CS 261: Data Structures
Week 6–7: Binary search
Lecture 7a: Augmented search trees

David Eppstein
University of California, Irvine

Spring Quarter, 2024
Ranking and unranking
In sorted arrays

\[ \text{Rank}(x) = \text{the position of } x \text{ in the array} \]
(or the position it would go if added to the array)
Can be found by binary search

\[ \text{Unrank}(i) = \text{the element at position } i \text{ in the array} \]
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 $\text{Rank}(x)$, 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

"x, y" means key value is x relative rank is y

Key 68 has 5 nodes in left subtree
Maintaining relative rank

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

On rotation:

- \( rr(x) \) stays unchanged
- \( rr(y) \) += \( rr(x) + 1 \)
- \( rr(y) \) -= \( rr(x) + 1 \)
Call the following recursive search with node = tree root:

```python
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:

```python
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)
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.

Range searching
Range searching

Find aggregate information about data elements within a query range \([\text{low}, \text{high}]\) 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
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”.
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 $[\text{low, high}]$

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, value = 1, $\oplus = \text{addition}$
- Range reporting, value($x$) = $\{x\}$, $\oplus = \text{set union}$
- Range minimum, value = priority, $\oplus = \text{minimization}$
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 \([\text{low}, \text{high}]\) at a given node:

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

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 ⊕ right subtree value ⊕ node’s value

Time per insertion or deletion $O(\log n)$
(under same assumptions on ⊕ 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 bounds
Lower bounds on data structures

We have seen:

- Optimality of binary heap for comparison-model priority queues
  Based on the ability to sort using heaps
  Can be sidestepped by using integer arithmetic and array indexing instead of only comparisons (e.g. flat trees)

- Impossibility of nontrivial set disjointness
  Based on unproven assumption (SETH)

This time: Lower bounds for range search
Proven rigorously in a very general computational model
Are augmented search trees 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
Warmup interview question: Static range sums

You are given an array of \( n \) numbers

Problem: process it so you can quickly find the range sum

\[
\]

Solution

Store an array of prefix sums

\[
\]

Return \( PS[j] - PS[i - 1] \)

Linear space and preprocessing, constant time per query
Prefix sum problem

Simplified version of the range sum problem
   (for lower bounds, simpler problem ⇒ stronger bound)

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

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


(If these operations are hard, so are the more general operations of insertion + deletion + range sum)
Log-time solution

Build a perfectly balanced binary tree with array $A$ at its leaves

Each internal node stores sums of its two children

Query($i$): sum up left children on search path to $A[i]$

Update: recompute node sums on path to root

Claim: No other data structure can achieve better $O$-notation

We need to define what an “other data structure” might be
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.

Measure communication between CPU and memory.

O(1) registers

CPU

Large main memory

[Diagram of cell probe model with CPU and main memory connected by communication lines.]
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

To compute a sequence of length $2^k$, consisting of all $k$-bit numbers in bit-reversed order, compute the same sequence recursively for $k - 1$ and use it twice:

```
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]
```

$[0] \rightarrow [0, 1] \rightarrow [0, 2, 1, 3] \rightarrow [0, 4, 2, 6, 1, 5, 3, 7] \rightarrow ...$

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.
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 \( \leq 1 \) 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 \( d = \#\text{descendants}/2 = \# \text{left updates} = \# \text{right queries} \)

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

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

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