Abstract Data Types and Data Structures

Computer Science S-111
Harvard University
David G. Sullivan, Ph.D.

Congrats on completing the first half!

- In the second half, we will study fundamental *data structures*.
  - ways of imposing order on a collection of information
  - sequences: lists, stacks, and queues
  - trees
  - hash tables
  - graphs

- We will also:
  - study *algorithms* related to these data structures
  - learn how to *compare* data structures & algorithms

- Goals:
  - learn to think more intelligently about programming problems
  - acquire a set of useful tools and techniques
Sample Problem I: Finding Shortest Paths

- Given a set of routes between pairs of cities, determine the shortest path from city A to city B.

Sample Problem II: A Data "Dictionary"

- Given a large collection of data, how can we arrange it so that we can efficiently:
  - add a new item
  - search for an existing item

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

- More generally, we'll learn how to characterize the efficiency of different data structures and their associated algorithms.
Example of Comparing Algorithms

• Consider the problem of finding a phone number in a phonebook.

• Let’s informally compare the time efficiency of two algorithms for this problem.

Algorithm 1 for Finding a Phone Number

```java
findNumber(person) {
    for (p = number of first page; p <= number of the last page; p++) {
        if person is found on page p {
            return the person’s phone number
        }
    }
    return NOT_FOUND
}
```

• If there were 1,000 pages in the phonebook, how many pages would this look at in the worst case?
• What if there were 1,000,000 pages?
Algorithm 2 for Finding a Phone Number

```java
findNumber(person) {
    min = the number of the first page
    max = the number of the last page
    while (min <= max) {
        mid = (min + max) / 2      // page number of the middle page
        if person is found on page mid {
            return the person's number
        } else if the person's name comes earlier in the book {
            max = mid - 1
        } else {
            min = mid + 1
        }
    }
    return NOT_FOUND
}
```

- If there were 1,000 pages in the phonebook, how many pages would this look at in the worst case?
- What if there were 1,000,000 pages?

Searching a Collection of Data

- The phonebook problem is one example of a common task: searching for an item in a collection of data.
  - another example: searching for a record in a database
  - Algorithm 1 is known as *sequential search*.
    - also called *linear search*
  - Algorithm 2 is known as *binary search*.
    - only works if the items in the data collection are sorted
Abstract Data Types

• An abstract data type (ADT) is a model of a data structure that specifies:
  • the characteristics of the collection of data
  • the operations that can be performed on the collection

• It’s abstract because it doesn’t specify how the ADT will be implemented.
  • does not commit to any low-level details

• A given ADT can have multiple implementations.

A Simple ADT: A Bag

• A bag is just a container for a group of data items.
  • analogy: a bag of candy

• The positions of the data items don’t matter (unlike a 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} isn’t a set, but it is a bag
A Simple ADT: A Bag (cont.)

• The operations we want a Bag to support:
  • 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
  • numItems(): get the number of items in the Bag
  • grab(): get an item at random, without removing it
    • reflects the fact that the items don’t have a position (and thus we can’t say "get the $5^{th}$ item in the Bag")
  • toArray(): get an array containing the current contents of the bag

• We want the bag to be able to store objects of any type.

Specifying an ADT Using an Interface

• In Java, we can use an interface to specify an ADT:

```java
public interface Bag {
    boolean add(Object item);
    boolean remove(Object item);
    boolean contains(Object item);
    int numItems();
    Object grab();
    Object[] toArray();
}
```

• An interface specifies a set of methods.
  • includes only the method headers
  • does not typically include the full method definitions

• Like a class, it must go in a file with an appropriate name.
  • in this case: Bag.java
Implementing an ADT Using a Class

• To implement an ADT, we define a class:
  ```java
  public class ArrayBag implements Bag {
      ...
      public boolean add(Object item) {
          ...
      }
  }
  ```

• When a class header includes an implements clause, the class must define all of the methods in the interface.
  • if the class doesn't define them, it won't compile

All Interface Methods Are Public

• Methods specified in an interface must be public, so we don't use the keyword public in the definition:
  ```java
  public interface Bag {
      boolean add(Object item);
      boolean remove(Object item);
      boolean contains(Object item);
      int numItems();
      Object grab();
      Object[] toArray();
  }
  ```

• However, when we actually implement the methods in a class, we do need to use public:
  ```java
  public class ArrayBag implements Bag {
      ...
      public boolean add(Object item) {
          ...
      }
  }
  ```
One Possible Bag Implementation

• One way to store the items in the bag is to use an array:
  ```java
  public class ArrayBag implements Bag {
      private ______________[] items;
      ...
  }
  ```

• What type should the array be?

• This allows us to store any type of object in the items array, thanks to the power of polymorphism:
  ```java
  ArrayBag bag = new ArrayBag();
  bag.add("hello");
  bag.add(new Rectangle(20, 30));
  ```

• How could we keep track of how many items are in a bag?

Another Example of Polymorphism

• An interface name can be used as the type of a variable:
  ```java
  Bag b;
  ```

• Variables with an interface type can refer to objects of any class that implements the interface:
  ```java
  Bag b = new ArrayBag();
  ```

• Using the interface as the type allows us to write code that works with any implementation of an ADT:
  ```java
  public void processBag(Bag b) {
      for (int i = 0; i < b.numItems(); i++) {
          ...
      }
  }
  ```

• the param can be an instance of any Bag implementation
• we must use method calls to access the object's internals, because the fields are not part of the interface
Memory Management: Looking Under the Hood

- To understand how data structures are implemented, you need to understand how memory is managed.

- There are three main types of memory allocation in Java.

- They correspond to three different regions of memory.

Memory Management, Type I: Static Storage

- Static storage is used for **class variables**, which are declared **outside any method** using the keyword **static**:

  ```java
  public class MyMethods {
      public static int numCompares;
      public static final double PI = 3.14159;
  }
  ```

- There is only one copy of each class variable.
  - shared by all objects of the class
  - Java's version of a global variable

- The Java runtime allocates memory for class variables when the class is first encountered.
  - this memory stays fixed for the duration of the program
Memory Management, Type II: Stack Storage

- Method parameters and local variables are stored in a region of memory known as the stack.

- For each method call, a new stack frame is added to the top of the stack.

```java
public class Foo {
    public static int x(int i) {
        int j = i - 2;
        if (i >= 6) {
            return i;
        }
        return x(i + j);
    }
    public static void main(String[] args) {
        System.out.println(x(5));
    }
}
```

- When a method completes, its stack frame is removed.

Memory Management, Type III: Heap Storage

- Objects are stored in a memory region known as the heap.

- Memory on the heap is allocated using the new operator:

```java
int[] values = new int[3];
ArrayBag b = new ArrayBag();
```

- new returns the memory address of the start of the object on the heap.
  - a reference!

- An object stays on the heap until there are no remaining references to it.

- Unused objects are automatically reclaimed by a process known as garbage collection.
  - makes their memory available for other objects
Two Constructors for the ArrayBag Class

```java
public class ArrayBag implements Bag {
    private Object[] items;
    private int numItems;
    public static final int DEFAULT_MAX_SIZE = 50;

    public ArrayBag() {
        this.items = new Object[DEFAULT_MAX_SIZE];
        this.numItems = 0;
    }

    public ArrayBag(int maxSize) {
        if (maxSize <= 0) {
            throw new IllegalArgumentException(
                "maxSize must be > 0");
        }
        this.items = new Object[maxSize];
        this.numItems = 0;
    }

    ...  
}
```

• As we've seen before, we can have multiple constructors.
  • the parameters must differ in some way

• The first one is useful for small bags.
  • creates an array with room for 50 items.

• The second one allows the client to specify the max # of items.

• If the user inputs an invalid maxSize, we throw an exception.
Example: Creating Two ArrayBag Objects

```java
// client
public static void main(String[] args) {
    ArrayBag b1 = new ArrayBag(2);
    ArrayBag b2 = new ArrayBag(4);
    ...
}
```

```java
// constructor
public ArrayBag(int maxSize) {
    ... // error-checking
    this.items = new Object[maxSize];
    this.numItems = 0;
}
```

• After the objects have been created, here’s what we have:

```
stack
b2
b1
args

heap
items
numItems
0
```

```
stack
b2
b1
args

heap
items
numItems
0
null
null
null
null
null
null
null
```
Adding Items

• We fill the array from left to right. Here’s an empty bag:

\[
\begin{array}{c|c|c|c|c}
\text{items} & \\
\hline
\text{numItems} & 0 \\
\end{array}
\]

• After adding the first item:

\[
\begin{array}{c|c|c|c|c}
\text{items} & \\
\hline
\text{numItems} & 1 \\
\end{array}
\]

“hello, world"

• After adding the second item:

\[
\begin{array}{c|c|c|c|c}
\text{items} & \\
\hline
\text{numItems} & 2 \\
\end{array}
\]

“hello, world”  “howdy”

Adding Items (cont.)

• After adding the third item:

\[
\begin{array}{c|c|c|c|c}
\text{items} & \\
\hline
\text{numItems} & 3 \\
\end{array}
\]

“hello, world”  “howdy”  “bye”

• After adding the fourth item:

\[
\begin{array}{c|c|c|c|c}
\text{items} & \\
\hline
\text{numItems} & 4 \\
\end{array}
\]

“hello, world”  “howdy”  “bye”  “see ya!”

• At this point, the ArrayBag is full!
  • it’s non-trivial to “grow” an array, so we don’t!
  • additional items cannot be added until one is removed
A Method for Adding an Item to a Bag

```java
public class ArrayBag implements Bag {
    private Object[] items;
    private int numItems;
    ...
    public boolean add(Object item) {
        if (item == null) {
            throw new IllegalArgumentException("no nulls");
        } else if (this.numItems == this.items.length) {
            return false; // no more room!
        } else {
            this.items[this.numItems] = item;
            this.numItems++;
            return true; // success!
        }
    }
    ...
}
```

• Initially, this.numItems is 0, so the first item goes in position 0.
• We increase this.numItems because we now have 1 more item.
• and so the next item added will go in the correct position!
Example: Adding an Item

```java
public static void main(String[] args) {
    String message = "hello, world";
    ArrayBag b = new ArrayBag(4);
    b.add(message);
}
```

```java
public boolean add(Object item) {
    else {
        this.items[this.numItems] = item;
        this.numItems++;
        return true;
    }
}
```

- add's stack frame includes:
  - item, which stores...
  - this, which stores...
• After the method call returns, add's stack frame is removed from the stack.
Extra Practice: Determining if a Bag Contains an Item

Let’s write the `ArrayBag contains()` method together.
- should return `true` if an object equal to `item` is found, and `false` otherwise.

```java
__________ contains(__________ item) {
}
```

- "hello, world"
- "oh my!"
- "what's in the bag?"

```
items: __________
numItems: 3
```

null null null null null null
Would this work instead?

- Let's write the ArrayBag contains() method together.
  - should return true if an object equal to item is found, and false otherwise.

```java
public boolean contains(Object item) {
    for (int i = 0; i < this.items.length; i++) {
        if (this.items[i].equals(item)) { // not ==
            return true;
        }
    }
    return false;
}
```

Another Incorrect contains() Method

```java
public boolean contains(Object item) {
    for (int i = 0; i < this.numItems; i++) {
        if (this.items[i].equals(item)) {
            return true;
        } else {
            return false;
        }
    }
    return false;
}
```

- What's the problem with this?
A Method That Takes a Bag as a Parameter

```java
public boolean containsAll(Bag otherBag) {
    if (otherBag == null || otherBag.numItems() == 0) {
        return false;
    }
    Object[] otherItems = otherBag.toArray();
    for (int i = 0; i < otherItems.length; i++) {
        if (!this.contains(otherItems[i])) {
            return false;
        }
    }
    return true;
}
```

• We use `Bag` instead of `ArrayBag` as the type of the parameter.
  • allows this method to be part of the `Bag` interface
  • allows us to pass in any object that implements `Bag`

• We must use methods in the interface to manipulate `otherBag`.
  • we can't use the fields, because they're not in the interface

A Type Mismatch

• Here are the headers of two `ArrayBag` methods:
  ```java
  public boolean add(Object item)
  public Object grab()
  ```

• Polymorphism allows us to pass `String` objects into `add()`:
  ```java
  ArrayBag stringBag = new ArrayBag();
  stringBag.add("hello");
  stringBag.add("world");
  ```

• However, this will not work:
  ```java
  String str = stringBag.grab();   // compiler error
  ```
  • the return type of `grab()` is `Object`
  • `Object` isn't a subclass of `String`, so polymorphism doesn't help!

• Instead, we need to use a type cast:
  ```java
  String str = (String)stringBag.grab();
  ```
  • this cast doesn't actually change the value being assigned
  • it just reassures the compiler that the assignment is okay