# DSC 190

## Data Structures & Algorithms

Lecture 4 | Part 1

## Dynamic Sets and Hashing

# Dynamic Set

- One of the most useful abstract data types.
- A collection of unique keys which supports:
  - insertion and deletion
  - membership queries: `x in set`
- Very similar to **dictionary**.

# Implementation #1

- Store $n$ elements in a dynamic array.
- Initial cost: $\Theta(n)$.
- Query: linear search, $O(n)$.
- Insertion: $\Theta(1)$ amortized.

# Implementation #2

- Store $n$ elements in a **sorted** dynamic array.
- Initial cost: $O(n \log n)$.
- Query: binary search, $\Theta(\log n)$.
- Insertion: $O(n)$
  - Must maintain sorted order, involves copies.

# Better Implementation

- Store $n$ elements in a **hash table**.
- Initial cost: $\Theta(n)$[1].
- Query: $\Theta(1)$.
- Insertion: $\Theta(1)$.

[1] All time complexities are average case.

# Today's Lecture

- We'll review hashing.
- See where hashing is **not** the right thing to do.
- Review binary search trees as an alternative.
- Next lecture: introduce **treaps**.

# Hashing

- One of the most important ideas in CS.
- Tons of uses:
  - Verifying message integrity.
  - Fast queries on a large data set.
  - Identify if file has changed in version control.

# Hash Function

- A **hash function** takes a (large) object and returns a (smaller) “fingerprint” of that object.

# How?

- Looking at certain bits, combining them in ways that look random.

# Hash Function Properties

- Hashing same thing twice returns the same hash.
- Unlikely that different things have same fingerprint.
  - But not impossible!

# Example

- MD5 is a **cryptographic** hash function.
  - Hard to “reverse engineer” input from hash.
- Returns a *really large* number in hex.

  `a741d8524a853cf83ca21eabf8cea190`

- Used to “fingerprint” whole files.

# Example

```
> echo ”My name is Justin” | md5
a741d8524a853cf83ca21eabf8cea190
> echo ”My name is Justin” | md5
a741d8524a853cf83ca21eabf8cea190
> echo ”My name is Justin!” | md5
f11eed2391bbd0a5a2355397c089fafd
```

# Another Use

- Want to place images into 100 bins.
- How do we decide which bin an image goes into?
- Hash function!
  - Takes in an image.
  - Outputs a number in {1, 2, ..., 100}.

# Hashing for Data Scientists

- Don’t need to know much about *how* hash function works.
- But should know how they are used.

# Hash Tables

- Create an array with pointers to $m$ linked lists.
  - Usually $m \approx$ number of things you'll be storing.

- Create hash function to turn input into a number in $\{0, 1, \ldots, m - 1\}$.

# Example

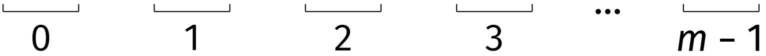
```
hash('hello') == 3
hash('data') == 0
hash('science') == 4
```

# Collisions

- The **universe** is the set of all possible inputs.
- This is usually much larger than $m$ (even infinite).
- Not possible to assign each input to a unique bin.
- If `hash(a) == hash(b)`, there is a **collision**.

# Chaining

- Collisions stored in same bin, in linked list.
- **Query**: Hash to find bin, then linear search.



# The Idea

- A good hash function will utilize all bins evenly.
  - Looks like uniform random distribution.
- If $m \approx n$, then only a few elements in each bin.
- As we add more elements, we need to add bins.

# Average Case

- $n$ elements in bin.
- $m$ bins.
- Assume elements placed randomly in bins[2].
- Expected bin size: $n/m$.

---

[2] Of course, they are placed deterministically.

# Analysis

- Query:
  - $\Theta(1)$ to find bin
  - $\Theta(n/m)$ for linear search.
  - Total: $\Theta(1 + n/m)$.
  - We usually guarantee $m = O(n)$, $\implies \Theta(1)$.
- Insertion: $\Theta(1)$.

# Worst Case

- Everything hashed to same bin.
  - Really unlikely!
  - Adversarial attack?

- Query:
  - $\Theta(1)$ to find bin
  - $\Theta(n)$ for linear search.
  - Total: $\Theta(n)$.

# Worst Case Insertion

- We need to ensure that $m \leq c \cdot n$.
  - Otherwise, too many collisions.

- If we add a bunch of elements, we'll need to increase $m$.

- Increasing $m$ means allocating a new array, $\Theta(m) = \Theta(n)$ time.

**Main Idea**

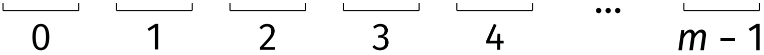
Hash tables support constant (expected) time insertion and membership queries.

# Hashing Downsides

- Hashing is like magic. Constant time access?!
- Comes at a cost: data now scattered “randomly”.
- Examples:
  - find max/min in hash table.
  - range query: all strings between `'a'` and `'c'`
- Must do a full loop over table!

# Example

```
hash('apple') == 3
hash('bill nye') == 0
hash('cassoway') == 4
```



# DSC 190

## Data Structures & Algorithms

Lecture 4 | Part 2

## Binary Search Trees

# Binary Search Trees

- An alternative way to implement dynamic sets.
- Slightly slower insertion, query.
- But preserves data in sorted order.

# 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:

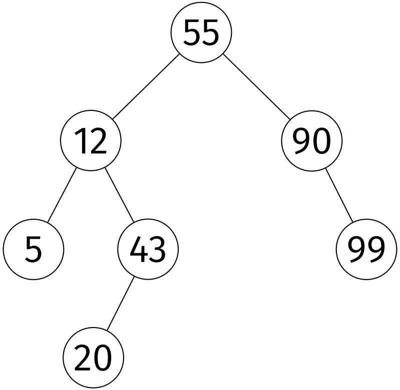
$$y.key \leq x.key$$

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

$$y.key \geq x.key$$

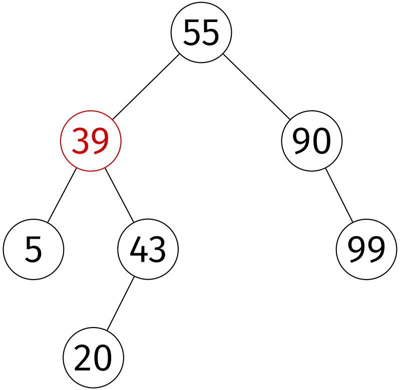
# Example

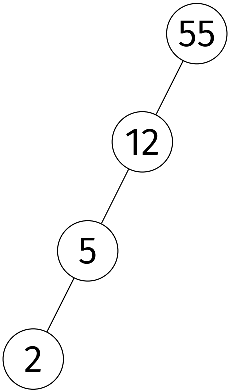
- This **is** a BST.



# Example

- This is **not** a BST.



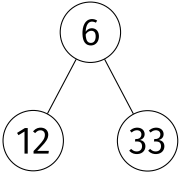



**Exercise**

Is this is a BST?

# Memory Representation

- Each element stored as a **node** at an arbitrary address in memory.
- Each node has a **key**[3] and pointers to **left child**, **right child**, and **parent** nodes (if they exist).



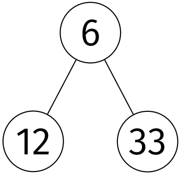

[3]We'll assume keys are unique, though this can be relaxed.

# In Python

```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

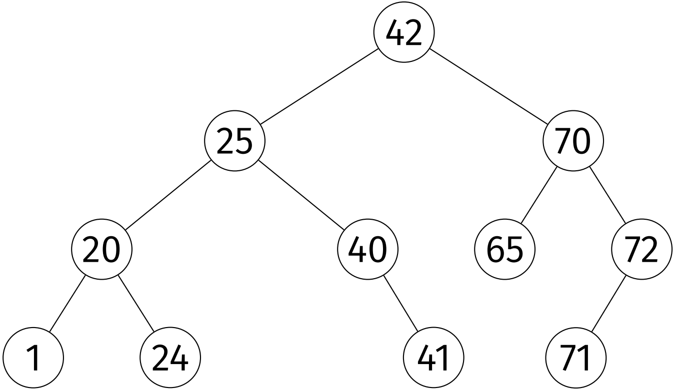



```
root = Node(6)
n1 = Node(12, parent=root)
root.left = n1
n2 = Node(33, parent=root)
root.right = n2
tree = BinarySearchTree(root)
```

# Operations on BSTs

- We will want to:
  - **traverse** the nodes in sorted order by key
  - **query** a key (is it in the tree?)
  - **insert** a new key
  - **delete** an existing key

# Inorder Traversal



```python
def inorder(node):
    if node is not None:
        inorder(node.left)
        print(node.key)
        inorder(node.right)
```

# Inorder Traversal

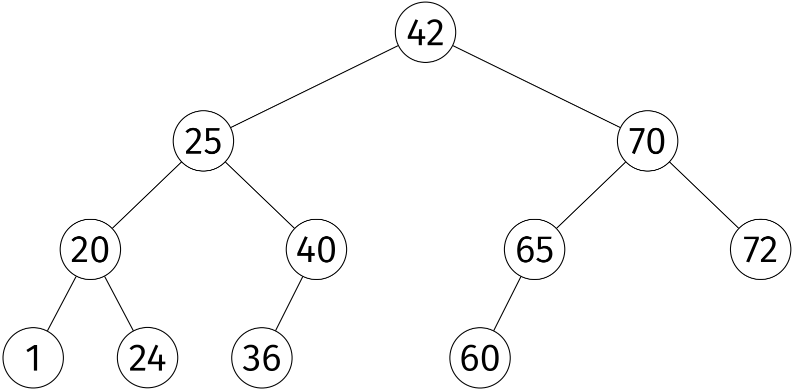
- Prints nodes in sorted order.
- Visits each node once, $\Theta(1)$ time in the call.
- Takes $\Theta(n)$ time.

# 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

```python
def query(self, target):
    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
```

# Queries, Analyzed

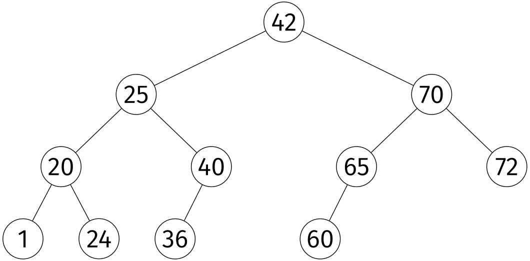
- Best case: $\Theta(1)$.
- Worst case: $\Theta(h)$, where $h$ is **height** of tree.

# 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.



```python
def insert(self, new_key):
    # assume new_key is unique
    current_node = self.root
    parent = None

    while current_node is not None:
        parent = current_node
        if current_node.key == new_key:
            raise ValueError(f'Duplicate key ”{new_key}” not allowed.')
        if current_node.key < new_key:
            current_node = current_node.right
        elif current_node.key > new_key:
            current_node = current_node.left

    new_node = Node(key=new_key, parent=parent)
    if parent is None:
        self.root = new_node
    elif parent.key < new_key:
        parent.right = new_node
    else:
        parent.left = new_node
```

# Insertion, Analyzed

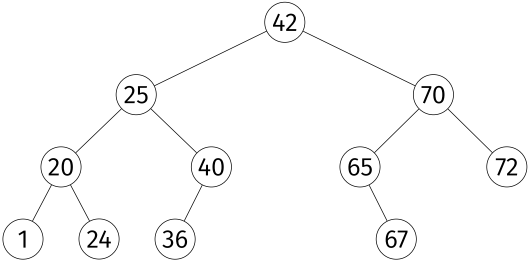
- Worst case: $\Theta(h)$, where $h$ is **height** of tree.

# Deletion

- **Given**: a key in the BST.
- **Modify**: the BST, deleting the key.
- Must **maintain** the BST properties.
- This is a little trickier.

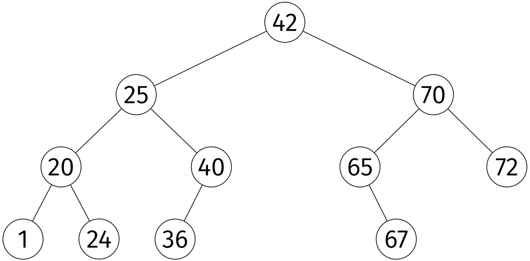
# Deletion: Case 1 (Easy)

- Delete 36 from the BST.



# Deletion: Case 2 (Tricky)

- Delete 42 from the BST.

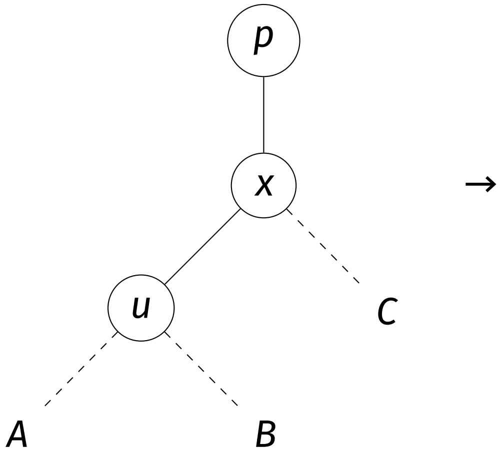


# Deletion

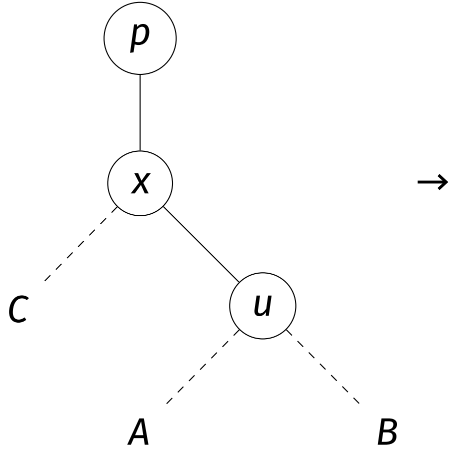
- If node has no children (leaf), **easy**.
- Otherwise, a little trickier.
- Idea: **rotate**[4] node to bottom, preserving BST. When it is a leaf, delete.

[4] Most books take a different approach with the same time complexity.

# (Right) Rotation



# (Left) Rotation



**Claim**

Left rotate and right rotate preserve the BST property.

```python
def _right_rotate(self, x):
    u = x.left
    B = u.right
    C = x.right
    p = x.parent

    x.left = B
    if B is not None: B.parent = x

    u.right = x
    x.parent = u

    u.parent = p

    if p is None:
        self.root = u
    elif p.left is x:
        p.left = u
    else:
        p.right = u
```

# Deletion Analyzed

- Each rotate takes $\Theta(1)$ time.
- $O(h)$ rotations until node becomes leaf.
- So $\Theta(h)$ time in the worst case.

## Main Idea

Insertion, deletion, and querying all take $\Theta(h)$ time in the worst case, where $h$ is the height of the tree.

# DSC 190

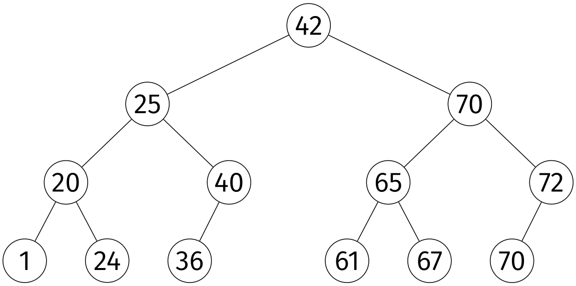
## DATA STRUCTURES & ALGORITHMS

Lecture 4 | Part 3

# Balanced and Unbalanced BSTs

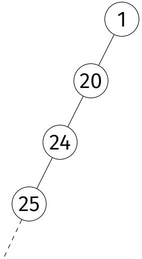
# Binary Tree Height

- In case of very balanced tree, $h$ grows **logarithmically** with $n$.
  - $h = \Theta(\log n)$
  - Query, insertion, deletion take worst case $\Theta(\log n)$ time.



# Binary Tree Height

- In the case of very unbalanced tree, $h$ grows **linearly** with $n$.
  - $h = \Theta(\log n)$
  - Query, insertion, deletion take worst case $\Theta(n)$ time.



# 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**.

# DSC 190

## Data Structures & Algorithms

Lecture 4 | Part 4

## Range Queries, Max, and Min

# Why use a BST?

- Even assuming a balanced tree, BSTs seem worse than hash tables.

| | BST | Hash Table[5] |
|---|---|---|
| query | $O(\log n)$ | $\Theta(1)$ |
| insertion | $O(\log n)$ | $\Theta(1)$ |

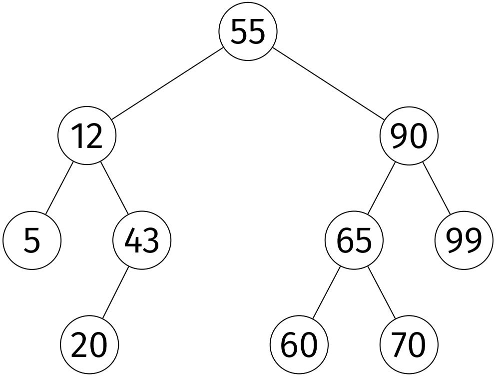
- So when are BSTs better?

[5] Average case times reported.

# Max/Min

- Consider finding the maximum element.
- Hash tables: $\Theta(n)$; must loop through all bins.
- BST: $\Theta(h)$, which is $O(\log n)$ if balanced

# Example



## Main Idea

Keeping track of the maximum can be done efficiently in any stream of numbers, provided that there are only **insertions**. But if **deletions** are allowed, BSTs can find the *next* maximum efficiently.

**Exercise**

How well do heaps work for this problem? Are they better? In what sense?

# Range Queries

- **Given:** a collection and an interval $[a, b]$
- **Retrieve:** all elements in the interval.
- Example:
  - collection: 55, 12, 5, 43, 20, 90, 65, 99, 60, 70
  - interval: [1, 30]
  - result: 5, 12, 20

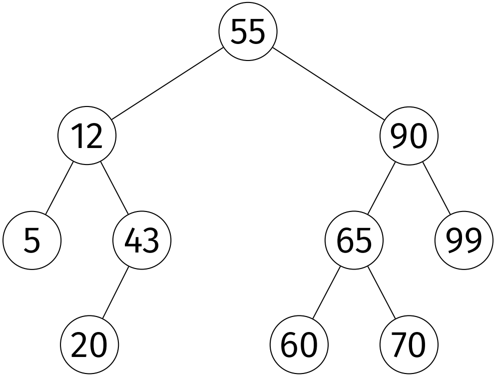
## Exercise

How quickly can this be performed with a hash table?

# Range Queries in BST

- Definitions:
  - The **ceiling** of $x$ in a BST is the smallest key ≥ $x$.
  - The **successor** of node $u$ is the smallest node > $x$.
- Strategy:
  - Find the **floor** of $a$
  - Repeatedly find the **successor** until > $b$

# Example



# Range Queries

- **ceiling** and **successor** both take $O(h) = O(\log n)$ in balanced trees
- If the are $k$ elements in the range, calling `successor` $k$ times gives complexity $O(k \log n)$.