Splay trees & link/cut trees

Ivan Smirnov
ifsmirnov@yandex.ru

September 27, 2018

1 Rationale

Link/cut tree Range query problems are usually stated like "given an array, compute something (minimum, sum, etc) for its subarrays (l_i, r_i)". Various data structures exist for this kind of problems, namely segment tree, treap and others, usually working in $O(\log n)$ time per query.

There is a natural extension of range queries to queries over paths in a tree. Here we generally use the heavy-light decomposition. It gives $O(\log^2 n)$ time per query, if each path is represented as a treap or a segment tree.

However, tree must be static in order to use the HLD. What if we want to add and remove edges to our tree, or even to reroot it? Link/cut tree develops the idea of the HLD, dynamically changing paths when the tree structure is modified. Same as the HLD, $O(\log^2 n)$ (amortized) bounds hold if we use treaps to store paths.

Splay tree Splay tree is a binary search tree which allows splitting and merging, like a treap, and gives $(O \log n)$ bounds per operation (though amortized). However, the main rationale for studying splay trees is that link/cut trees based on splay trees achieve $O(\log n)$ amortized complexity for operation. Furthermore, implementing link/cut is much simpler with splay trees.

2 Splay tree

Unlike other balanced binary search trees, splay trees do not store any additional information for balancing. Instead all balancing happen as a side effect of other operations. We define an operation $splay(v)$. It changes the structure of the tree and makes v the root, keeping the BST property. All others operations naturally derive from it.

1. **Splay** If x is root, do nothing. Else denote $p = parent(x)$.

   If x is root, perform $zig(v)$.

   If both x and p are right or left children, perform $zigzig(x)$.

   If both x is a left child and p a right child (or vice versa), perform $zigzag(x)$.

   Repeat until x is root.

   You can find *zig*, *zigzig* and *zigzag* pictures in this handout.

2. **Find** Find a corresponding vertex as in a BST, then splay it.

   NB: although find is a non-mutating operation, you must splay the found node. Otherwise complexity bounds could be violated.

3. **Insert** Insert new vertex into a BST, then splay it.

4. **Split** Let v be the greatest expected vertex in the left part. Splay it. Now v and its left subtree is the left part of split tree, v's right subtree is the right part.

5. **Merge** Let v be the greatest vertex in the left part, u be the smallest vertex in the right part. Splay both of them and set u as v's right child.

6. **Delete** Splay v, then merge its two subtrees.

Proposition. *Any sequence of n operations with a splay tree works in $O(n \log n)$ time.*

Proof sketch. We denote the size of the subtree of vertex v, including v, with $size(v)$. Also we define $rank(v) = \log_2 size(v)$ and introduce the potential $\Phi = \sum_v rank(v)$. Now for each balancing operation (zig, zigzig and zigzag) we want to bound $\Delta\Phi$ by the difference of ranks of v before and after the operation.

Afterwards it suffices to show that $\max \Phi \leq n \log n$ and use standard amortized analysis, saying that the potential cannot increase too much.

□

2.1 Implicit splay tree

Like implicit treap, splay tree can be modified to store arrays. Instead of $split(key)$, we introduce $split(index)$. After this operation the tree is split into two trees, fisrt of size $index$ and second containing all other vertices. $Find(key)$ should be replaced with $find(index)$ as well. These operations are possible if we additionally store the subtree size in each vertex.

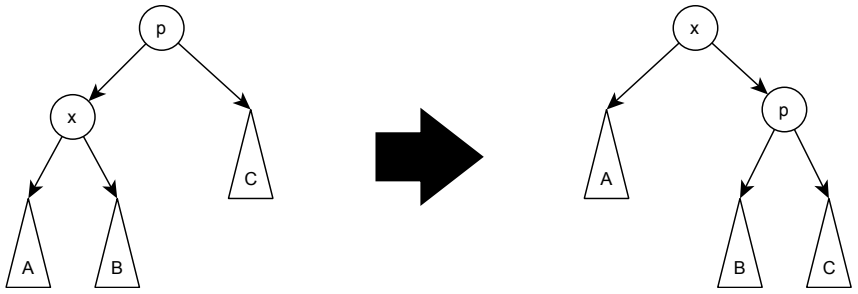

Figure 1: Zig

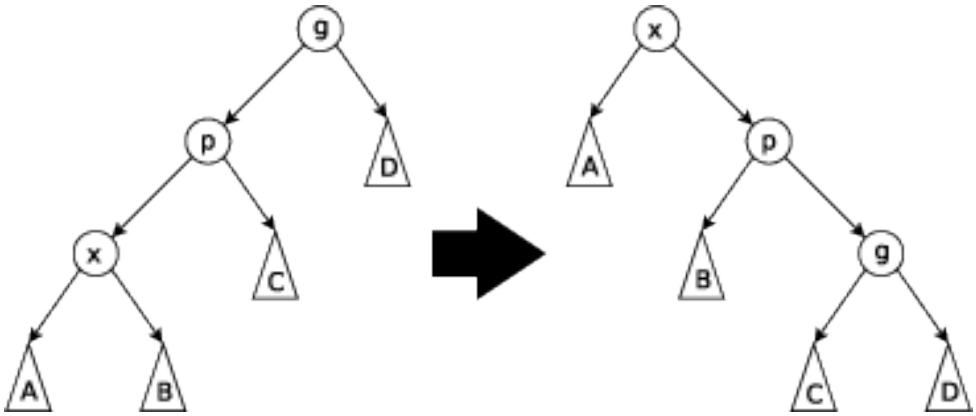

Figure 2: Zigzig

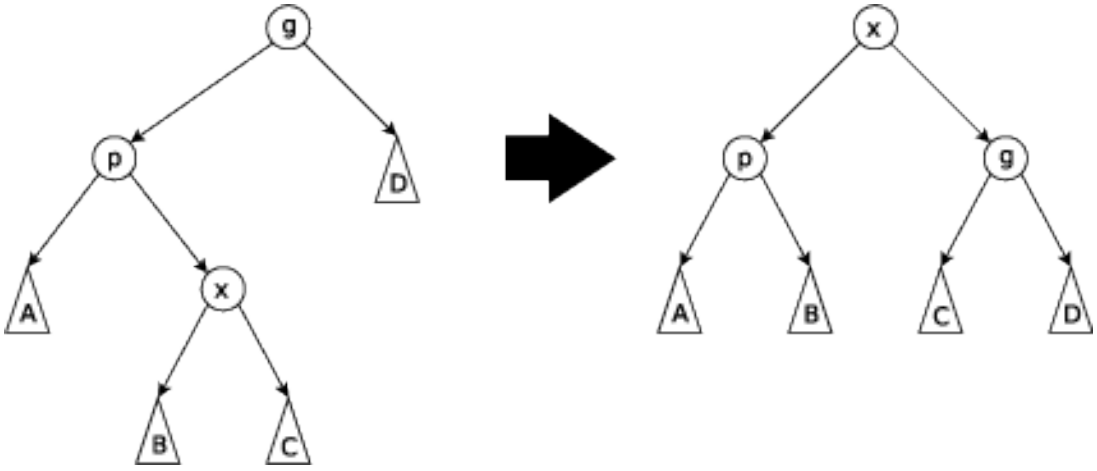

Figure 3: Zigzag

Images source: https://en.wikipedia.org/wiki/Splay_tree

3 Link/cut tree

We maintain a rooted tree. Vertices are grouped into vertical paths, each vertex belonging to exactly one path. Vertices of each path are stored in an (implicit) splay tree. The topmost vertex of the path has index zero, other vertices are indexed by depth.

These paths are called *preferred* paths. Edges in paths are called *preferred* edges.

Every path that does not contain the root should be linked to its parent. This link goes from the root of a corresponding splay tree to some vertex of the splay tree of the parent's path. These links are called *path-parent links*.

As in splay trees, there is a fundamental operation, namely expose. All others derive from it.

1. **Expose** If something should be done with some node v, it should first be *exposed*. After such operation there will be a preferred path between the root of the tree, denoted by r, and v. This path must end at v and not continue any deeper.

   Look at the path containing v. First, split this path into two such that v is the last vertex of the left part. In terms of splay tree, we splay v and cut off its right subtree. Note that path-parent link of the right subtree must be set to v.

   Then, until v is root, we follow path-parent links and do the same splitting operation with all paths on the way, linking all new paths together.

2. **FindRoot** To find the root of v, first expose v, then find the leftmost node in its splay tree. Do not forget to splay it afterwards!

3. **Cut** To cut the subtree rooted at v, first expose v's parent. After that v's path-parent link shows to v's parent. Remove this link. Now v's subtree is separated.

4. **Link** Let v be some vertex of a tree and u be the root of another tree. u does not have a left child in its splay tree. because that tree contains only one vertex. Now set v as a left child of u. Now u continues the preferred path of v.

Proposition. *Any sequence of n operations with a link/cut tree works in $O(n \log n)$.*

Proof sketch. We will show only $O(n \log^2 n)$ upper bound. Here we treat the splay tree as a blackbox giving us $O(\log n)$ per operation, so it suffices to show that there are only $O(\log n)$ exposes. We will count the number of new preferred edges formed.

The edge (u, v) is *heavy* if $size(v) \geq size(u)/2$ and *light* afterwards. Note that there are no more than $\log n$ light edges on any vertical path, so we can concentrate only on heavy ones.

If a heavy edge becomes preferred, it was unpreferred before. There are two cases. First, it might be unpreferred from the very beginning. There are at most $n - 1$ such edges.

Second, it might become unpreferred during some previous expose. That mean that during that expose some light edge became preferred. But at most $\log n$ light edges can become preferred during a single operations. So, at most $m \log n + (n - 1)$ heavy edges can become preferred during n operations. □

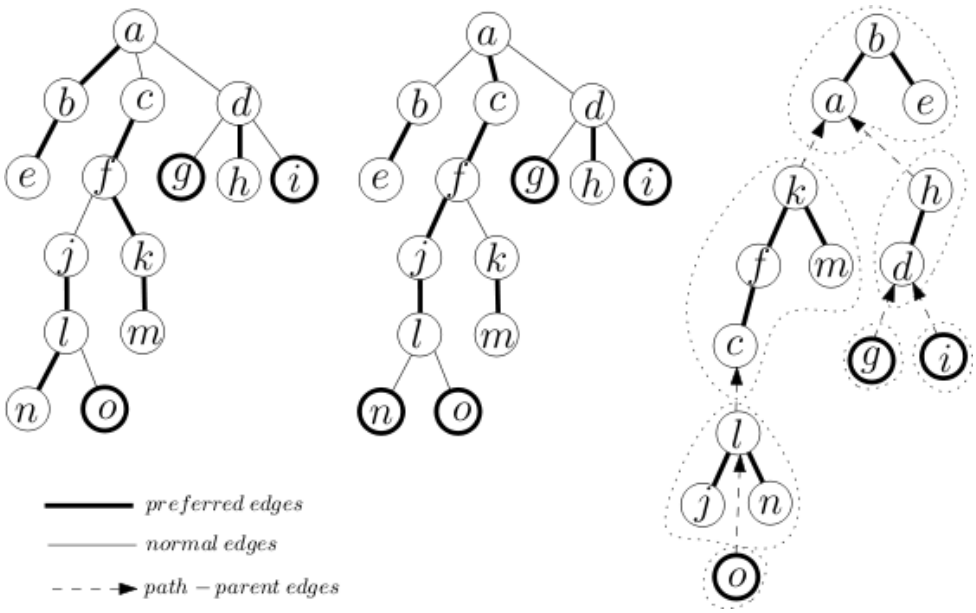

Image source: https://en.wikipedia.org/wiki/Link/cut_tree

4 Discussion

- How to store values in vertices and compute aggregates (say, minimum on a path)?
- And what about edges?
- How to store the subtree weight in a link/cut tree? Is it possible to generalize it to arbitrary subtree query?
- How to do rerooting?