Date: / /

JAVA

Keywords are case sensitive public, Public, PUBLIC are all different

keywords

```
public class Hello{
    ...
    //part /body
}
```

} code block

public - optional access modifier.

Method: It is a collection of statements (one or more) that performs an operation.

Main method - : entry point of any java code. Java looks for it when running any program

```
public class Hello{

    public static void main(String args[]){
        S.O.P ("Hello"); --> Statement
    }

}
```

} code block

Statement: a complete command to be executed and can include one or more expressions

Variables: a way to store information in our PC.

- can be changed, variable content
- stored in RAM

To define a variable we need to specify data type, then give it a name and optionally add an expression to intialize it with a value

```
int myNumber = 5;
```

→ Declaration statement.

↓

data type, name, optional set value

Expression: it is a construct that evaluates to a single value

Primitive Data Types - Total 8

int byte short float

boolean char long double.

int → min & max value

Integer.MIN_VALUE , Integer.MAX_VALUE

wrapper class.

Java uses the concept of wrapper class for all 8 primitive types.

int - min -2147483648

max 2147483647

★ Note: If you try to put a larger than max value, or smaller than min value, then you will get an overflow in case of max &

Date: / /

underflow in case of min.
The computer skips back to min or max number.
eg:

int i = 2147483647 + 1 ⟹ o/p -2147483648

int i = -2147483648 - 1 ⟹ o/p 2147483647

* int maxint = 2_147_483_647.
(can be written this way in Java 7 or higher to make it readable)

Byte - Byte.MIN-VALUE = -128 Max = +127.
small in size, might help in performance but today PC's have a lot of space so no need to worry about it.

Short -32768 to 32767

	width	
Byte:	8 bit	$\Rightarrow 2^7$
Short	16 bits	$\Rightarrow 2^{15}$
int	32 bits	$\Rightarrow 2^{31}$
long	64 bits.	$\Rightarrow 2^{63}$.

long:

long mylongValue = 100L;
L represents it is a long value (uppercase)

By default, java takes number as int. So this L is needed in case of long number.

Casting

⊗ byte mybyte = (myMinByteValue/2);

int default in java is int

=> (byte)(myMinByteValue/2)

↳ casting.

float - single precision no. → 32 bits => width = 32

double - double precision no => 64 bits => width = 64

Precision = format & amount of space occupied by the type.

* double is default type for real no's
* int is —"— whole no's.

float myfloat = 5f; 5.25 X error. float(5.25)

double mydouble = 5.25d;

char - can store a single character

- occupied 2 bytes = 16 bits. (bcoz it allows to store unicode chars also)

Unicode - international coding standard

char myunicode = '\u004A';

↳ unicode 'D' → unicode table mein se 004A liya

Date: / /

String → it is a class, easy to operate
→ immutable (cannot be deleted once created)
→ string + any data type ⇒ string.

Operator - Special symbols that perform specific operations.

Operand: object manipulated by operator.

Expression: is formed by combining variables, literals, method return values and operators.

- logical and → &&
- Logical OR ⇒ ||
- NOT !

- &, | → bitwise operator
- Ternary

```
boolean isCar = false;
boolean wasCar = isCar ? true : false.
```

Statement.

```
int [myint = 50];
    expression
```

Method Overloading :- a feature that allows us to have more than one method with same name, so long as we use different parameters.

eg: println.

→ improves readability, re-usability

→ easy to remember one method name instead of multiple

Changing return type does not include in method overloading

```
int sum(int a, int b)
float sum(int a, int b)
```
} → will give error.

→ change no. of parameters

→ datatype of parameters

→ changing sequence (int, double) (double, int)

switch

Can be used with byte, short, int or char, strings.

```
switch(var)
{
case 1: break;
case 2: break;

default: break;
}
```

```
switch ("January")
{
case january: break
default: brk;
}
```
//will go to default

case should be same

Java has 50 keywords.

Date: / /

a.toLowerCase().

a.toUpperCase();

* for

* while (condition) {

}

* do

{ → execute atleast once

} while(condition);

* int cnt = 1 ——→ initialize

while (cnt<=5){ → condition

cnt++ → increment.

}

* for(int i=1; i<=5; i++)

{

}

* do}

cnt++ ——→ inc.

}while(condition);

cond

Some Keywords

* continue

* break

Date: / /

* convert String into int

String str = "2018"

~~str~~.int n = Integer.parseInt(str);

//op 2018

String Str = "2018a"

int n = Integer.parseInt(str) // error since 2018a is not an int

→ Double.parseDouble

→ float.parseFloat

Scanner

Scanner scanner = new Scanner(System.in);

System.in → allows to type input in console which gets returned back to the program

new → used to create instance of class Scanner / create object

name = scanner.nextLine();

scanner.close(); //to release memory scanner was using.

scanner.nextInt();

IMP * scanner.nextLine(); → Handles enter key issue

- when you press enter after taking int it takes it as break, so code there hence use nextLine();

Date: / /

init yearBirth

SOP("Enter year");

** boolean hasNextInt = sc.hasNextInt(); //returns if no is int.

seyearBirth = sc.nextInt();

Classes

this.model = model → local var. of method

var. of the class.

Constructors - can be overloaded

- always public.

- you can call one constructor in other.

- this(); should always be first line in the constructor are calling other constructor inside it.

eg.

```
public class Car {

    public Car() {
        SOP("empty");
    }

    public Car(int model, String name)
    {
        this();
        this.model = model;
        this.name = name;
        SOP("not empty");
    }
```

→ Car(1, "Volvo");

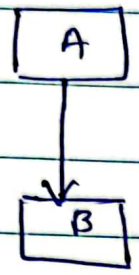
O/P - empty

not empty

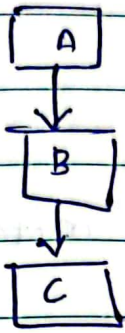
Inheritance

→ extends

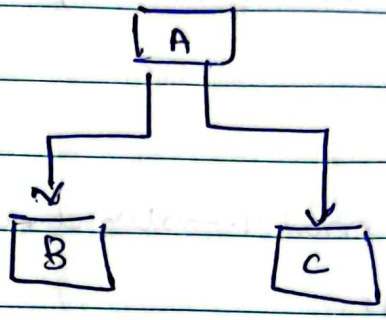
→ super();



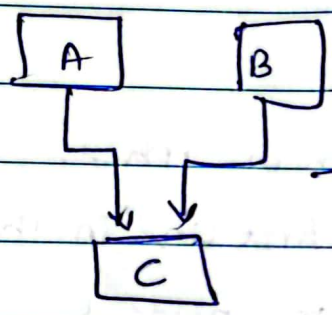
Single



multilevel

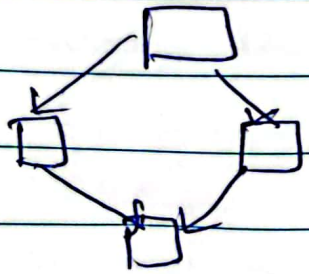


Hierarchial



→ Java does not support // can be achieved only through interfaces

multiple



→ does not support

Reference vs Object vs Instance vs class

Consider building a house

Class :- blueprint for a house, we can build as many houses as we like with that blueprint/plan

Instance: Each house you built, is an object / instance.

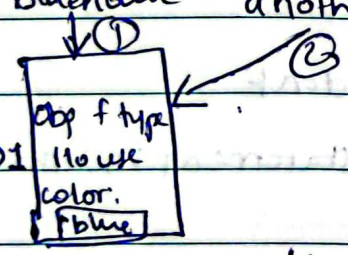
Reference: Each house has an address, ie- a reference

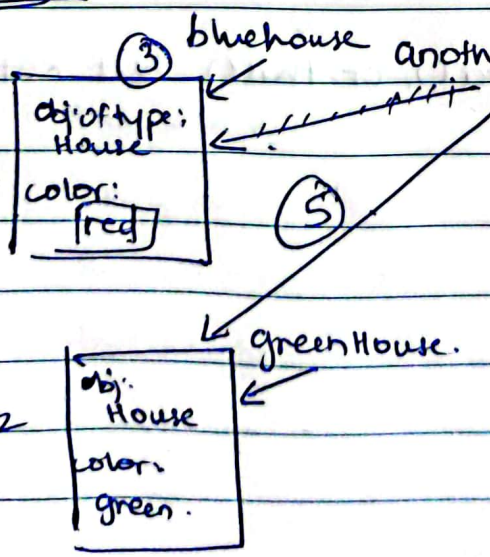
You can copy the reference as many times as u like but there is still one house

We can pass references as parameters to constructors & methods

eg: class House{
pvt String color;
public House (Str color)
{ this.color = color;}

public class Main {
① House blueHouse = new House("blue"
② House anotherHouse = blueHouse;
④ SOP (blueHouse.getColor()) //blue
⑤ SOP (anotherHouse.getColor()) //blue.
③ anotherHouse.setColor("red");
SOP (blueHouse. . .) //red
SOP (- - -) //red.





this vs super

super → is used to access/call parent class members
→ commonly used with method overriding

this → used to call current class members. This is required when we have a parameter with same name as an instance variable.

* Note: we can use both anywhere except in static areas (the static block or static methods) Any attempt at this will give compile time errors

this() → only used in constructor to call another overloaded constructor.
→ this(); should be first statement in a constructor.

super() → java by default calls it if we don't
→ call to super() must be first statement in the constructor

* A constructor can call super(); or this() but not both

Date: / /

Constructor chaining eg:

```
class Rectangle {
    prt int x;
    —11— y;
    —11— width;
    —11— height;

    public Rectangle{
        this(0,0);
    }

    public Rectangle(int width, int height)
    {
        this(0,0, width, height);
    }

    public Rectangle(int x, int y, int width, int height)
    {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }

}
```

Constructor Chaining

The last constructor has responsibility to initialize the variables.

actual initialization is happening here.

Overloading vs Overriding

Overloading - same name, diff parameters

- return type may or may not be different

- Compile time Polymorphism

- we can overload static & instance methods

- usually overloading happens inside a single class, but a method can be treated overloaded in subclass of that class

Overriding

- Runtime Polymorphism

- defining a method in child class that already exists in parent class with same signature (name, args)

- @Override - annotations.

- You cant override static methods

- It cant have a lower access modifier → parent: private, child: protected ✗ error

- only inherited methods can be overridden.

- constructors & private methods can't be overridden.

- final methods can't be overridden

- is-a relnship, there only u can overwrite

* Co-variant return type *

Date: / /

Static methods

- are declared using static modifier
- can't access instance methods & instance variables directly.
- usually used for operations that do not require any data from an instance of class (from 'this').
- we can't use 'this' keyword in static.
- does not require object to access it
  classname.methodname().

Instance methods

- belong to an instance of a class.
- to use it you have to instantiate the class first by using new keyword
- can access instance methods & var. directly
- can access static —11— ————————

Static var	Instance var
- declared by using static keyword	- belong to instance of a class
- aka static ~~instan~~ member variables	- represent state of specific instance.
- every instance of class shares same static variable	
- one change is reflected in all instances	

Inheritance - Is-A Vehicle → Car. car is a vehicle.

Composition has-A Computer → keyboard PC has a keyboard

eg.- Main.java - In main create bedroom

Bedroom.java - Bedroom has wall, lamp, etc.

Wall.java

Lamp.java

- Math.round() → returns long
- leapyear-

```
if (year%4 ==0)
  { if (year%100 ==0)
      {
        if (year%400 ==0)
          return true;
      }else return false;
  return true;
  }
else return false;
```

- Str.toLowerCase(), Str.toUpperCase()
- Math.sqrt(n) → long.
- Diagonal star.
- .equals("y")
- Arrays.toString(arr)
- Str.equals(str2)
- Str1.compareTo(str2)

=0	Equal
>0	S2>S1
<0	S1>S2

- cities.isEmpty() → check if null string.

Date : / /

default - int - 0 String - null.
boolean - false

Arrays :-

```
int[] array = new int[10];
 array[5] = 50;
double[] arr = new double[10];
```

Declaration

① dataType[] arr; ② dataType []arr ③ dataType arr[];

Instantiation.

```
arrayRefVar = new dataType[size];
```

Initialize

```
arrayRefVar[n] = 10;
```

a.length → length of array.

```
int[] a = {33, 34, 35};
```

for-each loop

```
for (data_type variable : array)
    {
      ---
    }

for(int i:arr)
 {
   sop(i);
 }.
```

Date : / /

2-D array:

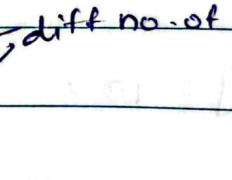
Declaration

- dataType[][] arrRefVar;
- dataType [][]arrRefVar;
- dataType arrRefVar[][];
- dataType[] []arrRefVar[];

```
int arr[][] = {{1,2,3},{4,5,6},{7,8,9}};
```

Jagged array – array of arrays with diff. no. of columns.

```
int arr[][] = new int[3][];
arr[0] = new int[3];
arr[1] = new int[4];
arr[2] = new int[2];
```



Section 8: Arrays, inbuilt Lists, Autoboxing & Unboxing

Initialization:

```
int[] myArray = new int[10];

int[] myArray = {1,2,3,4,5,6,7,8,9};
```

↓ valid only when arr is first declared

→ .length → returns length of the array.

myarr.length

Declaration	Instantiation
datatype[] arr;	arrayrefvar = new datatype[size];
datatype []arr;	
datatype arr[];	

for-each

```
for ( data-type variable : array)
  {
    ...
  }
```

```
for (int i : arr)
  {
    SOP(arr[i]);
  }
```

2D array.

Declaration

```
datatype[][] arrefvar;
datatype [][] arrefvar;
datatype arrRefvar[][];
datatype []arrefvar[];
```

```
int arr[][] = {{1,2,3}, {4,5,6}};
```

Jagged array: array of arrays with diff no. of columns

```
int arr[][] = new int[3][];
arr[0] = new int[3];
arr[1] = new int[4];
arr[2] = new int[2];
```

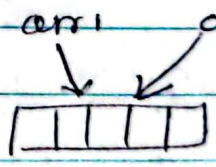
→ diff no. of columns.

Reference Type vs Value Types.

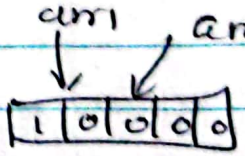
```
int n1 = 10
int n2 = n1
SOP(n1)    //10
SOP(n2)    //10
n2++;
SOP(n1)    //10
SOP(n2)    //11
```

Both variables work independently.
Reference types work differently.

```
int[] arr1 = new int[5];
int[] arr2 = arr1;
SOP(arr1);    // 0 0 0 0 0
SOP(arr2)     // 0 0 0 0 0
```




```
arr2[0] = 1;
SOP(arr1);    // 1 0 0 0 0
SOP(arr2);    // 1 0 0 0 0
```




```
modifyArray(int array)
{
array[0] = 2;
array = new int[] {1, 2, 3, 4, 5};
```

```
modifyarray(arr2);

sop(arr1);          // 2,0,0,0,0
sop(arr2);          // 2,0,0,0,0

arr2 = new int[] {4,5,6,7,8};

modifyarray(arr1);
sop(arr1);          // 2 0 0 0 0
sop(arr2);          // 4 5 6 7 8
```

Strings

- backed internally by char array.
- since arrays are immutable, so are strings.
- whenever a change in string is made, an entirely new string is created.

String str = "Geeks"; — Literal, stores in pool or string constant pool

or

String str = new String("Geeks"); - dynamically alloted. Stores in heap, need to internalize to store in JVM.

1) String class
2) StringBuffer
3) StringBuilder.

1) String class

- sequence of characters.
- 2 ways to initialize as above.

methods:

- int length()
- char charAt(int i)
- String substring(int i) → substr from index i to end
- String substring(int i, int j) → i to j-1.
- String concat(String str) →
- int indexOf(String s) → index of first occurence of s.
- int indexOf(String s, int i) → index counts from i
- int lastIndexOf(String s). - last occurence.
- boolean equals(Object o) o1.equals(o2)
- boolean equalsIgnoreCase(String s)
- int compareTo(String s2) s1 < s2 <0 s1 comes before s2
  s1 = s2 0
- compareToIgnoreCase() s1 > s2 >0 s1 comes after s2.

String toLowerCase()
String toUpperCase()
String trim()
String replace(char old, char new)

2) String Buffer
- mutable. (growable & writable char seq).
- extends from object class.
- safe for multiple threads, can be synchronized when needed

StringBuffer s = new StringBuffer(); // leaves room for 16 chars
StringBuffer s = new StringBuffer(20) // 20 chars
new StringBuffer("Geeks");

methods -
- append()
- length()
- capacity()
- charAt()
- delete()
deleteCharAt()
ensureCapacity().
insert() → at specified index.
reverse().
replace()-
- all methods of string class.
- toString().

3) StringBuilder:

- mutable.
- does not guarantee synchronization
- faster than StringBuffer.
- not safe for multiple threads.

```
StringBuilder str = new StringBuilder();
                                    (20); //capacity = 20
```

Methods - all as Buffer one.

List and Arraylist

(More on lists later)

Arraylist → a resizable list

name of var

groceryList

```
private ArrayList<String> = new ArrayList<String>();
```

- `ArrayList<String>` → datatype / object etc.
- `new ArrayList<String>()` → We have () bcoz ArrayList is a class. So this is a call to empty constructor.

→ `groceryList.add(item);` // to add an element to the list

→ `groceryList.size()` // how many elements in that list

→ `groceryList.get(i)` // get element at ith index of the list
// starts from 0.

→ `groceryList.set(position, newItem);`
- position → index
- newItem → new value to be stored at the index.

→ `groceryList.remove(position)`
- position → index.

→ `groceryList.contains(searchItem);`
// returns T or F if the list has the item or not

→ `groceryList.indexOf(searchItem)`
// returns index of item to be searched or -1 if not found

//copy one arraylist to other

① Arraylist<String> newArray = new Arraylist<String>();
newArray.addAll(grocerylist.getGrocerylist());
getter.

② Arraylist<String> newArray = new Arraylist<String>(
grocerylist.getGrocerylist());

③ //convert to array
String[] myArray = new String[grocerylist.getgroceylist().size()]
size of the actual list
myArray = grocerylist.getGrocerylist().toArray(myArray);

Autoboxing and Unboxing.

We cannot do something like

```
ArrayList<int> intArrayList = new ArrayList<int>();
```

X Error

we need a class ref here

int is a primitive type. It is not a class.

What to do?.

→ Create a class

```
class IntClass {
    private int myValue;
    //constructor, getter, setter
}
```

```
ArrayList<IntClass> intArrayList = new ArrayList<IntClass>();
intArrayList.add(new IntClass(5));
```

→ Integer class

```
ArrayList<Integer> newArrayList = new ArrayList<Integer>();
for (i=0; i<=10; i++)
  {
    newArrayList.add(Integer.valueOf(i));  → Autoboxing
  }
```

```
SOP(newArrayList.get(i).intValue());  → Unboxing.
```

[doubleValue()]

→ But u dont need to do this, java provides easy way

```
Integer myintVal=56;
int myInt = myintVal;
```

//use double if its a double etc

Linked List:

Index	Address	value
0	100	24
1	104	18
2	108	91
3	112	57
4	116	453

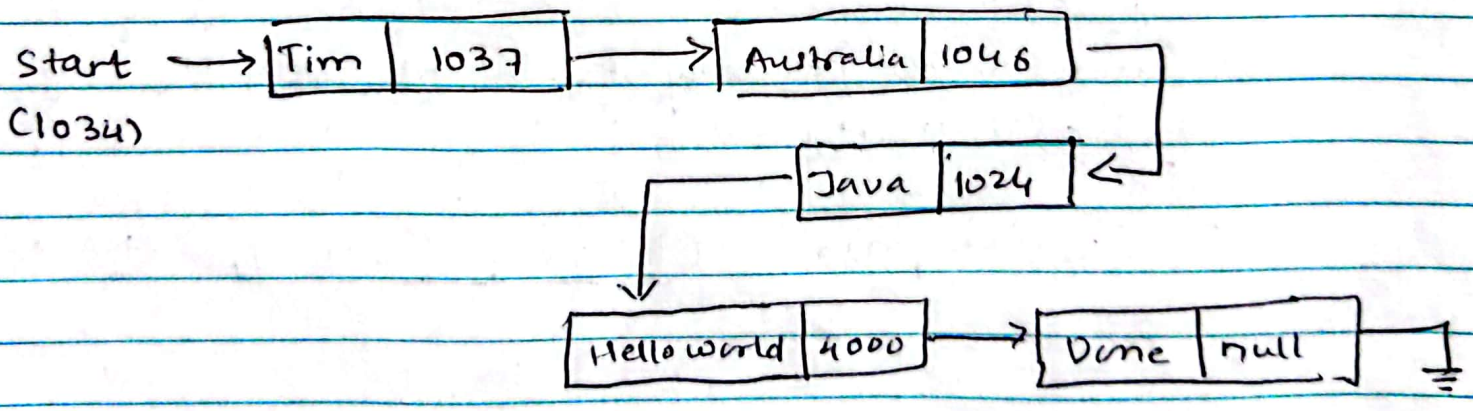
base index + (index - 1) * 4

for strings.

index	Addr.	Str. Addr
0	100	1034
1	108	1037
2	116	1046
3	124	1024
4	132	4000

1024 — Hello world
1034 — Tim
1037 — Australia
1046 — Java
4000 — Done.

Linked list → stores link to next item in the list.
→ also stores actual data



* If u want to add new rec, u just need to change two pointers

```
import java.util.LinkedList;
LinkedList<String> placestoVisit = new LinkedList<String>();
  placestoVisit.add("A");
    —"———— ("B");
        :
        :

// printing
Iterator<String> i = placestoVisit.iterator();
while(i.hasNext())
   {
     sop ( i.next());
   }.

placestoVisit.add(1, "C");        //index=1, value=C
 print.
 // A
    C
    B.

//Remove.
placestoVisit.remove(4);   // index.
```

* List iterator

```
ListIterator<String> itr = linkedList.listIterator();
 while(itr.hasNext())
  {
    int compare = itr.next().compareTo(newCity);
    if(compare ==0)
      { //same city exists
      }
    else if (compare > 0)        //to be added before
      {
        itr.previous();
        itr.add(newCity)
      }
```

```
else if (compare < 0)
    {
        // just move to next
    }

    itr.add(newcity);        // reached end of list
}
```

* The iterator is not pointing to first entry.
you need to use .next() to go to first entry.
.next() → returns next element in the list & advances cursor pos

* Java implements LinkedList as double linked list

Problem with next, prev.

A → B → C → D → E → F

case1 → forward
case2 → backward

A → B → C → (C) → ? Problem.
(1) (1) (2)

↑ pointing to c next if going forward.
So if I do previous, it points to B
But i want it to point to A.

So i need to do prev, if going forward
& forward, if going backwards.

boolean goingforward = true;

Case 1: go to next city

Aage ja raha hai to going-forward should be set to true. & extra .next

```
if (!goingforward) {
    if (listIterator.hasNext())
        {
            listIterator.next();
        }
    goingforward = true;
}
if (listIterator.hasNext())
{
sop ( now visiting + listIterator.next());
}
else {
 sop ( At the end of list).
}
```

Case 2: go to prev city.

Peeche - goingforward → false
extra .previous()

```
if (goingforward)
 {
   if (listIterator.hasPrevious())
     {
       listIterator.previous();
     }
   goingforward = false;
}
if (listIterator.hasPrevious())
   {
sop. (now visiting + listIterator.previous())
   }
else
   sop ( At the start of list)
```

Set: unordered collection with no duplicates.
- it is an interface.
- insertion order is not retained.

```
Set<obj> set = new Hashset<obj>();
```

Methods -

- add()
- addAll
- clear() → remove all
- contains(element)
- containsAll(collection)
- hashcode()
- isEmpty()
- iterator()
- remove(element)
- size
- toArray()

Operations
- union
- intersection

```
Set a, Set b
a.addAll(b).
```

- Intersection.

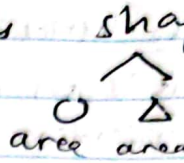
```
b.retainAll(a)
OR a.retainAll(b)
```

Operations

```
Set a, Set b
Set union = // contains a
union.addAll(b);

Set<Integers> intersection = new Hashset<>(a);
intersection.retainAll(b);

difference = new Hashset<>(a);
difference.removeAll(b).
```

Abstraction - display only essential details
- interface & abstract classes
- implementation hiding

(login, you dont know how it works)



Encapsulation - wrapping up data under a single unit.
- access modifiers (private var, public methods; getter, setter)
- data hiding
- restricting access to methods & attributes, prevent data being modified by accident

(Hashtable)

Inheritance: - reusability

Polymorphism: 1) Overload - 3 types.
2) Override.

Interface
- 100% abstraction.
only abstract methods
only static and final var.
method - public abstract
var → public static final.
implements.
can extend other interface only
- public members

Abstract class
abstract/non abstract met
can have final/non final
static/non static.
- extends.
can extend interfaces + other class:
- pub/pri/protected

cannot be used to create obj; must be inherited.

super - access parent class var/method

final - applicable to var, method, class.
var - constant var
method - prevents overriding
class - prevent inheritance (String class)

Static: Share var/method

static block - to init static var, runs only once when class is first loaded

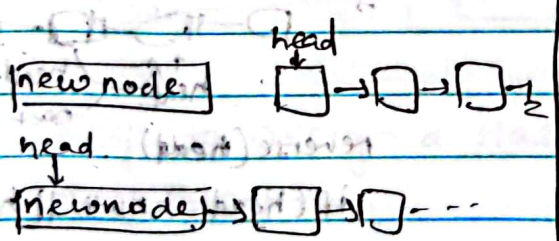
static methods - main(), used without creating object to acc

LinkedList - Insert / Delete.

```
class Node{
   int data;
   Node next;
   Node(int d)
   {
      data = d;
      next = null;
   }
}.

print(head)
{
  while(head != null)
      System.out(head.data)
      head = head.next;
}
```

Insert:

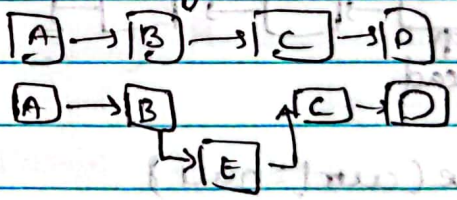
1) At front.



(1) newnod.next = head;

(2) head = neconode;

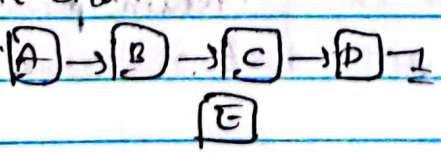
2) After a given node. (B)



e.next = b.next.

b.next = e.

(3) At end.

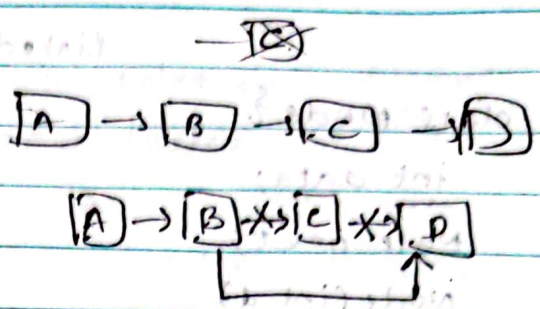


if head == null
e = head.

e.next = null;
Go to D
D.next = e;

Delete

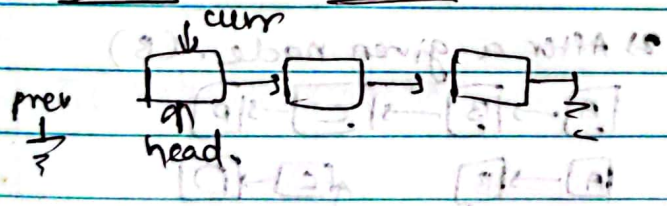
1) Find previous
2) change next of prev
3) Free memory of node to delete




```
delete(node n), head)
{
    curr = head;
    while(head→next != n)
    { head = head→next
    }
```

```
delete (node n, head)
{
    curr = head;
    if(head!=null & head.val = c)
    { head = null;
      return head;
    }
    while(curr.next != n)
        curr = curr.next;
    curr.next = n.next;
    n.next = null;
    free(n)
```

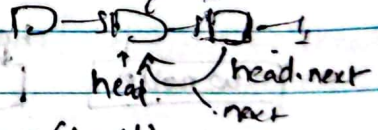
Reverse Iterative




```
while(curr != null)
{
    next = curr.next;
    curr.next = prev;
    prev = curr;
    curr = next;
}
return prev;
```

Recursive

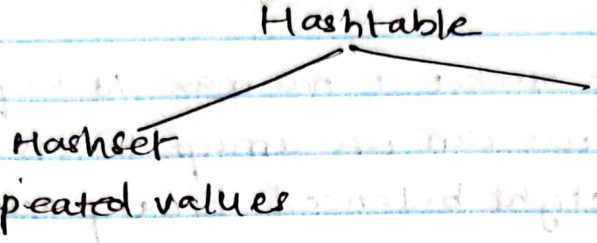
- go till end.




```
reverse(head)
if(head == null || head.next == null)
    return head;
listnode p = reverselist(head.next
head.next.next = head
head.next = null
return p.
}
```

Hashtable – hashmap, hashset.

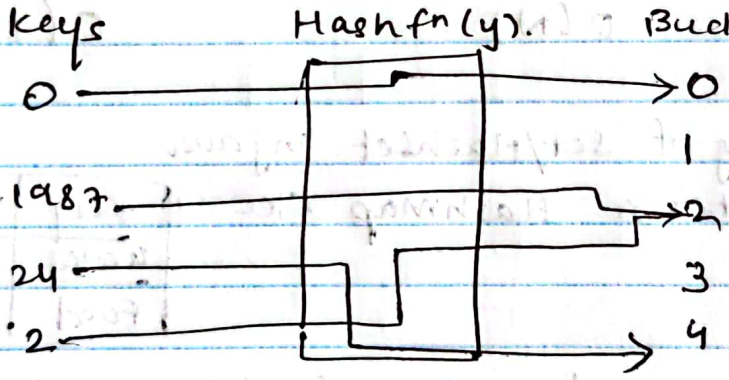
Hashtable: is a data structure which organizes data using hash functions in order to support quick insertion and search.



★ Principle of Hashtable

→ use a hash function to map keys to buckets.

eg. hash fn $y = x\%5$



★ Keys to Design a Hash Table

1. Hash function.

Hash fn will depend on the range of key values and the number of buckets.

- The idea is to assign a key to bucket as uniformly as pos
- ideal hash fn's 1-1 mapping b/w key & bucket.
- but usually its a tradeoff b/w the amount of buckets and the capacity of a bucket

2. Collision Resolution:-

1. How to organize values in same bucket?
2. What if too may values in same bucket?
3. How to search for a target value in a specific bucket

Lets say, in some bucket it has max N keys.

N = small => we can use arrays to store keys in same buc

N = large → height balanced binary search tree

N = max bucket size

	Array	Hashtable(LL)	Hashtable (BST)
nsert	O(1)	O(1)	O(1)
earch	O(N)	O(N)	O(log N)

Internal working of Set/Hashset in java

- Internally creates a HashMap like

key	val
Geeks	PRESENT
Food	Present

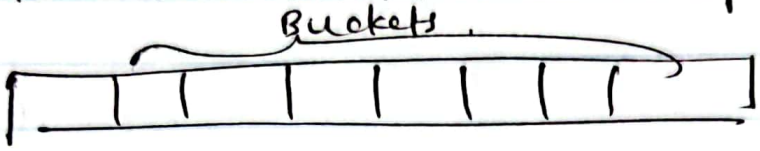
So when u call add method of hashset it internally calls put method of Hashmap with object u specified as key and 'PRESENT' as its value.

if (map.put(key, "present") == null, then add to hashset
↓ true internally hashmap

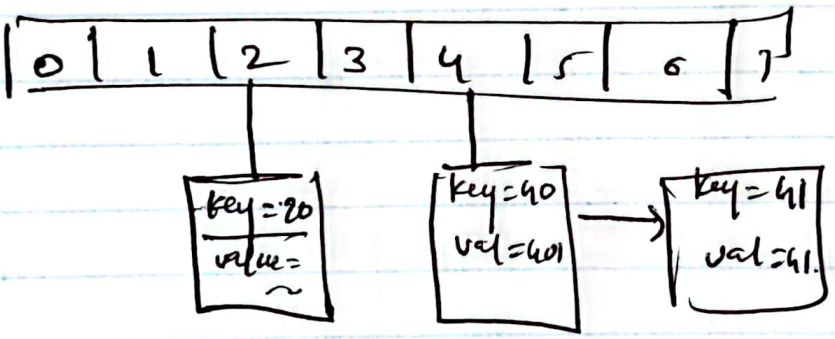
if (——) != null and hence not added.
↓ false:

Internal working of HashMap in java.

- Calculate hashcode() using the hash fn.
→ uses equals() method to compare keys.
- Buckets → one element of Hashmap array



While insertion / :-
- If no collision add to that bucket
- If key is same, update value
- If key is not same, add another node in the list



Java HashSet
add()
clear()
clone().
contains()
isEmpty()
iterator
remove(object o).
size().

HashMap.
put()
clear()
containsKey()
containsValue()
entrySet()
get().
keySet().
putAll()
remove().
remove(key, val)
replace(K,V).
size().

Search Algorithms

Linear search

linearSearch(-15)

arr = {20, 35, -15, 7, 55, 1, -22};

↑ ↑ ↑

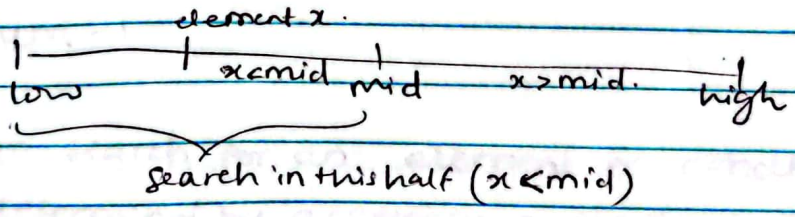
x x yes.

```
for (i=0; i< arr.length; i++)
    {
      if (arr[i] = =-15)
          return true.
    }
return false.
```

→ O(n)

Binary search

- Data must be sorted.



chooses element in middle of array & compares it against the search value.

1) low = 0, high = arr.length-1, mid = (low+high)/2.

```
while (low<= high)
  {
    mid = (low + high)/2;
    if (x == arr[mid]) return 1;
    else if (x < arr[mid])
        {
          high = mid-1;
        }
    else low = mid+1
  }
return 0;
```

2) Recursive

```
binsearch (arr[], x, low, high)
{
  mid = (high+low)/2
  if(low>high) return 0; //not found
  if (x>arr[mid]) return binsearch(arr, x, mid+1, high)
else if(x<arr[mid]) return —u— (arr, x, low, mid-1);
  else return 1 //found.
}
```

→ O(logn)

Space — O(1). for iterative algorithm.

Binary search

Overflow problem: $m=(l+r)/2$ - can overflow in java so use $m = l + (r-l)/2$ instead.

Template I

```
int binarysearch(int[] nums, target)
{
    if (nums == null || nums.length == 0)
        return -1

    int l = 0, r = nums.length - 1;
    while (l <= r)
    {
      m = l + (r-l)/2;
      if (nums[m] == target) return mid;
    else if (nums[m] < target) l = m+1;
      else r = mid-1;
      }
    return -1
}
```

Used to search for an element or condition which can be determined by accessing a single index in the array.

Template II

used to search for an element or condition which requires accessing the current index & its immediate right neighbour's index in the array.

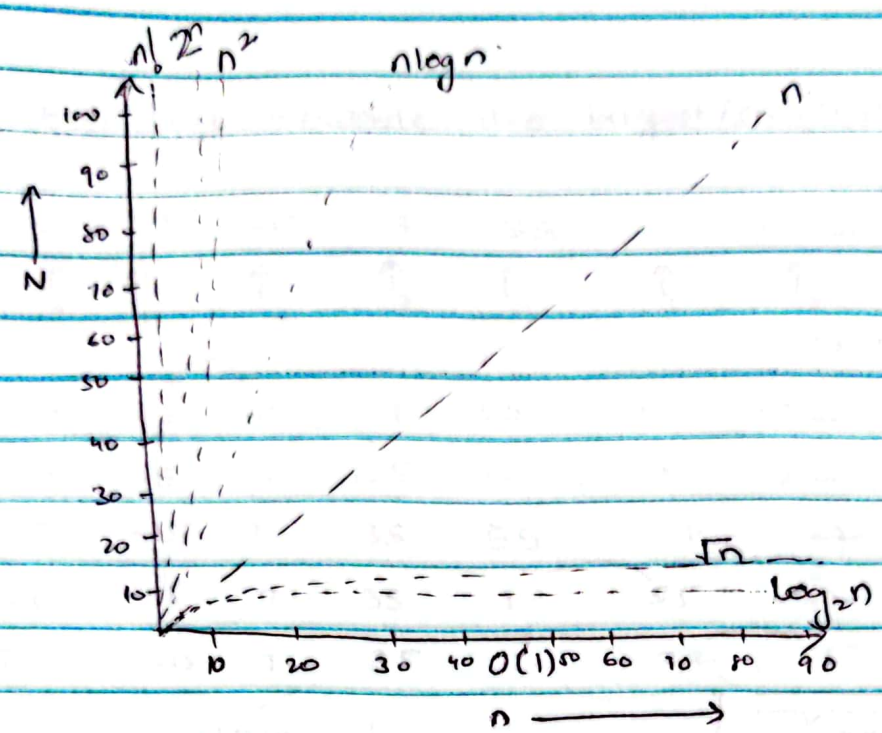
used when requires accessing current index and its immediate left and right neighbours index in the array.

Template -II

```
int binsearch( int[] nums, int target)
{
  if(nums ==null || nums.length==0).
        return -1;
  int l=0, r=nums.len;
  while ( l<r)
   {
   int mid = l +(r-l)/2;
   if (nums[mid]== target) return mid;
   else if (nums[mid] < target) left = mid+1;
   else right = mid;
  }
  if (left!= nums.len && nums[left] == target) return left
  return -1
}
```

Template -III

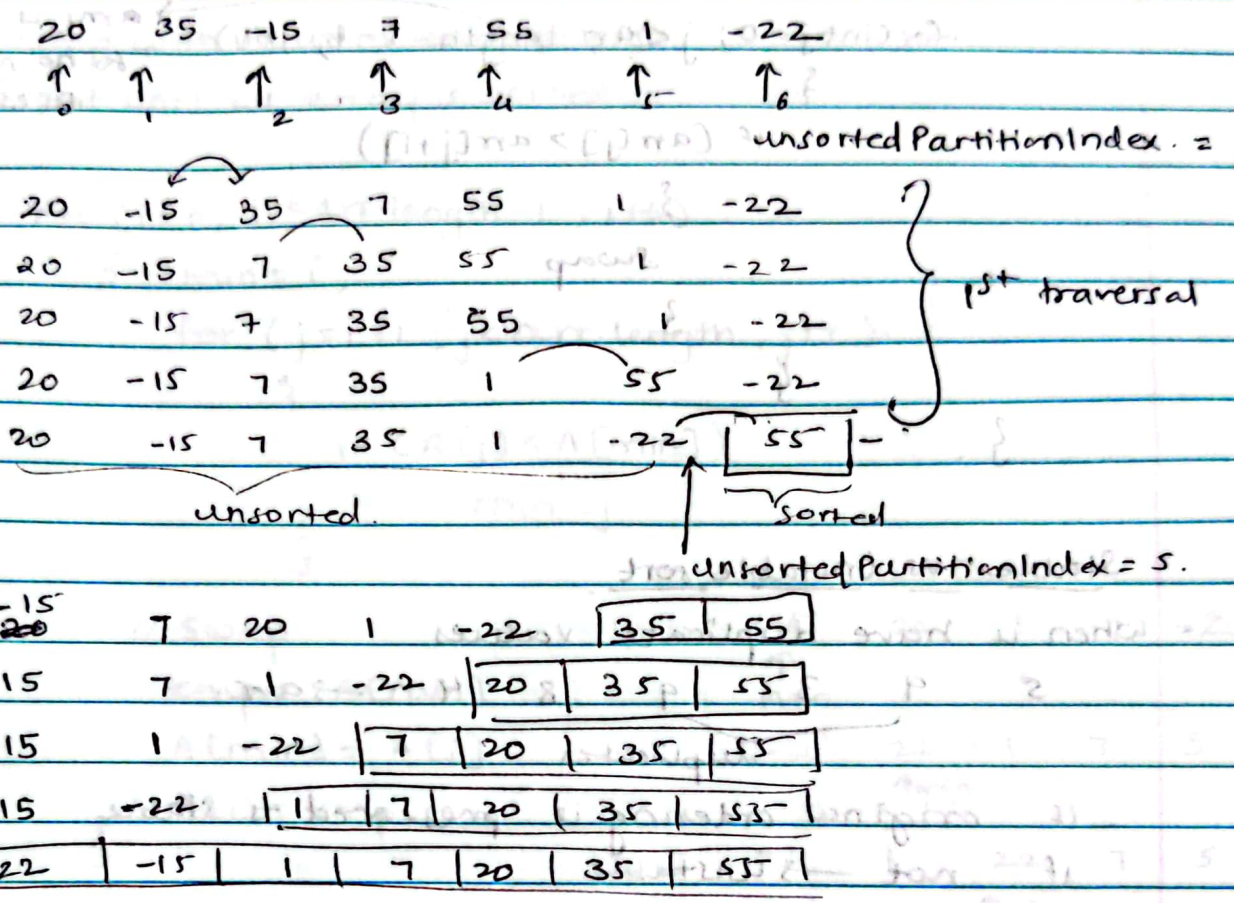
```
int binsearch(          )
 {
  if(nums == null || nums.length ==0)
      return -1;
  int l=0, r= nums.len-1;
  while( l+1 <r)
   {
   mid = l + (r-l)/2;
    if (nums[mid] == target)
     { return mid;
      }
    else if ( nums[mid] < target) { l=mid;}
    else r= mid;
  }
  if (nums[l]== target) return l;
```



← Time complexities & their graph.

$n! > 2^n > n^2 > n\log n > n > \sqrt{n} > \log n > O(1)$

Bubble Sort: - bubble the largest/smallest value to end



→ $O(n^2)$ → worst Best - $O(n)$. Avg: $O(n^2)$. Worst case space: - $O(1)$ auxilary.

→ Algorithm degrades quickly.

```
for (int lastUnsortedIndex = arr.length-1 ; lastUnsortedIndex > 0 ;lU ind
{
    for(int i=0 ; i<lastUnsortedIndex ; i++)
    {
        if (arr[i] > arr[i+1])
        {
            //swap
        }
    }
}
```

→ only adv → it can detect I/p is already sorted or not.

OR

```
for (int i=0; i<arr.length; i++)        ——> Run total element - 1 tim
    {
      for(int j=0; j<arr.length-i-1; j++)      ——> arry is sorted from en
          {                                        so no need to check
                                                   till end.
            if (arr[j] > arr[j+1])
                {
                  swap
                }
          }
    }
```

Stable vs Unstable sort.

- When u have duplicate values

  5 9 3 9 8 4 ——> arr.

  (9 9 — duplicates)

  - if original ordering is preserved ——> stable
    if not ——> unstable

- Stable preferred (consider string name, age, one sort in other)

Bubble ——> stable.

* Modified bubble sort

```
int unsortedarrayindex = arr.length-1
for (unsortedarrayindex; unsortedarrayindex>0; unso...--)
    {
      swapped=0
      for(int i=0; i<unsortedarrayindex; i++)
          {
            if(arr[i] > arr[i+1])        ——> Complexity O(n).
                {
                  // swap
                  swapped=1
                }
          }
    }
```

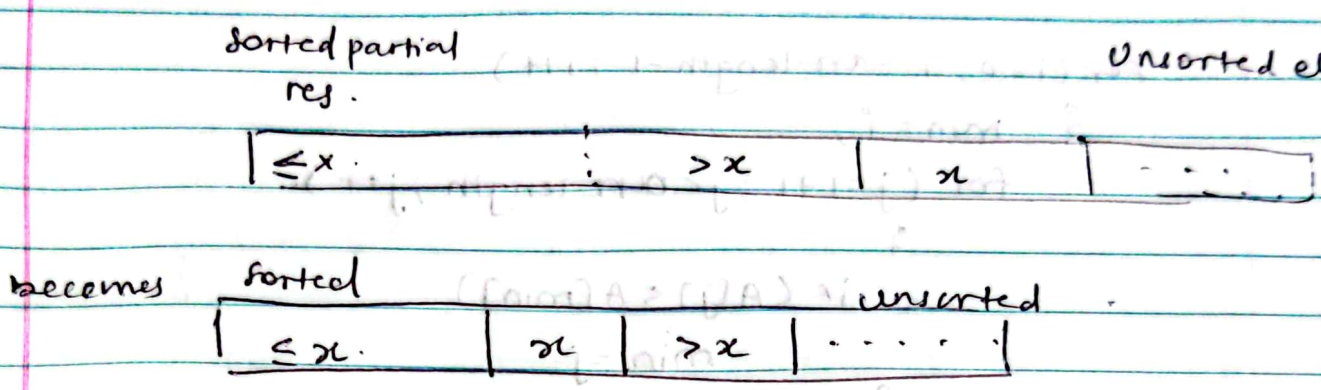
Selection sort - Repeatedly selects smallest element
- find min. in the list.
- Swap with value in current pos.
- Repeat until all array is sorted

```
for (i=0; i <arr.length-1 ; i++)
  {  min = i;
     for (j=i+1 ; j<arr.length ; j++).
        {
           if (A[j] < A[min])
              min = j
        }.
  // Swap
  temp = A[min]
  A[min] = A[i]
  A[i] = temp.
}
```

i=0	5 ↑min	22	1	7	-2
i=1	-2	22 ↑min	1	7	5
i=2	-2	1	22 ↑min	7	5
i=3	-2	1	5	7 ↑min.	22
i=4	-2	1	5	7	22

Worst case - $O(n^2)$
Best - $O(n^2)$
Avg $O(n^2)$
W. Space - $O(1)$.

Insertion sort :- inserts the element into correct position in already sorted list until no input elements remain.
The resulting array after k iterations has property that k+1 entries are sorted.




```
for (int i=1 , i < arr.length; i++)
  {
    int j;
    int partition = arr[i];
    j = i;
    while ( j>=1 && arr[j-1] > partition)
      {
          arr[j] = arr[j-1]      //shift to right.
          j--;
      }
    arr[j] = partition;
  }
```

Note : IMP
this should be first condition to check else j-1 goes to -1 & we get index out of bound when j=0

* Practically more efficient than selection & bubble, even when it is $O(n^2)$
- Stable

eg -

6 8 1 4 5 3 7 2

6 8 1 4 5 3 7 2

6 8 4 5 3 7 2

1 4 6 8 5 3 7 2

1 4 5 6 8 3 7 2

1 3 4 5 6 7 8 2 - missing iteration.

1 2 3 4 5 6 7 8.

Worst case - when all elements have to move right.

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

$\simeq O(n^2)$.

Worst - $O(n^2)$

Best - $O(n)$

Avg - $O(n^2)$.

W space - $O(n^2)$ total, $O(1)$ auxiliary.

- used when data is nearly sorted due to its adaptiveness or i/p is small.

	comparision	Inversion/Swap
Bubble.	$\frac{n^2}{2}$	$\frac{n^2}{2}$
Selection	$\frac{n^2}{2}$	n.
Insertion	$\frac{n^2}{4}$	$\frac{n^2}{8}$.

- Insertion sort is almost linear for partially sorted I/p.
- Selection sort is best suit for elements with bigger values & small keys.

Shell sort:

- variation of insertion short.
- Insertion short chooses the element using gap = 1.
- Shell sort → larger gap
- Goal → reduce amt. of shifting reqd
- last gap value is always 1.
- So, shell sort does some preliminary work (using gap values > 1) & then becomes ins. sort. By the time it is ins sort, elements are partially sorted. ⇒ less shifting reqd

Gap value -

Knuth sequence - $(3^k - 1)/2$

k	gap
1	1
2	4
3	13
4	40

Alternative -

gap = arrysize/2

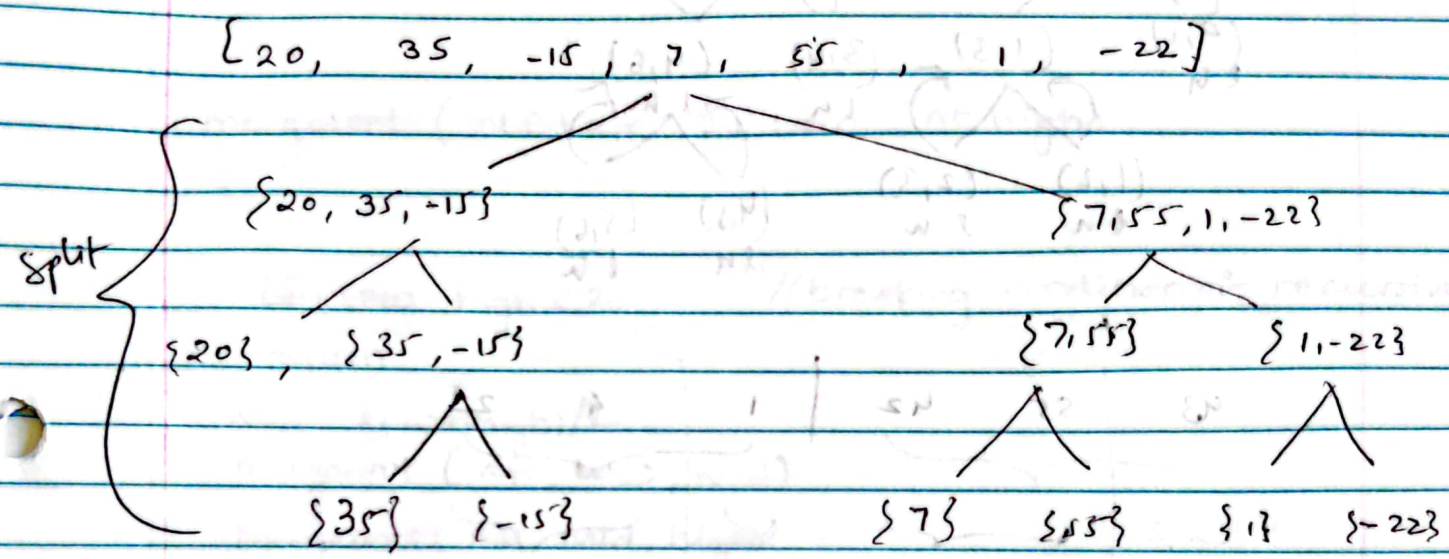
on each iteration gap = gap/2

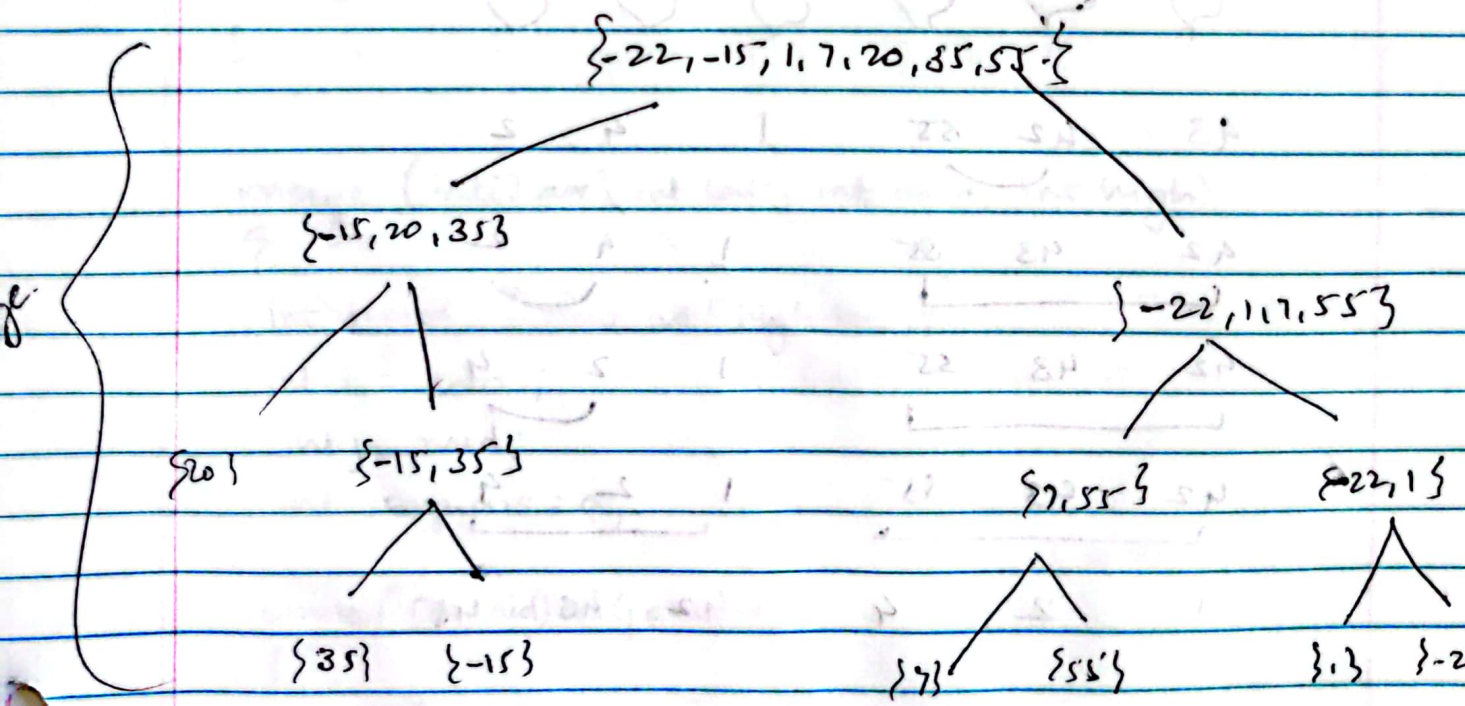
- In place algo.
- $O(n^2)$. Best known worst case = $O(n \log^2 n)$. Best - $O(n)$ Space = O(
but can perform much better than $O(n^2)$
- → unstable

```
for ( int gap = array.length/2 ; gap>=1; gap/2)
    {

      for(int i=gap ; i< arr.length; i++)
       {
       int j = i;
       int partition = arr[i];
       while(j>=gap && arr[j-gap]> partition)
              {
                arr[j]= arr[j-gap]
                j= j-gap
              }
       arr[j]= partition;
    }
}
```

Merge Sort

- Divide & Conquer algo, recursive.
- Split & Merge.
- logical spliting, no new arrays are created.
- not in place sort, uses temp. arrays.

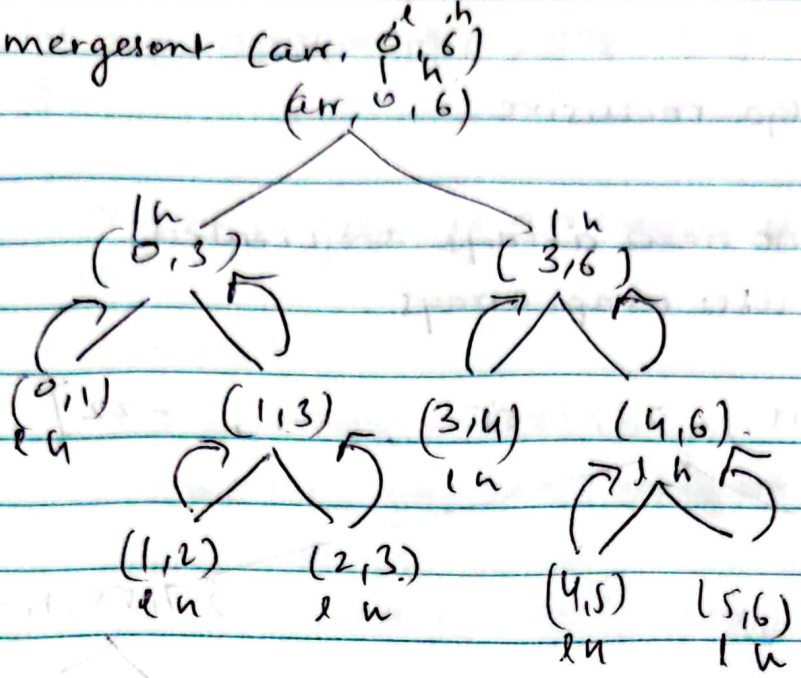


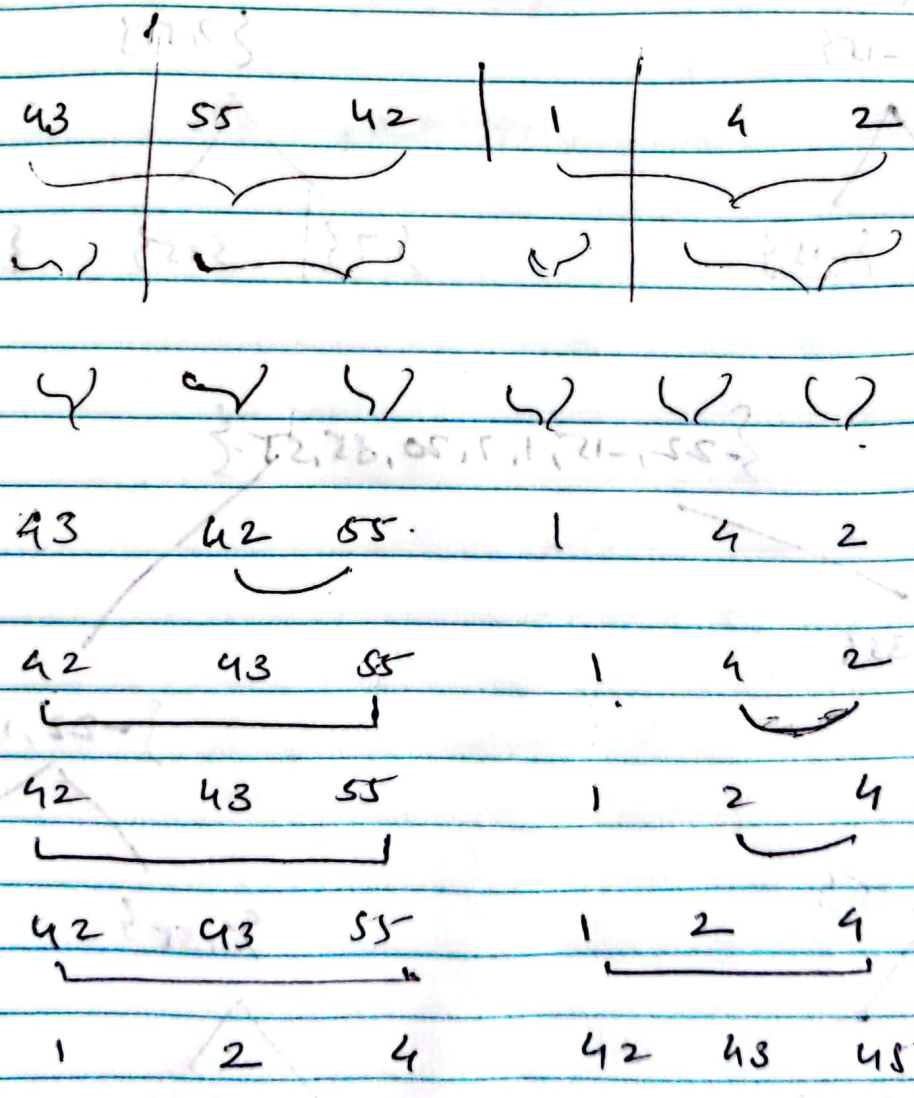


merge is not inplace.

eg- 43, 55, 42, 1, 4, 2

mergesort (arr, 0, 6)





```
main( str args[])
 {
    int arr =new{ 43, 55, 42, 1, 4, 2}
    mergesort (arr, 0, arr.length);

 }

mergesort (int[] arr, int low, int high)
{

  if (low - high < 2)        //breaking condition of recursive f
  return
 mid = (low+high)/2
 mergesort ( arr, low, mid) ;
 mergesort( arr, mid, high) ;
 merge ( arr, low, mid, high);
}


merge (int[] arr, int low, int mid, int high)
{
 int temp[] = new int[high-low];
 int i = low ;
 int j= mid;
 int temp-pos = 0;

 while ((i< mid) && (j<high))
 {
      if (arr[i] < arr[j])
           { temp[temp-pos]= arr[i]
             i++; temp-pos++;
```

```
        else
          {
            temp[temp_pos] = arr[j];
            j++; temp_pos++;
          }
    }

while(i<mid)                    // copy remaining from left array
 {
  temp[temp_pos] = arr[i];
   i++;
   temp_pos++;
 }

while(j<high)                   // copy from right array.
 {
   temp[temp_pos] = arr[j];
    j++;
   temp_pos++
 }

 int K = 0
 for( i=low; i<high; i++)
    {
      arr[i] = temp[k];
       k++;
    }

}
```

Heap.

- Priority Q is an ADT, Heap is a Data Structure.
- Heap is a way to implement PQ.

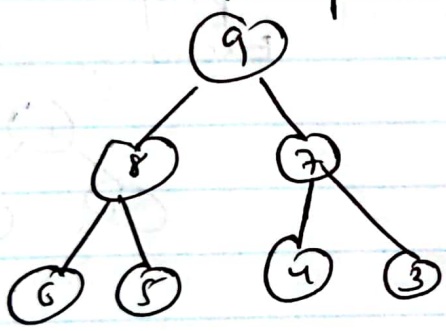
- PQ using array/LL → O(n) for other operations
  O(1) for insert/ delete.

- Heap → PQ - Insert/Delete = O(log N).
  max/min - O(1).

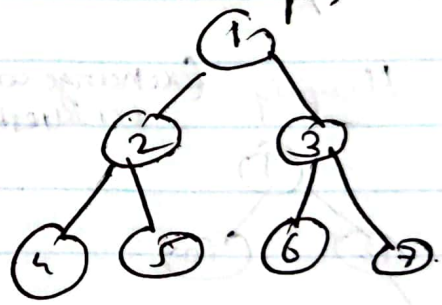
Heap :→

- a complete binary tree (all nodes filled, top to bottom, left to
- value of each node must be no. greater / no less than child nodes.

Max Heap



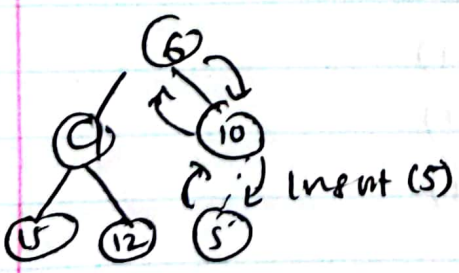
Min Heap.



Insert

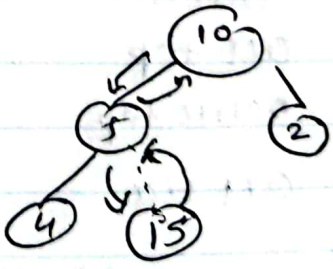
min heap

- Insert in bottom right subtree



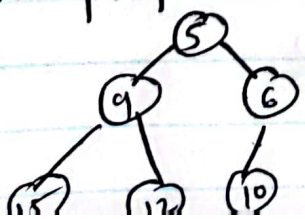
max heap.

- Insert into bottom of left

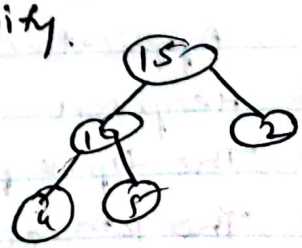


Inse

- Heapify.

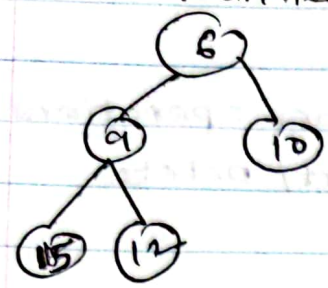


- Heapify.

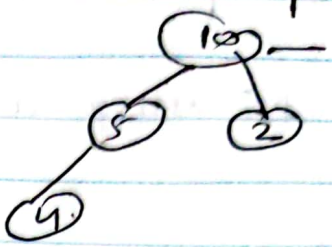


Heap Deletion.

min heap

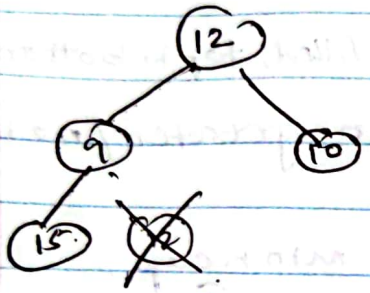


max heap.

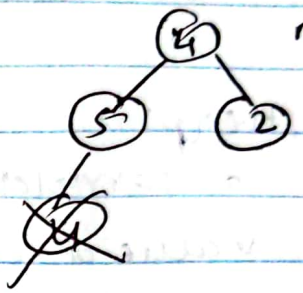


Delete 6

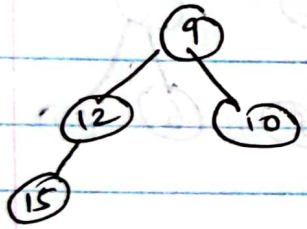
move bottom ~~left most~~ right element to root & delete that



Delete 10, make bottom right element as root & delete that el.

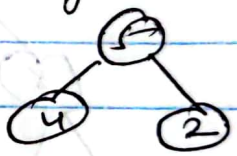


Heapify - (exchange with smallest child)



Heapify.

Exchange with largest chi



Method	Time	Space
construct	O(N)	O(N)
insert	O(logN)	O(1)
get top	O(1)	O(1)
delete top.	O(logN)	O(1)
get size.	O(1)	O(1)

Applications

* Heap sort — O(NlogN) Space O(N)
* The top-k problem. O(N + klogN)
* The kth element. - ~~—~~

maxheap (PQ)

PriorityQueue<Integer> maxHeap = new PQ<>(Collections.reverseOrder()

↓ for min heap dont use this

maxHeap.add(1)
maxHeap.add(3);
maxHeap.add(2);

SOP- maxHeap.toString()

maxHeap.peek()
maxHeap.poll()
maxHeap.size().
maxHeap.isEmpty()

PQ fns-
add(E e)
clear() → remove all.
comparator()
contains(Object o).
iterator()
offer()
peek()
poll()
remove(o)
size()
toArray()
~~toArray(T[] a)~~

Heap applications

Heap sort - O(nlogn)

Top k problem. & 3) kth element - similar approach.

Approach1 : top k largest elements

1. Construct max heap
2. Add all elements to it — O(n)
3. Traversing & deleting top element (pop() / poll()) and store value in result array T. — O(logn)
4. Repeat 3 until we have removed k largest elements. C

Time - O(klogn) + O(n) = O(n + klogn). Space - O(n). (step 2)

similar for smallest k elements.

Approach 2: top k largest elements.

1. Construct a min heap
2. Add elements to min heap one by one
3. When there are k elements in min heap compare current element with the top element of heap.
4. If curr is no larger than top, drop it and proceed to next.
5. If curr is larger than heap's top element, add the curr to min heap.
6. Repeat steps 2 & 3 until all elements have been iterated.

Now the k elements in min heap are k largest elements

Time: O(nlogk)

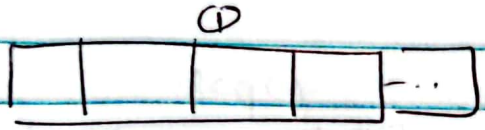
O(N-K)logk + Klogk. = O(nlogk)

Space - O(k).

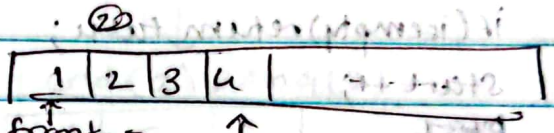
Queues

- ADT
- FIFO - first in first out.

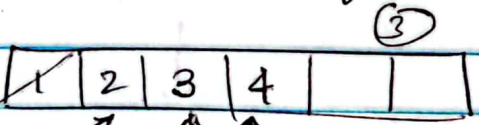
enqueue => add to end of queue.
dequeue => remove from front
peek => get item at front but dont remove it.

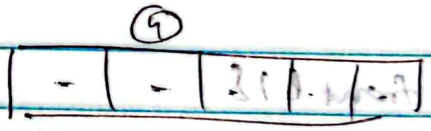


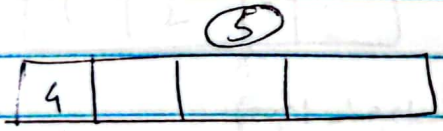
front = back = 0



// front is 0 always







⑥

Queue (LinkedList)

Queue using LinkedList –

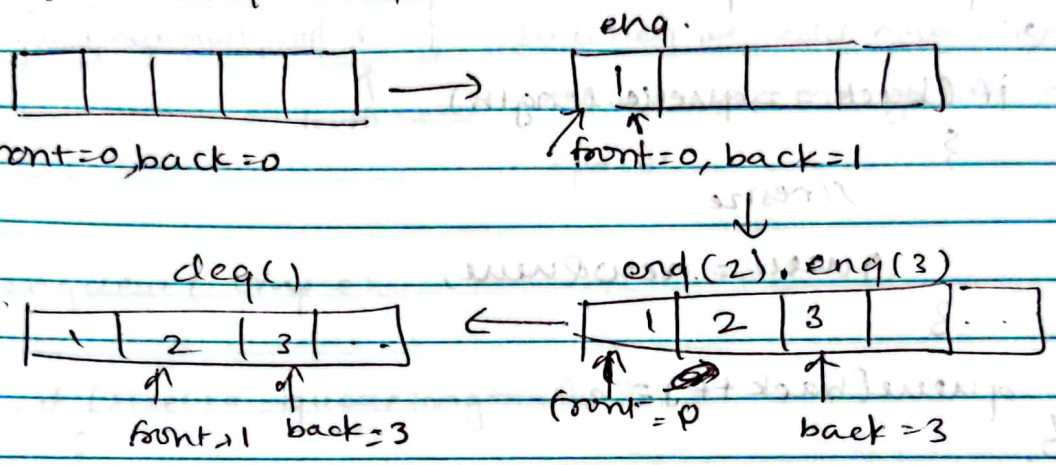
```
start = 0;
List list;
enqueue(x)
{
  list.add(x);
}

dequeue
{
  if(isempty) return false;
  start ++;
  return list.get(start)
}

front()
{
  return list.get(start);
}

isEmpty()
{
  return start >= list.size();
}
```

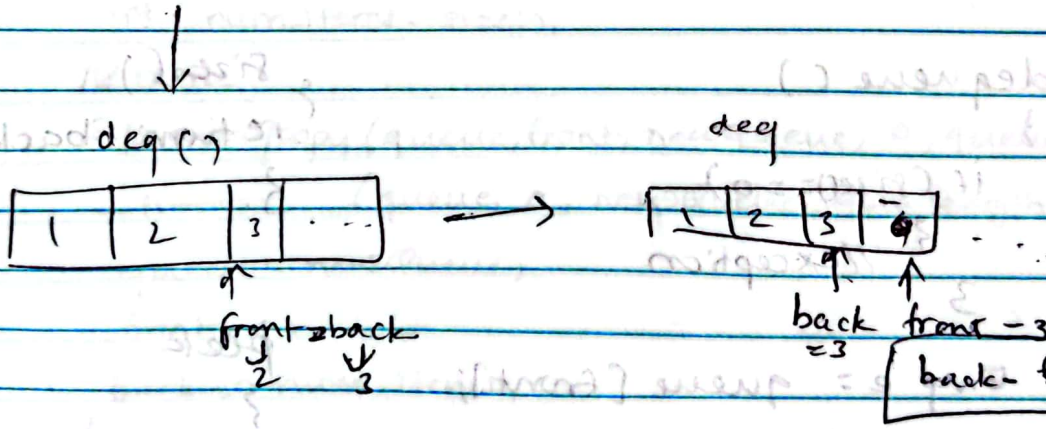
Circular Queue.

Queues (Array).

- ADT
- FIFO

enqueue, dequeue, peek





empty → ~~front-back~~ back-front ≥ 1 → not empty. ≥ or >

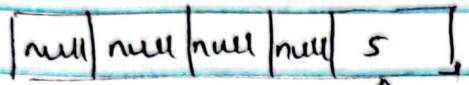
size = back-front.

```
public class ArrayQueue {

    private Employee[] queue;
    private int front=0, back=0;

    enqueue(Emp e)
    {
      if (back == queue.length)
        {
            //resize.
            queue = newQueue;
        }
      queue[back++] = e.
    }

    dequeue ()
    {
      if (size()==0)
        {
            //exception
        }
      Emp e = queue [front];
      queue[front] = null;
       front ++;

      if (size() == 0)
       { front = back = 0;
        }
    }

    size ()
    {
    return back - front;
    }

    peek
    {
        return queue [fron
    }
```

Circular Queue:

What if u have only n elements at a time in a queue.
You dont have to resize

eg- queue of size 5



So when we add new element, it is resized even though we have 4 empty spaces.

```
enqueue( Emp e).
{
  if (size() == queue.length-1)
   {
     int num_items = size();
    // resize
    sys:arraycopy(queue, front, newQueue, 0, queue.length-front)
    ——//——  (queue, 0, newQueue, que.length-front, back),
     queue = newQueue;
     front=0;
     back = num_items;
   }.
  queue[back] = e;
  if(back < queue.length-1)
   {
     back++
   }
  else
   {
     back=0
   }
```

```
size()
 {
   if (front <= back) return back - front
   else return back - front + queue.length;
 }

dequeue
 {
   if (size() == 0)
       }
          // no element exception
   }

  Emp e = queue[front];
  queue[front] = null;
   front++;

  if (size() == 0)
    { front = back = 0;
   }
else if (front == queue.length)
    {
      front = 0;
    }.
  return e;
}

printQueue()
  {
     if (front <= back)
        { for i -> front to back
                print
```

```
else
  {
     for (i = front ; i < queue.len; i++)
       {
          //print
     }

     for (i = 0, i < back; i++)
       {
          //print
     }
}.
```

Time - O(1) - if not resizing.
O(n) → if resizing

Java built in queues.

```
Queue<Integer> q = new LinkedList();
 q.peek();
 q.offer(5);
q.offer(3);
q.offer(8);    //push
 q.poll();     //pop
 q.size();     //size.
```

```
CircularQue{
 List<Integer> queue = new LinkedList<>();
front = back = 0;
int capacity;

 Size().
 { if (front<=rear) return r-f;
   return r-f+ capacity;
 }
enqueue(x)
 if(queue.size < capacity)
  {
   queue.add(x)
    rear++;
    return
  }
 if ( queue.size() == capacity)
 {
   if(size()== capacity)
      return
  else{
     if(f<=r)
      {
        r = 0
       queue.set( rear, x);
       r++
      }
     else
      {
        queue.add(x)
        r++
      }
  }
```

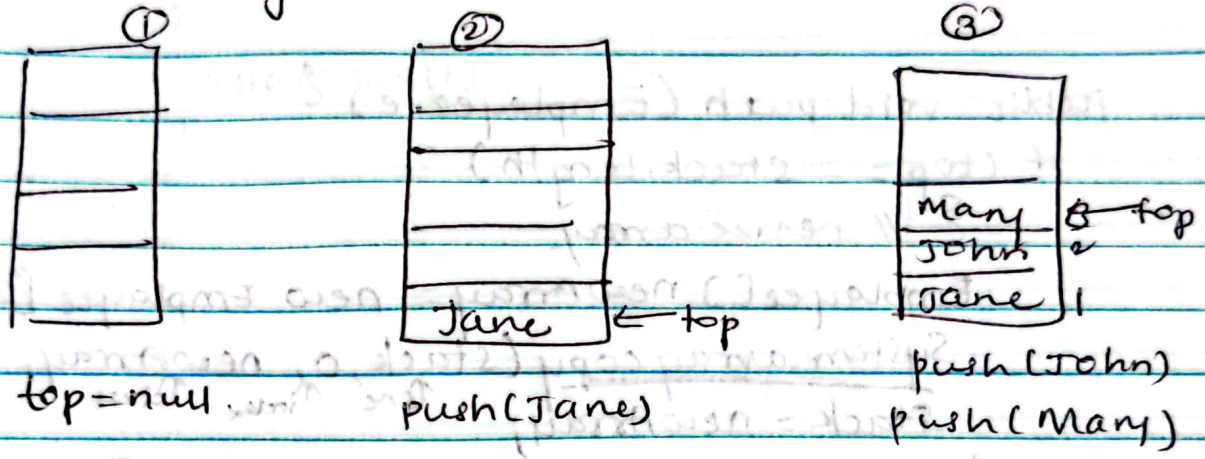
```
dequeue()
{
if(size()==0) return
 if (front == rear)
 {
    if f = 0
      r = 0
      return
 }

 front++
}.
```

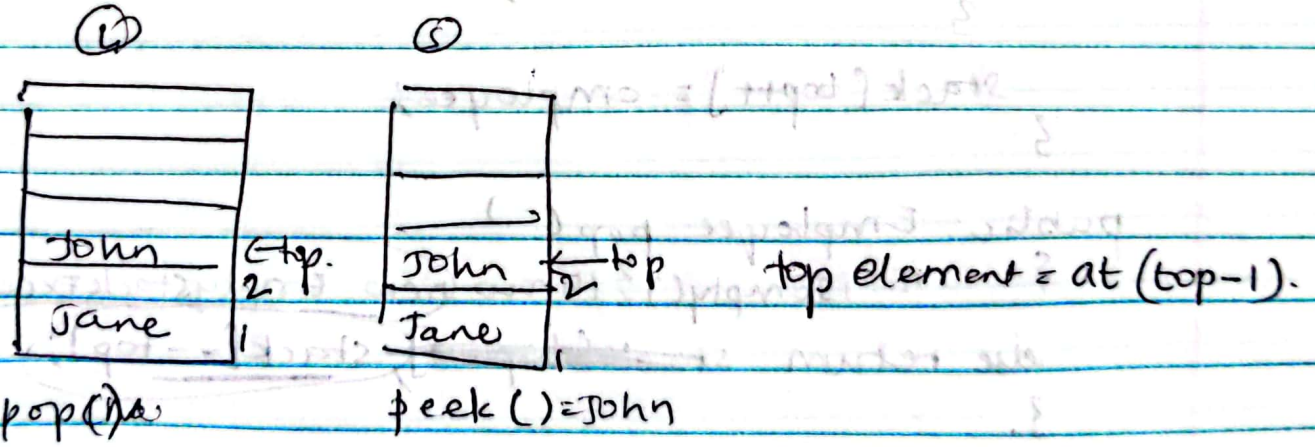
Stacks - an ADT.

LIFO → Last In First Out.

push: add an item to top.
pop - remove top item
peek → get top item without removing.

Ideal backing DS - linked list.





Time-	Push	Pop	Peek
Linked list	O(1)	O(1)	O(1)
Array	O(n)	O(1)	O(1)

(good if u know max items, so array needs not resized.
memory is tight then arrays.
else linked list

```
public class Arraystack{

    private Employee[] stack;
     private int top,        //=0 initially

    public Arraystack(int capacity)
     {
       Stack = new Employee[capacity];
     }


   public void push(Employee e) {
     if (top==stack.length)
         { // resize array.
         Employee[] newArray = new Employee[2*stack.len
         System.arraycopy(stack, 0, newArray, 0, stack.leng
                          ↑src  ↑index  ↑dest  ↑index
         Stack = newArray;
        }

      stack[top++] = employee;
   }

 public Employee pop( )
   { if isEmpty(){throw new EmptyStackException()}
   else return stack[top]; stack[--top];  stack[top] = n
   }.

 public boolean isEmpty()
     {
        return top==0;
     }

 public Employee peek() {
        if (isEmpty()){ throw new EmptyStackException();}
        return stack[top-1];    //we are not decrementing
                                  top here
```

```
public int size()
  {
    return top;
  }

public void printstack()
  {
   for(int i= top-1; i>=0 ;i--)
        {
          Syso(stack[i]);
        }
  }
```

Stacks- linkedlist

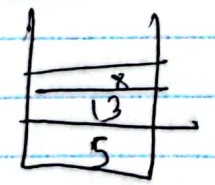
```
public class LinkedStack {                    // use class for abstraction else everyone will call any other fn of LinkedList

    private LinkedList<Employee> stack;
    public LinkedStack()
    {
        Stack = new LinkedList<Employee>();
    }

    public void push (Employee e)
    {
        stack.push(e);                        ——— stack.add(n)
    }

    public Employee pop()
    {
        return stack.pop()                    → stack.remove(stack.size()-1)
    }

    public Employee peek()
    {
        return stack.peek();                  → stack.get(stack.size()-1)
    }                                         return Stack.isEmpty()

    public boolean isEmpty()                  → return stack.size == 0;
    {
        return stack.isEmpty();
    }

}
```

Java built-in stack

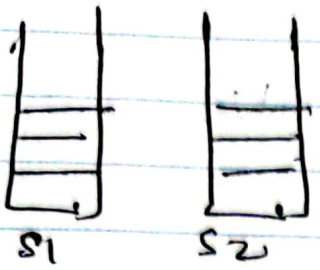
```
Stack<Integer> s = new Stack<>();

s.push(1)
s.push(2)

if (s.empty() == true)
{
  sop(empty)
  return;
}

s.pop();          // 2

s.peek();         // 1

s.size            // 1
```

search: finds element

search(5) = 3



~~Stack~~ Queue using 2 stacks.




```
push(x)
{
  while (S2.Isempty()
    {
      S1.push(S2.pop())
    }
  S1.push(x)
}
```

```
pop
{
while (S1.isempty)
  {
   S2.push(S1.pop())
  }
return S2.pop()
```

```
peek()
{
while(S1.isempty())
  {
   S2.push(S1.pop())
  }
return S2.peek()
}
```

```
empty
return S1.isempty() &&
       S2.isEmpty().
```

Abstract Data Type

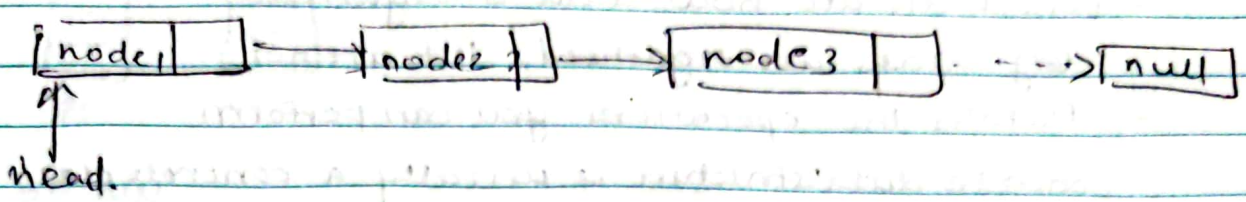
- Doesn't dictate how data is organised (eg arrays are organised, list are not)
- Dictates the operations you can perform
- Concrete data structure is usually a concrete class
- Abstract data type is usually an interface.

Arraylist

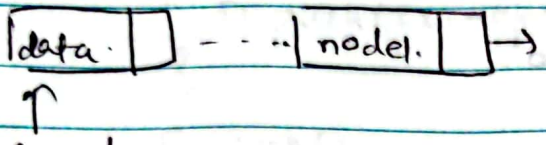
- Inital capacity by default = 10.
  .add( )
  .get(index)
  .isEmpty()
  .set(2,"JohnX")
  .size()
  .add(3,"Does")
  .toArray (~~new~~) → convert to array.
  .contains (obj)
  .indexOf
  .remove(3)

Vector: when u have to be thread-safe
→ when synchronization is needed
→ slower than arraylist

Linked list

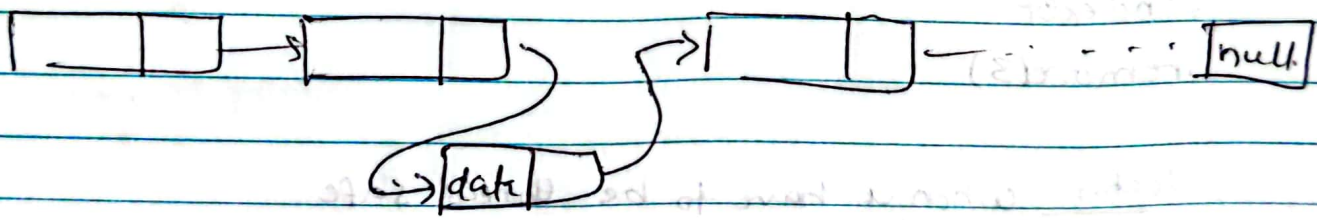


Insert.
at start —

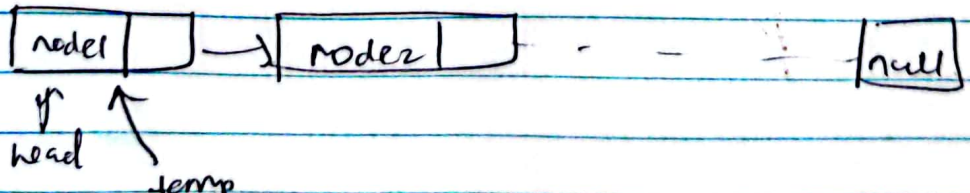


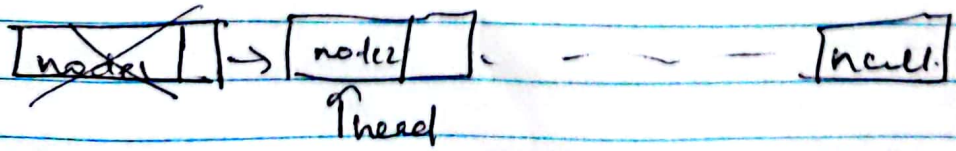
at end

random pos.

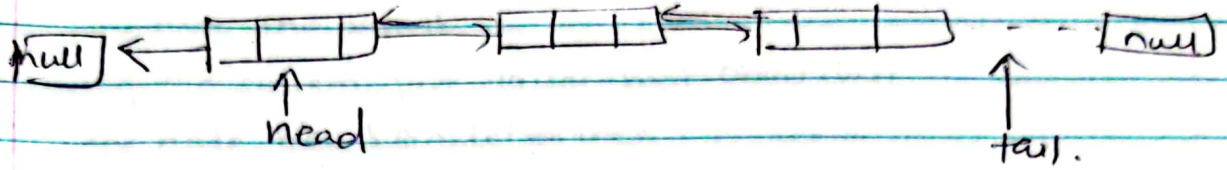


Similarly deletion
first node





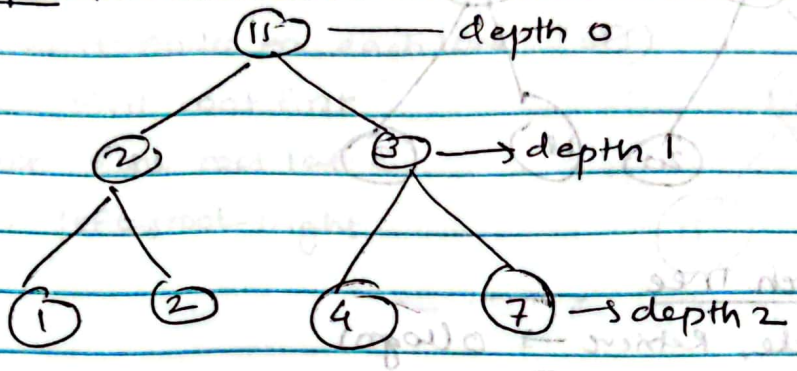
Doubly linked list-



// Task — Implement SLL, DLL, list out list methods.
addfirst, addlast etc.

Tree - a hierarchial data structure
- nodes, children, edges, parent (only one at max)
- root → no parent.

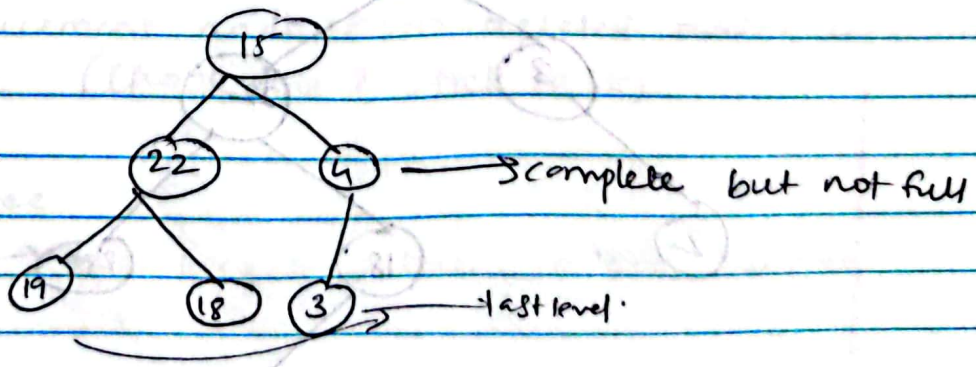
eg- file system, java classes - root - Object class
- one node tree → singleton tree.
- no cycles.



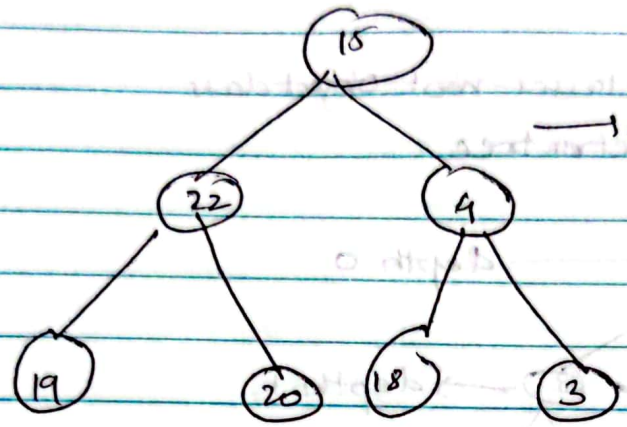
height → no. of edges to longest path from node to leaf

Binary Tree.
- Every node has 0, 1 or 2 child.
- left & right child

Complete Binary Tree - if every level except last one has 2 children and in last level all nodes are as left as possible



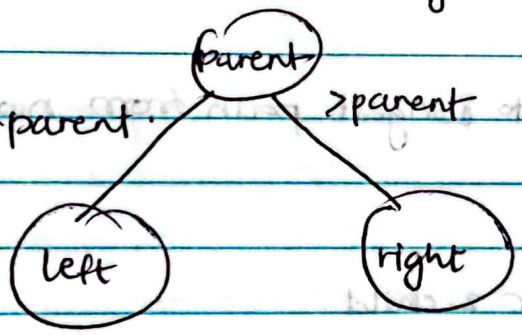
full binary tree - complete
— every node except leaf has 2 childres



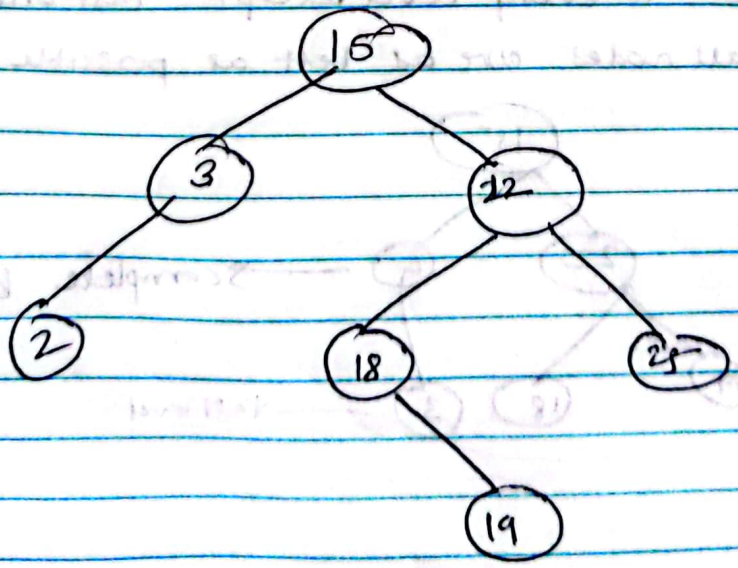
→ full

Binary search tree

Insert, Delete, Retrieve → O(logn).



Search - O(logn)



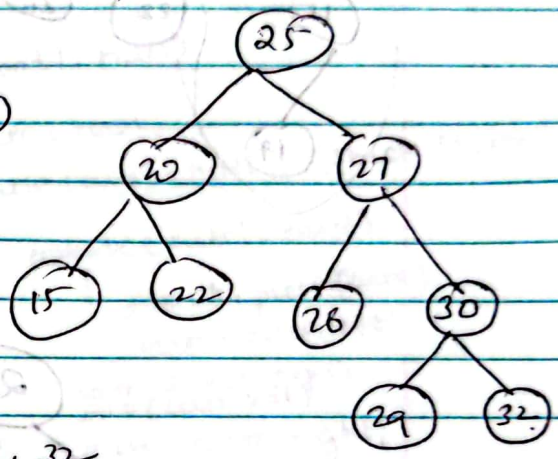
duplicates not allowed
or store in fix side (L or r)

- min → left edge till leaf
- max → right edge till leaf.

AVL, Red Black → *

Traversal

1) Level → visit nodes on each level (BFS)
2) Preorder - visit root first
3) Postorder - visit root last
4) Inorder - left → root → right



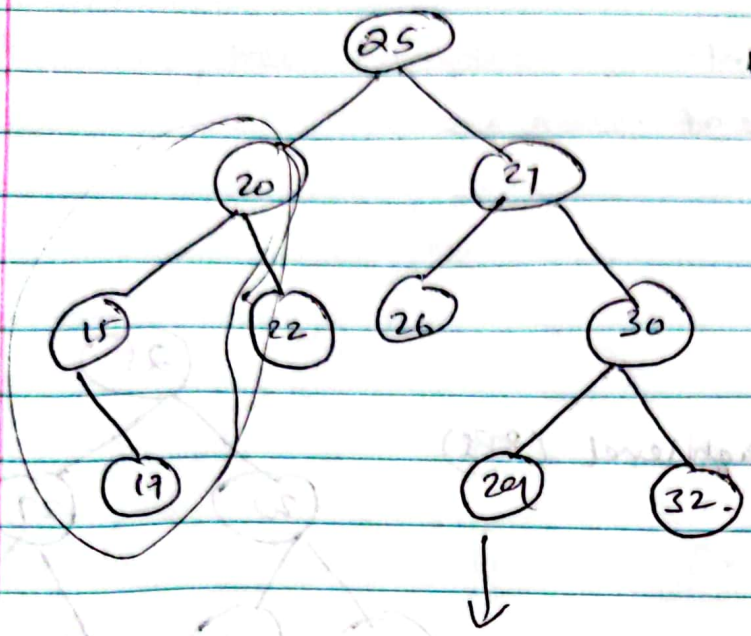
Level 25, 20, 27, 15, 22, 26, 30, 29, 32
Preorder 25, 20, 15, 22, 27, 26, 30, 29, 32
Postorder 15, 22, 20, 26, 29, 32, 30, 27, 25
Inorder 15, 20, 22, 25, 26, 27, 29, 30, 32. — sorted data

Delete —

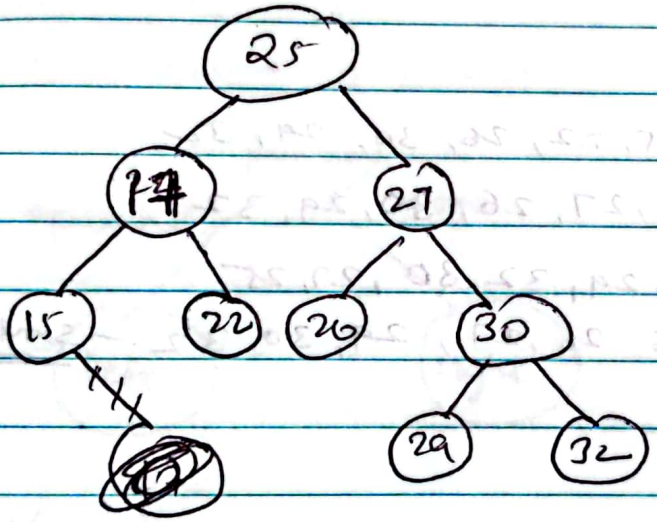
1) leaf — remove
2) Node has one child → replace child by node
3) 2 child.
Can take replacement node from deleted node's left subtree or right subtree (Choose one & stick to it)

If left subtree
→ take largest value in left subtree

If right subtree
→ take smallest value in right subtree.

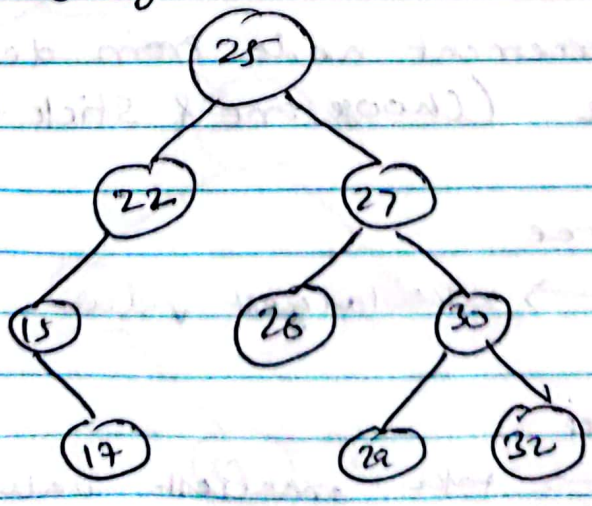


delete – 20,
(left subtree)

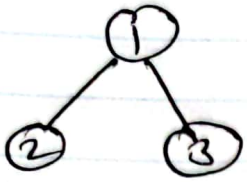


Delete 20 (Right subtree)

Op:



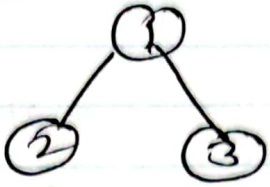
Tree Traversals



Inorder

left - root - right

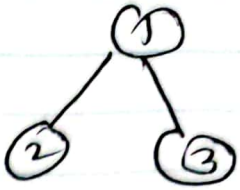
2 - 1 - 3



Preorder

root - left - right

1 2 3



Postorder

left - right - root

2 3 1

level order => BFS.

```
Inorder (Treenode root)
{
 if(root!=null)
 {
   inorder(root.left)
   sop(root.val)
  inorder(root.right)
 }
}
```

```
if(root!=null)
{
 curr=root;
 while(curr!=null || !stack.isEmpty())
 {
   while(curr!=null)
    {
      stack.push(curr)
      curr=curr.left;
    }
   curr=stack.pop()
   sop(curr.val)
   curr=curr.right;
 }
}
```

```
Preorder (Treenode root)
{ if(root!=null)
  { sop(root.val)
   preorder(root.left)
   preorder(root.right)
  }
}
```

```
stack.push(root)
curr=root;
while(!stack.isempty())
{
  if(curr.right!=null) stack.push(right)
  if(curr.left!=null) stack.push(left)
  curr=stack.pop()
  sop(curr.val)
}
```

```
Postorder(Treenode root)
{
  if(root!=null)
  {
   postorder(root.left)
   postorder(root.right)
   sop(root.val)
  }
}
```

```
Stack<Tree> stack = new stack<>();
 stack.push(root);
while(!stack.isEmpty())
{
 curr=stack.pop();
 if(curr.left!=null || curr.right!=null)
 { stack.push(curr);
  if(right!=null) stack.push(curr.right)
  if(left!=null) stack.push(curr.left)
  curr.right=null; curr.left=null;
 }
 else sop(curr.val)
}
```

Tree Traversal

Preorder traversal - (root, left, right)

```
preorder(root)
{
  if(root==null) return res;
   res.add(root.val);
  preorder(root.left);
 preorder(root.right);
  return res;
}
```

O(n) Time & Space

```
preorder(root)
{
if(root == null) return res;
Stack<Tree> st = new stack<>();
 st.push(root);
while(!st.isEmpty())
 {
   Treenode node = st.pop();
   res.add(node.val)
   if(node.right!=null)st.push(ra
                                 righ
   if(node.left!=null)
          st.push(node.left)
   }
   return res;
}
```

Time - O(n) O(n) Space

Inorder (left, root, right).

```
inorder(root)
{
  if(root==null) return res;
 inorder(root.left);
   res.add(root.val);
  inorder(root.right);
  return res;
}
```

O(n) Time & Space

```
inorder(root)
{
if(root==null) return root;
 Stack <TreeNode> st = new Stack
TreeNode curr = root;
while (curr!=null || !st.isEmpty
 {
   while(curr!=null)
    {
      st.push(curr)
      curr=curr.left;
    }
   curr=st.pop()
   res.add(curr.val)
  curr=curr.right;
 }
 return res;
}
```

O(n) - Time & Space

Height balanced BST

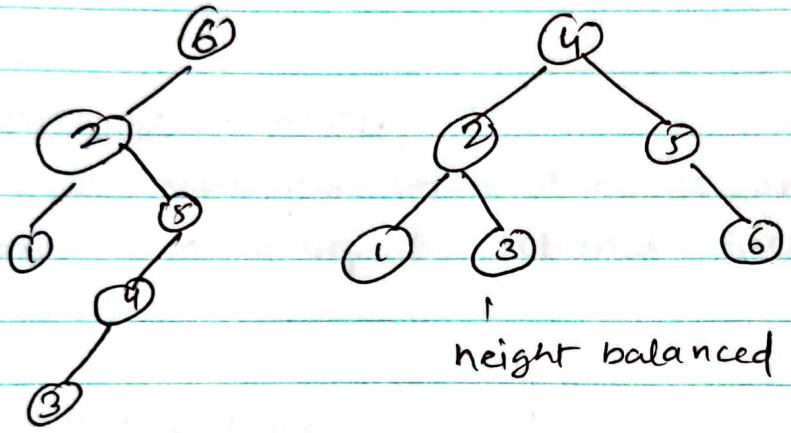
Depth of a node → no. of edges from root to the node

height of a node - no. of edges on longest path between that node and leaf

height of tree = height of root.

Height balanced BST -

- height is log(N) → N = no. of nodes
- height of two subtrees of every node never differs by more than 1'.



normal BST

height balanced

- improves performances
- as in BST most ops time depends on height O(h)
  ∴ if height is log N time is reduced from N to log N.

Popular height balanced BST.

- Red black tree
- AVL tree
- Splay tree
- Treap

- In Java. ~~Set~~ or Treeset OR C++ · set it is implemented using height balanced BST ∴ Time = O(log N)

Also in hashset, it is implemented by hash but if there are too many elements with same hash key it will cost O(N) to look for a specific element. where N is no. of elements with same hash key. Typically, height balanced BST will be used here to improve performance from O(N) to O(logN).

Top-Down / Bottom-up.

Top down. (preorder).

first visit the node to come up with some values and pass these values to its children. while calling the fn recursively.

eg.

```
return if root is null
if root is a leaf:
        answer = max(answer, depth)
max_depth(root.left, depth+1)
max_depth(root.right, depth+1)
```

Bottom up (postorder)

Call fn recursively for children node and then come up with answer according to returned values and value of curr node itself.

eg.

```
return 0 if root is null.
left_depth = max_depth(root.left)
right_depth = max_depth(root.right).
return max(left_depth, right_depth)+1.
```

BFS & DFS.

BFS - Template - 1. (Shortest path b/w root & target)

```
int BFS (Node root, Node target)
 Queue<Node> queue;
 int step = 0;
 add root to queue;
while(queue.is not empty){
  int size = queue.size();
  for (i=0 to size)
   {
   Node cur = the first node in queue;
   return step if cur is target;
   for (Node next: neighbor of cur)
     {
       add next to queue;
     }
   remove first node from queue;
  }
  step = step+1;
}
return -1;
 }
```

BFS - II (visited)

```
int BFS (Node root, Node target) {
    Queue<Node> queue;
    Set<Node> visited;
    int step = 0;

    add root to queue;
    add root to visited;

    while (!queue.isEmpty()) {
      int size = queue.size()
      for (i = 0 to size)
        {
          Node cur = first node in queue;
          return step if cur == target;
          for (Node next : in neighbors of cur)
          {
            if (next is not in visited)
              {
                add next to queue;
                add next to visited;
              }
          }
          remove first node from queue;
      }
      step = step + 1;
    }
    return -1
}
```

DFS - Path may not be shortest.

DFS.

I)

```
boolean dfs( Node cur, Node target, SetcNode> visited) {
 return true if cur == target;
 for ( next: each neighbour of cur) {
     if(next is not in visited)
       {
        add next to visited;
       return true if dfs( next, target, visited) = =true.
       }
   }
  return false;
}
```

II) Iterative

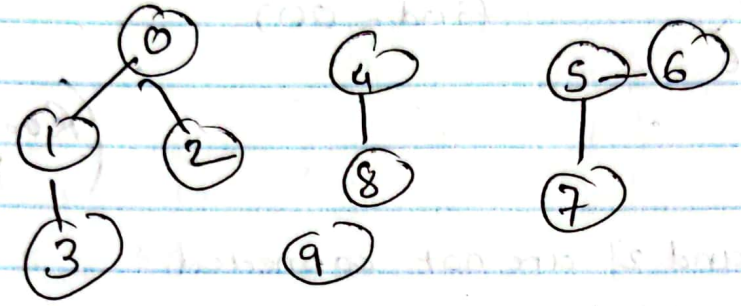
```
boolean dfs (int root, int target)
 {.
  SetcNode) visited;
  StackcNodes stack;
  add root to stack;
  while (stack is not empty)
   {
     Node cur = remove top element from stack;
     return true if cur is target;
     forcNode next: neighbors of cur)
      {
        if (next is not in visited)
            {
              aeld next to visited;
              add next to stack;
            }
      }
  return false;
}
```

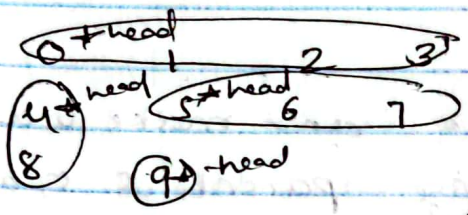
Graphs

- a non linear data structure consisting of vertices & edges.

Disjoint set / Union find

- used to find connectivity blw components of a network





Union
(0,1) (0,2) (0,3)
(4,8) (5,6) (5,7).

Check if connected -
(0,3) (4,5) (7,6)
same head ✗ ✗

1) Find: finds root of given vertex. eg- root of 6 is 5

2) Union: unions two vertices and makes their root nodes sa

2 ways to implement disjoint set

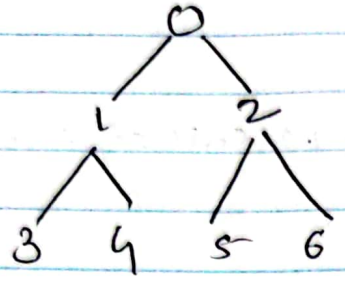
Quick find
- find faster O(1)
union - O(N)

Quick Union
Union faster. — O(N)-worst case.
find takes more time - O(N).
- more efficient than Quick Find

Worst case connect n elements
$O(N^2)$ —— $\leq O(N^2)$

Quick find

- store root nodes directly.



Array val	0	0	0	0	0	0	
Array index	0	1	2	3	4	5	6

find - O(1)

(Root stored directly in the array)

Union -

suppose 0 and 2 are not connected.

Parent(0) = 0

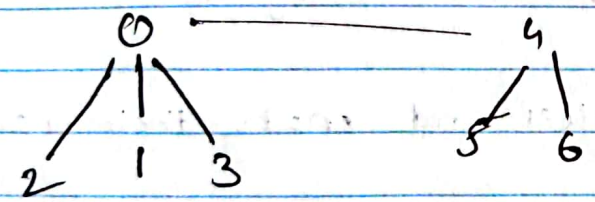
Par(2) = 2

now we connect 0-2 and make 0 as parent then we have to change parents of 5, 6 and so on.

O(N)

Quick Union

(We need to traverse to find root)



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

(find parent & connect them)

Array val	0	0	0	0	4	4	
ind	0	1	2	3	4	5	6

Space - O(n)

Construct	Find	Union	Connected
O(n)	O(1)	O(n)	O(1)

Quick find

```
int find(x)
    return root[x]
```

```
void union(x, y)
rootx = find(x)
rooty = find(y)
if (rootx != rooty)
    for (i in 0 root.length)
        if root[i]==rooty
            root[i]=rootx;
```

```
connected
return find(x) == find(y)
```

more efficient

Space - O(n)

O(n)	O(n)	O(n)	O(n)

Quick union.

```
int find(x).
if x == root[x];
    return x;
return root[x] = Find(root[x]
```

```
void union(x,y)
rootx = find(x)
rooty = find(y)
if rootx ≠ rooty
    root[rooty] = x;
```

Union by Rank

- Choose root of taller tree.
- is an optimization for quick union

find - O(logN)	O(logN)	
Union - O(logN)	O(logN)	
Connected - O(logN)	O(logN).	

Path Compression

- update root nodes of all traversed elements
- so next time u only have to travel 2 nodes.
- optimized find function

Why is quick union more eff?

Quick find - O(N) - union

Quick Union - find operation - worst case O(n).

So to connect n elements

Q. find - $O(n^2)$

Union $\leq O(n^2)$

Code with path compression and Union by rank

```
//rank[] = 1, root[i] = i.
find(x)
{
  if (root[x] == x) return x;
  return root[x] = find(root[x]);
}

Union(int x, int y)
{ rootX = find(x)
  rootY = find(y)
  if (rootX != rootY)
  {
    if (rank[rootX] == rank[rootY]
      {
        rank[rootX]++;
        root[rootY] = rootX;
      }
    else if (rank[rootX] > rank[rootY]
      {
        root[rootY] = rootX;
      }
    else
      {
        root[rootX] = rootY;
      }
```

Union find Constructer	find	Union	Connected
O(N)	$O(\alpha(N))$	$O(\alpha(N))$	$O(\alpha(N))$

$O(\alpha(N)) \Rightarrow O(1)$ on average.

Space - O(n)

Minimum Spanning Tree (MST)

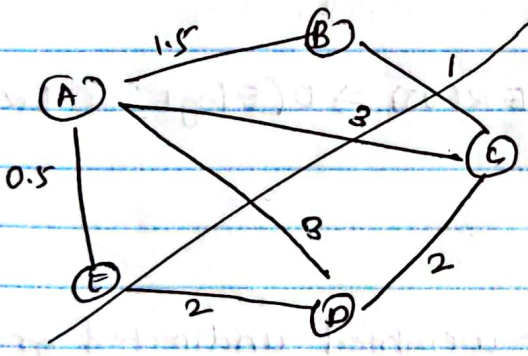
Spanning tree → is a connected subgraph in an undirected graph where all vertices are connected with minimum number of edges.

- An undirected graph can have multiple spanning trees.

MST - min cost spanning tree

Cut → is a partition of vertices in a graph into two disjoint subsets.

A crossing edge → is an edge that connects a vertex in one set with a vertex in other set.



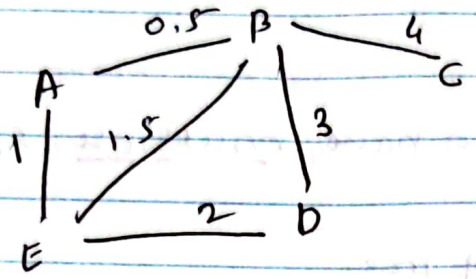
(A, B, E) (C, D) → disjoint subsets

AC, BC, AD, ED.
↳ crossing edges.

Cut Property:
For any cut C of the graph, if the weight of an edge E in the cut-set of C is strictly smaller than the weights of all other edges of cut-set of C, then this edge belongs to all MSTs of the graph.

Kruskal's algorithm – MST. of weighted undirected graph

algo



AB	0.5
AE	1
BE	1.5
ED	2
BD	3
BC	4

MST.

→ [AB, AE, ED, BC]

n-1 = 5-1 = 4.

1) Ascending sort all edges by their weight.
2) Add edges in that order in MST. Skip edges that would produce cycles in MST.
3. Repeat step 2 until n-1 edges are added.

Time – O(ElogE) – E-num edges.

1) sort – O(elogE)
2) Check edges – O(Eα(V)) ⇒ O(ElogE + Eα(V) = O(ElogE).

Space – O(V).

Prims Algorithm – MST of weighted undirected graph

2 sets – visited, unvisited

At each step take one vertex from unvisited, add it to visited & select min (weight) for edge going out of all visited edges

visited	A	AB	ABE	ABED	A,B,C,DE
non visited	BCDE	CDE	CD	C	-
edge	AB	AE	ED	BC	

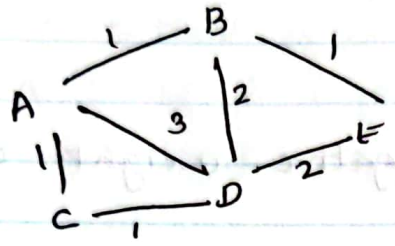
Time O(ElogV)

Space O(V).

Single Source shortest Path Algorithm

Shortest path in a weighted graph.

Edge relaxation:

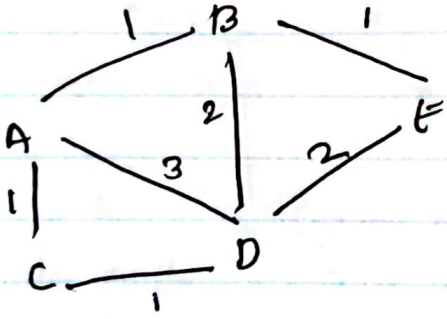


A → B 1
A → C 1
A → D ~~3~~ But A C-CD = 2 is shorter
A → E ∞. (no connection directly)

This is relaxation.

Dijkstra's - (only in graph with non negative weights).
Bellman ford (weighted directed graph along with ~~non~~ negative weights).

Dijkstra's - uses a greedy approach.
We take a starting point u as the center and gradually expand outward while updating shortest path to reach other vertices.



	Target vertex	Shortest dist	Prev vertex
1	(A)	0	-
3	(B)	1	A
2	(C)	1	A
4	(D)	2	C
	E	2	B

Select next vertex with min cost.

does not work for negative weights.

Time - $O(E + V \log V)$.

Space - $O(V)$.

Bellman Ford

- handles negative weight graphs.

Theorem1: In a graph with no negative-weight cycle, with N vertices, the shortest path between any vertices has atmost N-1 edges.

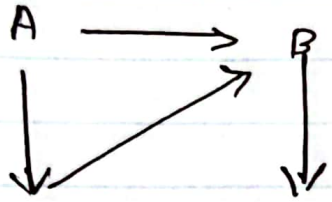
Theorem2: In a graph with negative weight cycle there is no shortest path

Kahn's Algorithm for topological sorting

course A ⟶ B ⟶ C ⟶ D

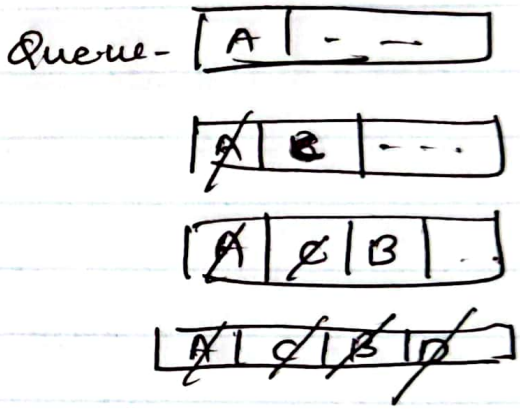
for D, we must first complete C, for C → B and so on.

Given vertices u and v to reach vertex v, we must have reached u first. In topological sorting u has to appear before v in the ordering.



A has no prereq.
B requires A, C.
C req. A
D req B, C.

Topological order - [A, C, B, D]



In degree - edges in
out degree - edges out

Calculate in degree for all nodes.
Add node with 0 in degree to queue
Poll from queue and mark that node visited
Subtract 1 from all nodes who have A as prereq
If indegree becomes 0 add it to queue
Repeat

Time - O(V.E). } without adjacency list
Iterate over E prereq for V nodes.
Space - O(V) - store indegree

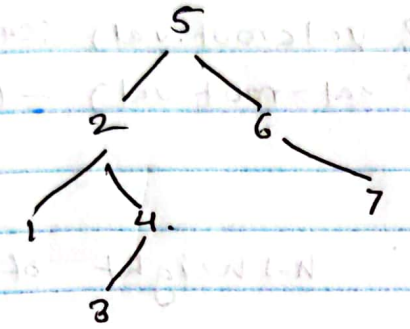
For a cyclic graph, there is no topological sorting.
Works only for directed acyclic graphs.

Time - O(V+E) Space - O(E) → using adj list.

Binary Search Tree

- A special form of binary tree
- The value in each node must be greater than (or equal to) any values in its left subtree but less than or equal to any values in right subtree.

eg.



★ Inorder traversal in BST will be in ascending order.

Search - Iterative

```
class Solution{
    public TreeNode searchBST(TreeNode root, int target)
    {
        TreeNode cur = root;
        while(cur!=null && cur.val!=target)
        {
            if(target < cur.val)
            {
                cur = cur.left;
            }
            else cur = cur.right;
        }
        return cur;
    }
```

Time - O(h) Space - O(1).

Search - recursive

```
searchBST( root, val)
 {
  if(root == null) return null;
  if(root.val == val) return root;

  if(root.left != null && val<root.val) searchBST( root.left, val)
  if(root.right != null && val>root.val) —u— (root.right, val)
 return null;
```

3.

Time O(h) space O(h) h → height of tree

Insert - Iterative Time - O(h) space O(1).

```
public TreeNode insert (Treenode root, int val).
 {
  TreeNode curr = root;
  if( root == null) return new TreeNode(val, null, null);

  while (true)
  {
    if (val > curr.~~right~~.val)
      {
        if (curr.right != null)
         { curr = curr.right;
         }
        else break;
      }
    else {
        if (curr.left != null)
          { curr = curr.left
          else break;
        }
  3.
  if (val > curr.val) curr.right = new treenode(val, null, null)
  else ——→ curr.left = ——u——
 return root
```

Insert - Recursive

```
insert {
   if(root == null) return new Treenode(val).

   if(val > root.val) root.right = insert(root.right, val);
   else root.left = insert( root.left, val);
  return root;
}
```

Time O(h) Space O(h)

Deletion. - Recursive

* Predecessor = before node
  ⇒ previous node or largest node before current one

* Successor = smallest node after current one
  = after node

case1 if node is leaf - delete

case2 - node has right child.
  replace node by its successor. Then recursively delete successor

case3 node has left child, no right child
  replace node by its predecessor. Then recursivelydelete predecessor

```
public int successor(Tree node root)   // one right, then leftmost
{
  root = root.right;
  while(root.left != null) root=root.left;
  return root.val;
}

predecessor                            // one left, then rightmost
{ root = root.left;
  while(root.right != null) root = root.right;
  return root.val
}
```

```
public TreeNode deleteNode ( root, key).
{.
  if(root ==null)  return null;

  //delete from right subtree
  if( key> root.val ) root.right = deleteNode (root.right, key).;
 else //delete from left subtree.
  else if (key<root.val) root.left = deleteNode(root.left,val);
  //delete the current node
  else {
     // node is leaf
   if(root.left ==null && root.right ==null)root =null;
   // node is not a leaf, has right child .
   else if (root.right! =null),
    {
      root.val =  successor(root)
      root.right = deleteNode(root.right, root.val);
    }
  // node is not leaf, no right child, has left child.
    else
    {
      root.val =  predecessor(root).
       root.left = deleteNode(root.left, root.val);
    }
  }

  return root;
  }
}
```

O(logn) O(h)
Time Space

Hashtable / Map / Dictionary

- ADT
- Provides access to data using key.
- key, value pair. (datatype don't have to match)
- Hash function maps keys to int
- In Java, hash fn is object.hashCode() // (can override)
- Collision - same hash value for ~~diff~~ same i/p

  Jane Jone → hash(Jone)

  Mike Jone → hash(Jone)

load factor: how full a hash table is

load factor = $\frac{\text{no. of items}}{\text{capacity}} = \frac{\text{size}}{\text{capacity}} = \frac{5}{10} = 0.5$

used to know when to resize the array backing the hash table

LF - too low → lot of empty space

- too high → collisions.

- can affect time complexity

☆ Linear probing } implement

☆ chaining (↑LinkedList)

DFS:

Uses-

- Traverse all vertices in a graph
- Traverse all paths b/w any two vertices in a graph

- O(V+E). - DFS.
- Space - O(V)

- Ideal DS - Stack

BFS

- Uses

- Traverse all vertices in a graph
- Find shortest path b/w 2 vertices, all edges have equal & positive weights.

O(V+E) - Time

Space - O(V)

Ideal DS - Queue

Recursion

Properties.

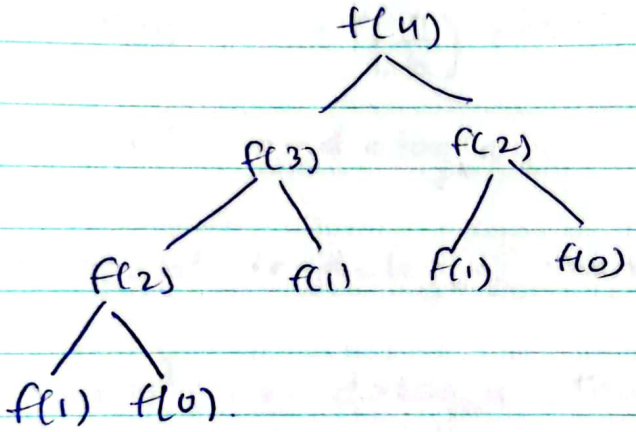
1. A base case - (or cases) - a terminating scenario that does not use recursion to produce an answer

2. A set of rules - or recurrence relation that reduces all other cases towards the base case.

Memoization

- to avoid duplicate calculations we store intermediate results

eg. Fibonacci series




```
public class fibonacci{
    Hashmap<Integer, Integer> map = new Hashmap<>();

  private int fib(int N)
 { if(map.containskey(N))
        { return map.get(N);
         }
    else int result;
     if(N<2) return N;
     else result = fib(N-1) + fib(N-2);

     map.put(N, result);
    return result;
 }.
```

Time complexity.

- Given a recursion algorithm, its time complexity O(T) is typically the product of number of recursion invocations (R) and time complexity of calculation O(S)

$$O(T) = R * O(S)$$

Tail Recursion - recursive fn call is the final instruction in the recursion fn. And there should be only one recursive call in the function.

- does not require recursion space.

Divide and conquer.

Recursively breaking the problem down into two or more subproblems until these subproblems become simple enough to be solved directly. Then combine the results of these subproblems to form the final solution

- naturally implemented in the form of recursion.

Divide → conquer → combine

Master Theorem

$$T(n) = aT\left(\frac{n}{b}\right) + f(n).$$

If $a > b^d$ ie- $d < \log_b a$ then $T(n) = O(n^{\log_b a})$

If $a = b^d$ ie $d = \log_b a$ then $T(n) = O(n^d \log n) = O(n^{\log_b a} \log n)$

If $a < b^d$ i.e- $d > \log_b a$. then $T(n) = O(n^d)$.

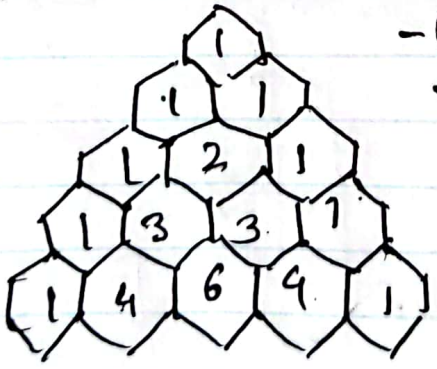
Recursion

function that calls itself.
base case - terminating case.
recurrence relation - reduces all other cases towards base case

eg - string reverse.

```
printReverse(char[] str)
{
    helper(0,str);
}

helper(int index, char[] str)
{
if(str==null || index>=str.length)
 return;

 helper(index+1, str);
 sysO(str[index]);
}
```

Pascal's Δ.



- leftmost & rightmost are always 1.
- for rest, each num is sum of two numbers directly above it in previous row.

$f(i,j) = 1$ where $j=1$ or $i=j$. // base case

$f(i,j) = f(i-1,j-1) + f(i-1,j)$ // recurrence reln

fibonacci - 0 1 1 2 3 5, 8, 13 - - - -

Time = no. of recursion iterations * complexity of calculations.

$O(T) = R * O(S)$.

Dynamic programming.

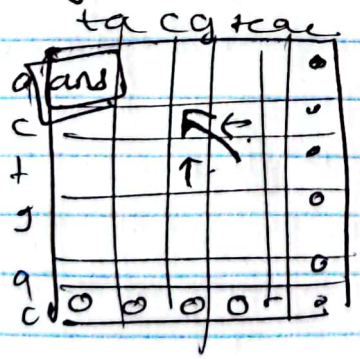
DP - overlapping subproblems.

Bottom up tabulation.	Top down memoization
- iteration from base cases	- recursion.
- fibonacci f(0) = 0 f(1) = 1 for i = 2 to n. f(i) = f(i-1) + f(i-2)	memo = hashmap function f(int i) if i == 0 || return i; if i is not in memo memo[i] = f(i-1) + f(i-2) return memo[i].
- runtime better. No overhead of recursion.	- easy to write

When to use DP

- max / min.
- decisions depend on earlier decisions.

longest common subsequence (str1, str2).




```
int[][] dp
for(int col = cStr2.length )
    for(row cstr1.length)
    {
    if (char at(row) = charat(col)
(bottomright+1) // dp[row][col] = 1+dp[row+1]
                                  [col+1
    else
dp[row][col] = max(dp[row+1,col] dp[row
    return dp[0][0].
```

Dynamic Programming

2 characteristics

1) overlapping subproblems - smaller version of original problem that are re-used multiple times.

2) optimal substructure - an optimal solution can be formed from optimal soln's to the overlapping subproblems of original problem.

2 ways to implement a DP problem

1) Bottom up (Tabulation). (Iteration)
starts at base cases.

eg.

```
f = array of len (n+1)
f[0] = 0
f[1] = 1
for i = 2 to n
  f[i] = f[i-1] + f[i-2]
```

2) Topdown (Recursion) (memoization)
- stores results.

```
memo = hashmap
Function F (Integer i);
  if i is 0 or 1:
    return i
  if i doesnt exist in memo:
    memo[i] = f(i-1) + f(i-2)
  return memo[i].
```

Which is better?

- Bottom up runtime faster. as iteration does not have overhead that recursion has.

- Top down is easy to write, because with recursion ordering of subproblems dont matters, whereas with tabulation we need to go through logical ordering of subproblems.

When to use DP

1) optimum value. (min/max/no. of ways of do something)
2) future decisions depend on earlier decisions.

framework for DP problems. (Ref- climbingstairs).
State → a variable that can sufficiently describe a scenario.

To solve a DP problem we need to combine 3 things -

1. A function or data structure that will compute/contain the answer to the problem for every given state.

eg: dp(i) → no. of ways to climb the ith step.
(literally original problem, but generalized for a given state)

2. A recurrence relation to transition between states.
   dp(i) = dp(i-1) + dp(i-2).
   - relates diff states with each other.

3. Base cases, so our recurrence relation doesnt go infinitely
   dp(1) = 1 dp(2) = 2.
   - which states can I find ans to without using DP?

```
class Solution{
  private int dp(int i){          // A fn that represents ans to the problem
                                  //  a given state.
    if(i<=2)return i;             //Base case
    return dp(i-1)+ dp(i-2);      // Recurrence reln.
  }
public int climbStairs(int n)
   {
     return dp(n)
   }
}
```

not a dp
no memoization

$O(2^n)$

I Best time to buy and sell a stock (atmost one transaction)

1) Two pointers l, r (l+1 to length), find max
2) Local min, max.

②

```
int min,
int maxprofit
for (i=0;i<len;i++)
{
  if(prices[i]<min)
    min=prices[i]
  else if(prices[i]-min>max)
    max=
return max
}
```

O(n)

①

```
for(i=0;i<len;i++)
{
  for(j=i+1;j<len;j++)
  {
    if(prices[j]-pri[i]>max)
      max=
  }
return max.
```

O(n²)

II. Buy and sell many times, maximize profit.

```
for(int i=1,i<len;i++)
{
  if(prices[i]>prices[i-1]// buy
  {
    maxprofit+=prices[i]-prices[i-1]
  }
}
return maxprofit
```

O(n)

III atmost 2 transactions.

```
buy1 = maxval buy2=maxval
p1=0, p2=0
for(i=0, price.len;i++)
{
  buy1= Math.min(buy1, prices[i])
  p1= Math.max(p1, prices[i]-buy1)
  buy2 = Math.min(buy2, prices[i]-p1)
  p2=Math.max(p2, prices[i]-buy2)
return p2;
```

O(n).

dp = new int [n][k+1][2]

O(nk)

IV at most k transactions. k max. sell hold 0 1

dp[day_number][used_transact_num][stockstatus].
prices.len.

1. keep holding dp[i][j][1] = dp[i-1][j][1]
2. not hold dp[i][j][0] = dp[i-1][j][0]
3. buy. j>0 dp[i][j][1] = dp[i-1][j-1][0] - prices[i].
4. sell dp[i][j][0] = dp[i-1][j][1] + prices[i]

dp[i][j][1] = max(dp[i-1][j][1], dp[i-1][j-1][0] - prices
dp[i][j][0] = max(dp[i-1][j][0], dp[i-1][j][1] + prices

V with cooldown, any no. of times.

```
int[] MP = new int[price.len+2]
for (i = price.len-1; i>=0; i--)
  {   int c1=0
    for(int sell = i+1; <price.len; sell++).
      {
        int profit = (prices[sell] - prices[i]) + MP[sell+2];
          c1 = Math.max(c1, profit).
      }
    MP[i] = Math.max(c1, MP[i+1])
  }
return MP[0];
```

O(n²).

Recursion.

1) letter combinations of a phone no.

```
map (2: abc 3: def ... ).
String phonedigits;
List<str> combinations = new ArrayList().

Public letterCombinations (String digits)
{ if(len==0) return;

  else{
    phonedigits = digits
    backtrack(0, index, new StringBuilder())
    return combinations;
  }
}

void backtrack(index, str)
{
  if( str.len == phonedigits.len()) {
    combinations.add(str).
    return
  }
Str possible = map.get(charAt(index))
for( char possiblelet: possible.toCharArray())
{
    str.append(possiblelet)
    backtrack( index+1, path)

    path.deleteCharAt(path.len()-1).
  }
}
```

"23"

O(4^N N)

2. Generate Parenthesis

```
generateParen(int n)
{
  List<Str> com = new ArrayList<>()
  generateAll(new char[2*n], 0, com);
  return com;
}

generateAll(char[] curr, pos, com);
{
  if (curr.length == pos)
    { if(valid(curr)
        { return com.add(new String(curr);
    }
  else {
    curr[pos] = '(';
    generateAll(curr, pos+1, com)
    curr[pos] = ')'
    generateAll(curr, pos+1, com)
  }

valid(curr)
{ bal = 0
  for(curr.len)
  if('(') bal++
  else bal--
  if bal = 0 true
    else false
}
```

$O(2^{2n} \cdot n)$ — sequences — valid.

3. Word search.

```
char[][] board.
rows, cols.

Public bool exist(board, word)
{
  for(i) rows
   {for(j) cols.
      {
       if(this.backtrack(row, col, word, 0))   index.
        return true;
      }
  ret false
}

backtrack(row, col, word, index)
{
  if(index >= word.len) true

  if(row<0, >rows, col<0, >cols, ||this.board[row][col] !=
                                          word.charAt(ind
  ret false

  bool ret = false
   board[r][c] = '#'
   //go in 4 directions.

   if ret break;

this.board[r][c] = word.charAt(index))
return ret;
```

3.

O(N 3^4). O(L)

Dynamic Programming

1. Coin change (coins[], amount) — Bottom up

```
{
    max = amount + 1;
    int[] dp = new int[max];
    Arrays.fill(dp, max)
    dp[0] = 0

    for(i = 1 to amount)
        for(each coin[j])
            {if(coin[j] <= i)
                {
                dp[i] = Math.min(dp[i], dp[i - coin[j]] + 1)
                }
    }
    return dp[amount] > amount ? -1 : dp[amount];
```

O(S * n) S = amount, n = coins.length

```
for(coin)
{ for(coin; amount, ++
  dp[a] += dp[a - coin]
}
```

2. word break (Str s, wordDict)

```
{
    Set<String> dict = new HashSet<>(WordDict);
bool dp[] = new boolean[s.len + 1]
    dp[0] = true
                                        O(n^3), O(n)
    for( i=0, i<=s.len; i++)
    {
        for(j=0, j<i; j++)
        {
            if(dp[j] && worddict.contains(s.substring(j, i))
                dp[i] = true;
                break.
            }
    }
```

Longest Increasing Subsequence.

```
int[] dp = new int[nums.len]
Arrays.fill(dp,1)

for(i=1, i<nums.len; i++)
    {
     for(j=0 ; j<i; j++)
       {
         if (nums[i] > nums[j]
             dp[i] = math.max(dp[i], dp[j]+1)
       }.

return max from dp array.   O(N²). O(N).
```

Maximum subarray. (int nums[])

```
int maxsubarray = Int.MinValue nums[0]
int cursubarray =   -u-  nums[0]

   for (i=1;  nums.length ; i++)
      {
         int num = nums[i]
         iCurrSubArr = Math.max( num, num+cursub
         MaxSub  -  -u- ( maxsub , cursub)

    }
  return maxsubarr
}
```

O(N), O(1).

Top down.

```
HashMap<Int, Int> memo = new HashMap<>();

private int dp(int i)
{
 if(i<=2) return i;
 if( !memo.containsKey(i)) {
     memo.put(i, dp(i-1)+dp(i-2));
 }
 return memo.get(i).
}

public int climb(int n)
 { return dp(n)

 }.
```

T = O(n). S = O(n)

Bottom up.

```
public int climbStairs(int n) {
    if(n==1) return 1;

    int[]dp = new int[n+1]
    dp[0] = 1
    dp[2] = 2

    for(i=3 to n)
     {
      dp[i] = dp[i-1] + dp[i-2]
     }
    return dp[n]
}
```

```
def getMinKeyboardPress(keyword):
    freq = {}
    count = 0;

    for i in keyword:
        if i in freq:
            freq[i] += 1
        else:
            freq[i] = 1

    for i in range(0, a):
        if bool(freq):
            max_key = max(freq, key=freq.get)
            occur = freq[max_key]
            freq.pop(max_key)
            count += i * occur
        else:
            break
        2x occur
        0, 1, 3x occur.
    return count
```

```
    freq = {}
    freq_final = {}

    for i in keyword:
        if i in freq:
            freq[i] += 1
        else:
            freq[i] = 1
            freq_final[i] = 0

    for i in longstring:
        if i in freq_final:
            freq_final[i] += 1

    for k, v in freq_final.items():
        freq_final[k] = v / freq[k]

    min_key = min(freq, key=freq
    return freq_final[min_key]
```

Spanning tree - tree, connects all vertices with least no. of edges
MST - min weight edges.

Kruskal -
sparse
easy
Unionfind.
Sort edges on weight,
add if not cycle,
$E \log V$

Prim.
dense.
PQ
- add one node CP
- add node w/ th shorest dist fr

$- E + V \log V$.

Dijkstras.
shortest path from s to all V.
$O((V+E) \log V)$

Bellman
$O(VE)$.
negative edges.

Select, bubble, ins - $O(n^2)$.
merge - $n \log n$.
quick - $O(n^2)$
heapsort ($n \log n$)

Linears - $O(n)$
Bin - $O(\log n)$.

Heap
Create - $O(n \log n)$
delete, minmax - $O(1)$
decr - $O(\log n)$
Ins - $O(\log n)$

Comparable & Comparator

Comparable - is an interface defining a strategy of comparing an object with other objects of same type. This is called class's "natural ordering".

In order to sort, you have to declare that object as comparable, and then use compareTo() method.

```
public class Player implements Comparable<Player>
{
@Override.
  public int compareTo(Player otherplayer){
      return Integer.compare(getRanking(), otherplayer.get
                                                Ranking()
  }.
}.
```

Integer.compare(x, y) → return

-1.	$x < y$
0	$x = 0$
1	$x > y$.

Comparator - The comparator interface defines a compare(arg1, arg2) method with two arguments that represent compared objects, and works similar to Comparable.compareTo() method.

```
public class PlayerAgeComparator implements Comparator<Playe
{
  @Override
  public int compare(Player one, Player two){
      return Integer.compare(one.getage(), two.getage());
}  }
```

How to use -

1) create an instance of comparator
2) use in sort.

eg → PlayerAgeComparator pagecomp = new PlayerAgeComparator();
Collections.sort(footballteam, pagecomp)

Java 8 Comparators

lambda exp & comparing() static factory method.

eg.

```
Comparator byRanking =
(Player player1, Player player2) -> Integer.compare(
        player1.getRanking(), player2.getRanking()
```

The Comparator.comparing method takes a method calculating property that will be used for comparing items and returns a matching comparator instance.

```
Comparator<Player> byRanking =
        Comparator.comparing(Player::getRanking);
```

Comparator vs comparable

comparable – good for defining default ordering.

So when to use Comparator-?

Sometimes we cant modify source code of the class whose objects we want to sort and cant use comparable. Comparators allows to avoid adding additional code to our domain classes.

We can define multiple different comparison strategies, which isnt possible using comparable

Subtraction trick – avoid → (Integer Overflows)

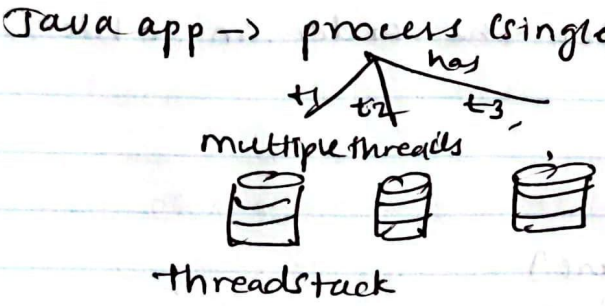
```
Comparator<Player> comp = (p1, p2) -> p1.getRank() - p2.getRa
```

Concurrency & Threads.

Process: a unit of execution.
Interchangably called application.

Each java app has its own memory space - HEAP.

Thread: a unit of execution within a process.
- Every process has atleast one thread - main thread
- may have multiple system threads.
- Every thread created by a process shares the process's memory & files *
- Thread has thread stack.



* Why multiple threads?
- long running task will run in background, no freezing
- api requires us to provide one.

Concurrency: one task doesn't have to complete before another can start.

Creation & execution

We can't guarantee the order in which threads will execute

1) Extend Thread class., override run method.

```
public class Main extends Thread {
    public void run() {
        SOP("new thread")
    }
```

execution
- create instance.
- call start() metho

2) Runnable interface

```
public class Main implements Runnable {
    public void run()
    { sop("Hi")
    }
}
```

How to run -

pass an instance of class to Thread obj constructor and then call thread's start method.

```
Thread th = new Thread(obj);
th.start();
```

* We cant guarantee the order in which threads will execute.

Just for fun

```
new Thread() {
    public void run()
    {
        sop("Hello");
    }
}.start();
```

setName("...xyz");
getName()
Sleep(3000); ms./ms,ns.

Interrupt: stop a thread from doing what it is doing to d
something else.

Two checks:
1) exception
2) periodic checks for interruption.
th.interrupt();

* join() → to terminate another
* sleep(3000); it waits for 3000 sec or return.

Threads share heap memory, thread stack is not shared

* Synchronized - only one thread can access at a time

(dont synchronize on local variables)

Threadsafe => all critical sections are synchronized

wait(); - release all locks, wait
notifyAll();
notify().

Thread can be suspended in middle unless
the operation is atomic.

Arraylist is not threadsafe - no race conditions.
Vector is threadsafe.

Multithreading - multiple threads of execution inside the same application.

Concurrency problem: Bcoz threads run at same time as other parts of program, there is no way in which order the code will run. When the threads and main program are reading & writing same var, the values are unpredictable. The problem that result from this are concurrency problem

Stop a thread. → thread.doStop()

Critical section → section of code that is executed by multiple threads where the sequence of execution of the threads makes a difference in the result of concurrent execution of critical section.

Race condition → is a concurrency problem that may occur inside a critical section.
1) Read - modify - write read 0 add 2 read 0 add 2 count = 2 instead of 3
2) check - then - act (eg - map - check & remove)

Preventing race condition -
1) ensure critical section is executed as an atomic code (synchronized block of Java code).
2) use locks, etc.

Synchronized blocks - thread safe
- can only be executed by a single thread at a time. thus avoid race condition.

limitation- allows only one thread at a time.
what if 2 threads only want to read? we can allow.

Thread signalling
wait()
notify()
notifyAll().

Deadlock - a → b → c.

Prevention-
1) lock ordering → make sure all locks are always taken in the same order by any thread.
2) lock timeout
3) Deadlock detection.
check if it has any locks required for other threads that is waiting for

Locks - thread synchronization mechanism.
Lock lock = new Lock()
lock.lock() lock.unlock();

Semaphore → thread synchronization construct that can be used either to send signals b/w threads to avoid missed signals or to guard critical section like you would with a lock.
take() → notify()
release → wait()
Signal.

Thread pools

Java Comparator

```
class NodeComparator implements Comparator<ListNode>
{
    public int compare(ListNode k1, ListNode k2)
    {
        if(k1.val > k2.val)
            return 1;
        else if (k1.val < k2.val)
            return -1;
        return 0;
    }
}
```

data type of obj

if doesn't work use

```
return Integer.compare(a,b)
```

```
PriorityQueue<ListNode> q = new PriorityQueue<>(new NodeComparator());
```

```
Collections.reverse(List)      - list reverse
Collections.reverseOrder()     - PQ in order
PQ<Integer> pq = new PQ<>(Collections.reverseOrder());
```

```
Char c.
  c - 'a'  = 0.    if c = a
  c - 'a'  = 1     if c = b
```

2d array len.

```
{0,1,2,3}
{1, 2, 3, 4, 5}
{1, 2}
arr[0].length = 4
arr[1].length = 5
arr[2].length = 2.
```

```
Arrays.copyOf(points,k)  -> return k elements from points array
Arrays.sort(points, (a,b) -> squared(a) - squared(b));
```

Comparable & Comparator

Java.util.Arrays.sort(arr); ↑order

for ↓ → collections.reverse(arr); after sorting

For object array. (eg-employee)

```
public class Employee implements Comparable<Employee> {
@Override
    public int compareTo (Employee employee)
        {
            return this.salary - employee.salary;
        }
}
```

compareTo. → returns x → {1, -1, 0}

obj1 > obj2 if x > 0

obj1 < obj2 if x < 0.

obj1 = obj2 if x = 0.

* Arrays.sort (employees., new comparator<Employee>()

```
{
  @Override
    public int compare( Employee e1,  e2)
      {
          return   e1.sal > e2.sal);
      }
};
```

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

Arrays.sort(intervals, (a,b) → Integer.compare(a[0], b[0]));

lamba functions - do not need name, can be implemented right in body of method.
parameter (optional), arrow token, body.

(a,b) → Integer.compare(a[0], b[0]);

parameter → expression
(p1, p2) → ——"——

Arrays.copyOf(input, 0, k)
——"—— (input, k, input.length)

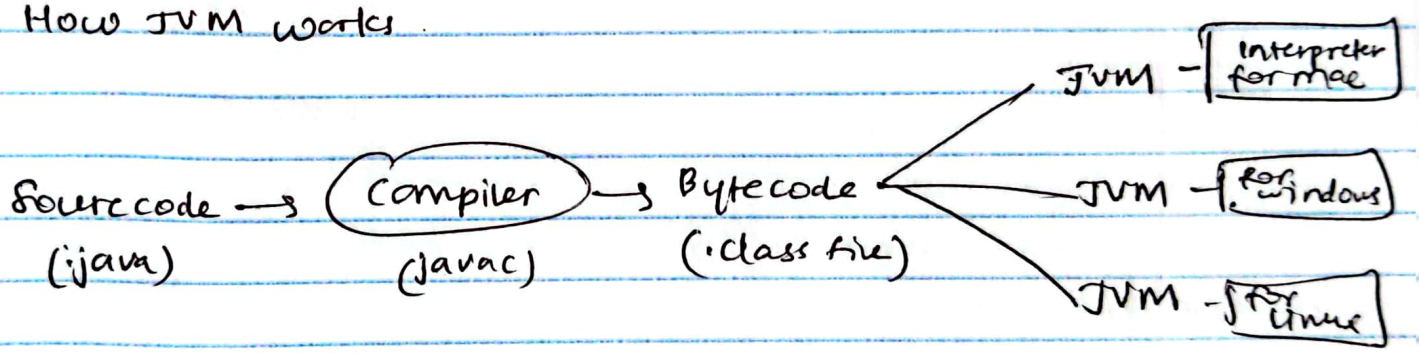
beginnersbook.com .

JDK – contains JRE, JVM, compiler, Java app launcher, Appletviewer etc.

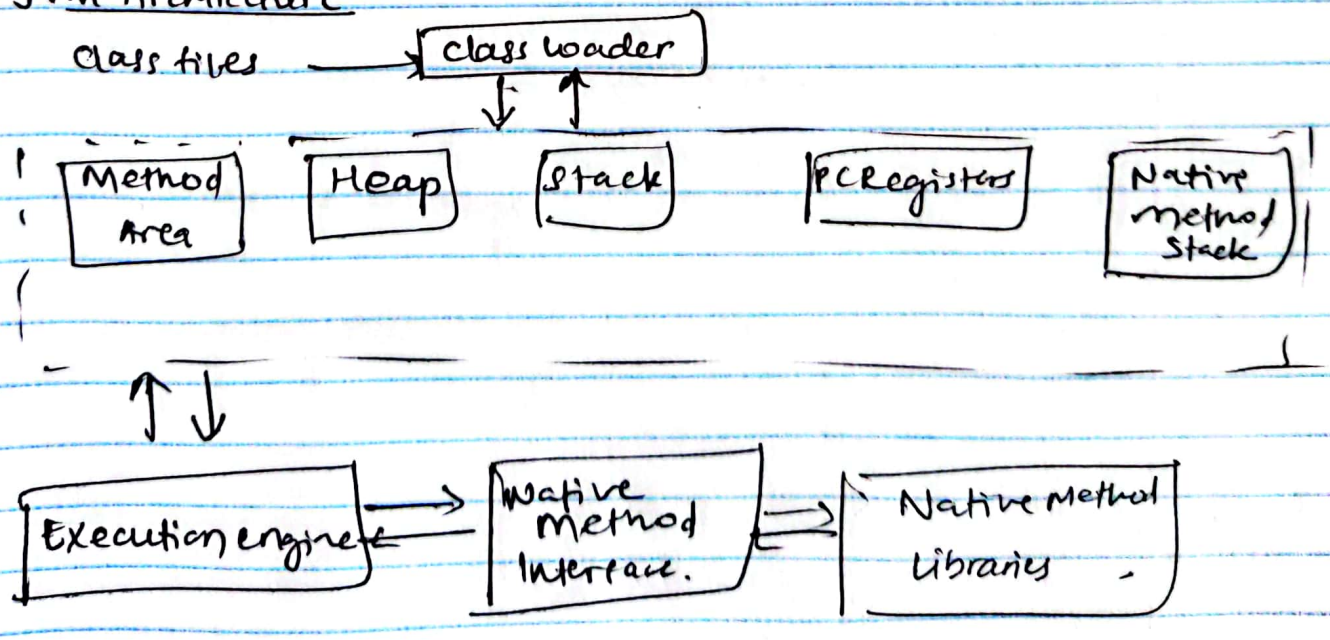
JRE - env on which jvm runs.
- contains JVM, class libraries, compiler, debugger.

JVM - makes java platform independent.
- its a virtual machine residing in your real machine and machine langfor JVM is bytecode.
- makes it easy for compiler as it has to generate bytecode for JVM rather than diff machine code for each machine

How JVM works



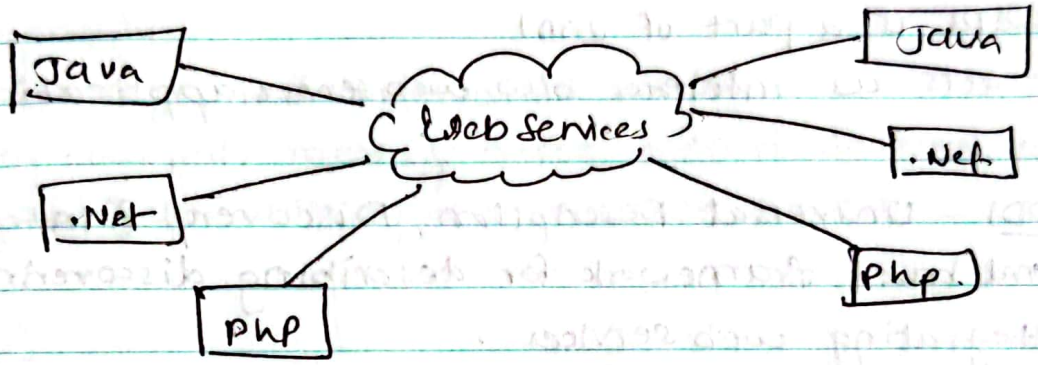
JVM Architechure



Web Services

Webservice -

- a client server application or application component for communication.
- the method of communication blw two devices over a network.
- provides a common platform for various applications written in diff. languages to communicate with each other over network.



Types

1) REST 2) SOAP.

features

- XML based
- loosely coupled
- Coarse grained
- Ability to be synchronous or asynchronous
- Support RPC
- Support Document exchange

Web Service components

1) SOAP
2) WSDL
3) UDDI

SOAP - Simple Object Access Protocol.
- XML based protocol for accessing web services.
- W3C recommendation for communication blw apps.
- Platform & language independent.

WSDL - Web Services Description Language.
- XML doc containing info about web services such as method name, method parameter and how to access it.
- WSDL is a part of UDDI
- It acts as interface blw web service applications.

UDDI - Universal Description, Discovery & Integration.
- XML based framework for describing, discovering and integrating web services.
- is a directory of web service interfaces described by WSDL, containing info about web services.

SOAP	REST
- is a protocol	is an architectural style.
- cant use REST as its a protocol	- REST can use SOAP because it's a concept & can use any protocol like HTTP, SOAP.
- uses services interfaces to expose the business logic	- uses URI to expose business logic
- standards to be strictly followed	- REST does not define much standards like SOAP.
- requires more bandwidth & resource.	- less bandwidth & resource.
- less preferred	- more profered
- permits ~~XLM~~ XML data format only	- allows xml, html, json etc.

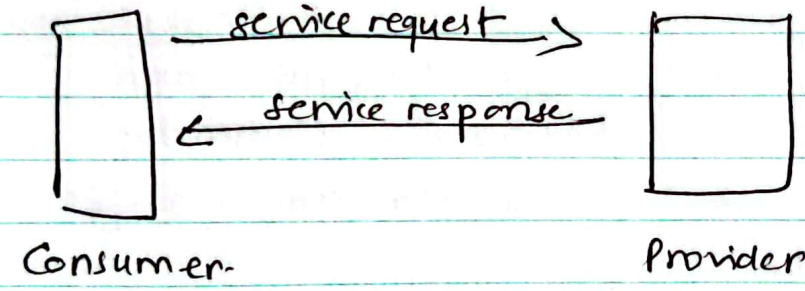
Service Oriented Architecture (SOA)

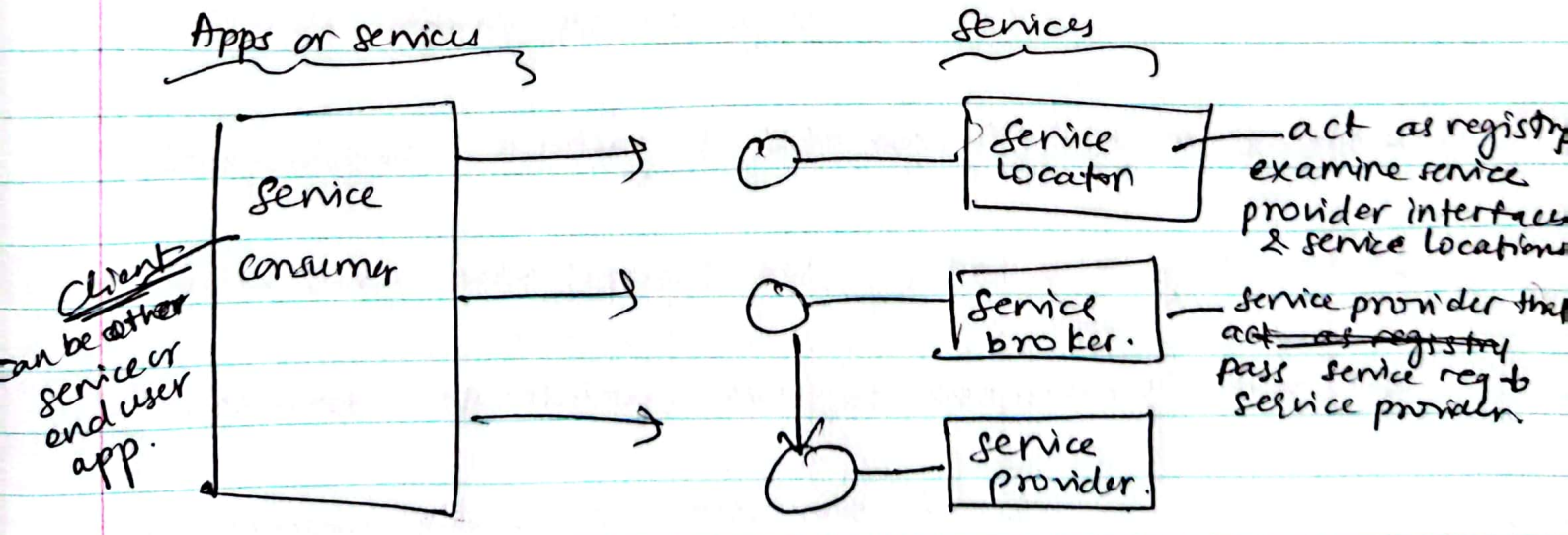
- is a design pattern which is designed to build distributed systems that deliver services to other applications through the protocol.
- is a concept only and not limited to only PL or platform.

Service.

- a well defined, self contained function that represents a unit of functionality
- can exchange info with other service.
- loosely coupled, message based communication model to communicate with apps and other services.

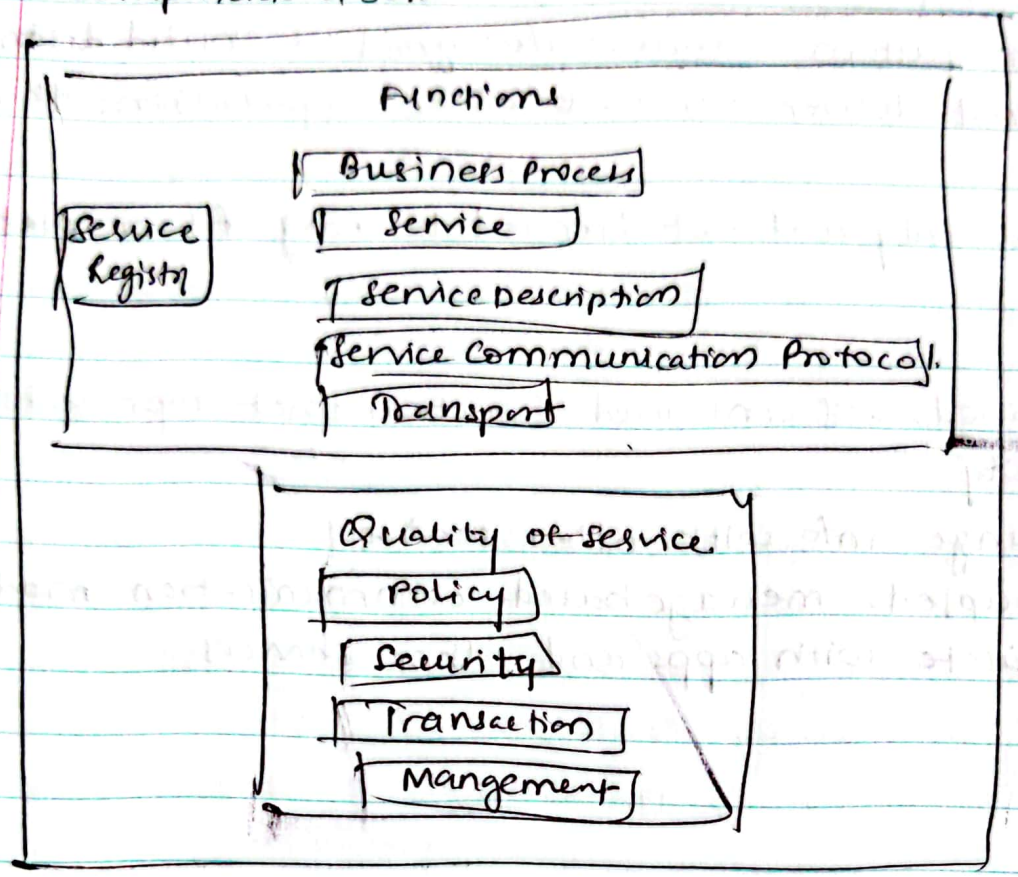
SOA





service provider - software entity that implements a service specification.

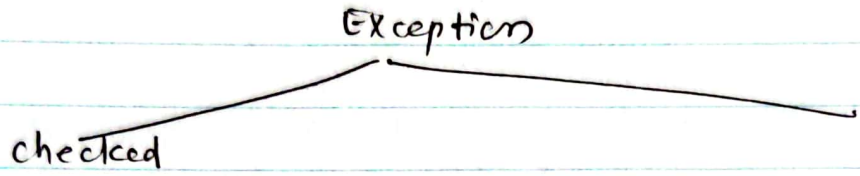
Components of SOA



Java Exceptions

Exception: problem in normal flow of execution. If the exception cannot be handled, then the execution gets terminated before it completes the task.

Exceptions are subclass of java.lang.Exception.



checked

are checked by compiler at the time of compilation.
- classes that extend Throwable class except runtime & error are called checked exception.
- must either declare exception using throws or try/catch.
eg - ClassNotFoundException

Unchecked.

not checked by compiler. The compiler doesn't force to handle this exceptions
eg. Arithmetic exception
ArrayIndexOutOfBounds.

try – put risky code here.
catch – catches exceptions.
finally - executed everytime, cleanup codes.

Exception propagation

- first thrown from method which is at top of stack, then if it doesn't catch it pops up method and moves to previous method and so on until caught.

Thread – join() → join one thread with end of currently running thread

heights[]
int arr[] = heights.clone(). → copies heights into arr / clone

nums[]
Collections.min(nums) // min of array
Collections.max(nums) // max of array
System.arraycopy(from, index_from, to, index_to, no. of elements)

Arrays.asList(s.split("\\s+"));
Collections.reverse(list);

Pair - javafx.util.
Pair<Integer, String> pair = new Pair<>(1, "one");
Integer key = pair.getKey()
String val = pair.getValue()

set1.retainAll(set2) → intersection of sets.

☆ String arr[] = new String[list.size()]
list.toArray(arr)

map.getOrDefault(a+b, 0)+1);

Queue<Integer> q = new PQ<>((n1,n2) -> count.get(n1) - count.ge

Map.keySet()

Arrays.sort(intervals, (a,b) -> (a[0] - b[0]));

PQ<int[]> pq = new PQ<>((a,b) -> (a[1] - b[1]));

set1.retainAll(set2)

```
Arrays.sort (logs, new Comparator<int[]>(){
  @Override
    public int compare (int[] log1, int[] log2){
      Integer tsp1 = new Integer (log1[0])
      —u tsp2 — - —u— (log2[0]);
      return tsp1.compareTo(tsp2);
    }.
  });.
```

☆ enum Color {GRAY, BLACK}.

use → return state.Color = Color.GRAY

hashmap

this.map.forEach((key,value) → Collections.sort(value));

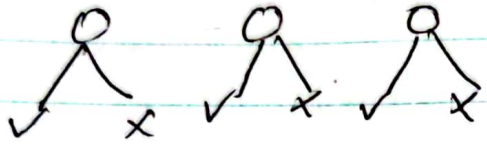
DP - Identify.

1) Recursion - Enhanced Recursion, overlapping.

2) optimal substructure

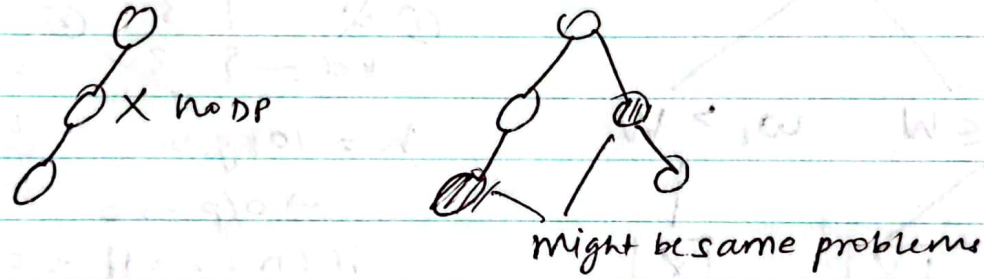
Recursion / DP.

1) Choice → to include or not include ⇒ Recursive



Recursion + overlap → DP

If there is a recursive fn and 2 calls, there is a probability of



2) optimal - min, max, largest, smallest

Recursive → memoization → top down.

↓

1) Choice diagram
2) Base condn
3) Smaller I/P

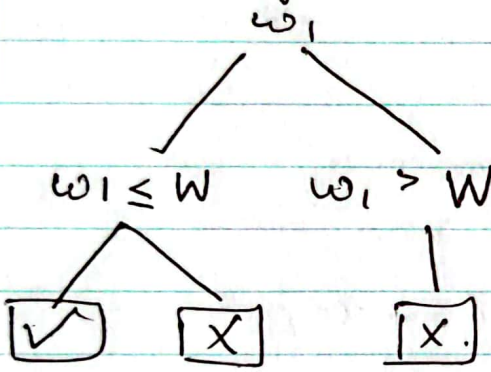
} Imp

Knapsack Problem.

- Fractional (Greedy).
- 0-1 (1 instance)
- Unbounded. (Unlimited supply of items).

weight[] [1 3 4 5]
val [] [1 4 5 7.]
W = 7kg
o/p - (max) profit

Choice diagram



Base condition

Think of smallest valid I/P.

① w = { } val = { }
② w = [1, 2] val = [2, 3]
W = 10kg. W = 0kg
→ o/p → 0.

if (n == 0 || w == 0) return 0;

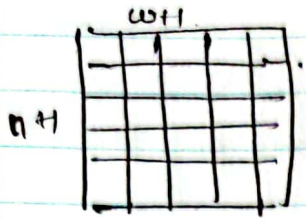
★ Recursive fn → call on smaller inputs everytime n → n-1 → n-2 ...

int t[][] = new int [n+1][w+1] - for memoization
Arrays.fill(t, -1);

```
int knapsack (int wt[], int val[], int w, int n)   // n: no. of el.
{
    if (n==0 || w==0) return 0;        // if (t[n][w] != -1) return t[n][w]
    if (wt[n-1] <= w)
        return max (val[n-1] + knapsack(wt, val, w-wt[n-1], n-1),   // t[n][w] =
                    knapsack(wt, val, w, n-1));
    else if (wt[n-1] > w)
        return knapsack(wt, val, w, n-1);   // t[n][w] =
}
```

★ How to choose var for memoization? → var which change
n, W.

Top down approach.



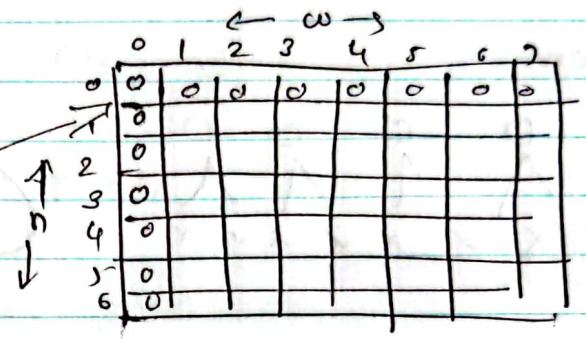
Use only table, no recursion.

Step 1- Initialize.

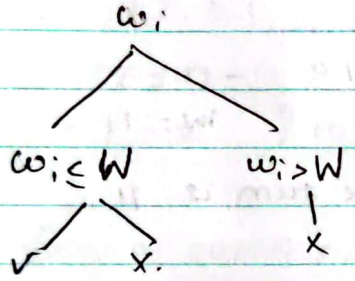
Step 2: Recursive to iterative.

1) Recursive

```
if (n == 0 || w == 0)
    return 0
```



2)




```
if (wt[n-1] <= W)
return
 max(val[n-1] + knapsack(wt,val,W-wt[n-1],n-1),
                 knapsack(wt,val,W,n-1))

else if (wt[n-1] > W)
    return knapsack(wt, val, W, n-1)
```

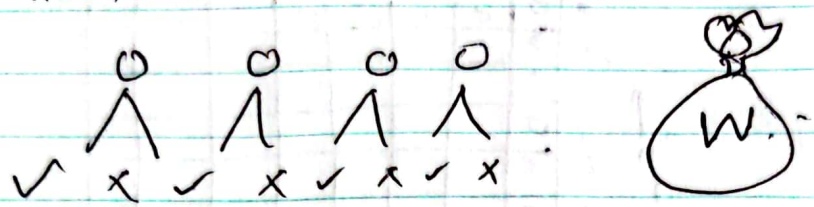
```
if (wt[n-1] <= W)
t[n][w] =
 max( val[n-1] +
        t[n-1][w-wt
 , t[w][n-1])

else.
t[n][w] = t[n-1][w]
```

```
code
int row 0, colo to 0.
for (i = 1 to n+1)
    for (j = 1 to W+1)
    { if (wt[i-1] <= j)
        t[i][j] = max ((val[i-1] + t[i-1][j-wt[i-1]]),
                        t[i-1][j])
      else t[i][j] = t[i-1][j]
} return t[n][w]
```

Based on knapsack -

1) Subset sum
2) Equal sum partition
3) Count of subset sum
4) Minimum subset sum diff-
5) Target sum
6) no. of subsets of given diff.

Pattern



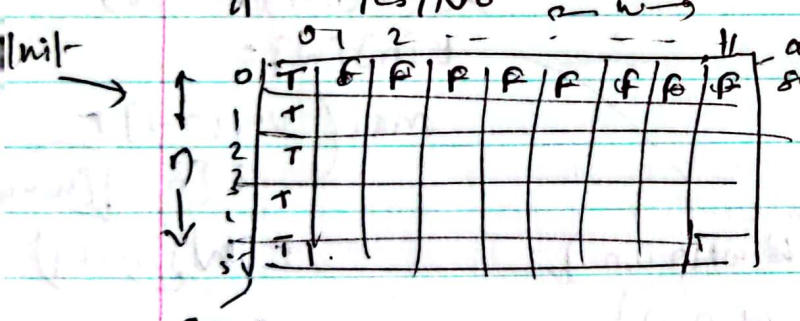
Subset sum

I/p arr[] : 2 3 7 8 10 — n = 5

sum : 11 W = 11

Q. find if arr contains a subset whose sum is 11.

O/p - Yes/No



s = 0.

arr = { }

O/p = { }

knapsack

```
if(wt[i-1] <= j)
  t[i][j] = max(val[i-1] + t[i-1][j-wt[i-1]],
                t[i-1][j])
else
  t[i][j] = t[i-1][j]
```

Subset sum

```
if(arr[i-1] <= j)
{t[i][j] = t[i][j-arr[i-1]]
}      || t[i-1][j]
else
  t[i][j] = t[i-1][j]
return t[n][sum]
                 W.
```

Equal sum Partition.

arr[]

o/p : T/F

Divide arr into 2 subsets with equal sum. Possible or not?

```
for(int i=0, i<=arr.size; i++)
    sum+ = arr[i];

if(sum%2 != 0) return false.

else subset_sum(arr, sum/2);
```

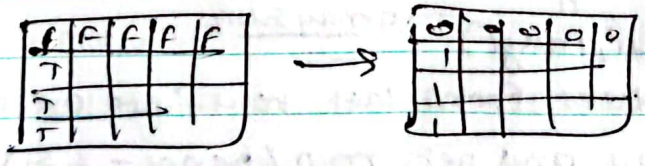
Count of subsets with a given sum.

arr = [2, 3, 5, 6, 8, 10]

sum = 10

o/p = 3. {10}, {8, 2} {5, 3, 2}.

Same as subset sum except

1) 

2) t[i][j] =

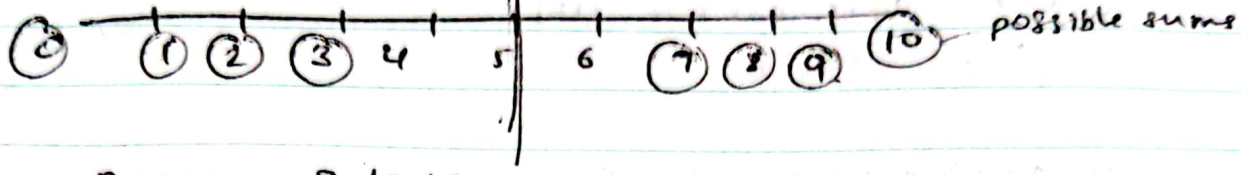
```
if(arr[i-1] <= sum)
    t[i][j] = t[i-1][j] + t[i-1][j - arr[i]]

else
    t[i][j] = t[i-1][j].

return t[n+1][sum+1].
```

* Minimum subset sum difference.

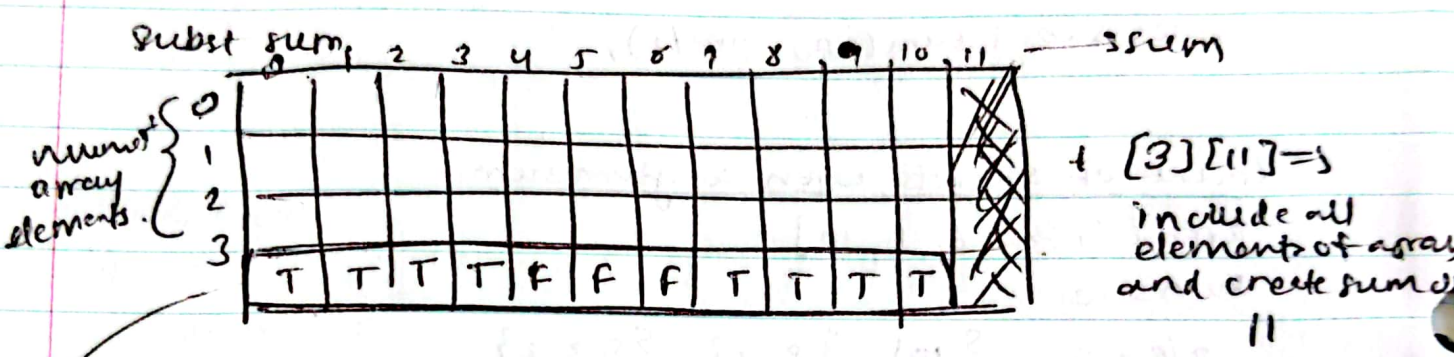
I/P arr = [ ]

o/p min(S1 - S2) Min diff b/w two subsets of array

{1, 2, 7} 7 + 2 + 1 = 10 = sum of arr.



Range : 0 to 10.



S1 + S2 (S1 - smaller S1 ≤ S2)

S1 + (Range - S1) * => Range - 2S1 → minimize.

1) Call subset sum(arr, range) array sum

2) Take all True values from last row. into an array (ans)

3) Traverse (ans) array and get min (Range - 2S1)

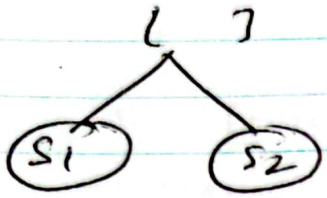
```
int min = Int.max
for (i=0, i<ans.size(); i++)
    {
    min = Math.min(min, Range - 2 * ans[i])
    }
return min.
```

* Count num of subsets with given diff in sum

arr [] {1, 1, 2, 3}

Diff = 1.

o/p = 3 (count)



diff

$S_1 - S_2 = diff$

$S_1 + S_2 = sum(arr)$

$2S_1 = diff + sum(arr)$

$S_1 = \frac{diff + sum(arr)}{2} = \frac{1+7}{2} = 4$

count = ? how many with S_1 as sum

* count no of subsets sum problem.

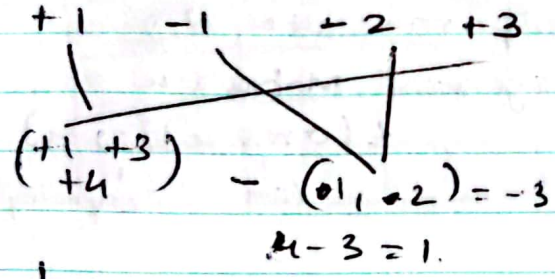
⇒ return (countofSubsetSum(arr, S_1);

Target sum.

arr = { 1 1 2 3 }

sum = 1. o/p = 3.

Add +/- infront of every element of array to get desired sum. Return how many combinations possible



↓

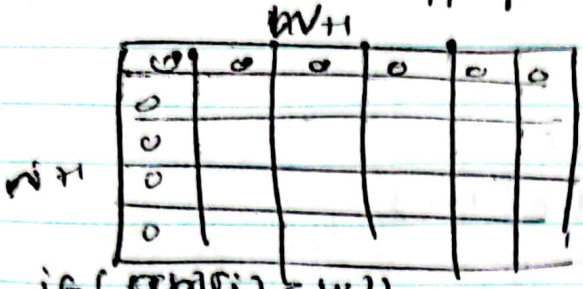
Subset sum diff. problem.

(arr, sum) = (arr, diff)

Target sum Subset sum with given diff.

Unbounded Knapsack

* unlimited no. of supply for items.



if (wt[i] ≤ w))

$$t[i][j] = \max\left(\begin{matrix} val[i-1] + t[\underline{i}][j] \\ t[i-1][j] \end{matrix}, \right)$$

instead of i-1 in 0/1 knapsack we can use item again

else -

$$t[i][j] = t[i-1][j]$$

* Rod Cutting (Unbounded knapsack)

I/p. length[] {1 2 3 4 5 6 7 8} → wt

price[] {1 5 8 9 10 17 17 20} → val

N = 8. → length of rod → W.

Cut the rod into any number of pieces to maximize price

Note - sometime array size may not be N.

t[n+1][length+1] → general. Matrix size

N N

t[arr size][N+1]

↓ con. ↓ capacity

* Coin change problem.: Maximum no. of ways.

I/p coins [1, 2, 3] unlimited

sum = 5

o/p. max ways to get sum 5 -

o/p - 5

2+3 = 5

1+2+2 = 5

1+1+3 = 5

1+1+1+1+1 = 5

1+1+1+2 = 5

Init

t[n+1][sum+1] — amount
 — size of array

	0	1	2		sum+1
0	1	0	0	0	0
1	1				
2	1				
n+1	1				

```
if (coin[i-1] ≤ j)
t[i][j] =  t[i][j - coin[i-1]] + t[i-1][j]
else
t[i][j] =  t[i-1][j]
```

* Coin change – II Minimum no. of coins (note** – set ∞ to int_max - 1 as you do +1 in the eqn).

coins[] = {1, 2, 3}

sum = 5

min no. of coins to get sum.

t[n+1][size+1] — amount
 — size of array

	0	1	2	3	4	...	sum+1
0	∞	∞	∞	∞	∞	∞	∞
1	0						
2	0						
	0						
n+1	0						

eg. coins = {3, 5, 2}

? coin = [3] sum = [4]

4/3 . not divisible. so ∞

else if 4/2 => 2

eg.

```
for (int i = 1; j < sum+1; j++)
    if (j % arr[0] == 0)
        t[i][j] = j/arr[0]
    else
        t[i][j] = ∞
```

coins

code

```
if (coin[i-1] <= j)
    t[i][j] = min ( t[i][j - coin[i-1]] + 1 , t[i-1][j]
else
    t[i][j] = t[i-1][j]
```

Longest common subsequence & variations.

1. Longest common substring.
2. Print LCS
3. Shortest common supersequence superseqence
4. Print SCS
5. min. no. of insertion & deletion a→b
6. Largest Repeating Subsequence
7. length of longest subsequence of a which is a substring in b
8. Subsequence Pattern matching
9. Count how many times a appear as subsequence in b.
10. longest Palindromic subsequence
11. Longest Palindromic substring
12. Count of palindromic substring
13. min no. of deletion in a string to make it a palindrome
14. min no of insertions in a string to make it a palindrome

subsequence - can have breaks.

substring - continuous

eg x: a b c d g h

y: a b e d f h r

Subseq - abdh

Substr - ab

* Longest common subsequence

i/p - x : a b c d g h
y : a b e d f r.

o/p - len of longest common subsequence → 4

Recursive -

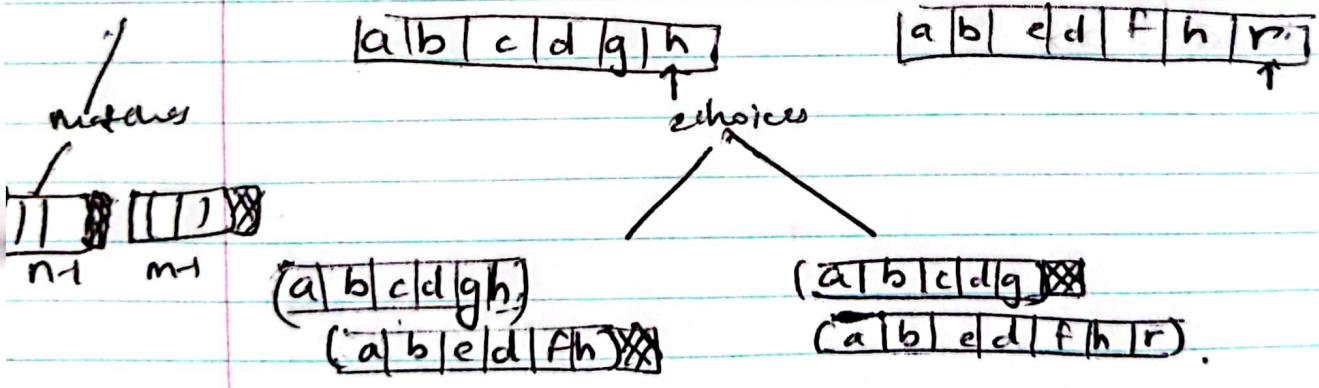
1) Base cond^n — think of smallest valid i/p.
2) choice diagram
3) smaller i/p, — start from end, char by char.

Base cond^n

- if (n == 0 || m == 0)
  return 0.

if (x[n-1] == y[m-1]) //same char.
  x → n-1
  y → m-1

else //diff char.




```
int LCS (String x, string y, int n, int m).
{ if (n==0 || m==0) return 0;        Base case
  if (x[n-1]==y[m-1])
        return 1 + LCS (x, y, n-1, m-1)
  else return max (LCS(x,y,n,m-1), LCS (x,y,n-1,m))
```

Recursive
Bottom up → RC. + table
top down - table only.

LCS (Bottom up - memorized)

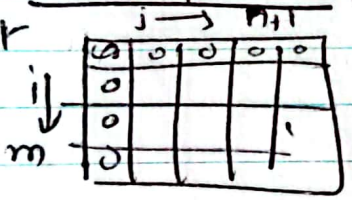
→ t[n+1][m+1]

→ Fill with -1

```
int LCS ( string x, string y, int m, int n)
{
  if (n==0 || m==0) return 0;

  if (t[m][n] != -1) return t[m][n];

  if(x[m-1] == y[n-1])
return   t[m][n] = 1 + LCS (x, y, m-1, n-1)
  else.
return t[m][n] = max (LCS (x, y, m-1, n), LCS(x, y, m, n-1)
```

LCS. (Top down).

int



→ for (int i = 1 to n)
  for (int j = 1 to n)
  {
    if (x[i-1] == y[j-1])
      t[i][j] = 1 + t[i-1][j-1]
    else
      t[i][j] = max (t[i-1][j], t[i][j-1]

  return t[m][n]

Time complexity LL vs arrays.

		Array	SLL	DLL
Access.	by index.	O(1)	O(N)	O(N)
Add.	before 1st node	O(N)	O(1)	O(1)
	after given node	O(N)	O(1)	O(1)
	after last node.	O(1)	O(1)	O(1)
Delete	first node	O(N)	O(1)	O(1)
	a given node	O(N)	O(N)	O(1)
	last node.	O(1)	O(N)	O(1)
search	a given node	O(N)	O(N)	O(N)

LL - insert/delete frequently
array - access element by index often

	Array	Hashtable (LL)	Hashtable (BST)
Insert	O(1)	O(1)	O(1)
Search	O(N)	O(N)	O(logN)

Search	Sort	worst	Best	Avg
Linear O(n)	Bubble	$O(n^2)$	O(n)	$O(n^2)$
Binary O(logn)	Selection	$O(n^2)$	O(n)	$O(n^2)$
	Insertion	$O(n^2)$.	O(n)	$O(n^2)$.
	Shell.	$O(n\log^2 n)$ // $O(n^4)$	O(n)	$O(n\log^2 n)$.
	merge	O(nlogn)	O(nlogn)	O(nlogn).
	Quick	$O(n^2)$	O(nlogn)	O(nlogn)
	Heapsort	O(nlogn)	O(nlogn)	O(nlogn)

Heap → Insert/Del - O(logN)
Max/min - O(1).
Construct - O(n)
Top k problem → O(N + klogN) k th largest/smallest - O(N + klogN)

BST. -	Insert	Delete	Retrieve	Search
	O(logn)	O(logn)	O(logn)	O(logn)
	O(h)	O(logn)	O(h)	O(h)

Bfs, Dfs - O(V+E)
space O(V)

Kruskal - O(ElogE). space O(V).
Prim. O(ElogV) space O(V)
non -ve Dijkstra O(E+VlogV) O(V)
(-ve also) Bellman
Kahns ~~O(VE)~~ O(V+E) list ~~O(V)~~ O(E) list.