Binary Search Trees

MSCI 240: Algorithms & Data Structures

WATERLOO | ENGINEERING

Engineering Course Critiques

https://evaluate.uwaterloo.ca

- Login using your Quest credentials
- Answer all questions in one sitting
- Hit Submit

UW Course Eval Project Pilot

New This Term!

When you submit your Course Critique, you'll be prompted to complete a second survey (11 questions) for this course; please complete!

Difficulties? Questions?
Contact kabecker@uwaterloo.ca

UNIVERSITY OF WATERLOO

lecture/tutorial swap (Nov 26 → Nov 19; Nov 28 → Dec 3)

- Mon, Nov 19: two lectures—normal lecture + lecture in tutorial
- Mon, Nov 26: tutorial in lecture time
- Wed, Nov 28: no class or tutorial (would be Monday's tutorial)
- Mon, Dec 3: two lectures—normal lecture + lecture in tutorial

slides by Mark Hancock 3

lecture summary

binary search tree (BST) motivation

searching a BST

adding to a BST

`x = change(x);`

complexity of add/search

balancing trees

Topic	Building Java Programs	Algorithms (Sedgewick)
classes, ADTs	chapter 8	1.2
arrays	chapter 7	
ArrayList<T>	chapter 10	1.3
Stack/Queue	chapter 14, (11)	1.3
LinkedList	chapter 16	1.3
Complexity	chapter 13	1.4
Searching		pp. 46-47
Sorting		chapter 2.1-2.3
Recursion	chapter 12	1.1 (p. 25)
Binary Trees	chapter 17	chapter 3.1-3.2
Dictionaries	chapter 18.1	chapter 3.4
Graphs	N/A (Wikipedia good)	chapter 4.1
Heaps/Priority Queues	chapter 18.2	chapter 2.4

motivation

cost of searching:
- linear search – $\Theta(n)$
- binary search – $\Theta(\log n)$,
  - but requires sort $\Theta(n \log n)$
  - alternative: keep array sorted!

cost of insert:
- unsorted – $\Theta(1)$
- sorted – $\Theta(n)$

linear search strategy:

- cost to insert = $\Theta(1)$
- cost to search = $\Theta(n)$

use when #inserts ≫ #searches

binary search strategy:

- cost to insert = $\Theta(n)$
- cost to search = $\Theta(\log n)$

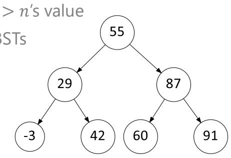
use when #searches ≫ #inserts

can we do insert and search at $\Theta(\log n)$?

a binary search tree (BST) is a binary tree where an in-order traversal will produce a sorted list

binary search tree property—for each node n:

- all elements in n's left subtree are $\leq n$'s value
- all elements in n's right subtree are $> n$'s value
- n's left and right subtrees are also BSTs



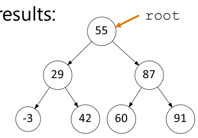
searching a BST

exercise: write a method called `contains` that searches the tree for a given integer, returning `true` if found

this is an instance method in a `SearchTree` class, which has a private member variable called `root`, which is an `IntTreeNode`

if a `SearchTree` variable `tree` referred to the tree below, the following calls would have these results:

```
tree.contains(29) → true
tree.contains(55) → true
tree.contains(63) → false
tree.contains(35) → false
```




```
public boolean contains(int value) {
    return contains(root, value);
}

private boolean contains(IntTreeNode node, int value) {
    if (node == null) {
        return false;
    } else if (node.data == value) {
        return true;
    } else if (value < node.data) {
        return contains(node.left, value);
    } else { // if (value > node.data)
        return contains(node.right, value);
    }
}
```


adding to a BST

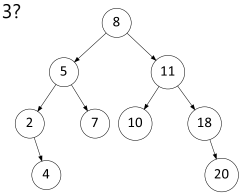
suppose we want to add the value 14 to the BST below
where should the new node be added?

where would we add the value 3?

where would we add 7?

if the tree is empty, where should a new value be added?

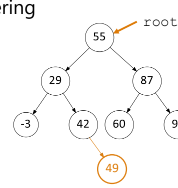
what is the general algorithm?



exercise: add a method `add` to the `SearchTree` class that adds a given integer value to the tree

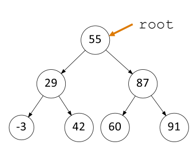
assume that the elements of the `SearchTree` constitute a legal binary search tree, and add the new value in the appropriate place to maintain ordering

```
tree.add(49);
```



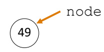

```
// an incorrect solution
public void add(int value) {
    add(root, value);
}

private void add(IntTreeNode node, int value) {
    if (node == null) {
        node = new IntTreeNode(value);
    } else if (value <= node.data) {
        add(node.left, value);
    } else { // if (value > node.data)
        add(node.right, value);
    }
}
```

why doesn't this solution work?



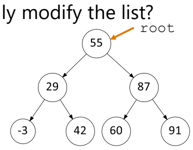
much like with linked lists, if we just modify what a local variable refers to, it won't change the collection

```
private void add(IntTreeNode node, int value) {
    if (node == null) {
        node = new IntTreeNode(value);
    }
    // ...
```



in the linked list case, how did weactually modify the list?

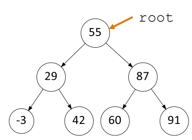
- by changing the `front`
- by changing a node's `next` field



for trees

- change `root`
- change a node's `left`/`right` field

```
// a poor (inelegant), but correct solution
public void add(int value) {
    if (root == null) {
        root = new IntTreeNode(value);
    } else {
        add(root, value);
    }
}

private void add(IntTreeNode node, int value) {
    if (value <= node.data) {
        if (node.left == null) {
            node.left = new IntTreeNode(value);
        } else {
            add(node.left, value);
        }
    } else { // if (value > node.data)
        if (node.right == null) {
            node.right = new IntTreeNode(value);
        } else {
            add(node.right, value);
        }
    }
}
```




```
x = change(x);
```

`String` methods that modify actually return a new one

if we want to modify a string variable, we must re-assign it

```
String s = "lil bow wow";
s.toUpperCase();
System.out.println(s);    // lil bow wow
s = s.toUpperCase();
System.out.println(s);    // LIL BOW WOW
```

we call this general algorithmic pattern **`x = change(x);`**

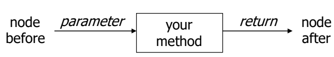
we will use this approach when writing methods that modify the structure of a binary tree

---

methods that modify a tree can use the following pattern:

input (parameter): old state of the node

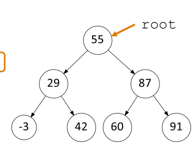
output (return): new state of the node



in order to actually change the tree, you must reassign:

```
root = change(root, parameters);
root.left = change(root.left, parameters);
root.right = change(root.right, parameters);
```

```
// a correct solution
public void add(int value) {
    root = add(root, value);
}

private IntTreeNode add(IntTreeNode node, int value) {
    if (node == null) {
        node = new IntTreeNode(value);
    } else if (value <= node.data) {
        node.left = add(node.left, value);
    } else { // if (value > node.data)
        node.right = add(node.right, value);
    }

    return node;
}
```



think about the case when `root` is a leaf…

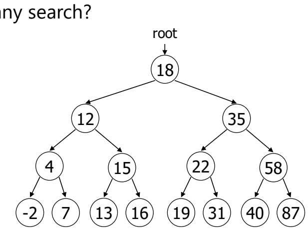
complexity of add/search

searching a BST

what is the maximum number of nodes you would need to examine to perform any search?

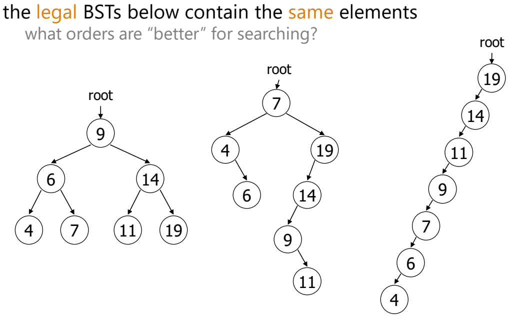
e.g., search for 31?

e.g., search for 6?



the legal BSTs below contain the same elements

what orders are "better" for searching?



what's the worst-case complexity for `add`/`contains`?

what's the average-case complexity for `add`/`contains`?

is there anything we could do to avoid the worst case?

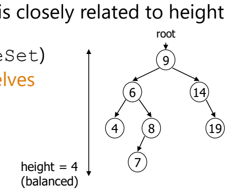
trees and balance

balanced tree: one whose subtrees (a) differ in height by at most 1 and (b) are themselves balanced

- a balanced tree of n nodes has a height of $\sim \log_2 n$
- a very unbalanced tree can have a height close to n

the runtime of `add`/`contains` is closely related to height

some tree collections (e.g. `TreeSet`) contain code to balance themselves as new nodes are added



binary search tree summary

the binary search tree property ensures elements in the left subtree are less than the root and elements in the right subtree are greater than the root

to modify a tree, need to change `root` or the `left`/`right` of some existing node; use the **`x = change(x);`** pattern

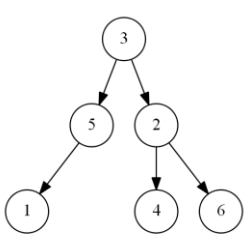
`add`/`contains` can be done in $O(\log n)$ time on balanced trees, but $O(n)$ on unbalanced trees

clicker questions

true or false: this is a valid binary tree

A. true

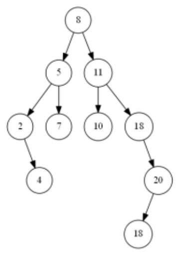
B. false



true or false: this is a valid binary search tree

A. true

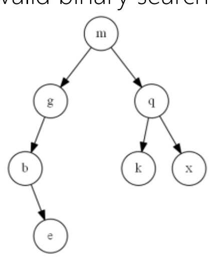
B. false



true or false: this is a valid binary search tree

A. true

B. false



next:

hash map implementations