Prathamesh Palve **D20A/Batch-A/02**

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:

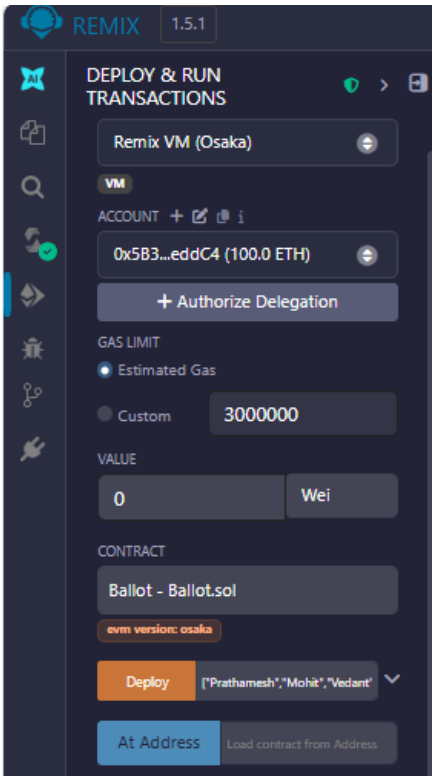
```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Ballot {

    struct Voter {
        uint weight;
        bool voted;
        address delegate;
        uint vote;
    }

    struct Proposal {
        string name;     // Changed from bytes32 to string
        uint voteCount;
    }

    address public chairperson;
    mapping(address => Voter) public voters;
    Proposal[] public proposals;
```

```
    constructor(string[] memory proposalNames) {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        for (uint i = 0; i < proposalNames.length; i++) {
            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
            }));
        }
    }

    function giveRightToVote(address voter) external {
        require(msg.sender == chairperson, "Only chairperson can give
right.");
        require(!voters[voter].voted, "Already voted.");
        require(voters[voter].weight == 0);

        voters[voter].weight = 1;
    }

    function vote(uint proposal) external {
        Voter storage sender = voters[msg.sender];

        require(sender.weight != 0, "No right to vote");
        require(!sender.voted, "Already voted");

        sender.voted = true;
        sender.vote = proposal;

        proposals[proposal].voteCount += sender.weight;
    }

    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;
            }
        }
    }
```

```
    function winnerName() external view returns (string memory) {
        return proposals[winningProposal()].name;
    }

    //  Additional Functionality
    function addProposal(string memory name) external {
        require(msg.sender == chairperson, "Only chairperson can add
proposal");

        proposals.push(Proposal({
            name: name,
            voteCount: 0
        }));
    }

    function getProposalVoteCount(uint proposal) external view returns
(uint) {
        return proposals[proposal].voteCount;
    }
}
```

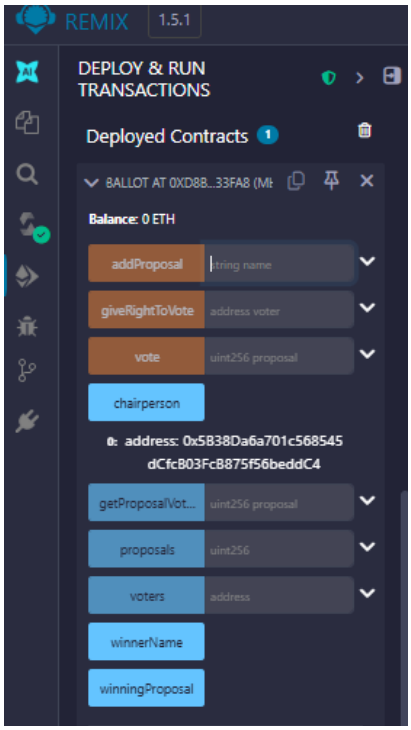
Step 1: Deploy Contract

3 proposals are created"Prathamesh","Mohit" and "Vedant" all with vote count 0.



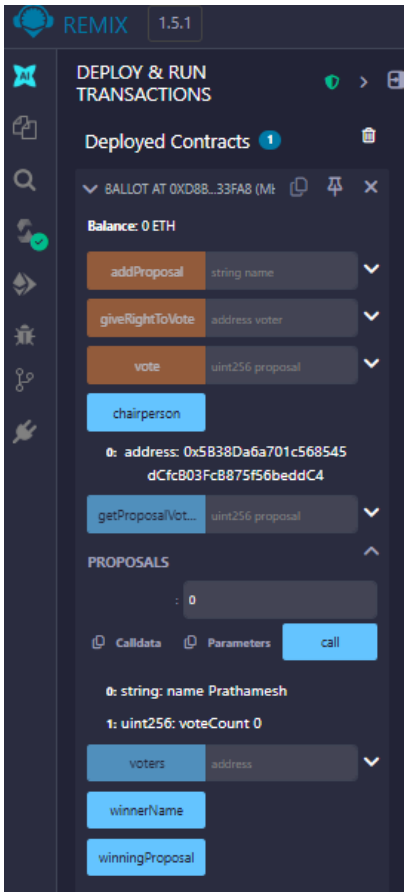
Step 2: Check Chairperson

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

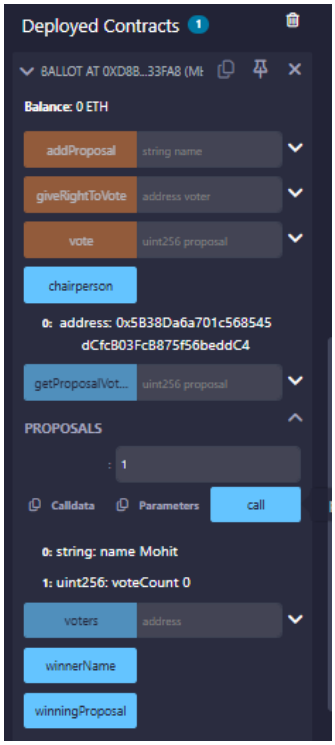


Step 3: Check proposals

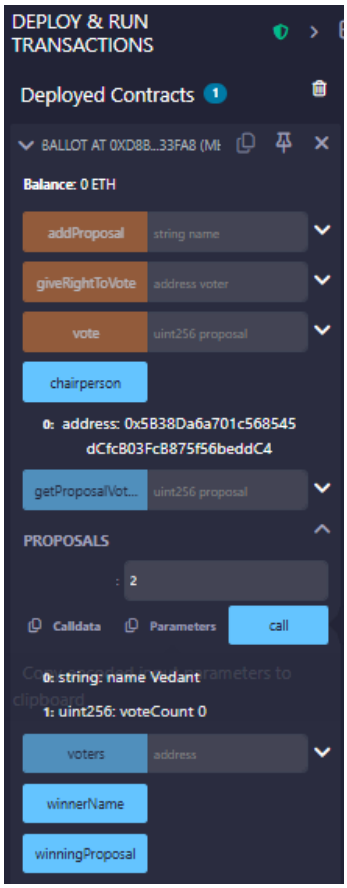
For Account 1- “Prathamesh”



For Account 2 - “Mohit”

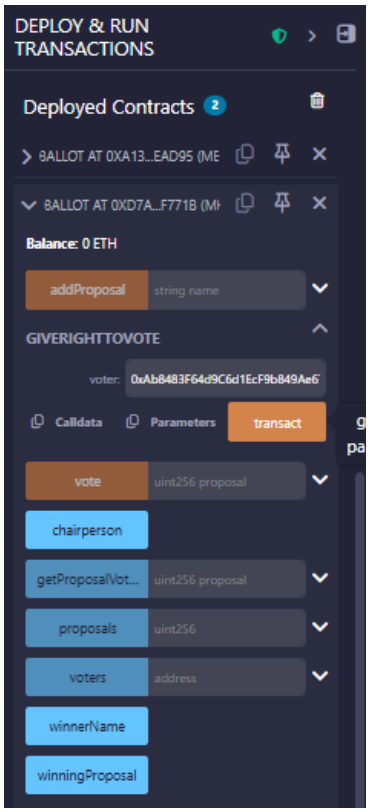


For Account 3 - “Vedant”



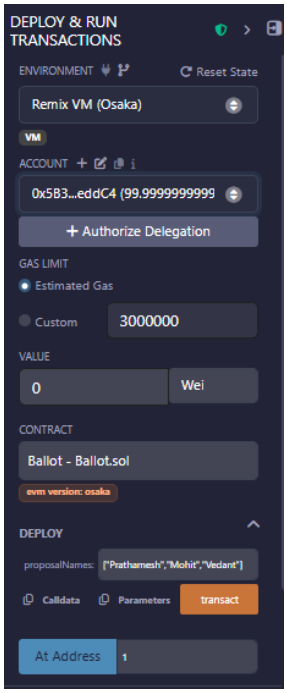
Step 4: Give voting right to another account

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

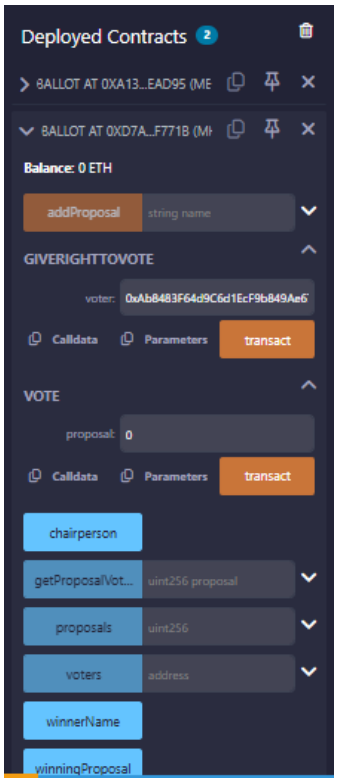


Step 5: Switch to Account 2

From Account dropdown select Account 2 - “Mohit”

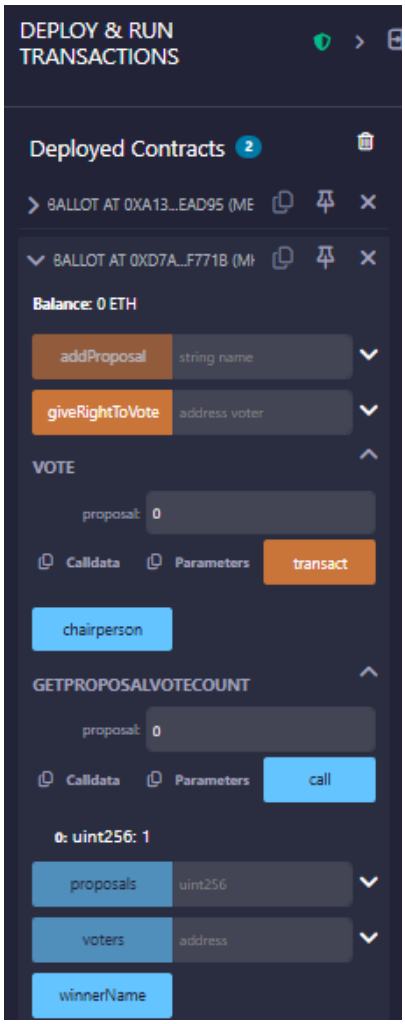


Now, vote for Account 1 - “Prathamesh” by **vote(0)**.



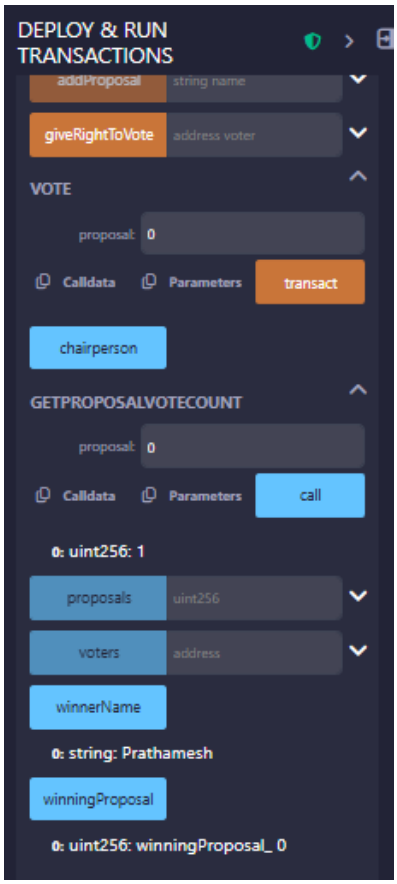
Step 6: Check Proposal

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



Step 7: Check winner

Here, Account 1 - “Prathamesh” 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.