# Queues

## Chapter 8

# Chapter Contents

8.1 Introduction to Queues

8.2 Designing and Building a Queue Class – Array Based

8.3 Linked Queues

8.4 Application of Queues: Buffers and Scheduling

8.5 Case Study: Center Simulation

# Chapter Objectives

- To study a queue as an ADT
- Build a static-array-based implementation of queues
- Build a dynamic-array-based implementation of queues
- Show how queues are used in I/O buffers and scheduling in a computer system
- (Optional) See how queues are used in simulations of phenomena that involve waiting in lines

# Introduction to Queues

- A queue is a waiting line – seen in daily life
  - A line of people waiting for a bank teller
  - A line of cars at a toll both
  - "This is the captain, we're 5th in line for takeoff"
- What other kinds of queues can you think of

# The Queue As an ADT

- A queue is a sequence of data elements
- In the sequence
  - Items can be removed only at the front
  - Items can be added only at the other end, the back
- Basic operations
  - Construct a queue
  - Check if empty
  - Enqueue (add element to back)
  - Front (retrieve value of element from front)
  - Dequeue (remove element from front)

# Example: Drill and Practice Problems

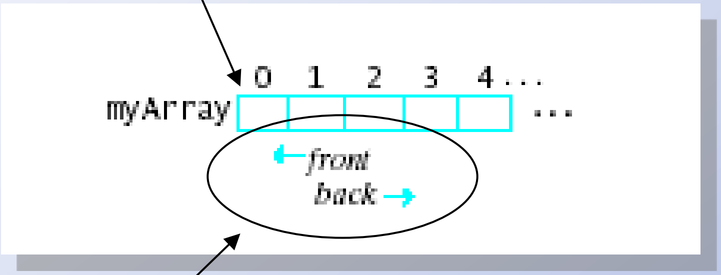
- We seek a program to present elementary math problems to students
- They are presented with a problem where they combine randomly generated integers
- When they respond and are
  - Successful – proceed to another problem
  - Unsuccessful – problem stored for asking again later
- The program will determine the number of problems and largest values used in the operation
- Then it will generate that many problems and place them in the problem queue

# Example: Drill and Practice Problems

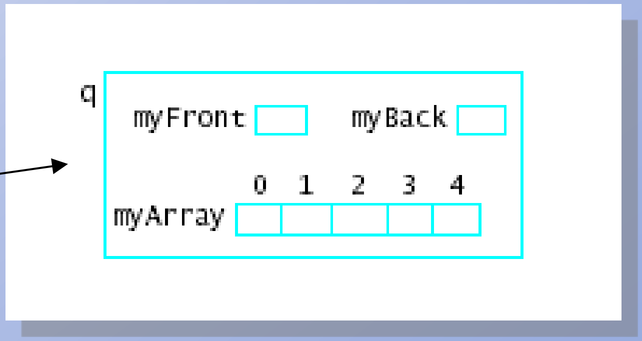
- For now, we will assume a **`queue`** class
- Note declaration of the **`AdditionProblem`** class, Fig 8.1A
- Implementation, **`AdditionProblem.cpp`**, Fig 8.1B
- View source code of drill and practice program, Fig. 8.1C

# Designing and Building a Queue Class   Array-Based

- Consider an array in which to store a queue

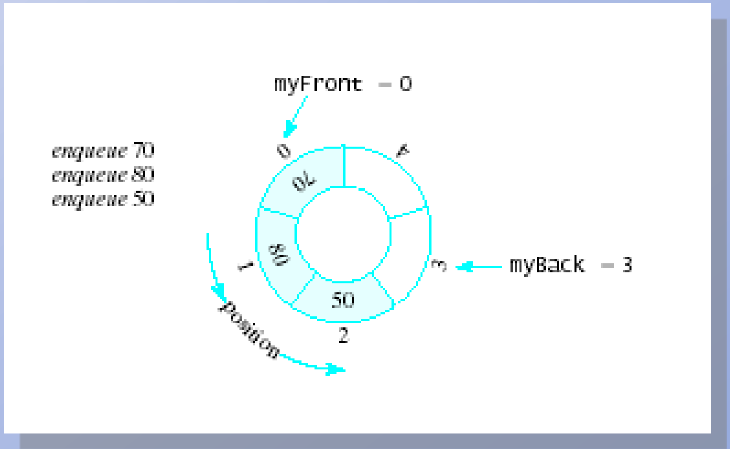


- Note additional variables needed
  - **myFront, myBack**
- Picture a queue object like this



# Designing and Building a Queue Class Array-Based

- Problems
  - We quickly "walk off the end" of the array
- Possible solutions
  - Shift array elements
  - Use a circular queue
  - Note that both empty and full queue gives `myBack == myFront`

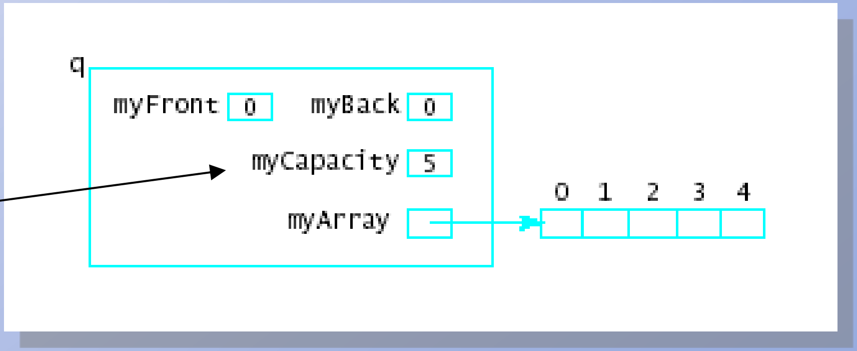


# Designing and Building a Queue Class Array-Based

- Using a static array
  - `QUEUE_CAPACITY` specified
  - Enqueue increments `myBack` using mod operator, checks for full queue
  - Dequeue increments `myFront` using mod operator, checks for empty queue
- Note declaration of Queue class, Fig 8.2A
- View implementation, Fig. 8.2B

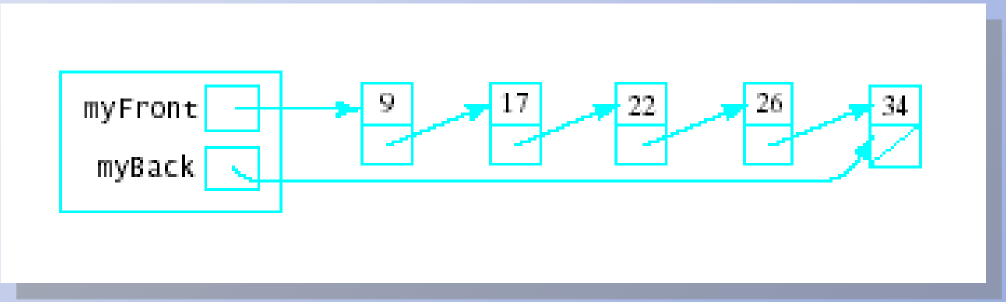
# Using Dynamic Array to Store Queue Elements

- Similar problems as with list and stack
  - Fixed size array can be specified too large or too small
- Dynamic array design allows sizing of array for multiple situations
- Results in structure as shown
  - `myCapacity` determined at run time

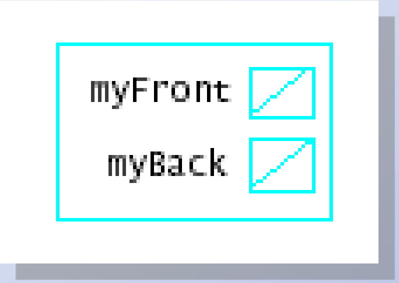


# Linked Queues

- Even with dynamic allocation of queue size
  - Array size is still fixed
  - Cannot be adjusted during run of program
- Could use linked list to store queue elements
  - Can grow and shrink to fit the situation
  - No need for upper bound (`myCapacity`)



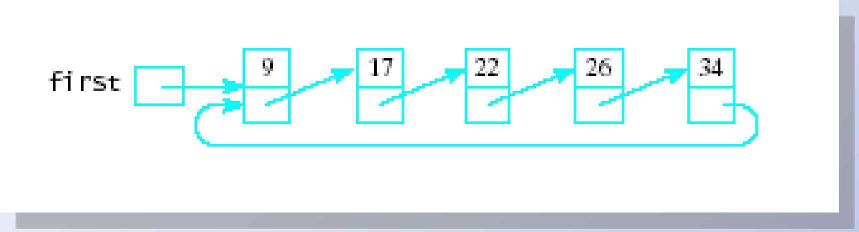
# Linked Queues



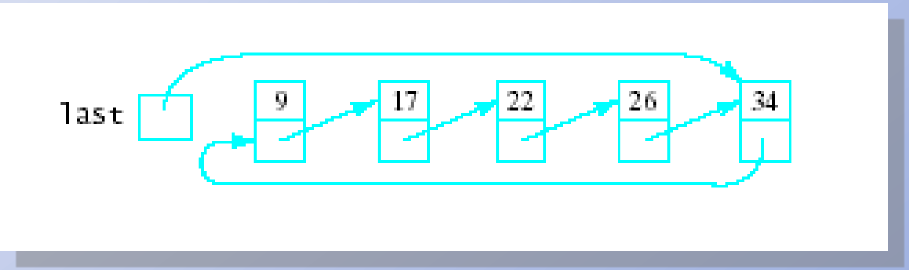
- Constructor initializes `myFront`, `myBack`
- Front
  - `return myFront->data`
- Dequeue
  - Delete first node (watch for empty queue)
- Enqueue
  - Insert node at end of list
- View `LQueue.h` declaration, Fig 8.3A
- Note definition, `LQueue.cpp`, Fig 8.3B
- Driver program, Fig. 8.3C

# Circular Linked List

- Possible to treat the linked list as circular

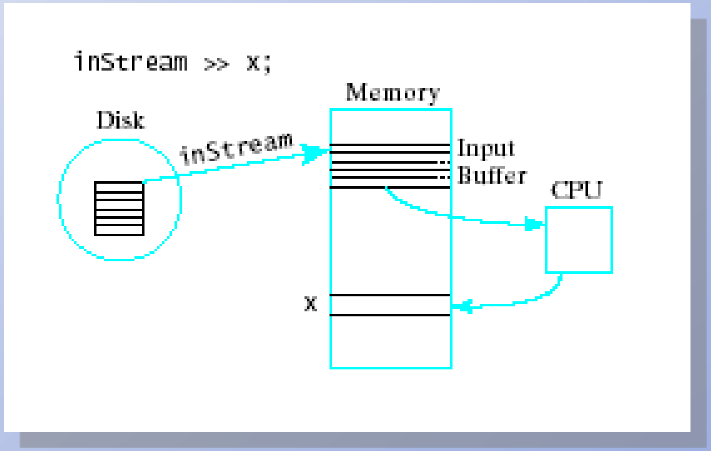


  - Last node points back to first node
  - Alternatively keep pointer to last node rather than first node



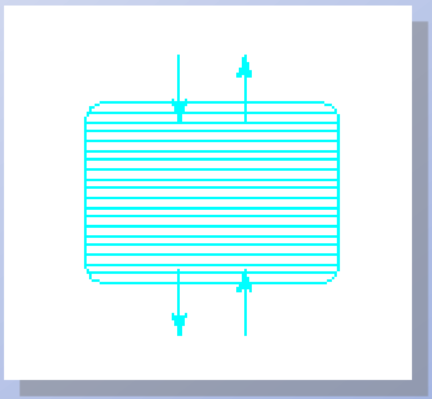
# Application of Queues: Buffers and Scheduling

- Important use of queues is I/O scheduling
  - Use buffers in memory to improve program execution
  - Buffer arranged in FIFO structure



# Application of Queues: Buffers and Scheduling

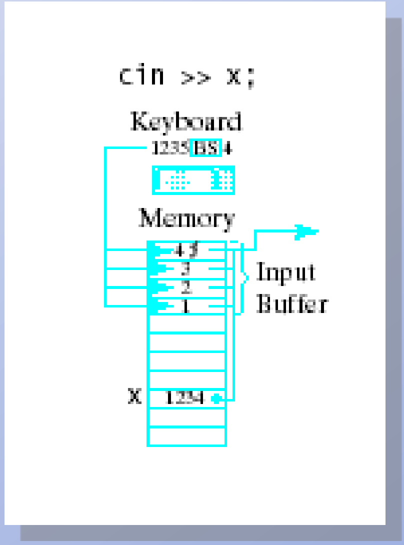
- Also times when insertions, deletions must be made from both ends
  - Consider a scrolling window on the screen



- This requires a double ended queue
  - Called a **deque** (pronounced "deck")
  - Could also be considered a double ended stack (or "dack")

# Application of Queues: Buffers and Scheduling

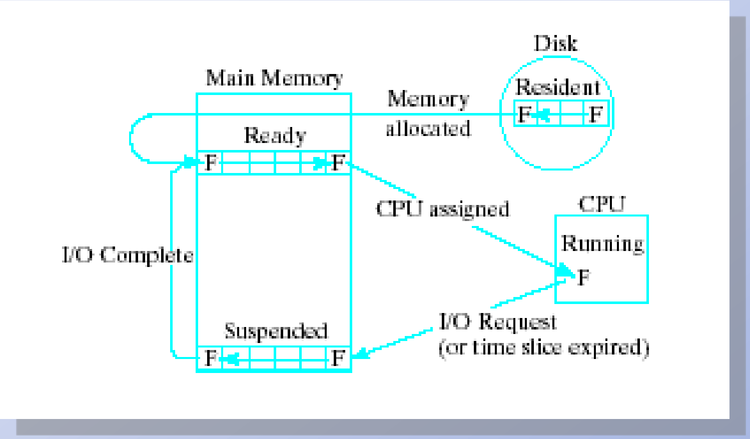
- Consider a keyboard buffer
  - Acts as a queue
  - But elements may be removed from the back of the queue with backspace key



- A printer spool is a queue of print jobs

# Application of Queues: Buffers and Scheduling

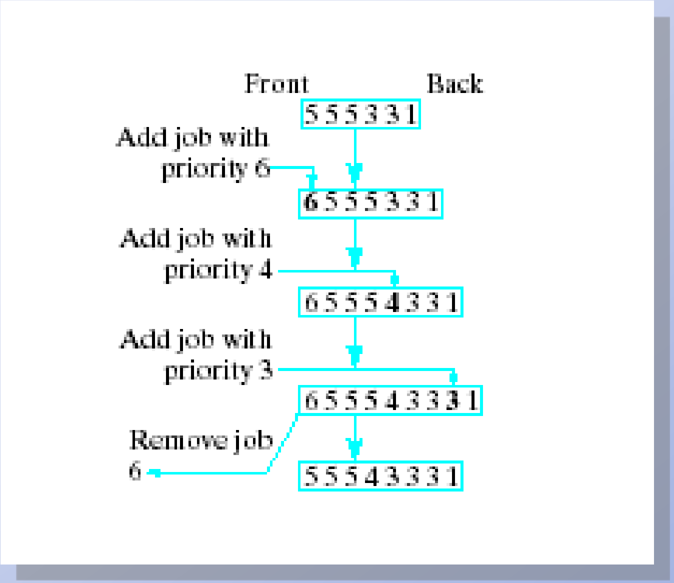
- Queues used to schedule tasks within an operating system



- Job moves from disk to ready queue

# Application of Queues: Buffers and Scheduling

- Ready queue may actually be a priority queue … job may get to "cut the line" based on its priority



# Case Study: Information Center Simulation

- Most waiting lines (queues) are dynamic
  - Their lengths grow and shrink over time
- Simulation models this dynamic process
  - Enables study of the behavior of the process
  - Modeled with one or more equations
- Queue behavior involves randomness
  - We will us pseudorandom number generator

# Problem Analysis and Specification

- Consider an information center
  - Calls arrive at random intervals
  - Placed in queue of incoming calls
  - When agent becomes available, services call at front of queue
- We will simulate receipt of "calls" for some number of "minutes"
  - we keep track of calls in the queue

# Problem Analysis and Specification

- Input to the simulation program
  - – Time limit
  - – Arrival rate of calls
  - – Distribution of service times
- Desired output
  - – Number of calls processed
  - – Average waiting time per call
- Note declaration of Simulation class, Fig. 8-4

# Problem Analysis and Specification

- Constructor
  - Initialize input data members
  - `myTimer`, `myIncomingCalls` initialized by their constructors
- The `run()` method
  - Starts and manages simulation
- The `checkForNewCall()` method
  - Random number generated, compared to myArrivalRate
  - If new call has arrived, create new Call object, place in queue

# Problem Analysis and Specification

- The **`service()`** method
  - Checks time remaining for current call
  - When done, retrieves and starts up next call from front of queue
- The **`display()`** method
  - Report generated at end of simulation
- View definition of function members Fig. 8-5A
- Note driver program for simulation, Fig. 8-5B
- Reference the Timer class and Call class used in the **`Simulation`** class