Contents

The Queue collections

Introduction

Abstract Data Type

Described abstractly, a queue is

- a finite collection of elements,
- in a particular order,
- that may contain the same element multiple times.

The fact that it may contain the same element multiple times makes it different from a set, the fact that it is ordered makes it different from a multiset.

Generally, it has operations to…

- … create an empty queue,
- … test for emptiness,
- … obtain the value of the element at "the end" of the queue ("peek"),
- … add an element at "the beginning" of the queue ("enqueue"),
- … remove an element at "the end" of the queue ("dequeue").

The fact that only the "last element" can be removed and that elements can only be added "at the front" is the main difference with the list abstract data type. Exactly like a people waiting in line for a service, queues implement a " first-in-first-out" (FIFO) principle.

Difference with array

Like lists, queues serve a similar purpose than arrays, but with a few notable differences:

- Queues do not need to have a number of elements fixed ahead of time,

- Queues automatically expand when elements are added,
- Queues automatically shrink when elements are removed.

Difference with lists

However, queues have differences with lists:

- Only the "last" element's value can be read (accessed),
- Adding an element can only be done "on the left side", that is, at the beginning of the queue.

Possible Implementation

Using Cells

One could implement queue by adapting our `CList` class, making sure that the read, remove and insert operations are limited to the "valid" elements.

Using Arrays

One could implement queue by using arrays, adding elements "at the end" of the array and removing them "from the beginning". This implementation has one major drawback, however: in this implementation, enqueue operation is $O(c)$, but dequeue is $O(n)$, since one need to "shift" all the elements by one after the first one has been removed.

An alternative is to use a *circular array*, where the beginning and the end of the array are "glued together". This requires to manipulate three `int` variables:

- `front`, pointing to the first element,
- `end`, pointing to the last element,
- and `size`, the number of elements in the queue.

An attribute `mArray` then contain the elements in the queue, *not necessarily in the "right" order*, since as the queue is dequeued, its "front" moves.

```
using System; // This is required for the exception.

class CQueue<T>
{
  private int front,
    end,
    size;
  private T[] mArray;
```

```
public CQueue(int capacity = 10)
{
  mArray = new T[capacity];
  // Note that front, end and size
  // are set to 0.
}

public void Clear()
{
  front = 0;
  end = 0;
  size = 0;
}

public bool IsEmpty()
{
  return size == 0;
}

public bool IsFull()
{
  return size == mArray.Length;
}

public void Enqueue(T newItem)
{
  if (!IsFull())
  {
    mArray[end] = newItem;
    Increment(ref end);
    size++;
  }
  else
    Resize();
}

public T Dequeue()
{
  size--;
  T data = mArray[front];
  Increment(ref front);
  return data;
}

public void Resize()
{
```

```
    throw new Exception(
      "Queue Resize is not implemented - you could
      ↪  implement it"
    );
  }

  // Increment must be done carefully:
  // what if we reached the "end of mArray"
  // but there is room available at the
  // beginning of the queue? Then the value
  // needs to become 0.
  private void Increment(ref int value)
  {
    value++;
    if (value == mArray.Length)
    {
      value = 0;
    }
  }

  public int Capacity
  {
    get { return mArray.Length - size; }
  }

  public override string ToString()
  {
    string returned = "";
    returned +=
      $"Front : {front}, end : {end}, size : {size},
↪   capacity: {Capacity}\n";
    for (int i = front; i < size + front; i++)
    {
      returned += mArray[i % mArray.Length] + ":";
    }
    return returned;
  }
}
```

(Download this code)

An implementation using `capacity` instead of `end` is also possible, but it requires to use the modulo operation (`%`) to compute the `end`, using

```
end = (front + size) % capacity;
```

Our implementation can recover the capacity of the queue by using:

```
capacity = mArray.Length - size;
```

.