Advanced Algorithms **Session 2015-16**

Lecture 8: Self adjusting data structures, Jan. 12, 2016

Faculty: K.R. Chowdhary *: Professor of CS*

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.*

8.1 Introduction to self-adjusting trees

A *splay* tree is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as *insertion*, *look-up* and *removal* in $O(\log\ n)$ amortized time. For many sequences of non-random operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown.

All normal operations on a binary search tree are combined with one basic operation, called *splaying*. Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree. One way to do this is to first perform a standard binary tree search for the element in question, and then use tree rotations in a specific fashion to bring the element to the top. Alternatively, a top-down algorithm can combine the search and the tree reorganization into a single phase.

8.1.1 Advantages and disadvantages

The self adjusting data structures provides good performance, as it is self-optimizing, in that frequently accessed nodes will move nearer to the root where they can be accessed more quickly. The worst-case height-though unlikely-is $O(n)$, with the average being $O(\log\ n)$. Having frequently used nodes near the root is an advantage for many practical applications: particularly useful for implementing caches and garbage collection algorithms.

Possibility of creating a persistent data structure version of splay trees-which allows access to both the previous and new versions after an update. This can be useful in functional programming. Working well with nodes containing identical keys-contrary to other types of self-balancing trees. All tree operations preserve the order of the identical nodes within the tree, which is a property similar to stable sorting algorithms. A carefully designed find operation can return the leftmost or rightmost node of a given key.

The disadvantage of splay trees is that the height of a splay tree can be linear. For example, this will be the case after accessing all n elements in ascending order. Since the height of a tree corresponds to the worst-case access time, this means that the actual cost of an operation can be high.

8.1.2 Operations

The operations on the self-adjusting trees are: splaying, join, split, insertion, and deletion.

8.1.2.1 Splaying

When a node x is accessed, a splay operation is performed on x to move it to the root. To perform a splay operation we carry out a sequence of splay steps, each of which moves x closer to the root. By performing a splay operation on the node of interest after every access, the recently accessed nodes are kept near the root and the tree remains roughly balanced, so that we achieve the desired amortized time bounds.

Each particular step depends on three factors:

- Whether x is the left or right child of its parent node, p,
- whether p is the root or not, and if not
- whether p is the left or right child of its parent, g (the grandparent of x).

It is important to remember to set gg (the great-grandparent of x) to now point to x after any splay operation. If gg is null, then x obviously is now the root and must be updated as such.

There are three types of splay steps, each of which has a left- and right-handed case. For the sake of brevity, only one of these two is shown for each type. These three types are:

Zig step: This step is done when p is the root. The tree is rotated on the edge between x and p. Zig steps exist to deal with the parity issue and will be done only as the last step in a splay operation and only when x has odd depth at the beginning of the operation (see fig. **??**).

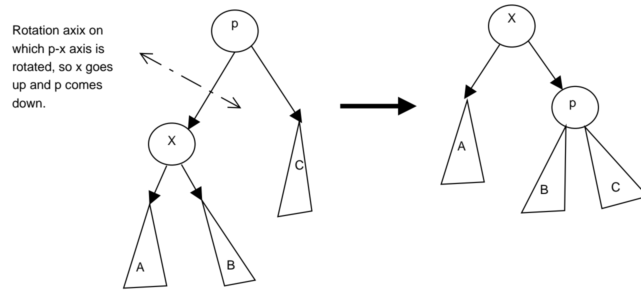

Figure 8.1: Zip Step.

Zig-zig step: This step is done when p is not the root and x and p are either both right children or are both left children. The picture below shows the case where x and p are both left children. The tree is rotated on the edge joining p with its parent g, then rotated on the edge joining x with p. Note that zig-zig steps are the only thing that differentiate splay trees from the rotate to root method introduced by Allen and Munro prior to the introduction of splay trees (see fig. **??**).

Zig-zag step: This step is done when p is not the root and x is a right child and p is a left child or vice versa. The tree is rotated on the edge between p and x, and then rotated on the resulting edge between x and g.

Join. Given two trees S and T such that all elements of S are smaller than the elements of T, the following steps can be used to join them to a single tree:

1. Splay the largest item in S. Now this item is in the root of S and has a null right child.
2. Set the right child of the new root to T.

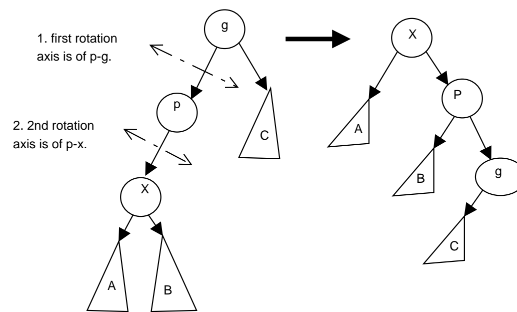

Figure 8.2: Zip zig.

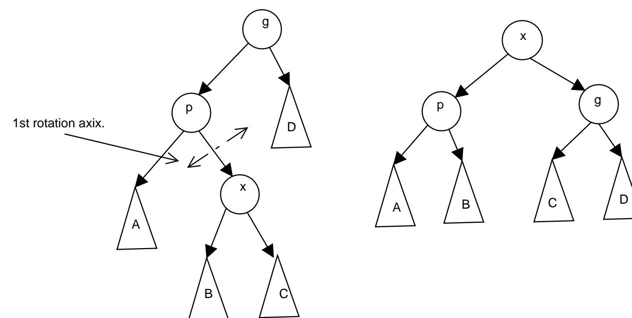

Figure 8.3: Zip zag.

Split. Given a tree and an element x, return two new trees: one containing all elements less than or equal to x and the other containing all elements greater than x. This can be done in the following way:

1. Splay x. Now it is in the root so the tree to its left contains all elements smaller than x and the tree to its right contains all element larger than x.

2. Split the right subtree from the rest of the tree.

Insertion. To insert a value x into a splay tree:

1. Insert x as with a normal binary search tree.

2. when an item is inserted, a splay is performed.

3. As a result, the newly inserted node x becomes the root of the tree.

Deletion. To delete a node x, use the same method as with a binary search tree: if x has two children, swap its value with that of either the rightmost node of its left sub tree (its in-order predecessor) or the leftmost node of its right subtree (its in-order successor). Then remove that node instead. In this way, deletion is reduced to the problem of removing a node with 0 or 1 children. Unlike a binary search tree, in a splay tree after deletion, we splay the parent of the removed node to the top of the tree.

8.2 Persistent data structure

A persistent data structure always preserves its previous version when it is modified. Such data structures are effectively immutable, as their operations do not (visibly) update the structure in-place, but instead always yield a new updated structure.

A data structure is partially persistent if all versions can be accessed but only the newest version can be modified. The data structure is fully persistent if every version can be both accessed and modified. If there is also a meld or merge operation that can create a new version from two previous versions, the data structure is called confluently persistent. Structures that are not persistent are called ephemeral.

These types of data structures are particularly common in logical and functional programming, and in a purely functional program all data is immutable, so all data structures are automatically fully persistent. Persistent data structures can also be created using in-place updating of data and these may, in general, use less time or storage space than their purely functional counterparts.

While persistence can be achieved by simple copying, this is inefficient in CPU and RAM usage, because most operations make only small changes to a data structure. A better method is to exploit the similarity between the new and old versions to share structure between them, such as using the same subtree in a number of tree structures. However, because it rapidly becomes infeasible to determine how many previous versions share which parts of the structure, and because it is often desirable to discard old versions, this necessitates an environment with garbage collection.

8.2.1 Trees

Consider a binary tree used for fast searching, where every node has the recursive invariant that subnodes on the left are less than the node, and subnodes on the right are greater than the node. For instance, the set of data:

$xs = [a, b, c, d, f, g, h]$

might be represented by the following binary search tree:

The next diagram is persistent tree. Notice two points: Firstly the original tree (xs) persists. Secondly many common nodes are shared between the old tree and the new tree. Such persistence and sharing is difficult to manage without some form of garbage collection (GC) to automatically free up nodes which have no live references, and this is why GC is a feature commonly found in functional programming languages.

8.3 Multidimensional Trees

A k-d tree (short for k-dimensional tree) is a space-partitioning data structure for organizing points in a k-dimensional space. k-d trees are a useful data structure for several applications, such as searches involving a multidimensional search key (e.g. range searches and nearest neighbor searches). k-d trees are a special case of binary space partitioning trees.

The k-d tree is a binary tree in which every node is a k-dimensional point. Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as half-spaces. Points to the left of this hyperplane are represented by the left subtree of that node and points right of the hyperplane are represented by the right subtree. The hyperplane direction is chosen in

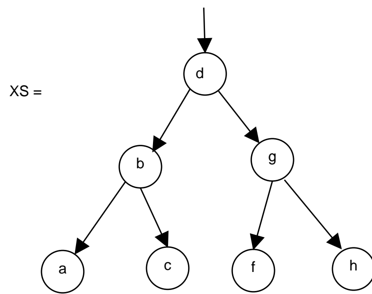

Figure 8.4: Original tree.

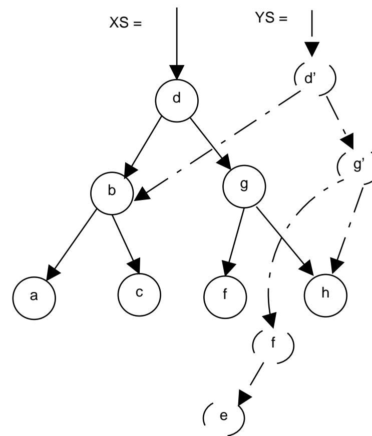

Figure 8.5: Modified tree.

the following way: every node in the tree is associated with one of the k-dimensions, with the hyperplane perpendicular to that dimension's axis. So, for example, if for a particular split the "x" axis is chosen, all points in the subtree with a smaller "x" value than the node will appear in the left subtree and all points with larger "x" value will be in the right subtree. In such a case, the hyperplane would be set by the x-value of the point, and its normal would be the unit x-axis.

8.3.1 Operations on k-d trees

Construction. Since there are many possible ways to choose axis-aligned splitting planes, there are many different ways to construct k-d trees.

- As one moves down the tree, one cycles through the axes used to select the splitting planes. For example, in a 3-dimensional tree, the root would have an x-aligned plane, the root's children would both have y-aligned planes, the root's grandchildren would all have z-aligned planes, the root's great-grandchildren would all have x-aligned planes, the root's great-great-grandchildren would all have y-aligned planes, and so on.)
- Points are inserted by selecting the median of the points being put into the subtree, with respect to their coordinates in the axis being used to create the splitting plane. (Note the assumption that we feed the entire set of n points into the algorithm up-front.)

Adding elements, Removing elements, Balancing, Nearest neighbour search, are the other operations.

References

[1] www.wikipedia.org.

[2] David P. Williamson and David B.Shmoys, "Design of approximation Algorithms", ebook (downloadable)

[3] Mikhail J. Atallah and Marina Blanton, “Algorithms and Theory of Computation handbook, Second Edition, Special Topics and Techniques”, CRC Press, Taylor and Francis Group - A Chapman and Hall Book, 2010.

[4] Allen B. Tucker, Jr., “The computer Science and Engineering Handbook,” *CRC Press*, 1997.