ABSTRACT DATA TYPES
SETS, LISTS, TREES, ETC.

Lecture 9
CS2110 – Fall 2013

References and Homework

2

- 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

3

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

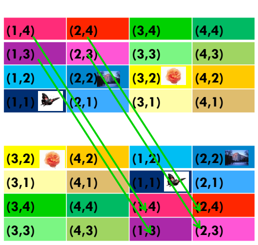
4

(1,4)	(2,4)	(3,4)	(4,4)
(1,3)	(2,3)	(3,3)	(4,3)
(1,2)	(2,2)	(3,2)	(4,2)
(1,1)	(2,1)	(3,1)	(4,1)

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

Understanding assignment A3

5



- 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

6

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)

7

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

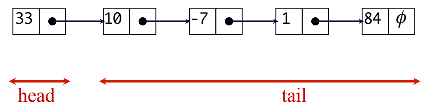
list or sequence, stack, queue — graph

map, dictionary

ADT Example: Linked List

8

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



ADT example: set (bunch of *different* values)

9

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

10

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

11

- Must specify array size at creation
- Need a variable to contain the number of elements
- Insert, delete require moving elements
- Must copy array to a larger array when it gets full

Class invariant: elements are, in order, in b[0..size-1]

size 4

b

24	-7	87	78				

unused

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

Java Collections Framework

12

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<E> in java.util Queue not in java.util

13

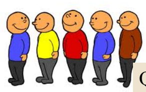
Stack<E>: Implements a stack:
size() isEmpty()
push(e) pop() peek()

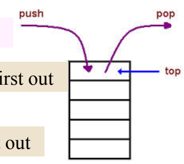
Queue Implement a queue:
size() isEmpty()
push(e) pop() peek()

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

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

Stack LIFO last in first out



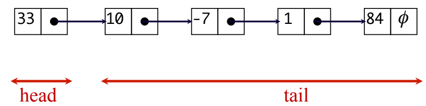


Queue: FIFO first in first out

Linked List

14

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



Access Example: Linear Search

15

```
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!

16

- 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

17

- 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

18

- 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

Recursive Search: Static method

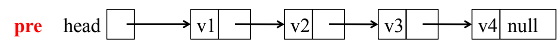
19

```
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());
}
```

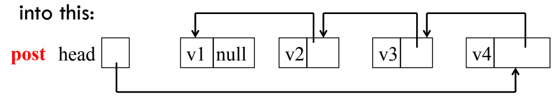
Iterative linked list reversal

20

Change this:

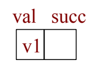


into this:



Reverse the list by changing head and all the succ fields

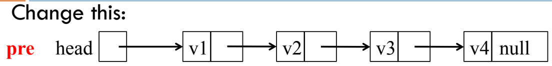
Legend:



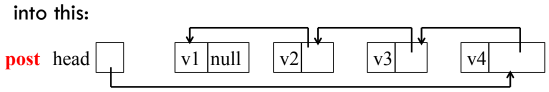
Iterative linked list reversal

21

Change this:



into this:

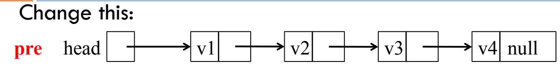


Use a loop, changing one succ field at a time. Getting it right is best done by drawing a general picture that shows the state of affairs before/after each iteration of the loop. Do this by drawing a picture that combines the precondition and postcondition.

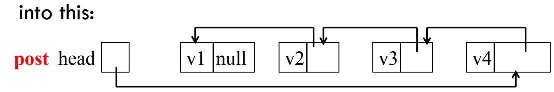
Iterative linked list reversal

22

Change this:



into this:



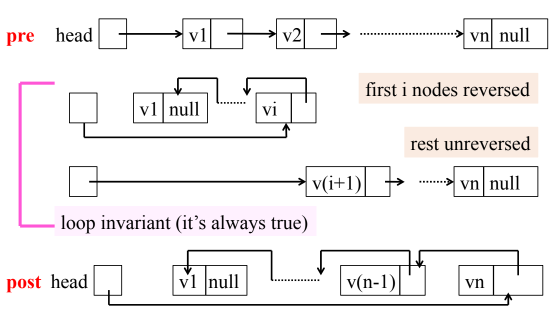
The loop will fix the succ fields of nodes beginning with the first one, then the second, etc.

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

The second part will not be reversed —look like post

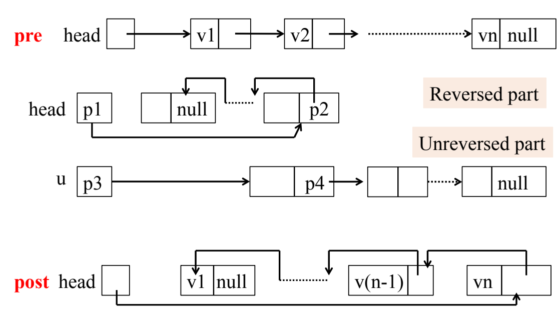
Iterative linked list reversal

23



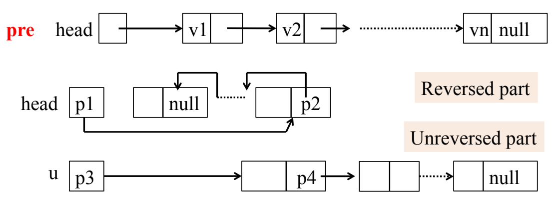
Iterative linked list reversal

24



Make the invariant true initially

25



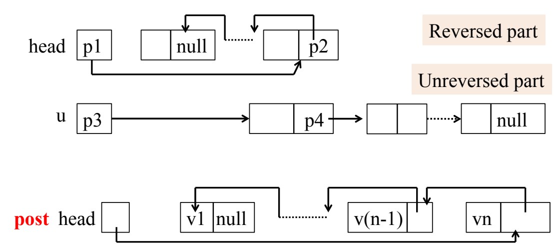
Initially, unreversed part is whole thing: u= head;
Reversed part is empty: head= null;

When to stop loop?

26

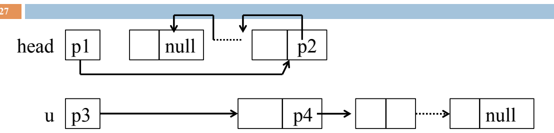
u= head; head= null;
while (u != null **)**

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

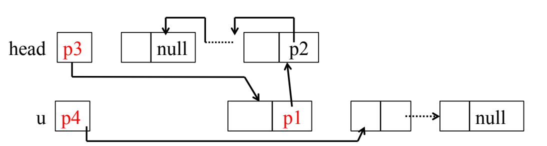


Loop body: move one node from u list to head list. Draw the situation after the change

27



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



Recursive Reverse

28

- Homework: Write a recursive function for Linked List Reversal!