52 Persistence

This chapter will be devoted to efficient conservation of history of data structures, so called *persistence*.

52.1 The Notion of Persistence

The original state of a typical data structures is lost when an operation is used to modify it. Of course, usually there is no harm coming from it. In certain situations however, we need to delve into the past. Let us therefore form a hierarchy of data structures with respect to preservation of history.

- *Ephemeral data structures* are the most widespread. Past states are destroyed by "destructive" updates and irretrievably lost.
- *Semi-persistent data structures* (sometimes partially persistent) still support modifying operations only on the most recent version. Updates produce a new version of the structure. Historical versions form a linear order. Past states can be reconstructed efficiently for reading.
- *(Fully-)persistent data structures* directly extend their semi-persistent counterparts. An update can be done on any version of the structure, resulting in a rooted tree of versions. An update corresponds to adding a new child vertex to an existing vertex in the tree.
- *(Purely) functional data structures* permit no changes to data once written. Naturally, persistence is guaranteed under such an assumption, as all versions appear as if just created by the last update. This enforced immutability completely freezes past version in time. This concept is intrinsic to functional languages such as Haskell, but proves to be useful even in imperative programming. Multi-threading is easier when there is confidence that existing objects will not be modified.

52.2 Basic Constructs

We will explore several easy concepts, which we will later combine to reach optimal persistent pointer-based structure. Before that however, we notice that some data structures are persistent in their default implementation. Take stack for example.

Persistent Stack

We may implement stack as a forward-list. In that case every item contains a pointer to the next and the structure is represented by a pointer to the head of the forward-list. This implementation is naturally persistent. We can work with earlier versions by pointing to previous heads of the forward-list. Pushing new element means creating a new head pointing to head of the version chosen to precede newly this created version. Deleting means setting the element which directly follows head of the chosen version as the new head.

Persistence through Path-Copying

Let us now turn to binary search trees. Binary search tree can be converted into a fully-persistent one rather easily if space complexity is not a concern. One straight-forward approach to achieve this is called *path-copying*. It is based on the observation that most of the tree does not change during an update.

When a new version of the tree should by created by delete or insert, copies are created for vertices that are to be hanged by the operation and their ancestors. This typically means that only path from the inserted/deleted vertex to the root is duplicated, plus constant number of other vertices close to the path (due to rebalancing). The changes are only applied to the copies. Pointers in new vertices are updated to point to new copies of corresponding vertices where those copies were created. The root of the newly created version is the copy of the original root. Thus the new version now shares some vertices with the old version. Here we tacitly assume that only pointers to children are stored. Updating the root in a tree with pointers to parents would involve creating copies for all nodes in the tree.

Insertion of a vertex G is depicted in figure 52.1. We can see an original tree with vertices A, B, C, D, E, F. The thick vertices are created by the insert operation.

This method yields a functional data structure since the old version of the tree was technically not modified and can still be used. With reasonable variants of binary search trees, achieved time complexity in a tree with n vertices is $\Theta(\log n)$ per operation and $\Theta(\log n)$ memory for insert/delete.

The downside of this method is the increased space complexity. There is no apparent construction that would not increase memory complexity by copying the paths.

This outlined method is not exclusive to binary search trees. It may be used to obtain functional variants of more variants of pointer data structures, of which binary search trees are a prime example.

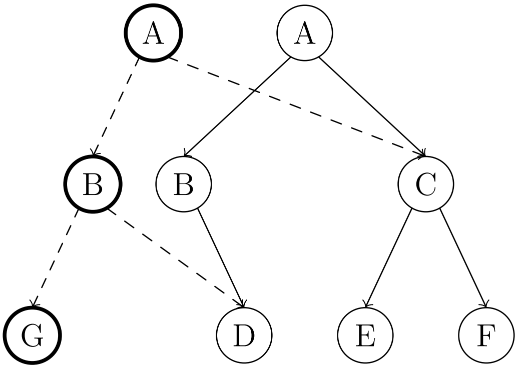

Figure 52.1: Path copying

Fat Nodes

To limit memory consumption over path-copying we may rather choose to let vertices carry their properties for all versions. This in reality means that we must store a collection of changes together with versions when those changes happened. This makes a vertex alike a dictionary with keys being versions and values being the descriptions of changes.

When semi-persistence is sufficient, upon arriving at a vertex asking for the state in version A, we do the following: We must go through the collection of changes to figure out what the state in version A should be. We start with the default values and apply all changes that happened earlier than A in chronological order overwriting if there are multiple changes for the same field or pointer. This process yields the correct state of the vertex for version A.

There are several options for protocol of changes. In fact, it might be easier to just copy all other fields the vertex possesses as well if one of them changes. One change will therefore hold new values for all fields and pointers of the vertex. By embedding some data structure into a fat node, identification of the correct fields may be faster.

For full-persistence we also need to resolve the issue of how to effectively determine which changes should be applied. It will be addressed later through introduction of total ordering of all versions.

Looking at the memory consumption, it appears clear that the amount consumed stems from the total number of changes done by the balancing algorithm across all operations.

What remains to solve though is to choose the right kind of collection for versions of changes for one vertex. Surely using linked lists or arrays will inevitably lead to unsatisfactorily inefficient processing of applicable changes. Unfortunately as it turns out, even other data structures will let us down here and complexity will increase.

With this approach, we can reach space complexity that is linear with the total number of changes in the tree. On the other hand execution time will be hurt. If we have a tree with m versions, time spent at each vertex increases to $\mathcal{O}(\log m)$ with collections inside vertices implemented as binary search trees. This can be improved by using more suitable data structures e.g. van Emde Boasian.

52.3 Pointer Data Structures and Semi-Persistence

Following up on the idea of fat nodes, let us limit their size to achieve identification of applicable version in constant time. We will explain this technique only for semi-persistence. Full persistence requires the use of a few extra tricks and establishing an ordering on the versions.

A *fat node* stores a collection of standard vertices indexed by versions. We call values of this collection *slots*. The maximum size of this collection is set to a constant which is to be determined later. Temporarily, we will allow the capacity of a fat node to be exceeded. This will have to be fixed however, before the ongoing operation finishes. By placing a restriction on the size we may circumvent the increased complexity of search within one vertex. Instead of copying the vertex, we simply add a new slot into the collection. Provided the maximum has not been exceeded yet, this insertion of a slot stops the propagation of changes toward the root. The reader should recall that this propagation was the major weakness of path-copying. Because of the limit on size of this collection, it may be implemented simply as a linked list.

Contents of one slot are a version handle, all pointers a vertex would have, and some fields, notably key and value as a bare minimum. Not all fields need to be versioned. For example balancing information may be stored only for the latest version, i.e., in a red-black trees color is only used for balancing and is thus not needed to be persisted for old versions. (Until full-persistence comes into play.)

One vertex in the original binary search then corresponds to a doubly-linked list of fat nodes. When the vertex changes, a new state of the vertex is written into a slot of the last fat node in the list. As all slots become occupied in the last fat node and it needs to be modified, new fat node is allocated.

Modifications of a single vertex during one operation are all written to single slot, there is no need for using more slots.

We will need to be able to determine for any fat node x which other fat nodes have their most recent slot pointing to x. This problem is resolved by adding some number of *inverse pointers* (in contrast to proper pointers of the binary search tree) to each series of fat nodes of the same vertex. We define an invariant of inverse pointers: If vertex v points to vertex u in the most recent version, then u must contain an inverse pointer to v. This invariant is maintained by also changing the inverse pointers whenever some proper pointers change.

When a new fat node x is allocated, one of its slots is immediately taken. Pointers must be updated in other fat nodes whose latest slot pointed to the fat node preceding x in the list. This is done by going through vertices pointed to by the inverse pointers and either creating slots (copying all values from the latest slot and replacing pointers to the predecessor by pointers to x), or directly updating the pointers if the slot for the right version is already present. Also inverse pointers must be updated to x for vertices pointed to by x in the latest version.

Recursive allocations may be triggered, which is not a problem if there is only a couple of them. This is ensured by setting the size of fat nodes suitably. The order in which these allocations are executed can be arbitrary. We can place an upper bound on the number of newly allocated fat nodes – total number of vertices in the tree (including deleted vertices). At most one new slot is occupied for every vertex in the tree.

Consider a tree with one vertex A and a sequence of updates. First B is inserted as a left child of A, then C is inserted as a left child of B, then D is inserted as a right child of A, finally B is deleted and C becomes left child of A. These operations are captured by a schema of fat vertices in figure **??**. Inverse pointers for the latest version are dotted.

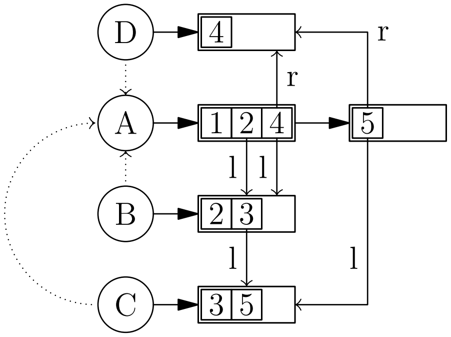

Figure 52.2: Fat Nodes

To take advantage of fat nodes, we need the balancing algorithm to limit the number of vertices that change in one operation. It is sufficient that the changes can be amortized to a constant number per update. Furthermore, we need a limit on the number of pointers that can target one vertex at one time.

Theorem: Consider any binary search tree balancing algorithm satisfying the following properties:

- There is a constant d such that for any n successive operations on initially empty tree, the number of vertex changes made to the tree is at most dn.
- There is a constant bound on the number of pointers to any one vertex at any time.

Then this algorithm with the addition of fat nodes for semi-persistence, consumes $\mathcal{O}(n)$ space for the entire history of n operations starting from an empty tree.

Proof: The number of pointer fields per vertex is denoted by p and the maximum number of vertices pointing to one vertex at a time by k. We then define the number of slots in one fat node as $s = k + 1$ and add k inverse pointers to vertices.

We define the potential of the structure as the total number of occupied slots in all fat nodes that are the last in their doubly-linked list. (Thus initially zero.) Allocation of a new fat node will cost one unit of energy. This cost can be paid from the potential or by the operation. We will show that the operation needs to be charged only a constant amount of energy per one vertex modification by the original algorithm (to compensate for the increase in potential or to pay for allocations), from which the proposition follows.

Allocation of a new fat node associated with the insertion of a new vertex into the tree is paid for by the insert operation.

All other modifications changing a field or a proper pointer must pay one unit of energy. Modification of inverse pointers does not need any extra space. For modifications triggered by the insert, delete, or the balancing algorithm directly the operation covers this cost. If a modification constitutes an insertion of a slot into an existing fat node, the unit of energy is deposited into the potential. Otherwise the unit is used to pay for allocation of a new fat node. This allocation causes the potential to decrease by k units. These can be used to give one unit of energy to each of up to k subsequent modification triggered by checking inverse pointers. If a slot for the same version already exists, no additional space is needed and the unit of potential is in vain.

All allocations of new fat nodes are thus paid for while the operation is charged only constant amount of work for every change it does. Consumed space complexity is bounded by the number of changes done by update operations. Thus space complexity is $\mathcal{O}(n)$. □

Regarding the time complexity, searching the correct slot in a fat node produces only constant overhead. Realizing that every operation can be charged on a memory allocation of a fat node such that there is a constant c depending only on the balancing algorithm such that for every allocated fat node the number of operations charged on it is at most c. Thus provided the conditions from the previous proposition, the cost to write changes into fat nodes is amortized to $\mathcal{O}(1)$ per operation.

Time complexity of operations with the tree depends on the original balancing algorithm used. With red-black trees for example, we get $\mathcal{O}(\log n)$ per operation from the original algorithm in addition to the amortized $\mathcal{O}(1)$. Here n is the size of the tree in the version that is queried.

52.4 Point Localization in a plane

Given a bounded connected subset of a plane partitioned into a finite set of faces, the goal is to respond to queries asking to which face a point p belongs. We limit ourselves to polygonal faces. One special case of particular importance of this problem is finding the closest point, i.e. when the faces are Voronoi cells.

To build the data structure we will follow a general idea of line sweeping. We start by sorting vertices of all faces by the first coordinate x. We will continually process these vertices in the order of increasing coordinate x. We maintain S, a sorted list of edges (in a semi-persistent binary search tree) during this processing. The list S contains edges that intersect with a sweeping line parallel to the secondary axis y in order of the intersections (sorted by the second coordinate y).

Initially the list is empty and we set the sweeping line to intersect the first vertex. When we start processing a new vertex, we imagine moving the sweeping line along the primary axis to intersect with the new vertex. We can easily observe that the order of edges cannot change during this virtual movement. (None of the edges can intersect except at a vertex.) It will happen, however, that edges must be either removed from the list or added to the list.

We cannot store keys inside S because the coordinates of intersections with the sweeping line change as it moves. To find the correct edge in S one must start in the root and at each vertex calculate the intersection with the sweeping line at the given x coordinate (in constant time).

We store pointers to the versions of S created by processing each vertex in the plane in a sorted array. During a query, the first step is to identify which version of S to use. This can be done via a binary search in versions of S. Then the face is identified by finding which edges in that version of S are closest to the searched point.

The number of vertices will be denoted by n, the number of edges is thus bounded by $3n$. This follows from the partitioning being a drawing of a planar graph. Complexity is therefore $\mathcal{O}(n \log n)$ for pre-processing and $\mathcal{O}(\log n)$ for one query.

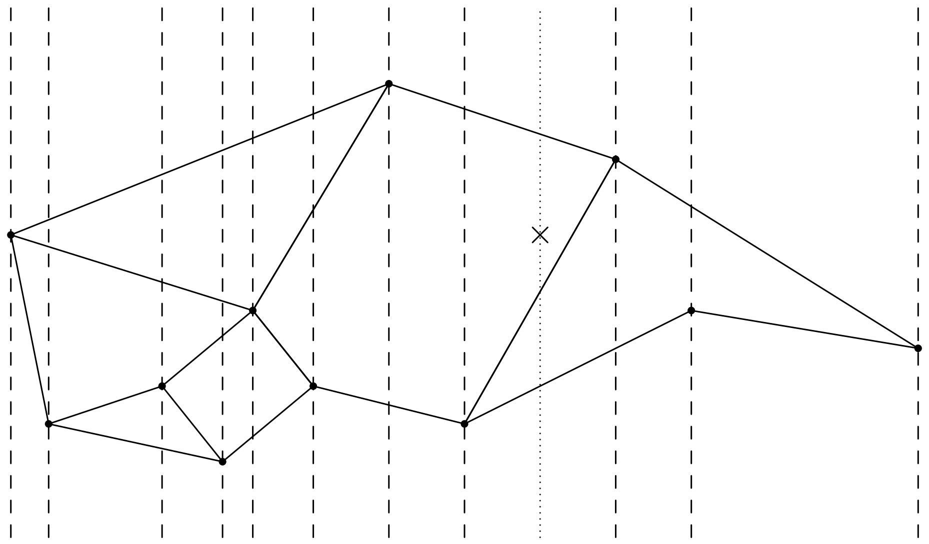
Figure 52.3: Point Localization

52.5 A Sketch of Full-Persistence

It is possible to obtain fully-persistent binary search trees with $\mathcal{O}(\log n)$ time complexity per operation if n is the total number of updates. Couple of obstacles must be overcome however.

- First of all, we need to decide on how to represent relationships between versions. This is done best via a rooted tree with ordered children. Root of the tree represents the initial empty tree. When an update on version v creates a new version, it is inserted into this tree as a child of v. We would like to introduce a dynamic linear order to all versions which would respect the structure of this version tree. Of course, comparison of version must be efficient, ideally taking $\mathcal{O}(1)$ and insert must take at most $\mathcal{O}(\log m)$. This problem is called *list ordering* and we address it in the next section.

- Another obstacle arises from worst-case complexity of updates. Imagine one update on the structure in version v taking $\omega(\log n)$ time or making changes to $\omega(1)$ nodes. Since this update can be hypothetically repeated unlimited number of times. Few binary search trees possess this property and those that do are typically very complicated. Some common types of binary search trees like red-black trees or weak-AVL trees can be represented in such a way that updates make changes to $\mathcal{O}(1)$ vertices.
- In semi-persistence it was not needed to store all information about nodes. It was sufficient to store pointers to children, key, and value for older versions. Fields useful for balancing were not needed for older versions. This is not true for full-persistence. All information must be stored.

52.6 List Ordering

Moving from semi-persistence to full persistence we encounter an obstacle – versions no longer form an implicit linear order. (By versions we mean states of the tree in between updates. We will also use some auxiliary versions not directly mappable to any such state.) Nonetheless, to work with fat nodes, we need to be able to determine slots that carry values correct for current version. To achieve this, we need to identify an interval of versions the current version would fall into. For this purpose we will try to introduce an ordering to versions of the persistent data structure.

Versions do form a rooted tree with the original version in the root. We can also order children of every version by the order of their creation (latest creation first). We then define the desired ordering as the order in which the versions are discovered by a depth-first search respecting the order of children.

With the ordering defined, we still need a way to efficiently represent it in memory. The operations we really need are two:

- `InsertSuccessor(Version)` – this operation will insert a new version between `Version` and its successor (if any). The newly created version is returned.
- `Compare(VersionA, VersionB)` – returns 1, -1, or 0 indicating whether `VersionA` precedes, succeeds, or is equal to `VersionB`.

We will strive to find a way to assign an integer to each version, these integers will be comparable in constant time. This assignment problem is called *list-labeling* and there are several options to tackling it.

Suppose we want to be able to assign up to m labels. The straight-forward idea would be to assign 0 to the first item and 2^m to an artificial upper bound. All newly inserted will be assigned the arithmetic mean of its successor and predecessor. Now, we see that if $v = (p+s)/2$ and $2^k \mid p, s$, then $2^{k-1} \mid v$. The first two integers are divisible by 2^m, so by induction item inserted by i-th subsequent inserted has label divisible by at least 2^{m-i}, thus guaranteeing capacity $\Omega(m)$.

Let us denote the total number of updates to the persistent tree by n. It would be reasonable to assume that arithmetic operations on non-negative integers less than or equal to n can be done in constant time. We are therefore permitted to create only $\mathcal{O}(\log n)$ versions using the straight-forward idea if constant time per operation is needed. This is not sufficient.

To preserve the speed of semi-persistence, `Compare` must take $\mathcal{O}(1)$ and `InsertSuccessor` $\mathcal{O}(\log n)$ at least amortized. Weight-balanced trees that were introduced in the previous chapter are ideally suited for this purpose.

Every vertex in the tree will correspond to one version.

For every vertex in the weight-balanced tree, we will store encoding of the path from the root to it in form of sequence of zeros and ones. 1 for going to a right child and 0 for a left child. Also distance from the root is stored in every vertex. We know that height of the tree must be logarithmic, so the encodings are made of $\mathcal{O}(\log n)$ each and can be interpreted as integers in binary notation. Therefore we can store these encodings as integers.

Comparison of versions then translates to comparison of these integers. Suppose we compare two vertices u and v with distances from the root d_u and d_v and paths encoded as integers e_u, e_v. We only want to compare the first d bits, where $d = min(d_u, d_v)$. In order to compare only the first d bits of the encodings, one of e_u, e_v can be divided by an appropriate power of two. If one of the compared numbers is greater, the corresponding vertex is clearly greater. In case of equality, either $u = v$, or one is an ancestor of the other. We can decide between the two options by comparing e_u and e_v for equality for example. If the latter is true and u is without loss of generality ancestor of v, the bit at position $d+1$ in the encoding of v determines whether v is the left or right subtree of u. We have demonstrated that the comparison can be made in $\mathcal{O}(1)$ time.

Inserting a successor version is also simple. Rebuilding of some subtree will not change order of versions as all path encodings will be recalculated. Integer arithmetic can be used to efficiently update encodings of paths to the root.

Our use of list ordering will involve one more trick. Suppose we have some versions a, b, c in that order and a fat node with slots for a and c. When we insert a successor a' to version a and a slot for that version, we by mistake modify also state of the vertex for version b. Therefore, we also need to undo the changes by creating a'', a successor to a', and inserting a slot for a'' directly after the slot for a'. This would result in a fat node with slots for a, a', a'', and c.

Before moving to the next section, we remark that Tsakalidis found a method to get $\mathcal{O}(1)$ amortized complexity for insert and delete with weight-balanced trees via indirection.

52.7 Ordered File Maintenance

In this section we will try to address the list-order problem in a bit different fashion. We want to store items n in our ordered list in one array of size $\mathcal{O}(n)$ – we allow interspersed empty records. This constraint forces us to leave our method of using paths in a tree as keys. On the other hand, in a sorted array comparison in constant time is simple.

As already mentioned, our structure at its core is an array. Conceptually however, we think of it as a complete binary tree. We also imagine that indirection is used on blocks of size roughly $\mathcal{O}(\log n)$ and these blocks are leaves of our conceptual tree.

Every level of our conceptual tree has a prescribed density of records. Precisely, density means that ratio of the count of all occupied records in all blocks which are leaves in a subtree of a given vertex to all records in leaves of that subtree. When we have a vertex v in distance i from the root, its density must fall into the interval $[1/2-i/(4h), 3/4+i/(4h)]$, where h is the height of the conceptual tree. This interval is called *standard density*. We can observe that the density constraint is more strict for nodes closer to root. For all nodes in the tree standard density will be maintained.

Insertion will work in two steps. First of all, new entry is inserted into the correct block. Then we check density of that block. If the density is nonstandard we proceed towards the root of the conceptual tree until a vertex with standard density is found. When we reach a node with standard density, records in its subtree must be distributed among its blocks evenly. This step also restores density for all other vertices inside the subtree. If even the root has nonstandard density (higher than standard of course), the size of the array is increased be a constant factor and all records distributed evenly inside it. In this case new size of blocks is chosen and new conceptual tree is considered.

Deletion is very similar to insertion with the difference that when root is found to have nonstandard density, the array must be shrunk by a constant multiple instead of expanded.

Now that we have described the algorithm, let us find out what the complexity is.

Theorem: A sequence of n inserts and deletes on empty OFM data structure takes $\mathcal{O}(n \log^2 n)$.

Proof: For simplicity we only analyze insert. Assume that redistribution is needed at level i during insert. That means in particular than one child has exceeded standard density of $\frac{3}{4} + \frac{i+1}{4h}$ while this vertex has density at most $\frac{3}{4} + \frac{i}{4h}$. This implies that since the last redistribution at least $\frac{1}{4h}$ times the count of records in this subtree have been inserted. Redistribution of records takes time linear with the size of the subtree. This means that to pay for this redistribution, it suffices for those inserted records to pay $\mathcal{O}(\log n)$. Of course this must be paid for every level of our conceptual tree, bringing the total to $\mathcal{O}(\log^2 n)$.

Similarly, if expansion of the array is needed, this can be amortized through paying a constant for every insert. □

Exercises

1. Consider what properties of (semi-)persistent binary search trees change when capacity of fat nodes is increased above the value used in the proof.

2. What if we defined fat nodes for semi-persistence differently? Suppose slots only contained values of changed fields and pointers. Thus each fat node would carry a set of default values for every field and pointer, these values are used if not overridden by a slot. How would the complexity change?