Kunal Kushwaha

08/12

Data structures ( ) ;

(1) Linear Search Algorithm :

→ size of array.

✓ Time Complexity : O(N) = worst case

O(1) = Best case

↳ constant

Time Complexity:

How our time will grow, as the input will grow.

2-D array :

```
static int[] function() {
    for (row = 0 ; row < arr.length ; row++) {
        for (int col = 0 ; col < arr[row].length ; col++) {
            if (arr[row][col] == target) {
                return new int[] {row, col};
            }
        }
    }
    return new int[][] {-1, -1};
```

Returning 2 array from function

Enhanced for-loop for 2-d arrays:

```
for (int[] a : arr)        // arr → array name
{
    for (int b : a) {
        if (b > max)
            max = b;
    }
}
```

Shortcut to find number of digits in number.

Ex: 1375

```
return (int)(Math.log10(1375)) + 1;   => 4
```

No. of digits is 4.

Binary Search Algorithm :

(Binary search is used only for sorted arrays (means asc(or)des order))

→ Take the middle element in array.

→ If target > middle element
↳ search in right part.

else
search in left part.

→ If middle element = = target element
// easy.

(mid = 4+5/2 = 4)

middle element

arr = [2, 4, 6, 7, 11, 12, 14, 20, 36, 48]
0 1 2 3 4 5 6 7 8 9

// better way to find this :

m = (s+e)/2 → If these are large values, This may exceed int range

m = s + (e-s)/2.

```
static int BinarySearch(int arr[], int target){
    int start = 0;
    int end = arr.length-1;
    while(start <= end){
        int mid = s+(e-s);
        if(target < arr[mid]) end = mid-1;
        else if(target > arr[mid]) start = mid+1;
        else return mid;
    }
    return -1;
```

Order-Agnostic Binary Search :

→ If we don't know whether the array is sorted in asc(or) descending then check first and last element of array.

if arr[s] < arr[e] ⇒ ascending.
arr[s] > arr[e] ⇒ descending.

Why only first & last.
If arr = [3, 3, 4, 5, 7]
(1)
It occurs problem

Problems: (Sequence of numbers that are sorted)

If problem give sorted array → First apply Binary search

① Ceiling of a given number.

arr = [2, 3, 5, 9, 14, 16, 18] , target = 6.

Ceiling = smallest element in the array greater or = target.

Ceiling (arr, target = 15) ⇒ (16) Ans.

Ceiling (arr, target = 4) ⇒ 5.

Ceiling (int arr[], int target)

```
int s = 0;
int e = arr.length;
while (s <= e) {
    int mid = (s+e)/2;
    if (target < arr[mid])
        end = mid-1;
    else start = mid+1;
}
return start;
```

while

↓

why Because after loop breaks it becomes

e (ans) s

↓

it is the ceiling of a given target.

② Floor of a number (

(number is

First and last position in a sorted array

[4, 5, 5, 5, 5, 5, 6, 8, 9] → ans = [1, 5]
 0 1 2 3 4 5

1 → First Index
5 → Last Index

```
static int[] sorted(int arr[], int target) {
    int ans[] = {-1, -1};
    ans[0] = BinarySearch(arr, target, true);
    ans[1] = BinarySearch(arr, target, false);
    return ans;
}

static int BinarySearch(int arr[], int target, Boolean isStartIndex) {
    int index = -1;
    int start = 0;
    int end = arr.length - 1;
    while (start <= end) {
        int mid = start + (end - start)/2;
        if (arr[mid] < target) start = mid + 1;
        else if (arr[mid] > target) end = mid - 1;
        else {
            index = mid; //potential answer
            if (isStartIndex) {
                end = mid - 1; //search lefthand side
            }
            else {
                start = mid + 1; //search righthand side for endIndex
            }
        }
    }
    return ans;
}
```

Ⓐ When we have to find length of subarray by using indices of array ⇒ (end-start+1).

Q: Binary Search in Infinite array (means we don't use arr.length).

arr = [2,3,5,6,7,8,10,11,12,15,20,23,30] ; target = 15

```
static int range(int arr[], int target) {

    int start = 0;
    int end = 1;
    while (arr[end] < target) // we have to double the range while arr[end] is lessthan target.
    {
        int newStart = end+1;
        end = end + (end-start+1)*2;
        start = newStart;
    }
    return BinarySearch(arr, target, start, end);

static int BinarySearch(int arr[], int target, int start, int end)
{
    |
    |
}
```

1 2 3 4 ⑤ 6 7 8
s e N.S e+1 target
No target this range. (double size)

Mountain Array (we don't have a target for anything).

Find peak in mountain array.

arr = [1, 2, 3, 4, 6, 4, 3, 2, 1]

Increasing part | Decreasing part

Algo:

1. arr[mid] < arr[mid+1] // we are in increasing part
   start = mid+1. // search right side of mid, but actually mid can't be ans, so start = mid+1.

2. arr[mid] > arr[mid+1] // we are in decreasing part.
   end = mid // search left side of array mid element for max/peak

Code:

```
static int search(int arr[], )
    int start = 0;
    int end = arr.length - 1;
while(start < end){ // not start <= end because when start = end
                    // it is peak element.
    int mid = start + (end - start)/2;
if(arr[mid] < arr[mid+1])
    start = mid+1;
else end = mid;
}
return start; //return end.
```

→ Search in a rotated sorted array!

distinct ⇒ The values are not duplicate.

Approach ①: (Pic in phone)

① Find pivot element in the array:

original array

Ex: [10, 12, 2, 4, 5, 7, 8, 9] → [2, 4, 5, 7, 8, 9, 10, 12]

Because of rotation element

asc

asc

pivot

② Binary Search in first half (0 to pivot)

otherwise, search in second half (pivot+1, end)

Finding pivot:

```
Static int findpivot (int arr[]) {
    int start = 0;
    int end = arr.length - 1;
    while (start <= end) {
        int mid = start + (end - start)/2;
        // 4 cases
        if (arr[mid] > arr[mid+1]) return mid;   // Because if mid = end, then mid+1 exceeds limit.
        if (arr[mid-1] > arr[mid]) return mid-1;   // mid > start
        if (arr[mid] <= arr[start])
            end = mid - 1; // leftpart
        else // start < mid
            start = mid + 1; // rightpart
```

arr = [4, 5, 6, 7, 0, 1, 2]

Case 1: pivot element = target

Case 2: target > start element // search space = (0, P-1)
(0, pivot)
why? coz all nos after pivot are < start

Case 3: target < start element // we know that all elements from (S, pivot) are going to be bigger than target
(P+1, end)
↓
arr.length-1)

Q:- Find how many times will this array is rotated?

arr = [4, 5, 6, 7, 0, 1, 2]
↓
pivot

⇒ So array is rotated "pivot" times.
means (pivot [redacted]) + 1
↓
Index.

Q:-

Binary Search in 2D array:

Q: matrix is sorted in a row-wise & col-wise manner.

	0	1	2	3
0	10	20	30	40
1	15	25	35	45
2	28	29	37	49
3	33	34	38	50

target = 37.

Relevance:
Try to see how to minimise search, when there is a large space.

N+M comparisons = 2N

O(N)

Algo:

Case 1: If current_element == target
→ are found

If current_element < target
→ row++; // If 35 is element, they 35<37, so before 35 mean 25, 15 are less than 37. So we can eliminate that row.

If current_element > target
→ col--;
// If 45 is element, then 45>37. So the column of all that are greater than target - so, we can eliminate columns.

Code:

```
static int[] search(int matrix[][],
                    int target) {

    int r = 0;
    int c = matrix.length - 1 // last column value

    while (r < matrix.length && c >= 0) {
        if (matrix[r][c] == target)
            return new int[] {r, c};
        if (matrix[r][c] < target)
            r++;
        else { c--; }
    }
    return new int[] {-1, -1};
```

Search in sorted matrix:

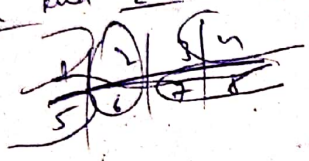
	1	2	3	4
1	1	2	3	4
2	5	6	7	8
3	9	10	11	12
4	13	14	15	16

Algo: Case 1: If current_element = = target
→ return index

Case-2: If current_element < target
// Ignore above rows

Case-3: If current_element > target
// Ignore after rows.

In the end 2 rows are remaining:
1) check whether the mid



Bubble sort Algorithm:

Space Complexity = O(1) // Constant, no extra space required i.e Copying the array

↓

Inplace sorting Algorithms

Time Complexity: Best Case → O(N) // sorted array

Worst Case → O(N^2) // sorted in opposite

↓ No. of Comparisons

As the size of the array is growing, no of Comparisons grows

why constants are ignored? we dont want any relationships we just only want mathematical function.

Stable sorting Algorithm: Order should be the same, when the value is same.

Selection sort:

1st step: (max element)

4 5 1 2 3 — n-1 comparisions

4 3 1 2 5 — n-2

2 3 1 4 5 — n-3

2 1 3 4 5 — n-4

1 2 3 4 5 — 0

Total comparisons

0, 1, ... -(n-1)

$\frac{(n-1)\to(n-1+1)}{2}$ ✓ Time Complexity

$\frac{n(n-1)}{2} = \frac{N^2}{2} - N$

Worst case → $O(N^2)$

Best case → $O(N^2)$

Stable = No.

// It performs well on small arrays

Insertion sort:

Worst case → $O(N^2)$ // desc sorted (5,4,3,2,1)

Best case → $O(N)$ // array is already is sorted

Cyclic sort :- Worst case → O(N)

→ When given numbers from range 1 to N

⇒ use cyclic sort.

Ideology

Algo:

Given:

0	1	2	3	4
3	5	2	1	4

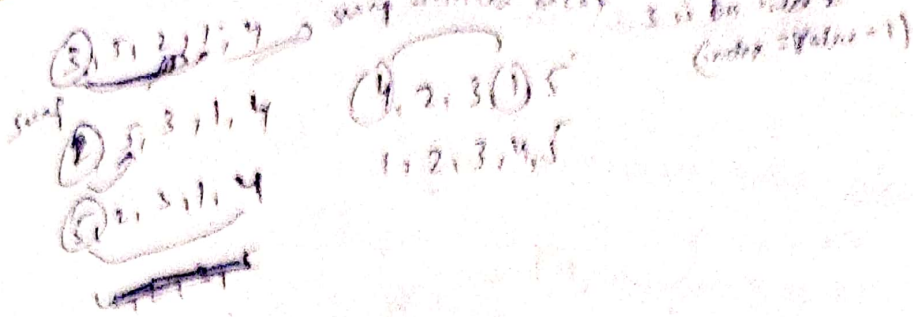
Actually after sorting 1 to N numbers

1, 2, 3, 4, 5

0 1 2 3 4

index = value - 1

Step 1 2 3 4



Code:

```
static void sort(int arr[]) {
    int i=0;
    while (i < arr.length) {
        int correct = index = arr[i] - 1;
        if (arr[correct - index] != arr[i]) {
            int temp = arr[i];
            arr[i] = arr[correct_index];
            arr[correct_index] = arr[i];
        }
        else { i++; }
    }
    print( Arrays.toString(arr));
```

Q:- Missing number.

~~arr = [ ]~~ arr = [4,0,2,1] ; N = 4

[0,1,2,3,4].

Case-1: If N is in the array

arr = [4,0,2,1] => After sorting

[0,1,2,4] [0,1,2,3]

Don't disturb 'N'

3 is the missing element.

Case-2: If N is not in the array.

arr = [1,0,3,2] => [0,1,2,3]

0 1 2 3

So, (N) is the missing element ✓

Bubble, Selection, Insertion, Cyclic sort, Merge & Quick sort.

Code:

```
public int missingNumber(int arr[])        // {0, 1, 2, 3}
    int i = 0
    while (i < arr.length)
    int correct_index = arr[i]
    if (arr[i] < arr.length && arr[i] != arr[correct_index])
      {
        swap( )
      }
    else i++;
    }
    //search for first missing number
    for (int i=0; i< arr.length; i++){
        if (arr[i] != i) return i;
    }
    return arr.length; //Case-2
```

Q:- Find All numbers that are missing in the array

Tips:- If range => 0, N

(a) every element will be index = value (correct_index = arr[i])

If range => 1, N

(b) every element will be index = value - 1 (correct_index = arr[i] - 1).

[4, 3, 2, 7, 8, 2, 3, 1], N = 8

Q: Find the duplicate number.

arr = [1, 3, 4, 2, 2].

1, 3, 4, 2, 2

1, 2, 4, 3, 2

1, 2, 3, 4, 2

check

if (arr[i] != i+1)

{ if whether the element at original index (= value - 1) != element at current index

{ swap }.

else otherwise { return arr[i]; found ans.

}

~~Q: Print the pattern~~

Patterns:

(A) How to approach a problem

Step-1: No. of lines = no. of rows.

= No. of times outer loop will run.

Ex: Total no. of rows = 4, so we can run outer loop for 4 times.

Step-2: Identify for every row number

→ How many col's are there.

→ Type of elements in col.

For Example: 4 rows, 4 columns.

1st row → 1 col

2nd row → 2 col

3rd row → 3 col

4th row → 4 col

Step-3: What do you need to print.

Recursion:

(Binary trees, linked list, Graphs, Dynamic programing, Heaps)

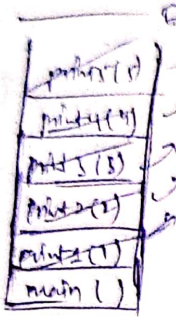
Pre-requisites
1. Functions
2. Memory management

A function that calls itself.

- Base condition
- Body
- Recursive call

→ All function calls are going to stack memory.

How function calls works Internally:




```
print1(int n){
    sop(1)
    print2(n)
}
print2(int n){
    sop(n)
    print3(n)
}
print3(int n){
    sop(n)
}
```

→ While the function is not finished executing, it will remain in stack.

→ When a function finishes executing, it is removed from stack and the flow of program is restored from where the function was called.

(comes out of stack, when function completes execution)

→ main() function is first function is inserted into stack because it is starting point of the program and last function removed from the stack.

→ If function having same signatures & same body, then we have to create different functions. So, instead of creating different functions, call the function recursively.

```
print(int n){
    sop(n)
    print(n+1);
}
```

Base condition

```
if(n==5){
    sop(5)
    return;
}
```

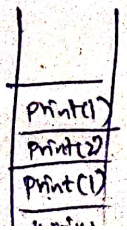
Same

Base Condition:

It is the condition where our recursion will stop making new calls.

→ If the function is called continuously, it will take memory seperately in stack.

Ex:-



Here print() calls 3 times with different arguments. So, it takes 3 spaces in stack

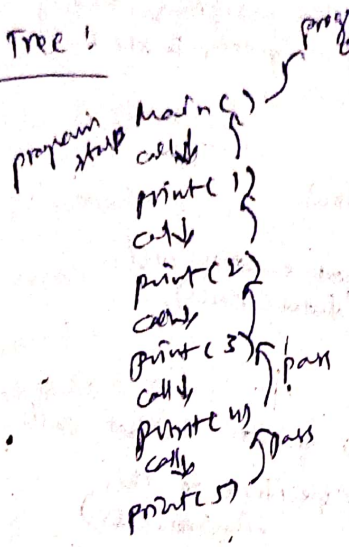
Stackoverflow: If memory of Computer/stack will exceed the limit it gives stackoverflow error.

why recursion?: It helps us in solving complex problems in a simple way.

→ You can convert any recursion functions into Interactions (for loops, while) and vice-versa.

→ Space Complexity: (When we call functions, it is taking space in stack constantly) (If not constant)

Recursion Tree:



How we know if solved using recursion
→ Try to break down into smaller problems.

Q: Find nth Fibonacci numbers:

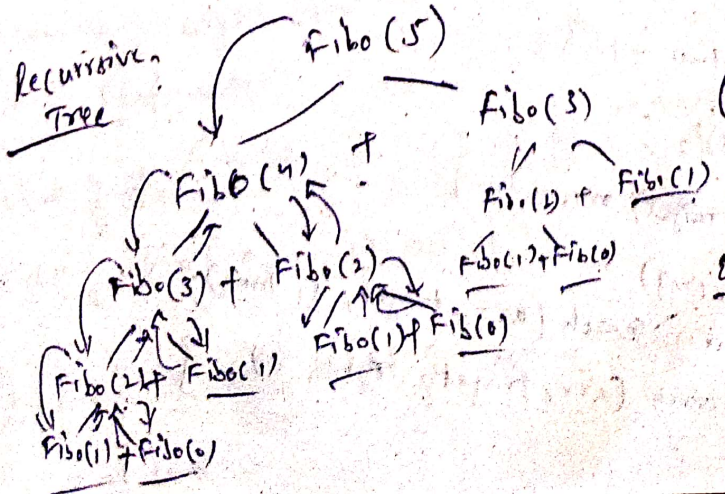
0 or 1st 2nd 3rd 4th 5th 6th 7th

0, 1, 1, 2, 3, 5, 8, 13 ...

$$Fibo(N) = Fibo(N-1) + Fibo(N-2)$$ → Recurrance Relation.

$$Fibo(N-1) + Fibo(N-3)$$

Recursive Tree



(A) Break it down into smaller problems

(B) The Base condition is represented by answers we already have

Eg: F(0) = 0, F(1) = 1

Code:

```
static int fib(int n) {
    // base condition
    if (n < 2) { return n; }   // Base condition is what values we already know.

    return fib(n-1) + fib(n-2);   // Recurrence relation
}
```

Steps to approach a problem:

1. Identify if you can break problem into smaller problems.
2. Write the recurrence relation, if needed
3. Draw the recursive tree. (see the flow of functions like how they are getting in stack)
4. base condition
5. variable points (which working with recursion)

Variables:

1. Arguements, ⇒ will go to next function call. (future calls).
2. Returntype
3. Body of function. ⇒ This will be specific to that call. (specified to this function call)

Tip: Make sure to return the result of a function call, of the return type. (If there are return type, when we call function recursively we have put 'return').

B.S:

```
static int search(int arr[], int target, int s, int e) {
    if (s > e) { return -1; }
    int m = s + (e - s)/2;
    if (arr[m] == target) return m;
    if (target < arr[m])
        return search(arr, target, s, m-1);
    else { return search(arr, target, m+1, e); }
}
```

Binary Tree: (LHS smaller, RHS Bigger) → Binary Search Tree

① Recursion
② OOPS.

Why? :
→ O(log N) {efficiently inserting & deleting elements}.
→ Ordered storage (elements are very ordered in manner)
→ Easy efficient.

Unbalanced Binary Tree:
→ O(N) search time.
→ Time Complexity

How to solve? : Self Balanced B.T (AVL)

→ If we are working with sorted data, B.T is inefficient.

• Where is it used?
① File systems
② Databases.
③ Network Routing.
④ Maths
⑤ Decision trees (ML)
⑥ Compression of files

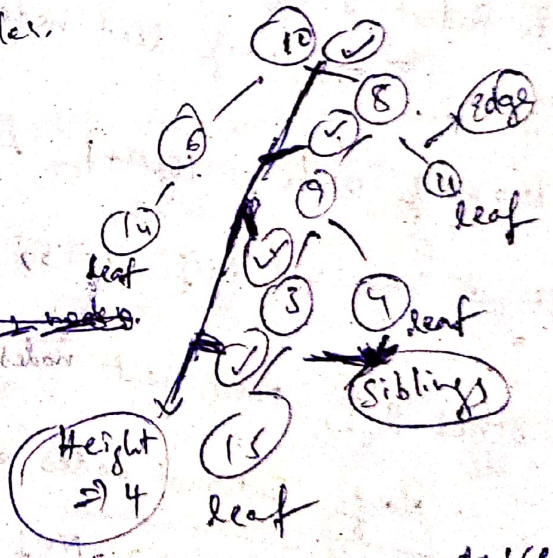
Properties:

① Size = Total no. of nodes/circles.

② child, parent & siblings.

③ Edge.

④ Height → ~~Edges~~ nodes.
↓
→ Max no. of edges from the node and leaf node.



Height of 10 (2, 4, 2, 3) ⇒ 4
Height of 8 = (3, 2, 1) ⇒ 3.

⑤ Level ⇒ Height of root − height of node) Difference.
↓
root level = 0.

⑥ Ancestor & descendants

⑦ Degree (No. of childrens)

Types of Binary Tree:

① Complete Binary Tree (All the levels full except last level full from left to right)

② Full B.T / Strict B.T (In this tree, every node contains either 0 or 2 children
Eg: Compression, segment trees)

③ Perfect B.T (All levels are full)

④ Height Balanced B.T (Avg height $O(\lg N)$
AVL

⑤ Skewed B.T (every node has only one child)
Height = $O(N)$, like LinkedList

⑥ Ordered B.T (Every node has some property)
Eg: BST i.e left smaller, right bigger.

Properties that help for solving questions:

→ In Perfect B.T, height = h (height starts from '0').
Total nodes = $2^{(h+1)} - 1$.

→ In Perfect B.T, height = h.
Total no. of leaf nodes = 2^h.

→ N = no. of leafs, $\log N + 1$ levels atleast.

→ If we have N' no. of nodes, min possible height = $\log(N+1)$ levels.

→ In strict B.T, Total leaf nodes = n.

No. of internal nodes = n − 1.

No. of leaf nodes = No. of Internal nodes + 1

→ No. of leaf nodes = 1 + No. of Internal nodes with 2 children.
(not including root node)

Implementation:

① Linked representation.

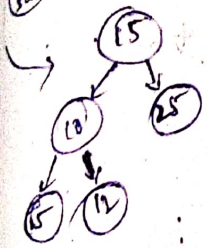
② Sequential (using array) ✗ (Not efficient)

Code: (Binary Tree)

Binary search Tree: ($\log(N)$) (L.H.S elements is smaller than root node / R.H.S " " larger than root node.)

→ Height difference ≤ 1 (any two nodes on the same level) ⇒ Balance Tree

(Not valid)



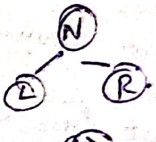
→ BFS & DFS:

Breadth → . Depth ↓

Traversal methods:

① Pre-order:

N → L → R



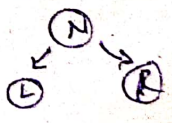
10, 20, 15, 13, 12

N — left — R

→ Used for evaluating math expression or making a copy.

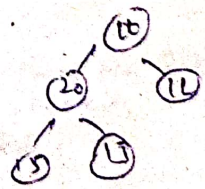
② In-order:

L → N → R.



15, 20, 13, 10, 12

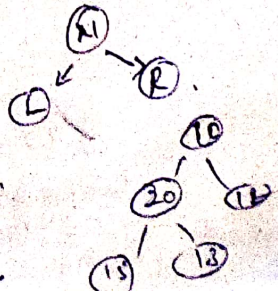
L — N — R



→ InBST, visit nodes in sorted manner.

③ Post order:

L → R → N.

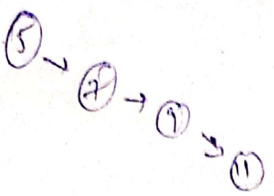


15, 13, 20, 12, 10.

L — R — N

④ delete Binary Tree.

Instead of O(log(N)) it is taking O(N) : It is the problem in BST.



→ For every node, the height(left tree) - height(right tree) $\leq -1, 0, 1$

→ It is known as Balanced Tree

→ To solve the problem of unbalanced Binary Tree, Self Balancing Binary Tree comes into play.

Ex: AVL Tree. (If tree is unbalanced, AVL makes Balanced automatically by adjusting itself).

AVL Tree:

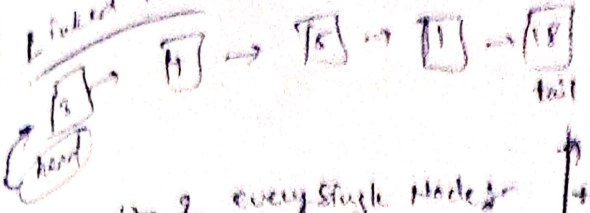
Algorithm:

(1) Insert normally a node (N)

(2) Start from node (N) & find the node that makes the tree unbalanced, bottom up.

(3) Use one of (4) rules.

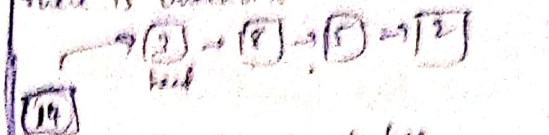
Linked List :



Representation { every single Node }

```
class Node {
    int val;
    Node next;
}
```

How to insert element to first, when there is LinkedList



14

By assigning this element has node.next = head
head = node (update head pointer to 14)

-> If there is no elements

Code :

```
public class LL {

    private class Node {
        private int value;
        private Node next;

        public Node (int value){
            this.value = value;
        }
        public Node (int value, Node next){
            this.value = value;
            this.next = next;
        }
    }

    private Node head;
    private Node tail;
    private int size;

    public LL( ){
        this.size = 0;
    }

    public void insertFirst (int val){
        Node node = new Node(val);
        node.next = head;
        head = node;
        if ( tail == null ){
            tail = head;
        }
        size ++;
    }

    public void display () {
        Node temp = head;
        while(temp != null) {
            System.out.print(temp.value + " -> ");
            temp = temp.next;
        }
    }
```

Don't change head pointer unless (or) until we changing the structure of LinkedList like add, delete etc. For display(), don't change head pointer.

(or) if(head == null) { head = node; tail = node; }

Insertion before head element.

If there are LinkedList elements

updating head element as node

If list is empty, we want add first element

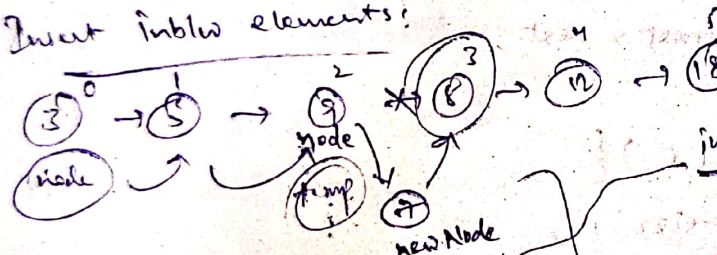
If represents list is empty

```
Class Main {
    psvm( ) {
        LL list = new LL( );
        list.insertFirst(3);
```

```
public void insertLast(int val) {
    Node node = new Node(val);
    tail.next = node.   // changing structure (place & insertion of this element)
    tail = node  // updating tail pointer to last node
    size++;
}
```

```
if(tail == null){
    insertFirst(val);
    return;  // Don't call below code
}
```

Insert in blw elements:



index = 3. (we have to locate 2nd index pointer)

```
public void insert(int val, int index) {
    if (index == 0) {
        insertFirst(val);
        return;
    }
    if (index == size) {
        insertLast(val);
        return;
    }
    Node temp = head;
    for (int i = 0; i < index; i++) {
        temp = temp.next;
    }
    // when creation of this node temp is (9). so temp.next is next node
    Node node = new Node(val, temp.next);
    temp.next = node;
    size++;
}
```

This will create connection blw (7) & (8)

delete first: head = head.next;

```
public int deleteFirst() {
    int val = head.value;
    head = head.next;
    if (head == null) {   // Because list contains only one element in list
        tail = null;
    }
    size--;
    return val;
}
```

~~Delete last~~:
get Node at that Index:

```
public Node get(int index) {
    Node node = head;
    for (int i=0; i<index; i++) {
        node = node.next;
    }
    return node;
}
```

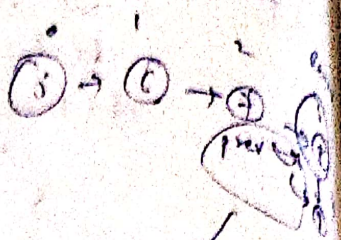
deleteLast element:

```
public int deleteLast() {
    if (size <= 1) {
        return deleteFirst;
    }
    Node secondLast = get(size-2);   // get second last element before
    int val = tail.value;            // get last element
    tail = secondLast;               // assign last element as second last element
    tail.next = null;                // and assign pointer as null.
    return val;                      // return original last element
}
```

= and Creation connection b/w ~~A~~ B.

delete a particular index:

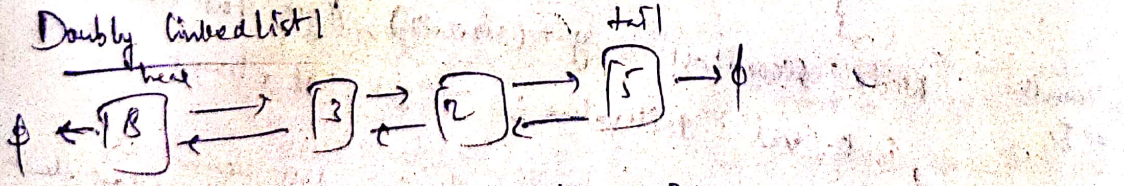
```
public int delete(int index){
    if (index == 0) {
        return deleteFirst();
    }
    if (index == size - 1){
        return deleteLast();
    }
    Node prev = get(index - 1);
    int val = prev.next.value;

    prev.next = prev.next.next;
    return val;
}
```



get Node:

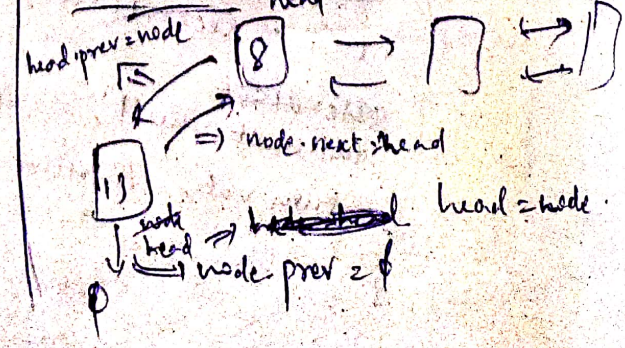
```
public Node find(int value){
    Node node = head;
    while(node != null){ //traversing through last elem
        if(node.value == value){
            return node; } //when it is found.
        node = node.next;
    }
    return null; //nothing is found
}
```

Doubly LinkedList




```
Class Node{
    int value;
    Node next;
    Node prev;
}
```

insert first head



Code:

```
public class DLL {
    Node head;
    private class Node {
        int val;
        Node next;
        Node prev;
        public Node(int val){
            this.val = val;
        }
        public Node(int val, Node next, Node prev){
            this.val = val;
            this.next = next;
            this.prev = prev;
        }
    }
    public void insertFirst(int val){
        Node node = new Node(val);
        node.next = head;
        node.prev = null;
        if(head != null){
            head.prev = node;
        }
        head = node;
    }
    public void display(){
        Node temp = head;
        Node last = null;
        while(temp != null){
            sop(temp.val + "->");
            last = temp;
            temp = temp.next;
        }
        sop("End");
        sop("Print in reverse order");
        while(last != null){
            sop(last.val + "->");
            last = last.prev;
        }
    }
    public void insertLast(int val){
        Node node = new Node(val);
        tail.next = node;
        node.prev = tail;
        node.next = null;
        tail = node;
    }
```

if it is not first element and there are some elements already

// if this is first node we are inserting it gives nullpointer exception

here by end of loop it gives last value and we uses it in below while loop

if we are using tail element

InsertLast() code:

```
public void InsertLast(int val){
    Node node = new Node(val);
    Node temp = head;
    while(temp.next != null){ // Because we don't have tail variable, so to find last element we have to traversal first
        temp = temp.next;
    }
    temp.next = node;
    node.prev = temp;
}
```

```
if(head == null){
    node.prev = null
    head = node;
    return;
}
node.next = null;
```

```
public void find(int value){
    Node temp = head;
    while(temp != null){
        if(temp.value == value){
            return temp;
        }
        temp = temp.next;
    }
    return null;
}
```

Inserting element:

temp.next = node

node.next = temp.next

node.prev = temp

node.next.prev = node

Check

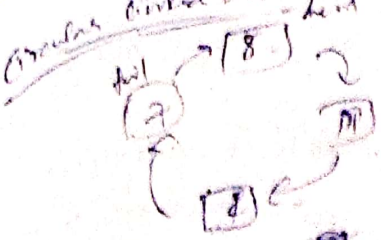
- If anything going to give null pointer exception

this may be null (Because it may be the last element)

Code:

```
public void insert(int after, int val){
    Node temp = find(after);
    if(temp == null){
        sop("doesn't exist")
        return;
    }
    Node node = new Node(val);
    node.next = temp.next;
    temp.next = node;
    node.prev = temp;
    if(node.next != null){
        node.next.prev = node;
    }
}
```

Circular Linked List



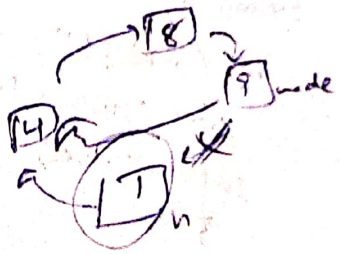

```
class node {
    int val;
    node next;
}
```

```
public void insert (int val){
    Node node = new Node(val);
    if (head == null){
        head = node;
        tail = node;
        return;
    }
    tail.next = node;
    node.next = head;
    tail = node;
```

(Here we will use do-while loop because it directly contains the portion)

```
public void display ( ){
    node temp = head;
    if (head != null) // checking whether the list is empty or not
    {
    do{
        Sop("temp.value + ");
        temp = temp.next;
    }
    while (temp != head);
    }
```

```
public void delete (int val){
    Node temp = head;
    if (temp == null){ return; }
    if (temp.val == val){ // if we have to delete first element
        head = head.next;
        tail.next = head;
        return;
    }
```




```
    do {                    temp
        Node n = node.next;
        if (n.value == val){
            temp.next = n.next;
            break;
        }
        temp = temp.next;
    } while (temp != head)
```

Stack: (FILO)

(Binary tree + BFS, DFS) Recursion to Iterative

Removal → pop()
add → push()

Implementation of stack:

```
public class CustomStack {
    protected int[] data;
    private static final int
        DEFAULT_SIZE = 10;
    int ptr = -1;
    public CustomStack() {
        this(DEFAULT_SIZE);
    }
    public CustomStack(int size){
        this.data = new int[size];
    }
public boolean push(int item){
    if (isFull()){
        sop("stack is full");
        return false;
    }
    ptr++;
    data[ptr] = item;
    return true;
}
Public
private isFull() {
    return ptr == data.length - 1
}
private boolean isEmpty() {
    return ptr == -1;
}
```

Queue:

peek() – It gets head, but doesn't remove

remove() – throws an exception when dynamic is empty

poll() – doesn't throw exception

add() – throws an exception when capacity limit crossed

offer() – doesn't throw an exception

Throws Exception

```
public int pop(){
    if(isEmpty()){
        throw new Exception("Cannot
        pop an item from an Empty stack");
    }
    int removed = data[ptr];
    ptr--;
    returned removed;
}
```

or else return data[ptr--];

throw Empty

```
public int peek(){
    return data[ptr];
    if(isEmpty()){
    throw new Exception("Stack is
                         Empty")
```

In dynamic stack array:

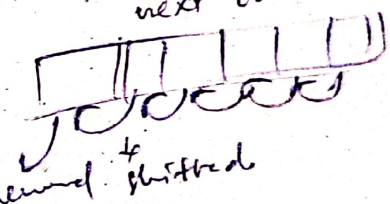
```
public boolean push(int item){
    if(isFull()){
        sop("stack is full")
    int temp[] = new int[data.length*2];
    for(int i=0; i<data.length; i++){
        temp[i] = data[i]
    }
    data = temp;}
    return super.push(item);
```

Fixed Implementation of Queue:

adding → O(1)
removal → O(N)
Because of shifting the items.

```
class Queue {
private int[] data;
private static final int DEFAULT_SIZE = 10;
int end = -1;
public CustomQueue() {
        this(DEFAULT_SIZE);
    }
public CustomQueue(int size) {
        this.data = new int[size];
    }
public int remove() throws Exception {
    if(isEmpty()){
        throw new Exception("Queue is empty.");
    }
    int removed = data[0]; //removes first item
    //shift the elements to left
    for(int i=0; i<end; i++){
        data[i-1] = data[i]; //the previous one is equal to next one.
    }
    end--;
    return removed;
```

Since it is a queue.



→ In order to remove an element in queue it taking O(N) time. So, to overcome this we use Circular queue

insertion → O(1)
removal → O(1)