In Class notes for July 3rd 2017 Data Structures

admin

- If you use code from somewhere— just cite it
- midterm is the Monday of the 4th week; Monday July 24
- 50% homework, 20% midterm, 30% final
- Office hours Tuesdays Thursdays Fridays in the evenings

lecture

What is a data structure?

- a way a organizing information

What is an algorithm?

- sequence of steps to perform on a data structure in order to solve certain tasks

An array stored in memory

- information stored in a location in memory
- arrays start at zero because you're indexing by offset from the first element of the array

Talking about algorithms

- we want to describe the algorithm by using big*oh because input, hardware, and other things are varied

Binary Search

- breaks arrays up into smaller sub*arrays
- if you're trying to find the middle of an array that has an odd, you can just use integer division because it will always floor it, per integer division.
- What are we actually doing when we use binary search

- we're doing n =2ˆx, then each iteration of the algorithm will give you 2ˆ{x-1}. In the worst case scenario, you'll be doing 2ˆ{x-x}, but we want to have it in terms of n, which will eventually give us big-oh of log_2 of n.
- We only want to keep track of the low and high points for the array, we dont want to be creating new arrays. This is for space efficiency reasons.
-

ArrayLists

- this is a class that wraps around an Array[]

example for ArrayLists:

```
ArrayList<AnyType> myArray = new ArrayList<Integer>();
```

```
AnyType myGenericType = new AnyType; // I can do this in my generic methods
```

A generic class is a data type that lets you use any datatype

LinkedList

- very different from an array, has a node with 2 values: data and a reference to the next element
- also a doubly linked list that has references to both previous and next node

proof by induction for:

- F_i < (5/3)ˆi when i /geq 1

Show base case:

Assume true for the base case and some i up to and including some arbitrary k:

- so then we have to prove that $ F_{k+1} < (5/3)ˆ{k+1} $
- this implies that $ F_{K+1} < (5/3)ˆk + (5/3)ˆ{k-1} $
- then you can pull a (3/5) out of both terms, but there are 2 in the second
- $ F_{K+1} < (15/25)(5/3)ˆk + (9/25)(5/3)ˆ{k-1} $
- $ F_{k+1} < (24/25)(5/3)ˆ{k+1} $

For Recursion:

1. There must be some kind of base case, a case that makes you stop
2. each time you call the function again, you need to be making progress towards the base case.
3. You have to assume that the recursive call works. For example, with the Fibonacci numbers, you have to assume that the recursive calls will work.
4. Don't repeat work, this is sometimes referred to as the compound interest rule

Example of Recursive Algorithm

```java
public class Factorial {
  // this method assumes non malicious user, not defined for
  public static int factorial(int x) {
    // base case
    if (x==0) {
      return 1;
    }

    return x*factorial(x-1);


  }

}
```

```java
public class Fib {

  public static int fib(int x) {
    // base case
    if (x == 0 || x == 1)
      return 1;
    // recursive call
    return fib(x-1) + fix(x-2);
  }

  public static final void main(String[] args) {
    int x = Integer.parseInt(args[0]);
    System.out.println(fib(x));

  }

}
```

Comments on the above:

- why is this bad? we're repeating work going all the day down the tree. We're computing the same thing over and over throughout the tree. This algorithm is in fact big-oh of 2ˆn
- we can use this Dynamic Programming, specifically memoizing the work you've already done. You can use an ArrayList for storing the work you've already done

Generics

- We can have both generic methods and generic types
- We want to write methods that can work on any data type
- We can also bound a what a method can work on with something like:
-

```
public static <? extends shape> exmaple(<? extends shape> x) { ... }
```

For Homework:

```
public boolean recursiveBinary(ArrayList<AnyType> subArray, int index){
    // want write a recursive
        // you want the parameters to look something like:
            // (array, begin, end, target)
        // the trick is to overload the method
        // you'll have two methods, one which is recursive, the other
        // which is not.
        // you want to make the recursive one private
}
```