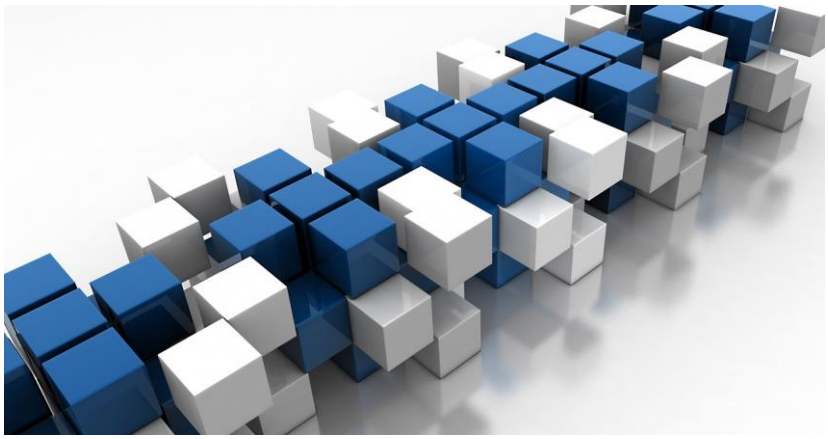

MAPS, HASH TABLES AND SETS

DATA STRUCTURES IN JAVA

Sercan Külcü | Data Structures In Java | 10.05.2023

Contents

Maps

A map is a data structure that can store a collection of key-value pairs. Maps are often used to store data that is not ordered, such as a list of tasks to be completed or a list of files.

In Java, maps are implemented using the Map interface. The Map interface defines a number of methods for adding, removing, and accessing elements in a map.

Some of the most common methods in the Map interface include:

- put(): Adds a key-value pair to the map.
- get(): Gets the value associated with a specific key in the map.
- remove(): Removes a key-value pair from the map.
- size(): Gets the number of key-value pairs in the map.

Maps are a powerful tool that can be used to store and manipulate data. By understanding how maps work, you can write more efficient and effective code.

Here are some examples of how maps can be used:

- A map can be used to store a list of tasks to be completed.
- A map can be used to store a list of files.
- A map can be used to store a list of websites to visit.
- A map can be used to store a list of friends on social media.

Maps are a versatile data structure that can be used to store a variety of data. By understanding how maps work, you can write more efficient and effective code.

Here are some additional details about maps:

- Maps are implemented using hash tables, which are data structures that store data based on a key.

- Maps are not synchronized, which means that multiple threads can access the map at the same time. This can lead to race conditions, which are errors that can occur when multiple threads are trying to access the same data at the same time.
- Maps are a good choice for storing data that is not ordered, such as a list of tasks to be completed or a list of files.
- Maps are a good choice for storing data that needs to be frequently accessed by key.

Here are some tips for using maps:

- Use maps when you need to store a collection of elements that is not ordered.
- Use maps when you need to be able to add and remove elements from the collection frequently.
- Use maps when you need to be able to access elements in the collection by key.

Maps are a powerful tool that can be used to store and manipulate data. By understanding how maps work and using them correctly, you can write more efficient and effective code.

HashMap

Maps are used to store and retrieve data in an efficient way. They are useful in situations where you need to quickly find a value based on its associated key. For example, consider a scenario where you need to store the grades of a class of students. You could use a map with the student's name as the key and their grade as the value. This way, you can easily retrieve the grade of a particular student by using their name as the key.

The basic operations that can be performed on a map include inserting a key-value pair, removing a key-value pair, and retrieving the value associated with a key. Let's take a look at how these operations are performed in Java.

To create a map in Java, you can use the HashMap class. Here's an example of how to create a map:

Map<String, Integer> grades = new HashMap<>();

In this example, we are creating a map that stores the grades of students as integers, with the keys being strings that represent their names.

To insert a key-value pair into the map, you can use the put method:

grades.put("John Doe", 90);

This code inserts the key-value pair "John Doe: 90" into the grades map.

To retrieve the value associated with a key, you can use the get method:

int johnsGrade = grades.get("John Doe");

This code retrieves the value associated with the key "John Doe" and stores it in the johnsGrade variable.

To remove a key-value pair from the map, you can use the remove method:

grades.remove("John Doe");

This code removes the key-value pair with the key "John Doe" from the grades map.

Maps support a variety of advanced operations, including iterating over the keys or values in the map, checking if a key or value is present in the map, and getting the size of the map. Let's take a look at some examples.

To iterate over the keys in a map, you can use the keySet method:

```
for (String name : grades.keySet()) {
    System.out.println(name);
}
```

This code iterates over the keys in the grades map and prints them out.

To iterate over the values in a map, you can use the values method:

```
for (int grade : grades.values()) {
    System.out.println(grade);
}
```

This code iterates over the values in the grades map and prints them out.

To check if a key or value is present in the map, you can use the containsKey and containsValue methods, respectively:

```
if (grades.containsKey("John Doe")) {
    System.out.println("John's grade is present in the map");
}


if (grades.containsValue(90)) {
    System.out.println("There is at least one student with a grade of 90");
}
```

These codes check if the key "John Doe" or the value 90 is present in the grades map.

To get the size of the map, you can use the size method:

int size = grades.size();

The size variable will now contain the number of key-value pairs in the grades map. If the grades map is empty, then size will be 0.

You can remove an entry from a map using the remove() method. The remove() method takes the key of the entry to be removed as an argument. For example, the following code removes the entry with key "John" from the grades map:

grades.remove("John");

After this code is executed, the grades map will no longer contain the entry with key "John".

You can check if a map contains a specific key or value using the containsKey() and containsValue() methods. The containsKey() method takes a key as an argument and returns true if the map contains an entry with that key, otherwise it returns false. The containsValue() method takes a value as an argument and returns true if the map contains at least one entry with that value, otherwise it returns false. For example, the following code checks if the grades map contains an entry with key "John":

```
if (grades.containsKey("John")) {
    System.out.println("John's grade is " + grades.get("John"));
} else {
    System.out.println("John is not in the map");
}
```

If the grades map contains an entry with key "John", then the code prints out "John's grade is " followed by the value associated with the "John" key. Otherwise, the code prints out "John is not in the map".

You can iterate over the entries in a map using a for-each loop. Each entry in the map is represented as a Map.Entry object, which has getKey() and getValue() methods that return the key and value of the entry, respectively. For example, the following code iterates over the entries in the grades map and prints out each key-value pair:

```
for (Map.Entry<String, Integer> entry : grades.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue());
}
```

This code iterates over the grades map and assigns each entry to the entry variable. Inside the loop, the code uses the getKey() and getValue() methods of the entry variable to get the key and value of the current entry, respectively. The code then prints out the key-value pair using the println() method.

Maps are an important data structure in Java that allow you to store key-value pairs. They can be used to implement a variety of algorithms and data structures, such as hash tables and graphs. In this chapter, we covered the basics of maps, including how to create a map, add and remove entries from a map, check if a map contains a key or value, and iterate over a map. With this knowledge, you should be able to use maps effectively in your own programs.

Sorted Maps

In the previous chapter, we learned about maps, which allow us to associate keys with values. In this chapter, we will dive into sorted maps, a special type of map that sorts its entries based on their keys. This makes it easier to perform certain operations, such as finding the highest or lowest key, or finding all keys within a certain range.

Java provides the SortedMap interface, which extends the Map interface and adds additional methods for dealing with sorted maps. The most important of these methods are:

Comparator<? super K> comparator(): returns the comparator used to sort the keys, or null if the natural ordering of the keys is used.

K firstKey(): returns the first (i.e., lowest) key in the map.

K lastKey(): returns the last (i.e., highest) key in the map.

SortedMap<K, V> headMap(K toKey): returns a view of the portion of the map whose keys are strictly less than toKey.

SortedMap<K, V> tailMap(K fromKey): returns a view of the portion of the map whose keys are greater than or equal to fromKey.

SortedMap<K, V> subMap(K fromKey, K toKey): returns a view of the portion of the map whose keys range from fromKey, inclusive, to toKey, exclusive.

The SortedMap interface is implemented by several classes in the Java Collections Framework, including TreeMap, which we will focus on in this chapter.

TreeMap is a class that implements the SortedMap interface using a red-black tree data structure. Like other map classes, it allows you to store key-value pairs and retrieve the value associated with a given key. However, TreeMap sorts its entries by the keys, using either the natural ordering of the keys or a custom comparator.

To create a TreeMap, you can use the following constructor:

TreeMap<K, V> treeMap = new TreeMap<>();

This creates a TreeMap that sorts its entries based on the natural ordering of the keys. If the keys are not comparable or you want to use a custom ordering, you can use the following constructor:

TreeMap<K, V> treeMap = new TreeMap<>(Comparator<? super K> comparator);

This creates a TreeMap that uses the given comparator to sort its entries.

To insert a key-value pair into a TreeMap, you can use the put() method:

treeMap.put(key, value);

To retrieve the value associated with a given key, you can use the get() method:

V value = treeMap.get(key);

Like other map classes, TreeMap provides several methods for iterating over its entries. However, because the entries are sorted by the keys, the iteration order is predictable.

To iterate over the keys in ascending order, you can use the keySet() method:

```
for (K key : treeMap.keySet()) {
    // do something with key
}
```

To iterate over the values in ascending order, you can use the values() method:

```
for (V value : treeMap.values()) {
    // do something with value
```

}

To iterate over the entries in ascending order, you can use the entrySet() method:

```
for (Map.Entry<K, V> entry : treeMap.entrySet()) {
    K key = entry.getKey();
```

Hash Tables

A hashtable is a data structure that can store a collection of key-value pairs. Hashtables are often used to store data that is not ordered, such as a list of tasks to be completed or a list of files.

In Java, hashtables are implemented using the Hashtable class. The Hashtable class defines a number of methods for adding, removing, and accessing elements in a hashtable.

Some of the most common methods in the Hashtable class include:

- put(): Adds a key-value pair to the hashtable.
- get(): Gets the value associated with a specific key in the hashtable.
- remove(): Removes a key-value pair from the hashtable.
- size(): Gets the number of key-value pairs in the hashtable.

Hashtables are a powerful tool that can be used to store and manipulate data. By understanding how hashtables work, you can write more efficient and effective code.

Here are some examples of how hashtables can be used:

- A hashtable can be used to store a list of tasks to be completed.
- A hashtable can be used to store a list of files.
- A hashtable can be used to store a list of websites to visit.
- A hashtable can be used to store a list of friends on social media.

Hashtables are a versatile data structure that can be used to store a variety of data. By understanding how hashtables work, you can write more efficient and effective code.

Here are some additional details about hashtables:

- Hashtables are implemented using hash tables, which are data structures that store data based on a key.

- Hashtables are synchronized, which means that all access to the hashtable is synchronized. This can be useful for preventing race conditions in multithreaded applications.
- Hashtables are a good choice for storing data that is not ordered, such as a list of tasks to be completed or a list of files.
- Hashtables are a good choice for storing data that needs to be frequently accessed by key.

Here are some tips for using hashtables:

- Use hashtables when you need to store a collection of elements that is not ordered.
- Use hashtables when you need to be able to add and remove elements from the collection frequently.
- Use hashtables when you need to be able to access elements in the collection by key.

Hashtables are a powerful tool that can be used to store and manipulate data. By understanding how hashtables work and using them correctly, you can write more efficient and effective code.

Here are some of the limitations of hashtables:

- Hashtables can be slow if the hash function is not good.
- Hashtables can be slow if the hashtable is full.
- Hashtables are not thread-safe by default.

Here are some of the advantages of hashtables:

- Hashtables are very efficient for storing and retrieving data by key.
- Hashtables are very scalable.
- Hashtables are very versatile.

Overall, hashtables are a powerful tool that can be used to store and manipulate data. By understanding the limitations and advantages of hashtables, you can use them effectively in your Java programs.

Set

A set is a data structure that can store a collection of elements. Sets are often used to store data that is unordered and unique, such as a list of names or a list of numbers.

In Java, sets are implemented using the Set interface. The Set interface defines a number of methods for adding, removing, and accessing elements in a set.

Some of the most common methods in the Set interface include:

- add(): Adds an element to the set.
- remove(): Removes an element from the set.
- contains(): Checks if an element is contained in the set.
- size(): Gets the number of elements in the set.

Sets are a powerful tool that can be used to store and manipulate data. By understanding how sets work, you can write more efficient and effective code.

Here are some examples of how sets can be used:

- A set can be used to store a list of tasks to be completed.
- A set can be used to store a list of files.
- A set can be used to store a list of websites to visit.
- A set can be used to store a list of friends on social media.

Sets are a versatile data structure that can be used to store a variety of data. By understanding how sets work and using them correctly, you can write more efficient and effective code.

Here are some additional details about sets:

- Sets are implemented using hash tables, which are data structures that store data based on a key.

- Sets are not synchronized, which means that multiple threads can access the set at the same time. This can lead to race conditions, which are errors that can occur when multiple threads are trying to access the same data at the same time.
- Sets are a good choice for storing data that is unordered and unique, such as a list of tasks to be completed or a list of files.
- Sets are a good choice for storing data that needs to be frequently accessed by key.

Here are some tips for using sets:

- Use sets when you need to store a collection of elements that is unordered and unique.
- Use sets when you need to be able to add and remove elements from the collection frequently.
- Use sets when you need to be able to access elements in the collection by key.

Sets are a powerful tool that can be used to store and manipulate data. By understanding how sets work and using them correctly, you can write more efficient and effective code.

Here are some of the limitations of sets:

- Sets can be slow if the hash function is not good.
- Sets can be slow if the set is full.
- Sets are not thread-safe by default.

Here are some of the advantages of sets:

- Sets are very efficient for storing and retrieving data by key.
- Sets are very scalable.
- Sets are very versatile.

Overall, sets are a powerful tool that can be used to store and manipulate data. By understanding the limitations and advantages of sets, you can use them effectively in your Java programs.

Multiset

A multiset is a data structure that can store a collection of elements. Multisets are similar to sets, but they allow duplicate elements.

In Java, multisets are implemented using the Multiset interface. The Multiset interface defines a number of methods for adding, removing, and accessing elements in a multiset.

Some of the most common methods in the Multiset interface include:

- add(): Adds an element to the multiset.
- remove(): Removes an element from the multiset.
- count(): Gets the number of times an element appears in the multiset.
- size(): Gets the total number of elements in the multiset.

Multisets are a powerful tool that can be used to store and manipulate data. By understanding how multisets work, you can write more efficient and effective code.

Here are some examples of how multisets can be used:

- A multiset can be used to store a list of tasks to be completed, where each task can appear multiple times.
- A multiset can be used to store a list of files, where each file can appear multiple times.
- A multiset can be used to store a list of websites to visit, where each website can appear multiple times.
- A multiset can be used to store a list of friends on social media, where each friend can appear multiple times.

Multisets are a versatile data structure that can be used to store a variety of data. By understanding how multisets work and using them correctly, you can write more efficient and effective code.

Here are some additional details about multisets:

- Multisets are implemented using hash tables, which are data structures that store data based on a key.
- Multisets are not synchronized, which means that multiple threads can access the multiset at the same time. This can lead to race conditions, which are errors that can occur when multiple threads are trying to access the same data at the same time.
- Multisets are a good choice for storing data that is unordered and allows duplicate elements, such as a list of tasks to be completed or a list of files.
- Multisets are a good choice for storing data that needs to be frequently accessed by key.

Here are some tips for using multisets:

- Use multisets when you need to store a collection of elements that is unordered and allows duplicate elements.
- Use multisets when you need to be able to add and remove elements from the collection frequently.
- Use multisets when you need to be able to access elements in the collection by key.

Multisets are a powerful tool that can be used to store and manipulate data. By understanding how multisets work and using them correctly, you can write more efficient and effective code.

Here are some of the limitations of multisets:

- Multisets can be slow if the hash function is not good.
- Multisets can be slow if the multiset is full.
- Multisets are not thread-safe by default.

Here are some of the advantages of multisets:

- Multisets are very efficient for storing and retrieving data by key.
- Multisets are very scalable.
- Multisets are very versatile.

Overall, multisets are a powerful tool that can be used to store and manipulate data. By understanding the limitations and advantages of multisets, you can use them effectively in your Java programs.

Multimap

A multimap is a data structure that can store a collection of key-value pairs. Multimaps are similar to maps, but they allow multiple values to be associated with a single key.

In Java, multimaps are implemented using the Multimap interface. The Multimap interface defines a number of methods for adding, removing, and accessing elements in a multimap.

Some of the most common methods in the Multimap interface include:

- put(): Adds a key-value pair to the multimap.
- get(): Gets the values associated with a specific key in the multimap.
- remove(): Removes a key-value pair from the multimap.
- size(): Gets the number of key-value pairs in the multimap.

Multimaps are a powerful tool that can be used to store and manipulate data. By understanding how multimaps work, you can write more efficient and effective code.

Here are some examples of how multimaps can be used:

- A multimap can be used to store a list of tasks to be completed, where each task can be associated with multiple people who are responsible for completing it.
- A multimap can be used to store a list of files, where each file can be associated with multiple users who have access to it.
- A multimap can be used to store a list of websites to visit, where each website can be associated with multiple tags that describe it.
- A multimap can be used to store a list of friends on social media, where each friend can be associated with multiple interests.
- Multimaps are a versatile data structure that can be used to store a variety of data. By understanding how multimaps work and

using them correctly, you can write more efficient and effective code.

Here are some additional details about multimaps:

- Multimaps are implemented using hash tables, which are data structures that store data based on a key.
- Multimaps are not synchronized, which means that multiple threads can access the multimap at the same time. This can lead to race conditions, which are errors that can occur when multiple threads are trying to access the same data at the same time.
- Multimaps are a good choice for storing data that is unordered and allows duplicate elements, such as a list of tasks to be completed or a list of files.
- Multimaps are a good choice for storing data that needs to be frequently accessed by key.

Here are some tips for using multimaps:

- Use multimaps when you need to store a collection of elements that is unordered and allows duplicate elements.
- Use multimaps when you need to be able to add and remove elements from the collection frequently.
- Use multimaps when you need to be able to access elements in the collection by key.

Multimaps are a powerful tool that can be used to store and manipulate data. By understanding how multimaps work and using them correctly, you can write more efficient and effective code.

Here are some of the limitations of multimaps:

- Multimaps can be slow if the hash function is not good.
- Multimaps can be slow if the multimap is full.
- Multimaps are not thread-safe by default.

Here are some of the advantages of multimaps:

- Multimaps are very efficient for storing and retrieving data by key.
- Multimaps are very scalable.
- Multimaps are very versatile.

Overall, multimaps are a powerful tool that can be used to store and manipulate data. By understanding the limitations and advantages of multimaps, you can use them effectively in your Java programs.