DOCUMENTATION

SUBMITTED BY: METATITANS

Aadil A A - 245268
Meera Javad- 245217
Swetha S -245154
Christo Shaji -245052

Space Complexity

Space complexity refers to the amount of memory or space required by an algorithm to solve a problem. It is a measure of the amount of memory used by an algorithm to store data, temporary variables, and other information during the execution of the algorithm.

Space complexity is typically expressed in terms of the amount of memory required by an algorithm as a function of the input size. The input size is usually denoted by the variable n, and space complexity is denoted by the notation O(f(n)), where f(n) is some function of n that describes the amount of memory required by the algorithm.

Types of space complexities:

1. Constant Space Complexity: An algorithm has a constant space complexity if the amount of memory used by the algorithm is independent of the input size. For example, an algorithm that sorts an array in place has a constant space complexity of O(1).

2. Linear Space Complexity: An algorithm has a linear space complexity if the amount of memory used by the algorithm is proportional to the input size. For example, an algorithm that creates an array of n elements has a linear space complexity of O(n).

3. Quadratic Space Complexity: An algorithm has a quadratic space complexity if the amount of memory used by the algorithm is proportional to the square of the input size. For example, an algorithm that creates a two-dimensional array of size n x n has a quadratic space complexity of O(n^2).

4. Exponential Space Complexity: An algorithm has an exponential space complexity if the amount of memory used by the algorithm grows exponentially with the input size. For example, an algorithm that generates all subsets of a set of n elements has an exponential space complexity of O(2^n).

It is important to consider space complexity when designing and analyzing algorithms, especially for problems that involve large data sets. Algorithms with low space complexity are usually preferred, as they require less memory and can be executed more efficiently.

How to calculate space complexity.

To calculate the space complexity of an algorithm, you need to determine how much memory the algorithm requires as a function of the input size. Here are the steps to follow:

Identify the variables and data structures used by the algorithm: Look for any variables or data structures used by the algorithm, such as arrays, lists, variables, and pointers.

Determine the space requirements of each variable and data structure: Determine how much memory is required to store each variable and data structure. This will depend on the data type and size of the variable or data structure.

Count the number of times each variable or data structure is used: Count how many times each variable or data structure is used in the algorithm. This will help you determine the overall space requirements of the algorithm.

Sum up the space requirements of all variables and data structures: Add up the space requirements of all variables and data structures used by the algorithm. This will give you an estimate of the total space required by the algorithm.

Express the space complexity in terms of the input size: Finally, express the space complexity as a function of the input size. This is usually done using big O notation, which gives an upper bound on the space requirements of the algorithm as the input size grows.

For example, consider the following algorithm that creates an array of n integers and then iterates over the array:

```
int[] createArray (int n) {
int[] arr = new int[n];
for (int i = 0; i < n; i++) {
arr[i] = i;
}
return arr;
}
```

To calculate the space complexity of this algorithm, we can follow these steps:

Variables and data structures: The algorithm uses an integer variable n and an integer array arr.

Space requirements: The integer variable n requires a constant amount of memory (e.g., 4

bytes), and the integer array arr requires n * sizeof(int) bytes of memory.

Variable usage: The variable n is used once to initialize the array, and the array arr is used n times in the loop.

Total space requirements: The total space required by the algorithm is 4 + n * sizeof(int) bytes.

Space complexity: The space complexity of the algorithm can be expressed as O(n) since the space required grows linearly with the input size.

Need for analyzing space complexity

Space complexity is an important metric for evaluating the efficiency of an algorithm, as it measures the amount of memory required by the algorithm to solve a problem. Here are some reasons why space complexity is important:

Memory constraints: Many computing systems, such as embedded systems, mobile devices, and cloud computing platforms, have limited memory resources. Algorithms with high space complexity may not be suitable for these systems and may cause the system to run out of memory or slow down.

Time-space trade-off: There is often a trade-off between time complexity and space complexity. Algorithms with high space complexity may be faster than algorithms with low space complexity, and vice versa. By analyzing the space complexity of an algorithm, we can choose the best algorithm for a given problem based on the available resources.

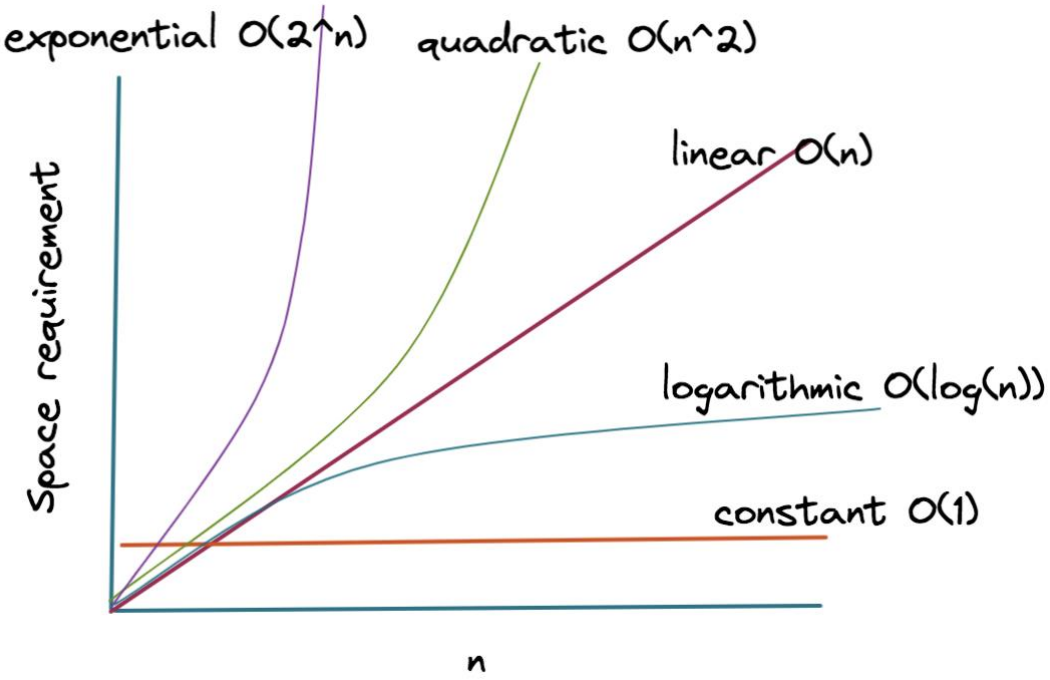
Algorithm optimization: Analyzing the space complexity of an algorithm can help identify opportunities for optimization. For example, if an algorithm requires a large amount of memory, we may be able to optimize the algorithm by reducing the number of data structures or variables used.

Algorithm design: Space complexity can also influence the design of algorithms. For example, if we know that memory is a scarce resource, we may design algorithms that use constant space or optimize the use of memory.

Overall, understanding space complexity is important for designing efficient algorithms that are well-suited to the available computing resources and constraints.

Space complexity of various algorithms

Algorithm	Worst case space complexity
Bubble Sort	O(1)
Selection Sort	O(1)
Insertion Sort	O(1)
Merge Sort	O(n)
Quick Sort	O(n)
Heap Sort	O(1)
Radix Sort	O(n + K) Where, k – range of array elements



STRINGS IN JAVA

Strings in Java are a sequence of characters stored as an object in memory. They are used to represent textual data and are one of the most used data types in Java.

Example: String str=”computer”

The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence interface.*

Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object

There are two ways to create String object:

1. By string literal
2. By new keyword

String literal:
String s = “Laptop”;

Using new keyword:
String s = new String (“Laptop”);

IMMUTABLE STRING IN JAVA

In Java, String object are immutable. Immutable simply means unmodifiable or unchangeable. Once String object is created its data or state can't be changed but a new String object is created.

For example: String s=”hello”;
s.concat(“world”);// concat method add string at the end.
System.out.println(s)//output is cat because string is an immutable object.

Explanation:
When JVM will execute statement String s = “hello”;, it will create a string object in the string constant pool and store “hello” in it.

When the next statement s.concat(“world”); will be executed by JVM, it will create two new objects because we are trying to modify the original content.

1. First, for every string literal “world”, JVM will create one copy of string object in the string constant pool.

2. The second object will be created in the heap with modified content “hello world”. Since string concatenation is executed at the runtime. Therefore, if a new object is required to create, this new object is always created in the heap area only, not in string constant pool.

Since this new object is not assigning with any reference variable, therefore, it is called unreferenced object and the garbage collector will automatically remove it from the memory.

Thus, the value of string s is not modified and still, ‘s’ is pointing to “hello” only. Therefore, the result is “hello”. This is the reason why string objects are called immutable in Java.

Following are some features of String which makes String objects immutable.

1. ClassLoader:

A ClassLoader in Java uses a String object as an argument. Consider, if the String object is modifiable, the value might be changed and the class that is supposed to be loaded might be different.

To avoid this kind of misinterpretation, String is immutable.

2. Thread Safe:

As the String object is immutable, we don't have to take care of the synchronization that is required while sharing an object across multiple threads.

3. Security:

As we have seen in class loading, immutable String objects avoid further errors by loading the correct class. This leads to making the application program more secure. Consider an example of banking software. The username and password cannot be modified by any intruder because String objects are immutable. This can make the application program more secure.

4. Heap Space:

The immutability of String helps to minimize the usage in the heap memory. When we try to declare a new String object, the JVM checks whether the value already exists in the String pool or not. If it exists, the same value is assigned to the new object. This feature allows Java to use the heap space efficiently.

JAVA STRING CLASS METHODS

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

No.	Method	Description
1	char charAt(int index)	It returns char value for the particular index
2	int length()	It returns string length
3	static String format(String format, Object... args)	It returns a formatted string.
4	static String format(Locale l, String format, Object... args)	It returns formatted string with given locale.
5	String substring(int beginIndex)	It returns substring for given begin index.
6	String substring(int beginIndex, int endIndex)	It returns substring for given begin index and end index.
7	boolean contains(CharSequence s)	It returns true or false after matching the sequence of char value.
8	static String join(CharSequence delimiter, CharSequence... elements)	It returns a joined string.
9	static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	It returns a joined string.
10	boolean equals(Object another)	It checks the equality of string with the given object.

11	boolean isEmpty()	It checks if string is empty.
12	String concat(String str)	It concatenates the specified string.
13	String replace(char old, char new)	It replaces all occurrences of the specified char value.
14	String replace(CharSequence old, CharSequence new)	It replaces all occurrences of the specified CharSequence.
15	static String equalsIgnoreCase(String another)	It compares another string. It doesn't check case.
16	String[] split(String regex)	It returns a split string matching regex.
17	String[] split(String regex, int limit)	It returns a split string matching regex and limit.
18	String intern()	It returns an interned string.
19	int indexOf(int ch)	It returns the specified char value index.
20	int indexOf(int ch, int fromIndex)	It returns the specified char value index starting with given index.
21	int indexOf(String substring)	It returns the specified substring index.
22	int indexOf(String substring, int fromIndex)	It returns the specified substring index starting with given index.
23	String toLowerCase()	It returns a string in lowercase.

24	String toLowerCase(Locale l)	It returns a string in lowercase using specified locale.
25	String toUpperCase()	It returns a string in uppercase.
26	String toUpperCase(Locale l)	It returns a string in uppercase using specified locale.
27	String trim()	It removes beginning and ending spaces of this string.
28	static String valueOf(int value)	It converts given type into string. It is an overloaded method.

JAVA STRINGBUFFER CLASS

Java StringBuffer class is used to create mutable (modifiable) String objects. The StringBuffer class in Java is the same as String class except it is mutable i.e., it can be changed.

Important Constructors of StringBuffer Class

Constructor	Description
StringBuffer()	It creates an empty String buffer with the initial capacity of 16.
StringBuffer(String str)	It creates a String buffer with the specified string..
StringBuffer(int capacity)	It creates an empty String buffer with the specified capacity as length.

StringBufferExample.java

```
class StringBufferExample{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
sb.append("Java");//now original string is changed
System.out.println(sb);//prints Hello Java
}
}
```

Output: Hello Java

JAVA STRINGBUILDER CLASS

Java StringBuilder class is used to create mutable (modifiable) String. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5

Important Constructors of StringBuilder class

Constructor	Description
StringBuilder()	It creates an empty String Builder with the initial capacity of 16.
StringBuilder(String str)	It creates a String Builder with the specified string.
StringBuilder(int length)	It creates an empty String Builder with the specified capacity as length.

StringBuilderExample.java

```
class StringBuilderExample{
public static void main(String args[]){
StringBuilder sb=new StringBuilder("Hello ");
sb.append("Java");//now original string is changed


System.out.println(sb);//prints Hello Java
}
}
```

STRING TOKENIZER IN JAVA

The **java.util.StringTokenizer** class allows you to break a String into tokens. It is simple way to break a String. It is a legacy class of Java. It doesn't provide the facility to differentiate numbers, quoted strings, identifiers etc. like Stream Tokenizer class.

Constructor	Description
StringTokenizer(String str)	It creates StringTokenizer with specified string.
StringTokenizer(String str, String delim)	It creates StringTokenizer with specified string and delimiter.
StringTokenizer(String str, String delim, boolean returnValue)	It creates StringTokenizer with specified string, delimiter and returnValue. If return value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens.

It doesn't provide the facility to differentiate numbers, quoted strings, identifiers etc. like StreamTokenizer class.

Example of Stringtokenizer in Java

Simple.java

```
import java.util.StringTokenizer;
public class Simple{
 public static void main(String args[]){
   StringTokenizer st = new StringTokenizer("my name is khan"," ");
    while (st.hasMoreTokens()) {
        System.out.println(st.nextToken());
    }
   }
}
```

Output:

```
my
name
is
khan
```

Time complexity

Time complexity is a measure of the amount of time required to execute an algorithm as a function of the size of its input. In Java, we can analyze the time complexity of an algorithm by counting the number of basic operations that it performs.

Here are some common time complexities and their corresponding examples of algorithms in Java:

O(1): constant time complexity, which means the algorithm takes the same amount of time to execute regardless of the size of the input. An example of an O(1) algorithm in Java is accessing an element in an array by index.

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

int element = array[3]; // O(1)
```

O(n): linear time complexity, which means the algorithm's time to execute grows linearly with the size of the input. An example of an O(n) algorithm in Java is iterating through an array or a list.

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

for (int i = 0; i < array.length; i++) { // O(n)
    System.out.println(array[i]);
}
```

O(n^2): quadratic time complexity, which means the algorithm's time to execute grows quadratically with the size of the input. An example of an O(n^2) algorithm in Java is a nested loop that iterates through an array or a list.

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

for (int i = 0; i < array.length; i++) { // O(n)
    for (int j = 0; j < array.length; j++) { // O(n)
        System.out.println(array[i] + ", " + array[j]);
    }
}
```

O(log n): logarithmic time complexity, which means the algorithm's time to execute grows logarithmically with the size of the input. An example of an O(log n) algorithm in Java is a binary search on a sorted array.

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

```
int target = 3;
int low = 0;
int high = array.length - 1;
while (low <= high) { // O(log n)
    int middle = (low + high) / 2;
    if (array[middle] == target) {
        System.out.println("Found target at index " + middle);
        break;
    } else if (array[middle] < target) {
        low = middle + 1;
    } else {
        high = middle - 1;
    }
}
```

O(n log n): logarithmic time complexity that is multiplied by a linear factor. An example of an O(n log n) algorithm in Java is merge sort, which sorts an array by dividing it into halves, recursively sorting each half, and then merging the two sorted halves.

```
public static void mergeSort(int[] arr, int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}


public static void merge(int[] arr, int left, int mid, int right) {
    int[] temp = new int[right - left + 1];
    int i = left, j = mid + 1, k = 0;
    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j]) {
```

```
            temp[k++] = arr[i++];
        } else {
            temp[k++] = arr[j++];
        }
    }
    while (i <= mid) {
        temp[k++] = arr[i++];
    }
    while (j <= right) {
        temp[k++] = arr[j++];
    }
    for (int x = 0; x < temp.length; x++) {
        arr[left + x] = temp[x];
    }
}


public static void main(String[] args) {
    int[] arr = {5, 2, 8, 4, 1, 9};
    mergeSort(arr, 0, arr.length - 1);
    System.out.println(Arrays.toString(arr));
}
```

Big O notation

The big O notation represents the upper bound of an algorithm's time complexity. It tells us how the algorithm's time to execute grows as the size of the input grows. For example, an algorithm with O(n) time complexity means that its time to execute is proportional to the size of the input, but it could be less than that in some cases.

The worst-case time complexity of an algorithm is the maximum amount of time it takes to execute for any input of size n. This is the most common way to analyze the time complexity of an algorithm, because it gives us an upper bound on the execution time that we can use to compare algorithms.

The best-case time complexity of an algorithm is the minimum amount of time it takes to execute for any input of size n. This is usually not very useful, because we usually want to know how an algorithm performs on average or in the worst case.

The average-case time complexity of an algorithm is the expected amount of time it takes to execute for a random input of size n. This is harder to analyze than the worst-case time complexity, because it requires knowledge of the input distribution and the algorithm's behavior on that distribution.

In Java, the time complexity of an algorithm depends on the operations that it performs on the data structures that it uses. For example, inserting an element into a sorted list takes O(n) time in the worst case, because we may need to shift all the elements to the right of the insertion point.

It is important to choose the right data structure and algorithm for the problem that we are solving in Java, because different data structures and algorithms have different time complexities for different operations.

•The time complexity of an algorithm can be improved by optimizing its code or using a faster data structure or algorithm. However, there is usually a trade-off between time complexity and space complexity, because faster algorithms or data structures usually require more memory.

•The time complexity of an algorithm can be measured empirically by running it on inputs of different sizes and measuring the time it takes to execute. However, this approach can be affected by external factors such as the hardware and software environment, so it is usually not very accurate or reliable. Theoretical analysis is usually more precise and reliable, but it requires knowledge of mathematics and computer science theory.

Calculate the time complexity

Identify the algorithm: First, you need to identify the algorithm used in the Java program. An algorithm is a set of instructions that solves a specific problem.

Count the operations: Next, count the number of basic operations that the algorithm performs. Basic operations are typically simple arithmetic operations, assignments, and comparisons.

Determine the input size: Determine the size of the input that the algorithm operates on. This could be the length of an array, the number of elements in a list, or any other relevant measure.

Express the time complexity: Finally, express the time complexity of the algorithm using Big O notation. Big O notation provides an upper bound on the growth rate of a function, and is used to describe the worst-case scenario for the algorithm's performance.

Here are some common time complexity expressions and their meanings:

O(1): Constant time. The algorithm takes the same amount of time regardless of the input size.

O(log n): Logarithmic time. The algorithm's running time increases logarithmically with the input size.

O(n): Linear time. The algorithm's running time increases linearly with the input size.

O(n^2): Quadratic time. The algorithm's running time increases quadratically with the input size.

O(2^n): Exponential time. The algorithm's running time doubles with each additional input.

Once you have determined the time complexity of the algorithm, you can use this information to make decisions about optimizing the program or selecting alternative algorithms.

Time complexity of binary search:

Binary search is an algorithm for finding the position of a target value within a sorted array. It works by repeatedly dividing the search interval in half until the target value is found or determined to be not in the array.

To calculate the time complexity of binary search, we need to count the number of basic operations that the algorithm performs, and express this count as a function of the input size.

The basic operations in binary search include:

Comparing the target value to the middle element of the array

Dividing the search interval in half

Checking if the target value has been found

Let n be the size of the array.

In the best case, the target value is found at the middle position of the array on the first comparison. In this case, the algorithm takes O(1) time, since it only performs one comparison.

In the worst case, the target value is not in the array, and the algorithm must continue dividing the search interval in half until it reaches a single element. This takes log2(n) iterations, since each division reduces the search interval by half. At each iteration, one comparison is

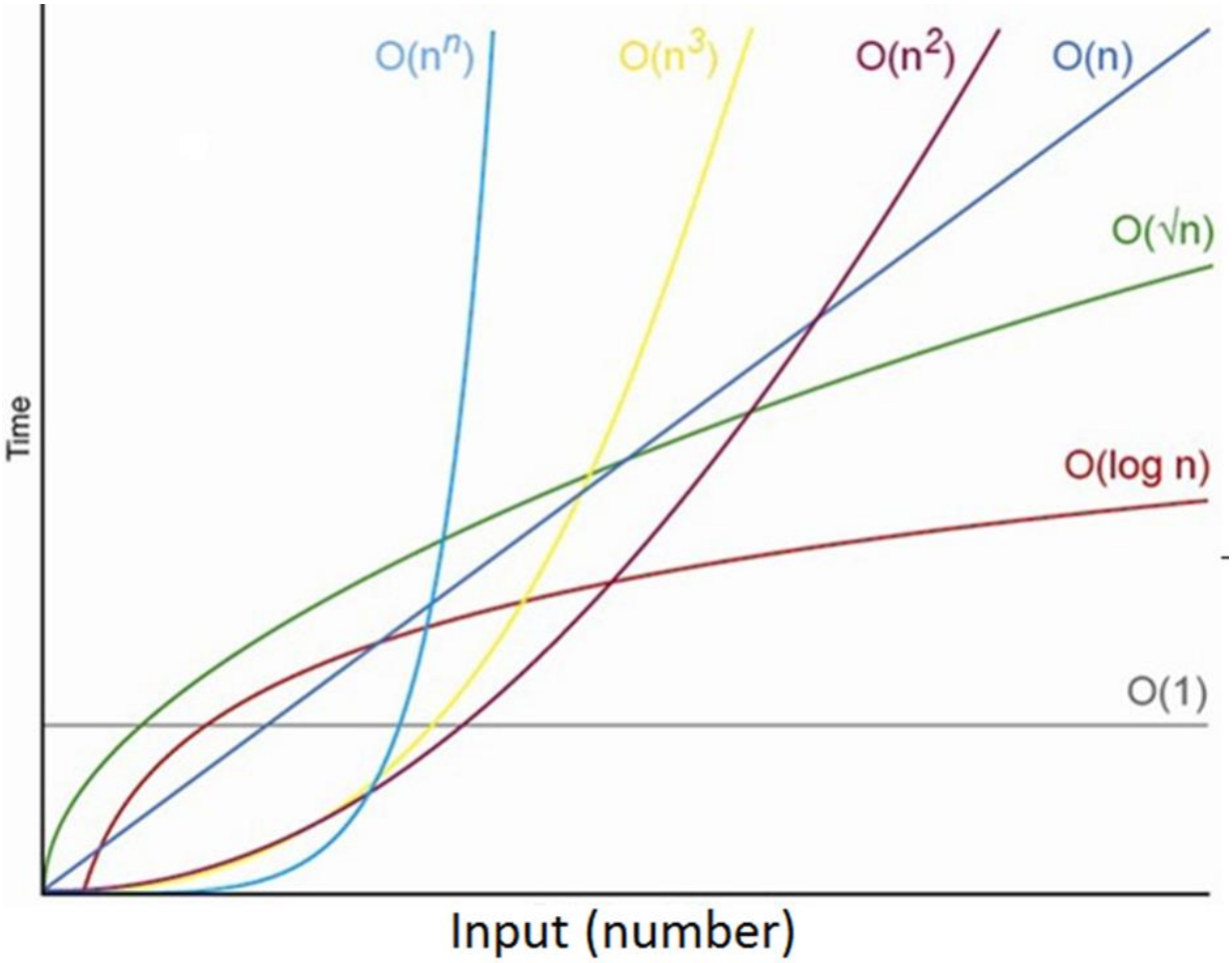
performed. Therefore, the total number of basic operations is log2(n) + 1 (one additional comparison to check if the target value is not in the array).

Thus, the time complexity of binary search in the worst case is O(log n).

In the average case, binary search also takes O(log n) time, since the target value is expected to be found in the middle of the array half of the time.

Time complexity of different algorithms

Data Structure	Worst Case Time Complexity			
	Access	Search	Insertions	Delete
Array	O(1)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)
Queue	O(n)	O(n)	O(1)	O(1)
Singly Linked List	O(n)	O(n)	Begin: O(1), End: O(n)	Begin: O(1), End: O(n)
Doubly Linked List	O(n)	O(n)	Begin: O(1), End: O(n)	Begin: O(1), End: O(n)
Binary Search Tree	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))



Illustration

Linear Time Complexity (O(n)):

The following code iterates through an array of size n and prints out each element once. The time complexity of this code is O(n) because it takes linear time to complete and the number of operations increases linearly with the input size (n).

```
int[] arr = {1, 2, 3, 4, 5};
for (int i = 0; i < arr.length; i++) {
    System.out.println(arr[i]);
}
```

Quadratic Time Complexity (O(n^2)):

The following code nests two loops that each iterate through an array of size n. This code has a time complexity of O(n^2) because it takes quadratic time to complete and the number of operations increases exponentially with the input size (n).

```
int[] arr = {1, 2, 3, 4, 5};
for (int i = 0; i < arr.length; i++) {
    for (int j = 0; j < arr.length; j++) {
        System.out.println(arr[i] + " " + arr[j]);
    }
}
```

Logarithmic Time Complexity (O(log n)):

The following code uses binary search to find the index of a specific value in a sorted array. The time complexity of this code is O(log n) because the search space is halved with each iteration, resulting in a smaller number of operations with larger input sizes.

```
int[] arr = {1, 2, 3, 4, 5};
int value = 3;
int left = 0, right = arr.length - 1;
while (left <= right) {
    int mid = (left + right) / 2;
    if (arr[mid] == value) {
        System.out.println("Found at index " + mid);
        break;
    } else if (arr[mid] < value) {
        left = mid + 1;
    } else {
        right = mid - 1;
    }
}
```

IMMUTABLE OBJECTS

Immutable objects are objects whose state cannot be modified after creation. In other words, once an immutable object is created, its value remains the same throughout its lifetime. Immutable objects are useful in programming because they are thread-safe, can be used in hash tables, and simplify code.

Examples of immutable objects:

Some examples of immutable objects in Java are:

- String: Once a string object is created, its value cannot be changed.
- Integer: Once an integer object is created, its value cannot be changed.
- LocalDate: A LocalDate object represents a date (year, month, day) and cannot be modified after creation.
- BigDecimal: A BigDecimal object represents a decimal value and cannot be modified after creation.

Creating immutable objects:

To create an immutable object, the object's state must be initialized in the constructor and all fields must be marked as final. Any method that change the state of the object should return a new object instead of modifying the existing one.

Here are some techniques to create immutable objects:

- Make all fields final: In Java, you can make all the fields of a class final to make it immutable. Final fields cannot be modified once they are initialized, so making all fields final ensures that the object cannot be modified.
- Use a constructor to initialize all fields: In Python, you can create a class with a constructor that initializes all the fields of the object. Once the object is created, the fields cannot be modified.

- Return a copy of the object: In some programming languages, you can create a copy of an object and return it, instead of the original object. This copy can be made immutable, so any attempts to modify it will create a new copy of the object.
- Use data structures that are inherently immutable: Some data structures, such as trees and linked lists, are inherently immutable. Once they are created, they cannot be modified.

- Use a functional programming style: Functional programming languages like Haskell and Clojure encourage immutability by default, making it easy to create immutable objects.

It's important to note that creating immutable objects can sometimes be less efficient than creating mutable objects, because every change to the object requires the creation of a new object. However, the benefits of immutability - such as thread safety, ease of reasoning about code, and fewer bugs - often make it worthwhile.

Here's an example code for implementing an immutable Person class in Java:

```
public final class Person {
   private final String name;
   private final int age;

   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }

   public String getName() {
      return name;
   }
```

```
 public int getAge() {
      return age;
   }
}
```

In this example, the Person class has two private final fields, name and age. The constructor initializes these fields, and there are no setter methods provided. The getter methods getName() and getAge() allow clients to access the values of the fields, but they cannot modify them.

Here's an example usage of the Person class:

```
Person person = new Person("Alice", 30);
System.out.println(person.getName()); // Output: Alice
System.out.println(person.getAge()); // Output: 30
```

In this example, a new Person object is created with the name "Alice" and age 30. The getName() and getAge() methods are called to retrieve the values of the fields, but there is no way to modify them once the object has been created.

Benefits of immutable objects

- Thread safety: Immutable objects are inherently thread-safe, because once an immutable object is created, it cannot be changed. This means that multiple threads can access the same object without the risk of race conditions or other concurrency issues.
- Security: Immutable objects cannot be modified after they are created, which means that any code that relies on them can be sure that their values will not be tampered with. This can be important in security-sensitive applications.
- Ease of use: Immutable objects are often easier to use than mutable objects, because you don't have to worry about accidentally modifying their state. This can lead to simpler and more maintainable code.
- Better performance: Immutable objects can be more efficient than mutable objects in certain situations, because they don't require locking or synchronization to ensure thread safety. They also make it easier to implement caching and other optimizations.

- Better debugging: Because immutable objects cannot be modified after they are created, it is easier to trace bugs and other issues back to their source. This can make it easier to diagnose and fix problems in your code.