DEQUE: Fila dupla

- inserir tanto pela frente (`front`) como pelo fim da fila (`back`)
- inserção: `push_back(x)`, `push_front(x)`
- remoção: `pop_back(x)`,`pop_front(x)`
- inspeção: `back(x)`,`front(x)`
- testes: `isEmpty()`,`isFull()`

Pretende-se que dê erro caso alguma operação não possa ser executada. Isto só pode ser feito Dafny que verifica todas as chamadas (com assert)

Implementação

- Lista circular `list` implementada num array. Um indicador `start` (inicio) e o tamanho da fila `size` tal que:
- front:`list[start]`
- back: `list[(start+size-1) mod list.Length]`

```
var q := new Deque(3);
assert q.isEmpty();
q.push_front(1);
assert q.front() == 1;
assert q.back() == 1;
q.push_front(2);
assert q.front() == 2;
assert q.back() == 1;
q.push_back(3);
assert q.front() == 2;
assert q.back() == 3;
assert q.isFull();
q.pop_back();
assert q.front() == 2;
assert q.back() == 1;
q.pop_front();
assert q.front() == 1;
assert q.back() == 1;
q.pop_front();
assert q.isEmpty();

class {:autocontracts} Deque {
    const list: array<T>;
```

```
    var start : nat;
    var size : nat;

   constructor (capacity: nat) {
       list := new T[capacity];
       start := 0;
       size := 0;
   }

   predicate method isEmpty() {
       size == 0
   }

   predicate method isFull() {
       size == list.Length
   }



   function method back() : T {
       list[(start + size - 1) % list.Length]
   }

   method push_back(x : T) {
       list[(start + size) % list.Length] := x;
       size := size + 1;
   }

   method pop_back() {
       size := size - 1;
   }



function method front() : T {
       list[start] }
method push_front(x : T) {
       if start > 0 {
           start := start - 1;
       }
       else {
           start := list.Length - 1;
       }
       list[start] := x;
       size := size + 1;
   }
```

```
method pop_front() {
       if start + 1 < list.Length {
           start := start + 1; }
       else {
           start := 0;}
       size := size - 1;
   }}
```

Contratos

- adicionamos `ghost` para descrever a Deque de forma abstracta:

```
ghost var elems: seq<T>;  // front at head,
                          // back at tail
ghost const capacity: nat;
```

- Invariante de classe (predicate $Valid()$) garante:
  - consistência das variáveis concretas
  - consistência das variáveis concretas e as abstractas

Valid()

```
predicate Valid()
{ 0 <= size <= list.Length  && 0 <= start < list.Length
  && capacity == list.Length
  && elems == if start + size <= list.Length
  then list[start..start + size]
  else list[start..] + list[..size-(list.Length-start)]
   }
```

Pré e Pós condições

```
constructor (capacity: nat)
 requires capacity > 0
 ensures elems == [] && this.capacity == capacity
{
   list := new T[capacity];
   start := 0;
   size := 0;
   this.capacity := capacity;
    elems := []; }
```

Pré e Pós condições

```
predicate method isEmpty()
      ensures isEmpty() <==> elems == []
    {
        size == 0
    }

    predicate method isFull()
      ensures isFull() <==> |elems| == capacity
    {
        size == list.Length
    }
```

Pré e Pós condições: back

```
 function method back() : T
      requires !isEmpty()
      ensures back() == elems[|elems| - 1]
    {
        list[(start + size - 1) % list.Length]
    }

    method push_back(x : T)
      requires !isFull()
      ensures elems == old(elems) + [x]
    {
        list[(start + size) % list.Length] := x;
        size := size + 1;
        elems := elems + [x];
    }
```

Pré e Pós condições:back

```
    method pop_back()
      requires !isEmpty()
      ensures elems == old(elems[..|elems|-1])
    {
        size := size - 1;
        elems := elems[..|elems|-1];
    }
```

Pré e Pós condições:front

```
function method front() : T
  requires !isEmpty()
  ensures front() == elems[0]
    {
        list[start]
    }
```

Pré e Pós condições:front

```
method push_front(x : T)
      requires !isFull()
      ensures elems == [x] + old(elems)
    {
        if start > 0
        {
            start := start - 1;
        }
        else
        {
            start := list.Length - 1;
        }
        list[start] := x;
        size := size + 1;
        elems := [x] + elems;
    }
```

Pré e Pós condições:front

```
    method pop_front()
      requires !isEmpty()
      ensures elems == old(elems[1..])
    {
        if start + 1 < list.Length
        {
            start := start + 1;
        }
        else
        {
            start := 0;
        }
        size := size - 1;
        elems := elems[1..];
    }
}
```