Chapter 18

Stacks and Queues

18.1 Stacks

The stackabstract data type allows access to only one element—the one most recently added. This location is referred to as the top of the stack.

 Consider how a stack of books might be placed into and removed from a cardboard box, assuming you can only move one book at a time. The most readily available book is at the top of the stack. For example, if you add two books—Book 1 and then Book 2—into the box, the most recently added book (Book 2) will be at the top of the stack. If you continue to stack books on top of one another until there are five, Book 5 will be at the top of the stack. To get to the least recently added book (Book 1), you first remove the topmost four books (Book 5, Book 4, Book 3, Book 2) — one at a time. Then the top of the stack would be Book 1 again.



Stack elements are added and removed in a last in first out (LIFO) manner. The most recent element added to the collection will be the first element to be removed from the collection. Sometimes, the only data that is readily needed is the most recently accessed one. The other elements, if needed later, will be in the reverse order of when they were pushed. Many real world examples of a stack exist. In a physical sense, there are stacks of books, stacks of cafeteria trays, and stacks of paper in a printer’s paper tray. The sheet of paper on the top is the one that will get used next by the printer.

For example, a stack maintains the order of method calls in a program. If main calls function1, that method calls function2, which in turn calls function3. Where does the program control go to when function3 is finished? After function3 completes, it is removed from the stack as the most recently added method. Control then returns to the method that is at the new top of the stack  function2.

 Here is a view of the stack of function calls shown in a thread named main. This environment (Eclipse) shows the first method (main) at the bottom of the stack. main will also be the last method popped as the program finishes — the *first* method called is the *last* one to execute. At all other times, the method on the top of the stack is executing. When a method finishes, it can be removed and the method that called it will be the next one to be removed form the stack of method calls.



The program output indicates the last in first out (or first in last out) nature of stacks:

Four method calls are on the stack

Two about to end

One about to end

main about to end

Another computer-based example of a stack occurs when a compiler checks the syntax of a program. For example, some compilers first check to make sure that [ ], { }, and ( ) are balanced properly. Thus, in a Java class, the final } should match the opening {. Some compilers do this type of symbol balance checking first (before other syntax is checked) because incorrect matching could otherwise lead to numerous error messages that are not really errors. A stack is a natural data structure that allows the compiler to match up such opening and closing symbols (an algorithm will be discussed in detail later).

A Stack Interface to capture the ADT

Here are the operations usually associated with a stack. (As shown later, others may exist):

* push place a new element at the "top" of the stack
* pop remove the top element and return a reference to the top element
* isEmpty return true if there are no elements on the stack
* peek return a reference to the element at the top of the stack

Programmers will sometimes add operations and/or use different names. For example, in the past, Sun programmers working on Java collection classes have used the name empty rather than isEmpty. Also, some programmers write their stack class with a pop method that does not return a reference to the element at the top of the stack. Our pop method will modify and access the state of stack during the same message.

 Again, a Java interface helps specify Stack as an abstract data type. For the discussion of how a stack behaves, consider that LinkedStack (a collection class) implements the OurStack interface, which is an ADT specification in Java.

**import** java.util.EmptyStackException;

**public** **interface** OurStack<E> {

/\*\* Check if the stack is empty to help avoid popping an empty stack.

 \* **@returns** true if there are zero elements in this stack.

 \*/

 **public** **boolean** isEmpty();

/\*\* Put element on "top" of this Stack object.

 \* **@param** The new element to be placed at the top of this stack.

 \*/

 **public** **void** push(E element);

/\*\* Return reference to the element at the top of this stack.

 \* **@returns** A reference to the top element.

 \* **@throws** EmptyStackException if the stack is empty.

 \*/

 **public** E peek() **throws** EmptyStackException;

/\*\* Remove element at top of stack and return a reference to it.

 \* **@returns** A reference to the most recently pushed element.

 \* **@throws** EmptyStackException if the stack is empty.

 \*/

 **public** E pop() **throws** EmptyStackException;

}

You might need a stack of integers, or a stack of string values, or a stack of some new class of Token objects (pieces of source code). One solution would be to write and test a different stack class for each new class of object or primitive value that you want to store. This is a good reason for developing an alternate solution— a generic stack.

 The interface to be implemented specifies the operations for a stack class. It represents the *abstract* specification. There is no particular data storage mentioned and there is no code in the methods. The type parameter <E> and return types E indicate that the objects of the implementing class will store any type of element. For example, push takes an E parameter while peek and pop return an E reference.

 The following code demonstrates the behavior of the stack class assuming it is implemented by a class named LinkedStack.

// Construct an empty stack that can store any type of element

OurStack stackOfStrings<String> = new LinkedStack<String>();

// Add three string values to the stack

stackOfStrings.push("A");

stackOfStrings.push("B");

stackOfStrings.push("C");

// Show each element before each element is removed in a LIFO order

while (! stackOfStrings.isEmpty()) {

 // Print the value of the element at the top as it is removed

 System.out.print(stackOfStrings.pop() + " ");

}

Output

C B A

Self-Check

18-1 Write the output generated by the following code:

OurStack<String> aStack = new LinkedStack<String> ();

aStack.push("x");

aStack.push("y");

aStack.push("z");

while (! aStack.isEmpty()) {

 out.println(aStack.pop());

}

18-2 Write the output generated by the following code:

OurStack<Character> opStack = new OurLinkedStack<Character>();

out.println(opStack.isEmpty());

opStack.push('>');

opStack.push('+');

opStack.push('<');

out.print(opStack.peek());

out.print(opStack.peek()); // careful

out.print(opStack.peek());

18-3 Write the output generated by the following code:

OurStack<Integer> aStack = new OurLinkedStack<Integer>();

aStack.push(3);

aStack.push(2);

aStack.push(1);

out.println(aStack.isEmpty());

out.println(aStack.peek());

aStack.pop();

out.println(aStack.peek());

aStack.pop();

out.println(aStack.peek());

aStack.pop();

out.println(aStack.isEmpty());

18.2 Stack Application: Balanced Symbols

Some compilers perform symbol balance checking before checking for other syntax errors. For example, consider the following code and the compile time error message generated by a particular Java compiler (your compiler may vary).

public class BalancingErrors

 public static void main(String[] args) {

BalancingErrors.java:1: '{' expected

public class BalancingErrors

 ^

 int x = p;

 int y = 4;

 in z = x + y;

 System.out.println("Value of z = " + z);

 }

}

Notice that the compiler did not report other errors, one of which is on line 3. There should have been an error message indicating p is an unknown symbol. Another compile time error is on line 5 where z is incorrectly declared as an in not int. If you fix the first error by adding the left curly brace on a new line 1 you will see these other two errors.

public class BalancingErrors **{ // <- add an opening curly brace**

 public static void main(String[] args) {

 int x = p;

 int y = 4;

 in z = x + y;

 System.out.println("Value of z = " + z);

 }

}

BalancingErrors.java:3: cannot resolve symbol

symbol : variable p

location: class BalancingErrors

 int x = p;

 ^

BalancingErrors.java:5: cannot resolve symbol

symbol : class in

location: class BalancingErrors

 in z = x + y;

 ^

2 errors

This behavior could be due to a compiler that first checks for balanced { and } symbols before looking for other syntax errors.

 Now consider how a compiler might use a stack to check for balanced symbols such as ( ) , { }, and [ ]. As it reads the Java source code, it will only consider opening symbols: ( { [, and closing symbols: ) } ]. If an opening symbol is found in the input file, it is pushed onto a stack. When a closing symbol is read, it is compared to the opener on the top of the stack. If the symbols match, the stack gets popped. If they do not match, the compiler reports an error to the programmer. Now imagine processing these tokens, which represent only the openers and closers in a short Java program: {{([])}}. As the first four symbols are read — all openers — they get pushed onto the stack.

*Java source code starts as:* {{([])}}

 [

 *push the first four opening symbols with* **[** *at the top. Still need to read* **])}}**

 (

 {

 {

The next symbol read is a closer: "]". The "[" would be popped from the top of the stack and compared to "]". Since the closer matches the opening symbol, no error would be reported. The stack would now look like this with no error reported:

 (

 *pop* [ *which matches* ]*. There is no error. Still need to read* **) }}**

 {

 {

The closing parenthesis ")" is read next. The stack gets popped again. Since the symbol at the top of the stack "(" matches the closer ")", no error needs to be reported. The stack would now have the two opening curly braces.

 *pop* ( *which matches). There is no error. Still need to read***}}**

{

 {

The two remaining closing curly braces would cause the two matching openers to be popped with no errors. It is the last-in-first-out nature of stacks that allows the first pushed opener "{" to be associated with the last closing symbol "}" that is read.

 Now consider Java source code with only the symbols (]. The opener "(" is pushed. But when the closer "]" is encountered, the popped symbol "(" does not match "]" and an error could be reported. Here are some other times when the use of a stack could be used to help detect unbalanced symbols:

1. If a closer is found and the stack is empty. For example, when the symbols are {}}. The opening { is pushed and the closer "}" is found to be correct. However when the second } is encountered, the stack is empty. There is an error when } is discovered to be an extra closer (or perhaps { is missing).
2. If all source code is read and the stack is not empty, an error should be reported. This would happen with Java source code of {{([])}. In this case, there is a missing right curly brace. Most symbols are processed without error. At the end, the stack *should* be empty. Since the stack is *not* empty, an error should be reported to indicate a missing closer.

This algorithm summarzies the previous actions.

Make an empty stack **s**

For each token (operators or operands) in the infix expression

 if token is an operand (valid integers only)

 s.push(operand)

 if token is an operator such as + \* - /

 right = s.pop()

 left = s.pop()

 push the value of the operator applied to the left and right

Self-Check

18-4 Write the errors generated when the algorithm above processes the following input file:

public class Test2 {

 public static void main(String[) args) {

 System.out.println();

 )) {

18.3 FIFO Queues

A first-in, first-out (FIFO) **queue**  pronounced “Q”  models a waiting line. Whereas stacks add and remove elements at one location — the top — queues add and remove elements at different locations. New elements are added at the back of the queue. Elements are removed from the front of the queue.

 front back

|  |
| --- |
| 4th |
| 3rd |
| 2nd |
| 1st |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1st | 2nd | 3rd | 4thtop**Stack** |

 **Queue**

Whereas stacks mimic LIFO behavior, queues mimic a first in first out (FIFO) behavior. So, for example, the queue data structure models a waiting line such as people waiting for a ride at an amusement park. The person at the front of the line will be the first person to ride. The most recently added person must wait for all the people in front of them to get on the ride. With a FIFO queue, the person waiting longest in line is served before all the others who have waited less time[[1]](#footnote-1).

 Another example of queue behavior can be found when several documents need to be printed at a shared printer. Consider three students, on the same network, trying to print one document each. Who gets their document printed first? If a FIFO queue is being used to store incoming print requests, the student whose request reached the print queue first will get printed ahead of the others. Now assume that the printer is busy and the print queue gets a print request from student #3 while a ducument is printing. The print queue would look something like this:

 front & back

|  |  |  |
| --- | --- | --- |
|  |  | #3 |

In this case the queue’s front element is also at the back end of the queue. The queue contains one element. Now add another request from student #1, followed by another request from student #2 for printing, and the print queue would look like this:

 front back

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | #3 | #1 | #2 |

Student #1 and student #2 requests were added to the back of queue. The print requests are stored in the order in which they arrived. As the printer prints documents, the document will be removed from the front. Once the printer has printed the current document, the document for student #3 will then be removed. Then the state of the queue will now look like this:

 front back

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | #1 | #2 |  |

A Queue Interface  Specifying the methods

There is no universally agreed upon set of operations; however the following is a reasonable set of operations for a FIFO Queue ADT.

* isEmpty Return true only when there are zero elements in the queue
* add Add an element at the back of the queue
* peek Return a reference to the element at the front of the queue
* remove Return a reference to the element at the front and remove the element

This leads to the following interface for a queue that can store any class of object.

**public** **interface** OurQueue<E> {

 /\*\*

 \* Find out if the queue is empty.

 \* **@returns** true if there are zero elements in this queue.

 \*/

 **public** **boolean** isEmpty();

 /\*\*

 \* Add element to the "end" of this queue.

 \* **@param** newEl element to be placed at the end of this queue.

 \*/

 **public** **void** add(E newEl);

 /\*\*

 \* Return a reference to the element at the front of this queue.

 \* **@returns** A reference to the element at the front.

 \* **@throws** NoSuchElementException if this queue is empty.

 \*/

 **public** E peek();

 /\*\*

 \* Return a reference to front element and remove it.

 \* **@returns** A reference to the element at the front.

 \* **@throws** NoSuchElementException if this queue is empty.

 \*/

 **public** E remove();

}

The following code demonstrates the behavior of the methods assuming OurLinkedQueue implements interface OurQueue:

OurQueue<Integer> q = **new** OurLinkedQueue<Integer>();

q.add(6);

q.add(2);

q.add(4);

**while** (!q.isEmpty()) {

 System.out.println(q.peek());

 q.remove();

}

Output

6 2 4

Self-Check

18-5 Write the output generated by the following code.

OurQueue<String> strinqQueue = **new** OurLinkedQueue<String>();

strinqQueue.add("J");

strinqQueue.add("a");

strinqQueue.add("v");

strinqQueue.add("a");

**while** (!strinqQueue.isEmpty()) {

 System.out.print(strinqQueue.remove());

}

18-6 Write the output generated by the following code until you understand what is going on.

OurQueue<String> strinqQueue = **new** OurLinkedQueue<String>();

strinqQueue.add("first");

strinqQueue.add("second");

**while** (! strinqQueue.isEmpty()) {

 System.out.println(strinqQueue.peek());

}

18-7 Write code that displays a message to indicate if each integer in a queue named intQueue is even or odd. The queue must remain intact after you are done. The queue is initialized with random integers in the range of 0 through 99.

OurQueue<Integer> intQueue = **new** OurLinkedQueue<Integer>();

Random generator = **new** Random();

**for**(**int** j = 1; j <= 7; j++) {

 intQueue.add(generator.nextInt(100));

}

// Your solution goes here

Sample Output (output varies since random integers are added)

28 is even

72 is even

4 is even

37 is odd

94 is even

98 is even

33 is odd

18.4 Queue with a Linked Structure

We will implement interface OurQueue with a class that uses a singly linked structure. There are several reasons to choose a linked structure over an array:

* It is easier to implement. (A programming project explains the trickier array-based implementation).
* The Big-O runtime of all algorithms is as efficient as if an array were used to store the elements. All algorithms can be O(1).
* An array-based queue would have add and remove methods, which will occasionally run O(n) rather than O(1). This occurs whenever the array capacity needs to be increased or decreased.
* It provides another good example of implementing a data structure using the linked structure introduced in the previous chapter.

Elements are removed from the "front" of a queue. New elements are added at the back of the queue. Both "ends" of the queue are frequently accessed. Therefore, this implementation of OurQueue will use two external references. Only one external reference to the front is required. However, this would make for O(n) behavior during add messages, since a loop would need to sequence through all elements before reaching the end. With only a reference to the front, all elements must be visited to find the end of the list before one could be added. Therefore, an external reference named back will be maintained in addition to front. This will allow add to be O(1). An empty OurLinkedQueue will look like this:



After q.add("First"), a queue of size 1 will look like this:



After q.add("Second"), the queue of size 2 will look like this:



This test method shows the changing state of a queue that follows the above pictures of memory.

 @Test

 **public** **void** testAddAndPeek() {

 OurQueue<String> q = **new** OurLinkedQueue<String>();

 assertTrue(q.isEmpty()); // front == null

 q.add("first");

 assertEquals("first", q.peek()); // front.data is "first"

 assertFalse(q.isEmpty());

 q.add("second"); // Change back, not front

 // Front element should still be the same

 assertEquals("first", q.peek());

 }

The first element is accessible as front.data. A new element is added by storing a reference to the new node into back.next and adjusting back to reference the new node at the end.

 Here is the beginning of class OurLinkedQueue that once again uses a private inner Node class to store the data along with a link to the next element in the collection. There are two instance variables to maintain both ends of the queue.

**public** **class** OurLinkedQueue<E> **implements** OurQueue<E> {

 **private** **class** Node {

 **private** E data;

 **private** Node next;

 **public** Node() {

 data = **null**;

 next = **null**;

 }

 **public** Node(E elementReference, Node nextReference) {

 data = elementReference;

 next = nextReference;

 }

 } // end class Node

 // External references to maintain both ends of a Queue

 **private** Node front;

 **private** Node back;

 /\*\*

 \* Construct an empty queue (no elements) of size 0.

 \*/

 **public** OurLinkedQueue() {

 front = **null**;

 back = **null**;

 }

 /\*\* Find out if the queue is empty.

 \* **@returns** true if there are zero elements in this queue.

 \*/

 **public** **boolean** isEmpty() {

 **return** front == **null**;

 }

 // More methods to be added . . .

}

This implementation recognizes an empty queue when front is null.

## add

The add operation will first check for the special case of adding to an empty queue. The code to add to a non-empty queue is slightly different. If the queue is empty, the external references front and back are both null.



In the case of an empty queue, the single element added will be at front of the queue and also at the back of the queue. So, after building the new node, front and back should both refer to the same node. Here is a before and after picture made possible with the code shown.

 // Build a node to be added at the end. A queue can

 // grow as long as the computer has enough memory.

 // With a linked structure, resizing is not necessary.

 **if** (**this**.isEmpty()) {

 front = **new** Node(element, **null**);

 back = front;

 }

When an add messages is sent to a queue that is not empty, the last node in the queue must be made to refer to the node with the new element. Although front must remain the same during add messages, back must be changed to refer the new element at the end.

 **else** {

 back.next = **new** Node(element);

 back = back.next;

 }

There are several viable variations of how algorithms could be implemented when a linked structure is used to store the collection of elements. The linked structure used here always maintains two external references for the front and back of the linked structure. This was done so add is O(1) rather than O(n). In summary, the following code will generate the linked structure shown below.

OurQueue<String> q = **new** OurLinkedQueue<String>();

q.add("first");

q.add("second");

q.add("third");



Self-Check

18-8 Draw a picture of what the memory would look like after this code has executed

 OurQueue<Double> q1 = **new** OurLinkedQueue<Double>();

 q1.add(5.6));

 q1.add(7.8));

18-9 Implement a toString method for OurLinkedQueue so this assertion would pass after the code in the previous self-check question:

 assertEquals("[a, b]", q2.toString());

## peek

The peek method throws a NoSuchElementException if the queue is empty. Otherwise, peek returns a reference to the element stored in front.data.

 /\*\*

 \* Return a reference to the element at the front of this queue.

 \* **@returns** A reference to the element at the front.

 \* **@throws** NoSuchElementException if this queue is empty.

 \*/

 **public** E peek() {

 **if** (**this**.isEmpty())

 **throw** **new** java.util.NoSuchElementException();

 **else**

 **return** front.data;

 }

The next two test methods verify that peek returns the expected value and that it does not modify the queue.

 @Test

 **public** **void** testPeek() {

 OurQueue<String> q = **new** OurLinkedQueue<String>();

 q.add(**new** String("first"));

 assertEquals("first", q.peek());

 assertEquals("first", q.peek());

 OurQueue<Double> numbers = **new** OurLinkedQueue<Double>();

 numbers.add(1.2);

 assertEquals(1.2, numbers.peek(), 1e-14);

 assertEquals(1.2, numbers.peek(), 1e-14);

 }

 @Test

 **public** **void** testIsEmptyAfterPeek() {

 OurQueue<String> q = **new** OurLinkedQueue<String>();

 q.add("first");

 assertFalse(q.isEmpty());

 assertEquals("first", q.peek());

 }

An attempt to peek at the element at the front of an empty queue results in a java.util.NoSuchElementException, as verified by this test:

 @Test(expected = NoSuchElementException.**class**)

 **public** **void** testPeekOnEmptyList() {

 OurQueue<String> q = **new** OurLinkedQueue<String>();

 q.peek();

 }

## remove

The remove method will throw an exception if the queue is empty. Otherwise, remove returns a reference to the object at the front of the queue (the same element as peek() would). The remove method also removes the front element from the collection.

 @Test

 **public** **void** testRemove() {

 OurQueue<String> q = **new** OurLinkedQueue<String>();

 q.add("c");

 q.add("a");

 q.add("b");

 assertEquals("c", q.remove());

 assertEquals("a", q.remove());

 assertEquals("b", q.remove());

 }

 @Test(expected = NoSuchElementException.**class**)

 **public** **void** testRemoveThrowsAnException() {

 OurQueue<Integer> q = **new** OurLinkedQueue<Integer>();

 q.remove();

 }

Before the front node element is removed, a reference to the front element must be stored so it can be returned after removing it.

E frontElement = front.data;

front's next field can be sent around the first element to eliminate it from the linked structure.

front = front.next;



Now the method can return a reference to firstElement. The linked structure would now look like this.
 

Another remove makes the list look like this.



Another remove message will return "third". The remove method should set front to null so isEmpty() will still work. This will leave the linked structure like this with back referring to a node that is no longer considered to be part of the queue. In this case, back will also be set to null.



Self-Check

18-10 Complete method remove so it return a reference to the element at the front of this queue while removing the front element. If the queue is empty, throw new NoSuchElementException().

 **public** E remove() {

Answers to Self-Checks

 18-1 z

 y

 x

 18-2 true

 <<<

 18-3 false

 1

 2

 3

 true

 18-4 Check symbols in Test2.java

 Abc.java:2 expecting ]

 Abc.java:4 expecting }

 Abc.java:4 expecting }

 missing }

 4 errors

 18-5 Java

 18-6 first

 first

 first

 ...

 first until someone terminates the program or the power goes out

 18-7 **int** size = intQueue.size();

 **for** (**int** j = 1; j <= size; j++) {

 **int** nextInt = intQueue.peek();

 **if** (nextInt % 2 != 0)

 System.out.println(nextInt + " is odd");

 **else**

 System.out.println(nextInt + " is even");

 intQueue.remove();

 intQueue.add(nextInt);

 }

18-8



18-9 **public** String toString() {

 String result = "[";

 // Concatenate all but the last one (if size > 0)

 Node ref = front;

 **while** (ref != back) {

 result += ref.data + ", ";

 ref = ref.next;

 }

 // Last element does not have ", " after it

 **if** (ref != **null**)

 result += ref.data;

 result += "]";

 **return** result;

 }

18-10 **public** E remove() **throws** NoSuchElementException {

 **if** (**this**.isEmpty())

 **throw** **new** NoSuchElementException();

 E frontElement = front.data;

 front = front.next;

 **if** (front == **null**)

 front = back = **null**;

 **return** frontElement;

 }

1. Note: A *priority queue* has different behavior where elements with a higher priority would be removed first. For example, the emergency room patient with the most need is attended to next, not the patient who has been there the longest. [↑](#footnote-ref-1)