

Data Structures

BTree

CS 225

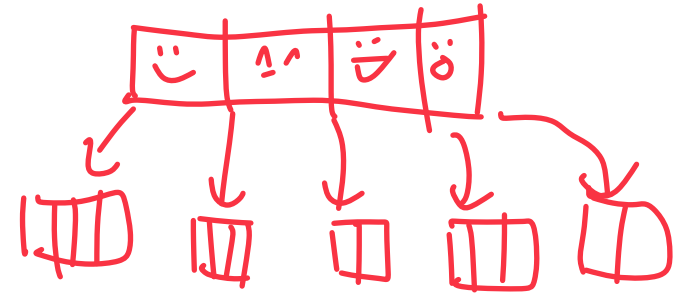
Brad Solomon

October 7, 2026



UNIVERSITY OF
ILLINOIS
URBANA - CHAMPAIGN

Department of Computer Science



Learning Objectives

Remind ourselves one (engineering) issue with trees

Introduce (and implement) the B Tree!

Summary of Balanced BST

Pros:

$O(\log N)$ for insert, find, remove

Optimal range queries in 1D

Cons:

$O(\log N)$ isn't that great

Large in-memory requirement



(Most) Data Structures so far...

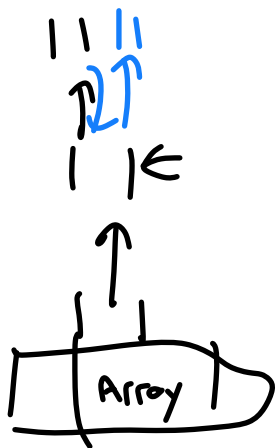
	Unsorted Array	Sorted Array	Unsorted Linked List	Sorted Linked List	Binary Tree	BST	AVL
Find	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(h) = O(n)$	$O(h) = O(\log n)$
Insert	$O(1)^*$ / $O(n)$	$O(n)$	$O(1)$	$O(n)$	???	$O(h) = O(n)$	$O(h) = O(\log n)$
Remove	$O(1)^*$ / $O(n)$	$O(n)$	$O(1)$ / $O(n)$	$O(n)$	???	$O(h) = O(n)$	$O(h) = O(\log n)$
Traverse	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
	60% 50%	20%	30%	60%	10%	6%	0%

What structure(s) seem to have bad tradeoffs?

[illegible]

Engineering vs Theory Efficiency

Smaller



	Time x1 billion	Like
L1 cache reference	0.5 seconds	Heartbeat 💖
Branch mispredict	5 seconds	Yawn 😴
L2 cache reference	7 seconds	Long yawn 😴😴😴
Mutex lock/unlock	25 seconds	Make coffee ☕
Main memory reference	100 seconds	Brush teeth
Compress 1K bytes	50 minutes	TV show 📺
Send 2K bytes over 1 Gbps network	5.5 hours	(Brief) Night's sleep 🛌
SSD random read	1.7 days	Weekend
Read 1 MB sequentially from memory	2.9 days	Long weekend
Read 1 MB sequentially from SSD	11.6 days	2 weeks for delivery 📦
Disk seek	16.5 weeks	Semester
Read 1 MB sequentially from disk	7.8 months	Human gestation 🐣
Above two together	1 year	🌍☀️
Send packet CA->Netherlands->CA	4.8 years	Ph.D. 🎓

Fast
Small

Very slow
but large

Larger

(Care of <https://gist.github.com/hellerbarde/2843375>)

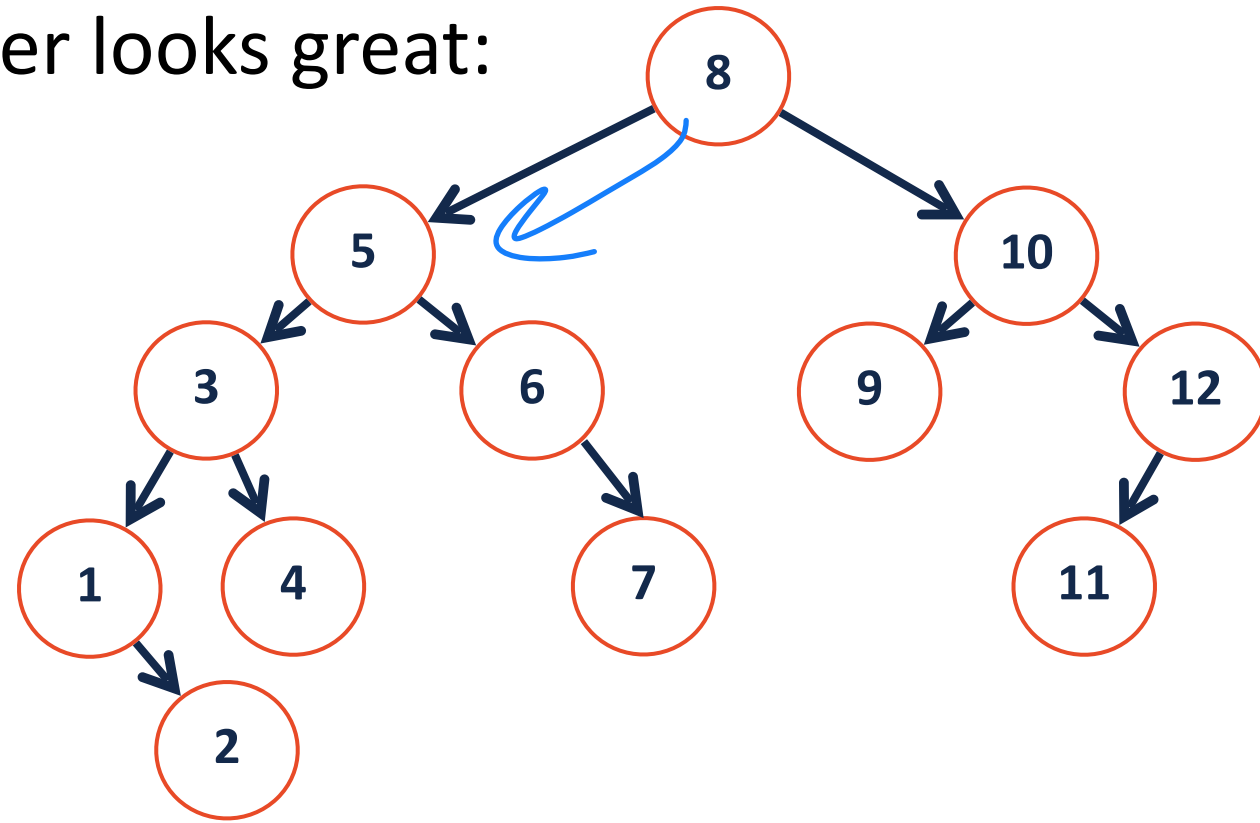
Engineering vs Theory Efficiency

In Big-O we have assumed uniform time for all operations, but this isn't always true.

0.0002 ms to lookup
L1 cache

However, seeking data from the cloud may take 40ms+.

...an $O(\lg(n))$ AVL tree no longer looks great:



Considering hardware limitations

Can we always fit our data in main memory?

No

Where else can we keep our data? RAM, cloud, SSDs

Does this match our assumption that all memory lookups are $O(1)$?

No!

BTree Design Motivations

When large seek times become an issue, we address this by:

0.0001 ms vs 40 ms

1) Keep the number of seeks low

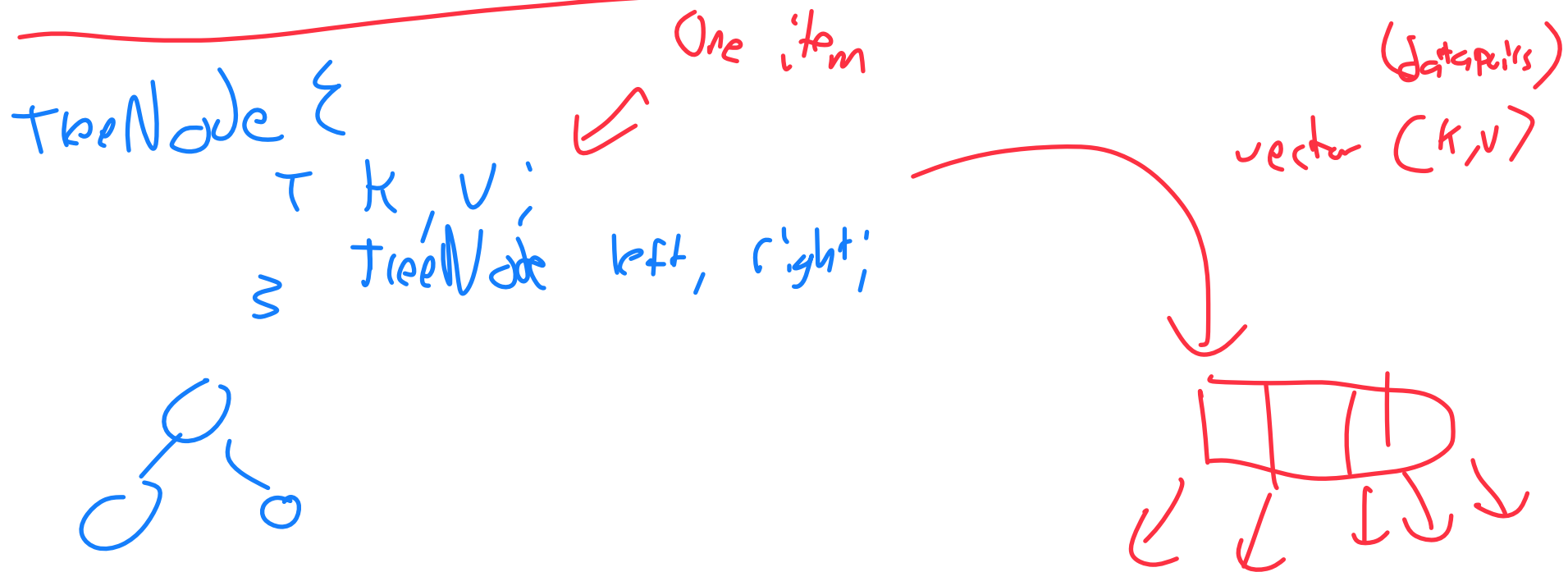
↕ what does mean for a tree?

keep our tree height small (bounded)

BTree Design Motivations

When large seek times become an issue, we address this by:

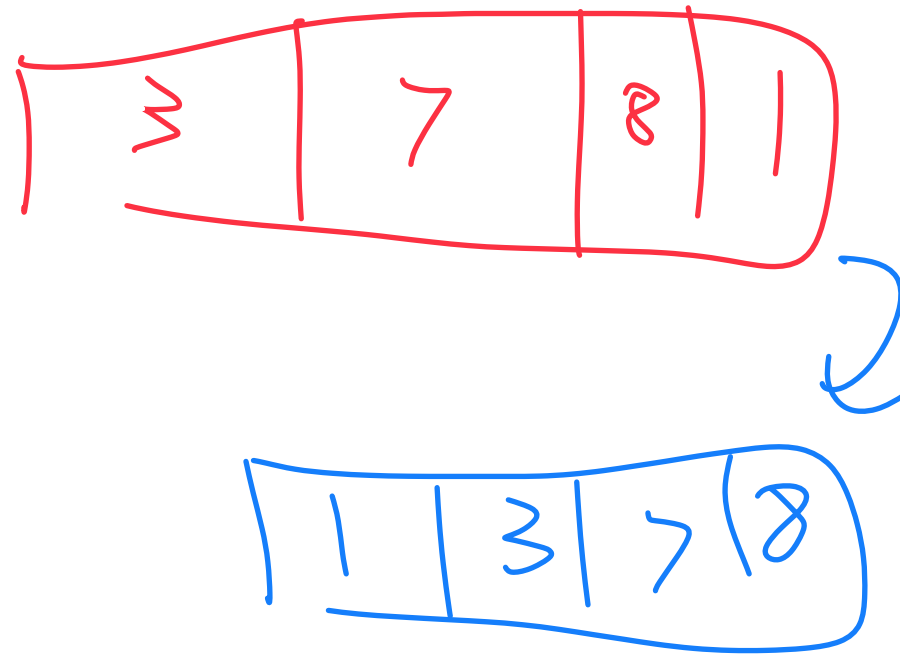
2) When possible keep data stored locally



BTree Design Motivations

When large seek times become an issue, we address this by:

3) Make sure the data we look up is relevant!



Sort
each
array!

BTree Design Motivations



When large seek times become an issue, we address this by:

- 1) Keep the number of seeks low
- 2) When possible keep data stored locally
- 3) Make sure the data we look up is relevant!

BTree Design Motivations

1) Keep the number of seeks low

Make a tree that is wide and short by...

& some other stuff

2) When possible keep data stored locally

Store more than one key in each node

3) Make sure the data we look up is relevant!

Make sure our tree is still ordered

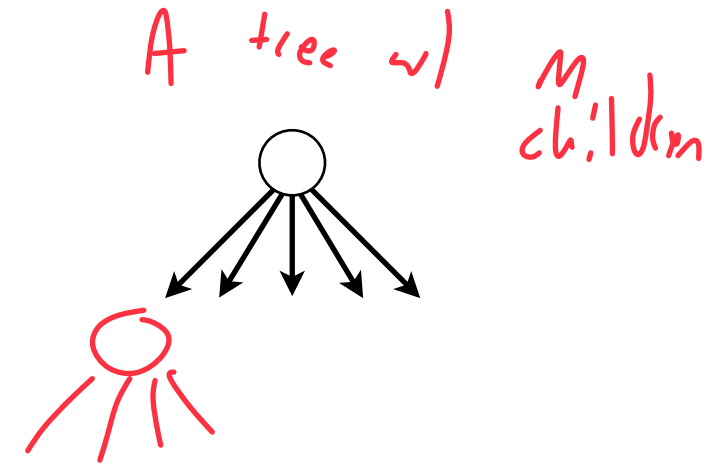
Both arrays &
overall tree is
sorted

BTree

A BTree (of order m) is a m-ary tree

vector $\langle \text{data pairs} \rangle$

vector $\langle \text{BTreeNode}^* \rangle$



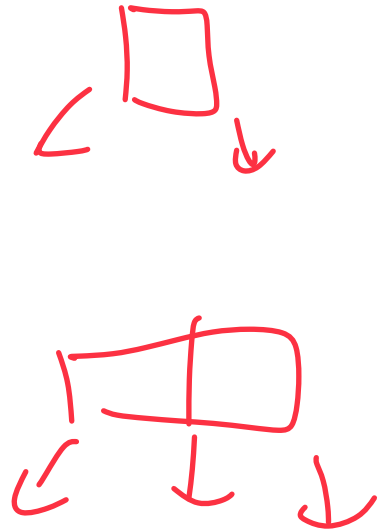
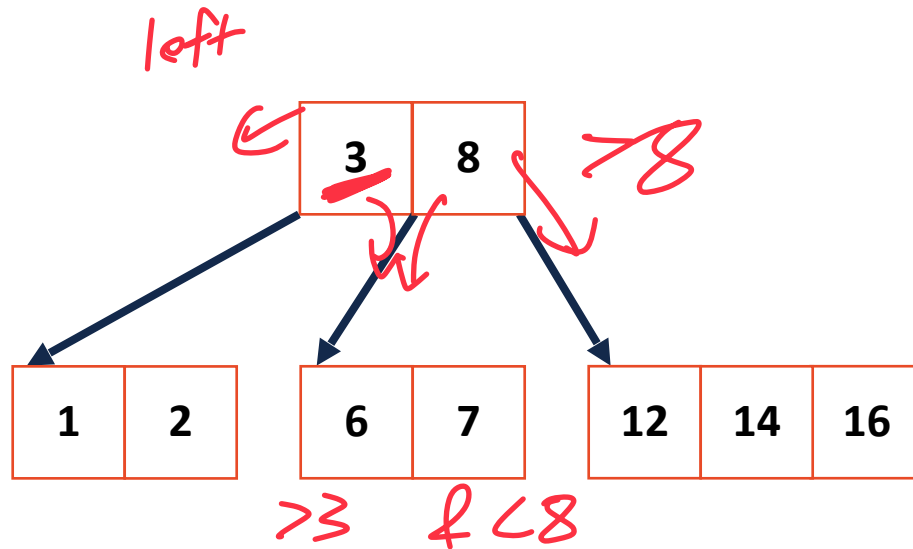
BTree

A BTree (of order m) is a m -ary tree

All nodes contain up to **$m-1$ keys in sorted order!**

An internal node of k keys has **$k+1$** children

m children



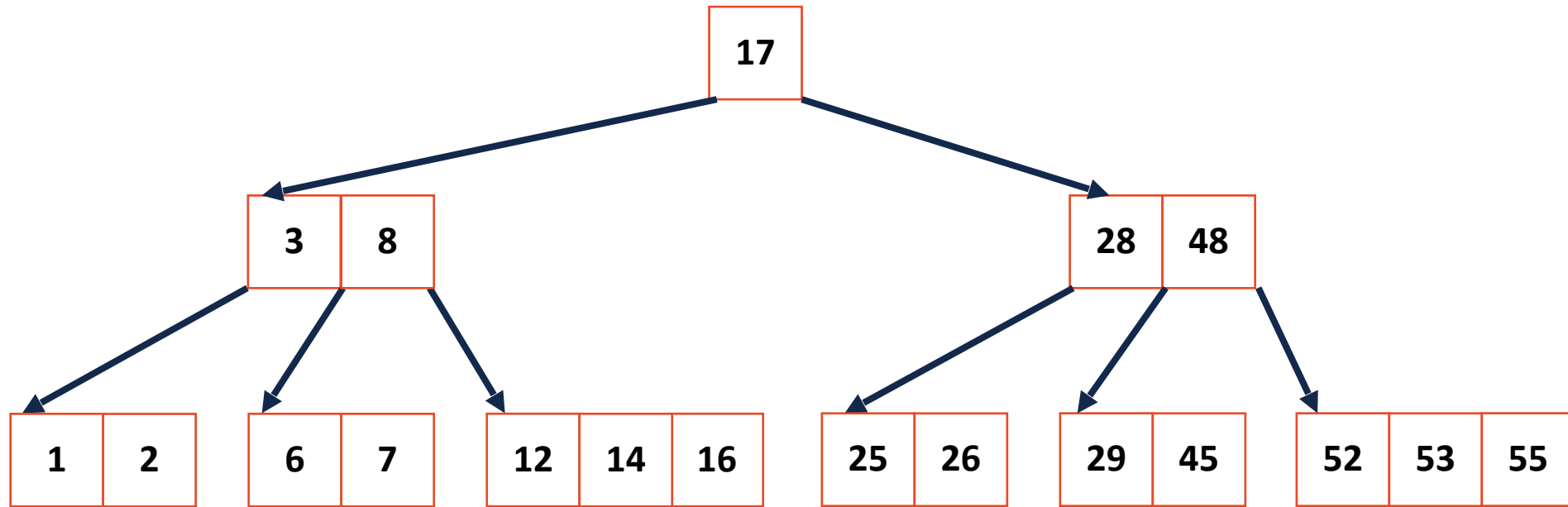
BTree



A BTree (of order m) is a m -ary tree

Nodes are ordered with up to **$m-1$** sorted keys and **m** children

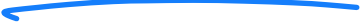
All leaves in a BTree are on the same level *← Insert Shows how this works!*



BTree ADT

Constructor

Insert



Find



Remove



BTree Node (of order m)

M = 5

<i>key</i> -3	5	8	13
<i>Tree's Value</i>			

```
1 struct BTreeNode {  
2     std::vector<DataPair> elements;  
3     std::vector<BTreeNode*> children;  
4 };
```



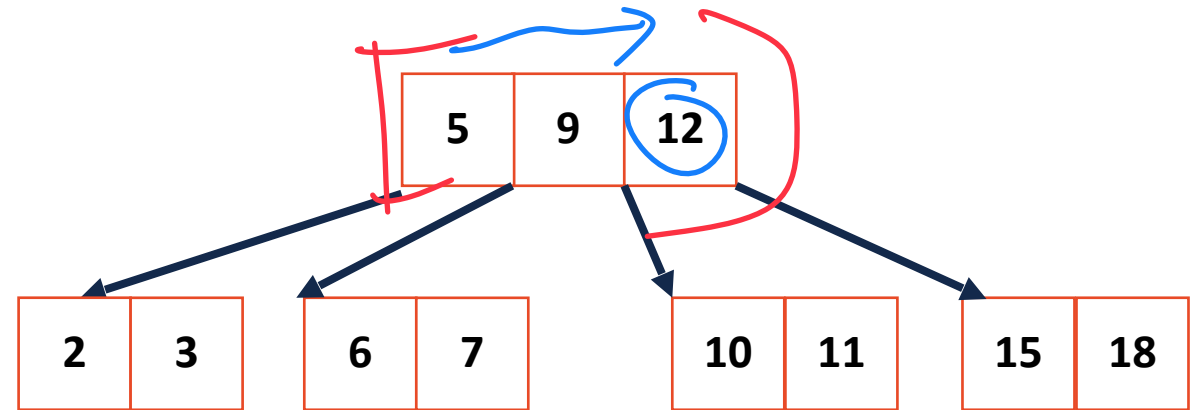
Reminder: Order is on keys not values (sorted)

BTree Find

Find(12)

1) Do array find()
↳ Naïve or

Binary search
(sorted array!) [both walk]



BTree Find

Base Case:

If root is empty, return

If leaf, do array find() and return

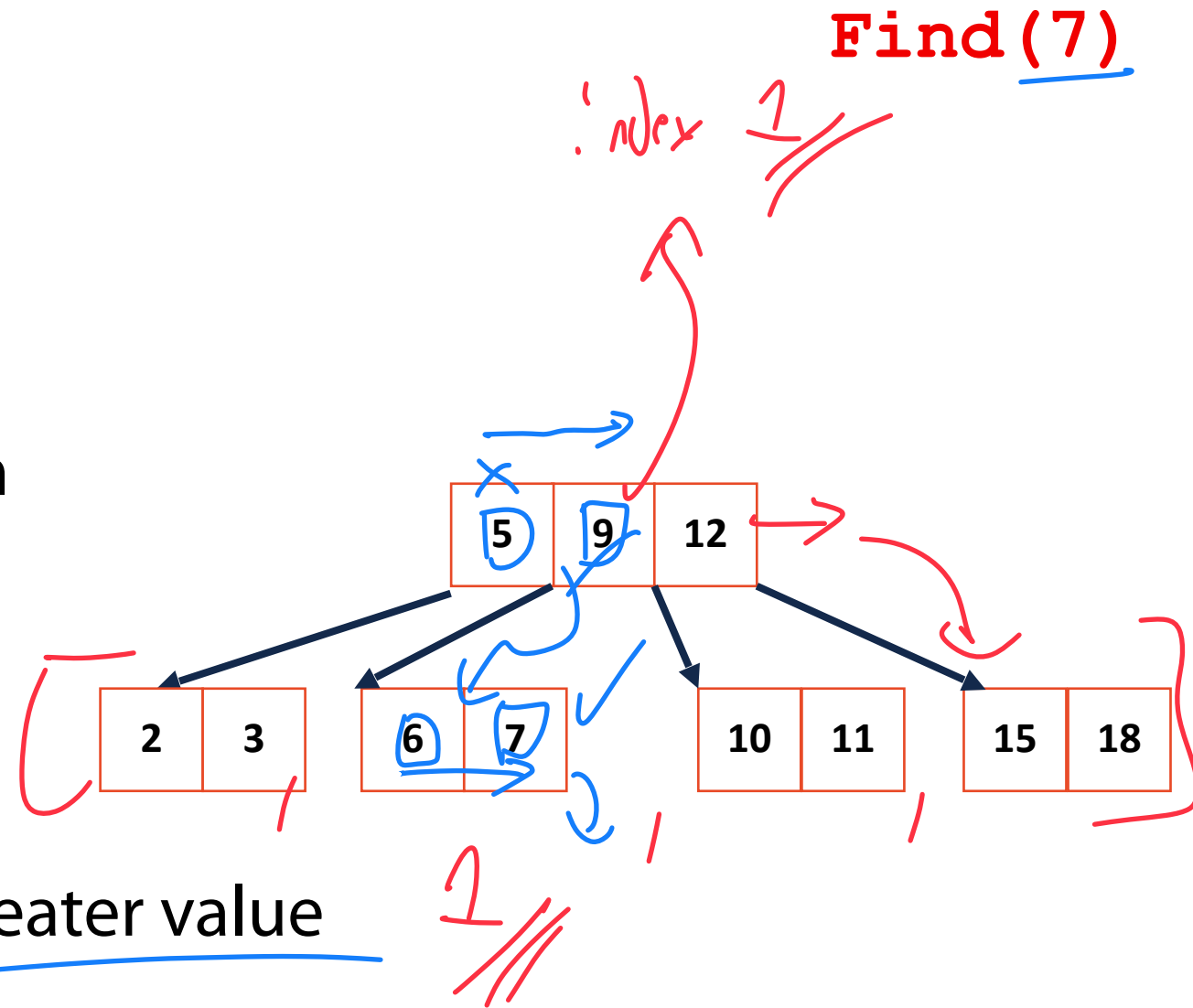
If array find, return immediately

Recursive Step:

Array find() for match or first greater value

Recurse on appropriate child

Tip: Index of first greater value is index of child we want to visit!



BTree Find

Find(7)

Base Case:

If root is empty, return

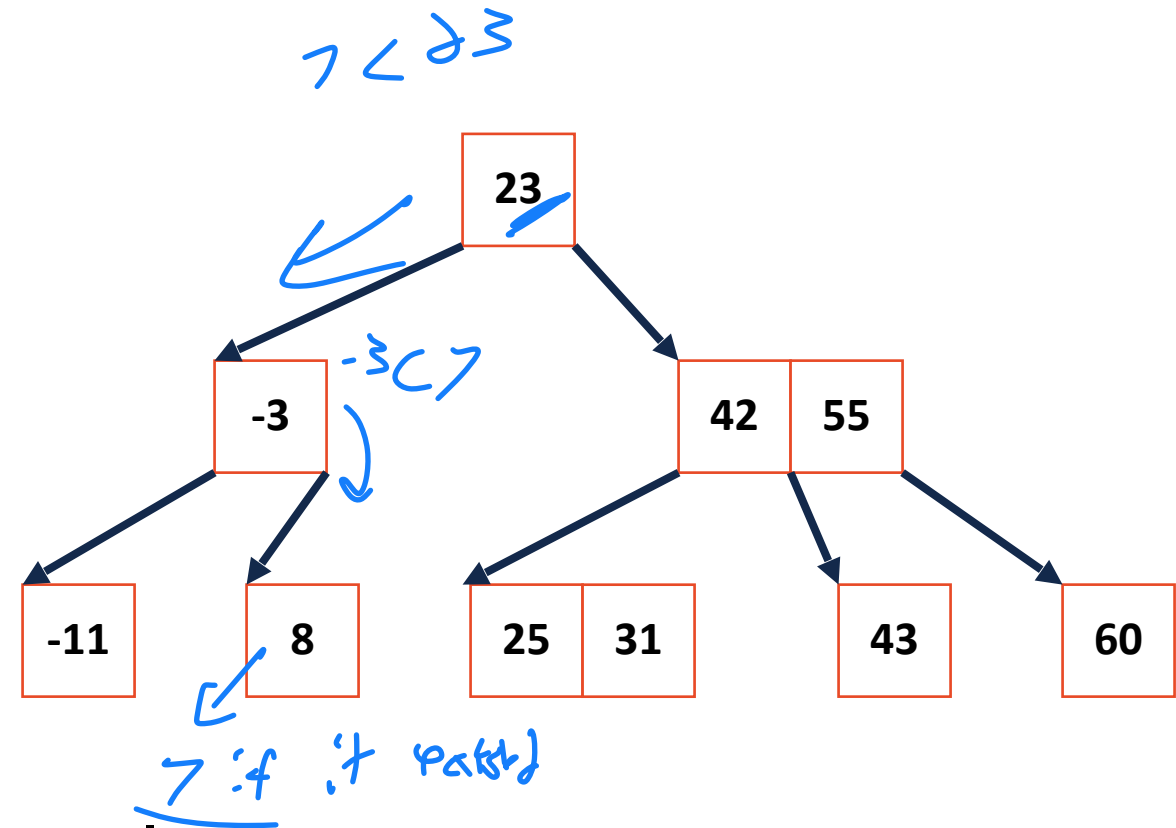
If leaf, do array find() and return

Recursive Step:

Array find() for match or first greater value

Recurse on appropriate child

Tip: Index of first greater value is index of child we want to visit!



BTree Find

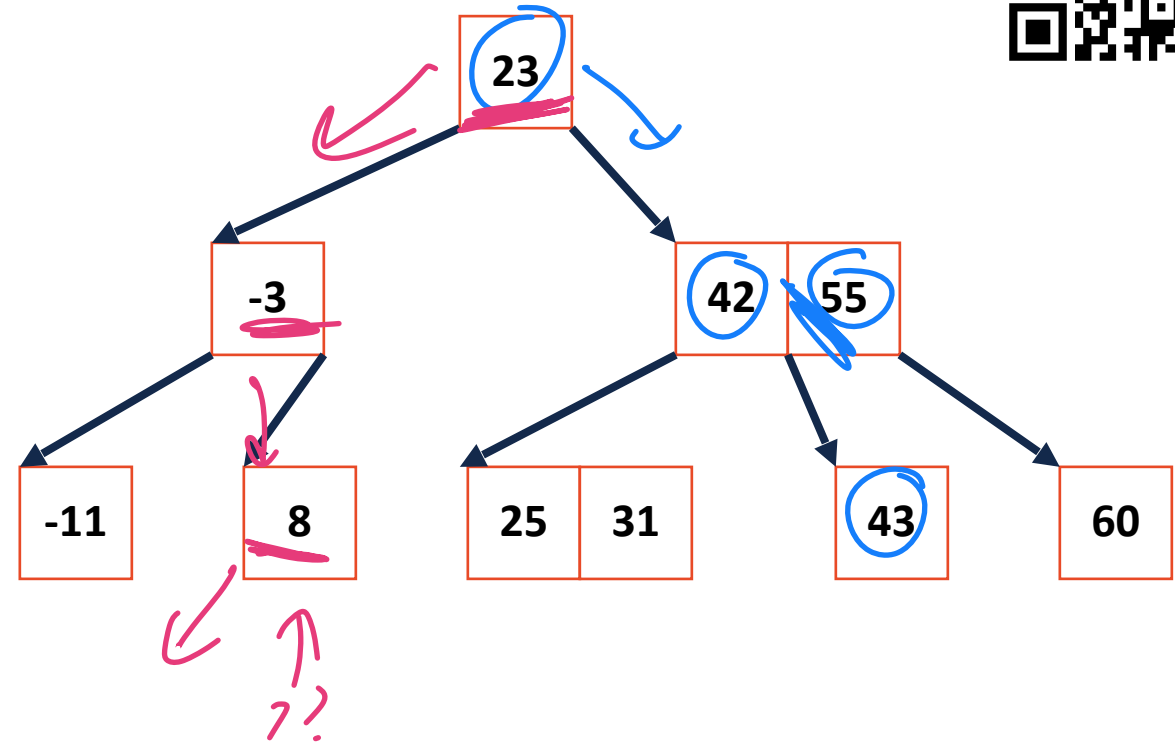
Find(43)



How many keys are looked at to complete this search?

↳ assume naive array search

Find(>)

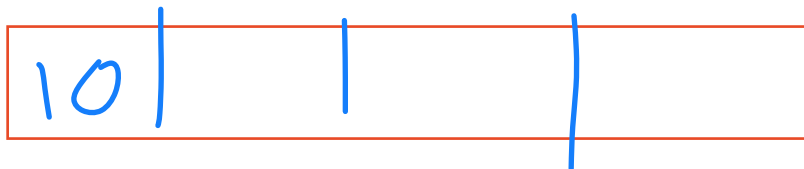


BTree Insertion

M = 5

Given an empty BTree, we make a new root node which has...

an empty vector!



Insert(10)

BTree Insertion

M = 5

Chain insertions fill our array in sorted order

10				
----	--	--	--	--

Insert (10)

5	10			
---	----	--	--	--

Insert (5)

5	7	10		
---	---	----	--	--

Insert (7)

5	7	9	10	
---	---	---	----	--

Insert (9)

2	5	7	9	10
---	---	---	---	----

Insert (2)

Repeated sorted
'inserts'!

Oh no! Bal.
At most $m-1$ (4) keys

BTree Insertion

Array is Search (previously)
 $O(\log 4)$ vs $O(4)$



M = 5

Problem 1: Sorted array insert is slow!

10				
----	--	--	--	--

5	10			
---	----	--	--	--

...

2	5	7	9	10
---	---	---	---	----



Insert (10)

Insert (5)

Insert (2)

$O(1)$

6%

$O(\log n)$

6%

$O(\log m)$

18%

$O(n)$

30%

Got tricked!

$O(m)$

45%

where m is a constant

What is the Big O to insert a new dataPair into a BTree Node?

The size of array

BTree Insertion

M = 5

Problem 1: Sorted array insert is slow!

10				
----	--	--	--	--

Insert (10)

5	10			
---	----	--	--	--

Insert (5)

...

2	5	7	9	10
---	---	---	---	----

Insert (2)

Solution: M is a constant! (So no its not)



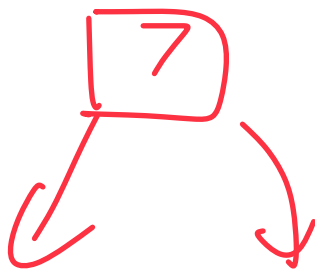
BTree Insertion

M = 5

Problem 2: A BTree of order **M** can only store **M-1** keys!

When a BTree node reaches **M** keys, what do we do?

- 1) Find median value
- 2) Push the median value up
↳ 1 node becomes 3 nodes
new root
new child
imp choice on what
happens to original



2	5	7	9	10
---	---	--------------	---	----

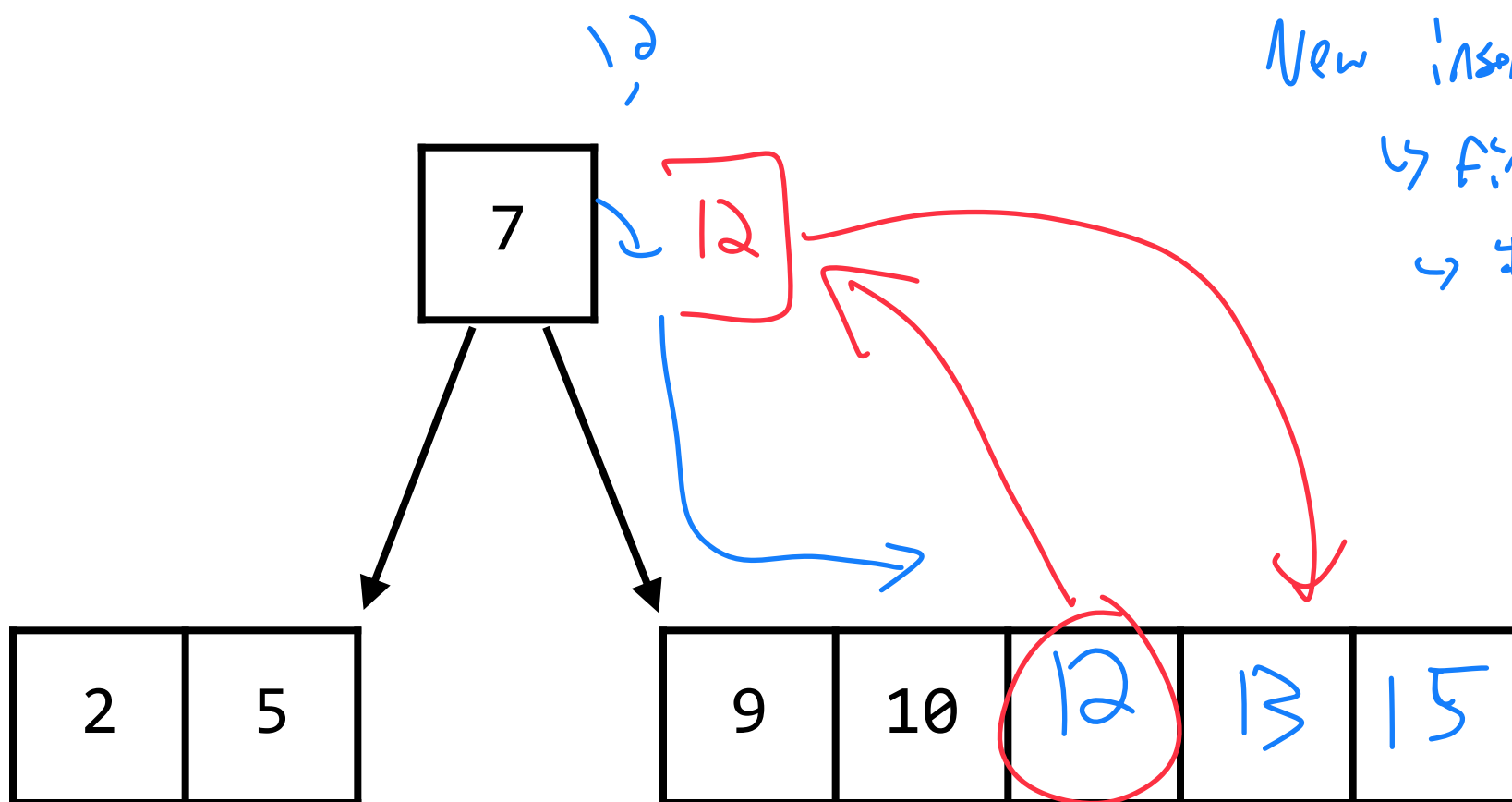
5 items! Max is 4!

BTree Insertion

M = 5

Problem 2: A BTree of order **M** can only store **M-1** keys!

Solution: When we hit **M** items, split and make a new **parent node**!



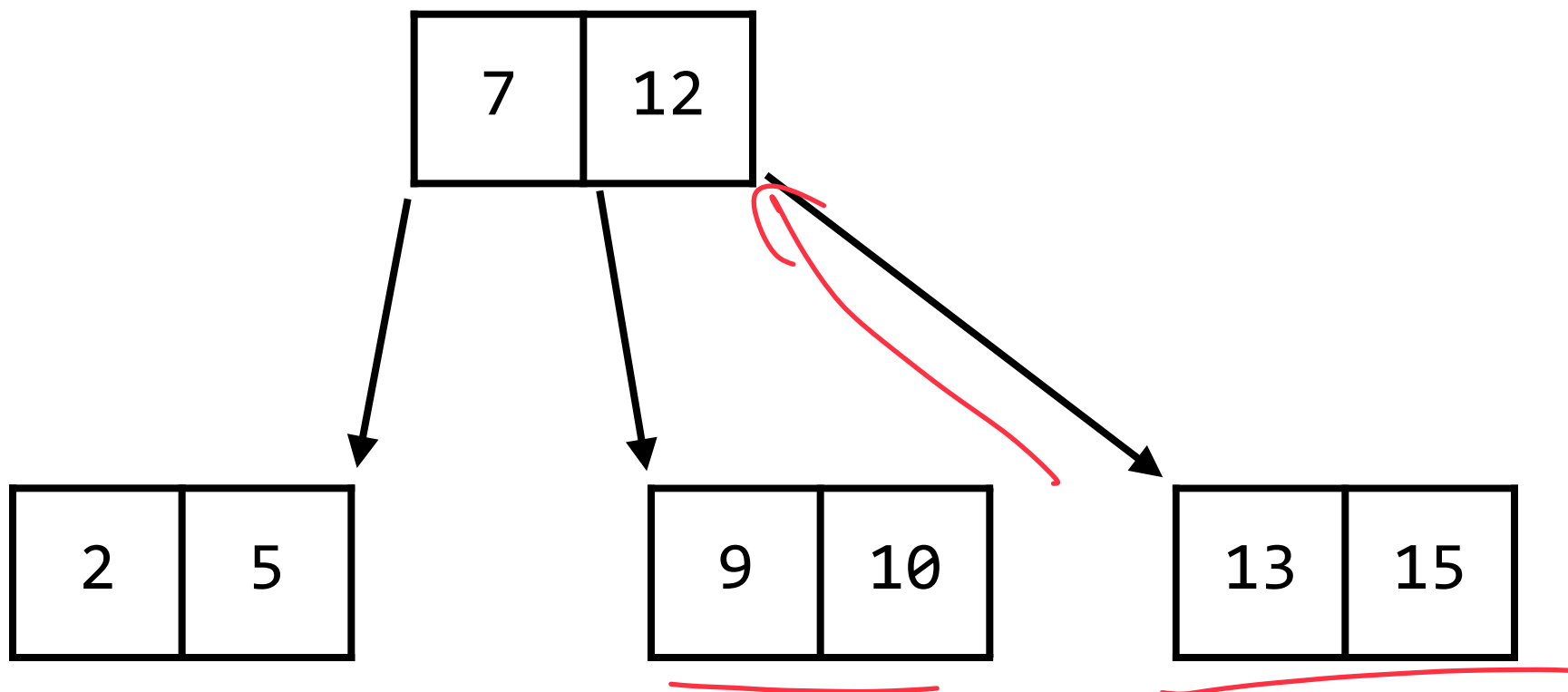
New insert from root
↳ find insert location
↳ insert in array @ leaf

BTree Insertion

M = 5

Problem 2: A BTree of order **M** can only store **M-1** keys!

Solution: When we hit **M** items, split and make a new **parent node**!



BTree Insertion

M = 5

Problem 3: I need to find median value AFTER inserting the **M**th value

10				
----	--	--	--	--

Insert (10)

5	10			
---	----	--	--	--

Insert (5)

...

2	5	7	9	10
---	---	---	---	----

Insert (2)

BTree Insertion

M = 5

Problem 3: I need to find median value AFTER inserting the **M**th value

10				
----	--	--	--	--

Insert (10)

5	10			
---	----	--	--	--

Insert (5)

...

2	5	7	9	10
---	---	---	---	----

Insert (2)

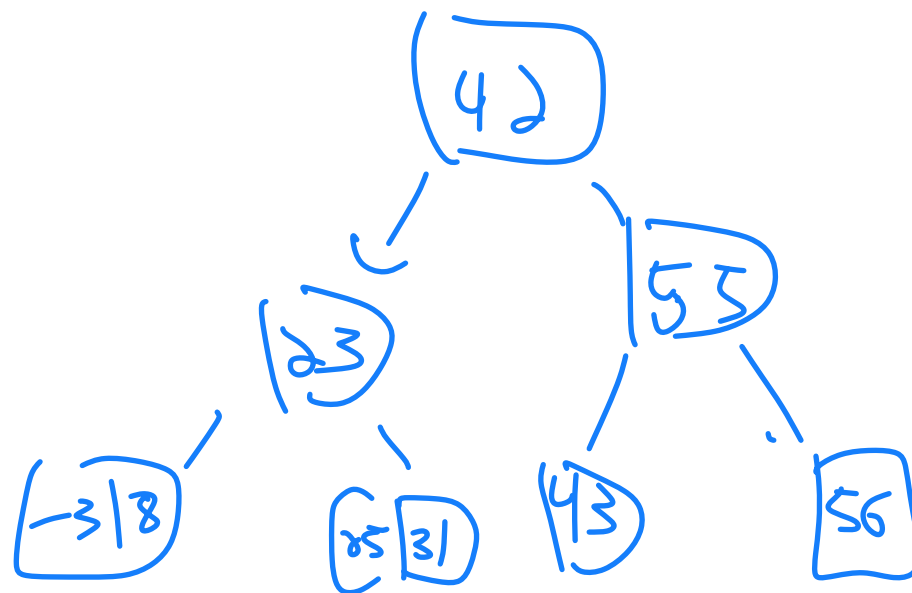
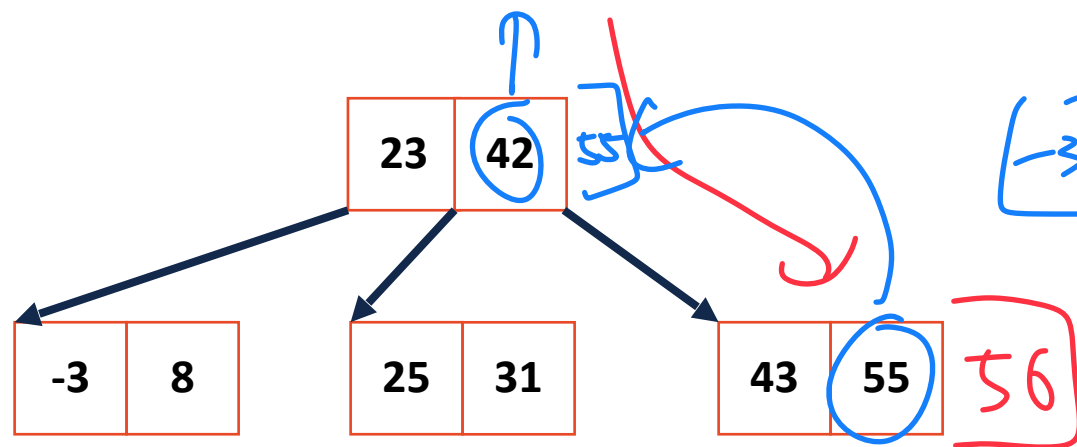
Non-Optimal Solution: Pre-allocate **M** size arrays for every node!

BTree Recursive Insert

Insert(56), M = 3



Insert always starts at a leaf but can propagate up repeatedly.



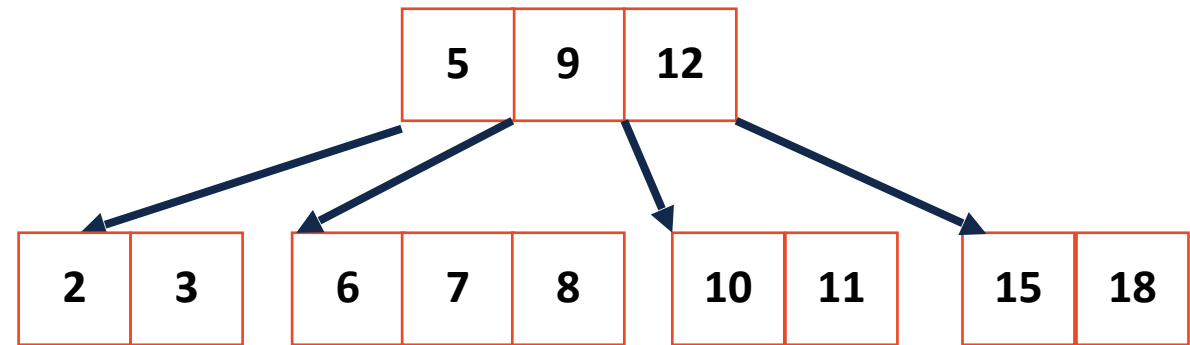
This is

Why B Tree always has leaves
at same level!

BTree Remove

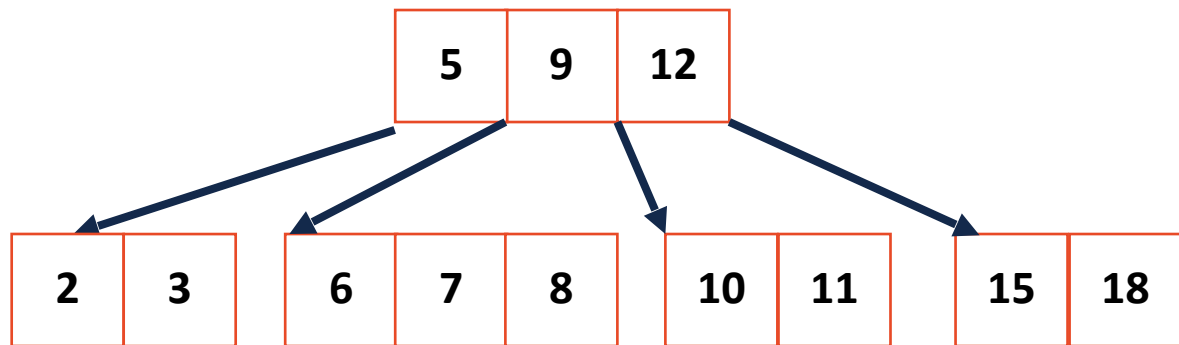
BTree removal is complicated! **It won't be part of the lab.**

However lets consider how we would handle the following cases...



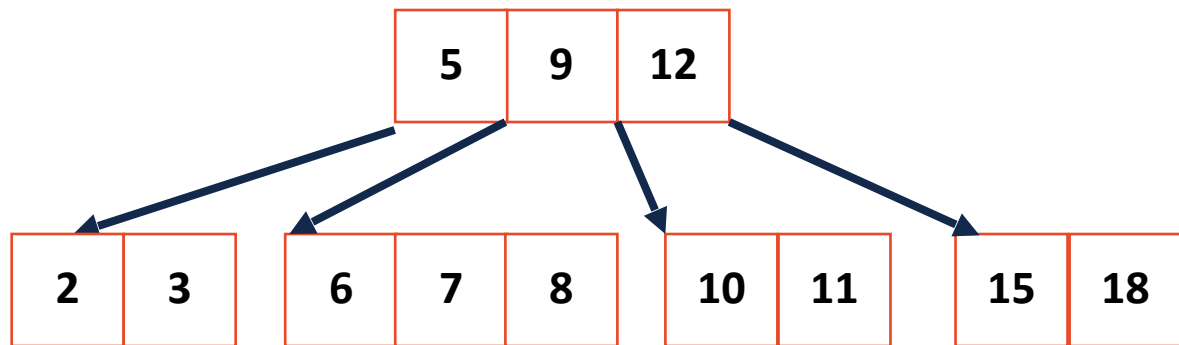
BTree Remove

Remove (8)



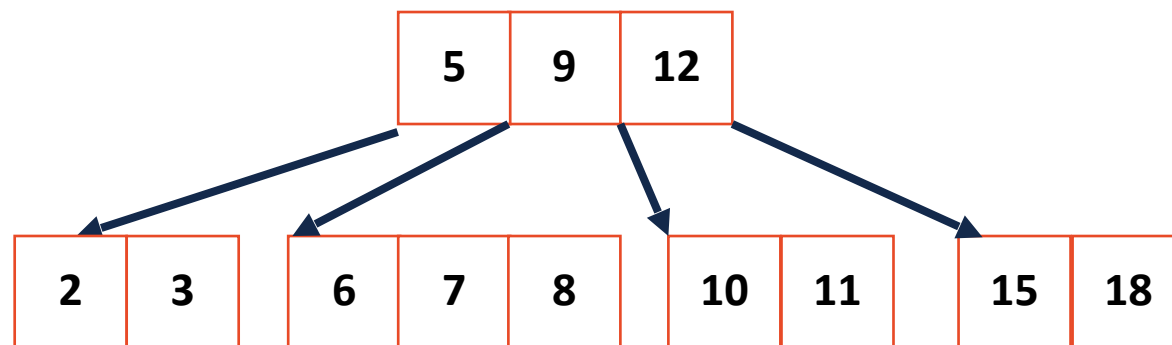
BTree Remove

Remove (2)



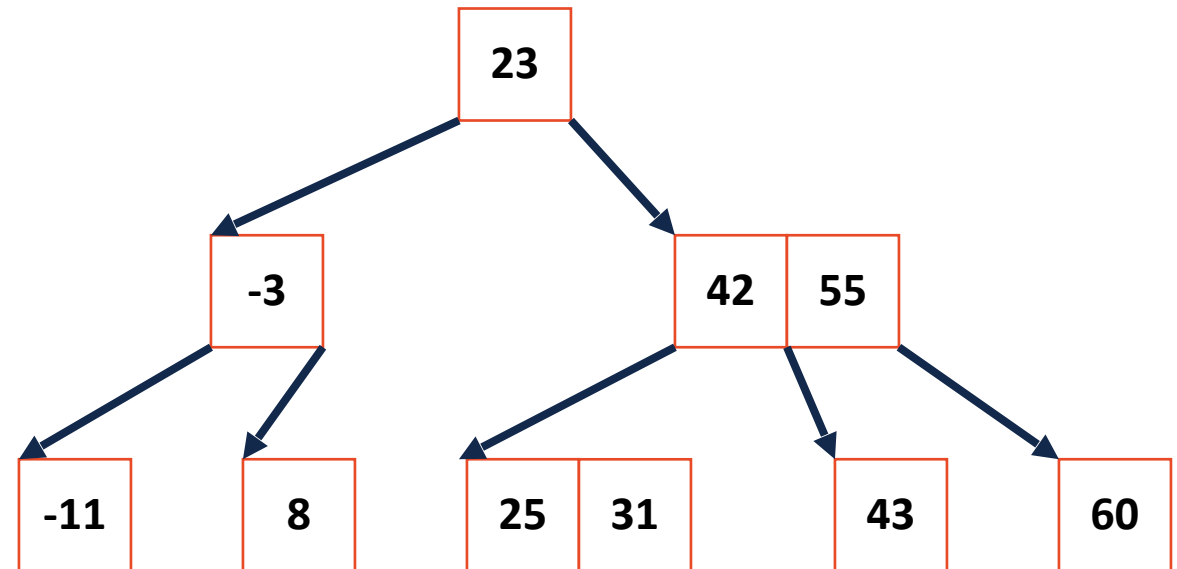
BTree Remove

Remove (15)



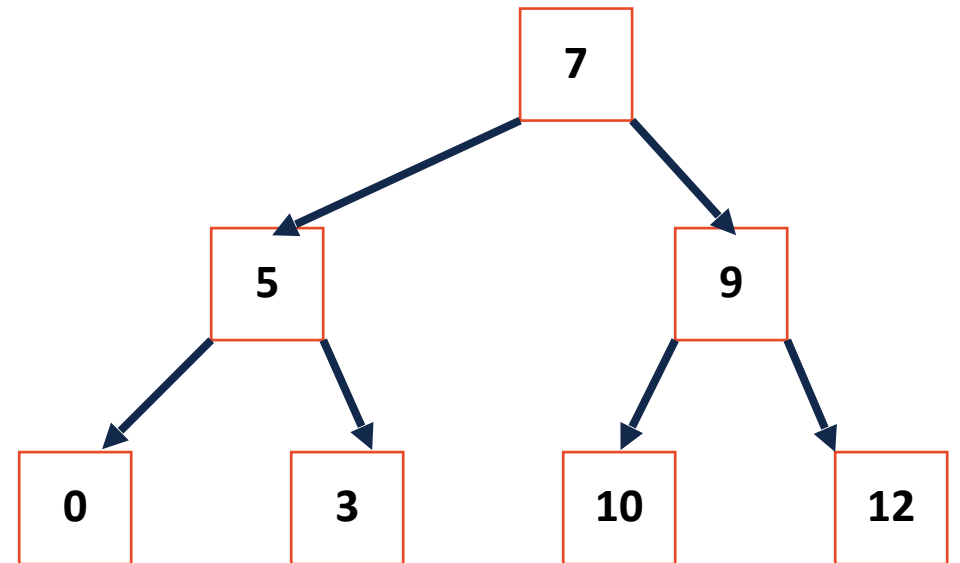
BTree Remove

Remove (42)



BTree Remove

Remove (5)



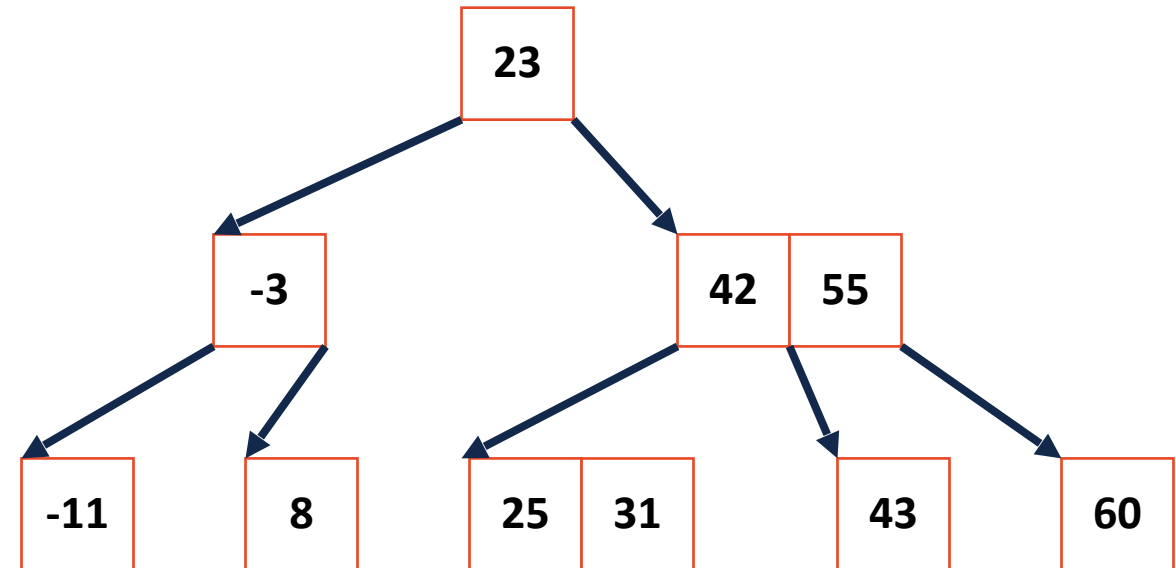
BTree Visualization/Tool

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

For next time: BTree Analysis

We've seen the ADT

What is the runtime for our BTree operations?

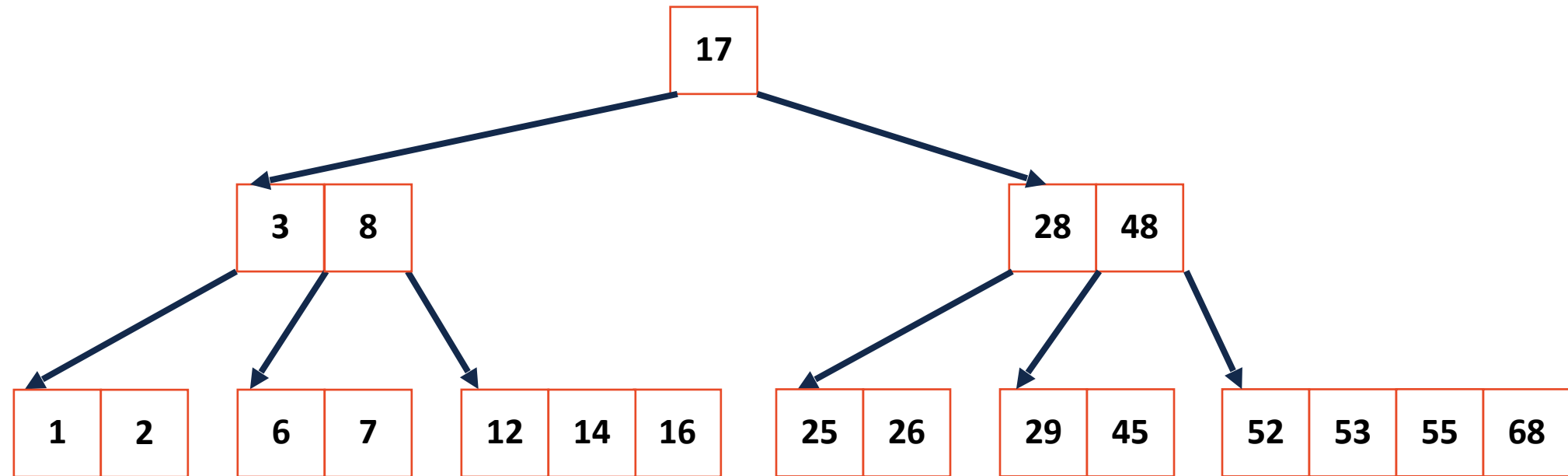


BTree Node (of order m)

Brainstorm together: What value of m should we be using?

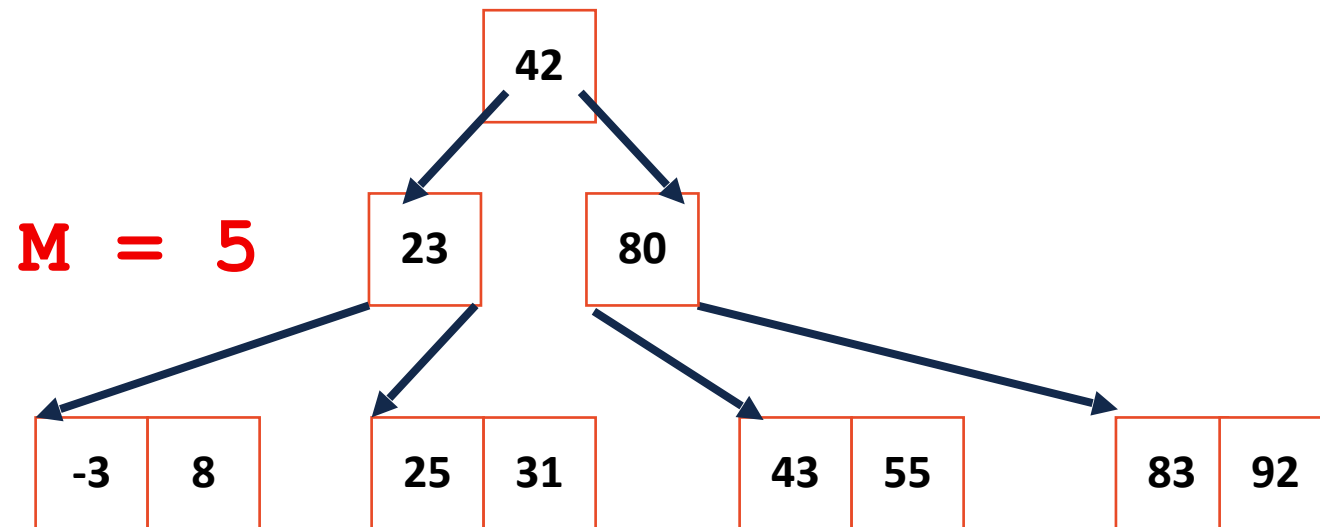
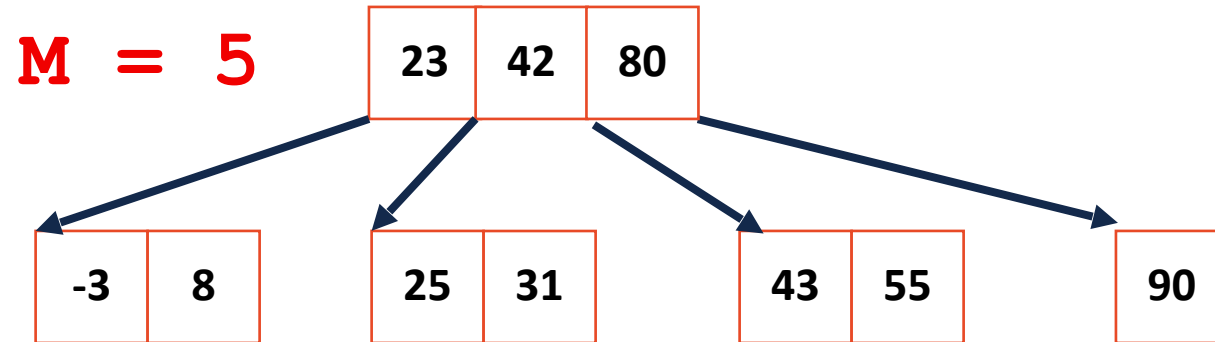
BTree

If I tell you this is a valid BTree, what is the value of m ?



BTree Size Restrictions

By definition we have max, but do we have min? Are these trees valid?





BTree Properties

A **BTree** of order **m** is an m-ary tree and by definition:

- All keys within a node are ordered
- All leaves contain no more than **m-1** keys.
- All internal nodes have exactly **one more child than keys**

Root nodes can be a leaf or have _____ children.

All non-root, internal nodes have _____ children.

All leaves in the tree are at the same level.