Advanced Algorithms | Trees and search | Teaching Block 1

(TA : Kheeran K. Naidu)

Problem sheet 2 (Solutions)

Please feel free to discuss these problems on the unit discussion board or directly with your colleagues. If you would like to have your answers marked, please either hand them in in person at a lecture or problems class. Submitted work will be marked as quickly as possible, ideally within one week of being handed in.

1. The dynamic predecessor problem can be defined to be the dynamic dictionary problem with the addition of the PREDECESSOR operation. Recall that given a set of integers Y , the PREDECESSOR of x (which may not be in Y) is the largest $v \in Y$ such that $v \leq x$. The dynamic predecessor problem could solved by either a self-balancing search tree (such as an AVL or Red-Black tree) or a van Emde Boas tree.

   (a) Give one advantage of using a van Emde Boas tree over a self-balancing search tree.

   Solution. Each operation takes $O(\log \log u)$ time which is significantly faster than a self balancing tree unless only few elements from the universe are stored ; i.e. when $n \in o(\log u)$ where u is the size of the universe and n is the number of elements stored.

   ✓

   (b) Give one advantage of using a self-balancing search tree over a van Emde Boas tree.

   Solution. It uses much less space. The space required for a self balancing tree is linear in the number of elements n, instead of the (much larger) size of the universe u.

   ✓

2. How can you perform DELETE in a van Emde Boas tree ? You should write your solution in the same style as in the lecture slides. What is the relevant recurrence relation for the time complexity of DELETE and what is its solution in terms of big O complexity ?

   Solution. See Figure 1 for the pseudocode. DELETE performs at most one recursive call in any of the cases while the other operations take at most $O(\log \log u)$ time as in the lectures. Hence, $T(u) = T(\sqrt{u}) + O(\log \log u)$. Using substitution and the Master Theorem we get a time complexity of $O(\log \log u)$.

   ✓

To perform *delete*(x) :
Determine which B[i] the element x belongs in
Let x_i be the element x after adjusting for offset
Let $\min_i$ and $\max_i$ be the min and max of B[i]
Case 1 *if* x_i is not stored in B[i]
do nothing
Case 2 *else if* $x_i = \min_i = \max_i$
remove $\min_i$ and $\max_i$
delete(i) in C
Case 3 *else if* $x_i = \min_i$
set $\min_i$ to *successor*(x_i)
delete($\min_i$) in B[i]
Case 4 *else if* $x_i = \max_i$
delete(x_i) in B[i]
set $\max_i$ to *predecessor*(x_i)
Case 5 *else*
delete(x_i) in B[i]

FIGURE 1 – DELETE operation

3. Looking at slide 189 of the van Emde Boas tree lectures slides, consider a version of van Emde Boas trees where the new minimum is always recursively inserted into the tree instead of being stored at only one level. Write down the recurrence relation for the time complexity of the ADD operation and give its solution in big O notation.

   Solution. In this case, the ADD operation will require up to two recursive calls such that
   $$T(u) = 2T(\sqrt{u}) + O(1).$$
   Then, by substitution of $u = 2^m$ and $T(2^m) = S(m)$ we get
   $$S(m) = 2S(\frac{m}{2}) + O(1)$$
   which can be solved by the Master Theorem to get a time complexity of $O(m) = O(\log u)$.

   ✓

4. In 2D orthogonal range search, justify why we the number of 1D lookups performed is $O(\log n)$ (see slide 165).

   Solution. The number of nodes on any path in a balanced binary tree is $O(\log n)$. Hence the path found from x_1 to x_2 when performing *lookup*(x_1, x_2, y_1, y_2) has $O(\log n)$ nodes, each of which requires a single 1D lookup.

✓

5. In some applications one is interested only in the number of points that lie in a range rather than in reporting all of them. Such queries are often referred to as range counting queries. In this case one would like to avoid having an additive term of $O(k)$ in the query time.

   (a) Describe how a 1-dimensional range tree can be adapted such that a range counting query can be performed in $O(\log n)$ time. Prove the query time bound.

   Solution. Build a balanced binary tree as in the lectures, but additionally store the (sorted) index of the element at each node during pre-processing. When performing $count(x_1, x_2)$, find the successor of x_1 and the predecessor of x_2 both in $O(\log n)$ time. Then use the corresponding indices to find the number of elements in the range in $O(1)$ time. Hence, the time complexity is $O(\log n)$.

   ✓

   (b) Using the solution to the 1-dimensional problem, describe how 2-dimensional range counting queries can be answered in $O\left(\log^2 n\right)$ time. Prove the query time.

   Solution. Follow the (unimproved) 2D range search using a balanced binary tree from the lectures, but replace the 1D lookup with the 1D count described in 5a. This performs $O(\log n)$ 1D counts each taking $O(\log n)$ time. Hence, the time complexity is $O(\log^2 n)$.

   ✓

   (c) (*)Describe how fractional cascading can be used to improve the running time by a factor of $O(\log n)$ for 2-dimensional range counting queries.

   Solution. The slow part of 1D count described in 5a is finding the successor and the predecessor. We can follow a fractional cascading approach and add a link to the successors of each element to reduce this step to time $O(1)$ as in the lectures. Additionally we can do the same thing for predecessors using no significant extra prep time or space, and reducing the time complexity of 1D count to $O(1)$. Hence, the overall time complexity of 2D count is $O(\log n)$.

   ✓