CS 261: Graduate Data Structures

Week 7: Range search and augmented binary search trees

David Eppstein
University of California, Irvine

Spring Quarter, 2021

Ranking and unranking

In sorted arrays

$\text{Rank}(x)$ = the position of x in the array
(or the position it would go if added to the array)

Can be found by binary search

$\text{Unrank}(i)$ = the element at position i in the array

Trivial to compute as $\text{Array}[i]$

For example, $\text{Unrank}(n/2)$ is the median

They are inverse operations:

- $\text{Rank}(\text{Unrank}(i)) = i$,
  if i is in the range of array indexes
- $\text{Unrank}(\text{Rank}(x)) = x$,
  if x is one of the values stored in the array

In dynamic binary search trees

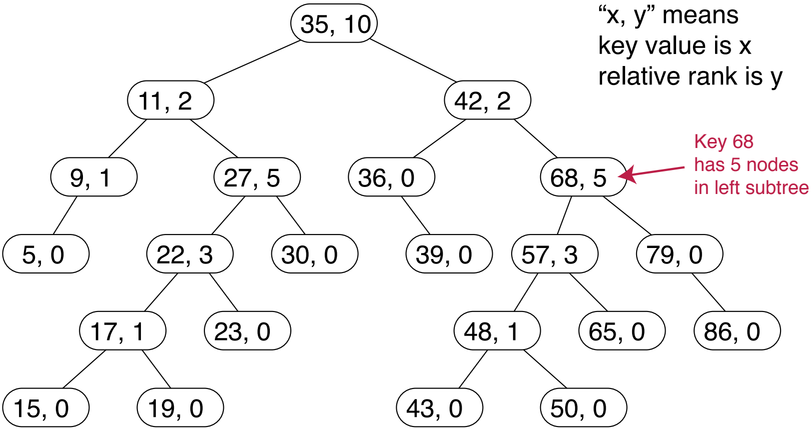
Rank and Unrank are well defined as the position of a given value in the sorted order, and the value at a given position

But it's not obvious how to compute them quickly!
It doesn't work to translate array search directly to trees

- In array binary search for Rank(x), we know the rank of each array cell
- In binary search trees, we cannot store a rank in each tree node, because each update would cause all later ranks to change, too many for fast updating
- There is no way to translate the trivial array Unrank algorithm into a tree algorithm

Augmented binary search trees

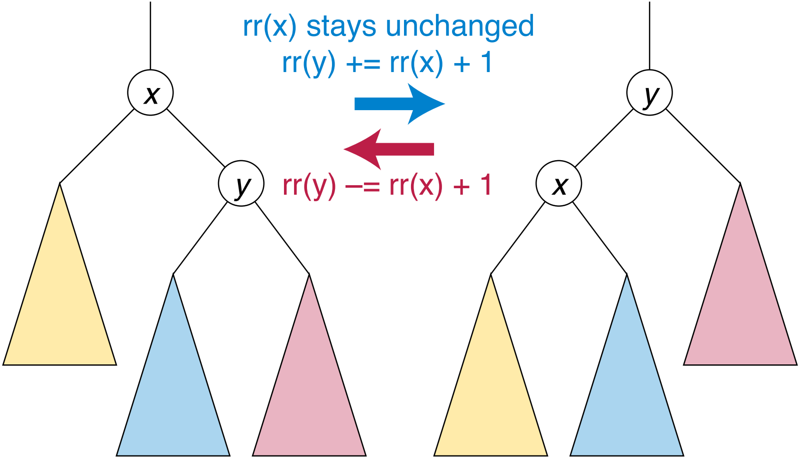
Store *relative rank* in each node: its position among it and its descendants = number of left descendants



Maintaining relative rank

On insertion or deletion: add or subtract one to all right ancestors

On rotation:



Ranking using relative ranks

Call the following recursive search with node = tree root:

```
def rank(x,node):
    if node == None:
        return 0
    else if x <= node.key:
        return rank(x,node.left)
    else:
        return rank(x,node.right) + node.relrank + 1
```

(In splay trees, add splay from last internal node on search path)

Unranking using relative ranks

Call the following recursive search with node = tree root:

```
def unrank(i,node):
    if i == node.relrank:
        return node.value
    else if i  < node.relrank:
        return unrank(i,node.left)
    else:
        return unrank(i - node.relrank - 1, node.right)
```

(In splay trees, add splay from last internal node on search path)

Ranking and unranking summary

By adding extra information (relative rank) to each node of a binary search tree, we can still update the tree in $O(\log n)$ time, and answer rank and unrank queries in the same time

Works with any rotation-based balanced binary search tree

Related recent research: Ranking and unranking dynamic sorted sets of n integers in the range $[0, n^c]$ can be done slightly faster: $O(\log n / \log \log n)$ per update or query

Pătrașcu and Thorup, "Dynamic Integer Sets with Optimal Rank, Select, and Predecessor Search", FOCS 2014, https://arxiv.org/abs/1408.3045

Range searching

Range searching

Find aggregate information about data elements within a query range [low,high] of values

(or within higher-dimensional regions)

- Range counting: Number of elements in range

  Compute ranks of left and right range endpoints and subtract

- Range reporting: List all elements in range

- Range minimum: Find minimum priority value in range

  (not minimum value – trivial as successor of left endpoint)

- Other more complex queries e.g. do a recursive range search on another attribute for elements within range

Range reporting

Call with node = tree root:

```
def report(low,high,node):
    if low < node.value:
        report(low,high,node.left)
    if low <= node.value <= high:
        output node.value
    if node.value < high:
        report(low,high,node.right)
```

Analysis of range reporting

Whenever we recurse into both children, we also output the node value

Every recursive call is one of:

- A node whose value is output
- A node on the search path for the low range endpoint (at which we search only the right child)
- A node on the search path for the high range endpoint (at which we search only the left child)

Time $= O(\text{number of nodes searched}) = O(\text{output size} + \log n)$

An algorithm whose time depends on output size and not just on input size is called "output sensitive".

Decomposable range search problems

Suppose:

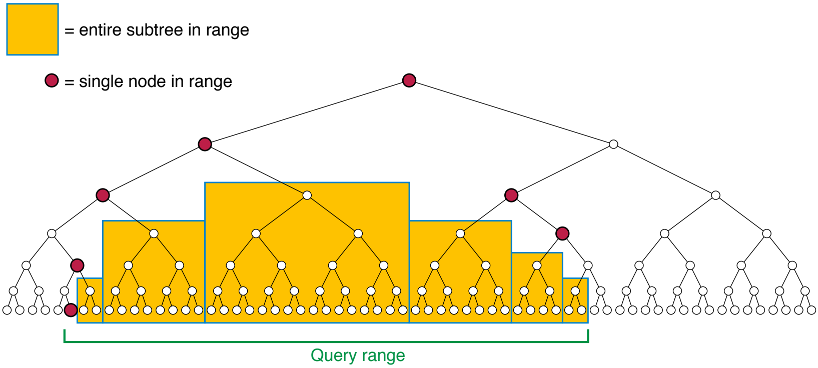
- We have a collection of key,value pairs with sorted keys
- An associative binary operation $\oplus$ operates on the values
- We want to find the result of applying $\oplus$ to the values whose keys are within a query range [low,high]

If we can decompose a range into disjoint sets, $S \cup T$, we can use $\oplus$ to combine results for each set: $\text{total} = \text{result}(S) \oplus \text{result}(T)$

Examples:

- Range counting, $\text{value} = 1$, $\oplus = \text{addition}$
- Range reporting, $\text{value}(x) = \{x\}$, $\oplus = \text{set union}$
- Range minimum, $\text{value} = \text{priority}$, $\oplus = \text{minimization}$

Partition of range into subtrees



Idea: search paths for range endpoints have length $O(\log n)$

We can decompose the range into $O(\log n)$ nodes on these two paths and $O(\log n)$ entire subtrees between them

Store $\oplus$ for each subtree, combine stored results for query total

Decomposable query algorithm

As we recurse, replace range endpoints by flag values $-\infty$ and $+\infty$ in subtrees for which endpoints are no longer relevant

Whole tree is in range when both endpoints are infinite

To query range [low,high] at a given node:

- If low $= -\infty$ and high $= +\infty$, return stored value for subtree
- If key $>$ high, return query(low, high, left child)
- If key $<$ low, return query(low, high, right child)
- Return query(low, $+\infty$, left child) $\oplus$ node's value $\oplus$ query($-\infty$, high, right child)

Time: $O(\log n)$ for operations with $\oplus$ time $O(1)$

Maintaining the stored subtree values

Whenever a node's stored subtree value might have changed

- We added or removed a descendant
- It was involved in a rotation

Recompute its subtree value as

left subtree value $\oplus$ right subtree value $\oplus$ node's value

Time per insertion or deletion $O(\log n)$

(under same assumptions on $\oplus$ time as for query)

Works for any balanced binary search tree

Range query summary

Using augmented search trees, we can:

Answer range counting or range minimization in time $O(\log n)$

Answer range reporting in time $O(\log n + \text{output})$

Handle insertions or deletions in time $O(\log n)$

Generalize to other decomposable range searching problems

Lower bound

Is it optimal?

We have seen that a very general class of dynamic range searching problems can be solved in time $O(\log n)$

Natural question: Is that the right time bound or can we do better?

Answer: we can prove $\Omega(\log n)$, for:

- Simple and natural range searching problem: range sum

  Data = ordered keys and numeric values

  Query = sum of values for key-value pairs with key in range

- A very general model of computing: cell probe model

  Only measure communication between CPU and memory

Prefix sum problem

Simplified model of the range sum problem

(for lower bounds, simpler problem $\Rightarrow$ stronger bound)

Maintain array $A[0] \ldots A[n-1]$ of numbers

Update(i, x): set $A[i]$ to new value x

Query(i): calculate $A[0] + A[1] + \cdots + A[i]$

(If we can handle these queries, we can also handle arbitrary range sum queries by subtracting prefix sums for start and end of range)

Prefix sum data structure

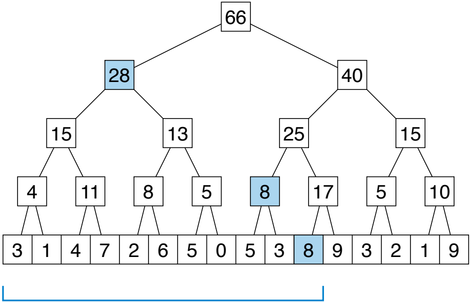
Store the values $A[i]$ in an array

Build binary tree with cells of A as leaves

Each node stores sum of descendants

Each update changes sums on leaf–root path

Query value $=$ sum of $O(\log n)$ tree nodes



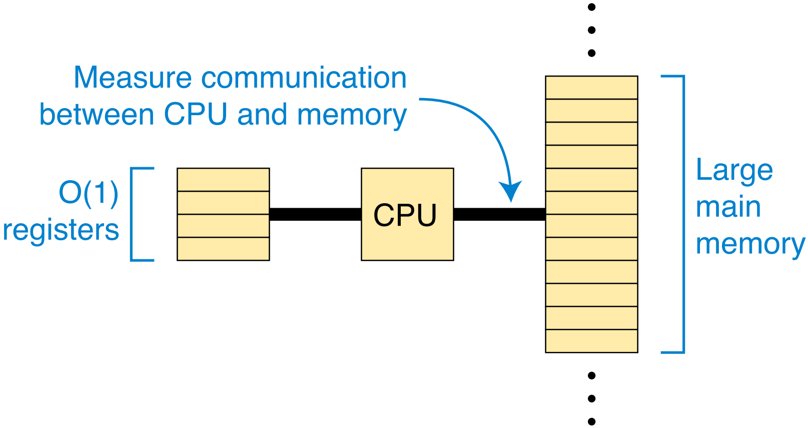
Query(10) = 3 + 1 + 4 + ... + 8 = 28 + 8 + 8 = 44

(T can be stored in a second array, depicted by left–right position of nodes, with tree structure derived from array index as in binary heap)

Cell probe model of computing

Central processor has $O(1)$ registers, each holding one word (binary value of length $w \geq \log_2 n$); memory has up to 2^w words

We count only steps that move a word between CPU and memory
$\Rightarrow$ lower bound doesn't depend on what other steps are allowed



Fitting prefix sums to cell probe model

We are going to prove a lower bound for
prefix sums of n w-bit binary numbers

(representation size of the input values should be
the same as the word size of the computer)

We will use $n =$ a power of two (unrelated to word size)

To avoid questions of integer overflow,
we will assume all arithmetic is modulo 2^w

(just do binary addition and ignore overflows)

Goal: Find a sequence of prefix sum operations that forces any
correct data structure to do a lot of CPU–memory communication

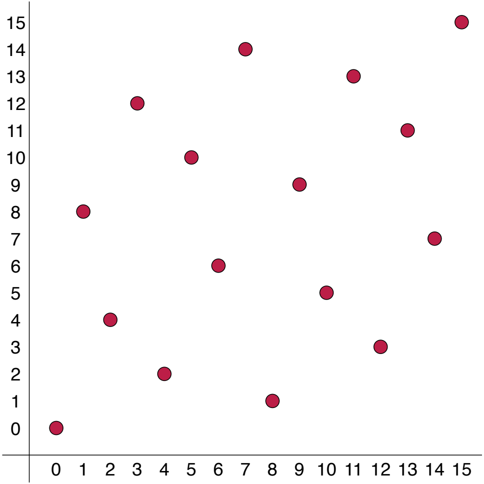
A special permutation of n

Assume $n = 2^k$

Define "bit reversal permutation" $r(i)$:

- Write i as a k-bit binary number
- Reverse the bits
- Interpret the result as a binary number

E.g. for $k = 8$,
$222_{10} = 11011110_2$
becomes
$01111011_2 = 123_{10}$



Computing sequence of bit-reversals

```
def bitrev(k):
    if k == 0:
        return [0]
    L = bitrev(k-1)
    return [2*x for x in L] + [2*x+1 for x in L]
```

Each value in the second half of the sequence is one plus the corresponding value in the first half

A difficult sequence of prefix-sum operations

Initialize all data values $A[i]$ to zero, then:

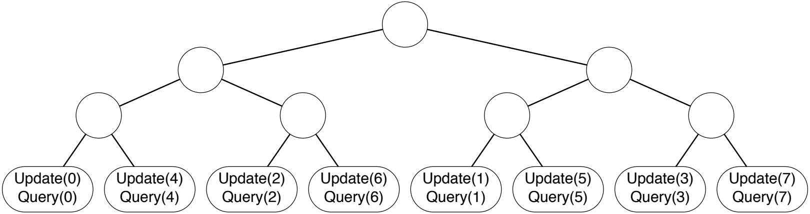
For each index i in `bitrev[k]`:

- Set $A[i]$ to be a random w-bit number
- Query the prefix sum $A[0] + \cdots + A[i]$

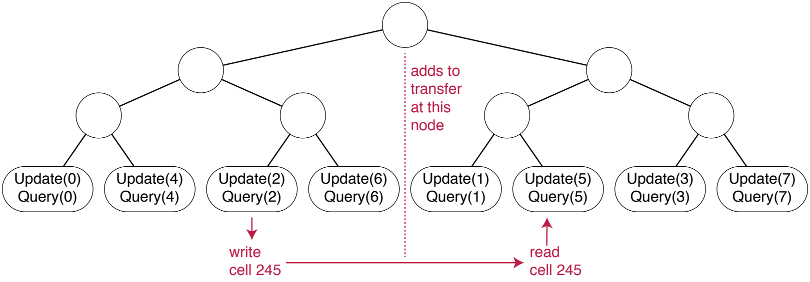
E.g. when $n = 8$, $k = 3$, we perform the operations
Update(0,random), Query(0), Update(4,random), Query(4),
Update(2, random), Query(2), Update(6,random), Query(6),
Update(1,random), Query(1), Update(5,random), Query(5),
Update(3,random), Query(3), Update(7,random), Query(7)

A binary tree on the sequence of operations

This is not a data structure! It's just a mathematical tree that we will use in the lower bound proof.



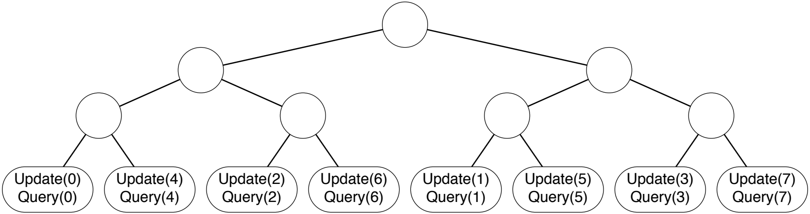
Information transfer



For any data structure for prefix sums, and any node x of this tree, define the *information transfer* of x to be the number of times an operation in the right descendants of x reads a memory cell that was last written during the operations in the left descendants of x

Each memory read contributes to information transfer at ≤ 1 node
$\Rightarrow$ total number of read steps $\geq$ total information transfer

Information transfer $\geq$ descendants/2



Information transfer = number of times an operation in node's right descendants reads a memory cell last written on the left

Let $d = \#\text{descendants}/2 = \#$ left updates $= \#$ right queries

There are 2^{wd} different possible values for the updates on the left, each of which would produce different query results on the right

(Independently from information derived from non-transfer reads)

$\Rightarrow$ for correct queries, information transfer $\geq d$

Finishing the lower bound

Information transfer at root node of tree: $\geq n/2$

Information transfer at ith level of tree:
2^i nodes with transfer $\geq n/2^{i+1}$, total $\geq n/2$

Total over whole tree: $\geq (n/2) \times \#$ levels $= (n/2)\log_2 n$

There are $2n$ prefix sum operations (updates and queries together)
$\Rightarrow$ average number of memory reads per operation $\geq \frac{1}{4}\log_2 n$

Every prefix sum data structure that fits into the cell probe model of computation requires $\Omega(\log n)$ time per operation

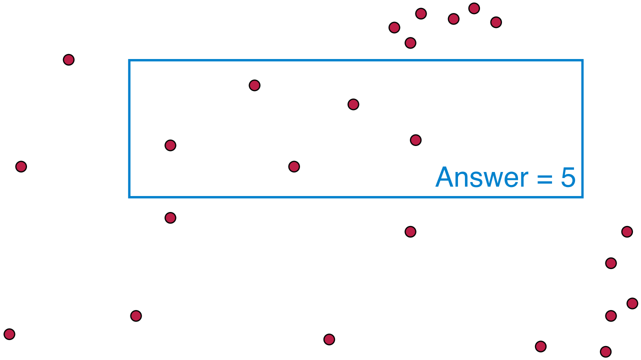
$\Rightarrow$ same is true for dynamic range sum data structures

Multi-level range search

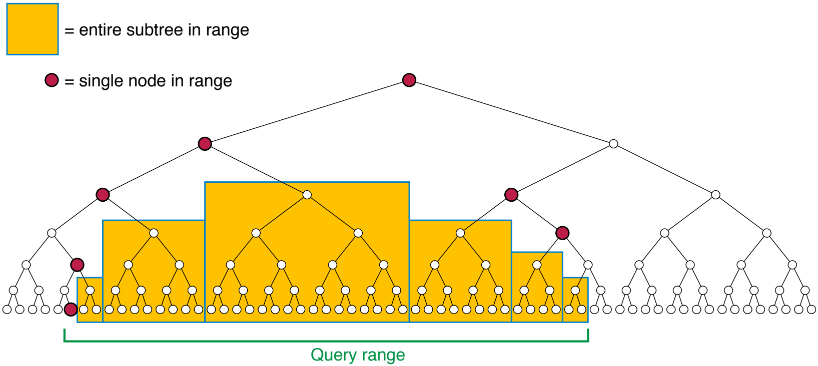
Example: Rectangular range counting

Data: 2d points represented as (x, y) coordinate pairs

Query: How many points are inside a given rectangle?



Binary search tree on x-coordinates



Query range: left and right x-coordinates of rectangle

Decomposes the points whose x-coordinate is in range into

- $O(\log n)$ individual points
- $O(\log n)$ larger sets of points

Multi-level structure

On each node of the x-coordinate tree, store

- the set of all (x, y) points descending from that node
- organized as a one-dimensional range counting structure on y-coordinates (for instance a sorted array)

To count points in a query rectangle:

- Perform query on x-range of rectangle
- For each individual point (x, y) found by query:

  Test whether y is in range
- For each subtree identified by query:

  Use 1d structure at subtree root to count descendants whose y coordinate is in range
- Add the results and return the total

Multi-level analysis

If x-tree is balanced $\Rightarrow$ each point contributes to y-structures in $O(\log n)$ ancestors $\Rightarrow$ total space is $O(n \log n)$

Each rectangle query makes $O(\log n)$ calls to one-dimensional y-structures $\Rightarrow$ query time is $O(\log^2 n)$

Doesn't work with rotation-based dynamic balanced trees but can be made dynamic using weight-balanced trees (when we rebuild a subtree we also rebuild the recursive structures stored in its nodes)

Fractional cascading

Related binary searches

In the multi-level structure for rectangular range counting, each query does $O(\log n)$ binary searches:

- In one-dimensional structures stored at certain tree nodes
- All searching for the same y-coordinates (top and bottom coordinates of query rectangle)
- In a *related* sequence of nodes (children of the nodes on a tree path)

Goal of fractional cascading: Speed up multiple related binary searches without paying too big a penalty in space

A simpler multi-binary-search problem

Data: k sorted lists of numbers $S_0, S_1, \ldots S_{k-1}$

Total length: $n = |S_0| + |S_1| + \cdots + |S_{k-1}|$

No repeated values, even in different lists

Query: find the successors of a given number q in each list

(s_i = successor of q in list S_i)

Example

Data:

- $S_0 = [0, 10, 20, 30, 40, 50, 60, 70]$
- $S_1 = [1, 2, 13, 25, 27, 51, 57]$
- $S_2 = [21, 22, 31, 32, 33, 41, 99]$
- $S_3 = [67, 68, 69]$

Total length $n = 8 + 7 + 7 + 3 = 25$

Query for $q = 24$ would find
$s_0 = 30 \quad s_1 = 25 \quad s_2 = 31 \quad s_3 = 67$

Naïve solutions

Do the binary searches separately

Space $= O(n)$ for storing each S_i as a sorted list

Query time $= O(k \log n)$ for k binary searches

Merge into one list

For each value x, store k-tuple of successors
for queries that return x as their smallest value

0:(0,1,21,67), 1:(10,1,21,67), 2:(10,2,21,67), 10:(10,13,21,67),
13:(20,13,21,67), 20:(20,25,21,67), 21:(30,25,21,67), ...

Binary search in merged sorted array + look up k-tuple

Space $O(kn)$, query time $O(k + \log n)$

Fractional cascading

Working backwards through the sequence of lists S_i, construct T_i: merged structure for (S_i + half the elements of T_{i+1})

Choosing the half of the elements that are in odd-numbered positions e.g. if $T = 1, 2, 3, 5, 7, 11, 20$ then $\frac{1}{2}T = 2, 5, 11$

So T_i consists of:

- A sorted array of the merged items from $S_i + \frac{1}{2}T_{i+1}$
- A dictionary mapping each merged item x to a pair (a, b) where one of a or b is x, and the other one is the successor of x in the other merged list
- When there is no successor in the other list, use $+\infty$

Example

- $S_3 = 67, 68, 69$ $\quad T_3 = S_3$ (nothing to merge) $\quad$ Half elements: 68
- $S_2 = 21, 22, 31, 32, 33, 41, 99$
- $T_2 =$ 21:(21,68), 22:(22,68), 31:(31,68), 32:(32,68), 33:(33,68), 41:(41,68), 68:(99,68), 99:(99,$+\infty$)
- Half the elements of T_2: 22, 32, 41, 99
- $S_1 = 1, 2, 13, 25, 27, 51, 57$
- $T_1 =$ 1:(1,22), 2:(2,22), 13:(13,22), 22:(25,22), 25:(25,32), 27:(27,32), 32:(51,32), 41:(51,41), 51:(51,99), 57:(57,99), 99:($+\infty$,99)
- Half the elements of T_1: 2, 22, 27, 41, 57
- $S_0 = 0, 10, 20, 30, 40, 50, 60, 70$
- $T_0 =$ 0:(0,2), 2:(10,2), 10:(10,22), 20:(20,22), 22:(30,22), 27:(30,27), 30:(30,41), 40:(40,41), 41:(50,41), 50:(50,57), 57:(60,57), 60:(60,$+\infty$), 70:(70,$+\infty$)

Searching fractionally cascaded lists

To find the successors of q:

- Binary search for successor t_0 in merged list T_0
- Set $i = 0$
- Then, repeat:
  - Use dictionary for T_i to find the pair (a, b)
    where $a = s_i =$ successor in S_i
    and b is successor in $\frac{1}{2}T_{i+1}$
  - Output s_i
  - Let c be the (skipped) element of T_{i+1} just before b
  - If $q < c$ then $t_{i+1} = c$ else $t_{i+1} = b$
  - Set $i = i + 1$

Example (continued)

To search for the successor of $q = 24$:

- Binary search in T_0 finds successor t_0: 27:(30,27)
- Output $s_0 = 30$, successor in S_0
- Successor in T_1 might be either 27 or previous item, 25
- Because $q < 25$, successor in T_1 is 25:(25,32)
- Output $s_1 = 25$, successor in S_1
- Successor in T_2 might be either 32 or previous item, 31
- Because $q < 31$, successor in T_2 is 31:(31,68)
- Output $s_2 = 31$, successor in S_2
- Successor in T_3 might be either 68 or previous item, 67
- Because $q < 67$, successor in T_3 is 67
- Output $s_3 = 67$, successor in S_3

Fractional cascading analysis

Query time

One binary search + $O(1)$ for each list after the first

Total $O(k + \log n)$

Space and set-up time

Each element of S_i contributes 1 to the length of T_i, $\frac{1}{2}$ to the length of T_{i-1}, $\frac{1}{4}$ to the length of T_{i-2}, ...

So the total space and total set-up time is $O(n)$

Best combination of time and space from naïve solutions

Also works for multi-level search trees, for example rectangular range counting with $O(n \log n)$ space and $O(\log n)$ query time

Summary

Summary

- Ranking and unranking operations; efficient dynamic implementation by augmenting search tree with relative ranks
- Types of range searching problems including range counting, range reporting, range minimum, and range sum; decomposable problems using associative binary operation
- Dynamic range searching by augmenting search tree with value of its subtree and decomposing range into a logarithmic number of subtrees and individual nodes
- Cell probe model of computing and lower bound on dynamic prefix sums
- Multi-level range search and multi-level augmented binary search trees
- Fractional cascading