1. Background

In this project, we will consider an implementation of a data structure in hardware. This may seem strange at first because typically data structures are considered as a topic that belongs to software (i.e., we implement or use data structures when we program). However, data structures appear also in hardware. For example, every modern processor has a cache.[‡]

1.1. **Dictionaries.** The data structure we will consider maintains multi-sets (i.e., sets with repeated elements). The analog of a multi-set in Python is called a "list". We refer to this data structure as a *dictionary*.[§]

A dictionary is specified by two parameters:

(1) A *universe*. The universe is simply a set from which elements are taken. We denote the universe by U.
(2) A size n that denotes the maximum number of elements that can be stored in the dictionary. Let $m(x) \in \mathbb{N}$ denote the *multiplicity* of x in the dictionary. Namely, if an element $x \in U$ is appears 3 times in the dictionary, then $m(x) = 3$. The size n dictates that $\sum_x m(x) \leq n$.

A dictionary supports three types of operations ¶

(1) insert(x). This operation inserts x, which means $m(x) \leftarrow m(x) + 1$.
(2) delete(x). This operation deletes x, which means $m(x) \leftarrow m(x) - 1$.
(3) query(x). This operation returns exists $\in \{0, 1\}$, where

$$exists \triangleq \begin{cases} 1 & if\ m(x) \geq 1 \\ 0 & if\ m(x) = 0 \end{cases}$$

You may assume that one never attempts to insert an element to a full dictionary (i.e., a dictionary is full if $\sum_x m(x) = n$). You may also assume that one never attempts to delete an element x if $m(x) = 0$.

1.2. **Quotienting.** Quotienting is a method for saving memory when storing a set of elements. The idea is as follows: For $k \in \mathbb{N}$, let $[k]$ denote the set $\{0, \ldots, k-1\}$. Fix $Q, R \in \mathbb{N}$, and consider a universe $U = [Q] \times [R]$. It is useful to view an element $(q, r) \in U$ as a pair consisting of a *quotient* q and a *remainder* r. Consider a multi-set $\{(q_i, r_i)\}_{i=1}^{n}$. To store the multi-set, we employ an array $A[0 : Q-1]$. The entry $A[q]$ in the array stores the multi-set $\{r_i \mid q_i = q\}$. This means that we can search for (q, r) by checking if the list $A[q]$ contains r.

Example 1. *Let $Q = \{0, 1\}^2$ and $R = \{0, 1\}^4$ (note that we are using binary strings rather than natural numbers, but this should not confuse you because we simply identify a string X with the number (X)). Consider the multi-set:* $\{(00, 0000), (00, 0100), (01, 1010), (11, 1111)\}$. *Then,*

$A[00] = \{0000, 0100\}$
$A[01] = \{1010\}$
$A[10] = \emptyset$
$A[11] = \{1111\}$.

1.3. **Fano-Elias Representation.** Quotienting suggests that one can represent a multiset of n pairs $\{(q_i, r_i)\}_{i=1}^{n}$ without storing the quotients q_i explicitly. Namely, instead of storing $\log_2 Q + \log_2 R$ bits per pair, we store

only $\log_2 R$ bits per pair. What are the challenges that we need to overcome so that such a saving can be actually realized?

(1) A naive implementation of a list per array entry $A[q]$ wastes space (by space we mean bits that need to be stored in flip-flops) because it consists of a "linked list". Namely, to store the list r_1, r_2, r_3, we point to a place where r_1 is stored. Next to r_1 we store a pointer to r_2, and so on. These pointers waste space. Can we avoid pointers?
(2) We need n array entries, some of which store empty lists (e.g., $A[10]$ in the example). Is there a way to decrease the waste of space due to empty lists?

The Fano-Elias Representation suggests a method to save space. We first introduce an assumption and notation.

Lexicographic Ordering. Consider a multi-set $M = \{(q_i, r_i)\}_{i=1}^{n}$. Assume that the multi-set is sorted in *lexicographic order*. Namely, for every $1 \le i < n$,

either (i) $q_i < q_{i+1}$, or (ii) $q_i = q_{i+1}$ and $r_i \le r_{i+1}$.

Notation. For $q \in Q$, let $n_q = |\{i \mid q_i = q\}|$. Namely, n_q is the length of the list of array entry $A[q]$.

The representation of the multi-set M uses two binary strings: a header and a body. The header is the binary string

$$\text{header} = 1^{n_0} \circ 0 \circ 1^{n_1} \circ 0 \ldots 1^{n_{Q-1}} \circ 0 .$$

Note that the length of the header is $n + Q$ bits. We think of the header as a concatenation of Q blocks, which we denote by $B_0 \ldots, B_{Q-1}$. Block B_q is the string $1^{n_q} \circ 0$.

Example 2. *If we continue with Example 1, we have* 4 *blocks in the header, namely,* header $= B_{00} \circ B_{01} \circ B_{10} \circ B_{11}$, *where*

$$B_{00} = 110$$
$$B_{01} = 10$$
$$B_{10} = 0$$
$$B_{11} = 10 .$$

The body is simply a concatenation of the remainders, namely,

$$\text{body} = r_1 \circ r_2 \circ \ldots \circ r_n .$$

Note that the length of the body is $n \cdot \log_2 R$. Moreover, thanks to the lexicographic ordering, the set of remainders that share the same quotient appear contiguously in the body.

Example 3. *If we continue with Example 1,* body = 0000◦0100◦1010◦1111.

Claim 1. *(exercise) The Fano-Elias representation uses* $Q + n + n \log_2 R$ *bits. If we set* $Q = n$*, then* $2 + \log_2 R$ *bits are needed per element. (Compare this with* $\log_2 Q + \log_2 R$ *bits per element in the naive representation!)*

1.4. **Pocket Dictionary.** A Pocket-Dictionary (PD) is a dictionary that utilizes the Fano-Elias representation. It supports insertions, deletions, and queries. We will assume that the maximum number of elements that can be stored in a PD is n_{pd}. We pad the header with zeros so that its length is fixed and equals $n_{pd} + Q$ (even if the size n of the multi-set is less than n_{pd}). Similarly, we pad the body with zeros so that its length is fixed and equals $n_{pd} \cdot \log_2 R$. This means that the PD is initialized as follows: header = $0^{2n_{pd}}$ and body = $0^{n_{pd} \log_2 R}$.

We now discuss how operations are executed over the PD.

(1) query(q, r). To find (q, r), we need to figure out two things:
   (a) What is n_q?
   (b) What is $P_q = \sum_{q' < q} n_{q'}$?

   The reason is that if we know n_q and P_q, then all we need to do is compare the remainders r_i for $i \in \{P_q, \ldots, P_q + n_q - 1\}$ with the remainder r. (Recall that these remainders appear in the body.) The output is 1 if and only if we find a match.

   The values n_q and P_q can be computed from the header. We leave it as an exercise to think how this can be done.

(2) insert(q, r). To insert (q, r) we do the following (assuming that the PD is not full).
   (a) Find block B_q in the header, and update $B_q \leftarrow 1 \circ B_q$. The other blocks remain unchanged. The number of zeros padded at the end of the header is reduced by 1.
   (b) Compute P_q and insert r to the body in the "correct" loca- tion. Formally, we are supposed to store the remainders in non-descending order within each block (i.e., per quotient). However, the search procedure we proposed did not rely on the order of the remainders within each block (as long as they are "grouped" by quotients). Hence, we can insert r as the first remainder in q's group, the last one, or any location in the group.

(3) delete(q, r). We leave this as an exercise.

1.5. **Goal.** The goal in this project is to design a PD as a synchronous circuit. What does this mean? It means that the header and body are stored as a *state*, namely, they are stored in flip-flops. The circuit is input an operation (i.e., insert or query) and data (i.e., (q, r)), and modifies the header and body according to the operation and the data.

Formally, a PD(n_{pd}, Q, R), where $n_{pd} \in \mathbb{N}$, $Q = \{0,1\}^k$ and $R = \{0,1\}^\ell$ is a synchronous circuit that implements a pocket dictionary of a multi-set of at most n_{pd} pairs in $Q \times R$.

Input: op$(t) \in \{0,1\}$, $q(t) \in Q$, $r(t) \in R$.

Output: exists$(t) \in \{0,1\}$.

Functionality: Let $D(t)$ denote the multi-set defined by the operations for $t^i < t$. Namely, $D(0) \leftarrow \emptyset$, and

$$D(t+1) \triangleq \begin{cases} D(t) \cup \{q(t), r(t)\} & if\ op(t) = 0\ (insert) \\ D(t) & if\ op(t) = 1\ (query) \end{cases}$$

The pocket dictionary maintains flip-flips that together store the strings $\mathsf{header}(t)$ and $\mathsf{body}(t)$. We require that $\mathsf{header}(t) \circ \mathsf{body}(t)$ represent $D(t)$ for every t.

The output satisfies $\mathsf{exists}(t) = 1$ if and only if $(q(t), r(t)) \in D(t)$. You may assume that $|D(t)| \leq n_{pd}$, for every t. In the Logisim implementation we denote elements by $X \in \{0,1\}^{k+\ell}$. The quotient consists of the k most significant bits of X and remainder consists of the ℓ least significant bits of X.