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

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 were 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 seach), 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 searcing 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 steps 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 hade (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 it 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 $\{0, \ldots, n-1\}$, 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 $\{0, \ldots, n-1\}$.

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.

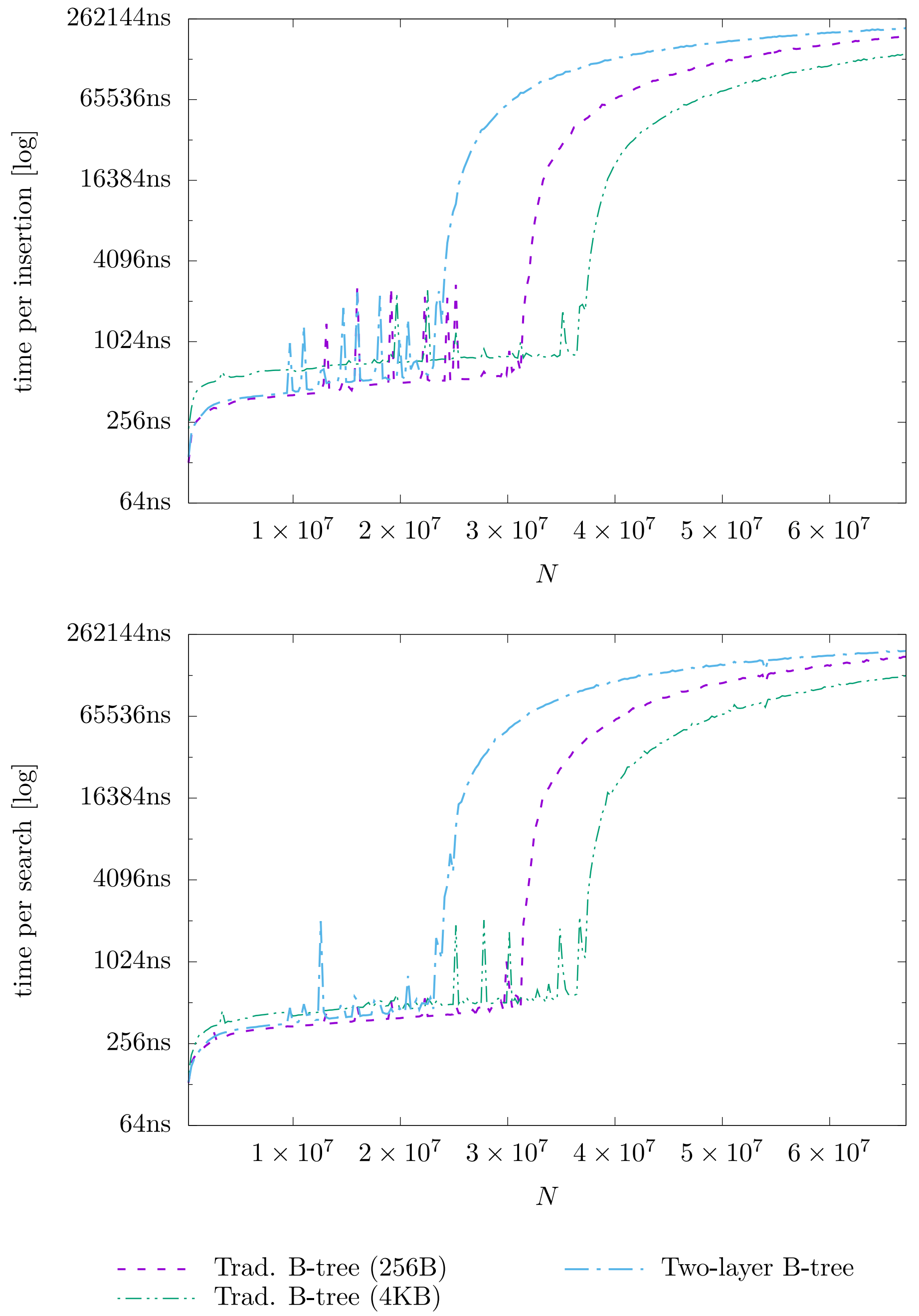

Figure 2.1: The results of the experiments on tree-based map performance. Top graph shows time per insertion, bottom graph shows time per search, both as a function of m. Note that time is plotted on a logarithmic scale.

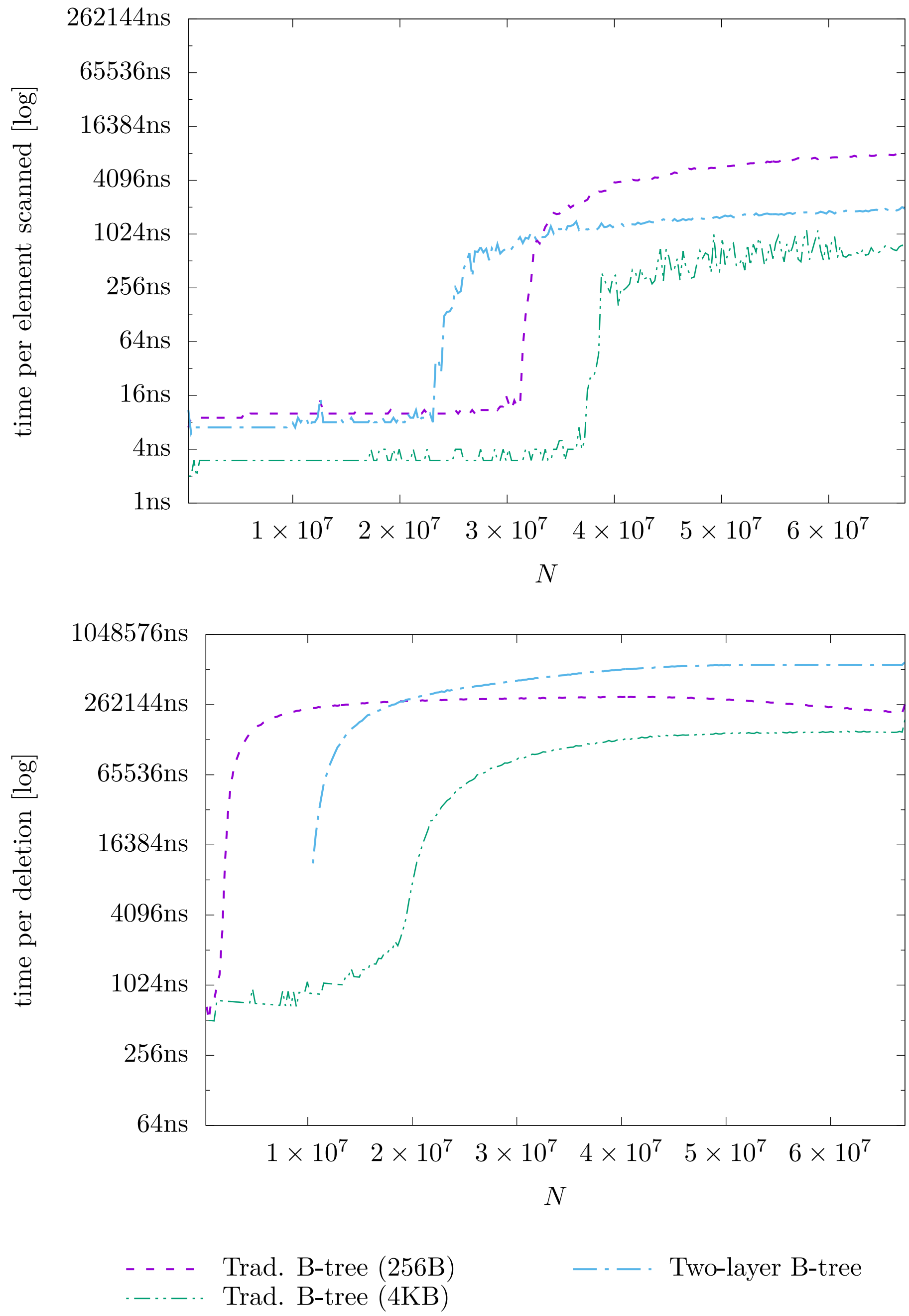

Figure 2.2: The results of the experiments on tree-based map performance. Top graph shows time per element scanned, bottom graph shows time per deletion, both as a function of m. Note that time is plotted on a logarithmic scale.

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 entiry 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 $h : U \to M$, where U is the universe of possible keys to insert and $M = \{0, 1, \ldots, m-1\}$. 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 $x_1, \ldots, x_k \in U$, $h(x_1), \ldots, h(x_k)$ 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, $x_1, \ldots, x_c$, and using those characters as lookups into c tables $T_1, \ldots, T_c$ 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, $h(x) = T_1[x_1] \oplus \cdots \oplus T_c[x_c]$.

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 $\{p, p+1\ldots, p+H-1\}$ 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 adressing 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 occuring.

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 lenght 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

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

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

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

$$\frac{m}{m^n}\sum_{k=H+1}^{n}O((n/k)^k)(k-H)^kO(m^{n-k})$$

$$= m\sum_{k=H+1}^{n}O((n/k)^{k+H})(k-H)^kO(m^{-k})$$

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

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:

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

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.

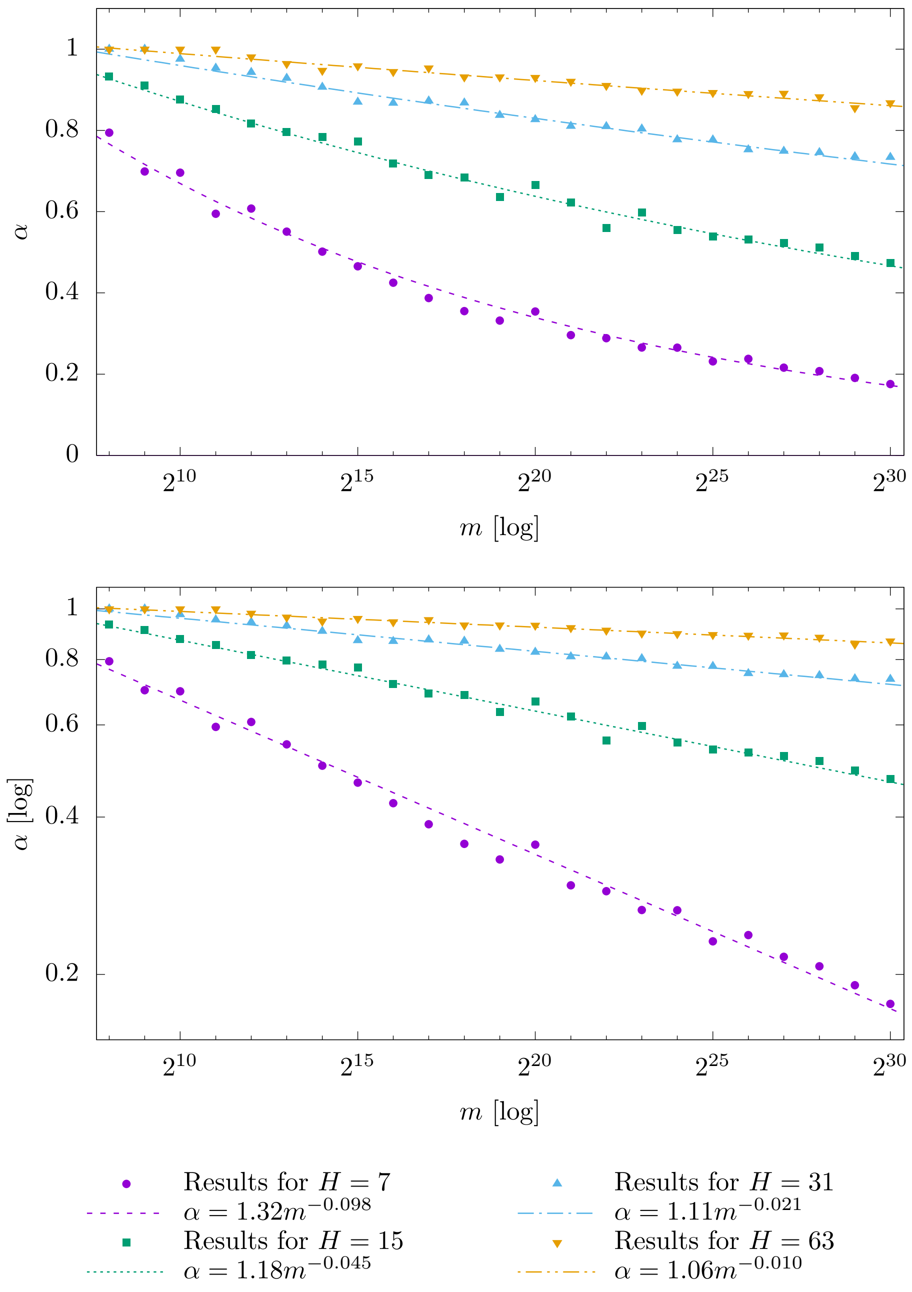

Figure 3.1: The results of the experiments on forced resize in hopscotch hashing. The upper graphs has the result plotted on a logarithmic horizontal axis and linear vertical axis. The lower graph has both axes logarithmic (see discussion).

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:

```
return t1[static_cast<uint8_t>(x)] ^
       t2[static_cast<uint8_t>(x>>8)] ^
       t3[static_cast<uint8_t>(x>>16)] ^
       t4[static_cast<uint8_t>(x>>24)] ^
       t5[static_cast<uint8_t>(x>>32)] ^
       t6[static_cast<uint8_t>(x>>40)] ^
       t7[static_cast<uint8_t>(x>>48)] ^
       t8[static_cast<uint8_t>(x>>56)];
```

Here `x` is the input as before `t1`, ..., `t8` 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 $\{0, \ldots, n-1\}$, 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-tres, 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 gaps 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.

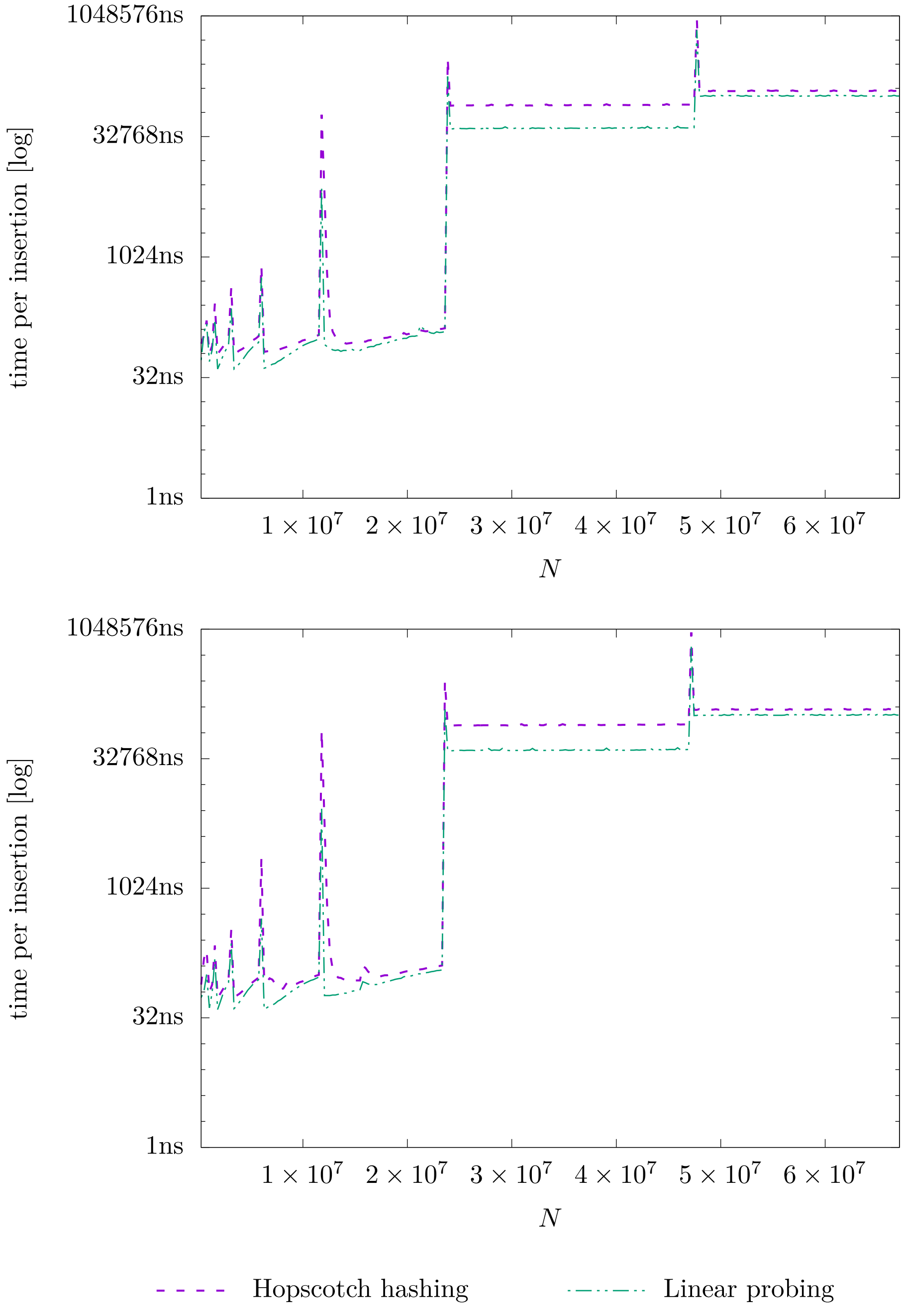

Figure 3.2: The results of the experiments on hash-based map performance. Top graph shows time per insertion with the random data-set, bottom graph shows same for the dense data-set, both as a function of m. Note that time is plotted on a logarithmic scale.

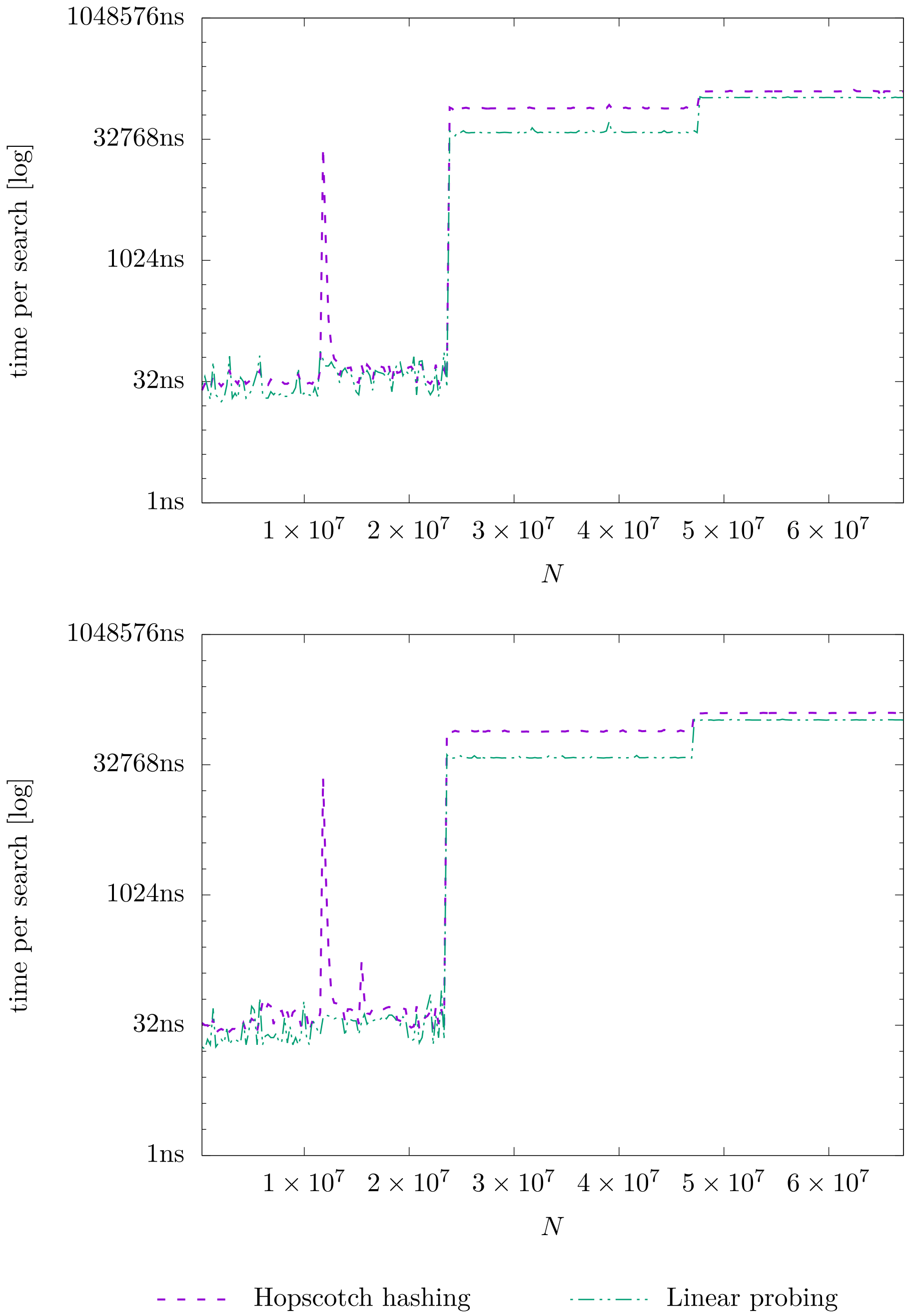

Figure 3.3: The results of the experiments on hash-based map performance. Top graph shows time per search with the random data-set, bottom graph shows same for the dense data-set, both as a function of m. Note that time is plotted on a logarithmic scale.

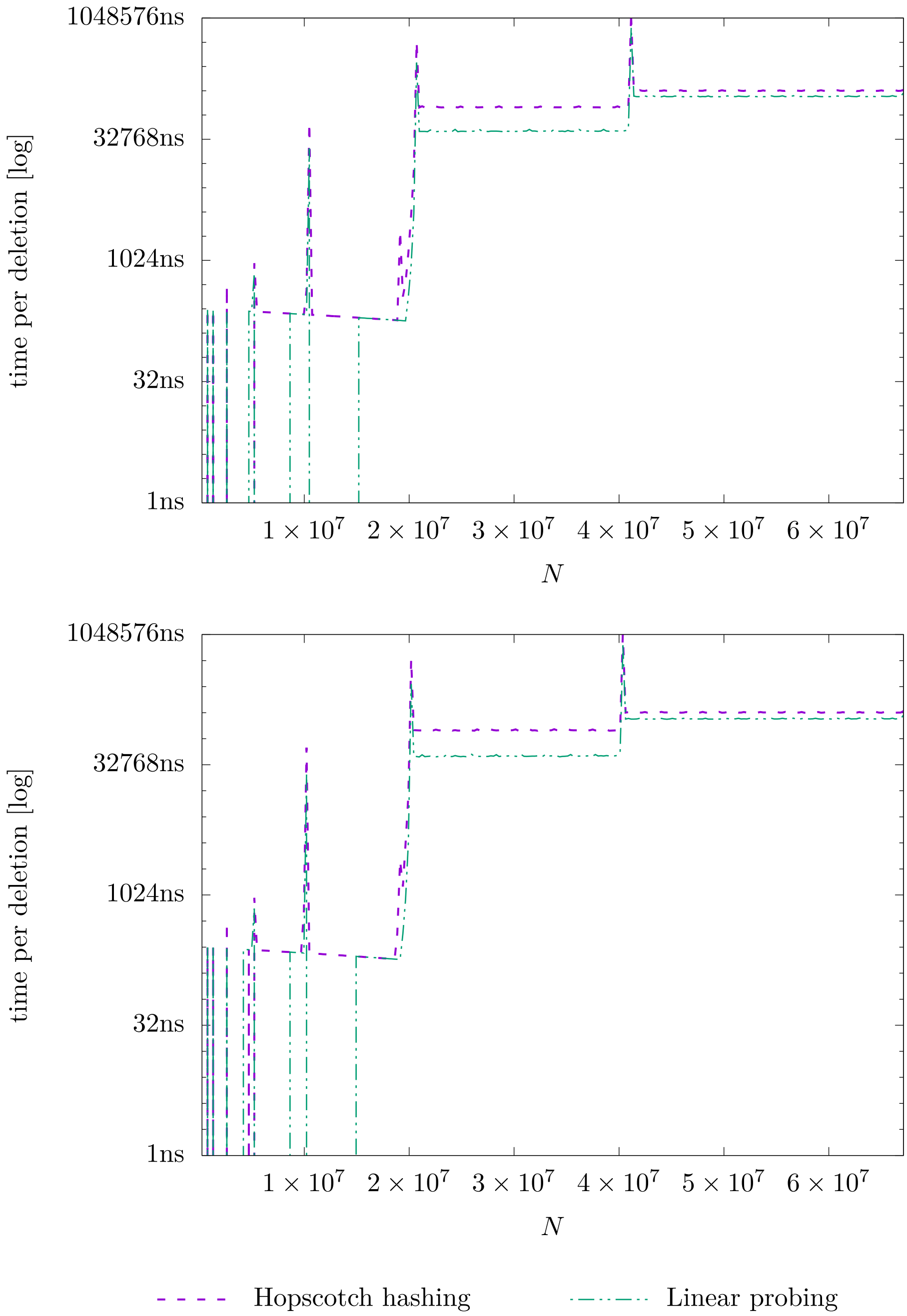

Figure 3.4: The results of the experiments on hash-based map performance. Top graph shows time per deletion with the random data-set, bottom graph shows same for the dense data-set, both as a function of m. Note that time is plotted on a logarithmic scale.

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 taylored 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. Pătrașcu 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. Pătrașcu 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

```
#pragma once
#include "double_tree_page_node.hpp"
#include <iterator>
#include <vector>

namespace double_tree
{

namespace detail
{

struct tree_position
{
    tree_position() = default;

    tree_position(const tree_position& other):
        page{other.page},
        sub_position{other.sub_position}
    {}

    tree_position(void* page_, page_position sub_position_):
        page{page_},
        sub_position{sub_position_}
    {}

    bool operator==(const tree_position& other) const
    {
        return page == other.page && sub_position == other.sub_position;
    }

    bool operator!=(const tree_position& other) const
    {
        return page != other.page || sub_position != other.sub_position;
    }

    void* page;
    page_position sub_position;
};

```

```
template <
    typename Element,
    typename ElementWrite,
    typename Key,
    typename KeyExtract,
    typename ValExtract>
struct kernel
{
private:
    // Defined below.
    template <typename T> class iterator_template;

    // Auxiliary structures for the page nodes.  A stem node does not need any
    // extra data, while the leaf nodes of a tree are linked together in a
    // linked list, so they need pointers to their previous and next nodes
    struct stem_aux
    {};

    struct leaf_aux
    {
        void* prev_pointer;
        void* next_pointer;
    };

    using stem_node = PageNode<
        std::pair<const Key, void*>,
        std::pair<Key, void*>,
        Key,
        extract::first,
        extract::second,
        stem_aux>;

    using leaf_node = PageNode<
        Element,
        ElementWrite,
        Key,
        KeyExtract,
        ValExtract,
        leaf_aux>;

    // DATA

    void* root_pointer;
    leaf_node* min_leaf_pointer;
    leaf_node* max_leaf_pointer;
    int stem_levels;
    KeyExtract key_extract_;
    ValExtract val_extract_;

public:
    // MEMBER TYPES

    using key_type        = Key;
    using value_type      = Element;
    using reference       = value_type&;
    using const_reference = const value_type&;
    using pointer         = value_type*;
    using const_pointer   = const value_type*;
    using iterator        = iterator_template<value_type>;
    using const_iterator  = iterator_template<const value_type>;

    // CONSTRUCTOR
```

```
    kernel()
    : root_pointer{new leaf_node},
      min_leaf_pointer{get_leaf_pointer(root_pointer)},
      max_leaf_pointer{get_leaf_pointer(root_pointer)},
      stem_levels{0}
    {
        auto& root = get_leaf(root_pointer);
        root.aux.prev_pointer = nullptr;
        root.aux.next_pointer = nullptr;
    }

    // ACCESSORS

public:
    auto& operator[](const Key& find_key) {
        auto position = find_implementation(find_key);
        return val_extract_(
            get_leaf(position.page).elem(position.sub_position));
    }
    const auto& operator[](const Key& find_key) const {
        auto position = find_implementation(find_key);
        return val_extract_(
            get_leaf(position.page).elem(position.sub_position));
    }

private:
    Element& elem(const tree_position position) {
        return get_leaf(position.page).elem(position.sub_position);
    }
    const Element& elem(const tree_position position) const {
        return get_leaf(position.page).elem(position.sub_position);
    }

    static stem_node* get_stem_pointer(void* const pointer) {
        return static_cast<stem_node*>(pointer);
    }
    static leaf_node* get_leaf_pointer(void* const pointer) {
        return static_cast<leaf_node*>(pointer);
    }

    static stem_node& get_stem(void* const pointer) {
        return *get_stem_pointer(pointer);
    }
    static leaf_node& get_leaf(void* const pointer) {
        return *get_leaf_pointer(pointer);
    }

    static const stem_node& get_stem(const void* const pointer) {
        return *static_cast<const stem_node*>(pointer);
    }
    static const leaf_node& get_leaf(const void* const pointer) {
        return *static_cast<const leaf_node*>(pointer);
    }

    // PREDICATES
public:
    bool empty() const
    {
        return stem_levels == 0 && get_leaf(root_pointer).empty();
    }

```

```
    // OPERATIONS
public:
    iterator find(const Key& find_key) {
        return {this, find_implementation(find_key)};
    }

    const_iterator find(const Key& find_key) const {
        return {this, find_implementation(find_key)};
    }

private:
    tree_position find_implementation(const Key& find_key) const
    {
        auto search_pointer = root_pointer;
        for (int depth = 0; depth < stem_levels; ++depth)
        {
            const auto& search_stem = get_stem(search_pointer);
            search_pointer =
                search_stem.elem(search_stem.find(find_key)).second;
        }
        const auto& leaf = get_leaf(search_pointer);
        return {search_pointer, leaf.find(find_key)};
    }

private:
    using Path = std::vector<tree_position>;

    Path find_path(const Key& find_key) const
    {
        Path result(stem_levels + 1);
        auto search_pointer = root_pointer;
        for (int depth = 0; depth < stem_levels; ++depth)
        {
            const auto& search_stem = get_stem(search_pointer);
            result[depth].page = search_pointer;
            result[depth].sub_position = search_stem.find(find_key);
            search_pointer =
                search_stem.elem(result[depth].sub_position).second;
        }
        const auto& search_leaf = get_leaf(search_pointer);
        result[stem_levels].page = search_pointer;
        result[stem_levels].sub_position = search_leaf.find(find_key);
        return result;
    }

    void split_root()
    {
        if (stem_levels > 0)
        {
            const auto old_root_pointer = root_pointer;
            auto& old_root = get_stem(old_root_pointer);

            if (old_root.oversized())
            {
                const auto new_pointer = old_root.split_one_leaf();
                auto& new_stem = get_stem(new_pointer);

                while (old_root.oversized())
                {
                    new_stem.borrow_prev(old_root);
                }

```

```
                root_pointer = new stem_node;
                auto& new_root = get_stem(root_pointer);

                new_root.insert({old_root.min_key(), old_root_pointer});
                new_root.insert({new_stem.min_key(), new_pointer});

                ++stem_levels;
            }
        }
        else
        {
            const auto old_root_pointer = root_pointer;
            auto& old_root = get_leaf(old_root_pointer);

            if (old_root.oversized())
            {
                const auto new_pointer = old_root.split_one_leaf();
                auto& new_leaf = get_leaf(new_pointer);

                while (old_root.oversized()) {
                    new_leaf.borrow_prev(old_root);
                }

                old_root.aux.next_pointer = new_pointer;
                new_leaf.aux.prev_pointer = old_root_pointer;
                max_leaf_pointer = new_pointer;

                root_pointer = new stem_node;
                auto& new_root = get_stem(root_pointer);

                new_root.insert({old_root.min_key(), old_root_pointer});
                new_root.insert({new_leaf.min_key(), new_pointer});

                ++stem_levels;
            }
        }
    }

public:
    void insert(const Element& new_elem)
    {
        split_root();

        const auto& new_key = key_extract_(new_elem);

        auto current_pointer = root_pointer;
        for (int depth = 0; depth < stem_levels - 1; ++depth)
        {
            auto& current_stem = get_stem(current_pointer);

            const auto target_pos = current_stem.find(new_key);
            const auto target_pointer = current_stem.elem(target_pos).second;
            auto& target_stem = get_stem(target_pointer);

            // Offload to previous sibling?
            if (target_stem.oversized() &&
                target_pos != current_stem.min_position())
            {
                const auto prev_pos = current_stem.prev_position(target_pos);
                const auto prev_pointer = current_stem.elem(prev_pos).second;
                auto& prev_stem = get_stem(prev_pointer);

```

```
                if (prev_stem.small()) {
                    while (target_stem.oversized()) {
                        prev_stem.borrow_next(target_stem);
                    }

                    current_stem.set_key(target_pos, target_stem.min_key());

                    if (new_key < target_stem.min_key()) {
                        if (new_key < prev_stem.min_key()) {
                            current_stem.set_key(prev_pos, new_key);
                        }
                        current_pointer = prev_pointer;
                    } else {
                        current_pointer = target_pointer;
                    }
                    continue;
                }
            }

            // Offload to next sibling?
            if (target_stem.oversized() &&
                target_pos != current_stem.max_position())
            {
                const auto next_pos = current_stem.next_position(target_pos);
                const auto next_pointer = current_stem.elem(next_pos).second;
                auto& next_stem = get_stem(next_pointer);

                if (next_stem.small()) {
                    while (target_stem.oversized()) {
                        next_stem.borrow_prev(target_stem);
                    }

                    current_stem.set_key(next_pos, next_stem.min_key());

                    if (new_key >= next_stem.min_key()) {
                        current_pointer = next_pointer;
                    } else {
                        if (new_key < target_stem.min_key()) {
                            current_stem.set_key(target_pos, new_key);
                        }
                        current_pointer = target_pointer;
                    }
                    continue;
                }
            }

            if (target_stem.oversized())
            {
                // Offload to new next sibling?
                const auto new_pointer = target_stem.split_one_leaf();
                auto& new_stem = get_stem(new_pointer);

                while (target_stem.oversized()) {
                    new_stem.borrow_prev(target_stem);
                }

                current_stem.insert({new_stem.min_key(), new_pointer});

                if (new_key >= new_stem.min_key()) {
                    current_pointer = new_pointer;
                } else {
                    if (new_key < target_stem.min_key()) {
```

```
                    current_stem.set_key(target_pos, new_key);
                }
                current_pointer = target_pointer;
            }
            continue;
        }

        if (new_key < target_stem.min_key()) {
            current_stem.set_key(target_pos, new_key);
        }
        current_pointer = target_pointer;
    }

    if (stem_levels > 0)
    {
        auto& current_stem = get_stem(current_pointer);

        const auto target_pos = current_stem.find(new_key);
        const auto target_pointer = current_stem.elem(target_pos).second;
        auto& target_leaf = get_leaf(target_pointer);

        // Offload to previous sibling?
        if (target_leaf.oversized() &&
            target_pos != current_stem.min_position())
        {
            const auto prev_pos = current_stem.prev_position(target_pos);
            const auto prev_pointer = current_stem.elem(prev_pos).second;
            auto& prev_leaf = get_leaf(prev_pointer);

            if (prev_leaf.small()) {
                while (target_leaf.oversized()) {
                    prev_leaf.borrow_next(target_leaf);
                }

                current_stem.set_key(target_pos, target_leaf.min_key());

                if (new_key < target_leaf.min_key()) {
                    if (new_key < prev_leaf.min_key()) {
                        current_stem.set_key(prev_pos, new_key);
                    }
                    prev_leaf.insert(new_elem);
                } else {
                    target_leaf.insert(new_elem);
                }
                return;
            }
        }

        // Offload to next sibling?
        if (target_leaf.oversized() &&
            target_pos != current_stem.max_position())
        {
            const auto next_pos = current_stem.next_position(target_pos);
            const auto next_pointer = current_stem.elem(next_pos).second;
            auto& next_leaf = get_leaf(next_pointer);

            if (next_leaf.small()) {
                while (target_leaf.oversized()) {
                    next_leaf.borrow_prev(target_leaf);
                }

                current_stem.set_key(next_pos, next_leaf.min_key());
```

```
                    if (new_key >= next_leaf.min_key()) {
                        next_leaf.insert(new_elem);
                    } else {
                        if (new_key < target_leaf.min_key()) {
                            current_stem.set_key(target_pos, new_key);
                        }
                        target_leaf.insert(new_elem);
                    }
                    return;
                }
            }

            // Offload to new next sibling?
            if (target_leaf.oversized())
            {
                const auto new_pointer = target_leaf.split_one_leaf();
                auto& new_leaf = get_leaf(new_pointer);

                while (target_leaf.oversized()) {
                    new_leaf.borrow_prev(target_leaf);
                }

                current_stem.insert({new_leaf.min_key(), new_pointer});

                if (target_leaf.aux.next_pointer != nullptr) {
                    get_leaf(target_leaf.aux.next_pointer).aux.prev_pointer =
                        new_pointer;
                }
                new_leaf.aux.prev_pointer = target_pointer;
                new_leaf.aux.next_pointer = target_leaf.aux.next_pointer;
                target_leaf.aux.next_pointer = new_pointer;

                if (max_leaf_pointer == target_pointer) {
                    max_leaf_pointer = new_pointer;
                }

                if (new_key >= new_leaf.min_key()) {
                    new_leaf.insert(new_elem);
                } else {
                    if (new_key < target_leaf.min_key()) {
                        current_stem.set_key(target_pos, new_key);
                    }
                    target_leaf.insert(new_elem);
                }
                return;
            }

            if (new_key < target_leaf.min_key()) {
                current_stem.set_key(target_pos, new_key);
            }
            target_leaf.insert(new_elem);
        }
        else
        {
            auto& current_leaf = get_leaf(current_pointer);
            current_leaf.insert(new_elem);
        }
    }

private:
    void insert_node_after(const Path& path, const int depth,
```

```
        const Key prev_min_key, void* const prev_pointer,
        const Key new_min_key, void* const new_pointer)
    {
        if (depth > 0)
        {
            const auto insert_pointer = path[depth - 1].page;
            auto& insert_stem = get_stem(insert_pointer);

            const auto split_pointer =
                insert_stem.insert(new_min_key, new_pointer);
            if (split_pointer)
            {
                auto& split_stem = *split_pointer;
                insert_node_after(path, depth - 1,
                    insert_stem.min_key(), insert_pointer,
                    split_stem.min_key(), split_pointer);
            }
        }
        else
        {
            root_pointer = new stem_node;
            auto& root = get_stem(root_pointer);

            root.insert(prev_min_key, prev_pointer);
            root.insert(new_min_key, new_pointer);

            ++stem_levels;
        }
    }

public:
    void erase(const Key& erase_key)
    {
        const auto path = find_path(erase_key);

        const auto erase_pointer = path[stem_levels].page;
        auto& erase_leaf = get_leaf(erase_pointer);

        const auto was_large = erase_leaf.large();
        // const auto old_key = erase_leaf.min_key();

        // Erase the element
        erase_leaf.erase(erase_key);
        // If this is the root we are done
        if (stem_levels == 0) { return; }

        auto parent_pos = path[stem_levels - 1].sub_position;
        auto& parent_stem = get_stem(path[stem_levels - 1].page);
        const auto parent_was_large = parent_stem.large();
        const auto old_key = parent_stem.key(parent_pos);

        // If the node is now empty
        if (erase_leaf.empty()) {
            delete &erase_leaf;
            parent_stem.erase(old_key);
        // Otherwise we must maintain the invariants
        } else {
            const auto prev_ptr = parent_pos == parent_stem.min_position() ?
                nullptr : static_cast<leaf_node*>(parent_stem.elem(
                        parent_stem.prev_position(parent_pos)).second);
            const auto prev_key = prev_ptr ? prev_ptr->min_key() : Key{};

```

```
            const auto next_ptr = parent_pos == parent_stem.max_position() ?
                nullptr : static_cast<leaf_node*>(parent_stem.elem(
                        parent_stem.next_position(parent_pos)).second);
            const auto next_key = next_ptr ? next_ptr->min_key() : Key{};

            bool did_grow = false;

            // If a large node turned small we must grow it large again
            if (was_large && erase_leaf.small()) {
                if (prev_ptr && prev_ptr->small()) {
                    while (erase_leaf.small() && !prev_ptr->empty()) {
                        erase_leaf.borrow_prev(*prev_ptr);
                    }
                }

                if (next_ptr && next_ptr->small()) {
                    while (erase_leaf.small() && !next_ptr->empty()) {
                        erase_leaf.borrow_next(*next_ptr);
                    }
                }

                did_grow = true;
            }

            if (prev_ptr && prev_ptr->empty()) {
                if (prev_ptr->aux.prev_pointer) {
                    get_leaf(prev_ptr->aux.prev_pointer).aux.next_pointer =
                        erase_pointer;
                }
                erase_leaf.aux.prev_pointer = prev_ptr->aux.prev_pointer;

                if (min_leaf_pointer == prev_ptr) {
                    min_leaf_pointer = &get_leaf(erase_pointer);
                }

                delete prev_ptr;
                parent_stem.erase(prev_key);
            }

            if (next_ptr && next_ptr->empty()) {
                if (next_ptr->aux.next_pointer) {
                    get_leaf(next_ptr->aux.next_pointer).aux.prev_pointer =
                        erase_pointer;
                }
                erase_leaf.aux.next_pointer = next_ptr->aux.next_pointer;

                if (max_leaf_pointer == next_ptr) {
                    max_leaf_pointer = &get_leaf(erase_pointer);
                }

                delete next_ptr;
                parent_stem.erase(next_key);
            }
            else if (next_ptr && next_ptr->min_key() != next_key) {
                parent_stem.set_key(
                    parent_stem.find(next_key), next_ptr->min_key());
            }

            if (erase_leaf.min_key() != old_key) {
                parent_stem.set_key(
                    parent_stem.find(old_key), erase_leaf.min_key());
            }
```

```
        }

        erase_helper(path, stem_levels - 1, parent_was_large);
    }

private:
    void root_collapse()
    {
        auto& root = get_stem(root_pointer);

        // If we have only one child we should collapse this level of th tree
        if (root.stem_levels == 0 &&
            root.get_leaf(root.min_leaf_index).count() == 1)
        {
            const auto old_root = root_pointer;
            root_pointer = root.elem({root.min_leaf_index, 0}).second;
            delete get_stem_pointer(old_root);
            --stem_levels;

            if (stem_levels > 0) root_collapse();
        }
    }

    void erase_helper(const Path& path, const int depth, bool was_large)
    {
        // If we are the root
        if (depth == 0) {
            root_collapse();
            return;
        }

        const auto erase_pointer = path[depth].page;
        auto& erase_stem = get_stem(erase_pointer);

        // const auto old_key = erase_stem.min_key();

        auto& parent_stem = get_stem(path[depth - 1].page);
        auto parent_pos = path[depth - 1].sub_position;
        const auto parent_was_large = parent_stem.large();
        const auto old_key = parent_stem.key(parent_pos);

        // If the node is now empty
        if (erase_stem.empty()) {
            delete &erase_stem;
            parent_stem.erase(old_key);
        // Otherwise we must maintain the invariants
        } else {
            const auto prev_ptr = parent_pos == parent_stem.min_position() ?
                nullptr : static_cast<stem_node*>(parent_stem.elem(
                        parent_stem.prev_position(parent_pos)).second);
            const auto prev_key = prev_ptr ? prev_ptr->min_key() : Key{};

            const auto next_ptr = parent_pos == parent_stem.max_position() ?
                nullptr : static_cast<stem_node*>(parent_stem.elem(
                        parent_stem.next_position(parent_pos)).second);
            const auto next_key = next_ptr ? next_ptr->min_key() : Key{};

            // If a large node turned small we must grow it large again
            if (was_large && erase_stem.small()) {
                if (prev_ptr && prev_ptr->small()) {
                    while (erase_stem.small() && !prev_ptr->empty()) {
                        erase_stem.borrow_prev(*prev_ptr);
```

```
                    }
                }

                if (next_ptr && next_ptr->small()) {
                    while (erase_stem.small() && !next_ptr->empty()) {
                        erase_stem.borrow_next(*next_ptr);
                    }
                }
            }

            if (prev_ptr && prev_ptr->empty()) {
                delete prev_ptr;
                parent_stem.erase(prev_key);
            }

            if (next_ptr && next_ptr->empty()) {
                delete next_ptr;
                parent_stem.erase(next_key);
            }
            else if (next_ptr && next_ptr->min_key() != next_key) {
                parent_stem.set_key(
                    parent_stem.find(next_key), next_ptr->min_key());
            }

            if (erase_stem.min_key() != old_key) {
                parent_stem.set_key(
                    parent_stem.find(old_key), erase_stem.min_key());
            }
        }

        erase_helper(path, depth - 1, parent_was_large);
    }

public:
    // ITERATOR GETTERS

    iterator begin() {
        return make_iterator({
            min_leaf_pointer,
            min_leaf_pointer->min_position()});
    }

    const_iterator begin() const {
        return make_const_iterator({
            min_leaf_pointer,
            min_leaf_pointer->min_position()});
    }

    const_iterator cbegin() const {
        return make_const_iterator({
            min_leaf_pointer,
            min_leaf_pointer->min_position()});
    }

    iterator end() {
        return make_iterator({
            max_leaf_pointer,
            max_leaf_pointer->end_position()});
    }

    const_iterator end() const {
        return make_const_iterator({
```

```
            max_leaf_pointer,
            max_leaf_pointer->end_position()});
    }

    const_iterator cend() const {
        return make_const_iterator({
            max_leaf_pointer,
            max_leaf_pointer->end_position()});
    }

private:
    iterator make_iterator(tree_position position) {
        return {this, position};
    }

    const_iterator make_const_iterator(tree_position position) const {
        return {this, position};
    }

    // ITERATOR TYPE

    template <typename T>
    class iterator_template : std::iterator<std::forward_iterator_tag, T>
    {
        friend kernel;

    public:
        iterator_template()
        : tree_(nullptr),
          position_()
        {}

        iterator_template(const iterator_template& other)
        : tree_(other.tree_),
          position_(other.position_)
        {}

        iterator_template& operator=(const iterator_template& other)
        {
            tree_ = other.tree_;
            position_ = other.position_;
        }

        ~iterator_template()
        {}

        reference operator*() {
            return tree_->elem(position_);
        }

        const_reference operator*() const {
            return tree_->elem(position_);
        }

        pointer operator->() {
            return &tree_->elem(position_);
        }

        const_pointer operator->() const {
            return &tree_->elem(position_);
        }

```

```
        iterator_template& operator++()
        {
            const auto& leaf = get_leaf(position_.page);
            if (leaf.aux.next_pointer != nullptr &&
                position_.sub_position == leaf.max_position())
            {
                position_.page = leaf.aux.next_pointer;
                const auto& next = get_leaf(position_.page);
                position_.sub_position = next.min_position();
            }
            else
            {
                position_.sub_position =
                    leaf.next_position(position_.sub_position);
            }
            return *this;
        }

        iterator_template operator++(int)
        {
            iterator_template old(*this);
            ++*this;
            return old;
        }

        bool operator==(const iterator_template& other) const {
            return tree_ == other.tree_ && position_ == other.position_;
        }

        bool operator!=(const iterator_template& other) const {
            return tree_ != other.tree_ || position_ != other.position_;
        }

    private:
        iterator_template(
            kernel* tree,
            tree_position position)
        : tree_(tree),
          position_(position)
        {}

        kernel* tree_;
        tree_position position_;
    };

public:
    void print() const
    {
        std::cout << "------------" << std::endl;
        print_node(root_pointer, 0);
    }

private:
    void print_node(void* pointer, int depth) const
    {
        if (depth < stem_levels) {
            const stem_node& stem = get_stem(pointer);
            std::cout << "treestem␣(" << depth << ",␣"
                << (int)stem.get_leaf(stem.min_leaf_index).count()
                << ")␣" << std::endl;
            std::cout << "--" << std::endl;
            stem.print();
```

```
            std::cout << "--" << std::endl;
            std::cout << std::endl;
            for (auto p = stem.min_position(); p != stem.end_position(); p =
                stem.next_position(p))
            {
                print_node(stem.elem(p).second, depth + 1);
            }
        } else {
            const leaf_node& leaf = get_leaf(pointer);
            std::cout << "treeleaf␣(" << depth << ")␣" << std::endl;
            std::cout << "--" << std::endl;
            leaf.print();
            std::cout << "--" << std::endl;
            std::cout << std::endl;
        }
    }
};

} // namespace detail

// SET

#define BASE detail::kernel<\
    Key,\
    Key,\
    Key,\
    extract::identity,\
    extract::identity>
template <
    class Key>
class set : public BASE
{};
#undef BASE

// MAP

#define BASE detail::kernel<\
    std::pair<const Key, T>,\
    std::pair<Key, T>,\
    Key,\
    extract::first,\
    extract::second>
template <
    class Key,
    class T>
class map : public BASE
{
public:
    using mapped_type = T;
};
#undef BASE

} // namespace double_tree
```

A.2 double_tree_line_node.hpp

```
// An array based node that fits in a cache line

#pragma once
#include <array>
#include <cstdint>
```

```
#include <iostream>
#include <limits>

namespace double_tree
{

namespace detail
{

// Define here the size in bytes of line nodes, as well as the types used to
// index inside them.  A line node index should be able to index all values in
// an array of key-value pairs of size equal to the line node.  See below for
// the exact requirements, which are a little less because of the size used up
// for bookkeeping data in each case

constexpr size_t line_node_size = 256;
using line_index = uint8_t;
constexpr line_index line_index_nil = std::numeric_limits<line_index>::max();

template <
    typename Element,
    typename ElementWrite,
    typename Key,
    typename KeyExtract,
    typename ValExtract,
    typename Aux>
struct alignas(line_node_size) line_node
{
public:
    // CONSTANTS

    static constexpr int max_count =
        (line_node_size - sizeof(line_index) - sizeof(Aux)) / sizeof(Element);
    static constexpr int min_count = max_count/2;

    // DATA

private:
    std::array<Element, max_count> elems_;
    line_index count_;
    KeyExtract key_extract_;
public:
    Aux aux;

    // ACCESSORS

public:
    line_index count() const { return count_; }

    void reset() { count_ = 0; }

    const Key& key(const line_index index) const {
        return key_extract_(elems_[index]);
    }

    Element& elem(const line_index index) {
        return elems_[index];
    }

    const Element& elem(const line_index index) const {
        return elems_[index];
    }
```

```

    void set_key(const line_index index, const Key& new_key) {
        key_extract_(elem_write(index)) = new_key;
    }

    void set_elem(const line_index index, const Element& new_element) {
        elem_write(index) = new_element;
    }

    line_index min_index() const { return 0; }
    line_index max_index() const { return (count_ == 0) ? 0 : count_ - 1; }
    line_index end_index() const { return count_; }

    const Key& min_key() const { return key(min_index()); }
    const Element& min_elem() const { return elem(min_index()); }

    const Key& end_key() const { return key(end_index()); }
    const Element& end_elem() const { return elem(end_index()); }

private:
    ElementWrite& elem_write(const line_index index) {
        return reinterpret_cast<ElementWrite&>(elems_[index]);
    }

    ElementWrite& min_elem_write() {
        return elem_write(min_index());
    }

    // PREDICATES

public:
    bool empty() const { return count_ == 0; }

    // Is node at maximum capacity?
    bool full() const { return count_ == max_count; }

    // Is node at minimum capacity?
    bool thin() const { return count_ < min_count; }

    // OPERATIONS

    // Return the index of the greatest key less than or equal to the one
    // given, or the minimum index if all keys are greater than the one given
    line_index find(const Key& find_key) const
    {
        if (min_key() > find_key) {
            return min_index();
        }
        for (int i = min_index() + 1; i < end_index(); ++i) {
            if (key(i) > find_key) {
                return i - 1;
            }
        }
        return count_ - 1;
    }

private:
    // Move all the elements from one index to another (inclusive) back one
    // place
    void move_one_back(const line_index begin, const line_index end)
    {
        std::move_backward(&elem(begin), &elem(end), &elem_write(end + 1));
```

```
    }

    // Move all the elements from one index to another (inclusive) forward one
    // place
    void move_one_forward(const line_index begin, const line_index end)
    {
        std::move(&elem(begin), &elem(end), &elem_write(begin - 1));
    }

    // Move all the elements from one index to another (inclusive) to another
    // node, starting from some index in that node
    void move_to(const line_index begin, const line_index end,
        line_node& dest_node, const line_index dest)
    {
        std::move(&elem(begin), &elem(end), &dest_node.elem_write(dest));
    }

public:
    void init()
    {
        count_ = 0;
    }

    void init_from(const Element* begin, const Element* end)
    {
        std::move(begin, end, &min_elem_write());
        count_ = end - begin;
    }

    void init_from(const line_node& other)
    {
        std::move(&other.min_elem(), &other.end_elem(), &min_elem_write());
        count_ = other.count_;
    }

    // Insert a new element.  Assumes the node is not full
    void insert(const Element& new_elem)
    {
        line_index insert_index = end_index();
        for (int i = min_index(); i < end_index(); ++i) {
            if (key(i) > key_extract_(new_elem)) {
                insert_index = i;
                break;
            }
        }

        move_one_back(insert_index, end_index());

        set_elem(insert_index, new_elem);

        ++count_;
    }

    // Split node in half
    void split(line_node& split_node)
    {
        // This node takes half and the odd of the elements
        const auto new_count = count_/2 + count_%2;

        // The split node takes the the half
        split_node.count_ = count_/2;

```

```
        move_to(new_count, end_index(), split_node, split_node.min_index());

        count_ = new_count;
    }

    // Erase an element.  If node is thin this will put the node under capacity
    void erase(const line_index erase_index)
    {
        if (erase_index < end_index()) {
            move_one_forward(erase_index + 1, end_index());
        }

        --count_;
    }

    // Erase an element while merging node with the previous node.  The
    // elements go into the previous node, so this one is left empty
    void merge_prev_erase(const line_index erase_index, line_node& prev_node)
    {
        move_to(0, erase_index,
            prev_node, prev_node.end_index());
        move_to(erase_index + 1, end_index(),
            prev_node, prev_node.end_index() + erase_index);

        prev_node.count_ += count_ - 1;
        count_ = 0;
    }

    // Erase an element while merging node with the next node.  The elements go
    // into this node, so the next node is left empty
    void merge_next_erase(const line_index erase_index, line_node& next_node)
    {
        move_one_forward(erase_index + 1, end_index());

        next_node.move_to(
            next_node.min_index(), next_node.end_index(),
            *this, end_index() - 1);

        count_ += next_node.count_ - 1;
        next_node.count_ = 0;
    }

    // Erase an element while borrowing one from the previous node
    void borrow_prev_erase(const line_index erase_index, line_node& prev_node)
    {
        move_one_back(min_index(), erase_index);
        prev_node.move_to(
            prev_node.end_index() - 1, prev_node.end_index(),
            *this, min_index());

        prev_node.count_ -= 1;
    }

    // Erase an element while borrowing one from the next node
    void borrow_next_erase(const line_index erase_index, line_node& next_node)
    {
        move_one_forward(erase_index + 1, end_index());
        next_node.move_to(
            next_node.min_index(), next_node.min_index() + 1,
            *this, end_index() - 1);

        next_node.move_one_forward(
```

```
            next_node.min_index() + 1, next_node.end_index());
        next_node.count_ -= 1;
    }

    // Print all the keys in a comma-seperated list followed by a newline
    void print() const
    {
        if (count_ > 0)
        {
            for (int index = 0; index < end_index(); ++index)
            {
                std::cout << key(index);
                if (index < end_index() - 1)
                {
                    std::cout << ", ";
                }
            }
        }
        std::cout << std::endl;
    }
};

} // namespace detail

} // namespace double_tree
```

A.3 double_tree_page_node.hpp

```
// A tree based node that fits in a memory page

#pragma once
#include "double_tree_line_node.hpp"
#include "extract.hpp"
#include <array>
#include <cassert>
#include <cstdint>
#include <iostream>
#include <limits>

namespace double_tree
{

namespace detail
{

// Define here the size in bytes of, as well as the types used to index inside
// them.  A page node index should be able to index all values in an array of
// line nodes of size equal to the page node.  See below for the exact
// requirements, which are a little less because of the size used up for
// bookkeeping data in each case

constexpr size_t page_node_size = 4096;
using page_index = uint8_t;
constexpr page_index page_index_nil = std::numeric_limits<page_index>::max();

// An index into a page node combined with an index into the line node at that
// index constitutes a position of an element in the page node

struct page_position
{
    page_position() = default;
```

```
    page_position(const page_position& other):
        line{other.line},
        elem{other.elem}
    {}

    page_position(page_index line_, line_index elem_):
        line{line_},
        elem{elem_}
    {}

    bool operator==(const page_position& other) const
    {
        return line == other.line && elem == other.elem;
    }

    bool operator!=(const page_position& other) const
    {
        return line != other.line || elem != other.elem;
    }

    page_index line;
    line_index elem;
};


template <
    typename Element,
    typename ElementWrite,
    typename Key,
    typename KeyExtract,
    typename ValExtract,
    typename Aux>
struct alignas(page_node_size) PageNode
{
    // Auxiliary structures for the line nodes.  A stem node does not need any
    // extra data, while the leaf nodes of a page are linked together in a
    // linked list, so they need the index of their previous and next nodes
    struct stem_aux
    {};

    struct leaf_aux
    {
        page_index prev_index;
        page_index next_index;
    };

    // The actual node types can now be defined
    using stem_node = line_node<
        std::pair<const Key, page_index>,
        std::pair<Key, page_index>,
        Key,
        extract::first,
        extract::second,
        stem_aux>;
    using leaf_node = line_node<
        Element,
        ElementWrite,
        Key,
        KeyExtract,
        ValExtract,
        leaf_aux>;
```

```

    // MEMORY SYSTEM

    // An entry in the pool memory.  Each can hold either a stem node, a leaf
    // node or a page index.  The page index is only used within the memory
    // system
    struct alignas(line_node_size) PoolEntry
    {
        PoolEntry() {}

        union {
            stem_node stem;
            leaf_node leaf;
            page_index prev_head;
        };
    };

    // Calculate how many entries the pool memory can hold
    static constexpr int pool_size =
        page_node_size - 6*sizeof(page_index) - 1 - sizeof(Aux);
    static constexpr int pool_count = pool_size/sizeof(PoolEntry);

    // The pool memory.  The back index points at the highest entry that has
    // never been allocated.  The head index points at the next entry to
    // allocate, and is either equal to the back index or lower.  If it is
    // equal the next head index is found by incrementing.  If it is lower it
    // points to an entry that has been deallocated, and when that was done the
    // previous head index was stored there, and we restore it.
    page_index allocate() {
        assert(free_count > 0);
        --free_count;
        page_index allocate_index = head_index;
        if (head_index == back_index) { head_index = ++back_index; }
        else { head_index = pool_memory[head_index].prev_head; }
        return allocate_index;
    }

    void deallocate(page_index deallocate_index) {
        ++free_count;
        pool_memory[deallocate_index].prev_head = head_index;
        head_index = deallocate_index;
    }

    // SMALL, LARGE, OVERSIZED NODES

    static constexpr int branchout = stem_node::max_count;

    // Here n is the pool entries left and b is the total branchout of the
    // previous level.  By subtracting b from n and multiplying b by the
    // branchout until we cover the rest of the nodes we figure out how many
    // levels of stem nodes we will maximally need
    static constexpr int max_stem_levels_helper(int n, int b) {
        return (n > b) ? 1 + max_stem_levels_helper(n - b, b*branchout) : 0;
    }
    static constexpr int max_stem_levels = max_stem_levels_helper(pool_count,
     1);
    static constexpr int max_levels = max_stem_levels + 1;

    bool small() const {
        return free_count > 2*max_levels - 1;
    }

```

```
    bool large() const {
        return free_count <= 2*max_levels - 1;
    }

    bool oversized() const {
        return free_count <= max_levels - 1;
    }

    // DATA

    std::array<PoolEntry, pool_count> pool_memory;
    page_index head_index;
    page_index back_index;
    page_index free_count;

    page_index root_index;
    page_index min_leaf_index;
    page_index max_leaf_index;
    uint8_t stem_levels;
    KeyExtract key_extract_;
    Aux aux;

    // CONSTRUCTOR

    PageNode()
    : head_index{0},
      back_index{0},
      free_count{pool_count},
      root_index{allocate()},
      min_leaf_index{root_index},
      max_leaf_index{root_index},
      stem_levels{0}
    {
        auto& root = get_leaf(root_index);
        root.init();
        root.aux.prev_index = page_index_nil;
        root.aux.next_index = page_index_nil;
    }

    // ACCESSORS

    stem_node& get_stem(const page_index index) {
        return pool_memory[index].stem;
    }
    leaf_node& get_leaf(const page_index index) {
        return pool_memory[index].leaf;
    }

    const stem_node& get_stem(const page_index index) const {
        return pool_memory[index].stem;
    }
    const leaf_node& get_leaf(const page_index index) const {
        return pool_memory[index].leaf;
    }

    const Key& key(const page_position position) const {
        return get_leaf(position.line).key(position.elem);
    }
    Element& elem(const page_position position) {
        return get_leaf(position.line).elem(position.elem);
    }
    const Element& elem(const page_position position) const {
```

```
        return get_leaf(position.line).elem(position.elem);
    }

    void set_key(const page_position position, const Key& new_key) {
        const Key old_key = get_leaf(position.line).key(position.elem);
        get_leaf(position.line).set_key(position.elem, new_key);
        if (position.elem == 0) {
            const auto path = find_path(old_key);
            update_key(
                path, stem_levels - 1, path[stem_levels - 1].elem, new_key);
        }
    }

    page_position min_position() const {
        return {min_leaf_index, get_leaf(min_leaf_index).min_index()};
    }
    page_position max_position() const {
        return {max_leaf_index, get_leaf(max_leaf_index).max_index()};
    }
    page_position end_position() const {
        return {max_leaf_index,
            (line_index)(get_leaf(max_leaf_index).max_index() + 1)};
    }

    page_position prev_position(const page_position position) const {
        const leaf_node& node = get_leaf(position.line);
        if (node.aux.prev_index != page_index_nil &&
            position.elem == node.min_index())
        {
            return {node.aux.prev_index,
                get_leaf(node.aux.prev_index).max_index()};
        }
        else
        {
            return {position.line, (line_index)(position.elem - 1)};
        }
    }

    page_position next_position(const page_position position) const {
        const leaf_node& node = get_leaf(position.line);
        if (node.aux.next_index != page_index_nil &&
            position.elem == node.max_index())
        {
            return {node.aux.next_index,
                get_leaf(node.aux.next_index).min_index()};
        }
        else
        {
            return {position.line, (line_index)(position.elem + 1)};
        }
    }

    const Key& min_key() const {
        return get_leaf(min_leaf_index).min_key();
    }

    const Element& min_elem() const {
        return get_leaf(min_leaf_index).min_elem();
    }

    const Key& max_key() const {
        return get_leaf(max_leaf_index).max_key();
```

```
    }

    const Element& max_elem() const {
        return get_leaf(max_leaf_index).max_elem();
    }

    // PREDICATES

    bool empty() const {
        return stem_levels == 0 && get_leaf(root_index).empty();
    }

    // OPERATIONS

    // Returns the position of the greatest key less than or equal to the one
    // given, or the minimum position if all keys are greater than the one
    // given
    page_position find(const Key& find_key) const
    {
        auto search_index = root_index;
        for (int depth = 0; depth < stem_levels; ++depth)
        {
            const auto& search_stem = get_stem(search_index);
            search_index = search_stem.elem(search_stem.find(find_key)).second;
        }
        const auto& search_leaf = get_leaf(search_index);
        return {search_index, search_leaf.find(find_key)};
    }

private:
    // This structure is used to record a path down the tree to some specific
    // element
    using Path = std::array<page_position, max_levels>;

    // Construct the path taken to find the key given
    Path find_path(const Key& find_key) const
    {
        Path result;
        auto search_index = root_index;
        for (int depth = 0; depth < stem_levels; ++depth)
        {
            const auto& search_stem = get_stem(search_index);
            result[depth].line = search_index;
            result[depth].elem = search_stem.find(find_key);
            search_index = search_stem.elem(result[depth].elem).second;
        }
        const auto& search_leaf = get_leaf(search_index);
        result[stem_levels].line = search_index;
        result[stem_levels].elem = search_leaf.find(find_key);
        return result;
    }

    // Construct the leftmost path down the stem
    Path min_path() const
    {
        // Record the leftmost path down the stem.
        Path result;
        auto search_index = root_index;
        for (int depth = 0; depth < stem_levels; ++depth)
        {
            const auto& search_stem = get_stem(search_index);
            result[depth].line = search_index;
```

```
            result[depth].elem = search_stem.min_index();
            search_index = search_stem.elem(result[depth].elem).second;
        }
        const auto& search_leaf = get_leaf(search_index);
        result[stem_levels].line = search_index;
        result[stem_levels].elem = search_leaf.min_index();
        return result;
    }

    // Construct the rightmost path down the stem
    Path max_path() const
    {
        // Record the leftmost path down the stem.
        Path result;
        auto search_index = root_index;
        for (int depth = 0; depth < stem_levels; ++depth)
        {
            const auto& search_stem = get_stem(search_index);
            result[depth].line = search_index;
            result[depth].elem = search_stem.max_index();
            search_index = search_stem.elem(result[depth].elem).second;
        }
        const auto& search_leaf = get_leaf(search_index);
        result[stem_levels].line = search_index;
        result[stem_levels].elem = search_leaf.max_index();
        return result;
    }

    void split_root()
    {
        if (stem_levels > 0)
        {
            const auto old_root_index = root_index;
            auto& old_root = get_stem(old_root_index);

            if (old_root.full())
            {
                const auto new_index = allocate();
                auto& new_stem = get_stem(new_index);
                old_root.split(new_stem);

                root_index = allocate();
                auto& new_root = get_stem(root_index);
                new_root.init();
                new_root.insert({old_root.min_key(), old_root_index});
                new_root.insert({new_stem.min_key(), new_index});

                ++stem_levels;
            }
        }
        else
        {
            const auto old_root_index = root_index;
            auto& old_root = get_leaf(old_root_index);

            if (old_root.full())
            {
                const auto new_index = allocate();
                auto& new_leaf = get_leaf(new_index);
                old_root.split(new_leaf);

                old_root.aux.next_index = new_index;
```

```
                new_leaf.aux.prev_index = old_root_index;
                new_leaf.aux.next_index = page_index_nil;
                max_leaf_index = new_index;

                root_index = allocate();
                auto& new_root = get_stem(root_index);
                new_root.init();
                new_root.insert({old_root.min_key(), old_root_index});
                new_root.insert({new_leaf.min_key(), new_index});

                ++stem_levels;
            }
        }
    }

public:
    // Insert a new element
    void insert(const Element& new_elem)
    {
        split_root();

        const auto& new_key = key_extract_(new_elem);

        if (stem_levels > 0)
        {
            auto current_index = root_index;
            for (int depth = 0; depth < stem_levels - 1; ++depth)
            {
                auto& current_stem = get_stem(current_index);

                const auto target_pos = current_stem.find(new_key);
                const auto target_index = current_stem.elem(target_pos).second;
                auto& target_stem = get_stem(target_index);

                if (new_key < target_stem.min_key()) {
                    current_stem.set_key(target_pos, new_key);
                }

                // Split target stem?
                if (target_stem.full()) {
                    const auto new_index = allocate();
                    auto& new_stem = get_stem(new_index);
                    target_stem.split(new_stem);

                    current_stem.insert({new_stem.min_key(), new_index});

                    if (new_key >= new_stem.min_key()) {
                        current_index = new_index;
                    } else {
                        current_index = target_index;
                    }
                } else {
                    current_index = target_index;
                }
            }

            auto& current_stem = get_stem(current_index);

            const auto target_pos = current_stem.find(new_key);
            const auto target_index = current_stem.elem(target_pos).second;
            auto& target_leaf = get_leaf(target_index);

```

```
            if (new_key < target_leaf.min_key()) {
                current_stem.set_key(target_pos, new_key);
            }

            // Split target leaf?
            if (target_leaf.full()) {
                const auto new_index = allocate();
                auto& new_leaf = get_leaf(new_index);
                target_leaf.split(new_leaf);

                current_stem.insert({new_leaf.min_key(), new_index});

                // Adjust the linked leaf list to fit in the new leaf
                if (target_leaf.aux.next_index != page_index_nil) {
                    get_leaf(target_leaf.aux.next_index).aux.prev_index =
                        new_index;
                }
                new_leaf.aux.next_index = target_leaf.aux.next_index;
                new_leaf.aux.prev_index = target_index;
                target_leaf.aux.next_index = new_index;

                // If the target leaf was the max leaf, the new max is the new
                // leaf
                if (max_leaf_index == target_index) {
                    max_leaf_index = new_index;
                }

                if (new_key >= new_leaf.min_key()) {
                    new_leaf.insert(new_elem);
                } else {
                    target_leaf.insert(new_elem);
                }
            } else {
                target_leaf.insert(new_elem);
            }
        }
        else
        {
            auto& current_leaf = get_leaf(root_index);
            current_leaf.insert(new_elem);
        }
    }

    void insert_min_leaf(const Key new_min_key, const page_index new_index)
    {
        if (stem_levels > 0)
        {
            split_root();

            auto current_index = root_index;
            for (int depth = 0; depth < stem_levels - 1; ++depth)
            {
                auto& current_stem = get_stem(current_index);
                const auto target_pos = current_stem.min_index();
                const auto target_index = current_stem.elem(target_pos).second;
                auto& target_stem = get_stem(target_index);
                current_stem.set_key(target_pos, new_min_key);

                // Split target stem?
                if (target_stem.full()) {
                    const auto new_index = allocate();
                    auto& new_stem = get_stem(new_index);
```

```
                target_stem.split(new_stem);
                current_stem.insert({new_stem.min_key(), new_index});
            }

            current_index = target_index;
        }

        auto& current_stem = get_stem(current_index);
        current_stem.insert({new_min_key, new_index});
    }
    else
    {
        const auto old_root_index = root_index;
        auto& old_root = get_leaf(old_root_index);

        root_index = allocate();
        auto& new_root = get_stem(root_index);
        new_root.init();
        new_root.insert({new_min_key, new_index});
        new_root.insert({old_root.min_key(), old_root_index});

        ++stem_levels;
    }
}

void insert_max_leaf(const Key new_min_key, const page_index new_index)
{
    if (stem_levels > 0)
    {
        split_root();

        auto current_index = root_index;
        for (int depth = 0; depth < stem_levels - 1; ++depth)
        {
            auto& current_stem = get_stem(current_index);
            const auto target_pos = current_stem.max_index();
            const auto target_index = current_stem.elem(target_pos).second;
            auto& target_stem = get_stem(target_index);

            // Split target stem?
            if (target_stem.full()) {
                const auto new_index = allocate();
                auto& new_stem = get_stem(new_index);
                target_stem.split(new_stem);
                current_stem.insert({new_stem.min_key(), new_index});
                current_index = new_index;
            } else {
                current_index = target_index;
            }
        }

        auto& current_stem = get_stem(current_index);
        current_stem.insert({new_min_key, new_index});
    }
    else
    {
        const auto old_root_index = root_index;
        auto& old_root = get_leaf(old_root_index);

        root_index = allocate();
        auto& new_root = get_stem(root_index);
        new_root.init();
```

```
            new_root.insert({old_root.min_key(), old_root_index});
            new_root.insert({new_min_key, new_index});

            ++stem_levels;
        }
    }

    // Erase an element.  The page might be left thin in which case the caller
    // should decide what to do about that
    void erase(const Key& erase_key)
    {
        const auto path = find_path(erase_key);

        const auto line = path[stem_levels].line;
        const auto elem = path[stem_levels].elem;
        auto& erase_leaf = get_leaf(line);

        // If the node is not root and thin, we must either merge or borrow
        if (stem_levels > 0 && erase_leaf.thin())
        {
            // The parent node, and index of the node in the parent
            const auto parent_line_index = path[stem_levels - 1].elem;

            // Do we have a previous sibling?
            if (erase_leaf.aux.prev_index != page_index_nil)
            {
                const auto prev_index = erase_leaf.aux.prev_index;
                auto& prev_leaf = get_leaf(prev_index);

                // Can we merge?
                if (erase_leaf.count() + prev_leaf.count()
                    <= leaf_node::max_count)
                {
                    // Merge the leaf with the element into its previous
                    // sibling
                    erase_leaf.merge_prev_erase(elem, prev_leaf);

                    // Adjust the linked leaf list to erase the merged node
                    if (erase_leaf.aux.next_index != page_index_nil) {
                        get_leaf(erase_leaf.aux.next_index).aux.prev_index =
                            prev_index;
                    }
                    prev_leaf.aux.next_index = erase_leaf.aux.next_index;

                    // If the erased node was the max leaf, the new max is the
                    // previous sibling
                    if (max_leaf_index == line) {
                        max_leaf_index = prev_index;
                    }

                    deallocate(line);

                    // Now we must erase the erased node from the stem
                    // structure
                    erase_node(path, stem_levels - 1, parent_line_index);
                }
                // If we can not merge, borrow
                else
                {
                    erase_leaf.borrow_prev_erase(elem, prev_leaf);

                    // Since the node with the element has a new minimum
```

```
                // element we must update the representative keys up the
                // tree
                update_key(path, stem_levels - 1,
                    parent_line_index, erase_leaf.min_key());
            }
        }
        // If we do not have a previous sibling, we should have a next
        else
        {
            const auto next_index = erase_leaf.aux.next_index;
            auto& next_leaf = get_leaf(next_index);

            // Can we merge?
            if (erase_leaf.count() + next_leaf.count()
                <= leaf_node::max_count)
            {
                // Merge the next sibling into the leaf with the element
                erase_leaf.merge_next_erase(elem, next_leaf);

                // Adjust the linked leaf list to erase the merged node
                if (next_leaf.aux.next_index != page_index_nil) {
                    get_leaf(next_leaf.aux.next_index).aux.prev_index =
                        line;
                }
                erase_leaf.aux.next_index = next_leaf.aux.next_index;

                // If the erased node was the max leaf, the new max is the
                // leaf with the element in it
                if (max_leaf_index == next_index) {
                    max_leaf_index = line;
                }

                deallocate(next_index);

                // If we erased the minimum element we must update the
                // representative keys up the tree
                if (elem == 0) {
                    update_key(path, stem_levels - 1,
                        parent_line_index, erase_leaf.min_key());
                }

                // Now we must erase the erased node from its parent
                erase_node(path, stem_levels - 1, parent_line_index + 1);
            }
            // If we can not merge, borrow
            else
            {
                erase_leaf.borrow_next_erase(elem, next_leaf);

                // Since the next leaf has a new minimum we must update the
                // representative keys up the tree
                update_key(path, stem_levels - 1,
                    parent_line_index + 1, next_leaf.min_key());

                // If we erased the minimum element we must update the
                // representative keys up the tree
                if (elem == 0) {
                    update_key(path, stem_levels - 1,
                        parent_line_index, erase_leaf.min_key());
                }
            }
        }
```

```
        }
        // Either we are the root or the stem to erase from is not thin
        else
        {
            erase_leaf.erase(elem);

            // If we are not the root and we erased the minimum element, we
            // must update the representative keys up the tree
            if (stem_levels > 0 && elem == 0)
            {
                const auto parent_line_index = path[stem_levels - 1].elem;
                update_key(path, stem_levels - 1,
                    parent_line_index, erase_leaf.min_key());
            }
        }
    }

    // Erase a node from the stem structure.  The depth given should be the
    // depth of the node to erase from
    void erase_node(const Path& path, const int depth, const line_index elem)
    {
        const auto line = path[depth].line;
        auto& erase_stem = get_stem(line);

        // If the node is not root and thin, we must either merge or borrow
        if (depth > 0 && erase_stem.thin())
        {
            // The parent node, and index of the node in the parent
            const auto parent_page_index = path[depth - 1].line;
            const auto parent_line_index = path[depth - 1].elem;
            const auto& parent = get_stem(parent_page_index);

            // Do we have a previous sibling?
            if (parent_line_index > 0)
            {
                const auto prev_index =
                    parent.elem(parent_line_index - 1).second;
                auto& prev_stem = get_stem(prev_index);

                // Can we merge?
                if (erase_stem.count() + prev_stem.count()
                    <= stem_node::max_count)
                {
                    // Merge the stem with the node into its previous sibling
                    erase_stem.merge_prev_erase(elem, prev_stem);

                    deallocate(line);

                    // Now we must erase the erased node from the stem
                    // structure
                    erase_node(path, depth - 1, parent_line_index);
                }
                // If we can not merge, borrow
                else
                {
                    erase_stem.borrow_prev_erase(elem, prev_stem);

                    // Since the stem with the node has a new minimum element
                    // we must update the representative keys up the tree
                    update_key(path, depth - 1,
                        parent_line_index, erase_stem.min_key());
                }
```

```
            }
            // If we do not have a previous sibling, we should have a next
            else
            {
                const auto next_index =
                    parent.elem(parent_line_index + 1).second;
                auto& next_leaf = get_stem(next_index);

                // Can we merge?
                if (erase_stem.count() + next_leaf.count()
                    <= stem_node::max_count)
                {
                    // Merge the next sibling into the stem with the node
                    erase_stem.merge_next_erase(elem, next_leaf);

                    deallocate(next_index);

                    // If we erased the minimum element we must update the
                    // representative keys up the tree
                    if (elem == 0) {
                        update_key(path, depth - 1,
                            parent_line_index, erase_stem.min_key());
                    }

                    // Now we must erase the erased node from its parent
                    erase_node(path, depth - 1, parent_line_index + 1);
                }
                // If we can not merge, borrow
                else
                {
                    erase_stem.borrow_next_erase(elem, next_leaf);

                    // Since the next stem has a new minimum we must update the
                    // representative keys up the tree
                    update_key(path, depth - 1,
                        parent_line_index + 1, next_leaf.min_key());

                    // If we erased the minimum element we must update the
                    // representative keys up the tree
                    if (elem == 0) {
                        update_key(path, depth - 1,
                            parent_line_index, erase_stem.min_key());
                    }
                }
            }
        }
        // Either we are the root or the stem to erase from is not thin
        else
        {
            erase_stem.erase(elem);

            // If we are not the root and we erased the minimum element, we
            // must update the representative keys up the tree
            if (depth > 0 && elem == 0) {
                const auto parent_line_index = path[depth - 1].elem;
                update_key(path, depth - 1,
                    parent_line_index, erase_stem.min_key());
            }

            // If we are the root and we now have only one child we should
            // collapse this level of the tree
            if (depth == 0 && erase_stem.count() == 1) {
```

```
                root_index = erase_stem.min_elem().second;
                deallocate(line);
                --stem_levels;
            }
        }
    }

    void update_key(const Path& path, const int depth,
        const line_index elem, const Key& new_key)
    {
        auto& stem = get_stem(path[depth].line);
        stem.set_key(elem, new_key);
        if (depth > 0 && elem == 0)
        {
            update_key(path, depth - 1, path[depth - 1].elem, new_key);
        }
    }

    void borrow_prev(PageNode& prev_page)
    {
        const auto prev_path = prev_page.max_path();
        const auto old_index = prev_path[prev_page.stem_levels].line;
        auto& old_leaf = prev_page.get_leaf(old_index);

        if (old_leaf.count() < leaf_node::min_count) {
            // TODO: This could be done more efficiently
            for (int i = 0; i < old_leaf.count(); ++i) {
                insert(old_leaf.elem(i));
            }
        } else {
            const auto this_path = min_path();

            // Copy the leaf
            auto new_index = allocate();
            auto& new_leaf = get_leaf(new_index);

            const auto next_index = this_path[stem_levels].line;
            auto& next_leaf = get_leaf(next_index);
            next_leaf.aux.prev_index = new_index;
            new_leaf.aux.prev_index = page_index_nil;
            new_leaf.aux.next_index = next_index;

            new_leaf.init_from(old_leaf);

            min_leaf_index = new_index;

            // Insert into this pages stem structure
            insert_min_leaf(new_leaf.min_key(), new_index);
        }

        // Erase the node from the other pages stem structure
        if (prev_page.stem_levels != 0) {
            prev_page.max_leaf_index = old_leaf.aux.prev_index;
            prev_page.get_leaf(prev_page.max_leaf_index).aux.next_index =
                page_index_nil;
            prev_page.deallocate(old_index);
            prev_page.erase_node(prev_path, prev_page.stem_levels - 1,
                prev_path[prev_page.stem_levels - 1].elem);
        } else {
            old_leaf.reset();
        }
    }
```

```

void borrow_next(PageNode& next_page)
{
    const auto next_path = next_page.min_path();
    const auto old_index = next_path[next_page.stem_levels].line;
    auto& old_leaf = next_page.get_leaf(old_index);

    if (old_leaf.count() < leaf_node::min_count) {
        // TODO: This could be done more efficiently
        for (int i = 0; i < old_leaf.count(); ++i) {
            insert(old_leaf.elem(i));
        }
    } else {
        const auto this_path = max_path();

        // Copy the leaf
        const auto new_index = allocate();
        auto& new_leaf = get_leaf(new_index);

        const auto prev_index = this_path[stem_levels].line;
        auto& prev_leaf = get_leaf(prev_index);
        prev_leaf.aux.next_index = new_index;
        new_leaf.aux.prev_index = prev_index;
        new_leaf.aux.next_index = page_index_nil;

        new_leaf.init_from(old_leaf);

        max_leaf_index = new_index;

        // Update the keys on the path
        auto new_key = next_page.min_key();
        for (int depth = 0; depth < next_page.stem_levels - 1; ++depth)
        {
            auto& stem = next_page.get_stem(next_path[depth].line);
            stem.set_key(next_path[depth].elem, new_key);
        }

        // Insert into this pages stem structure
        insert_max_leaf(new_leaf.min_key(), new_index);
    }

    // Erase the node from the other pages stem structure
    if (next_page.stem_levels != 0) {
        next_page.min_leaf_index = old_leaf.aux.next_index;
        next_page.get_leaf(next_page.min_leaf_index).aux.prev_index =
            page_index_nil;
        next_page.deallocate(old_index);
        next_page.erase_node(next_path, next_page.stem_levels - 1,
            next_path[stem_levels - 1].elem);
    } else {
        old_leaf.reset();
    }
}

PageNode* split_one_leaf()
{
    const auto this_path = max_path();

    // Copy the leaf
    const auto new_page_pointer = new PageNode;
    auto& new_page = *new_page_pointer;
    auto new_index = new_page.root_index;
```

```
        auto& new_leaf = new_page.get_leaf(new_index);

        const auto old_index = this_path[stem_levels].line;
        auto& old_leaf = get_leaf(old_index);
        new_leaf.init_from(old_leaf);

        // Erase the node from this pages stem structure
        if (stem_levels != 0) {
            max_leaf_index = old_leaf.aux.prev_index;
            get_leaf(max_leaf_index).aux.next_index = page_index_nil;
            deallocate(old_index);
            erase_node(this_path, stem_levels - 1,
                this_path[stem_levels - 1].elem);
        } else {
            old_leaf.reset();
        }

        return new_page_pointer;
    }

    void print() const
    {
        print_node(root_index, 0);
    }

    void print_tabs(int num) const
    {
        for (int i = 0; i < num; ++i)
        {
            std::cout << "   ";
        }
    }

    void print_node(page_index line, int depth) const
    {
        if (depth < stem_levels)
        {
            const stem_node& stem = get_stem(line);
            print_tabs(depth);
            std::cout << "stem (" << depth << ") ";
            stem.print();
            for (int i = 0; i < stem.count(); ++i) {
                print_node(stem.elem(i).second, depth + 1);
            }
        }
        else
        {
            const leaf_node& leaf = get_leaf(line);
            print_tabs(depth);
            std::cout << "leaf (" << depth << ") ";
            leaf.print();
        }
    }
};

} // namespace detail

} // namespace double_tree
```

A.4 etract.hpp

```
#pragma once

namespace extract
{

// Used as KeyExtract for the sets.
struct identity {
    template<typename U>
    constexpr auto operator()(U&& v) const -> decltype(std::forward<U>(v)) {
        return std::forward<U>(v);
    }
};

// Used as KeyExtract for the maps.
struct first {
    template<typename U>
    constexpr auto operator()(U&& v) const -> decltype(std::get<0>(v)) {
        return std::get<0>(v);
    }
};

// Used as MappedExtract for the maps.
struct second {
    template<typename U>
    constexpr auto operator()(U&& v) const -> decltype(std::get<1>(v)) {
        return std::get<1>(v);
    }
};

}
```

A.5 hopscotch.hpp

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

```
#pragma once

#include <algorithm>
#include <bitset>
#include <cassert>
#include <cmath>
#include <cstddef>
#include <cstring>
#include <functional>
#include <initializer_list>
#include <iterator>
#include <memory>
#include <utility>
#include <vector>

#include <iostream>

#include "extract.hpp"

namespace hopscotch
{
namespace detail
{
// KERNEL

```

```
template <
    class Value,
    class Key,
    class Hash,
    class KeyExtract,
    class MappedExtract,
    class KeyEqual,
    class Allocator>
class kernel
{
protected:
    // I have not seen much variation in performance from changing this, so it
    // is maxed out for flexibility. It is one from 64 because we use the same
    // bitset to store the hop info and whether a bucket has a value or not.
    static const size_t neighborhood_size = 63;

    // Defined below.
    template <typename T> class iterator_template;

public:
    // MEMBER TYPES

    using key_type        = Key;
    using value_type      = Value;
    using size_type       = std::size_t;
    using difference_type = std::ptrdiff_t;
    using hasher          = Hash;
    using key_equal       = KeyEqual;
    using key_extract     = KeyExtract;
    using mapped_extract  = MappedExtract;
    using allocator_type  = Allocator;
    using reference       = value_type&;
    using const_reference = const value_type&;
    using pointer         =
        typename std::allocator_traits<Allocator>::pointer;
    using const_pointer   =
        typename std::allocator_traits<Allocator>::const_pointer;
    using iterator        = iterator_template<value_type>;
    using const_iterator  = iterator_template<const value_type>;

    // CONSTRUCTORS, ET CETERA

    // Default constructor.
    explicit kernel(
        size_t bucket_count = 16,
        const hasher& hash = hasher(),
        const key_equal& equal = key_equal(),
        const allocator_type& alloc = allocator_type())
      // Instances of functors.
    : hash_{hash},
      equal_{equal},
      extract_{},
      alloc_{alloc},
      // Bucket vector.
      buckets_{alloc_},
      min_load_{0.3},
      max_load_{0.7},
      // We always have a number of buckets that is a power of two.
      bucket_count_{upper_power_of_two(bucket_count)},
      // ...
      size_{0},
      min_size_{size_t(bucket_count_ * min_load_)},
```

```
      max_size_{size_t(bucket_count_ * max_load_)}
    {
        // Allocate space.
        buckets_.resize(bucket_count_);
    }

    // Copy constructor.
    kernel(
        const kernel& other)
    : hash_{other.hash_},
      equal_{other.equal_},
      extract_{},
      alloc_{other.alloc_},
      buckets_{other.buckets_},
      min_load_{other.min_load_},
      max_load_{other.max_load_},
      bucket_count_{other.bucket_count_},
      size_{other.size_},
      min_size_{other.min_size_},
      max_size_{other.max_size_}
    {}

    // Copy constructor with allocator.
    kernel(
        const kernel& other,
        const allocator_type& alloc)
    : hash_{other.hash_},
      equal_{other.equal_},
      extract_{},
      alloc_{alloc},
      buckets_{other.buckets_, alloc},
      min_load_{other.min_load_},
      max_load_{other.max_load_},
      bucket_count_{other.bucket_count_},
      size_{other.size_},
      min_size_{other.min_size_},
      max_size_{other.max_size_}
    {}

    // Move constructor.
    kernel(kernel&& other)
    : kernel{}
    {
        swap(*this, other);
    }

    // TODO
    // kernel(kernel&& other, const allocator_type& alloc);

    // Swapping (member).
    void swap(kernel& other)
    {
        swap(*this, other);
    }

    // Swapping (friend).
    friend void swap(kernel& lhs, kernel& rhs)
    {
        using std::swap;
        swap(lhs.hash_, rhs.hash_);
        swap(lhs.equal_, rhs.equal_);
        swap(lhs.alloc_, rhs.alloc_);
```

```
        swap(lhs.buckets_, rhs.buckets_);
        swap(lhs.values_, rhs.values_);
        swap(lhs.min_load_, rhs.min_load_);
        swap(lhs.max_load_, rhs.max_load_);
        swap(lhs.bucket_count_, rhs.bucket_count_);
        swap(lhs.size_, rhs.size_);
        swap(lhs.min_size_, rhs.min_size_);
        swap(lhs.max_size_, rhs.max_size_);
    }

    // Copy assignment.
    kernel& operator=(kernel other)
    {
        swap(*this, other);
        return *this;
    }

    // ITERATOR GETTERS

    iterator begin() {
        return make_iterator(buckets_begin());
    }

    const_iterator begin() const {
        return make_const_iterator(buckets_begin());
    }

    const_iterator cbegin() const {
        return make_const_iterator(buckets_begin());
    }

    iterator end() {
        return make_iterator(buckets_end());
    }

    const_iterator end() const {
        return make_const_iterator(buckets_end());
    }

    const_iterator cend() const {
        return make_const_iterator(buckets_end());
    }

    // CAPACITY

    bool empty() const {
        return size_ == 0;
    }

    size_t size() const {
        return size_;
    }

    // CLEAR

    void clear()
    {
        for (int index = 0; index < bucket_count_; ++index)
        {
            auto& bucket = buckets_[index];
            bucket.memory()->~value_type();
            bucket.has_value(false);
```

```
            bucket.hop_info.clear();
        }
        size_ = 0;
    }

    // ERASE

    size_t erase(const key_type& key)
    {
        size_t erased = 0;

        const size_t virtual_index = index_from_key(key);
        auto& virtual_bucket = buckets_[virtual_index];

        size_t hop = next_hop(virtual_bucket);
        while (hop < neighborhood_size)
        {
            const size_t index = index_add(virtual_index, hop);
            auto& bucket = buckets_[index];
            if (equal_(key, extract_(*bucket.memory())))
            {
                bucket.memory()->~value_type();

                bucket.has_value(false);

                virtual_bucket.hop_info[hop] = false;

                ++erased;
            }
            hop = next_hop(virtual_bucket, hop);
        }

        size_ -= erased;

        // Rehash if this brought us below min load.
        if (size_ < min_size_ && size_ > 16) {
            rehash(bucket_count_/2);
        }

        return erased;
    }

    // COUNT

    size_t count(const Key& key) const
    {
        size_t result = 0;

        const size_t virtual_index = index_from_key(key);
        auto& virtual_bucket = buckets_[virtual_index];

        size_t hop = next_hop(virtual_bucket);
        while (hop < neighborhood_size)
        {
            const size_t index = index_add(virtual_index, hop);
            auto& bucket = buckets_[index];
            if (equal_(key, extract_(*bucket.memory())))
            {
                ++result;
            }
            hop = next_hop(virtual_bucket, hop);
        }
```

```
        return count;
    }

    // OPERATOR[]

    auto& operator[](const Key& key) {
        return mapped_extract_(
            *(buckets_[find(key, index_from_key(key))].memory()));
    }

    const auto& operator[](const Key& key) const {
        return mapped_extract_(
            *(buckets_[find(key, index_from_key(key))].memory()));
    }

    // FIND

    iterator find(const Key& key) {
        return make_iterator(find(key, index_from_key(key)));
    }

    const_iterator find(const Key& key) const {
        return make_const_iterator(find(key, index_from_key(key)));
    }

    // BUCKET INTERFACE

    size_t bucket_count() const {
        return bucket_count_;
    }

    // HASH POLICY

    float load_factor() const {
        return (float)size_/(float)bucket_count_;
    }

    float min_load_factor() const {
        return min_load_;
    }

    float max_load_factor() const {
        return max_load_;
    }

    void min_load_factor(float min_load)
    {
        min_load_ = min_load;

        // Check if we need to rehash.
        min_size_ = min_load_ * bucket_count_;
        if (size_ < min_size_) {
            rehash(bucket_count_/2);
        }
    }

    void max_load_factor(float max_load)
    {
        max_load_ = max_load;

        // Check if we need to rehash.
```

```
        max_size_ = max_load_ * bucket_count_;
        if (size_ > max_size_) {
            rehash(bucket_count_*2);
        }
    }

    void rehash(size_t count)
    {
        bucket_count_ = upper_power_of_two(count);

        // Create the new bucket vector, saving the old one.
        bucket_vector old_buckets{bucket_count_};
        std::swap(buckets_, old_buckets);

        // Set up the state so we can insert correctly.
        size_ = 0;
        min_size_ = min_load_ * bucket_count_;
        max_size_ = max_load_ * bucket_count_;

        // Rehash.
        for (size_t index = 0; index < old_buckets.size(); ++index)
        {
            auto& bucket = old_buckets[index];
            if (bucket.has_value())
            {
                insert(std::move(
                    *bucket.memory()), index_from_value(*bucket.memory()));
                // Detect recursive rehash and break.
                if (bucket_count_ != count) { break; }
            }
        }
    }

    void reserve(size_t count)
    {
        rehash(std::ceil((float)count/max_load_factor()));
    }

    // OBSERVERS

    hasher hash_function() const {
        return hash_;
    }

    key_equal key_eq() const {
        return equal_;
    }

    allocator_type get_allocator() const {
        return alloc_;
    }

protected:
    // BUCKET TYPE

    class bucket_type
    {
    public:
        void has_value(bool has_value) {
            hop_info[neighborhood_size] = has_value;
        }

```

```
        bool has_value() const {
            return hop_info[neighborhood_size];
        }

        std::bitset<neighborhood_size+1> hop_info;

        value_type* memory() {
            return reinterpret_cast<value_type*>(&memory_);
        }

        const value_type* memory() const {
            return reinterpret_cast<const value_type*>(&memory_);
        }

        struct { unsigned char _[sizeof(value_type)]; } memory_;
    };

    using bucket_vector = std::vector<bucket_type,
          typename allocator_type::template rebind<bucket_type>::other>;

    // ITERATOR TYPE

    template <typename T>
    class iterator_template : std::iterator<std::forward_iterator_tag, T>
    {
        friend kernel;

    public:
        iterator_template()
        : index_(0),
          buckets_(nullptr)
        {}

        iterator_template(const iterator_template& other)
        : index_(other.index_),
          buckets_(other.buckets_)
        {}

        iterator_template& operator=(const iterator_template& other)
        {
            index_ = other.index_;
            buckets_ = other.buckets_;
            return *this;
        }

        ~iterator_template()
        {}

        reference operator*() {
            return *(buckets_->operator[](index_).memory());
            }

        const_reference operator*() const {
            return *(buckets_->operator[](index_).memory());
        }

        pointer operator->() {
            return buckets_->operator[](index_).memory();
        }

        const_pointer operator->() const {
            return buckets_->operator[](index_).memory();
```

```
        }

        iterator_template& operator++()
        {
            do {
                ++index_;
            }
            while (index_ != buckets_->size() &&
                !buckets_->operator[](index_).has_value());
            return *this;
        }

        iterator_template operator++(int)
        {
            iterator_template old(*this);
            ++*this;
            return old;
        }

        bool operator==(const iterator_template& other) const {
            return index_ == other.index_ && buckets_ == other.buckets_;
        }

        bool operator!=(const iterator_template& other) const {
            return index_ != other.index_ || buckets_ != other.buckets_;
        }

    private:
        iterator_template(
            size_t index,
            bucket_vector* buckets)
        : index_(index),
          buckets_(buckets)
        {}

        size_t index_;

        bucket_vector* buckets_;
    };

    // DATA MEMBERS

    hasher hash_;
    key_equal equal_;
    key_extract extract_;
    mapped_extract mapped_extract_;
    allocator_type alloc_;

    bucket_vector buckets_;
    float min_load_;
    float max_load_;

    size_t bucket_count_;

    size_t size_;
    size_t min_size_;
    size_t max_size_;

    // UPPER POWER OF TWO

    size_t upper_power_of_two(size_t x) const
    {
```

```
        // This implementation was found here (adjusted for 64-bit):
        // http://graphics.stanford.edu/~seander/bithacks.html#RoundUpPowerOf2
        --x;
        x |= x >> 1;
        x |= x >> 2;
        x |= x >> 4;
        x |= x >> 8;
        x |= x >> 16;
        x |= x >> 32;
        ++x;
        return x;
    }

    // INDEX HELPER

    size_t index_from_value(const value_type& value) const {
        return index_from_key(extract_(value));
    }

    size_t index_from_key(const key_type& key) const {
        // This bitwise and is the same as doing a modulo because the bucket
        // count is guaranteed to be a power of two.
        return hash_(key) & (bucket_count_ - 1);
    }

    size_t index_add(size_t index, size_t x) const {
        // This bitwise and is the same as doing a modulo because the bucket
        // count is guaranteed to be a power of two.
        return (index + x) & (bucket_count_ - 1);
    }

    size_t index_sub(size_t index, size_t x) const {
        // As above, this corresponds to a modulo operation, except that we
        // always get a positive value this way (as is desired).
        return (index - x) & (bucket_count_ - 1);
    }

    // HOPPING

    size_t next_hop(const bucket_type& bucket, int prev = -1) const
    {
        const size_t mask = 0xffffffffffffffff << (prev + 1);
        const size_t hop_info = bucket.hop_info.to_ulong();
        return __builtin_ffsl(hop_info & mask) - 1;
    }

    // ITERATOR HELPERS

    size_t buckets_begin()
    {
        if (empty()) {
            return buckets_.size();
        } else {
            auto bucket_it = buckets_.begin();
            while (!bucket_it->has_value()) { ++bucket_it; }
            return bucket_it - buckets_.begin();
        }
    }

    size_t buckets_end() {
        return buckets_.size();
    }
```

```

    iterator make_iterator(size_t index) {
        return {index, &buckets_};
    }

    const_iterator make_const_iterator(size_t index) const {
        return {index, &buckets_};
    }

public:
    std::pair<iterator, bool>
    insert(value_type value)
    {
        size_t index = index_from_value(value);
        size_t res = find(extract_(value), index);

        if (res == buckets_.size()) {
            return {insert(std::move(value), index), true};
        } else {
            return {make_iterator(res), false};
        }
    }

    iterator
    insert(
        const_iterator hint,
        value_type value)
    {
        (void)hint;  // Silence 'unused parameter'
        return insert(std::move(value)).first;
    }

private:
    // INSERT IMPLEMENTATION

    iterator insert(value_type value, size_t virtual_index)
    {
        // Start by rehashing, if this will bring us above max load.
        if (size_ == max_size_) {
            // std::cout << std::endl;
            // std::cout << "rehash (max load)";
            // std::cout << std::endl;
            rehash(bucket_count_*2);
            return insert(value, index_from_value(value));
        }

        // Find the nearest free bucket, wrapping if we move past the end.
        size_t free_dist = 0;
        size_t free_index = virtual_index;
        while (buckets_[free_index].has_value()) {
            free_dist += 1;
            free_index = index_add(free_index, 1);
        }

        // Move buckets until we have a free bucket in the neighborhood of our
        // virtual bucket.
        while (free_dist > neighborhood_size - 1)
        {
            // Find a virtual bucket that has values stored in a bucket before
            // the free bucket we found.
            size_t virtual_move_dist = neighborhood_size - 1;
```

```
            size_t virtual_move_index = index_sub(free_index, virtual_move_dist
    );

            size_t move_hop;

            while (true)
            {
                auto& virtual_move_bucket = buckets_[virtual_move_index];
                auto hop_info = virtual_move_bucket.hop_info.to_ulong();
                move_hop = __builtin_ffsl(hop_info) - 1;

                if (move_hop < virtual_move_dist) {
                    break;
                } else {
                    // No luck, continue searching.
                    virtual_move_dist -= 1;
                    virtual_move_index = index_add(virtual_move_index, 1);

                    if (virtual_move_dist == 0)
                    {
                        // All possibilities exhausted: resize, rehash, and
                        // start over with the insertion.
                        rehash(bucket_count_*2);
                        return insert(value, index_from_value(value));
                    }
                }
            }

            // Move.
            const size_t move_dist = virtual_move_dist - move_hop;
            const size_t move_index = index_add(virtual_move_index, move_hop);

            auto& move_bucket = buckets_[move_index];
            auto& free_bucket = buckets_[free_index];
            new (free_bucket.memory())
                value_type{std::move(*move_bucket.memory())};
            move_bucket.memory()->~value_type();
            move_bucket.has_value(false);
            free_bucket.has_value(true);

            auto& virtual_move = buckets_[virtual_move_index];
            virtual_move.hop_info[move_hop] = false;
            virtual_move.hop_info[virtual_move_dist] = true;

            // The free bucket is now in the position of the moved bucket.
            free_dist -= move_dist;
            free_index = index_sub(free_index, move_dist);
        }

        // We should have a free bucket in the neighborhood now.
        auto& free_bucket = buckets_[free_index];
        new (free_bucket.memory()) value_type{std::move(value)};
        free_bucket.has_value(true);

        auto& virtual_bucket = buckets_[virtual_index];
        virtual_bucket.hop_info[free_dist] = true;

        ++size_;

        return {free_index, &buckets_};
    }

```

```
    // FIND IMPLEMENTATION

    size_t find(const key_type& key, size_t virtual_index) const
    {
        // Find the virtual bucket of this key.
        const auto& virtual_bucket = buckets_[virtual_index];

        // Go through each of the hops in the virtual bucket.
        size_t hop = next_hop(virtual_bucket);
        while (hop < neighborhood_size)
        {
            const size_t index = index_add(virtual_index, hop);
            const auto& bucket = buckets_[index];
            if (equal_(key, extract_(*bucket.memory()))) {
                return index;
            }
            hop = next_hop(virtual_bucket, hop);
        }

        // We found nothing, return end.
        return bucket_count_;
    }
};

} // namespace detail

// UNORDERED SET

#define BASE detail::kernel<\
    Key,\
    Key,\
    Hash,\
    extract::identity,\
    extract::identity,\
    KeyEqual,\
    Allocator>
template <
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<Key>>
class unordered_set : public BASE
{};
#undef BASE

// UNORDERED MAP

#define BASE detail::kernel<\
    std::pair<const Key, T>,\
    Key,\
    Hash,\
    extract::first,\
    extract::second,\
    KeyEqual,\
    Allocator>
template <
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<std::pair<const Key, T>>>
class unordered_map : public BASE
```

```
{
public:
    using mapped_type = T;
};
#undef BASE

} // namespace hopscotch
```

A.6 linear.hpp

```
#pragma once

#include <algorithm>
#include <bitset>
#include <cassert>
#include <cmath>
#include <cstddef>
#include <cstring>
#include <functional>
#include <initializer_list>
#include <iterator>
#include <memory>
#include <utility>
#include <vector>

#include <iostream>

namespace linear
{
namespace detail
{
// KERNEL

template <
    class Value,
    class Key,
    class Hash,
    class KeyExtract,
    class MappedExtract,
    class KeyEqual,
    class Allocator>
class kernel
{
protected:
    // Defined below.
    template <typename T> class iterator_template;

public:
    // MEMBER TYPES

    using key_type        = Key;
    using value_type      = Value;
    using size_type       = std::size_t;
    using difference_type = std::ptrdiff_t;
    using hasher          = Hash;
    using key_equal       = KeyEqual;
    using key_extract     = KeyExtract;
    using mapped_extract  = MappedExtract;
    using allocator_type  = Allocator;
```

```
    using reference       = value_type&;
    using const_reference = const value_type&;
    using pointer         =
        typename std::allocator_traits<Allocator>::pointer;
    using const_pointer   =
        typename std::allocator_traits<Allocator>::const_pointer;
    using iterator        = iterator_template<value_type>;
    using const_iterator  = iterator_template<const value_type>;

    // CONSTRUCTORS, ET CETERA

    // Default constructor.
    explicit kernel(
        size_t bucket_count = 16,
        const hasher& hash = hasher(),
        const key_equal& equal = key_equal(),
        const allocator_type& alloc = allocator_type())
      // Instances of functors.
    : hash_{hash},
      equal_{equal},
      extract_{},
      alloc_{alloc},
      // Bucket vector.
      buckets_{alloc_},
      min_load_{0.3},
      max_load_{0.7},
      // We always have a number of buckets that is a power of two.
      bucket_count_{upper_power_of_two(bucket_count)},
      // ...
      size_{0},
      min_size_{size_t(bucket_count_ * min_load_)},
      max_size_{size_t(bucket_count_ * max_load_)}
    {
        // Allocate space.
        buckets_.resize(bucket_count_);
    }

    // Copy constructor.
    kernel(
        const kernel& other)
    : hash_{other.hash_},
      equal_{other.equal_},
      extract_{},
      alloc_{other.alloc_},
      buckets_{other.buckets_},
      min_load_{other.min_load_},
      max_load_{other.max_load_},
      bucket_count_{other.bucket_count_},
      size_{other.size_},
      min_size_{other.min_size_},
      max_size_{other.max_size_}
    {}

    // Copy constructor with allocator.
    kernel(
        const kernel& other,
        const allocator_type& alloc)
    : hash_{other.hash_},
      equal_{other.equal_},
      extract_{},
      alloc_{alloc},
      buckets_{other.buckets_, alloc},
```

```
      min_load_{other.min_load_},
      max_load_{other.max_load_},
      bucket_count_{other.bucket_count_},
      size_{other.size_},
      min_size_{other.min_size_},
      max_size_{other.max_size_}
    {}

    // Move constructor.
    kernel(kernel&& other)
    : kernel{}
    {
        swap(*this, other);
    }

    // Swapping (member).
    void swap(kernel& other)
    {
        swap(*this, other);
    }

    // Swapping (friend).
    friend void swap(kernel& lhs, kernel& rhs)
    {
        using std::swap;
        swap(lhs.hash_, rhs.hash_);
        swap(lhs.equal_, rhs.equal_);
        swap(lhs.alloc_, rhs.alloc_);
        swap(lhs.buckets_, rhs.buckets_);
        swap(lhs.min_load_, rhs.min_load_);
        swap(lhs.max_load_, rhs.max_load_);
        swap(lhs.bucket_count_, rhs.bucket_count_);
        swap(lhs.size_, rhs.size_);
        swap(lhs.min_size_, rhs.min_size_);
        swap(lhs.max_size_, rhs.max_size_);
    }

    // Copy assignment.
    kernel& operator=(kernel other)
    {
        swap(*this, other);
        return *this;
    }

    // ITERATOR GETTERS

    iterator begin() {
        return make_iterator(buckets_begin());
    }

    const_iterator begin() const {
        return make_const_iterator(buckets_begin());
    }

    const_iterator cbegin() const {
        return make_const_iterator(buckets_begin());
    }

    iterator end() {
        return make_iterator(buckets_end());
    }

```

```
    const_iterator end() const {
        return make_const_iterator(buckets_end());
    }

    const_iterator cend() const {
        return make_const_iterator(buckets_end());
    }

    // CAPACITY

    bool empty() const {
        return size_ == 0;
    }

    size_t size() const {
        return size_;
    }

    // CLEAR

    void clear()
    {
        for (int index = 0; index < bucket_count_; ++index)
        {
            auto& bucket = buckets_[index];
            bucket.memory()->~value_type();
        }
        buckets_.clear();
        size_ = 0;
    }

    // ERASE

    size_t erase(const key_type& key)
    {
        size_t index = index_from_key(key);
        size_t erased_index;
        size_t erased_count = 0;

        while (buckets_[index].has_value) {
            auto& bucket = buckets_[index];

            if (equal_(key, extract_(*bucket.memory()))) {
                bucket.memory()->~value_type();
                bucket.has_value = false;
                erased_index = index;
                erased_count = 1;
                size_ -= 1;
                index = index_add(index, 1);
                break;
            }

            index = index_add(index, 1);
        }

        while (buckets_[index].has_value) {
            const auto& move_bucket = buckets_[index];
            const auto& hash = index_from_key(extract_(*move_bucket.memory()));

            if ((erased_index < index &&
                    (hash <= erased_index || hash > index)) ||
                (erased_index > index &&
```

```
                (hash <= erased_index && hash > index)))
        {
            auto& free_bucket = buckets_[erased_index];
            new (free_bucket.memory())
                value_type{std::move(*move_bucket.memory())};
            move_bucket.memory()->~value_type();

            buckets_[index].has_value = false;
            buckets_[erased_index].has_value = true;
            erased_index = index;
        }

        index = index_add(index, 1);
    }

    // Rehash if this brought us below min load.
    if (size_ < min_size_ && size_ > 16) {
        rehash(bucket_count_/2);
    }

    return erased_count;
}

// COUNT

size_t count(const Key& key) const
{
    size_t result = 0;

    const size_t virtual_index = index_from_key(key);

    while (buckets_[virtual_index]) {
        auto& value = buckets_[virtual_index];
        if (equal_(key, extract_(*value.memory())))
        {
            ++result;
        }
        virtual_index = index_add(virtual_index, 1);
    }

    return count;
}

// OPERATOR[]

auto& operator[](const Key& key) {
    return mapped_extract_(
        *(buckets_[find(key, index_from_key(key))].memory()));
}

const auto& operator[](const Key& key) const {
    return mapped_extract_(
        *(buckets_[find(key, index_from_key(key))].memory()));
}

// FIND

iterator find(const Key& key) {
    return make_iterator(find(key, index_from_key(key)));
}

const_iterator find(const Key& key) const {
```

```
        return make_const_iterator(find(key, index_from_key(key)));
    }

    // BUCKET INTERFACE

    size_t bucket_count() const {
        return bucket_count_;
    }

    // HASH POLICY

    float load_factor() const {
        return (float)size_/(float)bucket_count_;
    }

    float min_load_factor() const {
        return min_load_;
    }

    float max_load_factor() const {
        return max_load_;
    }

    void min_load_factor(float min_load)
    {
        min_load_ = min_load;

        // Check if we need to rehash.
        min_size_ = min_load_ * bucket_count_;
        if (size_ < min_size_) {
            rehash(bucket_count_/2);
        }
    }

    void max_load_factor(float max_load)
    {
        max_load_ = max_load;

        // Check if we need to rehash.
        max_size_ = max_load_ * bucket_count_;
        if (size_ > max_size_) {
            rehash(bucket_count_*2);
        }
    }

    void rehash(size_t count)
    {
        bucket_count_ = upper_power_of_two(count);

        // Create the new bucket vector, saving the old one.
        bucket_vector old_buckets(bucket_count_);
        std::swap(buckets_, old_buckets);

        // Set up the state so we can insert correctly.
        size_ = 0;
        min_size_ = min_load_ * bucket_count_;
        max_size_ = max_load_ * bucket_count_;

        // Rehash.
        for (size_t index = 0; index < old_buckets.size(); ++index)
        {
            if (old_buckets[index].has_value)
```

```
            {
                auto& bucket = old_buckets[index];
                insert(std::move(*bucket.memory()),
                    index_from_value(*bucket.memory()));
                // Detect recursive rehash and break.
                if (bucket_count_ != count) { break; }
            }
        }
    }

    void reserve(size_t count)
    {
        rehash(std::ceil((float)count/max_load_factor()));
    }

    // OBSERVERS

    hasher hash_function() const {
        return hash_;
    }

    key_equal key_eq() const {
        return equal_;
    }

    allocator_type get_allocator() const {
        return alloc_;
    }

protected:
    // BUCKET TYPE

    class bucket_type
    {
    public:
        value_type* memory() {
            return reinterpret_cast<value_type*>(&memory_);
        }

        const value_type* memory() const {
            return reinterpret_cast<const value_type*>(&memory_);
        }

        bool has_value;

    private:
        struct { unsigned char _[sizeof(value_type)]; } memory_;
    };

    using bucket_vector = std::vector<bucket_type,
          typename allocator_type::template rebind<bucket_type>::other>;

    // ITERATOR TYPE

    template <typename T>
    class iterator_template : std::iterator<std::forward_iterator_tag, T>
    {
        // NOT CONVERTED

        friend kernel;

    public:
```

```
        iterator_template()
        : index_(0),
          buckets_(nullptr)
        {}

        iterator_template(const iterator_template& other)
        : index_(other.index_),
          buckets_(other.buckets_)
        {}

        iterator_template& operator=(const iterator_template& other)
        {
            index_ = other.index_;
            buckets_ = other.buckets_;
            return *this;
        }

        ~iterator_template()
        {}

        reference operator*() {
            return *(buckets_->operator[](index_).memory());
            }

        const_reference operator*() const {
            return *(buckets_->operator[](index_).memory());
        }

        pointer operator->() {
            return buckets_->operator[](index_).memory();
        }

        const_pointer operator->() const {
            return buckets_->operator[](index_).memory();
        }

        iterator_template& operator++()
        {
            do {
                ++index_;
            }
            while (index_ != buckets_->size() &&
                !buckets_->operator[](index_).has_value);
            return *this;
        }

        iterator_template operator++(int)
        {
            iterator_template old(*this);
            ++*this;
            return old;
        }

        bool operator==(const iterator_template& other) const {
            return index_ == other.index_ && buckets_ == other.buckets_;
        }

        bool operator!=(const iterator_template& other) const {
            return index_ != other.index_ || buckets_ != other.buckets_;
        }

    private:
```

```
        iterator_template(
            size_t index,
            bucket_vector* buckets)
        : index_(index),
          buckets_(buckets)
        {}

        size_t index_;

        bucket_vector* buckets_;
    };

    // DATA MEMBERS

    hasher hash_;
    key_equal equal_;
    key_extract extract_;
    mapped_extract mapped_extract_;
    allocator_type alloc_;

    bucket_vector buckets_;
    float min_load_;
    float max_load_;

    size_t bucket_count_;

    size_t size_;
    size_t min_size_;
    size_t max_size_;

    // UPPER POWER OF TWO

    size_t upper_power_of_two(size_t x) const
    {
        // This implementation was found here (adjusted for 64-bit):
        // http://graphics.stanford.edu/~seander/bithacks.html#RoundUpPowerOf2
        --x;
        x |= x >> 1;
        x |= x >> 2;
        x |= x >> 4;
        x |= x >> 8;
        x |= x >> 16;
        x |= x >> 32;
        ++x;
        return x;
    }

    // INDEX HELPER

    size_t index_from_value(const value_type& value) const {
        return index_from_key(extract_(value));
    }

    size_t index_from_key(const key_type& key) const {
        // This bitwise and is the same as doing a modulo because the bucket
        // count is guaranteed to be a power of two.
        return hash_(key) & (bucket_count_ - 1);
    }

    size_t index_add(size_t index, size_t x) const {
        // This bitwise and is the same as doing a modulo because the bucket
        // count is guaranteed to be a power of two.
```

```
        return (index + x) & (bucket_count_ - 1);
    }

    size_t index_sub(size_t index, size_t x) const {
        // As above, this corresponds to a modulo operation, except that we
        // always get a positive value this way (as is desired).
        return (index - x) & (bucket_count_ - 1);
    }

    // ITERATOR HELPERS

    size_t buckets_begin()
    {
        if (empty()) {
            return buckets_.size();
        } else {
            auto bucket_it = buckets_.begin();
            while (!bucket_it->has_value) { ++bucket_it; }
            return bucket_it - buckets_.begin();
        }
    }

    size_t buckets_end() {
        return buckets_.size();
    }

    iterator make_iterator(size_t index) {
        return {index, &buckets_};
    }

    const_iterator make_const_iterator(size_t index) const {
        return {index, &buckets_};
    }

public:
    // INSERT

    std::pair<iterator, bool>
    insert(value_type value)
    {
        size_t index = index_from_value(value);
        size_t res = find(extract_(value), index);

        if (res == buckets_.size()) {
            return {insert(std::move(value), index), true};
        } else {
            return {make_iterator(res), false};
        }
    }

    iterator
    insert(
        const_iterator hint,
        value_type value)
    {
        (void)hint;  // Silence 'unused parameter'
        return insert(std::move(value)).first;
    }

private:
    // INSERT IMPLEMENTATION

```

```
    iterator insert(value_type value, size_t virtual_index)
    {
        // Start by rehashing, if this will bring us above max load.
        if (size_ == max_size_) {
            rehash(bucket_count_*2);
            return insert(value, index_from_value(value));
        }

        // Find the nearest free bucket, wrapping if we move past the end.
        size_t free_dist = 0;
        size_t free_index = virtual_index;
        while (buckets_[free_index].has_value) {
            free_dist += 1;
            free_index = index_add(free_index, 1);
        }

        // We should have a free bucket in the neighborhood now.
        auto& free_bucket = buckets_[free_index];
        new (free_bucket.memory()) value_type{std::move(value)};

        buckets_[free_index].has_value = true;

        ++size_;

        return {free_index, &buckets_};
    }

    // FIND IMPLEMENTATION

    size_t find(const key_type& key, size_t virtual_index) const
    {
        // Search from there until we find what we are looking for or an
        // empty bucket
        while (buckets_[virtual_index].has_value) {
            const auto& value = buckets_[virtual_index];
            if (equal_(key, extract_(*value.memory()))) {
                return virtual_index;
            }
            virtual_index = index_add(virtual_index, 1);
        }

        // We found nothing, return end.
        return bucket_count_;
    }
};

} // namespace detail

// UNORDERED SET

#define BASE detail::kernel<\
    Key,\
    Key,\
    Hash,\
    extract::identity,\
    extract::identity,\
    KeyEqual,\
    Allocator>
template <
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
```

```
    class Allocator = std::allocator<Key>>
class unordered_set : public BASE
{};
#undef BASE

// UNORDERED MAP

#define BASE detail::kernel<\
    std::pair<const Key, T>,\
    Key,\
    Hash,\
    extract::first,\
    extract::second,\
    KeyEqual,\
    Allocator>
template <
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<std::pair<const Key, T>>>
class unordered_map : public BASE
{
public:
    using mapped_type = T;
};
#undef BASE

} // namespace hopscotch
```

A.7 tabulation.hpp

```
#pragma once

#include <algorithm>
#include <bitset>
#include <cassert>
#include <cmath>
#include <cstddef>
#include <cstring>
#include <functional>
#include <initializer_list>
#include <iterator>
#include <memory>
#include <utility>
#include <vector>

#include <iostream>

namespace linear
{
namespace detail
{
// KERNEL

template <
    class Value,
    class Key,
    class Hash,
    class KeyExtract,
    class MappedExtract,
```

```
    class KeyEqual,
    class Allocator>
class kernel
{
protected:
    // Defined below.
    template <typename T> class iterator_template;

public:
    // MEMBER TYPES

    using key_type        = Key;
    using value_type      = Value;
    using size_type       = std::size_t;
    using difference_type = std::ptrdiff_t;
    using hasher          = Hash;
    using key_equal       = KeyEqual;
    using key_extract     = KeyExtract;
    using mapped_extract  = MappedExtract;
    using allocator_type  = Allocator;
    using reference       = value_type&;
    using const_reference = const value_type&;
    using pointer         =
        typename std::allocator_traits<Allocator>::pointer;
    using const_pointer   =
        typename std::allocator_traits<Allocator>::const_pointer;
    using iterator        = iterator_template<value_type>;
    using const_iterator  = iterator_template<const value_type>;

    // CONSTRUCTORS, ET CETERA

    // Default constructor.
    explicit kernel(
        size_t bucket_count = 16,
        const hasher& hash = hasher(),
        const key_equal& equal = key_equal(),
        const allocator_type& alloc = allocator_type())
      // Instances of functors.
    : hash_{hash},
      equal_{equal},
      extract_{},
      alloc_{alloc},
      // Bucket vector.
      buckets_{alloc_},
      min_load_{0.3},
      max_load_{0.7},
      // We always have a number of buckets that is a power of two.
      bucket_count_{upper_power_of_two(bucket_count)},
      // ...
      size_{0},
      min_size_{size_t(bucket_count_ * min_load_)},
      max_size_{size_t(bucket_count_ * max_load_)}
    {
        // Allocate space.
        buckets_.resize(bucket_count_);
    }

    // Copy constructor.
    kernel(
        const kernel& other)
    : hash_{other.hash_},
      equal_{other.equal_},
```

```
      extract_{},
      alloc_{other.alloc_},
      buckets_{other.buckets_},
      min_load_{other.min_load_},
      max_load_{other.max_load_},
      bucket_count_{other.bucket_count_},
      size_{other.size_},
      min_size_{other.min_size_},
      max_size_{other.max_size_}
    {}

    // Copy constructor with allocator.
    kernel(
        const kernel& other,
        const allocator_type& alloc)
    : hash_{other.hash_},
      equal_{other.equal_},
      extract_{},
      alloc_{alloc},
      buckets_{other.buckets_, alloc},
      min_load_{other.min_load_},
      max_load_{other.max_load_},
      bucket_count_{other.bucket_count_},
      size_{other.size_},
      min_size_{other.min_size_},
      max_size_{other.max_size_}
    {}

    // Move constructor.
    kernel(kernel&& other)
    : kernel{}
    {
        swap(*this, other);
    }

    // Swapping (member).
    void swap(kernel& other)
    {
        swap(*this, other);
    }

    // Swapping (friend).
    friend void swap(kernel& lhs, kernel& rhs)
    {
        using std::swap;
        swap(lhs.hash_, rhs.hash_);
        swap(lhs.equal_, rhs.equal_);
        swap(lhs.alloc_, rhs.alloc_);
        swap(lhs.buckets_, rhs.buckets_);
        swap(lhs.min_load_, rhs.min_load_);
        swap(lhs.max_load_, rhs.max_load_);
        swap(lhs.bucket_count_, rhs.bucket_count_);
        swap(lhs.size_, rhs.size_);
        swap(lhs.min_size_, rhs.min_size_);
        swap(lhs.max_size_, rhs.max_size_);
    }

    // Copy assignment.
    kernel& operator=(kernel other)
    {
        swap(*this, other);
        return *this;
```

```
    }

    // ITERATOR GETTERS

    iterator begin() {
        return make_iterator(buckets_begin());
    }

    const_iterator begin() const {
        return make_const_iterator(buckets_begin());
    }

    const_iterator cbegin() const {
        return make_const_iterator(buckets_begin());
    }

    iterator end() {
        return make_iterator(buckets_end());
    }

    const_iterator end() const {
        return make_const_iterator(buckets_end());
    }

    const_iterator cend() const {
        return make_const_iterator(buckets_end());
    }

    // CAPACITY

    bool empty() const {
        return size_ == 0;
    }

    size_t size() const {
        return size_;
    }

    // CLEAR

    void clear()
    {
        for (int index = 0; index < bucket_count_; ++index)
        {
            auto& bucket = buckets_[index];
            bucket.memory()->~value_type();
        }
        buckets_.clear();
        size_ = 0;
    }

    // ERASE

    size_t erase(const key_type& key)
    {
        size_t index = index_from_key(key);
        size_t erased_index;
        size_t erased_count = 0;

        while (buckets_[index].has_value) {
            auto& bucket = buckets_[index];

```

```
            if (equal_(key, extract_(*bucket.memory()))) {
                bucket.memory()->~value_type();
                bucket.has_value = false;
                erased_index = index;
                erased_count = 1;
                size_ -= 1;
                index = index_add(index, 1);
                break;
            }

            index = index_add(index, 1);
        }

        while (buckets_[index].has_value) {
            const auto& move_bucket = buckets_[index];
            const auto& hash = index_from_key(extract_(*move_bucket.memory()));

            if ((erased_index < index &&
                    (hash <= erased_index || hash > index)) ||
                (erased_index > index &&
                    (hash <= erased_index && hash > index)))
            {
                auto& free_bucket = buckets_[erased_index];
                new (free_bucket.memory())
                    value_type{std::move(*move_bucket.memory())};
                move_bucket.memory()->~value_type();

                buckets_[index].has_value = false;
                buckets_[erased_index].has_value = true;
                erased_index = index;
            }

            index = index_add(index, 1);
        }

        // Rehash if this brought us below min load.
        if (size_ < min_size_ && size_ > 16) {
            rehash(bucket_count_/2);
        }

        return erased_count;
    }

    // COUNT

    size_t count(const Key& key) const
    {
        size_t result = 0;

        const size_t virtual_index = index_from_key(key);

        while (buckets_[virtual_index]) {
            auto& value = buckets_[virtual_index];
            if (equal_(key, extract_(*value.memory())))
            {
                ++result;
            }
            virtual_index = index_add(virtual_index, 1);
        }

        return count;
    }
```

```

    // OPERATOR[]

    auto& operator[](const Key& key) {
        return mapped_extract_(
            *(buckets_[find(key, index_from_key(key))].memory()));
    }

    const auto& operator[](const Key& key) const {
        return mapped_extract_(
            *(buckets_[find(key, index_from_key(key))].memory()));
    }

    // FIND

    iterator find(const Key& key) {
        return make_iterator(find(key, index_from_key(key)));
    }

    const_iterator find(const Key& key) const {
        return make_const_iterator(find(key, index_from_key(key)));
    }

    // BUCKET INTERFACE

    size_t bucket_count() const {
        return bucket_count_;
    }

    // HASH POLICY

    float load_factor() const {
        return (float)size_/(float)bucket_count_;
    }

    float min_load_factor() const {
        return min_load_;
    }

    float max_load_factor() const {
        return max_load_;
    }

    void min_load_factor(float min_load)
    {
        min_load_ = min_load;

        // Check if we need to rehash.
        min_size_ = min_load_ * bucket_count_;
        if (size_ < min_size_) {
            rehash(bucket_count_/2);
        }
    }

    void max_load_factor(float max_load)
    {
        max_load_ = max_load;

        // Check if we need to rehash.
        max_size_ = max_load_ * bucket_count_;
        if (size_ > max_size_) {
            rehash(bucket_count_*2);
```

```
        }
    }

    void rehash(size_t count)
    {
        bucket_count_ = upper_power_of_two(count);

        // Create the new bucket vector, saving the old one.
        bucket_vector old_buckets(bucket_count_);
        std::swap(buckets_, old_buckets);

        // Set up the state so we can insert correctly.
        size_ = 0;
        min_size_ = min_load_ * bucket_count_;
        max_size_ = max_load_ * bucket_count_;

        // Rehash.
        for (size_t index = 0; index < old_buckets.size(); ++index)
        {
            if (old_buckets[index].has_value)
            {
                auto& bucket = old_buckets[index];
                insert(std::move(*bucket.memory()),
                    index_from_value(*bucket.memory()));
                // Detect recursive rehash and break.
                if (bucket_count_ != count) { break; }
            }
        }
    }

    void reserve(size_t count)
    {
        rehash(std::ceil((float)count/max_load_factor()));
    }

    // OBSERVERS

    hasher hash_function() const {
        return hash_;
    }

    key_equal key_eq() const {
        return equal_;
    }

    allocator_type get_allocator() const {
        return alloc_;
    }

protected:
    // BUCKET TYPE

    class bucket_type
    {
    public:
        value_type* memory() {
            return reinterpret_cast<value_type*>(&memory_);
        }

        const value_type* memory() const {
            return reinterpret_cast<const value_type*>(&memory_);
        }
```

```

        bool has_value;

    private:
        struct { unsigned char _[sizeof(value_type)]; } memory_;
    };

    using bucket_vector = std::vector<bucket_type,
          typename allocator_type::template rebind<bucket_type>::other>;

    // ITERATOR TYPE

    template <typename T>
    class iterator_template : std::iterator<std::forward_iterator_tag, T>
    {
        // NOT CONVERTED

        friend kernel;

    public:
        iterator_template()
        : index_(0),
          buckets_(nullptr)
        {}

        iterator_template(const iterator_template& other)
        : index_(other.index_),
          buckets_(other.buckets_)
        {}

        iterator_template& operator=(const iterator_template& other)
        {
            index_ = other.index_;
            buckets_ = other.buckets_;
            return *this;
        }

        ~iterator_template()
        {}

        reference operator*() {
            return *(buckets_->operator[](index_).memory());
            }

        const_reference operator*() const {
            return *(buckets_->operator[](index_).memory());
        }

        pointer operator->() {
            return buckets_->operator[](index_).memory();
        }

        const_pointer operator->() const {
            return buckets_->operator[](index_).memory();
        }

        iterator_template& operator++()
        {
            do {
                ++index_;
            }
            while (index_ != buckets_->size() &&
```

```
                !buckets_->operator[](index_).has_value);
            return *this;
        }

        iterator_template operator++(int)
        {
            iterator_template old(*this);
            ++*this;
            return old;
        }

        bool operator==(const iterator_template& other) const {
            return index_ == other.index_ && buckets_ == other.buckets_;
        }

        bool operator!=(const iterator_template& other) const {
            return index_ != other.index_ || buckets_ != other.buckets_;
        }

    private:
        iterator_template(
            size_t index,
            bucket_vector* buckets)
        : index_(index),
          buckets_(buckets)
        {}

        size_t index_;

        bucket_vector* buckets_;
    };

    // DATA MEMBERS

    hasher hash_;
    key_equal equal_;
    key_extract extract_;
    mapped_extract mapped_extract_;
    allocator_type alloc_;

    bucket_vector buckets_;
    float min_load_;
    float max_load_;

    size_t bucket_count_;

    size_t size_;
    size_t min_size_;
    size_t max_size_;

    // UPPER POWER OF TWO

    size_t upper_power_of_two(size_t x) const
    {
        // This implementation was found here (adjusted for 64-bit):
        // http://graphics.stanford.edu/~seander/bithacks.html#RoundUpPowerOf2
        --x;
        x |= x >> 1;
        x |= x >> 2;
        x |= x >> 4;
        x |= x >> 8;
        x |= x >> 16;
```

```
        x |= x >> 32;
        ++x;
        return x;
    }

    // INDEX HELPER

    size_t index_from_value(const value_type& value) const {
        return index_from_key(extract_(value));
    }

    size_t index_from_key(const key_type& key) const {
        // This bitwise and is the same as doing a modulo because the bucket
        // count is guaranteed to be a power of two.
        return hash_(key) & (bucket_count_ - 1);
    }

    size_t index_add(size_t index, size_t x) const {
        // This bitwise and is the same as doing a modulo because the bucket
        // count is guaranteed to be a power of two.
        return (index + x) & (bucket_count_ - 1);
    }

    size_t index_sub(size_t index, size_t x) const {
        // As above, this corresponds to a modulo operation, except that we
        // always get a positive value this way (as is desired).
        return (index - x) & (bucket_count_ - 1);
    }

    // ITERATOR HELPERS

    size_t buckets_begin()
    {
        if (empty()) {
            return buckets_.size();
        } else {
            auto bucket_it = buckets_.begin();
            while (!bucket_it->has_value) { ++bucket_it; }
            return bucket_it - buckets_.begin();
        }
    }

    size_t buckets_end() {
        return buckets_.size();
    }

    iterator make_iterator(size_t index) {
        return {index, &buckets_};
    }

    const_iterator make_const_iterator(size_t index) const {
        return {index, &buckets_};
    }

public:
    // INSERT

    std::pair<iterator, bool>
    insert(value_type value)
    {
        size_t index = index_from_value(value);
        size_t res = find(extract_(value), index);
```

```

        if (res == buckets_.size()) {
            return {insert(std::move(value), index), true};
        } else {
            return {make_iterator(res), false};
        }
    }

    iterator
    insert(
        const_iterator hint,
        value_type value)
    {
        (void)hint;  // Silence 'unused parameter'
        return insert(std::move(value)).first;
    }

private:
    // INSERT IMPLEMENTATION

    iterator insert(value_type value, size_t virtual_index)
    {
        // Start by rehashing, if this will bring us above max load.
        if (size_ == max_size_) {
            rehash(bucket_count_*2);
            return insert(value, index_from_value(value));
        }

        // Find the nearest free bucket, wrapping if we move past the end.
        size_t free_dist = 0;
        size_t free_index = virtual_index;
        while (buckets_[free_index].has_value) {
            free_dist += 1;
            free_index = index_add(free_index, 1);
        }

        // We should have a free bucket in the neighborhood now.
        auto& free_bucket = buckets_[free_index];
        new (free_bucket.memory()) value_type{std::move(value)};

        buckets_[free_index].has_value = true;

        ++size_;

        return {free_index, &buckets_};
    }

    // FIND IMPLEMENTATION

    size_t find(const key_type& key, size_t virtual_index) const
    {
        // Search from there until we find what we are looking for or an
        // empty bucket
        while (buckets_[virtual_index].has_value) {
            const auto& value = buckets_[virtual_index];
            if (equal_(key, extract_(*value.memory()))) {
                return virtual_index;
            }
            virtual_index = index_add(virtual_index, 1);
        }

        // We found nothing, return end.
```

```
        return bucket_count_;
    }
};

} // namespace detail

// UNORDERED SET

#define BASE detail::kernel<\
    Key,\
    Key,\
    Hash,\
    extract::identity,\
    extract::identity,\
    KeyEqual,\
    Allocator>
template <
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<Key>>
class unordered_set : public BASE
{};
#undef BASE

// UNORDERED MAP

#define BASE detail::kernel<\
    std::pair<const Key, T>,\
    Key,\
    Hash,\
    extract::first,\
    extract::second,\
    KeyEqual,\
    Allocator>
template <
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<std::pair<const Key, T>>>
class unordered_map : public BASE
{
public:
    using mapped_type = T;
};
#undef BASE

} // namespace hopscotch
```

Appendix B

Test code

B.1 correctness_test.hpp

```
#include "double_tree.hpp"
#include "tabulation.hpp"
#include "hopscotch.hpp"
#include "linear.hpp"
#include "longrand.hpp"

#include <algorithm>
#include <cassert>
#include <iostream>
#include <stx/btree_map.h>

using std::cerr;
using std::cout;
using std::endl;
using std::flush;
using std::pair;

#define DOUBLE_BTREE
// #define SLOW_INSERT
// #define SLOW_ERASE

template <typename K, typename V, size_t nodesize>
class btree_traits
{
public:
    static const bool selfverify = false;
    static const bool debug = false;
    static const int leafslots = nodesize / (sizeof(K) + sizeof(V));
    static const int innerslots = nodesize / (sizeof(K) + sizeof(void*));
    static const size_t binsearch_threshold = 256;
};

int main(int, char**)
{
    const size_t count = 1000000;

#if defined(HOPSCOTCH)
    hopscotch::unordered_map<uint64_t, uint64_t, tabulation<uint64_t>> map;
#elif defined(LINEAR)
```

```
    linear::unordered_map<uint64_t, uint64_t, tabulation<uint64_t>> map;
#elif defined(DOUBLE_BTREE)
    double_tree::map<uint64_t, uint64_t> map;
#endif

    // Insertion test
    srand(19);
    for (size_t i = 0; i < count; ++i)
    {
        cerr << "\rinsert␣" << i << "/" << count << flush;

        uint64_t key = rand();
        uint64_t val = rand();

        map.insert({key, val});
#if !defined(DOUBLE_BTREE)
        assert(map.size() == i + 1);
#endif

#if defined(SLOW_INSERT)
        srand(19);
        for (size_t j = 0; j < i + 1; ++j) {
            uint64_t key = rand();
            uint64_t val = rand();
            if (map[key] != val) {
                cout << "while␣inserting␣" << key << endl;
                cout << i << ":␣could␣not␣find␣" << val << "␣(" << j <<
                    ").␣found␣" << map[key] << endl;
                assert(false);
            }
        }
#endif  // SLOW_INSERT
    }

    // Iterator test
#if defined(DOUBLE_BTREE) || defined(TRAD_BTREE_CACHE) \\
    || defined(TRAD_BTREE_DISK)
    int itnr = 0;
    uint64_t prev;
    for (auto it = map.begin(); it != map.end(); ++it) {
        cerr << "\riterate␣" << itnr << "/" << count << flush;
        ++itnr;

        if (it != map.begin()) {
            assert(it->first > prev);
        }
        prev = it->first;
    }
#endif

    // Find test
    srand(19);
    for (size_t i = 0; i < count; ++i)
    {
        cerr << "\rfind␣" << i << "/" << count << flush;

        uint64_t key = rand();
        uint64_t val = rand();

        if (map[key] != val) {
            cout << endl << "could␣not␣find␣" << val <<"." << endl
                 << "found␣" << map[key] << "." << endl;
```

```
            assert(false);
        }
    }

    srand(19);
    for (size_t i = 0; i < count; ++i)
    {
        cerr << "\rerase␣" << i << "/" << count << flush;

        uint64_t key = rand();
        rand();

        map.erase(key);

#if defined(SLOW_ERASE)
        srand(19);

        for (size_t j = 0; j < i + 1; ++j) {
            rand(); rand();
        }
        for (size_t j = i + 1; j < count; ++j) {
            uint64_t key = rand();
            uint64_t val = rand();
            if (map[key] != val) {
                cout << i << ":␣could␣not␣find␣" << key << "␣(" << j <<
                    ").␣found␣" << map[key].first << endl;
                assert(false);
            }
        }
        for (size_t j = 0; j < i + 1; ++j) {
            rand(); rand();
        }
#endif  // SLOW_ERASE
    }

    assert(map.empty());
}
```

B.2 hopscotch_experiment.hpp

```
#include "tabulation.hpp"
#include <ctime>
#include <iostream>
#include <random>
#include <vector>
using std::vector;

// #define TABULATION

class stripped_hopscotch
{
protected:
    static const size_t neighborhood_size = 15;

public:
    explicit stripped_hopscotch(size_t bucket_count):
        bucket_count_{bucket_count},
        size_{0}
    {
        buckets_.resize(bucket_count_);
    }
```

```

    float load_factor() const {
        return (float)size_/(float)bucket_count_;
    }

    struct bucket_type
    {
        void has_value(bool has_value) {
            hop_info[neighborhood_size] = has_value;
        }

        bool has_value() const {
            return hop_info[neighborhood_size];
        }

        std::bitset<neighborhood_size+1> hop_info;
    };

    std::vector<bucket_type> buckets_;
    size_t bucket_count_;
    size_t size_;

    size_t index_add(size_t index, size_t x) const {
        return (index + x) & (bucket_count_ - 1);
    }

    size_t index_sub(size_t index, size_t x) const {
        return (index - x) & (bucket_count_ - 1);
    }

    size_t next_hop(const bucket_type& bucket, int prev = -1) const
    {
        const size_t mask = 0xffffffffffffffff << (prev + 1);
        const size_t hop_info = bucket.hop_info.to_ulong();
        return __builtin_ffsl(hop_info & mask) - 1;
    }

    bool insert(size_t virtual_index)
    {
        virtual_index &= (bucket_count_ - 1);

        // Find the nearest free bucket, wrapping if we move past the end.
        size_t free_dist = 0;
        size_t free_index = virtual_index;
        while (buckets_[free_index].has_value()) {
            free_dist += 1;
            free_index = index_add(free_index, 1);
        }

        // Move buckets until we have a free bucket in the neighborhood of our
        // virtual bucket.
        while (free_dist > neighborhood_size - 1)
        {
            // Find a virtual bucket that has values stored in a bucket before
            // the free bucket we found.
            size_t virtual_move_dist = neighborhood_size - 1;
            size_t virtual_move_index =
                index_sub(free_index, virtual_move_dist);

            size_t move_hop;

            while (true)
```

```
            {
                auto& virtual_move_bucket = buckets_[virtual_move_index];
                auto hop_info = virtual_move_bucket.hop_info.to_ulong();
                move_hop = __builtin_ffsl(hop_info) - 1;

                if (move_hop < virtual_move_dist) {
                    break;
                } else {
                    // No luck, continue searching.
                    virtual_move_dist -= 1;
                    virtual_move_index = index_add(virtual_move_index, 1);

                    if (virtual_move_dist == 0)
                    {
                        return false;
                    }
                }
            }

            // Move.
            const size_t move_dist = virtual_move_dist - move_hop;
            const size_t move_index = index_add(virtual_move_index, move_hop);

            buckets_[move_index].has_value(false);
            buckets_[free_index].has_value(true);

            auto& virtual_move = buckets_[virtual_move_index];
            virtual_move.hop_info[move_hop] = false;
            virtual_move.hop_info[virtual_move_dist] = true;

            // The free bucket is now in the position of the moved bucket.
            free_dist -= move_dist;
            free_index = index_sub(free_index, move_dist);
        }

        // We should have a free bucket in the neighborhood now.
        buckets_[free_index].has_value(true);
        buckets_[virtual_index].hop_info[free_dist] = true;
        ++size_;
        return true;
    }
};

int main(int, char**)
{
    for (size_t exp = 8; exp <= 30; ++exp) {
        const size_t count = static_cast<size_t>(1) << exp;
        stripped_hopscotch hs{count};

#ifdef TABULATION
        vector<uint64_t> elements(count);
        for (size_t i = 0; i < count; ++i) { elements[i] = i; }
        std::srand(std::time(nullptr));
        std::random_shuffle(elements.begin(), elements.end());
        tabulation<uint64_t> t;
#else
        std::mt19937_64 mt(std::time(nullptr));
#endif

        for (size_t i = 0; i < count; ++i) {
#ifdef TABULATION
            if (!hs.insert(t(elements[i]))) {
```

```
#else
            if (!hs.insert(mt())) {
#endif
                std::cout << "\r"
                          << exp << "␣"
                          << i << "/" << count << "␣"
                          << (float)i/(float)count << std::endl;
                break;
            } else {
                for (int p = 0; p < 10; ++p) {
                    if (i == p*count/10) {
                        std::cerr << "\r0." << p;
                    }
                }
            }
        }
    }
}
```

B.3 performance_clock.hpp

```
#pragma once

#include <chrono>
#include <cstdint>

namespace performance_clock
{
    class interval
    {
    public:
        void before();
        void after();

        uint64_t wall_time() const { return wall_time_; }
        uint64_t usr_time() const { return usr_time_; }
        uint64_t sys_time() const { return sys_time_; }
        // uint64_t min_faults() const { return min_faults_; }
        // uint64_t maj_faults() const { return maj_faults_; }

    private:
        using time_point =
            std::chrono::time_point<std::chrono::high_resolution_clock>;

        time_point wall_time_before_;
        uint64_t usr_time_before_;
        uint64_t sys_time_before_;
        // uint64_t min_faults_before_;
        // uint64_t maj_faults_before_;

        uint64_t wall_time_;
        uint64_t usr_time_;
        uint64_t sys_time_;
        // uint64_t min_faults_;
        // uint64_t maj_faults_;
    };
}
```

B.4 performance_clock.cpp

```
#include "performance_clock.hpp"

#include <sys/resource.h>

using std::chrono::time_point;
using std::chrono::high_resolution_clock;
using std::chrono::duration_cast;
using std::chrono::nanoseconds;

namespace performance_clock
{
    void interval::before()
    {
        // Record user time, system time, page faults
        rusage r_usage;
        getrusage(RUSAGE_SELF, &r_usage);
        usr_time_before_ =
            r_usage.ru_utime.tv_sec * static_cast<uint64_t>(1000000000) +
            r_usage.ru_utime.tv_usec * static_cast<uint64_t>(1000);
        sys_time_before_ =
            r_usage.ru_stime.tv_sec * static_cast<uint64_t>(1000000000) +
            r_usage.ru_stime.tv_usec * static_cast<uint64_t>(1000);
        // min_faults_before_ = r_usage.ru_minflt;
        // maj_faults_before_ = r_usage.ru_majflt;

        // Record wall time from std::chrono
        wall_time_before_ = high_resolution_clock::now();
    }

    void interval::after()
    {
        // Record wall time from std::chrono
        wall_time_ = duration_cast<nanoseconds>(
            high_resolution_clock::now() - wall_time_before_).count();

        // Record user time, system time, page faults
        rusage r_usage;
        getrusage(RUSAGE_SELF, &r_usage);
        usr_time_ =
            r_usage.ru_utime.tv_sec * static_cast<uint64_t>(1000000000) +
            r_usage.ru_utime.tv_usec * static_cast<uint64_t>(1000)
            - usr_time_before_;
        sys_time_ =
            r_usage.ru_stime.tv_sec * static_cast<uint64_t>(1000000000) +
            r_usage.ru_stime.tv_usec * static_cast<uint64_t>(1000)
            - sys_time_before_;
        // min_faults_ = r_usage.ru_minflt - min_faults_before_;
        // maj_faults_ = r_usage.ru_majflt - maj_faults_before_;
    }
}
```

B.5 performance_test.hpp

```
#include "tabulation.hpp"

#include "hopscotch.hpp"
#include "linear.hpp"

#include "performance_clock.hpp"
#include "longrand.hpp"

```

```
#include <algorithm>
#include <cassert>
#include <iomanip>
#include <iostream>
#include <vector>

#include <stx/btree_map.h>

using std::cerr;
using std::cout;
using std::endl;
using std::flush;
using std::pair;
using std::vector;

// #define DENSE
#define TRAD_BTREE_CACHE

template <typename K, typename V, size_t nodesize>
class btree_traits
{
public:
    static const bool selfverify = false;
    static const bool debug = false;
    static const int leafslots = nodesize / (sizeof(K) + sizeof(V));
    static const int innerslots = nodesize / (sizeof(K) + sizeof(void*));
    static const size_t binsearch_threshold = 256;
};

int main(int, char**)
{
    const size_t count = static_cast<size_t>(1)<<26;

    srand(35);
    vector<uint64_t> elements(count);

#ifdef DENSE
    // Create a list of randomly ordered numbers from a dense interval (0..count)
    for (size_t i = 0; i < count; ++i) { elements[i] = i; }
    std::random_shuffle(elements.begin(), elements.end());
#else
    // Create a list of random numbers
    for (size_t i = 0; i < count; ++i) { elements[i] = longrand(); }
#endif

    // Create a map with the chosen collision resolution method and hash function
#if defined(LINEAR)
    linear::unordered_map<uint64_t, uint64_t, tabulation<uint64_t>> map;
#elif defined(HOPSCOTCH)
    hopscotch::unordered_map<uint64_t, uint64_t, tabulation<uint64_t>> map;
#elif defined(DOUBLE_BTREE)
    double_tree::map<uint64_t, uint64_t> map;
#elif defined(TRAD_BTREE_CACHE)
    stx::btree_map<uint64_t, uint64_t, std::less<uint64_t>,
        btree_traits<uint64_t, uint64_t, 256>> map;
#elif defined(TRAD_BTREE_DISK)
    stx::btree_map<uint64_t, uint64_t, std::less<uint64_t>,
        btree_traits<uint64_t, uint64_t, 4096>> map;
#endif

```

```
    cout << std::fixed << std::setprecision(0);

    const size_t round_count = static_cast<size_t>(1)<<18;
    const size_t rounds = count/round_count;

    for (size_t i = 0; i < rounds; ++i) {
        // Insert
        performance_clock::interval insert_interval;
        insert_interval.before();
        for (size_t j = 0; j < round_count; ++j) {
            map.insert(std::make_pair(
                elements[i*round_count + j], i*round_count + j));
        }
        insert_interval.after();
        cout << "insert\t" << i;
        cout << "\t" << insert_interval.wall_time()/(double)round_count;
        cout << "\t" << insert_interval.usr_time()/(double)round_count;
        cout << "\t" << insert_interval.sys_time()/(double)round_count;
        cout << endl;

        // Find
        performance_clock::interval search_interval;
        search_interval.before();
        size_t search_i = rand()%(i+1);
        for (size_t j = 0; j < round_count; ++j) {
            volatile auto _ = map[elements[search_i*round_count + j]];
            (void)_;  // Silence 'unused variable'
        }
        search_interval.after();
        cout << "search\t" << i;
        cout << "\t" << search_interval.wall_time()/(double)round_count;
        cout << "\t" << search_interval.usr_time()/(double)round_count;
        cout << "\t" << search_interval.sys_time()/(double)round_count;
        cout << endl;

#if defined(DOUBLE_BTREE) || defined(TRAD_BTREE_CACHE) \\
        || defined(TRAD_BTREE_DISK)
        // Iterate
        performance_clock::interval iterate_interval;
        iterate_interval.before();
        size_t iterate_i = rand()%(i+1);
        size_t iterate_j = rand()%(round_count);
        auto it = map.find(map[elements[iterate_i*round_count + iterate_j]]);
        for (size_t j = 0; j < round_count; ++j) {
            volatile auto _ = *it;
            (void)_;  // Silence 'unused variable'
            ++it;
            if (it == map.end()) {
                it = map.begin();
            }
        }
        iterate_interval.after();
        cout << "iterate\t" << i;
        cout << "\t" << iterate_interval.wall_time()/(double)round_count;
        cout << "\t" << iterate_interval.usr_time()/(double)round_count;
        cout << "\t" << iterate_interval.sys_time()/(double)round_count;
        cout << endl;
#endif
    }

    // Shuffle the elements for erasing
    std::random_shuffle(elements.begin(), elements.end());
```

```
    for (size_t i = 0; i < rounds; ++i) {
        // Erase
        performance_clock::interval erase_interval;
        erase_interval.before();
        for (size_t j = 0; j < round_count; ++j) {
            map.erase(elements[i*round_count + j]);
        }
        erase_interval.after();
        cout << "erase\t" << i;
        cout << "\t" << erase_interval.wall_time()/(double)round_count;
        cout << "\t" << erase_interval.usr_time()/(double)round_count;
        cout << "\t" << erase_interval.sys_time()/(double)round_count;
        cout << endl;
    }
}
```