Anuprita Mhapankar **D20A** **34**

Blockchain Experiment 5

AIM: Deploying a Voting/Ballot Smart Contract

THEORY:

Q1: What is the relevance of required statements in the functions of Solidity Programs?

In Solidity, the required statement acts as a **guard condition** within functions. It ensures that only valid inputs or authorized users can execute certain parts of the code. If the condition inside require is not satisfied, the function execution stops immediately, and all state changes made during that transaction are reverted to their original state. This rollback mechanism ensures that invalid transactions do not corrupt the blockchain data.

For example, in a **Voting Smart Contract**, require can be used to check:

- Whether the person calling the function has the right to vote (require(voters[msg.sender].weight > 0, "Has no right to vote");).
- Whether a voter has already voted before allowing them to vote again.
- Whether the function caller is the chairperson before granting voting rights.

Thus, require statements enforce security, correctness, and reliability in smart contracts. They also allow developers to attach error messages, making debugging and contract interaction easier for users.

Q2: Understand the keywords mapping, storage and memory

- **mapping:**
  A mapping is a special data structure in Solidity that links keys to values, similar to a hash table. Its syntax is mapping(keyType => valueType).

  For example: mapping(address => Voter) public voters;

  Here, each address (Ethereum account) is mapped to a Voter structure. Mappings are very useful for contracts like Ballot, where you need to associate voters with their data (whether they voted, which proposal they chose, etc.). Unlike arrays, mappings do not have a length property and cannot be iterated over directly, making them **gas efficient** for lookups but limited for enumeration.

- **storage:**
  In Solidity, storage refers to the **permanent memory** of the contract, stored on the Ethereum blockchain. Variables declared at the contract level are stored in storage by default. Data stored in storage is persistent across transactions, which means once written, it remains available unless explicitly modified. However, because writing to blockchain storage consumes gas, it is more expensive. For example, a voter's information saved in the voters mapping remains available throughout the contract's lifecycle.

- **memory:**
  In contrast, memory is **temporary storage**, used only for the lifetime of a function call. When the function execution ends, the data stored in memory is discarded. Memory is mainly used for temporary variables, function arguments, or computations that don't

need to be permanently stored on the blockchain. It is cheaper than storage in terms of gas cost. For instance, when handling proposal names or temporary string manipulations, memory is often used.

Thus, a smart contract developer must balance between storage and memory to ensure efficiency and cost-effectiveness.

Q3: Why bytes32 instead of string?

In earlier implementations of the Ballot contract, bytes32 was used for proposal names instead of string. The reason lies in efficiency and gas optimization.

- **bytes32 is a fixed-size type**, meaning it always stores exactly 32 bytes of data. This makes storage simple, comparison operations faster, and gas costs lower. However, it limits proposal names to 32 characters, which is not very flexible for user-friendly names.
- **string is a dynamically sized type**, meaning it can store text of variable length. While it is easier for developers and users (since names can be written normally), it requires more complex handling inside the Ethereum Virtual Machine (EVM). This increases gas usage and may slow down comparisons or manipulations.

To make the system more user-friendly, modern implementations of the Ballot contract often convert from bytes32 to string. Tools like the Web3 Type Converter help developers easily switch between these two types for deployment and testing.

In summary, bytes32 is used when performance and gas efficiency are priorities, while string is preferred for readability and ease of use.

CODE:

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

/**
 * @title Ballot
 * @dev Implements voting process along with vote delegation
 */
contract Ballot {
    // This declares a new complex type which will
    // be used for variables later.
    // It will represent a single voter.
    struct Voter {
        uint weight; // weight is accumulated by delegation
        bool voted;  // if true, that person already voted
        address delegate; // person delegated to
        uint vote;   // index of the voted proposal
    }
```

```
    // This is a type for a single proposal.
    struct Proposal {
        // If you can limit the length to a certain number of bytes,
        // always use one of bytes1 to bytes32 because they are much
cheaper
        string name;
        uint voteCount; // number of accumulated votes
    }

    address public chairperson;

    // This declares a state variable that
    // stores a 'Voter' struct for each possible address.
    mapping(address => Voter) public voters;

    // A dynamically-sized array of 'Proposal' structs.
    Proposal[] public proposals;

    /**
     * @dev Create a new ballot to choose one of 'proposalNames'.
     * @param proposalNames names of proposals
     */
    constructor(string[] memory proposalNames) {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        // For each of the provided proposal names,
        // create a new proposal object and add it
        // to the end of the array.
        for (uint i = 0; i < proposalNames.length; i++) {
            // 'Proposal({...})' creates a temporary
            // Proposal object and 'proposals.push(...)'
            // appends it to the end of 'proposals'.
            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
            }));
        }
    }
```

```
     /**
     * @dev Give 'voter' the right to vote on this ballot. May only be
called by 'chairperson'.
     * @param voter address of voter
     */
    function giveRightToVote(address voter) external {
        // If the first argument of `require` evaluates
        // to 'false', execution terminates and all
        // changes to the state and to Ether balances
        // are reverted.
        // This used to consume all gas in old EVM versions, but
        // not anymore.
        // It is often a good idea to use 'require' to check if
        // functions are called correctly.
        // As a second argument, you can also provide an
        // explanation about what went wrong.
        require(
            msg.sender == chairperson,
            "Only chairperson can give right to vote."
        );
        require(
            !voters[voter].voted,
            "The voter already voted."
        );
        require(voters[voter].weight == 0, "Voter already has the right to
vote.");
        voters[voter].weight = 1;
    }

    /**
     * @dev Delegate your vote to the voter 'to'.
     * @param to address to which vote is delegated
     */
    function delegate(address to) external {
        // assigns reference
        Voter storage sender = voters[msg.sender];
        require(sender.weight != 0, "You have no right to vote");
        require(!sender.voted, "You already voted.");

        require(to != msg.sender, "Self-delegation is disallowed.");
```

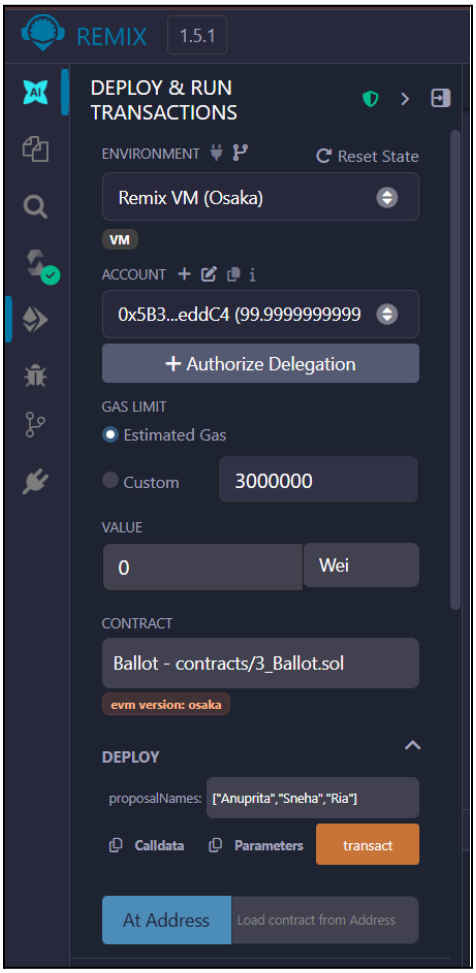
```
        // Forward the delegation as long as
        // 'to' also delegated.
        // In general, such loops are very dangerous,
        // because if they run too long, they might
        // need more gas than is available in a block.
        // In this case, the delegation will not be executed,
        // but in other situations, such loops might
        // cause a contract to get "stuck" completely.
        while (voters[to].delegate != address(0)) {
            to = voters[to].delegate;

            // We found a loop in the delegation, not allowed.
            require(to != msg.sender, "Found loop in delegation.");
        }

        Voter storage delegate_ = voters[to];

        // Voters cannot delegate to accounts that cannot vote.
        require(delegate_.weight >= 1);

        // Since 'sender' is a reference, this
        // modifies 'voters[msg.sender]'.
        sender.voted = true;
        sender.delegate = to;

        if (delegate_.voted) {
            // If the delegate already voted,
            // directly add to the number of votes
            proposals[delegate_.vote].voteCount += sender.weight;
        } else {
            // If the delegate did not vote yet,
            // add to her weight.
            delegate_.weight += sender.weight;
        }
    }

    /**
     * @dev Give your vote (including votes delegated to you) to proposal
'proposals[proposal].name'.
```

```
     * @param proposal index of proposal in the proposals array
     */
    function vote(uint proposal) external {
        Voter storage sender = voters[msg.sender];
        require(sender.weight != 0, "Has no right to vote");
        require(!sender.voted, "Already voted.");
        sender.voted = true;
        sender.vote = proposal;

        // If 'proposal' is out of the range of the array,
        // this will throw automatically and revert all
        // changes.
        proposals[proposal].voteCount += sender.weight;
    }

    /**
     * @dev Computes the winning proposal taking all previous votes into
account.
     * @return winningProposal_ index of winning proposal in the proposals
array
     */
    function winningProposal() public view
            returns (uint winningProposal_)
    {
        uint winningVoteCount = 0;
        for (uint p = 0; p < proposals.length; p++) {
            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
                winningProposal_ = p;
            }
        }
    }

    /**
     * @dev Calls winningProposal() function to get the index of the
winner contained in the proposals array and then
     * @return winnerName_ the name of the winner
     */
    function winnerName() external view
            returns (string memory winnerName_)
```

```
    {
        winnerName_ = proposals[winningProposal()].name;
    }
}
```

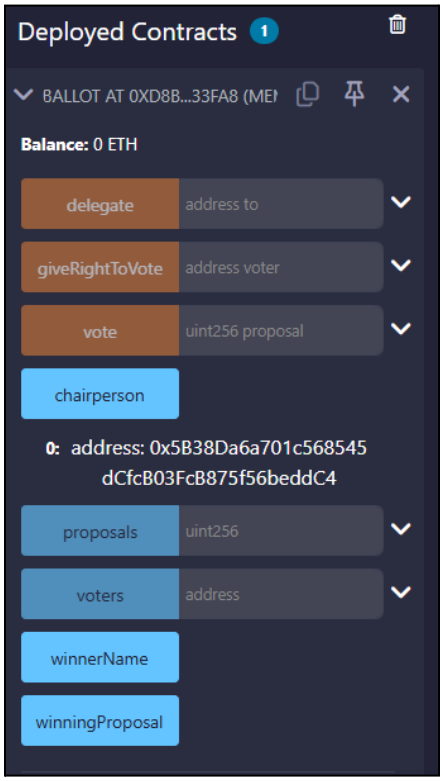
Step 1: Deploy Contract

3 proposals are created “Anuprita”, “Sneha” and “Ria” all with vote count 0.



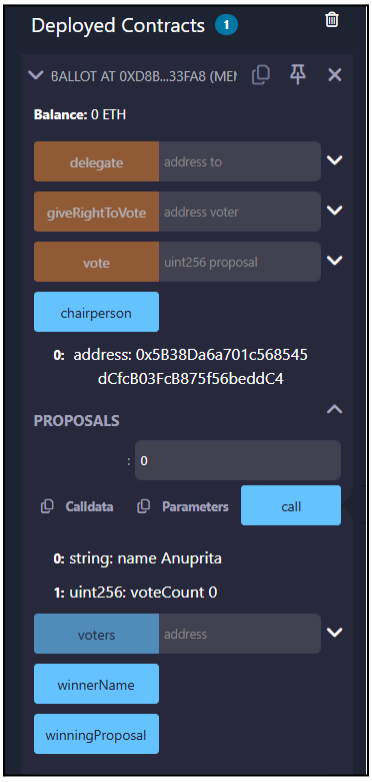
Step 2: Check Chairperson

Here, Account 1 - “Anuprita” is the chairperson.

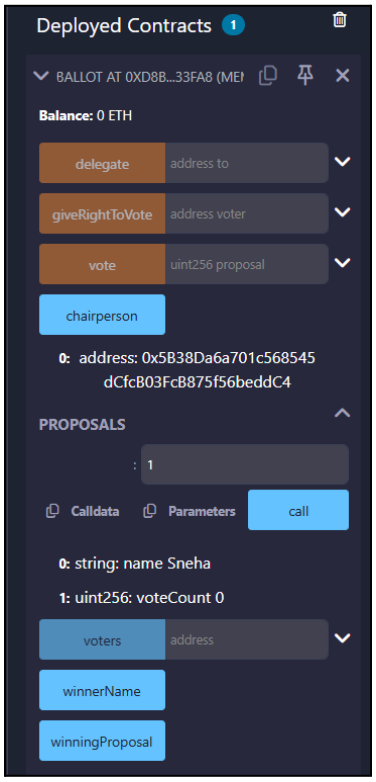


Step 3: Check proposals

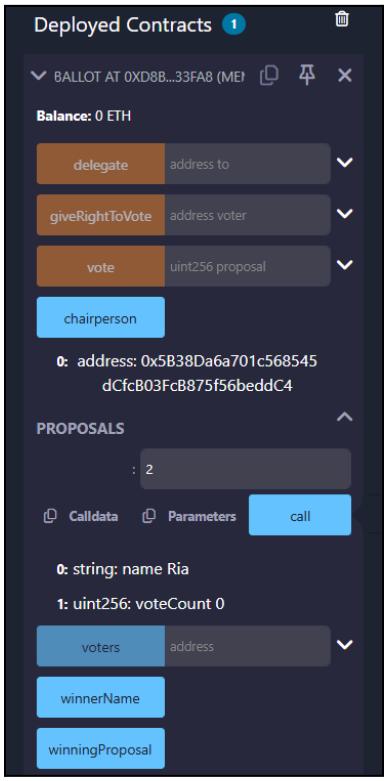
For Account 1- “Anuprita”



For Account 2 - “Sneha”

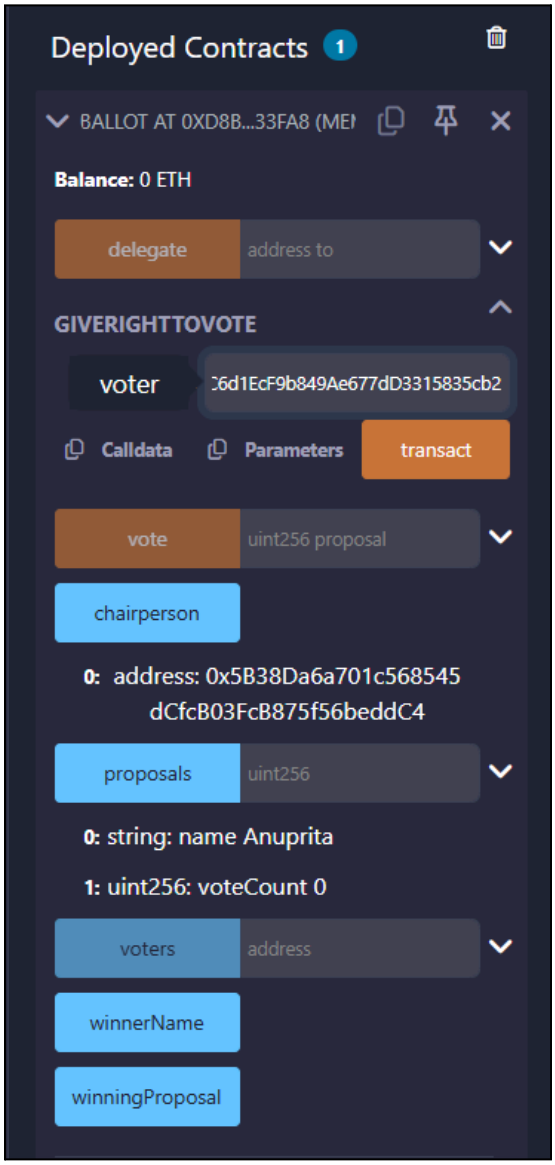


For Account 3 - “Ria”



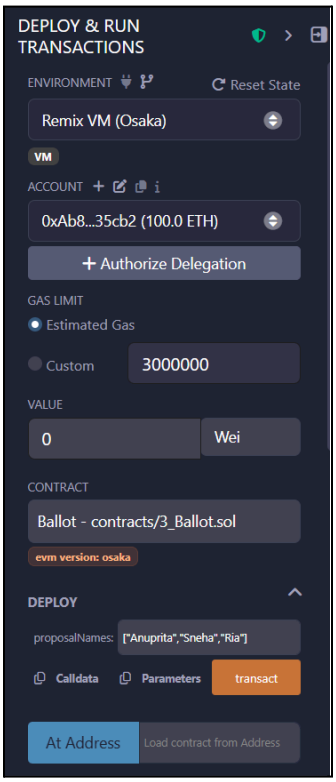
Step 4: Give voting right to another account

Keep account 1 - Anuprita selected and in **giveRightToVote(address voter)** paste the address of Account 2 - “Sneha” and click on the transact button.

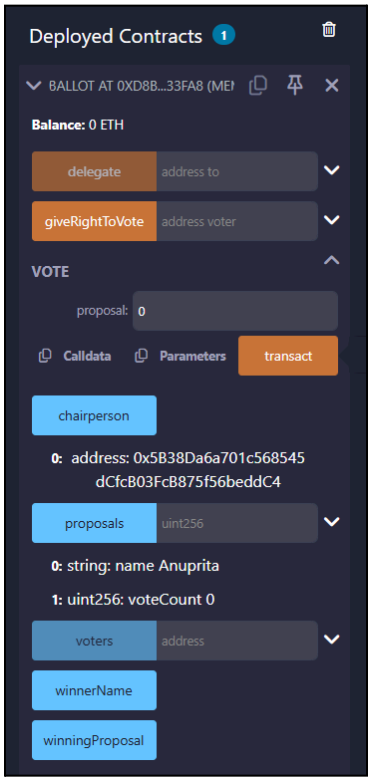


Step 5: Switch to Account 2

From Account dropdown select Account 2 - “Sneha”

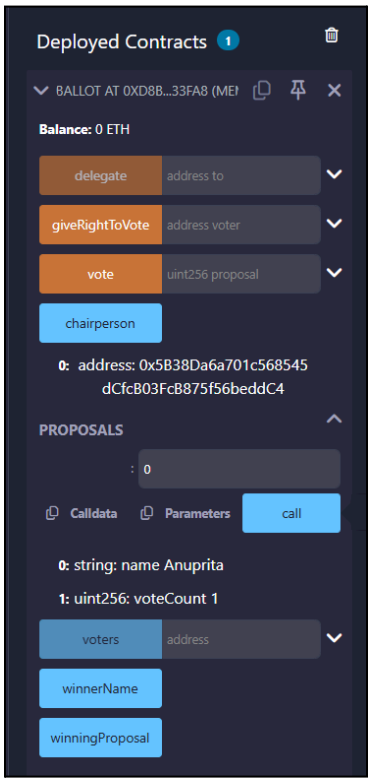


Now, vote for Account 1 - “Anuprita” by **vote(0)**.



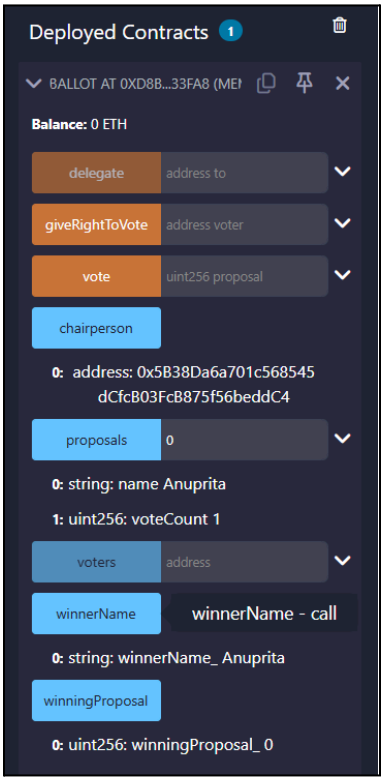
Step 6: Check Proposal

Here, the vote count for Account 1 - “Anuprita” increases.



Step 7: Check winner

Here, Account 1 - “Anuprita” is the winner as it has the highest votes.



CONCLUSION:

Through this experiment, the Voting/Ballot smart contract was successfully deployed using Solidity in the Remix IDE. Important concepts such as require statements, mapping, and data locations like storage and memory were understood while performing the contract execution. The difference between using bytes32 and string for proposal names was also studied, which helped in understanding gas efficiency and readability. Overall, this experiment helped in gaining practical knowledge about designing and deploying a voting smart contract on the blockchain.