ADVANCED ALGORITHMS · SESSION 2015–16

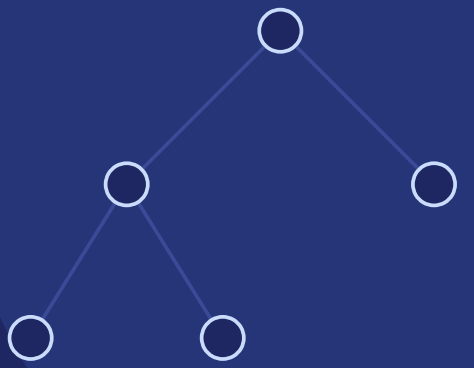
Lecture 10

Self-Adjusting Data Structures

Faculty: K.R. Chowdhary

Professor of CS

Splay trees, persistent data structures, and multidimensional (k-d) trees



Splay Trees

A splay tree is a self-adjusting binary search tree with one extra property: recently accessed elements are quick to access again. It performs insertion, look-up, and removal in O(log n) amortized time — and for many non-random sequences, outperforms other search trees even when the access pattern is unknown in advance.

THE ONE EXTRA OPERATION: SPLAYING

Splaying a tree for a given element rearranges it so that element ends up at the root — typically via a standard BST search followed by a specific sequence of tree rotations that walk the element up to the top.

A top-down variant instead combines the search and the reorganization into a single pass.

Advantages and Disadvantages

Advantages

- Self-optimizing: frequently accessed nodes migrate near the root for faster access.
- Worst-case height is O(n), but average height is O(log n) — great for caches and garbage collection.
- Naturally supports a persistent-data-structure version, useful in functional programming.
- Handles duplicate keys well and preserves their relative order — like a stable sort.

Disadvantage

A splay tree's height can become linear — e.g. right after accessing all n elements in strictly ascending order. Since height bounds worst-case access time, a single operation can then be expensive, even though the amortized cost over a sequence remains O(log n).

Anatomy of a Splay Step

Splaying is carried out as a sequence of splay steps, each moving x closer to the root. Which step applies depends on three factors:

1. Whether x is the left or right child of its parent p
2. Whether p is the root or not
3. If not, whether p is the left or right child of its parent g (the grandparent of x)

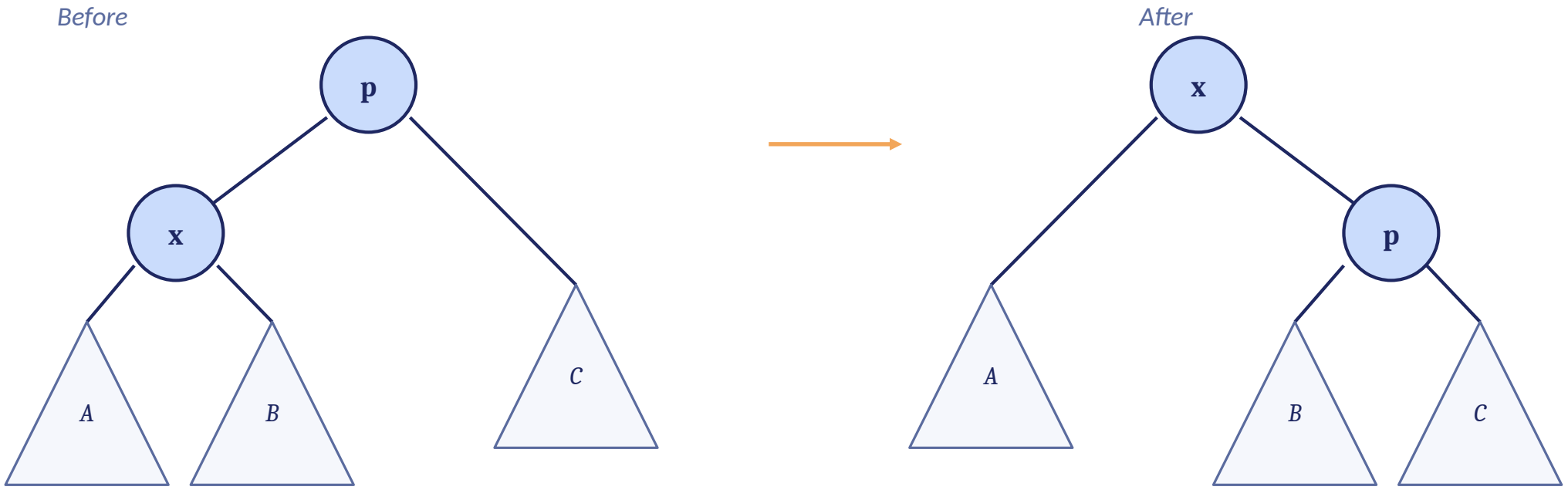
THE THREE STEP TYPES

Zig • Zig-Zig • Zig-Zag — each has a mirrored left/right case; we show one side of each.

Whenever a splay step runs, the great-grandparent gg must be updated to point at x — or, if gg is null, x is now the root.

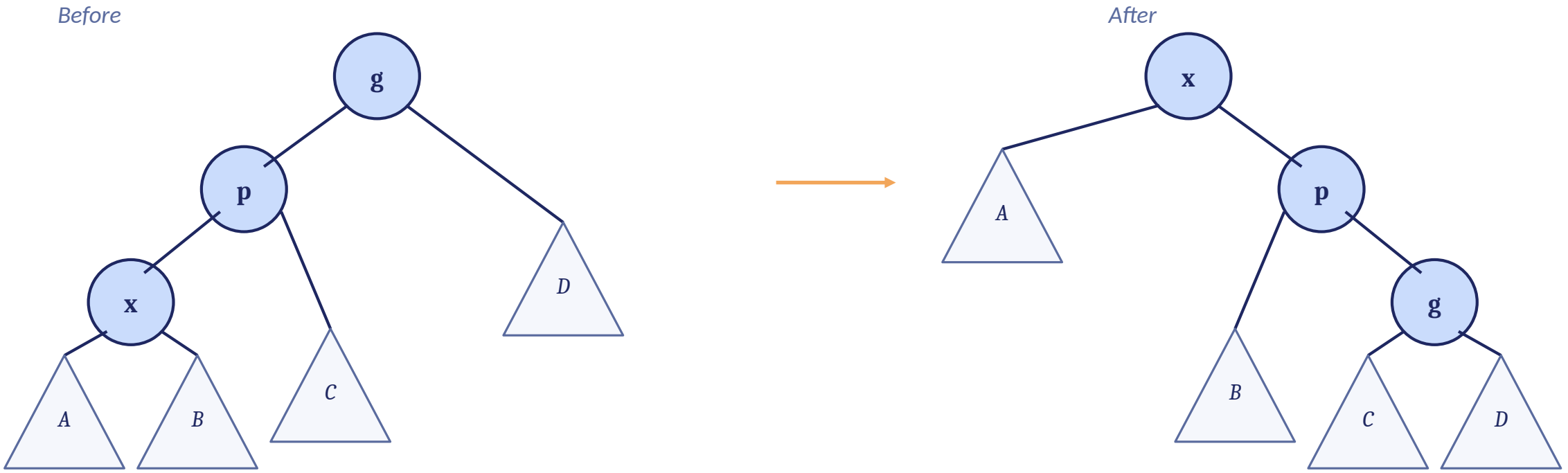
The Zig Step

Used when p is the root. Rotate on the edge between x and p. Zig steps only ever occur as the final step of a splay — exactly when x starts at odd depth.



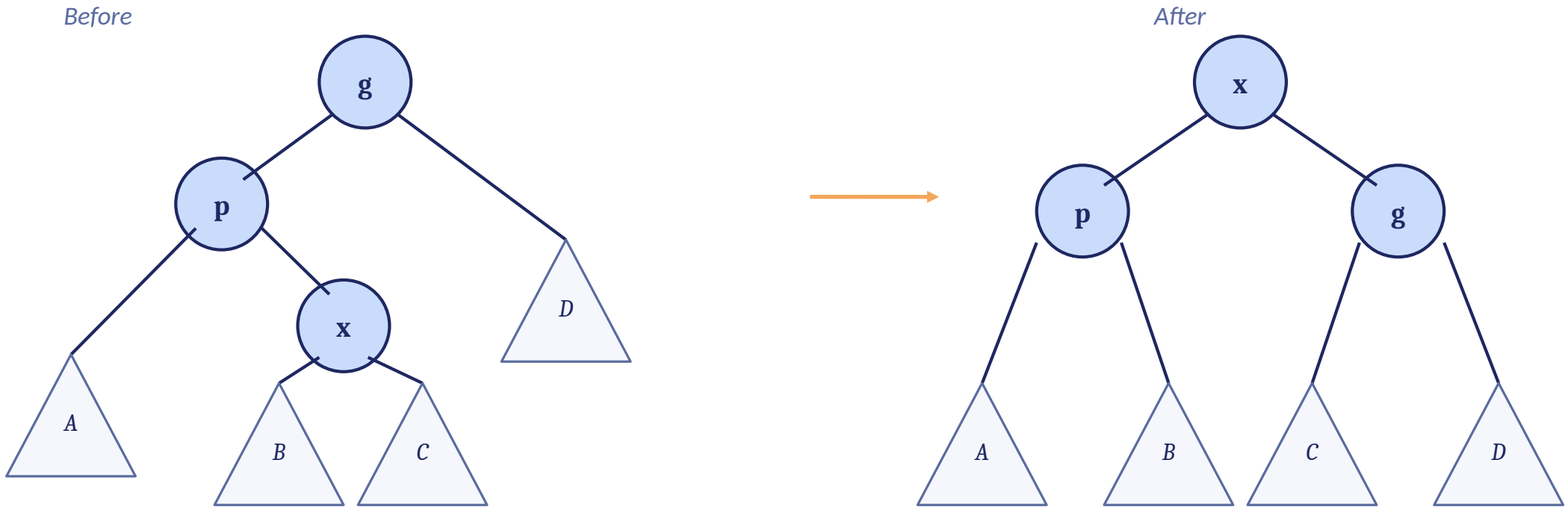
The Zig-Zig Step

Used when p is not the root and x, p are both left children (or both right). Rotate the g–p edge, then the p–x edge. This is the step that distinguishes splay trees from the older “rotate-to-root” method.



The Zig-Zag Step

Used when p is not the root and x, p are opposite-handed children (x is a right child, p a left child — or vice versa). Rotate the p–x edge, then the resulting x–g edge.



Join and Split

Join

Given trees S and T where every element of S is smaller than every element of T:

1. Splay the largest item in S — it becomes S's root, with a null right child.
2. Set that root's right child to T.

Result: a single tree containing all elements of S and T, in $O(\log |S|)$ time.

Split

Given a tree and an element x, produce two trees: elements ≤ x, and elements > x:

1. Splay x — it becomes the root. Its left subtree holds all elements < x; its right subtree holds all elements > x.
2. Detach the right subtree from the rest of the tree.

Insertion and Deletion

Insertion

1. Insert x exactly as in an ordinary binary search tree.
2. Splay x immediately after inserting.
3. The newly inserted x therefore becomes the tree's root.

Deletion

Delete x as in an ordinary BST: if x has two children, swap its value with its in-order predecessor or successor, then remove that (now childless-or-one-child) node instead — reducing to the 0-or-1-child case.

Unlike a plain BST: after deleting, splay the parent of the removed node up to the root.

Persistent Data Structures

A persistent data structure always preserves its previous version when modified. Operations don’t update it in place — they yield a new, updated structure, leaving old ones intact and effectively immutable.

Partial

All versions can be accessed, but only the newest can be modified.

Full

Every version can be both accessed and modified.

Confluent

A meld/merge operation can also combine two prior versions into a new one.

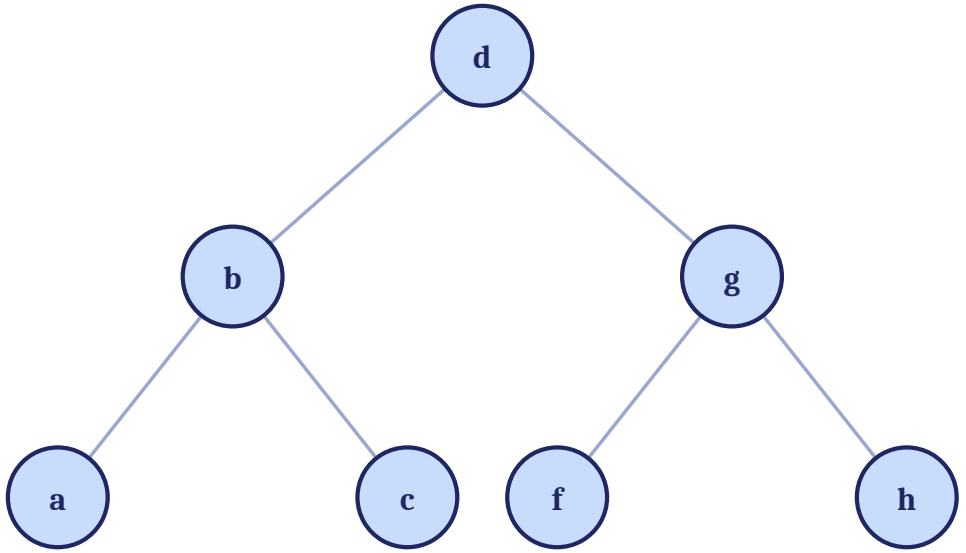
Ephemeral

The term for a structure that is NOT persistent at all.

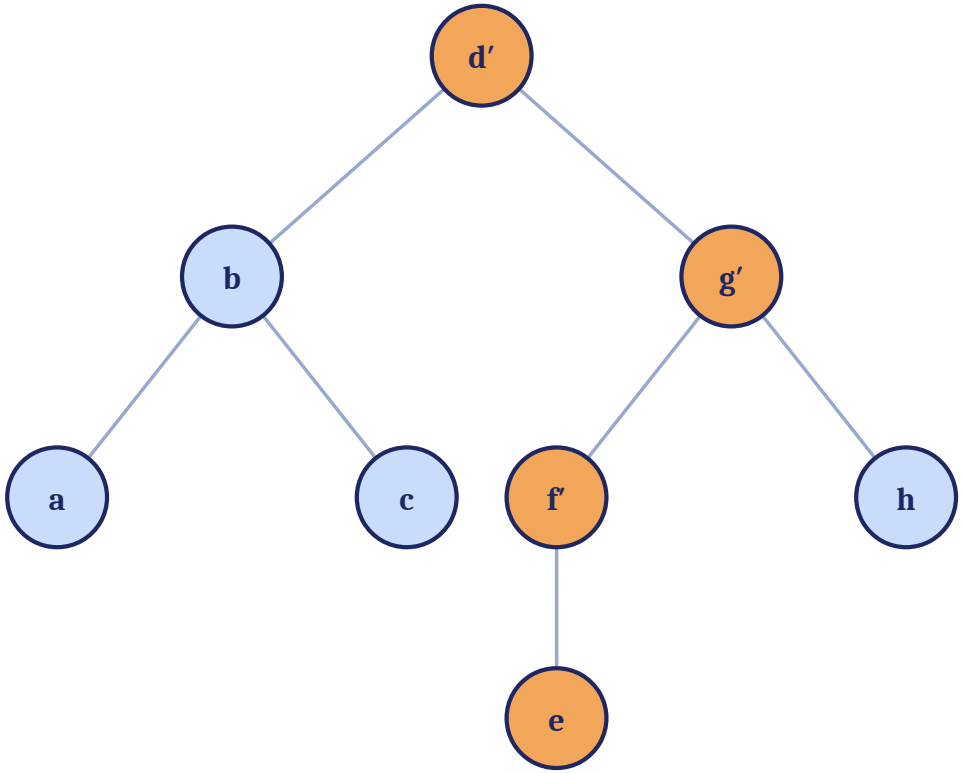
In purely functional programming, all data is immutable, so every structure is automatically fully persistent. Persistence can also be achieved via in-place updates, sometimes using less time/space than a purely functional approach.

Persistent Trees: Path Copying

Consider a BST over xs = [a, b, c, d, f, g, h]. Copying the whole tree on every update is correct but wasteful — instead, share structure between old and new versions wherever possible.



Original tree (xs)



New version (ys) — orange nodes are copies; b's subtree and h are shared

k-d Trees

A k-d tree (k-dimensional tree) is a space-partitioning binary tree where every node is a k-dimensional point. Each non-leaf node implicitly defines a splitting hyperplane, dividing space into two half-spaces — represented by its left and right subtrees.

CHOOSING THE SPLIT DIRECTION

Every node is associated with one of the k dimensions, and its hyperplane is perpendicular to that dimension's axis. E.g. if a node splits on the x-axis, points with smaller x go left, larger x go right — the hyperplane's normal is the unit x-axis.

Applications: multidimensional search keys, range searches, and nearest-neighbor searches. k-d trees are a special case of binary space-partitioning trees.

Construction and Operations

Construction

- Cycle through axes as you descend: e.g. in 3-D, root splits on x, its children on y, grandchildren on z, great-grandchildren back to x, and so on.
- Insert points by repeatedly choosing the median along the current axis — this assumes the full point set is known up front.

Adding elements

Removing elements

Balancing

Nearest-neighbor search

These are the other operations supported on k-d trees, alongside construction and search.

References

[1] **Wikipedia,**
"Splay tree; Persistent data structure; k-d tree" — www.wikipedia.org

[2] **David P. Williamson & David B. Shmoys,**
"The Design of Approximation Algorithms" — e-book (downloadable)

[3] **Mikhail J. Atallah & Marina Blanton,**
"Algorithms and Theory of Computation Handbook, Special Topics and Techniques, 2nd ed." — CRC Press, 2010

[4] **Allen B. Tucker, Jr.,**
"The Computer Science and Engineering Handbook" — CRC Press, 1997

Disclaimer: these notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the faculty.