Trees

R. J. Renka

Department of Computer Science & Engineering
University of North Texas

10/14/2011
4.1 Preliminaries

**Defn:** A (rooted) *tree* is a directed graph (set of nodes and edges) with a particular distinguished node (*root*) $r$ such that every other vertex can be reached from $r$ by a unique path.

A tree can also be defined recursively as a collection of nodes which, if not empty, consists of a root node $r$ and zero or more nonempty subtrees, each of whose roots are connected by a directed edge from $r$.

- The root of each subtree is a *child* of $r$; $r$ is a *parent* of each subtree root.
- For $N$ nodes there are exactly $N - 1$ edges because each node other than the root is connected to its parent and only its parent.
- Nodes with no children are *leaves*; nodes with the same parent are *siblings*. 
A path from $n_1$ to $n_k$ is a sequence of nodes $n_1, \ldots, n_k$ such that $n_i$ is the parent of $n_{i+1}$ for $1 \leq i < k$. The length of the path is the number of edges $k - 1$. There is a path of length 0 from every node to itself. There is exactly one path from the root to each node. The depth of node $n_i$ is the length of the unique path from $r$ to $n_i$, and $r$ therefore has depth 0. The height of $n_i$ is the length of the longest path from $n_i$ to a leaf. The height of a tree is the height of its root. The depth of a tree is that of its deepest leaf — the height. If there is a path from $n_1$ to $n_2$, $n_1$ is an ancestor of $n_2$, and $n_2$ is a descendant of $n_1$. If $n_1 \neq n_2$, $n_1$ is a proper ancestor of $n_2$ and $n_2$ is a proper descendant of $n_1$. 
Figures 4.1-4.2

**Figure 4.1** Generic tree

**Figure 4.2** A tree
Properties of the Tree in Figure 4.2

Identify the following in the tree of Figure 4.2

- root
- parent of F
- children of F
- leaves
- grandparent of H
- grandchildren of E
- path length from A to K
- depth of K
- height of K
- tree height
- tree depth
Since the number of children varies from 0 to $N - 1$ and is not known until the tree is constructed (and may change), it would be wasteful of space to reserve storage for a link to each child in each node. A better solution is to store the children in a linked list with tree nodes defined as follows.

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

**Figure 4.3** Node declarations for trees
Figure 4.4 First child/next sibling representation of the tree shown in Figure 4.2
4.1.2 Hierarchical File System

A hierarchical file system is a tree in which each node is either a directory or, in the case of leaves, a file. A pathname such as `/usr/mark/book/ch1.r` has root `/usr` and an edge for each additional `/`. Files in different directories may have identical names because the pathnames (paths from the root) are different. A directory is actually just a file that lists its children (but also includes links to itself and its parent (`.` and `..`) which violate the tree structure).

![UNIX directory](image)

**Figure 4.5** UNIX directory

R. J. Renka | Trees
void FileSystem::listAll( int depth = 0 ) const
{
    printName( depth ); // Print the name of the object
    if( isDirectory( ) )
    {
        for each file c in this directory (for each child)
        c.listAll( depth + 1 );
    }
}

**Figure 4.6** Pseudocode to list a directory in a hierarchical file system

**Preorder traversal:** Process the node first, then its children.

**Postorder traversal:** Process the children first, then the node.

**Inorder Traversal** in a binary tree: Process the left child, then the node, then the right child.
Figure 4.7 The (preorder) directory listing
Figure 4.8 UNIX directory with file sizes obtained via postorder traversal

```cpp
int FileSystem::size( ) const
{
    int totalSize = sizeOfThisFile( );

    if( isDirectory( ) )
        for each file c in this directory (for each child)
            totalSize += c.size( );

    return totalSize;
}
```

Figure 4.9 Pseudocode to calculate the size of a directory
Figure 4.10 Trace of the size function traversal
4.2 Binary Trees

**Defn:** A *binary tree* is a tree in which no node has more than two children.

The worst-case (maximum) depth is $N - 1$ when every node but one has one child.

The best case occurs when all non-leaf nodes have two children, and all leaves are on the same level (a *perfect binary tree*): for depth $k$,

$$N = \sum_{i=0}^{k} 2^i = 2^{k+1} - 1 \implies N + 1 = 2^{k+1} \implies$$

$$k + 1 = \log(N + 1) \implies k = O(\log N).$$

The average case (assuming all $N$-node trees are equally likely, where we distinguish between two-node trees with a left child and those with a right child) is depth equal to $O(\sqrt{N})$. Note that there is only one two-node tree, but there are two distinct two-node binary trees.
Figure 4.12 Worst-case binary tree

```c
struct BinaryNode
{
    Object element;   // The data in the node
    BinaryNode *left; // Left child
    BinaryNode *right; // Right child
};
```

Figure 4.13 Binary tree node class (pseudocode)
4.2.2 Expression Trees

An expression tree is a simple case of a parse tree used by a compiler. Nodes are operators and leaves are operands.

Figure 4.14 Expression tree for \((a + b\times c) + ((d\times e + f) \times g)\)

Inorder traversal produces an overly parenthesized *infix* expression. Postorder traversal produces a *postfix* expression

\[
a \ b \ c \ + \ d \ e \ * \ f \ + \ g \ * \ +
\]

Preorder traversal produces a *prefix* expression (used by Lisp):

\[
+ \ + \ a \ * \ b \ c \ * \ + \ * \ d \ e \ f \ g
\]
We saw an algorithm for converting an infix expression to a postfix expression in Chapter 3. We use the postfix expression to construct an expression tree as follows.

```
for each symbol s in the expression sequence
    if (s == operand)
        create a one-node tree; push pointer onto stack
    else    // (s == operator)
        pop T1 pointer
        pop T2 pointer
        create T with root s, left ptr T2, right ptr T1
        push T’s pointer
    endif
endfor
```

**Example:** $a \ b + \ c \ d \ e + * *$
A set of items with a total ordering can be stored in a binary search tree — a binary tree (the structure property) with the following order property: for every node $x$, the values of all items in the left subtree are strictly smaller than the item in $x$, and the values of all items in the right subtree are strictly larger than the item in $x$.

\[ \begin{align*}
\text{6} & \quad \text{2} & \quad \text{8} \\
\text{1} & \quad \text{4} & \quad \text{3} \\
\text{3} & \quad \text{7} & \quad \text{8}
\end{align*} \]

**Figure 4.15** Two binary trees (only the left tree is a search tree)
Figures 4.16 to 4.28 contain an implementation of a binary search tree as a class template with generic type `Comparable`. It will be shown that the average depth of a node in a binary search tree, if all insertion sequences are equally likely, is $O(\log N)$ — much smaller than the average depth of $O(\sqrt{N})$ of a binary tree. Therefore a recursive algorithm based on the recursive tree structure will probably not exceed stack space.

The implementation could be extended to allow duplicates. If the search key is the whole of the data item, we can just add an extra field, for frequency of occurrence, to each node. Otherwise all structures with the same key must be stored in a list.
The `<` operator must be defined for the particular Comparable type.

Any class can have `operator<` defined on one of its data members as, for example, Employee salary in Figure 1.21.

Figure 4.19 demonstrates the use of a function object (Section 1.6.4) as an alternative means of comparison. This adds flexibility for classes that can’t compare themselves, and allows for alternative methods such as case-insensitive string comparison.

The only data member is a pointer to the root node.

Public member functions call private member functions (which use recursion and include a pointer to a BinaryNode as additional argument) with the root as last argument.
In the case of private methods insert, remove, and makeEmpty, the pointer is passed by reference so that it can be changed. (Even root may have to be changed.)

Function findMin uses tail recursion, as does contains, but findMax does not. This just illustrates the choice. Either one could have been implemented either way. Note that the pointer argument t (the copy, since t is passed by value) is changed in findMax. A statement like t->right = t->right->right would change a BinaryNode data member, making findMax a mutator.

Insertion is like contains (requiring a search) but with a NULL link replaced by a new leaf unless x is a duplicate.
template <typename Comparable>
class BinarySearchTree
{
  public:
    BinarySearchTree( );
    BinarySearchTree( const BinarySearchTree & rhs );
    ~BinarySearchTree( );

    const Comparable & findMin( ) const;
    const Comparable & findMax( ) const;
    bool contains( const Comparable & x ) const;
    bool isEmpty( ) const;
    void printTree( ) const;

    void makeEmpty( );
    void insert( const Comparable & x );
    void remove( const Comparable & x );

    const BinarySearchTree & operator=( const BinarySearchTree & rhs );

  private:
    struct BinaryNode
    {
      Comparable element;
      BinaryNode * left;
      BinaryNode * right;
      
      BinaryNode( const Comparable & theElement, BinaryNode * lt, BinaryNode * rt )
      : element( theElement ), left( lt ), right( rt ) 
    };

    BinaryNode * root;

    void insert( const Comparable & x, BinaryNode * & t ) const;
    void remove( const Comparable & x, BinaryNode * & t ) const;
    BinaryNode * findMin( BinaryNode * t ) const;
    BinaryNode * findMax( BinaryNode * t ) const;
    bool contains( const Comparable & x, BinaryNode * t ) const;
    void makeEmpty( BinaryNode * & t );
    void printTree( BinaryNode * t ) const;
    BinaryNode * clone( BinaryNode * t ) const;
};
```cpp
/**
 * Returns true if x is found in the tree.
 */
bool contains( const Comparable & x ) const
{
    return contains( x, root );
}

/**
 * Insert x into the tree; duplicates are ignored.
 */
void insert( const Comparable & x )
{
    insert( x, root );
}

/**
 * Remove x from the tree. Nothing is done if x is not found.
 */
void remove( const Comparable & x )
{
    remove( x, root );
}
```

**Figure 4.17** Illustration of public member function calling private recursive member function
Figure 4.18 contains operation for binary search trees

```c
/**
 * Internal method to test if an item is in a subtree.
 * x is item to search for.
 * t is the node that roots the subtree.
 */

bool contains( const Comparable & x, BinaryNode * t ) const
{
    if( t == NULL )
        return false;
    else if( x < t->element )
        return contains( x, t->left );
    else if( t->element < x )
        return contains( x, t->right );
    else
        return true;    // Match
}
```
Figure 4.19 illustrates use of a function object to implement binary search tree.
Figures 4.20-21

1    /**
2    * Internal method to find the smallest item in a subtree t.
3    * Return node containing the smallest item.
4    */
5    BinaryNode * findMin( BinaryNode *t ) const
6    {
7        if( t == NULL )
8            return NULL;
9        if( t->left == NULL )
10           return t;
11        return findMin( t->left );
12    }

Figure 4.20 Recursive implementation of findMin for binary search trees

1    /**
2    * Internal method to find the largest item in a subtree t.
3    * Return node containing the largest item.
4    */
5    BinaryNode * findMax( BinaryNode *t ) const
6    {
7        if( t != NULL )
8            while( t->right != NULL )
9                t = t->right;
10           return t;
11    }

Figure 4.21 Nonrecursive implementation of findMax for binary search trees
**Figure 4.22** Binary search trees before and after inserting 5

```c
/**
 * Internal method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void insert( const Comparable & x, BinaryNode * & t )
{
    if( t == NULL )
        t = new BinaryNode( x, NULL, NULL );
    else if( x < t->element )
        insert( x, t->left );
    else if( t->element < x )
        insert( x, t->right );
    else
        ; // Duplicate; do nothing
}
```

**Figure 4.23** Insertion into a binary search tree
Deletion is the hardest operation. After the node $x$ to be removed is found there are three cases:

1. if $x$ is a leaf, remove it.

2. if $x$ has one child, replace its parent’s link by the link to the child.

3. if $x$ has two children,
   1. find the node $y$ with the smallest value in the right subtree,
   2. store that smallest value in node $x$, and
   3. remove $y$ from the right subtree — easy since it can’t have a left child.

Note the bias in favor of deeper left subtrees resulting from always deleting from the right subtree when $x$ has two children. We could find the largest element in the left subtree instead of the smallest element in the right subtree. Randomly choosing one or the other would apparently eliminate the bias, but is usually not necessary.
Figures 4.24-25

**Figure 4.24** Deletion of a node (4) with one child, before and after

![Diagram of Figure 4.24](image)

**Figure 4.25** Deletion of a node (2) with two children, before and after

![Diagram of Figure 4.25](image)
/**
 * Internal method to remove from a subtree.
 * x is the item to remove.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */

void remove( const Comparable & x, BinaryNode * & t )
{
    if( t == NULL )
        return; // Item not found; do nothing
    if( x < t->element )
        remove( x, t->left );
    else if( t->element < x )
        remove( x, t->right );
    else if( t->left != NULL && t->right != NULL ) // Two children
    {
        t->element = findMin( t->right )->element;
        remove( t->element, t->right );
    }
    else
    {
        BinaryNode *oldNode = t;
        t = ( t->left != NULL ) ? t->left : t->right;
        delete oldNode;
    }
}
Function remove is inefficient because it uses findMin to search the right subtree for the smallest node, and then recursively calls itself duplicating the search. Note that, when t points to a leaf, it is set to NULL, and the leaf is deleted.

If the number of deletions is expected to be small, we can use lazy deletion: merely mark elements as being deleted (or decrement the number of duplicates). If a deleted item is reinserted, the cost of allocating a new node is avoided. Even if the number of ‘deleted’ nodes is as large as the number of undeleted nodes, the expected increase in tree depth due to keeping the ‘deleted’ nodes is only a constant: $\log(2N) - \log(N) = \log(2N/N) = \log(2) = 1$ — a small time penalty.
/**
 * Destructor for the tree
 */

~BinarySearchTree( )
{
    makeEmpty( );
}

/**
 * Internal method to make subtree empty.
 */

void makeEmpty( BinaryNode * & t )
{
    if( t != NULL )
    {
        makeEmpty( t->left );
        makeEmpty( t->right );
        delete t;
    }
    t = NULL;
}
Figure 4.28

```c++
/**
 * Deep copy.
 */
const BinarySearchTree & operator=( const BinarySearchTree & rhs )
{
    if( this != &rhs )
    {
        makeEmpty( );
        root = clone( rhs.root );
    }
    return *this;
}

/**
 * Internal method to clone subtree.
 */
BinaryNode * clone( BinaryNode * t ) const
{
    if( t == NULL )
        return NULL;
    return new BinaryNode( t->element, clone( t->left ), clone( t->right ) );
}
```

Figure 4.28 operator= and recursive clone member function
The run times for makeEmpty, operator=, and printTree are $O(N)$. The run times for insert, remove, findMin, findMax, and contains are $O(d)$, where $d$ is the depth of the accessed item. We will show that, assuming all insertion sequences are equally likely, the expected depth is $O(\log N)$. Let $D(N)$ be the internal path length (sum of depths of all nodes) for $N$ nodes:

$$D(1) = 0, \quad D(N) = D(i) + D(N - 1 - i) + N - 1$$

for $i$ nodes in the left subtree, $0 \leq i \leq N - 1$. The last term comes from adding one depth level per node to get the depth relative to the root. In a binary search tree (but not all binary trees) the left and right subtree sizes depend only on the relative rank of the first element inserted; if all values are equally likely, then all subtree sizes are equally likely.
The average value of both $D(i)$ and $D(N - 1 - i)$ is thus

$$\frac{1}{N} \sum_{j=0}^{N-1} D(j) = \frac{1}{N} \sum_{j=0}^{N-1} D(N - 1 - j).$$

Hence

$$D(N) = \frac{2}{N} \sum_{j=0}^{N-1} D(j) + N - 1 \quad \Rightarrow$$

$$ND(N) = 2 \sum_{j=0}^{N-1} D(j) + N(N - 1) \quad \Rightarrow$$

$$(N - 1)D(N - 1) = 2 \sum_{j=0}^{N-2} D(j) + (N - 1)(N - 2)$$
Subtracting, we have

\[ ND(N) - (N - 1)D(N - 1) = 2D(N - 1) + 2N - 2 \Rightarrow \\
ND(N) = (N + 1)D(N - 1) + 2N - 2 \approx (N + 1)D(N - 1) + 2N. \]

Dividing by \( N(N + 1) \) gives

\[
\frac{D(N)}{N + 1} = \frac{D(N - 1)}{N} + \frac{2}{N + 1} \\
\frac{D(N - 1)}{N} = \frac{D(N - 2)}{N - 1} + \frac{2}{N} \\
\frac{D(N - 2)}{N - 1} = \frac{D(N - 3)}{N - 2} + \frac{2}{N - 1} \\
\ldots \\
\frac{D(2)}{3} = \frac{D(1)}{2} + \frac{2}{3}.
\]
Adding, we obtain

\[
\sum_{j=2}^{N} \frac{D(j)}{j+1} = \sum_{j=1}^{N-1} \frac{D(j)}{j+1} + 2 \sum_{j=3}^{N+1} \frac{1}{j} \Rightarrow
\]

\[
\frac{D(N)}{N+1} = \frac{D(1)}{2} + 2 \sum_{j=3}^{N+1} \frac{1}{j} = 2 \left( \sum_{j=1}^{N+1} \frac{1}{j} - \frac{3}{2} \right)
\]

\[
\approx 2(\ln(N+1) + \gamma - (3/2)) = O(\log N) \Rightarrow
\]

\[
D(N) = O(N \log N),
\]

where \( \gamma \approx .577 \) is Euler's constant. Hence the expected depth of a node is \( O(\log N) \).
The following figure depicts a randomly generated 500-node binary tree. The average depth is 9.98.

**Figure 4.29** A randomly generated binary search tree
Recall that our deletion algorithm destroys the property that all BST’s are equally likely. After $\Theta(N^2)$ alternating insertions and deletions the expected depth is $\Theta(\sqrt{N})$. The following figure depicts the result of $N^2 = 250,000$ random insert/remove pairs applied to the tree of Figure 4.29. The average depth is 12.51.

Figure 4.30 Binary search tree after $\Theta(N^2)$ insert/remove pairs
Problem: A series of insertions of pre-sorted data produces an unbalanced tree with no left children and requires $O(N^2)$ time.

We require a balance condition to ensure that the depth is $O(\log N)$. Requiring left and right subtrees to have the same height does not work as shown in Figure 4.31. Requiring that every node have left and right subtrees of the same height is too rigid. It is only satisfied by a perfect binary tree in which all leaves are at the same level.

An AVL tree (Adelson-Velskii and Landis) is a binary search tree with the following structure property: for every node, the heights of left and right subtrees can differ by at most 1, where the height of an empty tree is -1.

An example is displayed in Figure 4.32. The height of each node must be stored in the node structure.
Figures 4.31-32

Figure 4.31 A bad binary tree. Requiring balance at the root is not enough

Figure 4.32 Two binary search trees. Only the left tree is AVL.
The maximum height of an AVL tree can be shown to be roughly \(1.44 \log(N + 2) - 1.328 = O(\log N)\). In practice the height is usually only slightly larger than \(\log N\). This follows from the fact that the minimum number of nodes \(s(h)\) in an AVL tree of height \(h\) is defined by \(s(0) = 1\), \(s(1) = 2\), and \(s(h) = s(h - 1) + s(h - 2) + 1\): 1, 2, 4, 7, 12, 20, 33, 54, 88, 143, ... — similar to the Fibonacci sequence. An example appears in Figure 4.33.

Insertion of 6 into the AVL tree of Figure 4.32 violates the AVL balance condition at the node with value 8. More generally, nodes on the path from the root to the insertion point have their subtrees, and possibly their balance, altered. The path is followed in reverse order, updating heights. If a violation is found, it is corrected by a single or double rotation.
Figure 4.33 Smallest AVL tree of height 9
Suppose node $\alpha$ must be rebalanced. There are four cases: the new node was inserted into the

1. left subtree of the left child of $\alpha$
2. right subtree of the left child of $\alpha$
3. left subtree of the right child of $\alpha$
4. right subtree of the right child of $\alpha$

Cases 1 and 4 require a *single rotation*.

**Figure 4.34** Single rotation to fix case 1
The following properties characterize the trees depicted in Figure 4.34.

- $Y$ and $Z$ may be empty, but $X$ may not.
- There is a violation at $k_2$ because its left subtree is two levels deeper than its right subtree $Z$.
- Since $k_2$ satisfied the AVL condition before insertion into $X$, $Y$ cannot be at the same level as $X$.
- $Y$ cannot be at the same level as $Z$ because the first violation would have been encountered at $k_1$ if it were.
- The rotation moves $X$ up one level and $Z$ down one level, leaving $Y$ at the same level. This is more than is required. Only a few pointer changes are involved.
- The new height is the same as it was before the insertion, and therefore no further rotations are needed.

Example insertion sequence: 3, 2, 1, 4, 5, 6, 7
**Figures 4.35-4.36**

**Figure 4.35** AVL property destroyed by insertion of 6, then fixed by a single rotation

**Figure 4.36** Single rotation fixes case 4
Case 4 is the mirror image of case 1, and is fixed by a single rotation. Case 2, however, is not fixed by a single rotation, as shown in the following figure. Cases 2 and 3 require *double rotations* as depicted in Figures 4.38 and 4.39.

![Figure 4.37](image)

**Figure 4.37** Single rotation fails to fix case 2
Figures 4.38-4.39

Figure 4.38 Left-right double rotation to fix case 2

Figure 4.39 Right-left double rotation to fix case 3
Double Rotation continued

The following properties characterize the trees depicted in Figure 4.38.

- $A$ and $D$ may be empty, but not both $B$ and $C$.
- There is a violation at $k_3$.
- Either $B$ or $C$ is two levels deeper than $D$; $A$ is at the intermediate level.
- The double rotation restores the height to what it was before the insertion, and balancing is therefore complete.
- A double rotation is equivalent to two single rotations: $\alpha$’s child and grandchild followed by $\alpha$ and its new child.

Continued example insertion sequence: 16, 15, ..., 11, 10, 8, 9
A nonrecursive algorithm is significantly faster, but a recursive
algorithm is simpler, more elegant, and easier to code. Some of the
code for the latter is displayed in Figures 4.40, 4.41, 4.42, 4.44,
and 4.46. Note that it is not necessary to maintain links to parents.

```
struct AvlNode
{
    Comparable element;
    AvlNode *left;
    AvlNode *right;
    int height;

    AvlNode( const Comparable & theElement, AvlNode *lt,
             AvlNode *rt, int h = 0 )
        : element( theElement ), left( lt ), right( rt ), height( h )
};
```

**Figure 4.40** Node declaration for AVL trees

```
/**
 * Return the height of node t or -1 if NULL.
 */
int height( AvlNode *t ) const
{
    return t == NULL ? -1 : t->height;
}
```

**Figure 4.41** Function to compute height of an AVL node
```c
/**
 * Internal method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void insert( const Comparable & x, AvlNode * & t )
{
    if( t == NULL )
        t = new AvlNode( x, NULL, NULL );
    else if( x < t->element )
    {
        insert( x, t->left );
        if( height( t->left ) - height( t->right ) == 2 )
            if( x < t->left->element )
                rotateWithLeftChild( t );
            else
                doubleWithLeftChild( t );
    }
    else if( t->element < x )
    {
        insert( x, t->right );
        if( height( t->right ) - height( t->left ) == 2 )
            if( t->right->element < x )
                rotateWithRightChild( t );
            else
                doubleWithRightChild( t );
    }
    else
        ; // Duplicate; do nothing
    t->height = max( height( t->left ), height( t->right ) ) + 1;
}
```

**Figure 4.42** Insertion into an AVL tree
/**
 * Rotate binary tree node with left child.
 * For AVL trees, this is a single rotation for case 1.
 * Update heights, then set new root.
 */

void rotateWithLeftChild( AvlNode * & k2 )
{
    AvlNode *k1 = k2->left;
    k2->left = k1->right;
    k1->right = k2;
    k2->height = max( height( k2->left ), height( k2->right ) ) + 1;
    k1->height = max( height( k1->left ), k2->height ) + 1;
    k2 = k1;
}

**Figure 4.44** Routine to perform single rotation
/**
 * Double rotate binary tree node: first left child
 * with its right child; then node k3 with new left child.
 * For AVL trees, this is a double rotation for case 2.
 * Update heights, then set new root.
 */

void doubleWithLeftChild( AvlNode * & k3 )
{
    rotateWithRightChild( k3->left );
    rotateWithLeftChild( k3 );
}

Figure 4.46 Routine to perform double rotation
A splay tree is a binary search tree without the balance condition. Following each access to a node, the node is moved to the root by a series of AVL tree rotations. The basic idea is that a node that has been accessed is likely to be accessed again in the near future.

A simple strategy would be to rotate every node on the path with its parent. Consider the insertion sequence 1, 2, 3, ..., N. The total insertion cost is $O(N)$ which is the best we can hope for. But suppose we now access the elements in order. The cost is $\sum_{i=1}^{N-1} i = O(N^2)$, and the tree ends up where it started with only left children.
A *splaying* strategy uses double rotations in place of single rotations, except when the parent is the root. Let $X$ be a node on the path toward the root, and denote the parent and grandparent of $X$ by $P$ and $G$, respectively. There are two cases:

- **zig-zag** ($X = \text{right child of } P$, $P = \text{left child of } G$) or ($X = \text{left child of } P$, $P = \text{right child of } G$)
- **zig-zig** Both left children or both right children

The two cases are depicted in Figures 4.47 and 4.48. Note that zig-zag is the same double rotation used by an AVL tree, while zig-zig reverses the order of the single rotations.
Figures 4.47-4.48

**Figure 4.47** Zig-zag

**Figure 4.48** Zig-zig
Consider again the insertion sequence 1, 2, ..., N into an initially empty tree. We get all left children in $O(N)$ time as before, but then an access to 1 makes the depth of 2 about $N/2$ instead of $N - 1$, and access to 2 changes the tree depth to about $N/4$ instead of $N - 2$. We end up with depth roughly $\log N$. Figures 4.50 to 4.58 demonstrate this for $N = 32$.

When access paths are long, searches are expensive — $O(N)$, but the rotations reduce tree depth, making future accesses cheap. When access paths are short, searches are cheap — $O(1)$, but the rotations are not improving the balance.

**Theorem:** A sequence of $M$ consecutive tree operations, starting from an empty tree, take at most $O(M \log N)$ time, implying that the *amortized* time is $O(\log N)$. There are no bad sequences.
Figure 4.50 Result of splaying at node 1 a tree of all left children
Figures 4.51-4.52

Figure 4.51 Result of splaying the previous tree at node 2

Figure 4.52 Result of splaying the previous tree at node 3
Figure 4.53 Result of splaying the previous tree at node 4

Figure 4.54 Result of splaying the previous tree at node 5
Figures 4.55-4.56

Figure 4.55 Result of splaying the previous tree at node 6

Figure 4.56 Result of splaying the previous tree at node 7
Figures 4.57-4.58

Figure 4.57 Result of splaying the previous tree at node 8

Figure 4.58 Result of splaying the previous tree at node 9
Deletion: An access moves the node to be deleted to the root. Denote the subtrees by \( T_L \) and \( T_R \). Find the largest element in \( T_L \), and rotate it to the root of \( T_L \) which now has no right child. Then delete the root by making \( T_R \) the right child of \( T_L \).

Implementation of splay trees is simpler than AVL trees because there are fewer cases to consider, and no balance information (height) is needed. The code might be faster.
Recursive function `printTree` in Figure 4.59 uses *inorder traversal* to print a binary search tree in order. The run time is $O(N)$ since there is constant work per node. This demonstrates an $N \log N$ sort implicit in the BST.

Recursive function `height` in Figure 4.60 uses *postorder traversal*. Note that the height of a leaf is computed as 0. Since there is constant work per node, the time is $O(N)$.

A *preorder traversal* could be used to label each node with its depth.

In all three cases, the code should handle the NULL case first.

In a *level-order traversal*, all nodes at depth $d$ are processed before any node at depth $d + 1$; a queue is used instead of the implied stack used by recursion.
Figure 4.59 Routine to print a binary search tree in order

```c++
/**
 * Print the tree contents in sorted order.
 */
void printTree( ostream & out = cout ) const
{
    if( isEmpty( ) )
        out << "Empty tree" << endl;
    else
        printTree( root, out );
}

/**
 * Internal method to print a subtree rooted at t in sorted order.
 */
void printTree( BinaryNode *t, ostream & out ) const
{
    if( t != NULL )
    {
        printTree( t->left, out );
        out << t->element << endl;
        printTree( t->right, out );
    }
}
```

Figure 4.60 Routine to compute the height of a tree using a postorder traversal

```c
/**
 * Internal method to compute the height of a subtree rooted at t.
 */
int height(BinaryNode *t) {
    if (t == NULL)
        return -1;
    else
        return 1 + max(height(t->left), height(t->right));
}
```

Figure 4.60 Routine to compute the height of a tree using a postorder traversal
If the data structure is too large for memory, the big-O analysis no longer applies because disk access time dwarfs the cost of processor operations. A disk is an electromechanical device. The processor can execute millions of instructions for each disk access.

An $M$-ary search tree, $M \geq 2$, allows $M$-way branching. As $M$ increases, depth decreases for a well-balanced $N$-node tree, and the number of disk accesses decreases at the cost of more complex code to deal with the more complex data structure (but processor operations are essentially free). A complete binary tree has height $\sim \log_2 N$; a complete $M$-ary tree has height $\sim \log_M N$.

Instead of one key to specify a left or right branch in a binary search tree, we need up to $M - 1$ keys stored in non-leaf nodes; all data items are stored in leaves.
The following definition is popular and is referred to as a B+ tree, but may be altered in minor ways. A B-tree of order $M$ is an $M$-ary tree with the following properties.

1. The data items are stored in leaves.
2. Each nonleaf node stores up to $M - 1$ keys with key $i$ equal to the smallest key in subtree $i + 1$ for $i = 1, \ldots, M - 1$.
3. The root is either a leaf or has 2 to $M$ children (after enough insertions have occurred).
4. All nonleaf nodes except the root have $\lceil M/2 \rceil$ to $M$ children.
5. All leaves are at the same depth and have $\lceil L/2 \rceil$ to $L$ data items.
Example Application

The textbook example of Florida driving records is characterized by the following parameters.

- \( N = 10^7 \) data items (records)
- record length = 256 bytes
- key length = 32 bytes (name)
- node = disk block = 8192 bytes
- link = disk block number = 4 bytes
- nonleaf (internal) node size = \((M - 1)\) keys \times 32\) bytes/key + \(M\) links \times 4\) bytes/link = \(36M - 32\) \(\leq 8192\) for \(M = 228\).
- leaf node size = \(L\) records \times 256\) bytes/record = \(256L\) \(\leq 8192\) for \(L = 32\)

The maximum number of leaves is \(10^7\) records divided by the minimum number of records per leaf: \(10^7/16 = 625000\). With up to \(M^k\) nodes on level \(k\), the nodes are on level 3 because \(M^2 < 625000 < M^3\) for \(M = 228\). Levels 0 and 1 could be cached in memory so that disk access is needed only for levels 2 and 3.
An example with $M = L = 5$ is displayed in Figures 4.62 to 4.65. The insertion of 57 requires one disk write.

Figure 4.62 B-tree of order 5

Figure 4.63 B-tree after insertion of 57 into the tree in Figure 4.62
Figure 4.64 Insertion of 55 into the B-tree in figure 4.63 causes a split into two leaves.

The \( L + 1 \) items must be split into two leaves, each with at least \( \lceil L/2 \rceil \) items, requiring three disk writes (since the parent must be updated with the additional child whenever a split occurs. Splits are time-consuming but rare. With \( L = 32 \), for example, the two new leaves (of sizes 16 and 17) each have room for 15 or 16 more insertions.
Insertion of 40 into the B-tree in Figure 4.64 causes a split into two leaves and then a split of the parent node.

Insertion of 40 requires five disk writes.

If splitting goes all the way up to the root, the B-tree depth increases. This is the only way that can happen. We allow the root to have only two children in order to allow its split.

Another way to handle overflow of children is to put one up for adoption by a neighbor if there is room; e.g., insertion of 29 into Figure 4.65 by moving 32 to the next leaf. This requires only three disk writes but tends to keep nodes fuller.
**Deletion:** If the leaf’s number of items drops below $\lceil L/2 \rceil$, it must adopt a neighboring item if possible, or be combined with the neighbor otherwise. This may require that the parent adopt a neighbor. If the process propagates to the root, the tree height is reduced.

**Figure 4.66** B-tree after the deletion of 99 from the B-tree in Figure 4.65
STL containers `vector` and `list` are inefficient for searching. By using a balanced binary search tree, STL containers `set` and `map` guarantee logarithmic cost for insertion, deletion, and searching.

**Defn:** A `set` is an ordered container that does not allow duplicates.

The decimal digits, for example, constitute a set, while the first 10 digits of $\pi$ comprise a list.

The STL set contains nested types `iterator` and `const_iterator`, and methods `begin`, `end`, `size`, `empty`, `insert`, `erase`, and `find`. 
Recall that \texttt{insert} must return an iterator pointing to the new element, but since set insertion can fail (with a duplicate), it must also return a boolean. In the case of failure, the iterator represents the existing duplicate item. The STL therefore defines a class template \texttt{pair} that is little more than a struct with two members: \texttt{first} and \texttt{second}. There are two insert functions.

\begin{verbatim}
pair<iterator,bool> insert( const Object & x );
pair<iterator,bool> insert( iterator hint, const Object & x );
\end{verbatim}

The optional hint is used as a potential insertion point, and can reduce the operation count to $O(1)$; e.g.,

\begin{verbatim}
set<int> s;
for (int i = 0; i < 1000000; i++)
  s.insert(s.end(), i);
\end{verbatim}
There are three erase functions.

```cpp
int erase(const Object & x);
iterator erase(iterator itr);
iterator erase(iterator start, iterator end);
```

The first erases `x` if found and returns the number of items removed (0 or 1). The other two behave the same way as in `vector` and `list`: the second returns the iterator representing the item that followed `x` before the call, and `itr` becomes stale.

In place of `contains`, which returns a boolean, `set` provides `find` which returns an iterator representing the location of `x` if found, or the end marker otherwise.

```cpp
iterator find(const Object & x) const
```
By default, ordering is defined by the STL function object (Section 1.6.4) less<
Object>, which is implemented by invoking operator< for Object. An alternative ordering can be specified by instantiating the set template with a function object type; e.g.,

```cpp
set<string, CaseInsensitiveCompare> s;
s.insert( "Hello" );  s.insert( "HeLLo" );
cout << "The size is: " << s.size() << endl;
```

The printed value of s.size is 1 because the second string is a duplicate.
A map is an ordered collection of entries consisting of key/value pairs with unique keys. Values need not be unique; several keys can map to the same value. A map is a generalization of an array, allowing arbitrary keys in place of indices.

A map behaves like a set instantiated with a pair and a comparison function that refers only to the key. As with a set, an optional template parameter can be used to specify a comparison function different from less<KeyType>. Methods include begin, end, size, and empty, but the iterator is a key/value pair: *itr has type pair<KeyType,ValueType>. The map also supports insert, find, and erase. Function insert requires a pair<KeyType,ValueType> object. Function find requires only a key but returns an iterator that references a pair.
The syntax is simplified by an overloaded array index operator:

```
ValueType & operator[](const KeyType & key)
```

If `key` is present in the map, a reference to the value is returned. Otherwise, it is inserted with a default value, and a reference to that value is returned. The default value is obtained by applying a zero-parameter constructor, or 0 for a primitive type. There is no accessor version of `operator[]`, and it cannot therefore be used on a constant map.

In the following example code, names are mapped to salaries sorted by name.
Accessing values in a map

The code inserts "Pat" with value 0, then changes the value. In some applications it is necessary to distinguish between items in the map and items not in the map. Also, we may want the map to be constant (immutable), implying use of a const_iterator.
4.8.3 Implementation of set and map

A balanced binary search tree is necessary to get worst-case logarithmic time for insert, erase, and find. An AVL tree is less often used than a top-down red-black tree (Section 12.2).

**Problem:** How do we increment an iterator to advance to the next node?

The next node in the iteration is either the node in the current node’s right subtree that contains the minimum value or, if the right subtree is empty, the nearest ancestor that contains the current node in its left subtree. Following are storage-inefficient solutions.

1. Maintain (as iterator data) a stack containing the path to the current node. This is inelegant.
2. Have every node in the tree store its parent in addition to its children. This also makes the code clumsy.
3. Have each node maintain extra links: one to the next smaller node, and one to the next larger node.
A good solution, used by many STL implementations, is a *threaded tree* in which there are extra links, referred to as *threads*, only for nodes that have NULL left or right links. A boolean variable specifies whether a link is a standard tree link or a link to the inorder predecessor (left) or inorder successor (right) in the ordering.

To find the successor, it was only necessary to consider an ancestor when the right subtree is empty, in which case the successor is given by the right thread. Finding the predecessor is analogous. No upward links are necessary.

The $N + 1$ NULL pointers (half the space allocated to pointers) is wasted in a standard binary search tree.
Example Problem Find all words that can be changed into at least 15 other words by a single one-character substitution. We use a dictionary of \(\sim 89,000\) words, most of which have 6 to 11 characters. We map words to vectors of words in the equivalence class defined by one-character substitution.

The code in Figure 4.68 prints a solution. It uses constant references entry and words to simplify expressions and avoid unnecessary copies.

Figures 4.69 and 4.70 implement the solution with a quadratic-time algorithm. Note that the operator \([\cdot]\) stores string \(\text{words}[i]\) into the map \(\text{adjWords}\) with a vector of size 0 if it was not already there, and then adds \(\text{words}[j]\) to the vector, increasing the size to 1.

Figures 4.71 and 4.72 display solutions with improved efficiency.
void printHighChangeables( const map<string,vector<string> > & adjWords,
    int minWords = 15 )
{
    map<string,vector<string> >::const_iterator itr;

    for( itr = adjWords.begin( ); itr != adjWords.end( ); ++itr )
    {
        const pair<string,vector<string> > & entry = *itr;
        const vector<string> & words = entry.second;

        if( words.size( ) >= minWords )
        {
            cout << entry.first << " (" << words.size( ) << ")":";
            for( int i = 0; i < words.size( ); i++ )
                cout << " " << words[ i ];
            cout << endl;
        }
    }
}
Figure 4.69

// Returns true if word1 and word2 are the same length
// and differ in only one character.
bool oneCharOff( const string & word1, const string & word2 )
{
    if( word1.length() != word2.length() )
        return false;

    int diffs = 0;

    for( int i = 0; i < word1.length(); i++ )
        if( word1[ i ] != word2[ i ] )
            if( ++diffs > 1 )
                return false;

    return diffs == 1;
}

Figure 4.69 Routine to check if two words differ in only one character
Function to compute a map containing words as keys and a vector of words that differ in only one character as values. This version runs in 6.5 minutes on an 89,000-word dictionary.
Figure 4.71 Function to compute a map containing words as keys and a vector of words that differ in only one character as values. It splits words into groups by word length. This version runs in 77 seconds on an 89,000-word dictionary.

```cpp
1 // Computes a map in which the keys are words and values are vectors of words
2 // that differ in only one character from the corresponding key.
3 // Uses a quadratic algorithm, but speeds things up a little by
4 // maintaining an additional map that groups words by their length.
5 map<string,vector<string>> adjWords;
6 map<int,vector<string>> wordsByLength;

7 // Group the words by their length
8 for( int i = 0; i < words.size(); i++ )
9     wordsByLength[ words[i].length() ].push_back( words[i] );
10
11 // Work on each group separately
12 map<int,vector<string>>::const_iterator itr;
13 for( itr = wordsByLength.begin(); itr != wordsByLength.end(); ++itr )
14 {
15     const vector<string> & groupsWords = itr->second;
16
17     for( int i = 0; i < groupsWords.size(); i++ )
18         for( int j = i + 1; j < groupsWords.size(); j++ )
19             if( oneCharOff( groupsWords[i], groupsWords[j] ) )
20                 {
21                     adjWords[ groupsWords[i] ].push_back( groupsWords[j] );
22                     adjWords[ groupsWords[j] ].push_back( groupsWords[i] );
23                 }
24     return adjWords;
25 }
```
Figure 4.72 Function to compute a map containing words as keys and a vector of words that differ in only one character as values. This version runs in 5 seconds on an 89,000-word dictionary.

```cpp
1    // Computes a map in which the keys are words and values are vectors of words
2    // that differ in only one character from the corresponding key.
3    // Uses an efficient algorithm that is O(N log N) with a map.
4    map<string,vector<string>> computeAdjacentWords( const vector<string> & words )
5    {
6        map<string,vector<string>> adjWords;
7        map<int,vector<string>> wordsByLength;
8
9        // Group the words by their length
10       for( int i = 0; i < words.size(); i++ )
11          wordsByLength[ words[i].length() ].push_back( words[i] );
```
// Work on each group separately
map<int,vector<string>> ::const_iterator itr;
for (itr = wordsByLength.begin(); itr != wordsByLength.end(); ++itr )
{
    const vector<string> & groupsWords = itr->second;
    int groupNum = itr->first;

    // Work on each position in each group
    for (int i = 0; i < groupNum; i++)
    {
        // Remove a character in given position, computing representative.
        // Words with same representatives are adjacent; so populate a map
        map<string,vector<string>> repToWord;

        for (int j = 0; j < groupsWords.size(); j++)
        {
            string rep = groupsWords[j];
            rep.erase(1,1);
            repToWord[ rep ].push_back( groupsWords[j] );
        }

        // and then look for map values with more than one string
        map<string,vector<string>> ::const_iterator itr2;
        for (itr2 = repToWord.begin(); itr2 != repToWord.end(); ++itr2 )
        {
            const vector<string> & clique = itr2->second;
            if (clique.size() >= 2 )
            {
                for (int p = 0; p < clique.size(); p++)
                    for (int q = p + 1; q < clique.size(); q++)
                    {
                        adjWords[clique[p]].push_back( clique[q] );
                        adjWords[clique[q]].push_back( clique[p] );
                    }
            }
        }
    }
}
return adjWords;