# CS 225

**Data Structures**

## March 19 – AVL Applications

Brad Solomon

# Informal Early Feedback Reminder

## CS 225 All SP21: Data Structures (Evans, C)

Dashboard / My courses / CS 225 All SP21

- Announcements
- Lab Attendance
- Informal Early Feedback
- Lab Informal Early Feedback

# Learning Objectives

- Review Big O in the contexts of an AVL tree
- Formalize relationship between $n$ and $h$ in an AVL tree
- Prove $h$ has an upper bound of $O(log\ n)$
- Wrap up balanced binary trees

# AVL Tree Analysis

**For AVL tree of height h, we know:**

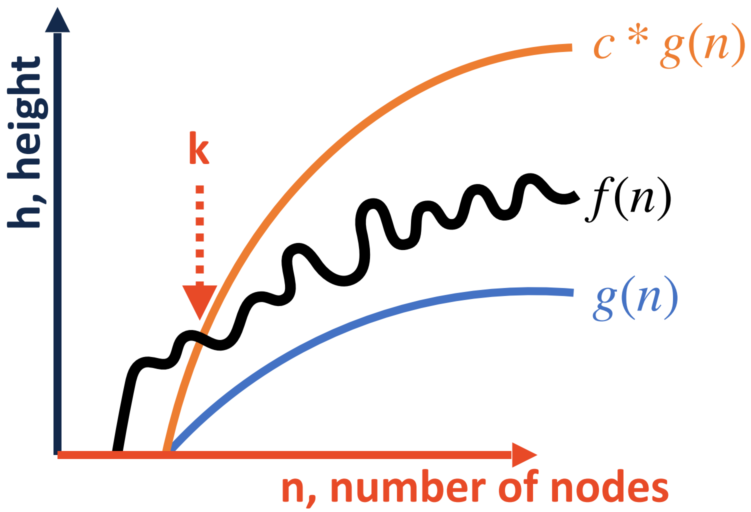
find runs in: ____________.

insert runs in: ____________.

remove runs in: ____________.

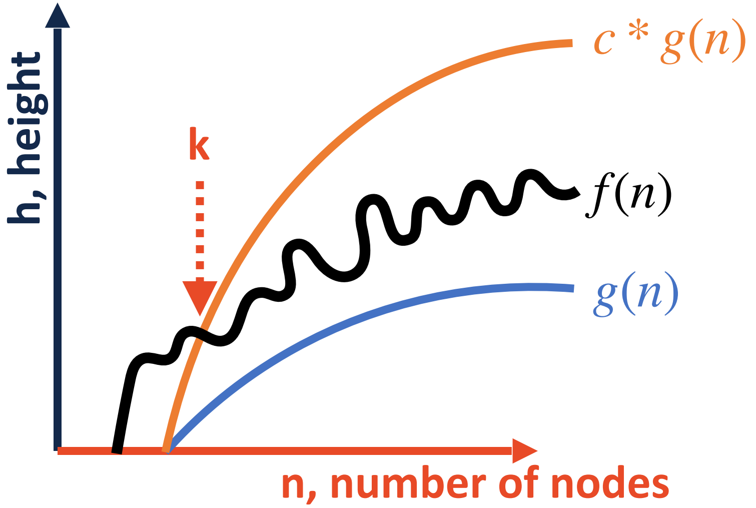
**We will argue that: h** is ___________.

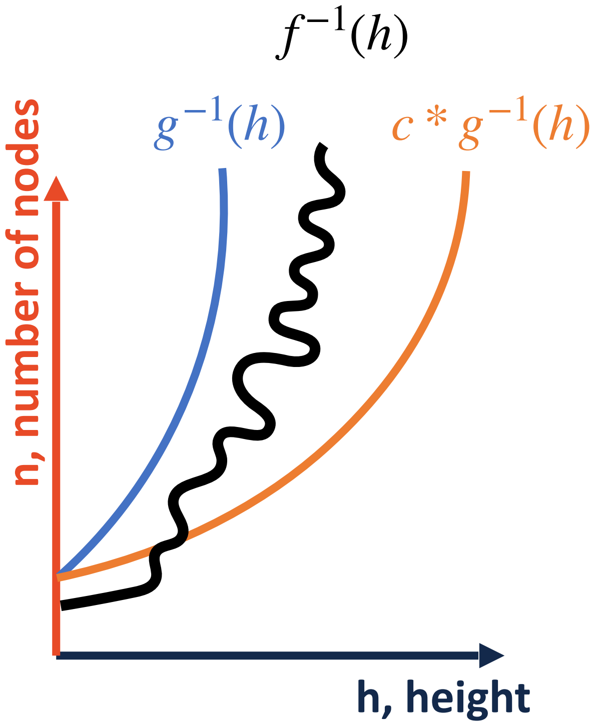
# AVL Tree Analysis




The height of the tree, **f(n)**, will always be <u>less than</u> **c × g(n)** for all values where **n > k**.

# AVL Tree Analysis




$f(n)$ = "Tree height given nodes"

$f^{-1}(h)$ = "Nodes in tree given height"

The number of nodes in the tree, $\mathbf{f^{-1}(h)}$, will always be <u>greater than</u> $\mathbf{c \times g^{-1}(h)}$ for all values where $\mathbf{n > k}$.

# Plan of Action

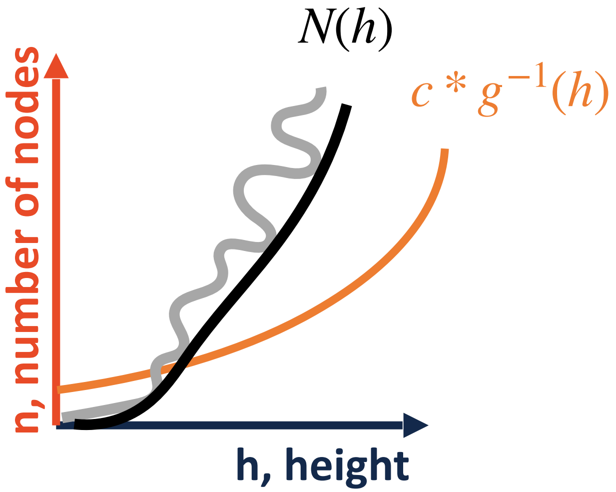
Since our goal is to find the lower bound on **n** given **h**, we can begin by defining a function given **h** which describes the smallest number of nodes in an AVL tree of height **h**:

$N(h)$ = minimum number of nodes in an AVL tree of height $h$

# Simplify the Recurrence

$$N(h) = 1 + N(h-1) + N(h-2)$$

$$N(h) \geq N(h) - 1$$



# State a Theorem

**Theorem:** An AVL tree of height h has at least ___________.

**Proof by Induction:**

I. Consider an AVL tree and let **h** denote its height.

II. Base Case: ________________

An AVL tree of height ____ has at least ____ nodes.

# Prove a Theorem

III. Base Case: ________________

An AVL tree of height ____ has at least ____ nodes.

# Prove a Theorem

IV. Induction Case: ___________________

If for all heights $i < h$, $N(i) \geq 2^{i/2}$

then we must show for height $h$ that $N(h) \geq 2^{h/2}$

# Prove a Theorem

V. Using a proof by induction, we have shown that:

…and inverting:

# AVL Runtime Proof

An upper-bound on the height of an AVL tree is **O( lg(n) )**:

**N(h)** := Minimum # of nodes in an AVL tree of height h

**N(h)** = 1 + **N(h-1)** + **N(h-2)**

$> 1 + 2^{h-1/2} + 2^{h-2/2}$

$> 2 \times 2^{h-2/2} = 2^{h-2/2+1} = 2^{h/2}$

**Theorem #1:**

**Every AVL tree of height h has at least $2^{h/2}$ nodes.**

# AVL Runtime Proof

An upper-bound on the height of an AVL tree is **O( lg(n) )**:

**# of nodes (n) ≥ N(h) > $2^{h/2}$**

**$n > 2^{h/2}$**

**$\lg(n) > h/2$**

**$2 \times \lg(n) > h$**

**$h < 2 \times \lg(n)$** ***, for $h \geq 1$***

**Proved: The maximum number of nodes in an AVL tree of height h is less than $2 \times \lg(n)$.**

# Summary of Balanced BST

**AVL Trees**

- Max height: 1.44 * lg(n)
- Rotations:

# Summary of Balanced BST

**AVL Trees**

- Max height: 1.44 * lg(n)
- Rotations:

  Zero rotations on find
  One rotation on insert
  O(**h**) == O(**lg(n)**) rotations on remove

**Red-Black Trees**

- Max height: 2 * lg(n)
- Constant number of rotations on insert (max 2), remove (max 3).

# Red-Black Trees in C++

C++ provides us a balanced BST as part of the standard library:

```
std::map<K, V> map;
```

**Lookup**

| | |
|---|---|
| count | returns the number of elements matching specific key (public member function) |
| find | finds element with specific key (public member function) |
| contains (C++20) | checks if the container contains element with specific key (public member function) |
| equal_range | returns range of elements matching a specific key (public member function) |
| lower_bound | returns an iterator to the first element *not less* than the given key (public member function) |
| upper_bound | returns an iterator to the first element *greater* than the given key (public member function) |

**Modifiers**

| | |
|---|---|
| clear | clears the contents (public member function) |
| insert | inserts elements or node (public member function) |
| insert_or_assign (C++17) | inserts an element or ass (public member function) |
| emplace (C++11) | constructs element in-pla (public member function) |
| emplace_hint (C++11) | constructs elements in-p (public member function) |
| try_emplace (C++17) | inserts in-place if the key does not exist, does nothing if the key exists (public member function) |
| erase | erases elements (public member function) |
| swap | swaps the contents (public member function) |
| extract (C++17) | extracts nodes from the container (public member function) |
| merge (C++17) | splices nodes from another container (public member function) |

# Red-Black Trees in C++

```
V & std::map<K, V>::operator[]( const K & )
```

```
void std::map<K, V>::erase( const K & )
```

# Why Balanced BST?

# Summary of Balanced BST

**Pros:**

- Running Time:

  - Improvement Over:

- Great for specific applications:

# Summary of Balanced BST

**Cons:**

- Running Time:

- In-memory Requirement:

# Trees in the Real World

**Q:** Can we always fit our data in main memory?

**Q:** Where else can we keep our data?

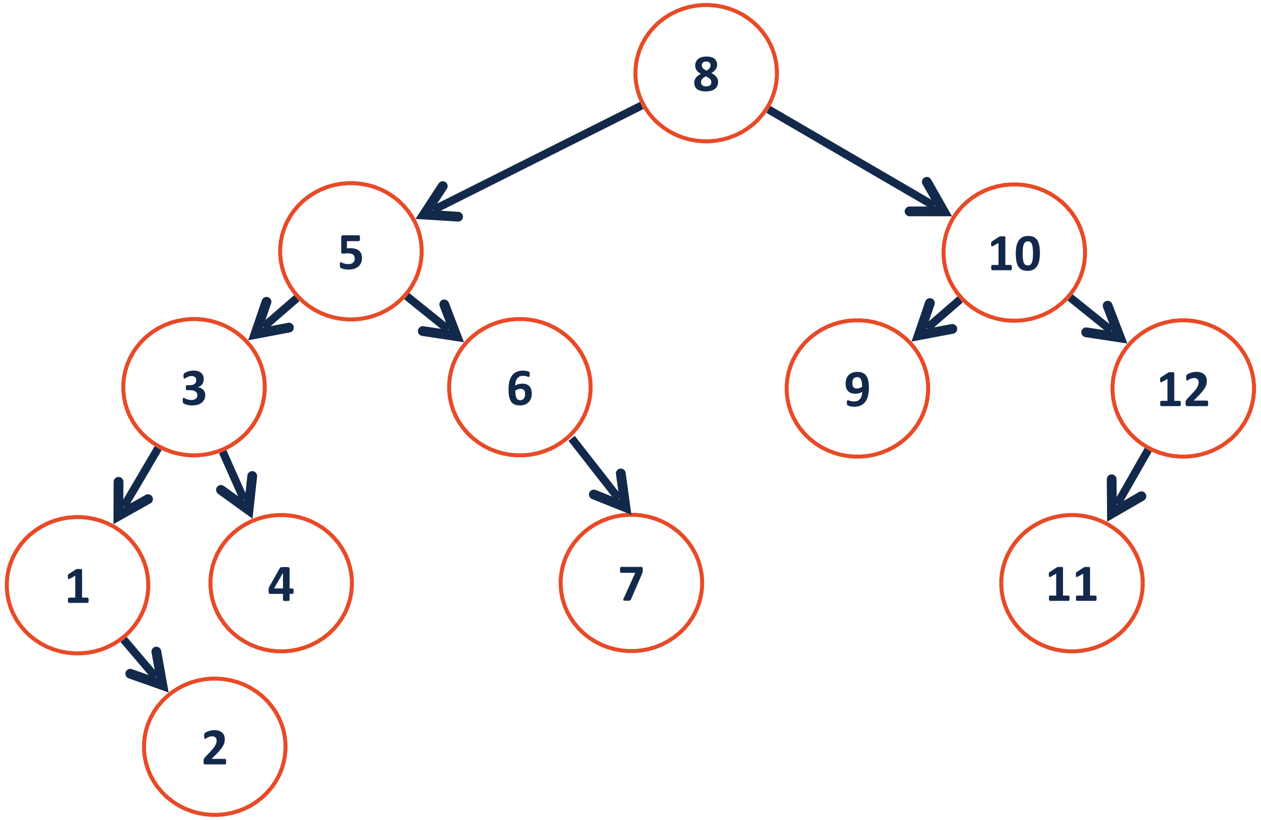
We assume constant time memory access, but the constant factor can be limiting in real world settings!

# Memory Hierarchy (Speed of access)

(Measured in 2011 at https://gist.github.com/hellerbarde/2843375)

| | Time x1 billion |
|---|---|
| L1 cache reference | 0.5 seconds |
| Branch mispredict | 5 seconds |
| L2 cache reference | 7 seconds |
| Mutex lock/unlock | 25 seconds |
| Main memory reference | 100 seconds |
| Compress 1K bytes | 50 minutes |
| Send 2K bytes over 1 Gbps network | 5.5 hours |
| SSD random read | 1.7 days |
| Read 1 MB sequentially from memory | 2.9 days |
| Read 1 MB sequentially from SSD | 11.6 days |
| Disk seek | 16.5 weeks |
| Read 1 MB sequentially from disk | 7.8 months |
| Above two together | 1 year |
| Send packet CA->Netherlands->CA | 4.8 years |

# AVLs in the Cloud



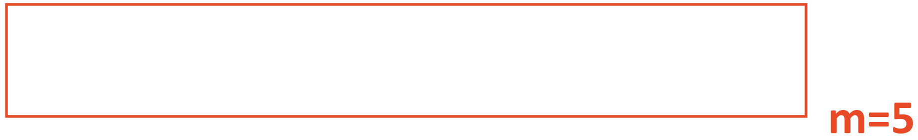
# BTree Motivations

Knowing that we have large seek times for data, we want to:

# BTree (of order m)

A **BTrees** of order **m** is an m-way tree:

- All keys within a node are ordered
- All nodes contain no more than **m-1** keys.



# BTree in the Real World

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -3 | 8 | 23 | 25 | 31 | 42 | 43 | 55 |

m=9

**Goal:** Minimize the number of reads!

Build a tree that uses ______________________ / node

[1 network packet]

[1 disk block]