

Computer Science II — CSci 1200

Lecture 11

Linked Lists, Part 2

Review from Lecture 10

- 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`

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.

Generalizations of Singly-Linked Lists

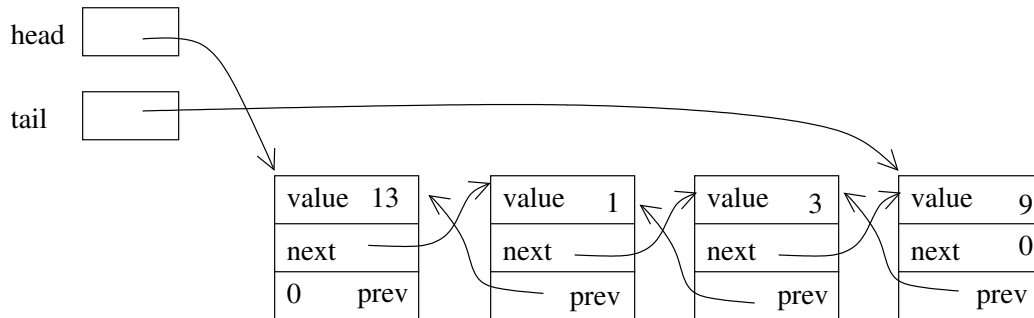
- Three common generalizations:
 - Doubly-linked: allows forward and backward movement through the nodes
 - Circularly linked: simplifies access to the tail, when doubly-linked
 - Dummy header node: simplifies special-case checks
- We will only consider doubly-linked, here

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;  
};
```

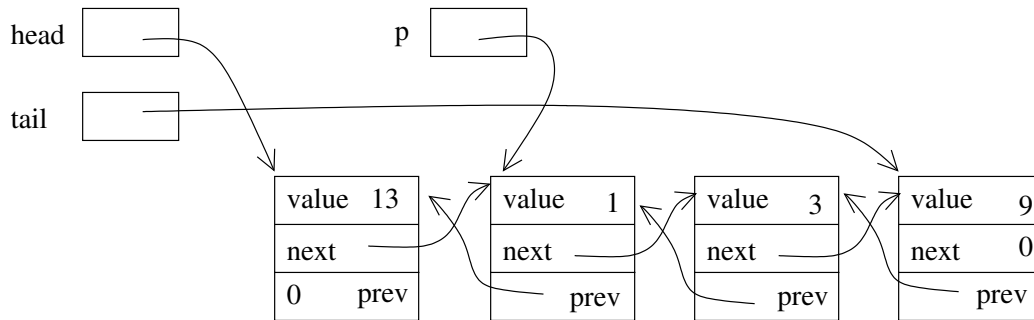
- Here is a picture of a doubly-linked list holding 4 integer values:



- Note that we now assume 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.
 - This is not strictly necessary, but it allows immediate access to the end of the list for push-back operations.

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.
- Suppose also that 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:



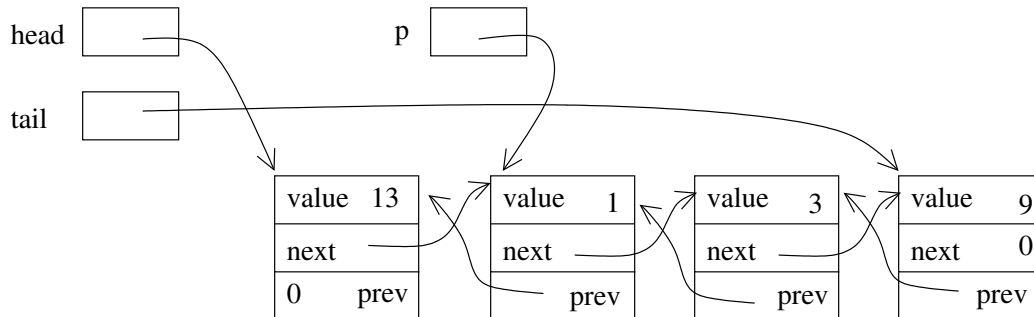
- What must happen?
 - The new node must be created, using another temporary pointer variable to hold its address.
 - Its two pointers must be assigned.
 - Two pointers in the current linked list must be adjusted. Which ones?

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.

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.

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 0.
- 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.

The `cs2list` Class — Overview

- We will write a templated class called `cs2list` that implements much of the functionality of the `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

The Node Class

- Here's the definition:

```
template <class T>
class Node {
public:
```

```

Node( ) : next_(0), prev_(0) {}
Node( const T& v ) : value_(v), next_(0), prev_(0) {}
T value_;
Node<T>* next_;
Node<T>* prev_;
};

```

- 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 0 (null).

The Iterator Class — Desired Functionality

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

The Iterator Class — Implementation

See code attached to the handout:

- 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.
- This requires that `cs2list<T>` be a **friend class** of `list_iterator<T>`.
 - When a class A states that another class B is a **friend**, this means that when any member function in B is working with an object of type A, the B member function has access to A's private member variables and functions.
 - In our case, this means `cs2list<T>` member functions have direct access to the private member variables of `list_iterator<T>` objects — the `Node<T>` pointers.
 - This is especially important for the `erase` and `insert` functions.
 - Note that friendship is granted, not claimed!

Now back to the details of `list_iterator<T>`...

- `operator*` dereferences the pointer and gives access to the contents of a node.
- The mechanism of stepping through the chain of the linked list is implemented by the increment and decrement operators.
- `operator==` and `operator!=` are defined and quite important. No other comparison operators are allowed.

The `cs2list` class — Overview and Prototype

- Job:
 - 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.
 - This means that the name `list_iterator<T>` is not used outside the function.
- 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`.

The `cs2list` class — Some Member Function 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. You will write one of these during lab.

Exercises

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