1 Preliminaries

Generally, a data structure is a "black box", which contains some data and provides *operations* on the data. Some operations are *queries* on the current state of the data, some are *updates* which modify the data. The data are encapsulated within the structure, so that they can be accessed only through the operations.

A *static* data structure is once built and then it answers an unlimited number of queries, while the data stay constant. A *dynamic* structure allows updates.

We usually separate the *interface* of the structure (i.e., the set of operations supported and their semantics) from its *implementation* (i.e., the actual layout of data in memory and procedures handling the operations).

1.1 Examples of data structures

Queues and stacks

A *queue* is a sequence of items, which supports the following operations:

Enqueue(x)	Append a new item x at the head.
Dequeue	Remove the item at the tail and return it.
IsEmpty	Test if the queue is currently empty.

Here the *head* is the last item of the sequence and *tail* is the first one.

There are two obvious ways how to implement a queue:

- A *linked list* – for each item, we create a record in the memory, which contains the item's data and a pointer to the next item. Additionally, we keep pointers to the head and the tail. Obviously, all three operations run in constant time.
- An *array* – if we know an upper limit on the maximum number of items, we can store them in a cyclically indexed array and keep the index of the head and the tail. Again, the time complexity of all operations is constant.

A similar data structure is the *stack* — a sequence where both addition and removal of items happen at the same end.

Sets and dictionaries

Another typical interface of a data structure is the *set*. It contains a finite subset of some *universe* $\mathcal{U}$ (e.g., the set of all integers). The typical operations on a set are:

INSERT(x)	Add an element $x \in \mathcal{U}$ to the set. If it was already present, nothing happens.
DELETE(x)	Delete an element $x \in \mathcal{U}$ from the set. If it was not present, nothing happens.
FIND(x)	Check if $x \in \mathcal{U}$ is an element of the set. Also called MEMBER or LOOKUP.
BUILD($x_1, \ldots, x_n$)	Construct a new set containing the elements given. In some cases, this can be faster than inserting the elements one by one.

Here are the typical implementations of sets together with the corresponding complexities of operations (n always denotes the cardinality of the set):

	INSERT	DELETE	FIND	BUILD
Linked list	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Array	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Sorted array	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n \log n)$
Binary search tree	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n \log n)$
Hash table	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$

An INSERT to a linked list or to an array can be performed in constant time if we can avoid checking whether the element is already present in the set.

While sorted arrays are quite slow as dynamic structures, they are efficient statically: once built in $\mathcal{O}(\log n)$ time per element, they answer queries in $\mathcal{O}(\log n)$.

Hash tables achieve constant complexity of operations only on average. This will be formulated precisely and proven in the chapter on hashing. Also, while the other implementations can work with an arbitrary universe as long as two elements can be compared in constant time, hash tables require arithmetic operations.

An useful extension of sets are *dictionaries*. They store a set of distinct *keys*, each associated with a *value* (possibly coming from a different universe). That is, they behave as generalized arrays indexed by arbitrary keys. Most implementations of sets can be extended to dictionaries by keeping the value at the place where the key is stored.

Sometimes, we also consider *multisets,* in which an element can be present multiple times. A multiset can be represented by a dictionary where the value counts occurrences of the key in the set.

Ordered sets

Sets can be extended by order operations:

MIN, MAX	Return the minimum or maximum element of the set.
SUCC(x)	Return the successor of $x \in \mathcal{U}$ — the smallest element of the set which is greater than x (if it exists). The x itself need not be present in the set.
PRED(x)	Similarly, the predecessor of x is the largest element smaller than x.

All our implementations of sets can support order operations, but their time complexity varies:

	MIN/MAX	PRED/SUCC
Linked list	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Array	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Sorted array	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
Binary search tree	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Hash table	$\mathcal{O}(n)$	$\mathcal{O}(n)$

A sequence can be sorted by n calls to INSERT, one MIN, and n calls to SUCC. If the elements can be only compared, the standard lower bound for comparison-based sorting implies that at least one of INSERT and SUCC takes $\Omega(\log n)$ time.

Similarly, we can define ordered dictionaries and multisets.

Exercises

1. Show how to implement SUCC in a balanced binary search tree.

2. Consider enumeration of all keys in a binary search tree using MIN and n times SUCC. Prove that although a SUCC requires $\Theta(\log n)$ time in the worst case, the whole enumeration takes only $\Theta(n)$ time.

1.2 Model of computation

As we need to discern minor differences in time complexity of operations, we have to specify our model of computation carefully. All our algorithms will run on the *Random Access Machine* (RAM).

The memory of the RAM consists of *memory cells.* Each memory cell contains a single integer. The memory cell is identified by its *address,* which is again an integer. For example, we can store data in individual *variables* (memory cells with a fixed address), in *arrays* (sequences of identically formatted items stored in contiguous memory cells), or in *records* (blocks of memory containing a fixed number of items, each with a different, but fixed layout; this works as a `struct` in C). Arrays and records are referred to by their starting address; we call these addresses *pointers,* but formally speaking, they are still integers.

We usually assume that memory for arrays and records is obtained dynamically using a *memory allocator,* which can be asked to reserve a given amount of memory cells and free it again when it is no longer needed. This means that size of each array must be known when creating the array and it cannot be changed afterwards.

The machine performs the usual arithmetic and logical operators (essentially those available in the C language) on integers in constant time. We can also compare integers and perform both conditional and unconditional jumps. The constant time per operation is reasonable as long as the integers are not too large — for concreteness, let us assume that all values computed by our algorithms are polynomial in the size of the input and in the maximum absolute value given in the input.

Memory will be measured in memory cells *used* by the algorithm. A cell is used if it lies between the smallest and the largest address accessed by the program. Please keep in mind that when the program starts, all memory cells contain undefined values except for those which hold the program's input.

1.3 Amortized analysis

There is a recurring pattern in the study of data structures: operations which take a long time in the worst case, but typically much less. In many cases, we can prove that the worst-case time of a sequence of m such operations is much less than m times the worst-case time of a single operation. This leads to the concept of *amortized complexity,* but before we define it rigorously, let us see several examples of such phenomena.

Flexible arrays — the aggregation method

It often happens than we want to store data in an array (so that it can be accessed in arbitrary order), but we cannot predict how much data will arrive. Most programming languages offer some kind of flexible arrays (e.g., `std::vector` in C++) which can be resized at will. We will show how to implement a flexible array efficiently.

Suppose that we allocate an array of some *capacity* C, which will contain some n items. The number of items will be called the *size* of the array. Initially, the capacity will be some positive constant and the size will be zero. Items start arriving one by one, we will be appending them to the array and the size will gradually increase. Once we hit the capacity, we need to *reallocate* the array: we create a new array of some higher capacity C', copy all items to it, and delete the old array.

An ordinary append takes constant time, but a reallocation requires $\Theta(C')$ time. However, if we choose the new capacity wisely, reallocations will be infrequent. Let us assume that the initial capacity is 1 and we always double capacity on reallocations. Then the capacity after k reallocations will be exactly 2^k.

If we appended n items, all reallocations together take time $\Theta(2^1 + 2^2 + \ldots + 2^k)$ for k such that $2^{k-1} < n \leq 2^k$ (after the k-th reallocation the array is large enough, but it wasn't before). This implies that $n \leq 2^k < 2n$. Hence $2^1 + \ldots + 2^k = 2^{k+1} - 2 \in \Theta(n)$.

We can conclude that while a single append can take $\Theta(n)$ time, all n appends also take $\Theta(n)$ time, as if each append took constant time only. We will say that the amortized complexity of a single append is constant.

This type of analysis is sometimes called the *aggregation method* — instead of considering each operation separately, we aggregated them and found an upper bound on the total time.

Shrinkable arrays — the accounting method

What if we also want to remove elements from the flexible array? For example, we might want to use it to implement a stack. When an element is removed, we need not change the capacity, but that could lead to wasting lots of memory. (Imagine a case in which we have n items which we move between n stacks. This could consume $\Theta(n^2)$ cells of memory.)

We modify the flexible array, so that it will both *stretch* (to $C' = 2C$) and *shrink* (to $C' = \max(C/2, 1)$) as necessary. If the intial capacity is 1, all capacities will be powers of two, again. We will try to maintain an invariant that $C \in \Theta(n)$, so at most a constant fraction of memory will be wasted.

An obvious strategy would be to stretch the array if $n > C$ and shrink it when $n < C/2$. However, that would have a bad worst case: Suppose that we have $n = C = C_0$ for some even C_0. When we append an item, we cause the array to stretch, getting $n = C_0 + 1$, $C = 2C_0$. Now we remove two items, which causes a shrink after which we have $n = C_0 - 1$, $C = C_0$. Appending one more item returns the structure back to the initial state. Therefore, we have a sequence of 4 operations which makes the structure stretch and shrink, spending time $\Theta(C_0)$ for an arbitrarily high C_0. All hopes for constant amortized time per operation are therefore lost.

The problem is that stretching a ''full'' array leads to an ''almost empty'' array; similarly, shrinking an ''empty'' array gives us an ''almost full'' array. We need to design better rules such that an array after a stretch or shrink will be far from being empty or full.

We will stretch when $n > C$ and shrink when $n < C/4$. Intuitively, this should work: both stretching and shrinking leads to $n = C/2 \pm 1$. We are going to prove that this is indeed a good choice.

Let us consider an arbitrary sequence of m operations (appends and removals). We split this sequence to *blocks,* where a block ends when the array is reallocated or when the whole sequence of operations ends. For each block, we analyze the cost of the reallocation at its end:

- The first block starts with capacity 1, so its reallocation takes constant time.
- The last block does not end with a reallocation.
- All other blocks start with a reallocation, so at their beginning we have $n = C/2$. If it ends with a stretch, n must have increased to C during the block. If it ends with a shrink, n must have dropped to $C/4$. In both cases, the block must contain at least $C/4$ operations. Hence we can redistribute the $\Theta(C)$ cost of the reallocation to $\Theta(C)$ operations, each getting $\Theta(1)$ time.

We have proven that total time of all operations can be redistributed in a way such that each operation gets only $\Theta(1)$ units of time. Hence a sequence of m operations takes $\Theta(m)$ time, assuming that we started with an empty array.

This is a common technique, which is usually called the *accounting method.* It redistributes time between operations so that the total time remains the same, but the worst-case time decreases.

Note: We can interpret the rules for stretching and shrinking in terms of *density* — the ratio n/C. We require that the density of a non-empty structure lies in the interval $[1/4, 1]$. When it leaves this interval, we reallocate and the density becomes exactly $1/2$.

Before the next reallocation, the density must change by a constant, which requires $\Omega(n)$ operations to happen.

Binary counters — the coin method

Now, we turn our attention to an ℓ-bit binary counter. Initially, all its bits are zero. Then we keep incrementing the counter by 1. For $\ell = 4$, we get the sequence `0000`, `0001`, `0010`, `0011`, `0100`, and so on. Performing the increment is simple: we scan the number from the right, turning `1`s to `0`s, until we hit a `0`, which we change to a `1` and stop.

A single increment can take $\Theta(\ell)$ time when we go from `0111...1` to `1000...0`. Again, we will show that in amortized sense it is much faster: performing n increments takes only $\Theta(n)$ time.

We can prove it by simple aggregation: the rightmost bit changes every time, the one before it on every other increment, generally the i-th bit from the right changes once in 2^{i-1} increments. The total number of changes will therefore be:

$$\sum_{i=0}^{\ell-1} \left\lfloor \frac{n}{2^i} \right\rfloor \leq \sum_{i=0}^{\ell-1} \frac{n}{2^i} \leq n \cdot \sum_{i=0}^{\ell-1} \frac{1}{2^i} \leq n \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n.$$

However, there is an easier and ‘‘more economical’’ approach to analysis.

Imagine that we have some *coins* and each coin buys us a constant amount of computation time, enough to test and change one bit. We will maintain an invariant that for every `1`, we have one coin — we can imagine that the coin is ‘‘placed on the bit’’.

When somebody comes and requests an increment, we ask for 2 coins. This is enough to cover the single change of `0` to `1`: one coin will pay for the change itself, the other will be placed on the newly created `1`. Whenever we need to change a `1` to `0`, we will simply spend the coin placed on that `1`. After all operations, some coins will remain placed on the bits, but there is no harm in that.

Therefore, 2 coins per operation suffice to pay for all bit changes. This means that the total time spent on n operations is $\mathcal{O}(n)$.

This technique is often called the *coin method*. Generally, we are paid a certain number of ‘‘time coins’’ per operation. Some of them will be spent immediately, some saved for the future and spent later. Usually, the saved coins are associated with certain features of the data — in our example, with `1` bits. (This can be considered a special case of the accounting method, because we redistribute time from operations which save coins to those which spend the savings later.)

The potential method

In the above examples, the total complexity of a sequence of operations was much better than the sum of their worst-case complexities. The proof usually involved increasing the cost of ''easy'' operations to compensate for an occassional ''hard'' one. We are going to generalize this approach now.

Let us have a data structure supporting several types of operations. Let us consider an arbitrary sequence of operations $O_1, \ldots, O_m$ on the structure. Each operation is characterized by its *real cost* R_i — this is the time spent by the machine on performing the operation.[1] The cost generally depends on the current state of the data structure, but we often bound it by the worst-case cost for simplicity. We will often simplify calculations by clever choices of the unit of time, which is correct as long as the real time is linear in the chosen cost. For example, we can measure time in bit changes.

We further assign an *amortized cost* A_i to each operation. This cost expresses our opinion on how much does this operation contribute to the total execution time. We can choose it arbitrarily as long as the sum of all amortized costs is an upper bound on the sum of all real costs. Therefore, an outside observer who does not see the internals of the computation and observes only the total time cannot distinguish between the real and amortized costs.

Execution of the first i operations takes real cost $R_1 + \ldots + R_i$, but we claimed that it will take $A_1 + \ldots + A_i$. We can view the difference of these two costs as a balance of a ''bank account'' used to save time for the future. This balance is usually called the *potential* of the data structure. It is denoted by $\Phi_i = (\sum_{j=1}^{i} A_j) - (\sum_{j=1}^{i} R_j)$. Before we perform the first operation, we naturally have $\Phi_0 = 0$. At every moment, we have $\Phi_i \geq 0$ as we can never owe time.

The *potential difference* $\Delta\Phi_i = \Phi_i - \Phi_{i-1}$ is then equal to $A_i - R_i$. It tells us if the i-th operation saves time ($\Delta\Phi_i > 0$) or it spends time saved in the past ($\Delta\Phi_i < 0$).

Examples:

- In the binary counter, the potential corresponds to the number of coins saved, that is to the number of `1` bits. The amortized cost of each operation is 2. The real cost is $1 + o$ where o is the number of trailing `1`s. We calculate that the potential difference is $2 - (1 + o) = 1 - o$. This matches that one `1` was created and o of them disappeared.
- When we analyze the stretchable and shrinkable array, the potential counts the number of operations since the last reallocation. All operations are assigned amortized

[1] We prefer to speak of an abstract cost instead of execution time, since the same technique can be used to analyze consumption of different kinds of resources — for example space or disk I/O.

cost 2. Cost 1 is spent on the append/removal of the element, the other 1 is saved in the potential. When a reallocation comes, the potential drops to 0 and we can observe that its decrease offsets the cost of the reallocation.

In both cases, there is a simple correspondence between the potential and the state or the history of the data structure. This suggests an inverse approach: first we choose the potential according to the state of the structure, then we use it to calculate the amortized complexity. Indeed, the formula $A_i - R_i = \Delta\Phi_i$ can be re-written as $A_i = R_i + \Delta\Phi_i$. This says that *the amortized cost is the sum of the real cost and the potential difference.*

The sum of all amortized costs is then

$$\sum_{i=1}^{m} A_i = \sum_{i=1}^{m} (R_i + \Phi_i - \Phi_{i-1}) = \left(\sum_{i=1}^{m} R_i\right) + \Phi_m - \Phi_0.$$

The second sum *telescopes* — each Φ_i except for the first one and the last one is added once and subtracted once, so it disappears.

As long as $\Phi_m \geq \Phi_0$, the sum of real costs is majorized by the the sum of amortized costs, as we wanted.

Let us summarize how the potential method works:

- We choose an appropriate potential function Φ, which depends on the current state of the structure and possibly also on its history. There is no universal recipe for choosing it, but we usually want the potential to increase when we get close to a costly operation.
- We try to prove that $\Phi_0 \leq \Phi_m$ (we ‘‘do not owe time’’). This is easy if our potential counts occurrences of something in the structure — this makes it always non-negative and zero at the start. If the inequality does not hold, we must compensate for the ‘‘borrowed time’’ by assigning non-zero amortized cost to data structure initialization or cleanup.
- We establish the amortized costs of operations from their real costs and the potential differences: $A_i = R_i + \Phi_i - \Phi_{i-1}$.
- If we are unable to calculate the costs exactly, we use an upper bound. It also helps to choose units of cost to simplify multiplicative constants.

Amortized analysis is helpful if we use the data structure inside an algorithm and we are interested only in the total time spent by the algorithm. However, programs interacting with the environment in real time still require data structures efficient in the worst case — imagine that your program controls a space rocket.

We should also pay attention to the difference between amortized and average-case complexity. Averages can be computed over all possible inputs or over all possible random bits generated in a randomized algorithm, but they do not promise anything about a specific computation on a specific input. On the other hand, amortized complexity guarantees an upper bound on the total execution time, but it does not reveal anything about distribution of this time betwen individual operations.

1.4 Local rebuilding and weight-balanced trees

It is quite common that a long-running data structure degrades over time. One such case is when instead of a proper DELETE, we just replace the deleted items by ''tombstones'', which are ignored on queries. After a while, our memory becomes filled with tombstones, leaving too little space for live items. Like humans, data structures need to be able to forget.

There is an easy fix, at least if we do not care about worst-case performance. We just rebuild the whole structure from scratch occasionally. If a rebuild takes $\Theta(n)$ time and we do it once per $\Theta(n)$ operations, we can pay for the rebuild by increasing the amortized cost of each operation by a constant. This way, we can guarantee that the tombstones occupy at most a constant fraction of total memory.

Sometimes, the structure degrades only locally, so we can save the day by rebuilding it locally, without performing an expensive total rebuild. We will demonstrate this technique on the example of lazily balanced binary search trees.

Let us recall the definition of perfect balance first:

Notation: For a node v of a binary tree:

- $T(v)$ denotes the subtree rooted at v (all descendants of v, including v itself).
- $s(v)$ is the *size* of v, defined as the cardinality of $T(v)$.
- $\ell(v)$ and $r(v)$ is the left and right child of v. If a child is missing, we have $\ell(v) = \emptyset$ and/or $r(v) = \emptyset$. We define that $T(\emptyset) = \emptyset$ and $s(\emptyset) = 0$.

Definition: A tree is *perfectly balanced* if for each node v, we have $|s(\ell(v)) - s(r(v))| \leq 1$.

It means that the ratio of the two sizes is very close to $1 : 1$. A perfectly balanced tree can be constructed from a sorted sequence of items in $\mathcal{O}(n)$ time, but it is notoriously hard to maintain (see exercise 1).

Let us relax the requirement and accept any ratio between $1 : 2$ and $2 : 1$. We will see that this still guarantees logarithmic height and it will be much easier to maintain. The proper

definition will be however formulated in terms of parent/child sizes to avoid division by zero on empty subtrees.

Definition: A node v is *in balance,* if for each its child c, we have $s(c) \leq 2/3 \cdot s(v)$. A tree is *balanced,* if all its nodes are in balance.

Lemma: The height of a balanced tree with n nodes is $\mathcal{O}(\log n)$.

Proof: Let us follow an arbitrary path from the root to a leaf and keep track of node sizes. The root has size n. Each subsequent node has size at most $2/3$ of its parent's size. At the end, we reach a leaf, which has size 1. The path can therefore contain at most $\log_{2/3}(1/n) = \log_{3/2} n = \mathcal{O}(\log n)$ edges. □

When we Insert a new item (deletions will be left as an exercise), we have to check that the balance invariant was not broken. To make this easy, we let each node keep track of its size. Every insertion adds a leaf to the tree, so we have to increase sizes of all nodes between that leaf and the root by one.

Along the path, we check that all nodes are still in balance. If they are, we are done. Otherwise, we find the highest out-of-balance node v and rebuild the whole subtree rooted there to make it perfectly balanced. This takes $\Theta(s(v))$ time.

We are being lazy: As long as the imbalance is small, we keep smiling and do nothing. The balance invariant guarantees logaritmic height and thus also worst-case logarithmic complexity of queries. When the shape of the tree gets seriously out of hand, we make a resolute strike and clean up a potentially large subtree. This takes a lot of time, but we will show that it happens infrequently enough to keep the amortized cost small.

Theorem: Amortized time complexity of the Insert operation is $\mathcal{O}(\log n)$.

Proof: We will define a certain potential. We would like to quantify, how far is the current tree from the perfectly balanced tree. Insertions should increase the potential gradually and when we need to rebalance a subtree, the potential should be large enough to pay for the rebalancing.

The potential will be defined as a sum of per-node contributions. Each node will contribute the difference of sizes of its left and right child. However, we must adjust the definition slightly to make sure that perfectly balanced trees have zero potential: if the difference is exactly 1, we clamp the contribution 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 contributions of these vertices will therefore increase by at most 2 (they will usually change by exactly one, but because of the clamping, it can jump between 0 and 2).

If no rebuild took place, we spent $\mathcal{O}(\log n)$ time on the operation and we increased the potential by another $\mathcal{O}(\log n)$, so the total amortized cost is $\mathcal{O}(\log n)$.

Let us consider a rebuild at a node v now. The invariant was broken for v and its child c. Without loss of generality, c is the left child. We have $s(\ell(v)) > 2/3 \cdot s(v)$, so the other subtree must be small: $s(r(v)) < 1/3 \cdot s(v)$. The contribution of v is therefore $\varphi(v) > 1/3 \cdot s(v)$. After the rebuild, this contribution becomes zero. Contributions of all other nodes in the subtree also become zero, while all other contributions stay the same.

The whole potential therefore decreases by at least $1/3 \cdot s(v)$. The real cost of the rebuild is $\Theta(s(v))$, so after multiplying the potential by a suitable constant, the real cost will be offset by the change in potential, yielding zero amortized cost of the rebuild. □

Note: The balance invariant based on subtree sizes (also called weights) was proposed in 1972 by Edward Reingold. His BB[α] trees are more complex, maintained using rotations with worst-case $\mathcal{O}(\log n)$ complexity.

Exercises

1. Show that either Insert or Delete in a perfectly balanced tree must have worst-case time complexity $\Omega(n)$. This is easy for perfectly balanced trees on $n = 2^k - 1$ vertices, whose shape is uniquely determined.

2. Show how to make a tree perfectly balanced in $\Theta(n)$ time.

3.* Solve the previous exercise without making another copy of the data. It is possible to rebalance a tree in just $\mathcal{O}(1)$ extra space.

4. Add a Delete operation, implemented like in ordinary BSTs. Use the same potential to analyze it.

5. Implement Delete using tombstones and global rebuilding.

6. What would go wrong if we forgot to add the exception for difference 1 in the definition of $\varphi(v)$?