CSCI 2270

Data Structures & Algorithms

Gabe Johnson

Lecture 13 Feb 13, 2013

Complexity
Introduction to B-Trees

Upcoming Test

Friday, Feb 15

Test 1

There is a directory with sample questions up on GitHub. Test is out of 40 points and accounts for about 12% of your total grade. We have a small army of graders so it is not impossible that your grade will appear on RetroGrade on Sunday.

Lecture Goals

1. Recap of Complexity
2. Questions for Test 1 Review?
3. Intro to B-Trees

Computational Complexity

Computational Complexity analysis is one of the most important intellectual tools you'll learn. Don't just be able to get the right answer. Understand the intuition behind why this is interesting, and how it can save you a *lot of mental duress*. If you know in advance that an algorithm is fast, slow, intractable, then you can plan your life around it.

Consider the length of time it takes to perform basic operations using real hardware...

Typical Operation Times

execute typical instruction	1/1,000,000,000 sec = 1 nanosec
fetch from L1 cache memory	0.5 nanosec
branch misprediction	5 nanosec
fetch from L2 cache memory	7 nanosec
Mutex lock/unlock	25 nanosec
fetch from main memory	100 nanosec
send 2K bytes over 1Gbps network	20,000 nanosec
read 1MB sequentially from memory	250,000 nanosec
fetch from new disk location (seek)	8,000,000 nanosec
read 1MB sequentially from disk	20,000,000 nanosec
send packet US to Europe and back	150 milliseconds = 150,000,000 nanosec

Source: Peter Norvig. http://norvig.com/21-days.html

Complexity Example

```
for (int i=0; i < N; i++) {
 for (int j=0; j < N; j++) {
  product[i][j] = i * j;
 }
}
```

What's the computational complexity of this algorithm? How many times will we run the inner loop?

We say this is **$O(n^2)$** because the runtime of the algorithm *is proportional to the square of the input size.*

Complexity: best, avg, worst

We're often concerned about the best, the average (or typical), and worst-case complexity of an algorithm. We can optimize for each, based on whatever situation we have. We can roll our own algorithm or we can use something well-known.

When to optimize

Best case: optimize for situations where we have reasonable understanding of the data, and know the best case will be common.

Average case: optimize for the typical case when input is not particularly weird, but we don't know what is coming. This is what most algos do.

Worst case: optimize worst-case scenario to prevent catastrophes: missile defense, air traffic control, real-time medical hardware, etc.

Sorting Algorithms

We used pointers (by way of C++'s vector template class) and recursion in the sorting homework.

Algorithms differ in complexity in their best/avg/worst case scenarios.

To sort **n** numbers (using best variant of each):

Algo	Worst	Avg	Best
bubble sort	$O(n^2)$	$O(n^2)$	O(n)
quick sort	$O(n^2)$	O(n log n)	O(n log n)
merge sort	O(n log n)	O(n log n)	O(n)

B-Trees. Brought to you by Indiana Jones

B-Trees! Finally.

Imagine you work at a museum. Indiana Jones brings things in from time to time, and it is your job to keep track of everything. He has been known to need access to these things in a mad rush. The problem is, there's a *lot* of stuff to keep track of.

And if Indiana can't get the thing he needs right now, the world may end in some biblical apocalypse.

Small Piles of Stuff

At first the museum collection is small, so you can put it all on the table next to your desk.

Over time the desk becomes cluttered, and you can't spot things easily. You need an organizational system.

You need an index. And rather than putting things on a desk, you need storage shelving.

Shelves of Stuff

You start out with a few shelving units, and keep a sheet of paper that has the artifact description next to its location.

Artifact J7-394 (Mystical Sphere): Unit 4, Shelf 2.
Artifact E-032 (Diamond Brain): Unit 2, Shelf 3.

The shelf approach works for a while. Eventually Indiana Jones brings so much stuff to the museum that you need to spread out into different *rooms*.

Shelves Are Not Full

It is time-consuming to move things around on the shelves all the time, so your strategy is to try to keep the shelves at about the same level of fullness, and every time you have to add a new shelf, you make sure it is only half-full. That way when new items come in, you don't have to do the expensive, time-consuming process of shuffling stuff around.

Rooms of Stuff

You expand your storage operation into separate rooms, much to the dismay of the graduate students who were working there and now have to hide under the awning on the roof. Add another layer to your index strategy. And of course, things keep moving around, since, you know, you have to keep all the mystical spheres together.

Artifact J7-394 (Mystical Sphere): Room 1, Unit 2, Shelf 1.
Artifact E-032 (Diamond Brain): Room 3, Unit 4, Shelf 3.

Now Buildings of Stuff.

The sheer volume of crap our cowboy archaeologist brings now demands the use of separate buildings. The index itself is now a three-ring binder, and you've decided to *index the index* by throwing dividers in there so you know which page to turn to easily.

The B-Tree Allegory

The preceding silly story about Indiana Jones and the Monster Museum of Doom is actually about B-Trees.

Remember that table with the relative times that it takes to perform operations based on what kind of hardware we're using?

The one that looks like this...

Norvig (Reloaded)

execute typical instruction	1/1,000,000,000 sec = 1 nanosec
fetch from L1 cache memory	0.5 nanosec
branch misprediction	5 nanosec
fetch from L2 cache memory	7 nanosec
Mutex lock/unlock	25 nanosec
fetch from main memory	100 nanosec
send 2K bytes over 1Gbps network	20,000 nanosec
read 1MB sequentially from memory	250,000 nanosec
fetch from new disk location (seek)	8,000,000 nanosec
read 1MB sequentially from disk	20,000,000 nanosec
send packet US to Europe and back	150 milliseconds = 150,000,000 nanosec

Source: Peter Norvig. http://norvig.com/21-days.html

Memory = Fast. Disk = Slow.

fetch from main memory	100 nanosec
fetch from new disk location (seek)	8,000,000 nanosec

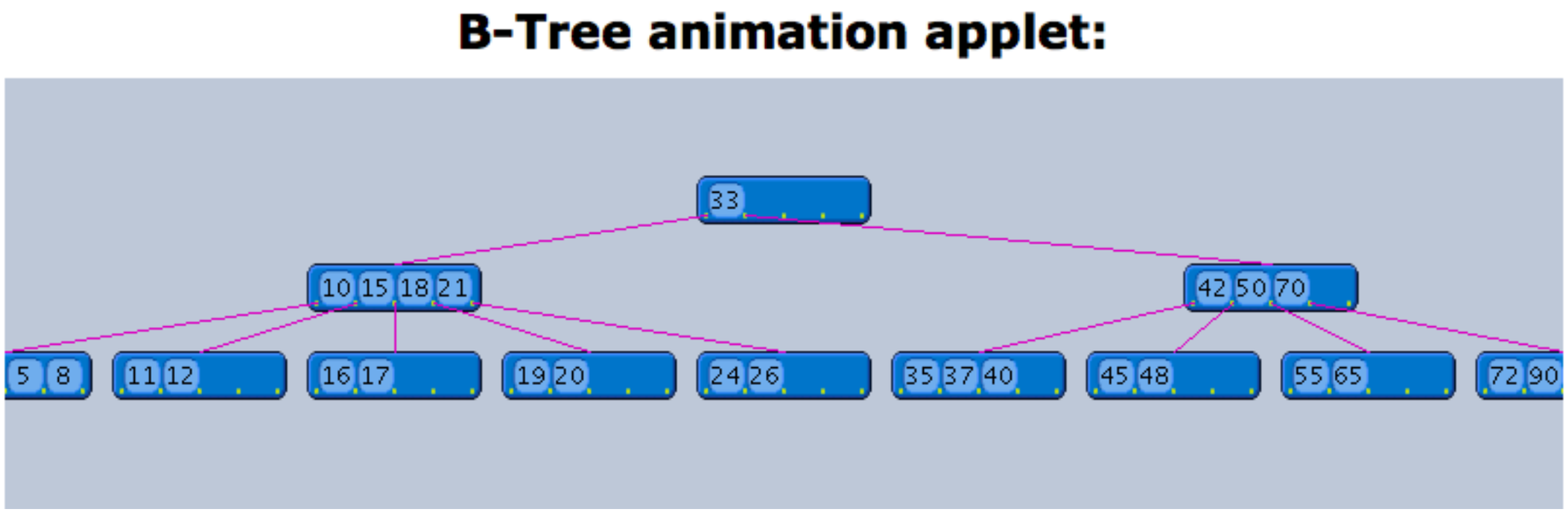
If these numbers are to be believed (and I do), fetching data from disk is 80,000 times slower than fetching from memory.

If a memory fetch is analogous to looking up the location of our data in our 3-ring binder, then a disk fetch is analogous to trudging across town to some warehouse where Artifact J8-382 happens to be.

B-Tree uses

We use B-Trees as a way to optimize our use of fast data stores while minimizing the interaction with slow ones. Classic examples are filesystems (the program that controls reads and writes of files with physical hardware) and databases (the same idea, but possibly spread out over multiple computers in a cluster).

B-Tree Interactive Demo



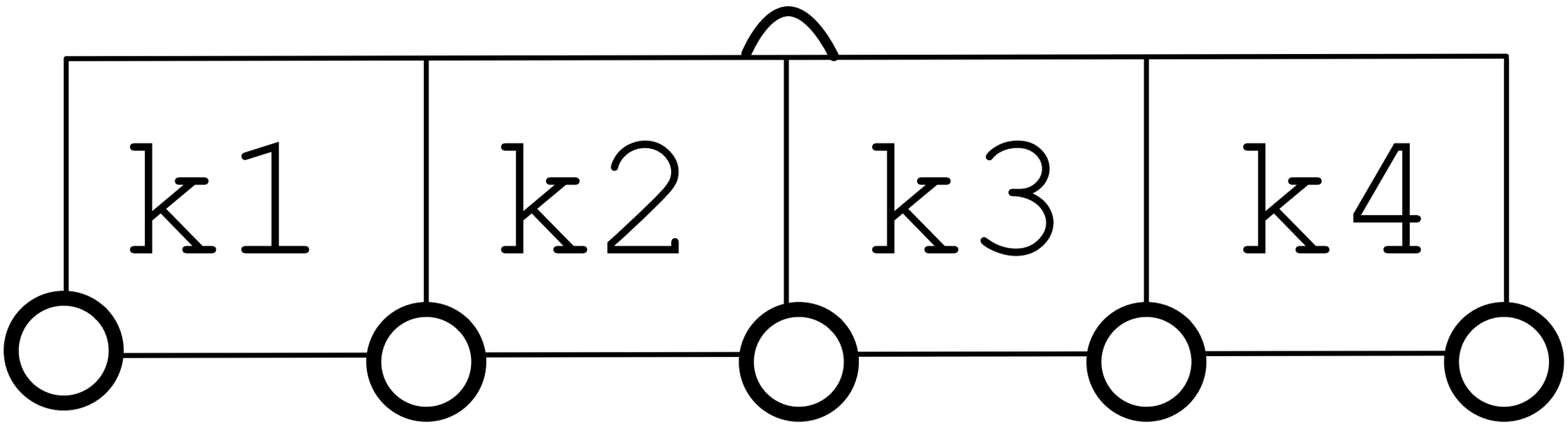
http://slady.net/java/bt/view.php?w=800&h=600

http://www.youtube.com/watch?v=coRJrclYbF4

Main Idea W/B-Trees

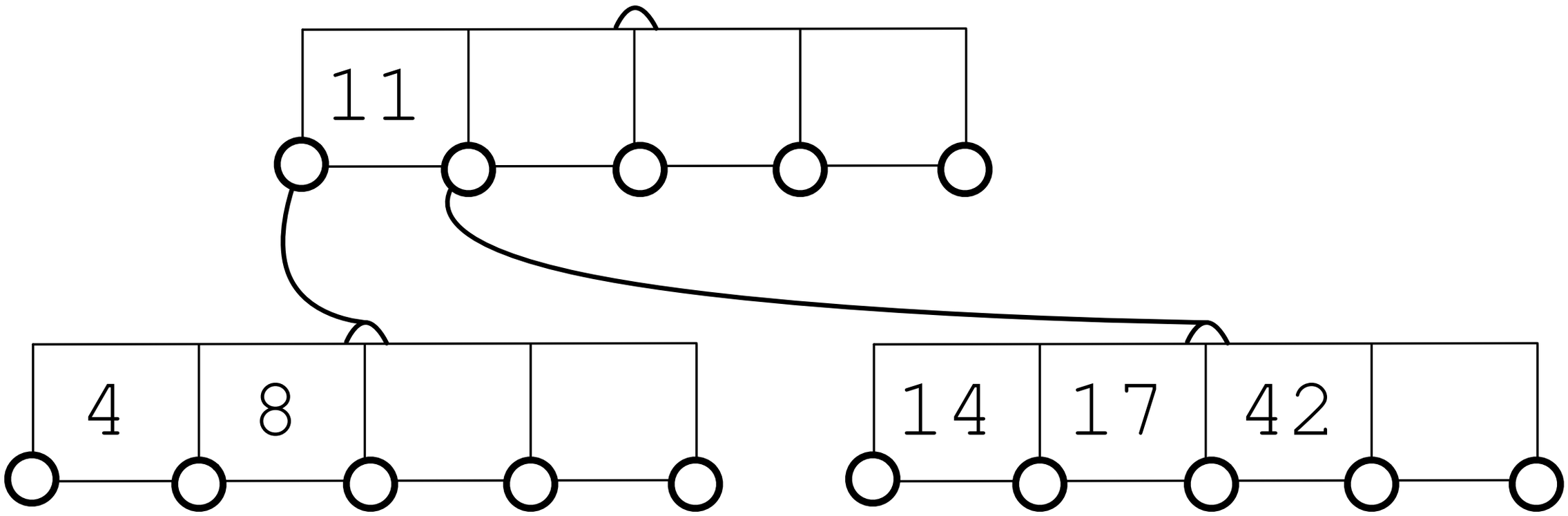
There are *nodes*, *keys, links,* and *information*.

A node has keys inside it. Here's a node with four keys labeled k1 through k4.



Keys and Links

A *Key* is a number that fills up a node from left to right. Notice how I draw links represented as circles that sit between key slots. That key represents a boundary. Everything less than 11 goes left, everything larger to the right. Duplicates can be allowed but for our purposes we won't do dupes.



Keys point to Information

A plain B-Tree's keys are all associated with the information that key refers to. Back to Indiana:

Key: J7-394

Information: Mystical Sphere

B-Tree variants like B+ Trees or B* Trees have different rules. There, the information we're looking for is only referred to in the leaf nodes. We're doing plain B-Trees.

Be aware of this difference as you scour the web.

Omitting the Info Part

For most of the diagrams I will omit the information part, since it would clutter things up. But keep in mind that every key you see has some payload data associated with it.

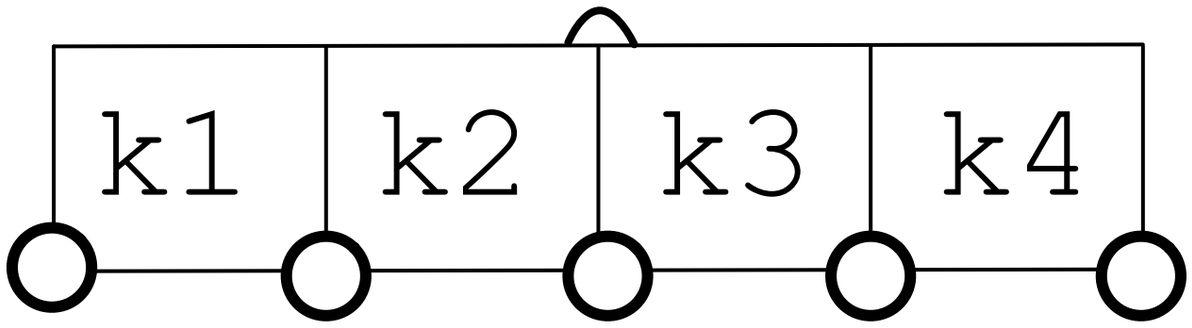
In a real situation the payload is something potentially big. Key 14 could refer to the collected works of Shakespeare. Key 17 could refer to my grocery list.

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 is not a leaf node.
4. A non-leaf node with k children contains k–1 keys.
5. All leaves appear in the same level, and carry information.

B-Tree Invariants:

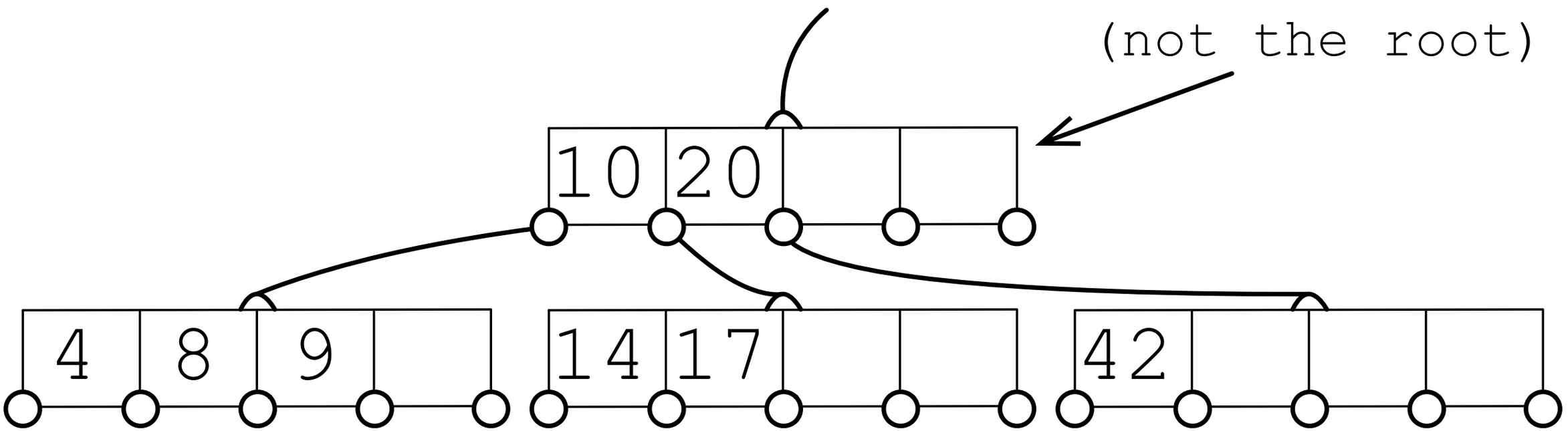
(1) Every node has at most *m* children.



In this case, m=5. There are four keys, which mean there are five slots 'between' these keys. After that, there's no more room.

B-Tree Invariants:

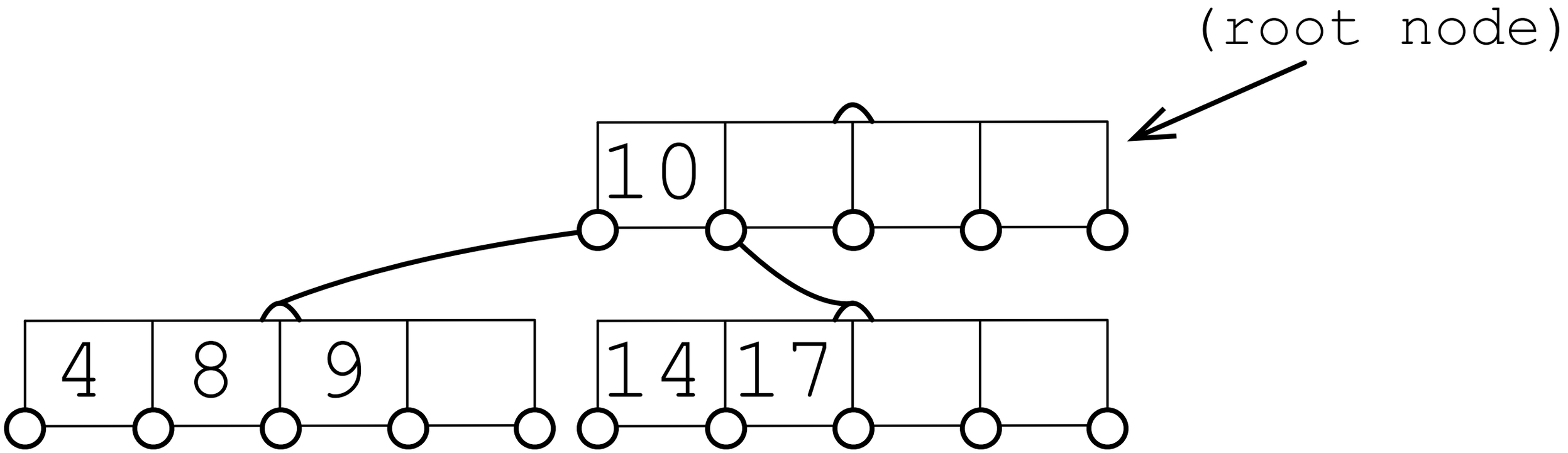
(2) Every non-leaf node has at least ceil(*m/2*) children.



m is the number of allowed children. In this case that is 5. 5/2 is 2.5. Round up to get the minimum number of children = 3.

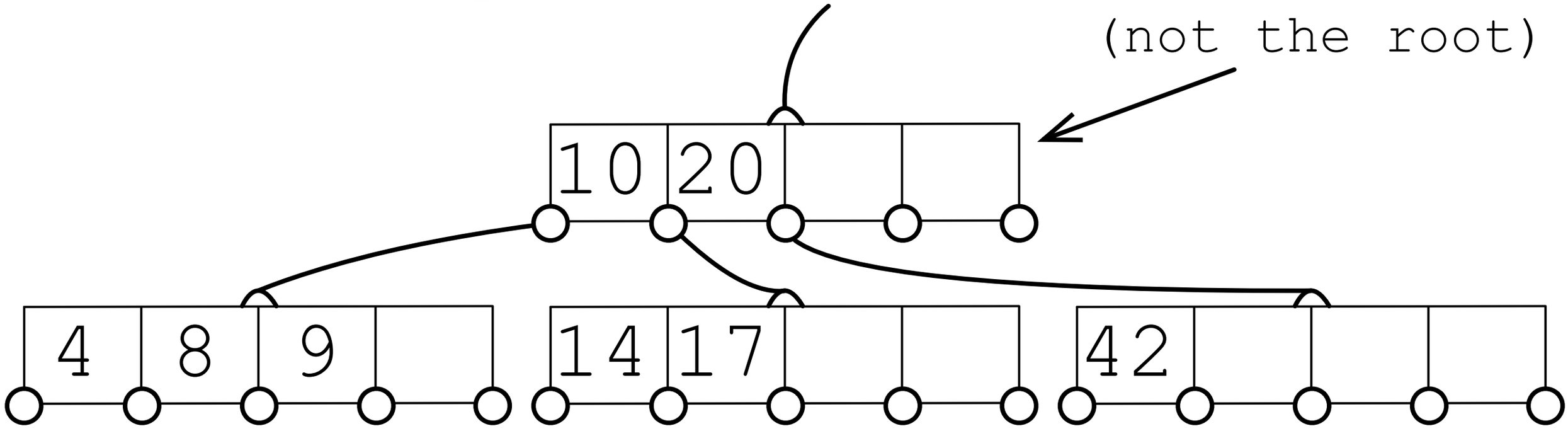
B-Tree Invariants:

(3) The root has at least two children if it is not a leaf node.



B-Tree Invariants:

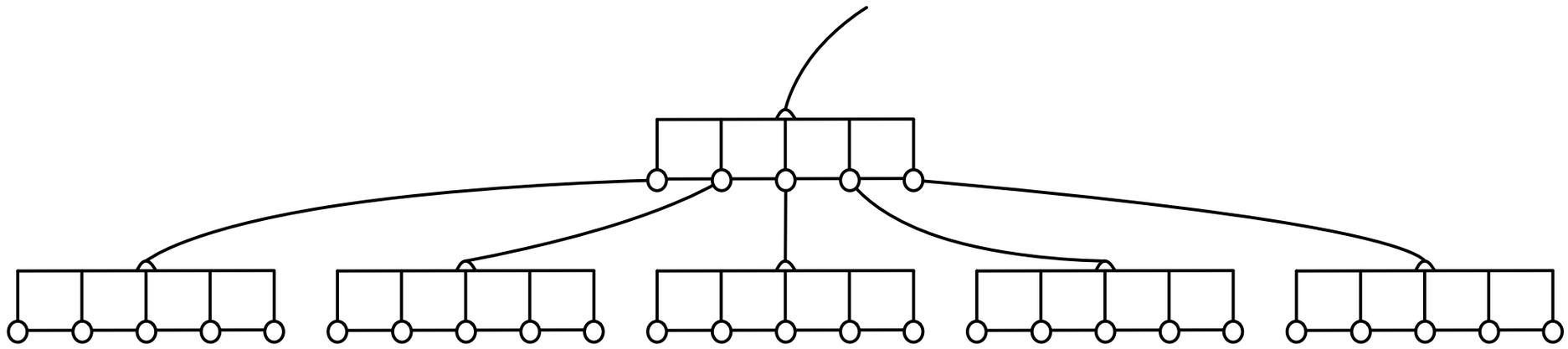
(4) A non-leaf node with k children contains k-1 keys.



Here you see the [10, 20] node has three child nodes and two keys. If it had four child nodes it would need to have three keys. Five child nodes would require four keys.

B-Tree Invariants:

(5) All leaves are at the same level and carry information (the payload).

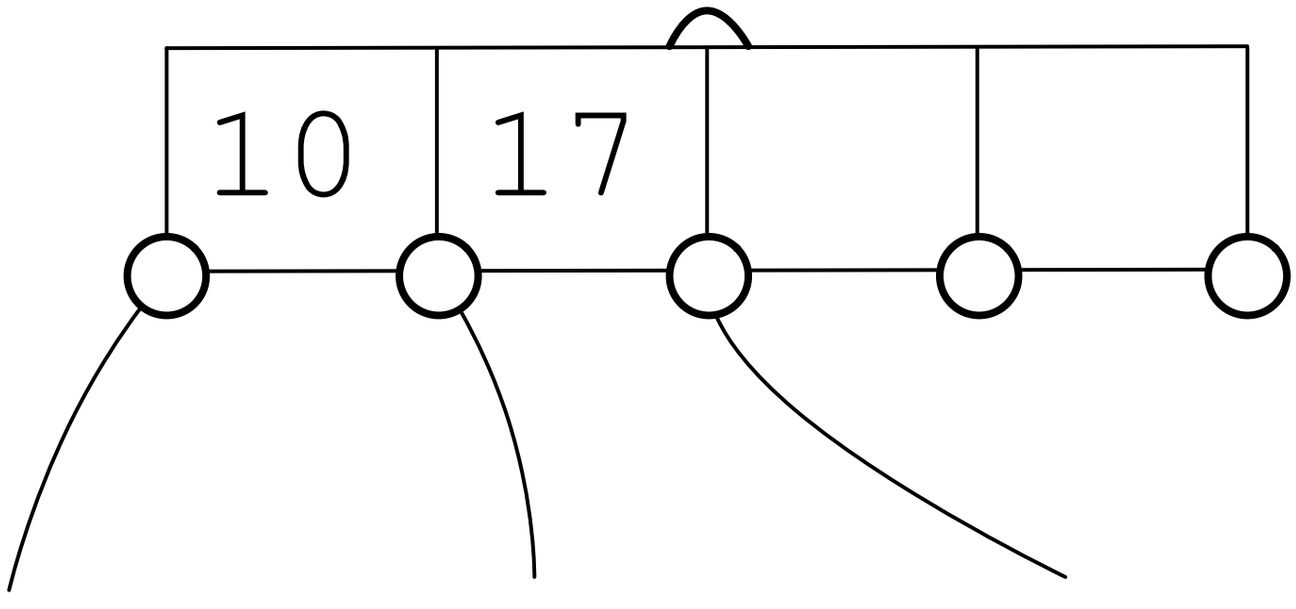


This means the tree is perfectly balanced from a node perspective. Keys may not be evenly distributed (meaning there is wasted space), so long as the nodes support the B-Tree invariant.

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>
```