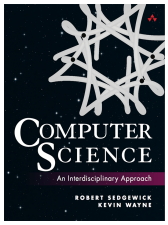
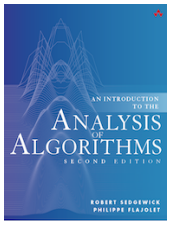




- Web Resources
- FAQ
- Data
- Code
- Errata
- Cheatsheet
- References
- Online Course
- Lecture Slides
- Programming Assignments

Custom Search

1.3 Bags, Queues, and Stacks

Several fundamental data types involve collections of objects. Specifically, the set of values is a collection of objects, and the operations revolve around adding, removing, or examining objects in the collection. In this section, we consider three such data types, known as the bag, the queue, and the stack. They differ in the specification of which object is to be removed or examined next.

APIs.

We define the APIs for bags, queues, and stacks. Beyond the basics, these APIs reflect two Java features: generics and iterable collections.

Bag

public class Bag<Item> implements Iterable<Item>	
Bag()	*create an empty bag*
void add(Item item)	*add an item*
boolean isEmpty()	*is the bag empty?*
int size()	*number of items in the bag*

FIFO queue

public class Queue<Item> implements Iterable<Item>	
Queue()	*create an empty queue*
void enqueue(Item item)	*add an item*
Item dequeue()	*remove the least recently added item*
boolean isEmpty()	*is the queue empty?*
int size()	*number of items in the queue*

Pushdown (LIFO) stack

public class Stack<Item> implements Iterable<Item>	
Stack()	*create an empty stack*
void push(Item item)	*add an item*
Item pop()	*remove the most recently added item*
boolean isEmpty()	*is the stack empty?*
int size()	*number of items in the stack*

- *Generics.* An essential characteristic of collection ADTs is that we should be able to use them for any type of data. A specific Java mechanism known as *generics* enables this capability. The notation `<Item>` after the class name in each of our APIs defines the name `Item` as a *type parameter*, a symbolic placeholder for some concrete type to be used by the client. You can read `Stack<Item>` as "stack of items." For example, you can write code such as

```
Stack<String> stack = new Stack<String>();
stack.push("Test");
...
String next = stack.pop();
```

  to use a stack for `String` objects.

- *Autoboxing.* Type parameters have to be instantiated as reference types, so Java automatically converts between a primitive type and its corresponding wrapper type in assignments, method arguments, and arithmetic/logic expressions. This conversion enables us to use generics with primitive types, as in the following code:

```
Stack<Integer> stack = new Stack<Integer>();
stack.push(17);        // autoboxing (int -> Integer)
```

```
int i = stack.pop();   // unboxing   (Integer -> int)
```

Automatically casting a primitive type to a wrapper type is known as *autoboxing*, and automatically casting a wrapper type to a primitive type is known as *unboxing*.

- *Iterable collections.* For many applications, the client's requirement is just to process each of the items in some way, or to *iterate* through the items in the collection. Java's *foreach* statement supports this paradigm. For example, suppose that `collection` is a `Queue<Transaction>`. Then, if the collection is iterable, the client can print a transaction list with a single statement:

```
for (Transaction t : collection)
   StdOut.println(t);
```

- *Bags.* A *bag* is a collection where removing items is not supported—its purpose is to provide clients with the ability to collect items and then to iterate through the collected items. Stats.java is a bag client that reads a sequence of real numbers from standard input and prints out their mean and standard deviation.

- *FIFO queues.* A *FIFO queue* is a collection that is based on the *first-in-first-out* (FIFO) policy. The policy of doing tasks in the same order that they arrive is one that we encounter frequently in everyday life: from people waiting in line at a theater, to cars waiting in line at a toll booth, to tasks waiting to be serviced by an application on your computer.

- *Pushdown stack.* A *pushdown stack* is a collection that is based on the *last-in-first-out* (LIFO) policy. When you click a hyperlink, your browser displays the new page (and pushes onto a stack). You can keep clicking on hyperlinks to visit new pages, but you can always revisit the previous page by clicking the back button (popping it from the stack). Reverse.java is a stack client that reads a sequence of integers from standard input and prints them in reverse order.

- *Arithmetic expression evaluation.* Evaluate.java is a stack client that evaluates fully parenthesized arithmetic expressions. It uses Dijkstra's 2-stack algorithm:
  - Push operands onto the operand stack.
  - Push operators onto the operator stack.
  - Ignore left parentheses.
  - On encountering a right parenthesis, pop an operator, pop the requisite number of operands, and push onto the operand stack the result of applying that operator to those operands.

  This code is a simple example of an *interpreter*.

Array and resizing array implementations of collections.

- *Fixed-capacity stack of strings.* FixedCapacityStackOfString.java implements a fixed-capacity stack of strings using an array.

- *Fixed-capacity generic stack.* FixedCapacityStack.java implements a generic fixed-capacity stack.

- *Array resizing stack.* ResizingArrayStack.java implements a generic stack using a *resizing array*. With a resizing array, we dynamically adjust the size of the array so that it is both sufficiently large to hold all of the items and not so large as to waste an excessive amount of space. We *double* the size of the array in `push()` if it is full; we *halve* the size of the array in `pop()` if it is less than one-quarter full.

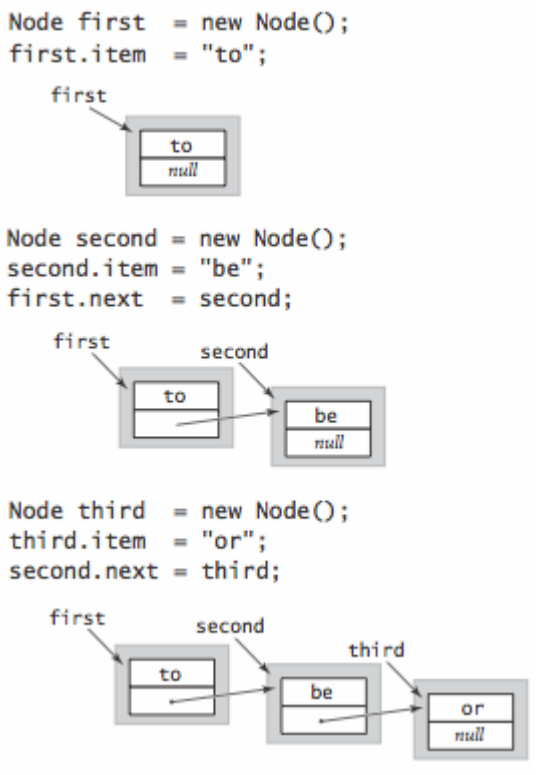
- *Array resizing queue.* ResizingArrayQueue.java implements the queue API with a resizing array.

Linked lists.

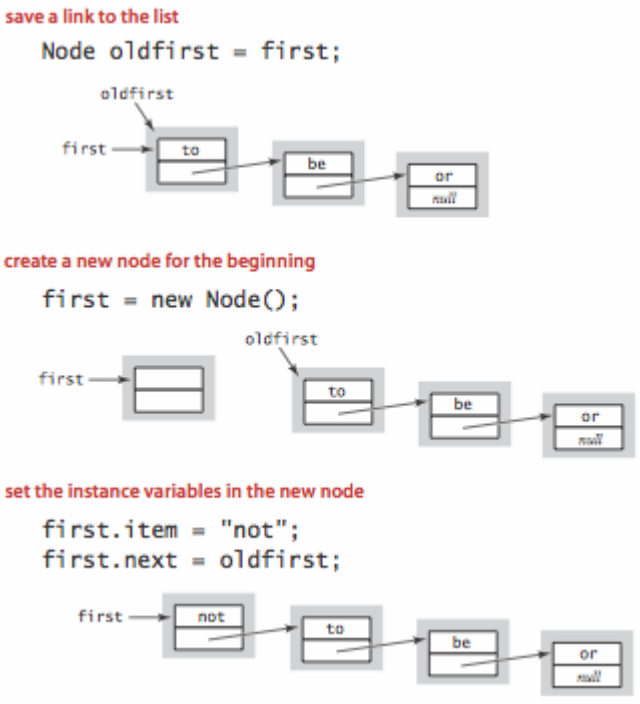
A *linked list* is a recursive data structure that is either empty (*null*) or a reference to a *node* having a generic item and a reference to a linked list. To implement a linked list, we start with a *nested class* that defines the node abstraction

```
private class Node {
   Item item;
   Node next;
}
```

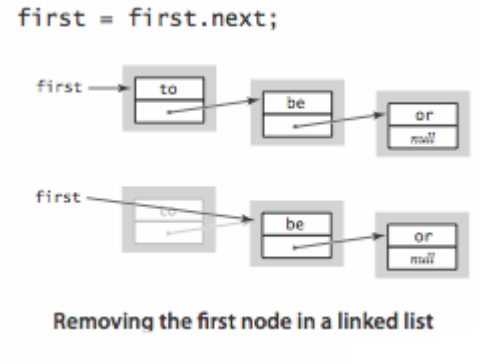
- *Building a linked list.* To build a linked list that contains the items `to`, `be`, and `or`, we create a `Node` for each item, set the item field in each of the nodes to the desired value, and set the `next` fields to build the linked list.



- *Insert at the beginning.* The easiest place to insert a new node in a linked list is at the beginning.

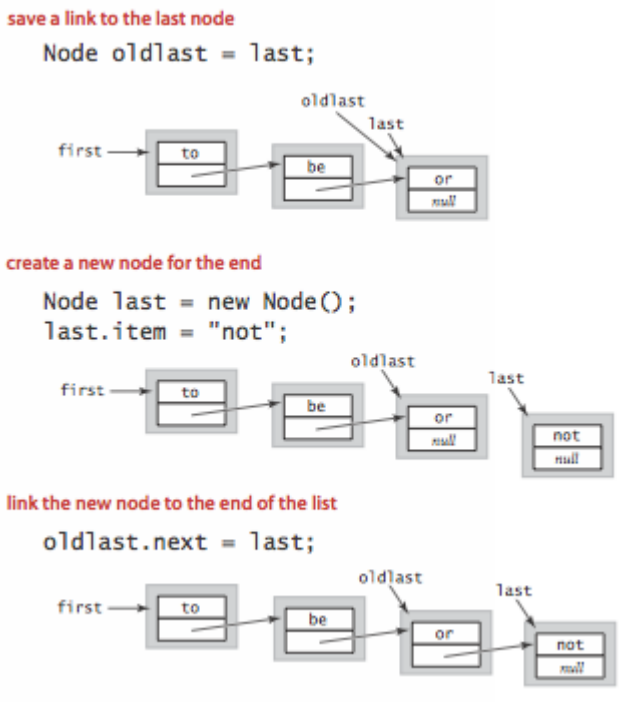


- *Remove from the beginning.* Removing the first node in a linked list is also easy.



Removing the first node in a linked list

- *Insert at the end.* To insert a node at the end of a linked list, we maintain a link to the last node in the list.



- *Traversal.* The following is the idiom for traversing the nodes in a linked list.

```
for (Node x = first; x != null; x = x.next) {
   // process x.item
}
```

Linked-list implementations of collections.

- *Linked list implementation of a stack.* Stack.java implements a generic stack using a linked list. It maintains the stack as a linked list, with the top of the stack at the beginning, referenced by an instance variable `first`. To `push()` an item, we add it to the beginning of the list; to `pop()` an item, we remove it from the beginning of the list.

- *Linked list implementation of a queue.* Program Queue.java implements a generic FIFO queue using a linked list. It maintains the queue as a linked list in order from least recently to most recently added items, with the beginning of the queue referenced by an instance variable `first` and the end of the queue referenced by an instance variable `last`. To `enqueue()` an item, we add it to the end of the list; to `dequeue()` an item, we remove it from the beginning of the list.

- *Linked list implementation of a bag.* Program Bag.java implements a generic bag using a linked list. The implementation is the same as Stack.java except for changing the name of `push()` to `add()` and removing `pop()`.

Iteration.

To consider the task of implementing iteration, we start with a snippet of client code that prints all of the items in a collection of strings, one per line:

```
Stack<String> collection = new Stack<String>();
...
for (String s : collection)
```

```
   StdOut.println(s);
...
```

This *foreach* statement is shorthand for the following `while` statement:

```
Iterator<String> i = collection.iterator();
while (i.hasNext()) {
   String s = i.next();
   StdOut.println(s);
}
```

To implement iteration in a collection:

- Include the following `import` statement so that our code can refer to Java's java.util.Iterator interface:

```
import java.util.Iterator;
```

- Add the following to the class declaration, a promise to provide an `iterator()` method, as specified in the java.lang.Iterable interface:

```
implements Iterable<Item>
```

- Implement a method `iterator()` that returns an object from a class that implements the `Iterator` interface:

```
public Iterator<Item> iterator() {
    return new ListIterator();
}
```

- Implement a nested class that implements the `Iterator` interface by including the methods `hasNext()`, `next()`, and `remove()`. We always use an empty method for the optional `remove()` method because interleaving iteration with operations that modify the data structure is best avoided.
    - The nested class `ListIterator` in Bag.java illustrates how to implement a class that implements the `Iterator` interface when the underlying data structure is a linked list.
    - The nested class `ArrayIterator` in ResizingArrayBag.java does the same when the underlying data structure is an array.

Autoboxing Q + A

Q. How does autoboxing handle the following code fragment?

```
Integer a = null;
int b = a;
```

A. It results in a run-time error. Primitive type can store every value of their corresponding wrapper type except `null`.

Q. Why does the first group of statements print `true`, but the second `false`?

```
Integer a1 = 100;
Integer a2 = 100;
System.out.println(a1 == a2);   // true

Integer b1 = new Integer(100);
Integer b2 = new Integer(100);
System.out.println(b1 == b2);   // false

Integer c1 = 150;
Integer c2 = 150;
System.out.println(c1 == c2);   // false
```

A. The second prints `false` because `b1` and `b2` are references to different Integer objects. The first and third code fragments rely on autoboxing. Surprisingly the first prints true because values between -128 and 127 appear to refer to the same immutable Integer objects (Java's implementation of `valueOf()` retrieves a cached values if the integer is between -128 and 127), while Java constructs new objects for each integer outside this range.

Here is another Autoboxing.java anomaly.

Generics Q + A

Q. Are generics solely for auto-casting?

A. No, but we will use them only for "concrete parameterized types", where each data type is parameterized by a single type. The primary benefit is to discover type-mismatch errors at compile time instead of run time. There are other more general (and more complicated) uses of generics, including wildcards. This generality is useful for handling subtypes and inheritance. For more information, see this Generics FAQ and this Java generics tutorial.

Q. Can concrete parameterized types be used in the same way as normal types?

A. Yes, with a few exceptions (array creation, exception handling, with `instanceof`, and in a class literal).

Q. Can I make the Node class static?

A. For LinkedStackOfString.java, you can do so with no other changes and save 8 bytes (of inner class overhead) per node. However, the nested class `Node` in LinkedStack.java uses the type information of `Item` from the outer class, so you would need to do a bit of extra work to make it static. Stack.java accomplishes this by making the nested class (and the nester iterator) generic: there are three separate generic type parameters, each of which is named `Item`.

Q. Why do I get a "can't create an array of generics" error when I try to create an array of generics?

```
public class ResizingArrayStack<Item> {
   Item[] a = new Item[1];
```

A. Unfortunately, creating arrays of generics is not possible in Java 1.5. The underlying cause is that arrays in Java are *covariant*, but generics are not. In other words, `String[]` is a subtype of `Object[]`, but `Stack<String>` is

not a subtype of `Stack<Object>`. To get around this defect, you need to perform an unchecked cast as in ResizingArrayStack.java. ResizingArrayStackWithReflection.java is an (unwieldy) alternative that avoids the unchecked cast by using reflection.

Q. So, why are arrays covariant?

A. Many programmers (and programming language theorists) consider covariant arrays to be a serious defect in Java's type system: they incur unnecessary run-time performance overhead (for example, see ArrayStoreException) and can lead to subtle bugs. Covariant arrays were introduced in Java to circumvent the problem that Java didn't originally include generics in its design, e.g., to implement `Arrays.sort(Comparable[])` and have it be callable with an input array of type `String[]`.

Q. Can I create and return a new array of a parameterized type, e.g., to implement a `toArray()` method for a generic queue?

A. Not easily. You can do it using reflection provided that the client passes an object of the desired concrete type to `toArray()` This is the (awkward) approach taken by Java's Collection Framework. GenericArrayFactory.java provides an alternate solution in which the client passes a variable of type `Class`. See also Neal Gafter's blog for a solution that uses type tokens.

Iterator Q + A

Q. Why is the construct called *foreach* if it uses the keyword `for`?

A. Other languages use the keyword `foreach`, but the Java developers did not want to introduce a new keyword and break backward compatibility.

Q. Are `Strings` iterable?

A. No.

Q. Are arrays `Iterable`?

A. No. You can use the foreach syntax with them. However, you can not pass an array to a method that expects an `Iterable` or return an array from a method which returns an `Iterable`. This would be convenient, but it doesn't work that way.

Q. What's wrong with the following code fragment?

```
String s;
for (s : listOfStrings)
   System.out.println(s);
```

A. The enhanced for loop requires that the iterating variable be declared inside the loop.

Exercises

1. Add a method `isFull()` to FixedCapacityStackOfStrings.java.
2. Give the output printed by `java Stack` for the input

```
it was - the best - of times - - - it   was - the - -
```

Solution. `was best times of the was the it (1 left on stack)`

3. Suppose that an intermixed sequence of (stack) *push* and *pop* operations are performed. The pushes push the integers 0 through 9 in order; the pops print out the return value. Which of the following sequence(s) could not occur?

```
(a)  4 3 2 1 0 9 8 7 6 5

(b)  4 6 8 7 5 3 2 9 0 1

(c)  2 5 6 7 4 8 9 3 1 0

(d)  4 3 2 1 0 5 6 7 8 9

(e)  1 2 3 4 5 6 9 8 7 0

(f)  0 4 6 5 3 8 1 7 2 9

(g)  1 4 7 9 8 6 5 3 0 2

(h)  2 1 4 3 6 5 8 7 9 0
```

Answer: (b), (f), and (g).

4. Write a stack client Parentheses.java that reads in sequence of left and right parentheses, braces, and brackets from standard input and uses a stack to determine whether the sequence is properly balanced. For example, your program should print `true` for `[()]{}{[()()]()}` and `false` for `[(])`.
5. What does the following code fragment print when `n` is 50? Give a high-level description of what it does when presented with a positive integer `n`.

```
Stack<Integer> s = new Stack<Integer>();
while (n > 0) {
   s.push(n % 2);
   n = n / 2;
}
while (!s.isEmpty())
    System.out.print(s.pop());
System.out.println();
```

Answer: Prints the binary representation of `N` (`110010` when `n` is `50`).

6. What does the following code fragment do to the queue `q`?

```
Stack<String> s = new Stack<String>();
while(!q.isEmpty())
   s.push(q.dequeue());
while(!s.isEmpty())
   q.enqueue(s.pop());
```

Answer: Reverses the items on the queue.

7. Add a method `peek` to Stack.java that returns the most recently inserted item on the stack (without popping it).

10. Write a filter Program InfixToPostfix.java that converts an arithmetic expression from infix to postfix.

11. Write a program EvaluatePostfix.java that that takes a postfix expression from standard input, evaluates it, and prints the value. (Piping the output of your program from the previous exercise to this program gives equivalent behavior to Evaluate.java.)
13. Suppose that a client performs an intermixed sequence of (queue) *enqueue* and *dequeue* operations. The enqueue operations put the integers 0 through 9 in order onto the queue; the dequeue operations print out the return value. Which of the following sequence(s) could not occur?

```
(a)  0 1 2 3 4 5 6 7 8 9

(b)  4 6 8 7 5 3 2 9 0 1

(c)  2 5 6 7 4 8 9 3 1 0

(d)  4 3 2 1 0 5 6 7 8 9
```

Answer: (b), (c), and (d).

14. Develop a class `ResizingArrayQueueOfStrings` that implements the queue abstraction with a fixed-size array, and then extend your implementation to use array resizing to remove the size restriction.

Solution: ResizingArrayQueue.java

Linked-List Exercises

Creative Problems

37. **Josephus problem.** In the Josephus problem from antiquity, *N* people are in dire straits and agree to the following strategy to reduce the population. They arrange themselves in a circle (at positions numbered from 0 to N-1) and proceed around the circle, eliminating every Mth person until only one person is left. Legend has it that Josephus figured out where to sit to avoid being eliminated. Write a `Queue` client Josephus.java that takes M and N from the command line and prints out the order in which people are eliminated (and thus would show Josephus where to sit in the circle).

```
 % java Josephus 2 7
1 3 5 0 4 2 6
```

42. **Copy a stack.** Create a new constructor for the linked-list implementation of Stack.java so that `Stack t = new Stack(s)` makes `t` reference a new and independent copy of the stack `s`.

Recursive solution: create a copy constructor for a linked list starting at a given `Node` and use this to create the new stack.

```
Node(Node x) {
   item = x.item;
   if (x.next != null) next = new Node(x.next);
}

public Stack(Stack<Item> s) { first = new Node(s.first); }
```

Nonrecursive solution: create a copy constructor for a single `Node` object.

```
Node(Node x) { this.item = x.item; this.next = x.next; }
```

```
public Stack(Stack<Item> s) {
   if (s.first != null) {
      first = new Node(s.first);
      for (Node x = first; x.next != null; x = x.next)
         x.next = new Node(x.next);
   }
}
```

45. **Stack generability.** Suppose that we have a sequence of intermixed *push* and *pop* operations as with our test stack client, where the integers 0, 1, ..., N-1 in that order (*push* directives) are intermixed with *N* minus signs (*pop* directives). Devise an algorithm that determines whether the intermixed sequence causes the stack to underflow. (You may use only an amount of space independent of *N*—you cannot store the integers in a data structure.) Devise a linear-time algorithm that determines whether a given permutation can be generated as output by our test client (depending on where the *pop* operations occur).

    Solution. The stack does not underflow unless there exists an integer *k* such that the first *k* pop operations occur before the first *k* push operations.

    If a given permutation can be generated, it is uniquely generated as follows: if the next integer in the permutation is in the top of the stack, pop it; otherwise, push the next integer in the input sequence onto the stack (or stop if N-1 has already been pushed). The permutation can be generated if and only if the stack is empty upon termination.

46. **Forbidden triple for stack generability. (R. Tarjan)** Prove that a permutation can be generated by a stack (as in the previous question) if and only if it has no *forbidden triple (a, b, c)* such that a < b < c with c first, a second, and b third (possibly with other intervening integers between c and a and between a and b).

    Partial solution. Suppose that there is a forbidden triple (a, b, c). Item c is popped before a and b, but a and b are pushed before c. Thus, when c is pushed, both a and b are on the stack. Therefore, a cannot be popped before b.

47. **Catenable queues, stacks, or steques.** Add an extra operation *catenation* that (destructively) concatenates two queues, stacks, or steques. *Hint*: use a circular linked list, maintaining a pointer to the last item.

50. **Fail-fast iterator.** Modify the iterator code in Stack.java to immediately throw a java.util.ConcurrentModificationException if the client modifies the collection (via `push()` or `pop()`) during iteration.

    Solution: Maintain a counter that counts the number of `push()` and `pop()` operations. When creating an iterator, store this value as an iterator instance variable. Before each call to `hasNext()` and `next()`, check that this value has not changed since construction of the iterator; if it has, throw an exception.

51. **Expression evaluation with precedence.** Write a program EvaluateDeluxe.java that extends Evaluate.java to handle expressions that are not fully parenthesized, using the standard precedence order for the operators +, -, *, and /.

Web Exercises

1. **Tail.** Write a program `Tail` so that `Tail k < file.txt` prints the last `k` lines of the file `file.txt`. Use `StdIn.readLine()`. Which data structure should you use?
2. **Bounded stack.** A *bounded stack* is a stack that has a capacity of at most N. (Applications: undo or history with finite buffer.)
3. **Delete ith element.** Create a data type that supports the following operations: `isEmpty`, `insert`, and `remove(int i)`, where the deletion operation deletes and returns the ith least recently added object on the

queue. Do it with an array, then do it with a linked list. What is the running time for each operation?

4. **Dynamic shrinking.** With the array implementations of stack and queue, we doubled the size of the array when it wasn't big enough to store the next element. If we perform a number of doubling operations, and then delete a lot of elements, we might end up with an array that is much bigger than necessary. Implement the following strategy: whenever the array is 1/4 full or less, shrink it to half the size. Explain why we don't shrink it to half the size when it is 1/2 full or less.
5. **Stack + max.** Create a data structure that efficiently supports the stack operations (pop and push) and also return the maximum element. Assume the elements are integers or reals so that you can compare them.

   Hint: use two stacks, one to store all of the elements and a second stack to store the maximums.

6. **PostScript.** *PostScript* is a stack-based language used by most printers. Implement a small subset of PostScript using a stack.
7. **Interview question.** Given a stack of an unknown number of strings, print out the 5th to the last one. It's OK to destroy the stack in the process. *Hint*: use a queue of 5 elements.
8. **Tag systems.** Write a program that reads in a binary string from the command line and applies the following (00, 1101) tag-system: if the first bit is 0, delete the first three bits and append 00; if the first bit is 1, delete the first three bits and append 1101. Repeat as long as the string has at least 3 bits. Try to determine whether the following inputs will halt or go into an infinite loop: 10010, 100100100100100100. Use a queue.
9. **Turing tape.** Implement an *one-dimensional Turing tape*. The tape consists of a sequence of cells, each of which stores an integer (that is initialized to 0). At any instant, there is a *tape head* which points to one of the cells. Support the following interface methods: `moveLeft` to move the tape head one cell to the left, `moveRight` to move the tape head one cell to the right, `look` to return the contents of the active cell, and `write(int a)` to change the contents of the active cell to `a`. *Hint*: use an `int` for the active cell, and two stacks for the left and right parts of the tape. Similar to text editor buffer.
10. **Palindrome checker.** Write a program that reads in a sequence of strings and checks whether it constitutes a palindrome. Ignore punctuation and spaces and case. (A MAN, A PLAN, A CANAL - PANAMA). Use one stack and one queue.
11. **Streaming algorithm.** Given a long sequence of items, design a data structure to store the k items most recently seen.
12. **2 M/M/1 queues.** Next customer is assigned to the smaller of the two queues. Use 2 FIFO queues. Perception that you always pick the longer line (or wrong lane) when approaching a toll plaza. Suppose two cars enter the toll plaza at the same time and pick different queues of the same length. Compute average length in time that one car will beat the other car by.
13. **M/M/k queue.** Compare k independent M/M/1 queues vs. M/M/k queue.
14. **M/G/1 queue.** Analyze queueing model with a different service distribution (G = general).
15. **Infix to postfix with precedence order.** Write a program to convert an infix expression to postfix. Scan the infix expression from left to right.
    - Operand: output it.
    - Left parentheses: push onto stack.
    - Right parentheses: repeatedly pop elements from the stack and output them until a left parenthesis is encountered. Discard both parentheses.
    - Operator with high precedence than top of stack: push onto stack.
    - Operator with lower or equal precedence than top of stack: repeatedly pop elements from the stack and output them until top of stack has higher precedence. Push the scanned operator onto the stack.

    Afterward, pop remaining elements off stack and output them.
16. **Check for duplicates.** Write a code fragment that determines if a bag contains any duplicate items. Use two nested iterators.
17. **Check for triplicates.** Write a code fragment that determines if a bag contains any item repeated at least three times. Use triply nested iterators.
18. **Equality.** Two queues are equal if they contain the same items in the same order. Two bags are equal if they contain the same items in any order.

19. **Set of integers.** Create a data type that represents a set of integers (no duplicates) between 0 and N-1. Support add(i), exists(i), remove(i), size(), intersect, difference, symmetricDifference, union, isSubset, isSuperSet, and isDisjointFrom. Include an iterator.
20. **Colon.** Experienced programmers know that it's usually a bad idea to write a loop like

```
for (double x = 0.0; x <= N; x += 0.1) {
   ..
}
```

since as a result of floating point precision, the loop will execute 10N times if N = xxx, and 10N + 1 times if N = yyy. Create an data type `Mesh` so that `x` ranges from `left` to `right` in increments of size `delta`. Assuming `right >= left`, the loop should execute *exactly* If `1 + floor((right - left) / delta)` times.

```
for (double x : new Mesh(left, right, delta)) {
   ..
}
```

This is how the colon operator works in MATLAB. You should also instrument your program so that it works even left > right and delta is negative.

21. **List iterators.** We might also want to include methods `hasPrevious()` and `previous()` for going backwards in the list. To implement `previous()` we can use a doubly-linked list. Program DoublyLinkedList.java implements this strategy. It uses Java's `ListIterator` interface to support moving forwards and backwards. We implement all optional methods, including `remove()`, `set()`, and `add()`. The method `remove()` deletes the last element returned by either `next()` or `previous()`. The method `set()` overwrites the value of the last element returned by either `next()` or `previous()`. The method `add()` inserts an element before the next element that would be returned by `next()`. It is legal to call `set()` and `remove()` only after a call either to `next()` or `previous()`, and with no intervening calls to either `remove()` or `add()`.

    We use a dummy head and tail node to avoid extra cases. We also store an extra variable `lastAccessed` which stores the node accessed in the most recent call to `next()` or `previous()`. After removing an element, we reset `lastAccessed` to `null`; this designates that calling `remove()` is illegal (until after a subsequent call to either `next()` or `previous()`.

22. **TwoWayIterator.** Define an interface `TwoWayIterator` that supports four methods: `hasNext()`, `hasPrevious()`, `next()`, and `previous()`. Implement a list that supports a `TwoWayIterator`. *Hint*: implement the list using an array or doubly linked list.
23. **Adding one bag to the end of another.** Write a method that adds the items of one bag b to the end of the invoking bag a. Assume both bags store items of the same type.

    Hint: iterate through the items of b using an iterator, and add each to the end of the invoking bag.

24. **Replace all.** Write a method that replaces all occurrences of an item `from` with the item `to` in a queue or stack.
25. **Adding a list to itself.** What is the result of the following code fragment?

```
List list1 = new ArrayList();
List list2 = new ArrayList();
list1.add(list2);
list2.add(list1);
System.out.println(list1.equals(list2));

List list = new ArrayList();
```

```
list.add(list);
System.out.println(list.hashCode());
```

Answer: stack overflow. The Java documents says that "while it is permissible for lists to contain themselves as elements, extreme caution is advised: the equals and hashCode methods are no longer well defined on a such a list."

26. **Song playlist.** Create an data type that supports the following operations: `enqueue` (add a new song to the end of the list), `play` (print the name of the next song), `skip` (skip the next song in the list, and do not prints its name) and `back` (go back one song). Use a list that supports an iterator with forward and backwards.
27. **Josephus.** Program Josephus.java computes Josephus number.
28. Will the following print out the integers 0 through 9 (in ascending order)?

```
int[] vals = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
for (int val : vals) {
   System.out.print(val + " ");
   StdRandom.shuffle(vals);     // mutate the array while iterating
}
System.out.println();
```

No. It will print out 10 values, but with some duplicates and not in ascending order. The iterator does not save a copy of the original array - instead, it uses the mutated copy.

29. **Queue with one access pointer.** Re-implement a queue, with all operations taking constant time, but only one instance variable (instead of two). *Hint*: use a circular linked list, maintaining a pointer to the last item.
30. **Steque.** A *stack-ended queue* or *steque* is a data type that supports push, pop, and enqueue. Knuth calls it an output-restricted deque. Implement it using a singly-linked list.
31. **Queue with two stacks.** Implement a queue with two stacks so that each queue operations takes a constant amortized number of stack operations. *Hint:* If you push elements onto a stack and then pop them all, they appear in reverse order. If you repeat this process, they're now back in order.

    Solution: QueueWithTwoStacks.java.

32. **Queue with a constant number of stacks.** Implement a queue with a constant number of stacks so that each queue operations takes a constant (worst-case) number of stack operations. *Warning:* Very high degree of difficulty.
33. **Stack with a queue.** Implement a stack with a single queue so that each stack operations takes a linear number of queue operations. *Hint:* to delete an item, get all of the elements on the queue one at a time, and put them at the end, except for the last one which you should delete and return. (Admittedly very inefficient.)
34. **Two stacks with a deque.** Implement two stacks with a single deque so that each operation takes a constant number of deque operations.
35. **Steque with two stacks. (R. Tarjan)** Implement a steque with two stacks so that each steque operation takes a constant amortized number of stack operations.
36. **Deque with a stack and a steque. (R. Tarjan)** Implement a deque with a stack and a steque so that each deque operation takes a constant amortized number of stack and steque operations.
37. **Deque with three stacks. (R. Tarjan)** Implement a deque with three stacks so that each deque operation takes a constant amortized number of stack operations.

38. *Multi-word search.* Program MultiwordSearch.java reads a sequence of query words q[1], ..., q[k] from the command line and a sequence of documents words d[1], ..., d[N] from standard input and finds the shortest interval in which the k words appear in the same order. (Here shortest means the number of words

in the interval.) That is find indices i and j such that d[i1] = q[1], d[i2] = q[2], ..., d[ik] = q[k] and i1 < i2 < ... < ik.

Answer: for each query word, create a sorted list of the indices where it appears in the document. Scan through lists 2 to k in that order, deleting indices at the front of each list until the the first elements of the resulting k lists are in ascending order.

```
q[1]: 50 123 555 1002 1066
q[2]: 33 44 93 333 606 613
q[3]: 60 200
q[4]: 12 18 44 55 203 495

q[1]: 50 123 555 1002 1066
q[2]: 93 333 606 613
q[3]: 200
q[4]: 203 495
```

The sequence of first elements on the lists forms the shortest interval containing the first element on list 1.

Now delete the first element on list 1. Repeatedly delete elements from list 2 until it agrees with list 1. Repeat for list 3, and so on until the whole array is in ascending order. Check this sequence of first elements, etc.

39. *M/M/1 queue.* The Markov/Markov/Single-Server model is a fundamental queueing model in operations research and probability theory. Tasks arrive according to a *Poisson process* at a certain rate λ. This means that λ customers arrive per hour. More specifically, the arrivals follow an exponential distribution with mean 1 / λ: the probability of k arrivals between time 0 and t is (λ t)^k e^(-λ t) / k!. Tasks are serviced in FIFO order according to a Poisson process with rate μ. The two M's standard for Markov: it means that the system is *memoryless*: the time between arrivals is independent, and the time between departures is independent.

    Analysis of M/M/1 model. We are interested in understanding the queueing system. If &lambda > μ the queue size increases without limit. For simple models like M/M/1 we can analyze these quantities analytically using probability theory. Assuming μ > λ, the probability of exactly n customers in the system is (λ / μ)^n (1 - λ / &mu).

    - L = average number of customers in the system = λ / (μ - λ).
    - L_Q = average number of customers in the queue = λ^2 / (μ (μ - λ)).
    - W = average time a customer spends in the system = 1 / (μ - λ).
    - W_Q = average time a customer spends in the queue = W - 1 / μ.

    Program MM1Queue.java For more complex models we need to resort to simulation like this. Variants: multiple queues, multiple servers, sequential multi-stage servers, using a finite queue and measuring number of customers that are turned away. Applications: customers in McDonalds, packets in an internet router,

40. *Listing files.* A Unix directory is a list of files and directories. Program Directory.java takes the name of a directory as a command-line argument and prints out all of the files contained in that directory (and any subdirectories) in level-order. It uses a queue.

41. *Interrupt handling.* When programming a real-time system that can be interrupted (e.g., by a mouse click or wireless connection), it is necessary to attend to the interrupts immediately, before proceeding with the current activity. If the interrupts should be handles in the same order they arrive, then a FIFO queue is the appropriate data structure.

42. *Library implementation.* Java has a built in library called `Stack`, but you should avoid using it. It has additional operations that are not normally associated with a stack, e.g., getting the ith element and adding an element to the bottom of the stack (instead of the top). Although having such extra operations may appear to be a bonus, it is actually a curse. We use ADTs not because they provide every available operation, but rather because they *limit* the types of operations we are allowed to perform! This prevents us from performing operations that we don't actually want. If we need more than just LIFO access, we should use a different data type. We can still build a stack data type from the Java libraries, but we are careful to limit the types of operations. There is no Java queue implementation.
43. *Load balancing.* N users must choose among N identical servers in a network. Goal: balance users across resources. Checking each resource to find an empty one (or least busy one) is too expensive. Instead, choose a random server. At any step, you should be able to see which jobs are on each machine. Program Server.java Plot distribution of loads. Theory: avg load = 1, max load = log N / log log N.
44. *Load balancing reloaded.* (Azar, Broder, Karlin, and Upfal) Choose two random resources. Insert on least busy of the two. Theory: avg load = 1, max load = log log N.
45. *Gridding.* Given N Euclidean points in the unit box and a parameter d, find all pairs that are within distance d. Divide box into a G-by-G grid where G = ceil(1/d). Put all points in a list in given grid cell. Any neighbor within distance d must be in that cell or one of its 8 neighbors. Program Grid.java implements this strategy using the helper data type Point2D.java.
46. *Java library.* Java contains library classes `LinkedList` and `ArrayList` that implement a list. Has wider interface than our `Sequence` data type: access an element by its index, delete an element, search for an element. No urns.
47. Add a method `dup()` to `Stack` that creates a second copy of the topmost element and pushes it onto the stack.
48. Add a method `exch()` to `Stack` that exchanges the top two elements on the stack.
49. Add a method `size()` to `Stack` that returns the number of elements on the stack.
50. Add a method `Item[] multiPop(int k)` to `Stack` that pops k elements from the stack and returns them as an array of objects.
51. Add a method `Item[] toArray()` to `Queue` that returns all N elements on the queue as an array of length N.
52. Write a recursive function that takes as input a queue, and rearranges it so that it is in reverse order. Hint: dequeue the first element, recursively reverse the queue, and the enqueue the first element.
53. Given a queue, create two new queues q1 and q2 so that q1 contains the even elements of q and q2 contains the odd elements, e.g., as in dealing a deck of cards.
54. What does the following code fragment do?

```
Queue<Integer> q = new Queue<Integer>();
q.enqueue(0);
q.enqueue(1);
for (int i = 0; i < 10; i++) {
    int a = q.dequeue();
    int b = q.dequeue();
    q.enqueue(b);
    q.enqueue(a + b);
    System.out.println(a);
}
```

55. What data type would you choose to implement an "Undo" feature in a word processor?
56. Suppose you have a single array of size N and want to implement two stacks so that you won't get overflow until the total number of elements on both stacks is N+1. How would you implement this?
57. Suppose that you implemented `push` in the linked list implementation of Stack.java with the following code. What is the mistake?

```
public void push(Item item) {
   Node second = first;
   Node first = new Node();
```

```
        first.item = item;
        first.next = second;
    }
```

Answer: By redeclaring `first`, you are create a new local variable named `first`, which is different from the instance variable named `first`.

58. **Minimum stack.** Design a data type that implements the following operations, all in constant time: push, pop, min. Assume that the items are `Comparable`.

    Solution:: maintain two stacks, one which contains all of the items and another which contains the minima. To push an item, push it on the first stack; if it is smaller than the topmost item on the second stack, push it on the second stack as well. To pop an item, pop it from the first stack; if it is the top item on the second stack, pop it from the second stack as well. To find the minimum, return the top item on the second stack.

59. **Doubling and halving.** What is the effect of replacing the halving test in ResizingArrayStack.java from `if (N > 0 && N == a.length/4) resize(a.length/2);` to `if (N == a.length/4) resize(2*N);`?
60. **Shunting-yard algorithm.** Implement Dijkstra's shunting-yard algorithm to convert an infix expression into a postfix expression. Support operator precedence, including both left and right associative operators.
61. **FIFO queue with random deletion.** Implement a data type that suports *insert an item*, *delete the item added least recently*, and *delete a random item*. Each operation should take constant expected amortized time per operation and should use space (at most) proportional to the numer of items in the data structure.
62. **Stock prices.** Given an array of daily stock prices `prices[]`, create an array `days[]` so that `days[i]` tells you how many days you have to wait, starting at day `i`, until the stock price exceeds `prices[i]`.

    Hint: your algorithm should take linear time and use a stack of array indices.

Last modified on January 30, 2018.