Merge Sort on a Linked List → ① Find mid ② Sort left ③ Sort right ④ Merge them

① LL → middle (mid).

② a) left half ← M.sort

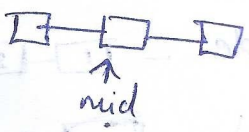
b) Right half ← M.sort

// breaking the list

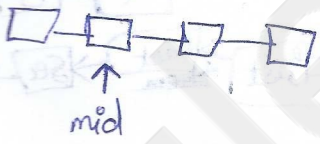
mid.next = null

③ Merge

Odd



Even



---

① Find the middle by slow fast approach

↳ add a corner case if(fast==null){ //even case

mid = prev }

↳ Prev will one node before slow.

② Right Head = mid.next

mid.next = null;

m.sort(head) left half

m.sort(rightHead) Right half] Sorting.

③ Merging

ⓐ Make temp Linked list

temp → head

temp.data = -1] dummy node

if (Left.data < Right.data){

temp.add(left.data); }

else temp.add(RightHead.data) }

if elements are remaing add all them.

↳ in final list remove first the dummy node

Zig Zag Linked List

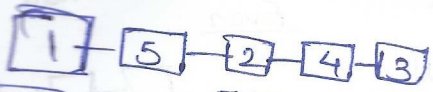
→ We have given a Linked List

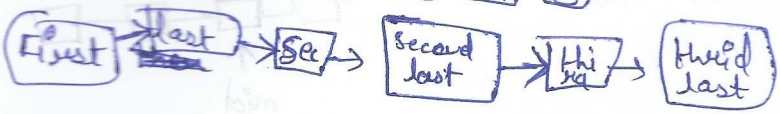
$L(1) \rightarrow L(2) \rightarrow L(3) \rightarrow L(4) \rightarrow L(n-1) \rightarrow L(n)$

We need to convert it in zig zag form

example –

→





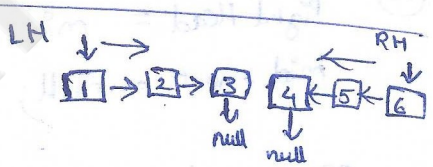
Approach

① find mid Node (mid = 1st half last node)
(slow & fast approch)

② Reverse second half

③ divide the both L.L (Because its hard to travel backwards in L.L)

④ Alternate merging



↳ Node L.H = 1st half head (Left head)

Node R.H = 2nd half head (next(left))

Node NextL, NextR. (next Right)

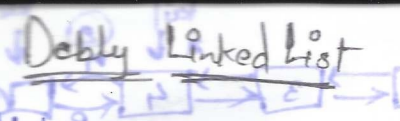
while (LH != null & RH != null) {

zigzag {
- nextL = LH.next
- LH.next = RH
- nextR = R.Hnext
- RH.next = nextL

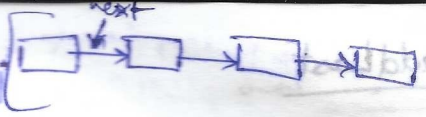
update {
- RH = nextR
- LH = nextL

}

Dobly Linked List



Singly L.L



- It have both next and prev.
- Therefore travelling backward in this is more easy.

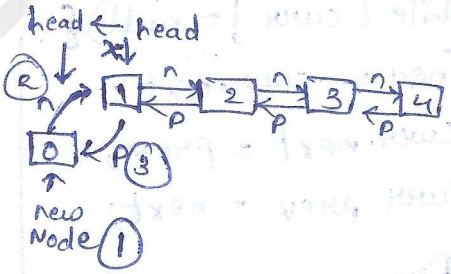
Dobly L.L

It have both next and previous.

Code →

```
Class DoubleLL {
    Class Node {
        int data;
        Node next;
        Node prev;
        Construction { public Node (int data) {
            this.data = data;
            this.next = null;
            this.prev = null; }
        public static Node head;
        public static Node tail;
        public static int size;
```

add First



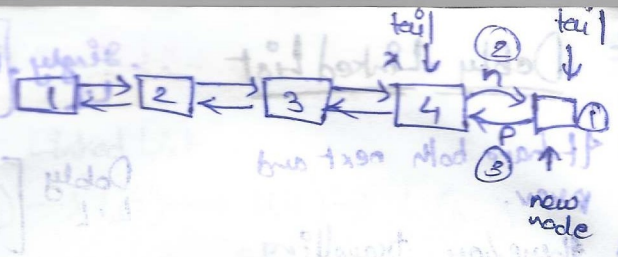
Step 1) Create node
Step 2) newNode.next = head;
Step 3) head.prev = newNode;
Step 4) head = newNode;

Corner Cases → `if (head = null) { head = tail = newNode; return }`

Remove First

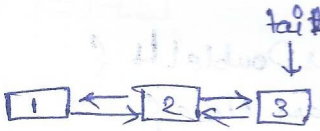
step 1) head = head.next;
step 2) head.prev = null;
step 3) size--;

addLast



Step ① Create newNode.
Step ② tail.next = newNode
Step ③ newNode.prev = tail
Step ④ tail = newNode.

RemoveLast




```
Node temp = tail;
```
Step ① `tail = tail.prev;`
Step ② `tail.next = null;`
`return temp.data;`

Reverse a Dobly LL.

```
Node curr = head;
Node prev = null;
Node next;

while (curr != null){
    next = curr.next;
    curr.next = prev;
    curr.prev = next;
update { Prev = curr
         curr = next
}
head = prev;
```

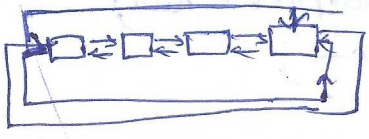
→ here just link the head with tail and tail with head

~~Example~~ → `head.prev = tail;`
`tail.next = head;`

Circular Linked List

→ Normal doubly L.L

Circular ^ doubly Linked List

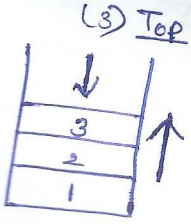


- Doubly list is asked rarely.

Stacks → Explicit (made by us)
implicit (made by computers automatically)

- Operations
  1. Push O(1)
  2. Pop O(1)
  3. Peek O(1)

  } Every operation for only top element



(LIFO)
Last In First Out

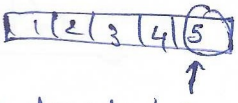
- Implementation -

Arrays
↓
fixed size

- stack == full

As array is of fixed size there we does not prefer this for stack.

ArrayList
variable size



last element = top

LinkedList
variable size

Head = Top

here head is top because head is the only element on which we can perform the tasks with O(1) time complexity.

Stack → using Array List.

Push → To add element

```
Public static void push(int data)
{ list.add(data); }
```

Pop → To remove

```
Public static int pop(){
// if list is empty
if(isEmpty()){
    return -1 }
int top = list.get(list.size()-1);
list.remove(list.get()-1);
return top;
}
```

Peek

```
Public static int peek(){
// if the list is empty
if(isEmpty()){
    return -1;
}
return list.get(list.size()-1);
}
```

isEmpty

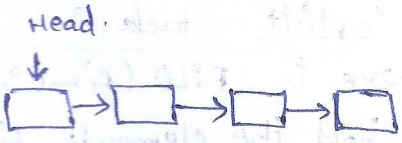
```
→ public static boolean isEmpty(){
    return list.size()==0;
}
```

Stack → using Linked list

↳ Here top is considered as head.

1) isEmpty → to check weather the L.L is empty or not.

```
public static boolean isEmpty(){
    return head==null;
}
```



2) push

↳ As the top is head always
So we would use add-
-First function.

3) pop

```
if(isEmpty()){
    return -1;}
int top=head.data
head=head.next;
return top;
```

4) peek

```
if(isEmpty()){
    return -1;}
return head.data;
```

Stack using JCF

Just write

```
import java.util.*;
Stack<Integer> s= new Stack<>();
```

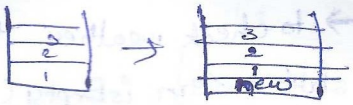
Submit Functions

1. push();
2. s.pop();
3. s.peek();
4. s.isEmpty();

amazon.

Push at the bottom of the stack

→ we will solve with recursion



→ In recursion there have implicit stack in which there is FILO (First in last out) it will hold the elements till we past the desired element at the base and then we will add the elements one by one. by our stack.

Reverse a String using stack

↳ Most easy thing to do in stack is this

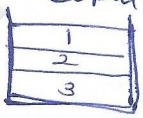
1. Just add the elements.
2. And then print the elements
3. Due to LIFO stack will be automatically reversed.

Approach
1. push all elements
2. Peek - pop.

Reverse a Stack → We are told to make a stack that is reversed that of original.

→ example Input

Ouput

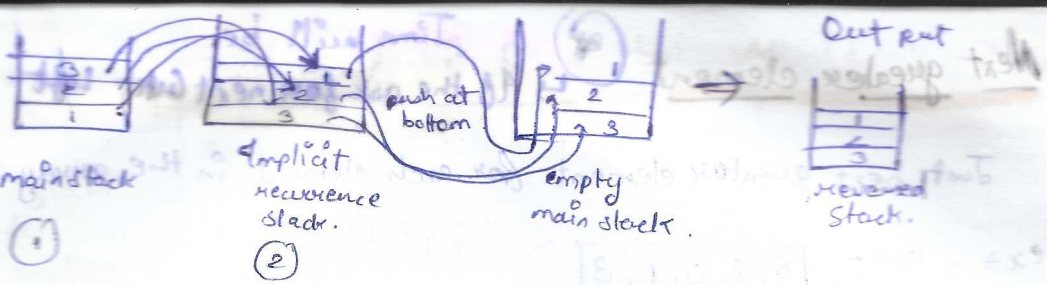


→ By bruteforce approach then will need extra memory So to same memory and time we will use recursion

Approach

Step 1 add all the elements in recursion stack
base case → until our stack become empty.

Step 2 back track all the element in the main stack but use push at bottom So the stack will be empty.

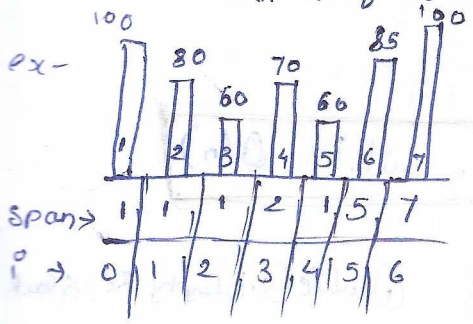


Q # Stock span problem

↓

max no. of consecative days for which price ≤ todays price including the day itself

i = index



∴ For this we derive a formula

Span = i − prev High (i)

ex- for day 4

⇒ 3 − 1 = 2

but this take so much time

$O(n^2)$

Time O(n)

• Now optimised soln using Stack ← our stack will store the indexes of the stock price

↳ we will make an stack which store the value of a stock

↳ if the current stock price is ~~great~~ smaller than previous stock price store the value of stock in the Stack.

↳ And in array store → [index of current] − [index of prev high]

but if the curr stock price is greater then prev stock price remove the prev from stack and check again until find a stock price that is greater.

↳ If found the the greater price stock

and if not and the stack became empty then store in array → [index of curr + 1]

Next greater element (#)

Time will be
↳ If the ask for next Gret left then

Just next greater element for each element in the array. ↳

ex → arr = [6, 8, 0, 1, 3]

next Greater = [8, -1, 1, 3, -1]

If found no greater element then -1 or if no element after that then also -1.

Brute Force

↳ Nested Loop

Time will be O(n)

- For Optimised with Stack →

[]arr = [6,8,0,1,3] (0 1 2 3 4)

next Greater [] = [8, -1, 1, 3, -1]

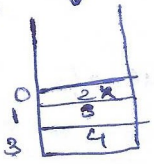
⇒ here we check from backwards.

① while (S.isEmpty && stack
S.pop() }

② if stack.empty
nextGreater = -1
else
next Greater = S.peak()

③ S.push (element)

Element ↓ Index ↓

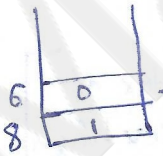


Stack ↓

→ for 3
stack is empty ∴ -1
and also push the index of 3 in stak

for 1
3 is greater
∴ 3 add in this → arr,
also push the index of 1

Ans → nextGreater: 8 | -1 | 1 | 3 | -1
but add backwards



for 8

→ for 0
1 is greater and next
∴ 1
push i of 0

→ For 8.
0 is smaller ∴ pop
1 is small ∴ pop
3 is small ∴ pop
Now empty ∴ -1
push index of 8

For 6, 8 is next greater and further element will be greater because 8 has poped them all out who where smaller than him

⇒ ∴ 8 add in next Greater
push index of 6

Done return next greater Array.

Valid Parentheses ☆

ex- String will be given

s = "()[]{}" ✓ s = "[]" ✗

s = "()" ✓ s = "[({})]" ✓

① opening will be first and each opening will have its own closing.

ex- ((({[]}())))

=> ① Opening bracket
s.push();

② Closing bracket
↳ check does its pair present at top
if not return False;
↳ if yes s.pop() and continue;

③ At last check if the string is empty or not
if empty → return true;
if not → return false;

also they can ask for next smaller left or right

start loop from 0 to 1

stack [top] != arr[i])

Duplicate Parenthesis ☆

↳ we have given an string

((a+b)+(c+d)) ✓ this does not contain any duplicate parenthesis.

(((a+b)+(c+d))) ✗ this contain a duplicate parenthesis.

((a)+(b)) ✓

* Logic → If there is no information bet' two parenthesis then it is duplicate. means they are of no use.

Approach (stack)

① if found and opening bracket or operator (+, -, *, /) or (a, b, c)
Push ();

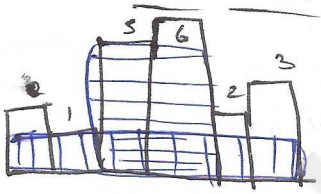
② if found closing
int count = 0;
try to find is closing by poping elements.
count ++ ; for the no. of elements poped.

if (count < 1) means no other data bet^n parenthesis means duplicate. if (count > 1) continue.

Code for this

→ int count = 0
while (s.peek() == '(') {
s.pop()
count ++ }
[count < 1 → duplicate] else (
s.pop() ← '('

Max Area in Histogram



maximum area in rectangle, return
it can be any rectangle

① Area = height x width.

② height → this will be the height of bar

③ width → For this


or


Next Smaller Left
Next smaller right
width = j - i - 1
width of bar bet^n two smaller bar.

Approach

For these we will use the concepts used in next greater

two arrays of nsl (next smaller left)

-1	-1	1	5	1	2	(i)

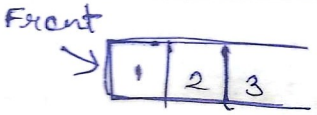
nsr (next smaller right)

1	6	2	2	6	6	(j)

④ then we will written the max area

* when no smaller left put -1
* when no smaller right put n

Queue



q.size()
- size of queue.

↳ FIFO → first in first out

↳ Can be made with Array, L.L, Stack?

↓

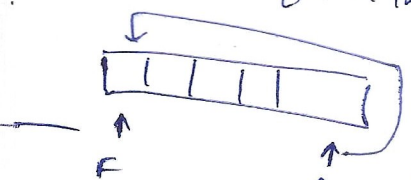
i) fixed size

ii) T.C of remove() O(n)

iii) But we use this for circular queue.

Circular queue.

↳ Here as we get towards end of the array it get to the first index by a special formula.



Rear = (rear + 1) % size.

q.add()

q.remove()

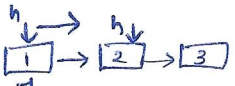
↳ So here the T.C is O(1).

Queue using Linked List

```
=> static class node {
       int data
       Node next;

       //constructor
       Node(int data) {
           this.data = data;
           this.next = null;
       }
   }
```

```
static class Queue {
    static Node head=null;
           tail=null

//empty
Public static boolean isEmpty() {
    if (head=tail=null) {
        return true;}
    }
    return false;
```

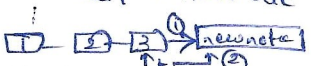
```
//remove
① check is empty
    → return -1;
② int val = head.data;
③ if single element
    if (tail==head) {
    head=tail=null;
    return val;
④ head = head.next;
    return val
```



This will be removed

```
#peek {
① Check isEmpty()
    → return -1;
return head.data;
}
```

```
//add (use FIFO
        First in last out)
Node newNode = new Node(data);
① check isEmpty
    then head=tail=newNode
② tail.next=newNode;
   tail = newNode
```



Queue using JCV

→ Queue is not written here because Queue is not a class.
→ Queue is a interface so we cannot made a class of queue.

```
import java.util.*;
public class Queue_B {
    public static void main(String args[]){
        Queue<Integer> q = new LinkedList<>();
    }
}
```

we can use → LinkedList<>(); or → ArrayDeque<>();

functions
1. add
2. remove
3. isEmpty
4. peek

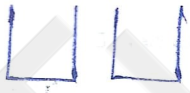
Queue using Two Stacks

by keeping ↓

push O(n) or pop O(n)

```
static class Queue {
    static Stack<Integer> s1 = new Stack<>();
    static Stack<Integer> s2 = new Stack<>();

    // empty
    public static boolean isEmpty(){
        return s1.isEmpty();
    }

    // add
    public static void add(int data){
        // transfering elements from s1 to s2
        while(!s1.isEmpty()){
            int temp = s1.pop();
            s2.push(temp);
        }
        // adding new element
        s1.push(data);
        // transfering the elements back
        while(!s2.isEmpty()){
            int temp = s2.pop();
            s1.push(temp);
        }
    }

    // remove
    public static int remove(){
        if(isEmpty()){
            System.out.println("Queue is empty");
            return -1;
        }
        return s1.pop();
    }

    // peek
    public static int peek(){
        if(isEmpty()){
            System.out.println("Queue is Empty");
            return -1;
        }
        return s1.peek();
    }
}
```

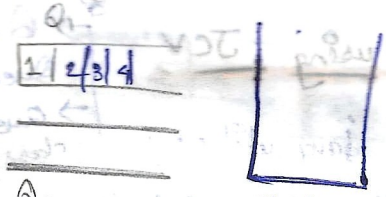
• Google Microsoft



① add
a) s1 empty add here ✓
b) s1 if not → s2 transfer all elements
c) s1 push
d) s2 → s1 push all elements back

• Use the other bucket to store the elements while adding element in first stack.

Stack using 2 Queues



// add (push)

- add the 1st element in Q1
- then add the elements in the queue that have already elements.

Pop

- put the elements one by one in another queue until we get to the last element.
- then at last element return the last element.
- For another pop use the filled queue as the primary one and other as secondary one and repeat the above process.

```
static class Stack {
    static Queue<Integer> q1 = new LinkedList<>();
    static Queue<Integer> q2 = new LinkedList<>();

    // isEmpty
    public static boolean isEmpty() {
        return q1.isEpty() && q2.isEmpty
    }

    // add
    public static void push(int data) {
        if(!q1.isEmpty()) {
            q1.add(data); }
        else { q2.add(data);
        } }

    // remove - O(n)
    public static int pop() {
        if (isEmpty()) {
            System.out.println("Stack is Empty");
            return -1;

        // here there are two cases
        int top = -1;
        if(!q1.isEmpty()) {
            while (!q1.isEmpty()) {
                top = q1.remove();
                if(q1.isEmpty()) {
                    break; }
                q2.add(top);
            }
        else { // case 2
            while(!q2.isEmpty()){
                top = q2.remove();
                if(q1.isEmpty()) {
                    break;
                }
                q1.
```

First non-repeating Letter in a stream of characters

Flipkart

ex- aabccxb T.C → Big O(n)

① freq [] = [26] - for 'a to z' storing the frequency of each character.

② Queue <character>

```
for (int i=0; i<str.length(); i++){
    ch = str.charAt(i);
    q.add(ch)
    freq[ch - 'a']] ++ ;

while (freq[q.peek()] != 1){
    q.remove() }
    if q.empty() → -1
    else print (q.peek());
}
```

output

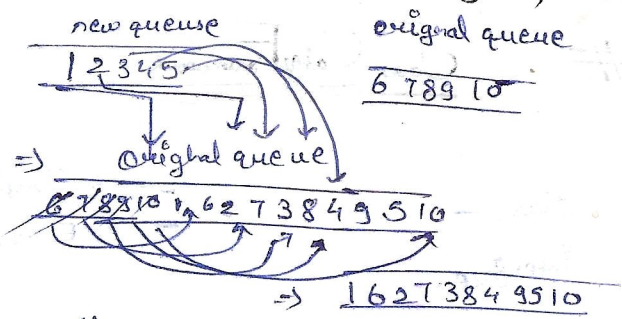
a -1 b bb x

*Almost in all questions in which it is written "stream of characters" Queue is used. ∴ Almost all !

Interleave or shuffle 2 halves of Queue (even Length)

ex- 1 2 3 4 5 6 7 8 9 10

⇒ 1 6 2 7 3 8 4 9 5 10



Step ① - size of q.size()

② Queue first → size/2 // make a new queue with first half elements original que.

③ remove first element from new queue and add at back of original

④ then transfer the first element of original at last

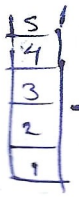
Repeat this until the Queue first became empty.

Queue Reversal

ex- 1 2 3 4 5

final → 5, 4, 3, 2, 1.

(1) Put the elements in stack 1 2 3 4 5 →



→ 5 4 3 2 1

(2) put back in queue

Deque (Double ended Queue)

* This is different from Dequeue

↓

This means removing elements from queue.

⇒ Queue of two mouth.

•Methods

1) addFirst();
2) addLast();
3) ~~add~~ removeFirst();
4) removeLast();
5) getFirst();
6) getLast()

} Meaning as you understand by name.

Implement using JCF (Java collection Frame work)

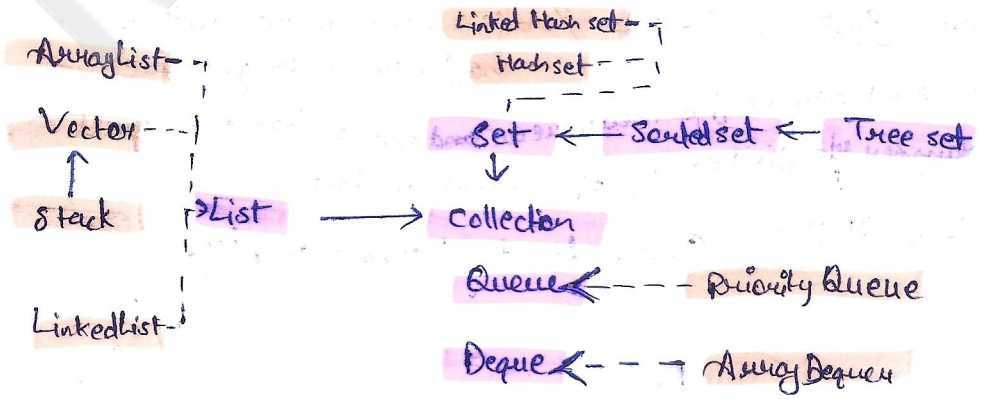
Java Collection Framework

Class —

Interface —

- - - → Implements

⟶ Extends.



Stack & Queue using Deque

→
- Push O(1)
- Pop O(1)
- Peek O(1)

- add
- remove
- peek.

→ **Stack.**

Push → addLast()
Pop → removeLast()
peek → getLast()

Queue

add → addLast();
remove → removeFirst();
Peek → getFirst();

Greedy Algorithm

↳ locally optimum choice at each stage.
and hope to achive a global optimum.

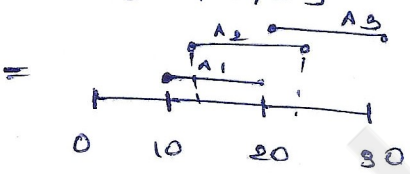
Pros
↳ simple and easy
↳ good enough T.C

Cons
↳ A lot of time
last sol^ is not optimum.

#Q. Activity Selection

↳ start time and end time of a activity will be given
one man can do only one work at a time (No multitask)
→ Find maximum activities that can be performed ?

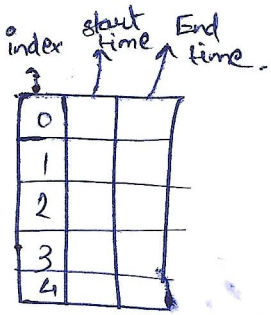
exc -
start = [10, 12, 20]
end = [20, 25, 30]



So max activites = 2
A1 and A3
⇒ because if we start A2 the start time of A3 will be missed and we will be able to do only one activity.

→ ~~So the question is sorted with end time~~
i) end time basis sort
A. first activity
↳ end time

Max no. of activities → Non overlapping (disjoint)
↳ start time >= last choosen action end time
Count ++

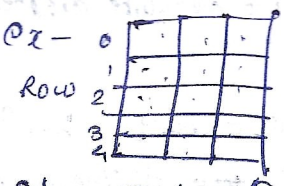


Steps
→ ① sort on end time basis
→ ② first end time will be the end time of last task.
③ for (i=0 ; i < end.length)
if (startTime >= EndTimeArr){
maxAct++;
ans.add (that activity index)
lastEnd = activities[i][2];

Sorting in 2D Array

→ Arrays.sort(activities, Comparator.comparingDouble(o -> o[2]))

on which column the data should be sorted



on the basis of this column the table will sort.

Fractional Knapsack

→ You are given a bag which ~~than~~ can hold limited kg of things, You have different things ~~in~~ different kg's and their sale value is different choose the things so that You have max value.

Value = [60, 100, 120]
weight = [10, 20, 30]

bag can hold = W = 50

(1) steps → find the rate ratio

(2) → make a 2D array having.

index	weight	rate

(3) → if bag can hold everything add everything.

(4) → rather that add that much that the bag can hold

(5) → priority with max rate ~~value~~ object first.

(6) → if not sorted sort the 2D array descending of rate columns.

```
for(int i=n-1; i>=0; i--){
  if(capacity >= weight[i]){
    total include → capacity - wt[i]
                  → value final + val[i]
  }else{
    value = val + (ratio * capacity);
    break;
  }
}
```

Min (positive) Absolute Difference Pairs

2 - 1 = 1 (+ve)

1 - 2 = -1 → |-1| = 1] → to here absolute difference is 1

→ We are given two arrays we need to make pairs bet^n their element as the sum of their absolute ~~sum~~ difference is minimum.

ex- A = [1, 2, 3]
B = [2, 1, 3]

ans = 0

→ So to get minimum sum the difference bet the elements should be minimum.

→ take the difference bet^n the elements as much closer to each other.

→ For this sort both arrays and take difference as their index wise

→ This will gave you minimum difference.

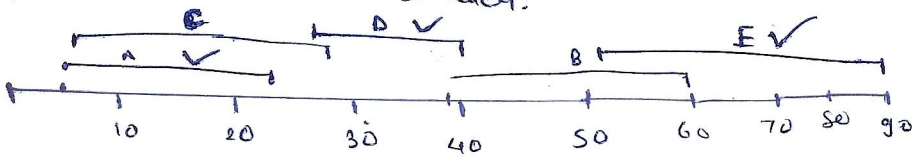
Max Length Chain of Pairs

→ Here some pairs of elements are given you need made a chain with them but if (a, b) and (c, d) are a pair and in chain then b < c ← must.

ex- Pairs =
(5, 24)
(39, 60)
(5, 28)
(27, 40)
(50, 90)

Approach → this question is same as activity selection question.

→ If we arrange them in a line then the next pair will only come their if the earlier pair right element is smaller.



→ So we can pick A D E

Max chain is 3 pairs.

Approach ① Make a 2D array of [index, element 1st, element 2nd]

② Sort on the basis of second element.

③ first pair will be the first indexed pair

④ Lastelement = 2nd element of first pair

⑤ for (i = 1 ; i < n ; i++) {

if (pair(start) > lastend) {

ans++; // increasing element in chain.

Last selected end update }

}

Indian coins

Q You are given a mount ex- 590 you need to give this much to some one in minimum no. of notes

ex - You have [1, 2, 5, 10, 20, 50, 100, 500, 2000].

So You will give 500 x 1
50 x 1
20 x 2

→ So here we have used greedy Approach to solve this - because we have used minimum no. of notes : -

but this approach will not work on every note system India has Canonical coin system so it works here.

Approach -

① Sort descending [2000, 500, 100, 50, 20, 10, 5, 2, 1]

② count = 0.

amount = 590

for (int i=0; i< n; i++) {

if (coin[i] < amount) {

while (coin[i] < amount) {

count ++

amount -= coin[i]; }

}

✱ `Arrays.sort(coins, Comparator.reverseOrder());`

to sort the array in reverse order.

But for this `Integer` this must be used to make array not primitive.

Job Sequencing Problem

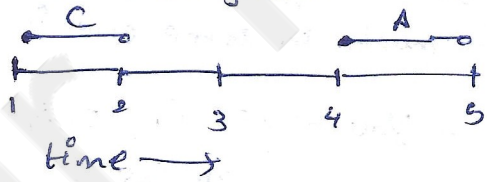
→ We are given some jobs, their dead lines and the payment we will get after doing them. each job takes 1 hour and each job only can be started before their dead line

ex-

	deadline	payment
Job A =	4 ,	20
Job B =	1 ,	10
Job C =	1 ,	40
Job D =	1 ,	30

So use greedy to get max profit

→ we will do job C = 40



now only job A can be done because it had dead line 4.

→ profit = 40+10
= 50

Approach

→ ① Sort Jobs (based on profit)

② time = 0

```
for (int i=0; i< jobs; i++){
  if (job(deadline) > time){
      add job in ans;
      time ++;
  }
}
```

✱ `Collections.sort(jobs, (obj1, obj2) → obj2.profit - obj1.profit);`

↳ code to sort objects.

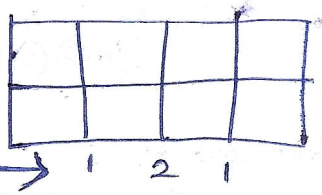
Chocola Problem

Microsoft Google

or Min Cost to cut board into squares.

⇒ We have give a chocolate bar

ex-



we need to break this is small peices such that there should be a single piece.

but the twist is here is that each cut cost something and the cost differ for each cut.

So we have to cut the chocolate in such a way that it cost minimum for cutting it.

4 imp things

i) We have to perform each cut

ii) After each cut chocolate breake in peices an then for each cut we need to pay seprately

iii) So we will do
   a) expencive cut first
   b) cheap cut at last.

iv) Cost of vertical → no. of horizontal peices × cut price.

Cost of horizontal cut → No. of vertical peices × cut price.

We will make two array

vertical cut = [4, 3, 2, 1, 1]

horizontal cut = [4, 2, 1]

→ Move the pointer on both

→ perform the cut first which is costlier.

Pseudo code

sort hori, verti in decending order

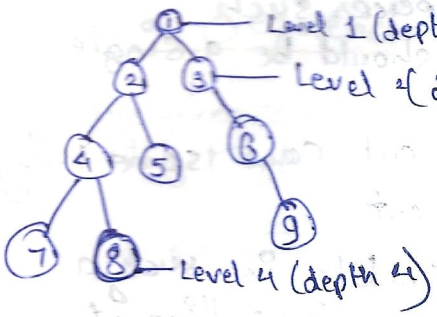
```
int h=0, v=0
int hori = 1, verti = 1
while (h < hori.length && v < vert.length){
    h < v ? verti cut : hori cut
}
At last remaint hori or verti cuts.
```

and add cost++;

Binary Tree

↳ This tree consist of max 2 nodes, 1 or 0.

- Levels and Subtree



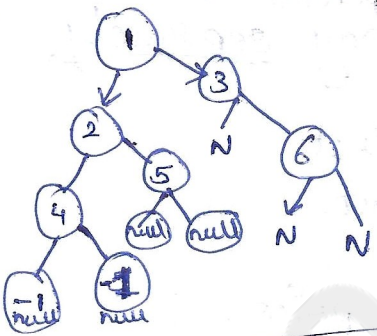
No. of leaves → 4
(7) (8) (5) (9)

Ancestors of 8
→ {4, 2, 1}

- Build Tree Preorder

1, 2, 4, -1, -1, 5, -1, -1, 3, -1, 6, -1, -1

→



-1 = Null

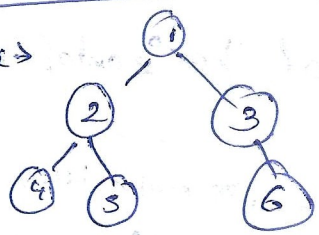
- First take left node then right.
- Null has no further node.
- So then choose right

DFS → Depth First Search

BFS → Breadth First Search

Tree Traversals

ex →



DFS

A) Preorder → Pre because root is printed first.

i) Root
ii) Left Subtree
iii) Right subtree

⇒ 1, 2, 4, 5, 3, 6

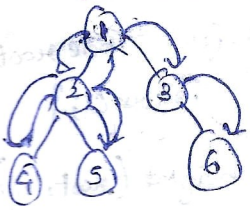
T.C = O(n)

```
public static void preOrder(Node root) {
    // base case
    if (root == null) {
        sysO(-1);
        return;
    }
    sysO(root.data + " ");
    preOrder(root.left);
    preOrder(root.right);
```

b) Inorder → Root comes in middle

DFS

i) Left Subtree
ii) Root
iii) Right subtree



⇒ 4 2 5 1 3 6

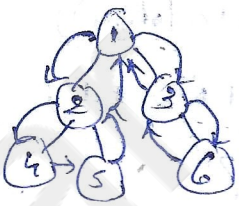
```
public static void InOrder (Node root) {
    if (root == null) {
        return;
    }
    inOrder (root.left);
    Sysp(root.data + " ");
    inOrder (root.right);
}
```

---

c) Postorder → root will be at last

DFS

i) Left subtree
ii) Right subtree
iii) root.




```
Public static void postOrder(Node root)
{ // base case
    if (root = null){
        return;
    }
    postOrder (root.left);
    postOrder (root.right);
    syso( root.data + " ");
}
```

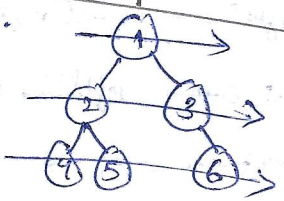
⇒ 4, 5, 2, 6, 3, 1

Breadth First Search

d) Level Order → print level wise.

T.C = O^n

In this we will use a Queue



BFS

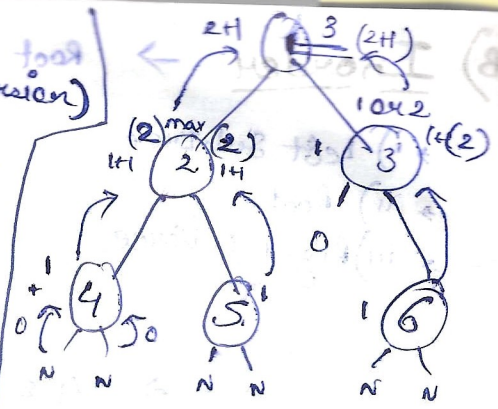
steps →
1) put root in Queue
2) Print element from ques (Pop Q)
3) add left and right element of root to Q of last element
4) also add null after adding both left and right leaves so to know about next lines.
5) another null will be added when the first null will be out of queue
6) If null is out and Q is empty stop.

⇒ 1, 2, 3, 4, 5, 6

Height of a tree (Recursion)

T.C = O(n)

```
static void height (Node root) {
    if (root == null) { return 0; }
    leftHeight = height (root.left);
    rightHeight = height (root.right);
    height = max (lh, rh) + 1;
}
```



Count of Nodes of tree (Recursive)

```
static void Count {
    if (root == null) {
        return 0;
    leftCount = Count (root.left);
    RightCount = Count (root.Right);
    int treeCount = L.C + R.C + 1;
    Sysо (treeCount);
}
```

Sum of Nodes (Recursive)

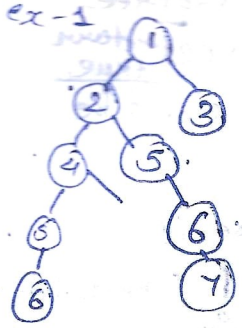
```
static int sum {
    if (root == null) {
        return 0; }
    leftSum = sum (root.left);
    RightSum = sum (root.right);
    int sum = leftSum + rightSum + root.data;
    return sum; }
```

Diameter of a Tree

T.C = O(n²)
(Approach no. 1)

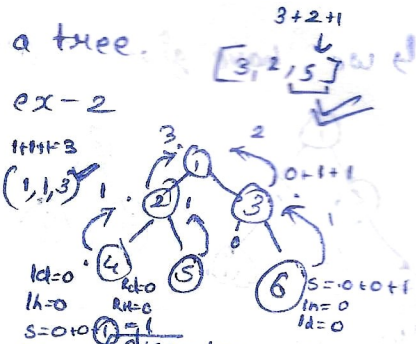
⇒ longest distance bet^n two node of a tree.

ex-1



diameter = 7
here it doesnot pass through root node.

ex-2



diameter = 5
here it pass through root

- diameter of that Node = (left diameter + Right diameter) + 1;

⇒
```
Static int diameter(Node root){
    if (root == null){
        return 0;
    }
    int leftDia = diameter(root.left);
    int rightDia = diameter(root.right);
    int leftHt = height(root.left);
    int rightHt = height(root.right);
    int selfDiameter = leftheight + Right heigh + 1
    return Math.max(selfDiameter, Math.max(leftDia, rightDiameter))
```

Diameter (Approach 2)

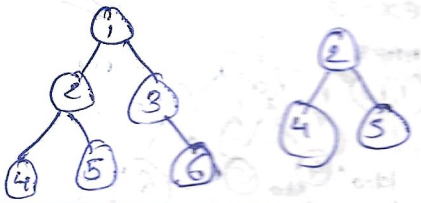
```
class Info {
    int diam
    int height }
```

↳ to save the time we will return both height + diameter using a class.

⇒
```
leftInfo = diam(root.left)   ] → includes [diamet, height]
rightInfo = diam(root.Right) ]   both

newInfo [ finalDiam = max(lInfo.dia, rInfoDia, lht+rht+1)
        [ finalHT   = max(lh, rh) + 1

return newInfo(Diam, ht);
```

Subtree of another tree

↳ we have given a full tree and a sub tree we have to find out is the sub tree is a subtree of Main tree → return true

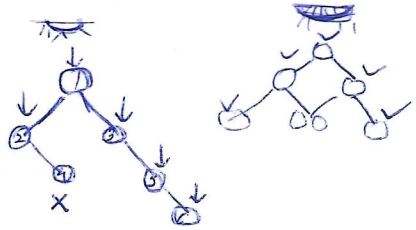


→ Node and position both should be same.

Step:
1. find sub root in tree ← using any traversal
2. check identical (subtree, node subtree)

Return false if
1. node.data != subroot.data
2. node == null || subroot == null
3. left subtree → non identical
4. right subtree → non identical

Top view of a tree



Output : 2, 1, 3, 5, 6

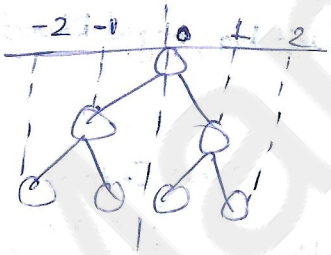
Hash Map →

map(key, value);

1. Create
HashMap<Integer, Integer> map = new Hash
2. Add (put)
map.put(key, value)
3. get
map.get(key)

*Key is always unique

Horizontal Distance



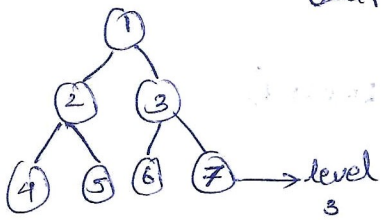
we will use level order traversal
↳ if we move root.right → HD + 1
↳ if we move root.left → HD - 1

Kth Level of a tree

• time Complexity = O(N)

Ex - k = 3

Output : 4 5 6 7



✡ first level will be zero.

Pseudo code:

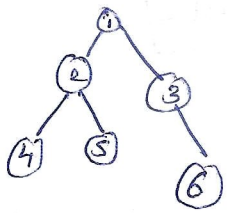
```
k level (root, level, k) {
    if (root == null)
        return;
    if (level == k) {
        print (root.data)
        return;
    }
    Klevel (root.left, level+1, k)
    Klevel (root.right, level+1, k) }
```

as we don't need to go further in tree.

Lowest Common Ancestor (LCA)

↳ the node at (first node) where the next if we go up in tree

Example –



$n_1 = 4,\ n_2 = 6$

output = (1)

$n_1 = (4)\ \ n_2 = (5)$

output = 2

Approach

Make array of path from leaf to root

a) ex – 6 → | 1 | 3 | 6 |

b) 4 → | 1 | 2 | 4 |

Now traverse both array and the last common element will be the LCA

Path finder function

step 1) check if the ~~node~~ node = value if we add to Array list return

2) if no add to Array list and check left and right of that node.

3) if both right and left return null means node is not available in that direction.

4) Remove that node and return false

5) if got false from the ahead node remove the current node and return false.

6) Return true only if get the key

Optimized approach T.C → O(n)

1) root = null
root = n1
root = n2
} return root

leftLca = (Node.left, n1, n2);
RightLca = (Node.right, n1, n2);

2) rightLca = null → return leftLca;
leftLca = null → return RightLca;
right & Left != null → return root;

Min distance bet^n two nodes

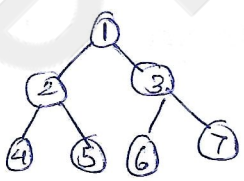
approach → ① dist 1 lca to n_1
② dist 2 lca to n_2

min distance = dist 1 + dist 2

⟹

```
if (root = null)
    return -1;

if (root.data = n)
    return 0;

left n search
right n search.

if (left = -1, right = -1) {
    return -1; }

else if (leftd > -1)
    return leftd + 1)

right > -1
else (return rightd+1);
```

Kth Ancestor of Node



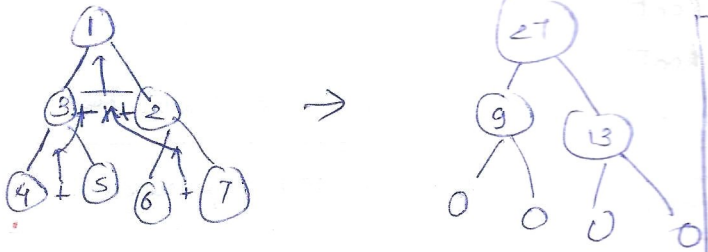
ex- n = 5
k = 2

Output = Node ①

Steps ① find my node

Transform to Sum Tree

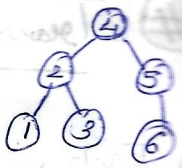
→ Each node = sum of left and right subtree.



approach

```
transform(node root){
    if (root = null){
        return 0;
    }
    int leftChild = transform(root.left);
    int rightChild = transform(root.right);
    int data = root.data;
    int newLeft = root.left == null ? 0 : root.left.data;
    int newRight = root.right == null ? 0 : root.right.data;
    root.data = newLeft + leftChild + newRight + rightChild;
    return data;
```

Binary Search tree



↳ Same as binary tree with some extra properties.

i) Left Subtree Nodes < Root
ii) Right Subtree Nodes > Root
iii) Left and right Subtree are also BST with no Duplicates.

★ So here Inorder Traversal → give sorted sequence

BST search Time Complexity = O(H)

→ if (k = root) { return; }
if (K > root) search at right side
if (K < root) search at left side

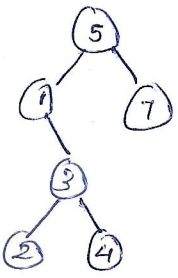
→ But in skewed tree ex-  time = O(n)
worst case

Common Strategy → Most problems will be solved using recursion i.e by dividing into sub problems and making recursive calls on subtrees.

Build a BST

→ Values [] = {5, 1, 3, 4, 2, 7}

- Inorder traversal
→ 1, 2, 3, 4, 5, 7



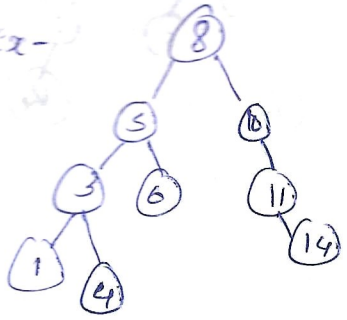
Approach → values[i] > root → put in right subtree
values[i] < root → put in left subtree.

Search in BST

T.C = O(H)

H = height of tree

ex-



key = 1

1. Root > key { search in left tree }
2. Root < key { search in right }
3. Root = key
   - return true
4. Else { if root == null { return false }

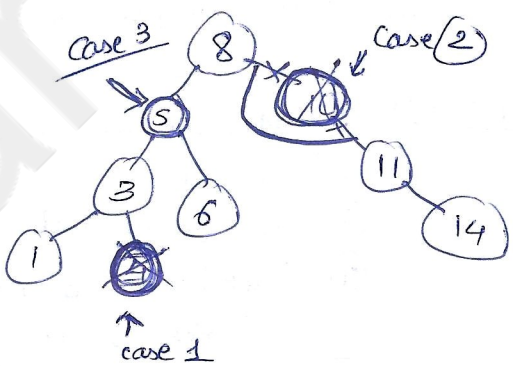
Delete a Node :

Approach

1. Find the node
2. if case (I) return null to the parent.
3. and the node will be deleted auto-matically.

Cases

1) No child (Leaf Node)
2) One child
3) Two child.



Case II

↳ Replace the node with child node

Case III

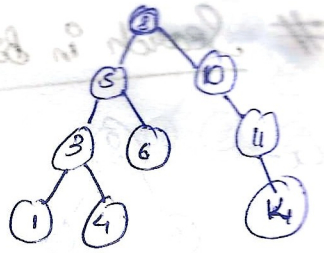
↳ If the node have 2 childs then the node should be deleted and the order of BST should also not be disturbed.

∴ we need to replace that node with inorder successor

1) Replace value with inorder successor
2) Delete the node for inorder successor

Inorder successor → Left most node in right subtree.

Print in Range



- Example from $k_1 = 5$ to $k_2 = 12$

output should be - 5, 6, 8, 10, 14

Case I if $k1 <= root <= k2$

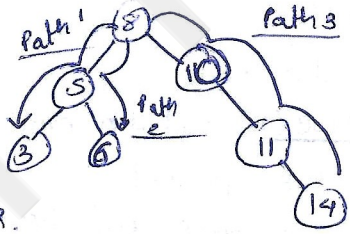
↳ L.R
↳ R.R] If the value of node lies between the range print it.

Case II

root > k_2 → ↳ R.R only

else root < k_1 → L R.L only.

Root to Leaf Paths



↳ we will store this path in array list.

Approach : (1) Add node to array List
(2) travel left
(3) if found null → print path. base case.
(4) back track to parent node and remove the node from arraylist
(5) then traven right.

Validate a BST

⇒ BST will be given and validate it.?

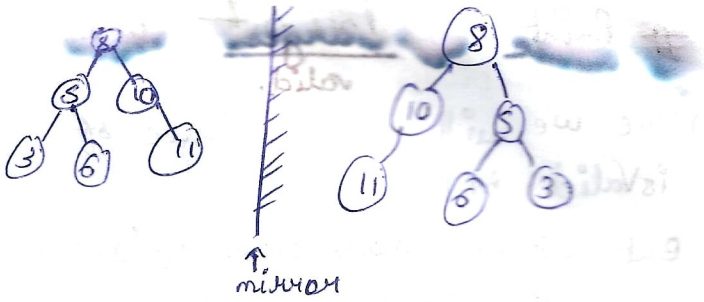
- Approach 1
  If inorder is sorted
  then → BST valid ✓

- Approach 2
  (1) ↳ take out max of left side = ml
  (2) take out min of right side = mr
  if (ml < root < mr) { then BST valid) ✓

Left subtree → min = min
max = root/parent

Right subtree → min = root/parent
max = max

Mirror a BST



Approach

1. take out mirror of left subtree
2. take out mirror of right subtree
3. swap them.

→ Mirror ⇒ ~~root.right~~ root.left ⇔ root.right

Sorted Array → Balanced BST (Minimum possible height)

arr = {3, 5, 6, 8, 10, 11, 12}

To get minimum height recursively →
- find mid
- take it as root
- half elements at left and half elements at right

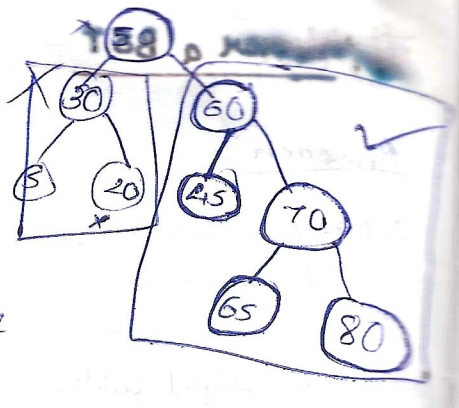
Pseudo code

```
→ CreateBST(arr, st, end){
    if (st > end)
        return null. }
int mid = (st+end)/2
Node root = new Node(arr[mid])
root.left = create BST(arr, st, mid-1)
root.right = create BST(arr, mid+1, end)
return root;
```

Convert a BST to Balanced BST

Approach → (1) take out inorder sequence of BST
(2) Convert the arr to Balanced BST

Print Largest Bst
valid.



↳ Here we will use the logic of isValid BST

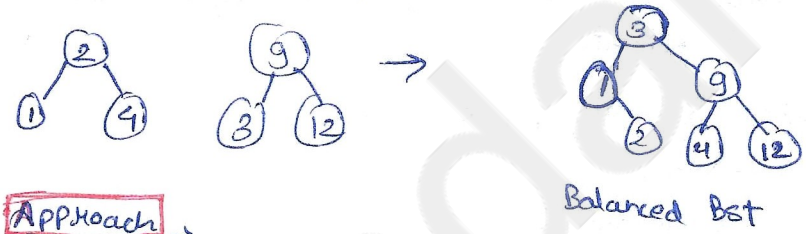
↳ But we will made class of "Info"

This whole thing will be calculated for each node
1. isBST (boolean) true or false
2. size = leftsize + rightsize + 1
3. min = Math.min(root.data, Left.min, right.min)
4. Max = Math.max(root.data, Left.Max, right.Max)

↳ we will make a static variable (max size)
when the isBST → true
max = Math.max(max, curr Info);

↳ And we will return max size.

Meaurge 2 BSTs



Balanced Bst

Approach →

1. BST 1 inorder sequence (sorted)
2. BST 2 inorder sequence (sorted)
3. Meaurge both arrays.
4. Sorted array → Balanced BST

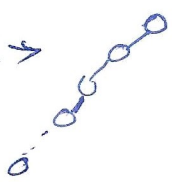
AVL Trees

→ only concept asked

Self-Balansing BST

↳ we required Balanced tree to reduce T.C

for ex - 10^5 nodes if it is skewed → tree
then TC = 10^5
but if balanced → 5 log 10
then



And $5 \log 10 << 10^5$

Heaps / Priority Queues

↳ give first priority to the element you want.

- Priority Queue using JCF

↳ `PriorityQueue <Integer> pq = new PriorityQueue <>();`

Functions ↳ At default the priority in integer is from lowest number → highest number.

1. add() TC O(log n)
2. remove() TC : O(log n)
3. peek() TC : O(1)
4. isEmpty()

To travel through Queue for each loop →

```
for(int x : q){
    if(x == k) return true;
}
return false;
```

To change the priority (decending type) add this

`PriorityQueue <Integer> pq = new PriorityQueue <> (Comparator. Reverse`

Priority Queue of objects. → For this we need to make our class comparable (Means give it the power so that it can compare)

↳ Syntax:

```
static class Student implements Comparable<Students> {
    String name;
    int rank;
    public Student(String name, int rank){
        this.name = name;
        this.rank = rank;
    }

    @Override
    public int compareTo(Student s2) {
        return this.rank - s2.rank;
    }
}
```

this.rank - s2.rank

S_1 [15] S_2 [15]

if $S_1 < S_2$ → -ve → S_2 is greater

$S_1 > S_2$ → +ve → S_1 is greater

$S_1 = S_2$ ⇒ equal.

Output: Output will be sorted on the basis of rank

→

ex -	rank
A	2
C	3
D	4
B	8.

* To make the order reverse use (Comparator.reverseOrder()) inside PriorityQueue argument section

---

- **Heap** { Max heap → PQ → max element ↑ priority
  Min heap → PQ → min element ↑ priority

- Properties

① Binary tree
↳ At most 2 children

② Complete binary Tree (CBT)

↳ CBT is a binary tree in which all levels are completely filled except, possibly the last one
↳ It is filled from left to right before new leaf.

→ Means a level should be completely filled before starting to put the elements in new level

ex→ ✓ ✗ ✓ ✗ ✓

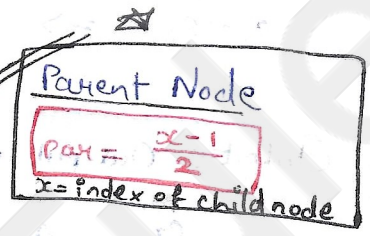
(3) Heap Order Property

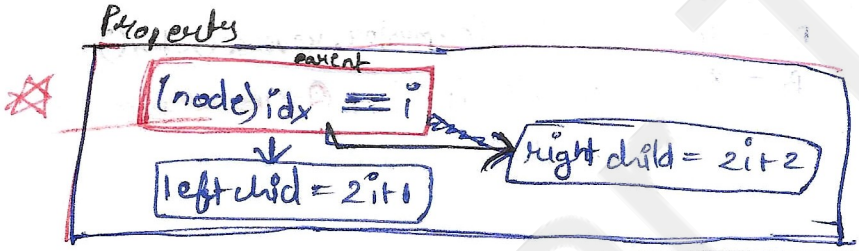
↳ a) Children node >= Parent (Min Heap)

b) Children node <= Parent (Max Heap)

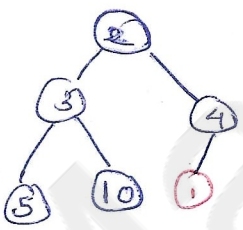
Heap Implementation ← we do it with the help of arrays

- We cannot use do this using class beause then the add function will take more than Log(n) time so we use Array or Arraylist.





Insert in Heap T.C = log(n)



2 | 3 | 4 | 5 | 10

→ 2, 3, 4, 5, 10, 1 (indices 0 1 2 3 4 5)

par swap

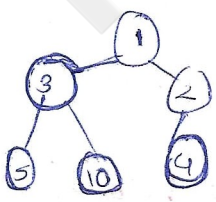
→ par = (5-1)/2 = 2

2, 3, 1, 5, 10, 4 (indices 0 1 2 3 4 5)

par = (2-1)/2 = 1/2 = 0.

(child < par) → swap

→ 1 3 2 5 10 4



Step ① - add at last index

Step ② - fix heap

Step③ - par = (x-1)/2 while

{Step④ - if (child val < par val)

swap (child, par)}

Get min in Heap.

Peek() → we know that in Min Heap minimum element of the Tree is root

```
public int peek() {
    return arr.get(0);
}
```

Delete in Heap → T.C = O(log n)

Steps 1) swap 1st and last node
2) remove last idx in array list
arr.remove(arr.size()-1).
3) fix heap ↓ using Heapify

Q. Remove min element

Heapify Function

recursive function

→ ① It compares root, leftchild, rightchild.
② put the min at root = i
→ 2nd min at left root = 2i+1
→ and large at Right root = 2i+2

Heap Sort → Ascending (Max heap) Descending (Min heap)

arr = 1, 2, 3, 4, 5

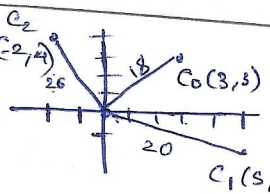
→ Max heap will bring the largest element at top (root)
→ the exchange it with last
→ heapify (size 1 se kam karke)
→ fir apna array sorted

Steps ① arr → Max heap on last level Non-leaf Nodes ↓ heapify() ↳ (maxheap)

Step ② → ⇒ Swap (1st and last) ⇒ heapify remove last

for (0 to n-1)

Nearby Cars → we are at origin (0,0) Q. find k nearest cars? k = 2



dist = $\sqrt{a^2+b^2}$

→ distance from origin →
C0 → 18
C1 → 26
C2 → 20

Approach
① → find the distance
② → Put that value in PriorityQueue
③ → take out first k values as your ans

Overside
Public int compareTo(Point p2){
return this.distSq - p2.distSq; }
→ comparing on the basis of distSq

• **Connect N Ropes** → but each time when 2 ropes connected they have charge. charge = length of x rope + length of y

[4, 3, 2, 6]

→ we want to connect all rope in minimum charge

→ As the charge is added each time so to have a minimum charge we will connect smallest rope first Doing this the charge will be lowest

→

	Charge
2+3	5
5+4	9
9+6	15
	29 //

Approach → ① Add all elements in PQ
② Remove first ② element
③ add them and push in PQ
④ do above 3 and 4 steps until PQ gets ~~empty~~ 1 element
⑤ last removed is ans wer.

✱ This question looks like solved by Greedy algorithm as just sort and ~~start~~ adding them but No it gives wrONG Answer

---

Weakest Soldier → 1 represent soldier
0 represent civillians.

ex → m=4, n=4, k=2

1 0 0 0 ← most weakest
1 1 1 1
1 0 0 0 ← second weakest
1 0 0 0

ans → R0, R2

Row is weak if
→ ① It is at top
② It has less no. of soldier.

Q. Find weekest (first two row) ?

Approach :

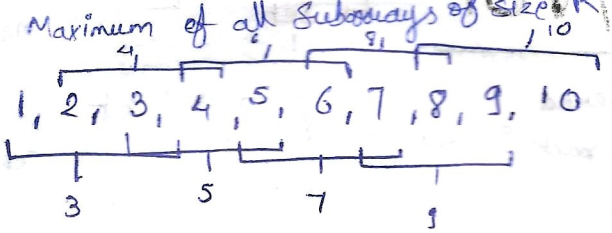
```
static class Row implements Comparable<Row> {
    int soldiers;
    int idx;
    Public Row(int soldiers, int idx) {
        this.soldiers = soldiers;
        this.idx = idx;
    }
    @Override
    Public int compareTo(Row r2) {
        if(this.soldiers == r2.soldiers) {
            return this.idx - r2.idx;
        }
        else {
            return this.soldiers - r2.soldiers;
        }
    }
```

If soldiers are same then compare on the basis of index

else compare on the basis of soldiers.

Sliding Window Maximum

Maximum of all subarrays of size K



k = 3
window of length 3

we need to take out maximum of three element

→ Approach ⊙ we will make max heap

(i) add K nums to PQ (1st window)

(ii) (i) PQ.peek() → window[0]

(ii) while (PQ.peek().idx <= (i-k)) PQ.remove()}

(iii) PQ.add(element next from arr)

(iv) wind[i] = pq.peek()

here as in the PQ the top element will be maximum then check weather the element at top is of that current window or Not

→ if not then remove it

→ otherwise keep it and then add the next element.

⊙ Adding many elements in PQ will do no harm at the top there always be the max element,

⊙ but check weather the element is from that current window or not

Length of result array

→ (n-k+1)

→ After making Priority Queue

```
// 1st window
for(int i=0; i<k; i++) {
    pq.add(new Pair(arr[i], i));
}
res[0] = pq.peek().val;

for(int i=k; i<arr.length; i++) {
    while(pq.size() > 0 && pq.peek().idx <= (i-k)){
        pq.remove();
    }
    pq.add(new Pair(arr[i], i));
    res[i-k+1] = pq.peek().val;
}
```

adding elements of 1st window

taking out first max window element

if the element in top is not of current window

remove it

HASHING

↳ Map → Hash map, Linked Hash map, Tree map
↳ Set → Hash set, Linked Hash set, Tree set.

↳ unordered (we get data back in random form)

Hash Map

(key, value)
↓
always unique

import java.util.HashMap;

HashMap <String, Integer> hm = new HashMap<>();
column one datatype, column 2 datatype

1. Put(key, value) → to update value
   ⇒ hp.put(key, value) → or to make new node
2. get(key) ⇒ hp.get() → To get the value
   → hm.get(key);
   if (key) is not present then You will get null value.
3. containsKey(key) → hp.containsKey() → To check is the data with that key is present or not
4. remove(key) → hp.remove() → To remove (key and value) Node.
   if passed a [key] ← Not existing ⇒ Null
5. Size → hp.size();
6. Empty → hp.isEmpty();
7. Clear → hp.clear();

keys	values
"one"	1
"two"	2
"three"	3
"four"	4

Iteration on HashMap

Set<Integer or String> keys = hm.keySet(); ①

or ② hm.entrySet()
↳ For all values set.

↳ this stores all the keys in set
ex- from last keys = [one, two, three, four]

- Foreach →

```
for(String K : keys){
    System.out.println("key="+K+" ,value="+hm.get(K));
}
```

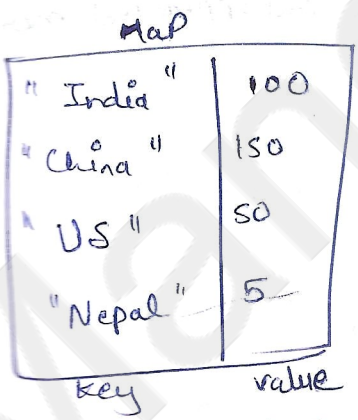
here → K will like Iterator.
→ it will iterate till the end.

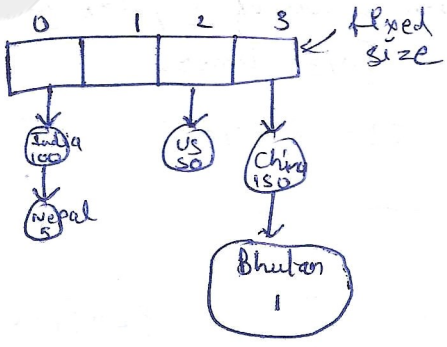
Implementation HashMap

was asked in uber to implement hash

Put(), get(), containsKey(), remove(), size() O(1)

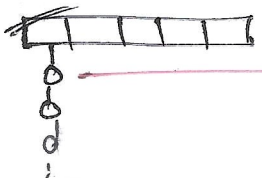
→ this all works in the java in the form Array of linked List





special Note

Rare case
Worst time complexity will be O(n). if we have such data whose bucket index giving by hash function is same



* When the datatype of the variables is not fixed we use generic to make them
→ ex <K, V> this way any datatype can come
→ used in implementation of hashmap.

Linked Hash Map → everything same just it maintain the order in which the elements are inserted in it

Keys are insertion order → It uses Doubly Linked List to maintain the order.

```
LinkedHash<  > hm = new LinkedHashMap<>();
```

Tree Map → put, get, remove are O(log n) (BST)

→ Keys are sorted → Here in implementation Red Black trees are used which are (self balancing trees)

```
TreeMap<  > hm = new TreeMap<>();
```

Q. Majority Element → TC = O(n)

given an array of n, size find the elements that appeared more the [n/3] times.

⇒ nums[] = {1, 3, 2, 5, 1, 3, 1, 5, 1}; n = 9

output = 1

1 is n/3 = 3 greater

ex nums [ ]
{1, 2}

output) {1, 2}

```
for(i=0 to arr.length)
    num = arr[i]
    if (map.containskey(num))
        map.put(num, map.get(num) + 1);
        adding with increasing frequency.
    else
        map.put(num, 1);
```

To traverse on a Map we need Keyset

```
Set<Integer> keys = map.keySet();
for (Integer key : keys){
}
```

- Shortcut for increasing frequency

* if the key exist then it will return it's frequency and if not then 0.

```
map.put(arr[i], map.getOrDefault(arr[i], 0)+1);
```

- Shortcut for "No need to store keyset in any other place"

direct implementation

```
for (Integer key : map.keySet()){
    if (map.get(key) > arr.length/3){
        Syso(key)}
}
```

Valid Anagram. T.C = O(n)

ex - "race" = "care"

Same no. of letters
Same value of letters with same frequency.

Approach :

1. HashMap → we will — HashMap<Character, Frequency> (char, int)
2. then when we will traverse the second string we will delete or remove the elements from map.
3. if Map is empty return True or False.
   if found such a element which is not available then return False

Hash set → implements using hash map.
↳ Collections of unique elements.

- No duplicates
- Un ordered
- Null is allowed

1) add (key)
2) contains (key)
3) remove (key)
4) set.size();
} O(1)

5) set.de...
6) set. is E...

```
HashSet < Integer > hs = new HashSet <>();
```

Iteration on HashSet

(1) Using Iterators

→ Provides a iterator
it → null
points towards null.

```
Iterator it = set.iterator();
while(it.hasNext()){
    print (it.next())
}
```

→ Checks is their is such a element which is not traversed till now.

→ Then automatically it shifts to next it.

(2) Using Advance loop

→ for each.

→ ~~for(key: keys)~~

ex-
```
HashSet <String> Cities = new HashSet<>();
Cities.add ("Nagpur");
Cities.add ("Mumbai");
```

```
for(String city : Cities){
    System.out.print(City+" ");
}
```

Linked HashSet → Implemented using linkedHash Map.
used doubly LL.

- Ordered elements.

T.C → remains same
just little performance drop.

```
LinkedHashSet <Integer> lhs = new LinkedHashSet<>();
```

Tree Set → tree map (Red Black) (self Balancing trees) T.C = O(log N)

- Sorted in according order
- Null is not allowed.
- No duplicates.

```
TreeSet<String> ts = new TreeSet<>();
```

Other properties are same.

t.clear();
t.isEmpty();

#6Q Count Distinct elements

→ use set as it contains no duplicates.
→ return set size.

#Q. Union and Intersection of arrays T.C = O(m+n)

arr1 = {7, 3, 9}
arr2 = {6, 3, 9, 2, 9, 4}

a) Union → take a set of all elements.

b) Intersection → i) add elements of one set
ii) traverse on the array of second.
iii) if the element present in set
↳ a) count it;
b) remove that so can't be count again.
iv) print count.

---

6. Find Itinerary from Tickets

Journey.
→ print all the places line wise, you visited.

Approach
i) find the starting point
↳ a) Make a reverse map.
b) find the place which is the start
c) if the place exists in key of main map but not in to

Key	Value
Chennai	Bengaluru
Mumbai	Delhi
Goa	Chennai
Delhi	Goa

to	from
B	C
D	M
C	G
G	D

→ Then just go from ~~that~~ key to value until last value is not a key.
Key to value

Largest subarray with sum = ~~target~~ 0

ex target = 0 → this much has no value ∴ 0

ex → {15, -2, 2, -8, 1, 7, 10, 23}

15 | 13 | 15 | 7 | 8 | 25 | 35

0 1 2 3 4 5 6 7

ans = 5 largest subarray with

1) brute force → nested loops

(key) Sum	(value) index
15	0
13	1
7	3
8	4
25	6

if (found and same sum) then

MaxSubarray = Math.Max(index) Map

because as we want sum = 0

So if any key is same then we can subtract the from it and get sum as zero

Count no. of subArrays with sum K → T.C ⇒ O(n)

Approach

Dry run → i, j

arr = {10, 2, -2, -20, 10}

K = -10

i → ith index
j = jth index

So Sum[0,j] - sum[0, i-1] = sum(i,j)

we will make Hash Map

∴ Sum[j] - sum[i] = k

~~Sum[j] = k - sum[i]~~

Sum[j] - k = sum[i]

cumulative sum, given

(0, 1) ← this should be already added otherwise it will never find a 0 and count will not increase

(sum, count)

→ ① First we will add (0, 1) in hash map k = −10

② we want K sum as a subarray.

So if current sum − k = 0 then it's equal to K

③. Each time we will check the

④ take sum = 10 2 −2 −20 10 did 0 and also frequency increases.

[10, 2, −2, −20, 10]

ans = 0

10 − (−10) = 20 does not exist

ans + 1

∴ ans = 1

12 − (−10) = 22 does not exist

10 − (−10) = 20 does not exist

−10 − (−10) = 0 exists

ans + get(10)

= ans + 2)

→ ans = 3

ans + get(0)

0 − (−10) = 10 exist and

(0, 1) ← already

(10, 2)

(12, 1)

(−10, 1)

Tries

→ Prefix tree
or Retrievel tree / Retrievel tree

ex - There, The, The → prefix

Questions are directly asked in companies like Google, Amazon, Microsoft.

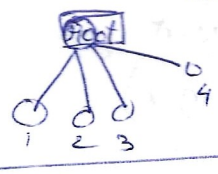
↳ Same prefix are saved only once
↳ used in string question

- What is Trie?

→ It is a (K-ary) tree.

→ Saves time as height is less

ex- K = 4 →

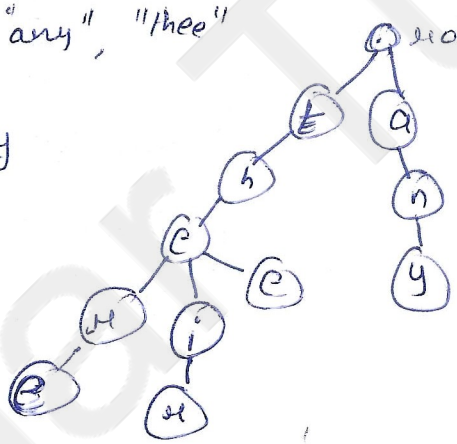


root (empty)

words[] =

⇒ "the", "a", "there", "their", "any", "thee"

→ As prefix is stored only once so time is saved.



⇒
```
class Node {
    Node[] children; 26 letters
    boolean endOfWord;
}
```

→ each node has 2 info
1. the letter array
2. end of word

(1) → to store which letter is it holding.

(2) → to check whether the word ends here or not.

Making trie

```
Static class Node{
    Node children[] = new Node[26];
    boolean eow = false;

    Node() {
        for(int i=0; i<26; i++){
            children[i] = null;
        }
    }
}
```

and also a empty root node

⇒ public static Node root = new

↓ this root
it is always empty
just stores which elements used in trie

Insert in trie — TC → O(L) (L = level of letters)

when we mins t-a → we will get idx
ex- b-a = 1 ∴ b is at 1st idx

1. After the root we will check level by level
2. like this each letter by letter will be stored.
3. And the place where word ends there we will store eow = true (end of word)

Pseudocode

curr = root
for(level = 0 to word.length())
if (curr.children[ch-'a'] == null) ✓
↳ new node create
else ✗
curr = curr.children[ch-'a']
idx

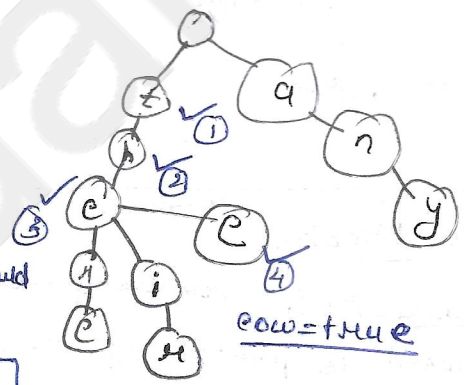
```
Public static void insert(String word) {
  Node curr = root;
  for(int level=0; level<word.length(); level++)
  { int idx = word.charAt(level) - 'a';
    if(curr.children[idx] == null){
      curr.children[idx] = new Node();
    }
    curr = curr.children[idx];
  }
  curr.eow = true
}
```

~~Word in Trie~~

Search in a Trie

* Can be asked as a seprate question

ex- "Thee"



1. check at each level
2. so at each level char should present.
3. And last node eow = true
∴ word exists

```
curr = root
for(level = 0 to key)
{ idx = key.charAt(level) - 'a';
  if(curr.child[idx] == null) { return false; }
curr = curr.children[idx]; }
return curr.eow == true;
```

This should also be the end of word.

Word Break Problem

Google

T.C ⇒ O(n).

key = "ilikesamsung"

words[] = { i, like, sam, samsung, mobile, ice }

Q. Can we break the key such that each part is present in the dictionary / words[] as a words?

⇒ i like samsung → return true.

Approach. — (1) create a trie from string.

```
public static boolean wordBreak(String key) {
    # base case
    if(key.length() == 0) return true;
    for(i=1 to i<=keylength)        // As the substring function is inclusive
        if(search(key.substring(0,i)) && wordBreak(key.substring(i))) {
            return true;
        }
    return false
}
```

Prefix Problem

30-40 min

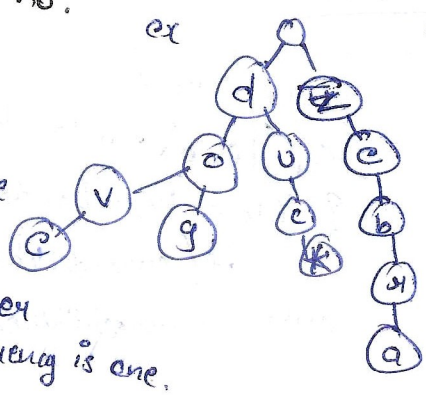
Microsoft, google

Find shortest unique prefix for every word in a given list.

Assume no word is prefix of another. → ex- app, apple ✗ no such case

⇒ Uska short name banao (nickname) aur sabka (nickname) unique ho.

⇒ ex- dov, dog, du, z.



⇒ Approach (1) trie create.

(2) also add frequency feature in trie.

(3) if the letter/char repeated increase the frequency.

(4) the traverse each word letter by letter till you find a letter whose frequency is one.

(5) ~~code~~ and then you will find unique prefix for every word

root.freq = -1; T.C = O(L)

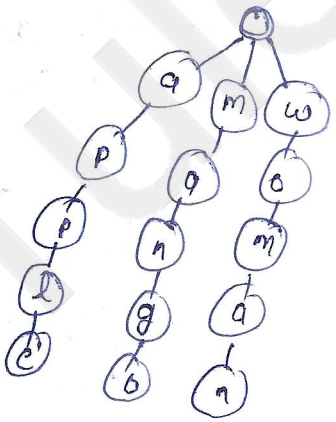
Q. Starts With Problem

↳ creates a function boolean which find the prefix is present or not

ex- words[] = { "apple", "app", "mango", "man", "woman" }

Prefix "app" output: true, as in apple app

Prefix "moon" output false



⇒ Approach → ① create trie.

② check one by one character in trie.

```
boolean startsWith(String prefix){
    Node curr = root;
    for(int i=0 to prefix.length())
        idx = prefix.charAT(i) - 'a';
        if (curr.children[idx] == null)
            return false
    else { curr = curr.child[idx]
    }
    return true;
}
```

Count Unique Substrings

Microsoft | Google

↳ a string is given count unique number of substrings.

ex- str = "ababa" → ① a ② ab ③ aba ④ abab ⑤ ababa ⑥ b ⑦ ba ⑧ bab ⑨ baba ⑩ "" ← empty string.

ans = 10

If you want unique strings all

⇒ unique substring = all prefix of all suffix unique

Imp

Trie stores unique prefix → count of Node of trie

Approaches

1. find all suffix of string
2. Creat trie and insert all suffix
3. Count nodes of trie
   - ⇒ Unique prefix = Unique Sub-string //

- suffix
  - ⇒ for(int i=0 to n) {
    str = str.substring(i)
    insert trie(str)
    }
- ⇒ Count nodes as Count in BST

```
int CountNodes(root) {
        if root=null;
        return 0; }
    Count=0;
for (int i=0 to 26)        <-- As we have 26 letter
  if (root.child[i]!=null)  <-- so we will check if further node is present or not each time.
  Count = CountNodes(root.Child[i])
  return count+1;
```

Longest Word with all Prefixes

⇒ ex- words = ["a", "banana", "app", "appl", "ap", "apply", "apple"]

ans = "apple"

⇒ ↳ because a ✓, ap ✓, app ✓, appl ✓, apple ✓

apply is also an answer

but apple < apply (lexicographically)

(If any question has ~~goes~~ the solution that goes with prefix remember Trie)

So we can solve this question using trie

→ we want a longest word from a trie who's all eof at each letter is true

→ this means all prefixes of word is present in the dictionary

Approach →

~~1~~ count the no. of root → children
↳ ① ! null
② eow = True.

Pseudo code

→ static String ans = "" // final ans.

void longest word (root, temp) { ← StringBuilder temp;

if (root == null) return ← // base case

for(int i=0. to 26)

if (root.child[idx] != null && child[idx].eow == true)

temp + addchar(root)

if (temp len > ans len)

ans = temp

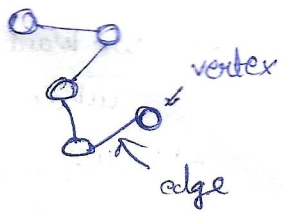
longestword(root.child[i], temp)

temp → delete last char of temp.

Here we did not want to sort the answer as in trie the letters are stores from a → z index ∴ it is already lexicographical order.

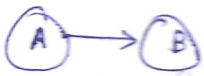
Graph

→ network nodes



Edges

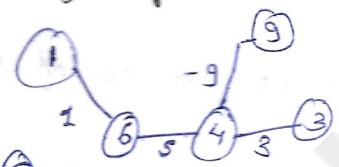
(1) Uni-Direction



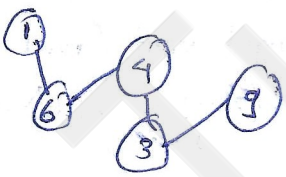
(2) Bidirectional / Un-Directed

Sometime
- weight of edge

ex-



1) Weighted undirected



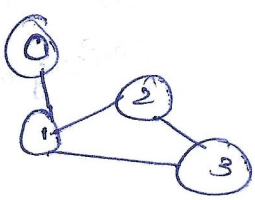
1) Unweighted undirected.

Storing a Graph

(1) Adjacency List → List of List
(2) Adjacency Matrix
(3) Edge List
(4) 2D Matrix (Implicit Graph)

⇒ Adjacency List → List of list

→ example-



⇒ 0 → (1) list[0]
1 → (0, 2, 3)
2 → (1, 3)
3 → (1, 2)

→ Data structures we can use

1) ArrayList<ArrayList>
2) Array<ArrayList> ——— ✓ we will go with this
3) hashmap<int, list>

List of Lists

① takes no extra space
② optimized

Array of Array Lists <Edge>

Edge: Src, destination, cost (weight), standard = 1

Type 2

- Adjacency Matrix

space → $O(V^2)$ V = vertices.

	0	1	2	3
0		1		
1	1	0	1	1
2	0	1	0	1
3	0	1	1	0

(i, j)

i — j if vertex → 1

i ↛ j if not → 0

disadvantages

↳ In time complexity can go till so big.

Type 3 Edge List → It stores the information of all the edges → we use this for some question

→ {{0,1},{1,2},{1,3},{2,3}}

if with edges → {{S, D, W}}

S: source, D: destination, W: weight

Type 4 Implicit Graph

↳ In some questions if it is given that you need to go from one point to another in 2D Matrix then it is the use of implisit graph.

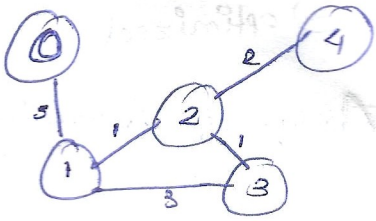
Application of Graph

1) MAPS (sorted network).
2) Social Networks.
3) Delivery network.
4) Physis and Chemistry (Molecules.
5) Routing of algorithm.
6. Machine learning Computational graph)
7. Dependancy Graph ex-servers,
8. computer vision → image segmentation.
9. Graph Databases ex-nebula.
10. Research (to map neurons)

Creating a Graph (Adjacency list)

ArrayList<Edge> graph

Edge = (src, dest, weight)



code

```
static class Edge {
    int src;
    int dest;
    int wt;
    Public Edge(int s, int d, int w){
        this.src = s;
        this.dest = d;
        this.wt = w;
    }
}
public static void main() {
    int V = 5;
```

Ex:

0 - {0,1,5}

1 → {1,0,5} {1,2,1} {1,3,3}

2 → {2,1,1} {2,3,1} {2,4,2}

3 → {3,1,3} {3,2,1}

4 → {4,2,2}

V = no. of nodes

```
ArrayList<Edge>[] graph = new ArrayList[V]; // null → empty arraylist
```

Array of ArrayList | which stores edge | name | no. of vertex

```
for (int i=0; i<V; i++) {
    graph[i] = new ArrayList<>();
}

// 0-vertex
graph[0].add(new Edge(0,1,5));

// 1-vertex
graph[1].add(new Edge(1,0,5));
graph[1].add(new Edge(1,2,1));
graph[1].add(new Edge(1,3,3));
```

same of 2,3,4 vertex.

Now printing 2's neighbours

→
```
for (int i=0; i<graph[2].size(); i++) {
    Edge e = graph[2].get(i)
    System.out.println(e.dest);
}
```

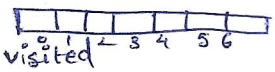
Graph Traversals

can be asked directly as a question

1. Breadth First Search (BFS)
2. Depth First Search (DFS)

Breadth FS → go to immediate neighbour first

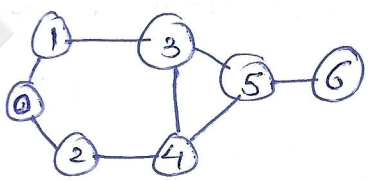
↳ we need to have a boolean array of vissited
↳ because we have cycle.
↳ so that each node is visited once.



empty queue list

→
1. get on any node
2. add its neighbours in Queue.
3. print that node and mark visited.
4. ~~get~~ pop element from Queue
   - if visited ✓
     - a) → remove from Queue
     - b) → (4) pop another elemen
   - if it is not visited then step
     - a) → (2) (3) (4)
5. while the queue is empty.

ex-



← this code is written in Computers.

T.C → O(V+E) → because we travel all vertices and edges.
(V = vertices, E = edges)

But if we used matrix here then the T.C will be $O(V^2)$ so we used adjancy list.

Output :
1, 2, 3, 4, 5, 6

DFS → keep going to first neighbour.

(depth first search) → can be done using stack or recursion (implicit stack)

Approach

(Recursion)

① dfs(graph, curr) { curr → visit → then vis[curr] = true

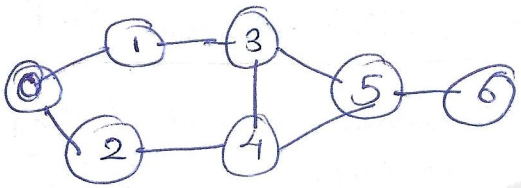
② for(int i=0 to k)

(!vis[curr])

dfs(neighbour(i))

}

vis → array of visited



T.C = O(V+E)

(vertices + edges)

Output : 0, 1, 3, 4, 2, 5, 6

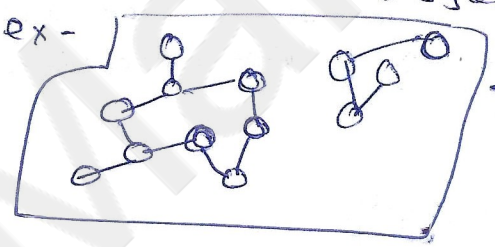
Q. Has Path ? src = 0, dest = 5

Is there any path from zero to 5?

→ there can also be disjoint graphs

T.C = O(V+E)

ex -



← this is a one graph.

Approach

→ i) src = dest = true

ii) or src → neighbour → dest.

hasPath(G, src, dest, vis) {

① src = dest return true;

② if(!vis[neighbour] && hasPath(neighbour, dest, vis)) return true;

return false;

}

② return