Chapter 8: Queues
OUTLINE

1. The abstract data type queue
2. Simple applications of the ADT queue
3. Implementation of the ADT queue
4. A summary of position-oriented ADTs
5. Application: simulation
6. Summary
The abstract data type queue
The abstract data type queue

Applications in real life

- BANK
- POST OFFICE
- RESTROOM
- BUS STATION
The abstract data type queue

- First-in, first-out (FIFO) ADT queue
  - New items enter a queue at its back or rear, and items leave a queue from its front

- ADT queue operations
  - Create an empty queue
  - Determine whether a queue is empty
  - Add a new item to the queue
  - Remove from the queue the item that was added earliest
  - Remove all the items from the queue
  - Retrieve from the queue the item that was added earliest

All operations are only allowed only at its two ends

<table>
<thead>
<tr>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>front</td>
</tr>
<tr>
<td>back</td>
</tr>
<tr>
<td>items</td>
</tr>
<tr>
<td>createQueue()</td>
</tr>
<tr>
<td>dequeueAll()</td>
</tr>
<tr>
<td>isEmpty()</td>
</tr>
<tr>
<td>enqueue(newitem)</td>
</tr>
<tr>
<td>dequeue()</td>
</tr>
<tr>
<td>peek()</td>
</tr>
</tbody>
</table>
**Example**

- `queue.createQueue();`
- `queue.enqueue(3);`
- `queue.enqueue(6);`
- `queue.enqueue(7);`
- `queue.peek();`
- `queue.dequeue();`
- `queue.enqueue(5);`

**Bonus question**

- `queue.enqueue(3);`
- `queue.peek(3);`
- `queue.dequeue(4);`
Simple applications of the ADT queue
Reading a String of Characters

- A queue can retain characters in the order in which you type them
  
  ```java
  while (not end of line){
    Read a new character ch
    aQueue.enqueue(ch)
  }
  ```

- Example 3 4 2 to decimal value
  
  ```java
  while (ch is blank){
    ch=aqueue.dequeue();
  }
  n=0;
  done=false;
  while (!done and ch is digit){
    n=10*n+integer that ch represents
    if (!aQueue.isEmpty()){
      ch=aQueue.dequeue();
    }
    else done=true;
  }
  ```
Simple applications of the ADT queue

- **Recognizing Palindromes**

  A string of characters that reads the same from left to right as it does from right to left

  - An example of palindromes
    - abcddcba, abcd$dcba

  - Queue in conjunction with a stack to recognize palindromes
    - **stack:** reverse the order of occurrences
    - **queue:** preserve the order of occurrences
Simple applications of the ADT queue

- abbcccbba

stack

front back

top
Simple applications of the ADT queue

**Recognizing Palindromes**

`+isPal(in Str:String):boolean`

```java
aQueue.createQueue();
aStack.createStack();

for(i= 1 through length){
    nextChar=ith character of str;
    aQueue.enqueue(nextChar);
    aStack.push(nextChar);
}

characterAreEqual=true;

while(aQueue is not empty and characterAreEqual is true){
    queueFront=aQueue.dequeue();
    stackTop=aStack.pop();
    if(queueFront not equal to stackTop){
        characterAreEqual=false;
    }
}
```

`return characterAreEqual;`
Implementations of the ADT queue
Implementations of the ADT queue

- Implementations
  - Reference-based
  - Array-based
  - ADT list-based

- QtackInterface
  - Six operations

- QueueException
  - enqueue, dequeue, peek

```
QtackInterface
createQueue()
dequeueAll()
isEmpty()
enqueue(newItem)
dequeue()
peek()
```
Implementations of the ADT queue

A reference-based implementation

- A queue based on linear linked list with two external references
  - One to the front
  - One to the back

\[\text{firstNode} \quad \rightarrow \quad \text{2} \quad \rightarrow \quad \text{4} \quad \rightarrow \quad \text{1} \quad \rightarrow \quad \text{7} \quad \rightarrow \quad \text{lastNode}\]
Implementations of the ADT queue

- A reference-based implementation
  - A queue based on circular linked list with one external reference

![Diagram of a circular linked list queue](image)

- Insertion?
- Deletion?
Implementations of the ADT queue

- A reference-based implementation
  - Inserting an item into an empty queue
    - newNode.next = newNode;
    - lastNode = newNode;
  - Inserting an item into a nonempty queue
    - 1. newNode.next = lastNode.next;
    - 2. lastNode.next = newNode;
    - 3. lastNode = newNode;
Implementations of the ADT queue

- A reference-based implementation
  - Deleting an item from a queue

```
1. firstNode = lastNode.next;
2. lastNode.next = firstNode.next;
```
Implementations of the ADT queue

A array-based implementation

An array with the front and back are specified

rightward drift: cause a queue-full condition even though the queue contains few entries
Implementations of the ADT queue

> A array-based implementation

- A circular array eliminates the problem of rightward drift
Implementations of the ADT queue

- **A array-based implementation**
  - **A problem with the circular array implementation**
    - Front and back cannot be used to distinguish between queue-full and queue-empty conditions
      - Front passes back when the queue becomes empty

![Queue with single item](image)
![Delete item—queue becomes empty](image)
A array-based implementation

- A problem with the circular array implementation
  front and back cannot be used to distinguish between queue-full and queue-empty conditions
  - back catches up to front when the queue becomes full

Queue with single empty slot

Insert 9—queue becomes full
Implementations of the ADT queue

A array-based implementation

- A problem with the circular array implementation
  front and back cannot be used to distinguish between queue-full and queue-empty conditions

- Solution
  - To detect queue-full and queue-empty conditions
    - Keep a count of the queue items
  - To initialize the queue, set
    - front to 0
    - back to MAX_QUEUE – 1
    - count to 0
Implementations of the ADT queue

A array-based implementation

A problem with the circular array implementation

- front and back cannot be used to distinguish between queue-full and queue-empty conditions

Solution

- To detect queue-full and queue-empty conditions
  - Keep a count of the queue items
- To initialize the queue, set
  - front to 0
  - back to MAX_QUEUE – 1
  - count to 0

Inserting into a queue

```c
back = (back+1) % MAX_QUEUE;
items[back] = newItem;
++count;
```

Deleting from a queue

```c
front = (front+1) % MAX_QUEUE;
--count;
```
Implementations of the ADT queue

A array-based implementation

Variations of the array-based implementation

- Use a flag `full` to distinguish between the full and empty conditions
- Declare `MAX_QUEUE + 1` locations for the array items, but use only `MAX_QUEUE` of them for queue items

```
front = (back+1) % (MAX_QUEUE+1);
front = back;
```
Implementations of the ADT queue

An implementation that uses the ADT list

Variations of the array-based implementation

- the item in position 1 of a list list represents the front of the queue
  - dequeue() → list.remove(1)
  - peek() → list.get(1)

- the item at the end of the list represents the back of the queue, the following implementations can be used
  - enqueue(newItem) → list.add(list.size()+1, newItem)
The Queue interface

Derived from interface Collection

- Add methods:
  - element: retrieves, but does not remove head
  - offer: inserts element into queue
  - peek: retrieves, but does not remove head
  - poll: retrieves and removes head
  - remove: retrieves and removes head
The Dequeue ("deck" double-ended) queue interface

- Allows us to insert and delete from either end
  - Useful methods: addFirst, addLast, peekFirst, peekLast, getFirst, getLast, removeFirst, removeLast
  - function as both a stack and a queue
  - Example: text editor
    - Input characters using "stack" functionality: backspace event causes a pop. Output characters using "queue" functionality.
All of the implementations of the ADT queue mentioned are ultimately either

- Array based
- Reference based

Fixed size versus dynamic size

- A statically allocated array
  - Prevents the enqueue operation from adding an item to the queue if the array is full
- A resizable array or a reference-based implementation
  - Does not impose this restriction on the enqueue operation

Reference-based implementations

- A linked list implementation
  - More efficient
- The ADT list implementation
  - Simpler to write
A summary of position-oriented ADTs
A summary of position-oriented ADTs

- Position-oriented ADTs
  - List
  - Stack
  - Queue

- Stacks and queues
  - Only the end positions can be accessed

- Lists
  - All positions can be accessed
Stacks and queues are very similar

- Operations of stacks and queues can be paired off as
  - `createStack` and `createQueue`
  - `Stack isEmpty` and `queue isEmpty`
  - `push` and `enqueue`
  - `pop` and `dequeue`
  - `Stack peek` and `queue peek`

ADT list operations generalize stack and queue operations

- `length`
- `add`
- `remove`
- `get`
Simulation
Simulation: bank service

(a) time = 0

(b) time = 12

(c) time = 20

(d) time = 38
Simulation: bank service

- An event-driven simulation
  - Simulated time is advanced to the time of the next event
  - Events are generated by a mathematical model that is based on statistics and probability

- A time-driven simulation
  - Simulated time is advanced by a single time unit
  - The time of an event, such as an arrival or departure, is determined randomly and compared with a simulated clock
The bank simulation is concerned with

- **Arrival events**
  - Indicate the arrival at the bank of a new customer
  - External events: the input file specifies the times at which the arrival events occur

- **Departure events**
  - Indicate the departure from the bank of a customer who has completed a transaction
  - Internal events: the simulation determines the times at which the departure events occur
An event list is needed to implement an event-driven simulation

An event list

- Keeps track of arrival and departure events that will occur but have not occurred yet
- Contains at most one arrival event and one departure event
Summary
The definition of the queue operations gives the ADT queue first-in, first-out (FIFO) behavior.

A reference-based implementation of a queue uses either:
- A circular linked list
- A linear linked list with a head reference and a tail reference

An array-based implementation of a queue is prone to rightward drift:
- A circular array eliminates the problem of rightward drift

To distinguish between the queue-full and queue-empty conditions in a queue implementation that uses a circular array, you can:
- Count the number of items in the queue
- Use a full flag
- Leave one array location empty
Models of real-world systems often use queues

- The event-driven simulation in this chapter uses a queue to model a line of customers in a bank

Simulations

- Central to a simulation is the notion of simulated time
  - In a time-driven simulation
    Simulated time is advanced by a single time unit
  - In an event-driven simulation
    Simulated time is advanced to the time of the next event

- To implement an event-driven simulation, you maintain an event list that contains events that have not yet occurred