6 Non-Binary Trees

Many organizations are hierarchical in nature, such as the military and most businesses. Consider a company with a president and some number of vice presidents who report to the president. Each vice president has some number of direct subordinates, and so on. If we wanted to model this company with a data structure, it would be natural to think of the president in the root node of a tree, the vice presidents at level 1, and their subordinates at lower levels in the tree as we go down the organizational hierarchy.

Because the number of vice presidents is likely to be more than two, this company's organization cannot easily be represented by a binary tree. We need instead to use a tree whose nodes have an arbitrary number of children. Unfortunately, when we permit trees to have nodes with an arbitrary number of children, they become much harder to implement than binary trees. We consider such trees in this chapter. To distinguish them from binary trees, we use the term **general tree**.

Section 6.1 presents general tree terminology. Section 6.2 presents a simple representation for solving the important problem of processing equivalence classes. Several pointer-based implementations for general trees are covered in Section 6.3. Aside from general trees and binary trees, there are also uses for trees whose internal nodes have a fixed number K of children where K is something other than two. Such trees are known as K-ary trees. Section 6.4 generalizes the properties of binary trees to K-ary trees. Sequential representations, useful for applications such as storing trees on disk, are covered in Section 6.5.

6.1 General Tree Definitions and Terminology

A **tree T** is a finite set of one or more nodes such that there is one designated node *R*, called the root of **T**. If the set $(\mathbf{T} - \{R\})$ is not empty, these nodes are partitioned into $n > 0$ disjoint subsets $\mathbf{T}_0$, $\mathbf{T}_1$, ..., $\mathbf{T}_{n-1}$, each of which is a tree, and whose roots R_1, R_2, ..., R_n, respectively, are children of *R*. The subsets $\mathbf{T}_i$ $(0 \leq i < n)$ are said to be **subtrees** of **T**. These subtrees are ordered in that $\mathbf{T}_i$ is said to come before

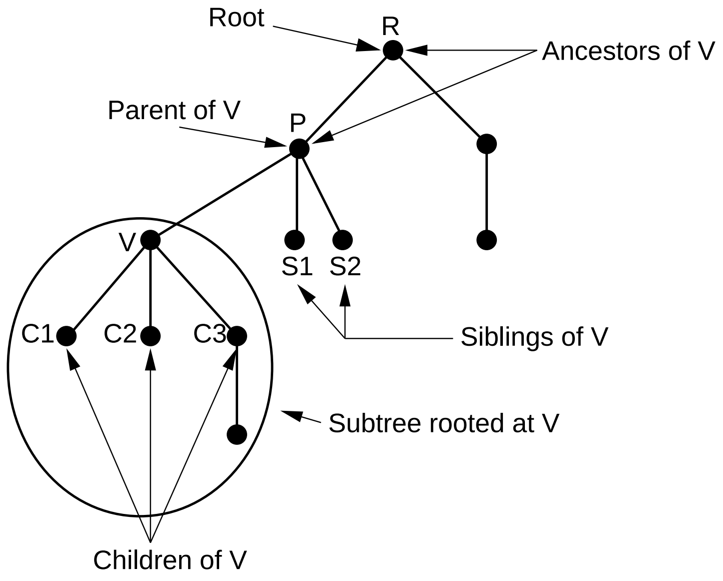


Figure 6.1 Notation for general trees. Node *P* is the parent of nodes *V*, *S1*, and *S2*. Thus, *V*, *S1*, and *S2* are children of *P*. Nodes *R* and *P* are ancestors of *V*. Nodes *V*, *S1*, and *S2* are called **siblings**. The oval surrounds the subtree having *V* as its root.

$\mathbf{T}_j$ if $i < j$. By convention, the subtrees are arranged from left to right with subtree $\mathbf{T}_0$ called the leftmost child of *R*. A node's **out degree** is the number of children for that node. A **forest** is a collection of one or more trees. Figure 6.1 presents further tree notation generalized from the notation for binary trees presented in Chapter 5.

Each node in a tree has precisely one parent, except for the root, which has no parent. From this observation, it immediately follows that a tree with n nodes must have $n - 1$ edges because each node, aside from the root, has one edge connecting that node to its parent.

6.1.1 An ADT for General Tree Nodes

Before discussing general tree implementations, we should first make precise what operations such implementations must support. Any implementation must be able to initialize a tree. Given a tree, we need access to the root of that tree. There must be some way to access the children of a node. In the case of the ADT for binary tree nodes, this was done by providing member functions that give explicit access to the left and right child pointers. Unfortunately, because we do not know in advance how many children a given node will have in the general tree, we cannot give explicit functions to access each child. An alternative must be found that works for an unknown number of children.

```
/** General tree node ADT */
interface GTNode<E> {
  public E value();
  public boolean isLeaf();
  public GTNode<E> parent();
  public GTNode<E> leftmostChild();
  public GTNode<E> rightSibling();
  public void setValue(E value);
  public void setParent(GTNode<E> par);
  public void insertFirst(GTNode<E> n);
  public void insertNext(GTNode<E> n);
  public void removeFirst();
  public void removeNext();
}

/** General tree ADT */
interface GenTree<E> {
  public void clear();      // Clear the tree
  public GTNode<E> root();  // Return the root
  // Make the tree have a new root, give first child and sib
  public void newroot(E value, GTNode<E> first,
                               GTNode<E> sib);
  public void newleftchild(E value); // Add left child
}
```

Figure 6.2 Interfaces for the general tree and general tree node

One choice would be to provide a function that takes as its parameter the index for the desired child. That combined with a function that returns the number of children for a given node would support the ability to access any node or process all children of a node. Unfortunately, this view of access tends to bias the choice for node implementations in favor of an array-based approach, because these functions favor random access to a list of children. In practice, an implementation based on a linked list is often preferred.

An alternative is to provide access to the first (or leftmost) child of a node, and to provide access to the next (or right) sibling of a node. Figure 6.2 shows class declarations for general trees and their nodes. Based on these two access functions, the children of a node can be traversed like a list. Trying to find the next sibling of the rightmost sibling would return `null`.

6.1.2 General Tree Traversals

In Section 5.2, three tree traversals were presented for binary trees: preorder, postorder, and inorder. For general trees, preorder and postorder traversals are defined with meanings similar to their binary tree counterparts. Preorder traversal of a general tree first visits the root of the tree, then performs a preorder traversal of each subtree from left to right. A postorder traversal of a general tree performs a postorder traversal of the root's subtrees from left to right, then visits the root. Inorder

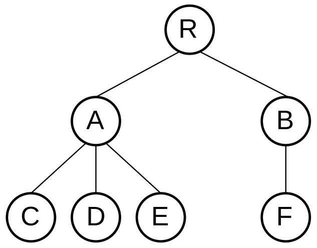


Figure 6.3 An example of a general tree.

traversal does not have a natural definition for the general tree, because there is no particular number of children for an internal node. An arbitrary definition — such as visit the leftmost subtree in inorder, then the root, then visit the remaining subtrees in inorder — can be invented. However, inorder traversals are generally not useful with general trees.

Example 6.1 A preorder traversal of the tree in Figure 6.3 visits the nodes in order $RACDEBF$.

A postorder traversal of this tree visits the nodes in order $CDEAFBR$.

To perform a preorder traversal, it is necessary to visit each of the children for a given node (say *R*) from left to right. This is accomplished by starting at *R*'s leftmost child (call it *T*). From *T*, we can move to *T*'s right sibling, and then to that node's right sibling, and so on.

Using the ADT of Figure 6.2, here is a Java implementation to print the nodes of a general tree in preorder. Note the **`for`** loop at the end, which processes the list of children by beginning with the leftmost child, then repeatedly moving to the next child until calling **`next`** returns **`null`**.

```
/** Preorder traversal for general trees */
static <E> void preorder(GTNode<E> rt) {
  PrintNode(rt);
  if (!rt.isLeaf()) {
    GTNode<E> temp = rt.leftmostChild();
    while (temp != null) {
      preorder(temp);
      temp = temp.rightSibling();
    }
  }
}
```

6.2 The Parent Pointer Implementation

Perhaps the simplest general tree implementation is to store for each node only a pointer to that node's parent. We will call this the **parent pointer** implementation. Clearly this implementation is not general purpose, because it is inadequate for such important operations as finding the leftmost child or the right sibling for a node. Thus, it may seem to be a poor idea to implement a general tree in this way. However, the parent pointer implementation stores precisely the information required to answer the following, useful question: "Given two nodes, are they in the same tree?" To answer the question, we need only follow the series of parent pointers from each node to its respective root. If both nodes reach the same root, then they must be in the same tree. If the roots are different, then the two nodes are not in the same tree. The process of finding the ultimate root for a given node we will call **FIND**.

The parent pointer representation is most often used to maintain a collection of disjoint sets. Two disjoint sets share no members in common (their intersection is empty). A collection of disjoint sets partitions some objects such that every object is in exactly one of the disjoint sets. There are two basic operations that we wish to support:

(1) determine if two objects are in the same set, and

(2) merge two sets together.

Because two merged sets are united, the merging operation is called UNION and the whole process of determining if two objects are in the same set and then merging the sets goes by the name "UNION/FIND."

To implement UNION/FIND, we represent each disjoint set with a separate general tree. Two objects are in the same disjoint set if they are in the same tree. Every node of the tree (except for the root) has precisely one parent. Thus, each node requires the same space to represent it. The collection of objects is typically stored in an array, where each element of the array corresponds to one object, and each element stores the object's value. The objects also correspond to nodes in the various disjoint trees (one tree for each disjoint set), so we also store the parent value with each object in the array. Those nodes that are the roots of their respective trees store an appropriate indicator. Note that this representation means that a single array is being used to implement a collection of trees. This makes it easy to merge trees together with UNION operations.

Figure 6.4 shows the parent pointer implementation for the general tree, called `ParPtrTree`. This class is greatly simplified from the declarations of Figure 6.2 because we need only a subset of the general tree operations. Instead of implementing a separate node class, `ParPtrTree` simply stores an array where each array element corresponds to a node of the tree. Each position i of the array stores the value for node i and the array position for the parent of node i. Class `ParPtrTree`

```
/** General Tree class implementation for UNION/FIND */
class ParPtrTree {
  private Integer [] array;      // Node array

  public ParPtrTree(int size) {
    array = new Integer[size];   // Create node array
    for (int i=0; i<size; i++)
      array[i] = null;
  }

  /** Determine if nodes are in different trees */
  public boolean differ(int a, int b) {
    Integer root1 = FIND(a);     // Find root of node a
    Integer root2 = FIND(b);     // Find root of node b
    return root1 != root2;       // Compare roots
  }

  /** Merge two subtrees */
  public void UNION(int a, int b) {
    Integer root1 = FIND(a);     // Find root of node a
    Integer root2 = FIND(b);     // Find root of node b
    if (root1 != root2) array[root2] = root1; // Merge
  }

/** @return The root of curr's tree */
public Integer FIND(Integer curr) {
  if (array[curr] == null) return curr;  // At root
  while (array[curr] != null) curr = array[curr];
  return curr;
}
```

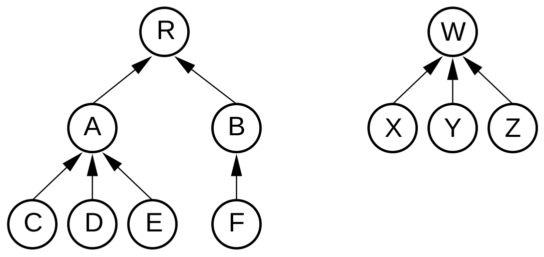
Figure 6.4 General tree implementation using parent pointers for the UNION/FIND algorithm.

is given two new methods, **differ** and **UNION**. Method **differ** checks if two objects are in different sets, and method **UNION** merges two sets together. A private method **FIND** is used to find the ultimate root for an object.

An application using the UNION/FIND operations should store a set of n objects, where each object is assigned a unique index in the range 0 to $n - 1$. The indices refer to the corresponding parent pointers in the array. Class **ParPtrTree** creates and initializes the UNION/FIND array, and methods **differ** and **UNION** take array indices as inputs.

Figure 6.5 illustrates the parent pointer implementation. Note that the nodes can appear in any order within the array, and the array can store up to n separate trees. For example, Figure 6.5 shows two trees stored in the same array. Thus, a single array can store a collection of items distributed among an arbitrary (and changing) number of disjoint subsets.

Consider the problem of assigning the members of a set to disjoint subsets called **equivalence classes**. Recall from Section 2.1 that an equivalence relation is



Parent's Index	/	0	0	1	1	1	2	/	7	7	7
Label	R	A	B	C	D	E	F	W	X	Y	Z
Node Index	0	1	2	3	4	5	6	7	8	9	10

Figure 6.5 The parent pointer array implementation. Each node corresponds to a position in the node array, which stores its value and a pointer to its parent. The parent pointers are represented by the position in the array of the parent. The root of any tree stores **`ROOT`**, represented graphically by a slash in the "Parent's Index" box. This figure shows two trees stored in the same parent pointer array, one rooted at *R*, and the other rooted at *W*.

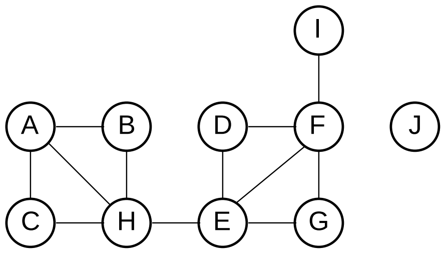


Figure 6.6 A graph with two connected components.

reflexive, symmetric, and transitive. Thus, if objects *A* and *B* are equivalent, and objects *B* and *C* are equivalent, we must be able to recognize that objects *A* and *C* are also equivalent.

There are many practical uses for disjoint sets and representing equivalences. For example, consider Figure 6.6 which shows a graph of ten nodes labeled *A* through *J*. Notice that for nodes *A* through *I*, there is some series of edges that connects any pair of the nodes, but node *J* is disconnected from the rest of the nodes. Such a graph might be used to represent connections such as wires between components on a circuit board, or roads between cities. We can consider two nodes of the graph to be equivalent if there is a path between them. Thus, nodes *A*, *H*, and *E* would be equivalent in Figure 6.6, but *J* is not equivalent to any other. A subset of equivalent (connected) edges in a graph is called a **connected component**. The goal is to quickly classify the objects into disjoint sets that correspond to the connected components. Another application for UNION/FIND occurs in Kruskal's algorithm for computing the minimal cost spanning tree for a graph (Section 11.5.2).

The input to the UNION/FIND algorithm is typically a series of equivalence pairs. In the case of the connected components example, the equivalence pairs would simply be the set of edges in the graph. An equivalence pair might say that object *C* is equivalent to object *A*. If so, *C* and *A* are placed in the same subset. If a later equivalence relates *A* and *B*, then by implication *C* is also equivalent to *B*. Thus, an equivalence pair may cause two subsets to merge, each of which contains several objects.

Equivalence classes can be managed efficiently with the UNION/FIND algorithm. Initially, each object is at the root of its own tree. An equivalence pair is processed by checking to see if both objects of the pair are in the same tree using method **`differ`**. If they are in the same tree, then no change need be made because the objects are already in the same equivalence class. Otherwise, the two equivalence classes should be merged by the **`UNION`** method.

Example 6.2 As an example of solving the equivalence class problem, consider the graph of Figure 6.6. Initially, we assume that each node of the graph is in a distinct equivalence class. This is represented by storing each as the root of its own tree. Figure 6.7(a) shows this initial configuration using the parent pointer array representation. Now, consider what happens when equivalence relationship (*A*, *B*) is processed. The root of the tree containing *A* is *A*, and the root of the tree containing *B* is *B*. To make them equivalent, one of these two roots is set to be the parent of the other. In this case it is irrelevant which points to which, so we arbitrarily select the first in alphabetical order to be the root. This is represented in the parent pointer array by setting the parent field of *B* (the node in array position 1 of the array) to store a pointer to *A*. Equivalence pairs (*C*, *H*), (*G*, *F*), and (*D*, *E*) are processed in similar fashion. When processing the equivalence pair (*I*, *F*), because *I* and *F* are both their own roots, *I* is set to point to *F*. Note that this also makes *G* equivalent to *I*. The result of processing these five equivalences is shown in Figure 6.7(b).

The parent pointer representation places no limit on the number of nodes that can share a parent. To make equivalence processing as efficient as possible, the distance from each node to the root of its respective tree should be as small as possible. Thus, we would like to keep the height of the trees small when merging two equivalence classes together. Ideally, each tree would have all nodes pointing directly to the root. Achieving this goal all the time would require too much ad-

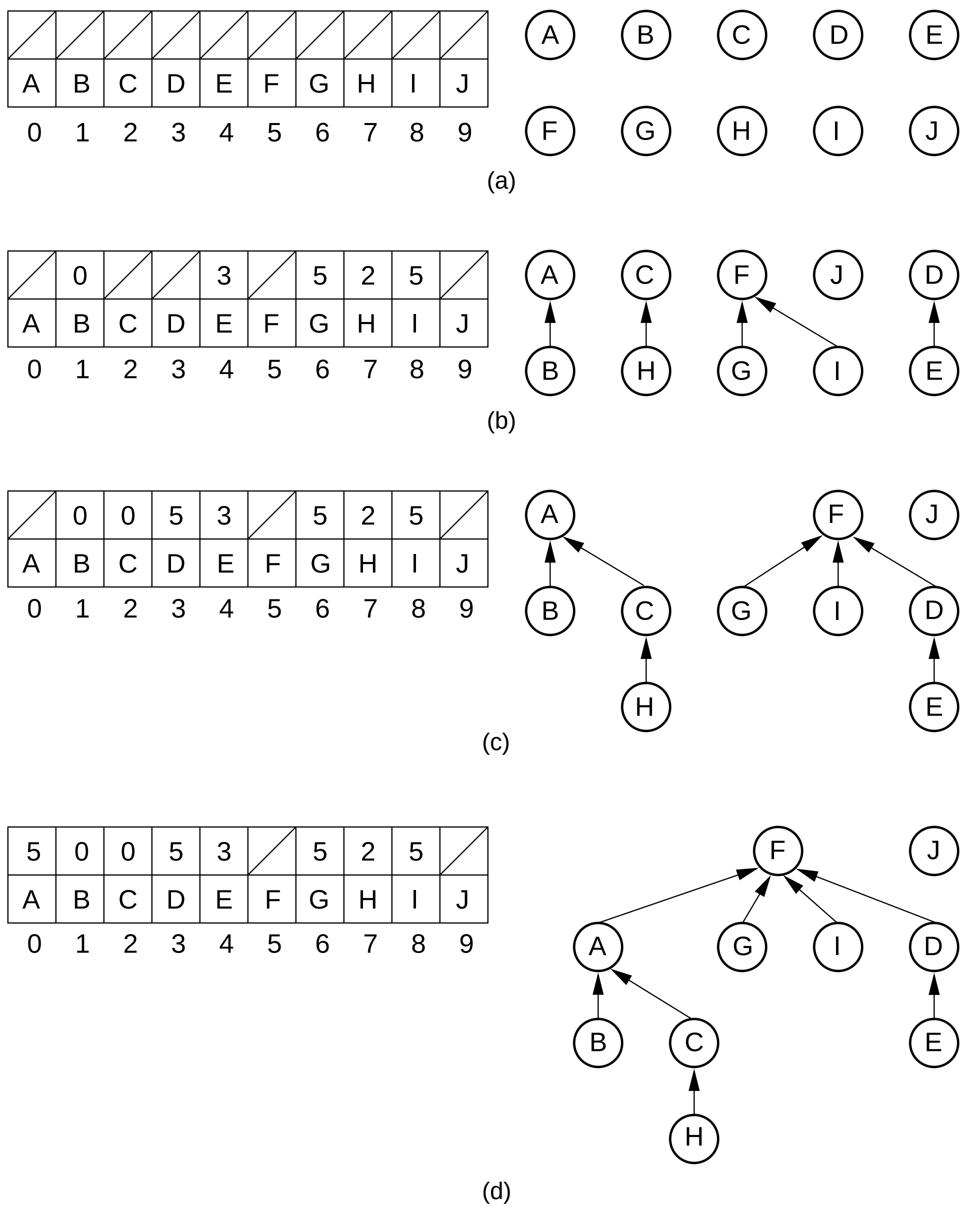


Figure 6.7 An example of equivalence processing. (a) Initial configuration for the ten nodes of the graph in Figure 6.6. The nodes are placed into ten independent equivalence classes. (b) The result of processing five edges: (A, B), (C, H), (G, F), (D, E), and (I, F). (c) The result of processing two more edges: (H, A) and (E, G). (d) The result of processing edge (H, E).

ditional processing to be worth the effort, so we must settle for getting as close as possible.

A low-cost approach to reducing the height is to be smart about how two trees are joined together. One simple technique, called the **weighted union rule**, joins the tree with fewer nodes to the tree with more nodes by making the smaller tree's root point to the root of the bigger tree. This will limit the total depth of the tree to $O(\log n)$, because the depth of nodes only in the smaller tree will now increase by one, and the depth of the deepest node in the combined tree can only be at most one deeper than the deepest node before the trees were combined. The total number of nodes in the combined tree is therefore at least twice the number in the smaller subtree. Thus, the depth of any node can be increased at most $\log n$ times when n equivalences are processed.

Example 6.3 When processing equivalence pair (*I*, *F*) in Figure 6.7(b), *F* is the root of a tree with two nodes while *I* is the root of a tree with only one node. Thus, *I* is set to point to *F* rather than the other way around. Figure 6.7(c) shows the result of processing two more equivalence pairs: (*H*, *A*) and (*E*, *G*). For the first pair, the root for *H* is *C* while the root for *A* is itself. Both trees contain two nodes, so it is an arbitrary decision as to which node is set to be the root for the combined tree. In the case of equivalence pair (*E*, *G*), the root of *E* is *D* while the root of *G* is *F*. Because *F* is the root of the larger tree, node *D* is set to point to *F*.

Not all equivalences will combine two trees. If equivalence (*F*, *G*) is processed when the representation is in the state shown in Figure 6.7(c), no change will be made because *F* is already the root for *G*.

The weighted union rule helps to minimize the depth of the tree, but we can do better than this. **Path compression** is a method that tends to create extremely shallow trees. Path compression takes place while finding the root for a given node *X*. Call this root *R*. Path compression resets the parent of every node on the path from *X* to *R* to point directly to *R*. This can be implemented by first finding *R*. A second pass is then made along the path from *X* to *R*, assigning the parent field of each node encountered to *R*. Alternatively, a recursive algorithm can be implemented as follows. This version of `FIND` not only returns the root of the current node, but also makes all ancestors of the current node point to the root.

```
public Integer FIND(Integer curr) {
  if (array[curr] == null) return curr; // At root
  array[curr] = FIND(array[curr]);
  return array[curr];
}
```

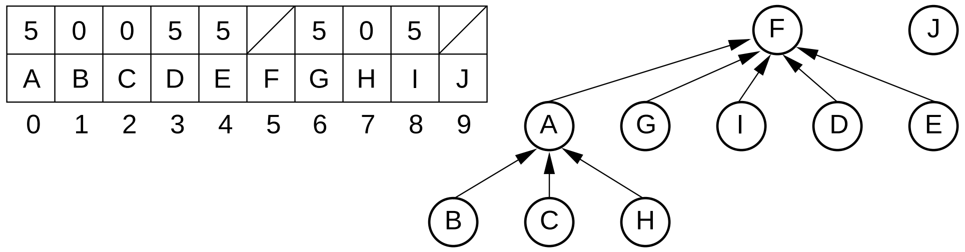


Figure 6.8 An example of path compression, showing the result of processing equivalence pair (*H*, *E*) on the representation of Figure 6.7(c).

Example 6.4 Figure 6.7(d) shows the result of processing equivalence pair (*H*, *E*) on the the representation shown in Figure 6.7(c) using the standard weighted union rule without path compression. Figure 6.8 illustrates the path compression process for the same equivalence pair. After locating the root for node *H*, we can perform path compression to make *H* point directly to root object *A*. Likewise, *E* is set to point directly to its root, *F*. Finally, object *A* is set to point to root object *F*.

Note that path compression takes place during the FIND operation, *not* during the UNION operation. In Figure 6.8, this means that nodes *B*, *C*, and *H* have node *A* remain as their parent, rather than changing their parent to be *F*. While we might prefer to have these nodes point to *F*, to accomplish this would require that additional information from the FIND operation be passed back to the UNION operation. This would not be practical.

Path compression keeps the cost of each FIND operation very close to constant. To be more precise about what is meant by "very close to constant," the cost of path compression for n FIND operations on n nodes (when combined with the weighted union rule for joining sets) is approximately[1] $\Theta(n \log^* n)$. The notation "$\log^* n$" means the number of times that the log of n must be taken before $n \leq 1$. For example, $\log^* 65536$ is 4 because $\log 65536 = 16$, $\log 16 = 4$, $\log 4 = 2$, and finally $\log 2 = 1$. Thus, $\log^* n$ grows *very* slowly, so the cost for a series of n FIND operations is very close to n.

Note that this does not mean that the tree resulting from processing n equivalence pairs necessarily has depth $\Theta(\log^* n)$. One can devise a series of equivalence operations that yields $\Theta(\log n)$ depth for the resulting tree. However, many of the equivalences in such a series will look only at the roots of the trees being merged, requiring little processing time. The *total* amount of processing time required for n operations will be $\Theta(n \log^* n)$, yielding nearly constant time for each equiva-

[1]To be more precise, this cost has been found to grow in time proportional to the inverse of Ackermann's function. See Section 6.6.

lence operation. This is an example of amortized analysis, discussed further in Section 14.3.

6.3 General Tree Implementations

We now tackle the problem of devising an implementation for general trees that allows efficient processing for all member functions of the ADTs shown in Figure 6.2. This section presents several approaches to implementing general trees. Each implementation yields advantages and disadvantages in the amount of space required to store a node and the relative ease with which key operations can be performed. General tree implementations should place no restriction on how many children a node may have. In some applications, once a node is created the number of children never changes. In such cases, a fixed amount of space can be allocated for the node when it is created, based on the number of children for the node. Matters become more complicated if children can be added to or deleted from a node, requiring that the node's space allocation be adjusted accordingly.

6.3.1 List of Children

Our first attempt to create a general tree implementation is called the "list of children" implementation for general trees. It simply stores with each internal node a linked list of its children. This is illustrated by Figure 6.9.

The "list of children" implementation stores the tree nodes in an array. Each node contains a value, a pointer (or index) to its parent, and a pointer to a linked list of the node's children, stored in order from left to right. Each linked list element contains a pointer to one child. Thus, the leftmost child of a node can be found directly because it is the first element in the linked list. However, to find the right sibling for a node is more difficult. Consider the case of a node M and its parent P. To find M's right sibling, we must move down the child list of P until the linked list element storing the pointer to M has been found. Going one step further takes us to the linked list element that stores a pointer to M's right sibling. Thus, in the worst case, to find M's right sibling requires that all children of M's parent be searched.

Combining trees using this representation is difficult if each tree is stored in a separate node array. If the nodes of both trees are stored in a single node array, then adding tree $\mathbf{T}$ as a subtree of node R is done by simply adding the root of $\mathbf{T}$ to R's list of children.

6.3.2 The Left-Child/Right-Sibling Implementation

With the "list of children" implementation, it is difficult to access a node's right sibling. Figure 6.10 presents an improvement. Here, each node stores its value and pointers to its parent, leftmost child, and right sibling. Thus, each of the basic

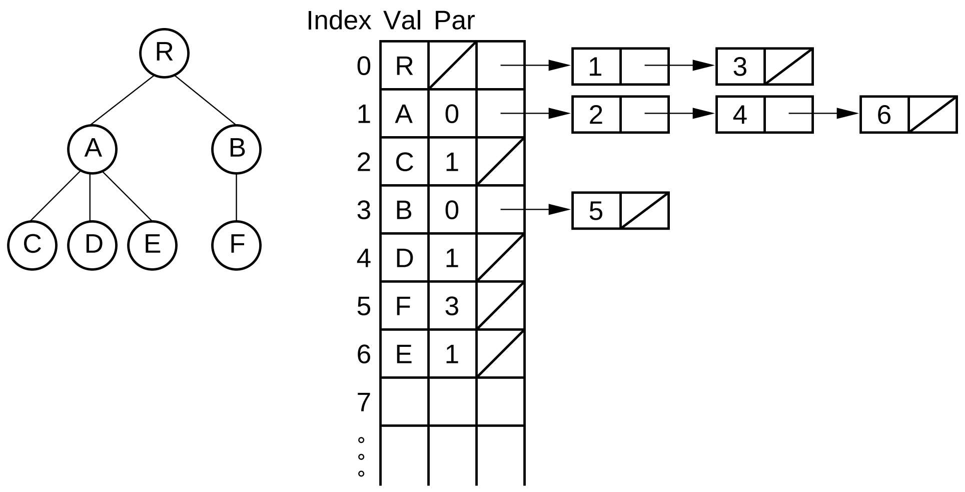


Figure 6.9 The "list of children" implementation for general trees. The column of numbers to the left of the node array labels the array indices. The column labeled "Val" stores node values. The column labeled "Par" stores indices (or pointers) to the parents. The last column stores pointers to the linked list of children for each internal node. Each element of the linked list stores a pointer to one of the node's children (shown as the array index of the target node).

ADT operations can be implemented by reading a value directly from the node. If two trees are stored within the same node array, then adding one as the subtree of the other simply requires setting three pointers. Combining trees in this way is illustrated by Figure 6.11. This implementation is more space efficient than the "list of children" implementation, and each node requires a fixed amount of space in the node array.

6.3.3 Dynamic Node Implementations

The two general tree implementations just described use an array to store the collection of nodes. In contrast, our standard implementation for binary trees stores each node as a separate dynamic object containing its value and pointers to its two children. Unfortunately, nodes of a general tree can have any number of children, and this number may change during the life of the node. A general tree node implementation must support these properties. One solution is simply to limit the number of children permitted for any node and allocate pointers for exactly that number of children. There are two major objections to this. First, it places an undesirable limit on the number of children, which makes certain trees unrepresentable by this implementation. Second, this might be extremely wasteful of space because most nodes will have far fewer children and thus leave some pointer positions empty.

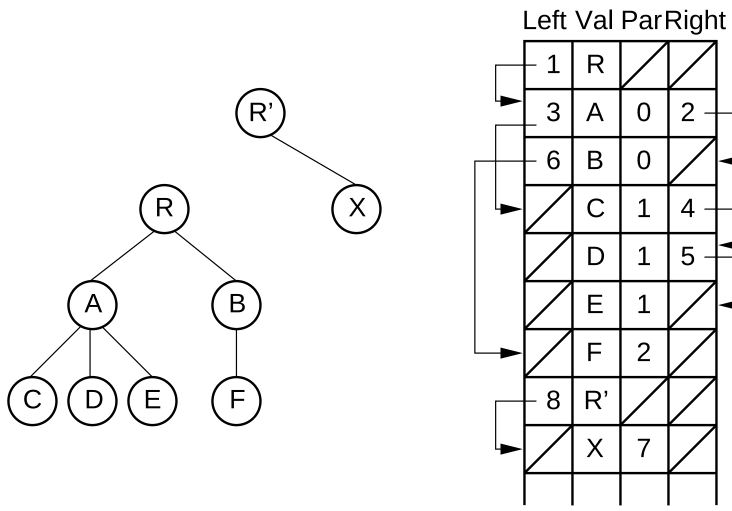


Figure 6.10 The “left-child/right-sibling” implementation.

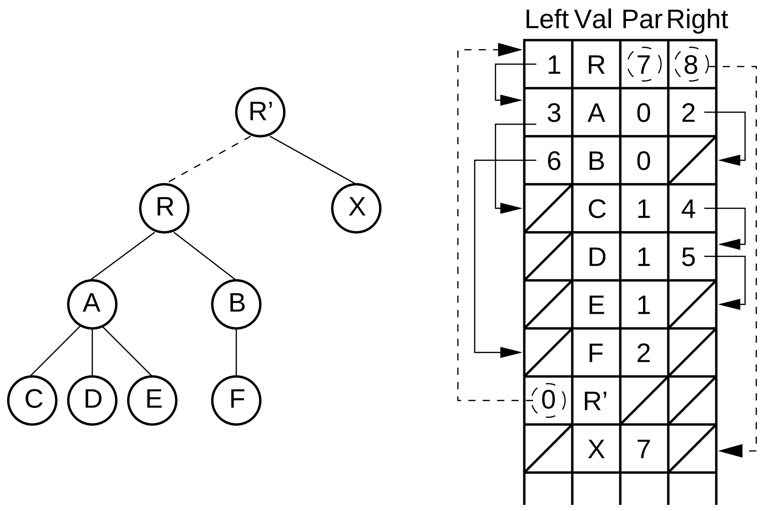


Figure 6.11 Combining two trees that use the “left-child/right-sibling” implementation. The subtree rooted at *R* in Figure 6.10 now becomes the first child of *R*′. Three pointers are adjusted in the node array: The left-child field of *R*′ now points to node *R*, while the right-sibling field for *R* points to node *X*. The parent field of node *R* points to node *R*′.

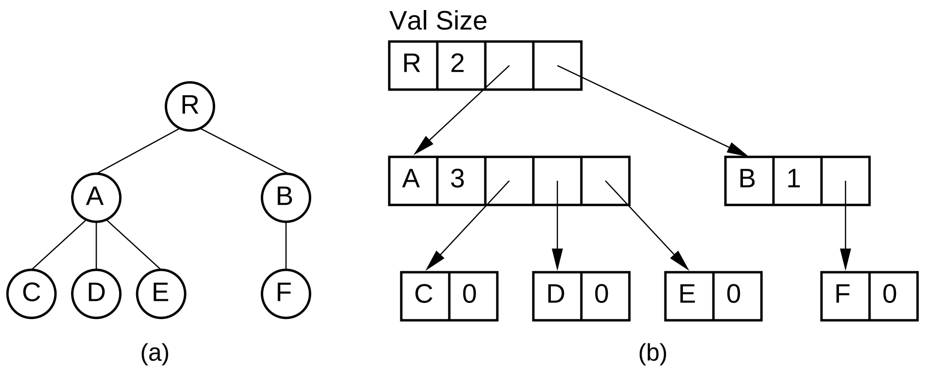


Figure 6.12 A dynamic general tree representation with fixed-size arrays for the child pointers. (a) The general tree. (b) The tree representation. For each node, the first field stores the node value while the second field stores the size of the child pointer array.

The alternative is to allocate variable space for each node. There are two basic approaches. One is to allocate an array of child pointers as part of the node. In essence, each node stores an array-based list of child pointers. Figure 6.12 illustrates the concept. This approach assumes that the number of children is known when the node is created, which is true for some applications but not for others. It also works best if the number of children does not change. If the number of children does change (especially if it increases), then some special recovery mechanism must be provided to support a change in the size of the child pointer array. One possibility is to allocate a new node of the correct size from free store and return the old copy of the node to free store for later reuse. This works especially well in a language with built-in garbage collection such as Java. For example, assume that a node *M* initially has two children, and that space for two child pointers is allocated when *M* is created. If a third child is added to *M*, space for a new node with three child pointers can be allocated, the contents of *M* is copied over to the new space, and the old space is then returned to free store. As an alternative to relying on the system's garbage collector, a memory manager for variable size storage units can be implemented, as described in Section 12.3. Another possibility is to use a collection of free lists, one for each array size, as described in Section 4.1.2. Note in Figure 6.12 that the current number of children for each node is stored explicitly in a `size` field. The child pointers are stored in an array with `size` elements.

Another approach that is more flexible, but which requires more space, is to store a linked list of child pointers with each node as illustrated by Figure 6.13. This implementation is essentially the same as the "list of children" implementation of Section 6.3.1, but with dynamically allocated nodes rather than storing the nodes in an array.

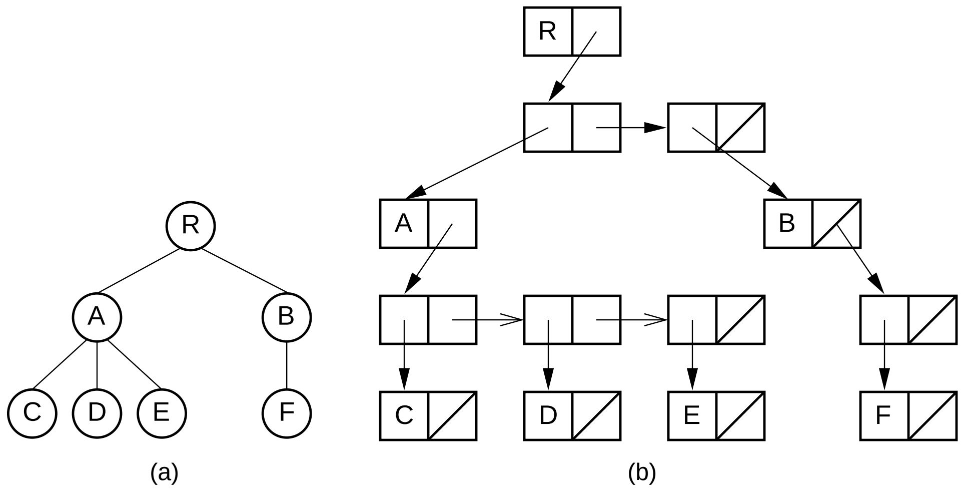


Figure 6.13 A dynamic general tree representation with linked lists of child pointers. (a) The general tree. (b) The tree representation.

6.3.4 Dynamic "Left-Child/Right-Sibling" Implementation

The "left-child/right-sibling" implementation of Section 6.3.2 stores a fixed number of pointers with each node. This can be readily adapted to a dynamic implementation. In essence, we substitute a binary tree for a general tree. Each node of the "left-child/right-sibling" implementation points to two "children" in a new binary tree structure. The left child of this new structure is the node's first child in the general tree. The right child is the node's right sibling. We can easily extend this conversion to a forest of general trees, because the roots of the trees can be considered siblings. Converting from a forest of general trees to a single binary tree is illustrated by Figure 6.14. Here we simply include links from each node to its right sibling and remove links to all children except the leftmost child. Figure 6.15 shows how this might look in an implementation with two pointers at each node. Compared with the implementation illustrated by Figure 6.13 which requires overhead of three pointers/node, the implementation of Figure 6.15 only requires two pointers per node. The representation of Figure 6.15 is likely to be easier to implement, space efficient, and more flexible than the other implementations presented in this section.

6.4 K-ary Trees

K-ary trees are trees whose internal nodes all have exactly K children. Thus, a full binary tree is a 2-ary tree. The PR quadtree discussed in Section 13.3 is an example of a 4-ary tree. Because K-ary tree nodes have a fixed number of children, unlike general trees, they are relatively easy to implement. In general, K-ary trees

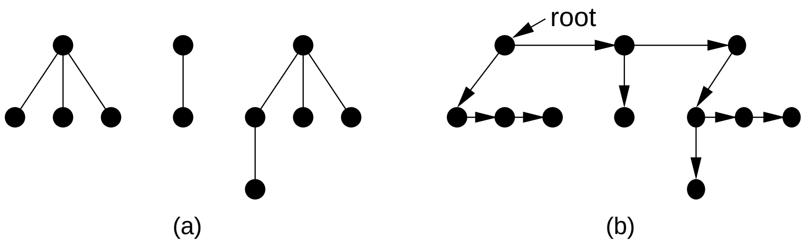


Figure 6.14 Converting from a forest of general trees to a single binary tree. Each node stores pointers to its left child and right sibling. The tree roots are assumed to be siblings for the purpose of converting.

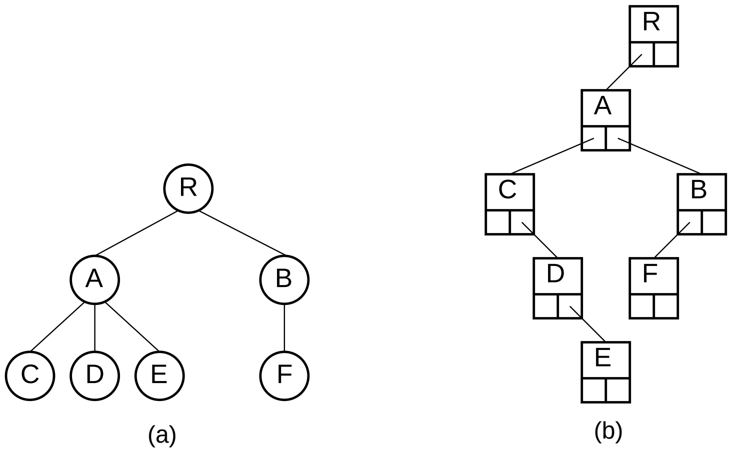


Figure 6.15 A general tree converted to the dynamic "left-child/right-sibling" representation. Compared to the representation of Figure 6.13, this representation requires less space.

bear many similarities to binary trees, and similar implementations can be used for K-ary tree nodes. Note that as K becomes large, the potential number of **`null`** pointers grows, and the difference between the required sizes for internal nodes and leaf nodes increases. Thus, as K becomes larger, the need to choose separate implementations for the internal and leaf nodes becomes more pressing.

Full and **complete** K-ary trees are analogous to full and complete binary trees, respectively. Figure 6.16 shows full and complete K-ary trees for $K = 3$. In practice, most applications of K-ary trees limit them to be either full or complete.

Many of the properties of binary trees extend to K-ary trees. Equivalent theorems to those in Section 5.1.1 regarding the number of NULL pointers in a K-ary tree and the relationship between the number of leaves and the number of internal nodes in a K-ary tree can be derived. We can also store a complete K-ary tree in an array, using simple formulas to compute a node's relations in a manner similar to that used in Section 5.3.3.

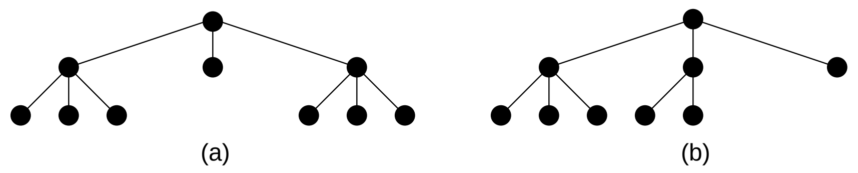


Figure 6.16 Full and complete 3-ary trees. (a) This tree is full (but not complete). (b) This tree is complete (but not full).

6.5 Sequential Tree Implementations

Next we consider a fundamentally different approach to implementing trees. The goal is to store a series of node values with the minimum information needed to reconstruct the tree structure. This approach, known as a **sequential** tree implementation, has the advantage of saving space because no pointers are stored. It has the disadvantage that accessing any node in the tree requires sequentially processing all nodes that appear before it in the node list. In other words, node access must start at the beginning of the node list, processing nodes sequentially in whatever order they are stored until the desired node is reached. Thus, one primary virtue of the other implementations discussed in this section is lost: efficient access (typically $\Theta(\log n)$ time) to arbitrary nodes in the tree. Sequential tree implementations are ideal for archiving trees on disk for later use because they save space, and the tree structure can be reconstructed as needed for later processing.

Sequential tree implementations can be used to **serialize** a tree structure. Serialization is the process of storing an object as a series of bytes, typically so that the data structure can be transmitted between computers. This capability is important when using data structures in a distributed processing environment.

A sequential tree implementation typically stores the node values as they would be enumerated by a preorder traversal, along with sufficient information to describe the tree's shape. If the tree has restricted form, for example if it is a full binary tree, then less information about structure typically needs to be stored. A general tree, because it has the most flexible shape, tends to require the most additional shape information. There are many possible sequential tree implementation schemes. We will begin by describing methods appropriate to binary trees, then generalize to an implementation appropriate to a general tree structure.

Because every node of a binary tree is either a leaf or has two (possibly empty) children, we can take advantage of this fact to implicitly represent the tree's structure. The most straightforward sequential tree implementation lists every node value as it would be enumerated by a preorder traversal. Unfortunately, the node values alone do not provide enough information to recover the shape of the tree. In particular, as we read the series of node values, we do not know when a leaf node has been reached. However, we can treat all non-empty nodes as internal nodes

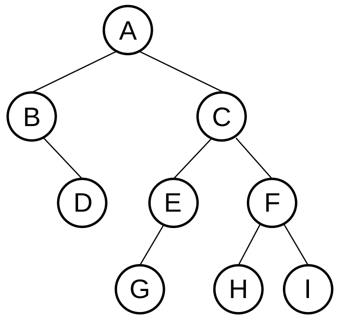


Figure 6.17 Sample binary tree for sequential tree implementation examples.

with two (possibly empty) children. Only **`null`** values will be interpreted as leaf nodes, and these can be listed explicitly. Such an augmented node list provides enough information to recover the tree structure.

Example 6.5 For the binary tree of Figure 6.17, the corresponding sequential representation would be as follows (assuming that '/' stands for **`null`**):

$$AB/D//CEG///FH//I// \qquad (6.1)$$

To reconstruct the tree structure from this node list, we begin by setting node *A* to be the root. *A*'s left child will be node *B*. Node *B*'s left child is a **`null`** pointer, so node *D* must be *B*'s right child. Node *D* has two **`null`** children, so node *C* must be the right child of node *A*.

To illustrate the difficulty involved in using the sequential tree representation for processing, consider searching for the right child of the root node. We must first move sequentially through the node list of the left subtree. Only at this point do we reach the value of the root's right child. Clearly the sequential representation is space efficient, but not time efficient for descending through the tree along some arbitrary path.

Assume that each node value takes a constant amount of space. An example would be if the node value is a positive integer and **`null`** is indicated by the value zero. From the Full Binary Tree Theorem of Section 5.1.1, we know that the size of the node list will be about twice the number of nodes (i.e., the overhead fraction is 1/2). The extra space is required by the **`null`** pointers. We should be able to store the node list more compactly. However, any sequential implementation must recognize when a leaf node has been reached, that is, a leaf node indicates the end of a subtree. One way to do this is to explicitly list with each node whether it is an internal node or a leaf. If a node *X* is an internal node, then we know that its two children (which may be subtrees) immediately follow *X* in the node list. If *X* is a leaf node, then the next node in the list is the right child of some ancestor of *X*, not the right child of *X*. In particular, the next node will be the child of *X*'s most recent ancestor that has not yet seen its right child. However, this assumes that each internal node does in fact have two children, in other words, that the tree is full. Empty children must be indicated in the node list explicitly. Assume that internal nodes are marked with a prime ($'$) and that leaf nodes show no mark. Empty children of internal nodes are indicated by '/', but the (empty) children of leaf nodes are not represented at all. Note that a full binary tree stores no **`null`** values with this implementation, and so requires less overhead.

Example 6.6 We can represent the tree of Figure 6.17 as follows:

$$A'B'/DC'E'G/F'HI \tag{6.2}$$

Note that slashes are needed for the empty children because this is not a full binary tree.

Storing n extra bits can be a considerable savings over storing n **`null`** values. In Example 6.6, each node is shown with a mark if it is internal, or no mark if it is a leaf. This requires that each node value has space to store the mark bit. This might be true if, for example, the node value were stored as a 4-byte integer but the range of the values sored was small enough so that not all bits are used. An example would be if all node values must be positive. Then the high-order (sign) bit of the integer value could be used as the mark bit.

Another approach is to store a separate bit vector to represent the status of each node. In this case, each node of the tree corresponds to one bit in the bit vector. A value of '1' could indicate an internal node, and '0' could indicate a leaf node.

Example 6.7 The bit vector for the tree if Figure 6.17 (including positions for the null children of nodes *B* and *E*) would be

$$11001100100 \tag{6.3}$$

Storing general trees by means of a sequential implementation requires that more explicit structural information be included with the node list. Not only must the general tree implementation indicate whether a node is leaf or internal, it must also indicate how many children the node has. Alternatively, the implementation can indicate when a node's child list has come to an end. The next example dispenses with marks for internal or leaf nodes. Instead it includes a special mark (we will use the ")" symbol) to indicate the end of a child list. All leaf nodes are followed by a ")" symbol because they have no children. A leaf node that is also the last child for its parent would indicate this by two or more successive ")" symbols.

Example 6.8 For the general tree of Figure 6.3, we get the sequential representation

$$RAC)D)E))BF))) \tag{6.4}$$

Note that F is followed by three ")" marks, because it is a leaf, the last node of B's rightmost subtree, and the last node of R's rightmost subtree.

Note that this representation for serializing general trees cannot be used for binary trees. This is because a binary tree is not merely a restricted form of general tree with at most two children. Every binary tree node has a left and a right child, though either or both might be empty. For example, the representation of Example 6.8 cannot let us distinguish whether node D in Figure 6.17 is the left or right child of node B.

6.6 Further Reading

The expression $\log^* n$ cited in Section 6.2 is closely related to the inverse of Ackermann's function. For more information about Ackermann's function and the cost of path compression for UNION/FIND, see Robert E. Tarjan's paper "On the efficiency of a good but not linear set merging algorithm" [Tar75]. The article "Data Structures and Algorithms for Disjoint Set Union Problems" by Galil and Italiano [GI91] covers many aspects of the equivalence class problem.

Foundations of Multidimensional and Metric Data Structures by Hanan Samet [Sam06] treats various implementations of tree structures in detail within the context of K-ary trees. Samet covers sequential implementations as well as the linked and array implementations such as those described in this chapter and Chapter 5. While these books are ostensibly concerned with spatial data structures, many of the concepts treated are relevant to anyone who must implement tree structures.

6.7 Exercises

6.1 Write an algorithm to determine if two general trees are identical. Make the algorithm as efficient as you can. Analyze your algorithm's running time.

6.2 Write an algorithm to determine if two binary trees are identical when the ordering of the subtrees for a node is ignored. For example, if a tree has root node with value R, left child with value A and right child with value B, this would be considered identical to another tree with root node value R, left child value *B*, and right child value *A*. Make the algorithm as efficient as you can. Analyze your algorithm's running time. How much harder would it be to make this algorithm work on a general tree?

6.3 Write a postorder traversal function for general trees, similar to the preorder traversal function named **`preorder`** given in Section 6.1.2.

6.4 Write a function that takes as input a general tree and returns the number of nodes in that tree. Write your function to use the **`GenTree`** and **`GTNode`** ADTs of Figure 6.2.

6.5 Describe how to implement the weighted union rule efficiently. In particular, describe what information must be stored with each node and how this information is updated when two trees are merged. Modify the implementation of Figure 6.4 to support the weighted union rule.

6.6 A potential alternative to the weighted union rule for combining two trees is the height union rule. The height union rule requires that the root of the tree with greater height become the root of the union. Explain why the height union rule can lead to worse average time behavior than the weighted union rule.

6.7 Using the weighted union rule and path compression, show the array for the parent pointer implementation that results from the following series of equivalences on a set of objects indexed by the values 0 through 15. Initially, each element in the set should be in a separate equivalence class. When two trees to be merged are the same size, make the root with greater index value be the child of the root with lesser index value.

(0, 2) (1, 2) (3, 4) (3, 1) (3, 5) (9, 11) (12, 14) (3, 9) (4, 14) (6, 7) (8, 10) (8, 7) (7, 0) (10, 15) (10, 13)

6.8 Using the weighted union rule and path compression, show the array for the parent pointer implementation that results from the following series of equivalences on a set of objects indexed by the values 0 through 15. Initially, each element in the set should be in a separate equivalence class. When two trees to be merged are the same size, make the root with greater index value be the child of the root with lesser index value.

(2, 3) (4, 5) (6, 5) (3, 5) (1, 0) (7, 8) (1, 8) (3, 8) (9, 10) (11, 14) (11, 10) (12, 13) (11, 13) (14, 1)

6.9 Devise a series of equivalence statements for a collection of sixteen items that yields a tree of height 5 when both the weighted union rule and path compression are used. What is the total number of parent pointers followed to perform this series?

6.10 One alternative to path compression that gives similar performance gains is called **path halving**. In path halving, when the path is traversed from the node to the root, we make the grandparent of every other node i on the path the new parent of i. Write a version of **`FIND`** that implements path halving. Your **`FIND`** operation should work as you move up the tree, rather than require the two passes needed by path compression.

6.11 Analyze the fraction of overhead required by the "list of children" implementation, the "left-child/right-sibling" implementation, and the two linked implementations of Section 6.3.3. How do these implementations compare in space efficiency?

6.12 Using the general tree ADT of Figure 6.2, write a function that takes as input the root of a general tree and returns a binary tree generated by the conversion process illustrated by Figure 6.14.

6.13 Use mathematical induction to prove that the number of leaves in a non-empty full K-ary tree is $(K-1)n+1$, where n is the number of internal nodes.

6.14 Derive the formulas for computing the relatives of a non-empty complete K-ary tree node stored in the complete tree representation of Section 5.3.3.

6.15 Find the overhead fraction for a full K-ary tree implementation with space requirements as follows:

- **(a)** All nodes store data, K child pointers, and a parent pointer. The data field requires four bytes and each pointer requires four bytes.
- **(b)** All nodes store data and K child pointers. The data field requires sixteen bytes and each pointer requires four bytes.
- **(c)** All nodes store data and a parent pointer, and internal nodes store K child pointers. The data field requires eight bytes and each pointer requires four bytes.
- **(d)** Only leaf nodes store data; only internal nodes store K child pointers. The data field requires four bytes and each pointer requires two bytes.

6.16
- **(a)** Write out the sequential representation for Figure 6.18 using the coding illustrated by Example 6.5.
- **(b)** Write out the sequential representation for Figure 6.18 using the coding illustrated by Example 6.6.

6.17 Draw the binary tree representing the following sequential representation for binary trees illustrated by Example 6.5:

$$\text{ABD//E//C/F//}$$

6.18 Draw the binary tree representing the following sequential representation for binary trees illustrated by Example 6.6:

$$\text{A}'/\text{B}'/\text{C}'\text{D}'\text{G}/\text{E}$$

Show the bit vector for leaf and internal nodes (as illustrated by Example 6.7) for this tree.

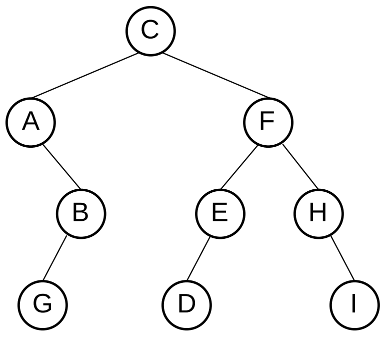


Figure 6.18 A sample tree for Exercise 6.16.

6.19 Draw the general tree represented by the following sequential representation for general trees illustrated by Example 6.8:

XPC)Q)RV)M))))

6.20 **(a)** Write a function to decode the sequential representation for binary trees illustrated by Example 6.5. The input should be the sequential representation and the output should be a pointer to the root of the resulting binary tree.

(b) Write a function to decode the sequential representation for full binary trees illustrated by Example 6.6. The input should be the sequential representation and the output should be a pointer to the root of the resulting binary tree.

(c) Write a function to decode the sequential representation for general trees illustrated by Example 6.8. The input should be the sequential representation and the output should be a pointer to the root of the resulting general tree.

6.21 Devise a sequential representation for Huffman coding trees suitable for use as part of a file compression utility (see Project 5.7).

6.8 Projects

6.1 Write classes that implement the general tree class declarations of Figure 6.2 using the dynamic "left-child/right-sibling" representation described in Section 6.3.4.

6.2 Write classes that implement the general tree class declarations of Figure 6.2 using the linked general tree implementation with child pointer arrays of Figure 6.12. Your implementation should support only fixed-size nodes that do not change their number of children once they are created. Then, reimplement these classes with the linked list of children representation of

Figure 6.13. How do the two implementations compare in space and time efficiency and ease of implementation?

6.3 Write classes that implement the general tree class declarations of Figure 6.2 using the linked general tree implementation with child pointer arrays of Figure 6.12. Your implementation must be able to support changes in the number of children for a node. When created, a node should be allocated with only enough space to store its initial set of children. Whenever a new child is added to a node such that the array overflows, allocate a new array from free store that can store twice as many children.

6.4 Implement a BST file archiver. Your program should take a BST created in main memory using the implementation of Figure 5.14 and write it out to disk using one of the sequential representations of Section 6.5. It should also be able to read in disk files using your sequential representation and create the equivalent main memory representation.

6.5 Use the UNION/FIND algorithm to implement a solution to the following problem. Given a set of points represented by their xy-coordinates, assign the points to clusters. Any two points are defined to be in the same cluster if they are within a specified distance d of each other. For the purpose of this problem, clustering is an equivalence relationship. In other words, points *A*, *B*, and *C* are defined to be in the same cluster if the distance between *A* and *B* is less than d and the distance between *A* and *C* is also less than d, even if the distance between *B* and *C* is greater than d. To solve the problem, compute the distance between each pair of points, using the equivalence processing algorithm to merge clusters whenever two points are within the specified distance. What is the asymptotic complexity of this algorithm? Where is the bottleneck in processing?

6.6 In this project, you will run some empirical tests to determine if some variations on path compression in the UNION/FIND algorithm will lead to improved performance. You should compare the following five implementations:

(a) Standard UNION/FIND with path compression and weighted union.

(b) Path compression and weighted union, except that path compression is done *after* the UNION, instead of during the FIND operation. That is, make all nodes along the paths traversed in both trees point directly to the root of the larger tree.

(c) Weighted union and path halving as described in Exercise 6.10.

(d) Weighted union and a simplified form of path compression. At the end of every FIND operation, make the node point to its tree's root (but don't change the pointers for other nodes along the path).

(e) Weighted union and a simplified form of path compression. Both nodes in the equivalence will be set to point directly to the root of the larger tree after the UNION operation. For example, consider processing the equivalence (A, B) where A' is the root of A and B' is the root of B. Assume the tree with root A' is bigger than the tree with root B'. At the end of the UNION/FIND operation, nodes A, B, and B' will all point directly to A'.