

# DSC 40B

## *Theoretical Foundations II*

Lecture 8 | Part 1

### Dynamic Sets

# Bookkeeping

- ▶ How do you store your books?

# Bookkeeping

- How do you store your books?



# Bookkeeping

- How do you store your books?



# Bookkeeping: Tradeoffs

- ▶ Messy:
  - ▶ No upfront cost.
  - ▶ Cost to search is high.
- ▶ Organized
  - ▶ Big upfront cost.
  - ▶ Cost to search is low.
- ▶ “Right” choice depends on how often we search.

# Data Structures and Algorithms

- ▶ **Data structures** are ways of organizing data to make certain operations faster.
- ▶ Come with an upfront cost (preprocessing).
- ▶ “Right” choice of data structure depends on what operations we’ll be doing in the future.

# Queries: Easy to Hard

- ▶ We've been thinking about **queries**.
  - ▶ Given a collection of data, is  $x$  in the collection?
- ▶ Querying is a fundamental operation.
  - ▶ Useful in a data science sense.
  - ▶ But also frequently performed in algorithms.
- ▶ There are several situations to think about.

## Situation #1: Static Set, One Query

- ▶ **Given:** an unsorted collection of  $n$  numbers (or strings, etc.).
- ▶ In future, you will be asked single query.
- ▶ Which is better: linear search or sort + binary search?



# Situation #1: Static Set, One Query

- ▶ **Given:** an unsorted collection of  $n$  numbers (or strings, etc.).
- ▶ In future, you will be asked single query.
- ▶ Which is better: linear search or sort + binary search?
  - ▶ Linear search:  $\Theta(n)$  worst case.
  - ▶ Binary search would require sorting first in  $\Theta(n \log n)$  worst case

## Situation #2: Static Set, Many Queries

- ▶ **Given:** an unsorted collection of  $n$  numbers (or strings, etc.).
- ▶ In future, you will be asked **many** queries.
- ▶ Which is better: linear search or sort + binary search?
  - ▶ Depends on number of queries!

## Exercise

Suppose you have a static set of  $n$  items. How long will it take<sup>a</sup> to perform  $k$  queries in total with:

1. linear search?
2. sort + binary search?

If  $k = n/10$ , which should you use?

What if  $k = \log n$ ?

---

<sup>a</sup>On average. Assume the best case is rare.

## Situation #3: Dynamic Set, Many Queries

- ▶ **Given:** a collection of  $n$  numbers (or strings, etc.).
- ▶ In future, you will be asked **many** queries *and* to **insert** new elements.
- ▶ Best approach: ?

# Binary Search?

- ▶ Can we still use binary search?
- ▶ **Problem:** To us binary search, we must maintain array in sorted order as we insert new elements.
- ▶ Inserting into array takes  $\Theta(n)$  time in worst case.
  - ▶ Must “make room” for new element.
  - ▶ Can we use linked list with binary search?

# Today

- ▶ Introduce (or review) **binary search trees**.
- ▶ BSTs support fast queries *and* insertions.
- ▶ Preserve sorted order of data after insertion.
- ▶ Can be modified to solve many problems efficiently.
  - ▶ Example: finding order statistics.

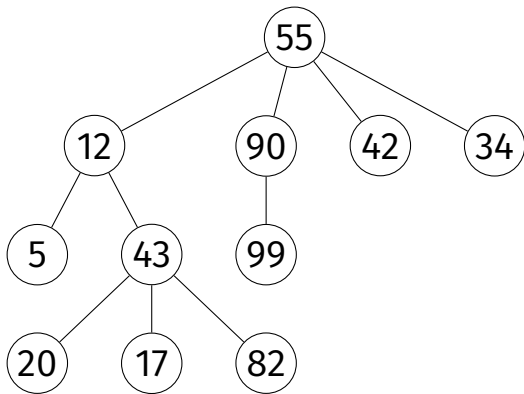
# DSC 40B

## *Theoretical Foundations II*

Lecture 8 | Part 2

### Binary Search Trees

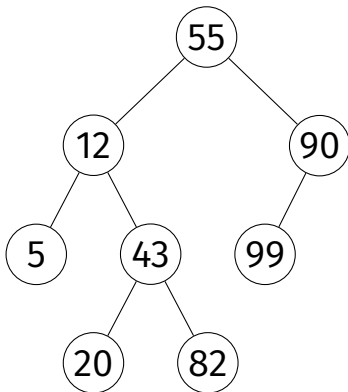
# Trees





# Binary Trees

- Each node has *at most* two children (left and right).



# Binary Search Tree

- ▶ A **binary search tree** (BST) is a binary tree that satisfies the following for any node  $x$ :
- ▶ if  $y$  is in  $x$ 's **left** subtree:

$$y.\text{key} \leq x.\text{key}$$

- ▶ if  $y$  is in  $x$ 's **right** subtree:

$$y.\text{key} \geq x.\text{key}$$

# Assumption (for simplicity)

- ▶ We'll assume keys are unique (no duplicates).
- ▶ if  $y$  is in  $x$ 's **left** subtree:

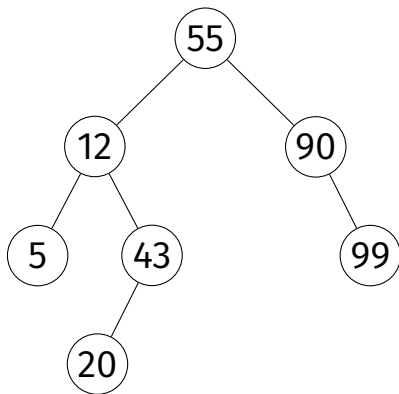
$$y.key < x.key$$

- ▶ if  $y$  is in  $x$ 's **right** subtree:

$$y.key > x.key$$

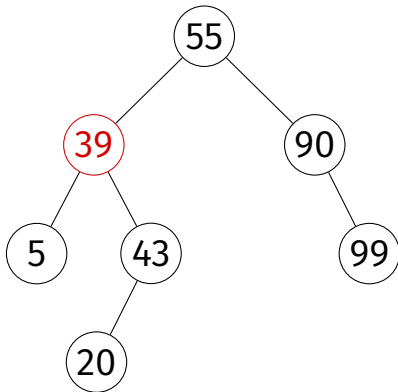
# Example

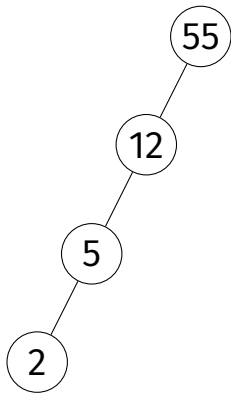
► This **is** a BST.



# Example

- This is **not** a BST.





## Exercise

Is this is a BST?

# Height

- ▶ The **height** of a tree is the number of edges from the root to any leaf.
- ▶ Suppose a binary tree has  $n$  nodes.
- ▶ The **tallest** it can be is  $\approx n$
- ▶ The **shortest** it can be is  $\approx \log_2 n$

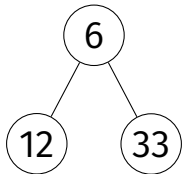
# In Python

```
class Node:
    def __init__(self, key, parent=None):
        self.key = key
        self.parent = parent
        self.left = None
        self.right = None

class BinarySearchTree:
    def __init__(self, root: Node):
        self.root = root
```



# In Python



```
root = Node(6)
n1 = Node(12, parent=root)
root.left = n1
n2 = Node(33, parent=root)
root.right = n2
tree = BinarySearchTree(root)
```

# DSC 40B

## *Theoretical Foundations II*

Lecture 8 | Part 3

### Queries and Insertions in BSTs

# Why?

- ▶ BSTs impose structure on data.
- ▶ “Not quite sorted”.
- ▶ Preprocessing for making insertions *and* queries faster.

# Operations on BSTs

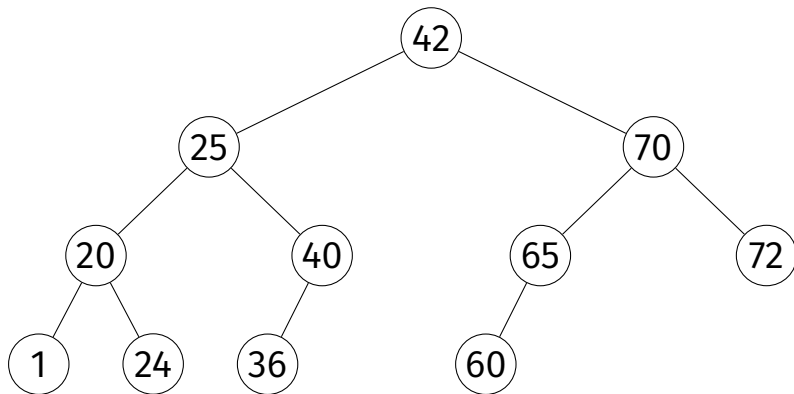
- ▶ We will want to:
  - ▶ **query** a key (is it in the tree?)
  - ▶ **insert** a new key

# Queries

- ▶ **Given:** a BST and a target,  $t$ .
- ▶ **Return:** **True** or **False**, is the target in the collection?

# Queries

► Is 36 in the tree? 65? 23?



# Queries

- ▶ Start walking from root.
- ▶ If current node is:
  - ▶ equal to target, return **True**;
  - ▶ too large ( $>$  target), follow left edge;
  - ▶ too small ( $<$  target), follow right edge;
  - ▶ **None**, return **False**

# Queries, in Python

```
def query(self, target):  
    """As method of BinarySearchTree."""  
    current_node = self.root  
    while current_node is not None:  
        if current_node.key == target:  
            return current_node  
        elif current_node.key < target:  
            current_node = current_node.right  
        else:  
            current_node = current_node.left  
    return None
```



## Exercise

Complete the recursive version of query.

```
def query_recursive(node, target):  
    """As a 'free function'."""  
    if node is None:  
        return False  
  
    if node.key == target:  
        ...  
    elif ...:  
        ...  
    else:  
        ...
```

# Queries (Recursive)

```
def query_recursive(node, target):  
    """As a 'free function'."""  
    if node is None:  
        return False  
  
    if node.key == target:  
        return node  
    elif node.key < target:  
        return query_recursive(node.right, target)  
    else:  
        return query_recursive(node.left, target)
```

# Queries, Analyzed

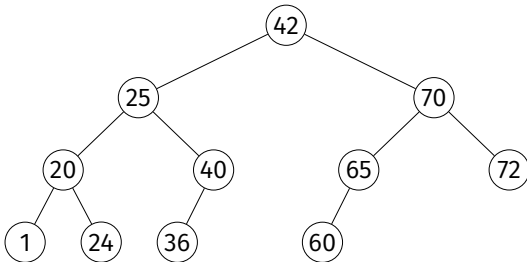
- ▶ Best case:  $\Theta(1)$ .
- ▶ Worst case:  $\Theta(h)$ , where  $h$  is **height** of tree.

# Insertion

- ▶ **Given:** a BST and a new key,  $k$ .
- ▶ **Modify:** the BST, inserting  $k$ .
- ▶ Must **maintain** the BST properties.

# Insertion

- Insert 23 into the BST.



## Insertion (The Idea)

- ▶ Traverse the tree as in query to find empty spot where new key should go, keeping track of last node seen.
- ▶ Create new node; make last node seen the parent, update parent's children.
- ▶ Be careful about inserting into empty tree!

```
def insert(self, new_key):  
    # assume new_key is unique  
    current_node = self.root  
    parent = None  
  
    # find place to insert the new node  
    while current_node is not None:  
        parent = current_node  
        if current_node.key < new_key:  
            current_node = current_node.right  
        else: # current_node.key > new_key  
            current_node = current_node.left  
  
    # create the new node  
    new_node = Node(key=new_key, parent=parent)  
  
    # if parent is None, this is the root. Otherwise, update the  
    # parent's left or right child as appropriate  
    if parent is None:  
        self.root = new_node  
    elif parent.key < new_key:  
        parent.right = new_node  
    else:  
        parent.left = new_node
```

# Insertion, Analyzed

- ▶ Worst case:  $\Theta(h)$ , where  $h$  is **height** of tree.



## Main Idea

Querying and insertion take  $\Theta(h)$  time in the worst case, where  $h$  is the height of the tree.

# DSC 40B

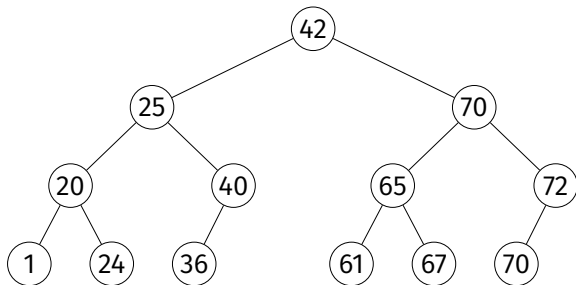
## *Theoretical Foundations II*

Lecture 8 | Part 4

### Balanced and Unbalanced BSTs

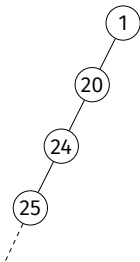
# Binary Tree Height

- ▶ In case of very balanced tree,  $h = \Theta(\log n)$ .
  - ▶ Query, insertion take worst case  $\Theta(\log n)$  time in a **balanced** tree.



# Binary Tree Height

- ▶ In the case of very unbalanced tree,  $h = \Theta(n)$ .
  - ▶ Query, insertion take worst case  $\Theta(n)$  time in **unbalanced** trees.



# Unbalanced Trees

- ▶ Occurs if we insert items in (close to) sorted or reverse sorted order.
- ▶ This is a **common** situation.

# Example

- ▶ Insert 1, 2, 3, 4, 5, 6, 7, 8 (in that order).

# Time Complexities

query	$\Theta(h)$
insertion	$\Theta(h)$

Where  $h$  is height, and  $h = \Omega(\log n)$  and  $h = O(n)$ .

## Time Complexities (Balanced)

query	$O(\log n)$
insertion	$O(\log n)$

Where  $h$  is height, and  $h = \Omega(\log n)$  and  $h = O(n)$ .



# Worst Case Time Complexities (Unbalanced)

query	$\Theta(n)$
insertion	$\Theta(n)$

- ▶ The worst case is **bad**.
  - ▶ Worse than using a sorted array!
- ▶ The worst case is **not rare**.

## Main Idea

The operations take linear time in the worst case **unless** we can somehow ensure that the tree is **balanced**.

# Self-Balancing Trees

- ▶ There are variants of BSTs that are **self-balancing**.
  - ▶ Red-Black Trees, AVL Trees, etc.
- ▶ Quite complicated to implement correctly.
- ▶ But their height is **guaranteed** to be  $\sim \log n$ .
- ▶ So insertion, query take  $\Theta(\log n)$  in worst case.

## Warning!

If asked for the time complexity of a BST operation, be careful! A common mistake is to say that insertion/query are  $\Theta(\log n)$  without being told that the tree is balanced.

## Main Idea

In general, insertion/query take  $\Theta(h)$  time in worst case. If tree is balanced,  $h = \Theta(\log n)$ , so they take  $\Theta(\log n)$  time. If tree is badly unbalanced,  $h = O(n)$ , and they can take  $O(n)$  time.

# DSC 40B

## *Theoretical Foundations II*

Lecture 8 | Part 5

**Augmenting BSTs**

# Modifying BSTs

- ▶ Perhaps more than most other data structures, BSTs must be modified (**augmented**) to solve unique problems.

# Order Statistics

- ▶ Given  $n$  numbers, the  **$k$ th order statistic** is the  $k$ th smallest number in the collection.



# Example

[99, 42, -77, -12, 101]

- ▶ 1st order statistic:
- ▶ 2nd order statistic:
- ▶ 4th order statistic:

# Dynamic Set, Many Order Statistics

- ▶ Quickselect finds any order statistic in linear expected time.
- ▶ This is efficient for a static set.
- ▶ Inefficient if set is dynamic.

# Goal

- ▶ Create a **dynamic** set data structure that supports fast computation of **any** order statistic.

# BST Solution

- ▶ For each node, keep attribute `.size`, containing # of nodes in subtree rooted at current node
- ▶ Property:<sup>1</sup>  
$$x.size = x.left.size + x.right.size + 1$$

---

<sup>1</sup>If a left or right child doesn't exist, consider its size zero.

# Computing Sizes

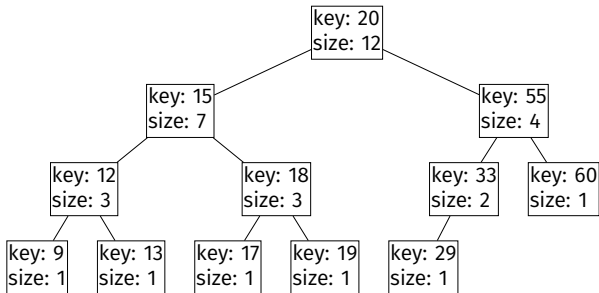
```
def add_sizes_to_tree(node):  
    if node is None:  
        return 0  
    left_size = add_sizes_to_tree(node.left)  
    right_size = add_sizes_to_tree(node.right)  
    node.size = left_size + right_size + 1  
    return node.size
```

## Note

- ▶ Also need to maintain size upon inserting a node.

# Computing Order Statistics

► 8th? 2nd? 12th



# Augmenting Data Structures

- ▶ This is just one example, but many more.
- ▶ Understanding how BSTs work is key to augmenting them.