Michael A. Bender, William Kuszmaul, Renfei Zhou: Optimal Non-Oblivious Open Addressing

presented by Petr Chmel

Definition 1 (Hash tables)**.**
An *open-addressed hash table* is an array that stores elements and empty slots in some order that supports the operations INSERT, DELETE, FIND.

If a hash table contains k elements and has size n, we say that its *load factor* is k/n.

A hash table is *oblivious* if the sequence of positions examined by FIND(u) is given by a fixed probe sequence. Otherwise it is *non-oblivious*.

Theorem 1 (Main theorem; partner hashing)**.**
There exists a non-oblivious open-addressed hash table with the following set of guarantees:

- **Load factor 1:** At any given moment, if there are n elements, then the hash table is an array of size n storing the elements in some order. There are no free slots and there is no external metadata.
- **Dynamic resizing:** The above guarantee remains true even as n changes over time. The hash table can be viewed as occupying a prefix of memory that, on insertions, increases its size by one, and on deletions, decreases its size by one.
- **Constant-time operations:** Each operation (insertion, deletion, and query) takes constant time with probability $1 - 1/\text{poly}(n)$, where n is the current size of the hash table.
- ***O*(1)-independent hash functions:** The hash table requires external access to just $O(1)$ hash functions that are each just $O(1)$-wise independent.

Theorem 2 (Main theorem, talk version)**.**
Assuming access to a constant number of fully random hash functions, there is an open-addressing hash table that maintains $n \cdot (1 - \frac{1}{\log^c n}) \pm O(1)$ keys in n slots (for any $c > 100$), supports insertions and deletions in $O(1)$ expected time, and supports queries in $O(1)$ worst-case time. The hash table requires $O(n \log n)$ bits of scratch space to initialize.

Proposition 1 (Retrieval data structure, Demaine, Meyer auf der Heide, Pagh, Pătraşcu, 2006)**.**
There is a dynamic retrieval data structure that maintains n key-value pairs, where the keys come from a poly(n)-sized universe U and the values are r-bit strings; that supports retrieval queries in $O(1)$ worst-case time; that supports insertions and deletions in $O(1)$ time with high probability in n; and that uses $O(n \log \log n + nr)$ bits of space.

Remark 1 (Some notation)**.**
Given n, we let $B = \log^{\Theta(1)} n$ be a parameter such that $n = Bm$; m is the number of bins, B is the number of slots in a single bin.
We have a fully-independent hash function $f : U \to [m]$ and a fully-independent hash function $g : [n/4] \to [m/2]$, which yields random numbers $r_0, \ldots, r_{n/4-1}$ where $r_i = g(i)$.
Every element in the hash table has a logical bin and an offset which together comprise the logical address, and it also has physical equivalents of these.

Algorithm 1: Finding the Partner of a Key

```
Function FindPartner(x):                    // Find the physical address of the partner of key x
    t ← QueryRetrieval(x)                   // If x not in the retrieval data structure, t ∈ [B] can be arbitrary
    return h(x) · B + t
```

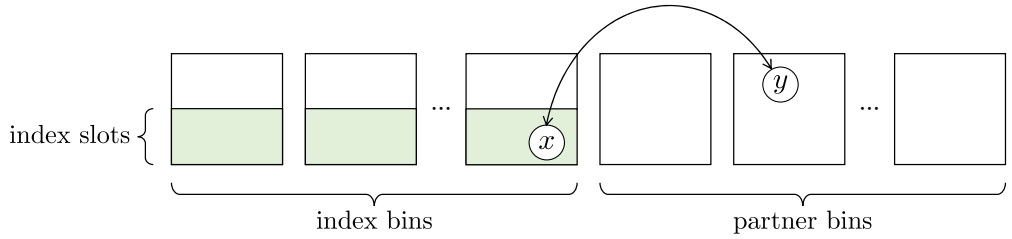

Figure 1: Index bins and partner bins. There are $m/2$ index and partner bins respectively, each containing $B = \operatorname{poly}\log n$ slots. To encode a value v_i in the i-th word, we swap the key x stored in slot $\texttt{index}(i)$ with an arbitrary self-loop y in the $\big((v_i \oplus r_i) + 1\big)$-th partner bin.

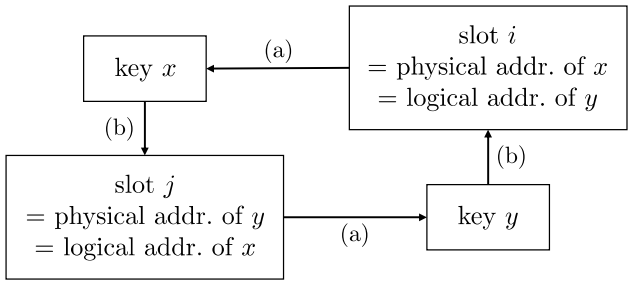

Figure 2: Using any one of x, y, i, j to recover the others.

Algorithm 2: Reading a RAM Entry

Function `ReadRAM`(i)**:**
  $x \leftarrow A[\texttt{index}(i)]$
  if $k \coloneqq h(x) - m/2 \geq 0$ **then** // x is hashed to the $(k+1)$-th partner bin
    return $k \oplus r_i$ // r_i are fixed random integers in $[m/2]$
  else // x is a self-loop
    return "RAM word i is empty"

Algorithm 3: Writing to a RAM Entry

Function `WriteRAM`$(i,\ v_i)$**:** // Write a new value v_i into word i
  $x \leftarrow A[\texttt{index}(i)]$
  if `ReadRAM`(i) *returns any non-empty value* **then**
    // We need to erase the original value before writing the new value
    Swap x with $A[\texttt{FindPartner}(x)]$ // Break the coupling pair containing index slot i
  $y \leftarrow$ find an arbitrary self-loop in the $\big((v_i \oplus r_i) + 1\big)$-th partner bin by random sampling
  Swap x with y

Algorithm 4: Querys to the Retrieval Data Structure

Function `QueryRetrieval`(x)**:**
  Call `ReadRAM` to query x on the retrieval data structure

Algorithm 5: Updating the Retrieval Data Structure

Function `UpdateRetrieval()`:

Simulate the update of the retrieval data structure by calling `ReadRAM` and `WriteRAM`.
In this process:

- When `WriteRAM` is going to relocate a key, we record the movement without actually moving it.
- When `ReadRAM` is probing a slot, it sees the slot in the state before the whole update, as we have not applied any changes to the keys and slots.

After the simulation, apply all key relocations atomically.

Algorithm 6: Key Queries

Function `QueryKey`(x):
- $s_1 \leftarrow$ `FindPartner`(x)
- **if** $A[s_1]$ *is empty* **then**
  - **return** "x is not in the hash table"
- $s_2 \leftarrow$ `FindPartner`($A[s_1]$)
- **if** $A[s_2] = x$ **then**
  - **return** "x is in slot s_2"
- **else**
  - **return** "x is not in the hash table"

Algorithm 7: Key Insertions

Function `InsertKey`(x):
- $k \leftarrow h(x)$
- $n_k \leftarrow$ the number of keys stored in bin k `// ` n_k `is stored in the RAM`
- Update $A[kB + n_k] \leftarrow x$
- Insert key-value pair (x, n_k) into the retrieval data structure
- Update $n_k \leftarrow n_k + 1$ in the RAM

Algorithm 8: Key Deletions

Function `DeleteKey`(x):
- **if** x *is in a coupling pair* **then**
  - Swap x with its partner. This will erase a word in the RAM. We copy the RAM word's value to a temporary variable. If this erased word is accessed in the following steps, we access the temporary copy instead.
- $k \leftarrow h(x)$
- $n_k \leftarrow$ the number of keys stored in bin k
- **if** $A[kB + n_k - 1]$ *is in a coupling pair* **then**
  - Swap $A[kB + n_k - 1]$ with its partner, and make a temporary copy of the erased word
- $t \leftarrow$ `QueryRetrieval`(x) `// ` $x = A[kB + t]$
- $y \leftarrow A[kB + n_k - 1]$
- Remove x from the table, and move y to $A[kB + t]$ (the slot formerly occupied by x)
- Delete key x from the retrieval data structure
- Update the value associated with y in the retrieval data structure to be t
  `// The logical address of` y `is now` $kB + t$
- Update $n_k \leftarrow n_k - 1$ in the RAM
- For all words erased in Algorithm 8, call `WriteRAM` to reencode them in RAM