as a pointer. This has proven to be a dangerous practice on some computers and the programmer is urged to define explicit return types for functions.

1.3 ALGORITHM SPECIFICATION

1.3.1 Introduction

The concept of an algorithm is fundamental to computer science. Algorithms exist for many common problems, and designing efficient algorithms plays a crucial role in developing large-scale computer systems. Therefore, before we proceed further we need to discuss this concept more fully. We begin with a definition.

Definition: An *algorithm* is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

(1) **Input.** There are zero or more quantities that are externally supplied.

(2) **Output.** At least one quantity is produced.

(3) **Definiteness.** Each instruction is clear and unambiguous.

(4) **Finiteness.** If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

(5) **Effectiveness.** Every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in (3); it also must be feasible. □

In computational theory, one distinguishes between an algorithm and a program, the latter of which does not have to satisfy the fourth condition. For example, we can think of an operating system that continues in a *wait* loop until more jobs are entered. Such a program does not terminate unless the system crashes. Since our programs will always terminate, we will use algorithm and program interchangeably in this text.

We can describe an algorithm in many ways. We can use a natural language like English, although, if we select this option, we must make sure that the resulting instructions are definite. Graphic representations called flowcharts are another possibility, but they work well only if the algorithm is small and simple. In this text we will present most of our algorithms in C, occasionally resorting to a combination of English and C for our specifications. Two examples should help to illustrate the process of translating a problem into an algorithm.

Example 1.1 [*Selection sort*]: Suppose we must devise a program that sorts a set of $n \geq 1$ integers. A simple solution is given by the following:

From those integers that are currently unsorted, find the smallest and place it next in the sorted list.

Although this statement adequately describes the sorting problem, it is not an algorithm since it leaves several unanswered questions. For example, it does not tell us where and how the integers are initially stored, or where we should place the result. We assume that the integers are stored in an array, *list*, such that the ith integer is stored in the ith position, *list* [i], $0 \leq i < n$. Program 1.2 is our first attempt at deriving a solution. Notice that it is written partially in C and partially in English.

```
for (i = 0; i < n; i++) {
  Examine list[i] to list[n-1] and suppose that the
  smallest integer is  at list[min];

  Interchange list[i] and list[min];
}
```

Program 1.2: Selection sort algorithm

To turn Program 1.2 into a real C program, two clearly defined subtasks remain: finding the smallest integer and interchanging it with *list* [i]. We can solve the latter problem using either a function (Program 1.3) or a macro.

```
void swap(int *x, int *y)
{/* both parameters are pointers to ints */
   int temp = *x;    /* declares temp as an int and assigns
                        to it the contents of what x points to */
   *x = *y; /* stores what y points to into the location
               where x points */
   *y = temp; /* places the contents of temp in location
                 pointed to by y */
}
```

Program 1.3: Swap function

Using the function, suppose a and b are declared as **int**s. To swap their values one would

say:

```
swap(&a, &b);
```

passing to *swap* the addresses of a and b. The macro version of swap is:

```
#define SWAP(x,y,t) ((t) = (x), (x) = (y), (y) = (t))
```

The function's code is easier to read than that of the macro but the macro works with any data type.

We can solve the first subtask by assuming that the minimum is $list[i]$, checking $list[i]$ with $list[i+1]$, $list[i+2]$, $\cdots$, $list[n-1]$. Whenever we find a smaller number we make it the new minimum. When we reach $list[n-1]$ we are finished. Putting all these observations together gives us *sort* (Program 1.4). Program 1.4 contains a complete program which you may run on your computer. The program uses the *rand* function defined in *math.h* to randomly generate a list of numbers which are then passed into *sort*. At this point, we should ask if this function works correctly.

Theorem 1.1: Function *sort*(*list*,*n*) correctly sorts a set of $n \geq 1$ integers. The result remains in $list[0], \cdots, list[n-1]$ such that $list[0] \leq list[1] \leq \cdots \leq list[n-1]$.

Proof: When the outer **for** loop completes its iteration for $i = q$, we have $list[q] \leq list[r]$, $q < r < n$. Further, on subsequent iterations, $i > q$ and $list[0]$ through $list[q]$ are unchanged. Hence following the last iteration of the outer **for** loop (i.e., $i = n-2$), we have $list[0] \leq list[1] \leq \cdots \leq list[n-1]$. □

Example 1.2 [*Binary search*]: Assume that we have $n \geq 1$ distinct integers that are already sorted and stored in the array *list*. That is, $list[0] \leq list[1] \leq \cdots \leq list[n-1]$. We must figure out if an integer *searchnum* is in this list. If it is we should return an index, i, such that $list[i] = searchnum$. If *searchnum* is not present, we should return -1. Since the list is sorted we may use the following method to search for the value.

Let *left* and *right*, respectively, denote the left and right ends of the list to be searched. Initially, $left = 0$ and $right = n-1$. Let $middle = (left+right)/2$ be the middle position in the list. If we compare $list[middle]$ with *searchnum*, we obtain one of three results:

(1) **searchnum < list[middle].** In this case, if *searchnum* is present, it must be in the positions between 0 and $middle - 1$. Therefore, we set *right* to $middle - 1$.

(2) **searchnum = list[middle].** In this case, we return *middle*.

(3) **searchnum > list[middle].** In this case, if *searchnum* is present, it must be in the positions between $middle + 1$ and $n - 1$. So, we set *left* to $middle + 1$.

```
#include <stdio.h>
#include <math.h>
#define MAX_SIZE 101
#define SWAP(x,y,t) ((t) = (x), (x)= (y), (y) = (t))
void sort(int [],int); /*selection sort */
void main(void)
{
   int i,n;
   int list[MAX_SIZE];
   printf("Enter the number of numbers to generate: ");
   scanf("%d",&n);
   if( n < 1 || n > MAX_SIZE) {
     fprintf(stderr, "Improper value of n\n");
     exit(EXIT_FAILURE);
   }
   for (i = 0; i < n; i++) {/*randomly generate numbers*/
      list[i] = rand() % 1000;
      printf("%d  ",list[i]);
   }
   sort(list,n);
   printf("\n Sorted array:\n ");
   for (i = 0; i < n; i++) /* print out sorted numbers */
      printf("%d  ",list[i]);
   printf("\n");
}
void sort(int list[],int n)
{
   int i, j, min, temp;
   for (i = 0; i < n-1; i++)  {
      min = i;
      for (j = i+1; j < n; j++)
         if (list[j] < list[min])
            min = j;
      SWAP(list[i],list[min],temp);
   }
}
```

Program 1.4: Selection sort

If *searchnum* has not been found and there are still integers to check, we recalculate *middle* and continue the search. Program 1.5 implements this searching strategy. The algorithm contains two subtasks: (1) determining if there are any integers left to check, and (2) comparing *searchnum* to *list*[*middle*].

```
while (there are more integers to check ) {
   middle = (left + right) / 2;
   if (searchnum < list[middle])
      right = middle - 1;
   else if (searchnum == list[middle])
           return middle;
        else left = middle + 1;
}
```

Program 1.5: Searching a sorted list

We can handle the comparisons through either a function or a macro. In either case, we must specify values to signify less than, equal, or greater than. We will use the strategy followed in C's library functions:

- We return a negative number (−1) if the first number is less than the second.
- We return a 0 if the two numbers are equal.
- We return a positive number (1) if the first number is greater than the second.

Although we present both a function (Program 1.6) and a macro, we will use the macro throughout the text since it works with any data type.

```
int compare(int x, int y)
{/* compare x and y, return -1 for less than, 0 for equal,
    1 for greater */
   if (x < y) return -1;
   else if (x == y) return 0;
        else return 1;
}
```

Program 1.6: Comparison of two integers

The macro version is:

```
#define COMPARE(x,y) (((x) < (y)) ? -1: ((x) == (y))? 0: 1)
```

We are now ready to tackle the first subtask: determining if there are any elements left to check. You will recall that our initial algorithm indicated that a comparison could cause us to move either our left or right index. Assuming we keep moving these indices, we will eventually find the element, or the indices will cross, that is, the left index will have a higher value than the right index. Since these indices delineate the search boundaries, once they cross, we have nothing left to check. Putting all this information together gives us *binsearch* (Program 1.7).

```
int binsearch(int list[], int searchnum, int left,
                                         int right)
{/* search list[0] <= list[1] <=  . . .  <= list[n-1] for
 searchnum. Return its position if found. Otherwise
 return -1 */
   int  middle;
   while (left <= right)  {
      middle = (left + right)/2;
      switch (COMPARE(list[middle], searchnum)) {
         case -1: left = middle + 1;
                  break;
         case 0 : return middle;
         case 1 : right = middle - 1;
      }
   }
   return -1;
}
```

Program 1.7: Searching an ordered list

The search strategy just outlined is called *binary search.* □

The previous examples have shown that algorithms are implemented as functions in C. Indeed functions are the primary vehicle used to divide a large program into manageable pieces. They make the program easier to read, and, because the functions can be tested separately, increase the probability that it will run correctly. Often we will declare a function first and provide its definition later. In this way the compiler is made aware that a name refers to a legal function that will be defined later. In C, groups of functions can be compiled separately, thereby establishing libraries containing groups of logically related algorithms.

1.3.2 Recursive Algorithms

Typically, beginning programmers view a function as something that is invoked (called) by another function. It executes its code and then returns control to the calling function. This perspective ignores the fact that functions can call themselves (*direct recursion*) or they may call other functions that invoke the calling function again (*indirect recursion*). These recursive mechanisms are not only extremely powerful, but they also frequently allow us to express an otherwise complex process in very clear terms. It is for these reasons that we introduce recursion here.

Frequently computer science students regard recursion as a mystical technique that is useful for only a few special problems such as computing factorials or Ackermann's function. This is unfortunate because any function that we can write using assignment, **if-else**, and **while** statements can be written recursively. Often this recursive function is easier to understand than its iterative counterpart.

How do we determine when we should express an algorithm recursively? One instance is when the problem itself is defined recursively. Factorials and Fibonacci numbers fit into this category as do binomial coefficients where:

$$\begin{bmatrix} n \\ m \end{bmatrix} = \frac{n!}{m!(n-m)!}$$

can be recursively computed by the formula:

$$\begin{bmatrix} n \\ m \end{bmatrix} = \begin{bmatrix} n-1 \\ m \end{bmatrix} + \begin{bmatrix} n-1 \\ m-1 \end{bmatrix}$$

We would like to use two examples to show you how to develop a recursive algorithm. In the first example, we take the binary search function that we created in Example 1.2 and transform it into a recursive function. In the second example, we recursively generate all possible permutations of a list of characters.

Example 1.3 [*Binary search*]: Program 1.7 gave the iterative version of a binary search. To transform this function into a recursive one, we must (1) establish boundary conditions that terminate the recursive calls, and (2) implement the recursive calls so that each call brings us one step closer to a solution. If we examine Program 1.7 carefully we can see that there are two ways to terminate the search: one signaling a success (*list*[*middle*] = *searchnum*), the other signaling a failure (the left and right indices cross). We do not need to change the code when the function terminates successfully. However, the **while** statement that is used to trigger the unsuccessful search needs to be replaced with an equivalent **if** statement whose **then** clause invokes the function recursively.

Creating recursive calls that move us closer to a solution is also simple since it requires only passing the new *left* or *right* index as a parameter in the next recursive call. Program 1.8 implements the recursive binary search. Notice that although the code has changed, the recursive function call is identical to that of the iterative function. □

```
int binsearch(int list[], int searchnum, int left,
                                          int right)
{/* search list[0] <= list[1] <= ... <= list[n-1] for
    searchnum. Return its position if found. Otherwise
    return -1 */
   int middle;
   if (left <= right) {
      middle = (left + right)/2;
      switch (COMPARE(list[middle], searchnum)) {
         case -1: return
            binsearch(list, searchnum, middle + 1, right);
         case 0 : return middle;
         case 1 : return
            binsearch(list, searchnum, left, middle - 1);
      }
   }
   return -1;
}
```

Program 1.8: Recursive implementation of binary search

Example 1.4 [*Permutations*]: Given a set of $n \geq 1$ elements, print out all possible permutations of this set. For example, if the set is $\{a, b, c\}$, then the set of permutations is $\{(a, b, c), (a, c, b), (b, a, c), (b, c, a), (c, a, b), (c, b, a)\}$. It is easy to see that, given n elements, there are $n!$ permutations. We can obtain a simple algorithm for generating the permutations if we look at the set $\{a, b, c, d\}$. We can construct the set of permutations by printing:

(1) a followed by all permutations of (b, c, d)

(2) b followed by all permutations of (a, c, d)

(3) c followed by all permutations of (a, b, d)

(4) d followed by all permutations of (a, b, c)

The clue to the recursive solution is the phrase "followed by all permutations." It implies that we can solve the problem for a set with n elements if we have an algorithm that works on $n - 1$ elements. These considerations lead to the development of Program 1.9. We assume that *list* is a character array. Notice that it recursively generates permutations until $i = n$. The initial function call is *perm* (*list*, 0, $n - 1$).

Try to simulate Program 1.9 on the three-element set $\{a, b, c\}$. Each recursive call

```
void perm(char *list, int i, int n)
{/* generate all the permutations of list[i] to list[n] */
   int j, temp;
   if (i == n) {
      for (j = 0; j <= n; j++)
         printf("%c", list[j]);
      printf("    ");
   }
   else {
   /* list[i] to list[n] has more than one permutation,
      generate these recursively */
      for (j = i; j <= n; j++) {
         SWAP(list[i],list[j],temp);
         perm(list,i+1,n);
         SWAP(list[i],list[j],temp);
      }
   }
}
```

Program 1.9: Recursive permutation generator

of *perm* produces new local copies of the parameters *list*, *i*, and *n*. The value of *i* will differ from invocation to invocation, but *n* will not. The parameter *list* is an array pointer and its value also will not vary from call to call. □

We will encounter recursion several more times since many of the algorithms that appear in subsequent chapters are recursively defined. This is particularly true of algorithms that operate on lists (Chapter 4) and binary trees (Chapter 5).

EXERCISES

In the last several examples, we showed you how to translate a problem into a program. We have avoided the issues of data abstraction and algorithm design strategies, choosing to focus on developing a function from an English description, or transforming an iterative algorithm into a recursive one. In the exercises that follow, we want you to use the same approach. For each programming problem, try to develop an algorithm, translate it into a function, and show that it works correctly. Your correctness "proof" can employ an analysis of the algorithm or a suitable set of test runs.

1. Consider the two statements:

   (a) Is $n = 2$ the largest value of n for which there exist positive integers x, y, and z such that $x^n + y^n = z^n$ has a solution?

(b) Store 5 divided by zero into x and go to statement 10.

Both fail to satisfy one of the five criteria of an algorithm. Which criterion do they violate?

2. Horner's rule is a strategy for evaluating a polynomial $A(x) =$

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 + a_0$$

at point x_0 using a minimum number of multiplications. This rule is:

$$A(x_0) = (\cdots ((a_n x_0 + a_{n-1}) x_0 + \cdots + a_1) x_0 + a_0)$$

Write a C program to evaluate a polynomial using Horner's rule.

3. Given n Boolean variables $x_1, \cdots, x_n$, we wish to print all possible combinations of truth values they can assume. For instance, if $n = 2$, there are four possibilities: <*true*, *true*>, <*false*, *true*>, <*true*, *false*>, and <*false*, *false*>. Write a C program to do this.

4. Write a C program that prints out the integer values of x, y, z in ascending order.

5. The pigeon hole principle states that if a function f has n distinct inputs but less than n distinct outputs then there are two inputs a and b such that $a \neq b$ and $f(a) = f(b)$. Write a C program to find the values a and b for which the range values are equal.

6. Given n, a positive integer, determine if n is the sum its divisors, that is, if n is the sum of all t such that $1 \leq t < n$ and t divides n.

7. The factorial function $n!$ has value 1 when $n \leq 1$ and value $n*(n-1)!$ when $n > 1$. Write both a recursive and an iterative C function to compute $n!$.

8. The Fibonacci numbers are defined as: $f_0 = 0$, $f_1 = 1$, and $f_i = f_{i-1} + f_{i-2}$ for $i > 1$. Write both a recursive and an iterative C function to compute f_i.

9. Write an iterative function to compute a binomial coefficient, then transform it into an equivalent recursive function.

10. Ackerman's function $A(m, n)$ is defined as:

$$A(m, n) = \begin{cases} n + 1 & \text{, if } m = 0 \\ A(m - 1, 1) & \text{, if } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{, otherwise} \end{cases}$$

This function is studied because it grows very quickly for small values of m and n. Write recursive and iterative versions of this function.

11. [***Towers of Hanoi***] There are three towers and 64 disks of different diameters placed on the first tower. The disks are in order of decreasing diameter as one scans up the tower. Monks were reputedly supposed to move the disk from tower

1 to tower 3 obeying the rules:

(a) Only one disk can be moved at any time.

(b) No disk can be placed on top of a disk with a smaller diameter.

Write a recursive function that prints out the sequence of moves needed to accomplish this task.

12. If S is a set of n elements the power set of S is the set of all possible subsets of S. For example, if $S = \{a, b, c\}$, then *powerset* $(S) = \{\ \{\}, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$. Write a recursive function to compute *powerset(S)*.

1.4 DATA ABSTRACTION

The reader is no doubt familiar with the basic data types of C. These include **char**, **int**, **float**, and **double**. Some of these data types may be modified by the keywords **short**, **long**, and **unsigned**. Ultimately, the real world abstractions we wish to deal with must be represented in terms of these data types. In addition to these basic types, C helps us by providing two mechanisms for grouping data together. These are the array and the structure. *Arrays* are collections of elements of the same basic data type. They are declared implicitly, for example, *int list*[5] defines a five-element array of integers whose legitimate subscripts are in the range $0 \cdots 4$. *Structs* are collections of elements whose data types need not be the same. They are explicitly defined. For example,

```
struct student {
       char lastName;
       int studentId;
       char grade;
       }
```

defines a structure with three fields, two of type character and one of type integer. The structure name is *student*. Details of C structures are provided in Chapter 2.

All programming languages provide at least a minimal set of predefined data types, plus the ability to construct new, or *user-defined types*. It is appropriate to ask the question, "What is a data type?"

Definition: A *data type* is a collection of *objects* and a set of *operations* that act on those objects. □

Whether your program is dealing with predefined data types or user-defined data types, these two aspects must be considered: objects and operations. For example, the data type **int** consists of the objects {0, +1, −1, +2, −2, $\cdots$, INT_MAX, INT_MIN}, where INT_MAX and INT_MIN are the largest and smallest integers that can be represented on your machine. (They are defined in *limits.h*.) The operations on integers are many,

and would certainly include the arithmetic operators +, –, *, /, and %. There is also testing for equality/inequality and the operation that assigns an integer to a variable. In all of these cases, there is the name of the operation, which may be a prefix operator, such as *atoi*, or an infix operator, such as +. Whether an operation is defined in the language or in a library, its name, possible arguments and results must be specified.

In addition to knowing all of the facts about the operations on a data type, we might also want to know about how the objects of the data type are represented. For example on most computers a **char** is represented as a bit string occupying 1 byte of memory, whereas an **int** might occupy 2 or possibly 4 bytes of memory. If 2 eight-bit bytes are used, then *INT_MAX* is $2^{15} - 1 = 32{,}767$.

Knowing the representation of the objects of a data type can be useful and dangerous. By knowing the representation we can often write algorithms that make use of it. However, if we ever want to change the representation of these objects, we also must change the routines that make use of it. It has been observed by many software designers that hiding the representation of objects of a data type from its users is a good design strategy. In that case, the user is constrained to manipulate the objects solely through the functions that are provided. The designer may still alter the representation as long as the new implementations of the operations do not change the user interface. This means that users will not have to recode their algorithms.

Definition: An *abstract data type (ADT)* is a data type that is organized in such a way that the specification of the objects and the specification of the operations on the objects is separated from the representation of the objects and the implementation of the operations. □

Some programming languages provide explicit mechanisms to support the distinction between specification and implementation. For example, Ada has a concept called a *package*, and C++ has a concept called a *class*. Both of these assist the programmer in implementing abstract data types. Although C does not have an explicit mechanism for implementing ADTs, it is still possible and desirable to design your data types using the same notion.

How does the specification of the operations of an ADT differ from the implementation of the operations? The specification consists of the names of every function, the type of its arguments, and the type of its result. There should also be a description of what the function does, but without appealing to internal representation or implementation details. This requirement is quite important, and it implies that an abstract data type is *implementation-independent*. Furthermore, it is possible to classify the functions of a data type into several categories:

(1) **Creator/constructor**: These functions create a new instance of the designated type.

(2) **Transformers**: These functions also create an instance of the designated type, generally by using one or more other instances. The difference between

constructors and transformers will become more clear with some examples.

(3) **Observers/reporters**: These functions provide information about an instance of the type, but they do not change the instance.

Typically, an ADT definition will include at least one function from each of these three categories.

Throughout this text, we will emphasize the distinction between specification and implementation. In order to help us do this, we will typically begin with an ADT definition of the object that we intend to study. This will permit the reader to grasp the essential elements of the object, without having the discussion complicated by the representation of the objects or by the actual implementation of the operations. Once the ADT definition is fully explained we will move on to discussions of representation and implementation. These are quite important in the study of data structures. In order to help us accomplish this goal, we introduce a notation for expressing an ADT.

Example 1.5 [*Abstract data type NaturalNumber*]: As this is the first example of an ADT, we will spend some time explaining the notation. ADT 1.1 contains the ADT definition of *NaturalNumber*.

ADT *NaturalNumber* is

objects: an ordered subrange of the integers starting at zero and ending at the maximum integer (*INT_MAX*) on the computer

functions:

for all $x, y \in$ *NaturalNumber*; *TRUE*, *FALSE* $\in$ *Boolean*
and where +, −, <, and == are the usual integer operations

NaturalNumber Zero( )	::=	0
Boolean IsZero(x)	::=	**if** (x) **return** *FALSE*
		else return *TRUE*
Boolean Equal(x, y)	::=	**if** ($x == y$) **return** *TRUE*
		else return *FALSE*
NaturalNumber Successor(x)	::=	**if** ($x ==$ *INT_MAX*) **return** x
		else return $x + 1$
NaturalNumber Add(x, y)	::=	**if** (($x + y$) <= *INT_MAX*) **return** $x + y$
		else return *INT_MAX*
NaturalNumber Subtract(x, y)	::=	**if** ($x < y$) **return** 0
		else return $x - y$

end *NaturalNumber*

ADT 1.1: Abstract data type *NaturalNumber*

The ADT definition begins with the name of the ADT. There are two main sections in the definition: the objects and the functions. The objects are defined in terms of the integers, but we make no explicit reference to their representation. The function definitions are a bit more complicated. First, the definitions use the symbols x and y to denote two elements of the data type *NaturalNumber*, while *TRUE* and *FALSE* are elements of the data type *Boolean*. In addition, the definition makes use of functions that are defined on the set of integers, namely, plus, minus, equals, and less than. This is an indication that in order to define one data type, we may need to use operations from another data type. For each function, we place the result type to the left of the function name and a definition of the function to the right. The symbols "::=" should be read as "is defined as."

The first function, *Zero*, has no arguments and returns the natural number zero. This is a constructor function. The function *Successor(x)* returns the next natural number in sequence. This is an example of a transformer function. Notice that if there is no next number in sequence, that is, if the value of x is already *INT_MAX*, then we define the action of *Successor* to return *INT_MAX*. Some programmers might prefer that in such a case *Successor* return an error flag. This is also perfectly permissible. Other transformer functions are *Add* and *Subtract*. They might also return an error condition, although here we decided to return an element of the set *NaturalNumber*. □

ADT 1.1 shows you the general form that all ADT definitions will follow. However, in most of our further examples, the function definitions will not be so close to C functions. In fact, the nature of an ADT argues that we avoid implementation details. Therefore, we will usually use a form of structured English to explain the meaning of the functions. Often, there will be a discrepency even between the number of parameters used in the ADT definition of a function and its C implementation. To avoid confusion between the ADT definition of a function and its C implementation, ADT names begin with an upper case letter while C names begin with a lower case letter.

EXERCISES

For each of these exercises, provide a definition of the abstract data type using the form illustrated in ADT 1.1.

1. Add the following operations to the *NaturalNumber* ADT: *Predecessor*, *IsGreater*, *Multiply*, *Divide*.
2. Create an ADT, *Set*. Use the standard mathematics definition and include the following operations: *Create*, *Insert*, *Remove*, *IsIn*, *Union*, *Intersection*, *Difference*.
3. Create an ADT, *Bag*. In mathematics a *bag* is similar to a *set* except that a *bag* may contain duplicate elements. The minimal operations should include: *Create*, *Insert*, *Remove*, and *IsIn*.
4. Create an ADT, *Boolean*. The minimal operations are *And*, *Or*, *Not*, *Xor* (Exclusive or), *Equivalent*, and *Implies*.