6.851: Advanced Data Structures Spring 2012

Lecture 15 — April 12, 2012

Prof. Erik Demaine *Scribes: Jelle van den Hooff (2012), Yuri Lin (2012)*
Anand Oza (2012), Andrew Winslow (2010)

1 Overview

In this lecture, we look at various data structures to solve problems about static trees: given a static tree, we perform some preprocessing to construct our data structure, then use the data structure to answer queries about the tree. The three problems we look at in this lecture are range minimum queries (RMQ), lowest common ancestors (LCA), and level ancestor (LA); we will support all these queries in constant time per operation, using linear space.

1.1 Range Minimum Query (RMQ)

In the range minimum query problem, we are given an array A of n numbers (to preprocess). In a query, the goal is to find the minimum element in a range spanned by $A[i]$ and $A[j]$:

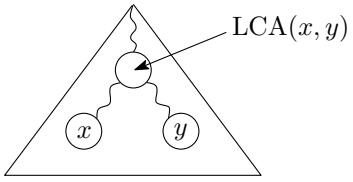
$$\begin{aligned}\text{RMQ}(i,j) &= (\text{arg})\min\{A[i], A[i+1], \ldots, A[j]\} \\ &= k, \text{ where } i \le k \le j \text{ and } A[k] \text{ is minimized}\end{aligned}$$

We care not only about the value of the minimum element, but also about the index k of the minimum element between $A[i]$ and $A[j]$; given the index, it is easy to look up the actual value of the minimum element, so it is a more general problem to find the index of the minimum element between $A[i]$ and $A[j]$.

The range minimum query problem is closely related to the lowest common ancestor problem.

1.2 Lowest Common Ancestor (LCA)

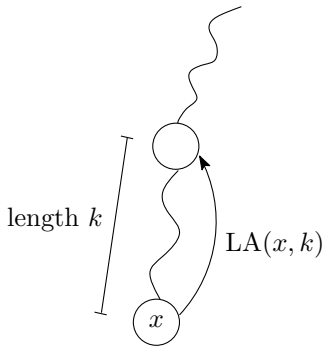
In the lowest common ancestor problem (sometimes less accurately referred to as "least" common ancestor), we want to preprocess a rooted tree T with n nodes. In a query, we are given two nodes x and y and the goal is to find their lowest common ancestor in T:



1.3 Level Ancestor (LA)

We are again given a rooted tree T to preprocess. Given a node x and an integer $k \leq \text{depth}(x)$, the query goal is to find the k^{th} ancestor of node x:

$$\text{LA}(x, k) = \text{parent}^k(x)$$



1.4 Similarity of Problems

All three problems will be solved in the word RAM model, though the use of this model is not as essential as it was in the integer data structures we discussed in previous lectures. Although lowest common ancestor and level ancestor seem like similar problems, fairly different techniques are necessary to solve them (as far as anyone knows). The range minimum query problem, however, is basically identical to that of finding the lowest common ancestor.

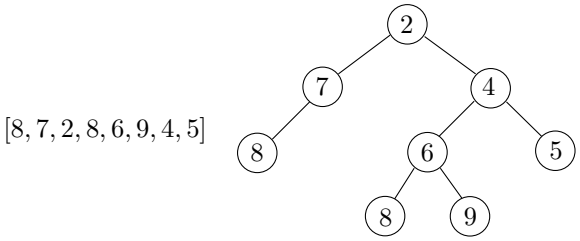
2 Reductions between RMQ and LCA

2.1 Cartesian Trees: Reduction from RMQ to LCA

A Cartesian tree is a nice reduction mechanism from an array A to a binary tree T that provides an equivalence between RMQ and LCA, and dates back to a 1984 paper by Gabow, Bentley, and Tarjan [1].

To construct a Cartesian tree, we begin with the minimum element of the array A, which we will call $A[i]$. This element becomes the root of the Cartesian tree T. Then the left subtree of T is a Cartesian tree on all elements to the left of $A[i]$ (which we can write as $A[< i]$), and the right subtree of T is likewise a Cartesian tree on the elements $A[> i]$.

An example is shown below for the array $A = [8, 7, 2, 8, 6, 9, 4, 5]$. The minimum of the array is 2, which gets promoted to the root. This decomposes the problem into two halves, one for the left subarray $[8, 7]$ and one for the right subarray $[8, 6, 9, 4, 5]$. 7 is the minimum element in the left subarray and becomes the left child of the root; 4 is the minimum element of the right subarray and is the right child of the root. This procedure continues until we get the binary tree in the diagram below.



In the case of ties between multiple equal minimum elements, two options are:

1. Break ties arbitrarily, picking one of the equal elements as the minimum.
2. Consider all of the equal elements to be one "node", making a non-binary tree.

Because the second option is slightly messier, in this class we will break ties arbitrarily, though this choice will not affect the answer.

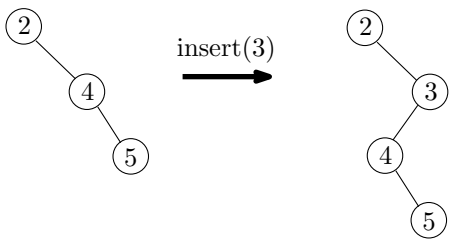
The resulting tree T is a min heap. More interestingly, the result of a range minimum query for a given range in the array A is the lowest common ancestor of the two endpoints in the corresponding Cartesian tree T. To see this, note that any RMQ for the range $A[j]$ through $A[k]$ where $j < i < k$ has $A[i]$ as the answer, as it is the minimal element of the array and is between j, k. We see that the LCA of $A[j]$ and $A[k]$ in the tree will also be the root (since they're part of the left-subtree and right-subtree respectively), but the root is simply $A[i]$ by construction. Any RMQ for a range completely to the left or to the right of $A[i]$ is now an RMQ on the respective subarray, which corresponds to a subtree of the root, so we can use induction to complete the correspondence between RMQ and LCA.

2.1.1 Construction in linear time

Construction of the Cartesian tree according to the naïve recursive algorithm will take at least $\Omega(n \lg n)$ time, and may even take quadratic time. Fortunately, Cartesian trees can be computed in linear time, using a similar method to that of building a compressed trie in linear time.

Walk through the array from left to right, inserting each element into the tree by walking up the right spine of the tree (starting from the leaf), and inserting the element in the appropriate place. Because we are building the tree from left to right, each inserted element will by definition be the rightmost element of the tree created so far.

For example, if we have a tree for which the subarray $[2, 4, 5]$ has been inserted, and the next element is 3, then insertion has the following result:



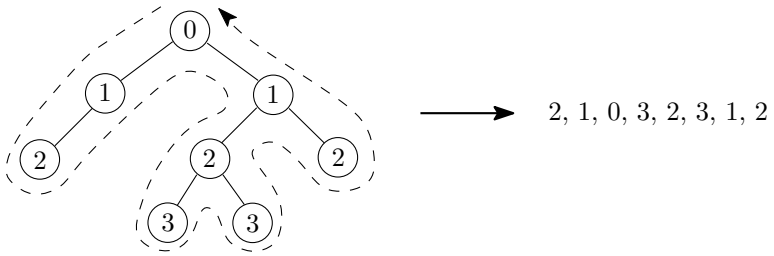
We walk up the right spine of the tree starting at 5, and continue until we reach 2 (the root), which is the first element smaller than 3. The edge between 2 and 4 is replaced with a new edge from 2 to 3, and the previous right subtree of 2 becomes the left subtree of 3.

Each such insertion takes constant amortized time: although sometimes paths may be long, each insertion only touches nodes along the right spine of the tree, and any node along the right spine that has been touched ends up in the left subtree of the inserted node. Any node along the right spine is touched at most once, and we can charge the expensive inserts to the decrease in length of the right spine.

Therefore, construction of Cartesian trees can be done in linear time, even in the comparison model.

2.2 Reduction from LCA to RMQ

We can also reduce in the other direction, from LCA to RMQ, by reconstructing an array A from a binary tree T. To do this, we perform an in-order traversal of the nodes in the tree. However, we must have numbers to use as the values of the array; to this end, we label each node with its depth in the tree.



This sequence behaves exactly like the original array $A = [8, 7, 2, 8, 6, 9, 4, 5]$, from which this tree was constructed. The result for $\text{RMQ}(i, j)$ on the resulting array A is the same as calling $\text{LCA}(i, j)$ on the input tree for the corresponding nodes.

2.3 RMQ universe reduction

In general the elements of an RMQ array may be in any arbitrary ordered universe. By chaining the two reductions above, first by building a Cartesian tree from the elements and then by converting back from the tree to an array of depths, we can reduce the range to the set of integers $\{0, 1, \ldots, n-1\}$. We can now assume that all of the inputs are small integers, which allows us to solve things in constant time in the word RAM model.

3 Constant time LCA and RMQ

3.1 Results

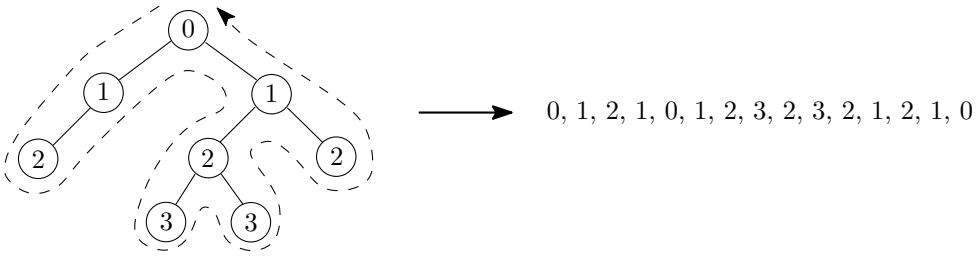
The LCA and RMQ problems can both be optimally solved with constant query time and linear storage space. The first known technique is documented in a 1984 paper by Harel and Tarjan [2].

In lecture, we looked at an algorithm based on a 2004 paper by Bender and Colton [3].

3.2 Reduction from LCA to ±1 RMQ

The algorithm by Bender and Colton solves a special case of the RMQ problem called ±1 RMQ, in which adjacent values differ by either $+1$ or -1.

First, we will take a look at a reduction from LCA to ±1 RMQ. The earlier reduction does not work as differences might have absolute values larger than 1. For our new approach, we perform an Eulerian tour based on the in-order traversal and write down every visit in our LCA array. At every step of the tour we either go down a level or up a level, so the difference between two adjacent values in our array is either $+1$ or -1.



Since every edge is visited twice, we have more entries in the array than in the original in-order traversal, but still only $O(N)$. To answer $\text{LCA}(x, y)$, calculate $\text{RMQ}(\text{in-order}(x), \text{in-order}(y))$ where $\text{in-order}(x)$ is the first in-order occurrence of the node x in the array. These occurrences can be stored while creating the array. Observe that this new array can also be created by filling in the gaps in the array from the original algorithm. Thus the minimum between any two original values does not change, and this new algorithm also works.

3.3 Constant time, $n \lg n$ space RMQ

A simple data structure than can answer RMQ queries in constant time but uses only $n \lg n$ space can be created by precomputing the RMQ for all intervals with lengths that are powers of 2. There are a total of $O(n \lg n)$ such intervals, as there are $\lg n$ interval-lengths that are powers of 2 no longer than n, and n possible start locations.

We claim that any queried interval is the (non-disjoint) union of two power of 2 intervals. Say the query has length k. Then the query can be covered by the two intervals of length $2^{\lfloor \lg k \rfloor}$ that touch the beginning and ending of the query. The query can be answered by taking the min of the two precomputed answers. The overlap is fine because an element can be included in the min twice without changing the result.

To solve the argmin version we just store the location of the minimum element.

3.4 Indirection to remove log factors

We have seen indirection used to remove log factors of time, but here we will apply indirection to achieve a $O(n)$ space bound. Divide the array into bottom groups of size $\frac{1}{2}\lg n$ (this specific constant will be used later). Then, store a parent array of size $2n/\lg n$ that stores the min of every group.

Now a query can be answered by finding the RMQ of a sequence in the parent array, and at most two RMQ queries in bottom groups to look in the bottom arrays at the edges of the query range. Note that we might have to answer two sided queries in a bottom group if a query is so small that it fits in one bottom group. For the parent array we can use the $n \lg n$ space algorithm as the logarithms cancel out.

3.5 RMQ on very small arrays

The only remaining question is how to solve the RMQ problem on arrays of size $n' = \frac{1}{2}\lg n$. The idea is to use lookup tables, since the total number of different possible arrays is very small.

Observe that we only need to look at the relative values in a group to find the location of the minimum element. This means that we can shift all the values so that the first element in the group is 0. Then, once we know the location, we can look in the original array to find the value of the minimum element.

Now we will use the fact the array elements differ by either $+1$ or -1. After shifting the first value to 0, there are now only $2^{\frac{1}{2}\lg n} = \sqrt{n}$ different possible bottom groups since every group is completely defined by its n' long sequence of $+1$ and -1s. This total of $\sqrt{n}$ is far smaller than the actual number of groups!

In fact, it is small enough so that we can store a lookup table for any of the n'^2 possible queries for any of the $\sqrt{n}$ groups in $O(\sqrt{n}(\frac{1}{2}\lg n)^2 \lg\lg n)$ bits, which easily fits in $O(n)$ space (the $\lg\lg n$ bits come from the fact that the index of the answer must be $< \log n$, since the small arrays only have size $< \log n$). Now every group can store a pointer into the lookup table, and all queries can be answered in constant time with linear space for the parent array, bottom groups, and tables.

3.6 Generalized RMQ

We have LCA using ± 1 RMQ in linear space and constant time. Since general RMQ can be reduced LCA using universe reduction, we also have a linear space, constant time RMQ algorithm.

4 Level Ancestor (LA)

First we introduce notation. Let $h(v)$ be the height of a node v in a tree (i.e. the length of the longest path to a leaf node in the tree from v). Given a node v and level l, $LA(v, l)$ is the ancestor a of v such that $h(a) - h(v) = l$. Put more simply, we go up the tree l levels from v and see where we end up.

Today we will study a variety of data structures with various preprocessing and query times to

solve $LA(v, l)$ queries. For a data structure which requires $f(n)$ query time and $g(n)$ preprocessing time, we will denote its running time as $\langle g(n), f(n)\rangle$. The following algorithms are taken from the set found in a paper from Bender and Farach-Colton [4].

4.1 Algorithm A: $\langle O(n^2), O(1)\rangle$

The basic idea is to use a look-up table with one axis corresponding to nodes and the other axis corresponding to levels. Then, we fill in the table using dynamic programming by increasing level. This is the *brute force* approach.

4.2 Algorithm B: $\langle O(n \lg n), O(\lg n)\rangle$

The basic idea is to use **jump pointers**, which are pointers that are stored at a node and reference one of the node's ancestors. For each node, we create jump pointers to ancestors at levels $1, 2, 4, \ldots, 2^k$. Queries are answered by repeatedly jumping from node to node, each time jumping more than half of the remaining levels between the current ancestor and goal ancestor. Therefore, the worst-case number of jumps is bounded by $O(\lg n)$. Preprocessing is done by filling in jump pointers using dynamic programming.

4.3 Algorithm C: $\langle O(n), O(\sqrt{n})\rangle$

The basic idea is to use a **longest path decomposition**. Split a tree recursively by removing the longest path it contains and iterating on the remaining connected subtrees. This decomposes the tree into disjoint paths, so each edge appears in exactly one path. Each path removed is stored as an array in top-to-bottom path order, and each array has a pointer from its first element (the root of the path) to its parent in the tree (an element of the path-array from the previous recursive level). A query is answered by moving upwards in this *tree of arrays*, traversing each array in $O(1)$ time. In the worst case the longest path decomposition may result in longest paths of sizes $k, k-1, \ldots, 2, 1$ each of which has only one child, resulting in a tree of arrays with height $O(\sqrt{n})$. Building the decomposition can be done in linear time by precomputing node heights once, then reusing them to find the longest paths quickly.

4.4 Algorithm D: $\langle O(n), O(\lg n)\rangle$

The basic idea is to use **ladder decomposition**. This is similar to longest path decomposition, but each path is extended by a factor of two backwards (up the tree past the root of the longest path). If the extended path reaches the root, it stops. Each node stores the highest ladder it is in (i.e. the one whose unextended path contains it, not any other ladders which might have been extended to contain it).

Each node v lies on a path of size at least the height of v, so it points to a ladder of at least twice that height. Thus each jump at least doubles the height, so one does at most $O(\lg n)$ ladder jumps (each constant time) before reaching an array that contains the queried ancestor, at which point we can index the remaining number of steps upward to retrieve it in constant time. Preprocessing is done similarly to Algorithm C.

4.5 Algorithm E: $\langle O(n \lg n), O(1) \rangle$

The idea is to combine jump pointers (Algorithm B) and ladders (Algorithm D). Each query will use one jump pointer and one ladder to reach the desired node. First a jump is performed to get at least halfway to the ancestor. The node jumped to is contained in a ladder which also contains the goal ancestor, because its path goes at least as far down as the point we queried from, and extends at least twice as high up as any point in its path. (Note: when we reach an element there might be mutliple ladders containing it, but we take the one that contained it in the original path decomposition, before extending into ladders.) Thus we get $O(1)$ queries.

4.6 Algorithm F: $\langle O(n), O(1) \rangle$

An algorithm developed by Dietz [5] also solves LA queries in $\langle O(n), O(1) \rangle$ but is more complicated. Here we combine Algorithm E with a reduction in the number of nodes for which jump pointers are calculated. The motivation is that if one knows the level ancestor of v at level l, one knows the level ancestor of a descendant of v at level l'. So we compute jump pointers only for leaves, guaranteeing every node has a descendant in this set. So far, preprocessing time is $O(n + L \lg n)$ where L is the number of leaves. Unfortunately, for an arbitrary tree, we only have the worse-case bound $L = O(n)$. If we could get L, the number of leaves to satisfy $L = O(\frac{n}{\log n})$, then we would be able to have linear space constant time query. This is what we do next!

4.6.1 Building a tree with $O(\frac{n}{\lg n})$ leaves

Split the tree structure into two components: a **macro-tree** at the root, and a set of **micro-trees** (of maximal size $\frac{1}{4} \lg n$) rooted at the leaves of the macro-tree. For a given query, if it is contained within a micro-tree (this can be easily checked by storing a pointer for every node in the micro-tree to the corresponding leaf in the macro-tree and checking if that node's height is too large), then we will use a lookup table to compute the answer. Otherwise, we'll simply use a combination of the tuned jump pointers (only keeping them at the leaves of the macro tree) and ladders for querying in the macro-tree.

But how many micro-trees can there be? Consider a depth-first search, keeping track of the orientation of of the ith edge, using 0 for downwards and 1 for upwards. A micro-tree can be described by a binary sequence, e.g. $W = 001001011$ where for a tree of size n, $|W| = 2n - 1$. So an upper bound the number of micro-trees possible is $2^{2n-1} = 2^{2(\frac{1}{4} \lg n)-1} = O(\sqrt{n})$. This is equivalent to saying there are at most 2^{2n-1} Euler tours of a tree, from which you can reconstruct the tree from the sequence of up and down moves. But this is a loose upper bound, as not all binary sequences are possible, e.g. $00000\ldots$ A valid micro-tree sequences has an equal number of zeros and ones and any prefix of a valid sequence as at least as many zeros as ones.

4.6.2 Use macro/micro-tree for a $\langle O(n), O(1) \rangle$ solution to LA

We will use macro/micro-trees to build a tree with $O(\frac{n}{\lg n})$ leaves and compute jump pointers only for its leaves ($O(n)$ time). We also compute all micro-trees and their look-up tables (see Algorithm A) in $O(\sqrt{n} \lg n)$ time. So total preprocessing time is $O(n)$. A query $LA(v, l)$ is performed in the

following way: If v is in the macro-tree, jump to the leaf descendant of v, then jump from the leaf and climb a ladder. If v is in a micro-tree and $LA(v, l)$ is in the micro-tree, use the look-up table for the leaf. If v is in a micro-tree and $LA(v, l)$ is not in the micro-tree, then jump to the leaf descendant of v, then jump from the leaf and climb a ladder. However, this upper bound will suffice for our purposes.

References

[1] H. Gabow, J. Bentley, R. Tarjan. Scaling and Related Techniques for Geometry Problems. In *STOC '84: Proc. 16th ACM Symp. Theory of Computing*, pages 135-143, 1984.

[2] Dov Harel, Robert Endre Tarjan, *Fast Algorithms for Finding Nearest Common Ancestors*, SIAM J. Comput. 13(2): 338-355 (1984)

[3] Michael A. Bender, Martin Farach-Colton, *The LCA Problem Revisited*, LATIN 2000: 88-94

[4] M. Bender, M. Farach-Colton. The Level Ancestor Problem simplified. Lecture Notes in Computer Science. 321: 5-12. 2004.

[5] P. Dietz. Finding level-ancestors in dynamic trees. Algorithms and Data Structures, 2nd Workshop WADS '91 Proceedings. Lecture Notes in Computer Science 519: 32-40. 1991.