TAD- Designs

TAD Stack
Stack = <<e_1, e_2, e_3, …, e >, top>
{inv: 0 <= Stack.size}
Operations:

Stack
“Create a empty Stack<”
Pre: -
post: Stack s = {}

push Stack x Element -> Stack
Adds the new element e to stack s
Preconditions: Stack s = <e_1, e_2, e_3, …, e > and element e or s = ∅ and element e
Postconditions: Stack s = <e_1, e_2, e_3, …, e , e> or s = <e>

pop Stack -> Stack
Destroys stack s freeing memory.
Preconditions: Stack s
Postconditions: -

TAD Queue
Queue = <<e_1, e_2, e_3, …, e , front, back>
{inv: 0 <= Queue.size}
Operations:

Queue
“Create a new queue ”
Pre: -
Post: list: Queue q = ∅

enqueue Queue X Element -> Queue
Inserts a new element e to the back of the queue q
Pre: Queue q = <e_1, e_2, e_3, …, e > and element e or q = ∅ and element e
Post: Queue q = <e_1, e_2, e_3, …, e , e> or q = <e>

dequeue Queue -> Element
Extracts the element in Queue q’s front
Pre: Queue q !=∅ , i.e. q = <e_1, e_2, e_3, …, e >
Post: Queue q = <e_1, e_2, e_3, …, e_{-1}> and Element e_1

front Queue -> Element
Recovers the value of the element on the front of the queue.
Pre: Queue q !=∅ , i.e. q = <e_1, e_2, e_3, …, e >
Postconditions: Element e_1

isEmpty
Determines if the Queue q is empty or not

Pre: Queue q
Post: True if q = ∅, False if q != ∅

~Queue
Destroys queue q freeing memory.
Pre: Queue q
Post: -

TAD HashTable
HashTable = {HashNode<HashNode>, Size = <size>}
{inv:HashTable.size>0 ^ HasTable.HasNode$_1$….HasTable.HasNode$_{size-1}$!=null}
Operations:

HashTable
Creates a new Hash table with n Hash nodes
Pre: -
pos: table: HashTable t = HashNode$_1$, … , HashNode$_{size-1}$

Insert(Key,Value)	Hastable x (Key,Value)	HashTable
Inserts a new value in the hash table given a key, assigned by a hash function and linear		
pre: the key and value must be not null and an index must be given for the hash function		
pos: table: {HashTable.HasNode$_n$.getKey and HashTable.HasNode$_n$.getValue} != null		

Remove(K key)	HashTable x Key	HashTable

Removes a value given a key
pre: the key must belong to the table and must be associated with a value which also belongs to the table
pos: table: HashTable.size= HashTable.size-1

get(key)	HasTable x key	HashNode
Returns an element of the table given its key		
pre: the HasTable must contain at least one key		
pos: HasNode | HashNode.key ∈ Hash Table ∨ HashNode = null		

~HashTable
Destroys the hash table freeing memory.
Pre: HashTable table
Post: -

TAD HashNode
HashNode = < Key<key> , Value<value> , HashNode<Hasnode> >
{inv: Key<key> ∈ Z}
Operations:

HashNode
Creates a new HashNode
Pre: -
post: HashNode node = {node.Key<k>!=null and node.Value <v>!=null}

getKey()	HashNode	Key
Gets the key of the HasNode		
Pre: HashNode node !=null		
Post: Key k != null		

getValue()	HashNode	Value
Gets the value of the HasNode		
Pre: HashNode node !=null		
Post: Value v != null		

setKey(Key)	Key x HashNode	HasNode
Sets a new key for the HasNode		
Pre: HashNode node !=null and Key k != null		
Post: HashNode node.Key != null		

setValue(Value)	Value x HashNode	HashNode
Sets a new value for the HasNode		
Pre: HashNode node !=null and Value v != null		
Post: HashNode node.Value != null		

setNext(HashNode)	Hashnode x HashNode	HashNode
Sets a the next HasNode to the current HasNode		
pre: HashNode current != null and Hashnode next != null		
pos: HashNode current.Next = next		

~HashNode

Destroys the hash node freeing memory.
Pre: HashNode node
Post: -