Kademlia Routing Tree

Christoffer Fink

September 25, 2025

Introduction

Let's explain how the general Kademlia routing tree works (for arbitrary $b \neq 1$). It's not actually that complicated, but the Kademlia paper doesn't explain it very well. I will fill in some details, and then it should not be that hard.

We only need to look at a few paragraphs of the paper to get the information we need. So I will quote them and then make some comments that hopefully fill in the necessary detail and clear everything up.

One more thing before we start. Remember: we can take two equivalent perspectives when we imagine the routing table/tree. We can assume that the node has ID $0 \cdots 0$, and then we can look at each bit of some other ID (a node we want to add or some target ID/key that we want to look up) and go left for 1 and right for 0. Or we assume that the node has some arbitrary ID u, and rather than of looking directly at the other ID's bits, we instead look at it's XOR with u. Assuming $u = 0 \cdots 0$ is a bit easier.

Section 2.4

Kademlia's basic routing table structure is fairly straight-forward given the protocol, though a slight subtlety is needed to handle highly unbalanced trees. The routing table is a binary tree whose leaves are k-buckets. Each k-bucket contains nodes with some common prefix of their IDs. The prefix is the k-bucket's position in the binary tree. Thus, each k-bucket covers some range of the ID space, and together the k-buckets cover the entire 160*-bit ID space with no overlap.*

This is pretty straightforward. Just to make sure everything is clear, these might be worth elaborating on:

- *"Each k-bucket contains nodes with some common prefix of their IDs."*
  - This just means that all of them must share a prefix of some minimum length. The first few bits of each ID are the same. **That's why they are in that particular bucket!**
- *"The prefix is the k-bucket's position in the binary tree."*
  - We find the bucket by going left/right depending on whether each bit matches or not. So we find the bucket (it's position) in the tree because it represents a certain range and all IDs it contains have a certain prefix. The prefix defines a series of left/right directions and takes us to the bucket. So it encodes its position in the tree.

Nodes in the routing tree are allocated dynamically, as needed. Figure 4 illustrates the process. Initially, a node u's routing tree has a single node – one k-bucket covering the entire ID space. When u learns of a new contact, it attempts to insert the contact in the appropriate k-bucket. If that bucket is not full, the new contact is simply inserted. Otherwise, if the k-bucket's range includes u's own node ID, then the bucket is split into two new buckets, the old contents divided between the two, and the insertion attempt repeated. If a k-bucket with a different range is full, the new contact is simply dropped.

- *"Nodes in the routing tree are allocated dynamically, as needed."*
  - This already implies splitting. We don't start (in the general case) with a list of buckets; we add new nodes to the tree as buckets (leaf nodes) are split.
- *"Initially, a node u's routing tree has a single node – one k-bucket covering the entire ID space."*
  - As long as there is space, all IDs go in this single bucket, until we have to split it.
- *"Otherwise, if the k-bucket's range includes u's own node ID, then the bucket is split into two new buckets, the old contents divided between the two, and the insertion attempt repeated."*
  - A full bucket is where it gets interesting. Do we split it?
  - **Important:** We can **always** split a bucket if it covers a range such that it includes the node's own ID. In other words, the node's ID would go into that bucket. In other other words, we can always split a "right" child. (Because $0 \cdots 0$ is all the way to the right.)
  - Note that, when $b = 1$, we're allowed to split **only** when the bucket contains u.
- *"If a k-bucket with a different range is full, the new contact is simply dropped."*
  - This is only true for $b = 1$!
  - That's why we end up with a linked list when we have $b = 1$.
  - With $b > 1$, we can sometimes also split on the left (when a bucket *does not* include the node's ID).

One complication arises in highly unbalanced trees. Suppose node u joins the system and is the only node whose ID begins 000*. Suppose further that the system already has more than k nodes with prefix* 001*. Every node with prefix* 001 *would have an empty k-bucket into which u should be inserted, yet u's bucket refresh would only notify k of the nodes. To avoid this problem, Kademlia nodes keep all valid contacts in a subtree of size at least k nodes, even if this requires splitting buckets in which the node's own ID does not reside. Figure 5 illustrates these additional splits. When u refreshes the split buckets, all nodes with prefix* 001 *will learn about it.*

This is where it gets a bit weird. The explanation of the problem is OK, but the solution is not clear.

- *"To avoid this problem, Kademlia nodes keep all valid contacts in a subtree of size at least k nodes, even if this requires splitting buckets in which the node's own ID does not reside."*
  - Translation: Don't throw away contacts just because we would need to split a bucket that does not contain u.

In other words, splitting on the left is sometimes allowed. When? Well, it's not actually explained until section 4.2 (*Accelerated Lookups*).

Section 4.2

Another optimization in the implementation is to achieve fewer hops per lookup by increasing the routing table size. Conceptually, this is done by considering IDs b bits at a time instead of just one bit at a time. As previously described, the expected number of hops per lookup is $\log_2 n$. By increasing the routing table's size to an expected $2^b \log_{2^b} n$ k-buckets, we can reduce the number of expected hops to $\log_{2^b} n$.

The next paragraph is where it gets interesting. This is the main paragraph that has the information we want.

Section 2.4 describes how a Kademlia node splits a k-bucket when the bucket is full and its range includes the node's own ID. The implementation, however, also splits ranges not containing the node's ID, up to $b-1$ levels. If $b=2$, for instance, the half of the ID space not containing the node's ID gets split once (into two ranges); if $b=3$, it gets split at two levels into a maximum of four ranges, etc. The general splitting rule is that a node splits a full k-bucket if the bucket's range contains the node's own ID or the depth d of the k-bucket in the routing tree satisfies $d \not\equiv 0 \pmod{b}$. (The depth is just the length of the prefix shared by all nodes in the k-bucket's range.) The current implementation uses $b=5$.

- *"The implementation, however, also splits ranges not containing the node's ID, up to $b-1$ levels."*
  - The earlier explanation was simplified. We can also split buckets that don't contain u.
  - This extra splitting is limited by the b parameter.
- *"The general splitting rule is that a node splits a full k-bucket if the bucket's range contains the node's own ID or the depth d of the k-bucket in the routing tree satisfies $d \not\equiv 0 \pmod{b}$."*
  - So here we have the additional rule.
  - We can always split a bucket that contains u (can always "split right").
  - Otherwise, we can still split depending on the value of b and the bucket's depth d.
  - Note that $d\%1 = 0$ for all d, because everything is evenly divisible by 1.
  - So, when $b=1$, $d\%b \neq 0$ is never true and the extra rule is never satisfied, and we can never "split left".
  - $d\%2 \neq 0$ is true for every other (every odd) depth.
  - However, as soon as we decide not to split, we don't get another chance to split that interval.
  - The tree is pruned and the path to the left is blocked.
  - That's why an interval that does not contain u only splits at $b-1$ depths (not every second or third depth for example).
- *"The depth is just the length of the prefix shared by all nodes in the k-bucket's range."*
  - This does **not** mean you look at the IDs that are in the bucket and compare them to see what the longest prefix is that they all share.
  - It's about the length of the longest prefix they **must** share so that they belong in that bucket.
  - So it has to do with the range of the bucket, not what IDs happen to be in the bucket.
  - The bucket has a certain range, which means all IDs that belong there must start with a certain prefix, and the rest of the ID could be anything.
  - In particular, the smallest and the largest ID in the range must share this prefix.
  - So you can compute the depth by just checking the common prefix of the lower and upper limit of the bucket's range! (See also the illustration in figure 1 and 2.)

  - You could also keep track of the depth explicitly, but that might actually be more difficult than computing it from the range.

Summary

- Start with a single bucket that covers the whole ID range.
- You split a full bucket when either
  - it contains the node's own ID (it's a "right" bucket), or
  - its depth d satisfies $d \not\equiv 0 \pmod b$, i.e. $d\%b \neq 0$.
- A bucket's depth d is the number of leading bits that the lower and upper limits of the bucket's range share. (Or keep track of it explicitly. Not recommended.)
- A bucket splits into two buckets that each cover half of the original bucket's ID range, and the contacts are put into the new buckets depending on which one they belong to.
- When you can't split a bucket, you have to choose whether to discard the new contact or remove one from the bucket to make room. Just like the simplified $b = 1$ case.

Illustrations

The illustrations assume IDs have 6 bits.

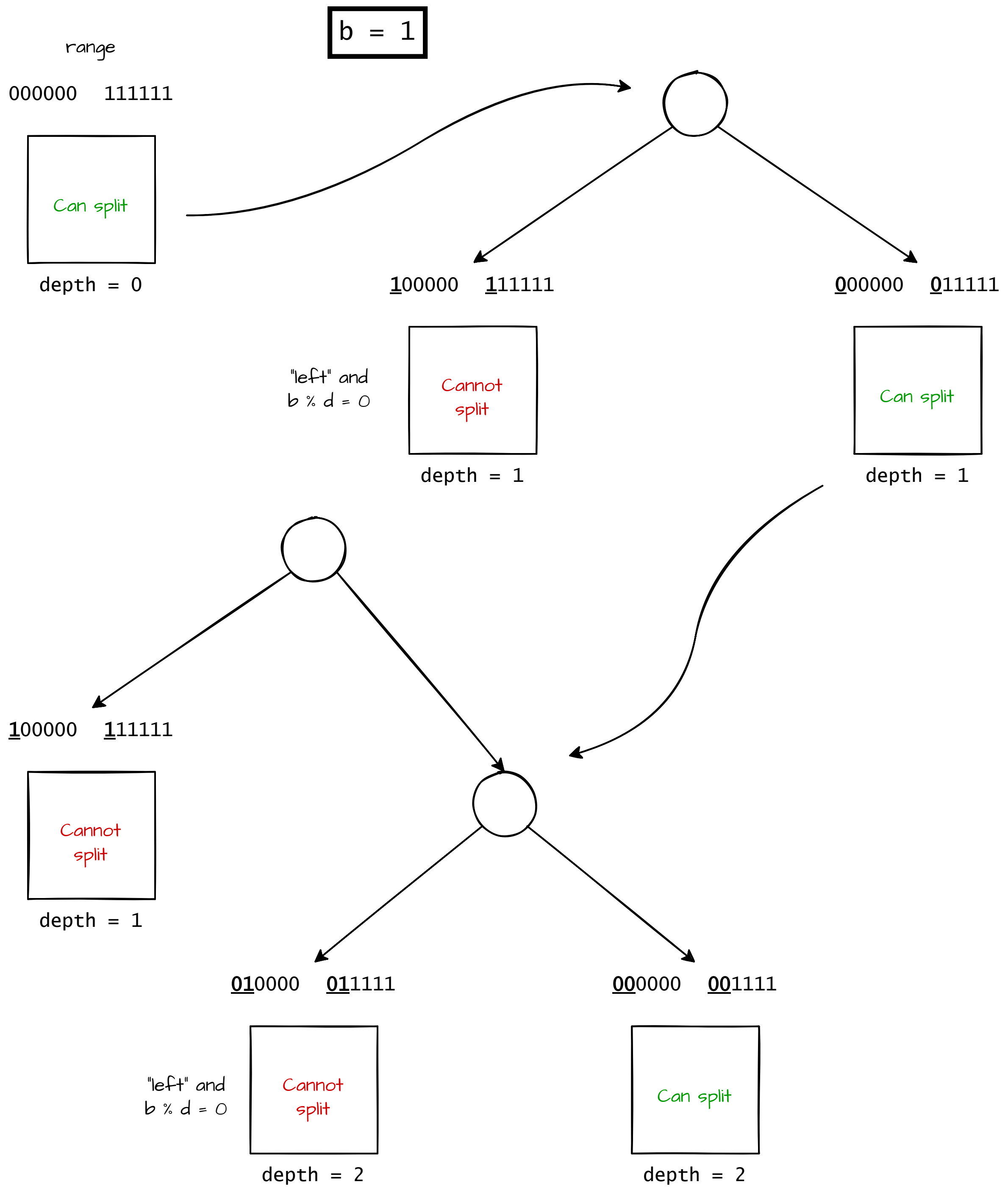

Figure 1: Splitting 3 times with $b = 1$. Notice the bold and underlined leading bits in the bucket ranges. That's the common prefix of every ID in the bucket. The number of bold bits is the depth.

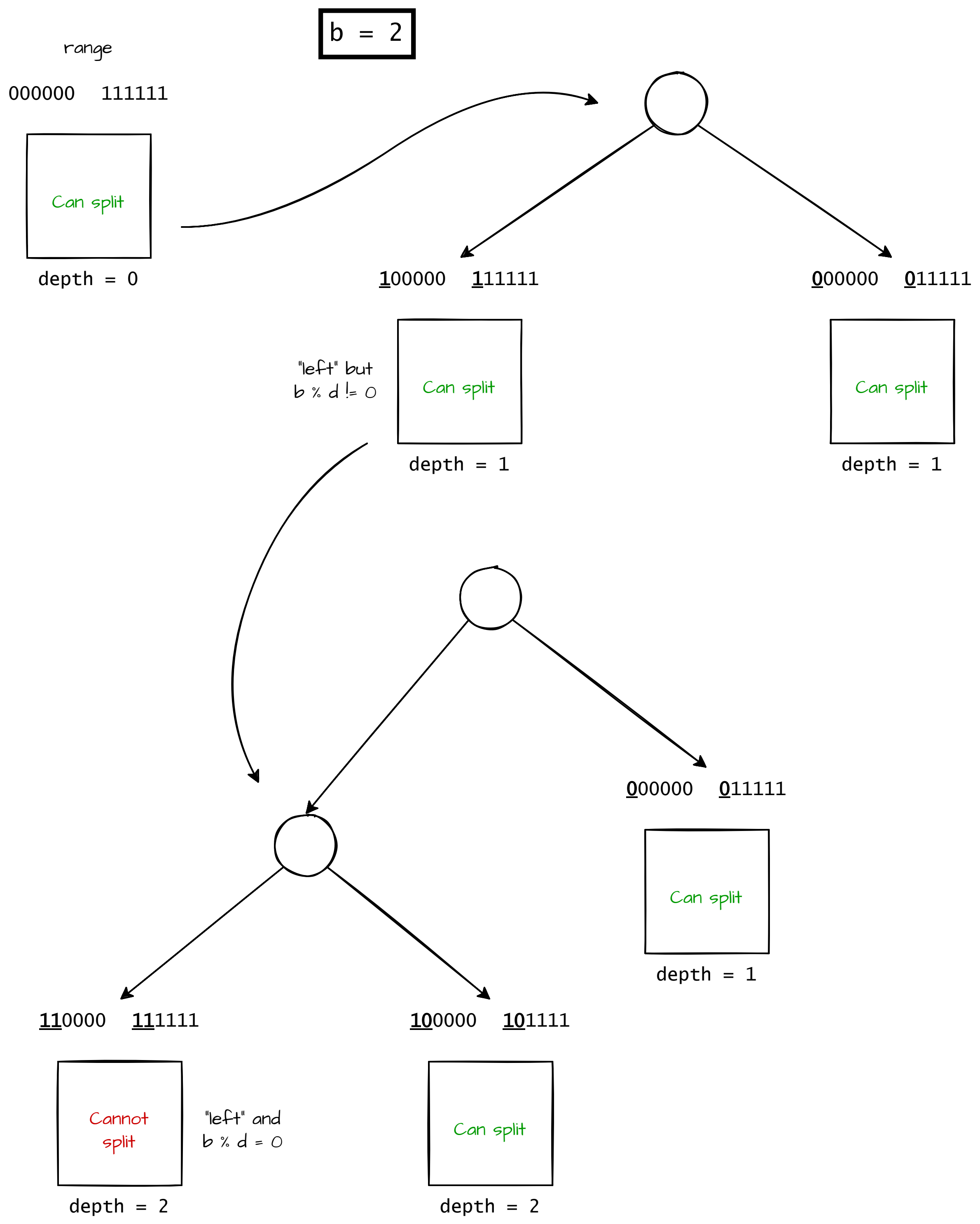

Figure 2: Splitting 3 times with $b = 2$.

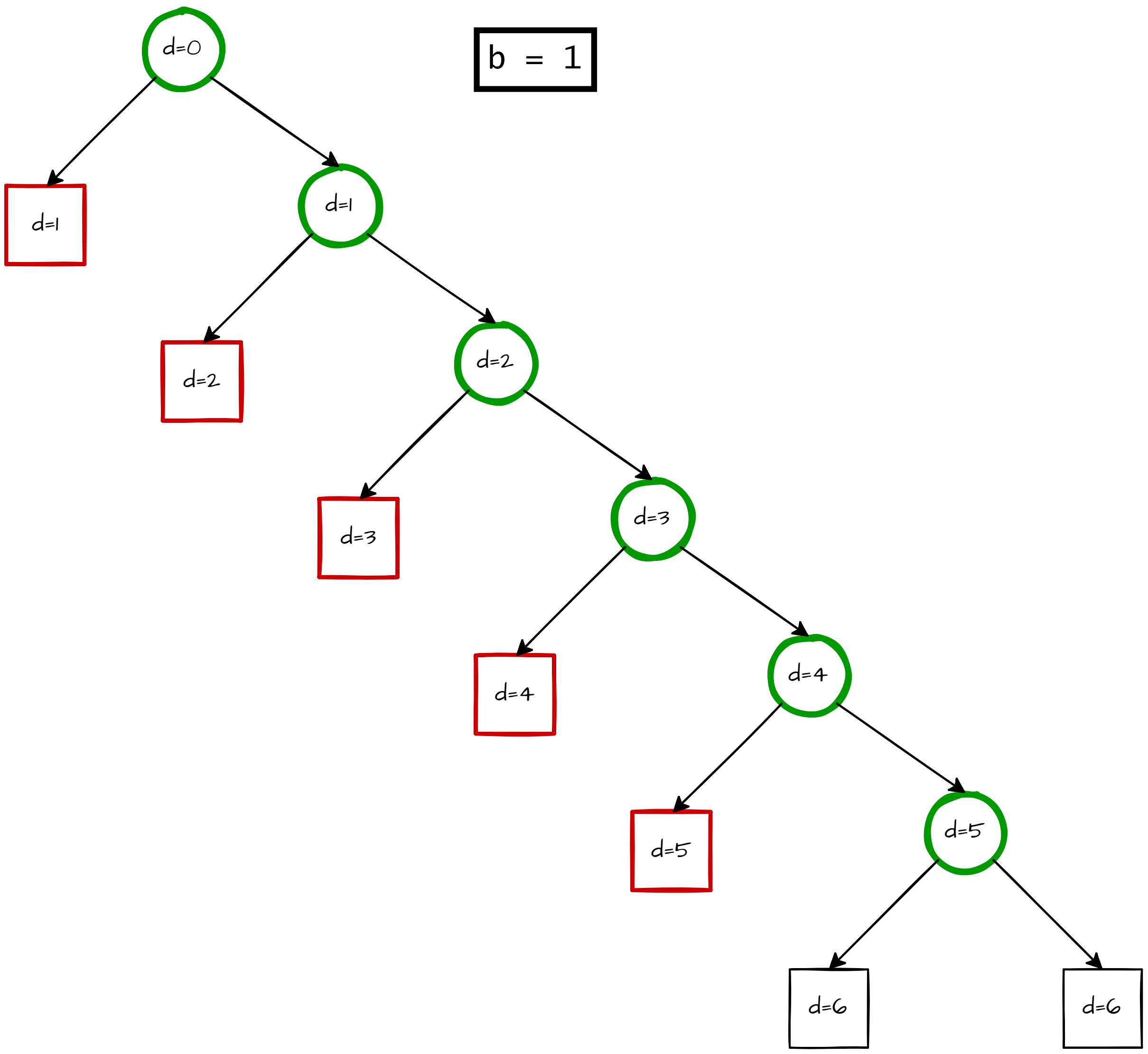

Figure 3: What happens if we keep splitting when $b = 1$.

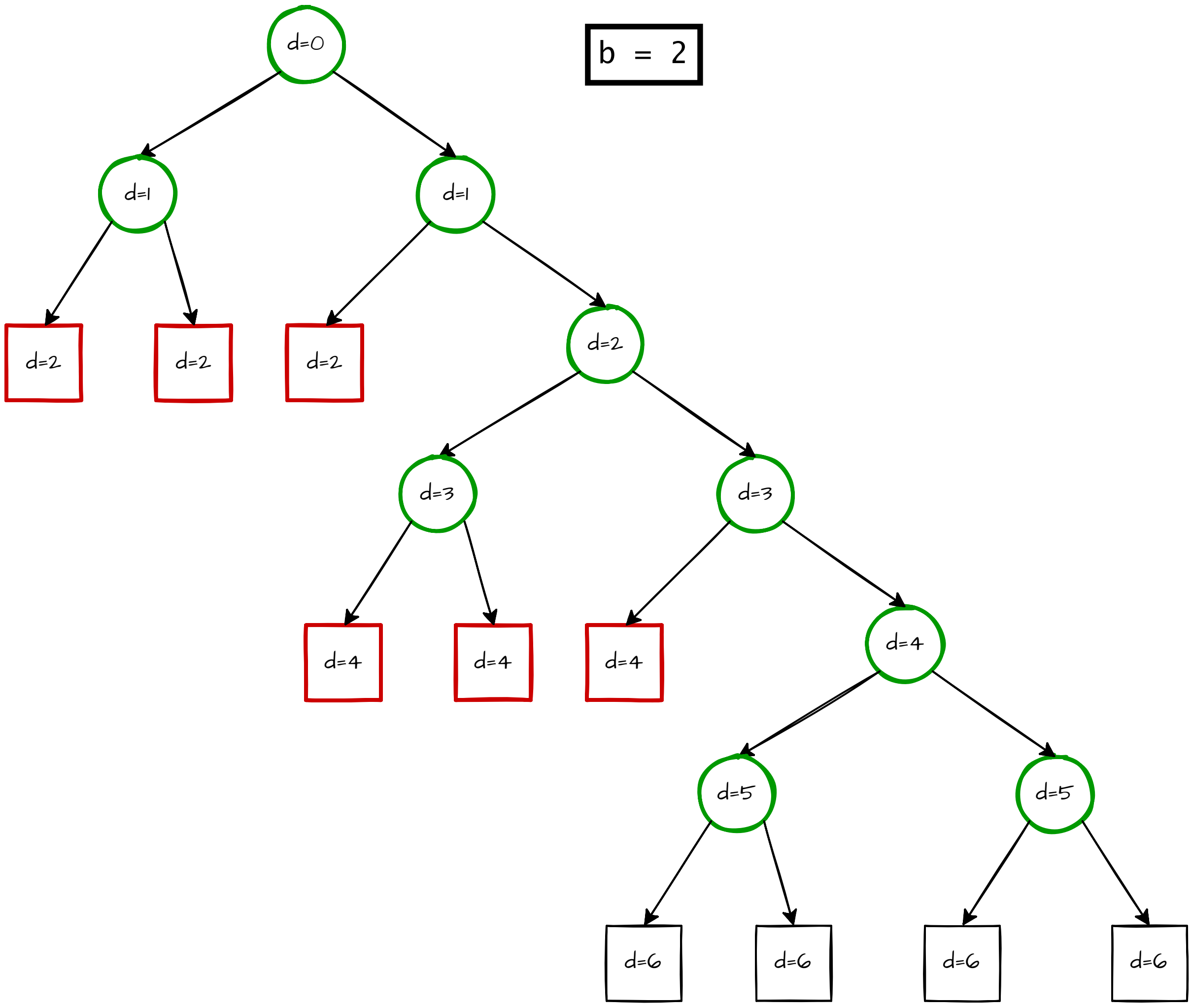

Figure 4: What happens if we keep splitting when $b = 2$.

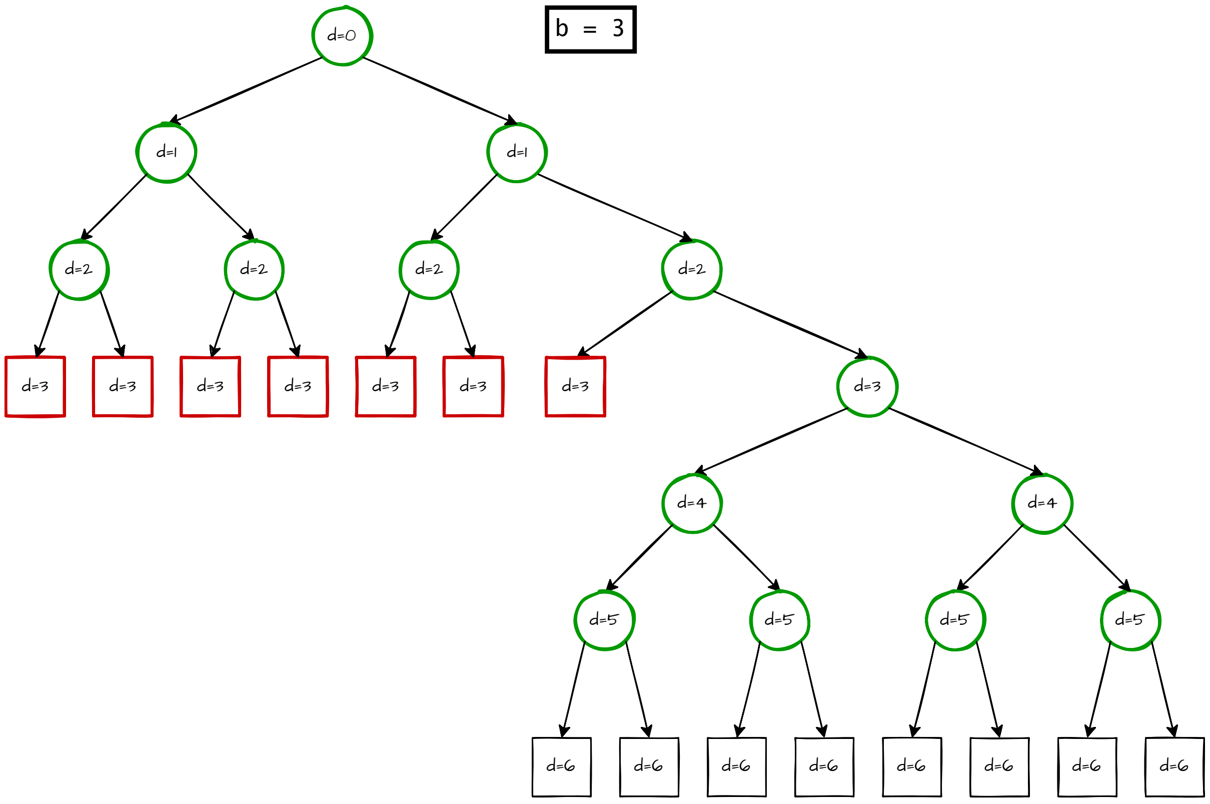

Figure 5: What happens if we keep splitting when $b = 3$.