5 Stacks, Queues, and Deques

Stacks, Queues, and Deques. These ubiquitous data types are simple but powerfully used everywhere in a computer system. They represent common and intuitive data-access patterns.

In this chapter, we will explore three fundamental and widely-used data types: stacks, queues, and double-ended queues (deques). They represent common data-access patterns such as last-in first-out (LIFO), first-in first-out (FIFO), and a mixture of the two. These data types show up in numerous places, ranging from the lowest-level of software such as supporting function calls to application software such as supporting multistep undos. Our focus will be on how to implement and use these data types.

5.1 First-In First-Out, Last-In First-Out

Stacks, queues, and deques are fundamental data types that maintain a collection of elements supporting adding and deleting an element. The distinguishing trait between these data types is which element to remove when a delete operation is called:

A stack removes the most recently-added item, analogous to the back button of our web browser or a pile of plates, which is meant to only be removed from the top, making the stack a last-in first-out (LIFO) data type. The last item to arrive is to first to depart.

A queue removes the least recently-added item, analogous to a line formed to order food or check-out lines at supermarkets, making the queue a first-in first-out (FIFO) data type. The earliest item to arrive is the first to depart.

A double-ended queue (deque) offers two delete operations, one like the stack's and one like the queue's. Hence, in a deque data type, it is possible to remove the most recently-added item, as well as to remove the least recently-added item.

The add and delete operations are commonly known, respectively, as `push` and `pop` for stacks, and enqueue and dequeue for queues. For double-ended queues, their operations do not appear to have well-accepted names; this chapter uses the terms `addFirst`, `addLast`, `removeFirst`, and `removeLast`.

Figure 5.1 schematically shows the names and actions of operations on stacks and queues. A stack is logically arranged as a pile of elements: the bottom is the oldest element and the top, the most recently-added element. Therefore, push and pop operations update the stack's top. A queue is logically arranged like a check-out line: the back keeps the oldest element and

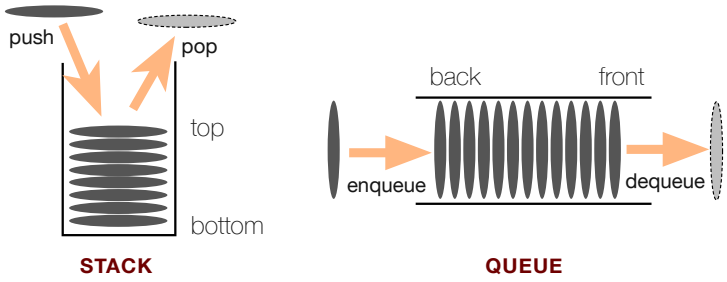


Figure 5.1: Stack and queue operations

the front holds the most recent element. Therefore, enqueue adds to the back and dequeue deletes from the front. A double-ended queue is like a queue but allows adding to and removing from both the front and the back.

5.2 Stacks

Remember from Chapter 1 that an abstract data type (ADT) defines a set of operations and their behaviors without prescribing how they must be implemented.

Among the trio in this chapter, the stack is arguably the simplest but has managed to show up in countless places. We will begin by defining the stack as an abstract data type (ADT). A *stack* keeps a collection of data elements while supporting the following operations:

- `push(e)` — adds element `e` to the top of the stack.
- `pop()` — removes and returns the element at the top of the stack.

Besides these defining operations, the following are convenience functions often provided by the stack data type:

- `top()` — returns without removing the top of the stack.
- `size()` — returns the number of elements in the stack.
- `isEmpty()` — returns whether the stack is empty.

Example. Consider a sequence of stack operations below, starting from an empty stack. The table shows the state of the stack, as well as the effects of the operations. We render the stack horizontally with the right end as the top of the stack.

Operation	Return Value	Stack's State
`push(3)`	–	{3}
`push(1)`	–	{3, 1}
`pop()`	1	{3}
`push(4)`	–	{3, 4}
`push(5)`	–	{3, 4, 5}
`top()`	5	{3, 4, 5}
`size()`	3	{3, 4, 5}
`pop()`	5	{3, 4}
`isEmpty()`	false	{3, 4}
`pop()`	4	{3}

Code 5.1: A minimal stack interface in Java.

```java
public interface Stack<T> {
    // add elt to the top of the stack
    void push(T elt);

    // remove and return the element at the top of
    // the stack (i.e., most-recently added).
    T pop();

    // return without removing the element at the
    // top of the stack.
    T top();

    // return the number of elements in the stack
    int size();

    // return a boolean indicating whether the
    // stack is empty
    boolean isEmpty();
}
```

This ADT translates to an interface in Java as shown in Code 5.1. The interface uses a generic type `T` as a placeholder for the type of the stack's elements. Despite the simplicity, this data type has found many real-world applications, including

- code parsing in compilers
- interpreter for the Postscript language for professional-grade printers
- how function calls are most often implemented.

We will see more detailed example applications later in the chapter.

Stack Implementations

First, let us mention that Java has a built-in `java.util.Stack` class. Here, we are learning to write one ourselves. For this, we will need a data storage container, preferably one that is compatible with our access patterns and that can be resized rather inexpensively. Two linear-storage data structures seem to fit the bill:

- The `ArrayList`, which supports appending and deleting the element in the rear in constant time. Furthermore, this data structure automatically resizes.
- The `LinkedList`, which supports appending and deleting the element in the rear in constant time. Furthermore, this data structure grows and shrinks its internal bookkeeping as elements are added and deleted.

Code 5.2: A stack implementation using the `ArrayList`.

```
1  import java.util.*;
2
3  public class ArrayListStack<T> implements Stack<T> {
4      private List<T> pile;
5
6      public ArrayListStack() {
7          pile = new ArrayList<T>();
8      }
9
10     // add elt to the top of the stack
11     public void push(T elt) { pile.add(elt); }
12
13     // remove and return the item at the top of
14     // the stack (most-recently added).
15     public T pop() {
16         T topElt = pile.remove(pile.size()-1);
17         return topElt;
18     }
19
20     // return without removing the item at the top
21     // of the stack.
22     public T top() { return pile.get(pile.size()-1); }
23
24     // return the number of elements in the stack
25     public int size() { return pile.size(); }
26
27     // return a boolean indicating whether the
28     // stack is empty
29     public boolean isEmpty() { return pile.isEmpty(); }
30 }
```

A Stack Implementation Using A Resizable Array

Worst Case vs. Amortized. If an operation runs in worst-case constant time, every call to that operation always takes constant time. If an operation runs in amortized constant time, some call to that operation can take longer (e.g., linear time) but in the long run, the total time taken by all these operations is no more than as if every call is constant time.

We will represent the collection of elements as an `ArrayList`, with the top of the stack being the rear end of the array. Therefore, the `push` operation amounts to appending to the array, and the `pop` operation amounts to deleting the element at the tail. Both are efficiently supported in $O(1)$ time. Code 5.2 shows an implementation of the stack data type using the `ArrayList`. Notice that the private member variable `pile` is the `ArrayList` that keeps the collection of items.

Since each of these operations takes constant time on the underlying data structure, each of the stack operations also runs in constant time. As was discussed in a previous chapter, the `ArrayList` operations can occasionally take more than constant time, but they amortize to constant time.

Code 5.3: A stack implementation using the `LinkedList`.

```
1  import java.util.*;
2
3  public class LinkedListStack<T> implements Stack<T> {
4      private LinkedList<T> pile;
5
6      public LinkedListStack() {
7          pile = new LinkedList<T>();
8      }
9
10     // add elt to the top of the stack
11     public void push(T elt) { pile.addFirst(elt); }
12
13     // remove and return the item at the top of
14     // the stack (most-recently added).
15     public T pop() {
16         T topElt = pile.pollFirst();
17         return topElt;
18     }
19
20     // return without removing the item at the top
21     // of the stack.
22     public T top() { return pile.peekFirst(); }
23
24     // return the number of elements in the stack
25     public int size() { return pile.size(); }
26
27     // return a boolean indicating whether the
28     // stack is empty
29     public boolean isEmpty() { return pile.isEmpty(); }
30 }
```

A Stack Implementation Using A Linked List

We will represent the collection of elements as a `LinkedList`, with the top of the stack being the front of the list. Notice that we could also associate the top of the stack with the back of the list. However, there is an advantage to using the front of the list: while Java's built-in `LinkedList` is doubly-linked and access to either end is a constant-time operation, the front of the list is easier to manage with less feature-full lists (e.g., a cons-list or a singly-linked list). With this design, the `push` operation amounts to prepending to the list, and the `pop` operation amounts to deleting the element at the front. Both are efficiently supported in $O(1)$ time. Code 5.3 shows an implementation of the stack data type using the `LinkedList`. Notice that the private member variable `pile` is the `LinkedList` that keeps the collection of items.

Since each of these operations takes constant time on the underlying data structure, each of the stack operations also runs in constant time. Unlike the

implementation using the `ArrayList`, this implementation features constant worst-case time for each operation, as opposed to amortized constant time.

Example Applications of Stacks

Example I: Line Editor

In the old days, text editors were not as convenient as the text editors we have today. The so-called line editors were pretty common. The basic idea is to type a sequence of symbols representing both the actual text and commands, with some special characters designated as commands:

- The pound sign `#`, for example, deletes the preceding character. Thus, the sequence of characters `"moo#d##eep"` results in the string `"meep"`.
- The at sign `@` "kills" the line—that is, resetting the current line back to an empty line. This means, if we type `"hellomorbidworld@pretty planet"`, we'll simply get `"pretty planet"`.

With a stack, it is easy to implement a function that evaluates what the user types so we have the resulting string. Deleting the most recent character corresponds to popping the stack, and killing the line is basically starting a new stack. There is a slight problem because at the end, the elements in the stack are presented in the wrong order—the reverse of the original string. This, however, is easy to rectify: just reverse the string before we return. In code, we have the following:

Code 5.4: Deriving a line in a line editor.

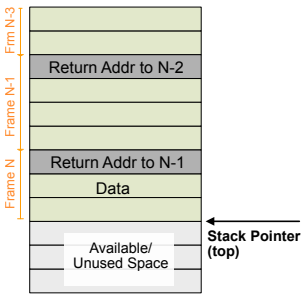
```
1  String computeLine(String inKeyStrokes) {
2      Stack<Character> line = new ArrayListStack<>();
3
4      for (int pos=0;pos<inKeyStrokes.length();pos++) {
5          char ch = inKeyStrokes.charAt(pos);
6
7          // delete the most-recent stroke
8          if (ch == '#')      line.pop();
9          // reset the line
10         else if (ch == '@') line = new ArrayListStack<>();
11         // here's the newest stroke
12         else                line.push(ch);
13     }
14
15     // At this point, line has the contents of the
16     // line - except in reversed order. To fix this,
17     // we'll build a string and reverse it.
18     StringBuilder netSB = new StringBuilder();
19     while (!line.isEmpty()) { netSB.append(line.pop()); }
20     return netSB.reverse().toString();
21 }
```

Notice that even though we are using the `StringBuilder` to help reverse the characters in this example, we could have used a stack to reverse a sequence. In general, a stack is often used to help reverse a sequence: if we push elements into a stack in sequence order, popping them one by one yields the elements in the reversed order, convenient for reversing a sequence.

Example II: Function Calls

We have studied recursive functions more or less as a black box. Internally, one magical thing that happens is the system/interpreter knows where we are in the execution flow. For example, in the factorial function, when we call `fac(5)`, it recursively calls `fac(4)`. But how does the system know to pass the return value of `fac(4)` to the body of `fac(5)`, which is waiting for it?

The answer is it keeps a stack. When a function is called, local variables (i.e. the environment) and where to return to are pushed onto the stack. And when a function returns, we pop the return address (where to go back to) and the environment of the place to go back to so that when it is time to return, it can restore the local variables and resume the running of the caller.



System Stack. Each call is stored in a frame with its (local) variable data and where to return to (return address) when this call finishes.

Example III: Expression Evaluator

How can we evaluate simple mathematical expressions such as $4+3/5.0-2*1.5$? With help from the stack, it is in fact not too difficult. To keep things simple, we will deal with fully-parenthesized expressions. For example, $4+3/5.0-2*1.5$ would be (clumsily) written as

```
((4 + (3 / 5.0)) - (2 * 1.5))
```

That is, every operator has two operands, and both that operator and its operands are surrounded by a pair of parentheses.

With this assumption, we can implement the two-stack algorithm of Dijkstra, which keeps two stacks—a value stack and an operator stack—and works as follows:

- Upon seeing a *value*, push that onto the value stack.
- Upon seeing an *operator*, push that onto the operator stack.
- Upon seeing a left parenthesis, ignore it.
- Upon seeing a right parenthesis, pop the operator and two values, carry out the computation, and push the result back onto the value stack.

Example. Consider the steps of running Dijkstra's two-stack algorithm on the input `((4 + (3 / 5.0)) - (2 * 1.5))`. Each line in the table below shows a token and the state of the two stacks after processing it. The stacks are rendered sideways with the top to the right.

Next Token	Value Stack	Operator Stack
(	{}	{}
(	{}	{}
4	{4}	{}
+	{4}	{+}
(	{4}	{+}
3	{4, 3}	{+}
/	{4, 3}	{+, /}
5.0	{4, 3, 5.0}	{+, /}
)	{4, 0.6}	{+}
)	{4.6}	{}
-	{4.6}	{-}
(	{4.6}	{-}
2	{4.6, 2}	{-}
*	{4.6, 2}	{-, *}
1.5	{4.6, 2, 1.5}	{-, *}
)	{4.6, 3}	{-}
)	{1.6}	{}

To turn this into code, we first write a function that evaluates an operator:

```
double opr(double u, String op, double v) {
    if      (op.equals("+")) return u + v;
    else if (op.equals("-")) return u - v;
    else if (op.equals("*")) return u * v;
    else if (op.equals("/")) return u / v;
    else                     return Double.NaN;
}
```

With this, the main logic can be implemented as shown below:

Code 5.5: Dijkstra's Two-Stack Expression Evaluation.

```
1  double eval(List<String> tokens) {
2      Stack<String> ops  = new ArrayListStack<>();
3      Stack<Double> vals = new ArrayListStack<>();
4      List<String> oprs = List.of("+", "-", "*", "/");
5
6      for (String tok : tokens) {
7          if      (tok.equals("(")) continue;
8          else if (oprs.contains(tok)) ops.push(tok);
9          else if (tok.equals(")")) {
10             String op = ops.pop();
11             double v  = vals.pop();
12             double u  = vals.pop();
13             vals.push(opr(u, op, v));
14         } // otherwise, this is a number
15         else vals.push(Double.parseDouble(tok));
16     }
17     return vals.top();
18 }
```

5.3 Queues

As a close cousin of the stack data type, a queue is another basic data type that shows up naturally in numerous places. While the stack follows a last-in first-out (LIFO) pattern, the queue does the opposite: the first element to arrive is also the first to depart—i.e., first-in first-out (FIFO). We will first define it as an abstract data type (ADT) and proceed to discuss how they are implemented. A *queue* keeps a collection of elements while supporting the following operations:

- `enqueue(e)` — adds element `e` to the back of the queue.
- `dequeue()` — removes and returns the element at the front of the queue.

Additionally, the following are convenience functions often provided by the queue data type:

- `front()` — returns without removing the element at the front of the queue.
- `size()` — returns the number of elements in the queue.
- `isEmpty()` — returns whether the queue is empty.

Example. Consider a sequence of queue operations below, starting from an empty queue. The table shows the state of the queue, as well as the effects of the operations. We render the queue horizontally with the right end as the front of the queue.

Operation	Return Value	Queue's State
`enqueue(3)`	–	`(3)`
`enqueue(1)`	–	`(1, 3)`
`dequeue()`	`3`	`(1)`
`enqueue(4)`	–	`(4, 1)`
`enqueue(5)`	–	`(5, 4, 1)`
`front()`	`1`	`(5, 4, 1)`
`size()`	`3`	`(5, 4, 1)`
`dequeue()`	`1`	`(5, 4)`
`isEmpty()`	`false`	`(5, 4)`
`dequeue()`	`4`	`(5)`

This ADT directly translates to the Java interface in Code 5.6. The interface uses a generic type `T` as a placeholder for the type of the queue's elements. The queue is such an intuitive data type. Their applications are often related to keeping objects/people and making sure they are "served" in the same order that they come in. Examples include

- Keeping a queue of requests for a shared resource (e.g., a printer).
- Lining up customers in a call center.
- Performing event simulations where the queue keeps future events to be worked on.
- Crawling a website to visit all pages reachable from a source page.

Code 5.6: A minimal queue interface in Java.

```java
public interface Queue<T> {
    // add elt to the back of the queue.
    void enqueue(T elt);

    // remove and return the element at the front of
    // the queue.
    T dequeue();

    // return without removing the element at the front
    // of the queue.
    T front();

    // return the number of elements in the queue.
    int size();

    // return a boolean indicating whether the queue
    // is empty.
    boolean isEmpty();
}
```

Queue Implementations

First, let us mention that Java has a built-in implementation of the queue data type. The built-in interface is known has `java.util.Queue` with method names that are unique to Java. Concrete implementations include `LinkedList` and `ArrayDeque`. Here, we are learning to write one ourselves. For this, we will need a data storage container, preferably one that is compatible with our access patterns and that can be resized rather inexpensively. Two linear-storage data structures meet this goal:

- The `LinkedList` supports adding an element to the rear and deleting the element at the front in constant time. Furthermore, this data structure readily grows and shrinks its internal bookkeeping as elements are added and deleted.
- Java's fixed-size array, although it does not grow and shrink automatically, can be used to support adding an element to the rear and deleting the element at the front in constant time with the help of two pointers.

A Queue Implementation Using A Linked List

We will represent the collection of elements as a `LinkedList`, with the front of the list being the queue's front. With this arrangement, the `enqueue` operation amounts to sticking an element to the rear and the `dequeue` operation is deleting the element at the front. Both require only $O(1)$ time on a doubly-linked list. Code 5.7 shows an implementation using this approach, where the private variable `q` is the linked list where the collection of elements is kept.

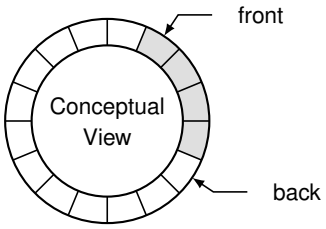
Code 5.7: A queue implementation using the `LinkedList`.

```
1  import java.util.LinkedList;
2
3  public LinkedListQueue<T> implements Queue<T> {
4      private LinkedList<T> q;
5
6      public LinkedListQueue() {
7          q = new LinkedList<>();
8      }
9
10     // add elt to the back of the queue.
11     public void enqueue(T elt) { q.addLast(elt); }
12
13     // remove and return the element at the front of
14     // the queue.
15     public T dequeue() { return q.removeFirst(); }
16
17     // return without removing the element at the front
18     // of the queue.
19     public T front() { return q.getFirst(); }
20
21     // return the number of elements in the queue.
22     public int size() { return q.size(); }
23
24     // return a boolean indicating whether the queue
25     // is empty.
26     public boolean isEmpty() { return q.isEmpty(); }
27 }
```

A Queue Implementation Using A Fixed-Size Array

At first glance, we might be tempted to use an `ArrayList`, like we did to implement a stack, because then resizing would be automatic. However, we face a big challenge in using an array-based data structure: Because the elements of our queue are kept contiguously, there is a fast end and a slow end. As an example, adding an element to or removing an element from the rear of an `ArrayList` takes constant time, but the same operations performed at the front will require linear time. This is because, to keep the elements contiguous, the rest of the elements have to be moved to the right location, incurring linear time.

Circular View. View the array as circular and use two indices to mark the front and the back of the queue.



All these obstacles can be easily avoided if we have a maximum capacity in mind. Indeed, the resizing lesson we applied to create the `ArrayList` can be used here so the capacity is about the same as the number of elements while keeping the amortized cost of each operation constant. For this reason, we will focus this lesson on building a queue with a preset capacity N.

We keep an array of size N. Conceptually, though, we view it as circular, with index 0 adjacent to index $N-1$. Additionally, we maintain two indices

into the array: `front` is the index that holds the element at the front of the queue and `back` is the index where the next incoming element will be stored. While it is possible to derive the current size from these two indices, we choose to simplify logic and keep an additional variable `n` for size. Notice that in a circular array, if the current index is `i`, the index `(i+1)%N` is next in the circular view, with the modulo operator accounting for wrapping around.

We implement this logic as shown in Code 5.8, where the elements are kept in the array `q`. Initially, both `front` and `back` refer to the same index. Readers are encouraged to reason why this is the right thing to do. The `enqueue` operation stores the new element at index `back` and advances it by one step. The `dequeue` operation fetches the element at index `front` and advances it by one step. To help Java's garbage collector, when an element is removed from the queue, we set that location to `null` to avoid dangling, rogue reference from the queue to that element.

5.4 Double-Ended Queues (Deques)

In many situations, we desire an all-in-one data type supporting both FIFO (like a queue) and LIFO (like a stack) access patterns. Such a data type is in demand because certain applications need both patterns on the same container and also because it can be had in the same constant-time performance as the traditional stack and queue. A double-ended queue (deque, pronounced DECK) looks much like a queue. Elements can be added to either end and can also be removed from either end. For concreteness, we will define it as an abstract data type. A *deque* keeps a collection of elements while supporting the following operations:

- `addFirst(e)` — adds element `e` at the front of the deque.
- `addLast(e)` — adds element `e` at the back of the deque.
- `removeFirst()` — removes and returns the element at the front of the deque.
- `removeLast()` — removes and returns the element at the back of the deque.

Additionally, the following are convenience functions often provided by the queue data type:

- `first()` — returns without removing the element at the front of the deque.
- `last()` — returns without removing the element at the back of the deque.
- `size()` — returns the number of elements in the deque.
- `isEmpty()` — returns whether the deque is empty.

Unlike the stack and queue data types, the names of deque operations are not as standard. We mostly follow Java's naming. Other common languages such as Python and C++ use different names but the idea is the same.

> **Example.** Consider a sequence of deque operations below, starting from an empty collection. The table shows the state of the deque, as well as the effects of the operations. We render the deque horizontally with the

right end as the front of the queue.

Operation	Return Value	Deque State
`addLast(3)`	–	`(3)`
`addFirst(1)`	–	`(3, 1)`
`addLast(4)`	–	`(4, 3, 1)`
`first()`	`1`	`(4, 3, 1)`
`removeLast()`	`4`	`(3, 1)`
`removeFirst()`	`1`	`(3)`
`size()`	`1`	`(3)`
`isEmpty()`	`false`	`(3)`

It is straightforward to translate this ADT to a Java interface; we left this as an exercise to the reader. Below we discuss two implementation options.

Implementation Options

Our discussion of the stack and queue data types show various implementation techniques. Here we will discuss how some of these techniques can be adopted to implement the deque datatype.

- The `LinkedList` nicely supports all the desired operations. In constant time, we can add an element to the front or rear and delete the element at the front or rear. Furthermore, this data structure readily grows and shrinks its internal bookkeeping as elements are added and deleted. In fact, the `LinkedList` class in Java lines up with the built-in `Deque` interface so much so that `LinkedList` implements the `Deque` interface.
- A fixed-size array, although it does not grow and shrink automatically, can be used to implement the deque data type. If we keep two pointers like we did for the queue, we can, in constant time, add an element to the front or rear, and delete the element at the front or rear. This option is especially useful when a maximum capacity is known up front or we are willing to resize the underlying array appropriately (e.g., using the doubling trick). Indeed, Java's built-in `java.util.ArrayDeque` is an implementation based on this idea with appropriate resizing.

Exercises

Exercise 5.1. What values are returned during the following series of stack operations, if we start from an empty stack?

```
push(7), push(9), pop(), push(2), push(5), pop(),
pop(), push(1), push(8), pop(), push(7), push(4),
pop(), pop(), push(6), pop()
```

Exercise 5.2. What values are returned during the following series of queue operations, if we start from an empty queue?

```
enqueue(7),  enqueue(9),  dequeue(),  enqueue(2),
enqueue(5),  dequeue(),   dequeue(),  enqueue(1),
enqueue(8),  dequeue(),   enqueue(7), enqueue(4),
dequeue(), dequeue(), enqueue(6), dequeue()
```

Exercise 5.3. What values are returned during the following sequence of deque operations, if we start from an empty deque?

```
addFirst(7), addLast(9), addLast(2), addFirst(5),
last(),  isEmpty(),  addFirst(1),  removeLast(),
addLast(8), first(), last(), addLast(7), size(),
removeFirst(), removeFirst()
```

Exercise 5.4. On a queue data type, define a *cycle* operation as taking the element at the front of the queue and move it to the rear, effectively causing the queue to rotate by one position. This can always be implemented using a `dequeue` followed by an `enqueue`. In this exercise, you will work with the fixed-size array implementation. You will implement a method `cycle(k)`, which performs the cycle operation exactly k times. Your method will work directly on the array without calling the existing `enqueue` and `dequeue` methods.

Exercise 5.5. Our current queue implementation using a fixed-size array only works with a preset (i.e., fixed) capacity. Use the resizing technique from a previous chapter to turn this implementation into a queue with $O(1)$-time support for all operations without being constrained by a preset capacity. If the queue stores n elements, it must use space $O(n)$.

Exercise 5.6. Similar to the last exercise, upgrade our deque implementation using a fixed-size array to resize automatically so it is not constrained by a preset capacity. If the deque stores n elements, it must use space $O(n)$.

Exercise 5.7. Extend the Dijkstra's two-stacks implementation discussed earlier in the chapter to support exponentiation and basic functions such as `log`, `sin`, `cos`. Your implementation will therefore support expressions such as `(2**(log(5/(1+10))*11))-1.9`.

Exercise 5.8. The *postfix notation* is an unambiguous way to write an arithmetic expression without any parentheses. As the name suggests, the operation is written last—after the operands. This means, if (e_1) **op** (e_2) is a normal fully-parenthesized expression, its equivalent postfix version is p_1 p_2 **op**, where p_i, $i = 1, 2$, is the postfix equivalent of e_i. As an example, the expression `((3 + 4)*(5 - 2)) / 4` has a postfix equivalent of `3 4 + 5 2 - 4 /`.

Describe a nonrecursive solution to evaluate a postfix expression.

Exercise 5.9. Write a program to turn a fully-parenthesized expression into a postfix equivalent form.

Exercise 5.10. Earlier we saw a two-stack solution for evaluating a *fully-parenthesized* expression. In this exercise, you will extend it to additionally support exponentiation $**$. You will extend it to also support expressions that are not fully parenthesized. To accomplish this, you will work in two steps, broken down into a few subtasks below. *Throughout, assume all numbers are floating-point numbers*, so the expression 5/2 evaluates to 2.5. Moreover, it is guaranteed that all the expressions your functions will be tested on evaluate to a number; you will not be asked for the result of 4/0, for example.

To make life a little easier, you will be working with input that is a list of tokens. For example, instead of taking in `"(1+41)*2"`, your functions will work with the list `{"(", "1", "+", "41", ")", "*", "2"}`. Such a list has type `List<String>` in Java.

Subtask I: Implement a function

```
double evalFullyParenthesized(List<String> tokens)
```

that takes an expression represented as a list of fully-parenthesized tokens and returns the result of evaluating that expression. This function must support `+`, `-`, `*`, `/`, `**` and an arbitrary nesting of parentheses. Remember that the algorithm we looked at previously does not yet support exponentiation (`**`).

Subtask II: Most expressions we write and encounter in real life, however, are *not* fully parenthesized. Early in life, we learned and have internalized certain precedence rules that make writing, for example, `1 + 3*5` unambiguous—it is $1 + (3 \times 5) = 16$. Similarly, we know that result of `1 + 3 * 5 - 21*2/7` is 10.0. The rules of precedence, or what many of us remember as the acronym PEMDAS, state that

- **P**arentheses have the highest precedence. This means, $2 * (5 - 1) = 2 \times 4 = 8$.
- **E**xponentiation has the next highest precedence $2 * 1 * *3 + 1 == ((2 \times (1^3)) + 1 = 3$.
- After that, **M**ultiplication and **D**vision have the same precedence;
- Below that, **A**ddition and **S**ubtraction have the same precedence;
- Finally, operators with the same precedence are evaluated left to right.

For this subtask, you are to implement a function

```
List<String> augmentExpr(List<String> tokens)
```

that takes a list of tokens and returns a list of tokens that is a fully-parenthesized version of the input expression. The input list may already be partially or fully parenthesized.

(*Hints:* This task can be solved by extending the Dijkstra's two-stack algorithm. You may wish to attempt this task in two stages. Again, you will scan the token list from left to right:

- *Stage I: No Partial Parentheses.* Keep two stacks like before, but what should happen when you see an operator? When should you push that to one of the stacks? And when should you not—but rather popping certain things out?
- *Stage II: Partial Parentheses.* What should be done upon seeing an open parenthesis? What about a close parenthesis? If popping, how far?)

Chapter Notes

The stack, queue, and double-ended queue data types are discussed in virtually any data structures books. Recommended further reading includes the book by Sedgewick and Wayne [SW11], the book by Goodrich et al. [GTG14], and Knuth's classic bible [Knu73]. Aside from traditional implementations of stacks and queued mentioned here, there is a neat way to implement a queue using two stacks [Oka99], popularized by the functional programming community. Exercise 5.10. is perhaps more easily solved using a tree representation; readers are encouraged to revisit it after completing §12.3.

Code 5.8: A queue implementation using a fixed-size array.

```
public class FixedArrayQueue<T> implements Queue<T> {
    private T[] q;
    int front, back, n;

    public LinkedListQueue(int N) {
        q = (T []) new Object[N];
        front = back = 0;
        n = 0;
    }

    // add elt to the back of the queue.
    public void enqueue(T elt) {
        q[back] = elt;
        back = (back+1)%N;
        n++;
    }

    // remove and return the element at the front of
    // the queue.
    public T dequeue() {
        T frontElt = q[front];
        q[front] = null;
        front = (front+1)%N;
        n--;
    }

    // return without removing the element at the front
    // of the queue.
    public T front() { return q[front]; }

    // return the number of elements in the queue.
    public int size() { return n; }

    // return a boolean indicating whether the queue
    // is empty.
    public boolean isEmpty() { return 0==n; }
}
```