6.851: Advanced Data Structures Spring 2010

Lecture 9 — March 4, 2010

Dr. André Schulz *Scribe: Paul Christiano*

1 Overview

Last lecture we defined the Least Common Ancestor (LCA) and Range Min Query (RMQ) problems. Recall that an LCA query asks for the deepest node in a given rooted tree which is an ancestor of each of two query nodes, while an RMQ query asks for the minimal value stored in some contiguous subarray of a given array. We showed last lecture that an LCA query is nothing more than an RMQ query on an Euler tour of the given tree (where the value assigned to any node is its depth in the tree). Moreover, this array has the property that the difference between any pair of consecutive elements is ± 1. Thus if we could solve the RMQ problem for so-called "± 1 arrays" we could solve LCA.

In the first part of lecture, we show how to answer RMQ queries ± 1 arrays and then using this data structure how to answer general RMQ queries.

In the second part of lecture, we present some models of computations on integers and then develop data structures to answer predecessor/successor queries on a set $S \subset \{1, 2, \ldots, u\}$: given a query integer x, we will be able to report the smallest element of S greater than x or the largest element of S less than x. We will develop van Emde Boas trees and y-fast trees, which both attain query times of $O(\log \log u)$ for this problem.

2 RMQ and LCA

We use the same notation from the last lecture; namely, we say an algorithm is $\langle f, g \rangle$ for a certain class of queries if it can answer queries from that class in g time with f preprocessing.

Throughout, we write A for the array on which we wish to perform RMQ queries, and n for the length of A.

2.1 $\langle O(n \log n), O(1) \rangle$ algorithm for RMQ

Our first data structure for RMQ uses a sparse lookup table M to answer queries. The width of this table is n, but the height is only $\log n$. The entry $M[i, j]$ stores the minimum index in $A[i, i + 2^j - 1]$. We can build this table in $O(n \log n)$ time using dynamic programming: $M[i, j] = \min\big(M[i, j-1], M[i + 2^{j-1}, j-1]\big)$.

Once we have built this table, we can answer a query RMQ$[i, j]$ in constant time as follows. Suppose $2^k < j - i < 2^{k+1}$. Then the minimum entry in $A[i, j]$ is simply the minimum of $A[i, i + 2^k - 1]$ and $A[j - 2^k + 1, i]$, both of which are stored in the sparse table.

2.2 $\langle O(n), O(1)\rangle$ algorithm for RMQ ± 1

Our next data structure reduces space by a factor of $O(\log n)$ with the same trick we used for LAQ in the last lecture. Namely, we break A into blocks of size $\frac{\log n}{2}$ and precompute all queries on small blocks. More precisely, form new arrays A' and B' of the length $\frac{2n}{\log n}$ such that $A'[i]$ is the minimum value stored in $A\left[\frac{i\log n}{2}, \frac{(i+1)\log n}{2}\right]$ and $B'[i]$ is the index of the element attaining that minimum. Then, form a new array $\text{SUB}[i] = A\left[\frac{i\log n}{2}, \frac{(i+1)\log n}{2}\right]$ for each $0 \le i \le \frac{2n}{\log n}$.

In order to answer a query RMQ$[i, j]$ with i and j in the same block, we answer an RMQ within that block. If i and j are in different blocks, then we need to do three things:

- Do an RMQ query in the block containing i to find the smallest element in that block which lies between i and j.
- Do an RMQ query in the block containing j to find the smallest element in that block which lies between i and j.
- Do an RMQ query in A' to find the smallest element lying in any of the block strictly between the one containing i and the one containing j, and then lookup in B' the index of this smallest element in A .

We use our $\langle O(n\log n), O(1)\rangle$ algorithm to answer query (iii), which is fast enough since A' has size $O(n/\log n)$. We cannot use this technique to answer (i) and (ii), because this would give total space use $O(n\log\log n)$. However, we can use the fact that there are not many combinatorially distinct subarrays SUB$[i]$ in order to save significantly on space.

In particular, if two subarrays differ by a fixed offset, then identical queries on those two subarrays will always return the same index. Thus we only need to build one lookup table for each of the distinct sequences of differences of consecutive elements. Call the signature of a subarray this sequence of differences. Since we are dealing with ± 1 arrays, this sequence of differences is in $\{-1, 1\}^{\log n/2}$. Thus there are only $\sqrt{n}$ distinct signatures. For each signature, we can build a lookup table for all of the $O(\log^2 n)$ possible RMQs. The total preprocessing and space use is $O(\sqrt{n}\log^2 n)$.

Now to answer queries of type (i) and (ii), or queries within a single block, lookup the signature of the block you want to query (store a table of these signatures) and then lookup the answer in the appropriate table. This clearly takes $O(1)$ operations.

We now have a $\langle O(n), O(1)\rangle$ algorithm for RMQ ± 1 and therefore for LCA.

2.3 $\langle O(n), O(1)\rangle$ algorithm for RMQ

We will reduce RMQ to LCA, for which we have just demonstrated a $\langle O(n), O(1)\rangle$ algorithm.

Recall that a Cartesian tree for a sequence A is a tree whose root contains a minimal element $A[x]$, with left subtree a Cartesian tree on $A[0, x]$ and right subtree a Cartesian tree on $A[x :]$.

In fact, an LCA for a Cartesian tree for A is identical to RMQ for A. Because A is an in-order traversal of its Cartesian tree, the least common ancestor of $A[i]$ and $A[j]$ is in $A[i, j]$. Moreover, all

of $A[i, j]$ is a descendant of this least common ancestor. By the construction of the Cartesian tree, it must be the case that this least common ancestor corresponds to a minimal element in $A[i, j]$.

The only remaining task is to construct the Cartesian tree in linear time from A. We have seen how to do this in the context of suffix arrays (it is a standard technique—if you add A one at a time onto a stack, popping off elements to make sure that the stack remains ordered, the execution history allows you to recover the Cartesian tree).

3 Integer Data Structures

Before presenting any integer data structures, we present some models for computations with integers (both to precisely state what we are looking for, and as a framework to prove lower bounds).

In each model, memory is arranged as a collection of words of some size (and potentially a small amount of working memory), and a fixed cost is charged for each operation. We also always make the *fixed universe assumption*. We assume that each data structure needs to answer queries with respect to a fixed set of integers $\mathcal{U} = \{1, 2, \ldots, u\}$.

The following models are given in order of decreasing power.

3.1 The Cell-Probe Model

- Memory is divided into cells of size w, where w is a parameter of the model.
- Reading to or writing from a memory cell costs 1 unit.
- Other operations are free

This model does not realistically reflect a modern processor, but its simplicity and generality make it appropriate for proving lower bounds.

3.2 Trandichotomous RAM

- Memory is divided into cells of size w, where $w \geq \log n$ depends on the input size n. In practice, this is a convenient and realistic assumption since the machine word size is normally large enough to index into the problem.
- There is some fixed set of operations, each of which modifies $O(1)$ memory cells.
- Cells can be addressed arbitrarily.

3.2.1 word-RAM

In word-RAM, the processor may use the $O(1)$ fixed operations $+, -, *, /, \%, \&, |, \hat{}, >>, <<, \ldots$ you would expect as primitives in a high-level language such as C.

3.2.2 AC^0-RAM

AC^0 is the class of functions which can be implemented by a circuit with polynomially many gates and constant depth. In AC^0-RAM, you may perform any AC^0 operation on a word. In particular, multiplication is not in AC^0, so it takes $O(\log w)$ steps to multiply.

3.3 Pointer Machine

This is a less general model than RAM or Cell-Probe; it describes data structures which store integers, but does not describe computations on those integers. A data structure in this model is a directed graph with bounded out-degree, each node of which stores a single memory cell. When the processor is examining one node, it can only move to adjacent nodes in the data structure or examine the contents of the memory cell in the current node.

3.4 Binary Search Tree

This is the model we discussed in the first two lectures. It is the same as the pointer machine data structure, with the additional stipulation that the underlying directed graph forms a valid binary search tree.

4 Successor / Predecessor Queries

We are interested in data structures which maintain a set of integers S and which are able to insert an integer into S, delete an integer from S, report the smallest element of S greater than some query x, or report the largest element of S less than some query x.

The data structures we discuss today will answer such queries in time $\log \log u$, where u is the universe size.

4.1 Van Emde Boas Trees

We store the elements of S as bit vector, ie an array A of length u with $A[i] = 1$ iff $i \in S$. We split this array into blocks of size $\sqrt{u}$, $\text{SUB}[A, 0], \text{SUB}[A, 1], \ldots, \text{SUB}[A, \sqrt{u}]$, and we recursively use the same data structure to facilitate searches within each of these blocks. Finally, we maintain a summary structure Summary$[A]$ which contains all integers i such that $\text{SUB}[A, i]$ is not empty.

Given a binary string x write $\text{high}(x)$ for the first $|x|/2$ digits and $\text{low}(x)$ for the last $|x|/2$ digits. When x is an integer with binary representation $\text{bin}(x)$, we write $\text{high}(x)$ for the number whose binary representation is $\text{high}(\text{bin}(x))$ and $\text{low}(x)$ for the number whose binary representation is $\text{low}(\text{bin}(x))$. Then given an integer x, it is stored in block $\text{SUB}[A, \text{high}(x)]$ and within that block it is at position $\text{low}(x)$.

To answer a query $\text{Successor}(A, x)$, we first compute $k = \text{Successor}(\text{SUB}[A, \text{high}(x)], \text{low}(x))$ (we are imagining $\text{SUB}[A, \text{high}(x)]$ as storing integers in the range $0, 1, \ldots \sqrt{u}$ for convenience instead of $\text{high}(x), \text{high}(x) + 1, \ldots, \text{high}(x) + \sqrt{u}$). If $k < \infty$, then $\text{Successor}(A, x) = k + \text{high}(x)\sqrt{u}$.

Otherwise, Successor(A, x) is not in the same block as x: we need to find out what block it is in. So we let $z = \text{Successor}(\text{Summary}[A], \text{high}(x))$ to find the next non-empty block. If $z = \infty$ then we report ∞, since there are no more non-empty blocks. If instead $z = k' < \infty$, then we report $\text{Successor}(\text{SUB}[A, k'], -\infty) + k' * \sqrt{u}$.

The running T of this data structure satisfies $T(u) = 3T(\sqrt{u}) + O(1)$. In order to get $T(u) = \log \log u$, we need to get rid of the coefficient 3; that is, we need to perform only one recursive query at each step.

To fix this, it is sufficient to store also the maximum and minimum element of S. Now given a query x, first check if $\text{low}(x) > \max[\text{SUB}[A, \text{high}(x)]]$. If not, then we need to do a single query in $\text{SUB}[A, \text{high}(x)]$. If so, then we need to do a query in $\text{Summary}[A]$ and then report the minimal element of the next non-empty block. In each case, we only need to do one recursive search and at most two $O(1)$ lookups.

If we modify this data structure very slightly we can make the space use $O(u)$ and modifications $O(\log \log u)$. The trick is to not store the minimum or maximum elements of your data structure in the structure itself—only store them in the max/min field. Then any item can be the maximum or minimum in at most one structure (in fact exactly one), and updates can be performed in $O(\log \log u)$ time.

Unfortunately, the size of a van Emde Boas tree is $\Theta(u)$, which is often very large compared to n. To address this we will develop y-fast trees, which obtain the same guarantees with high probability and have linear storage and preprocessing.

5 X-Fast Trees

Both x-fast trees and y-fast trees are due to Willard [1]. X-fast trees are an intermediate step towards y-fast trees, which we will describe first.

To motivate x-fast trees, imagine forming a binary tree on u leaves which stores a 1 at the i^{th} leave iff $i \in S$ and a 0 otherwise. We store a 1 in each node which has any descendant with a 1 in it and a 0 otherwise. For convenience, we also store a linked list on the elements of S.

The following algorithm computes either the predecessor or the successor of a query x in $\log u$ time. Starting from the leaf corresponding to x, continue walking up until we encounter a node v with a 1 in it. If we are willing to ignore whether we get a successor or predecessor, we can assume WLOG that the query is in the left subtree of v. Now walk down from v in the following way. As long as v's left child has a 1, walk left. Otherwise, walk right. In this way, you come to the leftmost leaf with a 1 in it which is to the right of the query, ie its successor in S. If you wanted the predecessor instead of the successor, (or if you got the successor and wanted the predecessor) just take one step in the linked list.

We don't really have to spend $\log u$ time walking down the tree. Every time we walk down from a certain node v we will get to the same point: we can precompute where we will end up and then jump there in one step in a query (at the cost of $\log u$ time to update this information). Walking up is still slow, so we will need a trick to speed it up.

We store all prefixes of all elements of S in a hash table (if we use dynamic perfect hashing we attain $O(1)$ time for all operations with high probability). Our walk up the binary search tree will

always terminate in the longest common prefix of the query point and a point in S. Once we have a hash table storing all such prefixes, we can determine by a binary search (in $\log\log u$ time rather than $\log u$ time) the maximum length of any such common prefix. Having done this, we can store in our hash table the leftmost and rightmost leaves which have that prefix and simply jump directly to that leaf.

This attains $\log\log u$ time for queries, but it takes $O(\log u)$ to update, and for each element we store $\log u$ prefixes giving total space of $O(n \log u)$.

6 Y-Fast Trees

To save a $\log u$ factor in the construction of x-fast trees, we employ a trick similar to the one we used to speed up RMQ and LAQ. We split S up into $\frac{n}{\log u}$ contiguous subsets $S = S_1 \cup S_2 \cup \cdots \cup S_k$ each of size $O(\log u)$. The idea is to store all of these trees in an x-fast tree, since we can now afford the extra $O(\log u)$, and then to just use a binary tree on these small sets since we can afford to spend $O(\log\log u)$.

To this end, pick separators r_i such that $\max S_i \leq r_i \leq \min S_{i+1}$, and place all of these separators in an x-fast tree. For each separator, r_i, store a link to a binary search tree storing all of the elements of S_i and a link to r_{i-1} and r_{i+1}.

To answer a query Successor(x), use the x-fast tree to find the smallest separator $r_i > x$ and then search for x in S_i and S_{i+1}. All of these steps take $O(\log\log u)$ time. Predecessor queries can be answered similarly.

Since the total space requirement of all of the binary search trees is linear, we obtain a total space requirement of $O(n + \log u \frac{n}{\log u}) = O(n)$. Each modification of one of the BSTs takes only $O(\log\log u)$ time, since the trees have size $O(\log u)$. A modification of the x-fast tree still takes $O(\log u)$ time, but this can be amortized to $o(\log\log u)$ of in the following way.

We allow the S_i to have size in $[1/2 \log u, 2 \log u]$. Once a BST gets too large, we split it into two BSTs. Once a BST gets too small, we merge it with one of the adjacent BSTs and then potentially split the result. These splits and merges, recomputation of the separators r_i, and adjustment of the x-fast tree can all be done in $O(\log u)$ time, so we just need to show that this happens in at most one out of every $\log u$ updates. However, after a split we can verify that a binary search tree has between $2/3 \log u$ and $3/2 \log u$ vertices, so does not need to split or merged again for another $O(\log u)$ operations. When we merge two trees, we pay in advance for the next time we will have to split the merged tree. Thus the total amortized time is $O(1)$ for each modification.

This completes the analysis. We obtain $O(n)$ space use, $O(\log\log u)$ time to answer a query with high probability, and $O(\log\log u)$ amortized time for modifications with high probability.

References

[1] D. Willard, *Log-logarithmic worst case range queries are possible in space $O(N)$*, Inform. Process. Lett. 17 (1983), pp. 81-89.