# Test cases and their Explanations

## 1.1 Should Add data

**Scenario:** Tests whether the `create` function successfully adds data to the contract's storage.

**Steps:**

1. Deploys the Manager contract.
2. Calls the `create` function with a new account address and a description.
3. Retrieves the stored data for the provided account.

**Expected Outcome**

1. The stored user address should match the provided account address.
2. The stored description should match the provided description.

```
it("should Add data", async function () {
  const { manager, owner, otherAccount } = await loadFixture(deployManager);
  manager.create(otherAccount, "description test")
  let data = await manager.entriesStore(otherAccount);
  expect(data.user).to.equal(otherAccount);
  expect(data.descrition).to.equal("description test");
});
```

## 1.2 Should Update data

**Scenario:** Tests whether the `update` function successfully updates existing data in the contract's storage.

**Steps:**

1. Deploys the Manager contract.
2. Calls the `create` function to add initial data.
3. Calls the `update` function with the same account address but a different description.
4. Retrieves the updated data for the provided account.

**Expected Outcome:**

1. The stored user address should remain unchanged.
2. The stored description should reflect the updated description.

```
it("should Update data", async function () {
  const { manager, owner, otherAccount } = await loadFixture(deployManager);
  manager.create(otherAccount, "description test")
  manager.update(otherAccount, "description Update")
  let data = await manager.entriesStore(otherAccount);
  expect(data.user).to.equal(otherAccount);
  expect(data.descrition).to.equal("description Update");
});
```

### 1.3 Should Remove data

**Scenario:** Tests whether the `remove` function successfully removes data from the contract's storage.

**Steps:**

1. Deploys the Manager contract.
2. Calls the `create` function to add data.
3. Retrieves the stored data for the provided account.
4. Calls the `remove` function to delete the data.

**Expected Outcome:**

1. After removal, the stored user address should be empty.

```
it("should Remove data", async function () {
  const { manager, owner, otherAccount } = await loadFixture(deployManager);
  manager.create(otherAccount, "description Remove")
  let data = await manager.entriesStore(otherAccount);
  await manager.remove(otherAccount);
  expect(await data.user).to.equal('0x');
});
```

### 1.4 Should not update non-existing data

**Scenario:** Tests whether attempting to update non-existing data results in a revert.

**Steps:**

1. Deploys the Manager contract.
2. Attempts to call the `update` function with an account that does not have existing data.

**Expected Outcome:**

1. The function call should revert with an error message indicating that the data with the given ID does not exist.

### 1.5 Should not delete non-existing data

**Scenario:** Tests whether attempting to delete non-existing data results in a revert.

**Steps:**

1. Deploys the Manager contract.
2. Attempts to call the remove function with an account that does not have existing data.

**Expected Outcome:**

1. The function call should revert with an error message indicating that the data with the given ID does not exist.