Summary

- A data structure is the organization of data in a computer's memory or in a disk file.
- The correct choice of data structure allows major improvements in program efficiency.
- Examples of data structures are arrays, stacks, and linked lists.
- An algorithm is a procedure for carrying out a particular task.
- In Java, an algorithm is usually implemented by a class method.
- Many of the data structures and algorithms described in this book are most often used to build databases.
- Some data structures are used as programmer's tools: They help execute an algorithm.
- Other data structures model real-world situations, such as telephone lines running between cities.
- A database is a unit of data storage composed of many similar records.
- A record often represents a real-world object, such as an employee or a car part.
- A record is divided into fields. Each field stores one characteristic of the object described by the record.
- A key is a field in a record that's used to carry out some operation on the data. For example, personnel records might be sorted by a `LastName` field.
- A database can be searched for all records whose key field has a certain value. This value is called a search key.

Questions

These questions are intended as a self-test for readers. Answers to the questions may be found in Appendix C.

1. In many data structures you can _____ a single record, _____ it, and _____ it.
2. Rearranging the contents of a data structure into a certain order is called _____ .

3. In a database, a field is
   a. a specific data item.
   b. a specific object.
   c. part of a record.
   d. part of an algorithm.
4. The field used when searching for a particular record is the _____ .
5. In object-oriented programming, an object
   a. is a class.
   b. may contain data and methods.
   c. is a program.
   d. may contain classes.
6. A class
   a. is a blueprint for many objects.
   b. represents a specific real-world object.
   c. will hold specific values in its fields.
   d. specifies the type of a method.
7. In Java, a class specification
   a. creates objects.
   b. requires the keyword `new`.
   c. creates references.
   d. none of the above.
8. When an object wants to do something, it uses a _____ .
9. In Java, accessing an object's methods requires the _____ operator.
10. In Java, `boolean` and `byte` are _____ .

(There are no experiments or programming projects for Chapter 1.)

quickly, in O(1) time, but searching takes slow O(N) time. In an ordered array you can search quickly, in O(logN) time, but insertion takes O(N) time. For both kinds of arrays, deletion takes O(N) time because half the items (on the average) must be moved to fill in the hole.

It would be nice if there were data structures that could do everything—insertion, deletion, and searching—quickly, ideally in O(1) time, but if not that, then in O(logN) time. In the chapters ahead, we'll see how closely this ideal can be approached, and the price that must be paid in complexity.

Another problem with arrays is that their size is fixed when they are first created with `new`. Usually, when the program first starts, you don't know exactly how many items will be placed in the array later, so you guess how big it should be. If your guess is too large, you'll waste memory by having cells in the array that are never filled. If your guess is too small, you'll overflow the array, causing at best a message to the program's user, and at worst a program crash.

Other data structures are more flexible and can expand to hold the number of items inserted in them. The linked list, discussed in Chapter 5, "Linked Lists," is such a structure.

We should mention that Java includes a class called `Vector` that acts much like an array but is expandable. This added capability comes at the expense of some loss of efficiency.

You might want to try creating your own vector class. If the class user is about to overflow the internal array in this class, the insertion algorithm creates a new array of larger size, copies the old array contents to the new array, and then inserts the new item. This whole process would be invisible to the class user.

Summary

- Arrays in Java are objects, created with the `new` operator.
- Unordered arrays offer fast insertion but slow searching and deletion.
- Wrapping an array in a class protects the array from being inadvertently altered.
- A class interface is composed of the methods (and occasionally fields) that the class user can access.
- A class interface can be designed to make things simple for the class user.
- A binary search can be applied to an ordered array.
- The logarithm to the base B of a number A is (roughly) the number of times you can divide A by B before the result is less than 1.
- Linear searches require time proportional to the number of items in an array.

- Binary searches require time proportional to the logarithm of the number of items.
- Big O notation provides a convenient way to compare the speed of algorithms.
- An algorithm that runs in O(1) time is the best, O(log N) is good, O(N) is fair, and O(N^2) is pretty bad.

Questions

These questions are intended as a self-test for readers. Answers may be found in Appendix C.

1. Inserting an item into an unordered array
   a. takes time proportional to the size of the array.
   b. requires multiple comparisons.
   c. requires shifting other items to make room.
   d. takes the same time no matter how many items there are.
2. True or False: When you delete an item from an unordered array, in most cases you shift other items to fill in the gap.
3. In an unordered array, allowing duplicates
   a. increases times for all operations.
   b. increases search times in some situations.
   c. always increases insertion times.
   d. sometimes decreases insertion times.
4. True or False: In an unordered array, it's generally faster to find out an item is not in the array than to find out it is.
5. Creating an array in Java requires using the keyword _____ .
6. If class A is going to use class B for something, then
   a. class A's methods should be easy to understand.
   b. it's preferable if class B communicates with the program's user.
   c. the more complex operations should be placed in class A.
   d. the more work that class B can do, the better.
7. When class A is using class B for something, the methods and fields class A can access in class B are called class B's _____.

8. Ordered arrays, compared with unordered arrays, are
   a. much quicker at deletion.
   b. quicker at insertion.
   c. quicker to create.
   d. quicker at searching.
9. A logarithm is the inverse of _____ .
10. The base 10 logarithm of 1,000 is _____ .
11. The maximum number of elements that must be examined to complete a binary search in an array of 200 elements is
    a. 200.
    b. 8.
    c. 1.
    d. 13.
12. The base 2 logarithm of 64 is _____ .
13. True or False: The base 2 logarithm of 100 is 2.
14. Big O notation tells
    a. how the speed of an algorithm relates to the number of items.
    b. the running time of an algorithm for a given size data structure.
    c. the running time of an algorithm for a given number of items.
    d. how the size of a data structure relates to the number of items.
15. O(1) means a process operates in _____ time.
16. Either variables of primitive types or _____ can be placed in an array.

Experiments

Carrying out these experiments will help to provide insights into the topics covered in the chapter. No programming is involved.

1. Use the Array Workshop applet to insert, search for, and delete items. Make sure you can predict what it's going to do. Do this both when duplicates are allowed and when they're not.
2. Make sure you can predict in advance what range the Ordered Workshop applet will select at each step.

3. In an array holding an even number of data items, there is no middle item. Which item does the binary search algorithm examine first? Use the Ordered Workshop applet to find out.

Programming Projects

Writing programs to solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied. (As noted in the Introduction, qualified instructors may obtain completed solutions to the Programming Projects on the publisher's Web site.)

2.1 To the `HighArray` class in the `highArray.java` program (Listing 2.3), add a method called `getMax()` that returns the value of the highest key in the array, or –1 if the array is empty. Add some code in `main()` to exercise this method. You can assume all the keys are positive numbers.

2.2 Modify the method in Programming Project 2.1 so that the item with the highest key is not only returned by the method, but also removed from the array. Call the method `removeMax()`.

2.3 The `removeMax()` method in Programming Project 2.2 suggests a way to sort the contents of an array by key value. Implement a sorting scheme that does not require modifying the `HighArray` class, but only the code in `main()`. You'll need a second array, which will end up inversely sorted. (This scheme is a rather crude variant of the selection sort in Chapter 3, "Simple Sorting.")

2.4 Modify the `orderedArray.java` program (Listing 2.4) so that the `insert()` and `delete()` routines, as well as `find()`, use a binary search, as suggested in the text.

2.5 Add a `merge()` method to the `OrdArray` class in the `orderedArray.java` program (Listing 2.4) so that you can merge two ordered source arrays into an ordered destination array. Write code in `main()` that inserts some random numbers into the two source arrays, invokes `merge()`, and displays the contents of the resulting destination array. The source arrays may hold different numbers of data items. In your algorithm you will need to compare the keys of the source arrays, picking the smallest one to copy to the destination. You'll also need to handle the situation when one source array exhausts its contents before the other.

2.6 Write a `noDups()` method for the `HighArray` class of the `highArray.java` program (Listing 2.3). This method should remove all duplicates from the array. That is, if three items with the key 17 appear in the array, `noDups()` should remove two of them. Don't worry about maintaining the order of the items. One approach is to first compare every item with all the other items and overwrite any duplicates with a `null` (or a distinctive value that isn't used for real keys). Then remove all the `null`s. Of course, the array size will be reduced.

last names. You want the algorithm to sort only what needs to be sorted, and leave everything else in its original order. Some sorting algorithms retain this secondary ordering; they're said to be *stable*.

All the algorithms in this chapter are stable. For example, notice the output of the `objectSort.java` program (Listing 3.4). Three persons have the last name of Smith. Initially, the order is Doc Smith, Lorraine Smith, and Paul Smith. After the sort, this ordering is preserved, despite the fact that the various Smith objects have been moved to new locations.

Comparing the Simple Sorts

There's probably no point in using the bubble sort, unless you don't have your algorithm book handy. The bubble sort is so simple that you can write it from memory. Even so, it's practical only if the amount of data is small. (For a discussion of what "small" means, see Chapter 15, "When to Use What.")

The selection sort minimizes the number of swaps, but the number of comparisons is still high. This sort might be useful when the amount of data is small and swapping data items is very time-consuming compared with comparing them.

The insertion sort is the most versatile of the three and is the best bet in most situations, assuming the amount of data is small or the data is almost sorted. For larger amounts of data, quicksort is generally considered the fastest approach; we'll examine quicksort in Chapter 7.

We've compared the sorting algorithms in terms of speed. Another consideration for any algorithm is how much memory space it needs. All three of the algorithms in this chapter carry out their sort *in place*, meaning that, besides the initial array, very little extra memory is required. All the sorts require an extra variable to store an item temporarily while it's being swapped.

You can recompile the example programs, such as `bubbleSort.java`, to sort larger amounts of data. By timing them for larger sorts, you can get an idea of the differences between them and the time required to sort different amounts of data on your particular system.

Summary

- The sorting algorithms in this chapter all assume an array as a data storage structure.
- Sorting involves comparing the keys of data items in the array and moving the items (actually, references to the items) around until they're in sorted order.

- All the algorithms in this chapter execute in $O(N^2)$ time. Nevertheless, some can be substantially faster than others.
- An invariant is a condition that remains unchanged while an algorithm runs.
- The bubble sort is the least efficient, but the simplest, sort.
- The insertion sort is the most commonly used of the $O(N^2)$ sorts described in this chapter.
- A sort is stable if the order of elements with the same key is retained.
- None of the sorts in this chapter require more than a single temporary variable, in addition to the original array.

Questions

These questions are intended as a self-test for readers. Answers may be found in Appendix C.

1. Computer sorting algorithms are more limited than humans in that
   a. humans are better at inventing new algorithms.
   b. computers can handle only a fixed amount of data.
   c. humans know what to sort, whereas computers need to be told.
   d. computers can compare only two things at a time.
2. The two basic operations in simple sorting are _____ items and _____ them (or sometimes _____ them).
3. True or False: The bubble sort always ends up comparing every item with every other item.
4. The bubble sort algorithm alternates between
   a. comparing and swapping.
   b. moving and copying.
   c. moving and comparing.
   d. copying and comparing.
5. True or False: If there are N items, the bubble sort makes exactly N*N comparisons.

6. In the selection sort,
   a. the largest keys accumulate on the left (low indices).
   b. a minimum key is repeatedly discovered.
   c. a number of items must be shifted to insert each item in its correctly sorted position.
   d. the sorted items accumulate on the right.
7. True or False: If, in a particular sorting situation, swaps take much longer than comparisons, the selection sort is about twice as fast as the bubble sort.
8. A copy is _____ times as fast as a swap.
9. What is the invariant in the selection sort?
10. In the insertion sort, the "marked player" described in the text corresponds to which variable in the `insertSort.java` program?
    a. `in`
    b. `out`
    c. `temp`
    d. `a[out]`
11. In the insertion sort, "partially sorted" means that
    a. some items are already sorted, but they may need to be moved.
    b. most items are in their final sorted positions, but a few still need to be sorted.
    c. only some of the items are sorted.
    d. group items are sorted among themselves, but items outside the group may need to be inserted in it.
12. Shifting a group of items left or right requires repeated _____.
13. In the insertion sort, after an item is inserted in the partially sorted group, it will
    a. never be moved again.
    b. never be shifted to the left.
    c. often be moved out of this group.
    d. find that its group is steadily shrinking.

14. The invariant in the insertion sort is that _____.
15. Stability might refer to
    a. items with secondary keys being excluded from a sort.
    b. keeping cities sorted by increasing population within each state, in a sort by state.
    c. keeping the same first names matched with the same last names.
    d. items keeping the same order of primary keys without regard to secondary keys.

Experiments

Carrying out these experiments will help to provide insights into the topics covered in the chapter. No programming is involved.

1. In `bubbleSort.java` (Listing 3.1) rewrite `main()` so it creates a large array and fills that array with data. You can use the following code to generate random numbers:

```
for(int j=0; j<maxSize; j++)           // fill array with
   {                                   // random numbers
   long n = (long)( java.lang.Math.random()*(maxSize-1) );
   arr.insert(n);
   }
```

   Try inserting 10,000 items. Display the data before and after the sort. You'll see that scrolling the display takes a long time. Comment out the calls to `display()` so you can see how long the sort itself takes. The time will vary on different machines. Sorting 100,000 numbers will probably take less than 30 seconds. Pick an array size that takes about this long and time it. Then use the same array size to time `selectSort.java` (Listing 3.2) and `insertSort.java` (Listing 3.3). See how the speeds of these three sorts compare.

2. Devise some code to insert data in inversely sorted order (99,999, 99,998, 99,997, ...) into `bubbleSort.java`. Use the same amount of data as in Experiment 1. See how fast the sort runs compared with the random data in Experiment 1. Repeat this experiment with `selectSort.java` and `insertSort.java`.
3. Write code to insert data in already-sorted order (0, 1, 2, ...) into `bubbleSort.java`. See how fast the sort runs compared with Experiments 1 and 2. Repeat this experiment with `selectSort.java` and `insertSort.java`.

As with the `infix.java` program (Listing 4.7), `postfix.java` doesn't check for input errors. If you type in a postfix expression that doesn't make sense, results are unpredictable.

Experiment with the program. Trying different postfix expressions and seeing how they're evaluated will give you an understanding of the process faster than reading about it.

Summary

- Stacks, queues, and priority queues are data structures usually used to simplify certain programming operations.
- In these data structures, only one data item can be accessed.
- A stack allows access to the last item inserted.
- The important stack operations are pushing (inserting) an item onto the top of the stack and popping (removing) the item that's on the top.
- A queue allows access to the first item that was inserted.
- The important queue operations are inserting an item at the rear of the queue and removing the item from the front of the queue.
- A queue can be implemented as a circular queue, which is based on an array in which the indices wrap around from the end of the array to the beginning.
- A priority queue allows access to the smallest (or sometimes the largest) item.
- The important priority queue operations are inserting an item in sorted order and removing the item with the smallest key.
- These data structures can be implemented with arrays or with other mechanisms such as linked lists.
- Ordinary arithmetic expressions are written in infix notation, so-called because the operator is written between the two operands.
- In postfix notation, the operator follows the two operands.
- Arithmetic expressions are typically evaluated by translating them to postfix notation and then evaluating the postfix expression.
- A stack is a useful tool both for translating an infix to a postfix expression and for evaluating a postfix expression.

Questions

These questions are intended as a self-test for readers. Answers may be found in Appendix C.

1. Suppose you push 10, 20, 30, and 40 onto the stack. Then you pop three items. Which one is left on the stack?
2. Which of the following is true?
   a. The pop operation on a stack is considerably simpler than the remove operation on a queue.
   b. The contents of a queue can wrap around, while those of a stack cannot.
   c. The top of a stack corresponds to the front of a queue.
   d. In both the stack and the queue, items removed in sequence are taken from increasingly high index cells in the array.
3. What do LIFO and FIFO mean?
4. True or False: A stack or a queue often serves as the underlying mechanism on which an ADT array is based.
5. Assume an array is numbered with index 0 on the left. A queue representing a line of movie-goers, with the first to arrive numbered 1, has the ticket window on the right. Then
   a. there is no numerical correspondence between the index numbers and the movie-goer numbers.
   b. the array index numbers and the movie-goer numbers increase in opposite left-right directions.
   c. the array index numbers correspond numerically to the locations in the line of movie-goers.
   d. the movie-goers and the items in the array move in the same direction.
6. As other items are inserted and removed, does a particular item in a queue move along the array from lower to higher indices, or higher to lower?
7. Suppose you insert 15, 25, 35, and 45 into a queue. Then you remove three items. Which one is left?
8. True or False: Pushing and popping items on a stack and inserting and removing items in a queue all take O(N) time.

9. A queue might be used to hold
   a. the items to be sorted in an insertion sort.
   b. reports of a variety of imminent attacks on the star ship Enterprise.
   c. keystrokes made by a computer user writing a letter.
   d. symbols in an algebraic expression being evaluated.
10. Inserting an item into a typical priority queue takes what big O time?
11. The term *priority* in a priority queue means that
   a. the highest priority items are inserted first.
   b. the programmer must prioritize access to the underlying array.
   c. the underlying array is sorted by the priority of the items.
   d. the lowest priority items are deleted first.
12. True or False: At least one of the methods in the `priorityQ.java` program (Listing 4.6) uses a linear search.
13. One difference between a priority queue and an ordered array is that
   a. the lowest-priority item cannot be extracted easily from the array as it can from the priority queue.
   b. the array must be ordered while the priority queue need not be.
   c. the highest priority item can be extracted easily from the priority queue but not from the array.
   d. All of the above.
14. Suppose you based a priority queue class on the `OrdArray` class in the `orderedArray.java` program (Listing 2.4) in Chapter 2, "Arrays." This will buy you binary search capability. If you wanted the best performance for your priority queue, would you need to modify the `OrdArray` class?
15. A priority queue might be used to hold
   a. passengers to be picked up by a taxi from different parts of the city.
   b. keystrokes made at a computer keyboard.
   c. squares on a chessboard in a game program.
   d. planets in a solar system simulation.

Summary

- Trees consist of nodes (circles) connected by edges (lines).
- The root is the topmost node in a tree; it has no parent.
- In a binary tree, a node has at most two children.
- In a binary search tree, all the nodes that are left descendants of node A have key values less than A; all the nodes that are A's right descendants have key values greater than (or equal to) A.
- Trees perform searches, insertions, and deletions in O(log N) time.
- Nodes represent the data objects being stored in the tree.
- Edges are most commonly represented in a program by references to a node's children (and sometimes to its parent).
- Traversing a tree means visiting all its nodes in some order.
- The simplest traversals are preorder, inorder, and postorder.
- An unbalanced tree is one whose root has many more left descendents than right descendants, or vice versa.
- Searching for a node involves comparing the value to be found with the key value of a node, and going to that node's left child if the key search value is less, or to the node's right child if the search value is greater.
- Insertion involves finding the place to insert the new node and then changing a child field in its new parent to refer to it.
- An inorder traversal visits nodes in order of ascending keys.
- Preorder and postorder traversals are useful for parsing algebraic expressions.
- When a node has no children, it can be deleted by setting the child field in its parent to `null`.
- When a node has one child, it can be deleted by setting the child field in its parent to point to its child.
- When a node has two children, it can be deleted by replacing it with its successor.
- The successor to a node A can be found by finding the minimum node in the subtree whose root is A's right child.
- In a deletion of a node with two children, different situations arise, depending on whether the successor is the right child of the node to be deleted or one of the right child's left descendants.

- Nodes with duplicate key values may cause trouble in arrays because only the first one can be found in a search.
- Trees can be represented in the computer's memory as an array, although the reference-based approach is more common.
- A Huffman tree is a binary tree (but not a search tree) used in a data-compression algorithm called Huffman Coding.
- In the Huffman code the characters that appear most frequently are coded with the fewest bits, and those that appear rarely are coded with the most bits.

Questions

These questions are intended as a self-test for readers. Answers may be found in Appendix C.

1. Insertion and deletion in a tree require what big O time?
2. A binary tree is a search tree if
   a. every non-leaf node has children whose key values are less than (or equal to) the parent.
   b. every left child has a key less than the parent and every right child has a key greater than (or equal to) the parent.
   c. in the path from the root to every leaf node, the key of each node is greater than (or equal to) the key of its parent.
   d. a node can have a maximum of two children.
3. True or False: Not all trees are binary trees.
4. In a complete binary tree with 20 nodes, and the root considered to be at level 0, how many nodes are there at level 4?
5. A subtree of a binary tree always has
   a. a root that is a child of the main tree's root.
   b. a root unconnected to the main tree's root.
   c. fewer nodes than the main tree.
   d. a sibling with the same number of nodes.
6. In the Java code for a tree, the _____ and the _____ are generally separate classes.

7. Finding a node in a binary search tree involves going from node to node, asking
   a. how big the node's key is in relation to the search key.
   b. how big the node's key is compared to its right or left children.
   c. what leaf node we want to reach.
   d. what level we are on.
8. An unbalanced tree is one
   a. in which most of the keys have values greater than the average.
   b. whose behavior is unpredictable.
   c. in which the root or some other node has many more left children than right children, or vice versa.
   d. that is shaped like an umbrella.
9. Inserting a node starts with the same steps as _____ a node.
10. Suppose a node A has a successor node S. Then S must have a key that is larger than _____ but smaller than or equal to _____.
11. In a binary tree used to represent a mathematical expression, which of the following is not true?
    a. Both children of an operator node must be operands.
    b. Following a postorder traversal, no parentheses need to be added.
    c. Following an inorder traversal, parentheses must be added.
    d. In pre-order traversal a node is visited before either of its children.
12. If a tree is represented by an array, the right child of a node at index n has an index of _____ .
13. True or False: Deleting a node with one child from a binary search tree involves finding that node's successor.
14. A Huffman tree is typically used to _____ text.
15. Which of the following is not true about a Huffman tree?
    a. The most frequently used characters always appear near the top of the tree.
    b. Normally, decoding a message involves repeatedly following a path from the root to a leaf.

c. In coding a character you typically start at a leaf and work upward.

d. The tree can be generated by removal and insertion operations on a priority queue.

Experiments

Carrying out these experiments will help to provide insights into the topics covered in the chapter. No programming is involved.

1. Use the Binary Tree Workshop applet to create 20 trees. What percentage would you say are seriously unbalanced?
2. Create a UML activity diagram (or flowchart, for you old-timers) of the various possibilities when deleting a node from a binary search tree. It should detail the three cases described in the text. Include the variations for left and right children and special cases like deletion of the root. For example, there are two possibilities for case 1 (left and right children). Boxes at the end of each path should describe how to do the deletion in that situation.
3. Use the Binary Tree Workshop applet to delete a node in every possible situation.

Programming Projects

Writing programs to solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied. (As noted in the Introduction, qualified instructors may obtain completed solutions to the Programming Projects on the publisher's Web site.)

8.1 Start with the `tree.java` program (Listing 8.1) and modify it to create a binary tree from a string of letters (like A, B, and so on) entered by the user. Each letter will be displayed in its own node. Construct the tree so that all the nodes that contain letters are leaves. Parent nodes can contain some non-letter symbol like +. Make sure that every parent node has exactly two children. Don't worry if the tree is unbalanced. Note that this will not be a search tree; there's no quick way to find a given node. You may end up with something like this: