Linked Structures

A primary alternative to arrays

Reading LDC Ch 14.10 - 14.14
Arrays versus Lists

- Arrays are limited in the sense they have a fixed size
  - Resizing as needed must be done carefully and is not efficient

- A list of linked LinearNode objects (aka: Linked List) is the primary alternative to an array-based implementation of a collection
The picture below is neat but it is not accurate.

The picture on the right is closer to what a Linked Structure looks like in the Memory of the computer.

Do you see what a **LinearNode** looks like?
Managing Linked Lists

is more complicated than managing an array

- A node may be inserted or deleted at any location
  - at the front of the list,
  - among the interior nodes, or
  - at the end of the list

- There are a few basic techniques when managing nodes on the list, no matter what the list is used to store

- Special care must be taken when dealing with the first or last node in the list so that the entire list is maintained appropriately
Inserting a node at the front
Inserting a node in the middle

front

current

node

1

2
Deleting the first node in the list

front

1 node

2

3
Deleting an interior node

front

previous 1 current

2
Let’s say we want to create a Linked collection of \texttt{Person} objects. What if we embed the link onto the object? Using only this one class, a linked structure is created:

- One \texttt{Person} object contains a link to another \texttt{Person} object.
- This second object contains a reference to a third \texttt{Person}, etc.
- This type of object is sometimes called \textit{self-referential}

```java
public class Person{
    private String name;
    private String address;
    private Person next;
    // and whatever else
}
```
A not-so-good Implementation Idea: Objects containing Links

- Advantage: A linked structure has capacity limited only by memory in the computer
  - Is a *dynamic structure* because its size grows and shrinks as needed to accommodate the number of objects stored

- Disadvantage: It cannot be reused!
A Much Better Implementation Idea: Objects without Links

- A flaw in the Person class is that the self-referential Person class must be designed so that it “knows” it may become a node in a linked list.
- This violates the goal of separating the implementation details from the parts of the system that use the collection.
- We better separate the details of the linked list structure from the objects that the list stores:
  Define a separate node class `LinearNode` that serves to link the objects together.
package javafoundations;

public class LinearNode<T> {  
    private LinearNode<T> next;
    private T element;
    //Constructor: Creates an empty node
    public LinearNode() {
        next = null;
        element = null;
    }
    //Constructor: Creates node with element
    public LinearNode(T elem) {  
    
    }
}
/**@return the node that follows this one */
public LinearNode<T> getNext()
{
    return next;
}

/** Sets the node that follows this one */
public void setNext (LinearNode<T> node)
{
    next = node;
}

/** @return element stored in this node */
public T getElement()
{
    return element;
}

/** Sets the element stored in this node */
public void setElement (T elem)
{
    element = elem;
}
LinkedStack:

Stack Implementation using Linear Nodes
(part of javafoundations)
package javafoundations;

public interface Stack<E> {

/** @param newItem is added to the top of the stack. */
public void push(E newItem) {

/** @return the removed top element from the stack. */
public E pop();

/** @return the top element of this stack without removing it. */
public E peek();

/** @return true iff the stack contains no elements. */
public boolean isEmpty();

/** @return the number of elements in the stack. */
public int size();

/** @return a string representation of the stack. */
public String toString();
}
Stack Animation

Adding an element

Removing an element

Top of stack
/** Represents a linked implementation of a stack. */
package javafoundations;

import javafoundations.exceptions.);

public class LinkedStack<T> implements Stack<T>
{
    private int count;
    private LinearNode<T> top;

    // Constructor: Creates an empty stack.
    public LinkedStack()
    {
        count = 0;
        top = null;
    }

    (more...)
What a LinkedStack looks like
/** @return the top element of the stack */
public T peek () throws EmptyCollectionException {
    if (count == 0) {
        throw new EmptyCollectionException
            ("Peek failed. Stack is empty.");
    }
}

/** @return true IFF stack is empty */
public boolean isEmpty() {
}

/** @return the number of elements on the stack */
public int size() {
}
Popping a node off a Stack

(we just need the element, though, not the node)
/** Removes the element at the top of this stack *
* @return a reference to it.
* @throws an EmptyCollectionException if stack empty *
*/

public T pop() throws EmptyCollectionException {
    if (count == 0)
        throw new EmptyCollectionException
            ("Pop failed. Stack is empty.");
    T result = top.getElement();
    top = top.getNext();
    count--;
    return result;
}
Pushing a node on to Stack
(slightly different than code in next slide)
/** Adds the element at the top of this stack
* @param element will be “pushed” onto the Stack. */

public void push(T element) {

    // We need a LinearNode to point to element!

}
```java
/** @return a string representation of this stack. */
public String toString() {
    String result = "<top of stack>\n";
    LinearNode current = top;

    while (current != null) {
        result += current.getElement() + "\n";
        current = current.getNext();
    }

    return result + "<bottom of stack>";
}
```