CMPT 225

Lecture 6 – **List** ADT class – **Step 4 – Testing**

Our **next linear data collection**

Learning Outcomes

- At the end of this lecture, a student will be able to:
  - design and use test cases

Last Lecture

- We can now …
  - Perform operations on a **linked list**
  - Create **linked list** of various configurations (SHSL list, DHSL list, DHDL list, …)
    - Know when to use them (know their ***forte***)
  - Step 3 – Implement a data collection **List** ADT class:
    - Using an **array** (heap-allocated)
    - Using a **linked list**
      - Create a **Node** class
  - Compare the efficiency of the methods for both implementations of our **List** ADT class

Why are we doing such comparison?

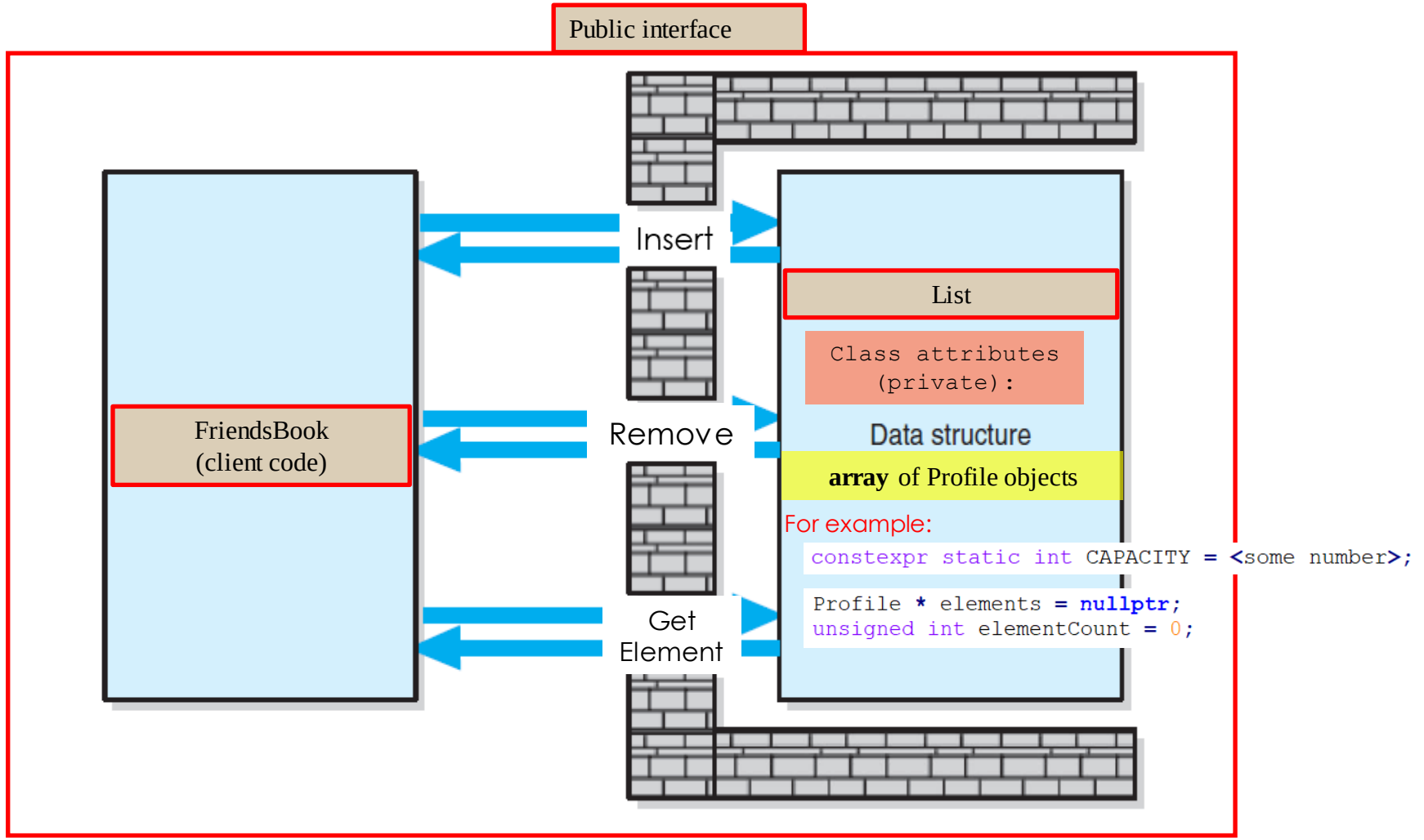
If your **complexity analysis** is a bit rusty, please, read **Review of Complexity Analysis and Big O Notation** posted under Lecture 6 as **Reading**

Today's Menu

- Results from comparing both **List** implementations (array versus linked list – both tables) done in Lecture 5 will soon be posted on our course web site!
- Finishing Step 3 – Implement a data collection **List** ADT class
  - Documentation
- **Step 4 – Compilation and Testing** of **List** ADT
  - Introduce ***white-box testing***, ***test cases*** and ***test driver***
- Introduce our **next linear data collection**

Summary

Array-based implementation List ADT class

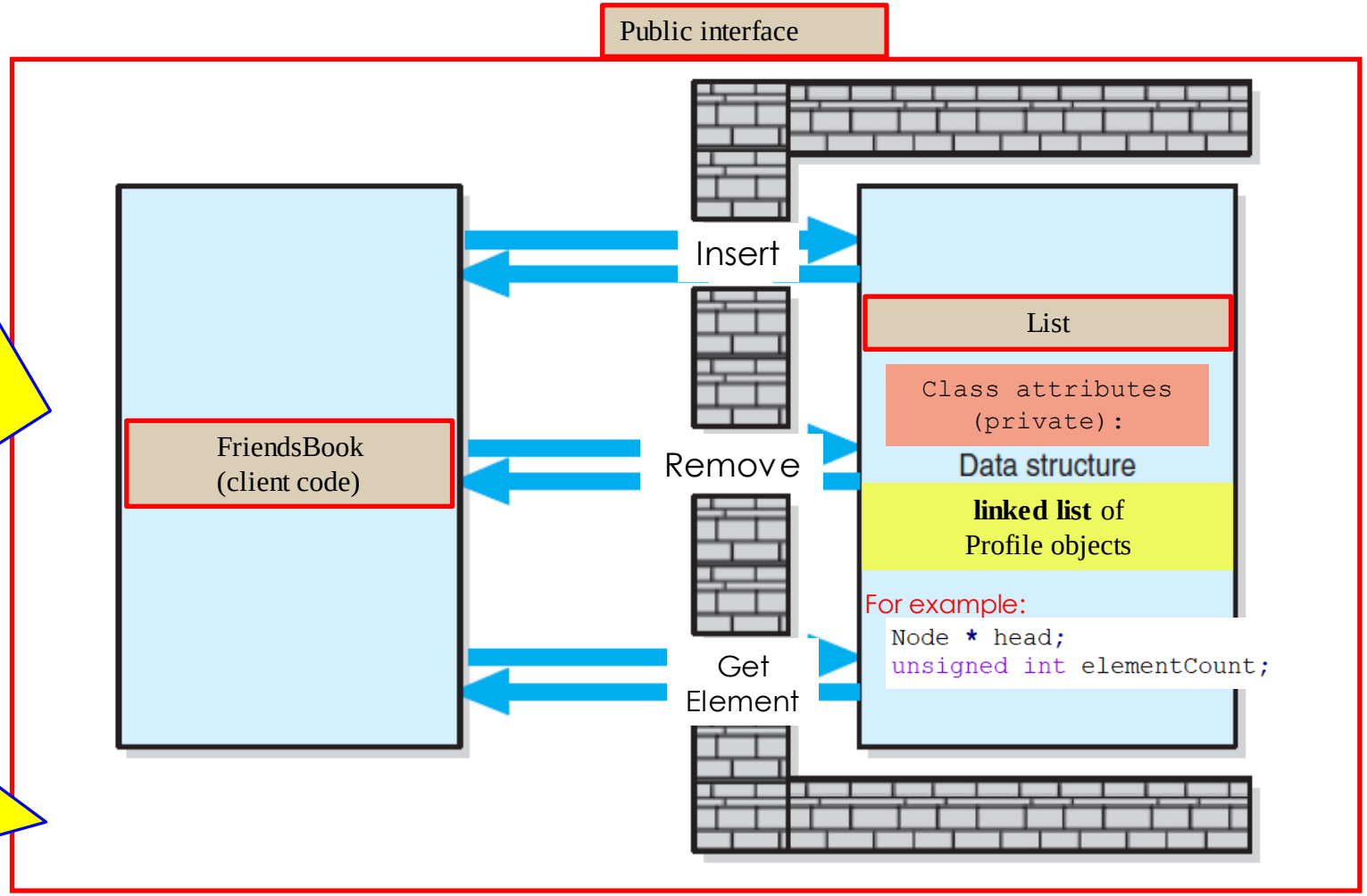


Summary

Linked list-based implementation List ADT class

Would we need to modify our **FriendsBook.cpp** (client code) if we were to replace the CDT of our **List** ADT class, i.e., its array, with a linked list?
Answer: _____

This illustrates the power of ADT classes, i.e., one of their advantages. ☺



Step 3 – Implement a data collection **List** ADT class - Documentation

- **Header comment block**
  - Filename (*.h or *.cpp)
  - Class description
  - Class invariant (if any)
  - Author
  - Creation/last modification date
- **Contract for each public and private methods**
  - Description -> single responsibility
  - Precondition: What must be true before the method is called
    - We need to test this (if possible) at beginning of the method
  - Postcondition: What is true after the method has executed
  - Time Efficiency (optional)
  - Exception Handling

This documentation is placed in both ***.h** and ***.cpp** files

Step 3 – Implement a data collection **List** ADT class – Documentation – An example

From **`MyADT`** class from our Assignment 1:

```
// Description: Inserts an element in the data collection MyADT.
//              Returns "true" when the insertion is successfull, otherwise "false".
// Precondition: newElement must not already be in the data collection MyADT.
// Postcondition: newElement inserted, MyADT's class invariants are still true
//                and the appropriate elementCount has been incremented.
// Time Efficiency:
bool insert(const Profile& newElement);

// Description: Removes an element from the data collection MyADT.
//              Returns "true" when the removal is successfull, otherwise "false".
// Precondition: The data collection MyADT is not empty.
// Postcondition: toBeRemoved (if found) is removed, MyADT's class invariants are still true
//                and the appropriate elementCount is decremented.
// Time Efficiency:
bool remove(const Profile& toBeRemoved);
```

Step 4 – Compilation and Testing – *white box testing strategy*, *test cases* and *test driver*

- From Wiki:

  White-box testing is a method of software testing that tests internal structures or workings of an application, as opposed to its functionality. In white-box testing, an internal perspective of the system is used to design test cases.

  Source: https://en.wikipedia.org/wiki/White-box_testing

- From Check Point Software:

  White box testing is a form of application testing that provides the tester with complete knowledge of the application being tested, including access to source code and design documents. This in-depth visibility makes it possible for white box testing to identify issues that are invisible to gray and black box testing.

  Source: https://www.checkpoint.com/cyber-hub/cyber-security/what-is-white-box-testing/

White (or clear) box testing strategy

- The idea is that the code "is in a box" and because the box is white (or clear), we can see the code to be tested.
- As opposed to having the code in a black box (black box testing strategy), where we would not be able to see the code to be tested.
- (From our Lab 2): Our goal is to execute each of the code statements in our program.
  - Most often, requires several test cases

Testing an application (like `circle_array.cpp` or `FriendsBook.cpp`)

- Our goal is to execute each of the code statements in the **`main`** function and all other functions (if any) at least once
  - Most often, requires several test cases

  - No need for a test driver in this situation
  - Testing is done by following each of the test cases (you have created) while executing the application
    - You enter the specific test data stated in your test case (if any)
    - You observe the results of the executing application by comparing your expected results (as stated in your test cases) against the actual results the application produces and prints

Testing a class (like `Circle` or `MyADT`)

- Our goal is to execute each method of the class at least once and to execute the code statements in each of these methods at least once
  - Most often, requires several test cases

- Create sufficient number of test cases in order to achieve complete coverage -> executing each statement in each of the class' methods at least once
- In this situation, you implement your test cases in a test driver program
- Testing is done by executing your test driver and comparing its results against your expected results (as stated in your test cases)

Step 4 – Compilation and Testing - *white-box testing strategy*, *test cases* and *test driver*

- A test case has three parts:
  1. *Specific* test data
  2. Expected results
  3. Actual results

Must be created before testing starts!

Obtained as we execute our solution to the problem

- How many test cases do we need?
- What are distinct test cases?

Step 4 – Compilation and Testing - *white-box testing strategy*, *test cases* and *test driver*

- Once you have designed your test cases, i.e., their test data and expected results, "implement" them by writing a test driver program for the class you are testing
- Each class must be tested using a test driver program
  - Goals of test driver:
    - Call each of the class' methods at least once
    - Break the code using invalid test data

Demo: Test driver

Using the TemperatureTestDriver.cpp of Lecture 2:

```
// Create a valid Celsius temperature
cout << endl << "Create a valid default Celsius temperature -> testing constructor Temperature()." << endl;
cout << "Expected Result: temperature = 0.0 C" << endl;
Temperature tempCelsius;
cout << "Actual Result: temperature = " << tempCelsius.getDegrees( ) << " " << tempCelsius.getScale() << endl;
```

How to structure your test driver:

1. **Comment**: Describe the test case - using the example above:
   `// Create a valid Celsius temperature`

2. Output the description of this test case - using the example above:
   `cout << endl << "Create a valid default Celsius temperature -> testing constructor Temperature()." << endl;`

3. Output the test data used in this test case, if any (no test data required in the test case used in the example above)

Demo: Test driver (cont'd)

4. Output the expected result - using the example on the previous slide:
```
cout << "Expected Result: temperature = 0.0 C" << endl;
```

5. Execute the test case - using the example on the previous slide:
```
Temperature tempCelsius;
```

6. Output the actual results - using the example on the previous slide:
```
cout << "Actual Result: temperature = " << tempCelsius.getDegrees( )
<< " " << tempCelsius.getScale() << endl;
```

7. Then visually verify that expected results match actual results.

   You can replace Step 6 with an assert():
```
assert( tempCelsius.getDegrees( ) == 0.0 &&
        tempCelsius.getScale() == 'C');
```

Note that in this test case, we also test `getDegrees()` and `getScale()` by calling them to verify that the default constructor executed successfully.

Our next linear data collection -> _____

- Activity

√ Learning Check

- We can now …
  - Create class documentation (contract) for our class
  - **Step 4 – Compilation and Testing** of **List** ADT
    - Describe ***white box testing strategy***
    - Create ***test cases*** based on this strategy
    - Implement a ***test driver*** for a class based on these test cases
- Describe the linear data collection Stack, design and implement a Stack ADT class and lastly use it to solve problem – right?

Next Lecture

- Brief review of **Complexity Analysis and Big O Notation**
  - Reading posted under Lecture 6
- Solve a problem using a Stack ADT class
- Design and implement a Stack ADT class
  - Define its Public Interface
  - Design (and draw) and implement Stack ADT using various data structures (CDTs)
  - Compare and contrast these various implementations using Big O notation
  - Give examples of real-life applications (problems) where we could use Stack to solve the problem