Space-efficient perfect hash

Consider the two-level perfect hash tables presented in [CLRS] and discussed in class. As already discussed, for a given set of n keys from the universe U, a random universal hash function $h : U \to [m]$ is employed where $m = n$, thus creating n buckets of size $n_j \geq 0$, where $\sum_{j=0}^{n-1} nj = n$. Each bucket j uses a random universal hash function $h_j : U \to [m]$ with $m = n_j^2$. Key x is thus stored in position $h_j(x)$ of the table for bucket j, where $j = h(x)$.

This problem asks to replace each such table by a bitvector of length $n = n_j^2$, initialized to all 0s, where key x is discarded and, in its place, a bit 1 is set in position $h_j(x)$ (a similar thing was proposed in Problem 4 and thus we can have a one-side error). Design a space-efficient implementation of this variation of perfect hash, using a couple of tips. First, it can be convenient to represent the value of the table size in unary (i.e., x zeroes followed by one for size x, so 000001 represents x = 5 and 1 represents x = 0). Second, it can be useful to employ a rank-select data structure that, given any bit vector B of b bits, uses additional o(b) bits to support in O(1) time the following operations on B:

- $rank_1(i)$: return the number of 1s appearing in the first i bits of B.
- $select_1(j)$: return the position i of the jth 1, if any, appearing in B (i.e. B[i] = 1 and $rank_1(i) = j$).

Operations $rank_0(i)$ and $select_0(j)$ can be defined in the same way as above. Also, note that o(b) stands for any asymptotic cost that is smaller than $\Theta(b)$ for $b \to \inf$.

SOLUTION

The solution uses five data structures:

`H` is the Header table, it substitute the first level hash table and is memorized using a rank-select data structure;

`T` is the bitvector in which we will sore the bits for the presence/absence of the elements, it plays the role of all the second level tables;

`A` is the table that store the a parameters for the hash functions of the second level;

`B` is the table that store the b parameters for the hash functions of the second level;

`P` is the table that store the p parameters for the hash functions of the second level.

Before continue we make some observations:

1. $\sum_{j=1}^{n} n_j = n$, where n_j is the number of elements in the bucket j (called B_j from now on), as seen in class since by definition n is the number of elements to be stored;
2. in each bucket B_j we use an universal hash function h_j with $m = n_j^2$, and we have already proved that it's perfect (no collision) with probability $\geq \frac{1}{2}$;
3. given that $E[\sum_{j=1}^{n} n_j^2] < 2n$ and so by *Markov's inequality* that $Pr[\sum_{j=1}^{n} n_j^2 \geq 4n] < 1/2$, we know that $\sum_{j=1}^{n} n_j^2 \leq 4n$ with probability greater or equal to $1/2$;
4. p, the parameter of the family of hash functions for the first level of hashing is a prime number greater than n, so for the *Bertrand's postulate* we know that we can choose a $p < 2n$ and we need less than $log_2 2n = log_2 2 + log_2 n = 1 + log_2 n$ to store it;
5. we take a and b (the parameters of the hash function in the first level) from $\mathbb{Z}_p$ and, since we know that we can choose a $p < 2n$, we need less than $log_2 2n = 1 + log_2 n$ bits for each one of them;
6. for each bucket B_j we choose randomly the parameters a_j and b_j of the hash function from $\mathbb{Z}_{p_j}$, with $p_j > n_j^2$ prime number (greater than the size of the bitvector), since we can choose $p_j < 2n_j^2$, the number of bits needed for each one of the two is less then $log_2 2{n_j}^2 = 1 + log_2 {n_j}^2$, the same number of bits are needed also for storing p_j itself.

Now we can speak more precisely about H and T.

H is simply the concatenation of the unary representation of n_1^2, n_2^2 ... n_n^2, so essentially n_1^2 times 0, followed by a 1 bit, then n_2^2 times 0, followed by a 1 bit, etc. We will use an additional amount of bits because of the rank-select data structure.

The table T is the concatenation of the bit vector of each bucket: the one for B_1 followed by the one for B_2, and so on, until B_n. Notice that the length of those buckets (and respectively of the vectors) can be different one from each other. For each bucket B_j we will call its bitvector Bv_j, we recall that the size of B_j is n_j and the size of Bv_j is n_j^2. We have

$$H = (n_1^2)_1 \,|(n_2^2)_1 \mid \ldots \mid (n_n^2)_1$$
$$T = Bv_1 \,|Bv_2 \mid \ldots \mid Bv_n$$

where $(\_)_1$ means the unary notation.

So we have that initially T is the concatenation of n_1^2 bits at 0, n_2^2 bits at 0 etc... Notice that in H we can consider the 1 bits as separator between the information about one bucket and the others, but we don't have anything similar in T: since we have concatenate all the bitvectors, we need some way to retrieve the range of any bitvector. To do so, we use the length of the bitvectors, contained in the header H.

The procedure for setting to 1 the bits relative to a given key x, and for looking whether the key is in S or not is the following:

$$\begin{aligned}
j &= h_{abpn}(x) \\
k &= H.select_1(j-1) \ \#\ \text{POSITION OF THE } j-1_{th}\ 1 \\
\texttt{base} &= H.rank_0(k) \ \#\ \text{NUMBER OF 0S BEFORE THE INDEX } k \\
a' &= A[j] \\
b' &= B[j] \\
p' &= P[j] \\
m' &= H.select_1(j) - H.select_1(j-1) - 1 \\
\texttt{offset} &= h_{a'b'p'm'}(x) \\
T[\texttt{base} + \texttt{offset}] &= 1 \ //\ return\ T[\texttt{base} + \texttt{offset}]
\end{aligned}$$

where a and b are the parameters of the hash function in the first level, and the last operation is $T[\texttt{base} + \texttt{offset}] = 1$ if we are filling the table (creation phase), and $return\ T[\texttt{base} + \texttt{offset}]$ if we are querying the hash table for key x. Notice that by definition of H, m' is the number of 0s between the $j-1_{th}$ and the j_{th} 1 and so it is n_j^2. Notice also that we can calculate base as $k - j$ since all but j bits are at 0 from the position 0 to the position k. We used the notation $h_{abpm}(x)$ for $((ax+b)\ mod\ p)\ mod\ m$.

If we are querying the table and the value returned is 0 then the element is not present, instead if it is 1 we say that the element is present, with a probability of error (one side error as usual).

The space occupied by the data structures used here is the following:

H : the number of $0s = \sum_{j=1}^{n} n_j^2$ plus the number of $1s = \#\text{BUCKETS} = n$ plus the extra space for the rank-select data structure, so totally $\sum_{j=1}^{n} n_j^2 + n + o(\sum_{j=1}^{n} n_j^2 + n) \leq 4n + n + o(4n+n) = 5n + o(n)$ with probability grater or equal $1/2$;

T : the same number of $0s$ in H, $\sum_{j=1}^{n} n_j^2 \leq 4n$ whit probability grater or equal to $1/2$;

A : one array of n elements, each one needs less than $1 + log_2 {n_j}^2$ bits, then the total number of bits is less than $\sum_{j=1}^{n}(1 + log_2 {n_j}^2) = n + \sum_{j=1}^{n}(2 log_2 n_j) = n + 2\sum_{j=1}^{n}(log_2 n_j) < n + 2\sum_{j=1}^{n} n_j = n + 2n = 3n$;

B : by the same reasoning made for A we have less than $3n$ bits.

P : by the same reasoning made for A we have less than $3n$ bits.

Considering the space needed for a, b and p of the first level hash function, the total space used is less than: $5n + 4n + 3n + 3n + 3n + 3(1 + log_2 n) + o(n) = 18n + o(n)$.

Notice that if we use $p_j = p$ for each bucket B_j, than the size of A is $n log_2 2n = n(1 + log_2 n) = n + n log_2 n$, which is a problem since $n log_2 n \notin o(n)$