

# CS 225

## Data Structures

*March 11 – Btrees*

*G Carl Evans*



## B-Trees

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

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

**However,** Our big-O has assumed uniform time for all operations.



## Vast Differences in Time

A **3GHz** CPU performs 3m operations in \_\_\_\_\_.

Old Argument: “Disk Storage is Slow”

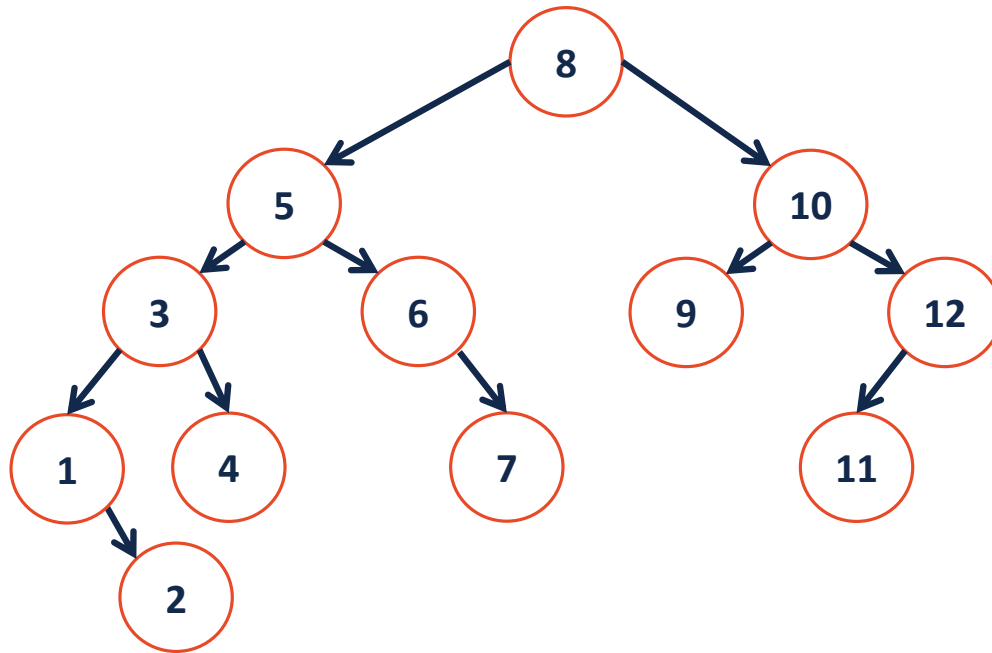
- Bleeding-edge storage is pretty fast:

**SSD**

- Large Disks (25 TB+) still have slow throughout:

New Argument: “The Cloud is Slow!”

# AVLs on Disk





## Real Application

Imagine storing Instagram profiles for everyone in the US:

**How many records?**

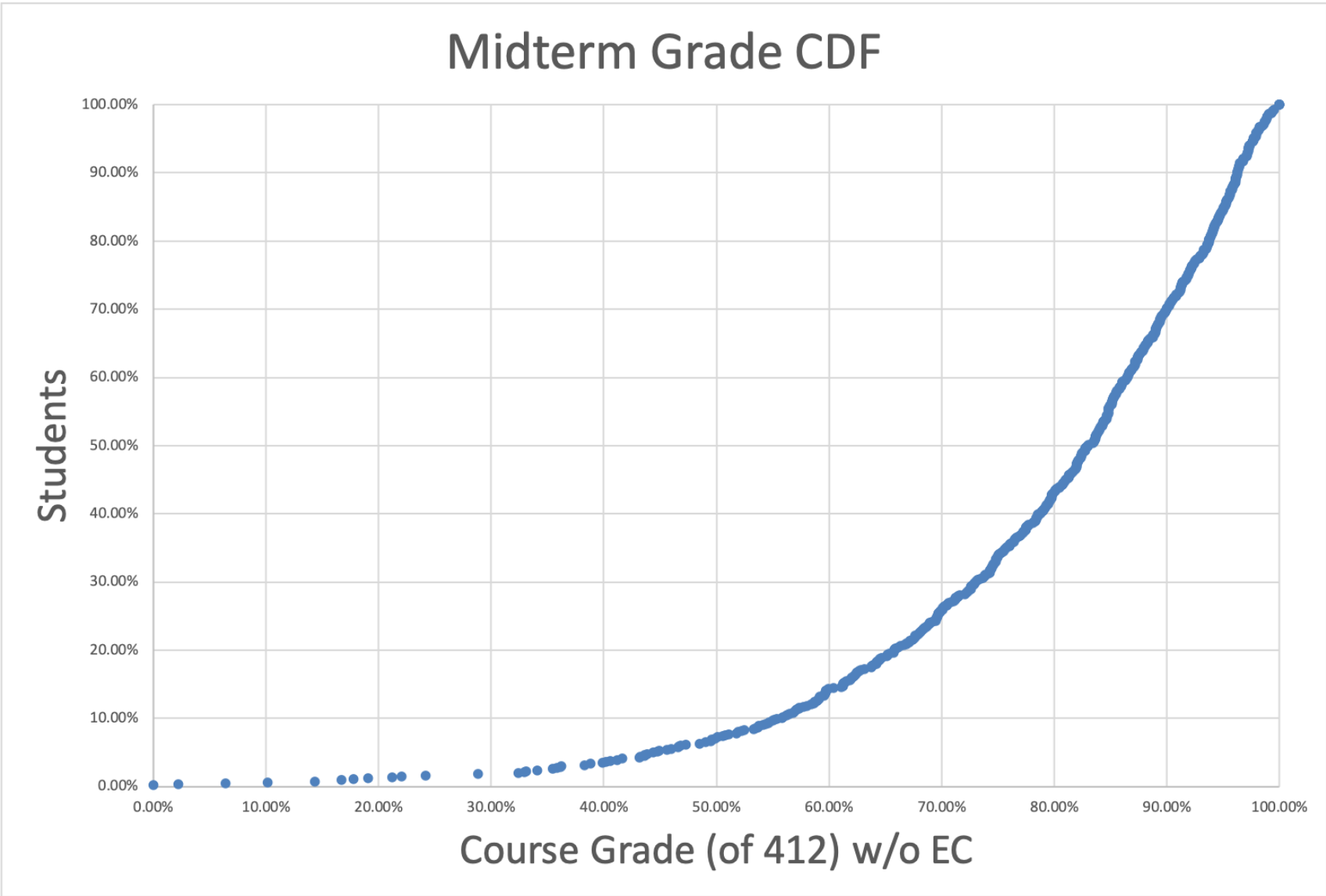
**How much data in total?**

**How deep is the AVL tree?**



# BTree Motivations

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



# BTree (of order m)

-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]



# BTree Insertion

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

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



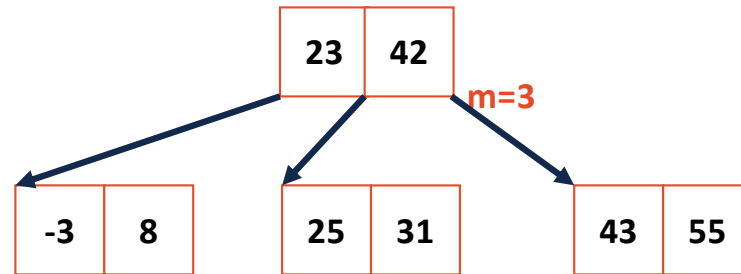


# BTree Insertion

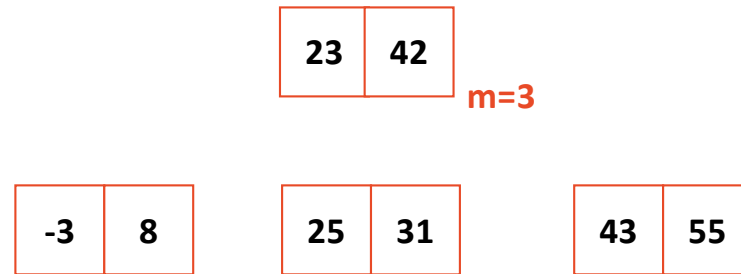
When a BTree node reaches **m** keys:



# BTree Recursive Insert



# BTree Recursive Insert





# BTree Visualization/Tool

<https://www.cs.usfca.edu/~galles/visualization/BTree.html>

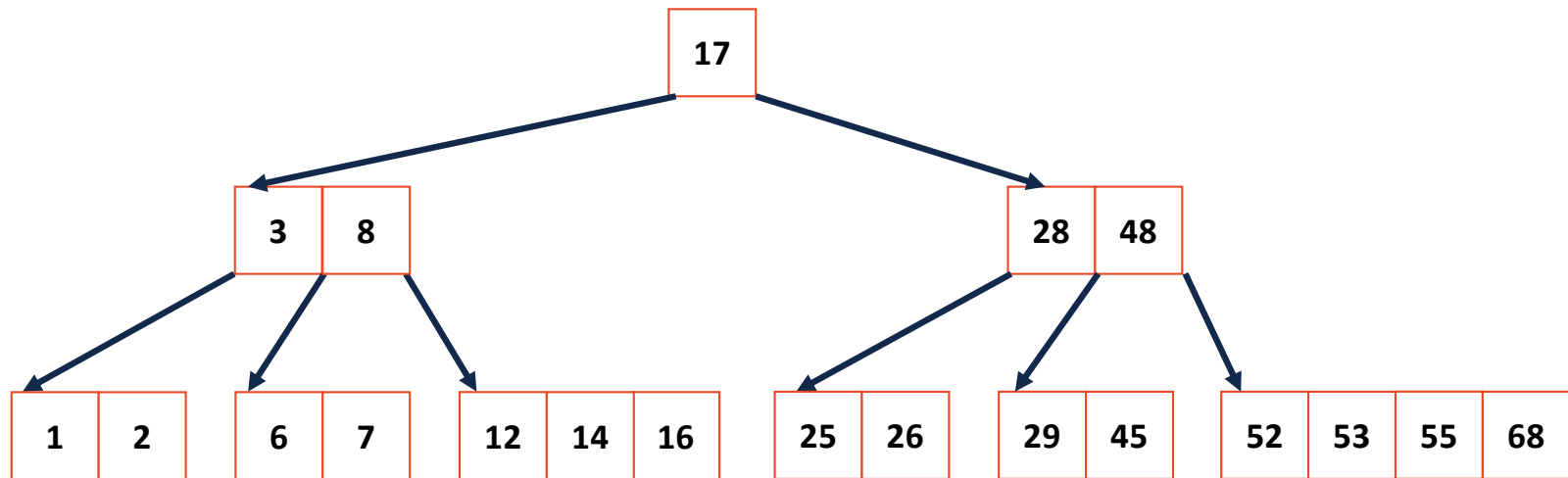


## Btree Properties

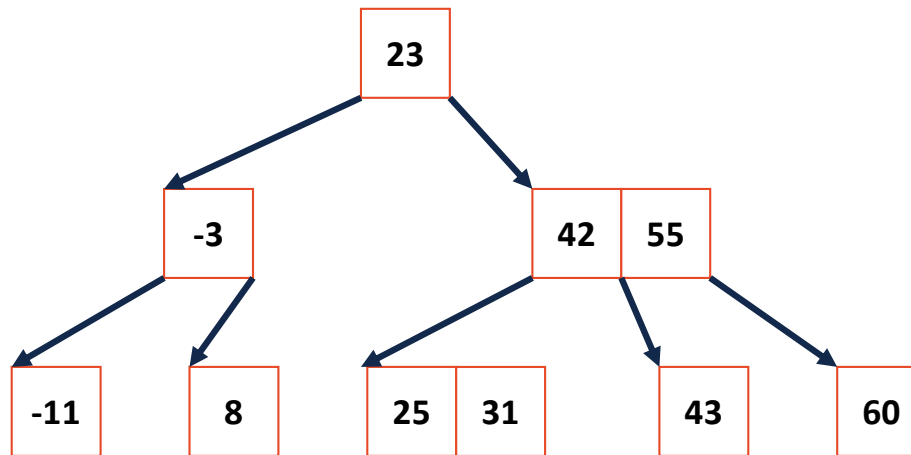
A **BTree** of order **m** is an m-way tree:

- All keys within a node are ordered
- All leaves contain hold no more than **m-1** nodes.
- All internal nodes have exactly **one more child than keys**
- Root nodes can be a leaf or have **[2, m]** children.
- All non-root, internal nodes have **[ceil(m/2), m]** children.
- All leaves are on the same level

# BTree



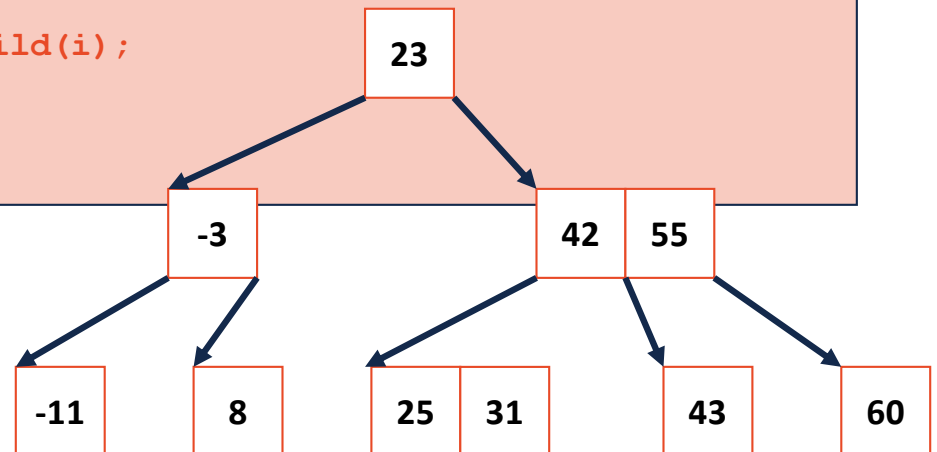
# BTree Search





# BTree Search

```
1 bool Btree::_exists(BTreeNode & node, const K & key) {
2
3     unsigned i;
4     for ( i = 0; i < node.keys_ct_ && key < node.keys_[i]; i++) { }
5
6     if ( i < node.keys_ct_ && key == node.keys_[i] ) {
7         return true;
8     }
9
10    if ( node.isLeaf() ) {
11        return false;
12    } else {
13        BTreeNode nextChild = node._fetchChild(i);
14        return _exists(nextChild, key);
15    }
16 }
```





## BTree Analysis

The height of the BTree determines maximum number of \_\_\_\_\_ possible in search data.

...and the height of the structure is: \_\_\_\_\_.

**Therefore:** The number of seeks is no more than \_\_\_\_\_.

*...suppose we want to prove this!*



## BTree Analysis

In our AVL Analysis, we saw finding an upper bound on the height (given  $n$ ) is the same as finding a lower bound on the nodes (given  $h$ ).

We want to find a relationship for BTrees between the number of keys ( $n$ ) and the height ( $h$ ).



# BTree Analysis

## **Strategy:**

We will first count the number of nodes, level by level.

Then, we will add the minimum number of keys per node (**n**).

The minimum number of nodes will tell us the largest possible height (**h**), allowing us to find an upper-bound on height.



# BTree Analysis

The minimum number of **nodes** for a BTree of order  $m$  **at each level:**

root:

level 1:

level 2:

level 3:

...

level  $h$ :



# BTree Analysis

The **total number of nodes** is the sum of all of the levels:



# BTree Analysis

The **total number of keys:**



## BTree Analysis

The **smallest total number of keys** is:

So an inequality about  **$n$** , the total number of keys:

Solving for  **$h$** , since  **$h$**  is the number of seek operations:





# BTree Analysis

Given  $m=101$ , a tree of height  $h=4$  has:

Minimum Keys:

Maximum Keys: