BIG DATA UNIT-2 BCA DS

What is Map Data Structure?

Map data structure (also known as a **dictionary** , **associative array** , or **hash map**) is defined as a data structure that stores a collection of key-value pairs, where each key is associated with a single value.

Maps provide an efficient way to store and retrieve data based on a unique identifier (the key).

Need for Map Data Structure

Map data structures are important because they allow for efficient storage and retrieval of key-value pairs. Maps provide the following benefits:

- **Fast Lookup:** Unordered maps allow for constant-time (O(1)) average-case lookup of elements based on their unique keys.
- **Efficient Insertion and Deletion:** Maps support fast insertion and deletion of key-value pairs, typically with logarithmic (O(log n)) or constant-time (O(1)) average-case complexity.
- **Unique Keys:** Maps ensure that each key is unique, allowing for efficient association of data with specific identifiers.
- **Flexible Data Storage:** Maps can store a wide variety of data types as both keys and values, providing a flexible and versatile data storage solution.
- **Intuitive Representation:** The key-value pair structure of maps offers an intuitive way to model and represent real-world data relationships.

Properties of Map Data Structure:

A map data structure possesses several key properties that make it a valuable tool for various applications:

- **Key-Value Association:** Maps allow you to associate arbitrary values with unique keys. This enables efficient data retrieval and manipulation based on keys.
- **Unordered (except for specific implementations):** In most map implementations, elements are not stored in any specific order. This means that iteration over a map will yield elements in an arbitrary order. However, some map implementations, such as **TreeMap** in Java, maintain order based on keys.
- **Dynamic Size:** Maps can grow and shrink dynamically as you add or remove elements. This flexibility allows them to adapt to changing data requirements without the need for manual resizing.
- **Efficient Lookup:** Maps provide efficient lookup operations based on keys. You can quickly find the value associated with a specific key using methods like **get()** or **[]** with an average time complexity of **O(1)** for hash-based implementations and **O(log n)** for tree-based implementations.

- **Duplicate Key Handling:** Most map implementations do not allow duplicate keys. Attempting to insert a key that already exists will typically overwrite the existing value associated with that key. However, some map implementations, like **multimap** in C++, allow storing multiple values for the same key.
- **Space Complexity:** The space complexity of a map depends on its implementation. Hash-based maps typically have a space complexity of **O(n)** , where **n** is the number of elements, while tree-based maps have a space complexity of **O(n log n)** .
- **Time Complexity:** The time complexity of operations like insertion, deletion, and lookup varies depending on the implementation. Hash-based maps typically have an average time complexity of **O(1)** for these operations, while tree-based maps have an average time complexity of **O(log n)** . However, the worst-case time complexity for tree-based maps is still **O(log n)** , making them more predictable and reliable for performance-critical applications.

Set	Map
Unique Values	keys are unique, but the values can be duplicated
Unordered Collection	Unordered Collection
Dynamic	Dynamic
Iterate over the set to retrieve the value.	Elements can be retrieved using their key
Set operations like union, intersection, and difference.	Maps are used for operations like adding, removing, and accessing key-value pairs.
Implemented using linked lists or trees	Implemented using linked lists or trees

What is Set Data Structure?

In computer science, a **set data structure** is defined as a data structure that stores a collection of distinct elements.
It is a fundamental Data Structure that is used to store and manipulate a group of objects, where each object is **unique** . The Signature property of the set is that it doesn't allow duplicate elements.

A **set is a mathematical model** for a collection of different things, a set contains elements or members, which can be mathematical objects of any kind numbers, symbols, points in space, lines, other geometrical shapes, variables, or even other sets.

A set can be implemented in various ways but the most common ways are:

1. **Hash-Based Set:** the set is represented as a hash table where each element in the set is stored in a bucket based on its hash code.
2. **Tree-based set:** In this implementation, the set is represented as a binary search tree where each node in the tree represents an element in the set.

Types of Set Data Structure:

The set data structure can be classified into the following two categories:

1. Unordered Set

An **unordered set** is an unordered associative container implemented using a hash table where keys are hashed into indices of a hash table so that the insertion is always randomized. All operations on the unordered set take constant time **O(1)** on an average which can go up to linear time **O(n)** in the worst case which depends on the internally used hash function, but practically they perform very well and generally provide a constant time lookup operation.

2. Ordered Set

An Ordered set is the common set data structure we are familiar with. It is generally implemented using balanced BSTs and it supports **O(log n)** lookups, insertions and deletion operations.

what is generics

Generics is a feature in Java that allows you to define classes, interfaces, and methods with type parameters. This means that you can write a generic class or method that can work with any data type (Integer, String, etc.), and the type is specified when you create an object or call a method.

1. Generics (Overview):

Generics help in:

- **Type Safety**: Generics enforce compile-time type checking. Without generics, we might get errors at runtime when using objects of incompatible types.
- **Code Reusability**: You can create a single method or class that works with different data types, instead of writing the same code for each type.

Generic Classes:

A **generic class** is a class that can work with any type. You declare a generic class by specifying a type parameter in angle brackets <T>.

Example:

```java
Copy code
// Generic class
class Box<T> {
    private T item;

    public void set(T item) {
        this.item = item;
    }

    public T get() {
        return item;
    }
}
```

```
// Usage
public class Main {
    public static void main(String[] args) {
        Box<Integer> intBox = new Box<>(); // Integer type
        intBox.set(10);
        System.out.println("Integer Value: " + intBox.get());

        Box<String> strBox = new Box<>(); // String type
        strBox.set("Hello");
        System.out.println("String Value: " + strBox.get());
    }
}
```

- In this example, Box<T> is a generic class, where T can be replaced with any type like Integer or String.

3. Type Parameters:

In Java generics, **type parameters** are the symbols used to define the generic types in a class or method. Common type parameters are:

- T – Type
- E – Element (used mostly in collections like List, Set, etc.)
- K – Key (for maps)
- V – Value (for maps)

You declare a type parameter in angle brackets <T>. When you instantiate the class, you specify the actual type to replace T.

Example with Multiple Type Parameters:

java

Copy code

```
class Pair<K, V> {
```

```
    private K key;
    private V value;

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

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }
}
```

Here, the class Pair<K, V> has two type parameters: K for the key and V for the value. You can create objects of this class like this:

java

Copy code

```
Pair<String, Integer> pair = new Pair<>("Age", 25);
```

Benefits of Generics:

- **Type Safety**: Avoids ClassCastException by catching invalid types at compile time.
- **Code Reusability**: Write a single generic class or method and reuse it with any type.

Wrapper Class

A Wrapper class in Java is a class whose object wraps or contains primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

Primitive Data Types and their Corresponding Wrapper Class

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean