```
//
// STREED.CPP - Suffix tree creation - debug version
//
// Mark Nelson, April, 1996
//
// This code has been tested with Borland C++ and
// Microsoft Visual C++.
//
// This program gets a line of input, either from the
// command line or from user input.  It then creates
// the suffix tree corresponding to the given text.
//
// This program is intended to be a supplement to the
// code found in STREE.CPP.  It contains a extensive
// debugging information, which might make it harder
// to read.
//
// This version of the program also gets around the
// problem of requiring the last character of the
// input text to be unique.  It does this by overloading
// operator[] for the input buffer object.  When you select
// T[ N ], you will get a value of 256, which is obviously
// going to be a unique member of the character string.
// This overloading adds some complexity, which just might
// make the program a little harder to read!
//
// In addition, there is some overloading trickery that lets
// you send T[i] to the output stream, and send the 256 value
// as the string "<EOF>".  Another convenience that adds
// code and complexity.
//

#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

//
// The maximum input string length this program
// will handle is defined here.  A suffix tree
// can have as many as 2N edges/nodes.  The edges
// are stored in a hash table, whose size is also
// defined here.  When I want to exercise the hash
// table a little bit, I set MAX_LENGTH to 6 and
// HASH_TABLE_SIZE to 13.
//

const int MAX_LENGTH = 1000;
const int HASH_TABLE_SIZE = 2179;  //A prime roughly 10% larger

//
// The Buffer class exists purely to overload operator[],
// which allows me to return 256 for T[ N ].  Note also
// that operator[] doesn't exactly return an int or a char
// like you might think.  Instead, it returns an Aux object,
// which is just really a wrapper around an integer.  This
// lets me write an operator<<() for Aux and ostream so that
// outputting 256 will actually print "<EOF>"
//

class Aux;

class Buffer {
```

```
    public :
        char data[ MAX_LENGTH ];
        int N;
        Aux operator[]( int size ) const;
};

//
// Since Aux is just a wrapper around an integer, all I
// need is a holder for the integer, a constructor,
// and a casting operator.
//

class Aux {
    public :
        int i;
        Aux( int rhs ){ i = rhs; }
        operator int(){ return i; }
};

//
// This is the insertion operator that I use to load up the
// data buffer from an input stream.  I also use the operator
// to set N to the correct value, which is one greater than
// the number of characters read in.  It is one greater because
// we are going to return the special EOF character from position
// T[N].  So for example, if you read in the string "ABC" from
// the input stream, N will be 3, and T[3] will return 256.
//

istream &operator>>( istream &s, Buffer &b )
{
    s >> b.data;
    assert( strlen( b.data ) < MAX_LENGTH );
    b.N = strlen( b.data );
    return s;
}

//
// When you look up a character from the buffer object,
// you actually get an Aux object.  This is okay, because
// Aux has a casting operator for int.  If you are expecting
// an int, the compiler will take care of converting for
// you automatically.  If you are sending the object to
// a stream, the Aux operator<<() will do that special
// conversion for the special value of 256.
//

inline Aux Buffer::operator[]( int i ) const
{
    if ( i >= N )
        return Aux( 256 );
    else
        return Aux( data[ i ] );
}

//
// Finally, the special version of the output
// stream insertion operator for objects of
// type Aux.  The unique value of 256 triggers
// some special output.
//
ostream &operator<<( ostream &s, Aux &a )
{
    if ( a.i == 256 )
        s << "<EOF>";
```

```
    else
        s << (char) a.i;
    return s;
}

//
// When a new tree is added to the table, we step
// through all the currently defined suffixes from
// the active point to the end point.  This structure
// defines a Suffix by its final character.
// In the canonical representation, we define that last
// character by starting at a node in the tree, and
// following a string of characters, represented by
// first_char_index and last_char_index.  The two indices
// point into the input string.  Note that if a suffix
// ends at a node, there are no additional characters
// needed to characterize its last character position.
// When this is the case, we say the node is Explicit,
// and set first_char_index > last_char_index to flag
// that.
//

class Suffix {
    public :
        int origin_node;
        int first_char_index;
        int last_char_index;
        Suffix( int node, int start, int stop )
            : origin_node( node ),
              first_char_index( start ),
              last_char_index( stop ){};
        int Explicit(){ return first_char_index > last_char_index; }
        int Implicit(){ return last_char_index >= first_char_index; }
        void Canonize();
};

//
// The suffix tree is made up of edges connecting nodes.
// Each edge represents a string of characters starting
// at first_char_index and ending at last_char_index.
// Edges can be inserted and removed from a hash table,
// based on the Hash() function defined here.  The hash
// table indicates an unused slot by setting the
// start_node value to -1.
//

class Edge {
    public :
        int first_char_index;
        int last_char_index;
        int end_node;
        int start_node;
        void Insert();
        void Remove();
        Edge();
        Edge( int init_first_char_index,
              int init_last_char_index,
              int parent_node );
        int SplitEdge( Suffix &s );
        static Edge Find( int node, int c );
        static int Hash( int node, int c );
};

//
//  The only information contained in a node is the
```

```
//  suffix link. Each suffix in the tree that ends
//  at a particular node can find the next smaller suffix
//  by following the suffix_node link to a new node.  Nodes
//  are stored in a simple array.
//
class Node {
    public :
        int suffix_node;
        Node() { suffix_node = -1; }
        static int Count;
};

//
// This is the hash table where all the currently
// defined edges are stored.  You can dump out
// all the currently defined edges by iterating
// through the table and finding edges whose start_node
// is not -1.
//

Edge Edges[ HASH_TABLE_SIZE ];

//
// This is the buffer that holds the input text, in the
// special class that supports the special overloading
//

Buffer T;

//
// The array of defined nodes.  The count is 1 at the
// start because the initial tree has the root node
// defined, with no children.
//

int Node::Count = 1; //We start with a single node 0 that has no children
Node Nodes[ MAX_LENGTH * 2 ];

//
// The default ctor for Edge just sets start_node
// to the invalid value.  This is done to guarantee
// that the hash table is initially filled with unused
// edges.
//

Edge::Edge()
{
    start_node = -1;
}

//
// I create new edges in the program while walking up
// the set of suffixes from the active point to the
// endpoint.  Each time I create a new edge, I also
// add a new node for its end point.  The node entry
// is already present in the Nodes[] array, and its
// suffix node is set to -1 by the default Node() ctor,
// so I don't have to do anything with it at this point.
//

Edge::Edge( int init_first, int init_last, int parent_node )
{
    first_char_index = init_first;
    last_char_index = init_last;
    start_node = parent_node;
```

```
    end_node = Node::Count++;
}

//
// Many of the debugging routines in the program
// use operator<<() to print out an edge.
//

ostream &operator<<( ostream &s, const Edge &edge )
{
    s << "Start, end nodes= "
      << edge.start_node
      << ", "
      << edge.end_node
      << " first, last = "
      << edge.first_char_index
      << ", "
      << edge.last_char_index
      << " \"";
    for ( int i = edge.first_char_index ; i <= edge.last_char_index ; i++ )
        s << T[ i ];
    s << "\"";
    return s;
}

//
// Edges are inserted into the hash table using this hashing
// function.
//

int Edge::Hash( int node, int c )
{
    return ( ( node << 8 ) + c ) % HASH_TABLE_SIZE;
}

//
// A given edge gets a copy of itself inserted into the table
// with this function.  It uses a linear probe technique, which
// means in the case of a collision, we just step forward through
// the table until we find the first unused slot.
//

void Edge::Insert()
{
    int i = Hash( start_node, T[ first_char_index ] );
    while ( Edges[ i ].start_node != -1 )
        i = ++i % HASH_TABLE_SIZE;
    Edges[ i ] = *this;
}

//
// Removing an edge from the hash table is a little more tricky.
// You have to worry about creating a gap in the table that will
// make it impossible to find other entries that have been inserted
// using a probe.  Working around this means that after setting
// an edge to be unused, we have to walk ahead in the table,
// filling in gaps until all the elements can be found.
//
// Knuth, Sorting and Searching, Algorithm R, p. 527
//

void Edge::Remove()
{
    int i = Hash( start_node, T[ first_char_index ] );
    while ( Edges[ i ].start_node != start_node ||
```

```
            Edges[ i ].first_char_index != first_char_index )
        i = ++i % HASH_TABLE_SIZE;
    for ( ; ; ) {
        Edges[ i ].start_node = -1;
        int j = i;
        for ( ; ; ) {
            i = ++i % HASH_TABLE_SIZE;
            if ( Edges[ i ].start_node == -1 )
                return;
            int r = Hash( Edges[ i ].start_node, T[ Edges[ i ].first_char_index ] );
            if ( i >= r && r > j )
                continue;
            if ( r > j && j > i )
                continue;
            if ( j > i && i >= r )
                continue;
            break;
        }
        Edges[ j ] = Edges[ i ];
    }
}

//
// The whole reason for storing edges in a hash table is that it
// makes this function fairly efficient.  When I want to find a
// particular edge leading out of a particular node, I call this
// function.  It locates the edge in the hash table, and returns
// a copy of it.  If the edge isn't found, the edge that is returned
// to the caller will have start_node set to -1, which is the value
// used in the hash table to flag an unused entry.
//

Edge Edge::Find( int node, int c )
{
    int i = Hash( node, c );
    for ( ; ; ) {
        if ( Edges[ i ].start_node == node )
            if ( c == T[ Edges[ i ].first_char_index ] )
                return Edges[ i ];
        if ( Edges[ i ].start_node == -1 )
            return Edges[ i ];
        i = ++i % HASH_TABLE_SIZE;
    }
}

//
// When a suffix ends on an implicit node, adding a new character
// means I have to split an existing edge.  This function is called
// to split an edge at the point defined by the Suffix argument.
// The existing edge loses its parent, as well as some of its leading
// characters.  The newly created edge descends from the original
// parent, and now has the existing edge as a child.
//
// Since the existing edge is getting a new parent and starting
// character, its hash table entry will no longer be valid.  That's
// why it gets removed at the start of the function.  After the parent
// and start char have been recalculated, it is re-inserted.
//
// The number of characters stolen from the original node and given
// to the new node is equal to the number of characters in the suffix
// argument, which is last - first + 1;
//

int Edge::SplitEdge( Suffix &s )
{
```

```
    cout << "Splitting edge: " << *this << "\n";
    Remove();
    Edge *new_edge =
      new Edge( first_char_index,
                first_char_index + s.last_char_index - s.first_char_index,
                s.origin_node );
    new_edge->Insert();
    Nodes[ new_edge->end_node ].suffix_node = s.origin_node;
    first_char_index += s.last_char_index - s.first_char_index + 1;
    start_node = new_edge->end_node;
    Insert();
    cout << "New edge: " << *new_edge << "\n";
    cout << "Old edge: " << *this << "\n";
    return new_edge->end_node;
}

//
// This routine prints out the contents of the suffix tree
// at the end of the program by walking through the
// hash table and printing out all used edges.  It
// would be really great if I had some code that will
// print out the tree in a graphical fashion, but I don't!
//

ostream &operator<<( ostream &s, const Suffix &str );

void dump_edges( Suffix s1 )
{
    cout << "Active prefix = " << s1 << "\n";
    cout << " Hash Start  End  Suf  first last  String\n";
    for ( int j = 0 ; j < HASH_TABLE_SIZE ; j++ ) {
        Edge *s = Edges + j;
        if ( s->start_node == -1 )
            continue;
        cout << setw( 4 ) << j << " "
             << setw( 5 ) << s->start_node << " "
             << setw( 5 ) << s->end_node << " "
             << setw( 3 ) << Nodes[ s->end_node ].suffix_node << " "
             << setw( 5 ) << s->first_char_index << " "
             << setw( 6 ) << s->last_char_index << "  ";
        for ( int l = s->first_char_index ; l <= ( s1.last_char_index < s->last_char_index
? s1.last_char_index : s->last_char_index ) ; l++ )
            cout << T[ l ];
        cout << "\n";
    }
    cout << "Hit any key to continue..." << flush;
    int c = getch();
    cout << "\n";
    if ( c == 0x1b || c == 3 || c == 0 )
        throw;
}

//
// A suffix in the tree is denoted by a Suffix structure
// that denotes its last character.  The canonical
// representation of a suffix for this algorithm requires
// that the origin_node by the closest node to the end
// of the tree.  To force this to be true, we have to
// slide down every edge in our current path until we
// reach the final node.

void Suffix::Canonize()
{
//
// There isn't any point in doing this if last_char_index < first_char_index,
```

```
// since that means that (first_char_index,last_char_index) refers to an
// empty string. An explicit state with an empty string is canonical by
// definition.
//
    if ( !Explicit() ) {
        Edge edge = Edge::Find( origin_node, T[ first_char_index ] );
        int edge_span = edge.last_char_index - edge.first_char_index;
        cout << "Canonizing";
        while ( edge_span <= ( last_char_index - first_char_index ) ) {
            first_char_index = first_char_index + edge_span + 1;
            origin_node = edge.end_node;
            cout << " " << *this;
            if ( first_char_index <= last_char_index ) {
               edge = Edge::Find( edge.end_node, T[ first_char_index ] );
               edge_span = edge.last_char_index - edge.first_char_index;
            }
        };
        cout << ".\n";
    }
}

//
// This debug routine prints out the value of a
// Suffix object.  In order to print out the
// entire suffix string, I have to walk up the
// tree to each of the parent nodes.  This is
// handled by the print_parents() routine, which
// does this recursively.
//

void print_parents( ostream &s, int node );

ostream &operator<<( ostream &s, const Suffix &str )
{
    s << "("
      << str.origin_node
      << ",("
      << str.first_char_index
      << ","
      << str.last_char_index
      << ") ";
    s << "\"";
        print_parents( s, str.origin_node );
        for ( int i = str.first_char_index ;
                  i <= str.last_char_index ;
                  i++ )
           s << T[ i ];
    s << "\"";
    s << ")";
    return s;
}

void print_parents( ostream &s, int node )
{
    if ( node != 0 )
        for ( int i = 0 ; i < HASH_TABLE_SIZE ; i++ ) {
            if ( Edges[ i ].end_node == node ) {
                print_parents( s, Edges[ i ].start_node );
                for ( int j = Edges[ i ].first_char_index ;
                          j <= Edges[ i ].last_char_index
                          ; j++ )
                    s << T[ j ];
                return;
            }
        }
```

```
}

//
// Adding a suffix line in AddPrefix() is really
// a simple operation.  All that needs to be done
// is to write out the correct value to the Nodes[]
// table in the correct place.  Since I've
// added some debug code here, it made sense to
// move it to a separate routine, even though it
// isn't being done that way in STREE.CPP
//
void AddSuffixLink( int &last_parent, int parent )
{
    if ( last_parent > 0 ) {
        cout << "Creating suffix link from node "
             << last_parent
             << " to node "
             << parent
             << ".\n";
        Nodes[ last_parent ].suffix_node = parent;
    }
    last_parent = parent;
}

//
// This routine constitutes the heart of the algorithm.
// It is called repetitively, once for each of the prefixes
// of the input string.  The prefix in question is denoted
// by the index of its last character.
//
// At each prefix, we start at the active point, and add
// a new edge denoting the new last character, until we
// reach a point where the new edge is not needed due to
// the presence of an existing edge starting with the new
// last character.  This point is the end point.
//
// Luckily for use, the end point just happens to be the
// active point for the next pass through the tree.  All
// we have to do is update it's last_char_index to indicate
// that it has grown by a single character, and then this
// routine can do all its work one more time.
//

void AddPrefix( Suffix &active, int last_char_index )
{
    int parent_node;
    int last_parent_node = -1;

    for ( ; ; ) {
        Edge edge;
        parent_node = active.origin_node;
        if ( active.Explicit() ) {
            edge = Edge::Find( active.origin_node, T[ last_char_index ] );
            if ( edge.start_node != -1 )
                break;
        } else { //implicit node, a little more complicated
            edge = Edge::Find( active.origin_node, T[ active.first_char_index ] );
            int span = active.last_char_index - active.first_char_index;
            if ( T[ edge.first_char_index + span + 1 ] == T[ last_char_index ] )
                break;
            parent_node = edge.SplitEdge( active );
        }
        Edge *new_edge = new Edge( last_char_index, T.N, parent_node );
        new_edge->Insert();
        cout << "Created edge to new leaf: " << *new_edge << "\n";
```

```
        AddSuffixLink( last_parent_node, parent_node );
        if ( active.origin_node == 0 ) {
            cout << "Can't follow suffix link, I'm at the root\n";
            active.first_char_index++;
        } else {
            cout << "Following suffix link from node "
                 << active.origin_node
                 << " to node "
                 << Nodes[ active.origin_node ].suffix_node
                 << ".\n";
            active.origin_node = Nodes[ active.origin_node ].suffix_node;
            cout << "New prefix : " << active << "\n";
        }
        active.Canonize();
    }
    AddSuffixLink( last_parent_node, parent_node );
    active.last_char_index++;  //Now the endpoint is the next active point
    active.Canonize();
};

void validate();

int main( int argc, char *argv[] )
{
    if ( argc > 1 ) {
        strcpy( T.data, argv[ 1 ] );
        assert( strlen( T.data ) < MAX_LENGTH );
        T.N = strlen( T.data );
    } else {
        cout << "Enter string: " << flush;
        cin >> T;
    }
    Suffix active( 0, 0, -1 );  // The initial active prefix
    for ( int i = 0 ; i <= T.N ; i++ ) {
        dump_edges( active );
        cout << "Step " << ( active.last_char_index + 1 )
             << " :  Adding ";
        for ( int j = 0 ; j <= i ; j++ )
            cout << T[ j ];
        cout << " to the tree\n";
        AddPrefix( active, i );
    }
    cout << "Done!\n";
//
// Once all N prefixes have been added, the resulting table
// of edges is printed out, and a validation step is
// optionally performed.
//
    dump_edges( active );
    validate();
    return 1;
};

//
// The validation code consists of two routines.  All it does
// is traverse the entire tree.  walk_tree() calls itself
// recursively, building suffix strings up as it goes.  When
// walk_tree() reaches a leaf node, it checks to see if the
// suffix derived from the tree matches the suffix starting
// at the same point in the input text.  If so, it tags that
// suffix as correct in the GoodSuffixes[] array.  When the tree
// has been traversed, every entry in the GoodSuffixes array should
// have a value of 1.
//
// In addition, the BranchCount[] array is updated while the tree is
```

```
// walked as well.  Every node count in the array has the
// number of child edges emanating from that node.  If the node
// is a leaf node, the value is set to -1.  When the routine
// finishes, every node should be a branch or a leaf.  The number
// of leaf nodes should match the number of suffixes (the length)
// of the input string.  The total number of branches from all
// nodes should match the node count.
//

char CurrentString[ MAX_LENGTH ];
char GoodSuffixes[ MAX_LENGTH ];
char BranchCounts[ MAX_LENGTH * 2 ] = { 0 };

int walk_tree( int start_node, int last_char_so_far );

void validate()
{
    for ( int i = 0 ; i < T.N ; i++ )
        GoodSuffixes[ i ] = 0;
    walk_tree( 0, 0 );
    int error = 0;
    for ( i = 0 ; i < T.N ; i++ )
        if ( GoodSuffixes[ i ] != 1 ) {
            cout << "Suffix " << i << " count wrong!\n";
            error++;
        }
    if ( error == 0 )
        cout << "All Suffixes present!\n";
    int leaf_count = 0;
    int branch_count = 0;
    for ( i = 0 ; i < Node::Count ; i++ ) {
        if ( BranchCounts[ i ] == 0 )
            cout << "Logic error on node "
                 << i
                 << ", not a leaf or internal node!\n";
        else if ( BranchCounts[ i ] == -1 )
            leaf_count++;
        else
            branch_count += BranchCounts[ i ];
    }
    cout << "Leaf count : "
         << leaf_count
         << ( leaf_count == (T.N+1) ? " OK" : " Error!" )
         << "\n";
    cout << "Branch count : "
         << branch_count
         << ( branch_count == (Node::Count - 1) ? " OK" : " Error!" )
         << endl;
}

int walk_tree( int start_node, int last_char_so_far )
{
    int edges = 0;
    for ( int i = 0 ; i <= 256 ; i++ ) {
        Edge edge = Edge::Find( start_node, i );
        if ( edge.start_node != -1 ) {
            if ( BranchCounts[ edge.start_node ] < 0 )
                cerr << "Logic error on node "
                     << edge.start_node
                     << '\n';
            BranchCounts[ edge.start_node ]++;
            edges++;
            int l = last_char_so_far;
            for ( int j = edge.first_char_index ; j <= edge.last_char_index ; j++ )
                CurrentString[ l++ ] = T[ j ]; //Conveniently, <EOS> casts down to '\0'
```

```
            if ( walk_tree( edge.end_node, l ) ) {
                if ( BranchCounts[ edge.end_node ] > 0 )
                        cerr << "Logic error on node "
                             << edge.end_node
                             << "\n";
                BranchCounts[ edge.end_node ]--;
            }
        }
    }
//
// If this node didn't have any child edges, it means we
// are at a leaf node, and can check on this suffix.  We
// check to see if it matches the input string, then tick
// off it's entry in the GoodSuffixes list.
//
    if ( edges == 0 ) {
        cout << "Suffix : ";
        for ( int m = 0 ; m < last_char_so_far ; m++ )
            cout << CurrentString[ m ];
        cout << "\n";
        GoodSuffixes[ strlen( CurrentString ) ]++;
        if ( strcmp( T.data + T.N - strlen( CurrentString ), CurrentString ) != 0 )
            cout << "Comparison failure!\n";
        return 1;
    } else
        return 0;
}
```