes = 1 → $(n/2^{h+1})$
h-1 → 2
h-2 → 4
i → n/4 (almost)
0 → n/2 (almost)

∴ Build Heap :-

$$\sum_{h=1}^{\log n} h \cdot \frac{n}{2^{h+1}}$$

leaf nodes → we don't do anything

$$\leq \frac{n}{2} \sum_{i=0} \frac{i}{2^i} \quad \left(\sum i x^i, x<1 \right)$$

but $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$

differentiate on both side

$$\sum i x^{i-1} = \frac{1}{(1-x)^2}$$

multiply both sides by i

$$\sum i x^i = \frac{x}{(1-x)^2}$$

So $\sum 1/2^i = \frac{x}{(1-1/2)^2}$

↳ So,

0 00 0000 ——— 000...0

0 1 2 logn

but, topdown-heapify costs less.
bottom up heapify costs more
Ω (log n)

↳ add(), delete-min(), change-key(i, x), Build-heap

O(logn) O(n)

min() → O(1)

↳ union of 2 heaps :-
↳ put both heaps in an array & call build heap $O(n_1+n_2)$

HEAP SORT :-

Given a sequence of #s, $a_0, a_1 \ldots a_{n-1}$
called build heap and delete min $(n-1)$ times.

Build_heap (A, n) → $\theta(n)$
for i = 0 to n → $\theta(n)$
  delete_min(A) → $\theta(\log n)$
  ↳ store in last node, as it is free now.

$\theta(n \log n)$.

↳ I had Binary tree representation of array, but now I actually store it as complete Binary tree.

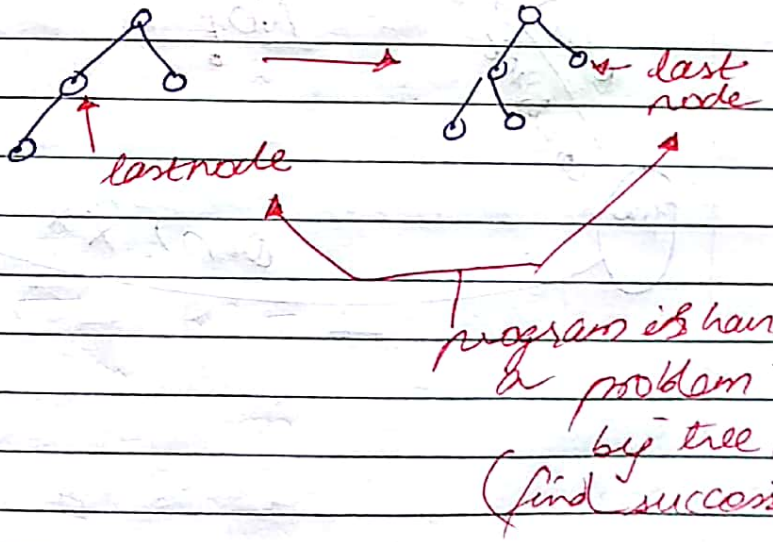
Can I still do all these operations?

↳ In case of array, I know exactly where to insert/delete (last node).

↳ I don't know where to add now?

Soln → first now, maintain an extra pointer for last node, which points to where I want to add/delete

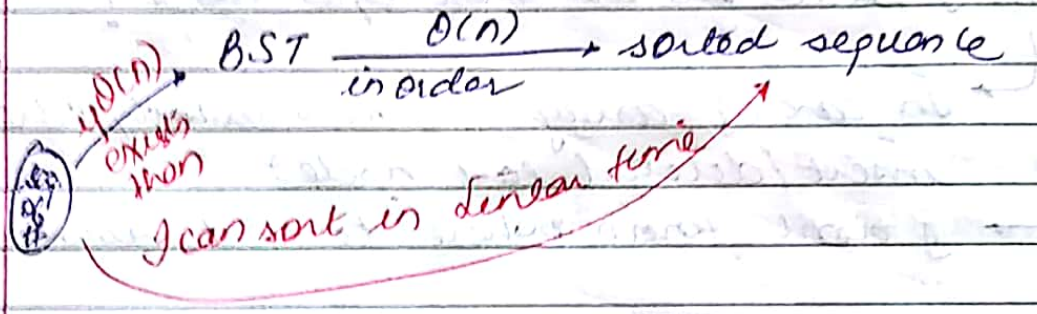
eg.



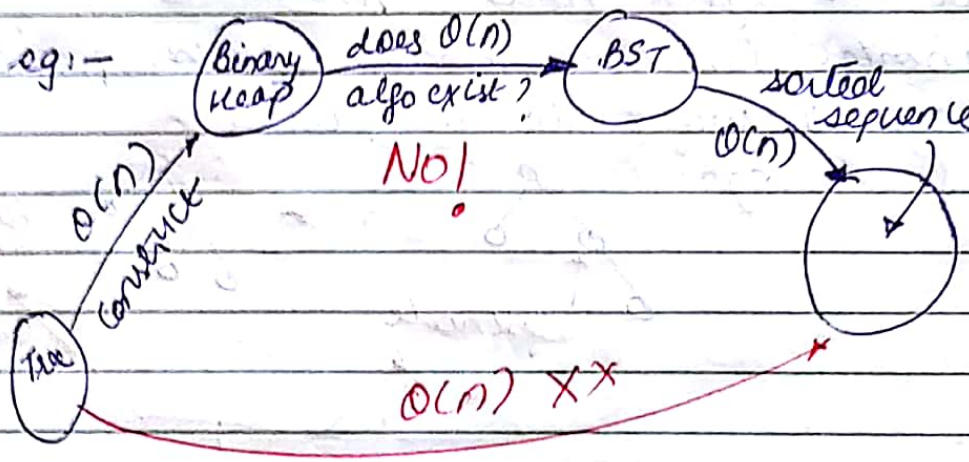
→ BBST → deletemin → O(log n) } much easier
add → O(log n) } implement

eg:- Given a sequence of ITs, I can build a heap in linear time.
Can I build a (B)BST in linear time?
(is there an algorithm)
[degenerate tree → O(n)
BBST → O(n log n)

No, I cannot build a BST in O(n) because comparison based sorting takes O(n log n)



eg:-



LECTURE-19

↳ stack :- push (x) → O(1)
pop (x) → O(1)
multipop (k) → O(k) → O(n) × n times = O(n²)

#of operations this algo does if we start with n=0 is 2n.

operation	amortized cost	actual cost
push	2	1
pop	0	1
multipop	0	k

max(k) = O(n) if we start with empty stack.

↳ amount of money at any time ≥ 0, so, I'll not starve.

↳ Basic idea behind amortized cost is that, we can afford to increase the cost of cheaper operation, but we need to decrease the cost of ~~sig~~ costlier operation significantly.

↳ How to get amortized cost?
define a potential function

$$\phi : D \longrightarrow \mathbb{R}^+$$

D → Data structure

$\mathbb{R}^+$ → maps to real positive values.

then, at i^{th} iteration.

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

$\bar{C}_i$ → amortized cost

C_i → actual cost

$\phi(D_i)$ → potential after operation

$\phi(D_{i-1})$ → potential before operation

potential difference can be negative.

$$\bar{C}_1 = C_1 + \phi(D_1) - \phi(D_0)$$
$$\bar{C}_2 = C_2 + \phi(D_2) - \phi(D_1)$$
$$\bar{C}_3 = C_3 + \phi(D_3) - \phi(D_2)$$
$$\vdots$$
$$\bar{C}_i = C_i + \phi(D_i) - \phi(D_{i-1})$$
$$\vdots$$
$$\bar{C}_n = C_n + \phi(D_n) - \phi(D_{n-1})$$

$$\sum_{i=1}^{n} \bar{C}_i = \sum_{i=1}^{n} C_i + \phi(D_n) - \phi(D_0)$$

→ if $\phi(D_n) \geq \phi(D_0)$

$$\sum C_i \leq \sum \bar{C}_i$$

$\phi(D_n) \not\geq \phi(D_0)$

eg: $C_i = \{0, 2\}$

$$\sum \bar{C}_i \leq 2n$$

→ upper summation

↳ finding amortized cost for stack above:-

ϕ = # of elements in the stack.

$\phi(D_0) = 0$ (empty stack)

push → actual cost + $\phi(D_{i+1}) - \phi(D_i)$

$1 + 1 - 0 = 2$

push = 2.

pop → actual cost + $\phi(D_{i+1}) - \phi(D_i)$

$1 + 0 - 1$

pop = 0.

multipop → actual cost + potential diff

$K + (-K)$

multipop = 0.

↳ so, the role of potential function is to decrease the potential of costliest operation considerably. so it should cancel something.

$\bar{C}_i \leq 2$ for above example,

$\Rightarrow \sum C_i \leq 2n$ for 'n' operations [1 to n]

$$\sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \bar{C}_i \leq 2n$$

$\Rightarrow \sum C_i \leq 2n$.

eg:- Binary increment variable.

```
00 ..... 0001
00 ..... 0010
00 ..... 0011
00 ..... 0100
    :
01 ..... 1111
11 ..... 1111
```

after n such operations

Sol:

```
Binary_Counter_increment(A)
{
    i = 0
    while (A_i == 1)
        A[i++] = 0
    A_i = 1
}
```

n-turies

O(n log n)

BCI operation

problem:- $\log_2 n$ happens only once in the program.

most operations $(n/2)$ happens at $i=0$, so only one flip.

bit #	# of operations
0 →	n
1 →	$n/2$
2 →	$n/2^2$
:	
i →	$n/2^i$

so, # of operations $\leq n \leq \sum \frac{1}{2^i} \leq 2n$.

Alternate :-

every step $0 \to 1$ flip (only 1)

several $1 \to 0$ $(0 \le K \le \log n)$

$\bar{c}_i$ → $0 \to 1$ → 2 (# of 1's)

$\bar{c}_i$ → $1 \to 0$ → 0 (# of 1's)

so, ϕ = # of 1's.

$\phi(D_0) = 0$ for all cases (& '0' one's)

$0 \to 1$ $\quad 1+1 = 2$

$1 \to 0$ $\quad 1-1 = 0$

K such operations.

$\bar{c}_i = 2$

$$\sum c_i \le \sum_{i=1}^{n} \bar{c}_i = 2n \Rightarrow \sum c_i \le 2n$$

eg: **Dynamic table :-**

10^5 element (buckets) [size of bucket at anytime $\le 10^5$)

10^4 such buckets,

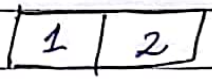
total m/m required = $10^4 \times 10^5 = 10^9$ XX

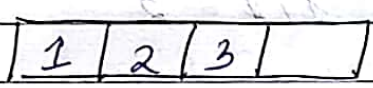
↳ initially, each bucket is empty.

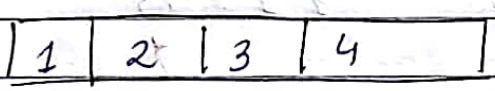
don't use linked list
(sequential access)

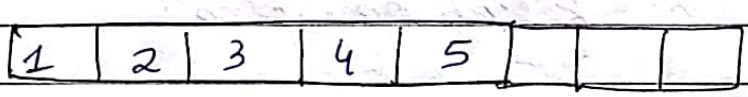
↳ throw an element in a bucket
↳ create a bucket of size 1

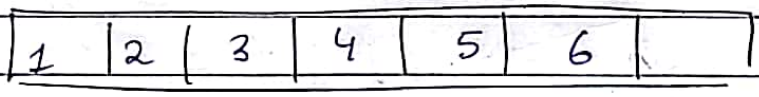
↳ insert another element, overflow!
↳ create new bucket of 2* size of previous
↳ copy its contents
↳ add element
↳ free space occupied by previous

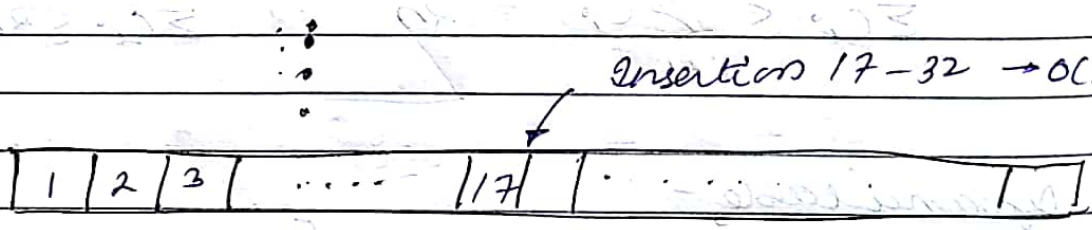












↳ If I know, actual size,
n - insertions = $O(1) \times n = O(n)$

↳ but in above example if it happens to be $O(n^2)$ not feasible
but
it is $O(kn)$ $k > 1$

so, $c_i = \begin{cases} 1 & i \neq 2^k + 1 \\ i & \text{if } i = 2^k + 1 \text{ for some } k \end{cases}$

$$\sum c_i = 1 \times n + \sum_{k=0}^{\log n} 2^k$$

$$= n + 2^{\log_2 n + 1} - 1$$

$$n + 2 \cdot 2^{\log_2 n} - 1$$

$$n + 2n - 1$$

$$3n$$

$\Rightarrow \sum c_i \leq 3n$.

↳ alternate:-

c_i → 1 if $i \neq 2^k + 1$
→ $1 + 2^k$ if $i = 1 + 2^k$

$\bar{c}_i \Rightarrow \phi$ = empty space ✗ $(\phi > 0)$

ϕ = no. of elements − empty slots

$\bar{c}_i$ → no copy → $1 + [1 - (-1)] = 3$ (increase by 1 / decrease by 1; after operation / before operation)

→ copy → $(k+1) + [[(k+1) - (k-1)] - (k - 0)] = 3$

$\sum C_i = \sum T_i = 2n$

OR.

ϕ = 2 × no. of elements − size of table.

→ no copy $k+2-1$

copy $(k+1) + [(2^{k+2}-2^k) - (2^k - k)]$

$= k+1+2-k = 3$.

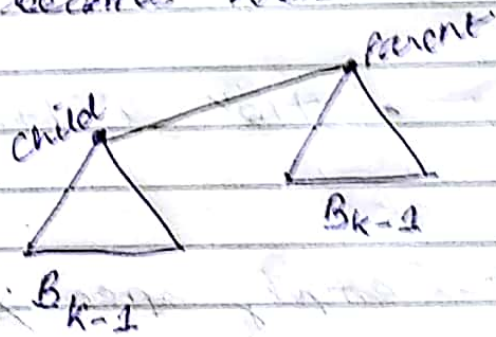
Thus, potential function may not be unique.

- Binomial Heaps :-

o (parent) ≤ (child)

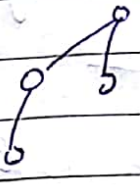
$B_0 = 0$ (empty tree)

(1) B_k = merger of two B_{k-1}, B_{k-1} trees. the tree with min value at root becomes root.

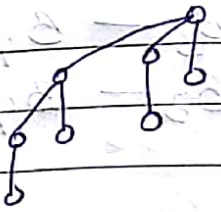


So, B_1 =

B_2 =



B_3 =



$N(B_{i-1}) = 2^{i-1}$

$N(B_i) = 2 \cdot 2^{i-1} = 2^i$

↓

No. of nodes in B_i tree

② Height (B_i) = $Ht(B_{i-1}) + 1$

$= i - 1 + 1$

$= i$

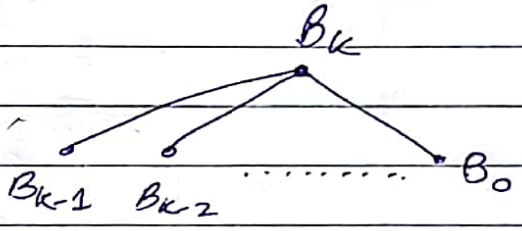
# of nodes in $B_i = 2^i$

$\Rightarrow ht(B_i) = \log n$.

③ $N(k, i) = {}^kC_i = \binom{k}{i}$

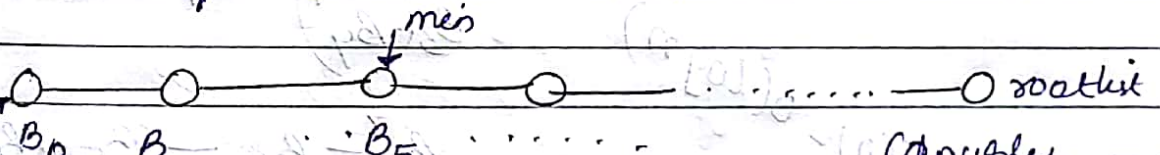
order k heap; element i

④



order/degree (B_k) = k.

↳ Binomial Heap = collection of Binomial trees.



↳ for each collection, we can have one copy only, i.e. there cant be two B_i's in the collection. If exist, merge them.

↳ each tree follows heap property.

↳ Union :- (same as adding two binary #s.

$B_5 B_4 B_3 B_2 B_1 B_0$

eg: 1 1 1 0 1 1
0 1 1 1 0 1
1 0 1 0 0 0
B_6 B_4 B_3

$L_1 = B_0 - B_1 - B_3 - B_4 - B_5$
$L_2 = B_0 - B_2 - B_3 - B_4$

$B_1 - B_1 - B_3 - B_4 - B_5$
$B_2 - B_3 - B_4$

$B_2 - B_3 - B_4 - B_5$
$B_2 - B_3 - B_4$

$B_3 - B_3 - B_4 - B_5$
$B_3 - B_4$

$B_4 - B_4 - B_5$
$B_3 - B_4$

$B_3 - B_4 - B_5 - B_5$

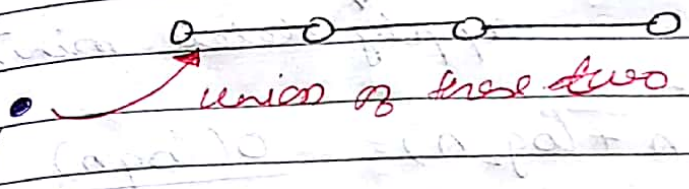
$T(n) = O(\log n)$

$L_1 \cup L_2 = B_3 - B_4 - B_5$

so, $n = \sum 2^i b^i$, $b \in \{0, 1\}$

→ depends on # of 1's in binary representation $= O(\log_2 n)$ (atmost)

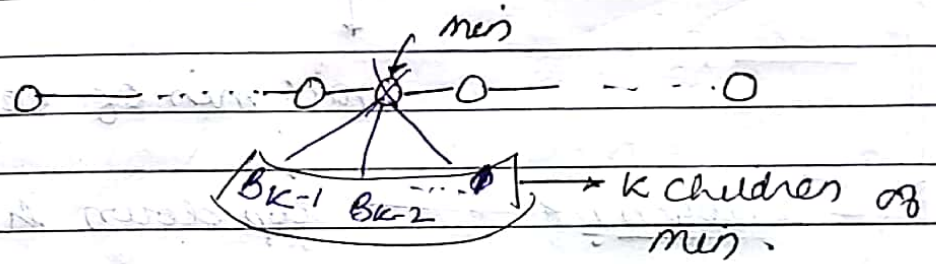
↳ add :- create one more node, and add/merge to the list.



so, add = $O(\log_2 n)$

↳ min :- as we have pointer to minimum node, it can be done is $O(1)$.

↳ delete min :- go to minimum node (B_k)
It has k children



merge them with root list.

delete_min = $O(\log_2 n)$.

↳ change-key. → decrease key
→ increase key.

go to node → decrease value,
↳ bottom up heapify.
↳ update min if required

same for increase (top down heapify)

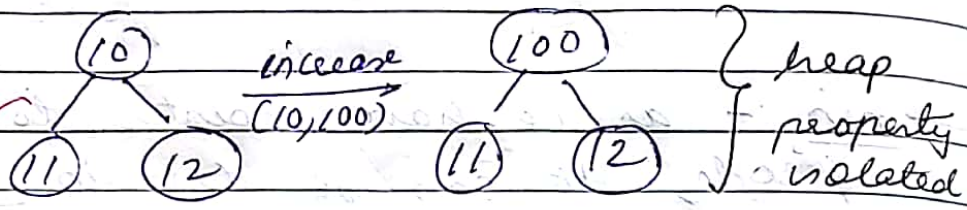
so, change key(n) → in B_k is $(\log n)$.
since ht(B_k) = $k < \log n$.

↳ delete general key :-

decrease it less than minimum
↓
apply delete minimum.

$\Theta(\log_2 n + \log_2 n) =$ $O(\log n)$.

↳ increase key



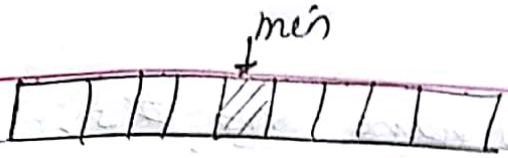
ht = log(n)
↓
pick min of each child
↓
$O(\log^2 n)$ ← top down heapify.

delete & insert
↓ ↓
$O(\log_2 n)$ $O(\log_2 n)$
↓
$\Theta(\log_2 n)$.

delete min } $O(\log n)$
k
else $O(1)$.

↳ If I want to perform add in $O(1)$ instead of $O(\log n)$

↳



Priority Queue

Binary Heap

(i) Add(X) = O(1) (add at begining) } Binary heap as linked list.

(ii) min = min (old min, X)

↳ to delete min in B. Heap → O(logn).
or linked list
↳ update min is a problem!
so, I build a Binomial heap & merge with original heap.

so, Build heap, gives 'n'
for (i = 1 to n) } O(nlogn)
add(i)

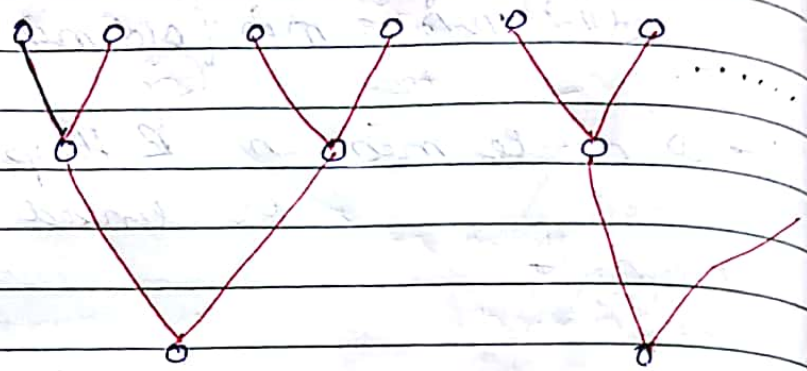
but there is an O(n) algo also!

Binomial Heaps.
↳ O(n)

add_element and merge.] this has one to one mapping with binary tree.

so, If I have n- #s,
↳ how to build a binomial heap efficiently?
↳ sol?

merges
$n/2$ B_0's
$n/4$ B_1's
$n/8$ B_2's



O(n).

↳ deletemin → O(n).

root list

linked list
↳ create tree from scratch.

↳ # of elements in linked list

(1) add() → 1 + 1 = 2.
(add to list, easy).

(2) min() → O(1)

(3) deletemin() → build binomial heap on linked list (n elements).

merges ← + (log n)

then, deletemin → $n_2 + \log n - n_p$
$= \log n$.

(4) decrease key. → $O(\log_2 n)$

(5) build heap → $O(n)$

(6) add. = $O(1)$ → everything is same as binomial heap.

(7) meggo 2 heaps & 2 linked lists. $O(\log n)$.

LECTURE - 20

Operation	Linked list	Binary Heap	Binomial Heap	Fibonacci Heap
insert	1	log n	log n	1
delete_min	n	log n	log n	log n
decreasekey	n	log n	log n	1
delete	n	log n	log n	log n
union	1	n	log n	1
findmin	n	1	1	1

too costly operation so we migrated to Binomial heap.

keep pointer to a minimum O(1)

→ Fibonacci Heap :-

↳ all operation costs are amortized costs, not actual costs.

↳ one delete min can take $\Theta(n)$.

n_1 = # of operations (delete / delete min)

n_2 = # of insert, decrease key, union, findmin

$$T(n) = O(n_2 + n_1 \log n)$$

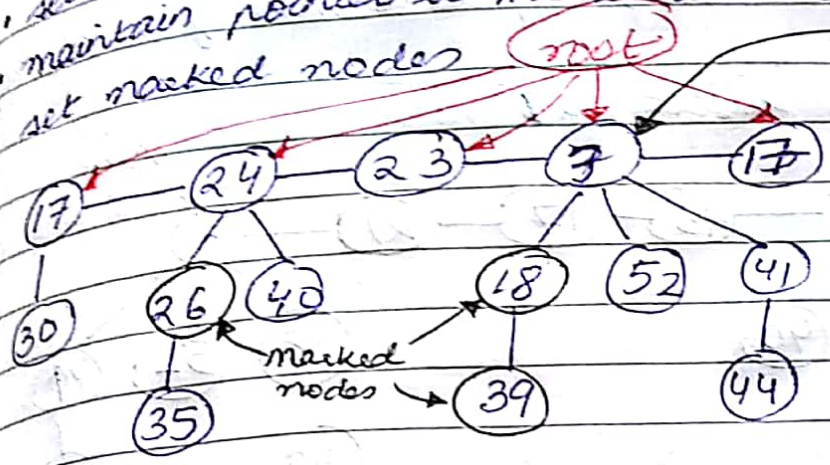
↳ idea :-

similar to binary heap, but less rigid structure.

- binomial heap :- eagerly consolidates after each insert (greedy)
- fibonacciheap :- lazily defer consolidation until next delete min.

Properties of fibonacci heaps:-

- set of ordered trees
- maintain pointer to minimum element
- set marked nodes



↳ root list is a doubly linked list.
↳ root has minimum element in each tree.

n = # of elements in the fibonacci heap = 14.
rank(x) = # of children of x
rank(H) = max rank of any node = 3.

max # of children

trees(H) = # of trees in heap = 5.
marks(H) = # of marked nodes in heap = 3.

↳ potential function:-

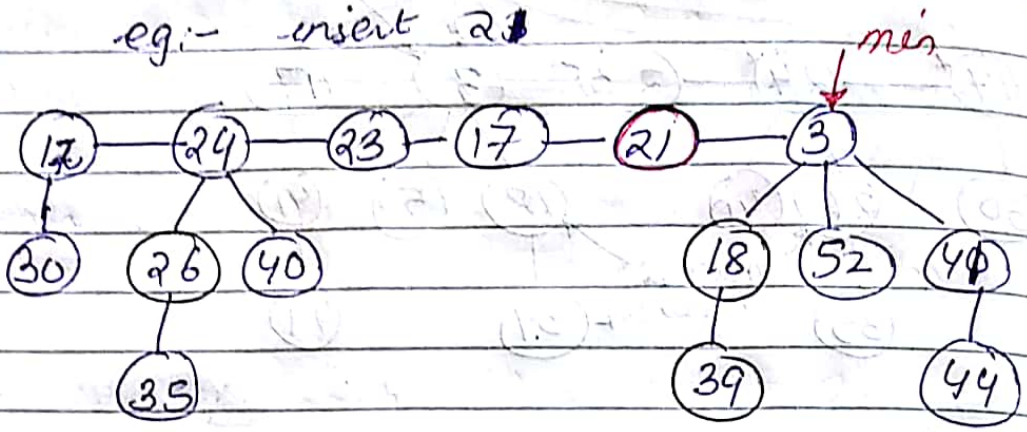
$$\phi = trees(H) + 2 \cdot marks(H)$$

so, for above tree,

trees(H) = 5
marks(H) = 3
$\phi(H) = 5 + 2 \times 3 = 11$.

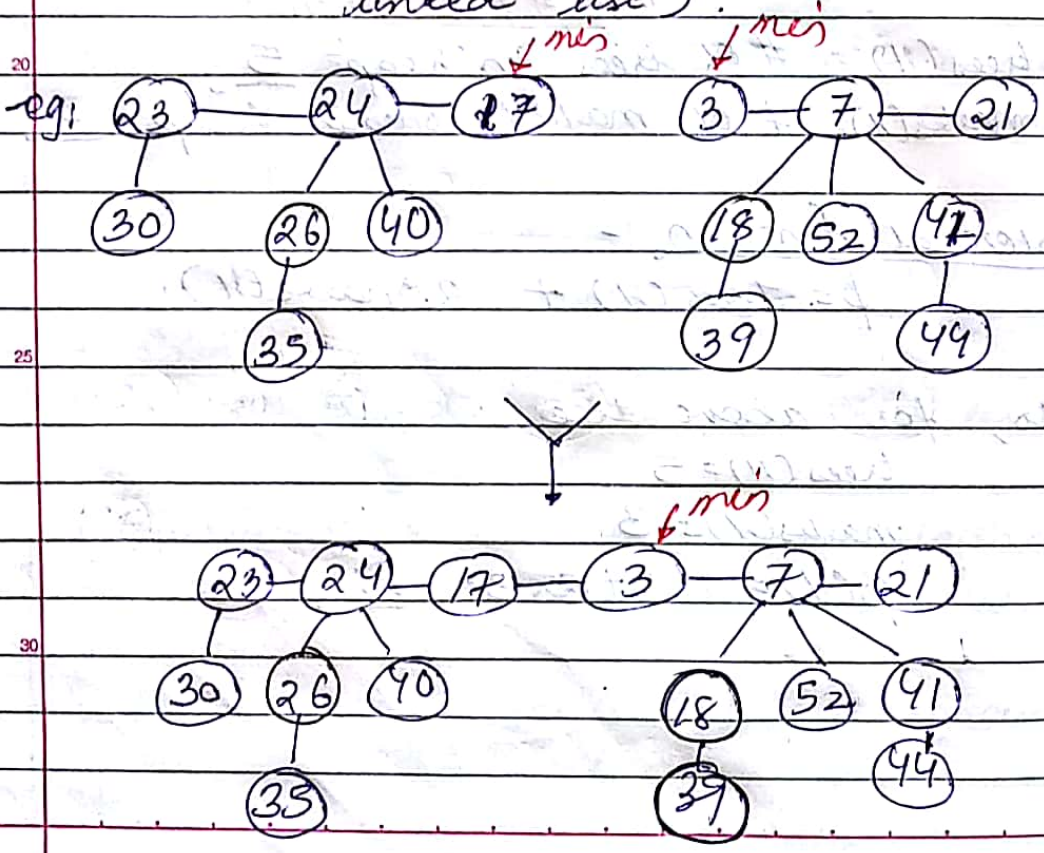
↳ insert.
- create a new singleton tree
- add to root list, update min if necessary

eg:- insert 21



change in potential = 1.
amortized cost = 1 + change in potential
= 1 + 1 = 2.
= O(1).

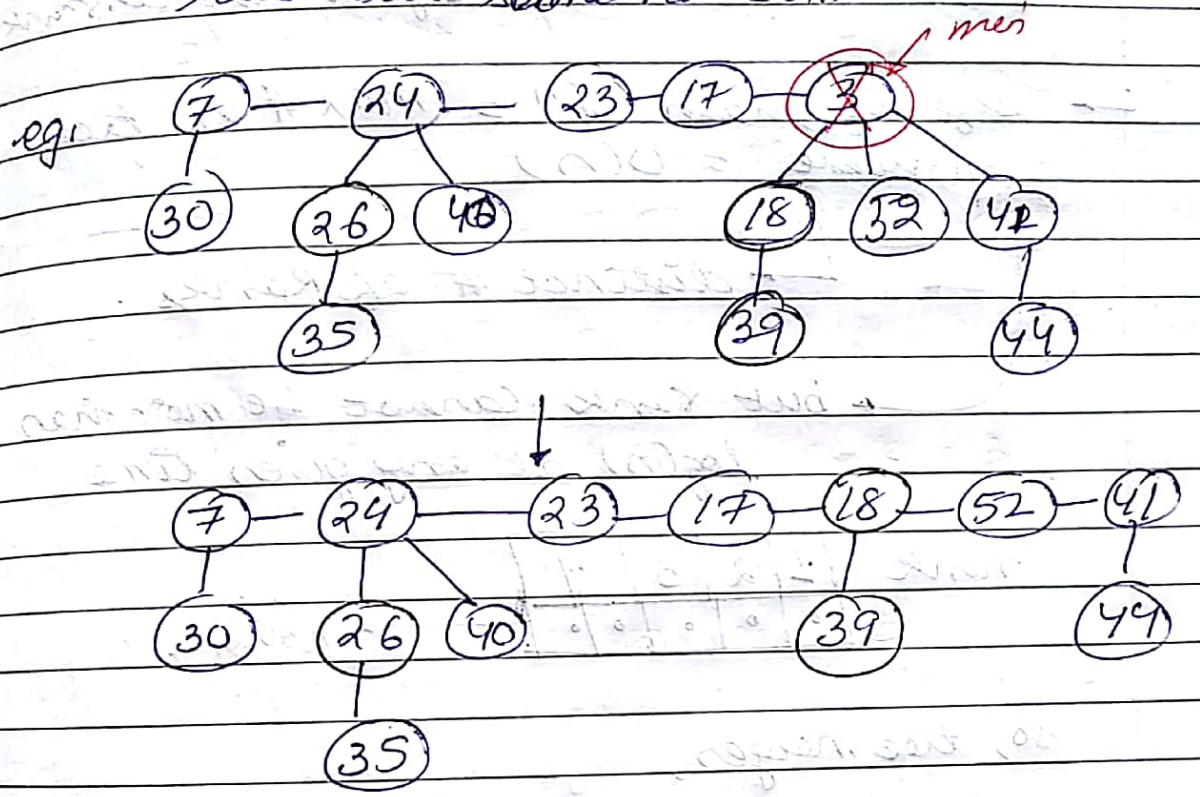
↳ Union :- (root list is circularly doubly linked list).



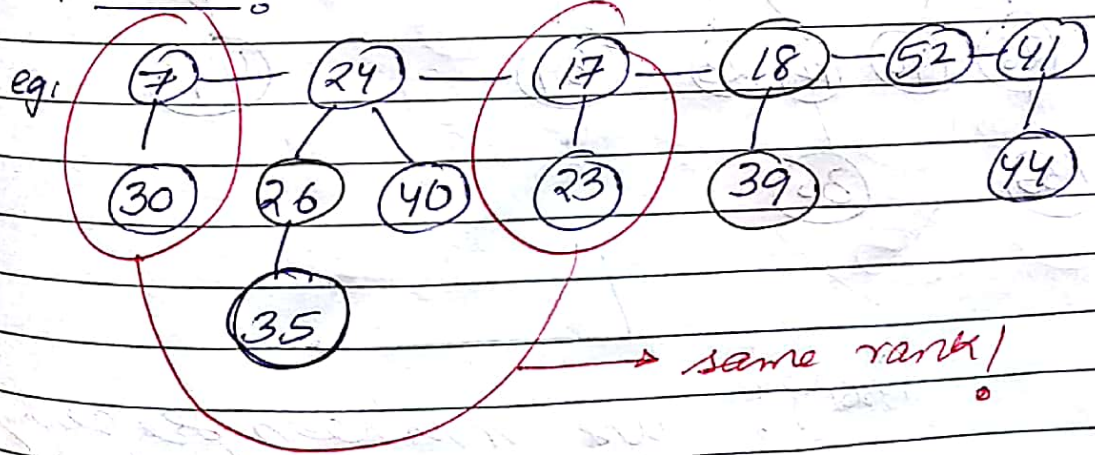
Amortized cost = 2 + 0 = O(1).

↳ delete min :- search new min O(n)
(decrease potential)

- delete min
- meld its children into root list, update min
- consolidate trees so that no two trees with same rank exist.



problem :- another tree with same rank.



I need a data structure to answer this

↓

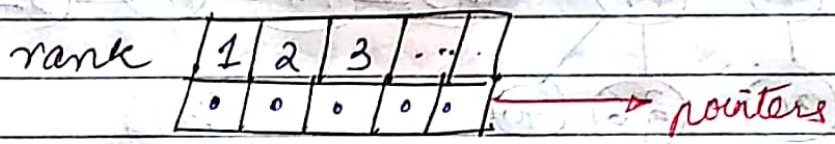
simple hash function.

either '0' or '1' pointer
- k^{th} index → none → not seen a tree yet whose rank is k
- If seen such a tree, T[i] gives its with rank = k

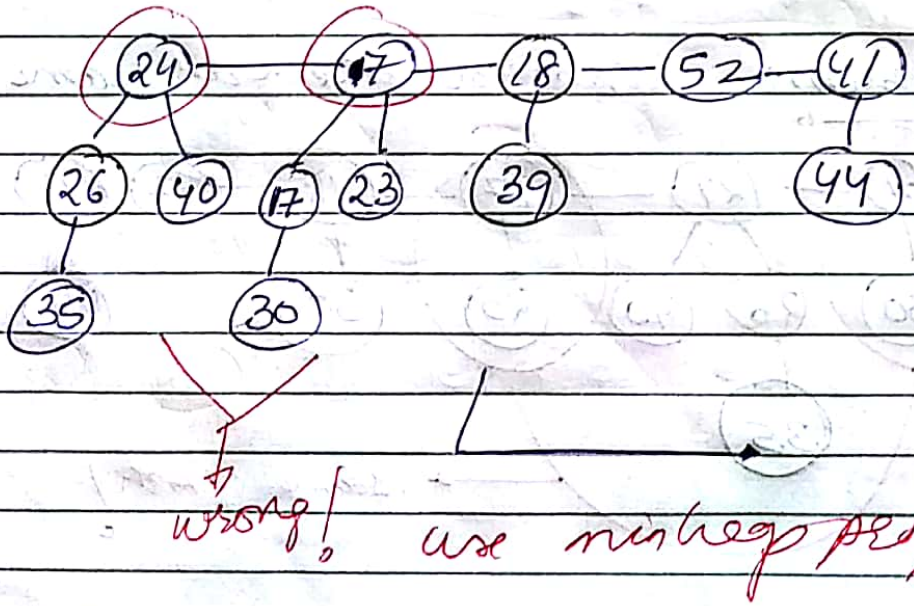
↳ size of hash fn = max # of trees possible = O(n)

↳ distinct # of Ranks.

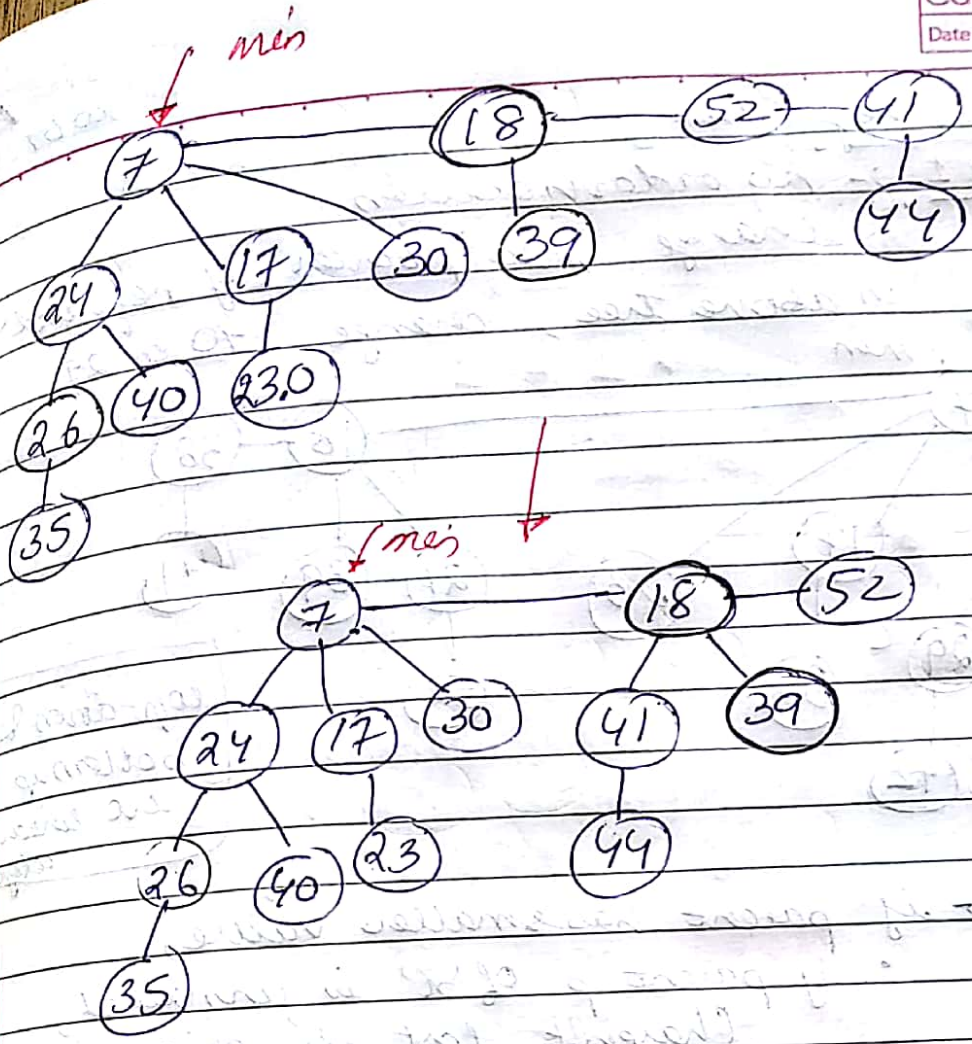
↳ but rank cannot be more than log(n) at any given time.



so, tree merges,



wrong! use mixtape properly!



stop!

↳ decrease in potential is not a problem for us, increase in potential is!

$$\phi(H) = trees(H) + 2\,marks(H)$$

so, in case of deletion

$$Rank(x) + (Rank(x) - 1) \underbrace{(-2)}_{\text{if min = marked node, else zero.}}$$

actual cost

(cutting the children)

- so, Cutting the children.

$$\leq 2Rank(x) \leq 2Rank(H) \leq O(\log n)$$

- Consolidation → merge $1 + (-1) = 0$ (# of trees decrease by 1.)

→ no merge $Rank(H) + \underbrace{1}_{\text{'0' also included}}$ $0 - k = k+1$ operations

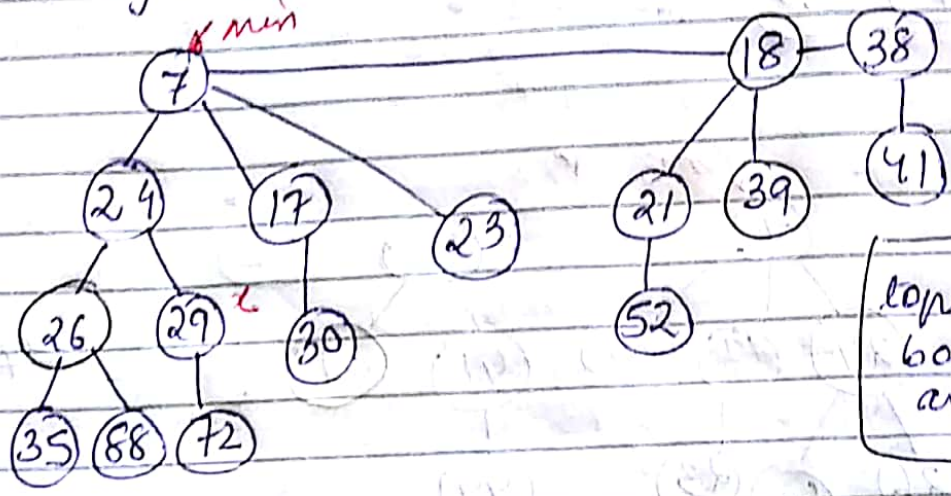
$= O(\log n)$

↳ **decrease key** :-

Case 1 :- no order violation.

- change min pointer if required.

eg: in above tree, change 40 to 27

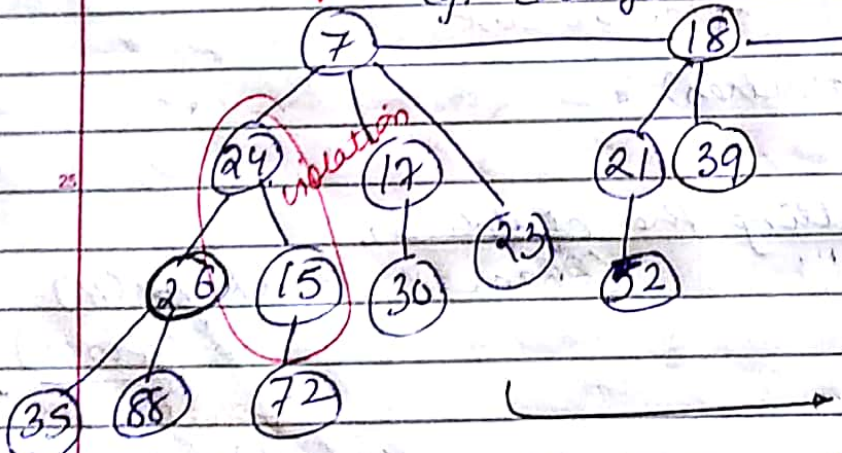


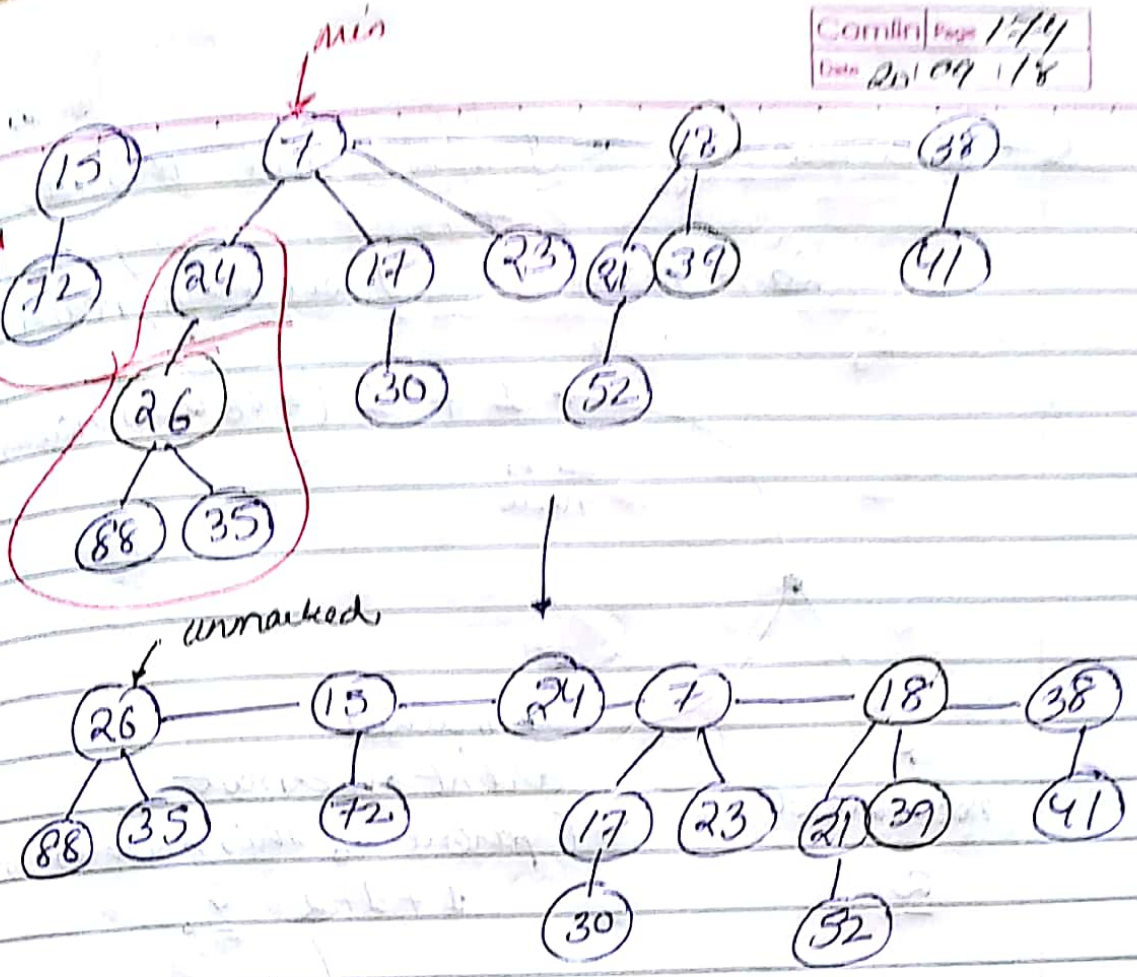
top-down & bottom up are linear algos.

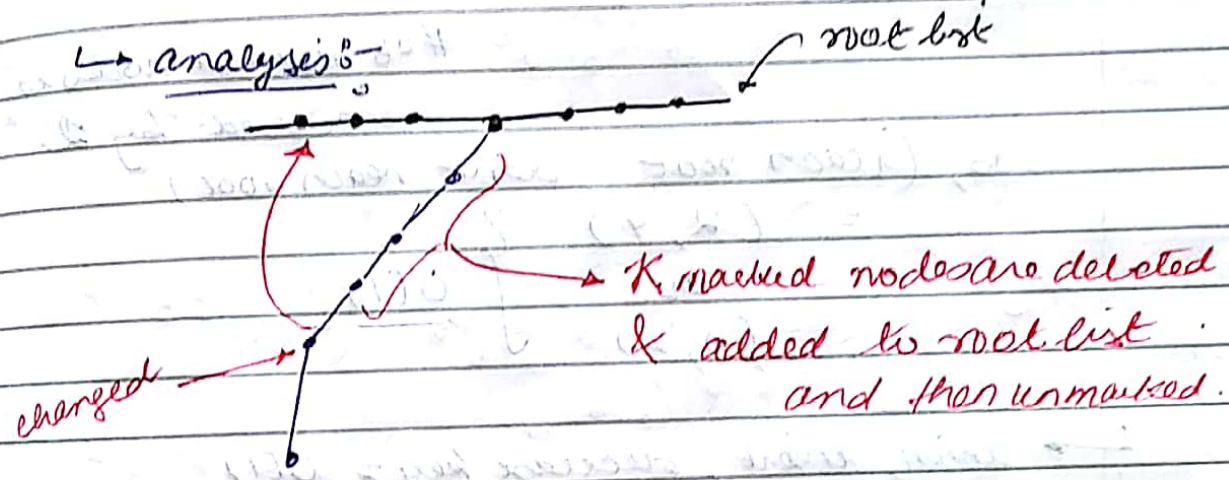
↳ if parent has smaller value,

- if parent p of 'x' is unmarked, (hasn't lost its child yet), mark it.
- otherwise, cut (p, meld into root list, unmark.
- do this recursively for all ancestors that loose a second child.

eg: change 29 → 15.







amortized complexity of k operations = O.

ϕ = Trees(H) + 2 * marked nodes in H.

ϕ = 1 + 1 + 2(-1)

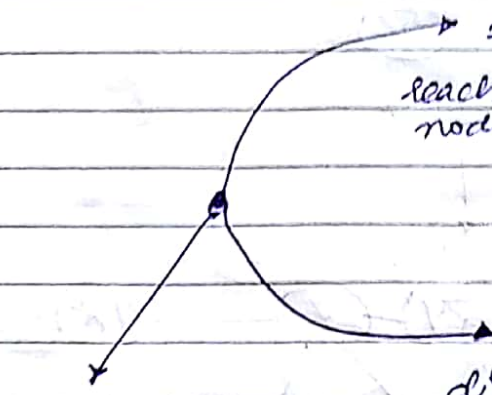
1 → actual cost to cut & add to root list

1 → trees increases by 1.

2(-1) → marked nodes decreases by 1.

$\equiv 0$

↳ decrease key :- I might add this to root list
+
last node that I see here, reach root.



→ 1 + 1 (#of trees increases) = 2.

→ marked
didn't reach root
(parent of this node is marked)
1 + 1 + 2 = 4.

not marked
2.

↓

#of marked nodes is increased by 2.

so, (reach root, dont reach root)
= (2, 4)
or
(0, 2)
} O(1).

↳ union, insert, decrease key = O(1).
• increase key → delete key
→ add key } O(log n).

• delete min → O(log n)
↓
If I prove rank(H) = log(n).

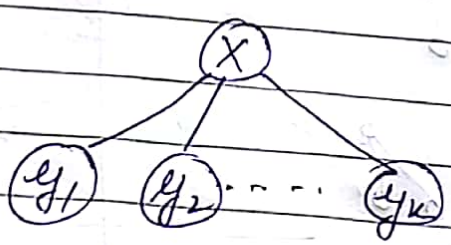
↳ **Lemma** :- If a node is tree.

Let x = node and

$y_1, y_2, \ldots, y_{k-1}, y_k$ its' children,

in order, they were linked to x

then,



Proof :-

↳ when y_i was linked to x, 'x' had atleast i-1 children :,

$y_1, y_2, \ldots, y_{i-1}$.

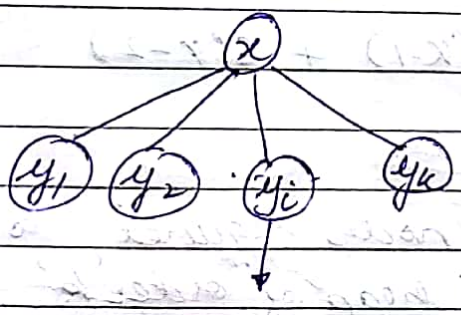
↳ since only trees ~~and~~ of equal rank are linked at a time,

$rank(y_i) = rank(x_i) \geq i-1$

↳ since then y_i has lost atmost one child

Or, y_i would have been cut!

↳ Thus, right now, $rank(y_i) \geq i-2$.

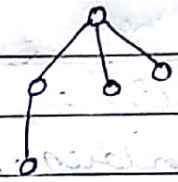
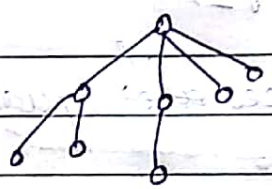
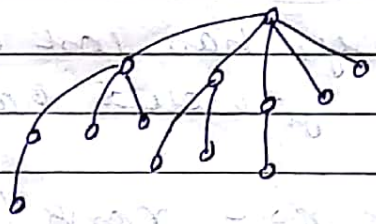


y_i became child when $rank(x) = rank(y_i) \geq i-1$

we might have cut some children of x

Page 173
Date 20/09/17

definition :- Let S_k be the smallest possible tree of rank 'k' that satisfy the property.

Tree	structure	# of nodes
F_0		1
F_1		2
F_2		3
F_3		5
F_4		8
F_5		13

thus, $S_k = S(k-1) + S(k-2)$

↓

min # of nodes required to make fibonacci heap of order k'.

fibonacci heap, with n-nodes in it has rank 'r'.

$S(r)$ = min # of nodes required if rank = r

$$n \geq S(r) = S(r-1) + S(r-2)$$

$$n \geq S(r-1) + S(r-2)$$

$$n \geq 2S(r-2)$$

$$n \geq 2^{r/2}$$

$$r \leq 2 \log n$$

$$\Rightarrow r = O(\log n)$$

$\therefore$ rank(H) $= O(\log n)$ (max rank in heap)

↳ Height(x) = O(n)

eg: (n) — (n-1) — (n-2) (min) (delete min)

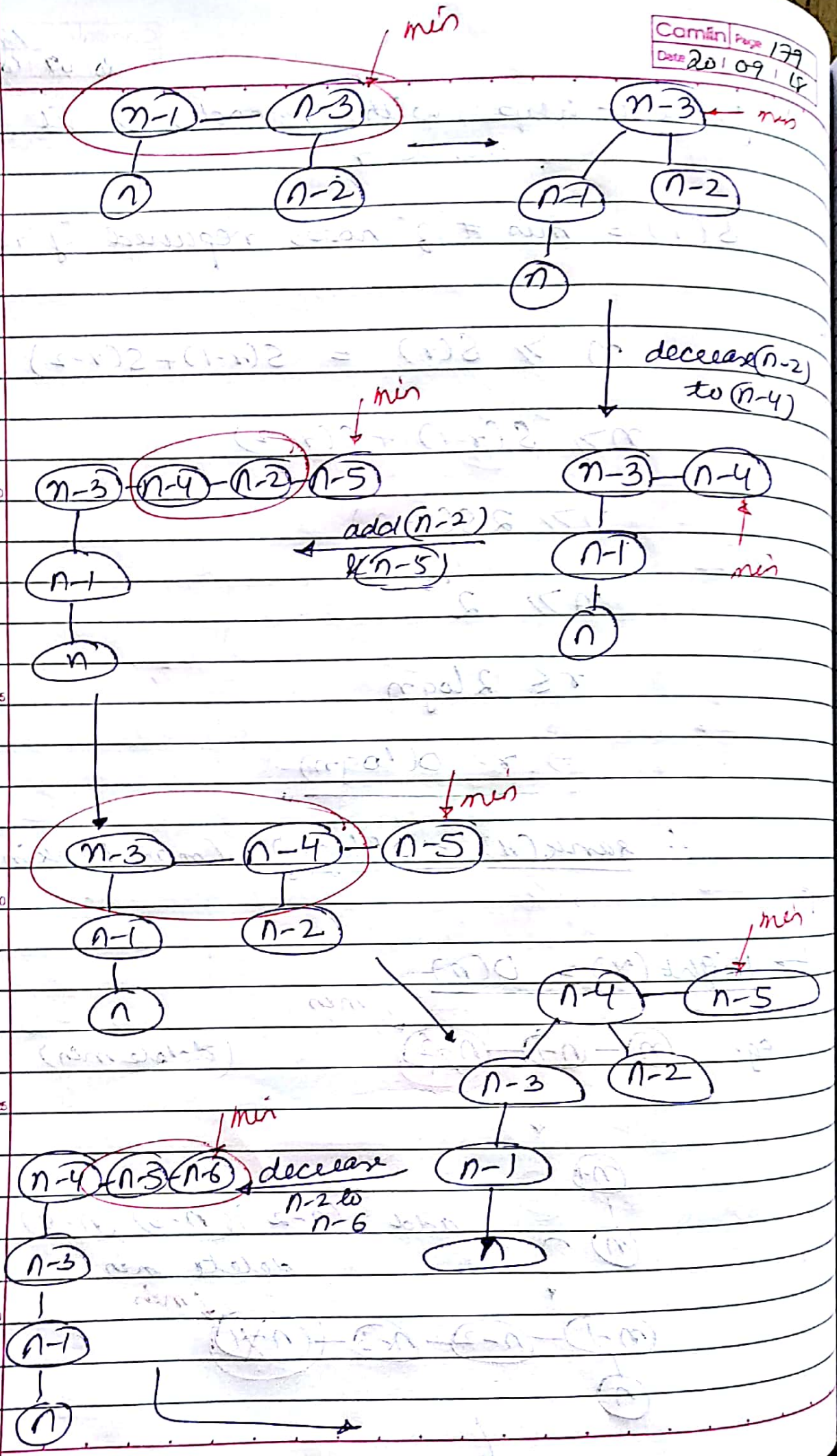
(n-1)

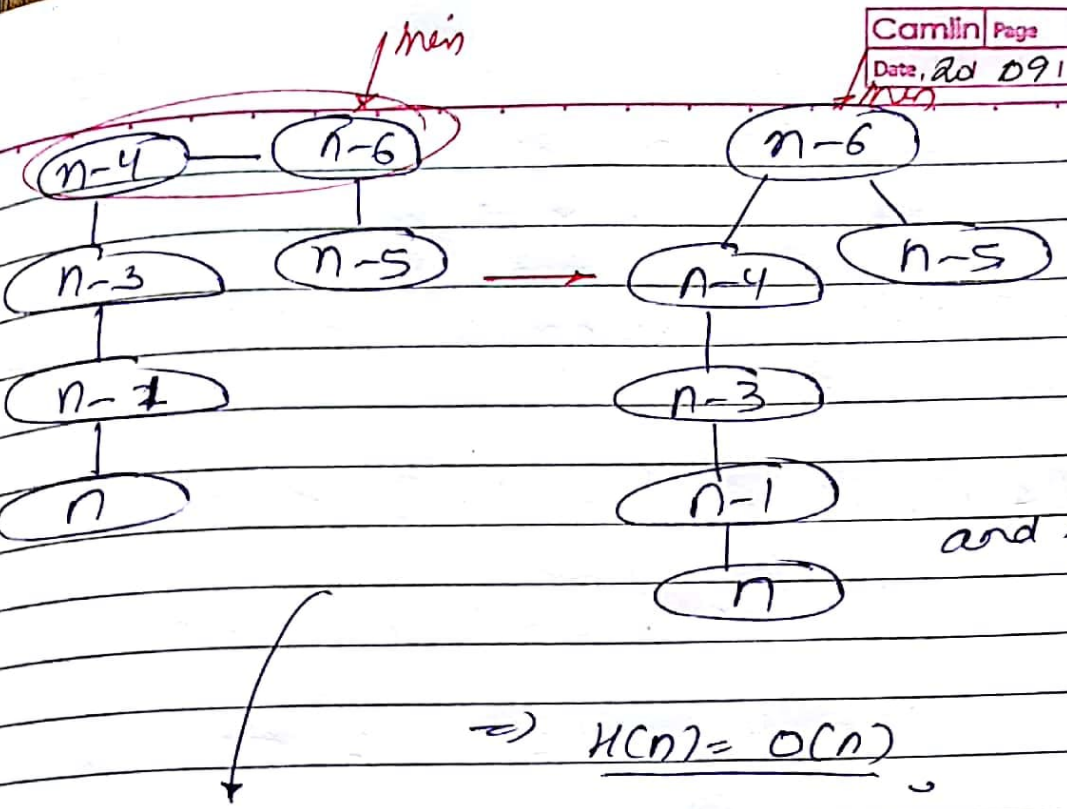
(n) add (n-2), (n-3), (n-4)

delete min

(n-1) — (n-2) — (n-3) — (n-4) (min)

(1)



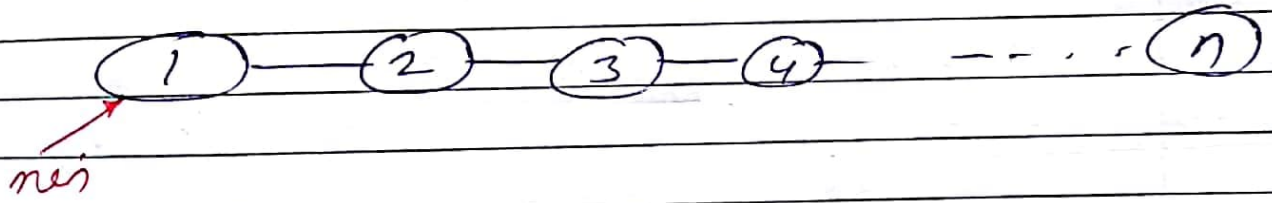


$\Rightarrow H(n) = O(n)$.

i.e. sequence of 4 operations increase the height by 1,

for horizontal,

add 1, 2, 3, 4, ... n



↳ increase key :- I can do deleting & adding?

↳ worry about childrens!

$\log(n)$ of childrens are possible

↓

cut them all → $O(\log n)$

heapify bottom up → $O(\log^2 n)$

better than deleting & adding again!