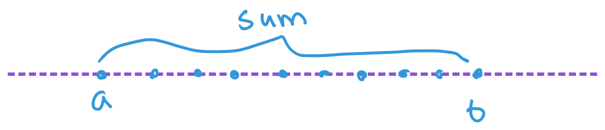
Range Queries and Dynamic Trees

Kent Quanrud

March 23, 2021

1 Range queries



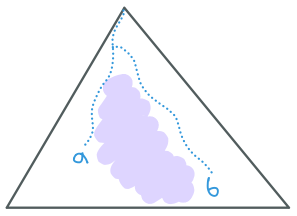
Suppose we have a family of n comparable keys $k_1 < \cdots < k_n$ where each key is associated with some data (e.g., a numerical value). A **range query** is specified by an interval[1] and some method of aggregation, and the output is the appropriately aggregation of values over all keys that lie in the interval. For example, if there are numeric values associated with each key, we might ask for

The sum of all values associated with all keys in $[a, b]$.

Here a key k is in $[a, b]$ if $a \leq k \leq b$. Similarly for (a, b), $(a, b]$, and $[a, b)$. Alternatively we can ask for the *max* of all values (with keys) in $[a, b]$. A slightly different kind of query occurs in a situation where each key may be associated with a list of values of some kind. We might ask for

The combined lists of all values associated with all keys in $[a, b]$*, in sorted order w/r/t their keys.*

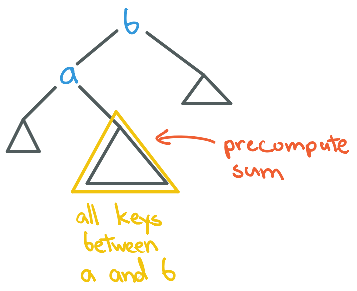
Not only do we want to serve these queries efficiently, but we want to keep serving them efficiently as the keys and values are updated dynamically.



One solution to the problem is with binary search trees. We insert all the keys in a binary search tree tree where each node also auxiliary pointers to the data associated with the key. Now, suppose we want to be able to compute the sum over an interval $[a, b]$. For a static tree, we would search for a and b and then sum over all the nodes "inside" the search paths. To avoid visiting all of these nodes, we can precompute the sum over all subtrees. Then we only have to visit the root of each subtree inside the search paths. Still the total time will be proportional to the path lengths, which may be very long.

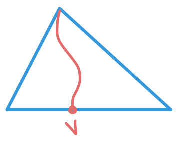
[1]open, closed, or half-open. We remind the reader that $[a, b] = \{x : a \leq x \leq b\}$, $(a, b) = \{x : a < x < b\}$, $(a, b] = \{x : a < x \leq b\}$, and $[a, b) = \{x : a \leq x < b\}$.

Instead we can use balanced binary search trees. Here we will focus on splay trees, continuing our discussion from last time. Again we maintain the sum of all subtrees, and point out that it is easy to update these sums with each local rotation. Now, suppose we want to query the sum over $[a, b]$. We assume a and b are both keys; otherwise replace a with the first key $> a$ and b with the first key $< b$. If we splay a and then b, then b will be the root, a will be the left child of b, and all keys between a and b will be in a's right subtree. Thus we sum the values of a, of b, and the precomputed sum of the a's right subtree.



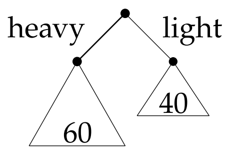
Clearly this approach extends more generally than sums – as long as the aggregation can be computed pairwise in any order (i.e., it is defined by an *associative* binary operator) in constant time, then we can maintain the aggregates at each subtree, and follow the above approach. This is more flexible then might appear. Take for example the second type of query where we want to list of all the values in $[a, b]$. We maintain, at each node, the total number of values in the subtree rooted at that node. We can use these quantities to search and splay for the key k that contains the ith value in $[a, b]$, for any fixed i, in $O(\log n)$ time per key. (There are slightly more involved approaches that can reduce the time to $O(\log n)$ plus $O(1)$ per key.)

2 Range queries on paths in a tree

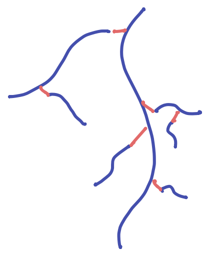


One way to generalize range queries on a line is in trees. The model is as follows. Let T be a rooted tree, whose nodes are associated with some values. We now consider aggregations over all the nodes on the path from a given vertex to a root. For example, given a node v, we may want to compute the sum of all values on the path from v to the root.

Let v be a node in a rooted tree, and w a child of v. We call w a **heavy child** of v if it contains at least half the nodes in the tree rooted at v. Any node has at most one heavy child. We call the edge from a parent to a heavy child a **heavy edge**. The paths of heavy edges partitions the nodes of a tree into paths, called the heavy path decomposition.



Edges that are not heavy are called **light**. For a vertex v, the **light depth** of a vertex v is defined as the number of light edges on the v to root path. Observe that

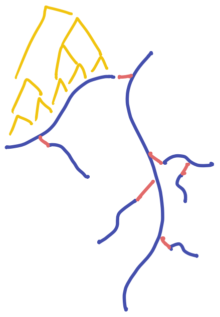


$$(\text{light depth } v) \leq \log_2(n)$$

because each light edge divides the number of remaining nodes in half.

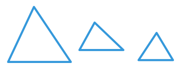
Theorem 1. *One can support path queries in a rooted tree in $O\left(\log^2(n)\right)$ amortized time per operation.*

Proof. We compute a heavy path decomposition over the tree. For each heavy path, we build a splay tree on that path keyed by depth, that supports range queries along that path. Now, when querying a node v, the root to v path breaks up into segments of heavy paths split up by light edges. There are at most $O(\log n)$ light edges, so each there are $O(\log n)$ heavy segments. We sum over each segment by a range query to the corresponding splay tree. Then we sum all the values. There are $O(\log n)$ queries to splay trees and each query takes $O(\log n)$ amortized time. ■

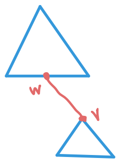


3 Dynamic trees

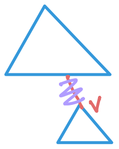
We extend the model from the previous section further by allowing the rooted tree T to change. More generally, we will design a data structure that maintains a forest of rooted trees. The rooted trees are updated by the following operations.



Link. Given two vertices w and v, where v is the root of a tree, `link(`w`,`v`)` make v a child of w.



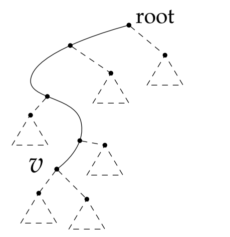
Cut. Given a vertex v that is not a root, `cut(`v`)` removes v's parent edge, making v the root of its own tree.



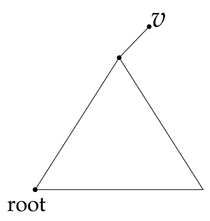
In the static setting, the heavy path decomposition reduces path aggregates to $O(\log n)$ range queries. But here the heavy paths can frequently change as edges are added and deleted, which would require us to start to rebuild many splay trees.

3.1 Access.

The `link-cut-tree` data structure maintains a *dynamic* path decomposition of each tree that rapidly responds to each operation. These changes are made through an operation called `access`. `Access` takes as input a node v, and updates the path decomposition so that the v's path is one of the paths in the path decomposition. `Access` also update the underlying splay trees on the fly to ensure that each path in the decomposition is still represented by a splay tree. In particular, after `access(`v`)`, there is a splay tree over the root-to-v, of which v is the root. As the last vertex in the path, v has no left child in the auxiliary tree.



Represented tree



v's auxiliary tree

The pictures above describe the arrangement after `access(`v`)`. The represented tree is on the left, and the root to v path is emphasized by solid lines. The dashed edges hanging off the root to v path are not in any path of the path decomposition. To be explicit, let us state the guarantees of `access(`v`)` that we invoke.

1. The root-to-v path is a path in the path decomposition.
2. v is the root of the splay tree representing its path.

We will eventually show that `access(`v`)` takes $O\left(\log^2(n)\right)$ amortized time.[2] Given `access`, we can implement `link`, `cut`, and path queries as follows.

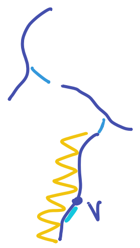
Cut. Recall that `cut` removes the parent edge of a node v. To implement this, we first call `access(`v`)`, which makes the root path to v part of the path decomposition with v the root of the auxiliary splay tree. In the splay tree, v has no right child, and the left child of v contains the rest of the path. We remove the edge from v to its left subtree.

Link. Recall that `link(`v`,`w`)` takes a node v, which is the root of some tree, and makes it a child of a node w, which is the child of another tree. To implement `link(`v`,`w`)`, we first access w. This gives us a splay tree over the root path to w. We also access v. Since v was already the root, v will be only vertex in its auxiliary tree. We set w to be v's parent, and we make w's auxiliary tree the left child of the (singleton) splay tree at v.

Find-root. Recall that `find-root(`v`)` returns the root of the tree containing v. To find the root, we access v, which gives us a splay tree containing the root-to-v path. The root has the smallest key in this tree. We `splay` the root to be the root of the auxiliary tree, and return it.

Aggregating over a path. Suppose the nodes were associated with numerical values, and we want to sum up the values along the root to v path for a node query. We extend the data structure so that all the auxiliary splay trees act as a range tree over its nodes. Assuming this is in place, we implement `sum-path(`v`)` by calling `access(`v`)`, and then return the sum in v's auxiliary tree. Of course, summation can be replaced with other aggregate operations as discussed in Section 1.

3.2 Analyzing `access`



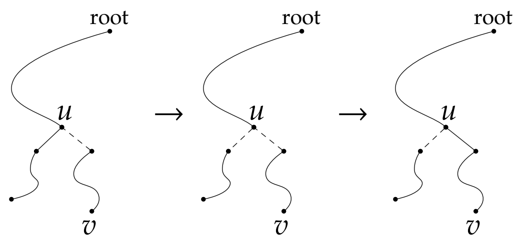
It remains to define and analyze `access`. Before defining `access`, we first introduce a helpful auxiliary function called `clip`. `clip` takes as input a vertex v. That vertex v is part of some auxiliary path in the path decomposition. `clip` adjusts the path decomposition so that v is the last vertex in its auxiliary path. Psuedocode for `clip` is as follows.

[2]One can obtain an improved bound of $O(\log n)$, but the analysis is a little too complicated to fit in this lecture. We refer the reader to [2] or [1] for that analysis.

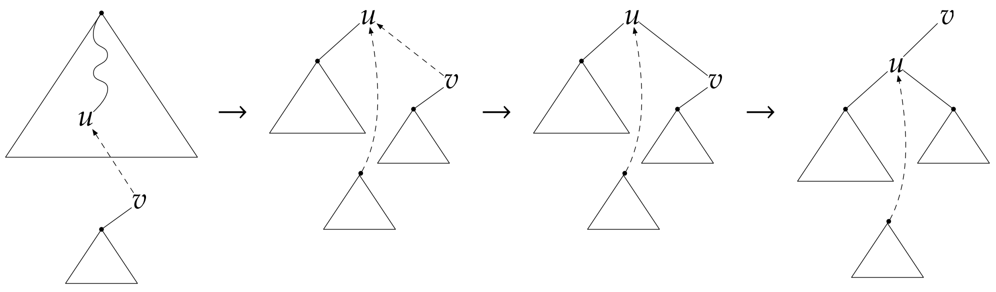
```
clip(v)
// Makes v the last vertex in its auxiliary path in the path decomposition. Makes v at the root
   of its auxiliary splay tree.
1. Splay v within its auxiliary tree.
// The right subtree of v in the auxliary tree represents the part of the path below v.
2. In the auxiliary tree, split off the right subtree of v as its own path,
   with v as the parent of the path.
```

`clip` consists of one splay operation and a constant number of pointer changes, hence takes $O(\log n)$ amortized time.

Now we describe `access`. `Access` takes as input a vertex v, and the goal is to adjust the path decomposition so that the root-to-v path is one of the paths. The first step is to `clip` v. Then, until v's auxiliary path is the entire root-to-v path, we extend v's auxiliary path as follows. We first look at the parent u of the auxiliary path. We `clip` u, make u the bottom of it's auxiliary path. We then attach v's auxiliary path to u's auxiliary path, making one long auxiliary path. These steps, in terms of their effect on the path decomposition, are illustrated below.



Above, the solid edges represent paths in the path decomposition, and dashed edges represent edges excluded from the path decomposition. In terms of the underlying splay trees, `clip`(u) makes u the root of its splay tree with no right subtree. We already have v at the root of its splay tree. We then make v the right child of u. Then we splay v to the top. The steps on the splay trees are diagrammed below.



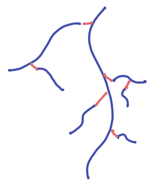
Above, the solid triangles represent splay trees. The dashed edges represent parent pointers in the represented tree. Pseudocode for `access` is as follows.

```
access($v$)

1. clip $v$
2. until the path for $v$ is the entire $v$ to root path
   A. let $u$ be the parent vertex of $v$'s auxiliary path.
   B. clip $u$
   C. attach the auxiliary tree for $v$ as the right subtree of $u$ in the
      auxiliary tree for $u$
   D. splay $v$
```

Lemma 3.1. *Suppose a call to* `access(`v`)` *requires C changes in the path decomposition. Then* `access(`v`)` *takes $O\big((C+1)\log(n)\big)$ amortized time.*

Proof. We first pay $\log(n)$ amortized time to call `clip(`v`)`. For each iteration of the loop, we pay $O(\log n)$ amortized time for a constant number of `splay` and `clip` operations for each iteration of the loop. Each iteration corresponds to a change in the path decomposition. ■



It remains to bound C, the total number of changes to the path decomposition. We do this by comparison to the heavy path decomposition. Recall that a vertex is heavy if its subtree represents at least half of the subtree of its parents (Section 2). Vertices that aren't heavy are called light. An edge is called heavy if the child is heavy and an edge is called light if the child is light.

We classify each edge in the represented tree as follows. First, an edge is either in one of the paths in the path decomposition, or not. Second, each edge is either heavy, or not. The combinations creates four disjoint categories of edges.

Let us say that an edge is **active** when it is becomes part of a path in the path decomposition. We say that an edge is **inactive** when it is not in the path decomposition. The cross product of

$$\begin{Bmatrix}\text{heavy}\\ \text{light}\end{Bmatrix} \times \begin{Bmatrix}\text{active}\\ \text{inactive}\end{Bmatrix}$$

gives four classes of edges at any point. Every time we change the path decomposition, we are either

1. Creating an active light edge.
2. Destroying a light active edge.
3. Destroying a heavy active edges.
4. Creating a heavy active edge.

Every time we change the path decomposition, we are demoting one edge and promoting another edge, and these two edges are siblings. In particular, at most one of them is heavy.

More precisely, let us define

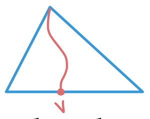
$$\begin{aligned} C &= \text{\# changes to the path decomposition,} \\ L &= \text{\# active light edges created,} \\ L' &= \text{\# light active edges destroyed,} \\ H &= \text{\# active heavy edges destroyed,} \\ H' &= \text{\# heavy active edges created.} \end{aligned}$$

Then

$$C \leq L + H + L' + H' \overset{(a)}{\leq} 2L + H + H' \overset{(b)}{<} 2(L + H) + n.$$

Here (a) is because every light edge we destroyed was previously created. (b) is because the number of heavy active edges we create, but do not later destroy, is less than n as there are only $n - 1$ active edges total. It remains to upper bound L and H for `access`, `link`, and `cut`.

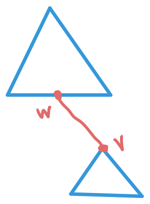
1. `Access(`v`)`. We have



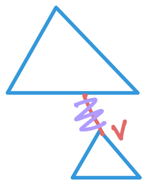
$$L + H \overset{(c)}{\leq} 2L + 1 \overset{(d)}{\leq} O\left(\log n\right),$$

for the following reasons. (c) is because every time a heavy edges is demoted, unless it was edge with parent v, a light edge is promoted instead. (d) is because there are at most $O\left(\log n\right)$ light edges along the root to v path.

2. `Link(`v`,`w`)`. When we link v to w, besides the work charged to calls to `access`, some light edges along w's path may become heavy, and some heavy edges hanging off of w's path may become light. However, the edges that become heavy are already promoted, and the edges that became light are already demoted, by the preceding called to `access`. Thus, excluding the parts charged to `access`, we have $L + H = 0$.



3. `Cut(`v`)`. When we cut v from its parent, some of the nodes on the root-to-v's path may become light. But there are at most $O\left(\log n\right)$ light edges from the root to v path. The cut may also destroy a heavy edge if v's parent heaby was heavy. Thus, excluding changes due to `access`, we have $O\left(\log n\right)$ with respect to the potential.



Thus we have given an amortized analysis that shows that each `access`, `link`, and `cut` introduces $O\left(\log n\right)$ amortized changes to the path decomposition. Meanwhile the amortized running time, by Lemma 3.1, is an additional $O\left(\log n\right)$ factor on top of that, in addition to a $O\left(n \log n\right)$ overhead from the n extra changes. Thus we have shown the following.

Lemma 3.2. *With $O\left(n \log n\right)$ additional overhead,* `access`, `link`, *and* `cut` *each take $O\left(\log^2(n)\right)$ amortized time.*

4 Roots, re-rooting, and an application to dynamic disjoint union

Recall the disjoint union problem which was important for MST. We have a set of n elements each initially in their own singleton sets. For any two elements x and y in different sets[3], we can call `union` to combine their sets. Meanwhile for any two elements we can query if they are in the same set.

This version of the disjoint union data structure is said to be *partially dynamic* – they allow us to take `union`'s of two sets, but we cannot "delete" a previous `union`. We want to extend this to a *fully dynamic* disjoint union data structure in the following sense. Suppose we call `union(`x`,`y`)` on two elements x and y. Later on, we want to be able to call "`split(`x`,`y`)`" to undo this union, while retaining all other `union`'s that might have occurred since.

An equivalent formulation is as follows. Let $F = (V, E)$ be a forest. Suppose edges are inserted into F (without introducing cycles) and deleted from F. The goal is to be able to query, for any two vertices u and v, whether u and v are connected in F. Edge insertions correspond to `union`'s, and edge deletions correspond to `split`'s.

This dynamic variant of disjoint-union can be resolved by link-cut trees with two new operations.

1. `root(`v`)` returns the root of the tree containing v.
2. `re-root(`v`)` reroot's v's tree to have root v.

`Union(`u`,`v`)` corresponds `reroot`-ing v and then `link`-ing the nodes. `Split(`u`,`v`)` corresponds to a `cut`. To query if two elements are in the same set, we call `root` on both and see if they are in the same set.

Implementing `root(`v`)` is immediate. We `access(`v`)`, and then splay the leftmost node in the auxiliary tree to find the root. To implement `re-root`, we first observe that this corresponds to reversing all the edges on the v-to-root path. To this end, we maintain, for each node, a bit flag that indicates whether or not the subtree is given in order or in reverse order. Then to reverse all the edges along a path in some splay tree, we flip this bit. Thus to `re-root(`v`)`, we `access(`v`)`, and flip the bit in the auxiliary tree.

As currently planned, the next lecture will discuss how to remove the restriction that F is a forest. That is, we want to able to query connectivity in a dynamically updated graph.

References

[1] Erik Demaine. *Dynamic Graphs 1*. Lecture 19 for MIT 6.851. 2012. URL: `https://courses.csail.mit.edu/6.851/spring12/lectures/L19.html`.

[2] Daniel Dominic Sleator and Robert Endre Tarjan. "Self-Adjusting Binary Search Trees". In: *J. ACM* 32.3 (1985), pp. 652–686. Preliminary version in STOC, 1983.

[3] Here we restrict ourselves to unions over elements in different sets. This distinction was not necessary before.