**QUEUE: ADT and CDT**

***Recap and Scaffolding***

**Consider the following definition of a queue:**

**Abstract Data Type** **Queue**

* Data : set of ordered elements. The order is established by the sequence of entry in a FIFO mode.
* Operations : Elements enter and exit the queue following the FIFO mode.

**The ADT Queue can be associated with many different CDTs:**

**Remember:** the CDT is the Concrete Data Type, i.e. the implementation of the ADT, consisting of:

* the **data structure** used to contain the ADT data
* the **procedures** implementing the ADT operations using the identified data structure

Here we examine two possible CDTs, both using an array.

In both cases, enqueue in a full array requires RESIZING of the array.

Resizing is most efficiently performed doubling the actual size.

**CDT1. Concrete Data Type Queue - Basic**

* Data : array of elements
* Operations:
  + **enqueue: use an index TAIL, pointing to the last entered element.**
  + **dequeue: always at index 0.**

Drawback of this CDT: left shift at each dequeue.

When the queue has many elements this is quite an expensive operation.

**Example:**

Queue - initially empty

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |

After enqueue of elements 1..5

Queue

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |

HEAD TAIL

After a dequeue (always followed by a left shift)

Queue

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 3 | 4 | 5 |  |

HEAD TAIL

A subsequent enqueue of element 6 will result

Queue

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 3 | 4 | 5 | 6 |

HEAD TAIL

CDT1 is easy to interpret and manage, but not efficient.

**CDT2. Concrete Data Type Queue - Powered**

* Data : array of elements
* Operations:
  + **enqueue: use an index TAIL, pointing to the last entered element.**
  + **dequeue: use an index HEAD, pointing to the first entered element.**

The power of the two indexes: no need to shift.

TAIL and HEAD go around the array from the beginning to the end and back to the beginning in order to use all possible spaces

**Example:**

Queue - initially empty

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |

After enqueue of elements 1..5

Queue

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |

HEAD TAIL

After a dequeue

Queue

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 2 | 3 | 4 | 5 |

HEAD TAIL

A subsequent enqueue of element 6 will result

Queue

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | 2 | 3 | 4 | 5 |

TAIL HEAD

After 4 dequeues

Queue

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 |  |  |  |  |

TAIL HEAD

CDT2 is less intuitive and more difficult to manage, but it is more efficient (no shift at dequeue, only update of index HEAD).

**Let’s reflect on the impact of the resizing needed when enqueueing in a full array.**

**Simulate the enqueue of element 7 and represent the resulting situation for both cases below:**

**CDT1. Concrete Data Type Queue - Basic**

Initial Queue

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 3 | 4 | 5 | 6 |

HEAD TAIL

→ enqueue 7

**Resulting queue?**

**CDT2. Concrete Data Type Queue - Powered**

Initial Queue

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | 2 | 3 | 4 | 5 |

TAIL HEAD

→ enqueue 7

**Resulting queue?**

**ANSWER KEY**

**CDT1. Concrete Data Type Queue - Basic**

Resulting Queue

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 3 | 4 | 5 | 6 | 7 |  |  |  |  |

HEAD TAIL

**CDT2. Concrete Data Type Queue - Powered**

Resulting Queue

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 6 | 7 |  |  |  |  | 2 | 3 | 4 | 5 |

TAIL HEAD

With regard to resizing, CDT2 is not more efficient.