CSCI 2270

Data Structures & Algorithms

Gabe Johnson

Lecture 15 Feb 18, 2013

Test-Driven Design
B-Tree Intro

Lecture Goals

1. Exam 1 Results
2. Design Saves Lives
3. Test-Driven Design
4. B-Tree Operations

Upcoming Homework Assignment

HW #4 **Due: Friday, Feb 22**

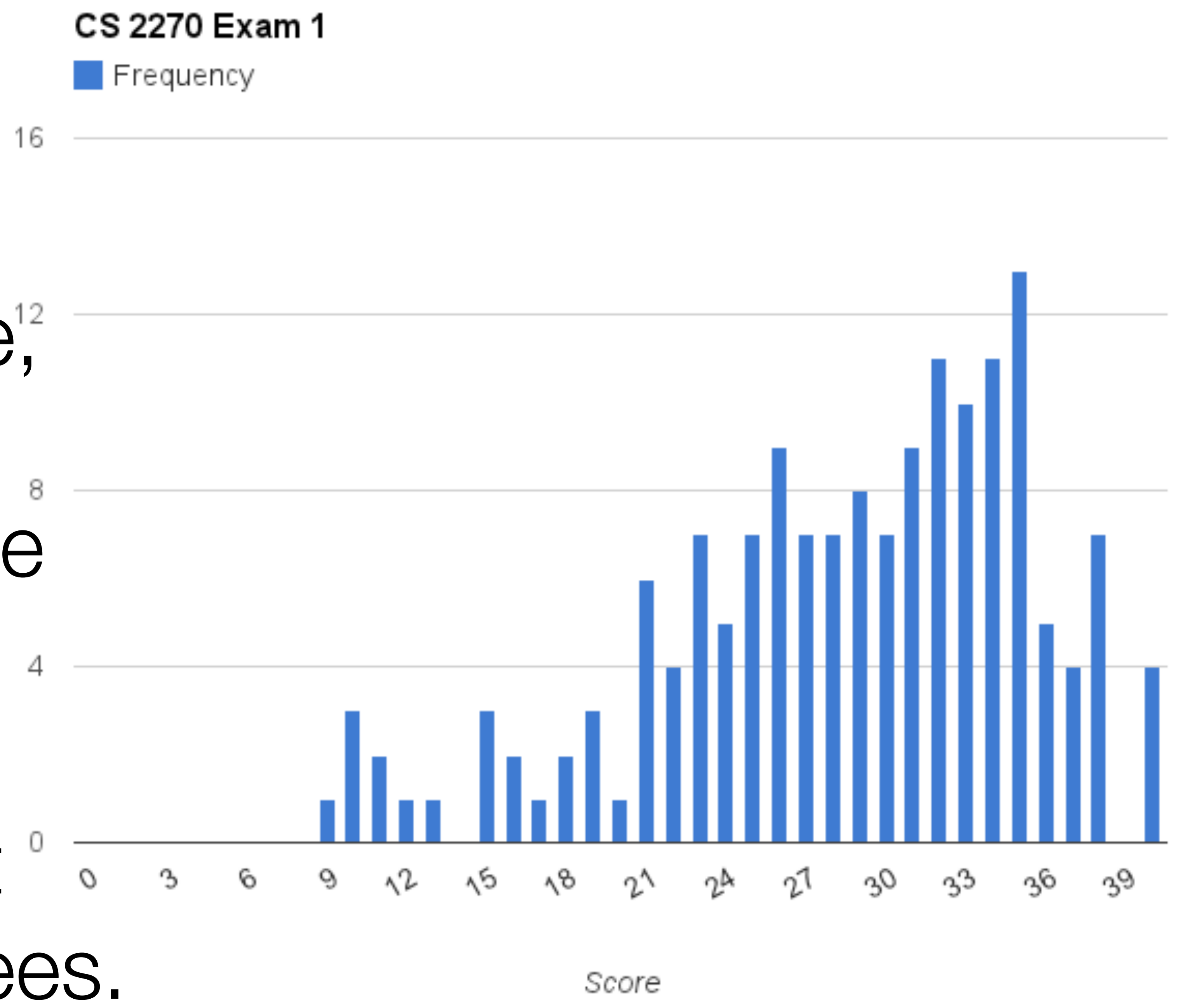
B-Tree Design Document

The B-Tree assignment is broken into two steps, and each are worth the full 15 points. The first step is to design an implementation strategy. How are you going to go about making a very complicated data structure and associated operations? You will make a ***plain text*** design document and email it **as an attachment** to your TA by 6:00pm on Friday and **you must** use the string **[b-tree]** in the **subject line**. More instructions later.

Exam 1 Results

Mean: 28.4
Std Dev: 7.1

If you got 27 or above, keep doing what you are doing and you'll be fine, most likely. This class is front-loaded with the tough stuff. It gets easier after B-Trees.



Sample Solution On GitHub

The solutions are up on GitHub in the tests/ directory. Take a look and if you think you can make a case for getting points back, contact your TA with a ***short***, *coherent* email message with your argument.

Design Saves Lives

Read about the radiation accidents involving a machine called “Therac-25”.

This is a medical device that is supposed to dose patients with a relatively low amount of radiation in most use cases.

Most. Not all.

Therac Design Flaws

Therac 25 has often given as an example of deeply flawed software design *and* user interface design.

For our purposes: the software that controls this thing was *not* designed or developed in a way that controlled software testing could be performed.

The safeties were supposedly done in software. But there were conditions where the safeties didn't work. This caused severe radiation poisoning and deaths.

Test-Driven Design Helps

One way to reduce (not eliminate) buggy software is to determine in advance how something should work, what the data structure and algorithmic invariants are, what the possible use cases are, and what environmental or situational conditions might alter performance.

Avoid Hubris

Requirements are hard.

Coming up with this list is a job of itself, so many programmers skip this step because they think they are Teh Best Coderz Evah™, but they're wrong.

There are many reasons to skip this step. Hubris is not one of them.

Requirements → Tests

When we have some requirements (like function and data structure invariants), we have a great starting place for writing tests:

```
/**
  * This function returns the sum of squares of the input array.
  **/
  int get_sum_of_squares(int[] input) { ... }
```

Without looking at this code, we can gather some requirements based on the documentation, and what we know from 7th grade math class.

1. What’s it do?

```
/**
  * This function returns the sum of squares of the input array.
  **/
  int get_sum_of_squares(int[] input) { ... }
```

We know it is going to compute some squares, and add them together.

2. What’s the input?

```
/**
  * This function returns the sum of squares of the input array.
  **/
  int get_sum_of_squares(int[] input) { ... }
```

We see from the function signature that it takes an array of integers, conveniently called ‘input’.

Think about what could be *wrong* with this input. Empty? Very big? Negative numbers? Zeros? Some of these might be problems, others might not be. You have to engage your brain.

3. What's the output?

```
/**
  * This function returns the sum of squares of the input array.
  **/
  int get_sum_of_squares(int[] input) { ... }
```

We see from the function signature that it returns a single integer. So we know that inside this function definition we'll need a *return* statement, and we have to guarantee that it actually runs.

Also: we know that squares are all positive, so the result should also be positive.

4. What potential problems?

```
/**
  * This function returns the sum of squares of the input array.
  **/
  int get_sum_of_squares(int[] input) { ... }
```

Say we get input that is very large. Maybe one of those numbers is so large that when we take its square, we don’t have enough room inside this computer’s *int* data type to store the result. Or, maybe we get that problem when we start adding the squares together.

5. Faulty Definition?

```
/**
  * This function returns the sum of squares of the input array.
  **/
  int get_sum_of_squares(int[] input) { ... }
```

We don't even know how much data is contained in the *input* array. In some languages we don't need this info passed in explicitly (Java, Python) but others we must be given the input size. Maybe the person who gave you this definition messed up? Maybe they spend their days hacking Java and it didn't occur to them that C++ is different. This is *always* a possibility.

Write Some Tests

```
/**
  * This function returns the sum of squares of the input array.
  **/
  int get_sum_of_squares(int[] input) { ... }
```

All of the observations we just made could potentially be turned into an external regression test (also known as a unit test). Other observations could be the basis of sanity checks inside your code.

Tests based on assertions

Most testing is based on the idea of assertions.

You assert that something is true, and if it is, don’t say anything. Only make noise when something is wrong. This makes it easy to spot problems.

In many languages/tools assertions can pinpoint problems by line number or by function.

`assert 7 < 10` this says nothing because it is ok.
`assert 7 > 10` this will tell you about a problem

Write Tests First

```
test_negative_sum()
    data = array [5, 3, 0, 9, -24]
    result = get_sum_of_squares(data)
    assert(result >= 0)

test_empty_input()
    result = get_sum_of_squares(empty array)
    assert result is 0

test_right_answer()
    result = get_sum_of_squares(array [3, 4, 5])
    assert result is 50
```

Then Write Code

```
/**
* This function returns the sum of squares of the input array.
**/
int get_sum_of_squares(int[] input) {
   // happy code goes here, because if we screw up, our
   // tests will tell us where and what.
}
```

Unit Testing

Software Engineers use this technique to a pretty extreme degree. We call it variously *Unit Testing, Regression Testing,* and there are loads of frameworks to help you do this. The most famous is called JUnit, by Kent Beck and Erich Gamma.

You can find C++ unit testing frameworks sitting around. Google has one. There's CPPUnit (cousin of JUnit). And the C++ unit testing we use in class was written by Alec Thilenius.

Don’t Need a Framework

For the B-Tree assignment, you won’t need to use a fancy framework (though, that would be awesome if you did).

You *will*, however, need to write functions that tell you if your B-Tree works. This includes all the invariants (e.g. the rules imposed on nodes and their relationships). Write code that tests for each invariant, and use the tests to ensure your code does the right thing after each insert/remove operation.

For Example: Size

Without knowing very much about B-Trees, we do know from experience with other trees that we can get a tree's size, and that adding new nodes should increase the tree's size.

```
test_insert_and_size (b_tree):
    initial_size = b_tree.size()
    b_tree.insert(42)
    assert b_tree.size() = initial_size + 1
```

Tests Test Your Assumptions

The previous slide was based on the assumption that adding the key *42* would cause the tree size to increase. But if our B-Tree does not allow duplicate keys (and ours does not), then we have a problem. So if you write test code and its assertions fail, the problem could be the B-Tree code, or it could be the assumptions encoded in your unit test.

Once faulty assumptions are recognized we can strengthen everything by updating our tests first.

Fixing Our Assumptions

```
test_insert_and_size (b_tree):
    initial_size = b_tree.size()
    has_42 = b_tree contains key(42)
    b_tree.insert(42)
    if has_42,
       assert b_tree.size() = initial_size
    otherwise,
       assert b_tree.size() = initial_size + 1
```

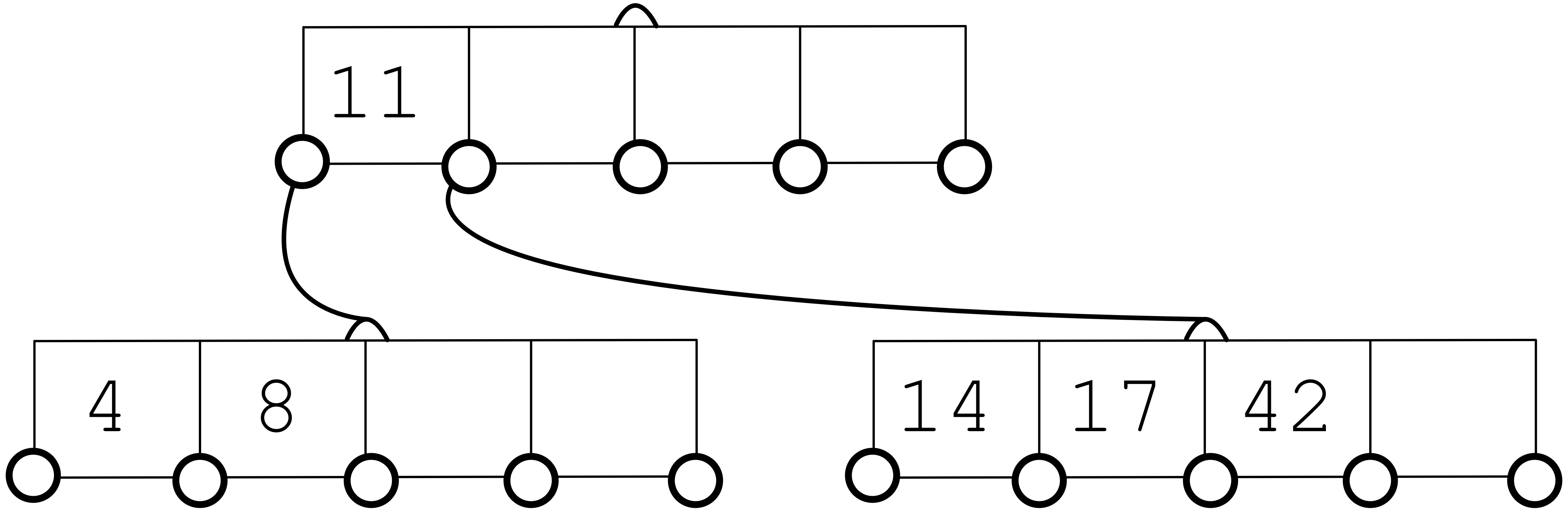
Decouple Tests If You Can

This test depends on the code that tells us if the tree contains a key. But we can’t rely on that has_key function working or not. It is hard to eliminate this, but there are techniques (that you’ll learn in a proper software engineering course).

```
test_insert_and_size (b_tree):
    initial_size = b_tree.size()
    has_42 = b_tree contains key(42)
    b_tree.insert(42)
    if has_42,
       assert b_tree.size() = initial_size
    otherwise,
       assert b_tree.size() = initial_size + 1
```

B-Tree Recap

Review the initial discussion of B-Trees from last Wednesday (Lecture 13).



B-Tree Invariants

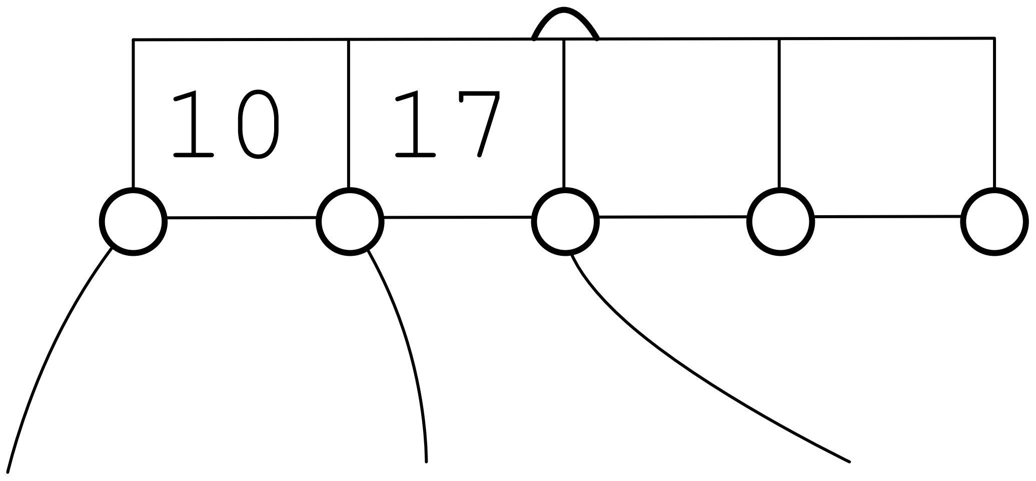
1. Every node has at most **m** children.
2. Every non-leaf node except root has at least **ceil(m/2)** children.
3. The root has at least **two** children if it isn't a leaf.
4. A node with **k** children contains **(k–1)** keys.
5. All leaves appear in the same level.

Node Definition

B-Tree Nodes:

Nodes have the following fields:

-> **num_keys**: number of keys the node is currently using.
-> **keys**: array of actual key values.
-> **is_leaf**: is the node a leaf? true/false.
-> **children**: array of child pointers. should be (num_keys + 1) of them.



```
num_keys = 2
keys     = [10, 17]
is_leaf  = false
children = <3-array>
```

B-Tree Documentation

This summary of B-Tree theory and implementation is good. I will base my RG tests on it.

D. Zhang. ***B Trees.*** Chapter 15 In Handbook of Data Structures and Applications, D. P. Mehta, S. Sahni (editors), Chapman & Hall/CRC, ISBN 1-5848-8435-5, 2004.

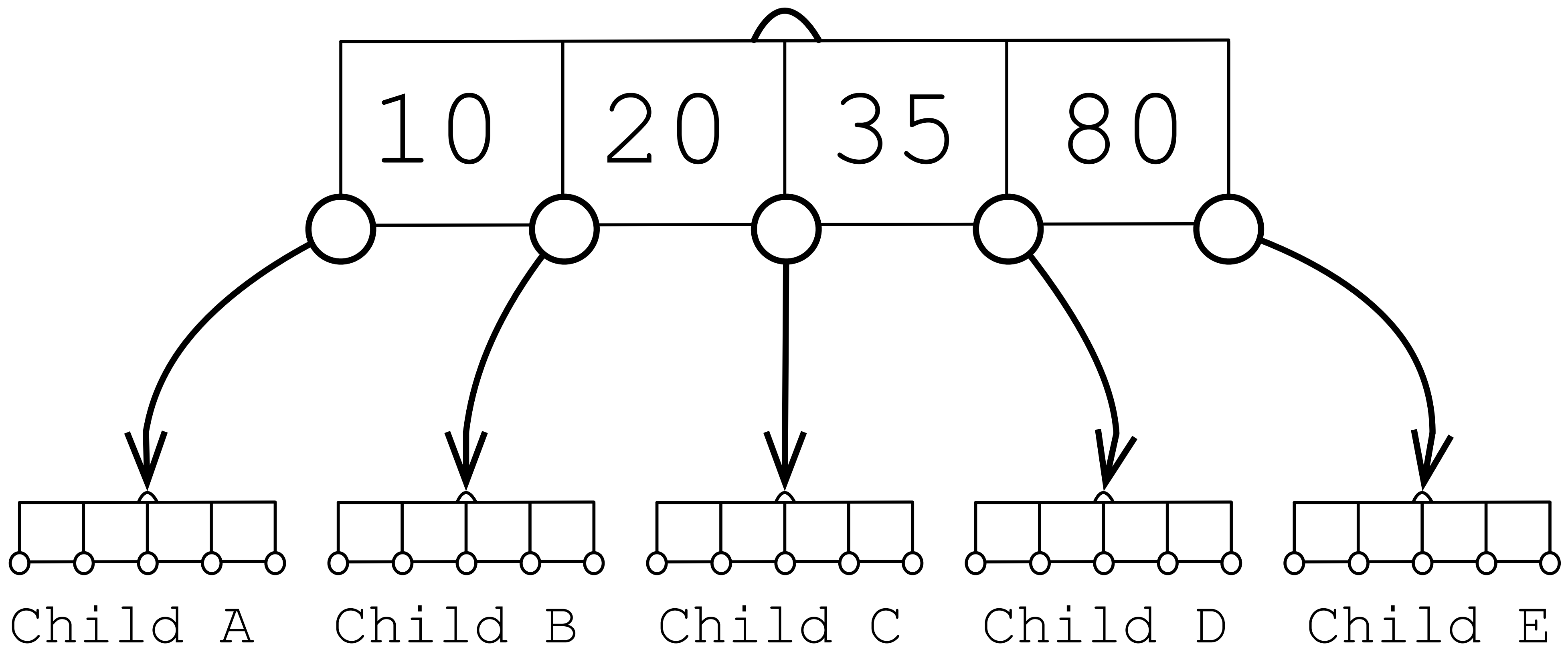
There is a link from the HW 4 directory on GitHub to the PDF of this.

B-Tree Query

We may be interested in knowing if the tree contains a given key. Or, we may want to get the value that key refers to.

Query Example

does the b-tree contain the key 35?
how about the key 63? where would we look?
how would we use the b-tree node data structure to determine which child to look at?



For Next Time:

Read **both** the PDF I reference from the HW 4 directory, **and** the Wikipedia page on plain B-Trees. We will do query, insert, size (two kinds) and delete next time.

There are confusing and missing parts to both. Part of this trip is recognizing when the directions are inadequate.

E.g. in the Zhang document, the author refers to 'the middle node'. How precisely is this found, and *which data does it search?*