Solidity CRUD- Part 2

Data Storage With Sequential Access, Random Access and Delete

Author: Rob Hitchens
Date: February, 2017

In part 1 of this series, we created an internal storage structure for table-like data in Solidity contracts. If you haven't reviewed part 1, you should do so now to familiarize yourself with the basic structure because it supports the delete operation we'll explore now.

The part 1 example contract includes a system of pointers and a simple (one-liner) function that checks the existence/non-existence of a key identifier. The example allows us to create, retrieve and update records in a table-like structure. It allow us to:

1) Insert a record with a key identifier
2) Retrieve a record by its key identifier
3) Update a record using its key identifier and new value
4) Obtain a count of the records that exist
5) Obtain a list of the keys in the system suitable for iterating over the keys
6) Check the existence of a key

For our delete operation, we require only a function that allows us to:

1) Remove a key

Our interface only needs to know the key to remove:

```
function deleteUser(address userAddress) public returns(bool success)
{}
```

We generally don't need to zero out data because we will accomplish our goal by logically removing Keys from our unordered list, called `userIndex`.

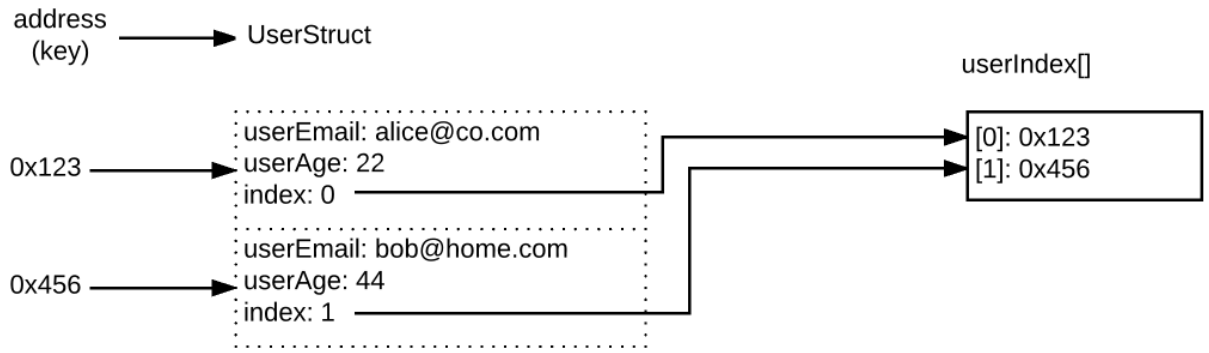
Wait. How can we know what row a given `userAddress` lives on in an unordered list?

A solution to this problem is to store the index row numbers in the mapped user structures because *we can look up that information using key values*. The stored structures should point to their respective index rows so we don't have a problem pinpointing index rows when we need them:

```
struct UserStruct {
      bytes32 userEmail;
      uint userAge;
      uint index; // the corresponding row number in the index
}
```

```
mapping (address => UserStruct) userStructs;

address[] userIndex; // unordered list of keys that actually exist
```



Great. As long as we record the index rows as we insert the records, we'll have no difficulty knowing what row we need to delete from the index when we want to remove a given key. Luckily, that's exactly what we started doing in Part 1, so this is all set up.

Wait. If we want to remove an array item from a random location in a large array, won't that require a huge reorganization (write, write, write = gas, gas, gas)? It will indeed, if you decide to move everything around. Even worse: that approach won't scale.

There's a solution to reorganizing this *unordered* list*:*

Consider this original list:

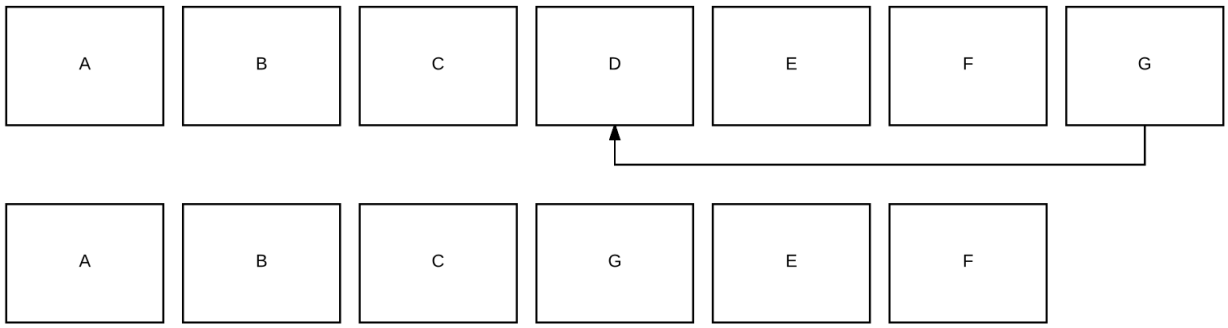
A B C D E F G

Suppose we want to delete "D".

We can reorganize the *unordered* index like this *in one move:*

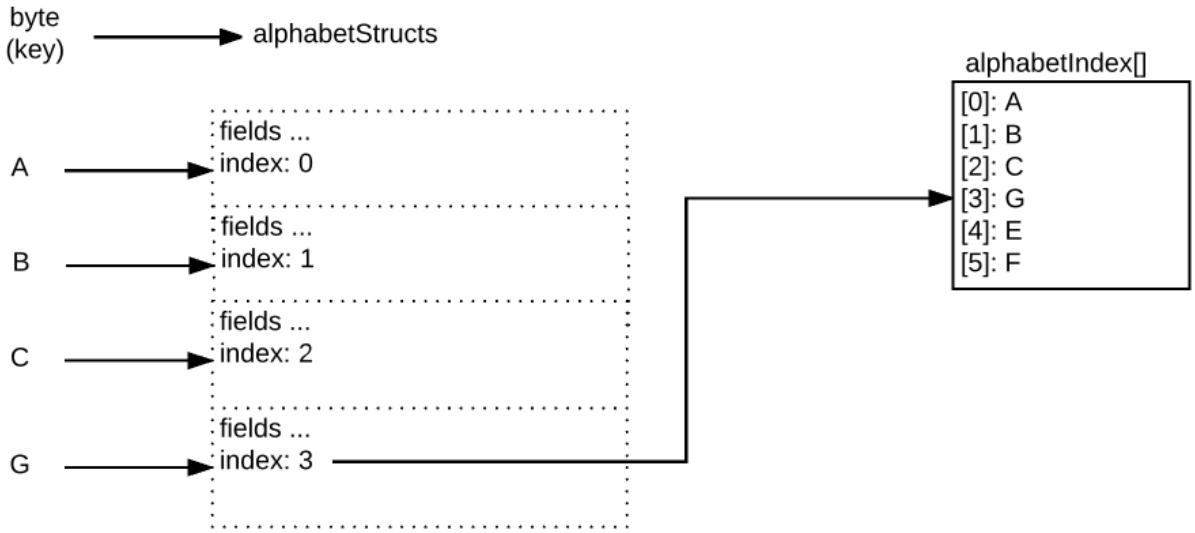
A B C G E F

All we did was move the last item into the slot we're deleting and let the list be one item shorter than it was before.



Suppose we were following the overall pattern being described, and these single-byte keys are references to mapped structs containing interesting fields about the alphabet (can't imagine what they would be, but let's go with this anyway). We have also been noting index rows for each key inserted. We record this in an additional `struct` member (`index`) included for this purpose. We would have a `mapping` of keys to `structs`, an index and pointers from the structs to index rows.

Adapting the pattern would look like this:



More precisely, delete is a few quick steps:

1. Look up the index row for the key we're planning to delete (D). It's stored in the mapped structure. We can go straight to "D" and see it's in index row 3 (original list).

   ```
   uint rowToDelete = alphabetStructs[key].index; // 3
   ```

2. Move whatever's in the last position of the index into the row we're deleting. "G" is the last item, so we will set row 3 to "G".

   ```
   Address keyToMove = alphabetindex[alphabetIndex.length-1]
   alphabetIndex[rowToDelete] = alphabetIndex[alphabetIndex.length-1]; // G
   ```

3. Don't forget to update G's index pointer. Following this pattern, there's a struct stored at "G" and it has an `index` that was originally set to 6; its original position. We moved "G", the key to move, to position 3, the row to delete. Therefore, we should update the pointer in G's `struct`.

```
alphabetStructs[keyToMove].index = rowToDelete;
```

4. G remains in the last row, which we don't want anymore. For the final step, we drop the last row of the index.

```
alphabetIndex.length--;
```

   The transforms from:

```
A B C G E F G
```

   to:

```
A B C G E F
```

The example code reorganizes the `userIndex` by initializing the `rowToDelete` and `keyToMove`,

```
uint rowToDelete = userStructs[userAddress].index;
address keyToMove = userIndex[userIndex.length-1];
```

and reorganizing in three steps:

```
userIndex[rowToDelete] = keyToMove;
userStructs[keyToMove].index = rowToDelete;
userIndex.length--;
```

Success.

In case it's unclear, removing the key from the index is as good of a delete as we are likely to get because:

1) The mapped struct (where the details still exist) is for the *exclusive* use of our contract. Sure, the data is still somewhere in the blockchain, but if this contract won't retrieve it for you (it won't), it's approximately the same as gone.
2) There's nothing we can do to prevent a determined adversary from finding data that once was but no longer is part of the current chain state. Overwriting it doesn't undo immutable history.

Finishing Up

As we did in part 1, we'll check that a key exists before we delete it and throw in case of an illogical request (deleting something that doesn't exist).

We'll also add an event emitter for deletes, since deletes are obviously a state change.

Gas Consumption and Performance

While not formally tested, gas consumption is expected to remain approximately consistent *at any scale* because each write operation proceeds in a step-by-step fashion with no branching or loops.

Representative write costs (Sample implementation):

- insertUser(): 89K
- deleteUser(): 26K
- updateUserEmail(): 8K
- updateUserAge(): 8K

The "constant" functions we use for read-only access are free, but gas is a useful proxy for the workload involved:

- getUserAtIndex(): 700 gas
- getUser(): 1,400 gas
- getUserCount(): 400 gas

These figures are rounded *up* from observed results January, 2017, solc 0.4.9.

Acknowledgement

I'd like to thank Xavier Lepretre, senior consultant at B9Lab, for his indispensable input and support.

Sample Implementation

For clarity, security is intentionally omitted.

Code is available at https://bitbucket.org/rhitchens2/soliditycrud

```
pragma solidity ^0.4.6;

contract UserCrud {

  struct UserStruct {
    bytes32 userEmail;
    uint userAge;
    uint index;
  }

  mapping(address => UserStruct) private userStructs;
  address[] private userIndex;

  event LogNewUser   (address indexed userAddress, uint index, bytes32 userEmail, uint userAge);
  event LogUpdateUser(address indexed userAddress, uint index, bytes32 userEmail, uint userAge);
  event LogDeleteUser(address indexed userAddress, uint index);

  function isUser(address userAddress)
    public
    constant
    returns(bool isIndeed)
  {
    if(userIndex.length == 0) return false;
    return (userIndex[userStructs[userAddress].index] == userAddress);
```

```
}

function insertUser(
  address userAddress,
  bytes32 userEmail,
  uint    userAge)
  public
  returns(uint index)
{
  if(isUser(userAddress)) throw;
  userStructs[userAddress].userEmail = userEmail;
  userStructs[userAddress].userAge   = userAge;
  userStructs[userAddress].index     = userIndex.push(userAddress)-1;
  LogNewUser(
      userAddress,
      userStructs[userAddress].index,
      userEmail,
      userAge);
  return userIndex.length-1;
}

function deleteUser(address userAddress)
  public
  returns(uint index)
{
  if(!isUser(userAddress)) throw;
  uint rowToDelete = userStructs[userAddress].index;
  address keyToMove = userIndex[userIndex.length-1];
  userIndex[rowToDelete] = keyToMove;
  userStructs[keyToMove].index = rowToDelete;
  userIndex.length--;
  LogDeleteUser(
      userAddress,
      rowToDelete);
  LogUpdateUser(
      keyToMove,
      rowToDelete,
      userStructs[keyToMove].userEmail,
      userStructs[keyToMove].userAge);
  return rowToDelete;
}

function getUser(address userAddress)
  public
  constant
  returns(bytes32 userEmail, uint userAge, uint index)
{
  if(!isUser(userAddress)) throw;
  return(
    userStructs[userAddress].userEmail,
    userStructs[userAddress].userAge,
    userStructs[userAddress].index);
}

function updateUserEmail(address userAddress, bytes32 userEmail)
  public
  returns(bool success)
{
  if(!isUser(userAddress)) throw;
  userStructs[userAddress].userEmail = userEmail;
  LogUpdateUser(
    userAddress,
    userStructs[userAddress].index,
    userEmail,
    userStructs[userAddress].userAge);
  return true;
}

function updateUserAge(address userAddress, uint userAge)
  public
  returns(bool success)
```

```
  {
    if(!isUser(userAddress)) throw;
    userStructs[userAddress].userAge = userAge;
    LogUpdateUser(
      userAddress,
      userStructs[userAddress].index,
      userStructs[userAddress].userEmail,
      userAge);
    return true;
  }

  function getUserCount()
    public
    constant
    returns(uint count)
  {
    return userIndex.length;
  }

  function getUserAtIndex(uint index)
    public
    constant
    returns(address userAddress)
  {
    return userIndex[index];
  }

}
```