Lecture Notes on Data Structures

Martin Mareš

Department of Applied Mathematics
Faculty of Mathematics and Physics
Charles University, Prague, Czech Republic

This is a preliminary set of lecture notes for my lectures on advanced (Master's level) data structures. Please keep in mind that it is a work in progress, likely to contain errors and definitely lacking polish. If you find any mistake (hard-to-read parts also count), please report them to the author by e-mail to *mj@ucw.cz*. Thanks and have fun!

Table of contents

Sections and exercises marked with one or two asterisks contain advanced material. We suggest that the reader skips them first time.

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 $2^{k-1} < n \le 2^k$ (after the k-th reallocation the array is large enough, but it wasn't before). This implies that $n \le 2^k < 2n$. 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:

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

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 $\varphi(v)$?

2 Splay trees

In this chapter, we will present self-adjusting binary search trees called the Splay trees. They were discovered in 1983 by Daniel Sleator and Robert Tarjan. They are based on a very simple idea: whenever we access a node, we bring it to the root by a sequence of rotations. Surprisingly, this is enough to guarantee amortized $\mathcal{O}(\log n)$ cost of all operations. In cases when the items are accessed with non-uniform probabilities, the Splay trees will turn out to be even superior to ordinary balanced trees.

2.1 Splaying

Let us consider an arbitrary binary tree. We define the operation $\textsc{Splay}(x)$, which brings the node x to the root using rotations. This can be obviously done by repeatedly rotating the edge above x until x becomes the root, but this does not lead to good amortized complexity (see exercise 1).

The trick is to prefer double rotations as shown in figure 2.1. If x is a left child of a left child (or symetrically a right child of a right child), we perform the *zig-zig* step. If x is a right child of a left child (or vice versa), we do the *zig-zag* step. When x finally becomes a child of the root, we perform a single rotation — the *zig* step.

We can see the process of splaying in figure 2.2: First we perform a zig-zig, then again a zig-zig, and finally a single zig. You can see that splaying tends to reform long paths to branched trees. This gives us hope that randomly appearing degenerate subtrees will not stay for long.

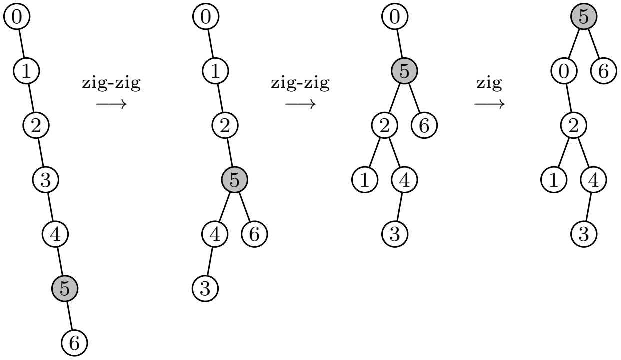

Figure 2.2: Steps of splaying the node 5

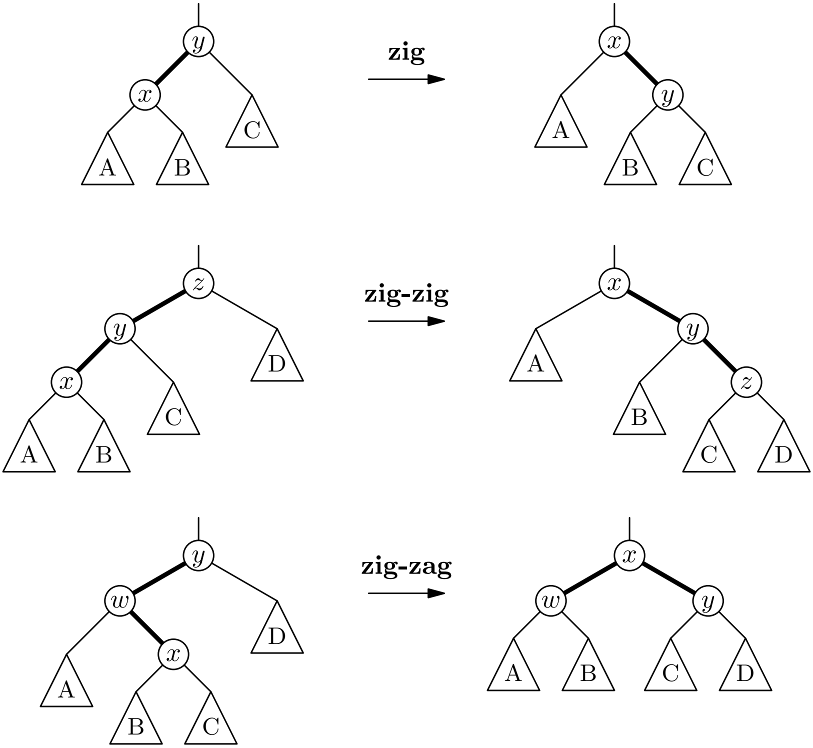

Figure 2.1: Three types of splay steps

Our amortized analysis will be based on a cleverly chosen potential. It might look magical, as if Sleator and Tarjan pulled it out of a magician's hat. But once we define the potential, the rest of the analysis becomes straightforward.

Notation:

- $T(v)$ denotes the subtree rooted at the node v.
- The *size* $s(v)$ is the cardinality of the subtree $T(v)$.
- The *rank* $r(v)$ is the binary logarithm of $s(v)$.
- The *potential* Φ of the splay tree is the sum of ranks of all its nodes.
- n is the total number of nodes in the tree.

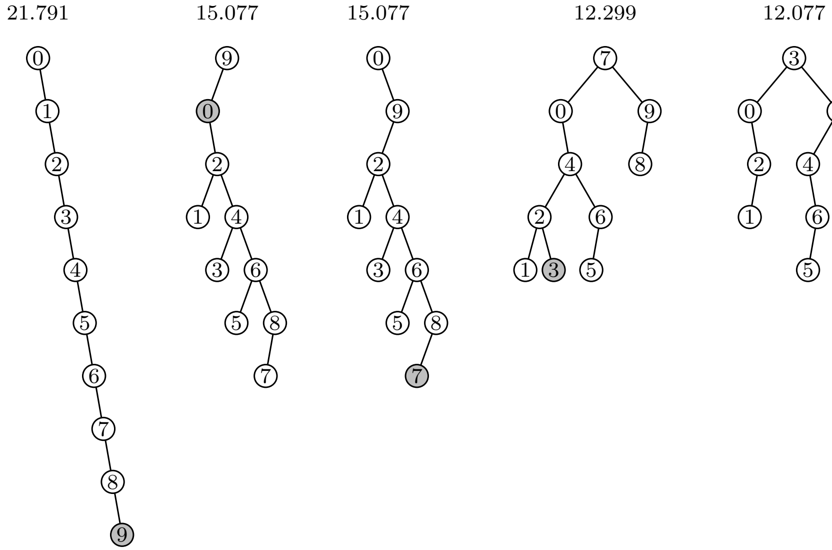

Figure 2.3: Evolution of potential during splays of nodes 9, 0, 7, 3

Observation: For any node v, we have $1 \le s(v) \le n$, $0 \le r(v) \le \log n$, $0 \le \Phi \le n \log n$.

Figure 2.3 suggests that higher potentials correspond to less balanced trees. By repeated splaying, the path gradually becomes a branching tree. The potential keeps decreasing; it decreases faster during expensive splays.

We are going to prove that this holds in general. We will quantify the cost of splaying by the number of rotations performed (so a zig-zig or zig-zag counts as two rotations). Real time complexity is obviously linear in this cost.

Theorem: The amortized cost of $\textsc{Splay}(x)$ is at most $3 \cdot (r'(x) - r(x)) + 1$, where $r(x)$ is the rank of the node x before the operation and $r'(x)$ the rank after it.

Proof: The amortized cost of the full $\textsc{Splay}$ is a sum of amortized costs of the individual steps. Let $r_1(x), \ldots, r_t(x)$ denote the rank of x after each step and $r_0(x)$ the rank before the first step.

We will use the following claim, which will be proven in the rest of this section:

> **Claim:** The amortized cost of the i-th step is at most $3r_i(x) - 3r_{i-1}(x)$, plus 1 if it is a zig step.

As there is at most one zig step, the total amortized cost becomes:

$$A \le \sum_{i=1}^{t} \bigl(3r_i(x) - 3r_{i-1}(x)\bigr) + 1.$$

This is a telescopic sum: each rank except r_0 and r_t participates once positively and once negatively. Therefore the right-hand side is equal to $3r_t(x) - 3r_0(x) + 1$ as claimed by the theorem. □

Corollary: As all ranks are logarithmic, the amortized cost of Splay is $\mathcal{O}(\log n)$.

When we perform a sequence of m splays, we can bound the real cost by a sum of the amortized costs. However, we must not forget to add the total drop of the potential over the whole sequence, which can be up to $\Theta(n \log n)$. We get the following theorem.

Theorem: A sequence of m Splays on an n-node binary tree runs in time $\mathcal{O}((n+m)\log n)$.

Bounding sums of logarithms

Analysis of individual steps will require bounding sums of logarithms, so we prepare a couple of tools first.

Lemma M (mean of logarithms): For any two positive real numbers α, β we have:

$$\log \frac{\alpha+\beta}{2} \ge \frac{\log\alpha + \log\beta}{2}.$$

Proof: The inequality holds not only for a logarithm, but for an arbitrary *concave* function f. These are the functions whose graphs lie above every line segment connecting two points on the graph. The natural logarithm is concave, because its second derivative is negative. The binary logarithm is a constant multiple of the natural logarithm, so it must be concave, too.

Let us consider a graph of a concave function f as in figure 2.4. We mark points $A = (\alpha, f(\alpha))$ and $B = (\beta, f(\beta))$. We find the midpoint S of the segment AB. Its coordinates are the means of coordinates of the endpoints A and B:

$$S = \left(\frac{\alpha+\beta}{2}, \frac{f(\alpha)+f(\beta)}{2}\right).$$

By concavity, the point S must lie below the graph, so in particular below the point

$$S' = \left(\frac{\alpha+\beta}{2}, f\left(\frac{\alpha+\beta}{2}\right)\right).$$

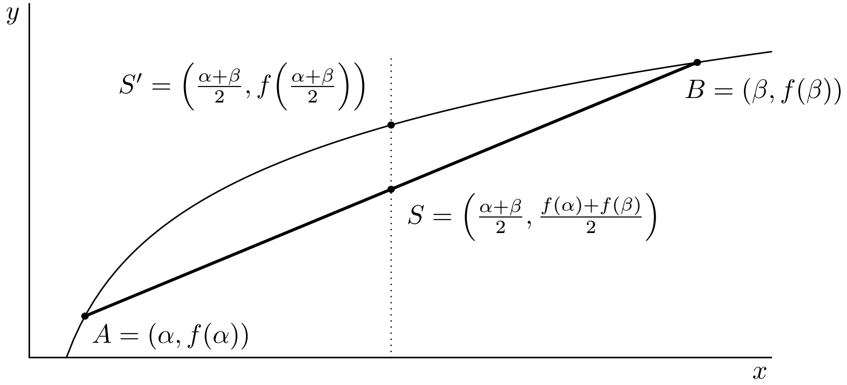
y
$S' = \left(\frac{\alpha+\beta}{2}, f\left(\frac{\alpha+\beta}{2}\right)\right)$
$B = (\beta, f(\beta))$
$S = \left(\frac{\alpha+\beta}{2}, \frac{f(\alpha)+f(\beta)}{2}\right)$
$A = (\alpha, f(\alpha))$
x

Figure 2.4: The mean inequality for a concave function *f*

Comparison of y-coordinates of the points S and S' yields the desired inequality. □

Note: The lemma also follows by taking logarithms of both sides of the Arithmetic Mean – Geometric Mean inequality $\sqrt{\alpha\beta} \leq (\alpha + \beta)/2$.

As $\log \frac{\alpha+\beta}{2} = \log(\alpha + \beta) - 1$, the lemma implies:

Corollary S (sum of logarithms): For positive α, β: $\log \alpha + \log \beta \leq 2 \log(\alpha + \beta) - 2$.

Now we calculate amortized costs of all three types of splay steps. In each case, x is the node being splayed, $r(x)$ is its rank before the step, and $r'(x)$ its rank after the step. We will use the same convention for s/s' and T/T'.

Zig-zag step

Let us follow figure 2.1 and consider, how the potential changes during the step. The only nodes whose ranks can change are w, x, and y. Thus the potential increases by $(r'(w) - r(w)) + (r'(x) - r(x)) + (r'(y) - r(y))$. The real cost of the operation is 2, so the amortized cost becomes:

$$A = 2 + r'(w) + r'(x) + r'(y) - r(w) - r(x) - r(y).$$

We want to prove that $A \leq 3r'(x) - 3r(x)$. We therefore need to bound all other ranks using $r(x)$ and $r'(x)$.

We invoke Corollary **S** on the sum $r'(w) + r'(y)$:

$$\begin{aligned} r'(w) + r'(y) &= \log s'(w) + \log s'(y) \\ &\le 2\log(s'(w) + s'(y)) - 2. \end{aligned}$$

The subtrees $T'(w)$ and $T'(y)$ are disjoint and they are contained in $T'(x)$, so we have $\log(s'(w) + s'(y)) \le \log s'(x) = r'(x)$. Thus:

$$r'(w) + r'(y) \le 2r'(x) - 2.$$

Substituting this to the inequality for A yields:

$$A \le 3r'(x) - r(w) - r(x) - r(y).$$

The other ranks can be bounded trivially:

$$\begin{aligned} r(w) &\ge r(x) \quad \text{because } T(w) \supseteq T(x), \\ r(y) &\ge r(x) \quad \text{because } T(y) \supseteq T(x). \end{aligned}$$

Zig-zig step

We will follow the same idea as for the zig-zag step. Again, the real cost is 2. Ranks can change only at nodes x, y, and z, so the amortized cost becomes:

$$A = 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z).$$

We want to get rid of all terms except $r(x)$ and $r'(x)$. To achieve this, we would like to invoke Corrolary **S** on a pair of subtrees, which are disjoint and their union contains almost all nodes. One such pair is $T(x)$ and $T'(z)$:

$$\begin{aligned} r(x) + r'(z) &= \log s(x) + \log s'(z) \\ &\le 2\log(s(x) + s'(z)) - 2 \\ &\le 2\log s'(x) - 2 = 2r'(x) - 2. \end{aligned}$$

This is equivalent to the inequality $r'(z) \le 2r'(x) - r(x) - 2$. Thus:

$$A \le 3r'(x) + r'(y) - 2r(x) - r(y) - r(z).$$

All other terms can be bounded trivially:

$$\begin{aligned} r(z) &= r'(x) &&\text{because } T(z) = T'(x), \\ r(y) &\geq r(x) &&\text{because } T(y) \supseteq T(x), \\ r'(y) &\leq r'(x) &&\text{because } T'(y) \subseteq T'(x). \end{aligned}$$

The claim $A \leq 3r'(x) - 3r(x)$ follows.

Zig step

The real cost is 1, ranks can change only at nodes x and y, so the amortized cost becomes:

$$A = 1 + r'(x) + r'(y) - r(x) - r(y).$$

By inclusion of subtrees, we have $r'(y) \leq r'(x)$ and $r(y) \geq r(x)$, hence:

$$A \leq 1 + 2r'(x) - 2r(x) \leq 1 + 3r'(x) - 3r(x).$$

The latter inequality holds, because by inclusion of subtrees, $r'(x) - r(x)$ is always non-negative.

Exercises

1. Prove that a naive splay strategy using only single rotations cannot have better amortized complexity than $\Omega(n)$. Consider what happens when splaying a path.

2. What is the potential of a path and of a perfectly balanced tree?

3.* *Top-down splaying:* In many operations with trees, we have to walk the path between the root and the current node twice: once down when we are looking up the node, once up when we are splaying it. Show how to combine splaying with the lookup, so that both can be performed during a single top-down pass.

4.* *Sequential traversal:* Prove that total cost of splaying all nodes in order of their keys is $\Theta(n)$.

5.* *Alternative analysis:* Alternatively, we can define the rank $r(v)$ as $\lfloor \log s(v) \rfloor$. Prove that this leads to a similar amortized bound. If $r(x) = r'(x)$ during a step, the difference pays for the step (if we assume that the real cost is scaled to 1). Otherwise, use that all ranks before the operation must have been equal.

2.2 Splaying in search trees

So far, we studied splaying in an otherwise static binary tree. Let us have a look on how to use the Splay tree as a binary search tree. It will turn out that we can use the ordinary Find, Insert, and Delete as on unbalanced search trees, provided that we always *splay the lowest visited node.* This alone will lower amortized cost of all operations to $\mathcal{O}(\log n)$.

Find

Suppose that Find found the given key at depth d. Going from the root to this node had real cost $\Theta(d)$ and it did not change the potential. The subsequent Splay will also have cost $\Theta(d)$ and it amortizes to $\mathcal{O}(\log n)$. We can therefore account the cost of the former part on the splay. This multiplies the real cost of splaying by a constant, so multiplying both the amortized cost and the potential by the same constant gives amortized complexity of the whole Find in $\mathcal{O}(\log n)$.

An unsuccessful Find also goes from the root down to some node, so if we splay the stopping node, the same argument applies. The Min operation is essentially an unsuccessful Find$(-\infty)$, so it is also $\mathcal{O}(\log n)$ amortized. The same goes with Max.

This is a useful general principle: *Whenever we walk from the root to a node and we splay that node, together it will have amortized cost $\mathcal{O}(\log n)$.*

Insert

An Insert tries to find the new key. If it succeeds, the key is already present in the set, so it stops, not forgetting to splay the discovered node. If it hits a null pointer, it connects a new leaf there with the new key. Again, we splay this node.

As usual, splaying the node will offset the cost of walking the tree, but there is one problem: while adding a leaf has constant real cost, it could increase the potential by a disastrous amount. We will prove that this is not the case.

Lemma: Adding a leaf to the tree increases potential by $\mathcal{O}(\log n)$.

Proof: Let us denote $v_1, \ldots, v_{t+1}$ the path from the root v_1 to the new leaf v_{t+1}. Again, we will use $r(v_i)$ for the rank before the operation and $r'(v_i)$ for the new rank. The rank $r'(v_{t+1})$ newly enters the potential; ranks $r(v_1), \ldots, r(v_t)$ increase. The potential difference is therefore:

$$\Delta\Phi = r'(v_{t+1}) + \sum_{i=1}^{t}(r'(v_i) - r(v_i)).$$

As leaves have size 1, the rank $r'(v_{t+1})$ is zero. As for the other ranks $r'(i)$: we know that $s'(v_i) = s(v_i) + 1$, but this is certainly at most $s(v_{i-1})$. So for $i > 1$, we have

$r'(v_i) \le r(v_{i-1})$. The sum therefore telescopes and we can reduce it to $\Delta\Phi \le r'(v_1) - r(v_t)$. This is $\mathcal{O}(\log n)$, because all ranks are at most $\log n$. □

The total amortized cost of Insert is therefore $\mathcal{O}(\log n)$.

Delete

The traditional implementation of Delete starts with finding the node. The actual deletion has three cases. Two of them are easy: we are removing either a leaf, or an internal node with one child. The case of a node with two children is solved by replacing the node (let's say) by the minimum of its right subtree, which reduces it to one of the easy cases.

Each deletion therefore consists of a walk from the root to a certain node (which is either the node with the given key, or its replacement, which is even deeper) and removal of this node. The cost of the walk can be offset by splaying the parent of the removed node; if there is no parent, the node had constant depth, so the cost is constant anyway.

Removing a node has constant real cost. The change in the potential is in favor of us: we are removing one node from the tree and decreasing the ranks of all its ancestors, so the potential difference is negative.

We can conclude that Delete runs in $\mathcal{O}(\log n)$ amortized time.

Splitting and joining

Alternatively, we can implement Insert and Delete using splits and joins of trees. It is easier to analyse and often also to implement. As we are considering multiple trees, we redefine the potential to sum ranks of all nodes of all trees together.

Insert starts with an unsuccessful Find. It stops in a node v, which is either a predecessor or a successor of the new key. We splay this node to the root. If it was a successor, we look for its left child ℓ. If there is none, we simply connect the new node as the left child. Otherwise we subdivide the edge $v\ell$ by the new node (we make the new node left child of v and we connect ℓ as the left child of the new node). If v was the predecessor, we do the same with the right child.

The cost of finding v is offset by the splay. Connecting the new node has constant real cost and it changes the potential by modifying at most three ranks. Regardless of how these ranks changed, the total difference is $\mathcal{O}(\log n)$.

Delete is similar:

1. We FIND the node and splay it to the root.
2. We remove the root, which splits the tree to a left subtree L and a right subtree R. If R is empty, we stop.
3. We find the minimum m of R and splay it. We note that m has no left child.
4. We connect the root of L as the left child of m.

Steps 1 and 3 consist of a walk followed by a splay, so they have $\mathcal{O}(\log n)$ amortized cost. Step 2 has constant real cost and it decreases the potential. Step 4 also has constant real cost and it increases the potential by increasing the rank of m, but all ranks are $\mathcal{O}(\log n)$.

This implementation of INSERT and DELETE therefore also runs in $\mathcal{O}(\log n)$ amortized time. We note that splits and joins are useful building blocks for other operations and they can be implemented easily and efficiently with splaying.

Conclusion

In the previous section, we proved a theorem bounding total time of a sequence of splays. We are going to prove a similar bound for a general sequence of set operations. Here we do not need the extra $\mathcal{O}(n \log n)$ term: if we start with an empty tree, the initial potential is zero and the final potential non-negative. The potential drop over the sequence is therefore non-positive.

Theorem: A sequence of m operations FIND, INSERT, and DELETE on an initially empty Splay tree takes $\mathcal{O}(m \log n)$ time, where n is the maximum number of nodes in the tree during the sequence.

Exercises

1. Show how to implement PRED and SUCC in $\mathcal{O}(\log n)$ amortized time.

2.3* Weighted analysis

Splay trees have surprisingly many interesting properties. Some of them can be proven quite easily by generalizing the analysis of *Splay* by putting different weights on different nodes.

We will assign a positive real *weight* $w(v)$ to each node v. The size of a node, which was the cardinality of its subtree, becomes the sum of weights. The definition of the rank and the potential will not change, but they will be based on the modified sizes. The original analysis therefore remains as a special case with all weights equal to 1.

Since weights are strictly positive, sizes will be also strictly positive. All ranks will therefore be well defined, but possibly negative. Our theory of amortized complexity does not break on negative potentials, but we must take care to include the total potential drop in our time bounds.

Surprisingly, the amortized cost of splaying a node x stays bounded by $3\cdot(r'(x)-r(x))+1$. The proof of this result relies on just two properties of sizes: First, all sizes must be positive, so all ranks are well defined and we can use the inequalities for logarithms. Second, we need the sizes to be monotonous: if $T(u)$ is a subset of $T(v)$, then $s(u) \le s(v)$, so $r(u) \le r(v)$. Both properties are satisfied in the weighted case, too.

However, the general $\mathcal{O}(\log n)$ bound on the amortized cost of splaying ceases to hold — ranks can be arbitrarily higher than $\log n$. We can replace it by $\mathcal{O}(r_{\rm max} - r_{\rm min})$, where $r_{\rm max} = \log(\sum_v w(v))$ is the maximum possible rank and $r_{\rm min} = \log \min_v w(v)$ the minimum possible rank.

We also have to be careful when analysing search tree operations:

- Find has the same complexity as Splay: $\mathcal{O}(r_{\rm max} - r_{\rm min})$.
- Insert based on adding a leaf and splaying it loses its beauty — when we add a leaf, the sum of rank differences no longer telescopes (and indeed, the amortized cost can be high).
- The second Insert performed by subdividing an edge below the root keeps working. Except for the splay, we are changing ranks of $\mathcal{O}(1)$ nodes, therefore the potential difference is $\mathcal{O}(r_{\rm max} - r_{\rm min})$ and so is the total amortized cost.
- The traditional version of Delete changes the structure by removing one leaf or one internal nodes with one child. Sizes of all other nodes do not increase, so their ranks do not increase either. Rank of the removed node could have been negative, so removing it can increase the potential by at most $-r_{\rm min}$. Total amortized cost of Delete is therefore also $\mathcal{O}(r_{\rm max} - r_{\rm min})$.
- Delete based on splits and joints is easier to analyse. Splitting a tree can only decrease sizes, so the rank does not increase. Joining roots of two trees changes only the rank of the new root, therefore the potential changes by $\mathcal{O}(r_{\rm max} - r_{\rm min})$. So the total amortized cost is again $\mathcal{O}(r_{\rm max} - r_{\rm min})$.

In the rest of this section, we will analyze Splay with different settings of weights. This will lead to stronger bounds for specific situations. In all cases, we will study sequences of accesses to a constant set of nodes — no insertions nor deletions will take place, only splays of the accessed nodes.

We will use W for the sum of weights of all nodes, so $r_{\rm max} = \log W$. This leads to the following bound on the cost of splaying:

Lemma W: The amortized cost of Splay(x) is $\mathcal{O}(\log(W/w(x)) + 1)$.

Proof: We are upper-bounding $3 \cdot (r'(x) - r(x)) + 1 = \mathcal{O}(\log(s'(x)/s(x)) + 1)$. As splaying makes x the root, we have $s'(x) = W$. As weights are non-negative, $s(x) \geq w(x)$. □

Warm-up: Uniform weights

Suppose that accesses to items are coming from a certain probability distribution. It is often useful to analyse the accesses by setting weights equal to access probabilities (assuming every item is accessed with a non-zero probability). This makes W always 1.

Let us experiment with the uniform distribution first. All n weights will be $1/n$. The size of each node therefore lies between $1/n$ and 1, ranks range from $-\log n$ to 0, so we have $-n \log n \leq \Phi \leq 0$.

By the above lemma, the amortized cost of a single Splay is $\mathcal{O}(\log(1/(1/n)) + 1) = \mathcal{O}(\log n)$. A sequence of m splays therefore takes $\mathcal{O}(m \log n)$ plus the total potential drop over the sequence, which is at most $n \log n$.

We got the same result as for unit weights. In fact, *scaling the weights by a constant factor never changes the result.* If we multiply all weights by a factor $c > 0$, all sizes are multiplied by the same factor, so all ranks are increased by $\log c$. Therefore, rank differences do not change. Potential is increased by $n \log c$, so potential differences do not change either.

Static optimality

Now consider that the accesses are coming from a general distribution, which can be very far from uniform. The actual cost of an access is a random variable, whose expected value can differ from $\Theta(\log n)$. If we know the distribution in advance, we can find an optimum static tree T, which minimizes the expected cost $\mathbb{E}_x[c_T(x)]$. Here $c_T(x)$ denotes the cost of accessing the item x in the tree T measured as the number of items on the path from the root to x.

The optimal static tree can be found in time $\mathcal{O}(n^2)$ by a simple algorithm based on dynamic programming. We will prove that even though the Splay trees possess no knowledge of the distribution, they perform at most constant-times worse than the optimal tree.

Let $p(x)$ be the probability of accessing the item x. This will be also the weight $w(x)$, so the total weight W is exactly 1. The amortized cost of the access to x will be $\mathcal{O}(\log(1/w(x)) + 1) = \mathcal{O}(\log(1/p(x)) + 1)$. Therefore the expected amortized cost is:

$$\mathbb{E}\left[\mathcal{O}\left(\log\frac{1}{p(x)} + 1\right)\right] = \mathcal{O}\left(1 + \sum_x p(x) \cdot \log\frac{1}{p(x)}\right).$$

The sum at the right-hand side is the Shannon entropy of the distribution. Since entropy is a lower bound on the expected code length of a prefix code, it is also a lower bound on the expected node depth in a binary search tree. Therefore, the expected amortized cost of splaying is $\mathcal{O}$ of the expected access cost in the optimal static tree. This holds even for expected real cost of splaying if the access sequence is long enough to average out the total potential drop.

There is however a more direct argument, which does not need to refer to information theory. We will state it for a concrete access sequence with given *access frequencies.*

Theorem (Static optimality of Splay trees): Let $x_1, \ldots, x_m$ be an access sequence on a set X, where every item $x \in X$ is accessed $f(x) > 0$ times. Let T be an arbitrary static tree on X. Then the total cost of accesses in a Splay tree on X is $\mathcal{O}$ of the total cost of accesses in T.

Proof: The tree T will process all accesses in time $\mathcal{O}(\sum_x f(x) \cdot c_T(x))$, where $c_T(x)$ is the (non-zero) cost of accessing the item in the tree T. For analysis of the Splay tree, we set the weight of each item x based on this cost: $w(x) = 3^{-c_T(x)}$. The sum of all weights is $W < 1$, because in an infinite regular binary tree, it would be exactly 1. As for the ranks, we have $r(x) = \log s(x) \le \log W < 0$, so all ranks are negative.

By Lemma **W**, the amortized cost of accessing x in the Splay tree is $\mathcal{O}(\log(W/w(x))+1) = \mathcal{O}(\log 3^{c_T(x)}+1) = \mathcal{O}(c_T(x)+1) = \mathcal{O}(c_T(x))$. The sum of amortized costs is therefore $\mathcal{O}$ of the cost in T, but we must not forget to add the total potential drop $\Phi_0 - \Phi_m \le -\Phi_m$. However, the negative sum of all ranks is $\mathcal{O}(\sum_x c_T(x))$, which is again $\mathcal{O}$ of the access cost in T, because all frequencies are non-zero. □

It is still an open question whether Splay trees are *dynamically optimal* — that is, at most $\mathcal{O}(1)$ times slower than the best possible dynamic tree which knows the access sequence beforehand and adjusts itself optimally.

Static finger bound

There is also a number of results of the type "if accesses are *local* (in some sense), then they are faster". A simple result of this kind is the static finger bound. Without loss of generality we can renumber the items to $1, \ldots, n$. We select one item f as a *finger.* Locality of access to an item i can be measured by its distance from the finger $|i - f|$. Then we have the following bound:

Theorem (Static finger bound): Consider a Splay tree on the set of items $X = \{1, \ldots, n\}$ and a finger $f \in X$. Accesses to items $x_1, \ldots, x_m \in X$ are processed in time $\mathcal{O}(n \log n + m + \sum_i \log(|x_i - f| + 1))$.

Proof: We set items weights to $w(i) = 1/(|i - f| + 1)^2$. As the sum of all weights contains each rational number $1/k^2$ at most twice, the sum of all weights W is at most

$2 \cdot \sum_{k\geq 1} k^{-2} = \pi^2/3$, so it is $\mathcal{O}(1)$. By Lemma **W**, the amortized cost of an access to x_i is $\mathcal{O}(\log(W/w(x_i)) + 1) = \mathcal{O}(\log(1 + |x_i - f|)^2 + 1) = \mathcal{O}(\log(1 + |x_i - f|) + 1)$.

We sum this over all accesses, but we need to add the total potential drop over the sequence. Since all sizes are between $1/n^2$ and W, ranks are between $-2\log n$ and a constant, so the potential lies between $-2n\log n$ and $\mathcal{O}(n)$. Therefore the potential drop $\Phi_0 - \Phi_m$ is $\mathcal{O}(n + n\log n) = \mathcal{O}(n\log n)$. □

Working set bound

A similar technique can be used to obtain a more interesting theorem: the cost of access to an item x can be bounded using the number of distinct items accessed since the previous access to x — the so-called *working set.* If there is no previous access to x, the working set contains all items accessed so far. The amortized cost of the access will be approximately the logarithm of the size of the working set.

We are going to compare the Splay tree with the *LRU (least-recently-used) list.* Initially, the list contains all items in order of their first access, starting at index 0. Items which are never accessed are placed in an arbitrary order at the end of the list. Whenever we access an item x_i, we move it from its position $p(x_i)$ to the start of the list. The items we skipped over are equal to the working set of x_i, so the size of the working set is $p(x_i)$.

Theorem (Working set bound): Consider a Splay tree on a set of n items. Accesses to items $x_1, \ldots, x_m$ are processed in time $\mathcal{O}(n\log n + m + \sum_i \log(1 + z_i))$, where z_i is the size of the working set for x_i.

Proof: We set item weights according to their positions in the current LRU list: $w(x) = 1/(p(x)+1)^2$. As each position occurs exactly once, the total weight W is again $\mathcal{O}(1)$. All sizes therefore range between $1/n^2$ and $\mathcal{O}(1)$, ranks range between $-2\log n$ and a constant, and the potential is between $-2n\log n$ and $\mathcal{O}(n)$. However, we must not forget that any change of the LRU list causes re-weighting and thus a change of the potential.

Let us analyze a cost of the access to x_i. As usual, we invoke Lemma **W** to bound the amortized cost of Splay by $\mathcal{O}(\log(W/w(x_i))+1) = \mathcal{O}(\log(p(x_i)+1)^2+1) = \mathcal{O}(\log(p(x_i)+1)+1) = \mathcal{O}(\log(1+z_i)+1)$.

Then we have to move x_i to the head of the LRU list (at least in our mind). How is the potential affected? The position of x_i decreases from $p(x_i)$ to 0, so the weight $w(x_i)$ increases from $1/(p(x_i) + 1)^2$ to 1. The positions of the skipped items increase by 1, so their weights decrease. The positions and weights of all other items stay the same. Therefore the size of x_i increases from at least $1/(p(x_i) + 1)^2$ to at most W, so its rank increases by $\mathcal{O}(\log W - \log 1/(p(x_i)+1)^2) = \mathcal{O}(1+\log(z_i+1))$. Sizes of all other items do not increase, because their subtrees do not contain x_i, which is currently the root. Total increase in the potential is therefore bounded by the amortized cost of the Splay itself.

We sum the amortized cost over all accesses, together with the total potential drop. As the absolute value of the potential is always $\mathcal{O}(n \log n)$, so is the potential drop. □

The working set property makes Splay tree a good candidate for various kinds of caches.

3 (a,b)-trees

In this chapter, we will study an extension of binary search trees. It will store multiple keys per node, so the nodes will have more than two children. The structure of such trees will be more complex, but we will gain much more straightforward balancing operations.

3.1 Definition and operations

Definition: A *multi-way search tree* is a rooted tree with specified order of children in every node. Nodes are divided to internal and external.

Each *internal node* contains one or more distinct keys, stored in increasing order. A node with keys $x_1 < \ldots < x_k$ has $k+1$ children $s_0, \ldots, s_k$. The keys in the node separate keys in the corresponding subtrees. More formally, if we extend the keys by sentinels $x_0 = -\infty$ and $x_{k+1} = +\infty$, every key y in the subtree $T(s_i)$ satisfies $x_i < y < x_{i+1}$.

External nodes carry no data and they have no children. These are the leaves of the tree. In a program, we can represent them by null pointers.

Observation: Searching in multi-way trees is similar to binary trees. We start at the root. In each node, we compare the desired key with all keys of the node. Then we either finish or continue in a uniquely determined subtree. When we reach an external node, we conclude that the requested key is not present.

The universe is split to open intervals by the keys of the tree. There is a 1-to-1 correspondence between these intervals and external nodes of the tree. This means that searches for all keys from the same interval end in the same external node.

As in binary search trees, multiway-trees can become degenerate. We therefore need to add further invariants to keep the trees balanced.

Definition: An *(a,b)-tree* for parameters $a \geq 2$ and $b \geq 2a-1$ is a multi-way search tree, which satisfies:

1. The root has between 2 and b children (unless it is a leaf, which happens when the set of keys is empty). Every other internal node has between a and b children.
2. All external nodes have the same depth.

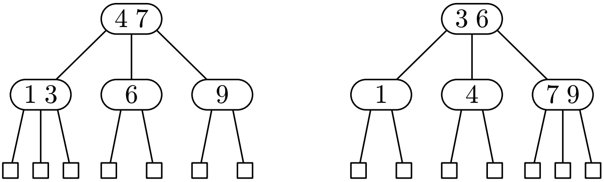

Figure 3.1: Two $(2,3)$-trees for the same set of keys

The requirements on a and b may seem mysterious, but they will become clear when we define operations. The minimum working type of (a,b)-trees are $(2,3)$-trees — see figure 3.1 for examples. Internal nodes are drawn round, external nodes are square.

Unlike in other texts, we will *not* assume that a and b are constants, which can be hidden in the $\mathcal{O}$. This will help us to examine how the performance of the structure depends on these parameters.

We will start with bounding the height of (a,b)-trees. Height will be measured in edges, a tree of height h has the root at depth 0 and external nodes at depth h. We will call the set of nodes at depth i the i-th *level*.

Lemma: The height of an (a,b)-tree with n keys lies between $\log_b(n+1)$ and $1+\log_a((n+1)/2)$.

Proof: Let us start with the upper bound. We will calculate the minimum number of keys in a tree of height h. The minimum will be attained by a tree where each node contains the minimum possible number of keys, so it has the minimum possible number of children. Level 0 contains only the root with 1 key. Level h contains only external nodes. The i-th level inbetween contains $2a^{i-1}$ nodes with $a-1$ keys each. Summing over levels, we get:

$$1+\sum_{i=1}^{h-1} 2a^{i-1}(a-1) = 1+2(a-1)\sum_{i=0}^{h-2} a^i = 1+2(a-1)\frac{(a^{h-1}-1)}{(a-1)} = 2a^{h-1}-1.$$

So $n \geq 2a^{h-1}-1$ in any tree of height h. Solving this for h yields $h \leq 1+\log_a((n+1)/2)$.

For the lower bound, we consider the maximum number of keys for height h. All nodes will contain the highest possible number of keys. Thus we will have b^i nodes at level i, each with $b-1$ keys. In total, the number of keys will reach

$$\sum_{i=0}^{h-1} b^i(b-1) = (b-1)\sum_{i=0}^{h-1} b^i = (b-1)\cdot\frac{b^h-1}{b-1} = b^h - 1.$$

Therefore in each tree, we have $n \le b^h - 1$, so $h \ge \log_b(n+1)$. □

Corollary: The height is $\Omega(\log_b n)$ and $\mathcal{O}(\log_a n)$.

Searching for a key

Find(x) follows the general algorithm for multi-way trees. It visits $\mathcal{O}(\log_a n)$ nodes, which is $\mathcal{O}(\log n/\log a)$. In each node, it compares x with all keys of the node, which can be performed in time $\mathcal{O}(\log b)$ by binary search. In total, we spend time $\Theta(\log n \cdot \log b/\log a)$.

If b is polynomial in a, the ratio of logarithms is $\Theta(1)$, so the complexity of Find is $\Theta(\log n)$. This is optimum since each non-final comparison brings at most 1 bit of information and we need to gather $\log n$ bits to determine the result.

Insertion

If we want to insert a key, we try to find it first. If the key is not present yet, the search ends in a leaf (external node). However, we cannot simply turn this leaf into an internal node with two external children — this would break the axiom that all leaves lie on the same level.

Instead, we insert the key to the parent of the external node — that is, to a node on the lowest internal level. Adding a key requires adding a child, so we add a leaf. This is correct since all other children of that node are also leaves.

If the node still has at most $b-1$ keys, we are done. Otherwise, we split the overfull node to two and distribute the keys approximately equally. In the parent of the split node, we need to replace one child pointer by two, so we have to add a key to the parent. We solve this by moving the middle key of the overfull node to the parent. Therefore we are splitting the overfull node to three parts: the middle key is moved to the parent, all smaller keys form one new node, and all larger keys form the other one. Children will be distributed among the new nodes in the only possible way.

This way, we have reduced insertion of a key to the current node to insertion of a key to its parent. Again, this can lead to the parent overflowing, so the splitting can continue, possibly up to the root. If it happens that we split the root, we create a new root with a single key and two children (this is correct, since we allowed less than a children in the root). This increases the height of the tree by 1.

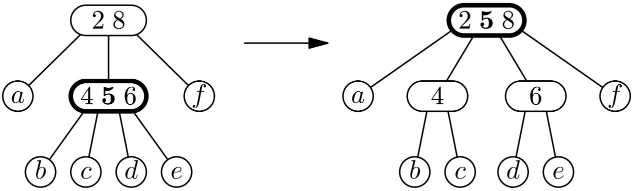

Figure 3.2: Splitting an overfull node on insertion to a $(2, 3)$-tree

Let us calculate time complexity of Insert. In the worst case, we visit $\Theta(1)$ nodes on each level and we spend $\Theta(b)$ time on each node. This makes $\Theta(b \cdot \log n / \log a)$ time total.

It remains to show that nodes created by splitting are not undersized, meaning they have at least a children. We split a node v when it reached $b + 1$ children, so it had b keys. We send one key to the parent, so the new nodes v_1 and v_2 will take $\lfloor (b-1)/2 \rfloor$ and $\lceil (b-1)/2 \rceil$ keys. If any of them were undersized, we would have $(b - 1)/2 < a - 1$ and thus $b < 2a - 1$. Voilà, this explains why we put the condition $b \geq 2a - 1$ in the definition.

Deletion

If we want to delete a key, we find it first. If it is located on the last internal level, we can delete it directly, together with one of the leaves under it. We still have to check for underflow, though.

Keys located on the higher levels cannot be removed directly — the internal node would lose one pointer and we would have a subtree left in our hands with no place to connect it to. This situation is similar to deletion of a node with two children in a binary search tree, so we can solve it similarly: We replace the deleted key by its successor (which is the leftmost key in the deleted key's right subtree). The successor lies on the last internal level, so it can be deleted directly.

The only remaining problem is fixing an undersized node. For a moment we will assume that the node is not the root, so it has $a - 2$ keys. It is tempting to solve the underflow by merging the node with one of its siblings. However, this can be done only if the sibling contains few keys; otherwise, the merged node could be overfull. But if the sibling is large, we can fix our problem by borrowing a key from it.

Let us be exact. Suppose that we have an undersized node v with $a - 2$ keys and this node has a left sibling ℓ separated by a key p in their common parent. If there is no left sibling, we use the right sibling and follow a mirror image of the procedure.

If the sibling has only a children, we merge nodes v and ℓ to a single node and we also move the key p from the parent there. This creates a node with $(a-2)+(a-1)+1 = 2a-2$

keys, which cannot exceed $b-1$. Therefore we reduced deletion from v to deletion from its parent.

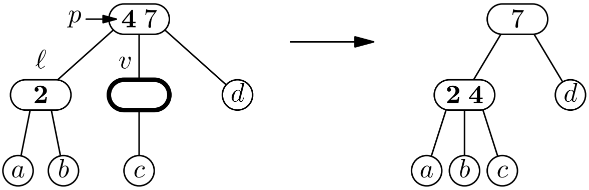

Figure 3.3: Merging nodes on deletion from a $(2,3)$-tree

On the contrary, if the sibling has more than a children, we disconnect its rightmost child c and its largest key m. Then we move the key m to the parent and the key p from the parent to v. There, p becomes the smallest key, before which we connect the child c. After this “rotation of keys”, all nodes will have numbers of children in the allowed range, so we can stop.

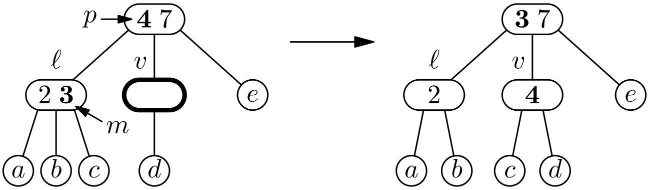

Figure 3.4: Borrowing a key from a sibling on deletion from a $(2,3)$-tree

In each step of the algorithm, we either produce a node which is not undersized, or we fix the underflow by borrowing from a sibling, or we merge two nodes. In the first two cases, we stop. In the third case, we continue by deleting a key on the next higher level, continuing possibly to the root. If the root becomes undersized, it has no keys left, in which case we make its only child the new root.

In the worst case, Delete visits $\Theta(1)$ nodes on each level and it spends $\Theta(b)$ time per node. Its time complexity is therefore $\Theta(b \cdot \log n / \log a)$.

The choice of parameters

For a fixed a, the complexity of all three operations increases with b, so we should set b small. The usual choices are the minimum value $2a-1$ allowed by the definition or $2a$. As we will see in the following sections, the latter value has several advantages.

If $b \in \Theta(a)$, the complexity of Find becomes $\Theta(\log n)$. Both Insert and Delete will run in time $\Theta(\log n \cdot (a/\log a))$. We can therefore conclude that we want to set a to 2 or possibly to another small constant. The best choices are the $(2,3)$-tree and the $(2,4)$-tree.

This is true on the RAM, or on any machine with ideal random-access memory. If we store the tree on a disk, the situation changes. A disk is divided to blocks (the typical size of a block is on the order of kilobytes) and I/O is performed on whole blocks. Reading a single byte is therefore as expensive as reading the full block. In this situation, we will set a to make the size of a node match the size of the block. Consider an example with 4 KB blocks, 32-bit keys, and 32-bit pointers. If we use the $(256,511)$-tree, one node will fit in a block and the tree will be very shallow: four levels suffice for storing more than 33 million keys. Furthermore, the last level contains only external nodes and we can keep the root cached in memory, so each search will read only 2 blocks.

The same principle actually applies to main memory of contemporary computers, too. They usually employ a fast *cache* between the processor and the (much slower) main memory. The communication between the cache and the memory involves transfer of *cache lines* — blocks of typical size around 64 B. It therefore helps if we match the size of nodes with the size of cache lines and we align the start of nodes to a multiple of cache line size. For 32-bit keys and 32-bit pointers, we can use a $(4,7)$-tree.

Other versions

Our definition of (a,b)-trees is not the only one used. Some authors prefer to store useful keys only on the last level and let the other internal nodes contain copies of these keys (typically minima of subtrees) for easy navigation. This requires minor modifications to our procedures, but the asymptotics stay the same. This version can be useful in databases, which usually associate potentially large data with each key. They have two different formats of nodes, possibly with different choices of a and b: leaves, which contain keys with associated data, and internal nodes with keys and pointers.

Database theorists often prefer the name *B-trees.* There are many definitions of B-trees in the wild, but they are usually equivalent to $(a,2a-1)$-trees or $(a,2a)$-trees, possibly modified as in the previous paragraph.

3.2 Amortized analysis

In an amortized sense, (a,b)-trees cannot perform better than in $\Omega(\log n)$, because all operations involve search, which can take $\mathcal{O}(\log n)$ repeatedly. However, if we already know the location of the key in the tree, the rest of the operation can amortize well. In particular, the amortized number of modified nodes is constant for some choices of a and b.

Theorem: A sequence of m INSERTs on an initially empty (a, b)-tree performs $\mathcal{O}(m)$ node modifications.

Proof: Each INSERT performs a (possibly empty) sequence of node splits and then it modifies one node. A split modifies two nodes: one existing and one newly created. Since each split creates a new node and the number of nodes never decreases, the total number of splits is bounded by the number of nodes at the end of the sequence, which cannot exceed m.

So we have $\mathcal{O}(m)$ splits, each modifying $\mathcal{O}(1)$ nodes, and $\mathcal{O}(m)$ modifications outside splits. □

When we start mixing insertions with deletions, the situation becomes more complicated. In a $(a, 2a-1)$-tree, it might happen that INSERT(x) forces splitting up to the root and the immediately following DELETE(x) undoes all the work and reverts the tree back to the original state. So we can construct an arbitrarily long sequence of operations which have $\Omega(\log n)$ cost each.

We already encountered this problem with the flexible arrays of section 1.3 — if the thresholds for stretching and shrinking are too close, the structure tends to oscillate expensively between two states. We cured this by allowing the structure extra "breathing space".

The same applies to (a, b)-trees. When we increase b to at least $2a$, it is sufficient to avoid oscillations and keep the amortized number of changes constant. For simplicity, we will prove this for the specific case $b = 2a$.

Theorem: A sequence of m INSERTs and DELETEs on an initially empty $(a, 2a)$-tree performs $\mathcal{O}(m)$ node modifications.

Proof: We define the cost of an operation as the number of nodes it modifies. We will show that there exists a potential Φ such that the amortized cost of splitting and merging with respect to Φ is zero or negative and the amortized cost of the rest of INSERT and DELETE is constant.

The potential will be a sum of node contributions. Every node with k keys contributes $f(k)$ to the potential, where f will be a certain non-negative function. We allow k from $a-2$ to $b = 2a$ to handle nodes, which are temporarily overflowed or underflowed. By definition, Φ is initially zero and always non-negative.

The function f should satisfy the following requirements:

1. $|f(i) - f(i+1)| \leq c$ for some constant c.

This means that if the number of keys in the node changes by 1, the node's contribution changes at most by a constant.

2. $f(2a) \geq f(a) + f(a-1) + c + 1$.

   This guarantees free splits: If we split a node with $2a$ keys, we release $f(2a)$ units of potential. We have to spend $f(a)$ and $f(a-1)$ units on the two new nodes and at most c on inserting a key to the parent node. Still, we have one unit to pay for the real cost. (Strictly speaking, the real cost is 3, but we can make it 1 by scaling the costs and the potential.)

3. $f(a-2) + f(a-1) \geq f(2a-2) + c + 1$.

   This guarantees free merges: We are always merging an underflowing node ($a-2$ keys) with a minimum allowed node ($a-1$ keys). The potential released by removing these nodes is spent on creation of the merged node with $2a-2$ keys, removing a key from the parent node and paying for the real cost of the operation.

By little experimentation, we can construct a function f meeting these criteria with $c = 2$:

k	$a-2$	$a-1$	a	$\ldots$	$2a-2$	$2a-1$	$2a$
$f(k)$	2	1	0	$\ldots$	0	2	4

The function is zero between a and $2a-2$.

An Insert begins by adding the new key, which has $\mathcal{O}(1)$ real cost and it changes the potential by $\mathcal{O}(1)$. Then it performs a sequence of splits, each with zero amortized cost.

A Delete removes the key, which has $\mathcal{O}(1)$ cost both real and amortized. If the node was undersized, it can perform a sequence of merges with zero amortized cost. Finally, it can borrow a key from a neighbor, which has $\mathcal{O}(1)$ real and amortized cost, but this happens at most once per Delete.

We conclude that the amortized cost of both operations is constant. As the potential is non-negative and initially zero, this guarantees that the total real cost of all operations is $\mathcal{O}(m)$. □

3.3 Top-down (a,b)-trees

Our implementation of Insert and Delete first traverses the tree from the root downwards, looking for the key. Then it performs the operation locally and traverses the same path upwards, fixing violations of invariants. We will show that if $b \geq 2a$, a single top-down traversal suffices.

Insert will employ *pre-emptive splitting* on the way down. It will maintain the invariant that the current node contains less than $b - 1$ keys (that is, the node is not completely full). Whenever it encounters a node with $b - 1$ keys, it splits the node. This requires adding a key to the parent, but we can be sure there is enough space for the key there. When we reach the bottommost level, we simply insert the new key there.

Delete is similar — we make sure that the current node has more keys than the minimum $a-1$. We accompish this by either borrowing from a sibling or by merging with a sibling. The latter requires pulling one key from the parent, which is always possible.

In chapter 9, we will use this approach to build a search tree with parallel access.

4 Heaps

Heaps are a family of data structures for dynamic selection of a minimum. They maintain a set of items, each equipped with a *priority.* The priorities are usually numeric, but we will assume that they come from some abstract universe and they can be only compared. Beside the priority, the items can carry arbitrary data, usually called the *value* of the item.

Heaps typically support the following operations. Since a heap cannot efficiently find an item by neither its priority nor value, operations that modify items are given an opaque identifier of the item. The identifiers are assigned on insertion; internally, they are usually pointers.

Insert(p, v)	Create a new item with priority p and value v and return its identifier.
Min	Find an item with minimum priority. If there are multiple such items, pick any. If the heap is empty, return $\emptyset$.
ExtractMin	Find an item with minimum priority and remove it from the heap. If the heap is empty, return $\emptyset$.
Decrease(id, p)	Decrease priority of the item identified by id to p.
Increase(id, p)	Increase priority of the item identified by id to p.
Delete(id)	Remove the given item from the heap. It is usually simulated by Decrease$(id, -\infty)$ followed by ExtractMin.
Build$((p_1, v_1), \ldots)$	Create a new heap containing the given items. It is equivalent to a sequence of Inserts on an empty heap, but it can be often implemented more efficiently.

By reversing the order of priorities, we can obtain a *maximal heap,* which maintains the maximum instead of the minimum.

Observation: We can sort a sequence of n items by inserting them to a heap and then calling ExtractMin n times. The standard lower bound on comparison-based sorting implies that at least one of Insert and ExtractMin must take $\Omega(\log n)$ time amortized.

We can implement the heap interface using a search tree, which yields $\mathcal{O}(\log n)$ time complexity of all operations except Build. Specialized constructions of heaps presented in this chapter will achieve $\mathcal{O}(\log n)$ amortized time for ExtractMin, $\mathcal{O}(n)$ for Build, and $\mathcal{O}(1)$ for all other operations.

Dijkstra's algorithm

Let us see one example where a heap outperforms search trees: the famous Dijkstra's algorithm for finding the shortest path in a graph.

TODO

Lemma: Dijkstra's algorithm with a heap runs in time $\mathcal{O}(n \cdot T_I(n) + n \cdot T_X(n) + m \cdot T_D(n))$. Here n and m are the number of vertices and edges of the graph, respectively. $T_I(n)$ is the amortized time complexity of Insert on a heap with at most n items, and similarly $T_X(n)$ for ExtractMin and $T_D(n)$ for Decrease.

4.1 Regular heaps

TODO

4.2 Binomial heaps

The binomial heap performs similarly to the regular heaps, but is has a more flexible structure, which will serve us well in later constructions. It supports all the usual heap operations. It is also able to Merge two heaps into one efficiently.

The binomial heap will be defined as a collection of binomial trees, so let us introduce these first.

Definition: The *binomial tree of rank k* is a rooted tree B_k with ordered children of each node such that:

1. B_0 contains only a root node.
2. B_k for $k > 0$ contains a root with k children, which are the roots of subtrees B_0, B_1, …, B_{k-1} in this order.[1]

If we wanted to be strictly formal, we would define B_k as any member of an isomorphism class of trees satisfying these properties.

Let us mention several important properties of binomial trees, which can be easily proved by induction on the rank. It might be useful to follow figures 4.1 and 4.2.

Observation:

[1] In fact, this works even for $k = 0$.

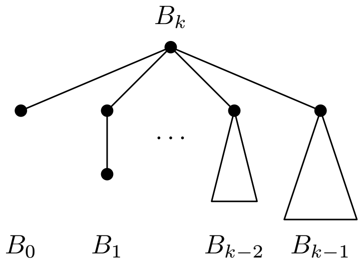

Figure 4.1: The binomial tree of rank k

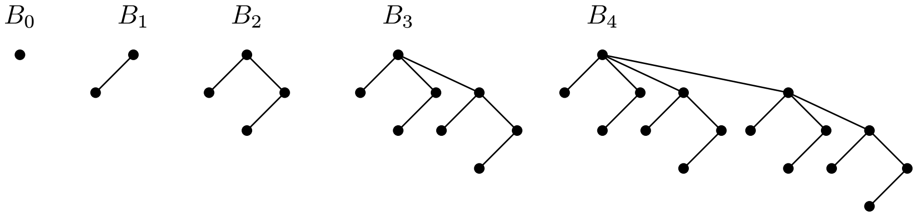

Figure 4.2: Binomial trees of small rank

- The binomial tree B_k has 2^k nodes on $k+1$ levels.
- B_k can be obtained by linking the roots of two copies of B_{k-1} (see figure 4.3).

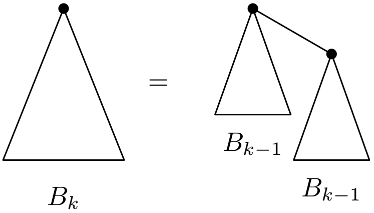

Figure 4.3: Joining binomial trees of the same rank

Definition: The *binomial heap* for a given set of items is a sequence $\mathcal{T} = T_1, \ldots, T_\ell$ of binomial trees such that:

1. The ranks of the trees are increasing: $\mathit{rank}(T_i) < \mathit{rank}(T_{i+1})$ for all i. In particular, this means that the ranks are distinct.

2. Each node of a tree contains a single item. We will use $p(v)$ for the priority of the item stored in a node v.

3. The items obey the *heap order* — if a node u is a parent of v, then $p(u) \le p(v)$.

Note: To ensure that all operations have the required time complexity, we need to be careful with the representation of the heap in memory. For each node, we will store:

- the rank (a rank of a non-root node is defined as the number of its children)
- the priority and other data of the item (we assume that the data can be copied in constant time; if they are large, we keep them separately and store only a pointer to the data in the node)
- a pointer to the first child
- pointers to the previous and next sibling (the children of a node form a circular doubly-linked list)
- a pointer to the parent (needed only for some operations, see exercise 4)

Since the sibling pointers play no role in tree roots, we can use them to chain trees in the heap. In fact, it is often useful to encode tree roots as children of a “meta-root”, which helps to unify all operations with lists.

As the trees in the heap have distinct ranks, their sizes are distinct powers of two, whose sum is the total number of items n. This is exactly the binary representation of the number n — the k-th digit (starting with 0) is 1 iff the tree B_k is present in the heap. Since each n has a unique binary representation, the shape of the heap is fully determined by its size. Still, we have a lot of freedom in location of items in the heap.

Corollary: A binomial heap with n items contains $\mathcal{O}(\log n)$ trees, whose ranks and heights are $\mathcal{O}(\log n)$. Each node has $\mathcal{O}(\log n)$ children.

Finding the minimum

Since the trees are heap-ordered, the minimum item of each tree is located in its root. To find the minimum of the whole heap, we have to examine the roots of all trees. This can be done in time linear in the number of trees, that is $\mathcal{O}(\log n)$.

If the Min operation is called frequently, we can speed it up to $\mathcal{O}(1)$ by caching a pointer to the current minimum. The cache can be updated during all other operations at the cost of a constant slow-down. We are leaving this modification as an exercise.

Merging two heaps

All other operations are built on Merge. This operation takes two heaps H_1 and H_2 and it constructs a new heap H containing all items of H_1 and H_2. The original heaps will be destroyed.

In Merge, we scan the lists of trees in each heap in the order of increasing ranks, as when merging two sorted lists. If a given rank is present in only one of the lists, we move that tree to the output. If we have the same rank in both lists, we cannot use both trees, since it would violate the requirements that all ranks are distinct. In this case, we *join* the trees to form B_{k+1} as in figure 4.3: the tree whose root has the smaller priority will stay as the root of the new tree, the other root will become its child. Of course, it can happen that rank $k + 1$ is already present in one or more lists, but this can be solved by further merging.

The whole process is similar to addition of binary numbers. We are processing the trees in order of increasing rank k. In each step, we have at most one tree of rank k in each heap, and at most one tree of rank k carried over from the previous step. If we have two or more trees of rank k, we join a pair of them and send it as a carry to the next step. This leaves us with at most one tree, which we can send to the output.

As a minor improvement, when we exhaust one of the input lists and we have no carry, we can link the rest of the other list to the output in constant time.

Let us analyze the time complexity. The maximum rank in both lists is $\mathcal{O}(\log n)$, where n is the total number of items in both heaps. For each rank, we spend $\mathcal{O}(1)$ time. If we end up with a carry after both lists are exhausted, we need one more step to process it. This means that Merge always finishes in $\mathcal{O}(\log n)$ time.

Inserting items

Insert is easy. We create a new heap with a single binomial tree of rank 0, whose root contains the new item. Then we merge this heap to the current heap. This obviously works in time $\mathcal{O}(\log n)$.

Building a heap of n given items is done by repeating Insert. We are going to show that a single Insert takes constant amortized time, so the whole Build runs in $\mathcal{O}(n)$ time. We observe that the Merge inside the Insert behaves as a binary increment and it runs in time linear in the number of bits changed during that increment. (Here it is crucial that we stop merging when one of the lists is exhausted.) Thus we can apply the amortized analysis of binary counters we developed in section 1.3.

Extracting the minimum

The ExtractMin operation will be again based on Merge. We start by locating the minimum as in Min. We find the minimum in the root of one of the trees. We unlink this tree from the heap. Then we remove its root, so the tree falls apart to binomial trees of all lower ranks (remember figure 4.1). As the ranks of these trees are strictly increasing, the trees form a correct binomial heap, which can be merged back to the current heap.

Finding the minimum takes $\mathcal{O}(\log n)$. Disassembling the tree at the root and collecting the sub-trees in a new heap takes $\mathcal{O}(\log n)$; actually, it can be done in constant time if we are using the representation with the meta-root. Merging these trees back is again $\mathcal{O}(\log n)$. We conclude that the whole ExtractMin runs in time $\mathcal{O}(\log n)$.

Decreasing and increasing

A Decrease can be performed as in a regular heap. We modify the priority of the node, which can break heap order at the edge to the parent. If this happens, we swap the two items, which can break the order one level higher, and so on. In the worst case, the item bubbles up all the way to the root, which takes $\mathcal{O}(\log n)$ time.

An Increase can be done by bubbling items down, but this is slow. As each node can have logarithmically many children, we spend $\mathcal{O}(\log n)$ time per step, which is $\mathcal{O}(\log^2 n)$ total. It is faster to Delete the item by decreasing it to $-\infty$ and performing an ExtractMin, and then inserting it back with the new priority. This is $\mathcal{O}(\log n)$.

Summary of operations

operation	*worst-case complexity*
Insert	$\Theta(\log n)$
Min (cached)	$\Theta(1)$
ExtractMin	$\Theta(\log n)$
Merge	$\Theta(\log n)$
Build	$\Theta(n)$
Decrease	$\Theta(\log n)$
Increase	$\Theta(\log n)$
Delete	$\Theta(\log n)$

We proved only the asymptotic upper bounds, but it is trivial to verify that these are tight on some inputs.

Compared with the binary heap, the binomial heap works in the same time, but it additionally supports efficient merging.

Exercises

1. The name of binomial trees comes from the fact that the number of nodes on the i-th level (starting with 0) in a rank-k tree is equal to the binomial coefficient $\binom{k}{i}$. Prove it.

2. How many leaves does the tree B_k have?

3. The binomial tree B_k is a spanning tree of the k-dimensional hypercube. And indeed a special one: show that it is a shortest path tree of the hypercube.

4. Show that if we need only Insert and ExtractMin, we can omit the parent pointers. Which other operations need them?

5. Show that we can save one more pointer per node by making the list of children single-linked. Keep in mind that joining two trees has to keep the children of the root ordered by rank.

6. Show that we can cache the minimum with only a constant slowdown of all operations.

7.* Define a *trinomial heap,* whose trees have sizes equal to the powers of 3. In a single heap, there will be at most two trees of any given rank. Show how to implement heap operations. Compare their complexity with the binomial heap.

4.3 Lazy binomial heaps

The previous construction is sometimes called the *strict binomial heap,* as opposed to the *lazy* version, which will be introduced in this section.

The lazy binomial heap is slow in the worst case, but fast in the amortized sense. It is obtained by dropping the requirement that the ranks of trees in a heap are sorted and distinct. This can lead to a very degenerate heap, which is just a linked list of single-node trees. Repeated ExtractMins on such heaps would be too slow, so we make each extraction *consolidate* the heap — bring it to the strict form with distinct ranks. The consolidation is expensive, but an amortization argument will show that all consolidations can be pre-paid by the other operations.

A Merge of two lazy heaps is trivial: we just concatenate their lists of trees. With circular lists, this can be done in constant time. Therefore both Merge and Insert run in $\mathcal{O}(1)$ time. A build makes n Inserts, so it runs in $\mathcal{O}(n)$ time.

Decrease does not need to be modified. It is local to the tree containing the particular item. Even in a lazy heap, the rank of this tree and its height are still $\mathcal{O}(\log n)$, and so is the time complexity of Decrease. Both Delete and Increase will be again reduced to Decrease, ExtractMin, and Insert.

ExtractMin and consolidation

An ExtractMin is implemented similarly to the strict heap. We find the tree whose root has minimum priority. This requires $\mathcal{O}(t)$ time, where t is the number of trees in the heap. Then we remove the tree from the heap, disassemble it to subtrees and merge these trees back to the heap. This can be done in constant time with circular lists.

Then we consolidate the heap by bucket sorting. We create an array of $\lfloor\log n\rfloor+1$ buckets, one for each possible rank. We process the trees one by one. If we have a tree of rank r, we look in the r-th bucket. If the bucket is empty, we put the tree there. Otherwise, we remove the tree from the bucket and join it with our tree. This increases its rank r by one. We repeat this until we find an empty bucket. Then we proceed to the next tree of the heap. At the end, we collect all trees from the buckets and put them back to the heap.

How much time does the consolidation take? We spend $\mathcal{O}(\log n)$ time on initializing the buckets and collecting the trees from them at the end. We add t trees and since each join decreases the number of trees by 1, we spend $\mathcal{O}(t)$ time on the joins. Therefore the whole consolidation runs in $\mathcal{O}(t + \log n)$ time.

We conclude that ExtractMin runs in $\mathcal{O}(t + \log n)$ time, which is $\mathcal{O}(n)$ in the worst case.

Amortized analysis

Let us show that while ExtractMin is very expensive in the worst case, it actually amortizes well.

We define a potential Φ equal to the total number of trees in all heaps. The potential is initially zero and always non-negative.

Merge has constant real cost and since it only redistributes trees between heaps, it does not change the potential. Hence its amortized cost is also $\mathcal{O}(1)$. An Insert creates a new node, which has constant real cost and it increases the potential by 1. Together with the merge, its amortized cost is $\mathcal{O}(1)$.

Decrease does not change the potential, so its amortized cost is equal to the real cost $\mathcal{O}(\log n)$.

The real cost of ExtractMin on a heap with t trees is $\mathcal{O}(t + \log n)$. At the end, all trees have distinct ranks, so the potential drops from t to at most $\log n$. The potential difference is therefore at most $\log n - t$, so after suitable scaling it offsets the $\mathcal{O}(t)$ term of the real cost. This makes the amortized cost $\mathcal{O}(\log n)$.

Build, Increase, and Delete are composed of the other operations, so we can just add the respective amortized costs.

Summary of operations

operation	*worst-case complexity*	*amortized complexity*
Insert	$\Theta(1)$	$\Theta(1)$
Min (cached)	$\Theta(1)$	$\Theta(1)$
ExtractMin	$\Theta(n)$	$\Theta(\log n)$
Merge	$\Theta(1)$	$\Theta(1)$
Build	$\Theta(n)$	$\Theta(n)$
Decrease	$\Theta(\log n)$	$\Theta(\log n)$
Increase	$\Theta(n)$	$\Theta(\log n)$
Delete	$\Theta(n)$	$\Theta(\log n)$

We proved only the asymptotic upper bounds, but it is trivial to verify that these are tight on some inputs.

Compared with its strict counterpart, the lazy binomial heap achives better amortized complexity of Insert and Merge. Still, Decrease is too slow to improve Dijkstra's algorithm.

Exercises

1. *Consolidation by pairing:* Consider a simplified procedure for consolidating a heap. When it joins two trees of the same rank, it immediately sends them to the resulting heap instead of trying to put them in the next higher bucket. This consolidation no longer guarantees that the resulting trees have distinct ranks, but prove that the amortized cost relative to the potential Φ is not harmed.

4.4 Fibonacci heaps

To make Decrease faster, we need to implement it in a radically different way. If decreasing a node's priority breaks heap order on the edge to the parent, we cut this edge and make the decreased node a root of its own tree. Repeated cuts can distort the shape of the trees too much, so we avoid cutting off more than one child of a non-root node. If we want to cut off a second child, we cascade the cut to make the parent a new root, too.

This idea leads to the Fibonacci heaps, discovered by Fredman and Tarjan in 1987. We will develop them from the lazy binomial heaps, but we drop all requirements on the shape of the trees.

Definition: The *Fibonacci heap* for a given set of items is a sequence $\mathcal{T} = T_1, \ldots, T_\ell$ of rooted trees such that:

1. Each node contains a single item. We will use $p(v)$ for the priority of the item stored in a node v.
2. The items obey the *heap order* — if a node u is a parent of v, then $p(u) \leq p(v)$.
3. Each node can be *marked* if it lost a child. Tree roots are never marked.

Note: The representation of each node in memory will contain:

- the rank, defined as the number of children (the rank of a tree is defined as the rank of its root)
- the priority and other data of the item
- the boolean mark
- a pointer to the first child
- a pointer to the previous and next sibling in a circular list of children
- a pointer to the parent

Operations

Merge works as in lazy binomial heaps: It just concatenates the lists of trees.

Insert creates a heap consisting of one unmarked node and merges it to the current heap. Build is an iterated Insert.

ExtractMin is again similar to the lazy binomial heap. The only difference is that when it disassembles a tree, it has to unmark all new roots.

Decrease changes the priority of the node. If it becomes lower than the priority of its parent, we Cut the node to make it a root.

Cut disconnects a node from the list of siblings, unmarks it, and connects it to the list of roots. If the original parent is not a root, it sets its mark. If the mark was already set, it calls Cut on the parent recursively.

As usual, Increase and Delete are reduced to the other operations.

Worst-case analysis

Worst-case complexity of operations easily follows from their description. The only exception is the consolidation inside ExtractMin, which needs to create an array of buckets indexed by ranks. In binomial heaps, the maximum rank was easily shown to be at most $\log n$. The trees in Fibonacci heaps can have less regular shapes, but it is also possible to prove that their ranks are at most logarithmic.

The analysis involves Fibonacci numbers, from which the heap got its name. We define the Fibonacci sequence by $F_0 = 0$, $F_1 = 1$, and $F_{n+2} = F_{n+1} + F_n$. It is well known that

$F_n = \Theta(\varphi^n)$, where $\varphi = (1+\sqrt{5})/2 \doteq 1.618$ is the golden ratio. The following lemma will be useful:

Lemma: $F_0 + F_1 + \ldots + F_k = F_{k+2} - 1$.

Proof: By induction on k. For $k = 0$, we have $F_0 = F_2 - 1$, that is $0 = 1 - 1$. In the induction step, we increase both sides of the equality by F_{k+1}. The right-hand side becomes $F_{k+2} - 1 + F_{k+1} = F_{k+3} - 1$ as needed. □

Now we use Fibonacci numbers to show that the sizes of trees grow exponentially with their rank.

Lemma: A node of rank k has at least F_{k+2} descendants (including itself).

Proof: We will proceed by induction on the height of the node. If the height is 0, the node is a leaf, so it has rank 0 and $F_2 = 1$ descendants.

Now, let u be a node of rank $k > 0$. Let $x_1, \ldots, x_k$ be its children in the order of their linking to u, and $r_1, \ldots, r_k$ their ranks. In a binomial heap, the ranks would be $0, \ldots, k-1$. Here, we are able to prove a weaker, but still useful claim:

> **Claim:** For every $i \geq 2$, we have $r_i \geq i - 2$.
>
> *Proof:* Consider the moment when x_i was linked to u. At that time, $x_1, \ldots, x_{i-1}$ were already the children of u, but there might have been more children, which were removed later. So the rank of u was at least $i - 1$. Nodes are linked only when we join two trees of the same rank, so the rank of x_i must have been also at least $i-1$. As x_i is not a root, it might have lost at most one child since that time, so its current rank is at least $i - 2$. □

The ranks $r_1, \ldots, r_k$ are therefore at least $0, 0, 1, 2, \ldots, k-2$. As the height of children is smaller than the height of the parent, we can apply the induction hypothesis, so the sizes of subtrees are at least $F_2, F_2, F_3, F_4, \ldots, F_k$. As $F_0 = 0$ and $F_1 = 1 = F_2$, the total number of descendants of u, including u itself, can be written as $F_0 + F_1 + F_2 + \ldots + F_k + 1$. By the previous lemma, this is equal to F_{k+2}. This completes the induction step. □

Corollary R: The ranks of all nodes are $\mathcal{O}(\log n)$.

Let us return back to the complexity of operations. Merge and Insert run in $\mathcal{O}(1)$ time. Build does n Inserts, so it runs in $\mathcal{O}(n)$ time.

ExtractMin on a heap with t trees uses $\mathcal{O}(\log n)$ buckets (because ranks are logarithmic by corrolary **R**), so it takes $\mathcal{O}(t + \log n)$ time.

Decrease does a Cut, which runs in time linear in the number of marked nodes it encounters. This can be up to $\mathcal{O}(n)$.

Amortized analysis

There are two procedures which can take $\mathcal{O}(n)$ time in the worst case: extraction of the minimum (which is linear in the number of trees) and cascaded cut (linear in the number of marked nodes it encounters). To amortize them, we define a potential, which combines trees with marked nodes:

$$\Phi := \textit{number of trees in all heaps} + 2 \cdot \textit{number of marked nodes}.$$

The potential is initially zero and always non-negative.

MERGE has constant real cost and it does not change the potential. Hence the amortized cost is also constant.

INSERT has constant real cost and it increases Φ by 1, so the amortized cost is constant.

EXTRACTMIN on a heap with t trees runs in real time $\mathcal{O}(t + \log n)$. The final number of trees is $\mathcal{O}(\log n)$ and marks are only removed, so the potential difference is $\mathcal{O}(\log n) - t$. With suitable scaling of units, this makes the amortized cost $\mathcal{O}(\log n)$.

CUT of a node whose parent is either unmarked or a root takes $\mathcal{O}(1)$ real time and changes the potential by $\mathcal{O}(1)$. If the parent is marked, we cascade the cut. This involves unmarking the parent, which decreases Φ by 2 units. Then we make the parent a root of a stand-alone tree, which increases Φ by 1. The total change of Φ is -1, which pays for the constant real cost of the cut. The amortized cost of the cascading is therefore zero, so the total amortized cost of CUT is $\mathcal{O}(1)$.

Summary of operations

operation	*worst-case complexity*	*amortized complexity*
INSERT	$\Theta(1)$	$\Theta(1)$
MIN (cached)	$\Theta(1)$	$\Theta(1)$
EXTRACTMIN	$\Theta(n)$	$\Theta(\log n)$
MERGE	$\Theta(1)$	$\Theta(1)$
BUILD	$\Theta(n)$	$\Theta(n)$
DECREASE	$\Theta(n)$	$\Theta(1)$
INCREASE	$\Theta(n)$	$\Theta(\log n)$
DELETE	$\Theta(n)$	$\Theta(\log n)$

As usual, we proved only the asymptotic upper bounds, but it is easy to verify that that are tight on some inputs (see exercise 2).

Dijkstra's algorithm with Fibonacci heap therefore runs in $\mathcal{O}(m+n\log n)$ time. For dense graphs ($m \approx n^2$), this is $\mathcal{O}(n^2)$; for sparse graphs ($m \approx n$), we get $\mathcal{O}(n \log n)$. We note that $\Omega(n \log n)$ is not easy to surpass, since Dijkstra's algorithm sorts vertices by their distance. There exist more efficient algorithms, but they always make further assumptions on the edge lengths — e.g., they assume that the lengths are small integers.

Exercises

1. Caching of minimum works in the Fibonacci heaps, too. Show how to do it and prove that the amortized cost of operations does not change.

2. Show that there is a sequence of $\mathcal{O}(n)$ operations on a Fibonacci heaps, which produces a heap, whose only tree is a path with all n nodes marked except for the root. This gives a lower bound for worst-case complexity of Decrease. It also implies that the trees of Fibonacci heap are not necessarily subtrees of the binomial trees of the same rank.

3. Is it possible to implement an Increase operation running in better than logarithmic amortized time?

4. For each k, construct a sequence operations on the Fibonacci heap, which produces a tree with exactly F_{k+2} nodes.

5.* Consider a Fibonacci-like heap which does not mark nodes. Show that there exists a sequence of operations, which produces a tree with n nodes and rank $r \approx \sqrt{n}$, where the children of the root have ranks $0, \ldots, r-1$. Then we can force this tree to disassemble and re-assemble again, which costs $\Theta(\sqrt{n})$ repeatedly.

6. Show that using a different potential, the amortized cost of ExtractMin can be lowered to $\mathcal{O}(1)$ at the expense of raising the amortized cost of Insert to $\mathcal{O}(\log n)$. Beware that the n here is the current number of items as opposed to the maximum number of items over the whole sequence of operations.

5 Caching

Processors of modern computers are faster than their memory by several orders of magnitude. To avoid waiting for data most of the time, they use caching. They employ a small, but very fast memory called the *cache,* which stores a copy of frequently used data. Some machines have multiple levels of caches, each slightly larger and slightly slower than the previous one. We can also work with data stored on a slow medium like a hard disk and cache it in our main memory. Even further: data stored on other machines in the network can be cached on a local disk.

In short, caching is ubiquitious and algorithms whose access to data can be cached well are much faster than their cache-unfriendly counterparts. In this chapter, we are going to develop several techniques for designing cache-efficient algorithms and data structures. However, we must start with re-defininining our model of computation.

5.1 Models of computation

External memory model

The first model we will consider is the *external memory model,* also known as the *I/O model.* It was originally designed for study of algorithms working with data on disks, which do not fit in the machine's main memory.

The model possesses two types of memory:

- *External memory* of potentially infinite size, organized in blocks of size B — since we are going to study asymptotics, units do not matter, so we will measure all sizes in arbitrary *items.*
- *Internal memory* for M items, also organized in B-item blocks.

All computations are performed in the internal memory, similarly to the Random Access Machine. External memory can be accessed only indirectly: a block of data has to be loaded to the internal memory first and if it is modified, it has to be written back to the external memory later. We can assume that the machine has special instructions for that.

In addition to time and space complexity, we are going to study the *I/O complexity* of algorithms: the maximum number of input/output operations (reads and writes of blocks) performed by the algorithm for a given size of input.

We will often reduce calculation of I/O complexity to just counting the number of block reads. In most cases, it is easy to verify that the number of writes is bounded by the number of reads, at least asymptotically. When intermediate results are written, they are read later. Unless the final output is asymptotically larger than the input, the number of writes needed to create the output is bounded by the number of reads needed to obtain the input.

Cache models

Machines, which accelerate access to main memory by using a cache, can be modelled in a similar way. The role of external memory is played by the machine's main memory. The internal memory is the cache. (Or perhaps we are caching data on a disk: then the external memory is the disk and internal memory is the main memory.)

There is one crucial difference, though. Unlike in the I/O model, block transfers between main memory and the cache are not controlled by the program itself, but by the machine. The program accesses data in the main memory and a part of the machine (which will be called the *cache controller*) handles transfers of the particular blocks between the main memory and the cache automatically.

We will assume that the cache controller works optimally — that is, it decides which blocks should be kept in the cache and which should be evicted in a way minimizing the total number of I/O operations. This assumption is obviously unrealistic: an optimal cache controller needs to know the future accesses. However, we will prove in section 5.4 that there exists a deterministic strategy which approximates this controller within a constant factor.

There are two varieties of this cache model: the *cache-aware* model, in which the algorithm knows the parameters of the cache (the block size B and the cache size M), and the *cache-oblivious* model, where it doesn't.

It will be easier to design algorithms for the cache-aware model, but cache-oblivious algorithms are more useful. One such algorithm can work efficiently on many machines with different cache architectures. And not only that: if the machine uses a hierarchy of several caches, the algorithm can be shown to be optimal (within a constant factor) with respect to all caches simultaneously.

5.2 Basic algorithms

Scanning an array

Let us consider a simple example first. We have an array of N items, which we read sequentially from the start to the end. This is called an *array scan.*

In the I/O model, we can make the array start at a block boundary. So we need to read $\lceil N/B \rceil \le N/B + 1$ consecutive blocks to scan all items. All blocks can be stored at the same place in the internal memory. This strategy is obviously optimal.

A cache-aware algorithm can use the same sequence of reads. Generally, we do not know the sequence of reads used by the optimal caching strategy, but any specific sequence can serve as an upper bound. For example, the sequence we used in the I/O model.

A cache-oblivious algorithm cannot guarantee that the array will be aligned on a block boundary (it does not know B). In the worst case, this can cost us an extra read — imagine an array of size 2 spanning two blocks.

Asymptotically speaking, the I/O complexity of an array scan is $\mathcal{O}(N/B + 1)$ in all three models. Please note that the additive 1 cannot be „hidden inside $\mathcal{O}$“, because for every $\varepsilon > 0$, we have $N/B \le \varepsilon$ for infinitely many pairs (N, B). Also, in caching models we cannot replace the $\mathcal{O}$ by Θ since some of the blocks could be already present in the cache.

Mergesort

The traditional choice of algorithm for sorting in external memory is Mergesort. Let us analyze it in our models. Merging two sorted arrays involves three sequential scans: one for each source array, one for the destination array. The scans are interleaved, but so we can interleave their I/O operations. We need 1 block of internal memory for each scan, but let us generally assume that $M/B \ge c$ for any fixed constant c. Therefore the merge runs in linear time and it transfers $\mathcal{O}(T/B + 1)$ blocks, where T is the total number of items merged.

Our Mergesort will proceed bottom-up. Items will be stored in a single array as a sequence of same-size *runs* (sorted ranges), except for the last run which can be shorter. We start with 1-item runs. In each pass, we double the size of runs by merging pairs of consecutive runs. After the i-th pass, we have 2^i-item runs, so we stop after $\lceil \log N \rceil$ passes.

Each pass runs in $\Theta(N)$ time, so the whole algorithm has time complexity $\Theta(N \log N)$. This is optimal for comparison-based sorting.

Let us analyze I/O complexity. All merges in a single pass actually form three scans of the whole array, so they perform $\mathcal{O}(N/B + 1)$ block transfers. All passes together transfer $\mathcal{O}(N/B \cdot \log N + \log N)$ blocks. The extra $\log N$ is actually exaggerated: the $+1$ in complexity of a single pass is asymptotically significant only if $N < B$, that is if the whole input is smaller than a block. In that case, all passes happily compute on the same cached block. We can therefore improve the bound to $\mathcal{O}(N/B \cdot \log N + 1)$ in all three models.

Multi-way Mergesort

Could we take advantage of a cache larger than 3 blocks? Yes, with a multi-way Mergesort. A K-way Mergesort combines K runs at once, so the number of passes decreases to $\lceil \log_K N \rceil = \lceil \log N / \log K \rceil$. A K-way merge needs to locate a minimum of K items in every step, which can be done using a heap. Every step therefore takes time $\Theta(\log K)$, so merging T items takes $\Theta(T \log K)$ and the whole Mergesort $\Theta(N \log K \cdot \log N / \log K) = \Theta(N \log N)$ for any K. (For $K = N$, we actually get Heapsort.)

If we have large enough cache during the merge, every input array has its own scan and the heap fits in cache. Then the total I/O complexity is $\mathcal{O}(T/B + K)$. The extra K is significant in the situation when all runs are small and each is located in a different block. This actually does not happen in Mergesort: all the runs are always consecutive in memory. Therefore $\mathcal{O}(T/B + 1)$ transfers are enough. As in ordinary 2-way Mergesort, all merges during a pass perform $\mathcal{O}(N/B + 1)$ transfers. Similarly, multiplying this by the number of passes yields $\mathcal{O}(N/B \cdot \log N / \log K + 1)$.

How large K does our cache allow? Each scan requires its own cached block, which is $K+1$ blocks total. Another $K - 1$ blocks are more than enough for the heap, so $M \geq 2BK$ is sufficient. If we know M and B, we can set $K = \lfloor M/2B \rfloor$. Then the I/O complexity will reach $\mathcal{O}(N/B \cdot \log N / \log(M/B) + 1)$. This is actually known to be optimal — surprisingly, there is a matching lower bound (FIXME: reference) not only for sorting, but for only permuting items.

In the cache-oblivious model, we have no chance to pick the right number of ways. Still, there exists a rather complicated algorithm called Funnelsort (FIXME: ref) which achieves the same I/O complexity, at least asymptotically. We refer readers to the original article.

5.3 Matrix transposition

Another typical class of algorithms working with large data deals with matrices. We will focus on a simple task: transposition of a square matrix.

Matrices are traditionally stored in row-major order.[1] That is, the whole $N \times N$ matrix is stored as a single array of N^2 items read row by row. Accessing the matrix row by row therefore involves a sequential scan of the underlying array, so it takes $\mathcal{O}(N^2/B + 1)$ block transfers.

What about column-by-column access? Reading a single column is much more expensive: if the rows are large, each item in the column is located in a different block. When we

[1] A particular exception is Fortran and MATLAB, which use column-major order for some mysterious reason.

switch to the next column, we usually need the same N blocks. But unless the cache is big (meaning $M \in \Omega(NB)$), most blocks are already evicted from the cache. The total I/O complexity can therefore reach $\Theta(N^2)$.

A simple algorithm for transposing a matrix walks through the lower triangle and swaps each item with the corresponding item in the upper triangle. However, this means that if we are accessing one of the triangles row-by-row, the other will be accessed column-by-column. So the whole transposition will transfer $\Theta(N^2)$ blocks.

Using tiles

We will show a simple cache-aware algorithm with better I/O complexity. We split the matrix to *tiles* — sub-matrices of size $d \times d$, with smaller, possibly rectangular tiles at the border if N is not a multiple of d. The number of tiles is $\lceil N/d \rceil^2 \le (N/d+1)^2 \in \mathcal{O}(N^2/d^2+1)$.

Each tile can be transposed on its own, then we have to swap each off-diagonal tile with its counterpart in the other triangle. Obviously, the time complexity is still $\Theta(N^2)$. With a bit of luck, a single tile will fit in the cache completely, so its transposition will be I/O-efficient.

First, we shall analyse the case in which the size of the matrix N is a multiple of the block size B. If it is so, we can align the start of the matrix to the beginning of a block, so the start of each row will be also aligned. If we set $d = B$, every tile will be also aligned and each row of the tile will be a complete block. If we have enough cache, we can process a tile in $\mathcal{O}(B)$ I/O operations. As we have N^2/B^2 tiles, the total I/O complexity is $\mathcal{O}(N^2/B)$.

For this algorithm to work, the cache must be able to hold two tiles at once. Since each tile contains B^2 items, this means $M \ge 2B^2$. An inequality of this kind is usually called *tall-cache property* and it is satisfied by most hardware caches (but not when caching a disk with large blocks in main memory). Intuitively, it means that if we view the cache as a rectangle whose rows are the blocks, the rectangle will be higher than it is wide. More generally, a tall cache has $M \in \Omega(B^2)$. For every possible constant in Ω, we can make tiles constant-times smaller to fit in the cache and the I/O complexity of our algorithm will not change asymptotically.

Now, what if N is not divisible by B? We lose all alignment, but we will prove that the algorithm still works. Consider a $B \times B$ tile. In the worst case, each row spans 2 blocks. So we need $2B$ I/O operations to read it into cache, which is still $\mathcal{O}(B)$. The cache must contain at least $4B^2$ items, but this is still within limits of our tall-cache assumption.

To process all $\mathcal{O}(N^2/B^2+1)$ tiles, we need $\mathcal{O}(N^2/B+B)$ operations. As usual, this can be improved to $\mathcal{O}(N^2/B+1)$ if we realize that the additional term is required only in cases where the whole matrix is smaller than a single block.

We can conclude that in the cache-aware model, we can transpose a $N \times N$ matrix in time $\Theta(N^2)$ with $\mathcal{O}(N^2/B+1)$ block transfers. This is obviously optimal.

Divide and conquer

Optimal I/O complexity can be achieved even in the cache-oblivious model, but we have to be a little bit more subtle. Since we cannot guess the right tile size, we will try to approximate it by recursive subdivisions. This leads to the following divide-and-conquer algorithm.

For a moment, we will assume that N is a power of 2. We can split the given matrix A to 2×2 *quadrants* of size $N/2 \times N/2$. Let us call them A_{11}, A_{12}, A_{21}, and A_{22}. The diagonal quadrants A_{11} and A_{22} will be transposed recursively. The other quadrants A_{12} and A_{21} will have to be transposed, but also swapped. (This is the same idea as in the previous algorithm, but with tiles of size $N/2$.)

However, transposing first and then swapping would spoil time complexity (try to prove this). We will rather apply divide and conquer on a more general problem: given two matrices, transpose them and swap them. This problem admits a similar recursive decomposition (see figure 5.1). If we split two matrices A and B to quadrants, we have to transpose-and-swap A_{11} with B_{11}, A_{22} with B_{22}, A_{12} with B_{21}, and A_{21} with B_{12}.

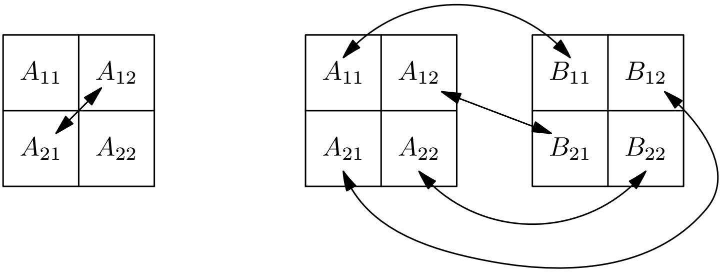

Figure 5.1: Divide-and-conquer matrix transposition

A single transpose-and-swap (TS) therefore recurses on 4-subproblems, which are again TS. A transpose (T) problem recurses on 3 sub-problems, two of which are T and one is TS. All these sub-problems have size $N/2$.

To establish time and I/O complexity, we consider the tree of recursion. Every node corresponds to a T or TS problem. It has 3 or 4 children for its sub-problems. At level i

(counted from the root, which is at level 0), we have at most 4^i nodes with sub-problems of size $N/2^i$. Therefore, the height of the tree is at most $\log N$ and it has at most $4^{\log N} = N^2$ leaves. Since all internal nodes have least 2 children (in fact, 3 or 4), there are less internal nodes than leaves.

In every TS leaf, we swap a pair of items. In every T leaf, nothing happens. Internal nodes only redistribute work and they do not touch items. So every node takes $\mathcal{O}(1)$ time and the whole algorithm finishes in $\mathcal{O}(N^2)$ steps.

To analyze I/O complexity, we focus on the highest level, at which the sub-problems correspond to tiles from the previous algorithm. Specifically, we will find the smallest i such that the sub-problem size $d = N/2^i$ is at most B. Unless the whole input is small and $i = 0$, this implies $2d = N/2^{i-1} > B$. Therefore $B/2 < d \leq B$.

To establish an upper bound on the optimal number of block transfers, we show a specific caching strategy. Above level i, we cache nothing — this is correct, since we touch no items. (Well, we need a little cache for auxiliary variables like the recursion stack, but this is asymptotically insignificant.) When we enter a node at level i, we load the whole sub-problem to the cache and compute the whole subtree in the cache. Doing this in all nodes of level i is equivalent to running the cache-aware algorithm for $d \in \Theta(B)$, which requires $\mathcal{O}(N^2/B)$ block transfers in total, provided that the cache is tall enough.

Finally, what if N is not a power of 2? When we sub-divide a matrix of odd size, we need to round one half of N down and the other up. This makes off-diagonal quadrants rectangular. Fortunately, it is easy to prove that all matrices we generate are either square, or almost-square, meaning that the lengths of sides differ by at most 1. We are leaving the proof as an exercise to the reader. For almost-square matrices, our reasoning about the required number of block transfers still applies.

We have reached optimal I/O complexity $\mathcal{O}(N^2/B + 1)$ even in the cache-oblivious model. The algorithm still runs in time $\Theta(N^2)$. In a real implementation, we can reduce its overhead by stopping recursion at sub-problems of some fixed size greater than 1. This comes at the expense of I/O complexity in very small caches, but these do not occur in practice.

The technique we have used is quite typical for design of cache-oblivious algorithms. We start with a cache-aware algorithm based on decomposing the problem to tiles of some kind. We show that for a particular tile size (typically related to parameters of the cache), the tiles fit in the cache. Then we design a cache-oblivious algorithm, which replaces knowledge of the right tile size by successively refined decomposition. At some level of the refinement — which is not known to the algorithm, but known to us during the analysis — the size of tiles is right up to a constant, so the I/O complexity matches the cache-aware algorithm up to a constant.

5.4 Model versus reality

Our theoretical models of caching assume that the cache is controlled in an optimal way. This is obviously impossible to achieve since it requires knowledge of all future memory accesses. Surprisingly, we will show that there exists a deterministic strategy which aproximates the optimal strategy well enough.

We will compare caching strategies by the following experiment: The main memory is divided to blocks, each identified by its address. We are given a sequence $a_1, a_2, \ldots, a_n$ of requests for particular blocks. The strategy controls a cache of M block-sized slots. *(Beware, unlike in the rest of this chapter, the cache size is measured in blocks, not items.)* Initially, each slot is either empty or it contains a copy of an arbitrary memory block. Every time the strategy receives a new request, it points to a slot containing the requested block, if there is any. Otherwise the cache *misses* and it has to replace contents of one slot by that block. The obvious measure of cache efficiency is the number C of cache misses. We will call it the *cost* of the strategy for the given access sequence.

The *optimal strategy (OPT)* knows the whole sequence of requests in advance. It is easy to see that the optimal strategy always uses the cache slot, whose block is needed farthest in the future (at best never). Let us denote the cost of the optimal algorithm by C_{OPT}.

We are looking for *online strategies,* which know only the current state of the cache, the current request, and possibly the history of past requests. A typical on-line strategy, which is often used in real caches, is *least-recently used (LRU).* It keeps the cache slots sorted by the time of their last use. When the cache misses, it replaces the slot which is longest unused. The cost of LRU shall be called C_{LRU}.

We would like to prove that LRU is k-competitive, that is $C_{\mathrm{LRU}} \le k \cdot C_{\mathrm{OPT}}$ for some constant k independent of M. Unfortunately, this is impossible to achieve:

Theorem: For every cache size M and every $\varepsilon > 0$, there exists a sequence of requests for which $C_{\mathrm{LRU}} \ge (1-\varepsilon) \cdot M \cdot C_{\mathrm{OPT}}$.

Proof: The sequence will consist of K copies of $1, \ldots, M+1$. The exact value of K will be chosen later.

The number of cache misses on the first M blocks depends on the initial state of the cache. When they are processed, LRU's cache will contain exactly blocks $1, \ldots, M$. The next request for $M+1$ will therefore miss and the least-recently used block 1 will be replaced by $M+1$. The next access will request block 1, which will be again a cache miss, so 2 will be replaced by 1. And so on: all subsequent requests will miss.

We will show a better strategy, which will serve as an upper bound on the optimum strategy. We divide the sequence to *epochs* of size M. Two consecutive epochs overlap in

$M-1$ blocks. Except for the initial epoch, every other epoch can re-use $M-1$ cached blocks from the previous epoch. When it needs to access the remaining block, it replaces the one block, which is not going to be accessed in the current epoch. Thus it has only one cache miss per M requests.

Except for the first epoch, the ratio between C_{LRU} and C_{OPT} is exactly M. The first epoch can decrease this ratio, but its effect can be diminished by increasing K. □

Still, all hope is not lost. If we give LRU an advantage of larger cache, the competitive ratio can be bounded nicely. Let us denote LRU's cache size M_{LRU} and similarly M_{OPT} for the optimal strategy.

Theorem: For every $M_{\mathrm{LRU}} > M_{\mathrm{OPT}} \geq 1$ and every request sequence, we have:

$$C_{\mathrm{LRU}} \leq \frac{M_{\mathrm{LRU}}}{M_{\mathrm{LRU}} - M_{\mathrm{OPT}}} \cdot C_{\mathrm{OPT}} + M_{\mathrm{OPT}}.$$

Proof: We split the request sequence to *epochs* $E_0, \ldots, E_t$ such that the cost of LRU in each epoch is exactly M_{LRU}, except for E_0, where it is at most M_{LRU}.

Now we consider an epoch E_i for $i > 0$. We distinguish two cases:

a) All blocks on which LRU missed are distinct. This means that the access sequence for this epoch contains at least M_{LRU} distinct blocks. OPT could have had at most M_{OPT} of them in its cache when the epoch began, so it still must miss at least $M_{\mathrm{LRU}} - M_{\mathrm{OPT}}$ times.

b) Otherwise, LRU misses twice on the same block b. After the first miss, b was at the head of the LRU list. Before the next miss, it must have been replaced, so it must have moved off the end of the LRU list. This implies that there must have been at least M_{LRU} other blocks accessed in the meantime. Again, OPT must miss at least $M_{\mathrm{LRU}} - M_{\mathrm{OPT}}$ times.

So in every epoch but E_0 the ratio $C_{\mathrm{LRU}}/C_{\mathrm{OPT}}$ is at most $M_{\mathrm{LRU}}/(M_{\mathrm{LRU}} - M_{\mathrm{OPT}})$. Averaging over all epochs gives the desired bound.

Epoch E_0 is different. First assume that both LRU and OPT start with an empty cache. Then all blocks on which LRU misses are distinct, so OPT must miss on them too and the ratio $C_{\mathrm{LRU}}/C_{\mathrm{OPT}}$ is at most 1. If LRU starts with a non-empty cache, the ratio can only decrease — when the block is already cached at the beginning, we save a cache miss, but the contents of the LRU list stay the same. When OPT starts with non-empty cache, it can save up to M_{OPT} misses, which is compensated by the extra M_{OPT} term in the statement of the theorem. □

Corollary: If we set $M_{\mathrm{LRU}} = 2 \cdot M_{\mathrm{OPT}}$, then $C_{\mathrm{LRU}} \leq 2C_{\mathrm{OPT}} + M_{\mathrm{OPT}}$. So on a long enough access sequence, LRU is $(2+\varepsilon)$-competitive.

This comparison may seem unfair: we gave LRU a larger cache, so we are comparing LRU on a cache of size M with OPT on a cache of size $M/2$. However, for all our algorithms the dependency of I/O complexity on the cache size M is such that when we change M by a constant factor, I/O complexity also changes by at most a constant factor. So emulating the optimal cache controller by LRU does not change the asymptotics.

6 Hashing

6.1 Systems of hash functions

Notation:

- The *universe* $\mathcal{U}$. Usually, the universe is a set of integers $\{0, \ldots, U-1\}$, which will be denoted by $[U]$.
- The set of *buckets* $\mathcal{B} = [m]$.
- The set $X \subset \mathcal{U}$ of n items stored in the data structure.
- Hash function $h : \mathcal{U} \to \mathcal{B}$.

Definition: Let $\mathcal{H}$ be a family of functions from $\mathcal{U}$ to $[m]$. We say that the family is *c-universal* for some $c > 0$ if for every pair x, y of distinct elements of $\mathcal{U}$ we have

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{c}{m}.$$

In other words, if we pick a hash function h uniformly at random from $\mathcal{H}$, the probability that x and y collide is at most c-times more than for a completely random function h.

Occasionally, we are not interested in the specific value of c, so we simply say that the family is *universal.*

Note: We will assume that a function can be chosen from the family uniformly at random in constant time. Once chosen, it can be evaluated for any x in constant time. Typically, we define a single parametrized function $h_a(x)$ and let the family $\mathcal{H}$ consist of all h_a for all possible choices of the parameter a. Picking a random $h \in \mathcal{H}$ is therefore implemented by picking a random a. Technically, this makes $\mathcal{H}$ a multi-set, since different values of a may lead to the same function h.

Theorem: Let $\mathcal{H}$ be a c-universal family of functions from $\mathcal{U}$ to $[m]$, $X = \{x_1, \ldots, x_n\} \subseteq \mathcal{U}$ a set of items stored in the data structure, and $y \in \mathcal{U} \setminus X$ another item not stored in the data structure. Then we have

$$\mathbb{E}_{h \in \mathcal{H}}[\#i : h(x_i) = h(y)] \leq \frac{cn}{m}.$$

That is, the expected number of items that collide with y is at most cn/m.

Proof: Let h be a function picked uniformly at random from $\mathcal{H}$. We introduce indicator random variables

$$A_i = \begin{cases} 1 & \text{if } h(x_i) = h(y), \\ 0 & \text{otherwise.} \end{cases}$$

Expectation of zero-one random variables is easy to calculate: $\mathbb{E}[A_i] = 0 \cdot \Pr[A_i = 0] + 1 \cdot \Pr[A_i = 1] = \Pr[A_i = 1] = \Pr[h(x_i) = h(y)]$. Because $\mathcal{H}$ is universal, this probability is at most c/m.

The theorem asks for the expectation of a random variable A, which counts i such that $h(x_i) = h(y)$. This variable is a sum of the indicators A_i. By linearity of expectation, we have $\mathbb{E}[A] = \mathbb{E}[\sum_i A_i] = \sum_i \mathbb{E}[A_i] \le \sum_i c/m = cn/m$. □

Corollary (Complexity of hashing with chaining): Consider a hash table with chaining which uses m buckets and a hash function h picked at random from a c-universal family. Suppose that the hash table contains items $x_1, \ldots, x_n$.

- Unsuccessful search for an item y distinct from all x_i visits all items in the bucket $h(y)$. By the previous theorem, the expected number of such items is at most cn/m.
- Insertion of a new item y to its bucket takes constant time, but we have to verify that the item was not present yet. If it was not, this is equivalent to unsuccessful search.
- In case of a successful search for x_i, all items visited in x_i's bucket were there when x_i was inserted (this assumes that we always add new items at the end of the chain). So the expected time complexity of the search is bounded by the expected time complexity of the previous insert of x_i.
- Unsuccessful insertion (the item is already there) takes the same time as successful search.
- Finally, deletion takes the same time as search (either successful or unsuccessful).

Hence, if the number of buckets m is $\Omega(n)$, expected time complexity of all operations is constant. If we do not know how many items will be inserted, we can resize the hash table and re-hash all items whenever n grows too much. This is similar to the flexible arrays from section 1.3 and likewise we can prove that the amortized cost of resizing is constant.

Definition: Let $\mathcal{H}$ be a family of functions from $\mathcal{U}$ to $[m]$. The family is (k,c)*-independent* for integer k ($1 \le k \le |\mathcal{U}|$) and real $c > 0$ iff for every k-tuple $x_1, \ldots, x_k$ of distinct elements of $\mathcal{U}$ and every k-tuple $a_1, \ldots, a_k$ of buckets in $[m]$, we have

$$\Pr_{h \in \mathcal{H}}[h(x_1) = a_1 \wedge \ldots \wedge h(x_k) = a_k] \le \frac{c}{m^k}.$$

That is, if we pick a hash function h uniformly at random from $\mathcal{H}$, the probability that the given items are mapped to the given buckets is at most c-times more than for a completely random function h.

Sometimes, we do not care about the exact value of c, so we simply say that a family is *k-independent,* if it is (k,c)-independent for some c.

Observation:

1. If $\mathcal{H}$ is (k,c)-independent for $k > 1$, then it is also $(k-1,c)$-independent.
2. If $\mathcal{H}$ is $(2,c)$-independent, then it is c-universal.

Constructions from linear congruence

Definition: For any prime p and $m \le p$, we define the family of linear functions $\mathcal{L} = \{h_{a,b} \mid a,b \in [p]\}$ from $[p]$ to $[m]$, where $h_{a,b}(x) = ((ax+b) \bmod p) \bmod m$.

Theorem: The family $\mathcal{L}$ is 2-universal.

Proof: Let x, y be two distinct numbers in $[p]$. First, we will examine the behavior of linear functions taken modulo p without the final modulo m. For an arbitrary pair $(a,b) \in [p]^2$ of parameters, we define

$$\begin{aligned} r &= (ax+b) \bmod p, \\ s &= (ay+b) \bmod p. \end{aligned}$$

This maps pairs $(a,b) \in [p]^2$ to pairs $(r,s) \in [p]^2$. We can verify that this mapping is bijective: for given (r,s), we can solve the above equations for (a,b). Since integers modulo p form a field, the system of linear equations (which is regular thanks to $x \ne y$) has a unique solution.

So we have a bijection between (a,b) and (r,s). This makes picking (a,b) uniformly at random equivalent to picking (r,s) uniformly at random.

Now, we resurrect the final modulo m. Let us count *bad pairs* (a,b), for which we have $h_{a,b}(x) = h_{a,b}(y)$. These correspond to bad pairs (r,s) such that $r \equiv s$ modulo m. Now we fix r and count s for which (r,s) is bad. If we partition the set $[p]$ to m-tuples, we get $\lceil p/m \rceil$ m-tuples, last of which is incomplete. A complete m-tuple contains exactly one number congruent to r, the final m-tuple one or zero.

Therefore for each r, we have at most $\lceil p/m \rceil \le (p+m-1)/m$ bad pairs. Since $m \le p$, this is at most $2p/m$. Altogether, we have p choices for r, so the number of all bad pairs is at most $2p^2/m$.

As there are p^2 pairs in total, the probability that a pair (r, s) is bad is at most $2/m$. Since there is a bijection between pairs (a, b) and (r, s), the probability of a bad pair (a, b) is the same. The family $\mathcal{L}$ is therefore 2-universal. □

Surprisingly, a slight modification of the family yields even 1-universality.

Definition: For any prime p and $m \le p$, we define the family of linear functions $\mathcal{L}' = \{h_{a,b} \mid a, b \in [p] \wedge a \neq 0\}$ from $[p]$ to $[m]$, where $h_{a,b}(x) = ((ax + b) \bmod p) \bmod m$.

Theorem: The family $\mathcal{L}'$ is 1-universal.

Proof: We shall modify the previous proof. The bijection between pairs (a, b) and (r, s) stays. The requirement that $a \neq 0$ translates to $r \neq s$. We therefore have $p(p - 1)$ pairs (r, s). How many of those are bad?

For a single r, we have one less bad choice of s (in the tuple containing r), so at most $\lceil p/m \rceil - 1 \le (p + m - 1)/m - 1 = (p - 1)/m$ bad choices. For all r together, we have at most $p(p-1)/m$ bad pairs, so the probability that a random pair is bad is at most $1/m$. □

Now, we turn back to the original linear family $\mathcal{L}$.

Theorem: The family $\mathcal{L}$ is $(2, 4)$-independent.

Proof: We fix $x, y \in [p]$ distinct and $i, j \in [m]$. To verify $(2, 4)$-independence, we need to prove that $\Pr[h_{a,b}(x) = i \wedge h_{a,b}(y) = j] \le 4/m^2$ for a random choice of parameters (a, b).

As in the proof of universality of $\mathcal{L}$, we consider the bijection between pairs (a, b) and (r, s). Hence the event $h_{a,b}(x) = i$ is equivalent to $r \equiv i$ modulo m and $h_{a,b}(y) = j$ is the same as $s \equiv j$, again modulo m. As the choices of r and s are independent, we can consider each event separately.

Let us count the values of r which are congruent to i modulo m. If we divide the set $[p]$ to m-tuples, each m-tuple contains at most one such r. As we have $\lceil p/m \rceil$ m-tuples and p possible values of r, the probability that $r \equiv i$ is at most $\lceil p/m \rceil / p \le (p + m - 1)/mp$. As $m \le p$, we have $p + m - 1 \le 2p$, so the probability is at most $2p/mp = 2/m$.

Similarly, the probability that $s \equiv j$ is at most $2/m$. Since the events are independent, the probability of their conjunction is at most $4/m^2$. This is what we needed for $(2, 4)$-independence. □

Composition of function families

The proofs of 2-universality and (2,4)-independence of the family $\mathcal{L}$ have a clear structure: they start with a $(2, 1)$-independent family from a field to the same field. Then they reduce the family's functions modulo m and prove that independence/universality degrades only slightly. This approach can be easily generalized.

Lemma M (composition modulo m): Let $\mathcal{H}$ be a $(2,c)$-independent family of functions from $\mathcal{U}$ to $[r]$ and $m < r$. Then the family $\mathcal{H} \bmod m = \{h \bmod m \mid h \in \mathcal{H}\}$ is $2c$-universal and $(2,4c)$-independent.

Proof: Consider universality first. For two given items $x_1 \neq x_2$, we should show that $\Pr_{h\in\mathcal{H}}[h(x_1) = h(x_2)] \leq 2c/m$. The event $h(x_1) \bmod m = h(x_2) \bmod m$ can be written as a union of disjoint events $h(x_1) = i_1 \wedge h(x_2) = i_2$ over all pairs (i_1, i_2) such that i_1 is congruent to i_2 modulo m. So we have

$$\Pr[h(x_1) = h(x_2)] = \sum_{i_1 \equiv i_2} \Pr[h(x_1) = i_1 \wedge h(x_2) = i_2].$$

By $(2,c)$-independence of $\mathcal{H}$, every term of the sum is at most c/r^2. How many terms are there? We have r choices of i_1, for each of them there are at most $\lceil r/m \rceil \leq (r + m - 1)/m \leq 2r/m$ congruent choices of i_2. Therefore the whole sum is bounded by $r \cdot (2r/m) \cdot (c/r^2) = 2c/m$ as we needed.

For 2-independence, we proceed in a similar way. We are given two items $x_1 \neq x_2$ and two buckets $j_1, j_2 \in [m]$. We are bounding

$$\Pr_h[h(x_1) \bmod m = j_1 \wedge h(x_2) \bmod m = j_2] = \sum_{\substack{i_1 \equiv j_1 \\ i_2 \equiv j_2}} \Pr[h(x_1) = i_1 \wedge h(x_2) = i_2].$$

Again, each term of the sum is at most c/r^2. There are at most $\lceil r/m \rceil \leq (r + m - 1)/m$ choices of i_1 and independently also of i_2. The sum is therefore bounded by

$$\frac{c}{r^2} \cdot \left(\frac{r+m-1}{m}\right)^2 \leq \frac{c}{r^2} \cdot \frac{(2r)^2}{m^2} = \frac{4c}{m^2},$$

which guarantees $(2,4c)$-independence of $\mathcal{H} \bmod m$. □

The previous proof can be extended to general k-independence. However, instead of $4c$, we get $2^k c$, which grows steeply with k. This is unavoidable for m close to r, but if we assume $r \gg m$, we can actually improve the constant even beyond the previous lemma.

Lemma K (k-independent composition modulo m): Let $\mathcal{H}$ be a (k,c)-independent family of functions from $\mathcal{U}$ to $[r]$ and m integer such that $r \geq 2km$. Then the family $\mathcal{H} \bmod m = \{h \bmod m \mid h \in \mathcal{H}\}$ is $(k, 2c)$-independent.

Proof: Let us extend the previous proof. Probability that $h(x_j) = i_j$ for all k values of j is bounded by:

$$\frac{c}{r^k} \cdot \left(\frac{r+m-1}{m}\right)^k = \frac{c}{m^k} \cdot \left(\frac{r+m-1}{r}\right)^k \leq \frac{c}{m^k} \cdot \left(1 + \frac{m}{r}\right)^k .$$

So the family $\mathcal{H}$ mod m is (k, c')-independent for

$$c' = c \cdot \left(1 + \frac{m}{r}\right)^k .$$

Since $\mathrm{e}^x \geq 1 + x$ for all real x, we have $(1 + m/r)^k \leq (\mathrm{e}^{m/r})^k = \mathrm{e}^{mk/r}$. By premise of the lemma, $mk/r \leq 1/2$, so $\mathrm{e}^{mk/r} \leq \mathrm{e}^{1/2} \leq 2$. Hence $c' \leq 2c$. □

Example: Let us test this machinery on the linear family $\mathcal{L}$. The family of all linear functions in a field is $(2, 1)$-independent (by the bijection between (a, b) and (r, s) from the original proofs). Taken modulo m, the family is 2-universal and $(2, 4)$-independent by Lemma **M**. Whenever $p \geq 4m$, it is even $(2, 2)$-independent by Lemma **K**.

The drawback of the reduction modulo m is that we needed a 2-independent family to start with. If it is merely universal, the modulo can destroy universality (exercise 5). However, composing an universal family with a 2-independent family (typically $\mathcal{L}$) yields a 2-independent family.

Lemma G (general composition): Let $\mathcal{F}$ be a c-universal family of functions from $\mathcal{U}$ to $[r]$. Let $\mathcal{G}$ be a $(2, d)$-independent family of functions from $[r]$ to $[m]$. Then the family $\mathcal{H} = \mathcal{F} \circ \mathcal{G} = \{f \circ g \mid f \in \mathcal{F}, g \in \mathcal{G}\}$ is $(2, c')$-independent for $c' = (cm/r + 1)d$.

Proof: Given distinct $x_1, x_2 \in \mathcal{U}$ and $i_1, i_2 \in [m]$, we should bound

$$\mathrm{Pr}_{h \in \mathcal{H}}[h(x_1) = i_1 \wedge h(x_2) = i_2] = \mathrm{Pr}_{f \in \mathcal{F}, g \in \mathcal{G}}\ [g(f(x_1)) = i_1 \wedge g(f(x_2)) = i_2].$$

It is tempting to apply 2-independence of $\mathcal{G}$ on the intermediate results $f(x_1)$ and $f(x_2)$, but unfortunately we cannot be sure that they are distinct. Still, universality of $\mathcal{F}$ should guarantee that they are almost always distinct. We will do it precisely now.

Let M be the *match* event $g(f(x_1)) = i_1 \wedge g(f(x_2)) = i_2$ and C the *collision* event $f(x_1) = f(x_2)$. For a random choice of f and g, we have

$$\begin{aligned}\Pr[M] &= \Pr[M \wedge \neg C] + \Pr[M \wedge C] \\ &= \Pr[M \mid \neg C] \cdot \Pr[\neg C] + \Pr[M \mid C] \cdot \Pr[C].\end{aligned}$$

by elementary probability. (If $\Pr[C]$ or $\Pr[\neg C]$ is 0, the conditional probabilities are not defined, but then the lemma holds trivially.) Each term can be handled separately:

$$\begin{array}{rl} \Pr[M \mid \neg C] \le d/m^2 & \text{by } (2,d)\text{-independence of } \mathcal{G} \\ \Pr[\neg C] \le 1 & \text{trivially} \\ \Pr[M \mid C] \le d/m & \text{for } i_1 \ne i_2\text{: the left-hand side is 0,} \\ & \text{for } i_1 = i_2\text{: } (2,d)\text{-independence implies } (1,d)\text{-independence} \\ \Pr[C] \le c/r & \text{by } c\text{-universality of } \mathcal{F} \end{array}$$

So $\Pr[M] \le d/m^2 + cd/mr$. We want to write this as c'/m^2, so $c' = d + cdm/r = (1 + cm/r)d$. □

Corollary: If $r \ge m$ (the usual case), $\mathcal{F} \circ \mathcal{G}$ is also $(2, c')$-independent for $c' = (c+1)d$.

Polynomial hashing

So far, we constructed k-independent families only for $k = 2$. Families with higher independence can be obtained from polynomials of degree k over a field.

Definition: For any field $\mathbb{Z}_p$ and any $k \ge 1$, we define the family of polynomial hash functions $\mathcal{P}_k = \{h_{\mathbf{t}} \mid \mathbf{t} \in \mathbb{Z}_p^k\}$ from $\mathbb{Z}_p$ to $\mathbb{Z}_p$, where $h_{\mathbf{t}}(x) = \sum_{i=0}^{k-1} t_i x^i$ and the k-tuple $\mathbf{t}$ is indexed from 0.

Lemma: The family $\mathcal{P}$ is $(k, 1)$-independent.

Proof: Let $x_1, \ldots, x_k \in \mathbb{Z}_p$ be distinct items and $a_1, \ldots, a_k \in \mathbb{Z}_p$ buckets. By standard results on polynomials, there is exactly one polynomial h of degree at most k such that $h(x_i) = a_i$ for every i. Hence the probability that a random polynomial of degree at most k has this property is $1/p^k$. □

Of course hashing from a field to the same field is of little use, so we usually consider the family $\mathcal{P}_k \bmod m$ instead. If we set p large enough, Lemma **K** guarantees:

Corollary: If $p \ge 2km$, the family $\mathcal{P}_k \bmod m$ is $(k, 2)$-independent.

The downside is that we need time $\Theta(k)$ both to pick a function from the family and to evaluate it for a single x.

Multiply-shift hashing

Simple and fast hash functions can be obtained from the combination of multiplication with bitwise shifts. On a 32-bit machine, we multiply a 32-bit element x by a random odd number a. We let the machine truncate the result to bottom 32 bits and then we use a bitwise shift by $32 - \ell$ bits to the right, which extracts topmost ℓ bits of the truncated result. Surprisingly, this yields a good hash function from $[2^{32}]$ to $[2^\ell]$.

We provide a more general definition using the bit slice operator: a *bit slice* $x\langle a : b\rangle$ extracts bits at positions a to $b-1$ from the binary expansion of x.[1] Therefore, $x\langle a : b\rangle = \lfloor x/2^a \rfloor \bmod 2^{b-a}$.

Definition: For any w (word length of the machine) and ℓ (width of the result), we define the multiply-shift family $\mathcal{M} = \{h_a \mid a \in [2^w], a \text{ odd}\}$ from $[2^w]$ to $[2^\ell]$, where $h_a(x) = (ax)\langle w-\ell : w\rangle$.

Claim: The family $\mathcal{M}$ is 2-universal.

Obviously, $\mathcal{M}$ is not 2-independent, because $h_a(0) = 0$ for all a. However, 2-independence requires only minor changes: increasing precision from w bits to roughly $w+\ell$ and adding a random number to make all buckets equiprobable.

Definition: Let $\mathcal{U} = [2^w]$ be the universe of w-bit words, ℓ the number of bits of result, and $w' \geq w + \ell - 1$. We define the multiply-shift family $\mathcal{M}' = \{h_{a,b} \mid a, b \in [2^{w'}], a \text{ odd}\}$ from $[2^w]$ to $[2^\ell]$, where $h_{a,b}(x) = (ax+b)\langle w'-\ell : w'\rangle$.

This is still easy to implement: for 32-bit keys and $\ell \leq 32$ bits of result, we can set $w' = 64$, compute multiplication and addition with a 64-bit result and then extract the topmost ℓ bits by shifting to the right by $64-\ell$ bits.

Claim: The family $\mathcal{M}'$ is 2-independent.

Tabulation hashing

To describe a completely random function from $[2^u]$ to $[2^\ell]$, we need a table of 2^u entries of ℓ bits each. This is usually infeasible (and if it were, there would be no reason for hashing), but we can split the u-bit argument to multiple parts, index smaller tables by the parts, and combine the tabulated values to the final result. This leads to a simple construction of hash functions with quite strong properties.

More precisely, let the universe be $\mathcal{U} = [2^{kt}]$ for some $t \geq 1$ (number of parts) and $k \geq 1$ (part size), and $[2^\ell]$ the set of buckets. We generate random tables $T_0, \ldots, T_{t-1}$ of 2^k entries per ℓ bits.

To evaluate the hash function for $x \in \mathcal{U}$, we split x to parts $x\langle ik : (i+1)k\rangle$, where $0 \leq i < t$. We index each table by the corresponding part and we XOR the results:

$$h(x) = \bigoplus_{0 \leq i < t} T_i[\, x\langle ik : (i+1)k\rangle \,].$$

[1] The least-significant bit is at position 0, the bit at position i has weight 2^i.

Tabulation hashing can be considered a family of hash functions, parametrized by the contents of the tables. Picking a function from the family takes time $\Theta(t \cdot 2^k)$ to initialize the tables, evaluating it takes $\Theta(t)$.

Claim: Tabulation hashing is 3-independent, but not 4-independent. (Assuming that it uses at least 2 tables.)

Nevertheless, we will see that tabulation hashing possesses some properties, which generally require higher independence.

Hashing of vectors using scalar products

So far, our universe consisted of simple numbers. We now turn our attention to hashing of more complex data, namely d-tuples of integers. We will usually view the tuples as d-dimensional vectors over some field $\mathbb{Z}_p$.

Similarly to the family of linear functions, we can hash vectors by taking scalar products with a random vector.

Definition: For a prime p and vector size $d \geq 1$, we define the family of scalar product hash functions $\mathcal{S} = \{h_{\mathbf{t}} \mid \mathbf{t} \in \mathbb{Z}_p^d\}$ from $\mathbb{Z}_p^d$ to $\mathbb{Z}_p$, where $h_{\mathbf{t}}(\mathbf{x}) = \mathbf{t} \cdot \mathbf{x}$.

Theorem: The family $\mathcal{S}$ is 1-universal. A function can be picked at random from $\mathcal{S}$ in time $\Theta(d)$ and evaluated in the same time.

Proof: Consider two distinct vectors $\mathbf{x}, \mathbf{y} \in \mathbb{Z}_p^d$. Let k be a coordinate for which $x_k \neq y_k$. As the vector product does not depend on ordering of components, we can renumber the components, so that $k = d$.

For a random choice of the parameter $\mathbf{t}$, we have (in $\mathbb{Z}_p$):

$$\Pr[h_{\mathbf{t}}(\mathbf{x}) = h_{\mathbf{t}}(\mathbf{y})] = \Pr[\mathbf{x} \cdot \mathbf{t} = \mathbf{y} \cdot \mathbf{t}] = \Pr[(\mathbf{x} - \mathbf{y}) \cdot \mathbf{t} = 0] =$$
$$= \Pr\left[\sum_{i=1}^{d} (x_i - y_i)t_i = 0\right] = \Pr\left[(x_d - y_d)t_d = -\sum_{i=1}^{d-1} (x_i - y_i)t_i\right].$$

For every choice of $t_1, \ldots, t_{d-1}$, there exists exactly one value of t_d for which the last equality holds. Therefore it holds with probability $1/p$. □

Note: There is clear intuition behind the last step of the proof: in a field, multiplication of a non-zero number by a uniformly random number yields a uniformly random number; similarly, adding a uniformly random number to any number yields a uniformly random result.

What if the field is too large and we want to reduce the result modulo some m? To use Lemma **M**, we need a 2-independent family, but $\mathcal{S}$ is merely universal.

We can use Lemma **G** and compose $\mathcal{S}$ with the family $\mathcal{L}$ of linear functions from $\mathbb{Z}_p$ to $[m]$. As $\mathcal{L}$ is $(2,4)$-independent, $\mathcal{S} \circ \mathcal{L}$ is $(2,8)$-independent. If $p \geq 4m$, $\mathcal{L}$ is $(2,2)$-independent and $\mathcal{S} \circ \mathcal{L}$ $(2,c')$-independent for $c' = (1+1/4) \cdot 2 = 5/2$.

Alternatively, we can add an additive constant to the scalar product:

Definition: For a prime p, vector size $d \geq 1$, we define the family of scalar product hash functions $\mathcal{S}' = \{h_{\mathbf{t},\beta} \mid \mathbf{t} \in \mathbb{Z}_p^d, \beta \in \mathbb{Z}_p\}$ from $\mathbb{Z}_p^d$ to $\mathbb{Z}_p$, where $h_{\mathbf{t},\beta}(x) = \mathbf{t} \cdot \mathbf{x} + \beta$.

Theorem: If $p \geq 4m$, the family $\mathcal{S}'$ mod m is $(2,2)$-independent. A function can be picked at random from the family in time $\Theta(d)$ and evaluated in the same time.

Proof: By exercise 9, the family $\mathcal{S}'$ is $(2,1)$-independent. By Lemma **K**, $\mathcal{S}'$ mod m is $(2,2)$-independent. □

Rolling hashes from polynomials

Another construction of hashing functions of vectors is based on polynomials, but this time the role of coefficients is played by the components of the vector and the point where the polynomial is evaluated is chosen randomly.

Definition: For a prime p and vector size d, we define the family of polynomial hash functions $\mathcal{R} = \{h_a \mid a \in \mathbb{Z}_p\}$ from $\mathbb{Z}_p^d$ to $\mathbb{Z}_p$, where $h_a(\mathbf{x}) = \sum_{i=0}^{d-1} x_{i+1} \cdot a^i$.

Theorem: The family $\mathcal{R}$ is d-universal. A function can be picked from $\mathcal{R}$ at random in constant time and evaluated for a given vector in $\Theta(d)$ time.

Proof: Consider two vectors $\mathbf{x} \neq \mathbf{y}$ and a hash function h_a chosen at random from $\mathcal{R}$. A collision happens whenever $\sum_i x_{i+1} a^i = \sum_i y_{i+1} a^i$. This is the same condition as $\sum_i (\mathbf{x} - \mathbf{y})_{i+1} a^i = 0$, that is if the number a is a root of the polynomial $\mathbf{x} - \mathbf{y}$. Since a polynomial of degree at most d can have at most d roots (unless it is identically zero), the probability that a is a root is at most d/p. This implies d-universality. □

As usual, we can reduce the range of the function modulo m. However, it is better to compose the family $\mathcal{R}$ with the $(2,4)$-independent family $\mathcal{L}$ of linear functions. Not only we get a 2-independent family as a result, but Lemma **G** guarantees that if p is sufficiently large, the big constant from d-universality disappears. Also, for large p, $\mathcal{L}$ becomes $(2,2)$-independent.

Corollary: Given a prime p and the number of buckets m such that $p \geq 4dm$, the compound family $\mathcal{R} \circ \mathcal{L}$ is $(2, 5/2)$-independent.

Hash functions of this kind play important role in the *Rabin-Karp string search algorithm.* Suppose that we are searching for a d-character substring ν (the needle) in a long text σ (the haystack). We pick a hash function h and calculate the hash $h(\nu)$ of the needle.

Then we slide a window of size d over the haystack and for each position of the window, we hash the contents of the window. Only if the hash of the window is equal to $h(\nu)$, we compare the window with the needle character-by-character.

For this algorithm, we need a hash function which can be recalculated in constant time when the window shifts one position to the right. Such functions are called *rolling hashes* and they are usually chosen from the family $\mathcal{R}$. Compare hashes of the window at positions j and $j+1$ (we read each window from right to left):

$$\begin{aligned} H_j &= h(\sigma_j, \ldots, \sigma_{j+d-1}) = \sigma_j a^{d-1} + \sigma_{j+1} a^{d-2} + \ldots + \sigma_{j+d-1} a^0, \\ H_{j+1} &= h(\sigma_{j+1}, \ldots, \sigma_{j+d}) = \sigma_{j+1} a^{d-1} + \sigma_{j+2} a^{d-2} + \ldots + \sigma_{j+d} a^0, \end{aligned}$$

We can observe that $H_{j+1} = aH_j - \sigma_j a^d + \sigma_{j+d}$, everything calculated in the field $\mathbb{Z}_p$. This is constant-time if we pre-calculate a^d.

Hashing strings

Finally, let us consider hashing of strings of variable length, up to a certain limit L. We will pad the strings to length exactly L by appending blank characters, reducing the problem to hashing of L-tuples.

We have to make sure that a blank does not occur anywhere else in the string — otherwise we get systematic collisions. The obvious choice is to encode the blank as 0, which allows to skip the parts of hash function which deal with blanks altogether.

The L-tuples can be hashed using the family of polynomial-based rolling hash functions. For large L, this is preferred over scalar-product hashing, which requires to generate a random L-component vector of parameters.

Exercises

1. Show that the family of all constant functions from $\mathcal{U}$ to $[m]$ is 1-independent.

2. Show that the family $\mathcal{L}$ is not 1-universal. Find the smallest c such that it is c-universal.

3. What if we modify the definition of $\mathcal{L}$, so that it forces $b = 0$? Is the modified family c-universal for some c? Is it 2-independent?

4. Prove that the family $\mathcal{L}$ is not 3-independent.

5. Show that taking an universal family modulo m sometimes destroys universality. Specifically, show that for each c and $m > 1$, there is a family $\mathcal{H}$ from some $\mathcal{U}$ to the same $\mathcal{U}$ which is 2-universal, but $\mathcal{H}$ mod m is not c-universal.

6. Prove that the family $\mathcal{M}$ is 2-universal.

7.* Prove that the family $\mathcal{M}'$ is $(2, c)$-independent for some c.

8. Prove that tabulation hashing is 3-independent, but not 4-independent (if there are at least two tables).

9. Show that the family $\mathcal{S}'$ is $(2, 1)$-independent.

10. Analyze expected time complexity of the Rabin-Karp algorithm, depending on haystack length, needle length, and the number of buckets.

11. Consider tabulation hashing for d-tuples. How good it is?

6.2 Cuckoo hashing

Sketch

We have two functions f, g mapping the universe $\mathcal{U}$ to $[m]$. Each bucket contains at most one item. An item $x \in \mathcal{U}$ can be located only in buckets $f(x)$ and $g(x)$. Lookup checks only these two buckets, so it takes constant time in the worst case.

Insert looks inside both buckets. If one is empty, we put the new item there. Otherwise we „kick out“ an item from one of these buckets and we replace it by the new item. We place the kicked-out item using the other hash function, which may lead to kicking out another item, and so on. After roughly $\log n$ attempts (this is called insertion timeout), we give up and rehash everything with new choice of f and g.

Theorem: Let $\varepsilon > 0$ be a fixed constant. Suppose that $m \geq (2 + \varepsilon)n$, insertion timeout is set to $\lceil 6 \log n \rceil$, and f, g chosen at random from a $\lceil 6 \log n \rceil$-independent family. Then the expected time complexity of Insert is $\mathcal{O}(1)$, where the constant in $\mathcal{O}$ depends on ε.

Note: Setting the timeout to $\lceil 6 \log m \rceil$ also works.

Note: It is also known that a 6-independent family is not sufficient to guarantee expected constant insertion time, while tabulation hashing (even though it is not 4-independent) is sufficient.

6.3 Linear probing

Open addressing is a form of hashing where each bucket contains at most one item — the items are stored in an array and the hash function produces the index of an array cell where the particular item should be stored. Of course, multiple items can collide in the same cell. So we have to consider alternative locations of items.

In general, we can define a *probe sequence* for each element of the universe. This is a sequence of possible locations of the element in the array in decreasing order of preference. The cells will be numbered from 0 to $m-1$ and indexed modulo m.

Insert follows the probe sequence until it finds an empty cell. Find follows the probe sequence until it either finds a matching item, or it finds an empty cell. Delete is more complicated, since we generally cannot remove items from their cells — a probe sequence for a different item could have been interrupted. Instead, we will replace deleted items with ‘‘tombstones’’ and rebuild the whole table occasionally.

We will study a particularly simple probe sequence called *linear probing*. We fix a hash function h from the universe to $[m]$. The probe sequence for x will be $h(x)$, $h(x)+1$, $h(x)+2$, …, taken modulo m.

Historically, linear probing was considered inferior to other probing methods, because it tends to produce *runs* (a.k.a. *clusters*) — continuous intervals of cells occupied by items. Once a large run forms, it is probable that the next item inserted will be hashed to a cell inside the cluster, extending the cluster even further. On the other hand, linear probing is quite friendly to caches, since it accesses the array sequentially. We will prove that if we use a strong hash function and we keep the table sparse enough (let us say at most half full), the expected size of runs will be constant.

Claim (basic properties of linear probing): Suppose that $m \geq (1+\varepsilon)\cdot n$. Then the expected number of probes during an operation is:

- $\mathcal{O}(1/\varepsilon^2)$ for a completely random hash functions
- $\mathcal{O}(1/\varepsilon^{13/6})$ for hash function chosen from a 5-independent family
- $\Omega(\log n)$ for at least one 4-independent family
- $\Omega(\sqrt{n})$ for at least one 2-independent family
- $\Omega(\log n)$ for multiply-shift hashing
- $\mathcal{O}(1/\varepsilon^2)$ for tabulation hashing

We will prove a slightly weaker version of the first bound. (All restrictions can be removed at the expense of making the technical details more cumbersome.)

Theorem: Let m (table size) be a power of two, $n \leq m/3$ (the number of items), h a completely random hash function, and x an item. Then the expected number of probes when searching for x is bounded by a constant independent of n, m, h, x, and the universe.

The rest of this section is dedicated to the proof of this theorem.

Without loss of generality, we can assume that $n = m/3$, since it clearly maximizes the expected number of probes. We need not worry that m is actually not divisible by 3 — errors introduced by rouding are going to be negligible.

We build a complete binary tree over the table cells. A node at height t corresponds to an interval of size 2^t, whose start is aligned on a multiple of the size. We will call such intervals *blocks.* A block is *critical* if more than $2/3 \cdot 2^t$ items are hashed to it. (We have to distinguish carefully between items *hashed* to a block by the hash function and those really *stored* in it.)

We will calculate probability (over random choice of the hash function) that a given block is critical. We will use the standard tool for estimating tail probabilities — the Chernoff inequality (the proof can be found in most probability theory textbooks).

Theorem (Chernoff bound for the right tail): Let $X_1, \ldots, X_k$ be independent random variables taking values in $\{0, 1\}$. Let further $X = X_1 + \ldots + X_k$, $\mu = \mathbb{E}[X]$, and $c > 1$. Then

$$\Pr[X \geq c\mu] \leq \left(\frac{\mathrm{e}^{c-1}}{c^c} \right)^{\mu}.$$

Lemma: Let B be a block of size 2^t. The probability that B is critical is at most q^{2^t}, where $q = (\mathrm{e}/4)^{1/3} \doteq 0.879$.

Proof: For each item x_i, we define an indicator random variable X_i, which will be 1 if x_i is hashed to the block B. The expectation of an indicator equals the probability of the indicated event, that is $2^t/m$.

$X = X_1 + \ldots + X_n$ counts the number of items hashed to B. By linearity of expectation, $\mu = \mathbb{E}[X] = \sum_i \mathbb{E}[X_i] = n \cdot 2^t/m$. As we assume that $n = m/3$, we have $\mu = 2^t/3$. So if a block is critical, we must have $X \geq 2\mu$. By the Chernoff bound with $c = 2$, this happens with probability at most $(\mathrm{e}/4)^{\mu} = (\mathrm{e}/4)^{2^t/3} = q^{2^t}$. □

Now, we are going to analyze *runs* — maximal intervals of consecutive non-empty cells (indexed modulo m). There is an empty cell before and after each run. Therefore, whenever an item is stored in a run, it must be also hashed there. Our ultimate goal is to estimate size of runs. In order to do this, we prove that a long run must contain at least one large critical block.

Lemma: Let R be a run of size at least $2^{\ell+2}$ and $B_0, \ldots, B_3$ the first 4 blocks of size 2^ℓ intersecting R. Then at least one of these blocks is critical.

Proof: The size of R is at least 4 times 2^ℓ, but R is generally not aligned on a start of a block. So R intersects between 4 and 5 blocks of size 2^ℓ. The first block B_0 contains at least one cell of R, blocks B_1 to B_3 contain 2^ℓ cells of R each. So the interval $L = R \cap (B_0 \cup \ldots \cup B_3)$ must contain at least $1 + 3 \cdot 2^\ell$ cells, all of them non-empty. Since there is an empty cell before L, each item stored in L is also hashed there.

We will show that if no block B_i is critical, there cannot be so many items hashed to L. Each non-critical block of size 2^ℓ has at most $2/3 \cdot 2^\ell$ items hashed to it, so our 4 blocks can receive at most $8/3 \cdot 2^\ell < 3 \cdot 2^\ell$ items. □

We consider the situation when we search for an item x. We start probing at position $h(x)$, so the number of probes is at most the size of the run containing the cell $h(x)$.

Lemma: Let R be a run containing $h(x)$ and $|R| \in [2^{\ell+2}, 2^{\ell+3})$. Then at least one of the following 12 blocks of size 2^ℓ is critical: the block containing $h(x)$, 8 blocks preceding it, and 3 blocks following it.

Proof: The run R intersects between 4 and 9 blocks of the given size, so it begins at most 8 blocks before the block containing $h(x)$. By the previous lemma, one of the first 4 blocks of R is critical. □

Hence, the probability that run length lies between $2^{\ell+2}$ and $2^{\ell+3}$ is bounded by the probability that one of the 12 blocks (chosen independently of the actual length of the run) is critical. By union bound and our estimate on the probability that a block is critical, we get:

Corollary: Let R be a run containing $h(x)$. The probability that $|R| \in [2^{\ell+2}, 2^{\ell+3})$ is at most $12 \cdot (\mathrm{e}/4)^{2^\ell/3} = 12 \cdot q^{2^\ell}$, where q is the constant introduced above.

Finally, we will use this to prove that the expected size of R is at most a constant. We cover all possible sizes by intervals $[2^{\ell+2}, 2^{\ell+3})$ together with an exceptional interval $[0, 3]$. We replace each size by the upper bound of its interval, which only increases the expectation. So we get:

$$\mathbb{E}[|R|] \le 3 \cdot \Pr[|R| \le 3] + \sum_{\ell \ge 0} 2^{\ell+3} \cdot \Pr[|R| \in [2^{\ell+2}, 2^{\ell+3}]].$$

We bound the former probability simply by 1 and use the previous corrolary to bound the latter probability:

$$\mathbb{E}[|R|] \le 3 + \sum_{\ell \ge 0} 2^{\ell+3} \cdot 12 \cdot q^{2^\ell} = 3 + 8 \cdot 12 \cdot \sum_{\ell \ge 0} 2^\ell \cdot q^{2^\ell} \le 3 + 96 \cdot \sum_{i \ge 1} i \cdot q^i.$$

Since the last sum converges for an arbitrary $q \in (0, 1)$, the expectation of $|R|$ is at most a constant. This concludes the proof of the theorem. □

Exercises

1. Show how to implement Delete without tombstones for linear probing.

6.4 Bloom filters

Bloom filters are a family of data structures for approximate representation of sets in a small amount of memory. A Bloom filter starts with an empty set. Then it supports insertion of new elements and membership queries. Sometimes, the filter gives a *false positive* answer: it answers yes even though the element is not in the set. We will calculate the probability of false positives and decrease it at the expense of making the structure slightly larger. False negatives will never occur.

A trivial example

We start with a very simple filter. Let h be a hash function from a universe $\mathcal{U}$ to $[m]$, picked at random from a c-universal family. For simplicity, we will assume that $c = 1$. The output of the hash function will serve as an index to an array $B[0 \ldots m-1]$ of bits.

At the beginning, all bits of the array are zero. When we insert an element x, we simply set the bit $B[h(x)]$ to 1. A query for x tests the bit $B[h(x)]$ and answers yes iff the bit is set to 1. (We can imagine that we are hashing items to m buckets, but we store only which buckets are non-empty.)

Suppose that we have already inserted items $x_1, \ldots, x_n$. If we query the filter for any x_i, it always answers yes. But if we ask for a y different from all x_i's, we can get a false positive answer if x falls to the same bucket as one of the x_i's.

Let us calculate the probability of a false positive answer. For a specific i, we have $\Pr_h[h(y) = h(x_i)] \le 1/m$ by 1-universality. By union bound, the probability that $h(y) = h(x_i)$ for some i is at most n/m.

We can ask an inverse question, too: how large filter do we need to push error probability under some $\varepsilon > 0$? By our calculation, $\lceil n/\varepsilon \rceil$ bits suffice. It is interesting that this size does not depend on the size of the universe — all previous data structures required at least $\log |\mathcal{U}|$ bits per item. On the other hand, the size scales badly with error probability: for example, a filter for 10^6 items with $\varepsilon = 0.01$ requires 100 Mb.

Multi-band filters

To achieve the same error probability in smaller space, we can simply run multiple filters in parallel. We choose k hash functions $h_1, \ldots, h_k$, where h_i maps the universe to a separate array B_i of m bits. Each pair (B_i, h_i) is called a *band* of the filter.

Insertion adds the new item to all bands. A query asks all bands and it answers YES only if each band answered YES.

We shall calculate error probability of the k-band filter. Suppose that we set $m = 2n$, so that each band gives a false positive with probability at most $1/2$. The whole filter gives a false positive only if all bands did, which happens with probability at most 2^{-k} if the functons $h_1, \ldots, h_k$ where chosen independently. This proves the following theorem.

Theorem: Let $\varepsilon > 0$ be the desired error probability and n the maximum number of items in the set. The k-band Bloom filter with $m = 2n$ and $k = \lceil \log(1/\varepsilon) \rceil$ gives false positives with probability at most ε. It requires $2n\lceil \log(1/\varepsilon) \rceil$ bits of memory and both INSERT and LOOKUP run in time $\mathcal{O}(k)$.

In the example with $n = 10^6$ and $\varepsilon = 0.01$, we get $m = 2 \cdot 10^6$ and $k = 7$, so the whole filter requires 14 Mb. If we decrease ε to 0.001, we have to increase k only to 10, so the memory consumption reaches only 20 Mb.

Optimizing parameters

The multi-band filter works well, but it turns out that we can fine-tune its parameters to improve memory consumption by a constant factor. We can view it as an optimization problem: given a memory budget of M bits, set the parameters m and k such that the filter fits in memory ($mk \le M$) and the error probability is minimized. We will assume that all hash functions are perfectly random.

Let us focus on a single band first. If we select its size m, we can easily calculate probability that a given bit is zero. We have n items, each of them hashed to this bit with probability $1/m$. So the bit remains zero with probability $(1 - 1/m)^n$. This is approximately $p = \mathrm{e}^{-n/m}$.

We will show that if we set p, all other parameters are uniquely determined and so is the probability of false positives. We will find p such that this probability is minimized.

If we set p, it follows that $m \approx -n/\ln p$. Since all bands must fit in M bits of memory, we want to use $k = \lfloor M/m \rfloor \approx -M/n \cdot \ln p$ bands. False positives occur if we find 1 in all bands, which has probability

$$(1-p)^k = \mathrm{e}^{k \ln(1-p)} \approx \mathrm{e}^{-M/n \cdot \ln p \cdot \ln(1-p)}.$$

As e^{-x} is a decreasing function, it suffices to maximize $\ln p \cdot \ln(1-p)$ for $p \in (0,1)$. By elementary calculus, the maximum is attained for $p = 1/2$. This leads to false positive probability $(1/2)^k = 2^{-k}$.

Let us return back to the original question: given n and ε, how large a filter do we need? We have to push 2^{-k} under ε, so we set $k = \lceil \log(1/\varepsilon) \rceil$, and $M = kn/\ln 2 \approx$

$n \cdot \log(1/\varepsilon) \cdot (1/\ln 2) \doteq n \cdot \log(1/\varepsilon) \cdot 1.44$. This improves the constant from the previous theorem from 2 to circa 1.44.

Note: It is known that any approximate membership data structure with false positive probability ε and no false negatives must use at least $n \log(1/\varepsilon)$ bits of memory. The optimized Bloom filter is therefore within a factor of 1.44 from the optimum.

Single-table filters

It is also possible to construct a Bloom filter, where multiple hash functions point to bits in a shared table. (In fact, this was the original construction by Bloom.) Consider k hash functions $h_1, \ldots, h_k$ mapping the universe to $[m]$ and a bit array $B[0, \ldots, m-1]$. Insert(x) sets the bits $B[h_1(x)], \ldots, B[h_k(x)]$ to 1. Lookup(x) returns yes, if all these bits are set.

This filter can be analysed similarly to the k-band version. We will assume that all hash functions are perfectly random and mutually independent.

Insertion of n elements sets kn bits (not necessarily distinct), so the probability that a fixed bit $B[i]$ is unset is $(1-1/m)^{nk}$, which is approximately $p = \mathrm{e}^{-nk/m}$. We will find the optimum value of p, for which the probability of false positives is minimized. For fixed m, we get $k = -m/n \cdot \ln p$.

We get a false positive if all bits $B[h_i(x)]$ are set. This happens with probability approximately[2] $(1-p)^k = (1-p)^{-m/n \cdot \ln p} = \exp(-m/n \cdot \ln p \cdot \ln(1-p))$. Again, this is minimized for $p = 1/2$. So for a fixed error probability ε, we get $k = \lceil \log(1/\varepsilon) \rceil$ and $m = kn/\ln 2 \doteq 1.44 \cdot n \cdot \lceil \log(1/\varepsilon) \rceil$.

We see that as far as our approximation can tell, single-table Bloom filters achieve the same performance as the k-band version.

Counting filters

An ordinary Bloom filter does not support deletion: when we delete an item, we do not know if some of its bits are shared with other items. There is an easy solution: instead of bits, keep b-bit counters $C[0 \ldots m-1]$. Insert increments the counters, Delete decrements them, and Lookup returns yes if all counters are non-zero.

However, since the counters have limited range, they can overflow. We will handle overflows by keeping the counter at the maximum allowed value $2^b - 1$, which will not be

[2] We are cheating a little bit here: the events $B[i] = 1$ for different i are not mutually independent. However, further analysis shows that they are very little correlated, so our approximation holds.

changed by subsequent insertions nor deletions. We say that the counter is *stuck.* Obviously, too many stuck counters will degrade the data structure. We will show that this happens with small probability only.

We will assume a single-band filter with one fully random hash function and m counters after insertion of n items. We will assume no deletions, which is the worst case. For a fixed counter value t, we have

$$\Pr[C[i] = t] = \binom{n}{t} \cdot \left(\frac{1}{m}\right)^t \cdot \left(1 - \frac{1}{m}\right)^{n-t},$$

because for each of $\binom{n}{t}$ t-tuples we have probability $(1/m)^t$ that the tuple is hashed to i and probability $(1 - 1/m)^{n-t}$ that all other items are hashed elsewhere. If $C[i] \geq t$, there must exist a t-tuple hashed to i and the remaining items can be hashed anywhere. Therefore:

$$\Pr[C[i] \geq t] \leq \binom{n}{t} \cdot \left(\frac{1}{m}\right)^t.$$

Since $\binom{n}{t} \leq (n\mathrm{e}/t)^t$, we have

$$\Pr[C[i] \geq t] \leq \left(\frac{n\mathrm{e}}{t}\right)^t \cdot \left(\frac{1}{m}\right)^t = \left(\frac{n\mathrm{e}}{mt}\right)^t.$$

As we already know that the optimum m is approximately $n/\ln 2$, the probability is at most $(\mathrm{e}\ln 2/t)^t$. By union bound, the probability that there exists a stuck counter is at most m-times more.

Example: A 4-bit counter is stuck when it reaches $t = 15$, which by our bound happens with probability at most $3.06 \cdot 10^{-14}$. If we have $m = 10^9$ counters, the probability that any is stuck is at most $3.06 \cdot 10^{-5}$. So for any reasonably large table, 4-bit counters are sufficient and they seldom get stuck. Of course, for a very long sequence of operations, stuck counters eventually accumulate, so we should preferably rebuild the structure occasionally.

7 Geometric data structures

There is a rich landscape of data structures for handling geometric objects. Let us perform a small survey. First, there are different objects which can be stored in the structure: points in $\mathbb{R}^d$, lines in $\mathbb{R}^d$, or even more complex shapes like polygons, polytopes, conic sections or splines.

Queries are asked about all objects which lie within a given *region* (or intersect its boundary). Usually, the region is one of:

- *a single object* — test if the given object is stored in the structure
- *a range* — a d-dimensional "box" $[a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_d, b_d]$; some of the intervals can extend to infinity
- *partial match* — some parameters of the object are specified precisely, the remaining ones can be arbitrary. E.g., report all points in $\mathbb{R}^3$ for which $x = 3$ and $z = 5$. This is a special case of a range, where each interval is either a single point or the whole $\mathbb{R}$. Partial match queries frequently occur in database indices, even in otherwise non-geometric situations.
- *a polygon*

We might want to *enumerate* all objects within the region, or just *count* them without enumerating them one by one. If there is a value associated with each object, we can ask for a sum or a maximum of values of all objects within the range — this is generally called an *aggregation* query.

In this chapter, we will consider the simple case of range queries on points in $\mathbb{R}^d$.

7.1 Range queries in one dimension

We show the basic techniques on one-dimensional range queries. The simplest solution is to use a sorted array. Whenever we are given a query interval $[a, b]$, we can locate its endpoints in the array by binary search. Then we either enumerate the items between the endpoints or we count them by subtracting indices of the endpoints. We only have to be careful and check whether the endpoints lies at an item or in a gap between items. Overall, we can answer any query in $\mathcal{O}(\log n + p)$, where n is the number of items and p the number of points enumerated. The structure can be built in time $\mathcal{O}(n \log n)$ by sorting the items and it is stored in $\mathcal{O}(n)$ space.

Another solution uses binary search trees. It is more complicated, but more flexible. It can be made dynamic and it can also answer aggregation queries.

Definition: Let T be a binary search tree with real-valued keys. For each node v, we define the set $\mathrm{int}(v)$ called the *interval of* v. It contains all real numbers whose search visits v.

Observation: We can see that the sets have the following properties:

- $\mathrm{int}(root)$ is the whole set $\mathbb{R}$.
- If v is a node with key $\mathrm{key}(v)$, left child $\ell(v)$ and right child $r(v)$, then:
  - $\mathrm{int}(\ell(v)) = \mathrm{int}(v) \cap (-\infty, \mathrm{key}(v))$
  - $\mathrm{int}(r(v)) = \mathrm{int}(v) \cap (\mathrm{key}(v), +\infty)$
- By induction, $\mathrm{int}(v)$ is always an open interval.
- All keys in the subtree of v lie in $\mathrm{int}(v)$.
- The definition of $\mathrm{int}(v)$ applies to external nodes, too. The intervals obtained by cutting the real line to parts at the keys in internal nodes are exactly the intervals assigned to external nodes.

This suggests a straightforward recursive algorithm for answering range queries.

```
Procedure RangeQuery($v, Q$)
Input: a root of a subtree $v$, query range $Q$
  1. If $v$ is external, return.
  2. If $\mathrm{int}(v) \subseteq Q$, report the whole subtree rooted at $v$ and return.
  3. If $\mathrm{key}(v) \in Q$, report the item at $v$.
  4. $Q_\ell \leftarrow Q \cap \mathrm{int}(\ell(v))$, $Q_r \leftarrow Q \cap \mathrm{int}(r(v))$
  5. If $Q_\ell \neq \emptyset$: call RangeQuery($\ell(v), Q_\ell$)
  6. If $Q_r \neq \emptyset$: call RangeQuery($r(v), Q_r$)
```

Let us analyze time complexity of this algorithm now.

Lemma: If the tree is balanced, RangeQuery called on its root visits $\mathcal{O}(\log n)$ nodes and subtrees.

Proof: Let $Q = [\alpha, \beta]$ be the query interval. Let a and b be tree nodes (internal or external) where search for α and β ends. We denote the lowest common ancestor of a and b by p.

Whenever we enter a node v with some interval $\mathrm{int}(v)$, the key $\mathrm{key}(v)$ splits the interval to two parts, corresponding to $\mathrm{int}(\ell(v))$ and $\mathrm{int}(r(v))$.

On the path from the root to p, Q always lies in one of these parts and we recurse on one child. In some cases, the current key lies in Q, so we report it.

When we enter the common ancestor p, the range Q lives in both parts, so we report key(p) and recurse on both parts.

On the ‘‘left path’’ from p to a, we encounter two situations. Either Q lies solely in the right part, so we recurse on it. Or Q crosses key(v), so we recurse on the left part and report the whole right subtree. Again, we report key(v) if it lies in Q.

The ‘‘right path’’ from p to b behaves symetrically: we recurse on the right part and possibly report the whole left subtree and/or the current key.

Since all paths contain $\mathcal{O}(\log n)$ nodes together, we visit $\mathcal{O}(\log n)$ nodes and report $\mathcal{O}(\log n)$ nodes and $\mathcal{O}(\log n)$ subtrees. □

Corollary: An enumeration query is answered in time $\mathcal{O}(\log n + p)$, where p is the number of items reported. If we precompute sizes of all subtrees, a counting query takes $\mathcal{O}(\log n)$ time. Aggregate queries can be answered if we precompute aggregate answers for all subtrees and combine them later. The structure can be built in $\mathcal{O}(n \log n)$ time using the algorithm for constructing perfectly balanced trees. It takes $\mathcal{O}(n)$ memory.

This query algorithm is compatible with most methods for balancing binary search trees. The interval int(v) need not be stored in the nodes — it can be computed on the fly when traversing the tree from the root. The subtree sizes or aggregate answers can be easily updated when rotating an edge: only the values in the endpoints of the edge are affected and they can be recomputed in constant time from values in their children. This way we can obtain a dynamic range tree with Insert and Delete in $\mathcal{O}(\log n)$ time.

7.2 Multi-dimensional search trees (k-d trees)

Binary search trees can be extended to k-dimensional trees, or simply k-d trees. Keys stored in nodes are points in $\mathbb{R}^k$, nodes at the ℓ-th level compare the $(\ell \bmod k)$-th coordinate. We will temporarily assume that no two points share the same value of any coordinate, so every point different from the node's key belongs either to the left subtree, or the right one.

Let us show a static 2-dimensional example. The root compares the x coordinate, so it splits the plane by a vertical line to two half-planes. The children of the root split their half-planes by horizontal lines to quarter-planes, and so on.

We can build a perfectly balanced 2-d tree recursively. We place the point with median x coordinate in the root. We split the points to the two half-planes and recurse on each half-plane, which constructs the left and right subtree. During the recursion, we alternate coordinates. As the number of points in the current subproblem decreases by a factor of two in every recursive call, we obtain a tree of height $\lceil \log n \rceil$.

Time complexity of building can be analyzed using the recursion tree: since we can find a median of m items in time $\mathcal{O}(m)$, we spend $\mathcal{O}(n)$ time per tree level. We have $\mathcal{O}(\log n)$ levels, so the whole construction runs in $\mathcal{O}(n \log n)$ time. Since every node of the 2-d tree contains a different point, the whole tree consumes $\mathcal{O}(n)$ space.

We can answer 2-d range queries similarly to the 1-d case. To each node v of the tree, we can assign a 2-d interval $\mathrm{int}(v)$ recursively. This again generates a hierarchy of nested intervals, so the RangeQuery algorithm works there, too. However, 2-d range queries can be very slow in the worst case:

Lemma: Worst-case time complexity of range queries in a 2-d tree is $\Omega(\sqrt{n})$.

Proof: Consider a tree built for the set of points $\{(i,i) \mid 1 \le i \le n\}$ for $n = 2^t - 1$. It is a complete binary tree with t levels. Let us observe what happens when we query a range $\{0\} \times \mathbb{R}$ (unbounded intervals are not necessary for our construction, a very narrow and very high box would work, too).

On levels where the x coordinate is compared, we always go to the left. On levels comparing y, both subtrees lie in the query range, so we recurse on both of them. This means that the number of visited nodes doubles at every other level, so at level t we visit $2^{t/2} \approx \sqrt{n}$ nodes. □

Note: This is the worst which can happen, so query complexity is $\Theta(\sqrt{n})$. General k-d trees for arbitrary k can be built in time $\mathcal{O}(n \log n)$, they require space $\mathcal{O}(n)$, and they answer range queries in time $\mathcal{O}(n^{1-1/k})$[1] This is quite bad in high dimension, but there is a matching lower bound for structures which fit in linear space.

Repeated values of coordinates can be handled by allowing a third child of every node, which is the root of a $(k-1)$-d subtree. There we store all points which share the compared coordinate with the parent's key. This extension does not influence asymptotic complexity of operations.

Dynamization is non-trivial and we will not show it.

[1] Constants hidden in the $\mathcal{O}$ depend on the dimension.

7.3 Multi-dimensional range trees

The k-dimensional search trees were simple, but slow in the worst case. There is a more efficient data structure: the *multi-dimensional range tree,* which has poly-logarithmic query complexity, if we are willing to use super-linear space.

2-dimensional range trees

For simplicity, we start with a static 2-dimensional version and we will assume that no two points have the same x coordinate.

The 2-d range tree will consist of multiple instances of a 1-d range tree, which we built in section 7.1 — it can be a binary search tree with range queries, but in the static case even a sorted array suffices.

First we create an x-tree, which is a 1-d range tree over the x coordinates of all points stored in the structure. Each node contains a single point. Its subtree corresponds to an open interval of x coordinates, that is a *band* in $\mathbb{R}^2$ (an open rectangle which is vertically unbounded). For every band, we construct a y-tree containing all points in the band ordered by the y coordinate.

If the x-tree is balanced, every node lies in $\mathcal{O}(\log n)$ subtrees. So every point lies in $\mathcal{O}(\log n)$ bands and the whole structure takes $\mathcal{O}(n \log n)$ space: $\mathcal{O}(n)$ for the x-tree, $\mathcal{O}(n \log n)$ for all y-trees.

We can build the 2-d tree recursively. First we create two lists of points: one sorted by the x coordinate, one by y. Then we construct the x-tree. We find the point with median x coordinate in constant time. This point becomes the root of the x-tree. We recursively construct the left subtree from points with less than median x coordinate — we can construct the corresponding sub-lists of both the x and y list in $\mathcal{O}(n)$ time. We construct the right subtree similarly. Finally, we build the y-tree for the root: it contains all the points and we can build it from the y-sorted array in $\mathcal{O}(n)$ time.

The whole building algorithm requires linear time per sub-problem, which sums to $\mathcal{O}(n)$ over one level of the x-tree. Since the x-tree is logarithmically high, it makes $\mathcal{O}(n \log n)$ for the whole construction.

Now we describe how to answer a range query for $[x_1, x_2] \times [y_1, y_2]$. First we let the x-tree answer a range query for $[x_1, x_2]$. This gives us a union of $\mathcal{O}(\log n)$ points and bands which disjointly cover $[x_1, x_2]$. For each point, we test if its y coordinate lies in $[y_1, y_2]$. For each band, we ask the corresponding y-tree for points in the range $[y_1, y_2]$. We spend $\mathcal{O}(\log n)$ time in the x-tree, $\mathcal{O}(\log n)$ time to process the individual points, $\mathcal{O}(\log n)$ in each y-tree, and $\mathcal{O}(1)$ per point reported. Put together, this is $\mathcal{O}(\log^2 n + p)$ if p points are reported ($p = 0$ for a counting query).

Handling repeated coordinates

We left aside the case of multiple points with the same x coordinate. This can be handled by attaching another y-tree to each x-tree node, which contains nodes sharing the same x coordinate. That is, x-tree nodes correspond to distinct x coordinates and each has two y-trees: one for its own x coordinate, one for the open interval of x coordinates covering its subtree. This way, we can perform range queries for both open and closed ranges.

Time complexity of BUILD stays asymptotically the same: the maximum number of y-trees containing a given point increases twice, so it is still $\mathcal{O}(\log n)$. Similarly for range queries: we query at most twice as much y-trees.

Multi-dimensional generalization

The 2-d range tree is easily generalized to multiple dimensions. We build a 1-d tree on the first coordinate (usually called the *primary tree*). Each node defines a hyperplane of all points sharing the given first coordinate and a d-dimensional band (the Cartesian product of an open interval and $\mathbb{R}^{d-1}$). For each hyperplane and each point, we build a secondary tree, which is a $(d-1)$-dimensional range tree.

Since every point lies in $\mathcal{O}(\log n)$ secondary trees, memory consumption is $O(\log n)$ times larger than for a $(d-1)$-dimensional tree. Therefore the d-dimensional tree requires $\mathcal{O}(n \log^{d-1} n)$ space, where the constant hidden in the $\mathcal{O}$ depends on d.

The data structure can be built by induction on dimension. We keep all points in d separate list, each sorted by a different coordinate. We build the primary tree as a perfectly balanced tree. In each node, we construct the $(d-1)$-dimensional secondary trees recursively. Since each point participates in $\mathcal{O}(\log n)$ secondary trees, each additional dimension multiplies time complexity by $\mathcal{O}(\log n)$. So a d-dimensional tree can be built in $\mathcal{O}(n \log^{d-1} n)$ time.

Range queries are again performed by recursion on dimension. Given a d-dimensional range $[a_1, b_1] \times \ldots \times [a_d, b_d]$, we first query the primary tree for the range $[a_1, b_1]$. This results in $\mathcal{O}(\log n)$ nodes and subtrees, whose secondary trees we ask for $[a_2, b_2] \times \ldots \times [a_d, b_d]$. Again, each additional dimensions slows queries down by a factor of $\mathcal{O}(\log n)$. Therefore a d-dimensional query requires $\mathcal{O}(\log^d n)$ time.

Dynamization

What if we want to make d-dimensional range trees dynamic? As usual, let us consider the 2-d case first. We could try using one of the many dynamic balanced binary search trees for 1-d trees. Alas, this is doomed to fail: if we rotate an edge in the x-tree, almost all points can change their bands, so in the worst case we need $\Theta(n)$ time to rebuild the corresponding y-trees.

However, rebuilding of subtrees comes relatively cheap: we can rebuild a single y-tree in linear time (we can collect the points in sorted order by recursion) and a m-node subtree of the x-tree in $\mathcal{O}(m \log m)$ time (we collect points in x order from the x-tree, in y order from the topmost y-tree and re-run Build).

We will use local rebuilds to keep the whole 2-d tree (and all its constituent 1-d trees) balanced. We will simply use the 1-d lazy weight-balanced trees of section 1.4 for all x-trees and y-trees.

Suppose that we want to Insert a new point. This causes up to one insertion to the x-tree (if the x coordinate was not used yet) and $\mathcal{O}(\log n)$ insertions to the y-trees. Every insertion to a y-tree can cause rebalancing of that tree, which costs $\mathcal{O}(\log n)$ amortized per tree, so $\mathcal{O}(\log^2 n)$ altogether. Insertion to the x-tree can cause rebalancing of the x-tree, which is more costly since rebuilding a subtree requires $\mathcal{O}(n \log n)$ time instead of $\mathcal{O}(n)$. So we have to spend $\mathcal{O}(\log^2 n)$ amortized to pre-pay this cost.

We have therefore shown that saving $\mathcal{O}(\log^2 n)$ time per Insert pays for all rebalancing of the constituent 1-d trees. The same argument applies to Delete. The whole argument can be extended by induction to d-d trees.

Let us summarize time complexity of all operations:

Build	$\mathcal{O}(n \log^{d-1} n)$
RangeQuery	$\mathcal{O}(\log^d n + p)$ for p reported points
Insert	$\mathcal{O}(\log^d n)$ amortized
Delete	$\mathcal{O}(\log^d n)$ amortized

The whole structure fits in $\mathcal{O}(n \log^{d-1} n)$ words of memory.

Fractional cascading

Time complexity of 2-d queries can be improved to $\mathcal{O}(\log n + p)$ by precomputing a little bit more. This translates to $\mathcal{O}(\log^{d-1} n + p)$ time per query in d-d case. We will show a static version, leaving dynamization as an exercise.

In the static case, y-trees can be simply sorted arrays. If we want to search for the range $[x_1, x_2] \times [y_1, y_2]$, we search the x-tree for $[x_1, x_2]$, which tells us to binary search for $[y_1, y_2]$ in $\mathcal{O}(\log n)$ sorted arrays. Each binary search per se requires $\mathcal{O}(\log n)$ time, but we can observe that these searches are somewhat correlated: each list is a sub-list of the list belonging to the parent node in the x-tree.

Whenever we have an array A in some node and its sub-array B in a child node, we precompute auxiliary pointers from A to B. For each element $a \in A$, we store its *predecessor* in B: the index of the largest element $b \in B$ such that $b \leq a$. If a itself is in B, the predecessor of a is simply its occurrence in b; otherwise, it is the next smaller element. To make this always defined, we will place $-\infty$ at the beginning of every array. It is obvious that the predecessors can be computed in time linear with the length of A and B, so their construction does not slow down BUILD asymptotically.

Now, consider a query for $[x_1, x_2] \times [y_1, y_2]$. We walk the x-tree from the root and visit nodes and/or bands which together cover $[x_1, x_2]$. For each visited node, we calculate the indices in the node's y-array which correspond to the boundary of $[y_1, y_2]$ (if the boundary values are not present, we take the next value towards the interior of the interval).

As we know the positions of the boundary in the parent array, we can use the precomputed predecessor pointers to update the positions to the child array in constant time. We simply use the predecessor pointers and shift the index by one position in the right direction if needed.

We therefore need $\mathcal{O}(\log n)$ for the initial binary search in the root y-array and $\mathcal{O}(\log n)$ steps per $\mathcal{O}(1)$ time in the subsequent y-arrays. This is $\mathcal{O}(\log n)$ altogether.

This trick is an instance of a more general technique called *fractional cascading,* which is generally used to find a given value in a collection of sorted lists, which need not be sub-lists of one another.

Exercises

1.* Show how to dynamize 2-d range trees with fractional cascading.

8 Strings

Notation:

- Σ is an *alphabet* – a finite set of *characters*.
- Σ^* is the set of all *strings* (finite sequences) over Σ.
- We will use Greek letters for string variables, Latin letters for character and numeric variables, and `typewriter` letters for concrete characters. We will make no difference between a character and a single-character string.
- $|\alpha|$ is the *length* of the string α.
- ε is the *empty string* — the only string of length 0.
- $\alpha\beta$ is the *concatenation* of strings α and β; we have $\alpha\varepsilon = \varepsilon\alpha = \alpha$ for all α.
- $\alpha[i]$ is the i-th character of the string α; characters are indexed starting with 0.
- $\alpha[i : j]$ is the *substring* $\alpha[i]\alpha[i+1]\ldots\alpha[j-1]$; note that $\alpha[j]$ is the first character *behind* the substring, so we have $|\alpha[i : j]| = j - i$. If $i \geq j$, the substring is empty. Either i or j can be omitted, the beginning or the end of α is used instead.
- $\alpha[\,:j]$ is the *prefix* of α formed by the first j characters. A word of length n has $n+1$ prefixes, one of them being the empty string.
- $\alpha[i :\,]$ is the *suffix* of α from character number i to the end. A word of length n has $n+1$ suffixes, one of them being the empty string.
- $\alpha[\,:\,] = \alpha$.
- $\alpha \leq \beta$ denotes *lexicographic order* of strings: $\alpha \leq \beta$ if α is a prefix of β or if there exists k such that $\alpha[k] < \beta[k]$ and $\alpha[\,:k] = \beta[\,:k]$.

8.1 Suffix arrays

Definition: The *suffix array* for a string α of length n is a permutation S of the set $\{0,\ldots,n\}$ such that $\alpha[S[i]:\,] < \alpha[S[i+1]:\,]$ for all $0 \leq i < n$.

Claim: The suffix array can be constructed in time $\mathcal{O}(n)$.

Before we prove the claim, let us convince you that suffix arrays are a very powerful tool for solving string problems.

Once we have the suffix array for a string α, we can easily locate all occurrences of a given substring β in α. Each occurrence corresponds to a suffix of α whose prefix is β. In the lexicographic order of all suffixes, these suffixes form a range. We can easily find the start and end of this range using binary search on the suffix array. We need $\mathcal{O}(\log|\alpha|)$ steps,

i	$S[i]$	$R[i]$	$L[i]$	*suffix*
0	14	3	0	ε
1	8	11	3	`ananas`
2	10	10	2	`anas`
3	0	7	2	`annbansbananas`
4	4	4	1	`ansbananas`
5	12	12	0	`as`
6	7	14	3	`bananas`
7	3	6	0	`bansbananas`
8	9	1	2	`nanas`
9	11	8	1	`nas`
10	2	2	1	`nbansbananas`
11	1	9	1	`nnbansbananas`
12	5	5	0	`nsbananas`
13	13	13	1	`s`
14	6	0	—	`sbananas`

Figure 8.1: Suffixes of `annbansbananas` and the arrays S, R, and L

each step involves string comparison with α, which takes $\mathcal{O}(|\beta|)$ time in the worst case. This makes $\mathcal{O}(|\beta| \log |\alpha|)$ total.

Corollary: Using the suffix array for α, we can enumerate all occurrences of a substring β in time $\mathcal{O}(|\beta| \log |\alpha| + p)$, where p is the number of occurrences reported. Only counting the occurrences costs $\mathcal{O}(|\beta| \log |\alpha|)$ time.

Note: With further precomputation, time complexity of searching can be improved to $\mathcal{O}(|\beta| + \log |\alpha|)$.

Definition: The *rank array* $R[0 \ldots n]$ is the inverse permutation of S. That is, $R[i]$ tells how many suffixes of α are lexicographically smaller than $\alpha[i:\,]$.

Note: The rank array can be trivially computed from the suffix array in time $\mathcal{O}(n)$.

Definition: The *LCP array* $L[0 \ldots n-1]$ stores the length of the *longest common prefix* of each suffix and its lexicographic successor. That is, $L[i] = \mathrm{LCP}(\alpha[S[i]:\,], \alpha[S[i+1]:\,])$, where $\mathrm{LCP}(\gamma, \delta)$ is the maximum k such that $\gamma[\,:k] = \delta[\,:k]$.

Claim: Given the suffix array, the LCP array can be constructed in time $\mathcal{O}(n)$.

Observation: The LCP array can be easily used to find the longest common prefix of any two suffixes $\alpha[i:\,]$ and $\alpha[j:\,]$. We use the rank array to locate them in the lexicographic order of all suffixes: they lie at positions $i' = R[i]$ and $j' = R[j]$ (w.l.o.g. $i' < j'$). Then we compute $k = \min(L[i'], L[i'+1], \ldots, L[j'-1])$. We claim that $\mathrm{LCP}(\alpha[i:\,], \alpha[j:\,])$ is exactly k.

First, each pair of adjacent suffixes in the range $[i', j']$ has a common prefix of length at least k, so our LCP is at least k. However, it cannot be more: we have $k = L[\ell]$ for some $\ell \in [i', j'-1]$, so the ℓ-th and $(\ell+1)$-th suffix differ at position $k+1$ (or one of the suffixes ends at position k, but we can simply imagine a padding character at the end, ordered before all ordinary characters.) Since all suffixes in the range share the first k characters, their $(k+1)$-th characters must be non-decreasing. This means that the $(k+1)$-th character of the first and the last suffix in the range must differ, too.

This suggests building a *Range Minimum Query* (RMQ) data structure for the array L: it is a static data structure, which can answer queries for the position of the minimum element in a given range of indices. One example of a RMQ structure is the 1-dimensional range tree from section 7.1: it can be built in time $\mathcal{O}(n)$ and it answers queries in time $\mathcal{O}(\log n)$. There exists a better structure with build time $O(n)$ and query time $\mathcal{O}(1)$.

Examples: The arrays we have defined can be used to solve the following problems in linear time:

- *Histogram of k-grams:* we want to count occurrences of every substring of length k. Occurrences of every k-gram correspond to ranges of suffixes in their lexicographic order. These ranges can be easily identified, because we have $L[\ldots] < k$ at their boundaries. We only have to be careful about suffixes shorter than k, which contain no k-gram.

- *The longest repeating substring of a string α:* Consider two positions i and j in α. The length of the longest common substring starting at these positions is equal to the LCP of the suffixes $\alpha[i:\,]$ and $\alpha[j:\,]$, which is a minimum over some range in L. So it is always equal to some value in L. It is therefore sufficient to consider only pairs of suffixes adjacent in the lexicographic order, that is to find the maximum value in L.

- *The longest common substring of two strings α and β:* We build a suffix array and LCP array for the string $\alpha\texttt{#}\beta$, using a separator # which occurs in neither α nor β. We observe that each suffix of $\alpha\texttt{#}\beta$ corresponds to a suffix of either α or β. Like in the previous problem, we want to find a pair of positions i and j such that the LCP of the i-th and j-th suffix is maximized. We however need one i and j to come from α and the other from β. Therefore we find the maximum $L[k]$ such that $S[k]$ comes from α and $S[k+1]$ from β or vice versa.

Construction of the LCP array: Kasai's algorithm

We show an algorithm which constructs the LCP array L in linear time, given the suffix array S and the rank array R. We will use α_i to denote the i-th suffix of α in lexicographic order, that is $\alpha[S[i] :]$.

We can easily compute all $L[i]$ explicitly: for each i, we compare the suffixes α_i and α_{i+1} character-by-character from the start and stop at the first difference. This is obviously correct, but slow. We will however show that most of these comparisons are redundant.

Consider two suffixes α_i and α_{i+1} adjacent in lexicographic order. Suppose that their LCP $k = L[i]$ is non-zero. Then $\alpha_i[1 :]$ and $\alpha_{i+1}[1 :]$ are also suffixes of α, equal to $\alpha_{i'}$ and $\alpha_{j'}$ for some $i' < j'$. Obviously, $\mathrm{LCP}(\alpha_{i'}, \alpha_{j'}) = \mathrm{LCP}(\alpha_i, \alpha_{i+1}) - 1 = k - 1$. However, this LCP is a minimum of the range $[i', j']$ in the array L, so we must have $L[i'] \geq k - 1$.

This allows us to process suffixes of α from the longest to the shortest one, always obtaining the next suffix by cutting off the first character of the previous suffix. We calculate the L of the next suffix by starting with L of the previous suffix minus one and comparing characters from that position on:

```
Algorithm BuildLCP
Input: A string $\alpha$ of length $n$, its suffix array $S$ and rank array $R$
1. $k \leftarrow 0$                                  ◁ The LCP computed in previous step
2. For $p = 0, \ldots, n-1$:                          ◁ Start of the current suffix in $\alpha$
3.     $k \leftarrow \max(k-1, 0)$                    ◁ The next LCP is at least previous − 1
4.     $i \leftarrow R[p]$                            ◁ Index of current suffix in sorted order
5.     $q \leftarrow S[i+1]$                          ◁ Start of the lexicographically next suffix in $\alpha$
6.     While $(p+k < n) \wedge (q+k < n) \wedge (\alpha[p+k] = \alpha[q+k])$:
7.         $k \leftarrow k+1$                         ◁ Increase k while characters match
8.     $L[i] \leftarrow k$                            ◁ Record LCP in the array L
Output: LCP array $L$
```

Lemma: The algorithm BuildLCP runs in time $\mathcal{O}(n)$.

Proof: All operations outside the while loop take $\mathcal{O}(n)$ trivially. We will amortize time spent in the while loop using k as a potential. The value of k always lies in $[0, n]$ and it starts at 0. It always changes by 1: it can be decreased only in step 3 and increased only in step 7. Since there are at most n decreases, there can be at most $2n$ increases before k exceeds n. So the total time spent in the while loops is also $\mathcal{O}(n)$. □

Construction of the suffix array by doubling

There is a simple algorithm which builds the suffix array in $\mathcal{O}(n \log n)$ time. As before, α will denote the input string and n its length. Suffixes will be represented by their starting position: α_i denotes the suffix $\alpha[i:\,]$.

The algorithm works in $\mathcal{O}(\log n)$ passes, which sort suffixes by their first k characters, where $k = 2^0, 2^1, 2^2, \ldots$ For simplicity, we will index passes by k.

Definition: For any two strings γ and δ, we define comparison of prefixes of length k: $\gamma =_k \delta$ if $\gamma[\,:k] = \delta[\,:k]$, $\gamma \le_k \delta$ if $\gamma[\,:k] \le \delta[\,:k]$.

The k-th pass will produce a permutation S_k on suffix positions, which sorts suffixes by $\le_k$. We can easily compute the corresponding ranking array R_k, but this time we have to be careful to assign the same rank to suffixes which are equal by $=_k$. Formally, $R_k[i]$ is the number of suffixes α_j such that $\alpha_j <_k \alpha_i$.

In the first pass, we sort suffixes by their first character. Since the alphabet can be arbitrarily large, this might require a general-purpose sorting algorithm, so we reserve $\mathcal{O}(n \log n)$ time for this step. The same time obviously suffices for construction of the ranking array.

In the $2k$-th pass, we get suffixes ordered by $\le_k$ and we want to sort them by $\le_{2k}$. For any two suffixes α_i and α_j, the following holds by definition of lexicographic order:

$$\alpha_i \le_{2k} \alpha_j \iff (\alpha_i <_k \alpha_j) \vee ((\alpha_i =_k \alpha_j) \wedge (\alpha_{i+k} \le_k \alpha_{j+k}))\,.$$

Using the ranking function R_k, we can write this as lexicographic comparison of pairs $(R_k[i], R_k[i+k])$ and $(R_k[j], R_k[j+k])$. We can therefore assign one such pair to each suffix and sort suffixes by these pairs. Since any two pairs can be compared in constant time, a general-purpose sorting algorithm sorts them in $\mathcal{O}(n \log n)$ time. Afterwards, the ranking array can be constructed in linear time by scanning the sorted order.

There remains a little problem: the suffixes α_i and α_j can be shorter than $2k$ characters. In that case, $i+k$ and/or $j+k$ can point outside α. This is easy to fix: we replace any out-of-range suffix by the empty suffix, whose rank is always zero. (Alternatively, we can imagine that α is padded by n more null characters, which are smaller than all regular characters. This way, all suffixes will be well defined and $\le_k$ will always compare exactly k characters.)

Overall, we have $\mathcal{O}(\log n)$ passes, each taking $\mathcal{O}(n \log n)$ time. The whole algorithm therefore runs in $\mathcal{O}(n \log^2 n)$ time. In each pass, we need to store only the input string α, the ranking array from the previous step, the suffix array of the current step, and the encoded pairs. All this fits in $\mathcal{O}(n)$ space.

We can improve time complexity by using Bucketsort to sort the pairs. As the pairs contain only numbers between 0 and n, we can sort in two passes with n buckets. This takes $\mathcal{O}(n)$ time, so the whole algorithm runs in $\mathcal{O}(n \log n)$ time. Please note that the first pass still remains $\mathcal{O}(n \log n)$, unless we can assume that the alphabet is small enough to index buckets. Space complexity stays linear.

9 Parallel data structures

Contemporary computers often have multiple processors.[1] To utilize all available computing power, programs typically start multiple *processes* or *threads*, which run in parallel on different processors. This brings a new challenge: designing data structures which support concurrent access. These are often called *parallel* or *concurrent* data structures.

Parallel computing involves a lot of technical details. We tried to keep the exposition as straightforward as possible, but you can find many details in footnotes.

9.1 Parallel RAM

Details of multi-processor computers are quite complex and they vary between machines. Therefore, we will study parallel algorithms in a simple theoretical model instead, like we did with sequential algorithms. Our model is called the *Parallel RAM (PRAM).* It consists of several instances of the Random-Access Machine. We will call each instance a *processor.*

Instructions of the machine can access data in *local memory* of the current processor and also in *global memory* shared among all processors. There are no instructions for computing directly with data in global memory — we always have to read the data to local memory, modify them there, and write them back to global memory.

All processors execute the same program. However, the program has access to the identifier of the current processor, so it can easily switch to different branches of code on different processors. The sequences of instructions executed by the processors are called *processes.*

Many variants of the PRAM model guarantee that the instructions are executed in lock-step – in a single tick of a global system clock, each processor executes a single instruction of its process. As this is not true on real hardware, we will make no such assumption and allow each processor to execute instructions at its own pace.

It remains to specify what happens when multiple processes access the same cell of the global memory simultaneously. Concurrent reads will be considered correct. Concurrent writes and combinations of reads and writes will have undefined behavior.

[1] Sometimes, these are called *cores* or *hardware threads,* but this is a purely technical difference. Unless you are programming at a very low level, these behave as separate processors.

You can argue that it is almost impossible to produce correct parallel programs on machines with such weak semantics. We agree, so we are going to extend the machine later in this chapter by introducing *locks* and *atomic operations* with well-defined semantics.

9.2 Locks

Let us start small. Imagine a simple counter in global memory called *cnt* with an increment operation. Since we cannot compute in global memory, the increment must consist of three steps:

```
Procedure GlobalInc
  1. $t \leftarrow cnt$
  2. $t \leftarrow t + 1$
  3. $cnt \leftarrow t$
```

On a single processor, it is certainly correct. But consider that $cnt = 1$ and two processors try to increment it concurrently. It can happen that both read *cnt* to their local variables, both increment the local variable to 2, and then both write 2 back to *cnt* (in either order). So the end result is 2 instead of 3.

This is a prototypical example of what is called a *race condition* – the result of a computation depends on the exact ordering of operations across processors.

Race conditions are frequently avoided using *synchronization primitives.* The simplest such primitive is a *mutex* (short for "mutual exclusion", sometimes also called a *lock*). At any given moment, the mutex is either unlocked or locked. It supports two operations:

- Lock – If the mutex is unlocked, lock it. If it is locked, wait until it is unlocked by somebody else and then lock it.
- UnLock – If the mutex is locked, unlock it. If it is unlocked, crash (this should not happen in a correct program).

At this moment, we do not know how to implement the mutex. For the time being, we can consider it an extra feature of our machine. Later, we will show that it can be constructed from atomic instructions.

Typically, each instance of a data structure has its own mutex. Every operation on the structure is wrapped in an Lock/UnLock pair. This guarantees that at most one process is working with the data structure at any given time (this is the mutual exclusion). Hence, all operations are *atomic* — if we perform an operation, any other observer sees it either not started, or completely finished.

The dreaded deadlock

Problems arise if we want to perform atomic operations which affect more than one instance of the data structure. Consider the following situation: We have doubly linked lists, each list with its own mutex. Insertion and deletion is easy, but what if we want to move an item x from list A to list B atomically? (This means that for any outside observer, the item is always seen either in A, or in B.) An obvious solution would be:

```
Procedure AtomicMove
  1. Lock the mutex of $A$.
  2. Lock the mutex of $B$.
  3. Delete $x$ from $A$.
  4. Insert $x$ to $B$.
  5. Unlock the mutex of $B$.[2]
  6. Unlock the mutex of $A$.
```

Although this seems to be intuitively correct, it can fail badly. What if a process P_1 tries to move an item x from A to B, while another process P_2 tries to move y from B to A? It can happen that when P_1 obtains the mutex for A, P_2 obtains the mutex for B. In the next step, P_1 wants the mutex for B, while P_2 wants the mutex for A. So each process is waiting for a mutex held by the other process, and both processes will wait infinitely long.

This is called a *deadlock* and next to the race condition, it is the most frequent bug in parallel programs. Generally, more than two processes can participate in a deadlock. Let us consider the *dependency graph.* It is a directed graph, whose vertices are processes and an edge from i to j means that process i is waiting for a mutex currently locked by process j. If there is a directed cycle in this graph, the processes are obviously deadlocked. Otherwise, the computation can proceed from the source vertices of the graph (those with no incoming edges).

There is a simple solution to deadlocks: Establish an *ordering of all mutexes* in the system and always lock mutexes in increasing order. E.g., we can order them by their addresses in memory. This way, no deadlock can occur, because a cycle in the dependency graph would imply a cycle in the order. (In fact, the order need not be total — a partial order suffices as long as all locks taken by every single process are totally ordered.)

This technique can prevent all deadlocks as long as we can tell in advance, which mutexes will be needed by an operation (we have to sort them first). This is not true in all cases: for example, we might want to move an item to a list chosen according to the item's value.

[2] The order of unlocks is arbitrary, but we prefer to have the lock-unlock pairs nested properly.

Granularity of locks

Sometimes, one lock per instance is not the best possible granularity. If we have many small data structures, individual locks can consume too much memory. In such cases, we can have an array of mutexes and a hash function mapping an address of an instance to a mutex in the array. This can be very efficient, but when constructing operations working with multiple instances, beware that two instances can be mapped to the same mutex.

On the other hand, if all processes spend most of their time accessing a single instance, the lock guarding this instance will become a performance bottleneck. If it is so, we should try splitting the data structure to parts guarded by different locks. This is particularly easy if the data structure is a hash table: we can assign a separate mutex to each bucket. As long as different processes operate on different buckets (which is quite likely), they run with no contention. Beware that this setup does not support rehashing — the main pointer to the array of buckets is not guarded by a lock, so it must stay read-only.

We will see more examples of fine-grained locking in section 9.3.

Problems with locking

Locking is conceptually simple (though tricky to implement correctly), but it is not a panacea. There are multiple issues associated with locking:

- *Deadlocks* — operations which need to acquire more than one lock can cause deadlocks. As we saw, deadlocks can be usually prevented by ordering the locks, but it is not always possible.

- *Lack of composability* — intuitively, if operations α and β can be each performed atomically, it should be possible to do the composition $\alpha\beta$ atomically, too. This is not the case with locks, especially if we are not allowed to look inside α and β. Even if we are allowed, the problems with deadlocks arise. Full composability would require a completely different approach, for example transactional semantics. We will not discuss it here.

- *Fairness* — generally, there is no guarantee that a single process will finish in finite time. If there is a lot of contention at a lock, a process can wait indefinitely if other processes can always obtain the lock faster. This situation differs from the deadlock, because if other processes are stopped, the current process finishes. This is usually called *starvation.* It can be prevented by implementing the locks in a way which guarantees fairness.

- *Priority inversion* — if we have a system where processes have different priorities (e.g., real-time calculations have a high priority, while interaction with the user a lower one), a high-priority process can be blocked by a low-priority process if the former waits on a lock held by the latter. The standard fix for that in real-time systems is *priority inheritance* — a process holding a lock has its priority raised to the maximum priority of processes currently waiting for the lock. This further complicates implementation of locks.

- *Performance* — although modern operating systems supply a well-optimized implementation of locks, it still slows down the program and consumes memory. Especially if locking is fine-grained.
- *Fault tolerance* — if a process is terminated abnormally, all locks it held are left locked forever. If any other process tries to access the data structures protected by these locks, it will wait forever. The operating system could break the locks forcefully, but that would make other processes access the data structure in a potentially inconsistent state.

Some (but not all) of these problems will be solved by lock-free data structures introduced in section 9.4.

9.3 Locking in search trees

Let us try to combine binary search trees with fine-grained locking. We add a mutex to each node of the tree in attempt to make operations as parallel as possible. We will consider operations Find, Insert, and Delete on the tree.

Locking a path

Trivial solution first. Whenever we want to operate on an item, we follow a path from the root downwards and we lock the nodes on the path as we visit them. Find just follows the path. Insert without balancing follows the path and adds a new leaf at the end of the path. Delete follows the path and it either finds a node with at most 1 child (which can be cut out), or a node with 2 children, which is to be replaced by its successor, but finding the successor just extends the path further down.

This is obviously correct as all decisions we make (to go to the left or to the right) remain valid until the end of the operation — all nodes which affected the decision are kept locked. Even better, we can add rebalancing, which traverses the same path from the bottom to the top.

It can be also easily proven that no deadlock can occur: we can order the nodes by their depth in the tree and keep nodes at the same depth incomparable. This is a partial order in which every downward path forms a chain.

However, there is one huge fault in this approach: every path contains the root. So all operations are serialized by the lock in the root and the other locks have no effect.

Locking a sliding window

If the only operations on the tree are Find and Insert with no balancing, there is no need to keep the nodes we already passed locked. It is sufficient to have locked only the current node and its child where we want to go. This corresponds to sliding a window of size 2 over the path and locking only the nodes inside the window.

In a single step of Find(x), we compare x with the key in the current node and read the pointer to the corresponding child. We did this with the current node locked, so we do not race with other processes modifying the same node. Then we lock the child, unlock the current node and make the child the new current node. If we are Inserting, we finish by creating a new node (it need not be locked, since we are the only process possessing a pointer to it) and attaching it as a child to the current node (whose lock we hold).

This approach allows much greater concurrency — even though all paths start at the root, the root is quickly unlocked and we can expect that the paths will soon diverge, especially when accessing random items. Still, if the number of processes exceeds the height of the tree, the contention at the root can be significant.

Deadlocks are still impossible, because we take the locks in the order of increasing depth in the tree.

However, it is not clear how to add Delete and most importantly balancing. Traditional methods of balancing (AVL or red-black) do not work as they need to propagate changes upwards, possibly up to the root.

Arguing correctness: serializability

We would like to argue that our construction is correct, but we need to give an appropriate definition of correctness first. For example, what is the right answer if we are searching for an item, while another process is inserting an item with the same key?

The standard way of formulating correctness in concurrent systems is using *serializability.* A sequence of operations on a data structure is *serializable* if there exists a linear (total) order on all operations such that (1) the result of each operation is consistent with the operations preceding it in the order, (2) from the point of view of every process, the order of operations executed by that process is consistent with the total order.

Try to prove that the previous construction with Insert and Find is correct in the sense of serializability.

Sometimes, a stronger concept is used instead of serializability. It is called *sequential consistency* and it requires a single linear order common to all data structures. Reasoning in this model is much easier, but it is inefficient to implement on current hardware.

Top-down (a,b)-trees

The idea of locking by sliding a window over a path is much better suited to (a,b)-trees. We will use the version of (a,b)-trees with top-down balancing from section 3.3.

In Insert, we hold a lock on the current node and its parent. In each step, we split the current node if needed. Then we unlock the parent, find the appropriate child, lock it, and move there.

Delete is similar, but besides the current node and its parent, we sometimes need to lock a sibling. This requires more careful ordering of locks to avoid deadlocks. The primary ordering will be still by level (depth), but at the same level we will order the locks from the left to the right. This can be a problem, if after examining the current node we decide to look at the left sibling. One solution is to always lock the left sibling before the current node, even if it is not accessed. For another, see exercise 2.

Another problem in Delete is deletion of a key which is not at the lowest level. This is usually solved by replacing the key by its successor, but when looking for the successor, we need to keep the current node locked. This can be slow (e.g., if the current node is the root, we are effectively locking the whole tree). If this is a problem, we can keep the key there and mark it as deleted.

Exercises

1. Modify top-down Delete so that it guarantees that at most a half of all keys is marked as deleted.

2. Prove that that we can safely lock the children of a common parent in arbitrary order if the parent is locked first.

9.4 Lock-free data structures

We already mentioned multiple problems with locking. In this section, we will investigate data structures that do not need locks.

Atomic instructions

Without locks, we need to achieve consistency by other means. Fortunately, real machines typically offer a small set of instructions which guarantee atomicity of some kind. We can add them to our parallel RAM, too. Here are typical examples of *atomic instructions.* They usually operate on cells of global memory called *atomic registers.* Operations on individual atomic registers are serializable.

- *Read and write* — the most basic guarantee is that an atomic register can be read or written as a whole. Concurrent accesses to the same atomic register are resolved in an unknown, but consistent order.[3]

- *Exchange* — exchange contents of an atomic register with a value in local memory. This is sufficient for implementing a mutex: an unlocked mutex will have a zero value. Locking the mutex will exchange the register with 1. If the original value is zero, we have succeeded. If it was 1, we retry. To unlock the mutex, we write a zero atomically.[4]

- *Test and set bit* — set a given bit of an atomic register and return its original value. This is another popular building block for locks.

- *Fetch and add* — add a number to an atomic register and return its original value.

- *Compare and swap (CAS)* — it is given an atomic register R and values *old* and *new*. If R equals *old*, it is changed to *new*. Otherwise R is not changed. In all cases, the original value of R is returned.

- *Load linked and store conditional (LL/SC)* — a linked load is a normal atomic read, but the processor remembers the address and keeps monitoring it for access by other processors. A later store conditional to the same address succeeds if the address was not written to by other processors. Otherwise it reports failure.[5]

Current hardware implements atomic read/write and either CAS or LL/SC. The other atomic operations are either implemented or they can be simulated using CAS or LL/SC. CAS can be simulated using LL/SC, but not vice versa — the difference will play an important role in the next section.

Lock-free stack

Let us build a simple lock-free implementation of a stack. It will be represented as a linked list of *nodes*. Each node will contain data of the item and an atomic pointer to the *next* item. We will also keep an atomic pointer to the *head* of the list (the most recent item in the stack).

[3] Surprisingly, real hardware does not guarantee atomicity for normal memory accesses. For example, a 64-bit value on a 32-bit machine has to be written by two instructions, so another process can see a partially written value. More subtly, writes which cross a cache block boundary are also not atomic.

[4] On a real operating system, we usually want to let the process sleep instead of actively checking the register in a loop. For our simplistic introduction, spinning in a loop will suffice. This known as a *spinlock*.

[5] Real machines have various restrictions on LL/SC. Usually, only a very limited number of addresses can be monitored simultaneously. Also, monitoring is often based on cache blocks, so a write to another variable within the same cache block can also trigger SC failure.

We will implement the operations Push (insert a new item at the top of the stack) and Pop (remove an item from the top of the stack) as follows.

```
Procedure Push
Input: A new node $n$ containing the item to be pushed.
Repeat:
    $h \leftarrow stack.head$
    $n.next \leftarrow h$
    If CAS$(stack.head, h, n) = h$:
        Return.
```

```
Procedure Pop
Repeat:
    $h \leftarrow stack.head$
    $s \leftarrow h.next$
    if CAS$(stack.head, h, s) = h$:
        Return $h$.
Output: A node removed from the top of the stack.
```

The CAS guarantees that if another process interferes with the operation, we detect the interference and restart the operation. In the worst case, we can loop indefinitely and never complete the operation — this is called a *livelock*. In practice, the livelock is extremely improbable, because there is a lot of random factors which affect scheduling of processes, so every process eventually succeeds.

The ABA problem

Livelocks aside, the implementation looks correct. It is tempting to prove that it is serializable by the order of CASes on the list head. There is however one subtle hole in this argument: it works only if we assume that all nodes ever pushed to the stack are distinct.

Let us see what can happen if they are not. We start with the situation displayed in the figure: a stack containing items A (at the top) and B.

One process performs:

```
Procedure Process1
$x \leftarrow$ Pop                    ◁ We have $x = A$.
$y \leftarrow$ Pop                    ◁ We have $y = B$.
Push$(x)$                             ◁ We push $A$ back to the stack.
```

Another process starts a Pop, but it will be slower. It manages to perform steps 2 and 3 before Process1 starts popping, but step 4 will proceed after Process1 is done. So the second process sets $h = A$ and $s = B$. When it executes its CAS, the *head* is A again, so the *head* is changed to B. But this is wrong — B should not be in the list any longer as it was already removed by Process1's second Pop.

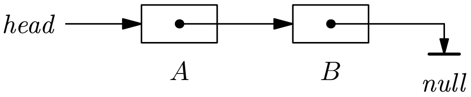

Figure 9.1: The list before Process1 starts

The core of the problem is that the original node A was confused with a logically different node stored at the same address.

This is called *the ABA problem* and it is a very typical fault of concurrent data structures. Let us consider some solutions:

- Use LL/SC instead of CAS. If step 2 of Pop is a LL and step 4 is a SC, the SC fails even if the *head* changed from A to B and back in the meantime.
- Have the machine support a *double-CAS* (alias DCAS or CAS2), which is a CAS on a pair of atomic registers simultaneously. Then we can replace the CAS in step 4 by

$$\mathrm{CAS2}(\langle \mathit{stack.head}, \mathit{h.next}\rangle, \langle h, n\rangle, \langle n, n\rangle),$$

  which detects that the head node was re-connected and it has a different successor now. Alas, no current processor supports CAS2.
- Have the machine support a *wide CAS* (WCAS), also called *double-width CAS* (DW-CAS). It is a CAS2 on two variables which are *adjacent* in memory. This can be used for *versioning of pointers:* each pointer will be accompanied by an integer version, which will be incremented every time the pointer is changed.[6] Then step 2 becomes

$$\langle h, v\rangle \leftarrow \langle \mathit{stack.head}, \mathit{stack.head\_version}\rangle$$

  and the test in step 4 becomes

$$\text{If } \mathrm{WCAS}(\langle \mathit{stack.head}, \mathit{stack.head\_version}\rangle, \langle h, v\rangle, \langle n, v+1\rangle) = \langle h, v\rangle\,.$$

[6] The version number can overflow, but as long as it does not wrap around in a single invocation of Pop, overflows are harmless. Also, a 64-bit version number is not likely to overflow within the lifetime of our program.

Unlike general CAS2, WCAS is often supported by hardware.

- Avoid recycling of nodes: for every Push, allocate a new node.

Memory allocation

Another subtle issue with concurrent data structures is memory allocation. With a locking data structure, we can free memory used by a node once we Pop the node off the stack. In the lock-free version, this could have disastrous consequences: even though the node is already popped, other processes can still access it. This happens if they obtained a pointer to the node before its Pop finished. Later, their CAS will reveal that the node is no longer present. But in the meantime, they might have accessed invalid memory and crashed.

The usual solution is to collect all unused nodes in a *free list* — an atomic list managed like our lock-free stack — and free them after we make sure that no processes reference them. There are multiple possibilities:

- *Global synchronization* — from time to time, we synchronize all processes at a point where they hold no pointers of their own, and free up all chunks from the free list. This is simple, but in many cases the synchronization points are hard to find.

- *Reference counting* — in every node, we keep an atomic counter of references to the node (i.e., local variables which point to the node). Again, we keep a free list. From time to time, we scan the free list and free up all nodes with zero references.

  Time to scan the list can be amortized nicely. If we have P processes and each of them holds at most R references, there are at most RP references at any given moment. Therefore if we start the scan once the list accumulates at least $2RP$ items, we always free up at least a half of the nodes in the list. So time spent on scanning can be charged on the freed nodes, each node paying $\mathcal{O}(1)$ time.

  Maintenance of the reference counters is surprisingly tricky. In the short time window before we read a pointer to a node from memory and increment the node's reference count, the node might have been already freed. We can figure out that this happened by re-reading the pointer. In this case, we decrement the counter back and retry. But if the node's memory was already recycled for different use, we might have temporarily corrupted an unrelated data structure. To avoid this, we will recycle memory only as nodes of the same memory layout with the reference counter at a fixed position.

  Let us modify Pop to handle reference counting:

Procedure PopRefCnt

```
1. Repeat:
2.     $h \leftarrow stack.head$
3.     Increment $h.ref\_cnt$.
4.     If $h \neq stack.head$:
5.         Decrement $h.ref\_cnt$ and retry the loop.
6.     $s \leftarrow h.next$
7.     if $\mathrm{CAS}(stack.head, h, s) = h$:
8.         Decrement $h.ref\_cnt$ and return $h$.
9.     Decrement $h.ref\_cnt$.
```

Output: A node removed from the top of the stack.

Note that we can safely decrease the *ref_cnt* in step 8, because as the „owner“ of the popped item, we are the only process which can free it. By the same argument, Push need not be modified because it references only the new node, which is owned by the current process.

- *Hazard pointers* – instead of keeping track of „hazardous references“ in each node, we collect all hazardous references in a single global array. The array is usually split to fixed-size blocks, each owned by a single process.

  Maintenance of hazard pointers is similar to that of reference counters. When a process wants to access a node, it sets one of its hazard pointers to that node. As with reference pointers, it is necessary to re-check that the node is still connected to the data structure.

  Let us see a version of Pop with hazard pointers. Again, Push need not be modified.

Procedure PopHP

```
1. Repeat:
2.     $h \leftarrow stack.head$
3.     $hp \leftarrow h$                          ◁ One of our hazard pointers
4.     If $h \neq stack.head$:
5.         Retry the loop.                      ◁ No need to reset hp here
6.     $s \leftarrow h.next$
7.     if $\mathrm{CAS}(stack.head, h, s) = h$:
8.         Let $hp \leftarrow \emptyset$ and return $h$.
```

Output: A node removed from the top of the stack.

When we want to scan the free list, we take a snapshot of the hazard pointer array and build a static data structure for the set of hazard pointers (in the simple case, it is a sorted array with binary search). For each node in the free list, we query the data structure to see if the node is still being accessed.

It takes some effort to prove that no node can be freed under our hands:

- Before a node enters the free list, it was disconnected from the data structure by a PopHP.
- If any concurrent PopHP reached step 6 with h pointing to the node, the node was not in the free list yet.
- So the hazard pointer in the concurrent PopHP is set before the node enters the free list.
- Hence the snapshot taken when freeing memory includes all relevant hazard pointers.

Hierarchy of concurrent data structures

Concurrent data structures differ in guarantees they provide. The typical guarantees form a hierarchy with each level stronger than the preceding one:

- *blocking* – the data structure is correct, but an operation can wait indefinitely, for example for a lock held by another process.
- *obstruction-free* — if all other processes stop, my operation will succeed in finite time.
- *lock-free* — if multiple processes execute operations, at least one of them will succeed in finite time. However, there is no guarantee of fairness — livelocks are allowed.
- *wait-free* — every operation is guaranteed to succeed in finite time.
- *bounded wait-free* — in addition to that, there is an upper bound on the time (typically a function of the number of processes competing for the data structure).

Our stack is indeed lock-free, but not wait-free.

Trouble with hardware and compilers*

In our discussion of concurrency, we made several simplifying assumptions, which are not always valid on real computers. We briefly mention what can go wrong, but will leave the details to texts on low-level programming.

Most importantly, we assumed that all observers see the same order of writes to global memory: if we write a node to the memory and then we make an atomic register point to this node, everybody who sees the new value of the atomic pointer will also see the contents of the node. This is generally not true in practice — writes to memory can be re-ordered by the hardware and different observers can see different order of writes.

To alleviate this problem, hardware usually supports *memory barrier* instructions. Memory operations issued before the barrier will complete earlier than memory operations issued after the barrier.[7]

Locking data structures need not care, because locks implicitly include memory barriers. Lock-free data structures typically require explicit barriers. Details vary between machines.

Also, reasoning about concurrent programs can be badly broken by program optimizations performed by compilers. Most compilers assume sequential semantics, so for example:

```
x = 1;
while (x == 1)
        // Do something, which does not involve x.
```

will be compiled to an infinite loop, even though another process can modify `x` and thus interrupt the loop. Another example is:

```
if (node_valid) {
        x = node;
        // Do something with x.
}
```

In sequential semantics, this can be rewritten as:

```
x = node;
if (node_valid) {
        // Do something with x.
}
```

which is not equivalent in a concurrent program.

To avoid problems of this type, compilers must be made aware that the variables can be accessed by other processes. For example, recent standards of the C and C++ languages define a formal memory model with concurrency, which includes explicit types for atomic variables.

[7] This is called a *full barrier.* There are also weaker barriers, for example one which orders writes, but not reads. Weaker barriers are faster than full ones.