CSCI 3901 : LAB -1 Basic Problem Solving

Team Members: Abhay Lohani (B00989230) , Shray Moza (B00987463)

List of items from the description that needed clarification/Elements of the problem statement that are not defined sufficiently :

- What type of data we are dealing with(are there any redundant data)
- What should be done if a map already has a key?
- In the event of a collision, how should several key-value pairs be arranged if every key produces the same hash value?
- Can a NULL key be used?
- Do we need to increase map size at any case? What is the expected usage scenario?
- What are the requirements for the key class in terms of overriding equals and hashCode?
- Was it giving answer for double key value?

Our decisions on the items that need clarification

- The code manages key-value pairings in which both the keys and values are generic types (K and V). This provides for a diverse set of data types for keys and values. Redundancy: There is no duplicate data in terms of keys. If a key already exists in the map, its value is modified rather than adding a duplicate entry. This ensures that each key is distinct in the map.
- When we add a key to a map using the put method, if that key already exists, the map updates the old value with the new one. This way, we won't have duplicate keys, and the map will always hold the latest value for each key.
- When two keys end up with the same hash value, the current implementation handles this by using a linear search within an ArrayList. All the key-value pairs are stored in one list, so if multiple keys have the same hash, they're just kept as separate Node objects in that list. If we use the put method and the key already exists, it updates the existing entry; if not, it adds a new one
- Yes Null key are being used

- The ArrayList in this implementation automatically resizes itself when needed. So, we don't have to worry about manually increasing the map size. It takes care of resizing on its own whenever the number of elements goes beyond its current capacity.
- To make sure everything works properly, the key class needs to override the equals method. This is crucial for correctly comparing keys. Without a proper equals method, the contains and get methods won't work as they should because they depend on equals to match keys.
- This implementation doesn't allow multiple values for the same key. If we add a key more than once, the put method just updates the existing value instead of adding a new entry. So, the map only keeps the latest value for each key, and any previous values get overwritten.

Evidence to substantiate our code is working:/Implementation is working

- We were able to implement all the methods "get", "put", "size", and "contains".
- In this Map implementation we tried to cover most the test cases and were able to pass those.

Assurance to proof that the code has some quality to it:

- Maintainability and Readability : our code has methods like put, get, contain, size that substantiate their own purpose.
- Efficiency and Correctness : the put method in the code has ability to update the value of a key if it pre exist rather than adding duplicate key pair.
- Testing : The main class include testing in main method
- Time complexity : All the methods described in classes simple loops leading to average time complexity O(n).

Difficulties that we encountered

- During the though process we have encountered difficulty when we were not able to do the methods in O(n) time complexity
- Designing the key-value storage from scratch was one of the difficult part.
- While implementing put method we had to go through hassle, because of the fact that put method first need to check if there already exist a key.
- Null handling can also be categorized under difficulties that we faced.

Dealing with Difficulties

- We optimized further the code so that it can be done in O(n) time complexity.
- We used the array list implementation to eradicate this difficulty.
- The implementation ensures that each key in the CustomMap is unique by updating the existing value if the key is already present. This prevents redundant entries and maintains a clean and manageable map.
- Null cases are being handled in this program.

What we did well in developing the implementation that could be use as an approach to implement another problem?

- **Enhancing Flexibility With Generics in the KeyValueStore Class**
  To ensure flexibility in handling various data types and promote reusability, we successfully incorporated generics into the KeyValueStore class. Generics allow us to write code that can work with different data types without sacrificing type safety. By reducing redundancy, generics enhance code maintainability and adaptability.

- **Modularity and Separation of Concerns**
  We improved testing and modularity by separating the key-value logic into its own class. This approach, especially valuable in UI development, allows for easier debugging and functional extension. By adhering to object-oriented principles, we achieve cleaner, more organized code.

- **Data Integrity through Encapsulation**
  We encapsulated the internal details of the Node class to safeguard data integrity. Encapsulation is akin to securing confidential bank data—it prevents misuse and facilitates future modifications while maintaining a clear boundary between internal implementation and external usage.

- **Ensuring System Integrity**
  The put and get methods effectively manage data, ensuring that duplicates are avoided. Such robust data management practices are essential for applications across various domains.