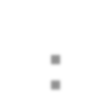
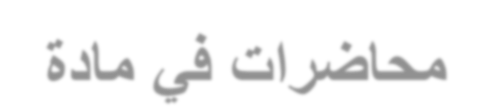
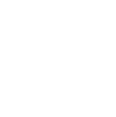
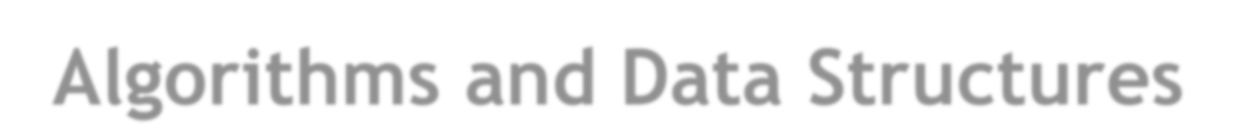
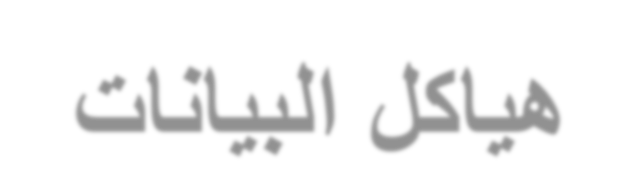
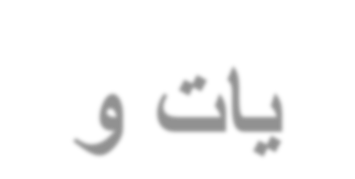
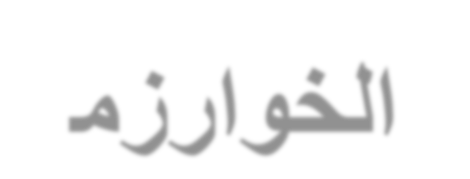
136



**Lecture # 6**

Queues

Outline

• Queue ADT

• Basic operations of queue

 Enqueue, dequeue, print, etc.

• Implementation of queue

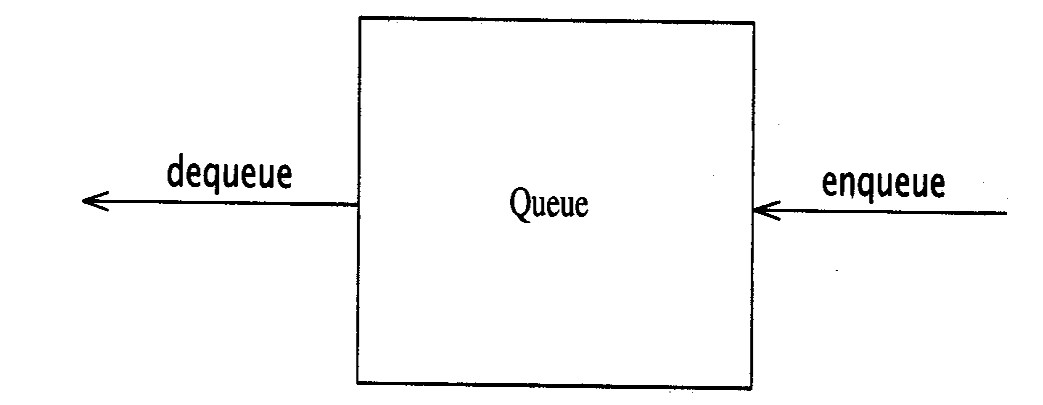
 Array

 Linked list

Queue ADT

• A ***queue*** is an abstract data type in which elements are added to

the ***rear*** and removed from the ***front***.



First In First Out (FIFO)

• Queues are known as FIFO (First In, First Out) lists.

 Like customers standing in a check-out line in a store, the first customer in is the first customer served.

A rear front

A front

A front



|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | D | rear | D | rear |
| B | rear | C  B | rear | C  B |  | C  B | front |

A front A

Enqueue and Dequeue

• Primary queue operations: Enqueue and Dequeue.

• Like check-out lines in a store, a queue has a front and a rear.

• **Enqueue** – insert an element at the rear of the queue.

• **Dequeue** – remove an element from the front of the queue.

Remove

(Dequeue)

front

rear

Insert

(Enqueue)

Implementation of Queue

• Just as stacks can be implemented as arrays or linked lists, so with queues.

• Dynamic queues have the same advantages over static queues as dynamic stacks have over static stacks.

• Let’s see how to use an array to implement a queue.

Queue Implementation of Array

• When an item is enqueued, the rear index moves forward.

• When an item is dequeued, the front index also moves forward by one element.

front rear

XXXXOOOOO

OXXXXOOOO (after 1 dequeue, and 1 enqueue) OOXXXXXOO (after another dequeue, and 2 enqueues) OOOOXXXXX (after 2 more dequeues, and 2 enqueues)

front rear

The problem here is that the rear index cannot move beyond the last element in the array.



Implementation using Circular Array

• Using a circular array.

• When an element moves past the end of a circular array, it wraps around to the beginning, e.g.

 After Enqueue(4):

OOOOO7963  4OOOO7963

 After Enqueue(4), the rear index moves from 3 to 4.

• How to detect an empty or full queue, using a circular array algorithm?

 Use a counter of the number of elements in the queue.

**Queue** Class

**class Queue**

**{**

**public:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Queue(int size = 10);** |  | **//** | **constructor** |
| **~Queue() { delete [] values;** | **}** | **//** | **destructor** |

**bool IsEmpty(void);**

**bool IsFull(void);**

**bool Enqueue(double x); bool Dequeue(double & x); void DisplayQueue(void);**

**private:**

**int front; // front index int rear; // rear index**

**int counter; // number of elements int maxSize; // size of array queue**

**double\* values; // element array**

**};**

• Attributes of **Queue**

 **front/rear**: front/rear index.

 **counter**: number of elements in the queue.

 **maxSize**: capacity of the queue.

 **values**: point to an array which stores elements of the queue.

• Operations of **Queue**

 **IsEmpty**: return true if queue is empty, return false otherwise.

 **IsFull**: return true if queue is full, return false otherwise.

 **Enqueue**: add an element to the rear of queue.

 **Dequeue**: delete the element at the front of queue.

 **DisplayQueue**: print all the data in the queue.

Create Queue

• **Queue(int size = 10)**

 Allocate a queue array of **size**. By default, **size = 10**.

 **maxSize** is set to **size**.

 **front** is set to **0**, pointing to the first element of the array.

 **rear** is set to **-1**. The queue is empty initially.

 **counter** is set to **0**.

**Queue::Queue(int size /\* = 10 \*/)**

**{**

**values = new double[size];**

**maxSize = size;**

**front = 0; rear = -1; counter = 0;**

**}**

**IsEmpty** and **IsFull**

• Since we keep track of the number of elements that are actually in

the queue: **counter**, it is easy to check if the queue is empty or full.

**bool Queue::IsEmpty()**

**{**

**if (counter == 0) return true;**

**else return false;**

**}**

**bool Queue::IsFull()**

**{**

**if (counter < maxSize) return false;**

**else return true;**

**}**

**Enqueue**

**bool Queue::Enqueue(double x)**

**{**

**if (IsFull()) {**

**cout << "Error: the queue is full." << endl;**

**return false;**

**}**

**else {**

**}**

**}**

**// calculate the new rear position (circular)**

**rear = (rear + 1) % maxSize;**

**// insert new item values[rear] = x;**

**// update counter counter++;**

**return true;**

**Dequeue**

**bool Queue::Dequeue(double & x)**

**{**

**if (IsEmpty()) {**

**cout << "Error: the queue is empty." << endl;**

**return false;**

**}**

**else {**

**}**

**}**

**// retrieve the front item x = values[front];**

**// move front**

**front = (front + 1) % maxSize;**

**// update counter counter--;**

**return true;**

Printing the elements

**void Queue::DisplayQueue()**

**{**

**cout << "front -->";**

**for (int i = 0; i < counter; i++)**

**{**

**if (i == 0) cout << "\t";**

**else cout << "\t\t";**

**cout << values[(front + i) % maxSize];**

**if (i != counter - 1)**

**cout << endl;**

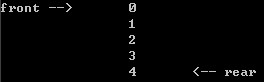
**else**

**}**

**}**

**cout << "\t<-- rear" << endl;**

Result

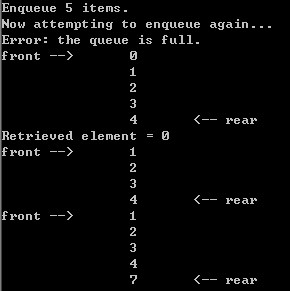


**Implementation of Queue – Using Queue**

Using **Queue**

Result

152



**int main(void) {**

**Queue queue(5);**

**cout << "Enqueue 5 items." << endl;**

**for (int x = 0; x < 5; x++)**

**queue.Enqueue(x);**

**cout << "Now attempting to enqueue again..." << endl;**

**queue.Enqueue(9); queue.DisplayQueue(); double value; queue.Dequeue(value);**

**cout << "Retrieved element = " << value << endl;**

**queue.DisplayQueue(); queue.Enqueue(7); queue.DisplayQueue();**

**return 0;**

**}**