Fully Dynamic kd-Tree

[Notes for the Training Camp]

Yufei Tao

ITEE
University of Queensland

Last time we learned a new data structure called the kd-tree, which can be constructed efficiently, but does not seem easy to update (why?). Today, we will explain how to extend it to support updates.

Interestingly, we will achieve the purpose using a technique that works for all the structures that are difficult to update, but can be constructed fast. The structure is called logarithmic rebuilding, and turns the kd-tree into a semi-dynamic structure that supports insertions.

The kd-tree, in fact, easily supports deletions. Combining this with logarithmic rebuilding gives our final fully dynamic kd-tree that supports both insertions and deletions.

Recall:

Range Reporting

Let $\mathbb{R}$ denote the set of real values. Let P be a set of n points in $\mathbb{R}^2$.

Given an axis-parallel rectangle q (namely q has the form $[x_1, x_2] \times [y_1, y_2]$), a range reporting query returns all the points of P that are covered by q.

Our objective is to store P in a data structure, so that we can answer all queries efficiently.

The kd-tree achieves the purpose with $O(n)$ space and $O(\sqrt{n} + k)$ query time (where k is the number of points reported), and can be constructed in $O(n \log n)$ time.

Next, we will show how the kd-tree can support a deletion in $O(\log n)$ time amortized, assuming no insertions.

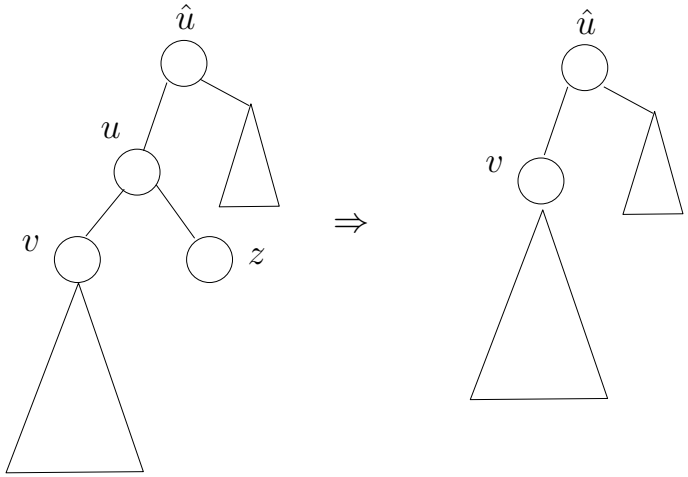
Deletion

Let N be the number of points when the kd-tree was first constructed. We will use n to denote the number of remaining points in the kd-tree.

Suppose that we are deleting a point p. First, we find the leaf z where p is stored. This can be done in $O(\log n)$ time by descending a single root-to-leaf path (why?).

Deletion

Then, we simply remove z from the tree. Let u be the (original) parent of z. If now u has only a single child v, remove u by asking v to take its place (after which v replaces u as a child of the parent of u).



The whole process finishes in $O(\log n)$ time.

> Note that the tree remains full, namely, every internal node has 2 child nodes.

Query Time after Deletions

Still $O(\sqrt{N}+k)$! Why?

- Think about our previous proof for the term $O(\sqrt{N})$. All the nodes that remain in the tree are exactly nodes in the original kd-tree before all the deletions! So, a vertical/horizontal line cannot intersect with (the rectangles of) more nodes than before!
- Think about our previous proof for the term $O(k)$. All we need is that the tree should be full.

How to ensure that the query time is always $O(\sqrt{n}+k)$?

Global Rebuilding

Let us rebuild the whole tree on the remaining points when n has dropped to $N/2$. The cost of this is $O(n \log n)$, or $O(\log n)$ amortized (why?)!

Space consumption: $O(N) = O(n)$.

Query time: $O(\sqrt{N} + k) = O(\sqrt{n} + k)$.

Now we have got:

> We can maintain a kd-tree on n points that consumes $O(n)$ space, answers a query in $O(\sqrt{n} + k)$ time, and supports a deletion in $O(\log n)$ amortized time.

Now let us attend to insertions by introducing the logarithmic rebuilding technique.

Logarithmic Rebuilding

Let us first consider that we have only insertions, but not deletions.

At any moment, we will maintain at most $h = O(\log n)$ kd-trees T_0, T_1, ..., T_{h-1}, such that the i-th ($i \in [1, h]$) tree stores precisely 2^i points. Each point is stored in only one kd-tree.

Logarithmic Rebuilding

To insert a new point p, we

- Identify the smallest $i \geq 0$ such that T_i is empty, and
- Destroy all of $T_0, T_1, ..., T_{i-1}$. Collect all the points there into a set S.
- Construct T_i on $S \cup \{p\}$.

Note that $|T_i| = 2^i$.

Analysis

Construction of T_i takes $O(2^i \log 2^i)$ time. Charge the cost on the 2^i points in T_i, each of which is amortized $O(\log 2^i) = O(\log n)$ time.

Each point can be charged only $O(\log n)$ times (it moves only to a bigger tree).

Amortized insertion time per point: $O(\log^2 n)$.

Querying the Structure

Simply search all the h trees $T_0, T_1, ..., T_{h-1}$.

Query time:

$$O(\sqrt{2^{h-1}} + \sqrt{2^{h-2}} + ... + \sqrt{2^0} + k) \quad = \quad O(\sqrt{n} + k).$$

Next, we will combine all the above discussion to make a fully dynamic structure that supports both insertions and deletions.

Structure

If $n \leq 4$, simply rebuild the whole tree every time there is an insertion or deletion.

Consider $n > 4$. We periodically rebuild the whole tree. Let N be the number of points in the previous rebuilding. We will rebuild the whole tree after there has been $N/2$ updates (regardless of they are insertions or deletions).

At any moment, we will maintain at most $h = O(\log N)$ kd-trees T_0, T_1, ..., T_{h-1}, such that the i-th ($i \in [1, h]$) tree stores at most 2^i points. Each point is stored in only one kd-tree.

Deletion

To delete a point p, first find the kd-tree T_i containing the point. Then, delete it from T_i.

Insertion

To insert a new point p, we

- Identify the smallest $i \geq 0$ such that

$$\sum_{i=0}^{i} |T_i| \leq 2^i.$$

- Destroy all of $T_0, T_1, ..., T_i$. Collect all the points there into a set S.
- Construct T_i on $S \cup \{p\}$.

Our structure:

- Consumes $O(n)$ space.
- Answers a query in $O(\sqrt{n} + k)$ time.
- Handles an insertion in $O(\log^2 n)$ amortized time.
- Handles a deletion in $O(\log n)$ amortized time.

How to prove it?