CSC 321: Data Structures

Fall 2018

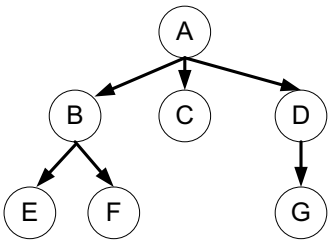
Trees & recursion

- trees, tree recursion
- BinaryTree class
- BST property
- BinarySearchTree class: override add, contains
- search efficiency

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

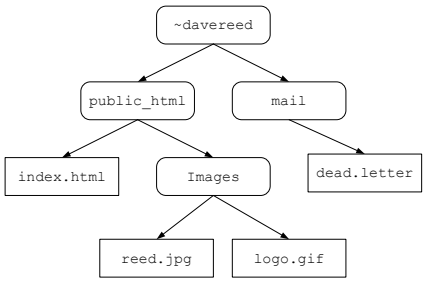
a tree where each node has at most 2 subtrees (children) is a *binary* tree

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:

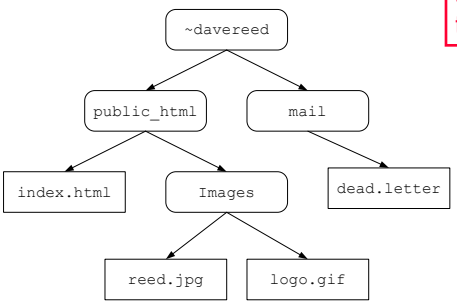
```
public static void ListAll(FileSystemObject current) {
   System.out.println(current.getName());
   if (current.isDirectory()) {
     for (FileSystemObject obj : current.getContents()) {
       ListAll(obj);
     }
   }
}
```

5

Recursively listing files

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

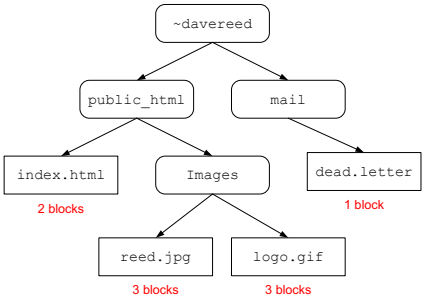
this method performs a *pre-order traversal*: prints the root first, then the subtrees



6

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

7

How deep is a balanced tree?

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

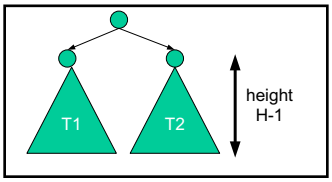
Proof (by induction):

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 $\lceil \log_2(N+1) \rceil$

8

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*

9

BinaryTree class

```
public class BinaryTree<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() { … }

    ////////////////////////////////////

    protected class TreeNode<E> {
        …
    }
}
```

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
- similar to MyLinkedList, TreeNode is an inner class

10

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

11

`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;
    }
}
```

12

`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);
    }
}
```

13

`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() {
    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());
}
```

14

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() + ",";
}
```

15

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) {


}
```

16

`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

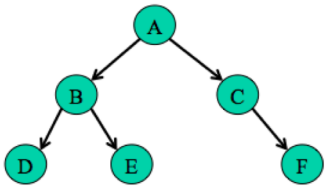
```
public void add(E value) {
    this.root = this.add(this.root, value);
}

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;
}
```

17

`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)

18

Induction and trees

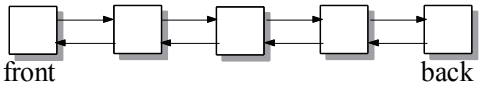
which of the following are true? prove/disprove

- in a full binary tree, there are more nodes on the bottom (deepest) level than all other levels combined
- in any (non-empty) binary tree, there will always be more leaves than non-leaves
- in any (non-empty) 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)

19

Searching linked lists

recall: a (linear) linked list only provides sequential access → O(N) searches



it is possible to obtain O(log N) searches using a tree structure

in order to perform binary search efficiently, must be able to

- access the middle element of the list in O(1)
- divide the list into halves in O(1) and recurse

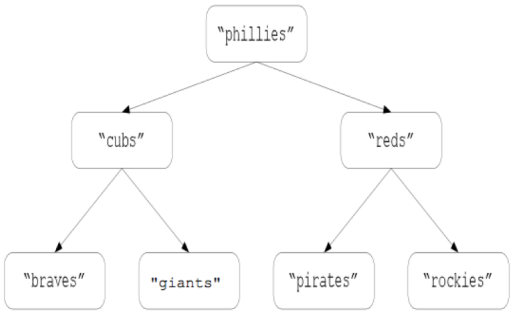
HOW CAN WE GET THIS FUNCTIONALITY FROM A TREE?

20

Binary search trees

a *binary search tree* is a binary tree in which, for every node:

- the item stored at the node is ≥ all items stored in its left subtree
- the item stored at the node is < all items stored in its right subtree



in a (balanced) binary search tree:

- middle element = root
- 1st half of list = left subtree
- 2nd half of list = right subtree

furthermore, these properties hold for each subtree

21

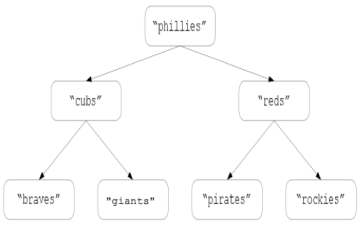
BinarySearchTree class

can use inheritance to derive BinarySearchTree from BinaryTree

```
public class BinarySearchTree<E extends Comparable<? super E>>
extends BinaryTree<E> {

    public BinarySearchTree() {
        super();
    }

    public void add(E value) {
        // OVERRIDE TO MAINTAIN BINARY SEARCH TREE PROPERTY
    }

    public void contains(E value) {
        // OVERRIDE TO TAKE ADVANTAGE OF BINARY SEARCH TREE PROPERTY
    }

    public void remove(E value) {
        // DOES THIS NEED TO BE OVERRIDDEN?
    }
}
```

22

Binary search in BSTs



to search a binary search tree:

1. if the tree is empty, NOT FOUND
2. if desired item is at root, FOUND
3. if desired item < item at root, then recursively search the left subtree
4. if desired item > item at root, then recursively search the right subtree

```
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 if (value.equals(current.getData())) {
        return true;
    }
    else if (value.compareTo(current.getData()) < 0) {
        return this.contains(current.getLeft(), value);
    }
    else {
        return this.contains(current.getRight(), value);
    }
}
```

23

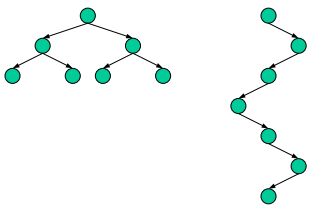
Search efficiency

how efficient is search on a BST?

- in the best case?
  O(1) if desired item is at the root
- in the worst case?
  O(height of the tree) if item is leaf on the longest path from the root

in order to optimize worst-case behavior, want a (relatively) balanced tree

- otherwise, don't get binary reduction
- e.g., consider two trees, each with 7 nodes



24

Search efficiency (cont.)

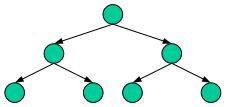
we showed that N nodes can be stored in a binary tree of height $\lceil \log_2(N+1) \rceil$

so, in a balanced binary search tree, searching is O(log N)

N nodes → height of $\lceil \log_2(N+1) \rceil$ → in worst case, have to traverse $\lceil \log_2(N+1) \rceil$ nodes

what about the average-case efficiency of searching a binary search tree?

- assume that a search for each item in the tree is equally likely
- take the cost of searching for each item and average those costs



costs of search

1

2 + 2

3 + 3 + 3 + 3

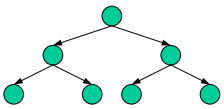
→ 17/7 → 2.42

define the *weight* of a tree to be the sum of all node depths (root = 1, …)

average cost of searching a BST = weight of tree / number of nodes in tree

25

Search efficiency (cont.)



costs of search

1

2 + 2

3 + 3 + 3 + 3

→ 17/7 → 2.42

~log N

costs of search

1
+ 2
+ 3
+ 4
+ 5
+ 6
+ 7

→ 28/7 → 4.00

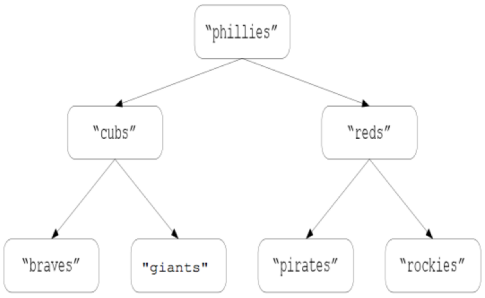
~N/2

26

Inserting an item

inserting into a BST

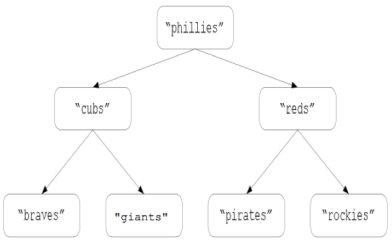
1. traverse edges as in a search
2. when you reach a leaf, add the new node below it




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

private TreeNode<E> add(TreeNode<E> current, E value) {
    if (current == null) {
        return new TreeNode<E>(value, null, null);
    }

    if (value.compareTo(current.getData()) <= 0) {
        current.setLeft(this.add(current.getLeft(), value));
    }
    else {
        current.setRight(this.add(current.getRight(), value));
    }
    return current;
}
```

27

Removing an item



recall BinaryTree remove

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)

CLAIM: as long as you select the rightmost (i.e., maximum) value in the left subtree, this remove algorithm maintains the BST property

WHY?

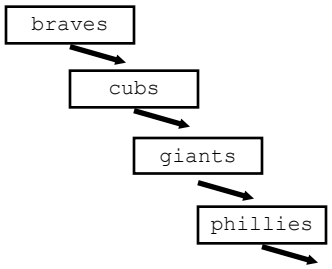
since the BinaryTree remove does this, no need to override

28

Maintaining balance

PROBLEM: random insertions (and removals) do not guarantee balance

- e.g., suppose you started with an empty tree & added words in alphabetical order
  braves, cubs, giants, phillies, pirates, reds, rockies, …



with repeated insertions/removals, can degenerate so that height is O(N)

- specialized algorithms exist to maintain balance & ensure O(log N) height
- or take your chances

29

HW5

you will verify (experimentally) the claim that:

if you add N *random* values to a binary search tree,
the resulting tree will be O(log N) in height and, as a result,
the average cost of searching that tree will also be O(log N).

to do so, you will

- add `height` and `weight` methods to the `BinaryTree` class
- add `avgSearchCost` method to `BinarySearchTree` class

- generate many random trees and calculate the average height and average cost

```
Number of values to be stored: 1000
Number of trees to generate: 100
Generating 100 trees with 1000 random values:
  min possible height = 10
  average tree height = 22.04
  average search cost = 11.53
```

- what patterns would you expect to see?

30