CSDS 233
Introduction to Data Structures

Erman Ayday

Welcome to CSDS 233

- An introductory CS course in computer science.
- Primary objective:
  - Introduction to various methods of organizing large amounts of data
  - Simple analysis of algorithms
  - Improvement of programming skills

- The instructor: Erman Ayday
  - Email: exa208@case.edu
  - Office: Nord
  - Class Hours: Tuesday, Thursday 1:00-2:15pm
  - Office Hours: By appointment

- Teaching assistants: See Canvas

Prerequisites

- ENGR 132

- Good knowledge of Java programming
  - Comfortable with object-oriented programming concepts
  - If your programming skills are not good, expect to put more sweat into the course
  - But: CSDS 233 is not about programming skills only, it is mainly about data structures and simple algorithm analysis

- Some mathematics background
  - Mostly simple algebra, understanding the basics of exponents and logarithms, etc.
  - Some simple calculations and growth rate of expressions
    - 1+2+3+…+1000=?
    - What grows faster, $\sqrt{N}$ or $\log N$?

Textbook

- Data Structures and Algorithm Analysis in Java (2nd or 3rd Edition), by Mark Allen Weiss, Addison-Wesley
- But lecture notes + Internet will do!
  - **If using Internet, beware of differences in formulations, terminology, and algorithm flavors**
- Note: Textbook is only supplementary and we are not going to follow it
  - You can read the corresponding sections for further information and practice

Course Requirements and Grading

- Attendance of lectures

- Assignments, 40%
  - 6 assignments
  - Each assignment is 50% written and 50% programming
  - Electronic submissions (Canvas)
  - Scan your drawings if needed
  - File name format: P1_YourCaseID_YourLastName.zip

- Submission of course evaluations, 1%

- Midterm exam, 25%

- Final exam, 35%

More on Assignments and Grading

- Assignments
  - Prepared by me and the TAs
  - Graded by the TAs
  - You will know who prepared and grades each assignment
  - Special office hours for particular assignment (given by the TAs that prepared the assignment)

- Everything will be curved (assignment, quizzes, and exams)

- No additional assignment for extra credit

Tentative Schedule

- Aug26 Course overview, basics of memory and OO programming
- Sep2 Basics of recursion, algorithm analysis - Assignment #1
- Sep9 Linked list
- Sep16 Stacks and queues; basics of trees - Assignment #2
- Sep23 Binary (search) trees; balanced trees
- Sep30 Huffman encoding; heaps/priority queue - Assignment #3
- Oct7 Heaps/priority queues
- Oct14 Midterm exam; exam review, basics of Hashing
- Oct21 – (no class on Oct22)
- Oct28 Hashing - Assignment #4
- Nov4 Basics of sorting, various sorting methods
- Nov11 Various sorting methods - Assignment #5
- Nov18 Basics of graphs; traversal
- Nov25 Shortest-path; Dijkstra’s algorithm - Assignment #6 (no class on Nov28)
- Dec2 Minimum spanning tree: Prim’s algorithm
- Final Exam – December 12 8am-11am

What's CSDS 233?

- We'll be studying fundamental *data structures*, as well as an introduction to *algorithms* that use these data structures to solve common problems correctly and efficiently.

- What is a data structure?

A way of organizing a collection of information or data

- Sequences: lists, stacks, and queues
- Trees
- Hash tables
- Graphs, etc.

- What is an algorithm?

A method to solve problems

- A procedure: takes an input and produces results
- Provides step-by-step "instructions" for solving a problem or accomplishing a task

Java Revisited

ADT, Encapsulation, and Generic Classes

Abstract Data Types

- Need an interface between…
  - …"common data structures" and high-level programming
  - …different high-level objects (modular development)

- An *abstract data type* (ADT) is the model of a data structure that specifies:
  - What operations can be performed on the data
  - But not how these operations are implemented

- To implement an ADT, we need to design data structures (to organize the data/information) and algorithms (to describe the procedures to complete desired tasks)
  - The objective of this course!

An Example: The Bag ADT

- As the name suggests, a bag is just a container for a group of data items.

- Some characteristics of a bag:
  - The positions of the data items don't matter (unlike a sorted list).
    - {3, 2, 10, 6} is equivalent to {2, 3, 6, 10}
  - The items do *not* need to be unique (unlike a set).
    - {7, 2, 10, 7, 5} is a bag but not a set

The Bag Operations

- Operations supported by the Bag ADT:
  - add(item): add item to the bag
  - remove(item): remove one occurrence of item (if any) from the bag
  - contains(item): check if item is in the bag
  - grab(): get an item at random, without removing it
  - numItems(): get the number of items in the bag

- The operations are provided to the programmer.
- … but NOT *how* the bag will be implemented.

The Bag Implementation

- Assumptions and design choices:
  - Bags can contain integers only.
    - We will consider a more general implementation later.
  - We will use an array to store the items.
    - Other design choices are possible (e.g., linked list).

Implementation of IntBag in Java

```
public class IntBag implements Bag {
    // instance variables (also known as fields, members, attributes, properties)
    private int[] items;  // items is a reference
    private int numItems;
    …
    // methods (also known as functions)
    public boolean add(int item) {
        if (numItems == items.length)
            return false; // no more room!
        else {
            items[numItems] = item;
            numItems++;
            return true;
        }
    }
    …
}
```

Accessing Private Fields

```
public class IntBag {
    private int[] items;
    private int numItems;
    …
    // methods
    public boolean add(int item) {
        if (numItems == items.length)
            return false; // no more
            room!
        else {
            items[numItems] = item;
            numItems++;
            return true;
        }
    }
    …
}
```

- Private fields are for the internal use by the implementation
  - Not exposed to ADT users.
  - Collectively form the data structures.
- A method can access a private field of its own object
- … or of *other* objects *from the same class*

```
public boolean addAll(IntBag other) {
    for (int i = 0; i < other.numItems; i++)
        add(other.items[i]);
}
```

Encapsulation Revisited

- Java uses *private* instance variables (and occasionally private helper methods) to hide the implementation of a class.
  - these private members can only be accessed inside methods that are part of the same class

```
class MyClass {
        …
        void myMethod() {
                IntBag b = new IntBag();
                b.items[0] = 17; // not allowed!
        …
```

b.add(17)

- Users are limited to the *public* methods of the class, as well as any public variables (usually limited to constants – why?).
  - public members can be accessed by methods of *any* class

Benefits of Encapsulation

- It prevents inappropriate changes to the state of an object:

```
class MyClass {
    …
    void myMethod() {
        IntBag b = new IntBag();
        b.addItem(7);
        b.addItem(22);
        b.numItems = 0; // not allowed
    …
```

- Make sure to use proper encapsulation in the classes that you write for this course!

More Bags, … and Genericity

- A bag of candy, a bag of apples, a bag of baseballs, … a bag of integers, a bag of floating-point numbers, …
  - A bag of objects

- Code reuse for different bags (various types of objects), rather than recode the same (or almost identical) logic (e.g. same algorithms) for different types

- Type-independent data structures and algorithms can be used more widely

- Accomplished through **Java Generics**

Using a Superclass to Implement Generic Classes

```
public class Bag {
    // instance variables (also known as
       fields, members, attributes,
       properties)
    private Object[] items;  // items is a
       reference
    private int numItems;
    …
    // methods
    public boolean add(Object item) {
        if (numItems == items.length)
            return false; // no more room!
        else {
            items[numItems] = item;
            numItems++;
            return true;
        }
    }
    …
}
```

```
public Object grabItem() {
    if (numItems == 0)
        return false; // nothing there
    else {
        return items[0];
    }
}
…
```

Using the Generic Class

- Type downcast for access generic class objects

```
public class Test1
{
   public static void main( String [ ] args )
   {
      Bag m = new Bag( );

      m.add( “37C” );
      String bodyTemp = (String) m.grabItem( );
      System.out.println( ”Temperature is: " + bodytemp );
   }
}
```

- The “add” method passes a string, so the actual object is String.
- The “grabItem” method cannot tell by itself what should be the return type.
  - A typecast is necessary!

Using the Generic Class (cont.)

- No restrictions on object types in a Bag

```
public class Test2
{
   public static void main( String [ ] args )
   {
      Bag m = new Bag( );

      m.add( "37C" );
      m.add(new Integer(96)); // Wrapper class!
      Integer temp = (Integer) m.grabItem( ); // Run-time error!
   }
}
```

Overview of the Semester – Toy Example

- Implement a phonebook
- Names and phone numbers

- Data structure?
- Operations?
- Instance variables?
- Efficiency of operations?

Example: Searching in a phonebook

- Data structure: phonebook representation
  - an array of 1,000,000 names (names[0..999999]) and an array of corresponding telephone numbers (phones[0..999999])
- Algorithm:
  - the procedure to find the telephone number by name

```
findNumber(person) {
    P = 0
    while (P < 1000000) {
      // Compare the P-th person in the array and person
      if (person.equals(names[P]))
                return phones[P]
      else
                P++
    }
    return NOT_FOUND
}
```

- How many iterations are required, on average?

Example cont'd: Same problem, different data structure

- Different data structure: a sorted array by name alphabetically, names[0..999999], and the corresponding array of numbers, phones[0..999999]

```
findNumber(person) {
      low = 0
      high = 999999

      while (low <= high) {
          P = floor((low + high) / 2)
          if (person.equal(names[P]))
               return phones[P]
           else
               if (person < names[P])
                      high = P – 1
                else
                      low = P + 1
       }
       return NOT_FOUND
}
```

- How many iterations are required, on average?

Example cont’d: So What to Use?

- Is a sorted array always better?

- We may need to efficiently perform different operations:
  - search for a telephone number
  - insert a new telephone number
  - remove a telephone number

- Some data structures provide better performance than others for a particular application.

- More generally, we’ll learn how to characterize and compare the efficiency of different data structures and their associated algorithms to perform different operations (for different applications).
  - Time efficiency: how many steps are needed?
  - Space efficiency: how much storage is needed?