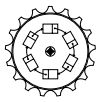
Chapter 10

Linear Structures

Concepts:
- ▷ Stacks
- ▷ Queues

"Rule Forty-two.
All persons more than a mile high to leave the court."...
"Well, I shan't go," said Alice; "Besides
that's not a regular rule: you just invented it now."
"It's the oldest rule in the book," said the King.
"Then it ought to be Number One," said Alice.
—Charles Lutwidge Dodgson

THE STATE OF SOME STRUCTURES REFLECTS THEIR HISTORY. Many systems—for example, a line at a ticket booth or, as Alice presumes, the King's book of rules—modify their state using two pivotal operations: *add* and *remove*. As with most data structures, when values are added and removed the structure grows and shrinks. *Linear structures*, however, shrink in a predetermined manner: values are removed in an order based only on the order they were added. All linear structures abide by a very simple interface:



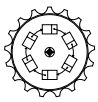
Linear

```
public interface Linear<E> extends Structure<E>
{
    public void add(E value);
    // pre: value is non-null
    // post: the value is added to the collection,
    //       the consistent replacement policy is not specified

    public E get();
    // pre: structure is not empty
    // post: returns reference to next object to be removed

    public E remove();
    // pre: structure is not empty
    // post: removes an object from store

    public int size();
    // post: returns the number of elements in the structure

    public boolean empty();
    // post: returns true if and only if the linear structure is empty
}
```

For each structure the `add` and `remove` methods are used to insert new values into the linear structure and remove those values later. A few utility routines are

also provided: `get` retrieves a copy of the value that would be removed next, while `size` returns the size of the structure and `empty` returns whether or not the linear structure is empty.

One thing that is important to note is that the `empty` method seems to provide the same features as the `isEmpty` method we have seen in previous structures. This is a common feature of ongoing data structure design—as structures evolve, aliases for methods arise. In this case, many native classes of Java actually have an `empty` method. We provide that for compatibility; we implement it in our abstract implementation of the `Linear` interface. The `empty` method simply calls the `isEmpty` method required by the `Structure` interface and ultimately coded by the particular `Linear` implementation.



`AbstractLinear`

```
abstract public class AbstractLinear<E> extends AbstractStructure<E>
    implements Linear<E>
{
    public boolean empty()
    // post: return true iff the linear structure is empty
    {
        return isEmpty();
    }

    public E remove(E o)
    // pre: value is non-null
    // post: value is removed from linear structure, if it was there
    {
        Assert.fail("Method not implemented.");
        // never reaches this statement:
        return null;
    }
}
```

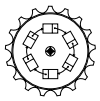
Since our `Linear` interface extends our notion of `Structure`, so we will also be required to implement the methods of that interface. In particular, the `remove(Object)` method is required. We use the `AbstractLinear` implementation to provide a default version of the `remove` method that indicates it is not yet implemented. The benefits of making `Linear` classes instances of `Structure` outweigh the perfect fit of the `Structure` interface.

Since `AbstractLinear` extends `AbstractStructure`, any features of the `AbstractStructure` implementation are enjoyed by the `AbstractLinear` interface as well.

We are now ready to implement several implementations of the `Linear` interface. The two that we will look at carefully in this chapter, *stacks* and *queues*, are the most common—and most widely useful—examples of linear data structures we will encounter.

10.1 Stacks

Our first linear structure is a *stack*. A stack is a collection of items that exhibit the behavior that *the last item in is the first item out*. It is a *LIFO* ("lie-foe") structure. The `add` method pushes an item onto the stack, while `remove` pops off the item that was pushed on most recently. We will provide the traditionally named methods `push` and `pop` as alternatives for `add` and `remove`, respectively. We will use these stack-specific terms when we wish to emphasize the LIFO quality. A nondestructive operation, `get`, returns the top element of the `Stack`—the element that would be returned next. Since it is meaningless to `remove` or `get` a `Stack` that is empty, it is important to have access to the size methods (`empty` and `size`). Here is the interface that defines what it means to be a `Stack`:



`Stack`

```
public interface Stack<E> extends Linear<E>
{
    public void add(E item);
    // post: item is added to stack
    //       will be popped next if no intervening add

    public void push(E item);
    // post: item is added to stack
    //       will be popped next if no intervening push

    public E remove();
    // pre: stack is not empty
    // post: most recently added item is removed and returned

    public E pop();
    // pre: stack is not empty
    // post: most recently pushed item is removed and returned

    public E get();
    // pre: stack is not empty
    // post: top value (next to be popped) is returned

    public E getFirst();
    // pre: stack is not empty
    // post: top value (next to be popped) is returned

    public E peek();
    // pre: stack is not empty
    // post: top value (next to be popped) is returned

    public boolean empty();
    // post: returns true if and only if the stack is empty

    public int size();
    // post: returns the number of elements in the stack
```

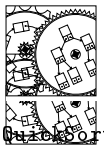
```
}
```

To maintain applications that are consistent with Java's `java.util.Stack` the alternative operations of `push` and `pop` may be preferred in some of our discussions.

10.1.1 Example: Simulating Recursion

Earlier we mentioned that tail recursive methods could be transformed to use loops. More complex recursive methods—especially those that perform multiple recursive calls—can be more complex to implement. In this section, we focus on the implementation of an iterative version of the quicksort sorting algorithm. Before we turn to the implementation, we will investigate the calling mechanisms of languages like Java.

Data available to well-designed methods come mainly from two locations: the method's parameters and its local variables. These values help to define the method's current *state*. In addition to the explicit values, implicit parameters also play a role. Let us consider the recursive version of quicksort we saw earlier:



QuickSort

```
private static void quickSortRecursive(int data[],int left,int right)
// pre: left <= right
// post: data[left..right] in ascending order
{
    int pivot;   // the final location of the leftmost value
    if (left >= right) return;
    pivot = partition(data,left,right);    /* 1 - place pivot */
    quickSortRecursive(data,left,pivot-1); /* 2 - sort small */
    quickSortRecursive(data,pivot+1,right);/* 3 - sort large */
    /* done! */
}
```

The *flow of control* in most methods is from top to bottom. If the machine stops execution, it is found to be executing one of the statements. In our recursive quicksort, there are three main points of execution: (1) before the first recursive call, (2) before the second recursive call, and (3) just before the return. To focus on what needs to be accomplished, the computer keeps a special reference to the code, called a *program counter*. For the purposes of our exercise, we will assume the program counter takes on the value 1, 2, or 3, depending on its location within the routine.

These values—the parameters, the local variables, and the program counter—reside in a structure called a *call frame*. Whenever a method is called, a new call frame is constructed and filled out with appropriate values. Because many methods may be active at the same time (methods can call each other), there are usually several active call frames. These frames are maintained in a *call stack*. Since the first method to return is the last method called, a stack seems appropriate. Figure 10.1 describes the relationship between the call frames of the call stack and a partial run of quicksort.

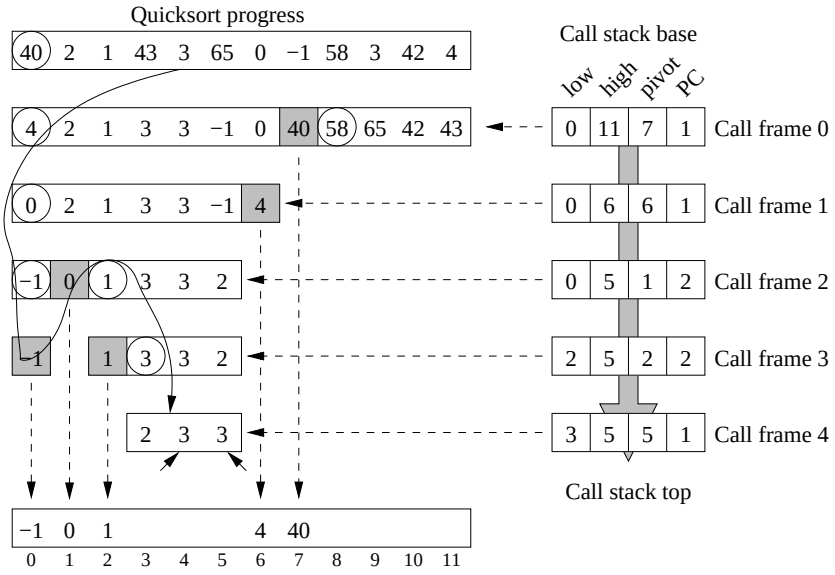


Figure 10.1 A partially complete quicksort and its call stack. The quicksort routine is currently partitioning the three-element subarray containing 2, 3, and 3. The curve indicates the progress of recursion in quicksort to this point.

Our approach is to construct a stack of frames that describes the progress of the various "virtually recursive" calls to quicksort. Each frame must maintain enough information to simulate the actual call frame of the recursive routine:

```
class callFrame
{
    int pivot;// location of pivot
    int low;  // left index
    int high; // right index
    int PC; // next statement (see numbering in recursive code)

    public callFrame(int l, int h)
    // post: generate new call frame with low and high as passed
    {
        low = l; high = h; PC = 1;
    }
}
```

Just as with real call frames, it's possible for variables to be uninitialized (e.g., `pivot`)! We can consider an iterative version of quicksort:

```
public static void quickSortIterative(int data[], int n)
// pre: n <= data.length
// post: data[0..n-1] in ascending order
{
    Stack<callFrame> callStack = new StackList<callFrame>();
    callStack.push(new callFrame(0,n-1));
    while (!callStack.isEmpty())
    {   // some "virtual" method outstanding
        callFrame curr = callStack.get();
        if (curr.PC == 1) { // partition and sort lower
            // return if trivial
            if (curr.low >= curr.high) { callStack.pop(); continue; }
            // place the pivot at the correct location
            curr.pivot = partition(data,curr.low,curr.high);
            curr.PC++;
            // sort the smaller values...
            callStack.push(new callFrame(curr.low,curr.pivot-1));
        } else if (curr.PC == 2) { // sort upper
            curr.PC++;
            // sort the larger values....
            callStack.push(new callFrame(curr.pivot+1,curr.high));
        } else { callStack.pop(); continue; } // return
    }
}
```

We begin by creating a new stack initialized with a `callFrame` that simulates first call to the recursive routine. The `low` and `high` variables of the frame are initialized to 0 and $n-1$ respectively, and we are about to execute statement 1. As long as there is a call frame on the stack, some invocation of quicksort is

still executing. Looking at the top frame, we conditionally execute the code associated with each statement number. For example, statement 1 returns (by popping off the top call frame) if `low` and `high` suggest a trivial sort. Each variable is prefixed by `curr` because the local variables reside within the call frame, `curr`. When we would execute a recursive call, we instead increment the "program counter" and push a new frame on the stack with appropriate initialization of local variables. Because each local variable appears in several places on the call stack, recursive procedures appear to have fewer local variables. The conversion of recursion to iteration, then, requires more explicit handling of local storage during execution.

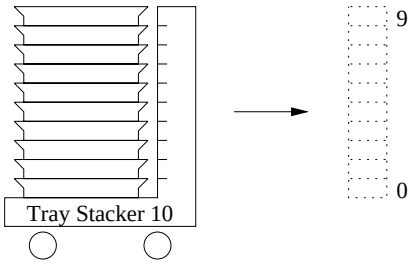
The sort continues until all the call frames are popped off. This happens when each method reaches statement 3, that is, when the recursive quicksort would have returned.

We now discuss two implementations of `Stacks`: one based on a `Vector` and one based on a `List`.

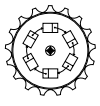
10.1.2 Vector-Based Stacks

Let us consider a traditional stack-based analogy: the storage of trays in a fast-food restaurant. At the beginning of a meal, you are given a tray from a stack. The process of removing a tray is the popping of a tray off the stack. When trays are returned, they are pushed back on the stack.

Now, assume that the side of the tray holder is marked in a rulerlike fashion, perhaps to measure the number of trays stacked. Squinting one's eyes, this looks like a sideways vector:



Following our analogy, the implementation of a `Stack` using a `Vector` can be accomplished by aligning the "top" of the `Stack` with the "end" of the `Vector` (see Figure 10.2). We provide two constructors, including one that provides the `Vector` with the initial capacity:



StackVector

```
protected Vector<E> data;

public StackVector()
// post: an empty stack is created
{
    data = new Vector<E>();
}
```

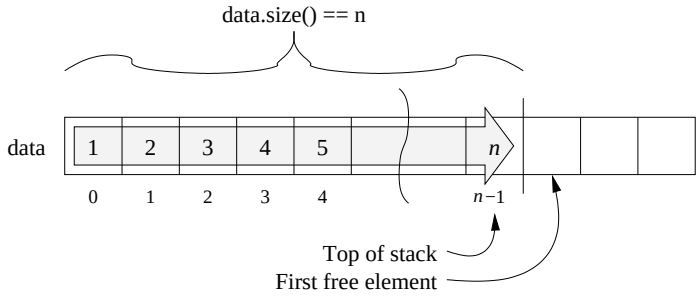


Figure 10.2 A `Vector`-based stack containing n elements. The top of the stack is implied by the length of the underlying vector. The arrow demonstrates the direction of growth.

```
public StackVector(int size)
// post: an empty stack with initial capacity of size is created
{
    data = new Vector<E>(size);
}
```

To `add` elements on the `Stack`, we simply use the `add` method of the `Vector`. When an element is to be `removed` from the `Stack`, we carefully remove the last element, returning its value.

```
public void add(E item)
// post: item is added to stack
//       will be popped next if no intervening add
{
    data.add(item);
}

public E remove()
// pre: stack is not empty
// post: most recently added item is removed and returned
{
    return data.remove(size()-1);
}
```

The `add` method appends the element to the end of the `Vector`, extending it if necessary. Notice that if the vector has n elements, this element is written to slot n, increasing the number of elements to $n + 1$. Removing an element reverses this process: it removes and returns the last element. (Note the invocation of our principle of symmetry, here. We will depend on this notion in our design of a number of `add` and `remove` method pairs.) The `get` method is like `remove`,

except that the stack is not modified. The size of the stack is easily determined by requesting the same information from the underlying `Vector`. Of course when the `Vector` is empty, so is the `Stack` it supports.[1]

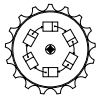
```
public boolean isEmpty()
// post: returns true if and only if the stack is empty
{
    return size() == 0;
}

public int size()
// post: returns the number of elements in stack
{
    return data.size();
}

public void clear()
// post: removes all elements from stack
{
    data.clear();
}
```

The `clear` method is required because the `Stack` indirectly extends the `Structure` interface.

10.1.3 List-Based Stacks

Only the top "end" of the stack ever gets modified. It is reasonable, then, to seek an efficient implementation of a `Stack` using a `SinglyLinkedList`. Because our `SinglyLinkedList` manipulates its head more efficiently than its tail, we align the `Stack` top with the head of the `SinglyLinkedList` (see Figure 10.3).

The `add` method simply performs an `addFirst`, and the `remove` operation performs `removeFirst`. Since we have implemented the list's `remove` operation so that it returns the value removed, the value can be passed along through the stack's `remove` method.



StackList

```
public void add(E value)
// post: adds an element to stack;
//       will be next element popped if no intervening push
{
    data.addFirst(value);
}

public E remove()
// pre: stack is not empty
```

[1] Again, the class `java.util.Stack` has an `empty` method that is analogous to our `isEmpty` method. We prefer to use `isEmpty` for consistency.

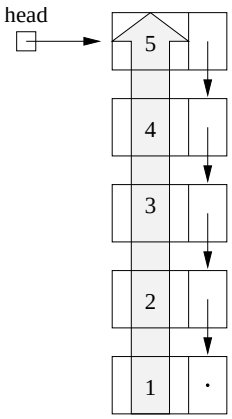


Figure 10.3 A stack implemented using a singly-linked list. The arrow demonstrates the direction of stack growth.

```
// post: removes and returns the top element from stack
{
    return data.removeFirst();
}
```

The remaining methods are simply the obvious wrappers for similar methods from the `SinglyLinkedList`. Because the stack operations are trivial representations of the linked-list operations, their complexities are all the same as the corresponding operations found in linked lists—each takes constant time.

It should be noted that any structure implementing the `List` interface would be sufficient for implementing a `Stack`. The distinction, however, between the various lists we have seen presented here has focused on providing quick access to the tail of the list. Since stacks do not require access to the tail, the alternative implementations of list structure do not provide any benefit. Thus we use the `SinglyLinkedList` implementation.

10.1.4 Comparisons

Clearly, stacks are easily implemented using the structures we have seen so far. In fact, the complexities of those operations make it difficult to decide which of the classes is better. How can we make the decision?

Let's consider each a little more carefully. First, in terms of time, both underlying structures provide efficient implementations. In addition, both structures provide (virtually) unlimited extension of the structure. Their difference stems from a difference in approach for expanding the structure. In the case of the vector, the structure is responsible for deciding when the structure is extended.

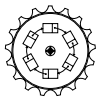
Because the structure grows by doubling, the reallocation of memory occurs increasingly less often, but *when* that reallocation occurs, it takes an increasingly long time. So, while the *amortized* cost of dynamically extending vectors is constant time per element, the incremental cost of extending the vector either is zero (if no extension actually occurs) or is occasionally proportional to the size of the vector. Some applications may not be tolerant of this great variation in the cost of extending the structure. In those cases the `StackList` implementation should be considered.

The constant incremental overhead of expanding the `StackList` structure, however, comes at a price. Since each element of the list structure requires a reference to the list that follows, there is a potentially significant overhead in terms of space. If the items stored in the stack are roughly the size of a reference, the overhead is significant. If, however, the size of a reference is insignificant when compared to the size of the elements stored, the increase in space used may be reasonable.

10.2 Queues

Most of us have participated in queues—at movie theaters, toll booths, or ice cream shops, or while waiting for a communal bathroom in a large family! A *queue*, like a stack, is an ordered collection of elements with tightly controlled access to the structure. Unlike a stack, however, *the first item in is the first item out*. We call it a *FIFO* ("fie-foe") structure. FIFO's are useful because they maintain the order of the data that run through them.

The primary operations of queues are to *enqueue* and *dequeue* elements. Again, to support the `Linear` interface, we supply the `add` and `remove` methods as alternatives. Elements are added at the *tail* of the structure, where they then pass through the structure, eventually reaching the *head* where they are removed. The interface provides a number of other features we have seen already:

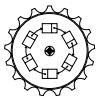


Queue

```
public interface Queue<E> extends Linear<E>
{
    public void add(E value);
    // post: the value is added to the tail of the structure

    public void enqueue(E value);
    // post: the value is added to the tail of the structure

    public E remove();
    // pre: the queue is not empty
    // post: the head of the queue is removed and returned

    public E dequeue();
    // pre: the queue is not empty
    // post: the head of the queue is removed and returned

    public E getFirst();
```

```
    // pre: the queue is not empty
    // post: the element at the head of the queue is returned

    public E get();
    // pre: the queue is not empty
    // post: the element at the head of the queue is returned

    public E peek();
    // pre: the queue is not empty
    // post: the element at the head of the queue is returned

    public boolean empty();
    // post: returns true if and only if the queue is empty

    public int size();
    // post: returns the number of elements in the queue
}
```

As with the `Stack` definition, the `Queue` interface describes necessary characteristics of a class, but not the code to implement it. We use, instead, the `AbstractQueue` class to provide any code that might be of general use in implementing queues. Here, for example, we provide the various traditional aliases for the standard operations `add` and `remove` required by the `Linear` interface:



AbstractQueue

```
public abstract class AbstractQueue<E>
    extends AbstractLinear<E> implements Queue<E>
{
    public void enqueue(E item)
    // post: the value is added to the tail of the structure
    {
        add(item);
    }

    public E dequeue()
    // pre: the queue is not empty
    // post: the head of the queue is removed and returned
    {
        return remove();
    }

    public E getFirst()
    // pre: the queue is not empty
    // post: the element at the head of the queue is returned
    {
        return get();
    }

    public E peek()
    // pre: the queue is not empty
    // post: the element at the head of the queue is returned
```

```
        {
            return get();
        }
    }
```

We will use this abstract class as the basis for the various implementations of queues that we see in this chapter.

10.2.1 Example: Solving a Coin Puzzle

As an example of an application of queues, we consider an interesting coin puzzle (see Figure 10.4). A dime, penny, nickel, and quarter are arranged in decreasing size in each of the leftmost four squares of a five-square board. The object is to reverse the order of the coins and to leave them in the rightmost four slots, in the least number of moves. In each move a single coin moves to the left or right. Coins may be stacked, but only the top coin is allowed to move. When a coin moves, it may not land off the board or on a smaller coin. A typical intermediate legal position is shown in the middle of Figure 10.4. From this point, the nickel may move right and the dime may move in either direction.

We begin by defining a "board state." This object keeps track of the positions of each of the four coins, as well as the series of board states that lead to this state from the start. Its implementation is an interesting problem (see Problem 1.14). We outline its interface here:

CoinPuzzle

```
class State
{
    public static final int DIME=0;    // coins
    public static final int PENNY=1;
    public static final int NICKEL=2;
    public static final int QUARTER=3;
    public static final int LEFT = -1; // directions
    public static final int RIGHT = 1;

    public State()
    // post: construct initial layout of coins

    public State(State prior)
    // pre: prior is a non-null state
    // post: constructs a copy of that state to be successor state

    public boolean done()
    // post: returns true if state is the finish state

    public boolean validMove(int coin, int direction)
    // pre: State.DIME <= coin <= State.QUARTER
    //      direction = State.left or State.right
    // post: returns true if coin can be moved in desired direction
```

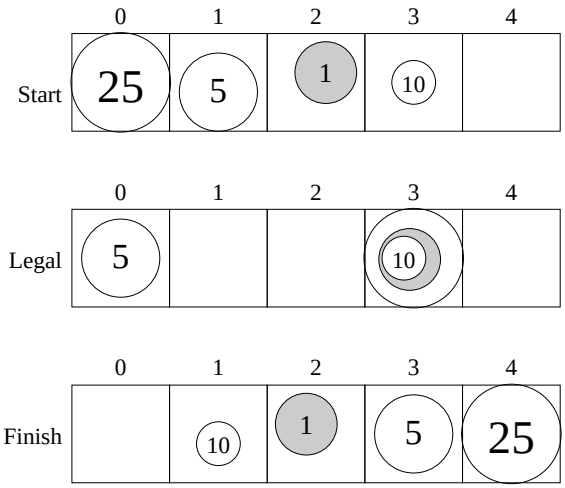


Figure 10.4 A four-coin puzzle. Top and bottom orientations depict the start and finish positions. A typical legal position is shown in the middle.

```
    public State move(int coin, int direction)
    // pre: coin and direction describe valid move
    // post: coin is moved in that direction

    public int printMoves()
    // post: print moves up to and including this point

    public int id()
    // post: construct an integer representing state
    //       states of coins are equal iff id's are equal
}
```

The parameterless constructor builds a board in the initial state, while the one-parameter constructor generates a new state to follow `prior`. The `done` method checks for a state that is in the final position. Coins (identified by constants such as `State.PENNY`) are moved in different directions (e.g., `State.LEFT`), but only if the move is valid. Once the final state is found, the intermediate states leading to a particular board position are printed with `printMoves`.

Once the `State` class has been defined, it is a fairly simple matter to solve the puzzle:

```
public static void main(String args[])
{
    Queue<State> pool = new QueueList<State>();
    State board = new State();
```

```
        BitSet seen = new BitSet(5*5*5*5);
        pool.add(board);
        while (!pool.isEmpty())
        {
            board = (State)pool.remove();
            if (board.done()) break;
            int moveCode = board.id();
            if (seen.contains(moveCode)) continue;
            seen.add(moveCode);
            for (int coin = State.DIME; coin <= State.QUARTER; coin++)
            {
                if (board.validMove(coin,State.LEFT))
                    pool.add(board.move(coin,State.LEFT));
                if (board.validMove(coin,State.RIGHT))
                    pool.add(board.move(coin,State.RIGHT));
            }
        }
        board.printMoves();
    }
```

We begin by keeping a pool of potentially unvisited board states. This queue initially includes the single starting board position. At each stage of the loop an unvisited board position is checked. If it is not the finish position, the boards generated by the legal moves from the current position are added to the state pool. Processing continues until the final position is encountered. At that point, the intermediate positions are printed out.

Because the pool is a FIFO, each unvisited position near the head of the queue must be at least as close to the initial position as those found near the end. Thus, when the final position is found, the distance from the start position (in moves) is a minimum!

A subtle point in this code is the use of the `id` function and the `BitSet`.[2] The `id` function returns a small integer (between 0 and $5^5 - 1$) that uniquely identifies the state of the board. These integers are kept in the set. If, in the future, a board position with a previously encountered state number is found, the state can safely be ignored. Without this optimization, it becomes difficult to avoid processing previously visited positions hundreds or thousands of times before a solution is found. We will leave it to the reader to find the solution either manually or automatically. (The fastest solution involves 22 moves.)

We now investigate three different implementations of queues, based on `Lists`, `Vectors`, and arrays. Each has its merits and drawbacks, so we consider them carefully.

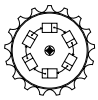
---

[2] `BitSets` are part of the `java.util` package, and we provide public implementations of these and other sets in the `structure` package. They are not discussed formally within this text.

10.2.2 List-Based Queues

Drawing upon our various analogies with real life, it is clear that there are *two* points of interest in a queue: the head and the tail. This two-ended view of queues makes the list a natural structure to support the queue. In this implementation, the head and tail of the queue correspond to the head and tail of the list. The major difference is that a queue is restricted to peeking at and removing values from the head and appending elements to the tail. Lists, on the other hand, allow adding and removing values from both ends.

This discussion leads us to the following protected field within the `QueueList` structure:



`QueueList`

```
protected List<E> data;
```

When we consider the constructor, however, we are forced to consider more carefully which implementation of a list is actually most suitable. While for the stack the `SinglyLinkedList` was ideal, that implementation only provides efficient manipulation of the head of the list. To get fast access to both ends of the list, we are forced to consider either the `DoublyLinkedList` (see Figure 10.5) or the `CircularList`. Either would be time-efficient here, but we choose the more compact `CircularList`. We now consider our constructor:

```
public QueueList()
// post: constructs a new, empty queue
{
    data = new CircularList<E>();
}
```

Notice that, because the `List` structure has unbounded size, the `Queue` structure built atop it is also unbounded. This is a nice feature, since many applications of queues have no easily estimated upper bound.

To add a value to the queue, we simply add an element to the tail of the list. Removing a value from the list provides what we need for the `remove` operation. (Note, once again, that the decision to have `remove` operations symmetric with the `add` operations was the right decision.)

```
public void add(E value)
// post: the value is added to the tail of the structure
{
    data.addLast(value);
}

public E remove()
// pre: the queue is not empty
// post: the head of the queue is removed and returned
{
    return data.removeFirst();
}
```

The needs of the `remove` operation are satisfied by the `removeFirst` operation on a `List`, so very little code needs to be written. The simplicity of these methods demonstrates how *code reuse* can make the implementation of structures less difficult. This is particularly dramatic when we consider the implementation of the size-related methods of the `QueueList` structure:

```
public E get()
// pre: the queue is not empty
// post: the element at the head of the queue is returned
{
    return data.getFirst();
}

public int size()
// post: returns the number of elements in the queue
{
    return data.size();
}

public void clear()
// post: removes all elements from the queue
{
    data.clear();
}

public boolean isEmpty()
// post: returns true iff the queue is empty
{
    return data.isEmpty();
}
```

Because the `QueueList` is a rewrapping of the `List` structure, the complexity of each of the methods corresponds to the complexity of the underlying `List` implementation. For `CircularLists`, most of the operations can be performed in constant time.

We now consider two implementations that do not use dynamically linked structures for their underpinnings. While these structures have drawbacks, they are useful when space is a concern or when the size of the queue can be bounded above.

10.2.3 Vector-Based Queues

The lure of implementing structures through code reuse can lead to performance problems if the underlying structure is not well considered. Thus, while we have advocated hiding the unnecessary details of the implementation within the object, it is important to have a sense of the *complexity* of the object's methods, if the object is the basis for supporting larger structures.

Principle 16 *Understand the complexity of the structures you use.*

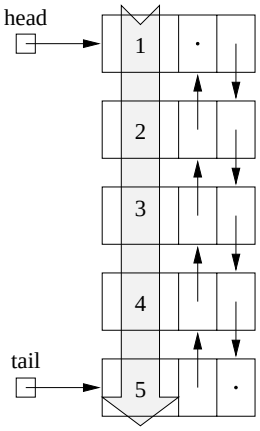


Figure 10.5 A `Queue` implemented using a `List`. The head of the `List` corresponds to the head of the `Queue`.

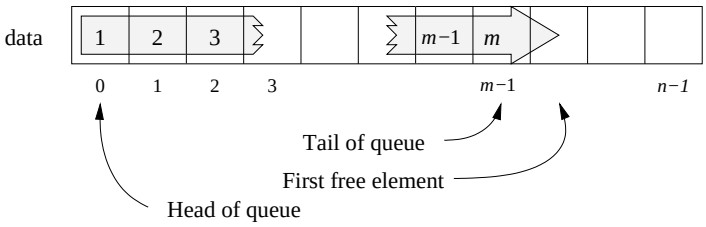


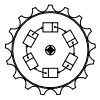
Figure 10.6 A `Queue` implemented atop a `Vector`. As elements are added (enqueued), the `Queue` expands the `Vector`. As elements are removed, the leftmost element is removed, shrinking the `Vector`.

We will return to this issue a number of times. Careful design of data structures sometimes involves careful reconsideration of the performance of structures.

Now, let's consider an implementation of `Queues` using `Vectors`. Recall that a `Vector` is a linear structure whose values can be randomly accessed and modified. We will, again, reconsider the `Vector` as a two-ended structure in our attempt to implement a queue (see Figure 10.6). The head of the queue will be found in the first location of the `Vector`, and the tail of the queue will be found in the last. (The index associated with each element will, essentially, enumerate the order in which they would be removed in the future.)

The constructor creates an empty `QueueVector` by creating an empty `Vector`. When an upper bound is provided, the second `QueueVector` constructor

passes that information along to the `Vector`:



QueueVector

```
protected Vector<E> data;

public QueueVector()
// post: constructs an empty queue
{
    data = new Vector<E>();
}

public QueueVector(int size)
// post: constructs an empty queue of appropriate size
{
    data = new Vector<E>(size);
}
```

Adding an element to the end of the queue is as simple as adding an element to the end of the `Vector`. This is accomplished through the `Vector` method `addElement`.

```
public void add(E value)
// post: the value is added to the tail of the structure
{
    data.add(value);
}
```

As with the `remove` method of the `StackVector` class, remove the first element of the `Vector`, and return the returned value.

```
public E remove()
// pre: the queue is not empty
// post: the head of the queue is removed and returned
{
    return data.remove(0);
}
```

As usual, the remaining methods rewrap similar `Vector` methods in the expected way. Because of the restricted nature of this linear structure, we take care not to publicize any features of the `Vector` class that violate the basic restrictions of a `Linear` interface.

When considering the complexity of these methods, it is important to keep in mind the underlying complexities of the `Vector`. For example, adding a value to the "end" of the `Vector` can be accomplished, on average, in constant time.[3] That method, `add`, is so special in its simplicity that it was distinguished from the general `add(int)` method: the latter operation has time complexity that is expected to be $O(n)$, where n is the length of the `Vector`. The worst-case

[3] It can vary considerably if the `Vector` requires reallocation of memory, but the average time (as we saw in Section 3.5) is still constant time.

behavior, when a value is added at the front of the `Vector`, is $O(n)$. All the existing elements must be moved to the right to make room for the new value.

A similar situation occurs when we consider the removal of values from a `Vector`. Unfortunately, the case we need to use—removing an element from the beginning of a `Vector`—is precisely the worst case. It requires removing the value and then shifting $n - 1$ elements to the left, one slot. That $O(n)$ behavior slows down our `remove` operation, probably by an unacceptable amount. For example, the process of adding n elements to a queue and then dequeuing them takes about $O(n^2)$ time, which may not be acceptable.

Even though the implementation seemed straightforward, we pay a significant price for this code reuse. If we could remove a value from the `Vector` without having to move the other elements, the complexity of these operations could be simplified. We consider that approach in our implementation of the `Queue` interface using arrays of objects.

10.2.4 Array-Based Queues

Fast typists, take note!

If we can determine an upper bound for the size of the queue we need, we can gain some efficiency because we need not worry so much about managing the memory we use. Keyboard hardware, for example, often implements a queue of keystrokes that are to be shipped off to the attached processor. The size of that queue can be easily bounded above by, say, several hundred keystrokes. Notice, by the way, that this does not limit the number of elements that can run through the queue, only the number of values that can be resident within the queue *simultaneously*.

Once the array has been allocated, we simply place enqueued values in successive locations within the array, starting at location 0. The head of the queue—the location containing the oldest element—is initially found at location 0. As elements are removed, we return the value stored at the head of the queue, and move the head toward the right in the array. All the operations must explicitly store and maintain the size of the queue in a counter. This counter should be a nonnegative number less than the length of the array.

One potential problem occurs when a value is removed from the very end of the array. Under normal circumstances, the head of the queue would move to the right, but we now need to have it wrap around. One common solution is to use modular arithmetic. When the head moves too far to the right, its value, `head`, is no longer less than the length of the array, `data.length`. After moving the head to the right (by adding 1 to `head`), we simply compute the remainder when dividing by `data.length`. This always returns a valid index for the array, `data`, and it indexes the value just to the right, in wrap-around fashion (see Figure 10.7). It should be noted, at this point, that remainder computation is reasonable for positive values; if `head` were ever to become negative, one must take care to check to see if a negative remainder might occur. If that appears to be a possibility, simply adding `data.length` to the value before remainder computation fixes the problem in a portable way.

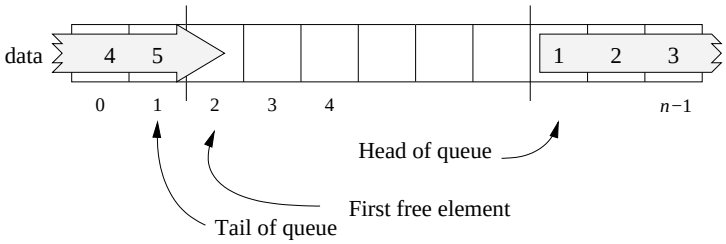
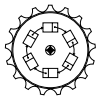


Figure 10.7 A `Queue` implemented atop an array. As elements are added (enqueued), the queue wraps from the back to the front of the array.



`QueueArray`

Our `QueueArray` implementation, then, requires three values: an array of objects that we allocate once and never expand, the index of the head of the queue, and the number of elements currently stored within the queue.

```
protected Object data[]; // an array of the data
protected int head; // next dequeue-able value
protected int count; // current size of queue
```

Given an upper bound, the constructor allocates the array and sets the `head` and `count` variables to 0. The initial value of `head` could, actually, be any value between 0 and `size-1`, but we use 0, for neatness.

42 isn't the ultimate answer, then?

```
public QueueArray(int size)
// post: create a queue capable of holding at most size values
{
    data = new Object[size];
    head = 0;
    count = 0;
}
```

The `add` and `remove` methods appear more complex than their `QueueVector` counterparts, but review of the implementation of the `Vector` class proves their similarity. The `add` method adds a value at the logical end of the queue—the first free slot in the array—in wrap-around fashion. That location is `count` slots to the right of the head of the queue and is computed using modular arithmetic; `remove` is similar, but carefully moves the `head` to the right. The reader should verify that, in a valid `QueueArray`, the value of `head` will never pass the value of `tail`.

```
public void add(E value)
// pre: the queue is not full
// post: the value is added to the tail of the structure
{
    Assert.pre(!isFull(),"Queue is not full.");
```

```
    int tail = (head + count) % data.length;
    data[tail] = value;
    count++;
}

public E remove()
// pre: the queue is not empty
// post: the head of the queue is removed and returned
{
    Assert.pre(!isEmpty(),"The queue is not empty.");
    E value = (E)data[head];
    head = (head + 1) % data.length;
    count--;
    return value;
}
```

The `get` method just provides quick access to the `head` entry of the array.

```
public E get()
// pre: the queue is not empty
// post: the element at the head of the queue is returned
{
    Assert.pre(!isEmpty(),"The queue is not empty.");
    return (E)data[head];
}
```

Because we are directly using arrays, we do not have the luxury of previously constructed size-oriented methods, so we are forced to implement these directly. Again, the cost of an efficient implementation can mean less code reuse—or increased original code and the potential for error.

```
public int size()
// post: returns the number of elements in the queue
{
    return count;
}

public void clear()
// post: removes all elements from the queue
{
    // we could remove all the elements from the queue
    count = 0;
    head = 0;
}

public boolean isFull()
// post: returns true if the queue is at its capacity
{
    return count == data.length;
}
```

```
public boolean isEmpty()
// post: returns true iff the queue is empty
{
    return count == 0;
}
```

Alternative Array Implementations

One aesthetic drawback of the implementation of queues described is that symmetric operations—`add` and `remove`—do not lead to symmetric code. There are several reasons for this, but we might uncover some of the details by reconsidering the implementations.

Two variables, `head` and `count`, are responsible for *encoding* (in an asymmetric way) the information that might be more symmetrically encoded using two variables, say `head` and `tail`. For the sake of argument, suppose that `head` points to the oldest value in the queue, while `tail` points to the oldest value *not* in the queue—the value to the right of the last value in the queue. A simple test to see if the queue is empty is to check to see if `head == tail`. Strangely enough this also appears to be a test to see if the queue is full! Since a queue should never be simultaneously empty and full, this problem must be addressed.

Suppose the array has length l. Then both `head` and `tail` have a range of 0 to $l - 1$. They take on l values apiece. Now consider all queues with `head` values stored beginning at location 0. The `head` is fixed at 0, but `tail` may take on any of the l values between 0 and $l - 1$, inclusive. The queues represented, however, have $l + 1$ potential states: an empty queue, a queue with one value, a queue with two values, up to a queue with l values. Because there are $l + 1$ queues, they cannot be adequately represented by a pair of variables that can support only l different states. There are several solutions to this conflict that we outline momentarily and discuss in problems at the end of the chapter.

1. A boolean variable, `queueEmpty`, could be added to distinguish between the two states where `head` and `tail` are identical. Code written using this technique is clean and symmetric. The disadvantage of this technique is that more code must be written to maintain the extra variable.

2. An array element logically to the left of the head of the queue can be reserved. The queue is full if there are $l - 1$ values within the queue. Since it would not be possible to add a value when the queue is full, the `tail` and `head` variables would never "become equal" through expansion. The only significant difference is the allocation of an extra reserved cell, and a change in the `isFull` method to see if the tail is just to the logical left of the head. The disadvantage of this technique is that it requires the allocation of another array element. When objects are large, this cost may be prohibitively expensive. (Again, in Java, an `Object` is a small reference to the actual memory used, so the cost is fairly insignificant. In other

languages, where instances are stored directly, and not as references, this cost may be prohibitively expensive.)

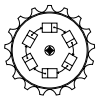
Our actual implementation, of course, provides a third solution to the problem. While we like our solution, data abstraction allows us to hide any changes we might make if we change our minds.

10.3 Example: Solving Mazes

To demonstrate the utility of stacks and queues, we consider the automated solution of a maze. A *maze* is simply a matrix of cells that are adjacent to one another. The user begins in a special *start cell* and seeks a path of adjacent cells that lead to the *finish cell*.

One general approach to solving a maze (or any other search problem) is to consider unvisited cells as potential tasks. Each unvisited cell represents the task of *finding a path from that cell to the finish*. Some cells, of course, may not be reachable from the start position, so they are tasks we seek to avoid. Other cells lead us on trails from the start to finish. Viewed in this manner, we can use the linear structure to help us solve the problem by keeping track of outstanding tasks—unvisited cells adjacent to visited cells.

In the following program, we make use of two abstract classes, `Position` and `Maze`. A `Position` is used to identify a unique location within a maze. Any `Position` can be transformed into another `Position` by asking it for an adjacent position to the north, south, east, or west. Here is the class:



MazeRunner

```
class Position
{
    public Position north()
    // post: returns position above

    public Position south()
    // post: returns position below

    public Position east()
    // post: returns position to right

    public Position west()
    // post: returns position to left

    public boolean equals(Object other)
    // post: returns true iff objects represent same position
}
```

The interface for the `Maze` class reads a maze description from a file and generates the appropriate adjacency of cells. We ignore the implementation that is not important to us at this point (it is available online):

```
class Maze
{
    public Maze(String filename)
    // pre: filename is the name of a maze file. # is a wall.
    //      's' marks the start, 'f' marks the finish.
    // post: reads and constructs maze from file <filename>

    public void visit(Position p)
    // pre: p is a position within the maze
    // post: cell at position p is set to visited

    public boolean isVisited(Position p)
    // pre: p is a position within the maze
    // pos: returns true if the position has been visited

    public Position start()
    // post: returns start position

    public Position finish()
    // post: returns finish position

    public boolean isClear(Position p)
    // post: returns true iff p is a clear location within the maze
}
```

Now, once we have the structures supporting the construction of mazes, we can use a `Linear` structure to organize the search for a solution in a `Maze`:

```
public static void main(String[] arguments)
{
    Maze m = new Maze(arguments[0]); // the maze
    Position goal = m.finish(); // where the finish is
    Position square = null; // the current position
    // a linear structure to manage search
    Linear<Position> todo = new StackList<Position>();

    // begin by priming the queue(stack) w/starting position
    todo.add(m.start());
    while (!todo.isEmpty()) // while we haven't finished exploring
    {
        // take the top position from the stack and check for finish
        square = todo.remove();
        if (m.isVisited(square)) continue; // been here before
        if (square.equals(goal)) {
            System.out.println(m); // print solution
            break;
        }
        // not finished.
        // visit this location, and add neighbors to pool
        m.visit(square);
```

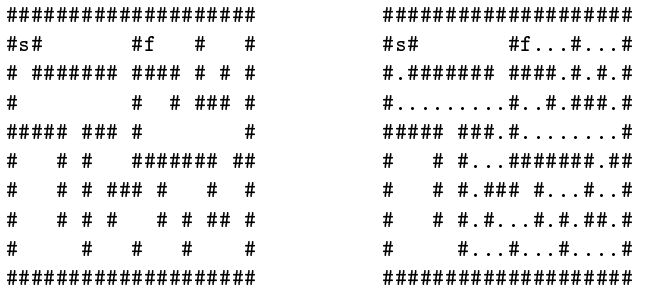


Figure 10.8 A classic maze and its solution found using a stack. Dots indicate locations in the maze visited during the solution process.

```
            if (m.isClear(square.north())) todo.add(square.north());
            if (m.isClear(square.west())) todo.add(square.west());
            if (m.isClear(square.south())) todo.add(square.south());
            if (m.isClear(square.east())) todo.add(square.east());
        }
    }
```

We begin by placing the start position on the stack. If, ultimately, the stack is emptied, a solution is impossible. If the stack is not empty, the top position is removed and considered, if not visited. If an unvisited cell is not the finish, the cell is marked visited, and open neighboring cells are added to the stack.

Notice that since the underlying data structure is a `Stack`, the order in which the neighboring positions are pushed on the stack is the reverse of the order in which they will be considered. The result is that the program prefers to head east before any other direction. This can be seen as it gets distracted by going east at the right border of the maze of Figure 10.8. Because stacks are LIFO structures, the search for a solution prefers to deepen the search rather than investigate alternatives. If a queue was used as the linear structure, the search would expand along a frontier of cells that are equidistant from the start. The solution found, then, would be the most direct route from start to finish, just as in the coin puzzle.

No: Go west young Maze!

10.4 Conclusions

In this chapter we have investigated two important linear structures: the `Stack` and the `Queue`. Each implements `add` and `remove` operations. Traditional implementations of `Stacks` refer to these operations as `push` and `pop`, while traditional `Queue` methods are called `enqueue` and `dequeue`. Since these structures are often used to solve similar problems (e.g., search problems), they share a common `Linear` interface.

There are many different ways to implement each of these linear structures, and we have investigated a number—including implementations using arrays. Because of the trade-offs between speed and versatility, each implementation has its own particular strengths. Still, for many applications where performance is less important, we can select an implementation and use it without great concern because a common interface allows us to freely swap implementations.

We have seen a number of examples that use `Linear` structures to solve complex problems. Since stacks are used to maintain the state of executing methods, we have seen that recursive programs can be converted to iterative programs that maintain an explicit stack. Two explicit search problems—the coin puzzle and the maze—have an obvious relation. Because the coin puzzle searches for a short solution, we use a queue to maintain the pool of goal candidates. For the maze, we chose a stack, but a queue is often just as effective. The coin puzzle can be thought of as a maze whose rules determine the location of the barriers between board positions.

Self Check Problems

Solutions to these problems begin on page 447.

10.1 Is a `Stack` a `Linear`? Is it a `List`? An `AbstractLinear`? A `Queue`?

10.2 Is a `Stack` a `List`? Is a `StackList` a `List`? Is a `StackList` a `Stack`?

10.3 Why might we use generic `Queues` in our code, instead of `QueueLists`?

10.4 Is it possible to construct a new `Queue` directly?

10.5 If you are in a line to wash your car, are you in a queue or a stack?

10.6 Suppose you surf to a page on the Web about gardening. From there, you surf to a page about flowers. From there, you surf to a flower seed distributor. When you push the "go back" button, you return to the flower page. What structure is your history stored in?

10.7 In a breadth-first search, what is special about the first solution found?

10.8 You are in a garden maze and you are optimistically racing to the center. Are you more likely to use a stack-based depth-first search, or a queue-based breadth-first search?

10.9 Why do we use modular arithmetic in the `QueueArray` implementation?

Problems

Solutions to the odd-numbered problems begin on page 474.

10.1 Suppose we push each of the integers $1, 2, \ldots, n$, in order, on a stack, and then perform $m \leq n$ `pop` operations. What is the final state of the stack?

10.2 Suppose we enqueue each of the integers $1, 2, \ldots, n$, in order, into a queue, and then perform $m \leq n$ `dequeue` operations. What is the final state of the queue?

10.3 Suppose you wish to fill a stack with a copy of another, maintaining the order of elements. Using only `Stack` operations, describe how this would be done. How many additional stacks are necessary?

10.4 Suppose you wish to reverse the order of elements of a stack. Using only `Stack` operations, describe how this would be done. Assuming you place the result in the original stack, how many additional stacks are necessary?

10.5 Suppose you wish to copy a queue into another, preserving the order of elements. Using only `Queue` operations, describe how this would be done.

10.6 In the discussion of radix sort (see Section 6.6) `bucketPass` sorted integer values based on a digit of the number. It was important that the sort was stable—that values with similar digits remained in their original relative order. Unfortunately, our implementation used `Vectors`, and to have `bucketPass` work in $O(n)$ time, it was important to add and remove values from the end of the `Vector`. It was also necessary to unload the buckets in reverse order. Is it possible to clean this code up using a `Stack` or a `Queue`? One of these two will allow us to unload the buckets into the data array in increasing order. Does this improved version run as quickly (in terms of big-O)?

10.7 Suppose you wish to reverse the order of elements of a queue. Using only `Queue` operations, describe how this would be done. (Hint: While you can't use a stack, you can use something similar.)

10.8 Over time, the elements 1, 2, and 3 are pushed onto the stack, among others, in that order. What sequence(s) of popping the elements 1, 2 and 3 off the stack are not possible?

10.9 Generalize the solution to Problem 10.8. If elements 1, 2, $3, \ldots, n$ are pushed onto a stack in that order, what sequences of popping the elements off the stack are not permissible?

10.10 Over time, the elements 1, 2, and 3 are added to a queue, in that order. What sequence(s) of removing the elements from the queue is impossible?

10.11 Generalize the solution to Problem 10.10. If elements $1, 2, 3, \ldots, n$ are added to a queue in that order, what sequences of removing the elements are not permissible?

10.12 It is conceivable that one linear structure is more general than another. (a) Is it possible to implement a `Queue` using a `Stack`? What is the complexity of each of the `Queue` operations? (b) Is it possible to implement a `Stack` using a `Queue`? What are the complexities of the various `Stack` methods?

10.13 Describe how we might efficiently implement a `Queue` as a pair of `Stacks`, called a "stack pair." (Hint: Think of one of the stacks as the head of the queue and the other as the tail.)

10.14 The implementation of `QueueLists` makes use of a `CircularList`. Implement `QueueLists` in a manner that is efficient in time and space using `Node` with a head and tail reference.

10.15 Burger Death needs to keep track of orders placed at the drive-up window. Design a data structure to support their ordering system.

10.5 Laboratory: A Stack-Based Language

Objective. To implement a PostScript-based calculator.

Discussion. In this lab we will investigate a small portion of a stack-based language called PostScript. You will probably recognize that PostScript is a file format often used with printers. In fact, the file you send to your printer is a program that instructs your printer to draw the appropriate output. PostScript is stack-based: integral to the language is an operand stack. Each operation that is executed pops its operands from the stack and pushes on a result. There are other notable examples of stack-based languages, including `forth`, a language commonly used by astronomers to program telescopes. If you have an older Hewlett-Packard calculator, it likely uses a stack-based input mechanism to perform calculations.

We will implement a few of the math operators available in PostScript.

To see how PostScript works, you can run a PostScript simulator. (A good simulator for PostScript is the freely available `ghostscript` utility. It is available from `www.gnu.org`.) If you have a simulator handy, you might try the following example inputs. (To exit a PostScript simulator, type `quit`.)

1. The following program computes $1 + 1$:

   ```
   1 1 add pstack
   ```

   Every item you type in is a *token*. Tokens include numbers, booleans, or symbols. Here, we've typed in two numeric tokens, followed by two symbolic tokens. Each number is pushed on the internal stack of operands. When the `add` token is encountered, it causes PostScript to pop off two values and add them together. The result is pushed back on the stack. (Other mathematical operations include `sub`, `mul`, and `div`.) The `pstack` command causes the entire stack to be printed to the console.

2. Provided the stack contains at least one value, the `pop` operator can be used to remove it. Thus, the following computes 2 and prints nothing:

   ```
   1 1 add pop pstack
   ```

3. The following "program" computes $1 + 3 * 4$:

   ```
   1 3 4 mul add pstack
   ```

   The result computed here, 13, is different than what is computed by the following program:

   ```
   1 3 add 4 mul pstack
   ```

   In the latter case the addition is performed first, computing 16.

4. Some operations simply move values about. You can duplicate values—the following squares the number 10.1:

```
10.1 dup mul pstack pop
```

   The `exch` operator to exchange two values, computing $1 - 3$:

```
3 1 exch sub pstack pop
```

5. Comparison operations compute logical values:

```
1 2 eq pstack pop
```

   tests for equality of 1 and 2, and leaves `false` on the stack. The program

```
1 1 eq pstack pop
```

   yields a value of `true`.

6. Symbols are defined using the `def` operation. To define a symbolic value we specify a "quoted" symbol (preceded by a slash) and the value, all followed by the operator `def`:

```
/pi 3.141592653 def
```

   Once we define a symbol, we can use it in computations:

```
/radius 1.6 def
pi radius dup mul mul pstack pop
```

   computes and prints the area of a circle with radius 1.6. After the pop, the stack is empty.

Procedure. Write a program that simulates the behavior of this small subset of PostScript. To help you accomplish this, we've created three classes that you will find useful:

`Token`

`Reader`

- `Token`. An immutable (constant) object that contains a double, boolean, or symbol. Different constructors allow you to construct different `Token` values. The class also provides methods to determine the type and value of a token.
- `Reader`. A class that allows you to read `Tokens` from an input stream. The typical use of a reader is as follows:

```
Reader r = new Reader();
Token t;
while (r.hasNext())
{
    t = (Token)r.next();
    if (t.isSymbol() && // only if symbol:
        t.getSymbol().equals("quit")) break;
    // process token
}
```

This is actually our first use of an `Iterator`. It always returns an `Object` of type `Token`.

- `SymbolTable`. An object that allows you to keep track of `String`–`Token` associations. Here is an example of how to save and recall the value of π:

`SymbolTable`

```
SymbolTable table = new SymbolTable();
// sometime later:
table.add("pi",new Token(3.141592653));
// sometime even later:
if (table.contains("pi"))
{
    Token token = table.get("pi");
    System.out.println(token.getNumber());
}
```

You should familiarize yourself with these classes before you launch into writing your interpreter.

To complete your project, you should implement the PostScript commands `pstack`, `add`, `sub`, `mul`, `div`, `dup`, `exch`, `eq`, `ne`, `def`, `pop`, `quit`. Also implement the nonstandard PostScript command `ptable` that prints the symbol table.

Thought Questions. Consider the following questions as you complete the lab:

1. If we are performing an `eq` operation, is it necessary to assume that the values on the top of the stack are, say, numbers?

2. The `pstack` operation should print the contents of the operand stack without destroying it. What is the most elegant way of doing this? (There are many choices.)

3. PostScript also has a notion of a *procedure*. A procedure is a series of `Tokens` surrounded by braces (e.g., `{ 2 add }`). The `Token` class reads procedures and stores the procedure's `Tokens` in a `List`. The `Reader` class has a constructor that takes a `List` as a parameter and returns a `Reader` that iteratively returns `Tokens` from its list. Can you augment your PostScript interpreter to handle the definition of functions like `area`, below?

```
/pi 3.141592653 def
/area { dup mul pi mul } def
1.6 area
9 area pstack
quit
```

Such a PostScript program defines a new procedure called `area` that computes πr^2 where r is the value found on the top of the stack when the procedure is called. The result of running this code would be

```
254.469004893
8.042477191680002
```

4. How might you implement the `if` operator? The `if` operator takes a boolean and a token (usually a procedure) and executes the token if the boolean is true. This would allow the definition of the absolute value function (given a less than operator, `lt`):

```
/abs { dup 0 lt { -1 mul } if } def
3 abs
-3 abs
eq pstack
```

   The result is `true`.

5. What does the following do?

```
/count { dup 1 ne { dup 1 sub count } if } def
10 count pstack
```

Notes:

10.6 Laboratory: The Web Crawler

Objective. To crawl over web pages in a breadth-first manner.

Discussion. Web crawling devices are a fact of life. These programs automatically venture out on the Web and internalize documents. Their actions are based on the links between pages. The data structures involved in keeping track of the progress of an avid web crawler are quite complex: imagine, for example, how difficult it must be for such a device to keep from chasing loops of references between pages.

In this lab we will build a web crawler that determines the distance from one page to another. If page A references page B, the distance from A to B is 1. Notice that page B may not have any links on it, so the distance from B to A may not be defined.

Here is an approach to determining the distance from page A to arbitrary page B:

- Start a list of pages to consider. This list has two columns: the page, and its distance from A. We can, for example, put page A on this list, and assign it a distance of zero. If we ever see page B on this list, the problem is solved: just print out the distance associated with B.
- Remove the first page on our list: call it page X with distance d from page A. If X has the same URL as B, B must be distance d from A. Otherwise, consider any link off of page X: either it points to a page we've already seen on our list (it has a distance d or less), or it is a new page. If it's a new page we haven't encountered, add it to the end of our list and associate with it a distance $d + 1$—it's a link farther from A than page X. We consider all the links off of page X before considering a new page from our list.

If the list is a FIFO, this process is a *breadth-first* traversal, and the distance associated with B is the shortest distance possible. We can essentially think of the Web as a large maze that we're exploring.

Procedure. Necessary for this lab is a class `HTML`. It defines a reference to a textual (HTML) web page. The constructor takes a URL that identifies which page you're interested in:

HTML

```
HTML page = new HTML("http://www.yahoo.com");
```

Only pages that appear to have valid HTML code can actually be inspected (other types of pages will appear empty). Once the reference is made to the page, you can get its content with the method `content`:

```
System.out.println(page.content());
```

Two methods allow you to get the URL's associated with each link on a page: `hasNext` and `nextURL`. The `hasNext` method returns `true` if there are more links you have not yet considered. The `nextURL` method returns a URL (a `String`) that is pointed to by this page. Here's a typical loop that prints all the links associated with a page:

```
int i = 0;
while (page.hasNext())
{
    System.out.println(i+": "+page.nextURL());
    i++;
}
```

For the sake of speed, the HTML method downloads 10K of information. For simple pages, this covers at least the first visible page on the screen, and it might be argued that the most important links to other pages probably appear within this short start (many crawlers, for example, limit their investigations to the first part of a document). You can change the size of the document considered in the constructor:

```
HTML page = new HTML("http://www.yahoo.com",20*1024);
```

You should probably keep this within a reasonable range, to limit the total impact on memory.

Write a program that will tell us the maximum number of links (found on the first page of a document) that are necessary to get from your home page to any other page within your personal web. You can identify these pages because they all begin with the same prefix. We might think of this as a crude estimate of the "depth" or "diameter" of a site.

Thought Questions. Consider the following questions as you complete the lab:

1. How do your results change when you change the buffer size for the page to 2K? 50K? Under what conditions would a large buffer change cause the diameter of a Web to decrease? Under what conditions would this change cause the diameter of a Web to increase?

Notes: