Simple List Algorithms

15-211
Fundamental Data Structures and Algorithms

Ananda Guna & Klaus Sutner

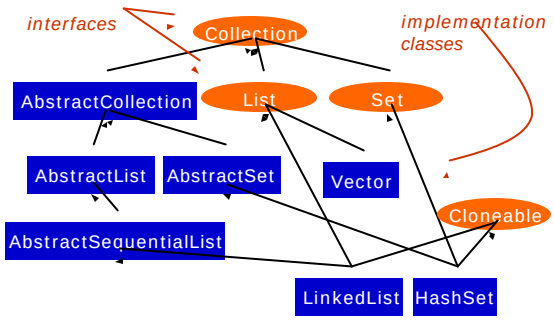
January 16, 2003

Based on lectures given by Peter Lee, Avrim Blum, Danny Sleator, William Scherlis, Ananda Guna & Klaus Sutner

In this lecture

- We will talk briefly about the collections API- more later
- We will talk about Abstract Data Types and Implementation of interfaces
- We will look at a recursive definitions on List class
- We will discuss counting the number of operations

J2SE Collection API (excerpt)



Java Collections API

- Resides in Java.util
- Java.util contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).
- Java Collections API interface provide structures and operations on them
  - isEmpty, size, add, contains, remove, clear, toArray, iterator
  - Iterator interface methods– hasNext, next, remove

Sample Collections

- ArrayList - Resizable-array implementation of the List interface.
- Arrays – include sorting, searching etc..
- LinkedList - Linked list implementation of the List interface.
- Stack - The Stack class represents a last-in-first-out (LIFO) stack of objects.
- Hashtable - This class implements a hashtable, which maps keys to values
- More on these later...

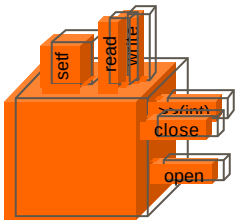
What are interfaces?

- an interface is a device or a system that unrelated entities use to interact
- You use an interface to define a protocol of behavior that can be implemented by any class anywhere in the class hierarchy. Interfaces are useful for the following:
  - Capturing similarities among unrelated classes without artificially forcing a class relationship.
  - Declaring methods that one or more classes are expected to implement.
  - Revealing an object's programming interface without revealing its class.

Why use interfaces?

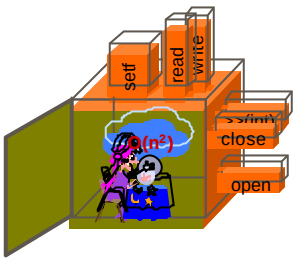
- Interfaces allow us to separate interface from implementation.
- If properly designed, then clients assume only the interface.
- Clients don't have to change, even if the underlying implementations change.

Interface vs implementation



8

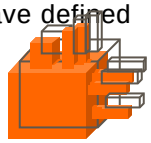
Interface vs implementation



9

Abstract data types

- When the allowable operations on a type of data are controlled in this manner, we say that we are using *abstract data types*.
- In our first example, we have defined the ADT of counters.



10

Abstract data types

- Abstract data types can make it easier to…
  - …make use of pre-existing code.
  - …work with a team.
  - …maintain code in the long run.
  - *…write down the results of your careful thinking about the problem.*

Abstract Data Types

- Abstract Classes
  - May contain implementation details, but cannot be used to instantiate objects. Used in inheritance (discuss later)
- Interfaces
  - Do not contain implementation details. Must be implemented by client classes

Interface Examples

```
// A recycling interface definition
public interface Recycling {
   /** Is the object recyclable */
   public boolean isRecyclable();
   /** Easy to move when recycling */
   public boolean isEasyToMove();
}
```

A Class implementing the Interface

```
public class Car implements Recycling
// A car class
{
   private boolean shift; // Data specific to a car
   Car(boolean shift) // Class constructor
   {
      this.shift = shift;
   }
   // Method specific to a car
    public boolean isManual() { return shift; }

   // Methods implementing the Recyclable interface
   public boolean isRecyclable() { return true; }
   public boolean isEasyToMove() { return false; }
}
```

Another Class implementing the Interface

```
class Washer implements Recycling
/** A washing machine class */
{
   int maxTemp; /** Data specific to a washing machine*/
   Washer(int maxTemp) /** Class constructor */
   {
      this.maxTemp = maxTemp;
   }
   /** Method specific to a washing machine */
   public int getMaxTemp() { return maxTemp; }

   /** Methods implementing the Recyclable interface*/
   public boolean isRecyclable() { return true; }
   public boolean isEasyToMove() { return true; }
}
```

A Bigger Example

Lists of integers

```
public interface IntListInterface {

    public int length ();

    public IntList append (int n);

    public String toString ();
}
```

One possible implementation

```
public class IntList implements IntListInterface {
    private int element;
    private IntListInterface next;

    public IntList (int n) {
         element = n;  next = null;
    }

    public int length () {
         if (next == null) return 1;
         else return 1 + next.length();
    }

    public IntListInterface append (int n) {
         if (next == null) next = new IntList(n);
         else next = next.append(n);
         return this;
    }

    public String toString () {
         String elt = Integer.toString(element);
         if (next == null) return elt;
         else return elt + " " + next.toString();
    }
}
```

Linked lists of integers

Empty lists?

- Question:
  - In this implementation, how is an empty list represented?

Null

- Null is commonly used in Java, mainly because of the influence from C and C++.
- However, in object-oriented programming, the use of null can lead to errors because *null is not an object*.
  - *Methods can be invoked only on objects.*

Excerpt of client code

```
…
IntListInterface mylist;
IntListInterface anotherList;

…

anotherList = myList.append(5);
```

If myList is null, then this will cause an exception to be raised (and the program to be halted).

Null is not an object

- One (crude) work-around is for the client code to check for null before invoking a method.

```
…
IntListInterface mylist;
IntListInterface anotherList;

…
if (myList != null) {
    anotherList = myList.append(5);
…
```

Null

- It is ok to use null in a Java program.
  - Especially when using null as a kind of "uninitialized" value.
- However, in many cases it is cleaner to avoid the use of null.
  - *So, how might we do this in this case?*

Empty lists

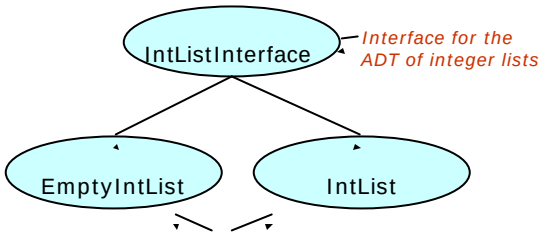
```
public class EmptyIntList implements IntListInterface {
    public int length () { return 0; }

    public IntListInterface append (int n) {
       return new IntList(n);
    }

    public String toString () {
         return "";
    }
}
```

Non-empty lists

```
public class IntList implements IntListInterface {

    private int element;
    private IntListInterface next;

    public IntList (int n) {
        element = n;  next = new EmptyIntList();
    }

    public int length () {
        return 1 + next.length();
    }

    public IntListInterface append (int n) {
        next = next.append(n);
        return this;
    }

    public String toString () {
        return Integer.toString(element) + " " + next.toString();
    }
}
```

Break

What have we done here?



Inductive definitions

- In fact, what we have done reflects something about the structure of the linked list ADT.
- An inductive definition:
  - *An integer list is either*
    - *empty, or* — Base case
    - *an integer paired with an integer list* — Inductive case

Inductive definitions of ADTs

- Many ADTs have inductive definitions.
- When they do, it is often possible to reflect this in the Java classes, just as we have done in this example.

More on Linked Lists

- What should a realistic linked list implementation look like?
- Not as obvious as is sounds.
- Example: does **`L.append(x)`** modify the list L ?

Our current implementation

```
public interface IntListInterface {…}


public class IntList implements IntListInterface {
      public IntListInterface append(int n) {
            next = next.append(n);
            return this; }
}


public class EmptyList implements IntListInterface {
      public IntListInterface append(int n) {
            return new IntList(n); }
}
```

Which ...

... prompts another question: What's the running time? Not "wall-clock" time, but the number of "steps"?

Answer: The append is handed down to the end of the list, where it eventually causes a miraculous transformation.

This could be as many as *n* "steps", where *n* is the length of the list.

A bit later, we will make some mathematical definitions to allow us to say that this is $O(n)$, or *linear time*.

Our current implementation

```
public interface IntListInterface {…}


public class IntList implements IntListInterface {
      public IntListInterface append(int n) {
            next = next.append(n);
            return this; }
}


public class EmptyList implements IntListInterface {
      public IntListInterface append(int n) {
            return new IntList(n); }
}
```

Mutation or not?

`L.append(12);`

changes L: there is one new node at the end.

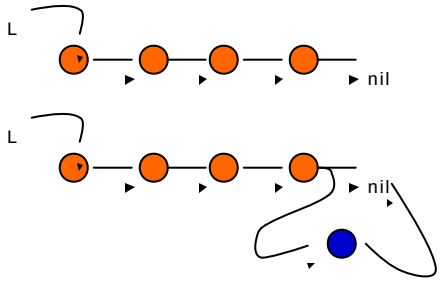
So we don't have to do a

`L = L.append(12);`

(except in one nasty case:

when L is empty.)

E.g.



How About Reversal?

```
public class IntList implements IntListInterface {
      public IntListInterface reverse() {
            IntListInterface t = next.reverse();
            if (t == null) return this;
            return t.append(element);
      }
}

public class EmptyList implements IntListInterface {
      public IntListInterface reverse() {
            return this; }
}
```

First Running Time

For EmptyList clearly just one "step".

This is called *constant time*, or $O(1)$

For IntList:

- the #steps for the tail, plus
- the #steps for append

Ignoring constants:

t(0) = 1

t(n) = n + t(n-1)

Solving for t would tell us how many "steps" it takes to reverse a list

easy:

t(n) = n + (n-1) + (n-2) + … 1 = n(n+1)/2

So, it's $O(n^2)$, a *quadratic* operation.

Can we do better? Reverse a list in linear time??

It turned out that there is a way to do a linear time reverse. (Think of array reversal)

Reversing a List - iterative

current ← *head*

head ← *null*

loop

nextNode ← *current*

current = current .next

nextNode . next = head

head ← *nextNode*

until current is null

- **What is the order of this algorithm?**
- **Can you trace the algorithm?**

How to Sort a List?

Sorting an array is easy and we know many algorithms.

But how do we sort a singly linked list?

As always, think inductively:

sort(nil) = nil

sort(L) = insert head(L) in "order" to sort(tail(L))

Code for ordered insert

```
public IntList(int x, IntList head) {
      num = x; next = head;
}
```

new constructor method

```
public IntList order_insert(int x) {
      if( x <= num )
            return new IntList(x,this);
      next = next.order_insert(x);
      return this;
}
```

tail takes care of placement

Analysis of ordered insert

This creates only one new object (the node holding x).

The running time depends on the position of x in the new list.

But in the *worst case* this could take *n* steps.

Analysis of sort()

sort(nil) = nil

sort(L) = insert the head into the right place in sort(tail(L))

t(0) = 1

t(n) = n + t(n-1)

which we already know to be "very roughly" n^2.

... and ...

... sort() creates a total of n new objects (we share the empty guy).

Question: How many steps are required for sorting an already sorted list?

Better sorting

- The sorting algorithm we have just shown is called *insertion sort*.
- It is OK for very small data sets, but otherwise is slow.
- Later we will look at several sorting algorithms that run in many fewer steps.

A preview of some questions

- Question: Insertion sort takes n^2 steps in the *worst case*, and n steps in the *best case*. What do we expect in the *average case*? What is meant by "average"?
- Question: What is the fastest that we could ever hope to sort? How could we prove our answer?

For Tuesday

- We will talk about solving simple recurrences
- Talk about Java class hierarchy
- Read Chapter 5 & 6
- Homework 0 is online now
  - **Due: Monday January 20th at midnight (11:59PM)**