(Linked) Lists

ADT

List Operations

Abstract Data Type

List: finite sequence of values *(sequence === ordered by position)*

- Construct an empty list
- Test for empty
- Get count of values
- Prepend to list (add at front)
- Append to list (add to end)
- Get value at position *N* in list *0 ≤ N < size of list*

?Insert value at position *N* *0 ≤ N < size of list*

?Remove value at position *N* *0 ≤ N < size of list*

?Find position of a value *or -1 if not found*

What challenges for implementing as an array?

Collections *(in general)*

Efficiency of a data structure can impact design choice

What are common operations?

How do these operations perform *(generally)*

- Linear? O(n) "on the order of n"
- Quadratic $O(n^2)$
- many others

Remove duplicates

Insert in the middle

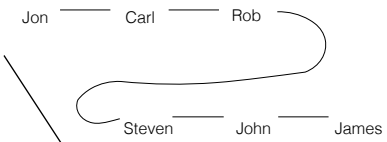
Search for match in unordered list

Linked List

Abstraction of a list of items

- "front" and "back" of list
- No fixed size (dynamically grows/shrinks)
- get item at position N
- Insert an item at position N
- Remove an item at position N
- Find position of a particular item
- Allow me to store anything

Linked List



Each item holds the value and a reference (aka pointer) to the next item in the list

Sometimes called "nodes" or "list elements"

- Node <E>
  - E value;
  - Node next;

Linked List itself consists of a head reference

Linked List

LinkedList<E> constructor

- Node head = null;

Node constructor (aValue, nextNode)

- value = aValue;
- next = nextNode;

LinkedList
add (aValue, position)

if position is at start of list?

else?

LinkedList
value remove (position)

if position is at end of list?

else?

Linked List: Node

New generic class Node <E>

has a ***value*** of type E

has a reference to another Node: ***next***

node attributes will be public!

we aren't hiding anything here

Constructor: Node (E val, Node<E> n)

Constructor: default constructor - null, null

Linked List

New generic class LinkedList<E>

private attributes: head -refers to first Node<E>
count -counts # of items

public default constructor to make empty list

getter for count
implement public void addFront (E val)
implement public String toString ( )

Test these: Create & print; create,add,print

Homework for Wednesday

public E removeFront( )
public E valueAt (int pos) *-return null if illegal*

EXTRA CREDIT

public addAt (E val, int pos) *-if pos>len add to end*
- if pos < 0 add to front

public E removeAt (int pos) *-if pos>len remove end*
- if pos < 0 remove front

Linked Loops (oops)

Beware

next points to itself, or something earlier in list

Be careful - think (draw)

How to tell…

toString taking forever?

Test cases have neither success nor failure

small blue triangle -still running

It WILL happen!

Oops

JUnit tests for infinite loops

```
@Test (timeout=TIMEOUT)
public void testFunction() {
```

timeout is in milliseconds (thousandths of a sec)
normal tests will take much less than 10 ms
so use ~1000 (1 second)

This causes FAILURE if the test takes too long

Use a constant (so you can easily change it)

Must comment it out for debugging/breakpoints!!!

Debugging

Use debugger

Test and fix one at a time *(start simple!)*

do your "due diligence" - if it fails, understand why

Linked List

If you have implemented just these four:

constructor - empty list
addFront(value)
removeFront(value)
count ( )

What is another name for this functionality?

Linked Structures

Advantages
- dynamic structure: memory allocated as needed
- add and remove don't require "re-shuffling"
- Stacks and Queues are easily implemented

Disadvantages
- extra storage required for pointers (references)
- Sequential access
- Reverse listing

Doubly-Linked Lists

Forward and Backward link

Modify Node to add: *prev* reference

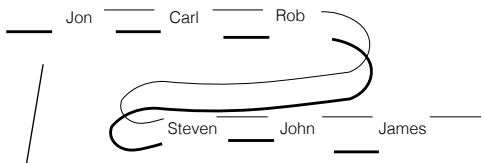
Modify LinkedList to add: *tail* reference

addFront & removeFront have special cases

must modify to keep head,tail, next, prev

first's prev is null; last's next is null

Doubly-Linked List



Each node
- Node <E>
  - E value;
  - Node next;
  - Node prev;

Linked List itself consists of
head and **tail** references

Dynamic Structures

Lists

one or two-way connections

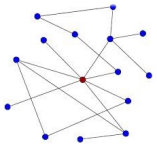
Webs/networkds

n-way connections

Graphs

rethink our maze as a series of 4-

Trees



Data Structures: Summary

List, Queue, Stack, etc.
Set of required operations

Many ways to implement any given data structure

Time of operations on the data structure may differ depending on implementation choice

Time complexity is a rough measure of speed

O(n) O(1) $O(n^2)$

Big O notation

characterizes the growth rate *as n becomes much bigger*