Data Structures
BTree Analysis

CS 225
Brad Solomon & G Carl Evans

October 6, 2023

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

CS 225 Extra Credit

Max is 100 pts

POTDs: 40 points

old

Early MP submissions: 40 points

+ 2

Extra credit projects: 40 points

20 pts new

An extra (13th) lab: 10 points

Above 70% participation in Informal Early Feedback: 5 points

Above ??% participation in ICES Evaluations: 5 points

Informal Early Feedback Released!

A larger anonymous survey designed to give feedback to staff

Collective extra credit opportunity! → As a class submit

Particularly interested in ways to improve lecture and labs.

MP Mosaics Quick Tips

1. Pay close attention to your recursion and default point constructor

↳ empty TreeNode ↳ (0, 0, 0)

↳ return TreeNode()

↳ closest point?

2. Individual mosaic tests are NOT comprehensive.

```
TEST_CASE("KDTree::findNearestNeighbor (2D), returns correct result",
"[weight=1][part=1]") {
  /* ... */
  compareBinaryFiles(fname, "../data/kdtree_"+to_string(K)+"_"+to_string(size)+"-expected.kd" );

  REQUIRE( tree.findNearestNeighbor(target) == expected );
```

first shape

Not

3. Take advantage of class resources:

Videos

- k-d tree : 2-D example
- (partition based) Quick Select
- findNearestNeighbor - Part 1: Explanation
- findNearestNeighbor - Part 2: Walkthrough

Learning Objectives

Finish implementing BTree ADT

Analyze the performance of the BTree

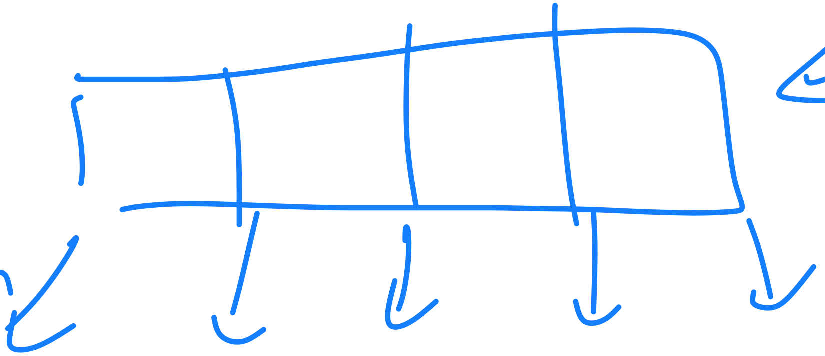
Seek is expensive

↳ cache up

memory is slow

B/c much data in one location

List of keys



List of children

Tree height is flat & tree is wide

BTree Recursive Insert

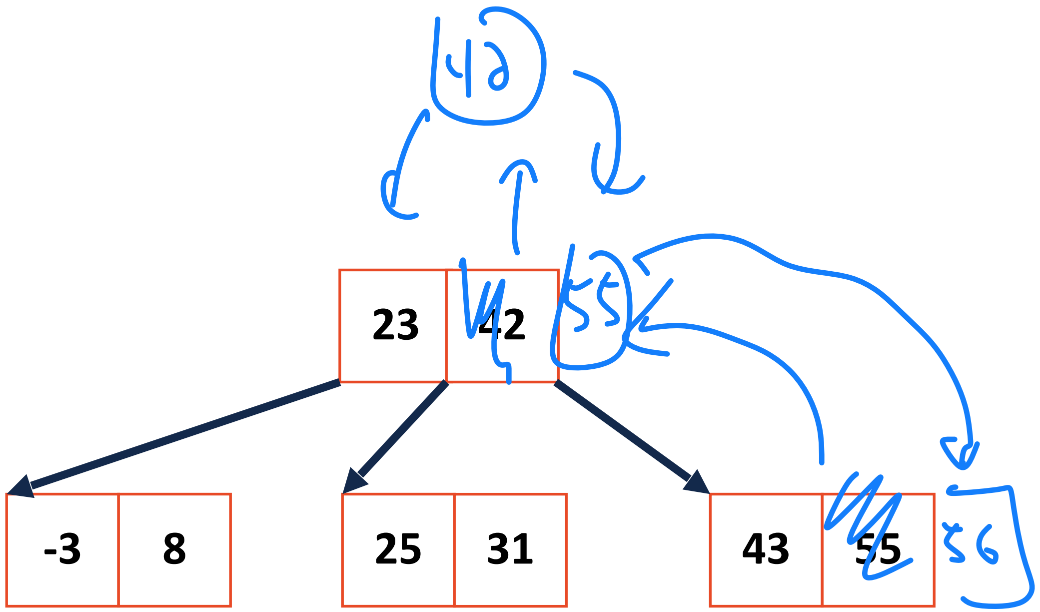
Insert(56), M = 3

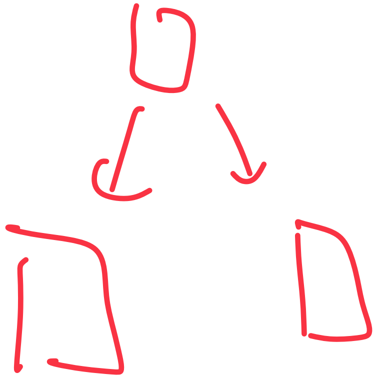
Insert always starts at a leaf but can propagate up repeatedly.

1) Recursively find leaf insert location

2) Recursively split tree up

M=3





BTree Size Restrictions

Faster?

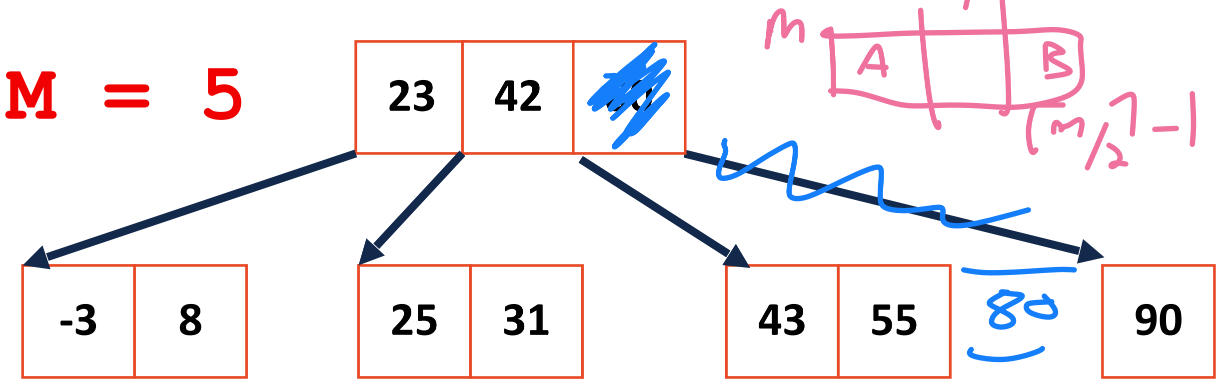
Max

vs Dynamic Array

↳ Slower? But Smaller!

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

M = 5



Not Valid!

← Not yet large enough to split!

↳ Non-root leaves have a min size

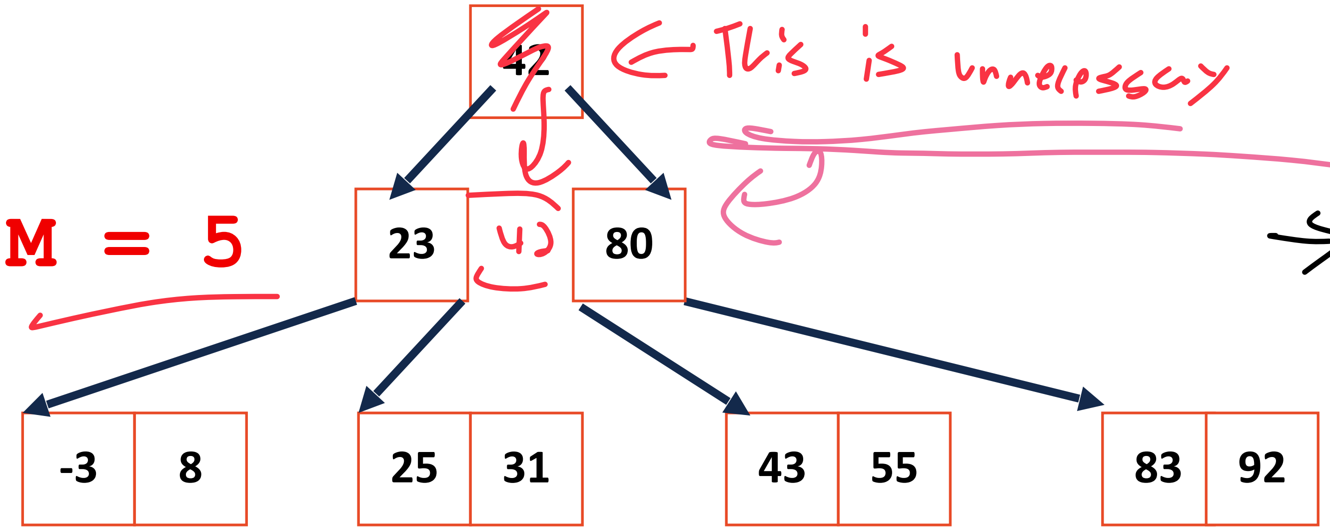
As soon as split once

↳ guarantee on size

(⌈M/2⌉ - 1)

↳ []

M = 5



← This is unnecessary

→ Non-root internal nodes have a min size

↳ M/2 children at least

BTree Properties

review

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

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

at least   at most

Root nodes can be a leaf or have $[2, m]$ children.

Derived from insert

All non-root, internal nodes have $[\lceil \frac{m}{2} \rceil, m]$ children.

at least   at most m

??

All leaves in the tree are at the same level.

BTree

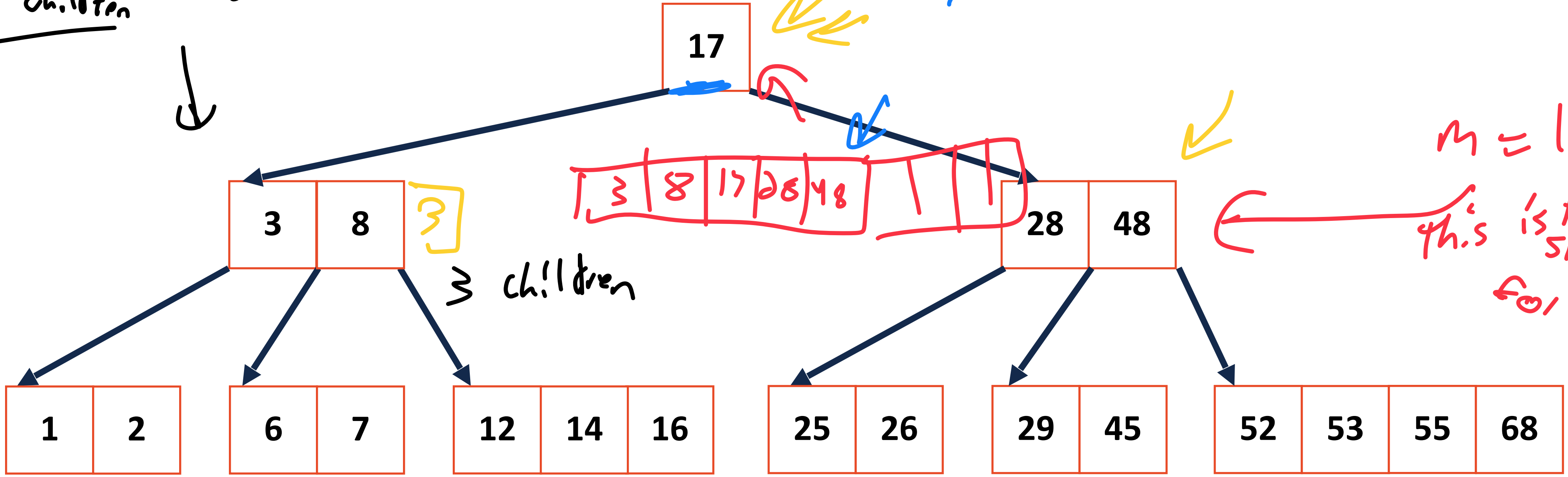
m = 5
2.5 ✓ → 3
~~6~~
3 ✓
7
3.5 → 4 X
$5 \le m \le 6$

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

Lower bound on children $\lceil \frac{m}{2} \rceil$

↳ write bounds on m

$\lceil \frac{10}{2} \rceil = 5$



3 children

3 | 8 | 17 | 28 | 48

m = 10 7

this is too small for m=10 min size

This leaf has 4 values

m ≥ 5

m > 4

BTree ADT

Constructor

Insert

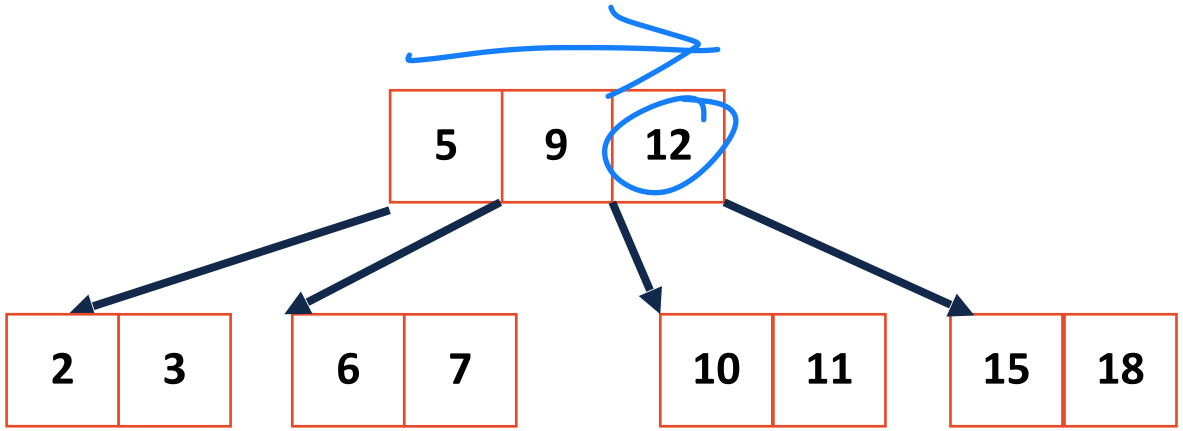
Find

Delete

BTree Find

Find(12)

1) Walk through array
   each node is a list
   ↳ use Array find()



BTree Find

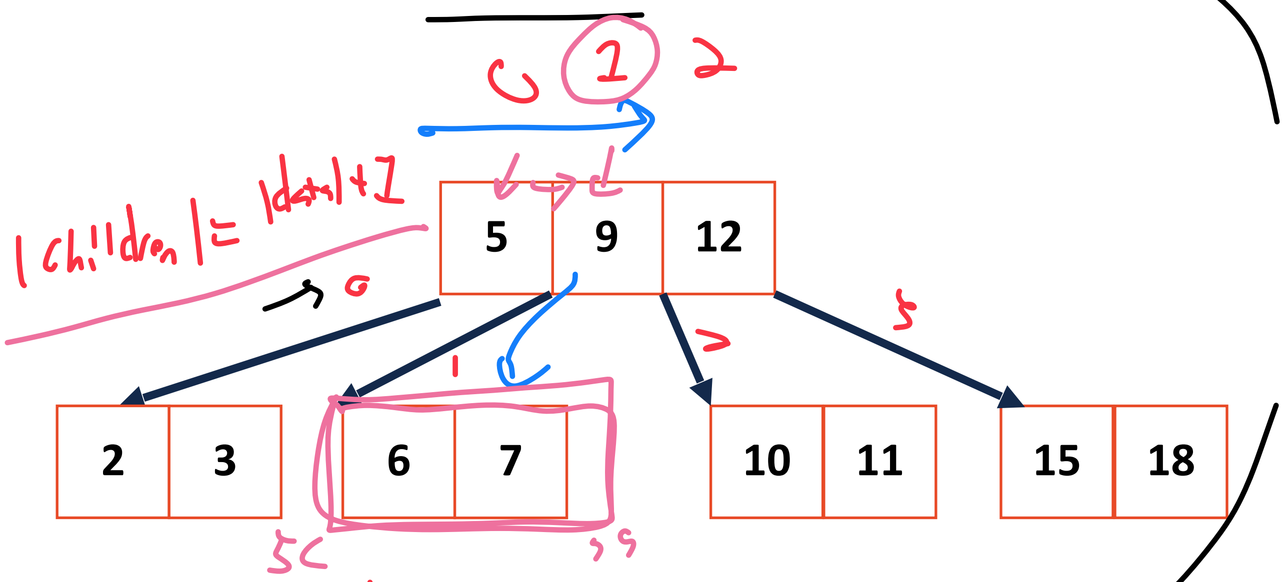
Find(7)

1) Array find()
2) Descend down appropriate child

BTree Node
↳ vector<Keys> data
vector<BTreeNode*> children

Can we do BS on array find?
↳ Yes!

where is first larger value?
i = 1



where is correct child?
i = 1

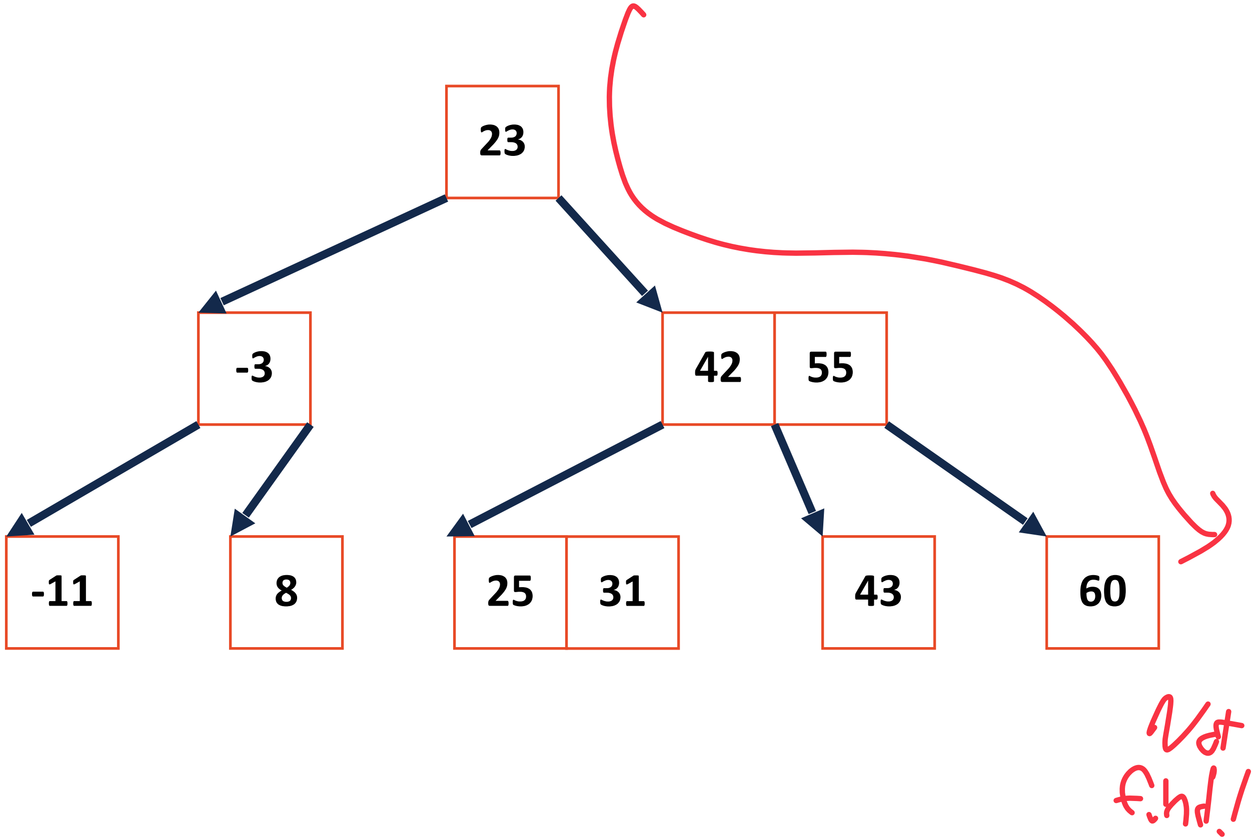
BTree Find

Find(70)

Must distinguish my base case of recursion

↳ Leaf?

↳ Null ptr?



BTree *Exists*

find is on lab :)

cute :) trick

```
bool Btree::_exists(BTreeNode & node, const K & key) {

  unsigned i;
  for ( i = 0; i < node.keycount_ && key > node.keys_[i]; i++) { }

  if ( i < node.keycount_ && key == node.keys_[i] ) {
    return true;
  }

  if ( node.isLeaf() ) {
    return false;
  } else {
    BTreeNode nextChild = node._fetchChild(i);
    return _exists(nextChild, key);
  }
}
```

No body

run off my array

query & is not

case for match!

?? escape clause

Base case

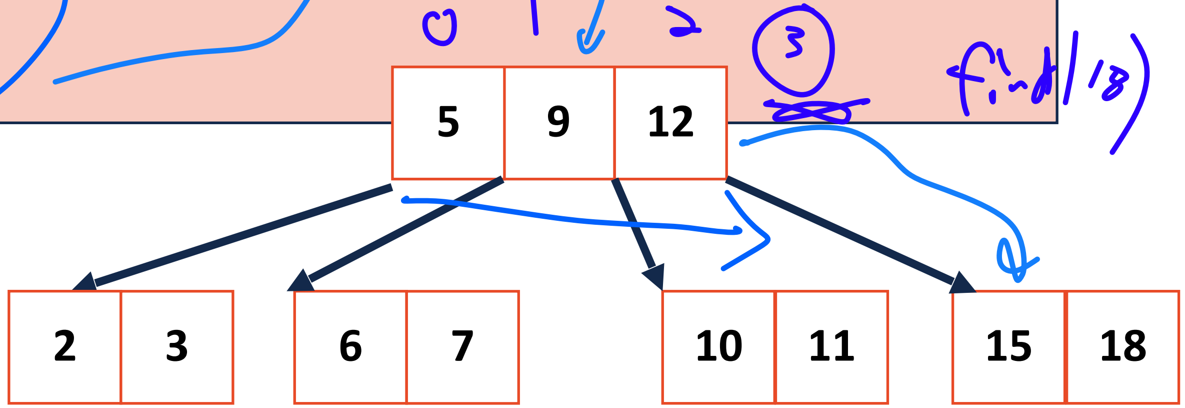
Left as exercise

Standard recursion

find(15)

or next value too large

find(18)



1) Reminder that fetch is slow

2) In class don't over think it

BTree *Exists*

Find (42)

empty body

```
bool Btree::_exists(BTreeNode & node, const K & key) {

  unsigned i;
  for ( i = 0; i < node.keycount_ && key > node.keys_[i]; i++) { }

  if ( i < node.keycount_ && key == node.keys_[i] ) {
    return true;
  }

  if ( node.isLeaf() ) {
    return false;
  } else {
    BTreeNode nextChild = node._fetchChild(i);
    return _exists(nextChild, key);
  }
}
```

40 >3

i = 0 → 42>23 ✓

i = 1 → 1<0 ✗

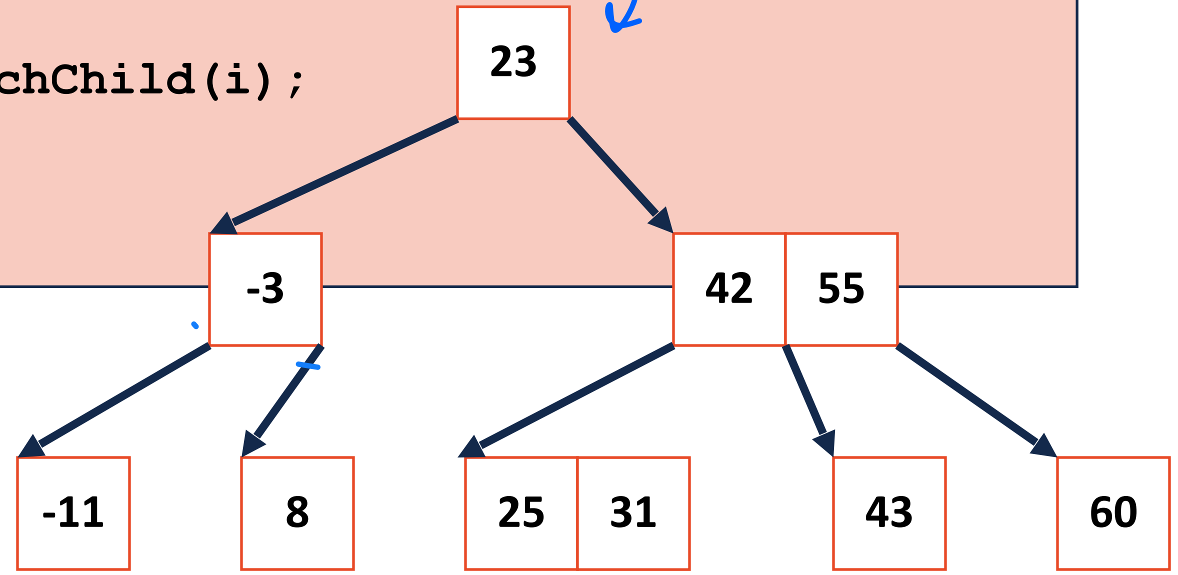
check for exact match

i = 1

escape case

No seg fault here!

No i = 1



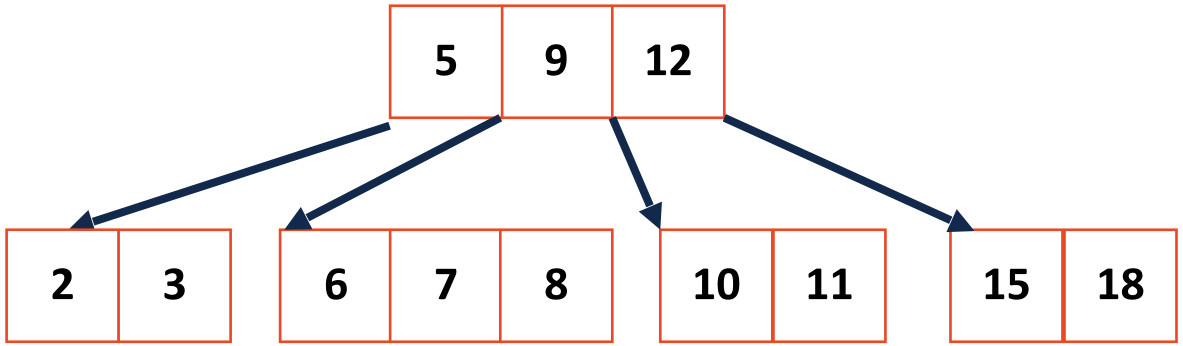
BTree Remove

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

However lets consider how we would handle the following cases…

m is our max # of children

m-1 is max # of keys



BTree Remove

Remove (8)

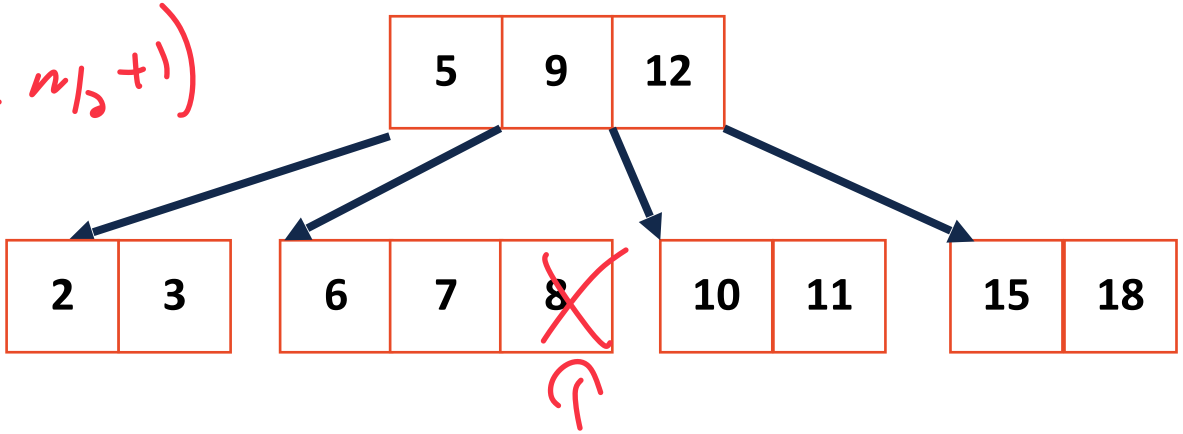
1) Find node

If node is a leaf And

my node is less than m/2

(at least m/2 +1)

2) This is array remove

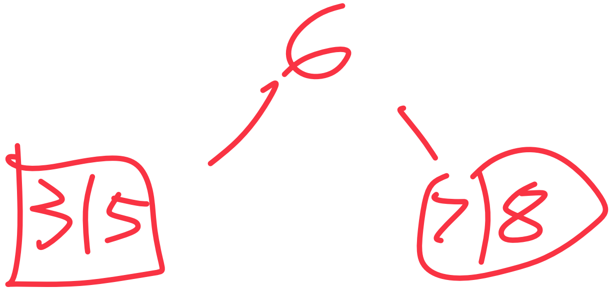


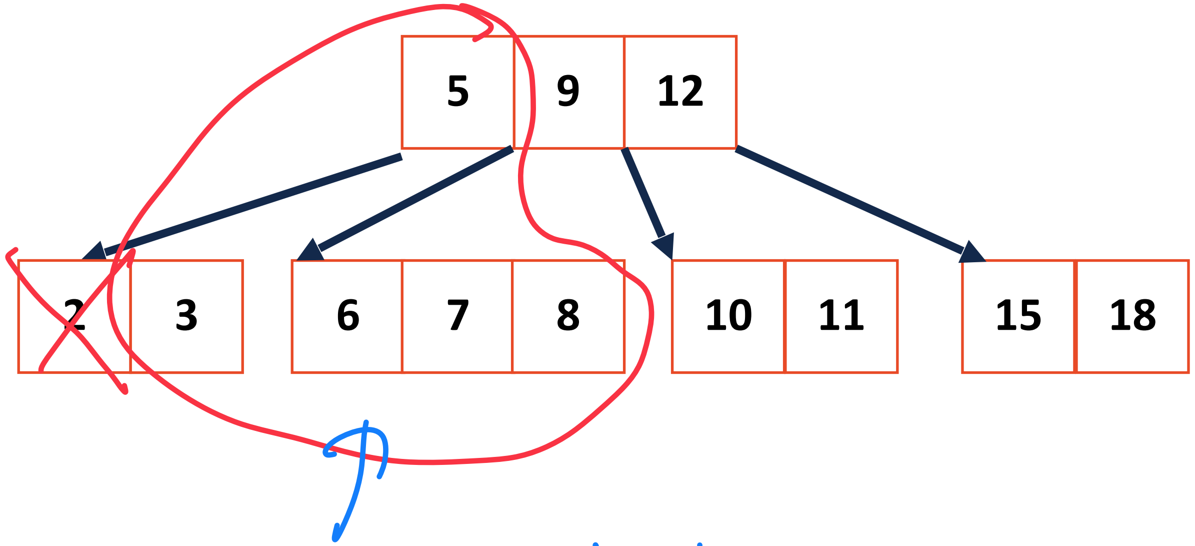
BTree Remove

M = 5, Remove(2)

If leaf is too small, adjust tree

↳ A quick rebalance



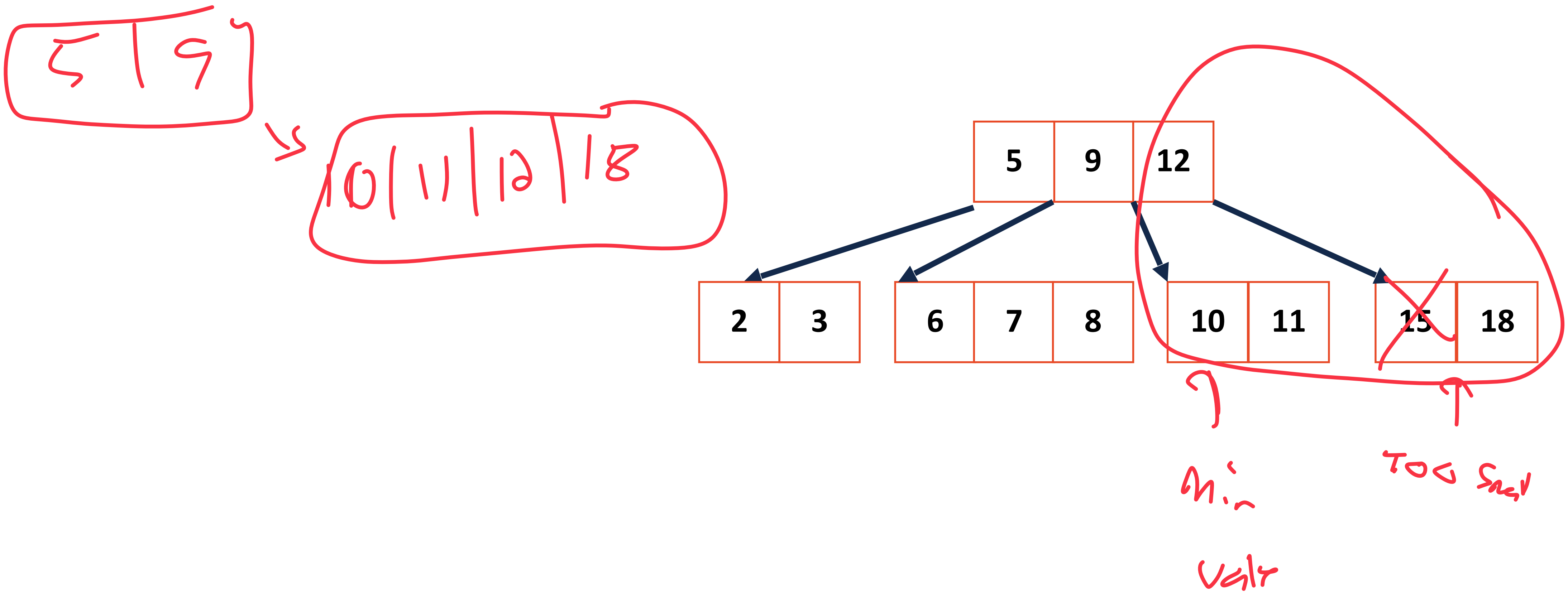


This child has extra values!

BTree Remove

M = 5, Remove(15)

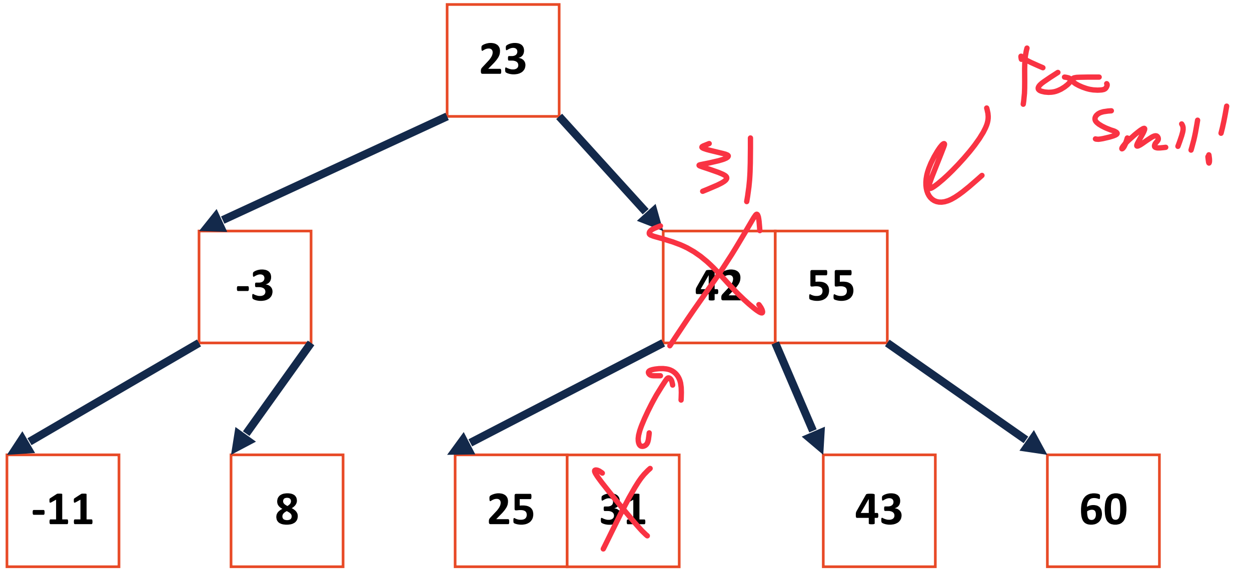
If not enough values, delete a node above



BTree Remove

M = 3, Remove(42)

Sometimes if internal node → can find <u>IOP</u>



BTree Remove

M = 3, Remove(5)

↳ Sometimes better to rebuild tree from scratch

if

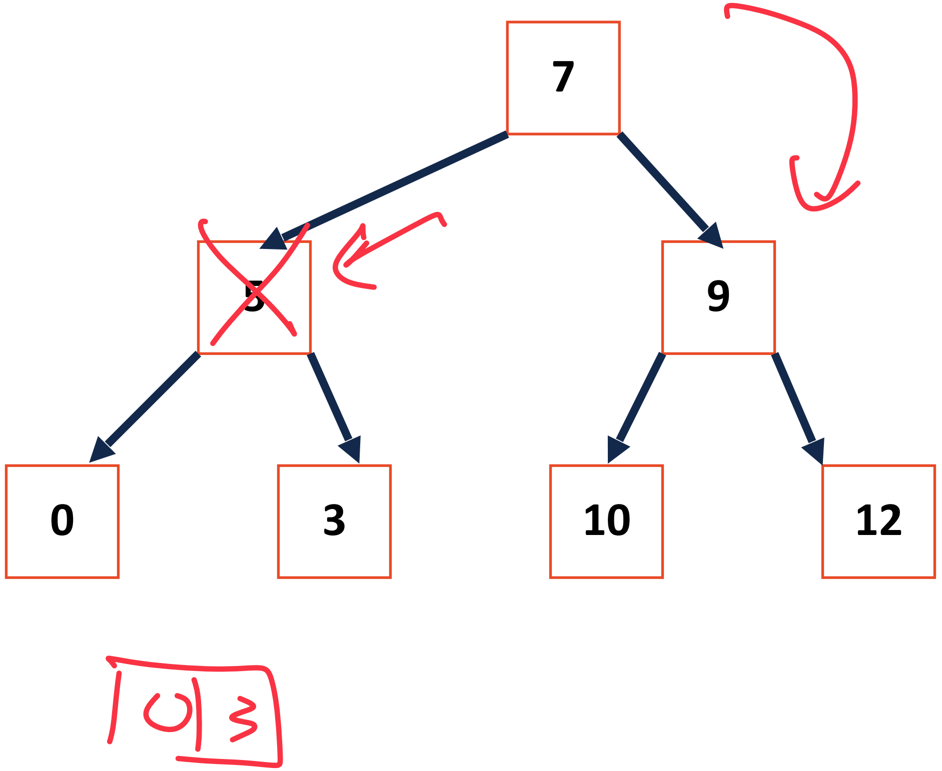
if

if

if

if

if



BTree Analysis

n is # keys

We've seen the ADT

All ops O(h)*

What is the runtime for BTree operations (ignoring remove)?

Find ( ) → O(

↳ (m log n) → O(log n)

height ↓ log_m(n)

m · h

more correct we will prove on Monday

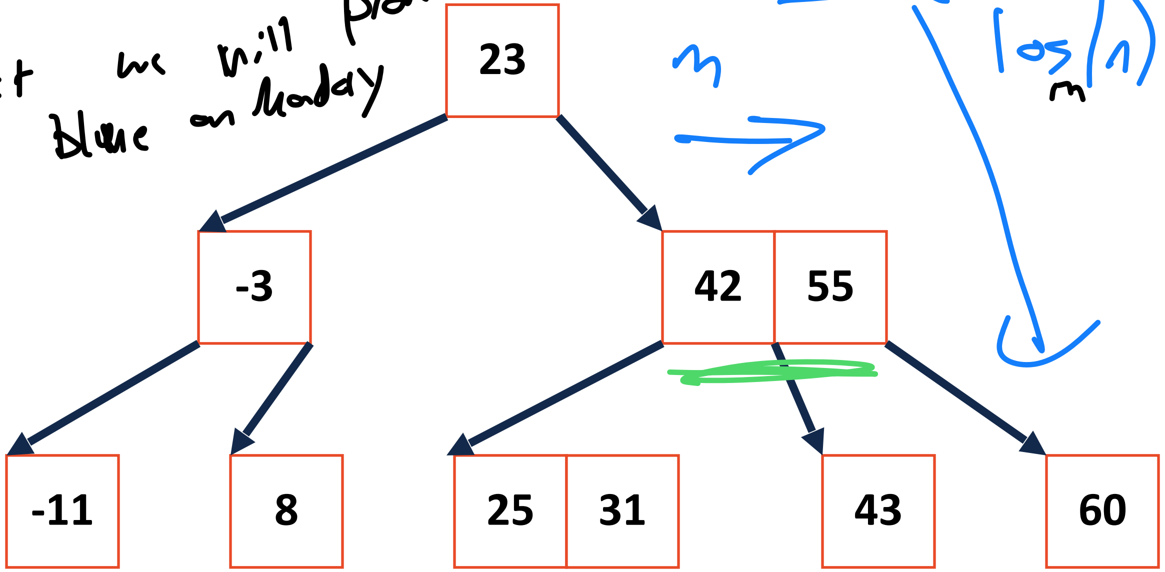
blue

In both cases we drop m term

m is a constant we control

(log m) · h also fine (w/ binary search)

m →



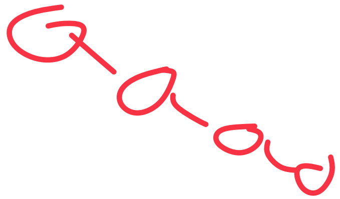
BTree Analysis

We saw for AVL that 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**).

Keys not nodes!

BTree Analysis



N nodes

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

...and the height of the structure is: $O(\log_m n)$.

Therefore: The number of seeks is no more than $O(\log_m n)$.

we control m

...suppose we want to prove this!

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.

Key Facts:

Root nodes can be a leaf or have **[2, m]** children.

All non-root, internal nodes have **[ceil(m/2), m]** children.

BTree Analysis

Minimum number of **nodes** for a BTree of order m **at each level:**

Root:

Level 1:

Level 2:

Level 3:

Level h:

BTree Analysis

$$t = \lceil \frac{m}{2} \rceil$$

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

$$1 + 2\sum_{k=0}^{h-1} t^k$$

$$\sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}$$

BTree Analysis

$$t = \lceil \frac{m}{2} \rceil$$

The **total number of nodes**:

$$1 + 2\frac{t^h - 1}{t - 1}$$

The **total number of keys**:

BTree Analysis

$t = \lceil \frac{m}{2} \rceil$

The **smallest total number of keys** is: $2t^h - 1$

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

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

BTree Analysis

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

Minimum Keys:

Maximum Keys:

BTree

The BTree is still used heavily today!

Improvements such as B+Tree and B*Tree exist far outside class scope

Thinking conceptually: Sorting a queue

How might we build a ‘queue’ in which our front element is the min?

Thinking conceptually: A tree without pointers

What class of (non-trivial) trees can we describe without pointers?