EEECS 2011 M: Fundamentals of Data Structures

Suprakash Datta
Office: LAS 3043

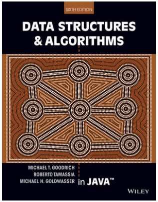
Course page: `http://www.eecs.yorku.ca/course/2011M`
Also on Moodle
Some slides are adapted from the authors' slides

Administrivia

- Lectures: Tue 6-9 pm (CLH C)
- Midterm (30%): in-class
- Final (50%): Scheduled by the registrar's office
- Homework (20%): 4 sets, mostly programming
- Office hours: Mon-Wed 3-4 pm or by appointment at LAS 3043.

Administrivia - 2

Textbook: Goodrich, M.T., Tamassia R. and Goldwasser, M.H. (2014). Data Structures and Algorithms in Java (6th ed.) John Wiley & Sons.



Getting instant feedback from you:

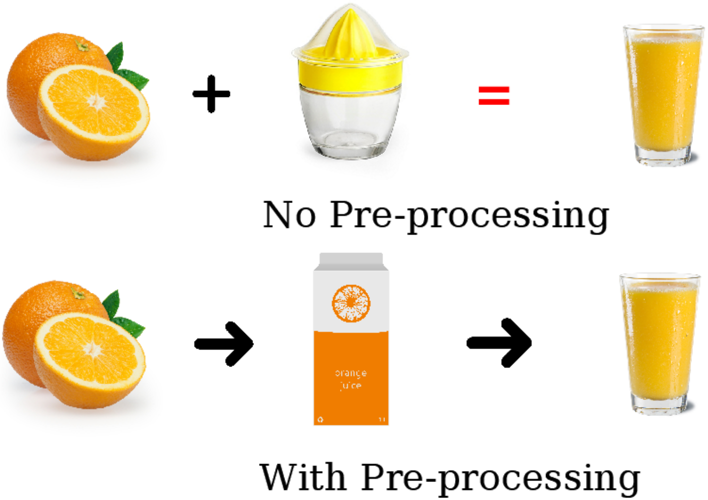
I would like to use iClicker.

Student manual: Click here

Homework, Grades

- We will be paperless, except midterm/final examination.
- All course information online – Moodle, partly mirrored in public course webpage
- All returned work will be in electronic form (including tests).

Why data structures?



Why data structures?

- 1021/2: Simple Java programming
- 2030: Object-oriented design, recursion, some simple algorithms
- 2011:
  - Organize data to answer queries efficiently
  - Maintain relationships between data items
  - Priority queues, trees, heaps, sets

An example: Representing a Set

Q: What operations do we need to support?

A: Addition, Deletion, Existence Query

1. An unordered array
   - Insertion - constant time
   - Deletion - linear time
   - Existence - linear time
2. A sorted array
   - Insertion - linear time
   - Deletion - linear time
   - Existence - logarithmic time

Representing a Set: Other ideas

1. A linked list
2. A binary search tree
3. A sorted array
4. Advanced ideas: Hashing, Union-Find with path compression

Example 2: A Priority Queue

Q: What operations do we need to support?

A: Insertion, Extract minimum, Change priority

1. A sorted array
   - Insertion - linear time
   - Extract min - constant time
   - Change priority - linear time
2. A linked list
   - Insertion - linear time
   - Extract min - constant time
   - Change priority - linear time
3. Advanced idea: A heap

Two levels of Abstraction

- Functional description, or programmer's abstract view
- Implementation: Details hidden from the user

Course Objectives

We will focus on two major goals:

- Fundamental data structures commonly used in the design of algorithms
  - know the classical data structures
  - master the use of abstraction, specification and program construction using modules
  - learn the basics of algorithm design for solving problems
- Precise and rigorous reasoning about programs
  - Correctness proofs with loop invariants
  - Running time analyses using asymptotic notation

Secondary Objective: abstracting out the algorithmic problem(s)

- Extract the algorithmic problem and ignore the "irrelevant" details
- Focuses your thinking, more efficient problem solving
- Programming contest problems teach this skill more effectively than many exercises in algorithms texts.

My expectations

- You will attend classes and labs/tutorials
- Want to solidify your programming and your algorithmic foundations
- Ask for help when needed
- Follow academic honesty regulations (see the class webpage for more details on policies).

Let me know when people are not understanding a topic.
I will also run anonymous iClicker polls regularly

To do well in this class

- Read the book, not just the slides
- Practice, practice, practice ...
- Ask for help early
- Follow along in class rather than take notes
- Keep up with the class. Ask questions in class or outside class
- Be timely – HW submitted late will not be graded

The Big Picture

- Programs = Data Structures + Algorithms
- Object-oriented programming (OOP): principled way of building programs
- Software Design: Building complex software using a design paradigm

The Big Picture - 2

- OOP with simple data: Objects consist of
    - Data items
    - Algorithms to construct, access and modify these items.
- OOP with complex data: Objects consist of
    - Data structures
    - Algorithms to construct, access and modify these structures.

Data Structures We Will Study

- Linear Data Structures: Arrays, Linked Lists, Stacks, Queues, Priority Queues
- Non-Linear Data Structures: Trees, Heaps, Hash Tables, Search Trees
- Graphs: Undirected Graphs, Directed Graphs, Directed Acyclic Graphs

Review of fundamentals

- Software Design
- Object-oriented programming (OOP)
- Java

Goals of Software Design

Software must be:

- Correct: Works correctly for all expected inputs
- Efficient
- Easy to read, understand and maintain
- Robust: Capable of handling unexpected inputs
- Reusable/Adaptable/Flexible
- Portable

Goals of OOP

- Robustness: handling unexpected inputs
- Adaptability: being able to evolve over time in response to changing environments
- Reusability: use as a component of different applications

OOP Design Principles

- Abstraction
- Encapsulation
- Modularity
- Hierarchical Organization

All of these are meant to manage complexity

Encapsulation

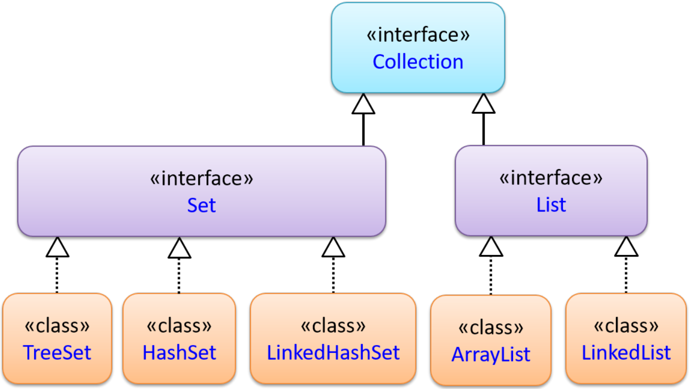
- Information hiding.
- objects reveal only what other objects need to see.
- Internal details are kept private.
- This allows the programmer to implement the object as they wish, as long as the requirements of the abstract interface are satisfied.

Modularity

- Complex software systems are hard to conceptualize, design and maintain.
- This is greatly facilitated by breaking the system up into distinct modules.
- Each module has a well-specified role.
- Modules communicate through well-specified interfaces.
- The primary unit for a module in Java is a package.

Hierarchical Design

Hierarchical class definitions allow efficient re-use of common software over different contexts.



Abstract Data Types

Exemplar of OOP design principles

- Abstraction is to distill a system to its most fundamental parts.

- Applying the abstraction paradigm to the design of data structures gives rise to abstract data types (ADTs).

Abstract Data Types

- An ADT is a model of a data structure that specifies
  - the type of data stored,
  - the operations supported on them, and
  - the types of parameters of the operations.

  The collective set of behaviors supported by an ADT is its public interface
- An ADT does NOT specify how it is implemented

Abstraction in Java

- A class serves as the primary means for abstraction in OOP.
- In Java, every variable is either a base type or is a reference to an object which is an instance of some class.
- Each class presents to the outside world a concise and consistent view of the objects that are its instances, without revealing too much unnecessary detail or giving others access to the inner workings of the objects.
- The class definition specifies its members (instance variables and methods)

Inheritance

A mechanism for modular and hierarchical organization.

- A (child) subclass extends a (parent) superclass.
- A subclass inherits (non-constructor) members of its superclass.
- A subclass can extend the superclass by providing brand-new data members and methods (besides those inherited from the superclass, other than constructors).

Java (unlike C++) is single inheritance

Interfaces

- The main structural element in Java that enforces an application programming interface (API) is an interface.
- An interface contains constants and abstract methods with no bodies; all public by default.
- It has no constructors and cannot be directly instantiated.
- A class that implements an interface, must implement all of the methods declared in the interface (no inheritance) to compile.

Abstract Classes

- An abstract class also cannot be instantiated, but it can define one or more methods that all implementations of the abstraction will have.
- Their sole purpose is to be extended.
- A class must be a subclass of an abstract class to extend it and implement all its abstract methods (or else be abstract itself).

Interfaces vs Abstract Classes

- A class that implements an interface, must implement **all** of the methods declared in the interface
- As a result, unlike abstract classes, interfaces are non-adaptable: you cannot add new methods to it without breaking its contract.
- However, interfaces offer great flexibility for its implementers: a class can implement any number of interfaces, regardless of where that class is in the class hierarchy.

Polymorphism

- Regardless of where it is in the inheritance tree, a class can implement several interfaces.
- This is multi-role playing (aka, mixin), not multiple inheritance.

Polymorphism

- Regardless of where it is in the inheritance tree, a class can implement several interfaces.
- This is multi-role playing (aka, mixin), not multiple inheritance.

To be continued...

We will continue with

- polymorphism,
- overloading,
- overriding,
- casting,
- generics, and
- exceptions

in the next lecture.