General Notes

14 March 2023 03:04 AM

A class is a group of objects which have common properties. A class can have some properties & functions (methods).
A function is a block of code which takes some input, performs some operations and returns some output.
The functions stored inside classes are called methods.
A variable is a container (storage area) used to hold data. Each variable should be given a unique name (identifier).

Each .java file can have **max one public class** & any number of non-public classes.
If a .java file has more than one class, then each class will be compiled into a **separate class** file.
A Java program can be **compiled without main()**.
On executing the program, it gives exception till JDK6 however gives error as main method not found in JDK 7+.

A **loop** consists of an *initialization statement, a test condition and an increment statement.*

Use long instead of int for large numbers esp. in add or multiply

Use Math.max(var1, var2) to find higher value

Any statement inside switch without start of any case then it will not be executed

Auto keyword is same as without writing auto to any variable. variable has scope in the block it is.

For-each loop:
Variable *arrayOfInt* is an array of Integers

```
for(int var : arrayOfInt) {
        .......
}
```

For non-null values, String.valueOf(str) is same as str.toString()

```
System.out.println(String.valueOf(str));  // This will print a String equal to "null"
System.out.println(str.toString()); // This will throw a NullPointerException
```

str.isEmpty( ) to check whether string is empty

To check whether a number is even OR odd by bitwise operator: (number & 1 == 1)
'&' will give output as 1 for each bit only when both numbers same location bit is 1; o/w 0

15		00001111
18		00010010
15 & 18		00000010

which equals 2

Taking User Input

```
import java.util.Scanner;

Scanner sc = new Scanner(System.in);
int n = sc.nextInt();
.
.
sc.close();
```

Always use curly braces in following format

```
if(....) {

}
```

Data Type

23 May 2024 01:04

Data types are declarations for variables. This determines the type and size of data associated with variables which is essential to know since different data types occupy different sizes of memory.
There are 2 types of Data Types :

- Primitive Data types : to store simple values (fix size)
- Non-Primitive Data types : to store complex values (vary in size & declared using new keyword)

A **constant** is a variable in Java which has a fixed value i.e. it cannot be assigned a different value once assigned (denoted by **final** keyword)

Data Type	Size	Description
`byte`	1 byte	Stores whole numbers from -128 to 127
`short`	2 bytes	Stores whole numbers from -32,768 to 32,767
`int`	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
`long`	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`float`	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
`double`	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
`boolean`	1 bit	Stores true or false values
`char`	2 bytes	Stores a single character/letter or ASCII values

Implicit Type-casting: Auto like int a = 2; double res = 15.65 + a;
Explicit Type-casting: Manually like int a = 5 + (int) 2.0;

JDK, JRE, JVM

23 May 2024 01:01

1. JDK provides the environment to develop & execute the Java program. JDK is a kit that includes two things:
    Development Tools (to provide an environment to develop your java programs)
    JRE (to execute your java program).
2. JRE (Java Runtime Environment) is an installation package that provides an environment to only run (not develop) the java program (or application) onto your machine. JRE is only used by those who only want to run Java programs that are end-users of your system.
3. JVM (Java Virtual Machine) is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for executing the java program line by line, hence it is also known as an interpreter.

Java Bytecode (.class file) is a platform-independent intermediate representation of the Java program that can be executed by any JVM. It is generated by Java compiler.
The JVM is what makes Java programs portable. We can compile a Java program on one platform and run it on another platform without having to recompile it.

Java is platform independent but JVM isn't. For each system like Windows or Mac OS, we need separate JVM.
JVM is responsible for translating the bytecode into machine code using just-in-time compiler.

JRE should be present for each respective OS/machine to run .class file.

Java Naming Rules

23 May 2024 01:00

In package name, everything is small even while we are combining two or more words in java.
In constants, we do use everything as uppercase and only '_' character is used even if we are combining two or more words in java.
Variable names should not start with underscore _ or dollar sign $ characters, even though both are allowed.
One-character variable names should be avoided except for temporary variables like I, j for loop.
Class name generally start by capital letter

Identifier
An identifier can be a class name, method name, variable name, or label.
Can’t start with digits, are case-sensitive, cannot be a reserved word / keyword.

Java is statically & strongly typed language

Patterns (star)

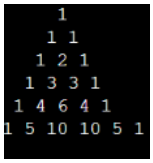
Friday, May 24, 2024 9:08 PM

Split the entire into n number of rows where pattern trend changes
Similarly each row into m set of columns where pattern trend change

```
int n = 5;

for(int i=1; i<=n; ++i) {
        for(int j=1; j<=i; ++j) {
                if(j==1 || i==j)
                        System.out.print("*");
                else
                        System.out.print(" ");
        }
        for(int j=1; j<=n-i; ++j)
                System.out.print("  ");
        for(int j=1; j<=i; ++j) {
                if(j==1 || i==j)
                        System.out.print("*");
                else
                        System.out.print(" ");
        }
        System.out.println();
}

for(int i=n; i>=1; --i) {
        for(int j=1; j<=i; ++j) {
                if(j==1 || i==j)
                        System.out.print("*");
                else
                        System.out.print(" ");
        }
        for(int j=1; j<=n-i; ++j)
                System.out.print("  ");
        for(int j=1; j<=i; ++j) {
                if(j==1 || i==j)
                        System.out.print("*");
                else
                        System.out.print(" ");
        }
        System.out.println();
}

System.out.println();
```

```
*         *
**       **
* *     * *
*   *   *   *
*    **    *
*   *  *   *
* *     * *
**       **
*         *
```

Pascal Triangle



The term is sum of prev row above two numbers.
Eg. 1+4=5; 4+6=10 etc

Solution #1: Use ArrayList to keep storing prev row values & re-initialise with empty later
Solution #2: Use formula: n!/(r! * (n-r)!) where n=row index & r=column index; index begins at 0
Solution #3: Print starting from number=1 at each row, vid next element = number*(rowIndex+1-colIndex) / colIndex ; eg. To get 6, term=4*(4+1-2)/2 = 6

LCM, HCF/GCD, Fibo

25 May 2024 15:51

Formula: **LCM = (x * y) / GCD**

```
//LCM
int large, small;
if(n1>n2) { large = n1; small = n2; }
else { large = n2; small = n1; }
//large = Math.max(n1, n2);

int multiple = large;

while(multiple%small != 0)
      multiple += large;
System.out.println(multiple);


//HFC or GCD
if(x == 0) return y;
if(y == 0) return x;

int t;
while(x%y != 0) {    //approach #2: while(y!=0)
    t = y;     y = x%y;    x = t;
}
return y;            //approach #2: return x;

//HCF by Recursion
if (a == 0) return b;
return gcd(b % a, a);
```

```
//Fibo Series
int n1 = 0, n2 = 1;

if(n >= 1) {
      System.out.print(n1+", ");

      int term = n2;
      for(int i=2; i<=n; ++i) {
            System.out.print(term + ", ");
            term = n1+n2;
            n1 = n2;
            n2 = term;
      }
}
```

Operators

14 March 2023 04:06 AM

Operator	Type	Associativity
()	Parentheses	Left to Right
[]	Array subscript	
.	Member selection	
++	Unary post-increment	Right to left
--	Unary post-decrement	
++	Unary pre-increment	Right to left
--	Unary pre-decrement	
+	Unary plus	
-	Unary minus	
!	Unary logical negation	
~	Unary bitwise complement	
(type)	Unary type cast	
*	Multiplication	Left to right
/	Division	
%	Modulus	
+	Addition	Left to right
-	Subtraction	
<<	Bitwise left shift	Left to right
>>	Bitwise right shift with sign extension	
>>>	Bitwise right shift with zero extension	
<	Relational less than	Left to right
<=	Relational less than or equal	
>	Relational greater than	
>=	Relational greater than or equal	
instanceof	Type comparison (objects only)	
==	Relational is equal to	Left to right
!=	Relational is not equal to	
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
|	Bitwise inclusive OR	Left to right
&&	Logical AND	Left to right
||	Logical OR	Left to right
? :	Ternary conditional	Right to left
=	Assignment	Right to left
+=	Addition assignment	
-=	Subtraction assignment	
*=	Multiplication assignment	
/=	Division assignment	
%=	Modulus assignment	

int i = 2;

i+1 % 3 == 0

i+ **1 % 3** == 0
i+1 == 0
i+1 == 0
2+1 == 0
3 == 0
3 == 0
false

Unary Operator: works with only one variable
(eg post & pre increment)

Binary opr: works vid 2 or more variable
(eg arithmetic opr)

Shorthand operators (*=) can't be used with String
Only += works only as concatenation, not as addition

Arithmetic Operators
*****
/
%
+
-

Relational Operators
<
<=
>
>=
==
!=

Logical Operators
&&
||
!

Assignment Operators
=
+=
-=
***=**
/=
%=

Bitwise Operators
~ 1's complement (reverses bit, also called Bitwise NOT)
<< Binary left shift
>> Binary right shift
//basically convert number into binary and perform bit shifting

& Bitwise AND
| Bitwise OR
^ Bitwise XOR

//For AND/OR convert number into binary and perform bit level AND/OR operations
XOR is Exclusive-OR which gives true if bits are different o/w false

Bit Manipulation

25 May 2024 23:01

4 operations

1) Get Bit: Get the particular bit of a number
2) Set Bit: Return new number by setting the particular bit in a number as 1
3) Clear: Return new number by setting the particular bit in a number as 0
4) Update: Return new number by setting the particular bit in a number as specified

BitMask Forumula: **1<<Position**

1) Get Bit (Bitwise AND Operation with shifted BitMask and our number)
Get the 3rd bit of number 4.
Binary number: 0100
Position: 2 (3rd bit from right with start index as 0)

Bit Mask will be 1<<2 which will evaluate as 0100
Now, perform Bitwise AND operation b/w number 4 and shifted BitMask
0100 AND 0100 gives result as 1000
Now, check whether result is 0 or not. If result is 0, bit is 0 o/w bit is 1

Logic of this: To whatever position we've to check the bit, we'll perform AND operation with a number having all bit as 0 except on the checking position. Bitwise AND operation will give true OR 1 only when both bits are 1 and eventually number will be 0 or some +ve number;

2) Set Bit (Bitwise OR Operation with shifted BitMask and our number)
Basic steps similar to Get Bit, however the resulting number after performing the Bitwise OR operation itself will be our final answer

3) Clear Bit (Bitwise AND Operation with 'Bitwise NOT of shifted BitMask' and our number)
Basic steps similar to Set Bit.
Logic: In Bitmask we have only our position bit as 1 and other Zero. But in our result we want our position bit to be Cleared (set as 0) and others as per the number. Hence, we take Bitwise NOT (~ or 1's complement) of our shifted Bitmask which gives all bits except our position bit as 1. Now AND of this with any number will ensure that position bit remains 0 and other Bits will be 1, if it was 1 in our actual number.

4) Update Bit
Has two cases:
Case #1 Update say i th Bit as 1 means we've to do **Set Bit** operation
Case #2 Update say i th Bit as 0 means we've to do **Clear Bit** operation

For any operation, basic understanding:
To get a particular bit, say 3 in number 12, we will count bit from right-hand side.
Position (Index) value will start as 0 from right-hand side. Hence, position will be 2 in our case.

Eg. Get 3rd bit of number n. (n = 0101)
Then, position will be 2

How to solve/perform bit operations
For each operation, create bit mask using bit mask formula where we'll shift binary number 1 with Position times using bit-wise operator; and then we'll perform some Bitwise operation b/w shifted bit mask number & our number.
Then based on Bitwise Operation output, we'll utilise the resulting number to get our answer.

Note: **Don't convert number to binary**
Use bitwise logical operator

```
System.out.print("Enter the number: ");
int number = sc.nextInt();

System.out.print("Enter the bit to fetch: ");
int bit = sc.nextInt();

int position = bit - 1;
int bitMask = 1 << position;

int result = bitMask & number;

if(result == 0) System.out.println("Bit " + bit + " is 0");
else System.out.println("Bit " + bit + " is 1");

System.out.print("Enter the bit to set: ");
bit = sc.nextInt();    position = bit-1;    bitMask = 1 << position;
result = bitMask | number;
System.out.println("Resulting number is " + result);

System.out.println("\nNumber: " + number);
System.out.print("Enter the bit to clear: ");
bit = sc.nextInt();    position = bit-1;    bitMask = 1 << position;
result = ~bitMask & number;
System.out.println("Resulting number is " + result);
```

//Program to see whether number is power of 2 or not

```
System.out.print("Enter the number: ");    int number = sc.nextInt();

//Number will be power of 2, if number has only one of the bit as 1
//Fetch right-most index bit, if it's 1 then add to count variable
//Keep right-shifting number until it's 0

int bitMask = 1 << 0; //which is 0001 OR 1 in decimal;
int count = 0;

while(number > 0) {
    int isBitOne = bitMask & number;
    if(isBitOne != 0) ++count;
    number = number>>1;
}
if(count == 1)  System.out.println("Number is power of 2");
else System.out.println("Number isn't power of 2");
```

Time & Space Complexity

14 March 2023 02:20 AM

Time Complexity
3 scenarios: Best, Average, Worst

Eg. Searching an element in an array
Best: O(1); other cases O(N)

We generally consider Worst case for our all usage.

For loop of n-iteration inside another for loop of m-iteration is example of O(N*M) which will eventually be O(N^2)

If loops are non-nested and two separate loops then O(N+M) which will be O(N)

Order of time complexity:
O(nn) > O(n!) > O(n3) > O(n2) > O(n.log(n)) > O(n.log(log(n))) > O(n) > O(sqrt(n)) > O(log(n)) > O(1)

Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.
Types of notations
1. O-notation: It is used to denote asymptotic upper bound. For a given function g(n), we denote it by O(g(n)). Pronounced as “big-oh of g of n”. It is also known as worst case time complexity as it denotes the upper bound in which the algorithm terminates.
2. Ω-notation: It is used to denote asymptotic lower bound. For a given function g(n), we denote it by Ω(g(n)). Pronounced as “big-omega of g of n”. It is also known as best case time complexity as it denotes the lower bound in which the algorithm terminates.
3. **Θ**-notation: It is used to denote the average time of a program.

Space Complexity
Space needed by variables.
Space complexity will be O(1) as we require space for 1 Scanner object, 1 variable n and 1 variable i

```
Scanner sc = new Scanner(System.in);
int n = sc.nextInt();

for(int i=0; i<n; i++) {
   System.out.println("hello");
```

Space complexity of an algorithm quantifies the amount of time taken by a program to run as a function of length of the input.
It is directly proportional to the largest memory your program acquires at any instance during run time.
For example: int consumes 4 bytes of memory.

Input and Output array space is not counted for Space Complexity. Only the array used in the process is counted.

Array

21 March 2023 02:44 AM

We can **append character** to a **String**
String str = "Hello"; char c = "W";
str = str + c; //str will be HelloW

Array has length as property not a method.
Hence small brackets () not needed.
For string, length() is a method

Format of **indexOf**
public int indexOf(String str)
public int indexOf(String str, int fromIndex)
public int indexOf(int char)
public int indexOf(int char, int fromIndex)

To replace only first occurrence of substring,
use str.**replaceFirst**(target string, replacement string)
str.replace("Hel", "halu");

Find Max & Min element of Array

```
//int max = arr[0], min = arr[0];
int max = Integer.MIN_VALUE, min = Integer.MAX_VALUE;
//ensure max variable is initialised as MIN_VALUE
constant not the MAX_VALUE & similar for min variable

for(int i=0; i<n; ++i) {
      if(arr[i] > max) max = arr[i];
      if(arr[i] < min) min = arr[i];
}
```

Binary Search (array should be sorted)

```
int beg = 0, end = n-1, mid = (beg + end) / 2;

while(arr[mid] != search && beg <= end) {
      if(arr[mid] > search) end = mid;
      else beg = mid;
      mid = (beg + end) / 2;
}
if(beg > end)    System.out.println("Not Found");
else  System.out.println("Found at position: " + (mid+1));
```

Arrays.sort(arr); has Time Complexity **n log n**

Anonymous array can be passed as argument to method
`new int[]{10,22,44,66}`

For each loop
for(int i:arr)

Used to iterate Arrays and Collections

1. Only iterates in forward direction
2. Can't track the index number in current iteration
3. Array data can't be modified using this

2-D Array (Matrix)

25 May 2024 17:32

2-D Array (Matrix)

```
int matrix[][] = new int[r][c];
System.out.print("Enter " + r*c + " numbers: ");
for(int i=0; i<r; ++i) {
    for(int j=0; j<c; ++j)
         matrix[i][j] = sc.nextInt();
}
```

//Search a number in Matrix

```
System.out.print("Enter the number to search: ");
int search = sc.nextInt();
for(int i=0; i<r; ++i) {
  for(int j=0; j<c; ++j) {
    if(matrix[i][j] == search) {
      System.out.println("Found at: " + (i+1) + ", " + (j+1));
      return;
    }
  }
}
```

Transpose of Matrix (interchange row & col values)

```
int transpose[][] = new int[c][r];
int x=0, y=0;

for(int i=0; i<c; ++i) {
  for(int j=0; j<r; ++j) {
    transpose[i][j] = matrix[x++][y];
     if(x==r) { x=0; ++y; }
  }
}
```

Spiral Print

Matrix

1 5 7 9 10 11
6 10 12 13 20 21
9 25 29 30 32 41
15 55 59 63 68 70
40 70 79 81 95 105

Spiral Print: 1 5 7 9 10 11 21 41 70 105 95 81 79 70 40 15 9 6 10 12 13 20 32 68 63 59 55 25 29 30 29

```
int rowBeg = 0, colBeg = 0, rowEnd = r-1, colEnd = c-1;

while(rowBeg <= rowEnd && colBeg <= colEnd) {
        for(int i=colBeg; i<=colEnd; ++i)
System.out.print(matrix[rowBeg][i] + " ");
        ++rowBeg;

        for(int i=rowBeg; i<=rowEnd; ++i)
System.out.print(matrix[i][colEnd] + " ");
        --colEnd;

        for(int i=colEnd; i>=colBeg; --i)
System.out.print(matrix[rowEnd][i] + " ");
        --rowEnd;

        for(int i=rowEnd; i>=rowBeg; --i)
System.out.print(matrix[i][colBeg] + " ");
        ++colBeg;
}
```

Functions

25 May 2024 01:31

A function is a block of code that performs a specific task.
Why are functions used?
If some functionality is performed at multiple places in software, then rather than writing the same code, again and again, we create a function and call it everywhere. This helps reduce code redundancy.
Functions make maintenance of code easy as we have to change at one place if we make future changes to the functionality.
Functions make the code more readable and easy to understand.

In Java, functions are saved in stack.
Saving anything in stack is called stack frame.

Methods vs Functions
When we call a function of a class using object, then it is called Method.

Static Keyword

14 March 2023 02:23 AM

static means something which is common.

Static variable (class attributes)
Variable which can be accessed directly thru class name without creating object
Eg. For student class, property school name will be same for all objects, hence it is better to set it as static. Also, if school name is later changed, it will get updated for all student objects.
Static variable are allocated memory only once irrespective of number of times the object of that class is created

Static block
Used to initialize the static data member.
Executed before main method when class is loaded. Eg. for Welcome msg

Static also used for Nested class

```
class Student {
  static String school;
  String name;
  public static void changeSchool(String schoolName) {
        name = schoolName;
 }
}

public class OOPS {
  public static void main(String args[]) {
     Student.school = "JMV";
     Student s1 = new Student();
     Student s2 = new Student();
     s1.name = "Meena";
     s2.name = "Beena";
     System.out.println(s1.school);
     System.out.println(s2.school);

  }
}
```

Static method
static method can't be overridden. Basically, overriding concept is not exhibited.
[Reason: **static methods are associated to class & not the objects hence, each static method will be particular to that specific class only**]
Phenomena is called Method Hiding where we have same signature method in base & derived class and methods are static. So, derived class will hide the method of base class.

static method is resolved at compile time, not run time for choosing which method to call.
IF only one of them is public & static and other is not then it will throw compilation error as u can't override. Eg. derived class method is static then also error.

static method can't access non-static methods or data members of class
this, super can't be use for static methods
NON-static methods can access both static & non-static variables

Java main method static because the object is not required to call a static method.
If it were a non-static method, JVM creates an object first then call main() method that will lead the problem of extra memory allocation.

```
//private/public static variables, static methods, getter & setter
class Student {
  private static String school;
  public static boolean schoolIsOpen;
  String studName;
  public static void setSchool(String schoolName) {
        school = schoolName;
  }
  public static String getSchool() {
        return school;
 }
}

public class StaticClass {
  public static void main(String args[]) {
     Student.setSchool("JMV");        Student.schoolIsOpen = false;
     Student s1 = new Student();      Student s2 = new Student();
     s1.studName = "Meena";           s2.studName = "Beena";
     System.out.println(s1.getSchool() + "\t" + s2.getSchool());      //JMV       JMV
     System.out.println(s1.schoolIsOpen + "\t" + s2.schoolIsOpen); //false     false

     Student.setSchool("MVJ");        Student.schoolIsOpen = true;
     System.out.println(s1.getSchool() + "\t" + s2.getSchool());      //MVJ       MVJ
     System.out.println(s1.schoolIsOpen + "\t" + s2.schoolIsOpen); //true      true
  }
}
```

String

25 May 2024 19:27

Java Strings are Immutable

No operations can be performed on the existing string; for any changes, new string will/has to be created

To concatenate: use '+' or .concat() method
To get size of String: str.length()
To get each character: str.charAt(i);
To replace character/string: str.replace("abc","xyz"); OR str.replace('a', 'x');

```
String result = "";
for(int i=0; i<str.length(); ++i) {
	if(str.charAt(i) == 'e')  result += 'i'; //replacing 'e' vid 'i'
	else   result += str.charAt(i);
} return result;
```

To compare string lexicographically: s1.compareTo(s2);
give 0 if same characters, if s1>s2 then +ve value, if s1<s2 then -ve value
To check if strings are equals character data: s1.equals(s2);
// Double equals '==' will not work as each String points to different memory location

```
new String("Tony") == new String("Tony")  //will always give false
```

IMPORTANT: https://sentry.io/answers/how-to-compare-strings-in-java#why-you-shouldnt-use--for-string-comparison

Substring: str.substring(startIndex, endIndex); //will give from startIndex till an index before endIndex
If no endIndex specified, then till end of string

ParseInt Method of Integer class

To convert a string to any particular data-type: int n = Integer.parseInt(data); //converts the string to a Integer data-type if proper o/w exception. Similarly, to convert any data to string, use **.toString()**: int n = 2542; String s = n.toString();

Taking String as Input

String s = sc.nextLine();

int n = sc.nextInt();
String str = sc.nextLine(); //will take empty string as input bcoz after enter the previous integer input, we'll press enter key which is a \n (newline character) and hence same is taken as input for the nextLine()

So, use extra *sc.nextLine()* to fix this:

```
int n = sc.nextInt();
sc.nextLine();
String str = sc.nextLine();
```

Shorthand operators (*=) can't be used with String
Only += works only as concatenation, not as addition

StringBuilder

26 May 2024 01:57

StringBuilder
Creating new String (performing operations on String) takes time as String variable has to point to newer location on each operation. Hence, StringBuilder class came into existence where same object is modified on every operation.

StringBuilder sb = new StringBuilder("MetaData");
System.out.println(sb); //prints MetaData
System.out.println(sb.charAt(4)); //gives 'D'

Methods of StringBuilder class
sb.setCharAt(int index, char ch) //changes character at index with ch character
sb.append(data) //adds given data (string/int/float/char) to end of original string
sb.insert(int index, data) //will insert data at specified index
sb.delete(int startIndex, int endIndex) //delete characters in the range

For equals(), first conver sb to String by using toString and then use equals()

System.out.print("Enter a string: ");
StringBuilder sb = new StringBuilder(sc.nextLine());
sc.close();

System.out.println(sb);

sb.append(" Jaidi, "); sb.append(29);
System.out.println(sb);

sb.setCharAt(11, 'Z');
System.out.println(sb);

sb.insert(0, "Mr. ");
System.out.println(sb);

sb.reverse();
System.out.println(sb);

//Reverse string
```
char temp = ' ';
for(int i=0; i<sb.length()/2; ++i) {
	int indexEnd = sb.length()-i-1;
	temp = sb.charAt(indexEnd);
	sb.setCharAt(indexEnd, sb.charAt(i));
	sb.setCharAt(i, temp);
}
```

//Binary to Decimal
```
for(int i=0; i<size; ++i) {
	String bit = "" + binary.charAt(i);
	decimal += Integer.parseInt(bit) *
Math.pow(2, size-i-1);
}
```

//Decimal to Binary
```
StringBuilder newBinary = new
StringBuilder("");
while(decimal != 0) {
	newBinary.append(decimal % 2);
	decimal = decimal / 2;
}
newBinary.reverse();
```

StringBuffer

26 May 2024 01:57

StringBuffer Class	StringBuilder Class
StringBuffer is present in Java.	StringBuilder was introduced in Java 5.
StringBuffer is synchronized. This means that multiple threads cannot call the methods of StringBuffer simultaneously.	StringBuilder is asynchronized. This means that multiple threads can call the methods of StringBuilder simultaneously.
Due to synchronization, StringBuffer is called a thread safe class.	Due to its asynchronous nature, StringBuilder is not a thread safe class.
Due to synchronization, StringBuffer is lot slower than StringBuilder.	Since there is no preliminary check for multiple threads, StringBuilder is a lot faster than StringBuffer.

```
//Java Program to demonstrate the performance of StringBuffer and StringBuilder classes.
public class ConcatTest{
   public static void main(String[] args){
      long startTime = System.currentTimeMillis();
      StringBuffer sb = new StringBuffer("Java");
      for (int i=0; i<10000; i++)
         sb.append("Tpoint");
      System.out.println("Time taken by StringBuffer: " + (System.currentTimeMillis() - startTime) +
"ms");
      startTime = System.currentTimeMillis();
      StringBuilder sb2 = new StringBuilder("Java");
      for (int i=0; i<10000; i++)
         sb2.append("Tpoint");
      System.out.println("Time taken by StringBuilder: " + (System.currentTimeMillis() - startTime) +
"ms");
   }
}
```

Time taken by StringBuffer: 16ms
Time taken by StringBuilder: 0ms

Java Keywords (Imp)

14 March 2023 02:26 AM

this keyword in OOPS

To pass the current object as a parameter to another method
To refer to the current class instance variable

Throw & Throws

Throw: To throw exception explicitly inside any method. Only single exception can be thrown.
Throws: To declare one or more exceptions in the method signature, which could be thrown by method.

Final, Finally, Finalize

Final
Variables: Value not changable
Method: Not overridable
Class: Not extendable, all methods also become final, variables are not final

Finally
Keyword in try-catch block, executed in all cases even if exception thrown.
Doesn't execute if JVM crashes.
Try code won't execute after an statement with exception is thrown..try then catch then finally
Try catch & then finally if smooth
If catch block doesn't have the exception which is occuring then, try is executed then finally and then exception is thrown

Finalize
Method used to perform clean up processing just before object is garbage collected.

Volatile

Volatile keyword: Modify the value of a variable by different threads. makes classes thread safe.
Multiple threads can use a method and instance of the classes at the same time. Can be used with variables & objects but not methods or classes.

Value of the volatile variable is not cached & always read from the main memory.

OOPs Concept

14 March 2023 02:27 AM

Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts defined below :
Object, Class, Inheritance, Polymorphism, Abstraction, Encapsulation

Class

26 May 2024 12:21

Class is a user-defined data type which defines its properties and its functions. Class is the only logical representation of the data. For example, Human being is a class. The body parts of a human being are its properties, and the actions performed by the body parts are known as functions. The class does not occupy any memory space till the time an object is instantiated.

Static Nested Class	Non-Static (Inner) Class
Can be instantiated independently	Requires an instance of the outer class
Independent of any instance of the outer class	Tightly associated with an instance of the outer class
Can access static members directly	Can access both static and non-static members
Cannot access instance fields directly	Can access instance fields directly
Cannot invoke non-static methods	Can invoke both static and non-static methods
Cannot access non-static nested classes	Can access both static and non-static nested classes

```
public class NestedClass {
	int size;

	static class A {
		void setSize(int data) {
			size = data;
			//error here as static class can't
			//access non-static data/methods of outer class
		}
	}
	class B {
		void setSize(int data) {
			size = data;
		}
	}

	public static void main(String[] args) {
		A a = new A();

		NestedClass nc = new NestedClass();
		B b1 = nc.new B();

		B b2 = new B();
		//error here; non-static nested class not
		//accessible vidout outer class object
	}
}
```

```
class Student {
  String name;
  int age;

  public void getInfo() {
    System.out.println("The name of this Student is " + this.name);
    System.out.println("The age of this Student is " + this.age);
  }
}

public class OOPS {
  public static void main(String args[]) {
    Student s1 = new Student();
    s1.name = "Shad";
    s1.age = 29;
    s1.getInfo();
  }
}

//Here class name is Student and has two properties viz. name, age
(called as attributes) and a function viz. getInfo() (called as behaviour)
```

Object

26 May 2024 12:22

Object is a run-time entity. It is an instance of the class. An object can represent a person, place or any other item.
An object can operate on both data members and member functions.

```
class Pen {
   String color;

   public void printColor() {
       System.out.println("The color of this Pen is " + this.color);
   }
}

public class OOPS {
   public static void main(String args[]) {
       Pen p1 = new Pen();
       p1.color = blue;

       Pen p2 = new Pen();
       p2.color = black;

       Pen p3 = new Pen();
       p3.color = red;

       p1.printColor();
       p2.printColor();
       p3.printColor();
   }
}
```

//Here objects are p1, p2, p3 each having their own value for properties (data-members) with a behaviour (member function) printColor()

When an object is created using a **new** keyword, then space is allocated for the variable in a **heap**, and the *starting address* is stored in the **stack** *memory*.

'this' keyword is used to refer to the current class instance variable OR pass the current object as an argument to another method

Constructor

Constructor returns instance created by **new** keyword

26 May 2024 12:34

A special method which is invoked automatically at the time of object creation.
Used to initialize the data members of new objects generally.

- Constructors have the same name as class or structure.
- Constructors don't have a return type. (Not even void)
- Constructors are only called once, at object creation.

3 types:

1. Non-Parameterized (default) Constructor: Has no argument. ***Created by default in Java ONLY when no other constructor is created by user.*** We can declare our own to customize the default values OR display some success message. Gets invoked when we create an object for any class.

```
class Person {
        String name;
        Brand(int id, String brandName) {    //Parameterised Constructor
                this.name = brandName;
        }
}
public static void main(String[] args) {
        Brand b1 = new Brand(5, "ENOVIA");
        Brand b2 = new Brand(); //Compilation error as constructor() undefined
        //As user already created one constructor, the default constructor is not created
        automatically in Java
}
```

2. Parameterized Constructor: Has arguments to initialize default values to data -members of object. Helps in creating object with distinct values easily.

3. Copy Constructor: Has another object as argument. Initialises the data -members with value same as that of data-members of another object. It is an overloaded constructor. In Java, it has to be user-defined while C++ has default also.

Unlike languages like C++, Java has no Destructor. Instead, Java has an *efficient garbage collector* that *deallocates memory automatically*.

Constructor Chaining
When we create an object of derived class, first the base class constructor followed by derived class constructor is invoked (using super keyword implicitly by Java). This is Constructor chaining. Process of calling one constructor from another constructor w.r.t current object using this, super keyword

```
class Brand {
        int uid; String name;

        //No-arg (default) Constructor
        Brand() {
                this.uid = 0;
                this.name = "";
        }

        //Parameterised Constructor
        Brand(int id, String brandName) {
                this.uid = id;
                this.name = brandName;
        }

        //Copy Constructor
        Brand(Brand brandObject) {
                this.uid = brandObject.uid;
                this.name = brandObject.name;
        }
}

public static void main(String[] args) {
        Brand b1 = new Brand();
        Brand b2 = new Brand(5, "ENOVIA");
        Brand b3 = new Brand(b2);
}
```

Variables created inside constructor has scope within constructor only
Create a class with two variables & a method displayValue() to display the value of class variables.
Create a constructor & create same name variable as class variable and initialize them with some custom value.

Output:
When we invoke the displayValue() method, it shows the default value of the variable based on data-type; not the custom-values set by constructor.

```
public class ScopeLearn {
 int a, b;
 public ScopeLearn() {
        int a = 5;
        int b = 2;
 }
 public void display() {
        System.out.println("A: " + a + "\tB: " + b);
 }
 public static void main(String[] args) {
        ScopeLearn sl = new ScopeLearn();
        sl.display();  //A: 0    B:0
 }
}
```

Encapsulation

26 May 2024 12:50

Wrapping (combining) up data and functions together into a single unit called class is called as Encapsulation.

In Encapsulation, the data is not accessed directly but thru special methods. The attributes are kept private and a public getter & setter methods are used to access/manipulate the attributes.

Thus, encapsulation makes the concept of data hiding possible.

Data hiding: Restrict access to members of an object, reducing the negative effect due to dependencies.
e.g. "protected", "private" access modifiers feature in Java

Package

26 May 2024 15:39

Package is a group of similar types of classes, interfaces and sub-packages.
Packages can be built-in or user defined.

Built-in packages: java, util, io etc.

import java.util.Scanner; //java is package and util is a sub-package however for convenience we say java.util package
import java.io.IOException; //java.io package

Note: Importing a package will import only the classes inside it and not the sub-package or their classes.

How to create package?
By default, eclipse creates our java file in default package.
We can think of a package like a folder which can have multiple files & sub-folders.
In Eclipse, we can create package from Navigation OR create a folder & create java file inside it.
We'll see that the package folderName; will be the first line of every java file we create inside this package.

Now, in order to access the public class inside this package from any other different place, we've to use:
import packageName.className;

And then we can access this class of that package.

Access Modifiers

26 May 2024 16:14

4 types

Public
The access level of a public modifier is everywhere.
It can be accessed from within the class, outside the class, within the package and outside the package.

Private
The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
We need to use public getter & setter methods to access/manipulate private data-members.

Protected
The access level of a protected modifier is within the package and outside the package through child class.
If child class is not created, it cannot be accessed from outside the package.

Default
The access level of a default modifier is only within the package. It cannot be accessed from outside the package.
If no access level is specified, it will be the default.

Abstraction

26 May 2024 16:25

Showing only essential/necessary features & details and hiding other details from user.

Abstraction is achieved in 2 ways :
- Abstract class
- Interfaces (Pure Abstraction)

Abstract Class
Must be declared with an abstract keyword
Can't be instantiated (i.e. objects cannot be created as it is an abstract (concept) class)
Can have abstract & non-abstract methods, constructors and static methods as well
Can have final (non-abstract type only) methods which restricts over-riding that function in derived classes

Note:
- **Abstract class can be created without any abstract method as well however if any class has at-least 1 abstract method, we must declare class as abstract**
- **A derived class from an abstract base class should be set as abstract if it doesn't implements all the abstracted methods of base class**
- abstract method are written generally as "*public abstract* void action();"
- abstract method can be written without scope brackets (curly braces); method have semi colon at its end

When object of derived class is created, constructor of base class followed by constructor of derived class is invoked, called as Constructor chaining.

Interface
All fields (attributes) in interfaces are public, static and final by default.
All methods are public & abstract by default.

A class that implements an interface must implement all the methods declared in the interface.
Interfaces support the functionality of multiple inheritance.

Use keyword **implements** after classname to implement an interface.
Any property (attribute) of interface is accessed directly with using **this** since it is static & final by default

```
interface Animal {
  int eyes = 2;
  void walk();
}
class Dog implements Animal {
  //int eyes = 5;   //if declared this here, Dog object eyes will be 5; class members value has higher precedence
  public void walk() {
     System.out.println("Dog walks on 4 legs and has " + eyes + " eyes.");
  }
}
public class LearnInterface {
  public static void main(String args[]) {
    Dog dog = new Dog ();
    dog.walk();
  }
}
```

Additional Note
Class data members value has precedence over interface variable value.

Create & initialize static final variable say 'var1' and declare a method getValue(); in Interface.
Create a class that implements above Interface method getValue() and create data member with same name 'var1' & a constructor to initialise the value of data member.
Create an object of class and invoke getValue() method.
Output will be value initialized by class constructor, it won't be the value initialized by Interface variable bcoz class data members value has precedence over interface variable value.

```
abstract class Animal {
	String name;

	public Animal() {
		System.out.println("Animal constructor invoked");
		this.name = this.getClass().getName();
	}

	public abstract void action();

	public void displayInfo() {
		System.out.println("Class Name is " + this.name);
		this.action();
	}
}

class Dog extends Animal {
	public Dog() {
		System.out.println("Dog constructor invoked");
	}
	public void action() {
		System.out.println("Dogs Bark");
	}
}

class Lion extends Animal {
	public void action() {
		System.out.println("Lion Roar");
	}
}

public class LearnAbstraction {
	public static void main(String[] args) {
		Dog d = new Dog();
		d.displayInfo();
		System.out.println();
		Lion l = new Lion();
		l.displayInfo();
	}
}
```

```
Animal constructor invoked
Dog constructor invoked
Class Name is Dog
Dogs Bark

Animal constructor invoked
Class Name is Lion
Lion Roar
```

Polymorphism

26 May 2024 13:06

Poly means "many" and morphism means "forms". Polymorphism means existing in more than one forms.
Ability to present the same interface for differing underlying forms (data types).
With polymorphism, each of these classes will have different underlying data.

2 types:

1. Static (compile-time) Polymorphism: polymorphism which is implemented at the compile-time

Eg. Function Overloading [Allows to exhibit different functionality with same function name depending on number & data-type of arguments OR the order of arguments such that the data-type differs]. We cannot declare two methods with the same signature and different return types. It will throw a compile-time error.
Note Methods can't be overloaded just by adding static keyword to method; compile time re-define error

2. Dynamic (run-time) Polymorphism: polymorphism which is implemented at the run-time

Eg. Function Overriding [Child class contains method already present in parent class. Hence, the child class overrides the method of the parent class (method definition can may be different). The call to the function is determined at runtime is known as runtime polymorphism]

```
//Dynamic Polymorphism
class Shape {
  public void perimeter() {
     System.out.println("Displays Perimeter of Shape");
  }
}
class Triangle extends Shape {
  public void perimeter(int s1, int s2, int s3) {
     System.out.println("Perimeter is " + s1+s2+s3);
  }
}
public static void main(String[] args) {
        Triangle t = new Triangle()
        t.perimeter();              //"Displays Perimeter of Shape"
        t.perimeter(3, 5, 2);       //"Perimeter is 10
        //Depending on method argument count, data-type either base class or derived class method is called
        at runtime thereby exhibiting runtime polymorphism
}
```

```
//Compile-Time Polymorphism
class Brand {
        int uid;
        String name;

        //Parameterised Constructor
        Brand(int id, String brandName) {
                this.uid = id;
                this.name = brandName;
        }

        //Overloaded Method display()
        public void display() {
                sysout("UID: " + uid + "\tName: " + name);
        }
        public void display(int id) {
                sysout("UID: " + id);
        }
        public void display(String brandName) {
                sysout("Name: " + brandName);
        }
}

public static void main(String[] args) {
        Brand b = new Brand(5, "ENOVIA");

        b.display();                //"UID: 5    Name: ENOVIA"
        b.display(b.name);          //"Name: ENOVIA"
        b.display(b.uid);           //"UID: 5"
}
```

Inheritance

26 May 2024 14:55

An object acquires all the properties and behaviours of its parent object automatically.
Thru this, we can reuse, extend or modify the attributes and behaviours which are defined in other classes.

In Java, the class which inherits the members of another class is called derived class OR sub-class and the class whose members are inherited is called base class OR super-class.
The derived class is the specialized class for the base class.

4 types:

1. Single Inheritance: When one class inherits another class, it is known as single level inheritance
2. Hierarchical Inheritance: When we have more than one derived class from a single base-class
3. Multi-Level Inheritance: When a class is derived from another derived class
4. Hybrid Inheritance: A combination of simple, multiple inheritance and hierarchical inheritance
   If we use combination of a multi-level with an hierarchical inheritance, then it is achievable without need of interfaces. However, involving Multiple Inheritance in a Hybrid Inheritance also is not allowed without Interfaces.

5. Multiple Inheritance: NOT supported in Java, gives compilation error when we inherit 2 classes, even if different methods are present. In Java, *Multiple Inheritance* is achieved thru Interfaces.

Overloading Inherited Methods
In derived class, we are allowed to overload the inherited methods from the base class.
Such overloaded methods neither hide nor override the superclass methods — they are new methods, unique to the subclass.

Derived class has all functions & variables from base class.
In case of both class having same variable/function name, Local is preferred while we use super keyword to refer to base class

```
//Single-Level Inheritance
class Shape {
  public void perimeter() {
    System.out.println("Displays Perimeter of Shape");
  }
}
class Triangle extends Shape {
  public void perimeter(int s1, int s2, int s3) {
    System.out.println(s1+s2+s3);
  }
}

//Hierarchical Inheritance
class Shape {
  public void perimeter() {
    System.out.println("Displays Perimeter of Shape");
  }
}
class Triangle extends Shape {
  public void perimeter(int s1, int s2, int s3) {
    System.out.println(s1+s2+s3);
  }
}
class Square extends Shape {
  public void perimeter(int s) {
    System.out.println(4*s);
  }
}

//Multi-Level Inheritance
class Shape {
  public void perimeter() {
    System.out.println("Displays Perimeter of Shape");
  }
}
class Triangle extends Shape {
  public void perimeter(int s1, int s2, int s3) {
    System.out.println(s1+s2+s3);
  }
}
class EquilateralTriangle extends Triangle {
  public void perimeter(int s) {
    System.out.println(3*s);
  }
}

//Hybrid Inheritance
class Shape {
  public void perimeter() {
    System.out.println("Displays Perimeter of Shape");
  }
}
class Triangle extends Shape {
  public void perimeter(int s1, int s2, int s3) {
    System.out.println(s1+s2+s3);
  }
}
class Square extends Shape {
  public void perimeter(int s) {
    System.out.println(4*s);
  }
}
class EquilateralTriangle extends Triangle {
  public void perimeter(int s) {
    System.out.println(3*s);
  }
}
```

Try-catch-finally

14 March 2023 03:04 AM

If exception is thrown in finally block:
That exception propagates out and up, and will (can) be handled at a higher level.
Your finally block will not be completed beyond the point where the exception is thrown.
If the finally block was executing during the handling of an earlier exception then that first exception is lost if it was not yet handled.

HashMap

21 March 2023 03:18 AM

getOrDefault(key, defaultValue) will return value if key exists o/w return default value specified

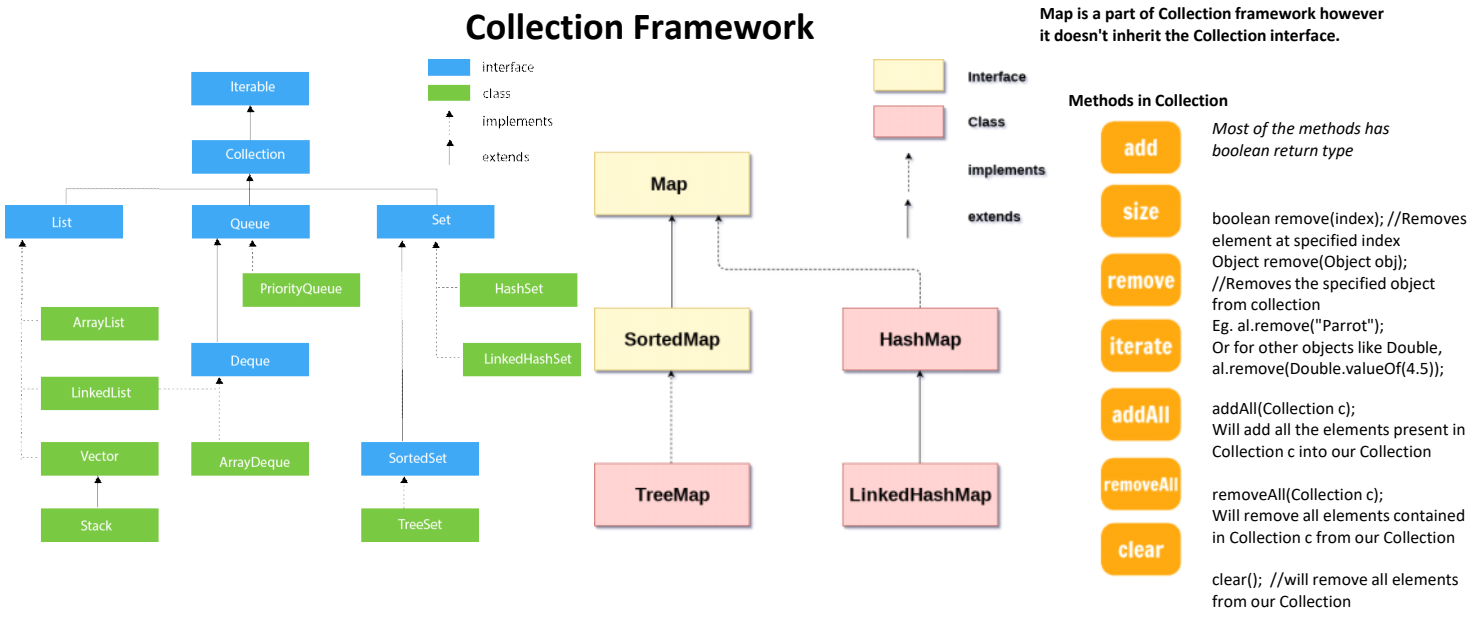
Collection Framework

27 May 2024 00:55

- Collection: A group of individual objects that represent a single entity is known as a collection. In Java, we've programming language then it will become Java Collection Framework.
- Collection Framework: To represent a group of objects as a single entity in the Java programming language we need classes and interfaces defined by the Collection Framework. In C++, you can compare Collection with Containers and Collection Framework with STL(Standard Template Library).
- Collection Interface: Interfaces specify what a class must do and not how. It is the blueprint of the class. It is the root interface of the Collection Framework that defines the most common methods that can be used for any collection objects. Or you can say it represents the individual object as a single entity.
- Collections Class: It is present in java.util package and is a member of Collection Framework. This class provides many utility methods for the collection object.

Difficulties were faced before java version 1.2 that lead birth to the concept of Collection:

1. Reduces the programming effort as the programmer does not have to worry about designing the Collection class. Also, abstraction is achieved by not writing the Collection class.
2. Increases program speed and quality as the programmer now is not worried about thinking best implementation for a specific data structure as of now for a programmer scope widens up and at the same time is now only worried about the best implementation that can drastically boost the performance of program irrespective of data structure constraints encountered earlier.
3. The API that implements these interfaces now has common sets of methods for any interfaces viz. Collection, Set, List, and Map.



ArrayList

14 March 2023 03:07 AM

Time Complexity

1. Add an element in end **add(data)**: O(1) [however in worst case, when dynamic array is full and new array is created & all element copied; its O(N)]
2. Insert in between of beginning **add(index, data)**: O(N)
3. Remove an element **remove(index) / remove(Object data)**: O(N) //further elements has to be shifted one place
4. Get (search/access) an element at a particular index **get(index)**: O(1)
5. Fetch index of an element **indexOf(data) / lastIndexOf(data)**: O(N) //each element is checked one-by-one
6. Check for presence of an element **contains(data)**: O(N)

Part of Java Collection Framework

- Array is fixed whereas ArrayList is variable size.
- Array can be used to store both primitive and non-primitive (object) data-types whereas ArrayList is used to store only objects type.

Eg. To store an int value, we use Wrapper class called as Integer class which is object. This is bcoz for each new object new memory is allocated and we can add that to ArrayList whereas for primitive types, memory is allocated first before it is defined.

- ArrayList is stored in Heap.

ArrayList Implementation is done using dynamic array, such that when array gets full, the entire data has to be copied to another dynamic array of larger (twice) size. Also to insert an element in between or in beginning (called as add(index, data)), the entire data has to be shifted by one place. Hence, insert is O(N) for average case whereas for insert at end (called as add(data)), it is O(1).

```
import java.util.ArrayList;

ArrayList<Integer> al = new ArrayList<>();
```

Elements are copied as reference
[Change in one affects the other]

```
ArrayList<Integer> gfg=new ArrayList<>();
ArrayList<Integer> gfg2=gfg;
```

Elements are copied as values
[Change in one won't affect the other]

```
ArrayList<Integer> gfg=new ArrayList<>();
ArrayList<Integer> gfg2=new ArrayList<>(gfg);
```

Array of ArrayList

```
ArrayList<Integer> al[] = new ArrayList[size];
for(int i=0; i<size; ++i)
al[i] = new ArrayList<Integer>();
```

Operations in ArrayList

1. Add an element

```
al.add(data);
al.add(index, data); //To insert/add at specific index
```

2. Get an element

```
al.get(index);
```

3. Remove/Delete an element

```
al.remove(index);   //al.remove(2); removes index 2 element
al.remove(Object data);
//al.remove(Integer.valueOf(5)); OR al.remove("Parrot");
```

4. Set an Element

```
al.set(index, data);  //To update the data at specific index
```

5. Print entire ArrayList

```
System.out.println(al);
```

6. Count of elements in ArrayList

```
al.size();
```

7. Iterate the ArrayList

Use a for loop from 0 to size of arraylist and fetch i th element of array list using al.get(i);
OR use for-each loop
Eg. `for(Integer i : al) {  sysout(i); }`

8. Check if an element is present in ArrayList

```
al.contains(data);  //true OR false
```

9. Searching for elements

```
al.indexOf(data);  //gives index of 1st occurrence of data in ArrayList
al.lastIndexOf(data); //gives last index of data in ArrayList
```

Sorting ArrayList

```
import java.util.Collections;
Collections.sort(al);  //Sorts in ascending order
//can be used for all class (Set, Map, List) under Collections framework
```

LinkedList

27 May 2024 15:02

- Variable Size
- Non-contiguous memory

LinkedList is made up of Nodes. Each Node has a data and a reference (address) to the next Node.
The 1st Node is called **Head** and last Node which has next node reference as *null* is called **Tail**

Node 1 has its data and info about location of Node 2
..
..
..
Last Node N has its data and since no further Node present, it's points to **null** Node.

New Functions in LinkedList
ll.addFirst(data);
ll.addLast(data);
l.removeFirst();
ll.removeLast();

LinkedList vs ArrayList

Insert /remove operations
LinkedList has O(1). ArrayList has O(N) generally but if inserted at end then O(1).
LinkedList is best when we need frequent manipulation (insert/remove) provided we've reference of the Node right before where manipulation is needed. [In ArrayList, irrespective of address, all data has to be shifted]

Search (**get()**) operations: ArrayList has O(1). LinkedList has O(N).
ArrayList is better when frequently searching/accessing value (based on index) but NOT manipulation (insert/remove)

3 Types of LinkedList

1. Singly LinkedList (Normal LinkedList)

Each Node has reference to next Node and last Node has reference as null

2. Doubly LinkedList

Each Node has reference to next as well as previous Node, wherein head has previous Node reference as null

3. Circular LinkedList

Same as Singly LinkedList except that last node (tail) has reference of 1st Node (head)

Ques: Write a method for LinkedList to delete all nodes which have values greater than any value, say X and return the head

```
Node removeGreaterNodes(int x) {

	while(head != null && head.data > x) {
		head = head.next;
	}

	if(head != null) {
		Node currNode = head;
		Node nextNode = head.next;

		while(nextNode != null) {
			if(nextNode.data > x)
				currNode.next = nextNode.next;
			else
				currNode = currNode.next;

			nextNode = currNode.next;
		}
	}

	return head;
}
```

Reference: https://stackoverflow.com/questions/43310129/deleting-nodes-greater-than-specified-value-from-linked-list
Problem: https://leetcode.com/problems/remove-linked-list-elements/

Reverse a LinkedList
Using Collections utility method: ll.reverse();

OR write your own:

```
//No extra memory usage except constant i.e. O(1) space
//Time: O(N)
public void reverseLinkedList() {
	if(head == null || head.next == null) {
	  return;
	}

	Node prevNode = null;
```

Time Complexity

1. Add/Remove: O(1) [assuming we've reference of prev node]
2. Get an element: O(N)

```
public class LinkedListScratch {

	Node head;
	private int size;

	LinkedListScratch() {
		size = 0;
	}

	/* For better streamlining, since Node is to be used with LinkedList
	 * only, we write the Node class inside the LinkedList class
	 * Since, we don't need non-static methods/members of LinkedList class
	 * to be accessed in Node class, we can declare Node class as static too
	 * Note: Node class can be written outside also. However, size variable
	*  will be in-accessible; manually change size in each method
	 */
	class Node {
		String data;
		Node next;

		Node(String data) {
			this.data = data;
			this.next = null;
			++size;
		}
	}

	public void addFirst(String data) {
		Node newNode = new Node(data);
		newNode.next = head;
		head = newNode;
	}

	void addLast(String data) {
		Node newNode = new Node(data);

		if(head == null) {
			head = newNode;
			return;
		}

		Node lastNode = head;

		while(lastNode.next != null)
			lastNode = lastNode.next;

		lastNode.next = newNode;
	}

	void printList() {
		Node currNode = head;

		while(currNode != null) {
			System.out.print(currNode.data + " ");
			currNode = currNode.next;
		}
		System.out.println();
	}

	void removeFirst() {
		if(head == null) {
			System.out.println("Empty LinkedList");
			return;
		}

		head = head.next;
		--size;
	}

	void removeLast() {
		if(head == null) {
			System.out.println("Empty LinkedList");
			return;
		}

		if(head.next == null) {
			head = null;
			--size;
			return;
		}

		Node currNode = head;
		Node lastNode = currNode.next; //Note: the data of lastNode
		won't be updated based on currNode

		while(lastNode.next != null) {
			currNode = currNode.next;
			lastNode = lastNode.next;
		}

		currNode.next = null;
```

```
            return;
        }

        Node prevNode = null;
        Node currNode = head;

        while(currNode != null) {
            Node nextNode = currNode.next;
            currNode.next = prevNode;

            prevNode = currNode;
            currNode = nextNode;
        }

        head = prevNode;
}
```

Logic: Take head as current node and a prevNode will be null initially.
Iterate till current node is not null, take nextNode as the next of current node.
Change the next of current node to point to prevNode.
Now, set the prevNode as current node; current node as the nextNode.
At last after the loop, set head to the node before the current node i.e. prevNode (tail node of original list), since current node is now a null.
https://leetcode.com/problems/reverse-linked-list/submissions/
https://www.geeksforgeeks.org/reverse-a-linked-list/

Recursive Approach to Reverse *(Here space complexity is O(N), Time same as Iterative approach above O(N))*

```
public Node reverseRecursiveLinkedList(Node head) {
        if(head == null || head.next == null) {
          return head;
        }

        //newHead will always be same i.e. original list tail
        //but we call this recursively to reverse each sub-linked list
        Node newHead = reverseRecursiveLinkedList(head.next);

        head.next.next = head;
        head.next = null;

        return newHead;
}

//In main method
rll.head = rll.reverseRecursiveLinkedList(rll.head);  //update original head vid head returned
rll.printList();
```

Logic: Handle empty or single node case, since not possible to reverse
Using recursion, we'll re-invoke the method recursively to reverse the list starting from the next node of the current head
Now, as next of our 2nd Node will point to 3rd node, we'll change it to point to our 1st Node:
Node node1 = head; Node node2 = head.next;   node2.next = node1;
Also, we've currently node1 next still pointed to node2 which needs to be removed:
node1.next = null;
Finally, return the value of the head of reversed list which we got from recursive call i.e. newHead
Reverse a Linked List | Iteratively & Recursively | DSA-One Course #37

Remove n th element from end of LinkedList
https://leetcode.com/problems/remove-nth-node-from-end-of-list

```
 public ListNode removeNthFromEnd(ListNode head, int n) {

    if(head.next == null) return null;

    int size = 0;

    ListNode curr = head;
    while(curr != null) {
      curr = curr.next;
      ++size;
    }

            if(n == size) {
                head = head.next;
                return head;
            }

            int posFromBeg = size - n + 1;
            int prevNodePos = posFromBeg - 1;

            ListNode currNode = head;

            for(int i=1; i<prevNodePos; ++i) {
                currNode = currNode.next;
            }

            currNode.next = currNode.next.next;
    return head;
  }
```

Time complexity - O(n) Space complexity - O(1)

Check for LinkedList is Palindrome or not

```
            }

            currNode.next = null;

            --size;
        }

        int size() {
            return this.size;
        }

        void addInMiddle(int index, String data) {

            if(index < 0 || index > size) {
                System.out.println("Invalid Index");
                return;
            }

            Node newNode = new Node(data);

            if(head == null)
                return;

            if(index == 0) {
                newNode.next = head;
                head = newNode;
                return;
            }

            Node currNode = head;

            for(int i=0; i<index-1; ++i) {
                currNode = currNode.next;
            }

            newNode.next = currNode.next;
            currNode.next = newNode;
        }

        public static void main(String[] args) {

            LinkedListScratch ll = new LinkedListScratch();

            ll.addFirst("55");
            ll.addLast("ooo");
            ll.addFirst("hggh");
            ll.addLast("b");
            ll.addFirst("89");
            ll.addLast("222");

            ll.printList();

            ll.removeFirst();
            ll.printList();

            ll.removeLast();
            ll.printList();

            System.out.println(ll.size());

            ll.addInMiddle(10, "QWERTY");
            ll.printList();

            ll.addInMiddle(2, "QWERTY");
            ll.printList();

            ll.addInMiddle(0, "QWERTY");
            ll.printList();

            ll.addInMiddle(6, "QWERTY");
            ll.printList();

            ll.addInMiddle(7, "QWERTY");
            ll.printList();
        }
}
```

```
89 hggh 55 ooo b 222
hggh 55 ooo b 222
hggh 55 ooo b
4
Invalid Index
hggh 55 ooo b
hggh 55 QWERTY ooo b
QWERTY hggh 55 QWERTY ooo b
QWERTY hggh 55 QWERTY ooo b QWERTY
QWERTY hggh 55 QWERTY ooo b QWERTY QWERTY
```

Reverse nodes b/w a start & end position in LinkedList
Reverse Linked List II - Leetcode 92 - Python

Check for LinkedList is Palindrome or not

Approach #1: O(N) for both Time & Space
Create an array/arraylist etc and save all data one by one of LinkedList check if they're palindrome.
OR Create a new LinkedList by reversing the Linkedlist and compare the Node of each element. They should be same for being Palindrome

Approach #2: O(N) for Time, O(1) for space
Find middle node. (use 2 pointers, one move 1 step & other moves 2 steps; till next & next to next of fast pointer is not null, keep going. At last slow will reach to mid node & fast reaches last or a node before last)
Reverse the linkedlist starting after the middle node
Compare the node for equality

```
public Node reverseLinkedList(Node headNode) {
	if(headNode == null || headNode.next == null) {
	  return headNode;
	}

	Node prevNode = null;
	Node currNode = headNode;

	while(currNode != null) {
		Node nextNode = currNode.next;
		currNode.next = prevNode;

		prevNode = currNode;
		currNode = nextNode;
	}

	headNode = prevNode;
	return headNode;
}

//Import code of rabbit, turtle strategy
public Node getMidNode() {
	Node slow = head;
	Node fast = head;

	while(fast.next != null && fast.next.next != null) {
		slow = slow.next;
		fast = fast.next.next;
	}

	return slow;
}

//Time: O(N); Space: O(1)
public boolean isPalindrome() {
	if(head == null || head.next == null) return true;

	Node midNode = getMidNode();

	//If odd number of nodes, the midNode will be the middle one
	//and we'll take it in 1st half; if even nodes [1,2,3,4], midNode will return 2
	Node midReversedList = reverseLinkedList(midNode.next);

	Node originalList = head;

	while(midReversedList != null) {

		if(originalList.data != midReversedList.data)
			return false;

		midReversedList = midReversedList.next;
		originalList = originalList.next;
	}

	return true;
}
```

Detect whether there exists a loop or cycle in a LinkedList
All Approaches: https://www.geeksforgeeks.org/detect-loop-in-a-linked-list/ (Hashing approach ko HashMap se hi kro, its sufficient)

Floyd's Cycle Detection Algorithm (Turtle & Hare Approach): Time O(N), Space O(1)

```
//Time O(N), Space O(1)
public boolean hasLoop() {
	if(head == null) return false;
	if(head.next == head) return true;

	//Floyd's Turtle (fast) & Hare (slow) Algo
	//Turtle goes 1 step a time whereas Hare goes 2 steps at a time
	//If there exists a loop, at some point the Turtle will meet Hare
	Node slow = head;
	Node fast = head;

	while(fast.next != null && fast.next.next != null) {
		slow = slow.next;
		fast = fast.next.next;
```

Reverse nodes b/w a start & end position in LinkedList
Reverse Linked List II - Leetcode 92 - Python
Reverse Linked List 2 (LeetCode 92) | Full simplified solution | Animations and Demo

```
//Reverse b/w startPos and endPos, both inclusive
//Time: O(N), Space: O(1)
public void reverseLinkedListBetween(int startPos, int endPos) {

	if(head == null || head.next == null || startPos == endPos)
		return;

	//Necessary to handle case where start position is 1
	//Create a dummy node whose next points to head
	Node tempNode = new Node("-1");
	tempNode.next = head;

	Node prevToStartPosNode = tempNode;
	Node startPosNode = head;

	//Find the Node from where reversal has to be done
	for(int i=1; i<startPos; ++i) {
		prevToStartPosNode = startPosNode;
		startPosNode = startPosNode.next;
	}

	// 'startPosNode' is the node from which reversal has to be started
	// Keep this Node & 'prevToStartPosNode' unaffected, so that it can be later used to link to Node after the end position
	Node startingPositionNode = startPosNode;

	/*
	 * Reversing the LinkedList starting from startPosNode now
	 */
	//Currently, assume we'll reverse list without connecting to Nodes before the startPos &
	//we'll start with current node as starting position node i.e. startPosNode
	Node prev = null;
	Node curr = startPosNode;

	//Now reverse list number of times as number of elements (endPos-startPos+1)
	//Eg. startPos=3, endPos=6;  No of elements to change: 3,4,5,6 = 4
	//bcoz the first node itself also has to be updated with
	//next as null and then further nodes
	for(int i=1; i<=endPos-startPos+1; ++i) {
		Node next = curr.next;
		curr.next = prev;
		prev = curr;
		curr = next;
	}

	//After reversal, the prev Node has the new head node
	//Now, we'll point this new head node (prev) to next of Node which was previous to startPosNode
	prevToStartPosNode.next = prev;

	//After reversal, the next of original head points to null
	//hence, next of actual startPosNode has been set to null using curr above
	//So, now we've to link it to Node after the Node of ending position, which is now curr
	startingPositionNode.next = curr;

	//needed bcoz in above two statements head will be lost if startPos=1
	head = tempNode.next;
}
```

Remove cycle, if there exists a loop or cycle in a LinkedList
Detect & Remove Cycle in a Linked List | Floyd's Cycle Detection Algorithm | DSA-One Course #39
https://www.geeksforgeeks.org/detect-and-remove-loop-in-a-linked-list/
We can detect cycle using HashMap also but space extra used.
Also, point where any existing key is next of current Node, cycle exists at that Node.

Approach: Floyd's

```
//To remove the loop/cycle
//Find node which causes cycle & set its next as null
public void removeLoop() {
	getCycleNode().next = null;
}

//Take two Nodes, the head Node and the Node where fast & slow meet
//Now start going next for each of them & Node where they're same, it's cycle Node
public Node getCycleNode() {

	if(head == null || head.next == null) return null;

	Node slow = head;
	Node fast = head;
```

```
        while(fast.next != null && fast.next.next != null) {
            slow = slow.next;
            fast = fast.next.next;

            //Comparison done after moving once bcoz initially both point to head Node
            if(slow == fast)
                return true;
        }

        return false;
}
```

Swap Nodes in Pairs
https://leetcode.com/problems/swap-nodes-in-pairs/

Given a linked list, swap every two adjacent nodes and return its head.
1->2->3->4->5->6 should become 2->1->4->3->6->5

```
//Time: O(N), Space: O(1)
//IMPORTANT: In cases where head of original is possible to change, create a temp node
//whose next points to head (or respective Node) of the original list; return it at end by temp.next
public Node swapPairs(Node head) {
        if(head == null || head.next == null) return head;

        //As per logic, original head.next is the final head which we'll have to return
        Node temp = new Node("0");
        temp.next = head.next;

        //Pair of Node A, Node B which after swapping
        //Node A ka next will point to 1st Node of next pair
        //Node B ka next will point to Node A of current pair
        //However, the previous pair's 2nd Node is still pointing
        //to Node A which is losing Node B in chain, as it's Node B
        //whose next is point to our Node A;
        //so update previous pairs 2nd Node ka next as Node B of current pair
        Node prev = new Node("0");

        Node node1 = head;
        Node node2 = head.next;

        do {
            Node node3 = node2.next;
            node2.next = node1;
            node1.next = node3;

            //As mentioned earlier, updating previous pair's 2nd Node as Node 2 of current pair
            prev.next = node2;

            if(node3 == null || node3.next == null) break;

            //Updating 2nd Node of the latest previous pair in prev
            prev = node1;

            node1 = node1.next;
            node2 = node1.next;
        }while(true);

        return temp.next;
}
```

```
        //if(head == null || head.next == null) return null;

        Node slow = head;
        Node fast = head;

        //For conditions where only one or two non-cycle nodes are present,
        //slow and fast keep pointed to head and gives slow == head and wrong result
        boolean hasCycle = false;

        while(fast.next != null && fast.next.next != null) {
            slow = slow.next;
            fast = fast.next.next;

            if(slow == fast) {
                hasCycle = true;
                break;
            }
        }

        if(!hasCycle) return null;

        Node headNode = head;
        Node meetingNode = slow;

        while(headNode != meetingNode) {
            headNode = headNode.next;
            meetingNode = meetingNode.next;
        }

        return meetingNode;
}
```

Stack

02 June 2024 02:29

Add (Push) / Remove (Pop) / Read (Peek) an element (normally stack operation is done at Head position): O(1) **Time Complexity**

Stack of chairs.

New chair can be added (push) only on top.
Chairs can be removed (pop) from top only.
Only top-most chair details is seen (peek).

LIFO: Last-in, First-out

3 operations:
Push (Add)
Pop (Remove)
Peek (View topmost element)

In Java, pop() will remove the top-most element and return it as well.

pop(), peek() on an empty stack throws **java.util.EmptyStackException**

```
//Implementation using Java Collections Stack class
import java.util.Stack;

public class StackClass {

	//Method to add an element at bottom of stack
	//Time: O(N), Space: O(N) Recursive calls uses implicit stack to store data
	public static void pushAtBottom(Stack<Integer> st, int data) {
		if(st.isEmpty()) {
			st.push(data);
			return;
		}

		int top = st.pop();
		pushAtBottom(st, data);

		st.push(top);
	}

	//Method to reverse stack
	//Time: O(N^2), Space: O(N)
	public static void reverse(Stack<Integer> st) {
		if(st.isEmpty())	return;

		int top = st.pop();
		reverse(st);
		//this recursive call is just to reach till stack is empty without
		//need to count number of elements and then once it's empty,
		//start pushing popped element at bottom in each recursive call

		pushAtBottom(st, top);

		//This also works however interviewer expects us to know above
//		for(int i=0; i<st.size(); ++i) {
//			int top = st.pop();
//			pushAtBottom(st, top);
//		}
	}

	public static void main(String[] args) {
		Stack<Integer> st = new Stack<>();
		st.push(5);
		st.push(17);
		st.push(12);
		pushAtBottom(st, 45);

		while(!st.isEmpty()) {
	 System.out.println(st.pop());
	}
		System.out.println();
		st = new Stack<>();
		st.push(1);
		st.push(2);
		st.push(3);
		st.push(4);
		st.push(5);
		reverse(st);

		while(!st.isEmpty()) {
	 System.out.println(st.pop());
	}
	  }
}
```

Logic:
To add at bottom, we start popping topmost element & storing in implicit recursion stack and once we reach bottom, we push the element to be added at bottom. Further, we push back the popped element to stack.

To reverse stack, recursive loop till we reach end of stack and keep pushing the

```
//Stack implementation using LinkedList
public class StackClassLL {
	Node head;
	class Node {
		int data;
		Node next;
		Node(int data) {
			this.data = data;
			this.next = null;
		}
	}
	public boolean isEmpty() {
		return head == null;
	}
	public void push(int data) {
		Node newNode = new Node(data);
		if(isEmpty()) {
			head = newNode;
			return;
		}
		newNode.next = head;
		head = newNode;
	}
	public int pop() {
		if(isEmpty())	return -1;
		int top = head.data;
		head = head.next;
		return top;
	}
	public int peek() {
		if(isEmpty())	return -1;
		return head.data;
	}
	public static void main(String[] args) {
		StackClassLL st = new StackClassLL();
		System.out.println(st.peek());
		st.push(5);
		System.out.println(st.peek());
		st.push(2);
		st.push(8);
		System.out.println(st.peek());
		System.out.println(st.pop());
		System.out.println(st.pop());
		System.out.println(st.pop());
		System.out.println(st.pop());
	}
}
```

```
//Stack implementation using ArrayList
import java.util.ArrayList;

public class StackClassAL {
	//static class can be directly accessed in
	main method()
	static class Stack {
		ArrayList<Integer> list = new
		ArrayList<Integer>();

		public void push(int data) {
			list.add(0, data);
		}

		public int pop() {
			if(list.size() == 0)
				return -1;

			return
			list.remove(list.size()-1);
		}

		public int peek() {
			if(list.size() == 0)
				return -1;

			return list.get(list.size()-1);
		}
	}

	public static void main(String[] args) {
		Stack st = new Stack();
		System.out.println(st.peek());
		st.push(5);
		st.push(17);
		st.push(12);
		System.out.println(st.peek());
		System.out.println(st.pop());
		System.out.println(st.peek());
		System.out.println(st.pop());
		System.out.println(st.peek());
	}
}
```

top-most popped element at bottom of stack

HashSet

07 June 2024 01:18

Collection of unique data.
Set doesn't allows duplicates.

HashSet
Allows storing unique values (values are always unique based on their hash code)
Order of insertion not persisted.
NULL is allowed.

3 main operations:
Insert an Element - **add()**: O(1)
Delete an Element - **remove()**: O(1)
Search an Element - **contains()**: O(1)

Other methods:
size()
clear()
isEmpty()
equals() - To compare whether two objects are set & elements also same

Traversing a HashSet
Since Set doesn't have any index, we use the concept of Iterator to traverse the element of a HashSet.

```
Iterator itr = hs.iterator();

while(itr.hasNext()) {
   System.out.println(itr.next());
}
```

```
import java.util.HashSet;
import java.util.Iterator;

public class HashSetClass {

	public static void main(String[] args) {

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

		hs.add(5);
		hs.add(7);
		hs.add(2);
		hs.add(2);
		hs.add(11);
		System.out.println(hs);
		hs.remove(7);
		System.out.println(hs.contains(2));
		System.out.println(hs);
		System.out.println();

		//Traversing using iterator
		Iterator itr = hs.iterator();
		while(itr.hasNext())
			System.out.print(itr.next() + " ");
		System.out.println();

		//Traversing using for-each loop
		for(Integer i : hs)
			System.out.print(i + " ");
	}
}
```

```
[2, 5, 7, 11]
true
[2, 5, 11]

2 5 11
2 5 11
```

Size of Union of two arrays

```
//Time: O(N), Space: O(N)
public static int union(int arr1[], int arr2[]) {
	HashSet<Integer> unique = new HashSet<>();

	for(int i : arr1)
		unique.add(i);
	for(int i : arr2)
		unique.add(i);

	return unique.size();
}

int arrA[] = {7, 3, 9};
int arrB[] = {6, 3, 9, 2, 9, 4};
System.out.println(union(arrA, arrB)); //6
```

Size of Intersection of two arrays

```
public static int intersection(int arr1[], int arr2[]) {
	HashSet<Integer> unique = new HashSet<>();
	int intersectionSize = 0;
	for(int i : arr1)
		unique.add(i);
	for(int i : arr2) {
		if(unique.contains(i)) {
			//To avoid matching in case an
			//element is present twice in array
			//Without this output is 3 since 9 is
			//present twice in 2nd array
			unique.remove(i);
			++intersectionSize;
		}
	}

	return intersectionSize;
}

int arrA[] = {7, 3, 9};
int arrB[] = {6, 3, 9, 2, 9, 4};
System.out.println(intersection(arrA, arrB)); //2
```

Get Itinerary (path from source to destination)

```
public static void getItinerary(HashMap<String, String> hm) {
	/*
	"Chennai", "Bengaluru", "Mumbai", "Delhi"
	"Goa", "Chennai", "Delhi", "Goa"

	Approach:
	Starting city will be a city which is not present in 'value' side.

	Implementation:
	Create a set of all values of HashMap and save as a HashSet. Now, fetch a key of
	HashMap which doesn't exist in HashSet; it will be starting city, say 'city'.
	Next, proceed to print the 'city' and fetch whether 'value' of this 'city' key exists.
	Further, re-assign the 'city' key as the 'value' associated to this city.
	Keep finding 'value' of each 'city' key till HashMap contains the 'key' i.e. key=value
	pair exists
	*/
	HashSet<String> hs = new HashSet<>(hm.values());
	String city = "";

	for(String s : hm.keySet()) {
		if(!hs.contains(s)) {
			city = s;
			break;
		}
	}

	System.out.print(city);

	while(hm.containsKey(city)) {
		city = hm.get(city);
		System.out.print(" -> " + city);
	}
}

HashMap<String, String> cityHM = new HashMap<>();
cityHM.put("Chennai", "Bengaluru");
cityHM.put("Mumbai", "Delhi");
cityHM.put("Goa", "Chennai");
cityHM.put("Delhi", "Goa");

getItinerary(cityHM);
//Mumbai -> Delhi -> Goa -> Chennai -> Bengaluru
```

Another approach, create a reverse HashMap of the same since in this problems case generally values as also unique in HashMap

HashMap

07 June 2024 01:59

Store as key-value pairs format.

Has 4 constructors:
HashMap() //default value for initialCapacity (buckets) is 16, loadFactor is 0.75
HashMap(int initialCapacity)
HashMap(int initialCapacity, float loadFactor)
HashMap(Map map)

3 main operations:
Insert an Element - **put**(key, value): O(1)
Delete the Element - **remove**(key): O(1)
Search an Element - **containsKey**(key): O(1)

Has unique KEYS and can have only one **null** key. Duplicate values are allowed.
No order persisted.
Values will get updated if existing key is re-inserted vid new values.

Internal Implementation of HashMap
HashMap is stored using Array of LinkedList type. *LinkedList<> linkList[] = new LinkedList[N];*
Each index of this Array is called Bucket.

0	1	2	3	<--- Index of Array (aka Bucket)
LinkedList 1	LinkedList 2	LinkedList 3	LinkedList 4	
Key/Value Pair 1	Key/Value Pair 6	Key/Value Pair 4	Key/Value Pair 2	
|		|	|	
|		|	|	
Key/Value LinkedList Pair 3		Key/Value Pair 8	Key/Value Pair 7	<--- Nodes of each
|			|	
|			|	
Key/Value Pair 5			Key/Value Pair 9	

Initially, the array is created for any size, say **N**; each element of array is a LinkedList.
The index of array i.e. each LinkedList is called as Bucket.

Data is stored in key, value pair as a Node of LinkedList from any one of the Bucket. To determine, which Bucket (array index) is to be used, a hashing function is used to return an index b/w 0 to N (N = size of array) for each key whose key value pair is to be added.

Internally, the hashing function has to use an *int **hashCode()*** method of *Object class* to generate a unique integer called as HashCode for each input data.
(Hashing always give same output for same input data)
Eg. *String data = "Test"; int data2 = 45; int code1 = data.hashCode(); int code2 = data2.hashCode();*

The HashCode can be a +ve or -ve integer, hence take Math.abs() for +ve value and then in order to get the array index for us, we take modulo of this by N
key.hashCode() % N

At index returned by Hashing where first Node is already present, phenomena is called as **Hash-Collision.**
First we search in that bucket whether the key exists in any of the Node of the LinkedList or not.
If key exists, the value is updated o/w the key-value pair is added as the new Node of the LinkedList.

Whenever a key-value pair is added, ensure that the constant lambda (λ) = n/N *(where n is number of key-value pairs and N is size of array/number of buckets)* is not exceeding our loadFactor (default 0.75).
[loadFactor means Array size will be increased when 75% of the bucket is filled i.e. n >= 75% of N]
If it exceeds, re-create another array of LinkedList with larger (2*N/3*N depends on Java version) size and re-hash each keys to get new bucket index based on newer array in order to re-save the pairs in the new array.

Default value for initialCapacity (bucket/array size) is 16, loadFactor is 0.75; then 75% of N = 0.75*16 = 12.0
Hence, when n>12, size is doubled

*Note: Although, hashCode() remains same for keys but since N is changed, **key.hashCode() % N** also changes*

JAVA 8 has enhancement to store as self-balancing Binary Search Tree instead of LinkedList

Find element present more than n/3 times

```
//Time: O(N), Space: O(N)
public static void findMajorityElements(int arr[]) {
	int n = arr.length;

	HashMap<Integer, Integer> hm = new HashMap<>();

	for(int i : arr) {
```

```
hm.put("QWERTY",5);
hm.put("INDIA",2);
hm.put("QWERTY",71);
hm.remove("QWERTY");
hm.containsKey("INDIA");
```

Note:
put() & remove() method returns the older value which is already associated to the specified key and null if it doesn't existed

Traversing a HashMap

```
for(Integer key : hm.keySet())
  System.out.println(key+ " " + hm.get(key));

 //Iteration (1)
 for(Map.Entry<Integer, Integer> e : hm.entrySet())
     System.out.println(e.getKey() + " " + e.getValue());

//Iteration (2)
Set<Integer> keys = hm.keySet();
for(Integer key : keys)
   System.out.println(key+ " " + hm.get(key));
```

```
import java.util.ArrayList;
import java.util.LinkedList;

public class HashMapScratch {
	//K, V is necessary so that we can make the class of generic data-type
	static class HashMap<K, V> {
		class Node {
			K key;
			V value;

			Node(K key, V value) {
				this.key = key;
				this.value = value;
			}
		}

		private int n;   //Number of Key-Value Pairs
		private int N;   //Number of Buckets
		private LinkedList<Node> buckets[]; //Array of LinkedList (Buckets)
		private double loadFactor;

		@SuppressWarnings("unchecked")
		HashMap() {
			this.n = 0;
			this.N = 4; //Taking initial array size (no of buckets) as 4

			//Initializing the array of LinkedList of size N
			buckets = new LinkedList[N];

			//Initializing a LinkedList in each bucket
			for(int i=0; i<this.N; ++i)
				buckets[i] = new LinkedList<>();

			this.loadFactor = 0.75;
		}

		private int getBucketIndex(K key) {
			int hashCode = Math.abs(key.hashCode());
			return hashCode % this.N;
		}

		private int searchLL(K key, int bucketIndex) {
			LinkedList<Node> ll = buckets[bucketIndex];

			for(int i=0; i<ll.size(); ++i) {
				if(ll.get(i).key == key)
					return i;
			}

			return -1;
		}

		private void reHash() {
			LinkedList<Node> tempBucket[] = this.buckets;

			this.N = this.N * 2;
			this.buckets = new LinkedList[this.N];

			for(int i=0; i<this.N; ++i)
				this.buckets[i] = new LinkedList<>();

			for(int i=0; i<tempBucket.length; ++i) {
				LinkedList<Node> ll = tempBucket[i];

				for(int j=0; j<ll.size(); ++j) {
```

```
        int n = arr.length;

        HashMap<Integer, Integer> hm = new HashMap<>();

        for(int i : arr) {
            if(hm.containsKey(i))
                hm.put(i, hm.get(i)+1);
            else
                hm.put(i, 1);
        }

        for(Integer i : hm.keySet()) {
            if(hm.get(i) > n/3)
                System.out.print(i + " ");
        }
    }

    int arr[] = {1,3,2,5,1,3,1,5,1};
    findMajorityElements(arr);  //1
    System.out.println();
    int arr2[] = {1,2};
    findMajorityElements(arr2); //1 2
```

```
                for(int i=0; i<tempBucket.length; ++i) {
                    LinkedList<Node> ll = tempBucket[i];

                    for(int j=0; j<ll.size(); ++j) {
                        Node tempNode = ll.get(j);
                        //this.buckets[i].add(new Node(tempNode.key, tempNode.value));
//Not using this, as we've to save data w.r.t new bucket index returned based
on new array size
                        put(tempNode.key, tempNode.value);

                    }
                }

            }

            public void put(K key, V value) {
                //Get bucket index based on HashCode of key
                int bucketIndex = getBucketIndex(key);

                //Check whether key is present at any index in the LinkedList
                int searchNodeIndex = searchLL(key, bucketIndex);

                if(searchNodeIndex == -1) {
                    //Add new pair as a Node of LinkedList
                    buckets[bucketIndex].add(new Node(key, value));
                    ++n;        //Increase the number of pairs

                    double lambda = (double) this.n / this.N;
                    if(lambda > this.loadFactor)
                        reHash();
                }
                else
                    buckets[bucketIndex].get(searchNodeIndex).value = value;
    b               //Update the value of the pair in Node of LinkedList
            }

            public boolean containsKey(K key) {
                int bucketIndex = getBucketIndex(key);
                int searchNodeIndex = searchLL(key, bucketIndex);
                if(searchNodeIndex == -1)  return false;
                else return true;
            }

            public V remove(K key) {
                int bucketIndex = getBucketIndex(key);
                int searchNodeIndex = searchLL(key, bucketIndex);

                if(searchNodeIndex == -1)  return null;
                else {
                    Node node = buckets[bucketIndex].remove(searchNodeIndex);
                    --n;
                    return node.value;
                }
            }

            public V get(K key) {
                int bucketIndex = getBucketIndex(key);
                int searchNodeIndex = searchLL(key, bucketIndex);
                if(searchNodeIndex == -1)  return null;
                else
                    return buckets[bucketIndex].get(searchNodeIndex).value;
            }

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

            public ArrayList<K> keySet() {
                ArrayList<K> keys = new ArrayList<>();

                for(int i=0; i<this.N; ++i) {
                    LinkedList<Node> ll = buckets[i];
                    for(int j=0; j<ll.size(); ++j)
                        keys.add(ll.get(j).key);
                }

                return keys;
            }
        }

        public static void main(String[] args) {

            HashMap<String, Integer> map = new HashMap<>();
            map.put("India", 190);
            map.put("China", 200);
            map.put("UK", 75);
            map.put("Bangladesh", 250);
            map.put("US", 50);

            System.out.println(map.containsKey("USA"));
            System.out.println(map.containsKey("US"));
```

```
            System.out.println(map.get("China"));
            map.put("UK", 60);

            for(String key : map.keySet())
                  System.out.println(key + ", " + map.get(key));

            map.remove("China");
            System.out.println();
            for(String key : map.keySet())
                  System.out.println(key + ", " + map.get(key));
      }
}
```

Time Complexity

27 May 2024 15:54

In general, we take Average Case time-complexity

Complexity (Best to Worst)
==========
O(1) < O(log n) < O(n) < O(n log n) < O(n^2) < O(2^n) < O(n!)

O(1) doesn't mean 1 operation only
This means that a constant number of operations which is independent on number of elements in a data-structure.

Collections
==========

List

ArrayList
Add/Remove an element in end: O(1)
Add/Remove an element in beginning or middle: O(N)
Get an element (search/access by index): O(1)
Check for presence of an element (indexOf/lastIndexOf/contains): O(N)

LinkedList
Add/Remove an element in beginning: O(1) //linkedlist has always head node reference
Add/Remove an element in middle or end: O(1)
//assuming we've reference of the node right before the one where we've to manipulate; o/w traversing to that node will require O(N)
Get an element (search/access by index): O(N)
Check for presence of an element (indexOf/lastIndexOf/contains): O(N)

Stack
Add (Push) / Remove (Pop) / Read (Peek) an element (normally stack operation is done at Head position): O(1)

HashSet / HashMap
Add: O(1)
//Best/Average is O(1) and Worst is O(N) when hashing() function is poorly implemented or co-incidentally several keys hashing returns same bucket leading to several pairs being added to same bucket LinkedList.
Remove: O(1)
Search (contains()): O(1)

	Add	***Insert*** *(add in middle)*	***Remove***	***Get*** *(search/access)*	***Contains***	
ArrayList	O(1)	ON)	O(N)	O(1)	O(N)	
LinkedList	O(1)	O(1)	O(1)	O(N)	O(N)	

Q: How Add/Remove is O(1) in LinkedList, when it takes a O(N) time to reach that Node to be deleted?