The Prospects of Persistence

Ayan Chowdhury & Ivan Wei

University of Michigan - Ann Arbor
Department of Computer Science and Engineering

November 26, 2024
EECS 498-009

Overview

1. Partially Persistent AVL Trees

Overview

1. Partially Persistent AVL Trees
2. Confluently Persistent Deques

Overview

1. Partially Persistent AVL Trees
2. Confluently Persistent Deques
3. Bits of Implementation

Persistence

Ephemeral

An ephemeral data structure loses its previous state when updated.

Persistence

Ephemeral

An ephemeral data structure loses its previous state when updated.

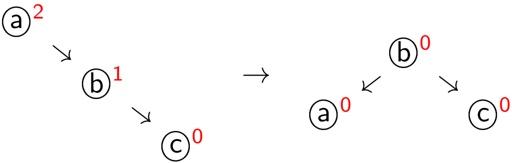
Partially Persistent

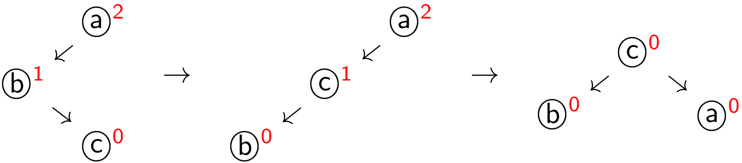
A partially persistent data structure keeps track of its state at every update, letting you query the state at any given time and update from the latest time

Ephemeral AVL Tree

A binary search tree which keeps track of the heights of nodes and balances accordingly to guarantee $O(\log n)$ queries and updates

Balancing an AVL Tree





Note that rebalancing a single node makes $O(1)$ pointer updates

Inserting in an AVL Tree

Insertion

We traverse down our tree comparing against the element we want to insert until we find a node with an empty child which can hold our element. We then point this node to a new node containing our new element. We then traverse back up the path reaching our inserted element and fix the heights and rebalance the tree.

Inserting in an AVL Tree

Proposition

An insertion modifies the pointers of $O(1)$ nodes in our tree.

Inserting in an AVL Tree

Proposition

An insertion modifies the pointers of $O(1)$ nodes in our tree.

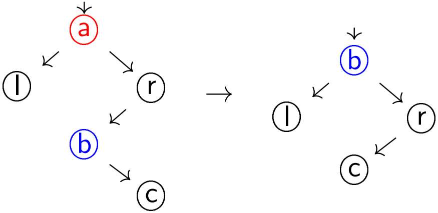
Proof:

We have that the inserted node results in one pointer update. When traversing up our tree, it holds that only a single rebalance will occur. This is because an insertion only adds elements to the tree, so when a rebalance occurs the height of the rebalanced node will become what it was originally.

Deleting in an AVL Tree

Deletion

We traverse down our tree comparing against the element we want to delete. If such an element exists, we consider different cases based on its children. If the node has no children, it is just removed. If it has one child it will be replaced by that child. If it has both children, it will be replaced by its successor, and pointers will be adjusted accordingly. We then traverse back up the path from either our deleted node or its successor and fix the heights and rebalance the tree.



Deleting in an AVL Tree

Note

After a deletion, a rebalance can cause the height of the entire subtree to decrease by 1, which can propogate rebalances up the tree. This results in $O(\log n)$ pointer changes.

Fat Node AVL

Modify an AVL tree to store a list of children pointers as opposed to just 1 set of children, and tag each pair with a timestamp.

Fat Node AVL

Fat Node AVL

Modify an AVL tree to store a list of children pointers as opposed to just 1 set of children, and tag each pair with a timestamp.

Updates append to the fat node with a new timestamp.

Fat Node AVL

Fat Node AVL

Modify an AVL tree to store a list of children pointers as opposed to just 1 set of children, and tag each pair with a timestamp.

Updates append to the fat node with a new timestamp.

Searches are given a timestamp and will binary search over the timestamps in a node to find the set of children to traverse.

Path Copy AVL

Path Copy AVL

Modify an AVL tree to store a list roots.

Path Copy AVL

Path Copy AVL

Modify an AVL tree to store a list roots.

Updates create a copy of each altered node and their ancestors. The copy of the root is added to the list of root nodes.

Path Copy AVL

Path Copy AVL

Modify an AVL tree to store a list roots.

Updates create a copy of each altered node and their ancestors. The copy of the root is added to the list of root nodes.

Searches will binary search to find a root with the corresponding timestamp and traverse down the tree given by that root.

Confluence

Fully Persistent

A fully persistent data structure keeps track of its state at every update, and lets you query and update at any given time.

Confluence

Fully Persistent

A fully persistent data structure keeps track of its state at every update, and lets you query and update at any given time.

Confluent

A confluent data structure is fully persistent and also allows for the versions of the structure at different times to be merged.

Deques

Deque

A deque is a linear collection which allows inserting to either end, querying either end, and deleting from either end.

Deques

Deque

A deque is a linear collection which allows inserting to either end, querying either end, and deleting from either end.

Proposition

Insertion on either end of a deque is equivalent to catenation, which takes two deques X and Y and returns the deque of the list of elements of X followed by the list of elements of Y.

Deque Tree

Deque Tree

A deque can be implemented as a tree, where the leafs give the elemtents, the left to right order of the leafs give the linear order over the elements.

Deque Tree

Deque Tree

A deque can be implemented as a tree, where the leafs give the elemtents, the left to right order of the leafs give the linear order over the elements.

Catenation

Deque trees X and Y can be catenated by taking the root of one tree and make it the left or rightmost child of the other root.

Deque Tree

Deque Tree

A deque can be implemented as a tree, where the leafs give the elemtents, the left to right order of the leafs give the linear order over the elements.

Catenation

Deque trees X and Y can be catenated by taking the root of one tree and make it the left or rightmost child of the other root.

Deletion

Deletions and queries are done by reading or deleting the leftmost or rightmost leaf.

Invariants of Deque Trees

Invariant

We maintain invariant that the internal nodes of our deque tree contain at least 2 children.

Linking by size

When catenating two deque trees, we make the tree with fewer leaves a child of the other.

Balancing Deque Trees

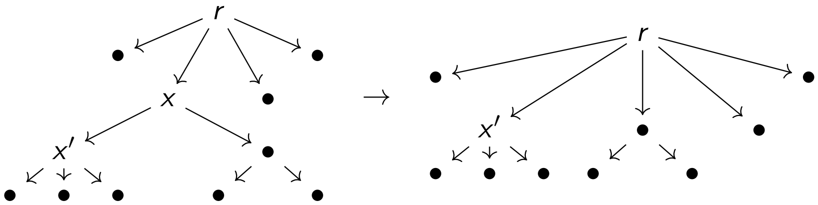
Deque Tree Pull

Let r be the root of a deque tree. Let x be the leftmost non-leaf child of r and let x' be the leftmost child of x. We define a pull to remove the edge between x and x', and make x' a new child of r, directly to the left of x. If x has only one child remaining, replace it with its child.

Balancing Deque Trees

Deque Tree Pull

Let r be the root of a deque tree. Let x be the leftmost non-leaf child of r and let x' be the leftmost child of x. We define a pull to remove the edge between x and x', and make x' a new child of r, directly to the left of x. If x has only one child remaining, replace it with its child.



Balancing Deque Trees

Proposition

A pull on a deque tree maintains its invariants.

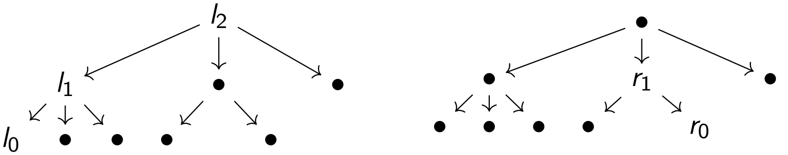
Theorem

If we link by size and pull k times after every deletion for some k, then the depth of the tree will be $O(\log n)$.

Spines of Deque Trees

Spine

We define a left spine of a deque tree to be a maximal bottom-up path $(x_0, \ldots, x_l)$ where x_0 is a leaf and x_i is the leftmost child of $x_i + 1$. We say the spine ends on x_l. Right spines are defined symmetrically.



Ephemeral Deque Tree with Spines

Let us store a deque tree via spines. Let T be a deque tree.

Ephemeral Deque Tree with Spines

Let us store a deque tree via spines. Let T be a deque tree.

We store $ls(T)$ and $rs(T)$, which are the left and right spines terminating at the root of T respectively.

Ephemeral Deque Tree with Spines

Let us store a deque tree via spines. Let T be a deque tree.

We store $ls(T)$ and $rs(T)$, which are the left and right spines terminating at the root of T respectively.

For an arbitrary node x with children $c_1, \ldots, c_i$, in left to right order, we store the right spine of c_1, the left spine of c_i, and the left and right spines of $c_2, \ldots, c_{i-1}$ in a linked list.

Ephemeral Deque Tree with Spines

Let us store a deque tree via spines. Let T be a deque tree.

We store $ls(T)$ and $rs(T)$, which are the left and right spines terminating at the root of T respectively.

For an arbitrary node x with children $c_1, \ldots, c_i$, in left to right order, we store the right spine of c_1, the left spine of c_i, and the left and right spines of $c_2, \ldots, c_{i-1}$ in a linked list.

We store spines as deques, with the leaves at the front. The elements in these deques are node values or pointers to the spines of its children.

Proposition

An internal node in deque tree is stored in exactly 2 spines, and is at the top of at least one of them.

Catenation

Consider deque trees X and Y. Without loss of generality, let us say that we catenate X and Y by placing the root of Y as the leftmost child of X.

Catenation

Consider deque trees X and Y. Without loss of generality, let us say that we catenate X and Y by placing the root of Y as the leftmost child of X.

First, we remove root(X) from $ls(X)$. Take this new $ls(X)$ and add it to the front of the list of children of root(X).

Catenation

Consider deque trees X and Y. Without loss of generality, let us say that we catenate X and Y by placing the root of Y as the leftmost child of X.

First, we remove root(X) from $ls(X)$. Take this new $ls(X)$ and add it to the front of the list of children of root(X).

Then add $rs(Y)$ to the front of the list of children of root(X).

Operations on Deque Tree with Spines

Catenation

Consider deque trees X and Y. Without loss of generality, let us say that we catenate X and Y by placing the root of Y as the leftmost child of X.

First, we remove root(X) from $ls(X)$. Take this new $ls(X)$ and add it to the front of the list of children of root(X).

Then add $rs(Y)$ to the front of the list of children of root(X).

Then we append root(X) to the end of $ls(Y)$, and set $ls(X)$ to be this new $ls(Y)$.

Confluent Deque Trees

Confluent Deque Trees

Let us assume the existence of persistent linked lists and confluent deques. We have that our persistent linked lists will point to spines given some timestamp. Let each element of any deque mark a timestamp of the persistent linked list of any of its nodes.

Proposition

Our confluently persistent deque T is then fully characterized by the timestamps of $ls(T)$ and $rs(T)$.

Confluent Deque Trees

We can remove the assumption of needing a confluent deque by defining our data structure inductively. Our structure only needs $O(\log n)$ confluently persistent deques to be implemented. Thus we can recurse, and when we have a sufficiently small deque, simply recreate copies rather than recursing further.

Theorem

A confluent deque tree implemented with spines can be implemented with $O(\log^* m)$ worst-case time and space per deletion, where m is the total number of deque operations, and $O(1)$ worst-case time and space for all other operations.

Bits of Implementation

Partially persistent AVL trees can be used to solve problems like 2-dimensional planar point search efficiently. You can solve other similar problems where you model some coordinate as time.

Bits of Implementation

Partially persistent AVL trees can be used to solve problems like 2-dimensional planar point search efficiently. You can solve other similar problems where you model some coordinate as time.

They can also serve general purpose use cases. Persistent data structures are well suited for a different flavor of computer science:

Functional Programming!

Persistence and Functional Programming

One of the important principles of functional programming are to minimize modified state, and especially minimize shared state.

One of the important principles of functional programming are to minimize modified state, and especially minimize shared state.

Persistent data structures are then one of the core ways of dealing with state. Rather than modifying memory, just append to it, and read from whatever timestamp you want to.

Persistence and Functional Programming

The paper detailing the construction of a confluently persistent deque mentioned that a fully functional implementation exists!

Designing our implementation was initially difficult due to a shift in perspective, but there were many pieces which nicely snapped together due to the nature of persistent data structures.

Node Arena

```rust
pub struct FatNodeAvl<Data: Ord> {
    node_arena: Vec<FatNode<Data>>,
    root_nodes: Vec<RootNode>,
    last_time: u64,
}

fn insert(&mut self, item: Self::Data) -> Self::Timestamp {
    // Allocation
    self.node_arena.push(FatNode {
        datum: item,
        height: 1,
        children: Vec::new(),
    });

    ...
}
```

Irrelevance of Order

```rust
fn rotate_right(&mut self, old_root_ptr: usize, timestamp: u64) -> usize {
    let old_root = &self.node_arena[old_root_ptr];

    let old_root_left = old_root.left;
    let old_root_right = old_root.right;

    let new_root = &self.node_arena[old_root_left];

    let new_root_left = new_root.left;
    let new_root_right = new_root.right;

    old_root.set_height[old_root_height];
    new_root.set_height[new_root_height];

    old_root.modify_left[timestamp, new_root_right];
    new_root.modify_right[timestamp, old_root_ptr];

    new_root
}
```

Batching Timed Updates

```
fn get_node(&self, update_cache: &HashMap<usize, CopyNode>, node_ptr: usize) -> CopyNode {
    match update_cache.get(&node_ptr) {
        Some(node) => node.clone(),
        None => self.node_arena[node_ptr].clone(),
    }
}

fn modify(
    &self, update_cache: &mut HashMap<usize, CopyNode>, node_ptr: usize,
    height: u64, new_left_ptr: Option<usize>, new_right_ptr: Option<usize>,
) {
    update_cache.insert(
        node_ptr,
        self.get_node(&update_cache, node_ptr)
            .update(height, new_left_ptr, new_right_ptr),
    );
}

fn modify_height(
    &self, update_cache: &mut HashMap<usize, CopyNode>,
    node_ptr: usize, height: u64,
) {
    self.modify(
        update_cache,
        node_ptr,
        height,
        self.get_node(&update_cache, node_ptr).left,
        self.get_node(&update_cache, node_ptr).right,
    );
}
```

You can refer to ‘Confluently Persistent Deques via Data-Structural Bootstrapping’ by Buchsbaum and Tarjan for more information about confluently persistent deques:

You can visit (https://github.com/Ca7Ac1/persistence) if you want to see our implementation of different persistent AVL trees.

References

1 A. L. Buchsbaum and R. E. Tarjan, "Confluently Persistent Deques via Data-Structural Bootstrapping," Journal of Algorithms, vol. 18, no. 3, pp. 513-547, May 1995, doi: 10.1006/jagm.1995.1020.

Thank you!

Questions?