Daniela Bonilla - A00372534, Tomás Ossa - A00372231, Juan Felipe Sinisterra - A00159604

Data Structures TADs

Algoritmos y Estructuras de Datos

Tarea Integradora 1

Stack TAD

TAD MyStack
MyStack = [Element<T>]
{**inv:** you can just add an Element on the top (LIFO → last-in-first-out)}
Primitive Operations:

Methods	**Operation type**	**Input**	**Output**
MyStack	Constructor		→ MyStack
push	Modifier	Element	→ MyStack
peek	Modifier		→ Element<T>
pop	Modifier		→ Element<T>
isEmpty	Analyzer		→ Boolean
toString	Constructor		→ String

MyStak():
Create an empty Stack
{pre:}
{post: MyStack was created}

push(Element):
Add an element to the top of MyStack
{pre: MyStack must to exist}

{post: A new element was added to MyStack}

peek():
Return the top of the MyStack without deleting it
{pre: MyStack must to exist} {pre: the top must be different from null}
{post: The top of MyStack was returned without deleting it}

pop():
Return and delete the top element of MyStack
{pre:MyStack must to exist} {pre: the top must be different from null}
{post:The top of MyStack was returned and deleted}

isEmpty():
Verify if the MyStack has elements or not.
{pre: MyStack must to exist}
{post: True if top == -1 False otherwise }

toString():
Print the elements of MyStack
{pre: MyStack must to exist} {pre: MyStack can’t be empty}
{post:The elements of MyStack were printed}

Linked List TAD

TAD MyLinkedList
MyLinkedList={Node=<Node>}
{**inv**: MyLinkedList.Node.next >= MyLinkedList.Node.prev ⇔ (only if) MyLinkedList.Node.prev != null }

Primitive operations:
 	
Node = Element of MyLinkedList
MyLinkedList = M.L.L

Métodos	**Tipo Operación**	**Entradas**	**Salida**
createNode	Constructor		→ Node
insertNode	Modifier	Node x Node x Node	
isEmpty	Analyzer		→ Boolean
existingNode	Analyzer	Node x Node	→ Boolean
deleteNode	Modifier	Node	
getNode	Analyzer	Node	→ Node

createNode():
*Create a new Node.
{pre:TRUE}
{post: Node created. If the list M.L.L is empty, then the first Node of the list is n}

insertNode(n, nPrev, nNext):
*Node n is inserted into the linked list so that its next Node (nNext) (if it exists) is strictly greater and its previous element (nPrev) (if it exists) is less or equal. The current node becomes the inserted

node. *
{pre: M.L.L has to be initialized } {pre: n exists (It must be different from null because we need to compare somehow)}
{post: The size of the list M.L.L increases one}

isEmpty():
Verify if the linked list has elements (Nodes) or not.
{pre: Linked list must to exist}
{post: True if M.L.L == null False otherwise }

existingNode(n, actualNode):
* Verify the existence of a Node n in the M.L.L list, the currentNode variable is used to navigate between the Nodes in the list and compare it with n until it is equal *
{pre: Linked list must to exist} {pre: n exists (It must be different from null because we need to compare somehow) }
{post: False if n doesn’t exist in M.L.L True if n exists in M.L.L }

deleteNode(n);
The current item is eliminated from the list as long as it exists. This verification is done with the ExistingNode() method. The Node before the deleted node has the next and previous relationships that the deleted Node had.
{pre: Linked list must to be initialized } {pre: n exists (It must be different from null because we need to compare somehow) } {pre: n exists in the list M.L.L }
{post: Node eliminated}

getNode(n):

* A Node n is obtained from the list as long as it exists. This verification is done with the ExistingNode() method. *
{pre: Linked list must to be initialized }   {pre: n exists (It must be different from null because we need to compare somehow) }   {pre: n exists in the list M.L.L }
{post: Returns the Node to get}

Queue TAD

TAD Queue
Queue=$x\{_n \dots\ x_3,\ x_2,\ x_1\}$
{**inv**: Xn = first element to leave and last element added.}
Primitive operations:

Métodos	**Tipo Operación**	**Entradas**	**Salida**
front	Analyzer		→ Element
rear	Analyzer		→ Element
enqueue	Modifier	Element	
dequeue	Modifier	Element	Element
empty	Analyzer		→ Boolean
size	Analyzer		Integer

front():
*Returns the element at the front of the queue
{pre: queue must not be empty and front element must not be null}
{post: returns the front element}

rear():
*Returns the element at the rear of the queue
{pre: queue must not be empty and rear element must not be null}
{post: returns the rear element}

enqueue():

*Adds an element to the bottom of the queue.
{pre: queue must be created}
{post: queue size increases by one, rear = newElement

dequeue():
*Gets and removes the front element from the queue
{pre: queue must be created}
{post: returns and removes front element. Queue size decreases by one}

empty();
*Verifies if the queue has no elements
{pre: queue must be created}
{post: returns true if the queue has no elements, returns false if the queue has elements}

size():
*Returns the amount of elements in the queue
{pre: queue must be created}
{post: Integer with the total amount of elements in the queue}