Unit Overview
Algorithmic Efficiency & Sorting

Major Topics
- Introduction to Analysis of Algorithms
- Introduction to Sorting
- Comparison Sorts I
- More on Big-O
- The Limits of Sorting
- Divide-and-Conquer
- Comparison Sorts II
- Comparison Sorts III
- Radix Sort
- Sorting in the C++ STL
Where Are We?
From the First Day of Class: Course Overview — Goals

After taking this class, you should:

- Have experience writing and documenting high-quality code.
- Understand how to write robust code with proper error handling.
- Be able to perform basic analyses of algorithmic efficiency, including use of “big-O” notation.
- Be familiar with various standard algorithms, including those for searching and sorting.
- Understand what data abstraction is, and how it relates to software design.
- Be familiar with standard data structures, including their implementations and relevant trade-offs.

The rest of the semester
We will also discuss this in more detail
Where Are We?
From the First Day of Class: Course Overview — Topics

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

- Advanced C++
- Software Engineering Concepts
- Recursion
- Searching
- Algorithmic Efficiency
- Sorting
- Data Abstraction
- Basic Abstract Data Types & Data Structures:
  - Smart Arrays & Strings
  - Linked Lists
  - Stacks & Queues
  - Trees (various types)
  - Priority Queues
  - Tables
- Other, as time permits: graph algorithms, external methods.

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.
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 client code can specify the type of data to be stored.
Unit Overview
Handling Data & Sequences

We now begin a unit on handling data and implementing Sequence data.

Major Topics
- Data abstraction
- Introduction to Sequences
- Smart arrays
  - Array interface
  - Basic array implementation
  - Exception safety
  - Allocation & efficiency
  - Generic containers
- Linked Lists
  - Node-based structures
  - More on Linked Lists
- Sequences in the C++ STL
- Stacks
- Queues

Some material is in chapters 3, 4, 6, and 7 in the text. After this, we will look at trees.
Abstraction: Separate the purpose of a module from its implementation.

Recall: Function, class, or other unit of code. Generally smaller than a package.

We have been doing functional abstraction. Now we look at data abstraction.
Data Abstraction
What is It?

In **data abstraction**, we separate the various aspects of dealing with data, from the implementation of the data:

- The conceptual form of the data.
- The **operations** available on the data.
- The method used to access the data.

Important concepts

- **Abstract data type** (ADT).
- **Interface**.
Data Abstraction
ADTs, Data Structures, Classes

Abstract data type (ADT):
- a collection of data, along with
- a set of operations on that data.

ADTs are independent of implementation, and even of programming language.

Data structure: 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.
- An important use of classes is 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
ADT Example

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 mentioned in class.
  - Data access (get/set).
  - Check equality.
  - Reorder.
  - Create.
  - Destroy.
  - Output.

We might store the data for a Triple in an obvious data structure: three variables.

And we might implement all this using a class with three data members, and member functions implementing the various Triple operations.
When we implement a data structure, the idea of abstraction requires that we have a well defined **interface**.

Designing a good interface can be difficult. Here are some characteristics of a good interface.

An interface should be **complete**.
- All required operations should be possible.

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.
Introduction to Sequences
What is a Sequence?

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.

![Sequence example](image)

Questions
- What operations do we perform on Sequences?
- How can we implement a Sequence?
- How do we decide which implementation best fits any given circumstance?
Introduction to Sequences
ADT Sequence — Definition

**ADT Sequence**

- **Data**
  - An ordered sequence of values, all 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**
    - Exchange the values of two given Sequences.
It is common to keep Sequence data sorted. However, this changes the operations available.

- Operations that mess up the ordering are now disallowed.
- New operations, that make use of the ordering, become possible.

Therefore, we define another ADT, SortedSequence.

- Essentially, a SortedSequence is a Sequence in which the items are always kept sorted according to some comparison function.
- This is similar to Sorted List, in the text on pages 133–134.
Introduction to Sequences
ADT SortedSequence — Draft

ADT **SortedSequence** (draft)

- **Data**
  - An ordered list of values, all 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.
  - **Resize**
    - Changes size of SortedSequence. Data for indices 0, ..., \(\min(\text{old size, new size})-1\) 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**
    - Exchange the values of two given SortedSequences.

---

**Problems**

- Pointless or problematic
- Iffy ...

**But if we get rid of the “problems”, how can we add new items?**
Introduction to Sequences
ADT SortedSequence — Improved

**ADT SortedSequence** (final)

**Data**
- An ordered list of values, all 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**
  - Exchange the values of two given SortedSequences.
- **Find**
  - Given value, find item(s) with equivalent value, if any exist.
Introduction to Sequences
ADT SortedSequence — What is it For?

In practice, the ordering of a SortedSequence is often of little importance. Rather, we are interested in items being easy to find. What can we do with this?

- First, 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 whose items are pairs, and a comparison function that compares only the first parts of each pair. What is this good for?

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

```
(102, "John Smith") (388, "Mary Jones") (497, "Zyzzy Zyzz") (562, "Ig Ogg") (732, "Adam A.")
```
We conclude that, despite the similarities of Sequence and SortedSequence, there is a fundamental difference.

- Sequence handles an item according to its position (index) in the container.
- SortedSequence handles an item primarily according to its value.

Two Types of ADTs

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

SortedSequence is a bit inadequate as a value-oriented ADT.

- We often do not care about SortedSequence being a Sequence.
- Rather, we want to use it to store Set or Table data.
- Maybe we should 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?
We wish to implement a Sequence using a “smart” array.
- It should know its size, be able to copy itself, etc.
  - Just like in Assignment 2.
- It should also be able to *change* its size.
  - Recall that the ADT has resize and various insert/remove operations.

Basic Ideas
- Use a C++ class. An object of the class implements a single Sequence.
- Many (most? all?) of the ADT Sequence operations should be implemented using class member functions.
- Use iterators, operators, ctors, and the dctor in the usual ways.
- *Every* function in the interface should exist in order to implement, or somehow make possible, an ADT operation.
Array Interface
By ADT Operation

Use iterators to handle positions, traversing.

ADT Operations

- CreateEmpty
  - Default ctor.
- CreateSized
  - Ctor given size.
- Destroy
  - Dctor.
- Copy
  - Copy ctor & copy assignment.
- LookUpByIndex
  - Bracket operator.
- Size
  - Member function “size”.
- Empty
  - Member function “empty”.
- Sort
  - Handle externally, using iterators. Use iterator-returning member functions “begin” and “end”.
- Resize
  - Member function “resize”.
- InsertByIter, InsertBeg, InsertEnd
  - Member function “insert” does InsertByIter.
  - Use in conjunction with iterator-returning functions to do InsertBeg, InsertEnd.
- RemoveByIter, RemoveBeg, RemoveEnd.
  - As above, using “remove”.
- Splice
  - Call resize, then copy using op[].
- Traverse
  - Use iterator-returning member functions “begin” and “end”.
- Swap
  - Member function “swap”.

Array Interface Summary

Ctors & Dctor
- Default ctor
- Ctor given size
- Copy ctor
- Dctor

Member Operators
- Copy assignment
- Bracket

Global Operators
- None

Associated Global Functions
- None

Named Public Member Functions
- size
- empty
- begin
- end
- resize
- insert
- remove
- swap
Array Interface Details

For most of the member functions in our class, it is pretty obvious what the function prototype should look like. However, three of them are a little tricky:

- **insert**
  - Takes an iterator and an item.
  - Inserts the item just before the position referenced by the iterator.
  - Return value is an iterator to the inserted item.

- **remove**
  - Takes an iterator.
  - Removes the item referenced by the iterator.
  - Return value is an iterator to the item following the one removed.

- **swap**
  - Takes another Sequence, by reference.
  - Exchanges the values of this Sequence and the given one.
  - No return value.