name: ***Nicoara Stefania-Timeea*** group: ***915***

**DOCUMENTATION**

*Problem statement*

19. Stack (implementation on a singly linked list with dynamic allocation) and Queue (implementation on a doubly linked list on an array)

*Chosen(given) problem*

Red-Back Card Game. Two players each receive n/2 cards, where each card can be red or black. The two players take turns; at every turn the current player puts the card from the upper part of his/her deck on the table. If a player puts a red card on the table, the other player has to take all cards from the table and place them at the bottom of his/her deck. The winner is the player that has all the cards. Simulate the game.

*ADT Stack - specification and interface*

The ADT Stack represents a container in which access to the elements is restricted to one end of the container, called the top of the stack. Because of this restricted access, the stack is said to have a LIFO policy: Last In, First Out.

The domain of the ADT Stack: S = {s|s is a stack with elements of type TElem}

* **init(s)**

Description: creates a new empty stack Pre: true Post: s ∈ S, s is an empty stack

* **destroy(s)**

Description: destroys a stack Pre: s ∈ S Post: s was destroyed

* **push(s, e)**

Description: adds a new element onto the stack Pre: s ∈ S, e is a TElem Post: s'∈ S, s'= s ⊕ e

* **pop(s)**

Description: removes the most recent element from the stack Pre: s ∈ S Post: pop ← e, e is a TElem, e is the most recent element from s, s'∈ S, s'= s \ e Throws: an underflow error if the stack is empty

* **getTop(s)**

Description: returns the most recent element from the stack (doesn't modify it) Pre: s ∈ S Post: top ← e, e is a TElem, e is the most recent element from s Throws: an underflow error if the stack is empty

* **isEmpty(s)**

Description: checks if the stack is empty Pre: s ∈ S Post: isEmpty <- true, if s has no elements | false, if s has elements

*ADT Stack - representation on a singly linked list with dynamic allocation*

**Node:**

elem: TElem next: ↑ Node

**Stack:**

top: ↑ Node

**subalgorithm init (s) is:**

*top <- NIL*

*end-subalgorithm*

**function isEmpty()**

*If top == NIL then*

*isEmpty <- true*

*else*

*isEmpty <- false*

*end-function*

**subalgorithm push(s, e) is:**

*Node\* n <- new Node*

*n.elem <-TElem*

*if isEmpty(s) then*

*n.next <- NIL*

*top <- n*

*else*

*n.next <- top*

*top <- n*

*end-if*

*end-subalgorithm*

**subalgorithm pop(s) is:**

*if isEmpty(s) then*

*@exception: Stack is empty*

*else*

*Node aux*

*aux = top*

*top = top.next*

*delete aux*

*end-if*

*end-subalgorithm*

**function getTop(s) is:**

*getTop <- top.elem*

*end-function*

*ADT Queue - specification and interface*

The ADT Queue represents a container in which access to the elements is restricted to the two ends of the container, called front and rear. Because of this restricted access, the queue is said to have a FIFO policy: First In First Out.

The domain of the ADT Queue: Q = {q|q is a queue with elements of type TElem}

* **init(q, size)**

Description: creates a new empty queue Pre: true Post: q ∈ Q, q is an empty queue

* **destroy(q)**

Description: destroys a queue Pre: q ∈ Q Post: q was destroyed

* **push(q, e)**

Description: adds a new element to the rear of the queue Pre: q ∈ Q, e is a TElem Post: q'∈ Q, q'= q ⊕ e, e is the element at the rear of the queue Throws: an overflow error if the queue is full

* **pop(q)**

Description: removes the element from the front of the queue Pre: q ∈ Q Post: pop ← e, e is a TElem, e is the element from the front of the q, q'∈ Q, q'= q \ e Throws: an underflow error if the queue is empty

* **getFront(q)**

Description: returns the element fom the front of the queue (doesn't modify it) Pre: q ∈ Q Post: top ← e, e is a TElem, e is the element from the front of the q Throws: an underflow error if the queue is empty

* **isEmpty()**

Description: checks if the queue is empty Pre: q ∈ Q Post: isEmpty <- true, if q has no elements | false, if q has elements

* **isFull(q)**

Description: checks if the queue is full Pre: q ∈ Q Post: isFull <- true, if q is full | false, if q is not full

*ADT Queue - representation on a doubly linked list on an array*

**Queue:**

capacity: Integer front: Integer rear: Integer nodes: Node[]

**Node:**

elem: TElem next: Integer prev: Integer

**subalgorithm init(q, size) is:**

*capacity <- size*

*front <- -1*

*rear <- -1*

*firstEmpty <- 0*

*nodes <- new Node[capacity]*

*end-subalgorithm*

**function isEmpty(q) is:**

*if front == -1 then*

*isEmpty <- true*

*else*

*isEmpty <-false*

*end-if*

*end-function*

**function isFull(q) is:**

*if firstEmpty(q) == -1 then*

*isFull <- true*

*else*

*isFull <- false*

*end-if*

*end-function*

**function getFirstEmpty(q) is:**

*for i <- 0, capacity execute*

*if nodes[i].elem == -1 then*

*getFirstEmpty <- i*

*end-if*

*end-for*

*getFirstEmpty <- -1*

*end-function*

**subalgorithm push(q, e) is:**

*if isEmpty(q) then*

*front <- 0*

*rear <- 0*

*firstEmpty <- 1*

*nodes[rear].elem <- e*

*nodes[rear].next <- -1*

*nodes[rear].prev <- rear – 1*

*else*

*rear <- rear + 1*

*nodes[rear].elem <- e*

*nodes[rear].next <- -1*

*nodes[rear -1].next <- rear*

*nodes[rear -1].prev <- rear - 2*

*firstEmpty <- getFirstEmpty(q)*

*if (firstEmpty == -1)*

*firstEmpty <- rear + 1*

*end-if*

*end-subalgorithm*

**subalgorithm pop(q) is:**

*if isEmpty(q) then*

*@exception: Queue is empty*

*else*

*nodes[front].elem <- -1*

*front <- nodes[front].next*

*end-if*

*end-subalgorithm*

**function getFront(q) is:**

*getFront <- nodes[front].elem*

*end-function*

*Tests*

//test Stack

void testIsEmptyStack()

{

Stack s;

assert(s.isEmpty() == true);

int TElem;

s.push(TElem = 0);

assert(s.isEmpty() == false);

}

void testPushStack()

{

Stack s;

int TElem;

s.push(TElem = 0);

assert(s.isEmpty() == false);

}

void testPopStack()

{

Stack s;

int TElem;

s.push(TElem = 0);

s.pop();

assert(s.isEmpty() == true);

}

void testGetTopStack()

{

Stack s;

int TElem;

s.push(TElem = 0);

assert(s.getTop() == 0);

}

//test Queue

void testIsEmptyQueue()

{

Queue q(10);

assert(q.isEmpty() == true);

int TElem;

q.push(TElem = 0);

assert(q.isEmpty() == false);

}

void testIsFullQueue()

{

Queue q(1);

int TElem;

q.push(TElem = 0);

assert(q.isFull() == true);

}

void testGetFirstEmptyQueue()

{

Queue q(10);

int TElem;

q.push(TElem = 0);

q.push(TElem = 0);

assert(q.getFirstEmpty() == -1);

q.pop();

assert(q.getFirstEmpty() == 0);

}

void testPushQueue()

{

Queue q(10);

int TElem;

q.push(TElem = 0);

assert(q.isEmpty() == false);

assert(q.getFront() == 0);

}

void testPopQueue()

{

Queue q(10);

int TElem;

q.push(TElem = 0);

q.pop();

assert(q.isEmpty() == true);

}

void tests()

{

testIsEmptyStack();

testPushStack();

testPopStack();

testGetTopStack();

testIsEmptyQueue();

testIsFullQueue();

testGetFirstEmptyQueue();

testPushQueue();

testPopQueue();

}