

Data Abstraction Sequence Data Interfaces to Data

CS 311 Data Structures and Algorithms

Lecture Slides

Monday, March 24, 2008

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

CHAPPELLG@member.ams.org

© 2005–2008 Glenn G. Chappell

Where Are We?

From the First Day of Class: Course Overview — Goals

After taking this class, you should:

- Have improved proficiency in programming and software design.
- Be able to apply basic ideas behind analysis of algorithmic efficiency, including “big- O ” notation.
- Be familiar with various standard algorithms, including those for searching and sorting.
- **Understand what an abstract data type is, and how ADTs relate to software design.**
- **Be familiar with standard data structures, including their implementations and relevant trade-offs.**

Where Are We?

From the First Day of Class: Course Overview — Topics

The following topics will be covered, *roughly* in order:

- Advanced C++
- Software Design Principles
- Recursion
- Searching
- Algorithmic Efficiency
- Sorting

DONE

- **Data Abstraction**

- **Basic Abstract Data Types & Data Structures:**

- **Smart Arrays & Strings**
- **Linked Lists**
- **Stacks & Queues**
- **Trees (various types)**
- **Priority Queues**
- **Tables**

Goal: Practical Generic Containers

A **container** is a data structure holding multiple items, usually all the same type.

A **generic** container is one that can hold objects of client-specified type.

- **Graph algorithms**

Where Are We? The Big Problem

For most of the rest of the semester, we will be addressing the following problem:

- We have a collection of data items, all of the same type, that we wish to store.
- We need to be able to access items [retrieve/find, traverse], add new items [insert] and eliminate items [delete].
- All this needs to be efficient in both time and space.

Solutions to this problem are called “**containers**”.

- There are many good ones.
- Which one we use depends on many factors, including what priority we place on the various requirements above.

We are particularly interested in **generic containers**: containers in which the client can specify the type of data to be stored.

Unit Overview

Sequences and Their Implementations

Next we begin a unit on data abstraction and data structures, with applications to Sequences and their implementations.

- Topics will include:
 - Data Abstraction
 - Sequence Data
 - Interfaces to Data
 - Data Structure Implementation
 - Exception Safety
 - Allocation & Efficiency
 - Generic Containers
 - Node-Based Structures
 - Linked Lists
 - Sequences in Practice
- Some material is in chapters 3 & 4 in the text. Material on exception safety is not in the text.

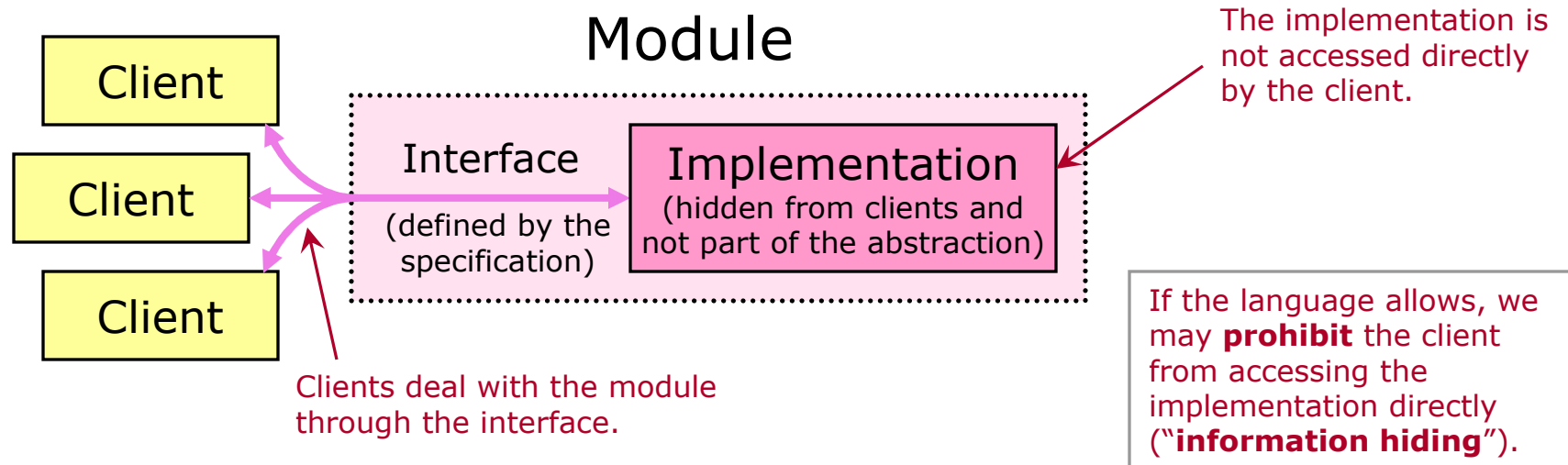
After this we will look at Stacks and Queues.

- Chapters 6 & 7.

Data Abstraction

What Is It?

Abstraction means separating the purpose of a module from its implementation.



We have been doing **functional abstraction**.

Next, we look at **data abstraction**.

- Data abstraction = applying the idea of abstraction to data.
- The primary concept we use is that of an **abstract data type**.

Data Abstraction

ADTs, Data Structures, and Classes [1/2]

An **abstract data type** (ADT) is:

- a **collection of data**, along with
- a **set of operations** on that data.
- ADTs are independent of implementation, and even of programming language.

A **data structure** is a construct within a programming language that stores a collection of data.

C++ and some other programming languages include **classes**, which facilitate object-oriented programming.

- Classes are is intended primarily for the implementation of data structures, each of which is often conceptually based on some ADT.
- However, one can implement data structures without using classes.

Data Abstraction

ADTs, Data Structures, and Classes [2/2]

Suppose we want to specify an ADT that holds exactly three pieces of information.

- We might call this ADT “Triple”.
- These are not assumed to be numeric or have any arithmetic properties at all. Rather, they are simply three pieces of data. Think of this as a list that always has size three.

What operations might such an ADT have? *The following were discussed in class.*

- *Get item*
- *Set item*
- *Create*
- *Destroy*
- *Check equality*
- *Reorder items*
- *Output*

These four form a complete, minimal interface.

Sequence Data

Introduction

A **Sequence** is a collection of items that are in some order.

- We will restrict our attention to **finite** Sequences, in which all items have the **same type**.
- It may help to think of an array here. However, there are other ways to store Sequences.

5	3	4	2	2	8	7	4	7	5	1	2
---	---	---	---	---	---	---	---	---	---	---	---

Questions

- What operations do we perform on Sequences?
- How can we implement a Sequence?
- How can we decide which implementation best fits any given circumstance?

Sequence Data

ADT Sequence — Definition

ADT **Sequence**

- **Data**
 - An ordered sequence of values, all the same type, indexed by 0, ..., size-1.
- **Operations**
 - **CreateEmpty**
 - Creates empty Sequence (with size 0, i.e., no data).
 - **CreateSized**
 - Given a size, create a Sequence with that size.
 - **Destroy**
 - Destroys a Sequence.
 - **Copy**
 - Make a copy of a given Sequence.
 - **LookUpByIndex**
 - Given a valid index, returns Sequence item in modifiable form.
 - **Size**
 - Returns size of Sequence.
 - **Empty**
 - Returns whether the Sequence is empty, that is, has size zero.
 - **Sort**
 - Sort a Sequence, using some given comparison function.
 - **Resize**
 - Changes size of Sequence. Data for indices 0, ..., $\min(\text{old size}, \text{new size})-1$ remains identical.
 - **InsertByIter**
 - Given an iterator (or pointer?) and an item, insert the item at the specified position.
 - **RemoveByIter**
 - Given an iterator, remove the item at that position.
 - **InsertBeg**
 - Given an item, insert it at the beginning.
 - **RemoveBeg**
 - Remove the first item.
 - **InsertEnd**
 - Like insertBeg, but at the end.
 - **RemoveEnd**
 - Like removeBeg, but at the end.
 - **Splice**
 - Move a contiguous subsequence from one Sequence to another.
 - **Traverse**
 - Performs some operation on every item in the Sequence, in order.
 - **Swap**
 - Swap the values of two given Sequences.

Sequence Data

ADT SortedSequence — Introduction

Our next ADT is SortedSequence.

- This is similar to Sorted List, in the text on pages 124-126.
- SortedSequence has many similarities with Sequence.
- The obvious difference is that the data are always kept sorted (that is, in order).

Sequence Data

ADT SortedSequence — Draft

ADT **SortedSequence**, Version 1

- Data
 - An ordered list of values, all the same type, indexed by 0, ..., size-1, **in ascending order**.
- Operations
 - CreateEmpty
 - Creates empty SortedSequence (with size 0, i.e., no data).
 - **CreateSized**
 - Given a size, create a SortedSequence with that size.
 - Destroy
 - Destroys a SortedSequence.
 - Copy
 - Make a copy of a given SortedSequence.
 - LookUpByIndex
 - Given a **valid index**, returns SortedSequence item in modifiable form.
 - Size
 - Returns size of SortedSequence.
 - Empty
 - Returns whether the SortedSequence is empty, that is, has size zero.
 - **Sort**
 - Sort a SortedSequence, using some given comparison function.

Iffy ...

Problems

- **Resize**
 - Changes size of SortedSequence. Data for indices 0, ..., min(old size, new size)-1 remains identical.
- **InsertByIter**
 - Given an iterator (or pointer?) and an item, insert the item at the specified position.
- RemoveByIter
 - Given an iterator, remove the item at that position.
- **InsertBeg**
 - Given an item, insert it at the beginning.
- RemoveBeg
 - Remove the first item.
- **InsertEnd**
 - Like insertBeg, but at the end.
- RemoveEnd
 - Like removeBeg, but at the end.
- **Splice**
 - Move a contiguous subsequence from one Sequence to another.
- Traverse
 - Performs some operation on every item in the SortedSequence, in order.
- Swap
 - Swap the values of two given SortedSequences.

Pointless or problematic

But if we get rid of the "problems",
how can we add new items?

Sequence Data

ADT SortedSequence — Improved

ADT **SortedSequence**, Version 2

- **Data**
 - An ordered list of values, all the same type, indexed by 0, ..., size-1, in ascending order, by some given comparison function.
- **Operations**
 - **CreateEmpty**
 - Creates empty SortedSequence (with size 0, that is, no data).
 - **Destroy**
 - Destroys a SortedSequence.
 - **Copy**
 - Make a copy of a given SortedSequence.
 - **LookUpByIndex**
 - Given a valid index, returns SortedSequence item in **non-modifiable** form.
 - **Size**
 - Returns size of SortedSequence.
 - **Empty**
 - Returns whether the SortedSequence is empty, that is, has size zero.
 - **InsertByValue**
 - Given an item, insert it.
 - **RemoveByValue**
 - Given a value, remove it.
 - **RemoveByIter**
 - Given an iterator, remove item at that position.
 - **Traverse**
 - Performs some operation on every item in the SortedSequence, in order.
 - **Swap**
 - Swap the values of two given SortedSequences.
 - **Find**
 - Given value, find item(s) with equivalent value, if any exist.

Sequence Data

ADT SortedSequence — What is it For?

In practice, the ordering of a SortedSequence is often not of primary importance. Rather, are interested in items being **easy to find**.

What can we do with this?

- First of all, we can store **“Set” data**. In a set, we only care **whether** an item is in the container, not **where** it is.

Now suppose we have a SortedSequence in which the items are pairs, and we specify a comparison function that, given two pairs, compares only the *first parts* of each. What is this good for?

- **Key-based look-up**.
 - The first part of each pair is the **key**.
- “Arrays” (sort of), where the thing between the brackets does not have to be a positive integer.
- That is, **Tables** (a.k.a. “dictionaries”, “associative arrays”, “maps”).

Sequence Data

ADT SortedSequence — P.O. vs. V.O.

We conclude that, despite the superficial similarity of Sequence and SortedSequence, there is a fundamental difference.

- Sequence deals with an item according to its **position** (that is, index) in the container.
- SortedSequence deals with an item primarily according to its **value**.

Two Types of ADTs

- Sequence is a **position-oriented** ADT.
- SortedSequence is a **value-oriented** ADT.

Now that we know what SortedSequence is “really” for, we can see that it is a bit inadequate as a value-oriented ADT.

- In practice, we do not care much about SortedSequence being a **Sequence**.
- Rather, we want to use it to manage “Set” or “Table” data.
- Maybe we can improve it if we break it away from its Sequence origins.
- Important Questions (to be examined later)
 - What do we really want from a value-oriented ADT?
 - How does one implement these in efficient ways?

Sequence Data

Availability of Sequence & SortedSequence

Data structures that (pretty nearly) implement the ADT Sequence are available in just about every modern computer language.

- C++ has `std::vector`, `std::deque`, `std::list`, and `std::basic_string`.
- And of course C++ arrays are a rudimentary form of Sequence.

Implementations of things like SortedSequence are less common.

- However, they are very often found **internally**.
- In implementations of the C++ standard library, an internal implementation of something like SortedSequence is usually used as the foundation for library classes `std::set`, `std::map`, `std::multiset`, and `std::multimap`.
- Languages like Python and Perl have the equivalent of the C++ `std::map` built-in (as "dict" and "hash", respectively). However, these are **not** implemented in terms of anything like SortedSequence .
 - Instead, a data structure called a Hash Table is used. We will discuss this later in the semester.

Interfaces to Data

Introduction

When we implement a data structure, we move from the abstract to the concrete.

The client needs to use the data structure. This is done through its **interface**.

We now look at some ideas related to the design of such interfaces.

Interfaces to Data

Characteristics of Good Interfaces

An interface should be **complete**.

- All relevant operations should be *possible*.
 - But what is “relevant”? Sometimes we place restrictions, as we will see with ADTs Stack and Queue.

We often strive for interfaces that are **minimal**.

- Avoid unnecessary functionality.

An interface should be **convenient**.

- Avoid making the interface a pain to use.

We want to facilitate **efficiency**.

- Allow the data to be dealt with efficiently.

We often want our interface to be **generic**.

- Avoid restricting possible implementations and internal data types.

- Some of the above goals can be at odds with each other. Designing a good, useful interface can be difficult.

These two often pull in opposite directions.

These two *can* pull in opposite directions.