Data Structures and Algorithms

Chapter 8
Queues

Objectives

- Learn about queues
- Examine various queue operations
- Learn how to implement a queue as an array
- Learn how to implement a queue as a linked list
- Discover queue applications
- Become aware of the STL `class queue`

Introduction

- Queue data structure
  - Elements added at one end (rear), deleted from other end (front)
  - First In First Out (FIFO)
  - Middle elements inaccessible

Queue Operations

- Two key operations
  - `addQueue`
  - `deleteQueue`
- Additional operations
  - `initializeQueue`, `isEmptyQueue`, `isFullQueue`, `front`, `back`
- `queueFront`, `queueRear` pointers
  - Keep track of front and rear
- See code on pages 453-454

Implementation of Queues as Arrays

- Four member variables
  - Array to store queue elements
  - Variables `queueFront`, `queueRear`
  - Variable `maxQueueSize`
- Using `queueFront`, `queueRear` to access queue elements
  - `queueFront`: first queue element index
  - `queueRear`: last queue element index
    - `queueFront` changes after each `deleteQueue` operation
    - `queueRear` changes after each `addQueue` operation

Implementation of Queues as Arrays (cont'd.)

- Execute operation
  - `addQueue(Queue,'A');`
- Execute
  - `addQueue(Queue,'B');`
  - `addQueue(Queue,'C');`
- Execute
  - `deleteQueue();`

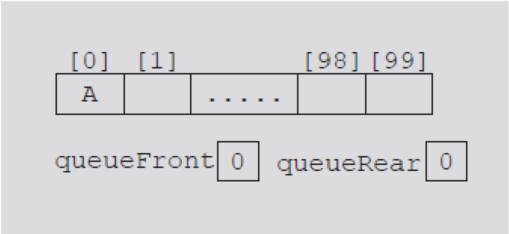

FIGURE 8-1 Queue after the first addQueue operation

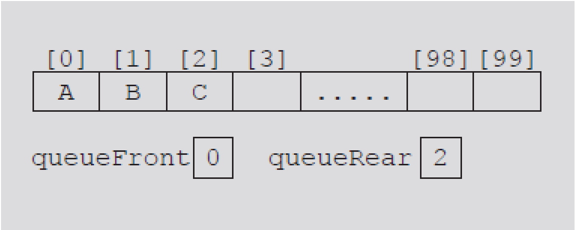

FIGURE 8-2 Queue after two more `addQueue` operations

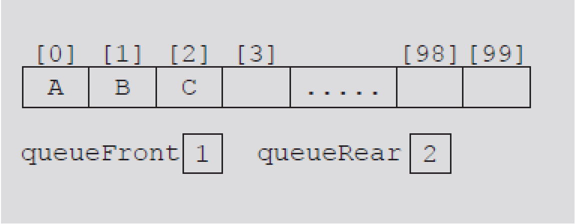

FIGURE 8-3 Queue after the `deleteQueue` operation

Implementation of Queues as Arrays (cont'd.)

- Consider the sequence of operations: AAADADADADADADADA...
  - Eventually index `queueRear` points to last array position
    - Looks like a full queue
    - Reality: queue has two or three elements, array empty in the front

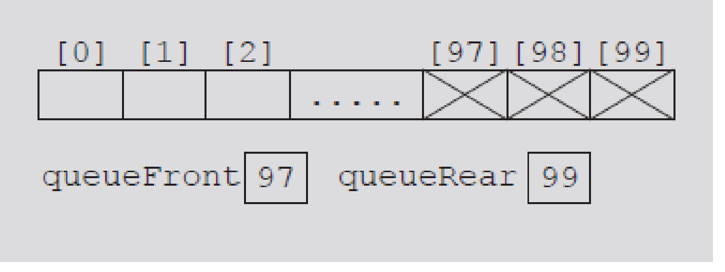

FIGURE 8-4 Queue after the sequence of operations AAADADADADADA...

Implementation of Queues as Arrays (cont'd.)

- First solution
  - Upon queue overflow to the rear
    - Check value of `queueFront`
    - If room in front: slide all queue elements toward first array position
  - Works if queue size very small
- Second solution: assume circular array

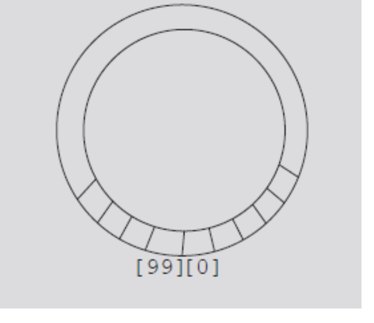

FIGURE 8-5 Circular queue

Implementation of Queues as Arrays (cont'd.)

- `queueRear = (queueRear + 1) % maxQueueSize;`
  - Advances `queueRear` (`queueFront`) to next array position

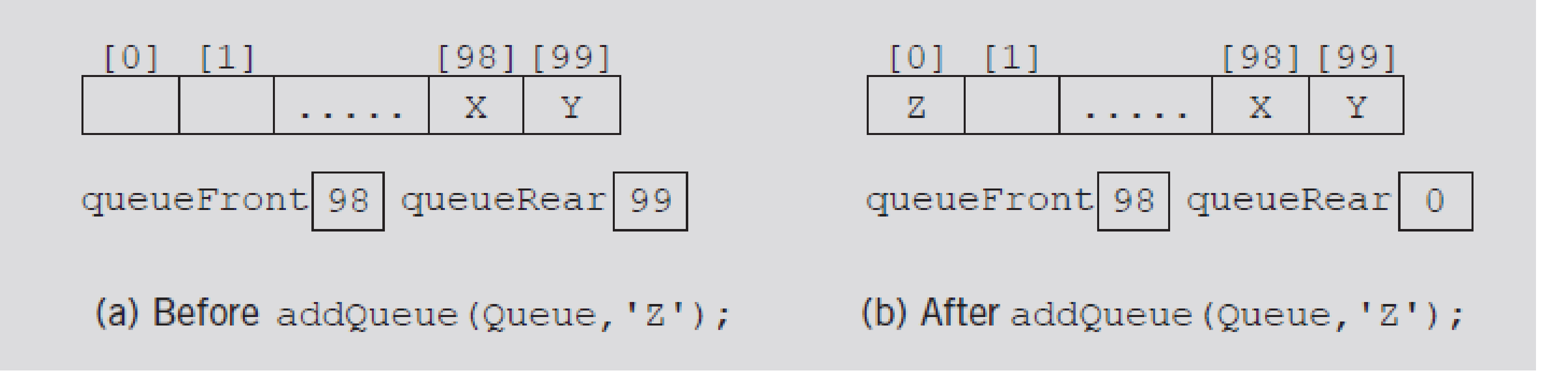

FIGURE 8-6 Queue before and after the `add` operation

Implementation of Queues as Arrays (cont'd.)

- If `queueRear` < `maxQueueSize` – 1
  - `queueRear` + 1 <= `maxQueueSize` – 1
  - (`queueRear` + 1) % `maxQueueSize` = `queueRear` + 1
- If `queueRear` == `maxQueueSize` – 1
  - `queueRear` + 1 == `maxQueueSize`
  - (`queueRear` + 1) % `maxQueueSize` = 0
- `queueRear` set to zero
  - First array position

Implementation of Queues as Arrays (cont'd.)

- Two cases with identical `queueFront`, `queueRear` values
  - Figure 8-7(b) represents an empty queue
  - Figure 8-8(b) represents a full queue

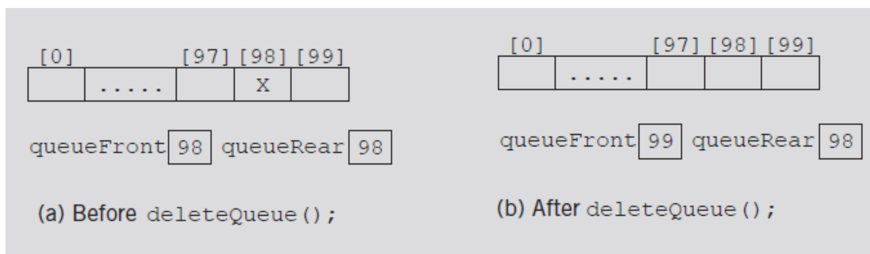

FIGURE 8-7 Queue before and after the `delete` operation

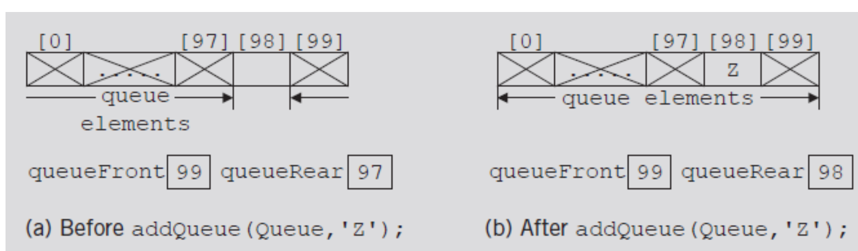

FIGURE 8-8 Queue before and after the `add` operation

Implementation of Queues as Arrays (cont'd.)

- First solution: use variable `count`
  - Incremented when new element added
  - Decremented when element removed
  - Functions `initializeQueue`, `destroyQueue` initialize count to zero

Implementation of Queues as Arrays (cont'd.)

- Second solution
  - `queueFront` indicates index of array position preceding first element of the queue
  - Assume `queueRear` indicates index of last element
    - Empty queue if `queueFront == queueRear`
  - Slot indicated by index `queueFront` is reserved
  - Queue is full
    - If next available space represents special reserved slot

Empty Queue and Full Queue

- Empty queue
  - If `count == 0`
- Full queue
  - If `count == maxQueueSize`

```
template <class Type>
bool queueType<Type>::isEmptyQueue() const
{
    return (count == 0);
} //end isEmptyQueue

template <class Type>
bool queueType<Type>::isFullQueue() const
{
    return (count == maxQueueSize);
} //end isFullQueue
```

Initialize Queue

- Initializes queue to empty state
  - First element added at the first array position
  - Initialize `queueFront` to zero, `queueRear` to `maxQueueSize` - one, `count` to zero

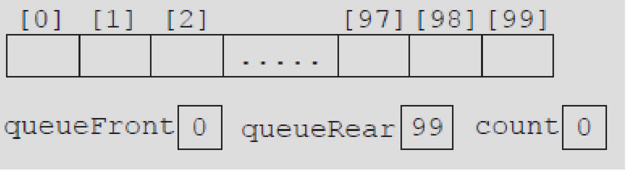

FIGURE 8-10 Empty queue

```
template <class Type>
void queueType<Type>::initializeQueue()
{
    queueFront = 0;
    queueRear = maxQueueSize - 1;
    count = 0;
} //end initializeQueue
```

Front

- Returns first queue element
  - If the queue nonempty
    - Element indicated by index `queueFront` returned
  - Otherwise
    - Program terminates

```
template <class Type>
Type queueType<Type>::front() const
{
    assert(!isEmptyQueue());
    return list[queueFront];
} //end front
```

Back

- Returns last queue element
  - If queue nonempty
    - Returns element indicated by index `queueRear`
  - Otherwise
    - Program terminates

```
template <class Type>
Type queueType<Type>::back() const
{
    assert(!isEmptyQueue());
    return list[queueRear];
} //end back
```

Add Queue

```
template <class Type>
void queueType<Type>::addQueue(const Type& newElement)
{
    if (!isFullQueue())
    {
        queueRear = (queueRear + 1) % maxQueueSize; //use the
                                    //mod operator to advance queueRear
                                    //because the array is circular
        count++;
        list[queueRear] = newElement;
    }
    else
        cout << "Cannot add to a full queue." << endl;
} //end addQueue
```

Delete Queue

```
template <class Type>
void queueType<Type>::deleteQueue()
{
    if (!isEmptyQueue())
    {
        count--;
        queueFront = (queueFront + 1) % maxQueueSize; //use the
                                //mod operator to advance queueFront
                                //because the array is circular
    }
    else
        cout << "Cannot remove from an empty queue" << endl;
} //end deleteQueue
```

Constructors and Destructors

```
template <class Type>
queueType<Type>::queueType(int queueSize)
{
    if (queueSize <= 0)
    {
        cout << "Size of the array to hold the queue must "
             << "be positive." << endl;
        cout << "Creating an array of size 100." << endl;

        maxQueueSize = 100;
    }
    else
        maxQueueSize = queueSize;   //set maxQueueSize to
                                    //queueSize

    queueFront = 0;                 //initialize queueFront
    queueRear = maxQueueSize - 1;   //initialize queueRear
    count = 0;
    list = new Type[maxQueueSize];  //create the array to
                                    //hold the queue elements
} //end constructor
```

Constructors and Destructors (cont'd.)

- Array storing queue elements
  - Created dynamically
  - When queue object goes out of scope
    - Destructor deallocates memory occupied by the array storing queue elements

```
template <class Type>
queueType<Type>::~queueType()
{
    delete [] list;
}
```

Linked Implementation of Queues

- Array implementation issues
  - Fixed array size
    - Finite number of queue elements
  - Requires special array treatment with the values of the indices `queueFront`, `queueRear`
- Linked implementation of a queue
  - Simplifies special cases of the array implementation
  - Queue never full
- See code on pages 464-465

Empty and Full Queue

- Empty queue if `queueFront` is `NULL`
- Memory allocated dynamically
  - Queue never full
  - Function implementing `isFullQueue` operation returns the value `false`

```
template <class Type>
bool linkedQueueType<Type>::isEmptyQueue() const
{
    return(queueFront == NULL);
} //end

template <class Type>
bool linkedQueueType<Type>::isFullQueue() const
{
    return false;
} //end isFullQueue
```

Initialize Queue

- Initializes queue to an empty state
  - Empty if no elements in the queue

```
template <class Type>
void linkedQueueType<Type>::initializeQueue()
{
    nodeType<Type> *temp;

    while (queueFront!= NULL)  //while there are elements left
                               //in the queue
    {
        temp = queueFront;  //set temp to point to the current node
        queueFront = queueFront->link;  //advance first to
                                        //the next node
        delete temp;    //deallocate memory occupied by temp
    }

    queueRear = NULL;  //set rear to NULL
} //end initializeQueue
```

addQueue, front, back, and deleteQueue Operations

- addQueue operation adds a new element at end of the queue
  - Access the pointer queueRear

```
template <class Type>
void linkedQueueType<Type>::addQueue(const Type& newElement)
{
   nodeType<Type> *newNode;
   newNode = new nodeType<Type>; //create the node
   newNode->info = newElement; //store the info
   newNode->link = NULL; //initialize the link field to NULL
   if (queueFront == NULL) //if initially the queue is empty
   {
      queueFront = newNode;
      queueRear = newNode;
   }
   else //add newNode at the end
   {
      queueRear->link = newNode;
      queueRear = queueRear->link;
   }
}//end addQueue
```

`addQueue`, `front`, `back`, and `deleteQueue` Operations (cont'd.)

- If queue nonempty
  - Operation `front` returns first element
  - Element indicated `queueFront` returned
- If queue empty: `front` terminates the program

```
template <class Type>
Type linkedQueueType<Type>::front() const
{
    assert(queueFront != NULL);
    return queueFront->info;
} //end front
```

`addQueue`, `front`, `back`, and `deleteQueue` Operations (cont'd.)

- If queue nonempty
  - Operation `back` returns last element
  - Element indicated by `queueRear` returned
- If queue empty: `back` terminates the program

```
template <class Type>
Type linkedQueueType<Type>::back() const
{
    assert(queueRear!= NULL);
    return queueRear->info;
} //end back
```

`addQueue`, `front`, `back`, and `deleteQueue` Operations (cont'd.)

- If queue nonempty
  - Operation `deleteQueue` removes first element
    - Access pointer `queueFront`

```
template <class Type>
void linkedQueueType<Type>::deleteQueue()
{
    nodeType<Type> *temp;

    if (!isEmptyQueue())
    {
        temp = queueFront;  //make temp point to the first node
        queueFront = queueFront->link; //advance queueFront

        delete temp;    //delete the first node

        if (queueFront == NULL) //if after deletion the
                                //queue is empty
            queueRear = NULL;   //set queueRear to NULL
    }
    else
        cout << "Cannot remove from an empty queue" << endl;
}//end deleteQueue
```

`addQueue`, `front`, `back`, and `deleteQueue` Operations (cont'd.)

- Default constructor
  - When queue object goes out of scope
    - Destructor destroys the queue
    - Deallocates memory occupied by the queue elements
  - Function definition similar to function `initializeQueue`

```
template<class Type>
linkedQueueType<Type>::linkedQueueType()
{
    queueFront = NULL; //set front to null
    queueRear = NULL;  //set rear to null
} //end default constructor
```

Queue Derived from the `class` `unorderedLinkedListType`

- Linked queue implementation
  - Similar to forward manner linked list implementation
  - Similar operations
    - `add Queue`, `insertFirst`
    - `initializeQueue` , `initializeList`
    - `isEmptyQueue`, `isEmptyList`
  - `deleteQueue` operation implemented as before
  - Same pointers
    - `queueFront` and `first`, `queueRear` and `last`

Priority Queues

- Queue structure ensures items processed in the order received
- Priority queues
  - Customers (jobs) with higher priority pushed to the front of the queue
- Implementation
  - Ordinary linked list
    - Keeps items in order from the highest to lowest priority
  - Treelike structure
    - Very effective
    - Chapter 10

Application of Queues: Simulation

- Simulation
  - Technique in which one system models the behavior of another system
- Computer simulation
  - Represents objects being studied as data
  - Actions implemented with algorithms
    - Programming language implements algorithms with functions
    - Functions implement object actions

Application of Queues: Simulation (cont'd.)

- Computer simulation (cont'd.)
  - C++ combines data, data operations into a single unit using classes
    - Objects represented as classes
    - Class member variables describe object properties
    - Function members describe actions on data
  - Change in simulation results occurs if change in data value or modification of function definitions occurs
  - Main goal
    - Generate results showing the performance of an existing system
    - Predict performance of a proposed system

Application of Queues: Simulation (cont'd.)

- Queuing systems
  - Computer simulations
    - Queues represent the basic data structure
  - Queues of objects
    - Waiting to be served by various servers
  - Consist of servers and queues of objects waiting to be served

Designing a Queuing System

- Server
  - Object that provides the service
- Customer
  - Object receiving the service
- Transaction time (service time)
  - Time required to serve a customer
- Queuing system consists of servers, queue of waiting objects
  - Model system consisting of a list of servers; waiting queue holding the customers to be served

Designing a Queuing System (cont'd.)

- Modeling a queuing system: requirements
  - Number of servers, expected customer arrival time, time between customer arrivals, number of events affecting system
- Time-driven simulation
  - Clock implemented as a counter
  - Passage of time
    - Implemented by incrementing counter by one
- Run simulation for fixed amount of time
  - Example: run for 100 minutes
    - Counter starts at one and goes up to 100 using a loop

Customer

- Has a customer number, arrival time, waiting time, transaction time, departure time
  - With known arrival time, waiting time, transaction time
    - Can determine departure time (add these three times)
- See `class customerType` code on pages 475-476
  - Implements customer as an ADT
- Member function definitions
  - Functions `setWaitingTime`, `getArrivalTime`, `getTransactionTime`, `getCustomerNumber`
    - Left as exercises

```
void customerType::setCustomerInfo(int cN, int arrvTime,
                                   int wTime, int tTime)
{
    customerNumber = cN;
    arrivalTime = arrvTime;
    waitingTime = wTime;
    transactionTime = tTime;
}

customerType::customerType(int cN, int arrvTime,
                           int wTime, int tTime)
{
    setCustomerInfo(cN, arrvTime, wTime, tTime);
}


int customerType::getWaitingTime() const
{
    return waitingTime;
}


void customerType::incrementWaitingTime()
{
    waitingTime++;
}
```

Server

- At any given time unit
  - Server either busy serving a customer or free
- String variable sets server status
- Every server has a timer
- Program might need to know which customer served by which server
  - Server stores information of the customer being served
- Three member variables associated with a server
  - `status`, `transactionTime`, `currentCustomer`

Server (cont'd.)

- Basic operations performed on a server
  - Check if server free
  - Set server as free
  - Set server as busy
  - Set transaction time
  - Return remaining transaction time
  - If server busy after each time unit
    - Decrement transaction time by one time unit
- See `class serverType` code on page 477
  - Implements server as an ADT
- Member function definitions

```
serverType::serverType()
{
    status = "free";
    transactionTime = 0;
}

bool serverType::isFree() const
{
    return (status == "free");
}

void serverType::setBusy()
{
    status = "busy";
}

void serverType::setFree()
{
    status = "free";
}

void serverType::setTransactionTime(int t)
{
    transactionTime = t;
}

void serverType::setTransactionTime()
{
    int time;

    time = currentCustomer.getTransactionTime();

    transactionTime = time;
}

void serverType::decreaseTransactionTime()
{
    transactionTime--;
}
```

Server List

- Set of servers
- `class serverListType`
  - Two member variables
    - Store number of servers
    - Maintain a list of servers
  - List of servers created during program execution
  - Several operations must be performed on a server list
  - See `class serverListType` code on page 481
    - Implements the list of servers as an ADT
  - Definitions of member functions

```
serverListType::serverListType(int num)
{
    numOfServers = num;
    servers = new serverType[num];
}

serverListType::~serverListType()
{
    delete [] servers;
}

int serverListType::getFreeServerID() const
{
    int serverID = -1;

    for (int i = 0; i < numOfServers; i++)
        if (servers[i].isFree())
        {
            serverID = i;
            break;
        }

    return serverID;
}

int serverListType::getNumberOfBusyServers() const
{
    int busyServers = 0;

    for (int i = 0; i < numOfServers; i++)
        if (!servers[i].isFree())
            busyServers++;

    return busyServers;
}
```

```cpp
void serverListType::setServerBusy(int serverID,
                              customerType cCustomer, int tTime)
{
    servers[serverID].setBusy();
    servers[serverID].setTransactionTime(tTime);
    servers[serverID].setCurrentCustomer(cCustomer);
}
void serverListType::setServerBusy(int serverID,
                                   customerType cCustomer)
{
    int time = cCustomer.getTransactionTime();

    servers[serverID].setBusy();
    servers[serverID].setTransactionTime(time);
    servers[serverID].setCurrentCustomer(cCustomer);
}
void serverListType::updateServers(ostream& outF)
{
    for (int i = 0; i < numOfServers; i++)
        if (!servers[i].isFree())
        {
           servers[i].decreaseTransactionTime();

           if (servers[i].getRemainingTransactionTime() == 0)
           {
               outF << "From server number " << (i + 1)
                    << " customer number "
                    << servers[i].getCurrentCustomerNumber()
                    << "\n     departed at clock unit "
                    << servers[i].getCurrentCustomerArrivalTime()
                    + servers[i].getCurrentCustomerWaitingTime()
                    + servers[i].getCurrentCustomerTransactionTime()
                    << endl;
               servers[i].setFree();
           }
        }
}
```

Waiting Customers Queue

- Upon arrival, customer goes to end of queue
  - When server available
    - Customer at front of queue leaves to conduct transaction
  - After each time unit, waiting time incremented by one
- Derive `class waitingCustomerQueueType` from `class queueType`
  - Add additional operations to implement the customer queue
  - See code on page 485

Main Program

- Run the simulation
  - Need information (simulation parameters)
    - Number of time units the simulation should run
    - The number of servers
    - Transaction time
    - Approximate time between customer arrivals
  - Function `setSimulationParameters`
    - Prompts user for these values
    - See code on page 487

Main Program (cont'd.)

- General algorithm to start the transaction

1. Remove the customer from the front of the queue.

```
customer = customerQueue.front();
customerQueue.deleteQueue();
```

2. Update the total waiting time by adding the current customer's waiting time to the previous total waiting time.

```
totalWait = totalWait + customer.getWaitingTime();
```

3. Set the free server to begin the transaction.

```
serverList.setServerBusy(serverID, customer, transTime);
```

Main Program (cont'd.)

- Use the Poisson distribution from statistics
  - Probability of *y* events occurring at a given time
    - Where λ is the expected value that y events occur at that time

$$P(y) = \frac{\lambda^y e^{-\lambda}}{y!}, y = 0, 1, 2, \ldots,$$

- Function `runSimulation` implements the simulation
  - Function `main` is simple and straightforward
    - Calls only the function `runSimulation`

Summary

- Queue
  - First In First Out (FIFO) data structure
  - Implemented as array or linked list
  - Linked lists: queue never full
- Standard Template Library (STL)
  - Provides a class to implement a queue in a program
- Priority Queue
  - Customers with higher priority pushed to the front
- Simulation
  - Common application for queues