ADT Stack - implementation on a doubly linked list on an array

Structure:

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.

When a new element is added, it will automatically be added at the top.
When an element is removed it will be removed automatically from the top. Only the element form the top can be accessed.

Because of this restriction, the stack is said to have a LIFO policy: Last In, First Out.

When a new stack is created, it can have a fixed capacity. If the number of elements in the stack is equal to this capacity, we say that the stack is full.

A stack with no elements is called an empty stack.

Interface:

- 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 $\in$ S, s is an empty stack

- destroy(s)
  - Description: destroys a stack
  - Pre: s $\in$ S
  - Post: s was destroyed

- push(s, e)
  - Description: pushes (adds) a new element onto the stack
  - Pre: s $\in$ S, e is a TElem
  - Post: s' $\in$ S, s' = s $\oplus$ e, e is the most recent element added to the stack
  - Throws: an overflow error if the stack is full

- pop(s)
  - Description: pops (removes) the most recent element from the stack
  - Pre: s $\in$ S
  - Post: pop <- e , e is a TElem, e is the most recent element from s, s' $\in$ S, s' = s $\ominus$ e
  - Throws: an underflow error if the stack is empty

- top(s)
  - Description: returns the most recent element from the stack (but it does not change the stack)
  - Pre: s $\in$ 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 (has no elements)
  - Pre: $s \in S$
  - Post:

    isEmpty <- true, if s has no elements
    false, otherwise

- isFull(s)
  - Description: checks if the stack is full
  - Pre: $s \in S$
  - Post:

    isFull <- true, if s is full
    false, otherwise

Implementation on linked list with dynamic allocation.

Node:
- info : TElem
- next : ↑ Node

Stack:
- top : ↑ Node

Because we use dynamic allocation we don't have a maximum capacity so the isFull function will always return False.

```
function isFull(s) is:
        isFull <- False
end_function
```

ADT Queue - implementation on a doubly linked list on an array.

Structure:

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.

When a new element is added (pushed), it has to be added to the rear of the queue.
When an element is removed (popped), it will be the one at the front of the queue.

Because of this restrictions, the queue is said to have a FIFO policy: First In First Out.

Interface:

The domain of the ADT Queue:

Q = { q | q is a queue with elements of type TElem }

- init(q)
  - 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: pushes (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: pops (removes) the element from the front of the queue
  - Pre: q ∈ Q
  - Post: pop <- e, e is a TElem, e is the element at the front of q, q′ ∈ Q, q′ = q ⊖ e
  - Throws: an underflow error if the queue is empty

- top(q)
  - Description: returns the element from the front of the queue, it does not change the queue
  - Pre: q ∈ Q
  - Post: top <- e, e is a TElem, e is the element from the front of q
  - Throws: an underflow error if the queue is empty

- isEmpty(s)
  - Description: checks if the queue is empty (has no elements)
  - Pre: q ∈ Q
  - Post:

isEmpty <- true, if q has no elements
false, otherwise

- isFull(q)
  - Description: checks if the queue is full
  - Pre: q ∈ Q
  - Post:

isFull <- true, if q is full false, otherwise

Implementation on doubly linked list on an array.

Node:

- info : TElem
- next : Integer
- prev: Integer

Queue:

- nodes : Node []
- front : Integer
- rear : Integer
- cap : Integer
- firstEmpty : Integer

Problem statement:

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. ADTs to be used: Stack (implementation on a singly linked list with dynamic allocation) and Queue (implementation on a doubly linked list on an array).

Implementations for Queue:

```
Function isEmpty ( ) is:
        If (cap = 0 ) then
                isEmpty<- true;
        isEmpty <- false;
        end-function
```

O(1) – complexity

```
Function isFull() is:
        If (cap == 52) then
                isFull<- true;
        isFull<- false;
end-function
```

O(1) – complexity

```
Function enqueue ( TElem a ) is:
        If (cap = 0) then
                Nodes[firstEmpty].info <- a;
                Nodes[firstEmpty].prev <- firstEmpty;
                Front <- firstEmpty;
                Rear <- firstEmpty;
                Cap <- Cap + 1;
                firstEmpty <- getFirstEmpty();
        Else
                Nodes[firstEmpty].info <- a;
                Nodes[rear].next <- firstEmpty;
                Nodes[firstEmpty].prev <- rear;
                Rear <- firstEmpty;
                Cap <- cap + 1;
                firstEmpty <- getFirstEmpty();
        end-if
end-function
```

O(n) – complexity. Because of getFirstEmpty

```
Function Dequeue() is:
        If (cap == 0)
                @throw underflow error
        End-if
        Aux : Integer
        Aux <- front;
        E : Integer
        E <- nodes[front].info;
        Front = nodes[front].next;
        Cap <- cap -1;
        Nodes[aux].prev = -1;
        Nodes[aux].next = -1;
End-function
```

O(1) – complexity

```
Function getFirstEmpty() is:
        For  I <-0, 51 execute
        If (nodes[i].next = -1 and nodes[i].prev = -1) then
                getFirstEmpty <- I;
        End-for
End-function
```

O(n) – complexity . in this case n = 52

```
Function top() is:
        If ( cap = 0 )  then
                @throw underflow error
        End-if
        Top <- nodes[front].info;
        End-function
        Function init() is:
        Front <- -1; rear <- -1 ;cap <- 0;firstEmpty <-0
        init <- this;
End-function;
```

Implementations for Stack:

```
Function isEmpty() is:
        If ( head = NIL ) then
                isEmpty <- true;
        end-if
        isEmpty <- true;
end-function
```

O(1) – complexity;

```
Function push( TElem a ) is:
        N : ↑ Node;
        N <- new Node;
        N->info <- a;

        If ( head != NIL ) then
                n->next = head;

        head <- n;
end-function
```

O(1) – complexity;

```
Function top () is :
        If ( head != NIL ) then
                Top <- head -> info;
        Else
                @Throw underflow exception
End-function
```

O(1) – complexity;

```
Function pop() is:
    If ( head = NIL )
        @throw underflow exception
    End-if
    E : integer;
    E <- head->info;

    Aux : ↑ Node;
    Aux <- head;
    Head <- head->next;

    Free(aux);

    Pop <- e;
End-function

Function init() is:
    Head <- NIL;
    Init <- this;
End-function;

Function destroy() is:
    Aux : ↑ Node;
    Aux2 : ↑ Node;
    Aux <- head;
    While (aux != NIL) then
        Aux2 <- aux->next;
        Free(aux);
        Aux <- aux2;
    End-while
End-function
```

Test functions for Queue:

Q : Queue

Assert Q.isEmpty() == true;
Assert !Q.isEmpty() == false;

Assert Q.getfirstEmpty() == 0;

Q.enqueue(3);

Assert Q.getfirstEmpty() == 1;

Assert Q.isEmpty() == false;
Assert !Q.isEmpty() == true;

Assert Q.top() == 3;
Assert Q.dequeue() == 3;

Assert Q.getfirstEmpty() == 0;

Assert Q.isEmpty() == true;
Assert !Q.isEmpty() == false;

For I <-0 , 51 execute
    q.enqueue(3);

Assert Q.isFull() == true;

Test functions for Stack:

S : Stack

Assert S.isEmpty() == true;
Assert !S.isEmpty() == false;

Assert S.isFull() == false;

S.push(3);

Assert S.isEmpty() == false;
Assert !S.isEmpty() == true;
Assert S.isFull() == false;

Assert S.top() == 3;
Assert S.pop() == 3;

Assert S.isEmpty() == true;
Assert !S.isEmpty() == false;
Assert S.isFull() == false;

Implementation for the problem:

```
Subalgorithm populate(Queue player1, Queue player2, int red, int black) is:
    randomNumber : Integer
    p1 : Integer
    p2 : Integer
    p1 <- red;
    p2 <- black;
    while( p1 != 0 ) execute
        @randomNumber <- random number generated between 1 or 2
        If (randomNumber = 1) then
            If( black != 0 ) then
                Black <- black – 1 ;
                Player1.enqueue(randomNumber);
                P1 <- p1 – 1;
            End-if
        Else
            If( red != 0 ) then
                Red <- red -1;
                Player1.enqueue(randomNumber);
                P1 <- p1 -1;
            End-if
        End-if
    End-while
        while( p2 != 0 ) execute
        @randomNumber <- random number generated between 1 or 2
        If (randomNumber = 1) then
            If( black != 0 ) then
                Black <- black – 1 ;
                Player2.enqueue(randomNumber);
                P2 <- p2 – 1;
            End-if
        Else
            If( red != 0 ) then
                Red <- red -1;
                Player2.enqueue(randomNumber);
                P2 <- p2 -1;
            End-if
        End-if
    End-while

End-subalgorithm
```

```
Subalgorithm simulate( Queue player1, Queue player2, Stack s ) is:
        Turn : Integer
        Turn <- 1;
        Card : Integer
        While(true) execute:
                @print turn
                If( player1.isEmpty() and player2.isEmpty() )
                        Simulate <- 3;
                End-if
                If(!player1.isEmpty()) then
                        Card <- player1.dequeue();
                        s.push(card);

                        if(card == 1 ) then
                                @print black
                        Else
                                @print red
                        End-if

                        If(s.top() == 2) then
                                While(!s.isEmpty()) execute
                                        Card <- s.pop()
                                        Player2.enqueue(card)
                                End-while
                        End-if
                        If(player1.isEmpty() && s.isEmpty())
                                Simulate <- 2;
                        End-if
                End-if
                If(!player2.isEmpty()) then
                        Card <- player2.dequeue();
                        s.push(card);

                        if(card == 1 ) then
                                @print black
                        Else
                                @print red
                        End-if

                        If(s.top() == 2) then
                                While(!s.isEmpty()) execute
                                        Card <- s.pop()
                                        Player1.enqueue(card)
```

```
                End-while
            End-if
            If(player2.isEmpty() && s.isEmpty())
                Simulate <- 1;
            End-if
        End-if
    End-while
End-subalgorithm
```