CSC 427: Data Structures and Algorithm Analysis

Fall 2011

Divide & conquer (part 2)

- binary trees
  - standard methods: add, contains, remove, size
  - other methods: numOccur, isLeaf, height, …
- binary search trees
  - BST property
  - overload binary tree methods: add, contains
  - search efficiency, balance

1

Dividing & conquering trees

since trees are recursive structures, most tree traversal and manipulation operations can be classified as *divide & conquer algorithms*

- 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*

2

BinaryTree class

```
public class BinaryTree<E> {
    protected TreeNode 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 store the root node

- the root field is "protected" instead of "private" to allow for inheritance
- the empty tree has a null root
- then, must implement methods for basic operations on the collection

3

`size` method

divide-and-conquer 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;
    }
}
```

4

`contains` method

divide-and-conquer 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);
    }
}
```

5

`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());
}
```

6

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() + ",";
}
```

7

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) {


}
```

8

`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 more consider efficient approaches for maintaining balance later

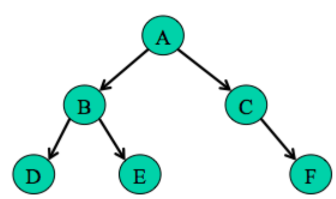
```
public void add(E value) {
    this.root = this.add(this.root, value);
}

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;
}
```

9

`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)

10

HW4: more BinaryTree methods

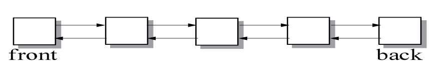
you are to implement and test the following:

- isLeaf: whether an item appears in the tree in a leaf node
- isParent: whether an item appears in the tree in a non-leaf node
- numLeaves: the number of leaf nodes in the tree
- numParents: the number of non-leaf nodes in the tree
- height: the height of the tree (i.e., length of longest path)
- weight: the weight of the tree (i.e., sums of depths of each node)

11

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

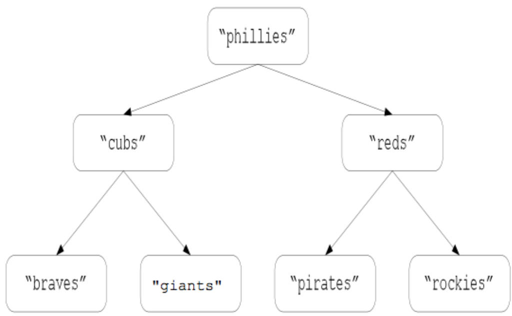
HOW CAN WE GET THIS FUNCTIONALITY FROM A TREE?

12

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

13

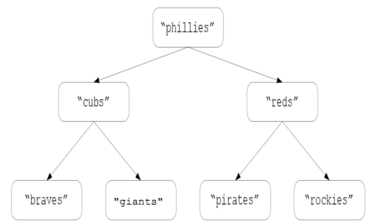
BinarySearchTree class

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?
    }
}
```

14

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);
    }
}
```

15

Search efficiency

how efficient is search on a BST?

- in the best case?

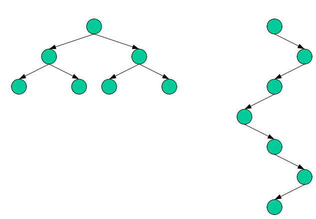
  O(1) if desired item is at the root

- 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



16

How deep is a balanced tree?

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

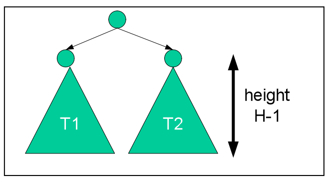
Proof (by induction):

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

when H = 1, $2^1 - 1 = 1$ node ✓

HYPOTHESIS: assume 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 $\lceil \log_2(N+1) \rceil$

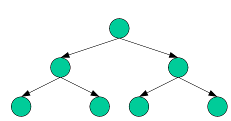
Search efficiency (cont.)

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

N nodes → height of ⌈log2(N+1)⌉ → in worst case, have to traverse ⌈log2(N+1)⌉ 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

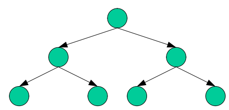
3 + 3 + 3 + 3

→ 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 tree = weight of tree / number of nodes in tree

Search efficiency (cont.)



costs of search

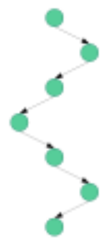
1

2 + 2

3 + 3 + 3 + 3

→ 17/7 → 2.42

~log N



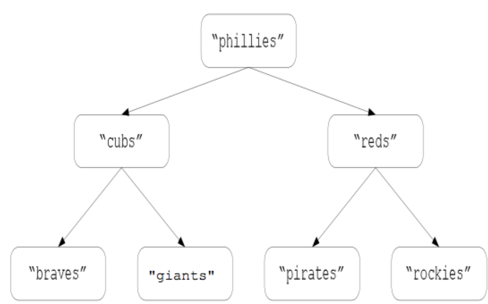
costs of search

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

→ 28/7 → 4.00

~N/2

Inserting an item

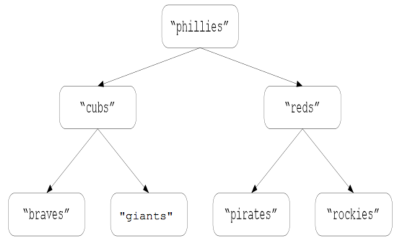


inserting into a BST

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;
}
```

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?

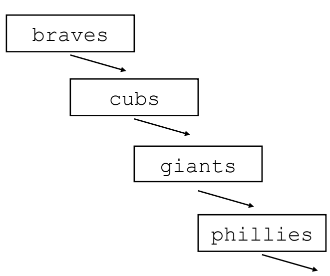
so, no need to override remove

21

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

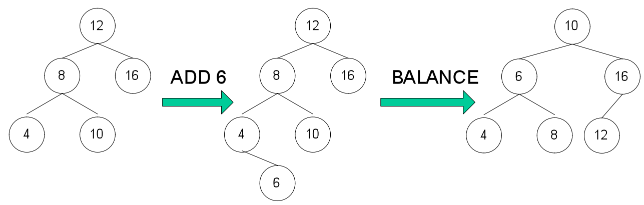
22

Balancing trees

on average, N random insertions into a BST yields O(log N) height

- however, degenerative cases exist (e.g., if data is close to ordered)

we can ensure logarithmic depth by maintaining balance



maintaining full balance can be costly

- however, full balance is not needed to ensure O(log N) operations (LATER)

23