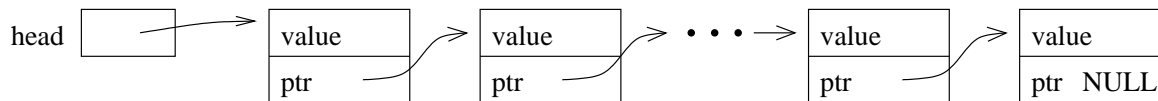


CSCI-1200 Computer Science II — Fall 2008

Lectures 10 & 11 — Linked Lists, Part I & II

Review from Lecture 9

- Returning references to member variables from member functions
- Review of iterators and iterator operations
- STL Lists, `erase` and `insert` on lists
- Differences between indices and iterators, differences between lists and vectors
- Introductory linked lists & operations on linked lists: Stepping through a list & Push back



Today's (Double) Lecture

- Basic linked list operations, continued: Insert & Remove
- Common mistakes
- Limitations of singly-linked lists
- Doubly-linked lists:
 - Structure
 - Insert
 - Remove
- Our own version of the STL `list<T>` class, named `cs2list`
- Implementing `list<T>::iterator`

11.1 Basic Mechanisms: Inserting a Node

- There are two parts to this: finding the location where the insert must take place, and doing the insert operation.
- We will ignore the find for now. We will also write only a code segment to understand the mechanism rather than writing a complete function.
- The insert operation itself requires that we have a pointer to the location **before** the insert location.
- If `p` is a pointer to this node, and `x` holds the value to be inserted, then the following code will do the insertion. Draw a picture to illustrate what is happening.

```
Node<T> * q = new Node<T>; // create a new node
q -> value = x;           // store x in this node
q -> next = p -> next;    // make its successor be the current successor of p
p -> next = q;           // make p's successor be this new node
```

- Note: This code will not work if you want to insert `x` in a new node at the *front* of the linked list. Why not?

11.2 Basic Mechanisms: Removing a Node

- There are two parts to this: finding the node to be removed and doing the remove operation.
- The remove operation itself requires a pointer to the node **before** the node to be removed.
- Removing the first node is an important special case.

11.3 Exercise: Remove a Node

Suppose `p` points to a node that should be removed from a linked list, `q` points to the node before `p`, and `head` points to the first node in the linked list. Write code to remove `p`, making sure that if `p` points to the first node that `head` points to what was the second node and now is the first after `p` is removed.

11.4 Exercise: List Copy

Write a *recursive* function to copy all nodes in a linked list to form a new linked list of nodes with identical structure and values. Here's the function prototype:

```
template <class T> void CopyAll(Node<T>* old_head, Node<T>*& new_head) {
```

11.5 Basic Linked Lists Mechanisms: Common Mistakes

Here is a summary of common mistakes. Read these carefully, and read them again when you have a problem that you need to solve.

- Allocating a new node to step through the linked list; only a pointer variable is needed.
- Confusing the `.` and the `->` operators.
- Not setting the pointer from the last node to `NULL`.
- Not considering special cases of inserting / removing at the beginning or the end of the linked list.
- Applying the `delete` operator to a node (calling the operator on a pointer to the node) before it is removed. Delete should be done after all pointer manipulations are completed.
- Pointer manipulations that are out of order. These can ruin the structure of the linked list.

11.6 Implementing Our Own List Class

- We will alter the structure of our linked list. Nodes will be templated and have two pointers, one going “forward” to the successor in the linked list and one going “backward” to the predecessor in the linked list. We will have a pointer to the beginning *and* the end of the list.

```
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_;
};
```

- We’ll reimplement the mechanisms discussed today and we will define list iterators as a class inside a class.

11.7 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.

11.8 Generalizations of Singly-Linked Lists

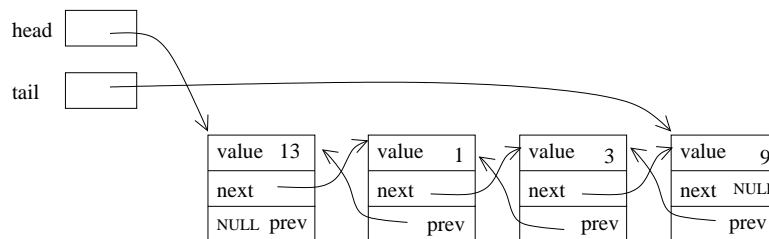
- Three common generalizations:
 - Doubly-linked: allows forward and backward movement through the nodes
 - Circularly linked: simplifies access to the tail, when doubly-linked
 - Dummy header node: simplifies special-case checks
- Today we will explore and implement a doubly-linked structure.

11.9 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;
};
```

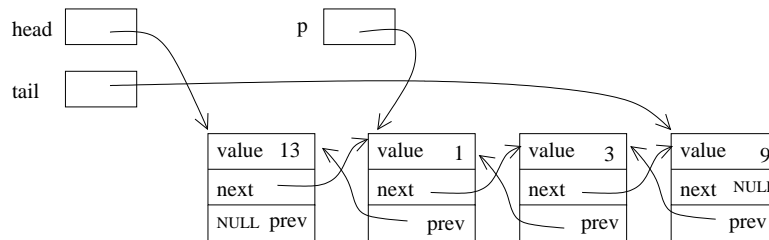
- Here is a picture of a doubly-linked list holding 4 integer values:



- 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.

11.10 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.

11.11 Removing from the Middle of a Doubly-Linked List

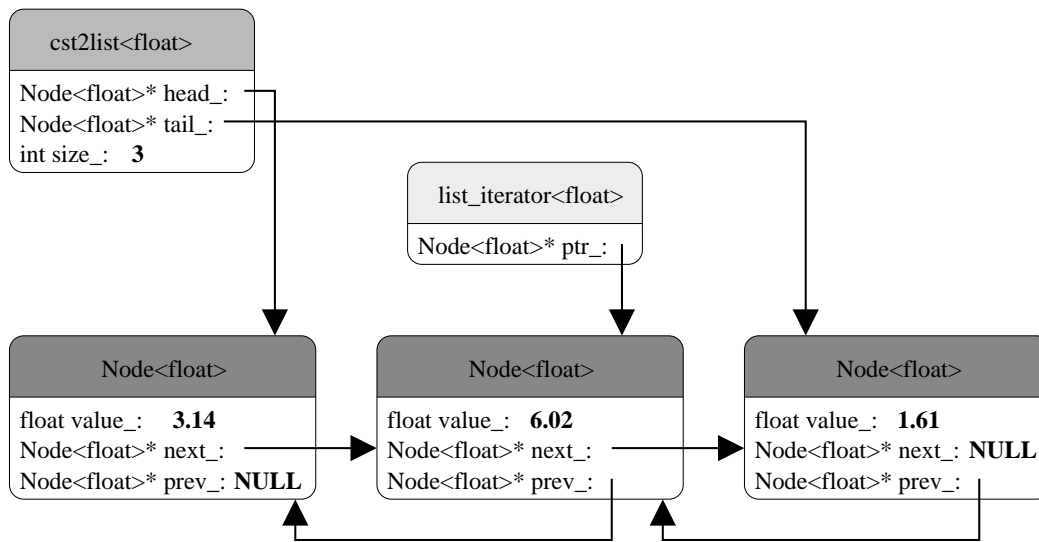
- Suppose now instead of inserting a value we want to remove the node pointed to by `p` (the node whose address is stored in the pointer variable `p`)
- Two pointers need to change before the node is deleted! All of them can be accessed through the pointer variable `p`.
- **Exercise:** write this code.

11.12 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.

11.13 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
- Below is a basic diagram showing how these three classes are related to each other:



- For each list object created by a program, we have one instance of the `cs2list` class, and multiple instances of the `Node`. For each iterator variable (of type `cs2list<T>::iterator`) that is used in the program, we create an instance of the `list_iterator` class.

11.14 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_;
};
  
```

11.15 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!=`.

11.16 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.

11.17 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`.

11.18 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.

11.19 Exercises

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

```

#ifndef 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) {
        ptr_ = ptr_->prev_;
        return *this;
    }
    list_iterator<T> & operator--() {
        ptr_ = ptr_->prev_;
        return *this;
    }
    list_iterator<T> & operator--(int) {
        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>
bool operator==(const cs2list<T>& lft, const cs2list<T>& rgt) {
    if (lft.size() != rgt.size()) return false;
    typename cs2list<T>::iterator lft_itr = lft.begin();
    typename cs2list<T>::iterator rgt_itr = rgt.begin();
    while (lft_itr != lft.end()) {
        if (*lft_itr != *rgt_itr) return false;
        lft_itr++;
        rgt_itr++;
    }
    return true;
}

template <class T>
bool operator!=(cs2list<T>& lft, cs2list<T>& rgt) { return !(lft == rgt); }

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

```