FAIRMONT STATE UNIVERSITY

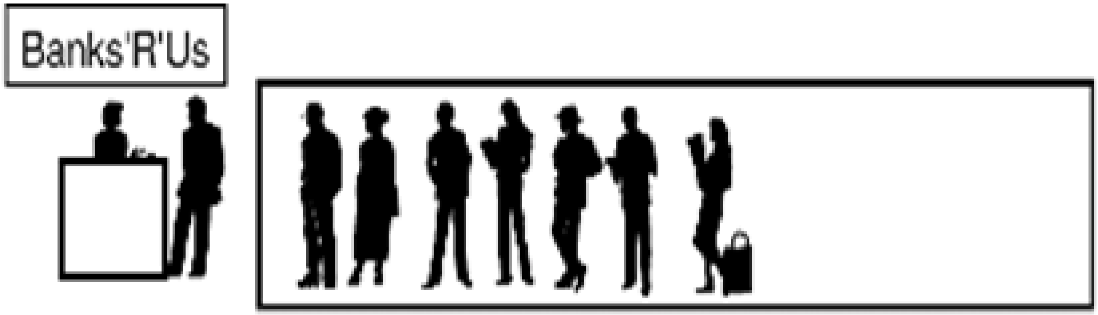
Queues

COMP 2270 – Data Structures

Fall 2014

Dr. Mahmood Hossain

Example



What is a Queue?

- A *queue* is a linear data structure
  - it is a list of homogenous items
  - addition of items can take place only at one end, called the *rear* of the queue
  - deletion of items can take place only at the other end, called the *front* of the queue
  - also known as *first-in-first-out* (FIFO) structure

Basic Operations

- Four basic operations
  - *enqueue*: adding an item at the rear of the queue
  - *dequeue*: deleting an item from the front of the queue
  - *queue front*: examining the item at the front of the queue
  - *queue rear*: examining the item at the rear of the queue

Example - Enqueue

Example - Dequeue

Example – Queue Front

Example – Queue Rear

Overflow and Underflow

- Queue overflow
  - occurs when one tries to perform *enqueue* in an already full queue

- Queue underflow
  - occurs when one tries to perform *dequeue/queue front/queue rear* in an empty queue

Implementing the Queue as an ADT

- Choose an internal data representation for the items in the queue (e.g., array or linked list)
- Implement the queue operations
  - initialize the queue
  - destroy the queue
  - enqueue
  - dequeue
  - queue front
  - queue rear
  - is queue empty
  - is queue full
  - count # of items

Implementation of Queues as Arrays

- Queue items are stored in an array
- Two variables should be used to keep track of the two ends of the queue
  - *front*: index of the first element of the queue
  - *rear*: index of the last element of the queue
- What should be the relationship between *front* and *rear*?

The **Queue** ADT

```
class Queue
{
   private:
      int* array; // each item is an int
      int size, count, front, rear;
   public:
      Queue(int s = 100);
      ~Queue();
      bool enqueue(int dataIn);
      bool dequeue(int& dataOut);
      bool queueFront(int& dataOut);
      bool queueRear(int& dataOut);
      bool isEmpty();
      bool isFull();
      int  getCount();
};
```

Enqueue and Dequeue

- Enqueue
  - advance *rear* to the next array position
  - add the element into the index *rear*
- Dequeue
  - retrieve the element from index *front*
  - advance *front* to the next array position
- So, *front* will change after each *dequeue* and *rear* will change after each *enqueue*

Exercise

- Create a queue using an array of size of 5 and draw the queue after the following operations are performed:

$E^{10}E^{15}E^{20}E^{25}E^{30}DDE^{35}DE^{40}D$

Member Functions

```
// constructor
Queue::Queue (int s)
{



}
```

Member Functions

```
// destructor
Queue::~Queue ()
{









}
```

Member Functions

```
bool Queue::enqueue (int dataIn)
{
   // should take care of overflow








}
```

Member Functions

```
bool Queue::dequeue (int& dataOut)
{
   // should take care of underflow





}
```

Member Functions

```
bool Queue::queueFront (int& dataOut)
{
   // should take care of underflow




}

bool Queue::queueRear (int& dataOut)
{
   // should take care of underflow




}
```

Member Functions

```
bool Queue::isEmpty ()
{




}

bool Queue::isFull ()
{




}
```

Member Functions

```
int Queue::getCount ()
{




}
```

Implementation of Queues as Linked Lists

- Queue items are stored in a linked list
- We need to know the front and rear of the queue i.e., we need to have two pointers
  - *front* is the pointer to the first node
  - *rear* is the pointer to the last node
- Operations
  - enqueue and queueRear done at the end of the list
  - dequeue and queueFront done at the beginning of the list

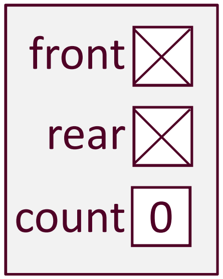
The **Queue** ADT (Linked Implementation)

```
class Queue
{
   private:
      Node *front, *rear;
      int count;
   public:
      Queue();
      ~Queue();
      bool enqueue(int dataIn);
      bool dequeue(int& dataOut);
      bool queueFront(int& dataOut);
      bool queueRear(int& dataOut);
      bool isEmpty();
      bool isFull();
      int  getCount();
};
```

The **Node** Type

```
struct Node
{
   int data;   // item stored at node
   Node *next; // pointer to next node
};
```

Queue Objects

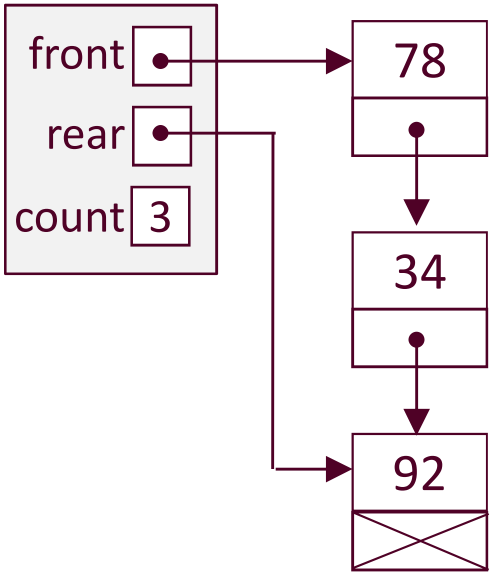


front

rear

count 0

Empty queue



front

rear

count 3

78

34

92

Non-empty queue

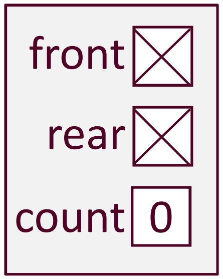
Member Functions

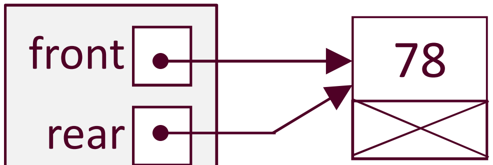
```
// constructor
Queue::Queue ()
{



}

// destructor
Queue::~Queue ()
{



}
```

Enqueue Example

Empty Queue

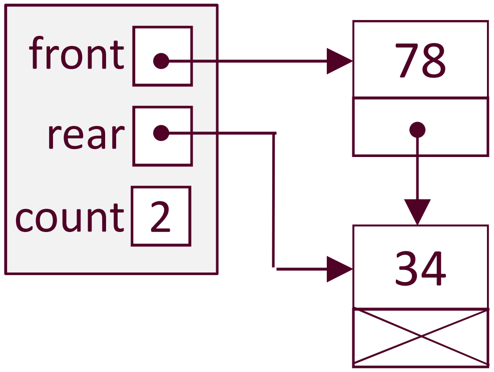
Insert 78

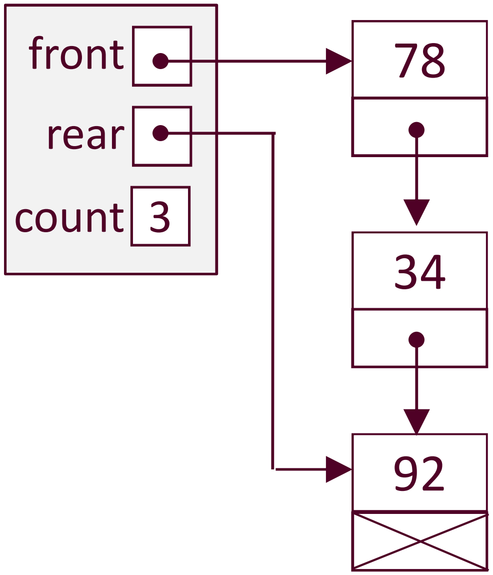




Nonempty Queue

Insert 92



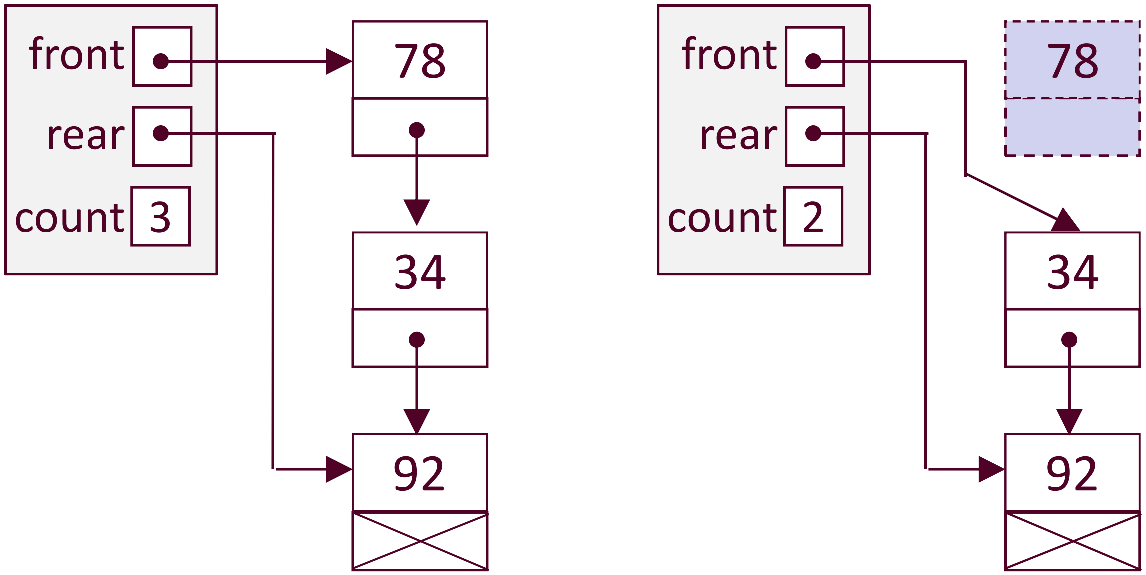


Member Functions

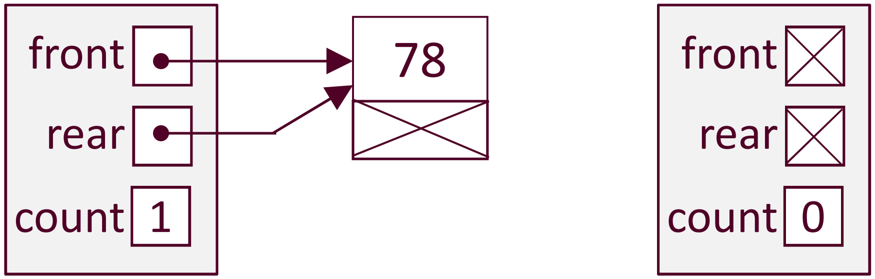
```
bool Queue::enqueue (int dataIn)
{
   // should take care of overflow










}
```

Dequeue Example

Queue with more than one item



Queue with one item



Member Functions

```
bool Queue::dequeue (int& dataOut)
{
   // should take care of underflow











}
```

Member Functions

```
bool Queue::queueFront (int& dataOut)
{
   // should take care of underflow




}

bool Queue::queueRear (int& dataOut)
{
   // should take care of underflow




}
```

Member Functions

```
bool Queue::isEmpty ()
{




}

bool Queue::isFull ()
{




}
```

Member Functions

```
int Queue::getCount ()
{




}
```

Exercise

- Write the code to create a queue using the **`Queue`** ADT and to populate the queue with integer values read from a file called **`"test.txt"`**.
- Write the code to delete and display all the items stored in the queue.