GROUP 3

INDEX	ACTIVITIES	%CONTRIBUTION
PS/CSC/20/0007	UI, Report, IssueItemController.java, DBStack.java, DashboardController.java, IssuedGoodsController.java	10
PS/CSC/20/0020	UI, BillsController.java, DBQueue.java, AddVendorController.java, AddItemController.java, Customer.java	10
PS/CSC/20/0029	UI, Report, LayoutController.java, DBQueue.java, DBStack.java, AddVendorController.java	10
PS/CSC/20/0050	UI, DBConnection.java, InventoryController.java, DBList.java, DBHashMap.java, DBQueue.java	10
PS/CSC/20/0051	UI, Report, DashboardController.java, IssuedItem.java, IssuedGoodsController.java, DBStack.java	10
PS/CSC/20/0054	UI, DBQueue.java, DBStack.java, DBList.java, LayoutController.java, UniqueRandomCodeGenerator.java	10
PS/CSC/20/0055	UI, InventoryController.java, DBStack.java, DBQueue.java, DBList.java, DBHashMap.java	10
PS/CSC/20/0073	UI, Report, IssuedGoodsController.java, DBQueue.java, VendorsController.java, BillsController.java	10
PS/CSC/20/0104	UI, Report, App.java, IssuedItem.java, Vendor.java, DBStack.java	10
PS/CSC/20/0133	UI, Bill.java, Item.java, Customer.java, DBQueue.java, VendorsController.java	10

CONTENTS

INTRODUCTION

This report presents an overview of custom data structure implementations of a database designed for inventory management. It is developed in Java and utilizes a MySQL database for data storage, providing dynamic resizing capabilities and type flexibility. This report will briefly discuss the implementation and analyze the space and time complexities of the main operations.

DBSTACK<T>

IMPLEMENTATION

DBStack<T> is a generic stack class that connects to a MySQL database, creating a table for the stack if it doesn't exist. The table schema contains information about stock items, such as name, category, quantity, unit of measurement, cost price, selling price, sold, expiry date, created at, and vendor.

The constructor of **DBStack<T>** initializes the database connection, creates the table (if not already present), and sets the stack size and top index. It also populates the table with default NULL values if the table is empty.

The **DBStack<T>** class provides several methods for stack operations, including push, pop, isEmpty, and isFull. Additionally, the expandStackSize method is responsible for doubling the capacity of the stack and updating the indices accordingly. The getItems method retrieves all items from the database as an ObservableList<Item> for further processing.

TIME COMPLEXITY ANALYSIS

PUSH

The push operation updates the row in the database with the new item's data. The time complexity of the push operation is O(1) since it performs a single database update operation. In the worst case, it may trigger an expansion of the queue, which has an O(n) time complexity.

POP

The pop operation retrieves an item from the stack and sets the corresponding row in the database to NULL values. This operation's time complexity is O(1) as it involves one database query for selection and one for updating.

ISEMPTY AND ISFULL

Both the isEmpty and isFull methods have a time complexity of O(1), as they involve simple comparisons of the top index with the stack's capacity.

EXPANDSTACKSIZE

This operation doubles the stack capacity by copying the existing items to new rows in the database and setting the old rows to NULL values. The time complexity of this operation is O(n), where n is the number of items in the stack before expansion, as it involves iterating through all rows and performing insert and update operations.

SPACE COMPLEXITY ANALYSIS

The space complexity of **DBStack<T>** is determined by the number of items stored in the database. As the stack grows, the expandStackSize method doubles the capacity, resulting in a space complexity of O(n), where n is the number of items in the stack. The database size will increase accordingly to accommodate the additional rows, but the overall space complexity remains linear.

DBQUEUE<T>

The **DBQueue** implementation provides a queue data structure backed by a MySQL database. It supports basic operations such as enqueue, dequeue, isEmpty, and isFull. Additionally, it can resize itself and fetch all items stored in the queue.

TIME COMPLEXITY ANALYSIS

ENQUEUE

Enqueue operation involves updating a single row in the database. In the worst case, it may trigger an expansion of the queue, which has an O(n) time complexity. However, this happens infrequently, and the amortized time complexity for enqueue remains O(1).

DEQUEUE

Dequeue operation involves fetching and updating rows in the database. Additionally, it requires shifting all the rows up by one position, resulting in an O(n) time complexity. Shifting is done so as to prevent excess unused space thereby increasing the space complexity.

ISEMPTY

The isEmpty operation checks the front and rear indices, making it an O(1) operation.

ISFULL

Similarly, the isFull operation compares the rear index to the capacity, making it an O(1) operation.

EXPANDQUEUESIZE

This operation doubles the queue capacity by adding new rows to the database, which has an O(n) time complexity.

DBLIST<T>

The **DBList** class is a generic class that represents a list-like data structure. The operations include adding items to the list, removing items from the list, getting items from the list, and setting items in the list.

TIME COMPLEXITY ANALYSIS

ADD

Methods to add an item to the list, either at a specific index or at the end of the list. It shifts the elements to the right of the specified index and inserts the new item. In the worst case the time complexity is O(n) but is O(1) for best case.

GET

Methods to get an item from the list at the specified index. It retrieves the item from the database table based on the index. Time complexity for this method is O(1).

REMOVE

Removes an item at a specified index and shifts the remaining elements to the left. It also removes the last item from the database table. Time complexity is O(n) for worst case and O(1) for best case.

SET

With a time complexity O(1), the set method replaces an item at a particular index with a new one.

SIZE

Returns the size of the list. Time complexity: O(1)

DBHASHMAP<T>

IMPLEMENTATION

DBHashMap is a generic class with key-value pairs of types K and V. Upon instantiation, the class sets up a connection to a MySQL database and creates a table if it does not exist. The table is defined with columns for key_hash, key_value, and value, where the key_hash is an integer, and both key_value and value are of type VARCHAR(255) and JSON, respectively.

TIME AND SPACE COMPLEXITY ANALYSIS

PUT

This method inserts or updates the key-value pair in the table. It first computes the key_hash using the key's hashcode and then converts the HashMap value to a JSON string. Depending on whether the key exists or not, the method either updates the existing entry or inserts a new entry in the table. The time complexity of this method is O(1), assuming that the database operations are performed in constant time. The space complexity is also O(1), as only a constant amount of additional space is needed for the conversion of the HashMap to a JSON string.

GET

This method retrieves the value associated with a given key from the table. It computes the key_hash using the key's hashcode, executes a SELECT query to find the value, and then converts the JSON string back to a HashMap before returning the result. The time complexity of this method is O(1), assuming that the database operations are performed in constant time. The space complexity is also O(1), as only a constant amount of additional space is needed for the conversion of the JSON string to a HashMap.

REMOVE

This method removes the entry associated with a given key from the table. It first retrieves the value associated with the key using the get() method, and if the value exists, it computes the key_hash using the key's hashcode and executes a DELETE query to remove the entry from the table. The time complexity of this method is O(1), assuming that the database operations are performed in constant time. The space complexity is also O(1), as only a constant amount of additional space is needed for the conversion of the JSON string to a HashMap and the retrieval of the key-value pair.

CONTAINSKEY

This method checks if a given key exists in the table. It computes the key_hash using the key's hashcode, executes a SELECT COUNT(*) query, and returns true if the count is greater than 0, indicating the key is present. The time complexity of this method is O(1), assuming that the database operations are performed in constant time. The space complexity is also O(1), as only a constant amount of additional space is needed for the retrieval of the key-value pair.