lecture09: Linked Lists

Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

24th June, 2013
Announcements

- mp2 due tonight
- mt1 tomorrow night!
- mt1 review instead of lab tomorrow morning
- mp3 released tonight, mp3.1 extra credit due Friday (6/28)
  - Note that mp3 is a solo MP!
  - Please also sign up for a code review slot on Piazza
A neat idea

The `struct` keyword is similar to the `class` keyword. The only difference is the default access scope: public for `struct`, private for `class`.

Here’s an interesting struct. What can we do with it?

```cpp
struct Node
{
    Node* next;
};

// same as
class Node
{
    public:
        Node* next;
};
```
We made the Node class a private member of the List class. That way, the List class can make use of it without exposing it to the client (encapsulation!).

class List
{
    public:
        // ...
    private:
        struct Node
        {
            // what else can go here?
            Node* next;
        };
};
Here’s a bare-bones linked list implementation. What do you imagine the public members would be?

```cpp
template <class T>
class List
{
    public:
        // ...
    private:
        struct Node
        {
            T data;
            Node* next;
        };

        Node* head; // or "first", or "start"
};
```
List ADT Functions

Here are the ADT functions for a List. Compare these to the Vector’s.

1. `insertFront`
2. `insertBack`
3. `removeFront`
4. `removeBack`
5. `size`
6. `empty`

Some other useful functions might be

- `print`
- `at`
- `sort`
Note that we are implementing the List ADT with a linked list. We could have chosen to implement the List with an array (in fact, we will look into other options later). For now, just be able to distinguish a **List** (an idea, or a concept describing a set of functions) and a **linked list** (an implementation of the List ADT).
Our List’s linked list implementation

```c++
List<int> list;
```

![Diagram of linked list](image)
Adding a node
Removing a node
Writing `insertFront`
/** @file list.cpp */

template <class T>
void List<T>::insertFront(const T & data)
{
    // now ok?
    if(head == NULL) // now ok?
    {
        head = new Node(data);
    }
    else
    {
        head->next = new Node(data);
    }
}
/** @file list.cpp */

template <class T>
void List<T>::insertFront(const T & data)
{
  // how about now?
  if(head == NULL)
  {
    head = new Node(data);
  }
  else
  {
    Node* front = new Node(data);
    front->next = head;
    head = front;
  }
}
insertBack

- How would you write insertBack?
- How fast would it be?
  - Any ideas to make it better?
- How about removeFront and removeBack?
  - Don’t want to give too much away about mp3!
Writing the print function iteratively

```cpp
/** @file list.h */

#include <iostream>
using namespace std;

template <class T>
class List
{
    public:
        void print() const;

    private:
        struct Node
        {
            Node* next;
            T data;
        };
        Node* head;
};

/** @file list.cpp */

template <class T>
void List<T>::print() const
{
    Node* cur = ;
    cout << "< ";
    while( )
    {
    }
    cout << ">
};

// print out < 1 2 3 4 5 >
```
Writing the print function recursively

```cpp
/**
 * @file list.cpp
 * Recursive printing
 */

template <class T>
void List<T>::print() const
{
    cout << '<';
    print(head);
    cout << '>' << endl;
}

template <class T>
void List<T>::print(const Node* cur) const
{
}
```
What about a reverse print function?

- Which function would you modify to print out the list backwards?
- Or would you write some entirely different function?
- Hint:
Spicing up linked lists

- A tail pointer (always points to the last element in the list)
- A size variable (update on inserts and removes, initially zero)
- Previous pointers?