MultiMap Implementation in OpenJDK

Nishant Yadav

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the Other Computer Engineering Commons
MultiMap Implementation in OpenJDK

A Project Report

Presented to

The Department of Computer Science
San José State University

In Partial Fulfillment

of the Requirements of the Class

CS 298

By

Nishant Yadav

Dec 2023
The Designated Project Committee Approves the Project Titled

Multimap Implementation in OpenJDK

by

Nishant Yadav

Approved for the Department of Computer Science

San Jose State University

December 2023

Dr. Robert Chun
Department of Computer Science

Dr. Katerina Potika
Department of Computer Science

Dr. Thomas Austin
Department of Computer Science
ACKNOWLEDGMENT

I would like to thank you and express my gratitude to Dr. Robert Chun, for his guidance throughout my final project. His constant support, motivation, and insightful feedback have been invaluable resources for me. I am truly grateful for his dedication and patience during the whole process.

I would also like to thank and express my gratitude to my committee members Dr. Katerina Potika, and Dr. Thomas Austin for their guidance, feedback, and time.

Finally, I would like to express my heartfelt gratitude to my family and friends for their unwavering support and motivation throughout my academic journey.
ABSTRACT

A key-value pair is an elementary data model in which a unique key is associated with a given value. This association between the key and the value allows for a quick lookup of data based on the key and hence is extensively used in programming languages, NoSQL databases, caches, session management, etc. In Java OpenJDK, this elementary data model is implemented by the interface Map, which allows efficient storage and retrieval of data but can only store a single value against each key. In this project, we have implemented a MultiMap data structure in OpenJDK which allows associating multiple values against a single key.

Implementation of the project consists of an interface MultiMap which is further implemented by an abstract class, and finally has three implementations ArrayListMultiMap, SetMultiMap, and TreeMultiMap. Each of these multimaps allows the association of multiple values against a single key and provides some unique properties for the associated values which will be discussed in the implementation part. The AbstractMultiMap class provides us with implementations of some methods and views that allow users to perform operations and queries on data in a multimap.

In the experiments section, we have compared the execution times of inserting different numbers of key-value pairs in various implementations of multimap. Mainly we have considered our implementation of MultiMap in OpenJDK, multimap in Google Guava library, and multimap in C++. From these experiments, we observed that our implementation of MultiMap in OpenJDK provided better performance for primitive datatypes such as Integer and Float, while Google Guava provided better performance for complex data structures such as String.

*Keywords: MultiMap, OpenJDK, Google Guava*
# Table of Contents

ACKNOWLEDGMENT .................................................................................................................. i

ABSTRACT ........................................................................................................................................ ii

I. INTRODUCTION .............................................................................................................................. 1

II. BACKGROUND .................................................................................................................................... 2

  2.1 Non-relational Databases (NoSQL) ................................................................................................. 3

  2.2 Redis Database ................................................................................................................................. 5

  2.3 Amazon DynamoDB ......................................................................................................................... 8

  2.4 Chronicle Map .................................................................................................................................. 10

  2.5 Google Guava ................................................................................................................................... 11

  2.6 Multimap in C++ ............................................................................................................................. 12

III. OPENJDK ........................................................................................................................................... 13

IV. IMPLEMENTATION OF MULTIMAP ................................................................................................. 17

  4.1 Types of MultiMap Implemented ..................................................................................................... 18

    4.1.1 ArrayListMultiMap ....................................................................................................................... 19

    4.1.2 SetMultiMap ............................................................................................................................... 20

    4.1.3 TreeMultiMap ............................................................................................................................. 21

  4.2 Use cases of MultiMap ..................................................................................................................... 25

    1. Graph Algorithms ............................................................................................................................ 25

    2. In Statistical Analysis ....................................................................................................................... 25

    3. Geographic Data ............................................................................................................................. 26
V. EXPERIMENTS

5.1 ArrayListMultiMap

5.1.1 Insertion with Integer key and Integer value

5.1.2 Insertion with Float key and Float value

5.1.3 Insertion with String key and String value

5.2 SetMultiMap

5.2.1 Insertion with Integer key and Integer value

5.2.2 Insertion with Float key and Float value

5.2.3 Insertion with String key and String value

5.3 TreeMultiMap

5.3.1 Insertion with Integer key and Integer value

5.3.2 Insertion with Float key and Float value

5.3.3 Insertion with String key and String value

VI. BUILDING THE OPENJDK

5.1 Hardware Requirements for Building OpenJDK

5.2 Operating System Requirements for Building OpenJDK

5.3 Toolchains Requirement for Building OpenJDK

5.4 Boot JDK Requirements

VII. CONCLUSION

VIII. FUTURE WORK

8.1 Improving the Performance of Complex Data Structures

8.2 Return Types for Values of All Implementations
8.3 A Key-Value Datastore with Support for MultiMap ........................................45

REFERENCES .................................................................................................................46
LIST OF TABLES

Table 1: Implementations of Multimap in Guava ................................................................. 12
Table 2: Implementations of MultiMap in OpenJDK ............................................................ 18
Table 3: Properties and Methods Supported by MultiMap ..................................................... 24
Table 4: Insertion times for Integer datatype ........................................................................... 27
Table 5: Insertion times for Float datatype ............................................................................. 29
Table 6: Insertion times for String datatype ............................................................................ 30
Table 7: Insertion times for Integer datatype ........................................................................... 31
Table 8: Insertion times for Float datatype ............................................................................. 32
Table 9: Insertion times for String datatype ............................................................................ 33
Table 10: Insertion times for Integer datatype ....................................................................... 34
Table 11: Insertion times for Float datatype .......................................................................... 35
Table 12: Insertion times for String datatype ......................................................................... 36
Table 13: Hardware Requirements for OpenJDK ................................................................. 38
Table 14: Operating System Requirements for OpenJDK .................................................... 39
Table 15: Toolchain Requirements for OpenJDK ................................................................. 39
LIST OF FIGURES

Figure 1. Key-value Pair .................................................................................................................. 1
Figure 2: Redis Implementation as Cache ...................................................................................... 8
Figure 3: Key-value Pairs in MultiMap ........................................................................................... 14
Figure 4: MultiMap Implementation Hierarchy ............................................................................... 17
Figure 5: ArrayListMultiMap with Integer Key Values ................................................................. 20
Figure 6: SetMultiMap with Integer Key Values and Unique Values .......................................... 21
Figure 7: Integer Key Values TreeMultiMap with Natural Ordering ........................................... 22
Figure 8: Integer Key Values TreeMultiMap with Reverse Natural Ordering ............................. 22
Figure 9: String Key Values TreeMultiMap with Natural Ordering ............................................. 22
Figure 10: Insertion times for Integer datatype ........................................................................... 28
Figure 11: Insertion times for Float datatype ............................................................................... 29
Figure 12: Insertion times for String datatype ............................................................................. 30
Figure 13: Insertion times for Integer datatype ........................................................................... 31
Figure 14: Insertion times for Float datatype ............................................................................... 32
Figure 15: Insertion times for String datatype ............................................................................. 33
Figure 16: Insertion times for Integer datatype ........................................................................... 34
Figure 17: Insertion times for Float datatype ............................................................................... 35
Figure 18: Insertion times for String datatype ............................................................................. 36
Figure 19: Bash Configure Command Output - Configuration Details ....................................... 40
Figure 20: Make Images and Version Commands Output ............................................................. 41
Figure 21: OpenJDK path in .zshrc file ....................................................................................... 42
Figure 22: Java Version after Building OpenJDK .......................................................................... 42
I. INTRODUCTION

A key-value pair is a data model which consists of two correlated values the key and the value. These two values do not correlate and are not related, the key is used just to fetch the corresponding data from the database. Most non-relational databases use key-value pairs as the data structure to store data rather than using tables as used by relational databases. Unlike relational databases, which provide users with the ability to define complex schemas, aggregations, and relationships, non-relational databases provide the user with flexible and fast read-and-write capabilities. Key-value-based databases fetch data based on the key and return the value without doing any complex operations.

Key-value pairs are also used as a data structure in multiple programming languages which have two parts: a unique identifier key and a value associated with the key. A map is such an implementation of a data structure that allows efficient storage and retrieval of the key-value pair. In Java, we have the Map interface which is implemented by multiple classes and

![Figure 1. Key-value Pair](image-url)
provides different implementations including HashMap, TreeMap, and LinkedHashMap. In C++, the std::map container provides an implementation of the map data structure. The main benefit of using a map data structure is quick storage and retrieval of data in an efficient manner. Maps also can store different kinds of data both in the key and the value, hence supporting a wide range of use cases. Overall map is a fundamental data structure in many programming languages and greatly improves the performance and code organization.

II. BACKGROUND
This chapter provides a literature survey and background about various key-value data stores, their benefits, and other details. The first section covers NoSQL databases, their features, and different types of NoSQL databases.

The first section describes an open-source in-memory distributed data structure store, Redis [4]. Various use cases of Redis include message broker, cache, and in-memory key-value database with different levels of durability. The following section covers Amazon DynamoDB which is another key-value database that also supports document-based data structures. Dynamo DB is a proprietary NoSQL database available only on Amazon Web Services. The third section is about Apache Cassandra, which is another open-source and free wide-column distributed datastore that is designed to handle large amounts of data across many servers.

2.1 Non-relational Databases (NoSQL)

Relational Database Management Systems (RDBMS) are widely used database systems in most applications. Using RDBMS provides various advantages which include simplicity to use and learn, Structured Query Language (SQL) support, standardized data definition language (DDL) and data manipulation language (DML) design, support for ACID constraints, and security [3]. Even with so many advantages and ease of usage, there are still many drawbacks of RDBMS which make it unsuitable for certain types of applications. These drawbacks are listed below –

1. SQL-based databases do not support the storage of unstructured data.

2. No support for scalability, and distributed processing.

3. Complex relationships between various entities.

4. Since data is present in different tables, complex queries may be required to retrieve data.
5. Most of today’s application generates data at a high rate and have no structure, hence RDBMS is not an appropriate choice for such applications.

To solve all these problems, we can use non-relational databases or NoSQL databases which do not use relational format to store data. These databases do not require defining complex relationships, and usage of complex queries to retrieve related data. When working on large amounts of unstructured data, we require a system that can run on distributed systems. NoSQL databases can easily run on a distributed cluster and can be horizontally scaled, whereas RDBMS can only be scaled vertically. Unlike relational databases which use tables having rows and columns to store data, NoSQL uses various data models such as key-value pairs, documents, graphs, and objects but doesn’t support ACID constraints on transactions. Some of the important features of NoSQL databases are given below –

1. Non-relational: NoSQL databases do not support operations such as joins that are usually supported by RDBMS databases.

2. Distributed: Most NoSQL databases support the storage of data in distributed systems.

3. Horizontally Scalable: RDBMS can only be scaled vertically, while NoSQL databases can be scaled horizontally since they can run on distributed clusters.

4. Schema-free: NoSQL databases are schema-free; hence data can be stored flexibly.

5. Replication: NoSQL databases allow easy replication of data in shards, hence providing high availability.

6. BASE Properties: ACID constraints are not followed by NoSQL databases, instead these databases follow BASE properties.

   - Basically Available: Availability of application at all the time
   - Soft State: Consistency is not mandatory
   - Eventual Consistency: Application should be consistent eventually
NoSQL databases provide different types of data models, each of which can be used as per the requirement. The most common data models are briefly described below –

1. **Key-Value Based**: This type of database uses a key-value-based data model to store data. Each key has a single or multiple values associated with it. The key should always be unique, and data can be stored in a schema-less manner.

2. **Document-Based**: This type of database stores data in documents, where each document is associated with a key. The user is required to specify the structure of data and documents rather than the database.

3. **Column Family Stores Databases**: This type of database stores data in column-oriented tables.

4. **Graph Databases**: This type of database is based on graph theory. The data is stored in a graph data model where the nodes and edges of the graph contain the data. These are mainly used to store semi-structured data and applications include storing data related to social networks, security, network, etc.

### 2.2 Redis Database

REmote DIctionary Server (Redis) is an in-memory key-value-based datastore that can be used as a cache, database, and message broker. Redis is written in C and hence provides high performance, and scalability, and can run without any external dependencies [5]. Redis is also known as a data structure server since its keys can contain different types of data such as strings, sets, sorted sets, hashes, lists, bitmaps, geospatial indexes, and streams. Redis is commonly used as a cache in applications, as it allows read and write operations with very high speed since it stores data in memory instead of putting it on disk. Some of the advantages of Redis are mentioned below –
1. **Atomic Operations:** Operations such as appending to a string, pushing an element to a list, etc. are atomic in Redis. This means that if data is accessed by two clients concurrently, Redis will contain the latest data.

2. **Top Performance:** Since Redis stores data in an in-memory dataset, it provides best-in-class performance. Also, the user can persist data to disk as per requirement.

3. **Replication:** Redis provides the capability for asynchronous replication in which it uses leader-follower replication which can be configured easily.

4. **Data Types:** As mentioned briefly above, Redis supports many data types.

5. **Use Cases:** Redis supports multiple datatypes for data storage, hence it can be used for multiple purposes such as cache, key-value database, message queue, etc.

    Redis provides support for various types of data structures to be stored as a value in addition to strings [6]. The key in the key-value pair is a binary-safe string. Below are the data types which can be used in values –

    1. **String:** This is the most basic data type in Redis. They are often used for caching and can hold a maximum of 512 MB of data. Operations on strings are highly efficient with constant time complexity for most operations.

    2. **Redis List:** List of strings that are stored in linked lists, hence are sorted by the order in which data is inserted. Because they are implemented as linked lists, they can also be used as stacks and queues. Used for queue management systems.

    3. **Sets:** Like sets in Java, sets in Redis are unordered in nature having unique values. Adding and removing data are constant time operations.

    4. **Hashes:** Like HashMap in Java, hashes are field-value pairs and are generally used to store objects.
5. *Sorted Sets*: Extension of sets, this also contains unique strings but also maintains the order in which data is entered.

6. *Streams*: Streams in Redis are append-only in nature, and hence are generally used for logging events.

7. *Geospatial Indexes*: A special type of data structure that is used for storing geospatial coordinates and allows to perform operations on data.

8. *Bitmaps*: Extension of strings, which allows us to perform bitwise operations on strings. Use case includes maintaining permissions etc.

9. *Bitfields*: Binary-encoded Redis strings that support atomic operations and can be used for managing counters.

10. *HyperLogLog*: This is used in probabilities to calculate the cardinality of a set.

Redis is mostly used for implementing cache in applications. Since it uses an in-memory datastore, it improves performance by saving trips to the backend database whenever data is available in the cache. While implementing the Redis server is kept in between the backend database and the application, when the application needs any data, it first looks for that data in the Redis cache, and if it is not found then only it will look in the database. This type of architecture is shown below in the figure.
2.3 Amazon DynamoDB

Amazon DynamoDB is a proprietary NoSQL key-value-based database from Amazon that provides high performance and easy maintenance. Since DynamoDB is a fully managed database and is available as a Platform as a Service (PaaS) it significantly reduces the administrative overhead of the consumer [8]. Scaling in a distributed cluster, resource provisioning, cluster setup, and software patching are all done automatically and are abstract to the user. Along with its high performance, the ease of usage is one of the major selling points of DynamoDB.

The core components or data model of DynamoDB consists of tables, items, and attributes. Each table consists of collections of items, and each item consists of collections of attributes [9]. Each item in a table is uniquely identified using primary keys and DynamoDB provides additional performance and querying using secondary indexes. These core components are further explained in detail below –

1. Tables: Like other databases, DynamoDB also stores data in tables, which are collections of data.
2. **Items**: Each item is a group of attributes that is unique and holds data related to one entity. Items can be considered as the row equivalent of a relational database.

3. **Attributes**: This is the smallest and elementary unit of data in DynamoDB. As items are row equivalent, attributes can be considered column equivalent of a relational database.

DynamoDB supports multiple datatypes like Redis. The supported data types can be categorized into three categories mentioned below [10] –

1. **Scalar Types**: Scalar types are used when we need to put only one value. This is like primitive datatypes of Java. E.g., String, number, Boolean, null, and number.

2. **Document Types**: Document datatype can be used to store data similarly as it is present in JSON documents. We can define complex structures having nested data. This makes it easy to keep correlated data in a single document.

3. **Set Types**: Sets in DynamoDB can only hold unique values and the order of insertion is not preserved.

DynamoDB uses two types of indexes to provide fast data access to the user. Creating a primary key for each table is mandatory in DynamoDB, but creating a secondary key is optional and depends on the requirements. A primary key is always unique and is used to identify a unique item in any table. If the user does not define a primary key on a table, DynamoDB will create one automatically. One or more secondary indexes can also be created on a table, and this allows to query of the data and provides flexibility while querying the database.
2.4 Chronicle Map

Chronicle Map is an open-source key-value datastore developed by Chronicle Software that is used in trading and financial market applications. It is an in-memory key-value store that is very fast and provides low-latency access to data [11]. The main feature of Chronicle Map is that it is an in-memory data store and hence provides great read and write speeds which are crucial in financial applications. It also provides the user with the ability to persist the data on a disk when no more memory is available (RAM). Chronicle Map is implemented in Java and the main feature is that it uses off-heap memory to store data rather than using the Java heap memory. Since data is not stored in the heap memory, the Java garbage collector does not run often and hence application gets a significant performance boost. Some of the features of Chronicle Map are mentioned below [12] –

1. *Ultra-low Latency:* Chronicle Map provides very low latency for read and write operations since data is present in memory.

2. *High Concurrency:* Chronicle Map provides high read concurrency; no read operations are blocked by other read operations. Write operations scale along with the number of available hardware threads.

3. *Persistence to disk:* Even though Chronicle Map uses memory to store its data, it provides an option to persist data to disk.

4. *Multi-master Replication:* This feature is not available in open-source implementation. Chronicle Map can provide eventual consistency, full redundancy, and asynchronous replication using the “last write wins” strategy [12].

Two specific use cases where Chronicle Map is used in various production applications are described below –

10
1. *Real-time Trading Systems:* High-Frequency Trading (HFT) is required to perform many transactions in fractions of seconds. Since Chronicle Map uses memory to store data, it provides high-speed access to data. Also, since Chronicle Map does not use Java heap memory it saves the overhead of garbage collection and hence provides much better performance.

2. *High Concurrent Systems:* As mentioned above, Chronicle Map can support multiple read operations with no blocking, and non-blocking write operations that scale with available hardware. Also, the commercial version also supports multi-master replication in a distributed cluster.

### 2.5 Google Guava

Guava is an open-source implementation of various new collection types developed by Google. In addition to the collections present in core Java libraries, Guava adds an implementation of additional collection types such as multimap and multiset, graph library, immutable collections, caching, etc.

The collection that we will be focusing on is the Multimap implementation in Google Guava. Guava supports various operations which can be done on a Multimap, some of these are mentioned below –

1. **put(K, V):** Adds a new value associated with a key.
2. **putAll(K, Iterable<V>):** Adds multiple values associated with a key.
3. **remove(K, V):** Removes the value associated with a given key and returns the value as a result.
4. **removeAll(K):** Removes all the values associated with a given key and returns them as a result.
5. replaceValues(K, Iterable<V>): Removes all the existing values associated with a given key and puts new values in place of them.

Google Guava provides the following implementations for Multimap:

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Keys behave like</th>
<th>Values behave like</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayListMultimap</td>
<td>HashMap</td>
<td>ArrayList</td>
</tr>
<tr>
<td>HashMultimap</td>
<td>HashMap</td>
<td>HashSet</td>
</tr>
<tr>
<td>LinkedListMultimap</td>
<td>LinkedHashMap</td>
<td>LinkedList</td>
</tr>
<tr>
<td>LinkedHashMultimap</td>
<td>LinkedHashMap</td>
<td>LinkedHashSet</td>
</tr>
<tr>
<td>TreeMultimap</td>
<td>TreeMap</td>
<td>TreeSet</td>
</tr>
<tr>
<td>ImmutableListMultimap</td>
<td>ImmutableMap</td>
<td>ImmutableList</td>
</tr>
<tr>
<td>ImmutableSetMultimap</td>
<td>ImmutableMap</td>
<td>ImmutableSet</td>
</tr>
</tbody>
</table>

Table 1: Implementations of Multimap in Guava

2.6 Multimap in C++

The Standard Template Library (STL) of C++ also provides an implementation of a multimap data structure. The C++ implementation of Multimap uses binary search trees for storing the data. The multimap contains a sorted list of key-value pairs and the keys are sorted according to the comparison function provider [17], and it allows duplicate keys.
III. OPENJDK

OpenJDK is an open-source implementation of the Java SE platform guidelines and provides a development kit that is widely used in the Java software development community. It is maintained by a community of developers and provides a high-performing, reliable, and secure platform for developing Java-based applications. Java Collections Framework is one of the core components of OpenJDK which provides a set of built-in data structures which gives developers the flexibility to manipulate the data in different ways based on requirements. A collection can be defined as a group of similar objects that can be manipulated together. Some of the advantages of the collections framework are –

1. Programming Effort: The collections framework significantly reduces the programming effort for developers since it provides the implementation of commonly used data structures and algorithms.

2. Increased Performance and Memory Efficiency: Implementations in the collections framework are designed to be memory-efficient and use different techniques for better performance.

3. Rich Functionality: In addition to many data structures, the collections framework provides various ways to manipulate the data present in them such as sorting, searching, filtering, and iterating.
Overall, the OpenJDK and Java Collection Framework provide various tools that make it easy for developers to build new applications. Some of the data structures present in Java Collections Framework are –

1. ArrayList
2. LinkedList
3. Stack
4. HashSet
5. HashMap
6. Queue

The Map interface is one of the core components of the Java Collections Framework in OpenJDK and there are different implementations of the interface which provide unique performance benefits. The basic property of a map is that it stores a value against a key and allows quick access to values stored in it. Three general-purpose implementations of the Map interface are HashMap, TreeMap, and LinkedHashMap. Each implementation provides unique benefits and hence can be used in different scenarios.
1. HashMap

This is the most used implementation of the Map interface. HashMap stores the key-value pairs in a hash table, which allows fast and easy retrieval of values based on their keys. HashMap provides the average case time complexity of O(1) for basic operations such as insertion, retrieval, and deletion. We use HashMap when we need fast access and the order in which keys are stored is not of significance.

2. TreeMap

TreeMap is a tree-based implementation of the Map interface and stores the key-value pair in a sorted data structure. It provides the time complexity of O(logn) for basic operations such as insertion, retrieval, and deletion. A TreeMap has a higher memory overhead since keys are to be kept in sorted order. Use cases include scenarios where we need keys to be sorted.

3. LinkedHashMap

LinkedHashMap maintains the order of insertion of key-value pairs in the data structure. This implementation also provides a time complexity of O(1) for basic operations such as insertion, retrieval, and deletion. LinkedHashMap has an even higher memory overhead compared to TreeMap since it maintains the insertion order of key-value pairs.

Each implementation of the Map interface in OpenJDK is optimized for different use cases and provides performance as per requirement. For example, HashMap implementation is mostly used where we need fast retrieval of values based on their keys, TreeMap
implementation is used where the keys are required to be in sorted order, and LinkedHashMap can be used where the order of insertion of key-value pairs is important.
IV. IMPLEMENTATION OF MULTIMAP

There are many benefits of the Map interface in Java Collections Framework such as quick lookup for value stored, but there are some limitations. The implementations of Map interfaces such as HashMap, TreeMap, and LinkedHashMap can only store a single value against a given key. A MultiMap is an extended version of the Map data structure, where we can store and retrieve multiple values for a given key efficiently. Hence, a MultiMap saves a developer the overhead of creating a new data structure for the value of each new key added to the data structure. In this project, we have added support for MultiMap in the Java Collections Framework of OpenJDK. The figure below shows the hierarchy for the implemented MultiMap.

Figure 4: MultiMap Implementation Hierarchy
Some of the unique properties of MultiMap implementation are:

1. **MultiMap.get(key):** If at any point the multimap contains a key, it will always return a non-null or empty collection. If the key was never present, or it has been removed using the remove() method, it will return null.

2. **MultiMap.contains(key):** If at any point the multimap contains a key, it will return true since there will be an empty collection associated with it. If the key was never present, or it has been removed using the remove() method, it will return false.

3. **MultiMap.size():** This will return the number of values present in the multimap, not the number of keys as a map returns. If we want to get the number of keys, we can use MultiMap.keySet().size() method which will return us the distinct number of keys present.

4. **MultiMap.entries():** This method returns all entries in key-value pairs, where each key will be associated with every value present in the collection associated with the key.

5. **MultiMap.toString():** This method will return a more realistic string representation of the multimap, in which each key will have all the values associated with it that are present in the collection associated with the key.

### 4.1 Types of MultiMap Implemented

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Keys behave like</th>
<th>Values behave like</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayListMultiMap</td>
<td>HashMap</td>
<td>ArrayList</td>
</tr>
<tr>
<td>HashMultiMap</td>
<td>HashMap</td>
<td>HashSet</td>
</tr>
<tr>
<td>TreeSetMultiMap</td>
<td>TreeMap</td>
<td>TreeSet</td>
</tr>
</tbody>
</table>

Table 2: Implementations of MultiMap in OpenJDK
4.1.1 ArrayListMultiMap

The first type of multimap implemented in OpenJDK is the ArrayListMultiMap. ArrayListMultiMap is a normal HashMap in which the value is of type ArrayList. The key and value are of generic type and hence can contain any object type. ArrayListMultiMap provides two constructors to create a new ArrayListMultiMap, default and parameterized. These are explained below:

- Default Constructor

A multimap can be created by using the default constructor. The default constructor creates a multimap with an initial capacity of 16 and each key would have an initial capacity of 3. These values are present in DEFAULT_INITIAL_CAPACITY and DEFAULT_VALUES_PER_KEY variables.

```java
 MultiMap<Integer, Integer> multimap = new ArrayListMultiMap<>();
```

- Parameterized Constructor

A multimap can also be created using the parameterized constructor in which we can pass a value for the initial expected capacity and expected number of values for each key. This will override the default values.

```java
 MultiMap<Integer, Integer> multimap = new ArrayListMultiMap<>(3, 16);
```
4.1.2 SetMultiMap

SetMultiMap is another type of multimap implementation in OpenJDK. In a SetMultiMap the key can be of any Object type and the value is a HashSet, hence each key will have a collection of unique values associated with it. SetMultiMap also provides two ways to construct a multimap. These are explained below:

- Default Constructor

This is like ArrayListMultiMap and contains a multimap with an initial capacity of 16 and each key would have an initial capacity of 3.

```java
MultiMap<Integer, Integer> multimap = new SetMultiMap<>();
```

- Parameterized Constructor

This is also like the parameterized constructor of ArrayListMultiMap, and we can provide the initial expected capacity and expected number of values for each key.

```java
MultiMap<Integer, Integer> multimap = new SetMultiMap<>(3, 16);
```
4.1.3 TreeMultiMap

TreeMultiMap is an implementation of a multimap that stores the data in sorted order. The value for a key in TreeMultiMap is of type TreeSet. By default, the keys and values will be stored according to their natural ordering. A user can also create a custom comparator for the key and value which can be used to store the data in the required sorted order.

TreeMultiMap also provides two types of constructors which are explained below:

- **Default Constructor**

  The default constructor will create a new TreeMultiMap in which the keys and the values associated with them will be sorted according to their natural ordering.

  ```java
  MultiMap<Integer, Integer> multimap = new TreeMultiMap<>();
  ```

- **Parameterized Constructor**

  If we don’t want to use the default natural ordering, we can also specify how the key and values should be sorted. For this, we can use the parameterized constructor which takes in two Comparator objects and stores data based on the ordering used by it.
MultiMap<Integer, Integer> multimap = new TreeMultiMap<>(keyComparator, valueComparator);

Figure 7: Integer Key Values TreeMultiMap with Natural Ordering

Figure 8: Integer Key Values TreeMultiMap with Reverse Natural Ordering

Figure 9: String Key Values TreeMultiMap with Natural Ordering
Each implementation of the MultiMap interface provides various kinds of operations that can be done on them such as clearing a whole multimap, checking the equality of two multimaps, checking if a multimap is empty, checking if a multimap contains a value, size, etc. The table below, explains and lists all the methods/operations, their return types, and the description that the implementation of MultiMap supports.
<table>
<thead>
<tr>
<th>Method Name</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear()</td>
<td>void</td>
<td>iterates over the collections in values and removes them, finally clears the multimap</td>
</tr>
<tr>
<td>containsKey(Object key)</td>
<td>boolean</td>
<td>returns true is this multimap contains a multimapping for the specified key</td>
</tr>
<tr>
<td>containsValue(Object value)</td>
<td>boolean</td>
<td>returns true is this multimap multimaps one or more keys to the specified value</td>
</tr>
<tr>
<td>entries()</td>
<td>Collection&lt;Entry&lt;K, V&gt;&gt;</td>
<td>returns a collection view of the multimappings contained in this multimap</td>
</tr>
<tr>
<td>equals(Object o)</td>
<td>boolean</td>
<td>compares the specified object with this multimap for equality</td>
</tr>
<tr>
<td>get(Object key)</td>
<td>Collection&lt;V&gt;</td>
<td>returns the collection of value to which the specified key is mapped, or null if this multimap contains no multimapping for the key</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>boolean</td>
<td>returns true if this multimap contains no key-value multimappings</td>
</tr>
<tr>
<td>keySet()</td>
<td>Set&lt;K&gt;</td>
<td>returns a set view of the keys contained in this multimap</td>
</tr>
<tr>
<td>put(K key, V value)</td>
<td>boolean</td>
<td>associates the specified value with the specified key in this multimap</td>
</tr>
<tr>
<td>remove(Object key, Object value)</td>
<td>boolean</td>
<td>removes the value for a key from this multimap if it is present</td>
</tr>
<tr>
<td>remove(Object key)</td>
<td>Collection&lt;V&gt;</td>
<td>removes and returns the values associated with a key if present</td>
</tr>
<tr>
<td>size()</td>
<td>int</td>
<td>returns the number of key-value multimappings in this multimap</td>
</tr>
<tr>
<td>toString()</td>
<td>String</td>
<td>returns a string representation of this multimap</td>
</tr>
</tbody>
</table>

Table 3: Properties and Methods Supported by MultiMap
4.2 Use cases of MultiMap

A MultiMap can store multiple values against a given key, hence there are various use cases of a MultiMap. Some of them are explained below which we will use in our examples –

1. Graph Algorithms

A graph is a non-linear type of data structure that consists of vertices and edges. Sometimes we also use nodes to represent vertices. In code, a graph can be represented in two different ways, Adjacency Matrix, and Adjacency List. We use the List data structure to represent graphs in these representations. Here, we can use MultiMap to represent a graph instead of an adjacency list, since in an adjacency list we need to iterate over the whole list to find a given node. Using a MultiMap will allow us for quick lookup of the adjacent nodes of a given node by keeping the node as a key in MultiMap.

2. In Statistical Analysis

While working on statistical data, we often use multiple list or array variables to represent different factors of the data. We can use a MultiMap in such statistical analysis use cases where we have multiple variables in the data that are required by the model. Ideally, we would keep one list for a single variable, but we can use a single MultiMap for the representation of such data, where the key would be the attribute and the value will hold all the values for that attribute. Since we are using a map accessing any attribute is a quick operation. Hence, using a MultiMap we can represent the whole in a concise way that is easy to manipulate and understand.
3. Geographic Data

A MultiMap can also be used to store geographic data. The key can be the main location and value can hold all the locations of interest which a user needs. An example can be to store multiple points of interest related to a city such as restaurants, museums, and parks. A MultiMap would be a perfect data structure to store data of such format since we can get all the important information by just looking at the key. It would allow us to store multiple data points associated with a single geographic location and would also allow fast access.
V. EXPERIMENTS

This section includes some experiments on the new implementation and does some comparison with existing implementations. These experiments consist of a comparison of the execution time of various operations performed on different implementations of multimap. Mainly we would be comparing the time taken in the insertion operation with a different number of keys and data types. For each set of experiments, we performed 10 runs with random values and have considered the best case out of the 10 runs. We would do the comparison between our implementation of MultiMap in OpenJDK and the Multimap present in the Google Guava library.

5.1 ArrayListMultiMap

5.1.1 Insertion with Integer key and Integer value

This experiment consists of inserting key-value pairs of Integer type. We would start with 100 keys and each key will be associated with 100 values and will gradually increase the number of keys. For this experiment, we will be comparing the insertion times of Google Guava, our implementation of MultiMap in OpenJDK, and multimap in C++ STL.

<table>
<thead>
<tr>
<th>Count</th>
<th>Google Guava</th>
<th>OpenJDK MultiMap</th>
<th>C++ multimap</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>2 ms</td>
<td>1 ms</td>
<td>0 ms</td>
</tr>
<tr>
<td>100,000</td>
<td>9 ms</td>
<td>6 ms</td>
<td>0 ms</td>
</tr>
<tr>
<td>400,000</td>
<td>18 ms</td>
<td>16 ms</td>
<td>0 ms</td>
</tr>
<tr>
<td>800,000</td>
<td>30 ms</td>
<td>26 ms</td>
<td>0 ms</td>
</tr>
<tr>
<td>1,000,000</td>
<td>33 ms</td>
<td>29 ms</td>
<td>1 ms</td>
</tr>
</tbody>
</table>

Table 4: Insertion times for Integer datatype
As we can see in the above graph, the execution time for insertion in C++ STL multimap is negligible when compared to the other two, hence we won’t be considering this in other experiments. It can also be noted that the time taken is increasing linearly with the number of key values inserted. We can also see that for Integer keys and values, our implementation in OpenJDK provides better performance compared to the one in Guava.

### 5.1.2 Insertion with Float key and Float value

This experiment consists of inserting key-value pairs of Float type. We would start with 100 keys and each key will be associated with 100 values and will gradually increase the number of keys.

<table>
<thead>
<tr>
<th>Count</th>
<th>Google Guava</th>
<th>OpenJDK MultiMap</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>2 ms</td>
<td>2 ms</td>
</tr>
<tr>
<td>100,000</td>
<td>10 ms</td>
<td>9 ms</td>
</tr>
</tbody>
</table>
As we can observe from the above graph, our implementation of MultiMap in OpenJDK performs better when compared to the implementation in Guava.

### 5.1.3 Insertion with String key and String value

This experiment consists of inserting key-value pairs of String type. For this experiment, we would start with 100 keys and each key will be associated with 100 values and will gradually increase the number of keys.
Count | Google Guava | OpenJDK MultiMap
--- | --- | ---
10,000 | 9 ms | 13 ms
100,000 | 26 ms | 32 ms
400,000 | 58 ms | 62 ms
800,000 | 79 ms | 89 ms
1,000,000 | 83 ms | 89 ms

Table 6: Insertion times for String datatype

From the above table and figure, it can be noticed that guava implementation of multimap provides better performance. Hence, we can say that guava implementation handles complex data structures better compared to OpenJDK implementation.
5.2 SetMultiMap

5.2.1 Insertion with Integer key and Integer value

This experiment consists of inserting key-value pairs of Integer type in SetMultiMap. For this experiment, we would have 100 values against each key and will increase the number of keys gradually. A higher number of keys and values leads to Heap Space exhauster error.

<table>
<thead>
<tr>
<th>Count</th>
<th>Google Guava</th>
<th>OpenJDK MultiMap</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>3 ms</td>
<td>2 ms</td>
</tr>
<tr>
<td>100,000</td>
<td>12 ms</td>
<td>10 ms</td>
</tr>
<tr>
<td>400,000</td>
<td>34 ms</td>
<td>27 ms</td>
</tr>
<tr>
<td>800,000</td>
<td>49 ms</td>
<td>46 ms</td>
</tr>
<tr>
<td>1,000,000</td>
<td>60 ms</td>
<td>57 ms</td>
</tr>
</tbody>
</table>

Table 7: Insertion times for Integer datatype

Figure 13: Insertion times for Integer datatype
From the table and figure above, we can infer that both implementations provide similar performance to 200,000 key-value pairs. After that, as the number of key-value pairs increases OpenJDK-based implementation of multimap provides better performance.

5.2.2 Insertion with Float key and Float value

This experiment is like the last one, but instead of using Integer datatype we will be using Float datatype for keys and values.

<table>
<thead>
<tr>
<th>Count</th>
<th>Google Guava</th>
<th>OpenJDK MultiMap</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>6 ms</td>
<td>5 ms</td>
</tr>
<tr>
<td>100,000</td>
<td>17 ms</td>
<td>15 ms</td>
</tr>
<tr>
<td>400,000</td>
<td>54 ms</td>
<td>52 ms</td>
</tr>
<tr>
<td>800,000</td>
<td>83 ms</td>
<td>87 ms</td>
</tr>
<tr>
<td>1,000,000</td>
<td>92 ms</td>
<td>95 ms</td>
</tr>
</tbody>
</table>

Table 8: Insertion times for Float datatype

Figure 14: Insertion times for Float datatype
From the above graph, we can see that OpenJDK-based implementation performs better till 600,000 key-value pairs, but after that, Guava provides slightly better performance.

5.2.3 Insertion with String key and String value

This experiment consists of inserting key-value pairs of String type in SetMultiMap. For this experiment, we start with 100 keys and go to 10,000 keys each having 100 values. A higher number of keys and values leads to Heap Space exhauster error.

<table>
<thead>
<tr>
<th>Count</th>
<th>Google Guava</th>
<th>OpenJDK MultiMap</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>12 ms</td>
<td>13 ms</td>
</tr>
<tr>
<td>100,000</td>
<td>32 ms</td>
<td>35 ms</td>
</tr>
<tr>
<td>400,000</td>
<td>74 ms</td>
<td>83 ms</td>
</tr>
<tr>
<td>800,000</td>
<td>110 ms</td>
<td>113 ms</td>
</tr>
<tr>
<td>1,000,000</td>
<td>125 ms</td>
<td>137 ms</td>
</tr>
</tbody>
</table>

Table 9: Insertion times for String datatype

![Figure 15: Insertion times for String datatype](image-url)
From the above graph, we can see that the Guava implementation of set based multimap provides better performance compared to OpenJDK implementation when we use String for key and values.

5.3 TreeMultiMap

5.3.1 Insertion with Integer key and Integer value

This experiment consists of inserting key-value pairs of Integer type in TreeMultiMap. We start with 100 keys and go to 1,000,000 keys each having 100 values.

<table>
<thead>
<tr>
<th>Count</th>
<th>Google Guava</th>
<th>OpenJDK MultiMap</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>4 ms</td>
<td>5 ms</td>
</tr>
<tr>
<td>100,000</td>
<td>18 ms</td>
<td>17 ms</td>
</tr>
<tr>
<td>400,000</td>
<td>54 ms</td>
<td>47 ms</td>
</tr>
<tr>
<td>800,000</td>
<td>89 ms</td>
<td>85 ms</td>
</tr>
<tr>
<td>1,000,000</td>
<td>105 ms</td>
<td>102 ms</td>
</tr>
</tbody>
</table>

Table 10: Insertion times for Integer datatype

![Figure 16: Insertion times for Integer datatype](image)
From the graph above, we can see that both implementations provide similar performance for smaller inputs, but OpenJDK MultiMap provides better performance when input size increases.

### 5.3.2 Insertion with Float key and Float value

This experiment is the same as the last one, but instead of using Integer, we would be using Float datatype.

<table>
<thead>
<tr>
<th>Count</th>
<th>Google Guava</th>
<th>OpenJDK MultiMap</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>4 ms</td>
<td>5 ms</td>
</tr>
<tr>
<td>100,000</td>
<td>19 ms</td>
<td>18 ms</td>
</tr>
<tr>
<td>400,000</td>
<td>57 ms</td>
<td>51 ms</td>
</tr>
<tr>
<td>800,000</td>
<td>92 ms</td>
<td>89 ms</td>
</tr>
<tr>
<td>1,000,000</td>
<td>116 ms</td>
<td>109 ms</td>
</tr>
</tbody>
</table>

Table 11: Insertion times for Float datatype

![Figure 17: Insertion times for Float datatype](image-url)
From the graph above, we can see that we get similar performance for smaller input size, but OpenJDK MultiMap provides better performance with increase in input size.

5.3.3 Insertion with String key and String value

This experiment consists of inserting key-value pairs of String type in TreeMultiMap. In this experiment, we start with 100 keys and 100 values for each key and go up to 1,000,000 keys.

<table>
<thead>
<tr>
<th>Count</th>
<th>Google Guava</th>
<th>OpenJDK MultiMap</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>13 ms</td>
<td>16 ms</td>
</tr>
<tr>
<td>100,000</td>
<td>50 ms</td>
<td>53 ms</td>
</tr>
<tr>
<td>400,000</td>
<td>121 ms</td>
<td>115 ms</td>
</tr>
<tr>
<td>800,000</td>
<td>190 ms</td>
<td>183 ms</td>
</tr>
<tr>
<td>1,000,000</td>
<td>222 ms</td>
<td>215 ms</td>
</tr>
</tbody>
</table>

Table 12: Insertion times for String datatype

![Figure 18: Insertion times for String datatype](image-url)
From the graph above, we can see that both implementation of MultiMap provides similar performance when we use String as a datatype. But OpenJDK MultiMap still performs slightly better than Guava MultiMap.

From all the above experiments, we can note that when it comes to primitive data structures such as Integer, our implementation of multimap performs better. But when it comes to data of complex type such as String the Guava implementation provides either similar or better performance when compared to our implementation.
VI. BUILDING THE OPENJDK

OpenJDK is a complex piece of software and there are certain software and hardware requirements to build the OpenJDK before we can use it. Ideally, one would download an already existing prebuilt OpenJDK package for your operating system and would install it on your personal computer. The details to install OpenJDK are present in [18]. For our scenario, since we would be building a custom version of the OpenJDK with MultiMap implementation in it, we need to build the package from the source code to use it in our projects [19].

5.1 Hardware Requirements for Building OpenJDK

Suggested hardware requirements are mentioned below –

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Minimum number of cores</th>
<th>Minimum RAM</th>
<th>Minimum free disk space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Building on x86</td>
<td>2 – 4 cores</td>
<td>2 - 4 GB</td>
<td>6 GB</td>
</tr>
<tr>
<td>Building on aarch64</td>
<td>8 cores</td>
<td>8 GB</td>
<td>6 GB</td>
</tr>
</tbody>
</table>

Table 13: Hardware Requirements for OpenJDK

5.2 Operating System Requirements for Building OpenJDK

OpenJDK supports all the main operating systems including Linux, macOS, AIX, and Windows. Below are the OS versions which are suggested and used by Oracle to build the OpenJDK –
<table>
<thead>
<tr>
<th>Operating System</th>
<th>Version Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>Oracle Enterprise Linux 6.4 / 7.6</td>
</tr>
<tr>
<td>macOS</td>
<td>Mac OS X 10.13 (High Sierra)</td>
</tr>
<tr>
<td>Windows</td>
<td>Windows Server 2012 R2</td>
</tr>
</tbody>
</table>

Table 14: Operating System Requirements for OpenJDK

5.3 Toolchains Requirement for Building OpenJDK

OpenJDK consists of native code respective to the target platform for which it is to be built. To compile this native code for the platform we are required to have some toolchains for the respective platforms. These toolchain requirements are mentioned below –

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Supported Toolchain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>gcc, clang</td>
</tr>
<tr>
<td>macOS</td>
<td>Apple Xcode (using clang)</td>
</tr>
<tr>
<td>Windows</td>
<td>Microsoft Visual Studio</td>
</tr>
</tbody>
</table>

Table 15: Toolchain Requirements for OpenJDK

5.4 Boot JDK Requirements

Additionally, we need a pre-existing JDK to build our new source code for OpenJDK. This JDK is called “boot JDK” since it is required to build a new version of OpenJDK from the source code. The ideal rule to make sure that the source code is compiled and built is to have a major smaller version installed compared to the version you are trying to build. For example, if we want to build JDK 9 we would want JDK 8 to be installed on our system. Further instructions can be found on the official page [19].
Once we have satisfied the hardware, operating system, and toolchain requirements we can use the steps mentioned below to build the OpenJDK.

1. Get the source code for OpenJDK with MultiMap

   `git clone https://git.openjdk.org/jdk/`

2. Run configure

   `bash configure`

   This step can fail because of missing dependencies. Follow the suggestion prompts or refer to the complete installation instructions.

   ![Bash Configure Command Output - Configuration Details](image-url)
3. Run make

make images

4. Verify the newly build custom JDK

```
./build/*/images/jdk/bin/java -version
```

This directory would contain the newly build OpenJDK image with MultiMap support.

![Figure 20: Make Images and Version Commands Output](image)

5. Run basic tests

```
make run-test-tier1
```

The final step is to point to the newly built image of our OpenJDK. This can be done by updating the PATH variable for JAVA_HOME in respective operating systems. For our setup, we have updated the JAVA_HOME variable in the “.zshrc” file in the home directory to point to the new image.
Finally, we can verify by executing the command “java --version” that our operating system is using the newly built OpenJDK.
VII. CONCLUSION

A map is a data structure that uses key-value pairs to store data and is implemented in almost all programming languages. It provides efficient storage and retrieval of data based on the key against which data is stored. Hence it is used in different tasks such as indexing, caching, and dictionary. Maps also allow different data types to be used as keys and values and provide different operations such as adding, removing, and updating key-value pairs. The map implementation in Java only allows the storage of a single value against a given key, hence there was a need to implement a multimap.

MultiMap implementation in OpenJDK will reduce developers' coding and debugging efforts by handling basic operations automatically. It will also help maintain clean code and decrease the number of lines of code significantly. This project provides an implementation of different types of MultiMap including ArrayListMultiMap, SetMultiMap, and TreeMultiMap having individual unique properties, and since these are directly integrated into the util library of OpenJDK it becomes very easy to use these. Also, integration in OpenJDK would provide access to a wide range of users, since OpenJDK is an open-source implementation of the Java SE platform guidelines and hence can be used by anyone.

To use the multimap implementation in OpenJDK, you can use the custom version of OpenJDK developed in this project. The installation process is similar to how you would install a normal version of OpenJDK. You have to put the OpenJDK build at the desired location and then update the JAVA_HOME path variable which points to the JDK.
Multimap implementation in OpenJDK will allow developers easy access to the data structure and would help in reducing developer's effort by providing an in-built implementation. Multimap would provide developers with an easy and efficient way to store one-to-many relationships, and there would be no need to develop this complex data structure. But based on the research and experiments there are some areas where further development can be done. These are mentioned below –

8.1 Improving the Performance of Complex Data Structures

In our experiments, we have noted that our implementation performs better when we use Integer data type for key and value. But the performance drops significantly when we use other data types such as String. Hence, we can work on improving the performance of multimap when we use data types such as String as key and value. One approach would be that we pre-define the maximum size of the String which can be stored in key and value.

8.2 Return Types for Values of All Implementations

In our current implementation, when we get the values associated with a specific key the multimap will return a Collection<V> which will have all the values for all four implementations. We can add support where instead of returning a generic collection, we would return an implementation-specific collection such as a List for ArrayListMultiMap. This would further reduce the developer's effort to typecast returned values in the required format.
8.3 A Key-Value Datastore with Support for MultiMap

All the key-value data stores which we have discussed in the background section only support storing a single value against a given key, hence only supporting map data structure. We can use our implementation of multimap to build a key-value data store that would support multimap along with a map as a data structure. Creating a data store would open up new use cases such as a Cache Library which can support one-to-many relationships.
REFERENCES


[16] What is Key-Value Store and Why is it Important?
    https://www.linkedin.com/pulse/what-key-value-store-why-important-ken-grohe/


[18] How to download and install prebuild OpenJDK packages
    https://openjdk.org/install/