Fractional Cascading and Range Trees

Tomer Adar

Illustrations based on slides by Amani Shhadi, Yufei Zheng - 郑羽霏

What We See Today

- How to search the successor of x in k sorted lists in $O(\log n + k)$.
- How to do it for just t of these k lists, in $O(\log n + t)$.
  - Well, not exactly, only for “nice” subsets, and for some constant parameter d.
- A 2D range query data structure with $O(\log n)$ query time.
  - Improvement of the easier, $O(\log^2 n)$ solution.
- To print or store the result points, not only count them, we add $O(k)$ time where k is the number of output points.

Terminology

- All lists today are ordered, even if not specified.
- **Successor**: the lowest item in list A that is greater than x.
  - $(x, A) \to \min\{y \in A | y > x\}$.
  - c++: upper_bound(begin, end, x) -> successor's iterator (or end if not found).
  - If exists, must be greater than x.
- **Lower Bound**: like successor, but can be equal to x.
  - $(x, A) \to \min\{y \in A | y \geq x\}$.
  - c++: lower_bound(begin, end, x).
- **Predecessor**: like successor, but the other way (maximum before x).

Repeated Search Problem

L_1	6	7	26	54
L_2	2	21	29	60
L_3	9	13	31	45

- We have k sorted lists, n elements each.
- Query: find the successor of x in **all** lists.
- Trivial solution: binary search in every list independently.
  - $O(k \log n)$ query time ☹
  - No extra space – still $O(kn)$ ☺
- Can we do better?

X=20

Repeated Search Problem

L_1	6	7	26	54
L_2	2	21	29	60
L_3	9	13	31	45

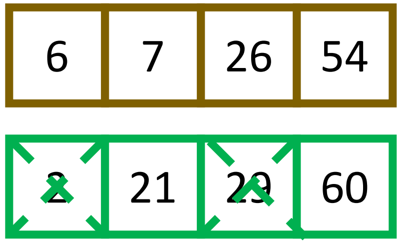
- We have k sorted lists, n elements each.
- Query: find the successor of x in **all** lists.
- Other solution: merge all lists, store k lower-bounds per entry.
  - $O(\log n + k)$ query time ☺
  - But $O(k^2 n)$ space ☹
- Can we have **both** $O(\log n + k)$ query time and $O(kn)$ space?

X=20

L	2	6	7	9	13	21	26	29	31	45	54	60
	0	0	1	2	2	2	2	3	3	3	3	4
	0	1	1	1	1	1	2	2	3	3	3	3
	0	0	0	0	1	2	2	2	2	3	4	4

Fractional Cascading – Simple Case

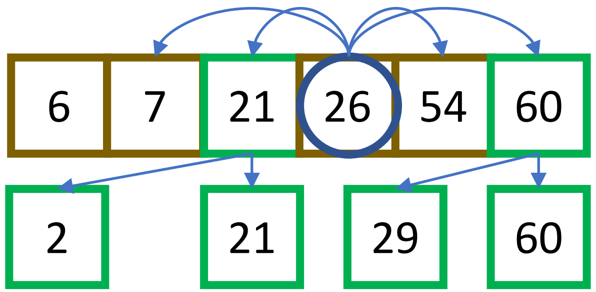
- Two lists, single x.
- We want to binary search once and then do $O(1)$ extra work.



- Take the first list.
- Take every-other-item in the second list.
- Merge them. Keep the second list.
- For each item in the merged list store its neighbors in each list:
  - Predecessor – the maximum-of-lower.
  - Lower Bound – the minimum-of-higher-or-equals.

Fractional Cascading – Simple Case

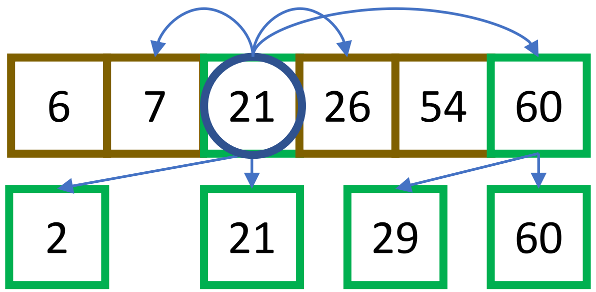
- Two lists, single x.
- We want to binary search once and then do $O(1)$ extra work.



- Take the first list.
- Take every-other-item in the second list.
- Merge them. Keep the second list.
- For each item in the merged list store its neighbors in each list:
  - Predecessor – the maximum-of-lower.
  - Lower Bound – the minimum-of-higher-or-equals.

Fractional Cascading – Simple Case

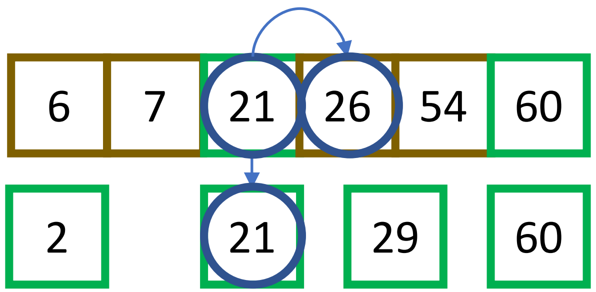
- Two lists, single x.
- We want to binary search once and then do $O(1)$ extra work.



- Take the first list.
- Take every-other-item in the second list.
- Merge them. Keep the second list.
- For each item in the merged list store its neighbors in each list:
  - Predecessor – the maximum-of-lower.
  - Lower Bound – the minimum-of-higher-or-equals.

Fractional Cascading – Simple Case

- Two lists, single x.
- We want to binary search once and then do $O(1)$ extra work.



- Search successor of x in the combined list.
- If belongs to list 1:
  - Found for list #1
  - For list #2: test both predecessor and lower bound, choose the better.
- If belongs to list 2:
  - The same.

Fractional Cascading - Another Step

- What if we had 3 lists?
  - Apply the 2-case on lists #2, #3. Call it list 2-3.
  - Apply it again for list 1 and the merged list. Call it list 1-(2-3)
  - Single binary search in list 1-(2-3).
  - We know how to find the successor of x in list 1 and also in list 2-3.
  - No need to search again! From list 2-3 we find the successor in list 2 and list 3.
  - The search in whole-list is $O(\log n)$, because list 1-(2-3) has at most $3n$ items.
  - Then just $O(1)$ work for each “layer”, which we have two of.
- Holds for $k \geq 2$: binary search - $O(\log kn)$, pointer work - $O(k)$.
- Space: $(n)_{\text{list }3} + \left(\frac{3}{2}n\right)_{\text{list }2-3} + \left(\frac{7}{4}n\right)_{\text{list }1-(2-3)} \leq 6n$ items.

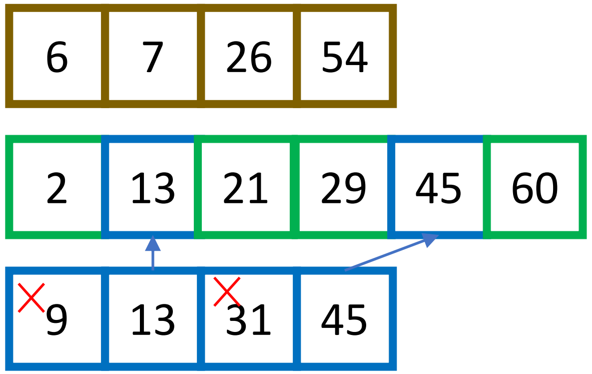
Fractional Cascading - Another Step

- What if we had 3 lists?

6	7	26	54
2	21	29	60
9	13	31	45

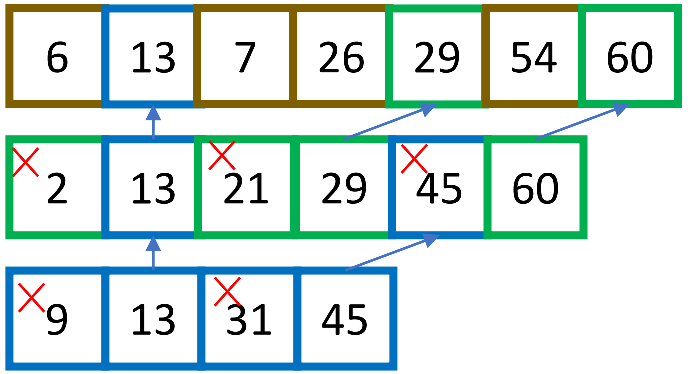
Fractional Cascading - Another Step

- What if we had 3 lists?



Fractional Cascading - Another Step

- What if we had 3 lists?



Fractional Cascading – The k-case

- How many items do we store for k lists?
- The last list has n items, which is less than $2n$.
- In every step we merge:
  - A brand-new list with n items and
  - **Every-other-item** of a list with less than $2n$ items.
- Also the merged list has less $2n$ items!
- **Space is $\boldsymbol{O(kn)}$**: we store k union-lists, each has less than $2n$ items.
- **Query time is $\boldsymbol{O(\log n + k)}$**:
  - $O(\log n)$ - for binary search in top union, which has less than $2n$ items.
  - $O(k)$ - a constant work to find the successors for every list.

Fractional Cascading – The General Case

- We have a directed, acyclic graph.
- Each vertex store a sorted list.
- Each edge's label is a segment $[a, b]$.
- The outgoing degree of a vertex is bounded by some constant $\deg$.
- Query: given x, walk along the edges whose label contain x. Find the successor of x in every list on the walked path.

Fractional Cascading – The General Case

- Basically the same solution.
- The k-list case was like a line tree, $d = 1$.
- Instead of ${}^{1}/_{2}$, we take every ${}^{1}/_{d+1}$ item in each list.
- And store $d + 1$ pointers “for the holes”, per item.
- Guaranteed: every $\frac{1}{d+1}$-union list has at most $2n$ items.
- Query complexity for t-path: $O(\log n + t \log d)$
  - After one search, every step require binary search in the $(d + 1)$-pointer list.

Range Queries

- In we have n points on a line (1D).
- How many points are there in a given segment $[a, b]$?
- What if the points are k-dimensional?

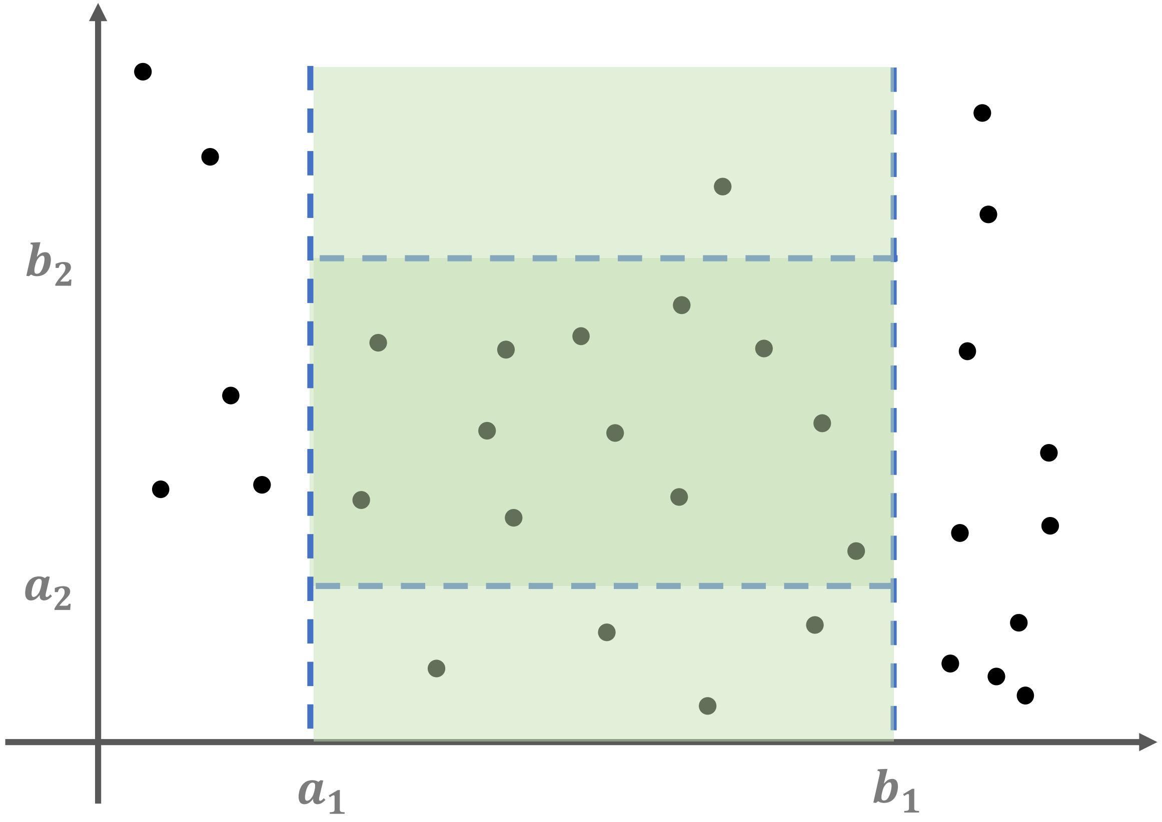
The 1D Case

- Balanced binary search tree
- Data only in leaves
- Internal node stores the maximum of its left subtree.
- Internal node also stores its subtree size.
- Query "count in [a,b]":
  - Search a - $O(\log n)$
  - Search b - $O(\log n)$
  - Sum the sizes of $O(\log n)$ subtrees in the path.

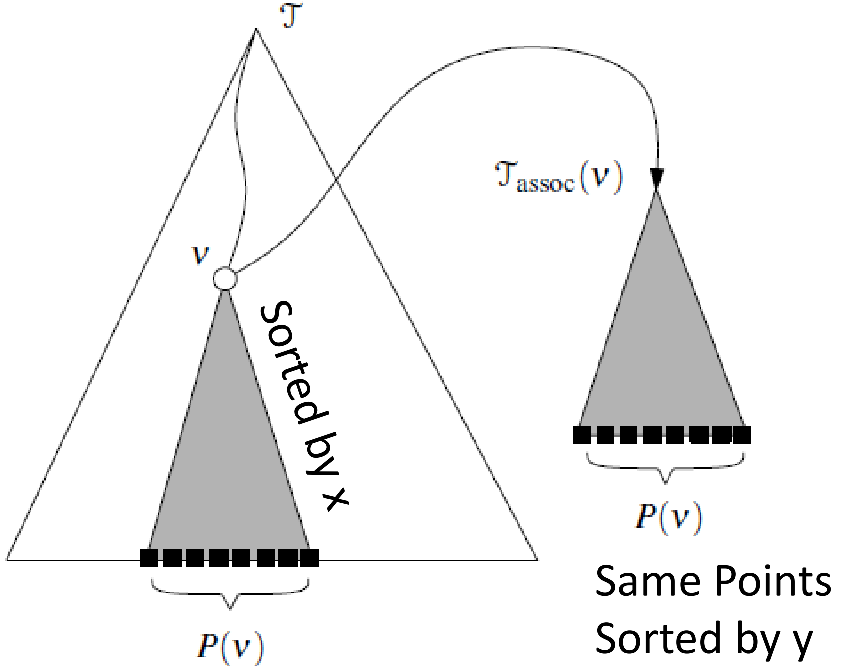
The dD case

Query$([\boldsymbol{a_1}, \boldsymbol{b_1}] \times [\boldsymbol{a_2}, \boldsymbol{b_2}])$

- x Query$([a_1, b_1])$
- y Query$([a_2, b_2])$



The dD case



- Range tree of range trees
- Sorted by a_1.
- Internal node stores a $(d-1)$-range-tree of the points in its subtree
  - without the first coordinate
- To search $[a_1, \dots, a_k] \times [b_1, \dots, b_k]$:
  - Find all $O(\log n)$ subtrees for which: $a_1 \le x_1 \le b_1$ holds for all points.
  - For each one, do a recursive query $[a_2, \dots, a_k] \times [b_2, \dots, b_k]$.
  - Sum the results.
- Total Time: $O\left(\log^d n\right)$, one log per dimension.

Can Do It Better!

- It is possible to do that in $O\left(\log^{d-1} n\right)$ time for $d > 1$.
- Do you have any idea?

Recall the first slide!

- The origin of the d-power is **repeated searches**.
- Every subtree is a **single-rooted, directed acyclic graph**.
- Can consolidate the searches using **Fractional Cascading**.
- Application: **2D Range Trees with $\boldsymbol{O(\log n)}$ query time**.

Repeated Searches

- In the 2D range tree
- We search **the same** $[a_2, b_2]$ range for $O(\log n)$ subtrees.
- It is exactly the use case of fractional cascading! ($\deg \leq 4$, why?)
- Application with:
    - First search in the rooted tree – all points sorted by y. $O(\log n)$.
    - Follow the successors along the way - $O(1)$ per subtree, $O(\log n)$ for all.
- Once got an $O(\log n)$ 2D range-tree, we use it as “bottom layer”
    - Still every dimension adds a log.
    - But we got the first two dimensions at the cost of one!

Range Trees

- Once got an $O(\log n)$ query time for the 2D case:
  - Make the dD range tree as before (tree of $(d-1)$D trees).
  - Every dimension adds a $\log n$ factor.
  - The base case is $d = 2$, whose query time is $O(\log n)$.
  - Giving $O\left(\log^{d-1} n\right)$ query time for $d \geq 2$.