Persistent Data Structures

Robin Visser

IOI Training Camp
University of Cape Town

4 March 2017

Overview

Definition

What exactly are *persistent data structures*?

Definition

What exactly are *persistent data structures*?

Two words: **Time Travel**

Definition

What exactly are *persistent data structures*?

Two words: **Time Travel**

- It's data structures which preserve previous versions of itself. Essentially: data structures with archaeology.

Definition

What exactly are *persistent data structures*?

Two words: **Time Travel**

- It's data structures which preserve previous versions of itself. Essentially: data structures with archaeology.
- A general concept which can be applied to any data structure.

Definition

What exactly are *persistent data structures*?

Two words: **Time Travel**

- It's data structures which preserve previous versions of itself. Essentially: data structures with archaeology.
- A general concept which can be applied to any data structure.
- If you can access previous versions but only modify the latest version, it's **partially persistent**. If you can access and modify all prior versions, it's **fully persistent**.

Example

Let's say you want to implement a balanced binary search tree with the following three operations:

Example

Let's say you want to implement a balanced binary search tree with the following three operations:

- **Insert** some value x at some time t
- **Delete** some value x at some time t
- **Find** if some value x was in the data structure at some time t.

Example

Let's say you want to implement a balanced binary search tree with the following three operations:

- **Insert** some value x at some time t
- **Delete** some value x at some time t
- **Find** if some value x was in the data structure at some time t.

Furthermore, we want all the above operations to run in $O(\log n)$ time.

Naive Approach

An easy **brute force** solution:

Every time we make some modification to the data structure at some time t, we can simply copy the entire data structure with the new modification and label it with the time stamp t.

Naive Approach

An easy **brute force** solution:

Every time we make some modification to the data structure at some time t, we can simply copy the entire data structure with the new modification and label it with the time stamp t.

To query a specific version of the data structure at a particular time, we then only have to do a single binary search, after which we can access the data structure at that time.

Naive Approach

An easy **brute force** solution:

Every time we make some modification to the data structure at some time t, we can simply copy the entire data structure with the new modification and label it with the time stamp t.

To query a specific version of the data structure at a particular time, we then only have to do a single binary search, after which we can access the data structure at that time.

This requires $O(n)$ extra space and time for each modification, hence we need a better approach

Offline Approach

- If all queries are given beforehand (i.e. **offline** algorithm) we can simply take in all input, sort the queries by time and then maintain the data structure without persistence as per normal.

Offline Approach

- If all queries are given beforehand (i.e. **offline** algorithm) we can simply take in all input, sort the queries by time and then maintain the data structure without persistence as per normal.
- Then, whenever we get to the version of the data structure which we need to query we then do the query operation and obtain some answer.

Offline Approach

- If all queries are given beforehand (i.e. **offline** algorithm) we can simply take in all input, sort the queries by time and then maintain the data structure without persistence as per normal.
- Then, whenever we get to the version of the data structure which we need to query we then do the query operation and obtain some answer.
- Since we have all the queries in increasing time order, we do not need to keep prior versions of our data structures (no persistence required).

Offline Approach

- If all queries are given beforehand (i.e. **offline** algorithm) we can simply take in all input, sort the queries by time and then maintain the data structure without persistence as per normal.
- Then, whenever we get to the version of the data structure which we need to query we then do the query operation and obtain some answer.
- Since we have all the queries in increasing time order, we do not need to keep prior versions of our data structures (no persistence required).
- Doing this for all queries, then rearranging the answers to the order they were originally given in, gives us a valid solution.

Fat Nodes

In general, we'll have to do operations online, so we can't rely on the offline approach. Our first attempt is to consider an approach known as keeping **Fat Nodes**:

Fat Nodes

In general, we'll have to do operations online, so we can't rely on the offline approach. Our first attempt is to consider an approach known as keeping **Fat Nodes**:

When updating our data structure, instead of replacing the value of some node to a new one, we instead keep an array of values in our node, keeping track of what values were in the node at what time.

Fat Nodes

In general, we'll have to do operations online, so we can't rely on the offline approach. Our first attempt is to consider an approach known as keeping **Fat Nodes**:

When updating our data structure, instead of replacing the value of some node to a new one, we instead keep an array of values in our node, keeping track of what values were in the node at what time.

Essentially: We add a *modification history* to each node.

Fat Nodes: Time Analysis

Every modification only takes $O(1)$ time and space, just adding a new value with timestamp to an array within the node. For full persistence, we would need to keep a version history tree (instead of just an array) within the node, so modification time would then be $O(\log m)$

Fat Nodes: Time Analysis

Every modification only takes $O(1)$ time and space, just adding a new value with timestamp to an array within the node. For full persistence, we would need to keep a version history tree (instead of just an array) within the node, so modification time would then be $O(\log m)$

To access nodes, we would need to do a binary search within each node as we traverse the tree in order to access the right pointers for some particular time. This gives a multiplicative slowdown factor of $O(\log m)$

Path Copying

Another approach is to instead just make a copy of the node with the new modification. We have to also then make copies of all ancestors of the node which point to the new node. This is called **path copying**

Path Copying

Another approach is to instead just make a copy of the node with the new modification. We have to also then make copies of all ancestors of the node which point to the new node. This is called **path copying**

Note that a new root will be copied for each modification made.

Path Copying

Another approach is to instead just make a copy of the node with the new modification. We have to also then make copies of all ancestors of the node which point to the new node. This is called **path copying**

Note that a new root will be copied for each modification made.

This improves access time to just a single binary search for the root, after which the data structure can be queried as normal. (extra additive $O(\log m)$ slowdown)

Path Copying

Another approach is to instead just make a copy of the node with the new modification. We have to also then make copies of all ancestors of the node which point to the new node. This is called **path copying**

Note that a new root will be copied for each modification made.

This improves access time to just a single binary search for the root, after which the data structure can be queried as normal. (extra additive $O(\log m)$ slowdown)

Modification time, however, is $O(n)$ since in the worst case, the entire data structure will have to be copied.

Combining the two approaches

Noting the two main approaches given: **Fat Nodes** and **Path Copying**, we can combine these two approaches to obtain a persistent data structure which takes $O(1)$ amortised space and $O(1)$ amortised time.

Combining the two approaches

Noting the two main approaches given: **Fat Nodes** and **Path Copying**, we can combine these two approaches to obtain a persistent data structure which takes $O(1)$ amortised space and $O(1)$ amortised time.

Using the idea of fat nodes, instead of making nodes arbitrarily fat, we just keep one additional space within each node to store a single modification to that node (with the corresponding time stamp). This could be a modification to the node's value, pointers or whatever property it might have.

Combining the two approaches

Noting the two main approaches given: **Fat Nodes** and **Path Copying**, we can combine these two approaches to obtain a persistent data structure which takes $O(1)$ amortised space and $O(1)$ amortised time.

Using the idea of fat nodes, instead of making nodes arbitrarily fat, we just keep one additional space within each node to store a single modification to that node (with the corresponding time stamp). This could be a modification to the node's value, pointers or whatever property it might have.

We'll call this space it's **modification box** (or **mod box**).

Combining the two approaches: Algorithm

When we do a *modification*, we simply check if it's mod box is empty. If so, we store the updated value in the mod box with the appropriate timestamp. If the box is full, we copy the node and immediately store the new value in this node (keeping the mod box empty). We then recurse all the necessary modifications to the parent.

Combining the two approaches: Algorithm

When we do a *modification*, we simply check if it's mod box is empty. If so, we store the updated value in the mod box with the appropriate timestamp. If the box is full, we copy the node and immediately store the new value in this node (keeping the mod box empty). We then recurse all the necessary modifications to the parent.

To *access* a node, we first do a binary search to find the correct root to start with. We then simply compare the time to the timestamp in the mod box. If the box is empty or the time is before the box's timestamp, then we just consider the original value of the node. Else, we consider the modified value given in the mod box.

Combining the two approaches: Time Analysis

To access a node, we just have to do a single $O(\log m)$ binary search to find the correct root, after which it's just a $O(1)$ slowdown for each node we visit (must just check the value in the mod box for each node).

Combining the two approaches: Time Analysis

To access a node, we just have to do a single $O(\log m)$ binary search to find the correct root, after which it's just a $O(1)$ slowdown for each node we visit (must just check the value in the mod box for each node).

To modify a node, this *could* potentially take many steps, perhaps copying $O(n)$ nodes for some particular modification. However, when a node is copied it creates a new node with an empty mod box, resulting in the next modification to that node only being a single write to the mod box (without having to copy parent nodes). Averaging out the time and space required for all modifications, we obtain an amortised time and space of $O(1)$.

Geometry example

We now consider a geometry problem which, at first, doesn't seem to have anything in common with persistent data structures.

Geometry example

We now consider a geometry problem which, at first, doesn't seem to have anything in common with persistent data structures.

Geometry problem

Given a plane with various simple polygons and several query points, determine for each query point how many polygons the point lies within.

Geometry example

We now consider a geometry problem which, at first, doesn't seem to have anything in common with persistent data structures.

Geometry problem

Given a plane with various simple polygons and several query points, determine for each query point how many polygons the point lies within.

Note that, in 1 dimension, the problem is equivalent to determine the number of intervals a point lies within. This can be solved using **interval trees** (not to be confused with the trees used in range-min queries)

Geometry example: Algorithm

We can solve the original 2D problem by considering one of the spatial dimensions as *time*.

Geometry example: Algorithm

We can solve the original 2D problem by considering one of the spatial dimensions as *time*.

We break the plane into vertical slices at each vertex or at any point where lines intersect.

Geometry example: Algorithm

We can solve the original 2D problem by considering one of the spatial dimensions as *time*.

We break the plane into vertical slices at each vertex or at any point where lines intersect.

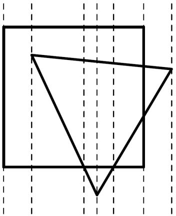

Figure: Example for two polygons, square and triangle

Geometry example: Algorithm

Note that within each vertical slice, no lines cross over, hence each slice is equivalent to the 1-dimensional case of noting how many intervals overlap with a point (plus a bit of linear algebra)

Geometry example: Algorithm

Note that within each vertical slice, no lines cross over, hence each slice is equivalent to the 1-dimensional case of noting how many intervals overlap with a point (plus a bit of linear algebra)

Now, given some point, we consider its x-coordinate and do a binary search to determine the vertical slice it lies in. Once we have the relevant vertical slice, all that's left is to determine the intervals within that slice that overlap with that point, which can be done in $O(\log n)$ time.

Geometry example: Algorithm

Note that within each vertical slice, no lines cross over, hence each slice is equivalent to the 1-dimensional case of noting how many intervals overlap with a point (plus a bit of linear algebra)

Now, given some point, we consider its x-coordinate and do a binary search to determine the vertical slice it lies in. Once we have the relevant vertical slice, all that's left is to determine the intervals within that slice that overlap with that point, which can be done in $O(\log n)$ time.

Keeping a separate interval tree for each vertical slice takes up $O(n^2 \log n)$ time and $O(n^2)$ space, hence we need a better approach.

Geometry example: Algorithm

Instead, we keep a single **persistent interval tree**, where each vertical slice in increasing x-coordinate corresponds to increasing intervals of time. Note that between two adjacent slices, that can only be one change, hence maintaining a persistent data structure by starting with an initial interval tree corresponding to the leftmost vertical slice and doing modifications as the x-coordinate increase gives us a solution which runs in $O(n \log n)$ preprocessing time, $O(n)$ space and $O(\log n)$ query time.

Therefore, the initial binary search to determine the vertical slice which the point lies in is equivalent to determining the *version* of the persistent interval tree which we must query.