

CSCI-1200 Computer Science II — Fall 2006

Lecture 18 — Linked Lists, Part I

Review from Lecture 17

- Building our own version of `std::vector`.
Why do this? So that we can customize or build our own container classes!
- Templated (generic) classes
- Classes w/ dynamically-allocated memory: To avoid bugs & memory leaks, *classes that allocate dynamic memory* must provide **the BIG 3**:
 - Copy constructor
 - Assignment operator
 - Destructor
- Dynamic resizing

Today's Class: Linked Lists, Part I

- Motivation: implementation of the `std::list` container class.
- Introductory example on linked lists.
- Basic linked list operations:
 - Stepping through a list
 - Push back
 - Insert
 - Remove
- Common mistakes

18.1 Motivation

- Thus far our discussion of how `list<T>` is implemented has been only intuitive: it is a “chain” of objects.
- Now we will look at the mechanism — *linked lists*.
- Learning this mechanism is good background for higher-level courses where the design of novel data structures is important.

18.2 Objects with Pointers / Linking Objects

- The two fundamental mechanisms of linked lists are:
 - creating objects with pointers as one of the member variables, and
 - making these pointers point to other objects of the same type.
- These mechanisms are illustrated in the following program:

```
#include <iostream>
using namespace std;

template <class T>
class Node {
public:
    T value;
    Node* ptr;
};
```

```

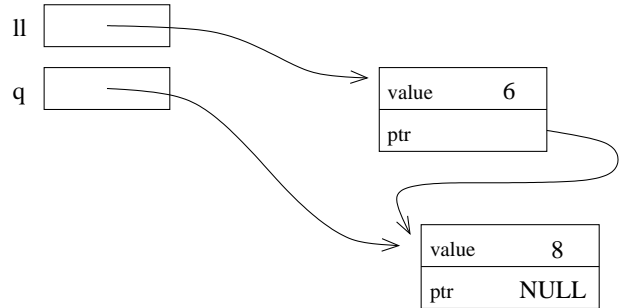
void main() {
    Node<int>* l1;      // l1 is a pointer to a (non-existent) Node
    l1 = new Node<int>; // Create a Node and assign its memory address to l1
    l1->value = 6;     // This is the same as (*l1).value = 6;
    l1->ptr = NULL;    // NULL == 0, which indicates a "null" pointer

    Node<int>* q = new Node<int>;
    q->value = 8;
    q->ptr = NULL;

    // set l1's ptr member variable to
    // point to the same thing as variable q
    l1->ptr = q;

    cout << "1st value: " << l1->value << "\n"
         << "2nd value: " << l1->ptr->value << endl;
}

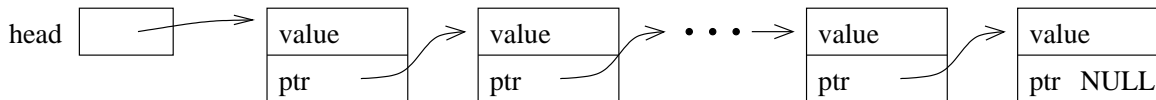
```



18.3 Definition: A Linked List

- The definition is recursive: A linked list is either:
 - Empty, or
 - Contains a node storing a value and a pointer to a linked list.
- The first node in the linked list is called the *head* node and the pointer to this node is called the *head* pointer. The pointer's value will be stored in a variable called **head**.

18.4 Visualizing Linked Lists



- The **head** pointer variable is drawn with its own box. It is an individual variable. It is important to have a separate pointer to the first node, since the “first” node may change.
- The objects (nodes) that have been dynamically allocated and stored in the linked lists are shown as boxes, with arrows drawn to represent pointers.
 - Note that this is a conceptual view only. The memory locations could be anywhere, and the actual values of the memory addresses aren't usually meaningful.
- The last node **MUST** have NULL for its pointer value — you will have all sorts of trouble if you don't ensure this!
- You should make a habit of drawing pictures of linked lists to figure out how to do the operations.

18.5 Basic Mechanisms: Stepping Through the List

- We'd like to write a function to determine if a particular value, stored in **x**, is also in the list.
- You can think of this as a precursor to the **find** function. Our function isn't yet returning an iterator, however.
- We can access the entire contents of the list, one step at a time, by starting just from the **head** pointer.
 - We will need a separate, local pointer variable to point to nodes in the list as we access them.
 - We will need a loop to step through the linked list (using the pointer variable) and a check on each value.

18.6 Exercise: Write `is_there`

```
template <class T> bool is_there(Node<T>* head, const T& x) {
```

18.7 Basic Mechanisms: Pushing on the Back

- Goal: place a new node at the end of the list.
- We must step to the end of the linked list, remembering the pointer to the last node.
 - This is an $O(n)$ operation and is a major drawback to the ordinary linked-list data structure we are discussing now. We will correct this drawback by creating a slightly more complicated linking structure in our next lecture.
- We must create a new node and attach it to the end.
- We must remember to update the `head` pointer variable's value if the linked list is initially empty.
 - Hence, in writing the function, we must pass the pointer variable **by reference**.

18.8 Exercise: Write `push_back`

```
template <class T> void push_back( Node<T>* & head, T const& value ) {
```

18.9 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?

18.10 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.

18.11 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.

18.12 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:

```
void CopyAll(Node* old_head, Node*& new_head)
```

18.13 Basic Linked Lists Mechanisms: Common Mistakes

Here is a summary of common mistakes. Read these carefully, and read them again when you have 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.

18.14 Looking Ahead to Lecture 19 — 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.