6.851: Advanced Data Structures Spring 2012

Lecture 9 — March 15, 2012

Prof. Erik Demaine

Scribes: Tim Kaler(2012)
Jenny Li(2012)
Elena Tatarchenko(2012)

1 Overview

This is the last lecture on memory hierarchies. Today's lecture is a crossover between cache-oblivious data structures and geometric data structures.

First, we describe an optimal cache-oblivious sorting algorithm called Lazy Funnelsort. We'll then see how to combine Lazy Funnelsort with the sweepline geometric technique to solve batched geometric problems. Using this sweepline method, we discuss how to solve batched orthogonal 2D range searching. Finally, we'll discuss online orthogonal 2D range searching, including a linear-space cache-oblivious data structure for 2-sided range series, as well as saving a log log factor from the normal 4-sided range search.

2 Lazy Funnelsort

A funnel merges several sorted lists into one sorted list in an output buffer. Suppose we'd like to merge K sorted lists of total size K^3. We can merge them in $O(\frac{K^3}{B} \lg_{M/B}(\frac{K}{B}) + K)$ memory transfers. *Note: The $+K$ term may be important in certain implementations, but generally the first term will dominate.*

A K-funnel is a complete binary tree with K leaves. Each of the subtrees in a K-funnel is recursively a $\sqrt{K}$-funnel. The edges of a $\sqrt{K}$-funnel point to buffers, each of which is size $K^{3/2}$. Because there are $\sqrt{K}$ buffers, the total size of the buffers in the K-funnel is K^2. See Figure 1 for a visual representation.

When the k-funnel is initialized, its buffers are all empty. At the very bottom, the leaves point to input lists.

The actual funnelsort algorithm is an $N^{1/3}$-way mergesort with $N^{1/3}$-funnel merger. *Note: We can only afford $N^{1/3}$-funnel as a merger because of the K^3 multiplier from above.*

2.1 Filling buffers

When filling a buffer, we'd like the end result to be the output buffer being completely full.

In order to fill a buffer, we must look to the two child buffers to supply the data (See Figure 2). We will merge the elements of the child buffers into the parent buffer, as long as both child buffers still contain items. We merge by picking and putting in the smaller element of the two child buffers

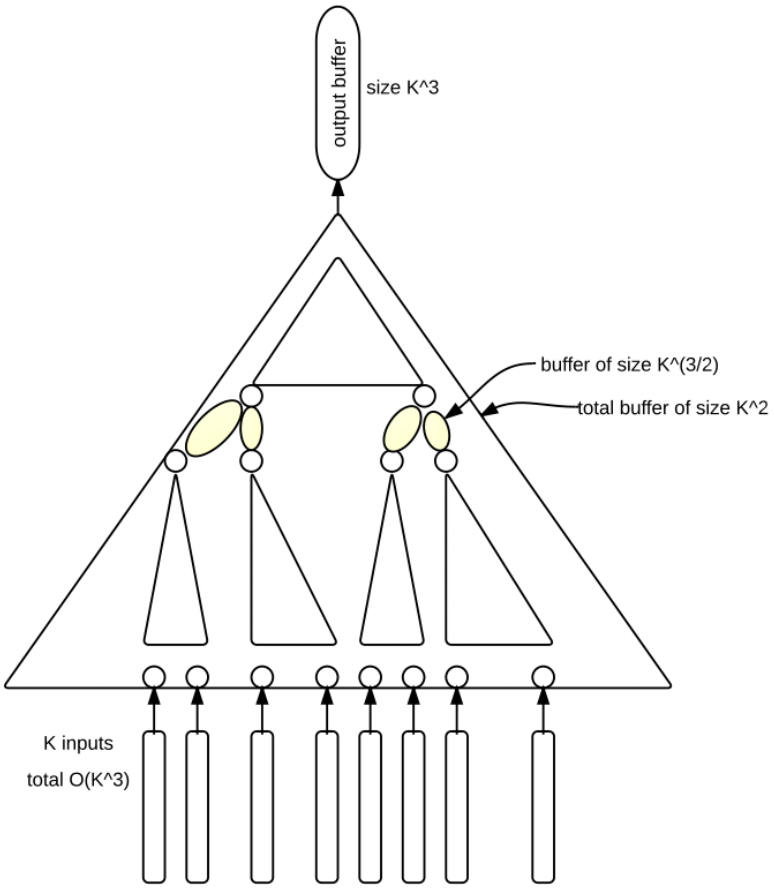

Figure 1: Memory layout for K-funnel sort.

(regular binary merge). Whenever one of the child buffers becomes empty, recursively fill it. We are only considering two input buffers and one resulting, merged buffer at a time. As described above, each leaf in the funnelsort tree has an input list which supplies the original data.

2.2 Distribution Sweeping via Lazy Funnelsort

The idea of distribution sweeping is that we can use funnelsort to sort, but it can also do a divide and conquer on the key value.

We can actually augment the binary merge of the filling algorithm to maintain auxilliary information about the coordinates. For example, we can, given a point, use this funnelsort to figure out its nearest neighbor.

3 Orthogonal 2D Range Searching

Given N points and some rectangles, we'd like to return which points are in which rectangles.

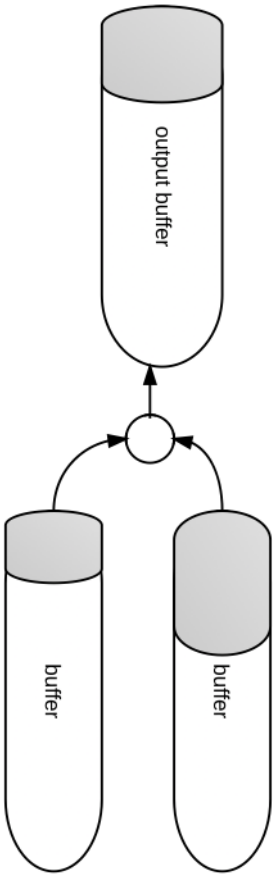

Figure 2: Filling a buffer.

3.1 Batched

The batched problem involves getting all N points and N rectangles first. We have all the information upfront, and we're also given a batch of queries and want to solve them all.

The optimal time is going to be $O(\frac{N}{B} \lg_{M/B}(\frac{N}{B}) + \frac{out}{B})$

3.1.1 Algorithm

First, count the number of rectangles containing each point (We need this to figure out what our buffer size needs to be in the funnel):

1. Sort the points and rectangle corners by x using lazy funnelsort.
2. Divide and conquer on x, where we mergesort by y and perform an upward sweepline algorithm. We have slabs L and R, as seen in Figure 3.

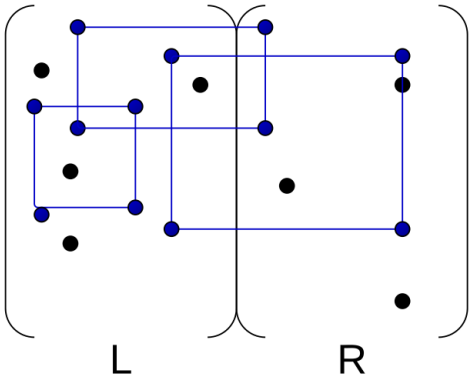

Figure 3: Left and Right slabs containing points and rectangles.

3. The problem of which points are in which rectangles are from when the rectangle completely spans one of the slabs, the rectangles in green in Figure 7, but not when the points of the rectangle are within that slab.

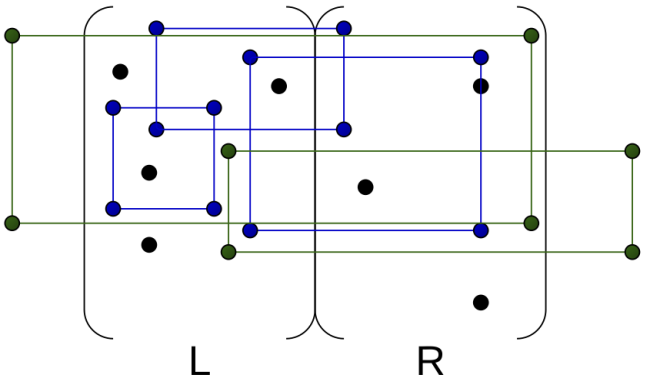

Figure 4: Slabs that also contain slabs that span either L or R.

4. Maintain the number of active rectangles (sliced by sweep line) that have a left corner in L and that completely span R. Increment C_L when we encounter a lower-left corner of a rectangle, and decrement when we encounter an upper-left corner.

5. Symmetrically maintain C_R on rectangles which span slab L.

6. When we encounter a point in L, add C_R to its counter. Similarly, add C_L to the counter of any point in R.

At this point, we can create buffers of the correct size. However, this is not optimal. The optimal solution is to carve the binary tree into linear size subtrees, and make each run in linear time.

3.2 Online Orthogonal 2D range search [2] [3]

It is possible to preprocess a set of N points in order to support range queries while incurring only $O(\log_B N + \frac{out}{B})$ external memory transfers.

We will consider three different kinds of range queries:

- 2-sided range queries : $(\leq x_q, \leq y_q)$
- 3-sided range queries : $([x_{q_{min}}, x_{q_{max}}], \leq y_q)$
- 4-sided range queries : $([x_{q_{min}}, x_{q_{max}}], [y_{q_{min}}, y_{q_{max}}])$

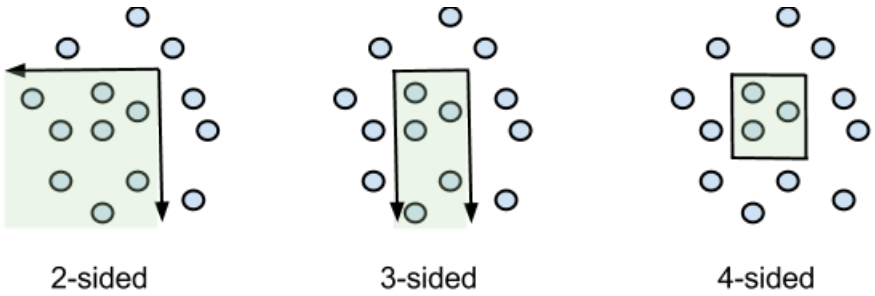

Figure 5: Depicts the three different types of range queries.

Only a small amount of extra space is needed to perform cache oblivious range queries. The below table compares the space required to perform range queries in the random access model, and the cache oblivious model.

Query Type	RAM	Cache Oblivious
2-sided	$O(N)$	$O(N)$
3-sided	$O(N)$	$O(N \lg N)$
4-sided	$O(N \frac{\lg N}{\lg \lg N})$	$O(N \frac{\lg^2 N}{\lg \lg N})$

3.2.1 2-sided range queries

First we will show how to construct a data structure that is capable of performing 2-sided range queries while using only linear space. Using this data structure, it will be possible to construct data structures for the 3-sided and 4-sided cases which have the desired space bounds.

At a high level, our data structure consists of a vEB BST containing our N points sorted on their y coordinate. Each point in the tree contains a pointer into a single array which has $O(N)$ size. This array contains one or more copies of each of the N points, and is organized in a special way. To report the points satisfying a 2-sided range query $(\leq x_q, \leq y_q)$ we find the point in the tree and follow a pointer into the array, and then scan the array until we encounter a point whose x coordinate is greater than x_q. The array's structure guarantees that we will only scan $O(out)$ points and that the distinct points scanned are precisely those satisfying the range query $(\leq x_q, \leq y_q)$.

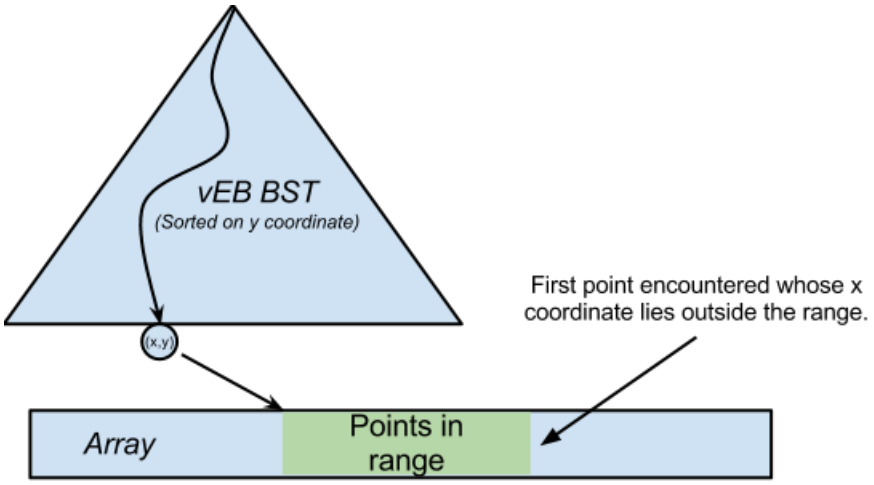

Figure 6: A high level description of the data structure which supports 2 sided range queries using only linear space.

We will proceed to describe the structure of the array and prove the following claims:

1. The high level procedure we described will find all points in $(\leq x_q, \leq y_q)$.
2. The number of scanned points is $O(out)$.
3. The array has size $O(N)$.

First Attempt

We will now outline two attempts, one unsuccessful, and one successful. We will begin by describing a simple structure for the array which will satisfy claims 1 and 2, but fail to have linear size (claim 3). This attempt will introduce concepts which will be useful in our second (and successful) attempt.

Definition 1. *A range $(\leq x_q, \leq y_q)$ is **dense** in an array S if*

$$|\{(x, y) \in S : x < x_q\}| \leq 2 \cdot \#\text{ points in } (\leq x_q, \leq y_q)$$

Definition 2. *A range is **sparse** with respect to S if it is not dense in S.*

Note that if a range is dense in an array S and S is sorted on x, then we can report all points within that range by scanning through S. Since the range is dense in S, we will scan no more than twice the number of points reported.

Our strategy will be to construct an array $S_0 S_1 \dots S_k$ so that for any query $(\leq x_q, \leq y_q)$ there exists an i for which that query is dense in S_i.

Consider the following structure:

Let $S_0 =$ all points (sorted by x coordinate)

Let $y_i =$ largest y where some query $(\leq x_q, \leq y)$ is sparse in S_{i-1}.

Let $S_i = S_{i-1} \cap (*, \leq y_i)$.

Then let our array be $S_0 S_1 \dots S_i$.

Now consider the query $(\leq x_q, \leq y_q)$. There is some i for which $y_i < y_q$, and for this i the query will be dense in S_{i-1}. For otherwise, y_q would be a larger value of y for which some query $(\leq x_q, \leq y)$ is sparse in S_{i-1}, contradicting the definition of y_i. Now we can construct our vEB BST and have each node point to the start of the subarray for which the corresponding query is dense. This data structure will allow us to find all points in a range while scanning only $O(out)$ points (satisfying claims 1 and 2). However, the array $S_0 S_1 \dots S_k$ may not have $O(N)$ size. Figure 7 shows an example in which the array's size will be quadratic in N.

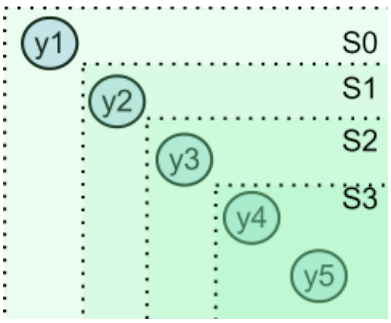

Figure 7: An example showing how our first attempt may construct an array with size quadratic in N.

Second (correct) Attempt

Let $x_i = \max x$ coordinate where $(\leq x, \leq y_i)$ is sparse in S_{i-1}.

Let $y_i =$ largest y where some query $(\leq x_q, \leq y)$ is sparse in S_{i-1}.

Let $P_{i-1} = S_{i-1} \cap (\leq x_i, *)$.

Let $S_i = S_{i-1} \cap ((*, \leq y_i) \cup (\geq x_i, *))$. (Note that this definition differs from that in our first attempt.)

Our array will now be $P_0 P_1 \dots P_{i-1} S_i \dots S_k$ (where S_j has $O(1)$ size for j between i and k.)

First, we show that the array is $O(N)$ in size. Notice that $|P_{i-1} \cap S_i| \leq \frac{1}{2}|P_{i-1}|$ since $(\leq x_i, \leq y_i)$ is sparse in S_{i-1}. Charge storing P_{i-1} to $(P_{i-1} - S_i)$. This results in each point only being charged once by a factor of $\frac{1}{1-\frac{1}{2}} = 2$. Which implies that the total space used is $\leq 2N$.

Next, notice that when an element is repeated in the array its x coordinate must be less than the x coordinate of the last seen point in the query. Therefore, by focusing on a monotone sequence of x coordinates we can avoid duplicates.

Finally, the total time spent scanning the array will be $O(out)$ because each point is repeated only $O(1)$ times. Therefore, this data structure can support $O(\log_B N + \frac{out}{B})$ 2-sided range queries while using only $O(N)$ space.

3.2.2 3-sided range queries

Maintain a vEB search tree in which the leaves are points keyed by x. Each internal node stores two 2-sided range query data structures containing the points in that node's subtree. Since each point appears in the 2-sided data structures of $O(\log N)$ internal nodes, the resulting structure uses $O(N \log N)$ space.

To perform the 3-sided range query $([x_{q_{min}}, x_{q_{max}}], \leq y_q)$, we first find the least common ancestor of $x_{q_{min}}$ and $x_{q_{max}}$ in the tree. We then perform the query $(\geq x_{q_{min}}, \leq y_q)$ in the left child's structure and the query $(\leq x_{q_{max}}, \leq y_q)$ in the right child's structure. The union of the points reported will be the correct answer to the 3-sided range query.

OPEN: 3-sided range queries with $O(\log_B N + \frac{out}{B})$ queries and $O(N)$ space.

3.2.3 4-sided range queries

Notice that we can easily achieve $O(\log_B N + \frac{out}{B})$ queries if we allow ourselves to use $O(N \log^2 N)$ space by constructing a vEB tree keyed on y which contains 3-sided range query structures.

However, it is possible to use only $O(N \frac{\log^2 N}{\log \log N})$ space by storing each point in only $O(\frac{\log N}{\log \log N})$ 3-sided range query structures.

Conceptually, we contract each $\frac{1}{2} \lg \lg N$ height subtrees into $\sqrt{\lg N}$ degree nodes. This results in a tree of height $O(\frac{\lg N}{\lg \lg N})$.

Now, as before, we store two 3-sided range query structures in each internal node containing the points in their subtree. Furthermore, we store $\lg N$ static search trees keyed by x on points in each interval of the node's $\sqrt{\lg N}$ children.

To perform the query $([x_{q_{min}}, x_{q_{max}}], [y_{q_{min}}, y_{q_{max}}])$ we first find the least common ancestor of $y_{q_{min}}$ and $y_{q_{max}}$ in the tree. Then we perform the query $([x_{q_{min}}, q_{q_{max}}], \geq y_{q_{min}})$ in the child node containing $y_{q_{min}}$, and the query $([x_{q_{min}}, q_{q_{max}}], \leq y_{q_{min}})$ in the child containing $y_{q_{max}}$. Finally, use the search tree keyed on x associated with the interval between the child containing $y_{q_{min}}$ and the child containing $y_{q_{max}}$ to perform the query$([x_{q_{min}}, x_{q_{max}}], *)$ for all the in-between children at once. This data structure supports $O(\log_B N + \frac{out}{B})$ 4-sided range queries, while using only $O(N \frac{\log^2 N}{\log \log N})$ space.

References

[1] Gerth Stølting Brodal and Rolf Fagerberg. Cache oblivious distribution sweeping. *In Proceedings of the 29th International Colloquium on Au- tomata, Languages, and Programming*, volume 2380 of Lecture Notes in Computer Science, pages 426-438, Malaga, Spain, July 2002.

[2] Lars Arge and Norbert Zeh. 2006. *Simple and semi-dynamic structures for cache-oblivious planar orthogonal range searching.* In Proceedings of the twenty-second annual symposium on Computational geometry (SCG '06). ACM, New York, NY, USA, 158-166.

[3] Lars Arge, Gerth Stlting Brodal, Rolf Fagerberg, and Morten Laustsen. 2005. *Cache-oblivious planar orthogonal range searching and counting.* In Proceedings of the twenty-first annual symposium on Computational geometry (SCG '05). ACM, New York, NY, USA, 160-169.