15 DATA STRUCTURES

Information often resides in data, but data is not always the same as information: certainly data is not the same as *useful* information. Nevertheless, often data—collected in experiments, gleaned from transactions, offered by individuals in questionnaires, or downloaded from other sources—does contain information of great value. The challenge is to extract the useful information from all that is available.

Data sources can become enormous and unwieldy. The first step toward transforming data into useful information is to organize the data so that it can be readily accessed, searched, manipulated, updated, simplified, and sometimes generalized. Data structures provide the basis of such work. Many of these structures were originated to facilitate the programming of complex data manipulations, but the principles underlying data structures are useful more generally, for constructing databases or data warehouses, for building efficient data retrieval systems, and ultimately for assisting with the processes of extracting information from data.

Basic data structures include lists, arrays, and trees. From these basics, more complex structures can be built.

We shall find that these basic structures often are used in concert and that one may be converted to another. For example, an effective way to sort a list is to transform it, either explicitly or implicitly, into a tree; then sort the tree and transform the results back to a list. In the process, the tree might be represented as an array. Data structures are fluid and adaptable, and come in numerous variations.

15.1 Lists

An obvious way to store data is sequentially, as a list, one item after another. Employee names might constitute a list, for example.

Abstractly, a **list** is an ordered set of objects (or items) of a given type. A list of length n can be represented as $(a_1, a_2, \ldots, a_n)$ where the a_i's are the objects. The

position of an object is its index i in the list. The objects themselves may be numbers, book catalog records, patient health profiles, gene descriptions, or names of state capitals. The objects in a list need not be numeric or alphabetic. A row of automobiles in a parking lot can be regarded as a list.

The objects in a list may be multidimensional. For example, items in a library catalog may include book title, author, Library of Congress catalog number, date of publication, publisher, date of acquisition, and availability status. Such an object is termed a **record** with individual portions of the record being **fields**.

For a list to be most useful, it must be possible to carry out certain basic operations on the list. For example, one may wish to **insert** additional items in the list, **delete** some items, **locate** an item or items that meet certain criteria, or move to the **next** or **previous** item. The ease with which such basic operations can be performed may depend on how the list is represented.

Two of the most important operations on a list are **sorting** and **searching**. Sorting is the process of arranging the items according to an ordering of the items. For example, it may be desired to sort items numerically or alphabetically. If an item is a record with several fields, the ordering is usually carried out with respect to a single field termed the **key** that uniquely identifies the entry in the list. Searching is the process of finding an item that meets a specific criterion, or concluding that no such item is in the list. Data structures facilitate sorting and searching, carried out by the basic operations mentioned in the previous paragraph.

Lists Represented by Arrays

Normally one thinks of a list as an array, the items being placed at successive locations. For example, the items might be written on successive rows of a sheet of lined paper or at successive locations in computer storage.

As a physical example, imagine the parking lot of a rental car agency located at an airport. The parking lot spaces are numbered consecutively. The rental cars also have identification numbers that serve as their keys. Only a small fraction of the total inventory of cars is in the lot at any one time.

Suppose the agency keeps track of available cars by storing them in numerical order in parking spaces, with the car of lowest identification number going in space number 1. When a car is returned to the lot, it must be put in its proper place between cars of lower and higher identification numbers. To make room for this car, all cars with higher identification numbers must be moved one space down the list to provide an opening (figure 15.1)

Likewise, when a car is rented and leaves the lot, all cars beyond it on the list must be moved up one space to close the gap. Clearly this is not an efficient way to store cars.

Inserting or deleting an element in a list that is implemented this way requires, on average, $n/2$ movements of items in the list, and even when the items are entries in a computer, this can be time consuming for large lists. On the other hand, if a particular item is found, finding the next or previous item is simple. One simply goes to the next or previous location.

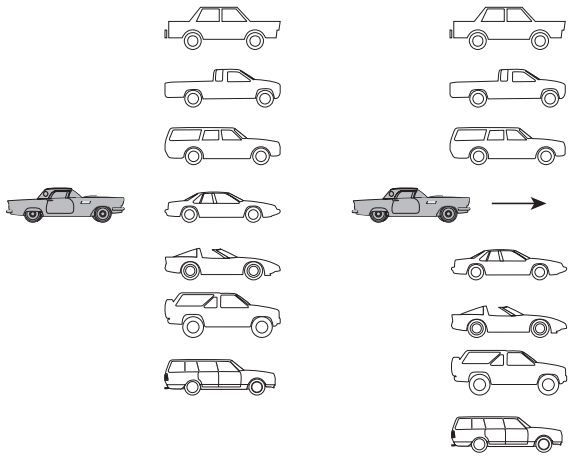


FIGURE 15.1 To insert a new vehicle in the parking lot, several others must be moved.

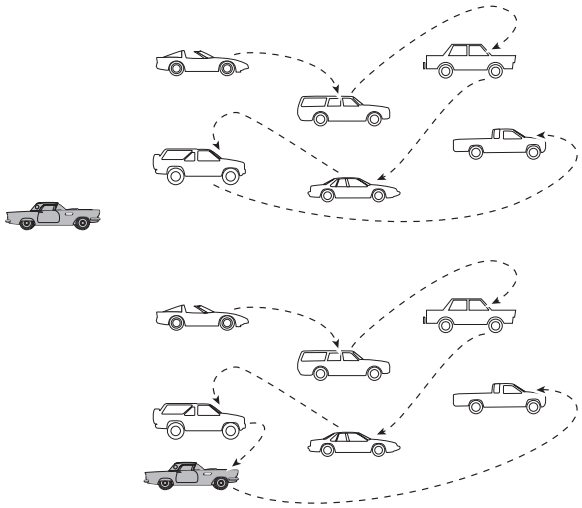


FIGURE 15.2 Each pointer gives the location of the next car. When a new car arrives, it can be inserted by updating the pointers, as shown in the bottom part of the figure when the entering vehicle is sixth on the list.

Linked Lists

The items of a list can be stored in arbitrary locations, provided that a record of their locations is kept. The simplest way to do this is with a **linked list**. In such a list the items are stored arbitrarily in the spaces available, but accompanying each item is a **pointer** to the location of the next item on the list.

In the parking lot example, cars can be stored in arbitrary spaces if on each car there is a pointer sign giving the location of the next car on the list (figure 15.2). To

find, say the fifth car (the car fifth on the list of available cars sorted by identification number), you start at the first car, read its pointer telling where the second car is, go to the second car and learn where the third car is, and so forth, until you reach car 5.

When a new car arrives, it can be parked in any available space. The pointer of the car preceding it in the sorted list is then changed to indicate the location of this new car, and the new car is given the pointer that the preceding car had, pointing to the next car. Hence, by changing one pointer and adding another, the list is updated to include the new car, and the ordering is preserved.

This procedure works identically for lists stored in a computer. Items can occupy arbitrary memory locations, with each item appended with a pointer indicating the location of the next item. By following the pointers, one can traverse the entire list in order.

It is also simple to remove an item. To do so, it is only necessary to change the pointer of the preceding object to be the pointer of the object being removed. This deletes the object from the list even if the object is not physically removed. With no pointer pointing to it, the object is essentially nonexistent.

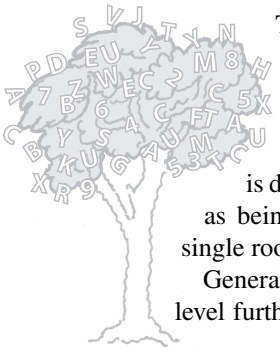
Although it is easy to move forward through a linked list, it is not easy to move backward. The pointers only point forward. This difficulty is solved by a **doubly linked list**, in which each object is accompanied by two pointers: one pointing to the successor item and the other to the predecessor.

Special Lists

Lists often have special uses that dictate a particular form of updating. One of these is the **stack** in which objects are entered one by one at the top of the list, each new addition causing the others to be pushed down one place. Objects are removed from the top as well, causing the other objects to move upward. The scheme is termed FILO, for "First In, Last Out." For example, if you make changes in a word processor, these changes are saved one at a time in a stack. Then if you decide to undo a change, the first change restored is the last that was made, since it comes off the top of the stack.

The sister to the stack is the **queue** in which objects are entered one by one at the top of the list, and removed from the bottom of the list. This is termed FIFO, for "First In, First Out." It simulates a queue of people waiting for service at the bank, or program steps waiting to be executed. The first in line is the first served.

15.2 Trees



Trees are valuable structures used in formal and informal representation, analysis, and manipulation of data. Trees represent structures such as organizational charts, genealogy (family trees), contest standings (as in a tennis ladder), and various other hierarchical structures.

In fact, a tree is basically a hierarchical arrangement of **nodes**. One node is designated as the **root**, and (although it is called a root) it is usually visualized as being at the top of the hierarchy. The simplest nonempty trees consist of a single root and no other nodes.

Generally, a node has a number of **children** nodes directly connected to it but one level further down the hierarchy. Every node i, except the root, has a single **parent**,

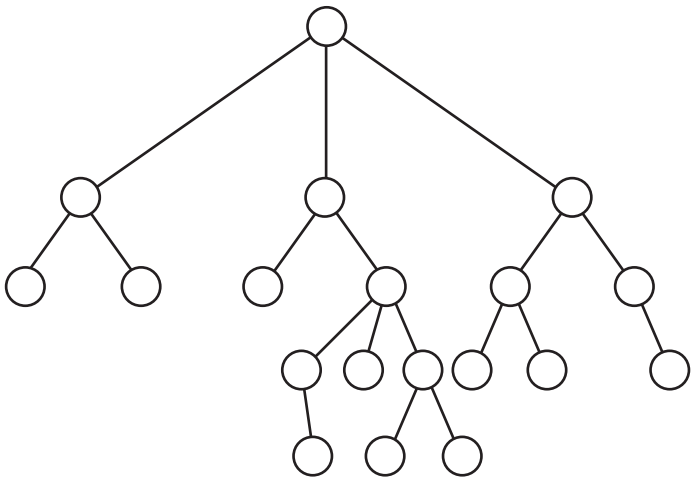


FIGURE 15.3 A tree. The root has three children, each of which has two children. There are a total of ten leaf nodes.

such that i is a child of this parent. A node without children is termed a **leaf**. The parent–child relation is described pictorially by lines connecting the corresponding nodes as shown in figure 15.3.

In a **binary tree** every node has at most two children. It is conventional to refer to a child in a binary tree as either a **left child** or a **right child**, where naturally the left child is the one located below and to the left of the parent and the right child is located below and to the right of the parent. Clearly, in a binary tree each node may have either no children, a left child, a right child, or both a left and right child.

Ordered Trees

It is often convenient to number the nodes systematically. In one simple method, the root is assigned number 1. Then at the next level numbers are assigned sequentially, starting from the left and working across to the right. This is continued through successive levels. The version of the tree of figure 15.3 numbered this way is shown in figure 15.4.

Other numbering strategies, useful in certain computational procedures, are discussed in the next section in the context of transversal.

Representation of Trees

One of the simplest ways to represent a tree is with a set of pointers that point down the tree. The position of the root is specified first. At every node, pointers to the locations of each of its children are listed. It is then possible to move from the root

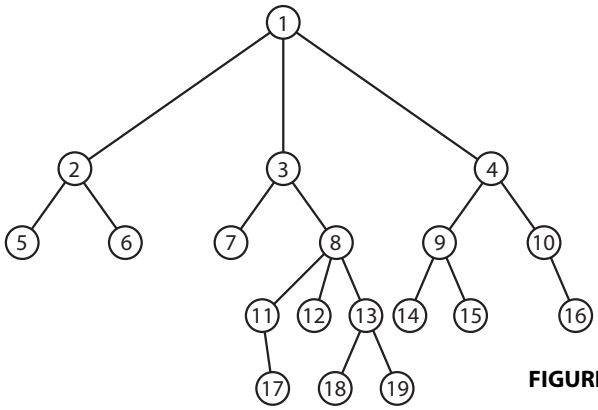
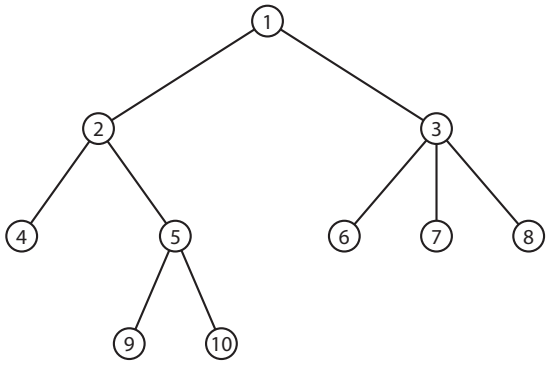


FIGURE 15.4 An ordered version of the tree in figure 15.3.



1	xxx	0
2	xxx	1
3	xxx	1
4	xxx	2
5	xxx	2
6	xxx	3
7	xxx	3
8	xxx	3
9	xxx	5
10	xxx	5

FIGURE 15.5 A tree and an array representation. The record of each node is placed in the array, followed by a pointer to the parent of the node.

1	xxx	2	3	
2	xxx	4	5	
3	xxx	6	7	8
4	xxx	N		
5	xxx	9	10	
6	xxx	N		
7	xxx	N		
8	xxx	N		
9	xxx	N		
10	xxx	N		

FIGURE 15.6 A list representation of the tree of fig. 15.5. Each node is represented by its contents and a list of its children. The symbol N denotes that the node has no children and is therefore a leaf node.

down a variety of paths, following one of the pointers at each node encountered. The entire structure of the tree is embodied in this pointer structure. Alternatively, each node may contain pointers to its parents. This too is sufficient to describe the tree.

A tree can be represented concretely as an array, in any one of several ways. Figure 15.5 shows a tree of record locations and the tree's representation as an array, which contains the records as well as the node numbers and pointer to the parent. The entire tree structure is embodied in this array.

A tree can also be represented as a series of lists. For example, a tree can be described by listing the children of each node. Figure 15.6 shows this representation for the tree of figure 15.5.

15.3 Traversal of Trees

Frequently it is desirable to traverse a tree, visiting every node to find one that satisfies a search criterion or to modify the contents of the nodes. If the tree is represented only by its parent–child relations, such a traverse must move systematically node by node: from parent to child, or from child to parent.

As an analogue, imagine that the connecting lines of the tree are pathways. To traverse the tree, one must walk on the paths in a route that visits every node. Some duplication is likely to be necessary—some nodes will be visited more than once—but we seek a systematic strategy. Such a strategy is illustrated in figure 15.7. The route indicated by the dotted line goes through every node, and it stays on the connecting paths. From the figure it is clear that if a complete cycle from the root back to the root is made, each leaf node will be visited only once, but others will be visited at least twice.

This traversal route can be used to order the nodes in one of several ways. The most direct numbering system is termed **preorder**. In this method, the nodes are numbered sequentially as they are passed the first time in the traversal that cycles the tree in the counterclockwise direction. The resulting node ordering for the tree of figure 15.7 is given by the numbers indicated in the nodes.

A special ordering for binary trees is termed **inorder**. In this method, the tree is again traversed according to the counterclockwise cycle, just as before. The leaf nodes are numbered the first (and only) time they are passed. However, other nodes are numbered the *second* time they are passed. The root, for example, will usually not be assigned number 1.

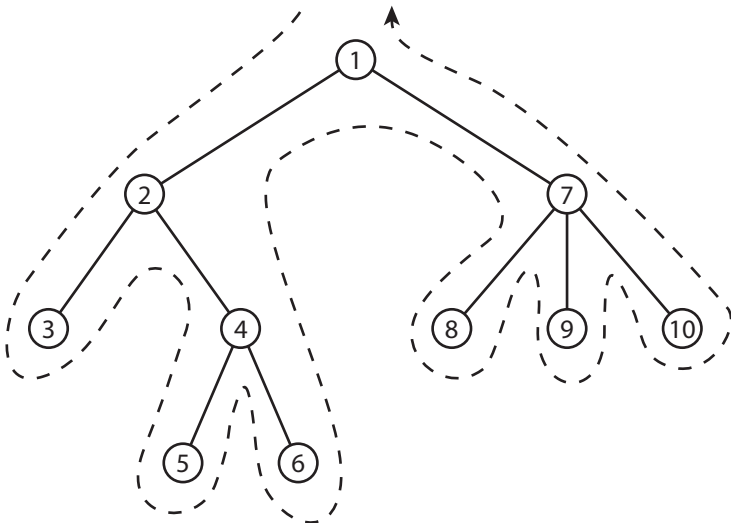


FIGURE 15.7 Traversal of a tree. A counterclockwise cycle defines a traversal that goes through every node at least once. The numbers in the nodes in this tree are those defined by the preorder method of ordering.

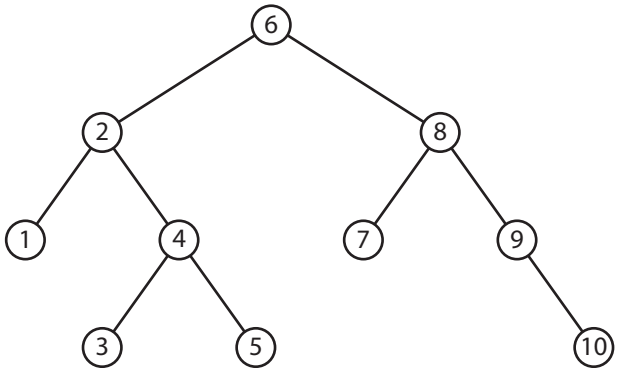


FIGURE 15.8 Inorder of a binary tree. Leaf nodes are numbered the first (and only) time they are passed. Other nodes are numbered the second time they are passed. A node without a left child is numbered before its right child.

There is a special case of this method that must be treated carefully. If a node has a single child and that child is a right child rather than a left child, then an artificial left child must be assigned to that parent. This artificial node does not get a number, but its existence insures that the parent will be visited twice before the traverse reaches the single right child. For example, in a tree consisting of a root and a single right child, the root would be numbered 1, because the root would be visited twice (the second time being after the visit to the artificial left child) before the right child is reached.

Another way to characterize the inorder order is that it numbers trees in LNR order; that is, in Left, Node, Right order. A left child is numbered first, followed by the node, followed by the right child. If there is no left child, then the node is numbered first.[1]

An example of a binary tree in inorder order is shown in figure 15.8. The numbers can be verified by making a counterclockwise cycle of the tree. Notice that node 9 has a single child, which is a right child. Hence, following the LNR rule, node 9 is numbered before node 10. This is the same ordering as would be obtained by appending an artificial left child to node 9, but not assigning it a number as the tree was traversed.

15.4 Binary Search Trees (BSTs)

The **binary search tree** is one of the most powerful of the basic data structures. Such trees lead to simple, yet highly efficient representations for searching and sorting data. It employs the inorder method of ordering.

A binary search tree is applicable when the objects to be processed possess key values that can be ranked (such as alphabetically or numerically). Construction of the

[1]The process can be described recursively by defining the general step at a node: visit left child and carry out the process, then number the current node, then visit right child and carry out the process. Start at the root.

tree and the search through it are governed by the order inherent in the key values. A binary search tree is built in such a way that an inorder transversal leads to a sorted ordering.

The process is simple. The first object becomes the root of the tree. The next object is compared with the root and becomes a left or right child of the root depending on whether its key value is less than the root or greater than (or equal to) the root. Subsequent objects are entered by comparing them first with the root, determining whether to go left or right, then continuing down the tree, making similar comparisons at every node encountered, until it becomes either the first or second child of a parent.

An example should clarify the procedure. Suppose we wish to alphabetize the following names: **Linda**, **Joanne**, **Carl**, **Robert**, **Jenna**, **Steve**, **Marion**, **Nancy**, **Ian**, **Jill**, **Susan**, **Fred**. The tree is built by taking the first name, **Linda**, as the root. The next name, **Joanne**, is then compared with the root. If it is lower in the alphabet, it becomes the left child, otherwise the right child. Thus **Joanne** becomes the left child of **Linda**. Then **Carl** goes left of **Linda** and left of **Joanne**. The complete binary search tree is shown in figure 15.9.

In the example, artificial nodes are adjoined as left children of **Carl**, **Marion**, and **Steve** to remind us how to define the inorder. Indeed, **Carl** is the first item in the inorder. The other items can be quickly ordered by traversing through the tree counterclockwise, leading to **Carl**, **Fred**, **Ian**, **Jenna**, **Jill**, **Joanne**, **Linda**, **Marion**, **Nancy**, **Robert**, **Steve**, **Susan**.

To search for a name, say **Nancy**, it is only necessary to follow the path downward. **Nancy** must be to the right of **Linda**, to the left of **Robert**, and to the right of **Marion**. Bingo! There she is. Alternatively, if a search is instituted for a name such as **Ralph** that is not on the list, one will progress down to a leaf node with no place farther to go, and hence conclude that **Ralph** is not on the list.

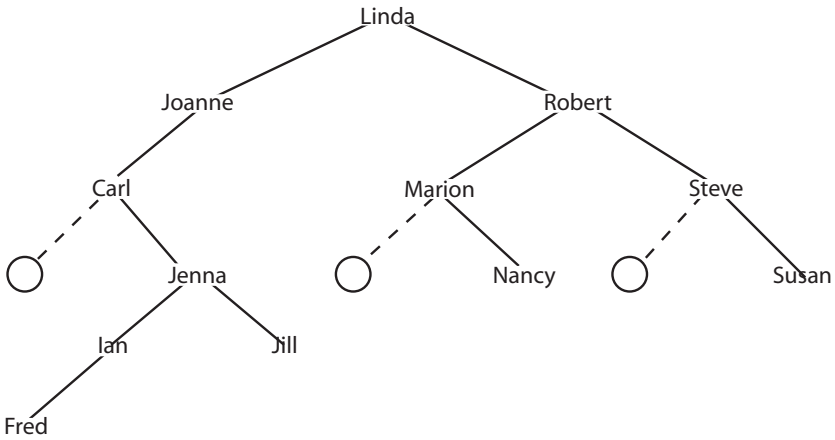


FIGURE 15.9 A binary search tree of names. The tree automatically puts the names in inorder as it is constructed.

Another example of a binary search tree is the tree of figure 15.8. It is the BST that would result from construction based on the unordered sequence 6, 2, 8, 4, 1, 3, 5, 9, 7, 10.

Binary search trees are used in many practical applications, such as airline reservation systems where individuals' names are entered sequentially as they book flights.

Average Path Length

Searching for an object in a BST entails traveling along the unique path from the root to the object. The total search time is proportional to the total number of comparisons required, and hence proportional to the length of the path.

The length of such a path in a BST with n nodes can vary widely, depending on the particular tree. The best case is when the tree is **balanced**, with each node having two children. In this case the total number of nodes n is of the form $n = 2^k - 1$ for some integer $k \geq 1$. The maximum length of a path, in terms of the number of nodes visited, is then $L_{\max} = k$; or in terms of n, $L_{\max} = \log_2 (n + 1)$.

The worst case is when each node, except the last (which is a leaf node), has only a single child. The length of a path from top to bottom is n. In general, therefore, the maximum path length varies between $\log (n + 1)$ and n.

TABLE 15.1
Worst and Best Path Lengths to a Leaf. In the worst case the path length of a tree with n nodes is n. For a balanced tree the path length is $\log(n+1)$.

n	$\log (n + 1)$
7	3
127	7
1,023	10
16,383	14
131,071	17
1,048,575	20
16,777,215	24
134,217,727	27
1,073,741,823	30

As shown in table 15.1, there is a tremendous difference between these two bounds for even modest values of n. For a tree of about 1 billion nodes, the length from the root to a leaf node is at most 30 if the tree is balanced. On the other hand, if the tree is completely unbalanced, the length is a billion.

It is of great importance to know what length might be expected in actual application. For $n = 1$ billion, is the number of required comparisons for a search closer to 1 billion or to 30?

This question can be addressed by considering a binary search tree with n objects, under the assumption that the ordering of the keys is initially random. Let $P(n)$ be the average path length to a random object where now the object is not necessarily at a leaf node. The following important result characterizes $P(n)$.

Theorem 15.1. The function $P(n)$ satisfies

$$P(n) \leq 1 + 2 \ln n \leq 1 + 1.386 \log n.$$

Proof: Define $Q(n)$ as the expected total number of node visits required to construct the entire binary search tree. The average number of visits to a particular node $P(n)$ is then $Q(n)$ divided by n.

The nodes are referred to by node numbers 1 though n, which are taken to be identical to the ranking of their keys. Hence the proper ordering is 1 through n. A step occurs when two elements are compared. Two elements i and j are compared at most once. We shall find the probability that i and j are compared, and for this purpose it can be assumed that $i < j$.

Consider the chain of values $i, i + 1, i + 2, \ldots, j$ that has $L = j + 1 - i$ members. The elements arrive randomly for placement. If any element k with $i < k < j$ arrives

before i or j, then i will never be compared with j, for i will be sent left of k and j will be sent right. Hence, i is compared with j only if i or j occurs before all other elements in the chain. The probability of i or j occurring first among the L elements is $2/L$.

The expected total number of comparisons is therefore $n_L \times 2/L$, where n_L is the number of chains of length L. This number is $N_L = n + 1 - L$, for $L = 2, 3, \ldots, n$. In addition, each node is considered to visit itself. Hence the total number of visits is

$$Q(n) = n + \sum_{L=2}^{n} n_L \frac{2}{L} = n + \sum_{L=2}^{n} 2\frac{(n+1-L)}{L}.$$

Using the standard approximation to the harmonic sum of $1/L$'s (see exercise 2),

$$\sum_{L=2}^{n} \frac{1}{L} \leq \int_{x=1}^{n} \frac{1}{x} \mathrm{d}x = \ln n, \tag{15.1}$$

gives $P(n) = Q(n)/n$ as

$$\begin{aligned} P(n) &= 1 + \sum_{L=2}^{n} 2\frac{n+1-L}{nL} \\ &= 1 - 2\left(1 - \frac{1}{n}\right) + 2\left(1 + \frac{1}{n}\right) \sum_{L=2}^{n} \frac{1}{L} \\ &\leq 1 + 2\ln n + 2\left[\frac{1}{n}(1 + \ln n) - 1\right]. \end{aligned} \tag{15.2}$$

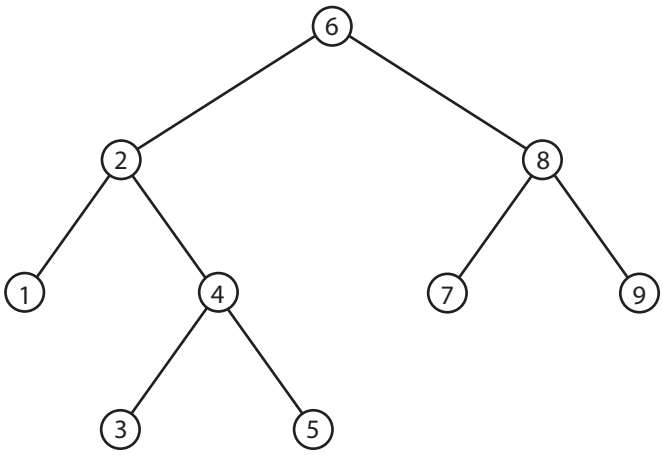
The term in brackets is always less than or equal to zero. Hence

$$P(n) \leq 1 + 2\ln n \leq 1 + 1.386 \log n. \blacksquare$$

The actual values of $P(n)$ are extremely close to the bound given by the theorem. If the bracketed term in equation (15.2) is included, a tighter upper bound $P^u(n)$ is obtained that is at most two steps less than $1 + 2\ln n$. A lower bound $P^l(n)$ can be constructed, by using a lower bound on the sum in equation (15.1) (see exercise 3) that is only about two steps less than $P^u(n)$. Hence the actual value of $P(n)$ is within two steps of either of these strong bounds. For example, the value of $P(1 \text{ billion})$ is between the bounds of 39.0602373 and 40.44653173.

Measures of efficiency as a function of the problem size n usually focus on the performance for large n. Typically, this asymptotic behavior is expressed in "big O" notation. A statement that the number of steps is $T(n) = \mathrm{O}(n^k)$ means that there is a constant $c \geq 0$ such that $T(n) \leq cn^k$ for sufficiently large n. Thus if $T(n) = 47 + 2n + 19n^2$, then $T(n)$ is $\mathrm{O}(n^2)$.

A stronger notion is defined by Θ notation. A statement that $T(n) = \Theta(n^k)$ means that there are positive constants c_1, c_2 such that $c_1 n^k \leq T(n) \leq c_2 n^k$ for sufficiently large n. Hence $\Theta(n^k)$ implies $\mathrm{O}(n^k)$, but the reverse implication is not necessarily true. With this notation, the path length of BSTs is at worst $\Theta(n)$, but on average $\Theta(\log n)$.



N	P	L	R
6	-	2	8
2	6	1	4
3	4	-	-
4	2	3	5
7	8	-	-
8	6	7	9
9	8	-	-

FIGURE 15.10 A binary tree and its representation as a table. The table shows node, parent, left child, right child.

Representation

A table representation of a binary tree listing the left and right children of various nodes facilitates rapid search through the tree. An example is shown in figure 15.10. To search for node 3, for instance, one begins at the root 6 and moves to the left child 2, then to the right child 4, then to the left child to arrive at 3. The parent pointers are not needed for this type of search, but they are useful when traversing a tree. For instance, after arriving at node 3 in the figure, which is seen to be a leaf because it has no children, the pointer to the parent makes it possible to move back up to node 4.

15.5 Partially Ordered Trees

A **partially ordered tree** is a binary tree that is balanced as much as possible and has all of its leaf nodes at the lowest level as far to the left as possible. Furthermore, the key value of any node is less than or equal to that of its children.

The first requirement implies that if there is a total of h levels, then the $(h-1)$-th level is full with 2^{h-1} nodes, and the h-th level has all of its nodes to the left. The second requirement means that as one moves down the tree along any path, the key value never decreases. An example of a partially ordered tree is shown in figure 15.11.

Partially ordered trees are sometimes used to represent **priority queues**, ordering the service of various customers or jobs. The first customer in the queue is represented by the root. When served, that node is eliminated and the tree is reconfigured to a new priority queue.

To reorder the tree when the root is eliminated, the root is replaced by the node in the tree at the lowest level and at the rightmost position. The tree is then still balanced as much as possible but with one less node than before at the lowest level. To restore the partial order, the new root is pushed down the tree, exchanging it with its child of smallest key value until its key value is no smaller than that of either of its children

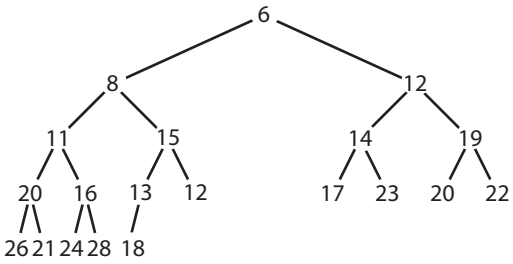


FIGURE 15.11 A partially ordered tree. Each node has a lower key value than its children, and the tree is balanced as much as possible, with all levels except the last full, and all nodes in the bottom level located to the left as far as possible.

or until it becomes a leaf node. Figure 15.12 shows the process of restoring the tree of figure 15.11 after the root node has been eliminated and replaced by the rightmost node at the bottom level.

The importance of partially ordered trees is derived from the efficiency of the restoration (push down) process. The maximum number of necessary exchanges is equal to the depth of the tree. This number is equal to at most $\log(n+1)$. Hence, the restoration process is a $O(\log n)$ process. We will later see how this can be used to advantage when sorting large lists.

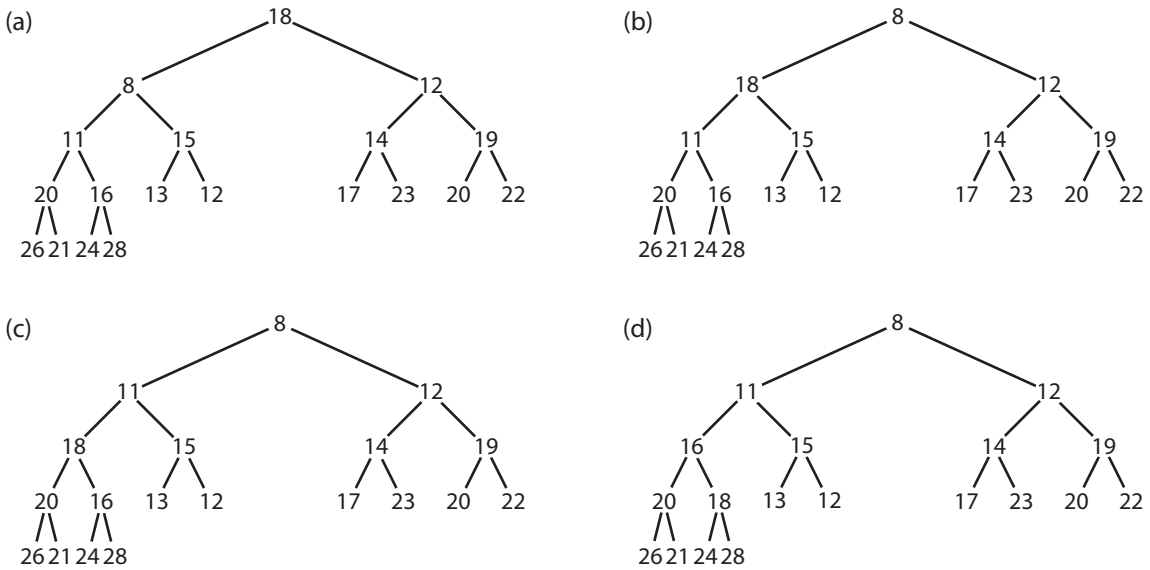


FIGURE 15.12 Pushing down a node. From figure 15.11 with the original root 6 dropped, the new root 18 is shown in (a). This root is exchanged with its smallest child 8 in (b). Then 18 is further exchanged with its new smallest child 11 in (c). Finally, 18 is exchanged with the new smallest child 16 as shown in (d). No further exchanges are necessary, and the tree is again partially ordered.

Heaps

Another advantage of partially ordered trees is that they can be stored efficiently in array form. This feature depends only on the balanced nature of the tree rather than its order, but the term **heap** usually refers to the partially ordered version.

Generally, the nodes of a partially ordered tree are numbered consecutively across each level. This numbering is independent of the key value. The root is number 1, its left child is 2, and this level is numbered up to 4. Because the tree is balanced as much as possible, the children of any node, say number i, are at node numbers $2i$ and $2i + 1$. Hence it is easy to move through the tree in array form. Suppose the tree of figure 15.11 is numbered that way. Then it can be represented by the following array.

node number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	15	16	17	18	19
key value	6	8	12	11	15	14	19	20	16	13	12	17	23	20	22	26	21	24	28	18

No pointers need be appended to the list, since the children of node i are located systematically at $2i$ and $2i + 1$. Note, for example, that the children of node 5 (with key value 15) are nodes $2 \cdot 5 = 10$ and $2 \cdot 5 + 1 = 11$ (with key values 13 and 12). Manipulations such as the push-down process can be carried out directly on the array representation of the tree.

15.6 Tries*

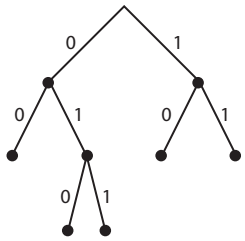


FIGURE 15.13 Codeword trie. Code words are read by beginning at the root and moving down a path to a leaf.

The term[2] **trie** is derived from retrieval and refers to a special type of tree useful for representing strings of data, such as words, codes, or numbers of several digits. The branches (or the nodes) of a trie are labeled with symbols in such a way that movement down the tree along a path from root to leaf defines an acceptable string. Tries were used in chapters 2 and 3 to study codes. Figure 15.13 is a trie representation of a Huffman code with codewords 00, 010, 011, 10, 11. The codewords are found by following the branches from the root to a leaf. Such a trie is a convenient way to verify that no codeword is a preface to another, for a preface would be found before reaching a leaf. Word tries are also used as dictionaries in some spell-checking programs.

In general, a valid string in the trie may in fact be a preface to other valid strings. THE is a preface to THESE, for instance. This situation is handled in a trie by introducing a special symbol, such as Δ, to indicate the end of a string. Figure 15.14 shows a partial trie for common English words. Note that there are three instances of the Δ symbol to signal that THE, TO, and TON are valid words even though they are prefaces to longer words.

When used as a dictionary words are checked by tracing a path down the trie, following the sequence of letters in a word being tested. If the word is in the trie, the path will end at a leaf node or a Δ. If a word is not in the trie, its path will reach a point where there is no appropriate branch or it will reach a leaf node before the word is complete.

[2]Tries are alternatively termed **digital search trees**.

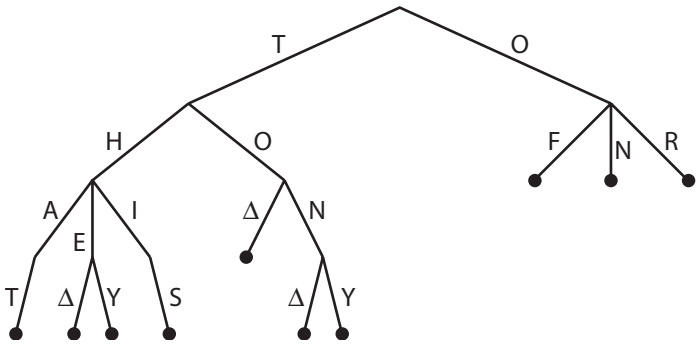


FIGURE 15.14 Word trie. Words are found by moving down a path. The Δ symbol indicates the end of an acceptable word. For example, THE and THEY are valid words in this tree.

15.7 Basic Sorting Algorithms

One of the most important applications of the data structures studied in this chapter is to the sorting of lists. Sorting may seem to be a trivial or routine operation, but sorting is an integral component of many sophisticated data analysis procedures, and hence can be regarded as fundamental to the extraction of information from data. As much as 25 percent of computer time worldwide is devoted to sorting and searching. Improvements in sorting efficiency can accordingly pay large dividends. Good sorting methods are concrete illustrations of the importance and power of data structure theory.

Sorting is, of course, the ordering of a list of objects, with the order determined by a key. A sort can be made relative to numerical or alphabetical order, by date, by string length, or by any other key quantity that can be ordered. Usually ties are allowed, in which case the tied objects are placed together in the sorted list.

Two of the simplest sorting methods are presented in this section. Although they are useful only for relatively short lists, they are a preview of, and provide motivation for, the more efficient methods discussed in the next two sections.

Bubble Sort

The name **bubble sort** expresses the view that this sorting process bubbles up low-valued key items, floating them over the higher-valued key items.

Suppose there are n items in a list, and imagine that they are listed vertically. Suppose also that we want to sort this list according to key value, with the item with lowest key value at the top. To begin the process, the bottom two items, in positions $n-1$ and n, are compared. If the bottom item has a key value less than the one above it, the two are exchanged; otherwise not. The two bottom items are then in order. The process then moves up one step, comparing the items at positions $n-2$ and $n-1$. They are exchanged if necessary to bring these two into proper order. This pair-wise comparison is continued up to the top of the list.

Math	**Art**	**Art**	**Art**	**Art**	**Art**
English	Math	**English**	**English**	**English**	**English**
History	English	Math	**Gymnastics**	**Gymnastics**	**Gymnastics**
Gymnastics	History	Gymnastics	Math	**History**	**History**
Language	Gymnastics	History	History	Math	**Language**
Art	Language	Language	Language	Language	Math

FIGURE 15.15 Bubble sort. Each successive column shows the result of an additional full pass. The boldface items have completed their upward bubbling.

After one complete pass, the lowest-valued item will be at the top, because once it is encountered in a comparison, it will be the lowest item in all subsequent comparisons and will thus be exchanged over and over again, bubbling up to the top. Another pass will bubble the second lowest item up to the second position.

Additional passes are made, although the k-th pass need not include the top $k-1$ items since they are already in proper order. All items will be properly ordered after at most $n-1$ passes. An example is shown in figure 15.15, where class titles are sorted alphabetically.

Measures of the efficiency of bubble sort focus on the number of comparisons or exchanges required. The best situation is when the list is initially in proper order, in which case $n-1$ comparisons and no exchanges are needed. The worst situation is when the list is initially in reverse order. Then the comparisons and exchanges in the first pass are both $n-1$ in number. Likewise, the k-th pass requires $n-k$ comparisons and exchanges. The total is $\sum_{k=1}^{n-1}(n-k) = n(n-1)/2$ comparisons and exchanges. Therefore in the best case, bubble sort is a $\Theta(n)$ process, while in the worst case it is a $\Theta(n^2)$ process.

The average number of exchanges required in bubble sort can be deduced from the following clever observation. Consider a list L with n items ordered randomly, and consider the list $\overline{L}$, which is ordered in the exact reverse of L. Suppose bubble sort is applied to each list separately. Two items i and j will be out of order in exactly one of the lists, and so at some point they will be exchanged in that list. Since this applies to any two items, there must be exactly one exchange, in either L or $\overline{L}$, for every pair of items. Since there are exactly $n(n-1)/2$ distinct pairs, sorting both L and $\overline{L}$ requires $n(n-1)/2$ exchanges. This means that, on average, $n(n-1)/4$ exchanges are required for a list of length n. Thus bubble sort is, on average, a $\Theta(n^2)$ process.[3] It can be shown that the average number of comparisons is also $\Theta(n^2)$.

Insertion Sort

In **insertion sort** items are inserted, one by one, into an incomplete list that is always properly sorted and that grows to full size. The result is the desired sorted list.

[3]It is assumed that all items have different key values.

Math	**English**	**English**	**English**	**English**	**Art**
English	**Math**	**History**	**Gymnastics**	**Gymnastics**	**English**
History		**Math**	**History**	**History**	**Gymnastics**
Gymnastics			**Math**	**Language**	**History**
Language				**Math**	**Language**
Art					**Math**

FIGURE 15.16 Insertion sort. Each successive column shows the result of an additional insertion of an item from the first column.

Again it is useful to imagine the list arranged vertically. The top item is considered, by itself, to be the single item in a short list of length 1; this short list is clearly in proper order. The second item in the main list is then inserted into the short list, and by an exchange if necessary, the new two-item list is properly ordered. Additional items are inserted one by one, keeping the partial list in order. When all items are inserted, the entire list is properly sorted. The details of an insertion sort applied to the same list used to illustrate a bubble sort are shown in figure 15.16.

The performance of insertion sort is similar to that of bubble sort. The number of exchanges is on average identical to the number required by bubble sort because the same symmetry argument applies. The average number of comparisons is, however, approximately one-half the number required by bubble sort, and for this reason insertion sort is considered superior to bubble sort. Both of these methods are $\Theta(n^2)$ processes on average.

The basic ideas of these algorithms, however, can be combined with tree structures to produce highly effective sorting algorithms, as discussed in the next section.

Information

From an information-theoretic viewpoint, the entropy associated with knowledge of the permutation embodied in the initial order of n items is $\log(n!)$. Since $\log(n!) \approx n\log(n/e)$, about $n\log(n/e)$ bits of information are needed to sort a list of length n.

Comparison of the order of two items constitutes a single bit. Hence, it might reasonably be inferred that there are sorting algorithms that on average require $\Theta(n\log n)$ comparisons. Furthermore, it is clear from the information-theoretic argument that this is the best that can be done. Two algorithms that achieve this average are presented in the following sections.

15.8 Quicksort

The sort algorithm considered most effective overall is **quicksort**. Its strategy is best understood as a practical implementation of the binary search tree discussed in section 15.4.

Tree Version of Insertion Sort

Imagine an insertion sort that inserts items one by one into a BST rather than into a linear list. When the tree is complete, the items can be read out in inorder to construct an ordered version of the original list. This is illustrated in figure 15.17.

This method can be extremely effective, with the one drawback that a tree must be constructed outside the original list. In other words, unlike bubble sort or insertion sort, this BST method does not take place within the list itself, but must build another structure as well.

The Quicksort Algorithm*

Quicksort provides a strategy that takes advantage of the BST structure but carries out the sort within the original list. The list is visualized as being laid out horizontally. To start, a root is selected and then, rather than processing each item in turn, all items with key values less than that of the root are placed to the left of the root, and all items with key values greater than or equal to that of the root are placed to the right. Those items now on the left can then be handled separately, using the same procedure, by selecting a lower-level root (called a pivot) for the left. Likewise, those items now on the right can be handled by a similar process. These processes are continued in each subgroup, leading to smaller subgroups, until the resulting subgroups contain only a single item or items that have equal key values.

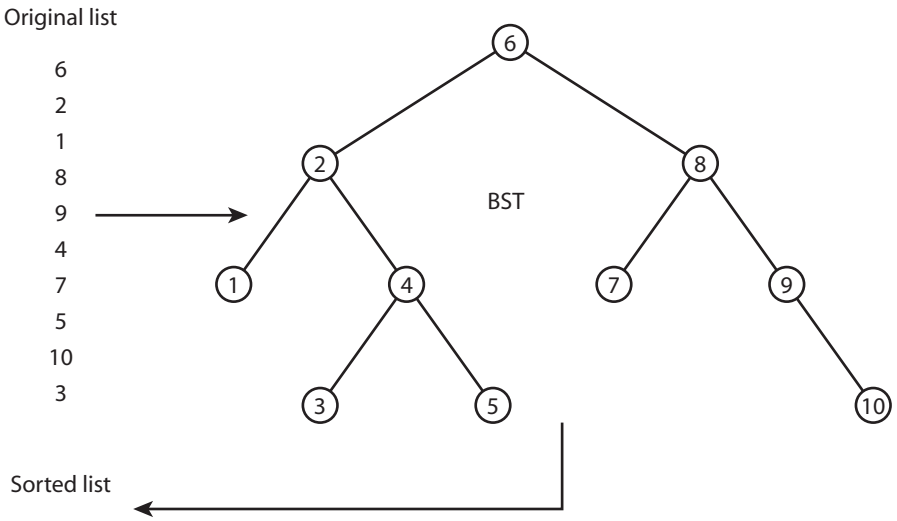


FIGURE 15.17 Insertion into a binary search tree (BST). Items from a list are inserted one by one into the BST; then the sorted version is read out to construct a sorted list.

One way to select the appropriate pivot for each group is to examine the two leftmost items and select the one with the largest key value. This guarantees that the pivot is not the item with the smallest key value.

Once the pivot is selected, some items must be moved left or right to their proper section of the list. For this purpose, left and right cursors are introduced. The left cursor begins at the far left and moves right until it encounters an element with key value equal to or greater than that of the pivot. The right cursor moves left until it encounters an item of key value less than that of the pivot. If the cursors have not met, the items they have encountered are swapped. Then the cursors continue their progress until they reach another stopping point, where another swap is made. This process continues until the cursors meet. The result is that the list is divided into two segments: a left-hand portion with all elements having key values less than that of the pivot and a right-hand portion with all items having key values greater than or equal to that of the pivot. These two segments are then processed individually in the same way, producing smaller segments, and so forth. If at any stage a segment consists of a single element or equal elements, that segment need not be processed further. Eventually, all segments will be of that type, and the sort is complete. An example is shown in figure 15.18.

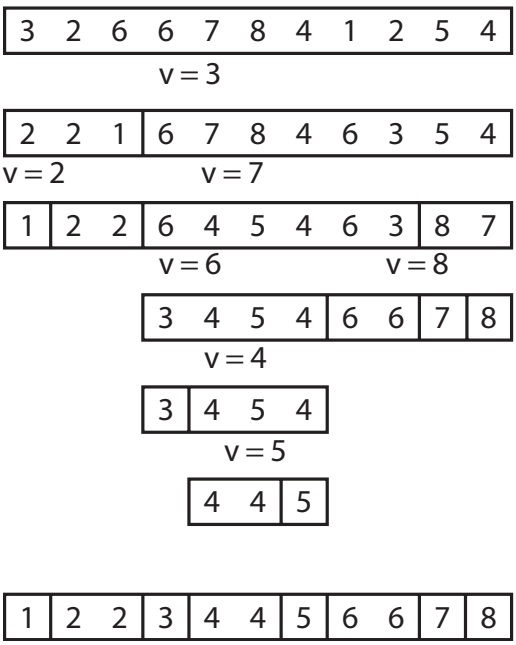


FIGURE 15.18 Quicksort. In the initial list, 3 is chosen as the pivot element (indicated by $v = 3$) since it is the larger of the first two elements. The left cursor is halted immediately at the 3. The right cursor advances leftward until it reaches 2, at which point the 3 and 2 are swapped. A further swap of 6 and 1 occurs. At that point the list is divided into two parts as shown by the separating bar in the figure. The individual portions are then processed in the same way. The final version of the list is shown in the last line.

Efficiency

Quicksort inherits its efficiency from the characteristics of the BST. The number of steps required in a path through a BST is in the worst case $\Theta(n)$ and in the best and average cases $\Theta(\log n)$. Sorting n numbers can be expected to require about n times as many steps, and accordingly, the worst performance of quicksort is $\Theta(n^2)$. The best and average cases are $O(n \log n)$ and $\Theta(n \log n)$, respectively. The $O(n \log n)$ performance is a huge improvement over bubble sort and insertion sort, and is consistent with the best performance implied by entropy considerations. A strategy to improve worst-case performance is to select the pivot points randomly. Then, against any particular set of input data, the expected number of steps is on average $\Theta(n \log n)$.

15.9 Heapsort

Heapsort is another tree-based sorting method, but it uses the partially ordered tree data structure rather than the BST. It has the (theoretical) advantage that it is at worst, best, and average a $\Theta(n \log n)$ process. Thus unlike quicksort, which may require $\Theta(n^2)$ operations in the worst case, heapsort is a $\Theta(n \log n)$ process in all cases.

First imagine that the list is to be entered into a partially ordered tree. There are two ways to do this. The first way can be viewed as a tree version of bubble sort. In this method items are initially entered into the tree in any order while simply assuring that the tree is balanced as much as possible. Then each item in the bottom level is compared with its parent, and if the parent has higher key value, the parent is swapped with its lowest-valued child. After the lowest level is processed in this way, the next higher level is processed in the same way, and so forth up to the top. This entire process is then repeated, starting again at the bottom level. After at most $O(n)$ such passes, the tree will be partially ordered.

The second method for achieving the partially ordered form can be viewed as a tree version of insertion sort. Items are entered one by one at the bottom level and moved up level by level until the key value of its parent is less than or equal to the key value of the new item.

Once the tree is partially ordered, the items can be sorted by using the push-down process discussed in section 15.5. As items are extracted from the tree, they are placed in a sorted version of the original list.

Heapsort can be carried out without constructing a tree separate from the original list by using the heap structure. The list itself is viewed as a tree by considering each item i as having items $2i$ and $2i + 1$ as its children.

Although heapsort's worst-case performance is far superior to that of quicksort and both are $O(n \log n)$ on average, experience has shown that heapsort is in practice rarely as efficient as quicksort.

Going from a linear list to a two-dimensional tree structure reduced the sorting time from $O(n^2)$ to $O(n \log n)$. One might suspect that going to a three-dimensional structure will give further improvement. However, consideration of entropy shows that $O(n \log n)$ is the best that can be achieved on average.

15.10 Merges

Frequently it is necessary to merge two lists L_1 and L_2, each of which has been sorted, into a new sorted list L containing all of the items in both L_1 and L_2. If these two lists can be simultaneously accommodated in the internal memory of the computer, there is a simple and effective method to attain the desired result.

The method begins by comparing the first item in each list, and inserting the item with the lowest key value of the two into the master list and removing it from its original list. This procedure is repeated until all items have been inserted, or until one of the original lists is exhausted, in which case the remainder of the surviving list is inserted.

This method can be extended to the merge of any number of sorted lists: simply compare the first of each and insert the one with lowest key value into the master list, deleting it from its original list.

It may be a law of nature that people seem to need more data than can be handled conveniently in their existing computing systems, and certainly more than can be stored in internal memory. Thus, historically, computer systems have employed tape drives, magnetic drums, and magnetic disks as external storage to augment internal storage.

Sorting huge lists that cannot be accommodated in internal memory is usually carried out by dividing the list into a number of smaller sublists, each of which can be sorted in internal memory. These separate sorted lists are then merged. The overall sorting and merging strategy moves segments of data back and forth between internal and external memory. There are many such strategies, the advantage of each depending somewhat on the physical characteristics and performance of internal and external storage devices.

Linear Time Sorting

There are a number of sorting algorithms that sort items in $O(n)$ time, which of course is faster than the $\Theta(n \log n)$ algorithms discussed in the past few sections. The difference is that these algorithms apply to special cases—cases where the key values of the items have some known structure. For example, **counting sort** applies to sorting integers known to lie in a fixed range 0 to k. These algorithms take advantage of the special structure to reduce direct comparisons between elements. As a simple example, in counting sort the proper placement of the element 0 is known to be at the top of the list; no comparison with other numbers is required.

15.11 EXERCISES

1. (Alphabetize) Insert these items into a BST (using alphabetical order): **Hockey**, **Baseball**, **Football**, **Tennis**, **Swimming**, **Ice skating**, **Badminton**, **Hopscotch**, **Basketball**, **Water polo**, **Crew**.
2. (Harmonic inequality) By graphically comparing $\int (1/x)\,dx$ to $\sum (1/k)$, prove that
$$\ln(n+1) - ln2 \leq \sum_{k=2}^{n} \frac{1}{k} \leq \ln n.$$
3. (Lower estimate) Use exercise 2 to find a lower bound $P^l(n)$ for the average path length of a BST, and show that this average length is in fact $\Theta(\log n)$.
4. (Balanced) Consider a binary tree balanced as much as possible. Suppose that all elements at the bottom level are first considered for exchange upward, then the next level, etc. However, a child is bubbled up only if its key value is less than that of its parent and no greater than that of its sibling. Show that once an element bubbles upward (after processing its entire level), it never moves down again. Hence, argue that putting an n-node tree in partial order with the bubble up process is an $O(n \log n)$ process.
5. (Perfectly balanced) A perfectly balanced binary tree has 2^{k-1} nodes at level k for each level k. If there are m levels, the total number of nodes is thus $n = \sum_{k=1}^{m} 2^{k-1} = 2^m - 1$.
   (a) Argue that the average length of a path from the root to a random node is
   $$L(m) = \frac{\sum_{k=1}^{m} k2^{k-1}}{2^m - 1}.$$
   (b) Show that
   $$L(m) = m - 1 + \frac{m}{2^m - 1} = m - 1 + \frac{m}{n}.$$
   That is, for large m, the average length is essentially equal to the length to the second-to-last level.
   Hint:
   $$\sum_{k=1}^{m} a^k = \frac{a - a^{m+1}}{1 - a}$$
   $$\sum_{k=1}^{m} ka^k = \frac{a + a^{m+1}[am - m - 1]}{(1-a)^2}.$$
6. (Bubble count) Consider the list $L = (5, 3, 1, 2, 4)$.
   (a) Sort the list L with bubble sort and count the number of exchanges required.
   (b) Sort the list $\overline{L}$, which has the reverse order of L, and count the number of exchanges required.
   (c) Is the sum of these exchanges equal to $n(n-1)/2$, where n is the length of the list?

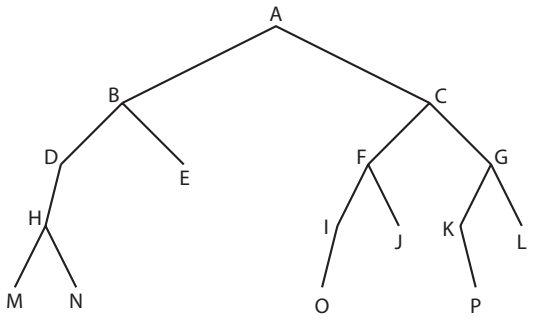


FIGURE 15.19 Tree for exercise.

7. (Order the tree) Given the tree in figure 15.19, order the elements in inorder and preorder.
8. (A quicksort) Do a quicksort of the following numbers: 3, 6, 7, 2, 9, 1, 4, using 6 as the initial root.

15.12 Bibliography

The material of this chapter is treated comprehensively in several good textbooks such as [2], [3], [4], and [5]. An especially concise and modern presentation that greatly influenced this chapter is [1]. The method for evaluating the average path length in a BST is adopted from the comprehensive text [5]. [6] is a valuable reference that provides depth and numerous extensions of the methods presented in this chapter.

References

[1] Aho, Alfred V., John E. Hopcroft, and Jeffrey Ullman. *Data Structures and Algorithms.* Reading, Mass.: Addison-Wesley, 1983.

[2] Lewis, T. G., and M. Z. Smith. *Applying Data Structures*. Boston: Houghton Mifflin, 1976.

[3] Reingold, Edward M., and Wilfred J. Hansen. *Data Structures in Pascal*. Boston: Little, Brown, 1986.

[4] Smith, Harry F. *Data Structures: Form and Function*. San Diego: Harcourt Brace Jovanovich, 1987.

[5] Cormen, Thomas H., Charles E. Leison, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 2nd ed. Cambridge: MIT Press, 2001.

[6] Knuth, Donald E. *The Art of Computer Programming.* Vol. 3, *Sorting and Searching*. Reading, Mass.: Addison-Wesley, 1973.