

A study of I/O efficient maps and sets

Henrik Thorup Andersen

Master's Thesis

Department of Computer Science
University of Copenhagen

Supervisor: Jyrki Katajainen

May 16, 2016

Abstract

Maps and sets are ubiquitous in modern computing, and I/O efficient implementations of these are of central interest to high performance applications. We present a novel variant on the B-tree, achieving the same asymptotical bounds on all operations for two different sets of parameters, in practice tuning for both disk and cache. We also discuss hopscotch hashing, presenting a new (partly empirical) analysis, to counter the faulty analysis presented in the original paper. We provide implementations and perform experiments to evaluate the performance of these data-structures in comparison with the more established I/O effective solutions to the map/set problem, the traditional B-tree and the linear probing hash map.

Contents

1	Introduction	3
1.1	I/O efficiency	3
1.2	The problem: maps and sets	4
2	Tree-based solutions	5
2.1	The traditional B-tree	5
2.1.1	Search	5
2.1.2	Scan	6
2.1.3	Insert	6
2.1.4	Delete	6
2.1.5	Other layers	6
2.2	The two-layer B-tree	7
2.2.1	Pool allocator	7
2.2.2	Inner tree characteristics	8
2.2.3	Balancing the outer trees	8
2.2.4	Search and scanning	9
2.2.5	Insert	9
2.2.6	Remove	10
2.3	Implementations	11
2.4	Experiments	12
2.4.1	Test environment	12
2.4.2	Test process	12
2.4.3	Results	13
2.5	Discussion	16
3	Hash-based solutions	18
3.1	Linear probing	18
3.1.1	Overview	18
3.1.2	Analysis	19
3.1.3	Recent results	19
3.2	Hopscotch hashing	20
3.2.1	Overview	20
3.2.2	Analysis	21
3.2.3	Forced resize	22
3.3	Implementation	24
3.4	Experiments	26
3.4.1	Hash function	26
3.4.2	Test data	26

3.4.3	Performance results	27
3.5	Discussion	27
4	Conclusion and further work	31
5	Bibliography	32
	Appendices	33
A	Implementation code	34
A.1	double_tree.hpp	34
A.2	double_tree_line_node.hpp	48
A.3	double_tree_page_node.hpp	53
A.4	extract.hpp	69
A.5	hopscotch.hpp	70
A.6	linear.hpp	83
A.7	tabulation.hpp	94
B	Test code	106
B.1	correctness_test.hpp	106
B.2	hopscotch_experiment.hpp	108
B.3	performance_clock.hpp	111
B.4	performance_clock.cpp	111
B.5	performance_test.hpp	112

Chapter 1

Introduction

1.1 I/O efficiency

A modern computer has several layers of memory organized in a hierarchy, with the fastest and smallest on top, closest to the processor, and each layer down getting progressively slower and larger. On the chip of modern processors are several pieces of memory called *caches*. The computer used for testing in this report, a typical desktop from a couple of years ago, has three layers of cache of sizes 32KB, 256KB and 6MB respectively. The transfer size between these caches is 64B, and this is also the transfer size between the cache and the main memory. The *main memory*, placed on the main board with a fast bus connecting to the processor, is the next layer down. On our computer, this memory has a capacity of 8GB. Further down the hierarchy is the *hard disk*. The transfer size to the hard disk on our computer is 4KB, while the drive itself has a capacity of 256GB.

One of the most important factors in how a program performs on a modern computer is then how well it utilizes the capacity of the interfaces between the layers in this memory hierarchy. The problem of designing algorithms and datastructures that use it well is thus of central importance.

The *external memory model* [1, 2, 3, 5] approaches this problem by modelling two layers within this hierarchy, called *fast* and *slow memory*, and minimising the number of transfers between these when executing some algorithm. The parameters of the model are the transfer-size between slow and fast memory, often measured in the number of elements that fit in one such transfer and labelled B , and the total capacity of fast memory, also typically measured in elements and labelled M . Slow memory is usually treated as unlimited.

The original focus of the external memory model was transfers between main memory and the hard disk, but as the cache has grown in importance, due to the discrepancy between improvements in CPU speeds and memory speeds, it has also become relevant in that realm. Vitter gives an overview of algorithms and data structures designed for this model [12].

Unfortunately, optimising for one of the interfaces of the memory hierarchy using the external memory model often does not help in other places of the hierarchy. An algorithm that is optimised for a typical disk page of 4KB might still suffer an excessive amount of cache misses, and vice versa. There are exceptions to this, as well as other models that do not have this shortcoming. In the *cache-oblivious model* [11], the size of the transfers between the two layers is treated as unknown, and it is shown that an algorithm that minimises the asymptotic number of transfers in this model does so in all layers of a hierarchy.

In this report we will be taking a different approach to that problem, by separately optimising for two different interfaces of the memory hierarchy, in effect using the external memory model twice, with two different sets of parameters. In practice this means that we will be optimising for both the interface between cache and main memory, and the interface between main memory and hard disk.

1.2 The problem: maps and sets

The two related problems of maintaining respectively a *map* and a *set* are central to computer science. A map is a collection of key-value pairs, while a set is just a collection of keys. For ease of presentation, we consider only the case where all keys are required to be unique. There should be only detail-work left in order to support the case of non-unique keys. We define the following operations on maps and sets:

- *Search*: retrieving an element from the map, given its key. For a set this serves only to check if the set contains that key.
- *Scan*: retrieving a sequence of elements from the map or set in order. This only applies if there is an ordering defined on the elements.
- *Insert*: inserting an element into the map or set.
- *Delete*: removing an element from the map or set, given its key.

One problem reduces easily to the other, so in this report we will from now on simply be referring to maps. The implementations we have done for this project are all in C++, and all support both the `std::map` and the `std::set` interfaces defined in the C++ standard, or the unordered variants of those, `std::unordered_map` and `std::unordered_set` [6].

Two different approaches have dominated the solutions to this problem: tree-based and hash-based. The *tree-based* approach requires an ordering of the elements. The basic idea is that each node of the tree represents a division of the elements into ranges within that ordering, separated by keys stored in the node. The *hash-based* approach requires a way to transform a key into a sufficiently random index into an array. The element is then ideally stored at that index, though because of collisions that is not always possible. We will look at I/O efficient variants of both approaches in turn.

Chapter 2

Tree-based solutions

The traditional I/O efficient tree-based map is the *B-tree*. The B-tree is optimal within the external memory model — that is, it is an asymptotically optimal solution for one layer of the memory hierarchy. In this chapter we will first provide a high-level overview of the B-tree, and then discuss a new variant that is asymptotically optimal for two separate levels of the memory hierarchy. We then present experiments that explore how these approaches perform in practice.

2.1 The traditional B-tree

The original B-tree was first presented by Bayer and McCreight [2]. What we will present here is in fact a variant of unknown origin sometimes known as the B⁺-tree, which has become very popular.

Let N denote the number of elements stored, M the number of elements that fit in fast memory, B the number of elements that will fit in one block transfer between slow and fast memory and R the number of elements visited by a scan.

A B-tree is a multi-way tree where all nodes are sized according to the size of the block transfer, such that they can contain a sequential array of $O(B)$ elements. All values are stored along with their keys in the leaves of the tree, so the inner nodes contain only copies of some keys and pointers to nodes on the next level. The nodes are kept so that only the root can ever go below half of its capacity.

2.1.1 Search

Let h be the height of the tree. During a search from the root of the tree, h nodes are visited. Since each node fits in a block, h I/O operations are performed. From the invariant that nodes (apart from the root) are never below half of their capacity, we get that $h = O(\log_B N)$. If we spend some fixed proportion of main memory on keeping the upper part of the tree loaded, we do not need any I/O operations for the first $O(\log_B M)$ levels, which gets us a $O(\log_B(N/M))$ bound on I/O operations for searching.

In each node visited we search to find the pointer to the node on the level below in which to continue the search, or in the leaves to find the value stored. This is done with a binary search taking $O(\lg B)$ work, and thus we have a bound of $O(\log_B N \times \lg B) = O(\lg N)$ on work for searching.

2.1.2 Scan

When analyzing scan operations we leave out the cost of obtaining the starting location, since that is just the cost of a search. The leaf nodes are linked together in a linked list, so that given a leaf node we can easily obtain its neighbors and scan R elements using $O(R/B)$ I/O operations and $O(R)$ work.

2.1.3 Insert

For inserting we first search to find the leaf node to insert to, taking $O(\lg N)$ work and $O(\log_B N/M)$ I/O operations. All elements after this position must be moved to make space, after which we write the new element, for a total of $O(B)$ work (or constant work if using a ring buffer). If the node is full we split it into two half full nodes, and if that makes its parent full we must split that and so on. In the worst case $h = O(\log_B N)$ splits are made, each taking $O(B)$ work. The worst-case work cost of splitting is then $O(B \log_B N)$, dominating the cost of insertion. Since nodes are visited on the path from the leaf to the root, $O(\log_B(N/M))$ I/O operations are needed, by the same argument used for search.

2.1.4 Delete

To maintain the invariant that each node must be at least half full we must sometimes merge nodes on deletion. It is not always possible to perform a merge since it might be that none of the adjacent nodes are small enough that the result can fit in one node. In those cases values must be *borrowed* from the adjacent nodes. Borrowing takes constant work (again, if using a ring buffer) and merging takes $O(B)$ work. As with splits, the worst case requires $h = O(\log_B N)$ merges, so that we get a worst case work cost of $O(B \log_B N)$, and by the same arguments as for search and insert we need $O(\log_B(N/M))$ I/O operations.

2.1.5 Other layers

Let us take a minute here to analyse what happens for other layers of the memory hierarchy. Suppose the B-tree is optimised for the parameters of the memory-disk interface. Let M_m be the number of elements that fit in main memory, B_m the number of elements that fit in one block transfer between the disk and the main memory (a disk page), M_c be the number of elements that fit in the lowest layer of the cache and B_c the number of elements that fit in one block transfer between the main memory and the cache (a cache line).

For each layer of the tree, the search algorithm makes a binary search, in which it visits $O(\lg B_m)$ elements. Each of these are in a different cache line, until the search closes in on one cache line. There is $O(\lg B_m)$ steps of the search, and the $O(\lg B_c)$ last ones are in the same cache line, so that makes $O(\log_{B_m} N \times \lg B_m/B_c)$ or $O(\lg N - \log_{B_m} N \times \lg B_c)$ cache I/Os for a search. For scanning, all of the elements are accessed in sequential arrays (after the initial search), so that makes $O(R/B_c)$ I/Os. For insertions and deletions, we might have to visit the entirety of all the nodes on the path, if splitting or merging, so that we get a worst case of $O(\log_{B_m} N \times B_m/B_c)$ cache I/Os.

If on the other hand the B^+ -tree is optimised for the cache-memory interface, that is for M_c and B_c , we can analyse how many disk I/Os are needed for the various operations. In that case, each node is entirely contained in one disk page (if alignment is properly taken care of), but each of the $O(\log_{B_c} N)$ nodes visited for a search, an insertion or a removal could be on a different disk page. These operations then all take $O(\log_{B_c} N)$ disk I/Os. Similarly, scanning takes $O(R/B_c)$ disk I/Os.

2.2 The two-layer B-tree

We want a data-structure that achieves the same I/O bounds as the B-tree, but for two separate sets of parameters — that is, for both the cache and the disk. The strategy we use to achieve this is to embed a B-tree inside each node of a B-tree. The outer nodes can then be sized according to the size of a disk page, while the inner nodes can be sized according to the size of a cache line. We will call these *page-nodes* and *line-nodes*, respectively. This strategy of course involves a number of complications, which we will deal with in the following.

2.2.1 Pool allocator

We need a mechanism to ensure that the trees embedded in the page-nodes are entirely contained within the memory of that node. Normally we allocate and deallocate nodes using the mechanisms provided by the operating system, and assume that this can be done in constant time, so our replacement mechanism needs to live up to that assumption as well. Also, we need to be able to initialize a new address space within a page-node in constant time. This is achieved by the *pool allocator*.

The pool allocator needs only to manage entries of one size, the size of a line-node, so it has an array of these entries ready for use. To support the balancing algorithms, our pool allocator keeps track of how many entries are currently allocated.

The *head* is the index of the entry that will be allocated next, and points to the first entry at initialization. The *back* is the greatest index that has never been allocated, and so also points to the first entry at initialization.

When an entry is deallocated, the previous value of the head is written into the memory of that entry, and the head is set to the index of that entry. When an

entry is allocated, if the current head is less than the current back (meaning that the entry has been previously allocated), we set the head to the head from before that entry was deallocated, which we read from the memory of that entry. If the allocated entry has never been allocated before (that is, it is equal to the current back), we increment both the head and the back. It is easy to see that initialization, allocation and deallocation all use constant work in this scheme.

2.2.2 Inner tree characteristics

Let us get some basics about the inner trees out of the way. The line-nodes of the inner trees are, like the nodes of a regular B-tree, sequential arrays holding $O(B_c)$ elements. All of the operations on the inner trees work exactly like their counterparts for a regular B-tree, with the exception that the trees can only grow to a certain amount of nodes. We denote the amount of line-nodes in a page by q , and the maximum number of nodes possible by q_{\max} , so we have that $q = O(q_{\max}) = O(B_m/B_c)$. This limitation in size is handled by making sure not to do any operations that could create more nodes than allowed (more on this below), so it has no impact on the operations within the line-nodes.

Because of the constraint on the amount of nodes, an inner tree has $O(B_c) \times O(B_m/B_c) = O(B_m)$ elements, so we can replace N with that where it occurs in the characteristics of the B-tree:

- Search takes $O(\lg B_m)$ work and $O(\log_{B_c}(B_m/M_c))$ cache I/Os.
- Insert takes $O(B_c \log_{B_c} B_m)$ work and $O(\log_{B_c}(B_m/M_c))$ cache I/Os.
- Delete takes $O(B_c \log_{B_c} B_m)$ work and $O(\log_{B_c}(B_m/M_c))$ cache I/Os.
- Scan takes $O(R)$ work and $O(R/B_c)$ cache I/Os.

The constraint on the amount of nodes also means that there is a maximum height, which we will denote $h_{\max} = \Theta(\log_{B_c} B_m)$. Because all the nodes are inside one disk page, all operations on an inner tree take just one disk I/O.

2.2.3 Balancing the outer trees

The balancing mechanisms of the B-tree rest on the notions of nodes being full or half-full, but for our outer trees we need something a bit different. Define a *large* page to be any page with $q \geq q_{\text{large}} = q_{\max} - 2h_{\max} + 1$, and a *small* page to be one that is not large. Since $h_{\max} = \Theta(\log_{B_c} B_m)$, large nodes have $q = \Theta(q_{\max}) = \Theta(B_m/B_c)$.

Let l be the number of leaves in a page and b be the minimum branching factor of the line-nodes (except for the root — for the moment, we disregard the fact that the root can have less branches). The number of nodes on the layer up from the leaves must be less than or equal to l/b , the next layer up from that less than or equal to l/b^2 , and so on. This means that

$$q \leq \sum_{i=0}^{h-1} \frac{l}{b^i} \leq \sum_{i=0}^{\infty} \frac{l}{b^i} = \frac{l}{1 - \frac{1}{b}} \implies l \geq q \left(1 - \frac{1}{b}\right) \implies l = \Theta(q),$$

since l is obviously $O(q)$. It is not hard to see that less branches on the root does not change that conclusion. Thus a large page has $l = \Theta(q_{\max}) = \Theta(B_m/B_c)$ leaves. Since each leaf has $\Theta(B_c)$ elements, a large page has $\Theta(B_m)$ elements.

In the worst-case scenario splitting or merging page would take $O(B_m/B_c)$ cache I/Os for each level of the tree, which is not as efficient as we would like. Instead we have to rely only on transferring line-nodes between pages. To do this while keeping the bound on the height of the tree, we define this invariant: *if a page is small, its adjacent pages must be large.*

We can see that this limits the height as we want, by thinking in pairs of adjacent pages: for any adjacent pair of pages one of them will be large, so together they will contain $\Theta(B_m)$ elements. So the tree will have a height of $\Theta(\log_{B_m} N)$.

2.2.4 Search and scanning

The only difference for searching and scanning is that we use the inner trees for searching and traversing inside pages. It is obvious from the above that a search takes $O(\log_{B_m} N)$ disk I/Os. For each of these we perform $O(\log_{B_c} B_m)$ cache I/Os, so that in total we perform $O(\log_{B_m} N \times \log_{B_c} B_m) = O(\log_{B_c} N)$ cache I/Os, as we wanted. Using the linked list, it is easy to see that scanning takes $O(R/B_m)$ and $O(R/B_c)$ disk and cache I/Os respectively.

2.2.5 Insert

When inserting into the inner trees, the line-nodes might split all the way up the height of the tree, and a new root could be created, so even if the pool allocator is not full (that is, even if $q < q_{\max}$) we can not guarantee that we will have space for the insertion. To guarantee an insertion we need to have $q \leq q_{\max} - h_{\max}$. We call a page that does not satisfy this *oversized*. The lowest value of q that an oversized node can have is then $q_{\text{oversized}} = q_{\max} - h_{\max} + 1$. An oversized page is always large. The process of insertion is as follows:

1. If the root is oversized:
 - (a) *Thin* the root (see below). This will result in a new page, since the root has no adjacent pages. Create a new root with the old root and the new page as children.
 - (b) Determine which of the two children of the new root the new element should go into and make that the current page.
2. If the root is not oversized, make it the current page.
3. While the current page is not a leaf:
 - (a) Determine which child of the current page the new element should go into. Call this the target page.
 - (b) If the target page is oversized:
 - i. *Thin* the target page (see below).

- ii. Determine which page the new element should go into and make that the current page (either the target page or an adjacent, potentially new, one).
- (c) If the target page is not oversized, make that the current page.
- 4. The current page is now a non-oversized leaf, insert the element into it.

As we go down the tree we make sure that each page on the path down is not full, the *thinning* of steps 1.a and 3.b.i. We do this by transferring line-nodes to an adjacent small page, or if none such exist, creating one. Each time, a total of h_{\max} of these transfers could be needed to get $q = q_{\max} - h_{\max}$. For each transfer, two situations are possible:

1. An adjacent page is small. We transfer a line-node from the full page into that one.
2. No adjacent page is small. We create a new page containing the line-node. The new page will be small, and the pages adjacent to it will be large, maintaining the invariant.

When we transfer a line-node it is possible that the element we are inserting should instead end up going to the page we are transferring to, because we are changing the minimum keys of the page. This is the reason for steps 1.b and 3.b.ii. It is not possible for the page we are transferring to to become full by the transfer, since a maximum of h_{\max} new line-nodes are created by the transfer, and we check that it has $q < q_{\max} - 2h_{\max}$.

When removing a line-node from the target page, we might end up merging line-nodes all the way up the inner tree. This can only happen once though, since we are removing leaves from the same stem, and if it has once been made full by a merge we can remove $h_{\max} - 1$ more leaves from it without causing another merge, by the assumption that $B_c > 2h_{\max}$. In the worst-case then this merging takes $O(\log_{B_c} B_m)$ cache I/Os.

Similarly, when inserting a line-node into an adjacent page we might cause splits all the way up the tree, but since we are inserting leaves into the same stem, if it has been split once we can insert $h_{\max} - 1$ more leaves into it without causing another split. As with merging, these splits then take $O(\log_{B_c} B_m)$ cache I/Os in the worst case.

The transfers of the line-nodes themselves, of which there are a maximum of h_{\max} also takes $O(\log_{B_c} B_m)$ cache I/Os in the worst case, so that this is the total worst-case cost of the process of thinning.

Insertion into an inner tree costs $O(\log_{B_c} B_m)$ cache I/Os as well. In the worst case we could cause insertions into all the pages all the way from the root down, from new page created. This gives us a total cost in cache I/Os of $O(\log_{B_m} N \times \log_{B_c} B_m) = O(\log_{B_c} N)$. Since we visit a constant number of pages for each level of the tree, the cost in disk I/Os is $O(\log_{B_m} N)$.

2.2.6 Remove

The process of removal is as follows:

1. Find the element to be removed (same process as searching). Set its page as the current page and mark it for removal.
2. While there is a current page with an element marked for removal.
 - (a) Remove the marked element.
 - (b) If the current page was small, it might now be empty. If so set the parent as the current page and mark the element corresponding to this page for removal.
 - (c) If the current page was large, has any adjacent small nodes, and became small from the removal, *grow* it (see below). If this results in an empty page, set the parent as the current page and mark the element corresponding to the empty page for removal.
 - (d) Otherwise we are done.

Going up the tree after the removal we make sure that no large page has been made small in a way that breaks the invariant, the *growing* of step 2.c. We do this by transferring line-nodes from adjacent small pages. If there are no small pages, the invariant is not broken. When removing from the inner trees, the line-nodes might merge all the way up the height of the tree, and the root could be collapsed, so that at most the inner tree loses h_{\max} line-nodes. So, for each grow step a total of h_{\max} of these transfers could be needed. If at any point there are no more small nodes to transfer from, the invariant has been established because the adjacent pages are now large and the current page can stay small.

Inserting into the current page can in the worst-case cause two cascades of splits in the inner tree, since we can only be inserting leaves into the two outermost of the lower stem nodes, by a similar argument as was used in the insertion case (see above). Similarly, we are only removing leaves from the outermost stems of the neighbours, and at most removing h_{\max} leaves, so only two cascades of merges can happen. By the same reasoning as with thin steps then, a grow step takes $O(\log_{B_c} B_m)$ cache I/Os in the worst case.

Removing a line-node from an inner tree takes $O(\log_{B_c} B_m)$ cache I/Os. In the worst case we could cause removals with associated grow steps in all pages in a path from the leaves to the root, giving us a total cost in cache I/Os of $O(\log_{B_m} N \times \log_{B_c} B_m) = O(\log_{B_c} N)$. Since we visit a constant number of pages for each level of the tree, the cost in disk I/Os is $O(\log_{B_m} N)$.

2.3 Implementations

We implemented the two-layered B-tree in C++. The implementation was done so that it can follow the interface of both `std::set` and `std::map` as defined in the C++ standard [6], though it does not completely implement all of the functionality there. It should be relatively easy to implement the rest. The code for our implementation is listed in appendix A.1-3.

We must note here that there seems to be some kind of memory bug in the deletion procedure, which has been very evasive. We discovered it late in the process since we had (perhaps naively) only been testing with 32-bit keys and values until the time came to do the experiments. We did all our experiments with 64-bit keys and values, and when doing that, the bug rears its head. It seems to occur somewhat randomly, but only on large input sizes, leading us to believe it is situated somewhere in ‘the outer layers’ of the data structure. On top of that, we have not been able to reproduce it with optimization turned off, so it is impossible to use a debugger to inspect the process when it occurs. We are confident that this problem could be solved with a determined effort. We do not believe that this had any effect on the actual performance of the structure. It did interfere with our experiments in that the program crashed, so that we did not get timings on deletions for all values of N .

The implementation used to test the performance of a traditional B-tree was `stx::btree_map`, available from <https://github.com/bingmann/stx-btree>. This is a mature implementation that performs well, and has been actively maintained since 2007. It follows the same interfaces as our implementation.

2.4 Experiments

2.4.1 Test environment

The experiments were carried out on an Intel i5-3750K processor with a clock speed of 3.4 GHz on an Intel Z77 chipset. The disk used is a Crucial M4 solid state drive connected to the SATA III port of the Z77 board. All of the tests were carried out under the Linux 4.5.4 kernel on a fresh install running nearly no other processes. The test programs were all compiled with gcc version 6.1.1 with optimization option `-O3` and no debug symbols.

For testing performance on disk we made use of the virtual memory system of the operating system. As the data structures exceeded memory capacity, pages were swapped to disk automatically, allowing us to free ourselves from the problem of using the file system with all its complexities. Memory was restricted to 1GB in order to push the data onto disk without using very large datasets, that would have taken too long to run.

2.4.2 Test process

The experiment that was performed was as follows: we inserted 2^{26} elements in batches of 2^{18} at a time. We used 64-bit unsigned integers (`uint64_t`) as both keys and values, resulting in 128-bit elements. The B-trees are not sensitive to the actual values of the keys inserted, since they work only by comparing them. The test data therefore consisted in just the list of integers $\{0, \dots, n-1\}$, in a pseudo-random permutation, as generated by the C++ standard library function `std::shuffle`. Since the values are of no consequence to the performance, these were just the list of integers $\{0, \dots, n-1\}$.

Each time we had inserted a batch we searched for 2^{18} elements that had already been inserted. The elements to search for were chosen by finding a random place in the array of elements to insert, before the last element that was inserted, and searching for the next 2^{18} elements in that array. Each time we also scanned 2^{18} elements, from a random starting position, again chosen at random from the array of elements to insert. After the insertion we removed all the elements, again in batches of 2^{18} elements at a time. The full code for this experiment is listed in appendix B.5.

In this way we obtained timings for these four operations for various values of N . The experiment was carried out on our new two-layer B-tree with inner nodes of 256KB and outer nodes of 4KB, as well as on two traditional B-trees, one tuned for cache with a node size of 256KB, and one tuned for disk with a node size of 4KB.

2.4.3 Results

The results are plotted on the graphs in figure 2.1 and 2.2. All of these graphs have N on the horizontal axis, plotted linearly, and time on the vertical axis, plotted logarithmically. The logarithmic time axis was chosen because of the large difference in time before and after the data hits the disk.

The first pair of graphs, on figure 2.1, show the time per insertion and time per search respectively. They are very similar. For all three test structures the graph has the same basic shape: at the very left end of the axis there is a sharp increase as the size of the datastructure increases beyond the size of the cache. A similar sharp increase is seen somewhere around the middle of the graph as the size increases beyond the size of the disk. The sections in between these all see a more gradual increase as the height of the trees increases gradually.

What mostly sets the data structures apart is when the sharp increase from memory to disk happens, as well as at what level the sections in between ‘rests’. Both the two-layer B-tree and the traditional B-tree optimized for cache take 300-500ns per insertion and search before hitting the disk, while the traditional B-tree optimized for disk performs a bit worse. The two-layer B-tree hits disk notably before the traditional B-tree optimized for disk, at around 2.3×10^7 elements as compared to about 3.1×10^7 elements. The traditional B-tree optimized for disk hits disk even later, at around 3.7×10^7 elements.

After hitting disk, the B-tree optimized for disk performs the best, followed by the one optimized for cache, and the two-layer B-tree performs worst. Looking at the very right of the graph, it seems as though the traditional B-tree for cache might soon overtake the two-layer B-tree. We regret that we did not perform the experiments with a larger dataset to see if this hypothesis is true.

The explanation for these results lies in the constants. The two-layer structure of our data-structure means that it only guarantees each node to be a quarter filled, while the traditional B-tree guarantees half-filled nodes. This effectively decreases the branch-out factor of the tree, increasing the height. It also increases the total size of the data-structure, explaining why it hits the disk that much earlier.

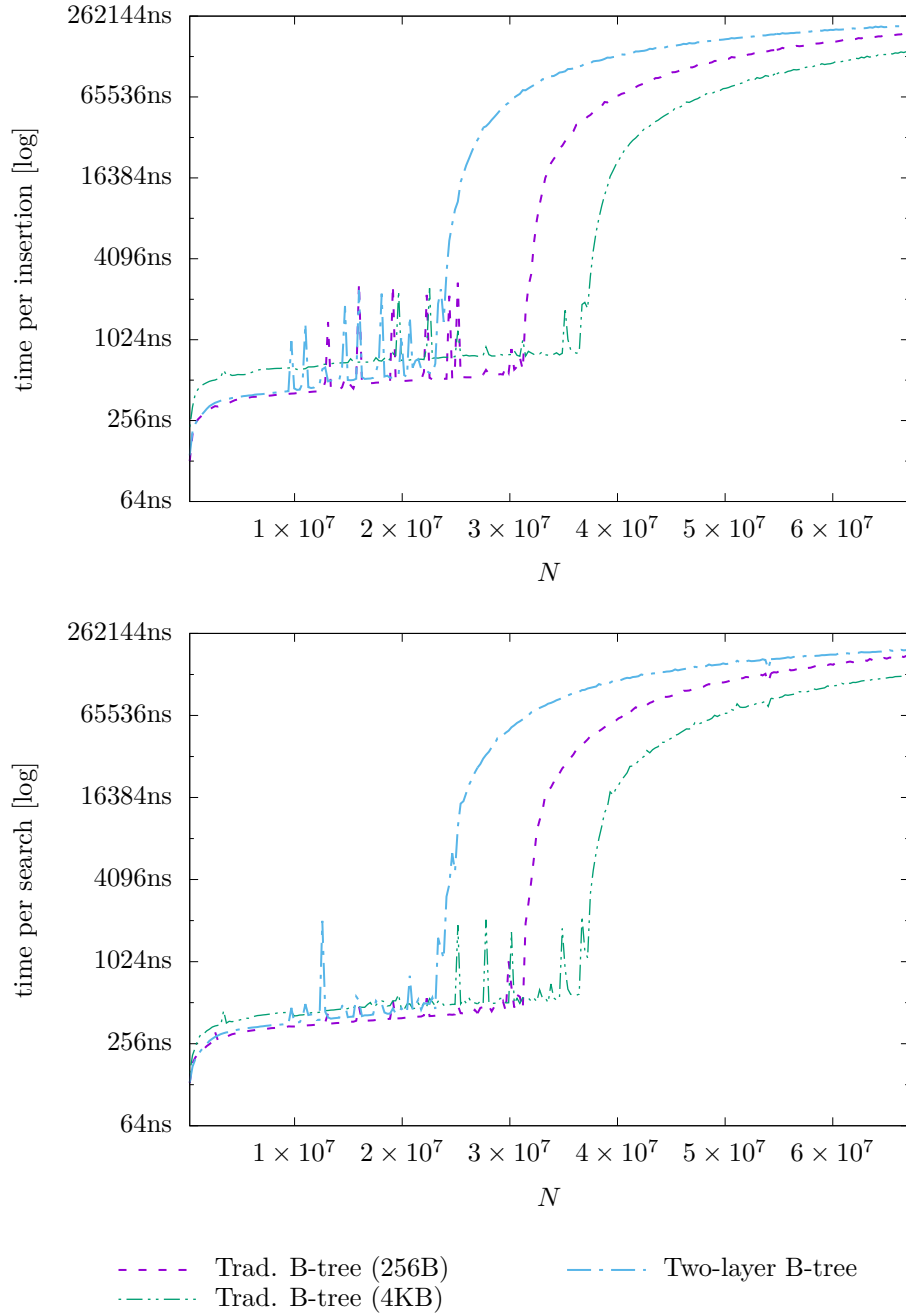


Figure 2.1: The results of the experiments on tree-based map performance. Top graph shows time per insertion, bottom graph shows time per search, both as a function of m . Note that time is plotted on a logarithmic scale.

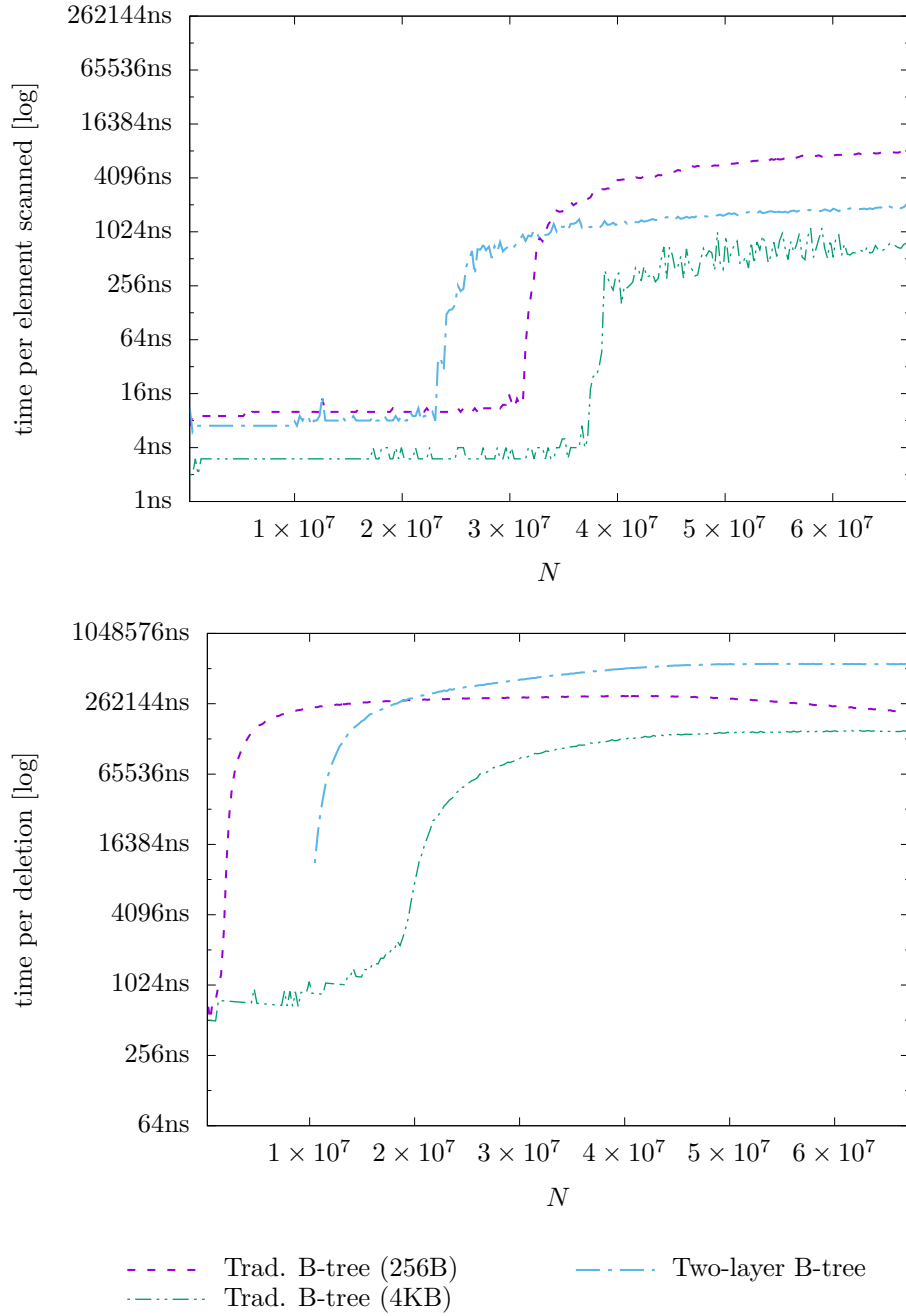


Figure 2.2: The results of the experiments on tree-based map performance. Top graph shows time per element scanned, bottom graph shows time per deletion, both as a function of m . Note that time is plotted on a logarithmic scale.

The two-layer B-tree should still have a larger branch-out factor than the B-tree optimized for disk though. An effect to take into account to explain this is that when a page is loaded from disk for the cache-optimized B-tree, there is a chance that that page also contains nodes that are needed further down the tree. This chance decreases as the data-structure grows, so it might still be, as mentioned above, that the two-layer B-tree performs better than the traditional one for cache when the dataset increases in size.

The third graph, figure 2.2 above, showing the time per element scanned, also shows the sharp increases as the data-structures hit the disk, at the same times as before. The sections in between are flatter because a scan does not need to traverse the height of the tree. The determining factor here is how much data is packed into each block transfer. The disk-optimized B-tree performs the best on that metric, also taking better advantage of cache prefetching. The two-layer B-tree and the cache-optimized B-tree perform about equally before hitting the disk, while the two-layer B-tree performs markedly better after hitting the disk.

An interesting thing to note here is that while the two-layer B-tree and the disk-optimized B-tree have almost flat graphs after hitting the disk, only increasing slightly due to the decreasing chance that a given page is already loaded to memory, the cache-optimized B-tree has a bit more marked increase after hitting the disk. We suspect that this is due to the effect noted above, that a given loaded page has a chance of holding nodes that will be needed soon, and that this chance decreases as the size increases.

The fourth graph, figure 2.2 below, showing the time per deletion, should be read from right to left, as we started out with a large data-structure and deleted elements. A small error in the logging code has resulted in decreased precision in this data, so that the apparent increase in performance for larger set sizes should be discounted. We see here also the missing data for our the two-layer B-tree, due to the bug in our implementation mentioned above.

We also see different times for ‘hitting the disk’, since the process is reversed. These indicate that all of the data-structures are worse at constraining themselves when scaling down than when scaling up, the cache-optimized B-tree remarkably so. We suspect that in the latter case this is due to some unknown implementation detail.

2.5 Discussion

As presented here, the performance of our data-structure is not impressive. The decreased branching factor caused by the lower guaranteed load per disk page is a large part of the explanation for this. This could be partly alleviated by using three-way merging in the inner trees, allowing for a two-thirds fill factor. We refrained from pursuing this idea since the complexity of the implementation was already taking us too much time, and that would add quite a bit more.

It should also be noted that all of these experiments were conducted in isolation, with the entirety capacity of the disk and cache lines available for use. The merit of our design, as opposed to for example the disk optimized B-tree, is that it

uses fewer cache transfers. But since a cache transfer is so much faster than a disk transfer, the effect of this on the execution time drowns. An application that uses the cache heavily for other things might benefit.

A note on the generality of our solution is also in order. We designed the data-structure, and the implementation, to be able to take B_m and B_c , as well as the data-types stored, as parameters. For the typical values of 4KB and 256B (to take advantage of prefetching) that are optimal on almost any modern desktop system, the internal trees will in most cases be very flat, in some cases only having the root and many leaves. If implementing this for a specific application, expected to run on such a system, it might be worth considering just using a single fat root, getting rid of the complexity required to handle a full tree.

Chapter 3

Hash-based solutions

Another approach to the map problem is to use a hash table. A hash table consists of an array of some size m (often a power of two), and an associated hash function $h : U \rightarrow M$, where U is the universe of possible keys to insert and $M = \{0, 1, \dots, m - 1\}$. For some element with key x , the position $h(x)$ corresponding to the hash value of the key is called the *bucket* of that element, and the basic idea is to place the element in that position. Unfortunately, if the keys are unknown when selecting h , we can not guarantee that two keys x and y do not result in a *collision*, that is, that $h(x) = h(y)$. The problem of what to do in that situation is what gives rise to the plethora of different hash tables. In this section we will explore addressing schemes for collision resolution in hash tables that are advantageous with regard to I/O efficiency.

3.1 Linear probing

3.1.1 Overview

The traditional I/O friendly strategy for collision resolution is *linear probing*. Linear probing is an *open addressing* scheme, meaning that all elements are stored in a single array and so not necessarily in their corresponding bucket (*closed addressing* associates some datastructure with each bucket, to hold the elements that hashed there). The approach of linear probing is the simple one of storing the element in the first unoccupied position after its bucket, treating the array as circular if reaching the end.

The procedure for searching for an element with key q is then to start at $h(q)$, and test each of the positions from that until we find either the element (for a successful search) or an unoccupied position (for an unsuccessful search). The procedure for inserting an element with key q is to find the first unoccupied position after $h(q)$, and insert the element there. Removal needs a bit more work, since just leaving an empty position could make searches for elements stop too early. Let e be the empty position. Removal searches forward from that position, and each time it encounters an element with $h(x) \leq e$, where x is

the key of that element, it moves it into e , and sets e to the previous position of that element. When it encounters an empty position it is done. Note that in the absence of some value $\lambda \in U$ to denote an empty position, each position in the array will need at least one bit besides the space for the elements for this purpose.

Unless the maximum number of elements is known beforehand, an open addressing scheme needs some strategy for resizing the array. Let n be the amount of positions currently occupied in the hash map, and $\alpha = n/m$ the proportion of occupied positions or *load-factor*. When the load-factor exceeds some threshold, a new larger hash map is initialized (typically double the size, since we are dealing with powers of two) and all the elements are re-inserted into that one. Likewise, if the load-factor goes below some threshold, the same is done but with a new hash map of half the size.

3.1.2 Analysis

Call a group of consecutive occupied positions in the array a *run*. The deciding factor for the work and I/O cost of all three procedures of a linear probing hash map (search, insert and removal) is the maximum run-length R . Without taking resizes into account, all these costs are worst-case $O(R)$ and $O(R/B)$, respectively. The cost of re-inserting upon a resize can be amortized over the insertions or removals needed to reach that proportion since the last resize, so that we get amortized costs of $O(R)$ work and $O(R/B)$ I/O for insertion and removal. For example, if we double the capacity each time we exceed the threshold, half the elements that are to be re-inserted will have been inserted since the last doubling, so we can charge two insertions extra for each insertion and we will have paid for the re-insertions. Note that the I/O bounds here are cache-oblivious, that is, they hold regardless of the actual parameters. It has been shown that these bounds therefore hold for all levels of a cache-hierarchy [11].

Assuming a truly random hash function, in a seminal result in algorithm analysis Knuth showed that the expected value of R is $O(1)$, given that the load-factor does not exceed some threshold [7, p. 536-537]. We do not of course have any practically implementable truly random hash functions, and a considerable amount of work has since been put into the analysis of linear probing using hash functions with more practical properties.

3.1.3 Recent results

The most influential paradigm for the analysis of practical hash functions is that of k -independence, introduced by Wegman and Carter [13]. Briefly, a family of hash functions is k -independent if for some fixed key $x \in U$, $h(x)$ is uniformly distributed in M , and for distinct keys $x_1, \dots, x_k \in U$, $h(x_1), \dots, h(x_k)$ are independent random variables. Working purely under this paradigm it has been shown that 5-independence is sufficient [8] and necessary [9] for the expected value of R , and by extension the expected work and I/O cost, to be $O(1)$, again

under the assumption that the load factor does not exceed some threshold. 5-independence is usually too slow for practical use in performance-critical data-structures though.

On a more positive note, it has also been shown that simple tabulation hashing [13], while only 3-independent, has other properties that gives an expected R of $O(1)$ under the same assumption [10]. Simple tabulation hashing is based on viewing a key x as a vector of c characters, x_1, \dots, x_c , and using those characters as lookups into c tables T_1, \dots, T_c containing randomly drawn values from M . The values found in this way are combined into the final hash value by bitwise exclusive-or, that is, $h(x) = T_1[x_1] \oplus \dots \oplus T_c[x_c]$.

3.2 Hopscotch hashing

3.2.1 Overview

We turn now to an interesting alternate scheme for collision resolution called *hopscotch hashing* [4]. Like linear probing it is an open addressing scheme, but hopscotch hashing makes the guarantee that elements are not stored too far away from their bucket, and so offers hard worst-case bounds for the work and I/O cost of search and removal. Unfortunately, the analysis in the original article is confusing and probably faulty, so we shall try to determine if the claims made in it hold or not.

The central idea is that each position of the array p is associated with a *neighbourhood*, consisting of positions $\{p, p + 1, \dots, p + H - 1\}$ in the array, where H is some constant chosen for the implementation. Typically H will match the word size of the machine with one bit left for indicating whether a value is stored in that position. An element will always be found in the neighbourhood of its bucket, if present. Each position in the array contains an H -bit array, the *hop-information* array, that indicates which of the H positions in the neighbourhood contains elements that are in the bucket of that position. (As noted earlier, any open addressing scheme where no value $\lambda \in U$ is reserved for indicating an empty position will need at least one bit of extra information per position in the array. Due to alignment, this will typically end up using more than that — so the extra data for the hop-information array might not be as big of a memory hit as it seems at first.)

Searching for an element with key x is a matter of looking in the hop-information array at $h(x)$ to see which positions in the neighbourhood contain elements with that hash-value, and checking each of those. This takes $O(H)$ work and $O(H/B)$ I/Os, both of which are constant since H is constant. Removing an element with key x is done by finding it, removing it from that position, as well as unsetting the bit corresponding to that position in the hop-information array at $h(x)$. Again this takes $O(H)$ work and $O(H/B)$ I/Os.

The procedure for insertion is obviously more involved, in order to maintain this structure. For inserting an element with key x , probe forward from $h(x)$ until an unoccupied position e is found, exactly as in linear probing. If $e \leq h(x) + h - 1$

we can simply insert the element at that position and mark it in the hop-information array at $h(x)$. If not, start at $e - H - 1$, check if the element at that position could be moved to e without leaving its neighbourhood, and if so, do that, marking the change in the hop-information array of the corresponding bucket. If not, we go to the next position and check if that one can be moved, and so on. Once such a move has been made we have a new empty position e' closer to $h(x)$ and we can start over. In this way we move the empty spot towards the neighbourhood of the element we are inserting, until we can insert it as usual. If at some point we can find no element to move forward, we increase the size of the map and rehash all elements.

An advantage of hopscotch hashing is that it works very well in a multi-threaded environment, insertions and removals are guaranteed to only touch the positions mentioned in the hop-information array of their bucket. Therefore, a very fine-grained locking mechanism can be implemented, basically having a lock for each bucket.

3.2.2 Analysis

The first thing to note here is that, assuming the same hash function, the exact same positions of the array will be occupied under hopscotch hashing as under linear probing — the same procedure is used to find the empty position, and after that any moves are only switching one occupied position for another. Thus, if we assume that the load-factor does not exceed a certain proportion, the results on the maximum length of consecutive occupied positions R under linear probing are directly applicable to hopscotch hashing as well.

The work and I/O cost of looking for an empty position is obviously $O(R)$ and $O(R/B)$ like in linear probing. The second part of insertion, moving the elements around to move the empty space into the correct neighbourhood, accesses the same part of the array as the first, so that the total I/O cost of insertion is $O(R/B)$. The worst-case work cost of moving elements around after the search is $O(R \times H)$, since it is conceivable that we only find an element to at the end of each search of the H positions behind the empty one. We regard H as a constant, so that this is $O(R)$ as well.

The same procedure can be used for resizing the array under hopscotch hashing as under linear probing, and the same amortized analysis can be made, assuming we can still make the assumption that the load-factor reaches some constant proportion before the resize is done. What is left then is to justify the assumption that the load factor does not exceed a certain proportion (this is easy, since we can maintain the rule that a resize is done once that proportion is reached), and that the probability of a resize before that proportion is reached is low. The latter is the subject of the following sections.

3.2.3 Forced resize

Theoretical perspective

Resizing happens either when the load-factor goes below some proportion (sizing down), exceeds some proportion, or when we cannot move an empty position into the desired neighbourhood (sizing up). The first two cases need not concern us anymore. Let f denote the first occupied position in the run containing $h(x)$, and e the empty position after that run. If $e > h(x) + H - 1$ we need to move an element in one of the positions $\{e - H + 1, \dots, e - 1\}$ into e to advance the algorithm. This is possible if any of the elements in those positions have a hash-value in that same range. Conversely, if it is *not* possible to make such a move, all of the elements in positions $\{f, \dots, e - 1\}$ must belong to positions $\{f, \dots, e - H\}$. So the probability of an insertion resulting in a failed move, and thus a resize, is equal to the probability of that situation occurring.

Let us start by assuming a truly random hash function, like the traditional analysis of linear probing. This gives us m^n equally likely ways to distribute the elements. Consider first the probability of $k + H$ elements hashing into some fixed set of adjacent buckets of length k . There are $\binom{n}{k+H}$ ways of choosing $k + H$ elements from the n elements, and k^{k+H} ways of distributing them among k buckets. Since we want exactly k adjacent buckets, the remaining elements can not go into the two adjacent buckets, but may go anywhere else (there can be more than $k + H$ elements in the chosen set), so there is $(m - 2)^{n-k-H}$ possible distributions of those. Thus there is a probability

$$\frac{1}{m^n} \binom{n}{k+H} k^{k+H} (m-2)^{n-k-H}$$

of satisfying our condition for some specific set of length k . For each k there are m possible sets of adjacent buckets, and k can range from 1 to $n - H$, so by a union bound we have at most a probability

$$\begin{aligned} & \frac{m}{m^n} \sum_{k=1}^{n-H} \binom{n}{k+H} k^{k+H} (m-2)^{n-k-H} \\ &= \frac{m}{m^n} \sum_{k=H+1}^n \binom{n}{k} (k-H)^k (m-2)^{n-k} \end{aligned}$$

of satisfying our condition in total. This rather complex expression does not look promising. Using asymptotic notation we can get some ideas of the determining factors. By $\binom{n}{k} = O((n/k)^k)$ this is

$$\begin{aligned} & \frac{m}{m^n} \sum_{k=H+1}^n O((n/k)^k) (k-H)^k O(m^{n-k}) \\ &= m \sum_{k=H+1}^n O((n/k)^{k+H}) (k-H)^k O(m^{-k}) \\ &= m \sum_{k=H+1}^n O((\alpha/k)^k) (k-H)^k. \end{aligned}$$

From this it seems clear that the probability of a forced resize is dependent on m — an unfortunate result, although from experience we know that forced resizes are very uncommon for small to medium problem sizes. We will not pursue this analysis any further, opting instead to perform experiments to explore this dependence further. In this way we can estimate the constants involved, in order to estimate when this dependence starts to become a problem.

Empirical perspective

In order to establish a hypothesis on the expected load-factor on forced resize for different values of m and H , we performed the following simple experiment. We initialized an empty hopscotch map of some size. We then bypassed the hash function and simply inserted random numbers directly into the map, acting as the hash values of hypothetical elements. The random values were drawn from the standard C++ function `std::mt19937_64` which is an implementation of a so-called Mersenne Twister, a high quality random number generator [6]. This approach was chosen as the closest approximation of a truly random hash function that was practical, in order to know the best possible behaviour of the hopscotch map.

We bypassed the automatic rehashing at the maximum load-factor to let it fill up until a forced rehash was needed, and recorded the load-factor at that point. To speed up the experiments for large values of m we produced a stripped down version of the data-structure which does not store the actual values. In this way we were able to perform the experiment with m ranging from 2^8 up to 2^{30} . The code for this experiment is listed in appendix B.2.

We performed the experiment six times for $H = 7, 15$ and 31 , and three times for $H = 63$ due to the long time it took to run it (for $H = 63$, 2^{30} elements amounts to 8GB of hop-information arrays). The results for $H = 7$ and $H = 15$ had deviations of up to 0.2 from the mean, and a standard deviation of 0.0405 and 0.0306 respectively. For $H = 31$ the maximal deviation was 0.071 and the standard deviation 0.0181, while for $H = 63$ the maximal deviation was 0.028 and the standard deviation 0.0092.

The mean values of α on forced resize are plotted in figure 3.1, with m on the horizontal axis and α on the vertical. The horizontal axis is logarithmic on both graphs, while the upper graph has α plotted on a linear axis and the lower graph has it on a logarithmic axis.

Let α_{resize} be the load-factor on a forced resize. The lower graph makes it clear that the expected α_{resize} is close to being a power of m . Functions are plotted to approximate the expected value of α_{resize} for each value of H , minimizing the deviation of our results. The coefficients and exponents in those functions seem to be functions of H , so that we get the following approximate relation:

$$E[\alpha_{\text{resize}}] = (1.05 + 2/H)m^{-2/(3H)}.$$

If we accept this, we can try to predict when forced resizes will become a problem. The maximum load-factor used in our implementation is 0.7. We have that

$$(1.05 + 2/63)m^{-2/(3 \times 63)} = 0.7 \Rightarrow m \approx 7.29 \times 10^{17} > 2^{59},$$

while

$$(1.05 + 2/31)m^{-2/(3 \times 31)} = 0.7 \Rightarrow m \approx 2.46 \times 10^9 > 2^{31}.$$

Experiments using actual values with a simple tabulation hash function showed no sign of giving any different results, though that does of course not prove that there are no bad cases.

In conclusion, this theoretical down-fall of the hopscotch hashing scheme is not likely to have any significant impact on practical use for small to medium sized data-sets.

3.3 Implementation

The implementation of hopscotch hashing was done in an earlier project. It follows the interfaces of `std::unordered_set` and `std::unordered_map` as defined in the C++ standard [6], though it does not completely implement all of the functionality there. There are no technical reasons that it could not implement it all, it is simply a matter of filling in the blanks.

The implementation uses the special function `__builtin_ffsl`, which is not part of the C++ standard, to efficiently access the individual bits of the hop-information array. This function serves as an interface to the ‘find first set’ instruction available on many modern processors. It greatly improves access speeds when the CPU is the main bottleneck. This function is available in both the `gcc` and `clang` compilers, and most other major compilers offer the same functionality under a different name. The same functionality could of course be implemented without a special instruction, albeit somewhat slower.

The implementation of linear probing was done for this project. It borrows a lot of the boilerplate from the hopscotch implementation, and follows the same interfaces.

Both the hopscotch map and the linear probing map needs some extra data for each position in the array. For the hopscotch map this includes the hop-information array, while linear probing needs only the one boolean that shows if a value is stored in that position or not. When implementing both data-structures we are then faced with the choice of interleaving the extra data with the elements themselves, or putting it into a separate array. The latter is best for memory utilization, since it allows us to pack the data better. This is especially true for linear probing, where it allows us to store the bits tightly packed, only using 1 bit per bucket, as opposed to the minimum of 8 bits per bucket needed when interleaving (because of addressing, each pointer must start on a byte on most architectures). On the other hand, interleaving makes better use of the I/Os, since all the data needed when accessing some part of the structure is stored in the same place. It also has the advantage, in the case of linear probing, of not needing slow bit-twiddling operations to access the boolean. These considerations led us to interleave the data for our implementations in this project.

The full implementation of both hash maps is listed in appendix A.5 and A.6.

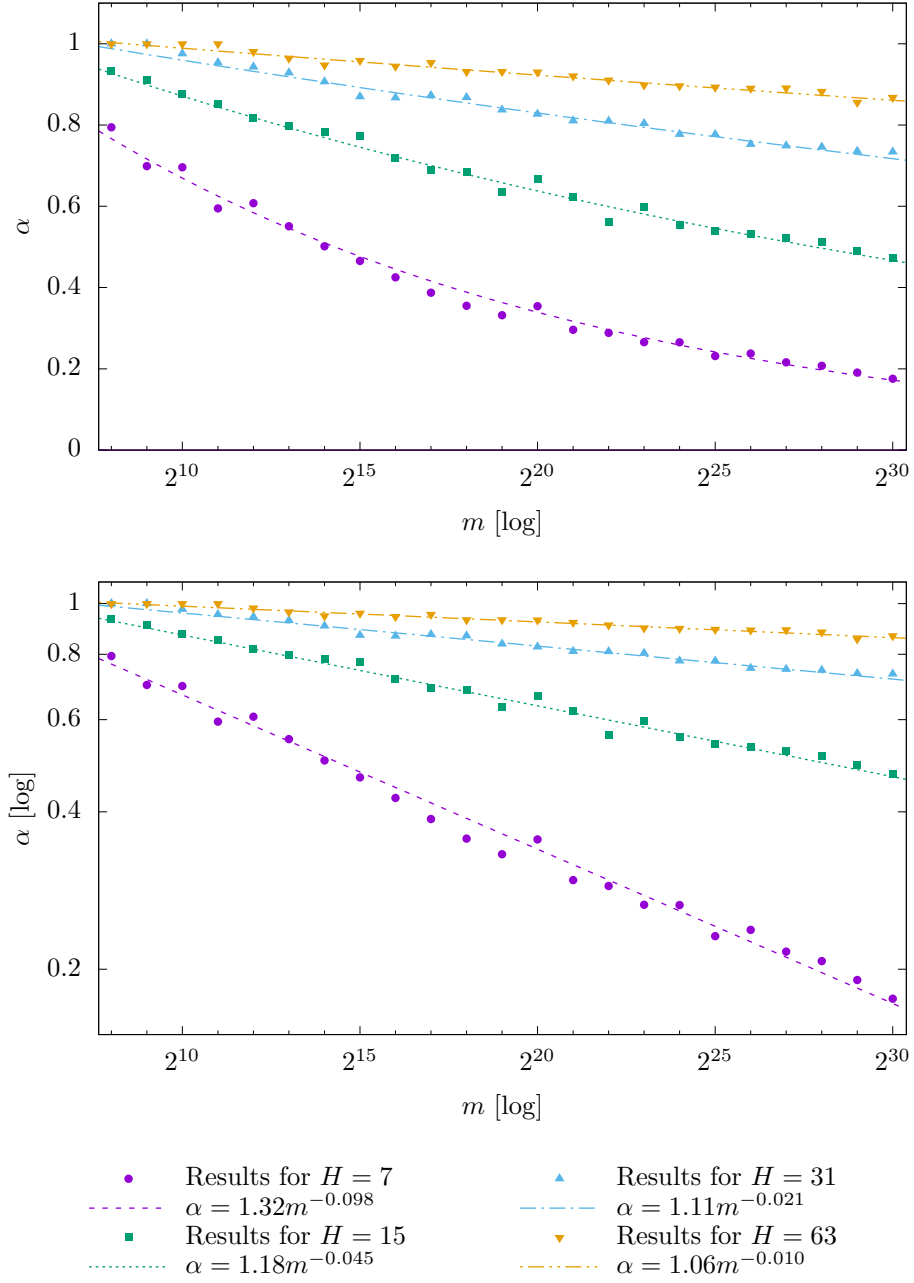


Figure 3.1: The results of the experiments on forced resize in hopscotch hashing. The upper graphs has the result plotted on a logarithmic horizontal axis and linear vertical axis. The lower graph has both axes logarithmic (see discussion).

3.4 Experiments

3.4.1 Hash function

For testing the hash maps, we used the tabulation hashing scheme described above. As we have described, in addition to being 3-independent, this scheme has been shown to have some very nice properties when using linear probing, which also carries over to hopscotch hashing. In addition, it is very simple to implement:

```

1 return t1[static_cast<uint8_t>(x)] ^
2       t2[static_cast<uint8_t>(x>>8)] ^
3       t3[static_cast<uint8_t>(x>>16)] ^
4       t4[static_cast<uint8_t>(x>>24)] ^
5       t5[static_cast<uint8_t>(x>>32)] ^
6       t6[static_cast<uint8_t>(x>>40)] ^
7       t7[static_cast<uint8_t>(x>>48)] ^
8       t8[static_cast<uint8_t>(x>>56)];

```

Here x is the input as before t_1, \dots, t_8 are each arrays of 256 random 64-bit integers. That is a total of 16kB, which fits in the L1-cache of a modern CPU, but the tables will of course have to compete with other data for a place in the cache, either being pushed out or pushing something else out.

All of the random 64-bit integers used for the hash function were drawn from `random.org`. The random numbers offered there are based on atmospheric noise. The full implementation of the hash function (excluding the 16kB of random numbers) is listed in appendix A.7.

3.4.2 Test data

Since the hash maps, as opposed to the B-trees, are sensitive to the actual values of the keys inserted, we have performed our experiments using two different sets of test data. The first was a list of pseudo-random 64-bit integers, as generated by the C++ standard library function `std::rand` (run twice and concatenated, since `std::rand` produces 32-bit integers). In theory, a completely random input makes the randomization of the hash function itself irrelevant. Since the numbers inserted here are not completely random (they are generated by a computer) this is not entirely so. Nevertheless, this dataset should not cause problems for any of our hash functions, and is intended show the performance of the data-structures on non-problematic input.

The other set of test data is the list of integers $\{0, \dots, n-1\}$, in a pseudo-random permutation, as generated by the C++ standard library function `std::shuffle`. Keys that are drawn from a densely packed interval is a case that often appears in practice, and is known to cause unreliable performance for some simple hashing schemes. This dataset is intended to model that case. Previous experiments show that this type of input does not cause problems for simple tabulation when used in linear probing [10].

3.4.3 Performance results

We carried out the same experiment for our hash maps as was described in the section on B-trees, with the exception that there was no scan step. The results are plotted in figures 3.2, 3.3 and 3.4. First note that none of the tests show any significant difference in the performance between the two data-sets. This is in line with previous results.

All of the graphs are remarkably similar, which is to be expected since all of the operations have expected constant time and I/O costs. There are spikes at the resizes in the insertion and erase graphs. The search graph for linear probing does not have these spikes. The search graph for hopscotch hashing has a very large spike around 1.2×10^7 coinciding with a resize, most probably due to cache invalidation from the resize.

Linear probing performs slightly better in all aspects, most probably due to the slightly smaller size. This is most visible just after hitting the disk where the more compact linear probing map has a larger chance of finding the section it needs already loaded in memory, due to this. This gap begins to close after the next resize.

3.5 Discussion

If the multithreading advantages of hopscotch hashing are desirable in a given application, with a small to medium amount of elements expected to be inserted, it seems to be a good solution, despite the theoretical possibility of early resizing. No early resizes were observed in practice, though it is still possible that bad cases exist. We suspect that the bad cases would be similar to bad cases causing overlong probes for linear probing, so that should be taken into account. For single-threaded use, linear probing is the best choice for an I/O efficient hash map in almost all cases.

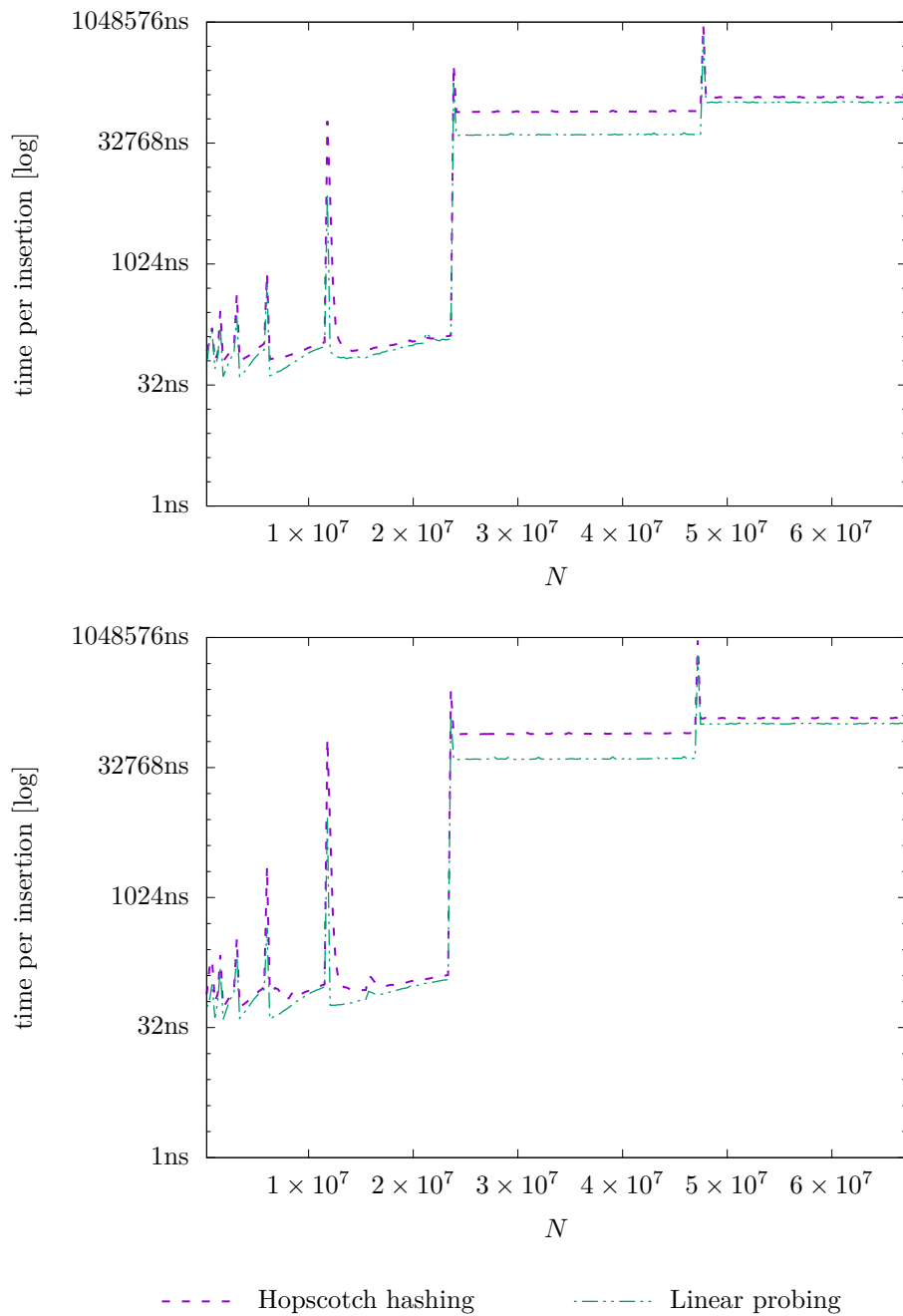


Figure 3.2: The results of the experiments on hash-based map performance. Top graph shows time per insertion with the random data-set, bottom graph shows same for the dense data-set, both as a function of m . Note that time is plotted on a logarithmic scale.

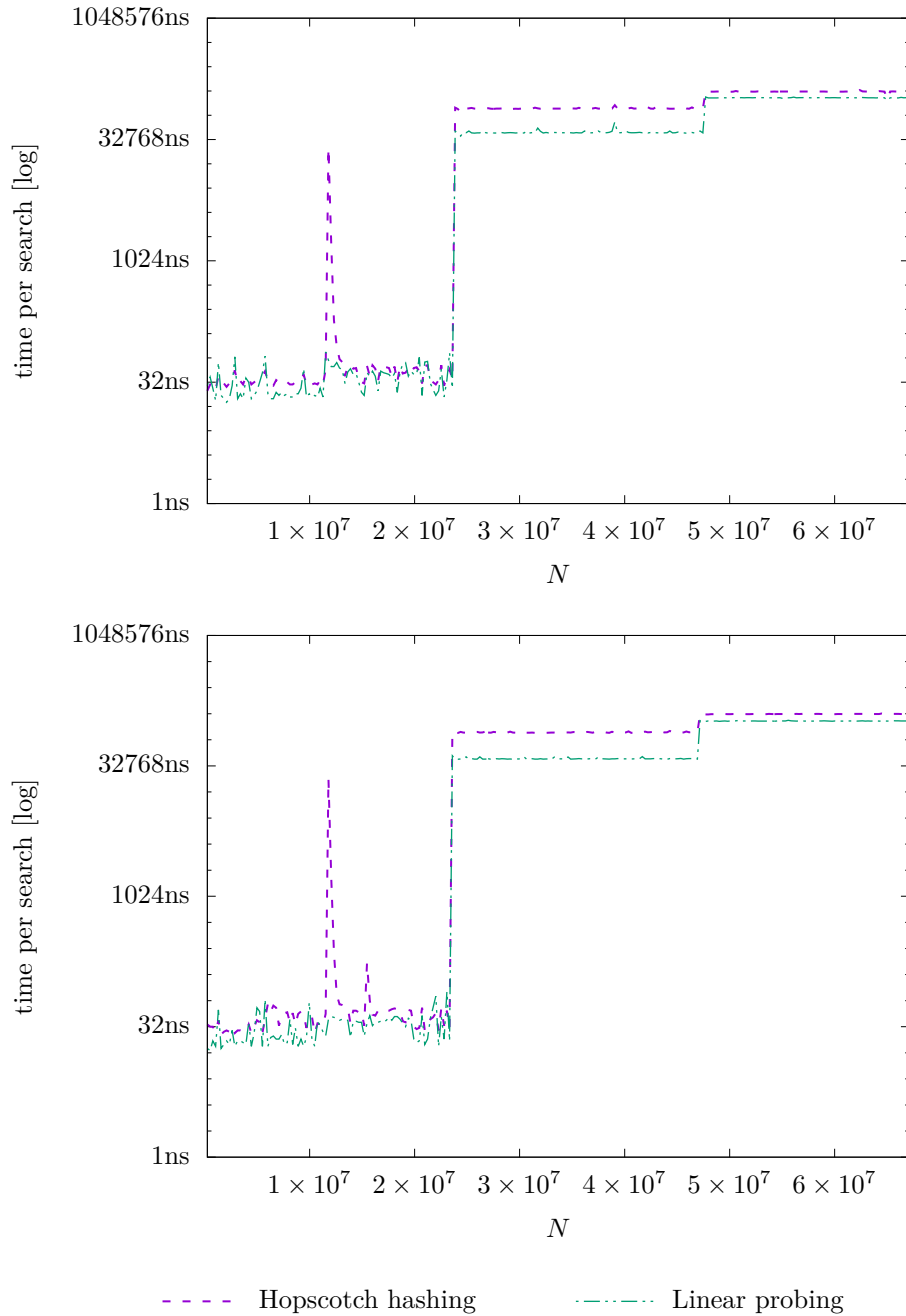


Figure 3.3: The results of the experiments on hash-based map performance. Top graph shows time per search with the random data-set, bottom graph shows same for the dense data-set, both as a function of m . Note that time is plotted on a logarithmic scale.

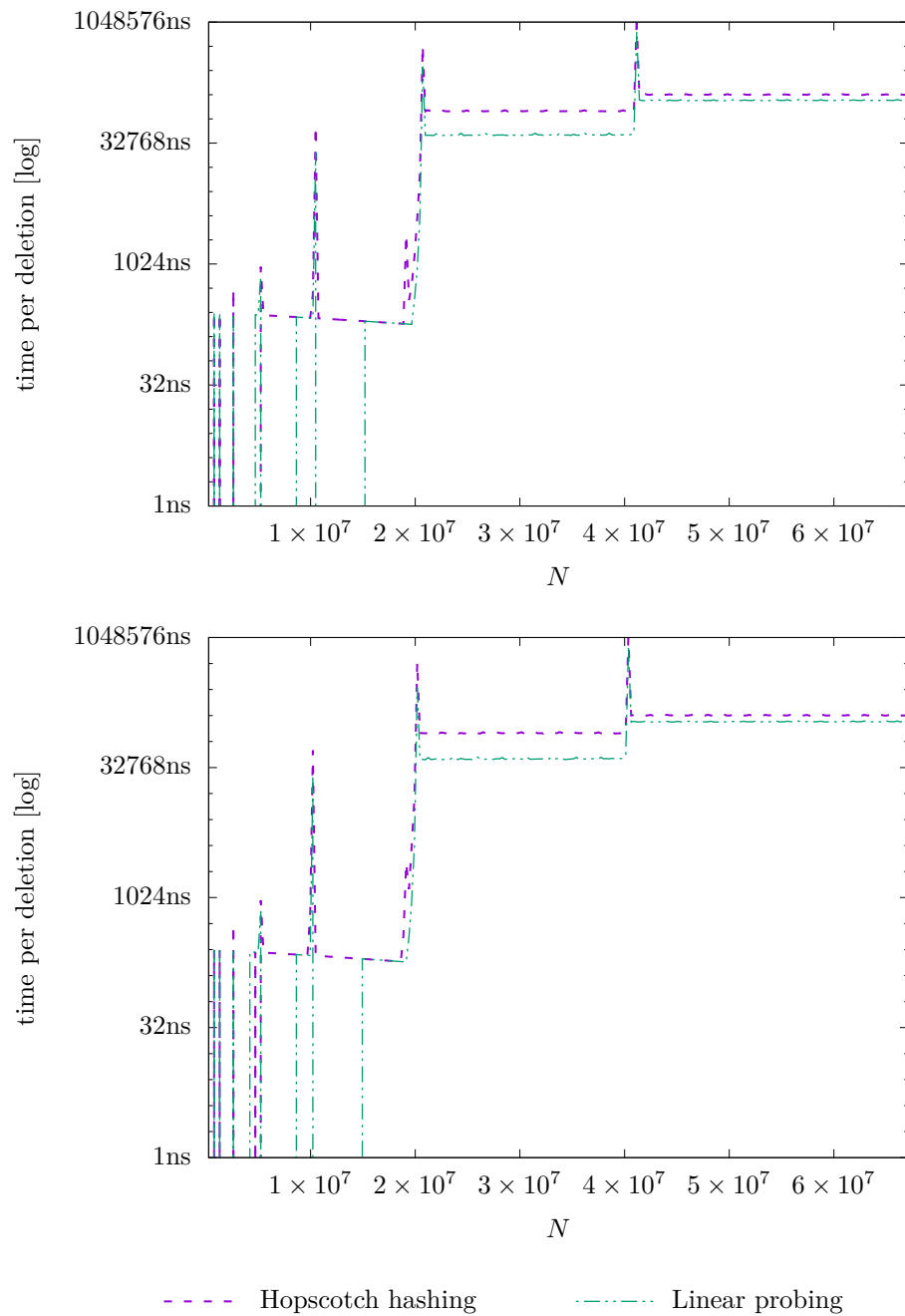


Figure 3.4: The results of the experiments on hash-based map performance. Top graph shows time per deletion with the random data-set, bottom graph shows same for the dense data-set, both as a function of m . Note that time is plotted on a logarithmic scale.

Chapter 4

Conclusion and further work

We have looked at two novel approaches to the map/set problem concentrating on I/O efficiency, and performed experiments to explore the advantages and disadvantages of these approaches, as well as the performance of the more traditional I/O efficient solutions. Comparing the experimental results it is clear that if scanning is not needed, and if the spikes in insertion and deletion times can be tolerated, a hash-based solution is better, while the tree-based solutions provide better worst-case running times for insertion and deletion at the cost of higher average running-times.

The two-layer B-tree did not provide promising results as a general solution, though we still feel it should still be considered for more specific needs, where the constants are known beforehand, so that it can be tailored more explicitly. Looking at three-way merging in the inner trees could improve the space efficiency and so the general performance of the data-structure.

A more rigorous analysis of hopscotch hashing seems difficult, especially because the constants would have to be maintained to show that the early resizing is only expected to happen at large sizes, as our experiments showed us. Against this theoretical disadvantage, hopscotch hashing has significant advantages for a multi-threaded environment, and should probably be considered when looking for a fast map or set in such a setting. For a single-threaded environment, linear probing is better.

Chapter 5

Bibliography

- [1] A. Aggarwal and J. S. Vitter. *The input/output complexity of sorting and related problems*. In Communications of the ACM, 31(9): 1116–1127, 1988.
- [2] R. Bayer and E. McCreight. *Organization and maintenance of large ordered indexes*. In Acta Informatica, 1(3): 173–189, 1972.
- [3] R. W. Floyd. *Permuting information in idealized two-level storage*. In Complexity of Computer Computations: 105–109, 1972.
- [4] M. Herlihy, N. Shavit, and M. Tzafrir. *Hopscotch hashing*. In Distributed Computing: 350–364, 2008.
- [5] C. A. R. Hoare. *Data structures in two-level store*. In Proceedings IFIP Congress, 1: 366–398, 1969.
- [6] *ISO/IEC 14882:2014(E) Programming Language C++*. International Organization for Standardization, 2014.
- [7] D. E. Knuth. *The art of computer programming: Volume 3, Sorting and searching, 2nd edition*. Addison-Wesley, 1997.
- [8] A. Pagh, R. Pagh, and M. Ružić. *Linear probing with constant independence*. In SIAM Journal on Computing, 39(3): 1107–1120, 2009.
- [9] M. Pătraşcu and M. Thorup. *On the k -independence required by linear probing and minwise independence*. In Proceedings 37th International Colloquium on Automata, Languages and Programming: 715–726, 2010.
- [10] M. Pătraşcu and M. Thorup. *The power of simple tabulation hashing*. Journal of the ACM, 59(3), 2012.
- [11] H. Prokop. *Cache-Oblivious Algorithms*. Thesis, Department of Electrical Engineering and Computer Science, MIT, 1999.
- [12] J. S. Vitter. *Algorithms and data structures for external memory*. Foundations and Trends in Theoretical Computer Science, 2(4): 305–474, 2008.
- [13] M. N. Wegman and L. Carter. *New classes and applications of hash functions*. Journal of Computer and System Sciences, 22(3): 265–279, 1981.

Appendices

Appendix A

Implementation code

A.1 double_tree.hpp

```
1 #pragma once
2 #include "double_tree_page_node.hpp"
3 #include <iterator>
4 #include <vector>
5
6 namespace double_tree
7 {
8
9     namespace detail
10    {
11
12        struct tree_position
13        {
14            tree_position() = default;
15
16            tree_position(const tree_position& other):
17                page{other.page},
18                sub_position{other.sub_position}
19            {}
20
21            tree_position(void* page_, page_position sub_position_):
22                page{page_},
23                sub_position{sub_position_}
24            {}
25
26            bool operator==(const tree_position& other) const
27            {
28                return page == other.page && sub_position == other.sub_position;
29            }
30
31            bool operator!=(const tree_position& other) const
32            {
33                return page != other.page || sub_position != other.sub_position;
34            }
35
36            void* page;
37            page_position sub_position;
38        };
39
```

```

40 template <
41     typename Element,
42     typename ElementWrite,
43     typename Key,
44     typename KeyExtract,
45     typename ValExtract>
46 struct kernel
47 {
48 private:
49     // Defined below.
50     template <typename T> class iterator_template;
51
52     // Auxiliary structures for the page nodes. A stem node does not need any
53     // extra data, while the leaf nodes of a tree are linked together in a
54     // linked list, so they need pointers to their previous and next nodes
55     struct stem_aux
56     {};
57
58     struct leaf_aux
59     {
60         void* prev_pointer;
61         void* next_pointer;
62     };
63
64     using stem_node = PageNode<
65         std::pair<const Key, void*>,
66         std::pair<Key, void*>,
67         Key,
68         extract::first,
69         extract::second,
70         stem_aux>;
71
72     using leaf_node = PageNode<
73         Element,
74         ElementWrite,
75         Key,
76         KeyExtract,
77         ValExtract,
78         leaf_aux>;
79
80     // DATA
81
82     void* root_pointer;
83     leaf_node* min_leaf_pointer;
84     leaf_node* max_leaf_pointer;
85     int stem_levels;
86     KeyExtract key_extract_;
87     ValExtract val_extract_;
88
89 public:
90     // MEMBER TYPES
91
92     using key_type          = Key;
93     using value_type       = Element;
94     using reference        = value_type&;
95     using const_reference  = const value_type&;
96     using pointer          = value_type*;
97     using const_pointer    = const value_type*;
98     using iterator         = iterator_template<value_type>;
99     using const_iterator   = iterator_template<const value_type>;
100
101     // CONSTRUCTOR

```

```

102
103     kernel()
104     : root_pointer{new leaf_node},
105       min_leaf_pointer{get_leaf_pointer(root_pointer)},
106       max_leaf_pointer{get_leaf_pointer(root_pointer)},
107       stem_levels{0}
108     {
109         auto& root = get_leaf(root_pointer);
110         root.aux.prev_pointer = nullptr;
111         root.aux.next_pointer = nullptr;
112     }
113
114     // ACCESSORS
115
116     public:
117         auto& operator[](const Key& find_key) {
118             auto position = find_implementation(find_key);
119             return val_extract_(
120                 get_leaf(position.page).elem(position.sub_position));
121         }
122         const auto& operator[](const Key& find_key) const {
123             auto position = find_implementation(find_key);
124             return val_extract_(
125                 get_leaf(position.page).elem(position.sub_position));
126         }
127
128     private:
129         Element& elem(const tree_position position) {
130             return get_leaf(position.page).elem(position.sub_position);
131         }
132         const Element& elem(const tree_position position) const {
133             return get_leaf(position.page).elem(position.sub_position);
134         }
135
136         static stem_node* get_stem_pointer(void* const pointer) {
137             return static_cast<stem_node*>(pointer);
138         }
139         static leaf_node* get_leaf_pointer(void* const pointer) {
140             return static_cast<leaf_node*>(pointer);
141         }
142
143         static stem_node& get_stem(void* const pointer) {
144             return *get_stem_pointer(pointer);
145         }
146         static leaf_node& get_leaf(void* const pointer) {
147             return *get_leaf_pointer(pointer);
148         }
149
150         static const stem_node& get_stem(const void* const pointer) {
151             return *static_cast<const stem_node*>(pointer);
152         }
153         static const leaf_node& get_leaf(const void* const pointer) {
154             return *static_cast<const leaf_node*>(pointer);
155         }
156
157         // PREDICATES
158     public:
159         bool empty() const
160         {
161             return stem_levels == 0 && get_leaf(root_pointer).empty();
162         }
163

```

```

164     // OPERATIONS
165 public:
166     iterator find(const Key& find_key) {
167         return {this, find_implementation(find_key)};
168     }
169
170     const_iterator find(const Key& find_key) const {
171         return {this, find_implementation(find_key)};
172     }
173
174 private:
175     tree_position find_implementation(const Key& find_key) const
176     {
177         auto search_pointer = root_pointer;
178         for (int depth = 0; depth < stem_levels; ++depth)
179         {
180             const auto& search_stem = get_stem(search_pointer);
181             search_pointer =
182                 search_stem.elem(search_stem.find(find_key)).second;
183         }
184         const auto& leaf = get_leaf(search_pointer);
185         return {search_pointer, leaf.find(find_key)};
186     }
187
188 private:
189     using Path = std::vector<tree_position>;
190
191     Path find_path(const Key& find_key) const
192     {
193         Path result(stem_levels + 1);
194         auto search_pointer = root_pointer;
195         for (int depth = 0; depth < stem_levels; ++depth)
196         {
197             const auto& search_stem = get_stem(search_pointer);
198             result[depth].page = search_pointer;
199             result[depth].sub_position = search_stem.find(find_key);
200             search_pointer =
201                 search_stem.elem(result[depth].sub_position).second;
202         }
203         const auto& search_leaf = get_leaf(search_pointer);
204         result[stem_levels].page = search_pointer;
205         result[stem_levels].sub_position = search_leaf.find(find_key);
206         return result;
207     }
208
209     void split_root()
210     {
211         if (stem_levels > 0)
212         {
213             const auto old_root_pointer = root_pointer;
214             auto& old_root = get_stem(old_root_pointer);
215
216             if (old_root.oversized())
217             {
218                 const auto new_pointer = old_root.split_one_leaf();
219                 auto& new_stem = get_stem(new_pointer);
220
221                 while (old_root.oversized())
222                 {
223                     new_stem.borrow_prev(old_root);
224                 }
225             }

```

```

226         root_pointer = new stem_node;
227         auto& new_root = get_stem(root_pointer);
228
229         new_root.insert({old_root.min_key(), old_root_pointer});
230         new_root.insert({new_stem.min_key(), new_pointer});
231
232         ++stem_levels;
233     }
234 }
235 else
236 {
237     const auto old_root_pointer = root_pointer;
238     auto& old_root = get_leaf(old_root_pointer);
239
240     if (old_root.oversized())
241     {
242         const auto new_pointer = old_root.split_one_leaf();
243         auto& new_leaf = get_leaf(new_pointer);
244
245         while (old_root.oversized()) {
246             new_leaf.borrow_prev(old_root);
247         }
248
249         old_root.aux.next_pointer = new_pointer;
250         new_leaf.aux.prev_pointer = old_root_pointer;
251         max_leaf_pointer = new_pointer;
252
253         root_pointer = new stem_node;
254         auto& new_root = get_stem(root_pointer);
255
256         new_root.insert({old_root.min_key(), old_root_pointer});
257         new_root.insert({new_leaf.min_key(), new_pointer});
258
259         ++stem_levels;
260     }
261 }
262 }
263
264 public:
265     void insert(const Element& new_elem)
266     {
267         split_root();
268
269         const auto& new_key = key_extract_(new_elem);
270
271         auto current_pointer = root_pointer;
272         for (int depth = 0; depth < stem_levels - 1; ++depth)
273         {
274             auto& current_stem = get_stem(current_pointer);
275
276             const auto target_pos = current_stem.find(new_key);
277             const auto target_pointer = current_stem.elem(target_pos).second;
278             auto& target_stem = get_stem(target_pointer);
279
280             // Offload to previous sibling?
281             if (target_stem.oversized() &&
282                 target_pos != current_stem.min_position())
283             {
284                 const auto prev_pos = current_stem.prev_position(target_pos);
285                 const auto prev_pointer = current_stem.elem(prev_pos).second;
286                 auto& prev_stem = get_stem(prev_pointer);
287

```



```

288     if (prev_stem.small()) {
289         while (target_stem.oversized()) {
290             prev_stem.borrow_next(target_stem);
291         }
292
293         current_stem.set_key(target_pos, target_stem.min_key());
294
295         if (new_key < target_stem.min_key()) {
296             if (new_key < prev_stem.min_key()) {
297                 current_stem.set_key(prev_pos, new_key);
298             }
299             current_pointer = prev_pointer;
300         } else {
301             current_pointer = target_pointer;
302         }
303         continue;
304     }
305 }
306
307 // Offload to next sibling?
308 if (target_stem.oversized() &&
309     target_pos != current_stem.max_position())
310 {
311     const auto next_pos = current_stem.next_position(target_pos);
312     const auto next_pointer = current_stem.elem(next_pos).second;
313     auto& next_stem = get_stem(next_pointer);
314
315     if (next_stem.small()) {
316         while (target_stem.oversized()) {
317             next_stem.borrow_prev(target_stem);
318         }
319
320         current_stem.set_key(next_pos, next_stem.min_key());
321
322         if (new_key >= next_stem.min_key()) {
323             current_pointer = next_pointer;
324         } else {
325             if (new_key < target_stem.min_key()) {
326                 current_stem.set_key(target_pos, new_key);
327             }
328             current_pointer = target_pointer;
329         }
330         continue;
331     }
332 }
333
334 if (target_stem.oversized())
335 {
336     // Offload to new next sibling?
337     const auto new_pointer = target_stem.split_one_leaf();
338     auto& new_stem = get_stem(new_pointer);
339
340     while (target_stem.oversized()) {
341         new_stem.borrow_prev(target_stem);
342     }
343
344     current_stem.insert({new_stem.min_key(), new_pointer});
345
346     if (new_key >= new_stem.min_key()) {
347         current_pointer = new_pointer;
348     } else {
349         if (new_key < target_stem.min_key()) {

```

```

350         current_stem.set_key(target_pos, new_key);
351     }
352     current_pointer = target_pointer;
353 }
354     continue;
355 }
356
357     if (new_key < target_stem.min_key()) {
358         current_stem.set_key(target_pos, new_key);
359     }
360     current_pointer = target_pointer;
361 }
362
363 if (stem_levels > 0)
364 {
365     auto& current_stem = get_stem(current_pointer);
366
367     const auto target_pos = current_stem.find(new_key);
368     const auto target_pointer = current_stem.elem(target_pos).second;
369     auto& target_leaf = get_leaf(target_pointer);
370
371     // Offload to previous sibling?
372     if (target_leaf.oversized() &&
373         target_pos != current_stem.min_position())
374     {
375         const auto prev_pos = current_stem.prev_position(target_pos);
376         const auto prev_pointer = current_stem.elem(prev_pos).second;
377         auto& prev_leaf = get_leaf(prev_pointer);
378
379         if (prev_leaf.small()) {
380             while (target_leaf.oversized()) {
381                 prev_leaf.borrow_next(target_leaf);
382             }
383
384             current_stem.set_key(target_pos, target_leaf.min_key());
385
386             if (new_key < target_leaf.min_key()) {
387                 if (new_key < prev_leaf.min_key()) {
388                     current_stem.set_key(prev_pos, new_key);
389                 }
390                 prev_leaf.insert(new_elem);
391             } else {
392                 target_leaf.insert(new_elem);
393             }
394             return;
395         }
396     }
397
398     // Offload to next sibling?
399     if (target_leaf.oversized() &&
400         target_pos != current_stem.max_position())
401     {
402         const auto next_pos = current_stem.next_position(target_pos);
403         const auto next_pointer = current_stem.elem(next_pos).second;
404         auto& next_leaf = get_leaf(next_pointer);
405
406         if (next_leaf.small()) {
407             while (target_leaf.oversized()) {
408                 next_leaf.borrow_prev(target_leaf);
409             }
410
411             current_stem.set_key(next_pos, next_leaf.min_key());

```

```

412         if (new_key >= next_leaf.min_key()) {
413             next_leaf.insert(new_elem);
414         } else {
415             if (new_key < target_leaf.min_key()) {
416                 current_stem.set_key(target_pos, new_key);
417             }
418             target_leaf.insert(new_elem);
419         }
420     }
421     return;
422 }
423
424 // Offload to new next sibling?
425 if (target_leaf.oversized())
426 {
427     const auto new_pointer = target_leaf.split_one_leaf();
428     auto& new_leaf = get_leaf(new_pointer);
429
430     while (target_leaf.oversized()) {
431         new_leaf.borrow_prev(target_leaf);
432     }
433
434     current_stem.insert({new_leaf.min_key(), new_pointer});
435
436     if (target_leaf.aux.next_pointer != nullptr) {
437         get_leaf(target_leaf.aux.next_pointer).aux.prev_pointer =
438             new_pointer;
439     }
440     new_leaf.aux.prev_pointer = target_pointer;
441     new_leaf.aux.next_pointer = target_leaf.aux.next_pointer;
442     target_leaf.aux.next_pointer = new_pointer;
443
444     if (max_leaf_pointer == target_pointer) {
445         max_leaf_pointer = new_pointer;
446     }
447
448     if (new_key >= new_leaf.min_key()) {
449         new_leaf.insert(new_elem);
450     } else {
451         if (new_key < target_leaf.min_key()) {
452             current_stem.set_key(target_pos, new_key);
453         }
454         target_leaf.insert(new_elem);
455     }
456     return;
457 }
458
459 if (new_key < target_leaf.min_key()) {
460     current_stem.set_key(target_pos, new_key);
461 }
462 target_leaf.insert(new_elem);
463 }
464 else
465 {
466     auto& current_leaf = get_leaf(current_pointer);
467     current_leaf.insert(new_elem);
468 }
469 }
470
471 private:
472 void insert_node_after(const Path& path, const int depth,
473

```

```

474     const Key prev_min_key, void* const prev_pointer,
475     const Key new_min_key, void* const new_pointer)
476 {
477     if (depth > 0)
478     {
479         const auto insert_pointer = path[depth - 1].page;
480         auto& insert_stem = get_stem(insert_pointer);
481
482         const auto split_pointer =
483             insert_stem.insert(new_min_key, new_pointer);
484         if (split_pointer)
485         {
486             auto& split_stem = *split_pointer;
487             insert_node_after(path, depth - 1,
488                 insert_stem.min_key(), insert_pointer,
489                 split_stem.min_key(), split_pointer);
490         }
491     }
492     else
493     {
494         root_pointer = new stem_node;
495         auto& root = get_stem(root_pointer);
496
497         root.insert(prev_min_key, prev_pointer);
498         root.insert(new_min_key, new_pointer);
499
500         ++stem_levels;
501     }
502 }
503
504 public:
505 void erase(const Key& erase_key)
506 {
507     const auto path = find_path(erase_key);
508
509     const auto erase_pointer = path[stem_levels].page;
510     auto& erase_leaf = get_leaf(erase_pointer);
511
512     const auto was_large = erase_leaf.large();
513     // const auto old_key = erase_leaf.min_key();
514
515     // Erase the element
516     erase_leaf.erase(erase_key);
517     // If this is the root we are done
518     if (stem_levels == 0) { return; }
519
520     auto parent_pos = path[stem_levels - 1].sub_position;
521     auto& parent_stem = get_stem(path[stem_levels - 1].page);
522     const auto parent_was_large = parent_stem.large();
523     const auto old_key = parent_stem.key(parent_pos);
524
525     // If the node is now empty
526     if (erase_leaf.empty()) {
527         delete &erase_leaf;
528         parent_stem.erase(old_key);
529     } // Otherwise we must maintain the invariants
530     else {
531         const auto prev_ptr = parent_pos == parent_stem.min_position() ?
532             nullptr : static_cast<leaf_node*>(parent_stem.elem(
533                 parent_stem.prev_position(parent_pos)).second);
534         const auto prev_key = prev_ptr ? prev_ptr->min_key() : Key{};
535     }

```

```

536     const auto next_ptr = parent_pos == parent_stem.max_position() ?
537         nullptr : static_cast<leaf_node*>(parent_stem.elem(
538             parent_stem.next_position(parent_pos)).second);
539     const auto next_key = next_ptr ? next_ptr->min_key() : Key{};
540
541     bool did_grow = false;
542
543     // If a large node turned small we must grow it large again
544     if (was_large && erase_leaf.small()) {
545         if (prev_ptr && prev_ptr->small()) {
546             while (erase_leaf.small() && !prev_ptr->empty()) {
547                 erase_leaf.borrow_prev(*prev_ptr);
548             }
549         }
550
551         if (next_ptr && next_ptr->small()) {
552             while (erase_leaf.small() && !next_ptr->empty()) {
553                 erase_leaf.borrow_next(*next_ptr);
554             }
555         }
556
557         did_grow = true;
558     }
559
560     if (prev_ptr && prev_ptr->empty()) {
561         if (prev_ptr->aux.prev_pointer) {
562             get_leaf(prev_ptr->aux.prev_pointer).aux.next_pointer =
563                 erase_pointer;
564         }
565         erase_leaf.aux.prev_pointer = prev_ptr->aux.prev_pointer;
566
567         if (min_leaf_pointer == prev_ptr) {
568             min_leaf_pointer = &get_leaf(erase_pointer);
569         }
570
571         delete prev_ptr;
572         parent_stem.erase(prev_key);
573     }
574
575     if (next_ptr && next_ptr->empty()) {
576         if (next_ptr->aux.next_pointer) {
577             get_leaf(next_ptr->aux.next_pointer).aux.prev_pointer =
578                 erase_pointer;
579         }
580         erase_leaf.aux.next_pointer = next_ptr->aux.next_pointer;
581
582         if (max_leaf_pointer == next_ptr) {
583             max_leaf_pointer = &get_leaf(erase_pointer);
584         }
585
586         delete next_ptr;
587         parent_stem.erase(next_key);
588     }
589     else if (next_ptr && next_ptr->min_key() != next_key) {
590         parent_stem.set_key(
591             parent_stem.find(next_key), next_ptr->min_key());
592     }
593
594     if (erase_leaf.min_key() != old_key) {
595         parent_stem.set_key(
596             parent_stem.find(old_key), erase_leaf.min_key());
597     }

```

```

598     }
599
600     erase_helper(path, stem_levels - 1, parent_was_large);
601 }
602
603 private:
604     void root_collapse()
605     {
606         auto& root = get_stem(root_pointer);
607
608         // If we have only one child we should collapse this level of th tree
609         if (root.stem_levels == 0 &&
610             root.get_leaf(root.min_leaf_index).count() == 1)
611         {
612             const auto old_root = root_pointer;
613             root_pointer = root.elem({root.min_leaf_index, 0}).second;
614             delete get_stem_pointer(old_root);
615             --stem_levels;
616
617             if (stem_levels > 0) root_collapse();
618         }
619     }
620
621     void erase_helper(const Path& path, const int depth, bool was_large)
622     {
623         // If we are the root
624         if (depth == 0) {
625             root_collapse();
626             return;
627         }
628
629         const auto erase_pointer = path[depth].page;
630         auto& erase_stem = get_stem(erase_pointer);
631
632         // const auto old_key = erase_stem.min_key();
633
634         auto& parent_stem = get_stem(path[depth - 1].page);
635         auto parent_pos = path[depth - 1].sub_position;
636         const auto parent_was_large = parent_stem.large();
637         const auto old_key = parent_stem.key(parent_pos);
638
639         // If the node is now empty
640         if (erase_stem.empty()) {
641             delete &erase_stem;
642             parent_stem.erase(old_key);
643         // Otherwise we must maintain the invariants
644         } else {
645             const auto prev_ptr = parent_pos == parent_stem.min_position() ?
646                 nullptr : static_cast<stem_node*>(parent_stem.elem(
647                     parent_stem.prev_position(parent_pos)).second);
648             const auto prev_key = prev_ptr ? prev_ptr->min_key() : Key{};
649
650             const auto next_ptr = parent_pos == parent_stem.max_position() ?
651                 nullptr : static_cast<stem_node*>(parent_stem.elem(
652                     parent_stem.next_position(parent_pos)).second);
653             const auto next_key = next_ptr ? next_ptr->min_key() : Key{};
654
655             // If a large node turned small we must grow it large again
656             if (was_large && erase_stem.small()) {
657                 if (prev_ptr && prev_ptr->small()) {
658                     while (erase_stem.small() && !prev_ptr->empty()) {
659                         erase_stem.borrow_prev(*prev_ptr);

```

```

660         }
661     }
662
663     if (next_ptr && next_ptr->small()) {
664         while (erase_stem.small() && !next_ptr->empty()) {
665             erase_stem.borrow_next(*next_ptr);
666         }
667     }
668 }
669
670 if (prev_ptr && prev_ptr->empty()) {
671     delete prev_ptr;
672     parent_stem.erase(prev_key);
673 }
674
675 if (next_ptr && next_ptr->empty()) {
676     delete next_ptr;
677     parent_stem.erase(next_key);
678 }
679 else if (next_ptr && next_ptr->min_key() != next_key) {
680     parent_stem.set_key(
681         parent_stem.find(next_key), next_ptr->min_key());
682 }
683
684 if (erase_stem.min_key() != old_key) {
685     parent_stem.set_key(
686         parent_stem.find(old_key), erase_stem.min_key());
687 }
688 }
689
690 erase_helper(path, depth - 1, parent_was_large);
691 }
692
693 public:
694     // ITERATOR GETTERS
695
696     iterator begin() {
697         return make_iterator({
698             min_leaf_pointer,
699             min_leaf_pointer->min_position()});
700     }
701
702     const_iterator begin() const {
703         return make_const_iterator({
704             min_leaf_pointer,
705             min_leaf_pointer->min_position()});
706     }
707
708     const_iterator cbegin() const {
709         return make_const_iterator({
710             min_leaf_pointer,
711             min_leaf_pointer->min_position()});
712     }
713
714     iterator end() {
715         return make_iterator({
716             max_leaf_pointer,
717             max_leaf_pointer->end_position()});
718     }
719
720     const_iterator end() const {
721         return make_const_iterator({

```

```

722         max_leaf_pointer,
723         max_leaf_pointer->end_position());
724     }
725
726     const_iterator cend() const {
727         return make_const_iterator({
728             max_leaf_pointer,
729             max_leaf_pointer->end_position()});
730     }
731
732 private:
733     iterator make_iterator(tree_position position) {
734         return {this, position};
735     }
736
737     const_iterator make_const_iterator(tree_position position) const {
738         return {this, position};
739     }
740
741     // ITERATOR TYPE
742
743     template <typename T>
744     class iterator_template : std::iterator<std::forward_iterator_tag, T>
745     {
746         friend kernel;
747
748     public:
749         iterator_template()
750         : tree_(nullptr),
751           position_()
752         {}
753
754         iterator_template(const iterator_template& other)
755         : tree_(other.tree_),
756           position_(other.position_)
757         {}
758
759         iterator_template& operator=(const iterator_template& other)
760         {
761             tree_ = other.tree_;
762             position_ = other.position_;
763         }
764
765         ~iterator_template()
766         {}
767
768         reference operator*() {
769             return tree_->elem(position_);
770         }
771
772         const_reference operator*() const {
773             return tree_->elem(position_);
774         }
775
776         pointer operator->() {
777             return &tree_->elem(position_);
778         }
779
780         const_pointer operator->() const {
781             return &tree_->elem(position_);
782         }
783

```



```

784     iterator_template& operator++()
785     {
786         const auto& leaf = get_leaf(position_.page);
787         if (leaf.aux.next_pointer != nullptr &&
788             position_.sub_position == leaf.max_position())
789             {
790                 position_.page = leaf.aux.next_pointer;
791                 const auto& next = get_leaf(position_.page);
792                 position_.sub_position = next.min_position();
793             }
794         else
795             {
796                 position_.sub_position =
797                     leaf.next_position(position_.sub_position);
798             }
799         return *this;
800     }
801
802     iterator_template operator++(int)
803     {
804         iterator_template old(*this);
805         ++*this;
806         return old;
807     }
808
809     bool operator==(const iterator_template& other) const {
810         return tree_ == other.tree_ && position_ == other.position_;
811     }
812
813     bool operator!=(const iterator_template& other) const {
814         return tree_ != other.tree_ || position_ != other.position_;
815     }
816
817     private:
818         iterator_template(
819             kernel* tree,
820             tree_position position)
821         : tree_(tree),
822           position_(position)
823         {}
824
825         kernel* tree_;
826         tree_position position_;
827     };
828
829     public:
830         void print() const
831         {
832             std::cout << "-----" << std::endl;
833             print_node(root_pointer, 0);
834         }
835
836     private:
837         void print_node(void* pointer, int depth) const
838         {
839             if (depth < stem_levels) {
840                 const stem_node& stem = get_stem(pointer);
841                 std::cout << "treestem_" << depth << ",□"
842                     << (int)stem.get_leaf(stem.min_leaf_index).count()
843                     << "□" << std::endl;
844                 std::cout << "--" << std::endl;
845                 stem.print();

```

```

846         std::cout << "--" << std::endl;
847         std::cout << std::endl;
848         for (auto p = stem.min_position(); p != stem.end_position(); p =
849             stem.next_position(p))
850         {
851             print_node(stem.elem(p).second, depth + 1);
852         }
853     } else {
854         const leaf_node& leaf = get_leaf(pointer);
855         std::cout << "treeleaf□(" << depth << ")□" << std::endl;
856         std::cout << "--" << std::endl;
857         leaf.print();
858         std::cout << "--" << std::endl;
859         std::cout << std::endl;
860     }
861 }
862 };
863
864 } // namespace detail
865
866 // SET
867
868 #define BASE detail::kernel<\
869     Key,\
870     Key,\
871     Key,\
872     extract::identity,\
873     extract::identity>
874 template <
875     class Key>
876 class set : public BASE
877 {};
878 #undef BASE
879
880 // MAP
881
882 #define BASE detail::kernel<\
883     std::pair<const Key, T>,\
884     std::pair<Key, T>,\
885     Key,\
886     extract::first,\
887     extract::second>
888 template <
889     class Key,
890     class T>
891 class map : public BASE
892 {
893 public:
894     using mapped_type = T;
895 };
896 #undef BASE
897
898 } // namespace double_tree

```

A.2 double_tree_line_node.hpp

```

1 // An array based node that fits in a cache line
2
3 #pragma once
4 #include <array>
5 #include <cstdint>

```

```

6 #include <iostream>
7 #include <limits>
8
9 namespace double_tree
10 {
11
12 namespace detail
13 {
14
15 // Define here the size in bytes of line nodes, as well as the types used to
16 // index inside them. A line node index should be able to index all values in
17 // an array of key-value pairs of size equal to the line node. See below for
18 // the exact requirements, which are a little less because of the size used up
19 // for bookkeeping data in each case
20
21 constexpr size_t line_node_size = 256;
22 using line_index = uint8_t;
23 constexpr line_index line_index_nil = std::numeric_limits<line_index>::max();
24
25 template <
26     typename Element,
27     typename ElementWrite,
28     typename Key,
29     typename KeyExtract,
30     typename ValExtract,
31     typename Aux>
32 struct alignas(line_node_size) line_node
33 {
34 public:
35     // CONSTANTS
36
37     static constexpr int max_count =
38         (line_node_size - sizeof(line_index) - sizeof(Aux)) / sizeof(Element);
39     static constexpr int min_count = max_count/2;
40
41     // DATA
42
43 private:
44     std::array<Element, max_count> elems_;
45     line_index count_;
46     KeyExtract key_extract_;
47 public:
48     Aux aux;
49
50     // ACCESSORS
51
52 public:
53     line_index count() const { return count_; }
54
55     void reset() { count_ = 0; }
56
57     const Key& key(const line_index index) const {
58         return key_extract_(elems_[index]);
59     }
60
61     Element& elem(const line_index index) {
62         return elems_[index];
63     }
64
65     const Element& elem(const line_index index) const {
66         return elems_[index];
67     }

```

```

68
69 void set_key(const line_index index, const Key& new_key) {
70     key_extract_(elem_write(index)) = new_key;
71 }
72
73 void set_elem(const line_index index, const Element& new_element) {
74     elem_write(index) = new_element;
75 }
76
77 line_index min_index() const { return 0; }
78 line_index max_index() const { return (count_ == 0) ? 0 : count_ - 1; }
79 line_index end_index() const { return count_; }
80
81 const Key& min_key() const { return key(min_index()); }
82 const Element& min_elem() const { return elem(min_index()); }
83
84 const Key& end_key() const { return key(end_index()); }
85 const Element& end_elem() const { return elem(end_index()); }
86
87 private:
88     ElementWrite& elem_write(const line_index index) {
89         return reinterpret_cast<ElementWrite&>(elems_[index]);
90     }
91
92     ElementWrite& min_elem_write() {
93         return elem_write(min_index());
94     }
95
96     // PREDICATES
97
98 public:
99     bool empty() const { return count_ == 0; }
100
101     // Is node at maximum capacity?
102     bool full() const { return count_ == max_count; }
103
104     // Is node at minimum capacity?
105     bool thin() const { return count_ < min_count; }
106
107     // OPERATIONS
108
109     // Return the index of the greatest key less than or equal to the one
110     // given, or the minimum index if all keys are greater than the one given
111     line_index find(const Key& find_key) const
112     {
113         if (min_key() > find_key) {
114             return min_index();
115         }
116         for (int i = min_index() + 1; i < end_index(); ++i) {
117             if (key(i) > find_key) {
118                 return i - 1;
119             }
120         }
121         return count_ - 1;
122     }
123
124 private:
125     // Move all the elements from one index to another (inclusive) back one
126     // place
127     void move_one_back(const line_index begin, const line_index end)
128     {
129         std::move_backward(&elem(begin), &elem(end), &elem_write(end + 1));

```

```

130     }
131
132     // Move all the elements from one index to another (inclusive) forward one
133     // place
134     void move_one_forward(const line_index begin, const line_index end)
135     {
136         std::move(&elem(begin), &elem(end), &elem_write(begin - 1));
137     }
138
139     // Move all the elements from one index to another (inclusive) to another
140     // node, starting from some index in that node
141     void move_to(const line_index begin, const line_index end,
142                line_node& dest_node, const line_index dest)
143     {
144         std::move(&elem(begin), &elem(end), &dest_node.elem_write(dest));
145     }
146
147     public:
148     void init()
149     {
150         count_ = 0;
151     }
152
153     void init_from(const Element* begin, const Element* end)
154     {
155         std::move(begin, end, &min_elem_write());
156         count_ = end - begin;
157     }
158
159     void init_from(const line_node& other)
160     {
161         std::move(&other.min_elem(), &other.end_elem(), &min_elem_write());
162         count_ = other.count_;
163     }
164
165     // Insert a new element. Assumes the node is not full
166     void insert(const Element& new_elem)
167     {
168         line_index insert_index = end_index();
169         for (int i = min_index(); i < end_index(); ++i) {
170             if (key(i) > key_extract_(new_elem)) {
171                 insert_index = i;
172                 break;
173             }
174         }
175
176         move_one_back(insert_index, end_index());
177
178         set_elem(insert_index, new_elem);
179
180         ++count_;
181     }
182
183     // Split node in half
184     void split(line_node& split_node)
185     {
186         // This node takes half and the odd of the elements
187         const auto new_count = count_/2 + count_%2;
188
189         // The split node takes the the half
190         split_node.count_ = count_/2;
191

```

```

192     move_to(new_count, end_index(), split_node, split_node.min_index());
193
194     count_ = new_count;
195 }
196
197 // Erase an element. If node is thin this will put the node under capacity
198 void erase(const line_index erase_index)
199 {
200     if (erase_index < end_index()) {
201         move_one_forward(erase_index + 1, end_index());
202     }
203
204     --count_;
205 }
206
207 // Erase an element while merging node with the previous node. The
208 elements go into the previous node, so this one is left empty
209 void merge_prev_erase(const line_index erase_index, line_node& prev_node)
210 {
211     move_to(0, erase_index,
212            prev_node, prev_node.end_index());
213     move_to(erase_index + 1, end_index(),
214            prev_node, prev_node.end_index() + erase_index);
215
216     prev_node.count_ += count_ - 1;
217     count_ = 0;
218 }
219
220 // Erase an element while merging node with the next node. The elements go
221 into this node, so the next node is left empty
222 void merge_next_erase(const line_index erase_index, line_node& next_node)
223 {
224     move_one_forward(erase_index + 1, end_index());
225
226     next_node.move_to(
227         next_node.min_index(), next_node.end_index(),
228         *this, end_index() - 1);
229
230     count_ += next_node.count_ - 1;
231     next_node.count_ = 0;
232 }
233
234 // Erase an element while borrowing one from the previous node
235 void borrow_prev_erase(const line_index erase_index, line_node& prev_node)
236 {
237     move_one_back(min_index(), erase_index);
238     prev_node.move_to(
239         prev_node.end_index() - 1, prev_node.end_index(),
240         *this, min_index());
241
242     prev_node.count_ -= 1;
243 }
244
245 // Erase an element while borrowing one from the next node
246 void borrow_next_erase(const line_index erase_index, line_node& next_node)
247 {
248     move_one_forward(erase_index + 1, end_index());
249     next_node.move_to(
250         next_node.min_index(), next_node.min_index() + 1,
251         *this, end_index() - 1);
252
253     next_node.move_one_forward(

```

```

254         next_node.min_index() + 1, next_node.end_index());
255         next_node.count_ -= 1;
256     }
257
258     // Print all the keys in a comma-seperated list followed by a newline
259     void print() const
260     {
261         if (count_ > 0)
262         {
263             for (int index = 0; index < end_index(); ++index)
264             {
265                 std::cout << key(index);
266                 if (index < end_index() - 1)
267                 {
268                     std::cout << ",";
269                 }
270             }
271         }
272         std::cout << std::endl;
273     }
274 };
275
276 } // namespace detail
277
278 } // namespace double_tree

```

A.3 double_tree_page_node.hpp

```

1  // A tree based node that fits in a memory page
2
3  #pragma once
4  #include "double_tree_line_node.hpp"
5  #include "extract.hpp"
6  #include <array>
7  #include <cassert>
8  #include <cstdint>
9  #include <iostream>
10 #include <limits>
11
12 namespace double_tree
13 {
14
15     namespace detail
16     {
17
18         // Define here the size in bytes of, as well as the types used to index inside
19         // them. A page node index should be able to index all values in an array of
20         // line nodes of size equal to the page node. See below for the exact
21         // requirements, which are a little less because of the size used up for
22         // bookkeeping data in each case
23
24         constexpr size_t page_node_size = 4096;
25         using page_index = uint8_t;
26         constexpr page_index page_index_nil = std::numeric_limits<page_index>::max();
27
28         // An index into a page node combined with an index into the line node at that
29         // index constitutes a position of an element in the page node
30
31         struct page_position
32         {
33             page_position() = default;

```

```

34
35     page_position(const page_position& other):
36         line{other.line},
37         elem{other.elem}
38     {}
39
40     page_position(page_index line_, line_index elem_):
41         line{line_},
42         elem{elem_}
43     {}
44
45     bool operator==(const page_position& other) const
46     {
47         return line == other.line && elem == other.elem;
48     }
49
50     bool operator!=(const page_position& other) const
51     {
52         return line != other.line || elem != other.elem;
53     }
54
55     page_index line;
56     line_index elem;
57 };
58
59
60 template <
61     typename Element,
62     typename ElementWrite,
63     typename Key,
64     typename KeyExtract,
65     typename ValExtract,
66     typename Aux>
67 struct alignas(page_node_size) PageNode
68 {
69     // Auxiliary structures for the line nodes. A stem node does not need any
70     // extra data, while the leaf nodes of a page are linked together in a
71     // linked list, so they need the index of their previous and next nodes
72     struct stem_aux
73     {};
74
75     struct leaf_aux
76     {
77         page_index prev_index;
78         page_index next_index;
79     };
80
81     // The actual node types can now be defined
82     using stem_node = line_node<
83         std::pair<const Key, page_index>,
84         std::pair<Key, page_index>,
85         Key,
86         extract::first,
87         extract::second,
88         stem_aux>;
89     using leaf_node = line_node<
90         Element,
91         ElementWrite,
92         Key,
93         KeyExtract,
94         ValExtract,
95         leaf_aux>;

```



```

96
97 // MEMORY SYSTEM
98
99 // An entry in the pool memory. Each can hold either a stem node, a leaf
100 // node or a page index. The page index is only used within the memory
101 // system
102 struct alignas(line_node_size) PoolEntry
103 {
104     PoolEntry() {}
105
106     union {
107         stem_node stem;
108         leaf_node leaf;
109         page_index prev_head;
110     };
111 };
112
113 // Calculate how many entries the pool memory can hold
114 static constexpr int pool_size =
115     page_node_size - 6*sizeof(page_index) - 1 - sizeof(Aux);
116 static constexpr int pool_count = pool_size/sizeof(PoolEntry);
117
118 // The pool memory. The back index points at the highest entry that has
119 // never been allocated. The head index points at the next entry to
120 // allocate, and is either equal to the back index or lower. If it is
121 // equal the next head index is found by incrementing. If it is lower it
122 // points to an entry that has been deallocated, and when that was done the
123 // previous head index was stored there, and we restore it.
124 page_index allocate() {
125     assert(free_count > 0);
126     --free_count;
127     page_index allocate_index = head_index;
128     if (head_index == back_index) { head_index = ++back_index; }
129     else { head_index = pool_memory[head_index].prev_head; }
130     return allocate_index;
131 }
132
133 void deallocate(page_index deallocate_index) {
134     ++free_count;
135     pool_memory[deallocate_index].prev_head = head_index;
136     head_index = deallocate_index;
137 }
138
139 // SMALL, LARGE, OVERSIZED NODES
140
141 static constexpr int branchout = stem_node::max_count;
142
143 // Here n is the pool entries left and b is the total branchout of the
144 // previous level. By subtracting b from n and multiplying b by the
145 // branchout until we cover the rest of the nodes we figure out how many
146 // levels of stem nodes we will maximally need
147 static constexpr int max_stem_levels_helper(int n, int b) {
148     return (n > b) ? 1 + max_stem_levels_helper(n - b, b*branchout) : 0;
149 }
150 static constexpr int max_stem_levels = max_stem_levels_helper(pool_count,
151     1);
152 static constexpr int max_levels = max_stem_levels + 1;
153
154 bool small() const {
155     return free_count > 2*max_levels - 1;
156 }

```

```

157     bool large() const {
158         return free_count <= 2*max_levels - 1;
159     }
160
161     bool oversized() const {
162         return free_count <= max_levels - 1;
163     }
164
165     // DATA
166
167     std::array<PoolEntry, pool_count> pool_memory;
168     page_index head_index;
169     page_index back_index;
170     page_index free_count;
171
172     page_index root_index;
173     page_index min_leaf_index;
174     page_index max_leaf_index;
175     uint8_t stem_levels;
176     KeyExtract key_extract_;
177     Aux aux;
178
179     // CONSTRUCTOR
180
181     PageNode()
182     : head_index{0},
183       back_index{0},
184       free_count{pool_count},
185       root_index{allocate()},
186       min_leaf_index{root_index},
187       max_leaf_index{root_index},
188       stem_levels{0}
189     {
190         auto& root = get_leaf(root_index);
191         root.init();
192         root.aux.prev_index = page_index_nil;
193         root.aux.next_index = page_index_nil;
194     }
195
196     // ACCESSORS
197
198     stem_node& get_stem(const page_index index) {
199         return pool_memory[index].stem;
200     }
201     leaf_node& get_leaf(const page_index index) {
202         return pool_memory[index].leaf;
203     }
204
205     const stem_node& get_stem(const page_index index) const {
206         return pool_memory[index].stem;
207     }
208     const leaf_node& get_leaf(const page_index index) const {
209         return pool_memory[index].leaf;
210     }
211
212     const Key& key(const page_position position) const {
213         return get_leaf(position.line).key(position.elem);
214     }
215     Element& elem(const page_position position) {
216         return get_leaf(position.line).elem(position.elem);
217     }
218     const Element& elem(const page_position position) const {

```

```

219     return get_leaf(position.line).elem(position.elem);
220 }
221
222 void set_key(const page_position position, const Key& new_key) {
223     const Key old_key = get_leaf(position.line).key(position.elem);
224     get_leaf(position.line).set_key(position.elem, new_key);
225     if (position.elem == 0) {
226         const auto path = find_path(old_key);
227         update_key(
228             path, stem_levels - 1, path[stem_levels - 1].elem, new_key);
229     }
230 }
231
232 page_position min_position() const {
233     return {min_leaf_index, get_leaf(min_leaf_index).min_index()};
234 }
235 page_position max_position() const {
236     return {max_leaf_index, get_leaf(max_leaf_index).max_index()};
237 }
238 page_position end_position() const {
239     return {max_leaf_index,
240         (line_index)(get_leaf(max_leaf_index).max_index() + 1)};
241 }
242
243 page_position prev_position(const page_position position) const {
244     const leaf_node& node = get_leaf(position.line);
245     if (node.aux.prev_index != page_index_nil &&
246         position.elem == node.min_index())
247     {
248         return {node.aux.prev_index,
249             get_leaf(node.aux.prev_index).max_index()};
250     }
251     else
252     {
253         return {position.line, (line_index)(position.elem - 1)};
254     }
255 }
256
257 page_position next_position(const page_position position) const {
258     const leaf_node& node = get_leaf(position.line);
259     if (node.aux.next_index != page_index_nil &&
260         position.elem == node.max_index())
261     {
262         return {node.aux.next_index,
263             get_leaf(node.aux.next_index).min_index()};
264     }
265     else
266     {
267         return {position.line, (line_index)(position.elem + 1)};
268     }
269 }
270
271 const Key& min_key() const {
272     return get_leaf(min_leaf_index).min_key();
273 }
274
275 const Element& min_elem() const {
276     return get_leaf(min_leaf_index).min_elem();
277 }
278
279 const Key& max_key() const {
280     return get_leaf(max_leaf_index).max_key();

```

```

281     }
282
283     const Element& max_elem() const {
284         return get_leaf(max_leaf_index).max_elem();
285     }
286
287     // PREDICATES
288
289     bool empty() const {
290         return stem_levels == 0 && get_leaf(root_index).empty();
291     }
292
293     // OPERATIONS
294
295     // Returns the position of the greatest key less than or equal to the one
296     // given, or the minimum position if all keys are greater than the one
297     // given
298     page_position find(const Key& find_key) const
299     {
300         auto search_index = root_index;
301         for (int depth = 0; depth < stem_levels; ++depth)
302         {
303             const auto& search_stem = get_stem(search_index);
304             search_index = search_stem.elem(search_stem.find(find_key)).second;
305         }
306         const auto& search_leaf = get_leaf(search_index);
307         return {search_index, search_leaf.find(find_key)};
308     }
309
310 private:
311     // This structure is used to record a path down the tree to some specific
312     // element
313     using Path = std::array<page_position, max_levels>;
314
315     // Construct the path taken to find the key given
316     Path find_path(const Key& find_key) const
317     {
318         Path result;
319         auto search_index = root_index;
320         for (int depth = 0; depth < stem_levels; ++depth)
321         {
322             const auto& search_stem = get_stem(search_index);
323             result[depth].line = search_index;
324             result[depth].elem = search_stem.find(find_key);
325             search_index = search_stem.elem(result[depth].elem).second;
326         }
327         const auto& search_leaf = get_leaf(search_index);
328         result[stem_levels].line = search_index;
329         result[stem_levels].elem = search_leaf.find(find_key);
330         return result;
331     }
332
333     // Construct the leftmost path down the stem
334     Path min_path() const
335     {
336         // Record the leftmost path down the stem.
337         Path result;
338         auto search_index = root_index;
339         for (int depth = 0; depth < stem_levels; ++depth)
340         {
341             const auto& search_stem = get_stem(search_index);
342             result[depth].line = search_index;

```

```

343         result[depth].elem = search_stem.min_index();
344         search_index = search_stem.elem(result[depth].elem).second;
345     }
346     const auto& search_leaf = get_leaf(search_index);
347     result[stem_levels].line = search_index;
348     result[stem_levels].elem = search_leaf.min_index();
349     return result;
350 }
351
352 // Construct the rightmost path down the stem
353 Path max_path() const
354 {
355     // Record the leftmost path down the stem.
356     Path result;
357     auto search_index = root_index;
358     for (int depth = 0; depth < stem_levels; ++depth)
359     {
360         const auto& search_stem = get_stem(search_index);
361         result[depth].line = search_index;
362         result[depth].elem = search_stem.max_index();
363         search_index = search_stem.elem(result[depth].elem).second;
364     }
365     const auto& search_leaf = get_leaf(search_index);
366     result[stem_levels].line = search_index;
367     result[stem_levels].elem = search_leaf.max_index();
368     return result;
369 }
370
371 void split_root()
372 {
373     if (stem_levels > 0)
374     {
375         const auto old_root_index = root_index;
376         auto& old_root = get_stem(old_root_index);
377
378         if (old_root.full())
379         {
380             const auto new_index = allocate();
381             auto& new_stem = get_stem(new_index);
382             old_root.split(new_stem);
383
384             root_index = allocate();
385             auto& new_root = get_stem(root_index);
386             new_root.init();
387             new_root.insert({old_root.min_key(), old_root_index});
388             new_root.insert({new_stem.min_key(), new_index});
389
390             ++stem_levels;
391         }
392     }
393     else
394     {
395         const auto old_root_index = root_index;
396         auto& old_root = get_leaf(old_root_index);
397
398         if (old_root.full())
399         {
400             const auto new_index = allocate();
401             auto& new_leaf = get_leaf(new_index);
402             old_root.split(new_leaf);
403
404             old_root.aux.next_index = new_index;

```

```

405         new_leaf.aux.prev_index = old_root_index;
406         new_leaf.aux.next_index = page_index_nil;
407         max_leaf_index = new_index;
408
409         root_index = allocate();
410         auto& new_root = get_stem(root_index);
411         new_root.init();
412         new_root.insert({old_root.min_key(), old_root_index});
413         new_root.insert({new_leaf.min_key(), new_index});
414
415         ++stem_levels;
416     }
417 }
418 }
419
420 public:
421     // Insert a new element
422     void insert(const Element& new_elem)
423     {
424         split_root();
425
426         const auto& new_key = key_extract_(new_elem);
427
428         if (stem_levels > 0)
429         {
430             auto current_index = root_index;
431             for (int depth = 0; depth < stem_levels - 1; ++depth)
432             {
433                 auto& current_stem = get_stem(current_index);
434
435                 const auto target_pos = current_stem.find(new_key);
436                 const auto target_index = current_stem.elem(target_pos).second;
437                 auto& target_stem = get_stem(target_index);
438
439                 if (new_key < target_stem.min_key()) {
440                     current_stem.set_key(target_pos, new_key);
441                 }
442
443                 // Split target stem?
444                 if (target_stem.full()) {
445                     const auto new_index = allocate();
446                     auto& new_stem = get_stem(new_index);
447                     target_stem.split(new_stem);
448
449                     current_stem.insert({new_stem.min_key(), new_index});
450
451                     if (new_key >= new_stem.min_key()) {
452                         current_index = new_index;
453                     } else {
454                         current_index = target_index;
455                     }
456                 } else {
457                     current_index = target_index;
458                 }
459             }
460
461             auto& current_stem = get_stem(current_index);
462
463             const auto target_pos = current_stem.find(new_key);
464             const auto target_index = current_stem.elem(target_pos).second;
465             auto& target_leaf = get_leaf(target_index);
466

```

```

467         if (new_key < target_leaf.min_key()) {
468             current_stem.set_key(target_pos, new_key);
469         }
470
471         // Split target leaf?
472         if (target_leaf.full()) {
473             const auto new_index = allocate();
474             auto& new_leaf = get_leaf(new_index);
475             target_leaf.split(new_leaf);
476
477             current_stem.insert({new_leaf.min_key(), new_index});
478
479             // Adjust the linked leaf list to fit in the new leaf
480             if (target_leaf.aux.next_index != page_index_nil) {
481                 get_leaf(target_leaf.aux.next_index).aux.prev_index =
482                     new_index;
483             }
484             new_leaf.aux.next_index = target_leaf.aux.next_index;
485             new_leaf.aux.prev_index = target_index;
486             target_leaf.aux.next_index = new_index;
487
488             // If the target leaf was the max leaf, the new max is the new
489             // leaf
490             if (max_leaf_index == target_index) {
491                 max_leaf_index = new_index;
492             }
493
494             if (new_key >= new_leaf.min_key()) {
495                 new_leaf.insert(new_elem);
496             } else {
497                 target_leaf.insert(new_elem);
498             }
499         } else {
500             target_leaf.insert(new_elem);
501         }
502     }
503     else
504     {
505         auto& current_leaf = get_leaf(root_index);
506         current_leaf.insert(new_elem);
507     }
508 }
509
510 void insert_min_leaf(const Key new_min_key, const page_index new_index)
511 {
512     if (stem_levels > 0)
513     {
514         split_root();
515
516         auto current_index = root_index;
517         for (int depth = 0; depth < stem_levels - 1; ++depth)
518         {
519             auto& current_stem = get_stem(current_index);
520             const auto target_pos = current_stem.min_index();
521             const auto target_index = current_stem.elem(target_pos).second;
522             auto& target_stem = get_stem(target_index);
523             current_stem.set_key(target_pos, new_min_key);
524
525             // Split target stem?
526             if (target_stem.full()) {
527                 const auto new_index = allocate();
528                 auto& new_stem = get_stem(new_index);

```

```

529         target_stem.split(new_stem);
530         current_stem.insert({new_stem.min_key(), new_index});
531     }
532
533     current_index = target_index;
534 }
535
536     auto& current_stem = get_stem(current_index);
537     current_stem.insert({new_min_key, new_index});
538 }
539 else
540 {
541     const auto old_root_index = root_index;
542     auto& old_root = get_leaf(old_root_index);
543
544     root_index = allocate();
545     auto& new_root = get_stem(root_index);
546     new_root.init();
547     new_root.insert({new_min_key, new_index});
548     new_root.insert({old_root.min_key(), old_root_index});
549
550     ++stem_levels;
551 }
552 }
553
554 void insert_max_leaf(const Key new_min_key, const page_index new_index)
555 {
556     if (stem_levels > 0)
557     {
558         split_root();
559
560         auto current_index = root_index;
561         for (int depth = 0; depth < stem_levels - 1; ++depth)
562         {
563             auto& current_stem = get_stem(current_index);
564             const auto target_pos = current_stem.max_index();
565             const auto target_index = current_stem.elem(target_pos).second;
566             auto& target_stem = get_stem(target_index);
567
568             // Split target stem?
569             if (target_stem.full()) {
570                 const auto new_index = allocate();
571                 auto& new_stem = get_stem(new_index);
572                 target_stem.split(new_stem);
573                 current_stem.insert({new_stem.min_key(), new_index});
574                 current_index = new_index;
575             } else {
576                 current_index = target_index;
577             }
578         }
579
580         auto& current_stem = get_stem(current_index);
581         current_stem.insert({new_min_key, new_index});
582     }
583     else
584     {
585         const auto old_root_index = root_index;
586         auto& old_root = get_leaf(old_root_index);
587
588         root_index = allocate();
589         auto& new_root = get_stem(root_index);
590         new_root.init();

```



```

591         new_root.insert({old_root.min_key(), old_root_index});
592         new_root.insert({new_min_key, new_index});
593
594         ++stem_levels;
595     }
596 }
597
598 // Erase an element. The page might be left thin in which case the caller
599 // should decide what to do about that
600 void erase(const Key& erase_key)
601 {
602     const auto path = find_path(erase_key);
603
604     const auto line = path[stem_levels].line;
605     const auto elem = path[stem_levels].elem;
606     auto& erase_leaf = get_leaf(line);
607
608     // If the node is not root and thin, we must either merge or borrow
609     if (stem_levels > 0 && erase_leaf.thin())
610     {
611         // The parent node, and index of the node in the parent
612         const auto parent_line_index = path[stem_levels - 1].elem;
613
614         // Do we have a previous sibling?
615         if (erase_leaf.aux.prev_index != page_index_nil)
616         {
617             const auto prev_index = erase_leaf.aux.prev_index;
618             auto& prev_leaf = get_leaf(prev_index);
619
620             // Can we merge?
621             if (erase_leaf.count() + prev_leaf.count()
622                 <= leaf_node::max_count)
623             {
624                 // Merge the leaf with the element into its previous
625                 // sibling
626                 erase_leaf.merge_prev_erase(elem, prev_leaf);
627
628                 // Adjust the linked leaf list to erase the merged node
629                 if (erase_leaf.aux.next_index != page_index_nil) {
630                     get_leaf(erase_leaf.aux.next_index).aux.prev_index =
631                         prev_index;
632                 }
633                 prev_leaf.aux.next_index = erase_leaf.aux.next_index;
634
635                 // If the erased node was the max leaf, the new max is the
636                 // previous sibling
637                 if (max_leaf_index == line) {
638                     max_leaf_index = prev_index;
639                 }
640
641                 deallocate(line);
642
643                 // Now we must erase the erased node from the stem
644                 // structure
645                 erase_node(path, stem_levels - 1, parent_line_index);
646             }
647             // If we can not merge, borrow
648             else
649             {
650                 erase_leaf.borrow_prev_erase(elem, prev_leaf);
651
652                 // Since the node with the element has a new minimum

```

```

653         // element we must update the representative keys up the
654         // tree
655         update_key(path, stem_levels - 1,
656                 parent_line_index, erase_leaf.min_key());
657     }
658 }
659 // If we do not have a previous sibling, we should have a next
660 else
661 {
662     const auto next_index = erase_leaf.aux.next_index;
663     auto& next_leaf = get_leaf(next_index);
664
665     // Can we merge?
666     if (erase_leaf.count() + next_leaf.count()
667         <= leaf_node::max_count)
668     {
669         // Merge the next sibling into the leaf with the element
670         erase_leaf.merge_next_erase(elem, next_leaf);
671
672         // Adjust the linked leaf list to erase the merged node
673         if (next_leaf.aux.next_index != page_index_nil) {
674             get_leaf(next_leaf.aux.next_index).aux.prev_index =
675                 line;
676         }
677         erase_leaf.aux.next_index = next_leaf.aux.next_index;
678
679         // If the erased node was the max leaf, the new max is the
680         // leaf with the element in it
681         if (max_leaf_index == next_index) {
682             max_leaf_index = line;
683         }
684
685         deallocate(next_index);
686
687         // If we erased the minimum element we must update the
688         // representative keys up the tree
689         if (elem == 0) {
690             update_key(path, stem_levels - 1,
691                     parent_line_index, erase_leaf.min_key());
692         }
693
694         // Now we must erase the erased node from its parent
695         erase_node(path, stem_levels - 1, parent_line_index + 1);
696     }
697 // If we can not merge, borrow
698 else
699 {
700     erase_leaf.borrow_next_erase(elem, next_leaf);
701
702     // Since the next leaf has a new minimum we must update the
703     // representative keys up the tree
704     update_key(path, stem_levels - 1,
705             parent_line_index + 1, next_leaf.min_key());
706
707     // If we erased the minimum element we must update the
708     // representative keys up the tree
709     if (elem == 0) {
710         update_key(path, stem_levels - 1,
711                 parent_line_index, erase_leaf.min_key());
712     }
713 }
714 }

```

```

715     }
716     // Either we are the root or the stem to erase from is not thin
717     else
718     {
719         erase_leaf.erase(elem);
720
721         // If we are not the root and we erased the minimum element, we
722         // must update the representative keys up the tree
723         if (stem_levels > 0 && elem == 0)
724         {
725             const auto parent_line_index = path[stem_levels - 1].elem;
726             update_key(path, stem_levels - 1,
727                       parent_line_index, erase_leaf.min_key());
728         }
729     }
730 }
731
732 // Erase a node from the stem structure. The depth given should be the
733 // depth of the node to erase from
734 void erase_node(const Path& path, const int depth, const line_index elem)
735 {
736     const auto line = path[depth].line;
737     auto& erase_stem = get_stem(line);
738
739     // If the node is not root and thin, we must either merge or borrow
740     if (depth > 0 && erase_stem.thin())
741     {
742         // The parent node, and index of the node in the parent
743         const auto parent_page_index = path[depth - 1].line;
744         const auto parent_line_index = path[depth - 1].elem;
745         const auto& parent = get_stem(parent_page_index);
746
747         // Do we have a previous sibling?
748         if (parent_line_index > 0)
749         {
750             const auto prev_index =
751                 parent.elem(parent_line_index - 1).second;
752             auto& prev_stem = get_stem(prev_index);
753
754             // Can we merge?
755             if (erase_stem.count() + prev_stem.count()
756                 <= stem_node::max_count)
757             {
758                 // Merge the stem with the node into its previous sibling
759                 erase_stem.merge_prev_erase(elem, prev_stem);
760
761                 deallocate(line);
762
763                 // Now we must erase the erased node from the stem
764                 // structure
765                 erase_node(path, depth - 1, parent_line_index);
766             }
767             // If we can not merge, borrow
768             else
769             {
770                 erase_stem.borrow_prev_erase(elem, prev_stem);
771
772                 // Since the stem with the node has a new minimum element
773                 // we must update the representative keys up the tree
774                 update_key(path, depth - 1,
775                             parent_line_index, erase_stem.min_key());
776             }

```

```

777     }
778     // If we do not have a previous sibling, we should have a next
779     else
780     {
781         const auto next_index =
782             parent.elem(parent_line_index + 1).second;
783         auto& next_leaf = get_stem(next_index);
784
785         // Can we merge?
786         if (erase_stem.count() + next_leaf.count()
787             <= stem_node::max_count)
788         {
789             // Merge the next sibling into the stem with the node
790             erase_stem.merge_next_erase(elem, next_leaf);
791
792             deallocate(next_index);
793
794             // If we erased the minimum element we must update the
795             // representative keys up the tree
796             if (elem == 0) {
797                 update_key(path, depth - 1,
798                     parent_line_index, erase_stem.min_key());
799             }
800
801             // Now we must erase the erased node from its parent
802             erase_node(path, depth - 1, parent_line_index + 1);
803         }
804         // If we can not merge, borrow
805         else
806         {
807             erase_stem.borrow_next_erase(elem, next_leaf);
808
809             // Since the next stem has a new minimum we must update the
810             // representative keys up the tree
811             update_key(path, depth - 1,
812                 parent_line_index + 1, next_leaf.min_key());
813
814             // If we erased the minimum element we must update the
815             // representative keys up the tree
816             if (elem == 0) {
817                 update_key(path, depth - 1,
818                     parent_line_index, erase_stem.min_key());
819             }
820         }
821     }
822 }
823 // Either we are the root or the stem to erase from is not thin
824 else
825 {
826     erase_stem.erase(elem);
827
828     // If we are not the root and we erased the minimum element, we
829     // must update the representative keys up the tree
830     if (depth > 0 && elem == 0) {
831         const auto parent_line_index = path[depth - 1].elem;
832         update_key(path, depth - 1,
833             parent_line_index, erase_stem.min_key());
834     }
835
836     // If we are the root and we now have only one child we should
837     // collapse this level of the tree
838     if (depth == 0 && erase_stem.count() == 1) {

```

```

839         root_index = erase_stem.min_elem().second;
840         deallocate(line);
841         --stem_levels;
842     }
843 }
844 }
845
846 void update_key(const Path& path, const int depth,
847               const line_index elem, const Key& new_key)
848 {
849     auto& stem = get_stem(path[depth].line);
850     stem.set_key(elem, new_key);
851     if (depth > 0 && elem == 0)
852     {
853         update_key(path, depth - 1, path[depth - 1].elem, new_key);
854     }
855 }
856
857 void borrow_prev(PageNode& prev_page)
858 {
859     const auto prev_path = prev_page.max_path();
860     const auto old_index = prev_path[prev_page.stem_levels].line;
861     auto& old_leaf = prev_page.get_leaf(old_index);
862
863     if (old_leaf.count() < leaf_node::min_count) {
864         // TODO: This could be done more efficiently
865         for (int i = 0; i < old_leaf.count(); ++i) {
866             insert(old_leaf.elem(i));
867         }
868     } else {
869         const auto this_path = min_path();
870
871         // Copy the leaf
872         auto new_index = allocate();
873         auto& new_leaf = get_leaf(new_index);
874
875         const auto next_index = this_path[stem_levels].line;
876         auto& next_leaf = get_leaf(next_index);
877         next_leaf.aux.prev_index = new_index;
878         new_leaf.aux.prev_index = page_index_nil;
879         new_leaf.aux.next_index = next_index;
880
881         new_leaf.init_from(old_leaf);
882
883         min_leaf_index = new_index;
884
885         // Insert into this pages stem structure
886         insert_min_leaf(new_leaf.min_key(), new_index);
887     }
888
889     // Erase the node from the other pages stem structure
890     if (prev_page.stem_levels != 0) {
891         prev_page.max_leaf_index = old_leaf.aux.prev_index;
892         prev_page.get_leaf(prev_page.max_leaf_index).aux.next_index =
893             page_index_nil;
894         prev_page.deallocate(old_index);
895         prev_page.erase_node(prev_path, prev_page.stem_levels - 1,
896                             prev_path[prev_page.stem_levels - 1].elem);
897     } else {
898         old_leaf.reset();
899     }
900 }

```

```

901
902 void borrow_next(PageNode& next_page)
903 {
904     const auto next_path = next_page.min_path();
905     const auto old_index = next_path[next_page.stem_levels].line;
906     auto& old_leaf = next_page.get_leaf(old_index);
907
908     if (old_leaf.count() < leaf_node::min_count) {
909         // TODO: This could be done more efficiently
910         for (int i = 0; i < old_leaf.count(); ++i) {
911             insert(old_leaf.elem(i));
912         }
913     } else {
914         const auto this_path = max_path();
915
916         // Copy the leaf
917         const auto new_index = allocate();
918         auto& new_leaf = get_leaf(new_index);
919
920         const auto prev_index = this_path[stem_levels].line;
921         auto& prev_leaf = get_leaf(prev_index);
922         prev_leaf.aux.next_index = new_index;
923         new_leaf.aux.prev_index = prev_index;
924         new_leaf.aux.next_index = page_index_nil;
925
926         new_leaf.init_from(old_leaf);
927
928         max_leaf_index = new_index;
929
930         // Update the keys on the path
931         auto new_key = next_page.min_key();
932         for (int depth = 0; depth < next_page.stem_levels - 1; ++depth)
933         {
934             auto& stem = next_page.get_stem(next_path[depth].line);
935             stem.set_key(next_path[depth].elem, new_key);
936         }
937
938         // Insert into this pages stem structure
939         insert_max_leaf(new_leaf.min_key(), new_index);
940     }
941
942     // Erase the node from the other pages stem structure
943     if (next_page.stem_levels != 0) {
944         next_page.min_leaf_index = old_leaf.aux.next_index;
945         next_page.get_leaf(next_page.min_leaf_index).aux.prev_index =
946             page_index_nil;
947         next_page.deallocate(old_index);
948         next_page.erase_node(next_path, next_page.stem_levels - 1,
949             next_path[stem_levels - 1].elem);
950     } else {
951         old_leaf.reset();
952     }
953 }
954
955 PageNode* split_one_leaf()
956 {
957     const auto this_path = max_path();
958
959     // Copy the leaf
960     const auto new_page_pointer = new PageNode;
961     auto& new_page = *new_page_pointer;
962     auto new_index = new_page.root_index;

```

```

963     auto& new_leaf = new_page.get_leaf(new_index);
964
965     const auto old_index = this_path[stem_levels].line;
966     auto& old_leaf = get_leaf(old_index);
967     new_leaf.init_from(old_leaf);
968
969     // Erase the node from this pages stem structure
970     if (stem_levels != 0) {
971         max_leaf_index = old_leaf.aux.prev_index;
972         get_leaf(max_leaf_index).aux.next_index = page_index_nil;
973         deallocate(old_index);
974         erase_node(this_path, stem_levels - 1,
975                 this_path[stem_levels - 1].elem);
976     } else {
977         old_leaf.reset();
978     }
979
980     return new_page_pointer;
981 }
982
983 void print() const
984 {
985     print_node(root_index, 0);
986 }
987
988 void print_tabs(int num) const
989 {
990     for (int i = 0; i < num; ++i)
991     {
992         std::cout << "  ";
993     }
994 }
995
996 void print_node(page_index line, int depth) const
997 {
998     if (depth < stem_levels)
999     {
1000         const stem_node& stem = get_stem(line);
1001         print_tabs(depth);
1002         std::cout << "stem_" << depth << " ";
1003         stem.print();
1004         for (int i = 0; i < stem.count(); ++i) {
1005             print_node(stem.elem(i).second, depth + 1);
1006         }
1007     }
1008     else
1009     {
1010         const leaf_node& leaf = get_leaf(line);
1011         print_tabs(depth);
1012         std::cout << "leaf_" << depth << " ";
1013         leaf.print();
1014     }
1015 }
1016 };
1017
1018 } // namespace detail
1019
1020 } // namespace double_tree

```

A.4 etract.hpp

```

1 #pragma once
2
3 namespace extract
4 {
5
6 // Used as KeyExtract for the sets.
7 struct identity {
8     template<typename U>
9     constexpr auto operator()(U&& v) const -> decltype(std::forward<U>(v)) {
10         return std::forward<U>(v);
11     }
12 };
13
14 // Used as KeyExtract for the maps.
15 struct first {
16     template<typename U>
17     constexpr auto operator()(U&& v) const -> decltype(std::get<0>(v)) {
18         return std::get<0>(v);
19     }
20 };
21
22 // Used as MappedExtract for the maps.
23 struct second {
24     template<typename U>
25     constexpr auto operator()(U&& v) const -> decltype(std::get<1>(v)) {
26         return std::get<1>(v);
27     }
28 };
29
30 }

```

A.5 hopscotch.hpp

Note that this code was first produced for another project. Some adjustments and fixes were made for this project.

```

1 #pragma once
2
3 #include <algorithm>
4 #include <bitset>
5 #include <cassert>
6 #include <cmath>
7 #include <cstddef>
8 #include <cstring>
9 #include <functional>
10 #include <initializer_list>
11 #include <iterator>
12 #include <memory>
13 #include <utility>
14 #include <vector>
15
16 #include <iostream>
17
18 #include "extract.hpp"
19
20 namespace hopscotch
21 {
22     namespace detail
23     {
24         // KERNEL
25

```



```

26 template <
27     class Value,
28     class Key,
29     class Hash,
30     class KeyExtract,
31     class MappedExtract,
32     class KeyEqual,
33     class Allocator>
34 class kernel
35 {
36 protected:
37     // I have not seen much variation in performance from changing this, so it
38     // is maxed out for flexibility. It is one from 64 because we use the same
39     // bitset to store the hop info and whether a bucket has a value or not.
40     static const size_t neighborhood_size = 63;
41
42     // Defined below.
43     template <typename T> class iterator_template;
44
45 public:
46     // MEMBER TYPES
47
48     using key_type      = Key;
49     using value_type    = Value;
50     using size_type     = std::size_t;
51     using difference_type = std::ptrdiff_t;
52     using hasher        = Hash;
53     using key_equal     = KeyEqual;
54     using key_extract   = KeyExtract;
55     using mapped_extract = MappedExtract;
56     using allocator_type = Allocator;
57     using reference     = value_type&;
58     using const_reference = const value_type&;
59     using pointer       =
60         typename std::allocator_traits<Allocator>::pointer;
61     using const_pointer =
62         typename std::allocator_traits<Allocator>::const_pointer;
63     using iterator      = iterator_template<value_type>;
64     using const_iterator = iterator_template<const value_type>;
65
66     // CONSTRUCTORS, ET CETERA
67
68     // Default constructor.
69     explicit kernel(
70         size_t bucket_count = 16,
71         const hasher& hash = hasher(),
72         const key_equal& equal = key_equal(),
73         const allocator_type& alloc = allocator_type())
74     // Instances of functors.
75     : hash_{hash},
76       equal_{equal},
77       extract_{},
78       alloc_{alloc},
79       // Bucket vector.
80       buckets_{alloc_},
81       min_load_{0.3},
82       max_load_{0.7},
83       // We always have a number of buckets that is a power of two.
84       bucket_count_{upper_power_of_two(bucket_count)},
85       // ...
86       size_{0},
87       min_size_{size_t(bucket_count_ * min_load_)},

```

```

88     max_size_{size_t(bucket_count_ * max_load_)}
89     {
90         // Allocate space.
91         buckets_.resize(bucket_count_);
92     }
93
94     // Copy constructor.
95     kernel(
96         const kernel& other)
97     : hash_{other.hash_},
98       equal_{other.equal_},
99       extract_{},
100      alloc_{other.alloc_},
101      buckets_{other.buckets_},
102      min_load_{other.min_load_},
103      max_load_{other.max_load_},
104      bucket_count_{other.bucket_count_},
105      size_{other.size_},
106      min_size_{other.min_size_},
107      max_size_{other.max_size_}
108     {}
109
110     // Copy constructor with allocator.
111     kernel(
112         const kernel& other,
113         const allocator_type& alloc)
114     : hash_{other.hash_},
115       equal_{other.equal_},
116       extract_{},
117       alloc_{alloc},
118       buckets_{other.buckets_, alloc},
119       min_load_{other.min_load_},
120       max_load_{other.max_load_},
121       bucket_count_{other.bucket_count_},
122       size_{other.size_},
123       min_size_{other.min_size_},
124       max_size_{other.max_size_}
125     {}
126
127     // Move constructor.
128     kernel(kernel&& other)
129     : kernel{}
130     {
131         swap(*this, other);
132     }
133
134     // TODO
135     // kernel(kernel&& other, const allocator_type& alloc);
136
137     // Swapping (member).
138     void swap(kernel& other)
139     {
140         swap(*this, other);
141     }
142
143     // Swapping (friend).
144     friend void swap(kernel& lhs, kernel& rhs)
145     {
146         using std::swap;
147         swap(lhs.hash_, rhs.hash_);
148         swap(lhs.equal_, rhs.equal_);
149         swap(lhs.alloc_, rhs.alloc_);

```

```

150     swap(lhs.buckets_, rhs.buckets_);
151     swap(lhs.values_, rhs.values_);
152     swap(lhs.min_load_, rhs.min_load_);
153     swap(lhs.max_load_, rhs.max_load_);
154     swap(lhs.bucket_count_, rhs.bucket_count_);
155     swap(lhs.size_, rhs.size_);
156     swap(lhs.min_size_, rhs.min_size_);
157     swap(lhs.max_size_, rhs.max_size_);
158 }
159
160 // Copy assignment.
161 kernel& operator=(kernel other)
162 {
163     swap(*this, other);
164     return *this;
165 }
166
167 // ITERATOR GETTERS
168
169 iterator begin() {
170     return make_iterator(buckets_begin());
171 }
172
173 const_iterator begin() const {
174     return make_const_iterator(buckets_begin());
175 }
176
177 const_iterator cbegin() const {
178     return make_const_iterator(buckets_begin());
179 }
180
181 iterator end() {
182     return make_iterator(buckets_end());
183 }
184
185 const_iterator end() const {
186     return make_const_iterator(buckets_end());
187 }
188
189 const_iterator cend() const {
190     return make_const_iterator(buckets_end());
191 }
192
193 // CAPACITY
194
195 bool empty() const {
196     return size_ == 0;
197 }
198
199 size_t size() const {
200     return size_;
201 }
202
203 // CLEAR
204
205 void clear()
206 {
207     for (int index = 0; index < bucket_count_; ++index)
208     {
209         auto& bucket = buckets_[index];
210         bucket.memory()->value_type();
211         bucket.has_value(false);

```

```

212         bucket.hop_info.clear();
213     }
214     size_ = 0;
215 }
216
217 // ERASE
218
219 size_t erase(const key_type& key)
220 {
221     size_t erased = 0;
222
223     const size_t virtual_index = index_from_key(key);
224     auto& virtual_bucket = buckets_[virtual_index];
225
226     size_t hop = next_hop(virtual_bucket);
227     while (hop < neighborhood_size)
228     {
229         const size_t index = index_add(virtual_index, hop);
230         auto& bucket = buckets_[index];
231         if (equal_(key, extract_(*bucket.memory())))
232         {
233             bucket.memory()->~value_type();
234
235             bucket.has_value(false);
236
237             virtual_bucket.hop_info[hop] = false;
238
239             ++erased;
240         }
241         hop = next_hop(virtual_bucket, hop);
242     }
243
244     size_ -= erased;
245
246     // Rehash if this brought us below min load.
247     if (size_ < min_size_ && size_ > 16) {
248         rehash(bucket_count_/2);
249     }
250
251     return erased;
252 }
253
254 // COUNT
255
256 size_t count(const Key& key) const
257 {
258     size_t result = 0;
259
260     const size_t virtual_index = index_from_key(key);
261     auto& virtual_bucket = buckets_[virtual_index];
262
263     size_t hop = next_hop(virtual_bucket);
264     while (hop < neighborhood_size)
265     {
266         const size_t index = index_add(virtual_index, hop);
267         auto& bucket = buckets_[index];
268         if (equal_(key, extract_(*bucket.memory())))
269         {
270             ++result;
271         }
272         hop = next_hop(virtual_bucket, hop);
273     }

```

```

274         return count;
275     }
276 }
277 // OPERATOR []
278
279 auto& operator[](const Key& key) {
280     return mapped_extract_(
281         *(buckets_[find(key, index_from_key(key))].memory()));
282 }
283
284 const auto& operator[](const Key& key) const {
285     return mapped_extract_(
286         *(buckets_[find(key, index_from_key(key))].memory()));
287 }
288
289 // FIND
290
291 iterator find(const Key& key) {
292     return make_iterator(find(key, index_from_key(key)));
293 }
294
295 const_iterator find(const Key& key) const {
296     return make_const_iterator(find(key, index_from_key(key)));
297 }
298
299 // BUCKET INTERFACE
300
301 size_t bucket_count() const {
302     return bucket_count_;
303 }
304
305 // HASH POLICY
306
307 float load_factor() const {
308     return (float)size_/(float)bucket_count_;
309 }
310
311 float min_load_factor() const {
312     return min_load_;
313 }
314
315 float max_load_factor() const {
316     return max_load_;
317 }
318
319 void min_load_factor(float min_load)
320 {
321     min_load_ = min_load;
322
323     // Check if we need to rehash.
324     min_size_ = min_load_ * bucket_count_;
325     if (size_ < min_size_) {
326         rehash(bucket_count_/2);
327     }
328 }
329
330 void max_load_factor(float max_load)
331 {
332     max_load_ = max_load;
333
334     // Check if we need to rehash.
335

```

```

336     max_size_ = max_load_ * bucket_count_;
337     if (size_ > max_size_) {
338         rehash(bucket_count_*2);
339     }
340 }
341
342 void rehash(size_t count)
343 {
344     bucket_count_ = upper_power_of_two(count);
345
346     // Create the new bucket vector, saving the old one.
347     bucket_vector old_buckets{bucket_count_};
348     std::swap(buckets_, old_buckets);
349
350     // Set up the state so we can insert correctly.
351     size_ = 0;
352     min_size_ = min_load_ * bucket_count_;
353     max_size_ = max_load_ * bucket_count_;
354
355     // Rehash.
356     for (size_t index = 0; index < old_buckets.size(); ++index)
357     {
358         auto& bucket = old_buckets[index];
359         if (bucket.has_value())
360         {
361             insert(std::move(
362                 *bucket.memory()), index_from_value(*bucket.memory()));
363             // Detect recursive rehash and break.
364             if (bucket_count_ != count) { break; }
365         }
366     }
367 }
368
369 void reserve(size_t count)
370 {
371     rehash(std::ceil((float)count/max_load_factor()));
372 }
373
374 // OBSERVERS
375
376 hasher hash_function() const {
377     return hash_;
378 }
379
380 key_equal key_eq() const {
381     return equal_;
382 }
383
384 allocator_type get_allocator() const {
385     return alloc_;
386 }
387
388 protected:
389     // BUCKET TYPE
390
391     class bucket_type
392     {
393     public:
394         void has_value(bool has_value) {
395             hop_info[neighborhood_size] = has_value;
396         }
397     }

```

```

398     bool has_value() const {
399         return hop_info[neighborhood_size];
400     }
401
402     std::bitset<neighborhood_size+1> hop_info;
403
404     value_type* memory() {
405         return reinterpret_cast<value_type*>(&memory_);
406     }
407
408     const value_type* memory() const {
409         return reinterpret_cast<const value_type*>(&memory_);
410     }
411
412     struct { unsigned char _[sizeof(value_type)]; } memory_;
413 };
414
415 using bucket_vector = std::vector<bucket_type,
416     typename allocator_type::template rebind<bucket_type>::other>;
417
418 // ITERATOR TYPE
419
420 template <typename T>
421 class iterator_template : std::iterator<std::forward_iterator_tag, T>
422 {
423     friend kernel;
424
425 public:
426     iterator_template()
427         : index_(0),
428           buckets_(nullptr)
429     {}
430
431     iterator_template(const iterator_template& other)
432         : index_(other.index_),
433           buckets_(other.buckets_)
434     {}
435
436     iterator_template& operator=(const iterator_template& other)
437     {
438         index_ = other.index_;
439         buckets_ = other.buckets_;
440         return *this;
441     }
442
443     ~iterator_template()
444     {}
445
446     reference operator*() {
447         return *(buckets_->operator[] (index_).memory());
448     }
449
450     const_reference operator*() const {
451         return *(buckets_->operator[] (index_).memory());
452     }
453
454     pointer operator->() {
455         return buckets_->operator[] (index_).memory();
456     }
457
458     const_pointer operator->() const {
459         return buckets_->operator[] (index_).memory();

```

```

460     }
461
462     iterator_template& operator++()
463     {
464         do {
465             ++index_;
466         }
467         while (index_ != buckets_->size() &&
468             !buckets_->operator[](index_).has_value());
469         return *this;
470     }
471
472     iterator_template operator++(int)
473     {
474         iterator_template old(*this);
475         ++*this;
476         return old;
477     }
478
479     bool operator==(const iterator_template& other) const {
480         return index_ == other.index_ && buckets_ == other.buckets_;
481     }
482
483     bool operator!=(const iterator_template& other) const {
484         return index_ != other.index_ || buckets_ != other.buckets_;
485     }
486
487     private:
488         iterator_template(
489             size_t index,
490             bucket_vector* buckets)
491         : index_(index),
492           buckets_(buckets)
493         {}
494
495         size_t index_;
496
497         bucket_vector* buckets_;
498     };
499
500     // DATA MEMBERS
501
502     hasher hash_;
503     key_equal equal_;
504     key_extract extract_;
505     mapped_extract mapped_extract_;
506     allocator_type alloc_;
507
508     bucket_vector buckets_;
509     float min_load_;
510     float max_load_;
511
512     size_t bucket_count_;
513
514     size_t size_;
515     size_t min_size_;
516     size_t max_size_;
517
518     // UPPER POWER OF TWO
519
520     size_t upper_power_of_two(size_t x) const
521     {

```



```

522     // This implementation was found here (adjusted for 64-bit):
523     // http://graphics.stanford.edu/~seander/bithacks.html#RoundUpPowerOf2
524     --x;
525     x |= x >> 1;
526     x |= x >> 2;
527     x |= x >> 4;
528     x |= x >> 8;
529     x |= x >> 16;
530     x |= x >> 32;
531     ++x;
532     return x;
533 }
534
535 // INDEX HELPER
536
537 size_t index_from_value(const value_type& value) const {
538     return index_from_key(extract_(value));
539 }
540
541 size_t index_from_key(const key_type& key) const {
542     // This bitwise and is the same as doing a modulo because the bucket
543     // count is guaranteed to be a power of two.
544     return hash_(key) & (bucket_count_ - 1);
545 }
546
547 size_t index_add(size_t index, size_t x) const {
548     // This bitwise and is the same as doing a modulo because the bucket
549     // count is guaranteed to be a power of two.
550     return (index + x) & (bucket_count_ - 1);
551 }
552
553 size_t index_sub(size_t index, size_t x) const {
554     // As above, this corresponds to a modulo operation, except that we
555     // always get a positive value this way (as is desired).
556     return (index - x) & (bucket_count_ - 1);
557 }
558
559 // HOPPING
560
561 size_t next_hop(const bucket_type& bucket, int prev = -1) const
562 {
563     const size_t mask = 0xffffffffffff << (prev + 1);
564     const size_t hop_info = bucket.hop_info.to_ulong();
565     return __builtin_ffsl(hop_info & mask) - 1;
566 }
567
568 // ITERATOR HELPERS
569
570 size_t buckets_begin()
571 {
572     if (empty()) {
573         return buckets_.size();
574     } else {
575         auto bucket_it = buckets_.begin();
576         while (!bucket_it->has_value()) { ++bucket_it; }
577         return bucket_it - buckets_.begin();
578     }
579 }
580
581 size_t buckets_end() {
582     return buckets_.size();
583 }

```

```

584     iterator make_iterator(size_t index) {
585         return {index, &buckets_};
586     }
587 }
588
589     const_iterator make_const_iterator(size_t index) const {
590         return {index, &buckets_};
591     }
592
593 public:
594     std::pair<iterator, bool>
595     insert(value_type value)
596     {
597         size_t index = index_from_value(value);
598         size_t res = find(extract_(value), index);
599
600         if (res == buckets_.size()) {
601             return {insert(std::move(value), index), true};
602         } else {
603             return {make_iterator(res), false};
604         }
605     }
606
607     iterator
608     insert(
609         const_iterator hint,
610         value_type value)
611     {
612         (void)hint; // Silence 'unused parameter'
613         return insert(std::move(value)).first;
614     }
615
616 private:
617     // INSERT IMPLEMENTATION
618
619     iterator insert(value_type value, size_t virtual_index)
620     {
621         // Start by rehashing, if this will bring us above max load.
622         if (size_ == max_size_) {
623             // std::cout << std::endl;
624             // std::cout << "rehash (max load)";
625             // std::cout << std::endl;
626             rehash(bucket_count * 2);
627             return insert(value, index_from_value(value));
628         }
629
630         // Find the nearest free bucket, wrapping if we move past the end.
631         size_t free_dist = 0;
632         size_t free_index = virtual_index;
633         while (buckets_[free_index].has_value()) {
634             free_dist += 1;
635             free_index = index_add(free_index, 1);
636         }
637
638         // Move buckets until we have a free bucket in the neighborhood of our
639         // virtual bucket.
640         while (free_dist > neighborhood_size - 1)
641         {
642             // Find a virtual bucket that has values stored in a bucket before
643             // the free bucket we found.
644             size_t virtual_move_dist = neighborhood_size - 1;

```

```

645         size_t virtual_move_index = index_sub(free_index, virtual_move_dist
        );
646
647         size_t move_hop;
648
649         while (true)
650         {
651             auto& virtual_move_bucket = buckets_[virtual_move_index];
652             auto hop_info = virtual_move_bucket.hop_info.to_ulong();
653             move_hop = __builtin_ffsl(hop_info) - 1;
654
655             if (move_hop < virtual_move_dist) {
656                 break;
657             } else {
658                 // No luck, continue searching.
659                 virtual_move_dist -= 1;
660                 virtual_move_index = index_add(virtual_move_index, 1);
661
662                 if (virtual_move_dist == 0)
663                 {
664                     // All possibilities exhausted: resize, rehash, and
665                     // start over with the insertion.
666                     rehash(bucket_count*2);
667                     return insert(value, index_from_value(value));
668                 }
669             }
670         }
671
672         // Move.
673         const size_t move_dist = virtual_move_dist - move_hop;
674         const size_t move_index = index_add(virtual_move_index, move_hop);
675
676         auto& move_bucket = buckets_[move_index];
677         auto& free_bucket = buckets_[free_index];
678         new (free_bucket.memory())
679             value_type{std::move(*move_bucket.memory())};
680         move_bucket.memory()->~value_type();
681         move_bucket.has_value(false);
682         free_bucket.has_value(true);
683
684         auto& virtual_move = buckets_[virtual_move_index];
685         virtual_move.hop_info[move_hop] = false;
686         virtual_move.hop_info[virtual_move_dist] = true;
687
688         // The free bucket is now in the position of the moved bucket.
689         free_dist -= move_dist;
690         free_index = index_sub(free_index, move_dist);
691     }
692
693     // We should have a free bucket in the neighborhood now.
694     auto& free_bucket = buckets_[free_index];
695     new (free_bucket.memory()) value_type{std::move(value)};
696     free_bucket.has_value(true);
697
698     auto& virtual_bucket = buckets_[virtual_index];
699     virtual_bucket.hop_info[free_dist] = true;
700
701     ++size_;
702
703     return {free_index, &buckets_};
704 }
705

```

```

706 // FIND IMPLEMENTATION
707
708 size_t find(const key_type& key, size_t virtual_index) const
709 {
710     // Find the virtual bucket of this key.
711     const auto& virtual_bucket = buckets_[virtual_index];
712
713     // Go through each of the hops in the virtual bucket.
714     size_t hop = next_hop(virtual_bucket);
715     while (hop < neighborhood_size)
716     {
717         const size_t index = index_add(virtual_index, hop);
718         const auto& bucket = buckets_[index];
719         if (equal_(key, extract_(*bucket.memory()))) {
720             return index;
721         }
722         hop = next_hop(virtual_bucket, hop);
723     }
724
725     // We found nothing, return end.
726     return bucket_count_;
727 }
728 };
729
730 } // namespace detail
731
732 // UNORDERED SET
733
734 #define BASE detail::kernel<\
735     Key,\
736     Key,\
737     Hash,\
738     extract::identity,\
739     extract::identity,\
740     KeyEqual,\
741     Allocator>
742 template <
743     class Key,
744     class Hash = std::hash<Key>,
745     class KeyEqual = std::equal_to<Key>,
746     class Allocator = std::allocator<Key>>
747 class unordered_set : public BASE
748 {};
749 #undef BASE
750
751 // UNORDERED MAP
752
753 #define BASE detail::kernel<\
754     std::pair<const Key, T>,\
755     Key,\
756     Hash,\
757     extract::first,\
758     extract::second,\
759     KeyEqual,\
760     Allocator>
761 template <
762     class Key,
763     class T,
764     class Hash = std::hash<Key>,
765     class KeyEqual = std::equal_to<Key>,
766     class Allocator = std::allocator<std::pair<const Key, T>>>
767 class unordered_map : public BASE

```

```
768 {
769 public:
770     using mapped_type = T;
771 };
772 #undef BASE
773
774 } // namespace hopscotch
```

A.6 linear.hpp

```
1 #pragma once
2
3 #include <algorithm>
4 #include <bitset>
5 #include <cassert>
6 #include <cmath>
7 #include <cstdint>
8 #include <cstring>
9 #include <functional>
10 #include <initializer_list>
11 #include <iterator>
12 #include <memory>
13 #include <utility>
14 #include <vector>
15
16 #include <iostream>
17
18 namespace linear
19 {
20     namespace detail
21     {
22         // KERNEL
23
24         template <
25             class Value,
26             class Key,
27             class Hash,
28             class KeyExtract,
29             class MappedExtract,
30             class KeyEqual,
31             class Allocator>
32         class kernel
33         {
34         protected:
35             // Defined below.
36             template <typename T> class iterator_template;
37
38         public:
39             // MEMBER TYPES
40
41             using key_type      = Key;
42             using value_type    = Value;
43             using size_type     = std::size_t;
44             using difference_type = std::ptrdiff_t;
45             using hasher        = Hash;
46             using key_equal     = KeyEqual;
47             using key_extract   = KeyExtract;
48             using mapped_extract = MappedExtract;
49             using allocator_type = Allocator;
```

```

50 using reference      = value_type&;
51 using const_reference = const value_type&;
52 using pointer        =
53     typename std::allocator_traits<Allocator>::pointer;
54 using const_pointer  =
55     typename std::allocator_traits<Allocator>::const_pointer;
56 using iterator        = iterator_template<value_type>;
57 using const_iterator = iterator_template<const value_type>;
58
59 // CONSTRUCTORS, ET CETERA
60
61 // Default constructor.
62 explicit kernel(
63     size_t bucket_count = 16,
64     const hasher& hash = hasher(),
65     const key_equal& equal = key_equal(),
66     const allocator_type& alloc = allocator_type())
67     // Instances of functors.
68 : hash_{hash},
69   equal_{equal},
70   extract_{},
71   alloc_{alloc},
72   // Bucket vector.
73   buckets_{alloc_},
74   min_load_{0.3},
75   max_load_{0.7},
76   // We always have a number of buckets that is a power of two.
77   bucket_count_{upper_power_of_two(bucket_count)},
78   // ...
79   size_{0},
80   min_size_{size_t(bucket_count_ * min_load_)},
81   max_size_{size_t(bucket_count_ * max_load_)}
82 {
83     // Allocate space.
84     buckets_.resize(bucket_count_);
85 }
86
87 // Copy constructor.
88 kernel(
89     const kernel& other)
90 : hash_{other.hash_},
91   equal_{other.equal_},
92   extract_{},
93   alloc_{other.alloc_},
94   buckets_{other.buckets_},
95   min_load_{other.min_load_},
96   max_load_{other.max_load_},
97   bucket_count_{other.bucket_count_},
98   size_{other.size_},
99   min_size_{other.min_size_},
100  max_size_{other.max_size_}
101 {}
102
103 // Copy constructor with allocator.
104 kernel(
105     const kernel& other,
106     const allocator_type& alloc)
107 : hash_{other.hash_},
108   equal_{other.equal_},
109   extract_{},
110   alloc_{alloc},
111   buckets_{other.buckets_, alloc},

```

```
112     min_load_{other.min_load_},
113     max_load_{other.max_load_},
114     bucket_count_{other.bucket_count_},
115     size_{other.size_},
116     min_size_{other.min_size_},
117     max_size_{other.max_size_}
118 {}
119
120 // Move constructor.
121 kernel(kernel&& other)
122 : kernel{}
123 {
124     swap(*this, other);
125 }
126
127 // Swapping (member).
128 void swap(kernel& other)
129 {
130     swap(*this, other);
131 }
132
133 // Swapping (friend).
134 friend void swap(kernel& lhs, kernel& rhs)
135 {
136     using std::swap;
137     swap(lhs.hash_, rhs.hash_);
138     swap(lhs.equal_, rhs.equal_);
139     swap(lhs.alloc_, rhs.alloc_);
140     swap(lhs.buckets_, rhs.buckets_);
141     swap(lhs.min_load_, rhs.min_load_);
142     swap(lhs.max_load_, rhs.max_load_);
143     swap(lhs.bucket_count_, rhs.bucket_count_);
144     swap(lhs.size_, rhs.size_);
145     swap(lhs.min_size_, rhs.min_size_);
146     swap(lhs.max_size_, rhs.max_size_);
147 }
148
149 // Copy assignment.
150 kernel& operator=(kernel other)
151 {
152     swap(*this, other);
153     return *this;
154 }
155
156 // ITERATOR GETTERS
157
158 iterator begin() {
159     return make_iterator(buckets_begin());
160 }
161
162 const_iterator begin() const {
163     return make_const_iterator(buckets_begin());
164 }
165
166 const_iterator cbegin() const {
167     return make_const_iterator(buckets_begin());
168 }
169
170 iterator end() {
171     return make_iterator(buckets_end());
172 }
173
```

```

174     const_iterator end() const {
175         return make_const_iterator(buckets_end());
176     }
177
178     const_iterator cend() const {
179         return make_const_iterator(buckets_end());
180     }
181
182     // CAPACITY
183
184     bool empty() const {
185         return size_ == 0;
186     }
187
188     size_t size() const {
189         return size_;
190     }
191
192     // CLEAR
193
194     void clear()
195     {
196         for (int index = 0; index < bucket_count_; ++index)
197         {
198             auto& bucket = buckets_[index];
199             bucket.memory()->~value_type();
200         }
201         buckets_.clear();
202         size_ = 0;
203     }
204
205     // ERASE
206
207     size_t erase(const key_type& key)
208     {
209         size_t index = index_from_key(key);
210         size_t erased_index;
211         size_t erased_count = 0;
212
213         while (buckets_[index].has_value) {
214             auto& bucket = buckets_[index];
215
216             if (equal(key, extract_(*bucket.memory()))) {
217                 bucket.memory()->~value_type();
218                 bucket.has_value = false;
219                 erased_index = index;
220                 erased_count = 1;
221                 size_ -= 1;
222                 index = index_add(index, 1);
223                 break;
224             }
225
226             index = index_add(index, 1);
227         }
228
229         while (buckets_[index].has_value) {
230             const auto& move_bucket = buckets_[index];
231             const auto& hash = index_from_key(extract_(*move_bucket.memory()));
232
233             if ((erased_index < index &&
234                 (hash <= erased_index || hash > index)) ||
235                 (erased_index > index &&

```



```

236         (hash <= erased_index && hash > index)))
237     {
238         auto& free_bucket = buckets_[erased_index];
239         new (free_bucket.memory())
240             value_type{std::move(*move_bucket.memory())};
241         move_bucket.memory()->~value_type();
242
243         buckets_[index].has_value = false;
244         buckets_[erased_index].has_value = true;
245         erased_index = index;
246     }
247
248     index = index_add(index, 1);
249 }
250
251 // Rehash if this brought us below min load.
252 if (size_ < min_size_ && size_ > 16) {
253     rehash(bucket_count_/2);
254 }
255
256 return erased_count;
257 }
258
259 // COUNT
260
261 size_t count(const Key& key) const
262 {
263     size_t result = 0;
264
265     const size_t virtual_index = index_from_key(key);
266
267     while (buckets_[virtual_index]) {
268         auto& value = buckets_[virtual_index];
269         if (equal_(key, extract_(*value.memory())))
270             {
271                 ++result;
272             }
273         virtual_index = index_add(virtual_index, 1);
274     }
275
276     return result;
277 }
278
279 // OPERATOR[]
280
281 auto& operator[](const Key& key) {
282     return mapped_extract_(
283         *(buckets_[find(key, index_from_key(key))].memory()));
284 }
285
286 const auto& operator[](const Key& key) const {
287     return mapped_extract_(
288         *(buckets_[find(key, index_from_key(key))].memory()));
289 }
290
291 // FIND
292
293 iterator find(const Key& key) {
294     return make_iterator(find(key, index_from_key(key)));
295 }
296
297 const_iterator find(const Key& key) const {

```

```
298     return make_const_iterator(find(key, index_from_key(key)));
299 }
300
301 // BUCKET INTERFACE
302
303 size_t bucket_count() const {
304     return bucket_count_;
305 }
306
307 // HASH POLICY
308
309 float load_factor() const {
310     return (float)size_/(float)bucket_count_;
311 }
312
313 float min_load_factor() const {
314     return min_load_;
315 }
316
317 float max_load_factor() const {
318     return max_load_;
319 }
320
321 void min_load_factor(float min_load)
322 {
323     min_load_ = min_load;
324
325     // Check if we need to rehash.
326     min_size_ = min_load_ * bucket_count_;
327     if (size_ < min_size_) {
328         rehash(bucket_count_/2);
329     }
330 }
331
332 void max_load_factor(float max_load)
333 {
334     max_load_ = max_load;
335
336     // Check if we need to rehash.
337     max_size_ = max_load_ * bucket_count_;
338     if (size_ > max_size_) {
339         rehash(bucket_count_*2);
340     }
341 }
342
343 void rehash(size_t count)
344 {
345     bucket_count_ = upper_power_of_two(count);
346
347     // Create the new bucket vector, saving the old one.
348     bucket_vector old_buckets(bucket_count_);
349     std::swap(buckets_, old_buckets);
350
351     // Set up the state so we can insert correctly.
352     size_ = 0;
353     min_size_ = min_load_ * bucket_count_;
354     max_size_ = max_load_ * bucket_count_;
355
356     // Rehash.
357     for (size_t index = 0; index < old_buckets.size(); ++index)
358     {
359         if (old_buckets[index].has_value)
```

```

360         {
361             auto& bucket = old_buckets[index];
362             insert(std::move(*bucket.memory()),
363                 index_from_value(*bucket.memory()));
364             // Detect recursive rehash and break.
365             if (bucket_count_ != count) { break; }
366         }
367     }
368 }
369
370 void reserve(size_t count)
371 {
372     rehash(std::ceil((float)count/max_load_factor()));
373 }
374
375 // OBSERVERS
376
377 hasher hash_function() const {
378     return hash_;
379 }
380
381 key_equal key_eq() const {
382     return equal_;
383 }
384
385 allocator_type get_allocator() const {
386     return alloc_;
387 }
388
389 protected:
390     // BUCKET TYPE
391
392     class bucket_type
393     {
394     public:
395         value_type* memory() {
396             return reinterpret_cast<value_type*>(&memory_);
397         }
398
399         const value_type* memory() const {
400             return reinterpret_cast<const value_type*>(&memory_);
401         }
402
403         bool has_value;
404
405     private:
406         struct { unsigned char _[sizeof(value_type)]; } memory_;
407     };
408
409     using bucket_vector = std::vector<bucket_type,
410         typename allocator_type::template rebind<bucket_type>::other>;
411
412     // ITERATOR TYPE
413
414     template <typename T>
415     class iterator_template : std::iterator<std::forward_iterator_tag, T>
416     {
417         // NOT CONVERTED
418
419         friend kernel;
420
421     public:

```

```

422     iterator_template()
423     : index_(0),
424       buckets_(nullptr)
425     {}
426
427     iterator_template(const iterator_template& other)
428     : index_(other.index_),
429       buckets_(other.buckets_)
430     {}
431
432     iterator_template& operator=(const iterator_template& other)
433     {
434         index_ = other.index_;
435         buckets_ = other.buckets_;
436         return *this;
437     }
438
439     ~iterator_template()
440     {}
441
442     reference operator*() {
443         return *(buckets_->operator[] (index_).memory());
444     }
445
446     const_reference operator*() const {
447         return *(buckets_->operator[] (index_).memory());
448     }
449
450     pointer operator->() {
451         return buckets_->operator[] (index_).memory();
452     }
453
454     const_pointer operator->() const {
455         return buckets_->operator[] (index_).memory();
456     }
457
458     iterator_template& operator++()
459     {
460         do {
461             ++index_;
462         }
463         while (index_ != buckets_->size() &&
464              !buckets_->operator[] (index_).has_value);
465         return *this;
466     }
467
468     iterator_template operator++(int)
469     {
470         iterator_template old(*this);
471         ++*this;
472         return old;
473     }
474
475     bool operator==(const iterator_template& other) const {
476         return index_ == other.index_ && buckets_ == other.buckets_;
477     }
478
479     bool operator!=(const iterator_template& other) const {
480         return index_ != other.index_ || buckets_ != other.buckets_;
481     }
482
483 private:

```

```

484     iterator_template(
485         size_t index,
486         bucket_vector* buckets)
487     : index_(index),
488       buckets_(buckets)
489     {}
490
491     size_t index_;
492
493     bucket_vector* buckets_;
494 };
495
496 // DATA MEMBERS
497
498 hasher hash_;
499 key_equal equal_;
500 key_extract extract_;
501 mapped_extract mapped_extract_;
502 allocator_type alloc_;
503
504 bucket_vector buckets_;
505 float min_load_;
506 float max_load_;
507
508 size_t bucket_count_;
509
510 size_t size_;
511 size_t min_size_;
512 size_t max_size_;
513
514 // UPPER POWER OF TWO
515
516 size_t upper_power_of_two(size_t x) const
517 {
518     // This implementation was found here (adjusted for 64-bit):
519     // http://graphics.stanford.edu/~seander/bithacks.html#RoundUpPowerOf2
520     --x;
521     x |= x >> 1;
522     x |= x >> 2;
523     x |= x >> 4;
524     x |= x >> 8;
525     x |= x >> 16;
526     x |= x >> 32;
527     ++x;
528     return x;
529 }
530
531 // INDEX HELPER
532
533 size_t index_from_value(const value_type& value) const {
534     return index_from_key(extract_(value));
535 }
536
537 size_t index_from_key(const key_type& key) const {
538     // This bitwise and is the same as doing a modulo because the bucket
539     // count is guaranteed to be a power of two.
540     return hash_(key) & (bucket_count_ - 1);
541 }
542
543 size_t index_add(size_t index, size_t x) const {
544     // This bitwise and is the same as doing a modulo because the bucket
545     // count is guaranteed to be a power of two.

```

```

546     return (index + x) & (bucket_count_ - 1);
547 }
548
549 size_t index_sub(size_t index, size_t x) const {
550     // As above, this corresponds to a modulo operation, except that we
551     // always get a positive value this way (as is desired).
552     return (index - x) & (bucket_count_ - 1);
553 }
554
555 // ITERATOR HELPERS
556
557 size_t buckets_begin()
558 {
559     if (empty()) {
560         return buckets_.size();
561     } else {
562         auto bucket_it = buckets_.begin();
563         while (!bucket_it->has_value) { ++bucket_it; }
564         return bucket_it - buckets_.begin();
565     }
566 }
567
568 size_t buckets_end() {
569     return buckets_.size();
570 }
571
572 iterator make_iterator(size_t index) {
573     return {index, &buckets_};
574 }
575
576 const_iterator make_const_iterator(size_t index) const {
577     return {index, &buckets_};
578 }
579
580 public:
581     // INSERT
582
583     std::pair<iterator, bool>
584     insert(value_type value)
585     {
586         size_t index = index_from_value(value);
587         size_t res = find(extract_(value), index);
588
589         if (res == buckets_.size()) {
590             return {insert(std::move(value), index), true};
591         } else {
592             return {make_iterator(res), false};
593         }
594     }
595
596     iterator
597     insert(
598         const_iterator hint,
599         value_type value)
600     {
601         (void)hint; // Silence 'unused parameter'
602         return insert(std::move(value)).first;
603     }
604
605 private:
606     // INSERT IMPLEMENTATION
607

```

```

608     iterator insert(value_type value, size_t virtual_index)
609     {
610         // Start by rehashing, if this will bring us above max load.
611         if (size_ == max_size_) {
612             rehash(bucket_count_*2);
613             return insert(value, index_from_value(value));
614         }
615
616         // Find the nearest free bucket, wrapping if we move past the end.
617         size_t free_dist = 0;
618         size_t free_index = virtual_index;
619         while (buckets_[free_index].has_value) {
620             free_dist += 1;
621             free_index = index_add(free_index, 1);
622         }
623
624         // We should have a free bucket in the neighborhood now.
625         auto& free_bucket = buckets_[free_index];
626         new (free_bucket.memory()) value_type{std::move(value)};
627
628         buckets_[free_index].has_value = true;
629
630         ++size_;
631
632         return {free_index, &buckets_};
633     }
634
635     // FIND IMPLEMENTATION
636
637     size_t find(const key_type& key, size_t virtual_index) const
638     {
639         // Search from there until we find what we are looking for or an
640         // empty bucket
641         while (buckets_[virtual_index].has_value) {
642             const auto& value = buckets_[virtual_index];
643             if (equal(key, extract_(*value.memory()))) {
644                 return virtual_index;
645             }
646             virtual_index = index_add(virtual_index, 1);
647         }
648
649         // We found nothing, return end.
650         return bucket_count_;
651     }
652 };
653
654 } // namespace detail
655
656 // UNORDERED SET
657
658 #define BASE detail::kernel<\
659     Key,\
660     Key,\
661     Hash,\
662     extract::identity,\
663     extract::identity,\
664     KeyEqual,\
665     Allocator>
666 template <
667     class Key,
668     class Hash = std::hash<Key>,
669     class KeyEqual = std::equal_to<Key>,

```

```

670     class Allocator = std::allocator<Key>>
671     class unordered_set : public BASE
672     {};
673 #undef BASE
674
675 // UNORDERED MAP
676
677 #define BASE detail::kernel<\
678     std::pair<const Key, T>,\
679     Key,\
680     Hash,\
681     extract::first,\
682     extract::second,\
683     KeyEqual,\
684     Allocator>
685 template <
686     class Key,
687     class T,
688     class Hash = std::hash<Key>,
689     class KeyEqual = std::equal_to<Key>,
690     class Allocator = std::allocator<std::pair<const Key, T>>>
691 class unordered_map : public BASE
692 {
693 public:
694     using mapped_type = T;
695 };
696 #undef BASE
697
698 } // namespace hopscotch

```

A.7 tabulation.hpp

```

1 #pragma once
2
3 #include <algorithm>
4 #include <bitset>
5 #include <cassert>
6 #include <cmath>
7 #include <cstdint>
8 #include <cstring>
9 #include <functional>
10 #include <initializer_list>
11 #include <iterator>
12 #include <memory>
13 #include <utility>
14 #include <vector>
15
16 #include <iostream>
17
18 namespace linear
19 {
20     namespace detail
21     {
22         // KERNEL
23
24         template <
25             class Value,
26             class Key,
27             class Hash,
28             class KeyExtract,
29             class MappedExtract,

```



```

30     class KeyEqual,
31     class Allocator>
32 class kernel
33 {
34 protected:
35     // Defined below.
36     template <typename T> class iterator_template;
37
38 public:
39     // MEMBER TYPES
40
41     using key_type      = Key;
42     using value_type    = Value;
43     using size_type    = std::size_t;
44     using difference_type = std::ptrdiff_t;
45     using hasher        = Hash;
46     using key_equal     = KeyEqual;
47     using key_extract   = KeyExtract;
48     using mapped_extract = MappedExtract;
49     using allocator_type = Allocator;
50     using reference     = value_type&;
51     using const_reference = const value_type&;
52     using pointer       =
53         typename std::allocator_traits<Allocator>::pointer;
54     using const_pointer =
55         typename std::allocator_traits<Allocator>::const_pointer;
56     using iterator      = iterator_template<value_type>;
57     using const_iterator = iterator_template<const value_type>;
58
59     // CONSTRUCTORS, ET CETERA
60
61     // Default constructor.
62     explicit kernel(
63         size_t bucket_count = 16,
64         const hasher& hash = hasher(),
65         const key_equal& equal = key_equal(),
66         const allocator_type& alloc = allocator_type())
67         // Instances of functors.
68         : hash_{hash},
69         equal_{equal},
70         extract_{},
71         alloc_{alloc},
72         // Bucket vector.
73         buckets_{alloc_},
74         min_load_{0.3},
75         max_load_{0.7},
76         // We always have a number of buckets that is a power of two.
77         bucket_count_{upper_power_of_two(bucket_count)},
78         // ...
79         size_{0},
80         min_size_{size_t(bucket_count_ * min_load_)},
81         max_size_{size_t(bucket_count_ * max_load_)}
82     {
83         // Allocate space.
84         buckets_.resize(bucket_count_);
85     }
86
87     // Copy constructor.
88     kernel(
89         const kernel& other)
90     : hash_{other.hash_},
91       equal_{other.equal_},

```

```

92     extract_ {},
93     alloc_{other.alloc_},
94     buckets_{other.buckets_},
95     min_load_{other.min_load_},
96     max_load_{other.max_load_},
97     bucket_count_{other.bucket_count_},
98     size_{other.size_},
99     min_size_{other.min_size_},
100    max_size_{other.max_size_}
101 {}
102
103 // Copy constructor with allocator.
104 kernel(
105     const kernel& other,
106     const allocator_type& alloc)
107 : hash_{other.hash_},
108   equal_{other.equal_},
109   extract_ {},
110   alloc_{alloc},
111   buckets_{other.buckets_, alloc},
112   min_load_{other.min_load_},
113   max_load_{other.max_load_},
114   bucket_count_{other.bucket_count_},
115   size_{other.size_},
116   min_size_{other.min_size_},
117   max_size_{other.max_size_}
118 {}
119
120 // Move constructor.
121 kernel(kernel&& other)
122 : kernel{}
123 {
124     swap(*this, other);
125 }
126
127 // Swapping (member).
128 void swap(kernel& other)
129 {
130     swap(*this, other);
131 }
132
133 // Swapping (friend).
134 friend void swap(kernel& lhs, kernel& rhs)
135 {
136     using std::swap;
137     swap(lhs.hash_, rhs.hash_);
138     swap(lhs.equal_, rhs.equal_);
139     swap(lhs.alloc_, rhs.alloc_);
140     swap(lhs.buckets_, rhs.buckets_);
141     swap(lhs.min_load_, rhs.min_load_);
142     swap(lhs.max_load_, rhs.max_load_);
143     swap(lhs.bucket_count_, rhs.bucket_count_);
144     swap(lhs.size_, rhs.size_);
145     swap(lhs.min_size_, rhs.min_size_);
146     swap(lhs.max_size_, rhs.max_size_);
147 }
148
149 // Copy assignment.
150 kernel& operator=(kernel other)
151 {
152     swap(*this, other);
153     return *this;

```

```

154     }
155
156     // ITERATOR GETTERS
157
158     iterator begin() {
159         return make_iterator(buckets_begin());
160     }
161
162     const_iterator begin() const {
163         return make_const_iterator(buckets_begin());
164     }
165
166     const_iterator cbegin() const {
167         return make_const_iterator(buckets_begin());
168     }
169
170     iterator end() {
171         return make_iterator(buckets_end());
172     }
173
174     const_iterator end() const {
175         return make_const_iterator(buckets_end());
176     }
177
178     const_iterator cend() const {
179         return make_const_iterator(buckets_end());
180     }
181
182     // CAPACITY
183
184     bool empty() const {
185         return size_ == 0;
186     }
187
188     size_t size() const {
189         return size_;
190     }
191
192     // CLEAR
193
194     void clear()
195     {
196         for (int index = 0; index < bucket_count_; ++index)
197         {
198             auto& bucket = buckets_[index];
199             bucket.memory()->~value_type();
200         }
201         buckets_.clear();
202         size_ = 0;
203     }
204
205     // ERASE
206
207     size_t erase(const key_type& key)
208     {
209         size_t index = index_from_key(key);
210         size_t erased_index;
211         size_t erased_count = 0;
212
213         while (buckets_[index].has_value) {
214             auto& bucket = buckets_[index];
215

```

```

216         if (equal_(key, extract_(*bucket.memory()))) {
217             bucket.memory()->~value_type();
218             bucket.has_value = false;
219             erased_index = index;
220             erased_count = 1;
221             size_ -= 1;
222             index = index_add(index, 1);
223             break;
224         }
225
226         index = index_add(index, 1);
227     }
228
229     while (buckets_[index].has_value) {
230         const auto& move_bucket = buckets_[index];
231         const auto& hash = index_from_key(extract_(*move_bucket.memory()));
232
233         if ((erased_index < index &&
234             (hash <= erased_index || hash > index)) ||
235             (erased_index > index &&
236             (hash <= erased_index && hash > index)))
237         {
238             auto& free_bucket = buckets_[erased_index];
239             new (free_bucket.memory())
240                 value_type{std::move(*move_bucket.memory())};
241             move_bucket.memory()->~value_type();
242
243             buckets_[index].has_value = false;
244             buckets_[erased_index].has_value = true;
245             erased_index = index;
246         }
247
248         index = index_add(index, 1);
249     }
250
251     // Rehash if this brought us below min load.
252     if (size_ < min_size_ && size_ > 16) {
253         rehash(bucket_count_/2);
254     }
255
256     return erased_count;
257 }
258
259 // COUNT
260
261 size_t count(const Key& key) const
262 {
263     size_t result = 0;
264
265     const size_t virtual_index = index_from_key(key);
266
267     while (buckets_[virtual_index]) {
268         auto& value = buckets_[virtual_index];
269         if (equal_(key, extract_(*value.memory())))
270         {
271             ++result;
272         }
273         virtual_index = index_add(virtual_index, 1);
274     }
275
276     return result;
277 }

```

```

278
279 // OPERATOR[]
280
281 auto& operator[] (const Key& key) {
282     return mapped_extract_(
283         *(buckets_[find(key, index_from_key(key))].memory()));
284 }
285
286 const auto& operator[] (const Key& key) const {
287     return mapped_extract_(
288         *(buckets_[find(key, index_from_key(key))].memory()));
289 }
290
291 // FIND
292
293 iterator find(const Key& key) {
294     return make_iterator(find(key, index_from_key(key)));
295 }
296
297 const_iterator find(const Key& key) const {
298     return make_const_iterator(find(key, index_from_key(key)));
299 }
300
301 // BUCKET INTERFACE
302
303 size_t bucket_count() const {
304     return bucket_count_;
305 }
306
307 // HASH POLICY
308
309 float load_factor() const {
310     return (float)size_/(float)bucket_count_;
311 }
312
313 float min_load_factor() const {
314     return min_load_;
315 }
316
317 float max_load_factor() const {
318     return max_load_;
319 }
320
321 void min_load_factor(float min_load)
322 {
323     min_load_ = min_load;
324
325     // Check if we need to rehash.
326     min_size_ = min_load_ * bucket_count_;
327     if (size_ < min_size_) {
328         rehash(bucket_count_/2);
329     }
330 }
331
332 void max_load_factor(float max_load)
333 {
334     max_load_ = max_load;
335
336     // Check if we need to rehash.
337     max_size_ = max_load_ * bucket_count_;
338     if (size_ > max_size_) {
339         rehash(bucket_count_*2);

```

```

340     }
341 }
342
343 void rehash(size_t count)
344 {
345     bucket_count_ = upper_power_of_two(count);
346
347     // Create the new bucket vector, saving the old one.
348     bucket_vector old_buckets(bucket_count_);
349     std::swap(buckets_, old_buckets);
350
351     // Set up the state so we can insert correctly.
352     size_ = 0;
353     min_size_ = min_load_ * bucket_count_;
354     max_size_ = max_load_ * bucket_count_;
355
356     // Rehash.
357     for (size_t index = 0; index < old_buckets.size(); ++index)
358     {
359         if (old_buckets[index].has_value)
360         {
361             auto& bucket = old_buckets[index];
362             insert(std::move(*bucket.memory()),
363                 index_from_value(*bucket.memory()));
364             // Detect recursive rehash and break.
365             if (bucket_count_ != count) { break; }
366         }
367     }
368 }
369
370 void reserve(size_t count)
371 {
372     rehash(std::ceil((float)count/max_load_factor()));
373 }
374
375 // OBSERVERS
376
377 hasher hash_function() const {
378     return hash_;
379 }
380
381 key_equal key_eq() const {
382     return equal_;
383 }
384
385 allocator_type get_allocator() const {
386     return alloc_;
387 }
388
389 protected:
390     // BUCKET TYPE
391
392     class bucket_type
393     {
394     public:
395         value_type* memory() {
396             return reinterpret_cast<value_type*>(&memory_);
397         }
398
399         const value_type* memory() const {
400             return reinterpret_cast<const value_type*>(&memory_);
401         }

```

```

402         bool has_value;
403
404     private:
405         struct { unsigned char _[sizeof(value_type)]; } memory_;
406     };
407
408     using bucket_vector = std::vector<bucket_type,
409         typename allocator_type::template rebind<bucket_type>::other>;
410
411     // ITERATOR TYPE
412
413     template <typename T>
414     class iterator_template : std::iterator<std::forward_iterator_tag, T>
415     {
416     private:
417         // NOT CONVERTED
418
419     public:
420         friend kernel;
421
422     public:
423         iterator_template()
424         : index_(0),
425           buckets_(nullptr)
426         {}
427
428         iterator_template(const iterator_template& other)
429         : index_(other.index_),
430           buckets_(other.buckets_)
431         {}
432
433         iterator_template& operator=(const iterator_template& other)
434         {
435             index_ = other.index_;
436             buckets_ = other.buckets_;
437             return *this;
438         }
439
440         ~iterator_template()
441         {}
442
443         reference operator*() {
444             return *(buckets_->operator[] (index_).memory());
445         }
446
447         const_reference operator*() const {
448             return *(buckets_->operator[] (index_).memory());
449         }
450
451         pointer operator->() {
452             return buckets_->operator[] (index_).memory();
453         }
454
455         const_pointer operator->() const {
456             return buckets_->operator[] (index_).memory();
457         }
458
459         iterator_template& operator++()
460         {
461             do {
462                 ++index_;
463             }
464             while (index_ != buckets_->size() &&

```

```

464         !buckets_->operator[](index_).has_value);
465     return *this;
466 }
467
468 iterator_template operator++(int)
469 {
470     iterator_template old(*this);
471     ++*this;
472     return old;
473 }
474
475 bool operator==(const iterator_template& other) const {
476     return index_ == other.index_ && buckets_ == other.buckets_;
477 }
478
479 bool operator!=(const iterator_template& other) const {
480     return index_ != other.index_ || buckets_ != other.buckets_;
481 }
482
483 private:
484     iterator_template(
485         size_t index,
486         bucket_vector* buckets)
487     : index_(index),
488       buckets_(buckets)
489     {}
490
491     size_t index_;
492
493     bucket_vector* buckets_;
494 };
495
496 // DATA MEMBERS
497
498 hasher hash_;
499 key_equal equal_;
500 key_extract extract_;
501 mapped_extract mapped_extract_;
502 allocator_type alloc_;
503
504 bucket_vector buckets_;
505 float min_load_;
506 float max_load_;
507
508 size_t bucket_count_;
509
510 size_t size_;
511 size_t min_size_;
512 size_t max_size_;
513
514 // UPPER POWER OF TWO
515
516 size_t upper_power_of_two(size_t x) const
517 {
518     // This implementation was found here (adjusted for 64-bit):
519     // http://graphics.stanford.edu/~seander/bithacks.html#RoundUpPowerOf2
520     --x;
521     x |= x >> 1;
522     x |= x >> 2;
523     x |= x >> 4;
524     x |= x >> 8;
525     x |= x >> 16;

```



```

526     x |= x >> 32;
527     ++x;
528     return x;
529 }
530
531 // INDEX HELPER
532
533 size_t index_from_value(const value_type& value) const {
534     return index_from_key(extract_(value));
535 }
536
537 size_t index_from_key(const key_type& key) const {
538     // This bitwise and is the same as doing a modulo because the bucket
539     // count is guaranteed to be a power of two.
540     return hash_(key) & (bucket_count_ - 1);
541 }
542
543 size_t index_add(size_t index, size_t x) const {
544     // This bitwise and is the same as doing a modulo because the bucket
545     // count is guaranteed to be a power of two.
546     return (index + x) & (bucket_count_ - 1);
547 }
548
549 size_t index_sub(size_t index, size_t x) const {
550     // As above, this corresponds to a modulo operation, except that we
551     // always get a positive value this way (as is desired).
552     return (index - x) & (bucket_count_ - 1);
553 }
554
555 // ITERATOR HELPERS
556
557 size_t buckets_begin()
558 {
559     if (empty()) {
560         return buckets_.size();
561     } else {
562         auto bucket_it = buckets_.begin();
563         while (!bucket_it->has_value) { ++bucket_it; }
564         return bucket_it - buckets_.begin();
565     }
566 }
567
568 size_t buckets_end() {
569     return buckets_.size();
570 }
571
572 iterator make_iterator(size_t index) {
573     return {index, &buckets_};
574 }
575
576 const_iterator make_const_iterator(size_t index) const {
577     return {index, &buckets_};
578 }
579
580 public:
581     // INSERT
582
583     std::pair<iterator, bool>
584     insert(value_type value)
585     {
586         size_t index = index_from_value(value);
587         size_t res = find(extract_(value), index);

```

```

588
589     if (res == buckets_.size()) {
590         return {insert(std::move(value), index), true};
591     } else {
592         return {make_iterator(res), false};
593     }
594 }
595
596 iterator
597 insert(
598     const_iterator hint,
599     value_type value)
600 {
601     (void)hint; // Silence 'unused parameter'
602     return insert(std::move(value)).first;
603 }
604
605 private:
606     // INSERT IMPLEMENTATION
607
608     iterator insert(value_type value, size_t virtual_index)
609     {
610         // Start by rehashing, if this will bring us above max load.
611         if (size_ == max_size_) {
612             rehash(bucket_count*2);
613             return insert(value, index_from_value(value));
614         }
615
616         // Find the nearest free bucket, wrapping if we move past the end.
617         size_t free_dist = 0;
618         size_t free_index = virtual_index;
619         while (buckets_[free_index].has_value) {
620             free_dist += 1;
621             free_index = index_add(free_index, 1);
622         }
623
624         // We should have a free bucket in the neighborhood now.
625         auto& free_bucket = buckets_[free_index];
626         new (free_bucket.memory()) value_type{std::move(value)};
627
628         buckets_[free_index].has_value = true;
629
630         ++size_;
631
632         return {free_index, &buckets_};
633     }
634
635     // FIND IMPLEMENTATION
636
637     size_t find(const key_type& key, size_t virtual_index) const
638     {
639         // Search from there until we find what we are looking for or an
640         // empty bucket
641         while (buckets_[virtual_index].has_value) {
642             const auto& value = buckets_[virtual_index];
643             if (equal(key, extract_(*value.memory()))) {
644                 return virtual_index;
645             }
646             virtual_index = index_add(virtual_index, 1);
647         }
648
649         // We found nothing, return end.

```

```
650         return bucket_count_;
651     }
652 };
653
654 } // namespace detail
655
656 // UNORDERED SET
657
658 #define BASE detail::kernel\<
659     Key,\
660     Key,\
661     Hash,\
662     extract::identity,\
663     extract::identity,\
664     KeyEqual,\
665     Allocator>
666 template <
667     class Key,
668     class Hash = std::hash<Key>,
669     class KeyEqual = std::equal_to<Key>,
670     class Allocator = std::allocator<Key>>
671 class unordered_set : public BASE
672 {};
673 #undef BASE
674
675 // UNORDERED MAP
676
677 #define BASE detail::kernel\<
678     std::pair<const Key, T>,\
679     Key,\
680     Hash,\
681     extract::first,\
682     extract::second,\
683     KeyEqual,\
684     Allocator>
685 template <
686     class Key,
687     class T,
688     class Hash = std::hash<Key>,
689     class KeyEqual = std::equal_to<Key>,
690     class Allocator = std::allocator<std::pair<const Key, T>>
691 class unordered_map : public BASE
692 {
693 public:
694     using mapped_type = T;
695 };
696 #undef BASE
697
698 } // namespace hopscotch
```

Appendix B

Test code

B.1 correctness_test.hpp

```
1 #include "double_tree.hpp"
2 #include "tabulation.hpp"
3 #include "hopscotch.hpp"
4 #include "linear.hpp"
5 #include "longrand.hpp"
6
7 #include <algorithm>
8 #include <cassert>
9 #include <iostream>
10 #include <stx/btree_map.h>
11
12 using std::cerr;
13 using std::cout;
14 using std::endl;
15 using std::flush;
16 using std::pair;
17
18 #define DOUBLE_BTREE
19 // #define SLOW_INSERT
20 // #define SLOW_ERASE
21
22 template <typename K, typename V, size_t nodesize>
23 class btree_traits
24 {
25 public:
26     static const bool selfverify = false;
27     static const bool debug = false;
28     static const int leafslots = nodesize / (sizeof(K) + sizeof(V));
29     static const int innerslots = nodesize / (sizeof(K) + sizeof(void*));
30     static const size_t binsearch_threshold = 256;
31 };
32
33 int main(int, char**)
34 {
35     const size_t count = 1000000;
36
37     #if defined(HOPSCOTCH)
38         hopscotch::unordered_map<uint64_t, uint64_t, tabulation<uint64_t>> map;
39     #elif defined(LINEAR)
```

```

40     linear::unordered_map<uint64_t, uint64_t, tabulation<uint64_t>> map;
41 #elif defined(DOUBLE_BTREE)
42     double_tree::map<uint64_t, uint64_t> map;
43 #endif
44
45     // Insertion test
46     srand(19);
47     for (size_t i = 0; i < count; ++i)
48     {
49         cerr << "\rinsert_" << i << "/" << count << flush;
50
51         uint64_t key = rand();
52         uint64_t val = rand();
53
54         map.insert({key, val});
55 #if !defined(DOUBLE_BTREE)
56         assert(map.size() == i + 1);
57 #endif
58
59 #if defined(SLOW_INSERT)
60         srand(19);
61         for (size_t j = 0; j < i + 1; ++j) {
62             uint64_t key = rand();
63             uint64_t val = rand();
64             if (map[key] != val) {
65                 cout << "while_inserting_" << key << endl;
66                 cout << i << ":_could_not_find_" << val << "(" << j <<
67                     ")_found_" << map[key] << endl;
68                 assert(false);
69             }
70         }
71 #endif // SLOW_INSERT
72     }
73
74     // Iterator test
75 #if defined(DOUBLE_BTREE) || defined(TRAD_BTREE_CACHE) ||
76     || defined(TRAD_BTREE_DISK)
77     int itnr = 0;
78     uint64_t prev;
79     for (auto it = map.begin(); it != map.end(); ++it) {
80         cerr << "\riterate_" << itnr << "/" << count << flush;
81         ++itnr;
82
83         if (it != map.begin()) {
84             assert(it->first > prev);
85         }
86         prev = it->first;
87     }
88 #endif
89
90     // Find test
91     srand(19);
92     for (size_t i = 0; i < count; ++i)
93     {
94         cerr << "\rfind_" << i << "/" << count << flush;
95
96         uint64_t key = rand();
97         uint64_t val = rand();
98
99         if (map[key] != val) {
100             cout << endl << "could_not_find_" << val << "." << endl
101                 << "found_" << map[key] << "." << endl;

```

```

102         assert(false);
103     }
104 }
105
106 srand(19);
107 for (size_t i = 0; i < count; ++i)
108 {
109     cerr << "\rerase_" << i << "/" << count << flush;
110
111     uint64_t key = rand();
112     rand();
113
114     map.erase(key);
115
116 #if defined(SLOW_ERASE)
117     srand(19);
118
119     for (size_t j = 0; j < i + 1; ++j) {
120         rand(); rand();
121     }
122     for (size_t j = i + 1; j < count; ++j) {
123         uint64_t key = rand();
124         uint64_t val = rand();
125         if (map[key] != val) {
126             cout << i << ":_could_not_find_" << key << "(" << j <<
127                 ")_found_" << map[key].first << endl;
128             assert(false);
129         }
130     }
131     for (size_t j = 0; j < i + 1; ++j) {
132         rand(); rand();
133     }
134 #endif // SLOW_ERASE
135 }
136
137     assert(map.empty());
138 }

```

B.2 hopscotch_experiment.hpp

```

1 #include "tabulation.hpp"
2 #include <ctime>
3 #include <iostream>
4 #include <random>
5 #include <vector>
6 using std::vector;
7
8 // #define TABULATION
9
10 class stripped_hopscotch
11 {
12 protected:
13     static const size_t neighborhood_size = 15;
14
15 public:
16     explicit stripped_hopscotch(size_t bucket_count):
17         bucket_count_{bucket_count},
18         size_{0}
19     {
20         buckets_.resize(bucket_count_);
21     }

```

```

22
23 float load_factor() const {
24     return (float)size_/(float)bucket_count_;
25 }
26
27 struct bucket_type
28 {
29     void has_value(bool has_value) {
30         hop_info[neighborhood_size] = has_value;
31     }
32
33     bool has_value() const {
34         return hop_info[neighborhood_size];
35     }
36
37     std::bitset<neighborhood_size+1> hop_info;
38 };
39
40 std::vector<bucket_type> buckets_;
41 size_t bucket_count_;
42 size_t size_;
43
44 size_t index_add(size_t index, size_t x) const {
45     return (index + x) & (bucket_count_ - 1);
46 }
47
48 size_t index_sub(size_t index, size_t x) const {
49     return (index - x) & (bucket_count_ - 1);
50 }
51
52 size_t next_hop(const bucket_type& bucket, int prev = -1) const
53 {
54     const size_t mask = 0xfffffffffff << (prev + 1);
55     const size_t hop_info = bucket.hop_info.to_ulong();
56     return __builtin_ffsl(hop_info & mask) - 1;
57 }
58
59 bool insert(size_t virtual_index)
60 {
61     virtual_index &= (bucket_count_ - 1);
62
63     // Find the nearest free bucket, wrapping if we move past the end.
64     size_t free_dist = 0;
65     size_t free_index = virtual_index;
66     while (buckets_[free_index].has_value()) {
67         free_dist += 1;
68         free_index = index_add(free_index, 1);
69     }
70
71     // Move buckets until we have a free bucket in the neighborhood of our
72     // virtual bucket.
73     while (free_dist > neighborhood_size - 1)
74     {
75         // Find a virtual bucket that has values stored in a bucket before
76         // the free bucket we found.
77         size_t virtual_move_dist = neighborhood_size - 1;
78         size_t virtual_move_index =
79             index_sub(free_index, virtual_move_dist);
80
81         size_t move_hop;
82
83         while (true)

```

```

84     {
85         auto& virtual_move_bucket = buckets_[virtual_move_index];
86         auto hop_info = virtual_move_bucket.hop_info.to_ulong();
87         move_hop = __builtin_ffsl(hop_info) - 1;
88
89         if (move_hop < virtual_move_dist) {
90             break;
91         } else {
92             // No luck, continue searching.
93             virtual_move_dist -= 1;
94             virtual_move_index = index_add(virtual_move_index, 1);
95
96             if (virtual_move_dist == 0)
97             {
98                 return false;
99             }
100        }
101    }
102
103    // Move.
104    const size_t move_dist = virtual_move_dist - move_hop;
105    const size_t move_index = index_add(virtual_move_index, move_hop);
106
107    buckets_[move_index].has_value(false);
108    buckets_[free_index].has_value(true);
109
110    auto& virtual_move = buckets_[virtual_move_index];
111    virtual_move.hop_info[move_hop] = false;
112    virtual_move.hop_info[virtual_move_dist] = true;
113
114    // The free bucket is now in the position of the moved bucket.
115    free_dist -= move_dist;
116    free_index = index_sub(free_index, move_dist);
117 }
118
119 // We should have a free bucket in the neighborhood now.
120 buckets_[free_index].has_value(true);
121 buckets_[virtual_index].hop_info[free_dist] = true;
122 ++size_;
123 return true;
124 }
125 };
126
127 int main(int, char**)
128 {
129     for (size_t exp = 8; exp <= 30; ++exp) {
130         const size_t count = static_cast<size_t>(1) << exp;
131         stripped_hopscotch hs{count};
132
133         #ifdef TABULATION
134             vector<uint64_t> elements(count);
135             for (size_t i = 0; i < count; ++i) { elements[i] = i; }
136             std::srand(std::time(nullptr));
137             std::random_shuffle(elements.begin(), elements.end());
138             tabulation<uint64_t> t;
139         #else
140             std::mt19937_64 mt(std::time(nullptr));
141         #endif
142
143         for (size_t i = 0; i < count; ++i) {
144             #ifdef TABULATION
145                 if (!hs.insert(t(elements[i]))) {

```



```

146 #else
147     if (!hs.insert(mt())) {
148 #endif
149         std::cout << "\r"
150             << exp << "\n"
151             << i << "/" << count << "\n"
152             << (float)i/(float)count << std::endl;
153         break;
154     } else {
155         for (int p = 0; p < 10; ++p) {
156             if (i == p*count/10) {
157                 std::cerr << "\r0." << p;
158             }
159         }
160     }
161 }
162 }
163 }

```

B.3 performance_clock.hpp

```

1 #pragma once
2
3 #include <chrono>
4 #include <cstdint>
5
6 namespace performance_clock
7 {
8     class interval
9     {
10    public:
11        void before();
12        void after();
13
14        uint64_t wall_time() const { return wall_time_; }
15        uint64_t usr_time() const { return usr_time_; }
16        uint64_t sys_time() const { return sys_time_; }
17        // uint64_t min_faults() const { return min_faults_; }
18        // uint64_t maj_faults() const { return maj_faults_; }
19
20    private:
21        using time_point =
22            std::chrono::time_point<std::chrono::high_resolution_clock>;
23
24        time_point wall_time_before_;
25        uint64_t usr_time_before_;
26        uint64_t sys_time_before_;
27        // uint64_t min_faults_before_;
28        // uint64_t maj_faults_before_;
29
30        uint64_t wall_time_;
31        uint64_t usr_time_;
32        uint64_t sys_time_;
33        // uint64_t min_faults_;
34        // uint64_t maj_faults_;
35    };
36 }

```

B.4 performance_clock.cpp

```

1 #include "performance_clock.hpp"
2
3 #include <sys/resource.h>
4
5 using std::chrono::time_point;
6 using std::chrono::high_resolution_clock;
7 using std::chrono::duration_cast;
8 using std::chrono::nanoseconds;
9
10 namespace performance_clock
11 {
12     void interval::before()
13     {
14         // Record user time, system time, page faults
15         rusage r_usage;
16         getrusage(RUSAGE_SELF, &r_usage);
17         usr_time_before_ =
18             r_usage.ru_utime.tv_sec * static_cast<uint64_t>(1000000000) +
19             r_usage.ru_utime.tv_usec * static_cast<uint64_t>(1000);
20         sys_time_before_ =
21             r_usage.ru_stime.tv_sec * static_cast<uint64_t>(1000000000) +
22             r_usage.ru_stime.tv_usec * static_cast<uint64_t>(1000);
23         // min_faults_before_ = r_usage.ru_minflt;
24         // maj_faults_before_ = r_usage.ru_majflt;
25
26         // Record wall time from std::chrono
27         wall_time_before_ = high_resolution_clock::now();
28     }
29
30     void interval::after()
31     {
32         // Record wall time from std::chrono
33         wall_time_ = duration_cast<nanoseconds>(
34             high_resolution_clock::now() - wall_time_before_).count();
35
36         // Record user time, system time, page faults
37         rusage r_usage;
38         getrusage(RUSAGE_SELF, &r_usage);
39         usr_time_ =
40             r_usage.ru_utime.tv_sec * static_cast<uint64_t>(1000000000) +
41             r_usage.ru_utime.tv_usec * static_cast<uint64_t>(1000)
42             - usr_time_before_;
43         sys_time_ =
44             r_usage.ru_stime.tv_sec * static_cast<uint64_t>(1000000000) +
45             r_usage.ru_stime.tv_usec * static_cast<uint64_t>(1000)
46             - sys_time_before_;
47         // min_faults_ = r_usage.ru_minflt - min_faults_before_;
48         // maj_faults_ = r_usage.ru_majflt - maj_faults_before_;
49     }
50 }

```

B.5 performance_test.hpp

```

1 #include "tabulation.hpp"
2
3 #include "hopsotch.hpp"
4 #include "linear.hpp"
5
6 #include "performance_clock.hpp"
7 #include "longrand.hpp"
8

```

```

 9 #include <algorithm>
10 #include <cassert>
11 #include <iomanip>
12 #include <iostream>
13 #include <vector>
14
15 #include <stx/btree_map.h>
16
17 using std::cerr;
18 using std::cout;
19 using std::endl;
20 using std::flush;
21 using std::pair;
22 using std::vector;
23
24 // #define DENSE
25 #define TRAD_BTREE_CACHE
26
27 template <typename K, typename V, size_t nodesize>
28 class btree_traits
29 {
30 public:
31     static const bool selfverify = false;
32     static const bool debug = false;
33     static const int leafslots = nodesize / (sizeof(K) + sizeof(V));
34     static const int innerslots = nodesize / (sizeof(K) + sizeof(void*));
35     static const size_t binsearch_threshold = 256;
36 };
37
38 int main(int, char**)
39 {
40     const size_t count = static_cast<size_t>(1)<<26;
41
42     srand(35);
43     vector<uint64_t> elements(count);
44
45 #ifdef DENSE
46     // Create a list of randomly ordered numbers from a dense interval (0..
47     // count)
48     for (size_t i = 0; i < count; ++i) { elements[i] = i; }
49     std::random_shuffle(elements.begin(), elements.end());
50 #else
51     // Create a list of random numbers
52     for (size_t i = 0; i < count; ++i) { elements[i] = longrand(); }
53 #endif
54
55     // Create a map with the chosen collision resolution method and hash
56     // function
57 #if defined(LINEAR)
58     linear::unordered_map<uint64_t, uint64_t, tabulation<uint64_t>> map;
59 #elif defined(HOPSCOTCH)
60     hopscotch::unordered_map<uint64_t, uint64_t, tabulation<uint64_t>> map;
61 #elif defined(DOUBLE_BTREE)
62     double_tree::map<uint64_t, uint64_t> map;
63 #elif defined(TRAD_BTREE_CACHE)
64     stx::btree_map<uint64_t, uint64_t, std::less<uint64_t>,
65     btree_traits<uint64_t, uint64_t, 256>> map;
66 #elif defined(TRAD_BTREE_DISK)
67     stx::btree_map<uint64_t, uint64_t, std::less<uint64_t>,
68     btree_traits<uint64_t, uint64_t, 4096>> map;
69 #endif

```

```

69     cout << std::fixed << std::setprecision(0);
70
71     const size_t round_count = static_cast<size_t>(1)<<18;
72     const size_t rounds = count/round_count;
73
74     for (size_t i = 0; i < rounds; ++i) {
75         // Insert
76         performance_clock::interval insert_interval;
77         insert_interval.before();
78         for (size_t j = 0; j < round_count; ++j) {
79             map.insert(std::make_pair(
80                 elements[i*round_count + j], i*round_count + j));
81         }
82         insert_interval.after();
83         cout << "insert\t" << i;
84         cout << "\t" << insert_interval.wall_time()/(double)round_count;
85         cout << "\t" << insert_interval.usr_time()/(double)round_count;
86         cout << "\t" << insert_interval.sys_time()/(double)round_count;
87         cout << endl;
88
89         // Find
90         performance_clock::interval search_interval;
91         search_interval.before();
92         size_t search_i = rand()%(i+1);
93         for (size_t j = 0; j < round_count; ++j) {
94             volatile auto _ = map[elements[search_i*round_count + j]];
95             (void)_; // Silence 'unused variable'
96         }
97         search_interval.after();
98         cout << "search\t" << i;
99         cout << "\t" << search_interval.wall_time()/(double)round_count;
100        cout << "\t" << search_interval.usr_time()/(double)round_count;
101        cout << "\t" << search_interval.sys_time()/(double)round_count;
102        cout << endl;
103
104        #if defined(DOUBLE_BTREE) || defined(TRAD_BTREE_CACHE) \\  

105            || defined(TRAD_BTREE_DISK)
106            // Iterate
107            performance_clock::interval iterate_interval;
108            iterate_interval.before();
109            size_t iterate_i = rand()%(i+1);
110            size_t iterate_j = rand()%(round_count);
111            auto it = map.find(map[elements[iterate_i*round_count + iterate_j]]);
112            for (size_t j = 0; j < round_count; ++j) {
113                volatile auto _ = *it;
114                (void)_; // Silence 'unused variable'
115                ++it;
116                if (it == map.end()) {
117                    it = map.begin();
118                }
119            }
120            iterate_interval.after();
121            cout << "iterate\t" << i;
122            cout << "\t" << iterate_interval.wall_time()/(double)round_count;
123            cout << "\t" << iterate_interval.usr_time()/(double)round_count;
124            cout << "\t" << iterate_interval.sys_time()/(double)round_count;
125            cout << endl;
126        #endif
127    }
128
129    // Shuffle the elements for erasing
130    std::random_shuffle(elements.begin(), elements.end());

```

```
131
132     for (size_t i = 0; i < rounds; ++i) {
133         // Erase
134         performance_clock::interval erase_interval;
135         erase_interval.before();
136         for (size_t j = 0; j < round_count; ++j) {
137             map.erase(elements[i*round_count + j]);
138         }
139         erase_interval.after();
140         cout << "erase\t" << i;
141         cout << "\t" << erase_interval.wall_time()/(double)round_count;
142         cout << "\t" << erase_interval.user_time()/(double)round_count;
143         cout << "\t" << erase_interval.sys_time()/(double)round_count;
144         cout << endl;
145     }
146 }
```