Algorithm Tutorial: Introduction to the B-Tree Data Structure

By **Jordan Hudgens** - October 5, 2016

With your knowledge of the basic functionality of binary search trees, you're ready to move onto a more practical data structure, the **B-Tree**.

First and foremost, it's important to understand that B-Tree does not stand for Binary Tree or Binary Search Tree.

The `B` in B-Tree technically doesn't represent a word. However some common characteristics can be summarized with words that begin with `B`, which is most likely the origin of the name.

For example, B-Trees are:

- **Balanced** – this is a self balancing data structure, which means that performance can be guaranteed when B-Trees are utilized.
- **Broad** – as opposed to binary search trees, which grow vertically, B-Trees expand horizontally, so saying that they're broad` is an apt description.
- **Bayer** – lastly the creator of B-Trees was named **Bayer Rudolf**. In all actuality this is probably the reason why B-Trees got their name.

Real World Example of B-Trees

If you've made it this far through the algorithm and data structure bootcamp course you may notice a trend. I like to always pair abstract concepts up with real world behavior. In the computer science world this process is called reification. I feel so strongly about this concept that I named my consulting company Reifio.

So what's a real world example of B-Trees? My favorite illustration of how this data structure behaves can be found in the world of shopping. Imagine that you were looking for a pair of new headphones. You have a few approaches.

You could go to every store in the world until you happened to find product. As you can imagine this would be a horrible way to shop. Instead you could go to Best Buy because you know they carry electronics.

Once arriving at Best Buy you could look at each of the aisle descriptions to see where the headphones are displayed. Once you have found the correct aisle you can pick out the headphones that you want.

Notice how the goal of this process is to narrow down the focus of a search? This is how B-Tress work. By organizing data in a specific manner, we can leverage B-Trees so that we don't waste our time looking through data that has a zero percent chance of storing the data that we're looking for.

When are B-Trees Used in Applications?

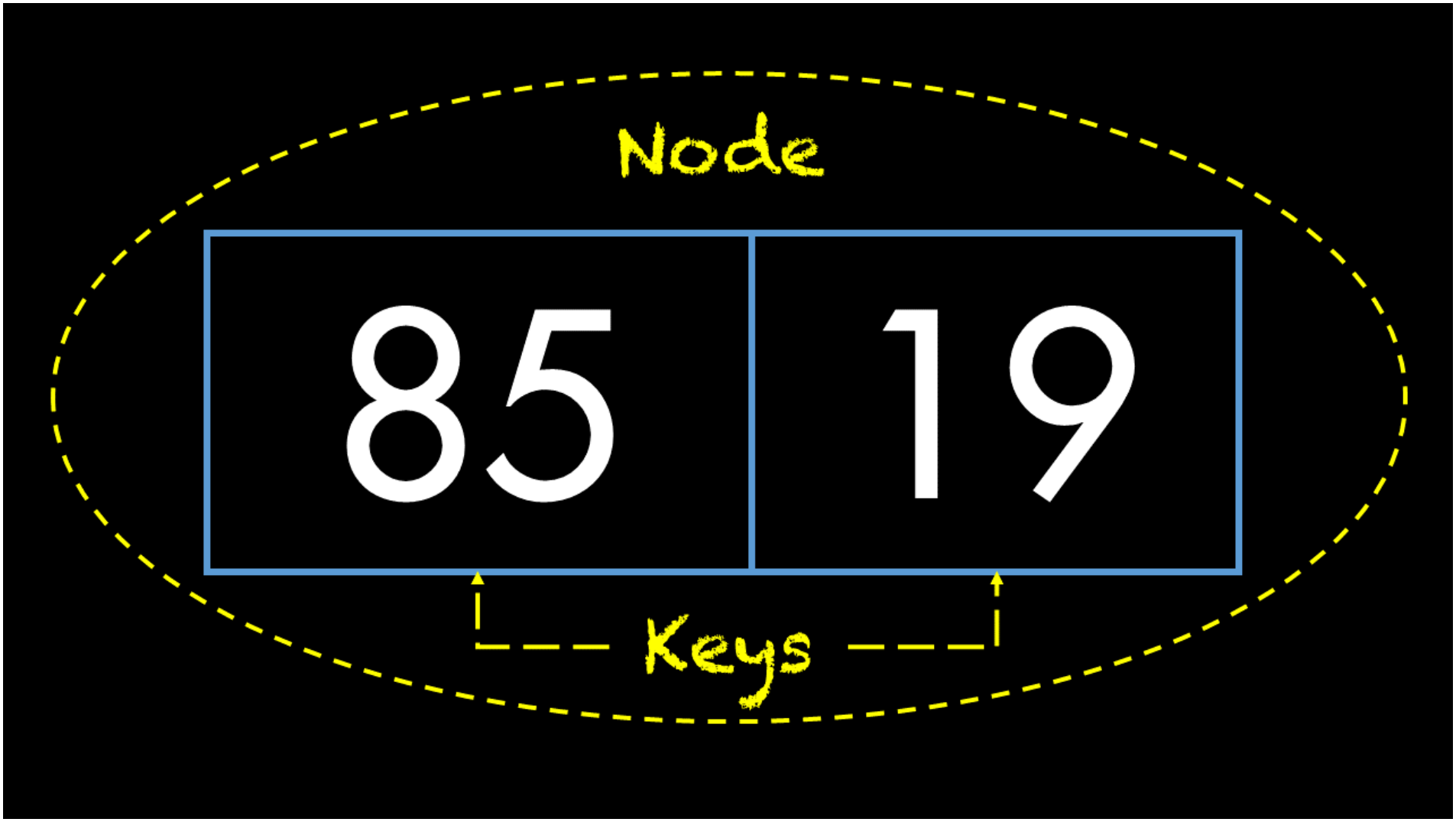
Many data structures, such as binary search trees were created to be stored in memory. However B-Trees have a different purpose. They focus on large amounts of data. The typical usage for B-Trees in applications is for database indices.

If you've never used a database index before, an index is essentially a copy of a database column that allows for fast queries for common searches. For example, if you have a database table that stores users, you can use an index to quickly search for users by their email address.

Since information such as a list of users would be too difficult to store in memory, B-Trees allow for rapid searches on external hard disks, such as in database files.

When it comes to working with B-Trees I like to start with understanding when they are used. Now that you have a good idea for their practical purpose, let's take a deeper dive into the B-Tree structure.

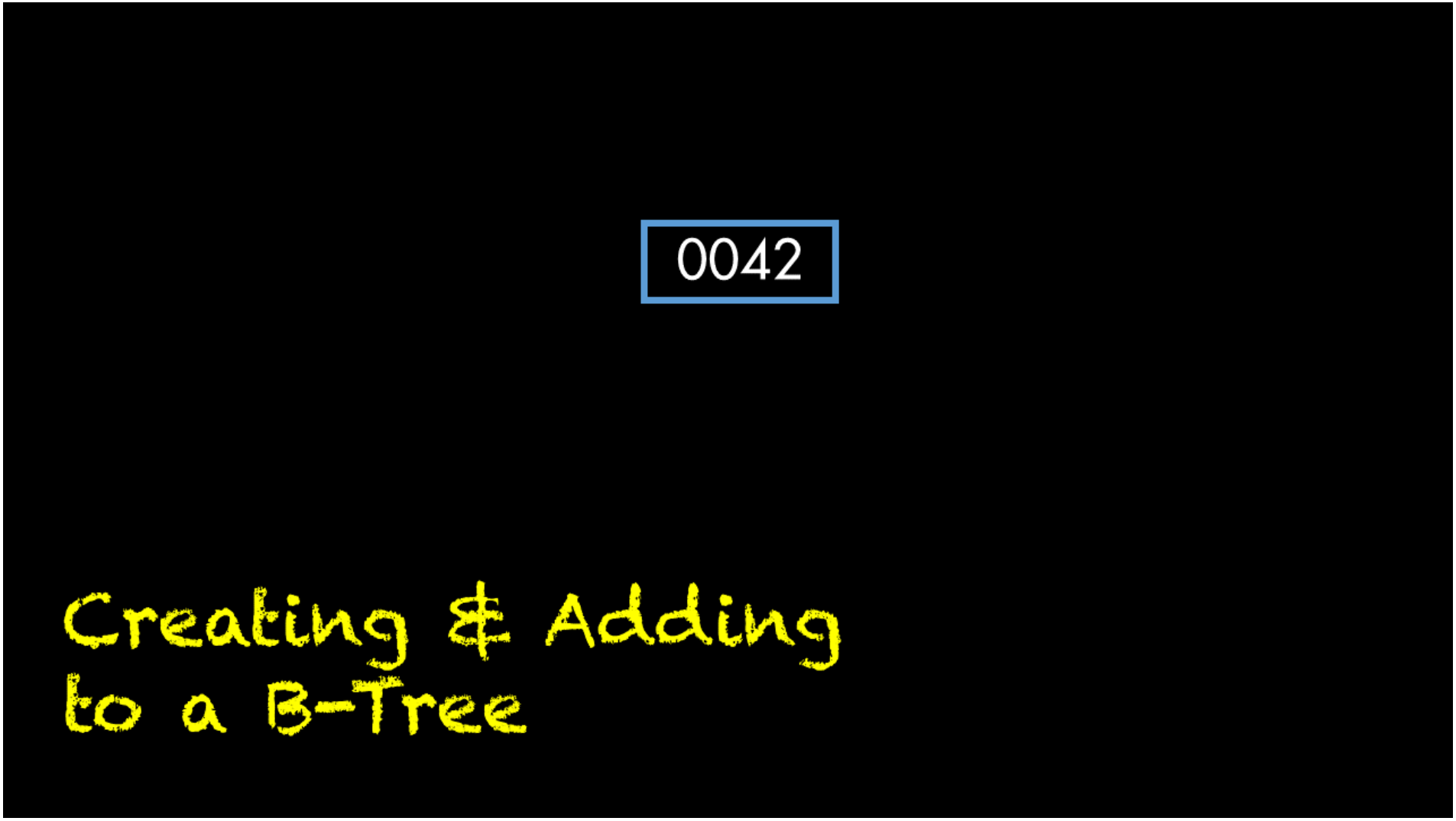
B-Tree Structure



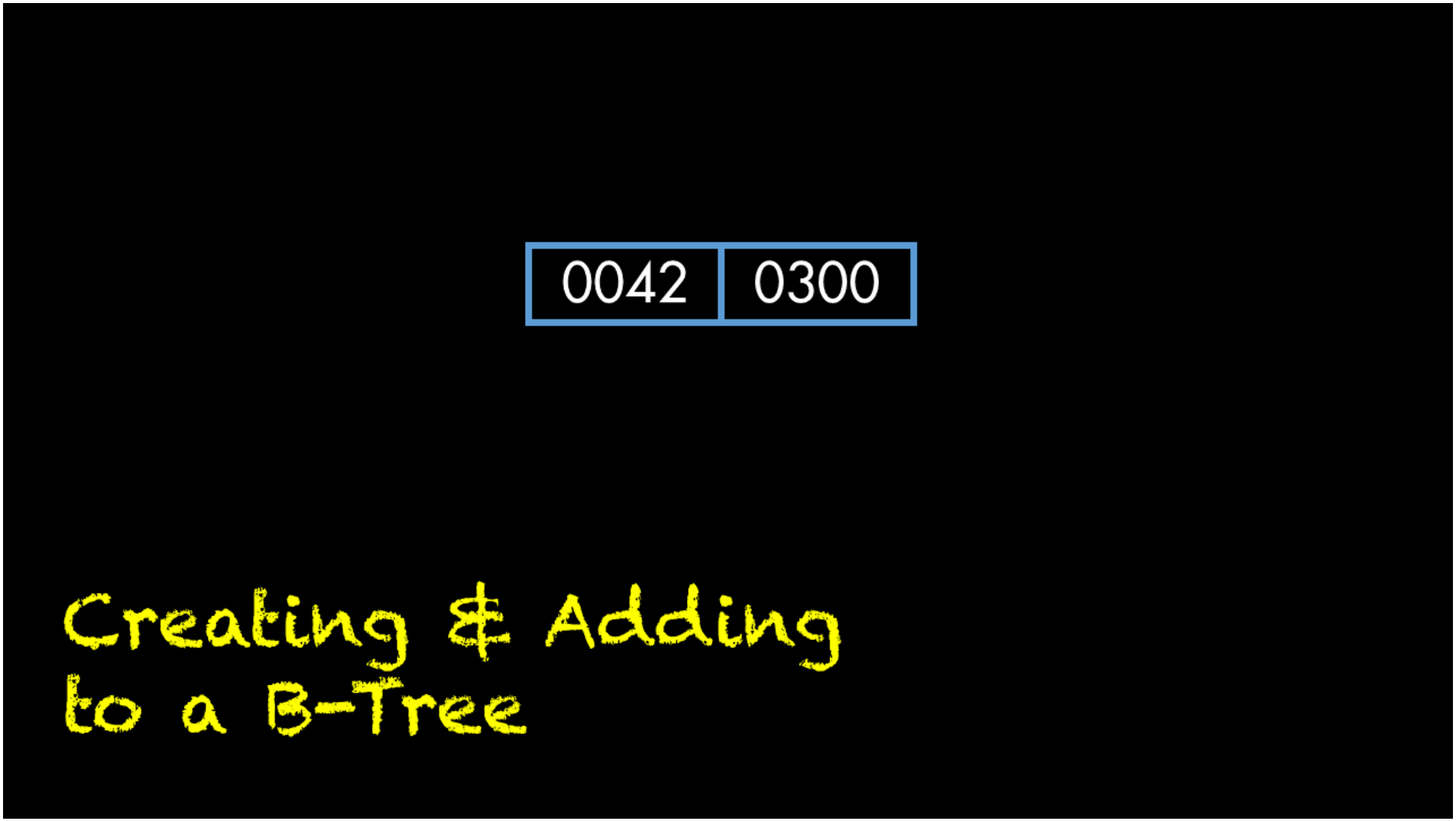
Before we dive into the full structure let's take a look at a single node. B-Tress are setup differently from binary search trees. Instead of nodes storing a single value, B-Tree nodes have the ability to store multiple values, which are called `keys` .

Creating and Adding to a B-Tree

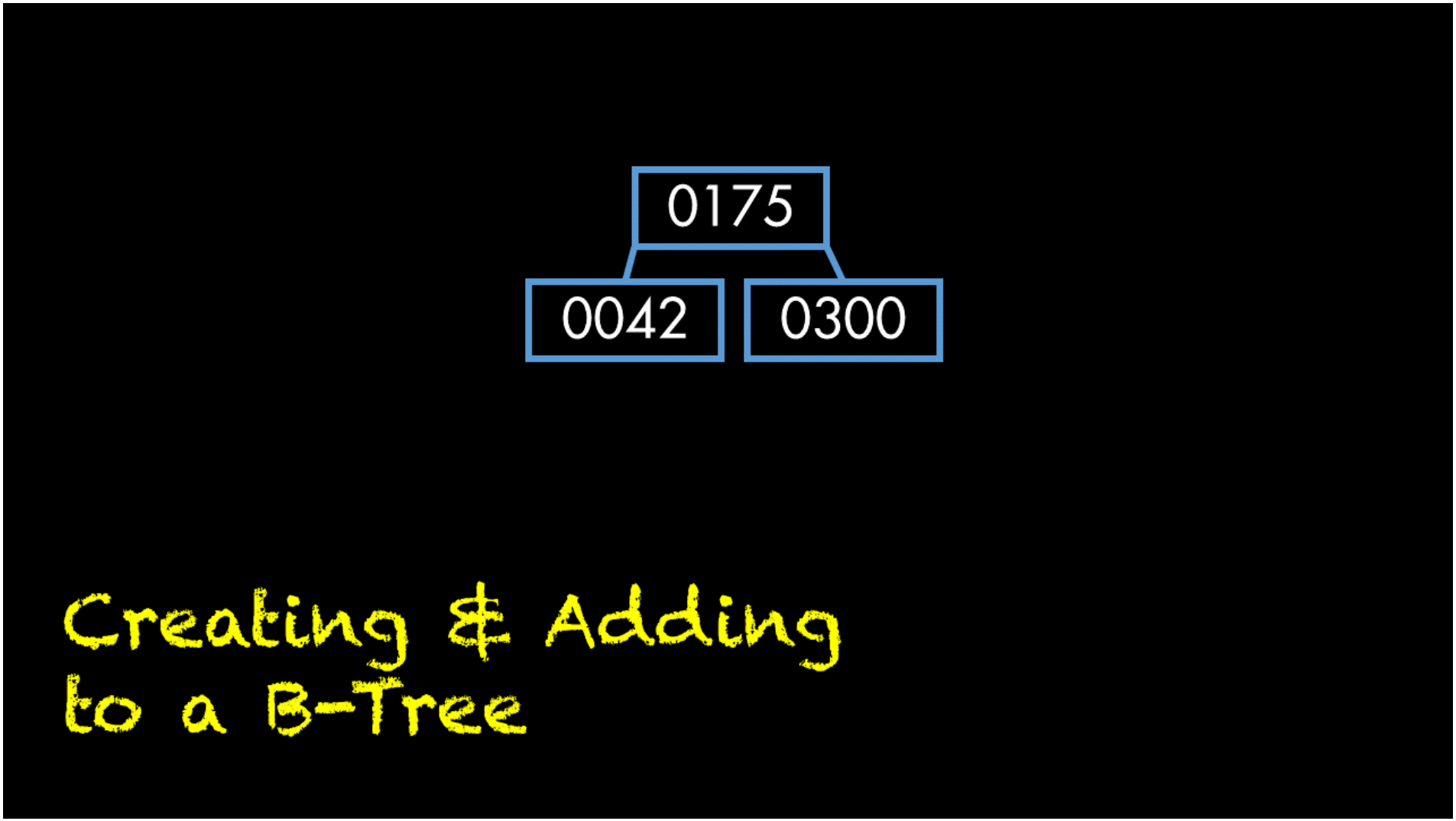
Let's walk through how to create a B-Tree.



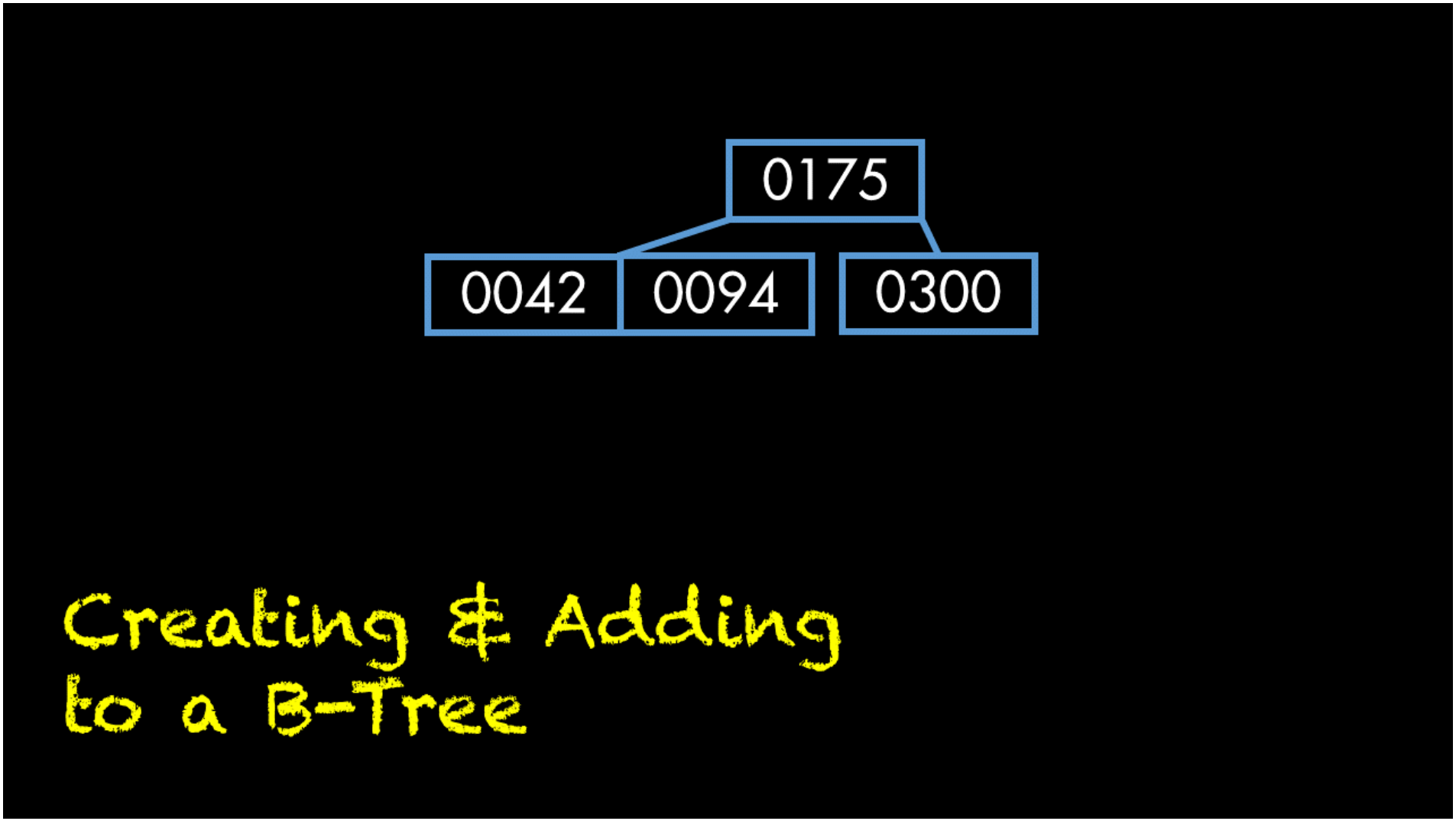
Starting with the integer `0042` our B-Tree will start with a single node which will contain a single key. Make sure to note that this is different from a binary search tree, where the `0042` value would be the node itself.



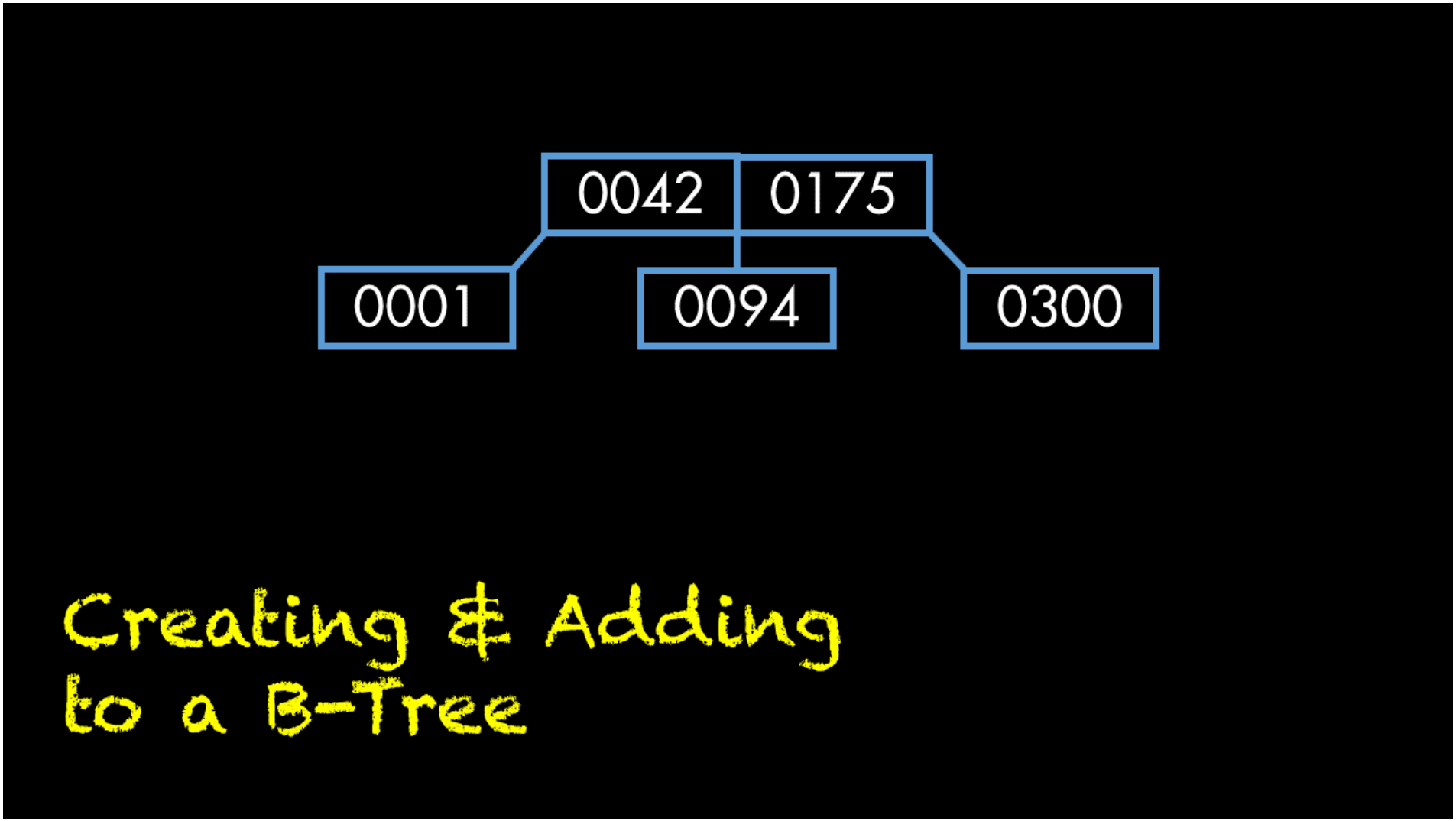
In order to add the value `0300` we will add the key to the same node, so the node will now contain the keys of `0042` and `0300`.


0175
0042
0300
Creating & Adding
to a B-Tree

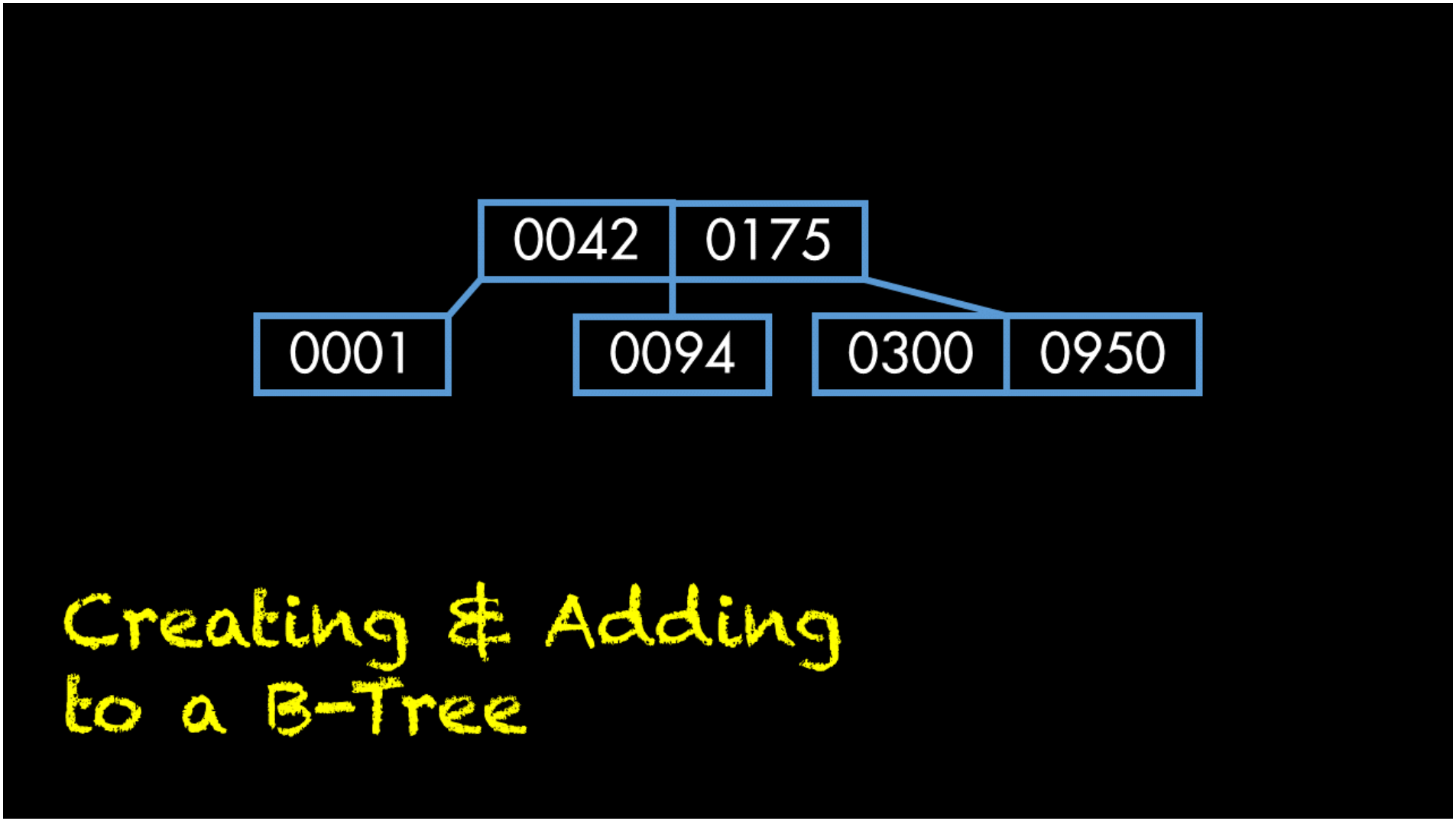
In the next step we see how the self balancing behavior works for B-Trees. In order to add `0175` to the tree we will split the other two values into their own nodes and since `0175` is between the two values we will make it the parent of the other two values.



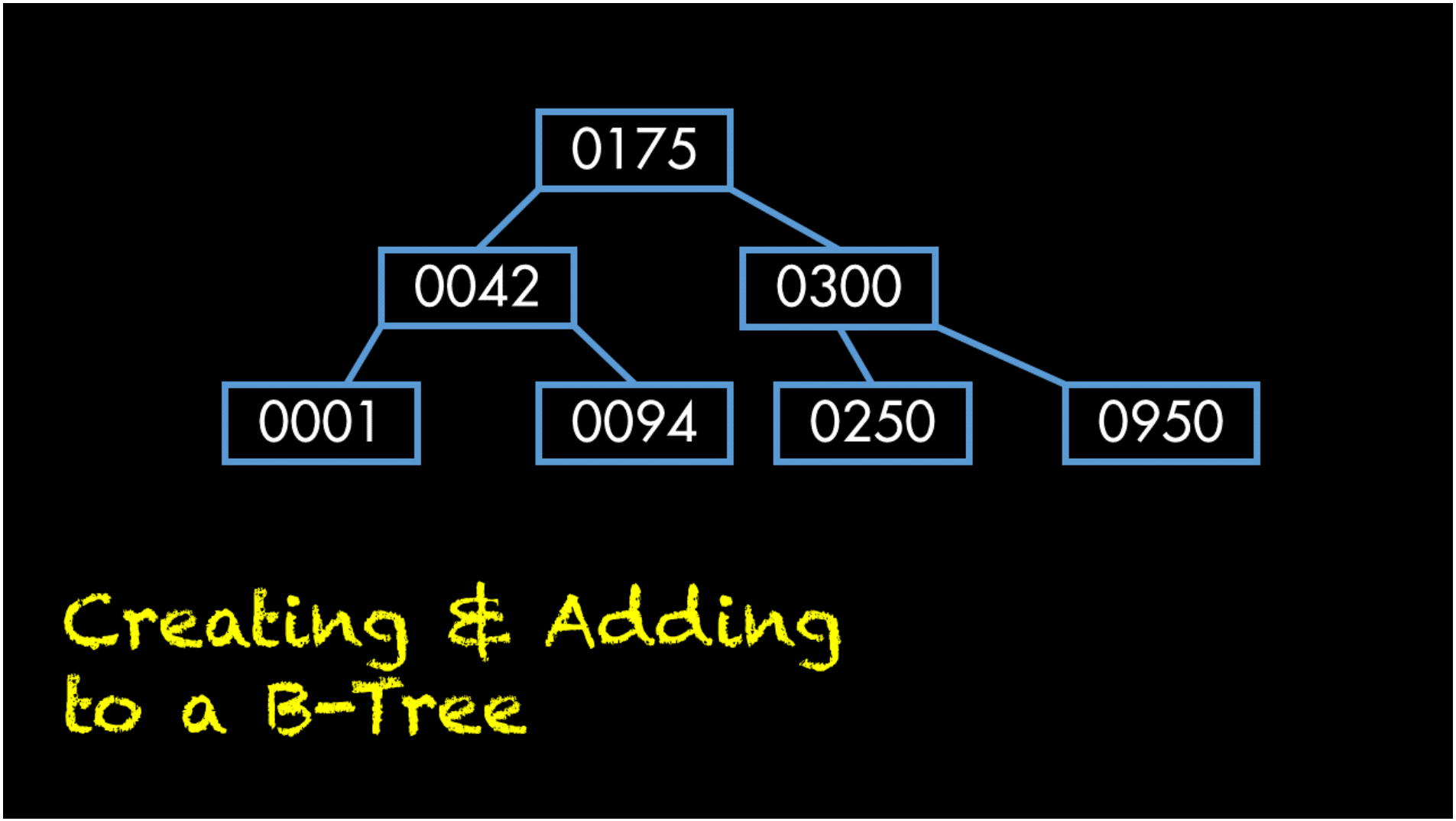
If we want to add the value `0094` to the tree we can insert it as a key inside of the same node that contains the `0042` key. This keeps the integrity of the tree intact.



Now if we want to add the value `0001` we will move `0042` up so it will share a node with the `0175` key. And the new parent node will have three distinct child nodes. Notice how this differs from the behavior of binary search trees. B-Trees are not limited by the two child rule.



If we want to add the large value of `0950` we can insert it at the end of the tree, so it will share a node with the `0300` key.



Lastly, but very importantly, if we want to add the value `0250` to the tree we will have to change the entire structure of the tree. This new value will force us to add a new level to the tree. The `0250` will be a child of the `0300` node since it's less than `0250`. Notice in the illustration how we move the `0175` up to a new level of the tree. This allows us to maintain tree balance while still being able to add new values to the tree.

Searching through a B-Trees

Searching through a B-Tree is very similar to searching through a binary tree. The key difference comes when you run into a node that has multiple keys. In cases like this you will traverse the tree and when you find a node that should contain the value you will look at both of the keys to see if they equal the

value that you're searching for.

Jordan Hudgens

http://devcamp.com