Trees

1 Introduction

Trees are an abstraction that can be used to represent hierarchical relationships, such as genealogies, evolutionary trees, corporate structures, and file systems.

Previous data types have been linear in nature. Trees are a way to represent a specific kind of non-linear abstraction in which there is a *parent/child* type of relationship. For the moment, ignore the fact that most animals require two parents to create children. Instead assume *asexual reproduction*, or *agamogenesis*. Then every parent can have multiple children, but each child has but a single parent. Let us think about such a species for now.

Consider the set of all individuals in such a specie's population. Define a binary relationship e on this set as follows: for any two members of this set x and y, *e(x,y)* is true if and only if x is the parent of y. We can represent x and y as little circles on a paper. We call such circles ***nodes***. If *e(x,y)* is true, we can connect x and y with an arrow leading from x to y. We call these arrows ***directed edges***. A directed edge from x to y indicates that x is the parent of y, or that y is the child of x. We can also write a directed edge as an ordered pair *(x,y)*.

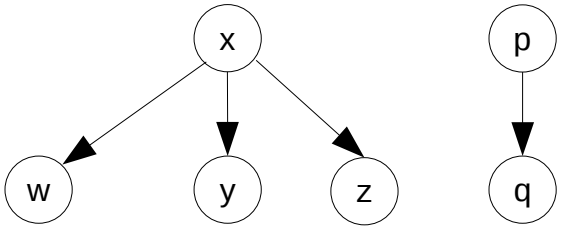


Figure 1: Ordered pairs in a set.

In Figure 1, the directed edges are (x, y), (x, z), (x, w), and (p, q) and nothing else. Thus, x is the parent of w, y, and z, and p is the parent of q. If p and x are both children of a node a, then the population would be represented as in Figure 2.

You can imagine that we could arrange the nodes on a very large piece of paper in such a way that they look like an upside-down tree.

2 Trees

There is a lot of terminology to learn, and many definitions are about to follow. There are two basic types of trees, ***general trees*** and ***n-ary trees***. We begin with general trees.

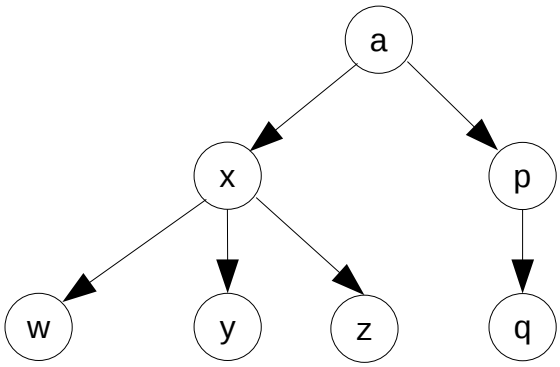


Figure 2: Ordered pairs with common ancestor.

2.1 General Trees

Definition 1. A ***general tree*** T consists of a possible empty set of nodes. If it is not empty, it consists of a unique node r called the ***root*** of T and zero or more non-empty trees $T_1, T_2, \ldots, T_k$ such that there is a directed edge from r to each of the roots of $T_1, T_2, \ldots, T_k$. A ***subtree*** of a tree T is any tree whose root is a node of T.

In Figure 2, a is the root of the tree. It has two subtrees, whose roots are x and p. Notice that in a general tree, the subtrees are non-empty – it has to have at least one node – but that there may be zero subtrees, meaning that a general tree can have exactly one node with no subtrees at all. In Figure 2, the nodes w, y, z, and q are the roots of trees with no subtrees, and p is the root of a tree with a single subtree whose root is q.

Definition 2. A ***forest*** is a collection of non-empty general trees.

Remark. You can always create a tree from a forest by creating a new root node and making it the parent of the roots of all of the trees in the forest. Conversely, if you lop off the root of a tree, what is left is a forest.

2.1.1 Applications of General Trees

In a file system, a node represents each file, and if the file is a directory, then it is an internal node whose children are the files contained in the directory. Some file systems do not restrict the number of files per folder, implying that the number of children per node is varying and unbounded.

In computational linguistics, as sentences are parsed, the parser creates a representation of the sentence as a tree whose nodes represent grammatical elements such as predicates, subjects, prepositional phrases, and so on. Some elements such as subject elements are always internal nodes because they are made up of simpler elements such as nouns and articles. Others are always leaf nodes, such as nouns. The number of children of the internal nodes is unbounded and varying.

In genealogical software, the tree of descendants of a given person is a general tree because the number of children of a given person is not fixed.

A tree need not be drawn in a way that looks like a hierarchy and yet it still is a tree. Figure 3 contains a fragment of the evolutionary tree, with its root at the center of a circle and the subtrees radiating away from the center.

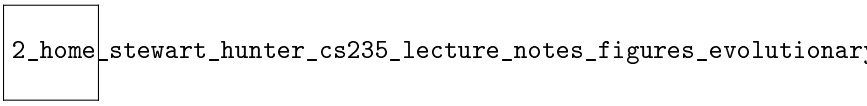


Figure 3: Evolutionary tree: Diagrammatic representation of the divergence of modern taxonomic groups from their common ancestor (from Wikipedia http://en.wikipedia.org/wiki/Phylogenetic_tree.)

2.2 Tree Terminology

The remainder of this terminology applies to all kinds of trees, not just general trees.

Definition 3. A node in a tree is called a ***leaf node*** or an ***external node*** if it has no children.

In Figure 2, the leaf nodes are w, y, z, and q.

Definition 4. A node is called an ***internal node*** if it is not a leaf node.

An internal node has at least one child. In Figure 2, the internal nodes are a, x, and p.

Just as it makes sense to talk about siblings, grandparents, grandchildren, ancestors, and descendants with respect to people, so it is with nodes in a tree.

Definition 5. Two nodes are ***siblings*** if they have the same parent.

Definition 6. The ***grandparent*** of a node is the parent of the parent of the node, if it exists. A ***grandchild*** of a node is a child of a child of that node if it exists. More generally, a node p is an ***ancestor*** of a node t if either p is the parent of t, or there exists a node q such that q is a parent of t and p is an ancestor of q. If p is an ancestor of t, then t is a ***descendant*** of p.

In Figure 2, `a` has 4 grandchildren.

Notice that the definition of an ancestor is recursive. Many definitions of properties or relationships having to do with trees are recursive because a tree is essentially a recursively defined structure.

Definition 7. The ***degree of a node*** is the number of children of that node. The ***degree of a tree*** is the maximum degree of all of the nodes in the tree.

Notice that a tree can have many nodes with small degree, but if one node in it has large degree, then the tree itself has that degree. The degree of a tree is just one of many quantitative properties of a tree.

Definition 8. A ***path*** from a node n_1 to node n_k is a sequence of nodes n_1, n_2, ..., n_k such that n_i is the parent of n_{i+1} for $1 \leq i < k$. The ***length*** of a path is the number of edges in the path, not the number of nodes!

Note 9. Some books will define the length of a path as the number of nodes in the path, not the number of edges. This will have an effect on the remaining definitions and the details of many theorems and proofs of these theorems. It is important to know which definition is being used. In these notes, it is always the number of edges in the path.

If we start at the root and travel down the paths from the root, we can define a notion of the levels of a tree. The root is at the first level, sometimes labeled 0 and sometimes 1. We will label the level of the root 1. The children of the root are at level 2. More generally, the level of a node n in the tree is the level of its parent plus 1. All nodes at a given level are reachable from the root in the same number of steps.

The height of a tree is another concept that is not universally agreed upon. However, the standard definition (which differs from the one used in the textbook), is as follows.

Definition 10. The ***height*** of a node is the length of the longest path from the node to any of its children.

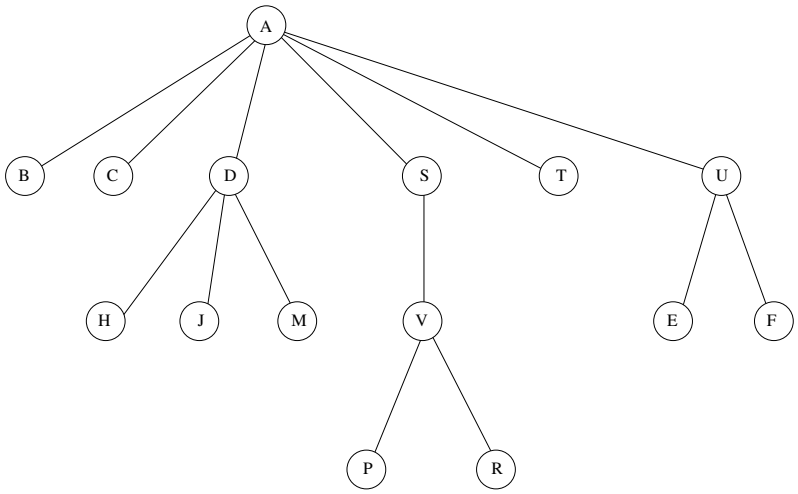


Figure 4: A general tree.

This implies that all leaf nodes have height = 0. The height of the node z in Figure 2 is 0. The height of the node labeled p is 1, and the root has height 2. Again, many books will state that the height of leaf nodes is 1, not 0.

Exercise. Height can be defined recursively. Write its definition.

Definition 11. The ***height*** of a tree is the height of the root node of the tree. Another way to put it is that it is the length of the longest path in the tree.

The height of the tree in Figure 2 is 2 since the longest paths are of length 2.

Definition 12. The ***depth*** of a node is the length of the path from the root to the node. The root has depth 0.

Exercise. Depth can also be defined recursively. Write that definition.

3 Tree Implementations

Because one does not know the maximum degree that a tree may have, and because it is inefficient to create a structure with a very large fixed number of child entries, the most extensible implementation of a general tree uses a linked list to store the children of a node. It is not exactly what you might imagine immediately; it is a bit more subtle. A tree node contains an element and two pointers: one to the leftmost-child of the node and another to the sibling that is to the immediate right of a node:

```
struct TreeNode
{
    Object element;
    TreeNode * firstChild;
    TreeNode * nextSibling;
};
```

Figure 5 illustrates how this structure is used to represent the tree in Figure 5. The advantage of this representation is that this same node can be used to represent all nodes in the tree.

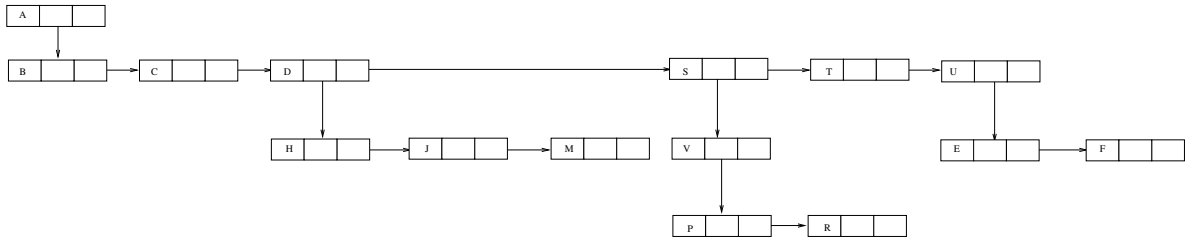


Figure 5: Implementation of tree from Figure 4.

3.1 General Tree Traversal

There are a few different ways to traverse a tree, some more efficient than others depending on the tree implementation. If a tree represents a directory hierarchy, then a ***pre-order traversal*** could be used to print the files and folders in the hierarchy in a natural way, much the way one sees them listed in a file browser window. The following pseudo-code description of such a function does this. It is written as if it were a member function of some class that represents a file (which also includes directories, which are really files.) Assume that `printname(int depth)` is a function that prints the name of the current file indented in proportion to its integer parameter, `depth`.

```
1  void file::listAll( int depth = 0) const
2  {
3      printname(depth);
4      if (isDirectory() )
5          foreach child c in this directory
6              c.listAll(depth + 1);
7  }
```

The algorithm is a pre-order traversal because it visits the root of every subtree (which is a directory) prior to visiting any of the children. The pseudo-code of the algorithm does not specify the order that the foreach loop uses to visit all of the children in a directory, but for the sake of precision, let us assume that they are visited in a "left-to-right" order. Bear in mind that a general tree has no notion of left and right. The easiest implementation will just descend the `firstChild` pointer of the directory node and then travel along the `nextSibling` pointers until it reaches a node that has no `nextSibling` (i.e., its `nextSibling` pointer is null.) If `printname()` prints a word with `depth` many tab characters preceding it, then this will print an indented listing of the directory tree, with files at depth d having d tabs to their left. For the tree in Figure 4, the output of this algorithm would be

```
A
    B
    C
    D
        H
        J
        M
    S
        V
            P
            R
    T
    U
        E
        F
```

Notice that the children are listed in dictionary order, because the children of each node were stored that way in the tree structure implementation.

There is no single notion of in-order traversal because of the fact that the number of subtrees varies from one node to the next and the root may be visited in many positions relative to its children. However, one can define ***post-order traversals*** of general trees. One use of post-order is in computing disk block usage for each directory. For example, the UNIX `du` command will display the amount of disk space used by every file, and cumulatively, for every directory in its command-line argument. In order to do this, it must obtain the usage of the child nodes before the parent node. The general algorithm would be of the form

```
1  int file::disk_usage() const
2  {
3
4      int size = usage(); // some function that counts disk blocks of the current fi
5
6      if (isDirectory() )
7          foreach child c in this directory
8              size = size + c.disk_usage();
9      return size;
10 }
```

The `usage()` function is a member function that returns the number of disk blocks used by the current file. Line 8 contains a recursive call of the `disk_usage()` function on child `c`. There will be no infinite recursion if the directory structure has no links back to ancestors.

4 Binary Trees

4.1 N-ary Trees

An ***n-ary tree*** is not the same thing as a general tree. The distinction is that, in an n-ary tree the degree of any node is bounded by n, i.e., it can never be greater than n. The formal definition is

Definition 13. An ***n-ary tree*** is a set S of nodes that is either empty, or has a distinguished node called the ***root*** and n, possibly empty n-ary subtrees of the root.

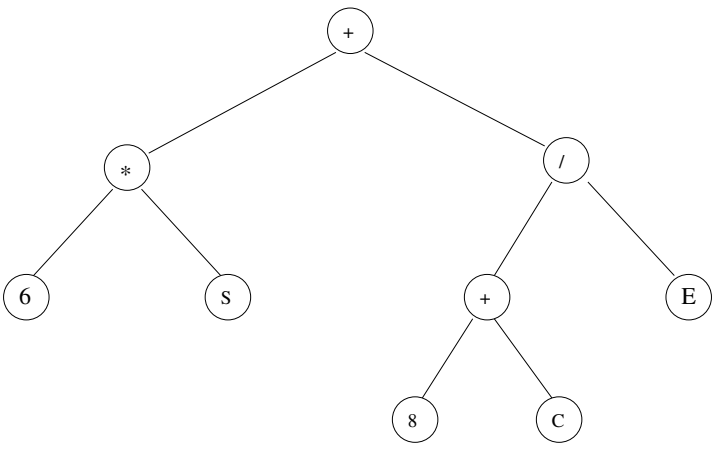
A general tree cannot be empty. It always has at least one node, but it might not have any subtrees. In contrast, an n-ary tree may be empty, but it always has all of its subtrees, which might be empty. So a general tree with one subtree has one subtree, but a 3-ary tree with one non-empty subtree technically has 3 subtrees, two of which are empty.

We will be interested in just the special case of $n = 2$. When $n = 2$ the tree is called a binary tree.

4.2 Binary Trees

A binary tree is not a special kind of tree. General trees do not distinguish among their children, but binary trees do. To be precise,

Definition 14. A ***binary tree*** is either empty, or it has a distinguished node called the ***root*** and a ***left*** and ***right*** binary sub-tree.



Notice that in a binary tree, the subtrees are ordered; there is a left subtree and a right subtree. Since binary trees can be empty, either or both of these may be empty. In a general tree there is no ordering of the sub-trees. The root of the left subtree, if it not empty, is called the ***left child*** of the root of the tree, and the root of the right subtree, if it is not empty, is called the ***right child***.

The most important applications of binary trees are in compiler design and in search structures. One use, for example, in compilers is as a representation of algebraic expressions, regardless of whether they are written in prefix, postfix, or infix. In such a tree, the operator is the root and its left operand is its left subtree and its right operand is its right subtree. This applies recursively to the subtrees. The leaf nodes of the tree are simple operands.

Figure 6: An expression tree representing $6 * S + (8 + C)/E$

The binary tree in Figure 6 is an example of such a tree. It is an unambiguous representation of an algebraic expression. The compiler can use it to construct machine code for the expression.

The other important application of binary trees is as search trees. Informally, a search tree is a tree that is designed to make finding items in the tree "fast". By "fast" we mean better than O(n) on average. Later we will see how this is done.

4.2.1 Binary Tree Properties

There are a few interesting questions that we can ask about binary trees, mostly related to the number of nodes they can contain at various heights, and what heights they might be if we know how many nodes they have.

To start, we give a name to a particular shape of binary tree, the one that has as many nodes as it possibly can have for its height.

Definition 15. A ***full binary tree*** is a tree that, for its height, has the maximum possible number of nodes.

This means that every node that is not a leaf node has two children, and that all leaf nodes are on the same level in the tree. If there were an internal node with just one child, we could add another child without making the tree taller and this would be a tree with more nodes than the maximum, which is impossible. If not all leaf nodes were at the same level, then there would be one either higher or lower than the remaining leaf nodes. If it were lower, we could add children to it without increasing the height, making a tree with more nodes, again impossible. If it were at a higher level, then we could pick any leaf node at a lower level and add a child to it without increasing the height of the tree and this tree would have more nodes, again impossible. Figure 7 depicts a full binary tree of height 3.

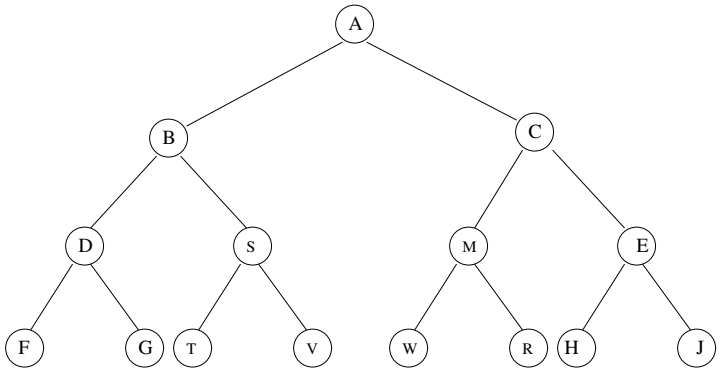


Figure 7: Full binary tree

How many nodes are at depth d in a full binary tree of height h, for $d \leq h$? Let $f(d)$ denote the number of nodes at depth d in a full binary tree. We claim that $f(d) = 2^d$.

Proof. Assume $h \geq 0$. Let $d = 0$. There is a single root node, so $f(0) = 1 = 2^0$. Assume $h > 0$, and that for $d < h$, the hypothesis is true. Then at depth d, there are 2^d nodes. Since each of these nodes has exactly 2 children, there are $2 \cdot 2^d = 2^{d+1}$ nodes at depth $d + 1$. If $d = h$ then there are no nodes at depth $d + 1$. Thus, for all $d \leq h$, $f(d) = 2^d$. □

It follows from this that the number of leaf nodes in a full binary tree of height h is 2^h.

A full binary tree is a good thing. Why? Because all algorithms that do things to binary trees start at the root and have to visit the other nodes by traveling paths from the root. If the tree is packed densely, it means the average path length is smaller than if the same number of nodes were not in a full tree. How many nodes are in a full binary tree of height h? Since each level d has 2^d nodes, there are $1 + 2^1 + 2^2 + 2^3 + \cdots + 2^h = (2^{h+1} - 1)/(2 - 1) = 2^{h+1} - 1$ nodes in a full binary tree. This is important enough to state as a theorem:

Theorem 16. *A full binary tree of height h has $2^{h+1} - 1$ nodes.*

Thus, the number of nodes in full binary trees of ever increasing heights are 1, 3, 7, 15, 31, 63, and so on, corresponding to trees of heights 0, 1, 2, 3, 4, and 5 respectively.

Certain types of binary trees that are "nearly" full are also given a name. One such type of binary tree is called a ***complete binary tree***.

Definition 17. A binary tree of height h is ***complete*** if the subtree of height $h - 1$ rooted at the root of the tree is a full binary tree, and if a node at depth $h - 1$ has any children, then all nodes to the left of that node have two children, and if it has only one child, that child is a left child.

Figure 8 depicts a complete binary tree of height 3. Notice that it is like a full tree with part of its bottom level removed, from the right towards the left. This is how a complete tree must appear. Complete binary trees can be used to implement an abstract data type called a ***priority queue***.

Definition 18. A ***degenerate binary tree*** is a binary tree all of whose nodes except the leaf node has exactly one child.

The tree in Figure 9 is a degenerate tree of height 3. Notice that if we "straightened out" the edges, it would look just like a list. This is what characterizes degenerate trees – they are essentially lists. A degenerate tree has exactly one edge for every node except the single leaf node, so a degenerate tree with n nodes has height$n - 1$.

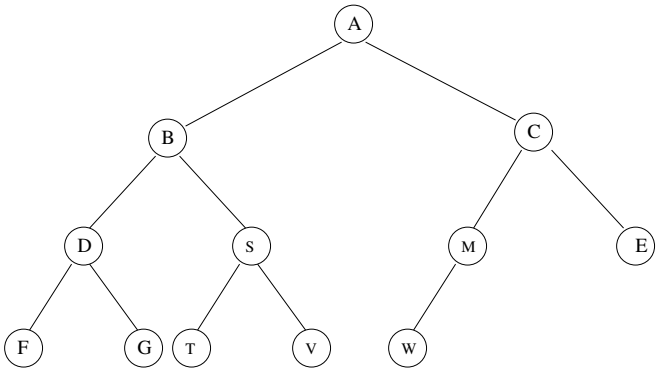


Figure 8: Complete binary tree of height 3.

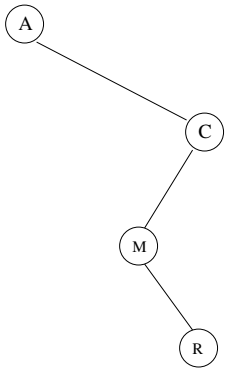


Figure 9: Degenerate binary tree of height 3.

Now we can answer some questions about binary trees.

What is the maximum height of a binary tree with n nodes?

The tallest binary tree with n nodes must be degenerate. If this were false, it would mean there is at least one node with two children. We could remove a child and attach it to a node in the bottom level of the tree, making the tree taller, which contradicts the assumption that this was the tallest tree possible with n nodes. The height of a degenerate tree with n nodes is $n-1$, so we have proved:

Theorem 19. *The maximum height of a binary tree with n nodes is n-1.*

What is the minimum height of a binary tree with n nodes?

Before we derive the answer mathematically, first consider an example. Suppose n=24. The tallest full binary tree with less than or equal to 24 nodes is the one that has 15 nodes and is of height 3. Because we cannot add another node to this tree without making it taller, it is clear that the minimum height of a binary tree with 24 nodes must be greater than 3. Now consider the shortest full binary tree that has at least 24 nodes. The tree with 31 nodes is that tree, and its height is 4. Can a binary tree with 24 nodes have height 4? It certainly can. We just have to remove 7 nodes from the full tree of height 4 to create such a tree. We can use this argument more generally.

Let h be the smallest integer such that $n \leq 2^{h+1} - 1$. This implies that $n > 2^h - 1$, because if it were false, then $n \leq 2^h - 1$ would be true and therefore $h - 1$ would be an integer smaller than h for which $n \leq 2^h - 1 = 2^{(h-1)+1} - 1$. This h is therefore the unique integer for which $2^h - 1 < n \leq 2^{h+1} - 1$. Stated in terms of binary trees, h is the height of the smallest full binary tree that has at least n nodes, and $h - 1$ is the height of the tallest full binary tree that has strictly less than n nodes. This implies that a tree with n nodes must be strictly taller than $h - 1$, but may be of height h, because a binary tree of height h can have up to $2^{h+1} - 1$ nodes.

It follows from the preceding argument that the minimum height of a binary tree with n nodes is the smallest integer h for which $n \leq 2^{h+1} - 1$. We can determine h as a function of n as follows:

$$\begin{array}{rccc} & 2^h - 1 & < n & \leq 2^{h+1} - 1 \\ \iff & 2^h & < n+1 & \leq 2^{h+1} \\ \iff & h & < \log_2(n+1) & \leq h+1 \end{array}$$

If $\log_2(n+1) = h+1$ then $h+1 = \lceil \log_2(n+1) \rceil$, and $h = \lceil \log_2(n+1) \rceil - 1$. If $\log_2(n+1) < h+1$, then $h = \lceil \log_2(n+1) \rceil - 1$. This proves

Theorem 20. *The minimum height of a binary tree with n nodes is* $\lceil \log_2(n+1) \rceil - 1$.

Note. It is also true that $\lceil \log_2(n+1) \rceil - 1 = \lfloor \log_2 n \rfloor$.

5 The Binary Tree ADT

There are several operations that a binary tree class should support. There are the usual suspects, such as creating empty trees, destroying trees, and perhaps getting properties such as their height or the number of nodes they contain. In addition, we need methods of inserting new data, removing data, and searching to see if a particular data item is in the tree. These are the same types of methods that lists supported, and we will include them in our interface.

Unlike the abstract data types based on lists, binary trees also need to provide traversals – methods of visiting every node in the tree, perhaps to print out their contents, to replicate a tree, or to modify the data in all nodes. In general, a binary tree should allow the client to supply a processing function to a traversal so that as the traversal visits each node it can apply that function to the node. Languages like C and C++ allow function parameters, so this is possible, and we will see how to do this later.

For now we start by exploring the different ways to traverse a binary tree, and only after that do we flesh out the interface for a binary tree abstract data type.

5.1 Binary Tree Traversals

There have to be systematic, i.e., algorithmic, methods of processing each node in a binary tree. The type of processing is irrelevant to the algorithm that traverses the tree. It might be retrieving a value or modifying a value in some specific way. We can assume that there is some specific function, named `visit()`, that is applied to each node as it is visited by the traversal algorithm.

There are three important algorithms for traversing a binary tree: ***in-order***, ***pre-order***, and ***post-order***. They are easy to describe recursively because binary trees are essentially recursively defined structures.

All traversals of a binary tree must visit the root and all nodes in its left and right subtrees. If these traversals are described recursively, then there are three different steps that can take place:

1. visit the root
2. visit the left subtree (in the same order as the tree rooted at the root is visited)
3. visit the right subtree (in the same order as the tree rooted at the root is visited)

Because these three actions are independent, there are six different permutations of them:

```
1, 2, 3
1, 3, 2
2, 1, 3
2, 3, 1
3, 1, 2
3, 2, 1
```

In three of these the right subtree is visited before the left subtree, and the traversals is otherwise analogous to the three in which the left subtree is visited before the right subtree. Because visiting the left subtree before the right subtree is usually more useful, the three orderings in which the right is visited before the left are not usually discussed. The remaining ones, with left preceding right, are the ones customarily used. They are:

```
1, 2, 3
2, 1, 3
2, 3, 1
```

and these three permutations correspond to the following:

5.1.1 Pre-Order:

- visit the root
- visit the left subtree
- visit the right subtree

5.1.2 In-Order:

- visit the left subtree
- visit the root
- visit the right subtree

5.1.3 Post-Order:

- visit the left subtree
- visit the right subtree
- visit the root

In each of these it is implicit that the visits to the subtrees are carried out recursively using the same permutation of actions as at the top level. For example, an in-order traversal of the tree in Figure 6 would first go left from the root to the node labeled "*". It does not apply the `visit()` function to this node yet, but would instead apply recursively and go left again to the node labeled "6". Since this node has no left child, the attempt to go left returns back to the 6, visits 6, attempts to visit the right child of 6, which is empty, and therefore backs up to the parent of 6, which is *. It now visits * and then goes to the right child of *, which is S. After processing S, it backs up to * and then backs up to the root node +, which it now visits. The same logic happens on the right hand side of the tree. We will work through some complete examples shortly.

If we assume for the moment that a tree node has the representation

```
struct tree_node
{
    item_type   item;
    tree_node   *left;
    tree_node   *right;
};
```

and that a `visit()` function is defined using the following `typedef` statement[1]

```
typedef void (*visit_function)(item_type data);
```

then the three traversals can be defined precisely by the following functions

```
void in_order ( binary_tree *t, visit_function visit )
{
    if ( t != NULL ) {
        in_order( t->left,  visit);
        visit(t->item);
        in_order( t->right, visit);
    }
}

void pre_order ( binary_tree *t, visit_function visit )
{
    if ( t != NULL ) {
        visit(t->item);
        pre_order( t->left,  visit);
        pre_order( t->right, visit);
    }
}
void post_order ( binary_tree *t, visit_function visit )
{
    if ( t != NULL ) {
        post_order( t->left,  visit);
        post_order( t->right, visit);
        visit(t->item);
    }
}
```

5.1.4 Examples

Given the binary tree in Figure 10, the pre-order, in-order, and post-order traversals are as follows. Work through them carefully to understand how they are applied.

Pre-order: 50, 25, 15, 40, 30, 80, 75, 90, 85

In-order: 15, 25, 30, 40, 50, 75, 80, 85, 90

Post-order: 15, 30, 40, 25, 75, 85, 90, 80, 50

[1] If this `typedef` is new to you, the way to read it is that the name of the function is what is being defined. The `typedef` tells the compiler that the symbol "`visit_function`" is the name of a function type, and that this function type has a signature consisting of a `void` return type and a single parameter of type `item_type`. Any place where an object is declared to be of type `visit_function`, the compiler will expect that object to be used as a function with this signature.

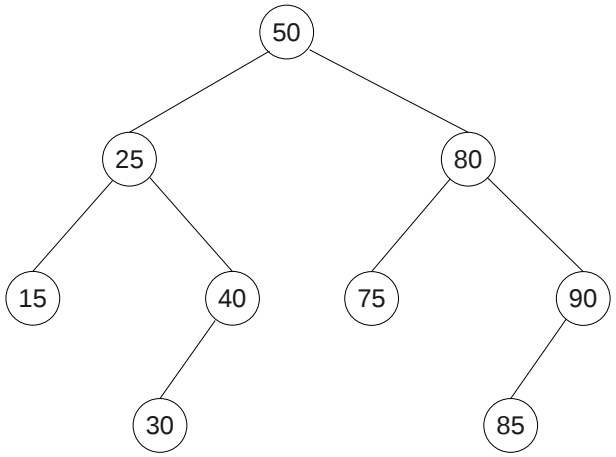


Figure 10: A binary tree

Observations

- Notice that among them, the in-order traversal visits the nodes in such a way that they are in sorted order. This is not a coincidence; it it because the tree in Figure 10 is a ***binary search tree***. In Section 7, we will see that an in-order traversal of a binary search tree always visits the nodes in ascending order.
- Also note that each of these traversals visits each node exactly once and that therefore, they are $O(n)$ algorithms, where n is the number of nodes in the tree.
- The function that is called to process each node, the `visit()` function, must use only the exposed operations of the binary tree. If the traversals are member functions of a binary tree class, then they provide a means for client code to traverse the tree, but not the means to access private data.

5.2 The Binary Tree ADT Interface

We can now return to designing an interface. The following is a list of operations that ought to be exposed.

1. Create a new, empty binary tree.
2. Create a one-node binary tree. This is a convenient function to give to clients.
3. Destroy a binary tree, i.e., deallocate all of its resources.
4. Insert data into the root of the tree.
5. Get data from the root of the tree.
6. Append an existing binary tree as the left (or right) subtree of the root.
7. Get the left (or right) subtree of the root.
8. Detach the left (or right) subtree and save as a new binary tree.
9. Copy one binary tree to another.
10. Return the height of the tree.

11. Return the number of nodes in the tree.
12. Traverse the tree in-order.
13. Traverse the tree pre-order.
14. Traverse the tree post-order.

There may be other useful methods, but they can be derived from these. For example, it may be useful to provide a method that constructs a binary tree from the data for a root and two existing binary trees, one left and one right. That can be derived using methods 2, 4, and 6 above. The above list does not contain a method to attach a single node as the left or right child of the root. That operation can be carried out by creating a one node binary tree and using method 6. A formal ADT for a binary tree is specified as follows.

```
create_binarytree([in] item_type) throw tree_exception
// Create a one-node tree whose root contains the specified item.
// Throw an exception if this fails.
// This is  a constructor.

create_binarytree([in] item_type new_item,
                  [inout] binary_tree left_subtree,
                  [inout] binary_tree right_subtree) throw tree_exception
// Create a binary tree whose root contains the new_item and whose
// left and right subtrees are left_subtree and right_subtree respectively.
// Throw an exception if this fails.
// On return it releases the references to the left and right subtrees.
// This is a constructor.

destroy_binarytree()
// Destroy a binary tree.
// This is a destructor. It deallocates all memory belonging to the tree.

int height() const
// This returns the height of the tree.
int size() const
// This returns the number of nodes in the tree.

void get_root_item([out] item_type root_item) const throw tree_exception
// Returns the item stored in the root of the tree.
// Throws an exception if the tree is empty.

void set_root-item([in] item_type new_item )  throw tree_exception
// Stores new_item into the root of the tree, replacing any value if it exists.
// Throws an exception if the tree is empty.

void attach_left_subtree( [inout] binary_tree left_tree ) throw tree_exception
// If the tree is not empty and the root does not have a left child,
// the left_tree is attached as the left subtree of the root, and the
// reference to it is removed.
// Otherwise an exception is thrown.

void attach_right_subtree( [inout] binary_tree right_tree ) throw tree_exception
// If the tree is not empty and the root does not have a right child,
// the right_tree is attached as the right subtree of the root, and the
// reference to it is removed.
```

```
// Otherwise an exception is thrown.

void detach_left_subtree( [out] binary_tree left_tree ) throw tree_exception
// If the tree is not empty, the left subtree is detached from the root
// and a reference to it is returned in the left_tree parameter. If the
// left subtree is empty a NULL reference is returned.
// Otherwise an exception is thrown.

void detach_right_subtree( [out] binary_tree right_tree ) throw tree_exception
// If the tree is not empty, the right subtree is detached from the root
// and a reference to it is returned in the right_tree parameter. If the
// right subtree is empty a NULL reference is returned.
// Otherwise an exception is thrown.

binary_tree get_left_subtree() const throw tree_exception
// If the tree is not empty, a copy of the left subtree is returned.
// Otherwise an exception is thrown.

binary_tree get_right_subtree() const throw tree_exception
// If the tree is not empty, a copy of the right subtree is returned.
// Otherwise an exception is thrown.

binary_tree copy() const
// Return a copy of the tree.

void pre_order_tree ([in] visit_function visit);
// Traverse the tree using the pre-order algorithm, applying the function
// visit() to each node as it is visited.

void in_order_tree  ([in] visit_function visit);
// Traverse the tree using the in-order algorithm, applying the function
// visit() to each node as it is visited.

void post_order_tree([in] visit_function visit);
// Traverse the tree using the post-order algorithm, applying the function
// visit() to each node as it is visited.
```

6 Implementation of a Binary Tree

Although it is possible to implement a binary tree with an array, we will focus on a pointer-based implementation. We begin with the basic building block: the tree node.

Modifying the preceding definition of a tree node that was based on a C struct, adding a private constructor (which will be justified shortly), and declaring the binary tree class to be a friend class:

```
class tree_node
{
private:
    tree_node ()   {};
    // Non-default constructor:
    tree_node (const  item_type  & node_item,
               tree_node *left_tree  = NULL,
               tree_node *right_tree = NULL):
```

```
                    item(node_item),
                    left (left_tree),
                    right (right_tree) {}
        item_type    item;
        tree_node   *left;
        tree_node   *right;
        friend class binary_tree;
    };
```

The `tree_node` class is not exposed to any client software. It has no public members, not even a constructor. Without a public constructor, no code can create an instance of it, unless that code belongs to a friend of this class. By making the `binary_tree` class a friend of the `tree_node` class, we allow the `binary_tree` class to create instances of tree nodes, which is how the tree can grow.

6.1 Binary Tree Class Interface

The `binary_tree` class interface is given below. The protected methods are not declared as private, so that a class that is publicly derived from derived from this class can access those methods. In addition, the pre- and post-conditions for each method are omitted to save space, because it is essentially a refinement of the ADT described earlier.

```
// this declares a function type that can be applied to the item stored in a tree node
typedef void (*visit_function)(item_type& );

class binary_tree
{
public:
    // constructors:

    binary_tree  ();                              // default
    binary_tree  ( const item_type & root_item); // param-1 constructor given item
    binary_tree  ( const item_type & root_item,
                   binary_tree    & left_tree,
                   binary_tree    & right_tree); // param-3 constructor
    binary_tree  ( const binary_tree    & tree); // copy constructor

    // destructor:
     ~binary_tree();

    // binary tree operations:
    int         size()       const;
    int         height()     const;
    item_type   get_root()   const                              throw(tree_exception);
    void        set_root(const item_type & new_item)            throw(tree_exception);

    void        attach_left_subtree (binary_tree & left_tree)  throw(tree_exception);
    void        attach_right_subtree(binary_tree & right_tree) throw(tree_exception);

    void        detach_left_subtree(binary_tree & left_tree)  throw(tree_exception);
    void        detach_right_subtree(binary_tree & right_tree) throw(tree_exception);

    binary_tree get_left_subtree () const;
```

```
    binary_tree get_right_subtree() const;
    // traversals: (just wrappers)
    void        pre_order  (visit_function visit);
    void        in_order   (visit_function visit);
    void        post_order (visit_function visit);

protected:
    binary_tree (tree_node *nodePtr);  // private constructor

    // A function to copy the tree to a new tree
    void         copy_tree  (tree_node *tree_ptr,
                             tree_node *& new_ptr) const;
    // private function called by public destructor
    void         destroy    (tree_node *& tree_ptr);

    // returns pointer to root
    tree_node   *get_root_ptr() const;

    // set root pointer to given value
    void        set_root_ptr (tree_node *new_root);

    // Given a pointer to a node, retrieve pointers to its children
    void        get_children (tree_node *node_ptr,
                             tree_node *& left_child,
                             tree_node *& right_child) const;

    // Given a pointer to a node, set pointers to its children
    void        set_children (tree_node *nodePtr,
                             tree_node *left_child,
                             tree_node *right_child);
    // Helpers for size and height
    int         get_height( tree_node *tree) const;
    int         get_size( tree_node *tree)   const;

    // The three private traversal functions. These are recursive
    // whereas the public ones are just wrappers that call these.
    // These need access to pointers whereas public cannot have it.
    void        pre_order    (tree_node *treePtr,
                              visit_function visit);

    void        in_order     (tree_node *treePtr,
                              visit_function visit);

    void        post_order   (tree_node *treePtr,
                              visit_function visit);
private:
    // pointer to root of tree
    tree_node *root;
};
```

6.2 Implementation of Methods

The textbook contains the implementations of all of these functions. In general, they are all very simple. They all are either recursive, or perform a simple task such as getting or setting the private members of the object. What makes them a bit more than trivial is that they all handle the various error conditions that can arise by throwing exceptions. The error handling code is the majority of the code. Another bit of complexity has to do with holding and releasing references, which is explained below.

Most tree algorithms are expressed easily using recursion. On the other hand, to do so requires that the parameter of the algorithm is a pointer to the root of the current node, but this is a problem, because one does not want to expose the node pointers to the object's clients . Nodes should be hidden from the object's clients. Put another way, the clients should not know and not care how the data is structured inside the tree; in fact, they should not even know that it is a tree! The binary tree should be a black box with hooks to the methods it makes available to its clients, and no more. Therefore, the sensible solution is to create a "wrapper" method that the client calls that wraps a call to a recursive function. This is exactly how the traversals, the copy function, the destructor, and the functions that get subtrees work. Although I omit the implementations of most of these functions, you should not skip them; you have to make sure you understand how this class implementation works!

What follows demonstrates a typical method that wraps a private (or protected) recursive method; in particular we implement the destructor. First we define a recursive, protected `destroy()` function that uses a post-order traversal to delete all of the nodes of the tree. It has to be post-order because it first has to delete the nodes of the subtrees, and only then can it remove the root of the tree. If we deleted the root first, we could not get to the subtrees, unless we saved the pointers to them. This is inefficient.

```
binary_tree::destroy (tree_node *& tree_ptr)
{
    // postorder traversal
    if (tree_ptr  !=  NULL) {
        destroy (tree_ptr->left);  //
        destroy (tree_ptr->right);
        delete tree_ptr;
        tree_ptr = NULL;
    }
}
```

Having defined this hidden, recursive function, the destructor is simply

```
binary_tree::~binary_tree()
{
    destroy(root);
}
```

As a second example, we implement the copy constructor. First we define the recursive, hidden function that copies a tree:

```
void binary_tree::copy_tree(tree_node *tree_ptr,
                            tree_node *& new_ptr) const
{
    // preorder traversal
    if (tree_ptr != NULL) {
        // copy node
        new_ptr   = new tree_node(tree_ptr->item, NULL, NULL);
        copy_tree  ( tree_ptr->left,  new_ptr->left);
```

```
            copy_tree  ( tree_ptr->right, new_ptr->right);
        }
        else
            new_ptr = NULL;  // copy empty tree
    }
```

Unlike the `destroy()` function, this uses a pre-order traversal, because it first has to create the root node of the new tree and only then can it create its subtrees. Having defined this function, the public copy constructor is trivial:

```
binary_tree::binary_tree( const binary_tree & tree); // copy constructor
{
    copy_tree(tree.root, root);
}
```

Before we look at some more challenging functions, we dispense with the problem of getting the tree's height. Again we wrap the recursive helper function by the public method. The recursive height-computing function is

```
int  binary_tree::get_height( tree_node *tree) const
{
    if ( NULL == tree  )
        return 0;
    else
        return 1 + max (get_height(tree->left), get_height( tree->right));
}
```

In other words, the height of a binary tree is defined recursively; it is one more than the heights of the larger of its two subtrees. The public method is just

```
int binary_tree::height( const binary_tree & tree) const
{
    return get_height(root);
}
```

Exercise 21. The size of a tree is also defined recursively. How?

The harder methods are those that attach and detach subtrees. An implementation of a method to attach a left subtree to a root node checks first for whether the root is a `NULL` pointer. If so it throws an exception. If not, it checks whether there exists a left subtree already. If so, it throws a different exception. If not, it sets the left child pointer of the root to point to the root of the argument tree, and then it sets the argument tree pointer to `NULL`, to prevent the client code from being able to manipulate the subtree internally after it has been attached to the tree:

```
void binary_tree::attach_left_subtree (binary_tree & left_tree) throw(tree_exception);
{
    if (0 == size() )
        throw tree_exception("Tree Exception: Empty tree");
    else if (root->left != NULL)
        throw tree_exception ("TreeException: Cannot overwrite left subtree");
    else {
        root->left = left_tree.root; // the pointer, not the node
        left_tree.root = NULL;       // prevent client from accessing tree
   }
}
```

To be clear, the argument is passed by reference. The `left_tree` parameter is not a copy of the tree to be attached but the actual tree. The assignment

```
root->left = left_tree.root
```

makes the `left` pointer in the binary tree on which this is called get a copy of the `root` pointer of the `left_tree` passed to the function. Now `root->left` points to the tree's root node. After the call we want to make sure that the client code cannot access this tree anymore. Setting `left_tree.root` to `NULL` ensures that this tree's nodes can no longer be accessed because `left_tree` was passed by reference.

Detaching the left subtree and providing it as a standalone tree to the caller requires first checking that the tree is not empty, and if not, using the private binary tree constructor (the one that is given a pointer to a root and constructs a tree from it) and assigning the newly created tree to the argument, after which the left child pointer is set to `NULL`, to detach it from the tree:

```
void binary_tree::detach_left_subtree (binary_tree & left_tree)  throw(tree_exception)
{
    if (is_empty())
        throw tree_exception("TreeException: Empty tree");
    else    {
       left_tree = binary_tree(root->left);
       root->left = NULL;
   }
}
```

In both of these functions, what happened was that a reference (i.e., a pointer) to a subtree was transferred either from client to tree or vice versa. The total number of references remained one: either the client lost it and the tree gained it or vice versa. The idea of making sure that no more than a single code entity holds a reference to an object is a way to reduce errors in code.

Providing a copy of a left or right subtree is a different story. In this case, there is no need to remove a reference because the data structure is being replicated. The code for this follows. Notice that the copy function is the private copy function and that its arguments are pointers to tree nodes. The function has to use a constructor to convert the root pointer to a tree in the final return statement.

```
binary_tree binary_tree::get_left_subtree() const
{
    tree_node *subtree;
    if (is_empty())
        return binary_tree();
    else    {
        copy_tree(root->left, subtree);
        return binary_tree(subtree);
   }
}
```

Lastly, this is a small piece of code to demonstrate how to return pointers in the private code. Notice that in order to pass the value of a pointer back to the caller, the pointer itself must be passed by reference (in C++, or as a double pointer in C).

```
void get_children (tree_node *node_ptr,
                   tree_node *& left_child,
```

```
                    tree_node *& right_child) const
{
    left_child  = nodePtr->left;
    right_child = nodePtr->right;
}
```

7 Binary Search Trees

Let S be a set of values upon which a total ordering relation, $<$, is defined. For example, S can be a set of numbers or strings. A ***binary search tree*** (***BST***) T for the ordered set (S,$<$) is a binary tree with the following properties:

- Each node of T has a value called its *label*. If p and q are nodes, then we write $p < q$ to mean that the label of p is less than the label of q.
- For each node $n \in T$, if p is a node in the left subtree of n, then $p < n$.
- For each node $n \in T$, if p is a node in the right subtree of n, then $n < p$.
- For each element $s \in S$ there exists a node $n \in T$ such that $s = n$.

Binary search trees are binary trees that store elements in such a way that insertions, deletions, and search operations never require more than $O(h)$ operations, where h is the *height* of the tree. Minimally, a binary search tree class should support insertion, deletion, search, a test for emptiness, and a find-minimum operation. A find-minimum is useful because very often one needs to know the smallest value in a set. It would also be useful to support a list-all operation that lists all elements in sorted order. Since a binary search tree is a container, it also needs to provide methods for creating an empty instance, for making a copy of an existing tree, and for destroying instances of trees.

Observe from the definition of a BST that every node in the tree is the root of a BST, and that the left and right subtrees of a node are also BSTs. This implies that many algorithms for processing BSTs can be stated recursively and intuitively.

7.1 Examples

There can be many different binary search trees for the same data set. The topology depends upon the insertion algorithm, the deletion algorithm, and the order in which the keys are inserted and deleted. The three trees shown in Figure 11 each represent the set $S = \{15, 20, 25, 30, 40, 50, 75, 80\}$. The tree in Figure 11(a) is of height 3, that in Figure 11(b) has height 4, and the one in Figure 11(c) has height 2. You will notice that there cannot be a binary search tree of height less than 2 for this set because it has 7 elements, and a full tree of height 1 has 3 elements, so the shortest tree that can contain 7 elements is the full tree of height 2. Of course, there could be an even worse tree than the ones shown, of height 6. There can be many of these, actually. (How many?)

Exercise 22. How many different binary search trees are there for a set of n unique elements? Could it be related to the number of permutations of the set? In other words, does each distinct permutation of the set correspond to a unique binary search tree? Could there be more or fewer trees for a given permutation? Try it with $n = 3$ and $n = 4$ to start.

7.2 Algorithms

The algorithms we need to consider are the find operation (searching), insertion, deletion, and find_minimum.

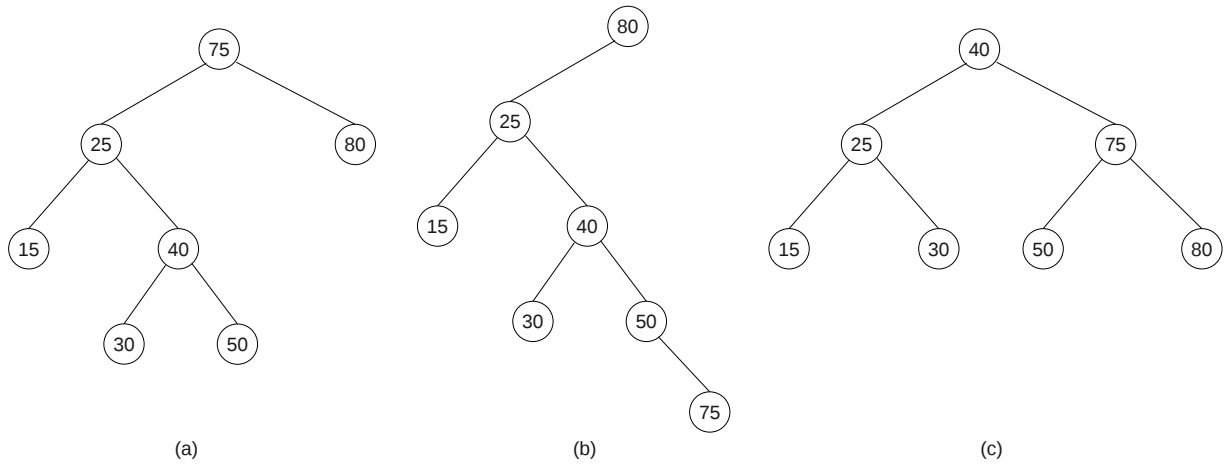


Figure 11: Three different binary search trees for the same data set.

Searching

Consider first of all how to search for an item in a binary search tree. A recursive algorithm is of the form

```
search( current_node, item)
{
    if ( the current node is empty )
        return an indication that the item is not in the tree;
    else {
        if (  item <   current node's item )
            search ( left subtree, item);
        else if (  item >  current node's item )
            search ( right subtree, item);
        else
            the item is in the current node;
    }
}
```

The test for emptiness must be done first of course. If the node is empty, it means that the search has descended the tree to the place where the item should be found if it were in the tree, but that it is not there. For example, if we searched the tree in Figure 10 for the key 77, we would compare 77 to 50 and descend the right subtree, comparing 77 to 80 next. Because $77 < 80$, we descend the left subtree and compare it to 75. Because $77 > 75$, we descend the right subtree of 75, which is empty. At this point we discover that 77 is not in the tree.

This search algorithm descends a level each time it compares the key to a node in the tree. Therefore, in the worst case, it will compare the key to h values (up to $2h$ comparisons), where h is the height of the tree. Since the height of the tree can be proportional to the number of elements in the worst case, this search will take $O(n)$ steps in the worst case.

Insertion

To insert an element in a tree, we should search for it, and if we do not find it, then the place at which we should have found it is the place at which we want to insert it. The tricky part is that, once we have found it is not there, we have lost a pointer to the node whose child it should be. Careful programming can prevent this. The following pseudo-code includes some actual C++ code so that you can see how this is handled.

```
void insert(tree_node * & current, item_type new_item )
{
    if ( current is empty )
         current = new node containing new_item ;

    else if ( new_item < current->item )
         insert( current->left, new_item );

    else if ( new_item > current->item )
         insert( current->right, new_item );

    else // do nothing because it is already in tree
         ;
}
```

The `insert` function is passed by reference a pointer to the root of a tree in which to search for the item. If that pointer is `NULL`, then a new node is allocated and the address is stored in that call-by-reference pointer. Because it is passed by reference, in the recursive calls, the pointers `current->left` and `current->right` will contain a pointer to the newly allocated node when the call returns, assuming they were `NULL` before the call. This is how the new node is attached to the tree in the correct place.

The `insert` function is the means by which trees can be created. Start with an empty tree and call insert on it to fill it with the data from the input source. Since the data is always added at the end of an unsuccessful search, as new leaf nodes, the insertions ultimately grow new levels in the tree. In other words, the first item is placed at the root, the second is either the left or right child of the root, the third might be a grandchild or the child that the previous item did not become. Each successive insertion either fills out an existing level or starts a new one.

Deletion

Deleting an item is more complex than insertion, because of the possibility that the item to be deleted is not a leaf node. In other words, when the item to be deleted is a leaf node, it is pretty simple – just delete that node. If the item has a single child, there is also a relatively simple task to be performed: delete the node and make the only child the child of the node's parent (i.e., let the parent of the deleted node adopt that node's child). If the item, however, has two children, then it is more complex: find the smallest node in the node's right subtree and copy it into the node, effectively deleting that element. Then delete the node that it came from. That node cannot possibly have two children because if it did, one would have to be smaller than the node, contradicting the assumption that it was the smallest node in the right subtree.

A pseudo-code version of the deletion algorithm is

```
void delete(tree_node * & current, item_type item_to_delete )
{
    if ( current is empty )
         return;  // the item was not found

    else if ( item_to_delete < current->item )
         delete( current->left, item_to_delete );

    else if ( item_to_delete > current->item )
         delete( current->right, item_to_delete );

    else {
        // item is equal to the item in the node; it is found
```

```
            // Check how many children it has
            if ( current->left != NULL && current->right != NULL ) {
                // It has two children. We need to replace it by the
                // smallest item in the right subtree. Assume there
                // is a function, find_min() that returns a pointer
                // to the smallest item in a tree.

                // get the pointer to the smallest item in right subtree
                temp_ptr = findMin( current->right );

                // Copy the item into the current node
                current->item = temp_ptr->item;

                // Recursively call delete to delete the item that was just
                // copied. It is in the right subtree.
                delete( current->right, current->item  );
            }
            else {
                // The current node has at most one child. Copy the value of
                // current temporarily
                old_node = current;

                // If the left child is not empty, then make the left child the
                // child of the parent of current. By assigning to current this
                // achieves that.
                // If the left child is empty, then either the right is empty or it is not
                // In either case we can set current to point to its right child.
                if ( current->left != NULL )
                    current = current->left;
                else
                    current = current->right;

                // Delete the node that current used to point to
                delete old_node;
            }
        }
    }
```

The above deletion algorithm depends on a find_min function. Finding the minimum in a tree is a relatively easy task, since it is always the leftmost node in the tree. By "leftmost", we mean that it is reached by traveling down the left-child edges until the left-child edge is `NULL`. The node whose left-child pointer is `NULL` is the minimum in the tree. This can be expressed recursively as follows:

```
    find_min( tree_node *current )
    {
        if ( current == NULL )
            return NULL;

        if ( current->left == NULL )
            return current;

        return find_min( current->left );
    }
```

This could be done iteratively as well, because it is a simple example of ***tail recursion***, i.e., recursion at the end of the function.

I pointed out in an earlier example that the in-order traversal of a binary search tree always results in visiting the nodes in sorted, ascending order. We can formalize this:

Theorem. *An in-order traversal of a binary search tree visits the nodes in ascending sorted order.*

Proof. This can be proved by induction on the height of the tree. It is trivially true for a tree of height 0. Suppose it is true of all binary search trees of height at most h. Let T be a BST of height h+1. Then T consists of a root, a left subtree, and a right subtree. Since the left and right subtrees are each at most height h, an in-order traversal of them visits their nodes in sorted order. Since the in-order visits the left subtree first, then the root, and then the right subtree, and since the nodes in the left subtree are all smaller than the root, which is smaller than all nodes in the right subtree, the sequence of nodes visited is in ascending order. □