ABSTRACT DATA TYPES
SETS, LISTS, TREES, ETC.
Text:
- Chapters 10, 11 and 12

Homework: Learn these List methods, from http://docs.oracle.com/javase/7/docs/api/java/util/List.html
- add, addAll, contains, containsAll, get, indexOf, isEmpty, lastIndexOf, remove, size, toArray
- myList = new List(someOtherList)
- myList = new List(Collection<T>)
- Also useful: Arrays.asList()
Introduction to Danaus

Simulation of a Butterfly on an island, with water, cliffs, trees. A3, just fly around in a specific way. A5, A5, collect info about the island, A6 collect flowers, etc. Aroma, wind.
Understanding assignment A3

<table>
<thead>
<tr>
<th>(1,1)</th>
<th>(2,1)</th>
<th>(3,1)</th>
<th>(4,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,2)</td>
<td>(2,2)</td>
<td>(3,2)</td>
<td>(4,2)</td>
</tr>
<tr>
<td>(1,3)</td>
<td>(2,3)</td>
<td>(3,3)</td>
<td>(4,3)</td>
</tr>
<tr>
<td>(1,4)</td>
<td>(2,4)</td>
<td>(3,4)</td>
<td>(4,4)</td>
</tr>
</tbody>
</table>

- A 4x4 park with the butterfly in position (1,1), a flower and a cliff.
Understanding assignment A3

- A 4x4 park with the butterfly in position (1,1), a flower and a cliff.

- The same park! The map “wraps” as if the park lives on a torus!
Summary of These Four Lectures

Discuss Abstract Data Type (ADT): set of values together with operations on them: Examples are:

- set, bag or multiset
- list or sequence, stack, queue
- map, dictionary
- tree, binary tree, BST
- graph

Look at various implementations of these ADTs from the standpoint of speed and space requirements. Requires us to talk about

Asymptotic Complexity: Determining how much time/space an algorithm takes.

Loop invariants: Used to help develop and present loops that operate on these data structures—or any loops, actually.
Abstract Data Type (ADT)

An Abstract Data Type, or ADT:
A type (set of values together with operations on them), where:
- We state in some fashion what the operations do
- We may give constraints on the operations, such as how much they cost (how much time or space they must take)

We use ADTs to help describe and implement many important data structures used in computer science, e.g.:

set, bag or multiset               tree, binary tree, BST
list or sequence, stack, queue     graph
map, dictionary
ADT Example: Linked List

- **Head** = first element of the list
- **Tail** = rest of the list

```
33  10  -7  1  84  \phi
```
ADT example: set (bunch of different values)

Set of values: Values of some type E (e.g. int)

Operations:

1. Create an empty set (using a new-expression)
2. size() – size of the set
3. add(v) – add value \( v \) to the set (if it is not in)
4. delete(v) – delete \( v \) from the set (if it is in)
5. isIn(v) – = “\( v \) is in the set”

Constraints: size takes constant time. 
add, delete, isIn take expected (average) constant time but may take time proportional to the size of the set.

We learn about hashing later on, it gives us such an implementation
Java Collections Framework

Java comes with a bunch of interfaces and classes for implementing some ADTs like sets, lists, trees. Makes it EASY to use these things. Defined in package java.util.

Homework: Peruse these two classes in the API package:

ArrayList<E>: Implement a list or sequence — some methods:
- add(e)
- add(i, e)
- remove(i)
- remove(e)
- indexOf(e)
- lastIndexOf(e)
- contains(e)
- get(i)
- set(i, e)
- size()
- isEmpty()

Vector<E>: Like ArrayList, but an older class

They use an array to implement the list!

i: a position. First is 0
e: an object of class E
Maintaining a list in an array

- Must specify array size at creation
- Need a variable to contain the number of elements
- Insert, delete require moving elements

When list gets full, create a new array of twice the size, copy values into it, and use the new array

Class invariant: elements are, in order, in \( b[0..\text{size}-1] \)
Java Collections Framework

**Homework:** Peruse following in the API package:

**LinkedList<E>:** Implement a list or sequence —some methods:

- `add(e)`
- `add(i, e)`
- `remove(i)`
- `remove(e)`
- `indexOf(e)`
- `lastIndexOf(e)`
- `contains(e)`
- `get(i)`
- `set(i, e)`
- `size()`
- `isEmpty()`
- `getFirst()`
- `getLast()`

Uses a doubly linked list to implement the list or sequence of values

i: a position. First is 0
e: an object of class E
Stack\langle E \rangle \text{ in java.util} \quad \text{Queue not in java.util}

**Stack\langle E \rangle:** Implements a stack:
- size()
- isEmpty()
- push(e)
- pop()
- peek()

**Queue:** Implement a queue:
- size()
- isEmpty()
- push(e)
- pop()
- peek()

peek: get top or first value but don’t remove it

Stack is actually a subclass of Vector, So you can use all of Vector’s methods

Stack LIFO last in first out

Queue: FIFO first in first out
Linked List

- Head = first element of the list
- Tail = rest of the list
public static boolean search(T x, ListCell c) {
    while(c != null) {
        if (c.getDatum().equals(x)) return true;
        c = c.getNext();
    }
    return false;
}
Why would we need to write code for search? It already exists in Java utils!

- Good question! In practice you should always use `indexOf()`, `contains()`, etc.

- But by understanding how to code search, you gain skills you’ll need when working with data structures that are more complex and that don’t match predefined things in Java utils.

- General rule: *If it already exists, use it.* But for anything you use, know how you would code it!
Recursion on Lists

- Recursion can be done on lists
  - Similar to recursion on integers

- Almost always
  - Base case: empty list
  - Recursive case: Assume you can solve problem on the tail, use that in the solution for the whole list

- Many list operations can be implemented very simply by using this idea
  - Although some are easier to implement using iteration
Recursive Search

- **Base case: empty list**
  - return false

- **Recursive case: non-empty list**
  - if data in first cell equals object x, return true
  - else return the result of doing linear search on the tail
public static boolean search(T x, ListCell c) {
    if (c == null) return false;
    if (c.getDatum().equals(x)) return true;
    return search(x, c.getNext());
}
Iterative linked list list reversal

Change this:

pre  head → v1 → v2 → v3 → v4 → null

into this:

post head → v1 → null → v2 → v3 → v4

Reverse the list by changing head and all the succ fields

Legend:

val  succ
v1
Iterative linked list reversal

Change this:

pre  head  v1  v2  v3  v4  null

into this:

post head  v1  null  v2  v3  v4

Use a loop, changing one succ field at a time. Getting it right is best done by drawing a general picture that shows the state of affairs before/after each iteration of the loop. Do this by drawing a picture that combines the precondition and postcondition.
Iterative linked list list reversal

**Change this:**

pre  head  \( v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow \text{null} \)

**into this:**

post head  \( v_1 \rightarrow \text{null} \) \( v_2 \rightarrow v_3 \rightarrow v_4 \)

The loop will fix the succ fields of nodes beginning with the first one, then the second, etc.

The first part of the list will be reversed —look like pre

The second part will not be reversed —look like post
Iterative linked list list reversal

**pre**

- head
- v1
- v2
- \( v(n-1) \)
- null

**post**

- head
- v1
- null
- v(\( n-1 \))
- vn

Loop invariant (it’s always true):

- First \( i \) nodes reversed
- Rest unreversed

Diagram:

- Head connects to v1
- v1 connects to v2
- v2 connects to \( v(n-1) \)
- \( v(n-1) \) connects to vn
- Loop invariant
Iterative linked list list reversal

**pre**

```
head --> v1 --> v2 --> ... --> vn null
```

**head**

```
p1 --> null --> p2
```

**u**

```
p3 --> p4 --> null
```

**Reversed part**

**Unreversed part**

**post**

```
head --> v1 null --> v(n-1) --> vn
```

```
``
Initially, unreversed part is whole thing:  \( u = \text{head}; \)
Reversed part is empty:
\( \text{head} = \text{null}; \)
When to stop loop?

u = head; head = null;

while ( u != null )

Upon termination, unreversed part is empty: u == null.
Continue as long as u != null

post head v1 null v(n-1) vn

Reversed part

Unreversed part
Loop body: move one node from u list to head list. Draw the situation after the change

u = head; head = null;
while (u != null) {
    Node t = head; head = u; u = u.succ; head.succ = t;
}

u = head; head = null;
while (u != null) {
    Node t = head; head = u; u = u.succ; head.succ = t;
}
Homework: Write a recursive function for Linked List Reversal!