**CHC5223\_CW1\_Report**

Student\_ID:202118010418

Catalogue

[CHC5223\_CW1\_Report 1](#_Toc163419386)

[Student\_ID:202118010418 1](#_Toc163419387)

[1 Task\_1 2](#_Toc163419388)

[1.1 Code: 2](#_Toc163419389)

[1.2 Explanation: 4](#_Toc163419390)

[2 Task\_2 7](#_Toc163419391)

[2.1 Code 7](#_Toc163419392)

[2.2 Explanation 11](#_Toc163419393)

[3 Task\_3 19](#_Toc163419394)

[3.1 Code 19](#_Toc163419395)

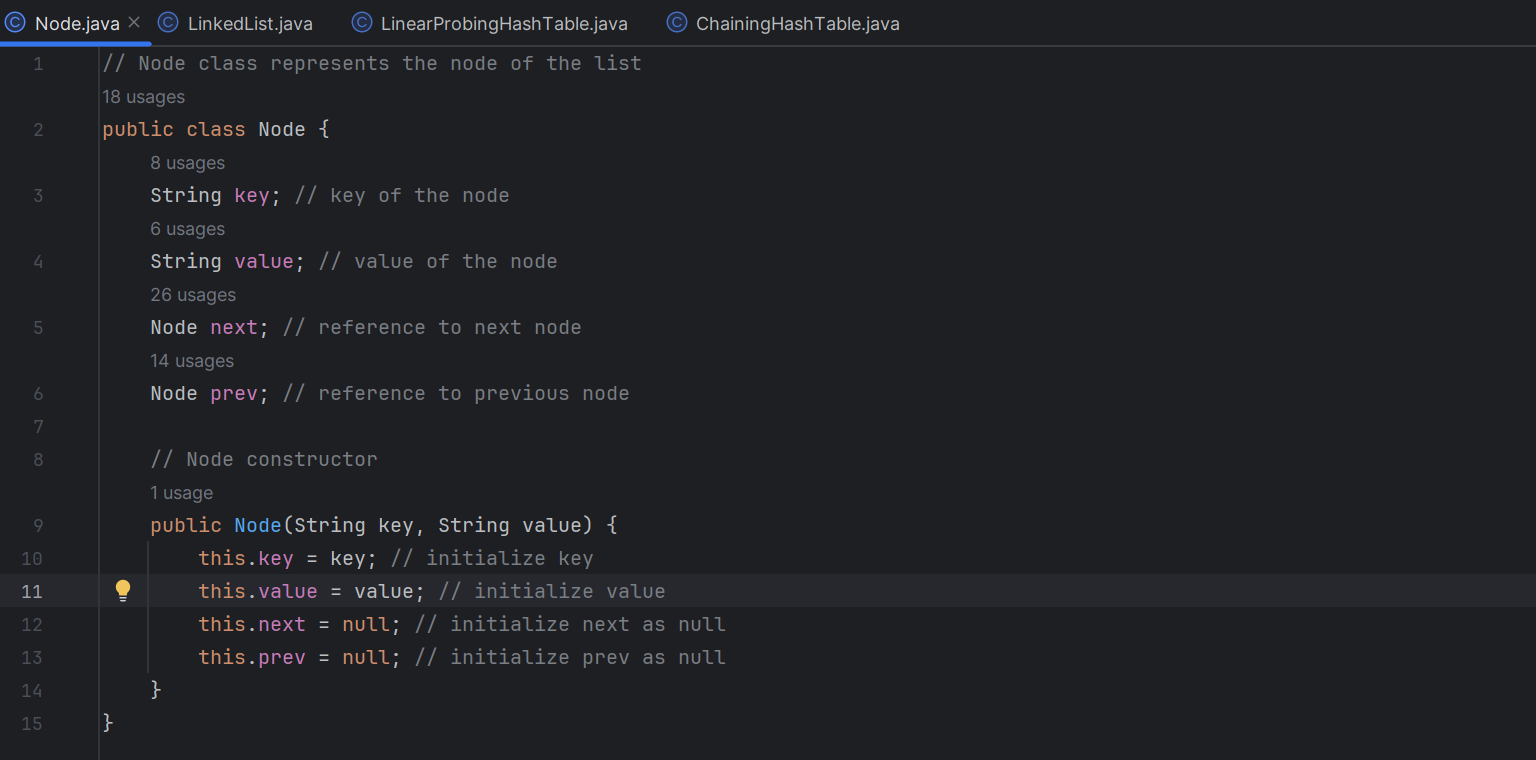
[3.2 Explanation 22](#_Toc163419396)

[4 Task\_4 30](#_Toc163419397)

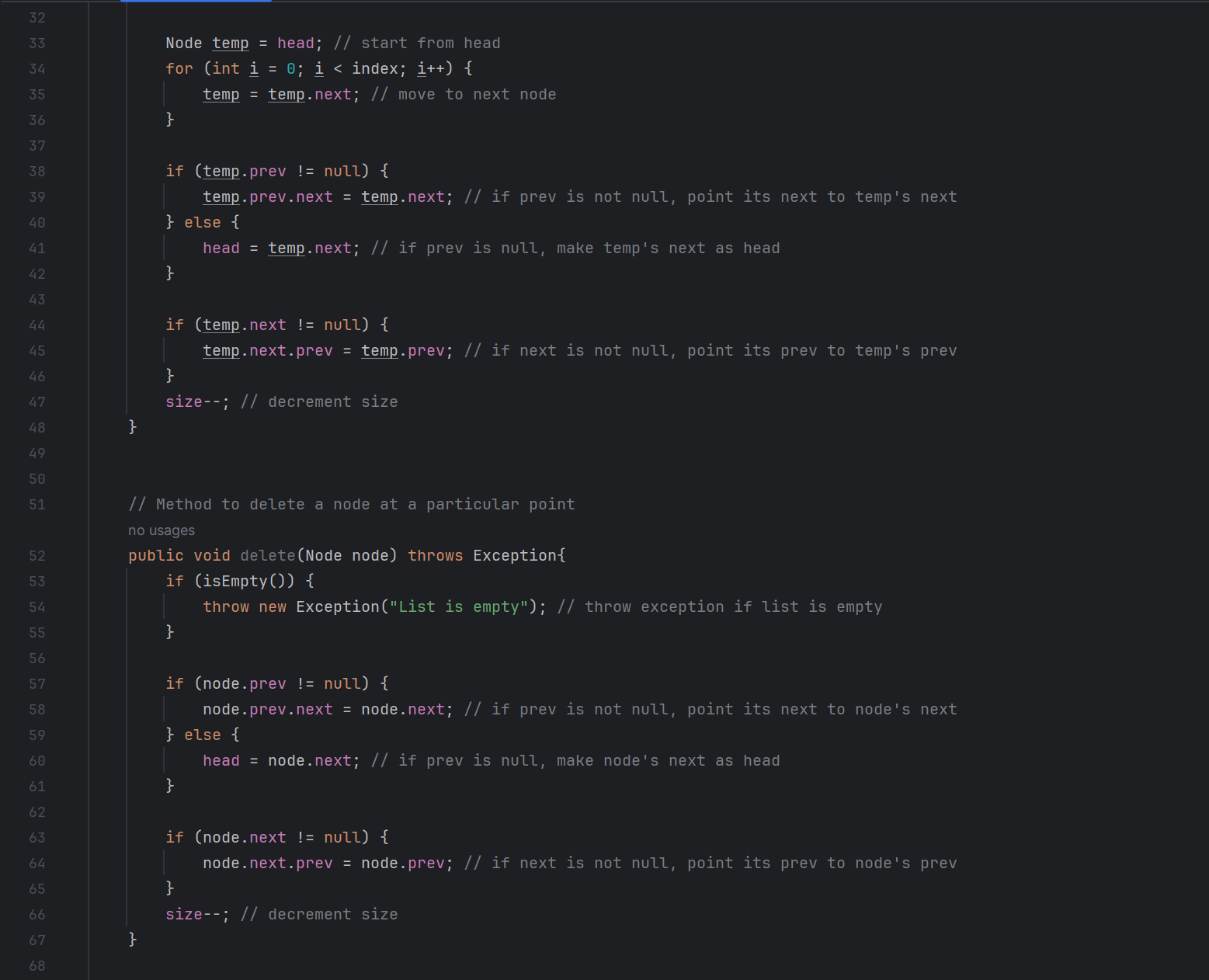
[4.1 Test Code 30](#_Toc163419398)

[4.2 Explanation 31](#_Toc163419399)

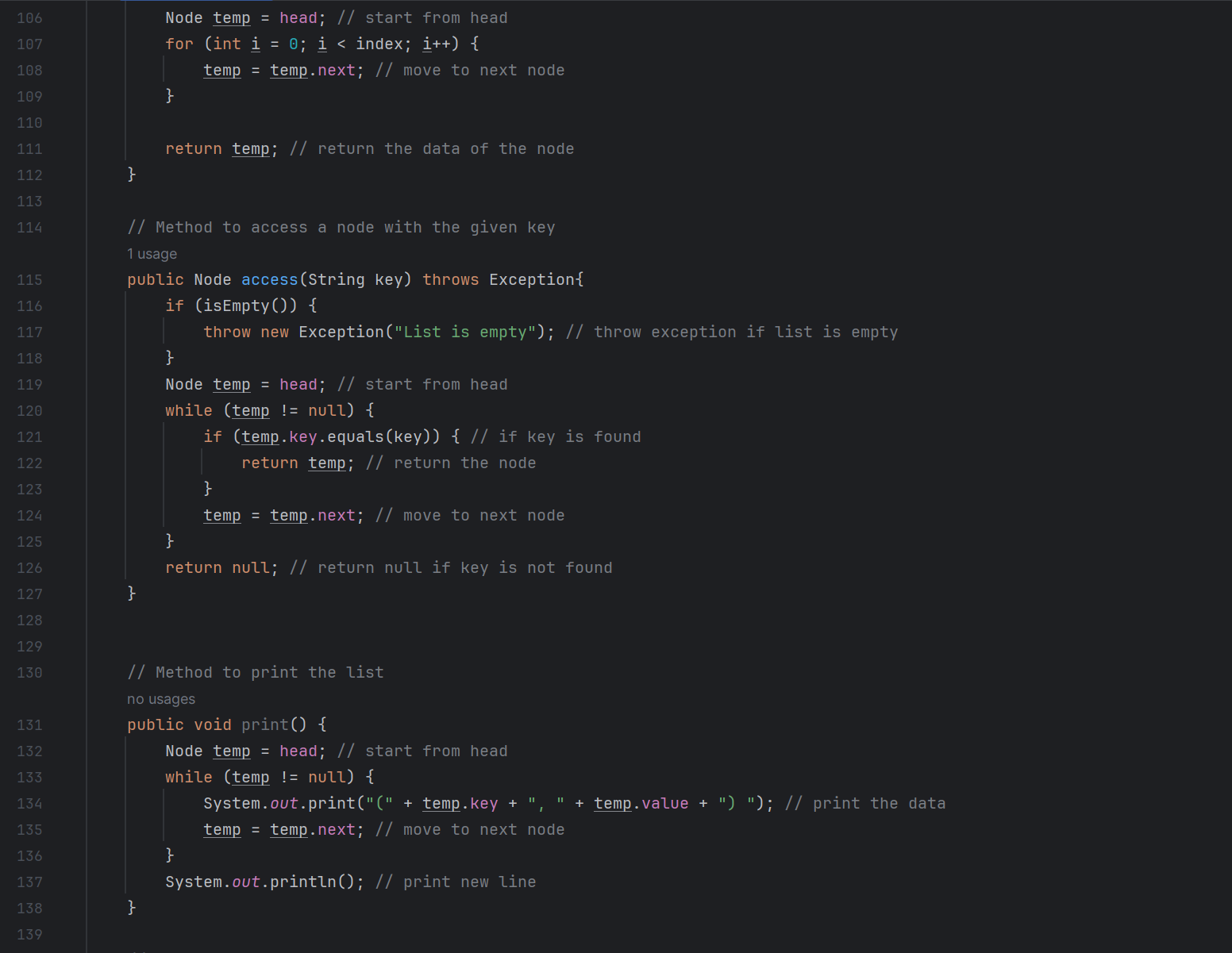
1. Task\_1
   1. Code:













* 1. Explanation:

The doubly linked list will be implemented using two classes: ***Node*** and ***LinkedList***.

The Node class will represent an individual node in the list. Each node will have 2 value of type String (represents the ***key*** and ***value*** of a key-value pair, because in task 3 the data is indicated as key-value pairs), and two pointers, ***next*** and ***prev***, pointing to the next and previous nodes in the list respectively.

The ***LinkedList*** class represents a doubly linked list. It contains the following methods:

* + - * LinkedList (): This is the constructor of the LinkedList class. It initializes the head of the list to null and the size of the list to 0.
      * insert (String key, String value): This method inserts a new node at the front of the list. It creates a new node with the given key and value, sets the next of the new node to the current head, and then updates the head to the new node. If the head was not null, it also sets the previous of the head to the new node. It then increments the size of the list.
      * delete (String key): This method deletes the first node with a given key from the list. It throws an exception if the list is empty. It traverses the list to find the node with the given key and then updates the next of the previous node and the previous of the next node to remove the current node from the list.
      * delete (int index): This method deletes a node at a given index. It throws an exception if the list is empty or if the index is out of bounds. It traverses the list to the node at the given index and then updates the next of the previous node and the previous of the next node to remove the current node from the list.
      * delete (Node node): This method deletes a given node from the list. It throws an exception if the list is empty. It updates the next of the previous node and the previous of the next node to remove the current node from the list.
      * access (int index): This method returns the node at a given index. It throws an exception if the list is empty or if the index is out of bounds. It traverses the list to the node at the given index and then returns that node.
      * access (String key): This method returns the first node with a given key. It throws an exception if the list is empty. It traverses the list to find the node with the given key and then returns that node.
      * isEmpty (): This method returns true if the list is empty (i.e., the size is 0) and false otherwise.
      * size (): This method returns the number of nodes in the list.
      * print (): This method prints the keys and values of all nodes in the list from the head to the tail.
      * clear (): This method removes all nodes from the list by setting the head to null and the size to 0.

1. ***Question: The double-linked list should be a generic data structure that can store elements of string data type.***

Answer:

To create a general-purpose bidirectional linked list that can store string data types, I did the following:

First, I create a generic class called Node, which will represent the nodes in the linked list. Each Node has two key, value string fields to store the data (usually it should be a data field, but considering the data to be accessed later is a key-value pair, so I changed it accordingly), and two node type fields next and prev, pointing to the next and previous nodes in the linked list, respectively.

Then, I create a class called DoublyLinkedList, which will implement the main functions of a bidirectional linked list, such as adding elements, deleting elements, and traversing the list.

1. ***Question: You must create a Node class that represents each element in the doubled-linked list.***

Answer: Refer to Answer 1

1. ***Question: You must create a LinkedList class that represents a doubly linked list which should include methods for inserting, deleting, accessing specific elements, checking empty, returning size, and other operations you want to implement.***

Answer: It has been implemented in code

1. ***Question: The insertion operation should be done at the front of the list.***

Answer:

// Method to insert a new node at the front of the list

public void insert(String key, String value) {

Node newNode = new Node(key, value); // create a new node

newNode.next = head; // point new node's next to head

if (head != null) {

head.prev = newNode; // if head is not null, point head's prev to new node

}

head = newNode; // make new node as head

size++; // increment size

}

This method creates a new node with the supplied keys and values, points the new node next to the current header of the list, and then makes the new node the new header of the list. If the list is not empty, it also points the lead of the old header to the new node. Finally, it increases the size of the list.

1. ***Question: The implementation should include error handling to handle errors such as deleting elements from an empty list and accessing out-of-bounds.***

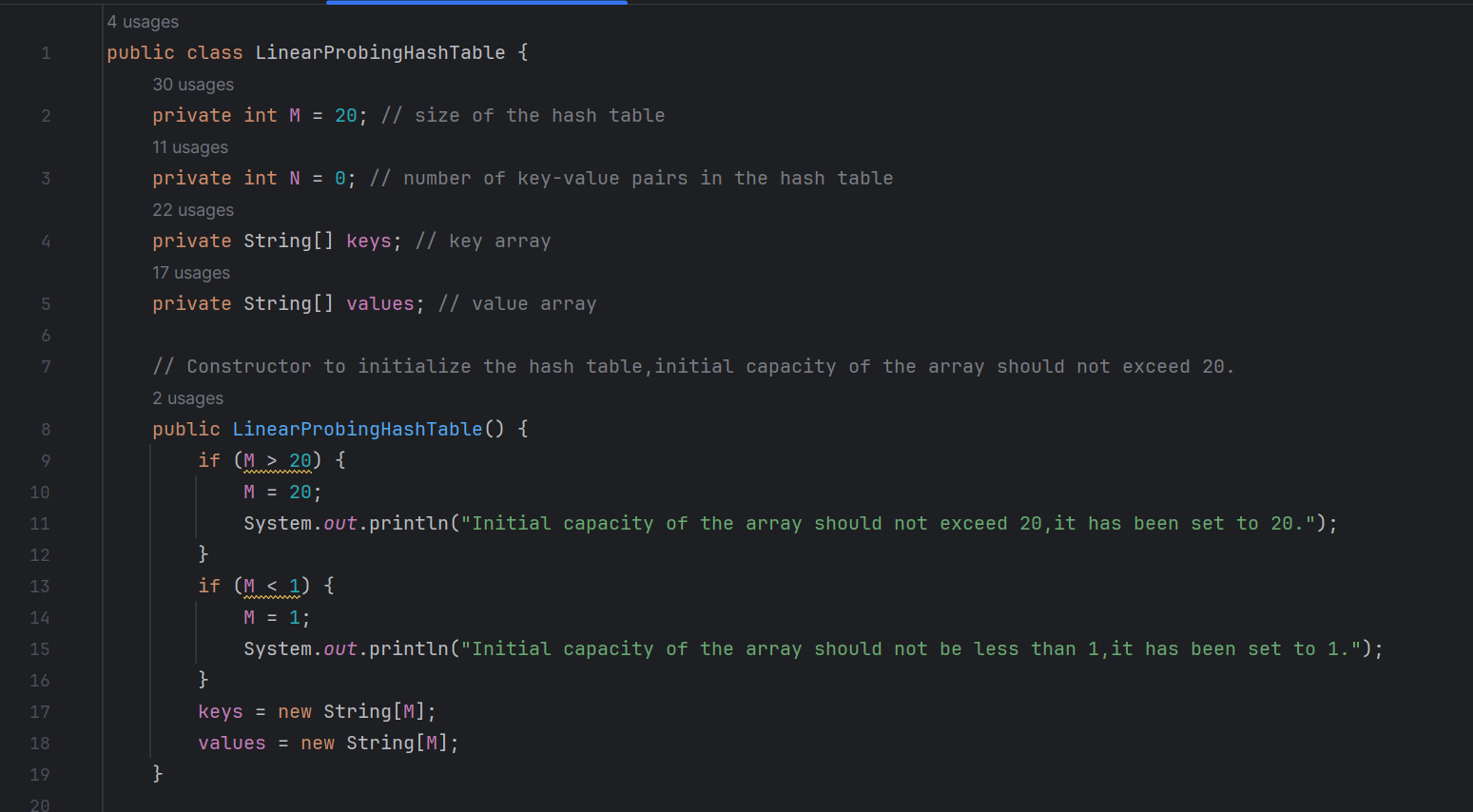
***Answer:***

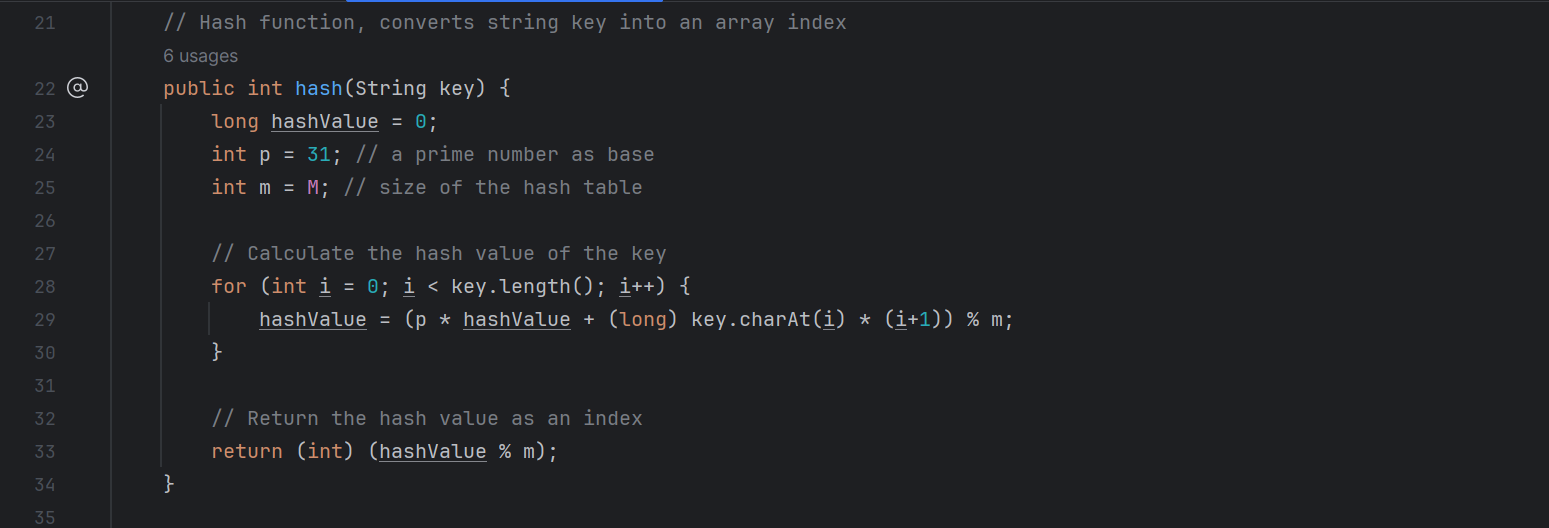
In the delete(int index), delete(Node node), and delete(String key) methods, there are checks to see if the list is empty before proceeding with the deletion. If the list is empty, an exception is thrown with the message "List is empty".

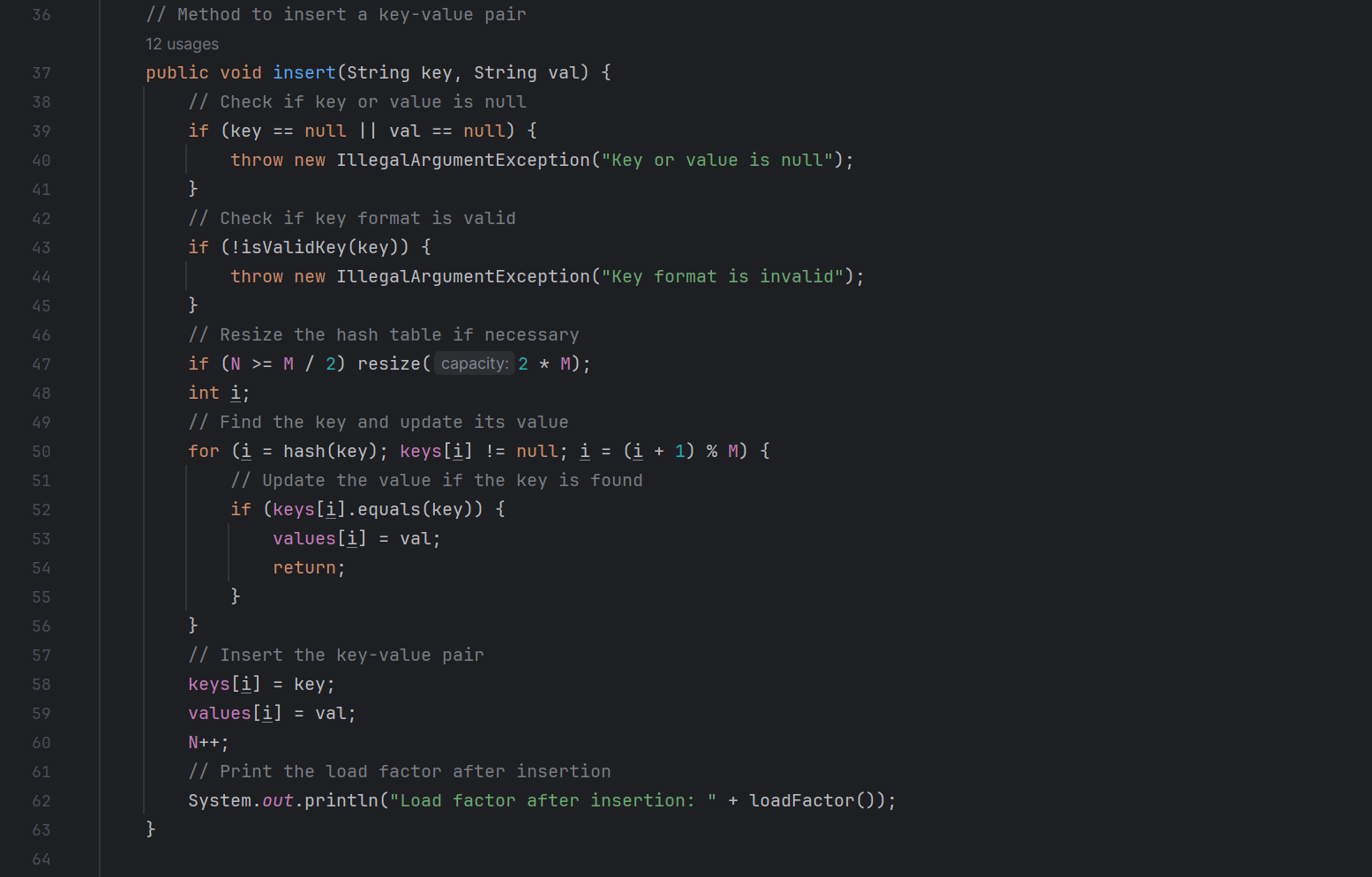
In the delete(int index) and access(int index) methods, there are checks to see if the provided index is out of bounds (less than 0 or greater than or equal to the size of the list). If the index is out of bounds, an exception is thrown with the message "Index out of bounds".

In the delete(String key) method, if the key is not found in the list, an exception is thrown with the message "Element not found".

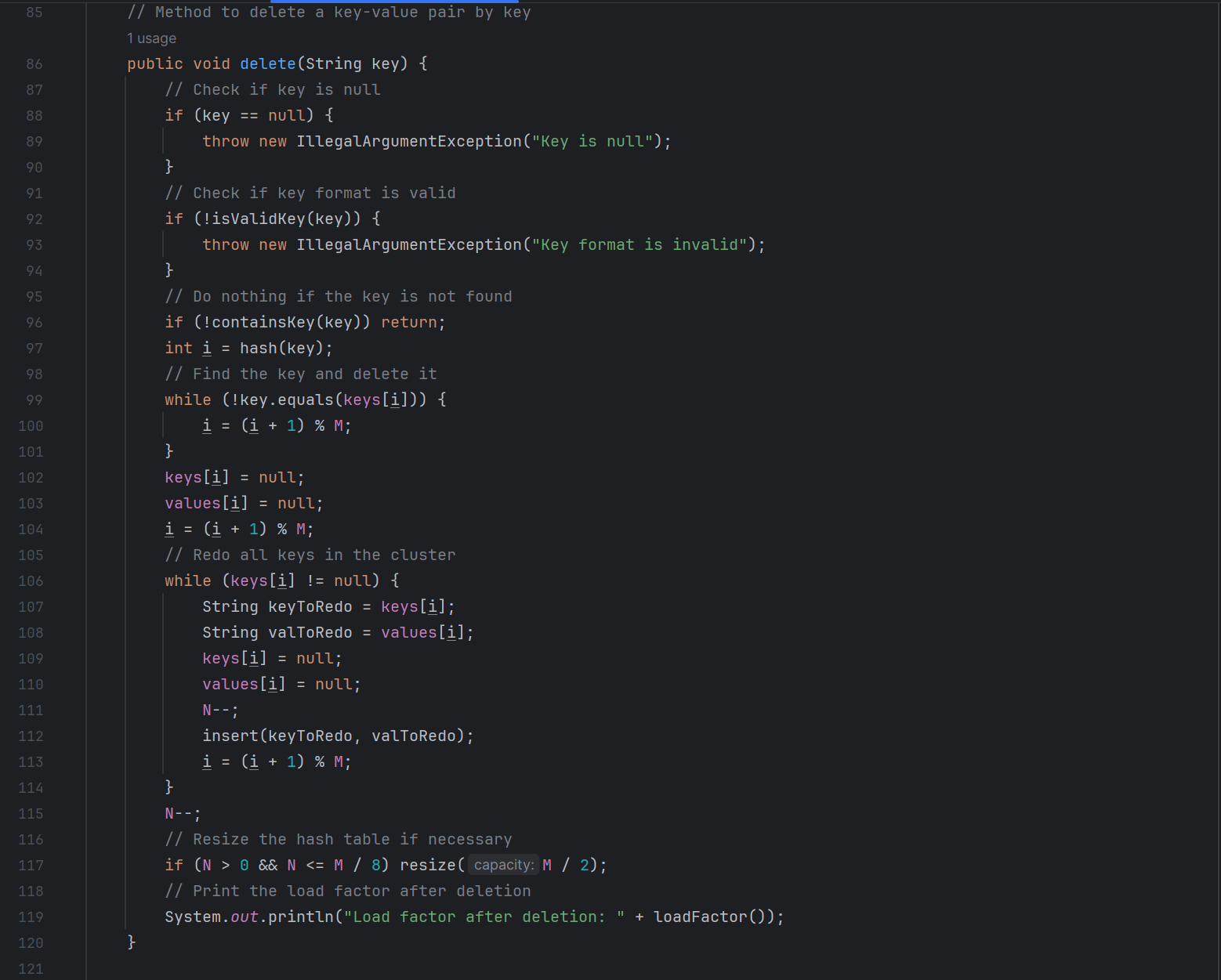
1. Task\_2
   1. Code

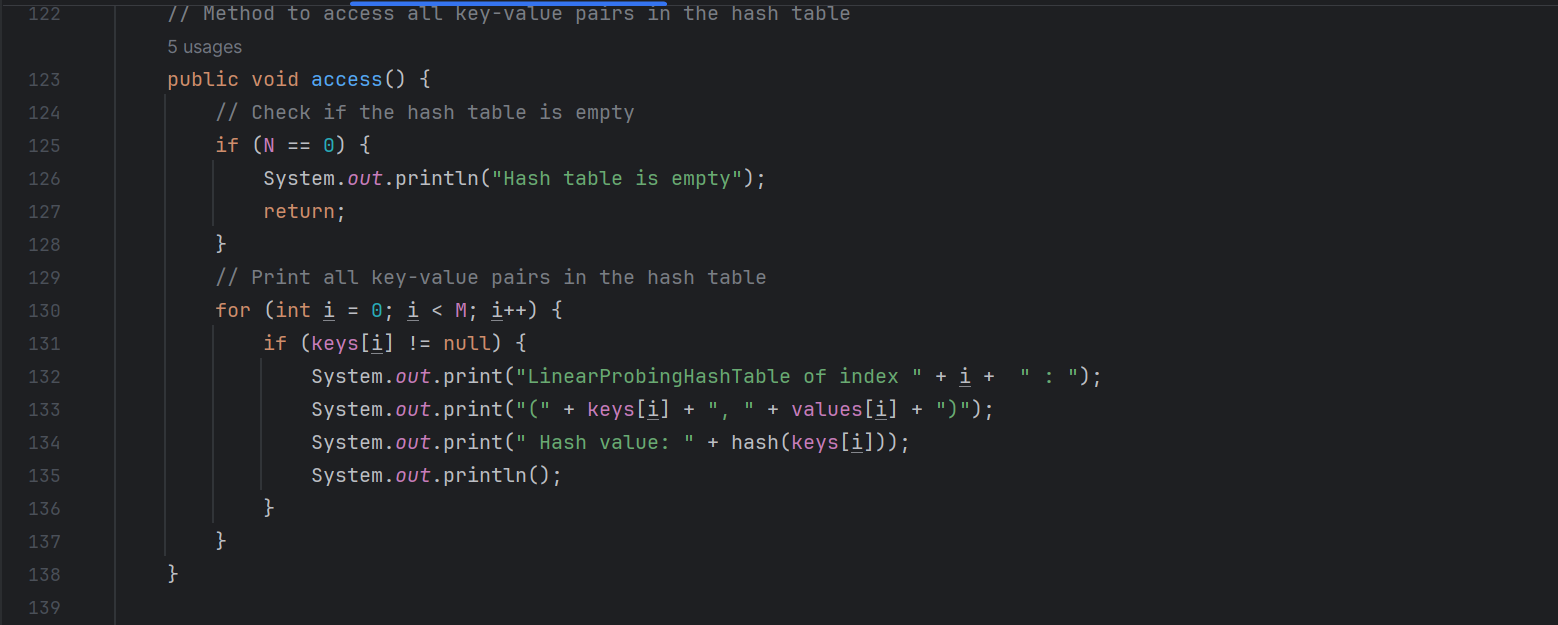




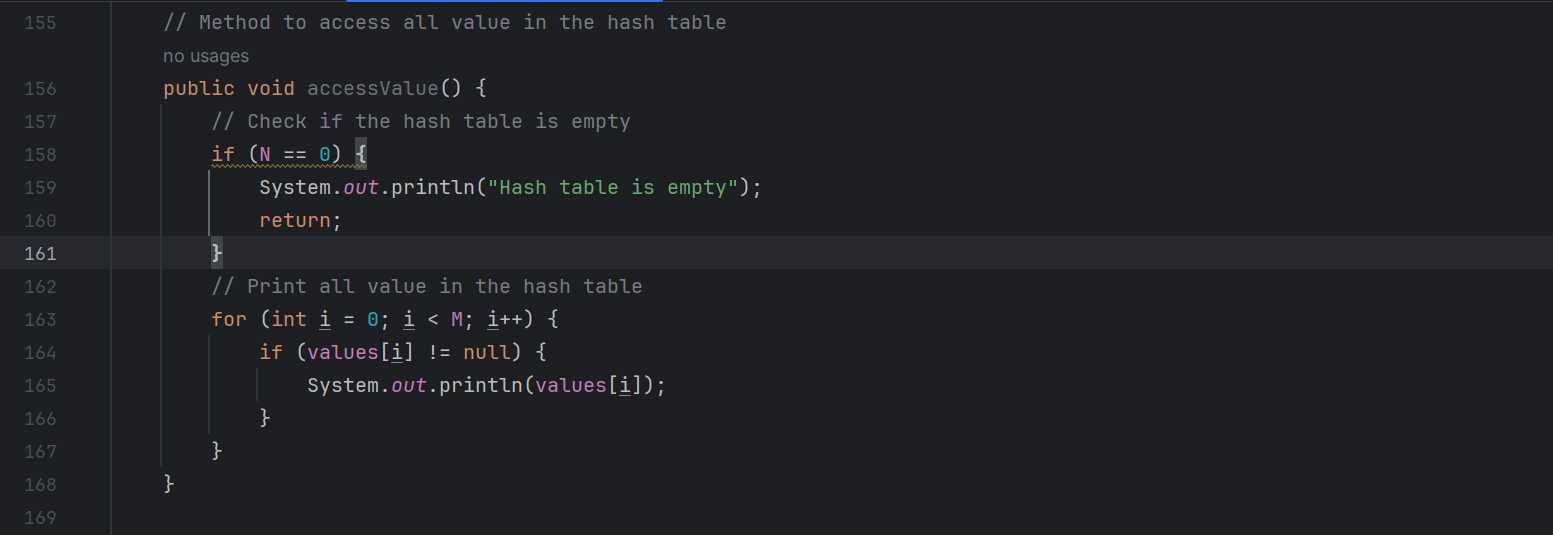


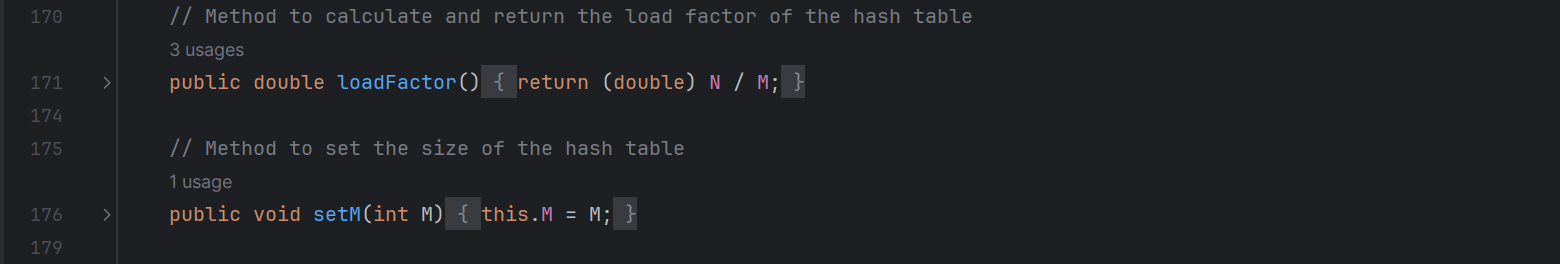




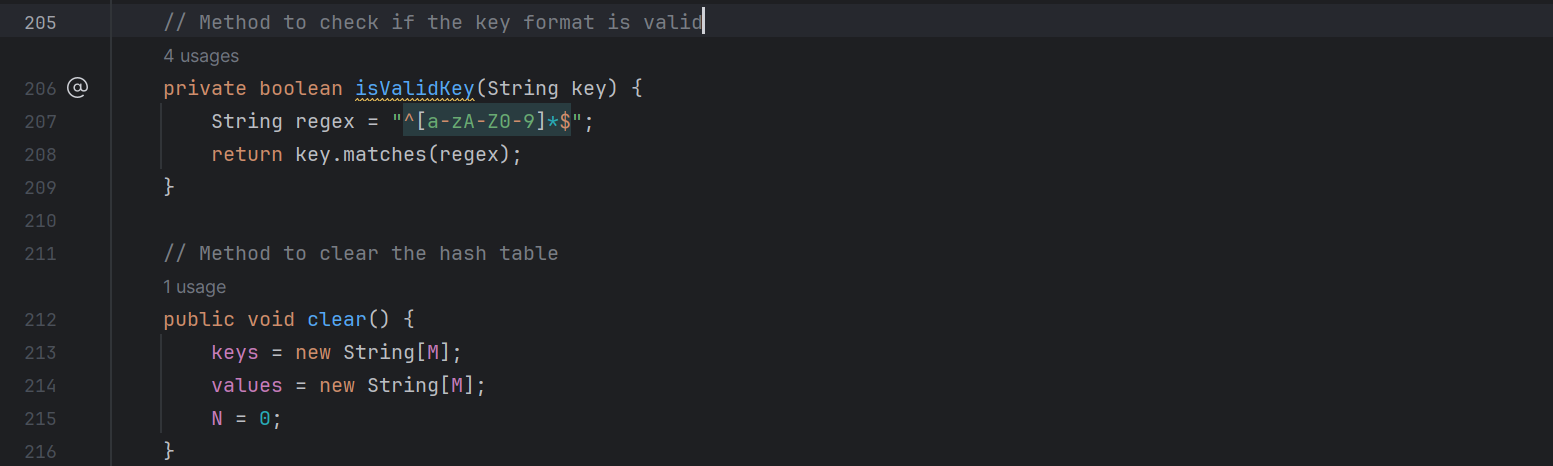














* 1. Explanation

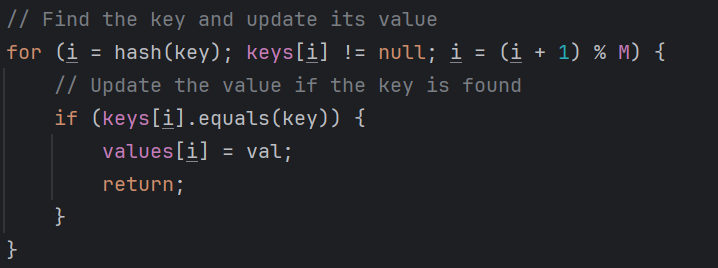
The ***LinearProbingHashTable*** class represents a hash table that uses linear probing to resolve collisions. It contains the following methods:

* LinearProbingHashTable (): This is the constructor of the LinearProbingHashTable class. It initializes the size of the hash table M to 20, the number of key-value pairs N to 0, and creates the keys and values arrays. If the initial capacity of the array exceeds 20, it will be set to 20. If it is less than 1, it will be set to 1.
* hash (String key): This method calculates and returns the hash value of a given key. It uses a prime number as the base and the size of the hash table as the modulus. The hash value is calculated by iterating over each character in the key, multiplying the current hash value by the base, adding the ASCII value of the character, and then taking the modulus of the size of the hash table.
* insert (String key, String val): This method inserts a key-value pair into the hash table. It first checks if the key or value is null or if the key format is invalid. If the load factor of the hash table is greater than or equal to 0.5, it resizes the hash table to twice its current size. It then calculates the hash value of the key and finds the first available slot in the hash table by linear probing. If the key is already in the hash table, it updates its value. Otherwise, it inserts the key-value pair and increments N.
* search (String key): This method returns the value associated with a given key. It first checks if the key is null or if the key format is invalid. It then calculates the hash value of the key and finds the key in the hash table by linear probing. If the key is found, it returns its associated value. Otherwise, it returns null.
* delete (String key): This method deletes a key-value pair by key. It first checks if the key is null or if the key format is invalid. It then calculates the hash value of the key and finds the key in the hash table by linear probing. If the key is found, it deletes the key-value pair and rehashes all keys in the same cluster. If the load factor of the hash table is less than or equal to 0.125, it resizes the hash table to half its current size.
* access (): This method prints all key-value pairs in the hash table. It iterates over the keys and values arrays and prints each key-value pair along with its hash value.
* accessKey (): This method prints all keys in the hash table. It iterates over the keys array and prints each key.
* accessValue (): This method prints all values in the hash table. It iterates over the values array and prints each value.
* loadFactor (): This method calculates and returns the load factor of the hash table, which is the ratio of the number of key-value pairs N to the size of the hash table M.
* setM (int M): This method sets the size of the hash table to a given value.
* getM (): This method returns the size of the hash table.
* resize (int capacity): This method resizes the hash table to a given capacity. It creates a new LinearProbingHashTable with the given capacity, rehashes all keys in the old hash table into the new hash table, and then switches to the new hash table.
* isValidKey (String key): This method checks if the key format is valid. It uses a regular expression to check if the key contains only alphanumeric characters.
* clear (): This method clears the hash table by creating new keys and values arrays and setting N to 0.
* containsKey (String key): This method checks if the hash table contains a specific key. It calculates the hash value of the key and finds the key in the hash table by linear probing. If the key is found, it returns true. Otherwise, it returns false.
* containsValue (String value): This method checks if the hash table contains a specific value. It iterates over the values array and checks if the value is found. If the value is found, it returns true. Otherwise, it returns false.

1. ***Question:*** ***You must create a LinearProbingHashTable class that represents a hash table by using the linear probing way for collision resolution. The initial capacity of the array should not exceed 20.***

***Answer:***

LinearProbingHashTable(): This constructor initializes the hash table with a maximum initial capacity of 20. If the initial capacity exceeds 20, it is set to 20. If it is less than 1, it is set to 1.



In this code,for (i = hash(key); keys[i] ! = null; This line of code i = (i + 1) % M) is the key to implementing linear probing. It first calculates the hash value of the key and then checks if the location is already occupied. If it is already occupied, it tries the next location (i = (i + 1) % M) until it finds an empty location.

1. ***Question:*** ***You must devise a hash function that can work well for string-type data. The hash function devised should minimize the occurrence of collisions. You must not use the Java built-in hashCode method, though you can experiment with it.***

***Answer:***

// Hash function, converts string key into an array index  
public int hash(String key) {  
 long hashValue = 0;  
 int p = 31; // a prime number as base  
 int m = M; // size of the hash table  
  
 // Calculate the hash value of the key  
 for (int i = 0; i < key.length(); i++) {  
 hashValue = (p \* hashValue + (long) key.charAt(i) \* (i+1)) % m;  
 }  
  
 // Return the hash value as an index  
 return (int) (hashValue % m);  
}

In order to make hash functions work well, my design is based on the following principles:

1. Use a prime number (in this case, 31) as the base number. This is a common practice because prime numbers can help reduce hash conflicts.

2. Use modular arithmetic to ensure that the hash value is within the size range of the hash table. This is necessary because we need to use the hash value as an array index.

3. For each character in the key, multiply the current hash value by the base, then add the ASCII value of the character by its position (i+1), and then take the size of the modular hash table. This ensures that different characters and character positions affect the final hash value.

This hash function distributes the keys evenly across the hash table, thus minimizing the occurrence of conflicts.

1. ***Question:*** ***The implementation can handle errors such as null keys or keys with unexpected formats.***

***Answer:***

In the insert, search, delete, and containsKey methods, there are checks to see if the key is null or if the key format is valid. If the key is null or the format is invalid, an IllegalArgumentException is thrown with the appropriate message.

Here are the relevant parts of the code:

public void insert(String key, String val) {  
 // Check if key or value is null  
 if (key == null || val == null) {  
 throw new IllegalArgumentException("Key or value is null");  
 }  
 // Check if key format is valid  
 if (!isValidKey(key)) {  
 throw new IllegalArgumentException("Key format is invalid");  
 }  
 // ... rest of the code ...  
}  
  
// Method to get the value associated with a key  
public String search(String key) {  
 // Check if key is null  
 if (key == null) {  
 throw new IllegalArgumentException("Key is null");  
 }  
 // Check if key format is valid  
 if (!isValidKey(key)) {  
 throw new IllegalArgumentException("Key format is invalid");  
 }  
 // ... rest of the code ...  
}  
  
// Method to delete a key-value pair by key  
public void delete(String key) {  
 // Check if key is null  
 if (key == null) {  
 throw new IllegalArgumentException("Key is null");  
 }  
 // Check if key format is valid  
 if (!isValidKey(key)) {  
 throw new IllegalArgumentException("Key format is invalid");  
 }  
 // ... rest of the code ...  
}

1. ***Question:*** ***The implementation should include methods for inserting, searching, deleting, and accessing key-value pairs.***

***Answer:***

Code has implemented above. There some introduction:

insert(String key, String val): This method inserts a key-value pair into the hash table. It checks if the key or value is null and if the key format is valid. If the load factor (the ratio of the number of key-value pairs to the size of the hash table) is greater than or equal to 0.5, it resizes the hash table to twice its current size. It then finds an empty slot for the key-value pair using linear probing.

search(String key): This method returns the value associated with a given key. It checks if the key is null and if the key format is valid. It then finds the key using linear probing and returns its associated value.

delete(String key): This method deletes a key-value pair from the hash table. It checks if the key is null and if the key format is valid. It then finds the key using linear probing and deletes it. It also rehashes all keys in the same cluster. If the load factor is less than or equal to 0.125, it resizes the hash table to half its current size.

access(), accessKey(), accessValue(): These methods print all key-value pairs, keys, and values in the hash table, respectively.

These methods ensure that the hash table can perform the basic operations of inserting, searching, deleting, and accessing key-value pairs.

1. ***Question:*** ***The implementation of the inserting operation can resize the table efficiently according to the strategy you design if the hash table is too full.***

***Answer:***

The insert method in the LinearProbingHashTable class in my project includes a resizing operation that is triggered when the load factor (the ratio of the number of key-value pairs to the size of the hash table) is greater than or equal to 0.5.

Here's the relevant part of the insert method:

***// Method to insert a key-value pair***

***public void insert(String key, String val) {***

***// Check if key or value is null***

***if (key == null || val == null) {***

***throw new IllegalArgumentException("Key or value is null");***

***}***

***// Check if key format is valid***

***if (!isValidKey(key)) {***

***throw new IllegalArgumentException("Key format is invalid");***

***}***

***// Resize the hash table if necessary***

***if (N >= M / 2) resize(2 \* M);***

***// ... rest of the code ...***

***}***

And here's the resize method that is called:

// Method to resize the hash table

private void resize(int capacity) {

System.out.println("Because the load factor is too high, the hash table has been resized to " + capacity);

LinearProbingHashTable t = new LinearProbingHashTable();

t.M = capacity;

t.keys = new String[t.M];

t.values = new String[t.M];

// Rehash all keys in the old array

for (int i = 0; i < M; i++) {

if (keys[i] != null) {

t.insert(keys[i], values[i]);

}

}

// Switch to the new arrays

keys = t.keys;

values = t.values;

M = t.M;

System.out.println("Reload completed!");

}

When the load factor is too high, the hash table is resized to twice its current size. A new hash table is created with the new capacity, and all the key-value pairs in the old hash table are rehashed and inserted into the new hash table. The old arrays are then replaced with the new ones. This strategy ensures that the hash table remains efficient as it grows.

1. ***Question:*** ***The implementation of the deleting operation can handle the situation when the key is not found.***

***Answer:***

In my delete method, if no key is found in the hash table, the method simply returns without doing anything. This is done by using the containsKey method to first check if the key exists in the hash table before proceeding with the deletion.

Here's the relevant part of the delete method:

// Do nothing if the key is not found  
if (!containsKey(key)) return;

Here is the containsKey method:

// Method to check if the hash table contains a specific key  
public boolean containsKey(String key) {  
 // Check if key is null  
 if (key == null) {  
 throw new IllegalArgumentException("Key is null");  
 }  
 // Check if key format is valid  
 if (!isValidKey(key)) {  
 throw new IllegalArgumentException("Key format is invalid");  
 }  
 // Return true if key is found  
 return search(key) != null;  
}

It first checks if the provided key is null. If it is, an IllegalArgumentException is thrown with the message "Key is null".

Next, it checks if the key format is valid by calling the isValidKey method. If the key format is not valid, an IllegalArgumentException is thrown with the message "Key format is invalid".

Finally, it calls the search method with the key as the argument. The search method returns the value associated with the key if it exists in the hash table, and null otherwise. Therefore, if the search method returns a non-null value, it means the key exists in the hash table, and the containsKey method returns true. If the search method returns null, it means the key does not exist in the hash table, and the containsKey method returns false.

1. ***Question:*** ***The implementation can keep track of the load factor of the hash table and display it after each insertion or deletion.***

***Answer:***

Yes, the `LinearProbingHashTable` class in my project already includes a method `loadFactor()` that calculates and returns the load factor of the hash table. The load factor is calculated as the ratio of the number of key-value pairs (`N`) to the size of the hash table (`M`).

In the `insert` and `delete` methods, after a key-value pair is inserted or deleted, the load factor is printed to the console. Here are the relevant parts of the `insert` and `delete` methods:

// Method to insert a key-value pair

public void insert(String key, String val) {

// ... rest of the code ...

// Print the load factor after insertion

System.out.println("Load factor after insertion: " + loadFactor());

}

// Method to delete a key-value pair by key

public void delete(String key) {

// ... rest of the code ...

// Print the load factor after deletion

System.out.println("Load factor after deletion: " + loadFactor());

}

This implementation ensures that the load factor of the hash table is tracked and displayed after each insertion or deletion, which can be useful for monitoring the performance of the hash table.

1. ***Question:*** ***The implementation of the searching operation can search for the key and return the corresponding value if the key is found.***

***Answer:***

Yes, the `search` method in the `LinearProbingHashTable` class in my project is designed to search for a key and return the corresponding value if the key is found.

Here's how it works:

```java

// Method to get the value associated with a key

public String search(String key) {

// Check if key is null

if (key == null) {

throw new IllegalArgumentException("Key is null");

}

// Check if key format is valid

if (!isValidKey(key)) {

throw new IllegalArgumentException("Key format is invalid");

}

// Find the key and return its associated value

for (int i = hash(key); keys[i] != null; i = (i + 1) % M) {

if (keys[i].equals(key)) {

return values[i];

}

}

// Return null if key is not found

return null;

}

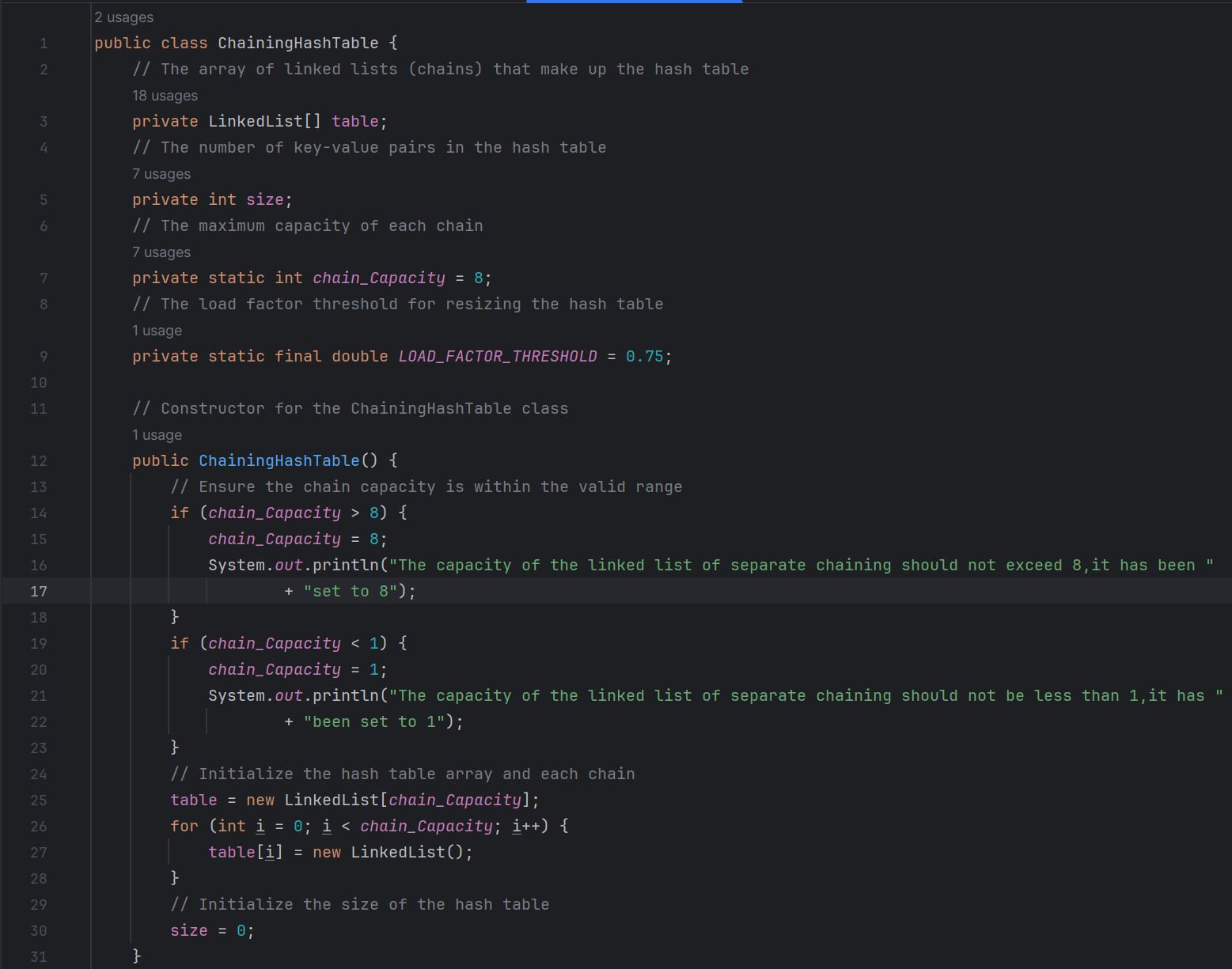
```

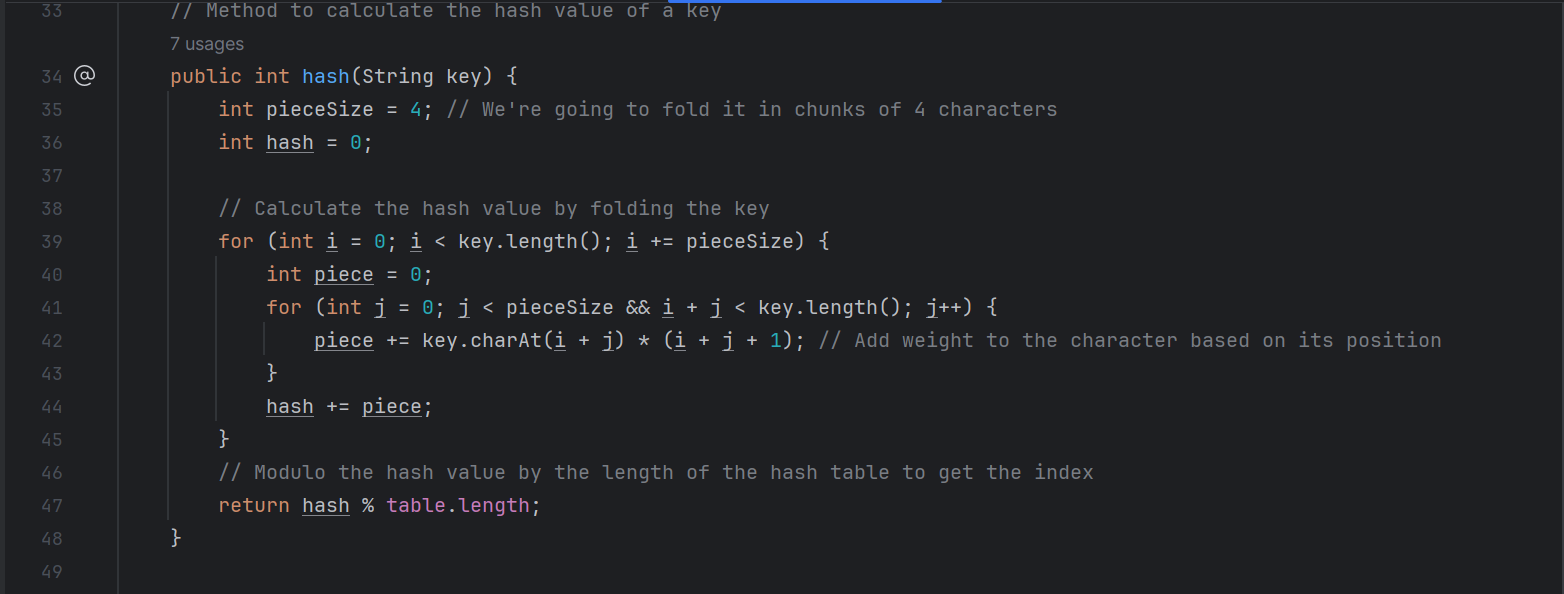
This method first checks if the provided key is null or if its format is invalid. If either of these conditions is true, it throws an `IllegalArgumentException`.

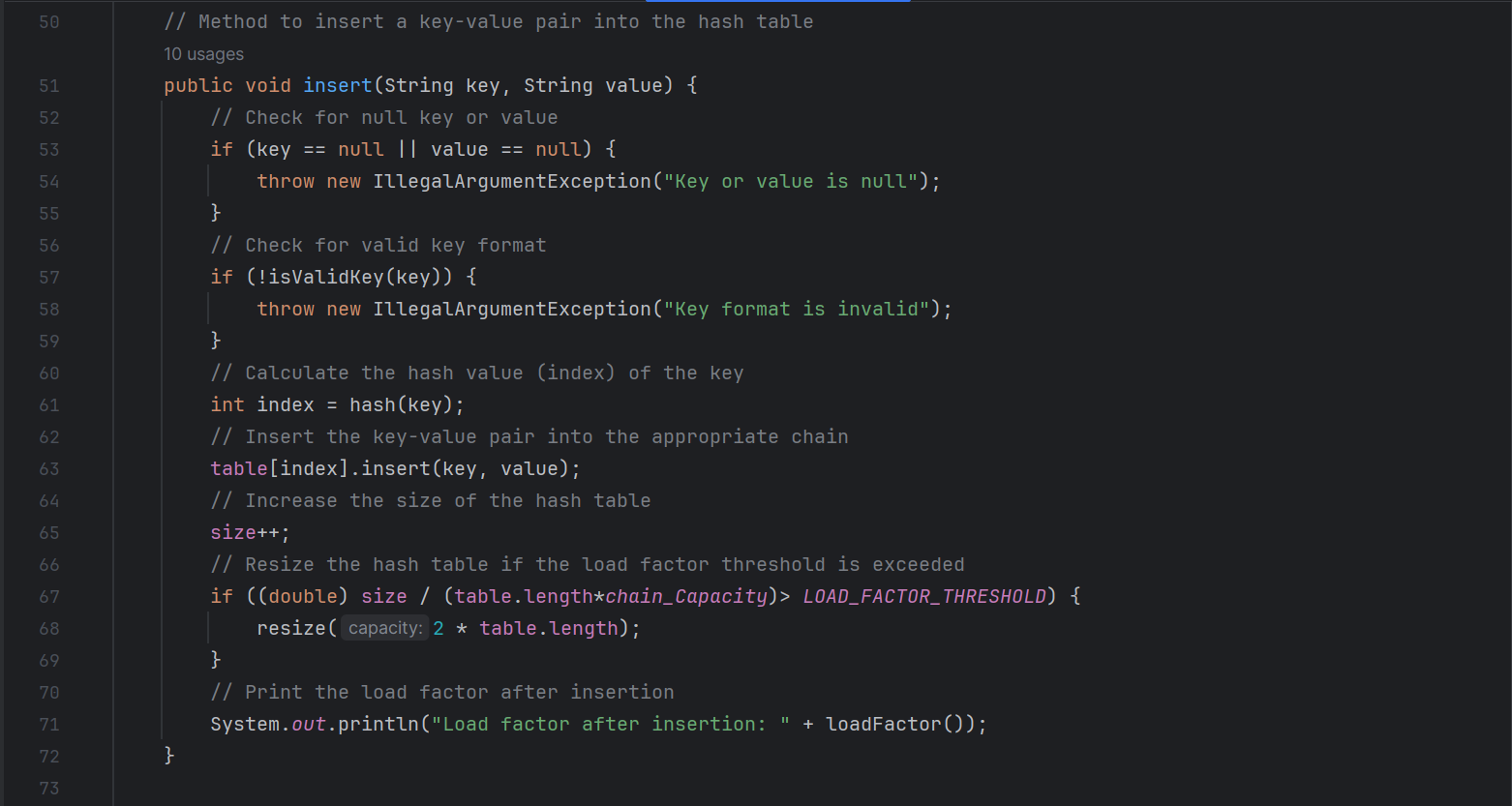
Next, it calculates the hash of the key to find the index where the key should be located in the hash table. It then iterates through the keys in the hash table starting from this index. If it finds a key that matches the provided key, it returns the corresponding value.

If the method iterates through all the keys in the hash table without finding a match, it returns null. This indicates that the key is not found in the hash table.

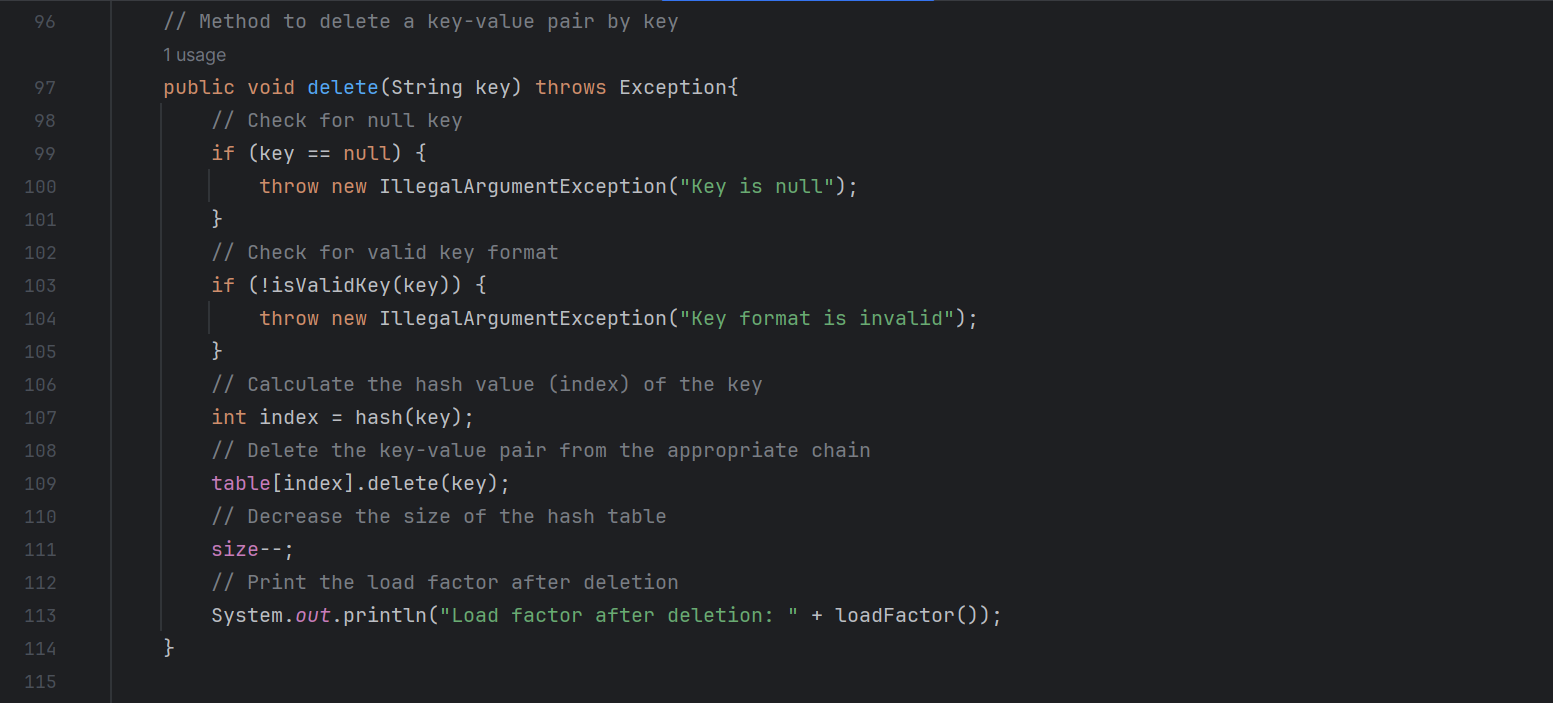
1. Task\_3
   1. Code

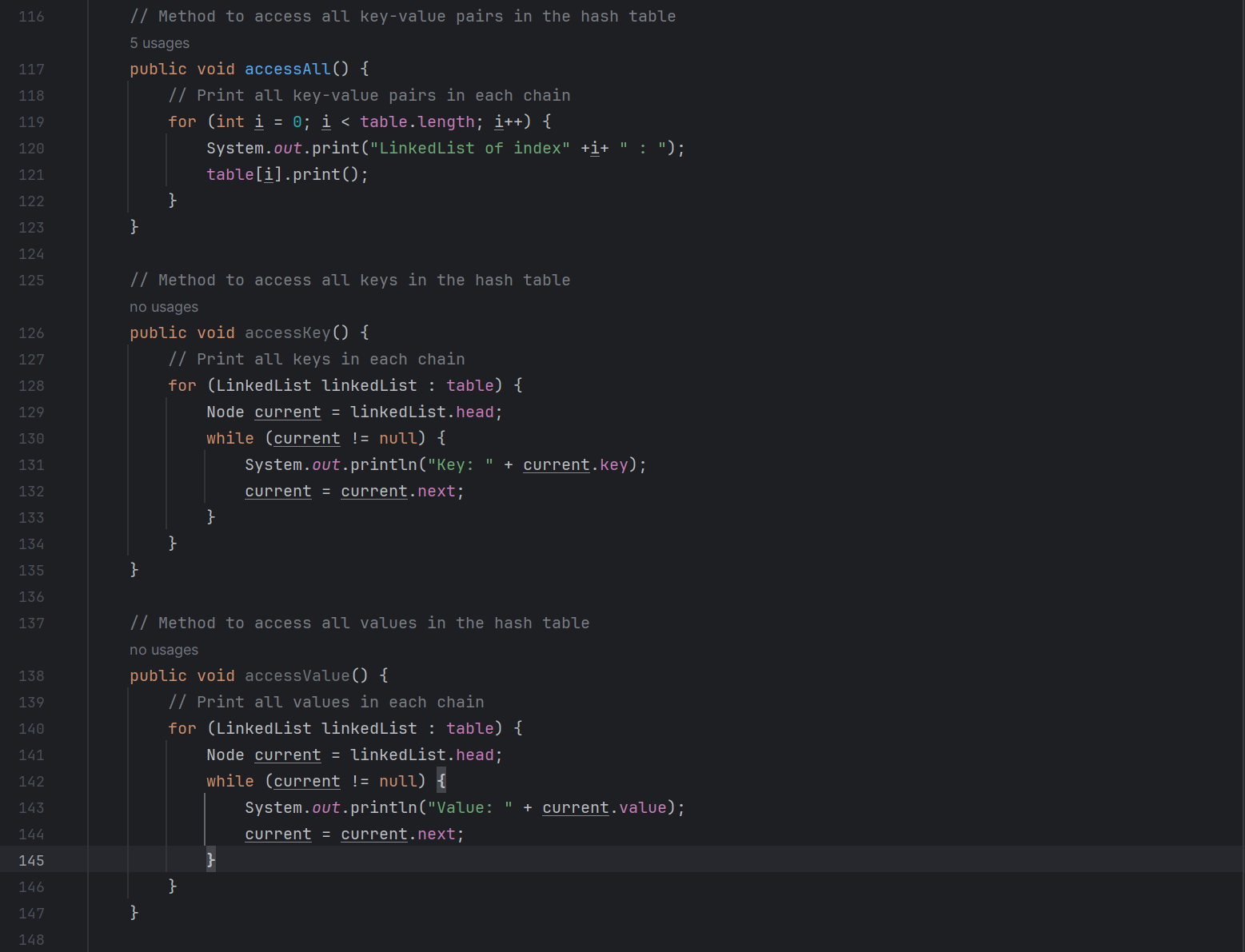


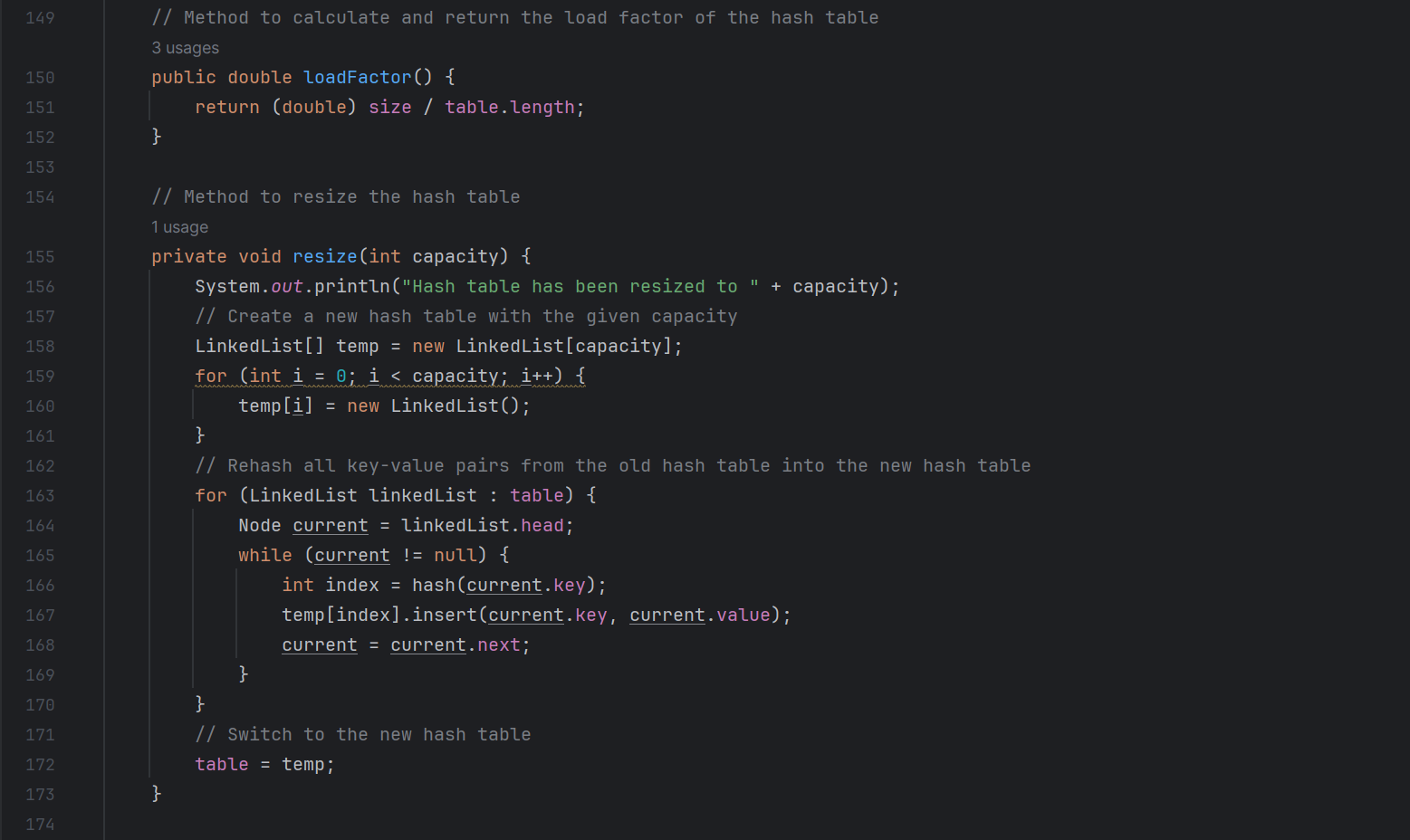


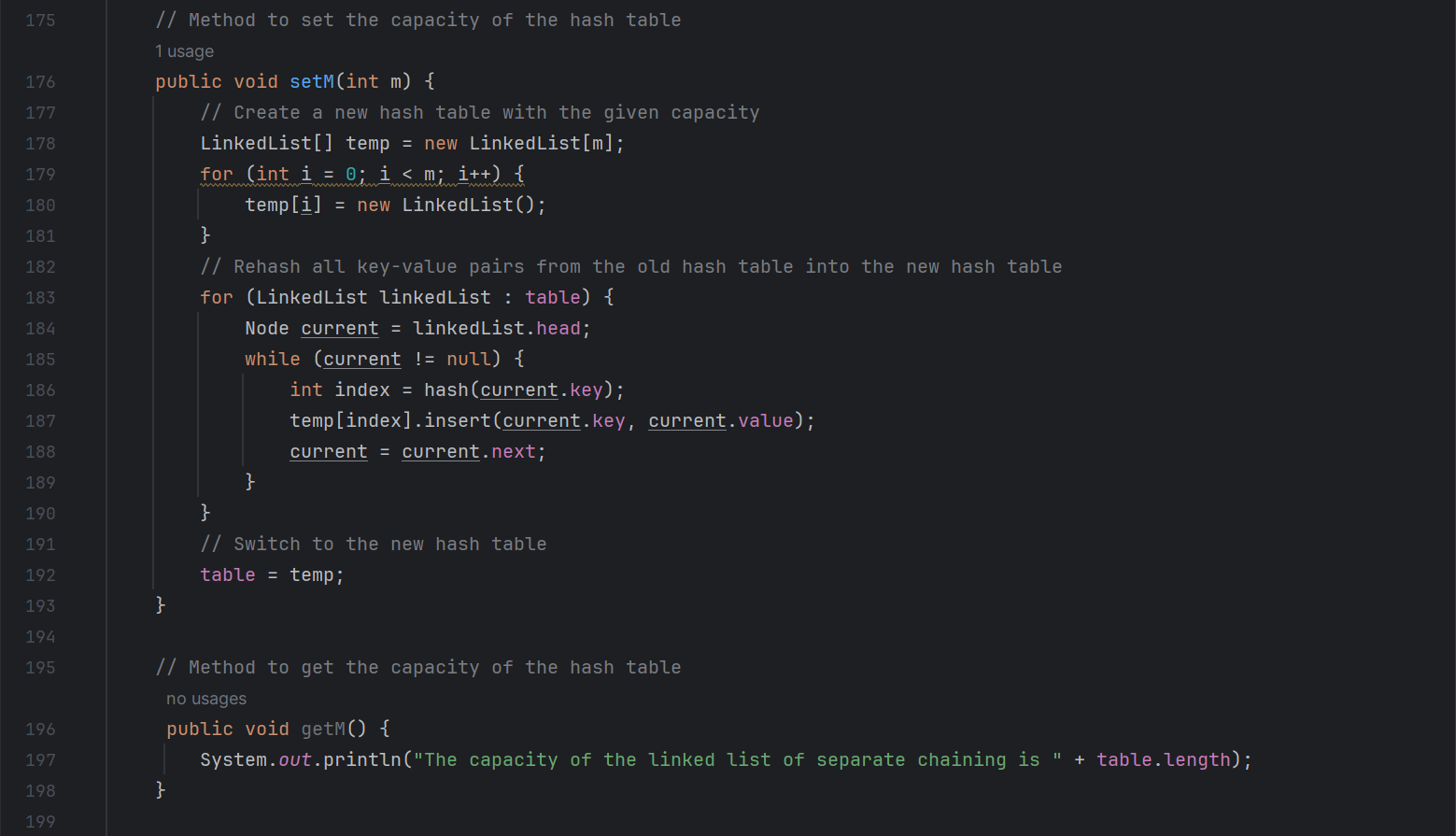


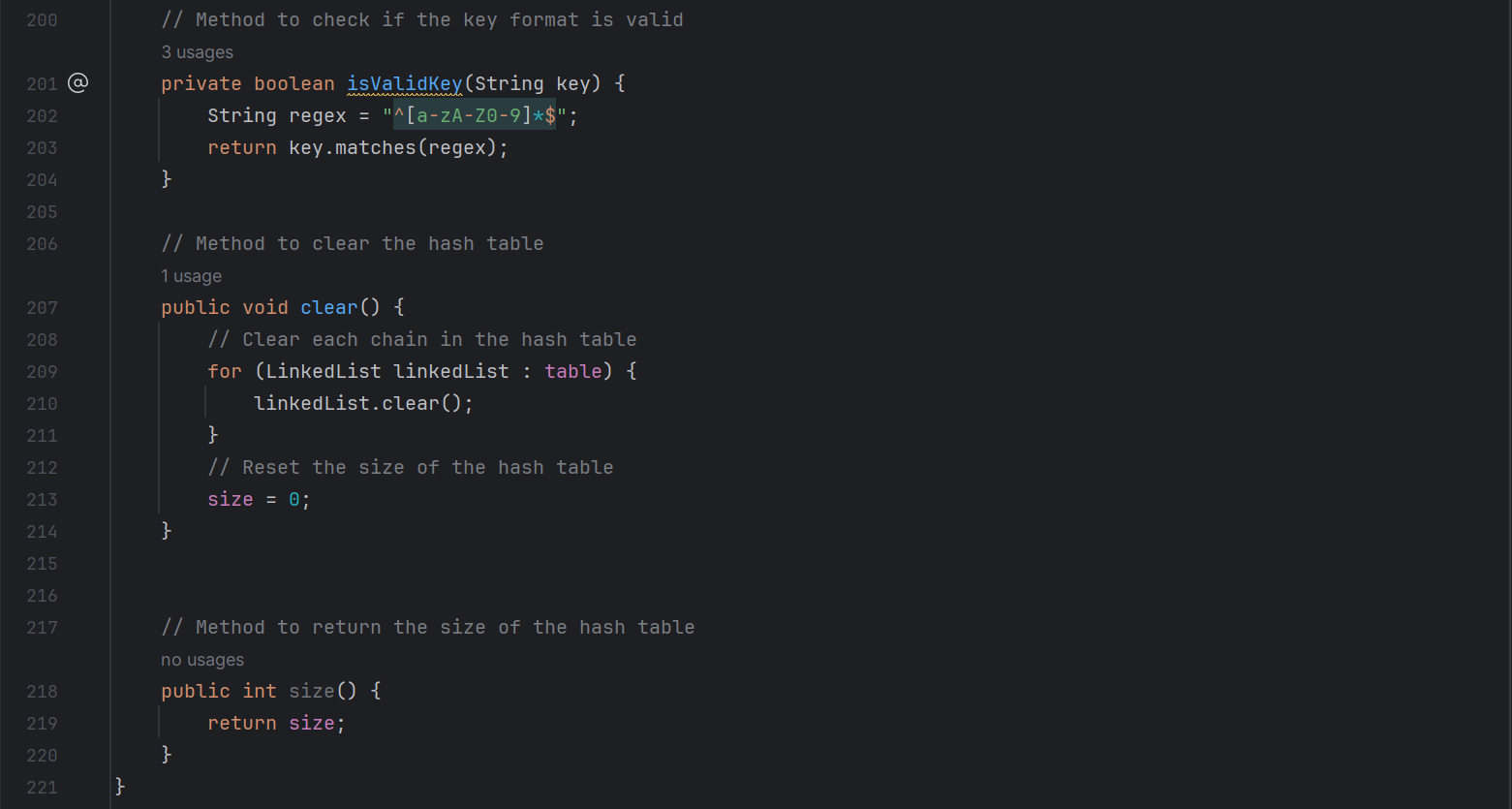












* 1. Explanation

The ChainingHashTable class represents a hash table that uses separate chaining as a collision resolution strategy. Here are the detailed explanations of its methods:

* ChainingHashTable (): This is the constructor of the ChainingHashTable class. It initializes the size of the hash table size to 0, and creates the table array with a size of chain\_Capacity. Each element in the table array is a LinkedList object, which represents a chain in the hash table.
* hash (String key): This method calculates and returns the hash value of a given key. It uses a folding method to calculate the hash value, where it divides the key into chunks of 4 characters, calculates the sum of the ASCII values of the characters in each chunk, and then adds up all the sums. The hash value is then taken modulo the length of the table array to get the index in the hash table.
* insert (String key, String value): This method inserts a key-value pair into the hash table. It first checks if the key or value is null or if the key format is invalid. It then calculates the hash value of the key to get the index in the hash table, and inserts the key-value pair into the corresponding chain. If the load factor of the hash table (the ratio of the number of key-value pairs to the size of the hash table) exceeds a certain threshold, it resizes the hash table to twice its current size.
* search (String key): This method returns the value associated with a given key. It first checks if the key is null or if the key format is invalid. It then calculates the hash value of the key to get the index in the hash table, and searches for the key in the corresponding chain. If the key is found, it returns its associated value. Otherwise, it throws an exception.
* delete (String key): This method deletes a key-value pair by key. It first checks if the key is null or if the key format is invalid. It then calculates the hash value of the key to get the index in the hash table, and deletes the key from the corresponding chain.
* accessAll (): This method prints all key-value pairs in the hash table. It iterates over each chain in the table array and prints all key-value pairs in each chain.
* accessKey (): This method prints all keys in the hash table. It iterates over each chain in the table array and prints all keys in each chain.
* accessValue (): This method prints all values in the hash table. It iterates over each chain in the table array and prints all values in each chain.
* loadFactor (): This method calculates and returns the load factor of the hash table, which is the ratio of the number of key-value pairs size to the size of the hash table table.length.
* resize (int capacity): This method resizes the hash table to a given capacity. It creates a new hash table with the given capacity, rehashes all key-value pairs from the old hash table into the new hash table, and then switches to the new hash table.
* setM (int m): This method sets the size of the hash table to a given value. It creates a new hash table with the given size, rehashes all key-value pairs from the old hash table into the new hash table, and then switches to the new hash table.
* getM (): This method returns the size of the hash table.
* isValidKey (String key): This method checks if the key format is valid. It uses a regular expression to check if the key contains only alphanumeric characters.
* clear (): This method clears the hash table by creating a new table array and setting size to 0.
* size (): This method returns the number of key-value pairs in the hash table.

1. ***Question:*** ***You must create a ChainingHashTable class that represents a hash table by using the separate chaining way for collision resolution.***

***Answer:***

The ChainingHashTable class maintains an array of LinkedList objects, each of which represents a chain in the hash table. This is declared as private LinkedList[] table.

In the ChainingHashTable class, conflicts are resolved using the split chaining method. When two or more keys have the same hash value (i.e. conflict), the keys are placed in the same linked list. Each linked list represents a slot in the hash table.

Here is the relevant code snippet:

// Method to insert a key-value pair into the hash table

public void insert(String key, String value) {

// Calculate the hash value (index) of the key

int index = hash(key);

// Insert the key-value pair into the appropriate chain

table[index].insert(key, value);

// Increase the size of the hash table

size++;

// Resize the hash table if the load factor threshold is exceeded

if ((double) size / (table.length\*chain\_Capacity)> LOAD\_FACTOR\_THRESHOLD) {

resize(2 \* table.length);

}

}

In the insert operation, the hash value of the key is calculated first, and then the key-value pair is inserted in the corresponding linked list. In the search and delete operation, the hash value of the key is also calculated first, and then the search or delete operation is carried out in the corresponding linked list. This way, even if there are multiple keys with the same hash value, they can be distinguished by a linked list, thus resolving conflicts.

1. ***Question:*** ***You must use the doubly linked list devised in task 1 to implement the separate chaining way. The*** ***capacity of the linked list of separate chaining should not exceed 8.***

***Answer:***

Here is the linkedlist I used:

// The array of linked lists (chains) that make up the hash table  
private LinkedList[] table;

Here is the code that I limit the capacity of the linked list of separate chaining should not exceed 8:

// Ensure the chain capacity is within the valid range  
if (*chain\_Capacity* > 8) {  
 *chain\_Capacity* = 8;  
 System.*out*.println("The capacity of the linked list of separate chaining should not exceed 8,it has been "  
 + "set to 8");  
}  
if (*chain\_Capacity* < 1) {  
 *chain\_Capacity* = 1;  
 System.*out*.println("The capacity of the linked list of separate chaining should not be less than 1,it has "  
 + "been set to 1");  
}

1. ***Question:*** ***You must devise a hash function that can work well for string-type data. The hashing strategy of the hash function should be designed differently from that of task 2 and should minimize the occurrence of collisions. You must not use the Java built-in hashCode method, though you can experiment with it.***

***Answer:***

Unlike the previous hash function, this time I use the fold method, which divides the key into multiple blocks, then calculates the hash value of each block, and finally adds the hash values. This method can effectively map long keys to smaller index ranges

Here is the new hashcode:

// Method to calculate the hash value of a key  
public int hash(String key) {  
 int pieceSize = 4; // We're going to fold it in chunks of 4 characters  
 int hash = 0;  
  
 // Calculate the hash value by folding the key  
 for (int i = 0; i < key.length(); i += pieceSize) {  
 int piece = 0;  
 for (int j = 0; j < pieceSize && i + j < key.length(); j++) {  
 piece += key.charAt(i + j) \* (i + j + 1); // Add weight to the character based on its position  
 }  
 hash += piece;  
 }  
 // Modulo the hash value by the length of the hash table to get the index  
 return hash % table.length;  
}

1. The pieceSize variable defines the size of each block, which is set here to 4 characters.

2. The hash variable is used to store the final hash value.

The outer for loop traverses each character of the key in pieceSize steps.

3. The inner for loop processes each block and calculates the hash value of the block. The hash value here is the ASCII value of each character in the block multiplied by its position (weighted) and then obtained by adding these values.

4. The hash value of each block is added to the hash variable.

5. Finally, the cumulative hash value is modelled on the length of the hash table, and the result is the index of the key in the hash table.

1. ***Question:*** ***The implementation can handle errors such as null keys or keys with unexpected formats.***

***Answer:***

The implementation principle and logic are the same as the previous one.

1. ***Question:*** ***The implementation should include methods for inserting, searching, deleting, and accessing key-value pairs, as well as determining load factor***

***Answer:***

Code has implemented above; the function description is:

The insert method is used to insert a key-value pair into the hash table. It calculates the hash value of the key to determine the index of the chain where the key-value pair should be inserted. Then, it inserts the key-value pair into the appropriate chain using the insert method of the LinkedList class.

The search method is used to search for a key in the hash table and return the associated value. It calculates the hash value of the key to determine the index of the chain where the key should be located. Then, it searches for the key in the appropriate chain using the access method of the LinkedList class.

The delete method is used to delete a key-value pair from the hash table. It calculates the hash value of the key to determine the index of the chain where the key should be located. Then, it deletes the key-value pair from the appropriate chain using the delete method of the LinkedList class.

The loadFactor method is used to calculate and return the load factor of the hash table, which is the ratio of the number of key-value pairs to the size of the hash table.

The accessAll, accessKey, and accessValue methods are used to access all key-value pairs, keys, and values in the hash table, respectively.

1. ***Question:*** ***The implementation of the inserting operation can resize the table efficiently if the hash table is too full.***

***Answer:***

The `insert` method in the `ChainingHashTable` class in my project is designed to insert a key-value pair into the hash table and resize the table if necessary. Here's how it works:

```java

// Method to insert a key-value pair into the hash table

public void insert(String key, String value) {

// Check for null key or value

if (key == null || value == null) {

throw new IllegalArgumentException("Key or value is null");

}

// Check for valid key format

if (!isValidKey(key)) {

throw new IllegalArgumentException("Key format is invalid");

}

// Calculate the hash value (index) of the key

int index = hash(key);

// Insert the key-value pair into the appropriate chain

table[index].insert(key, value);

// Increase the size of the hash table

size++;

// Resize the hash table if the load factor threshold is exceeded

if ((double) size / (table.length\*chain\_Capacity)> LOAD\_FACTOR\_THRESHOLD) {

resize(2 \* table.length);

}

// Print the load factor after insertion

System.out.println("Load factor after insertion: " + loadFactor());

}

```

This method first checks if the provided key or value is null or if the key format is invalid. If any of these conditions is true, it throws an `IllegalArgumentException`.

Next, it calculates the hash of the key to find the index where the key-value pair should be inserted in the hash table. It then inserts the key-value pair into the appropriate chain using the `insert` method of the `LinkedList` class.

After inserting the key-value pair, it increases the size of the hash table. If the load factor (the ratio of the number of key-value pairs to the size of the hash table) exceeds a certain threshold (0.75 in this case), it resizes the hash table to twice its current length using the `resize` method.

The `resize` method creates a new hash table with the specified capacity and rehashes all key-value pairs from the old hash table into the new one. This ensures that the load factor of the hash table remains within an acceptable range, which helps to maintain the efficiency of the hash table operations.

1. ***Question:*** ***The implementation of the deleting operation can handle the situation when the key is not found.***

***Answer:***

The implementation principle and logic are the same as the previous one.

1. ***Question:*** ***The implementation can keep track of the load factor of the hash table and display it after each insertion or deletion.***

***Answer:***

The implementation principle and logic are the same as the previous one.

1. ***Question:*** ***The implementation of the searching operation can search for the key and return the corresponding value if the key is found.***

***Answer:***

Yes, the `search` method in the `ChainingHashTable` class in my project is designed to search for a key in the hash table and return the corresponding value if the key is found.

Here's how it works:

```java

// Method to get the value associated with a key

public String search(String key) throws Exception {

// Check for null key

if (key == null) {

throw new IllegalArgumentException("Key is null");

}

// Check for valid key format

if (!isValidKey(key)) {

throw new IllegalArgumentException("Key format is invalid");

}

// Calculate the hash value (index) of the key

int index = hash(key);

// Search for the key in the appropriate chain

Node node = table[index].access(key);

// Throw an exception if the key is not found

if (node == null) {

throw new Exception("Key not found");

}

// Return the value associated with the key

return node.value;

}

```

This method first checks if the provided key is null or if the key format is invalid. If any of these conditions is true, it throws an `IllegalArgumentException`.

Next, it calculates the hash of the key to find the index where the key should be located in the hash table. It then searches for the key in the appropriate chain using the `access` method of the `LinkedList` class.

If the key is found, it returns the value associated with the key. If the key is not found, it throws an `Exception` with the message "Key not found".

1. ***Question:*** ***The implementation of the hash table can resize the table capacity according to the strategy you designed.***

***Answer:***

Yes, the `ChainingHashTable` class in my project has a `resize` method that resize the hash table according to a specific strategy.

Here's how it works:

```java

// Method to resize the hash table

private void resize(int capacity) {

System.out.println("Hash table has been resized to " + capacity);

// Create a new hash table with the given capacity

LinkedList[] temp = new LinkedList[capacity];

for (int i = 0; i < capacity; i++) {

temp[i] = new LinkedList();

}

// Rehash all key-value pairs from the old hash table into the new hash table

for (LinkedList linkedList : table) {

Node current = linkedList.head;

while (current != null) {

int index = hash(current.key);

temp[index].insert(current.key, current.value);

current = current.next;

}

}

// Switch to the new hash table

table = temp;

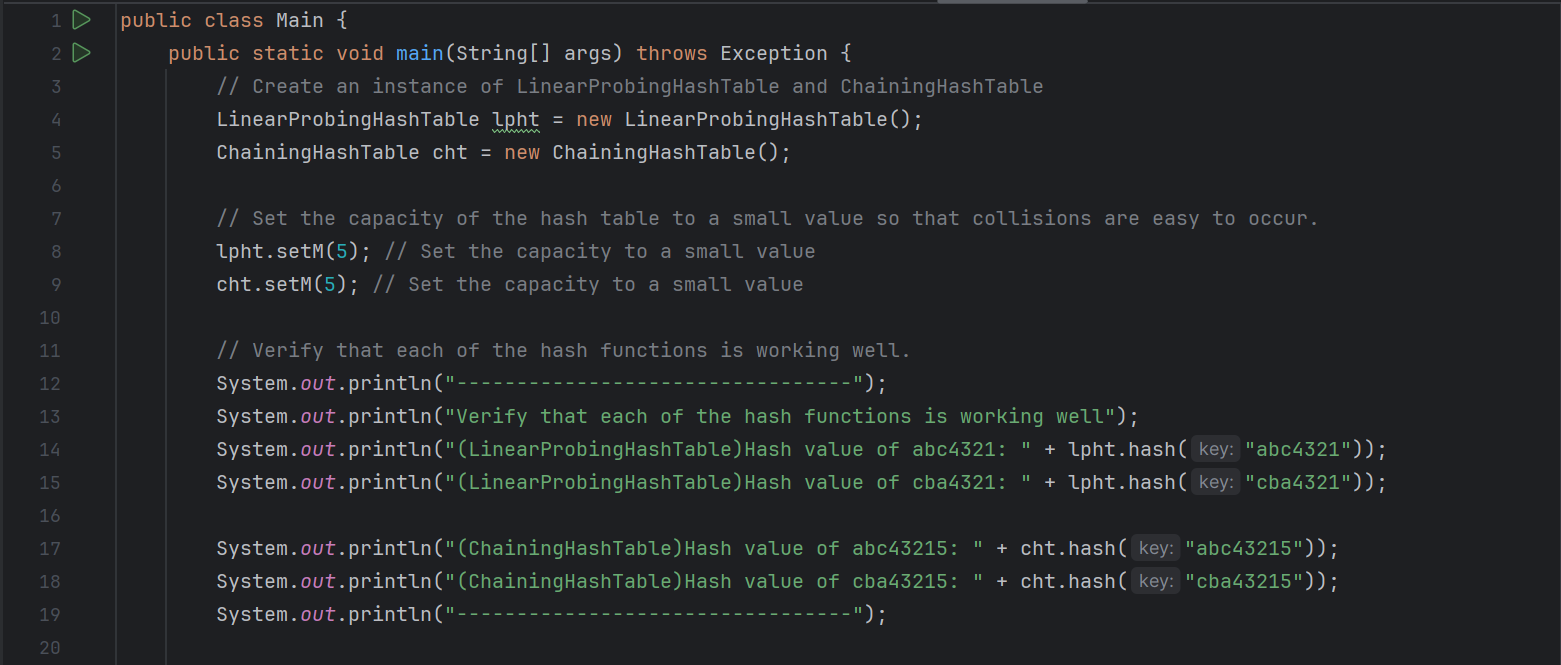
}

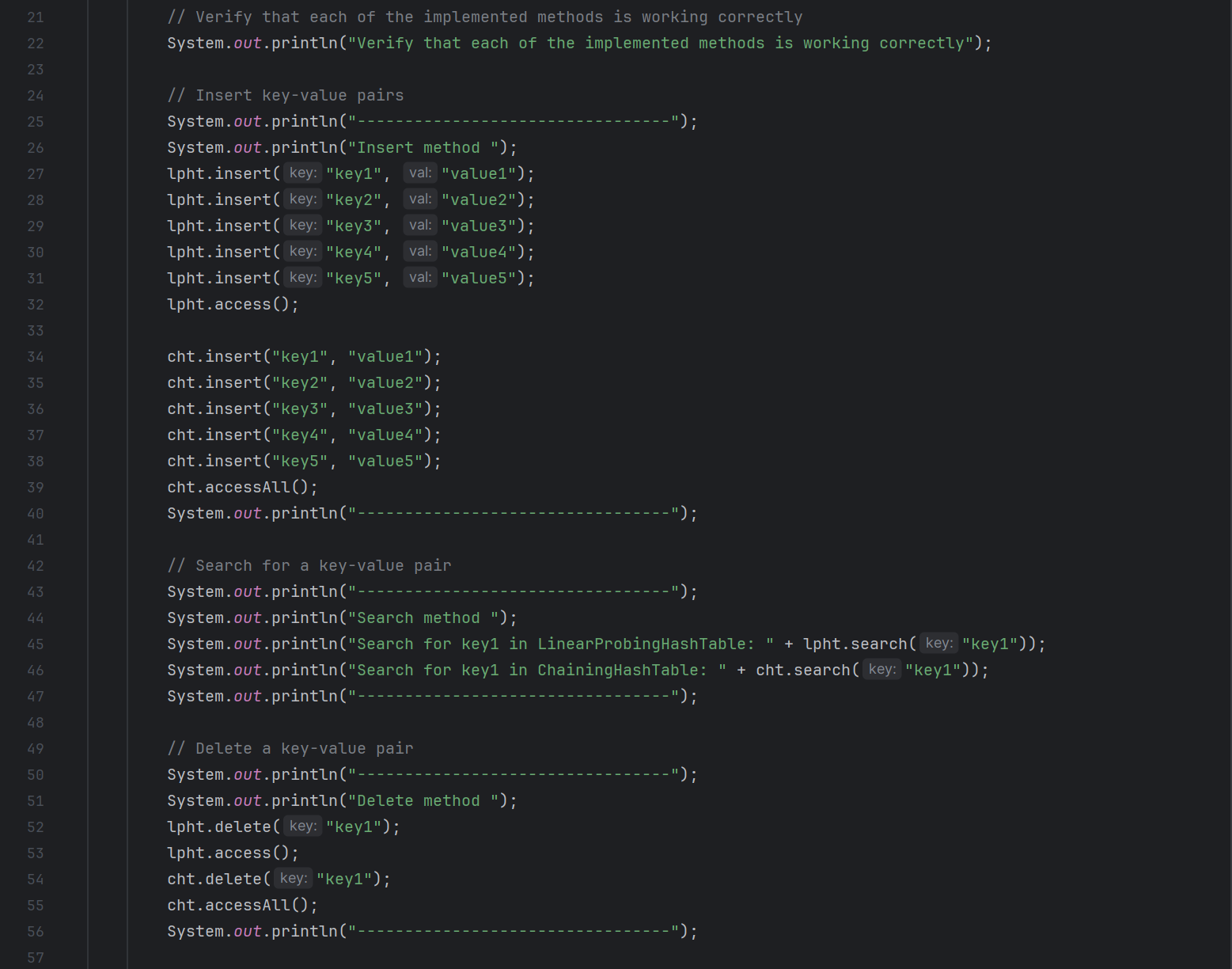
```

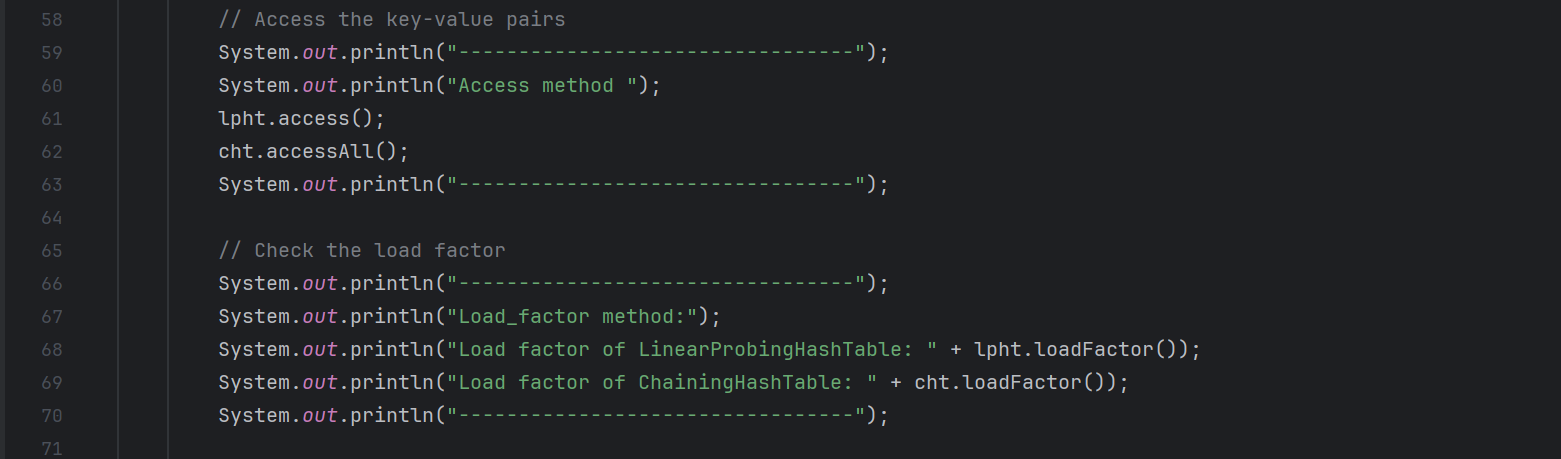
This method is called when the load factor of the hash table (the ratio of the number of key-value pairs to the size of the hash table) exceeds a certain threshold (0.75 in this case). When this happens, the hash table is resized to twice its current size.

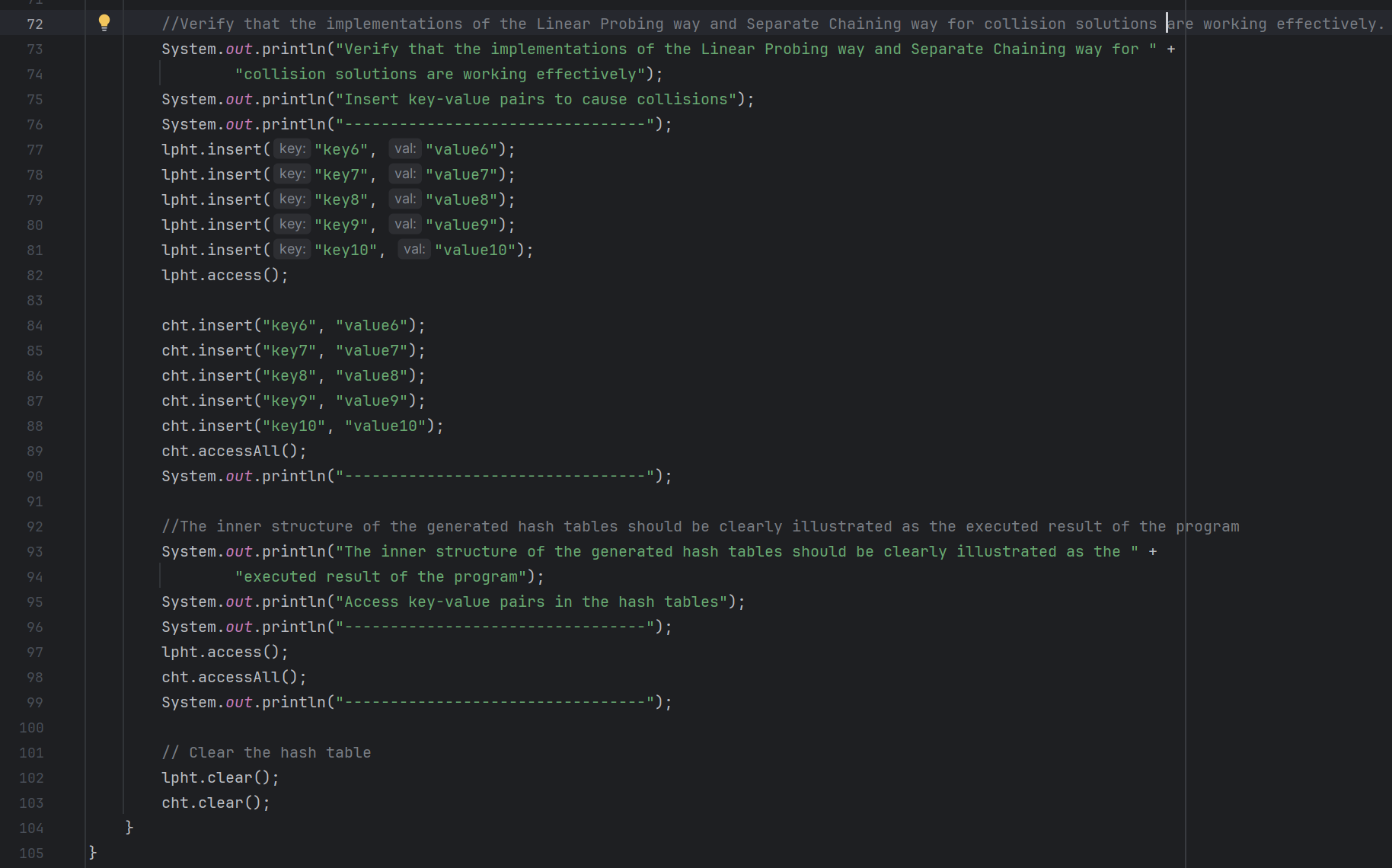
The `resize` method creates a new hash table with the specified capacity and rehashes all key-value pairs from the old hash table into the new one. This ensures that the load factor of the hash table remains within an acceptable range, which helps to maintain the efficiency of the hash table operations.

1. Task\_4
   1. Test Code



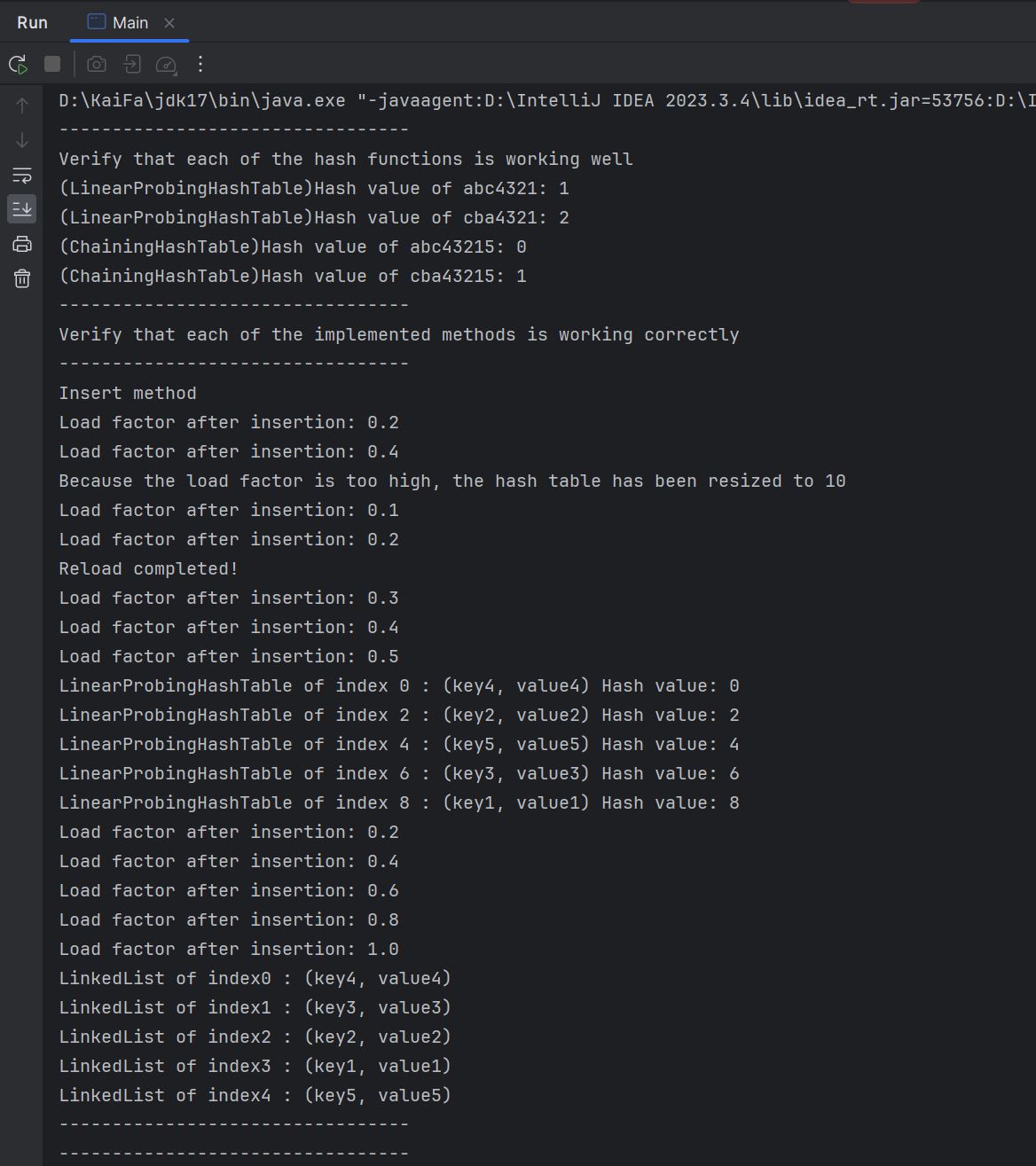


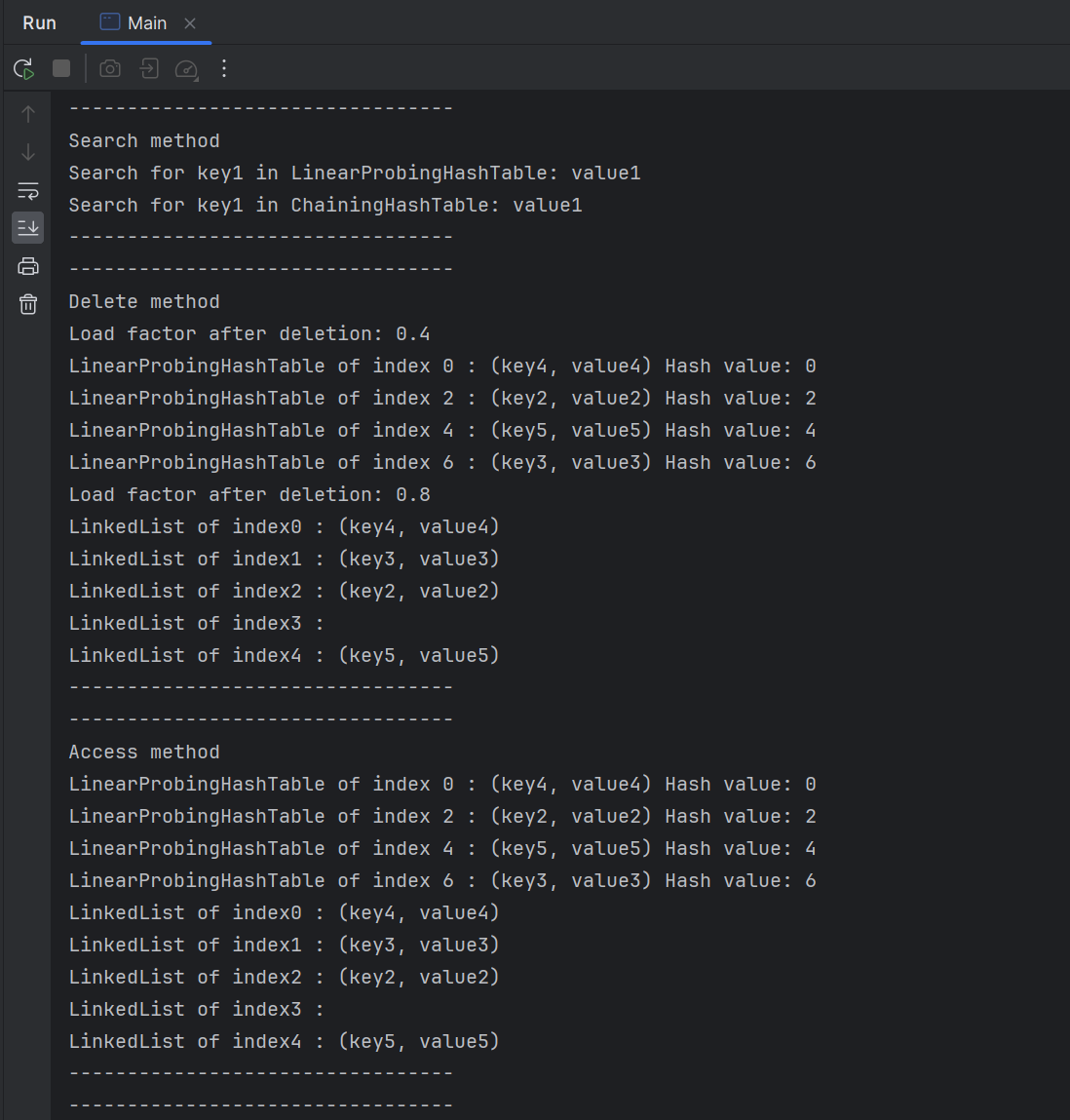


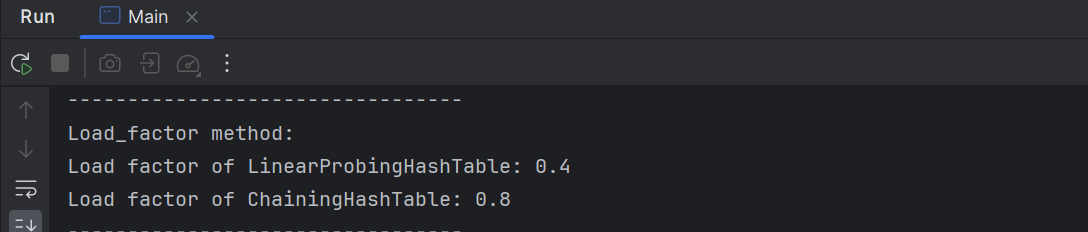


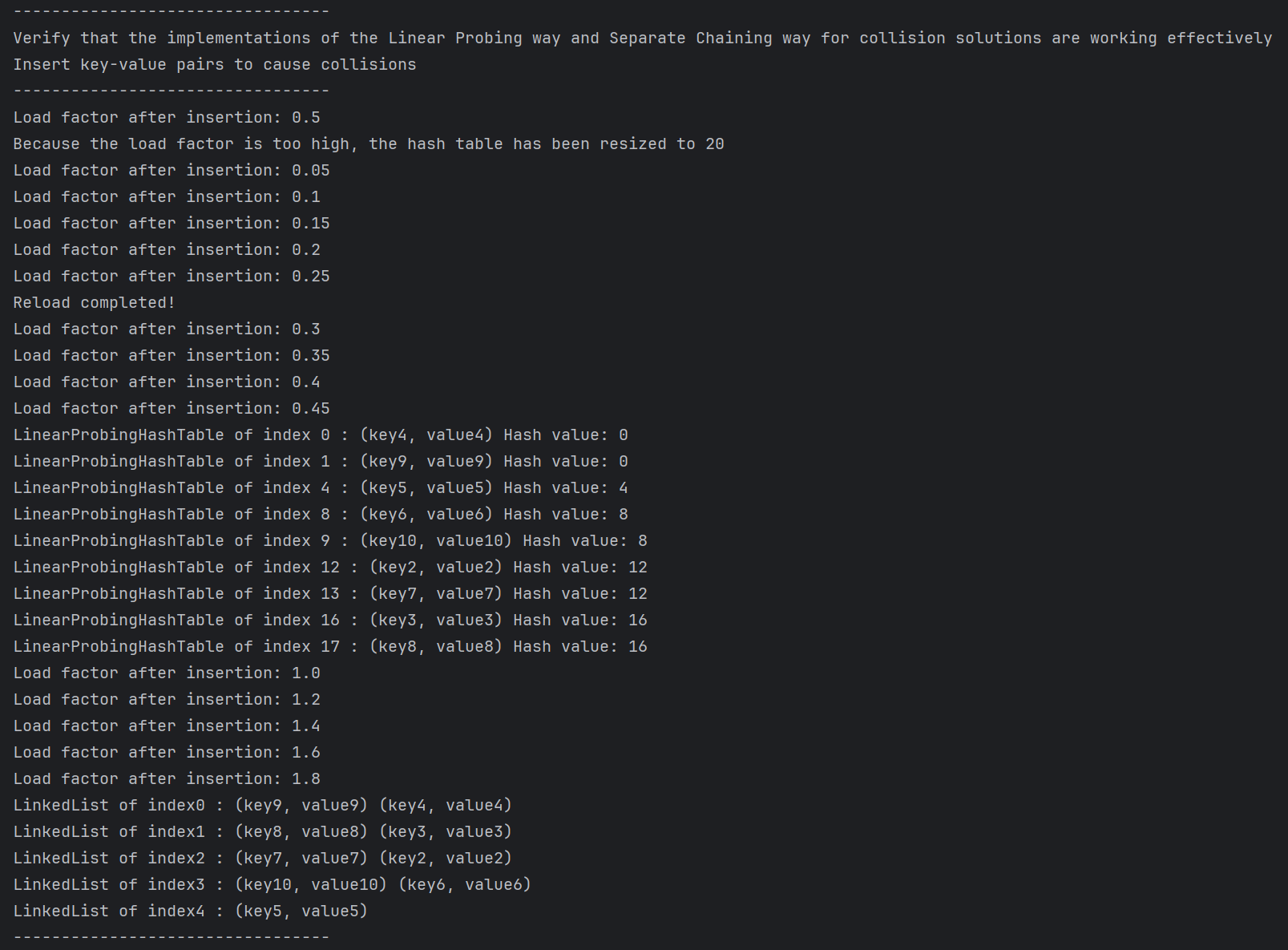
* 1. Explanation

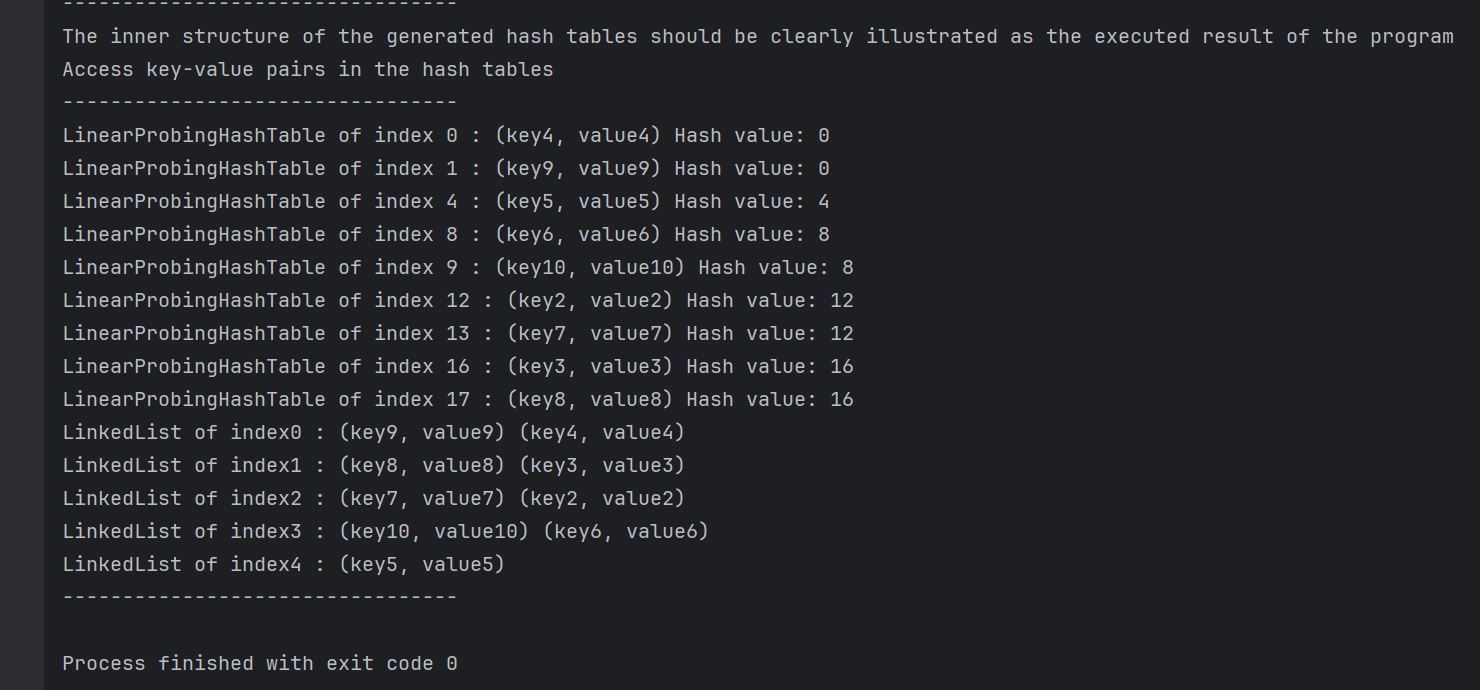
**1.Test Result:**











**Test Analysis:**

**2.Contrast and analyze the two hash tables generated based on the same set of test cases given:**

The two hash tables generated are LinearProbingHashTable and ChainingHashTable. Both hash tables are implemented to store key-value pairs and handle collisions, but they use different strategies.

* LinearProbingHashTable: This hash table uses linear probing to resolve collisions. When a collision occurs, it checks the next slot in the array until it finds an empty slot. The hash table dynamically resizes itself when the load factor is too high or too low to ensure the efficiency of the operations. From the logs, we can see that the keys are not necessarily stored in the order they were inserted due to the collision resolution strategy. Also, when the load factor exceeds a certain threshold (0.5 in this case), the hash table is resized to maintain operation efficiency.
* ChainingHashTable: This hash table uses separate chaining to resolve collisions. Each key-value pair is stored in a linked list (chain) at a specific index in the hash table. The index is calculated using a hash function on the key. If multiple keys hash to the same index, they are stored in the same chain. The hash table dynamically resizes itself when the load factor is too high to ensure the efficiency of the operations. From the logs, we can see that the keys that hash to the same index are stored in the same linked list.

**3.Contrast and analyze the difference between the two hash functions you devised based on the same set of test cases given:**

Both hash functions take a string key as input and return an integer that represents the index in the hash table. However, they use different methods to calculate the hash value.

* LinearProbingHashTable hash function: It uses a prime number as the base and the size of the hash table as the modulus. The hash value is calculated by iterating over each character in the key, multiplying the current hash value by the base, adding the ASCII value of the character, and then taking the modulus of the size of the hash table.
* ChainingHashTable hash function: It uses a folding method to calculate the hash value, where it divides the key into chunks of 4 characters, calculates the sum of the ASCII values of the characters in each chunk, and then adds up all the sums. The hash value is then taken modulo the length of the table array to get the index in the hash table.

**4.Give a rationale and detailed analysis of the effects of two different strategies of collision solution:**

* Linear Probing: This method is simple and easy to implement. It provides good cache performance because it stores all elements in a contiguous block of memory. However, it can suffer from primary clustering where a large number of consecutive slots become occupied, which can lead to long probe sequences and decreased performance.
* Separate Chaining: This method avoids clustering by using a linked list to store keys that hash to the same index. It allows the hash table to maintain performance even when the load factor exceeds 1. However, it requires additional memory to store the pointers in the linked list, and it does not provide good cache performance because the keys are not stored in a contiguous block of memory.

In the given test cases, both methods effectively resolved collisions and maintained the performance of the hash table. There was no obvious clustering, and the data were evenly distributed.