CHAPTER

8 Data Abstractions

In this chapter we investigate how data arrangements other than the cell-by-cell organization provided by a computer's main memory can be simulated—a subject known as data structures. The goal is to allow the data's user to access collections of data as abstract tools rather than force the user to think in terms of the computer's main memory organization. Our study will show how the desire to construct such abstract tools leads to the concept of objects and object-oriented programming.

We introduced the concept of data structure in Chapter 6, where we learned that high-level programming languages provide techniques by which programmers can express algorithms as though the data being manipulated were stored in ways other than the cell-by-cell arrangement provided by a computer's main memory. We also learned that the data structures supported by a programming language are known as primitive structures. In this chapter we will explore techniques by which data structures other than a language's primitive structures can be constructed and manipulated—a study that will lead us from traditional data structures to the object-oriented paradigm. An underlying theme throughout this progression is the construction of abstract tools.

8.1 Basic Data Structures

We begin our study by introducing some basic data structures that will serve as examples in future sections.

Arrays and Aggregates

In Section 6.2, we learned about the data structures known as arrays and aggregate types. Recall that an **array** is a "rectangular" block of data whose entries are of the same type. The simplest form of array is the one-dimensional array, a single row of elements with each position identified by an index. A one-dimensional array with 26 elements could be used to store the number of times each alphabet letter occurs in a page of text, for example. A two-dimensional array consists of multiple rows and columns in which positions are identified by pairs of indices—the first index identifies the row associated with the position, the second index identifies the column. An example would be a rectangular array of numbers representing the monthly sales made by members of a sales force—the entries across each row representing the monthly sales made by a particular member and the entries down each column representing the sales by each member for a particular month. Thus, the entry in the third row and first column would represent the sales made by the third salesperson in January.

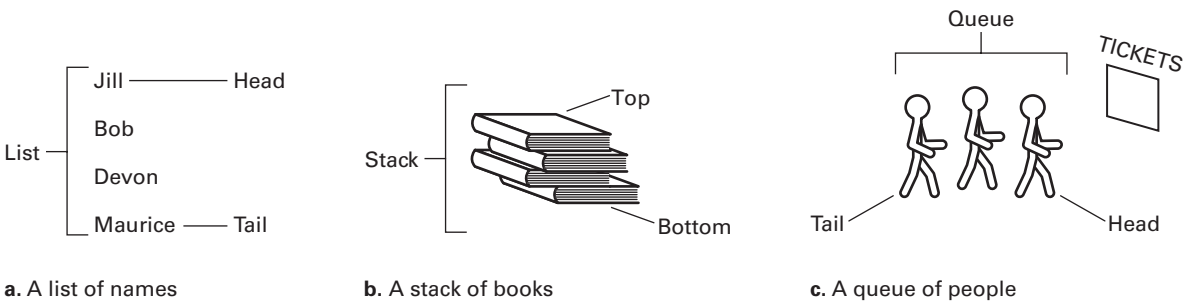
In contrast to an array, recall that an **aggregate type** is a block of data items that might be of different types and sizes. The items within the block are usually called **fields.** An example of an aggregate type would be the block of data relating to a single employee, the fields of which might be the employee's name (an array of type character), age (of type integer), and skill rating (of type float). Fields in an aggregate type are usually accessed by field name, rather than by a numerical index number.

Lists, Stacks, and Queues

Another basic data structure is a **list,** which is a collection whose entries are arranged sequentially (Figure 8.1a). The beginning of a list is called the **head** of the list. The other end of a list is called the **tail.**

Almost any collection of data can be envisioned as a list. For example, text can be envisioned as a list of symbols, a two-dimensional array can be envisioned as a list of rows, and music recorded on a CD can be envisioned as a list of sounds. More traditional examples include guest lists, shopping lists, class enrollment lists, and inventory lists. Activities associated with a list vary depending on the situation.

Figure 8.1 Lists, stacks, and queues



a. A list of names **b.** A stack of books **c.** A queue of people

In some cases we may need to remove entries from a list, add new entries to a list, "process" the entries in a list one at a time, change the arrangement of the entries in a list, or perhaps search to see if a particular item is in a list. We will investigate such operations later in this chapter.

By restricting the manner in which the entries of a list are accessed, we obtain two special types of lists known as stacks and queues. A **stack** is a list in which entries are inserted and removed only at the head. An example is a stack of books where physical restrictions dictate that all additions and deletions occur at the top (Figure 8.1b). Following colloquial terminology, the head of a stack is called the **top** of the stack. The tail of a stack is called its **bottom** or **base.** Inserting a new entry at the top of a stack is called **pushing** an entry. Removing an entry from the top of a stack is called **popping** an entry. Note that the last entry placed on a stack will always be the first entry removed—an observation that leads to a stack being known as a **last-in, first-out,** or **LIFO** (pronounced "LIE-foe") structure.

This LIFO characteristic means that a stack is ideal for storing items that must be retrieved in the reverse order from which they were stored, and thus a stack is often used as the underpinning of backtracking activities. (The term **backtracking** refers to the process of backing out of a system in the opposite order from which the system was entered. A classic example is the process of retracing one's steps in order to find one's way out of a forest.) For instance, consider the underlying structure required to support a recursive process. As each new activation is started, the previous activation must be set aside. Moreover, as each activation is completed, the last activation that was set aside must be retrieved. Thus, if the activations are pushed on a stack as they are set aside, then the proper activation will be on the top of the stack each time an activation needs to be retrieved.

A **queue** is a list in which the entries are removed only at the head and new entries are inserted only at the tail. An example is a line, or queue, of people waiting to buy tickets at a theater (Figure 8.1c)—the person at the head of the queue is served while new arrivals step to the rear (or tail) of the queue. We have already met the queue structure in Chapter 3 where we saw that a batch processing operating system stores the jobs waiting to be executed in a queue called the job queue. There we also learned that a queue is a **first-in, first-out,** or **FIFO** (pronounced "FIE-foe") structure, meaning that the entries are removed from a queue in the order in which they were stored.

Queues are often used as the underlying structure of a buffer, introduced in Chapter 1,which is a storage area for the temporary placement of data being

transferred from one location to another. As the items of data arrive at the buffer, they are placed at the tail of the queue. Then, when it comes time to forward items to their final destination, they are forwarded in the order in which they appear at the head of the queue. Thus, items are forwarded in the same order in which they arrived.

Trees

A **tree** is a collection whose entries have a hierarchical organization similar to that of an organization chart of a typical company (Figure 8.2). The president is represented at the top, with lines branching down to the vice presidents, who are followed by regional managers, and so on. To this intuitive definition of a tree structure we impose one additional constraint, which (in terms of an organization chart) is that no individual in the company reports to two different superiors. That is, different branches of the organization do not merge at a lower level. (We have already seen examples of trees in Chapter 6 where they appeared in the form of parse trees.)

Each position in a tree is called a **node** (Figure 8.3). The node at the top is called the **root node** (if we turned the drawing upside down, this node would represent the base or root of the tree). The nodes at the other extreme are called **terminal nodes** (or sometimes **leaf nodes**). We often refer to the number of nodes in the longest path from the root to a leaf as the **depth** of the tree. In other words, the depth of a tree is the number of horizontal layers within it.

At times we refer to tree structures as though each node gives birth to those nodes immediately below it. In this sense, we often speak of a node's ancestors or descendants. We refer to its immediate descendants as its **children** and its immediate ancestor as its **parent.** Moreover, we speak of nodes with the same parent as being **siblings.** A tree in which each parent has no more than two children is called a **binary tree.**

If we select any node in a tree, we find that that node together with the nodes below it also have the structure of a tree. We call these smaller structures **subtrees.** Thus, each child node is the root of a subtree below the child's parent. Each such subtree is called a **branch** from the parent. In a binary tree, we often speak of a node's left branch or right branch in reference to the way the tree is displayed.

Figure 8.2 An example of an organization chart

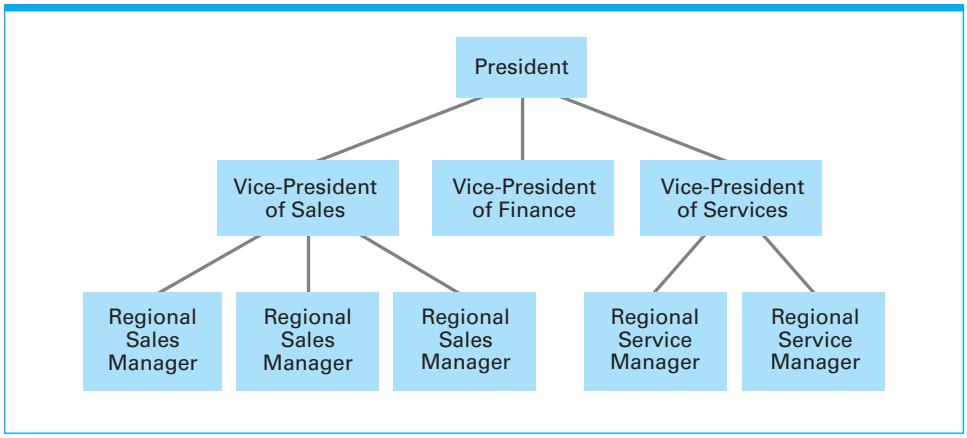
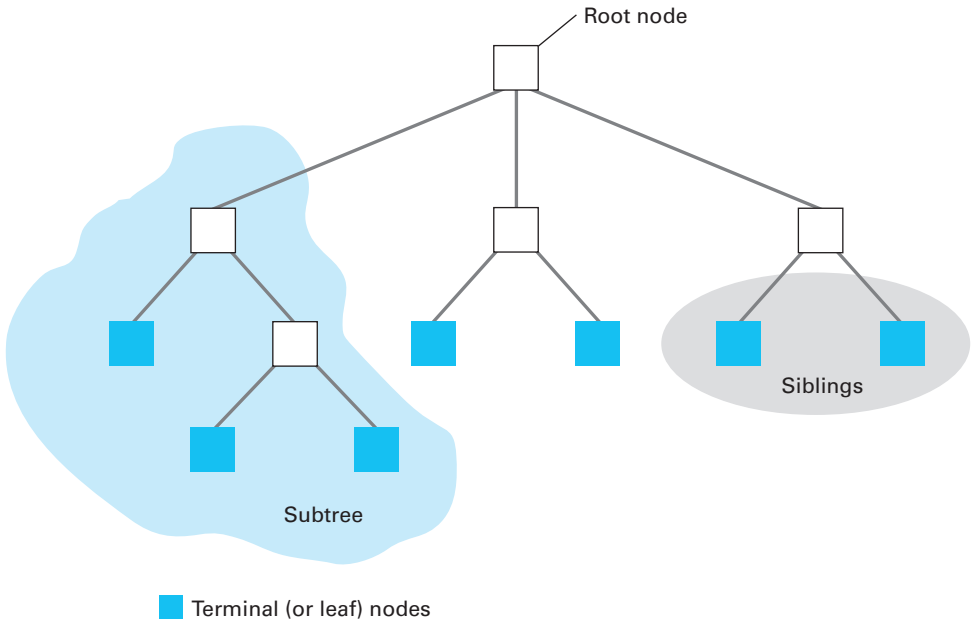


Figure 8.3 Tree terminology



Questions & Exercises

1. Give examples (outside of computer science) of each of the following structures: list, stack, queue, and tree.
2. Summarize the distinction between lists, stacks, and queues.
3. Suppose the letter A is pushed onto an empty stack, followed by the letters B and C, in that order. Then suppose that a letter is popped off the stack and the letters D and E are pushed on. List the letters that would be on the stack in the order they would appear from top to bottom. If a letter is popped off the stack, which letter will be retrieved?
4. Suppose the letter A is placed in an empty queue, followed by the letters B and C, in that order. Then suppose that a letter is removed from the queue and the letters D and E are inserted. List the letters that would be in the queue in the order they would appear from head to tail. If a letter is now removed from the queue, which letter will it be?
5. Suppose a tree has four nodes A, B, C, and D. If A and C are siblings and D's parent is A, which nodes are leaf nodes? Which node is the root?

8.2 Related Concepts

In this section we isolate three topics that are closely associated with the subject of data structures: abstraction, the distinction between static and dynamic structures, and the concept of a pointer.

Abstraction Again

The structures presented in the previous section are often associated with data. However, a computer's main memory is not organized as arrays, lists, stacks, queues, and trees but is instead organized as a sequence of addressable memory cells. Thus, all other structures must be simulated. How this simulation is accomplished is the subject of this chapter. For now we merely point out that organizations such as arrays, lists, stacks, queues, and trees are abstract tools that are created so that users of the data can be shielded from the details of actual data storage and can be allowed to access information as though it were stored in a more convenient form.

The term *user* in this context does not necessarily refer to a human. Instead, the meaning of the word depends on our perspective at the time. If we are thinking in terms of a person using a PC to maintain bowling league records, then the user is a human. In this case, the application software (perhaps a spreadsheet software package) would be responsible for presenting the data in an abstract form convenient to the human—most likely as an array. If we are thinking in terms of a server on the Internet, then the user might be a client. In this case, the server would be responsible for presenting data in an abstract form convenient to the client. If we are thinking in terms of the modular structure of a program, then the user would be any module requiring access to the data. In this case, the module containing the data would be responsible for presenting the data in an abstract form convenient to the other modules. In each of these scenarios, the common thread is that the user has the privilege of accessing data as an abstract tool.

Static Versus Dynamic Structures

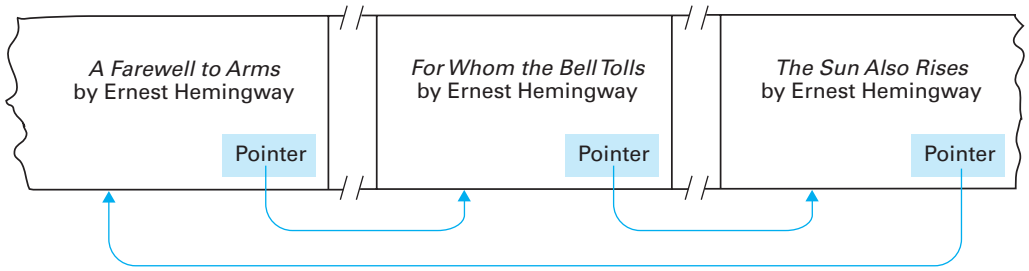
An important distinction in constructing abstract data structures is whether the structure being simulated is static or dynamic, that is, whether the shape or size of the structure changes over time. For example, if the abstract tool is a list of names, it is important to consider whether the list will remain a fixed size throughout its existence or expand and shrink as names are added and deleted.

As a general rule, static structures are more easily managed than dynamic ones. If a structure is static, we need merely to provide a means of accessing the various data items in the structure and perhaps a means of changing the values at designated locations. But, if the structure is dynamic, we must also deal with the problems of adding and deleting entries as well as finding the memory space required by a growing data structure. In the case of a poorly designed structure, adding a single new entry could result in a massive rearrangement of the structure, and excessive growth could dictate that the entire structure be transferred to another memory area where more space is available.

Pointers

Recall that the various cells in a machine's main memory are identified by numeric addresses. Being numeric values, these addresses themselves can be encoded and stored in memory cells. A **pointer** is a storage area that contains such an encoded address. In the case of data structures, pointers are used to record the location where data items are stored. For example, if we must repeatedly move an item of data from one location to another, we might designate a fixed location to serve as a pointer. Then, each time we move the item, we can

Figure 8.4 Novels arranged by title but linked according to authorship



update the pointer to reflect the new address of the data. Later, when we need to access the item of data, we can find it by means of the pointer. Indeed, the pointer will always "point" to the data.

We have already encountered the concept of a pointer in our study of CPUs in Chapter 2. There we found that a register called a program counter is used to hold the address of the next instruction to be executed. Thus, the program counter plays the role of a pointer. In fact, another name for a program counter is **instruction pointer.**

As an example of the application of pointers, suppose we have a list of novels stored in a computer's memory alphabetically by title. Although convenient in many applications, this arrangement makes it difficult to find all the novels by a particular author—they are scattered throughout the list. To solve this problem, we can reserve an additional memory cell within each block of cells representing a novel and use this cell as a pointer to another block representing a book by the same author. In this manner the novels with common authorship can be linked in a loop (Figure 8.4). Once we find one novel by a given author, we can find all the others by following the pointers from one book to another.

Many modern programming languages include pointers as a primitive data type. That is, they allow the declaration, allocation, and manipulation of pointers in ways reminiscent of integers and character strings. Using such a language, a programmer can design elaborate networks of data within a machine's memory where pointers are used to link related items to each other.

Questions & Exercises

1. In what sense are data structures such as arrays, lists, stacks, queues, and trees abstractions?
2. Describe an application that you would expect to involve a static data structure. Then describe an application that you would expect to involve a dynamic data structure.
3. Describe contexts outside of computer science in which the pointer concept occurs.

8.3 Implementing Data Structures

Let us now consider ways in which the data structures discussed in the previous section can be stored in a computer's main memory. As we saw in Chapter 6, these structures are often provided as primitive structures in high-level programming languages. Our goal here is to understand how programs that deal with such structures are translated into machine-language programs that manipulate data stored in main memory.

Storing Arrays

We begin with techniques for storing arrays.

Suppose we want to store a sequence of 24 hourly temperature readings, each of which requires one memory cell of storage space. Moreover, suppose we want to identify these readings by their positions in the sequence. That is, we want to be able to access the first reading or the fifth reading. In short, we want to manipulate the sequence as though it were a one-dimensional array.

We can obtain this goal merely by storing the readings in a sequence of 24 memory cells with consecutive addresses. Then, if the address of the first cell in the sequence is x, the location of any particular temperature reading can be computed by subtracting one from the index of the desired reading and then adding the result to x. In particular, the fourth reading would be located at address $x + (4 - 1)$, as shown in Figure 8.5.

This technique is used by most translators of high-level programming languages to implement one-dimensional arrays. When the translator encounters a declaration statement such as

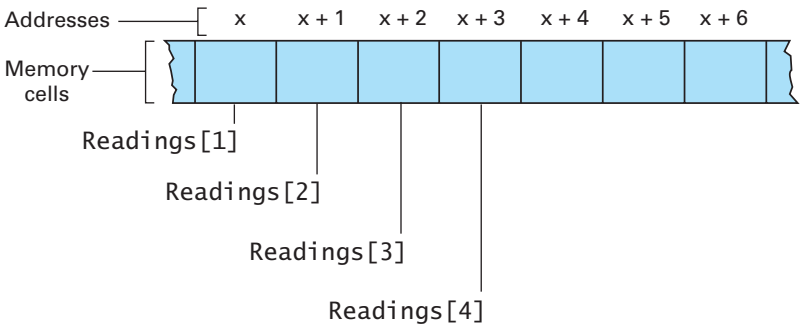
```
int Readings[24];
```

declaring that the term `Readings` is to refer to a one-dimensional array of 24 integer values, the translator arranges for 24 consecutive memory cells to be set aside. Later in the program, if it encounters the assignment statement

```
Readings[4] = 67;
```

requesting that the value 67 be placed in the fourth entry of the array `Readings`, the translator builds the sequence of machine instructions required to place the

Figure 8.5 The array of temperature readings stored in memory starting at address x



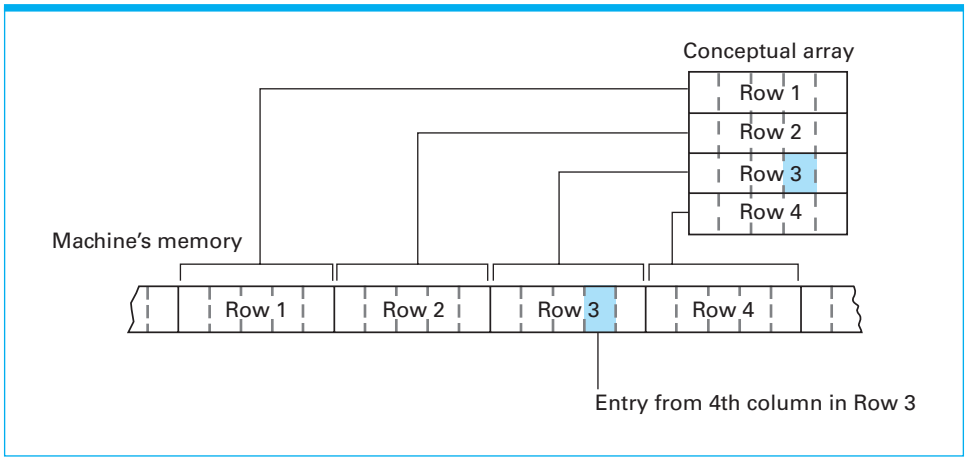
value 67 in the memory cell at address $x + (4 - 1)$, where x is the address of the first cell in the block associated with the array `Readings`. In this manner, the programmer is allowed to write the program as though the temperature readings were actually stored in a one-dimensional array. (Caution: In the languages Python, C, C++, C#, and Java, array indices start at 0 rather than 1, so the fourth reading would be referenced by `Readings[3]`. See question 3 at the end of this section.)

Now suppose we want to record the sales made by a company's sales force during a one-week period. In this case, we might envision the data arranged in a two-dimensional array, where the values across each row indicate the sales made by a particular employee, and the values down a column represent all the sales made during a particular day.

To accommodate this need, we first recognize that the array is static in the sense that its size does not vary as updates are made. We can therefore calculate the amount of storage area needed for the entire array and reserve a block of contiguous memory cells of that size. Next, we store the data in the array row by row. Starting at the first cell of the reserved block, we store the values from the first row of the array into consecutive memory locations; following this, we store the next row, then the next, and so on (Figure 8.6). Such a storage system is said to use **row major order** in contrast to **column major order** in which the array is stored column by column.

With the data stored in this manner, let us consider how we could find the value in the third row and fourth column of the array. Envision that we are at the first location in the reserved block of the machine's memory. Starting at this location, we find the data in the first row of the array followed by the second, then the third, and so on. To get to the third row, we must move beyond both the first and second rows. Since each row contains five entries (one for each day of the week from Monday through Friday), we must move beyond a total of 10 entries to reach the first entry of the third row. From there, we must move beyond another three entries to reach the entry in the fourth column of the row. Altogether, to reach the entry in the third row and fourth column, we must move beyond 13 entries from the beginning of the block.

Figure 8.6 A two-dimensional array with four rows and five columns stored in row major order



Implementing Contiguous Lists

The primitives for constructing and manipulating arrays that are provided in most high-level programming languages are convenient tools for constructing and manipulating contiguous lists. If the entries of the list are all the same primitive data type, then the list is nothing more than a one-dimensional array. A slightly more involved example is a list of ten names, each of which is no longer than eight characters, as discussed in the text. In this case, a programmer could construct the contiguous list as a two-dimensional array of characters with ten rows and eight columns, which would produce the structure represented in Figure 8.6 (assuming that the array is stored in row major order).

Many high-level languages incorporate features that encourage such implementations of lists. For example, suppose the two-dimensional array of characters proposed above was called `MemberList`. Then in addition to the traditional notation in which the expression `MemberList[3,5]` refers to the single character in the third row and fifth column, some languages adopt the expression `MemberList[3]` to refer to the entire third row, which would be the third entry in the list.

The preceding calculation can be generalized to obtain a formula for converting references in terms of row and column positions into actual memory addresses. In particular, if we let c represent the number of columns in an array (which is the number of entries in each row), then the address of the entry in the ith row and jth column will be

$$x + (c \times (i - 1)) + (j - 1)$$

where x is the address of the cell containing the entry in the first row and first column. That is, we must move beyond $i - 1$ rows, each of which contains c entries, to reach the ith row and then $j - 1$ more entries to reach the jth entry in this row. In our prior example $c = 5$, $i = 3$, and $j = 4$, so if the array were stored starting at address x, then the entry in the third row, fourth column would be at address $x + (5 \times (3 - 1)) + (4 - 1) = x + 13$. The expression $(c \times (i - 1)) + (j - 1)$ is sometimes called the **address polynomial.**

Once again, this is the technique used by most translators of high-level programming languages. When faced with the declaration statement

```
int Sales[8, 5];
```

declaring that the term `Sales` is to refer to a two-dimensional array of integer values with 8 rows and 5 columns, the translator arranges for 40 consecutive memory cells to be set aside. Later, if it encounters the assignment statement

```
Sales[3, 4] = 5;
```

requesting that the value 5 be placed in the entry at the third row and fourth column of the array `Sales`, it builds the sequence of machine instructions required to place the value 5 in the memory cell whose address is $x + 5 \times (3 - 1) + (4 - 1)$, where x is the address of the first cell in the block associated with the array `Sales`. In this manner, the programmer is allowed to write the program as though the sales were actually stored in a two-dimensional array.

Storing Aggregates

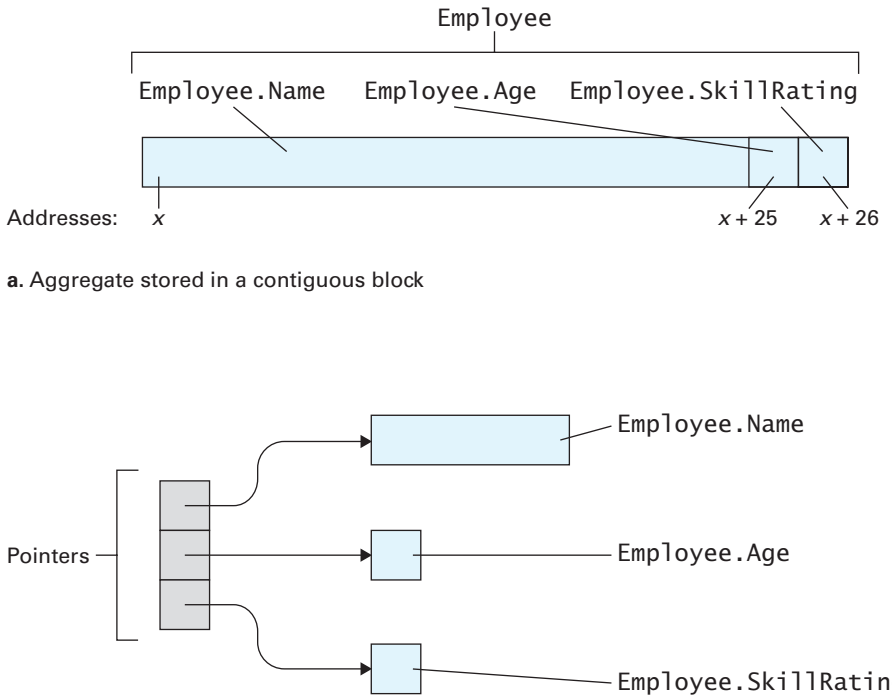
Now suppose we want to store an aggregate called `Employee` consisting of the three fields: `Name` of type character array, `Age` of type integer, and `SkillRating` of type float. If the number of memory cells required by each field is fixed, then we can store the aggregate in a block of contiguous cells. For example, suppose the field `Name` required at most 25 cells, `Age` required only one cell, and `SkillRating` required only one cell. Then, we could set aside a block of 27 contiguous cells, store the name in the first 25 cells, store the age in the 26th cell, and store the skill rating in the last cell (Figure 8.7a).

With this arrangement, it would be easy to access the different fields within the aggregate. A reference to a field can be translated to a memory cell by knowing the address where the aggregate begins, and the offset of the desired field within that aggregate. If the address of the first cell were *x,* then any reference to `Employee.Name` (meaning the `Name` field within the aggregate `Employee`) would translate to the 25 cells starting at address *x* and a reference to `Employee.Age` (the `Age` field within `Employee`) would translate to the cell at address $x + 25$. In particular, if a translator found a statement such as

```
Employee.Age = 22;
```

in a high-level program, then it would merely build the machine language instructions required to place the value 22 in the memory cell whose address is $x + 25$.

Figure 8.7 Storing the aggregate type `Employee`



a. Aggregate stored in a contiguous block

b. Aggregate fields stored in separate locations

An alternative to storing an aggregate in a block of contiguous memory cells is to store each field in a separate location and then link them together by means of pointers. More precisely, if the aggregate contains three fields, then we find a place in memory to store three pointers, each of which points to one of the fields (Figure 8.7b). If these pointers are stored in a block starting at address x, then the first field can be found by following the pointer stored at location x, the second field can be found by following the pointer at location $x + 1$, and so forth.

This arrangement is especially useful in those cases in which the size of the aggregate's fields is dynamic. For instance, by using the pointer system the size of the first field can be increased merely by finding an area in memory to hold the larger field and then adjusting the appropriate pointer to point to the new location. But if the aggregate were stored in a contiguous block, the entire structure would have to be altered.

Storing Lists

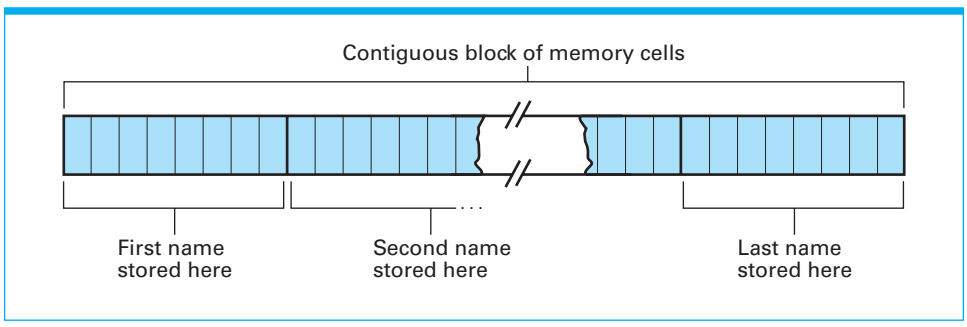
Let us now consider techniques for storing a list of names in a computer's main memory. One strategy is to store the entire list in a single block of memory cells with consecutive addresses. Assuming that each name is no longer than eight letters, we can divide the large block of cells into a collection of subblocks, each containing eight cells. Into each subblock we can store a name by recording its ASCII code using one cell per letter. If the name alone does not fill all the cells in the subblock allocated to it, we can merely fill the remaining cells with the ASCII code for a space. Using this system requires a block of 80 consecutive memory cells to store a list of 10 names.

The storage system just described is summarized in Figure 8.8. The significant point is that the entire list is stored in one large block of memory, with successive entries following each other in contiguous memory cells. Such an organization is referred to as a **contiguous list.**

A contiguous list is a convenient storage structure for implementing static lists, but it has disadvantages in the case of dynamic lists where the deletion and insertion of names can lead to a time-consuming shuffling of entries. In a worst-case scenario, the addition of entries could create the need to move the entire list to a new location to obtain an available block of cells large enough for the expanded list.

These problems can be simplified if we allow the individual entries in a list to be stored in different areas of memory rather than together in one large, contiguous block. To explain, let us reconsider our example of storing a list of names

Figure 8.8 Names stored in memory as a contiguous list



(where each name is no more than eight characters long). This time we store each name in a block of nine contiguous memory cells. The first eight of these cells are used to hold the name itself, and the last cell is used as a pointer to the next name in the list. Following this lead, the list can be scattered among several small nine-cell blocks linked together by pointers. Because of this linkage system, such an organization is called a **linked list.**

To keep track of the beginning of a linked list, we set aside another pointer in which we save the address of the first entry. Since this pointer points to the beginning, or head, of the list, it is called the **head pointer.**

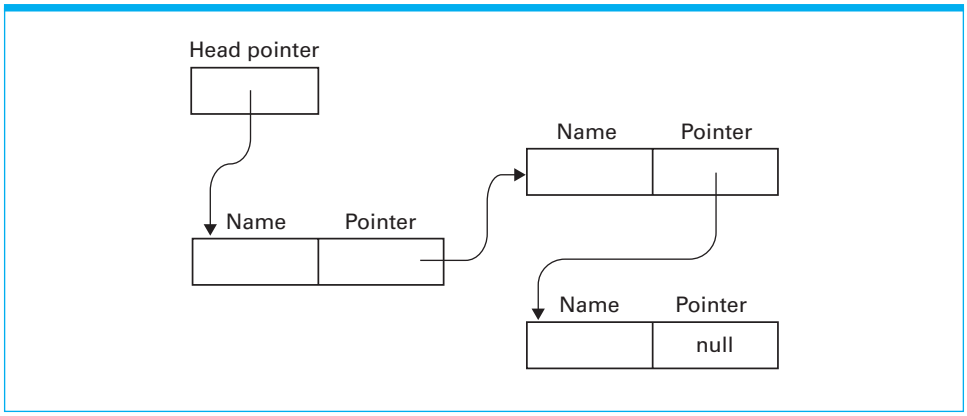
To mark the end of a linked list, we use a **null pointer** (also known as a **NIL pointer** in some languages, or the None object in Python), which is merely a special bit pattern placed in the pointer cell of the last entry to indicate that no further entries are in the list. For example, if we agree never to store a list entry at address 0, the value zero will never appear as a legitimate pointer value and can therefore be used as the null pointer.

The final linked list structure is represented by the diagram in Figure 8.9, in which we depict the scattered blocks of memory used for the list by individual rectangles. Each rectangle is labeled to indicate its composition. Each pointer is represented by an arrow that leads from the pointer itself to the pointer's addressee. Traversing the list involves following the head pointer to find the first entry. From there, we follow the pointers stored with the entries to hop from one entry to the next until the null pointer is reached.

To appreciate the advantages of a linked list over a contiguous one, consider the task of deleting an entry. In a contiguous list this would create a hole, meaning that those entries following the deleted one would have to be moved forward to keep the list contiguous. However, in the case of a linked list, an entry can be deleted by changing a single pointer. This is done by changing the pointer that formerly pointed to the deleted entry so that it points to the entry following the deleted entry (Figure 8.10). From then on, when the list is traversed, the deleted entry is passed by because it no longer is part of the chain.

Inserting a new entry in a linked list is only a little more involved. We first find an unused block of memory cells large enough to hold the new entry and a pointer. Here we store the new entry and fill in the pointer with the address of the entry in the list that should follow the new entry. Finally, we change the pointer associated with the entry that should precede the new entry so that it points to

Figure 8.9 The structure of a linked list



A Problem with Pointers

Just as the use of flowcharts led to tangled algorithm designs (Chapter 5), and the haphazard use of `goto` statements led to poorly designed programs (Chapter 6), undisciplined use of pointers has been found to produce needlessly complex and error-prone data structures. To bring order to this chaos, many programming languages restrict the flexibility of pointers. For example, Java does not allow pointers in their general form. Instead, it allows only a restricted form of pointers called references. One distinction is that a reference cannot be modified by an arithmetic operation. For example, if a Java programmer wanted to advance the reference `Next` to the next entry in a contiguous list, he or she would use a statement equivalent to

```
redirect Next to the next list entry
```

whereas a C programmer would use a statement equivalent to

```
assign Next the value Next + 1
```

Note that the Java statement better reflects the underlying goal. Moreover, to execute the Java statement, there must be another list entry, but if `Next` already pointed to the last entry in the list, the C statement would result in `Next` pointing to something outside the list—a common error for beginning, and even seasoned, C programmers.

the new entry (Figure 8.11). After we make this change, the new entry will be found in the proper place each time the list is traversed.

Storing Stacks and Queues

For storing stacks and queues, an organization similar to a contiguous list is often used. In the case of a stack, a block of memory, large enough to accommodate the stack at its maximum size, is reserved. (Determining the size of this block can often be a critical design decision. If too little room is reserved, the stack will ultimately exceed the allotted storage space; however, if too much room is reserved, memory space will be wasted.) One end of this block is designated as

Figure 8.10 Deleting an entry from a linked list

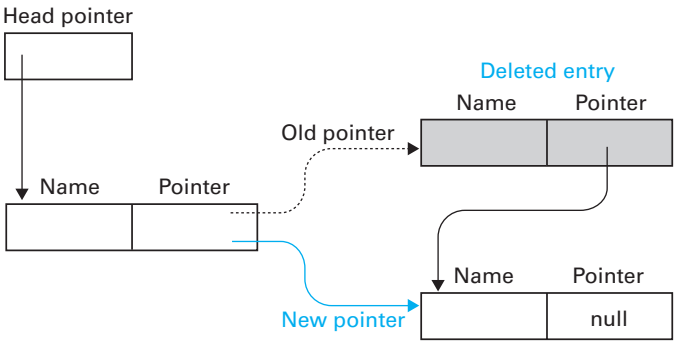
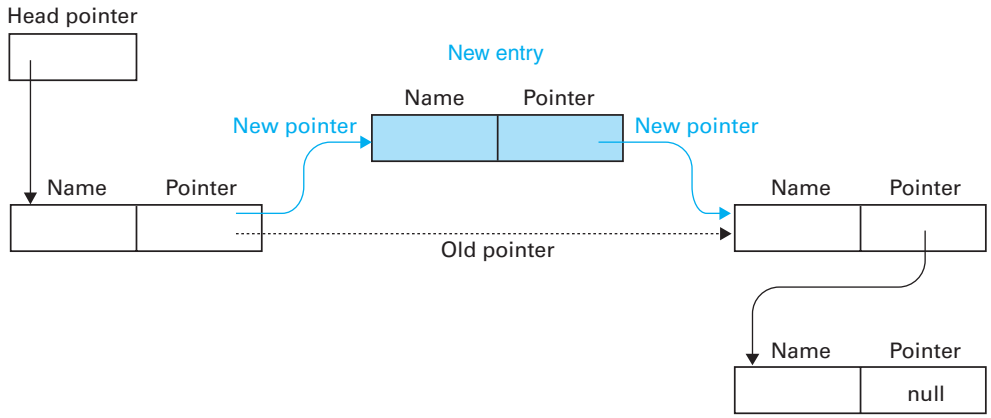


Figure 8.11 Inserting an entry into a linked list



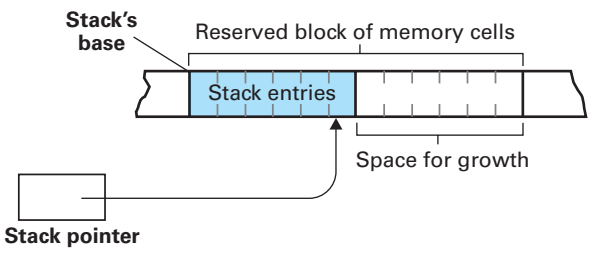
the stack's base. It is here that the first entry to be pushed onto the stack is stored. Then each additional entry is placed next to its predecessor as the stack grows toward the other end of the reserved block.

Observe that as entries are pushed and popped, the location of the top of the stack will move back and forth within the reserved block of memory cells. To keep track of this location, its address is stored in an additional memory cell known as the **stack pointer.** That is, the stack pointer is a pointer to the top of the stack.

The complete system, as illustrated in Figure 8.12, works as follows: To push a new entry on the stack, we first adjust the stack pointer to point to the vacancy just beyond the top of the stack and then place the new entry at this location. To pop an entry from the stack, we read the data pointed to by the stack pointer and then adjust the stack pointer to point to the next entry down on the stack.

The traditional implementation of a queue is similar to that of a stack. Again we reserve a block of contiguous cells in main memory large enough to hold the queue at its projected maximum size. However, in the case of a queue we need to perform operations at both ends of the structure, so we set aside two memory cells to use as pointers instead of only one as we did for a stack. One of these

Figure 8.12 A stack in memory



pointers, called the **head pointer,** keeps track of the head of the queue; the other, called the **tail pointer,** keeps track of the tail. When the queue is empty, both of these pointers point to the same location (Figure 8.13). Each time an entry is inserted into the queue, it is placed in the location pointed to by the tail pointer, and then the tail pointer is adjusted to point to the next unused location. In this manner, the tail pointer is always pointing to the first vacancy at the tail of the queue. Removing an entry from the queue involves extracting the entry pointed to by the head pointer and then adjusting the head pointer to point to the next entry in the queue.

A problem with the storage system as described thus far is that, as entries are inserted and removed, the queue crawls through memory like a glacier (see again Figure 8.13). Thus we need a mechanism for confining the queue to its reserved block of memory. The solution is simple. We let the queue migrate through the block. Then, when the tail of the queue reaches the end of the block, we start inserting additional entries back at the original end of the block, which by this time is vacant. Likewise, when the last entry in the block finally becomes the head of the queue and this entry is removed, the head pointer is adjusted back to the beginning of the block where other entries are, by this time, waiting. In this manner, the queue chases itself around within the block as though the ends of the

Figure 8.13 A queue implementation with head and tail pointers. Note how the queue crawls through memory as entries are inserted and removed

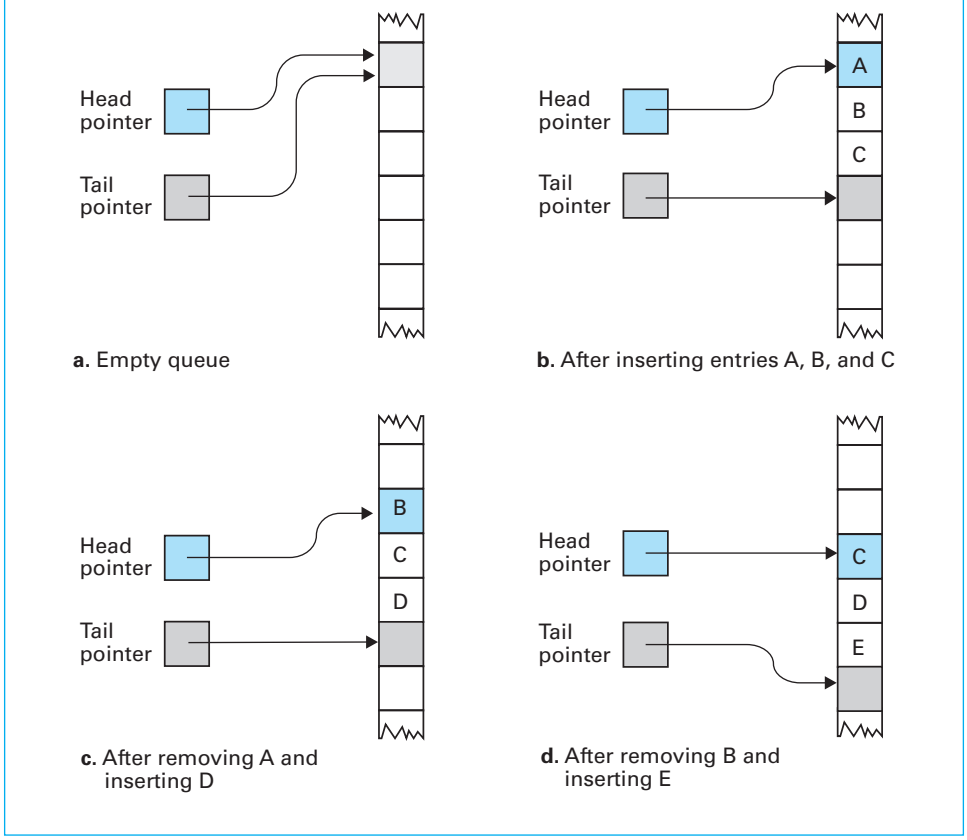
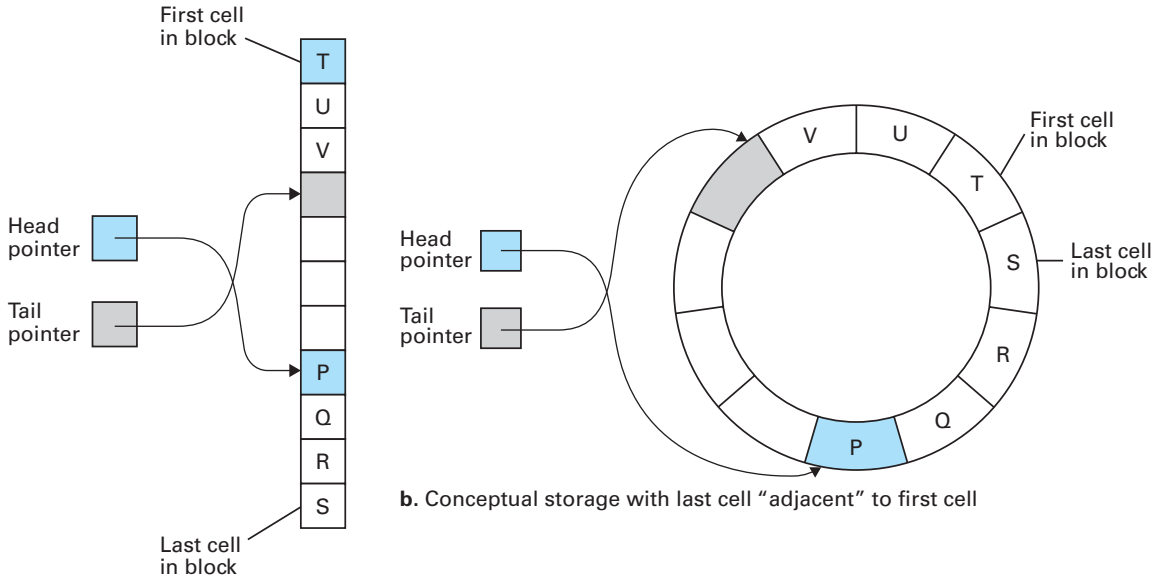


Figure 8.14 A circular queue containing the letters P through V



block were connected to form a loop (Figure 8.14). The result is an implementation called a **circular queue.**

Storing Binary Trees

For the purpose of discussing tree storage techniques, we restrict our attention to binary trees, which we recall are trees in which each node has at most two children. Such trees normally are stored in memory using a linked structure similar to that of linked lists. However, rather than each entry consisting of two components (the data followed by a next-entry pointer), each entry (or node) of the binary tree contains three components: (1) the data, (2) a pointer to the node's first child, and (3) a pointer to the node's second child. Although there is no left or right inside a machine, it is helpful to refer to the first pointer as the **left child pointer** and the other pointer as the **right child pointer** in reference to the way we would draw the tree on paper. Thus each node of the tree is represented by a short, contiguous block of memory cells with the format shown in Figure 8.15.

Figure 8.15 The structure of a node in a binary tree

Storing the tree in memory involves finding available blocks of memory cells to hold the nodes and linking these nodes according to the desired tree structure. Each pointer must be set to point to the left or right child of the pertinent node or assigned the null value if there are no more nodes in that direction of the tree. (This means that a terminal node is characterized by having both of its pointers assigned null.) Finally, we set aside a special memory location, called a **root pointer,** where we store the address of the root node. It is this root pointer that provides initial access to the tree.

An example of this linked storage system is presented in Figure 8.16, where a conceptual binary tree structure is exhibited along with a representation of how that tree might actually appear in a computer's memory. Note that the actual arrangement of the nodes within main memory might be quite different from the conceptual arrangement. However, by following the root pointer, one can find the root node and then trace any path down the tree by following the appropriate pointers from node to node.

An alternative to storing a binary tree as a linked structure is to use a single, contiguous block of memory cells for the entire tree. Using this approach, we store the tree's root node in the first cell of the block. (For simplicity, we assume that each node of the tree requires only one memory cell.) Then we store the left child of the root in the second cell, store the right child of the root in the third cell, and in general, continue to store the left and right children of the node found in cell n in the cells $2n$ and $2n + 1$, respectively. Cells within the block that represent locations not used by the tree are marked with a unique bit pattern that indicates the absence of data. Using this technique, the same tree shown in Figure 8.16 would be stored as shown in Figure 8.17. Note that the system is essentially that of storing the nodes across successively lower

Figure 8.16 The conceptual and actual organization of a binary tree using a linked storage system

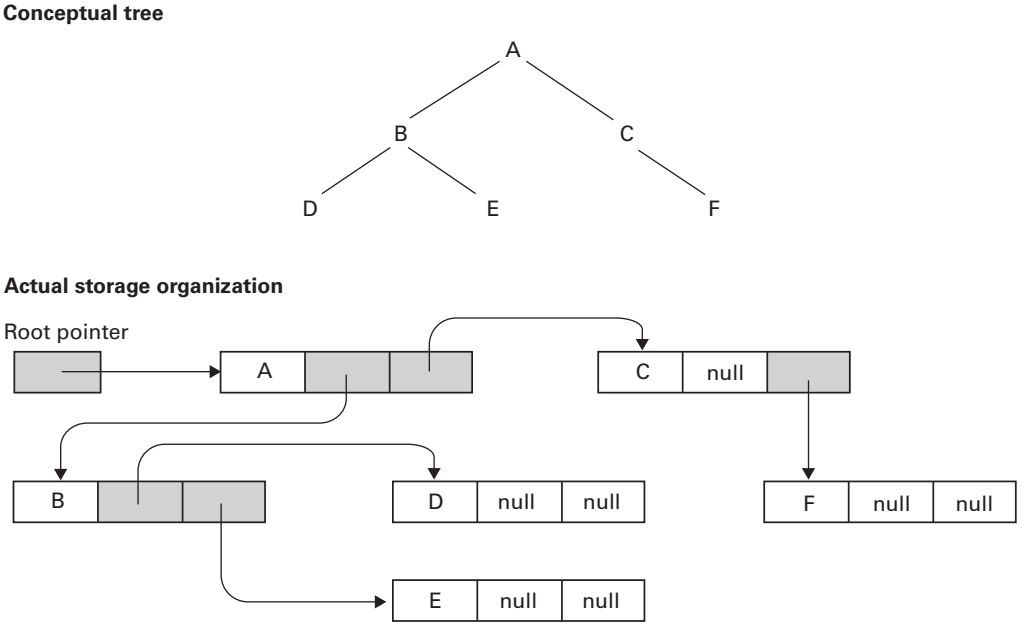
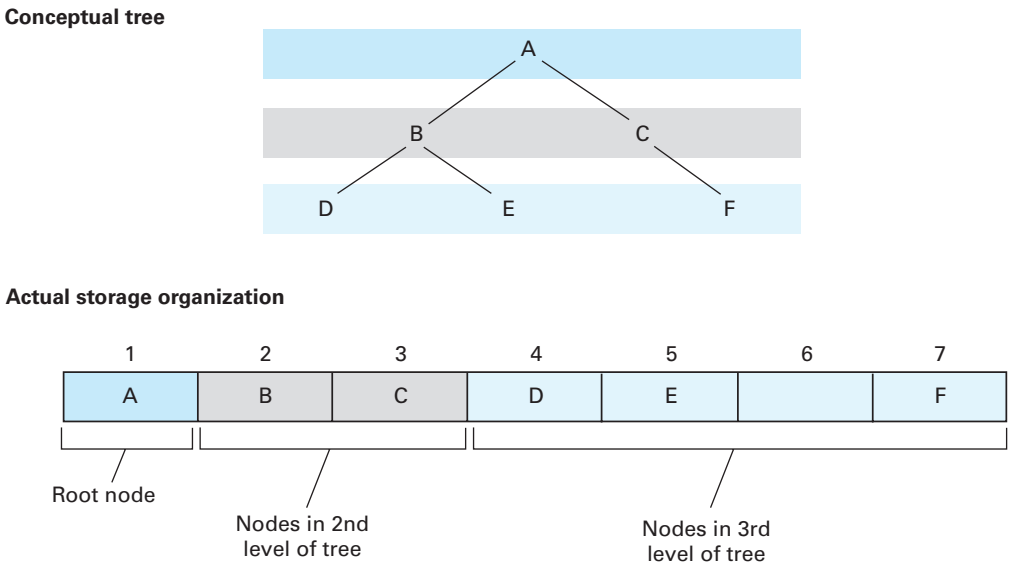


Figure 8.17 A tree stored without pointers



levels of the tree as segments, one after the other. That is, the first entry in the block is the root node, followed by the root's children, followed by the root's grandchildren, and so on.

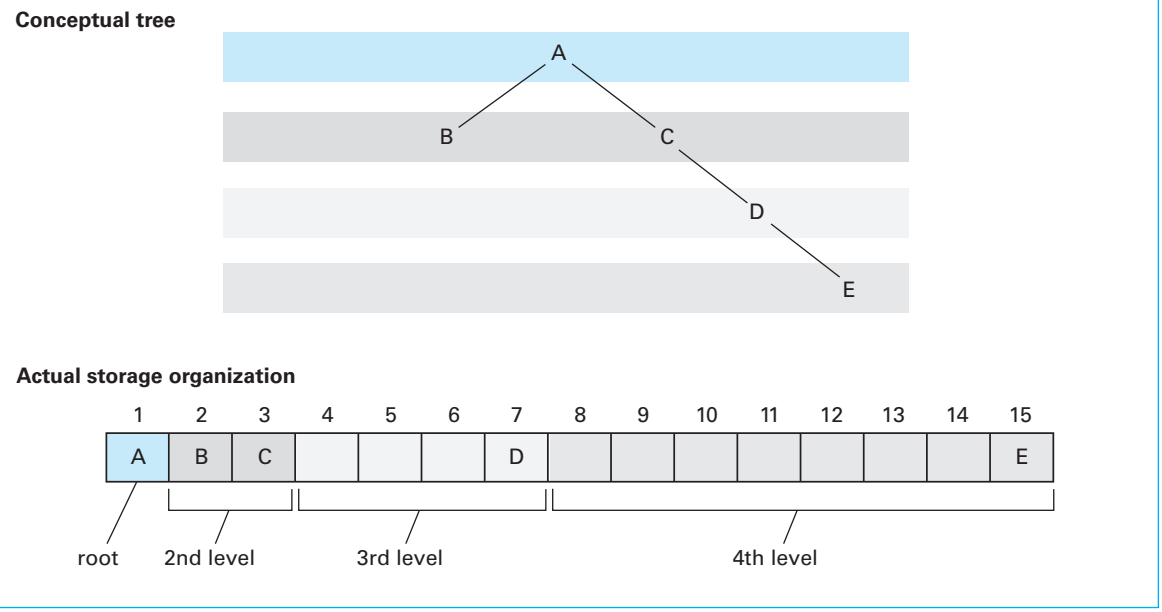
In contrast to the linked structure described earlier, this alternative storage system provides an efficient method for finding the parent or sibling of any node. The location of a node's parent can be found by dividing the node's position in the block by 2 while discarding any remainder (the parent of the node in position 7 would be the node in position 3). The location of a node's sibling can be found by adding 1 to the location of a node in an even-numbered position or subtracting 1 from the location of a node in an odd-numbered position. For example, the sibling of the node in position 4 is the node in position 5, while the sibling of the node in position 3 is the node in position 2. Moreover, this storage system makes efficient use of space when the binary tree is approximately balanced (in the sense that both subtrees below the root node have the same depth) and full (in the sense that it does not have long, thin branches). For trees without these characteristics, though, the system can become quite inefficient, as shown in Figure 8.18.

Manipulating Data Structures

We have seen that the way data structures are actually stored in a computer's memory is not the same as the conceptual structure envisioned by the user. A two-dimensional array is not actually stored as a two-dimensional rectangular block, and a list or a tree might actually consist of small pieces scattered over a large area of memory.

Hence, to allow the user to access the structure as an abstract tool, we must shield the user from the complexities of the actual storage system. This means that instructions given by the user (and stated in terms of the abstract tool) must be converted into steps that are appropriate for the actual storage system. In the

Figure 8.18 A sparse, unbalanced tree shown in its conceptual form and as it would be stored without pointers



case of arrays, we have seen how this can be done by using an address polynomial to convert row and column indices into memory cell addresses. In particular, we have seen how the statement

```
Sales[3, 4] = 5;
```

written by a programmer who is thinking in terms of an abstract array can be converted into steps that perform the correct modifications to main memory. Likewise, we have seen how statements such as

```
Employee.Age = 22;
```

referring to an abstract aggregate type can be translated into appropriate actions depending on how the aggregate is actually stored.

In the case of lists, stacks, queues, and trees, instructions stated in terms of the abstract structure are usually converted into the appropriate actions by means of functions that perform the required task while shielding the user from the details of the underlying storage system. For example, if the function `insert` were provided for inserting new entries into a linked list, then J. W. Brown could be inserted in the list of students enrolled in Physics 208 merely by executing a function call such as

```
insert("Brown, J.W.", Physics208)
```

Note that the function call is stated entirely in terms of the abstract structure—the manner in which the list is actually implemented is hidden.

As a more detailed example, Figure 8.19 presents a function named `printList` for printing a linked list of values. This function assumes that the first entry of the list is pointed to by a field called `Head` within the aggregate called `List`, and that each entry in the list consists of two pieces: a value ("`Value`") and a pointer

Figure 8.19 A function for printing a linked list

```
def PrintList(List):
  CurrentPointer = List.Head
  while (CurrentPointer != None):
    print(CurrentPointer.Value)
    CurrentPointer = CurrentPointer.Next
```

to the next entry ("`Next`"). In the figure, the special Python value `None` is used as the null pointer. Once this function has been developed, it can be used to print a linked list as an abstract tool without being concerned for the steps actually required to print the list. For example, to obtain a printed class list for Economics 301, a user need only perform the function call

```
printList(Economics301ClassList)
```

to obtain the desired results. Moreover, if we should later decide to change the manner in which the list is actually stored, then only the internal actions of the function `printList` must be changed—the user would continue to request the printing of the list with the same function call as before.

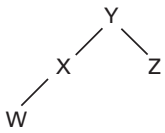
Questions & Exercises

1. Show how the array below would be arranged in main memory when stored in row major order.

5	3	7
4	2	8
1	9	6

2. Give a formula for finding the entry in the *i*th row and *j*th column of a two-dimensional array if it is stored in column major order rather than row major order.
3. In the Python, C, C++, Java, and C# programming languages, indices of arrays start at 0 rather than at 1. Thus the entry in the first row, fourth column of an array named `Array` is referenced by `Array[0][3]`. In this case, what address polynomial is used by the translator to convert references of the form `Array[i][j]` into memory addresses?
4. What condition indicates that a linked list is empty?
5. Modify the function in Figure 8.19 so that it stops printing once a particular name has been printed.
6. Based on the technique of this section for implementing a stack in a contiguous block of cells, what condition indicates that the stack is empty?

7. Describe how a stack can be implemented in a high-level language in terms of a one-dimensional array.
8. When a queue is implemented in a circular fashion as described in this section, what is the relationship between the head and tail pointers when the queue is empty? What about when the queue is full? How can one detect whether a queue is full or empty?
9. Draw a diagram representing how the tree below appears in memory when stored using the left and right child pointers, as described in this section. Then, draw another diagram showing how the tree would appear in contiguous storage using the alternative storage system described in this section.



8.4 A Short Case Study

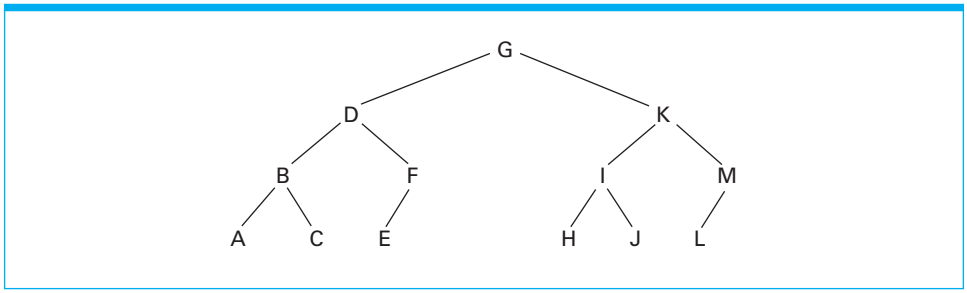
Let us consider the task of storing a list of names in alphabetical order. We assume that the operations to be performed on this list are the following:

search for the presence of an entry,
print the list in alphabetical order, and
insert a new entry

Our goal is to develop a storage system along with a collection of functions to perform these operations—thus producing a complete abstract tool.

We begin by considering options for storing the list. If the list were stored according to the linked list model, we would need to search the list in a sequential fashion, a process that, as we discussed in Chapter 5, could be very inefficient if the list becomes long. We will therefore seek an implementation that allows us to use the binary search algorithm (Section 5.5) for our search procedure. To apply this algorithm, our storage system must allow us to find the middle entry of successively smaller portions of the list. Our solution is to store the list as a binary tree. We make the middle list entry the root node. Then we make the middle of the remaining first half of the list the root's left child, and we make the middle of the remaining second half the root's right child. The middle entries of each remaining fourth of the list become the children of the root's children and so forth. For example, the tree in Figure 8.20 represents the list of letters A, B, C, D, E, F, G, H, I, J, K, L, and M. (We consider the larger of the middle two entries as the middle when the part of the list in question contains an even number of entries.)

Figure 8.20 The letters A through M arranged in an ordered tree



To search the list stored in this manner, we compare the target value to the root node. If the two are equal, our search has succeeded. If they are not equal, we move to the left or right child of the root, depending on whether the target is less than or greater than the root, respectively. There we find the middle of the portion of the list that is necessary to continue the search. This process of comparing and moving to a child continues until we find the target value (meaning that our search was successful) or we reach a null pointer (`None`) without finding the target value (meaning that our search was a failure).

Figure 8.21 shows how this search process can be expressed in the case of a linked tree structure. The Python `elif` keyword is a shortcut for "`else: if ...`". Note that this function is merely a refinement of the function in Figure 5.14, which is our original statement of the binary search. The distinction is largely cosmetic. Instead of stating the algorithm in terms of searching successively smaller segments of the list, we now state the algorithm in terms of searching successively smaller subtrees (Figure 8.22).

Having stored our "list" as a binary tree, you might think that the process of printing the list in alphabetical order would now be difficult. However, to print the list in alphabetical order, we merely need to print the left subtree in alphabetical order, print the root node, and then print the right subtree in alphabetical

Figure 8.21 The binary search as it would appear if the list were implemented as a linked binary tree

```
def Search(Tree, TargetValue):
  if (Tree is None):
    return None            # Search failed
  elif (TargetValue == Tree.Value):
    return Tree            # Search succeeded
  elif (TargetValue < Tree.Value):
    # Continue search in left subtree.
    return Search(Tree.Left, TargetValue)
  elif (TargetValue > Tree.Value):
    # Continue search in right subtree.
    return Search(Tree.Right, TargetValue)
```

Garbage Collection

As dynamic data structures grow and shrink, storage space is used and released. The process of reclaiming unused storage space for future use is known as **garbage collection.** Garbage collection is required in numerous settings. The memory manager within an operating system must perform garbage collection as it allocates and retrieves memory space. The file manager performs garbage collection as files are stored in and deleted from the machine's mass storage. Moreover, any process running under the control of the dispatcher might need to perform garbage collection within its own allotted memory space.

Garbage collection involves some subtle problems. In the case of linked structures, each time a pointer to a data item is changed, the garbage collector must decide whether to reclaim the storage space to which the pointer originally pointed. The problem becomes especially complex in intertwined data structures involving multiple paths of pointers. Inaccurate garbage collection routines can lead to loss of data or to inefficient use of storage space. For example, if garbage collection fails to reclaim storage space, the available space will slowly dwindle away, a phenomenon known as a **memory leak.**

order (Figure 8.23). After all, the left subtree contains all the elements that are less than the root node, while the right subtree contains all the elements that are greater than the root. A sketch of our logic so far looks like this:

```
if (tree not empty):
  print the left subtree in alphabetical order
  print the root node
  print the right subtree in alphabetical order
```

Figure 8.22 The successively smaller trees considered by the function in Figure 8.21 when searching for the letter J

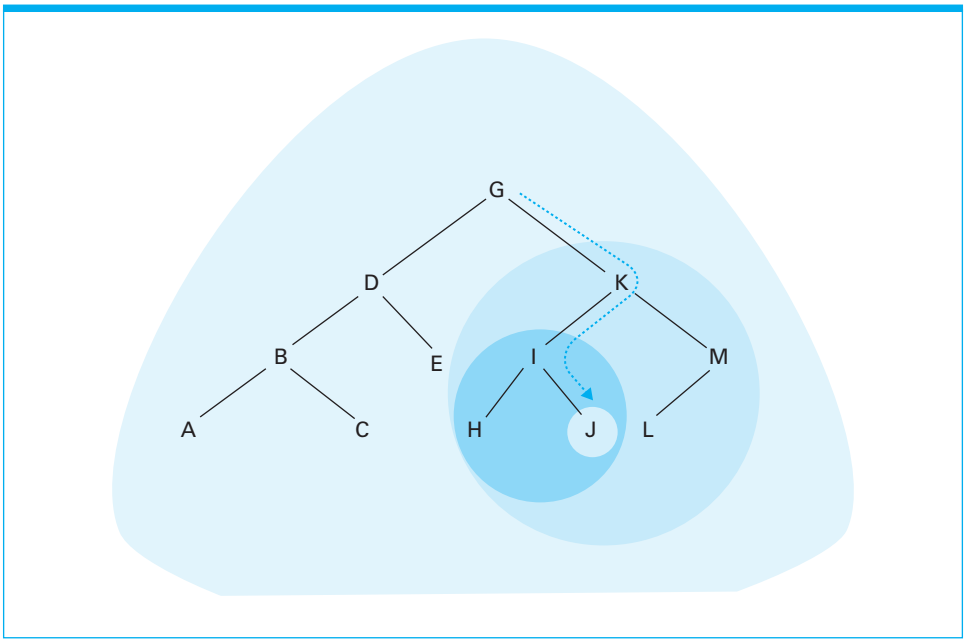
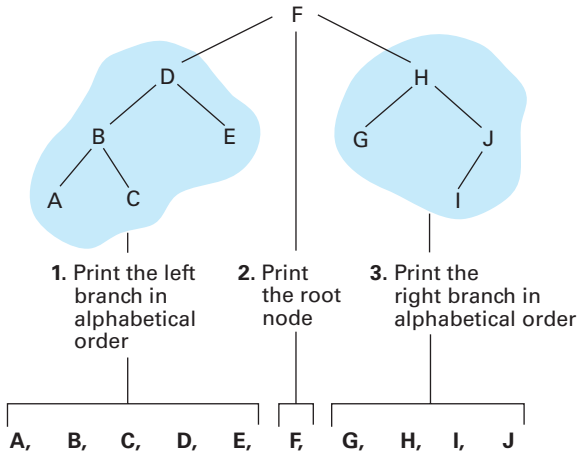


Figure 8.23 Printing a search tree in alphabetical order



This outline involves the tasks of printing the left subtree and the right subtree in alphabetical order, both of which are essentially smaller versions of our original task. That is, solving the problem of printing a tree involves the smaller task of printing subtrees, which suggests a recursive approach to our tree printing problem.

Following this lead, we can expand our initial idea into a complete Python function for printing our tree as shown in Figure 8.24. We have assigned the function the name `PrintTree` and then requested the services of `PrintTree` for printing the left and right subtrees. Note that the termination condition of the recursive process (reaching a null subtree, "`None`") is guaranteed to be reached, because each successive activation of the function operates on a smaller tree than the one causing the activation.

The task of inserting a new entry in the tree is also easier than it might appear at first. Your intuition might lead you to believe that insertions might require cutting the tree open to allow room for new entries, but actually the node being added can always be attached as a new leaf, regardless of the value involved. To find the proper place for a new entry, we move down the tree along the path that we would follow if we were searching for that entry. Since the entry is not in

Figure 8.24 A function for printing the data in a binary tree

```
def PrintTree(Tree):
  if (Tree != None):
    PrintTree(Tree.Left)
    print(Tree.Value)
    PrintTree(Tree.Right)
```

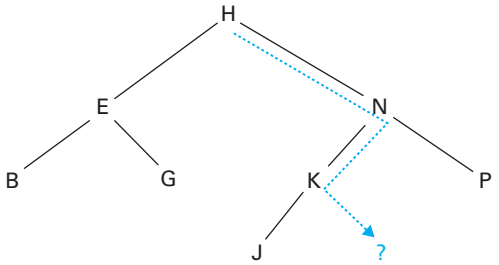
the tree, our search will lead to a null pointer. At this point we will have found the proper location for the new node (Figure 8.25). Indeed, this is the location to which a search for the new entry would lead.

A function expressing this process in the case of a linked tree structure is shown in Figure 8.26. It searches the tree for the value being inserted (called `NewValue`) and then places a new leaf node containing `NewValue` at the proper location. Note that if the entry being inserted is actually found in the tree during the search, no insertion is made. The Python code in Figure 8.26 uses the function call `TreeNode()` to create a new aggregate to serve as a fresh leaf in the linked tree structure. This requires additional code outside of the figure to identify `TreeNode` as user-defined type, as we will see in the next section.

We conclude that a software package consisting of a linked binary tree structure together with our functions for searching, printing, and inserting provides a complete package that could be used as an abstract tool by our hypothetical application. Indeed, when properly implemented, this package could be used without concern for the actual underlying storage structure. By using the procedures in the package, the user could envision a list of names stored in alphabetical order, whereas the reality would be that the "list" entries are actually scattered among blocks of memory cells that are linked as a binary tree.

Figure 8.25 Inserting the entry M into the list B, E, G, H, J, K, N, P stored as a tree

a. Search for the new entry until its absence is detected



b. This is the position in which the new entry should be attached

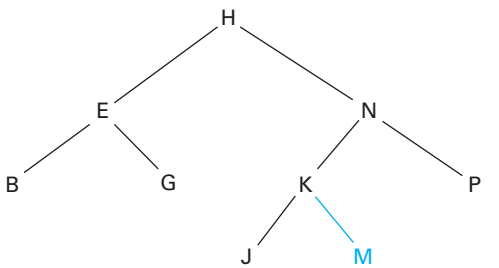


Figure 8.26 A function for inserting a new entry in a list stored as a binary tree

```
def Insert(Tree, NewValue):
  if (Tree is None):
    # Create a new leaf with NewValue
    Tree = TreeNode()
    Tree.Value = NewValue
  elif (NewValue < Tree.Value):
    # Insert NewValue into the left subtree
    Tree.Left = Insert(Tree.Left, NewValue)
  elif (NewValue > Tree.Value):
    # Insert NewValue into the right subtree
    Tree.Right = Insert(Tree.Right, NewValue)
  else:
    # Make no change.
  return Tree
```

Questions & Exercises

1. Draw a binary tree that you could use to store the list R, S, T, U, V, W, X, Y, and Z for future searching.
2. Indicate the path traversed by the binary search algorithm in Figure 8.21 when applied to the tree in Figure 8.20 in searching for the entry J. What about the entry P?
3. Draw a diagram representing the status of activations of the recursive tree-printing algorithm in Figure 8.24 at the time node K is printed within the ordered tree in Figure 8.20.
4. Describe how a tree structure in which each node has as many as 26 children could be used to encode the correct spelling of words in the English language.

8.5 Customized Data Types

In Chapter 6 we introduced the concept of a data type and discussed such elementary types as integer, float, character, and Boolean. These data types are provided in most programming languages as primitive data types. In this section we consider ways in which a programmer can define his or her own data types to fit more closely the needs of a particular application.

User-Defined Data Types

Expressing an algorithm is often easier if data types other than those provided as primitives in the programming language are available. For this reason, many modern programming languages allow programmers to define additional data types,

using the primitive types as building blocks. The most elementary examples of such "home-made" data types are known as **user-defined data types,** which are essentially conglomerates of primitive types collected under a single name.

To explain, suppose we wanted to develop a program involving numerous variables, each with the same aggregate structure consisting of a name, age, and skill rating. One approach would be to define each variable separately as an aggregate type (Section 6.2). A better approach, however, would be to define the aggregate to be a new (user-defined) data type and then to use that new type as though it were a primitive.

Recalling the example from Section 6.2, in C the statement

```
struct
{
  char  Name[25];
  int   Age;
  float SkillRating;
} Employee;
```

defines a new aggregate, called `Employee`, containing fields called `Name` (of type character), `Age` (of type integer), and `SkillRating` (of type float).

In contrast, the C statement

```
struct EmployeeType
{
  char  Name[25];
  int   Age;
  float SkillRating;
};
```

does not define a new aggregate variable, but defines a new aggregate type, `EmployeeType`. This new data type could then be used to declare variables in the same way as a primitive data type. That is, in the same way that C allows the variable *x* to be declared as an integer using the statement

```
int x;
```

the variable `Employee1` could be declared to be of the type `Employee` with the statement

```
struct EmployeeType Employee1;
```

Then, later in the program, the variable `Employee1` would refer to an entire block of memory cells containing the name, age, and skill rating of an employee. Individual items within the block could be referenced by expressions such as `Employee1.Name` and `Employee1.Age`. Thus, a statement such as

```
Employee1.Age = 26;
```

might be used to assign the value 26 to the `Age` field within the block known as `Employee1`. Moreover, the statement

```
struct EmployeeType DistManager, SalesRep1, SalesRep2;
```

could be used to declare the three variables `DistManager`, `SalesRep1`, and `SalesRep2` to be of type `EmployeeType` just as a statement of the form

```
float Sleeve, Waist, Neck;
```

is normally used to declare the variables `Sleeve`, `Waist`, and `Neck` to be of the primitive type `float`.

It is important to distinguish between a user-defined data type and an actual item of that type. The latter is referred to as an **instance** of the type. A user-defined data type is essentially a template that is used in constructing instances of the type. It describes the properties that all instances of that type have but does not itself constitute an occurrence of that type (just as a cookie-cutter is a template from which cookies are made but is not itself a cookie). In the preceding example, the user-defined data type `EmployeeType` was used to construct three instances of that type, known as `DistManager`, `SalesRep1`, and `SalesRep2`.

Abstract Data Types

User-defined data types such as C's structs and Pascal's records play an important role in many programming languages, helping the software designer to tailor the representation of data to the needs for a particular program. Traditional user-defined data types, however, merely allow programmers to define new storage systems. They do not also provide operations to be performed on data with these structures.

An **abstract data type (ADT)** is a user-defined data type that can include both data (representation) and functions (behavior). Programming languages that support creation of ADTs generally provide two features: (1) syntax for defining the ADT as single unit, and (2) a mechanism for hiding the internal structure of the ADT from other parts of the program that will make use of it. The first feature is an important organizational tool for keeping the data and functions of an ADT together, which simplifies maintenance and debugging. The second feature provides reliability, by preventing other code outside of the ADT from accessing its data without going through the functions that have been provided for that purpose.

To clarify, suppose we wanted to create and use several stacks of integer values within a program. Our approach might be to implement each stack as an array of 20 integer values. The bottom entry in the stack would be placed (pushed) into the first array position, and additional stack entries would be placed (pushed) into successively higher entries in the array (see question 7 in Section 8.3). An additional integer variable would be used as the stack pointer. It would hold the index of the array entry into which the next stack entry should be pushed. Thus each stack would consist of an array containing the stack itself and an integer playing the role of the stack pointer.

To implement this plan, we could first establish a user-defined type called StackType with a C statement of the form

```
struct StackType
{
  int StackEntries[20];
  int StackPointer = 0;
};
```

(Recall that in languages such as C, C++, C#, and Java, indices for the array `StackEntries` range from 0 to 19, so we have initialized `StackPointer` to the

value 0.) Having made this declaration, we could then declare stacks called StackOne, StackTwo, and StackThree via the statement

```
struct StackType StackOne, StackTwo, StackThree;
```

At this point, each of the variables StackOne, StackTwo, and StackThree would reference a unique block of memory cells used to implement an individual stack. But what if we now want to push the value 25 onto StackOne? We would like to avoid the details of the array structure underlying the stack's implementation and merely use the stack as an abstract tool—perhaps by using a function call similar to

```
push(25, StackOne);
```

But such a statement would not be available unless we also defined an appropriate function named push. Other operations we would like to perform on variables of type StackType would include popping entries off the stack, checking to see if the stack is empty, and checking to see if the stack is full—all of which would require definitions of additional functions. In short, our definition of the data type StackType has not included all the properties we would like to have associated with the type. Moreover, any function in the program can potentially access the StackPointer and StackEntries fields of our StackType variables, bypassing the careful checks we would design into our push and pop functions. A sloppy assignment statement in another part of the program could overwrite a data element stored in the middle of the stack data structure, or even destroy the LIFO behavior that is characteristic of all stacks.

What is needed is a mechanism for defining the operations that are allowed on our StackType, as well as for protecting the internal variables from outside interference. One such mechanism is the Java language's interface syntax. For example, in Java, we could write

```
interface StackType
{
  public int pop(); /* Return the item at top of stack */
  public void push(int item); /* Add a new item to stack */
  public boolean isEmpty(); /* Check if stack is emtpy */
  public boolean isFull(); /* Check if stack is full */

}
```

Alone, this abstract data type does not specify how the stack will be stored, or what algorithms will be used to execute the push, pop, isEmpty, and isFull functions. Those details (which have been *abstracted* away in this interface) will be specified in other Java code elsewhere. However, like our user-defined data type before, programmers are able to declare variables or function parameters to be of type StackType.

We could declare StackOne, StackTwo, and StackThree to be stacks with the statement

```
StackType StackOne, StackTwo, StackThree;
```

Later in the program (these three variables start out as null references and must be instantiated with concrete Java classes before use—but we are not concerned with those details here), we could push entries onto these stacks with statements such as

```
StackOne.push(25);
```

which means to execute the push function associated with StackOne using the value 25 as the actual parameter.

As opposed to the more elementary user-defined data types, abstract data types are complete data types, and their appearance in such languages as Ada in the 1980s represented a significant step forward in programming language design. Today, object-oriented languages provide for extended versions of abstract data types called classes, as we will see in the next section.

Questions & Exercises

1. What is the difference between a data type and an instance of that type?
2. What is the difference between a user-defined data type and an abstract data type?
3. Describe an abstract data type for implementing a list.
4. Describe an abstract data type for implementing checking accounts.

8.6 Classes and Objects

As we learned in Chapter 6, the object-oriented paradigm leads to systems composed of units called objects that interact with each other to accomplish tasks. Each object is an entity that responds to messages received from other objects. Objects are described by templates known as classes.

In many respects, these classes are actually descriptions of abstract data types (whose instances are called objects). For example, Figure 8.27 shows how a class known as StackOfIntegers can be defined in the languages Java and C#. (The equivalent class definition in C++ has the same structure but slightly different syntax.) Note that this class provides a body for each of the functions declared in the abstract data type StackType. In addition, this class contains an array of integers called StackEntries, and an integer used to identify the top of the stack within the array called StackPointer.

Using this class as a template, an object named StackOne can be created in a Java or C# program by the statement

```
StackType StackOne = new StackOfIntegers();
```

or in a C++ program by the statement

```
StackOfIntegers StackOne();
```

Later in the programs, the value 106 can be pushed onto StackOne using the statement

```
StackOne.push(106);
```

or the top entry from StackOne can be retrieved and placed in the variable OldValue using the statement

```
OldValue = StackOne.pop();
```

Figure 8.27 A stack of integers implemented in Java and C#

```
class StackOfIntegers implements StackType
{
  private int[] StackEntries = new int[20];
  private int StackPointer = 0;

  public void push(int NewEntry)
  {  if (StackPointer < 20)
       StackEntries[StackPointer++] = NewEntry;
  }

  public int pop()
  {  if (StackPointer > 0) return StackEntries[--StackPointer];
       else return 0;
  }

  public boolean isEmpty()
  {      return (StackPointer == 0);    }

  public boolean isFull()
  {      return (StackPointer >= MAX);    }
}
```

These features are essentially the same as those associated with abstract data types. There are, however, distinctions between classes and abstract data types. The former is an extension of the latter. For instance, as we learned in Section 6.5, object-oriented languages allow classes to inherit properties from other classes and to contain special methods called constructors that customize individual objects when they are created. Moreover, classes can be associated with varying degrees of encapsulation (Section 6.5), allowing the internal properties of their instances to be protected from misguided shortcuts, while exposing other fields to be available externally.

We conclude that the concepts of classes and objects represent another step in the evolution of techniques for representing data abstractions in programs. It is, in fact, the ability to define and use abstractions in a convenient manner that has led to the popularity of the object-oriented programming paradigm.

The Standard Template Library

The data structures discussed in this chapter have become standard programming structures—so standard, in fact, that many programming environments treat them very much like primitives. One example is found in the C++ programming environment, which is enhanced by the Standard Template Library (STL). The STL contains a collection of predefined classes that describe popular data structures. Consequently, by incorporating the STL into a C++ program, the programmer is relieved from the task of describing these structures in detail. Instead, he or she needs merely to declare identifiers to be of these types in the same manner that we declared `StackOne` to be of type `StackOfIntegers` in Section 8.6.

Questions & Exercises

1. In what ways are abstract data types and classes similar? In what ways are they different?
2. What is the difference between a class and an object?
3. Describe a class that would be used as a template for constructing objects of type queue-of-integers.

8.7 Pointers in Machine Language

In this chapter we have introduced pointers and have shown how they are used in constructing data structures. In this section we consider how pointers are handled in machine language.

Suppose that we want to write a program in the machine language described in Appendix C to pop an entry off the stack as described in Figure 8.12 and place that entry in a general-purpose register. In other words, we want to load a register with the contents of the memory cell that contains the entry on top of the stack. Our machine language provides two instructions for loading registers: one with op-code 2, the other with op-code 1. Recall that in the case of op-code 2, the operand field contains the data to be loaded, and in the case of op-code 1, the operand field contains the address of the data to be loaded.

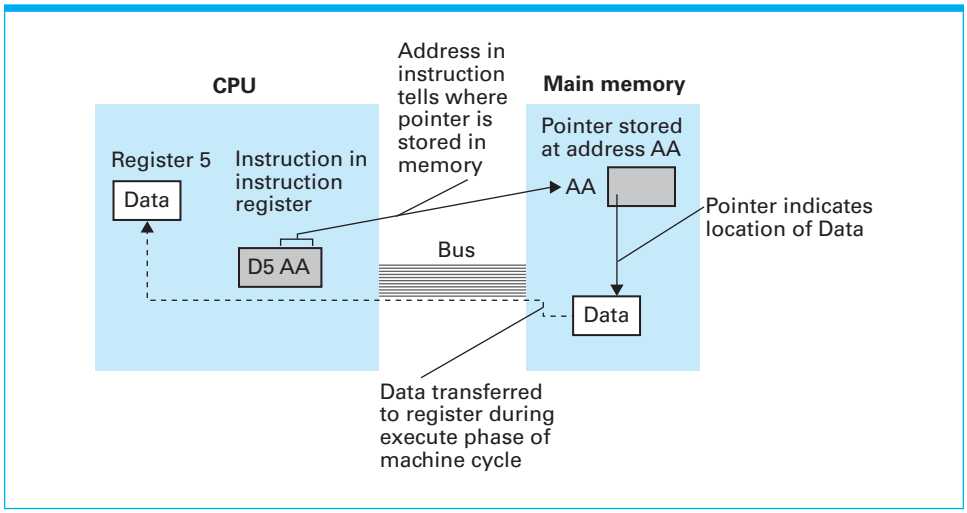
We do not know what the contents will be, so we cannot use op-code 2 to obtain our goal. Moreover, we cannot use op-code 1, because we do not know what the address will be. After all, the address of the top of the stack will vary as the program is executed. However, we do know the address of the stack pointer. That is, we know the location of the address of the data we want to load. What we need, then, is a third op-code for loading a register, in which the operand contains the address of a pointer to the data to be loaded.

To accomplish this goal we could extend the language in Appendix C to include an op-code D. An instruction with this op-code could have the form DRXY, which would mean to load register R with the contents of the memory cell whose address is found at address XY (Figure 8.28). Thus if the stack pointer is in the memory cell at address AA, then the instruction D5AA would cause the data at the top of the stack to be loaded into register 5.

This instruction, however, does not complete the pop operation. We must also subtract one from the stack pointer so that it points to the new top of the stack. This means that, following the load instruction, our machine language program would have to load the stack pointer into a register, subtract one from it, and store the result back in memory.

By using one of the registers as the stack pointer instead of a memory cell, we could reduce this movement of the stack pointer back and forth between registers and memory. But this would mean that we must redesign the load instruction so that it expects the pointer to be in a register rather than in main memory. Thus, instead of the earlier approach, we might define an instruction with op-code D to have the form DR0S, which would mean to load register R

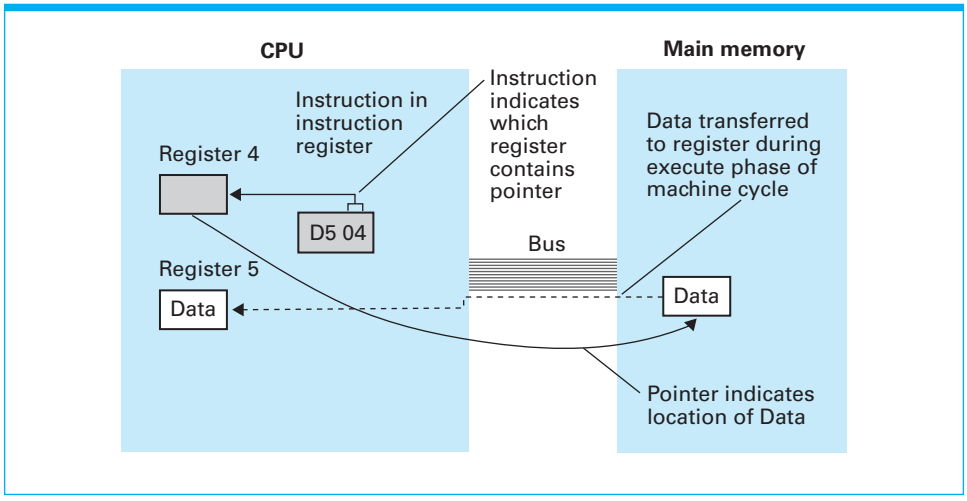
Figure 8.28 Our first attempt at expanding the machine language in Appendix C to take advantage of pointers



with the contents of the memory cell pointed to by register S (Figure 8.29). Then, a complete pop operation could be performed by following this instruction with an instruction (or instructions) to subtract one from the value stored in register S.

Note that a similar instruction is needed to implement a push operation. We might therefore extend the language described in Appendix C further by introducing the op-code E so that an instruction of the form ER0S would mean to store the contents of register R in the memory cell pointed to by register S. Again, to complete the push operation, this instruction would be followed by an instruction (or instructions) to add one to the value in register S.

Figure 8.29 Loading a register from a memory cell that is located by means of a pointer stored in a register



These new op-codes D and E that we have proposed not only demonstrate how machine languages are designed to manipulate pointers, they also demonstrate an addressing technique that was not present in the original machine language. As presented in Appendix C, the machine language uses two means of identifying the data involved in an instruction. The first of these is demonstrated by an instruction whose op-code is 2. Here, the operand field contains the data involved explicitly. This is called **immediate addressing.** The second means of identifying data is demonstrated by instructions with op-codes 1 and 3. Here the operand fields contain the address of the data involved. This is called **direct addressing.** However, our proposed new op-codes D and E demonstrate yet another form of identifying data. The operand fields of these instructions contain the address of the address of the data. This is called **indirect addressing.** All three are common in today's machine languages.

Questions & Exercises

1. Suppose the machine language described in Appendix C has been extended as suggested at the end of this section. Moreover, suppose register 8 contains the pattern DB, the memory cell at address DB contains the pattern CA, and the cell at address CA contains the pattern A5. What bit pattern will be in register 5 immediately after executing each of the following instructions?

   a. 25A5
   b. 15CA
   c. D508

2. Using the extensions described at the end of this section, write a complete machine language routine to perform a pop operation. Assume that the stack is implemented as shown in Figure 8.12, the stack pointer is in register F, and the top of the stack is to be popped into register 5.
3. Using the extensions described at the end of this section, write a program to copy the contents of five contiguous memory cells starting at address A0 to the five cells starting at address B0. Assume your program starts at address 00.
4. In the chapter, we introduced a machine instruction of the form DR0S. Suppose we extended this form to DRXS, meaning "Load register R with the data pointed to by the value in register S plus the value X." Thus the pointer to the data is obtained by retrieving the value in register S and then incrementing that value by X. The value in register S is not altered. (If register F contained 04, then the instruction DE2F would load register E with the contents of the memory cell at address 06. The value of register F would remain 04.) What advantages would this instruction have? What about an instruction of the form DRTS—meaning "Load register R with the data pointed to by the value in register S incremented by the value in register T"?

Chapter Review Problems

(Asterisked problems are associated with optional sections.)

1. Draw pictures showing how the array below appears in a machine's memory when stored in row major order and in column major order:

A	B	C	D
E	F	G	H
I	J	K	L

2. Suppose an array with six rows and eight columns is stored in row major order starting at address −50 (base 10). If each entry in the array requires two memory cells, what is the address of the entry in the fifth row and seventh column? What if each entry requires three memory cells?
3. Rework question 2 assuming column major order rather than row major order.
4. Write a function such that if an element in an M × N matrix is 0, its entire row and column are set to 0.
5. Why is a contiguous list considered to be a convenient storage structure for implementing static lists, but not for implementing dynamic lists? Explain your answer.
6. Suppose you want to insert the number 3 in the list of numbers 1, 2, 4, 5, 6, 7, 8. What activities are required to insert the number 3 in the list, assuming that the order of the list is to be maintained?
7. The following table represents the contents of some cells in a computer's main memory along with the address of each cell represented. Note that some of the cells contain letters of the alphabet, and each such cell is followed by an empty cell. Place addresses in these empty cells so that each cell containing a letter together with the following cell form an entry in a linked list in which the letters appear in alphabetical order. (Use zero for the null pointer.) What address should the head pointer contain?

Address	Contents
11	C
12	
13	G
14	
15	E
16	
17	B
18	
19	U
20	
21	F
22	

8. The following table represents a portion of a linked list in a computer's main memory. Each entry in the list consists of two cells: The first contains a letter of the alphabet; the second contains a pointer to the next list entry. Alter the pointers so that the letter N is no longer in the list. Then replace the letter N with the letter G and alter the pointers so that the new letter appears in the list in its proper place in alphabetical order.

Address	Contents
30	J
31	38
32	B
33	30
34	X
35	46
36	N
37	40
38	K
39	36
40	P
41	34

9. The following table represents a linked list using the same format as in the preceding problems. If the head pointer contains the value 44, what name is represented by

the list? Change the pointers so that the list contains the name Jean.

Address	Contents
40	N
41	46
42	I
43	40
44	J
45	50
46	E
47	00
48	M
49	42
50	A
51	40

10. Which of the following routines correctly inserts `NewEntry` immediately after the entry called `PreviousEntry` in a linked list? What is wrong with the other routine?

Routine 1:

```
1. Copy the value in the pointer
   field of PreviousEntry into the
   pointer field of NewEntry.
2. Change the value in the pointer
   field of PreviousEntry to the
   address of NewEntry.
```

Routine 2:

```
1. Change the value in the pointer
   field of PreviousEntry to the
   address of NewEntry.
2. Copy the value in the pointer
   field of PreviousEntry into the
   pointer field of NewEntry.
```

11. Design a function to find the `Kth` element in a single linked list with `n` elements.

12. Design a function to check whether the elements of a single linked list with `n` elements, form a palindrome.

13. Design a function for counting the nodes of a linked list.

14. a. Suppose you were given a linked list with `n` elements. Design an algorithm to remove duplicates from an unsorted linked list.

b. If the use of a temporary buffer is not permitted to solve part (a), how would you solve the problem?

15. Sometimes a single linked list is given two different orders by attaching two pointers to each entry rather than one. Fill in the table below so that by following the first pointer after each letter one finds the name Carol, but by following the second pointer after each letter one finds the letters in alphabetical order. What values belong in the head pointer of each of the two lists represented?

Address	Contents
60	O
61	
62	
63	C
64	
65	
66	A
67	
68	
69	L
70	
71	
72	R
73	
74	

16. The table below represents a stack stored in a contiguous block of memory cells, as discussed in the text. If the base of the stack is at address 10 and the stack pointer contains the value 12, what value is retrieved by a pop instruction? What value is in the stack pointer after the pop operation?

Address	Contents
10	F
11	C
12	A
13	B
14	E

17. Draw a table showing the final contents of the memory cells if the instruction in question 16 had been to push the letter D on the stack rather than to pop a letter. What would the value in the stack pointer be after the push instruction?

18. How would you design a stack which, in addition to the traditional push and pop operations, also supports an operation called `min` which returns the element having the minimum value in the stack? `push`, `pop`, and `min` should all operate in O(1) time.

19. Describe how a single array can be used to implement three stacks.

20. Suppose you were given a stack and you were allowed to use one additional stack, without copying the elements into any other data structure. Write a program to sort the stack in ascending order (biggest items on the top). The stack supports `push`, `pop`, `peek`, and `isEmpty` operations.

21. Suppose you were given three stacks and you were only allowed to move entries one at a time from one stack to another. Design an algorithm for reversing two adjacent entries on one of the stacks.

22. Suppose we want to create a stack of names that vary in length. Why is it advantageous to store the names in separate areas of memory and then build the stack out of pointers to these names rather than allowing the stack to contain the names themselves?

23. Design a function called `TwoStacks` which implements a queue by using two stacks.

24. Suppose a stack consists of boxes of specific width, height, and depth. Boxes can be stacked on the top if the underlying box is larger in all dimensions, but cannot be rotated. Try to implement the tallest stack, where the height of a stack is the sum of the heights of each box.

25. Suppose the entries in a queue require one memory cell each, the head pointer contains the value 11, and the tail pointer contains the value 17. What are the values of these pointers after one entry is inserted and two are removed?

26. The Towers of Hanoi is a classic puzzle consisting of 3 towers, and N disks of different sizes which can slide onto any tower. The puzzle starts with the disks sorted in an ascending order from top to bottom. It has the following constraints:

a. Only one disk can be moved at a time.
b. A disk is moved from the top of one tower onto the next tower.
c. A disk can only be placed on top of a larger disk.

Write a program to move the disks from the first tower to the last using stacks.

27. Design a method that counts all the possible ways in which a person can run up the stairs, if he can jump one, two or three steps at a time.

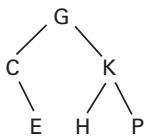
28. Suppose you were given two queues and you were only allowed to move one entry at a time from the head of a queue to the tail of either. Design an algorithm for reversing two adjacent entries in one of the queues.

29. The table below represents a tree stored in a machine's memory. Each node of the tree consists of three cells. The first cell contains the data (a letter), the second contains a pointer to the node's left child, and the third contains a pointer to the node's right child. A value of 0 represents a null pointer. If the value of the root pointer is 55, draw a picture of the tree.

Address	Contents
40	G
41	0
42	0
43	X
44	0
45	0
46	J
47	49
48	0
49	M
50	0
51	0
52	F
53	43
54	40
55	W
56	46
57	52

30. The table below represents the contents of a block of cells in a computer's main memory. Note that some of the cells contain letters of the alphabet, and each of those cells is followed by two blank cells. Fill in the blank cells so that the memory block represents the tree that follows. Use the first cell following a letter as the pointer to that node's left child and the next cell as the pointer to the right child. Use 0 for null pointers. What value should be in the root pointer?

Address	Contents
30	C
31	
32	
33	H
34	
35	
36	K
37	
38	
39	E
40	
41	
42	G
43	
44	
45	P
46	
47	



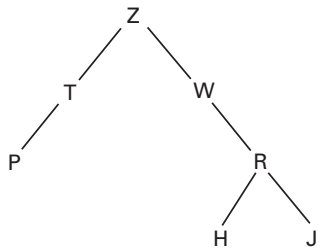
31. Write an algorithm to create a binary search tree with minimal height, for a given sorted array of integers.

32. Design a nonrecursive algorithm to replace the recursive one represented in Figure 8.24. Use a stack to control any backtracking that might be necessary.

33. Apply the recursive tree-printing algorithm of Figure 8.24. Draw a diagram representing the nested activations of the algorithm (and the current position in each) at the time node X is printed.

34. Implement a function to check if a binary tree is balanced. A balanced tree is defined to be a tree in which the heights of two subtrees of any node never differ by more than one.

35. Draw a diagram showing how the binary tree below appears in memory when stored without pointers using a block of contiguous memory cells as described in Section 8.3.



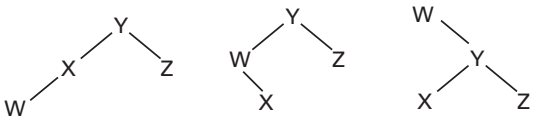
36. Suppose each node in a binary tree contains a value. Design a method that prints all the paths which sum to a specified value. The path can start at an arbitrary node.

37. Give an example in which you might want to implement a list (the conceptual structure) as a tree (the actual underlying structure). Give an example in which you might want to implement a tree (the conceptual structure) as a list (the actual underlying structure).

38. Suppose a binary tree Tl has millions of nodes, and another tree T2, has hundreds of nodes. Create an algorithm to decide whether T2 is a subtree of Tl. T2 would be a subtree of Tl if there is a node "n" in Tl whose subtree is identical to T2.

39. Design an algorithm to check whether a route between two nodes exists in a directed graph.

40. Identify the trees below whose nodes would be printed in alphabetical order by the algorithm in Figure 8.24.



41. Modify the function in Figure 8.24 to print the "list" in reverse order.

42. Suppose a call center has three levels of employees—respondent, manager, and director. An incoming telephone call must first be allocated to a respondent who is free. If the respondent cannot handle the call, the call must be escalated to a manager. If the manager is occupied, then the call should be escalated to a director. Design the classes and data structures for this problem. Implement a method `dispatchCall()` which assigns a call to the first available employee.

43. Design a procedure for finding and deleting a given value from a tree stored in the fashion of Figure 8.20.

44. In the traditional implementation of a tree, each node is constructed with a separate pointer for each possible child. The number of such pointers is a design decision and represents the maximum number of children any node can have. If a node has fewer children than pointers, some of its pointers are simply set to null. But such a node can never have more children than pointers. Describe how a tree could be implemented without limiting the number of children a node could have.

45. Using pseudocode modeled on the C struct statement introduced in Section 8.5, define a user-defined data type representing data regarding an employee of a company (such as name, address, job assignment, pay scale, and so on).

46. Using pseudocode similar to the Java class syntax of Figure 8.27, sketch a definition of an abstract data type representing a list of names. In particular, what structure would contain the list and what functions would be provided to manipulate the list? (You do not need to include detailed descriptions of the functions.)

47. Using pseudocode similar to the Java class syntax of Figure 8.27, sketch a definition of an abstract data type representing a queue. Then give pseudocode statements showing how instances of that type could be created and how entries could be inserted in and deleted from those instances.

48. a. What is the difference between a user-defined data type and a primitive data type?
b. What is the difference between an abstract data type and a user-defined data type?

49. Identify the data structures and procedures that might appear in an abstract data type representing an address book.

50. Identify the data structures and procedures that might appear in an abstract data type representing a simple spacecraft in a video game.

51. Modify Figure 8.27 and the StackType interface from Section 8.5 so that the class defines a queue rather than a stack.

52. In what way is a class more general than a traditional abstract data type?

***53.** Using instructions of the form DR0S and ER0S as described at the end of Section 8.7, write a complete machine language routine to push an entry onto a stack implemented as shown in Figure 8.12. Assume that the stack pointer is in register F and that the entry to be pushed is in register 5.

***54.** Suppose a Boolean expression consisting of the symbols 0, 1, &, |, and A, and a desired Boolean result value is given. Implement a function to count the number of ways to parenthesize the expression, such that it evaluates to the result. For example, if the expression is 1 A 01011 and the desired result is false (0), the output will be 1A((010) 11) and 1 A (91 (011)).

***55.** What advantages does an instruction of the form DR0S as described in Section 8.7 have over an instruction of the form DRXY? What advantage does the form DRXS as described in question 4 of Section 8.7 have over the form DR0S?

Social Issues

The following questions are intended as a guide to the ethical/social/legal issues associated with the field of computing. The goal is not merely to answer these questions. You should also consider why you answered as you did and whether your justifications are consistent from one question to the next.

1. Suppose a software analyst designs a data organization that allows for efficient manipulation of data in a particular application. How can the rights to that data structure be protected? Is a data structure the expression of an idea (like a poem) and therefore protected by copyright or do data structures fall through the same legal loopholes as algorithms? What about patent law?
2. To what extent is incorrect data worse than no data?
3. In many application programs, the size to which a stack can grow is determined by the amount of memory available. If the available space is consumed, then the software is designed to produce a message such as "stack overflow" and terminate. In most cases this error never occurs, and the user is never aware of it. Who is liable if such an error occurs and sensitive information is lost? How could the software developer minimize his or her liability?
4. In a data structure based on a pointer system, the deletion of an item usually consists of changing a pointer rather than erasing memory cells. Thus when an entry in a linked list is deleted, the deleted entry actually remains in memory until its memory space is required by other data. What ethical and security issues result from this persistence of deleted data?
5. It is easy to transfer data and programs from one computer to another. Thus it is easy to transfer the knowledge held by one machine to many machines. In contrast, it sometimes takes a long time for a human to transfer knowledge to another human. For example, it takes time for a human to teach another human a new language. What implications could this contrast in knowledge transfer rate have if the capabilities of machines begin to challenge the capabilities of humans?
6. The use of pointers allows related data to be linked in a computer's memory in a manner reminiscent of the way many believe information is associated in the human mind. How are such links in a computer's memory similar to links in a brain? How are they different? Is it ethical to attempt to build computers that more closely mimic the human mind?
7. Has the popularization of computer technology produced new ethical issues or simply provided a new context in which previous ethical theories are applicable?
8. Suppose the author of an introductory computer science textbook wants to include program examples to demonstrate concepts in the text. However, to obtain clarity, many of the examples must be simplified versions of what would actually be used in professional-quality software. The author knows that the examples could be used by unsuspecting readers and ultimately could find their way into significant software applications in which more robust techniques would be more appropriate. Should the author use the simplified examples, insist that all examples be robust even if doing so decreases their demonstrative value, or refuse to use such examples unless clarity and robustness can both be obtained?

Additional Reading

Carrano, F. M., and T. Henry. *Data Abstraction and Problem Solving with C++: Walls and Mirrors,* 6th ed. Boston, MA: Addison-Wesley, 2012.

Gray, S. *Data Structures in Java: From Abstract Data Types to the Java Collections Framework.* Boston, MA: Addison-Wesley, 2007.

Main, M. *Data Structures and Other Objects Using Java,* 4th ed. Boston, MA: Addison-Wesley, 2011.

Main, M., and W. Savitch. *Data Structures and Other Objects Using C++*, 4th ed. Boston, MA: Addison-Wesley, 2010.

Prichard, J., and F.M. Carrano. *Data Abstraction and Problem Solving with Java: Walls and Mirrors,* 3rd ed. Boston, MA: Addison-Wesley, 2010.

Shaffer, C. A. *Practical Introduction to Data Structures and Algorithm Analysis,* 2nd ed. Upper Saddle River, NJ: Prentice Hall, 2001.

Weiss, M. A. *Data Structures and Problem Solving Using Java,* 4th ed. Boston, MA: Addison-Wesley, 2011.

Weiss, M. A. *Data Structures and Algorithm Analysis in C++*, 4th ed. Boston, MA: Addison-Wesley, 2013.

Weiss, M. A. *Data Structures and Algorithm Analysis in Java,* 3rd ed. Boston, MA: Addison-Wesley, 2011.