A COMPARISON OF LINKED LIST IMPLEMENTATIONS WITH AND WITHOUT A HEADER NODE

(See previously-distributed handout of full code for complete implementation of no header node version.)

```
/*
 * studentlisth.cc: Implementation of StudentList
 * This implementation uses a linked list with a header node
 *
 * Copyright (c) 2001. 2003, 2013 - Russell C. Bjork
 */

...
#define _first _header
#include "studentlist.h"
...
StudentList::StudentList()
: _header(new Node("", 0))
{ }

bool StudentList::isEmpty() const
{
    return _header -> _next == NULL;
}

void StudentList::makeEmpty()
{
    Node * p = _header -> _next, * d;
    ...
    _header -> _next = NULL;
}

void StudentList::insert(string name, int year)
{
    // Determine where it goes ...


    Node * p = _header -> _next, * q = _header;
    ...



    q -> _next = n;
}
```

```
/*
 * studentlist.cc: Implementation of StudentList
 * This implementation uses a simple linked list
 *
 * Copyright (c) 2001, 2003, 2013 - Russell C. Bjork
 */

...

#include "studentlist.h"
...
StudentList::StudentList()
: _first(NULL)
{ }

bool StudentList::isEmpty() const
{
    return _first == NULL;
}

void StudentList::makeEmpty()
{
    Node * p = _first, * d;
    ...
    _first = NULL;
}

void StudentList::insert(string name, int year)
{
    // Determine where it goes ... if q remains NULL,
    // the new node will be first on the list, so we
    // will set the external pointer to it.

    Node * p = _first, * q = NULL;
    ...
    if (q == NULL)
         _first = n;
    else
         q -> _next = n;
}
```

```
string StudentList::getFirst() const
{
    return _header -> _next -> _name;
}

void StudentList::removeFirst()
{
    Node * d = _header -> _next;
    _header -> _next = d -> _next;
    delete d;
}

bool StudentList::remove(string name)
{
    // Find the node to be deleted. ...
remains


    Node * p = _header -> _next, * q = _header;
    ...




    q -> _next = p -> _next;
    ...
}

void StudentList::print() const
{
    Node * p = _header -> _next;
    ...
}

StudentList::StudentList(const StudentList & rhs)
: _header(new Node("", 0))
{
    * this = rhs;
}
```

```
string StudentList::getFirst() const
{
    return _first -> _name;
}

void StudentList::removeFirst()
{
    Node * d = _first;
    _first = _first -> _next;
    delete d;
}

bool StudentList::remove(string name)
{
    // Find the node to be deleted. ...  If q
             // NULL the node was at the beginning of the
    // list, so we must reset the external pointer.

    Node * p = _first, * q = NULL;
    ...

    if (q == NULL)
         _first = p -> _next;
    else
         q -> _next = p -> _next;
    ...
}

void StudentList::print() const
{
    Node * p = _first;
    ...
}

StudentList::StudentList(const StudentList & rhs)
: _first(NULL)
{
    * this = rhs;
}
```

```
StudentList & StudentList::operator =
                  (const StudentList & rhs)
{
    ...
    // If rhs has any nodes, copy them




    Node * p = _header;
    Node * prhs = rhs._header -> _next;
    ...

    return * this;
}

StudentList::~StudentList()
{
    makeEmpty();
    delete _header;
}
```

```
StudentList & StudentList::operator =
                  (const StudentList & rhs)
{
    ...
    // If rhs has any nodes, copy them

    if (rhs._first != NULL)
    {
         _first = new Node(rhs._first -> _name,
                           rhs._first -> _year);
         Node * p = _first;
         Node * prhs = rhs._first -> _next;
         ...
    }
    return * this;
}

StudentList::~StudentList()
{
    makeEmpty();

}
```