CS367 Announcements
Monday, July 8, 2013

- H3 due Mon Today (July 8th) 6pm
- Midterm Thursday July 11th in class
  - material through July 3rd
  - review sheet to be posted

Last Time

- Comparable interface
- Linked Lists (cont.)
- LinkedList variations

Today

- Complexity Review
- Finish Linked Lists
- Stacks and Queues

H2 Problem 5: Complexity

Assume two `ArrayLists` have items added to them so that `L1` has N items and `L2` has M items, and a third list `L3` is empty.

- What is the **worst-case** time complexity for each code fragment below?
- Further, identify whether the problem size depends on N, M, both, or neither.

Assume the cost of expanding an array is O(1) and that the `equals()` method runs in constant time as well. Express the complexity using "Big-O" notation.

```
A.
if (!L2.contains(obj)) {
   L2.add(obj);
} else {
   L2.remove(obj);
   L1.add(0,obj);
}

B.
while(L1.contains(obj)) {
   L1.remove(0);
   if (L2.size() > L1.size()){
      L2.remove(L2.size() - 1);
   }
}

C.
for (int k = 0; k < L1.size(); k++) {
   for (int j = 0; j < L2.size(); j++) {
      if (L1.get(k).equals(L2.get(j)) {
         L3.add(L1.get(k));
      }
   }
}
```

H2 Problem 5: Complexity

A) O(M+N)

In the worst case scenario the problem time is linear and dependent on the size of both M and N.

```
if (!L2.contains(obj)) {    //O(M), worst case must search entire list
   L2.add(obj);             //O(1), add last so no shifting and assuming
                            //      expand is O(1)
} else {
   L2.remove(obj);          //O(M), worst case must search entire list
   L1.add(0,obj);           //O(N), add at front does shifting
}
```

O(M) + max(O(1), O(M) + O(N)) = O(M) + O(M+N) = O(2M+N) = O(M+N)

B) O(N^2)

The problem size is determined by the size of L1, which is N items.
(All operations on L2 occur in constant time)

```
while(L1.contains(obj)) {        //up to N iterations
                                 //O(N), search entire list each iteration
   L1.remove(0);                 //O(N), shift all remaining elements
   if (L2.size() > L1.size()){   //O(1)
      L2.remove(L2.size() - 1);  //O(1), removing from end of list
   }
}
```

N * (O(N) + O(N) + O(1) + O(1)) = O(N^2)

C) O(N*M)

L1 contains N items and L2 contains M items so the problem size depends on both M and N.

```
for (int k = 0; k < L1.size(); k++) {     //N iterations
   for (int j = 0; j < L2.size(); j++) {  //M iterations
      if (L1.get(k).equals(L2.get(j)) {   //O(1)
         L3.add(L1.get(k));               //O(1), add last so no shift
                                          //      and expand is constant
      }
   }
}
```

N * M * 1 = O(N*M)

Linked List Variations

Singly-linked chains of nodes

with tail reference

with header node

Doubly-linked chains of nodes

Circular singly-linked chains of nodes

Circular doubly-linked chains of nodes

Comparing List ADT Implementations

Space Requirements

Array:

Singly-linked list:

Circular singly-linked list:

Doubly-linked list:

Comparing List ADT Implementations

Time Requirements

	constructor	add(E)	add(int,E)	contains(E)	size	isEmpty	get(int)	remove(int)
Array	O(1)	O(N)			O(1)	O(1)		
Singly-Linked List (SLL)	O(1)	O(1)			O(1)	O(1)		
Circular SLL	O(1)	O(1)			O(1)	O(1)		
Doubly-LL	O(1)	O(1)			O(1)	O(1)		
Circular DLL	O(1)	O(1)			O(1)	O(1)		

Improving `add(E)` When Array is full

"Naïve" Approach

"Shadow Array" Improvement

Comparing List ADT Implementations

Ease of Implementation

Array:

Singly-linked list:

Circular singly-linked list:

Doubly-linked list:

Iterators and Linked Lists

Recall `Iterator` interface

```
public interface Iterator<E> {
    boolean hasNext();
    E next();      //NoSuchElementException if no next
    void remove(); //UnsupportedOperationException if not implemented
```

Goal: create `LinkedListIterator` class

Recall `SimpleArrayListIterator` (from the Lists reading)

```
public class SimpleArrayListIterator implements Iterator<Object> {
    private SimpleArrayList list;
    private int curPos;

    public SimpleArrayListIterator(SimpleArrayList list) {
        this.list = list;
        curPos = 0;
    }

    public boolean hasNext() {
        return (curPos < list.size());
    }

    public Object next() {
        if (!hasNext()) throw new NoSuchElementException();
        Object result = list.get(curPos);
        curPos++;
        return result;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Implementing LinkedListIterator

```
import java.util.*;

public class LinkedListIterator<E> implements Iterator<E> {

    LinkedListIterator(             ){


    }

    public boolean hasNext() {


    }

    public E next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }









    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Position-oriented ADTs