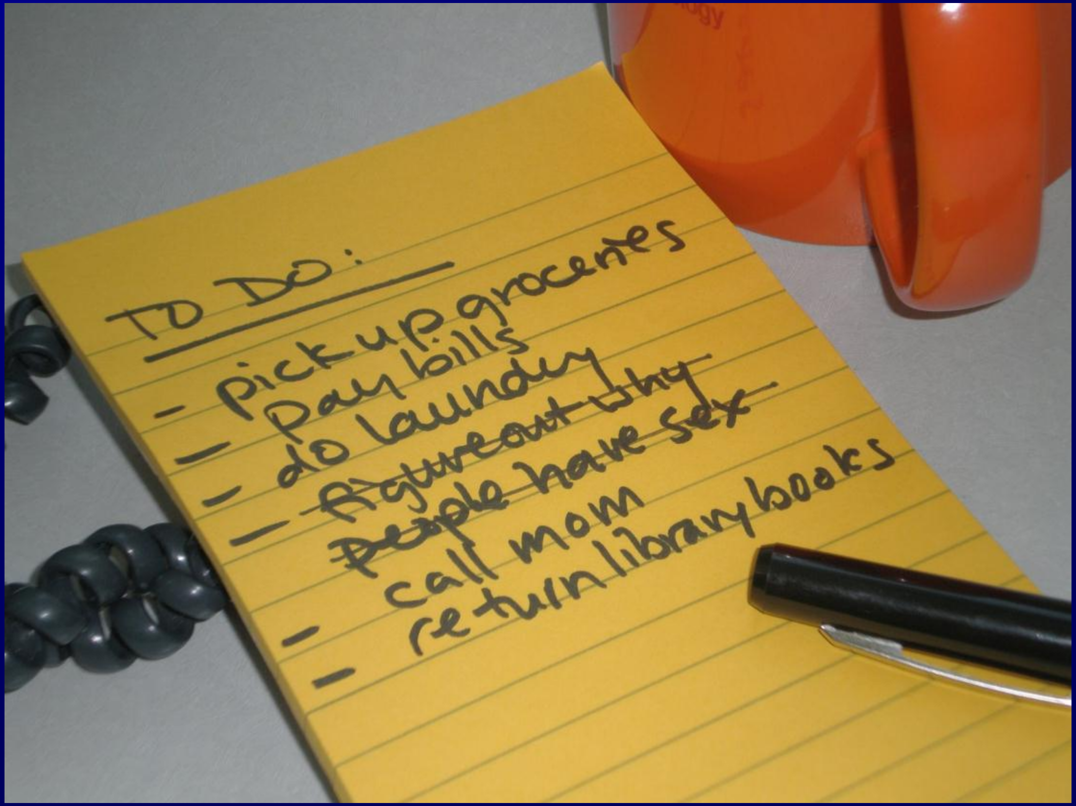
ArrayLists & LinkedLists Analysis

- ArrayList Class (again)
- Linked Lists (again)
- Algorithm analysis (continuation)

- Amortized runtime

Lists as Data Structures

- *Lists* are ordered collections of data
  - They can contain duplicate values
  - They do have indexes 0 .. length-1



TO DO:

- pick up groceries
- pay bills
- do laundry
- ~~figure out why~~
~~people have sex~~
- call mom
- return library books

Java's List<E> interface

- List<E> represents a sequence of element that can be accessed by index. It may have duplicate values.

- Java has two implementations of List<E> interface:
  - ArrayList<E>
  - LinkedList<E>

- The only difference, besides the implementation, is the run times of their methods.

Methods in the ArrayList class

- add(object), add(index, object)
  - adds element (an object) to the end of the list
  - adds element (an object) at specified index, relocating the rest
- set(index, object)
  - changes object at specified element
- get()
  - returns an element (an object)
- remove(index), remove(obj)
  - deletes an element given a value or an index
- size()
- contains(obj)
  - searches for obj

ArrayList running times

- size()
  - O(1). Why?

- add(E object)
  - O(1). Why?

- remove(int index)
  - Expected is O(n). Why?
  - Best is O(1). Why?

- How does Java implements the ArrayList Class?

ArrayList running times

- get(int index) and set(int index, E object)
  - Expected?
  - Worst?
  - Best?

- add(int index, E object)
  - Expected?
  - Worst?
  - Best?

Methods in the LinkedList class

- add(object), add(index, object)
    - adds element to the end of the list
- clear()
    - removes all the elements in the list
- get(index)
    - returns an element
- remove(index), remove(obj),
    - deletes an element given a value or an index
- size()
- set(index, obj)
    - changes the value of a element given an index
- contains(obj)
    - searches for obj

Other Methods in a LinkedList class

- addFirst(), addLast()
  - adds an element
- getFirst(), getLast()
  - returns an element
- removeFirst(), removeLast(),
  - deletes an element

LinkedList running times

- size()
  - O(1). Why?

- add(E object)
  - O(1). Why?

- remove(int index)
  - Expected is O(n). Why?
  - Best is O(1). Why?

- How does Java implements the LinkedList Class?

LinkedList running times

- get(int index) and set(int index, E object)
  - Expected?
  - Worst?
  - Best?

- add(int index, E object)
  - Expected?
  - Worst?
  - Best?

Question

- Why do the *ArrayList* class and the *LinkedList* class in Java have the same methods?
  - Is this a coincidence?

Expanding the size of an array

- When the array is full, we create a new array that is larger and copy all the elements to the new array.
    - What’s the cost of doing so?

```
private void expand() {
        int biggerSize = ------------------;
        E[] largerArray = (E[]) new Object[biggerSize];
        for (int i = 0; i < size; i++) {
                largerArray[i] = values[i];
        }
        values = largerArray;
}
```

Expanding the size of an array

- What size should we make *biggerSize* ?

n	**size**	**copies**
1	1	0
2	2	1
3	4	1 + 2 (1 copy when you expand from 1 to 2, and 2 copies when you expand from 2-4)
5	8	1 + 2 + 4
…		
n	n	$1 + 2 + 4 + \ldots + n = n*(n+1)/2 = \mathbf{O(n^2)}$

Expanding the size of an array

- Suppose INITIAL_CAPACITY is 10. How many copies would you make to add n elements?

n	**size**	**copies**
10	10	0
20	20	10
30	30	10 + 20
40	40	10 + 20 + 30
...		
10*k	10*k	10 + 20 + 30 + ... + = 10*(1+2+3+...+n/10) = 10*(n/10)*((n/10)+1)/2 = **$O(n^2)$**

It doesn't matter how big INITIAL_CAPACITY is. The cost of n add() operations is $O(n^2)$.

On average, the cost of a single add() operation is O(n).

Better idea …

- Double the size of the array for every expand

n	**size**	**copies**	
1	1	0	
2	2	1	
4	4	1 + 2 = 3	= O(n)
8	8	1 + 2 + 4 + 8 = 15 = 2*n-1	= O(n)
16	16	1 + 2 + 4 + 8 + 16 = 31 = 2*n-1	= O(n)
32	32	1 + 2 + 4 + 8 + 16 + 32 = 2*n-1	= O(n)

Amortized runtimes

- *Amortized runtime* is the expected runtime per operation of a sequence of n operations.

- It is not the same as the *average runtime*.

Example: add(E) is O(n) because we next to expand the array quite often. Now, lets’ add and delete using remove(index). Then we don’t need to expand as often. Thus, the amortize time of a single call to add(E) is O(1)!

Readings

- Java API for ArrayList class
- Java API for LinkedList class
- Java API for List interface

- Watch the video *Linked List Efficiency Analysis* (Linked List Unit folder)

Homework

- Exam I next Thursday Oct. 8
- No homework due next week
- No quiz next week
- Exam review session
  - This weekend
  - TBA on Piazza