In order to support the keyword functionality by using cuckoo hashing, we need to change both the server side and the client side query. We need to implement both the Lookup and Insertion algorithms. A simple pseudocode can be found in wiki page(https://en.wikipedia.org/wiki/Cuckoo_hashing).

The followings are a list of functions we need to modify.

- Modify the generate database part in `server.cpp` to a new function `generate_cuckoo_data` . This function mainly deal with the Insertion algorithm for cuckoo hashing. The server should try inserting untill it success. All hash functions must be made public. The client can "download" the hash functions that works. The output of this function should be a `std::vector<Entry>` , which can then be set by `set_database` easily.
- Modify `generate_query` to `generate_cuckoo_query` in `client.cpp` . Given a keyword to this new function, it first hashes the keyword to two entry indices by using the downloaded hash function. Then use `generate_query` twice to get two encoded queries.
- Another new function on the client side is required for selecting the correct output.

Design for entires

Let $F : \{0,1\}^* \mapsto \{0,1\}^\mu$ be a random oracle that hashes any key to a fixed size string. Let $(\mathbf{key}, \mathbf{value})$ be a key-value pair. An entry in the database should look like: $F(\mathbf{key}) \| \mathbf{value} \in \{0,1\}^{\text{entry\_size}}$, where $\|$ stans for concatenation. This is the same as the hash function defined in SparsePIR. From this design, both the server and clients need to store μ, the width of the hashed key. This can be added to a field in `PirParams` . Let this variable be `size_t hashed_key_width_ = 0` by default. (Better name?)

One naive way for supporting both original OnionPIR and keyword feature is to ask server and client to check this new field when running the keyword version functions. Otherwise, this value should be ignored.

Server design

The following is a naive pseudocode for `generate_cuckoo_data` . We will use zero index.

```
function generate_cuckoo_data (F, hashed_key_width, max_iter,
entry_size){
 for i := 0... max_iter - 1 do {
   h1, h2 := fresh hash functions mapping from any key to
[num_entries]
   // First, we generate all data.
   data := Entry vector[num_entries]
   for j := 0...num_entries-1 do {
     data[j] := generate_kv_pair(F, j, hashed_key_width, entry_size)
   }
   // Then, we insert them using cuckoo hashing subroutine.
   cuckoo_hash_table := Entry vector[num_entries * 2]  // we use two
hash functions
   insert_result := cuckoo_insert(h1, h2, data, cuckoo_hash_table)
   if insert_result = TRUE {
     Return cuckoo_hash_table
   }
 }
 Report FAILURE
}
```

Helper function `generate_kv_pair` .

```
function generate_kv_pair(F, key_id, hashed_key_width, entry_size) {
 // F is a hash function for hashing key to a bit string of length
hashed_key_width.
 hashed_key := F(key_id, hashed_key_width)
 value := byte vector[entry_size - hashed_key_width]
 Entry := hashed_key||value
 Return Entry
}
```

Client design

TODO