DATA STRUCTURES

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

DATA STRUCTURES

ARRAYLIST - Resizable, includes null, access by index
ARRAY - needed size, iteration (n)
B-TREE - Self-balancing **tree** data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time.
BINARY TREE - Does not have to be ordered, but a BINARY SEARCH TREE does
BST - guarantees order (left side less, right side greater),
DICTIONARY - Key, Value. Kind of like a map. OBSOLETE

GRAPH - A set of vertex/nodes connected by edges. Formally, a **graph** is a set of vertices and a binary relation between vertices, adjacency.
HASHSET<key> - no order, no duplicates values, null is allowed, used for fast access,
HASHMAP<key, value> - permits null values and the null key. No duplicate keys, but duplicate values.
Not synchronized (not at same time)
HASHTABLE<key, value> (BUCKETS) - synchronized (occurs at same rate). Threadsafe. Does not allow null values.
looks like:
****
*
**
HEAP - Heap guarantees that elements on higher levels are greater (for max-heap) or smaller (for min-heap)
LINKEDHASHSET<key> - predictable iteration order. Maintains a doubly-linked list running through all of its entries
LINKEDHASHMAP<key, value> - like a hashmap but Preserves insertion order
LIST - ordered from insertion, null values
MAP - key, value
QUEUE - FIFO - first entered->last entered
SET - No duplicates, unordered
STACK - LIFO - last entered->first entered
TREEMAP - ordered by the key.
TREESET- ordered, no duplicates

-HashSet vs HashMap vs HashSet-

1. **Hashtable (BUCKETS) is synchronized (occur at the same time or rate.), whereas HashMap is not.** This makes HashMap better for non-threaded applications, as unsynchronized Objects typically perform better than synchronized ones.
2. **HashMap** - It allows null for both key and value. Not synchronized. Not thread safe,
3. **Hashtable** does not allow null keys or values. **HashMap** allows one null key and any number of null values.
4. **Hashset** - does not allow duplicates. You also use its contain method to check whether the object is already available in HashSet.

Interface vs Abstract Class
Methods - Interface can have only abstract methods.
Methods - Abstract class can have abstract and non-abstract methods.
Variables - **Interface** - In a Java**,** are by default final.

Variables - **Abstract class** may contain non-final variables.

USE

Interface - It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

Abstraction is a process of hiding the implementation details and showing only how or what it works (showing), how it works (hiding)

Abstract class shape

Abstract area()

Square extends shape

@override area() { }

```
public interface MusicList {
        public int getNumChannels();
        public float getSampleRate();
        public int getNumSamples();
}
public class MusicLinkedList implements MusicList {
        @Override
        public int getNumChannels() { }
}
```

String not mutable (cannot be changed). a StringBuilder is mutable (can be changed).

MEMORY

Virtual memory - memory that appears to exist as main storage although most of it is supported by data held in secondary storage - doesn't have direct access to CPU (Magnetic tape, hard drives, floppy)

Physical memory - RAM

Memory allocation in Java?

Memory on stack - function arguments, local variables, return values and return addresses are stored -An item is added when a function call is executed, and it is removed upon return from a function;

Static Memory Allocation - The allocation of memory at **compile time**

Heap memory- storage is allocated dynamically. **-Block of main memory**

Dynamic Memory Allocation. Is achieved **using certain functions like malloc(), calloc(), realloc()**

What is Memory Leak? - ANS - When programmers create a memory in heap and forget to delete it. FREE MEMORY TO AVOID MEMORY LEAK

Public, Protected, Private, no modifier

ACCESS

Modifier	Class	Package	Sub Class outside pac	All class
public	**Y**	**Y**	**Y**	**Y**
protected	**Y**	**Y**	**Y**	**N**
no modifier	**Y**	**Y**	**N**	**N**
private	**Y**	**N**	**N**	**N**

VISIBILITY

Modifier	Class	Package	Sub Class outside pac	All class
public	**Y**	**Y**	**Y**	**Y**
protected	**Y**	**Y**	**Y**	**N**
no modifier	**Y**	**Y**	**N**	**N**
private	**Y**	**N**	**N**	**N**

F**irst data column** indicates whether the class itself has access to the member defined by the access level. As you can see, a class always has access to its own members.
S**econd column** indicates whether classes in the same package as the class (regardless of their parentage) have access to the member.
T**hird column** indicates whether subclasses of the class declared outside this package have access to the member.
F**ourth column** indicates whether all classes have access to the member.

PUBLIC class, in which case that class is visible to all classes everywhere.
NO MODIFIER class - it is visible only within its own package

At the member level, you can also use the public modifier or no modifier (*package-private*) just as with top-level classes, and with the same meaning. For members, there are two additional access modifiers: private and protected.
PRIVATE modifier specifies that the **member can only be accessed in its own class.**
PROTECTED modifier specifies that the member **can only be accessed within its own package** (as with ***package-private***) and, in addition, by a subclass of its class in another package.

STATIC VARIABLE

The static keyword in java is **used for memory management** mainly

- The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees,college name of students etc.
- The static variable gets memory only once in class area at the time of class loading.

STATIC makes your program **memory efficient** (i.e it saves memory).

```
1. //Program of static variable
2. class Student8 {
3.    int rollno;
```

```
4.  String name;
5.  static String college ="ITS";
6.
7.  Student8(int r,String n){
8.  rollno = r;
9.  name = n;
10. }
11. void display (){System.out.println(rollno+" "+name+" "+college);}
12.
13. public static void main(String args[]){
14. Student8 s1 = new Student8(111,"Karan");
15. Student8 s2 = new Student8(222,"Aryan");
16.
17. s1.display();
18. s2.display();
19. }
20.}
// Output:    111 Karan ITS
              222 Aryan ITS
```

Sony
-what are static variables used for? Why use them/when to use them? If you can pass static variables to other functions and classes, how would you do that?
-When to use error handling exceptions. Why use them? What happens to the program if you do not use error handling? What happens to your program if you don't use error handling for REST or importing information/data?
-How do you override a function from the parent class in inheritance?
-What is final?
-What is a hashmap and when do you use it? What happens on the Java end of a hash map/Explain hashing?
-Name some JUnit Annotations. When do you use them and why?
-Talk about how you connected your Spring application to your SQL database. What did you use and why?

BFS vs DFS
BFS is going to use more memory depending on the branching factor... however, BFS is a complete algorithm... meaning if you are using it to search for something in the lowest depth possible, BFS will give you the optimal solution. BFS space complexity is O(b^d)... the branching factor raised to the depth (can be A LOT of memory).
DFS on the other hand, is much better about space however it may find a suboptimal solution. Meaning, if you are just searching for a path from one vertex to another, you may find the suboptimal solution (and stop there) before you find the real shortest path. DFS space complexity is O(|V|)... meaning that the most memory it can take up is the longest possible path.
They have the same time complexity.
In terms of implementation, BFS is usually implemented with Queue, while DFS uses a Stack.

Breadth First Search (BFS) and Depth First Search (DFS) are two popular algorithms to search an element in Graph or to find whether a node can be reachable from root node in Graph or not.

Difference between BFS and DFS

S. No.	Breadth First Search (BFS)	Depth First Search (DFS)
1.	BFS visit nodes level by level in Graph.	DFS visit nodes of graph depth wise. It visits nodes until reach a leaf or a node which doesn't have non-visited nodes.
2.	A node is fully explored before any other can begin.	Exploration of a node is suspended as soon as another unexplored is found.
3.	Uses Queue data structure to store Un-explored nodes.	Uses Stack data structure to store Un-explored nodes.
4.	BFS is slower and require more memory.	DFS is faster and require less memory.
5.	Some Applications: ● Finding all connected components in a graph. ● Finding the shortest path between two nodes. ● Finding all nodes within one connected component. ● Testing a graph for bipartiteness.	Some Applications: ● Topological Sorting. ● Finding connected components. ● Solving puzzles such as maze. ● Finding strongly connected components. ● Finding articulation points (cut vertices) of the graph.

RANDOM SHIT PEOPLE TOLD ME TO USE TO STUDY FOR INTERVIEWS

Know programming language (data structures, algorithms): networks, web, OS, mobile, etc

Websites:

-Princeton algorithms course,
- stanford algorithms: design and analysis,
-coursera/course/algo
https://www.interviewbit.com/
Glassdoor - https://www.glassdoor.com/Interview/internships-interview-questions-SRCH_KO0,11.htm
Careercup
Leetcode
hackerrank
http://practice.geeksforgeeks.org/company-tags

Books

Intro to Algo, cormen
Programming Interview Exposed, Mongan
How to think about Algorithms, edmonds
Cracking the Code

How To Solve Problems
Constraints - what should you return, problems
Ideas: of how to solve problem
Test Cases: output
Code

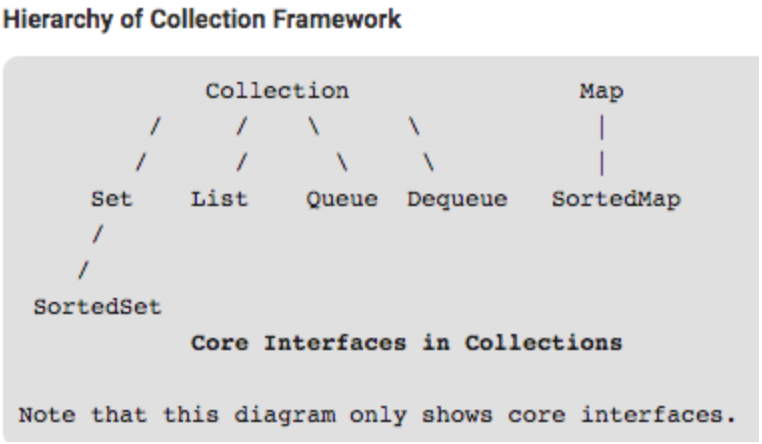
Problem Example - Sort
00110000
00000011

Problems
Event planner - write out all events that do not conflict with each other
-hashing funct - two files are exactly the same except one byte - how do you find byte (binary search)
Start point of matrix, end point, get all points, Loop through points

I like cake -> I ekil ekac

Prepare
Make Connections
Don't fall into: Imposter syndrome

COLLECTIONS



Collection : Root interface with basic methods like:
add(), remove(), contains(), isEmpty(), addAll(), ... etc.

Set : **Doesn't allow duplicates**. Example implementations of Set interface are HashSet (Hashing based) and TreeSet (balanced BST based). Note that TreeSet implements SortedSet.

List : Can contain duplicates and **elements are ordered**. Example implementations are LinkedList (linked list based) and ArrayList (dynamic array based)

Queue : Typically order elements in FIFO order except exceptions like PriorityQueue.

Deque : Elements can be inserted and removed at both ends. Allows both LIFO and FIFO.

Map : Contains **Key value pairs**. Doesn't allow duplicates. Example implementation are HashMap and TreeMap. TreeMap implements SortedMap.

The difference between Set and Map interface is that in
Set we have only keys, whereas in
Map, we have key, value pairs.

POLYMORPHISM

```
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}
```

Now, the Deer class is considered to be polymorphic since this has multiple inheritance. Following are true for the above examples −

- A Deer IS-A Animal
- A Deer IS-A Vegetarian
- A Deer IS-A Deer
- A Deer IS-A Object

There are four main OOP concepts in Java. These are:

- **Abstraction.** Abstraction means using simple things to represent complexity. We all know how to turn the TV on, but we don't need to know how it works in order to enjoy it. In Java, abstraction means simple things like **objects**, **classes**, and **variables** represent more complex underlying code and data. This is important because it lets avoid repeating the same work multiple times.
- **Encapsulation.** This is the practice of keeping fields within a class private, then providing access to them via public methods. It's a protective barrier that keeps the data and code safe within the class itself. This way, we can re-use objects like code components or variables without allowing open access to the data system-wide.
- **Inheritance.** This is a special feature of Object Oriented Programming in Java. It lets programmers create new classes that share some of the attributes of existing classes. This lets us build on previous work without reinventing the wheel.
- **Polymorphism.** This Java OOP concept lets programmers use the same word to mean different things in different contexts. One form of polymorphism in Java is **method overloading**. That's when different meanings are implied by the code itself. The other form is **method overriding**. That's when the different meanings are implied by the values of the supplied variables. See more on this below.

Difference between Java and Python - Java is a statically typed (This implies that static typing has to do with the explicit declaration (or initialization) of variables before they're employed - int num = 5) and Python is a dynamically typed (This implies that dynamic typed

languages do not require the explicit declaration of the variables before they’re used.). Num = 5