2. Fundamental Data Structures

Data Structures is a term synonymous with data containers (or memory objects) used for storage of problem data for processing by an algorithm. Data structures include arrays, lists, trees, stacks, queues, hash tables, etc. It turns out that, most of the time, the algorithm's efficiency is directly related to the kind of data structure employed by the algorithm. On the other hand, for a certain algorithm, it may be possible to identify (and define) certain operations that are executed frequently, and whose order of running time would affect the algorithm's order of running time. Thus, it may be best to devise customized data structures to efficiently implement the required operations.

From a programming viewpoint, it is best to view (and implement) any particular data structure as an *abstract data type* (ADT). Similar to the concept of a data type, an ADT represents a well-defined set of operations on data, and it provides an abstraction that allows us to separate the interface from various implementations. In plain English, an ADT is specified via two separate pieces of program code:

1. An interface specification that provides the declarations (i.e., method headings) of various public methods.
2. An implementation whose details are not visible outside of the ADT.

Furthermore, by their nature as data containers, the various data structures possess common properties that ought to be abstracted into some predefined interfaces. This is the approach used to implement data structures in program libraries used with Java and C#. For example, the .NET Framework provides classes for various data structures, including queues and hash tables, as part of the *System.Collections* namespace. These classes implement a number of predefined interfaces and follow standard guidelines that allow for automatic memory management and synchronization (when a particular data-structure object is shared by multiple threads). We will discuss these programming issues in details in Chapter 11.

This chapter concentrates on the algorithmic aspects related to the efficient implementation of some important data structures and their applications. The following topics are discussed.

- Graphs, trees and their associated data structures
- Priority queues and their application to sorting and discrete event simulation
- Disjoint-sets data structure and its application to finding cycles in graphs
- Hashing and its application to text search
- Height-balanced trees

2.1 Graphs and Trees

A graph is a set of vertices (nodes, points) connected by edges (arcs, lines). More formally, we define two types of graphs: *directed graphs* and *undirected graphs*.

A *directed graph* $G=(V,E)$ consists of a set of vertices (nodes) V, and a set of edges (arcs) E, where an edge is an *ordered pair* of vertices. A graph is best visualized by a diagram (graphical representation) like that shown in Figure 2.1. For this graph, the set of vertices is $V=\{a, b, c, d\}$, and the set of edges is $E = \{ (a,b), (a,c), (c,b), (b,d), (d,c) \}$.

A graph is *undirected* if every edge is an *unordered pair* of vertices. In other words, for any two vertices x and y, the edge (x,y) is the same as the edge (y,x). Note that in the graphical representation, an undirected edge (x,y) is simply drawn as a line (without arrows) connecting x with y.

A graph is normally used to model some kind of relationship among objects. For example, we can have a set of courses and the prerequisite relationship among them. In this case, each course is represented as a vertex, and we draw a directed edge from course x to course y if course x is a prerequisite for course y. Undirected graphs are appropriate for representing *symmetric* relations. Consider the relation of human friendship among the students of a university. If a student x is a friend of another student y, then, clearly, y is a friend of x. Therefore, we would use an undirected graph to model the friendship relation.

Most of the time it suffices to deal with *simple graphs*; a graph is *simple* if it has no *parallel edges* and no *self-loops*. Two edges are parallel if they share the same source and end vertices. A self-loop is an edge from a vertex to itself.

A *weighted graph* is where each edge is assigned a real number representing the weight (cost) of the edge.

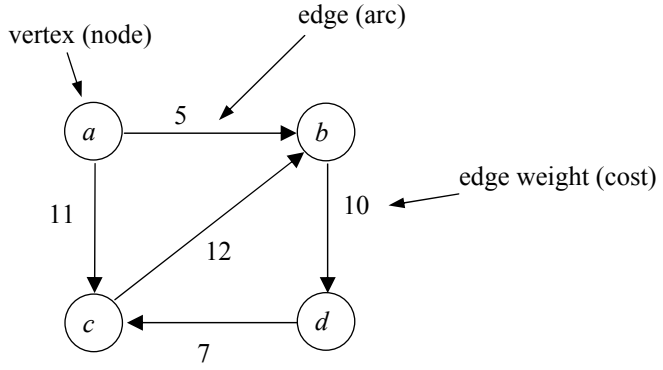


Figure 2.1 A directed weighted graph and some associated terminology.

Two vertices are said to be *adjacent* if they are connected by an edge. An edge is said to be *incident* to its endpoint vertices. For an undirected graph, we define the *degree* of a vertex v as the number of edges incident into v. For a directed graph, we define the *out-degree* [*in-degree*] of a vertex v as the number of edges outgoing from [incoming into] v. For example, for the graph in Figure 2.1, *in-degree*$(a) = 0$; *out-degree*$(a) = 2$.

2.1.1 Elementary Graph Definitions

An undirected graph is *complete* if every pair of vertices is connected by an edge. The complete graph on n vertices is denoted as K_n and has $\binom{n}{2} = n(n-1)/2$ edges.

A graph $G=(V,E)$ is *bipartite* if there exists a way to partition the set of vertices V into two disjoint sets V_1 and V_2, such that every edge in E has one vertex in V_1 and the other vertex in V_2.

A *complete bipartite* graph, denoted by $K_{m,n}$, consists of $m+n$ vertices, with each of the first m vertices connected to all of the other n vertices, and no other vertices.

A path from vertex v_1 to vertex v_k is a sequence of vertices $v_1, v_2, \ldots, v_k$ where $(v_i,v_{i+1}) \in E$. For example, in reference to the graph shown in Figure 2.1, the sequence *a*, *c*, *b* is a path, whereas the sequence *b*, *d*, *a* is not. The *length* of a path is the number of edges along that path.

An undirected graph is *connected* if for every pair of vertices (u,v), there is at least one path from u to v; otherwise, the graph is disconnected. On the other hand, a directed graph can be *strongly connected*, *weakly connected* or *disconnected*. The graph is *strongly connected* if for every pair of vertices u and v, there is a path from u to v, and a path from v to u. The graph is *weakly connected* if its associated undirected graph (i.e., view every edge as undirected) is connected.

A *simple path* is a path that contains no repeated vertices. A *cycle* (*circuit*) is a path from a vertex to itself. For an undirected graph, a cycle must have at least three edges, to avoid counting every edge as a cycle. A *simple cycle* is a cycle with no repeated vertices except for the first and last vertex.

A *Hamiltonian cycle* (*circuit*) is a simple cycle that includes all the vertices in the graph. On the other hand, an *Euler cycle* (*circuit*) is a cycle that includes every edge in the graph exactly once. (Note: *Euler* is pronounced "*oiler*".) Interestingly enough, there is a simple and fast algorithm for determining if a graph has an Euler cycle, but no such algorithm exists to determine if a graph has a Hamiltonian cycle. The latter problem is known to be an *NP-complete* problem — this means that it is unlikely that there will be an efficient (polynomial-time) algorithm to solve this problem for general graphs.

A graph $G'=(V',E')$ is a *subgraph* of a graph $G=(V,E)$, if $V' \subseteq V$, $E' \subseteq E$, and for every edge $(u,v) \in E'$, u and v must be contained in V'. The *complement* of a graph $G=(V,E)$ is a graph $G'=(V', E')$, where $V'=V$ and $(u,v) \in E'$ if and only if $(u,v) \notin E$.

Two graphs $G=(V,E)$ and $G'=(V',E')$ are *isomorphic* if there is a mapping f: $V \rightarrow V'$, where f is a one-to-one correspondence mapping, and for all $u,v \in V$, $(u,v) \in E$ if and only if $(f(u),f(v)) \in E'$.
Graph isomorphism is known to be a difficult problem; yet, it is not known if it is NP-complete.

2.1.2 Trees

Although trees are special kind of graphs, they are of considerable importance on their own. A tree is the right model for a hierarchical structure (or nested containment). For example, a university consists of a number of colleges; each college, in turn, consists of a number of departments and a department hosts a number of academic programs. Figure 2.2 shows the graph (tree) for some university U_x. The tree succinctly tells that U_x hosts two colleges (C_1, C_2), three departments (D_1-D_3) and four programs (P_1-P_4), with the edges indicating containment among these objects.

Because the containment relation is asymmetric (i.e., it is a *parent–child* kind of relation), an indication of the direction must be explicitly present and consistently used. We can: (1) have all edges directed toward parents or (2) have all edges directed away from parents, or (3) use undirected edges but show children at a level lower than that of their parents.

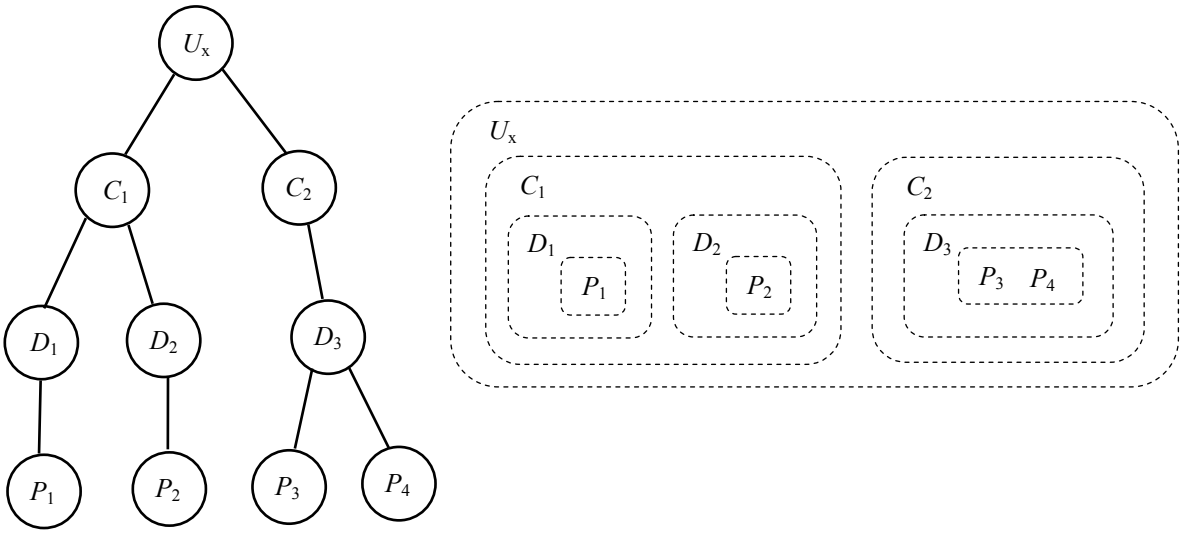


Figure 2.2 A tree models the hierarchical structure (nested containment) of a university.

Viewed as an undirected graph, the tree of Figure 2.2 has a distinctive characteristic: It is connected and has no cycles. This will be true of most hierarchical structures that we encounter in practice; therefore, we standardize on the following definition for a tree.

Definition of a Tree. A tree is a connected undirected graph without cycles.

Trees possess some interesting properties, including the following:

- A tree with n nodes, has $n-1$ edges.
- The removal of any edge disconnects the tree.
- The addition of an edge (between two existing vertices) creates a cycle.
- A tree with 2 or more nodes has at least two leaves — a *leaf* is defined to be any vertex of degree=1.

Because of their importance, trees have evolved with a terminology of their own. Terms such as *root*, *leaf*, *parent*, *child*, etc. are best understood with the aid of a visual representation as that sketched in Figure 2.3.

The tree of Figure 2.2 is an example of a *rooted tree*. A rooted tree is one where a particular vertex has been designated as a root (parentless) and all edges are directed (either away from the root or toward the root — note that the direction is implied if children are drawn at a level lower to their parent).

Note that in a rooted tree, the child subtrees of the root are themselves rooted trees (subtrees). Thus, we can use the following (recursive) definition for a rooted tree.

Definition. Rooted Tree: A rooted tree is either empty or consists of a node that has child rooted trees.

Among rooted trees, the class of binary trees is of special importance.

Definition. Binary Tree: A binary tree is a rooted tree where every node has at most two children.

A *binary search tree* is just one type of a binary tree. This is a binary tree with keys (numbers or any type of ordered objects) stored at the nodes and is consistent with the following definition.

Definition. Binary Search Tree: A binary search tree is a binary tree where for every node x, keys in the left subtree of $x \leq key(x) <$ keys of the right subtree of x.

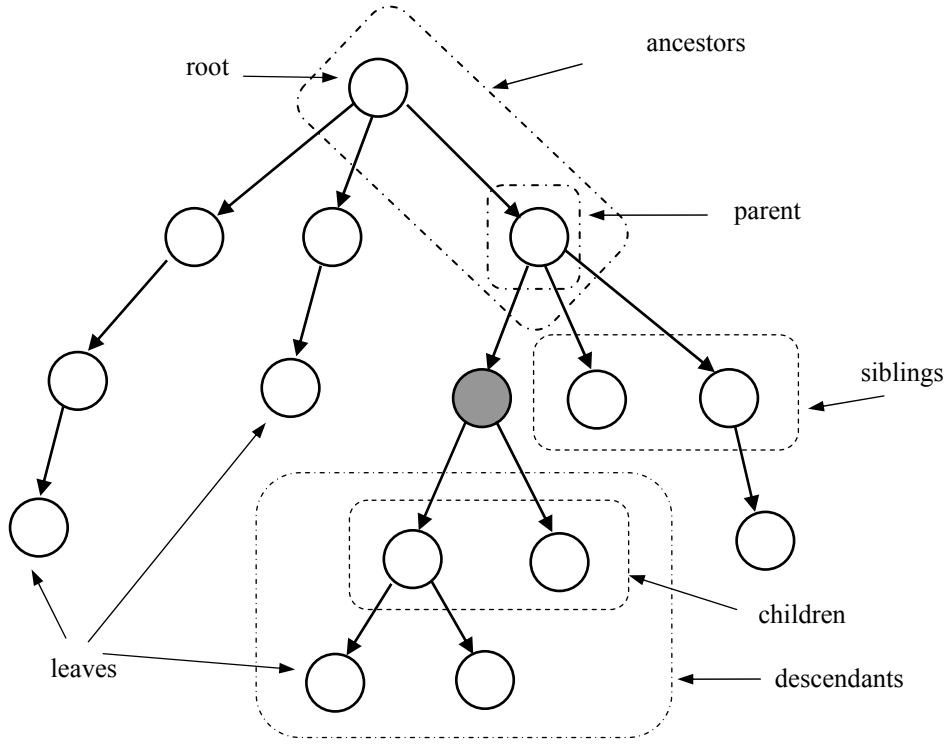


Figure 2.3 Terminology for trees; labels (except for *root* and *leaves*) are from the viewpoint of the shaded node.

2.1.3 Data Structures for Graphs

Various data structures are used to represent graphs. For a particular algorithm, the choice of which data structure to use depends largely on the type of data-access operations performed by the algorithm. Common to all graph representations is that the vertices of a graph are normally identified as 1, 2, …, n. This is done regardless of the labels (or the "real" objects) associated with the vertices because, as far as graph algorithms are concerned, it is more efficient to manipulate integers than any other data type. If needed, information about graph vertices can be stored in an auxiliary array (of type *string* or *object*) *VertexInfo*[1..n] where *VertexInfo*[i] stores information for vertex i. However, this structure should be accessed only for input-output purposes.

Edge-List Structure

The simplest data structure to represent (store) a graph is to maintain its edges in a list. Conceptually, the graph is represented as a list of tuples:

$G = \{ (u,v) \}$ [or, if G is weighted, $G = \{ (u,v,w_{uv}) \}$], for all $(u,v) \in E$.

For an undirected graph it suffices to store an edge (u,v) once (the edge (v,u) is implied). To tell the type of graph, we can use a *directed* flag (of type Boolean) as part of the structure. The list can be represented compactly as an array. However, to permit addition and deletion of edges, it is better to use a linked list. Figure 2.4 shows the edge-list representation. If an array is used, then for a graph with m edges, we can use an $m \times 2$ array (or $m \times 3$ array if the graph is weighted) *Edges*[1..m,0..1]; that is, if the i-th edge is (a,b), then we have *Edges*[i,0]=a and *Edges*[i,1]= b. The edge-list structure is usually complemented with information about the size of the graph, including the number of vertices (*VertexCount*) and the number of edges (*EdgeCount*). Note that *VertexCount* can help in determining which vertices are isolated (you can also indicate this by adding a dummy edge $(v,0)$ for every isolated vertex v).

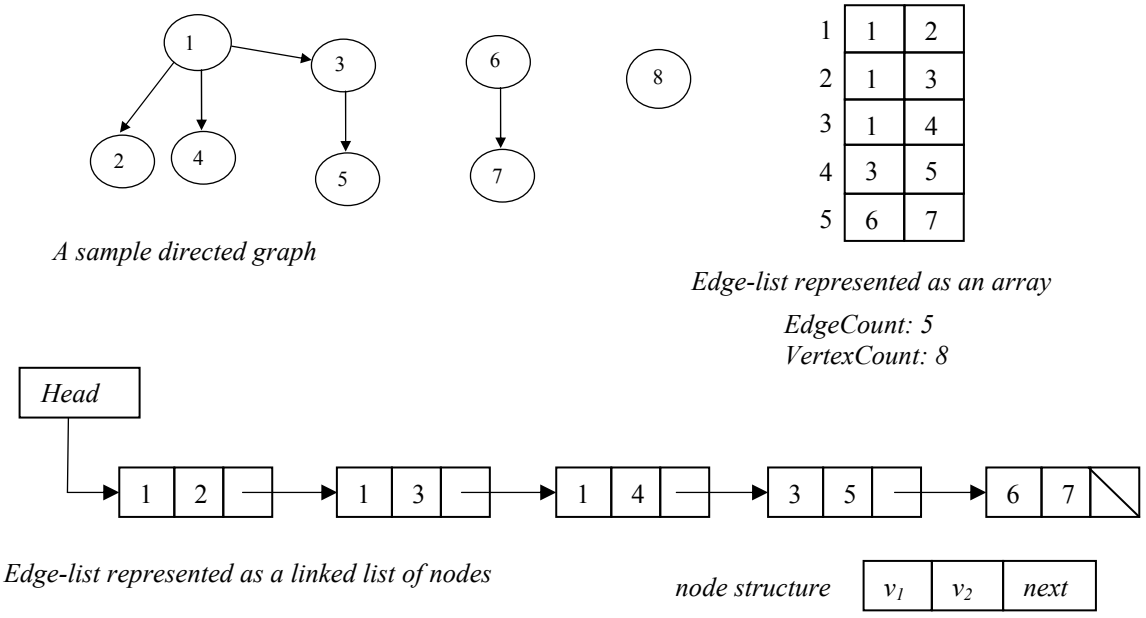


Figure 2.4 Graph representation using an edge-list structure.

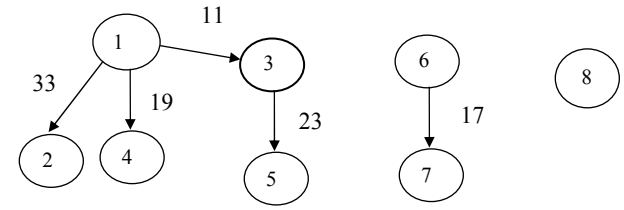
Using the edge-list structure, we can loop through all edges in O(|*E*|) time. Insertion of an edge can be done in O(1) time (e.g., always insert at the front of the list) but deletion takes O(|*E*|) time because it requires locating the edge first. Thus, in applications where deletion is frequent, it is better to replace the linked-list by some dynamic-set structure such as a hash table or a binary search tree.

The edge-list representation is space efficient and it is a good choice for external storage (as a text file) and graph construction. Some algorithms, e.g., *Kruskal's MST* algorithm, expect the graph to be given in this form.

An important operation needed by many graph algorithms (e.g., graph traversal algorithms such as depth-first search or breadth-first search) is to access all edges incident to a given vertex. Using the edge-list structure, this operation takes O(|*E*|) time because all edges have to be inspected. Thus, if this operation needs to be done for all (or most) vertices but one vertex at a time, it would take O(|*V*|×|*E*|) time. In such cases, it is more efficient to use the *adjacency-lists* structure.

Adjacency-Lists Structure

The adjacency-lists structure, as illustrated by the example graph of Figure 2.5, maintains several lists, one per vertex. (Note: This structure is more appropriately named as an *array of adjacency lists*). For a graph with *n* vertices, we utilize an array *AdjList*[1..*n*], where *AdjList*[*v*] is a reference (a pointer) to the list that contains all vertices adjacent to *v* (i.e., all vertices *w* where (*v*,*w*) is an edge). Each element on the list for vertex *v* specifies some vertex *w* and thus it corresponds to the edge whose *from*-vertex is *v* and whose *to*-vertex is *w*. This representation allows a direct and quick access to all edges incident to a vertex, which speeds up many graph algorithms. For any one vertex, the order of running time for accessing all edges incident to a vertex is O(|*List*(*v*)|) where |*List*(*v*)| is the size of the list associated with vertex *v*. Therefore, to do this operation for all vertices implies a running time = O($\sum_{all\ vertice,\ v}$ |*List*(*v*)|) = O(|*E*|). Of course, if |*E*| is much smaller than |*V*|, it is more accurate to state the running time as O(|*V*| +|*E*|).



A sample directed weighted graph

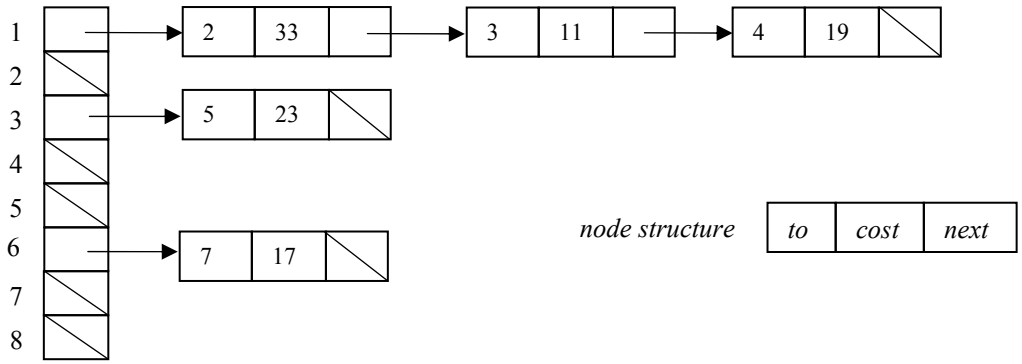


Figure 2.5 Graph representation using an adjacency-lists structure.

Adjacency-Matrix Structure

The adjacency matrix, $A[1..n,1..n]$, is an $n \times n$ matrix where the entry $A[i,j]$ is set to 1 if there is an edge from vertex i to vertex j; otherwise, $A[i,j]$ is set to 0. The adjacency matrix permits a constant-time test of whether there is an edge (a,b) for any arbitrarily chosen vertices a and b. For an undirected graph, the adjacency matrix is symmetric in relation to the main diagonal; that is, for all i,j in $[1,n]$, $A[i,j]=A[j,i]$. For a weighted graph, we can let $A[i,j]$ be the weight (cost) of the edge (i,j). In this case, the matrix is appropriately named the *adjacency cost-matrix*. Figure 2.6 shows an example.

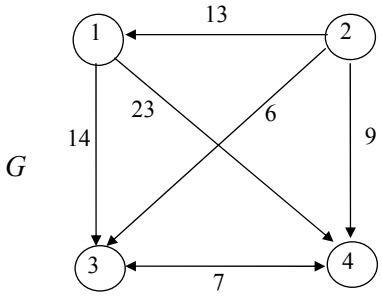
There are few graph algorithms where the adjacency matrix is most suited such as *Floyd's all-pairs shortest-paths algorithm*. In general, the adjacency matrix is less efficient than adjacency lists for certain operations. For example, to find the edges incident to a particular vertex, one must examine $|V|$ entries. Also, for a *sparse graph* (a graph is considered *sparse* if $|E| \approx |V|$, because normally, $|E| \approx |V|^2$), using an adjacency matrix is wasteful of space since most of the matrix entries are zeros.

Incidence-Matrix Structure

Some authors define the incidence matrix of a graph as a (0,1)-matrix which has a row for each vertex and column for each edge, where the entry (i,j) is set to 1 iff vertex i is incident to edge j. Clearly, such representation is wasteful of space. Alternatively, for our purpose, we define the incidence matrix as the matrix $IM[1..n,0..(n-1)]$ as follows (see Figure 2.6):

- $IM[i,0]$ is set to the degree (or out-degree, in case the graph is directed) of vertex i, and
- The i-th row $IM[i,1..k]$ lists the vertices that are adjacent to i (i.e., $IM[i,j]=x$ iff (i,x) is an edge).

Thus, the incidence matrix is more or less the adjacency-lists representation but without using pointers. It is appropriate to use if the graph is unchanged after its initial construction. Furthermore, the incidence matrix can be represented more compactly as a *jagged matrix* (an array of arrays).



$$\begin{pmatrix} 0 & \infty & 14 & 23 \\ 13 & 0 & 6 & 9 \\ \infty & \infty & 0 & 7 \\ \infty & \infty & 7 & 0 \end{pmatrix}$$

Adjacency cost-matrix

$$\begin{pmatrix} 2 & 3 & 4 & 0 \\ 3 & 1 & 3 & 4 \\ 1 & 4 & 0 & 0 \\ 1 & 3 & 0 & 0 \end{pmatrix}$$

Incidence matrix

$$\begin{pmatrix} 2 & (3,14) & (4,23) & 0 \\ 3 & (1,13) & (3,6) & (4,9) \\ 1 & (4,7) & 0 & 0 \\ 1 & (3,7) & 0 & 0 \end{pmatrix}$$

Incidence matrix with edge weights

Figure 2.6 A weighted directed graph G, its adjacency-matrix representation, and its incidence-matrix representation.

A Class for Graphs

Listing 2.1 shows the basic structure of a class for graphs. Because there is no single representation that is suited for all graph algorithms, it is best to allow the user to select the appropriate representation for the algorithm at hand. Thus, the constructor for our *Graph* class is given by the following:

```
public Graph(GraphStructure gs, bool Directed, bool Weighted, bool Bipartite)
```

The first parameter is to select the graph structure. Other parameters are to decide the attributes of the graph: *directed*, *weighted*, *bipartite*. The graph is then constructed through program-code (such as the call to *GenRandomGraph*) or read (as edge-list kind of information) from a text file using the call *ReadGraph* (not shown here). Once the graph is created, the appropriate structure among (*EdgeList*, *AdjList*s, *AdjMatrix*, *AdjCostMatrix*) will be available for processing.

Here is a code fragment to create a random undirected graph with 500 vertices and then print all edges (u,v) where u=2v:

```
Graph graph = new Graph(GraphStructure.AdjLists, false, false, false);
graph.GenRandomGraph(500, 0.1f); // a random graph where probability Of an edge is 10%

for(int i = 1; i <= graph.VertexCount; i++)
  foreach (AdjListEntry ent in graph.AdjLists[i])
      if (ent.vertex== 2*i) Console.WriteLine("({0},{1})", i, ent.vertex);
```

The source code for the class is maintained as a text file "*graph.cs*" and is incorporated into various programming projects including graph coloring, bipartite matching and others, which can be found at this book's website.

```
public enum GraphStructure {EdgeList, AdjLists, AdjMatrix, AdjCostMatrix}
class Edge // used by EdgeList Rep.
{  internal int from, to;
   internal float weight;
   internal string label;
   internal Edge(int from, int to, float weight, string label)
   { this.from = from;  this.to = to; this.weight = weight; this.label = label; }
}

struct AdjListEntry // used by AdjList Rep.
{  internal int vertex;
   internal string label;
   internal AdjListEntry(int Vertex, string Label){ this.vertex = Vertex; this.label = Label; }
}

public class Graph
{  public int VertexCount = 0;
   public int LeftVertexCount = 0; // count of left-side vertices if the graph is bipartite
   public ArrayList[] AdjLists; // For AdjacencyLists Rep.
   public ArrayList EdgeList; // For EdgeList Rep.
   public bool[,] AdjMatrix; // For Adjacency Matrix Rep.
   public float[,] AdjCostMatrix; // For Adjacency Cost-Matrix Rep.

   bool Directed = false; bool Weighted = false; bool Bipartite = false;
   GraphStructure Structure;

 public Graph(GraphStructure gs, bool Directed, bool Weighted, bool Bipartite)
 { this.Structure = gs; this.Directed = Directed; this.Weighted = Weighted;
   this.Bipartite = Bipartite; }

 public void GenRandomGraph(int n, float p)
 {  Random rand = new Random((int) DateTime.Now.Ticks);
    VertexCount= n;
    ArrayList TempList = new ArrayList();

    for(int i = 1; i < n; i++)
       for(int j = i+1; j <= n; j++)
       {  float rnd = (float) rand.NextDouble();
          if (rnd <= p) { Edge e = new Edge(i,j,0,""); TempList.Add(e); }
       }
    if (Structure == GraphStructure.EdgeList) { EdgeList = TempList; return; }
    if (Structure == GraphStructure.AdjLists)  BuildAdjList(TempList);
    if (Structure == GraphStructure.AdjMatrix) BuildAdjMatrix(TempList);
 }

 private void BuildAdjList(ArrayList list)
 {  AdjLists = new ArrayList[VertexCount + 1];
    for(int i = 1; i <= VertexCount; i++)
        AdjLists[i] = new ArrayList();

    foreach (Edge e in list)
    {  AdjListEntry entry = new AdjListEntry(e.to, "");
       AdjLists[e.from].Add(entry);
       if (!Directed)
       { entry = new AdjListEntry(e.from, ""); AdjLists[e.to].Add(entry); }
    }
 }
}
```

Listing 2.1 Basic structure for our *Graph* class.

2.2 Priority Queues

Definition. Priority Queue: A priority queue is a data structure that stores a collection of items with keys (see the following Note1) and supports two basic operations: *Insert*: insert a new item, and *DeleteMax*: delete (and return) the item with the largest key.

Note 1: This is a prevalent misuse of terminology; throughout this section, we use the term *key* to mean *priority*. The misuse can be attributed to the fact that this data-structure was created in the first place for efficient sorting of *keys*. In a proper implementation of the priority-queue structure, one must distinguish between the keys used to identify the items and item priorities. We consider the implementation details for the priority-queue structure in Section 11.5.3.

Note 2: The preceding definition assumes a *maximizing priority queue*. A *minimizing priority queue* supports *DeleteMin* (instead of DeleteMax), which deletes the item with the smallest key.

Other operations that are normally supported by a priority queue include *FindMax*, which returns the maximum element without deleting it, and *ChangeKey*, which changes the priority of an item (internally, we assume the existence of some kind of index structure to locate the item in the queue from its *real* key). There are numerous options for implementing priority queues. One implementation is to use an *unordered array* as the underlying data structure. The deletemax operation is implemented by scanning the array to find the maximum, then exchanging the maximum item with the last item of the array, and then returning the maximum after decrementing the count representing the number of elements in the array.

For a priority queue, it is easy to find an implementation where either insert or deletemax operations takes constant time — see Table 2.1, but it is more difficult to find an implementation where both operations run fast. A popular and efficient implementation for a priority queue data-structure is to use a *heap data-structure*. Using a heap, both insert and deletemax operations take $\Theta(\log n)$ time.

	Insert	***DeleteMax***	***FindMax***	***ChangeKey***
unordered array	O(1)	O(n)	O(n)	O(1)
ordered array	O(n)	O(1)	O(1)	O(n)
unordered list	O(1)	O(n)	O(n)	O(1)
ordered list	O(n)	O(1)	O(1)	O(n)
heap	O($\log n$)	O($\log n$)	O(1)	O($\log n$)

Table 2.1 Worst-case time of Priority Queue operations for various implementations.

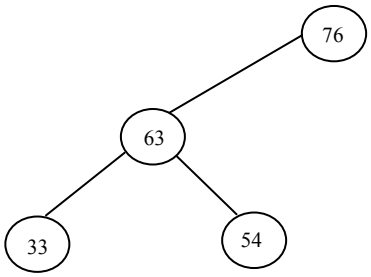
Definition. The Heap Data-Structure: A heap data-structure (or simply, a heap) is a binary tree that satisfies the following properties.

1. The *complete-tree* property: The tree is (almost) complete. This means that all levels in the tree, except possibly the lowest level, are complete (i.e., level i has 2^i nodes), and on the lowest level the nodes appear in a left-to-right order.

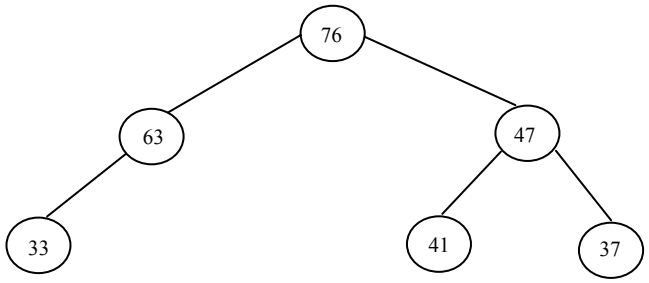
2. The *heap-order* property: For *max-heap*: For every node x, $key(x) \geq$ keys of left child and right child of x.

For *min-heap*: For every node x, $key(x) \leq$ keys of left child and right child of x.

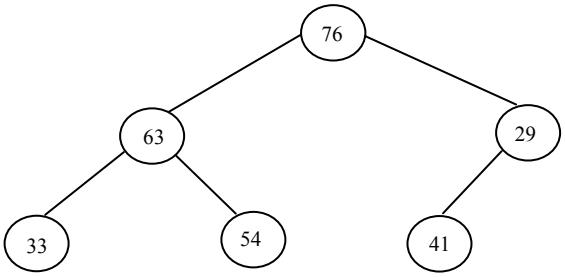
Example 2.1 Figure 2.7 shows examples of binary trees that are not heaps.



Not a heap because a level is started before the previous level is completed



Not a heap because the lowest level is not filled in left-to-right order



Not a heap because the heap-order property is violated at node with key = 29 (i.e., $29 \geq 41$ is false)

Figure 2.7 Examples of binary trees that are not heaps.

The heap-order property implies that in a max-heap, for every node x, $key(x) \geq keys$ of left and right subtrees of x. Similarly, in a min-heap, for every node x, $key(x) \leq keys$ of left and right subtrees of x. Thus, in a max-heap [min-heap] the largest [smallest] element is at the root. Also, in a max-heap [min-heap], for any node x, all the nodes on a path from x to a leaf node are ordered in decreasing [increasing] order by key. Finally, a heap tree is an example of a *height-balanced tree* whose height is logarithmic in the number of nodes.

Lemma 2.1 *The height of a heap tree containing n nodes is* $\lfloor log\ n \rfloor$.

Proof: If the heap tree is of height h, it consists of the levels: 0, 1, …, h. Levels 0 through h–1 have the maximum number of nodes totaling $2^0+2^1+\ldots+2^{h-1} = 2^h - 1$. The lowest level h has between 1 and 2^h nodes. Thus, the number of nodes n is such that:

$$2^h - 1 + 1 \le n \le 2^h - 1 + 2^h \Rightarrow 2^h \le n < 2^{h+1} \Rightarrow h \le log\ n < h+1 \text{ (and because } h \text{ is an integer)} \Rightarrow h = \lfloor log\ n \rfloor .$$

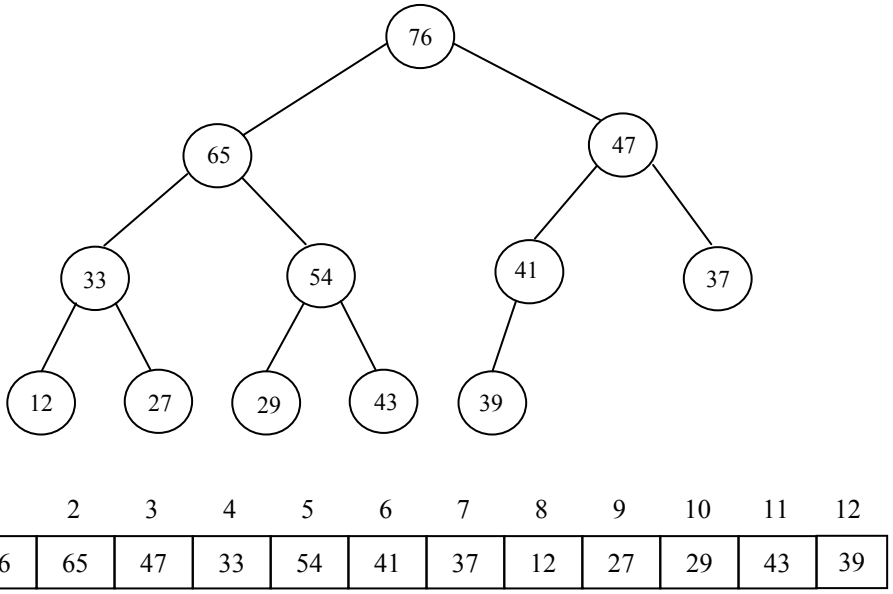


Figure 2.8 A heap and its array representation.

Array-Based Implementation for a Heap

A heap, being a binary tree, can be implemented as a linked structure (i.e., using either explicit pointers or objects — see Section 11.5). However, an array-based implementation is often more effective since it does away with the overhead (runtime manipulation and memory-space) associated with pointers.

In the array-based implementation of a heap with n elements, we use an array $A[1..n]$ and the following rules:

- The root is stored at location (index) 1.
- If a node is stored at location i then its left child is stored at $2i$, and its right child is stored at $2i+1$.

Figure 2.8 shows a heap and its array representation.

Note that the complete-tree property ensures that in the array A representing a heap, all slots $A[1]$ to $A[n]$ are filled and that there are no gaps (empty slots) between elements. Furthermore, in the array A representing a heap, the parent of $A[j]$ is $A[j/2]$. Thus, the parent of $A[n]$ (the last node) is $A[n/2]$.

Basic Heap Operations: Walkup, Walkdow

A basic operation on a heap is to restore the heap-order property once the key of a particular heap element is modified. Let x denote the node whose key has been modified. It turns out that there are three possibilities (assuming max-heap):

1. The heap-order property is still satisfied at x. In this case, do nothing.
2. $key(x)$ > key of parent of x. In this case, *Walkup* x.
3. $key(x)$ < key of left (or right) child of x. In this case, *Walkdown* x.

The walkup process essentially moves the modified element x up the tree via a series of element swaps until the heap-order property is restored.

Example 2.2 Figure 2.9 illustrates the walkup process. The figure shows a situation where the key of $A[5]$ is modified to 80. We need to execute *Walkup*(A,5) because $A[5] > A[5/2]$. Walkup starts by swapping x with its parent. Note that following this first swap, the subtree now rooted at location 5 is a heap; however, there is the possibility that the element x, brought to location 2, needs to be walked up further. Therefore, we again compare x with $A[1]$ and see that a swap is needed. In general, the process of swapping x with its parent continues as long as x is greater than its parent or x becomes the root node.

On the other hand, the walkdown process can be viewed as the reverse of walkup. It essentially moves the modified element x down the tree via a series of element swaps until the heap-order property is restored.

Example 2.3 Figure 2.10 illustrates the walkdown process. The figure shows a situation where the key of $A[2]$ is modified to 23. We need to execute *Walkdown*(A,2), since $A[2]$ < either $A[4]$ or $A[5]$. The first swap involves x and the child (either left child or right child) of x having the largest key. Thus, we swap $A[2]$ with $A[5]$. Next we swap $A[5]$ with the largest of $A[10]$, $A[11]$, and so on. In general, the process of swapping x with its largest child continues as long as x is smaller than its largest child, or x becomes a leaf node.

Listing 2.2 shows walkdown and walkup algorithms for the array-based implementation of a max-heap.

Running-Time Analysis of Walkdown (and Walkup)

By examining the code for walkdown given in Listing 2.2, we see that in the while-loop, walkdown executes one iteration (at most two element comparisons) and moves down the tree by one level. Thus, the number of iterations (or comparisons) executed by walkdown on a heap of size n is proportional to the height of the heap tree, which is $O(\log n)$. Hence, the worst-case time complexity of walkdown on a heap of size n is $O(\log n)$. Similar analysis applied to walkup shows that the worst-case time complexity of walkup on a heap of size n is $O(\log n)$.

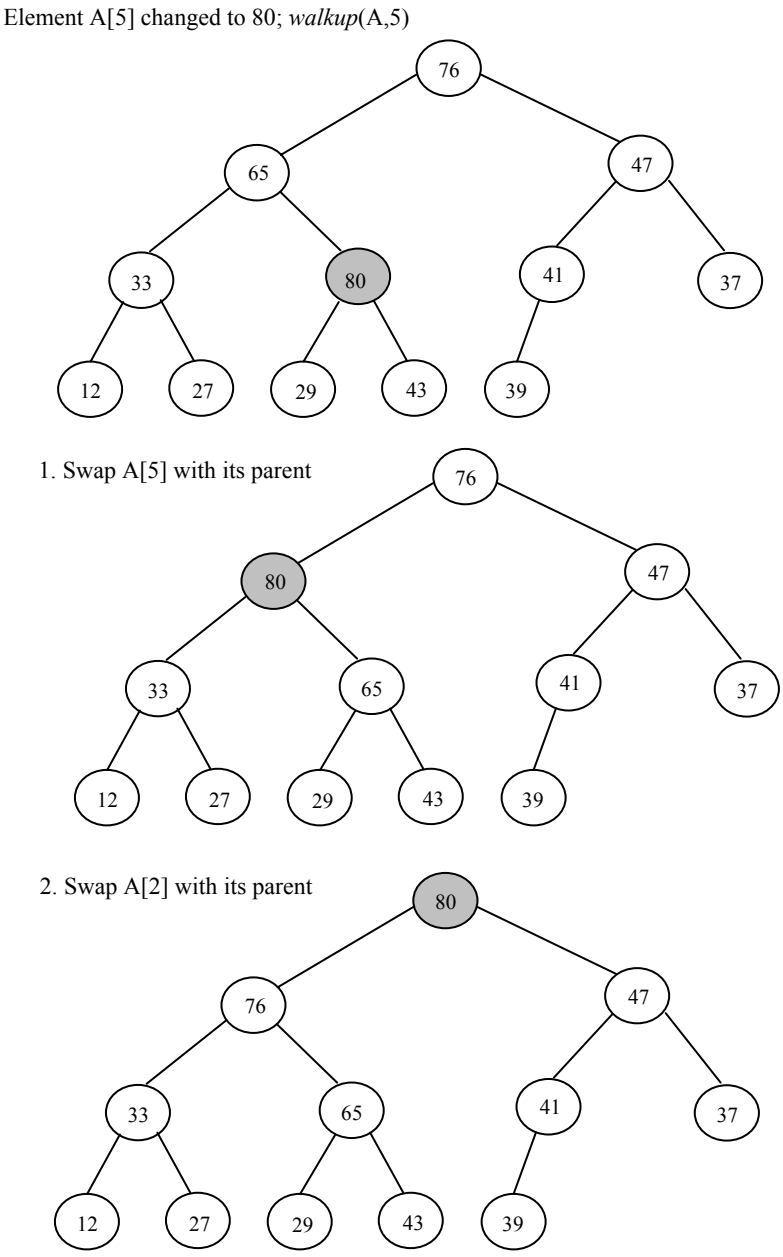
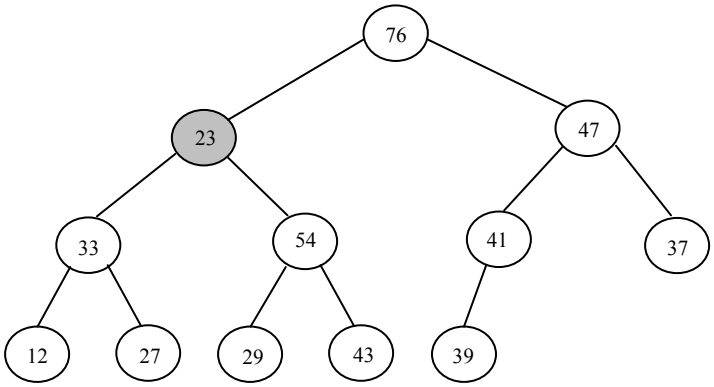
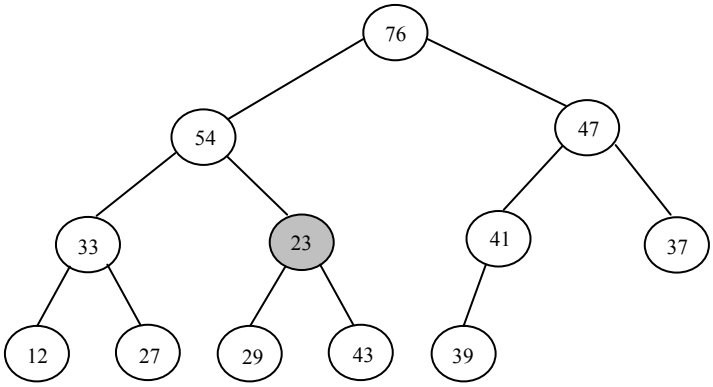


Figure 2.9 Walkup operation on a max-heap.

Element A[2] changed to 23; *walkdown*(A,2)



1. Swap A[2] with maximum of A[4], A[5]



2. Swap A[5] with maximum of A[10], A[11]

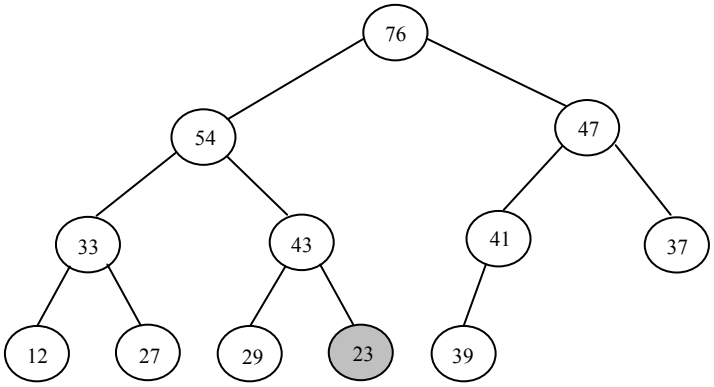


Figure 2.10 Walkdown operation on a max-heap.

```
void Walkdown(int[] A, int i, int n)
{ // walkdown the value at location i in the heap A[1..n]
  int item = A[i];
  while (true)
  { int j = 2*i; // j points to left child
    if (j > n)  break; // break if there is no left child
    // check if there is a right child and update j to point to the largest child
    if ( (j < n) && (A[j+1] > A[j]) )  j++;
    if (item >= A[j]) break;
    // need to swap A[i] with A[j] but a half-swap suffices since the value
    // to walkdown (item) remains unchanged throughout the loop
    A[i] = A[j]; // move A[j] up
    i = j; // location of next node to move down
  }
  A[i] = item;
}

void Walkup(int[] A, int i)
{ // walkup the value at location i in the heap A[1..n]
  int item = A[i];
  while (true)
  { int j = i/2; // j points to parent
    if (j < 1)  break; // break if there is no parent
    if (item <= A[j]) break;
    // need to swap A[i] with A[j] but a half-swap suffices since the value
    // to walkup (item) remains unchanged throughout the loop
    A[i] = A[j]; // move A[j] down
    i = j; // location of next node to move up
  }
  A[i] = item;
}
```

Listing 2.2 Walkdown and walkup algorithms for the array-based implementation of a max-heap.

Heap Operation: Insert

To preserve the complete-tree property of the heap, it is logical that the element be inserted at the location following the last existing element. Thus, the size of the heap is extended by one element to accommodate the added element but then to ensure that that the heap-order property is satisfied, we execute walkup from the newly added location.

Heap Operation: DeleteMax

It is obvious that this operation must return the data stored at the root and somehow delete the root. However, instead of physically deleting the root node, we overwrite its data with the last node' data and then delete the last node. This way, the complete-tree property of the heap is maintained. Finally, to ensure that the heap-order property is satisfied, we execute walkdown starting from the root.

A heap is normally used to implement a priority queue, where enqueue and dequeue operations would correspond to heap-Insert and heap-DeleteMax (or DeleteMin). Such implementation is discussed in Section 11.4.4.

Heap Operation: MakeHeap (Bottom-Up Heap Construction)

We show how to construct a heap out of unsorted elements. The basic idea is as follows:

1. Place all the unsorted elements in a complete binary tree. Note: For the array-based implementation of the heap, we only need to ensure that the elements are stored in the array starting from location 1.
2. Go through the nodes of the tree in a backward order, starting from the parent of the last node, and execute walkdown on each of these nodes. As this is done, the following invariant holds: *Each subtree rooted at a node for which walkdown has been executed is a heap*. Thus, when walkdown the root is executed at the end of this algorithm, the whole tree is a heap.

Listing 2.3 shows the algorithm program code for the array representation of a heap. A worked-out example is given in Figure 2.11.

```
Input: An integer array A[1..n]
Output: The array A converted into a heap

void MakeHeap(int[] A,int n)
{  for(int i = n/2; i > 0;i--)
      walkdown(A,i,n);
}
```

Listing 2.3 The MakeHeap algorithm to convert an array into a heap.

Analysis of MakeHeap

Recall that the cost of walkdown on a heap of size n is proportional to the height of the heap tree, which is O($log\ n$). A rough analysis of MakeHeap suggests that it is O($n\ \ log\ n$) since it does $n/2$ iterations, each iteration costing O($log\ n$). However, precise calculation shows that MakeHeap is O(n). To see this, assume that the heap tree is a complete binary tree of height $k-1$ (i.e., $n=2^k-1$), and note that the (worst case) number of iterations to walkdown a particular node depends on the level of the node. In this case, for all the leaf nodes (there are 2^{k-1} leaf nodes), we do nothing. Then for each of the nodes at the level next to the lowest level, we do one iteration, and so on. Thus,

$$\textit{Number of iterations} \le (2^{k-1})*0 + (2^{k-2})*1 + (2^{k-3})*2 + (2^{k-4})*3 + ... + 1*(k-1) = O(n) \qquad 2.1$$

Exercise 2.1 Carry out detailed derivation to establish the correctness of Equation 2.1.

2.2.1 Heapsort

Heapsort is a worst-case O(n *log* n) sorting algorithm proposed by J. Williams [Wil64]. The algorithm represents a direct application of the heap data structure. It is not difficult to see that once a sequence of n elements is stored into a priority queue (or a heap) then executing DeleteMax n times will output the elements in a sorted order. This is the idea of Heapsort, but with some shortcuts if we are to directly manipulate the array representing the heap. The algorithm is given in Listing 2.4. A worked-out example is shown in Figure 2.11 and Figure 2.12.

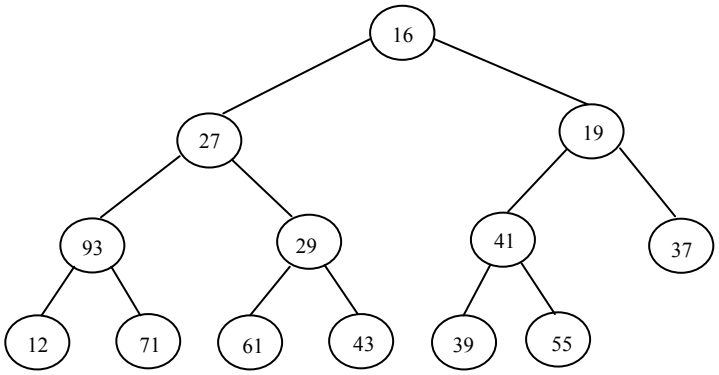
The algorithm consists of two phases. In the first phase, we transform the array into a heap using MakeHeap we saw earlier. The second phase essentially executes $n-1$ DeleteMax operations where every deleted element is moved toward the end of the array (and considered outside of the heap). For example, in the first iteration of Phase 2 (i.e., $i=n$), we are, in effect, deleting the root by swapping the root with A[n], and then executing walkdown from root with A[n] considered outside the heap — note that the last parameter (which tells the size of the heap) in the call to walkdown is $n-1$ for $i=n$.

At the end of the first iteration of Phase 2, the largest element will end up at location n (its proper sorted location), the second-largest element will end up at the root, and the heap size becomes $n-1$. The iteration after that will bring the second-largest element to location $n-1$, and so on.

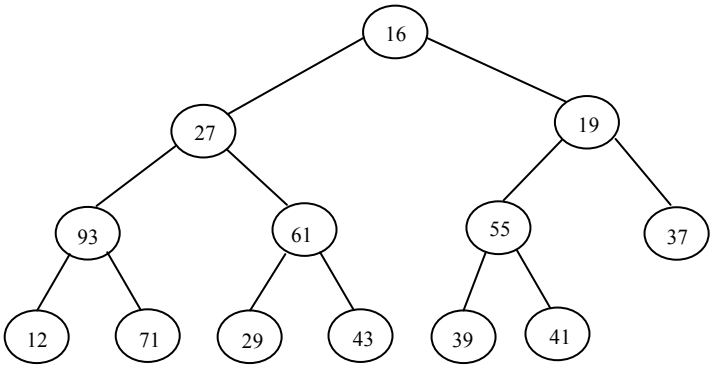
```
Input: An integer array A[1..n]
Output: The array A sorted in nondecreasing order

void Heapsort(int[] A,int n)
{  int i,x;
// Phase 1: MakeHeap: repeatedly call walkdown starting at the parent of last node
   for(i = n/2; i > 0;i--)
       walkdown(A,i,n);
// Phase 2: repeatedly swap the root with the last node and walkdown the root
   for(i = n; i > 1;i--) // suffices to do n-1 iterations
   {  // swap A[1] (root) with A[i];
      x = A[1]; A[1] = A[i]; A[i] = x;
      // walkdown the root
      walkdown(A,1,i-1);
   }
}
```

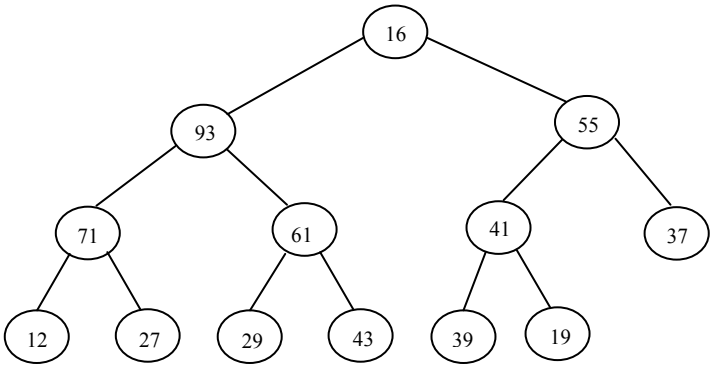
Listing 2.4 Heapsort algorithm.



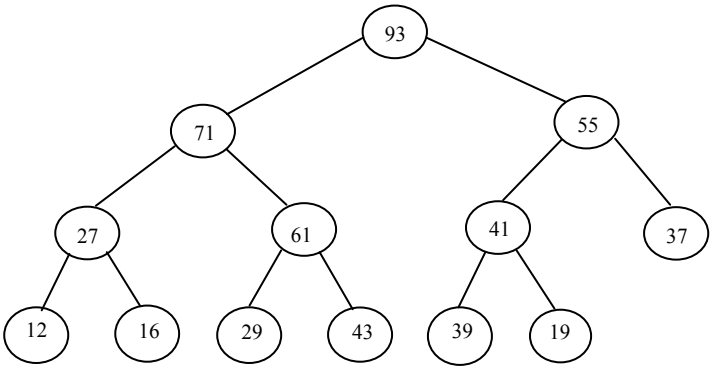
Initial tree corresponding to A[1..13] = [16, 27, 19, 93, 29, 41, 37, 12, 71, 61, 43, 39, 55].



The tree after executing *walkdown*(A,6,13), *walkdown*(A,5,13), *walkdown*(A,4,13).



The tree after executing *walkdown*(A,3,13), *walkdown*(A,2,13).



The tree after executing *walkdown*(A,1,13).

Figure 2.11 MakeHeap: Phase 1 of Heapsort.

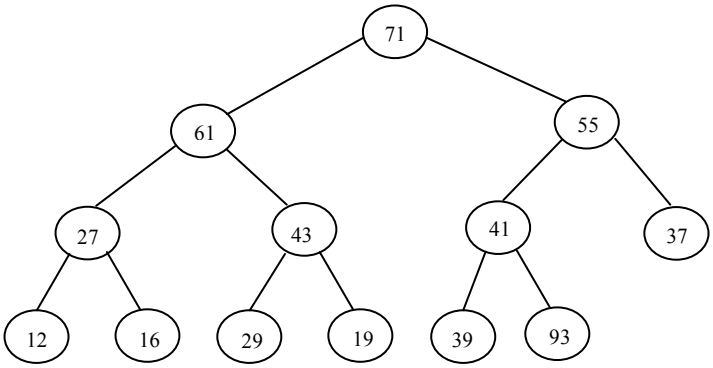


Figure 2.12 The heap tree after executing the first iteration of Phase 2 of Heapsort: the largest element is at its sorted location and the second-largest element is at the root.

Analysis of Heapsort

Recall that the time complexity of a walkdown operation on a heap of size n is proportional to the height of the heap tree, which is O(*log n*). We have already shown that Phase 1 of Heapsort is O(n). Phase 2 executes $n-1$ iterations; each iteration is at most O(*log n*) time. Thus, Phase 2 is O(*n log n*). The algorithm, as a whole, consists of two code blocks A (Phase 1) and B (Phase 2) executed in sequence, which gives the order that is the maximum of {*Order*(A), *Order*(B)}. Hence, we conclude that the worst-case running time of Heapsort is O(*n log n*).

Example 2.4 Show that Heapsort on n integer elements is Θ(*n log n*).

Solution: We need to show that Heapsort is both O(*n log n*) and Ω(*n log n*), to conclude that it is Θ(*n log n*). The preceding analysis has shown that Heapsort is O(*n log n*). Next, we show that Heapsort is Ω(*n log n*). Consider the first $n/2$ iterations of Phase 2. In any of these iterations, we are walking down the root in a heap of size at least $n/2$, for a time of *log* $n/2$. Thus, the time of Phase2 is at least $n/2$ *log* ($n/2$). This latter expression is Ω(*n log n*).

Exercise 2.2 Given the array $A[1..9] = [2, 5, 9, 8, 10, 13, 12, 22, 50]$. Is it a min-heap? Justify your answer.

Exercise 2.3 Another algorithm to construct a heap out of an n-element array $A[1..n]$ is to start with a single element heap containing $A[1]$ then extend it one element at a time by inserting, in order, the elements: $A[2]$, $A[3]$, …, $A[n]$. In other words, the algorithm is as follows:

```
for i = 2 to n
    Walkup(A,i);
```

Show that this algorithm is Ω(*n log n*). Hint: What if "walkup $A[i]$ for $i = n/2$ to n" results in $A[i]$ ending as the root?

Exercise 2.4 For the example shown in Figure 2.12, draw the heap tree after executing the second iteration of Phase 2 of Heapsort.

2.2.2 Application of Queues: Discrete Event Simulation

Queues and priority queues serve as necessary data structures for many efficient algorithms. Several of these are found in graph theory, and they will be discussed later. For now, we will give some simple examples of queue usage. Virtually every real-life waiting line is a queue. For instance, waiting lines at ticket counters are queues, because service is first-come first-served. Similar examples of queues include waiting lines at a bank or a bus stop and a group of cars at a toll booth. In an operating systems (OS), queues are used for scheduling and systems management such as process queues and print queues. When jobs are submitted to a printer, they are arranged in order of arrival. Thus, essentially, jobs sent to a printer are placed in a queue. Occasionally, strict adherence to the queue-definition is not observed; for example, a print job that is already in the queue can be canceled which amounts to a deletion from the middle of the queue.

Generally, it would be useful to know how long users expect to wait in a queue, how long the queue gets, and other similar questions. A specialized area of modern mathematics, called *Queuing theory*, has sprung to address such questions. The answer depends on how frequently users arrive to the queue and how long it takes to service a user. Both of these parameters can be specified by probability distribution functions. In simple cases, such as the single operator (single server) case, an answer can be computed analytically, where the solution is given as a mathematical formula. However, the problem becomes more complex if we allow for multiple operators. In such a situation, we can resort to *simulation*. Simulation uses the computer to emulate the dynamic (time-based) behavior of a real-world system. In our case, the simulation would emulate the real-life queue and can show us how the waiting time would decrease as the number of operators increase and can help us decide on the minimum numbers of operators that lead to acceptable waiting times.

Let us consider a system, such as a bank (or a barber shop), where customers arrive and wait in a line until one of k tellers (or barbers) is available. Customer arrival and customer service time, each, is governed by a probability distribution function. We would like to run a simulation experiment to compute statistics such as the average waiting time for a customer or the average number of customers waiting in line.

The simulation will consist of processing two relevant events: (1) a customer arriving and (2) a customer departing after being serviced. We can use the arrival and service-time probability functions to generate an input stream consisting of ordered pairs of arrival time and service time for each customer, sorted by arrival time (alternatively, the service time for a particular customer can be decided at the time the customer begins service). The times need not be expressed as exact time of the day; rather, they are expressed in *ticks* (which could correspond to seconds, minutes or any time unit appropriate for the experiment) starting form time zero (the time at the start of a simulation run).

Time-Driven Versus Event-Driven Simulation

In a *time-driven simulation*, we maintain a variable that holds the current time, and we increment it by some fixed time-amount at every step of the simulation. The simulation starts with a simulation clock set to zero ticks. We then advance the clock one tick at a time, checking to see if there is an event. If there is, we process the event(s) and compile statistics. When there are no customers left in the input stream and all the tellers are free, then the simulation is over.

The problem with this simulation strategy is that its running time does not depend on the number of customers or events (there are two events per customer), but instead depends on the total number of ticks. To see why this is important, suppose we change the clock units to milliticks (and multiplied all the times in the input by 1000 because they are now in milliticks). The result would be a simulation that takes 1000 times longer!

An alternative approach to time-driven simulation is *event-driven simulation*. The idea is to advance the clock to the time of the *next event* at each stage. This is conceptually easy to do. At any point, the next event that can

occur is either (a) the next customer arrives or (b) one of the customers at a teller departs. Because all the times when the events will happen are available, we just need to find the event that happens nearest in the future, and process that event. We can use a priority queue to store these events, and we use an ordinary queue for the customers waiting in line. If the event is an arrival, we enqueue the customer to the waiting-line queue. If the event is a departure, processing includes updating various statistics and checking the waiting-line queue to move the next customer into service, and inserting its departure-event into the event queue. The simulation ends when the event queue becomes empty. If there are n customers (and thus $2n$ events) and k tellers, then the running time of the simulation would be $O(n \log n)$ because computing and processing each event takes $O(\log n)$.

Sampling with the Inverse Method

The inverse method makes use of a random number generator that generates uniformly random real numbers in [0,1], to produce random samples from probability distributions with a closed-form cumulative density function (CDF), such as the exponential distribution. (For further information on this method and other alternative methods, the reader is referred to *Generating Samples from Probability Distributions* in [Lar81] or *Random Sampling Mechanism* in [Elh06].)

The method consists of two steps: generating a random number R and computing the random variable x from the CDF that corresponds to R.

Because the CDF values, or $F(x) = \text{Prob}(y \leq x)$ are in [0,1], and the R values are in [0,1], the value of x is computed by determining the inverse of CDF at R or $x = F^{-1}(R)$.

Consider the exponential distribution whose probability density function (PDF) is given as:

$$p(x) = \lambda e^{-\lambda x} \text{ where } x > 0$$

To determine a random sample x from p(x), we first determine the CDF of the exponential distribution by integrating p(x) from 0 to x, which results in:

$$F(x) = 1 - \lambda e^{-\lambda x} \text{ where } x > 0$$

The random sample is then determined as $x=F^{-1}(R)$. This results in the following:

$$x = -\frac{1}{\lambda}\ln(1-R)$$

For example, five customers ($\lambda=5$) enter a grocery store each hour, and for R=0.75, the time period until the next customer arrival is $t = -(1/5)\ln(1-0.75) = 0.227$ hour or 16.62 minutes.

2.3 Union-Find for Disjoint Sets

The Union-Find Problem. Maintain a collection of disjoint subsets out of the set {1, 2, …, n} and implement the following operations efficiently:

1. *Find*(x): Return the set containing element x.
2. *Union*(x,y): Replace the set containing x and the set containing y by their union.

The structure that supports the preceding operations is known as *uninon-find ADT*, and has been found useful for many problems. One way to represent a collection of sets is to store each set in a linked list or a tree. However, a compact data-structure to represent the whole collection — this would not be possible if the collections consisted of sets that are not disjoint — is to utilize a *single* array P (P for *Parent*) that indicates set membership via the parent relation associated with the nodes of the tree. In such representation, we can have the root element act as a set identifier.

Figure 2.13 shows an example where n=12 and some three disjoint sets S_1, S_2 and S_3. Thus, we use a parent array P[1..n] where P[i] is the element (node) that is a parent of element i. Because the root element r has no parent, we set P[r] to zero or any other special value that cannot be a valid element. Besides set membership information, we are interested in the heights of various trees. For this, we use another array, *Rank*[1..n]. However, we only need to specify the height of a tree once; so a proper place to store this information is at the array index corresponding to the root node.

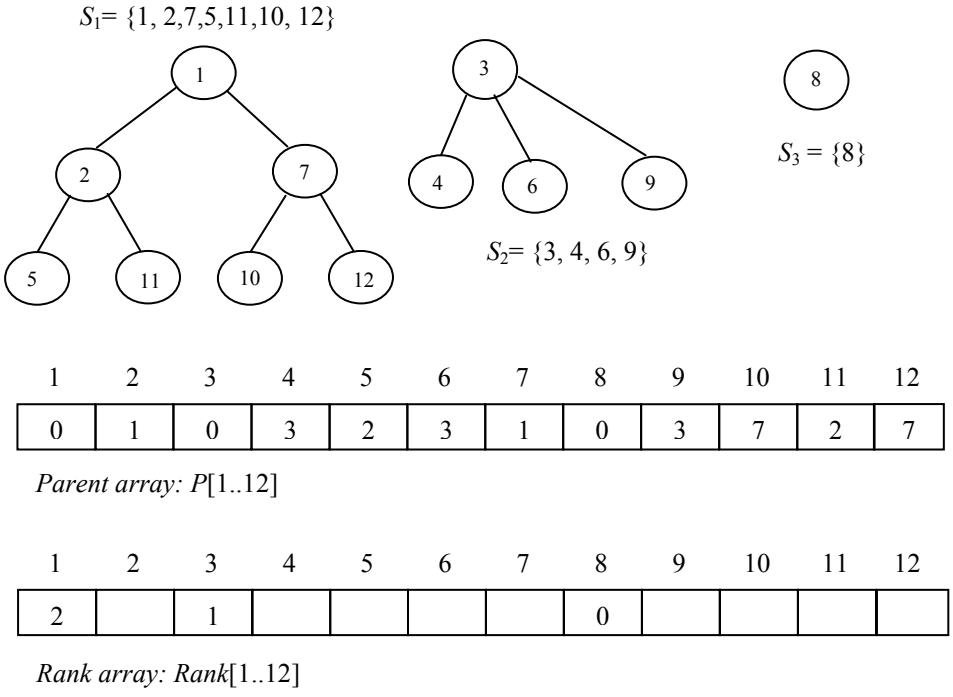


Figure 2.13 *Parent* and *Rank* arrays used for the union-find algorithms.

```
public class UnionFind
{ int[] P; int[] Rank;
  public UnionFind(int n) // SetInitialize
  { P = new int[n + 1]; // Allocate space for Parent (P) array
    Rank = new int[n + 1]; // Allocate space for Rank array
    // Have every element in 1, 2 .. n in a set (a tree) by itself
    for(int i = 1; i <= n; i++)
    { P[i] = 0; Rank[i] = 0; }
  }

  void Union(int x, int y)
  { // replace the set containing x and the set containing y by their union
    int u = Find(x); // Find the set to which x belongs
    int v = Find(y); // Find the set to which y belongs
    if (u==v) return;
    P[v] = u; // Have the tree rooted at v a child of u
  }

  int Find(int x)
  { // return the set (i.e., root element) containing x
    int z = x;
    while (P[z] > 0) { z = P[z]; }
    return z;
  }
}
```

Listing 2.5 Initial implementation for union-find algorithms.

Initial Implementation for Union-Find Operations

Listing 2.5 shows a straightforward implementation for the union and find operations that uses *P* and *Rank* arrays as data structures. We have opted to implement these methods as part of a class, with *P* and *Rank* arrays as private members. The proper initialization for these data structures is performed within the class constructor, which takes an integer *n* as an input parameter and allocates integer arrays *P* and *Rank* to accommodate *n* elements. These arrays are properly initialized to represent the initial state for the union-find problem, where each element (from 1 to *n*) is in a set by itself. The *Union*() method is self-explanatory; its main job is to merge two trees (T_1 rooted at *u* and T_2 rooted at *v*) into a single tree. This can be done by either having T_1 become a child of T_2 (i.e., make *u* a child of *v* by setting *P*[*u*]=*v*) or T_2 become a child of T_1 (i.e., make *v* a child of *u* by setting *P*[*v*]=*u*). The code given in Listing 2.5 opts for the latter option. Finally, the *Find*() method finds (and returns) the root element of tree containing *x* by climbing the tree upward from node *x* until finding the root node (i.e., a node *z* where *P*[*z*]=0).

The preceding implementation suffers from a serious problem that causes the union operation to run in $\Theta(n)$. Also, certain sequences of n union operations may take $\Theta(n^2)$. To see this, consider the sequence of union operations given in Figure 2.14. This sequence leads to a skewed tree (i.e., the tree degenerating into a list). In this case the last union operation *Union*(n,1) issues *Find*(1) and *Find*(n). In this case, *Find*(1) does $n-1$ iterations climbing from element 1 until reaching the root element $n-1$. Similarly, for the operation *Union*($n-1$,1), *Find*(1) executes $n-2$ iterations. Thus, the total number of iterations is given as, $(n-1)+(n-2)+ \ldots +1 = n(n-1)/2 = \Theta(n^2)$.

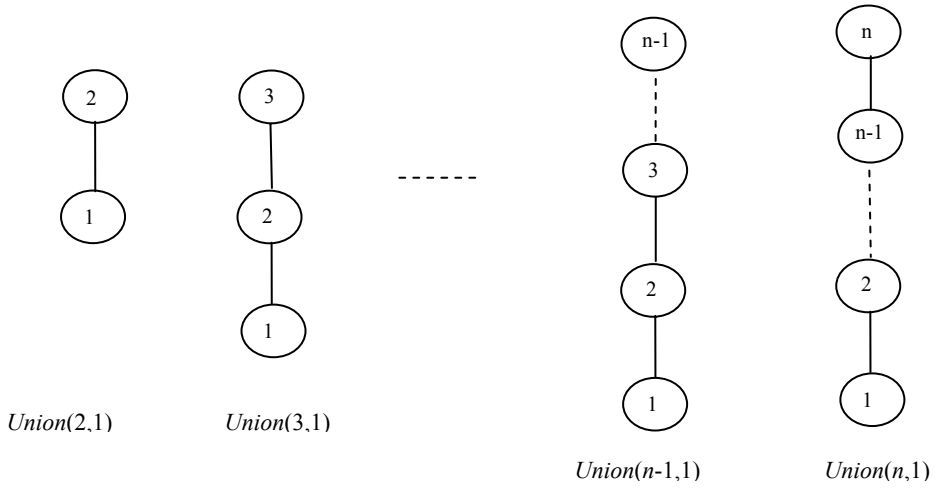


Figure 2.14 Trees resulting from executing a series of union operations: *Union*(2,1), *Union*(3,1), ..., *Union*(n,1).

Union by Rank

The preceding problem can be avoided if the trees constructed by union are always height balanced. In what follows the term *rank* is used as a synonym for *height*. To get height-balanced trees, we modify the union algorithm into a *union-by-rank* where to union two tree T_x and T_y of different ranks, we make the tree having the smaller rank T_x a child of the other tree T_y. This way, the rank of the new tree T_y is unchanged. On the other hand, if T_x and T_y have the same rank r, then (as far as the height of the new tree is concerned) it does not matter whether we make T_x a child of T_y or T_y a child of T_x, because in either case the resulting tree will be of rank = $1+r$.

Listing 2.7 gives the program code for union-by-rank, while Figure 2.15 gives a worked-out example.

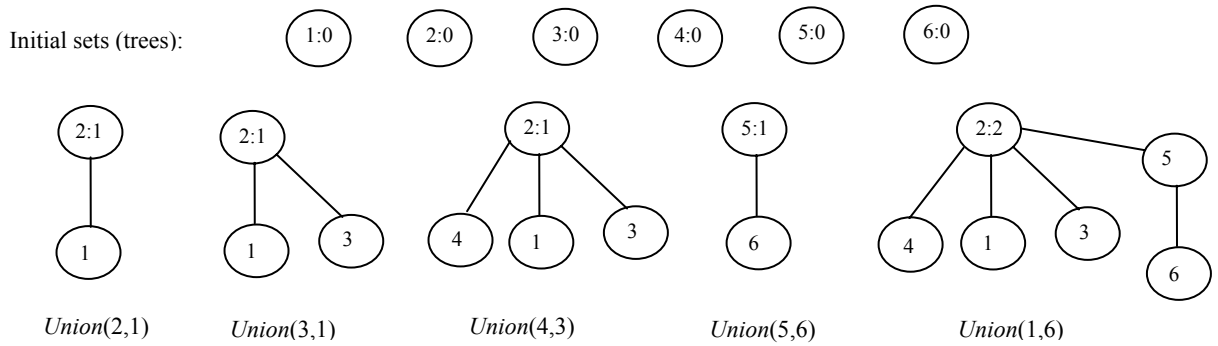


Figure 2.15 Trees resulting from executing a series of union-by-rank operations; a root label is specified as, *Node ID*:*Tree Rank*.

```
void Union(int x, int y)
{  // Replace the set containing x and the set containing y by their union
   int u = Find(x); // Find the set to which x belongs
   int v = Find(y); // Find the set to which y belongs
   if (u==v) return;
   // root of tree of larger rank is the new root (i.e., have v as a child of u)
   if (Rank[u] >= Rank[v]) P[v] = u;
   else P[u] = v;
   // Rank changes by +1 if u and v have same rank; for this case, u is the new root
   if (Rank[u] == Rank[v]) Rank[u]++;
}
```

Listing 2.6 Union (by rank) algorithm.

Lemma 2.2 *Using union-by-rank, the height* (*rank*) *of a k-node tree is at most* $\lfloor \log k \rfloor$.

Proof: (Note that in the lemma we use the *floor* function because the height is an integer).
The proof is by strong induction on k. The base step is $k = 1$. Since a one-node tree has height zero, the statement is true in this case. Next, suppose that $k > 1$, and for all $p < k$, the height of a p-node tree is at most $\lfloor \log p \rfloor$. Let T be a k-node tree. Since $k > 1$, T is the union of two trees: T_1 of height h_1 with $k_1 < k$ nodes and T_2 of height h_2 with $k_2 < k$ nodes. By the induction hypothesis, $h_1 \leq \lfloor \log k_1 \rfloor$ and $h_2 \leq \lfloor \log k_2 \rfloor$. There are two cases to consider: either $h_1 \neq h_2$ or $h_1 = h_2$. If $h_1 \neq h_2$, the height of T is given by:

$$\textit{maximum}\ \{h_1, h_2\} \leq \textit{maximum}\ \{\lfloor \log k_1 \rfloor, \lfloor \log k_2 \rfloor\} \leq \lfloor \log k \rfloor \qquad 2.2$$

On the other hand, if $h_1 = h_2$, the height of $T = 1+h_1 = 1+h_2$ (because the union of two trees with the same height increases the height by 1). For Equation 2.3 given below, we will use $1+h_1$ if $k_1 \leq k_2 \Rightarrow k_1 \leq k/2$; otherwise, we will use $1+h_2$ and in this case, $k_1 > k_2 \Rightarrow k_2 \leq k/2$. Thus, assuming $k_1 \leq k_2$, the height of T is

$$1+h_1 \leq 1+\lfloor \log k_1 \rfloor \leq 1+\lfloor \log k/2 \rfloor = 1+\lfloor \log k - \log 2 \rfloor = 1+\lfloor \log k - 1 \rfloor = \lfloor \log k \rfloor \qquad 2.3$$

This completes the proof.

Find with Path Compression

We can further reduce the height of the trees by doing additional tweaking to the Find() algorithm presented earlier. The idea is simply to have all nodes on the path traversed by find become *direct* children of the root. This is illustrated in Figure 2.16.

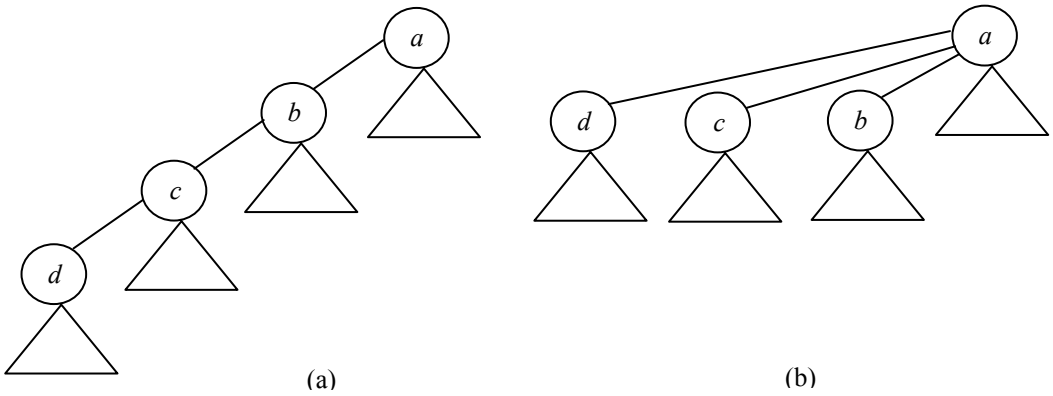


Figure 2.16 *Find* (*with path compression*): *Find*(*d*) transforms the tree (a) into the tree (b) (triangles represent subtrees).

As illustrated by the program code in Listing 2.7, path compression is implemented by having *Find*(*x*) execute two loops, both traversing the path from *x* to the root through parent chain. The purpose of the first loop is to find the root. In the second loop, the parent of every node on the path (from *x* to the root) is reset to the root. Note that the implementation of find-with-path compression alters the structure of the tree but does not care to update the rank information to reflect the tree's height. Figuring out the proper height following path compression requires excessive overhead with little overall pay-off. Figure 2.17 shows an example where both union-by-rank and find-with-path-compression are used.

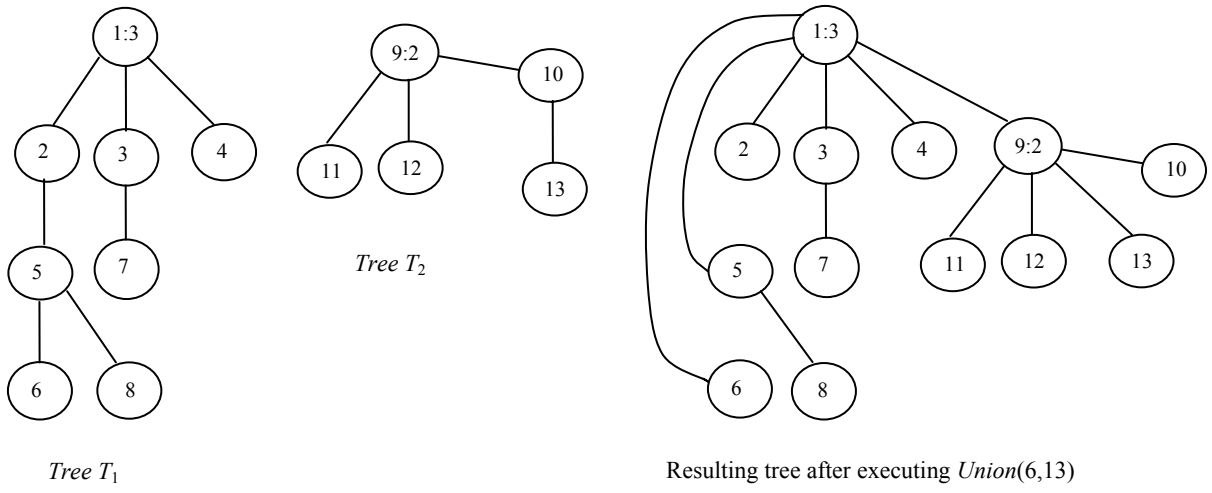


Figure 2.17 Illustration of *union-by-rank* together with *find-with-path-compression*; note that *Union*(6,13) executes calls to *Find*(6) and *Find*(13).

```
int Find(int x)
{  // return the set containing x and compress the path from x to root
   int z = x;
   // Find the root
   while (P[z] > 0) { z = P[z]; }
   int root = z;
   // Start at x again and reset the parent of every node on the path from x to root
   z = x;
   while (P[z] > 0)
   {  int t = P[z];
      P[z] = root; // reset Parent of z to root
      z = t;
   }
   return root;
}
```

Listing 2.7 An algorithm for *Find* (with path compression).

Running-Time Complexity of Union-Find

We have already shown that if union-by-rank is used then the height of the tree is O(*log n*) in the worst case. This implies that the cost (running-time) of any single union or find operation is O(*log n*). Thus, a sequence of *m* union and find operations (arbitrarily interleaved) will cost O(*m log n*). If, in addition to union-by-rank, we use path compression then it can be proven that the cost of *m* union and find operations is O(*m* $log^* n$). The proof is complex and uses *amortized analysis*. Amortized analyasis considers the cost of an algorithm over a sequence of operations, instead of considering the cost of a single operation.

The value of $log^* n$ is the number of times you must iterate the log_2 function on *n* before getting a number less than or equal to 1. (So, $log^* 4=2$, $log^* 16=3$, $log^* 65536=4$, $log^* 2^{65536}=5$). Therefore, for all practical purposes, $log^* n \leq 5$. This implies that the cost for *m* union and find operations is almost O(*m*). Note: Don't conclude from this that the cost of a single union (or find) operation is O(1); rather, it is still O(*log n*).

Exercise 2.5 We can do away with the *Rank* array, if whenever *u* is a root node, we set *P*[*u*] to the negative of the tree rank. Thus, *u* is a root whenever *P*[*u*] ≤ 0. Rewrite *union-by-rank* and *find-with-path-compression* to take this into account.

Exercise 2.6 *Union by Count*. Similar to Lemma 2.2, prove that using *Union by Count*, the height of a *k*-node tree is at most $\lfloor log\ k \rfloor$. In union by count, the union of two trees with different node count, the tree having a smaller count is made a child of the root of the other tree.

2.3.1 Applications of Union-Find ADT

The union-find ADT has been found to be useful in several applications. We discuss some of these next.

Equivalence Classes

The union-find ADT is handy for computing the equivalence classes associated with an *equivalence relation R*.

A *binary relation* on a set S is a set of ordered pairs (a,b) where $a,b \in S$. Thus, a binary relation on a set S is a subset of $S \times S$. A binary relation R on a set S is an *equivalence relation* if it is *reflexive*, *symmetric*, and *transitive*. These properties are defined as follows:

- *Reflexivity*: For any a in S, $(a,a) \in R$.
- *Symmetry*: For any a and b in S, if $(a,b) \in R$, then $(b,a) \in R$.
- *Transitivity*: For any a, b, and c in S, if $(a,b) \in R$ and $(b,c) \in R$, then $(a,c) \in R$.

An equivalence relation R on a set S, partitions (partitioning a set means splitting it into *disjoint nonempty subsets*) S into *equivalence classes* $S_1, S_2, \ldots, S_k$ where each S_i is defined such that if $a \in S_i$ then $S_i = \{ x \mid (a,x) \in R\}$.

Example 2.5 Give the equivalence classes for the following relation, R. R is defined on the set $S = \{1,2, \ldots 10\}$ where $(a, b) \in R$ iff $a \bmod 3 = b \bmod 3$.

Solution: All elements that have the same remainder when divided by 3 are related by R and, therefore, belong to a particular equivalence class. Thus, the equivalence classes correspond to the different remainders of division by 3, namely $\{0,1,2\}$. In this case, the equivalence classes are: $\{3,6,9\}$, $\{1,4,7,10\}$, $\{2,5,8\}$. These sets, respectively, represent the elements where the remainder of division by 3 is 0, 1, or 2.

We can find the equivalence classes of an equivalence relation R on a set S by the following algorithm: Start with an instance of a union-find ADT of size n where $n = |S|$. Then for each pairs (a,b) in R, execute *Union*(a,b). The equivalence classes are then simply the sets of the union-find ADT.

Finding Connected Components in an Undirected Graph

Another example of an equivalence relation is the "connected" relation on the vertices of an undirected graph $G=(V,E)$, where two vertices are connected if there is a path between them. A path from vertex a to vertex b is a sequence of edges (could be empty) from a to b, where the end vertex of one edge is the start vertex of the next edge in the edge sequence. Clearly, such a relation is reflexive (i.e., for any vertex a there is a path from a to a consisting of an empty sequence of edges), symmetric (if there is a path from a to b then the path-in-reverse is a path from b to a), and transitive (if there is a path from a to b and there is a path from b to c then there is a path from a to c obtained by traversing the a-b path then the b-c path). Therefore, the connected relation on an undirected graph partitions the vertices of the graph into equivalences classes (in graph terminology these are the *connected components*).

The connected components of a given undirected graph $G=(V,E)$ is simply a partition of the vertices of the graph into a collection of disjoint sets. These can be easily seen to correspond to the sets of the union-find ADT obtained from the following algorithm: Start with an instance of a union-find ADT of size n where $n = |V|$. Then for each edge (a,b), execute *Union(a,b)*. The running time of this algorithm is $O(|E| \log |V|)$. The problem can also be solved in $O(|E|+|V|)$ by *depth-first search* (Section 7.1)

Finding a Cycle in an Undirected Graph

Consider the following variation of the preceding algorithm that finds the connected components. If, upon encountering the edge (*a*,*b*) for the first time, we see that *Find*(*a*) = *Find*(*b*), then this means that that there is already a path from *a* to *b* and then the edge (*a*,*b*) completes the cycle. The algorithm is given in Listing 2.8. This algorithm returns a *Yes*/*No* answer regarding the presence of a cycle in the input graph. However, for a *Yes*-answer, it would be appropriate to return the vertex sequence representing the cycle. The cycle is the path from *a* to *b* followed by the edge (*b*,*a*) — or the path from *b* to *a* followed by the edge (*a*,*b*) — but how do we find the path from *a* to *b*? Here is a good idea. The graph *G′* consisting of the edges seen so far is a *forest* (i.e., a set of trees). If this tree is maintained using its own parent-array structure (separate from that used by the union-find ADT), the path from *a* to *b* can be recovered by starting at *b* and climbing up via a series of parent links until reaching *a* which produces the path from *b* to *a*.

```
Input: An undirected graph G; vertices are labeled 1,2 …, n.
Output: True if G contains a cycle; False, otherwise.

bool FindCycle(Graph G)
{  n = number of vertices in G;
   UnionFind uf = new UnionFind(n);

   for each edge (a,b) in G
   { if uf.Find(a) = uf.Find(b) return true;
     uf.Union(a,b);
   }
   return false;
}
```

Listing 2.8 An algorithm for finding a cycle in an undirected graph.

Exercise 2.7 Using the idea suggested above, rewrite the *FindCycle*() method given in Listing 2.8 into a complete algorithm that returns (as a string) a vertex sequence representing the cycle found. Analyze the complexity of the resulting algorithm.