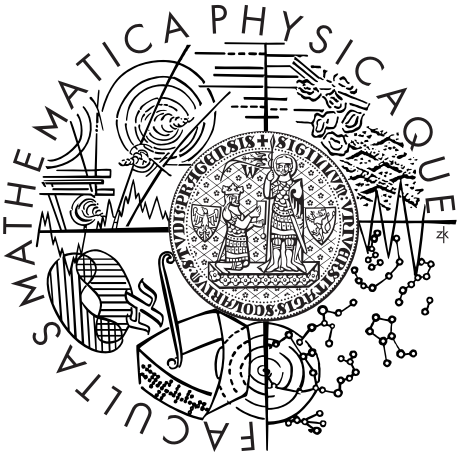
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Michael Pokorný

Praktické datové struktury

Department of Applied Mathematics

Supervisor of the bachelor thesis: Mgr. Martin Mareš, Ph.D.
Study programme: Computer science
Study branch: General computer science

Prague 2015

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

I wish to express my thanks to my advisor, Martin “MJ” Mareš, and everyone who ever organized or will organize the KSP seminar, for mentoring not only myself, but also countless other aspiring computer scientists over many years. I also thank my awesome parents and my awesome friends for being awesome.

Title: Praktické datové struktury

Author: Michael Pokorný

Department: Department of Applied Mathematics

Supervisor: Mgr. Martin Mareš, Ph.D., Department of Applied Mathematics

Abstract: In this thesis, we implement several data structures for ordered and unordered dictionaries and we benchmark their performance in main memory on synthetic and practical workloads. Our survey includes both well-known data structures (B-trees, red-black trees, splay trees and hashing) and more exotic approaches (k-splay trees and k-forests).

Keywords: cache-oblivious algorithms, data structures, dictionaries, search trees

Contents

Introduction

It is a well-known fact that there is a widening gap between the performance of CPUs and memory. To optimize memory accesses, modern computers include several small and fast caches between the CPU and main memory. However, if programs access data mostly randomly, I/O can become the performance bottleneck, even for programs running only in main memory (e.g., RAM).

Standard models of computation only measure the performance of algorithms in the number of CPU instructions executed and the number of memory cells used. These models aren't well suited for environments where different memory cells may have vastly different access speeds, for example depending on which caches currently hold the particular memory cell. The *external memory model* is a simple extension of the RAM model that is better suited for analyzing algorithms in memory hierarchies. This model adds a *block transfer* measure, which is the number of *blocks* transferred between a fast cache and a slow *external memory*. While the original motivation was developing fast algorithms for data stored on hard disks, the model can also be used to reason about boundaries between internal memory and caches.

Cache-oblivious algorithms are a class of external memory algorithms that perform equally well given any cache size and block size. Thanks to this property, they need little tuning for efficient implementation. Modern CPU architectures perform various optimizations that can change the effective size of a block, so tuning an algorithm by setting a good block size might be difficult. The external memory model and cache-oblivious algorithms are introduced in more detail in chapter 1.

This thesis explores the performance implications of memory hierarchies on data structures implementing *dictionaries* or *sorted dictionaries*. Especially unsorted dictionaries are very useful in practice, as illustrated by modern programming languages, which frequently offer a built-in data type for unsorted dictionaries (e.g., `std::unordered_map` in C++, `dict` in Python, or hashes in Perl).

Dictionaries maintain a set of N key-value pairs with distinct keys. Let us denote the set of stored keys by K. The keys are selected from an ordered *key universe* $\mathcal{U}$. The FIND operation find the value associated with a given key, or it reports that no such value exists. INSERT inserts a new key-value pair and DELETE removes a key-value pair given its key. Sorted dictionaries additionally allow queries on the set of keys K ordered by the ordering of $\mathcal{U}$. Given a key k, which may or may not be present in the dictionary, FINDNEXT and FINDPREVIOUS find the closest larger or smaller key present in the dictionary.

We implemented and benchmarked several common and uncommon data structures for sorted and unsorted dictionaries. Our choice of data structures was intended to sample simple dictionaries for RAM, cache-friendly dictionaries for external memory and self-adjusting structures. A high-level description of our implementation and benchmarks is presented in chapter 8. The results are discussed in chapter 9.

The most common data structure implementing an unsorted dictionary is a hash table. We describe some variants of hash tables in chapter 2. Most hash tables have expected $\mathcal{O}(1)$ time for all operations, which makes them quite prac-

tical. On the other hand, there is no simple way to implement sorted dictionaries efficiently via hash tables.

Ordered dictionaries are usually maintained in search trees. AVL trees, red-black trees and B-trees are examples of *balanced search trees*: a design invariant maintains their height low. The speeds of FIND, INSERT and DELETE mostly depend on the number of nodes touched by the operations, which is $\Theta(h)$, where h is the height of the tree. AVL trees and red-black trees are binary, so $h = \Theta(\log N)$. B-trees store $\Theta(b)$ keys per node, where $b \geq 2$, and they maintain all leaves at the same depth, so operations on B-trees touch $\Theta(\log_b N)$ nodes.

Splay trees are binary search trees with a self-adjusting rule, but splay trees are not balanced, as this rule does not guarantee a small height in all cases. The structure of a splay tree depends not only on the sequence of INSERTs and DELETEs, but also on executed FINDs. The self-adjusting rule (*splaying*) puts recently accessed nodes near the root, so FINDs on frequently or recently accessed keys are faster than in balanced trees. On the other hand, splaying can put the splay tree in a degenerate state, in which accessing an unlucky node touches $\Theta(N)$ nodes. Fortunately, the amortized number of node reads per FIND turns out to be $\mathcal{O}(\log N)$.

Splay trees are good at storing nonuniformly accessed dictionaries, but the expected number of nodes read to FIND a key is generally a factor of $\Theta(\log b)$ higher than in B-trees. We briefly introduce two self-adjusting data structures designed for both exploiting non-uniform access patterns and to perform less I/O: *k-splay trees* in chapter 5 and *k-forests* in chapter 6.

In chapter 7, we describe *cache-oblivious B-trees*. The number of memory operations for cache-oblivious B-tree operations is within a constant factor of the bounds given by a B-tree optimally tuned for the memory hierarchy. In practice, we found the performance of cache-oblivious B-trees quite competitive, especially on large datasets with uniform access patterns.

The chapters on hashing and cache-oblivious B-trees are based on the Advanced Data Structures course as taught by Erik Demaine on MIT in spring 2012. Lecture videos and other materials are available through MIT OpenCourseWare.[1]

Notation and conventions

In this thesis, logarithms are in base 2 unless stated otherwise. Chapter 7 uses the *hyperfloor* operation $\lfloor\!\lfloor x \rfloor\!\rfloor$, which is defined as $2^{\lfloor \log x \rfloor}$.

We denote bitwise XOR as $\oplus$, and similarly $\bigoplus_{i=1}^{N} a_i = a_1 \oplus \ldots \oplus a_N$.

This thesis contains several references to memory sizes. The base units of memory size are traditionally powers of two to simplify some operations on addresses. When discussing memory, we shall follow this convention, so 1 kB = 1024 B, 1 MB = 1024 kB, and so on.

[1] `http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-851-advanced-data-structures-spring-2012/index.htm`

1. Models of memory hierarchy

1.1 RAM

The performance of algorithms is usually measured in the RAM (*Random Access Machine*[1]) model. The ideal RAM machine has an infinite memory divided into cells addressed by integers. One instruction of the RAM machine takes constant time and it can compute a fixed arithmetic expression based on values of cells and store it in another cell. The expressions may involve reading cells, including cells pointed to by another cell. The exact set of arithmetic operations allowed on the values differs from model to model, but a typical set would include the usual C operators (`+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `~`, ...). A complete set of instructions also requires a conditional jump (based on the value of an expression). A more thorough treatment of various types of RAM is included in [35].

The model is also restricted by the *word size* (denoted w), which is the number of bits allocated for a cell. Choosing a too large w results in an unreasonably strong model. A typical choice of w is $\Theta(\log N)$, where N is size of input.

RAM algorithms are judged by their *time* and *space complexity*. The time complexity is the number of executed instructions as a function of input size and the space complexity is the maximum cell index accessed during execution.[2]

The time and space used by real computer programs is obviously related to the predictions made by the RAM model. However, the RAM model is far from equivalent with real computers, especially so when we are dealing with very large inputs. One significant source of this discrepancy is the assumption that all memory accesses cost the same $\mathcal{O}(1)$ time. In today's computers, the difference between accessing 1 kB cached in L1 cache and accessing another 1 kB in virtual memory paged-out to a disk is so large that it cannot be ignored.

1.2 Memory hierarchies

There are two major types of memory chips: *static* and *dynamic*. While static memory stores information as active states of semiconductors, dynamic memory uses capacitors, which lose charge over time. To avoid losing information, contents of dynamic memory must be periodically refreshed by "reading" the data and writing it back. Dynamic memory also has destructive reads: a read removes charge from capacitors, which must then be re-charged to keep the data. In general, static memory is fast and expensive, while dynamic memory is slow, cheap and more space-efficient. Main memories are usually dynamic.

CPU speeds are growing faster (60% per year) than speeds of memory (10% per year) [2]. A fast CPU needs to wait several hundred cycles to access main

[1] Another meaning of the abbreviation "RAM" is *Random Access Memory*. To avoid confusing the model and the type of computer memory, we will only use "RAM" to refer to the model.

[2] The number of overwritten cells would be a simpler definition of space complexity, but that would allow unrealistic "dirty tricks" like maintaining an unordered set with deterministic $\mathcal{O}(1)$ time for both updates and lookups by interpreting the key as a memory location.

- Processor registers. Their latency is usually one CPU cycle. The installed Intel Core i5-3230M CPU has 16 logical 64-bit general-purpose registers (plus some registers for specific purposes, e.g., vector operations).
- Level 1 data and instruction caches (L1d, L1i), both 8-way associative and 32 kB in size. Their latency is several CPU cycles.
- Level 2 (L2) cache, 8-way associative, 256 kB in size. It is an order of magnitude slower than the L1 cache. Again, each CPU core has a separate L2 cache.
- Level 3 (L3) cache, 12-way associative, 3 MB in size. The latency is higher than that of the L2 cache. Shared between CPU cores.
- Main memory, 6 GB.
- Disks: a 120 GB SSD and a 500 GB HDD.

Figure 1.1: The memory hierarchy of a Lenovo ThinkPad X230 laptop

memory. For example, Intel Core i7 and Xeon 5500 processors with clock frequencies in GHz need approximately 100 ns for a memory access access [34]. To alleviate the costs of memory transfers, several levels of static memory caches are inserted between the CPU and the main memory. The caches are progressively smaller and faster the closer they are to the CPU. The difference between speeds of memory levels reaches several orders of magnitude.

Larger and slower memories are also faster when accessing data physically close to recently accessed places. The hard disk is the most apparent example of this behavior – reading a byte at the current position of the drive head takes very little time compared to reading a random position, which incurs a slow disk seek. Similarly, modern CPUs contain a prefetcher, which tries to speculatively retrieve data from main memory before it is actually accessed. Modern memory is also structured as a matrix, and reading a location consists of first reading its row, and then extracting the column. Reading from the last accessed row is faster, since the row can be kept in a prefetch buffer.

Furthermore, if we need to wait for a long time to get our data from a slow store, it is wise to ask for a lot of data at once. Imagine an HTTP connection moving a 10 MB file across the Atlantic: the time to compose and send the HTTP request and to parse a response dwarfs the latency of the connection. Memory hierarchies transport data between memories in *blocks*. In the context of caches and main memory, these blocks are called *cache lines* and their usual size is 64 B. Operating systems usually perform disk I/O in batches of 4 kB, while physical HDD blocks typically span either 512 B, or 4 kB.

While programs may explicitly control transfers between the disk and main memory, the L1, L2 and L3 caches are under control of hardware. It is possible to advise the processor via special instructions about some characteristics of memory access (e.g., by requesting prefetching), but programs cannot explicitly set which memory blocks should be stored in caches.

A cache maintains up to m memory blocks called *cache lines*, typically of

64 B. We will first describe *fully associative caches*. When the CPU tries to access a memory location, the cache is checked first before accessing main memory. If the cache doesn't contain the requested data, it is loaded into the cache. When the cache becomes full (i.e., when all m cache lines are used), it needs to evict presently loaded cache lines before caching new ones. The evicted cache line is picked according to a *cache replacement policy*. The most commonly implemented cache replacement policy is LRU (*least recently used*), which evicts the memory line that was accessed least recently. A theoretically useful, but practically impossible policy is the *optimal replacement policy*, which always evicts the cache line that will be needed farthest in the future.

Fully associative caches can store an arbitrary set of m cached lines at any time. Because full associativity would be hard to implement in hardware, actual caches have a limited associativity. A t-way associative cache only allows each memory location to reside in one of t possible locations in the cache. If all t possible locations for a memory line are already occupied, the cache replacement policy evicts one of the t lines to make space fresh data. In a fully associative cache, we have $t = m$.

In hardware implementations of this scheme, t-way associative caches are divided into disjoint *sets*. Each set acts as a fully associative cache of with t slots. Finding a memory location in the cache is performed by first extracting the index of the set, and then scanning the t slots within the set that may possibly contain the sought location.

The associativity of caches t is usually a small power of two, like 8 or 16. The set of possible cache locations for lines in memory is determined by simply masking out a few address bits. This approach implies that accessing too many locations with addresses differing exactly by a power of two may result in *cache aliasing* – the accessed locations will all fall within the same set, and the cache will be able to store only t useful cache lines at once, which greatly degrades performance.

Programs which have *spatial* and *temporal locality of reference* will perform better on computers with hierarchical memory. Several models have been developed to benchmark algorithms designed for hierarchical memory.

1.3 The external memory model

The simplest model of memory hierarchies is the *external memory model*, which has only two levels of memory: the fast and small *cache* and the slow and potentially infinite *disk*.[3] The motivation for omitting all other levels is that for practical purposes, only one gap between memory levels is usually the bottleneck. For example, the bottleneck in a 10 TB database would probably be disk I/O, because only a negligible part of the database would fit into main memory and lower cache levels. If the time to make a query is 30 ms, optimizing transfers between L2 and L3 caches would be unlikely to help.

The cache and the disk are divided to blocks of size B. The size of the cache in bytes is fixed and is denoted M, and the number of blocks in the cache is

[3]We choose this naming to avoid the ambiguous term "memory", which may mean both "main memory" in relation to a disk, or an "external memory" in relation to a cache.

denoted m, such that $M = m \cdot B$. In the context of main memory, B is the size of a cache line (64 B).

In the external memory model, we allow free computation on data stored in the cache. The time complexity is measured in the number of *block transfers*, which read or write an entire block to the disk. Algorithms are allowed to explicitly transfer blocks, which corresponds to the boundary between main memory and disks in computers. The space complexity is the index of the last overwritten block. The parameters B and M are known to the program, which allows it to tune the transfers to the specific hierarchy.

Known results in the external memory model (extended to handle an array of disks allowing simultaneous access) are comprehensively surveyed in [57], including optimal sorting algorithms (in $\Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ block transfers), algorithms for computational geometry, graph algorithms and dictionary data structures.

1.4 Cache-oblivious algorithms

Cache-oblivious algorithms run in the external memory model. Their distinguishing property is that cache-oblivious algorithms are not allowed access to M and B. They may only assume the existence of a main memory and a cache. One further assumption required by some cache-oblivious algorithms is that the cache is *tall*, meaning that it can store more blocks than a block can store bits ($m = \Omega(B)$).

The algorithm must perform well independent of the values of M and B. On a machine with multiple levels of memory hierarchy, this means that the algorithm performs well on every cache boundary at the same time. Thanks to this property, cache-oblivious algorithms need little tuning to the computer they run on. Interestingly, automatically optimized cache boundaries include not only the usual L1, L2 and L3 caches, but also the *translation look-aside buffer* (TLB), which caches the mapping of pages from virtual to physical memory.

Cache-oblivious algorithms cannot control block transfers between the cache and the memory: they operate in uniformly addressed memory. This is similar to real computers: L1, L2 and L3 caches are also out of the program's control. The cache is, however, assumed to be unrealistically powerful: it is fully associative and the replacement policy is optimal. Fortunately, fully associative caches with optimal replacement can be simulated with a $2\times$ slowdown on $2\times$ larger LRU or FIFO caches [53]. The performance of many cache-oblivious algorithms is only slowed down by a constant factor when we halve the number of blocks, so cache-oblivious algorithms can usually be adopted in real world programs with only a constant slowdown compared to the model.

Many results from the external memory model have been repeated in the cache-oblivious model, including optimal sorting in $\Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ block transfers and permuting in $\Theta(\min\{N, \frac{N}{B} \log_{M/B} \frac{N}{B}\})$ block transfers. On the other hand, the hiding of cache parameters in the cache-oblivious model does slightly reduce its strength. No cache-oblivious algorithm can achieve $\Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ block transfers for sorting without the tall cache assumption [8]. Cache-oblivious algorithms can also permute either in $\Theta(N)$ or in $\Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ block transfers, but it is impossible for a cache-oblivious algorithm to pick the faster option. In chapter 7, we show a data structure that is a cache-oblivious equivalent of B-trees.

2. Hash tables

When implementing an unordered dictionary, hashing is the most common tool.[1] Hashing is a very old idea and the approaches to hashing are numerous. Hashing techniques usually allow expected constant-time Find, Insert and Delete at the expense of disallowing FindNext and FindPrevious. Certain schemes provide stronger than expected-time bounds, like deterministic constant time for Finds in cuckoo hashing (described in section 2.4).

The idea of hashing is reducing the size of the large key universe $\mathcal{U}$ to a smaller *hash space* H via a *hashing function* $h : \mathcal{U} \to H$. In this chapter, let us denote the size of the hash space M.[2] For a key k, we call $h(k)$ the *hash* of k.

The size of the hash space is selected small enough to allow using hashes of keys as indices in an array, which we call the *hash table*. The intuition of hashing is that if a key k hashes to x ($h(k) = x$), then the key is associated with the x-th slot of the hash table.

As long as all inserted keys have distinct hashes, hash tables are easy: each slot in the hash table will be either empty, or it will contain one key-value pair. Finds, Inserts and Deletes then all consist of just hashing the key and performing one operation in the associated slot of the hash table, which takes just constant time and a constant number of memory transfers. Unfortunately, unless we deliberately pick the hash function in advance to fit our particular key set, we need a way to deal with *collisions*, which occur when some keys $k_1 \neq k_2$ have the same hash value.

Specific collision resolution strategies differ between hashing approaches.

2.1 Separate chaining

Separate chaining stores the set of all key-value pairs with $h(k) = i$ in slot i. Let us call this set the *collision chain* for hash i, and denote its size C_i. The easiest solution is storing a pointer to a linked list of colliding key-value pairs in the hash table. If there are no collisions in an occupied slot, an easy and common optimization is storing the only key-value pair directly in the hash table. However, following a longer linked list is not cache-friendly.

If we use separate chaining, the costs of all hash table operations become $\mathcal{O}(1)$ for slot lookup and $\mathcal{O}(|C_i|)$ for scanning the collision chain. Provided we pick a hash function that evenly distributes hashes among keys, we can prove that the expected length of a collision chain is short.

To keep the expected chain length low, every time the hash table increases or decreases in size by a constant factor, we rebuild it with a new M picked to be $\Theta(N)$. The rebuild time is $\mathcal{O}(1)$ amortized per operation.

If we pick the hash function h at random (i.e., by independently randomly assigning $h(k)$ for all $k \in \mathcal{U}$), we have $\forall i \in H, k \in \mathcal{U} : \Pr[h(k) = i] = 1/M$. By linearity of expectation, we have $\mathbb{E}[C_i] = N/M$, which is constant if we maintain $M = \Theta(N)$. Since the expected length of any chain is constant, the expected

[1] Indeed, one of the many established meanings of the word "hash" is "unordered dictionary".

[2] No hashing schemes presented in this chapter are specific to external memory, so we do not need M to denote the cache size in this chapter.

time per operation is also constant. The N/M ratio is commonly called the *load factor*.

However, storing a random hash function would require $|\mathcal{U}| \log M$ bits, which is too much: a typical hash table only stores a few keys from a very large universe. In practice, we pick a hash function from a certain smaller family according to a formula with some variables chosen at random. Given a family $\mathcal{H}$ of hashing functions, we call $\mathcal{H}$ *universal* if $\Pr_{h \in \mathcal{H}}[h(x) = h(y)] = \mathcal{O}(\frac{1}{M})$ for any $x, y \in \mathcal{U}$. For any universal family of hash functions, the expected time per operation is constant when using chaining.

A family $\mathcal{H}$ is t-independent if the hashes of any set of t distinct keys $k_1, \ldots k_t$ are "asymptotically independent":

$$\forall h_1 \ldots h_t \in H : \Pr_{h \in H}[h(k_1) = h_1 \wedge \ldots \wedge h(k_t) = h_t] = \mathcal{O}(m^{-t}).$$

2-independent hash function families are universal, but universal families are not necessarily 2-independent. Trivially, random hash functions are k-wise independent for any $k \leq |\mathcal{U}|$.

Some common universal families of hash functions include:

- $h(k) = ((a \cdot k) \bmod p) \bmod M$, where p is a prime greater than $|\mathcal{U}|$ and $a \in \{0, \ldots p-1\}$ [11]. The collision probability for this hash function is $\frac{\lfloor p/M \rfloor}{p-1}$, so it becomes less efficient if M is close to p.

- $(a \cdot k) \gg (\log u - \log m)$ for $M, |\mathcal{U}|$ powers of 2 [19]. This hash function replaces modular arithmetic by bitwise shifts, which are faster on hardware.

- *Simple tabulation hashing* [11]: interpret the key $x \in K$ as a vector of c equal-size components $x_1, \ldots x_c$. Pick c random hash functions $h_1, \ldots h_c$ mapping from key components x_i to H. The hash of x is computed by taking the bitwise XOR of hashes of all components x_i:

  $$h(x) = h_1(x_1) \oplus h_2(x_2) \oplus \ldots \oplus h_c(x_c).$$

  Simple tabulation hashes take time $\mathcal{O}(c)$ to compute in the general RAM model. Storing the c individual hash functions needs space $\mathcal{O}(c \cdot |\mathcal{U}|^{1/c})$.

  Simple tabulation hashing is 3-independent and not 4-independent, but a recent analysis in [46] showed that it provides surprisingly good properties in some applications, some of which we will mention later.

While random hash functions combined with chaining give an expected $\mathcal{O}(1)$ time per operation, with high probability (i.e., $P = 1 - N^{-c}$ for an arbitrary choice of c) there is at least one chain with $\Omega(\log N / \log \log N)$ items. The high-probability worst-case bound on operation time also applies to hash functions with a high independence ($\Omega(\log N / \log \log N)$) [47] and simple tabulation hashing [46].

2.2 Perfect hashing

Perfect hashing avoids the issue of collisions by picking a hashing function that is collision-free for the given key set. If we fix the key set K in advance (i.e.,

if we perform *static hashing*), we can use a variety of algorithms which produce a collision-free hashing function at the cost of some preprocessing time. If the hash function produces no empty slots (i.e., $M = N$), we call it *minimal*.

For example, HDC (*Hash, Displace and Compress*, [4]) is a randomized algorithm that can generate a perfect hash function in expected $\mathcal{O}(N)$ time. The hash function can be represented using 1.98 bits per key for $M = 1.01N$, or more efficiently if we allow a larger M. All hash functions generated by HDC can be evaluated in constant time. The algorithm can be generalized to build k-perfect hash functions, which allow up to k collisions per slot. The *C Minimum Perfect Hashing Library*, available at `http://cmph.sourceforge.net/`, implements HDC along with several other minimum perfect hashing algorithms.

A perfect hashing scheme, commonly referred to as *FKS hashing* after the initials of the authors, was developed in [24] and later extended to allow updates in [21]. FKS hashing takes worst-case $\mathcal{O}(1)$ time for queries (an improvement over expected $\mathcal{O}(1)$ with chaining) and expected amortized $\mathcal{O}(1)$ for updates.

FKS hashing is two-level. The first-level hashing function f partitions the set of keys K into M buckets $B_1, \dots B_M$. Denote their sizes as b_i. Every bucket is stored in a separate hash table, mapping B_i to an array of size $\Theta(b_i^2) = \beta b_i^2$ via its private hash function g_i. The constant β will be picked later. Each function g_i is injective: buckets may contain no collisions.

If we pick the first-level hash function f from a universal family $\mathcal{F}$, the expected total size of all buckets is linear, so an FKS hash table takes expected linear space:

$$\mathbb{E}\left[\sum_{i=1}^{N} b_i^2\right] = \sum_{i=1}^{N} \sum_{\substack{j \in \{1,\dots N\} \\ j \neq i}} \Pr_{f \in \mathcal{F}}[f(k_i) = f(k_j)] = \mathcal{O}\left(N^2 \cdot \frac{1}{M}\right) = \mathcal{O}(N)$$

We pick f at random from $\mathcal{F}$ until we find one that will need at most $\sum_{i=1}^{N} b_i = \alpha N$ space, where α is an appropriate constant. Picking a proper α yields expected $\mathcal{O}(N)$ time to pick f.

To select a second-level hash function g_i, we pick one randomly from a universal family $\mathcal{G}$ until we find one that gives no collisions in B_i. By universality of g_i, the expected number of collisions in B_i is constant:

$$\mathbb{E}_{g_i \in \mathcal{G}}[\#\text{ of collisions in } B_i] = \binom{b_i}{2} \cdot \mathcal{O}\left(\frac{1}{b_i^2}\right) = \mathcal{O}(1)$$

By tuning the constant β, we can make the expected number of collisions small (e.g., $\leq \frac{1}{2}$), so we can push the probability of having no collisions above a constant (e.g., $\geq \frac{1}{2}$). This ensures that for every bucket, we will find an injective hash function in expected $\mathcal{O}(1)$ trials.

To FIND a key k, we simply compute $f(k)$ to get the right bucket, and we look at position $g_{f(k)}(k)$ in the bucket, which takes deterministic $\mathcal{O}(1)$ time.

We maintain $\mathcal{O}(N)$ buckets, and whenever N increases or decreases by a constant factor, we rebuild the whole FKS hash table in expected time $\mathcal{O}(N)$, which amortizes to expected $\mathcal{O}(1)$ per update. Each bucket B_i has $\mathcal{O}(b_i^2)$ slots. Whenever b_i increases or decreases by a constant factor (e.g., 2), we resize the reservation by the constant factor's square (e.g., 4). The expected amortized time for INSERT and DELETE is $\mathcal{O}(1)$. [20] enhances this to $\mathcal{O}(1)$ with high probability.

2.3 Open addressing

When we attempt to INSERT a new pair with key k into an occupied slot $h(k)$, we can accept the fact that the desired slot is taken, and we can start trying out alternate slots $a(k,1), a(k,2), \ldots$ until we succeed in finding an empty slot. This way, each hash table slot is again either empty or occupied by one key-value pair, but keys will not necessarily fall into their desired slot $h(k)$. We call this approach *open addressing*.

Examples of choices of $a(k,x)$ include:

- *Linear probing*: $a(k,x) = (h(k) + x) \bmod M$
- *Quadratic probing*: $a(k,x) = (h(k) + x^2) \bmod M$
- *Double hashing*: $a(k,x) = [h(k) + x \cdot (1 + h'(k))] \bmod M$, where h' is a secondary hash function

When using this family of strategies, one also needs to slightly change FIND and DELETE: FIND must properly traverse all possible locations that may contain the sought key, and DELETE and INSERT must ensure that FIND will know when to abort the search.

To illustrate this point, consider linear hashing with $h(A) = h(C) = 1$ and $h(B) = 2$. After inserting keys A and B, slots 1 and 2 are occupied. Inserting C will skip slots 1 and 2, and C will be inserted into slot 3. When we try to look for the key C later, we need to know that there are exactly 2 keys that hash to 1 (namely, A and C), so we won't abort the search prematurely after only seeing A and B.

The ends of collision chains can be marked for example by explicitly maintaining the lengths of collision chains in an array, or by marking the ends of chains with a bit flag.

All INSERTs must traverse the entire collision chain to make sure the inserted key is not in the hash table yet. When we DELETE a key, we need to ensure that the collision chain does not drift too far into alternative slots, so we traverse the entire collision chain and move the last key-value pair in the chain to the slot we deleted from.

By using *lazy deletion*, one can avoid traversing the entire collision chain in DELETEs. Deleted elements are not replaced by elements from the end of the chain, but they are instead just marked as deleted. FIND then skips over deleted elements and INSERTs are allowed to overwrite them. FINDs also "execute" the deletions by overwriting the first encountered deleted slot by any pair from further down the chain. An analysis of lazy deletions is presented in [12].

The reason why some implementations use quadratic probing and double hashing over linear probing is that linear probing creates long chains when space is tight. A chain covering hash values $[i; j]$ forces any key hashing to this interval to take $\mathcal{O}(j - i)$ time per operation and to extend the chain further.

However, linear probing performs well if we can avoid long chains: it has much better locality of reference than quadratic or double hashing. If $M = (1 + \varepsilon)N$, then using a random hash function gives expected time $\mathcal{O}(1/\varepsilon^2)$ with linear probing [33]. 5-independent hash function suffice to get this bound [40], and 4-independence is not enough [45]. [46] gives a proof that simple tabulation

hashing, which is only 3-independent and which can be implemented faster than usual 5-independent schemes, also achieves $\mathcal{O}(1/\varepsilon^2)$.

In section 9.5, we present our measurements of the performance of linear probing with simple tabulation hashing.

2.4 Cuckoo hashing

A cuckoo hash table is composed of two hash tables L and R of equal size $M = (1+\varepsilon)\cdot N$. Two separate hashing functions h_ℓ and h_r are associated with L and R. Each key-value pair (k, v) is stored either in L at position $h_\ell(k)$, or in R at position $h_r(k)$.

The cuckoo hash table can also be visualized as a bipartite "cuckoo graph" G, where parts are slots in L and R. Edges correspond to stored keys: a stored key k connects $h_\ell(k)$ in L and $h_r(k)$ in R.

Denote the vertices and edges of G as V and E. Let us define the *incidence graph* G_I of G. The vertices of G_I are vertices and edges of G, and the edges of G_I are $\{\{x, \{x, y\}\} : x \in V, \{x, y\} \in E\}$. The cuckoo graph G can be represented as a valid cuckoo hash table if and only if G_I has a perfect matching.

FINDs take $\mathcal{O}(1)$ worst-case time: they compute $h_\ell(k)$ and $h_r(k)$ and look at the two indices in L and R. The two reads are independent and can potentially be executed in parallel, especially on modern CPUs with instruction reordering.

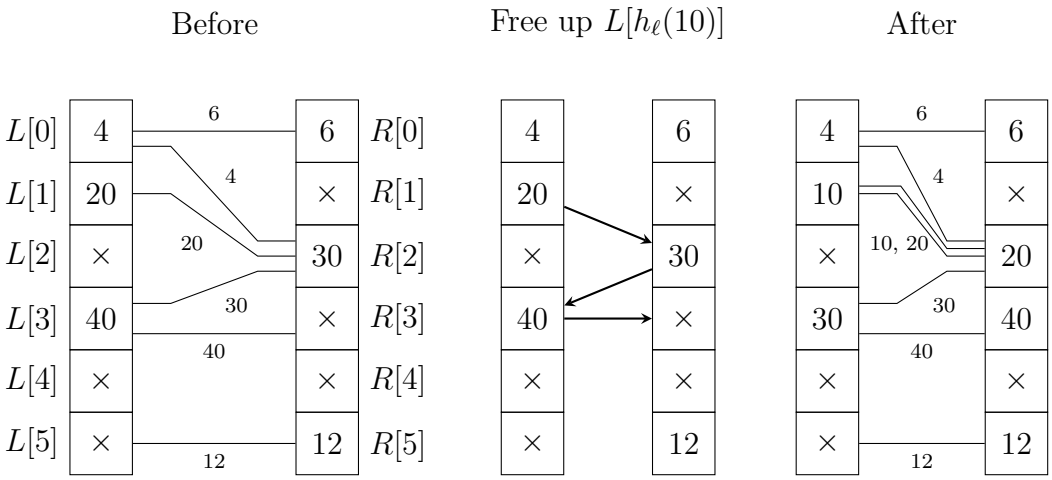
DELETEs take $\mathcal{O}(1)$ time to find the deleted slot and to mark it unused. Since the load factor of the cuckoo hash table needs to be kept between two constants, DELETEs end up taking $\mathcal{O}(1)$ amortized time to account for rebuilds.

To INSERT a new pair (k, v), we examine slots $L[h_\ell(k)]$ and $R[h_r(k)]$. If either slot is empty, (k, v) is inserted there. Otherwise, INSERT tries to free up one of the slots, for example $L[h_\ell(k)]$. If $L[h_\ell(k)]$ currently stores the key-value pair (k_1, v_1), we attempt to move it to $R[h_r(k_1)]$. If $R[h_r(k_1)]$ is occupied, we continue following the path $k, k_1, k_2, \ldots$ until we either find an empty slot, or we find a cycle. The name "cuckoo hashing" refers to this procedure by analogy with cuckoo chicks pushing eggs out of host birds' nests.

If we find a cycle both on the path from $L[h_\ell(k)]$ and on the path from $R[h_r(k)]$ (we can follow both at the same time), the incidence graph of the cuckoo graph with the added edge $(h_\ell(k), h_r(k))$ has no matching. To uniquely assign slots to key-value pairs, we need to pick new hashing functions h_ℓ and h_r and to rebuild the entire structure.

To guarantee good performance, cuckoo hashing requires good randomness properties of the hashing functions. With $\Theta(\log N)$-independent hash functions, INSERTs take expected amortized time $\mathcal{O}(1)$ and the failure probability (i.e., the probability that an INSERT will force a full rebuild) is $\mathcal{O}(1/N)$ [42]. 6-independence is not enough: some 6-independent hash functions lead to a failure probability of $\mathcal{O}(1-\frac{1}{N})$ [13]. The build failure probability when using simple tabulation hashing is $\mathcal{O}(N^{1/3})$. [46] demonstrates that storing all keys (a, b, c), where $a, b, c \in [N^{1/3}]$, is a degenerate case with $\Omega(N^{1/3})$ build failure probability.

Examples of $\Theta(\log N)$-independent hashing schemes include:



INSERT(10); $h_\ell(10) = 1$, $h_r(10) = 2$

Figure 2.1: Inserting into a cuckoo hash table

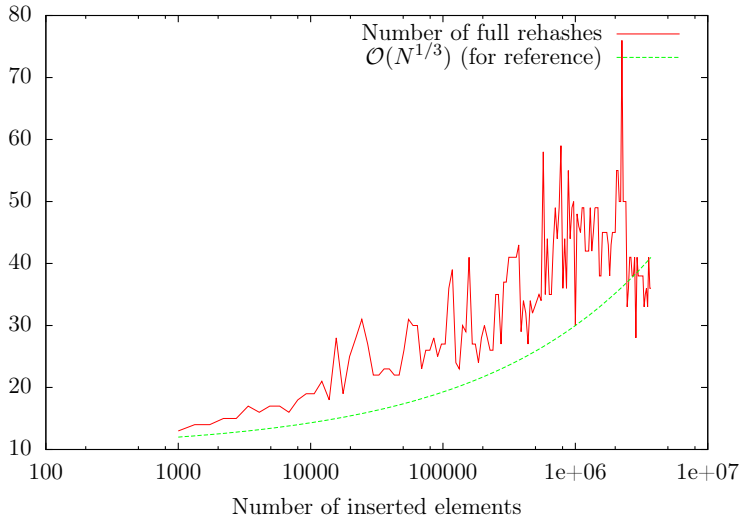


Figure 2.2: Degenerate performance of cuckoo hashing with simple tabulation ($\Theta(N^{1/3})$ rebuilds) as measured by our test program (`bin/experiments/cuckoo-cube`).

- For any k, a k-independent hash function family is [59]:

$$h(x) = \left(\sum_{i=0}^{k-1} a_i x^i \bmod p \right) \bmod M$$

  In this equation, p is a prime greater than $|\mathcal{U}|$ and all a_i are chosen at random from $\mathbb{Z}_p$. This hash function family is considered slow due to needing $\Theta(\log N)$ time and using integer division by p.[3] Evaluation of this hash function takes $\mathcal{O}(k)$ steps on the RAM.

- [55] presents a k-universal scheme that splits x into c components (as in simple tabulation hashing), expands x into a *derived key* z of $\mathcal{O}(ck)$ components, and then applies simple tabulation hashing to z: $h(x) = \bigoplus h_i(z_i)$. When applied in our setting, where c is fixed and $k = \mathcal{O}(\log N)$, the scheme takes time $\mathcal{O}(k)$ to evaluate. According to the authors, the scheme also has very competitive speed.

- In order to outperform B-trees on the RAM, we would like to use a hashing function which evaluates faster than $\log N$. A scheme from [51] needs only $\mathcal{O}(1)$ time to evaluate. Unfortunately, constructing a hash function by this scheme has high constant factors, so it is not very practical [41].

For simplicity, our implementation of cuckoo hashing uses simple tabulation hashing. As shown in section 9.5, INSERTs were significantly slower than with linear probing. Our work could be extended by trying more sophisticated and more independent hash functions.

2.5 Extensions of hashing

Several methods have been developed for maintaining hash tables in an external memory setting without expensive rehashing. Some are presented in chapter 9 of [57].

In [32], cuckoo hash tables are extended with a small constant-sized *stash* (e.g., 3 or 4 items), which lowers the probability of build failures and enhances performance (both theoretically and practically).

Cuckoo hashing can be generalized to more than two subtables. The INSERT procedure then performs slot eviction via a random walk. According to [38], experiments suggest that cuckoo hash tables with more functions may be significantly more effective than simple cuckoo hashing on high load factors. For example, 3 subtables perform well with loads up to 91%. However, more subtables would lead to more potential cache misses in FIND on large hash tables.

[3] As [11] points out, using Mersenne primes (i.e., $2^i - 1$ for some i) enables a faster implementation of the modulo operation. On 64-bit keys, we could use $p = 2^{89} - 1$. It might not be possible to generalize this approach, because the question whether there are infinitely many Mersenne primes remains open.

3. B-trees

The B-tree is a very common data structure for storing sorted dictionaries. They were introduced in 1970 as a method of efficiently maintaining an ordered index [3]. B-trees are particularly useful where the external memory model is accurate, because all information in every transferred block is actively used in searching. For example, the ext4, Btrfs and NTFS filesystems use variants of B-trees to store directory contents and the MongoDB and MariaDB databases use B-trees for indexing. Red-black trees, which are (in the left-leaning variant described by [49]) isomorphic to B-trees for $b = 2$, are commonly used to maintain in-memory dictionaries, for example in the GNU ISO C++ Library as the implementation of the `std::map` STL template. B-trees are also an optimal sorted dictionary data structure for the external memory model assuming the only operation allowed on keys is comparison.

We will use B-trees as a baseline external memory data structure and as a building block for more complex structures.

B-trees are a generalization of balanced binary search trees. Nodes of a B-tree keep up to $2b$ key-value pairs in sorted order. An inner node of a binary search tree with key X contains two pointers pointing to children with keys $< X$ and $\geq X$. In B-trees, inner nodes contain $k \in [b; 2b]$ keys $K_1 \dots K_k$. There are $k + 1$ child pointers in internal nodes: one for every interval $[K_1; K_2), \dots [K_{k-1}; K_k)$, plus one for $(-\infty; K_1)$ and $(K_k; \infty)$ each. Leaf nodes of a B-tree store $[b; 2b]$ key-value pairs.

B-trees can store values for keys either within all nodes (internal or leaves), or only in leaves. If values are stored only in leaves, keys stored within internal nodes are only used as pivots, and may not necessarily have a value within leaves – if a key is deleted, it is removed from its leaf along with its node, but internal nodes may retain a copy of the key. Our implementation stores values only inside leaves to allow tighter packing of internal nodes. Some sources refer to the variant storing values only within leaves as a *B+ tree*.

The essence of the $[b; 2b]$ interval is that when we try to insert the $(2b + 1)$-th key, we can instead split the node into two valid nodes of size b and borrow the middle key to insert the newly created node into the parent. Similarly, when we delete the b-th key, we first try to borrow some nodes from neighbours to create two valid nodes of size $[b; 2b]$, which works when there are at least $2b$ keys in both nodes combined. When there are $2b-1$ keys, we instead bring down one key from the parent and combine the two neighbours with the borrowed key into a new valid node with $b + (b - 1) + 1 = 2b$ keys.

In binary search trees, searching for a key K is performed by binary search. Starting in the root node, we compare K with the key stored in the node and we go left or right depending on the result. If the current node's key equals K, we fetch the value from the node and abort the search.

The process of finding a key-value pair in a B-tree generalizes this by picking the child pointer corresponding to the only range that contains the sought key K. To pick the child pointer, we can either do a simple linear scan in $\Theta(b)$, or we can use binary search to get $\Theta(\log b)$ if b is not negligible. Once we reach a leaf, we scan its keys and return the value for K (if such a value exists).

To INSERT a key-value pair in a B-tree, we first find the right place to put the new key-value pair, and then walk back up the tree. If an overfull node (with more than $2b$ keys) is encountered, it is split to two nodes of $\leq b$ keys. One of the keys from the split node is inserted to the parent along with a pointer to the new node. When we split the root node, we immediately create a new root node above the two resulting nodes.

Deletions employ a similar procedure: if the current node is underfull (i.e., if it has $< b$ keys), it is merged with one of its siblings. If there are not enough keys among the siblings for two nodes, one of the siblings takes ownership of all keys and the other sibling is recursively deleted from the parent. In this case, the corresponding key from the parent is pulled down into the merged node. The only exception is the root node, which may have $[1; 2b]$ keys. If deleting from the root would leave the root with only one child (and zero keys), we instead make the only child of the root the new root.

For B-trees that store values within both internal nodes and leaves, some deletions will need to remove a key used as a pivot within an internal node. In this case, the deleted key needs to be replaced by the minimum of its left subtree, or the maximum of its right subtree. Since our implementation stores values only inside leaves, deletions will always start at a leaf, which slightly simplifies them.

Thus, updates to the B-tree keep the number of keys of all nodes within $[b; 2b]$, so the depth of the tree is kept between $\log_2 bN$ and $\log_b N$ and the FIND operation takes time $\Theta(\log b \cdot \log_b N) = \Theta(\log N)$ and $\Theta(\log_b N)$ block transfers. FINDNEXT and FINDPREVIOUS also take time $\Theta(\log N)$, but if they are invoked after a previous FIND, we can keep a pointer to the leaf containing the key and accordingly adjust it to find the next or previous key, which runs in $\mathcal{O}(1)$ amortized time. By maintaining $\Theta(\log_b N)$ fingers into a stack of nodes, we can scan a contiguous range of keys of size k in $\Theta(\log_b N + k)$ time.

The INSERT and DELETE operations may spend up to $\Theta(b)$ time splitting or merging nodes on each level, so they run in time $\Theta(b \cdot \log_b N)$.

The main benefit of B-trees over binary trees is the tunable parameter b, which we can choose to be $\Theta(B)$. Practically, if the B-tree is stored in a disk, b can be tuned to fit one B-tree node in exactly one physical block. Thus, in the external memory model, FIND, INSERT and DELETE all take $\mathcal{O}(1)$ block transfers on every level, yielding $\Theta(\log_B N)$ block transfers per operation.

This is optimal assuming the only allowed operation on keys is comparison: reading a block of $\Theta(B)$ keys will only give us $\log B$ bits of information, since the only useful operation can be comparing the sought key K with every key we retrieved. We need $\log N$ bits to represent the index of the key-value pair we found, so we need at least $\log N / \log B = \log_B N$ block transfers.

4. Splay trees

Splay trees are a well-known variant of binary search trees introduced by [54] that adjust their structure with every operation, including lookups. Splay tree operations take $\mathcal{O}(\log N)$ amortized time, which is similar to common balanced trees, like AVL trees or red-black trees.

However, unlike BSTs, splay trees adjust their structure according to the access pattern, and a family of theorems proves that various non-random access patterns are fast on splay trees.

Let us also denote stored keys as $K_1, \ldots K_N$ in sorted order. Splay trees are binary search trees, so they are composed of nodes, each of which contains one key and its associated value. A node has up to two children, denoted *left* and *right*. The left subtree of a node with key K_i contains only keys smaller than K_i and, symetrically, the right subtree contains only keys larger than K_i.

Splay trees are maintained via a heuristic called *splaying*, which moves a specified node x to the root of the tree by performing a sequence of edge rotations along the path from the root to x. Rotations cost $\mathcal{O}(1)$ time each and rotating any edge maintains the soundness of search trees. Rotations are defined by figure 4.1.

To splay a node x, we perform *splaying steps*, which rotate edges between x, its parent p and possibly its grandparent g until x becomes the root. Splaying a node of depth d takes time $\Theta(d)$. Up to left-right symmetry, there are three cases of splaying steps:

Case 1 (*zig*): If p is the root, rotate the px edge. (This case is terminal.)

Case 2 (*zig-zig*): If p is not the root and x and p are both left or both right children, rotate the edge gp, then rotate the edge px.

Case 3 (*zig-zag*): If p is not the root and x is a left child and p is a right child (or vice versa), rotate the edge px and then rotate the edge now joining g with x.

To FIND a key K_F in a splay tree, we use binary search as in any binary search tree: in every node, we compare its key K_i with K_F, and if $K_i \neq K_F$, we continue to the left subtree or to the right subtree. If the subtree we would like to descend into is empty, the search is aborted, since K_F is not present in the tree. After the search finishes (successfully or unsuccessfully), we splay the last

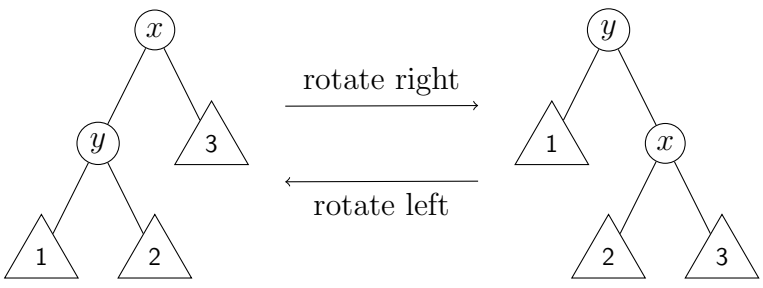


Figure 4.1: Left and right rotation of the edge between x and y.

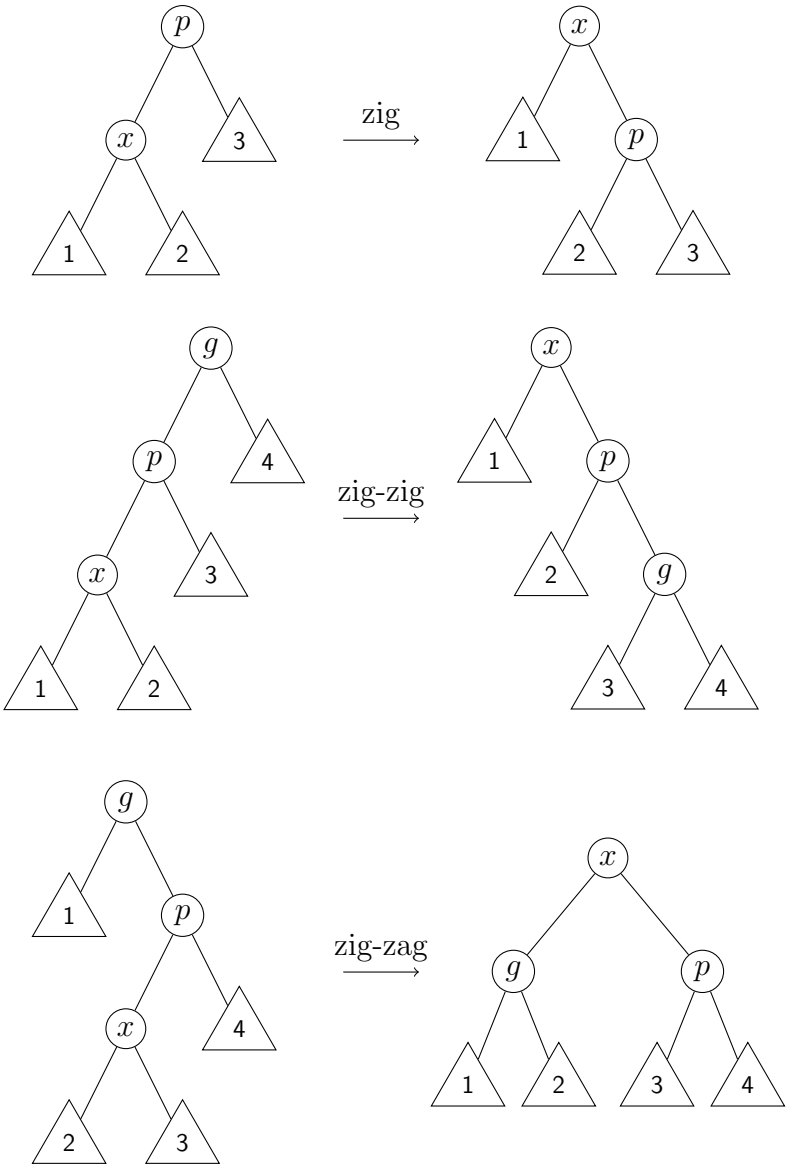


Figure 4.2: The cases of splay steps.

visited node. An INSERT is similar: we find the right place for the new node, insert it, and finally splay it up.

DELETEs are slightly more complex. We first find and splay the node x that we need to delete. If x has one child after splaying, we delete it and replace it by its only child. One possible way to handle the case of two children is finding the rightmost node x^- in x's left subtree and splaying it just below x. By definition, x^- is the largest in x's left subtree, so it must have no right child after splaying below x. We delete x and link its right subtree below x^-.

4.1 Time complexity bounds

In this section, we follow the analysis of splay tree performance from [54].

For the purposes of analysis, we assign a fixed positive weight $w(K_i)$ to every key K_i. Setting $w(K_i)$ to $1/N$ allows us to prove that the amortized complexity of the splay operation is $\mathcal{O}(\log N)$, which implies that splay tree FINDs and updates also take amortized logarithmic time. Proofs of further complexity results will use a similar analysis with other choices of the key-weight assignment function w.

Let us denote the subtree rooted at a node x as T_x and the key stored in node x as $T[x]$. Define the size $s(x)$ of a node x as $\sum_{y \in T_x} w(T[y])$ and the rank $r(x)$ as $\log s(x)$. For amortization, let us define the potential Φ of the tree to be the sum of the ranks of all its nodes. In a sequence of M accesses, if the i-th access takes time t_i, define its amortized time a_i to be $t_i + \Phi_{i-1} - \Phi_i$. Expanding and telescoping the total access time yields $\sum_{i=1}^{M} t_i = \Phi_0 - \Phi_M + \sum_{i=1}^{m} a_i$.

We will bound the magnitude of every a_i and of the total potential drop $\Phi_0 - \Phi_M$ to obtain an upper bound on $\sum_{i=1}^{M} t_i$. Since every rotation can be performed in $\mathcal{O}(1)$ pointer assignments, we can measure the time to splay a node in the number of rotations performed (or 1 if no rotations are needed).

Lemma 1 (Access Lemma)**.** *The amortized time a to splay a node x in a tree with root t is at most $3(r(t) - r(x)) + 1 = \mathcal{O}(\log(s(t)/s(x)))$.*

Proof. If $x = t$, we need no rotations and the cost of splaying is 1. Assume that $x \neq t$, so at least one rotation is needed. Consider the j-th splaying step. Let s and s', r and r' denote the size and rank functions just before and just after the step, respectively. We show that the amortized time a_j for the step is at most $3(r'(x) - r(x)) + 1$ in case 1 and at most $3(r'(x) - r(x))$ in case 2 or case 3. Let p be the original parent of x and g the original grandparent of x (if it exists).

The upper bound on a_j is proved by examining the structural changes made by each case of a splay step, as shown on figure 4.2.

Case 1 (*zig*): The only needed rotation of the px edge may only change the rank of x and p, so $a_j = 1 + r'(x) + r'(p) - r(x) - r(p)$. Because $r(p) \geq r'(p)$, we also have $a_j \leq 1 + r'(x) - r(x)$. Furthermore, $r'(x) \geq r(x)$, so $a_j \leq 1 + 3(r'(x) - r(x))$.

Case 2 (*zig-zig*): The two rotations may change the rank of x, p and g, so $a_j = 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)$. The zig-zig step moves x to the original position of g, so $r'(x) = r(g)$. Before the step, x was p's child, and after the step, p was x's child, so $r(x) \leq r(p)$ and $r'(p) \leq r'(x)$. Thus

$a_j \leq 2 + r'(x) + r'(g) - 2r(x)$. We claim that this is at most $3(r'(x) - r(x))$, that is, that $2r'(x) - r(x) - r(g) \geq 2$.

$2r'(x) - r(x) - r'(g) = -\log \frac{s(x)}{s'(x)} - \log \frac{s'(g)}{s'(x)}$. We have $s(x) + s'(g) \leq s'(x)$.

If $p, q \geq 0$ and $p + q \leq 1$, $\log p + \log q$ is maximized by setting $p = q = \frac{1}{2}$, which yields $\log p + \log q = -2$. Thus $2r'(x) - r(x) - r(g) \geq 2$ and $a_j \leq 3(r'(x) - r(x))$.

Case 3 (*zig-zag*): The amortized time of the zig-zag step is $a_j = 2 + r'(x) + r'(p) + r'(g) - r(x) - r(p) - r(g)$. $r'(x) = r(g)$ and $r(x) \leq r(p)$, so $a_j \leq 2 + r'(p) + r'(g) - 2r(x)$. We claim that this is at most $2(r'(x) - r(x))$, i.e., $2r'(x) - r'(p) - r'(g) \geq 2$. This can be proven similarly to case 2 from the inequality $s'(p) + s'(g) \leq s'(x)$. Thus we have $a_j \leq 2(r'(x) - r(x)) \leq 3(r'(x) - r(x))$.

Telescoping $\sum a_j$ along the splayed path yields an upper bound of $a \leq 3(r'(t) - r(x)) + 1$ for the entire splaying operation. □

Note that over any sequence of splay operations the potential Φ may only drop by up to $\sum_{i=1}^{N} \log(W/w(K_i))$ where $W = \sum_{i=1}^{N} w(K_i)$, since the size of the node containing K_i must be between $w(K_i)$ and W.

Theorem 1 (Balance Theorem)**.** *A sequence of M splay operations on a splay tree with N nodes takes time $\mathcal{O}((M + N) \log N + M)$.*

Proof. Assign $w(K_i) = 1/N$ to every key K_i. The amortized access time for any key is at most $3 \log N + 1$. We have $W = 1$, so the potential drops by at most $\sum_{i=1}^{N} \log(1/w(K_i)) = \sum_{i=1}^{N} \log N = N \log N$ over the access sequence. Thus the time to perform the access sequence is at most $(3 \log N + 1)M + N \log N = \mathcal{O}((M + N) \log N + M)$. □

When we FIND a key in a splay tree, we first perform a binary search for the node with the key, and then we splay the node. Since the splay operation traverses the entire path from the node to the root, we can charge the bundle together the cost of the binary search with the cost of the splay operation, so according to the balance theorem, the amortized time for a FIND is $\mathcal{O}(\log N)$. Thus, in the amortized sense, lookups in a splay tree are as efficient as in usual balanced search trees (like AVL or red-black trees).

Splay trees provide stronger time complexity guarantees than just amortized logarithmic time per access. For any key K_i, let $q(K_i)$ be the number of occurences of K_i in an access sequence.

Theorem 2 (Static Optimality Theorem)**.** *If every key is accessed at least once, the total access time is $\mathcal{O}(M + \sum_{i=1}^{N} q(K_i) \log \frac{M}{q(K_i)})$.*

Proof. Assign a weight of $q(i)/M$ to every key i. For any key i, its amortized time per access is $3(r(t) - r(x)) + 1 = -3r(x) + 1 = -3 \log(s(x)) + 1 \leq -3 \log(w(x)) + 1 = O(\log(M/q(i)))$. Since $W = 1$, the net potential drop over the sequence is at most $\sum_{i=1}^{N} \log(M/q(i))$. The result follows. □

Note that the time to perform the access sequence is equal to $\mathcal{O}(M \cdot (1 + H(P)))$, where $H(P)$ is the entropy of the access probability distribution defined as $P(i) = q(i)/M$. All static trees must take time $\Omega(M \cdot (1 + H(P)))$ for the access sequence. A proof of this lower bound is available in [1]. Since the taken for accesses in splay trees equals the asymptotic lower bound for static search trees, they are *statically optimal*, i.e., they are as fast as any static search tree.

Let us denote the indices of accessed keys in an access sequence S as $k_1, \ldots k_M$ so that $S = (K_{k_i})_{i=1}^{M}$.

Theorem 3 (Static Finger Theorem)**.** *Given any chosen key K_f, the total access time is $\mathcal{O}(N \log N + M + \sum_{j=1}^{M} \log(|k_j - f| + 1))$.*

Proof. Proof included in [54]. □

As proven in [15] and [14], it is fast to access keys that are close to recently accessed keys in the order of keys in K:

Theorem 4 (Dynamic Finger Theorem)**.** *The cost of performing an access sequence is $\mathcal{O}(N + M + \sum_{i=2}^{M} \log(|k_i - k_{i-1}| + 1))$.*

For any $j \in \{1, \ldots M\}$, define $t(j)$ to be the number of distinct keys accessed since the last access of key K_{k_j}. If K_{k_j} was not accessed before, $t(j)$ is picked to be the number of distinct keys accessed so far.

Theorem 5 (Working Set Theorem)**.** *The total access time is $\mathcal{O}(N \log N + M + \sum_{j=1}^{M} \log(t(j) + 1))$.*

Proof. Proof included in [54]. □

The working set theorem implies that most recently accessed keys (the "working set") are cheap to access, which makes splay trees behave well when the access pattern exhibits temporal locality.

To reiterate, the *static optimality theorem* claims that splay trees are as good as any static search tree. The *working set theorem* proves that the speed of accessing any small subset does not depend on how large the splay tree is. By the *dynamic finger theorem*, it is fast to access keys that are close to recently accessed keys in the order of keys in K.

All of these results are implied by the unproven *dynamic optimality conjecture*, which proposes that splay trees are dynamically optimal binary search trees: if a dynamic binary search tree optimally tuned for an access sequence S needs $\mathrm{OPT}(S)$ operations to perform the accesses, splay trees are presumed to only need $\mathcal{O}(\mathrm{OPT}(S))$ steps. Generally, a dynamic search tree algorithm X is called k-competitive, if for any sequence of operations S and any dynamic search tree algorithm Y the time $T_X(S)$ to perform operations S using X is at most $k \cdot T_Y(S)$. Splay trees were proven to be $\mathcal{O}(\log N)$-competitive in [54], but the dynamic optimality conjecture, which states that splay trees are $\mathcal{O}(1)$-competitive, remains an open problem.

4.2 Alternate data structures

Since the dynamic optimality conjecture has resisted attempts at solving for more than 20 years with little progress, a new approach was proposed in [18]. The authors suggest building alternative search trees with small non-constant competitive factors, and invented the *Tango tree*, which is $\mathcal{O}(\log \log N)$-competitive. However, worst-case performance of Tango trees is not very good: on pathological sequences, they underperform plain BSTs by a factor of $\Theta(\log \log N)$. [26] extends splaying with some housekeeping operations below the splayed element and proves that this *chain-splay* heuristic is $\mathcal{O}(\log \log N)$-competitive.

Finally, [58] presents *multi-splay trees*. Multi-splay trees are $\mathcal{O}(\log \log N)$-competitive and they have all important proven properties of splay trees. They also satisfy the *deque property* ($\mathcal{O}(M)$ time for M double-ended queue operations), which is only conjectured to apply to splay trees. Unlike Tango trees, they have an amortized access time of $\mathcal{O}(\log N)$. While an access in splay trees may take time up to $\mathcal{O}(N)$, multi-splay trees guarantee a more desirable worst-case access time of $\mathcal{O}(\log^2 N)$.

4.3 Top-down splaying

Simple implementations of splaying first find the node to splay, and then splay along the traversed path. The traversed path may be either kept in an explicit stack, which is progressively shortened by splay steps, or the tree may be augmented with pointers to parent nodes. Explicitly keeping a stack is wasteful, because it requires dynamically allocating up to N node pointers when splaying. On the other hand, storing parent pointers makes nodes less memory-efficient, especially with small sizes of keys.

A common optimization from the original paper [54] is *top-down splaying*, which splays while searching. Top-down splaying maintains three trees while looking up a key k: the left tree L, the right tree R and the middle tree M. L holds nodes with keys smaller than k and R holds nodes with keys larger than k. M contains yet unexplored nodes. Nodes are only inserted into the rightmost empty child pointer of L and the leftmost empty child pointer of R. The locations of the rightmost empty child pointer of L and leftmost empty child pointer of R are maintained while splaying. Denote these locations L_+ and R_-.

Initially, L and R are both empty and M contains the entire splay tree.

Every splaying step processes up to 2 upper levels of M. All nodes in these levels are put either under L_+, or R_- depending on whether their keys are larger or smaller than k. L_+ and R_- are then updated to point to the new empty child pointers. Finally, when M has k in the root, we compose the root of M with its two children, L, and R.

An implementation of top-down splaying in C by Sleator is available at `http://www.link.cs.cmu.edu/link/ftp-site/splaying/top-down-splay.c`.

4.4 Applications

A drawback of splay trees is that accessing the $\mathcal{O}(\log N)$ amortized nodes on the access path may incur an expensive cache miss for every node – splay trees make no effort to exploit the memory hierarchy. In contrast, lookups in B-trees and cache-oblivious B-trees only cause up to $\Theta(\log_B N)$ cache misses, which is a multiplicative improvement of $\log B$.

On the other hand, splaying automatically adjusts the structure of the splay tree to speed up access to recently or frequently accessed nodes. By the working set theorem, accessing a small set of keys is always fast, no matter how big the splay tree is. While B-trees and cache-oblivious B-trees are also faster on smaller working sets, the complexity of any access remains $\mathcal{O}(\log_B N)$, so accessing a small working set is slower on larger dictionaries.

Splay trees are frequently used in practice where access patterns are assumed to be non-random. For example, splay trees are used in the FreeBSD kernel to store the mapping from virtual memory areas to addresses.[1] Unfortunately, splay trees are much slower than BSTs on more random access patterns: splaying always performs costly memory writes. For some applications (e.g., real-time systems), worst-case $\Theta(N)$ performance is also unacceptable (especially if the splay tree stores data entered by an adversary).

[1] See `sys/vm/vm_map.h` in the FreeBSD source tree (lines 169–174 at SVN revision 271244); available at `https://svnweb.freebsd.org/base/head/sys/vm/vm_map.h?view=markup`.

5. k-splay trees

Splay trees are useful for exploiting the regularity of access patterns. On the other hand, they do not make good use of memory blocks – every node is just binary, so walking to a node in depth d takes $\mathcal{O}(\log d)$ block transfers. B-trees improve this by a factor of $\log B$ by storing more keys per node.

Self-adjusting k-ary search trees (also known as *k-splay trees*) developed in [50] use a heuristic called *k-splaying* to dynamically adjust their structure according to the query sequence. We decided to implement and benchmark this data structure because of the potential to combine self-adjustment for a specific search sequence with cache-friendly node sizes.

Similarly to B-trees, nodes in k-ary splay trees store up to $k-1$ key-value pairs and k child pointers. All nodes except the root are *complete* (i.e., they contain exactly $k-1$ pairs).

B-tree operations take $\Theta(\log_b N)$ node accesses. It can be proven that k-splay tree operations need at most $\mathcal{O}(\log N)$ amortized node accesses, which matches ordinary splay trees, but unfortunately k-splay trees do not match the optimal $\mathcal{O}(\log_k N)$ bound given by B-trees. This conjecture from [50] was disproved in [37], where a family of counterexample k-ary trees was constructed. For any k-ary tree from this family, there exists a key which requires $\log N$ node accesses to k-splay, and which produces another tree from the same family after being k-splayed.

k-splay trees are proven in [50] to be $(\log k)$-competitive compared to static optimal k-ary search trees.

The k-splaying operation is conceptually similar to splaying in the original splay trees of Sleator and Tarjan, but k-splaying for $k = 2$ differs from splaying in original splay trees. To disambiguate these terms, we will reserve the term *splaying* for Sleator and Tarjan's splay trees and we will only use *k-splaying* to denote the operation on k-ary splay trees.

In our description of k-splaying, we shall begin with the assumption that all nodes on the search path are complete, and we will later modify the k-splaying algorithm to allow *incomplete* and *overfull* nodes. Because splay trees store one key per node, it is not necessary to distinguish between splaying a node and splaying a key. In k-ary splay trees, we always k-splay nodes.

As with Sleator and Tarjan's splaying, k-splaying is done in steps.

Splaying steps of Sleator and Tarjan's splay trees involve 3 nodes connected by 2 edges, except for the final step, which may involve the root node and one of its children connected by an edge. In every step, the splayed node (or key) moves from the bottom of a subtree to the root of a subtree. Splaying stops when the splayed node becomes the root of the splay tree.

In k-splay trees, k-splaying steps involve k edges connecting $k+1$ nodes, except for the final step which may involve fewer edges. The objective of k-splaying is to move a *star node* to the root of the tree. In every k-splaying step, we start with the star node at the bottom of a subtree, and we change the structure of the subtree so that the star node becomes its root. The initial star node is the last node reached on a search operation, and k-splaying will move the content of this node closer to the root. However, the structural changes performed in k-splaying

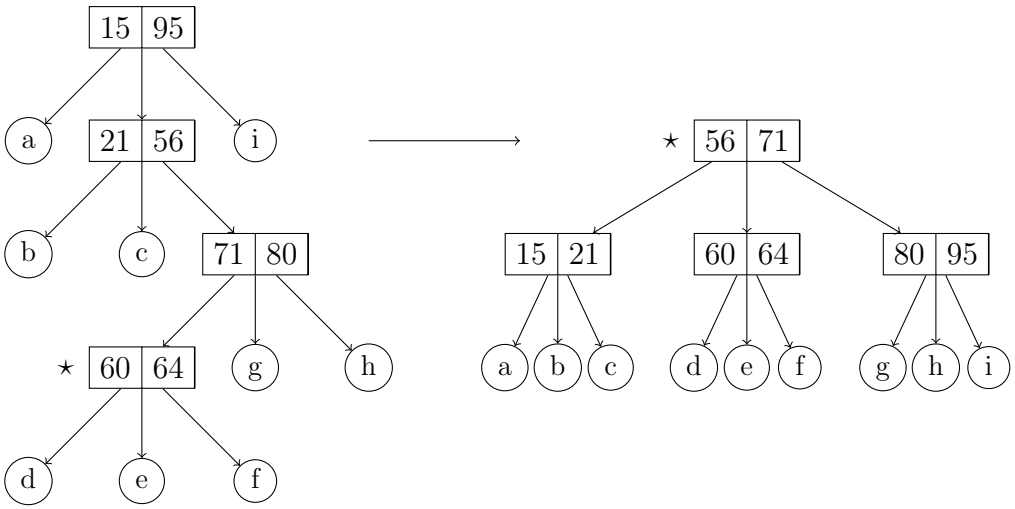


Figure 5.1: Non-terminal 3-splaying step with marked star nodes.

steps shuffle the keys and pointers of involved nodes, so after a k-splay step, the star node may no longer contain the key we originally looked up. Unlike splay trees, the distinction between k-splaying a node and a key is therefore significant.

There are two types of k-splay steps: *non-terminal steps* on $k+1$ nodes, and *terminal steps* on less than $k+1$ nodes. We consider first the non-terminal steps.

Non-terminal k-splay steps examine the bottom $k+1$ nodes in the search path to the current star node. Denote the nodes $p_0, \dots p_k$, where p_i is the parent of p_{i+1} and p_k is the current star node. A non-terminal step traverses these nodes in DFS order and collects *external children* (i.e., children that are not present in $\{p_0, \dots p_k\}$) and all key-value pairs of $p_0, \dots p_k$ in sorted order. These children and key-value pairs are *fused* into a "virtual node" with $(k+1) \cdot (k-1)$ keys and $(k+1) \cdot (k-1) + 1$ children. Finally, this "virtual node" is broken down into a fully balanced k-ary search tree of depth 2. There is exactly one way to arrange this k-ary search tree, because all its nodes must be complete. The root of this search tree becomes the new star node and a new k-splay step starts.

Terminal k-splay steps occur when we are k-splaying at most k nodes, with the highest one always being the root of the k-splay tree. Unlike non-terminal steps, terminal steps have too few nodes in the splayed path, so they leave the root underfull. Otherwise, terminal steps are the same as non-terminal steps.

DELETE and INSERT operations employ variations of the same procedure. We look up the node to update and we insert or delete the key within the node. Then, we k-splay the node, while allowing the star node to be temporarily overfull or underfull. After k-splaying the updated node, the root becomes the star node. If it has only one child, we merge it with the root, and if the root has more than k children, we split it.

By performing an analysis similar to splay trees, [50] shows that the number of node reads performed in a sequence of m FINDs in a k-splay tree with N keys is $\mathcal{O}(m \log N + \frac{N}{k} \log N)$.

6. k-forests

The performance of k-splay trees may suffer on access sequences chosen by an adversary: it is known that certain access sequences must take $\Omega(\log N)$ block transfers per element. A better worst-case block transfer bound of $\mathcal{O}(\log_k N)$ for FIND, INSERT and DELETE operations is achieved by *k-forests* [36]. A k-forest is a self-adjusting dictionary data structure based on a sequence of k-ary search trees of increasing size. A simpler randomized version of k-forests achieves the same expected bound on block transfers.

It can be proven that a generalization of the Static Optimality theorem also applies to k-forests: an element with access probability p will have average lookup time $\mathcal{O}\left(\min\{\log_k \frac{1}{p}, \log_k N\}\right)$. k-forests also have the working set property.

On the other hand, k-forests do not have fast finger queries. Scanning the keys in a k-forest in increasing order requires $\Omega(N \log_k N)$ block transfers, as every predecessor or successor query needs $\Omega(\log_k N)$ block transfers.

We are unaware of any data structure providing good speed for both static searches and finger queries with a $\log_B$ reduction in block transfers over splay trees.

A k-forest containing N elements is a sequence of $\mathcal{O}(\log_k N)$ k-ary search trees $T_1, T_2, \ldots$. Each tree T_i contains up to $s(i)$ key-value pairs, where $s(i) = k^{2^i} - 1$. The trees need to support INSERTs and DELETEs in $\mathcal{O}(\log_k s(i))$ time, so B-trees with $b = \mathcal{O}(k)$ are a natural choice for representing the trees.

Every tree, with the possible exception of the last one, is kept full, so if we represent the trees as B-trees, then T_1 has height 1, T_2 has height 2, T_3 has height 4, and so on. Each key-value pair inserted into the data structure is always kept in one tree T_i. The self-adjusting operation moves elements between trees to keep more recently accessed elements in smaller trees that are faster to access. A deterministic version of this operation keeps the $s(1)$ most recently accessed items in tree T_1, then the next $s(2)$ most recently accessed items in T_2, and so on.

To FIND a key, we scan the trees T_i in increasing order of i. If we scan all trees and find no matches, the search is unsuccessful. Otherwise, the key-value pair is removed from its tree T_i and inserted into T_0. We then *demote* one item from tree T_0 to T_1, then from T_1 to T_2, and so on until all trees meet their size bounds.

Demotion can be implemented deterministically, or randomly. Deterministic, or *LRU demotion*, maintains information about the time of last access to elements in every tree. The element pushed down to the next tree is always the least recently accessed one, so the trees $T_1, T_2, \ldots$ are maintained in sorted order with respect to last access time. *Random demotion* is simpler to implement and it requires no auxiliary data: we simply select a random key from T_i and demote it to T_{i+1}. Both demotion approaches are equivalent in the sense that the expected time bounds arising from random demotion equal the deterministic time bounds of LRU demotion.

INSERT and DELETE operations are similar to FIND. INSERT inserts to T_1 and then demotes elements from $T_1, T_2, \ldots$ until the forest is fixed. DELETE finds the pair in its tree T_i and removes it, and then promotes items from all

trees $T_{i+1}, T_{i+2}, \ldots$ Since both INSERT and DELETE need to promote and demote items in potentially every tree T_i, they both access $\mathcal{O}(\log N)$ nodes.

As in our description of splay trees, let K be the set of keys, let $\{k_i : i \leq M\}$ be the sequence of indices of accessed keys and for any $j \in \{1, \ldots M\}$, let $t(j)$ be the number of distinct keys accessed since the last access of key K_{k_j}. [36] proves the following theorems analogous to desirable properties of splay trees:

Theorem 6 (Working Set Theorem for k-forests)**.** *With LRU demotion, the j-th lookup accesses $\mathcal{O}(\log_k t(j))$ nodes. With random demotion, it accesses expected $\mathcal{O}(\log_k t(j))$ nodes.*

Again, let $q(K_i)$ be the number of accesses to K_i.

Theorem 7 (Static Optimality Theorem for k-forests)**.** *Accessing a sequence of keys $\{K_{k_i} : i \leq M\}$ requires $\mathcal{O}(\sum_{i=1}^{N} q(K_i) \log_k \frac{M}{q(K_i)})$ node accesses.*

We implemented k-forests with random demotion to benchmark their performance, especially to examine the constant multiplicative factors taken by promoting and demoting in every operation. Compared to k-splay trees, k-forests also provide better asymptotic bounds on the number of transfers per operation, so we wanted to see which choice is faster in practice.

6.1 Related results

[30] and [10] extend k-forests to build data structures with the *unified property*, which is a strong generalization of the dynamic finger property and the working set property of splay trees. It is not known whether splay trees also have the unified property. The data structure from [30] also improves upon splay trees by guaranteeing a worst-case access time of $\mathcal{O}(\log N)$. Unfortunately, the structure (as described by the authors) is static and highly complex: each node of the structure stores 21 pointers.

[7] uses a similar doubly exponential construction to build a self-adjusting skip list. Operations on this skip list take time $\mathcal{O}(\log t(j))$, which matches the working set bound on splay trees. This structure is also shown to be dynamically optimal among skip lists constrained by some restrictions (e.g., allowing at most b consecutive forward steps on any level for a fixed b), which implies that this restriction of skip lists does not allow better performance than $\mathcal{O}(\log t(j))$. Skip lists in this *SL-B model* can be simulated in B-trees via a construction from [17]. This representation yields a B-tree that is dynamically optimal within a restricted class in the external memory model. We are not aware of any implementations or benchmarks of this data structure.

7. Cache-oblivious B-trees

The *cache-oblivious B-tree* introduced by [5] is a data structure which replicates the asymptotic performance of B-trees in a cache-oblivious setting. FINDs and updates on cache-oblivious B-trees require $\Theta(\log_B N)$ [1] amortized block transfers, which is similar to cache-aware B-trees. Scanning a range of t keys also takes $\Theta(\log_B N + t/B)$ time.

We include cache-oblivious B-trees in our survey, because while they are simultaneously optimal on every cache boundary, they are also considerably more complex. The performance bounds the variant we implemented provides are also only amortized, while B-trees also guarantee a fast worst case. As we describe in section 9.3, our implementation of cache-oblivious B-trees is significantly faster than simple B-trees on large dictionaries.

The original cache-oblivious B-tree as described in [5] used weight-balanced B-trees. We implement a simplified version of the structure from [9], which is composed of three components. The *van Emde Boas layout* of binary trees is a way of cache-obliviously maintaining full binary trees in memory such that reading or updating any root-to-leaf path in root-to-leaf or leaf-to-root order takes $\Theta(\log_B N)$ block transfers. Then, the *packed memory array* lets us store a "cache-oblivious linked list". The items of this "linked list" will represent key-value pairs, and later groups of key-value pairs. The packed memory array has fast updates in the amortized sense. Inserting and deleting items overwrites $\Theta(\frac{\log^2 N}{B})$ amortized memory blocks. Finally, we combine the two previous parts with *indirection* to obtain the desired $\Theta(\log_B N)$ time for FIND, INSERT and DELETE.

7.1 The van Emde Boas layout

The van Emde Boas layout is a way of mapping the nodes of a full binary tree of height h to indices $0 \dots 2^h - 2$. There are $N = 2^h - 1$ nodes. Other layouts of full binary trees include the BFS or DFS order.

Surprisingly, the van Emde Boas layout was not invented by van Emde Boas. The name comes from the earlier *van Emde Boas priority queue*, which was invented by a team led by Peter van Emde Boas [56]. The van Emde Boas priority queue uses a height splitting scheme similar to the van Emde Boas layout. The cache-oblivious application of this idea was devised in [25].

The advantage of storing a full binary tree in the van Emde Boas layout is that is lets us read the sequence of keys from the root to any leaf using $\Theta(\log_B N)$ block transfers, which matches the FIND cost of B-trees without the need to know B beforehand. In contrast, the same operation would cost $\Theta(\log N - \log B)$ block transfers in the BFS or DFS order.

The van Emde Boas layout is defined recursively. To find the van Emde Boas layout of a full binary tree of height h, we split the tree to bottom subtrees of height $\lfloor\!\lfloor h-1 \rfloor\!\rfloor$ and one top subtree of height $h - \lfloor\!\lfloor h-1 \rfloor\!\rfloor$. The subtrees are

[1] Strictly speaking, the bound is $\Theta(1 + \log_B N)$ because we perform $\mathcal{O}(1)$ block transfers even when $B \in \omega(N)$. We decide to use the reasonable assumption that $B < N$ to simplify some bounds in this section.

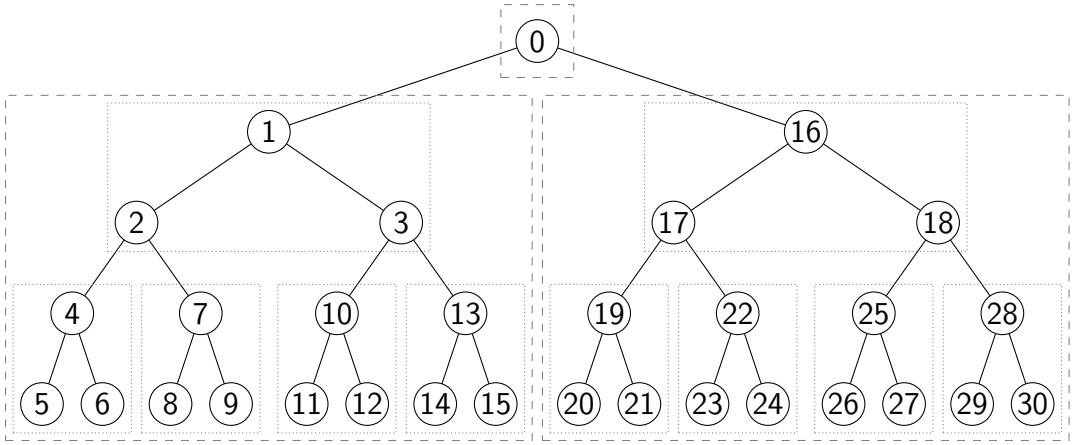


Figure 7.1: The van Emde Boas layout of a full binary tree of height 5. Boxes mark recursive applications of the construction. Note that the indices within every box are contiguous, so at some level of detail, reading a box will take $\mathcal{O}(1)$ block transfers.

recursively aligned in the van Emde Boas layout and then laid out: the top tree first, followed by the bottom trees in left-to-right order. The van Emde Boas layout of a one-node tree is trivial.

Theorem 8. *In a tree stored in the van Emde Boas layout, reading the sequence of nodes on any root-leaf path costs* $\Theta(\log_B N)$ *block transfers.*

Proof. First, consider only van Emde Boas layouts with h a power of two. Examine recursive applications of the van Emde Boas construction. The first recursion splits the tree of height h into a top tree of height $h - \lfloor\!\lfloor h-1 \rfloor\!\rfloor = h/2$ and a bottom tree of height $\lfloor\!\lfloor h-1 \rfloor\!\rfloor = h/2$. The second recursion splits all trees of height $h/2$ into trees of height $h/4$, and so on. Denote the heights of constructed bottom trees at recursion level i as $h_i = h/2^i$.

Note that at every level of recursion, every tree is stored in contiguous memory. Assume that any node can be stored in one machine word. A cache block of B words accomodates B nodes, so there is some level of recursion i for which trees of height h_i fit into a cache block, but trees of the next larger height h_{i-1} do not. In other words, $2^{h_i} - 1 < B$ and $2^{h_i - 1} - 1 \geq B$, from which $h_i = \Theta(\log B)$ follows.

Root-to-leaf paths contain $\Theta(\log N)$ nodes, and these nodes span $\Theta(\frac{\log N}{h_i})$ trees of height h_i. Since each such tree can be read using one block transfer, we need $\Theta(\frac{\log N}{h_i}) = \Theta(\log_B N)$ block transfers to traverse any root-to-leaf path.

If h is not a power of two, recursively generated top and bottom trees may no longer have heights that are powers of two. However, note that if we again divide the tree into subtrees until atomic subtrees of height h_i fit into cache blocks, almost all trees encountered on any root-to-leaf path will still have height h_i. The only exception will be the tree that contains the root, which may have height less than h_i. The original argument still works: we need $\Theta(\log_B N)$ block transfers for traversing a root-to-leaf path, including an $\mathcal{O}(1)$ added for reading the tree containing the root. □

The van Emde Boas layout thus makes a fine data structure for querying static

data, but it does not allow inserting or deleting nodes. We will need to combine it with another data structure to allow updates.

7.1.1 Efficient implementation of implicit pointers

A useful property of the van Emde Boas layout is that it is fully specified by the height h of the tree, so there is no need to keep pointers to child nodes. The positions of left and right children of a node can be calculated from h when they are needed. This is particularly useful when B is small and when the size of pointers is similar to the size of keys, because not storing pointers lets us effectively use a higher branching factor than B-trees with explicit pointers could. For example, if we store 8-byte keys in a B-tree and if we align nodes to a 64-byte cache line, each node can only store up to 3 keys. In contrast, 64 bytes of a van Emde Boas layout store 8 keys.

We will only use the van Emde Boas order to walk in the direction from the root node to a leaf or in reverse. Given the *van Emde Boas ID* of a node (denoted as numbers inside the nodes in figure 7.1), we can easily calculate the van Emde Boas IDs of its children by a recursive procedure. This procedure either returns *internal* node pointers, referencing actual nodes of the tree, or it returns *external* indices, which represent virtual nodes below the leaves, counted from left to right.

function GetChildren(n: node van Emde Boas ID, h: tree height)
  if $h = 1$ and $n = 0$ **then return** external (0,1)
  $h_\downarrow \leftarrow \lfloor\!\lfloor h - 1 \rfloor\!\rfloor$ ▷ Calculate top and bottom heights
  $h_\uparrow \leftarrow h - h_\downarrow$
  $N_\uparrow, N_\downarrow \leftarrow 2^{h_\uparrow} - 1, 2^{h_\downarrow} - 1$ ▷ Calculate top and bottom tree sizes
  if $n < N_\uparrow$ **then**
    $\ell, r \leftarrow$ GetChildren(n,$h_\uparrow$)
    if ℓ and r are internal **then**
      return internal (ℓ,r)
    else ▷ ℓ and r point to bottom tree roots
      return internal ($N_\uparrow + \ell \cdot N_\downarrow$, $N_\uparrow + r \cdot N_\downarrow$)
    end if
  else
    $i \leftarrow (n - N_\uparrow)/N_\downarrow$ ▷ The node n lies within the i-th bottom tree.
    $b \leftarrow N_\uparrow + i \cdot N_\downarrow$ ▷ b is the root of the i-th bottom tree.
    $\ell, r \leftarrow$ GetChildren($n - b$, $h_\downarrow$)
    if ℓ and r are internal **then**
      return internal ($\ell + b$, $r + b$)
    else
      $e \leftarrow 2^{h_\downarrow}$ ▷ Adjust indices by e external nodes per bottom tree.
      return external ($\ell + i \cdot e$, $r + i \cdot e$)
    end if
  end if
end function

The cost of this procedure in the cache-oblivious model is $\mathcal{O}(1)$, because it can be implemented using constant memory by modifying the tail recursion into a loop. Unfortunately, on a real computer, this is not quite the case – every call

of this procedure performs $\Theta(\log \log N)$ instructions and the cost of calling this procedure $\Theta(\log N)$ times between the root and a leaf is not negligible compared to the cost of memory transfers. Indeed, this calculation of implicit pointers can become the performance bottleneck of the cache-oblivious B-tree. This can be slightly alleviated by caching the results for trees of small height, which allows us to stop the recursion early.

As described in [9], at the low cost of precomputing $\mathcal{O}(h)$ items for every height h of the binary tree, we can perform root-leaf traversals in constant time per traversed level.

The main observation is that for any node in a fixed depth d, performing the recursive construction until the selected node is the root of a bottom tree will progress through the same sizes of top and bottom trees.

For every depth $d \in [2; h]$, let us precompute the size $B[d]$ of bottom trees rooted in depth d, the size $T[d]$ of the corresponding top tree and the depth $D[d]$ of the top tree's root. The data takes $\mathcal{O}(\log N)$ memory and it can be computed in $\mathcal{O}(\log N)$ time by an iterative procedure. Table 7.1 shows the values of these arrays for the 5-level van Emde Boas layout shown from figure 7.1.

d	$B[d]$	$T[d]$	$D[d]$
0	–	–	–
1	15	1	0
2	1	1	1
3	3	3	1
4	1	1	3

Table 7.1: $B[d]$, $T[d]$ and $D[d]$ for the 5-level van Emde Boas layout.

While traversing a root-to-leaf path, we shall maintain the depth d of the current node X, the index i of the current node in BFS order and an array $Pos[j]$ of van Emde Boas order indices of nodes passed in every depth $j < d$ during this traversal.

As the bits of the BFS index i correspond to left and right turns made during the traversal, the $\log(T[d]+1)$ least significant bits of i are the index of the unique bottom tree rooted by the node X. Because $T[d]$ is always in the form $2^k - 1$, we can find those bits quickly by calculating $i \ \& \ T[d]$.

Because the current node X is the root of the $(i \ \& \ T[d])$-th bottom tree of size $B[d]$ after a top tree of size $T[d]$ rooted in $Pos[D[d]]$, it follows that the van Emde Boas index of the current node can be calculated in $\mathcal{O}(1)$ time as:

$$Pos[d] = Pos[D[d]] + T[d] + (i \ \& \ T[d]) \cdot B[d]$$

Our root-to-leaf traversal starts by setting $i \leftarrow 0, d \leftarrow 0, Pos[0] \leftarrow 0$. Navigation to the left or right child of the current node is performed by updating the BFS index i (setting it to $2i+1$ or $2i+2$ respectively), incrementing the depth d and calculating the new $Pos[d]$. We can also return one level higher by reversing the update of i and decrementing d.

In our root-to-leaf traversals, reading the value of a non-leaf node N will be followed by reading the value of its right child R. Based on the value in R, we will then either descend further below R, or we will descent into the subtree below N's

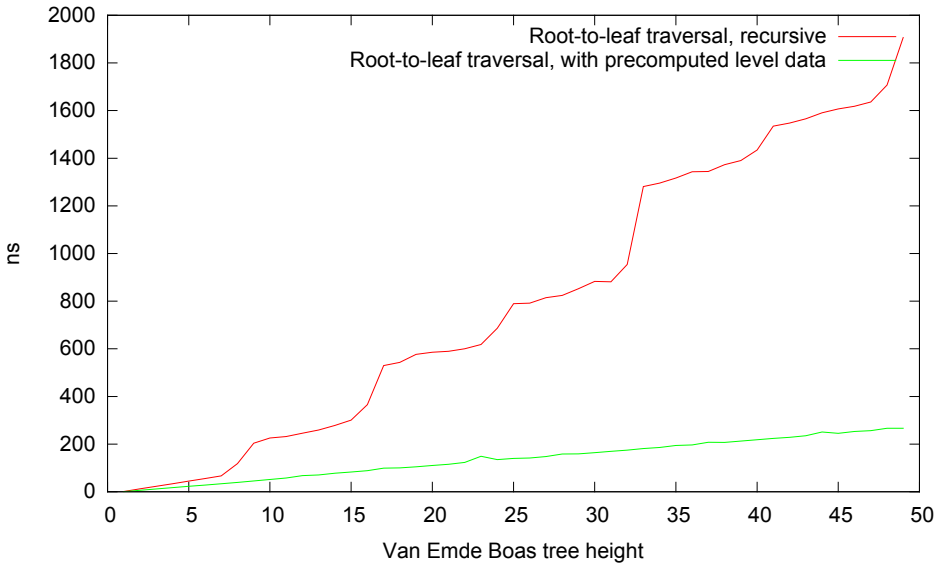


Figure 7.2: Average time for computing node pointers in root-to-leaf traversals of van Emde Boas layouts, implemented trivially and with precomputed level data.

left child L. We can slightly optimize this access pattern by calculating $Pos[d]$ for L from $Pos[d]$ for R by simply subtracting $T[d]$, because $B[d]$ will decrease by 1. This saves us one relatively expensive multiplication instruction whenever moving to the left sibling.

As evident from figure 7.2, using precomputed level data saves a considerable amount of computation, especially with deeper van Emde Boas layouts.

7.2 Ordered file maintenance

Ordered file maintenance algorithms maintain N items stored in a physical array of size $\mathcal{O}(N)$. Aside from reading from any slot in the array, we can ask the algorithm to Insert a new item before or after a given index in the array, and Delete($index$), which deletes the item at a given index. Updates are allowed to move elements of the array, but the relative order of all stored items must remain the same.

While the interface is similar to a simple linked list, storing the items in a contiguous array of size $\mathcal{O}(N)$ lets us scan a range of size k using only the optimal $\Theta(k/B)$ block reads, which improves upon linked lists by a factor of B.

To give the maintenance algorithm some freedom, empty gaps of up to a constant maximum size may be present between two adjacent items in the array. Without gaps, Insert and Delete would have to be implemented by shifting elements to the right or left in $\Theta(N/B)$ block transfers.

The *packed memory array* is a data structure that maintains an ordered file. Updates (Inserts and Deletes) of a packed memory array rewrite a contiguous range of amortized size $\Theta(\log^2 N)$. Thanks to the range being contiguous, updates will incur $\Theta(\log^2 N/B)$ amortized block transfers.

The array is divided into logical *leaf blocks* of constant size $\mathcal{O}(\log N)$. A *virtual full binary tree* is built above those leaf blocks. Every node of this binary tree represents the range covering all the leaf blocks below. This tree is only conceptual

– it is never stored in memory. The root of the virtual tree is the range covering the entire ordered file, and the children of any range are its left and right halves. For simplicity, the virtual tree assumes that the number of leaf blocks is 2^x. Any ranges that do not fall within the actual backing array are ignored.

To properly describe the structure, we need to define some terms. The *capacity* of a certain range of the array is its size and the *density* of a range is the number of items present in the range divided by its capacity. The packed memory array maintains certain bounds on the densities of subranges of the array.

The densities in the leaf blocks are kept between 1/4 and 1. If an update keeps the density of the leaf block within thresholds, we perform the update by rewriting all $\Theta(\log N)$ items of the leaf block. When the range covering the leaf block becomes too sparse or too dense, we walk up the virtual tree of ranges until we find a node that fits our density requirements. The parent range is obtained by doubling the size of the child range, extending the child on the left or on the right (depending on whether the child range was a right or left child of its parent range).

The density requirements become stricter on larger ranges. In particular, the density of a node of depth d within a tree of height h is kept between $\frac{1}{2} - \frac{1}{4}\frac{d}{h}$ and $\frac{3}{4} + \frac{1}{4}\frac{d}{h}$.[2] When we find a node that is *within threshold*, we uniformly redistribute the items in its range. If no such node exists, we resize the entire structure by a constant factor, which lets us amortize the costs of this resizing to a multiplicative constant.

We claim that while this redistribution may reorganize a range of size up to $\mathcal{O}(N)$, the redistribution of a node puts all nodes below it well within their thresholds, so the next reorganization will be needed only after many more updates.

The search for a node to redistribute is implemented as two interspersed scans, one extending the scanned range to the left, one to the right. The redistribution can be done by collecting all items in the array on the left side using a left-to-right scan, followed by a right-to-left scan putting items into their final destinations. Thus, redistributing a range of K items takes $\Theta(K/B)$ block transfers.

Theorem 9. *The block-transfer cost of an update of the packed-memory array is* $\Theta(\frac{\log^2 N}{B})$ *amortized.*

Proof. Suppose we need to redistribute some node X of depth d after performing an insertion. Since we are redistributing this node, it is within threshold, but one of its children, say Y, is not. Therefore the density of X is at most $\frac{3}{4} + \frac{1}{4}\frac{d}{h}$, while the density of Y is more than $\frac{3}{4} + \frac{1}{4}\frac{d+1}{h}$. After we redistribute the items within X, the density of Y will drop by $1/h$. Thus, if we denote the capacity of Y as $|Y|$, we will need to insert at least $|Y|/h$ items under any child of X before we need to rebalance X again. Thus, we can charge the $\mathcal{O}(|X|)$ cost of redistributing X to insertions into Y, which gives us amortized $\mathcal{O}(\log N)$ redistribution steps per insertion into Y.

Since the node Y has $h = \Theta(\log N)$ ancestors, we can amortize $\Theta(\log N \cdot \log N)$ redistribution steps per insertion, which is $\Theta(\frac{\log^2 N}{B})$ in block transfers. The proof of the deletion cost is analogous. □

[2]The constants are arbitrary and control the tradeoff between memory consumption and time complexity of rebuilding.

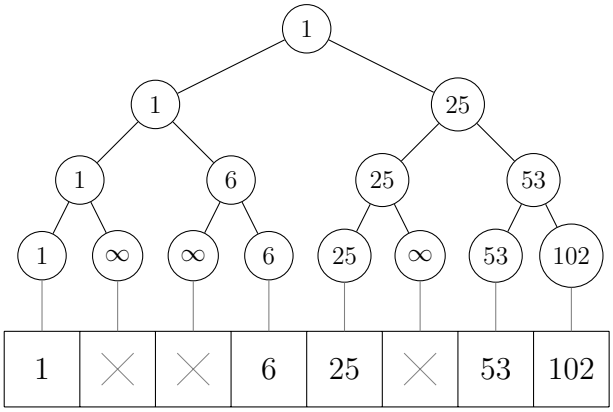


Figure 7.3: The bootstrapping structure of the cache-oblivious B-tree

7.3 Cache-oblivious B-tree

The first step of constructing a cache-oblivious B-tree is a bootstrapping structure, which is a combination of a full binary tree in the van Emde Boas layout and the packed memory array. The packed memory array stores inserted key-value pairs sorted by the key. Nodes of the tree map to ranges in the array, with leaves mapping to individual slots and the root mapping to the entire ordered file. The nodes of the tree contain dictionary keys. Leaves store the keys stored in their corresponding ordered file slots, or ∞ if the slot is empty. Each internal node of the tree stores the minimum of its subtree.

FINDing a key in the bootstrapping structure is done by binary search on the tree. The binary search either finds the ordered file slot which contains the key-value pair, or it finds an empty slot. Either way, walking down the tree costs $\Theta(\log_B N)$ block transfers.

INSERTs and DELETEs first walk to the position the key would normally occupy, using $\Theta(\log_B N)$ block transfers. The key is then inserted into the ordered file, which costs $\Theta(\frac{\log^2 N}{B})$ amortized block transfers. Finally, the tree needs to be updated to reflect the reorganization of the packed memory array.

Theorem 10. *Updating K consecutive leaves of a tree in van Emde Boas order takes $\mathcal{O}(\frac{K}{B} + \log_B N)$ memory transfers.*

Proof. To propagate new minima in the tree, we need to update the K changed leaves and their parents. To get the right memory transfer cost, we update the tree in-order: if node x has children y and z which both need to be updated, we visit them in this order: x, y, x, z, x.

We need to update K leaves with new values and their parents. Just as in the analysis of root-to-leaf traversal cost in van Emde Boas trees, we look at the level of recursion where the atomic trees start fitting into cache blocks. Call the largest tree units that fit into cache lines *atoms* and the next larger units *chunks*.

Consider first the bottom two levels of atoms. Within every large chunk, we jump between two atoms (a parent and one of its children), which fit into a cache block. If we have at least 2 blocks in the cache, updating any chunk will cost $\mathcal{O}(C/B)$ memory transfers, where C is the size of a chunk. Thus, we can update the bottom two levels using $\mathcal{O}(K/B)$ memory transfers.

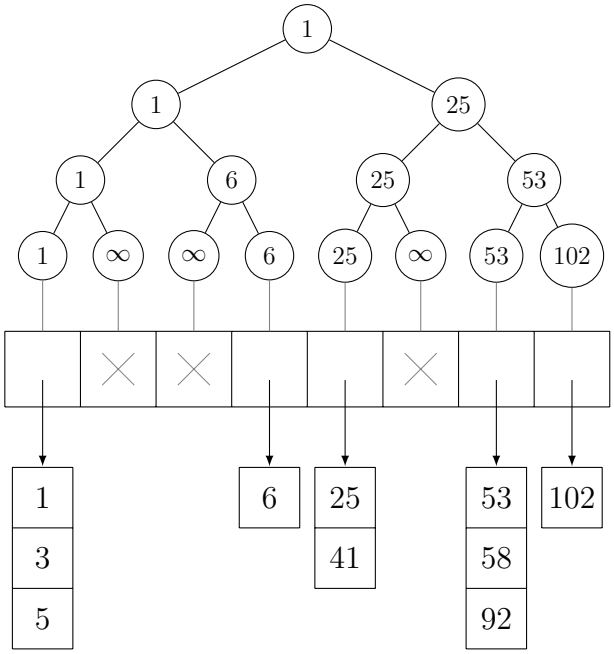


Figure 7.4: Full cache-oblivious B-tree with indirection

Next, let us separately consider updating the tree below and above the lowest common ancestor of the K. Updating all nodes above the lowest common ancestor means simply walking a root-to-node path, which takes $\mathcal{O}(\log_B N)$ transfers.

Above the bottom two levels, there are $\mathcal{O}(K/B)$ nodes to update. We can afford to spend an entire block transfer for updating each of these nodes.

We spend $\mathcal{O}(K/B + \log_B N)$ memory transfers in total. □

Thus, the block transfer cost of updates is $\Theta(\log_B N + \frac{\log^2 N}{B})$, which is $\Theta(\frac{\log^2 N}{B})$ more than B-trees. FIND already matches B-trees with $\Theta(\log_B N)$ block transfers. The bootstrapping structure is as good as B-trees for $B = \Omega(\log N \log\log N)$. Unfortunately, this is not a realistic assumption in internal memory where cache lines are relatively small, so we need to remove the $\Theta(\frac{\log^2 N}{B})$ term from updates to fully match B-trees.

To remove this term, we will use this bootstrapping data structure with *indirection* to make the final cache-oblivious B-tree.

The final *cache-oblivious B-tree* will store key-value pairs partitioned into *pieces* in disjoint intervals. A piece is an array of $P = \Theta(\log N)$ slots, which contains between $P/4$ and P key-value pairs. Since pieces are physical arrays, reading or fully rewriting a piece takes $\Theta(\frac{\log N}{B})$ block transfers. The *bootstrapping structure* stores pointers to the $\Theta(N/\log N)$ pieces as values, keyed by their minimal keys.

To FIND a key in the cache-oblivious B-tree, we walk down the bootstrapping structure in $\Theta(\log_B(N/\log N)) = \Theta(\log_B N)$ to find the piece that may contain the key, and scan it in $\Theta(P/B) = \Theta(\log N/B)$ transfers, so we still need only $\Theta(\log_B N)$ block transfers for FINDs. FINDNEXT and FINDPREVIOUS work similarly.

INSERTs and DELETEs first find the appropriate piece to update, for which they need $\Theta(\log_B(N/\log N))$ block transfers. If the piece can be updated without getting under $P/4$ or over P items, we rewrite the piece in $\Theta(\frac{\log N}{B})$, removing or

inserting the key. If we changed the minimum of the piece, we also need to walk up the tree in $\Theta(\log_B(N/\log N))$ to propagate the change.

If the piece is too full, we split the piece into two pieces of size $P/2$. The splitting takes $\Theta(\frac{\log N}{B})$ block transfers. Afterwards, the new piece needs to be inserted to the bootstrapping structure in $\Theta(\frac{\log^2(N/\log N)}{B})$ amortized transfers. Similarly, if the piece is too sparse, we merge the piece with one of its neighbors. A neighbor can be found in $\mathcal{O}(1)$, because the pieces are stored in the packed memory array, which guarantees gaps of constant size. If the two pieces have more than $3P/4$ items, we only "borrow" some keys from the neighbor and update the bootstrapping structure in $\Theta(\log_B N)$ to reflect new piece minima. If there are less than $3P/4$ items in total, the pieces get merged and one of them will be deleted from the bootstrapping structure in $\Theta(\frac{\log^2(N/\log N)}{B})$.

Thus, updates take $\Theta(\log_B N)$ plus $\Theta(\frac{\log^2 N}{B})$ every time we need a split or a merge. However, we don't need splits and merges too often: splitting a full piece creates two pieces of size $P/2$, which will be only split or merged again after $\Theta(P)$ more updates. Merging two pieces yields a piece with between $P/2$ and $3P/4$ items. This new piece will also be updated only after $\Theta(P)$ more updates. Thus, we can charge each update $\Theta(\frac{1}{\log N}\frac{\log^2 N}{B})$ for splits and merges, which gives us $\Theta(\log_B N + \frac{\log N}{B}) = \Theta(\log_B N)$ amortized time for updates.

Finally, updates may necessitate a change in piece size P. If P no longer fits, we pick a new P by adding or subtracting a small constant Δ_P and we globally rebuild the structure. Because piece sizes need to change after the data structure increases or decreases in size by a factor of 2^{Δ_P}, we can allow rebuilds to take up to $\Theta(N \log_B N)$ block transfers – the amortized cost of rebuilds will be $\mathcal{O}(1)$ per item.

7.4 Enhancements

An alternative deamortized ordered file maintenance data structure developed by [61] performs $\Theta(\frac{\log^2 N}{B})$ block transfers in the worst case. If we momentarily disregard the cost of rebuilding when N expands or shrinks by a constant factor, a cache-oblivious B-tree backed by this structure would limit the worst-case number of block transfers from $\Theta(\frac{N}{\log N})$ to just $\Theta(\frac{\log^2 N}{B} + \log_B N)$ while keeping the amortized bound at $\Theta(\log_B N)$.

As we observed with applying the van Emde Boas layout, removing auxiliary data can greatly speed up operations – the tree in van Emde Boas layout has half the effective depth of a B-tree, because cache lines can fit more keys when pointers are implicit. *Implicit data structures* are designed to waste as little memory on auxiliary information as possible. Implicit dictionaries are allowed to only store a permutation of the stored data and $\mathcal{O}(1)$ memory cells of additional information. Bookkeeping information, such as pointers or small integers of b bits, is usually encoded by a pairwise permutation of $2b$ keys, with $x_{2i} < x_{2i+1}$ corresponding to 1 and $x_{2i} > x_{2i+1}$ corresponding to 0.

An overview of progress on implicit dictionaries is given in [23]. The optimal bound of worst-case $\Theta(\log_B N)$ block transfers per FIND or update has been

reached in an implicit setting in [22]. Similarly to the simple cache-oblivious B-tree, the construction uses the deamortized version of the packed memory array in conjunction with the van Emde Boas layout. To avoid holes in the packed memory array, the elements of the packed memory array are chunks of keys. Several other modifications are necessary to allow updates in size, which requires building the structure on the fly. We decided not to implement the implicit cache-oblivious B-tree due to its large complexity. We also believe the data structure may be much slower on practical data sets than equivalent structures with free slots or pointers, since decoding auxiliary data from the implicit representation requires reading large permutations of keys.

8. Implementation

To measure the performance of various data structures, we implemented them in the C programming language (using the C11 revision). We chose this language for several reasons:

- The language is low-level, which enables a high degree of tuning by hand. Our data structures are also not slowed down by common features of high-level languages, such as garbage collection.
- C toolchains are mature and they can be expected to optimize code well.
- The C language is very portable and most other languages can call C functions. This enables our data structures to be potentially easily used in other projects.

Our code was developed using the GNU C Compiler (version 4.9.2) on Linux. We used the Git version control system, with our repository hosted at `https://github.com/MichalPokorny/thesis`. The code includes a test suite, which tests implementation details of particular data structures as well as general fuzz tests for checking that implemented data structures behave like proper dictionaries. We used Travis CI (`https://travis-ci.org/`) to automatically test our code after submission.

All dictionaries assume 64-bit keys and values (represented as the `uint64_t` type). It would be easy to allow keys and values of arbitrary size, but such a choice would be likely to slow down operations on the data structures, because a range of compiler optimizations would be impossible in code assuming general lengths. For example, copying keys and values would need to be a general loop or `memcpy`/`memmove` call, while assuming a constant key/value size of 64 bits lets us copy in one CPU instruction. Using the C++ language, we could potentially use templates to make our data structures both general and optimized. If we are willing to sacrifice some code style, we could make the C code behave similarly by moving the implementation into a header file, which could be configured to generate a general data structure tuned to specific parameters prior to inclusion, as in the following example:

```
struct my_value {
        uint64_t stuff[32];
};
#define IMPLEMENTATION_PREFIX myimpl
#define IMPLEMENTATION_KEY_TYPE uint64_t
#define IMPLEMENTATION_VALUE_TYPE struct my_value
#include "btree/general.impl.h"

void usage() {
        dict* btree;
        dict_init(&btree, &dict_myimpl_btree);
        // ...
}
```

For example, the LibUCW library (`http://www.ucw.cz/libucw/`) uses this pattern for generic binomial heaps and hash tables. We have decided to forgo generality to keep the code clean and readable.

8.1 General dictionary API

All implemented data structures can be used through a common API defined in `dict/dict.h`. To encapsulate implementation details from users of the API, we have used a common C "trick", where we represent dictionaries as "virtual table pointers" and "private data". The common API represents a dictionary as a pointer to an opaque structure. The type pointed to is declared in `dict/dict.h` as:

```
typedef struct dict_s dict;
```

The `struct dict_s` structure is defined in `dict/dict.c` to prevent exposure of implementation details from users including `dict/dict.h`:

```
struct dict_s {
        void* opaque;
        const dict_api* api;
};
```

The `opaque` pointer is maintained by the concrete implementation. `dict_api` is a "virtual method table" type defined in `dict/dict.h`:

```
typedef struct {
        int8_t (*init)(void**);
        void (*destroy)(void**);

        bool (*find)(void*, uint64_t key, uint64_t *value);
        bool (*insert)(void*, uint64_t key, uint64_t value);
        bool (*delete)(void*, uint64_t key);

        const char* name;

        // More function pointers.
} dict_api;
```

The `init` and `destroy` functions are passed a pointer to the `opaque` pointer and they respectively initialize and deinitialize it. The `name` field is a human-readable name of the data structure. Other fields are given the `opaque` pointer as the first argument and they act as implementations of their respective general versions operating on general dictionaries declared in `dict/dict.h`:

```
bool dict_find(dict*, uint64_t key, uint64_t *value);
bool dict_insert(dict*, uint64_t key, uint64_t value);
bool dict_delete(dict*, uint64_t key);
```

Each of the functions above returns `true` on success and `false` on failure. In particular, `dict_find` and `dict_delete` fail if the dictionary does not contain the `key` and `dict_insert` fails if the dictionary already contains the inserted `key`.

Bigger problems, such as internal inconsistencies or failures to allocate memory are signalled by logging a message and exitting the program. While such resignation to error handling is not appropriate in dependable software, our code is indended only for experimental purposes, where fatal failure is acceptable. A more robust implementation would instead react to failure by freeing temporary resources, rolling back any changes and returning an error code.

Dictionary data structures may optionally implement successor and predecessor lookups. The public API consists of the functions `dict_next` and `dict_prev`.

If the data structure does not implement these operations, it may set the appropriate fields in `dict_api` to `NULL`. The `dict_api_allows_order_queries` and `dict_allows_order_queries` functions detect whether ordered dictionary operations are available for the given implementation or dictionary.

```
bool dict_api_allows_order_queries(const dict_api*);
bool dict_allows_order_queries(const dict*);
bool dict_next(dict*, uint64_t key, uint64_t *next_key);
bool dict_prev(dict*, uint64_t key, uint64_t *prev_key);
```

If we iterate over a dictionary in key order, many data structures can benefit from keeping a "finger" on the last returned key-value pair. For example, if we enumerate the keys of a B-tree in order without auxilliary data, we need to perform $\mathcal{O}(\log N)$ operations on every query. This can be easily improved to $\mathcal{O}(1)$ amortized by stashing the last visited node between queries. For simplicity, we did not implement order queries with a finger. While this makes our data structures slower on key enumerations, we believe the penalty is roughly similar on all data structures, so our results should still be representative enough.

Every data structure implementation exposes a global variable of type `const dict_api` named `dict_mydatastructure`. For example, a B-tree can be used as follows:

```
#include <inttypes.h>
#include <stdio.h>

#include "dict/btree.h"
#include "dict/dict.h"

void example_btree(void) {
        dict* btree;
        dict_init(&btree, &dict_btree);
        if (!dict_insert(btree, 1, 100)) {
                printf("Error␣inserting␣1␣->␣100.\n");
        }
        // ...
        const bool found = dict_find(btree, 42, &value);
        if (found) {
                printf("Found␣42␣->␣\%" PRIu64 ".\n",
                                value);
        } else {
                printf("No␣value␣for␣42.\n");
        }
        // ...
        if (!dict_delete(btree, 1)) {
                printf("Failed␣to␣remove␣key␣1.\n");
        }
        dict_destroy(&btree);
}
```

This approach requires an indirect jump through the "virtual method table" whenever we write implementation-agnostic code. Removing this indirect jump by directly compiling against a particular implementation is likely to slightly speed up the code. On the other hand, our tests are usually long-running, so in the typical case, the target of the indirect jump will be cached, which should imply a very low overhead. Even if this indirect jumping were expensive, it would

not significantly affect our results – every data structure under test is invoked through indirect jumping, so the performance penalty is the same on every data structure. While absolute data structure speed may be impacted by this design decision, relative differences between data structures should stay the same with a virtual call or without.

Finally, we have decided to reserve one key for internal purposes. This key is defined in `dict/dict.h` as the macro `DICT_RESERVED_KEY` and its value is `UINT64_MAX`. Inserting this key into a dictionary or deleting it will result in an error. Reserving this value is useful for representing unused slots in nodes of B-trees or k-splay trees.

We implemented the following dictionary data structures:

- `dict_array`, declared in `dict/array.h`: a sorted array of key-value pairs with $\mathcal{O}(N)$ insertions and deletions, indended as a benchmark on very small inputs.

- `dict_btree`, declared in `dict/btree.h`: a B-tree. The value of b may be selected by changing compile-time constants. Our eventual choice of b is described in section 9.1.

  In the variant that aligns b to fit one node exactly into a 64 B cache line, `dict_btree` is, in fact, a 2-3-4 tree. This is a slight relaxation of B-trees (chapter 3) that allows inner nodes to contain $k \in [1; 3]$ keys. An inner node needs to keep k keys and $k + 1$ pointers, so, with 64-bit key and pointers and 64 B cache lines, an inner node fitting into a cache line can keep at most 3 keys. However, leaves hold k keys and k values of 64 bits each, which lets us put up to 4 key-value pairs into each leaf.

- `dict_cobt`, declared in `dict/cobt.h`: a cache-oblivious B-tree (chapter 7).

- `dict_htcuckoo`, declared in `dict/htcuckoo.h`: cuckoo hash table with simple tabulation hashing (section 2.4).

- `dict_htlp`, declared in `dict/htlp.h`: an open-addressing hash table with linear probing and simple tabulation hashing (section 2.3).

- `dict_rbtree`, a red-black tree. We simply wrap an implementation from LibUCW 6.4 (`http://www.ucw.cz/libucw/`). This structure is included as a representative of balanced trees without cache-optimized design.

- `dict_splay`, declared in `dict/splay.h`: a splay tree (chapter 4).

- `dict_ksplay`, declared in `dict/ksplay.h`: a k-splay tree (chapter 5).

8.2 Performance measurement

Our goal is to have fast data structures with reasonable memory usage. We measure the performance of the various dictionary implementations on several experimental setups.

We tracked the time it took to conduct each experiment using the POSIX function `clock_gettime`. To better understand the bottlenecks, we also tracked

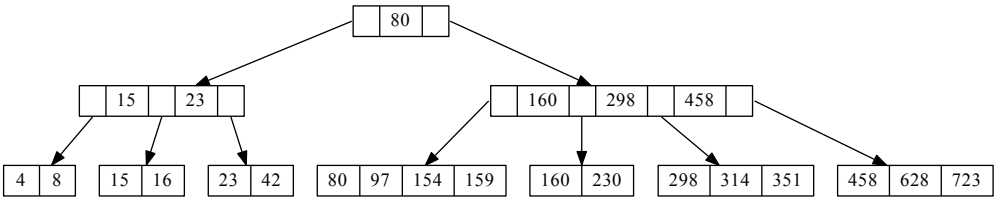


Figure 8.1: A 2-3-4 tree with bigger leaves (one configuration of `dict_btree`)

certain hardware events of interest using the `perf_events` API of the Linux kernel. This API is a generic wrapper around platform-specific performance counters. Events such as cache misses or branch mispredictions increment these counters. Details about performance monitoring on Intel CPUs can be found in chapters 18 and 19 of the Intel 64 and IA-32 Architectures Software Developer's Manual[31].

8.3 Synthetic experiments

Each synthetic experiment setup has a tunable *size* S (i.e., the maximum number of stored key-value pairs needed by the experiment). The synthetic setups include:

- *Random inserts and finds*: We generate pseudorandom key-value pairs by applying two deterministic invertible functions $k(i)$ and $v(i)$ for i between 0 and $S - 1$. These simple functions use a Feistel network[1] to provide "sufficiently randomly behaving" key-value pairs while letting us derive i from $k(i)$ or $v(i)$, which aids debugging.

  After the dictionary is seeded, we read items from it in random order. Because the difference between speeds of successful and unsuccessful FINDs is significant on some dictionaries, a fraction of reads will request nonexistant keys. This fraction is set per-experiment.

  On this experiment, we separately measure the performance of insertions and finds.

- *Left-to-right scans*: We seed the structure with S items as above. Then we traverse the dictionary from the minimum key to the maximum key, reading the values as we go. We don't include the seeding in our measurements. Left-to-right scans are only available on implementations that allow predecessor and successor queries.

- *Working set patterns*: The working set experiment needs a fixed integer parameter $W < S$. We again randomly seed the dictionary with S items. Then, S times, we access the key-value pair with key $k(i)$, where i is picked pseudorandomly from $\{0, \ldots, W - 1\}$. This access pattern is intended to simulate datasets which have a mostly-static non-uniform access probability distribution.

[1] A Feistel network is a simple construction for building symmetric ciphers from one-way functions. A treatment of several ciphers based on Feistel networks is included in [48].

- *Word occurrence counting*: We load a large text document and we tokenize it into words. Each word is then normalized to lowercase and transformed into a 64-bit integer via a simple hash function h.[2] We create a dictionary storing counts of word occurrences keyed by word hashes. Each word w is inserted into the dictionary by either inserting a new $\langle h(w), 1\rangle$ pair, or deleting an existing pair for $h(w)$ and inserting a new one with an incremented value. This usage pattern approximates building a search index on the document. [3]

The code of synthetic experiments is located in `experiments/performance`.

8.4 Non-synthetic experiments

A survey of the performance of different binary search tree implementations on concrete real-life workloads was presented in [44]. In particular, splay trees were reported to be several times faster than AVL or red-black trees when maintaining virtual memory areas of the Mozilla browser, VMware or the Squid HTTP proxy. The author also simulated the workload of the mapping from peer IP addresses to IP datagram identification nonces. On this workload, splay trees underperformed balanced search trees.

To investigate the practical performance of our data structure implementations, we decided to collect recordings of dictionary access patterns in real programs and to collect performance metrics on replays of these recordings.

8.4.1 Mozilla Firefox

The Mozilla Firefox web browser contains an unordered dictionary template class named `nsTHashtable` in `xpcom/glue/nsTHashtable.h`. The default implementation uses double hashing. The template class implements the FIND, INSERT and DELETE operations. Additionally, it supports enumerating all key-value pairs in the structure in an arbitrary order.

We placed simple logging into the implementations of FIND, INSERT and DELETE. For simplicity, we did not log enumerations. Except splay trees, every data structure we implemented can be extended to implement arbitrary-order enumerations in $\mathcal{O}(N/B)$ block transfers, so we believe the effect of enumerations on the relative performance differences between data structures would be relatively small.

After a browsing session, we counted the number of operations logged on every `nsTHashtable` instance and we selected the top 10 instances with most operations.

Unlike our framework, `nsTHashtable` allows arbitrary-size keys and values. We did not record stored values, as this experiment was only indended to gather practical access patterns, which depend only on keys. For hash tables with keys smaller than 64 bits, we simply zero-padded the keys for our data structures.

[2] We use the 64-bit version of the FNV-1 hash function (`http://www.isthe.com/chongo/tech/comp/fnv/`).

[3] We used *The Complete Works of William Shakespeare* from Project Gutenberg (`http://www.gutenberg.org/ebooks/100`).

Bigger keys were compressed by applying the FNV-1 hash function to individual octets.

Presumably for the sake of optimization, `nsTHashtable` has a more liberal interface than we implemented. Aside from searches and updates, `nsTHashtable` also supports enumerating all key-value pairs in an arbitrary order (i.e., depending on the internal hash function). The user is also allowed to update values in enumerated key-value pairs at no extra FIND cost. We skipped arbitrary enumeration when replaying recorded operations. Because further calls to `nsTHashtable` may depend on the arbitrary order of enumeration, we expect that `nsTHashtable` would behave better on recorded operations than any of our structures. For example, since `nsTHashtable` will enumerate keys in the order in which they are stored within `nsTHashtable`, `nsTHashtable` will reap the rewards of cache friendliness, while our data structures will be much more likely to read from unpredictable locations.

Fairly comparing data structures in the presence of arbitrary-order enumeration would require actually running our source code within the instrumented program. We did not pursue this direction further in this thesis, but benchmarking data structures directly where they are used may provide inspiration for better optimizations. Particularly interpreters of dynamic programming languages with built-in dictionary types, like Python or Ruby, could prove interesting test subjects.

Our solution to recording the operations was very simple: we applied a small patch to `nsTHashtable` that printed out operations to standard output, where we collected them and grouped them by the `this` pointer. To further simplify the patch, we made no attempt to synchronize output, and we processed the output by a simple script that removed any lines that could not be successfully parsed. The fraction of removed unparseable lines was approximately 23%.

8.4.2 Geospatial database

To test the performance of FINDNEXT and FINDPREVIOUS, we simulated the workload of a simple database of geospatial data supporting local searching. The source code for this experiment is stored in `experiments/cloud`. It can be built by running `make bin/experiments/cloud`. Before running the experiment, input data must be downloaded by running `./download.sh` from within `experiments/cloud`.

We used data from *Extended Edited Synoptic Cloud Reports from Ships and Land Stations Over the Globe, 1952-2009 (NDP-026C)* [28]. The main part of this dataset consists of 196 compressed plaintext files containing one cloud report per line. Aside from many other fields, each report contains the date and UTC hour, the longitude and latitude of the reporting ship or land station and the measured sea level pressure and wind speed. Our simulated database loads all air temperature and wind speed measurements and indexes them by their coordinates. Given a position on the globe, the database can return a set of reports taken close to the position.

To implement this database in the context of our general dictionary API, we map from latitude-longitude pairs to dictionary keys via a locality-preserving mapping called the *Z-order* (or *Morton order* after its inventor, who introduced it

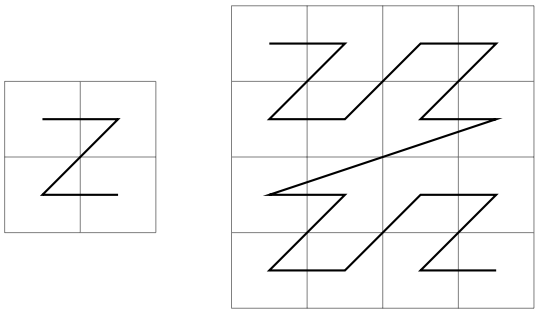


Figure 8.2: Z-order in two dimensions

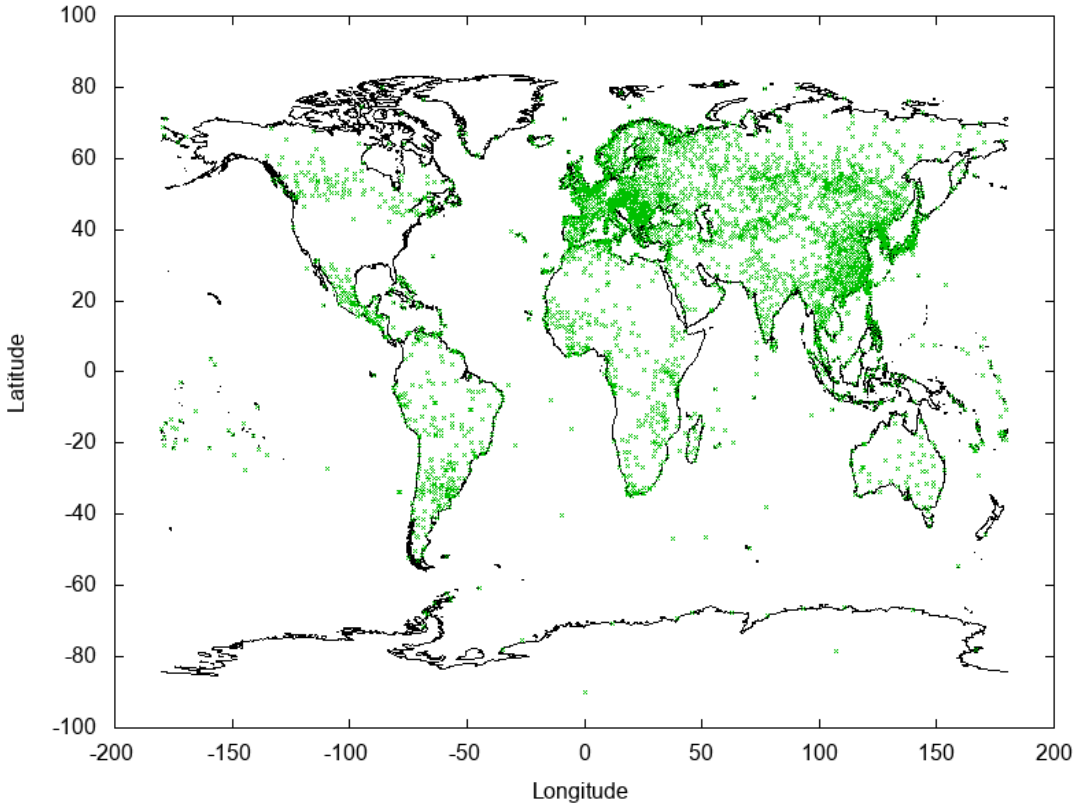
in [39]). The Z-order computes a one-dimensional key from a position in multiple dimensions simply by interleaving the bits of the coordinates. One-dimensional distances implied by the Z-order only approximate the original two-dimensional distances, so the Z-order is not useful for exact closest-point queries – we only use it as an approximation. For the sake of simplicity, we also interpret latitude-longitude angles as plane coordinates, without any attempt to merge 180° W and 180° E.

Because stations at one position usually give multiple reports, the key for our dictionary also needs to contain a unique identifier of the report, which we create by counting the reports from each unique position and append after the Z-order code of the position. Some stations also do not provide the wind speed or air pressure. Records from stations which provide neither are not loaded, and records with missing values are also skipped at query time.

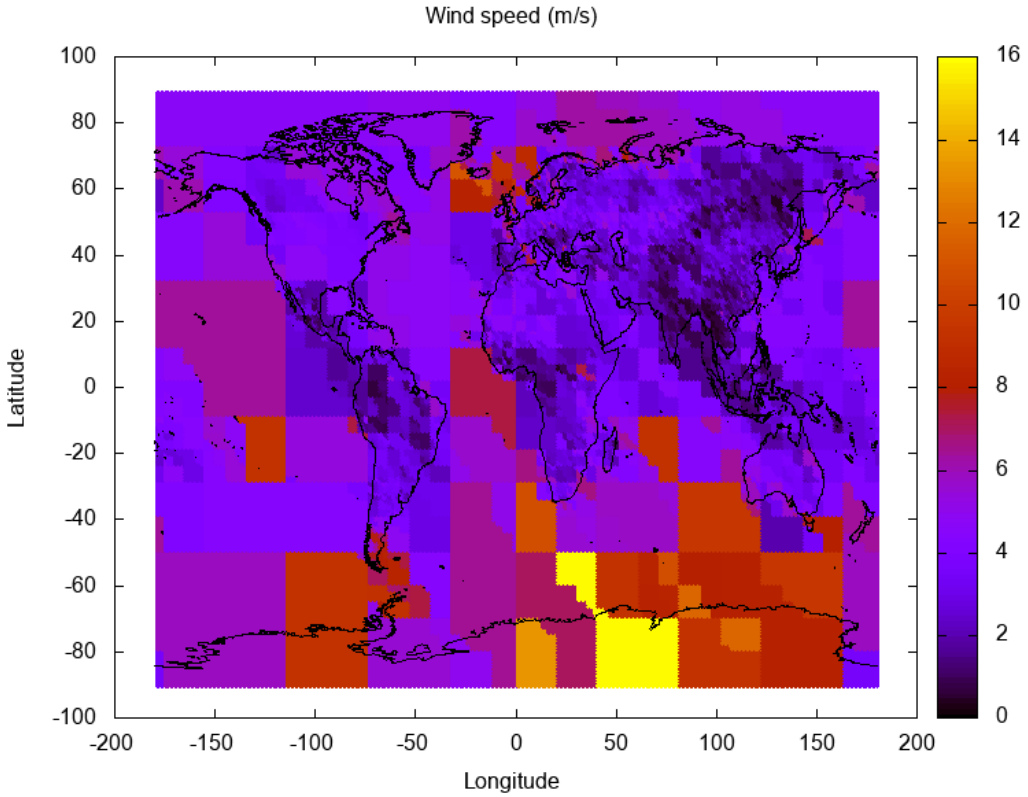
A possible alternative would be using a *Peano curve* (introduced in [43], later generalized into *Hilbert curves* of arbitrary dimension by [29]). A desirable property of Peano curves is that if two-dimensional points x and y map to neighbouring images in the one-dimensional space, then x and y were neighbours in the two-dimensional space. As apparent from figure 8.2, this does not hold in the Z-order. We decided to use the Z-order for the simplicity of its mapping function.

Queries are implemented simply by mixed calls to `dict_prev` and `dict_next` until we collect enough results. A non-toy database might perform some additional filtering to ensure only reports within a certain distance are returned.

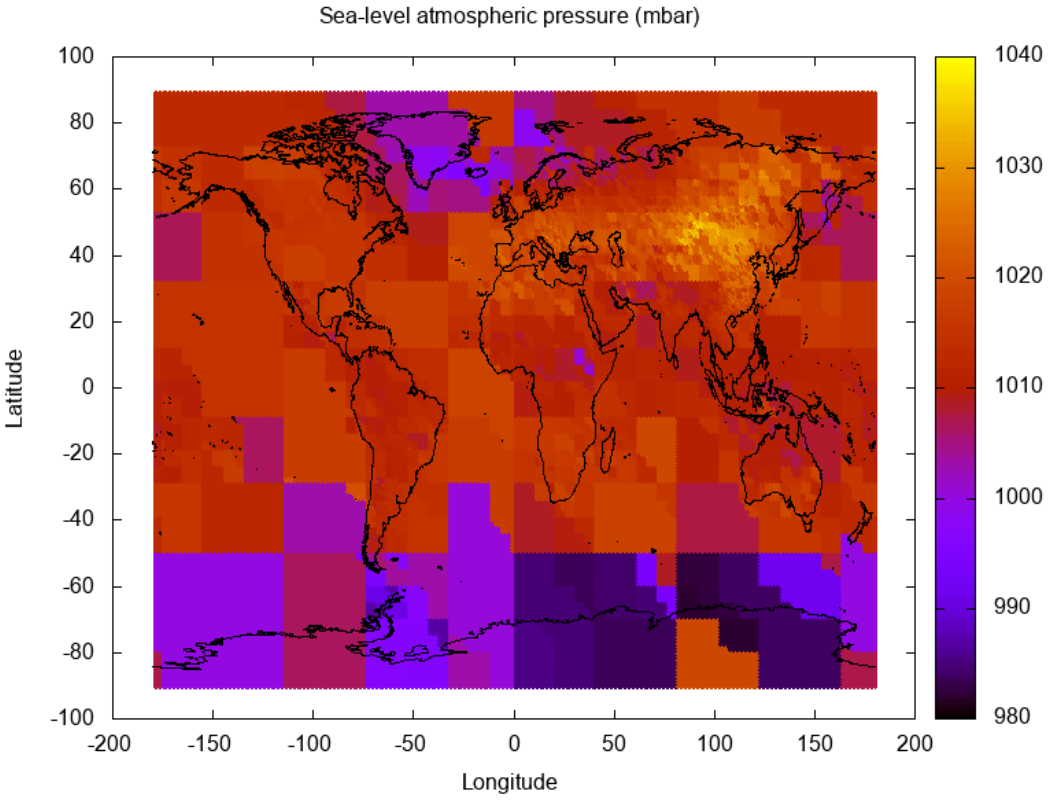
Our usage of this simulated database will pick S random points on the globe. For each point, we fetch R close records and we calculate the average recorded sea-level atmospheric pressure and wind speed in these records. Atmospheric pressures and wind speeds are packed into 64-bit values, which are stored in the ordered dictionary.



(a) Stations providing wind speed or air pressure data



(b) Average wind speeds



(c) Average sea-level atmospheric pressure

Figure 8.3: Results from the cloud data experiment. Each query sampled 1000 close measurements.

9. Results

9.1 B-trees and the choice of b

As noted in chapter 8, our implementation of cache-aware B-trees (`dict_btree`) was written for generic bounds on node capacities. We experimented with several values of the B-tree parameter b, which is tied to the size of a node. B-trees with large values of b are shallow, so a search for a key follows fewer pointers, but on the other hand, reading large nodes requires more slow memory transfers. The results are presented in figure 9.1.

On dictionaries larger than 10^5 keys, a node size of 256 B performed the best. However, on smaller dictionaries, it appears that 128 B might be a slightly better choice. For all further experiments, we fixed the node size to 256 B (which yields at most 15 keys per internal node).

Our code for B-trees uses linear search when looking within nodes, as we originally did not expect the need for large nodes. Since one node of 256 B spans 4 cache lines, binary searching within nodes might require fewer memory transfers, and hence be faster.

9.2 Basic cache-aware and unaware structures

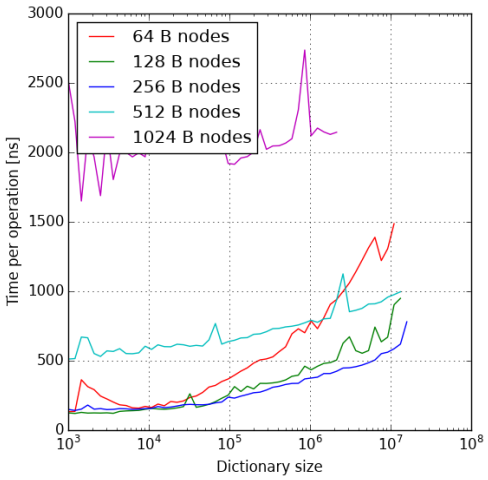
To assess the benefits of optimizing for cache hierarchies, we compared our optimized B-trees to simpler, cache-unaware dictiories: red-black trees borrowed from LibUCW and a simple sorted array with binary search and $\mathcal{O}(N)$ updates. Note that in the external memory model, a binary search needs $\Theta(\log N - \log B)$ block transfers, which is asymptotically larger than the $\Theta(\log_B N)$ required by B-trees. Thus, theoretically lookups on B-trees should be more efficient than binary searches on nontrivially large dictionaries.

Figures 9.2 and 9.3 present the results. The slow speed of operations in red-black trees (compared to binary search and B-trees) is unsurprising. It was, however, unexpected to find that lookups in B-trees start to overtake binary search only on very large dictiories (with at least 10^7 keys).[1] Let us repeat here that B-tree FINDs might be sped up by replacing linear scans within B-tree nodes by binary search and repeating the process of selecting a good value of b.

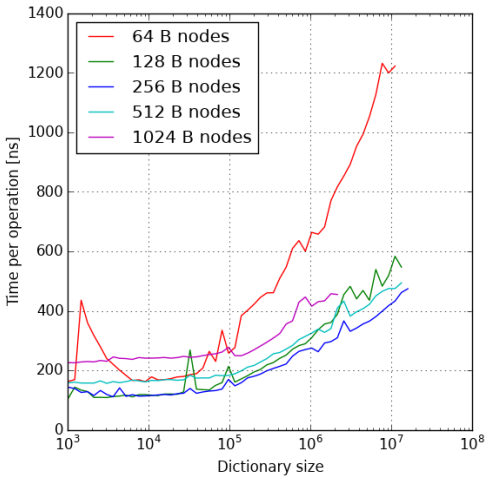
9.3 Cache-oblivious B-tree

A comparison between hand-optimized B-trees and cache-oblivious B-trees is presented in figure 9.4. Overall, cache-oblivious B-trees roughly matched the performance of cache-aware B-trees. The speed of random FINDs was almost the same

[1] Originally, we fixed the node size of our B-trees to 64 B, thinking it was the obviously correct choice (i.e., the size of a cache line). Such B-trees (or, to be more precise, 2-3-4 trees) were only slightly faster than red-black trees, and much slower than simple binary search. These findings prompted our suspicions and led us to trying other node sizes. As figure 9.1 illustrates, our initial choice was actually particularly unlucky. The moral of this short story is that optimizing for modern machines is harder than it may seem.



(a) Random FINDs, 50% successful (other success rates produce similar results)



(b) Random INSERTs

Figure 9.1: The effect of different choices of b on the performance of B-trees.

in both structures. While cache-aware B-trees did slightly better on working set access patterns, cache-oblivious B-trees had faster FINDNEXT and FINDPREV operations. INSERT operations were about 1.5× slower in cache-oblivious B-trees.

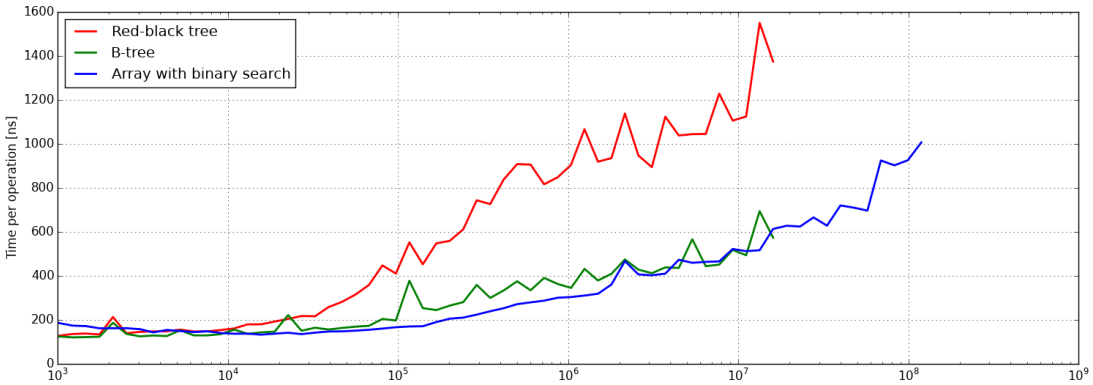
9.4 Self-adjusting structures

Figures 9.5 and 9.6 show a benchmark of our implementations of self-adjusting dictionaries: splay trees, k-splay trees and k-forests. Both figures include B-trees as a benchmark and red-black trees as a representative of cache-unaware tree balancing schemes. The k-forests are backed by our optimized B-trees, and their k parameter is picked to match our choice of b (i.e., $k = 16$). In k-splay trees, we picked $k = 3$; since each k-splay step takes $\Theta(k^2)$ time, we expect that higher values k would not give better performance.

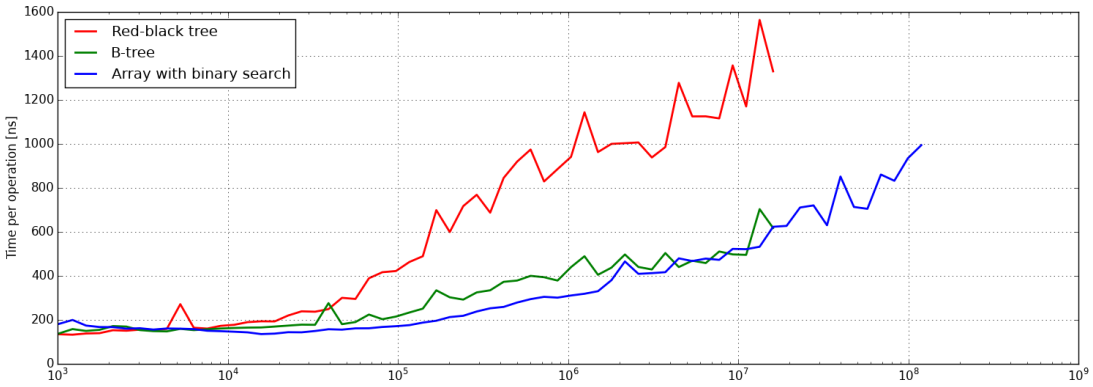
Splay trees are the canonical self-adjusting structure we would like to outperfrom. As expected, splay trees are much slower than optimized B-trees. Red-black trees are also faster than splay trees, through less dramatically than B-trees. Interestingly, splay trees were slower in every experiment, including experiments specifically aimed at exploiting the working set property.

Regarding the speed of random FINDs, neither k-forests nor k-splay trees managed to outperform splay trees. Interestingly, unsuccessful FINDs in k-forests are much faster than successful ones. This suggests that reading from all trees of a k-forest is much cheaper than promoting within a limited number of trees. It might be possible to improve k-forests by switching to a backing structure with simpler demotions and promotions.

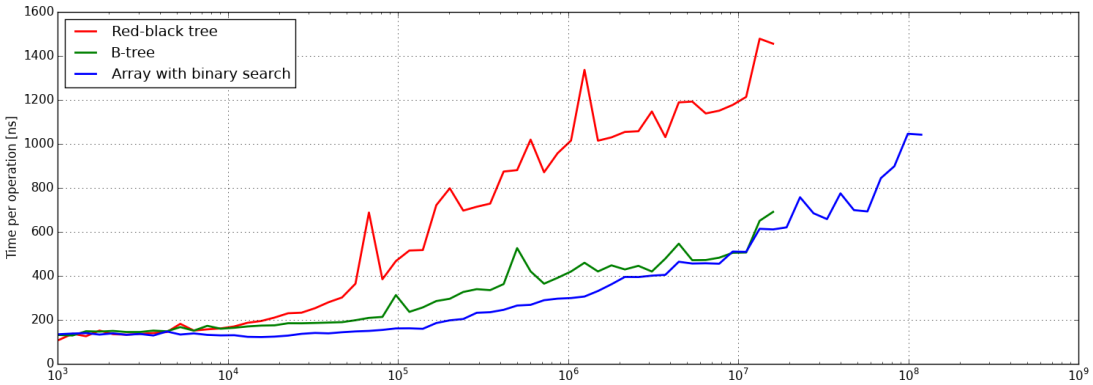
Profiling our performance tests running on k-splay trees showed that about 21% of time is spent in `ksplay_walk_to`, which finds the correct leaf for a key and puts the path to the leaf into a buffer. About 37% of the time is taken by `ksplay_step`, which performs a splaying step, along with helper functions it calls (`flatten_explore`, `compose_twolevel`).



(a) 100% success rate



(b) 50% success rate



(c) 0% success rate

Figure 9.2: Performance of random FINDs in basic dictionaries.

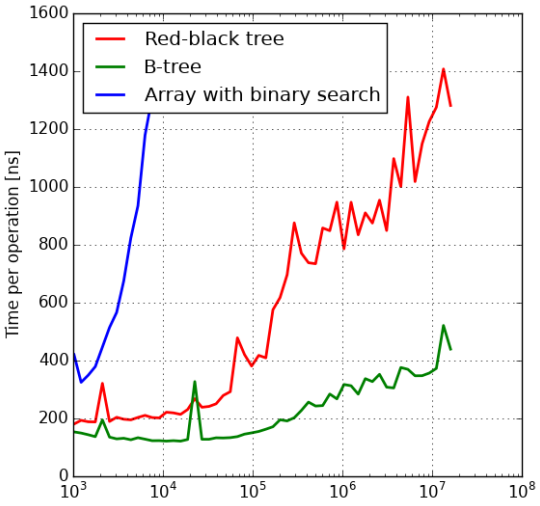


Figure 9.3: Performance of random INSERTs in basic dictionaries.

One suboptimal property of our implementation is that it dynamically allocates the k-splayed path on every k-splay. Memory allocation alone accounts for about 20% of CPU time. We experimented with replacing memory allocation with a static array. This change brought the performance of k-splay trees somewhat closer to splay trees, but splay trees still won in every test (see figure 9.7).

As suggested in [50], it should be possible to remove the need to store the splayed path by top-down k-splaying. Unfortunately, even top-down k-splaying needs to touch $\Theta(k^2)$ keys and pointers in every k-splay step, so we believe top-down k-splaying would not significantly reduce the high cost of `ksplay_step`.

Our measurements on small working sets (figure 9.6b, c) illustrate that splay trees, k-forests and k-splay trees have the working set property. However, even through B-trees do not have the theoretical working set property, they easily outperformed self-adjusting data structures on benchmarks explicitly designed to benefit structures optimized for small working sets.

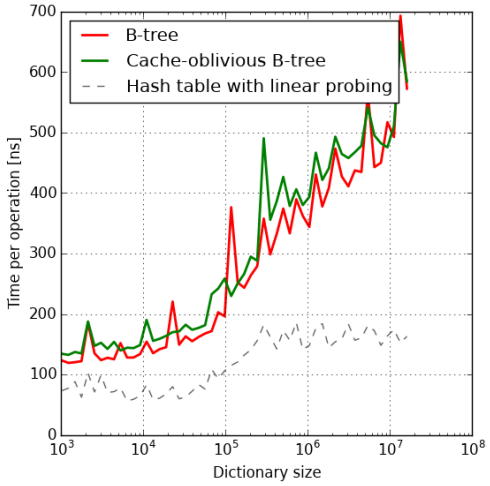
9.5 Hashing

We compared a hash table with linear probing to a cuckoo hash table. Both hash tables used a bytewise simple tabulation hash function. The cuckoo hash table maintained a load factor between 1/4 and 1/2 and the hash table with linear probing had a load factor between 1/4 and 3/4.

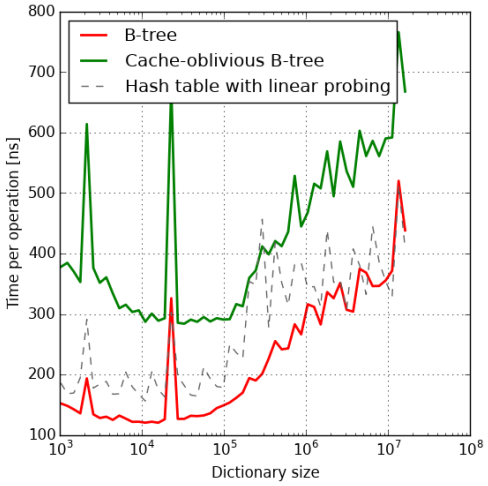
Since cuckoo hash tables guarantee worst-case $\mathcal{O}(1)$ time for lookups, we expected both successful and unsuccessful random FINDs to be measurably slower with linear probing. On random successful FINDs (figure 9.8a), cuckoo hashing did only very slightly better than linear probing. On unsuccessful FINDs (figure 9.8c), the benefits of cuckoo hashing were more pronounced. On the other hand, linear probing allows a higher load factor. (Forcing cuckoo hashing to fill the hash tables more fully significantly slows down INSERTs.)

INSERTs into cuckoo hash tables proved significantly slower. One possible reason might be the limited independence guaranteed by simple tabulation hashing.

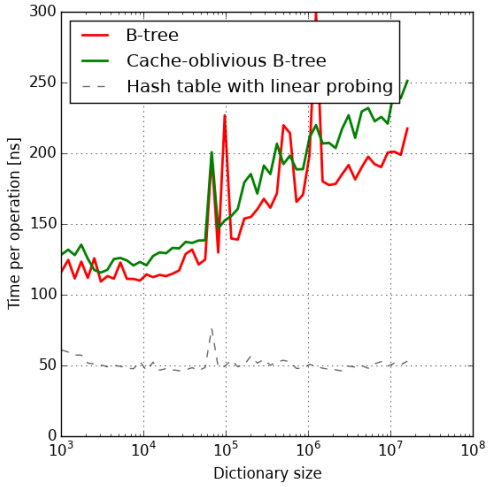
Another interesting fact is that insertions into B-trees with less than $\approx 10^7$



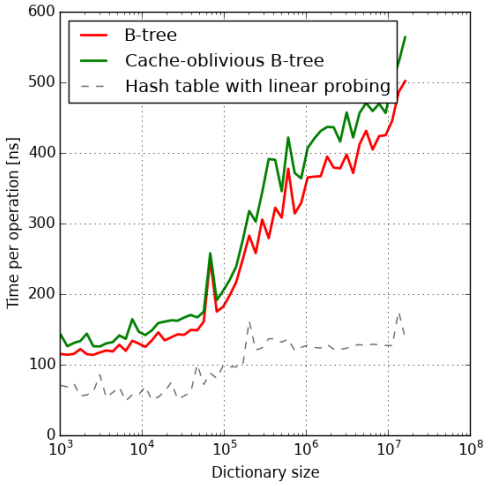
(a) Random FINDs, 100% successful (graphs for other success rates are similar)



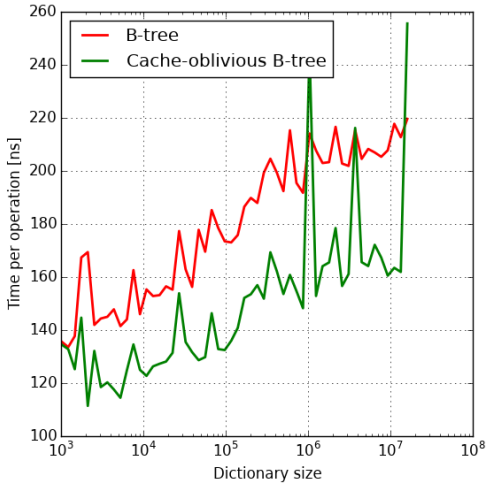
(b) Random INSERTs



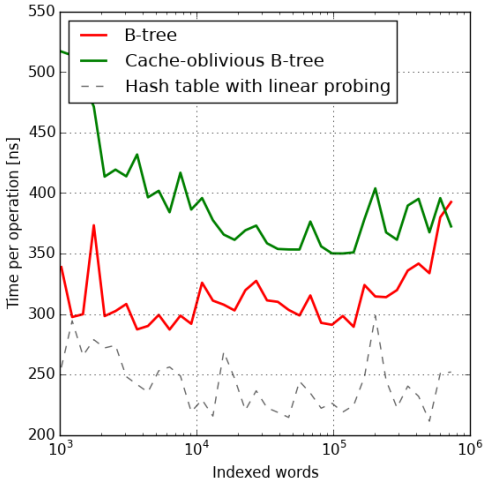
(c) Successful FINDs, working set of 1 000 keys



(d) Successful FINDs, working set of 100 000 keys

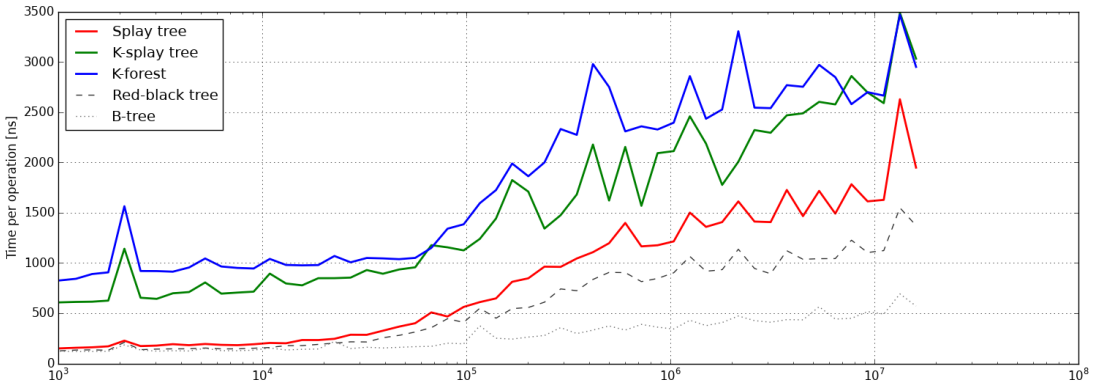


(e) Left-to-right scans over whole dictionary

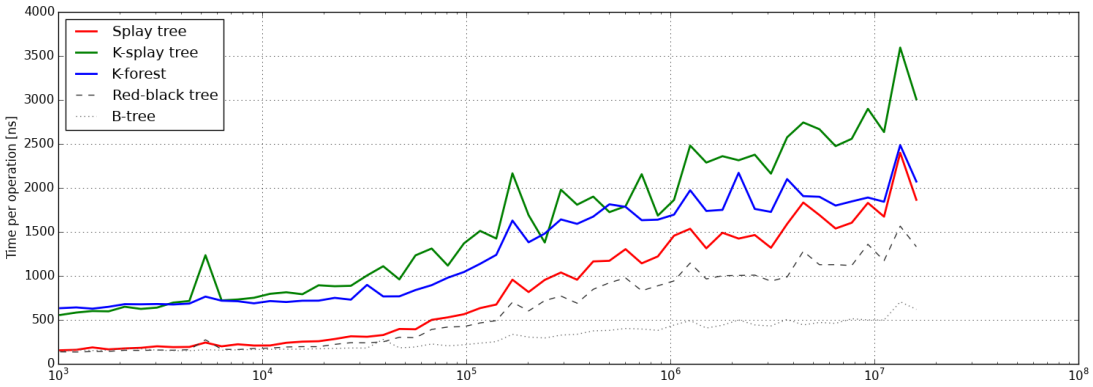


(f) Word occurrence counting

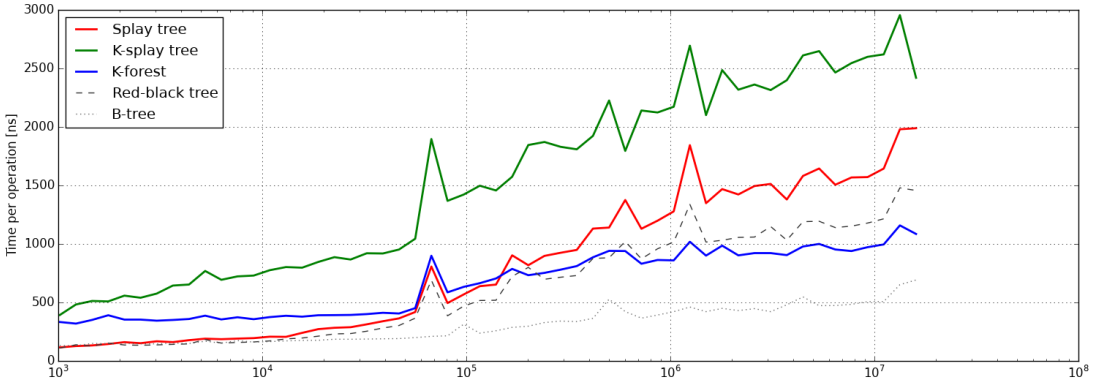
Figure 9.4: Benchmarks of cache-oblivious B-trees. Measurements of hash table with linear probing included for reference.



(a) 100% success rate

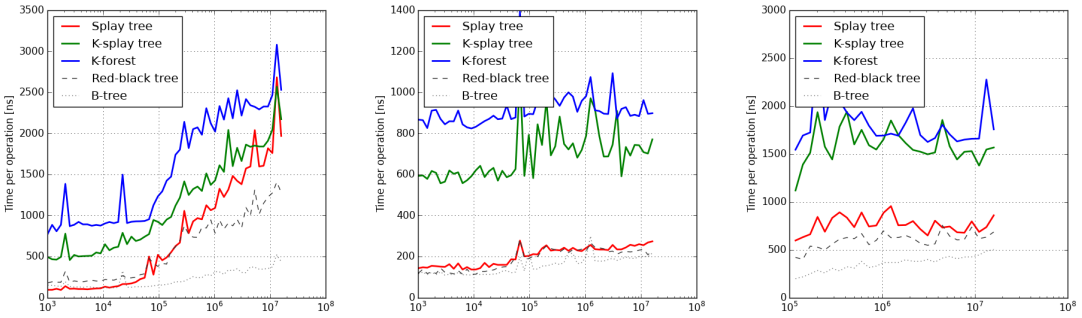


(b) 50% success rate



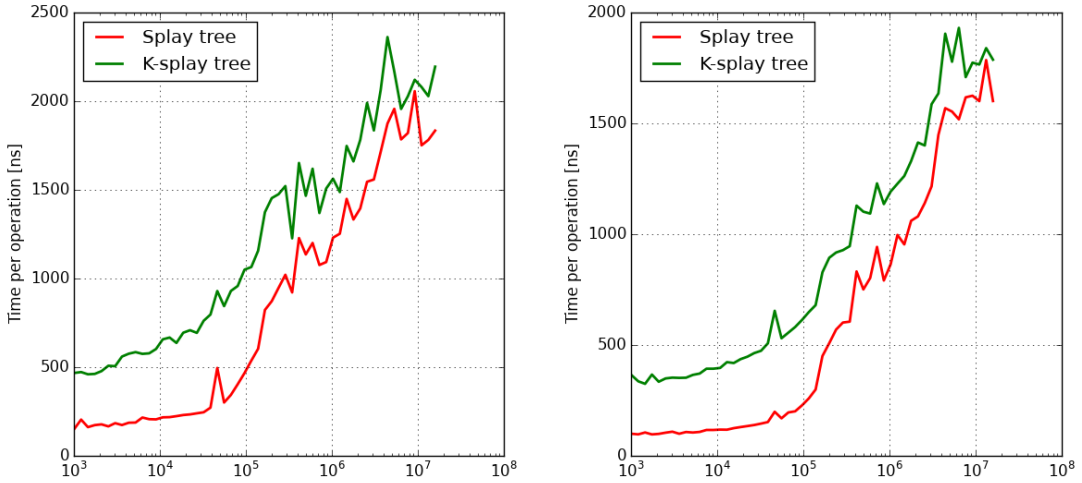
(c) 0% success rate

Figure 9.5: Performance of random FINDs in self-adjusting structures



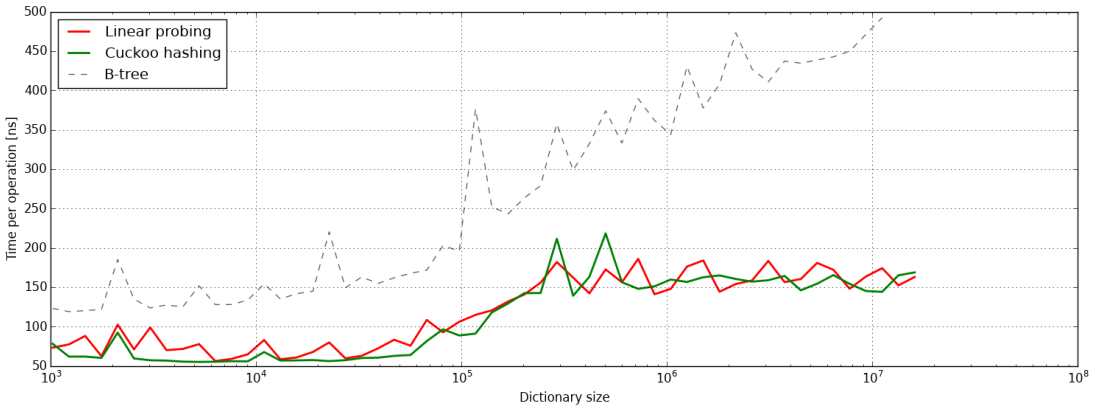
(a) Random INSERTs (b) Successful FINDs, working set of 1 000 keys (c) Successful FINDs, working set of 100 000 keys

Figure 9.6: Benchmarks of self-adjusting structures.

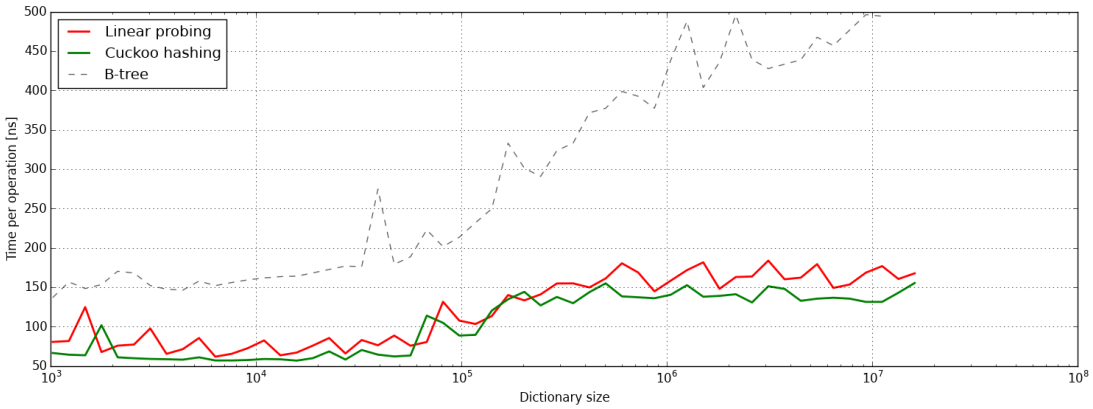


(a) Random FINDs, 50% success rate (b) Random INSERTs

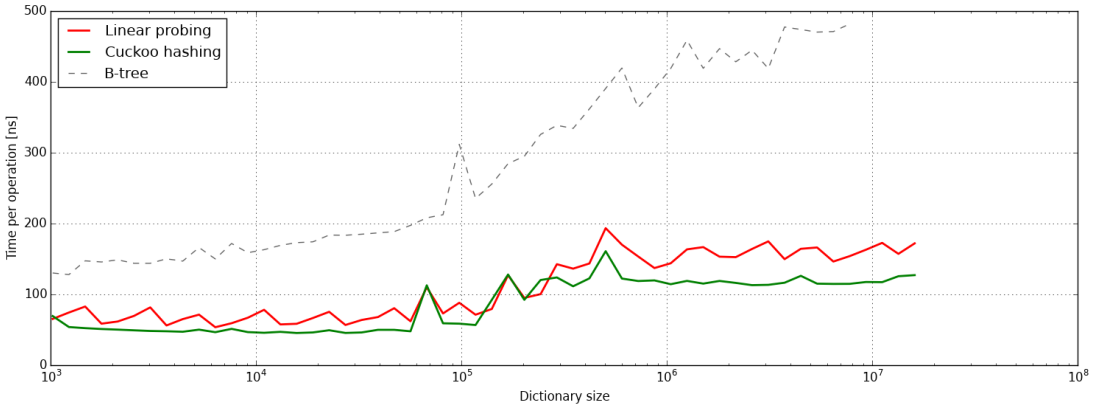
Figure 9.7: The performance of k-splay trees with removed temporary heap allocation, compared to splay trees



(a) 100% success rate

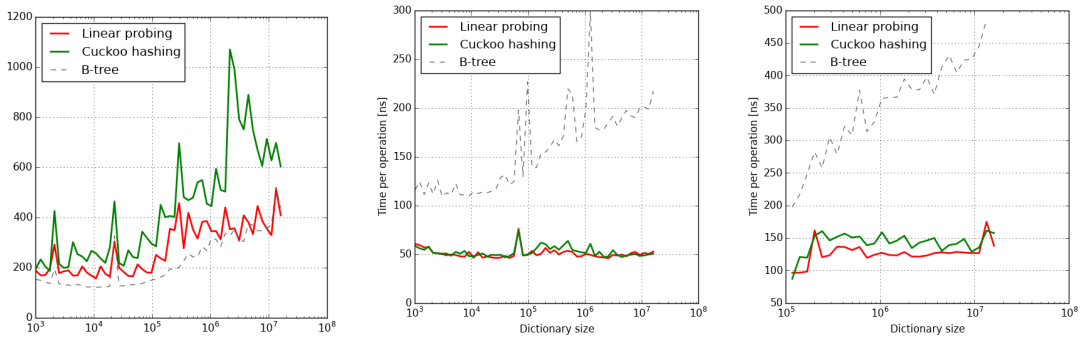


(b) 50% success rate



(c) 0% success rate

Figure 9.8: Performance of random FINDs in cuckoo hashing and linear probing



(a) Random INSERTs (b) Successful FINDs, working set of 1 000 keys (c) Successful FINDs, working set of 100 000 keys

Figure 9.9: Performance of cuckoo hashing and linear probing on INSERTs and FINDs in working sets

keys were faster than inserting into hash tables with linear probing.

9.6 Practical experiments

9.6.1 Mozilla Firefox

The hash table operations collected from the Firefox session are compressed in `data/firefox-htable.tar.gz`. We replayed the operations on all our dictionary implementations. We normalized the measured times by the maximum time taken for each recording to allow easier comparison of relative performance. Table 9.1 summarizes the results.

Overall, B-trees performed worse than our synthetic experiments would suggest. In most cases, they were ourperformed by red-black trees, which were slower on all synthetic experiments. One possible explanation is the presence of DELETEs in the recording – we did not perform any experiment to explicitly measure their speed.

Interestingly, cache-oblivious B-trees (COBT) do not appear to have the same problem as cache-optimized B-trees. This suggests that our choice of b in section 9.1 may have been too biased towards large dictionaries, which may be slowed down by a different bottleneck than small dictionaries. Cache-oblivious B-trees avoid committing to one b via cache-obliviousness.

The winning structure in all recordings were either cuckoo hash tables, splay trees, or red-black trees.

By inspecting the recordings, we found that splay trees are better than cuckoo hash tables on recordings with many updates, while cuckoo tables win when there are many more FINDs. We believe a data structure that would dynamically switch its internal representation between these three data structures based on the access pattern may be a good compromise on small dictionaries.

Interestingly, linear probing does not behave as well on Firefox recordings as in synthetic experiments, in which it performed as well as cuckoo hashing in random FINDs and slightly better on INSERTs. One possible reason for this is a higher incidence of failed INSERT and FIND operations. As we observed in section 9.5, failed FINDs are faster with cuckoo hashing, as they take only

Recording	Array	Red-black tree	B-trees	COBT	Splay	k-forest	k-splay	Linear probing	Cuckoo
`0x7f47930300e0`	14	17	16	13	15	92	100	7	**4**
`0x7fff84eb8f98`	13	10	24	14	13	76	100	11	**6**
`0x7fff1d9c7738`	16	16	22	16	14	77	100	15	**6**
`0x7fff84eb9148`	15	10	20	14	13	78	100	11	**7**
`0x7f566e3cb0e0`	11	8	17	13	12	98	100	13	**7**
`0x7f567b64a830`	100	8	5	6	**1**	31	30	4	4
`0x7f4791631c60`	6	7	33	10	**5**	100	58	9	7
`0x7f5671faab80`	16	13	31	25	**10**	48	100	59	20
`0x7f47a6e31058`	8	**6**	12	14	9	15	100	19	17
`0x7f565ca63148`	8	**8**	34	13	13	75	100	15	10
Average	21	10	21	14	11	69	88	16	9

Table 9.1: Results of the Mozilla Firefox experiment. Normalized by maximum time (lower is better). Sorted by which implementation is fastest. Produced by `experiments/vcr/make_table.py`.

Dictionary	Grid sampling	Random sampling
Cache-oblivious B-tree	20.47 s	20.27 s
Splay tree	6.21 s	6.67 s
k-splay tree	41.21 s	47.53 s
B-tree	21.77 s	21.27 s

Table 9.2: Results of the cloud database experiment, generated by `bin/experiments/cloud --max_year=2005`.

2 memory transfers as any other FIND, while an unsuccessful FIND with linear probing needs to traverse an entire chain.

9.6.2 Geospatial database

The results of the cloud database experiment are outlined in table 9.2. Cache-oblivious B-trees slightly outperformed standard B-trees. Splay trees were surprisingly fast. One factor that might cause them to perform so well might be that stations usually produce a large amount of reports over their lifetime, so when we fetch S closest reports for a point, it is likely that all such reports will be the first S reports reported by the closest station. Splay trees may thus keep old reports near the top. In this experiment, we picked $S = 1\ 000$. Also, as can be seen on figure 8.3, the distribution of stations on the globe is very uneven, so stations "assigned" to a large area can be kept closer to the root.

Additionally, we tried two algorithms for sampling query points on the globe. The first algorithm simply iterated through points on the latitude-longitude grid with whole number coordinates (i.e., 179.0° W 90.0° N through 180.0° E 90.0° S). The second algorithm arbitrarily picked $64\ 800 = 180 \cdot 360$ random query points. The algorithm can be selected by passing different values of the `--scatter` flag to the `bin/experiments/cloud` binary (`--scatter=GRID` for grid sampling, or `--scatter=RANDOM --samples=64800` for random sampling).

Self-adjusting structures (splay trees and k-splay trees) did better on the first one, which shows that they were probably able to take advantage of the more predictable access pattern.

Conclusion

We confirmed that cache-oblivious B-trees are competitive with more common data structures in main memory. A hand-optimized B-tree does better on some operations, but at the cost of picking the b for one cache boundary in advance. As we have seen on the Firefox recordings, parameters good for one size of dictionaries may also do not so well in other settings.

Our implementations of k-splay trees and k-forests were outperformed by traditional splay trees. In our opinion, k-forests alone are not a practical replacement for simple splay trees. However, a more optimized implementation of k-splaying (with removed heap allocation and with top-down k-splaying) might reach parity with splay trees or slightly beat them on large dictionaries.

Experiments on data recorded from Mozilla Firefox suggest that splay trees, cuckoo hash tables and red-black trees do well on most dictionaries as they are used by desktop software. A particularly interesting conclusion that can be drawn from table 9.1 is that hashing with linear probing may be (somewhat counter-intuitively) slower than splay trees. Larger databases that require order queries also seem to be best served by splay trees, followed by cache-oblivious B-trees and cache-aware B-trees.

Suggestions for further research

The dictionary problem is well-explored, including structures for efficient in particular models of computation and practical libraries for real computers. The space-time limitations of this thesis unfortunately did not permit us to completely survey all of them.

The performance of FINDs seems to be also dependent on the rate of success. Unsuccessful accesses could potentially be filtered out early by adding a *Bloom filter*[6] before the dictionary. We suggest measuring the effect in practical settings. Our measurements from Mozilla Firefox may be biased towards unsuccessful FINDs due to the way we filtered out unparseable lines.

In the Firefox recording, we found that either cuckoo hash tables, splay trees, or red-black trees always won among our sample of data structures. Analyzing the recorded access pattern may allow predicting which data structure would win. Based on such analysis, it may be possible to develop a data structure that would switch its internal representation based on the previously seen access pattern. Another approach may be collecting the access pattern first, and then automatically choosing the best data structure at compile time.

B-trees and their variants

Our implementation of B-trees was largely unoptimized. In particular, we used a linear scan when looking within nodes. On small nodes, this is not very different from the speed of binary search, but since we found that nodes larger than 64 B cache lines did very well, binary search within these large nodes might make a difference.

Many variants of B-trees were developed in the settings of on-disk databases. One of them, *B*-trees* [16], use some extra bookkeeping to maintain a higher branching factor (i.e., fuller internal nodes). This may also be a useful optimization in main memory.

Cache-oblivious B-trees

The cache-oblivious B-tree shows promise: without any tuning, it almost matched a hand-optimized B-tree. By choosing the tuning constants (i.e., the minimum and maximum density and piece sizes) more carefully, we might obtain a better match. Another possible point of improvement may be the van Emde Boas layout. For example, we might choose a simpler representation for small trees that fit within a single cache line (e.g., the BFS layout).

Other self-adjusting structures

We believe benchmarking various non-traditional self-adjusting structures, like Tango trees from [18] or multi-splay trees from [58], could give interesting results. Splay trees are very performant on practical access patterns, but their worst-case $\mathcal{O}(N)$ time for FINDs limits their usefulness in time-sensitive systems. Some self-adjusting structures provide better worst-case access times (e.g., $\mathcal{O}(\log N)$ in multi-splay trees). On the other hand, the code for splay trees would likely be considerably simpler, so the complexity of operations might outweight time saved by fewer cache misses in the worst case.

Other approaches

Several data structures based on tries have also been proposed for the dictionary problem. One example are *y-fast tries* [60], which allow FINDs in time $\mathcal{O}(1)$ and updates and predecessor/successor queries in time $\mathcal{O}(\log\log|U|)$, where U is the key universe. They require $\mathcal{O}(N\log\log|U|)$ memory. The RAM model is required by y-fast tries – they assume that RAM operations can be performed on keys in constant time. The original motivation for developing y-fast tries was lowering the memory requirements of van Emde Boas queues ($\mathcal{O}(|U|)$) while keeping the same time for operations (at least in amortized expected value).

Judy arrays are highly optimized dictionaries developed at Hewlett-Packard [52, 27]. HP sources claim that Judy arrays are very efficient in practice. The data structure is based on sparse 256-ary tries with implementation tricks that compress the representation. Unfortunately, we found the existing documentation highly complex and sometimes incomplete. On the other hand, an open-source implementation is available at `http://judy.sourceforge.net/`, so it should be possible to independently analyze the data structure and possibly to improve upon it.

Bibliography

[1] Abramson, N. *Information Theory and Coding.* Electronics Series. McGraw-Hill, 1963.

[2] Ailamaki, A. Database architectures for new hardware. In *Proceedings of the 30th International Conference on Very Large Data Bases* (2004), vol. 30 of *VLDB '04*, VLDB Endowment, pp. 1241–1241.

[3] Bayer, R., and McCreight, E. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control* (1970), SIGFIDET '70, ACM, pp. 107–141.

[4] Belazzougui, D., Botelho, F. C., and Dietzfelbinger, M. Hash, displace, and compress. In *ESA* (2009), A. Fiat and P. Sanders, Eds., vol. 5757 of *Lecture Notes in Computer Science*, Springer, pp. 682–693.

[5] Bender, M. A., Demaine, E. D., and Farach-Colton, M. Cache-oblivious B-trees. *SIAM Journal on Computing 35*, 2 (2005), 341–358.

[6] Bloom, B. H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM 13*, 7 (July 1970), 422–426.

[7] Bose, P., Douïeb, K., and Langerman, S. Dynamic optimality for skip lists and B-trees. In *Proceedings of the 19th annual ACM-SIAM symposium on Discrete algorithms* (2008), Society for Industrial and Applied Mathematics, pp. 1106–1114.

[8] Brodal, G. S., and Fagerberg, R. On the limits of cache-obliviousness. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing* (2003), pp. 307–315.

[9] Brodal, G. S., Fagerberg, R., and Jacob, R. Cache oblivious search trees via binary trees of small height. In *SODA '02 Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete algorithms* (2002), pp. 39–48.

[10] Bădoiu, M., Cole, R., Demaine, E. D., and Iacono, J. A unified access bound on comparison-based dynamic dictionaries. *Theoretical Computer Science 382*, 2 (2007), 86–96. Latin American Theoretical Informatics.

[11] Carter, J. L., and Wegman, M. N. Universal classes of hash functions. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing* (1977), STOC '77, ACM, pp. 106–112.

[12] Celis, P., and Franco, J. The analysis of hashing with lazy deletions. *Information Sciences 62*, 1–2 (1992), 13–26.

[13] Cohen, J., and Kane, D. M. Bounds on the independence required for cuckoo hashing. *ACM Transactions on Algorithms* (2009).

[14] Cole, R. On the dynamic finger conjecture for splay trees. Part II: The proof. *SIAM Journal on Computing 30* (2000), 44–85.

[15] Cole, R., Mishra, B., Schmidt, J., and Siegel, A. On the dynamic finger conjecture for splay trees. Part I: Splay sorting log n-block sequences. *SIAM Journal on Computing 30* (2000), 1–43.

[16] Comer, D. Ubiquitous B-Tree. *ACM Computing Surveys 11*, 2 (June 1979), 121–137.

[17] Dean, B. C., and Jones, Z. H. Exploring the duality between skip lists and binary search trees. In *Proceedings of the 45th Annual Southeast Regional Conference* (2007), ACM-SE 45, ACM, pp. 395–399.

[18] Demaine, E. D., Harmon, D., Iacono, J., and Pătraşcu, M. Dynamic optimality—almost. *SIAM Journal on Computing 37*, 1 (May 2007), 240–251.

[19] Dietzfelbinger, M., Hagerup, T., Katajainen, J., and Penttonen, M. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms 25*, 1 (1997), 19–51.

[20] Dietzfelbinger, M., and Heide, F. M. a. d. A new universal class of hash functions and dynamic hashing in real time. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming* (London, UK, UK, 1990), ICALP '90, Springer-Verlag, pp. 6–19.

[21] Dietzfelbinger, M., Karlin, A. R., Mehlhorn, K., Meyer auf der Heide, F., Rohnert, H., and Tarjan, R. E. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput. 23*, 4 (1994), 738–761.

[22] Franceschini, G., and Grossi, R. Optimal worst-case operations for implicit cache-oblivious search trees. In *Algorithms and Data Structures: 8th International Workshop, WADS 2003, Ottawa, Ontario, Canada, July 30 - August 1, 2003. Proceedings* (2003), vol. 2748 of *Lecture Notes in Computer Science*, pp. 114–126.

[23] Franceschini, G., Grossi, R., Munro, J. I., and Pagli, L. Implicit B-trees: New results for the dictionary problem. In *43rd Symposium on Foundations of Computer Science (FOCS 2002), 16-19 November 2002, Vancouver, BC, Canada, Proceedings* (2002), pp. 145–154.

[24] Fredman, M. L., Komlós, J., and Szemerédi, E. Storing a sparse table with $\mathcal{O}(1)$ worst case access time. *Journal of the ACM 31*, 3 (June 1984), 538–544.

[25] Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. Cache-oblivious algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on* (1999), IEEE, pp. 285–297.

[26] Georgakopoulos, G. F. How to splay for $\log \log n$-competitiveness. In *Experimental and Efficient Algorithms. 4th International Workshop,*

WEA 2005, Santorini Island, Greece, May 10-13, 2005. Proceedings (2005), pp. 570–579.

[27] Gobeille, R., and Baskins, D. Data structure and storage and retrieval method supporting ordinality based searching and data retrieval, 2004. US Patent 6,735,595.

[28] Hahn, C., Warren, S., and Eastman, R. Extended edited synoptic cloud reports from ships and land stations over the globe, 1952-2009 (ndp-026c). , 1999. Accessed: 2015-04-09.

[29] Hilbert, D. Ueber die stetige abbildung einer line auf ein flächenstück. *Mathematische Annalen 38*, 3 (1891), 459–460.

[30] Iacono, J. Alternatives to splay trees with $\mathcal{O}(\log n)$ worst-case access times. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, 2001), SODA '01, Society for Industrial and Applied Mathematics, pp. 516–522.

[31] Intel 64 and IA-32 architectures software developer's manual V3. `http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html`. Accessed: 2015-05-09.

[32] Kirsch, A., Mitzenmacher, M., and Wieder, U. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing 39*, 4 (2009), 1543–1561.

[33] Knuth, D. Notes on "open" addressing, 1963.

[34] Levinthal, D. Performance analysis guide for Intel Core i7 processor and Intel Xeon 5500 processors. `https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf`. Accessed: 2015-05-09.

[35] Mareš, M. The saga of minimum spanning trees. *Computer Science Review 2*, 3 (2008), 165–221.

[36] Martel, C. U. Self-adjusting multi-way search trees. *Information Processing Letters 38*, 3 (1991), 135–141.

[37] McClurkin, D. J., and Georgakopoulos, G. F. Sphendamnœ: A proof that k-splay fails to achieve $\log_k n$ behaviour. In *Advances in Informatics: 8th Panhellenic Conference on Informatics, PCI 2001* (2001), pp. 480–496.

[38] Mitzenmacher, M. Some open questions related to cuckoo hashing. In *Algorithms-ESA 2009*. Springer, 2009, pp. 1–10.

[39] Morton, G. M. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company, 1966.

[40] Pagh, A., Pagh, R., and Ruzic, M. Linear probing with constant independence. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing* (2007), STOC '07, ACM, pp. 318–327.

[41] Pagh, R. *Hashing, Randomness and Dictionaries*. PhD thesis, University of Aarhus, 2002.

[42] Pagh, R., and Rodler, F. F. Cuckoo hashing. *Journal of Algorithms 51*, 2 (2004), 122–144.

[43] Peano, G. Sur une courbe, qui remplit toute une aire plane. *Mathematische Annalen 36*, 1 (1890), 157–160.

[44] Pfaff, B. Performance analysis of BSTs in system software. *SIGMETRICS Perform. Eval. Rev. 32*, 1 (June 2004), 410–411.

[45] Pătraşcu, M., and Thorup, M. On the k-independence required by linear probing and minwise independence. In *Automata, Languages and Programming*, S. Abramsky, C. Gavoille, C. Kirchner, F. Meyer auf der Heide, and P. Spirakis, Eds., vol. 6198 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010, pp. 715–726.

[46] Pătraşcu, M., and Thorup, M. The power of simple tabulation hashing. In *Proceedings of the 43rd Annual ACM Symposium on Theory of Computing* (2011), STOC '11, ACM, pp. 1–10.

[47] Schmidt, J. P., Siegel, A., and Srinivasan, A. Chernoff-Hoeffding bounds for applications with limited independence. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, 1993), SODA '93, Society for Industrial and Applied Mathematics, pp. 331–340.

[48] Schneier, B. *Applied Cryptography (2nd Edition): Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., 1995.

[49] Sedgewick, R. Left-leaning red-black trees. `http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf`. Accessed: 2015-05-09.

[50] Sherk, M. Self-adjusting k-ary search trees. *Journal of Algorithms 19*, 1 (July 1995), 25–44.

[51] Siegel, A. On universal classes of extremely random constant-time hash functions. *SIAM Journal on Computing 33*, 3 (Mar. 2004), 505–543.

[52] Silverstein, A. Judy IV shop manual. `http://judy.sourceforge.net/doc/shop_interm.pdf`, 2002. Accessed: 2015-04-20.

[53] Sleator, D. D., and Tarjan, R. E. Amortized efficiency of list update and paging rules. *Communications of the ACM 28*, 2 (1985), 202–208.

[54] Sleator, D. D., and Tarjan, R. E. Self-adjusting binary search trees. *Journal of the ACM 32*, 3 (July 1985), 625–686.

[55] Thorup, M., and Zhang, Y. Tabulation based 4-universal hashing with applications to second moment estimation. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, 2004), SODA '04, Society for Industrial and Applied Mathematics, pp. 615–624.

[56] Van Emde Boas, P., Kaas, R., and Zijlstra, E. Design and implementation of an efficient priority queue. *Mathematical Systems Theory 10* (1977), 99–127.

[57] Vitter, J. S. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys 33*, 2 (June 2001), 209–271.

[58] Wang, C. C., Derryberry, J., and Sleator, D. D. $\mathcal{O}(\log \log n)$-competitive dynamic binary search trees. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm* (Philadelphia, PA, USA, 2006), SODA '06, Society for Industrial and Applied Mathematics, pp. 374–383.

[59] Wegman, M. N., and Carter, J. L. New hash functions and their use in authentication and set equality. *Journal of computer and system sciences 22*, 3 (1981), 265–279.

[60] Willard, D. E. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters 17*, 2 (1983), 81–84.

[61] Willard, D. E. A density control algorithm for doing insertions and deletions in a sequentially ordered file in a good worst-case time. *Information and Computation 97*, 2 (April 1992), 150–204.

List of Figures

List of Abbreviations

BFS Breadth-first search (on a rooted tree).

BST Balanced (binary) search tree, e.g., an AVL tree or a red-black tree.

COBT Cache-oblivious B-tree (see chapter 7).

CPU Central processing unit.

DFS Depth-first search (on a rooted tree).

HDD Hard disk drive.

RAM Random access machine (see chapter 1).