```
//
// CPTN278_A6_List_Bettle1.cpp
//
// Hal Bettle
// 21 February 2009
//
// List Class Body File.
//
// List implementation is dynamically linked list based.
// List uses an External Node.
// List has Error checking.
// List remains sorted.
// No duplicate entries allowed.
//
#include "CPTN278_A6_List_Bettle1.h"

List::List()
{
    // initialize the list
    begin = 0;
    end = 0;
    walking = 0;
    count = 0;
}

bool List::Is_Empty(void)
{
    if ( count == 0 )
    {
        return true;
    }
    else
    {
        return false;
    }
}

bool List::Is_Full(void)
{
    return false;
}

bool List::Insert(Node *item)
{
    Node *insert_point = 0;
    Node *element = new Node;
    element->set_number( item->get_number() );

    if ( ! List::Is_Full() ) // Check for full list
    {
        if ( List::Is_Empty() ) // if list empty unconditional Insert
        {
            element->set_flink( begin );
            element->set_blink( end );
            begin = element;
            end = element;
            count++;
            return true;
        }
        else // search for insertion point
        {
            insert_point = Search ( element );
            if ( insert_point == 0 ) // Insert at end of list
            {
                element->set_flink( 0 );
                element->set_blink( end );
                end->set_flink( element );
                end = element;
                count++;
                return true;
            }
            else if ( insert_point->get_number() != item->get_number() )
            {
                if ( insert_point == begin ) // Insert and begining of list
                {
                    element->set_flink( begin );
                    element->set_blink( 0 );
                    begin->set_blink( element );
                    begin = element;
                }
                else // Insert somewhere in the middle
                {
                    element->set_flink( insert_point );
                    element->set_blink( insert_point->get_blink() );
                    ( insert_point->get_blink() )->set_flink ( element );
                    insert_point->set_blink( element );
                }
                count++;
                return true;
            }
        }
    }
    delete element;
    return false;
}

bool List::Remove(Node *item)
{
    Node *remove_point = 0;

    if ( ! List::Is_Empty() )
    {
        if ( ( remove_point = Search ( item ) ) != 0 ) // if we find it
        {
            if ( begin == end ) // only one item on list
            {
                begin = 0;
                end = 0;
            }
            else if ( remove_point == begin ) //remove from begining
            {
                ( begin->get_flink() )->set_blink( 0 );
                begin = begin->get_flink();
            }
            else if ( remove_point->get_flink() == 0 ) // remove from end
            {
                ( end->get_blink() )->set_flink( 0 );
                end = end->get_blink();
            }
            else // remove from somewhere in the middle
            {
                ( remove_point->get_blink() )->set_flink( remove_point->get_flink() );
                ( remove_point->get_flink() )->set_blink( remove_point->get_blink() );
            }

            count--;
            delete remove_point;
            return true;
        }
    }
    return false;
}
```

```
void List::Display(void)
{
    walking = begin;
    for ( int i = 0; i < count; i++ )
    {
        cout << "Item = ";
        walking->display();
        cout << "." << endl;
        walking = walking->get_flink();
    }
    return;
}

Node *List::Search(Node *item)
{
    walking = begin;
    if ( ! List::Is_Empty() )
    {
        // walking != 0 check must be first else second test can fail.
        while ( ( walking != 0 ) && ( item->get_number() > walking->get_number() ) )
        {
            walking = walking->get_flink();
        }
    }
    return walking;
}

List::~List()
{
    while ( ! List::Is_Empty() )
    {
        List::Remove( begin );
    }
}
```