MIT 6.851 - Advanced Data Structures Lecture 1 - Persistent Data Structures

Lecture 1 - Persistent Data Structures

The first two lectures discuss temporal data structures, which are data structures that allow you to access past versions of the data structure. This is useful for applications like undo functionality in text editors or version control systems. It is kind of like a time machine for data structures.

The first lecture introduces the concept of persistent data structures, which are data structures that preserve their previous versions when modified, and for which you can access any version at any time.

Types of Persistence

There are four types of persistence:

1. **Partial Persistence**: You can access any version of the data structure, but you can only modify the most recent version. This means that the versions are ordered linearly.
2. **Full Persistence**: You can access any version and modify any version. This means that the versions form a tree structure, where each version can have multiple children.
3. **Confluent Persistence**: You can access any version and modify any version, and you can also merge two versions together. This means that the versions form a directed acyclic graph (DAG) structure, where each version can have multiple parents.
4. **Functional Persistence**: This is a special case of full persistence where the data structure is immutable, meaning that once it is created, it cannot be modified. Instead, you create a new version of the data structure with the modifications.

We will look at how to implement partial and full persistence, and discuss features of confluent and functional persistence, but some aspects of confluent and functional persistence are still open problems.

Partial Persistence

We use the pointer machine model to implement all types of persistence. This means that the data structures we consider will have nodes and pointers, such as trees, linked lists, etc. We can implement partial persistence efficiently if we are guaranteed that the number of pointers going *to* a node is bounded by a constant.

To convert any pointer machine data structure to a partially persistent data structure, we can use the following approach. For each node in the original data structure, we create a new node that contains:

- A read-only area of the fields of the node containing their original values (values that the fields were initialized to).
- An area for back-pointers, such that for every pointer that comes into this node, we store a pointer going back to the node that points to this node. This is needed so that when this node eventually gets "full", we can move this data to a new node, in which case the pointers pointing to this node will need to point to the new node instead.
- A constant amount of space for storing modifications to the fields of this node. This is like a log which will be written to every time a field of this node is modified.

We claim that using this approach, we can implement partial persistence in $O(1)$ amortized time per operation.

1. Reading a field is simple: we just read the value of the field, and then go through the mods log and see if the value was changed.
2. To write to a field, we first check if the mods log is full. If it is not, we just write the new value to the mods log by storing a tuple of the form `(version, field, new_value)`. If the mods log is full, we create a new node `n'` that contains the same fields with their latest values, the same back pointers, and a new empty mods log. We then look at all the pointers coming into the old node `n`, and for each pointer to `n` from a different node `m`, we recursively call a write on `m` to change the given pointer to point to `n'` instead of `n`.

Writes can clearly take a lot of work if the mods log is full, but it turns out that the amortized cost of writes is still $O(1)$, because we only create a new node when the mods log is full, and we only do this once per node. The number of nodes created is proportional to the number of writes, so the amortized cost is still $O(1)$.

Full Persistence

To implement full persistence, we can use a similar approach to the one we used for partial persistence, but now the versions are not linear, but rather form a tree structure. Most of the process for converting a pointer machine data structure to a

fully persistent data structure is the same as for partial persistence, but we need to make some changes to handle the tree structure.

We "linearize" this tree of versions by simply using its inorder traversal. Furthermore, we use a special data structure called an order-maintenance data structure, which allows us to tell if a version is an ancestor of another in $O(1)$ time, and also lets us insert versions before or after any other version in $O(1)$ time. This is useful because we need to be able to insert new versions into the tree structure as we modify the data structure.

Now for each node we store `2(d + p + 1)` mods, where `d` is the number of fields in the node, and `p` is the number of back pointers from this node to other nodes. The `+1` is only for the amortized analysis to work.

1. Reading a field is the same as in partial persistence: we read the value of the field and then go through the mods log to see if the value was changed. However, we need to check the order-maintenance data structure to see if the version we are reading is an ancestor of the current version. We only consider the mods that are in the current version or in an ancestor of the current version.
2. To write to a field, we first check if the mods log is full. If it is not, we just write the new value to the mods log by storing a tuple of the form `(version, field, new_value)`. If the mods log is full, we create a new node `n'` and essentially split `n` into two nodes: `n` and `n'`. We store half of the modes in `n` and the other half in `n'`. We then determine all nodes which need to point to `n'` instead of `n`, and recursively call a write on those nodes to change the pointers to point to `n'` instead of `n`.