ADTs, Stacks and Queues

Readings - Chapter 6
Abstract Data Types
Abstract Data Types (ADTs)

- An abstract data type (ADT) is a mathematically specified entity that defines a set of its instances.

- An ADT specifies:
  - What data is stored
  - Interface - Operations on the data
    - manipulation
    - access
  - a set of axioms (preconditions and post conditions) that define what the operations do (not how)
Why Abstract Data Types (ADTs)

- serve as specifications of requirements for the building blocks of an algorithm
- encapsulate the data structure and the algorithms that works on the data structure
- separate issues of correctness and efficiency
The data stored are
- Stock
- Shares
- Price
- Buy/Sell

The operations supported are
- **New()**: ADT
- **Add(B:ADT, e:Element)**: ADT
- **Remove(B:ADT, e:Element)**: ADT
- **isIn(B:ADT, e:Element)**: boolean
ADT Example - Order Book of a Stock Trading System - Dynamic Set

- The data stored are:
  - Stock
  - Shares
  - Price
  - Buy/Sell

- The operations supported are:
  - **New()**: ADT
  - **Add**(B:ADT, e:Element): ADT
  - **Remove**(B:ADT, e:Element): ADT
  - **isIn**(B:ADT, e:Element): boolean
ADT Example - Order Book of a Stock Trading System - Dynamic Set

- The data stored are
  - Stock
  - Shares
  - Price
  - Buy/Sell

- The operations supported are
  - **New()**: ADT
  - **Add(B:ADT, e:Element)**: ADT
  - **Remove(B:ADT, e:Element)**: ADT
  - **isIn(B:ADT, e:Element)**: boolean
ADT Example - Order Book of a Stock Trading System - Dynamic Set

- The data stored are
  - Stock
  - Shares
  - Price
  - Buy/Sell

- The operations supported are
  - **New()**: ADT
  - **Add**(B:ADT, e:Element): ADT
  - **Remove**(B:ADT, e:Element): ADT
  - **isIn**(B:ADT, e:Element): boolean
Axioms that define the methods

- \( \text{isIn}(\text{New}(), e) - \text{false} \)
- \( \text{isIn}(\text{Add}(B, e), e) - \text{true} \)
- \( \text{isIn}(\text{Add}(B, f), e) - \text{isIn}(B, e) \text{ if } e \neq f \)
- \( \text{isIn}(\text{Remove}(B, e), e) - \text{false} \)
- \( \text{isIn}(\text{Remove}(B, f), e) = \text{isIn}(B, e), \text{ if } e \neq f \)
The Stack ADT

- Container of abstract objects that are inserted and removed according to the last-in-first-out (LIFO) principle
- Objects can be inserted at any time, but only the most-recently inserted object can be removed
  - Inserting an object - push
  - removing an object - pop
The Stack ADT - Specification

- **Data Object**
- **Operations**
  - `new()`:ADT - creates a new stack
  - `push(S:ADT, o:Object):ADT` - Inserts object `o` onto the top of stack `S`
  - `pop(S:ADT):ADT` - removes the top object of stack `S`; if the stack is empty an error occurs
The Stack ADT - Specification (contd)

- **Auxiliary operations**
  - $\text{top}(S:\text{ADT})$: *Object* - returns the top object of stack $S$ without removing it; if the stack is empty, an error occurs.
  - $\text{size}(S:\text{ADT})$: *integer* returns the number of objects in stack $S$.
  - $\text{isEmpty}(S:\text{ADT})$: *boolean* - indicates if stack $S$ is empty.

- **Axioms**
  - $\text{pop}(\text{push}(S, o)) = S$
  - $\text{top}(\text{push}(S, o)) = o$
# Stack - Example

<table>
<thead>
<tr>
<th>Method</th>
<th>Return Value</th>
<th>Stack Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>push(5)</td>
<td>–</td>
<td>(5)</td>
</tr>
<tr>
<td>push(3)</td>
<td>–</td>
<td>(5, 3)</td>
</tr>
<tr>
<td>size()</td>
<td>2</td>
<td>(5, 3)</td>
</tr>
<tr>
<td>pop()</td>
<td>3</td>
<td>(5)</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>false</td>
<td>(5)</td>
</tr>
<tr>
<td>pop()</td>
<td>5</td>
<td>()</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>true</td>
<td>()</td>
</tr>
<tr>
<td>pop()</td>
<td>null</td>
<td>()</td>
</tr>
<tr>
<td>push(7)</td>
<td>–</td>
<td>(7)</td>
</tr>
<tr>
<td>push(9)</td>
<td>–</td>
<td>(7, 9)</td>
</tr>
<tr>
<td>top()</td>
<td>9</td>
<td>(7, 9)</td>
</tr>
<tr>
<td>push(4)</td>
<td>–</td>
<td>(7, 9, 4)</td>
</tr>
<tr>
<td>size()</td>
<td>3</td>
<td>(7, 9, 4)</td>
</tr>
<tr>
<td>pop()</td>
<td>4</td>
<td>(7, 9)</td>
</tr>
<tr>
<td>push(6)</td>
<td>–</td>
<td>(7, 9, 6)</td>
</tr>
<tr>
<td>push(8)</td>
<td>–</td>
<td>(7, 9, 6, 8)</td>
</tr>
<tr>
<td>pop()</td>
<td>8</td>
<td>(7, 9, 6)</td>
</tr>
</tbody>
</table>

Stacks
Stack ADT specification in JAVA?

Interfaces

- **Interface** - collection of method declarations with no data and no bodies
  - write down the method names and the **parameters** (essentially the **types**)
  - when a **class** implements an **interface**, it must implement all the methods declared in the interface.
  - Separating **interface** and **implementation** is a useful programming technique.
Stack Interface in Java

- **java.util** has a built-in stack data structure class - **we will define our own interface and not use the built-in class**

```java
public interface Stack<E> {
    // accessor methods
    public int size(); // returns the number of objects in the stack
    public boolean isEmpty(); // returns true if the stack is empty, false otherwise
    public E top(); // returns the top object of the stack

    // manipulation/update methods
    public void push(E o); // inserts the object at the top of the stack
    public E pop(); // removes and returns the object at the top of the stack
}
```
Stack ADT specification in JAVA?

Exceptions

- unexpected events that occur during the execution of a program
  - useful for handling errors
- when an error (or an exceptional event is encountered, the Java code **throws** an exception
- Upon throwing an exception, the control flow exits from the current method and goes to the parent calling method
  - responsibility of handling the error can be delegated to the parent method
Exceptions - Example

```java
public void UpdateStudentScore(Student s) throws StudentNotFoundException {
    ...
    if (s is not part of CSL201)
        throw new StudentNotFoundException(s + "is not a student in CSL201");
    ...
}

public void UpdateClassScore() {
    ...
    try {
        CSL201.UpdateStudentScore(s);
    }
    catch (StudentNotFoundException e) {
        System.out.println(e);
    }
    ...
}
```
More Exceptions - Try and Catch

- try - guarded fragment of code that might thrown an exception when executed.
- catch - block of code to which control jumps on catching an exception. The block contains code to analyze the exception and apply an appropriate solution.

- If an exception is never caught in a method, it will propagate upwards along the sequence of method calls until the user sees it.

- Exceptions are essentially Java Classes
  - Section 2.4
Stack Interface in Java - Finally

- **java.util** has a built-in stack data structure class - we will define our own interface and not use the built-in class

```java
public interface Stack<E> {
    // accessor methods
    public int size(); // returns the number of objects in the stack
    public boolean isEmpty(); // returns true if the stack is empty, false otherwise
    public E top() throws StackEmptyException; // returns the top object of the stack

    // manipulation/update methods
    public void push(E o); // inserts the object at the top of the stack
    public E pop() throws StackEmptyException; // removes and returns the object at the top of the stack
}
```
Array-Based Stack Implementation in Java

- Simple way to implement a stack
  - specify the maximum size $N$ of our stack
- Add elements left to right
- variable $t$ keeps track of the top element of the array stack $S$

- array indices start at 0, so we initialize $t$ to -1
Array-Based Stack
Implementation in Java - 1

```java
public class ArrayStack<E> implements Stack<E> {
    /** Default array capacity. */
    public static final int CAPACITY = 1000; // default array capacity

    /** Generic array used for storage of stack elements. */
    private E[] data; // generic array used for storage

    /** Index of the top element of the stack in the array. */
    private int t = -1; // index of the top element in stack

    /** Constructs an empty stack using the default array capacity. */
    public ArrayStack() { this(CAPACITY); } // constructs stack with default capacity

    /** Constructs and empty stack with the given array capacity. *
     * @param capacity length of the underlying array *
     */
    public ArrayStack(int capacity) { // constructs stack with given capacity
        data = (E[]) new Object[capacity];
    }
}
```
/**
 * Returns the number of elements in the stack.
 * @return number of elements in the stack
 */

public int size() { return (t + 1); }

/**
 * Tests whether the stack is empty.
 * @return true if the stack is empty, false otherwise
 */

public boolean isEmpty() { return (t == -1); }

/**
 * Inserts an element at the top of the stack.
 * @param e the element to be inserted
 * @throws StackFullException if the array storing the elements is full
 */

public void push(E e) throws StackFullException {
    if (size() == data.length) throw new StackFullException("Stack is full");
    data[++t] = e;  // increment t before storing
}
/**
 * Returns, but does not remove, the element at the top of the stack.
 * @return top element in the stack (or null if empty)
 * @throws StackEmptyException if the array storing the stack is empty
 */
public E top() {
    if (isEmpty()) throw new StackEmptyException("Stack is empty");
    return data[t];
}

/**
 * Removes and returns the top element from the stack.
 * @return element removed
 * @throws StackEmptyException if the array storing the stack is empty
 */
public E pop() {
    if (isEmpty()) throw new StackEmptyException("Stack is empty");
    E answer = data[t];
    data[t] = null; // dereference to help garbage collection
    t--;
    return answer;
}
Array-Based Stack
Performance and Limitations

- **Performance**
  - implementation is simple and efficient
    - methods performed in $O(1)$
  - Space complexity - $O(N)$

- **Limitations**
  - maximum capacity of the array has to be specified a priori
    - too large - wastage of space
    - too small - insufficient for a given application
  - **StackFullException** is specific to this implementation
  - **StackEmptyException** is required by the interface
A Growable Array-Based Stack

- On StackFullException, replace the array $S$ with a larger one and continue processing the push operations.

Algorithm push(o)
if size()=N then
  A new array of length $f(N)$
for i ← 0 to N-1
  $A[i] ← S[i]$
$S ← A$
t ← t+1
$S[t] ← o$

- How large should the new array be?
  - tight strategy - add a constant $f(N) = N+c$
  - growth strategy - double up $f(N) = 2N$
Comparing Tight and Growth Strategies

- **cost model**
  - **regular push** - when array capacity is not exceeded; adds one element
    - costs 1 unit
  - **special push** - when a new array is created and elements of the old array are copied before adding one element.
    - costs $f(N) + N + 1$ units
Tight Strategy (c=4)

- start with an array size of 0

<table>
<thead>
<tr>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b</td>
</tr>
<tr>
<td>a b c</td>
</tr>
<tr>
<td>a b c d</td>
</tr>
<tr>
<td>a b c d e</td>
</tr>
<tr>
<td>a b c d e f</td>
</tr>
<tr>
<td>a b c d e f g</td>
</tr>
<tr>
<td>a b c d e f g h</td>
</tr>
<tr>
<td>a b c d e f g h i</td>
</tr>
<tr>
<td>a b c d e f g h i j</td>
</tr>
<tr>
<td>a b c d e f g h i j k</td>
</tr>
<tr>
<td>a b c d e f g h i j k l</td>
</tr>
</tbody>
</table>

Stacks

4 + 1
1
1
1
8 + 4 + 1
1
1
1
1 2 + 8 + 1
1
1
1
1

cost of special push 2n+5
for a total of n pushes

cost is O(n^2/c)
Growth Strategy (double the size)

- start with an array size of 0

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a b c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a b c d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a b c d e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a b c d e f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a b c d e f g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a b c d e f g h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a b c d e f g h i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a b c d e f g h i j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a b c d e f g h i j k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a b c d e f g h i j k l</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 + 0 + 1</td>
<td>2 + 1 + 1</td>
<td>4 + 2 + 1</td>
<td>8 + 4 + 1</td>
<td>16 + 8 + 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Stacks

cost of special push 3n+1
for a total of n pushes
cost is O(n)
public class LinkedStack<E> implements Stack<E> {

    /** The primary storage for elements of the stack */
    private SinglyLinkedList<E> list = new SinglyLinkedList<>(); // an empty list

    /** Constructs an initially empty stack. */
    public LinkedStack() { } // new stack relies on the initially empty list

    /**
     * Returns the number of elements in the stack.
     * @return number of elements in the stack
     */
    public int size() { return list.size(); }
/**
 * Tests whether the stack is empty.
 * @return true if the stack is empty, false otherwise
 */
public boolean isEmpty() { return list.isEmpty(); }

/**
 * Inserts an element at the top of the stack.
 * @param element the element to be inserted
 */
public void push(E element) { list.addFirst(element); }
public E top() {
    if (isEmpty()) throw new StackEmptyException ("Stack is empty");
    return list.first();
}

public E pop() {
    if (isEmpty()) throw new StackEmptyException ("Stack is empty");
    return list.removeFirst();
}
Applications of Stacks

Direct applications

- Page-visited history in a Web browser
- Undo sequence in a text editor
- Chain of method calls in the Java Virtual Machine

Indirect applications

- Auxiliary data structure for algorithms
- Component of other data structures
Stack Applications -
Chain of Method Calls in JVM

- The Java Virtual Machine (JVM) keeps track of the chain of active methods with a stack.
- When a method is called, the JVM pushes on the stack a frame containing:
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a method ends, its frame is popped from the stack and control is passed to the method on top of the stack.
- Allows for recursion

```java
main() {
  int i = 5;
  foo(i);
}

foo(int j) {
  int k;
  k = j+1;
  bar(k);
}

bar(int m) {
  ...
}
```

© 2014 Goodrich, Tamassia, Goldwasser
14 – 3 * 2 + 7 = (14 – (3 * 2)) + 7

Operator precedence
* has precedence over +/-

Associativity
operators of the same precedence group evaluated from left to right
Example: (x – y) + z rather than x – (y + z)

Idea: push each operator on the stack, but first pop and perform higher and equal precedence operations.
Two stacks:
- opStk holds operators
- valStk holds values
- Use $ as special “end of input” token with lowest precedence

Algorithm **doOp()**

```plaintext
x ← valStk.pop();
y ← valStk.pop();
op ← opStk.pop();
valStk.push( y op x )
```

Algorithm **repeatOps( refOp )**:

```plaintext
while ( valStk.size() > 1 ∧ prec(refOp) ≤ prec(opStk.top()) )
doOp()
```

**Algorithm EvalExp()**

Input: a stream of tokens representing an arithmetic expression (with numbers)
Output: the value of the expression

```plaintext
while there’s another token z
  if isNumber(z) then
    valStk.push(z)
  else
    repeatOps(z);
  opStk.push(z)
repeatOps($);
return valStk.top()
```
Algorithm on an Example Expression

14 ≤ 4 − 3 * 2 + 7

Operator ≤ has lower precedence than +/-
Computing Spans (not in book)

- Using a stack as an auxiliary data structure in an algorithm
- Given an array $X$, the span $S[i]$ of $X[i]$ is the maximum number of consecutive elements $X[j]$ immediately preceding $X[i]$ and such that $X[j] \leq X[i]$
- Spans have applications to financial analysis
  - E.g., stock at 52-week high
Quadratic Algorithm

Algorithm spans1(X, n)

Input array X of n integers
Output array S of spans of X

S ← new array of n integers

for i ← 0 to n - 1 do

s ← 1

while s ≤ i ∧ X[i - s] ≤ X[i] do

s ← s + 1

S[i] ← s

return S
Computing Spans with a Stack

- We keep in a stack the indices of the elements visible when “looking back”
- We scan the array from left to right
  - Let \( i \) be the current index
  - We pop indices from the stack until we find index \( j \) such that \( X[i] < X[j] \)
  - We set \( S[i] \leftarrow i - j \)
  - We push \( i \) onto the stack
Efficient Algorithm

- Each index of the array
  - Is pushed into the stack exactly once
  - Is popped from the stack at most once
- The statements in the while-loop are executed at most $n$ times
- Algorithm $spans2$ runs in $O(n)$ time

Algorithm $spans2(X, n)$

```plaintext
S ← new array of $n$ integers
A ← new empty stack

for $i ← 0$ to $n - 1$ do
    while (¬A.isEmpty() ∧ $X[A.top()] ≤ X[i]$ ) do
        A.pop()
        if A.isEmpty() then
            $S[i] ← i + 1$
        else
            $S[i] ← i - A.top()$
        A.push($i$)
    return $S$
```

© 2014 Goodrich, Tamassia, Goldwasser
Efficient Algorithm

Algorithm \textit{spans2}(X, n)

\begin{algorithmic}
\State $S \leftarrow \text{new array of } n \text{ integers}$
\State $A \leftarrow \text{new empty stack}$
\For{$i \leftarrow 0 \text{ to } n - 1$}
\While{($\neg A.\text{isEmpty}() \land X[A.\text{top}()] \leq X[i]$)}
\State $A.\text{pop}()$
\EndWhile
\If{$A.\text{isEmpty}()$}
\State $S[i] \leftarrow i + 1$
\Else
\State $S[i] \leftarrow i - A.\text{top}()$
\EndIf
\State $A.\text{push}(i)$
\EndFor
\State \Return $S$
\end{algorithmic}

© 2014 Goodrich, Tamassia, Goldwasser
import java.io.*;
import java.util.Scanner;

public class FileScan{
    public static void main(String[] args) throws IOException {
        Scanner s = null;
        try {
            s = new Scanner(new BufferedReader(new FileReader("test.txt")));

            while (s.hasNext()) {
                System.out.println(s.next());
            }
        } finally {
            if (s != null) {
                s.close();
            }
        }
    }
}
Queues

Readings - 6.2 and 6.3
The Queue ADT

- The insertion and removal routines of the Queue ADT follow the first-in-first-out (FIFO) principle.
- Elements may be inserted at any time, but only the element which has been in the queue the longest may be removed.
  - Inserting an object (at the rear) – enqueue
  - Removing an object (from the front) - dequeue
The Queue ADT - Specification

- Data Object
- The queue supports the following fundamental methods:
  - `New()`:ADT – creates an empty queue
  - `Enqueue(Q:ADT, o:element):ADT` – inserts the object o at the rear of the queue
  - `Dequeue(Q:ADT):element` – removes the object from the front of the queue; an error occurs if the queue is empty
  - `First(Q:ADT):element` – returns, but does not remove the object at the front of the queue; an error occurs if the queue is empty.
The Queue ADT - Specification

- **Auxiliary methods**
  - `Size(Q:QDT): integer` – returns the number of elements in the queue
  - `isEmpty(Q:ADT): boolean` – indicates whether the queue is empty

- **Axioms**
  - `Front(Enqueue(New(), v)) = v`
  - `Dequeue(Enqueue(New(), v)) = New()`
  - `Front(Enqueue(Enqueue(Q,w),v)) = Front(Enqueue(Q,w))`
  - `Dequeue(Enqueue(Enqueue(Q,w), v)) = Enqueue(Dequeue(Enqueue(Q,w),v))`
### Example

<table>
<thead>
<tr>
<th>Operation</th>
<th>Output</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>enqueue(5)</td>
<td>–</td>
<td>(5)</td>
</tr>
<tr>
<td>enqueue(3)</td>
<td>–</td>
<td>(5, 3)</td>
</tr>
<tr>
<td>dequeue()</td>
<td>5</td>
<td>(3)</td>
</tr>
<tr>
<td>enqueue(7)</td>
<td>–</td>
<td>(3, 7)</td>
</tr>
<tr>
<td>dequeue()</td>
<td>3</td>
<td>(7)</td>
</tr>
<tr>
<td>first()</td>
<td>7</td>
<td>(7)</td>
</tr>
<tr>
<td>dequeue()</td>
<td>7</td>
<td>()</td>
</tr>
<tr>
<td>dequeue()</td>
<td>null</td>
<td>()</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>true</td>
<td>()</td>
</tr>
<tr>
<td>enqueue(9)</td>
<td>–</td>
<td>(9)</td>
</tr>
<tr>
<td>enqueue(7)</td>
<td>–</td>
<td>(9, 7)</td>
</tr>
<tr>
<td>size()</td>
<td>2</td>
<td>(9, 7)</td>
</tr>
<tr>
<td>enqueue(3)</td>
<td>–</td>
<td>(9, 7, 3)</td>
</tr>
<tr>
<td>enqueue(5)</td>
<td>–</td>
<td>(9, 7, 3, 5)</td>
</tr>
<tr>
<td>dequeue()</td>
<td>9</td>
<td>(7, 3, 5)</td>
</tr>
</tbody>
</table>
public interface Queue<E> {
    // accessor methods
    int size(); // returns the number of elements in the queue
    boolean isEmpty(); // returns true if the queue is empty, false otherwise

    // manipulation methods
    void enqueue(E e); // adds the element to the rear of the queue
    E first() throws QueueEmptyException; // returns the first element of the queue
    E dequeue() throws QueueEmptyException; // removes and returns the first element of the queue
}
Array-based Queue

- Use an array of size $N$ in a circular fashion
- Two variables keep track of the front and size:
  - $f$ index of the front element
  - $sz$ number of stored elements
- When the queue has fewer than $N$ elements, array location $r = (f + sz) \mod N$ is the first empty slot past the rear of the queue.

![Diagram showing normal and wrapped-around configurations of a queue]

© 2014 Goodrich, Tamassia, Goldwasser
Queue Operations

- We use the modulo operator (remainder of division)

```
Algorithm size()
return sz

Algorithm isEmpty()
return (sz == 0)
```

![](image.png)
Operation enqueue throws an exception if the array is full – Implementation specific exception

Algorithm `enqueue(o)`

```
if size() = N then
    throw QueueFullException
else
    r ← (f + sz) mod N
    Q[r] ← o
    sz ← (sz + 1)
```

© 2014 Goodrich, Tamassia, Goldwasser
Note that operation dequeue throws QueueEmptyException if the queue is empty.

Algorithm `dequeue()`

```
if isEmpty() then
    throw QueueEmptyException
else
    o ← Q[f]
    f ← (f + 1) mod N
    sz ← (sz - 1)
    return o
```
public class ArrayQueue<E> implements Queue<E> {
    // instance variables
    public static final int CAPACITY = 1000; // default array capacity
    private E[] data; // generic array used for storage
    private int f = 0; // index of the front element
    private int sz = 0; // current number of elements

    // constructors
    public ArrayQueue() {this(CAPACITY);} // constructs queue with default capacity
    public ArrayQueue(int capacity) {
        data = (E[]) new Object[capacity]; // safe cast; compiler may give warning
    }

    // methods
    /**
     * Returns the number of elements in the queue.
     * @return number of elements in the queue
     */
    public int size() { return sz; }

    public boolean isEmpty() { return (sz == 0); } // Tests whether the queue is empty.
/**
 * Inserts an element at the rear of the queue.
 * This method runs in O(1) time.
 * @param e   new element to be inserted
 * @throws QueueFullException if the array storing the elements is full
 */

public void enqueue(E e) throws QueueFullException {
    if (sz == data.length) throw new QueueFullException("Queue is full");
    int avail = (f + sz) % data.length; // use modular arithmetic
    data[avail] = e;
    sz++;
}

/**
 * Returns, but does not remove, the first element of the queue.
 * @return the first element of the queue
 * @throws QueueEmptyException if the array storing the elements is empty
 */

public E first() throws QueueEmptyException {
    if (isEmpty()) throws new QueueEmptyException();
    return data[f];
}
/**
 * Removes and returns the first element of the queue.
 * @return element removed
 * @throws QueueEmptyException if the queue is empty
 */

public E dequeue() throws QueueEmptyException {
    if (isEmpty()) throw new QueueEmptyException();
    E answer = data[f];
    data[f] = null;  // dereference to help garbage collection
    f = (f + 1) % data.length;
    sz--;  
    return answer;
}
Applications of Queues

- **Direct applications**
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming

- **Indirect applications**
  - Auxiliary data structure for algorithms
  - Component of other data structures
We can implement a round robin scheduler using a queue \( Q \) by repeatedly performing the following steps:

1. \( e = Q\text{.dequeue()} \)
2. Service element \( e \)
3. \( Q\text{.enqueue}(e) \)
Double-Ended Queue - Deque

- Deque supports insertion and deletion from the front and back of the queue
- The deque ADT supports the following fundamental methods
  - \textbf{InsertFirst}(Q:\text{ADT}, e:element):ADT – inserts e at the beginning of the deque
  - \textbf{InsertLast}(Q:\text{ADT}, e:element):ADT – inserts e at the end of the deque
  - \textbf{RemoveFirst}(Q:\text{ADT}):ADT – removes the first element
  - \textbf{RemoveLast}(Q:\text{ADT}):ADT – removes the last element
  - \textbf{First}(Q:\text{ADT}): element – returns the first element
  - \textbf{Last}(Q:\text{ADT}): element – returns the last element
Deque implementations

- **Singly linked list**
  - head

- **Doubly linked list**
  - constant time operations

Queues