**Aim:** Write a program in solidity to create Student data. Use the following constructs:

Structures

Arrays

Fallback

Deploy this as smart contract on Ethereum and Observe the transaction fee and Gas values.

**Objectives:** Understand and explore the working of blockchain technology and its application.

**Course Outcome:** Interpret the basic concepts in blockchain technology and its application.

**Theory:**

**Struct**

Structs in Solidity allows you to create more complicated data types that have multiple properties. You can define your own type by creating a struct. They are useful for grouping together related data. Structs can be declared outside of a contract and imported in another contract. Generally, it is used to represent a record. To define a structure struct keyword is used, which creates a new data type.

**Syntax:**

```
struct <structure_name> {
   <data type> variable_1;
   <data type> variable_2;
}
```

For accessing any element of the structure, 'dot operator' is used, which separates the struct variable and the element we wish to access. To define the variable of structure data type structure name is used.

**2. Arrays**

Arrays are data structures that store the fixed collection of elements of the same data types in which each and every element has a specific location called index. Instead of creating numerous individual variables of the same type, we just declare one array of the required size and store the elements in the array and can be accessed using the index. In Solidity, an array can be of fixed

size or dynamic size. Arrays have a continuous memory location, where the lowest index corresponds to the first element while the highest represents the last.

**Creating an Array**

To declare an array in Solidity, the data type of the elements and the number of elements should be specified. The size of the array must be a positive integer and data type should be a valid Solidity type

**Syntax:**

<data type> <array name>[size] = <initialization>

**Fixed-size Arrays**

The size of the array should be predefined. The total number of elements should not exceed the size of the array. If the size of the array is not specified then the array of enough size is created which is enough to hold the initialization.

Example: In the below example, the *contract* Types are created to demonstrate how to declare and initialize fixed-size arrays.

```
// Solidity program to demonstrate
// creating a fixed-size array
pragma solidity ^0.5.0;

// Creating a contract
contract Types {

    // Declaring state variables
    // of type array
    uint[6] data1;

    // Defining function to add
    // values to an array
```

```
    function array_example() public returns (
    int[5] memory, uint[6] memory){

        int[5] memory data
        = [int(50), -63, 77, -28, 90];
        data1
        = [uint(10), 20, 30, 40, 50, 60];

        return (data, data1);
}
}
```

**Dynamic Array:**

The size of the array is not predefined when it is declared. As the elements are added the size of array changes and at the runtime, the size of the array will be determined.

Example: In the below example, the *contract* Types are created to demonstrate how to create and initialize dynamic arrays.

```
// Solidity program to demonstrate
// creating a dynamic array
pragma solidity ^0.5.0;

// Creating a contract
contract Types {

    // Declaring state variable
    // of type array. One is fixed-size
    // and the other is dynamic array
    uint[] data
    = [10, 20, 30, 40, 50];
    int[] data1;
```

```
    // Defining function to
    // assign values to dynamic array
    function dynamic_array() public returns(
    uint[] memory, int[] memory){

        data1
        = [int(-60), 70, -80, 90, -100, -120, 140];
        return (data, data1);
    }
}
```

**Push:**

Push is used when a new element is to be added in a dynamic array. The new element is always added at the last position of the array.

Example: In the below example, the contract Types first initializes an array[data], and then more values are pushed into the array.

```
// Solidity program to demonstrate
// Push operation
pragma solidity ^0.5.0;
// Creating a contract
contract Types {
    // Defining the array
    uint[] data = [10, 20, 30, 40, 50];
        // Defining the function to push
    // values to the array
    function array_push( ) public returns(uint[] memory){
        data.push(60);
        data.push(70);
        data.push(80);
        return data;
    }
}
```

**Fall Back Function**

The solidity fallback function is executed if none of the other functions match the function identifier or no data was provided with the function call. Only one unnamed function can be assigned to a contract and it is executed whenever the contract receives plain Ether without any data. To receive Ether and add it to the total balance of the contract, the fallback function must be marked payable. **If no such function exists, the contract cannot receive Ether through regular transactions and will throw an exception.**

**Properties of a fallback function:**

1. Has no name or arguments.
2. If it is not marked **payable**, the contract will throw an exception if it receives plain ether without data.
3. Can not return anything.
4. Can be defined once per contract.
5. It is also executed if the caller meant to call a function that is not available
6. It is mandatory to mark it external.
7. It is limited to 2300 gas when called by another function. It is so for as to make this function call as cheap as possible.

**Example:** In the below example, the Contract is created to demonstrate different conditions for different fallback function.

```
pragma solidity ^0.4.0;

// Creating a contract
contract GeeksForGeeks
{
	// Declaring the state variable
	uint x;

	// Mapping of addresses to their balances
	mapping(address => uint) balance;

	// Creating a constructor
	constructor() public
	{
		// Set x to default
		// value of 10
```

```
        x=10;

    }

    // Creating a function
    function SetX(uint _x) public returns(bool)
    {
        // Set x to the
        // value sent
        x=_x;
        return true;
    }

    // This fallback function
    // will keep all the Ether
    function() public payable
    {
        balance[msg.sender] += msg.value;
    }
}

// Creating the sender contract
contract Sender
{
function transfer() public payable
{
    // Address of GeeksForGeeks contract
    address _receiver =
                0xbcc0185441de06F0452D45AEd6Ad8b98017796fb;

    // Transfers 100 Eth to above contract
    _receiver.transfer(100);
}
}
```

**Explanation:**

**1. Contract GeeksForGeeks:** It has a variable x which is set to the default value 10 in the constructor(). The contract has a function called SetX(uint _x) which sets the function value to the desired parameter sent during the function call. The below declaration creates address to value map called balance which maps the addresses to their balance.

mapping(address => uint) balance;

**2. Contract Sender:** This is a completely independent and unrelated contract. It sends a value in Ether to the contract GeeksForGeeks. The contract does not know the mechanism of the

GeeksForGeeks contract. Sending a transaction without any message and only Ether can cause an error.

The below statements declare a variable _receiver of the address type. It explicitly stores the address of contract GeeksForGeeks. It then uses address.transfer(value) to transfer Ether to the contract.

```
address _receiver = 0xbcc0185441de06F0452D45AEd6Ad8b98017796fb;
//Address of GeeksForGeeks contract
_receiver.transfer(100);
```

**3. Function() public payable:**

The function below is a fallback function. It is declared payable which allows it to accept transfer value. It is called in two cases

A contract receives only Ether and no data.

No function calls matched even though the account received data.

This helps us to protect the function from throwing an error. In this program, the contract GeeksForGeeks receives only Ether and the fallback function uses the value received to add to the balance related to the sending address.

```
function() public payable
    {
        balance[msg.sender] += msg.value;
    }
```

**Implementation:**

**Write a program in solidity to create Student data. Use the following constructs:**

**Structures**

**Arrays**

**Fallback**

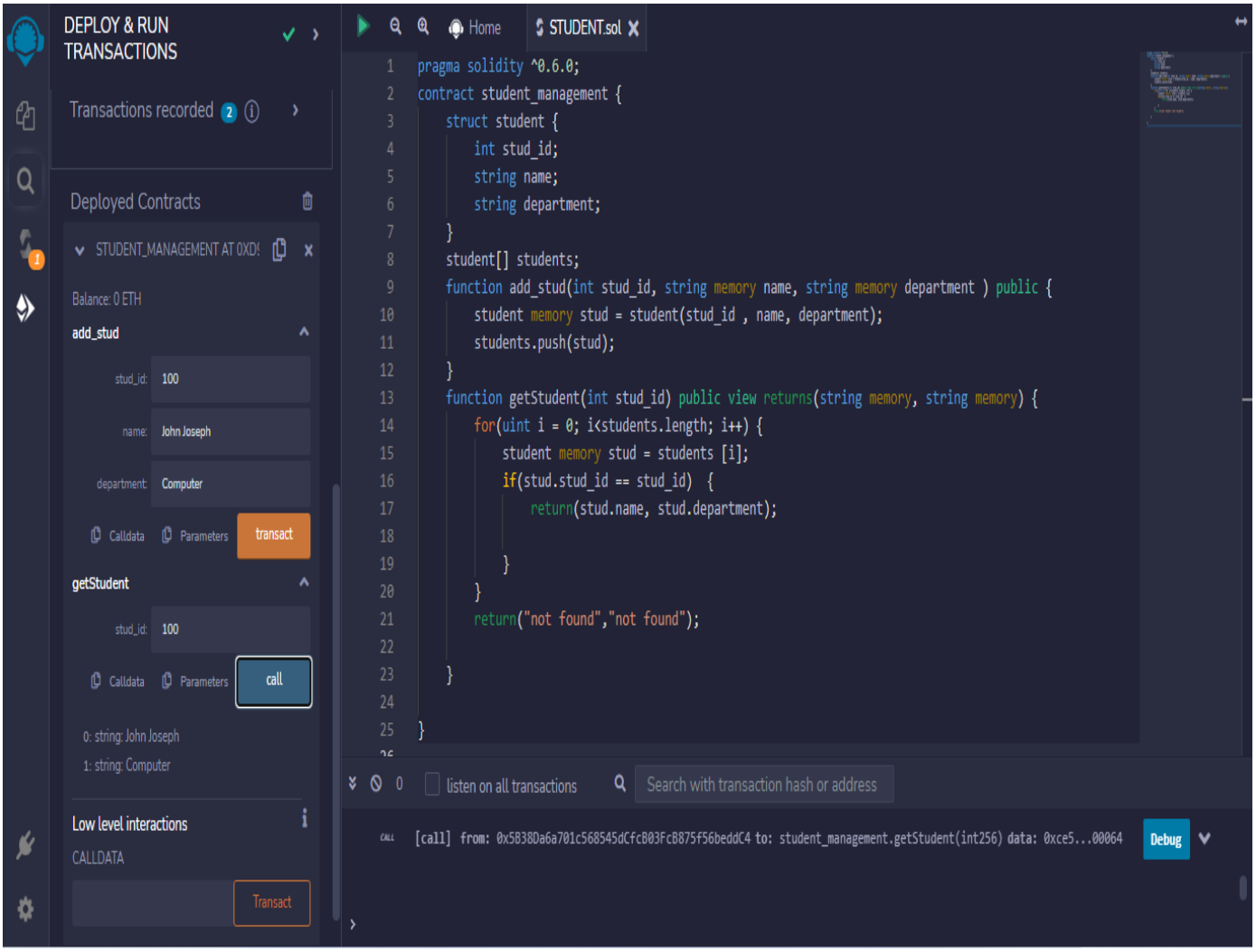
**Deploy this as smart contract on Ethereum and Observe the transaction fee and Gas values.**

**//code**

```
pragma solidity ^0.6.0;
contract student_management {
    struct student {
        int stud_id;
```

```
        string name;
        string department;
    }
    student[] students;
    function add_stud(int stud_id, string memory name, string memory department ) public {
        student memory stud = student(stud_id , name, department);
        students.push(stud);
    }
    function getStudent(int stud_id) public view returns(string memory, string memory) {
        for(uint i = 0; i<students.length; i++) {
            student memory stud = students [i];
            if(stud.stud_id == stud_id)  {
                return(stud.name, stud.department);
            }
        }
        return("not found","not found");
    }
}
```

**OUTPUT:**




**Conclusion:** In this way we have successfully implemented program in solidity to create Student data using constructs Structures, Arrays & Fallback.