You're older than you've ever been and now you're even older
And now you're even older
And now you're even older
You're older than you've ever been and now you're even older
And now you're older still

— They Might be Giants, "Older", *Mink Car* (1999)

2 Static-to-Dynamic Transformations

A ***search problem*** is abstractly specified by a function of the form $Q\colon \mathscr{X} \times 2^{\mathscr{D}} \to \mathscr{A}$, where $\mathscr{D}$ is a (typically infinite) set of *data objects*, $\mathscr{X}$ is a (typically infinite) set of *query objects*, and $\mathscr{A}$ is a set of valid *answers*. A ***data structure*** for a search problem is a method for storing an arbitrary finite data set $D \subseteq \mathscr{D}$, so that given an arbitrary query object $x \in \mathscr{X}$, we can compute $Q(x, D)$ quickly. A *static* data structure only answers queries; a *dynamic* data structure also allows us to modify the data set by inserting or deleting individual items.

A search problem is ***decomposable*** if, for any pair of disjoint data sets D and D', the answer to a query over $D \cup D'$ can be computed in constant time from the answers to queries over the individual sets; that is,

$$Q(x, D \cup D') = Q(x, D) \diamond Q(x, D')$$

for some commutative and associative binary function $\diamond\colon \mathscr{A} \times \mathscr{A} \to \mathscr{A}$ that can be computed in $O(1)$ time. I'll use $\perp$ to denote the answer to any query $Q(x, \varnothing)$ over the empty set, so that $a \diamond \perp = \perp \diamond a = a$ for all $a \in \mathscr{A}$. Simple examples of decomposable search problems include the following.

- **Dictionary:** Data objects and query objects have the same arbitrary type; a query asks whether a query object x is a member of the data set D. Here, $\mathscr{A}$ is the set of booleans, $\diamond = \vee$, and $\perp = \textsc{False}$.

- **Range minimum queries:** Data objects are elements of some totally ordered universe $\mathscr{U}$, stored in an array $A[1 \,..\, n]$, and query objects are pairs (i, j) of indices where $i \leq j$; a query asks for the minimum element of the subarray $A[i \,..\, j]$. Here, $\mathscr{A} = \mathscr{U}$, $\diamond = \min$, and $\perp = \infty$.

- **Rectangle counting:** Data objects are points in the plane; query objects are rectangles; a query asks for the number of points in a given rectangle. Here, $\mathscr{A} = \mathbb{N}$, $\diamond = +$, and $\perp = 0$.

- **Nearest neighbor:** Data objects are points in some metric space; query objects are points in the same metric space; a query asks for the distance from a given query point to the nearest point. Here, $\mathscr{A}$ is the set of non-negative real numbers, $\diamond = \min$, and $\perp = \infty$.

- **Triangle emptiness:** Data objects are points in the plane; query objects are triangles; a query asks whether any data point lies in a given query triangle. Here, $\mathscr{A}$ is the set of booleans, $\diamond = \vee$, and $\perp = \textsc{False}$.

- **Interval stabbing:** Data objects are intervals on the real line; query objects are points on the real line; a query asks for the subset of data intervals that contain a given query point. Here, $\mathscr{A}$ is the set of all finite sets of real intervals, $\diamond = \cup$, and $\perp = \varnothing$.

2.1 Insertions Only (Bentley and Saxe* [3])

First, I'll describe a general transformation that adds the ability to insert new data objects into a static data structure, originally due to Jon Bentley and his PhD student James Saxe* [3]. Suppose we have a static data structure that can store any set of n items in space $S(n)$, after $P(n)$ preprocessing time, and answer an arbitrary query in time $Q(n)$. We will construct a new data structure with size $S'(n) = O(S(n))$, preprocessing time $P'(n) = O(P(n))$, query time $Q'(n) = O(\log n) \cdot Q(n)$, and *amortized* insertion time $I'(n) = O(\log n) \cdot P(n)/n$. In the next section, we will see how to achieve this insertion time even in the worst case.

Our data structure consists of $\ell = \lfloor \lg n \rfloor$ *levels* $L_0, L_1, \dots, L_{\ell-1}$. Each level L_i is either empty or a static data structure storing exactly 2^i items. Observe that for any value of n, there is a unique set of levels that must be non-empty, specified by the 1s in the binary representation of n. To answer a query, we perform queries in every non-empty level and combine the results. (This is where we require the queries to be decomposable.) For simplicity, the following pseudocode assumes that $\text{Query}(x, L_i)$ returns $\perp$ if L_i is empty; recall that $ans \diamond \perp = ans$.

```
NewQuery($x$):
  $ans \leftarrow \perp$
  for $i \leftarrow 0$ to $\ell - 1$
      $ans \leftarrow ans \diamond \text{Query}(x, L_i)$
  return $ans$
```

The total query time is clearly at most $\sum_{i=0}^{\ell-1} Q(2^i) < \ell \cdot Q(n) = O(\log n) \cdot Q(n)$, as claimed. Moreover, if $Q(n) > n^{\varepsilon}$ for any $\varepsilon > 0$, the sum is a geometric series dominated by its largest term, so the total query time is actually $O(Q(n))$.

The insertion algorithm follows the standard algorithm for incrementing a binary counter, where the presence or absence of each L_i plays the role of the ith least significant bit. We find the smallest empty level k; build a new data structure L_k containing the new item and all the items stored in $L_0, L_1, \dots L_{k-1}$; and finally discard all the levels smaller than L_k. See Figure 1 for an example.

```
Insert($x$):
  Find minimum $k$ such that $L_k = \varnothing$
  $L_k \leftarrow \{x\} \cup \bigcup_{i<k} L_i$      ⟨⟨takes $P(2^k)$ time⟩⟩
  for $i \leftarrow 0$ to $k - 1$
      destroy $L_i$
```

During the lifetime of the data structure, each item will take part in the construction of $\lg n$ different data structures. Thus, if we charge

$$I'(n) = \sum_{i=0}^{\lg n} \frac{P(2^i)}{2^i} = O(\log n) \frac{P(n)}{n}.$$

for each insertion, the total charge will pay for the cost of building all the static data structures. If $P(n) > n^{1+\varepsilon}$ for any $\varepsilon > 0$, the amortized insertion time is actually $O(P(n)/n)$.

2.2 Lazy Rebuilding (Overmars* and van Leeuwen [5, 4])

We can modify this general transformation to achieve the same space, preprocessing, and query time bounds, but now with *worst-case* insertion time $I'(n) = O(\log n) \cdot P(n)/n$, using a technique

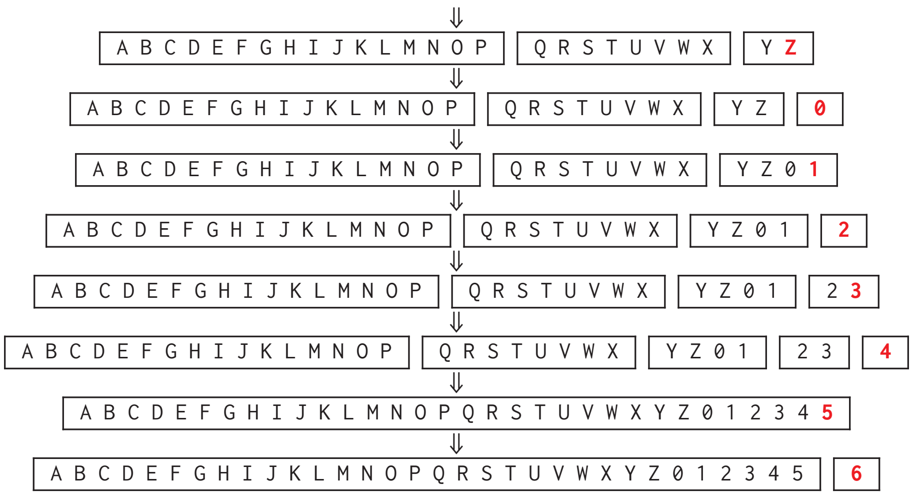
A B C D E F G H I J K L M N O P | Q R S T U V W X | Y Z
A B C D E F G H I J K L M N O P | Q R S T U V W X | Y Z | 0
A B C D E F G H I J K L M N O P | Q R S T U V W X | Y Z 0 1
A B C D E F G H I J K L M N O P | Q R S T U V W X | Y Z 0 1 | 2
A B C D E F G H I J K L M N O P | Q R S T U V W X | Y Z 0 1 | 2 3
A B C D E F G H I J K L M N O P | Q R S T U V W X | Y Z 0 1 | 2 3 | 4
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 | 6

Figure 1. The 27th through 33rd insertions into a Bentley/Saxe data structure

called *lazy rebuilding*. Obviously we cannot get fast updates in the worst case if we are ever required to build a large data structure all at once. The key idea is to stretch the construction time out over several insertions.

As in Bentley and Saxe's amortized structure, we maintain $\ell = \lfloor \lg n \rfloor$ *levels*. But now each level i consists of four static data structures: $Oldest_i$, $Older_i$, Old_i, and New_i. Each of the "old" data structures is either empty or contains exactly 2^i items; moreover, if $Oldest_i$ is empty then so is $Older_i$, and if $Older_i$ is empty then so is Old_i. The fourth data structure New_i is either empty or a *partially* built structure that will *eventually* contain 2^i items. Every item is stored in exactly one "old" data structure (at exactly one level) and *at most* one "new" data structure.

The query algorithm is almost unchanged; we separately query every *old* data structure and combine the results. (The *new* structures are used only to suppose insertions.)

```
NewQuery($x$):
    $ans \leftarrow \bot$
    for $i \leftarrow 0$ to $\ell - 1$
        $ans \leftarrow ans \diamond \text{Query}(x, Oldest_i)$
        $ans \leftarrow ans \diamond \text{Query}(x, Older_i)$
        $ans \leftarrow ans \diamond \text{Query}(x, Old_i)$
    return $ans$
```

(Again, this pseudocode assumes that $\text{Query}(x, \varnothing) = \bot$.) As before, the new query time is $O(\log n) \cdot Q(n)$, or $O(Q(n))$ if $Q(n) > n^\varepsilon$.

The insertion algorithm passes through the levels from largest to smallest. At each level i, if both $Oldest_{i-1}$ and $Older_{i-1}$ happen to be non-empty, we execute $P(2^i)/2^i$ steps of the algorithm to construct New_i from $Oldest_{i-1} \cup Older_{i-1}$. Once New_i is completely built, we move it to the oldest available slot on level i, delete $Oldest_{i-1}$ and $Older_{i-1}$, and rename Old_{i-1} to $Oldest_{i-1}$. Finally, we create a singleton structure at level 0 that contains the new item.

```
AGE($i$):
    if $Oldest_i = \varnothing$
        $Oldest_i \leftarrow New_i$
    else if $Older_i = \varnothing$
        $Older_i \leftarrow New_i$
    else
        $Old_i \leftarrow New_i$
    $New_i \leftarrow \varnothing$
```

```
LAZYINSERTION($x$):
    for $i \leftarrow \ell - 1$ down to 1
        if $Oldest_{i-1} \neq \varnothing$ and $Older_{i-1} \neq \varnothing$
            spend $P(2^i)/2^i$ time executing $New_i \leftarrow Oldest_{i-1} \cup Older_{i-1}$
            if $New_i$ is complete
                destroy $Oldest_{i-1}$ and $Older_{i-1}$
                $Oldest_{i-1} \leftarrow Old_{i-1}$
                $Old_{i-1} \leftarrow \varnothing$
                AGE($i$)
    $New_0 \leftarrow \{x\}$
    AGE(0)
```

Each insertion clearly takes $\sum_{i=0}^{\ell-1} O(P(2^i)/2^i) = O(\log n) \cdot P(n)/n$ time, or $O(P(n)/n)$ time if $P(n) > n^{1+\varepsilon}$ for any $\varepsilon > 0$. The only thing left to check is that the algorithm actually works! Specifically, how do we know that Old_i is empty whenever we call AGE(i)?

The key insight is that the modified insertion algorithm mirrors the standard algorithm to increment a non-standard binary counter, where every bit is either 2 or 3, except the most significant bit, which might be 1. It's not hard to prove by induction that this representation is unique; the correctness of the insertion algorithm follows immediately. Specifically, AGE(i) is called on the nth insertion—or in other words, the ith "bit" is incremented—if and only if $n = k \cdot 2^i - 2$ for some integer $k \geq 3$.

Figure 2 (on the next page) shows the modified insertion algorithm in action.

Exercise 1. *Suppose we increase the time in the third line of* LAZYINSERTION *from $P(2^i)/2^i$ to $P(2^i)/2^{i-1}$. Prove that the modified insertion algorithm is still correct, and that if we start with an empty data structure, every component Old_i is **always** empty.*

2.3 Deletions via (Lazy) Global Rebuilding: The Invertible Case

Under certain conditions, we can modify the logarithmic method to support deletions as well as insertions, by periodically rebuilding the entire data structure.

Perhaps the simplest case is when the binary operation $\diamond$ used to combine queries has an inverse $\bar{\diamond}$; for example, if $\diamond = +$ then $\bar{\diamond} = -$. In this case, we main two insertion-only data structures, a *main* structure M and a *ghost* structure G, with the invariant that every item in G also appears in M. To insert an item, we insert it into M. To delete an item, we *insert* it into G. Finally, to answer a query, we compute $Q(x, M) \,\bar{\diamond}\, Q(x, G)$.

The only problem with this approach is that the two component structures M and G might become significantly larger than the ideal structure storing $M \setminus G$, in which case our query and insertion times become inflated. To avoid this problem, we rebuild our entire data structure from scratch—building a new main structure containing $M \setminus G$ and a new empty ghost structure—whenever the size of G exceeds half the size of M. Rebuilding requires $O(P(n))$ time, where n is the number of items in the new structure. After a global rebuild, there must be at least $n/2$ deletions before the next global rebuild. Thus, the total amortized time for each deletion is $O(P(n)/n)$ plus the cost of insertion, which is $O(P(n) \log n/n)$.

> There is one minor technical point to consider here. Our earlier amortized analysis of insertions relied on the fact that large local rebuilds are always far apart. Global rebuilding destroys that assumption. In particular, suppose M has $2^k - 1$ elements and G has $2^{k-1} - 1$ elements, and we perform four operations: insert, delete, delete, insert.

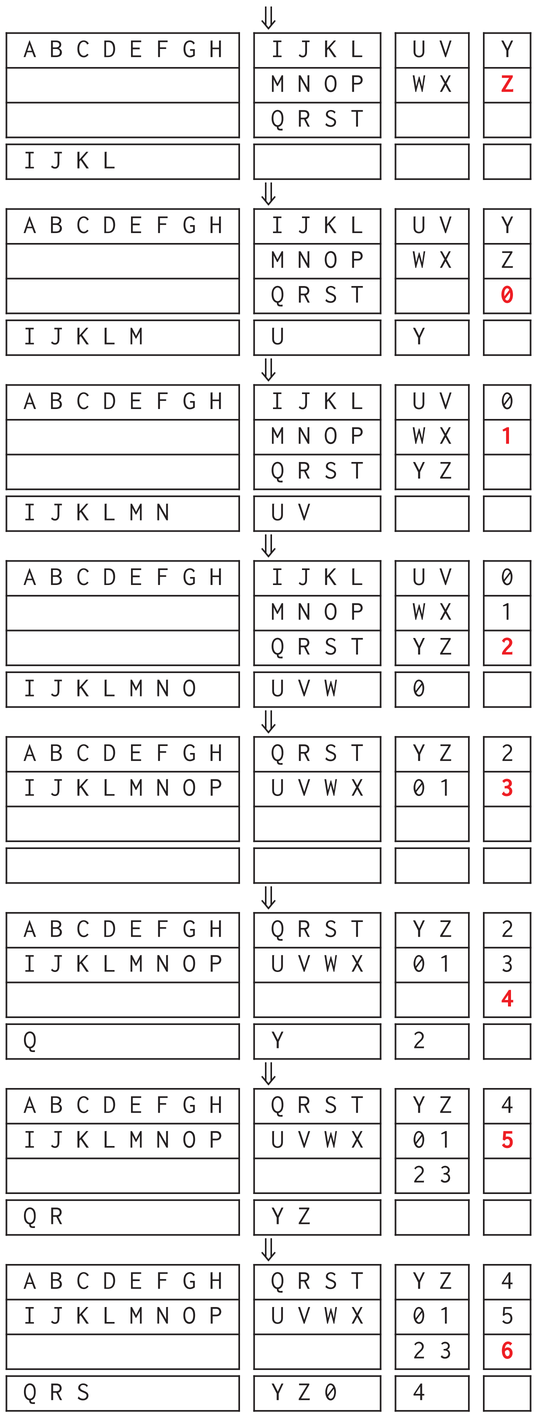
Figure 2. The 27th through 33rd insertions into a Overmars/van Leeuwen data structure

- The first insertion causes us to rebuild M completely.
- The first deletion causes us to rebuild G completely.
- The second deletion triggers a global rebuild. The new M contains $2^{k-1}-1$ items.
- Finally, the second insertion causes us to rebuild M completely.

Another way to state the problem is that a global rebuild can put us into a state where we don't have enough insertion credits to pay for a local rebuild. To solve this problem, we scale the amortized cost of deletions by a constant factor. When a global rebuild is triggered, a constant fraction of the accumulated charge pays for the global rebuild itself; the remainder pays for the first local rebuild at each level of the new main structure, since $\sum_{i=0}^{\lg n} P(2^i) = O(P(n))$.

We can achieve the same deletion time *in the worst case* by performing the global rebuild lazily. Now we maintain *three* structures: a static main structure M, an *insertion* structure I, and a *deletion* structure D. Most of the time, we insert new items into I, delete items by inserting them into D, and evaluate queries by computing $Q(x,M) \diamond Q(x,I) \,\overline{\diamond}\, Q(x,D)$.

However, when $|D| > (|M|+|I|)/2$, we freeze I and D and start building three new structures M', I', and D'. Initially, all three new structures are empty. Newly inserted items go into the new insertion structure I'; newly deleted items go into the new deletion structure D'. To answer a query, we compute $Q(x,M) \diamond Q(x,I) \diamond Q(x,I') \,\overline{\diamond}\, Q(x,D) \,\overline{\diamond}\, Q(x,D')$. After every deletion (that is, after every insertion into the new deletion structure D'), we spend $8 \cdot P(n)/n$ time building the new main structure M' from the set $(I \cup M) \setminus D$. After $n/8$ deletions, the new static structure is complete; we destroy the old structures I, M, D, and revert back to our normal state of affairs. The exact constant 8 is unimportant, it only needs to be large enough that the new main structure M' is complete before the start of the next global rebuild.

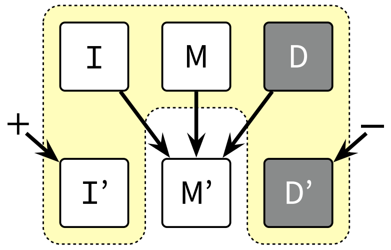

Figure 3. A high-level view of the deletion structure for invertible queries, during a lazy global rebuild.

With lazy global rebuilding, the *worst-case* time for a deletion is $O(P(n)\log n/n)$, exactly the same as insertion. Again, if $P(n) = \Omega(n^{1+\varepsilon})$, the deletion time is actually $O(P(n)/n)$.

2.4 Deletions for Non-Invertible Queries

To support both insertions and deletions when the function $\diamond$ has no inverse, we have to assume that the base structure already supports *weak deletions* in time $D(n)$. A weak deletion is *functionally* exactly the same as a regular deletion, but it doesn't have the same effect on the *cost* of future queries. Specifically, we require that the cost of a query after a weak deletion is no higher than the cost of a query before the weak deletion. Weak deletions are a fairly mild requirement; many data structures can be modified to support them with little effort. For example, to weakly delete an item x from a binary search tree begin used for simple membership queries (Is x in the set?), we simply mark all occurrences of x in the data structure. Future membership queries for x would find it, but would also find the mark(s) and thus return FALSE.

I should emphasize here that not all query problems support weak deletions. In particular, I am unaware of any data structure for range minimum queries or nearest-neighbor queries that supports weak deletions.

If we are satisfied with amortized time bounds, adding insertions to a weak-deletion data structure is straightforward. As before, we maintain a sequence of levels, where each level is either empty or a base structure. For purposes of insertion, each non-empty level L_i has *nominal* size 2^i, but it may actually store fewer elements. To delete an item, we first determine which level contains it, and then *weakly* delete it from that level. To make the first step possible, we also maintain an auxiliary dictionary (for example, a hash table) that stores a list of pointers to occurrences of each item in the main data structure. The insertion algorithm is essentially unchanged, except for the (small) additional cost of updating this dictionary. When the total number of undeleted items is less than half of the total *nominal* size of the non-empty levels, we rebuild the entire from scratch. The amortized cost of an insertion is $O(P(n)\log n/n)$, and the amortized cost of a deletion is $O(P(n)/n + D(n))$.

Once again, we can achieve the same time bounds in the worst case by spreading out both local and global rebuilding, but the details are more complex. I'll first describe the high-level architecture of the data structure and discuss how weak deletions are transformed into regular deletions, and then spell out the lower-level details for the insertion algorithm.

2.4.1 Transforming Weak Deletions into Real Deletions

For the moment, assume that we already have a data structure that supports insertions in $I(n)$ time and *weak* deletions in $D(n)$ time. A good example of such a data structure is the *weight-balanced B-tree* defined by Arge and Vitter [1].

Our global data structure has two major components; a main structure M and a *shadow copy* S. Queries are answered by querying the main structure M. Under normal circumstances, insertions and weak deletions are made directly in both structures. When more than half of the elements of S have been weakly deleted, we trigger a global rebuild. At that point, we freeze S and begin building two new clean structures M' and S'. The reason for the shadow structure is that we cannot copy from S while it is undergoing other updates.

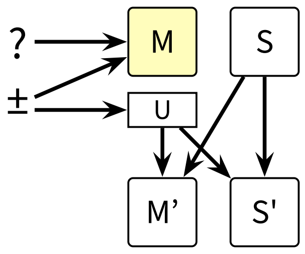

Figure 4. A high-level view of the deletion structure for non-invertible queries, during a lazy global rebuild

During a global rebuild, our data structure has four component structures M, S, M', and S' and an *update queue* U, illustrated above. Queries are evaluated by querying the main structure M as usual. Insertions and (weak) deletions are processed directly in M. However, rather than handling them directly in the shadow structure S (which is being copied) or the new structures

M' and S' (which are not completely constructed), all updates are inserted into the update queue U.

M' and S' are incrementally constructed in two phases. In the first phase, we build new data structures containing the elements of S. In the second phase, we execute the stream of insertions and deletions that have been stored in the update queue U, in both M' and S', in the order they were inserted into U. In each phase, we spend $O(I(n))$ steps on the construction for each insertion, and $O(P(n)/n + D(n))$ steps for each deletion, where the hidden constants are large enough to guarantee that each global rebuild is complete well before the next global rebuild is triggered. In particular, in the second rebuild phase, each time an update is inserted into U, we must process and remove at least two updates from U. When the update queue is empty, the new data structures M' and S' are complete, so we destroy the old structures M and S and revert to "normal" operation.

2.4.2 Adding Insertions to a Weak-Deletion-Only Structure

Now suppose our given data structure does not support insertions or deletions, but does support weak deletions in $D(n)$ time. A good example of such a data structure is the *kd-tree*, originally developed by Bentley [2].

To add support for insertions, we modify the lazy logarithmic method. As before, our main structure consists of $\lg n$ levels, but now each level consists of *eight* base structures New_i, Old_i, $Older_i$, $Oldest_i$, $SNew_i$, $SOld_i$, $SOlder_i$, $SOldest_i$, as well as an deletion queue D_i. We also maintain an auxiliary dictionary recording the level(s) containing each item in the overall structure. As the names suggest, each active structure $SFoo_i$ is a shadow copy of the corresponding active structure Foo_i. Queries are answered by examining the active old structures. New_i and its shadow copy $SNew_i$ are incrementally constructed from the shadows $SOlder_{i-1}$ and $SOldest_{i-1}$ and from the deletion queue D_i. Deletions are performed directly in the active old structures and in the shadows that are *not* involved in rebuilds, and are inserted into deletion queues at levels that are being rebuilt. At each insertion, if level i is being rebuilt, we spend $O(P(2^i)/2^i)$ time on that local rebuilding. Similarly, for each deletion, if the appropriate level i is being rebuilt, we spend $O(D(2^i))$ time on that local rebuilding. The constants in these time bounds are chosen so that each local rebuild finishes well before the next one begins.

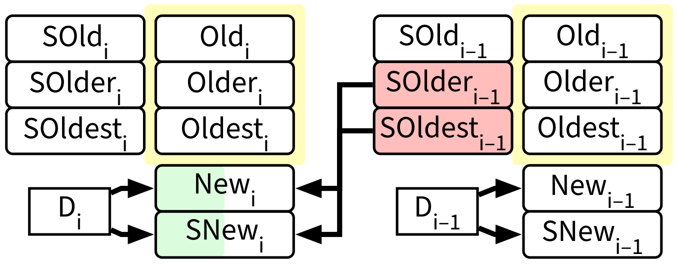

Figure 5. Two levels of our lazy dynamic data structure.

Here are the insertion and deletion algorithms in more detail:

```
AGE($i$):
    if $Oldest_i = \varnothing$
        $Oldest_i \leftarrow New_i$;  $SOldest_i \leftarrow SNew_i$
    else if $Older_i = \varnothing$
        $Older_i \leftarrow New_i$;  $SOlder_i \leftarrow SNew_i$
    else
        $Old_i \leftarrow New_i$;  $SOld_i \leftarrow SNew_i$
    $New_i \leftarrow \varnothing$;  $SNew_i \leftarrow \varnothing$
```

```
LAZYINSERT($x$):
    for $i \leftarrow \ell - 1$ down to 1
        if $Oldest_{i-1} \neq \varnothing$ and $Older_{i-1} \neq \varnothing$
            spend $O(P(2^i)/2^i)$ time building $New_i$ and $SNew_i$ from $SOldest_{i-1} \cup SOlder_{i-1}$
            if $New_i$ and $SNew_i$ are complete
                destroy $Oldest_{i-1}$, $SOldest_{i-1}$, $Older_{i-1}$, and $SOlder_{i-1}$
                $Oldest_{i-1} \leftarrow Old_{i-1}$;  $Old_{i-1} \leftarrow \varnothing$
                $SOldest_{i-1} \leftarrow SOld_{i-1}$;  $SOld_{i-1} \leftarrow \varnothing$
        else if $D_i \neq \varnothing$
            spend $O(P(2^i)/2^i)$ time processing deletions in $D_i$ from $New_i$ and $SNew_i$
            if $D_i = \varnothing$
                AGE($i$)
    $New_0 \leftarrow \{x\}$; $SNew_0 \leftarrow \{x\}$
    AGE(0)
```

```
DELETE($x$):
    find level $i$ containing $x$
    if $x \in Oldest_i$
        WEAKDELETE($x, Oldest_i$)
        if $Older_i \neq \varnothing$
            Add $x$ to $D_{i+1}$
            Spend $O(D(2^{i+1}))$ time building $New_{i+1}$ and $SNew_{i+1}$
        else
            WEAKDELETE($x, SOldest_i$)
    else if $x \in Older_i$
        WEAKDELETE($x, Older_i$)
        Add $x$ to $D_{i+1}$
        Spend $O(D(2^{i+1}))$ time building $New_{i+1}$ and $SNew_{i+1}$
    else if $x \in Old_i$
        WEAKDELETE($x, Old_i$);  WEAKDELETE($x, SOld_i$)
```

2.4.3 The Punch Line

Putting both of these constructions together, we obtain the following worst-case bounds. We are given a data structure that the original data structure requires space $S(n)$, can be built in time $P(n)$, answers decomposable search queries in time $Q(n)$, and supports weak deletions in time $D(n)$.

- The entire structure uses $O(S(n))$ space and can be built in $O(P(n))$ time.
- Queries can be answered in time $O(Q(n)\log n)$, or $O(Q(n))$ if $Q(n) > n^{\varepsilon}$ for any $\varepsilon > 0$.
- Each insertion takes time $O(P(n)\log n/n)$, or $O(P(n)/n)$ if $P(n) > n^{1+\varepsilon}$ for any $\varepsilon > 0$.

- Each deletion takes time $O(P(n)/n + D(n)\log n)$, or $O(P(n)/n + D(n))$ if $D(n) > n^{\varepsilon}$ for any $\varepsilon > 0$.

References

[1] Lars Arge and Jeffrey Scott Vitter. Optimal external memory interval management. *SIAM J. Comput.* 32(6):1488–1508, 2003.

[2] Jon L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18:509–517, 1975.

[3] Jon L. Bentley and James B. Saxe*. Decomposable searching problems I: Static-to-dynamic transformation. *J. Algorithms* 1(4):301–358, 1980.

[4] Mark H. Overmars*. *The Design of Dynamic Data Structures*. Lecture Notes Comput. Sci. 156. Springer-Verlag, 1983.

[5] Mark H. Overmars* and Jan van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Inform. Process. Lett.* 12(4):168–173, 1981.

*Starred authors were PhD students at the time that the work was published.