A COMPARISON OF DOUBLY AND SINGLY LINKED LIST IMPLEMENTATIONS
(BOTH CIRCULAR WITH A HEADER NODE)
(See previously-distributed handouts of full code for no header version and changes for header and circular versions.)

```
/*
 * studentlistd.cc: Implementation of StudentList
 *
 * This implementation uses a doubly linked circular list
 * with a header node.
 * The circular structure assumes that all real entries will
 * have year > 0
 *
 * Copyright (c) 2001, 2003, 2013 - Russell C. Bjork
 */

...

class StudentList::Node
{
    public:

        Node(string name, int year)
        : _name(name), _year(year),
          _next(NULL), _prev(NULL)
        { }

        ...
        Node * _next, * _prev;
};

StudentList::StudentList()
: _header(new Node("", 0))
{
    _header -> _next = _header;
    _header -> _prev = _header;
}
```

```
/*
 * studentlisthc.cc: Implementation of StudentList
 *
* This implementation uses a circularly linked list with a
 * header node.
* The circular structure assumes that all real entries will
 * have year > 0
 *
 * Copyright (c) 2001, 2003, 2013 - Russell C. Bjork
 */

...

class StudentList::Node
{
    public:

        Node(string name, int year)
        : _name(name), _year(year), _next(NULL)

        { }

        ...
        Node * _next;
};

StudentList::StudentList()
: _header(new Node("", 0))
{
    _header -> _next = _header;

}
```

```
bool StudentList::isEmpty() const -- identical
```

```
void StudentList::makeEmpty()
{
    ...
    _header -> _next = _header;
    _header -> _prev = _header;
}

void StudentList::insert(string name, int year)
{
    ...
    // Determine where it goes. p will point to the node |
    // that belongs just AFTER the new node; we can
    // determine its predecessor from backward links

    Node * p = _header -> _next;
    while (p -> _year >= year)
         p = p -> _next;



    // Link it in     // Link it in

    n -> _next = p;
    n -> _prev = p -> _prev;
    p -> _prev -> _next = n;
    p -> _prev = n;
}
```

```
void StudentList::makeEmpty()
{
    ...
    _header -> _next = _header;

}

void StudentList::insert(string name, int year)
{
    ...
    // Determine where it goes. p will point to the node
    // that belongs just AFTER the new node; q to the
    // node that goes just BEFORE it.

    Node * p = _header -> _next, * q = _header;
    while (p -> _year >= year)
    {
         q = p;
         p = p -> _next;
    }



    n -> _next = p;
    q -> _next = n;


}
```

bool StudentList::getFirst() const -- identical

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

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

    delete d;
}
```

```
bool StudentList::remove(string name)
{
    Node * p = _header -> _next;

    // Find the node to be deleted. ... We can determine
    // its predecessor from backward links

    while (p != _header && p -> _name != name)
         p = p -> _next;




    ...
    p -> _prev -> _next = p -> _next;
    p -> _next -> _prev = p -> _prev;
    ...
}
```

```
bool StudentList::remove(string name)
{
    Node * p = _header -> _next, * q = _header;

    // Find the node to be deleted.  ... q will point
    // to the node BEFORE it

    while (p != _header && p -> _name != name)
    {
         q = p;
         p = p -> _next;
    }
    ...
    q -> _next = p -> _next;

    ...
}
```

bool StudentList::print() const -- identical

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

```
StudentList::StudentList(const StudentList & rhs)
: _header(new Node("", 0))
{
    _header -> _next = _header;

    * this = rhs;
}
```

```
StudentList & StudentList::operator =
                  (const StudentList & rhs)
{
    ...
    while(prhs != rhs._header)
    {
         p->_next=new Node(prhs->_name,prhs->_year);
         p -> _next -> _prev = p;
         p = p -> _next;
         prhs = prhs -> _next;
    }
    p -> _next = _header;
    _header -> _prev = p;
    return * this;    return * this;
}
```

```
StudentList & StudentList::operator =
                    (const StudentList & rhs)
{
    ...
    while(prhs != rhs._header)
    {
         p->_next=new Node(prhs->_name,prhs->_year);

         p = p -> _next;
         prhs = prhs -> _next;
    }
    p -> _next = _header;



}
```

StudentList::~StudentList() -- identical