C++ Abstract Data Types

Topics to cover

- Vectors (Ch 22)
- Linked Lists (Ch 17)
- Templates (Ch 14)
- Stacks/Queues (Ch 18)
The List ADT

- List: A collection of elements of the same type

- What member variables would a list have?
  - Size
  - Max Size
  - List Contents

- What are some common operations on a list?
  - size()
  - print()
  - insert()
  - remove()
  - at()
Vector

The vector class provides an *array-backed* implementation of the List ADT.

**Benefits**
- Fast insertion at end of list (`vector.push_back(...)`)  
- Fast direct access (`vector[i]`)  

**Problems**
- Require contiguous blocks of memory  
- Insert/remove at middle or front is slow  
  - requires lots of copying data around  
- Big overhead when list needs to resize  
  - have to make a copy of entire list
Linked Lists

- Rather than rely on direct indexes, use pointers to organize and process data
- Make a list of entries, but make each entry responsible for knowing where the entry after it is located
- Individual list items are called "Nodes"
  - A Node has two parts
    - Data
    - Link to another Node
If we want to make a list of integers, our Node class would look something like this:

```cpp
class Node {
    public:
        // The data section
        int data;

        // The link
        Node *next;
};
```
Linked Lists

If we have Node objects, what does it take to make a list?

- A pointer to the first Node in the list
- The size of the list
- Functions to manipulate the list

The List only has to point to the first Node... all the other Nodes will point to each other

```
head

3
\  \  \
  \  
 \  

15
\  
  \  
  

8
\  

NULL
```

Copyright © William C. Cheng
Linked Lists

If we have Node objects, what does it take to make a list?
- A pointer to the first Node in the list
- The size of the list
- Functions to manipulate the list

The List only has to point to the first Node... all the other Nodes will point to each other
class LinkedList  //currently a list of integers
{
    private:
        Node* head;  //pointer to the start of the list
        int length;  //size of the list

    public:
        LinkedList();
        ~LinkedList();
        int size() const;
        void print() const;
        int& at(int index) const;
        //etc.
};
Linked Lists

Accessing list data (without looping)

```
cout << head;       // address of item #1
cout << head->data; // prints 3

cout << head->next; // address of item #2
cout << head->next->data; // prints 15

cout << head->next->next; // address of item #3
cout << head->next->next->data; // prints 8
```
Traversing a linked list

Walk through the list item by item by following the pointers

```cpp
void LinkedList::print() const {
    Node *current = head;

    while (current != NULL) {
        cout << current->data << endl;
        current = current->next;
    }
}
```
**Linked Lists (Traversal)**

Traversing a linked list
- Walk through the list item by item by following the pointers

```cpp
void LinkedList::print() const {
    for (Node *cur=head; cur != NULL; cur=cur->next) {
        cout << cur->data << endl;
    }
}
```
Linked Lists (Insert)

- Adding an item to the list
  - If you had to pick, where would you add a new item by default?
    - e.g. vectors prefer push_back(...)

- How do you add an item?
  - Create the new item
  - Set the new item’s outgoing link
  - Set a current list item to point to the new item
Linked Lists (Insert)

Let’s insert 256 after 3

Step 1: Get a pointer to the Node just before where you want to insert. We’ll just use head for now.
Step 2: Make a new node to insert

Node *newOne = new Node();
newOne->data = 256;
newOne->next = NULL;
Linked Lists (Insert)

Step 3: Set newOne’s outgoing link to its right hand neighbor in the list

newOne->next = head->next;
Step 4: Set an incoming link to newOne from its left hand neighbor in the list

head->next = newOne;
Linked Lists (Insert)

Done!
- Don’t need to touch newOne because it will just go out of scope eventually

What if we need to insert in the middle of the list?
- Loop until you find the node right before where you want to insert your new node

1. Head
2. 3
3. 256
4. 15
5. 8
6. newOne
Linked Lists (Remove)

How do you remove an item?

- Get a pointer to the node you want to remove
- Set the node’s incoming link to point past it to its outgoing link
- Call `delete` on the node that’s been cut out of the chain
Let’s remove the Node with value 15

Step 1: Get a pointer to the node you want to remove

Node* toDelete = head->next;
Step 2: Set the removal item’s incoming link to point to its outgoing link

\[ \text{head} \rightarrow \text{next} = \text{toDelete} \rightarrow \text{next}; \]
Step 3: Call `delete` on the Node with 15

```c
delete toDelete;
```
Linked Lists (Remove)

Done!

- You don’t actually need to call `delete` for this to work, but then you’d have a memory leak

What if we need to remove in the middle of the list?

- Loop and find pointers to the node you want to remove AND the node before it

![Diagram of linked list with head pointing to node 3 and node 8]
Insertion at the front of the list is **FAST**

- What could we do if we wanted to speed up insertions at the back of the list?
**Linked Lists**

- Insertion at the front of the list is *FAST*
  - What could we do if we wanted to speed up insertions at the back of the list?

- Add a "tail" pointer to go along with "head"
  - Insert at front updates "head"
  - Insert at back updates "tail"
Linked Lists

Traversing the list in one direction is easy

What if we want to traverse backward or make insertion/removal a bit easier?
Traversing the list in one direction is easy
- What if we want to traverse backward or make insertion/removal a bit easier?

Make a Doubly Linked List
- Each Node now points to its next \textit{AND} previous
Linked Lists

- Traversing the list in one direction is easy
  - What if we want to traverse backward or make insertion/removal a bit easier?

- Make a Doubly Linked List
  - Each Node now points to its next AND previous

- The real reason to use a doubly linked list is to make insertion/removal fast
Arrays/Vectors vs. Linked Lists

Arrays/Vectors

- **Advantages:**
  - Fast direct access with indexing
  - Fast insertion at end of list

- **Disadvantages:**
  - Requires one contiguous block of memory
  - Insert/remove at front or middle requires re-ordering of data
  - Must dynamically resize periodically

- Conclusion:
  - Great for lists where you don’t insert or remove often, but do a lot of direct element access
Arrays/Vectors vs. Linked Lists

Linked Lists

- **Advantages:**
  - Fast insertion & removal
  - Easy to grow/shrink list dynamically
  - Nodes can be located anywhere in memory (don’t need to be contiguous)

- **Disadvantages:**
  - No direct indexing

- **Conclusion:**
  - Great for lists where you insert or remove often, but only do traversals, not direct element access
Flexible Linked Lists

The Linked Lists we made are useful, but what if we want a Linked List to store something besides \textit{int} types? Do we have to define a whole new List class for each data type? That seems absurd.

What about that syntax we use on vectors?

\begin{verbatim}
vector<int> numbers;
\end{verbatim}
CS102
Extra Slides
Bill Cheng

http://merlot.usc.edu/cs102-s12
/*
 * 1) Create a LinkedList class.
 * 2) Define class Node to have two members, data and next.
 * 3) Define a overloaded constructor for class Node.
 * 4) Define class LinkedList to have two members, head and length.
 * 5) Need a constructor and a destructor.
 * 6) Public member functions:
 *    - int size() const
 *    - bool empty() const
 *    - void print() const
 *    - int& at(int index)
 *    - Node* find(int value) const
 *    - bool contains(int value) const
 *    - void append(int value)
 *    - void prepend(int value)
 *    - void remove(Node* node_ptr)
 *    - void erase(int index)
 */
```cpp
#ifndef ADT_LINKEDLIST_H_  
#define ADT_LINKEDLIST_H_  

class Node  
{  
  public:  
    int data;  
    Node* next;  
  public:  
    Node(int value);  
};

class LinkedList  
{  
  private:  
    Node* head;  
    int length;  
  public:  
    LinkedList();  
    ~LinkedList();  
    int size() const;  
    bool empty() const;  
    void print() const;  
    int &at(int index);  
    Node* find(int value) const;  
    bool contains(int value) const;  
    void append(int value);  
    void prepend(int value);  
    void remove(Node* node_ptr);  
    void erase(int index);  
};

#endif
```
#include <iostream>
#include <stdexcept>
#include "adt_linkedlist.h"

using namespace std;

Node::Node(int value)
{
    this->data = value;
    this->next = NULL;
}

LinkedList::LinkedList()
{
    this->length = 0;
    this->head = NULL;
}

LinkedList::~LinkedList()
{
    Node *next_node=NULL;
    for (Node *node_ptr=this->head; node_ptr != NULL; node_ptr=next_node) {
        next_node = node_ptr->next;
        delete node_ptr;
    }
}
int LinkedList::size() const
{
    return(this->length);
}

bool LinkedList::empty() const
{
    return(this->length == 0);
}

void LinkedList::print() const
{
    for (Node *node_ptr=this->head; node_ptr != NULL;
         node_ptr=node_ptr->next) {
        cout << node_ptr->data << " ";
    }
    cout << endl;
}
int& LinkedList::at(int index)
{
    if(index < 0 || index >= this->length) {
        throw out_of_range("index out of bounds");
    }
    Node *node_ptr;
    for (node_ptr=this->head; node_ptr != NULL;
         node_ptr=node_ptr->next) {
        if (index == 0) {
            break;
        }
        index--;
    }
    return node_ptr->data;
}

Node* LinkedList::find(int value) const
{
    for (Node* node_ptr=this->head; node_ptr != NULL;
         node_ptr=node_ptr->next) {
        if (node_ptr->data == value) {
            return node_ptr;
        }
    }
    return NULL;
}
bool LinkedList::contains(int value) const
{
    Node* node_ptr=find(value);
    return node_ptr != NULL;
}

void LinkedList::append(int value)
{
    if (this->head == NULL) {
        Node *new_node=new Node(value);
        this->head = new_node;
    } else {
        Node *last_node=NULL;
        for (Node *node_ptr=this->head; node_ptr != NULL; 
             node_ptr=node_ptr->next) {
            last_node = node_ptr;
        }
        Node *new_node=new Node(value);
        last_node->next = new_node;
    }
    this->length++;
}

void LinkedList::prepend(int value)
{
    Node *new_node=new Node(value);
    new_node->next = this->head;
    this->head = new_node;
    this->length++;
}
void LinkedList::remove(Node* target_node_ptr)
{
    Node* prev_ptr=NULL;
    Node *node_ptr;
    for (node_ptr=this->head; node_ptr != NULL && node_ptr !=
         target_node_ptr; node_ptr=node_ptr->next) {
        prev_ptr = node_ptr;
    }
    if (node_ptr == NULL) {
        throw target_node_ptr;
    } else if (prev_ptr == NULL) {
        this->head = target_node_ptr->next;
        delete target_node_ptr;
    } else {
        prev_ptr->next = target_node_ptr->next;
        delete target_node_ptr;
    }
}
void LinkedList::erase(int index) {
    if(index < 0 || index >= this->length) {
        throw out_of_range("index out of bounds");
    }
    Node* prev_ptr=NULL;
    Node *node_ptr;
    for (node_ptr=this->head; node_ptr != NULL; 
     node_ptr=node_ptr->next) {
        if (index == 0) {
            break;
        }
        index--;
        prev_ptr = node_ptr;
    }
    if (prev_ptr == NULL) {
        this->head = node_ptr->next;
        delete node_ptr;
    } else {
        prev_ptr->next = node_ptr->next;
        delete node_ptr;
    }
}
/*
  * 1)  Test out the operations on the new LinkedList class.
  * 2)  Create an instance of the LinkedList class.
  * 3)  Print out whether the list is empty or not.
  * 4)  for (i=0; i < 6; i++), call append() to append i
  *     to the list.  Print the size and content of the list
  *     after each push_back().
  * 5)  for (j=11; j > 6; j--), call prepend() to
  *     prepend j to the list.  Print the size and content of
  *     the list after each push_front().
  * 6)  Print out whether the list is empty or not.
  * 7)  Prints the 6th element on the list.
  * 8)  Prints the 2nd element on the list.
  * 9)  Prints if the list contains the integer 55.
  * 10) Prints if the list contains the integer 4.
  * 11) Remove the 1st element of the list and print the list.
  * 12) Remove the 5th element of the list and print the list.
*/
#include <iostream>
#include <iomanip>
#include "adt_linkedlist.h"
using namespace std;

int main()
{
    LinkedList* list = new LinkedList();
    cout << "Empty = " << boolalpha << list->empty() << endl;
    for(int i=0; i < 6; i++)
    {
        list->append(i);
        cout << "List size = " << list->size() << endl;
        list->print();
    }
    for(int j=11; j > 6; j--)
    {
        list->prepend(j);
        cout << "List size = " << list->size() << endl;
        list->print();
    }
    cout << "Empty = " << boolalpha << list->empty() << endl;
    cout << "list->at(5) = " << list->at(5) << endl;
    cout << "list->at(1) = " << list->at(1) << endl;
    cout << "contains(55) = " << list->contains(55) << endl;
    cout << "contains(4) = " << list->contains(4) << endl;
    list->erase(0);
    list->print();
    list->erase(5);
    list->print();
}