

Topic 13

Abstract Data Types and Data Structures

"Get your data structures correct first, and the rest of the program will write itself."
- David Jones

Abstract Data Types

- ▶ An *Abstract Data Type* is:
 - "A set of data values and associated operations that are precisely specified independent of any particular implementation."
- ▶ Our List example
 - `new List` returns an empty List, `[]`
 - `list.add(a)` adds `a` to the end of the list `[original_elements, a]`
 - `get(x)` returns the element at position `x` in the List
 - `size()` returns the size of the List

Abstract Data Types

- ▶ Java interfaces are a form of Abstract Data Types
 - list operations to be performed without any implementation details
 - [Java List interface](#)

`java.util`

Interface `List<E>`

All Superinterfaces:

[Collection<E>](#), [Iterable<E>](#)

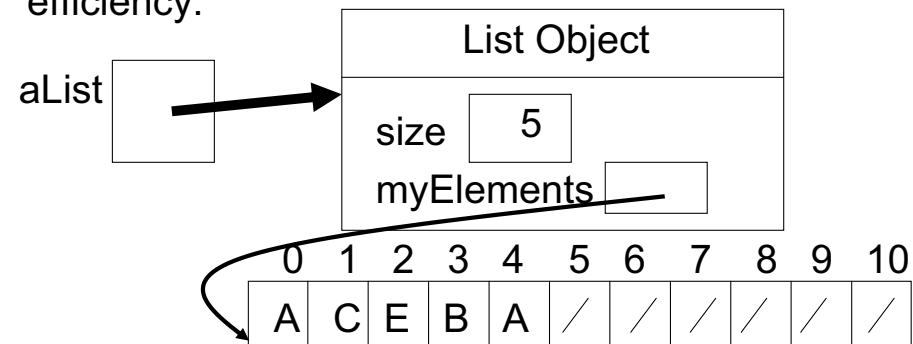
All Known Implementing Classes:

[AbstractList](#), [AbstractSequentialList](#), [ArrayList](#), [AttributeList](#), [CopyOnWriteArrayList](#), [LinkedList](#), [RoleList](#), [RoleUnresolvedList](#), [Stack](#), [Vector](#)

[A, C, E, B, A]

Data Structures

- ▶ A *Data Structure* is:
 - an implementation of an abstract data type *and*
 - "An organization of information, usually in computer memory", for better algorithm efficiency."



Data Structure Concepts

- ▶ Data Structures are containers:
 - they hold other data
 - arrays are a data structure
 - ... so are lists
- ▶ Other types of data structures:
 - stack, queue, tree, binary search tree, hash table, dictionary or map, set, and on and on
 - www.nist.gov/dads/
 - en.wikipedia.org/wiki/List_of_data_structures
- ▶ Different types of data structures are optimized for certain types of operations



Core Operations

- ▶ Data Structures will have 3 core operations
 - a way to add things
 - a way to remove things
 - a way to access things
- ▶ Details of these operations depend on the data structure
 - Example: List, add at the end, access by location, remove by location
- ▶ More operations added depending on what data structure is designed to do

ADTs and Data Structures in Programming Languages

- ▶ Modern programming languages usually have a library of data structures
 - [Java collections framework](#)
 - [C++ standard template library](#)
 - [.Net framework](#) (small portion of VERY large library)
 - Python lists and tuples
 - Lisp lists

Data Structures in Java

- ▶ Part of the Java Standard Library is the *Collections Framework*
 - In class we will create our own data structures and discuss the data structures that exist in Java
- ▶ A library of data structures
- ▶ Built on two interfaces
 - Collection
 - Iterator
- ▶ <http://java.sun.com/j2se/1.5.0/docs/guide/collections/index.html>

The Java Collection interface

- ▶ A generic collection
- ▶ Can hold any object data type
- ▶ Which type a particular collection will hold is specified when declaring an instance of a class that implements the Collection interface
- ▶ Helps guarantee *type safety* at compile time

Methods in the Collection interface

```
public interface Collection<E>
{
    public boolean add(E o)
    public boolean addAll(Collection<? extends E> c)
    public void clear()
    public boolean contains(Object o)
    public boolean containsAll(Collection<?> c)
    public boolean equals(Object o)
    public int hashCode()
    public boolean isEmpty()
    public Iterator<E> iterator()
    public boolean remove(Object o)
    public boolean removeAll(Collection<?> c)
    public boolean retainAll(Collection<?> c)
    public int size()
    public Object[] toArray()
    public <T> T[] toArray(T[] a)
}
```

Implementing Collection

- ▶ A class that implemented the Collection interface and did not add any more operations or requirements would be a *Bag*
- ▶ A Bag is a data structure.
- ▶ Items in the Bag have no implied order
 - from the users perspective of the Bag there is no first item or second item or ...
- ▶ Duplicates allowed. Can have multiple copies of equal objects in the Bag
 - differentiates a Bag from a *Set*

Why Reinvent the Wheel?

- ▶ Why study ADTs?
- ▶ Why reimplement some of them?
- ▶ How many of you will actually go out and create your own linked list data structure from scratch?
- ▶ The goal is to learn *how* to use data structures and how to *create* them.
- ▶ Knowledge of data structures sometimes used as a litmus test.

Don't re-invent
the **wheel**



List Class Revisited

"He's making a list
And checking it twice;
Gonna find out Who's naughty and nice
Santa Claus is coming to town"
- *Santa Claus is Coming to Town*



Our List Class

- ▶ Created a List class as an example of encapsulation and power of inheritance and polymorphism
 - also known as a dynamic array
- ▶ Underlying storage container was an array
- ▶ Maintained extra capacity because it "seemed like a good idea"

add Method

```
public class GenericList {
    private Object[] theArray;
    private int theSize;

    public void add(Object value) {
        if (theArray.length == theSize) {
            this.increaseSize();
        }
        theArray[theSize] = value;
        theSize++;
    }

    private void increaseSize() {
        Object[] newArray = new Object[theSize * 2];
        for (int i = 0; i < theSize; i++)
            newArray[i] = theArray[i];

        theArray = newArray;
    }
}
```

Best case Big O?
Worst case Big O?
Average case Big O?

Big O of the Default add?

- ▶ Normal Big O analysis fails
- ▶ "Most" adds are inexpensive
 - $O(1)$
- ▶ A few adds are very expensive due to resizing
 - $O(N)$
- ▶ What is the average case?
- ▶ To determine average case we'll do an *amortized* analysis
 - amortized is a term from accounting. Spread the cost of something over time.

Simplified Amortized Analysis

- ▶ Add N items to the list, always at end of list
- ▶ Assume internal container starts out at size 1
- ▶ Assume if we resize container we double the size

Item #	Size of Container	Work to Resize	Work to Add
1	1	0	1
2	1 -> 2	1	1
3	2 -> 4	2	1
4	4	0	1
5	4 -> 8	4	1
6	8	0	1
7	8	0	1
8	8	0	1
9	8 -> 16	8	1
...			
N - 1	N - 1	0	1
N	N - 1 -> 2N - 2	N - 1	1

Total Work to Add N items

- ▶ N steps to add:

```
myCon[size] = item;  
size++;
```

- ▶ The work in resizing is the sum of a *geometric sequence*

$$1 + 2 + 4 + 8 + 16 + \dots + (N-1)/2 + N-1$$

$$a_n = a_1 * r^{n-1}$$

$$\text{sum terms 1 through } n = (r * a_n - a_1) / (r - 1)$$

$$a_1 = 1, a_n = N - 1, r = 2,$$

$$\text{sum} = (2 * (N - 1) - 1) / (2 - 1) = 2N - 3$$

Average Work Per Item

- ▶ Total Work is $N + 2N - 3 = 3N - 3$
- ▶ Divide by the N items to amortize the cost:
 $(3N - 3) / N \text{ items} = 3 - 3/N \text{ steps per item}$
- ▶ The average work per item is $3 - 3/N$
- ▶ The amortized Big O is $O(1)$
- ▶ What would the amortized Big O of adding at the end of the list be if we resized the list by 50 elements every time we needed to resize?

insert Method

```
public void insert(int index, Object value) {  
    if(theArray.length == theSize) {  
        this.increaseSize();  
    }  
  
    for(int i = theSize - 1; i >= index; i--){  
        theArray[i + 1] = theArray[i];  
    }  
  
    theArray[index] = value;  
    theSize++;  
}
```

- ▶ Best case, worst case, and average case Big O?

get Method

```
public Object get(int index) {
    assert index < theSize :
        "List index out of bounds: " + index;
    return theArray[index];
}
```

- ▶ Best case, worst case, and average case Big O?

Other Methods

- ▶ Best, Worst, Average Case Big O for
 - remove method
 - insert all method
 - toString
 - equals
 - size

The Java ArrayList Class

- ▶ Implements the List interface and uses an array as its *internal storage container*
- ▶ It is a list, not an array
- ▶ The array that actual stores the elements of the list is hidden, not visible outside of the ArrayList class
- ▶ all actions on ArrayList objects are via the methods
- ▶ ArrayLists are generic.
 - They can hold objects of any type!

Some of the ArrayList Methods

- ▶ int size()
- ▶ boolean add(E x)
- ▶ E get(int index)
- ▶ E set(int index, E x)
- ▶ void add(int index, E x)
- ▶ E remove(int index)
- ▶ Iterator<E> iterator()
- ▶ ListIterator<E> listIterator()
- ▶ NOT all of the methods of the ArrayList class

ArrayList's (Partial) Class Diagram

