Chapter 2

Abstract Data Types

The second idea at the core of computer science, along with algorithms, is data. In a modern computer, data consists fundamentally of binary bits, but meaningful data is organized into primitive data types such as integer, real, and boolean and into more complex data structures such as arrays and binary trees. These data types and data structures always come along with associated operations that can be done on the data. For example, the 32-bit *int* data type is defined both by the fact that a value of type *int* consists of 32 binary bits but also by the fact that two *int* values can be added, subtracted, multiplied, compared, and so on. An array is defined both by the fact that it is a sequence of data items of the same basic type, but also by the fact that it is possible to directly access each of the positions in the list based on its numerical index. So the idea of a data type includes a specification of the possible *values* of that type together with the *operations* that can be performed on those values.

An algorithm is an abstract idea, and a program is an implementation of an algorithm. Similarly, it is useful to be able to work with the abstract idea behind a data type or data structure, without getting bogged down in the implementation details. The abstraction in this case is called an "abstract data type." An abstract data type specifies the values of the type, but not how those values are represented as collections of bits, and it specifies operations on those values in terms of their inputs, outputs, and effects rather than as particular algorithms or program code.

Definition 2.1. An **Abstract Data Type**, or **ADT**, consists of (a) a specification of the possible values of the data type and (b) a specification of the operations that can be performed on those values in terms of the operations' inputs, outputs, and effects.

We are all used to dealing with the primitive data types as abstract data types. It is quite likely that you don't know the details of how values of type *double* are represented as sequences of bits. The details are, in fact, rather complicated. However, you know how to work with *double* values by adding them, multiplying them, truncating them to values of type *int*, inputting and outputting them, and so on. To work with values of type *double*, you only need to know how to use these operations and what they do. You don't need to know how they are implemented.

When you create your own new data types—stacks, queues, trees, and even more complicated structures—you are necessarily deeply involved in the data representation and in the implemen-

tation of the operations. However, when you **use** these data types, you don't want to have to worry about the implementation. You want to be able to use them the way you use the built-in primitive types, as abstract data types. This is in conformance with one of the central principles of software engineering: *encapsulation*, also known as information hiding, modularity, and separation of interface from implementation.

2.1 Stacks and Queues

Let's consider two familiar data structures, stacks and queues. Generally, a stack or queue is designed to contain items of some specified type. For example, we can have stacks of integers or queues of strings. The type of the items contained in a stack or queue is called its *base type*.

A possible value of a stack—the data that the stack contains at a given time—is a sequence of items belonging to its base type. A possible value of a stack of integers is a sequence of integers. Exactly the same thing could be said about a queue of integers. So, what's the difference between a stack and a queue? The difference is the set of operations defined for the data type. In particular, the difference is in the meaning of the operation that removes one item from the data structure: A stack's *pop* operation removes the item that has been on the stack for the shortest amount of time, while a queue's *dequeue* operation removes the item that has been on the queue the longest.

A stack is a last in, first out (LIFO) data structure. The item that is in line to be the next one removed from the stack is the one that was added last, that is most recently. The operations on a stack include adding and removing items. Typically, we also specify operations for making a stack empty and for testing whether it is empty. Since we consider a stack to be a stack of items of some specified base type, the base type is also part of the definition. Formally, we get a different stack ADT for each different base type. So, we define the ADT "*stack of BaseType*" where *BaseType* could be integer, string, or any other data type.

Definition 2.2. For any data type, *BaseType*, we define an abstract data type **Stack of BaseType**. The possible values of this type are sequences of items of type *BaseType* (including the sequence of length zero). The operations of the ADT are:

> **push(x):** Adds an item, x, of type *BaseType* to the stack; that is, modifies the value of the stack by appending x to the sequence of items that is already on the stack. No return value.
>
> **pop:** Removes an item from the stack and returns that item. The item removed is the one that was added most recently by the *push* operation. It is an error to apply this operation to an empty stack.
>
> **makeEmpty:** Removes all items from the stack. No return value.
>
> **isEmpty:** Returns a value of type *boolean*, which is true if the stack contains no items and is false if there is at least one item on the stack.

Note that the definition of the ADT does not just list the operations. It specifies their meaning, including their effects on the value of the stack. It also specifies the possible errors that can occur when operations are applied. Errors are a particularly difficult thing to deal with in an abstract way. Ideally, we should know what will happen if the error occurs, but in practice this often depends

on a particular implementation of the ADT. So, in the specification of an ADT, we generally won't specify the effect of using an operation incorrectly. If we are imagining an implementation in Java or C++, we chould specify that the error will throw an exception of some specified type. This would allow users of the ADT to design error-handling strategies for their programs.

We will use this definition of the stack ADT, but it is not the only possible definition. In some variations, for example, the *pop* operation removes an item from the stack but does not return it, and there is a separate *top* operation that tells you the item on the top of the stack without removing it. Some implementations of stacks have a fixed maximum size. These stacks can become "full," and it is an error to try to push an item onto a full stack. Such stacks should have an *isFull* operation to test whether the stack is full. However, this operation would be useless (it would always return the value true) for stack implementations that don't impose a maximum size. On the whole, it seems better to leave it out of the general definition.

The definition of the *queue* ADT is very similar, with operations named *enqueue* and *dequeue* replacing *push* and *pop*:

Definition 2.3. For any data type, *BaseType*, we define an abstract data type **Queue of BaseType**. The possible values of this type are sequences of items of type *BaseType* (including the sequence of length zero). The operations of the ADT are:

> **enqueue(x):** Adds an item, x, of type *BaseType* to the queue; that is, modifies the value of the queue by appending x to the sequence of items that is already on the queue. No return value.
>
> **dequeue:** Removes an item from the queue and returns that item. The item removed is the one that has been in the queue the longest. It is an error to apply this operation to an empty queue.
>
> **makeEmpty:** Removes all items from the queue. No return value.
>
> **isEmpty:** Returns a value of type *boolean*, which is true if the queue contains no items and is false if there is at least one item in the queue.

A queue is a first in, first out (FIFO) data structure. The item that is next in line is the one that arrived first in the queue and that has been waiting in the queue the longest.

2.2 Concrete Data Types

An abstract data type is not something that can be used directly in a program. For that, you need an actual implementation of the ADT, a "concrete" data type that says exactly how the data of the ADT are represented and how the operations are coded. In Java or C++, and ADT can be implemented as a class. Here, for example, is an implementation of the Stack of *int* ADT in C++. Typically in C++, a class is defined in two files, a header (or ".h") file and an implementation (or "cc") file:

```
// The header file, intstack.h
```

```
class IntStack {

   public:
      IntStack();  // A constructor for this class.

      void push(int x);   // The operations of the Stack ADT.
      int pop();
      void makeEmpty();
      bool isEmpty();

   private:
      int data[100];  // Contains the items on the stack.
      int top;        // Number of items on the stack.

};


// The file intstack.cc

#include "intstack.h"

IntStack::IntStack() {
   top = 0;   // A stack is empty when it is first created.
}

void IntStack::push(int x) {
   if (top == 100)     // ERROR; throw an exception.
      throw "Attempt to push onto a full stack";
   data[top] = x;
   top++;
}

int IntStack::pop() {
   if (top == 0)       // ERROR; throw an exception
      throw "Attempt to pop from an empty stack";
   top--;
   return data[top];
}

void IntStack::makeEmpty() {
   top = 0;
}
```

```
bool IntStack::isEmpty() {
   return (top == 0);
}
```

Note that the data for the stack is stored in the private part of the class. This prevents code that uses the class from making any use of the specific representation of the data—that data is simply inaccessible. This ensures that if the data representation is changed, code that uses the class will **not** have to be modified.[1]

Although it would be perfectly adequate in many applications, this is not a particularly good implementation. First of all, it does not implement the ADT exactly as we defined it, because an *IntStack* as defined here can hold no more than 100 items. The possible values of an *IntStack* are sequences of 100 or fewer items, rather than completely general sequences. As for programming style, throwing an exception of type *char**, as is done here in the *push* and *pop* methods, is not considered good style.

If we want a reusable class that can be used in a variety of programs, we should fix these problems. We can eliminate the maximum size limitation by using a dynamic array to store the stack data and increasing the size of the array when the stack becomes full. If we are worried about handling exceptions neatly, we could also define a class specifically to represent errors that can be generated when using a stack. One way to do this is to make this a nested class inside the *IntStack* class:

```
// The header file, intstack.h, VERSION 2

class IntStack {

   public:

      class Error {  // Nested class to represent stack errors.
          // This class contains no data -- the existence
          // of the object is the only information we need
          // about it, since there is only one type of stack error.
      };

      IntStack();   // A constructor for this class.
      ~IntStack();  // Destructor.

      void push(int x);   // The operations of the Stack ADT.
```

[1] Unfortunately, that code **will** have to be recompiled. Even though the code does not refer to the data representation directly, the compiler does make use of the data representation when it compiles the code. For example, the compiler needs to know how much space to leave for a variable of type *IntStack*. This size is encoded into the compiled code. If the data representation changes, the size of an *IntStack* variable will change and the compiled code will have to be different.

```
      int pop();
      void makeEmpty();
      bool isEmpty();

   private:
      int *data;  // Pointer to a dynamic array that contains
                  //    the items on the stack.
      int size;   // Current size of the dynamic array.
      int top;    // Number of items on the stack.

      // Technical point:  disallow copy constructor and
      // assignment operator by making them private.  The
      // defaults would cause problems, and I don't need them.
      IntStack(const IntStack& x);
      IntStack& operator=(const IntStack& x);
};


// The file intstack.cc, VERSION 2

#include "intstack.h"

IntStack::IntStack() {
   data = new int[10]; // Start, arbitrarily, with space for 10 items.
   size = 10;
   top = 0;   // A stack is empty when it is first created.
}

IntStack::~IntStack() {
   delete[] data;  // To avoid a memory leak, we need a destructor
}                  //    to delete the dynamic array.

void IntStack::push(int x) {
   if (top == size) {  // Array is full, so make a new, bigger array.
      int *temp = new int[2*size];
      for (int i = 0; i < size; i++)
         temp[i] = data[i];
      delete[] data;
      data = temp;
      size = 2*size;
   }
   data[top] = x;
```

```
   top++;
}

int IntStack::pop() {
   if (top == 0)
      throw Error(); // Throw an exception of type IntStack::Error.
   top--;
   return data[top];
}

void IntStack::makeEmpty() {
   top = 0;
}

bool IntStack::isEmpty() {
   return (top == 0);
}
```

When we implement an ADT, we should also consider its efficiency. For example, we could apply the type of asymptotic analysis that was covered in Chapter 1 to the operations of the ADT. In our implementation of the *Stack* ADT, the member functions are instance methods in a class. This implies that the stack itself is an implicit input to the operations, even though it is not listed in the parameter lists. We can therefor ask how the run time of the operation depends on the size of the stack.

Most of the instance methods in the *IntStack* are clearly $\Theta(1)$; that is, they run in constant time independent of the size of the stack. The exception is *push*. This method runs in constant time when $top < size$. However, when *top* is equal to *size*, a new array is created and the elements in the old array are copied to the new array. This clearly has a run time that is $\Theta(n)$, where n is the size of the stack. So *push* has a best-case run time that is $\Theta(1)$ and a worst case run time that is $\Theta(n)$.

2.3 The Standard Template Library

For any given type *BaseType*, there is a *Stack of BaseType* ADT. The code for implementing these various ADT's would be almost identical to the *IntStack* class. It would only be necessary to change the type *int* to *BaseType* wherever it appears. It seems silly to have to do this very mechanical process by hand. The C++ language provides **templates** as a way of automating the process. It is possible to define a single Stack template that can then be used to create stacks with any given base type. A Stack template is much more reusable than a stack class for a single base type.

In fact, C++ already comes with a variety of templates that implement some common ADT's, including stacks and queues. This collection of templates is called the **Standard Template Library**, or **STL**. The STL stack and queue templates differ significantly from the ADT's that we

defined above, but the essential functionality is there.

To use an STL stack, you should put the line "`#include <stack>`" at the beginning of your source code file. You can then declare stacks of any base type. The template syntax for the type "Stack of *BaseType*" is `stack<BaseType>`. For example, to use a stack of *int*, you can simply declare a variable

```
stack<int>  nums;
```

and for a stack of C++ strings, you would say

```
stack<string>  words;
```

If s is one of these stacks, of any base type, then several operations are defined. First, `s.push(x)`, where the type of x is the base type of the stack, pushes x onto the stack, as you would expect. The operation `s.pop()` removes the top element of the stack. Note that in contrast to our *pop* operation, it does not return a value. The function `s.top()` returns the top item on the stack, but it does not remove this item. So, calling `s.top()` followed by `s.pop()` is equivalent to the *pop* operation as we defined it above.[2] Finally, the function `s.empty()` returns a *bool* value that tells whether the stack is empty. This is what we called *isEmpty* in our ADT. The STL stack does not have an operation equivalent to our *makeEmpty*.

Similarly, to use an STL queue, you need "`#include <queue>`" at the start of your file. You can then declare variables such as "`queue<Customer> theLine;`". The STL queue uses the terms "push" and "pop" rather than "enqueue" and "dequeue," and just as with stacks, *pop* removes an item without returning it. The operation for determining the value of the item at the front of the queue is called *front*. So, if q is a variable of type `queue<BaseType>`, then: `q.push(x)` adds x to the back of the queue, `q.pop()` deletes the next item from the front of the queue, `q.front()` returns the value of the front item without removing it from the queue, and `q.empty()` returns a *bool* value that tells you whether the queue is empty.

A C++ programmer should ordinarily use the STL versions of common data structures rather than re-implementing them. Of course, when you do this, you lose the control of knowing the exact implementation that is used, but you should probably trust the designers of the STL to provide implementations that will be as efficient as possible in a wide variety of applications.

2.4 Lists

Both stacks and queues are essentially lists of items, but they are restricted in that only certain positions in the list are accessible for adding, deleting, and retrieving items. Often, we need to work with lists that don't have these restrictions. By a **list**, we will mean a sequence of items of a given base type, where items can be added, deleted, and retrieved from any position in the list. A list can be implemented as an array, or as a dynamic array to avoid imposing a maximum size. An alternative implementation is a linked list, where the items are stored in nodes that are

[2]Many people, including Bjarne Stroustrup, the original inventor of C++, don't like the STL definition of *pop*.

linked together with pointers. These two implementations have very different characteristics; some operations that are efficient ($\Theta(1)$) for one of the implementations are much less efficient ($\Theta(n)$) for the other.

It is difficult to specify a single list ADT that covers both arrays and linked lists. One problem is the representation of a *position* in the list. Array positions are naturally represented by integer indices; linked list positions are naturally represented by pointers. The Standard Template Library finesses this difficulty by introducing another abstraction: the *iterator*. An iterator is a data type whose values represent positions in a list or other data structure. Iterators greatly increase the generality of the STL, but they are also one of the main reasons behind its reputation for complexity. Here, we will look at a possible list ADT that uses the concept of "position" in a list, without trying for the same level of generality that is achieved in the STL.

Definition 2.4. For any data type, *BaseType*, we define an abstract data type **List of BaseType**. The possible values of this type are sequences of items of type *BaseType* (including the sequence of length zero). We assume that there is an associated type *position*. A value belonging to the *position* type specifies a position in a particular list. There is one position value for each item in the list, plus a position value that represents a position at the end of the list, just after the last item. The operations of the ADT are:

begin(): Returns a *position* at the beginning of the list.

end(): Returns a *position* at the end of the list. Note that there is no item at this position, but adding an item at this position will append an item onto the end of the list.

next(p), where p is a *position*: Returns the position that follows p. It is an error to call this for the end position of the list.

prev(p), where p is a *position*: Returns the position that precedes p. It is an error to call this for the beginning position of the list.

get(p), where p is a *position*: Returns the item of type *BaseType* at position p. It is an error if p is the end of the list.

set(p,x), where p is a *position* and x is of type *BaseType*: Replaces the item at position p with x. It is an error if p is the end of the list.

insert(p,x), where p is a *position* and x is of type *BaseType*: Inserts x into the list at position p. Note that if p is the beginning of the list, x becomes the first item in the list, and if p is the end of the list, then x becomes the last item. This operation increases the length of the list by one.

remove(p), where p is a *position*: Deletes the item at position p. It is an error if p is the end position of the list. This operation decreases the length of the list by one.

find(x), where x is of type *BaseType*: Returns the position of the first occurrence of x in the list. If x does not occur in the list, then the end position of the list is returned.

nth(k), where k is an integer: Returns a value of type position that specifies the k^{th} position in the list. If there is no such position in the list, the end position is returned.

length(): Returns an integer value giving the number of items in the list.

Most of the things that we might want to do with lists can be defined in terms of these operations. Most applications will only use a few of them. There is no implementation of this ADT in which all the operations are efficient, so the choice of implementation should depend on which operations are needed for a given application.

The main choice is between arrays and linked lists. In the case of an array with n elements, a "position" is simply an integer in the range from 0 to n. Note that the operation $nth(k)$ is trivial for this representation, since it simply returns k, as long as k is in the legal range. In a linked list representation, a position can be a pointer to one of the nodes in the list, but there are some subtleties involved that we will look at later

For an array, the *insert*, *remove*, and *find* operations have worst-case run time that is $\Theta(n)$, where n is the size of the list. All the other operations are $\Theta(1)$. Insert and remove are inefficient for arrays because they can involve moving a lot of array elements. When an item is inserted at a given position in an array, all the array elements that follow that position must be moved down one space in the array to make room for the new element. This can mean moving up to n items, if the position is at the beginning of the array. Similarly, deleting an item from a given position means moving all the items that follow that position up one space in the array. The *find* operation might require looking at every item in the array (but this will be true no matter what implementation is used). If an application of lists does not require insertion and deletion, or uses them only rarely, then it is probably most appropriate to use an array implementation.

For linked lists, *insert* and *remove* can be done efficiently, in constant time. However, linked lists do not support "random access." Given an integer k, there is no way to go directly to the k^{th} item in the list. The only way to get there is to start at the beginning of the list and follow k links. This operation has a $\Theta(n)$ run time. In terms of our ADT, this means that the operation $nth(k)$ is not efficient for linked lists. But in applications where random access to list elements is not needed and where *insert* and *delete* are common operations, a linked list implementation of lists might be preferable to an array implementation. The decision is not automatic, though: A linked list uses more memory than an array, since it requires extra space to store pointers, and some constant-time operations on lists, such as *next*, are a little more complicated than the same operations on arrays.

The term "linked list" does not fully define a data structure. There are several variations on the linked list idea. A *simple linked list* consists of a set of nodes. Each node contains one of the items in the list, and it contains a pointer to the next node in the list. The list as a whole is represented as a pointer variable that points to the first node in the list, or is `NULL` if the list is empty.

Unfortunately, a simple linked list is a very poor choice to represent the *List* ADT! For example, the list operation *end* is a $\Theta(n)$ operation for simple linked lists, since the only way to find the end of a simple linked list is to start at the beginning and follow the entire chain of links to the end. Similarly, the *prev* operation is inefficient for a simple linked list because there is no easy way to move from a node in a simple linked list back to the previous node. And as a matter of fact, it's not even clear how to represent a "position" in a linked list—just using a pointer to a node won't quite work, since for a list of length n, there are $n + 1$ positions but only n different pointers to nodes.

There are several modifications that can be made to simple linked lists to fix some of these problems. These are common features that are found in linked lists in many real applications:

Header node: A header node is an extra node at the beginning of the list that does not contain a list item. It contains a pointer to the first item in the list, if there is one. An empty list contains only the header node. The list as a whole is represented by a pointer to the header node, and this value is never `NULL` since the header node always exists. With a header node, a list of n items contains $n+1$ nodes, so it is possible to use pointers to represent $n+1$ different positions in the list.[3]

Tail pointer: A tail pointer is a pointer to the last node in the list. By maintaining a tail pointer, it becomes possible to go directly to the end of the list, instead of traversing the list from the beginning. For an empty list, the tail pointer points to the header node, if one is used.

Doubly linked list: In a doubly-linked list, each node contains two pointers. One points to the next node in the list, and the second one points to the previous node. In a doubly linked list, the *prev* operation becomes efficient, since it only requires following a pointer. On the other hand, maintaining the extra pointers adds a bit to the run time of several other operations, so doubly-linked lists should be avoided in applications where the *prev* operation is not needed.

We can give an efficient linked list implementation of the *List* ADT using lists that have all three of these features. A simple modification that uses singly linked lists instead of doubly linked lists would also be possible. There is one more subtle point: we represent the position of an item in a list by a pointer to the node that *precedes* the node that actually contains the item—even though it might seem more natural to use a pointer to the node that actually *contains* the item. For example, a pointer to the header node represents the position of the first item in the list, which is actually stored in the node that follows the header node. The tail pointer points to the last node in the list, but as a position, it does not represent the position of the last item in the list; it represents a position that is past the end of the list, following all the items.

If we didn't do this, it wouldn't be easy to represent the $n+1$ positions in an n-element list. Furthermore, to do a *remove* operation on a linked list, we need to have a pointer to the node that precedes the one that we want to remove. By representing the "position" of the item that we want to remove by a pointer to the preceding node, we already have our hands on the pointer that we will need when we perform the operation. In a singly linked list there would otherwise be no efficient way to get this pointer. A similar comment applies to the *insert* operator.

Here, then, is a complete C++ implementation of the *List* ADT using doubly linked lists with header nodes and tail pointers:

```
// The header file, intlist.h:

#include <string>

class ListError { // A class to represent errors in list operations.
```

[3]The use of a header node also happens to make *insert* and *delete* a little more efficient, since they no longer have to treat insertion or deletion at the start of the list as a special case

```
   public:
      std::string message;
      ListError(std::string str) {
         message = str;
      }
};


class ListNode;               // Define a type named "position"
typedef ListNode *position;   //     that is a synonym for ListNode*.
                              // The actual ListNode class is defined
                              //     in the .cc file.

class IntList {

   public:

      IntList();   // Constructor for making an empty list.
      ~IntList();  // Destructor, needed to avoid a memory leak.

      position begin();               // The operations of the List ADT
      position end();
      position next(position p);
      position prev(position p);
      int get(position p);
      void set(position p, int x);
      void insert(position p, int x);
      void remove(position p);
      position find(int x);
      position nth(int n);
      int length();

   private:
      position head, tail;  // Head and tail pointers.

      // Technical point:  disallow copy constructor and
      // assignment operator by making them private.  The
      // defaults would cause problems, and I don't need them.
      IntList(const IntList& x);
      IntList& operator=(const IntList& x);
};
```

```
// The implementation file, intlist_2.cc, using doubly-linked lists:

#include <cstdlib>
#include "intlist.h"

struct ListNode {
   int item;
   ListNode *next;
   ListNode *prev;
};

IntList::IntList() {
   head = tail = new ListNode();
   head->next = head->prev = NULL;
}


IntList::~IntList() {
   while (head != NULL) {
      ListNode *save = head;
      head = head->next;
      delete save;
   }
}


position IntList::begin() {
   return head;
}


position IntList::end() {
   return tail;
}


position IntList::next(position p) {
   if (p == tail)
      throw ListError("Attempt to move past the end of a list");
   return p->next;
```

```
}


position IntList::prev(position p) {
   if (p == head)
      throw ListError("Attempt to move past the beginning of a list");
   return p->prev;
}


int IntList::get(position p) {
   if (p == tail)
      throw ListError(
               "Attempt to refer to a location past the end of a list");
   return p->next->item;
}


void IntList::set(position p, int x) {
   if (p == tail)
      throw ListError(
               "Attempt to refer to a location past the end of a list");
   p->next->item = x;
}


void IntList::insert(position p, int x) {
   ListNode *newnode = new ListNode();
   newnode->item = x;
   newnode->next = p->next;
   newnode->prev = p;
   p->next = newnode;
   if (newnode->next != NULL)
      newnode->next->prev = newnode;
   if (p == tail)
      tail = newnode;
}


void IntList::remove(position p) {
   if (p == tail)
      throw ListError(
```

```
                "Attempt to refer to a location past the end of a list");
   ListNode *deletednode = p->next;
   if (deletednode->next != NULL)
      deletednode->next->prev = deletednode->prev;
   p->next = deletednode->next;
   if (deletednode == tail)
      tail = p;
   delete deletednode;
}


position IntList::find(int x) {
   ListNode *runner = head;
   while (runner != tail && runner->next->item != x)
      runner = runner->next;
   return runner;
}


position IntList::nth(int n) {
   ListNode *runner = head;
   for (int i = 0; i < n; i++) {
       if (runner == tail)
          break;
       runner = runner->next;
   }
   return runner;
}

int IntList::length() {
   int n = 0;
   ListNode *runner = head;
   while (runner != tail) {
      n++;
      runner = runner->next;
   }
   return n;
}
```

Exercises

1. Assume that you already have the class *IntList* which implements the *List* ADT. Using this class, write implementation classes for the *Stack* and *Queue* ADT's. Each of your classes should have a private instance variable of type *List* that holds the data for the stack or queue.

2. What is the run-time efficiency of the *length* operator in the *IntList* class? How would you modify the the class so that this operation has constant run time?

3. Write an implementation class for the *List* ADT that uses an array to hold the items in the list. You can use a static array, and put a maximum size on the lists that can be represented. Or you can be more ambitious by using a dynamic array and allowing the list to grow to arbitrary size.

4. The *find* operation for lists uses at most n steps to search a list of n items. We might also ask how many steps this operation uses **on average**. We will assume that we will only search for items that are, in fact, in the list. If all items in the list have an equal probability of being selected, then the average number of steps to find an item is $n/2$. But what if some items are more likely to be chosen than others? It seems like we could improve the average by putting the items that are most likely to be chosen near the front of the list, so that they will be found quickly. In general, though, we have no way of knowing in advance which items are more likely to be selected. One approach to the problem is a sort of dynamically self-optimizing list. The idea is simple: each time you search for an item in the list, move that item to the beginning of the list. Items that are selected most often will tend to spend more of their time near the front of the list, where they will be found quickly. A somewhat less drastic approach is to move the selected item one position towards the front of the list, instead of moving it all the way to the front.

Write a program to implement and test this idea. The program should use the *IntList* class. First, create a list that contains the integers from 1 to 100 in a random order. Choose random integers in this range and search for them in the list—but make some of the integers much more likely to be selected than others. Count the number of steps it takes to do each search, and find the average number of steps over a very large number of trials. (Because you need to count the number of steps, you won't be able to use the *find* operation that is already defined in the *IntList* class; you will have to define your own find function.) A more ambitious experiment could also find the average numerical position of each integer in the list over the course of time. In addition to writing the program, write a short paper discussing the experiment you did and the results that you got.