CSE 12: Basic data structures and object-oriented design

Jacob Whitehill
jake@mplab.ucsd.edu

Lecture Two
2 Aug 2011
Review from last lecture

- In computer science, all data must ultimately be represented as a binary sequence.

- Data structures are necessary to encode useful information in binary sequences.

- Data structures may vary in their time complexity, space complexity, and “code complexity” (human effort).
Review from last lecture

• It is important to learn the fundamental data structures of computer science so you don’t keep having to “rediscover the wheel”.

• The fundamental data structures covered in this course include: lists, stacks, queues, heaps, trees, hash tables, and graphs.
Fundamental data structures

• 5 of these structures (list, stack, queue, heap, hash table) are useful as collections to support add/retrieve/remove operations.

• In coarse English, a collection is useful if the user wants to “put data in it”, and later “pull data out of it.”

• E.g., you’re writing a program to manage the financial aid of all UCSD students. You want “some structure” (collection) to hold all the UCSDStudent objects -- you don’t want to manage them yourself.
Fundamental data structures

• Different collections have different time and space costs for the add/retrieve/remove operations.

• Which collection is best depends on which operations your code calls most often.
Fundamental data structures

- 2 of these structures (tree, graph) are useful to represent connectivity relationships among data:
  - Trees can represent hierarchical relationships (e.g., heredity).
  - Graphs can represent arbitrary relationships between pairs of data (e.g., Facebook friends).
Fundamental data structures

• In this course we will develop all of these data structures as Abstract Data Types (ADTs).

• In this lecture I hope to:
  • Explain abstraction from a computer system’s perspective.
  • Motivate building data structures as ADTs.
  • Introduce our first ADT of the course: the list.
Data structures you’re already familiar with.
Data structures you already know

• In prior coursework you have already worked with some simple data structures:

• **Arrays:**

```java
int[] numbers = new int[100];
...
numbers[5] = 16;
```
Data structures you already know

- In prior coursework you have already worked with some simple data structures:
  - **Arrays**: collection of related variables specified by an index:

    ```java
    int[] numbers = new int[100];
    ...
    numbers[5] = 16;
    ```

    More convenient than declaring 100 variables!
    ```java
    int number1, number2, number3, ... number100;
    ```
Data structures you already know

- **Strings:**

  ```java
  String firstName = "Jimmy";
  String lastName = "Carter";
  String fullName = firstName + " " + lastName;
  System.out.println("Hello, " + fullName);
  ```
Data structures you already know

- **Strings**: a finite sequence of characters:

```java
String firstName = "Jimmy";
String lastName = "Carter";
String fullName = firstName + " " + lastName;
System.out.println("Hello, " + fullName);
```

String data structure allows you to “add” strings together.
Data structures you already know

- In other languages (e.g., C), a string is simply an array of characters:

  ```c
  char str1[32] = "angry"; // str of max len 32
  char str2[32] = "bird";
  ```

- You can’t concatenate two strings simply by “adding” them:

  ```c
  char str3[64] = str1 + str2; // This is incorrect
  ```
Data structures you already know

• Records:

class Customer {
    String _name;
    int _age;
    float _accountBalance;
}


Data structures you already know

• **Records**: a group of related variables:

```java
class Customer {
    String _name;
    int _age;
    float _accountBalance;
}
```

Records alleviate the burden of maintaining “whose name goes with whose age and whose balance?”
Data structures you already know

• Simple data structures like arrays, strings, and records provide conveniences to the programmer.

• However, these structures are not physically present anywhere in the computer.

• They are *not real*; they are *abstract*.
  
  Merriam-Webster: existing in thought or as an idea but not having a physical or concrete existence

• In contrast, **bits** (0/1) are physically present -- they encode whether a particular transistor is on/off.
Abstraction for convenience.
Memory abstraction

- Even the “one long sequence of 1’s and 0’s” from last lecture is abstract:
  - Computers may contain multiple memory chips.
  - In physical reality, there are multiple binary sequences -- one for each memory chip.

2GB

2GB

2GB
If we want to write to or read from a particular byte of memory, we must specify both which chip (A, B, or C) and which location on that chip (anywhere from 0 to 2,147,483,647).
Foray into computer architecture

- Somewhere between your source code and the memory chips, the determination of “which memory chip” must be made...

Source code (e.g., `x = y + 5;`)

```
x = y + 5;
```

Compiler

Machine instructions

Memory bus

CPU

Memory controller

Memory chips
Memory abstraction

• The memory controller provides a “convenient illusion”:
  • It allows the CPU, compiler, and ultimately our Java code to “pretend” there’s only one large bank of memory of size 6GB.
  • No need to specify “memory chip A, B, or C”.
  • Just specify the byte location you’re interested in (anywhere from 0 to 6442450943).
  • This illusion is called an “abstraction”.
Memory abstraction

- Memory controller must “translate” between “abstract” requests of the CPU and “reality” of multiple memory chips.

Request from CPU:
“Store 123 into memory location 2214592512”.

Hmm, 2214592512 is on chip B.

“Store 123 into location...”
Memory abstraction

- Thanks to this “memory abstraction”, the CPU, operating system, Java compiler, and ultimately you—the-programmer don’t have to worry about which memory chip your variables are stored.

Source code (e.g., x = y + 5 ;)

Compiler

Machine instructions

Abstraction: pretend we have just one “bank” of memory.

Reality: we have multiple memory chips.

Memory controller

Memory bus

CPU

Memory chips
Memory addresses

- The memory controller provides us with the “abstraction” of viewing memory as one, long sequence of bytes (8 bits each).

- Each location in the memory bank is called an address.

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11110110</td>
</tr>
<tr>
<td></td>
<td>11001001</td>
</tr>
<tr>
<td></td>
<td>01010001</td>
</tr>
<tr>
<td></td>
<td>01011000</td>
</tr>
<tr>
<td>4</td>
<td>11101000</td>
</tr>
<tr>
<td></td>
<td>11100000</td>
</tr>
<tr>
<td></td>
<td>01000100</td>
</tr>
<tr>
<td></td>
<td>11011110</td>
</tr>
<tr>
<td>8</td>
<td>01100101</td>
</tr>
<tr>
<td></td>
<td>00101001</td>
</tr>
<tr>
<td></td>
<td>01101111</td>
</tr>
<tr>
<td></td>
<td>00010111</td>
</tr>
<tr>
<td>3221225468</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>10101111</td>
</tr>
<tr>
<td></td>
<td>01101110</td>
</tr>
<tr>
<td></td>
<td>01011111</td>
</tr>
</tbody>
</table>
Memory controller implements **OneLongBinarySequence** abstraction

- The memory controller is responsible for *implementing* this abstraction.
- The memory controller must handle requests/messages from the CPU and respond to them appropriately.
- Example requests:
  - “Store value 123 into address 2152420584.”
  - “Fetch the value stored at address 2152420584.”
OneLongBinarySequence abstraction

• We can conceive of this OneLongBinarySequence as an “abstract object.”

• The abstract object responds to messages/requests sent to it.

• In Java, you send a message to an object by calling one of its methods.

Message: storeValue(123, 216523416)
class OneLongBinarySequence {
    void storeValue (int address, int value) {
        // Determine which memory card "address" is on
        // Determine the address *within* that card
        // Write "value" to the correct address on the
        // correct card.
    }

    int fetchValue (int address) {
        // ...
    }
}
Programming language abstractions

• In this course, we will deal with abstractions primarily at the **programming language** and **data structure** level.

• Programming languages allow us to refer to data using meaningful variable names, e.g.,

```cpp
int imageWidth;
```

instead of referring to particular memory addresses, e.g., `4938248`. 
Programming language abstractions

- Example:

```c
void addNumbers () {
    int num1 = 13, num2 = 27;
    int num3 = num1 + num2;
}
```

The compiler/interpreter implements the abstraction, i.e., translates between variable names and memory addresses.
Point to emphasize

• Abstraction can be very *convenient*:
  • The OneLongBinarySequence is more convenient than having to know which memory card a particular byte is stored.
  • A variable name is easier to remember than an integer memory address.
Abstraction to hide details.
Data structure abstractions

• In this course, we will study some of the fundamental data structures of computer science: list, stack, queue, heap, tree, hash table, and graph.

• Each of these provides a convenient abstraction to the programmer.

• We implement these data structures as abstract data types (ADTs).
Abstract data types

• An abstract data type (ADT) provides the programmer with a convenient “container” for storing data.

• For instance, a list is an abstraction for a container of ordered elements that can grow as we add more elements to it.

• Typically, ADTs are implemented within a library of code (e.g., Java RE).

• The programmer interacts with the ADT by calling various methods on it.
Abstract data types

• The details of how the methods are implemented are generally not visible to the “user”.

• The “user” is the programmer who wants to use the ADT to manage his/her data.

• The user doesn’t necessarily care how the ADT is implemented, as long as the methods work according to the interface specification.

• This allows flexibility in the implementation of the ADT.
ADT example

• This discussion of abstract data types may be getting “abstract”.

• Let’s concretify things by introducing one of the classics: a list.
Lists

- Sometimes you need to manage a collection of variables:
  - Students enrolled at UCSD.
  - Customers who buy stuff from your company.
  - List of programs currently running on your machine.
Lists

• So...just use an array:

    Student[] ucsdStudents = new Student[50];
Linked lists

• But what if the number of students is not known ahead of time?

• We could just allocate a really big array with room to spare.

    ucsdStudents = new Student[1000];
Why not use an array?

• There are two problems with this:
  • It is wasteful -- many elements of `ucsdStudents` will never be used.
  • If we try to allocate too big an array, then the initialization may fail, due to:
    • Lack of free memory; or
    • Lack of contiguous free memory (i.e., available in one big block).
Why not use an array?

- Ok, fine -- start out with a small array, and make it bigger when it's full.
- But it's annoying for the programmer to have to keep "enlarging the array".
- What we want is an object that manages the array for us.
- We don't really care how it's done, as long as it works.
  - We're not concerned with the details.
What we want

• What we want is some data structure that has the following capabilities:

• We can add elements (e.g., Students) to it, and it will store them.

• The data structure should automatically “grow” itself as needed in an “efficient” manner (much more later).

• It should not use memory wastefully.
What we want

• We can retrieve a particular element specified by index $i$.

• We can remove a particular element specified by index $i$. 
• Here’s a Java specification of what we want:

```java
class List {
    
    // Adds the specified element to the end of the list.
    // Takes O(1) time.
    void add (Object element) { ... } 

    // Returns the element contained in the list at index i if it exists. Else, throws NoSuchElementException.
    Object get (int i) throws NoSuchElementException { ... } 

    // Removes the element contained in the list at index i if it exists. Else, throws NoSuchElementException.
    void remove (int i) throws NoSuchElementException { ... }

    
}
```

For now, just take this to mean “quickly”.
List interface specification

• Notice the things we **don’t** care about:

class List {

  // Adds the specified element to the end of the list.
  // Takes O(1) time.
  void add (Object element) { ... }  

  // Returns the element contained in the list at index
  // i if it exists. Else, throws NoSuchElementException.
  Object get (int i) throws NoSuchElementException { ... }

  // Removes the element contained in the list at index
  // i if it exists. Else, throws NoSuchElementException.
  void remove (int i) throws NoSuchElementException { ... }

}
List interface specification

- Notice the things we **do** care about:

```java
class List {
    // We care about what the methods **return**.
    // Adds the specified element to the end of the list.
    // Takes O(1) time.
    void add (Object element) { ... }

    // Returns the element contained in the list at index
    // i if it exists. Else, throws NoSuchElementException.
    Object get (int i) throws NoSuchElementException { ... }

    // Removes the element contained in the list at index
    // i if it exists. Else, throws NoSuchElementException
    void remove (int i) throws NoSuchElementException { ... }
}
```

We care what the methods do. We care about what exceptions it might throw (more later).
List specification

- A description of methods...
  - What the methods do.
  - What parameters they take.
  - What they return.
  - What exceptions they might throw.
- ...is known as an interface.
- An interface in Java contains:
  - No instance variables.
  - No method bodies.
List interface

An interface consists of method signatures.

```java
interface List {
    // Adds the specified element to the end of the list.
    // Takes O(1) time.
    void add (Object element);

    // Returns the element contained in the list at index
    // i if it exists. Else, throws NoSuchElementException.
    Object get (int i) throws NoSuchElementException;

    // Removes the element contained in the list at index
    // i if it exists. Else, throws NoSuchElementException
    void remove (int i) throws NoSuchElementException;
}
```

A method signature consists of the method name, parameters, return type, and exceptions thrown.
Using a list

• Before we can use a List object, we first need some class that implements the List interface.

• The List is an abstraction -- we can’t create List objects by writing:

  ```java
  List list = new List(); // Won’t compile
  ```

• The reason is that List is just a description of what a list should do -- not how it would actually work.
Implementing the `List` interface

- In order to create an instance of `List`, you must first create a (concrete) `class` that `implements` the (abstract) `List` interface.

- What does this mean?
  - It means that we must implement the `body` of every method whose signature was defined in the interface.
Implementing the (concrete) List interface

class ListImpl implements List {
    private Object[] _array;
    private int _numElements;

    void add (Object element) {
        ...  
        _array[_numElements++] = element;
    }

    Object get (int i) throws NoSuchElementException {
        ...
    }

    void remove (int i) throws NoSuchElementException {
        ...
    }
}

Tell the compiler explicitly that ListImpl implements the List interface.
Creating a List

• Now that we (hypothetically) have a ListImpl implementation of List, we can create a List object:

```java
List list = new ListImpl(); // ok!
list.add(new Student("Bertha", 18));
...
```
Abstraction for good software design.
Why separate interface from implementation?

- So far, creating a `List` interface and a `ListImpl` implementation hasn’t bought us very much.

- Why is it useful?
Why separate interface from implementation?

1. Separating interface from implementation facilitates a *division of labor* among members of a software development team.

User: I’ll work on the graphical front-end to manage a list of UCSD students.

Implementor: Fabulous. I’ll create the List implementation itself.

Photos courtesy of Google Image Search.
Why separate interface from implementation?

1. Separating interface from implementation facilitates a division of labor among members of a software development team.

1. Both the implementors and users of the ADT agree on the interface.

2. The implementor implements the interface (writes the ADT method bodies).

3. The user calls the interface methods.

Photos courtesy of Google Image Search.
Why separate interface from implementation?

List interface

User

writes:

List list = new ListImpl();
list.add(new Student());
...

Implementor

writes:

class ListImpl implements List {
...

    void add (Object o) {
        _array[_numElements++] = o;
    }
}

Photos courtesy of Google Image Search.
Why separate interface from implementation?

2. Programming an application that uses objects of an interface type is more flexible.

• If a new, better implementation comes out, you can switch by changing one line of code.
Why separate interface from implementation?

// Create the list
List list = new ListImpl();

// Do lots of stuff with the list
list.add(new Student("Maurice", 16));
list.add(someOtherStudent);
...
Student s = (Student) list.get(15);
...
Why separate interface from implementation?

// Create the list
List list = new ListImplImproved();

// Do lots of stuff with the list
list.add(new Student(“Maurice”, 16));
list.add(someOtherStudent);
...
Student s = (Student) list.get(15);
...

None of the remaining code has to change at all!

Substitute a different implementation.
Why *not* an ADT?

- There are a few situations where you would *not* want to implement a data structure as an ADT.
- Encapsulating a data structure into an ADT incurs a small amount of time cost and space cost.
- In performance-critical programs (e.g., real-time systems, small-memory systems), this overhead might be a real problem.
- However, in the vast majority of programming scenarios, using data structures as ADTs is the right choice.
Implementing a List ADT.
List implementations

- Let’s finally talk about how to implement a List with the three methods `add`, `get`, and `remove`.

- We will cover two kinds of list implementations:
  - Array lists.
  - Linked lists.
Array lists

• Let’s go back to our “sketch” of how to manage a list that could “grow” when more elements were added:

• Start with a small array.

• If it gets full, make the array larger.

• Hide these details from the “user” -- the programmer using the ArrayList implementation -- behind the “wall of abstraction” provided by the List interface.
ArrayLists

• In our ArrayList ADT, we will store the data added by the `add(o)` method in an `Object[]`.

• This `Object[]` is the “underlying storage” of the ADT.

• In 1960s parlance, this is called the “backing store” of the data structure.

• What would be the “backing store” of the `OneLongBinarySequence` abstraction that the memory controller implements?
It is often useful to depict ADTs graphically:

```java
Object[] _underlyingStorage;

// Stores the number of boxes actually in use
int _numStoredElements;
```
ArrayLists

- Consider:
  
  ```java
  Object o1 = "Object1";
  Object o2 = "Object2";
  Object o3 = "Object3";
  ```

  ```java
  Object[] _underlyingStorage;
  ```

  ```java
  // Stores the number of boxes actually in use
  int _numStoredElements;
  ```

  ```java
  Element index: 0 1 2 3 4 5 6 7
  ```
ArrayLists

• Consider:
  ```java
  arrayList.add(o1);
  ```

```java
Object[] _underlyingStorage;
```
ArrayLists

- Consider:
  ```java
  arrayList.add(o1);
  arrayList.add(o2);
  ```

Object[]
_underlyingStorage;

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>o1</td>
<td>o2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Element index:

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
</table>
```

// Stores the number of
// boxes actually in use
int _numStoredElements;

Tuesday, August 2, 2011
ArrayLists

- Consider:
  ```java
  arrayList.add(o1);
  arrayList.add(o2);
  arrayList.add(o3);
  ```

```java
Object[] _underlyingStorage;
```

Element index:

```
0 1 2 3 4 5 6 7
```

// Stores the number of boxes actually in use
int _numStoredElements;
ArrayLists

- Consider:
  ```
  // More adds...
  ```

- After adding 8 objects to the list, the array is full. (How do we know?)

- If the user calls `add` again, we must enlarge the backing store.

```
Object[] _underlyingStorage;
```

```
<table>
<thead>
<tr>
<th>01</th>
<th>02</th>
<th>03</th>
<th>04</th>
<th>05</th>
<th>06</th>
<th>07</th>
<th>08</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
```

// Stores the number of boxes actually in use
```
int _numStoredElements;
```

Tuesday, August 2, 2011
Enlarging an array

• First, what does it mean to “enlarge” an array?

• In Java, once an array is allocated, its size cannot be changed:

```java
Object[] array = new Object[8];
array.length++; // this is nonsense
```
Enlarging an array

• Instead, we must allocate a new, larger array, and copy the old array data into the new array.
Enlarging an array

• Instead, we must allocate a new, larger array, and copy the old array data into the new array:

```java
// Allocate initial array
Object[] array = new Object[8];

... // The array gets filled up

// Create a new, larger array
Object[] largerArray = new Object[16];
// Copy the array data into the new array
for (int i = 0; i < array.length; i++) {
    largerArray[i] = array[i];
}
// Replace the old array with the new one
array = largerArray;
```
Enlarging the array

- After “enlarging” the array, we have:

```java
Object[] _underlyingStorage;

Element index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

// Stores the number of // boxes actually in use int _numStoredElements; 8
```
Enlarging an array

- It would be a pain to do this in every application we write in which we need a flexibly-sized array.

- Implementing this “array resizing” in a List ADT once-and-for-all is more efficient and more reliable.
Enlarging the array: implementation issues

- When should we resize the array?
- How do we keep track of how full the current array is?
- By how much should we enlarge the array?
Unannounced quiz 1