

# CSCI-1200 Computer Science II — Fall 2006

## Lecture 19 — Linked Lists, Part II

### Announcements: Test 3 Information

- Test 3 will be held **Tuesday, November 14th, 2006 from 2-3:50pm in West Hall Auditorium**. No make-ups will be given except for emergency situations, and even then a written excuse from the Dean of Students office will be required.
- Coverage: Lectures 1-18, Labs 1-11, HW 1-8.
- Closed-book and closed-notes *except for 1 sheet of notes on 8.5x11 inch paper (front & back) that may be handwritten or printed*. Computers, cell-phones, palm pilots, calculators, PDAs, music players, etc. are not permitted and must be turned off.
- All students must bring their Rensselaer photo ID card.
- Practice problems are available on the course website. Solutions will be posted on Sunday.
- The TAs will hold special “Test Prep Office Hours” on Monday, November 13th from 5-8pm (location: TBA). Bring your questions!

### Review from Lecture 18

- Introductory example on linked lists,
- Basic linked list operations: stepping through a list, push back, insert, remove
- Common mistakes

### Today’s Lecture

- Limitations of singly-linked lists
- Doubly-linked lists:
  - Structure
  - Insert
  - Remove
- Our own version of the `list<T>` class
- `list<T>::iterator`

#### 19.1 Limitations of Singly-Linked Lists

- We can only move through it in one direction
- We need a pointer to the node **before** the node that needs to be deleted.
- Appending a value at the end requires that we step through the entire list to reach the end.

#### 19.2 Generalizations of Singly-Linked Lists

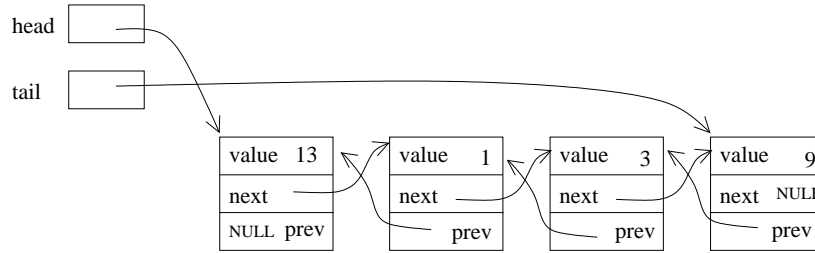
- Three common generalizations:
  - Doubly-linked: allows forward and backward movement through the nodes
  - Circularly linked: simplifies access to the tail, when doubly-linked
  - Dummy header node: simplifies special-case checks
- Today we will explore and implement a doubly-linked structure.

### 19.3 The Structure of Doubly-Linked Lists

- For the next few examples, we will use the simple node class:

```
class Node {  
public:  
    int value;  
    Node* next;  
    Node* prev;  
};
```

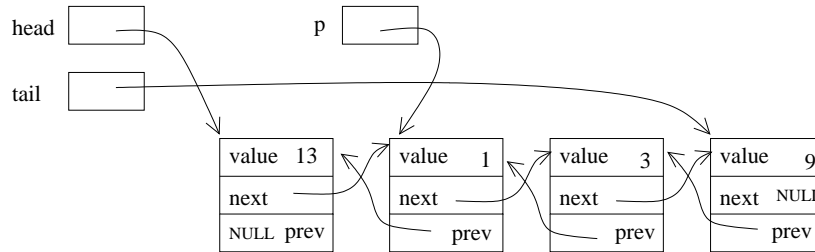
- Here is a picture of a doubly-linked list holding 4 integer values:



- Note that we now assume that we have both a **head** pointer, as before and a **tail** pointer variable, which stores the address of the last node in the linked list.
- The tail pointer is not strictly necessary, but it allows immediate access to the end of the list for efficient push-back operations.

### 19.4 Inserting in the Middle of a Doubly-Linked List

- Suppose we want to insert a new node containing the value 15 following the node containing the value 1. We have a temporary pointer variable, *p*, that stores the address of the node containing the value 1. Here's a picture of the state of affairs:



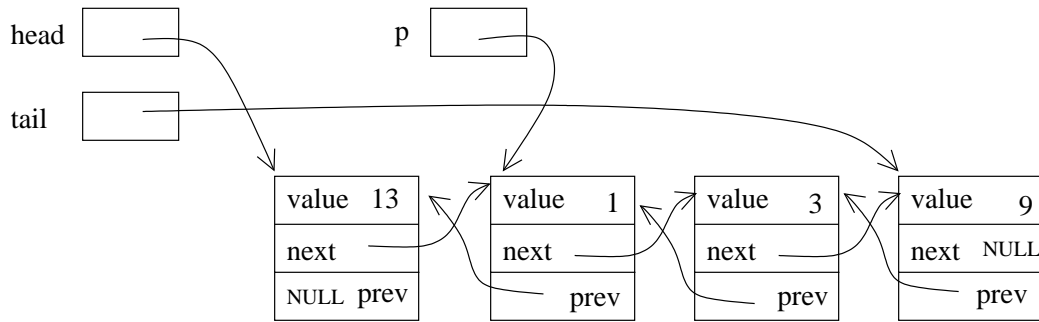
- What must happen?
  - The new node must be created, using another temporary pointer variable to hold its address.
  - Its two pointers must be assigned.
  - Two pointers in the current linked list must be adjusted. Which ones?

Assigning the pointers for the new node **MUST** occur before changing the pointers for the current linked list nodes!

- At this point, we are ignoring the possibility that the linked list is empty or that *p* points to the tail node (*p* pointing to the head node doesn't cause any problems).
- **Exercise:** write the code as just described.

## 19.5 Removing from the Middle of a Doubly-Linked List

- Suppose now instead of inserting a value we want to remove the node pointed to by `p` (the node whose address is stored in the pointer variable `p`)



- Two pointers need to change before the node is deleted! All of them can be accessed through the pointer variable `p`.
- **Exercise:** write this code.

## 19.6 Special Cases of Remove

- If `p==head` and `p==tail`, the single node in the list must be removed and both the `head` and `tail` pointer variables must be assigned the value `NULL`.
- If `p==head` or `p==tail`, then the pointer adjustment code we just wrote needs to be specialized to removing the first or last node.
- All of these will be built into the `erase` function that we write as part of our `cs2list` class.

## 19.7 The `cs2list` Class — Overview

- We will write a templated class called `cs2list` that implements much of the functionality of the `std::list<T>` container and uses a doubly-linked list as its internal, low-level data structure.
- Three classes are involved:
  - The node class
  - The iterator class
  - The `cs2list` class itself

## 19.8 The Node Class

- It is ok to make all members public because individual nodes are never seen outside the list class.
- Note that the constructors all initialize the pointers to `NULL`.

```
template <class T>
class Node {
public:
    Node( ) : next_(NULL), prev_(NULL) {}
    Node( const T& v ) : value_(v), next_(NULL), prev_(NULL) {}
    T value_;
    Node<T>* next_;
    Node<T>* prev_;
};
```

## 19.9 The Iterator Class — Desired Functionality

- Increment and decrement operators (will be operations on pointers).
- Dereferencing to access contents of a node in a list.
- Two comparison operations: `operator==` and `operator!=`.

## 19.10 The Iterator Class — Implementation

- (See attached code)
- Separate class
- Stores a pointer to a node in a linked list
- Constructors initialize the pointer — they will be called from the `cs2list<T>` class member functions.
  - `cs2list<T>` is a friend class to allow access to the pointer for `cs2list<T>` member functions such as `erase` and `insert`.
- `operator*` dereferences the pointer and gives access to the contents of a node.
- Stepping through the chain of the linked-list is implemented by the increment and decrement operators.
- `operator==` and `operator!=` are defined, but no other comparison operators are allowed.

## 19.11 The `cs2list` Class — Overview

- Manages the actions of the iterator and node classes
- Maintains the head and tail pointers and the size of the list
- Manages the overall structure of the class through member functions
- Three member variables: `head_`, `tail_`, `size_`
- Typedef for the `iterator` name
- Prototypes for member functions, which are equivalent to the `std::list<T>` member functions
- Some things are missing, most notably `const_iterator` and `reverse_iterator`.

## 19.12 The `cs2list` class — Implementation Details

- Many short functions are in-lined
- Clearly, it must contain the “big 3”: copy constructor, `operator=`, and destructor. The details of these are realized through the private `copy_list` and `destroy_list` member functions.

## 19.13 Exercises

1. Write `cs2list<T>::push_front`
2. Write `cs2list<T>::erase`

## cs2list.h

```

#ifdef cs2list_h
#define cs2list_h
// A simplified implementation of a generic list container class,
// including the iterator, but not the const iterators. Three
// separate classes are defined: a Node class, an iterator class, and
// the actual list class. The underlying list is doubly-linked, but
// there is no dummy head node and the list is not circular.
// -----
// NODE CLASS
template <class T>
class Node {
public:
    Node(): next_(NULL), prev_(NULL) {}
    Node(const T& v) : value_(v), next_(NULL), prev_(NULL) {}
// REPRESENTATION
    T value_;
    Node<T>* next_;
    Node<T>* prev_;
};
// A "forward declaration" of this class is needed
template <class T> class cs2list;
// -----
// LIST ITERATOR
template <class T>
class list_iterator {
public:
    list_iterator() : ptr_(NULL) {}
    list_iterator(Node<T>* p) : ptr_(p) {}
    list_iterator(list_iterator<T> const& old) : ptr_(old.ptr_) {}
    ~list_iterator() {}

    list_iterator<T> & operator=(const list_iterator<T> & old) {
        ptr_ = old.ptr_; return *this; }
// dereferencing operator gives access to the value at the pointer
    T& operator*() { return ptr_->value_; }

// increment & decrement operators
    list_iterator<T> & operator++() {
        ptr_ = ptr_->next_;
        return *this;
    }
    list_iterator<T> operator++(int) {
        list_iterator<T> temp(*this);
        ptr_ = ptr_->next_;
        return temp;
    }
    list_iterator<T> & operator--() {
        ptr_ = ptr_->prev_;
        return *this;
    }
    list_iterator<T> operator--(int) {
        list_iterator<T> temp(*this);
        ptr_ = ptr_->prev_;
        return temp;
    }
};

friend class cs2list<T>;
// Comparisons operators are straightforward
friend bool operator==(const list_iterator<T>& l, const list_iterator<T>& r) {
    return l.ptr_ == r.ptr_; }
friend bool operator!=(const list_iterator<T>& l, const list_iterator<T>& r) {
    return l.ptr_ != r.ptr_; }

private:
// REPRESENTATION
    Node<T>* ptr_; // ptr to node in the list
};
// -----
// LIST CLASS DECLARATION
// Note that it explicitly maintains the size of the list.
template <class T>
class cs2list {
public:
    cs2list() : head_(NULL), tail_(NULL), size_(0) {}
    cs2list(const cs2list<T>& old) { this->copy_list(old); }
    ~cs2list() { this->destroy_list(); }
    cs2list& operator=(const cs2list<T>& old);

    int size() const { return size_; }
    bool empty() const { return head_ == NULL; }
    void clear() { this->destroy_list(); }

    void push_front(const T& v);
    void pop_front();
    void push_back(const T& v);
    void pop_back();

    const T& front() const { return head_->value_; }
    T& front() { return head_->value_; }
    const T& back() const { return tail_->value_; }
    T& back() { return tail_->value_; }

    typedef list_iterator<T> iterator;
    iterator erase(iterator itr);
    iterator insert(iterator itr, T const& v);
    iterator begin() { return iterator(head_); }
    iterator end() { return iterator(NULL); }

private:
    void copy_list(cs2list<T> const & old);
    void destroy_list();
// REPRESENTATION
    Node<T>* head_;
    Node<T>* tail_;
    int size_;
};

```

## cs2list.h

```
// -----  
// LIST CLASS IMPLEMENTATION  
template <class T>  
cs2list<T>& cs2list<T>::operator= (const cs2list<T>& old) {  
    if (&old != this) {  
        this->destroy_list();  
        this->copy_list(old);  
    }  
    return *this;  
}  
  
template <class T>  
void cs2list<T>::push_back(const T& v) {  
  
}  
  
template <class T>  
void cs2list<T>::push_front(const T& v) {  
  
}  
  
template <class T>  
void cs2list<T>::pop_back() {  
  
}  
  
template <class T>  
void cs2list<T>::pop_front() {  
  
}  
  
template <class T>  
typename cs2list<T>::iterator cs2list<T>::erase(iterator itr) {  
  
}  
  
template <class T>  
typename cs2list<T>::iterator cs2list<T>::insert(iterator itr, T const& v) {  
  
}  
  
template <class T>  
void cs2list<T>::copy_list(cs2list<T> const & old) {  
  
}  
  
template <class T>  
void cs2list<T>::destroy_list() {  
  
}  
  
}  
  
#endif
```