ADDENDUM Y Dispensers and dictionaries

To this point, the only containers we have seen have been lists. We conclude with a brief look at two additional kinds of containers: *dispensers* and *dictionaries*. A *dispenser* is a container to which we can freely add items but restricts item access and removal.

Figure y.1 **A gumball dispenser.**

A *dictionary* is a container in which elements are accessed by *key*. For instance, a telephone directory is a dictionary in which items are accessed by name. The name serves as a key to access the telephone number.

We specify dictionaries and three very commonly used dispensers: stacks, queues, and priority queues. We consider straightforward implementations and see how *ListImplementation* classes can be adapted to provide dispenser implementations. Efficient implementations of priority queues and dictionaries, however, are beyond the scope of the text.

y.1 Dispensers

As we've said, a dispenser is a container that restricts access to its elements. In particular, there is one element in the container that we will call the *current element*. This is the only element that can be accessed or removed from the container. Removing the current element, of course, causes another element to become current. Adding an element to the dispenser may or may not change the current element, depending on the type of dispenser.

We assume that a dispenser has three essential features: a method for adding items to the container, a method for removing items, and a method for accessing items. Different kinds of dispensers use different names for these operations, but we'll refer to them generically as `add`, `remove`, and `get`:

```
void add (Element element)
```
Add the specified element to this dispenser.

```
void remove ()
```
Remove the current element from this dispenser.

```
Element get ()
```
The current element of this dispenser.

The query `get` allows us to access the current item, and the command `remove` removes the current item from the dispenser.

There are many ways to define and implement dispensers. One common approach combines access and removal into one operation. That is, there is a method

```
Element dispenseItem ()
```

that removes an item from the dispenser, and returns the removed item. That is,

```
obj = dispenser.dispenseItem();
```

is equivalent to

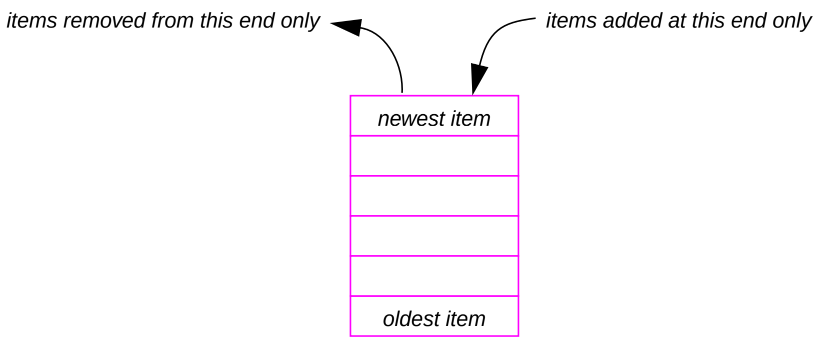
```
obj = dispenser.get();
dispenser.remove();
```

While `dispenseItem` may capture the intuitive semantics of the word "dispenser" – a dispenser gives us one of its items, thereby removing it from the dispenser – we adopt the former approach. We prefer to differentiate commands from queries, and not unnecessarily introduce public queries that change an object's state.

y.2 Stacks

A *stack* is a simple dispenser in which the current item is the container item that has been added most recently. That is, it is the youngest or newest item in the container. Stacks can be implemented efficiently, and are used in many diverse applications.

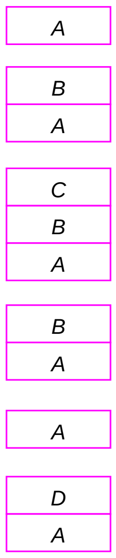
An obvious model of a stack is a list in which items are added, accessed, and removed from one end only. The name "stack" derives from the image conveyed by this model. A stack is sometimes called a *last-in first-out*, or *LIFO*, list. The end to which items are added and from which they are removed is referred to as the *top*. (The other end is obviously the *bottom*.)


items removed from this end only
items added at this end only
newest item
oldest item

In the context of a stack, the features `get`, `add`, and `remove` are traditionally named `top`, `push`, and `pop`. The specification is given in Listing y.1. The following illustrates adding items to and removing items from a stack.

Assume `s` is an initially empty `Stack`:

```
s.push(A);  // s.top() is A

s.push(B);  // s.top() is B

s.push(C);  // s.top() is C

s.pop();    // s.top() is B

s.pop();    // s.top() is A

s.push(D);  // s.top() is D
```



stack: a dispenser in which the current element is the container element most recently added to the container.

Listing y.1 The interface *Stack*

Interface Stack<Element>

```
public interface Stack<Element>
```

Dispenser adhering to a last-in/first-out discipline.

Queries

```
public boolean isEmpty ()
```

This *Stack* contains no elements.

```
public boolean isFull ()
```

This *Stack* contains a maximum number of elements.

```
public Element top ()
```

The element of this *Stack* that was most recently added.

require:

```
!this.isEmpty()
```

Commands

```
public void push (Element element)
```

Add the specified element to this *Stack*.

require:

```
!this.isFull()
element != null
```

ensure:

```
!this.isEmpty()
```

```
public void pop ()
```

Remove the element of this *Stack* that was most recently added.

require:

```
!this.isEmpty()
```

```
public void clear ()
```

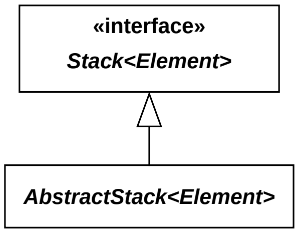
Remove all the elements from this *Stack*.

ensure:

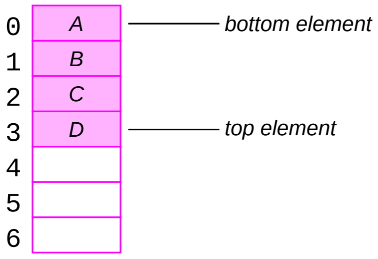
```
this.isEmpty()
```

y.2.1 *Stack* implementations

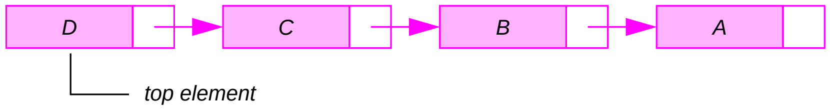
We can use the same pattern for implementing dispensers as we did for implementing lists. That is, we can provide a generic abstract class to serve as a basis for implementations. However, with the specification we have given for *Stack*, there is not much for the abstract class to do except provide a `toString` method and a default implementation of `clear`.



We can clearly build stacks in the same ways that we built lists, with arrays or linked structures. In an array-based implementation, we make the element with highest index the top. This allows adding and removing to be done in constant time, without the need to shuffle the entire array:



With a linked implementation, we make the first element the top. Again this allows us to add and remove elements in constant time:



Now we have already developed classes for manipulating array-based and linked lists: specifically, *BoundedList* and *LinkedList*. These classes have features for adding, removing, and accessing arbitrary elements. But the specifications of these features are not exactly as we want them. For instance, there is no command `push` to add an element to the front of a *LinkedList*. We must use the command `add`:

```
list.add(0,element);
```

We translate *Stack* features into *BoundedList* or *LinkedList* features by *wrapping* or *adapting* in much the same way as we built *DefaultList* by wrapping the class *Vector*. That is, we make the existing class a component of the new class, and forwarded responsibili-

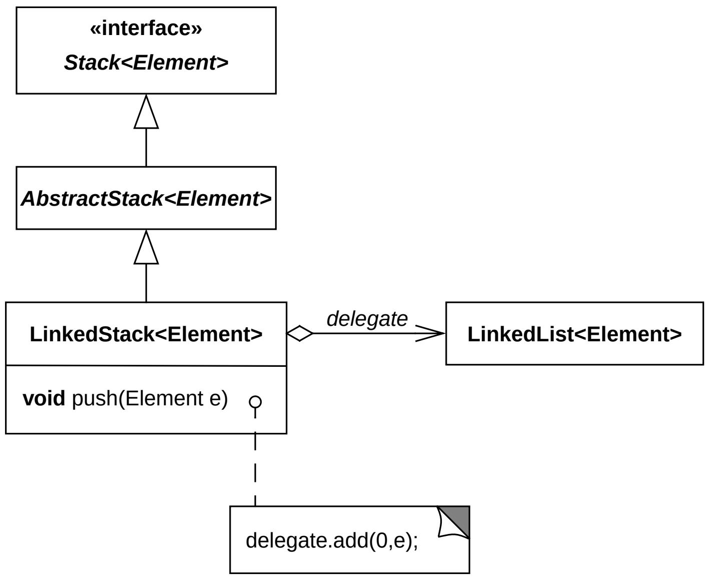


Figure y.2 **An adapter wrapping the class *LinkedList*.**

ties to the component. For instance, we can define a class *LinkedStack* with a *LinkedList* component as shown in Figure y.2.

It is also possible to adapt an existing class by extending it. For example, as illustrated in Figure y.3, we can define an adapter that extends *LinkedList* and implements *Stack*. *Stack* features are implemented by calls to appropriate *LinkedList* methods.

We can easily construct similar adapters for other *List*-like classes such as *BoundedList*.

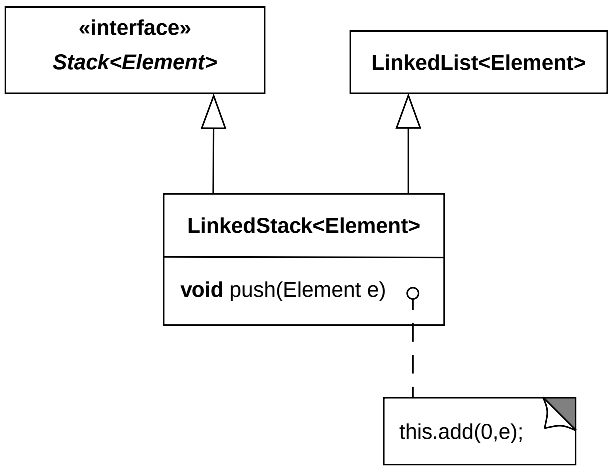
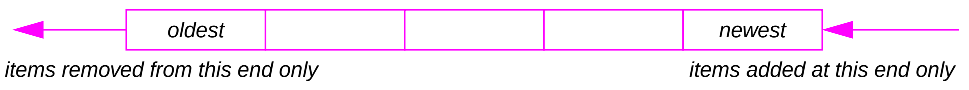


Figure y.3 **An adapter extending the class *LinkedList*.**

y.3 Queues

A *queue* is a simple dispenser in which the current item is the one least recently added to the container; that is, the oldest item in the container. Queues are also commonly used in many applications.

An obvious model of a queue is a list in which items are added to one end, and accessed and removed from the other end:


oldest
newest
items removed from this end only
items added at this end only

A queue is sometimes called a *first-in first-out*, or *FIFO*, list. The end at which items are added is the *rear* of the queue, and the end from which items are removed is the *front*.

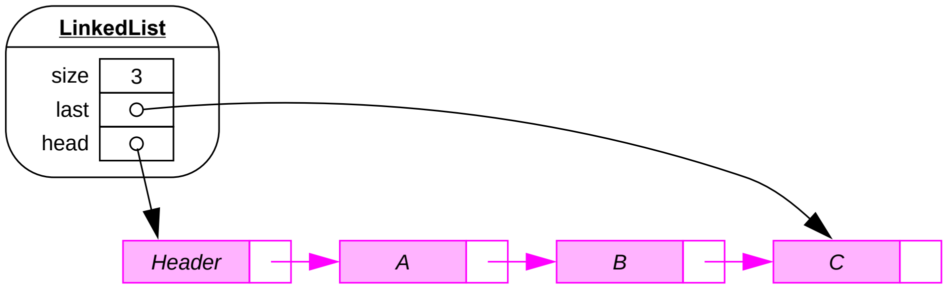
In the context of a queue, the features `add` and `remove` are often called *enqueue* and *serve*. We name the fundamental dispenser features `front`, `append`, and `remove`, and give a specification in Listing y.2.

queue: a dispenser in which the current element is the container element least recently added to the container.

y.3.1 *Queue* implementations

Linked implementations

As for stacks, we can develop array-based and linked implementations for queues. If a *LinkedList* maintains references to both ends of the list, elements can be added to either end in constant time.



However, constant time deletes can be done only from the front of the list. This implies that we should make the front of the queue be the front of the *LinkedList*. An adapter class can easily be defined in much the same way as was done for *Stack*.

Listing y.2 The interface *Queue*

Interface Queue<Element>

```
public interface Queue<Element>
```

Dispenser adhering to a first-in/first-out discipline.

Queries

```
public boolean isEmpty ()
```

This *Queue* contains no elements.

```
public boolean isFull ()
```

This *Queue* contains a maximum number of elements.

```
public Element front ()
```

The element of this *Queue* that was least recently added.

require:

```
!this.isEmpty()
```

Commands

```
public void append (Element element)
```

Add the specified element to this *Queue*.

require:

```
!this.isFull()
element != null
```

ensure:

```
!this.isEmpty()
```

```
public void remove ()
```

Remove the element of this *Queue* that was least recently added.

require:

```
!this.isEmpty()
```

```
public void clear ()
```

Remove all the elements from this *Queue*.

ensure:

```
this.isEmpty()
```

Circular arrays

If we attempt to use the class *BoundedList* to implement queues, we encounter a problem. While we can add and remove elements from one end of the list in constant time, adding and removing elements from the other end requires shuffling the list, and takes linear time. Thus if we implement queues with *BoundedLists*, we can make either `append` or `remove` constant time (by appropriately choosing which end of the list is the front of the queue), but the other command will be linear.

An array-based approach that permits all queue methods to operate in constant time is to view an array logically as a circular structure, in which the element with index 0 follows the highest indexed element. This is illustrated in Figure y.4.

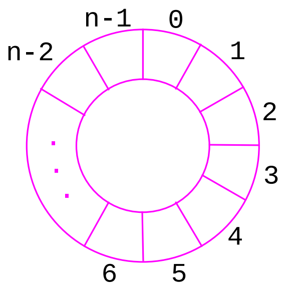
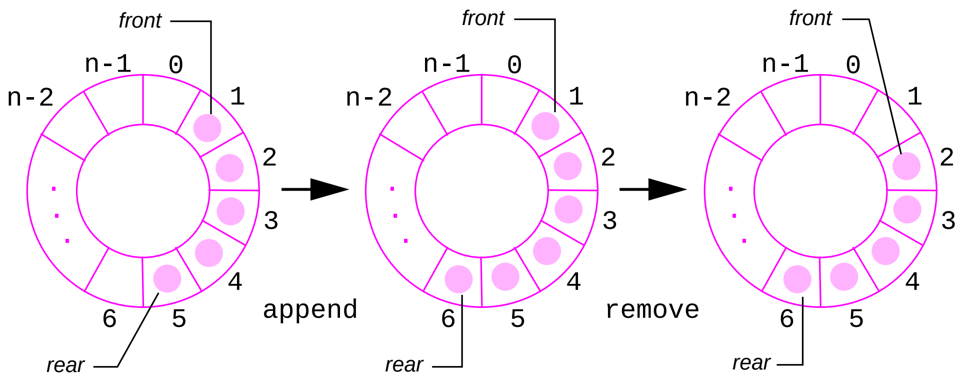


Figure y.4 **A circular array of size *n*.**

The queue occupies a set of contiguous array elements, and "circulates" through the array as items are added and removed.



The implementation maintains two indexes, `front` and `rear`, identifying the front and rear elements of the queue; `front` is advanced when an item is removed from the queue, and `rear` is advanced when an item is added.

Note that when the queue has one element, `front == rear`. Removing an element increments `front`. Thus the relationship between these indexes is the same for both the full queue and the empty queue. That is, `(rear+1)%n == front` for both the empty and full queue, where `n` is the length of the array. The implementation is straightforward and is given in Listing y.3.

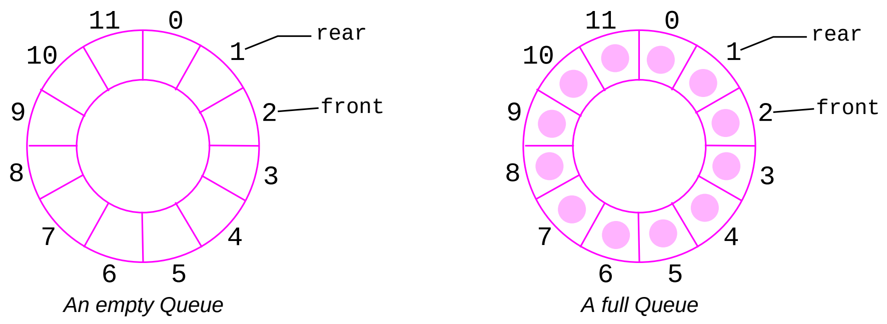


Figure y.5 **Empty and full *Queues*, implemented with a circular array.**

Listing y.3 The class *CircularQueue*

```
/**
 * Circular array implementation of the interface Queue
 */
class CircularQueue<Element> implements Queue<Element> {

   private Object[] elements;
   private int front;   // index of the front Queue item
   private int rear;    // index of the rear Queue item
   private int size;    // size of the Queue

   /**
    * Create a CircularQueue with specified maximum size.
    * @require    maxSize >= 0
    * @ensure     this.isEmpty()
    */
   public CircularQueue (int maxSize) {
      elements = new Object[maxSize];
      size = 0;
      front = 0;
      rear = maxSize-1;
   }

   /**
    * The maximum size of this Queue.
    * @ensure     this.maxSize() >= 0
    */
   public int maxSize () {
      return elements.length;
   }
```

continued

Listing y.3 **The class *CircularQueue (cont'd)***

```
/**
 * The number of elements in this Queue.
 * @ensure     this.size() >= 0 &&
 *             this.size() <= this.maxSize()
 */
public int size () {
   return this.size;
}

/**
 * This Queue contains no elements.
 */
public boolean isEmpty () {
   return this.size == 0;
}

/**
 * This Queue contains a maximum number of elements.
 */
public boolean isFull () {
   return this.size == elements.length;
}

/**
 * The element of this Queue least recently added.
 * @require    !this.isEmpty()
 */
public Element front () {
   return (Element)elements[front];
}

/**
 * Add a new element to this Queue.
 * @require    element != null && !this.isFull()
 * @ensure     !this.isEmpty()
 */
public void append (Element element) {
   rear = next(rear);
   elements[rear] = element;
   size = size+1;
}
```

continued

Listing y.3 **The class *CircularQueue (cont'd)***

```
   /**
    * Remove the element of this Queue that was least
    * recently added.
    * @require    !this.isEmpty()
    */
   public void remove () {
      front = next(front);
      size = size-1;
   }

   /**
    * Remove all the elements from this Queue.
    * @ensure     this.isEmpty()
    */
   public void clear () {
      size = 0;
      front = 0;
      rear = maxSize-1;
   }

   /**
    * A String representation of this Queue.
    */
   public String toString () {
      String s = "[";
      if (size > 0) {
         s = s + elements[front].toString();
         int i;
         for (i = front; i != rear; i = next(i))
            s = s + ", " + elements[next(i)].toString();
      }
      s = s + "]";
      return s;
   }

   /**
    * The next index, mod length of the array.
    */
   private int next (int index) {
      return (index+1) % elements.length;
   }

}
```

y.4 Priority queues

Suppose there is an ordering on the component class of a container, and we want to retrieve the items based on the ordering. The container contains instances of the class *Student*, for example, and we want to retrieve *Student*'s in order of final grade. In such a situation, a *priority queue* is an appropriate structure to use.

A *priority queue* is a dispenser in which the current item is a largest item in the container with respect to some given ordering. The ordering is called a *priority*, and a first item with respect to the ordering is referred to as a *highest priority* item. We name the dispenser features `highest`, `add`, and `remove`, and give a specification in Listing y.4.

priority queue: a dispenser in which the current element is a largest container element with respect to some given ordering.

Listing y.4 **The interface *PriorityQueue***

Interface PriorityQueue<Element>

public interface `PriorityQueue<Element>`
Dispenser adhering to a priority-out discipline.

Queries

public boolean `isEmpty ()`
This *PriorityQueue* contains no elements.

public boolean `isFull ()`
This *PriorityQueue* contains a maximum number of elements.

public `Element highest ()`
An element of this *PriorityQueue* with highest priority.

require:
`!this.isEmpty()`

ensure:
for each element `e` in this *PriorityQueue*
`!this.priority().inOrder(e,this.highest())`

public `Order<Element> priority ()`
The priority used to order this *PriorityQueue*.

continued

Listing y.4 **The interface *PriorityQueue (cont'd)***

Commands

```
public void add (Element element)
```
Add the specified element to this *PriorityQueue*.

require:
`!this.isFull()`

ensure:
`!this.isEmpty()`

```
public void remove ()
```
Remove the element `this.highest()` from this *PriorityQueue*.

require:
`!this.isEmpty()`

```
public void clear ()
```
Remove all the elements from this *PriorityQueue*.

ensure:
`this.isEmpty()`

y.4.1 *PriorityQueue* implementations

An obvious way to implement a priority queue is with the class *OrderedList*, introduced in Section 14.3. Recall that an ordering is provided when an *OrderedList* is created, and the elements on the list are maintained in increasing order. Otherwise, the features of an *OrderedList* are similar to those of a *List*.

There are a number of different and very efficient approaches to implementing *PriorityQueues*. A discussion of these structures, however, is beyond the scope of this text.

y.5 Dictionaries

A *dictionary*, sometimes called a *key-value table*, is a container in which elements are accessed by *key*. We think of the entries in a dictionary as having two components, a *key* and an associated *value* – a *key-value pair*. In a dictionary of the English language, for instance, an English word is the key and the definition is the value. In a telephone directory, a name is the key and the telephone number is the value. When we use a dictionary, we have a key and are interested in obtaining the associated value.

We can also consider the key to be an attribute of the entry, rather than a separate component. For instance, we can view the telephone directory as containing records that consist of name, address, and telephone number. The key – the name – is simply one of the record attributes. We adopt the former point of view, though, as it is more consistent with standard Java library classes.

The features of a dictionary are similar to those of a dispenser. The fundamental difference is that a key must be provided to access or delete an item. A question that arises is whether there can be several elements in the dictionary with the same key. We assume keys are unique. That is, we assume there can be at most one element in the dictionary with any given key. We give an elementary specification in Listing y.5.

We can build a straightforward implementation of a dictionary with a *List* whose elements are key-value pairs. The methods `get` and `remove` simply search the *List* to locate the item with the given key. However, there are much better ways to implement dictionaries. As with priority queues, a discussion of the implementing structures are beyond the scope of the text.

Finally, we mention that the standard package `java.util` defines an interface *Map* that serves as a superclass for dictionary variants.

dictionary: a container in which the elements are accessed by key.

Listing y.5 The interface *Dictionary*

Interface Dictionary<Key, Element>

`public interface` `Dictionary<Key, Element>`

Container in which elements are uniquely accessed by key.

Queries:

`public boolean` `isEmpty ()`

This *Dictionary* contains no entries.

`public` `Element get (Key key)`

The element of this *Dictionary* associated with the specified key. *null* if there is no entry with the specified key.

continued

Listing y.5 **The interface *Dictionary (cont'd)***

Commands:

```
public void add (Key key, Element value)
```

Add an entry with the specified key and value to this *Dictionary.* If this *Dictionary* already contains an entry with the specified key, the value associated with this entry is replaced by the specified value.

ensure:

```
!this.isEmpty()
```

```
public void remove (Key key)
```

Remove the entry `this.get(key)` from this *Dictionary.* If this *Dictionary* does not contain an entry with the specified key, this method does nothing.

```
public void clear ()
```

Remove all the entries from this *Dictionary.*

ensure:

```
this.isEmpty()
```

y.6 Summary

With this chapter we conclude an introductory overview of containers by briefly discussing the abstraction dispenser. We discussed the fundamental notion of a dispenser and its principal variants, stacks, queues, and priority queues. We also considered dictionaries. For each of these we presented a specification, and discussed elementary implementations using existing list implementations such as *BoundedList* and *LinkedList*.

These last few chapters dealing with containers should be viewed as an introduction to data structuring within the context of object orientation and the methodology used in this text. Our intention has been to use these topics as a case study for presenting design choices for class and library design, and also to suggest how the methodology might be continued to subsequent topics.

EXERCISES

y.1 Complete a *Stack* implementation based on the class *BoundedList*.

y.2 Complete a *Stack* implementation based on the class *LinkedList*.

y.3 Implement the class *PriorityQueue*.

y.4 Implement the class *Dictionary.*

y.5 Carefully read the specifications for the class *java.util.Hashtable*. Define an implementation of *Dictionary* based on this class.

y.6 A *prefix integer expression* can be defined to be either an integer, or two prefix integer expressions preceded by a binary operator:

prefixIntegerExpression =
integer or
binaryOperator prefixIntegerExpression prefixIntegerExpression

For instance, the following are prefix integer expressions:

```
1     2     + 1 2       * + 1 2 4      * + 1 2 + 1 2
```

To evaluate an expression, evaluate the subexpressions and then apply the operator. For instance,

```
* + 1 2 + 1 2  ⇒  * 3 + 1 2  ⇒  * 3 3  ⇒  9
```

Class *Operator* models binary operators, and class *Operand* modes integers. Both are subclasses of class *Token*.

A *TokenList* can now be interpreted as a prefix integer expression.

Define classes *Operator*, *Operand*, and *Token*. Implement a method that uses a stack to evaluate prefix integer expressions.

GLOSSARY

dictionary: a container in which elements are accessed by key. Also known as a *key-value table*.

dispenser: a container that allows access and removal of elements in a predetermined way. A current element is distinguished as the element that can be accessed and removed.

priority queue: a dispenser in which the current element is the largest item in the container with respect to some given ordering.

queue: a dispenser in which the current element is the container element least recently added to the container.

stack: a dispenser in which the current element is the container element most recently added to the container.

18 Addendum yDispensers and dictionaries