CSC148: Week 11
http://www.cdf.utoronto.ca/~csc148h/summer/

Sophia Huynh
Summer 2018
Announcements

● Assignment 2 due August 3rd
● Exercise 8 and Lab 8 are up
  ○ Exercise 7 remarks due Sunday @ 11PM
● Midterm marks are released
  ○ Scans + solutions up by the weekend
Office Hours

- Monday: 2 - 5PM (BA5287)
- Thursday: 1- 5 PM (BA3219)

(There might be more on Wednesday; but I'm not sure yet. I'll announce it if there are.)
Outline

● Binary Search Trees
● Efficiency
Binary Trees
Binary Search Tree

- Similar to Binary Trees
- All values in the left subtree < the root
- All values in the right subtree > the root
- Assume we don't have duplicates.
(Not a) Binary Search Tree
(Not a) Binary Search Tree
6 is not > 7, so it can't be in 7's right subtree!
(Not a) Binary Search Tree

All values here must be > 5!
(Not a) Binary Search Tree

All values here must be > 4 (and < 5)!

All values here must be > 8!

1 3 2 7 6 9
Binary Search Tree
Binary Search Tree
Which value could we remove in order to make this a Binary Search Tree?

A) 1  B) 2  C) 4  D) 8
Answer: B) 2
Which value could we remove in order to make this a Binary Search Tree?

A) 1  B) 2  C) 4  D) 8
Where should we insert a 7?
Right-side →

Answer:

D
Where should we insert a 7?
Should we insert a 7?

Must be in 9's left subtree.
Where should we insert a 7?

Must be in 5's right subtree.
Where should we insert a 7?

Right-side →

Must be in 8's left subtree.
Where should we insert a 7?

Still a Binary Search Tree!
Different Trees, Same Values
Different Trees, Same Values

1

2

3

4
Different Trees, Same Values
Different Trees, Same Values
Different Trees, Same Values

● All of them adhere to BST (Binary Search Tree) properties
● More on the effect of different trees later.
class BinarySearchTree

    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

    # Same as a BinaryTree!
contains(t, value)

- Return whether value appears in t or not.
- Do not visit more nodes than necessary!
contains(t, 7)

Is this 7?
contains(t, 7)

No; 7 > 5. Check the right subtree.
contains(t, 7)
contains(t, 7)

No; 7 < 8. Check the left subtree.
contains(t, 7)

We found 7!
contains(t, value)

- Return whether value appears in t or not.
- Do not visit more nodes than necessary!
BinaryTree's contains()

if t is None:
    return False

if t.value == value:
    return True

return contains(t.left, value) or contains(t.right, value)
BinaryTree's contains()

```python
if t is None:
    return False
if t.value == value:
    return True
return contains(t.left, value) or contains(t.right, value)
```

Looks at both left and right subtrees!
contains()

if t is None:
    return False

if t.value == value:
    return True

Use the same base case for BSTs.
contains()

if t is None:
  return False

if t.value == value:
  return True

if value < t.value:
  return contains(t.left, value)

If the value is < the root, we only check the left subtree.
contains()

if t is None:
    return False

if t.value == value:
    return True

if value < t.value:
    return contains(t.left, value)

if value > t.value:
    return contains(t.right, value)
insert(t, value)

- Insert value into t
- For a normal BinaryTree: we could insert a value anywhere.
  - Insert is weird to write in that case, without providing rules.
- For a BST: There's only 1 place to insert into.
Examples

insert(None, 4)
Examples

insert(t, 2)

4
Examples

```
insert(t, 2)
```

![Diagram showing tree structure with nodes labeled 2 and 4 connected by an edge.]
Examples

insert(t, 3)
insert(t, value)

- Insert so that it maintains BST properties
- Return the root of the BST
  - This lets our recursion work smoothly and to handle the case of None (i.e. we want to return the new BinarySearchTree created).
insert(t, value)

if t is None:
    return BinarySearchTree(value)

- Assume there are no duplicates.
- Return the root of the BST.
**insert(t, value)**

- 2 cases to consider:
  - value < t.value
  - value > t.value
insert(t, value)

- 2 cases to consider:
  - value < t.value
    - Insert into the left subtree!
    - t.left should be whatever's returned by insert(t.left, value)
  - value > t.value
insert(t, value)

if t is None:
    return BinarySearchTree(value)

if value < t.value:
    t.left = insert(t.left, value)
insert(t, value)

if t is None:
    return BinarySearchTree(value)

if value < t.value:
    t.left = insert(t.left, value)
else:
    t.right = insert(t.right, value)
def insert(t, value):
    if t is None:
        return BinarySearchTree(value)
    if value < t.value:
        t.left = insert(t.left, value)
    else:
        t.right = insert(t.right, value)

    return t
find_max(t)

- Return the largest value in t.
- Assume t is not None
- Use BST properties to your advantage!
  - Left values < t.value < right values
find_max(t)

- Return the largest value in t.
- Assume t is not None
- Use BST properties to your advantage!
  - Left values < t.value < right values
find_max(t)

if t.right:
    return find_max(t.right)
def find_max(t):
    if t.right:
        return find_max(t.right)
    return t.value
find_max(t)

if t.right:
    return find_max(t.right)

return t.value

This is the same as
if t.right is not None
Aside: if t

- Python tries to compare any non-boolean values to True/False
  - Non-empty values are considered True
  - Empty values are considered False
Aside: if t

x = 5

if x:
    print("Yes")
else:
    print("No")
Aside: if t

x = 5

if x:
    print("Yes")
else:
    print("No")

"Yes" gets printed since 5 is non-empty.
Aside: if t

```python
x = 0
if x:
    print("Yes")
else:
    print("No")
```
Aside: if t

x = 0

if x:
    print("Yes")
else:
    print("No")

"No" gets printed since 0 is considered empty.
Aside: if t

- Empty values:
  - 0
  - None
  - Empty list []
  - Empty dictionary {}
  - Empty string ''
- Everything else is 'non-empty'.
if t.right

This is the same as:

if t.right is not None
if not t.right

This is the same as:

if t.right is None
if not t.right

This is the same as:

if t.right is None

Saves me a bit of space/time.
find_max(t)

if t.right:
    return find_max(t.right)

return t.value
delete(t, value)

- Delete from BST and return the root
- t is None?
- value < t.value?
- value > t.value?
- value == t.value and...
  - No children?
  - Only a left or a right child?
  - Both children?
delete(t, value)

- Delete from BST and return the root
- t is None? **There's nothing to delete!**
- value < t.value?
- value > t.value?
- value == t.value and...
  - No children?
  - Only a left or a right child?
  - Both children?
delete(t, value)

if t is None:
    return None
delete(t, value)

- Delete from BST and return the root
- t is None? There's nothing to delete!
- value < t.value? **Delete from t.left**
- value > t.value? **Delete from t.right**
- value == t.value and...
  - No children?
  - Only a left or a right child?
  - Both children?
delete(t, value)

if t is None:
    return None

if value < t.value:
    t.left = delete(t.left, value)

elif value > t.value:
    t.right = delete(t.right, value)
delete(t, value)

- Delete from BST and return the root
- t is None? There's nothing to delete!
- value < t.value? Delete from t.left
- value > t.value? Delete from t.right
- value == t.value and...
  - No children?
  - Only a left or a right child?
  - Both children?
delete(t, 3)
delete(t, 3)
delete(t, 2)
delete(t, 2)
delete(t, 1)
delete(t, 1)
delete(t, 3)

For a leaf, we just return None.
def delete(t, value):
    if t is None:
        return None

    if value < t.value:
        t.left = delete(t.left, value)
    elif value > t.value:
        t.right = delete(t.right, value)
    else:
        if not t.left and not t.right:
            return None

    return t
delete(t, 2)

3 would become the new root
def delete(t, value):
    else:
        if not t.left and not t.right:
            return None
        if t.left and not t.right:
            return t.left
        if t.right and not t.left:
            return t.right
If we delete 5, what would our new root be?
**delete(t, value)**

- If there are 2 children:
  - Delete, but want to keep our BST properties.
  - Whatever we replace the root with should be > everything in the left subtree and < everything in the right subtree
delete(t, value)

Replace t.value with:

- The largest value in the left subtree
- The smallest value in the right subtree
**delete(t, value)**

Replace t.value with:

- **The largest value in the left subtree**
  
  We wrote find_max() so let's use this.
  
  or

- **The smallest value in the right subtree**
delete(t, value)

else:
    if not t.left and not t.right:
        return None
    if t.left and not t.right:
        return t.left
    if t.right and not t.left:
        return t.right
    max_value = find_max(t.left)

    # Placeholder for max_value
delete(t, 5)

4 is the max of the left subtree.
delete(t, 5)

4 is the max of the left subtree.
delete(t, value)

else:
    if not t.left and not t.right:
        return None
    if t.left and not t.right:
        return t.left
    if t.right and not t.left:
        return t.right
    max_value = find_max(t.left)
    t.value = max_value
delete(t, 5)

Make the root's value 4.
delete(t, 5)

Delete 4 from the left subtree.
delete(t, value)

else:
    if not t.left and not t.right:
        return None
    if t.left and not t.right:
        return t.left
    if t.right and not t.left:
        return t.right
    max_value = find_max(t.left)
    t.value = max_value
    t.left = delete(t.left, max_value)
delete(t, value)

else:
    if not t.left and not t.right:
        return None
    if t.left and not t.right:
        return t.left
    if t.right and not t.left:
        return t.right
    max_value = find_max(t.left)
    t.value = max_value
    t.left = delete(t.left, max_value)

    return t
Break
Resume in 10 minutes :)
Runtime Analysis

[5, 3, 2, 1, 4]

"How long would calling contains() take (relative to the size of the list):"
list contains

for item in lst:
    if item == value:
        return True

return False
Runtime Analysis

[5, 3, 2, 1, 4]

"How long would calling contains() take (relative to the size of the list)?"

● At best: How many items do we need to look at?
● At worst?
Runtime Analysis

[5, 3, 2, 1, 4]

"How long would calling contains() take (relative to the size of the list)?"

● At best: How many items do we need to look at? **One item: if the first item == value, we return instantly.**

● At worst?
"How long would calling `contains()` take (relative to the size of the list)?"

- **At best:** How many items do we need to look at? One item: if the first item \(==\) value, we return instantly.
- **At worst?** All of the items! I.e. looking for a number not in the list.
Runtime Analysis

- Contains for a list:
  - Best case: 1st item
  - Worst case: $n$ items

- We care about how much time is taken relative to the size of our input (list, tree, etc.)
  - "If our input has a size of $n$, how does our runtime scale?"
Runtime Analysis

- O(1): Constant time
  - No matter how the size of our input changes, we always perform the same number of steps.
Runtime Analysis

- **O(1): Constant time**
  - No matter how the size of our input changes, we always perform the same number of steps.

- **O(n): Linear time**
  - The amount of work we do is linear in proportion to the size of our input.
Runtime Analysis

- $O(n^2)$: Quadratic time
  - The amount of work we do grows quadratically relative to our input size.
  - i.e. For every item we look at, we have to look at every other item too.
Runtime Analysis

- $O(lgn)$: Logarithmic time
  - The size of the input grows exponentially with regards to the work we do.
Runtime Analysis

- **O(lgn)**: Logarithmic time
  - The size of the input grows exponentially with regards to the work we do.

- **O(nlgn)**:
  - A bit more than linear time. I'll talk more about this next week.

- **O(2^n)**: Exponential time
  - Exponential work relative to our input.
Runtime Analysis

● More detail next week
● For now: "Different programs can take different amounts of time to run."
Binary Search Tree Runtimes

1. How many nodes do we have to visit (relative to the size of our entire BST)?
2. How much work do we have to do at each node (relative to the size of our entire BST)?

Runtime is (1) * (2).
Nodes visited for contains(t, 6)
Nodes visited for contains(t, 6)
Nodes visited for contains(t, 10)
Worst case: we check through the longest path (= height of the tree)
Nodes visited for contains(t, value)
contains(t, value)

2x the nodes, but we still only check 4 of them.
Runtime of contains()

- We visit at most [height of our BST] nodes
- Checking the value of a node is $O(1)$ time (constant)
- Worst case runtime for contains:

\[ O(\text{height of BST} \times 1) = O(\text{height of BST}) \]
Balance of BSTs

Completely unbalanced!
Height == Number of nodes
Balance of BSTs

Completely balanced! Can't make this BST any smaller!
Heights of BSTs

- For a BST with $n$ nodes
- Completely unbalanced:
  - Height $= O(n)$
- Completely balanced:
  - Height $= O(\log(n))$
$O(lg(n))$ 

- 7 nodes
- Height of 3
- $lg(7)$ is roughly 2.8
  - Rounds up to 3
$O(\log(n))$

- Consider how many nodes we can have at each level
- 1st level: 1 node
- 2nd level: 2 nodes ($1 \times 2$)
- 3rd level: 4 nodes ($2 \times 2$)

$2^{(i-1)}$ nodes at level $i$
"How many levels do we need?"
- Height of 1: 1 node
- Height of 2: 3 nodes
- Height of 3: 7 nodes
$O(\lg(n))$

- Equivalently: "How many times can we divide by 2"
- $7 / 2 == 3.5$
- $3.5 / 2 == 1.75$
- $1.75 / 2 == 0.875$
- 3 times in total! (Or $\lg(n)$)
  - $\lg(n) == \log_2(n)$ -- We like base 2 in CS :)
Things to Remember

● **The best height for a BST is** \( \lg(n) \)
  ○ This is for a completely balanced BST

● **The worst height is** \( n \)
  ○ This is for a completely unbalanced BST

● **The runtime is the number of nodes we have to look at** *the amount of work done at each node*
How many nodes do we have to look at to insert 3 into this BST?

A) 2  B) 3  C) 4
Left-side

Answer:

C) 4
How many nodes do we have to look at to insert 3 into this BST?

A) 2  B) 3  C) 4
How many nodes do we have to look at to insert 6 into this BST?

A) 3  B) 4  C) 5
Answer:

C) 5
How many nodes do we have to look at to insert 6 into this BST?

A) 3    B) 4    C) 5
Binary Search Trees

- We know exactly where to search for something!
Searching in a Binary Tree
Searching in a Binary Tree

Have to check every subtree!
Searching in a Binary Search Tree
Searching in a Binary Search Tree

We know exactly where to look!
Searching

- **Binary Tree:** $O(n)$ always
- **Binary Search Tree:** $O(\text{height})$
  - $O(n)$ for an unbalanced BST
  - $O(\lg(n))$ for a balanced BST
- **If we have 1000 nodes, that's the difference between checking 1000 nodes and \~10 nodes!**
Efficiency

- More about efficiency next week!
- Sorting algorithms, queues, stacks, linked lists, etc.
  - You know of Bubble, Selection, and Insertion Sort. These took $O(n^2)$ time...
  - But we can do better (using recursion)!
Homework

- Assignment 2 due next Friday
  - Office hours on Monday + Thursday
- Exercise 8 is out (due Thursday)
  - Lab 8 is out
- Midterm marks are out
  - Scans will be up over the weekend