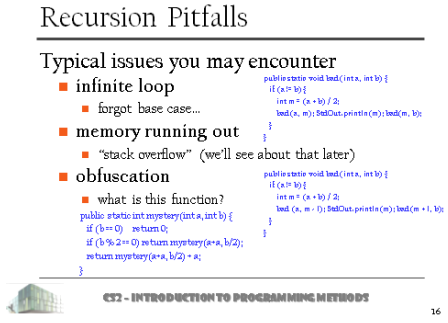
CS 2

Introduction to Programming Methods

Last Week: Convex Hull

Discussed recursion some more

- and finished with CH



Recursion Pitfalls

Typical issues you may encounter

- infinite loop
  - forgot base case...
- memory running out
  - "stack overflow" (we'll see about that later)
- obfuscation
  - what is this function?

```
public static int mystery(int a, int b) {
  if (b == 0)    return 0;
  if (b % 2 == 0) return mystery(a+a, b/2);
  return mystery(a+a, b/2) + a;
}
```

```
public static void bad(int a, int b) {
  if (a != b) {
    int m = (a + b) / 2;
    bad(a, m); StdOut.println(m); bad(m, b);
  }
}
```

```
public static void bad(int a, int b) {
  if (a != b) {
    int m = (a + b) / 2;
    bad(a, m - 1); StdOut.println(m); bad(m + 1, b);
  }
}
```


Today's Show

Data structures you ought to know

- plethora out there
  - arrays, lists, stacks, queues, deques, hashtable, trees
  - [most exist in Java and C++ already – STL library]
- best choice is *extremely* problem-dependent
  - no free lunch
  - but often cheaper/better lunch based on usage

Arrays

You know them well by now

- reserved memory space
- access by index
- pros
  - super simple (matrix-like); low memory overhead (just a block, no extra fat)...
- cons
  - need to know size early; ordered insertion in O(n); removal in O(n); indices can change anytime; no good for maintaining a "list" (ex: students in CS2...)

```
int items[];
items = new int[MAX];
... items[0] ....item[i]...
```

Dynamic Arrays (like std::vector)

Special behavior when access out of bounds

- keep track of current size
  - for instance, with a private int variable currSize
- if access or insertion beyond currSize?
  - create bigger array
    - good compromise: twice the size of the current one
  - copy the old array into the new one
    - garbage collection will take care of the rest in Java; not in C++
    - clean up after yourself in general
  - can be used for shrinkage too
- no change on insertion/deletion: still slow...

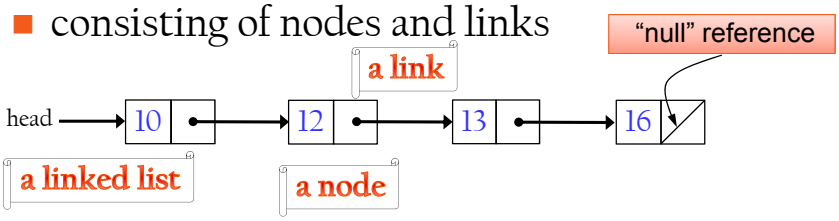
Linked Lists

If you...

- want to add or delete elements efficiently
  - no resizing; O(1) cost
- but have no need for direct access

A linked list is a dynamic data structure

- consisting of nodes and links



Codewise I

Typical Java implementation of a node

```
class LLNode {
   public int item;
   public LLNode next;
}
```

(content)

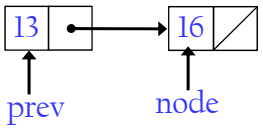
(links to another node)

Constructors

```
public LLNode(int newItem) {
    item = newItem;
    next = null;
  } // end constructor
```

```
public LLNode(int newItem, LLNode nextNode) {
    item = newItem;
    next = nextNode;
  } // end constructor
```

```
LLNode node = new LLNode (new Integer(16));
LLNode prev = new LLNode (new Integer(13), node);
```



Codewise II

Basic management

- set a value
- read a value
- set the link
- get the link

```
public void setItem(int newItem) {
  item = newItem;
 } // end setItem

 public int getItem() {
  return item;
 } // end getItem

 public void setNext(LLNode nextNode) {
  next = nextNode;
 } // end setNext

 public LLNode getNext() {
  return next;
 } // end getNext
```

Linked List

Got a node class, now what?

- define a list start/head
  - private LLNode head (or via pointer in C++)
- maybe a length
  - private int numNodes
  - [could be recomputed on the fly]
- constructor
  - and variants

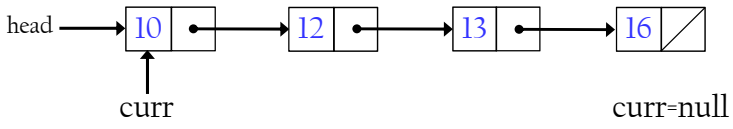
```
public LinkedList() {
   head = null;
   numNodes = 0;
 } // end default constructor
```

Linked List Contents

How to display the list values in order?

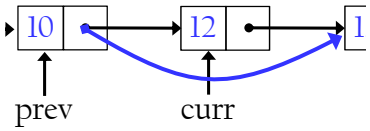
```
for (LLNode curr = head; curr != null; curr = curr.getNext()) {
        System.out.print(curr.getItem() + " "); } // end for
```

- circulate/iterate through list node by node



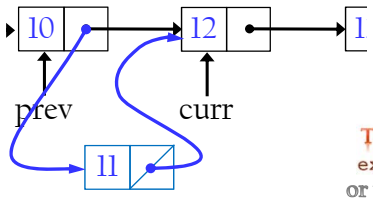
Adding and Deleting Elements

Deleting an item referenced by curr



Clean up after yourself...

Adding an item between prev and curr



WARNING!
TREAT HEAD DIFFERENTLY
`ex: head = head.getNext();`
or use a sentinel (dummy node)

Fancier Variants of Linked Lists

No ez access to the previous elmt of a list...
unless you create a doubly linked list

- now a node consists of: item, next, & prev

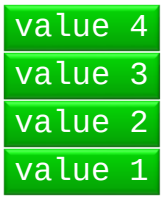
- now can traverse in any direction
  - costs 50% more memory, though...
  - and more opportunity to mess up the links

Circular list?

- trivial change (keep a reference to be able to start *somewhere*)

Stack

Think: cafeteria trays, pez dispenser...



As its name indicates...

- push: stack up one more
- pop: unpile the top element

Last element in, first element out

- LIFO data structure
- push when no memory left: stack overflow
- pop when nothing left: stack underflow

Convenient to track/remember (think postits)

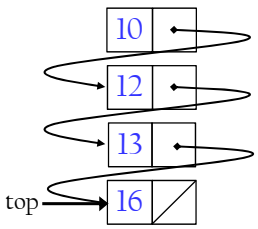
Stack Implementation

One way is with a linked list

- head always indicates the top of the stack
- push and pop as easy as an update of top
  - push: add item at list's beginning
  - pop: remove the first item

Array ok too!

- prescribed size

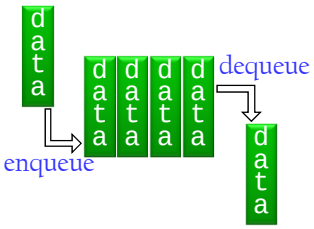



```
Stack s = new Stack();
s.push(16);
s.push(13);
s.push(12);
s.push(10);
s.pop();
```

Queue

Think: lines at bank, bus,...

Data structure allowing items to be added at the end, and removed from the front



- FIFO (first in, first out)
  - very useful for, e.g., printer jobs
    - also called first come first served
- deque: double-ended queue
  - push-back, push-front, pop-back, pop-front
  - but in Java: offerLast, offerFirst, pollLast, pollFirst

Trees

Think: family/decision tree, directory structure,...

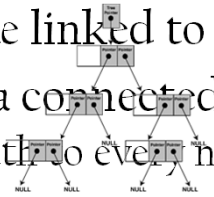
A set of nodes with a single starting point

- called the **root**—usually shown on top

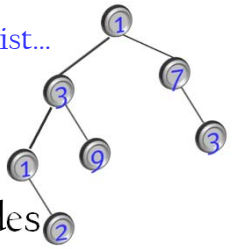
Each node linked to other node(s) by **edge**

A tree is a connected graph

- ∃ a path to every node in the tree



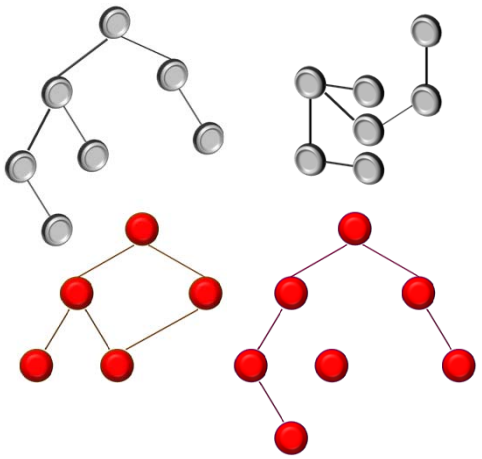
not linear linked list...

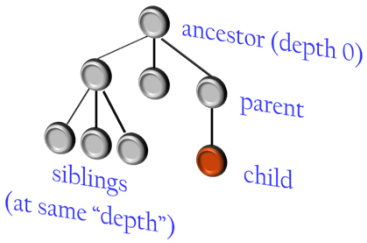


No cycles in a tree

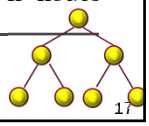
- one less edge than number of nodes

Tree Zoo and Terminology





leaf: node with no children
path: sequence of nodes
n-ary tree: at most n children
full n-ary tree:
depth d contains n^d nodes



Generic Binary Tree Node Class

stands for an arbitrary type (Integer, String, …)

```
public class TreeNode<T> {
  private T item;
  private TreeNode<T> leftChild;
  private TreeNode<T> rightChild;

 // Initializes node with item and no children, by default
  public TreeNode(T newItem) {
    item = newItem;
    leftChild  = null;
    rightChild = null;
  } // end constructor

 // Initializes node with two children references.
  public TreeNode(T newItem,
                  TreeNode<T> left, TreeNode<T> right) {
    item = newItem;
    leftChild  = left;
    rightChild = right;
  } // end constructor

// Returns the item field.
  public T getItem() { return item; } // end getItem

  // Sets item field to newItem.
  public void setItem(T newItem) {
    item  = newItem;
  }  // end setItem

  // Returns reference to the left child.
  public TreeNode<T> getLeft() {
    return leftChild;
  }  // end getLeft

 // Sets left child reference.
  public void setLeft(TreeNode<T> left) {
    leftChild  = left;
  }  // end setLeft

 // Returns reference to right child.
  public TreeNode<T> getRight() {
    return rightChild;
  }  // end getRight

 // Sets right child reference.
  public void setRight(TreeNode<T> right) {
    rightChild  = right;
  }  // end setRight

}  // end TreeNode
```

Creation of a Small Tree

```
TreeNode<Integer> root=new TreeNode<Integer>(new Integer(20));
TreeNode<Integer> root.setLeft(
   new TreeNode<Integer>(new Integer(5),
          new TreeNode<Integer>(new Integer(4)),    // left
          new TreeNode<Integer>(new Integer(12)))   // right
                                                );
```

Small tree, of "height" 3

- length of longest path from root

```
public int heightOfBinaryTree(TreeNode<T> node)
{  if (node == null) { return 0; }
   else {
      return 1 +  Math.max(heightOfBinaryTree(node.getLeft()),
                           heightOfBinaryTree(node.getRight));
   } //endif
}
```