DOCUMENTATION DISCRETE STRUCTURES INTEGRATIVE TASK1

Team: Martín Gómez A00399958, Julio Prado A00399637 , Alejandro Mejía A00399937

List of items:

ENGINEERING METHOD

Phase 1- Problem identification:

The problem is about information management. The user needs to save, delete and modify data related to tasks. In addition, he needs to be able to view task information in a specific way, sorted by arrival time or priority level.

The necessity is, indeed, real because all humans have responsibilities. A program that helps to organize obligations, potentially, could enhance productivity and time management.

REQUIREMENT SPECIFICATION

Client: **User:**	**ICESI UNIVERSITY** Students from Icesi university
Functional Requirements	**FR1. Manage tasks** **FR1.1. Add task** **FR1.2. Delete task** **FR1.3. Modify task** **FR2. Display task list** **FR2.1. Display task list sorted by importance level** **FR2.2. Display task list sorted by arrival time** **R3. Undo actions**
Problem Context	The user needs software for task management. Every task has a title, a description and a deadline. The task can have priority or be unprioritized. The user can see his tasks. Unprioritized tasks must be displayed sorted by their arrival time and priority tasks must be displayed sorted by priority level. The client also requires there to be an option for undoing actions.
Non-functional requirements	**NFR1. Easy to use interface on any device** **NFR2. Easy maintenance and updating of the system.** **NFR3. Use of generics**

Name and identifier	*[FR1.1 -Add task]*		
Summary	*When choosing the option to add a task, the system will ask the user to enter a title, a description, a deadline and a boolean that says whether the task has priority or not. If and only if the task has priority, the system will request the user to write a priority level.*		
Inputs	**Input Name**	**Data Type**	**Valid Value Conditions**
	title	String	*It cannot be empty and be equal to the title of a task that was added previously.*
	description	String	*It cannot be empty*
	deadline	GregorianCalendar	*It cannot be empty*
	hasPriority	boolean	
	priorityLevel	int	*It only applies if hasPriority is equal to true. The possible priority levels are:* *1. Low Priority* *2. Priority* *3. high Priority*
Result or postcondition	The task is added to the hashmap in any case. If the task has priority, it will be added to the priority queue. If the task is unprioritized, it will be added to the lifo(stack).		
Outputs	**Output Name**	**Data Type**	**Format**
	msg	String	

Identifier and name	*[FR1.2: Delete task]*		
Summary	*The user may want to delete a task, for doing so the name of the task is requested.*		
Inputs	**Input name**	**Datatype**	**Condition valid values**
	title	String	

Result or postcondition	The task is erased from the system		
Outputs	**Output name**	**Datatype**	**Format**
	Message	String	text String

Name and identifier	*[FR1.3 -Modify task]*		
Summary	*The user is able to modify each one of the fields that a task has.*		
Inputs	**Input Name**	**Data Type**	**Valid Value Conditions**
	fieldToModify	String	*It has to be equal to title, description, deadline, hasPriority or priority level.*
	newFieldValue	String/int/boolean/gregorianCalendar	*If fieldToModify is the title, newFiled must not be empty or match any other title task.*
Result or postcondition	If the user wants the task to be prioritized, the system will request a priority level immediately . Also, the task will be modified according to the prompts.		
Outputs	**Output Name**	**Data Type**	**Format**
	msg	String	

Identifier and name	*[FR2.1: Display task list sorted by importance level]*		
Summary	*The system must display a list of the tasks that have priority. This list must be sorted by importance level.*		
Inputs	**Input name**	**Datatype**	**Condition valid values**
Result or postcondition	The list is displayed.		
Outputs	**Output name**	**Datatype**	**Format**
	Message	String	text String

Identifier and name	*[FR2.1: Display task list sorted by arrival time]*		
Summary	*The system must display a list of the tasks that don't have priority. This list must be sorted by arrival time.*		
Inputs	**Input name**	**Datatype**	**Condition valid values**
Result or postcondition	The list is displayed.		
Outputs	**Output name**	**Datatype**	**Format**
	Message	String	text String

Identifier and name	*[FR3. Undo actions]*		
Summary	*The system must be able to undo the most recent user's action.*		
Inputs	**Input name**	**Datatype**	**Condition valid values**
Result or postcondition	If there is an action in the stack, the system will undo it. Otherwise not.		
Outputs	**Output name**	**Datatype**	**Format**
	Message	String	text String

Phase 2- Compilation of necessary information:

In order to develop this phase, and taking into consideration the required product is a system, we will make use of two different requirement elicitation techniques: 1. Quality Function Deployment and 2. Use Case Approach

REQUIREMENT ELICITATION

1. **Quality Function Deployment**

In this technique customer satisfaction is of prime concern, hence it emphasizes on the requirements which are valuable to the customer. As the identification of stakeholders (in this case just user and client), and the listing of the requirements is already done from the first phase of the engineering method, we just need to classify 3 types of requirements:

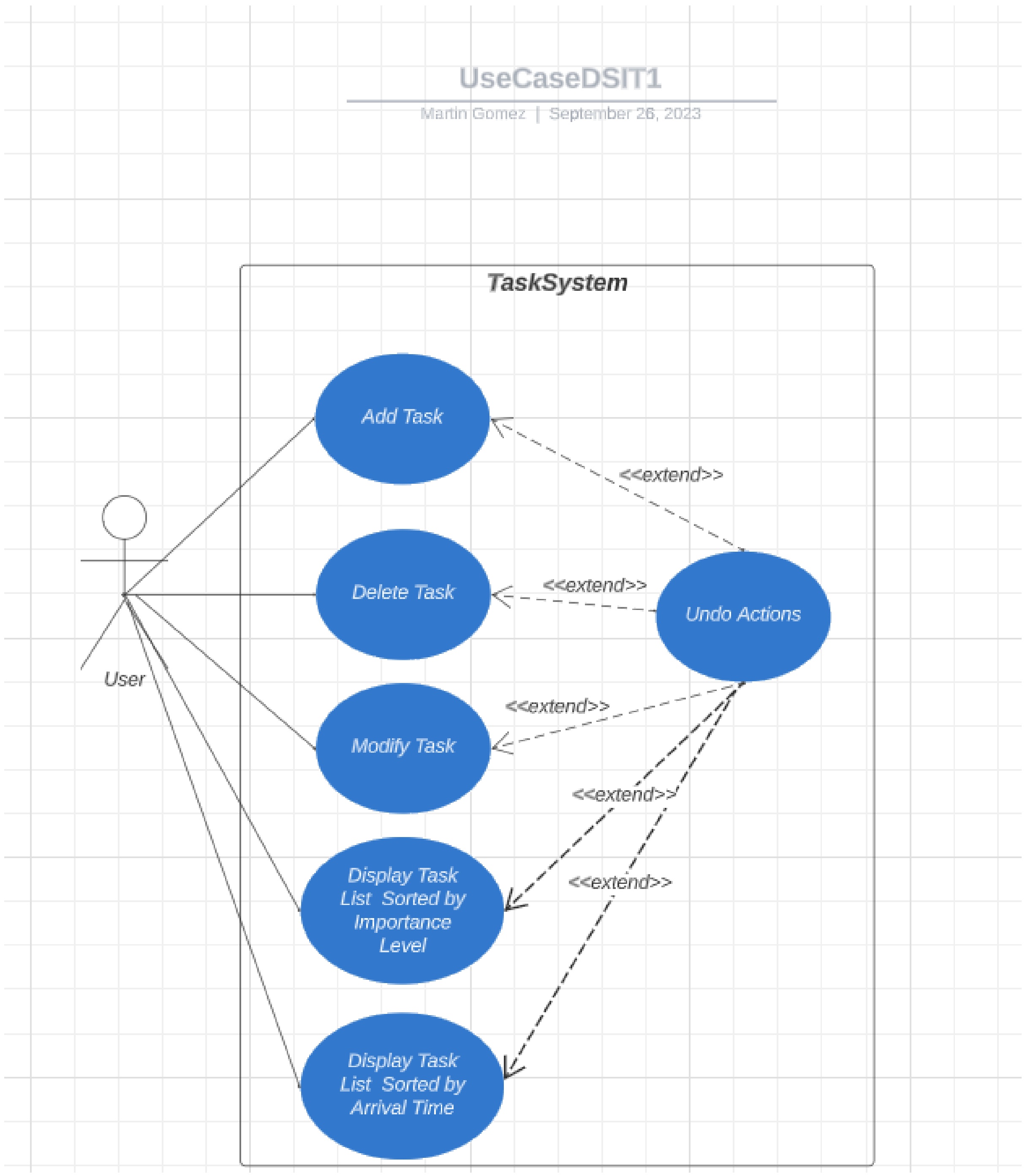
- **Normal requirements :** In this the objective and goals of the proposed software are discussed with the customer.
- **Expected requirements :** These requirements are so obvious that the customer need not explicitly state them.
- **Exciting requirements :** It includes features that are beyond customer's expectations and prove to be very satisfying when present.

Listing of normal requirements: All functional requirements listed in the first phase of the engineering method (manage tasks, display task list, undo actions).

Listing of expected requirements: Non functional requirements listed in the first phase of the engineering method(easy to use interface on any device, easy maintenance and updating of the system, use of generics)

Listing of exciting requirements: The implementation of a more interactive interface than console, like javafx or the development of a front end with the use of html, javaScript, and css.

2. **Use Case Approach**



Phase 3: Creative solution searching

FR1 Add Task, Delete Task and Modify task: In this case, since we are looking for a data structure to store our tasks and reminders, our options are quite diverse. We could use Fixed-Size Arrays, ArrayList, Stacks, Queues, Binary Search Trees, HashTables, linked lists or more. Additionally, all of these options allow us to make modifications or even delete a task, each with its own pros and cons.

FR2.1 Display task list sorted by importance level: Here, we aim to organize tasks based on a level of priority. In this case, various data structures can be used, such as Queues (Non-priority), Stacks, Linked Lists or Double Linked Lists, a priority queue, Fixed-Size Arrays.

FR2.2 Display task list sorted by arrival time: In this case, it will be easier to find a data structure that serves us to sort tasks by arrival order. Here, it is possible to implement Linked Lists, Queues, Stacks, ArrayLists, or Fixed-Size Arrays.

FR3 Undo actions: To undo an action, it would be very beneficial for the data structure we use to be capable of preserving a certain order of modifications made to a file. Among the data structures that could help us are Stacks, Linked Lists, ArrayLists, Fixed-Size Arrays, Binary Search trees.

Phase 4: Transition from idea formulation to preliminary designs.

FR1 Add Task, Delete Task and Modify task:

- **Fixed-Size Arrays**: The Fixed-Size Arrays are memory-efficient as they allocate a fixed amount of memory to store tasks, which can be beneficial if the number of tasks is small or if is defined.
- **ArrayList**: ArrayList is a dynamic data structure that can resize as needed to accommodate new tasks. Also these structures provide fast access to tasks using indexes, making it easy to retrieve and modify specific tasks.
- **Stacks and Queues**: If a specific approach is needed for tasks management, stacks can be useful for an "undo" system, and queues for managing tasks in the order of arrival.
- **Binary Search Tree**: Binary Trees allow efficient searches based on key properties, such as due date or priority making it easy to retrieve and modify specific tasks.
- **HashTable**: They provide fast access to specific tasks through a hash function, which is efficient for searching and modifying tasks by their unique key, such as a unique identifier. A
- **Linked Lists**: Linked lists are flexible in tasks insertions and deletion, which is beneficial when tasks are added and removed frequently.

FR2.1 Display task list sorted by importance level:

- **Queues**: Queues are suitable when we need to manage tasks based on the order they arrived, following a "First In, First Out" (FIFO) approach. This is useful to ensure tasks are handled in the order they were created, regardless of their level of importance.
- **Stacks**: Stacks are ideal when we need a "Last In, First Out" (LIFO) approach. Stacks can be used creatively to manage tasks based on importance levels if we assign a priority value to each task. While they are typically used for "Last In, First Out"

(LIFO) operations, it is possible to implement a custom sorting mechanism to display tasks by importance.
- **Linked Lists or Double Linked Lists**: Linked lists allow flexible task management and sorting based on priorities. Double linked lists enable efficient access both forward and backward in the list. Also here we have the chance to customize task sorting based on their level of importance or priority.
- **Priority Queue**: A priority queue automatically handles task sorting based on their importance. It allows quick access to the most important tasks, facilitating the management of priority tasks.
- **Fixed-Size Arrays**: Fixed-size arrays are memory-efficient and can be useful when the number of tasks is limited and doesn't change frequently.

FR2.2 Display task list sorted by arrival time:
- **Linked List**: Linked lists allow flexible task management and sorting based on arrival time. Double linked lists enable efficient access both forward and backward in the list. Also here we have the chance to customize task sorting based on their arrival time.
- **Queues**: Queues are ideal if we want to display tasks in the order they arrived (First In, First Out), aligning perfectly with the requirement to show tasks by arrival time. Queues are simple and easy-to-understand structures, making them easy to implement and maintain.
- **Stacks**: While not the desired order for this requirement, stacks could be used in reverse order (Last In, First Out) if we want to reverse the task order before displaying them, which is not the case. However, this would require additional data transformation. Stacks are straightforward structures and may be suitable for simple use cases.
- **ArrayLists**: ArrayLists offer fast access through indexes, facilitating the retrieval and sorting of tasks by arrival time. Also we can dynamically resize them based on the number of tasks, which is beneficial if the quantity of tasks varies.
- **Fixed-Size Arrays**: Fixed-size arrays are memory-efficient and provide fast access through indexes, which can be advantageous when the number of tasks is constant, which is not the case.

FR3 Undo actions:
- **Stacks**: Stacks are ideal when we need a “Last In, First Out” (LIFO) approach. This means that the most recent action is the first to be undone. This aligns well with the concept of “undoing” actions in reverse chronological order.
- **Linked Lists**: With the linked list we can maintain a chronological history of actions in the order they occurred. This facilitates reversing actions in the correct order.
- **ArrayLists**: ArrayLists are dynamic data structures that can grow as needed. We can use them to maintain a dynamic history of actions as they occur.
- **Fixed-Size Arrays**: Fixed-size arrays are memory-efficient and can be useful if the history of actions is not expected to be very large.

- **Binary Search Trees**: Binary Search Trees can keep actions sorted based on some key property, such as timestamps. This makes it easy to undo actions based on their chronological order.

Phase 5: Evaluation and best option selection.

FR1 Add Task, Delete Task and Modify task:

- **Fixed-Size Arrays**: It would be useful for a fixed number of tasks where a significant increase in the number of tasks is not expected. Fixed-size arrays are memory-efficient and provide quick access to elements by index. However, they have limitations in terms of scalability if the number of tasks is variable.

- **ArrayList**: It offers the advantage of being a dynamic data structure that can resize as needed to accommodate new tasks. It allows efficient addition, deletion, and modification of tasks. However, the cost of resizing can be a performance drawback when adding a large number of tasks.

- **Stacks and Queues**: They can be useful for a more specific approach to task management. Stacks may be suitable for an "undo" system, where each action is stacked and can be undone in a Last-In, First-Out (LIFO) order. Queues could be useful for managing tasks in a specific arrival order, especially if a First-In, First-Out (FIFO) order is needed.

- **Binary Search Tree**: They would be useful if we want to efficiently organize tasks for searching or accessing based on some key property, such as a due date. Binary search trees can facilitate searching and modifying specific tasks based on their characteristics. But this could be a little bit too much for this case, this would be a complex structure if we constantly want to search for a task.

- **HashTable**: They are suitable if we need quick access to specific tasks using a unique key, such as a unique identifier for each task. Hash tables provide fast access through the hash function, which is efficient for searching and modifying tasks by their unique key. In this case we can agree that is the most proper structure for the tasks storage and searching.

- **Linked Lists**: They are useful when we want a balance between efficiency in task insertion and deletion and the ability to access specific tasks based on their position in the list. Linked lists allow flexible task management and sorting based on priorities or dates.

For this evaluation we are going to have this criterions:

- Efficiency in memory and quick access. (Efficiency)
- Ability to handle a **large** variable number of tasks. (Handle ability)
- Efficiency in the operations of adding, deleting, and modifying tasks. (Efficiency in the operations)

*One "X" is one point

Structure	Efficiency	Handle ability	Efficiency in the operations	Total
Fixed size arrays			X	1
ArrayList		X	X	2
Stacks and Queues		X		1
Binary Search Tree		X	X	2
HashTable	X	X	X	**3**
LinkedList		X		1

For this reason we choose **HashTable** to manage and comply with this requirement.

FR2.1 Display task list sorted by importance level:

- **Queues**: Queues are efficient for maintaining FIFO (First In, First Out) order and can be used to organize tasks based on their arrival order. But they are not efficient for sorting tasks by priority level as they do not provide a natural way to assign priorities to tasks.

- **Stacks**: Stacks are useful for organizing tasks based on arrival order (LIFO - Last In, First Out). They can also be used to implement the "undo" functionality. However, they are not suitable for sorting tasks by priority and do not allow for easy retrieval of the most prioritized tasks.

- **Linked Lists or Double Linked Lists**: Linked lists can maintain a custom order of tasks and are flexible for organizing tasks by priority or other criteria. Insertion and deletion of elements in a linked list can be less efficient than in some other data structures, especially in very long lists.

- **Priority Queue**: This structure is specifically designed to maintain elements in order of priority, directly fulfilling the requirement to organize tasks by importance level.

Additionally, it offers efficiency in the insertion and retrieval of prioritized elements, making it suitable for this task. Although they can be more complex to implement than some other data structures and may require extra effort in terms of maintenance. This is the best choice.

- **Fixed-Size Arrays**: Fixed-size arrays are simple and efficient in terms of space and time if the size is constant. They can be suitable if the maximum number of tasks is known in advance. They are not ideal if the number of tasks can vary widely and do not offer dynamic size management.

  For this evaluation we are going to have this criterions:
    - Ability to maintain an order based on the importance of tasks. (Maintain Order)
    - Ability to handle a **large** variable number of tasks. (Handle ability)
    - Efficiency in adding, deleting, and modifying tasks **based on their importance,** so if the structure isn't capable of maintaining an order based on the importance of tasks this point can't do it. (Efficiency in the operations)

  *One “X” is one point

Structure	Maintain Order	Handle ability	Efficiency in the operations	Total
Fixed size arrays				0
ArrayList		X		2
Stacks		X		1
Queues		X		1
Priority Queue	X	X	X	3
LinkedList or DoubleLinkedList		X		1

For this reason we choose the **Priority Queue** to manage and comply with this requirement.

FR2.2 Display task list sorted by arrival time:

- **Linked List**: Linked lists are flexible and efficient for inserting and deleting elements at any position, making them suitable for maintaining a record of tasks sorted by arrival time. They are easy to implement. Although searching for a specific element by arrival time may require traversing the list from the beginning, which is not as efficient as other structures for this purpose.

- **Queues**: A Queue follows the FIFO (First In, First Out) principle, which means that elements are organized in the order they arrived. This makes it natural and efficient to maintain a chronological record of tasks based on their arrival time. Queues are efficient in inserting elements at the arrival end. When a new task arrives, it is simply enqueued at the end of the queue.

- **Stacks**: Although following the LIFO (Last In, First Out) principle, it could be used to maintain a chronological record. It is efficient for adding tasks at the arrival end and removing them at the exit end. An important cons is that it is not efficient for accessing specific tasks by their arrival time without popping previous elements.
- **ArrayLists**: ArrayLists are efficient for accessing elements by index, making it easy to retrieve specific tasks by arrival time. They can also dynamically grow to accommodate new tasks. Inserting and deleting elements in the middle of an ArrayList can be costly in terms of time and space.

- **Fixed-Size Arrays**: Fixed-size arrays are simple and efficient for accessing elements by index. They can be suitable if the maximum and constant size of the list is known. They cannot accommodate more elements than their fixed size, which could be a limitation if the task list is dynamic.

For this evaluation we are going to have this criterions:

  - Ability to maintain a chronological order based on the arrival time of tasks. (Maintain Order).
  - Ability to handle a **large** variable number of tasks. (Handle ability)
  - Efficiency in adding, deleting, and modifying tasks based on their **arrival time**. (Efficiency in the operations).

*One “X” is one point

Structure	Maintain Order	Handle ability	Efficiency in the operations	Total
Fixed size arrays	X		X	2
ArrayList	X	X	X	3
Stacks		X		1
Queues	X	X	X	3
LinkedList	X	X		2

Here we have a tie between the ArrayList and the Queue, however the queue has a FIFO nature which makes it an optimal option, providing that small advantage for this requirement. For this reason we choose the **Queue** to manage this requirement.

FR3 Undo actions:

- **Stacks**: A Stack follows the Last In, First Out (LIFO) principle, meaning that the last action performed is at the top of the stack and can be efficiently undone. Stacks are easy to implement and use. We can add new actions to the stack as they are performed and undo the last action very simply. With a Stack, we don't need to keep a complete record of all actions, which can be more memory-efficient compared to other data structures.

- **Linked Lists**: Can maintain a complete record of all actions in chronological order. Efficient for undoing actions in order, either moving forward or backward in the history, if we want to include this option. But this structure is not as efficient for undoing the very last action compared to a stack. Requires more memory than a stack or an array to maintain a complete history.

- **ArrayLists**: Efficient for accessing elements at a specific position in the history of modifications. Good performance in both reading and writing. Not as efficient for undoing the very last action compared to a stack. The size of the ArrayList needs to be managed if using a fixed size.

- **Fixed-Size Arrays**: Space-efficient if the maximum number of recorded modifications is known in advance. Constant-time performance in access and modification. Limited by the fixed size, which can be an issue if the history of actions is large. Not as efficient for undoing the very last action compared to a stack.

- **Binary Search Trees**: Can maintain a complete record of all actions in chronological or key-based order. Efficient for searching and undoing specific actions based on their position in the tree. Although this requires more implementation effort compared to the previous structures. Performance may degrade if the tree is unbalanced. It's a complex structure.

  For this evaluation we are going to have this criterions:
  - Ability to undo actions in Last In, First Out (LIFO) order, **and resize** the structure with efficiency to avoid the excess of memory use. This is a really important requirement. (Undo in LIFO order and resize).
  - Efficient use of memory to maintain a **large** history of actions. (Handle ability)
  - Efficiency in accessing past actions. (Efficiency in access)

*One “X” is one point

Structure	Undo in LIFO order and resize	Handle ability	Efficiency in access	Total
Fixed size arrays			X	1
ArrayList	X	X	X	3
Stacks	X	X	X	3
Binary Search Tree		X	X	2
LinkedList		X	X	2

Here we have another tie, in this case it is between the ArrayList and the Stack, however the Stack has a LIFO nature which makes it an optimal option, providing that small advantage for this requirement.
For this reason we choose the **Stack** to manage this requirement.

ANALYSIS

TIME AND SPATIAL COMPLEXITY ANALYSIS OF A COUPLE IMPLEMENTED ALGORITHMS

*In a 64 bits structure

Double Linked List

private <U>Node<T> getFirstNodeWithInstance(U object,BiPredicate<T,U> equals)		**Size in memory**	**Quantity (Size memory)**	**Temporal Complexity**
1	*Node<T> firstNode = null;*	**352 bits**	**1**	**1**
2	*Node <T> current=head;*	**416 bits**	**1**	**1**
3	*boolean flag=false;*	**1 bit**	**1**	**1**
4	*while(current!=null&&!flag){*	**128 bits**	**n+1**	**n+1**
5	*if(equals.test(current.getValue(),object)){*	**321 bit**	**n**	**n**
6	*firstNode= current;*	**64 bits**	**n**	**n**
7	*flag=true;*	**1 bit**	**n**	**n**

8	*}*
	current=current.getNext();
		192 bits	**n**	**n**

9	*}*
	return firstNode;
	}
		64 bits	**1**	**1**

	Total:	**1539 bits**	**5n+5 = O(n)**	**5n + 5 = O(n)**

DoubleLinkedList

public T removeLast(){		**Size in memory**	**Quantity (Size memory)**	**Temporal Complexity**
1	*T valueToDelete;*	**64 bits**	**1**	**1**
2	*if(size==1){*	**32 bits**	**1**	**1**
3	valueToDelete= tail.getValue();	**544 bits**	**1**	**1**
4	*head=null;*	**64 bits**	**1**	**1**

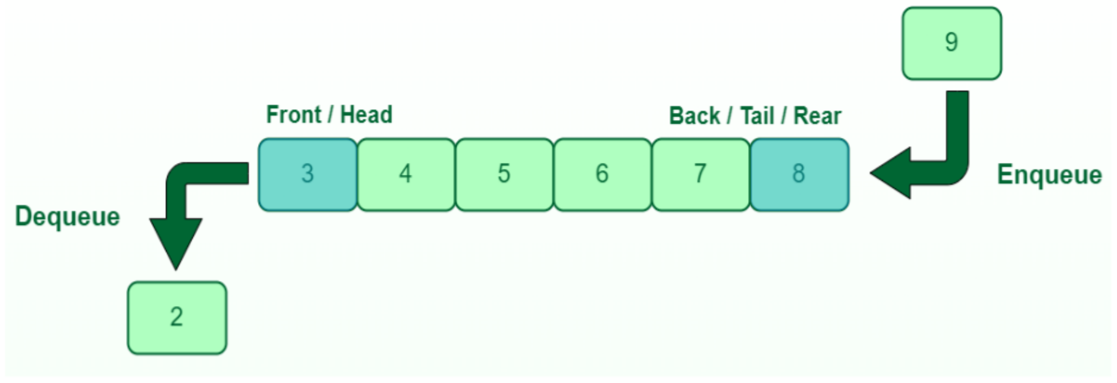
5	tail=null;	**64 bits**	**1**	**1**
6	size--;	**32 bits**	**1**	**1**
7	}else if(size>1){	**32 bits**	**1**	**1**
8	valueToDelete= tail.getValue();	**544 bits**	**1**	**1**
9	tail=tail.getPrev();	**544 bits**	**1**	**1**
10	tail.getNext().setPrev(null);	**608 bits**	**1**	**1**
11	tail.setNext(null);	**480 bits**	**1**	**1**
12	size--;	**32 bits**	**1**	**1**
13	}else{ valueToDelete=null; }	**64 bits**	**1**	**1**

	return valueToDelete;
	
	}
		64 bits	**1**	**1**

	Total:	**3168 bits**	**14 = O(1)**	**14 = O(1)**

DESIGN

TAD OF USED DATA STRUCTURES

Queue			
 9 Front / Head Back / Tail / Rear 3 4 5 6 7 8 Enqueue Dequeue 2			
Attributes $E=\{e_0,e_1,e_2,....e_{size-1}\}$ size			
{inv: e_0 is the only element that can be accessed and deleted}			
Primitive operations			type
create		queue	constructor
add	queue x element	queue	modifier
remove	queue	queue	analyzer
peek	queue	element	analyzer
size	queue	int	analyzer
empty	queue	boolean	analyzer

Create(queue)
“Creates an empty queue”
pre:{true}
{post: E->{ Ø} ∧ size->0 }

Add(queue x e)
“Adds an element to the queue”
pre{ true}
post{ E->{e_0,e_1,e_2,....e} ∧ size->size+1 }

Remove(queue)
“Removes the earliest-added element that belongs to E”
pre{ size - > 0 }
post{(E->Ø ∧ size->0 if size==1 E->{e_1,e_2,e_3,....e_{size-1}} ∧ size->size-1 if size>1 }

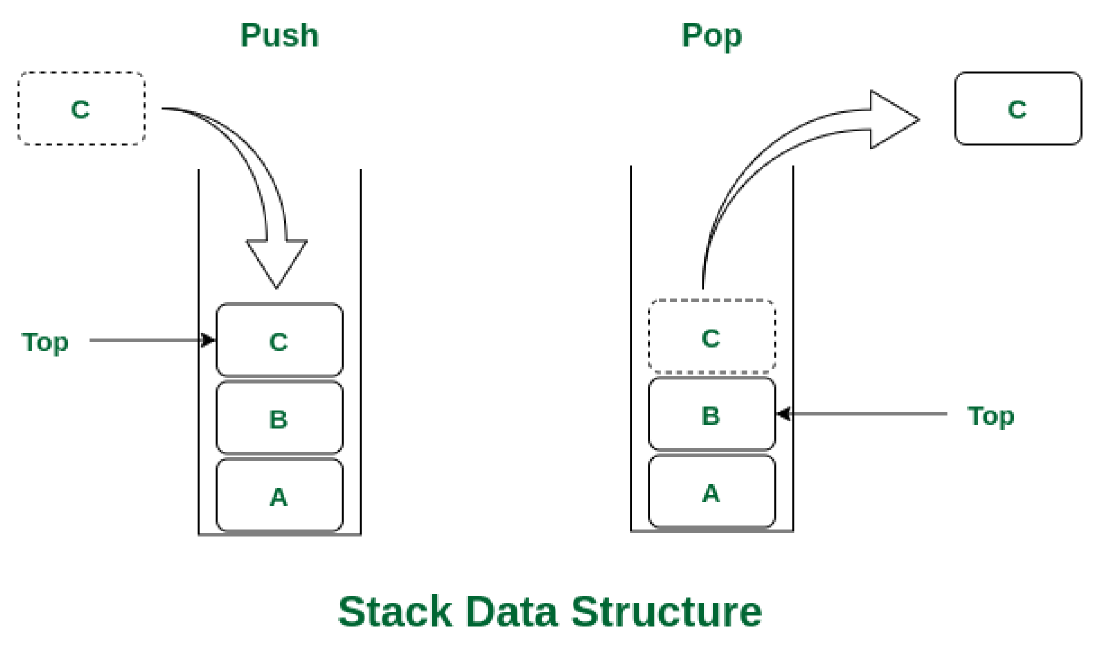
peek():
“Returns the earliest-added element that belongs to E”
pre{ size>0 }

post{ return e_0}

size(queue):
“Returns how many elements are in E”
pre{ true }
post{ return size}

empty(queue):
“Returns whether the queue is empty or not ”
pre{ true }
post{ true if size==0, else false}

Stack



Attributes			
$E=\{e_0,e_1,e_2,\dots.e_{size-1}\}$ size			
{inv: e_{size-1} is the only element that can be accessed and deleted}			
Primitive operations			
create		stack	constructor
add	stack x element	stack	modifier
remove	stack	stack	modifier
top	stack	element	analyzer
size	stack	int	analyzer
empty	stack	boolean	analyzer

Create(stack)
“Creates an empty stack”
pre:{true}
{post: E->{ Ø} ∧ size->0 }

Add(queue x e)
“Adds an element to the stack”
pre{ true}
post{ E->{e_0,e_1,e_2,....e} ∧ size->size+1 }

Remove(stack)
“Removes the latest-added element that belongs to E”
pre{ size - > 0 }
post{(E->Ø ∧ size->0 if size==1 E->{e_0,e_1,e_2,....e_{size-2}} ∧ size->size-1 if size>1 }

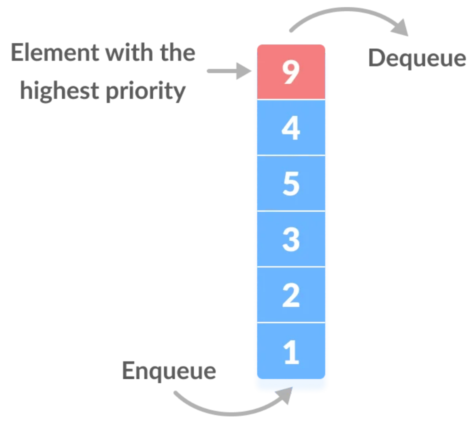
top():
“Returns the latest-added element that belongs to E”
pre{ size>0 }

post{ return e_{size-1}}

size(stack):
“Returns how many elements are in E”
pre{ true }
post{ return size}

empty(stack):
“Returns whether the stack is empty or not ”
pre{ true }
post{ true if size==0, else false}

Priority queue



Attributes
$E=\{e_0,e_1,e_2,\ldots e_{size-1}\}$
size
{inv: $\forall e \in E$ getPriority(e_i)> getPriority(e_{i+1})}

Primitive operations			type
create		priorityQueue	constructor
add	priorityQueue x element	priorityQueue	modifier
remove	priorityQueue	priorityQueue	analyzer
peek	priorityQueue	element	analyzer
size	priorityQueue	int	analyzer
increase Priority	priorityQueue x Element x newPriority		analyzer
empty	priorityQueue	boolean	analyzer

Create(queue)
“Creates an empty priority queue”

pre:{true}
{post: E->{ Ø} $\wedge$ size->0 }

Add(queue x e)
“Adds an element to the priority queue”
pre{ true}
post{ E->{e_0,e_1,...e…e_{2},....} $\wedge$ size->size+1 }

Remove(queue)
“Removes the element with the highest priority that belongs to E”
pre{ size > 0 }

post{(E->Ø $\wedge$ size->0   if size==1
          E->{e_1,e_2,e_3,....e_{size-1}} $\wedge$ size->size-1   if size>1
}

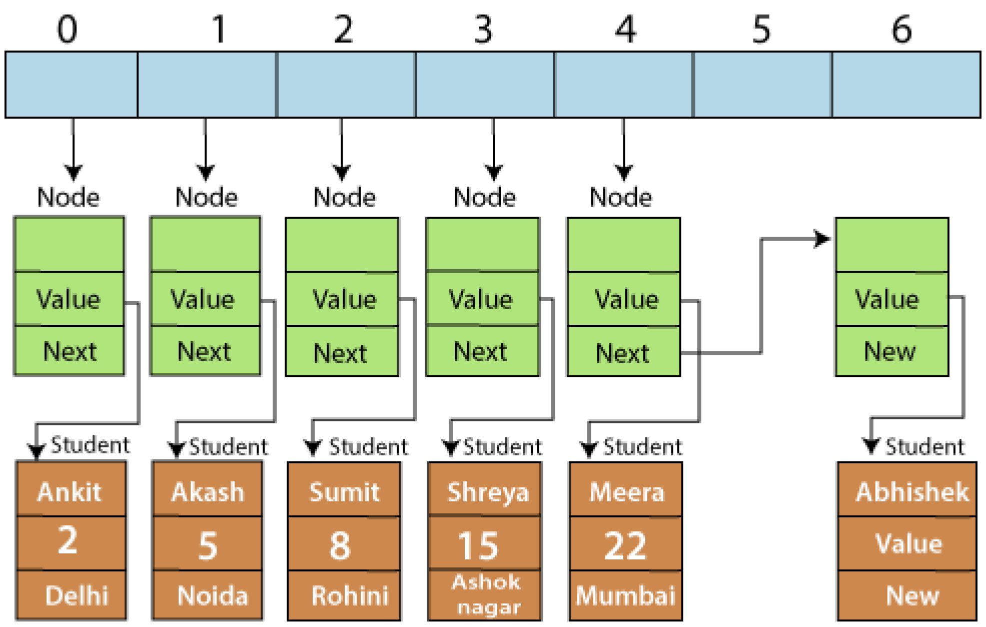
peek():
“Returns the element with the highest priority that belongs to e”
pre{ size>0 }
post{ return e_0}

size(queue):
“Returns how many elements are in E”
pre{ true }
post{ return size}

empty(queue):
“Returns whether the priority queue is empty or not ”
pre{ true }
post{ true if size==0, else false}

increase Priority(queue x element x newPriority):
“Increases the priority of an element”
pre{ e $\in$ E $\wedge$ priority(e)<=newPriority}
post{priority(e)-> newPriority}

HashTable



Attributes

table = {e_1,e_2,e_3,...} (LinkedList)

size

{inv: $\forall$ e $\in$ table e_x.getNext = e_{x+1}}

{inv: $\forall$ e $\in$ table $key_e \in$ e} // The keys are unique for each element e

{inv: $\forall$ e $\in$ table e = null $\lor$ e<- key $\land$ value}

{inv: size $\geq$ 0}

Primitive operations			type
create		HashTable	constructor
put	HashTable x key x value	HashTable	modifier
get	HashTable x key	value	Analyzer
size	HashTable	int	Analyzer
hashFunction	Hastable x key	int	Analyzer
remove	HashTable x key	HashTable	Analyzer

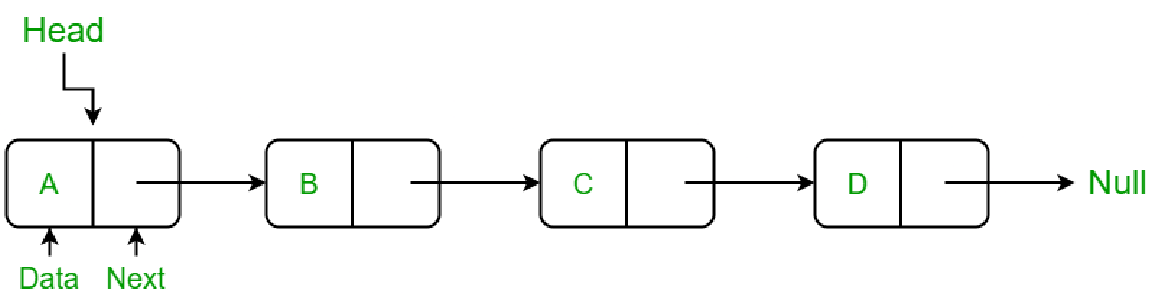
Create(HashTable)
"Creates an empty HashTable"
pre:{true}
{post: table->{ Ø} $\wedge$ size->0 }

put(hashtable x key x value)
“Adds an element to the hashtable”
pre{ true}
post{ E->{e_0,e_1,...e…e_2,....} $\wedge$ size->size+1}

remove (hashTable x key)
"Removes an element from the hashTablewith the specified key"
pre: { (key $\in$ e) $\in$ table}
post: table -> { e0, e1, ..., en } $\wedge$ size -> size - 1

size(hashTable)
"Returns the number of elements in the hashTable"
pre: { true }
post: { result = size }

hashFunction(HashTable x key)
"Transform the key in a valid position of the table"
pre: { true }
post: { index } if the the index isn't valid then returns a -1

List			
			
Attributes E - the type of elements in this list size			
Primitive operations			Type
create		list	Constructor
add	list x element	list	Modifier
remove	list x index	list	Modifier
get	list x index	element	Analyzer

size	list	int	Analyzer
isEmpty	list	boolean	Analyzer
contains	list x element	boolean	Analyzer
indexOf	list x element	int	Analyzer
clear	list	list(empty)	Modifier

Create(list)
"Creates an empty list"
pre:{true}
{post: E->{ Ø} ∧ size->0 }

add(list x e)
“Adds an element to the list”
pre{ true}
post{ E->{e_0,e_1,...e…$e_{2,}$....} ∧ size->size+1}

remove (list x index)
"Removes an element from the list at the specified index"
pre: { 0 <= index < size }
post: E -> { e0, e1, ..., en } ∧ size -> size - 1

get (list x index)

"Retrieves an element from the list at the specified index"
pre: { 0 <= index < size }
post: { result = e }

size(list)
"Returns the number of elements in the list"
pre: { true }
post: { result = size }

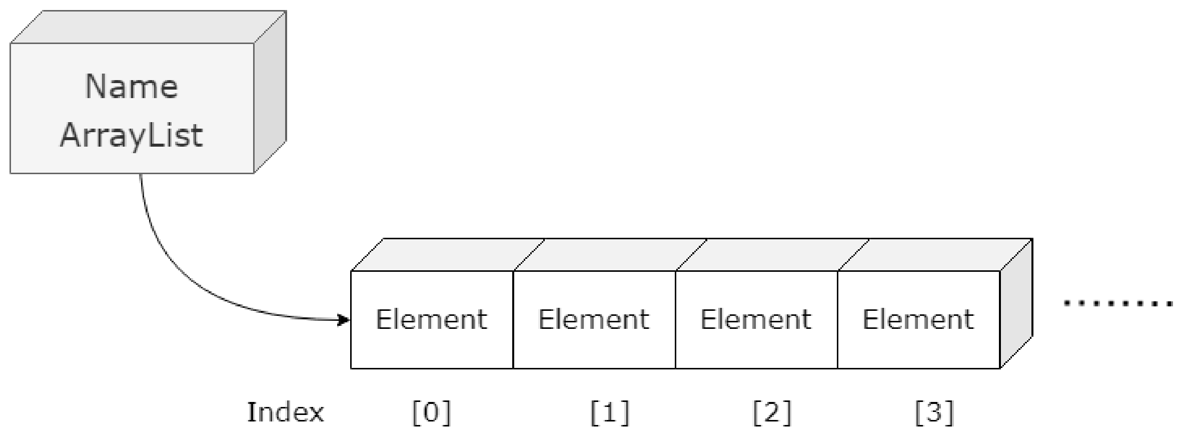
isEmpty(list)
"Checks if the list is empty"
pre: { true }
post: { result = (size == 0) }

contains(list x element)
"Checks if the list contains a specific element"
pre: { true }
post: { result = (element is in E) }

indexOf(list x element)
"Returns the index of the first occurrence of a specific element"
pre: { true }
post: { result = index or -1 if not found }

clear(list)
"Removes all elements from the list"
pre: { true }
post: E -> { Ø } ∧ size -> 0

ArrayList



Attributes

E - the type of elements in this list

size

Primitive operations			Type
create		ArrayList	Constructor
add	Arraylist x element	Arraylist	Modifier
remove	Arraylist x index	Arraylist	Modifier
get	Arraylist x index	element	Analyzer
size	Arraylist	int	Analyzer
isEmpty	Arraylist	boolean	Analyzer
contains	list x element	boolean	Analyzer
indexOf	Arraylist x element	int	Analyzer
clear	Arraylist	Arraylist(empty)	Modifier

Create(Arraylist)
"Creates an empty Arraylist"
pre:{true}
{post: E->{ Ø} ∧ size->0 }

add(Arraylist x e)
“Adds an element to the Arraylist”
pre{ true}
post{ E->{e_0,e_1,...e…e_2,....} ∧ size->size+1}

remove (Arraylist x index)

"Removes an element from the Arraylist at the specified index"
pre: { 0 <= index < size }
post: E -> { e0, e1, ..., en } ∧ size -> size - 1

get (Arraylist x index)
"Retrieves an element from the Arraylist at the specified index"
pre: { 0 <= index < size }
post: { result = e }

size(Arraylist)
"Returns the number of elements in the Arraylist"
pre: { true }
post: { result = size }

isEmpty(Arraylist)
"Checks if the Arraylist is empty"
pre: { true }
post: { result = (size == 0) }

contains(Arraylist x element)

"Checks if the Arraylist contains a specific element"
pre: { true }
post: { result = (element is in E) }

indexOf(list x element)
"Returns the index of the first occurrence of a specific element"
pre: { true }
post: { result = index or -1 if not found }

clear(Arraylist)
"Removes all elements from the Arraylist"
pre: { true }
post: E -> { Ø } ⋀ size -> 0

TEST CASES

Doubly Linked List

Scenery configuration

Name	**Scenery**
setUpEmptyList()	This method initializes an empty doubly linked list of Integers with the name of "list".
setUpListWithElements()	This method calls setUpEmptyList() and then adds elements to the list. In this case, numbers from 0 to 39 are added to the list.

Tests

Test Name	**Used Setups**	**Method Under Test**	**Description**
testGetFirstInstanceNull()	setUpEmptyList()	getFirstInstance()	Tests that the getFirstInstance() method returns null for each element in an empty list.
testGetFirstInstance()	setUpListWithElements()	getFirstInstance()	Tests that the getFirstInstance method correctly returns elements in a list with elements.
testDeleteFirstInstance()	setUpListWithElements()	removeFirstInstance()	Tests that the removeFirstInstance() method correctly removes the first instance of an element and returns **true**. Additionally, verifies that the method returns **false** when attempting to remove the same instance again.

Hash Table

Scenery configuration

Name	Scenery
setUp1EmptyHashTable()	This method initializes an empty hash table with the name of table.
setUp2HashTableWithElements()	This method calls setUp1EmptyHashTable() and then adds elements to the hash table. Task objects with specific titles and descriptions are added. The task titles go from 0 to 999, starting with the common name “title”.

Tests

Test Name	Used Setups	Method Under Test	Description
nullTest()	setUp1EmptyHashTable()	get()	Tests that the get() method returns **null** for each element in an empty hash table.
getTest()	setUp2HashTableWithElements()	get()	Tests that the get() method correctly returns elements in a hash table with elements
removeFalseTest()	setUp1EmptyHashTable()	remove()	Tests that the remove() method returns false for each element in an empty hash table.
removeTrueTest()	setUp2HashTableWithElements()	remove()	Tests that the remove() method returns true the first time attempting to remove an element and false in subsequent attempts.

overwriteElements()	setUp2HashTableWithElements()	add()
		get()
			Tests that the add() method correctly overwrites elements in the hash table.

sizeOverwriteElements()	setUp2HashTableWithElements()	add()
		size()
		get()
			Tests that the size of the hash table does not change after overwriting elements.

sizeTest()	setUp2HashTableWithElements()	remove()
		size()
			Tests that the size of the hash table decreases correctly after removing elements.

Max Heap

Scenery configuration

Name	Scenery
setUpEmptyHeap()	This method initializes an empty Max Heap with the name of “heap”.
setUpHeapWithElements()	This method calls setUpEmptyHeap() and then adds different tasks to the Max Heap (heap).

Tests

Test Name	Used Setups	Method Under Test	Description
testIncreaseKey()	setUpHeapWithElements()	increaseKey()	Tests that the increaseKey() method correctly increments the key of an element in the Max Heap and verifies that the Max Heap property is maintained.
testDecreaseKey()	setUpHeapWithElements()	decreaseKey()	Tests that the decreaseKey() method correctly decrements the key of an element in the Max Heap and verifies that the Max Heap property is

			maintained.
testHeapSort()	setUpHeapWithElements()	extractMax()	Tests that the extractMax() method extracts elements from the Max Heap in descending order.
testInsert()	setUpEmptyHeap()	insert() extractMax()	Tests that the insert() method correctly adds elements to the Max Heap and then extracts the elements in descending order, verifying that the Max Heap property is maintained.
modifyKey()	setUpHeapWithElements()	modifyKey() extractMax()	Tests that the modifyKey() method correctly modifies the key of an element in the Max Heap and verifies that the Max Heap property is maintained.

Queue

Scenery configuration

Name	**Scenery**
setUpEmptyQueue()	This method initializes an empty queue, with the name of "queue", using a doubly linked list implementation (DoublyLinkedList).
setUpQueueWithElements()	This method calls setUpEmptyQueue() and then enqueues elements into the queue. In this case, numbers from 0 to 99 are enqueued.

Tests

Test Name	Used Setups	Method Under Test	Description
testDequeue()	setUpQueueWithElements()	dequeue()	Tests that the dequeue() method correctly removes elements from the queue in the expected order.

testFront()	setUpEmptyQueue()	enqueue()
		front()
			Tests that the enqueue() method adds the element to the queue, and the front() method correctly returns the front of the queue without removing it.

testSize()	setUpQueueWithElements()	dequeue()
		size()
			Tests that the size() method correctly returns the current size of the queue after dequeuing operation.

testEmpty()	setUpQueueWithElements()	empty()
		dequeue()
			Tests that the empty() method returns **true** when the queue is empty and **false** when it has elements.

Stack

Scenery configuration

Name	Scenery
setUpEmptyStack()	This method initializes an empty stack with the name “stack” using a doubly linked list.
setUpStackWithElements()	This method calls setUpEmptyStack() and then inserts elements into the stack. In this case, numbers from 0 to 99 are added to the stack.

Tests

Test Name	**Used Setups**	**Method Under Test**	**Description**
testPop()	setUpStackWithElements()	pop()	Tests that the pop() method correctly returns elements in the reverse order in which they were inserted into the stack.

testTop()	setUpEmptyStack()	top()
		push()
			Tests that the top() method correctly returns the element at the top of the stack without removing it.

testSize()	setUpStackWithElements()	size()
		pop()
			Tests that the size() method correctly returns the size of the stack after performing pop() operations.

testEmpty()	setUpStackWithElements()	empty()	Tests that the empty method returns **true** if the stack is empty and **false** if it has elements.