Lecture 9: Augmentation

This lecture covers augmentation of data structures, including

- easy tree augmentation
- order-statistics trees
- finger search trees, and
- range trees

The main idea is to modify "off-the-shelf" common data structures to store (and update) additional information.

Easy Tree Augmentation

The goal here is to store $x.f$ at each node x, which is a function of the node, namely f(subtree rooted at x). Suppose $x.f$ can be computed (updated) in $O(1)$ time from x, *children* and *children*.f. Then, modification a set S of nodes costs $O(\#$ of ancestors of $S)$ to update $x.f$, because we need to walk up the tree to the root. Two examples of $O(\lg n)$ updates are

- AVL trees: after rotating two nodes, first update the new bottom node and then update the new top node
- 2-3 trees: after splitting a node, update the two new nodes.
- In both cases, then update up the tree.

Order-Statistics Trees (from 6.006)

The goal of order-statistics trees is to design an Abstract Data Type (ADT) interface that supports the following operations

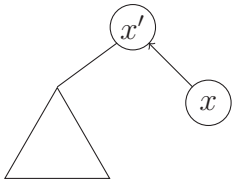
- insert(x), delete(x), successor(x),
- rank(x): find x's index in the sorted order, i.e., # of elements $< x$,
- select(i): find the element with rank i.

We can implement the above ADT using easy tree augmentation on AVL trees (or 2-3 trees) to store subtree size: $f(\text{subtree}) = \#$ of nodes in it. Then we also have $x.size = 1 + \sum c.size$ for c in $x.children$.

As a comparison, we cannot store the rank for each node. In that case, insert$(-\infty)$ will change the ranks for *all* nodes.

rank(x) can be computed as follows:

- initialize rank $= x.left.size + 1$ [1]
- walk up from x to root, whenever taking a left move $(x \to x')$, rank $+=$ $x'.left.size + 1$



select(i) can be implemented as follows:

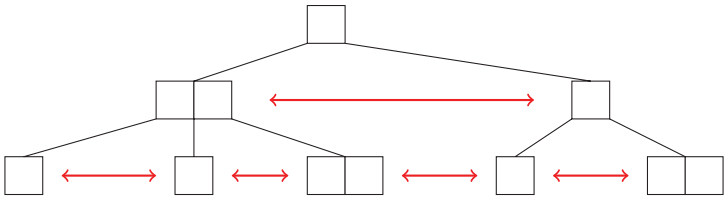
- $x =$ root
- rank $= x.left.size + 1$ [2]
- if $i =$ rank: return x
- if $i <$ rank: $x = x.left$
- if $i >$ rank: $x = x.right$, $i \mathrel{-}=$ rank

Finger Search Trees

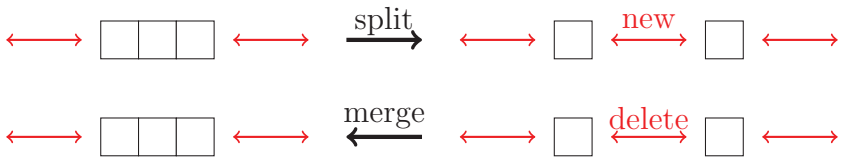
The goal of finger search trees [Brown and Tarjan, 1980] is that, if we already have node y, we want to search x from y in $O\left(\lg|\text{rank}(y) - \text{rank}(x)|\right)$ time. Intuitively, we would like the search of x to be fast if we already have a node y that is close to x.

One idea is to use **level-linked** 2-3 trees, where each node has pointers to its next and previous node on the same level.

[1] omit the +1 term if indices start at 0.
[2] same as above.



The level links can be maintained during split and merge.



We store all keys in the leaves. Non-leaf nodes do not store keys; instead, they store the min and max key of the subtree (via easy tree augmentation).

Then the original top-down search(x) (without being given y) can be implemented as follows:

- start from the root, look at min & max of each child c_i
- if $c_i.\min \le x \le c_i.\max$, go down to c_i
- if $c_i.\max \le x \le c_{i+1}.\min$, return $c_i.\max$ (as predecessor) or $c_{i+1}.\min$ (as successor)

search(x) from y can be implemented as follows. Initialize v to the leaf node containing y (given), and then in a loop do

- if $v.\min \le x \le v.\max$ (this means x is in the subtree rooted at v), do top-down search for x from v and return
- elif $x < v.\min$: $v = v.prev$ (the previous node in this level)
- elif $x > v.\max$: $v = v.next$ (the next node in this level)
- $v = v.parent$

Analysis. We start at the leaf level, and go up by 1 level in each iteration. At step i, level link at height i skips roughly c^i keys (ranks), where $c \in [2, 3]$. Therefore, if $|\text{rank}(y) - \text{rank}(x)| = k$, we will reach the subtree containing x in $O(\lg k)$ steps, and the top-down search that follows is also $O(\lg k)$.

Orthogonal Range Searching and Range Trees

Suppose we have n points in a d-dimension space. We would like a data structure that supports **range query** on these points: find all the points in a give **axis-aligned box**. An axis-aligned box is simply an interval in 1D, a rectangle in 2D, and a cube in 3D.

To be more precise, each point x_i (for i from 1 to n) is a d-dimension vector $x_i = (x_{i1}, x_{i2}, \ldots, x_{id})$. Range-query$(a, b)$ takes two points $a = (a_1, a_2, \ldots, a_d)$ and $b = (b_1, b_2, \ldots, b_d)$ as input, and should return a set of indices $\{i \mid \forall j, a_j \leq x_{ij} \leq b_j\}$.

1D case

We start with the simple case of 1D points, i.e., all x_i's and a and b are scalars.

Then, we can simply use a sorted array. To do range-query(a, b), we simply perform two binary searches for a and b, respectively, and then return all the points in between (say there are k of them). The complexity is $O(\lg n + k)$.

Sorted arrays are inefficient for insertion and deletion. For a dynamic data structure that supports range queries, we can use finger search tree from the previous section. Finger search trees support efficient insertion and deletion. To do range-query(a, b), we first search for a, and then keep doing finger search to the right by 1 until we exceed b. Each finger search by 1 takes $O(1)$, so the total complexity is also $O(\lg n + k)$.

However, neither of the above approaches generalizes to high dimensions. That's why we now introduce range trees.

1D range trees

A 1D range tree is a complete binary search tree (for dynamic, use an AVL tree). Range-query(a, b) can be implemented as follows:

- search(a)
- search(b)
- find the least common ancestor (LCA) of a and b, v_{split}
- return the nodes and subtrees "in between". There are $O(\lg n)$ nodes and $O(\lg n)$ subtrees "in between".

Analysis. $O(\lg n)$ to implicitly represent the answer. $O(\lg n + k)$ to output all k answers. $O(\lg n)$ to report k via subtree size augmentation.

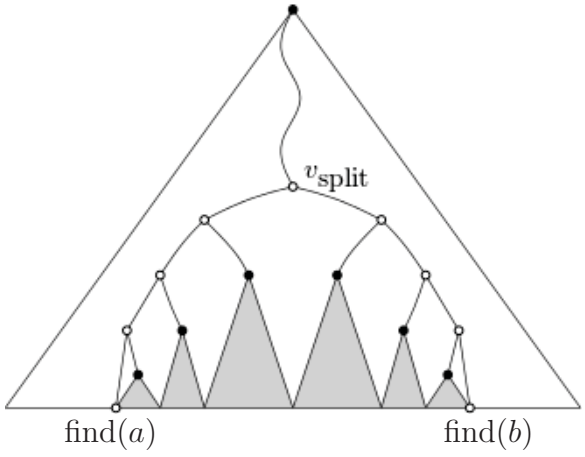


Figure 1: 1D range tree. Range-query(a, b) returns all hollow nodes and shaded subtrees. Image from Wikipedia http://en.wikipedia.org/wiki/Range_tree

2D range trees

A 2D range tree consists of a primary 1D range tree and many secondary 1D range trees. The primary range stores all points, keyed on the first coordinate. Every node v in the primary range tree stores all points in v's subtree in a secondary range tree, keyed on the second coordinate.

Range-query(a, b) can be implemented as follows:

- use the primary range tree to find all points with the correct range on the first coordinate. Only implicitly represent the answer, so this takes $O(\lg n)$.
- for the $O(\lg n)$ nodes, manually check whether their second coordinate lie in the correct range.
- for the $O(\lg n)$ subtrees, use their secondary range tree to find all points with the correct range on the second coordinate.

Analysis. $O(\lg^2 n)$ to implicitly represent the answer, because we will find $O(\lg^2 n)$ nodes and subtrees in secondary range trees. $O(\lg^2 n + k)$ to output all k answers. $O(\lg^2 n)$ to report k via subtree size augmentation.

Space complexity is $O(n \lg n)$. The primary subtree is $O(n)$. Each point is duplicated up to $O(\lg n)$ times in secondary subtrees, one per ancestor.

d-D range trees

Just recurse: primary 1D range tree $\to$ secondary 1D range trees $\to$ tertiary 1D range trees $\to \cdots$

Range-query complexity: $O(\lg^d n + k)$.
Space complexity: $O(n \lg^{d-1} n)$.

See 6.851 for Chazelle's improved results: $O(\lg^{d-1} n + k)$ range-query complexity and $O\left(n \left(\frac{\lg n}{\lg \lg n}\right)^{d-1}\right)$ space complexity.

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2015