CS102
C++ Abstract
Data Types & Link Lists
The List ADT

What is an ADT?
- An abstract data type that separates the logical properties from the implementation details (Encapsulation, Information Hiding)

The List ADT
- We can define all the types of things we want to do with a List without really specifying how they should be implemented
  - insert, remove, search, etc.

The C++ vector is a good example
- How does vector work?
adl_list.h

/*
  1) Write an abstract data type (ADT) representation of a List of integers. This class is purely an interface (it has no actual implementation details).
  2) The contains() method checks to see if an integer is stored in the list.
  3) The push_front() method pushes an integer into the front of the list (prepends).
  4) The push_back() method appends an integer to the front of the list.
  5) The at() method returns a reference to the integer at the given index.
  6) The erase() method erases the integer at the given index. (In STL’s list, the argument is NOT an integer.)
  7) The empty() method returns true if the list is empty.
  8) The size() method returns the number of elements in the list.
  9) The print() method prints all the integers in the list.
*/
#ifndef ADT_LIST_H
#define ADT_LIST_H

class List {
    public:
        virtual bool contains(int val) const = 0;
        virtual void push_front(int val) = 0;
        virtual void push_back(int val) = 0;
        virtual int& at(int index) = 0;
        virtual void erase(int index) = 0; // different from STL
        virtual bool empty() const = 0;
        virtual int size() const = 0;
        virtual void print() const = 0;
    }

#define ADT_LIST_H_
/*
 * 1) The Vector class is a subclass of the List class.
 * 2) What does it need?
 * 3) Constructor and destructor.
 * 4) All the member functions of List.
 * 5) What private members should it have?
 *    - length to know how many elements are stored
 *    - capacity to know how many locations are allocated
 *    - data to store the integers - in this implementation,
 *      we will use a dynamic array
 * 6) Because it needs to be resized from time to time,
 *    need to have a private resize() method
 */
#ifndef ADT_VECTOR_H
#define ADT_VECTOR_H_
#include "adt_list.h"

class Vector : public List {
    protected:
        int length;
        int* data;
        int capacity;

    public:
        Vector();
        ~Vector();

        virtual bool contains(int val) const;
        virtual void push_front(int val);
        virtual void push_back(int val);
        virtual int& at(int index);
        virtual void erase(int index); //different from STL
        virtual bool empty() const;
        virtual int size() const;
        virtual void print() const;

    private:
        void resize();
};

#endif
adt_vector.cpp

/*
 * 1) Implement the Vector class, which is a subclass of the
 *    pure virtual base List class.
 * 2) Constructor: dynamically allocate an array to hold all
 *    of our data
 * 3) Destructor: must destroy the dynamic data that we
 *    allocated in the constructor
 * 4) bool Vector::empty() const
 * 5) int Vector::size() const
 * 6) bool Vector::contains(int val) const
 * 7) void Vector::push_front(int val) - add item to the
 *    front of the list
 * 8) void Vector::push_back(int val)
 * 9) void int& Vector::at(int index) - return the item at
 *    the requested index or throw an exception if the index
 *    is bad
 * 10) void Vector::erase(int index) - erase the item at the
 *     given index or throw an exception if the index is bad
 * 11) void Vector::print() const - print the vector to cout
 * 12) void Vector::resize() - resize the vector by doubling
 *     the capacity of its internal array
 */
```cpp
#include <iostream>
#include <stdexcept>
#include "adt_vector.h"

using namespace std;

#define INITIAL_CAPACITY 1

Vector::Vector()
{
  this->length = 0;
  this->capacity = INITIAL_CAPACITY;
  this->data = new int[INITIAL_CAPACITY];
}

Vector::~Vector()
{
  delete [] this->data;
  this->data = NULL;
}

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

int Vector::size() const
{
  return(this->length);
}
```
bool Vector::contains(int val) const
{
    for(int i=0; i < this->length; i++) {
        if(this->data[i] == val) {
            return(true);
        }
    }
    return(false);
}

void Vector::push_front(int val)
{
    if(this->length == this->capacity) {
        resize();
    }
    for(int i=this->length; i > 0; i--) {
        this->data[i] = this->data[i-1];
    }
    this->data[0] = val;
    this->length++;
}

void Vector::push_back(int val)
{
    if(this->length == this->capacity) {
        resize();
    }
    this->data[this->length] = val;
    this->length++;
}
int& Vector::at(int index)
{
    if(index < 0 || index >= this->length) {
        throw out_of_range("index out of bounds");
    }
    return(this->data[index]);
}

void Vector::erase(int index)
{
    if(index < 0 || index >= this->length) {
        throw out_of_range("index out of bounds");
    }
    for(int i=index; i < this->length-1; i++) {
        this->data[i] = this->data[i+1];
    }
    // What if this is not a vector of int but a vector of objects?
    // Need to invoke the destructor of this->data[length-1]
    // Can only do this if we know this->data[length-1]’s data type
    this->length--;
}

void Vector::print() const
{
    for(int i=0; i < this->length; i++) {
        cout << this->data[i] << " ";
    }
    cout << endl;
}
void Vector::resize()
{
    int* oldArr = this->data;
    this->capacity *= 2;
    this->data = new int[this->capacity];
    for(int i=0; i < this->length; i++) {
        this->data[i] = oldArr[i];
    }
    delete [] oldArr;
}
adt_main.cpp

/*

1) Test out the operations on the new Vector class.
   Pass it around as a List instead of a Vector for
   flexibility (more on that later).
2) Create a vector and treat it as a List.
3) Print out whether the list is empty or not.
4) for (i=0; i < 6; i++), call push_back() 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 push_front() 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) Erase the 1st element of the List and print the list.
12) Erase the 5th element of the List and print the list.
*/
#include <iostream>
#include <iomanip>
#include "adt_list.h"
#include "adt_vector.h"
using namespace std;

int main()
{
  List* list = new Vector();
  cout << "Empty = " << boolalpha << list->empty() << endl;
  for(int i=0; i < 6; i++) {
    list->push_back(i);
    cout << "List size = " << list->size() << endl;
    list->print();
  }
  for(int j=11; i > 6; j--) {
    list->push_front(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();
}
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:

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
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
Linked Lists

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();
        ~LinkedList();

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

Accessing list data (without looping)

```cpp
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

head

```
3

15

8
```
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.
Linked Lists (Insert)

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;
Linked Lists (Insert)

Step 4: Set an incoming link to newOne from its left hand neighbor in the list

\[ \text{head} \rightarrow \text{next} = \text{newOne}; \]
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

```
head ← newOne
```

```
3 256 15 8
```
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 \textit{delete} on the node that’s been cut out of the chain

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{linked_lists_remove.png}
\caption{Linked List Removal}
\end{figure}
Linked Lists (Remove)

Let’s remove the Node with value 15

Step 1: Get a pointer to the node you want to remove
Node* toDelete = head->next;

head

3

15

8

toDelete
Step 2: Set the removal item’s incoming link to point to its outgoing link

```c
head->next = toDelete->next;
```

```
    +-----------------+
    |                 |
    |                 |
    |     3          |
    |     +----------+
    |                 |
    |     15         |
    |     +----------+
    |                 |
    |     8          |
    +-----------------+
```

Linked Lists (Remove)
Step 3: Call `delete` on the Node with 15
```cpp
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 a linked list with nodes and pointers](image)
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?
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"

![Diagram of linked list with head and tail pointers]
Linked Lists

Traversing the list in one direction is easy

What if we want to traverse backward or make insertion/removal a bit easier?
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
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 *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?

`vector<int> numbers;`
CS102
Extra Slides
*/

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)

*/
#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
```cpp
#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;
    }
}
**adt_main.cpp**

/*
 * 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 > 0; 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();
}