The methods in java gets stored as stacks in the memory, one on top of another

When the main method calls the other functions, it gets stored above main, and when that function returns a value, that stack which stored the function gets cleared, when main method is done it also gets cleared.

Methods and Functions:

Methods called through objects of class

Functions called directly

Time Complexity: Relation between Input Size and Running Time (Operations)

Time complexity directly proportional input size

Best case : Minimum

Average Case: average

Worst Case: wont exceed this time

3 ways to do time complexity

Best Case : omega(1)

Avg: theta(n+1/2)

Worst: O(n) //Time Complexity for competitive coding

SPACE COMPLEXITY

1 SPACE COMPLEXITY : normally if you have some variables in your code those will be the space complexity, even if you increase the input size of a variable it wont affect the space complexity of the program.

2 Space complexity

Arrays: In case of arrays if you increase the number of elements to be stored in the array, it will increase the space complexity

Array initialization

Int [] arr = new int[10]; // new: used for making a new space inside memory

Memory address gets stored in hexadecimal

Default initialization:

Objects : null

Int:0

Float: 0.0

String:””

Boolean : false

Array : depends of type of array

2D Array :

Int [][] arr = new int[rows][coloumns];

We use .CompareTo() method to check equality of two strings s1 and s2

Eg: s1.CompareTo(s2);

It returns 0 if s1 = s2

+ve if s1> s2 (a is lowest, z is highest, if equal goes to next character )

-ve if s1<s2

When we initialize a variable

String s = “Alan Saji”;

s gets stored in stack

Alan Saji gets stored in heap

<< left shift: shift to left with zeros adding on places 3 << 1 =? 010 << 1 = 100

>> right shift: shift to right with zeros adding on places

Bit manipulation

Get:

Set: set to 1 if its 0 or keep it 1

Clear: set to 0

Update: if 1 then 0 , if 0 then 1

GET BIT OPERATION

Get the 3rd bit (position 2) of a number n. (n=0101) {position counting starts from right to left starting with zero}

* 0 1(3rdbit) 0 1

Process:

Bit Mask: 1<<i (we have to do this bit mask manipulation ( bit mask is an extra number we have to apply in the process) (here i is the position)

Operation: AND (we have to do ‘and’ operation with the original n and the bit mask )

Execution:

* 1<<2 = 0001 << 2 = 0100
* 0100 & 0101(original n)
* 0100 => this implies the 3rd bit was 1 because the result is non-zero, if it was 0000 then the 3rd bit would be zero (we got the 3rd bit by GET operation)

SET BIT OPERATION

Set the 2nd bit (position 1) of a number n. n = 0101

Process:

Bit mask: 1<<i

Operation: OR

Execution:

* 0001 << 1 = 0010
* 0010 OR 0101 => 0111 (hence we set the 2nd bit to 1) the number 5 has changed to 7

CLEAR BIT OPERATION

Clear the 3rd bit of a number n , n = 0101

PROCESS:

Bit mask = 1 << i

Operation : AND with NOT( of bitmask)

Execution:

* 1 << 2 = 0001 << 2 = 0100
* !(0100) => ~(0100) => 1011
* 1011 AND 0101 => 0001 ( 0001 is one in decimal

UPDATE BIT OPERATION:

Update the 2nd bit position(1) of a number n to 1 , n = 0101

For 0:

Bitmask : 1<<i

Operation : AND with NOT (of bitmask)

For 1:

Bitmask: 1<<i

Operation : OR

SORTING

BUBBLE SORT , SELECTION SORT, INSERTION SORT