6.854 **Dan Ports**
Advanced Algorithms `drkp@mit.edu`
2004/11/23 Collaborators: {`sarahl`,`pramook`}

Problem Set 11

Problem 1:

Suppose we are given N horizontal and vertical line segments in the plane. We give an external-memory sweep-line algorithm for identifying the K intersections between them. We begin by generating a list of x-coordinates of interest (left and right endpoints of horizontal segments and vertical segment x-coordinates), and sorting them. We then sweep a vertical line across the plane from left to right, maintaining a buffer tree sorted by y-coordinate of active line segments. For each point,

- if it is a left endpoint of a horizontal segment, we INSERT the segment into the buffer tree, ordered by its y-coordinate.
- if it a right endpoint of a horizontal segment, we DELETE the y-coordinate. (Note that each y-coordinate is inserted no more than once, since no two horizontal segments intersect.)
- if it is a vertical segment, we perform a BATCHED-RANGE-SEARCH over the y-coordinates of the endpoints. Any matching horizontal segment intersects the vertical segment, so we output the intersection. (We tag both the BATCHED-RANGE-SEARCH and the INSERT records in the buffer tree with segment identifiers, so we know which segments are intersecting.)

Once the full plane has been swept, we perform a FLUSH operation to ensure that all pending output from the buffer tree is produced.

This algorithm requires $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}+\frac{K}{B}\right)$ memory accesses. There are $O(N)$ points along which the line is swept, since each segment contributes at most two. The initial sorting step requires $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ memory accesses. For each of the $O(N)$ positions of the sweep line, we perform a INSERT, DELETE, or BATCHED-RANGE-SEARCH; the first two operations require $O\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ amortized memory accesses, and the latter requires $O\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}+\frac{k}{B}\right)$ amortized memory accesses, where k is the size of the output produced by that BATCHED-RANGE-SEARCH operation. So the total number of memory accesses required by the sweep-line portion of the algorithm is $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}+\frac{K}{B}\right)$. Performing the final FLUSH step requires $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ memory accesses. So the total cost is $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}+\frac{K}{B}\right)$ memory accesses.

Problem 2:

a) We give a procedure for adding a pointer from each node to its successor's successor.

We keep one copy of the linked list in the original ordering, and make another copy which we sort by the successor field. This requires $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ memory transfers using the black box. We then scan through these two lists in lockstep. Node i in the originally-ordered list (call this N_i) will be the successor of node i in the successor-ordered list (call this S_i). So we can add a successor's-successor pointer from S_i to N_i's successor.

This requires $O\left(\frac{N}{B}\right)$ memory transfers: a standard scanning procedure can be used. So the total cost is $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$.

b) We now label each node with the outcome of a fair coin flip. Let $\alpha = 1/4$. We can obtain αN selected nodes in expectation by selecting the nodes which are marked H by the coin flip *and* whose successors are marked T. This gives $\frac{N}{4}$ nodes in expectation: the probability that any node x is selected is

$$\Pr\left[(x = \mathsf{H}) \cap (\textsc{Succ}(x) = \mathsf{T})\right] = \frac{1}{2}\left(\frac{1}{2}\right) = \frac{1}{4}$$

since the coin flips are fair. So by linearity of expectation $\frac{N}{4}$ nodes are selected in expectation.

This requires $O\left(\frac{N}{B}\right)$ memory transfers once we have the originally-ordered and successor-ordered sorted lists described above. The assignment of a coin-flip value to each node can be performed by scanning over the lists in lockstep, and selecting the nodes S_i such that S_i is marked H and N_i is marked T can also be performed by scanning over the list. So $O\left(\frac{N}{B}\right)$ memory transfers are required for scanning, plus $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ for the sort.

c) We now remove all nodes which are not selected from the list. We use a scanning approach: we scan over the sorted originally-ordered and successor-ordered lists in lockstep again, partitioning the nodes into two output lists: if a node is selected, we place it in the first output list, and if it is not, we place it in the second output list. In addition, as we process a node, we add a pointer from each node (a N_i) to its predecessor (S_i). Since memory accesses are sequential, this can be performed using $O\left(\frac{N}{B}\right)$ memory accesses for the scanning, plus the sorting time.

d) Now suppose we have recursively solved the list-ranking problem on the set of unselected elements. We show how to extend this solution to the original list-ranking problem. Recall that nodes were selected such that no two selected nodes were adjacent; so each selected node has both a predecessor and a successor in the unselected set. We sort the unselected elements by their (now-computed) rank, and sort the selected elements by their predecessor's rank. Then we scan over the two lists in lockstep. The ith item S_i in the selected list will have the ith item U_i in the unselected list as its predecessors. So S_i's rank will be one greater than the rank of U_i, and we can assign it this rank. Computing this requires $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ memory accesses for sorting, and $O\left(\frac{N}{B}\right)$ for scanning, so it is dominated by the sorting cost.

e) Each problem of size N reduces to solving the list-ranking problem over the set of unselected elements, which has expected size $\frac{3}{4}N$, plus a merging cost of $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$. So the recurrence is

$$T(N) = T\left(\frac{3}{4}N\right) + O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$$

and we also have the standard base case

$$T(M) = O\left(\frac{M}{B}\right)$$

so the first level of the recursion tree (which has no branches) has cost $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$, and the next has cost $O\left(\frac{3}{4}\frac{N}{B}\log_{\frac{M}{B}}\frac{\frac{3}{4}N}{B}\right)$, etc., so the total cost is

$$\sum_{i=0}^{\log_{\frac{3}{4}}\frac{M}{N}} O\left(\left(\frac{3}{4}\right)^i\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right) \le \sum_{i=0}^{\infty} O\left(\left(\frac{3}{4}\right)^i\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right) = O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$$

as it is simply a standard geometric series.

Problem 3:

a) Suppose we are given an array of N nodes in arbitrary order with left and right child lists. We show how to augment the nodes with their parent pointers and whether they are left or right children of their parent.

We begin by identifying the left children. We make a copy of the list and sort it by the value of the left child pointer. We then scan through the newly sorted list. For each element x in the left-child-ordered list, we add a parent pointer from $x.\mathsf{left\text{-}child}$ to x, and mark $x.\mathsf{left\text{-}child}$ as a left-child. This is a scanning algorithm on the sorted list. But it also accesses the originally-ordered list in a scanning order: since left-child-ordered list is sorted by left-child, the left children of the list will be in a sequential ordering of the original list. Thus the number of memory accesses is bounded above by the amount needed for scanning, $O\left(\frac{N}{B}\right)$. The dominating factor is the cost required for sorting, $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$.

This procedure identifies the left children; a symmetric procedure identifies the right children in the same time bound.

b) Now consider the problem of generating an Euler tour of the tree (with edges doubled to make it feasible). Note that an Euler tour satisfies the following local property at any vertex x: the vertex first traverses the edge $(x.\mathsf{parent} \to x)$, then $(x \to x.\mathsf{left\text{-}child})$, then traverses x's left subtree and eventually returns via the edge $(x.\mathsf{left\text{-}child} \to x)$, then leaves via $(x \to x.\mathsf{right\text{-}child})$ to traverse x's right subtree, then eventually returns via $(x.\mathsf{right\text{-}child} \to x)$, and finally leaves via the $(x \to x.\mathsf{parent})$ edge. Indeed, this property, if satisfied for all x, is sufficient to define an Euler tour over a tree.

Thus at most 6 edges are required in an Euler tour per vertex: three entering the vertex, and three leaving. (This is in the case where a node has both left and right children; if one or both subtrees are missing, then we simply ignore the two edges that correspond to that subtree.) We can therefore represent the Euler tour in the following way: for every vertex x, we "divide" it into three vertices x_1, x_2, and x_3, numbered in the order they are traversed in the Euler tour. Then the tour takes a path

$$\mathsf{parent} \to x_1 \to \mathsf{left\ subtree} \to x_2 \to \mathsf{right\ subtree} \to x_3 \to \mathsf{parent}$$

Note that each vertex has only one incoming and one outgoing edge, so it can easily be expressed as a linked list.

Our procedure for generating the Euler tour is as follows: we scan over the array of nodes, and generate an output array in which each input node is replaced with the three corresponding output nodes. For each node x, let p be its parent, and l and r be its left and right subchildren respectively. Then we output x_1, x_2, and x_3 with the following links:

$$\begin{aligned}
x_1.\mathsf{prev} &= \begin{cases} p_1 & \text{if } x \text{ is a left subchild of } p \\ p_2 & \text{if } x \text{ is a right subchild of } p \end{cases} \\
x_1.\mathsf{next} &= l_1 \\
x_2.\mathsf{prev} &= l_3 \\
x_2.\mathsf{next} &= r_1 \\
x_3.\mathsf{prev} &= r_3 \\
x_3.\mathsf{next} &= \begin{cases} p_2 & \text{if } x \text{ is a left subchild of } p \\ p_3 & \text{if } x \text{ is a right subchild of } p \end{cases}
\end{aligned}$$

Note that if x has no left subchild, we simply change $x_1.\mathsf{next}$ to point to x_2 and set $x_2.\mathsf{prev}$ accordingly, and similarly if x has no right subchild. It is straightforward (but tedious) to verify the correctness and consistency of the resulting linked list, which is the Euler tour.

This is produced using a scanning algorithm, so the required memory is $O\left(\frac{N}{B}+1\right)$. The resulting linked list has 3 nodes for each node in the input list, so its size is $O(N)$.

c) Suppose we wish to compute the size of the subtree rooted at each vertex. We can do so by generating the Euler tour using the procedure described above and computing the list ranking of the tour in $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$.

Recall that the Euler tour takes the path

$$x.\mathsf{parent} \to x_1 \to x\text{'s }\mathsf{subtree} \to x_2 \to x.\mathsf{parent}$$

so the size of the subtree rooted at x is the number of (distinct) vertices the tour passes between when it reaches x_1 and when it leaves x_3, inclusive. Each vertex appears three times in this tour (as a_1, a_2, and a_3), so the size of the subtree rooted at x is

$$\textsc{Subtree-Size}(x) = \frac{\textsc{List-Rank}(x_3) - \textsc{List-Rank}(x_1) + 1}{3}$$

We can compute this value for all x by scanning over the Euler tour array in $O\left(\frac{N}{B}\right)$, so the total number of memory accesses required is $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$.

Problem 4:

a) Suppose the input stream contains n (non-distinct) elements, more than $\frac{n}{2}$ of which are equal to some distinct value A. We show that at the end of the stream, the register contains value A.

Suppose that at the end of the stream the register contains some element $B \neq A$. This leads to a contradiction. Every time that A appeared in the input stream, there are two cases: if some other value stays in the register, then the A lowers its count and there must have been a non-A element before the A in the stream; if the register contains A, then it increments the counter, and since in order for A to not be in the register at the end, the counter eventually reaches zero, and so there must be a corresponding non-A element somewhere after the A in the stream. Thus, for each A element, there is a corresponding non-A element in the stream. So since there are at least $\frac{n}{2}+1$ A elements in the stream, there must be at least twice as many non-A elements. But there are only n elements in the stream. This is a contradiction $\not\leftrightarrow$.

b) We maintain k counters with k registers. We initialize the registers to empty and the counters to zero. For each element in the stream,

- if the element is contained in one of the registers, we increment the associated counter.
- if the element is not contained in any of the registers, and there are any empty registers (with associated counter of zero), we place the new element in some empty register and set its counter to 1.
- if the element is not contained in any of the registers, and there are no empty registers, we decrement each counter by 1, and empty any registers whose associated counter now equals zero. If a register has been emptied, we place the new element in it and set the associated counter value to 1.

The proof of correctness for this algorithm is essentially the same as for the $k = 1$ case: each element that is not in a register can decrement k counters, so for each occurrence of an element that does not wind up in a register at the end of the algorithm, there must be k occurrences of other elements. Thus, if the number of occurrences of this element is x, then there must be at least kx other elements, and so the total number of elements is at least $x + kx$. But there are n elements, so

$$\begin{aligned} x + kx &\leq n \\ x &\leq \frac{n}{1+k} \end{aligned}$$

and thus all the elements that occur with frequency more than $\frac{1}{1+k}$ will be in a register at the end.

Problem 5:

$\leq$ 10 hours