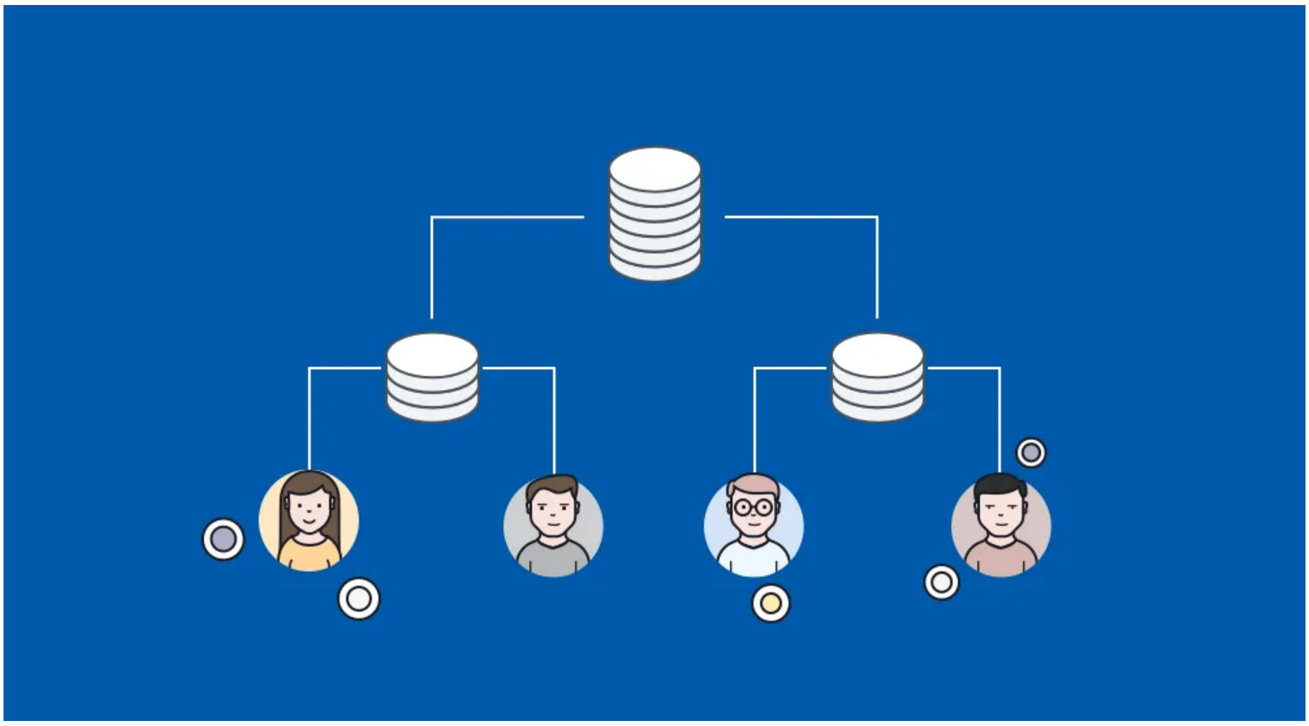


An Efficient Data Structure for Blockchain Sortition

Keeping Track of Dynamic Stakes in a Lottery Like Game Can Be Costly in Terms of Computation, If Not Done Right. Let Us Look at a Suitable Approach…

Enrique Piqueras · Follow
Published in Kleros
5 min read · Aug 28, 2018

Drawing parties from a set of token holders using random numbers with a likelihood proportional to the amount of tokens they hold is very common in decentralized applications. This is very simple if you only allow adding stake. But, what if you want to make the staked amount freely editable throughout drawing rounds? This is where things get a bit more complicated…

The Naive Approach

The simplest way to implement a system like this would be to create a virtual list that holds stake segments for every address that is participating. You could have a storage variable that keeps track of the list size and every time someone stakes tokens you do:

```
tokenHolder.segmentStart = segmentSize;
segmentSize += _stake;
tokenHolder.segmentEnd = segmentSize;
```

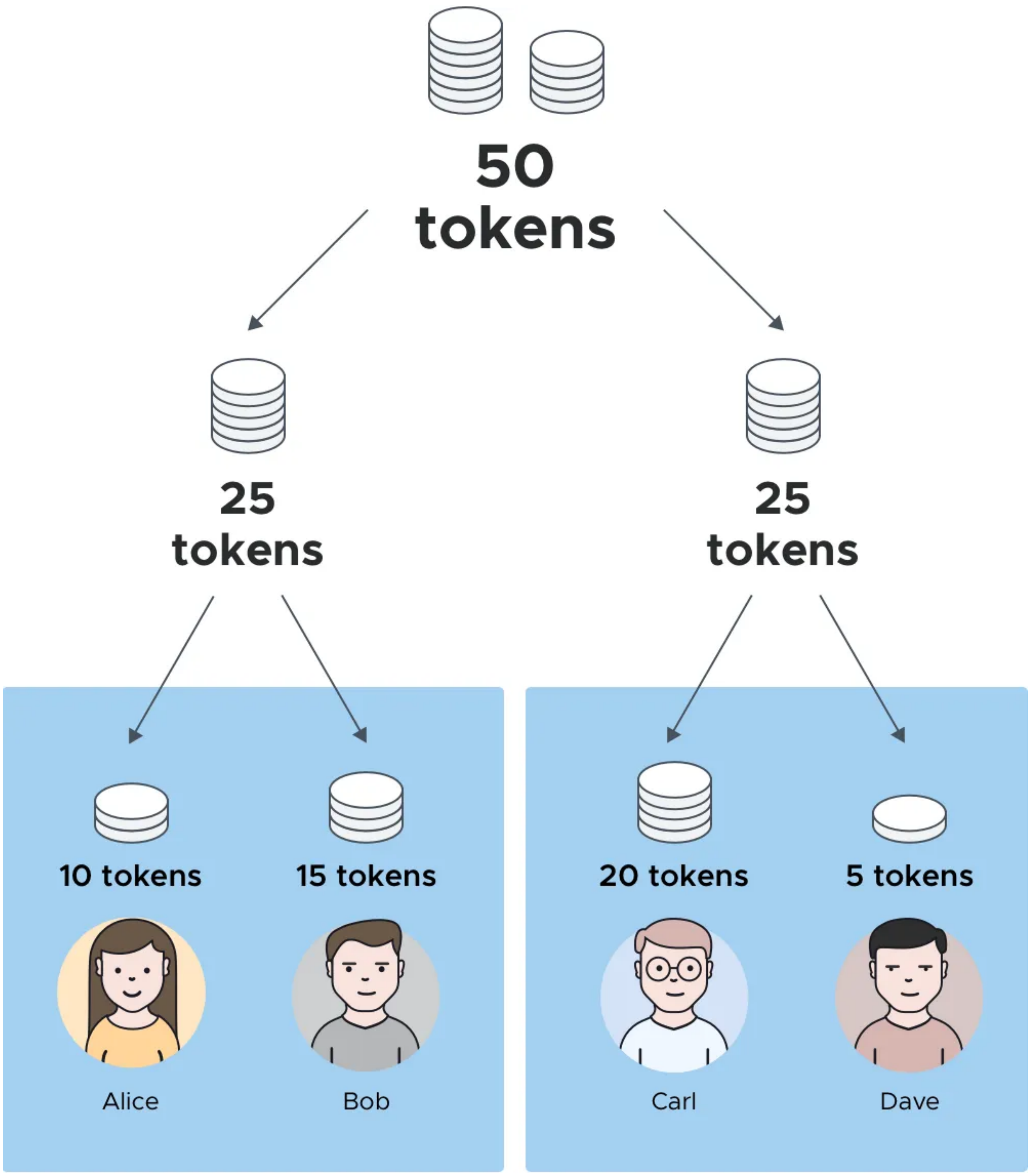
There are two big problems with this approach (although, depending on your use case, they might not be too bad). First, if a token holder wishes to change or remove their stake, you are out of luck. Splicing this list would be very costly. Second, figuring out who is drawn would require you to iterate over every token holder like this:

```
// O(n) where n is the number of token holders with stake
for (uint i = 0; i < tokenHolders.length; i++)
    if (currentDrawnNumber < tokenHolders[i].segmentEnd)
        return tokenHolders[i].address
```

In a system of any reasonable size, this would be very costly to do for every draw. You also open up a `O(d * a)` griefing factor, where `d` is the number of draws and `a` is the number of token holders, because attackers can create lots of accounts with minimal stakes. This means that the only reasonable approach is to have token holders check for themselves if they are drawn or not. This severely limits the logic you can implement in the smart contract that revolves around who is drawn and makes notifications very hard to implement in a decentralized fashion.

Sum Trees to the Rescue

In search of a better approach, our CTO, Clément Lesaege, found the answer in the shape of a more sophisticated data structure: the K-Ary Sum Tree. K-Ary Sum Trees are trees where every non-leaf node has `K` children and holds the sum of the values of all of them. This means that the actual values we care about are all leaf nodes, but organizing them in this way proves to be very useful for performance. Here is what a binary sum tree (`K=2`) where Alice has 10 tokens staked, Bob has 15 tokens staked, Carl has 20 tokens staked, and Dave has 5 tokens staked would look like:



An example of a binary sum tree.

Now, say we draw the number 13. You start from the root and look at its children:

where in the range of `[0-24, 25-49]` does 13 fall into? `0-24`, so you go left. You do the same for `[0-9, 10-24]` and go right into a leaf node. You draw Bob.

Now, say we draw the number 27. Again, you start from the root, but this time 27 falls into `25-49`, so you go right. Now, you repeat the same process for `[0-20, 21-24]`, but with 2 instead of 27, because you "spent" 25 by skipping the left node in the first level. You draw Carl.

So this data structure lets us draw with `O(K * log(n)/log(K))` complexity, where `K` is the number of children per node. `log(n)/log(K)` is the number you have to raise `K` by to equal the number of leaf nodes in a balanced tree, in other words, the number of levels in the tree. You multiply that by `K`, because in the worst case, you have to look at every child of every node you traverse to figure out which one to go to next.

Sets, removes, and appends have `O(log(n)/log(K))` complexity, because you start at a given leaf node and work your way up all the levels to update its parents. Note that we keep a stack of vacant nodes as a cheap way of allowing removals while still keeping the tree balanced. These operations are very easy to implement:

- **Set:** Set the new value and update parents.
- **Remove:** Set the value to 0, push the node to a stack of vacant nodes, and update parents. Note that you are not actually removing the node from the data structure, hence `n` never decreases.
- **Append:** Pop a vacant node from the previously mentioned stack if there are any, otherwise append a new node, and set its value.

Cutting Costs in the Implementation

Because nodes in the tree never actually get removed (they are just set to 0 and pushed to a stack) we can cut a pretty big corner when it comes to implementing this. We just need a list for the vacant node stack and a list for the actual values in the tree. For the above tree, the list would look like this: `[50, 25, 25, 10, 15, 20, 5]`. Node traversal can be achieved easily with the following equations:

- `C-th` child of node `i`: `K*i + c`. The rationale is that you have to skip every child node of every node before `i`. So `K*i`, and then add `c` to get the child of `i` that you want.

- Parent of node `i`: `(i-1)/K`. The rationale is, how many nodes with `K` children each did you have to skip to get to `i`.
- Check if node `i` is a first/left-most child: `(i-1)%K === 0`. The rationale is similar to the one for finding the parent, only this time you check if the remainder is 0 which means `i` is a first/left-most child.

Our Implementation

You can find our own open source implementation of a k-ary sum tree factory (you can pick how many children to have per node) in the main Kleros smart contract repo's data structures folder: https://github.com/kleros/kleros/tree/master/contracts/data-structures. We also made a contract that inherits that functionality to create sortition sum trees with draw functionality that is ready to be used in any contract that requires sortition.

Join Kleros!

Join the community chat on ***Telegram***.

Visit our ***website***.

Follow us on ***Twitter***.

Join our ***Slack*** *for developer conversations.*

Contribute on ***Github***.

Blockchain
Ethereum
Smart Contracts
Computer Science
Algorithms