19. Predecessor Search in Cell Probe Model - part 1

Lecturer: Jaikumar Radhakrishnan *Scribe: Swagato Sanyal*

In this and the next two lectures, we will study the complexity of some data structure problems. Our goal is to use communication complexity to prove lower bounds on the complexity of these problems. However, in this lecture we will just give upper bounds. We will describe the *cell probe model*, introduce two problems in this model, and describe algorithms for these problems. The first problem is the *Dictionary Problem (static)*, and the second is the *Predecessor Problem*. In subsequent lectures we will see lower bounds on the complexities of these problems.

19.1 Cell Probe Model

We briefly describe the model here. Let $U = [m]$ be a universe of size m, and let $S \subseteq U$ with $|S| = n$. An algorithm is supplied with such an S, and it is supposed to answer queries about elements of S or even U. The set S is stored in memory in *cells*, each of $\log m$ bits. The execution of the algorithm has two stages:

1. Preprocessing: On receiving S, store S in memory in some suitable form. We denote the space used, measured in number of cells, by s.

2. Query: Given $x \in U$, return some information about x depending on the problem. Let t denote the maximum number of memory cell probes the algorithm takes to process each query.

The performance of the algorithm is measured only by the parameters s and t. The time taken by the preprocessing step is not counted. In the 'Query' step, the only thing that is important is number of memory probes. Note that no information is carried over from the preprocessing to the query stage unless explicitly stored in the data structure. One can think of this as two different algorithms: one for preprocessing and one for queries.

For each S, once it has been preprocessed, the execution of the Query algorithm can be represented by a decision tree. Given an $x \in U$, the algorithm proceeds as follows: depending on x it chooses a memory cell and *probes* it (reads its contents). Depending on the contents of that cell it probes some other cell, and so on. Let us fix S. For every $x \in U$, we have a decision tree. The nodes are labeled by pointers to memory cells. The root's label is the pointer to the cell probed first (which is decided completly by x). Each node has a child for every possible outcome of probing the location it points to. If t is the query complexity, then the depth of each tree is at most t.

19.2 The Dictionary Problem

In the dictionary problem, we are given an S which we need to store in memory in some form. For each $x \in S$, we will have a memory cell in a data structure T, which contains x

and a pointer to some memory location containing auxillary data about x. In addition the algorithm might allocate some additional cells which will help it in processing queries.

Given $u \in U$, the problem is to find i such that $T[i] = u$, or report that $u \notin S$. The objective is to simultaneously reduce s and t.

A completely naive solution is to store the characteristic bit vector of the set S in the preprocessing phase, and answer every query with a single probe. This scheme has $s = m$ and $t = 1$.

The standard solution is to maintain a sorted array for storing S, and to use binary search for locating elements. For this scheme, $s = O(n)$ and $t = O(\log n)$.

We can do much better for the dictionary problem using the *Fredman-Komlós-Szemerédi (FKS) scheme* from [FKS84]. It achieves $s = O(n)$ and $t = O(1)$. We describe this scheme now.

Theorem 19.1 (Fredman-Komlós-Szemerédi)**.** *There exists a solution to the dictionary problem with $s = O(n)$ and $t = O(1)$.*

The idea is to use hashing. A first attempt would be to find, in the preprocessing phase, a good hashing function $h : [m] \to [n]$ that maps S without collisions. The algorithm would then store a description of h, and information about $x \in S$ in the cell numbered $h(x)$. In the Query phase, the algorithm on input x would read h, compute $h(x)$ and look up that cell. An obvious problem here is that h must have a compact description (otherwise reading h itself will need too many probes). While this may or may not be possible, we can achieve something weaker: we can find an h with a compact description which, though not collision-free, results in sufficiently small buckets. Then we can find, for each bucket, a second-level hash function that is collision-free and has a compact description. Putting this together, both the storage requirement and the number of probes will be small. We now give the details.

We start with a claim. Pick a hash function $h : [m] \longrightarrow [n]$ uniformly at random from a family $\mathcal{H}$ of pairwise independent hash functions. For $i \in [n]$, let S_i be the ith bucket; $S_i = \{j \in S : h(j) = i\} = h^{-1}(i) \cap S$. Let K_i be the size of the ith bucket; $K_i = |h^{-1}(i) \cap S|$. The claim below shows that the expected sum of the squared bucket sizes is $Ox(n)$.

Claim 19.2. $E_h[\sum_{i \in [n]} K_i^2] = O(n)$.

Proof. $\forall u, v \in S$, let $\chi_{u,v}$ be the indicator variable of the event $h(u) = h(v)$ over the choice of h. Now, for each $u \in S$, $\Sigma_{v \in S} \chi_{u,v}$ is the number of elements in S that u clashes with, and is hence equal to $K_{h(u)}$. Since for all $u \in S_i$ the sum $\Sigma_{v \in S} \chi_{u,v}$ is equal to K_i, and since $|S_i| = K_i$, we have $E[\Sigma_{i \in [n]} K_i^2] = E[\Sigma_{i \in [n]} \Sigma_{u \in S_i} \Sigma_{v \in S} \chi_{u,v}] = E[\Sigma_{u,v \in S} \chi_{u,v}] = n + n(n-1)/n \leq 2n$. □

Now we can describe the preprocessing algorithm. For every $S \subseteq [m]$ of size n, Claim 19.2 guarantees the existence of a hash function $h : [m] \longrightarrow [n]$ for which $E[\Sigma_{i \in [n]} K_i^2] = O(n)$. Fix such an h. From now onwards we will use K_i to denote the value taken by the random variable K_i when this h that we have fixed is chosen as the hash function. For each i with $K_i \neq 0$, let $\mathcal{H}_i$ be a family of pairwise independent hash functions $[m] \longrightarrow [2K_i^2]$. If we pick an h_i randomly from this family, then the probability that h_i has a collision within

S_i is at most $\frac{1}{2K_i^2}\binom{K_i}{2} \leq 1/4$. Thus there is one function h_i which is collision-free within S_i. For each i fix one such h_i. The algorithm proceeds as follows. Given S, it determines $h, h_1, \ldots, h_n$ as above. It allocates n chunks of memory, the i-th being of size $2K_i^2$. Call the i-th chunk C_i. An array of n cells is allocated, one cell for each h_i. The i-th cell, say p_i, contains the address of the first cell of C_i. An additional array of size $n+1$ is used to store a description of the functions $h, h_1, \ldots, h_n$. The storage required is thus $O(n)$ for all the chunks together (by Claim 19.2), plus whatever is required to store the functions.

The Query algorithm on input $x \in [m]$ proceeds as follows: Read the description of h and compute $h(x) = i$. Read cell p_i and the description of h_i. Adding the contents of cell p_i to $h_i(x)$ gives a location $m(x)$. If the cell at this location does not contain x, then conclude that $x \notin S$. If it does, then read on for auxillary information about x. The choices of h and h_i's ensure that for every $x \in S$ we are mapped to a distinct memory cell.

It remains to describe how we efficiently store and compute the functions h and h_i's. Take $\mathcal{H} = \{(ax+b) \mod n : a, b \in [m], a \neq 0\}$ and $\mathcal{H}_i = \{(ax+b) \mod 2K_i^2 : a, b \in [m], a \neq 0\}$. Then h and each h_i can be described using 2 cells (for storing a and b) and computable in constant time. Hence $s \in O(n)$ and $t \in O(1)$. This completes the proof of Theorem 19.1.

19.3 The Predecessor Problem

For a non-empty set S, and for every $x \in U$, the predecessor $\mathsf{Pred}_S(x)$ is defined as

$$\mathsf{Pred}_S(x) = \begin{cases} \max\{y \in S : y \leq x\} & \text{if such a } y \in S \text{ exists} \\ -1 & \text{otherwise} \end{cases}$$

In this section we will prove an upper bound on s and t for the Predecessor Problem. We will present an algorithm for which $s = O(n \log m)$ and $t = O(\log \log m)$. This is not the best algorithm. The best algorithm for the cell probe model is due to *Beame and Fich* ([BF02]), who show how to achieve $s = O(n \log m)$ and $t = O(\min\{\frac{\log \log m}{\log \log \log m}, \sqrt{\frac{\log n}{\log \log n}}\})$. They have shown this bound to be tight for deterministic algorithms. In the next lecture we will see that this bound is optimal even if we allow randomization.

We use *X-tries* (also called van Emde Boas trees, see [CLRS09]) to design a solution. One can think of each element of $U = [m]$ as a $\log m$ bit binary string (which can just be the binary representation of the element; assume that m is a power of 2). We create a complete binary tree of depth $\log m$, whose leaves correspond to elements of $[m]$. Each edge from a node to its left child is labelled 0 and each edge from a node to its right child is labelled 1, and the labels of the edges along the path from the root to a leaf u when concatenated give the binary representation of u. Call this tree T. We edit this tree by deleting all leaves that correspond to vertices not in S, all nodes that become leaves because of these deletions and so on. Finally we have a binary tree whose leaves are exactly the elements of S. Call it T'. The number of leaves in T' is n. As in every intermediate level there can be at most n nodes, and there are $O(\log m)$ levels, the size of T' is $O(n \log m)$. Note that every element of S is its own predecessor. For an element $u \in U \setminus S$, let v be the deepest ancestor of u in T that is also present in T'. (Such a v must exist since at least one ancestor of u, the

root of T, is in T'.) By the construction above, v is not a leaf in T'. Now there are two possibilities.

1. u is in the right subtree of v in T. By choice of v, v does not have a right child in T', but is not a leaf, so it has a left child. Clearly in this case the predecessor of u is the right most leaf of the left subtree of v in T'. In the preprocessing step we will identify such nodes v and create a link pointing from v to the rightmost leaf of its left subtree.

2. u is in the left subtree of v in T. By choice of v, v does not have a left child in T'. To find a predecessor, we need to go up from v until we find a node with a left child, go to the left subtree, and report the rightmost leaf there. So in the proprocessing step we put a link from v to the rightmost leaf in the left subtree rooted at the deepest ancestor of v with a left child. If there is no such ancestor of v in T', then we link from v to a special node that will denote a -1 value for Pred_S.

Thus for each $u \in U$, once we get to the node v, we immediately obtain the predecessor by following the links. Thus we must be able to efficiently find the node v, given u. Let the bit string corresponding to u be $b_1 b_2 \dots b_k$ where $k = O(\log m)$. Then the path from the root to the node v we are looking for is $b_1 \dots b_p$ where $p = \max\{x \leq k : b_1 \dots b_x \text{ is the label of a node in } T'\}$. (Note: $x = k$ exactly when $u \in S$.) The idea is to perform a binary search in T' to identify v. If we can check whether a binary string $b_1 b_2 \dots b_l$ forms a path from the root to some node in T' with $O(1)$ probes, then we can get to the node v with $O(\log k)$ probes. This will give us the desired $O(\log \log m)$ probe solution.

It remains to show that the search can be carried out using constant probes. We maintain an FKS data structure for each level of T', storing vertices of T' in that level and as auxillary information all the ppinters leaving nodes at that level. Given a vertex string $b = b_1 b_2 \dots b_l$, the FKS scheme for level l tells us with $O(1)$ probes whether the corresponding node is there in the tree or not. If it is there, the FKS data structure also gives us the pointers leaving from that node. So once the binary search concludes and we find the maximum x as above, we can follow the appropriate links set up in the preprocessing step to get to the predecessor of u.

References

[BF02] Paul Beame and Faith E. Fich. *Optimal bounds for the predecessor problem and related problems.* Journal of Computer and System Sciences, 65(1):38–72, 2002. (Preliminary version in *31st STOC*, 1999). doi:10.1006/jcss.2002.1822.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, 3 edition, 2009.

[FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. *Storing a sparse table with O(1) worst case access time.* Journal of the ACM, 31(3):538–544, 1984. (Preliminary version in *23rd FOCS*, 1982). doi:10.1145/828.1884.