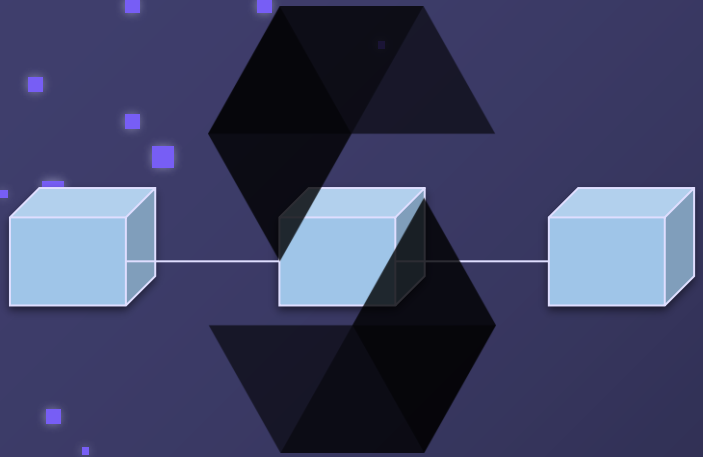


BLOCKCHAIN
LAB 2 – Linked List

Dr. Mohammed Hataba
TA: Marwa Alazab

Table of Contents

Structs

More complex data type. For this, Solidity provides structs, Structs allow you to create more complicated data types that have multiple properties.

```
struct Person {
  uint age;
  string name;
}
```

Arrays

There are two types of arrays in Solidity: fixed arrays and dynamic arrays:

```
// Array with a fixed length of 2 elements:
uint[2] fixedArray;
// another fixed Array, can contain 5 strings:
string[5] stringArray;
// a dynamic Array - has no fixed size, can keep
growing:
uint[] dynamicArray;
```

Arrays

You can also create an array of structs. Using the previous chapter's Person struct:

```
Person[] people; // dynamic Array, we can keep adding to it
```

Struct & Arrays

```
struct Person {
  uint age;
  string name;
}

Person[] public people;
```

Struct & Arrays

```
// create a New Person:
Person satoshi = Person(172, "Satoshi");

// Add that person to the Array:
people.push(satoshi);
```

Struct & Arrays

Note that array.push() adds something to the end of the array, so the elements are in the order we added them. See the following example:

```
uint[] numbers;
numbers.push(5);
numbers.push(10);
numbers.push(15);
// numbers is now equal to [5, 10, 15]
```

Functions

This is a function named eatHamburgers that takes 2 parameters: a string and a uint. For now the body of the function is empty. Note that we're specifying the function visibility as public. We're also providing instructions about where the _name variable should be stored- in memory. This is required for all reference types such as arrays, structs, mappings, and strings.

```
function eatHamburgers(string memory _name, uint _amount) public
{

}
```

_variableName

Note: It's convention (but not required) to start function parameter variable names with an underscore (_) in order to differentiate them from global variables. We'll use that convention throughout our tutorial.

Memory vs Storage

```
pragma solidity ^0.5.0;
contract HelloWorld {
  uint[5] public numbers=[1, 2, 3, 4, 5];
  function memory_working() public view returns (uint[5] memory)
  {
    uint[5] memory A = numbers;
    A[0] = 99;
    return numbers;
     }
  function storage_working() public returns (uint[5] memory)
  {
    uint[5] storage B = numbers;
    B[0] = 0;
    return numbers;
  }
}
```

Visibility

```
string name1 = “Name 1”;

string private name2 = “Name2”;

string internal name3 = “Name3”;

string public name4 = “Name4”;
```

Visibility

Functions are public by default. This means anyone (or any other contract) can call your contract's function and execute its code.

```
uint[] numbers;

function _addToArray(uint _number) private {
  numbers.push(_number);
}
```

_functionName

As you can see, we use the keyword private after the function name. And as with function parameters, it's convention to start private function names with an underscore (_).

Variables

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;
contract Variables {
    // State variables are stored on the blockchain.
    string public text = "Hello";
    uint public num = 123;
    function doSomething() public {
        // Local variables are not saved to the blockchain.
        uint i = 456;
        // Here are some global variables
        uint timestamp = block.timestamp; // Current block timestamp
        address sender = msg.sender; // address of the caller
    }
}
```

Pure Vs view

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;
contract ViewAndPure {
    uint public x = 1;
    // Promise not to modify the state.
    function addToX(uint y) public view returns (uint) {
        return x + y;
    }
    // Promise not to modify or read from the state.
    function add(uint i, uint j) public pure returns (uint) {
        return i + j;
    }
}
```

Mapping

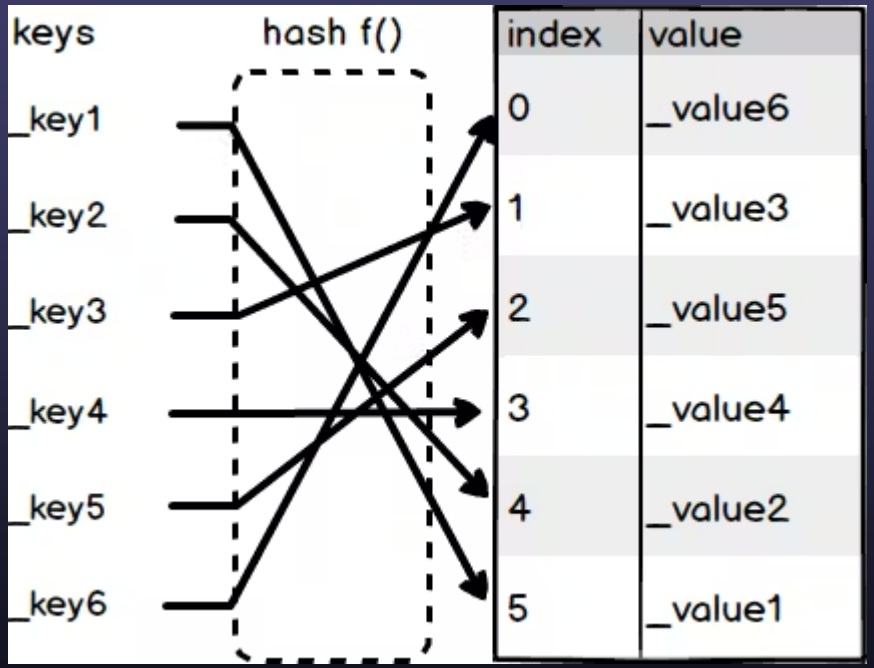
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;
contract Mapping {
    // Mapping from address to uint
    mapping(address => uint) public myMap;
    function get(address _addr) public view returns (uint) {
        // Mapping always returns a value.
        // If the value was never set, it will return the default value.
        return myMap[_addr];
    }
    function set(address _addr, uint _i) public {
        // Update the value at this address
        myMap[_addr] = _i;
    }
    function remove(address _addr) public {
        // Reset the value to the default value.
        delete myMap[_addr];
    }
}
```

Mapping

Maps are created with the syntax mapping(keyType => valueType).

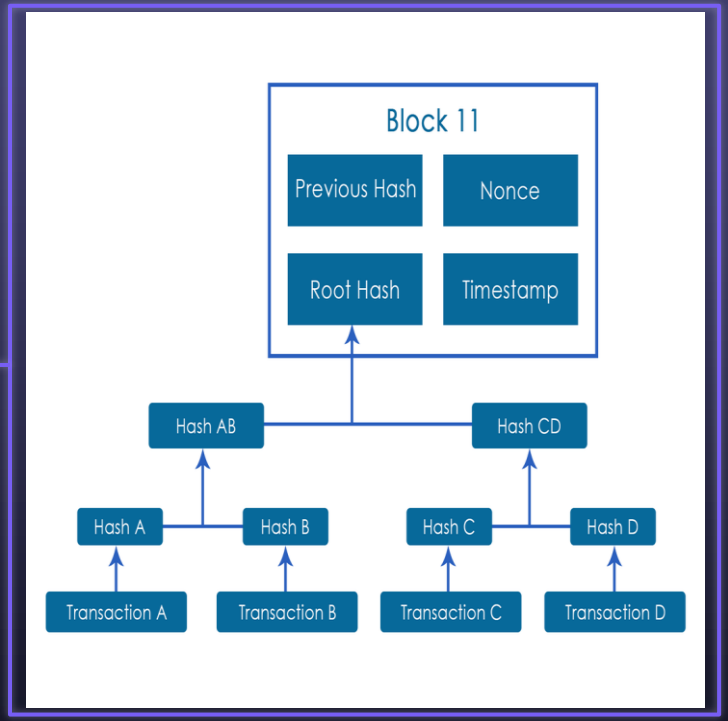
The keyType can be any built-in value type, bytes, string, or any contract.

valueType can be any type including another mapping or an array.



Data Structure

Linked List, Merkle Tree, Hash Functions



Task

Linked List

A linked list is a linear data structure that includes a series of connected nodes. Here, each node stores the data and the address of the next node

Steps

Define Node structure
Define the head
Insertion function
Retrieving function
Linked List Size

Resources

- https://mlsdev.com/blog/156-how-to-build-your-own-blockchain-architecture
- https://www.youtube.com/watch?v=NuyzuNBFWxQ
- https://emn178.github.io/online-tools/sha256.html
- https://app.cadena.dev/course/ethereum-101/ZHjzLozd3mCsAcgMfeHE
- https://www.youtube.com/watch?v=hMwdd664_iw&list=PLO5VPQH6OWdULDcret0S0EYQ7YcKzrigz&index=1
- https://solidity-by-example.org/hello-world/
- https://medium.com/@solidity101/100daysofsolidity-understanding-view-and-pure-functions-in-solidity-18-eec8057d9b97#:~:text=View%20functions%20in%20Solidity%20are,to%20the%20contract's%20state%20variables.