There is a good bit of work going on to get ethereum to scale[https://ethresear.ch/c/sharding]. Some of these ideas include something called stateless transactions[https://ethresear.ch/search?q=stateless]. This idea reduces the amount of data that a miner has to store by asking anyone submitting a transaction to include a merkle proof of all the data that the transaction touches. Clients can calculate this data by locally running the transaction and building up the list of variables that they need to provide proof for. Merkle proofs allow you to prove that a value is part of a set.

You can read more about merkle proofs here[https://en.wikipedia.org/wiki/Merkle_tree] and here[https://hackernoon.com/merkle-trees-181cb4bc30b4] and here[https://medium.com/@evankozliner/merkle-tree-introduction-4c44250e2da7]. (If you have no idea what a merkle tree is, don't skip these intros or this article will make little sense to you. Ethereum uses patricia merkle trees which are described here[https://github.com/ethereum/wiki/wiki/Patricia-Tree].

Reading the research on these areas made me start to think about the idea of stateless contracts.

In this article we are going to build two stateless contracts. The first is for a simple bitcoin style token where the balances aren't kept anywhere in our contract. The second will solve a problem that comes up in the first contract that will be solved using patricia trees. This is a long article on something that ended up being a fruitless endeavor(it costs too much gas to be practical), but it is very instructive on how blockchains work and I think you will learn a lot. My

hope is that someone will see a more practical application for these ideas and they will end up being useful in an application outside of a token use case.

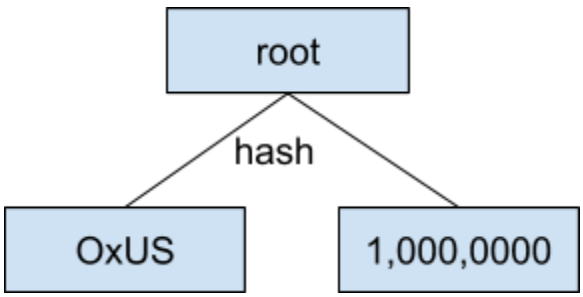
The basic question was ‘can we create a stateless ERC20 token where the balances aren’t on the blockchain?’ The general approach here was that our token contract was going to need to store a merkle root, so it isn’t technically stateless, but it is at least stateless enough that the balances aren’t taking up a bunch of space on the chain. We quickly dismissed being fully ERC20 compliant because we are going to have to provide a proof of our balance to each function and this will break ERC20 compatibility.

So let's create a basic token that hold our balance off chain:

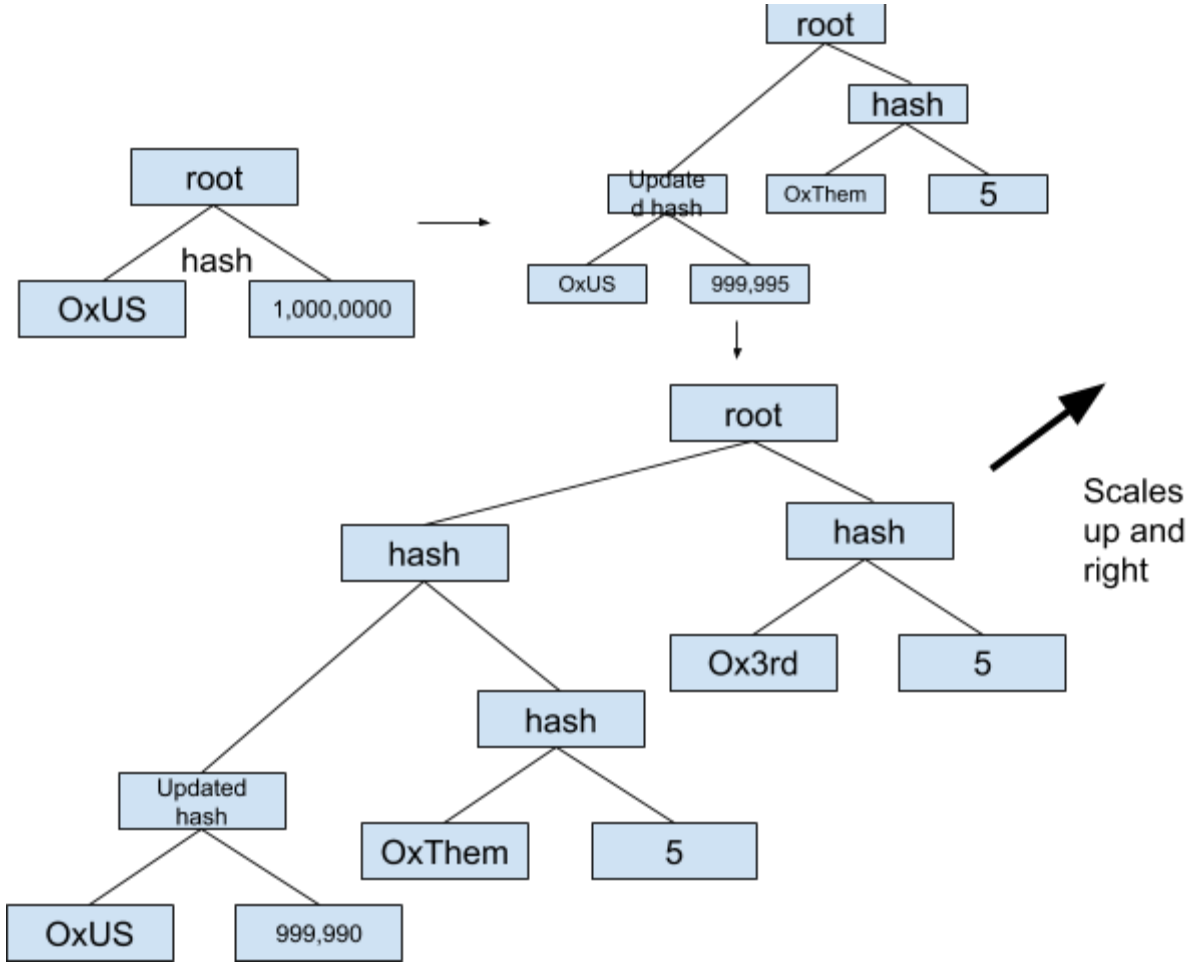
```
pragma solidity ^0.4.18;


contract StatelessToken1 {


   address public owner;
   bytes32 public root;
   uint256 public totalSupply;


   function StatelessToken1() public{
       owner = msg.sender;
       totalSupply = 1000000 * 10**18;
       root = keccak256(bytes32(msg.sender), bytes32(totalSupply));
   }


}
```

In this simple contract we generate a million tokens and give them to us. Instead of giving the balance to our address in a mapping such as mapping(address=> uint) we take a has of our address and the amount of tokens and store this in the root variable.

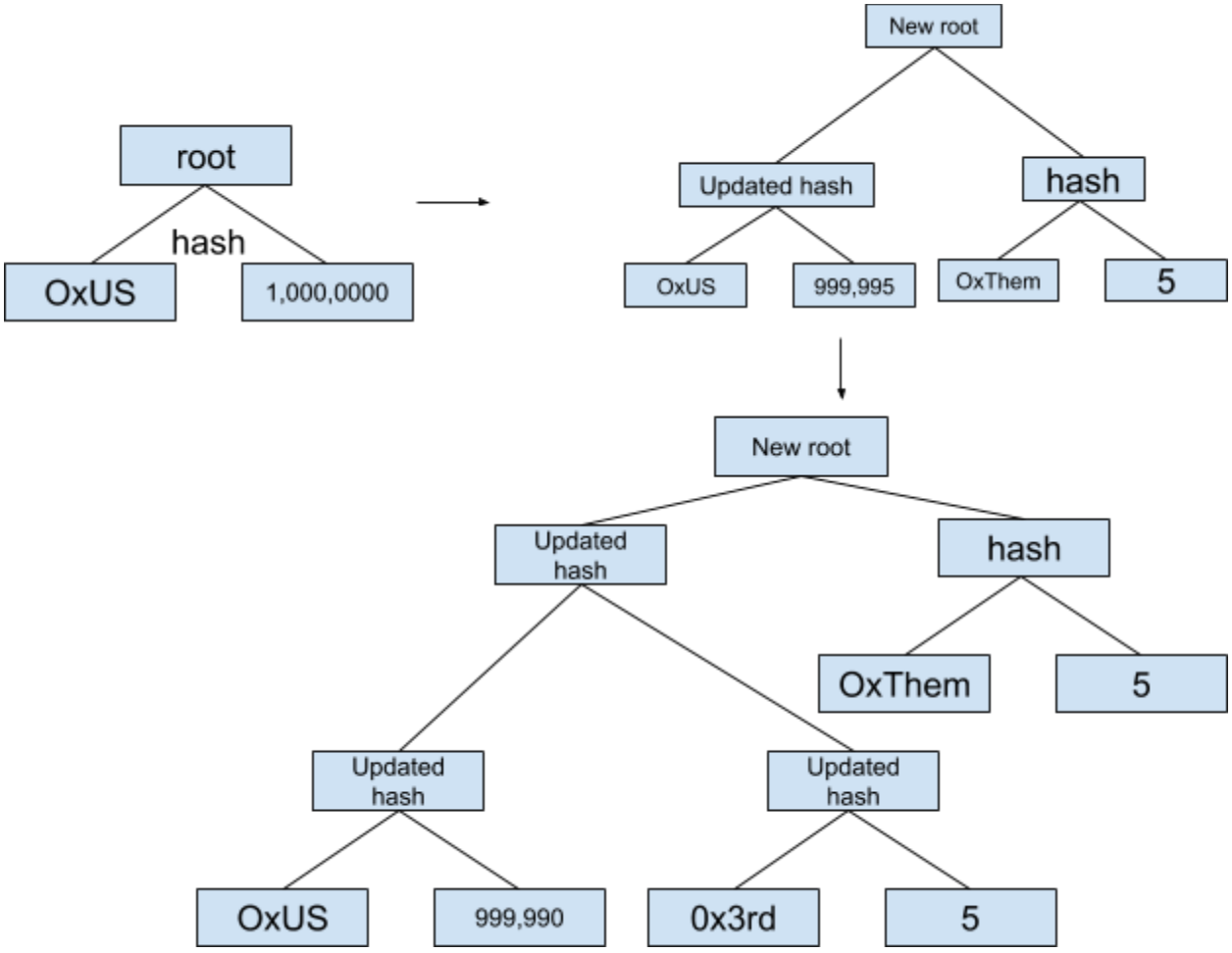
Right now our tree looks like this:



We have a couple different options on how to modify this tree. The first we tried was:



In this scenario we attach each new transaction at the top and update our hash down to the update value. The concern here is that you have an ever growing tree that always gets taller. So we instead went the other way and did the following:



This works better because or pyramid will stay flatter as users pass tokens around. Unfortunately this leaves us with the problem that if someone pays you after you pay them you will have tokens at two different places in the tree. The place where you originally had tokens and the new split next to the person that paid you back. To combat this we'll need some combining and collecting functions.

So now if we want to transfer these tokens around we are going to have to provide a proof that we own them to our transfer function, and for that we are going to need a proof of some kind. Ideally we'd have some database somewhere where we would keep all the details of of our merkle tree. This is what blockchains do when miner's keep the whole state on their computer. But in our case miner's aren't going to have to keep a DB of of our balances because we will provide the proof each time we transfer tokens. The miners have the root and that will be enough to validate the transfer.

For this example we are going to just keep track of our hashes by hand with a few assumptions. The new balance always goes on the right and the old balance goes on the left. Pretty simple. We will need to verify the proof of our balances. To do this I've written the following verifyProof

function that assumes that the proof comes in the form of a bytes32 array where item 0 is the key and item 1 is the balance, 2 or 3 will be the hash of 1 and 2 and the function will need to determine which it is by using the hash. This is repeated until the end of the proof. The final hash should match the given root or the function will throw:

```
   //utility function that can verify merkel proofs
   //todo: can be optimized
   //todo: move to library
   function verifyProof(bytes32 dataPath, bytes32 dataBytes, bytes32[] proof, bytes32
_root) constant public returns(bool){
      bytes32 lastHash;
      bytes32 emptyBytes;

      for(uint thisProof = 0; thisProof < proof.length; thisProof++){
        if(thisProof == 0){ //the key
          require(dataPath == proof[thisProof] );
        } else if(thisProof == 1){ //the value
          require(dataBytes == proof[thisProof]);
          lastHash = keccak256(dataPath, dataBytes);
        } else if(thisProof == proof.length - 1){ //the root
           require(lastHash == proof[thisProof]);
        } else{
          if(proof[thisProof] == lastHash){ //new value is on the right
            lastHash = keccak256(lastHash, proof[thisProof + 1]);
            thisProof++;
          } else { //new value is on the left
            require(proof[thisProof + 1] == lastHash);
            lastHash = keccak256(proof[thisProof], lastHash);
            thisProof++;
          }
        }

       }

      require(lastHash == _root);
      return true;
    }
```

This function is fairly efficient, although it would be easy to streamline. You shouldn't have to pass in the derived hashes since you're already calculating those. You could use some kind of map where 0 is left and 1 is right(we will do this in the next example) or use the fact that the

hashes must match so just do it both ways. You'd want to do a gas calc and figure out how to best do this.

So now we need to actually do the transfer. Here are the two functions that do that:

```
function Transfer(address _to, uint256 _amount, bytes32[] proof){
       //use the verifyProof function to make sure we actually own these tokens
       require(verifyProof(bytes32(msg.sender), proof[1], proof, root));
       //make sure we aren't spending more than we have
       require(_amount < uint256(proof[1]));
       //set the root to the new root
       root = splitBalance(bytes32(msg.sender), bytes32(uint256(proof[1]) - _amount),
bytes32(_to), bytes32(_amount), proof);
   }

   //this function splits an existing balance in two and calculates the root hashes
   // all the way up the tree to the root.
   function splitBalance(bytes32 _sender, bytes32 _senderBalance, bytes32 _to, bytes32
_toBalance, bytes32[] proof) constant public returns(bytes32){
      bytes32 lastHash;
      bytes32 emptyBytes;

      bytes32 newLeaf;

      require(_toBalance > 0);
      //calculate the hash of the new leaf
      newLeaf = keccak256(keccak256(_sender, _senderBalance),
keccak256(_to,_toBalance));

      bytes32 newHash = newLeaf;

       // cycle through the proof and update the hashes to the top
      for(uint thisProof = 0; thisProof < proof.length; thisProof++){
        if(thisProof == 0){
          //do nothing, this is what we are replacing
        } else if(thisProof == 1){
          lastHash = keccak256(proof[0], proof[1]);
          if(lastHash == proof[2]){
           NewBranch(_sender, _senderBalance, _to,_toBalance, proof[2]);
```

```
            } else {
                NewBranch(_sender, _senderBalance, _to,_toBalance, proof[3]);
            }

            SwapLeaf(lastHash, newLeaf);
        } else if(thisProof == proof.length -1){

            //do nothing we have newHash
        }
        else{
            if(proof[thisProof] == lastHash){
                lastHash = keccak256(lastHash, proof[thisProof + 1]);
                newHash = keccak256(newHash, proof[thisProof + 1]);
                SwapLeaf(lastHash, newHash);
                //todo: check and swap equivalence in block
                thisProof++;
            } else {
                lastHash = keccak256(proof[thisProof], lastHash);
                newHash = keccak256(proof[thisProof], newHash);
                SwapLeaf(lastHash, newHash);
                //todo: check and swap equivalence in block
                thisProof++;
            }
        }

      }

      return newHash;
    }
```

Using this function we should be able to load our contract in remix and start sending data around. Before you do though I'll include the whole contract. I'll also add some events so we can keep track of our proofs and tree structure manually.

```
pragma solidity ^0.4.18;

contract StatelessToken1 {

   address public owner;
```

```
    bytes32 public root;
    uint256 public totalSupply;

    event NewBranch(bytes32 indexed addressFrom, bytes32 amountFrom, bytes32 indexed
addressTo, bytes32 amountTo, bytes32 indexed parent);
    event SwapLeaf(bytes32 indexed oldLeaf, bytes32 indexed newLeaf);

    function StatelessToken1() public{
        owner = msg.sender;
        totalSupply = 1000000 * 10**18;
        root = keccak256(bytes32(msg.sender), bytes32(totalSupply));
        NewBranch(0x0, 0x0, bytes32(msg.sender), bytes32(totalSupply), 0x0);
    }

    function Transfer(address _to, uint256 _amount, bytes32[] proof){
        require(verifyProof(bytes32(msg.sender), proof[1], proof, root));
        require(_amount < uint256(proof[1]));
        root = splitBalance(bytes32(msg.sender), bytes32(uint256(proof[1]) - _amount),
bytes32(_to), bytes32(_amount), proof);
    }

    function splitBalance(bytes32 _sender, bytes32 _senderBalance, bytes32 _to, bytes32
_toBalance, bytes32[] proof) constant public returns(bytes32){
      bytes32 lastHash;
      bytes32 emptyBytes;

      bytes32 newLeaf;

      require(_toBalance > 0);

      newLeaf = keccak256(keccak256(_sender, _senderBalance),
keccak256(_to,_toBalance));

      bytes32 newHash = newLeaf;

      for(uint thisProof = 0; thisProof < proof.length; thisProof++){
        if(thisProof == 0){
          //do nothing, this is what we are replacing
        } else if(thisProof == 1){
```

```
            lastHash = keccak256(proof[0], proof[1]);
            if(lastHash == proof[2]){
             NewBranch(_sender, _senderBalance, _to,_toBalance, proof[2]);
            } else {
                NewBranch(_sender, _senderBalance, _to,_toBalance, proof[3]);
            }

            SwapLeaf(lastHash, newLeaf);
          } else if(thisProof == proof.length -1){

             //do nothing we have newHash
          }
          else{
            if(proof[thisProof] == lastHash){
              lastHash = keccak256(lastHash, proof[thisProof + 1]);
              newHash = keccak256(newHash, proof[thisProof + 1]);
              SwapLeaf(lastHash, newHash);
              //todo: check and swap equivalence in block
              thisProof++;
            } else {
              lastHash = keccak256(proof[thisProof], lastHash);
              newHash = keccak256(proof[thisProof], newHash);
              SwapLeaf(lastHash, newHash);
              //todo: check and swap equivalence in block
              thisProof++;
            }
          }

        }

        return newHash;
      }


    //utility function that can verify merkel proofs
    //todo: can be optimized
    //todo: move to library
    function verifyProof(bytes32 dataPath, bytes32 dataBytes, bytes32[] proof, bytes32
_root) constant public returns(bool){
```

```
        bytes32 lastHash;
        bytes32 emptyBytes;

        for(uint thisProof = 0; thisProof < proof.length; thisProof++){
          if(thisProof == 0){
            require(dataPath == proof[thisProof] );
          } else if(thisProof == 1){
            require(dataBytes == proof[thisProof]);
            lastHash = keccak256(dataPath, dataBytes);
          } else if(thisProof == proof.length - 1){
             require(lastHash == proof[thisProof]);
          } else{
            if(proof[thisProof] == lastHash){
              lastHash = keccak256(lastHash, proof[thisProof + 1]);
              thisProof++;
            } else {
              require(proof[thisProof + 1] == lastHash);
              lastHash = keccak256(proof[thisProof], lastHash);
              thisProof++;
            }
          }

         }

        require(lastHash == _root);
        return true;
    }



}
```

Create a new StatelessToken1 contract on the javascript VM with the owner address of 0xca35b7d915458ef540ade6068dfe2f44e8fa733c. You should see a NewBranch function and a SwapLeaf function. This will give us a proof of our funds of:

["0x000000000000000000000000ca35b7d915458ef540ade6068dfe2f44e8fa733c",
"0x0000000000000000000000000000000000000000000000000d3c21bcecceda1000000",
"0x4443014fa5601b06c1d03c375bd830fd5a3bfb47165f8b165e220956667cc959"]

This proof is [address, balance, currentRoot].

We can verify this proof by passing the following to the verifyProof function(cut and paste the following into the verifyProof function in the Remix contract:

"0x000000000000000000000000ca35b7d915458ef540ade6068dfe2f44e8fa733c","0x0000000000000000000000000000000000000000000000d3c21bcecceda1000000",["0x000000000000000000000000ca35b7d915458ef540ade6068dfe2f44e8fa733c","0x0000000000000000000000000000000000000000000000d3c21bcecceda1000000","0x4443014fa5601b06c1d03c375bd830fd5a3bfb47165f8b165e220956667cc959"],"0x4443014fa5601b06c1d03c375bd830fd5a3bfb47165f8b165e220956667cc959"

It should return true. Change something like the root at the end and you'll see that it will return false.

Now transfer 50 tokens to 0x14723a09acff6d2a60dcdf7aa4aff308fddc160c using the transfer function with the input:

"0x14723a09acff6d2a60dcdf7aa4aff308fddc160c",
"0x14723a09acff6d2a60dcdf7aa4aff308fddc160c", 50,
["0x000000000000000000000000ca35b7d915458ef540ade6068dfe2f44e8fa733c","0x0000000000000000000000000000000000000000000000d3c21bcecceda1000000","0x4443014fa5601b06c1d03c375bd830fd5a3bfb47165f8b165e220956667cc959"]

You are going to see two events occur. The first is a newBranch event that shows you the structure of the new branch that has the from side and the to side as well as the place that that branch is inserted. The second event tells you that the leaf 0x4443014fa5601b06c1d03c375bd830fd5a3bfb47165f8b165e220956667cc959 has been swapped out for 0xf0083e7b1d4f4b0b06f313d774ccef179607ea75a680b494f412b8542d570999.

Now we have two accounts holding tokens but only one root variable in our contract.

We also have the following two proofs that we can use to further send tokens around:

Account 0xca35b7d915458ef540ade6068dfe2f44e8fa733c
["0x000000000000000000000000ca35b7d915458ef540ade6068dfe2f44e8fa733c",
"0x0000000000000000000000000000000000000000000000d3c21bcecceda0ffffce",
"0x78bb46f72deed75bf09fd8453a273394a94086ef8ad2801a4f003c578d191588",
"0xe17579cdcc9ec1f0e773c0052664d4101074cc942700e4db8c2343699a716a72",
"0xF0083E7B1D4F4B0B06F313D774CCEF179607EA75A680B494F412B8542D570999"]

Account 0x14723a09acff6d2a60dcdf7aa4aff308fddc160c
["0x000000000000000000000000014723a09acff6d2a60dcdf7aa4aff308fddc160c","0x0000000000000000000000000000000000000000000000000000000000000032","0x78bb46f72deed75bf0

9fd8453a273394a94086ef8ad2801a4f003c578d191588","0xe17579cdcc9ec1f0e773c0052664d4101074cc942700e4db8c2343699a716a72","0xF0083E7B1D4F4B0B06F313D774CCEF179607EA75A680B494F412B8542D570999"]

The execution cost of our contract was only 12,494 gas. Normally we would expect it to be at least 45,000 as we update one balance in a mapping(5,000 gas) and add a new mapping to the receiver(20,000 gas x 2). We also have a really short proof at this point so that amount may increase over time. The transaction gas is 39,900 which I think is higher than a normal transfer on an ERC20 because of additional bytes in the input parameters. These seem to be pretty cheap though.

Now lets transfer 500 tokens to 0x4b0897b0513fdc7c541b6d9d7e929c4e5364d2db by calling Transfer with:

"0x4b0897b0513fdc7c541b6d9d7e929c4e5364d2db", 500,
["0x0000000000000000000000ca35b7d915458ef540ade6068dfe2f44e8fa733c","0x0000000000000000000000000000000000000000000000000d3c21bcecceda0ffffce","0x78bb46f72deed75bf09fd8453a273394a94086ef8ad2801a4f003c578d191588","0xe17579cdcc9ec1f0e773c0052664d4101074cc942700e4db8c2343699a716a72","0xF0083E7B1D4F4B0B06F313D774CCEF179607EA75A680B494F412B8542D570999"]

This transaction costs a little more(15,000 execution gas, 47,000 transaction gas), but now we have three balances on our system.

Our proofs are now:

Account 0xca35b7d915458ef540ade6068dfe2f44e8fa733c
["0x0000000000000000000000ca35b7d915458ef540ade6068dfe2f44e8fa733c","0x0000000000000000000000000000000000000000000000000d3c21bcecceda0fffdda","0x251e9297a67b0dad6685a19928b7a0c12e567f1b6bac95ffadde3bcdcc113ea1",
"0x95b1c1225870387f790aafbb6c39fe64da6b7dfe7950ddb73eb21855d63fc432","0xe30e43066705830a3f664b229b96a40d0b14b10bdf43210feb8480e4d5d9ea33","0xe17579cdcc9ec1f0e773c0052664d4101074cc942700e4db8c2343699a716a72","0x0fc579d49ecf4ddb529d243475bbe1b9680cdfc3429a937b65f1f617b287d0e1"]

Account 0x14723a09acff6d2a60dcdf7aa4aff308fddc160c
["0x0000000000000000000000014723a09acff6d2a60dcdf7aa4aff308fddc160c","0x0000000000000000000000000000000000000000000000000000000000000032","0xe30e43066705830a3f664b229b96a40d0b14b10bdf43210feb8480e4d5d9ea33","0xe17579cdcc9ec1f0e773c0052664d4101074cc942700e4db8c2343699a716a72","0x0fc579d49ecf4ddb529d243475bbe1b9680cdfc3429a937b65f1f617b287d0e1"]

Account "0x4b0897b0513fdc7c541b6d9d7e929c4e5364d2db"
["0x0000000000000000000000004b0897b0513fdc7c541b6d9d7e929c4e5364d2db",
"0x00000000000000000000000000000000000000000000000000000000000001f4","0x251e9297a67b0dad6685a19928b7a0c12e567f1b6bac95ffadde3bcdcc113ea1",

"0x95b1c1225870387f790aafbb6c39fe64da6b7dfe7950ddb73eb21855d63fc432","0xe30e43066705830a3f664b229b96a40d0b14b10bdf43210feb8480e4d5d9ea33","0xe17579cdcc9ec1f0e773c0052664d4101074cc942700e4db8c2343699a716a72","0x0fc579d49ecf4ddb529d243475bbe1b9680cdfc3429a937b65f1f617b287d0e1"]

Now lets move 10 of 0x14723a09acff6d2a60dcdf7aa4aff308fddc160c(make sure you change your executing address in remix) to 0x4b0897b0513fdc7c541b6d9d7e929c4e5364d2db and see how this works using the transfer with: "0x4b0897b0513fdc7c541b6d9d7e929c4e5364d2db", 10, ["0x0000000000000000000000014723a09acff6d2a60dcdf7aa4aff308fddc160c","0x0000000000000000000000000000000000000000000000000000000000000032","0xe30e43066705830a3f664b229b96a40d0b14b10bdf43210feb8480e4d5d9ea33","0xe17579cdcc9ec1f0e773c0052664d4101074cc942700e4db8c2343699a716a72","0x0fc579d49ecf4ddb529d243475bbe1b9680cdfc3429a937b65f1f617b287d0e1"]

This works! And now we see the weakness in this schema when we look at the resulting proofs:

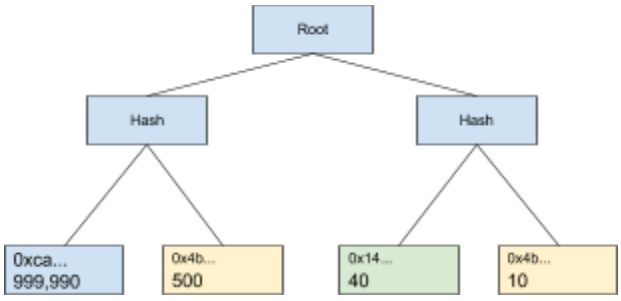
Account 0xca35b7d915458ef540ade6068dfe2f44e8fa733c ["0x0000000000000000000000ca35b7d915458ef540ade6068dfe2f44e8fa733c","0x0000000000000000000000000000000000000000d3c21bcecceda0fffdda","0x251e9297a67b0dad6685a19928b7a0c12e567f1b6bac95ffadde3bcdcc113ea1", "0x95b1c1225870387f790aafbb6c39fe64da6b7dfe7950ddb73eb21855d63fc432","0xe30e43066705830a3f664b229b96a40d0b14b10bdf43210feb8480e4d5d9ea33","0x2938917a3b8bd55748081eef4944ff558894f11d59208a45b423b2e9b1c35351","0x9681cde0bdcb384f29361c8ffa3e34a04602253171a52c278eb7528ca6500e08"]

Account 0x14723a09acff6d2a60dcdf7aa4aff308fddc160c ["0x0000000000000000000000014723a09acff6d2a60dcdf7aa4aff308fddc160c", "0x0000000000000000000000000000000000000000000000000000000000000028","0xf58fad2639ecaab94b52ba5c81be97be3e9675d743da9e33991559f02d43dd51","0x80f2bd00854b7c2e70183447d115a36655823c1c0e5d91f777f0f2425c8d5e72","0xe30e43066705830a3f664b229b96a40d0b14b10bdf43210feb8480e4d5d9ea33","0x2938917a3b8bd55748081eef4944ff558894f11d59208a45b423b2e9b1c35351","0x9681cde0bdcb384f29361c8ffa3e34a04602253171a52c278eb7528ca6500e08"]

Account "0x4b0897b0513fdc7c541b6d9d7e929c4e5364d2db" - balance 1 ["0x0000000000000000000000004b0897b0513fdc7c541b6d9d7e929c4e5364d2db", "0x00000000000000000000000000000000000000000000000000000000000001f4","0x251e9297a67b0dad6685a19928b7a0c12e567f1b6bac95ffadde3bcdcc113ea1", "0x95b1c1225870387f790aafbb6c39fe64da6b7dfe7950ddb73eb21855d63fc432","0xe30e43066705830a3f664b229b96a40d0b14b10bdf43210feb8480e4d5d9ea33","0x2938917a3b8bd55748081eef4944ff558894f11d59208a45b423b2e9b1c35351","0x9681cde0bdcb384f29361c8ffa3e34a04602253171a52c278eb7528ca6500e08"]

Account "0x4b0897b0513fdc7c541b6d9d7e929c4e5364d2db" - balance 2
["0x0000000000000000000000004b0897b0513fdc7c541b6d9d7e929c4e5364d2db",
"0x000000000000000000000000000000000000000000000000000000000000000a","0xf58fad2639ecaab94b52ba5c81be97be3e9675d743da9e33991559f02d43dd51","0x80f2bd00854b7c2e70183447d115a36655823c1c0e5d91f777f0f2425c8d5e72","0xe30e43066705830a3f664b229b96a40d0b14b10bdf43210feb8480e4d5d9ea33","0x2938917a3b8bd55748081eef4944ff558894f11d59208a45b423b2e9b1c35351","0x9681cde0bdcb384f29361c8ffa3e34a04602253171a52c278eb7528ca6500e08"]

In merkle tree looks like this:



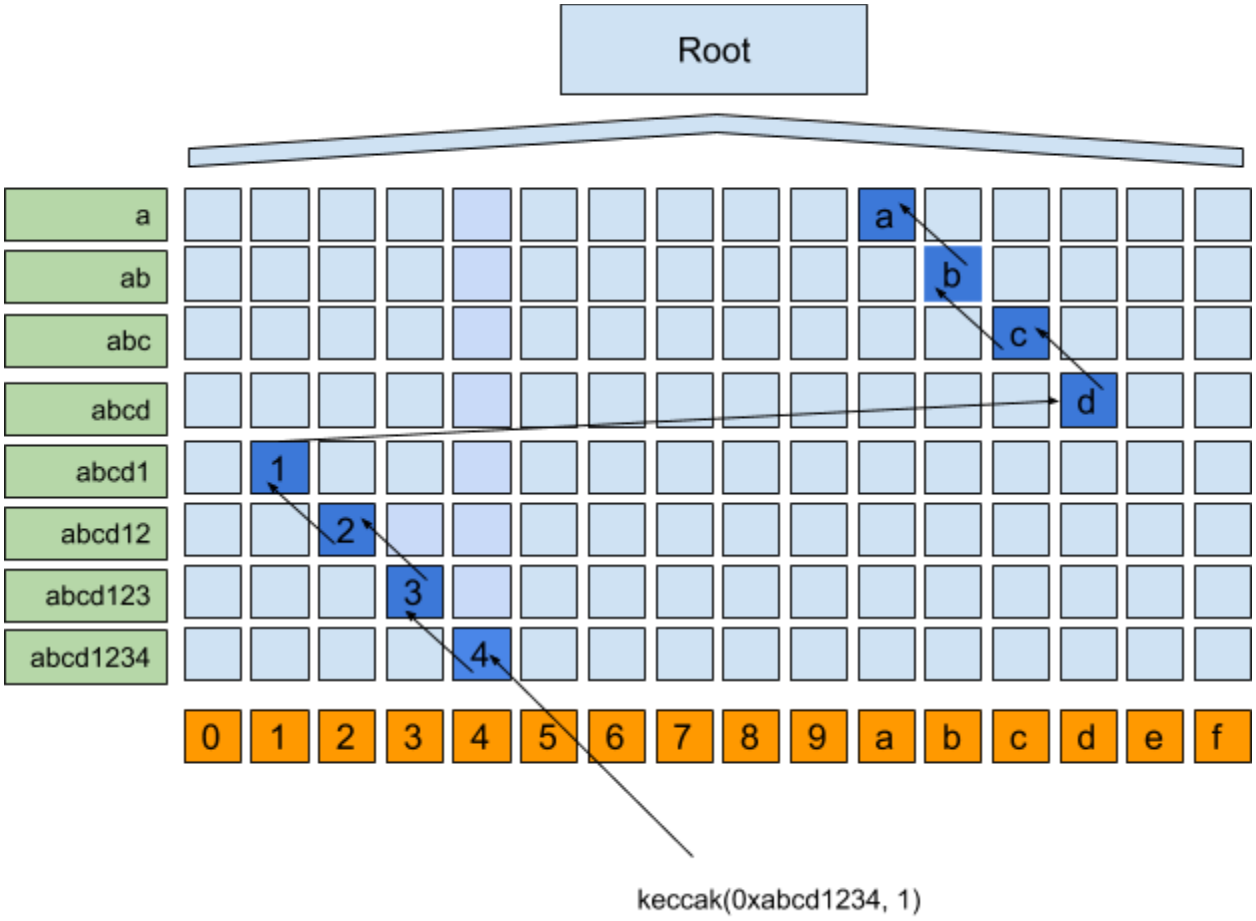
As you can see "0x4b0897b0513fdc7c541b6d9d7e929c4e5364d2db" has a balance in two places. They have a total of 510 tokens across two different UTXOs(unspent transaction outputs). If they want to spend 505 tokens in one transaction they will have to combine their UTXOs first. To do this you would need some kind of atomic combine and reduce function to clean up the 'dust'. I'll leave it to the reader to implement these functions. Reverting a leaf with a 0 balance should be easy enough by just replacing the branch with the remaining leaf. Updating the remote leaf is where things get tough. This also exposes a big flaw with this system as I have it set up. Once you zero out one leaf it is going to swap out hashes all the way up to the root. When you go to do the second combining function at the other leaf some of the items in that proof will have switched out from under you.

This is a big issue because you are basically limited to one transaction per block unless you implement some kind of tracking mechanism for swapped hashes. In theory this tracking is really easy just track a mapping of swapped hashes, but in practice this begins to cost a lot of gas. 40,000 gas for each swap! We've now lost any advantage we had on the gas front. And as our proofs get longer the costs go even higher. I've asked about the potential for short term storage op codes that would alleviate the problem, but there doesn't seem to be much traction. Perhaps this example contract will increase the visibility of the problem[https://ethresear.ch/t/block-persistent-storage/817/5].

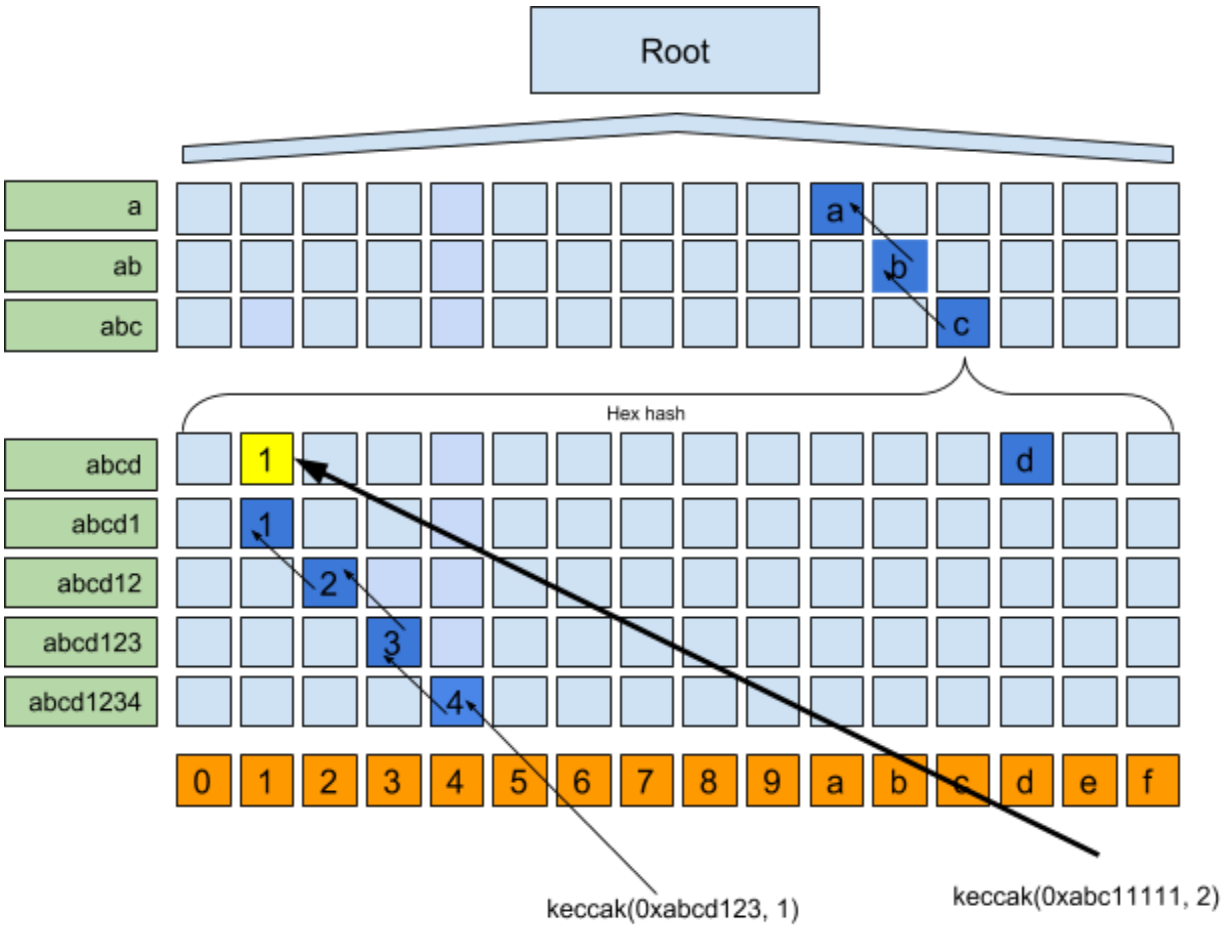
In trying to solve this problem eloquently I ran across a great solution. Ethereum! It turns out this problem has been solved before. Ethereum uses Merkle Patricia Trees to track state,

transactions, and logs(three different trees). Merkle patricia Trees are interesting because their structure implies that everything that you could store has a place already allocated to it. Because of this when we want to update a balance for an address we can know the proof for the start balance before hand. Even, and especially if, the balance is 0. So I put StatelessToken1 aside and started on StatelessToken2.

The key to a patricia tree vs the kind we were doing in part 1 is that a patricia tree takes hashes of 16 items at a time and these 16 items correspond to the hex value of the possible half bytes of each item you want in your set. Values in ethereum default to 32 bytes, but for a short example, let's take something that is 4 bytes long: 0xabcd1234. A 4 byte value has 8 half bytes: a, b, c, d, 1, 2, 3 and 4. Say the value of this key was 1. We would hash keccak(0xabcd123, 1) = 0xffd53db7d2ddd1b87d1578b8fbd9940d72d14bb00810c4d77bfc79d086f282b4. So at the address (a, b, c, d, 1, 2, 3, 4) we would put 0xffd53db7d2ddd1b87d1578b8fbd9940d72d14bb00810c4d77bfc79d086f282b4 . It would look something like this:



Now take the above and make the height 64 instead of 8 and that is the calculation we are looking at doing. If we had another key value pair of 0xabc1111, 2 the hash calculation would merge with our original at the abc layer as below(yellow box).



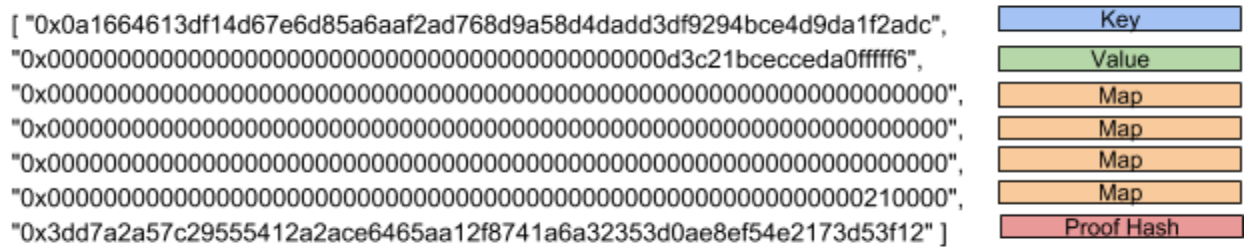
Most of the items are light blue and are null. In my specific implementation I assume that if there is only one value on a row, that we carry the last hash from the layer below up until we get to the next layer with a population greater than 1. I don't think this is how Ethereum actually does things, but it helps save us from some pretty big hash calculation along the way.

Ok, so now we are going to build a token on this. First we need a good way to produce proofs. I've proposed the following proof structure:

[
Key,
Value,
Map1,
Map2,

Map3,
Map4,
Hash1,
….
HashN
]
The Maps are binary maps that show where hashes need to be inserted. Since each layer has a length of 16(0-f) and we have 64 layers we need 64*16 bits = 1024 bits = 128 bytes. Thus we have 4 maps.

A proof will look something like:



There is a lot of wasted space for small data sets, but getting solidity to do a bunch of dynamic compression stuff seems hard and overkill for this example. Real Ethereum uses something RLP Encoding and decoding and our maps could probably benefit from that but the we'd have to build a decoder in solidity[https://github.com/ethereum/wiki/wiki/RLP].

I've created a simple class that keeps track of items in our set and can produce proofs. It isn't an efficient solution and will bog down if it has to deal with a big data set:

```
class PatriciaTree
 constructor: ()->
   @web3Utils = require('web3-utils')
   @root = null
   @items = []
   @lookUps = {}
 addItem: (key, value)=>
```

```
  #adds and updates items in our collection
  @lookUps[key] = [key, value]
  for thisItem in @items
    if thisItem[0] is key
      thisItem[1] = value
      return
  @items.push [key, value]
bin2hex: (b) ->
  #utility function to translate binary to hex
  b.match(/.{4}/g).reduce ((acc, i) ->
    acc + parseInt(i, 2).toString(16)
  ), ''
getProof: (key)=>
  #calculates a proof for a given key
  proof = []
  thisHash = null
  if !@lookUps[key]?
    #if we don't have an item then its value is 0
    proof.push key
    proof.push "0x0000000000000000000000000000000000000000000000000000000000000000"
  else
    #item 0 = key
    #item 1 = value
    proof.push @lookUps[key][0]
    proof.push @lookUps[key][1]
    thisHash = @web3Utils.soliditySha3(proof[0],proof[1])

  #place holders for our value map
  proof.push("")
  proof.push("")
  proof.push("")
  proof.push("")

  mapString = ""

  #trim off the 0x
  thisPath = key.slice(2)

  while thisPath.length > 0
```

```
      #loop until we get to the top
      thisPath = thisPath.substring(0,thisPath.length - 1)

      if thisPath.length == 0
        #we are at the top, use a special placeholder
        thisPath = "_"

      foundPath = @lookUps[thisPath]

      mapSubString = ""
      if foundPath?
        if @isPathSingular(thisPath) && thisHash isnt null
          # if the path is singular and we don't currently have a hash, our map can be
all zeros for this set of 16
          mapSubString = mapSubString + "0000000000000000"
        else
          for thisItem in foundPath
            #loop through each item in this path

            if thisItem?
              #if the item exists we need to put a 1 in our binary map
              mapSubString = mapSubString + "1"
              if thisItem isnt thisHash
                #push the item onto the proof
                proof.push thisItem
            else
              #if the item doesn't exist we need to put a 0 on our binary map
              mapSubString = mapSubString + "0"
          #calculate the hash of this layer
          thisHash = @calcPathHash(thisPath)
      else
        #if the path doesn't exist we can spit out 0 and use our current hash
        mapSubString = mapSubString + "0000000000000000"
      if thisPath is "_"
        thisPath = ''

      mapString = mapString + mapSubString

    #convert the binary to hex
```

```
    bigHex = @bin2hex(mapString)

    #split the hex up
    proof[2] = "0x" + bigHex.substring(0, 64)
    proof[3] = "0x" + bigHex.substring(64, 128)
    proof[4] = "0x" + bigHex.substring(128, 192)
    proof[5] = "0x" + bigHex.substring(192, 256)
    return proof
  seedPath: (hash, location)=>
    #creates a 16 length bytes32 array with our current hash in the location it should
be in
    resultMap = []
    for thisPos in [0...16]
      if parseInt(location,16) == thisPos
        resultMap.push hash
      else
        resultMap.push null
    return resultMap
  isPathSingular: (path)=>
    #determines if the current path has 0 or 1 items in it
    thisPath = @lookUps[path]
    count = 0
    for thisItem in thisPath
      if thisItem?
        count = count + 1
        if count > 1
          return false
    return true
  calcPathHash: (path)=>
    #calculates the hash of a path
    emptyItem =
        t:"bytes"
        v:"0x0000000000000000000000000000000000000000000000000000000000000000"
    foundPath = @lookUps[path]
    newHash = @web3Utils.solioditySha3(
      if !foundPath[0]? then emptyItem else {t:"bytes", v:foundPath[0]}
      if !foundPath[1]? then emptyItem else {t:"bytes", v:foundPath[1]}
      if !foundPath[2]? then emptyItem else {t:"bytes", v:foundPath[2]}
      if !foundPath[3]? then emptyItem else {t:"bytes", v:foundPath[3]}
```

```
    if !foundPath[4]? then emptyItem else {t:"bytes", v:foundPath[4]}
    if !foundPath[5]? then emptyItem else {t:"bytes", v:foundPath[5]}
    if !foundPath[6]? then emptyItem else {t:"bytes", v:foundPath[6]}
    if !foundPath[7]? then emptyItem else {t:"bytes", v:foundPath[7]}
    if !foundPath[8]? then emptyItem else {t:"bytes", v:foundPath[8]}
    if !foundPath[9]? then emptyItem else {t:"bytes", v:foundPath[9]}
    if !foundPath[10]? then emptyItem else {t:"bytes", v:foundPath[10]}
    if !foundPath[11]? then emptyItem else {t:"bytes", v:foundPath[11]}
    if !foundPath[12]? then emptyItem else {t:"bytes", v:foundPath[12]}
    if !foundPath[13]? then emptyItem else {t:"bytes", v:foundPath[13]}
    if !foundPath[14]? then emptyItem else {t:"bytes", v:foundPath[14]}
    if !foundPath[15]? then emptyItem else {t:"bytes", v:foundPath[15]})
  return newHash
update: (path, location, hash)=>
  #updates the layer of a path
  thisPath = @lookUps[path]
  if !thisPath?
    #this path doesn't exist so seed it
    thisPath = @seedPath(hash, location)
    if thisPath.length > 16
      throw new error(location)
    @lookUps[path] = thisPath
  else
    #update this path with the hash at the given location
    @lookUps[path][parseInt(location, 16)] = hash
  newHash = hash

  if @isPathSingular(path)
    #if the path is singulare we can just pass our existing has up a layer
    if path isnt "_"
      newPath = path.substring(0,path.length - 1)
      if newPath.length == 0
        newPath = "_"
      #recursively call
      newHash = @update(newPath, path.substring(path.length - 1), hash)
    else
      #we found our root
      @root = newHash
  else
```

```
      #update the hash value
      emptyItem =
        t:"bytes"
        v:"0x0000000000000000000000000000000000000000000000000000000000000000"

      newHash = @calcPathHash(path)
      if path isnt "_"
        #recursively call
        newPath = path.substring(0,path.length - 1)
        if newPath.length == 0
          newPath = "_"
        newHash = @update(newPath, path.substring(path.length - 1), newHash)
      else
        #we found our root
        @root = newHash
    return newHash
  buildTree: ()=>
    for thisItem in @items
      #loop through each item in our collection and calculate the root
      thisHash =
@web3Utils.soliditySha3({t:"bytes",v:thisItem[0].slice(2)},{t:"bytes",v:thisItem[1].sl
ice(2)})
      thisPath = thisItem[0].slice(2)
      aByte = thisPath[thisPath.length - 1]
      findPath = thisPath.substring(0, thisPath.length - 1)
      @root = @update(findPath, aByte, thisHash)
    return

exports.PatriciaTree = PatriciaTree
```

Using this javascript object we can easily produce proofs for our data sets. We can call the .addItem(key, value) function and pass in a key, value. Once we have all the values that we want in the object we call .buildTree(). Once the tree is built we can call .getProof(key) and it will spit out a proof. To update values just call addItem(key, value) with the same keys and items will be updated. Currently .buildTree recalcs the entire data space so it is far from efficient. If this were a real system we'd only need to update the branches that were updating and we'd use some kind of database to keep track of all existing branches.

Now that we can produce proofs we want to start transferring tokens around. Our contract generates 1 million tokens(with 18 decimal places). I'll go ahead and give the entire contract and then speak about each function although comments are included in the code.

```
pragma solidity ^0.4.18;

contract StatelessToken2 {

    address public owner;
    bytes32 public patriciaRoot;
    uint256 public totalSupply;

    event NewBalance(address owner, bytes32 key, bytes32 value);
    event NewRoot(bytes32 newRoot);
    event calcingHex(bytes32[16] list);

    function StatelessToken2() public{
        //constructor
        owner = msg.sender;
        totalSupply = 1000000 * 10**18;
        patriciaRoot =  keccak256(keccak256(keccak256("balance"),bytes32(msg.sender)),
bytes32(totalSupply));
        NewBalance(msg.sender, keccak256(keccak256("balance"),bytes32(msg.sender)),
bytes32(totalSupply));
    }

    function verifySuperProof(bytes32[] proof) constant public returns(bytes32){
        //proof[0] = key
        //proof[1] = value

        bytes32 lastHash;

        if(proof[1] == 0x0){
            //if the first value is 0 then the hash will be 0 up until we hit the merge
point with existing data
            lastHash = 0x0;
        } else {
            lastHash = keccak256(proof[0], proof[1]);
        }
```

```
        bytes32[16] memory list;
        uint currentPosition;

        uint thisKeyByte = 0;
        uint currentKeyPosition = 0;
        bool bFoundHash = false;

        //we split into 4 groups so we can follow the map in proof[2-5]
        for(uint thisHashGroup = 0; thisHashGroup < 4 ; thisHashGroup++){
            for(uint thisHashByte = 0; thisHashByte < 32 ; thisHashByte++){

                //create a list for the current level
                (list, bFoundHash, currentPosition) = createList([proof[2 +
thisHashGroup][thisHashByte], proof[2 + thisHashGroup][thisHashByte + 1]],
getKeyHalf(currentKeyPosition, thisKeyByte, proof), currentPosition, lastHash, proof);


                if(bFoundHash){
                    //if we found data in this level we need to calculate the hex hash
                    calcingHex(list);
                    lastHash = hexHash(list);
                }

                bFoundHash = false;
                currentKeyPosition++;

                if(currentKeyPosition % 2 == 0){
                    //each key position need to advance every two ints
                    thisKeyByte++;
                }

                thisHashByte ++;
            }
        }

        return  lastHash;
    }
```

```
   function superTransfer(address destination, uint256 amount, bytes32[] memory
destinationProof, bytes32[] memory senderProof) public returns(bytes32) {
        //this function transfers tokens from one address to another

        //make sure we are sending a positive amount of tokens
        require(uint256(senderProof[1]) >= amount);

        //make sure the sender actually sent this transaction
        require(keccak256(keccak256("balance"), bytes32(msg.sender)) ==
senderProof[0]);

        //makesure that the proof is proving the destination
        require(keccak256(keccak256("balance"), bytes32(destination)) ==
destinationProof[0]);

        //make sure that the destination proof renders to the current patriciaRoot
        require(verifySuperProof(destinationProof) == patriciaRoot);

        //makes sure that the sender proof renders to the current patriciaRoot
        require(verifySuperProof(senderProof) == patriciaRoot);



        uint commonPosition;
        uint8 senderHalf;
        uint8 destinationHalf;

        //calculate the common position where the sender balance meets the destination
balance
        (commonPosition, senderHalf, destinationHalf) =
findCommonPosition(senderProof[0],destinationProof[0]);

        //update the sender proof
        senderProof[1] = bytes32(uint256(senderProof[1]) - amount);

        //update the destination proof
        destinationProof[1] = bytes32(uint256(destinationProof[1]) + amount);

        bytes32 senderHash;
        bytes32 destinationHash;
```

```
        bytes32[16] memory list;
        uint256 currentPosition;
        uint256 irrPosition;

        //build the sender Proof to the join point
        (senderHash, list, currentPosition) = buildSuperProofToPosition(senderProof,
commonPosition);

        //build the dest Poof to the join point
        (destinationHash, list, irrPosition) =
buildSuperProofToPosition(destinationProof, commonPosition);

        //get the current state of the join layer
        (list,,currentPosition) = createList([senderProof[2 +
(commonPosition/16)][(commonPosition % 16) *2],
senderProof[2+(commonPosition/16)][((commonPosition % 16) *2)+ 1]],
getKeyHalf(commonPosition % 2, commonPosition / 2, senderProof), currentPosition,
senderHash, senderProof);

        //update the list witht he new values
        list[senderHalf] = senderHash;
        list[destinationHalf] = destinationHash;

        //recalc the has
        senderHash = hexHash(list);
        calcingHex(list);

        //calculate the new root from the join point up to the root
        patriciaRoot =  buildSuperProofPastPosition(senderProof, commonPosition + 1,
currentPosition, senderHash);

        //log events
        NewRoot(patriciaRoot);
        NewBalance(msg.sender, senderProof[0],senderProof[1]);
        NewBalance(destination, destinationProof[0], destinationProof[1]);


        return patriciaRoot;
    }
```

```
   function createList(byte[2] memory pair, uint16 keyHalf, uint currentPosition,
bytes32 lastHash, bytes32[] proof) pure returns(bytes32[16] memory list, bool
bFoundHash, uint newPosition){
       //this function builds a list layer based on a proof and the map

       if(pair[0] == 0x0 && pair[1]== 0x0){
           //we can save a lot of gas by skipping this if the map is 00000000 00000000
           return;
       }

       //convert the map to to a binary list
       bool[16] memory mapList = buildHashLayerMap(pair[0],pair[1]);
       newPosition = currentPosition;

       //loop throug the binary list and build the hex list of hashes
       for(uint thisListPosition = 0; thisListPosition < 16; thisListPosition++){

           if(thisListPosition == keyHalf){
               //add our current hash value at our current position
               list[thisListPosition]= lastHash;
           } else if (mapList[thisListPosition]){
               //if the map has a 1 we need to add the current has an let our function
know to move on to the next hash
               if(proof[6 + newPosition] == lastHash){
                   newPosition++;
               }
               list[thisListPosition] = proof[6 + newPosition];
               bFoundHash = true;
               newPosition++;
           } else {
               //if th emap has a zero, put 0x0 in the spot
               list[thisListPosition] = 0x0;
           }
       }
   }

   function buildHashLayerMap(byte firstByte, byte secondByte) pure returns(bool[16]
result){
```

```
        //converts two bytes to a 16 length true / false list
        for(uint thisPosition = 0; thisPosition < 8; thisPosition++){
        result[7 - thisPosition] =  2**thisPosition == uint8(firstByte &
byte(2**thisPosition ));
        }

        for(thisPosition = 0; thisPosition < 8; thisPosition++){
        result[7 - thisPosition + 8] =  2**thisPosition == uint8(secondByte &
byte(2**thisPosition ));
        }
    }


   function hexHash(bytes32[16] list) pure returns (bytes32 newHash){
        //calculates the has of a list of 16 hashes
        //if there is only one item we only need to return the one hash we findor
        //empty hash

        uint count = 0;
        bytes32 lastHash = 0x0;
        for(uint thisItem = 0; thisItem < 16; thisItem ++){
            if(list[thisItem] != 0x0){
                lastHash = list[thisItem];
                count++;
                if(count > 1){

                    break;
                }
            }
        }

        if(count > 1){
            //we found more than one hash so calc the hash
            newHash = keccak256(list);
        }
        else{
            //we found one or zero so return the last has we found
            newHash = lastHash;
        }
```

```
    }

    function findCommonPosition(bytes32 key1, bytes32 key2) pure returns(uint
currentPosition, uint8 keyHalf1, uint8 keyHalf2){

        //cycles through the address of the source and dest keys and finds where their
patricia root calcs should merge

        currentPosition = 63;
        for(uint thisHashByte = 0; thisHashByte < 32 ; thisHashByte++){

        keyHalf1 = uint8(key1[thisHashByte]) / 16;
        keyHalf2 = uint8(key2[thisHashByte]) / 16;
        if(keyHalf1 != keyHalf2){
            return;
        }

        currentPosition--;

        keyHalf1 = uint8(key1[thisHashByte] & byte(0x0F));
        keyHalf2 = uint8(key2[thisHashByte] & byte(0x0F));
        if(keyHalf1 != keyHalf2){
            return;
        }

        currentPosition--;



        }

        return;

    }

    function getKeyHalf(uint currentKeyPosition, uint thisKeyByte, bytes32[] proof)
pure returns (uint8 keyHalf){
        //find the current key half
        if(currentKeyPosition % 2 == 0 && thisKeyByte < 32){
            keyHalf = uint8(proof[0][32 - thisKeyByte - 1] & byte(0x0F));
```

```
        } else {
            keyHalf = uint8(proof[0][32 - thisKeyByte - 1]) / 16;
        }
    }
```

```
   function buildSuperProofToPosition(bytes32[] proof, uint position) constant public
returns(bytes32 lastHash, bytes32[16] memory list, uint256 currentPosition) {
        //calcs an update hash up to a common position
        if(proof[1] == 0x0) {
            lastHash = 0x0;
        } else {
            lastHash = keccak256(proof[0], proof[1]);
        }


        uint thisKeyByte = 0;
        uint currentKeyPosition = 0;
        bool bFoundHash = false;

        for(uint thisHashGroup = 0; thisHashGroup < 4 ; thisHashGroup++) {

            for(uint thisHashByte = 0; thisHashByte < 32 ; thisHashByte++) {
                //get the current list at this layer
                (list, bFoundHash, currentPosition) = createList([proof[2 +
thisHashGroup][thisHashByte], proof[2 + thisHashGroup][thisHashByte + 1]],
getKeyHalf(currentKeyPosition, thisKeyByte, proof), currentPosition, lastHash, proof);

                if(bFoundHash){
                    //recalc a hash
                    calcingHex(list);
                    lastHash = hexHash(list);
                }

                bFoundHash = false;
                currentKeyPosition++;
```

```
                if(currentKeyPosition % 2 == 0){
                    thisKeyByte++;
                }

                thisHashByte ++;

                if((thisHashGroup * 16) + (thisHashByte / 2) >= position - 1) {
                    return;
                }
            }
        }
        return;
    }

    function buildSuperProofPastPosition(bytes32[] proof, uint position, uint256
currentPosition, bytes32 lastHash) constant public returns(bytes32) {
        // calculates the patricia root for a proof past a common position
        if(position >= 63){
            return lastHash;
        }
        uint thisKeyByte = position/2;
        uint currentKeyPosition = 0;
        bool bFoundHash = false;
        bytes32[16] memory list;

        uint thisHashGroup = position/16;

        for(thisHashGroup = thisHashGroup; thisHashGroup < 4 ; thisHashGroup++){
            uint thisHashByte;
            if(thisHashGroup == position/16){

                thisHashByte = (position * 2) - (thisHashGroup * 16 * 2);

            } else {
                thisHashByte = 0;
            }

            for(thisHashByte = thisHashByte; thisHashByte < 32 ; thisHashByte++){
```

```
                //find the list at the current list
                (list, bFoundHash, currentPosition) = createList([proof[2 +
thisHashGroup][thisHashByte], proof[2 + thisHashGroup][thisHashByte + 1]],
getKeyHalf(currentKeyPosition, thisKeyByte, proof), currentPosition, lastHash, proof);

                if(bFoundHash){
                    calcingHex(list);
                    lastHash = hexHash(list);
                }
                bFoundHash = false;
                currentKeyPosition++;
                if(currentKeyPosition % 2 == 0){
                    thisKeyByte++;
                }
                thisHashByte ++;
            }
        }
        return lastHash;
    }
}
```

Our constructor takes a simple hash of kaccak('balance',msg.sender) and 1,000,000 *10^18 to get our initial patricia root. According to our rules, if there is only one item in a particular layer then that hash carries to the next layer without doing a hash. As a result of our super sparse data set we end up with our initial hash sitting at the very top. I took the hash of 'balance', value so that our tree could hold a nearly infinite set of data instead of having all addresses clustered in one area of the tree. (Addresses are 20 bytes so if we just went with addresses everything would meet at the 40th level instead of at the 64th level).

Once our contract is initialized we can calculate our proof fairly easily. Just add the item to our javascript object, build the tree, and ask for the proof. If we instantiate this from our javascript vm remix and address 0xca…, we see the following New Balance event:

"args": ["0xca35b7d915458ef540ade6068dfe2f44e8fa733c",
"0xf2bfd1f27ee61a938027158a9ac3dd5051ad775b401f0230f6a574ec0d3c1c6b",
"0x0000000000000000000000000000000000000000000000d3c21bcecceda1000000"

We call
pt.addItem("0xf2bfd1f27ee61a938027158a9ac3dd5051ad775b401f0230f6a574ec0d3c1c6b","0x0000000000000000000000000000000000000000000000d3c21bcecceda1000000"), then

pt.buildTree(), then
pt.getProof("0xf2bfd1f27ee61a938027158a9ac3dd5051ad775b401f0230f6a574ec0d3c1c6b").

The proof looks as follows:

['0xf2bfd1f27ee61a938027158a9ac3dd5051ad775b401f0230f6a574ec0d3c1c6b',
'0x0000000000000000000000000000000000000000d3c21bcecceda1000000',
'0x0000000000000000000000000000000000000000000000000000000000000000',
'0x0000000000000000000000000000000000000000000000000000000000000000',
'0x0000000000000000000000000000000000000000000000000000000000000000',
'0x0000000000000000000000000000000000000000000000000000000000000000']

Notice that his is extremely sparse and that there are no hashes after the map. This is by design for a tree with only one item in it.

At this point we run into some limitations with ReMix and I'll be filing some bugs with them.

1. Our VerifySuperProof function is just too cumbersome for the remix IDE. I don't know if the JS stack gets too deep or if we are just calling too many functions, but I could not get the function to complete.
2. When I did get some functions to work, bytes32[] arrays were being interpreted in a very odd way and not being imported into the functions correctly.

Fortunately things work just fine in truffle so most of my tests were written and debugged using that setup. The following test will transfer 10 tokens from the owner to a random address until we run out of addresses in our test:

```
it "transfer tokens 2", ->
   #create a new instance
   instance = await StatelessTokenBackup.new(from: paccount[1])

   startingBalanceProof = [

web3Utils.soliditySha3(web3Utils.soliditySha3('balance'),web3Utils.padLeft(paccount[1]
,64))
     "0x0000000000000000000000000000000000000000d3c21bcecceda1000000"
     "0x0000000000000000000000000000000000000000000000000000000000000000"
     "0x0000000000000000000000000000000000000000000000000000000000000000"
     "0x0000000000000000000000000000000000000000000000000000000000000000"
     "0x0000000000000000000000000000000000000000000000000000000000000000"
   ]
```

```
    console.log 'starting Balance Proof built manually'
    console.log startingBalanceProof

    #test that the verify superProof call works
    expectedRoot = await instance.verifySuperProof.call(startingBalanceProof)
    patriciaRoot = await instance.patriciaRoot.call()

    assert.equal expectedRoot, patriciaRoot, 'roots didnt match'

    console.log "initial root:"
    console.log patriciaRoot

    console.log 'determining proofs'

    #create a new tree object
    tree = new Tree.PatriciaTree()
    tree.addItem(startingBalanceProof[0],startingBalanceProof[1])

    tree.buildTree()

    console.log 'original root'
    console.log tree.root

    #make sure the object's root matches the contract root
    assert.equal tree.root, patriciaRoot, 'roots didnt match'


    sourceBalance =
web3Utils.soliditySha3(web3Utils.soliditySha3('balance'),web3Utils.padLeft(paccount[1]
,64))

    newAmount = web3Utils.toWei("1000000","ether")
    newAmount = new web3Utils.BN(newAmount)
    removeAmount = new web3Utils.BN("10")

    #in this test we will look for items that match match at the start of the key
    #use testrpc -l 9000000 -a 1000 -e 1000000000000 to generate a bunch of addreses
    for thisAccount in [0..900]
```

```
      destbalanckey = web3Utils.soliditySha3(web3Utils.soliditySha3('balance'),
web3Utils.padLeft(paccount[100 + thisAccount],64))
      if(destbalanckey.substring(2,4) isnt sourceBalance.substring(2,4))
        continue
      startingBalanceProof = tree.getProof(web3Utils.padLeft(sourceBalance,64))
       destProof =
tree.getProof(web3Utils.soliditySha3(web3Utils.soliditySha3('balance'),
web3Utils.padLeft(paccount[100 + thisAccount],64)))
      console.log startingBalanceProof
      console.log destProof
      console.log paccount[100 + thisAccount]

      console.log 'calling super transfer'
      trx = await instance.superTransfer(paccount[100 + thisAccount],
web3Utils.toWei("10","wei"), destProof, startingBalanceProof, from: paccount[1])
      console.log 'gas used: ' + trx.receipt.gasUsed

      patriciaRoot = await instance.patriciaRoot.call()
      console.log 'updating amounts'

      newAmount = newAmount.sub(removeAmount)

      #recalc the tree

tree.addItem(sourceBalance,web3Utils.padLeft(web3Utils.numberToHex(newAmount),64))

tree.addItem(web3Utils.soliditySha3(web3Utils.soliditySha3('balance'),web3Utils.padLef
t(paccount[100 + thisAccount],64)), web3Utils.padLeft(web3Utils.numberToHex(10),64))
      tree.buildTree()

      console.log 'newRoot'
      console.log tree.root

      console.log patriciaRoot
      assert.equal tree.root, patriciaRoot, 'roots did not match'
```

If you run this you will see that we have success making transfers, but they are costing 1,000,000 gas. This is an almost comical amount considering that a normal transfer for an ERC20 token should be less than 60,000 gas.

I'd be very interested in any ideas for reducing the gas cost. I suspect that some assembly would simplify some of the loops and reduce the stack usage quite a bit.

You can also see the program outputting the proofs as it runs. For example, to transfer 10 tokens from one address to another after a few itterations we see a sender proof:

['0x6c48b84df6eee09bae1c1bcf4097733a5f9bbf5d8c21dabb29afe9b4e0a1dc90',
'0x0000000000000000000000000000000000000000d3c21bcecceda0ffffd8',
'0x0000000000000000000000000000000000000000000000000000000000000000',
'0x0000000000000000000000000000000000000000000000000000000000000000',
'0x0000000000000000000000000000000000000000000000000000000000000000',
'0x0000000000000000000000000000000000000000000000000a80a00000000',
'0x270bc792da9d645eec3a2b421604e307fce38016d84c52cbe68d033d13a659ec',
'0xba111b184f189fa213000ebd54c30dffb7905a37af3684810dc3339c644ba70c',
'0x61c01b61bf991eb361f585fee09c7f1073206642f34f3f8c4b1348f633e44203',
'0x908b6dd95f18c225b40acf351bb34a0e218137dc875d5fdd425cff881d3d32d4']

And a destination proof (notice the second(green) value is 0 because this address currently has no tokens.

['0x6c1af89d9167249debdfb2b4c1b571aa1f357b681260987f96974a37d5e3b7c6',
'0x0000000000000000000000000000000000000000000000000000000000000000',
'0x0000000000000000000000000000000000000000000000000000000000000000',
'0x0000000000000000000000000000000000000000000000000000000000000000',
'0x0000000000000000000000000000000000000000000000000000000000000000',
'0x0000000000000000000000000000000000000000000000000a80a00000000',
'0x270bc792da9d645eec3a2b421604e307fce38016d84c52cbe68d033d13a659ec',
'0xba111b184f189fa213000ebd54c30dffb7905a37af3684810dc3339c644ba70c',
'0x01b96c8979920ff903b925943df352cf10a5c4bf72fdcea2ce2b36a210dbc3fb',
'0x61c01b61bf991eb361f585fee09c7f1073206642f34f3f8c4b1348f633e44203',
'0x908b6dd95f18c225b40acf351bb34a0e218137dc875d5fdd425cff881d3d32d4']

The moral of the story isn't so much that this should be done, but that it can be done. I hope you've learned some things along the way. If you have any bright ideas on where this could be use, please let me know.

Specific areas where I think gas could be saved:

1. Looping through the map there may be some ways to load the map with much less gas and without loops.

2. Real Ethereum uses something called RLP. Maybe a solidity library exists to process this and maybe it is really efficient. I'm guessing not.