51 Dynamization

A data structure can be, depending on what operations are supported:

- *static* if all operations after building the structure do not alter the data,
- *semidynamic* if data insertion is possible as an operation,
- *fully dynamic* if deletion of inserted data is allowed along with insertion.

Static data structures are useful if we know the structure beforehand. In many cases, static data structures are simpler and faster than their dynamic alternatives.

A sorted array is a typical example of a static data structure to store an ordered set of n elements. Its supported operations are INDEX(i) which simply returns i-th smallest element in constant time, and FIND(x) which finds x and its index i in the array using binary search in time $\mathcal{O}(\log n)$.

However, if we wish to insert a new element to already existing sorted array, this operation will take $\Omega(n)$ – we must shift the elements to keep the sorted order. In order to have a fast insertion, we may decide to use a different dynamic data structure, such as a binary search tree. But then the operation INDEX slows down to logarithmic time.

In this chapter we will look at techniques of *dynamization* – transformation of a static data structure into a (semi)dynamic data structure. As we have seen with a sorted array, the simple and straight-forward attempts often lead to slow operations. Therefore, we want to dynamize data structures in such way that the operations stay reasonably fast.

51.1 Structure rebuilding

Consider a data structure with n elements such that modifying it may cause severe problems that are too hard to fix easily. In such case, we give up on fixing it and rebuild it completely anew.

If building such structure takes time $\mathcal{O}(f(n))$ and we perform the rebuild after $\Theta(n)$ modifying operations, we can amortize the cost of rebuild into the operations. This adds an amortized factor $\mathcal{O}(f(n)/n)$ to their time complexity, given that n does not change asymptotically between the rebuilds.

Examples:

- An array is a structure with limited capacity c. While it is dynamic (we can insert or remove elements at the end), we cannot insert new elements indefinitely. Once we run out of space, we build a new structure with capacity $2c$ and elements from the old structure. Since we insert at least $\Theta(n)$ elements to reach the limit from a freshly rebuilt structure, this amortizes to $\mathcal{O}(1)$ amortized time per an insertion, as we can rebuild an array in time $\mathcal{O}(n)$.

- Another example of such structure is an y-fast trie. It is parametrized by block size required to be $\Theta(\log n)$ for good time complexity. If we let n change enough such that $\log n$ changes asymptotically, the proven time complexity no longer holds. We can save this by rebuilding the trie once n changes by a constant factor (then $\log n$ changes by a constant additively). This happens no sooner than after $\Theta(n)$ insertions or deletions.

- Consider a data structure where instead of proper deletion of elements we just replace them with ''tombstones''. When we run a query, we ignore them. After enough deletions, most of the structure becomes filled with tombstones, leaving too little space for proper elements and slowing down the queries. Once again, the fix is simple – once at least $n/2$ of elements are tombstones, we rebuild the structure. To reach $n/2$ tombstones we need to delete $\Theta(n)$ elements.

Local rebuilding

In many cases, it is enough to rebuild just a part of the structure to fix local problems. Once again, if a structure part has size k, we want to have done at least $\Theta(k)$ operations since its last rebuild. This then allows the rebuild to amortize into other operations.

One of such structures is a binary search tree. We start with a perfectly balanced tree. As we insert or remove nodes, the tree structure degrades over time. With a particular choice of operations, we can force the tree to degenerate into a long vine, having linear depth.

To fix this problem, we define a parameter $1/2 < \alpha < 1$ as a *balance limit*. We use it to determine if a tree is balanced enough.

Definition: A node v is balanced, if for each its child c we have $s(c) \leq \alpha s(v)$. A tree T is balanced, if all its nodes are balanced.

Lemma: If a tree with n nodes is balanced, then its height is $\mathcal{O}(\log_{1/\alpha} n)$.

Proof: Choose an arbitrary path from the root to a leaf and track the node sizes. The root has size n. Each subsequent node has its size at most αn. Once we reach a leaf, its size is 1. Thus the path can contain at most $\log_{1/\alpha} n$ edges. □

Therefore, we want to keep the nodes balanced between any operations. If any node becomes unbalanced, we take the highest such node v and rebuild its subtree $T(v)$ into a perfectly balanced tree.

For α close to $1/2$ any balanced tree closely resembles a perfectly balanced tree, while with α close to 1 the tree can degenerate much more. This parameter therefore controls how often we cause local rebuilds and the tree height. The trees defined by this parameter are called $BB[\alpha]$ trees.

Rebuilding a subtree $T(v)$ takes $\mathcal{O}(s(v))$ time, but we can show that this happens infrequently enough. Both insertion and deletion change the amount of nodes by one. To unbalance a root of a perfectly balanced trees, and thus cause a rebuild, we need to add or remove at least $\Theta(n)$ vertices. We will show this more in detail for insertion.

Theorem: Amortized time complexity of the INSERT operation is $\mathcal{O}(\log n)$, with constant factor dependent on α.

Proof: We define a potential as a sum of ''badness'' of all tree nodes. Each node will contribute by the difference of sizes of its left and right child. To make sure that perfectly balanced subtrees do not contribute, we clamp difference of 1 to 0.

$$\Phi := \sum_v \varphi(v), \quad \text{where}$$
$$\varphi(v) := \begin{cases} |s(\ell(v)) - s(r(v))| & \text{if at least 2,} \\ 0 & \text{otherwise.} \end{cases}$$

When we add a new leaf, the size of all nodes on the path to the root increases by 1. The contribution to the potential is therefore at most 2.

We spend $\mathcal{O}(\log n)$ time on the operation. If all nodes stay balanced and thus no rebuild takes place, potential increases by $\mathcal{O}(\log n)$, resulting in amortized time $\mathcal{O}(\log n)$.

Otherwise, consider the highest unbalanced node v. Without loss of generality, the invariant was broken for its left child $l(v)$, thus $s(l(v)) > \alpha \cdot s(v)$. Therefore, the size of the other child is small: $s(r(v)) < (1-\alpha) \cdot s(v)$. The contribution of v is therefore $\varphi(v) > (2\alpha - 1) \cdot s(v)$.

After rebuilding $T(v)$, the subtree becomes perfectly balanced. Therefore for all nodes $u \in T(v)$ the contribution $\varphi(u)$ becomes zero. All other contributions stay the same. Thus, the potential decreases by at least $(2\alpha - 1) \cdot s(v) \in \Theta(s(v))$. By multiplying the potential by a suitable constant, the real cost $\Theta(s(v))$ of rebuild will be fully compensated by the potential decrease, yielding zero amortized cost. □

51.2 General semidynamization

Let us have a static data structure S. We do not need to know how the data structure is implemented internally. We would like to use S as a ''black box'' to build a (semi)dynamic data structure D which supports queries of S but also allows element insertion.

This is not always possible, the data structure needs to support a specific type of queries answering *decomposable search problems*.

Definition: A *search problem* is a mapping $f : U_Q \times 2^{U_X} \to U_R$ where U_Q is an universe of queries, U_X is an universe of elements and U_R is set of possible answers.

Definition: A search problem is *decomposable*, if there exists an operator $\sqcup : U_R \times U_R$ computable in time $\mathcal{O}(1)$[1] such that $\forall A, B \subseteq U_X$, $A \cap B = \emptyset$ and $\forall q \in U_Q$:

$$f(q, A \cup B) = f(q, A) \sqcup f(q, B).$$

Examples:

- Let $X \subseteq \mathcal{U}$. Is $q \in X$? This is a classic search problem where universes U_Q, U_X are both set $\mathcal{U}$ and possible replies are $U_R = \{\text{true}, \text{false}\}$. This search problem is decomposable, the operator $\sqcup$ is a simple binary OR.
- Let X be set of points on a plane. For a point q, what is the distance of q and the point $x \in X$ closest to q? This is a search problem where $U_Q = U_X = \mathbb{R}^2$ and $U_R = \mathbb{R}_0^+$. It is also decomposable – $\sqcup$ returns the minimum.
- Let X be set of points of a plane. Is q in convex hull of X? This search problem is not decomposable – it is enough to choose $X = \{a, b\}$ and $q \notin X$. If $A = \{a\}$ and $B = \{b\}$, both subqueries answer negatively. However, the query answer is equivalent to whether q is a convex combination of a and b.

For a decomposable search problem f we can thus split (decompose) any query into two queries on disjoint element subsets, compute results on them separately and then combine them in constant time to the final result. We can further chain the decomposition on each subset, allowing to decompose the query into an arbitrary amount of subsets.

We can therefore use multiple data structures S as blocks, and to answer a query we simply query all blocks, and then combine their answers using $\sqcup$. We will show this construction in detail.

[1] The constant time constraint is only needed for a good time complexity of D. If it is not met, the construction will still work correctly. Most practical composable problems meet this condition.

Construction

First, let us denote a few parameters for the static and dynamic data structure.

Notation: For a data structure S containing n elements and answering a decomposable search problem f and the resulting dynamic data structure D:

- $B_S(n)$ is time complexity of building S,
- $Q_S(n)$ is time complexity of query on S,
- $S_S(n)$ is the space complexity of S,

- $Q_D(n)$ is time complexity of query on D,
- $S_D(n)$ is the space complexity of D,
- $\bar{I}_D(n)$ is *amortized* time complexity of insertion to D.

We assume that $Q_S(n)$, $B_S(n)/n$, $S_S(n)/n$ are all non-decreasing functions.

We decompose the set X into blocks B_i such that $|B_i| \in \{0, 2^i\}$, $\bigcup_i B_i = X$ and $B_i \cap B_j = \emptyset$ for all $i \neq j$. Let $|X| = n$. Since $n = \sum_i n_i 2^i$ for $n_i \in \{0, 1\}$, its binary representation uniquely determines the block structure. Thus, the total number of blocks is at most $\log n$.

For each nonempty block B_i we build a static structure S of size 2^i. Since f is decomposable, a query on the structure will run queries on each block, and then combine them using $\sqcup$:

$$f(q, x) = f(q, B_0) \sqcup f(q, B_1) \sqcup \ldots \sqcup f(q, B_i).$$

Lemma: $Q_D(n) \in \mathcal{O}(Q_s(n) \cdot \log n)$.

Proof: Let $|X| = n$. Then the block structure is determined and $\sqcup$ takes constant time, $Q_D(n) = \sum_{i:B_i \neq \emptyset} \big(Q_S(2^i) + \mathcal{O}(1)\big)$. Since $Q_S(x) \leq Q_S(n)$ for all $x \leq n$, the inequality holds. □

Lemma: $S_D(n) \in \mathcal{O}(S_S(n))$.

Proof: For $|X| = n$ let $I = \{i \mid B_i \neq \emptyset\}$. Then for each $i \in I$ we store a static data structure S with 2^i elements contained in this block. Therefore, $Q_D(n) = \sum_{i \in I} Q_S(2^i)$. Since $S_S(n)$ is assumed to be non-decreasing,

$$\sum_{i \in I} Q_S(2^i) \leq \sum_{i \in I} \frac{Q_S(2^i)}{2^i} \cdot 2^i \leq \frac{S_S(n)}{n} \cdot \sum_{i=0}^{\log n} 2^i \leq \frac{S_S(n)}{n} \cdot n.$$

□

It might be advantageous to store the elements in each block separately so that we do not have to inspect the static structure and extract the elements from it, which may take additional time.

An insertion of x will act like an addition of 1 to a binary number. Let i be the smallest index such that $B_i = \emptyset$. We create a new block B_i with elements $B_0 \cup B_1 \cup \ldots \cup B_{i-1} \cup \{x\}$. This new block has $1 + \sum_{j=0}^{i-1} 2^j = 2^i$ elements, which is the required size for B_i. At last, we remove all blocks $B_0, \ldots, B_{i-1}$ and add B_i.

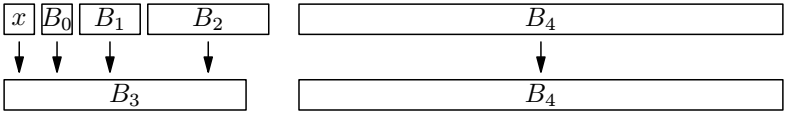

Figure 51.1: Insertion of x in the structure for $n = 23$, blocks $\{x\}$, B_0 to B_2 merge to a new block B_3, block B_4 is unchanged.

Lemma: $\bar{I}_D(n) \in \mathcal{O}(B_S(n)/n \cdot \log n)$.

Proof: Since the last creation of B_i there had to be least 2^i insertions. Amortized over one element this cost is $B_S(2^i)/2^i$. As this function is non-decreasing, we can lower bound it by $B_S(n)/n$. However, one element can participate in $\log n$ rebuilds during the structure life. Therefore, each element needs to store up cost $\log n \cdot B_S(n)/n$ to pay off all rebuilds. □

Theorem: Let S be a static data structure answering a decomposable search problem f. Then there exists a semidynamic data structure D answering f with parameters

- $Q_D(n) \in \mathcal{O}(Q_S(n) \cdot \log_n)$,
- $S_D(n) \in \mathcal{O}(S_S(n))$,
- $\bar{I}_D(n) \in \mathcal{O}(B_S(n)/n \cdot \log n)$ amortized.

In general, the bound for insertion is not tight. If $B_S(n) = \mathcal{O}(n^\varepsilon)$ for $\varepsilon > 1$, the logarithmic factor is dominated and $\bar{I}_D(n) \in \mathcal{O}(n^\varepsilon)$.

Example:

If we use a sorted array using binary search to search elements in a static set, we can use this technique to create a dynamic data structure for general sets. It will require $\Theta(n)$ space and the query will take $\Theta(\log^2 n)$ time as we need to binary search in each list. Since building requires sorting the array, building one requires $\Theta(n \log n)$ and insertion thus costs $\Theta(\log^2 n)$ amortized time.

We can speed up insertion time. Instead of building the list anew, we can merge the lists in $\Theta(n)$ time, therefore speeding up insertion to $\mathcal{O}(\log n)$ amortized.

Worst-case semidynamization

So far we have created a data structure that acts well in the long run, but one insertion can take long time. This may be unsuitable for applications where we require a low latency. In such cases, we would like that each insertion is fast even in the worst case.

Our construction can be deamortized for the price that the resulting semidynamic data structure will be more complicated. We do this by not constructing the block at once, but decomposing the construction such that on each operation we do does a small amount of work on it until eventually the whole block is constructed.

However, insertion is not the only operation, we can also ask queries even during the construction process. Thus we must keep the old structures until the construction finishes. As a consequence, more than one block of each size may exist at the same time.

For each rank i let B_i^0, B_i^1, B_i^2 be complete blocks participating in queries. No such block contains a duplicate element and union of all complete blocks contains the whole set X.

Next let B_i^* be a block in construction. Whenever two blocks B_i^a, B_i^b of same rank i meet, we will immediately start building B_{i+1}^* using elements from $B_i^a \cup B_i^b$.

This construction will require 2^{i+1} steps until B_{i+1}^* is finished, allocating enough time for each step. Once we finish B_{i+1}^*, we add it to the structure as one of the three full blocks and finally remove B_i^a and B_i^b.

We will show that, using this scheme, this amount of blocks is enough to book-keep the structure.

Lemma: At any point of the structure's life, for each rank i, there are at most three finished blocks and at most one block in construction.

Proof: For an empty structure, this certainly holds.

Consider a situation when two blocks B_i^0 and B_i^1 meet and B_i^1 has just been finalized. Then we start constructing B_{i+1}^*. 2^{i+1} steps later B_{i+1} is added and blocks B_i^0, B_i^1 are removed.

There may appear a new block B_i^2 earlier. However, this can only happen 2^i steps later. For the fourth block B_i^3 to appear, another 2^i steps are required. The earliest time is then $2 \cdot 2^i = 2^{i+1}$ steps later, during which B_{i+1}^* has been already finalized, leaving at most two blocks together and no block of rank $i+1$ in construction. □

An insertion is now done by simply creating new block B_0. Next, we additionally run one step of construction for each B_j^*. There may be up to $\log n$ blocks in construction.

Theorem: Let S be a static data structure answering a decomposable problem f. Then there exists semidynamic structure with parameters

- $Q_D(n) \in \mathcal{O}(Q_S(n) \cdot \log_n)$,
- $S_D(n) \in \mathcal{O}(S_S(n))$,
- $I_D(n) \in \mathcal{O}(B_S(n)/n \cdot \log n)$ worst-case.

Proof: Since there is now a constant amount of blocks of each rank, the query time and space complexities have increased by a constant compared to previous technique.

Each insertion builds a block of size 1 and then runs up to $\log n$ construction steps, each taking $B_S(2^i)/2^i$ time. Summing this together, we get the required upper bound. □

Full dynamization

For our definition of search problems, it is not easy to delete elements, as anytime we wished to delete an element we would need to take apart and split a structure into a few smaller ones. This could never be able to amortize to decent deletion time.

Instead of that, we will want the underlying static structure to have an ability to cross out elements. These elements will no longer participate in queries, but they will count towards the structure size and complexity.

Once we have ability to cross out elements, we can upgrade the semidynamic data structure to support deletion. We add a binary search tree or another set structure which maps each element to a block it lives in. For each element we keep a pointer on its instance in the BST. When we build a new block, we can update all its current elements in the tree in constant time (and insert the new one in logarithmic time).

Insertion time complexity then will always take at least logarithmic time and space requirements increase by the BST.

Deletion then finds an element in the BST, locates it in the corresponding block and crosses it out. We also keep the count of crossed out elements. If this count becomes a certain fraction of all elements, we rebuild the structure completely.

Before having to rebuild the whole structure, we cross-out at least $\Theta(n)$ elements, so the deletion time can be amortized and it will result in same time complexity as insertion.

There also exists an worst-case version of full dynamization which carefully manipulates with blocks, splitting and merging them as required. The details are somewhat complicated, so we will not look at them further.