CSC 321: Data Structures

Fall 2013

Proofs & trees

- proof techniques
  - direct proof, proof by contradiction, proof by induction
- trees
- tree recursion
- BinaryTree class

Direct proofs

the simplest kind of proof is a logical explanation or demonstration

CLAIM: The best case for sequential search is O(1)

PROOF: Suppose the item to be found is in the first index. Then sequential search will find it on the first check. You can't find something in fewer than one check.

CLAIM: you can add to either end of a doubly-linked list in O(1) time.

PROOF:

- add at front

```
front = new DNode(3, null, front);          → O(1)
if (front.getNext() == null) {              → O(1)
    back = front;                           → O(1)
}
else {
   front.getNext.setPrevious(front);        → O(1)
}
```

- add at back

```
back = new DNode(3, back, null);            → O(1)
if (back.getPrevious() == null) {           → O(1)
    front = back                            → O(1)
}
else {
    back.getPrevious.setNext(back);         → O(1)
}
```

Proof by contradiction

to disprove something, all you need to do is find a counter-example

CLAIM: every set has an even number of elements.

DISPROOF: { 4 }

however, you can't prove a general claim just by showing examples

CLAIM: every even number is divisible by 4.

PROOF?: 4 % 4 = 0, 8 % 4 = 0, 12 % 4 = 0, …

to prove a claim by contradiction

- assume the opposite and find a logical contradiction

CLAIM: there is no largest integer

PROOF: Assume there exists a largest integer. Call that largest integer N.
But N+1 is also an integer (since the sum of two integers is an integer), and N+1 > N.
This contradicts our assumption, so the original claim must be true.

Proof by induction

inductive proofs are closely related to recursion

- prove a parameterized claim by building up from a base case

To prove some property is true for all N ≥ C (for some constant C):

BASE CASE: Show that the property is true for C.
HYPOTHESIS: Assume the property is true for all n < N
INDUCTIVE STEP: Show that that the property is true for N.

CLAIM: 1+2+…+N = N(N+1)/2

BASE CASE: N =1. 1 = 1(1+1)/2 ✓
HYPOTHESIS: Assume the relation holds for all n < N, e.g., 1+2+…+(N-1) = (N-1)N/2.
INDUCTIVE STEP: Then 1+2+…+(N-1)+N = [1+2+…+(N-1)]+N
= (N-1)N/2 + N
= $(N^2 – N)/2 + 2N/2$
= $(N^2 + N)/2$
= N(N+1)/2 ✓

Proof by induction

CLAIM: The recurrence relation Cost(N) = Cost(N-1) + C has the closed-form solution Cost(N) = CN

BASE CASE: N =1. Cost(1) = Cost(0) + C = C

HYPOTHESIS: Assume the relation holds for n < N, e.g., Cost(N-1) = C(N-1)

INDUCTIVE STEP: Then Cost(N) = Cost(N-1) + C by definition

= C(N-1) + C by induction hypothesis

= C((N-1) + 1)

= CN ✓

FUNDAMENTAL THEOREM OF ARITHMETIC: every integer N > 1 is either prime or the product of primes

BASE CASE?

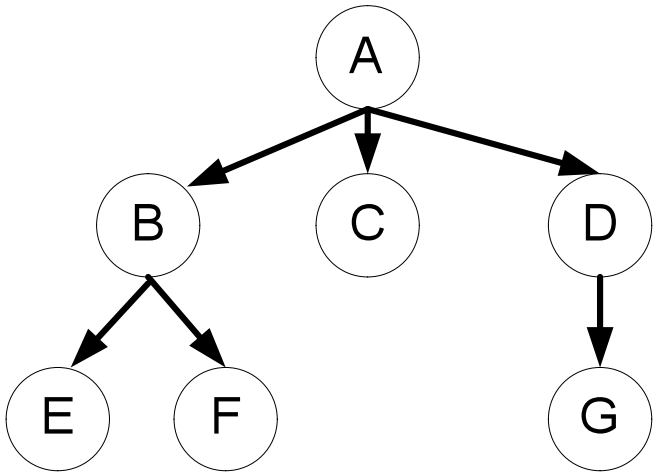
HYPOTHESIS?

INDUCTIVE STEP?

Trees

a tree is a nonlinear data structure consisting of nodes (structures containing data) and edges (connections between nodes), such that:

- one node, the *root,* has no *parent* (node connected from above)
- every other node has exactly one parent node
- there is a unique path from the root to each node (i.e., the tree is connected and there are no cycles)

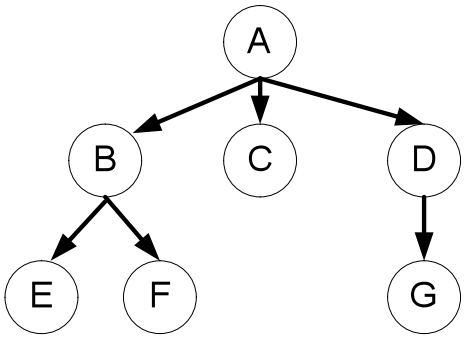


nodes that have no children (nodes connected below them) are known as *leaves*

Recursive definition of a tree

trees are naturally recursive data structures:

- the empty tree (with no nodes) is a tree
- a node with subtrees connected below is a tree



empty tree

tree with 1 node (empty subtrees)

tree with 7 nodes

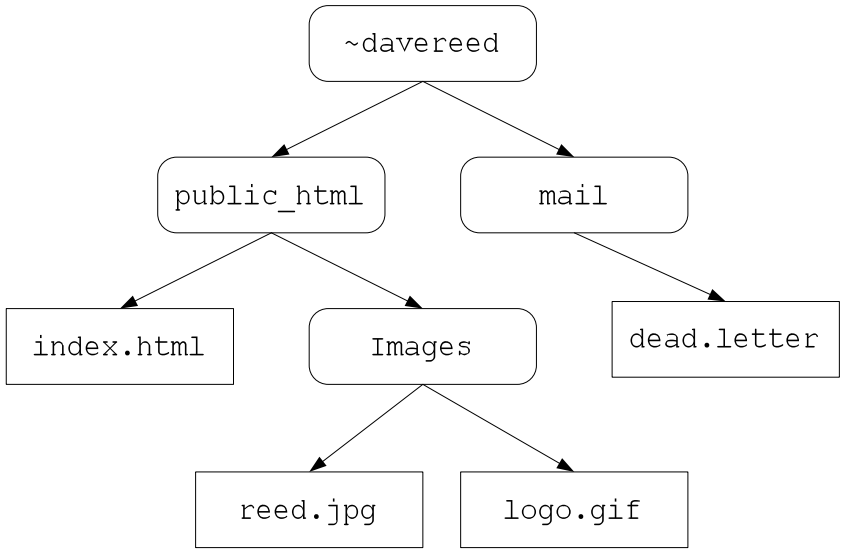
a tree where each node has at most 2 subtrees (children) is a *binary* tree

Trees in CS

trees are fundamental data structures in computer science

example: file structure

- an OS will maintain a directory/file hierarchy as a tree structure
- files are stored as leaves; directories are stored as internal (non-leaf) nodes



descending down the hierarchy to a subdirectory

⇳

traversing an edge down to a child node

DISCLAIMER: directories contain links back to their parent directories, so not strictly a tree

Recursively listing files

to traverse an arbitrary directory structure, need recursion

to list a file system object (either a directory or file):

1. print the name of the current object
2. if the object is a directory, then
   - recursively list each file system object in the directory

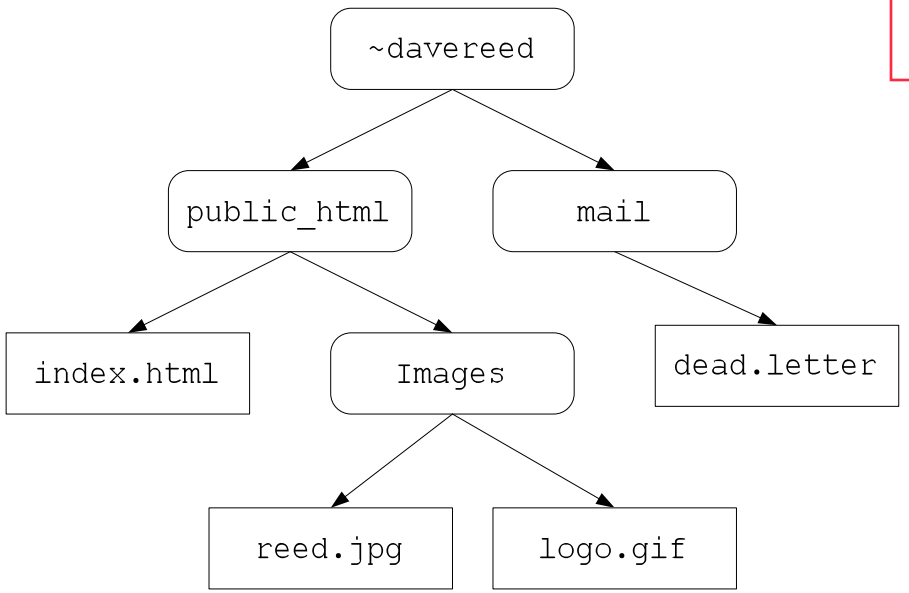
in pseudocode:

```
public static void ListAll(FileSystemObject current) {
   System.out.println(current.getName());
   if (current.isDirectory()) {
     for (FileSystemObject obj : current.getContents()) {
       ListAll(obj);
     }
   }
}
```

Recursively listing files

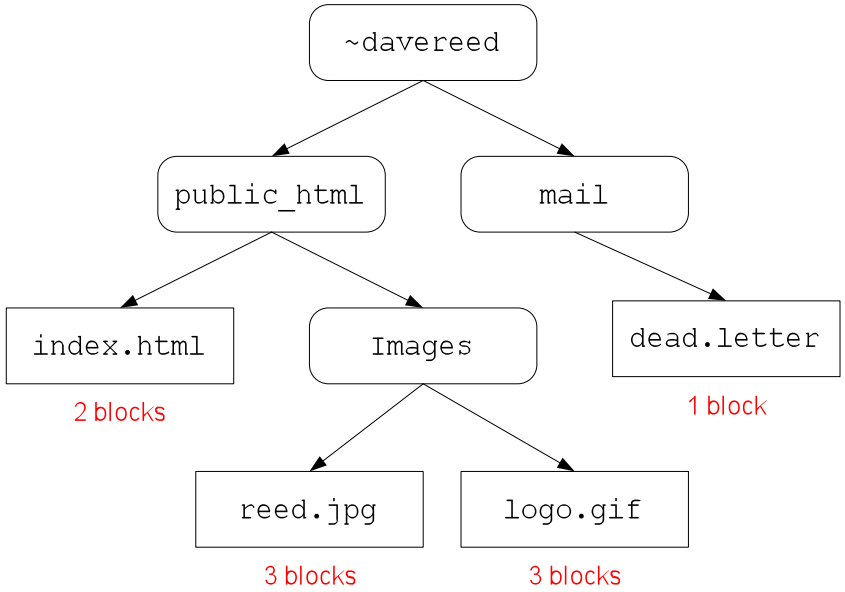
```
public static void ListAll(FileSystemObject current) {
   System.out.println(current.getName());
   if (current.isDirectory()) {
     for (FileSystemObject obj : current.getContents()) {
       ListAll(obj);
     }
   }
}
```

this method performs a *pre-order traversal*: prints the root first, then the subtrees



UNIX `du` command

in UNIX, the du command lists the size of all files and directories



from the `~davereed` directory:

```
unix> du -a
2  ./public_html/index.html
3  ./public_html/Images/reed.jpg
3  ./public_html/Images/logo.gif
7  ./public_html/Images
10 ./public_html
1  ./mail/dead.letter
2  ./mail
13 .
```

```
public static int du(FileSystemObject current) {
   int size = current.blockSize();
   if (current.isDirectory()) {
     for (FileSystemObject obj : current.getContents()) {
       size += du(obj);
     }
   }
   System.out.println(size + " " + current.getName());
   return size;
}
```

this method performs a *post-order traversal*: prints the subtrees first, then the root

How deep is a balanced tree?

CLAIM: A binary tree with height H can store up to 2^H-1 nodes.

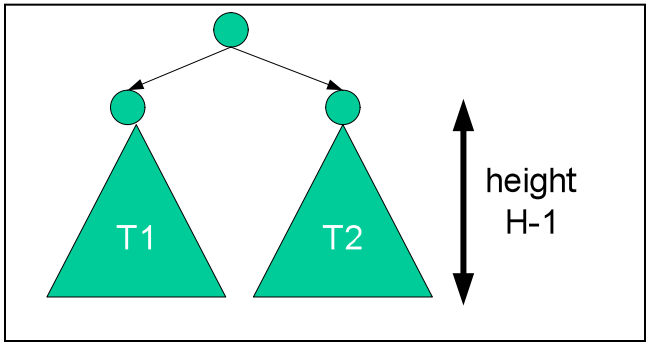
Proof (by induction):

BASE CASES: when H = 0, 2^0 - 1 = 0 nodes ✓

when H = 1, 2^1 - 1 = 1 node ✓

HYPOTHESIS: Assume for all h < H, e.g., a tree with height H-1 can store up to 2^{H-1} -1 nodes.

INDUCTIVE STEP: A tree with height H has a root and subtrees with height up to H-1.



By our hypothesis, T1 and T2 can each store 2^{H-1} -1 nodes, so tree with height H can store up to

$1 + (2^{H-1} - 1) + (2^{H-1} - 1) =$

$2^{H-1} + 2^{H-1} - 1 =$

2^H -1 nodes ✓

equivalently: N nodes can be stored in a binary tree of height $\lceil \log_2(N+1) \rceil$

Trees & recursion

since trees are recursive structures, most tree traversal and manipulation operations are also recursive

- can divide a tree into root + left subtree + right subtree
- most tree operations handle the root as a special case, then recursively process the subtrees

- e.g., to display all the values in a (nonempty) binary tree, divide into
  1. *displaying the root*
  2. *(recursively) displaying all the values in the left subtree*
  3. *(recursively) displaying all the values in the right subtree*

- e.g., to count number of nodes in a (nonempty) binary tree, divide into
  1. *(recursively) counting the nodes in the left subtree*
  2. *(recursively) counting the nodes in the right subtree*
  3. *adding the two counts + 1 for the root*

BinaryTree class

```
public class BinaryTree<E> {
    protected class TreeNode<E> {
        …
    }
    protected TreeNode<E> root;

    public BinaryTree() {
        this.root = null;
    }

    public void add(E value) { … }

    public boolean remove(E value) { … }

    public boolean contains(E value) { … }

    public int size() { … }

    public String toString() { … }
}
```

to implement a binary tree, need to link together tree nodes

- the root of the tree is maintained in a field (initially null for empty tree)
- the root field is "protected" instead of "private" to allow for inheritance
- *recall:* a protected field is accessible to derived classes, otherwise private

TreeNode class

```
protected class TreeNode<E> {
    private E data;
    private TreeNode<E> left;
    private TreeNode<E> right;

    public TreeNode(E d, TreeNode<E> l, TreeNode<E> r) {
        this.data = d;
        this.left = l;
        this.right = r;
    }

    public E getData() { return this.data; }

    public TreeNode<E> getLeft() { return this.left; }

    public TreeNode<E> getRight() { return this.right; }

    public void setData(E newData) { this.data = newData; }

    public void setLeft(TreeNode<E> newLeft) {
        this.left = newLeft;
    }

    public void setRight(TreeNode<E> newRight) {
        this.right = newRight;
    }
}
```

virtually same as DNode class

- change the field & method names to reflect the orientation of nodes
- uses left/right instead of previous/next

`size` method

recursive approach:

BASE CASE: if the tree is empty, number of nodes is 0

RECURSIVE: otherwise, number of nodes is
(# nodes in left subtree) + (# nodes in right subtree) + 1 for the root

note: a recursive implementation requires passing the root as parameter

- will have a public "front" method, which calls the recursive "worker" method

```
public int size() {
    return this.size(this.root);
}

private int size(TreeNode<E> current) {
    if (current == null) {
        return 0;
    }
    else {
        return this.size(current.getLeft()) +
               this.size(current.getRight()) + 1;
    }
}
```

`contains` method

recursive approach:

BASE CASE: if the tree is empty, the item is not found

BASE CASE: otherwise, if the item is at the root, then found

RECURSIVE: otherwise, search the left and then right subtrees

```
public boolean contains(E value) {
    return this.contains(this.root, value);
}

private boolean contains(TreeNode<E> current, E value) {
    if (current == null) {
        return false;
    }
    else {
        return value.equals(current.getData()) ||
               this.contains(current.getLeft(), value) ||
               this.contains(current.getRight(), value);
    }
}
```

`toString` method

must traverse the entire tree and build a string of the items

- there are numerous patterns that can be used, e.g., in-order traversal

BASE CASE: if the tree is empty, then nothing to traverse

RECURSIVE: recursively traverse the left subtree, then access the root, then recursively traverse the right subtree

```
public String toString() {
    if (this.root == null) {
        return "[]";
    }
    String recStr = this.toString(this.root);
    return "[" + recStr.substring(0,recStr.length()-1) + "]";
}

private String toString(TreeNode<E> current) {
    if (current ==  null) {
        return "";
    }
    return this.toString(current.getLeft()) +
           current.getData().toString() + "," +
           this.toString(current.getRight());
}
```

Alternative traversal algorithms

pre-order traversal:

BASE CASE: if the tree is empty, then nothing to traverse

RECURSIVE: access root, recursively traverse left subtree, then right subtree

```
private String toString(TreeNode<E> current) {
    if (current ==  null) {
        return "";
    }
    return current.getData().toString() + "," +
           this.toString(current.getLeft()) +
           this.toString(current.getRight());
}
```

post-order traversal:

BASE CASE: if the tree is empty, then nothing to traverse

RECURSIVE: recursively traverse left subtree, then right subtree, then root

```
private String toString(TreeNode<E> current) {
    if (current ==  null) {
        return "";
    }
    return this.toString(current.getLeft()) +
           this.toString(current.getRight()) +
           current.getData().toString() + ",";
}
```

Exercises

```
/** @return the number of times value occurs in the tree with specified root */
public int numOccur(TreeNode<E> root, E value) {



}
```

```
/** @return the sum of all the values stored in the tree with specified root */
public int sum(TreeNode<Integer> root) {



}
```

```
/** @return the maximum value in the tree with specified root */
public int max(TreeNode<Integer> root) {


}
```

`add` method

how do you add to a binary tree?

- ideally would like to maintain balance, so (recursively) add to smaller subtree
- big Oh?
- we will consider more efficient approaches for maintaining balance later

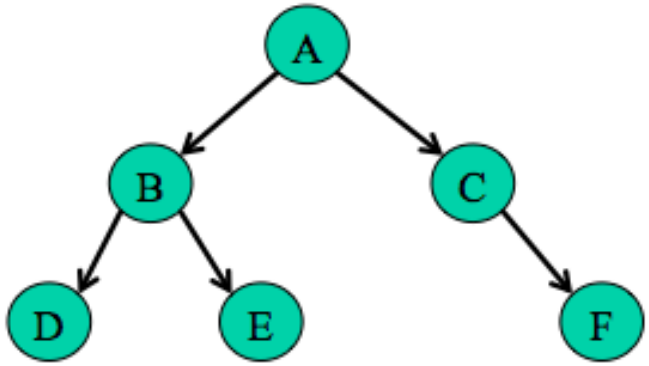
```
public void add(E value) {
    this.root = this.add(this.root, value);
}

private TreeNode<E> add(TreeNode<E> current, E value) {
    if (current == null) {
        current = new TreeNode<E>(value, null, null);
    }
    else if (this.size(current.getLeft()) <= this.size(current.getRight())) {
        current.setLeft(this.add(current.getLeft(), value));
    }
    else {
        current.setRight(this.add(current.getRight(), value));
    }
    return current;
}
```

`remove` method

how do you remove from a binary tree?

- tricky, since removing an internal node means rerouting pointers
- must maintain binary tree structure



simpler solution

1. find node (as in search)
2. if a leaf, simply remove it
3. if no left subtree, reroute parent pointer to right subtree
4. otherwise, replace current value with a leaf value from the left subtree (and remove the leaf node)

DOES THIS MAINTAIN BALANCE?
(you can see the implementation in BinaryTree.java)

Induction and trees

which of the following are true? prove/disprove

- in a full binary tree, there are more nodes on the bottom (deepest) level than all other levels combined

- in any binary tree, there will always be more leaves than non-leaves

- in any binary tree, there will always be more empty children (i.e., null left or right fields within nodes) than children (i.e., non-null fields)

- the number of nodes in a binary tree can be defined by the recurrence relation:

  numNodes(T) = numNodes(left subtree of T) + numNodes(right subtree of T) + 1