CMPT 225

A grocery list

Lecture 3 – Data collection List – as an ADT

Learning Outcomes

- At the end of this lecture, a student will be able to:
  - define ***data collection ADT*** (designed and implemented as an abstract data type - ADT) and ***data structure*** (concrete data type - CDT) and differentiate between the two
  - define one of the concrete data types, namely *array*, and demonstrate, simulate, and trace its operations
  - convert specifications into high-level design, apply software engineering and design concepts, and express OO design in UML class diagrams
  - write C++ code
  - encapsulate methods and variables for an ADT into a C++ class

Last Lecture

- Introducing the concept of Abstract Data Type (ADT)
  - Definition
  - How to design an ADT
  - How to implement an ADT in C++
  - How to test an ADT
- Example: Temperature class
  - Implemented as an ADT

4 steps of the software development process:

Step 1 - Problem statement + Requirements
Step 2 – Design
Step 3 – Implementation
Step 4 – Compilation and Testing

Let's review some of this …

Let's have a look at **Slides 20 to 26** from our **Lecture 2** as a way of reviewing!

- Implementing a class as an ADT
  - Introducing the **Wall** metaphor
- Implementing a class as a non-ADT

- Homework from last lecture:
  - Advantages and disadvantages of ADT

Today's menu

- Overview of data collections
- Introduce our first data collection: List
- Design List as an ADT
  1. Design the visible section of our List ADT class
     - Its public interface –> its ***public*** section (the gaps in the wall)
  2. Design the invisible section of our List ADT class
     - Its ***private*** section (what is hidden behind the wall)
- Look at arrays (one of the concrete data structures)

Data Collection versus Data Structure

(abstract data type - **ADT**) (concrete data type - **CDT**)

Not every **ADT** class is a data collection class, but every data collection class is an **ADT** class.

- ADT class that models a collection of data
- Example: List

- Data structures are constructs available as part of a programming language
- Examples:
  - array
  - linked list (nodes and pointers)
- Used as member attributes (data) of an ADT
- Hidden behind the wall

Categories of **data organizations**

- **Linear**
  - Data organization in which each element has a unique predecessor (except for the first element, which has none) and a unique successor (except for the last element, which has none)
- **Non-Linear**
  - Data organization in which there is no first element, no last element and for each element, there is no concept of a predecessor and a successor

Categories of **data organizations** – cont'd

- **Hierarchical**
  - Data organization in which each element has only one predecessor -> its parent (except for the first element, which has none) and up to many successors (except for the last element(s), which has none)
- **Graph**
  - Data organization in which each element can have many predecessors and many successors

What is a List in a real world?

- Characteristics?
  - Duplications allowed?
  - Sorted?
- Any constraints or limitations?

A grocery list

List as data collection in the software world

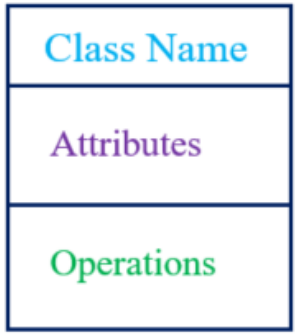
- Let's introduce our first data collection: List while solving a problem!
- Step 1 - Problem Statement + Requirements
  - FriendsBook Application
  - Design and implement an application that maintains the data for a simple social network.
  - Each person in the network must have a profile that contains the person's name, optional image, current status and a list of friends.
  - Your application must allow a user to join the network by creating a profile, leave the network, modify the profile, search for other profiles, and add friends.

Source: Walls and Mirrors Textbook 6th Edition - Programming Problem 11 - Chapter 9 - Page 287

Step 2 – Design

- Describe the behaviour of our FriendsBook application by listing the steps it will perform when it will execute:
  - Display menu
    - join the network (create a profile)
    - leave the network
    - modify the profile
    - search for other profiles
    - add friends.
  - Read user choice
  - Perform requested action
    - Ask user for more info, if needed
  - Display results
  - Repeat the above until user quits

Step 2 – Design



FriendsBookApp
- collection of profiles
+ join + leave + modify + search

Profile
- name - image - status - number of friends - friends[0 .. *]

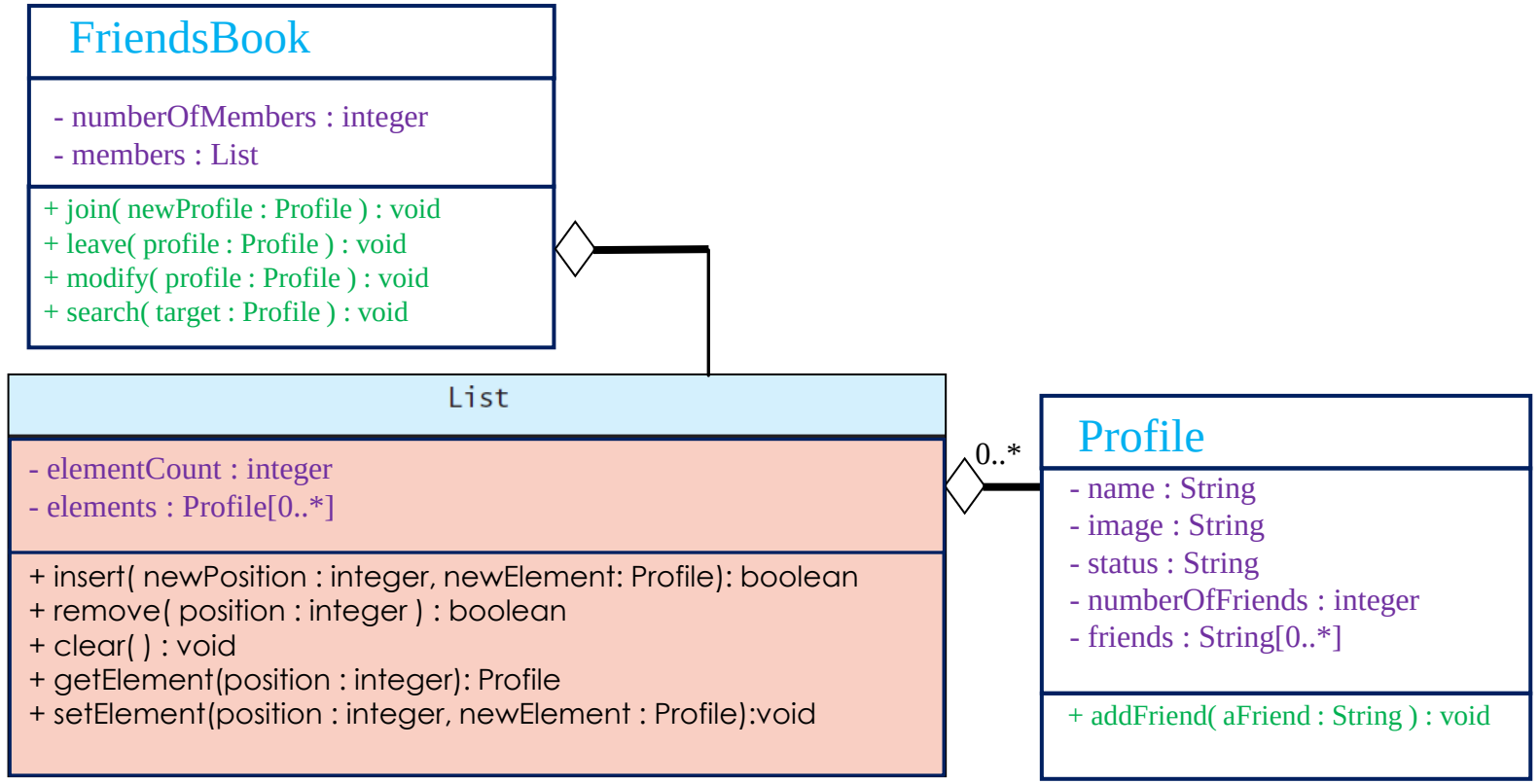
Collection of Profiles
- profile[0 .. *] - number of profiles

Step 2 – Design – Result - OPTIONAL

Optional –
If you are curious, here is a complete UML class diagram Describing our design so far.

Note: We shall not cover the UML syntax used on this page in this course. We shall only cover the UML syntax covered on the previous slide.

Record our design using a UML class diagram



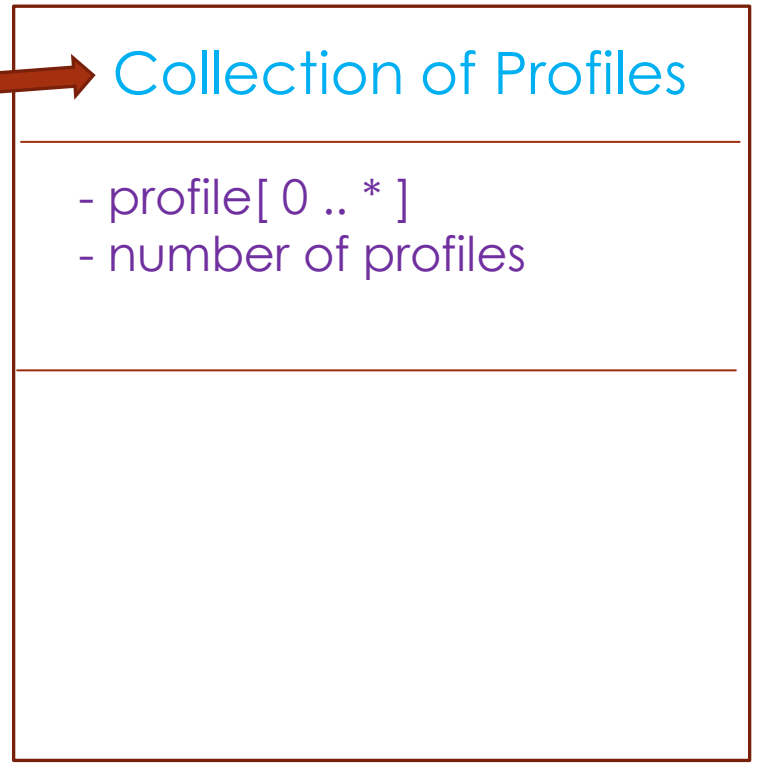
Step 2 – Design

- As part of Step 2, we need to decide which **data collection** to use

  and whether data collection should keep elements in a certain sort order

  and whether duplicated elements are allowed

  and what kind of **data structure** we shall use to implement it.



- So we decide to design a **List** as the **data collection** ADT class

  considering the problem we are solving, it makes no sense to allow for duplicated elements and to keep the elements in sort order, but in sort order of what?

Terminology

Two kinds of List

- **Position-oriented**
  - Operations done based on position of data
- **Value-oriented**
  - Operations done based on value of data

Step 2 – Design List's public interface

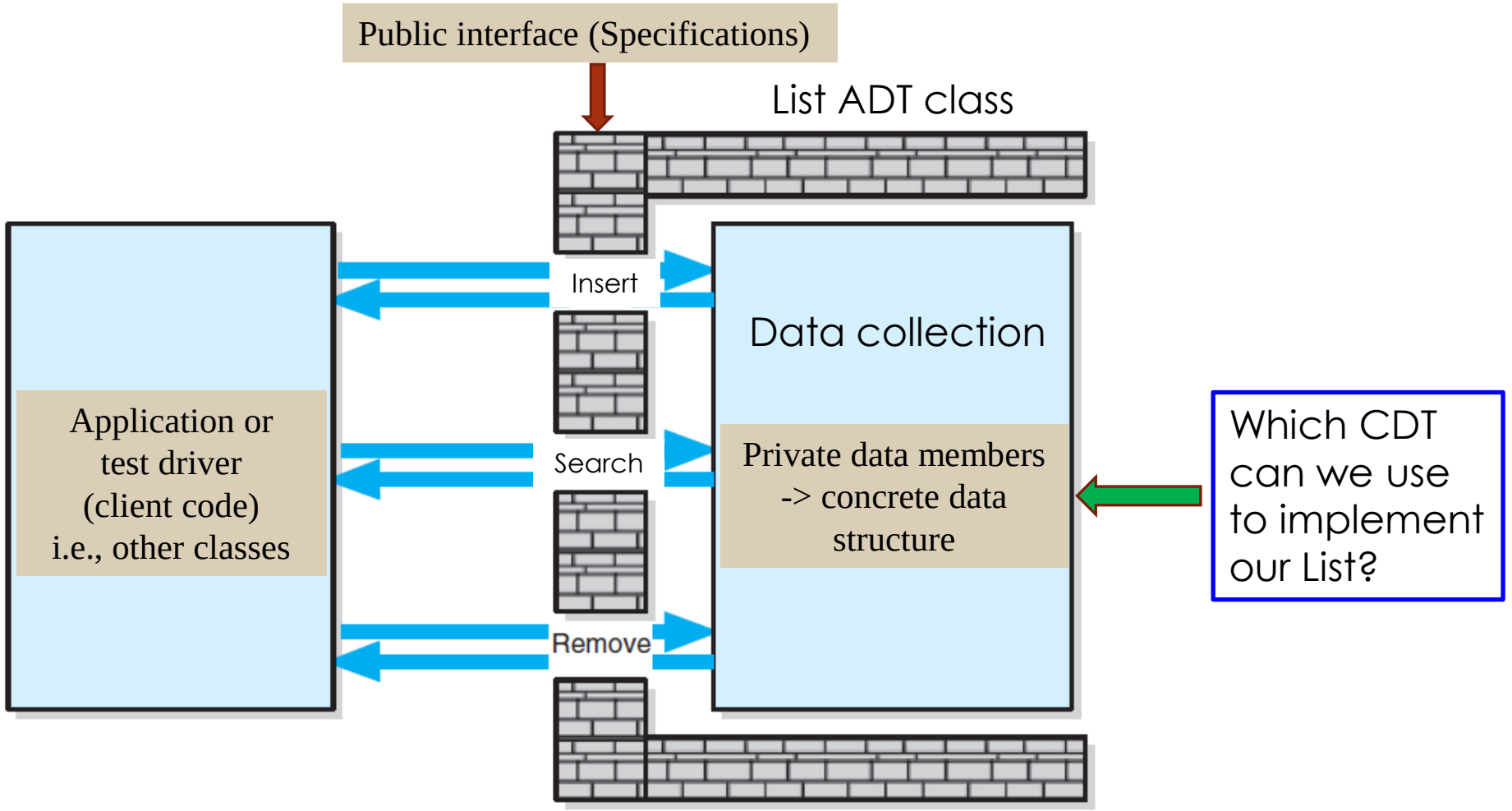
- insert
-
-
-
-
-

=> Always design operations optimizing their time efficient

Example of value-oriented List operations

- insert an element in the list at location determined by its value
  - insert(element)
- remove a specific element
  - remove(element)
- remove all the elements from the list
  - removeAll( )
- get a specific element
  - *element* get(element)
- *how many elements are in the list*
  - *integer getElementCount( )*

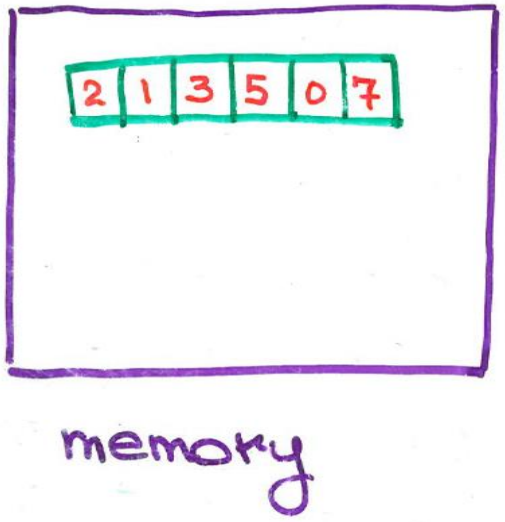
List as an Data Collection ADT class

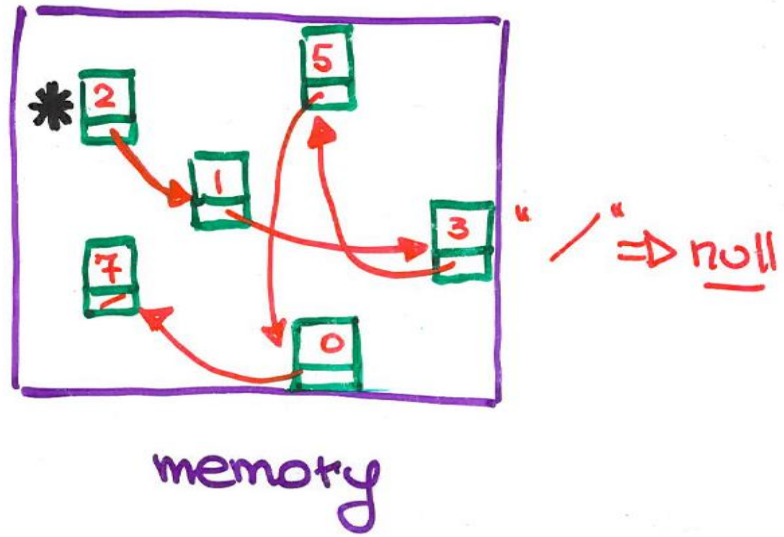


The wall of a data collection ADT isolates its data structure from the client code that uses it.

Introduce our first concrete data structure: Array

- Concrete data structure (CDT)
- Contiguous memory locations are used when an array is allocated
- Indexed data structure
  - **Direct** access (as opposed to **sequential** access)





What can we do with an array

- Insert element into it – how?
- Remove element from it – how?
  - no need to "erase" an element from an array cell, simply overwrite it
- Traverse (iterate through)
- Search for (find/get) a particular element – may not visit every element
- "Expand" (resize) an array
  (see Expandable Array under **Lecture 3** for more information)

- Note: Easier to manage when there are no gaps in the array!

Advantages of arrays

- Indexing (e.g., elements[3]) is very time efficient
- Because array cells are assigned contiguous memory locations, traversing an array is very simple

Disadvantages of arrays

- All elements stored in an array must have the same data type
- In C++, an array does not know its size (capacity), i.e., the number of cells it has
  - We must keep track of its size using a variable
- Size is required when creating an array
  - If we overestimate the size of our array -> we waste memory
  - If we underestimate the size of our array -> we risk running out of space
    - Solution: expand array -> How expensive is this operation?

      (see Expandable Array under **Lecture 3** for more information)
- In C++, there are no bound check on array indices, so we must ensure that our index is 0 <= index < array size

Note about **STL in C++**

- Most of the time, in a software development project, we do not design and implement data collections ADT classes, instead we make use of what is already available
  - Examples: STL vectors

- But in CMPT 225, we will design our own data collection ADT classes -> Why?

- This means that in our assignments and exams, we cannot make use of library data collections ADT classes (like STL vectors)

√ Learning Check

- We can now …
  - Explain the difference between a data collection (designed and implemented as an abstract data type - ADT) and data structure (concrete data type - CDT) and differentiate between the two
  - List different categories of data organization
  - Describe a List ADT
  - Design a List as an ADT
    - Design its public interface (***public*** section)
    - Design its invisible/hidden section (***private*** section)
      - Using concrete data structure
  - Look at arrays (one of the concrete data structures)

Next Lectures

- Step 3 – Implementation of the array-based List ADT
  - Introduce dynamically allocated memory
  - Introduce linked lists (another concrete data structure)
  - Implement the link-based List ADT and test it
- Step 4 – Compilation and Testing of the array-based List ADT
  - Introduce *test cases*