Lecture 2

- Expressing algorithms.
- Review of data structures.

Our syntax for algorithms

We observe in the context of our review of data structures below that every data structure is typically associated with its set of algorithms. Consequently, before we discuss data structures, we clarify the syntax we adopt for algorithms. Our *model of computation*, i.e., kind of computer on which we run our algorithms, is single-threaded, with execution one "instruction" or "step" at a time. We express an algorithm using *pseudo-code*. In our syntax for pseudo-code:

- Every algorithm has inputs, outputs. Either or both may be the empty string.
- We assume we have a basic data type, which is integer. Note that I say "integer" and not, for example, "int" as in some programming languages such as C. We mean "integer" in the full mathematical sense, i.e., our integers are unbounded.
- We allow variables, and assignment of values to variables.

  Note: I use "$\leftarrow$" for assignment, and "=" for equality testing.
- We allow checking conditions using **if** statements, and looping using **while** and **for**.

- We allow more complex data structures to be specified with a natural syntax — e.g., see the example isInArray algorithms below each of which refers to an array data structure.

- We allow subroutines, which themselves are algorithms.

Following are three examples that help illustrate our syntax for pseudo-code.

Example 1:

```
isInArray(A[1,...,n], i)
1 foreach j from 1 to n do
2     if i = A[j] then return 1
3 return 0
```

Example 2:

```
BinSearch(A[0,...,n-1], lo, hi, i)
1 while lo ≤ hi do
2     mid ← ⌊(lo+hi)/2⌋
3     if A[mid] = i then return true
4     if A[mid] < i then lo ← mid + 1
5     else hi ← mid - 1
6 return false
```

Example 3:

```
RandMedian(A[1,...,n])
1 while true do
2     i ← uniformly random choice from 1,...,n
3     c ← 0
4     foreach j from 1 to n do
5         if A[j] < A[i] then c ← c + 1
6     if c = (n-1)/2 then return i
```

Data structures

A data structure is a way to organize and retrieve data. An example of a data structure is a *set*. A data structure typically has distinguishing characteristics, which we can think of as a semantics of that data structure. For example, a set has two distinguishing characteristics: (i) every item in the set is unique, and, (ii) the items in a set are unordered. As a consequence, when one attempts to add an item to a set, the set may remain unchanged. And if one accesses the members of the set one after another, no particular order in which they are returned is to be assumed. This is exactly the semantics of the add() and iterator() methods of the Set interface in Java:

> `https://docs.oracle.com/javase/7/docs/api/java/util/Set.html#add(E)`
>
> `https://docs.oracle.com/javase/7/docs/api/java/util/Set.html#iterator()`

Another example data structure is an *array*. We associate two distinguishing characteristics with an array: (i) fixed size: once allocated, the size of an array cannot be changed, and, (ii) random access: with an array, there is the notion of an *index* with which we address each of the n entries in the array; accessing an entry at any valid index takes time independent of, or "constant in," n. We denote access to the i^{th} index of an array A as $A[i]$.

Given two data structures, for example, *set* and *array*, an interesting and meaningful question to ask is one we can call that of *expressive power*: suppose you have an implementation of one of those data structures, e.g., *array*, and perhaps some other basic types such as integers and ordered pairs, can you realize the other? Even if the answer is 'yes,' one may ask what the cost of doing so is.

For example, to realize a set from an array, we observe that adding a new member to the set may require reallocating a new array, as they may no longer be room to add a new member. Thus, for this operation in the worst-case, add() to the set has cost that is linear in the number of members of the set. Even if we have sufficient space to add the new item, we would need to check whether this item is a duplicate of a member that is already in the set, which in turn can cost linear-time if the array is not sorted.

Every data structure is typically associated with a number of algorithms, for example, to create an instance of the data structure, and to store in and retrieve data from it. For example, the algorithm for `add()` to a set has domain the cartesian product of the set of all sets and the set of all possible members, and maps $\langle S, a \rangle$ to the set $S \cup \{a\}$.

We now revert to CLRS for more discussions on: (i) graphs, (ii) trees, (iii) their chapter on 'elementary data structures,' and, (iv) Binary Search Trees.

B.4 Graphs

This section presents two kinds of graphs: directed and undirected. Certain definitions in the literature differ from those given here, but for the most part, the differences are slight.

A ***directed graph*** (or ***digraph***) G is a pair (V, E), where V is a finite set and E is a binary relation on V. The set V is called the ***vertex set*** of G, and its elements are called ***vertices*** (singular: ***vertex***). The set E is called the ***edge set*** of G, and its elements are called ***edges***. Figure B.2(a) is a pictorial representation of a directed graph on the vertex set $\{1, 2, 3, 4, 5, 6\}$. Vertices are represented by circles in the figure, and edges are represented by arrows. Note that ***self-loops***—edges from a vertex to itself—are possible.

In an ***undirected graph*** $G = (V, E)$, the edge set E consists of *unordered* pairs of vertices, rather than ordered pairs. That is, an edge is a set $\{u, v\}$, where $u, v \in V$ and $u \neq v$. By convention, we use the notation (u, v) for an edge, rather than the set notation $\{u, v\}$, and (u, v) and (v, u) are considered to be the same edge. In an undirected graph, self-loops are forbidden, and so every edge consists of exactly two distinct vertices. Figure B.2(b) is a pictorial representation of an undirected graph on the vertex set $\{1, 2, 3, 4, 5, 6\}$.

Many definitions for directed and undirected graphs are the same, although certain terms have slightly different meanings in the two contexts. If (u, v) is an edge in a directed graph $G = (V, E)$, we say that (u, v) is ***incident from*** or ***leaves*** vertex u and is ***incident to*** or ***enters*** vertex v. For example, the edges leaving vertex 2 in Figure B.2(a) are $(2, 2)$, $(2, 4)$, and $(2, 5)$. The edges entering vertex 2 are $(1, 2)$ and $(2, 2)$. If (u, v) is an edge in an undirected graph $G = (V, E)$, we say that (u, v) is ***incident on*** vertices u and v. In Figure B.2(b), the edges incident on vertex 2 are $(1, 2)$ and $(2, 5)$.

If (u, v) is an edge in a graph $G = (V, E)$, we say that vertex v is ***adjacent*** to vertex u. When the graph is undirected, the adjacency relation is symmetric. When the graph is directed, the adjacency relation is not necessarily symmetric. If v is adjacent to u in a directed graph, we sometimes write $u \rightarrow v$. In parts (a) and (b) of Figure B.2, vertex 2 is adjacent to vertex 1, since the edge $(1, 2)$ belongs to both

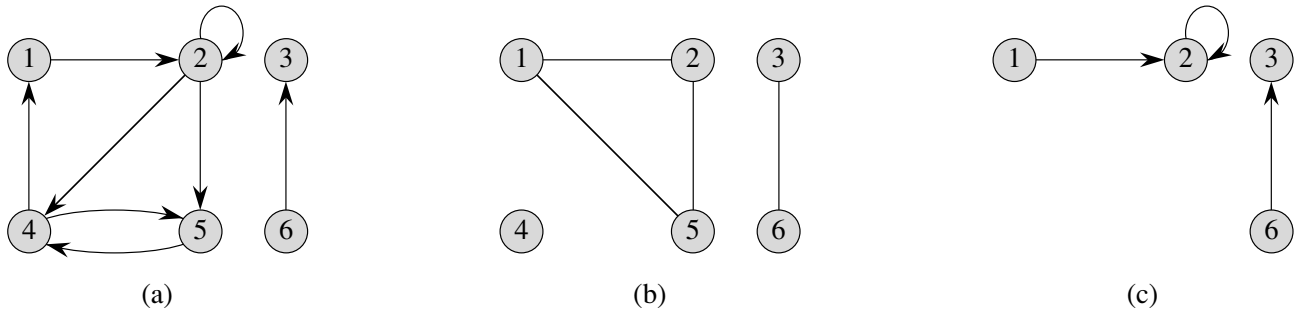

Figure B.2 Directed and undirected graphs. **(a)** A directed graph $G = (V, E)$, where $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)\}$. The edge $(2, 2)$ is a self-loop. **(b)** An undirected graph $G = (V, E)$, where $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{(1, 2), (1, 5), (2, 5), (3, 6)\}$. The vertex 4 is isolated. **(c)** The subgraph of the graph in part (a) induced by the vertex set $\{1, 2, 3, 6\}$.

graphs. Vertex 1 is *not* adjacent to vertex 2 in Figure B.2(a), since the edge $(2, 1)$ does not belong to the graph.

The ***degree*** of a vertex in an undirected graph is the number of edges incident on it. For example, vertex 2 in Figure B.2(b) has degree 2. A vertex whose degree is 0, such as vertex 4 in Figure B.2(b), is ***isolated***. In a directed graph, the ***out-degree*** of a vertex is the number of edges leaving it, and the ***in-degree*** of a vertex is the number of edges entering it. The ***degree*** of a vertex in a directed graph is its in-degree plus its out-degree. Vertex 2 in Figure B.2(a) has in-degree 2, out-degree 3, and degree 5.

A ***path*** of ***length*** k from a vertex u to a vertex u' in a graph $G = (V, E)$ is a sequence $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ of vertices such that $u = v_0$, $u' = v_k$, and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, k$. The length of the path is the number of edges in the path. The path ***contains*** the vertices $v_0, v_1, \ldots, v_k$ and the edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$. (There is always a 0-length path from u to u.) If there is a path p from u to u', we say that u' is ***reachable*** from u via p, which we sometimes write as $u \overset{p}{\rightsquigarrow} u'$ if G is directed. A path is ***simple*** if all vertices in the path are distinct. In Figure B.2(a), the path $\langle 1, 2, 5, 4 \rangle$ is a simple path of length 3. The path $\langle 2, 5, 4, 5 \rangle$ is not simple.

A ***subpath*** of path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a contiguous subsequence of its vertices. That is, for any $0 \le i \le j \le k$, the subsequence of vertices $\langle v_i, v_{i+1}, \ldots, v_j \rangle$ is a subpath of p.

In a directed graph, a path $\langle v_0, v_1, \ldots, v_k \rangle$ forms a ***cycle*** if $v_0 = v_k$ and the path contains at least one edge. The cycle is ***simple*** if, in addition, $v_1, v_2, \ldots, v_k$ are distinct. A self-loop is a cycle of length 1. Two paths $\langle v_0, v_1, v_2, \ldots, v_{k-1}, v_0 \rangle$ and $\langle v'_0, v'_1, v'_2, \ldots, v'_{k-1}, v'_0 \rangle$ form the same cycle if there exists an integer j such that $v'_i = v_{(i+j) \bmod k}$ for $i = 0, 1, \ldots, k - 1$. In Figure B.2(a), the path $\langle 1, 2, 4, 1 \rangle$

forms the same cycle as the paths $\langle 2, 4, 1, 2\rangle$ and $\langle 4, 1, 2, 4\rangle$. This cycle is simple, but the cycle $\langle 1, 2, 4, 5, 4, 1\rangle$ is not. The cycle $\langle 2, 2\rangle$ formed by the edge $(2, 2)$ is a self-loop. A directed graph with no self-loops is ***simple***. In an undirected graph, a path $\langle v_0, v_1, \ldots, v_k\rangle$ forms a ***(simple) cycle*** if $k \geq 3$, $v_0 = v_k$, and $v_1, v_2, \ldots, v_k$ are distinct. For example, in Figure B.2(b), the path $\langle 1, 2, 5, 1\rangle$ is a cycle. A graph with no cycles is ***acyclic***.

An undirected graph is ***connected*** if every pair of vertices is connected by a path. The ***connected components*** of a graph are the equivalence classes of vertices under the "is reachable from" relation. The graph in Figure B.2(b) has three connected components: $\{1, 2, 5\}$, $\{3, 6\}$, and $\{4\}$. Every vertex in $\{1, 2, 5\}$ is reachable from every other vertex in $\{1, 2, 5\}$. An undirected graph is connected if it has exactly one connected component, that is, if every vertex is reachable from every other vertex.

A directed graph is ***strongly connected*** if every two vertices are reachable from each other. The ***strongly connected components*** of a directed graph are the equivalence classes of vertices under the "are mutually reachable" relation. A directed graph is strongly connected if it has only one strongly connected component. The graph in Figure B.2(a) has three strongly connected components: $\{1, 2, 4, 5\}$, $\{3\}$, and $\{6\}$. All pairs of vertices in $\{1, 2, 4, 5\}$ are mutually reachable. The vertices $\{3, 6\}$ do not form a strongly connected component, since vertex 6 cannot be reached from vertex 3.

Two graphs $G = (V, E)$ and $G' = (V', E')$ are ***isomorphic*** if there exists a bijection $f : V \to V'$ such that $(u, v) \in E$ if and only if $(f(u), f(v)) \in E'$. In other words, we can relabel the vertices of G to be vertices of G', maintaining the corresponding edges in G and G'. Figure B.3(a) shows a pair of isomorphic graphs G and G' with respective vertex sets $V = \{1, 2, 3, 4, 5, 6\}$ and $V' = \{u, v, w, x, y, z\}$. The mapping from V to V' given by $f(1) = u$, $f(2) = v$, $f(3) = w$, $f(4) = x$, $f(5) = y$, $f(6) = z$ is the required bijective function. The graphs in Figure B.3(b) are not isomorphic. Although both graphs have 5 vertices and 7 edges, the top graph has a vertex of degree 4 and the bottom graph does not.

We say that a graph $G' = (V', E')$ is a ***subgraph*** of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. Given a set $V' \subseteq V$, the subgraph of G ***induced*** by V' is the graph $G' = (V', E')$, where

$$E' = \{(u, v) \in E : u, v \in V'\} \ .$$

The subgraph induced by the vertex set $\{1, 2, 3, 6\}$ in Figure B.2(a) appears in Figure B.2(c) and has the edge set $\{(1, 2), (2, 2), (6, 3)\}$.

Given an undirected graph $G = (V, E)$, the ***directed version*** of G is the directed graph $G' = (V, E')$, where $(u, v) \in E'$ if and only if $(u, v) \in E$. That is, each undirected edge (u, v) in G is replaced in the directed version by the two directed edges (u, v) and (v, u). Given a directed graph $G = (V, E)$, the ***undirected version*** of G is the undirected graph $G' = (V, E')$, where $(u, v) \in E'$ if and

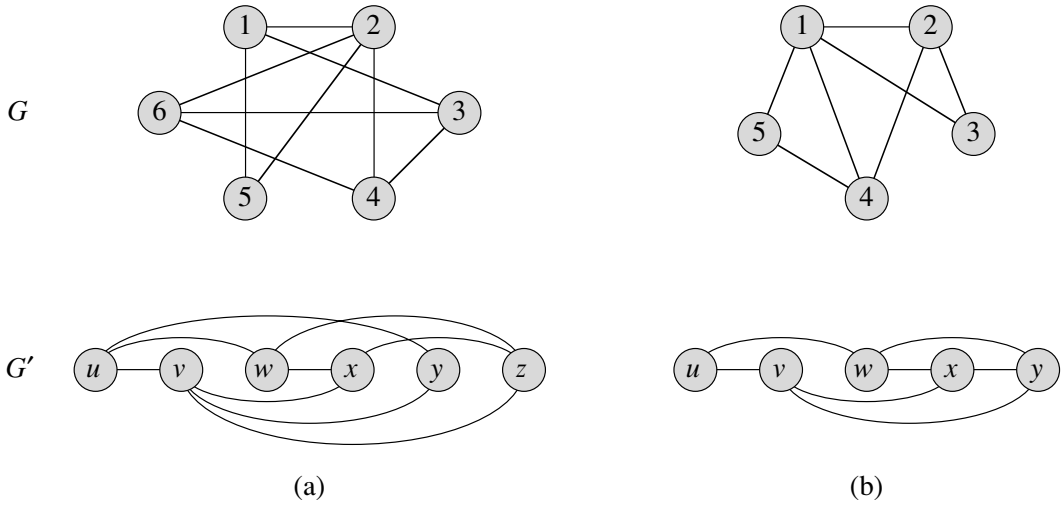

Figure B.3 **(a)** A pair of isomorphic graphs. The vertices of the top graph are mapped to the vertices of the bottom graph by $f(1) = u$, $f(2) = v$, $f(3) = w$, $f(4) = x$, $f(5) = y$, $f(6) = z$. **(b)** Two graphs that are not isomorphic, since the top graph has a vertex of degree 4 and the bottom graph does not.

only if $u \neq v$ and $(u, v) \in E$. That is, the undirected version contains the edges of G "with their directions removed" and with self-loops eliminated. (Since (u, v) and (v, u) are the same edge in an undirected graph, the undirected version of a directed graph contains it only once, even if the directed graph contains both edges (u, v) and (v, u).) In a directed graph $G = (V, E)$, a ***neighbor*** of a vertex u is any vertex that is adjacent to u in the undirected version of G. That is, v is a neighbor of u if $u \neq v$ and either $(u, v) \in E$ or $(v, u) \in E$. In an undirected graph, u and v are neighbors if they are adjacent.

Several kinds of graphs are given special names. A ***complete graph*** is an undirected graph in which every pair of vertices is adjacent. A ***bipartite graph*** is an undirected graph $G = (V, E)$ in which V can be partitioned into two sets V_1 and V_2 such that $(u, v) \in E$ implies either $u \in V_1$ and $v \in V_2$ or $u \in V_2$ and $v \in V_1$. That is, all edges go between the two sets V_1 and V_2. An acyclic, undirected graph is a ***forest***, and a connected, acyclic, undirected graph is a ***(free) tree*** (see Section B.5). We often take the first letters of "directed acyclic graph" and call such a graph a ***dag***.

B.5 Trees

As with graphs, there are many related, but slightly different, notions of trees. This section presents definitions and mathematical properties of several kinds of trees. Sections 10.4 and 22.1 describe how trees can be represented in a computer memory.

B.5.1 Free trees

As defined in Section B.4, a ***free tree*** is a connected, acyclic, undirected graph. We often omit the adjective "free" when we say that a graph is a tree. If an undirected graph is acyclic but possibly disconnected, it is a ***forest***. Many algorithms that work for trees also work for forests. Figure B.4(a) shows a free tree, and Figure B.4(b) shows a forest. The forest in Figure B.4(b) is not a tree because it is not connected. The graph in Figure B.4(c) is neither a tree nor a forest, because it contains a cycle.

The following theorem captures many important facts about free trees.

Theorem B.2 (Properties of free trees)
Let $G = (V, E)$ be an undirected graph. The following statements are equivalent.

1. G is a free tree.
2. Any two vertices in G are connected by a unique simple path.
3. G is connected, but if any edge is removed from E, the resulting graph is disconnected.
4. G is connected, and $|E| = |V| - 1$.
5. G is acyclic, and $|E| = |V| - 1$.
6. G is acyclic, but if any edge is added to E, the resulting graph contains a cycle.

Proof (1) $\Rightarrow$ (2): Since a tree is connected, any two vertices in G are connected by at least one simple path. Let u and v be vertices that are connected by two distinct simple paths p_1 and p_2, as shown in Figure B.5. Let w be the vertex at which the paths first diverge; that is, w is the first vertex on both p_1 and p_2 whose successor on p_1 is x and whose successor on p_2 is y, where $x \neq y$. Let z be the first vertex at which the paths reconverge; that is, z is the first vertex following w on p_1 that is also on p_2. Let p' be the subpath of p_1 from w through x to z, and let p'' be the subpath of p_2 from w through y to z. Paths p' and p'' share no vertices except their endpoints. Thus, the path obtained by concatenating p' and the reverse of p'' is a cycle. This contradicts our assumption that G is a tree. Thus, if G is a tree, there can be at most one simple path between two vertices.

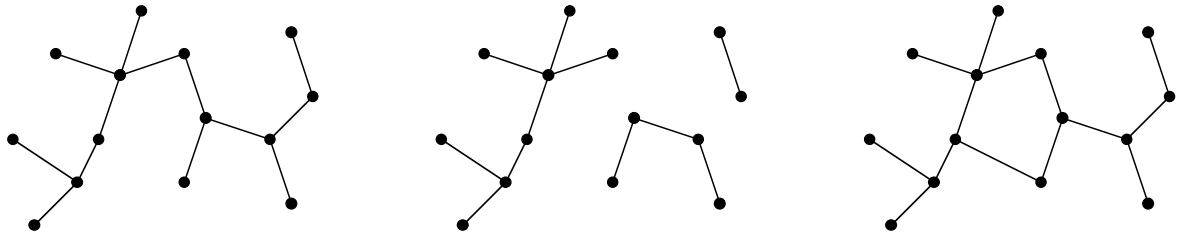

Figure B.4 **(a)** A free tree. **(b)** A forest. **(c)** A graph that contains a cycle and is therefore neither a tree nor a forest.

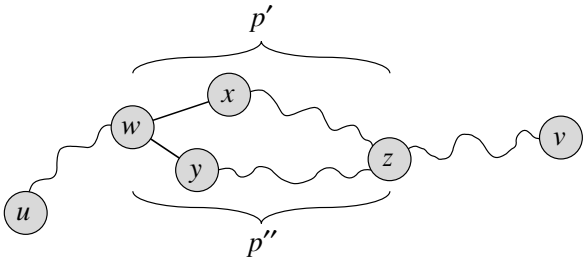

Figure B.5 A step in the proof of Theorem B.2: if (1) G is a free tree, then (2) any two vertices in G are connected by a unique simple path. Assume for the sake of contradiction that vertices u and v are connected by two distinct simple paths p_1 and p_2. These paths first diverge at vertex w, and they first reconverge at vertex z. The path p' concatenated with the reverse of the path p'' forms a cycle, which yields the contradiction.

(2) $\Rightarrow$ (3): If any two vertices in G are connected by a unique simple path, then G is connected. Let (u, v) be any edge in E. This edge is a path from u to v, and so it must be the unique path from u to v. If we remove (u, v) from G, there is no path from u to v, and hence its removal disconnects G.

(3) $\Rightarrow$ (4): By assumption, the graph G is connected, and by Exercise B.4-3, we have $|E| \geq |V| - 1$. We shall prove $|E| \leq |V| - 1$ by induction. A connected graph with $n = 1$ or $n = 2$ vertices has $n - 1$ edges. Suppose that G has $n \geq 3$ vertices and that all graphs satisfying (3) with fewer than n vertices also satisfy $|E| \leq |V| - 1$. Removing an arbitrary edge from G separates the graph into $k \geq 2$ connected components (actually $k = 2$). Each component satisfies (3), or else G would not satisfy (3). Thus, by induction, the number of edges in all components combined is at most $|V| - k \leq |V| - 2$. Adding in the removed edge yields $|E| \leq |V| - 1$.

(4) $\Rightarrow$ (5): Suppose that G is connected and that $|E| = |V| - 1$. We must show that G is acyclic. Suppose that G has a cycle containing k vertices $v_1, v_2, \ldots, v_k$, and without loss of generality assume that this cycle is simple. Let $G_k = (V_k, E_k)$

be the subgraph of G consisting of the cycle. Note that $|V_k| = |E_k| = k$. If $k < |V|$, there must be a vertex $v_{k+1} \in V - V_k$ that is adjacent to some vertex $v_i \in V_k$, since G is connected. Define $G_{k+1} = (V_{k+1}, E_{k+1})$ to be the subgraph of G with $V_{k+1} = V_k \cup \{v_{k+1}\}$ and $E_{k+1} = E_k \cup \{(v_i, v_{k+1})\}$. Note that $|V_{k+1}| = |E_{k+1}| = k + 1$. If $k + 1 < |V|$, we can continue, defining G_{k+2} in the same manner, and so forth, until we obtain $G_n = (V_n, E_n)$, where $n = |V|$, $V_n = V$, and $|E_n| = |V_n| = |V|$. Since G_n is a subgraph of G, we have $E_n \subseteq E$, and hence $|E| \geq |V|$, which contradicts the assumption that $|E| = |V| - 1$. Thus, G is acyclic.

(5) $\Rightarrow$ (6): Suppose that G is acyclic and that $|E| = |V| - 1$. Let k be the number of connected components of G. Each connected component is a free tree by definition, and since (1) implies (5), the sum of all edges in all connected components of G is $|V| - k$. Consequently, we must have $k = 1$, and G is in fact a tree. Since (1) implies (2), any two vertices in G are connected by a unique simple path. Thus, adding any edge to G creates a cycle.

(6) $\Rightarrow$ (1): Suppose that G is acyclic but that if any edge is added to E, a cycle is created. We must show that G is connected. Let u and v be arbitrary vertices in G. If u and v are not already adjacent, adding the edge (u, v) creates a cycle in which all edges but (u, v) belong to G. Thus, there is a path from u to v, and since u and v were chosen arbitrarily, G is connected. ■

B.5.2 Rooted and ordered trees

A ***rooted tree*** is a free tree in which one of the vertices is distinguished from the others. The distinguished vertex is called the ***root*** of the tree. We often refer to a vertex of a rooted tree as a ***node***[4] of the tree. Figure B.6(a) shows a rooted tree on a set of 12 nodes with root 7.

Consider a node x in a rooted tree T with root r. Any node y on the unique path from r to x is called an ***ancestor*** of x. If y is an ancestor of x, then x is a ***descendant*** of y. (Every node is both an ancestor and a descendant of itself.) If y is an ancestor of x and $x \neq y$, then y is a ***proper ancestor*** of x and x is a ***proper descendant*** of y. The ***subtree rooted at x*** is the tree induced by descendants of x, rooted at x. For example, the subtree rooted at node 8 in Figure B.6(a) contains nodes 8, 6, 5, and 9.

If the last edge on the path from the root r of a tree T to a node x is (y, x), then y is the ***parent*** of x, and x is a ***child*** of y. The root is the only node in T with

[4]The term "node" is often used in the graph theory literature as a synonym for "vertex." We shall reserve the term "node" to mean a vertex of a rooted tree.

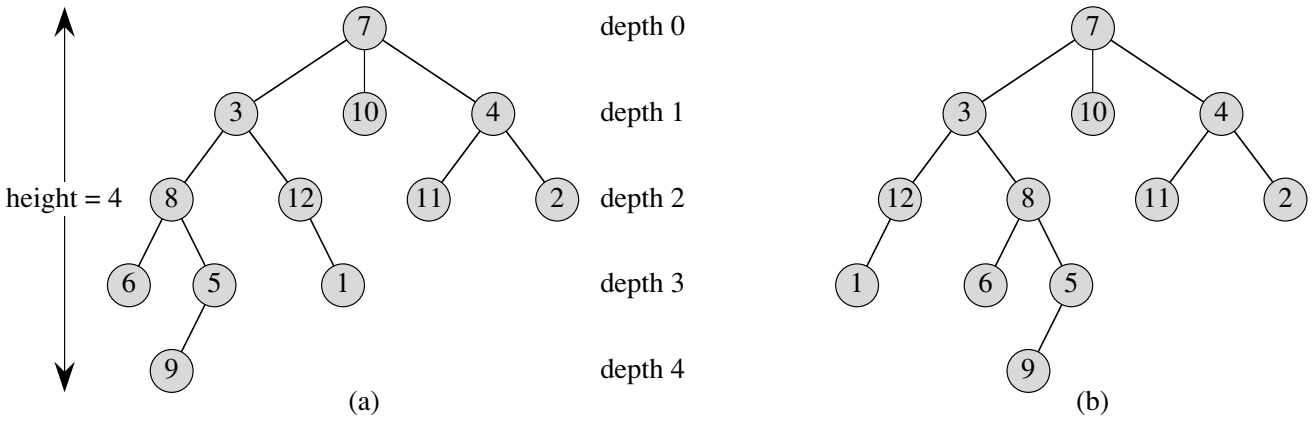

Figure B.6 Rooted and ordered trees. **(a)** A rooted tree with height 4. The tree is drawn in a standard way: the root (node 7) is at the top, its children (nodes with depth 1) are beneath it, their children (nodes with depth 2) are beneath them, and so forth. If the tree is ordered, the relative left-to-right order of the children of a node matters; otherwise it doesn't. **(b)** Another rooted tree. As a rooted tree, it is identical to the tree in (a), but as an ordered tree it is different, since the children of node 3 appear in a different order.

no parent. If two nodes have the same parent, they are ***siblings***. A node with no children is an ***external node*** or ***leaf***. A nonleaf node is an ***internal node***.

The number of children of a node x in a rooted tree T is called the ***degree*** of x.[5] The length of the path from the root r to a node x is the ***depth*** of x in T. The ***height*** of a node in a tree is the number of edges on the longest simple downward path from the node to a leaf, and the height of a tree is the height of its root. The height of a tree is also equal to the largest depth of any node in the tree.

An ***ordered tree*** is a rooted tree in which the children of each node are ordered. That is, if a node has k children, then there is a first child, a second child, ..., and a kth child. The two trees in Figure B.6 are different when considered to be ordered trees, but the same when considered to be just rooted trees.

B.5.3 Binary and positional trees

Binary trees are defined recursively. A ***binary tree*** T is a structure defined on a finite set of nodes that either

---

[5]Notice that the degree of a node depends on whether T is considered to be a rooted tree or a free tree. The degree of a vertex in a free tree is, as in any undirected graph, the number of adjacent vertices. In a rooted tree, however, the degree is the number of children—the parent of a node does not count toward its degree.

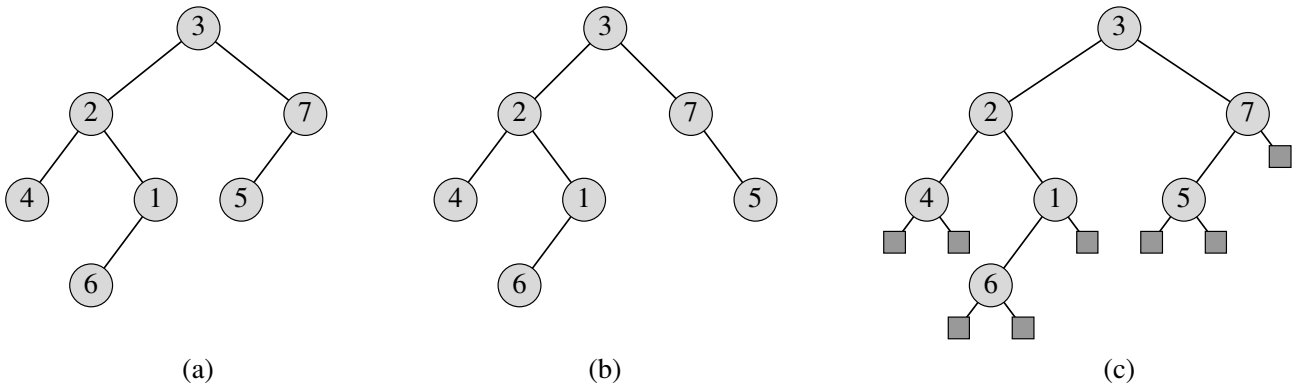

Figure B.7 Binary trees. **(a)** A binary tree drawn in a standard way. The left child of a node is drawn beneath the node and to the left. The right child is drawn beneath and to the right. **(b)** A binary tree different from the one in (a). In (a), the left child of node 7 is 5 and the right child is absent. In (b), the left child of node 7 is absent and the right child is 5. As ordered trees, these trees are the same, but as binary trees, they are distinct. **(c)** The binary tree in (a) represented by the internal nodes of a full binary tree: an ordered tree in which each internal node has degree 2. The leaves in the tree are shown as squares.

- contains no nodes, or
- is composed of three disjoint sets of nodes: a ***root*** node, a binary tree called its ***left subtree***, and a binary tree called its ***right subtree***.

The binary tree that contains no nodes is called the ***empty tree*** or ***null tree***, sometimes denoted NIL. If the left subtree is nonempty, its root is called the ***left child*** of the root of the entire tree. Likewise, the root of a nonnull right subtree is the ***right child*** of the root of the entire tree. If a subtree is the null tree NIL, we say that the child is ***absent*** or ***missing***. Figure B.7(a) shows a binary tree.

A binary tree is not simply an ordered tree in which each node has degree at most 2. For example, in a binary tree, if a node has just one child, the position of the child—whether it is the ***left child*** or the ***right child***—matters. In an ordered tree, there is no distinguishing a sole child as being either left or right. Figure B.7(b) shows a binary tree that differs from the tree in Figure B.7(a) because of the position of one node. Considered as ordered trees, however, the two trees are identical.

The positioning information in a binary tree can be represented by the internal nodes of an ordered tree, as shown in Figure B.7(c). The idea is to replace each missing child in the binary tree with a node having no children. These leaf nodes are drawn as squares in the figure. The tree that results is a ***full binary tree***: each node is either a leaf or has degree exactly 2. There are no degree-1 nodes. Consequently, the order of the children of a node preserves the position information.

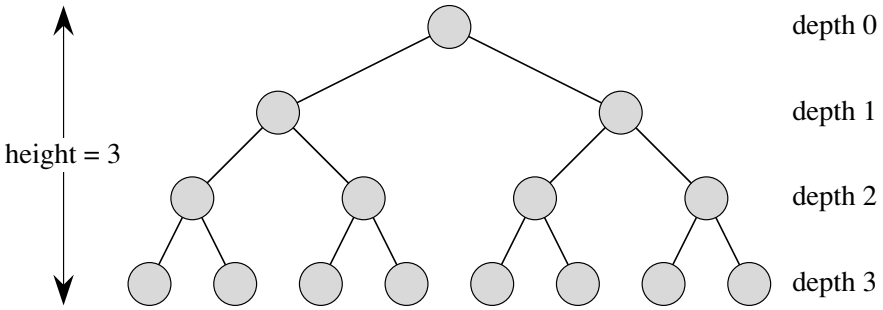

Figure B.8 A complete binary tree of height 3 with 8 leaves and 7 internal nodes.

The positioning information that distinguishes binary trees from ordered trees can be extended to trees with more than 2 children per node. In a ***positional tree***, the children of a node are labeled with distinct positive integers. The ith child of a node is ***absent*** if no child is labeled with integer i. A ***k-ary*** tree is a positional tree in which for every node, all children with labels greater than k are missing. Thus, a binary tree is a k-ary tree with $k = 2$.

A ***complete k-ary tree*** is a k-ary tree in which all leaves have the same depth and all internal nodes have degree k. Figure B.8 shows a complete binary tree of height 3. How many leaves does a complete k-ary tree of height h have? The root has k children at depth 1, each of which has k children at depth 2, etc. Thus, the number of leaves at depth h is k^h. Consequently, the height of a complete k-ary tree with n leaves is $\log_k n$. The number of internal nodes of a complete k-ary tree of height h is

$$\begin{aligned} 1 + k + k^2 + \cdots + k^{h-1} &= \sum_{i=0}^{h-1} k^i \\ &= \frac{k^h - 1}{k - 1} \end{aligned}$$

by equation (A.5). Thus, a complete binary tree has $2^h - 1$ internal nodes.

10 Elementary Data Structures

In this chapter, we examine the representation of dynamic sets by simple data structures that use pointers. Although many complex data structures can be fashioned using pointers, we present only the rudimentary ones: stacks, queues, linked lists, and rooted trees. We also discuss a method by which objects and pointers can be synthesized from arrays.

10.1 Stacks and queues

Stacks and queues are dynamic sets in which the element removed from the set by the DELETE operation is prespecified. In a ***stack***, the element deleted from the set is the one most recently inserted: the stack implements a ***last-in, first-out***, or ***LIFO***, policy. Similarly, in a ***queue***, the element deleted is always the one that has been in the set for the longest time: the queue implements a ***first-in, first-out***, or ***FIFO***, policy. There are several efficient ways to implement stacks and queues on a computer. In this section we show how to use a simple array to implement each.

Stacks

The INSERT operation on a stack is often called PUSH, and the DELETE operation, which does not take an element argument, is often called POP. These names are allusions to physical stacks, such as the spring-loaded stacks of plates used in cafeterias. The order in which plates are popped from the stack is the reverse of the order in which they were pushed onto the stack, since only the top plate is accessible.

As shown in Figure 10.1, we can implement a stack of at most n elements with an array $S[1..n]$. The array has an attribute $top[S]$ that indexes the most recently inserted element. The stack consists of elements $S[1..top[S]]$, where $S[1]$ is the element at the bottom of the stack and $S[top[S]]$ is the element at the top.

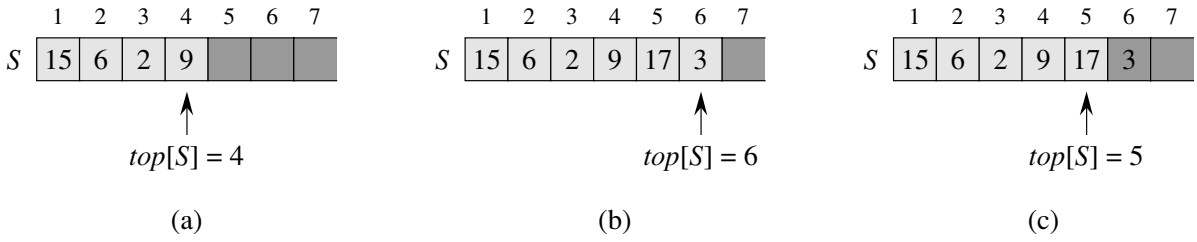

Figure 10.1 An array implementation of a stack S. Stack elements appear only in the lightly shaded positions. **(a)** Stack S has 4 elements. The top element is 9. **(b)** Stack S after the calls PUSH$(S, 17)$ and PUSH$(S, 3)$. **(c)** Stack S after the call POP(S) has returned the element 3, which is the one most recently pushed. Although element 3 still appears in the array, it is no longer in the stack; the top is element 17.

When $top[S] = 0$, the stack contains no elements and is ***empty***. The stack can be tested for emptiness by the query operation STACK-EMPTY. If an empty stack is popped, we say the stack ***underflows***, which is normally an error. If $top[S]$ exceeds n, the stack ***overflows***. (In our pseudocode implementation, we don't worry about stack overflow.)

The stack operations can each be implemented with a few lines of code.

```
STACK-EMPTY(S)
1  if top[S] = 0
2     then return TRUE
3     else  return FALSE

PUSH(S, x)
1  top[S] ← top[S] + 1
2  S[top[S]] ← x

POP(S)
1  if STACK-EMPTY(S)
2     then error "underflow"
3     else  top[S] ← top[S] − 1
4           return S[top[S] + 1]
```

Figure 10.1 shows the effects of the modifying operations PUSH and POP. Each of the three stack operations takes $O(1)$ time.

Queues

We call the INSERT operation on a queue ENQUEUE, and we call the DELETE operation DEQUEUE; like the stack operation POP, DEQUEUE takes no element

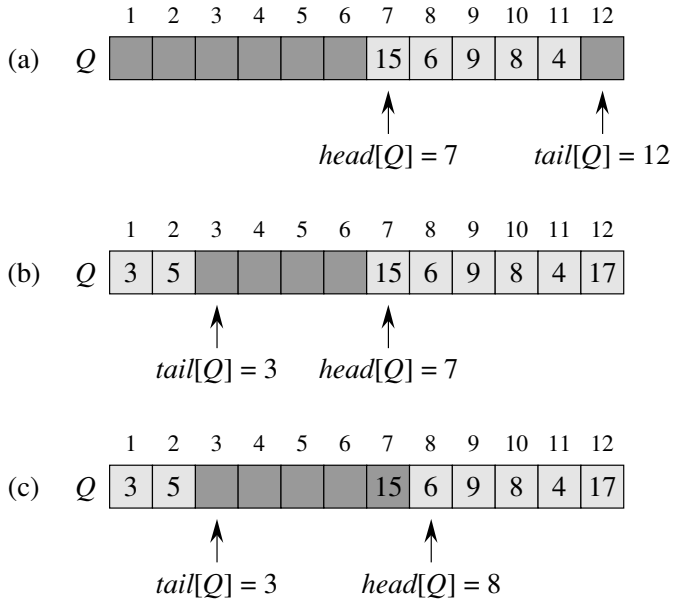

Figure 10.2 A queue implemented using an array $Q[1..12]$. Queue elements appear only in the lightly shaded positions. **(a)** The queue has 5 elements, in locations $Q[7..11]$. **(b)** The configuration of the queue after the calls ENQUEUE$(Q, 17)$, ENQUEUE$(Q, 3)$, and ENQUEUE$(Q, 5)$. **(c)** The configuration of the queue after the call DEQUEUE(Q) returns the key value 15 formerly at the head of the queue. The new head has key 6.

argument. The FIFO property of a queue causes it to operate like a line of people in the registrar's office. The queue has a ***head*** and a ***tail***. When an element is enqueued, it takes its place at the tail of the queue, just as a newly arriving student takes a place at the end of the line. The element dequeued is always the one at the head of the queue, like the student at the head of the line who has waited the longest. (Fortunately, we don't have to worry about computational elements cutting into line.)

Figure 10.2 shows one way to implement a queue of at most $n - 1$ elements using an array $Q[1..n]$. The queue has an attribute $head[Q]$ that indexes, or points to, its head. The attribute $tail[Q]$ indexes the next location at which a newly arriving element will be inserted into the queue. The elements in the queue are in locations $head[Q], head[Q]+1, \ldots, tail[Q]-1$, where we "wrap around" in the sense that location 1 immediately follows location n in a circular order. When $head[Q] = tail[Q]$, the queue is empty. Initially, we have $head[Q] = tail[Q] = 1$. When the queue is empty, an attempt to dequeue an element causes the queue to underflow. When $head[Q] = tail[Q] + 1$, the queue is full, and an attempt to enqueue an element causes the queue to overflow.

In our procedures ENQUEUE and DEQUEUE, the error checking for underflow and overflow has been omitted. (Exercise 10.1-4 asks you to supply code that checks for these two error conditions.)

```
ENQUEUE(Q, x)
1  Q[tail[Q]] ← x
2  if tail[Q] = length[Q]
3     then tail[Q] ← 1
4     else tail[Q] ← tail[Q] + 1
```

```
DEQUEUE(Q)
1  x ← Q[head[Q]]
2  if head[Q] = length[Q]
3     then head[Q] ← 1
4     else head[Q] ← head[Q] + 1
5  return x
```

Figure 10.2 shows the effects of the ENQUEUE and DEQUEUE operations. Each operation takes $O(1)$ time.

10.2 Linked lists

A ***linked list*** is a data structure in which the objects are arranged in a linear order. Unlike an array, though, in which the linear order is determined by the array indices, the order in a linked list is determined by a pointer in each object. Linked lists provide a simple, flexible representation for dynamic sets, supporting (though not necessarily efficiently) all the operations listed on page 198.

As shown in Figure 10.3, each element of a ***doubly linked list*** L is an object with a *key* field and two other pointer fields: *next* and *prev*. The object may also contain other satellite data. Given an element x in the list, $next[x]$ points to its successor in the linked list, and $prev[x]$ points to its predecessor. If $prev[x] = \text{NIL}$, the element x has no predecessor and is therefore the first element, or ***head***, of the list. If $next[x] = \text{NIL}$, the element x has no successor and is therefore the last element, or ***tail***, of the list. An attribute $head[L]$ points to the first element of the list. If $head[L] = \text{NIL}$, the list is empty.

A list may have one of several forms. It may be either singly linked or doubly linked, it may be sorted or not, and it may be circular or not. If a list is ***singly linked***, we omit the *prev* pointer in each element. If a list is ***sorted***, the linear order of the list corresponds to the linear order of keys stored in elements of the list; the minimum element is the head of the list, and the maximum element is the tail. If the list is ***unsorted***, the elements can appear in any order. In a ***circular list***, the *prev* pointer of the head of the list points to the tail, and the *next* pointer of the tail of the list points to the head. The list may thus be viewed as a ring of elements. In the remainder of this section, we assume that the lists with which we are working are unsorted and doubly linked.

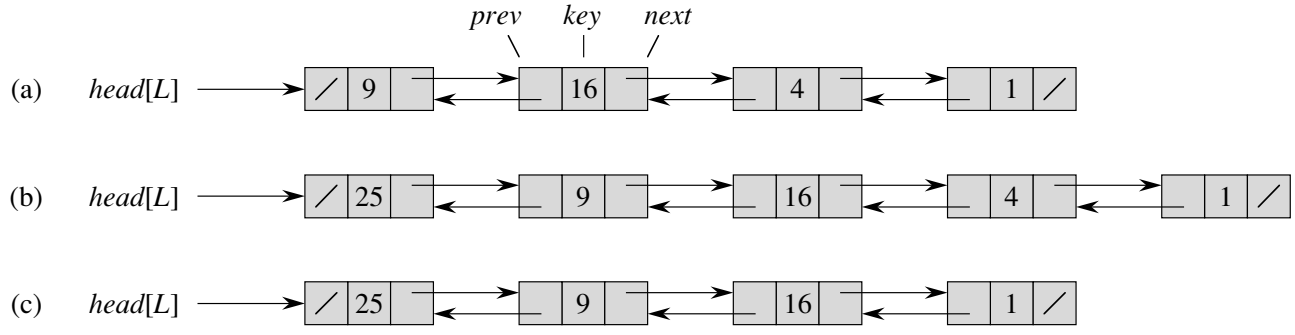

Figure 10.3 **(a)** A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with fields for the key and pointers (shown by arrows) to the next and previous objects. The *next* field of the tail and the *prev* field of the head are NIL, indicated by a diagonal slash. The attribute $head[L]$ points to the head. **(b)** Following the execution of LIST-INSERT(L, x), where $key[x] = 25$, the linked list has a new object with key 25 as the new head. This new object points to the old head with key 9. **(c)** The result of the subsequent call LIST-DELETE(L, x), where x points to the object with key 4.

Searching a linked list

The procedure LIST-SEARCH(L, k) finds the first element with key k in list L by a simple linear search, returning a pointer to this element. If no object with key k appears in the list, then NIL is returned. For the linked list in Figure 10.3(a), the call LIST-SEARCH$(L, 4)$ returns a pointer to the third element, and the call LIST-SEARCH$(L, 7)$ returns NIL.

```
LIST-SEARCH(L, k)
1  x ← head[L]
2  while x ≠ NIL and key[x] ≠ k
3        do x ← next[x]
4  return x
```

To search a list of n objects, the LIST-SEARCH procedure takes $\Theta(n)$ time in the worst case, since it may have to search the entire list.

Inserting into a linked list

Given an element x whose *key* field has already been set, the LIST-INSERT procedure "splices" x onto the front of the linked list, as shown in Figure 10.3(b).

```
LIST-INSERT(L, x)
1  next[x] ← head[L]
2  if head[L] ≠ NIL
3     then prev[head[L]] ← x
4  head[L] ← x
5  prev[x] ← NIL
```

The running time for LIST-INSERT on a list of n elements is $O(1)$.

Deleting from a linked list

The procedure LIST-DELETE removes an element x from a linked list L. It must be given a pointer to x, and it then "splices" x out of the list by updating pointers. If we wish to delete an element with a given key, we must first call LIST-SEARCH to retrieve a pointer to the element.

```
LIST-DELETE(L, x)
1  if prev[x] ≠ NIL
2     then next[prev[x]] ← next[x]
3     else head[L] ← next[x]
4  if next[x] ≠ NIL
5     then prev[next[x]] ← prev[x]
```

Figure 10.3(c) shows how an element is deleted from a linked list. LIST-DELETE runs in $O(1)$ time, but if we wish to delete an element with a given key, $\Theta(n)$ time is required in the worst case because we must first call LIST-SEARCH.

Sentinels

The code for LIST-DELETE would be simpler if we could ignore the boundary conditions at the head and tail of the list.

```
LIST-DELETE′(L, x)
1  next[prev[x]] ← next[x]
2  prev[next[x]] ← prev[x]
```

A ***sentinel*** is a dummy object that allows us to simplify boundary conditions. For example, suppose that we provide with list L an object $nil[L]$ that represents NIL but has all the fields of the other list elements. Wherever we have a reference to NIL in list code, we replace it by a reference to the sentinel $nil[L]$. As shown in Figure 10.4, this turns a regular doubly linked list into a ***circular, doubly linked list with a sentinel***, in which the sentinel $nil[L]$ is placed between the head and

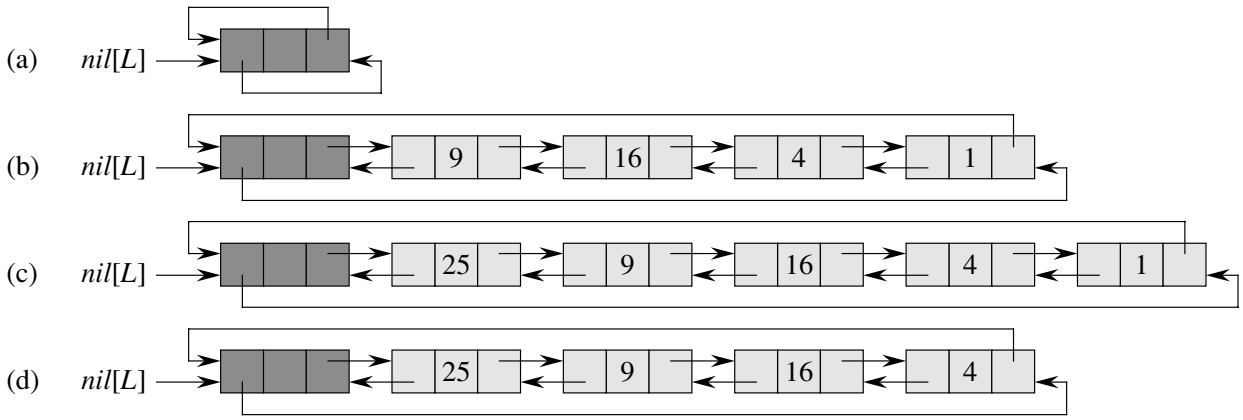

Figure 10.4 A circular, doubly linked list with a sentinel. The sentinel $nil[L]$ appears between the head and tail. The attribute $head[L]$ is no longer needed, since we can access the head of the list by $next[nil[L]]$. **(a)** An empty list. **(b)** The linked list from Figure 10.3(a), with key 9 at the head and key 1 at the tail. **(c)** The list after executing LIST-INSERT$'(L, x)$, where $key[x] = 25$. The new object becomes the head of the list. **(d)** The list after deleting the object with key 1. The new tail is the object with key 4.

tail; the field $next[nil[L]]$ points to the head of the list, and $prev[nil[L]]$ points to the tail. Similarly, both the *next* field of the tail and the *prev* field of the head point to $nil[L]$. Since $next[nil[L]]$ points to the head, we can eliminate the attribute $head[L]$ altogether, replacing references to it by references to $next[nil[L]]$. An empty list consists of just the sentinel, since both $next[nil[L]]$ and $prev[nil[L]]$ can be set to $nil[L]$.

The code for LIST-SEARCH remains the same as before, but with the references to NIL and $head[L]$ changed as specified above.

```
LIST-SEARCH'(L, k)
1  x ← next[nil[L]]
2  while x ≠ nil[L] and key[x] ≠ k
3        do x ← next[x]
4  return x
```

We use the two-line procedure LIST-DELETE$'$ to delete an element from the list. We use the following procedure to insert an element into the list.

```
LIST-INSERT'(L, x)
1  next[x] ← next[nil[L]]
2  prev[next[nil[L]]] ← x
3  next[nil[L]] ← x
4  prev[x] ← nil[L]
```

Figure 10.4 shows the effects of LIST-INSERT′ and LIST-DELETE′ on a sample list.

Sentinels rarely reduce the asymptotic time bounds of data structure operations, but they can reduce constant factors. The gain from using sentinels within loops is usually a matter of clarity of code rather than speed; the linked list code, for example, is simplified by the use of sentinels, but we save only $O(1)$ time in the LIST-INSERT′ and LIST-DELETE′ procedures. In other situations, however, the use of sentinels helps to tighten the code in a loop, thus reducing the coefficient of, say, n or n^2 in the running time.

Sentinels should not be used indiscriminately. If there are many small lists, the extra storage used by their sentinels can represent significant wasted memory. In this book, we use sentinels only when they truly simplify the code.

10.3 Implementing pointers and objects

How do we implement pointers and objects in languages, such as Fortran, that do not provide them? In this section, we shall see two ways of implementing linked data structures without an explicit pointer data type. We shall synthesize objects and pointers from arrays and array indices.

A multiple-array representation of objects

We can represent a collection of objects that have the same fields by using an array for each field. As an example, Figure 10.5 shows how we can implement the linked list of Figure 10.3(a) with three arrays. The array *key* holds the values of the keys currently in the dynamic set, and the pointers are stored in the arrays *next* and *prev*. For a given array index x, $key[x]$, $next[x]$, and $prev[x]$ represent an object in the linked list. Under this interpretation, a pointer x is simply a common index into the *key*, *next*, and *prev* arrays.

In Figure 10.3(a), the object with key 4 follows the object with key 16 in the linked list. In Figure 10.5, key 4 appears in $key[2]$, and key 16 appears in $key[5]$, so we have $next[5] = 2$ and $prev[2] = 5$. Although the constant NIL appears in the *next* field of the tail and the *prev* field of the head, we usually use an integer (such as 0 or -1) that cannot possibly represent an actual index into the arrays. A variable L holds the index of the head of the list.

In our pseudocode, we have been using square brackets to denote both the indexing of an array and the selection of a field (attribute) of an object. Either way, the meanings of $key[x]$, $next[x]$, and $prev[x]$ are consistent with implementation practice.

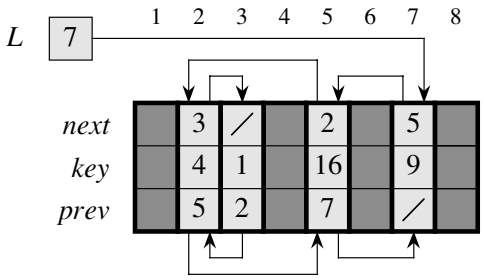

Figure 10.5 The linked list of Figure 10.3(a) represented by the arrays *key*, *next*, and *prev*. Each vertical slice of the arrays represents a single object. Stored pointers correspond to the array indices shown at the top; the arrows show how to interpret them. Lightly shaded object positions contain list elements. The variable L keeps the index of the head.

A single-array representation of objects

The words in a computer memory are typically addressed by integers from 0 to $M - 1$, where M is a suitably large integer. In many programming languages, an object occupies a contiguous set of locations in the computer memory. A pointer is simply the address of the first memory location of the object, and other memory locations within the object can be indexed by adding an offset to the pointer.

We can use the same strategy for implementing objects in programming environments that do not provide explicit pointer data types. For example, Figure 10.6 shows how a single array A can be used to store the linked list from Figures 10.3(a) and 10.5. An object occupies a contiguous subarray $A[j\,..\,k]$. Each field of the object corresponds to an offset in the range from 0 to $k - j$, and a pointer to the object is the index j. In Figure 10.6, the offsets corresponding to *key*, *next*, and *prev* are 0, 1, and 2, respectively. To read the value of $prev[i]$, given a pointer i, we add the value i of the pointer to the offset 2, thus reading $A[i + 2]$.

The single-array representation is flexible in that it permits objects of different lengths to be stored in the same array. The problem of managing such a heterogeneous collection of objects is more difficult than the problem of managing a homogeneous collection, where all objects have the same fields. Since most of the data structures we shall consider are composed of homogeneous elements, it will be sufficient for our purposes to use the multiple-array representation of objects.

Allocating and freeing objects

To insert a key into a dynamic set represented by a doubly linked list, we must allocate a pointer to a currently unused object in the linked-list representation. Thus, it is useful to manage the storage of objects not currently used in the linked-list representation so that one can be allocated. In some systems, a ***garbage collector*** is responsible for determining which objects are unused. Many applications,

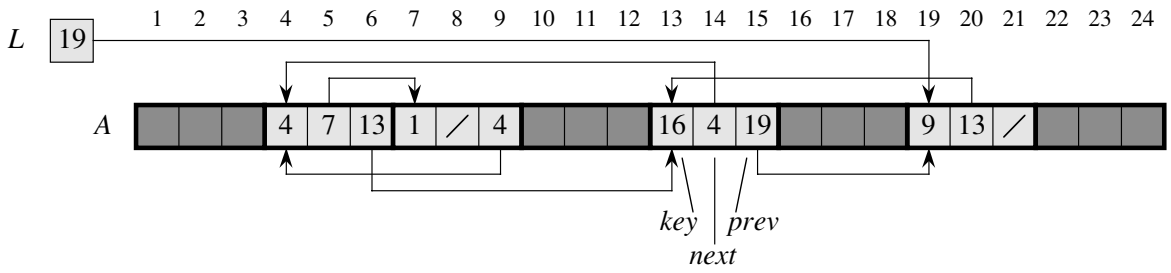

Figure 10.6 The linked list of Figures 10.3(a) and 10.5 represented in a single array A. Each list element is an object that occupies a contiguous subarray of length 3 within the array. The three fields *key*, *next*, and *prev* correspond to the offsets 0, 1, and 2, respectively. A pointer to an object is an index of the first element of the object. Objects containing list elements are lightly shaded, and arrows show the list ordering.

however, are simple enough that they can bear responsibility for returning an unused object to a storage manager. We shall now explore the problem of allocating and freeing (or deallocating) homogeneous objects using the example of a doubly linked list represented by multiple arrays.

Suppose that the arrays in the multiple-array representation have length m and that at some moment the dynamic set contains $n \leq m$ elements. Then n objects represent elements currently in the dynamic set, and the remaining $m-n$ objects are ***free***; the free objects can be used to represent elements inserted into the dynamic set in the future.

We keep the free objects in a singly linked list, which we call the ***free list***. The free list uses only the *next* array, which stores the *next* pointers within the list. The head of the free list is held in the global variable *free*. When the dynamic set represented by linked list L is nonempty, the free list may be intertwined with list L, as shown in Figure 10.7. Note that each object in the representation is either in list L or in the free list, but not in both.

The free list is a stack: the next object allocated is the last one freed. We can use a list implementation of the stack operations PUSH and POP to implement the procedures for allocating and freeing objects, respectively. We assume that the global variable *free* used in the following procedures points to the first element of the free list.

```
ALLOCATE-OBJECT()
1  if free = NIL
2     then error "out of space"
3     else x ← free
4          free ← next[x]
5          return x
```

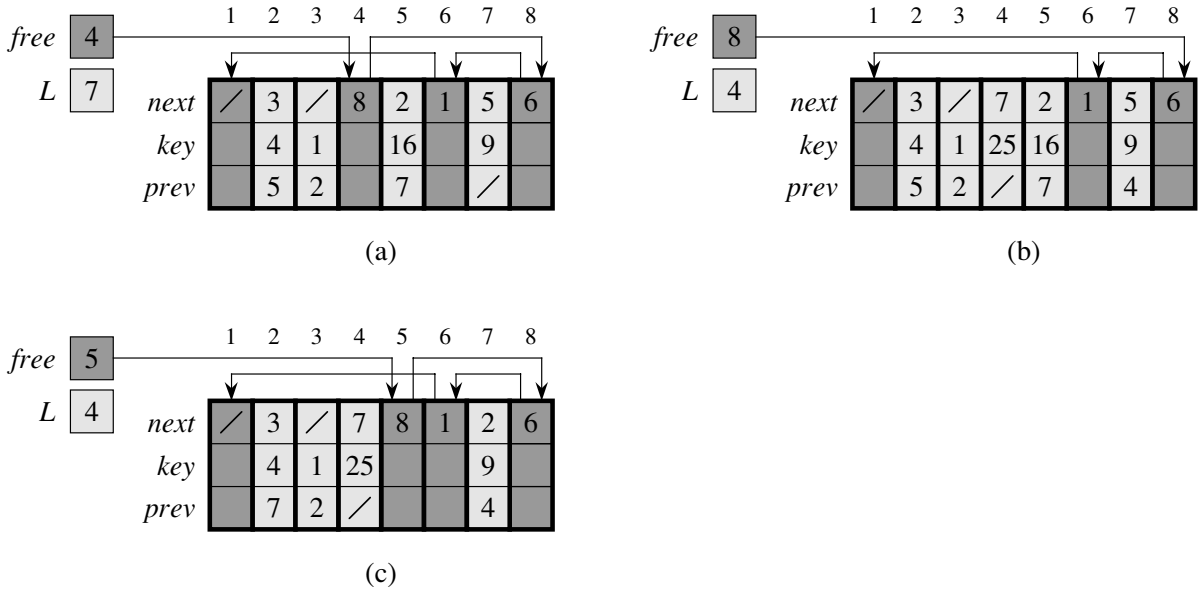

Figure 10.7 The effect of the ALLOCATE-OBJECT and FREE-OBJECT procedures. **(a)** The list of Figure 10.5 (lightly shaded) and a free list (heavily shaded). Arrows show the free-list structure. **(b)** The result of calling ALLOCATE-OBJECT() (which returns index 4), setting *key*[4] to 25, and calling LIST-INSERT(L, 4). The new free-list head is object 8, which had been *next*[4] on the free list. **(c)** After executing LIST-DELETE(L, 5), we call FREE-OBJECT(5). Object 5 becomes the new free-list head, with object 8 following it on the free list.

```
FREE-OBJECT(x)
1  next[x] ← free
2  free ← x
```

The free list initially contains all n unallocated objects. When the free list has been exhausted, the ALLOCATE-OBJECT procedure signals an error. It is common to use a single free list to service several linked lists. Figure 10.8 shows two linked lists and a free list intertwined through *key*, *next*, and *prev* arrays.

The two procedures run in $O(1)$ time, which makes them quite practical. They can be modified to work for any homogeneous collection of objects by letting any one of the fields in the object act like a *next* field in the free list.

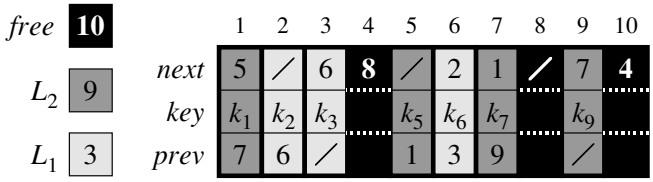

Figure 10.8 Two linked lists, L_1 (lightly shaded) and L_2 (heavily shaded), and a free list (darkened) intertwined.

10.4 Representing rooted trees

The methods for representing lists given in the previous section extend to any homogeneous data structure. In this section, we look specifically at the problem of representing rooted trees by linked data structures. We first look at binary trees, and then we present a method for rooted trees in which nodes can have an arbitrary number of children.

We represent each node of a tree by an object. As with linked lists, we assume that each node contains a *key* field. The remaining fields of interest are pointers to other nodes, and they vary according to the type of tree.

Binary trees

As shown in Figure 10.9, we use the fields p, *left*, and *right* to store pointers to the parent, left child, and right child of each node in a binary tree T. If $p[x] = \text{NIL}$, then x is the root. If node x has no left child, then $left[x] = \text{NIL}$, and similarly for the right child. The root of the entire tree T is pointed to by the attribute $root[T]$. If $root[T] = \text{NIL}$, then the tree is empty.

Rooted trees with unbounded branching

The scheme for representing a binary tree can be extended to any class of trees in which the number of children of each node is at most some constant k: we replace the *left* and *right* fields by $child_1, child_2, \ldots, child_k$. This scheme no longer works when the number of children of a node is unbounded, since we do not know how many fields (arrays in the multiple-array representation) to allocate in advance. Moreover, even if the number of children k is bounded by a large constant but most nodes have a small number of children, we may waste a lot of memory.

Fortunately, there is a clever scheme for using binary trees to represent trees with arbitrary numbers of children. It has the advantage of using only $O(n)$ space for any n-node rooted tree. The ***left-child, right-sibling representation*** is shown in Figure 10.10. As before, each node contains a parent pointer p, and $root[T]$ points to the root of tree T. Instead of having a pointer to each of its children, however, each node x has only two pointers:

1. $left\text{-}child[x]$ points to the leftmost child of node x, and
2. $right\text{-}sibling[x]$ points to the sibling of x immediately to the right.

If node x has no children, then $left\text{-}child[x] = \text{NIL}$, and if node x is the rightmost child of its parent, then $right\text{-}sibling[x] = \text{NIL}$.

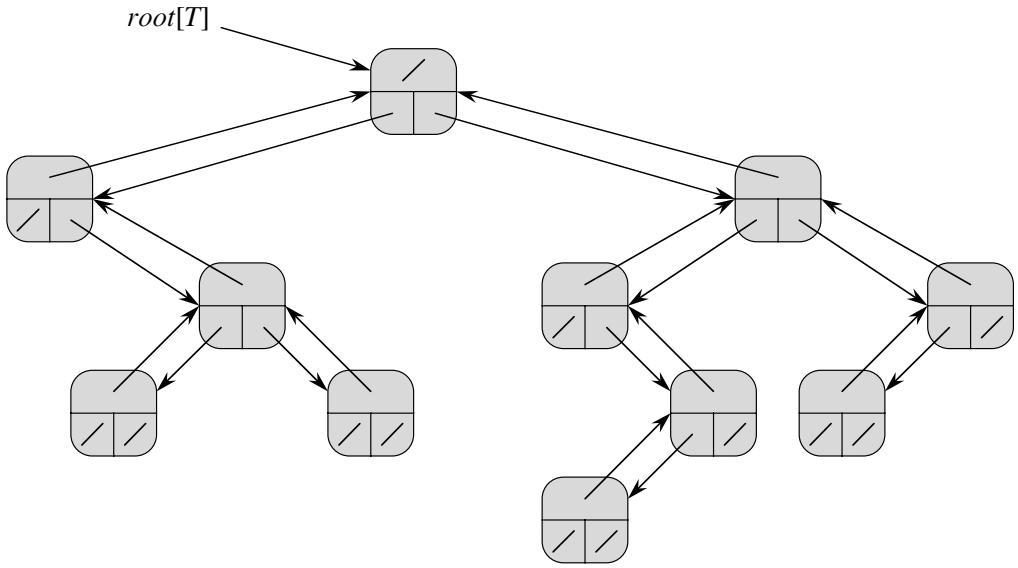

Figure 10.9 The representation of a binary tree T. Each node x has the fields $p[x]$ (top), $left[x]$ (lower left), and $right[x]$ (lower right). The *key* fields are not shown.

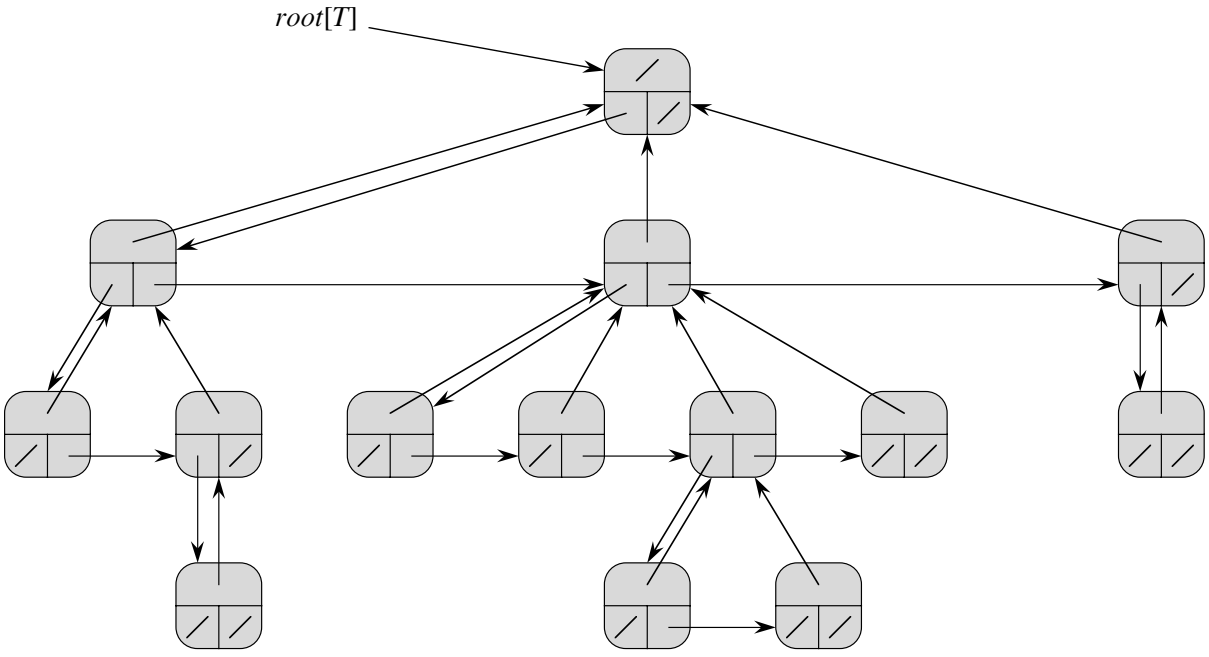

Figure 10.10 The left-child, right-sibling representation of a tree T. Each node x has fields $p[x]$ (top), $left\text{-}child[x]$ (lower left), and $right\text{-}sibling[x]$ (lower right). Keys are not shown.

Other tree representations

We sometimes represent rooted trees in other ways. In Chapter 6, for example, we represented a heap, which is based on a complete binary tree, by a single array plus an index. The trees that appear in Chapter 21 are traversed only toward the root, so only the parent pointers are present; there are no pointers to children. Many other schemes are possible. Which scheme is best depends on the application.

12.1 What is a binary search tree?

A binary search tree is organized, as the name suggests, in a binary tree, as shown in Figure 12.1. Such a tree can be represented by a linked data structure in which each node is an object. In addition to a *key* field and satellite data, each node

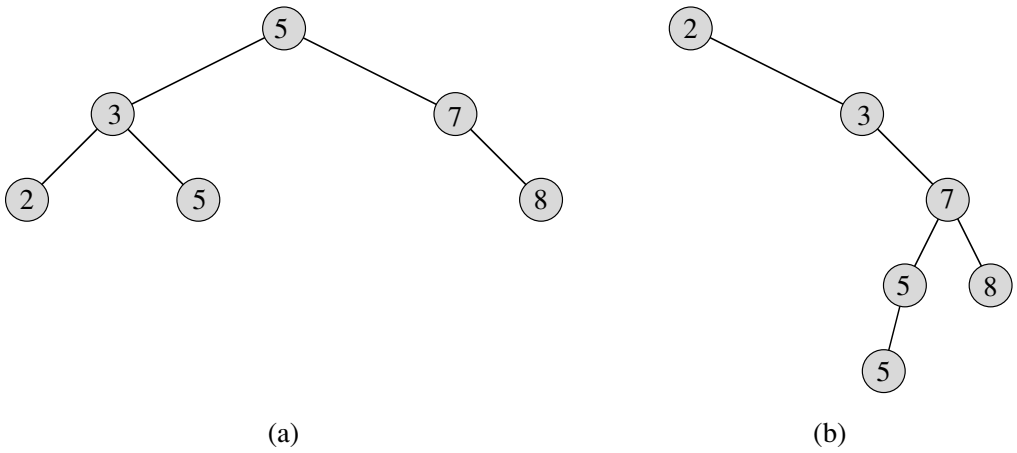

Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $key[x]$, and the keys in the right subtree of x are at least $key[x]$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. **(b)** A less efficient binary search tree with height 4 that contains the same keys.

contains fields *left*, *right*, and p that point to the nodes corresponding to its left child, its right child, and its parent, respectively. If a child or the parent is missing, the appropriate field contains the value NIL. The root node is the only node in the tree whose parent field is NIL.

The keys in a binary search tree are always stored in such a way as to satisfy the ***binary-search-tree property***:

> Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $key[y] \leq key[x]$. If y is a node in the right subtree of x, then $key[x] \leq key[y]$.

Thus, in Figure 12.1(a), the key of the root is 5, the keys 2, 3, and 5 in its left subtree are no larger than 5, and the keys 7 and 8 in its right subtree are no smaller than 5. The same property holds for every node in the tree. For example, the key 3 in Figure 12.1(a) is no smaller than the key 2 in its left subtree and no larger than the key 5 in its right subtree.

The binary-search-tree property allows us to print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an ***inorder tree walk***. This algorithm is so named because the key of the root of a subtree is printed between the values in its left subtree and those in its right subtree. (Similarly, a ***preorder tree walk*** prints the root before the values in either subtree, and a ***postorder tree walk*** prints the root after the values in its subtrees.) To use the following procedure to print all the elements in a binary search tree T, we call INORDER-TREE-WALK($root[T]$).

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2     then INORDER-TREE-WALK(left[x])
3          print key[x]
4          INORDER-TREE-WALK(right[x])
```

As an example, the inorder tree walk prints the keys in each of the two binary search trees from Figure 12.1 in the order 2, 3, 5, 5, 7, 8. The correctness of the algorithm follows by induction directly from the binary-search-tree property.

It takes $\Theta(n)$ time to walk an n-node binary search tree, since after the initial call, the procedure is called recursively exactly twice for each node in the tree—once for its left child and once for its right child. The following theorem gives a more formal proof that it takes linear time to perform an inorder tree walk.

Theorem 12.1
If x is the root of an n-node subtree, then the call INORDER-TREE-WALK(x) takes $\Theta(n)$ time.

Proof Let $T(n)$ denote the time taken by INORDER-TREE-WALK when it is called on the root of an n-node subtree. INORDER-TREE-WALK takes a small, constant amount of time on an empty subtree (for the test $x \neq$ NIL), and so $T(0) = c$ for some positive constant c.

For $n > 0$, suppose that INORDER-TREE-WALK is called on a node x whose left subtree has k nodes and whose right subtree has $n - k - 1$ nodes. The time to perform INORDER-TREE-WALK(x) is $T(n) = T(k) + T(n - k - 1) + d$ for some positive constant d that reflects the time to execute INORDER-TREE-WALK(x), exclusive of the time spent in recursive calls.

We use the substitution method to show that $T(n) = \Theta(n)$ by proving that $T(n) = (c + d)n + c$. For $n = 0$, we have $(c + d) \cdot 0 + c = c = T(0)$. For $n > 0$, we have

$$\begin{aligned} T(n) &= T(k) + T(n - k - 1) + d \\ &= ((c + d)k + c) + ((c + d)(n - k - 1) + c) + d \\ &= (c + d)n + c - (c + d) + c + d \\ &= (c + d)n + c \ , \end{aligned}$$

which completes the proof. ■

12.2 Querying a binary search tree

A common operation performed on a binary search tree is searching for a key stored in the tree. Besides the SEARCH operation, binary search trees can support such queries as MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR. In this section, we shall examine these operations and show that each can be supported in time $O(h)$ on a binary search tree of height h.

Searching

We use the following procedure to search for a node with a given key in a binary search tree. Given a pointer to the root of the tree and a key k, TREE-SEARCH returns a pointer to a node with key k if one exists; otherwise, it returns NIL.

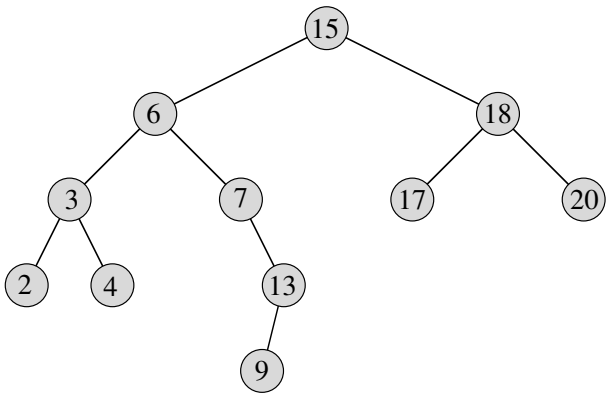

Figure 12.2 Queries on a binary search tree. To search for the key 13 in the tree, we follow the path $15 \to 6 \to 7 \to 13$ from the root. The minimum key in the tree is 2, which can be found by following *left* pointers from the root. The maximum key 20 is found by following *right* pointers from the root. The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.

```
TREE-SEARCH(x, k)
1  if x = NIL or k = key[x]
2     then return x
3  if k < key[x]
4     then return TREE-SEARCH(left[x], k)
5     else  return TREE-SEARCH(right[x], k)
```

The procedure begins its search at the root and traces a path downward in the tree, as shown in Figure 12.2. For each node x it encounters, it compares the key k with $key[x]$. If the two keys are equal, the search terminates. If k is smaller than $key[x]$, the search continues in the left subtree of x, since the binary-search-tree property implies that k could not be stored in the right subtree. Symmetrically, if k is larger than $key[x]$, the search continues in the right subtree. The nodes encountered during the recursion form a path downward from the root of the tree, and thus the running time of TREE-SEARCH is $O(h)$, where h is the height of the tree.

The same procedure can be written iteratively by "unrolling" the recursion into a **while** loop. On most computers, this version is more efficient.

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ key[x]
2        do if k < key[x]
3              then x ← left[x]
4              else  x ← right[x]
5  return x
```

Minimum and maximum

An element in a binary search tree whose key is a minimum can always be found by following *left* child pointers from the root until a NIL is encountered, as shown in Figure 12.2. The following procedure returns a pointer to the minimum element in the subtree rooted at a given node x.

```
TREE-MINIMUM(x)
1  while left[x] ≠ NIL
2        do x ← left[x]
3  return x
```

The binary-search-tree property guarantees that TREE-MINIMUM is correct. If a node x has no left subtree, then since every key in the right subtree of x is at least as large as $key[x]$, the minimum key in the subtree rooted at x is $key[x]$. If node x has a left subtree, then since no key in the right subtree is smaller than $key[x]$ and every key in the left subtree is not larger than $key[x]$, the minimum key in the subtree rooted at x can be found in the subtree rooted at $left[x]$.

The pseudocode for TREE-MAXIMUM is symmetric.

```
TREE-MAXIMUM(x)
1  while right[x] ≠ NIL
2        do x ← right[x]
3  return x
```

Both of these procedures run in $O(h)$ time on a tree of height h since, as in TREE-SEARCH, the sequence of nodes encountered forms a path downward from the root.

Successor and predecessor

Given a node in a binary search tree, it is sometimes important to be able to find its successor in the sorted order determined by an inorder tree walk. If all keys are distinct, the successor of a node x is the node with the smallest key greater than $key[x]$. The structure of a binary search tree allows us to determine the successor of a node without ever comparing keys. The following procedure returns the successor of a node x in a binary search tree if it exists, and NIL if x has the largest key in the tree.

```
TREE-SUCCESSOR(x)
1  if right[x] ≠ NIL
2     then return TREE-MINIMUM(right[x])
3  y ← p[x]
4  while y ≠ NIL and x = right[y]
5        do x ← y
6           y ← p[y]
7  return y
```

The code for TREE-SUCCESSOR is broken into two cases. If the right subtree of node x is nonempty, then the successor of x is just the leftmost node in the right subtree, which is found in line 2 by calling TREE-MINIMUM($right[x]$). For example, the successor of the node with key 15 in Figure 12.2 is the node with key 17.

On the other hand, as Exercise 12.2-6 asks you to show, if the right subtree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. In Figure 12.2, the successor of the node with key 13 is the node with key 15. To find y, we simply go up the tree from x until we encounter a node that is the left child of its parent; this is accomplished by lines 3–7 of TREE-SUCCESSOR.

The running time of TREE-SUCCESSOR on a tree of height h is $O(h)$, since we either follow a path up the tree or follow a path down the tree. The procedure TREE-PREDECESSOR, which is symmetric to TREE-SUCCESSOR, also runs in time $O(h)$.

Even if keys are not distinct, we define the successor and predecessor of any node x as the node returned by calls made to TREE-SUCCESSOR(x) and TREE-PREDECESSOR(x), respectively.

In summary, we have proved the following theorem.

Theorem 12.2

The dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR can be made to run in $O(h)$ time on a binary search tree of height h. ■

12.3 Insertion and deletion

The operations of insertion and deletion cause the dynamic set represented by a binary search tree to change. The data structure must be modified to reflect this change, but in such a way that the binary-search-tree property continues to hold. As we shall see, modifying the tree to insert a new element is relatively straightforward, but handling deletion is somewhat more intricate.

Insertion

To insert a new value v into a binary search tree T, we use the procedure TREE-INSERT. The procedure is passed a node z for which $key[z] = v$, $left[z] = \text{NIL}$, and $right[z] = \text{NIL}$. It modifies T and some of the fields of z in such a way that z is inserted into an appropriate position in the tree.

```
TREE-INSERT(T, z)
 1  y ← NIL
 2  x ← root[T]
 3  while x ≠ NIL
 4        do y ← x
 5           if key[z] < key[x]
 6              then x ← left[x]
 7              else  x ← right[x]
 8  p[z] ← y
 9  if y = NIL
10     then root[T] ← z                ▷ Tree T was empty
11     else  if key[z] < key[y]
12              then left[y] ← z
13              else  right[y] ← z
```

Figure 12.3 shows how TREE-INSERT works. Just like the procedures TREE-SEARCH and ITERATIVE-TREE-SEARCH, TREE-INSERT begins at the root of the tree and traces a path downward. The pointer x traces the path, and the pointer y is maintained as the parent of x. After initialization, the **while** loop in lines 3–7 causes these two pointers to move down the tree, going left or right depending on the comparison of $key[z]$ with $key[x]$, until x is set to NIL. This NIL occupies the position where we wish to place the input item z. Lines 8–13 set the pointers that cause z to be inserted.

Like the other primitive operations on search trees, the procedure TREE-INSERT runs in $O(h)$ time on a tree of height h.

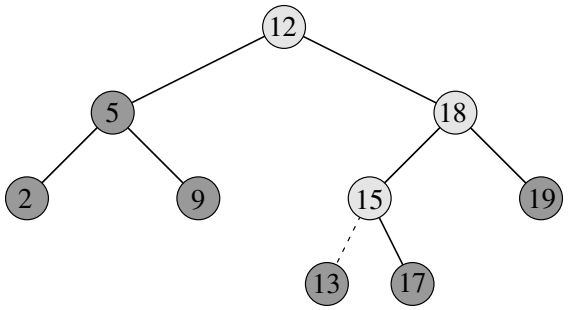

Figure 12.3 Inserting an item with key 13 into a binary search tree. Lightly shaded nodes indicate the path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item.

Deletion

The procedure for deleting a given node z from a binary search tree takes as an argument a pointer to z. The procedure considers the three cases shown in Figure 12.4. If z has no children, we modify its parent $p[z]$ to replace z with NIL as its child. If the node has only a single child, we "splice out" z by making a new link between its child and its parent. Finally, if the node has two children, we splice out z's successor y, which has no left child (see Exercise 12.2-5) and replace z's key and satellite data with y's key and satellite data.

The code for TREE-DELETE organizes these three cases a little differently.

```
TREE-DELETE(T, z)
 1  if left[z] = NIL or right[z] = NIL
 2     then y ← z
 3     else y ← TREE-SUCCESSOR(z)
 4  if left[y] ≠ NIL
 5     then x ← left[y]
 6     else x ← right[y]
 7  if x ≠ NIL
 8     then p[x] ← p[y]
 9  if p[y] = NIL
10     then root[T] ← x
11     else if y = left[p[y]]
12             then left[p[y]] ← x
13             else right[p[y]] ← x
14  if y ≠ z
15     then key[z] ← key[y]
16          copy y's satellite data into z
17  return y
```

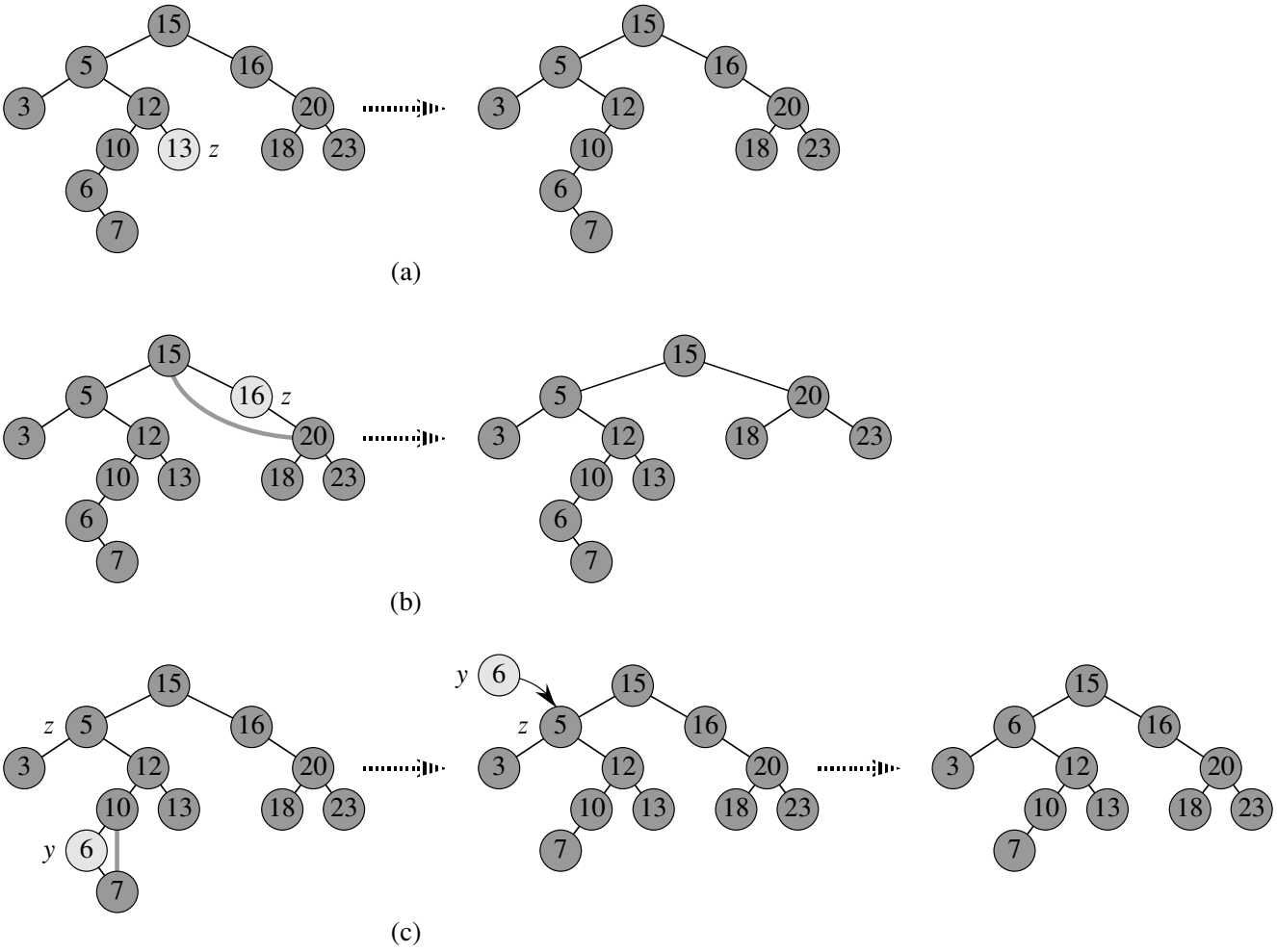

Figure 12.4 Deleting a node z from a binary search tree. Which node is actually removed depends on how many children z has; this node is shown lightly shaded. **(a)** If z has no children, we just remove it. **(b)** If z has only one child, we splice out z. **(c)** If z has two children, we splice out its successor y, which has at most one child, and then replace z's key and satellite data with y's key and satellite data.

In lines 1–3, the algorithm determines a node y to splice out. The node y is either the input node z (if z has at most 1 child) or the successor of z (if z has two children). Then, in lines 4–6, x is set to the non-NIL child of y, or to NIL if y has no children. The node y is spliced out in lines 7–13 by modifying pointers in $p[y]$ and x. Splicing out y is somewhat complicated by the need for proper handling of the boundary conditions, which occur when $x = \text{NIL}$ or when y is the root. Finally, in lines 14–16, if the successor of z was the node spliced out, y's key and satellite data are moved to z, overwriting the previous key and satellite data. The node y is returned in line 17 so that the calling procedure can recycle it via the free list. The procedure runs in $O(h)$ time on a tree of height h.

In summary, we have proved the following theorem.

Theorem 12.3

The dynamic-set operations INSERT and DELETE can be made to run in $O(h)$ time on a binary search tree of height h. ■