Lecture Notes: The logarithmic method

Yufei Tao
Chinese University of Hong Kong
taoyf@cse.cuhk.edu.hk

15 Apr, 2012

The topic of this lecture is about *indexing*. Namely, given an input set S of elements, we would like to *preprocess* S by creating a data structure on it. Then, given any query, we can utilize the structure to answer it efficiently, in particular, with much lower cost than if we did not have the structure. As an example, let us recall when the binary search tree – a fundamental structure at the undergraduate level – would be useful. Let S be a set of real values; given a real value q, a *predecessor query* finds the maximum value in S that are at most q. To support such queries efficiently, we can create a binary search tree on S, after which any query can be answered in $O(\lg n)$ time where $n = |S|$. This is much faster than answering q without using any structure – in that case, $O(n)$ time is compulsory.

We consider only *decomposable problems*. Formally, assume that S is divided into two disjoint partitions: S_1, S_2, namely, $S_1 \cap S_2 = \emptyset$ and $S_1 \cup S_2 = S$. A query on S is *decomposable* if the answer can be obtained in constant time from the answers of the same query on S_1 and S_2, respectively. For example, the predecessor problem mentioned earlier is clearly decomposable: to find the predecessor in S of a real value q, we can first find the predecessors of q in S_1, S_2 respectively, and then return the larger of those two predecessors.

In many applications, we need to make our structure *dynamic*. Namely, we want to support two types of updates: (i) *insertion*, where an element e is added in S, and (ii) *deletion*, where an element existing in S is removed – in both cases, the structure on S must be modified accordingly, so that it is still a valid index on the updated S. For the predecessor problem, the binary search tree can be updated in $O(\lg n)$ time per insertion and deletion. For some problems, however, designing a dynamic structure that supports fast updates can be rather difficult. The *nearest neighbor problem* is an example. In this problem, we are given a set S of n two-dimensional points in $\mathbb{R}^2$. Given a point $q \in \mathbb{R}^2$, we want to find the data point in S that is *nearest* to q by Euclidean distance. There is an elegant, static, structure based on the *Voronoi diagram* [3], which consumes $O(n)$ space, and answers each query in $O(\lg n)$ time. Unfortunately, updating a Voronoi diagram turns out to be rather challenging, and as a result, making the nearest neighbor structure dynamic very difficult. There has been some recent breakthrough on this problem [2], where a dynamic structure that can be updated in $O(\lg^6 n)$ time is given. The structure is quite theoretical, and is not easy to implement.

It turns out that life is much easier if we only want our structure to be *semi-dynamic*, that is, we want to perform only insertions, but *not* deletions. Although not as useful as (fully) dynamic structures, semi-dynamic indexes also play an important role in practice, when the dataset of the underlying application only expands, but never shrinks. We will learn in this lecture a clever technique called the *logarithmic method* that can convert a static structure to a semi-dynamic structure. The only requirement is that the structure can be *constructed* efficiently. The static nearest neighbor structure mentioned earlier can be built in $O(n \lg n)$ time. We will see that the logarithmic method permits us to obtain (quite effortlessly) a semi-dynamic structure that consumes

$O(n)$ space, answers a query in $O(\lg^2 n)$ time, and supports an insertion in $O(\lg^2 n)$ amortized time.

1 The logarithmic method

This method, sometimes also called *logarithmic rebuilding*, is due to Bentley and Saxe [1]. Assume that we have already designed a static structure. Let the input set S be empty initially. Next, we will explain how to maintain a semi-dynamic structure on S as new elements are inserted in S. Once again, deletions are not allowed.

Let $n = |S|$ be the number of elements that have been inserted so far (at the beginning, $n = 0$). At all times, S is divided into $h = \lfloor \lg n \rfloor + 1$ *partitions* $S_0, ..., S_{h-1}$. Namely, these partitions are mutually disjoint, and their union equals S. Partition S_i ($0 \leq i \leq h-1$) either is empty, or *must* have size 2^i – in the former case, we say that S_i is *off*, whereas in the latter, we say that S_i is *on*. In any case, each S_i is indexed by a static structure T_i (if S_i is off, then T_i is empty).

Whether S_i is on or off is determined by an interesting rule. Let us write the value of n in binary form, which is an h-bit binary string. Refer to the *least* significant bit as *bit 0*, and in general, the second least significant bit as *bit 1*, and so on. Hence, the most significant bit is *bit* $h-1$. Then, S_i is on if bit i is 1; otherwise, S_i is off.

Insertion. Now we are ready to explain how to handle an insertion. Let e be the new element to be added to S. We first find the smallest i such that S_i is empty. If i is 0, then we simply create an S_0 with only e itself (remember that S_0, if not empty, is allowed to contain only 1 element), and build T_0. If $i > 0$, on the other hand, we union all the elements of $S_0, ..., S_{i-1}$, together with e, into S_i. Note that S_i now contains

$$2^0 + 2^1 + ... + 2^{i-1} + 1 = 2^i$$

elements, namely, exactly the size that a non-empty S_i is supposed to have. Now, we empty all $S_0, ..., S_{i-1}$ (because all their elements have moved to S_i), and destroy $T_0, ..., T_{i-1}$ accordingly. Finally, we build T_i for S_i from scratch. Note that, before the insertion, all of $S_0, ..., S_{i-1}$ were on while S_i was off. Afterwards, $S_0, ..., S_{i-1}$ are off, while S_i is on.

It may be inspiring to point out that, an insertion is performed in a manner drastically different from what happens in a binary search tree. In the above algorithm, we never really "updated" any particular T_i. Instead, all we did was only to *destroy* (i.e., discarding a structure) and then *rebuild*.

Query. To answer a query on S, we issue it on each of $S_0, ..., S_{h-1}$ using the structures on them, respectively. Since the query is decomposable (recall that we restrict our attention to decomposable problems only in this lecture), the final answer can be obtained from the answers of $S_0, ..., S_{h-1}$ in $O(h) = O(\lg n)$ time.

2 Analysis

Let us characterize the efficiency of a static structure by its space usage, query time, and construction time. Specifically, suppose that, on an input set of size n, our static structure occupies at most $spc(n)$ space, answers a query in at most $qry(n)$ time, and can be constructed in at most $bld(n)$ time. Next, we will analyze the space, query, and insertion cost of the semi-dynamic structure obtained by the logarithmic method.

Space. This is quite obvious: since $S_0, ..., S_{h-1}$ form a partition of S, it follows that the total space

consumption is

$$\sum_{i=0}^{i=h-1} spc(|S_i|) \leq \sum_{i=0}^{i=h-1} spc(2^i).$$

Query. Recall that we answer a query by issuing it on each of $T_0, ..., T_{h-1}$. Clearly, the cost of performing the query on each T_i $(0 \leq i \leq h-1)$ is at most $qry(2^i)$, noticing that T_i indexes at most 2^i elements. Hence, the total cost of all the h queries is

$$\sum_{i=0}^{i=h-1} qry(2^i).$$

Finally, we spend $O(h)$ time combining all their results into the final answer, but this is dominated by the above summation (noticing that each term in the summation is at least 1).

Insertion. It is easy to see that not all insertions can be performed with low cost: an insertion may trigger the reconstruction of an index on a sizable partition, and hence, incur expensive cost. However, it turns out that the *amortized* cost per insertion is low. We will show this using a charging argument.

Recall that, to insert an element, we first find the smallest i such that S_i is empty. Then, we destroy $T_0, ..., T_{i-1}$, and rebuild T_i from their elements. Since T_i indexes exactly 2^i elements, we know that the reconstruction of T_i takes at most $bld(2^i)$ time. We charge the cost over the 2^i elements that are now in T_i. Hence, each element bears at most $\frac{1}{2^i} \cdot bld(2^i)$ cost.

After n insertions, how much cost can an element be charged this way? The crucial observation is that an element can only move from S_i to S_j such that $i < j$, namely, the subscript monotonically increases each time the element needs to bear extra cost. Hence, the element can be amortized at most

$$\sum_{i=0}^{i=h-1} \left(\frac{1}{2^i} \cdot bld(2^i) \right)$$

cost. This is the amortized cost per insertion.

We thus have obtained:

Theorem 1. *Our semi-dynamic structure uses at most $\sum_{i=0}^{i=h-1} spc(2^i)$ space, answers a query in $O(\sum_{i=0}^{i=h-1} qry(2^i))$ time, and supports an insertion in at most $\sum_{i=0}^{i=h-1} \left(\frac{1}{2^i} \cdot bld(2^i)\right)$ amortized time.*

Application. Now let us use the above theorem to obtain a semi-dynamic structure for the nearest neighbor problem. As mentioned before, on an input set of size n, there is a static structure that uses at most $spn(n) = O(n)$ space, answers a query in at most $qry(n) = O(\lg n)$ time, and can be constructed in at most $bld(n) = O(n \lg n)$ time. We prove:

Lemma 1. *For the nearest neighbor problem, there is a semi-dynamic structure that uses $O(n)$ space, answers a query in $O(\lg^2 n)$ time, and supports an insertion in $O(\lg^2 n)$ amortized time.*

Proof. It follows directly from Theorem 1 (note: $\frac{1}{2^i} \cdot bld(2^i) = \frac{1}{2^i} \cdot O(2^i \cdot \lg 2^i) = O(\lg n)$). □

References

[1] J. L. Bentley and J. B. Saxe. Decomposable searching problems I: Static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980.

[2] T. M. Chan. A dynamic data structure for 3-d convex hulls and 2-d nearest neighbor queries. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1196–1202, 2006.

[3] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3 edition, 2008.