today's agenda

- assignments
  - Assignment 4 is out and due on Tuesday
- a quick look back
  - abstract data types
- linked lists
- copying data structures
  - shallow copy
  - deep copy
- binary trees
last time...

an abstract data type or ADT is data and operations on that data that are precisely specified independent of any implementation

the operations may or may not have efficiency guarantees

a data structure is a systematic approach to storing and accessing data so that it can be used efficiently for a specific purpose

a data structure is the implementation of an ADT

real world $\rightarrow$ ADT $\rightarrow$ data structure (class)
last time...

we saw several abstract data types

- list
- stack
- queue
- priority queue
- set
- dictionary
now let’s look at

linked lists (again)
now let’s look at
copying data
What does this method do?

```java
public static int mystery(int[] numbers){
    int n = numbers.length;
    for(int i=1; i<n; i+=1){
        numbers[i] = Math.max(numbers[i], numbers[i-1]);
    }
    return numbers[n-1];
}
```

How would you write the contract for this method?

- pre-conditions
- post-conditions
- side effects
So what happened here?

```java
public static int mystery(int[] numbers) {
    /* numbers = input_argument_numbers; */
    int n = numbers.length;
    for (int i = 1; i < n; i += 1) {
        numbers[i] = Math.max(numbers[i], numbers[i - 1]);
    }
    return numbers[n - 1];
}
```

the method has three variables when it is called (in its activation record)

- `numbers` is an input parameter
- `n` is a local variable to the method
- `i` is a local variable to the method (scope is restricted to for loop)

Java passes input arguments by value. When `mystery` is called, the input parameter `numbers` is assigned the value of the input
So what happened here?

```java
public static int mystery(int[] numbers){
    /* numbers = input_argument_numbers; */
    int n = numbers.length;
    for(int i=1; i<n; i+=1){
        numbers[i] = Math.max(numbers[i], numbers[i-1]);
    }
    return numbers[n-1];
}
```

when Java assigns the input parameter variable it uses a **shallow copy**.

the assignment operator = always performs a shallow copy. For reference data types, = copies the **reference** (and not the data of the object)
let's trace through the memory model

```java
public static int mystery(int[] numbers){
    /* numbers = input_argument_numbers; */
    int n = numbers.length;
    for(int i=1; i<n; i+=1){
        numbers[i] = Math.max(numbers[i], numbers[i-1]);
    }
    return numbers[n-1];
}

public static void main(String[] args){
    int[] n = new int[]{1,3,6,2,-10,20,10};
    int m = mystery(n);
}
```
Shallow versus Deep copy

A **shallow copy** of reference data types simply copies the *reference*.

```java
Student one = new Student("cat", 12332);
Student two = one; // shallow copy of student object
```

After the shallow copy, the variables `one` and `two` are now **aliases** of each other. They each refer/point to the same place in memory.

```java
two.setName("dog");
System.out.println(one.getName()); // outputs "dog"
```

With aliases, changing the data of one will change the data of the other. This is sometimes the behaviour you want and sometimes not.

The assignment operator `=` always does a shallow copy.

When passing objects into a function Java always does a shallow copy. (other languages may be different)
Shallow versus Deep copy

a **deep copy** makes a copy of all the data in the object.

```java
Student one = new Student("cat", 12332);
Student two = new Student();
two.setName(one.getName()); // manual deep copy
two.setID(one.getID()); // of a student object
```

_one_ and _two_ have the same data but are not aliases of each other. Changing the data of one has no affect on the other.

After a deep copy there should be no shared memory (except for Strings or other **immutable** data)

with a shallow copy _one == two_ is true and _one.equals(two)_ is likely false*

with a deep copy _one == two_ is false and _one.equals(two)_ is likely true*

*Assuming a good definition of **.equals**
Shallow versus Deep copy

public class Student{
    String name;
    int id;
    Date dob;
    Course[] courses;
}

public class Course{
    String name;
    String semester;
    String instructor;
    String grade;
}

How would you do a deep copy of a Student object?

public Student deepCopy(){...}
now let’s look at

binary trees
binary trees

a **binary tree** is another abstract data type