Integrating task
task management and reminder systems

DANIEL MEJIA | JEAN GALLEGO

Content

1. INTRODUCTION.

Efficient task and reminder management is essential for personal and professional productivity and success. To address this need, the development of a task and reminder management system has been proposed that will allow users to effectively add, organize and manage their daily to-dos.

This project aims to design and implement a comprehensive system that addresses all dimensions of task management, from the storage of information to the presentation of an intuitive user interface and advanced functionalities. The system will be based on efficient algorithms and data structures to ensure optimal performance.

1.1. Components and functionalities (SGTR)

1.1.1. Task storage and reminders

We will use a hash table to store tasks and reminders. Each entry will have a unique identifier as a key and the task/reminder information as a value. This information includes the title, description, due date, and priority.

1.1.2. User Interface

We will design a user interface that allows users to add, modify, and delete tasks and reminders. Users will be able to see a list of all tasks and reminders, sorted by due date or priority.

1.1.3. Priority Management

There will be two categories of tasks: "priority" and "non-priority".

- For priority tasks, we will use a priority queue to organize them according to their importance. When a user adds a new priority task, it will be added to the queue according to its importance, which will ensure that important tasks are handled first.
- Non-priority tasks are handled on a first-in-first-out (**FIFO**) basis.

1.1.4. Undo function

We will implement a method to undo the actions performed by a user in the system. We will use a stack (**LIFO**) to keep track of the actions performed. The general process for the "Undo" function will be:

1. Create an action stack to keep track of the user's actions.
2. Record each action performed by the user in the stack, including details of the action and the task affected.
3. Implement a last action undo method that unstacks the last action and undoes the corresponding action based on the information stored in the stack.
4. Provide the user interface with an "Undo" option to undo the last action performed.

With these features, our task and reminder management system will be efficient and allow users to organize and manage their tasks effectively.

2. REQUIREMENTS ANALYSIS.

1.2.Case study:

Client	**Marlon Gomez**
User	The system's end-users will consist of individuals seeking to efficiently manage their daily tasks and reminders.
Functional Requirements	**R1. Add Tasks**
	R2. Modify Tasks
	R3. Remove Tasks
	R4. Task Classification
	R5. Sort Tasks
	R6. Undo Actions
Non-Functional Requirements	Backup and Recovery
	Browser Compatibility
	scalability
	Data Security

2.1.1. Functional Requirement N°1

<table>
<tr><th colspan="4">R1</th></tr>
<tr><td>Name</td><td colspan="3">Add Task</td></tr>
<tr><td>overview</td><td colspan="3">Allow users to add new tasks by specifying title, description, due date, and priority.</td></tr>
<tr><td rowspan="2">Inputs</td><td>Input Name</td><td>Data Type</td><td>VVC</td></tr>
<tr><td>taskDetails</td><td>String (Title and Description), Date (Due Date), Integer (Priority)</td><td>Title and Description should not be empty, Due Date should be in the future, Priority should be within a defined range.</td></tr>
<tr><td colspan="4">Result or Postcondition</td></tr>
<tr><td rowspan="2">Outputs</td><td>Output Name</td><td>Data Type</td><td>Format</td></tr>
<tr><td>taskAdded</td><td>Boolean</td><td>True if the task is added successfully, False if there is an error.</td></tr>
</table>

2.1.2. Functional Requirement N°2

R2			
Name	**Modify Tasks**		
overview	Allow users to edit existing tasks to update information such as title, description, or due date.		
Inputs	***Input Name***	***Data Type***	***VVC***
	updateTaskDetails	String (Title and Description), Date (Due Date)	Title and Description should not be empty, Due Date should be in the future, Priority should be within a defined range.
Result or Postcondition			
Outputs	***Output Name***	***Data Type***	***Format***
	taskAdded	Boolean	True if the task is modified successfully, False if there is an error.

2.1.3. Functional Requirement N°3

R3			
Name	**Delete Tasks**		
overview	Allow users to delete tasks that are no longer relevant.		
Inputs	***Input Name***	***Data Type***	***VVC***
	taskID	Integer	Task ID should correspond to an existing task.
Result or Postcondition			
Outputs	***Output Name***	***Data Type***	***Format***
	taskDeleted	Boolean	True if the task is deleted successfully, False if there is an error.

2.1.4. Functional Requirement N°4

R4			
Name	**Categorize Tasks**		
overview	Allow users to categorize tasks as "Priority" or "Non-priority" when creating them.		
Inputs	***Input Name***	***Data Type***	***VVC***
	taskCategory	String	Should be either "Priority" or "Non-priority"
Result or Postcondition			
Outputs	***Output Name***	***Data Type***	***Format***
	taskCategorized	Boolean	True if the task is deleted successfully, False if there is an error.

1.1.1. Functional Requirement N°5

R5			
Name	**Sort Tasks**		
overview	Allow users to view a list of all their tasks, sorted by due date or priority.		
Inputs	***Input Name***	***Data Type***	***VVC***
	sortingOption	String	Should be either "Due Date" or "Priority"
Result or Postcondition			
Outputs	***Output Name***	***Data Type***	***Format***
	taskList	List Of Tasks	Display the tasks in the selected sorting order.

2.1.5. Functional Requirement N°6

R6			
Name	**Undo Actions**		
overview	Implement a feature that allows users to undo the last action performed in the system.		
Inputs	***Input Name***	***Data Type***	***VVC***
	undoRequest	*N/a*	*N/a*
Result or Postcondition			
Outputs	***Output Name***	***Data Type***	***Format***
	actionUndone	*Boolean*	True if the last action is successfully undone, False if there are no actions to undo or if there is an error.

3. ANALYSIS.

In computer science, algorithm analysis is an essential process for understanding the performance and efficiency of algorithms used to solve problems. Two critical aspects of this analysis are time complexity and space complexity. Time complexity refers to the time an algorithm takes to complete its task as a function of input size, while space complexity refers to the amount of memory or storage space an algorithm uses as a function of input size.

In this section, we will perform a detailed analysis of the time and space complexity of at least two algorithms implemented in this project. This analysis will allow us to evaluate and compare the performance of these algorithms in terms of efficiency and resources used. By better understanding the complexity of our algorithms, we will be able to make informed decisions about which is the best choice for a given application.

1.3. Implemented algorithm time complexity analysis.

During this analysis, we will break down each line of code within the put and get methods and evaluate the individual time complexity of each method. To fully understand the performance of these methods, we will identify the key operations that influence execution time and consider both best and worst cases.

3.1.1. Hash tables - Put Method

```
public void put(K key, V value){

    int index = hash(key); // O(1) - hash index calculation.
    table[index] = new HTNode<>(key, value); // O(1) - Assignment to the hash table
    size++; // O(1) - Increase in size.

}
```

3.1.1.1. Key operations.

1. **Calculate the hash index:** The first key operation is to compute the hash index of the key. This is done using a hash function, which is generally of constant time, resulting in **O(1).**
2. **Assignment to the hash table:** The next operation is to map the value in the hash table to the index calculated in the previous step. This is a constant time operation. **O(1).**
3. **Size increment:** After inserting the element into the hash table, the size of the data structure (size) is incremented. This operation is also time constant. **O(1).**

3.1.1.2. Best case:

In the best case, when there are no collisions (each key maps to a unique index), all these operations are constant time (O(1)).

3.1.1.3. Worst case:

In the worst case, when there are collisions and multiple keys must map to the same hash index, the execution time may increase due to the index lookup and the possibility of reassignment. In the worst case, the complexity could be O(n), where n is the number of colliding elements in each index.

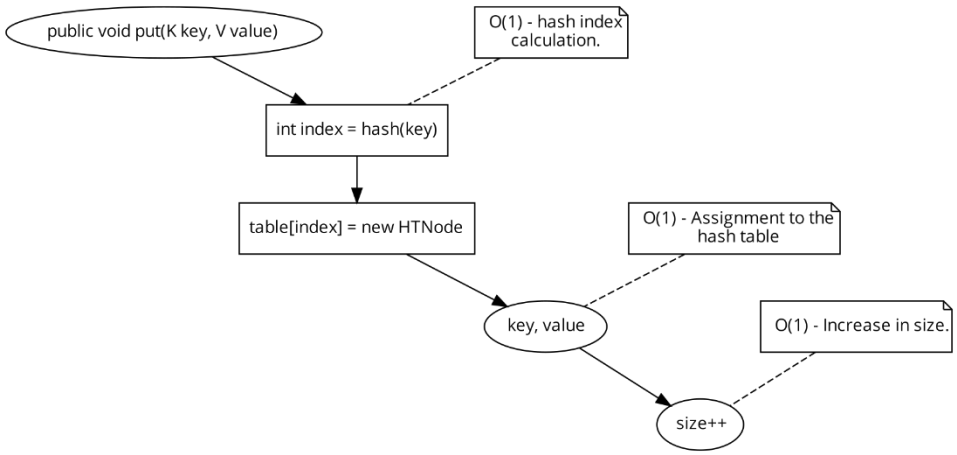

Figure 1 flowchart

3.1.2. Hash tables – Get Method.

```
public V get(K key) {

    V value = null; // O(1) - Initialization of value.
    int i = hash(key); // O(1) - Hash index calculation.
    if(table[i] != null && table[i].getKey().equals(key)){// O(1) - Checking and accessing the value.

        value = table[i].getValue(); // O(1) - Access to the value.
    }

    return value;
}
```

3.1.2.1. Key operations.

1. **Calculate the hash index:** The first key operation is to compute the hash index of the key. This is done using a hash function, which is generally of constant time, resulting in **O(1).**

2. **Check and access the value:** The next step is to check if the computed index contains an element and if the key in this element matches the given key. If the match is successful, the associated value is accessed. This check and access is time constant, **O(1).**

3.1.2.2. Efficiency

The efficiency of the get method depends on the distribution of keys and the implementation of the hash function used. In general, the get method tends to be efficient for most searches but may be affected by collisions in exceptional situations.

1.4. Implemented algorithm space complexity analysis.

In this analysis, we will examine in detail the amount of memory used by the methods . We will evaluate memory usage in terms of constant memory (fixed memory) and additional memory (memory that varies according to inputs and operations). To do this, we will look at the data structures used and how they are handled in each method.

3.1.3. Task Manager – Add Task Method

```
public String addTask(int key, int id, String title, String desc, String limitDate, int prio){
    String msg = ""; // Constant space O(1)

    if(taskTable.getSize() != 10){ // Constant space O(1) (assuming getSize is O(1))
        Priority priority = null; // Constant space O(1)

        if(prio == 1){ // Constant space O(1)
            priority = Priority.PRIORITY; // Constant space O(1)
        }
        else if(prio == 2){ // Constant O(1) space.
            priority = Priority.NONPRIORITY; // Constant space O(1)
        }

        Task newTask = new Task(id, title, limitDate, desc, priority); // Additional space to create a Task object.

        taskTable.put(key, newTask); // Additional space to add an element to the hash table

        if(priority == Priority.PRIORITY){ // Constant space O(1)
            priorityQueue.enqueue(newTask); // Additional space to add an item to the priority queue
        }
        else if(priority == Priority.NONPRIORITY){ // Constant O(1) space.
            nonPrioQueue.enqueue(newTask); // Additional space to add an item to the non-priority queue
        }

        msg = "Task added."; // Constant O(1) space.
        actions.push(new ActionRecord(key,newTask, Action.ADD)); // Additional space to add an item to the stack
    }
    else{
        msg = "Table full!"; // Constant space O(1)
    }

    return msg; // Constant O(1) space.
}
```

3.1.3.1. Analysis Memory usage.

Constant space:

local variables such as msg, priority, newTask, newDesc, newDate, and other local variables.

Additional space:

- taskTable (HashTable): The amount of space required depends on the size of the hash table. In this case, a hash table is created with an initial capacity of 10, so the additional space depends on this capacity.
- PriorityQueue: The space required depends on the number of items in the priority queue at the time.
- nonPrioQueue: The amount of space required depends on the number of items in the non-priority queue at the time.
- actions (stack): The space required depends on the number of action records stored in the stack.

3.1.4. Task Manager – Undo Method

```
public String undo() {
    String msg = "";  // Constant space O(1)
    ActionRecord action = actions.pop();  // Constant space O(1) for pop


    if(action != null) {  // Constant space O(1) for checking if action is not null


        if (action.getType() == Action.ADD) {  // Constant space O(1)
            Task addedTask = action.getTask();  // Constant space O(1)
            int addedTKey = action.getTKey();  // Constant space O(1)
            taskTable.remove(addedTKey);  // Additional space to remove an element
            msg = "Add action undone.";  // Constant space O(1)
        } else if (action.getType() == Action.MODIFY) {  // Constant space O(1)
            int modifiedTKey = action.getModifiedTKey();  // Constant space O(1)
            String modifiedDesc = action.getModifiedDesc();  // Constant space O(1)
            String modifiedDate = action.getModifiedDate();  // Constant space O(1)
            Task modifiedTask = taskTable.get(modifiedTKey);  // Constant space O(1)
            modifiedTask.setDesc(modifiedDesc);  // Constant space O(1)
            modifiedTask.setLimitDate(modifiedDate);  // Constant space O(1)
            msg = "Modify action undone.";  // Constant space O(1)
        } else if (action.getType() == Action.REMOVE) {  // Constant space O(1)
            Task removedTask = action.getTask();  // Constant space O(1)
            int removedTKey = action.getTKey();  // Constant space O(1)
            taskTable.put(removedTKey, removedTask);  // Additional space to put an element
            msg = "Remove action undone.";  // Constant space O(1)
        }
    } else {
        msg = "No actions recorded.";  // Constant space O(1)
    }
    return msg;  // Constant space O(1)
}
```

3.1.4.1. Analysis Memory usage.

Constant space:

Local variables such as msg, action, addedTask, addedTKey, modifiedTKey, modifiedDesc, modifiedDate, removedTask, removedTKey, and other local variables.

Additional space:

Depending on the action being undone, hash table elements and queues may be added or removed. The additional space depends on the specific action being undone.

4. DESIGN.

1.5. Hash Tables TAD:

HashTable <K,V>
HashTable<K,V> *= {Capacity = <capacity>, Size = <size>, Table =<table>}*
{inv.:capacity > 0}
PrimitiveOperations: • *HashTable: Integer x <K, V > x HashTable → <HashTable>* • *Hash: K → <Integer>* • *Put: <K,V> x <HashTables> → <HashTable>* • *Get: <K> → <V>* • *Remove <K> → <HashTables>*

HashTable (Capacity)
"Creates a HashTable with a given capacity"
Pre: *Capacity > 0* ***Post:*** *Creates a new hash table with capacity = <capacity>*

Get (K)
"Puts a value V in the Hash table given a key K to which the hash function is applied"
Pre: *K != 0 | V with key K should be already on the hash table* ***Post:*** *Puts the value on* the hash table

Put (K, V)
"Puts a value V in the Hash table given a key K to which the hash function is applied"
Pre: *K != 0* ***Post:*** *Puts the value on* the hash table

Hash (K)
"Creates a HashTable with a given capacity"
Pre: *K != 0* ***Post:*** *Returns an integer that represent the hashed key*

HTNode <K,V>

HTNode<K, V> *= {Key = <K>, Value = <V>}*

{inv.: Key != 0}

PrimitiveOperations:

- ***HTNode:*** *K x V x HTNode → <HTNode>*
- ***GetKey****: HTNode → <K>*
- ***GetValue****: HTNode → <V>*

HTNode (K,V)

"Creates a node of the HashTable, which contains a K key and a V value"

Pre: *K != 0, V != NULL*
Post: *HTNode = {Key: K. Value: V}*

GetKey ( )

"Returns the key K that the node contains"

Pre: *K != NULL*
Post: *<Key>*

GetValue ( )

"Returns the key K that the node contains"

Pre: *K != NULL*
Post: *<Key>*

1.6. DoubleLinkedList TAD

DoubleLinkedList < T >
DoubleLinkedList <T> = *{Head = <head>, Tail = <tail>, Elements = <numElements> }*
{inv.: numElements >= 0}
PrimitiveOperations: • ***DoubleLinkedList***: → *<DoubleLinkedList>* • ***Add***: *T x DoubleLinkedList* → *<DoubleLinkedList>* • ***Delete***: *T x DoubleLinkedList* → *<DoubleLinkedList>* • ***Size***: *DoubleLinkedList* → *<DoubleLinkedList>* • ***IsEmpty***: *DoubleLinkedList* → *<DoubleLinkedList>*

DoubleLinkedList ( )
"Creates an empty Double Linked List"
Pre: - ***Post:*** *DoubleLinkedList = {Head: null, Tail, null, numElements: 0}*

Add (I,T)
"Adds a T value to the linked list, creating a new DoubleLinkedNode"
Pre: *{I.isEmpty == True} v { I.isEmpty == False}* ***Post:*** *DoubleLinkedNode = < Value: T >*

Delete (T)
"Deletes a T value from the linked list"
Pre: *I.isEmpty == False* ***Post:*** *DoubleLinkedNode.Vakye = null*

Size ( )
"Returns the number of elements of the linked list"
Pre: - ***Post:*** *<numElements>*

IsEmpty ( )
"Returns a Boolean that informs if the list is empty or not"
Pre: - ***Post:*** *TRUE if Head = null, FALSE if head != null*

DoubleLinkedNode <T>

DoubleLinkedNode<T> = *{Value = T, Next* = DoubleLinkedNode<T>, Prev = DoubleLinkedNode<T>}

{inv.: }

PrimitiveOperations:

- ***DoubleLinkedNode:*** *T →DoubleLinkedNode*
- ***GetNext****: DoubleLinkedNode → DoubleLinkedNode*
- ***SetNexr****: DoubleLinkedNode → DoubleLinkedNode*
- ***GetPrev****: DoubleLinkedNode → DoubleLinkedNode*
- ***SetPrev****: DoubleLinkedNode → DoubleLinkedNode*
- ***GetValue****: DoubleLinkedNode → DoubleLinkedNode*

DoubleLinkedNoted(T)

"Creates a new node which contains a T value"

Pre: -
Post: *n = {Value: T, Next: null }*

GetNext (n)

"Returns the next node of a node"

Pre: -
Post: *<n.Next>*

SetNext (n, nN)
"Sets the next node of a node"
Pre: *-* ***Post:*** *<n.Next = nN>*

GetPrev (n)
"Returns the prev node of a node"
Pre: *-* ***Post:*** *<n.Prev>*

SetPrev (n, nN)
"Sets the prev node of a node"
Pre: *-* ***Post:*** *<n.Prev = nN>*

SetPrev (n, nN)
"Returns the T value of the node"
Pre: *-* ***Post:*** *<n.Value>*

1.7.Stack TAD

Stack < T >
Stack <T> *= {Top = <T>, Stack= <DoubleLinkedList<T>>}*
{inv.: }
PrimitiveOperations: • ***Stack***: → *Stack* • ***Push***: *T x Stack* → *Stack* • ***Pop***: *Stack* → *T* • ***Top***: *Stack* → *T* • ***Empty***: *Stack* → *Boolean*

Stack ( )
"Creates a new empty stack"
Pre: - ***Post:*** *Stacks s = {Top = null}*

Push (T,s)
"Add an element T to the stack, becoming the new top"
Pre: - ***Post:*** *{s{T}, s.Top = T}*

Pop (s)
"Returns an remove the top element of the stack"
Pre: *-* ***Post:*** *{s.Top} ^ {s.Delete}*

Top (s)
"Returns the top element of the stack without removing"
Pre: *-* ***Post:*** *s.Top*

Empty(s)
"Returns a Boolean that informs if the stack is empty or not"
Pre: *-* ***Post:*** *{s.IsEmpty}*

1.8. Queue TAD

Queue < T >
Queue<T> = { DoubleLinkedList <T> = <queue>} }
{inv.: }
PrimitiveOperations: • ***Queue***: → *Queue* • ***Enqueue***: *T x Queue* → *Queue* • ***Dequeue***: *Queue* → *T* • ***Peek***: *Queue* → *T* • ***IsEmpty***: *Queue* → *Boolean*

Queue ( )
"Creates an empty queue"
Pre: - ***Post:*** *Queue p = {queue = SingleLinkedList<T>}*

Enqueue (q,T)
"Adds a T value to the queue"
Pre: *{q.IsEmpty = TRUE} v {q.IsEmpty = FALSE}* ***Post:*** *The value is added to the queue*

Dequeue (q)
"Removes the first element and returns it"
Pre: *{q.IsEmpty = FALSE}* ***Post:*** *q.Value, <q.Value = null>*

Peek(q)
"Returns the first element of the queue without removing"
Pre: *{q.IsEmpty = FALSE}* ***Post:*** *q.Value*

IsEmpty (q)
"Returns TRUE if the queue is empty, if not, returns FALSE"
Pre: *Queue q* ***Post:*** *TRUE if q == null, Flase if q != null*

1.9. Priority Queue TAD

PriorityQueue < T >
PeiorityQueue <T> = *{ DoubleLinkedList <T> = <queue>} }*
{inv.: }
PrimitiveOperations: • ***PriorityQueue***: → *PriorityQueue* • ***Enqueue***: *T x PriorityQueue* → *PriorityQueue* • ***Dequeue***: *PriorityQueue* → *T* • ***Peek***:*PriorityQueue* → *T* • ***IsEmpty***: *PriorityQueue* → *Boolean*

PriorityQueue ( )
"Creates an empty priority queue."
Pre: - ***Post:*** *Queue p = {queue = DoubleLinkedList<T>}*

Enqueue (T)
"Adds an element with its priority to the priority queue."
Pre: - ***Post:*** *contains the element with priority item*

Dequeue ()
"Removes and returns the element with the highest priority from the priority queue."
Pre: *<queue> = { IsEmpty = FALSE}* ***Post:*** *no longer contains the element returned*

Peek ()
"Returns the element with the highest priority from the priority queue without removing it."
Pre: *<queue> = { IsEmpty = FALSE}* ***Post:*** *remains unchanged, and the returned element has the highest priority*

Peek ()
"Checks if the priority queue is empty."
Pre: *Queue q* ***Post:*** *TRUE if q == null, Flase if q != null*

1.10. Test Cases

4.1.1. Hash Table Test Cases

Name	*Class*	*Scenario*
Scenario 1	*HashTableTest*	*Create an instance of `HashTable` with a capacity of 10. Then, insert a key-value pair ("key", 1) into the hash table. Verify that the retrieved value for the key "key" is equal to 1.*
Scenario 2	*HashTableTest*	*Create an instance of HashTable with a capacity of 10. Then, insert three key-value pairs ("key1", 1), ("key2", 2), and ("key3", 3) into the hash table. Verify that the retrieved values for keys "key1," "key2," and "key3" are equal to 1, 2, and 3, respectively.*
Scenario 3	*HashTableTest*	*Create an instance of HashTable with a capacity of 10. Then, insert a key-value pair ("key", 1) into the hash table, and remove the key "key" using the remove() operation. Verify that the value associated with the key "key" cannot be retrieved.*
Scenario 4	*HashTableTest*	*Create an instance of HashTable with a capacity of 10. Then, insert a key-value pair ("key", null) into the hash table. Verify that the value retrieved for the key "key" is null.*

Objective of the test: *Verify the correct insertion and retrieval of a key-value pair in the hash table.*

Class	*Scenario method*	*Scenario*	*input values*	*Results*
HashTable	*test_put_and_get*	*Scenario 1*	*- Capacity: 10* - Key-value pair to insert: ("key", 1)	*The retrieved value for the key "key" should be equal to 1.*

Objective of the test: *Verify the correct insertion and retrieval of multiple key-value pairs in the hash table*

Class	*Scenario method*	*Scenario*	*input values*	*Results*
HashTable	*test_put_multiple_and_get*	*Scenario 2*	*Capacity: 10* *Key-value pairs to insert: ("key1", 1), ("key2", 2), ("key3", 3)*	*The retrieved values for keys "key1," "key2," and "key3" should be equal to 1, 2, and 3, respectively.*

Objective of the test: *Verify the proper removal of a key-value pair from the hash table using remove().*

Class	*Scenario method*	*Scenario*	*input values*	*Results*
HashTable	*test_remove*	*Scenario 3*	*Capacity: 10Key-value pair to insert: ("key", 1)*	*The value retrieved for the key "key" should be null after removal.*

Objective of the test: *Verify the proper insertion of a null value into the hash table.*

Class	*Scenario method*	*Scenario*	*input values*	*Results*
HashTable	*test_put_null_value*	*Scenario 4*	*Capacity: 10Key-value pair to insert: ("key", null)*	*The value retrieved for the key "key" should be null.*

4.1.2. DoubleLinkedList Test Cases

Name	*Class*	*Scenario*
Scenario 1	*DoubleLinkedListTest*	*Create an instance of DoubleLinkedList and add three elements ("apple," "banana," "orange") to the list. Then, search for the element "banana" using the search() method. Verify that the value of the found element is equal to "banana."*
Scenario 2	*DoubleLinkedListTest*	*Create an instance of DoubleLinkedList and add three elements ("apple," null, "orange") to the list. Then, search for the null element using the search() method. Verify that the value of the found element is null.*
Scenario 3	*DoubleLinkedListTest*	*Create an instance of an empty DoubleLinkedList. Attempt to delete the last element from the list using the removeLast() method. Verify that the result is null, as there are no elements in the list.*
Scenario 4	*DoubleLinkedListTest*	*Create an instance of DoubleLinkedList and add three elements ("apple," "banana," "orange") to the list. Then, search for the element "grape" using the search() method. Verify that the result is null because "grape" is not present in the list.*

Objective of the test: *Verify that searching for an element in the list using the search() method works correctly.*

Class	*Scenario method*	*Scenario*	*input values*	*Results*
DoubleLinkedList	*test_search_element*	*Scenario 1*	*Elements added to the list: "apple," "banana," "orange"Element to search for: "banana"*	*The value of the found element should be equal to "banana."*

Objective of the test: *Verify that adding a null element to the list using the add() method works correctly.*

Class	*Scenario method*	*Scenario*	*input values*	*Results*
DoubleLinkedList	*test_add_null_element*	*Scenario 2*	*Elements added to the list: "apple," null, "orange"Element to search for: null*	*The value of the found element should be null.*

Objective of the test: *Verify that deleting the last element from an empty list using the removeLast() method works correctly.*

Class	*Scenario method*	*Scenario*	*input values*	*Results*
DoubleLinkedList	*test_delete_last_element_empty_list*	*Scenario 3*	*Empty list*	*The result of the removeLast() operation should be null.*

Objective of the test: *Verify that searching for an element not present in the list using the search() method works correctly.*

Class	*Scenario method*	*Scenario*	*input values*	*Results*
DoubleLinkedList	*test_search_element_not_present*	*Scenario 4*	*Elements added to the list: "apple," "banana," "orange"Element to search for: "grape"*	*The result of the search() operation should be null because "grape" is not present in the list.*

4.1.3. Priority Queue Test Cases

Name	*Class*	*Scenario*
Scenario 1	*PriorityQueueTest*	*Create an instance of PriorityQueue for integers and enqueue the elements 1, 2, and 3 in ascending order. Verify that dequeuing these elements retrieves them in ascending order: 1, 2, 3.*
Scenario 2	*PriorityQueueTest*	*Create an instance of PriorityQueue for integers and enqueue the elements 3, 2, and 1 in descending order. Verify that dequeuing these elements retrieves them in ascending order: 1, 2, 3.*
Scenario 3	*PriorityQueueTest*	*Create an instance of PriorityQueue for integers and enqueue the elements 2, 1, and 3 in random order. Verify that dequeuing these elements retrieves them in ascending order: 1, 2, 3.*
Scenario 4	*PriorityQueueTest*	*Create an instance of PriorityQueue for integers and enqueue a null element. Verify that dequeuing the null element retrieves null.*

Objective of the test: *Verify that enqueuing elements in ascending order maintains the correct order in the priority queue.*				
Class	***Scenario method***	***Scenario***	***input values***	***Results***
PriorityQueue	*test_enqueue_ascending_order*	*Scenario 1*	*Elements enqueued in ascending order: 1, 2, 3*	*Dequeuing the elements should retrieve them in ascending order: 1, 2, 3.*

Objective of the test: *Verify that enqueuing elements in descending order maintains the correct order in the priority queue.*				
Class	***Scenario method***	***Scenario***	***input values***	***Results***
PriorityQueue	*test_enqueue_descending_order*	*Scenario 2*	*Elements enqueued in descending order: 3, 2, 1*	*Dequeuing the elements should retrieve them in ascending order: 1, 2, 3.*

Objective of the test: *Verify that enqueuing elements in random order maintains the correct order in the priority queue.*				
Class	***Scenario method***	***Scenario***	***input values***	***Results***
PriorityQueue	*test_enqueue_random_order*	*Scenario 3*	*Elements enqueued in random order: 2, 1, 3*	*Dequeuing the elements should retrieve them in ascending order: 1, 2, 3.*

Objective of the test: *Verify that enqueuing a null element in the priority queue works correctly.*				
Class	***Scenario method***	***Scenario***	***input values***	***Results***
PriorityQueue	*test_enqueue_null_element*	*Scenario 4*	*Element enqueued: null*	*Dequeuing the null element should retrieve null.*

4.1.4. Queue Test Cases

Name	*Class*	*Scenario*
Scenario 1	*QueueTest*	*Create an instance of Queue for integers and enqueue the element 1, then dequeue it. After that, attempt to dequeue another element, which should be null. Then, enqueue the element 1 again and verify that dequeuing it retrieves the value 1.*
Scenario 2	*QueueTest*	*Create an instance of Queue for integers and enqueue elements 1, 2, and 3. Then, call the peek operation and verify that it returns 1. Call peek again and ensure it still returns 1 without affecting the queue.*
Scenario 3	*QueueTest*	*Create an instance of Queue for integers. Initially, the queue should be empty. Enqueue an element (1), and verify that the queue is no longer empty. Dequeue the element, and confirm that the queue is empty again.*
Scenario 4	*QueueTest*	*Create an instance of Queue for integers and enqueue a single element, which is the integer 1. Then, call the peek operation and verify that it correctly returns 1.*

Objective of the test: *Verify that enqueuing and dequeuing the same element multiple times works correctly.*

Class	*Scenario method*	*Scenario*	*input values*	*Results*
QueueTest	*test_enqueue_dequeue _same_element_multiple_times*	*Scenario 1*	*Element enqueued: 1*	*The first dequeue operation should retrieve 1, and the second dequeue operation should retrieve null. After enqueuing 1 again, dequeuing it should retrieve 1.*

Objective of the test: *Verify that the peek operation returns the first element without removing it from the queue.*

Class	*Scenario method*	*Scenario*	*input values*	*Results*
QueueTest	*test_peek*	*Scenario 2*	*Elements enqueued: 1, 2, 3*	*The first call to peek should return 1, and the second call to peek should also return 1 without affecting the queue.*

Objective of the test: *Verify that the queue correctly reports whether it is empty after enqueuing and dequeuing elements.*

Class	*Scenario method*	*Scenario*	*input values*	*Results*
QueueTest	*test_isEmpty_after_enqueue_dequeue*	*Scenario 3*	*Element enqueued: 1*	*The queue should initially be empty, then not empty after enqueuing 1, and empty again after dequeuing 1.*

Objective of the test: *Verify that the peek operation works correctly when the queue contains only one element.*

Class	*Scenario method*	*Scenario*	*input values*	*Results*
QueueTest	*test_peek_works_with_one_element*	*Scenario 4*	*Element enqueued: 1*	*The call to peek should return 1 when the queue contains only one element.*

4.1.5. Stack Test Cas

Name	*Class*	*Scenario*
Scenario 1	*StackTest*	*Create an instance of Stack for double values, which initially is empty. Attempt to pop an element from the empty stack and verify that it returns null.*
Scenario 2	*StackTest*	*Create an instance of Stack for string values. Push a null element onto the stack and then pop it. Verify that the pop operation returns null.*
Scenario 3	*StackTest*	*Create an instance of Stack for string values. Push the string "test" onto the stack and then call the top() operation. Verify that it correctly returns "test."*
Scenario 4	*StackTest*	*Create an instance of Stack for string values. Push the string "test" onto the stack, then push a null element onto the stack. Call the top() operation and verify that it correctly returns null.*

Objective of the test: *Verify that popping an empty stack returns null.*

Class	*Scenario method*	*Scenario*	*input values*	*Results*
StackTest	*test_pop_empty_stack_returns_null*	*Scenario 1*	*Stack is empty*	*Popping an element from an empty stack should return null.*

Objective of the test: *Verify that pushing a null value onto the stack and then popping it returns null.*

Class	*Scenario method*	*Scenario*	*input values*	*Results*
QueueTest	*test_push_null_and_pop_returns_null*	*Scenario 2*	*Element pushed: null*	*Popping a null element from the stack should return null.*

Objective of the test: *Verify that pushing a value onto the stack and then calling top() returns that value.*

Class	*Scenario method*	*Scenario*	*input values*	*Results*
QueueTest	*test_push_value_then_top_returns_value*	*Scenario 3*	*Element pushed: "test"*	*The call to top() should return "test."*

Objective of the test: *Verify that pushing a string onto the stack, then pushing a null value, and calling top() returns the null value.*

Class	*Scenario method*	*Scenario*	*input values*	*Results*
QueueTest	*test_push_string_then_push_null_then_top_returns_null*	*Scenario 4*	*Elements pushed: "test," null*	*The call to top() should return null after pushing a null element onto the stack.*