Chapter 3

Persistence

This chapter introduces the topic of *persistence* in data structures. A dynamic data structure evolves through time as elements are inserted and deleted. Usually, operations on the data structure are always performed on the most recent version. In contrast, a *persistent* data structure is one that allows access to all previous versions of the data structure.

Since it may not immediately be obvious that persistence is useful, we begin this chapter with a motivating example from the field of computational geometry. As it happens, persistence plays a central role in many algorithms for computational geometry problems.

3.1 Next Element Search

Let $S = \{s_1, \ldots, s_n\}$ be a set of horizontal line segments in the plane. The *next element search* problem asks us to proprocess S so that, given query point q we can return the element of S that is directly above q (see Figure 3.1), or **nil** if no such element exists.

The next element search problem can be solved in the following way (see Figure 3.2): Sort the endpoints of S by x-coordinate. Create a dictionary D that will store intervals sorted by their y-coordinates. We use the sorted list of endpoints to sweep the plane with a line from left to right. When

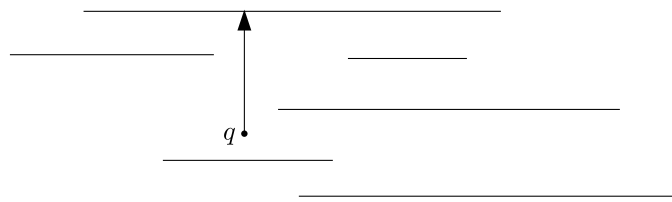

Figure 3.1: The correct answer for query point q is the line segment directly above q.

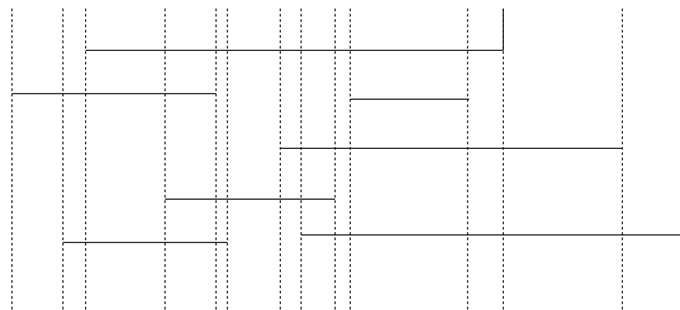

Figure 3.2: The next element search problem can be solved by partitioning the plane into vertical strips and building a dictionary for each strip.

the line passes over the left endpoint of a segment we insert that segment into D. When the line passes over the right endpoint of a segment we delete that segment from D. In both cases, we save a copy of D and store all these copies sorted by the x-coordinate of the endpoint that created them.

To perform next element search for a query point q, we perform a binary search on the x-coordinate of q to find the copy D' of D that contains exactly the segments that are above and below q. We then search D' to find the segment that is directly above q. The first binary search takes $O(\log n)$ time and the second search takes $O(\log n)$ time if we implement D using an efficient dictionary data structure. Therefore, the cost of performing next element search with this data structure is $O(\log n)$.

What is the cost of constructing the data structure? Clearly the dictionary D never contains more than n elements, so it can be copied in $O(n)$ time. Since we have to do this $2n$ times, the overall cost of copying is $O(n^2)$. The costs of other operations (sorting endpoints and insertions and deletions into D) is $O(n \log n)$, so the total cost of building the data structure is $O(n^2)$.

Plane sweep provides a nice simple, intuitive solution to the next element search problem. However, if we want to do better with the plane sweep approach we need a faster way to copy the dictionary D. But if D has size $\Omega(n)$ how can we copy it any faster? The trick is that each copy of D differs only from the previous one by one insertion or deletion, so we only need to copy the parts that change.

3.2 Path Copying

Suppose we have a dictionary implemented as a balanced binary search tree T. Recall from Chapter 2 that when we insert a key k into a balanced binary search tree we create a new leaf containing k and then rebalance T by performing rotations on some of the nodes on the path, $\text{path}(k, T)$, from the newly created leaf to the root of T. These rotations only modify the nodes of $\text{path}(k, T)$, so the entire insertion process only modifies the nodes of $\text{path}(k, T)$.

Suppose that, before we perform insertion of the key k into the tree T, we first copy all nodes on $\text{path}(k, T)$ (see Figure 3.3). We now have two search trees; the original tree T is still valid, and the a new tree T' whose root is a copy of the root of T (see Figure 3.3). Now, if we do an insertion on T'

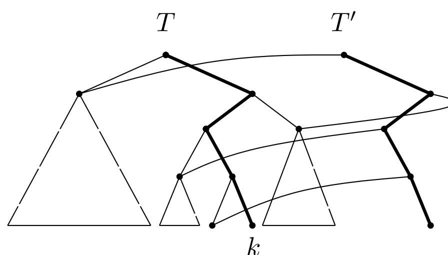

Figure 3.3: Before inserting k into T we copy all nodes on path(k, T).

we know that the insertion only modifies nodes on path(k, T′), which are not nodes of T. Therefore, after the insertion, T′ is balanced binary search tree that contains k and T is a balanced binary search tree that does not contain k. Copying path(k, T) before insertion does not increase the insertion time by more than a constant factor, and requires $O(\log n)$ additional space.[1]

Next we turn to the problem of deletions. To delete key k, we first simulate the deletion of key k to determine which nodes of T will be modified by the deletion. Now, for each node v that is modified, we copy v and all the nodes on the path from v to the root of T, doing this in such a way as to avoid copying any node more than once. This gives us a new tree T′ whose root is a copy of root(T). We then perform the deletion on T′. Since this deletion does not modify any nodes of T, we are left with a tree T that contains k and a tree T′ that does not contain k. Furthermore, any reasonable deletion algorithm runs in $O(\log n)$ time, and only modifies node v if it reaches v by following a path from root(T) to v. Therefore, the number of nodes copied by this scheme, and hence the overall running time and space requirement is $O(\log n)$.

For concrete examples of insertions and deletions, consider a treap T as described in Section 2.2. To insert the key k we copy path(k, T), append a new node (with key k) and then use rotations to move k upwards in the tree. It is easy to verify that the only nodes modified by these rotations are those on path(k, T).[2] To delete a key k, we make a copy of path(k, T) and then use rotations to move k downwards in T until it becomes a leaf. Each rotation modifies two nodes: One node is a copy of the node with key k. Therefore, before performing the rotation, we make a copy of the second node and do the rotation with the copy rather than the original.

None of the work of copying increases the running time of treap operations by more than a constant factor, so the expected running times for insertion and deletion are still $O(\log n)$. Furthermore, the number of nodes copied during an insertion or deletion does not exceed the running time, so the expected number of nodes copied during each insertion and deletion is $O(\log n)$.

We have just shown how to implement a dictionary so that a sequence of insertions and deletions $o_1, \ldots o_n$ results in a sequence of dictionaries $D_0, \ldots, D_n$ where D_i is the result of operations $D_1, \ldots, D_i$. Each D_i can be searched in $O(\log m_i)$ time where m_i is the number of keys stored in D_i. We call such a dictionary a *persistent dictionary*. The following theorem states the performance of this dictionary

Theorem 5. *There exists a persistent dictionary data structure that supports* Insert, Delete *and* Search *in* $O(\log n)$ *time and requires* $O(n \log n)$ *storage for a sequence of* n *operations.*

[1] In the case of randomized search trees the space bound is in the expected sense.

[2] This is true if nodes only contain pointers to their left and right children. If they also contain pointers to their parents then the nodes adjacent to this path must also be copied.

Using this data structure for the next element search problem we obtain the following corollary.

Theorem 6. *There exists a data structure that takes as input a set* S *of* n *horizontal line segments and after* $O(n \log n)$ *preprocessing requiring* $O(n \log n)$ *storage, answers next element search queries on* S *in* $O(\log n)$ *time.*

3.3 Generalized Persistence

Path copying is a nice simple method for implementing persistence in binary search trees, but it is *ad hoc*. It's not obvious how to extend it for data structures other than binary search trees. Next we give a more general strategy for implementing persistence.

Suppose we have a pointer-based data structure which we model as a directed graph G. Each vertex of G has some constant number $c \geq 2$ of outgoing edges (representing pointers) and a label (representing data). The restriction to a constant number of outgoing edges is, in many cases, not really a restriction since we can simulate one node with many outgoing edges by many nodes that we link together as a linked list. Another possibility is to simulate a node with many outgoing edges as a binary tree whose leaves represent the edges.

The real restriction we place on G is that it have *bounded in-degree*. That is, no vertex of G has more than d edges leading into it, for some constant $d > 1$.

The operations we allow on G are $\textsc{Create-Node}(G)$ which creates a new vertex with no outgoing or incoming edges, $\textsc{Change-Label}(v, x)$ which changes the value of the data stored at vertex v to be x, and $\textsc{Change-Edge}(v, i, u)$ which changes the ith outgoing edge of node v so that it points to node u, where u may be **nil**. After each update operation, the global time t advances by 1 unit. This gives us a sequence of graphs $G_0, \ldots, G_t$ where $G_{t'}$ denotes the graph G at time t'.

For a persistent graph representation we would like an application to have access to $G_{t'}$ for any $0 \leq t' \leq t$. Of course, to access $G_{t'}$ an application needs to have a pointer to some node v that exists at time t'. How this is done depends on the application, but in most cases it is obvious (e.g., for search trees it is usually the root of the tree). The access operations we allow are $\textsc{Label}(v, t')$ which returns the label of the node v at time t' and $\textsc{Edge}(v, i, t')$ which returns the ith outgoing edge of v at time t'.

In order to implement this, we represent each vertex v of G as a table (array of structures). Each table contains $d + 1$ rows and has one column for a label (data), c columns for outgoing edges (pointers), and one column which indicates the time at which the corresponding row was filled in. In addition to this, v maintains an array inedges(v) of d pointers that keep track of the (at most d) other vertices of G that have edges leading to v.

Creating a vertex. A call to $\textsc{Create-Node}$ simply creates (and returns a pointer to) a new table in which all rows are empty and whose inedges values are all set to nil.

Changing an edge. In a call to $\textsc{Change-Edge}(v, i, u)$, two cases can occur:

Case 1: The table for node v has an empty row. In this case, we modify the table for node v by coping the last non-empty row into the first empty row and then modifying the entry for edge i in the new row so that it contains u. At the same time, we update the time column for the newly added row so that it contains the current global time t.

We then update inedges arrays for u and for the vertex w that was previously contained in the column for the ith outgoing edge of v. We add an entry to inedges(u) containing a pointer to the table for v (if no such entry existed previously), and we delete an entry that points to the table for v from inedges(w).

Case 2: The table for node v has no empty row. In this case, we make a new table for node v with a call to CREATE-NODE. We then copy the last row out of the old table into the first row of the new table. As before, we modify the entry for edge i in the first row of the new table so that it points to u and we modify the time so that it contains the current time t.

Next we modify the inedges arrays for every vertex w such that the edge (v, w) exists in G. For every edge (v, w) in the first row of the new table, we change the entry for v in inedges(w) (which pointed to the old table) so that it points to the new table.

At this point, there still exist up to d references to the old table for node v. To get rid of these, we recursively modify every node in inedges(v) so that it points to the new table for v. More precisely, if w contains the edge (w, v) as it's ith edge, then we call CHANGE-EDGE(w, i, v), where v is a pointer to the new table for v.

At this point we note that if there is some external reference to node v, then this reference should be updated to point to the new table for v. For example, if v is the root of a binary search tree then any accesses to the tree at time $t' \geq t$ will have to start at the new table for v (until the time the new table is copied).

Changing a label. The implementation of CHANGE-LABEL(v, x) is exactly the same as a call to CHANGE-EDGE(v, i, u) except that, instead of updating the column for edge i, we update the label column. Because of this, there is no need to update the inedges array for u.

Accessing label and edge data. We say that a table for node v in this implementation is *active* during the time interval $[t_1, t_2)$, where t_1 is the time at which the table was first created with a call to CREATE-NODE and t_2 is the time at which the table was first copied as part of Case 2 of the procedure for changing an edge or label. It follows that for any time t such that $t_1 \leq t' \leq t$, there is exactly one table for node v that is active.

To access an outgoing edge of v or a label of v at time $t' \leq t$, we use the table for v that was active at time t'. To determine the value of the edge or label, we look in the last row whose time value is less than or equal to t'. If the vertex v had label x at time t', then it is clear that the label in this row is x.

Similarly, if the vertex v had edge (v, w) as its ith outgoing edge at time t' then the entry for edge i in this row points to a table for w. We claim that this table for w is active at time t'. Clearly, the table must be active at some time $t'' \leq t'$, otherwise a pointer to this table could not have existed

at time t'. Therefore, the only way the table for w could not be active at time t' is if the table were copied (as part of Case 2 above) at some time $t'' \leq t'$. But in this case, the table for v would have been updated at time t'' to point to the new table for w.

Thus, if we start at the table for node v that is active at time t' and only follow edges as described above then we can reach only tables that were active at time t'. Any such table corresponds to a vertex w such that there is a path from v to w in G_t. In other words, this scheme is correct.

Analysis. How efficient is this scheme? To determine this, we use an accounting argument, also called a *credit scheme*. A credit can be thought of as a unit of currency that can pay for the cost of creating a new table. In fact, every newly created table will be paid for with 1 credit.

In addition to this, tables can accumulate credits which they can pay for later. The accumulation of credits at a table satisfies the following *credit invariant*: If the table is active, it has exactly the same number of credits as rows that have been filled in. Otherwise (the table is non-active) it has no credits.

We claim that the above credit scheme can be maintained if we insert two credits every time the user calls CREATE-VERTEX and one credit every time the user calls CHANGE-EDGE or CHANGE-LABEL. Note that we only insert a credit when a user calls one of these functions. As part of their implementation, they may call each other or themselves recursively, but we do not create new credits for these internal calls.

CREATE-VERTEX creates a new active table with exactly one row. If we insert two credit during a call to CREATE-VERTEX then we can use one credit to pay for the cost of creating the table and give one credit to the table so that the credit invariant is maintained.

CHANGE-EDGE has two cases. In Case 1, we add one row to an existing table. In this case we give our newly inserted credit to this table so that the credit scheme is maintained. Case 2 is more complicated. In this case, one (full) table becomes inactive, a new active table with one full row is created and up to d recursive calls are made.

Before the call to CHANGE-EDGE, the old (full) table stores $d+1$ credits and it becomes inactive, so we have $d+2$ credits at our disposal. We use one credit to pay for creating the new table and we give one more credit to this new table to satisfy the credit invariant. We are left with the cost of paying for d recursive calls, each of which requires one input credit. Luckily we have d credits left and we use each of these as input to one recursive call.

The accounting for CHANGE-LABEL is exactly the same as for CHANGE-EDGE.

We have just shown the following result.

Lemma 1. *n_1 calls to* CREATE-NODE, *and n_2 calls to* CHANGE-EDGE *and* CHANGE-LABEL *results in the creation of at most $2n_1 + n_2$ tables.*

Each table has size $O(d)$ and, excluding recursive calls, filling in a table requires $O(d^2)$ work (the extra d factor comes from updating inedges arrays at other nodes). Therefore, the amount of work done during a sequence of operations is bounded by $O(d^2)$ times the number of tables created. Each access operation (EDGE and LABEL) can be done in $O(d)$ time, or in $O(\log d)$ time if we use binary

search on the rows.

Theorem 7. *There exists a data structure that can complete any sequence of* n CREATE-NODE, CHANGE-EDGE *and* CHANGE-LABEL *operations and* m EDGE *and* LABEL *operations in* $O(nd^2 + m \log d)$ *time.*

3.4 Discussion and References

By now, the use of persistence is standard in data structure papers, so much so that most authors, after defininining a dynamic data structure simply cite Driscoll *et al* [1] to give a persistent version. Path copying was discovered independently by many authors, including Myers [3, 4], Krijnen and Meertens [2], Reps, Teitelbaum and Demers [5], and Swart [7].

Sarnak and Tarjan [6] first used a version of the generalized persistence scheme with red-black trees to give a data structure for the next-element search problem requiring $O(n)$ space. The generalized persistence mechanism of Section 3.3 is due to Driscoll *et al* [1].

In computational geometry, a dynamic data structure for a problem in d-dimensions can often be combined with persistence and plane-sweep to yields a static data structure in $d+1$ dimensions. Our next-element search data structure is an example of this, since a binary search tree can be thought of as a next-element search structure for 1-dimensional data. Unfortunately, this trick only works once.

Bibliography

[1] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989.

[2] T. Krijnen and L. G. L. T. Meertens. Making B-trees work for B. Technical Report 219/83, The Mathematical Center, Amsterdam, 1983.

[3] E. W. Myers. AVL dags. Technical Report 82-9, Department of Computer Science, University of Arizona, 1982.

[4] E. W. Myers. Efficient applicative data structures. In *Conference Record eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 66–75, 1984.

[5] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5:449–477, 1983.

[6] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, July 1986.

[7] G. Swart. Efficient algorithms for computing geometric intersections. Technical Report #85-01-02, Department of Computer Science, University of Washington, Seattle, 1985.