NTNU – Trondheim
Norwegian University of
Science and Technology

Representing sets in C++

A practical investigation

Lars Greger Nordland Hagen

Master of Science in Computer Science
Submission date: February 2014
Supervisor: Magnus Lie Hetland, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

To Ingeborg,
for loving support and
patience during my thesis work.

Summary

The standard C++ classes for storing ordered sets and maps were created at a time when the latencies of the memory hierarchy were not as dominant a factor of performance as they are today. Consequently, the restrictions placed on a conforming implementation of the C++ standard forces a design similar to a balanced binary search tree. These structures have many desirable qualities, but do not make effiecient use of the memory hierarchy.

This report presents alternative ordered set structures which conform to a subset of the C++ standard demands. Drawbacks and strengths of these alternative structures are discussed, and running time for a number of use cases, set sizes and element types is measured.

These experiments show that relaxing the requirements of the C++ standard ordered set definition can give large gains in performance.

Oppsummering

Klassene i C++-standarden som sttter ordnede mengder ble designet i en tid da latency i minnehierarkiet ikke var en like dominerende faktor for ytelse som det er idag. En konsekvens av dette er at restriksjonene som C++ plegger en korrekt implementasjon tvinger ordnede mengder til implementert med et balansert binrt sketre.

Denne reapporten presenterer alternative strukturer for lagre ordnede mengder, og som sttter en undermengde av kravene i C++ standarden. Styrker of svakheter ved disse strukturene diskuteres, og kjretider for et utvalg use cases, mengdestrrelser of elementtyper er mlt.

Disse eksperimentene viser at ved fire p noen av kravene for ordnede mengder i C++ standarden, kan gi store gevinster i ytelse.

Acknowledgements

I would like to thank my advisor, Magnus L. Hetland, for giving me the guidance and support necessary to complete my thesis work.

Preface

This project was motivated by early experiments of the author on the relative performance of ordered set structures implemented in C++. The author was subsequently encouraged by his advisor, M. L. Hetland, to turn this work into a thesis.

The project is more focused on practical and simple solutions rather than theoretical and possibly complex solutions, as the goal is to contribute to the C++ community with simple, generic structures that may increase performance in a variety of problem domains.

The reader is assumed to be familiar with C++, but details of the standard library will be explained when necessary. Specifically, the reader should be familiar with the clear distinction between value and reference semantics, and the related distinction between copy and move semantics.

Table of Contents

Summary i

Oppsummering i

Acknowledgements ii

Preface iii

Table of Contents vii

List of Tables ix

List of Figures xi

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Outline . . . 2
- 1.3 Related Work . . . 2

2 Desirable Qualities of Data Structures 5
- 2.1 Running time . . . 5
  - 2.1.1 Theoretical Models . . . 5
- 2.2 Copying and Moving . . . 7
  - 2.2.1 Hard Move/Copy Characteristics of Elements . . . 8
  - 2.2.2 Soft Move/Copy Characteristics of Elements . . . 9
- 2.3 Iterator Validity . . . 9
  - 2.3.1 Validity vs. Cache Efficiency . . . 10
  - 2.3.2 Validity Guarantees . . . 10
- 2.4 Exception Guarantees . . . 11
- 2.5 Element Requirements . . . 15
- 2.6 Thread Safety . . . 15

3 Data Structures **17**
3.1 Set . . . 18
3.2 Flat Set . . . 20
3.2.1 Custom Flat Set . . . 23
3.3 Circular Set . . . 23
3.3.1 Circular Iterators . . . 24
3.4 Slide Set . . . 25
3.5 Merge Set . . . 28
3.5.1 Merge Iterator . . . 32
3.6 Slack Set . . . 34
3.6.1 Proxy Slack Set . . . 38
3.7 Adaptive Slack Set . . . 38
3.8 B^+ Set . . . 41
3.8.1 Comparable Proxy . . . 43
3.8.2 Unrolled Linked List . . . 44
3.9 Comparable Hash Set . . . 44

4 Use cases **49**
4.1 Element Types . . . 49
4.2 Operation Sets . . . 50
4.2.1 Search . . . 50
4.2.2 Insertion . . . 50
4.2.3 Deletion . . . 51
4.2.4 Iteration . . . 51
4.2.5 Mixed Insertion/Deletion . . . 51

5 Results **53**
5.1 Experimental Setup . . . 53
5.2 Tuning . . . 53
5.3 Search . . . 54
5.3.1 Random Order . . . 54
5.3.2 Ascending Order . . . 54
5.4 Insertion . . . 55
5.4.1 Random Order . . . 55
5.4.2 Ascending Order . . . 56
5.4.3 Descending Order . . . 58
5.4.4 Middle Order . . . 58
5.5 Deletion . . . 60
5.6 Iteration . . . 61
5.7 Mixed Inserton/Deletion . . . 62

6 Discussion **65**
6.1 Data Structures . . . 65
6.1.1 Set . . . 65
6.1.2 Flat Set . . . 66
6.1.3 Circular Set . . . 66

6.1.4 Slide Set 66
6.1.5 Merge Set 67
6.1.6 Slack Set 67
6.1.7 Adaptive Slack Set 67
6.1.8 B^+ Set 68
6.1.9 Comparable Hash Set 68

7 Conclusion **69**
7.1 Future Work 69

List of Tables

3.1 Running times for std::set 19
3.2 Running times for boost::flat_set 22
3.3 Running times for circular_set 25
3.4 Running times for slide_set 28
3.5 Running times for merge_set 30
3.6 Running times for slack_set 37
3.7 Running times for adaptive_slack_set 41
3.8 Running times for b_plus_set 42

List of Figures

3.1 Typical structure of `std::set` 18
3.2 Structure of `boost::flat_set` 20
3.3 Structure of `circular_set` 24
3.4 Structure of `slide_set` 26
3.5 Ordinary insertion into `slide_set` 26
3.6 Relocating insertion into `slide_set` 26
3.7 Structure of `merge_set` 30
3.8 Invalidation of iterator in `merge_set` 32
3.9 Conceptual view of a Packed Memory Array 34
3.10 Structure of `slack_set`, using `static_slide_set` for chunks 35
3.11 Insertion into `slack_set`, triggering redistribution 36
3.12 Structure of the `proxy_slack_set` searching cache 38
3.13 Structure of the `b_plus_set` 42
3.14 Structure of `comparable_hash_set` 45
3.15 Structure of `comparable_hash_set2` 46

5.1 Searching for elements in strucure with size n 55
5.2 Searching for elements in strucure with size n 56
5.3 Inserting n elements in random order 57
5.4 Inserting n elements in ascending order 58
5.5 Inserting n elements in descending order 59
5.6 Inserting n elements in middle order 60
5.7 Deleting all n elements in random order 61
5.8 Iterating over structure with n elements 62
5.9 Mixed insertion/deletion in structure of size n 63

Chapter 1

Introduction

1.1 Motivation

Storing and querying a dynamic set of elements is a basic building block of many algorithms. Consequently, readily available generic solutions to this task are essential for most general-purpose programming languages. For a performance-oriented language like C++, it is also important that the generic solution is competitive with custom-made solutions for the most common use cases.

An ordered set is a set of elements for which some total order is defined. It is traversed in sorted order, and consequently allows for efficient range queries. The standard C++ class template for ordered sets is `std::set`. Even though the C++ standard does not specify implemententaion details, the requirements for a conforming implementation are such that the implementation must be similiar to a balanced binary search tree, like Red-Black Trees or AVL trees [2, 5]. These structures are optimal for any comparison-based structure when analyzed in the Random Access Machine (RAM) model, but have practical limitations. Specifically, performance is poor for some tasks because of inefficient use of the memory hierarchy, which is not captured in the RAM model.

These observations motivate looking at other models besides the RAM model, and reevaluating the requirements of C++ ordered sets. The Cache-Oblivious (COB) model will be utilized, to model the movement of data across a single memory hierarchy boundary. We will also look at models attempting to model several levels of the memory hierarchy.

Modern hardware is very complex, and so is difficult to model with precision. In addition to the theoretical models, we will measure the running times of different structures, for several different use cases. Based on these measurements, we will try to explain what constant time factors contribute to make a structure more or less practical.

1.2 Outline

In Chapter 2 the most important aspects of a useful set structure are discussed, to be used as a reference when assessing individual structures. The first and foremost aspect is running time, because a very slow data structure is not likely to be useful to anyone. Section 2.1 will present the theoretical models used to analyze running time, and also discuss some practiacl considerations not captured by any of the theoretical models.

Section 2.2 will discuss the possible upper bounds placed upon an data structure with regards to copying and moving elements. Correlated with copies and moves is the iterator validity gurantees that a structure can offer, which is discussed in Section 2.3. Different levels of error handling capabilities will be discussed in Section 2.4. Although it is not a focus area of this paper, thread safety will be briefly discussed in Section 2.6.

Chapter 3 presents the data structures that are compared in this paper. Included among these are the C++ standard `std::set` [7], as well as the simpler `boost::flat_set` [16]. All the other structures are implemented specifically for this paper, though they are not completely novel structures. This has been done in order to be able to better understand the design trade-offs of each structure, and have full control over the iterator validity and exception guarantees offered.

In Chapter 4 we present the use cases on which empirical performance tests are performed. A single use case is a set of operations to be performed on the data structures, as well as the type of element which is to be stored in the sets. The goal of these use cases is to find the situations in which performance of the different structures are significantly different.

Results from these performance tests are presented in Chapter 5, and form the basis for the discussion of Chapter 6. In this discussion, the performance results will be combined with the other desirable aspects of the data structures. For example, a large gain in performance can make up for a relaxed exception guarantee, but a small performance gain will probably not. There is another reason for ignoring small performance gains: the tests performed in this paper are for single implementations compiled with a single compiler on a single computer, and in all likelihood some results will not generalize. When ignoring small performance gains, it is more likely that the performance gains that are considered will also generalize better.

The discussion will lead to a recommendation, presented in Chapter 7. This is intended to be a practical recommendation for a developer in need of an efficient ordered set structure, as well as a guide to future work on developing practical ordered set structures.

1.3 Related Work

The cache inefficiency of balanced binary search trees in general, have been noted by [20, 24], and many others. Solutions to this problem are mainly centered around different variations of the B-tree [8, 9, 11, 12], cache-sensitive layout of binary search trees [9, 11, 18] and Packed Memory Arrays [9, 10]. Using these approaches, it is possible to create ordered set structures that are optimal in both the RAM and COB models. In these papers, little attention is given to the element types, usually assumed to be some small record or integer. Running times are only assessed for fairly large n ($> 10^6$). As the data structures

are not presented as proposals for inclusion in a language like C++, important topics like iterator validity, exception guarantees and simplicity of implementation are not discussed.

On the more practical side, dissatisfaction with the performance of `std::set` is expressed in [4], and the practical advice given there for implementing a small simple set structure lead to the development of `boost::flat_set` [16]. As the `boost::flat_set` was intended for use as a drop-in replacement for `std::set`, considerations like iterator validity and exception guarantees are addressed. However, this structure was only intended for very small sets, or for large sets only if insertions happen in sorted order.

The Judy Array structure [25] was developed to handle large sets of integers, where the individual bits of the integer is used to build a trie-structure, essentially using a range of k bits of the integer in each node to efficiently branch into 2^k possible child nodes. As this approach abandons comparisons as the basis of search, there is a substantial loss of generality. However, depending on the model used for analysis, it is possible to use this approach to support $O(1)$ search and efficient range queries, which is not possible when relying on comparisons. Inspired by this structure, we have developed hash-based ordered structures, discussing how this less general approach can be extended to more than just integers, if not to all element types.

Chapter 2

Desirable Qualities of Data Structures

2.1 Running time

Running time is the primary characteristic of data structures considered in this paper. Some loss in running time may be acceptable if rewarded by better error handling, lower memory usage etc., but in general a data structure that is much slower than its alternatives will not be used. For each structure, running time for different operations will be analyzed theoretically, as well as measured. The theoretical models serves as motivation for implementing data structures, and makes it possible to define running time independent from implementation and hardware. It also provides possible constraints for an implementation-independent definition of the data structures, suitable for standardization.

2.1.1 Theoretical Models

Several models exist for analyzing running time for algorithms. Different models vary in the level of detail modeled. Simple model make analysis easy, and results general. More advanced models do generally make analysis more complex, and sometimes less general, but aims for more realistic results than simpler models. For example, in this report we are often interested in the effects of memory access patterns on running time, which are not modeled by the simplest model (RAM).

Random Access Machine Model (RAM)

The RAM model [13] is a simple model, and considers all memory as arranged in a single unbounded array. The memory may be accessed at any position in constant time. This is a simple model, and is used for requirements on data structures in standard C++. It does not model any cache, however, and is therefore unable to distiguish between algorithms with different memory access patterns.

External Memory Model (I/O)

The external memory model, or I/O model [28], seeks to take access patterns more into account by introducing a two-level storage system. The first level is bounded by size M, while the second is unbounded in size. Data can only be used when it resides in the first level, and must be brought in from the second level in blocks of size B. Originally, the first level in this respect was main memory, while the second level was disk storage. Later, as main memory sizes have grown and CPU caches have been introduced, the model is also used to model the tranfers between a small CPU cache and an conceptually unbounded main memory.

The parameters M, B are constants known by the program, and running time is measured by the number of cache lines brought into the cache during the run of the program. It is assumed that this cost dominates running time, such that tranfers from cache to registers and computation costs are negligible. The cache/memory layers in this model can model any two layers in the memory hierarchy. It could be the program working set in memory and pages stored on disk, where a page is brought into memory by a page fault, but the model can also be used to model transfers between on-chip caches. There are two main problems with using this model: Most algorithms developed in this model assume that the program knows the parameters M, B, therefore the same program may not be able to port its performance across different hardware. Second, the model only optimizes transfers between two layers. For all other layers, all transfers are free. An algorithm optimized for few cache misses might still get many page faults, and vice versa.

Cache-Oblivious Model (COB)

The Cache-Oblivious Model was introduced by Prokop in 1999 [22], and is a modification of the external memory model. The only difference from the external memory model above is that the parameters M, B are not known by the program, but this turns out to make a large difference. The algorithms that perform optimally in this model are often more complex than algorithms created in the external memory model. For example, the B-tree is optimized for the external memory model, because the branching factor is chosen such that a node fits in a cache line, but this is not possible in the cache-oblivious model.

The benefit of the the model, however, is that when an optimal algorithm is found, it is optimal for transfers between any two neighboring layers of the memory hierarchy [22]. This means that in theory, it is optimal for any memory hierarchy. Because of this generality, the COB model is used instead of the external memory model for analyzing two-level transfers in this report.

The optimality property may unfortunately not be used to predict running time as a function of input size n on a memory hierarchy, however. The first problem is that cost of memory transfers at different levels are not part of the model, and are thus incomparable. Transfers at the lowest levels of the memory hierarchy tend to be slow, but infrequent, while at the upper levels transfers tend to be cheap, but frequent. Without knowing the transfer cost of each level relative to the transfer cost of other levels, the total cost at each level is incomparable to the cost at any other level. In addition, since B, M are specific to each layer, the total transfer cost of each layer is a function of B_i, M_i and n for each layer i.

Uniform Memory Hierarchy Model (UMH)

The Uniform Memory Hierarchy model (UMH) is a model of memory transfers introduced by [3]. Instead of modelling memory transfers between two levels as in the external memory model and cache-oblivious model, the UMH models transfers between a potentially infinite sequence of memory modules. The model is based on the Hierarchical Memory (HM) model, in which every memory model has its own total size, cache line size and transfer cost. The HM can realistically model a concrete memory hierarchy, but is very complicated and analysis is difficult. The UMH is an effort to remedy this by introducing a general pattern that all the modules follow. Let n_i be the number of cache lines at level i, and s_i the size of each cache line, where $s_0 = 1$. We define the *aspect ratio* of a module to be the fraction n_i/s_i, which has the value α for all levels in UMH. Furthermore, we define the *packing factor* to be the fraction s_i/s_{i-1}, which in UMH is a constant ρ for all levels. At each level i there is a cost t_i to transfer one bit between level i and $i+1$. UMH allows for an arbitrary transfer cost function $f(i)$, which only depends on the levels. Since the bits must be transferred in a cache line of size ρ^i, the total cost is $\rho^i \cdot f(i)$. To avoid further complicating the model, we will be using $f(i) \equiv 1$ for all analysis. (TODO: begrunne dette bedre) This implies a constant bandwidth between all levels, but exponentially increasing latencies.

It follows from this description that the module at level i has cache line size ρ^i, and that the number of cache lines is $\alpha\rho^i$. Thus the total size of the module is $\alpha\rho^{2i}$. When discussing an algorithm on an input of size n, it is assumed at the outset that the input resides in the first module it can fit into. The first level u such that $n \leq \alpha\rho^{2u}$ is $\lceil \frac{1}{2}\log_\rho \frac{n}{\alpha} \rceil$. If we assume that the input fits exactly into level u, the cache line size ρ^u may also be expressed as $\sqrt{\frac{n}{\alpha}}$.

A major implication of this model is that no useful algorithm is faster than $O(\sqrt{n})$, as this is the time required to read a cache line from the lowest layer. We base this on the assumption that every bit of the input can potentially be read, since this is the only reason to include it in the input.

2.2 Copying and Moving

The number of copies and moves performed by a data structure can significantly affect the use cases it may support and performance for different element types. Furthermore, the copy and move semantics of the data structure interacts with both iterator validity and error handling.

The copy/move characteristics of element types are separated into two categories, hard and soft. Hard characteristics define which operations are supported by the element type, while soft characteristics are the cost copy/move operations. The hard characteristics of the element type determines which operations the data structure may support, as well as the asymptotic number of element copies and moves per operation. Actual performance is also affected by the soft characteristics of element types.

In theory, it is possible for the asymtotic number of copies and moves to depend on soft characteristics of the element type. A data structure could choose appropriate layouts and paramters based on the cost of copies/moves relative to the cost of cache misses, for

instance. This would, however, require either compile-time knowledge of these characteristics, given explicitly by the user, or run-time testing of copy/move costs. The compile-time option requires possibly tedious annotation of element types by the user, while the run-time approach adds a level of overhead. Because of these reasons, and in the interest of limiting the scope, none of these approaches are considered in this report.

2.2.1 Hard Move/Copy Characteristics of Elements

In this paper we consider four possible types of elements, based on whether they are copyable and/or movable. Technically, more possiblities exist, but we assume for simplicity that assignment operators and constructors are consistent, such that any copy-constructible element is copy-assignable, and likewise for moving.

Non-movable, non-copyable elements are rare, but may exist. The only way to support such elements is to support the `emplace` function, and construct the element directly where it will reside until erased. For this to be possible, the position in which to place the element must be known without any information about the element, which is impossible for most structures in this paper. These element typs are supported by the standard `std::set`, because all structure may be defined by pointers, such that the relative position of elements in memory is irrelevant.

Move-only types are much more common and include `std::unique_ptr`. The primary challenge of such element types is that no copies of the element kan be kept in a search structure, which is required by structures like the B^+-tree.

Another challenge that comes with move-only types is with reorganization during insertion into the data structure. The C++ `std::set` definition requires that the `insert` function returns an iterator to the element inserted. As one goal of this report is to present possible drop-in replacements to `std::set`, we wish to support the same interface for all structures. There is a challenge to supporting this, however. All data structures discussed in this report have a set of invariants that must be respected by all operations. Typically, a single insertion operation will complete quickly without violating invariants, but will often bring the data structure closer to such a violation. As this insertion operation must involve finding a suitable place for the new element, returing an iterator pointing to this element is trivial.

Once in a while the quick version of the operation leaves the structure in an invalid state, and a costly reorganization must be performed to restore invariants. During this reorganization, the element just inserted might be relocated, thus invalidating any iterator that was to be returned from the insertion function. When an insertion triggers a reorganization, a simple strategy is to forget the position of the element, but keep a copy. When the reorganiztion completes, a search is made using the copy, and the correct position is returned by the insertion function. As creating such a copy is not possible with move-only elements, the position of the element must be kept track of during the reorganization, complicating the reorganization code and possibly adversely affecting performance. If this performance hit is significant, a possible option would be to extend the interface with an insertion function which does not return an iterator to the inserted element. There is also another reason why this interface extension might be a good idea. The data structures that may invalidate an iterator during insertion returns iterators of lower value to the user, since they may be invalidated by any subsequent insertion. If this possible invalidation makes

the iterators of no use to the user, the data structure might as well not go the unnessecary expense of returning a correct iterator.

2.2.2 Soft Move/Copy Characteristics of Elements

By soft characteristics we mean the cost of copying and moving elements. For example, it is very common for elements to have a fairly expensive copy operation, and a cheap move operation. The archetype for this behaviour is a large `std::string`. The `std::string` object typically consists of a pointer to a chunk of memory containing the actual characters of the string. A move may be performed by simply copying the pointer value. A copy operation, however, involves allocationg a new chunk of memory and copying all the characters from the original string. For such types, copies should be kept to a minimum and optimally should be thought of as move-only. However, the necessary copies mentioned in Section 2.2.1 are not usually a big problem, as search structures are usually much smaller than the entire data structure, and total reorganizations of data structures are rare.

If the elements are copy-only and copies are expensive, or both copy and move is expensive, this is a bigger restriction. This is the case for large `std::array` elements. In this case all reorganization of the structure is expensive. Structures that do less reorganization, possibly at the cost of being less cache-efficient, might then be preferred.

2.3 Iterator Validity

In C++, iterators are a unifying interface for pointing to elements in a data structure. An iterator can be used for accessing an element in the data structure, and for efficiently aquiring an iterator poiting to the next or previous element. In practice, this is achieved by having the iterator point to one or more substructure(s) of the data structure. For a `boost↩ ::flat_set`, the iterator will point directly to the element, while for a `std::set`, the iterator is typically a pointer to the tree node containing the element.

Because the iterator is conceptually a pointer, it is usually invalidated by moving the element pointed to. There are ways to avoid this invalidation, but this will incur some additional cost.

One possible way to avoid invalidation is by storing back pointers from elements to all iterators pointing to the element. Then these iterators may be updated if the element is moved. The drawback of this method is that the cost of updates to the data structure will depend on the number of iterators in existence. Since these iterators may be stored at arbitrary locations, these updates would also not be cache efficient.

It is also possible to store proxy iterators, pointers to a proxy object that points to the actual element. Each time an element is moved, it updates its proxy object to point correctly. As several iterators may point to the same proxy, this avoids updating several iterators when moving items. However, because these proxy objects are never moved, they are stored in arbitrary locations in memory. When a bulk of contiguous elements is moved, the move itself is cache efficient, but the updates to the proxy objects will not be.

Finally, it is possible for the iterator to not store a pointer at all, but keep a copy of the element to which it conceptually points. Operations on the pointer is achieved by first searching for the actual element in the structure. This will likely have a severe performance

impact. To avoid always searching for the element, the iterator may additionally have a pointer to the last known location of the object. When the iterator is used, some check must be performed to see if the pointer still points to a valid element. If it does, the element pointed to can be compared to the copy kept by the iterator. If there is a mismatch, the element is probably close, such that a local search can be performed much quicker than a full global search.

Because of the added complexity of keeping iterators valid when moving elements, we assume for the rest of this report that iterators are always invalidated by moving elements.

2.3.1 Validity vs. Cache Efficiency

Assuming that all element moves invalidate iterators, there is an inherent trade-off between iterator validity and cache efficiency. If an element is never to be moved, the optimal position in memory may only depend on the elements inserted thus far, not taking into account elements to be added in the future. Furthermore, the optimal placement of each element when all elements are added, i.e. in contiguous order, would not be the optimal placement for all intermediate steps, as elements would be spread over too large an area.

This is the core reason for the inefficiency of `std::set`. The demand that iterators are valid until the element is erased prohibits a cache effiecient structure. In all other structures presented in this paper, the requirement of iterator validity is relaxed, since otherwise few improvements in performance are possible.

2.3.2 Validity Guarantees

We distinguish between levels of guarantees for a single operation, and levels of guarantees for entire structures. The guarantees for entire structures are defined in terms of the guarantees for their operations.

This report distinguishes between three levels of guarantees for functions, from weakest (1) to strongest (3).

1. Any iterator may be invalidated.
2. Conditions for iterator invalidation by the operation are well defined, and may be checked by the user prior to the function call
3. Iterators are only invalidated for elements that are deleted by the operation

The second guarantee is a heterogeneous category, but usually supports some useful guarantee that naturally follows from the properties of specific data structures. For example, the `boost::flat_set` is modelled as a linear array of elements in order. The smallest element is placed at the first possible position in the array (left), and the array grows in the direction of the largest element (right). Consequently, inserting an element x means moving all elements larger than x one position to the right. All the iterators for elements larger than x are invalidated, while all iterators for elements smaller than x are not. Before inserting x, the user can check whether an iterator might be invalidated, by comparing x to the element pointed to by the iterator. Although such conditional validity guarantees may

have few use cases, it may be fruitful to identify such guarantees when they do not overly restrict the implementation.

For structures, three roughly corresponding levels are defined:

1. No iterators are invalidated by `const` functions. No guarantees are made for non-const functions.

2. No iterators are invalidated by `const` functions. For all non-`const` functions, conditions for iterator invalidation, and may be checked by the user prior to the function call

3. Iterators are only invalidated by functions deleting the element to which they point.

The first level of guarantee is very useful for reasoning about the code that is using the data structure. Since iterator invalidation has observable consequences for the user, invalidating iterators during a `const` operation is a breach of the `const` contract, namely that it should not change the structure in a manner observable to the user. This requirement is even more important in a multi-threaded context.

There are some interesting data structures that do not fulfill the weakest guarantee. In [9], a cache-efficient linked list is presented, that moves elements during traversal, such that they are placed more optimally in memory during subsequent traversals. As traversing a structure is conceptually a `const` operation, this is in violation of the weakest guarantee. Because of the problems associated with violating the weakest guarantee, such structures are not discussed further in this report.

The third and strongest guarantee effectively demands that no elements are moved after being inserted. We call the iterators of such structures *stable*. This is almost as strict as the no-copy/no-move hard characteristic discussed in section 2.2.1. In the strict no-copy/no-move case, the element must be constructed in the memory location where it is to reside until destruction. When supporting the strongest validity guarantee, however, the element may still be constructed at an arbitrary location, and compared to other elements to find a better final location. In addition, if the element type is copyable, a search structure may be built on top of the non-moving elements, and this search structure may be rebuilt to support more cache-efficient search.

2.4 Exception Guarantees

A theoretical description of a data structure will usually not include a full specification of how to handle potential errors. No such specification is needed if errors are assumed not to occur, or if errors cause the entire program to abort. If the user wishes to recover from errors, however, it is useful to be able to reason about the state of the data structure after certain errors have occured. For example, knowing that all destructors are correctly run for elements removed from the data structure may be needed to avoid leaking resources. In other cases, stronger error handling guarantees may ensure that less work is lost because of an error. For example, this is the case for errors occurring during reallocation of `std::vector`. When adding a new element to the vector, a reallocation might be triggered. A larger memory block is allocated, and elements are copied from the old memory block. If an error occurs during copying, the original block is still intact, and the new block may

simply be discarded. This leaves the structure in the state in which it was prior to the last insertion attempt. As this behaviour is guaranteed by `std::vector`, the user of `std::vector`↩ may be able to recover from the error faster then would be possible if the user had to assume that the entire structure was corrupted.

Some sources of errors are practically unavoidable, for example allocating memory. In C++, the standard error reporting mechanism is through exceptions, so error handling will be discussed as exception handling. Potential sources of exceptions that will be discussed are:

- Memory allocation by data structure
- Element constructor
- Element copy - by assignment or constructor
- Element move - by assignment or constructor

As element comparisons are typically read-only operations, we do not consider them as potential sources of unavoidable errors. Neither do we consider exceptions thrown from inside destructors, as there is generally no good way to deal with such exceptions [26].

When considering a data structure, it is important to consider what guarantees the structure can make on its state if any of these exceptions do occur. It is useful to separate between exceptions caused by structure library code (mainly memory allocation), and the exceptions caused by user code. Potential guarantees will be discussed for the structures developed in this project, even though some of those guarantees are yet to be implemented and/or tested. That is, we will discuss what guarantees can be made without changing the data layaout and without adversely affecting performance. We consider the following possible levels of exception guarantees:

1. **Weak guarantee** - Exceptions cause bad state. No resources are leaked.
2. **Basic guarantee** - Exceptions cause undefined, but valid state. No resources are leaked.
3. **Partial guarantee** - Exceptions cause partially defined, valid state. No resources are leaked.
4. **Strong guarantee** - Exceptions leave the state unchanged. No resources are leaked.
5. **No-throw guarantee** - Exceptions cannot occur.

Other levels of exception safety could be defined. However, these levels summarize the most important characteristics of exception handling discussed in this report, and does so in a simple manner. Levels 2, 4 are 5 are defined as the *basic guarantee*, *strong guarantee* and *no-throw guarantee*, respectively, by [1]. Additonally, we have introduced the *weak guarantee* and the *partial guarantee*. In the rest of this section, the motivation for each of these levels is presented.

A structure that has absolutely no defined state after an exception is quite unsafe, and will not be considered in this report. We require for all structures that destructors are

correctly called for all elements when an exception is thrown, for the same reason. As noted earlier, the assumption is made that all element destructors are `noexcept`, as this is critical to correct exception handling, as argued in [26].

The weak guarantee is most useful for guaranteeing that resources are not lost. Since the state is otherwise undefined, it may be difficult to reason about the state, so the structure will most likely be cleared and thrown away. An alternative to this guarantee would be complete cleanup of the structure, as if `clear()` had been called. Since the empty state is a valid state, this would satisfy the basic guarantee. However, leaving the decision of when to clean up to the user allows for delaying this work to a more appropriate time. In some situations, the best option for the structure may be to enter a bad state, which only supports `clear()`, assignment and destruction. Enough structure is kept to support cleanup in these three scenarios, but parts of the structure relevant to search, insertion and other operations may be lost.

An example of the usefulness of bad state can be illustrated by a sorted vector with unique values. When inserting an element into this vector, all larger elements must first be moved one step to the right. Assuming the element type is non-movable, the elements must be copied one step to the right. Furthermore, if the copy operation may throw exceptions, this can happen when trying to copy any of the elements. When a copy operation fails in the middle of this insertion operation, how can the structure ensure that the state is made valid?. The ordered vector now has a gap in which resides the undefined result of an interrupted copy assignment operation. It is the users responsibility to ensure that the interruption of the copy operation does not lead to memory leaks, but other than that the element may be in any state. It is certainly unlikely to be a unique value respecting the sorted order of the set, so the invariant of the sorted vector with unique values is broken. Three possible options are:

- Try to redo the failed copy and continue, returning normally. This would attempt to satisfy the no-throw guarantee
- Try to roll back the copying done so far before rethrowing the exception, thus satifying the strong guarantee
- Delete all elements from the point of failure to the end of the vector, satisfying a partial guarantee.

As argued in [1], the first two approaches are ill-advised, as they try to redo operations that have already failed, and may very likely fail again. For example, the failures might be due to an allocator that is out of memory, so the retries might very well go on in an endless cycle. The third option is possible, because we have imposed the requirement on elements that the destructor is `nothrow`. However, this involves linear time cleanup, that the user may prefer to do at a later time. In this case the bad state guarantee might be the most preferable.

The basic guarantee specifies that all invariants of a data structure holds even when an exception occurs. Note that this guarantee has more implications for an ordered set structure than it has for a sequential container like `std::vector`. For example, a `std::vector` may fulfill the basic guarantee in insertion operations like the one mentioned above, because it does not need to respect any ordering among its elements.

The partial guarantee may be useful for specific operations. Consider bulk insertion into the back of a `std::vector`. Given a `std::vector` along with the begin and end iterator of an input range, the function should insert all elements of the input range into the back of the `std::vector`. Let us further assume that the iterators are input iterators, meaning they may only be used once, and it is not possible to query the size of the input range before inserting. Without knowing the size of the input range, it is impossible to know whether a reallocation will be necessary, so elements are inserted directly into the original memory chunk, possibly until a reallocation is triggered. If the reallocation fails, there is a trade-off to be made: All elements inserted thus far could be destructed, and the structure could thus fall back to the state prior to the function call, thus fulfilling the third level guarantee. Another possibility is to return the current valid state, even though only elements up to some point were inserted. The latter approach is an example of the second level guarantee, in this case that a prefix of the input range is inserted. The user may then look at the new size of the `std::vector` to find the number of elements inserted. The structure has already avoided the work of destructing the inserted elements. If the user is able the save work by resuming the insertion at the point of failure, the total benefit can be significant. Currently the C++ standard has no such partially defined guarantees [7], but they will be discussed in this report nonetheless, motivated by the case above.

The strong guarantee is the highest guarantee given by insertions in the C++ standard library. This guarantee makes reasoning about state easy, since there is only one possible state after failure.

The no-throw guarantee is difficult to support for many operations in practice. Any operation that needs to allocate memory risks allocation errors. In addition, any operation that calls user code that might throw must deal with exceptions. This is often the case for element copying, when elements perform memory allocation as part of the copy operation (e.g. `std::string`).

Standard C++ containers are *non-intrusive*, which means that they perform their own memory allocation and do not expose implementation structures such as node types to the user. Because they allocate memory, operations like insertion can not be `noexcept`, but this is not true for *intrusive* containers. Intrusive containers do expose internal structure to the user, allowing the user to perform all memory allocation. Consider for example a search tree. The standard non-intrusive way to create a new node is for the user to pass an element to the search tree. The search tree then allocates a node object, and copies or moves the element into this node. The user need not be aware of this node object, as it is not part of the interface of the search tree. In an intrusive search tree, however, the node type is made explicitly visible to the user. The user may create such a node on either the stack or the heap. The user then passes a reference to this node to the search tree, and the search tree updates both the referenced node object and other node objects to include the new node into the search tree. As the node objects may be allocated on the stack, this may potentially involve no heap allocations. This makes it possible to eliminate the risk of heap allocation errors, but might of course eventually lead to stack overflow. An introduction to intrusive data structures can be found in [15]. Even though intrusive containers have interesting qualities, they require a radically different interface from the standard containers in C++, and do a worse job of abstracting away implementation details. For this reason, intrusive data structures are not covered in this report.

2.5 Element Requirements

When writing generic libraries, an important goal is to be as general as possible without sacrificing performance. This means putting as few requirements on the types of elements supported as possible. Some of these requirements are discussed above, specifically copy-/move requirements in Section 2.2 and exception specifications in Section 2.4.

In addition, the element types must support some comparison operation, otherwise the notion of an ordered set is meaningless. In C++, the standard way to conform to support comparison is by overloading the $<$ operator, or by passing a custom comparator object to the ordered set structure. All structures in this report assume one of these operations to be present. Additionally, this comparison function must be transitive, irreflexive, and asymmetric.

Some structures presented in this report demand that elements support a larger interface, and some structures also base performance characteristics on assumptions on the distribution of elements. This does make the structures less generic, but may be justified with a large gain in performance.

2.6 Thread Safety

Data structures designed specifically for multi-threaded usage is beyond the scope of this paper. However, the restrictions placed upon iterator validity and error handling are in part motivated by multi-threaded use. Similar to the standard C++ library specification [7], const operations should be considered read operations in a multi-threaded context. As such, any concurrent execution of const member functions is valid, and should not be observably different from single-threaded execution.

The iterator validity guarantee of `const` may be seen as a consequence of the thread safety guarantee, as a const operation on one thread should never invalidate an iterator held by another thread.

No restrictions are placed upon non-const member functions, as it is assumed that the user will provide the necessary synchronization protocols, with full knowledge of when such protocols are needed.

Chapter 3

Data Structures

This chapter presents the data structures that are considered in this report as possible implementations of an ordered set. Most structures are implemented specifically for this report, and their implementation will be presented in detail. For other structures, like `std::set` and `boost::flat_set`, a briefer overview is presented. The decision to implement these structures from scratch gave several benefits:

- **Simplicity of explanation** - Instead of viewing structures as a black box, guessing which strategies might have been employed, or trying to gleen these details from studying source code, all implementation details are available to the author. This has been a great aid in explaining performance characteristics of the data structures.

- **Ease of modification** - Implementing the data structures gave the necessary understanding to try out several modifications of the data structures.

- **Common interface** - Many available implementations of data structures do not follow the C++ standard interface, and some are not even type generic.

- **Common underlying structures** - Ensuring that all structures use the same basic structures when sensible can make comparisons fairer. This was highlighted when implementing array-based alternatives to `boost::flat_set`. The new implementations used `std::vector` as an underlying array instead of `boost::vector`, and this resulted in a large speedup for similar operations ($\sim$ 6x). Switching to a custom implementation of `flat_set` using `std::vector`, the difference was eliminated.

- **Fun** - When given the choice between writing new implementations, and studying/modifying other implementations, the former was certainly a more motivating prospect for the author.

The data layout and operation of each structure will be presented. We will also discuss to what degree the structure may fulfill the desired qualities presented in Chapter 2. Where design choices reflect trade-offs between fulfilling conflicting demands, the rationale for the final choice will be given.

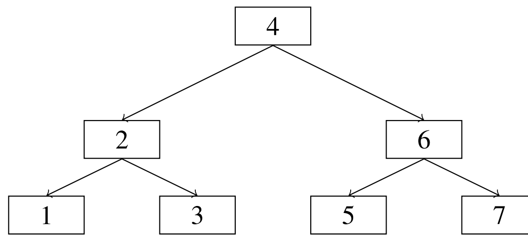

Figure 3.1: Typical structure of `std::set`

3.1 Set

The C++ standard version of an ordered set is `std::set`, described in [7]. Although the C++ standard does not explicitly enforce a specific implementation of `std::set`, the specification of running times for its operations, along with iterator validity guarantees, directs the implementation to some variant of a binary search tree, e.g. red-black trees [5] or AVL trees [2]. This is oulined below:

One important characteristic of `std::set` is that it requires that elements may be compared with the $<$ operator, or that an alternative comparison object is supplied by the user. As this comparison result is the only information available for searching, the worst case search time must be $O(\log n)$. This can be be seen from the fact that each comparison operation restricts the range of possible positions to one of two possible subdivisions, delineated by the elements to which the query element is compared. The best worst-case running time is achieved when the division is balanced, that is, when both division have size roughly equal to $n/2$. The number of comparisons needed is then $\log_2 n$. More generally, a constant number k comparisons can be made in each step. This would divide the range into $k + 1$ subdivisions in each step, thus requiring about $k \log_{k+1} n$ comparisons in total.

As `std::set` requires searches in $O(\log n)$ time, any implementation must find good comparison elements in constant time. A good comparison element is an element that is close to the middle of the remaining search range, such that a comparison between this element and the query element leads to a large reduction in the remaining search range. This is achieved in array-based structures by binary search [21], calculating the position of the next comparison element in constant time at each step. It is also achived by balanced search trees, both binary and B-trees. It is not achieved by skip lists [23], as the comparison elements are not guaranteed to reduce the remaining search range by a constant factor.

Another important requirement of the `std::set` is that it must fulfill the strictest possible iterator validity guarantee. This means that iterators to an element must be valid from the insertion of an element until the element is deleted from the `std::set`, no matter what other operations are run on the `std::set` in between. Due to reasons explained in Section 2.3, this means that elements may not be moved after their creation. This restriction means that array-based implementations of `std::set` are not possible, because it is not possible to keep a dynamic set of elements ordered in memory using $O(n)$ space, without occasionally moving some elements. Neither are B-trees possible, because splitting and merging its nodes require moving elements around in memory.

Although we have not completely ruled out the possibility that radically different structures from balanced binary search trees may exist that fulfill all properties of `std::set`, we assume for the remainder of this report that `std::set` is implemented as a balanced binary search tree.

Running time

For the RAM model, the running times listed in this section are not the result of analysis, but simply a restatement of the requirements defining `std::set`. For the COB and UMH models, however, the results are based on the assumption that `std::set` is implemented as a balanced binary search tree.

Searching a balanced binary tree takes $O(\log n)$ in both the RAM and COB models. In the UMH model, however, the running time is much worse. We assume that all the data initially reside in level $\frac{1}{2}\log_\rho \frac{n}{\alpha}$ and that cache line size is $\sqrt{\frac{n}{\alpha}}$. There is no guarantee that any of the nodes that must be traversed during search reside on the same cache line, therefore the $\log n$ cache lines might need to be read from the slowest cache. As each cache line requires $\sqrt{\frac{n}{\alpha}}$ time to read, the total search time is $O(\sqrt{n}\log n)$.

When elements are added to or deleted from a `std::set`, some rotations are sometimes necessary to keep the tree balanced. Although the number of rotations may be $O(\log n)$, the amortized cost is a constant number of rotations [27]. These rotations therefore have amortized cost $O(1)$ in the RAM and COB model, but $\sqrt{\frac{n}{\alpha}}$ in the UMH model. The UMH cost is due to the fact that each rotation requires following a pointer to an arbitrary memory location, therefore it might have to read from the slowest cache level.

For insertions and deletions there are two versions, one requiring a search and one which does not. For insertion this means supplying the position of a neighboring element together with element to be inserted. No search is needed, but rotations might still be needed. For deletions, the user may supply either the value of the element to be deleted, of its position. In the value case, a search is needed, but for the position case only rotations are needed.

Operation	RAM	COB	UMH	Dominating factor
Search	$\log n$	$\log n$	$\sqrt{n}\log n$	Tree search
Insertion, w/o position	$\log n$	$\log n$	$\sqrt{n}\log n$	Tree search
Insertion, w/ position	1	1	$\sqrt{n}$	Rotation
Deletion, value	$\log n$	$\log n$	$\sqrt{n}\log n$	Tree search
Deletion, position	1	1	$\sqrt{n}$	Rotation
Iteration	n	n	$n^{3/2}$	

Table 3.1: Running times for `std::set`

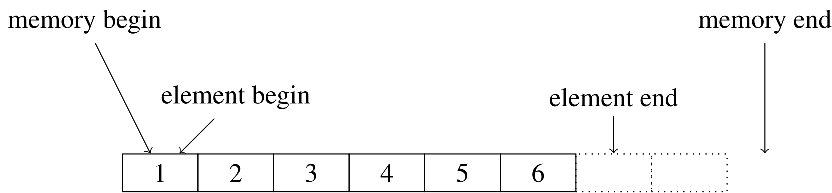

Figure 3.2: Structure of `boost::flat_set`

Copy and Moving

The `std::set` never moves elements after they are included, and may even support no-copy/no-move elements. Such elements may be constructed in an arbitrary location, and pointers are then updated to include the element into the structure.

Iterator Validity

The `std::set` gives the best possible guarantee for iterator validity, no iterators are invalidated until the element to which they point are deleted.

Exception guarantees

When errors occur during any operation of `std::set`, this leaves the `std::set` in the same state as it was before the start of the operation. This is the strong guarantee of Section 2.4, and is the strongest possible guarantee when errors might occur, either because of memory allocation done by the `std::set` itself, or by user code exceptions.

Furthermore, assuming that comparison is `noexcept`, the no-throw guarantee may be given for deletions.

Element Requirements

The `std::set` only requires that element types support comparison.

3.2 Flat Set

Flat set is a simple array-based set structure developed as part of the Boost Container Library [16], based on the recommendations of [4]. Elements are stored contiguously and in order in a dynamic array, as seen in Figure 3.2. This means that searches can be performed by simple binary search, and iterating over all elements is as fast as for `std::vector`. Insertions are performed by moving all elements larger than the inserted element one position to the right before inserting the element. Deletions are performed by moving all elements larger than the deleted element one position to the left.

Running time

Most operations on the `boost::flat_set` consist of a combination of searching, moving elements one step to the right or to the left, and the amortized cost of infrequently reallocating the underlying dynamic array. As searching is done by binary search, this can be performed in $O(\log n)$ time. In the cache-oblivious model, the running time is slightly better. After k iterations, the search range is reduced to size $\frac{n}{2^k}$. When the search range reaches size B, at most two cache lines are needed in the cache, and no more cache lines need to be brought in from memory. This happens when $\frac{n}{2^k} = B$, so we have $k = \log_2 \frac{n}{B}$, thus running time of search is $O(\log \frac{n}{B})$.

When analyzing binary search in the UMH model, it is assumed that the entire data structure is located in the first cache level into which it fits, which is at level $\frac{1}{2}\log_\rho \frac{n}{\alpha}$. The cache line size at this level is $\sqrt{\frac{n}{\alpha}}$, which is also the time required to read a cache line. Similar to the case for the cache-oblivious model, data must be read from this layer until the range is small enough to fit into the cache line. This requires $\log_2 \left(n/\sqrt{\frac{n}{\alpha}}\right)$ iterations of binary search, which is equal to $\frac{1}{2}\log_2 n\alpha$. Thus, the time required in the lowest cache level is $\frac{1}{2}\sqrt{\frac{n}{\alpha}}\log_2 n\alpha$. For each subsequent level of the cache hierarchy, the remaining search range must be reduced to the new cache line size, which means reducing the range by a factor ρ. This requires $\log_2\rho$ steps. The time taken for each cache line transfer is reduced by a factor ρ for each layer, which means the time taken for all layers except the lowest is $\log_2\rho \cdot \left(\sqrt{\frac{n}{\alpha}}/\rho + \sqrt{\frac{n}{\alpha}}/\rho^2 + \cdots + 1\right)$. This is dominated by the cache line transfers at the lowest level, such that total search time is $O(\sqrt{n} \cdot \log n)$.

Analyzing the running time of moving elements one position to the right or left is a simpler task. In the standard RAM model, this can be done in $O(n)$ time. Furthermore, since this task uses the cache optimally, it can be done in $O(n/B)$ cache line reads in the COB model. Again, the UMH model is a bit more complex, but still simpler than for search. Since transfer time is equal to cache line size at each level, each transfer cost may be amortized over the subsequent reads and writes of the same cache line, thus each level has running time $O(n)$. Since reads and writes are not concurrent at all levels of the memory hierarchy, the sum of the cost at each level must be used as the upper bound on total running time. As there are $\frac{1}{2} \log_\rho \frac{n}{\alpha}$ levels, total running time is $O(n \log n)$. This might seem very slow for an operation usually thought of as linear, but keep in mind that n random accesses has a running time of $O(n^{3/2})$ in this model, so the sequential access pattern of the element moves can still be characterized as relatively efficient in this model.

Analysis for the amortized cost of reallocation is done for growing an array because of insertions only, as the `boost::flat_set` does not automatically reduce in size when elements are deleted. Such a reallocation may only be triggered when $\Theta(n)$ elements have been added to the set. The reallocation requires moving n elements, which takes $O(n)$ and $O(n/B)$ time in the RAM and COB model, respectively. Thus, in both these models, the amortized cost is $O(1)$. For the UMH model, however, the cost of reallocation is $O(n \log n)$. The amortized cost is therefore $O(\log n)$.

The running time for iteration is defined as the running time of iterating over the entire set. For the RAM model, this is simply $O(n)$, while for COB, it is $O(n/B)$. For the UMH model, this takes $O(n \log n)$ time, based on the same analysis as for moving elements.

The running times of various operations are shown in table 3.2. For insertion and deletion at the back of the set, there are two different versions. When the element to be

inserted is larger than all elements already in the set, this can be discovered by a simple comparison to the largest element already in the set. If this check is not performed, a full binary search may be needed. The `boost::flat_set` does not perform this check, but it would be a very simple addition, therfore the time taken with this optimization is included in the table.

For front and back insertions and deletions, it is assumed that several such operations are performed sequentially, such that the beginning/end of the underlying array is already in the fastest level cache.

Operation	RAM	COB	UMH	Dominating factor
Search	$\log n$	$\log \frac{n}{B}$	$\sqrt{n} \cdot \log n$	Binary search
Insertion, random or front	n	n/B	$n \log n$	Element moves
Insertion, back, w/ search	$\log n$	$\log \frac{n}{B}$	$\sqrt{n} \cdot \log n$	Binary search
Insertion, back, w/o search	1	1	$\log n$	Reallocation
Deletion, random or front	n	n/B	$n \log n$	Element moves
Deletion, back, w/ search	$\log n$	$\log \frac{n}{B}$	$\sqrt{n} \cdot \log n$	Binary search
Deletion, back, w/o search	1	1	$\log n$	Reallocation
Iteration	n	n/B	$n \log n$	

Table 3.2: Running times for `boost::flat_set`

Copying and Moving

The `boost::flat_set` requires that elements are copyable or movable. Copying or moving elements dominate the running time for most operations, so the structure is excpected to be most useful for small objects.

Iterator Validity

The `boost::flat_set` follows the second level iterator validity guarantee of section 2.3, namely that conditions for iterator invalidation may be checked by the user. For both insertions and deletions, iterators to elements larger than the inserted/deleted element are invalidated, because these elements are moved one step to the right/left. Furthemore, an insertion might invalidate all iterators if a reallocation is triggered. As this will only happen when the `boost::flat_set` size is equal to its capacity, this condition may be checked by the user.

Exception Guarantees

The exception guarantees that may be given by the operations of `boost::flat_set` depends on the possible sources of errors. The simplest sources to handle are memory allocation

during reallocation. If a reallocation is triggered, the memory is allocated before copying or moving any elements. If `boost::flat_set` fails to allocate the necessary memory, there is no modification of the original memory chunk, and the strong exception guarantee holds.

For copying or moving, however, the situation is more complicated. Copying or moving is needed in two situations: Shifting elements to the right/left during ordinary insertion/deletion, and during reallocation.

For reallocation, the guarantees provided are the same as for `std::vector`. If elements are copied, the original memory chunk is not altered during reallocation, and thus it is possible to simply discard the new memory chunk and return to the old state if exceptions occur. Thus copying reallocation can respect the strong guarantee. If elements are moved, the original memory chunk will usually be altered during this operation, thus it may not be used as a fallback. Thus, if the move operations may throw, it is impossible to give the strong guarantee.

Shifting elements during ordinary insertion/deletion is more unsafe, because it is done in-place, which means there exists no complete backup in case of copy/move failure. As discussed in Section 2.4, there are two viable options when such an error occurs: Delete all elements from the point of failure to the last element, or simply return a bad state. The first option is an instance of the partial guarantee, since it may be guaranteed that the `boost::flat_set` still includes all elements smaller than the inserted/deleted element. The second option is an instance of the weak guarantee. As far as the author can tell, `boost::flat_set` follows the second approach. Thus, if the element type has either a `noexcept` copy or `noexcept` move operation, `boost::flat_set` may give the strong exception guarantee. If both copying and moving may throw exceptions, the `boost::flat_set` may only give the weak guarantee.

Element Requirements

The `boost::flat_set` only requires that element types support comparison.

3.2.1 Custom Flat Set

In preliminary experiments with `boost::flat_set`, it was discovered that insertion into the middle of the set was much slower than similar insertions into a `std::vector`. As the `boost::flat_set` is based on the underlying `boost::vector`, the two vector implementations were tested against each other, and showed large differences in performance on the test platform.

For this reason, a custom implementation using `std::vector` was developed, called `custom_flat_set`. In addttion to changing the underlying array, the search procedure was augmented by checking the last element of the range, to allow for $O(1)$ time insertion in ascending order.

3.3 Circular Set

As the `boost::flat_set` is built upon an underlying dynamic array, it can only grow efficiently in one direction, similar to `std::vector`. Inserting an element in the middle of the array causes all elements to the right of the inserted element to be moved. This means a

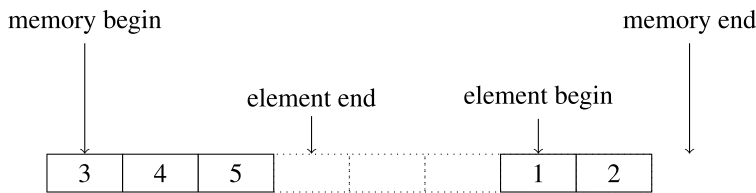

Figure 3.3: Structure of `circular_set`

random insertion will cause $\frac{n}{2}$ moves on average. Insertions at the lower end of the set will cause n moves.

The `circular_set` uses a circular buffer representation of a sequence to store elements in sorted order, as shown if Figure 3.3. This means that the sequence of elements can wrap around the end of the underlying memory buffer, and the sequence can therefore shrink and expand equally efficient in both directions. When inserting an element into the middle of the sequence, the elements can be pushed left or right depending on the position of the inserted element, never causing more then $\frac{n}{2}$ elements to be moved. On average, $\frac{n}{4}$ elements will be moved during one random insertion.

3.3.1 Circular Iterators

To support iteration of the `circular_set`, a new type of iterator was implemented. This iterator consists of three iterators to its underlying range:

- current, pointing to the actual element in the underlying range
- first, corresponding to the memory begin position in Figure 3.3
- last, corresponding to the memory end positions in Figure 3.3

When the current iterator reaches the last iterator, it wraps around to the first iterator. Compared to using the underlying iterators directly, there is some overhead associated with this circular iterator. First, it uses three times the amount of memory, as it must store three underlying iterators. Second, for every incrementation of the current iterator, it must be compared to the last iterator to check if it should wrap around to the first iterator.

Running Time

In the asymptotic sense, the running time for most operations on the `circular_set` is equal to that of `boost::flat_set`. The only difference is for insertions and deletions in the front of the set. As an extra optimization, the search operation of `circular_set` starts by checking the query element against the extreme values of the `circular_set`. This eliminates the need for a complete binary search if the element is to be placed in the front or the back of the `circular_set`.

Operation	RAM	COB	UMH	Dominating factor
Search	$\log n$	$\log \frac{n}{B}$	$\sqrt{n} \cdot \log n$	Binary search
Insertion, random	n	n/B	$n \log n$	Element moves
Insertion, front or back	1	1	$\log n$	Reallocation
Deletion, random	n	n/B	$n \log n$	Element moves
Deletion, front or back	1	1	$\log n$	Reallocation
Iteration	n	n/B	$n \log n$	

Table 3.3: Running times for `circular_set`

Copying and Moving

Although the `circular_set` copies or moves about half as many elements the `boost::flat_set`, this still dominates the cost of insertions or deletions.

Iterator Validity

Because elements on both sides of an inserted/deleted may affected by a insertion/deletion (though not at the same time), `circular_set` follows the first and lowest level of iterator validity guarantee. This means that any iterator may be invalidated by insertions and deletions.

Exception Guarantees

The handling of exceptions during insertion/deletion in `circular_set` is similar to that of `boost::flat_set`, which is discussed in Section 3.2. If the element type has either a `nothrow` copy operation or a `nothrow` move operation, `circular_set` can give the strong guarantee. If none of these operations are `nothrow`, `circular_set` can only offer the weak guarantee.

Element Requirements

The `circular_set` only requires that element types support comparison.

3.4 Slide Set

As seen in section 3.3, the `circular_set` improves upon some aspects of `boost::flat_set`, namely more efficient random inserts and much more efficient front insertion. These improvements come at a cost, however, as the circular buffer structure is more complex than a simple linear array. Most notable is tha fact that elements are no longer guaranteed to be located contiguously in memory.

The aim of the `slide_set` is to retain the benefits of the `circular_set` without the complicated structure. This is achieved, at keast in theory, by a very simple mechanism. The

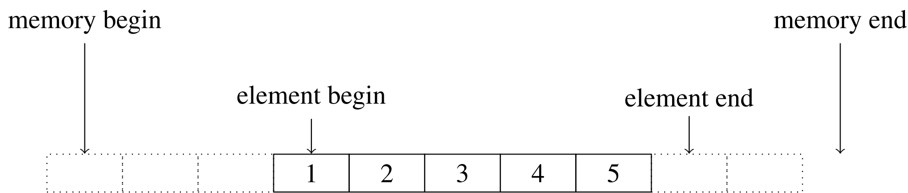

Figure 3.4: Structure of `slide_set`

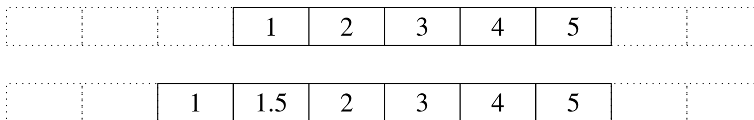

Figure 3.5: Ordinary insertion into `slide_set`

basic idea is to store elements contiguously in order in the middle of the underlying memory buffer, making insertions at both ends efficient, as shown in Figure 3.5. This is possible until the elements reach the end of the memory buffer in either direction. The next insertion in this end must force a move of many elements ($> n/2$). Instead of moving all the elements one step towards the other end, all the elements are moved to the middle of the memory buffer, thus restoring some order. This case is shown in Figure 3.6.

Random insertions in this structure is about as fast as is possible with any structure that stores all elements in order with no gaps, namely $\frac{n}{4}$ moves on average. As this is also the class of structures that supports iterators with no overhead over raw pointers, this is arguably an important class of structures.

For insertions in order, the analysis is slightly more complex. We will look at front insertions, with no loss of generality. Consider a `slide_set` with its n elements in the middle of a memory buffer, with a gap of $\frac{n}{2}$ elements at both sides. The $\frac{n}{2}$ first elements will be inserted without causing a move. The next insertion will trigger a move, creating new gaps of size $\frac{n}{4}$ before the element is inserted. As the gaps will be halved in each step, the number of steps before the buffer is full is $\Theta(\log n)$, during which n elements are inserted. Given that each of these steps cause $\Theta(n)$ elements to be moved, the total cost is $\Theta(n \cdot \log n)$. As a consequence, the amortized cost of each insertion is $\Theta(\log n)$. The next insertion will then cause an expansion of the underlying memory buffer, such that $2n$ elements are now stored in the middle of the buffer with size n gaps at each side. The same analysis will apply for the next insertions.

There is a way, however, to remove this $\Theta(\log n)$ factor. Instead of filling the array

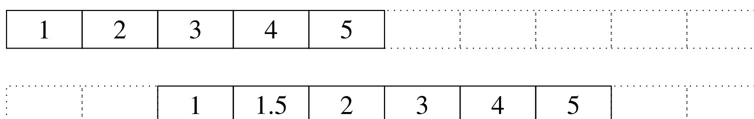

Figure 3.6: Relocating insertion into `slide_set`

completely before expanding the memory buffer, it is possible to set the maximum number of elements to some constant fraction $0 < \alpha < 1$ of the buffer size. Suppose that the memory buffer has just been doubled in size, and now has size m. This means the there are $\alpha \cdot \frac{m}{2}$ elements in the middle of the buffer, and that gaps are now of size $(m - \alpha \cdot \frac{m}{2})/2 = m(\frac{1}{2} - \frac{\alpha}{4})$. The buffer will be expanded when the gaps are $(m - \alpha \cdot m)/2 = m\frac{1-\alpha}{2}$. As the gaps are halved at each step, the number of steps k is approximately given by

$$\frac{m(\frac{1}{2} - \frac{\alpha}{4})}{2^k} = m\frac{1-\alpha}{2},$$

which gives

$$k = \log_2\left(\frac{2-\alpha}{1-\alpha}\right) - 1.$$

For a constant α, this gives a constant number of steps, each moving $\Theta(m)$ elements, giving $\Theta(m)$ moves in total. The number of elements inserted in total is $\alpha \cdot m - \alpha\frac{m}{2} = \alpha\frac{m}{2}$, which is $\Theta(m)$. The amortized cost is therefore $\Theta(1)$.

Any choice of α will cause the asymptotic bound of $\Theta(1)$ insertions, but the value chosen has practical implications. A small α will cause earlier reallocations and fewer relocations in the same memory buffer. On the other hand, this will also cause the `slide_set` to use more memory, as the occupancy of the memory buffer will be lower on average. A large α value will use less memory, but as α tends toward 1, running time for ordered insertion will tend toward $\Theta(\log n)$.

For a small enough α, it is possible to eliminate the need for relocations altogether during ordered insertion. Assume that a `slide_set` with n elements has just been reallocated to a memory chunk of size $2m$. Since the reallocation was triggered, we must have $n \geq \alpha m$. In addition, as the previous memory buffer had size m, we also have $n \leq m$. The gaps at each side has size $\frac{2m-n}{2} = m - \frac{n}{2}$. The ordered insertion will fill one of these gaps before triggering either a relocation to the middle or a reallocation to a larger memory buffer. At this point there are $n + m - \frac{n}{2} = \frac{n}{2} + m$ elements in the `slide_set`, and in order to trigger reallocation we need

$$\frac{n}{2} + m \geq \alpha \cdot 2m,$$

but since we have $n \geq \alpha m$, it is enough to find an α such that

$$\frac{\alpha m}{2} + m \geq \alpha \cdot 2m.$$

This is accomplished by choosing α to be $\frac{3}{2}$. Even though this eliminates the need for relocations, it means up to $3n$ memory may be used to store n elements.

Running Time

The analysis of the running time for `slide_set` is largely equal to that of `circular_set`, found in Section 3.3.

Operation	RAM	COB	UMH	Dominating factor
Search	log n	log $\frac{n}{B}$	$\sqrt{n}$ · log n	Binary search
Insertion, random	n	n/B	n log n	Element moves
Insertion, front or back	1	1	log n	Reallocation
Deletion, random	n	n/B	n log n	Element moves
Deletion, front or back	1	1	log n	Reallocation
Iteration	n	n/B	n log n	

Table 3.4: Running times for `slide_set`

Copying and moving

In addition to the copies or moves that a `circular_set` must perform, the `slide_set` must perform relocating moves. For a sensible value of α, between 0.8 and 0.9, about 1.5 to 2.5 relocations may be necessary between two consecutive reallocations. As these relocations cost n each, and are amortized over the $n/2$ elements inserted between reallocations, this means that the amortized number of element relocations per insertion can be as high as 3 to 5. This could make the `slide_set` slower than the `circular_set` for ordered insertion, particularly for elements types with costly copy/move operations.

Iterator Validity

Because elements on both sides of an inserted/deleted may affected by a insertion/deletion (though not at the same time), `slide_set` follows the first and lowest level of iterator validity guarantee. This means that any iterator may be invalidated by insertions and deletions.

Exception Guarantees

The handling of exceptions during insertion/deletion in `slide_set` is similar to that of `boost::flat_set` and `circular_set`, which is discussed in Section 3.2. If the element type has either a `nothrow` copy operation or a `nothrow` move operation, `slide_set` can give the strong guarantee. If none of these operations are `nothrow`, `slide_set` can only offer the weak guarantee.

Element Requirements

The `slide_set` only requires that element types support comparison.

3.5 Merge Set

Inserting k random elements into a set of size n takes $O(k(n+k))$ for ordered array structures like `boost::flat_set`, `circular_set` and `slide_set`, because each insertion may trigger

the movement of $O(n + k)$ elements. Inserting k ordered values in a single operation, however, can be done in $O(n + k)$ time, as the two ranges can simply be merged. This is the basic idea behind the `merge_set`: Initially insert new elements into an input buffer, and periodically merge the input buffer and the main array into a new main array. The goal of this structure is to keep some the benefits of a simple ordered array, while using the input buffer to speed up the insertion procedure. Figure 3.7 shows a `merge_set` with an ordered array as the main structure and using a balanced binary search tree as an input buffer.

Although the internal structure of the `merge_set` is separated into two structures, this can not be allowed to affect its interface. To support viewing the two structures as a unified structure, we have developed a merging iterator, with which it is possible to view separate ordered ranges as a single range. This is achieved by internally keeping two pointers, one for each structure, pointing to two separate elements. Logically, the merging iterator points to the smallest of the two elements. When the merging iterator is incremented the internal iterator pointing to the smallest element is incremented. The merging iterator is explained in detail in Section 3.5.1.

When inserting an element into a `merge_set`, both the main array and input buffer is searched for the element. We define the *lower bound element* of a query element q in a set S to be the smallest element that is equal to or larger than q in S. This search returns iterators to the lower bound elements in both structures. If the element does not already exist, it is inserted into the input buffer, and a merging iterator is returned. This merging iterator internally points to the newly inserted element in the input buffer, as well as to the lower bound element in the main array.

How large should the input buffer be in relation to the main array, and what structure should be used for the input buffer? Let the main array have size n, and let $f(n)$ denote the size of the input buffer. Furthermore, let $g(m)$ denote the time taken to insert an element into an input buffer of size m. Filling the input buffer would then have a running time of $O(f(n) \cdot g(f(n)))$, and the resulting merge operation would have a running time of $O(n + f(n))$. The amortized running time for insertion would then be

$$O\left(\frac{f(n) \cdot g(f(n)) + n + f(n)}{f(n)}\right)$$

$$= O\left(g(f(n)) + \frac{n}{f(n)}\right) .$$

As the first term of this expression grows with growing $f(n)$, and the second term shrinks with growing $f(n)$, the optimal $f(n)$ is one that makes the two terms equal. If the input buffer is also an ordered array, then insertion runs in linear time, so $g(f(n)) = f(n)$. The solution of $f(n) = n/f(n)$ is $f(n) = \sqrt{n}$, so the input buffer should be proportional to the square root of the main array size. Using a balanced binary search tree, however, we have $g(f(n)) = \log_2 f(n)$, and the optimal $f(n)$ is a solution to $\log_2 f(n) = n/f(n)$. The exact solution is

$$f(n) = \frac{n \cdot \log_2 2}{W(n) \cdot \log_2 2} ,$$

where $W(z)$ is the Lambert W function [14]. As the Lambert W function is $\Theta(\log n)$, the running time for insertions is $\Theta(\log n)$. The $\Theta(\log n)$ running time for insertions is also

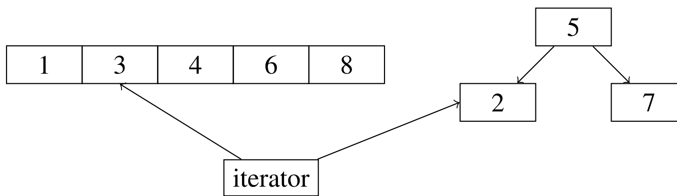

Figure 3.7: Structure of `merge_set`

achived by simply using $f(n) = \Theta(n)$, so this value will be used instead in practice. We will only describe and measure running times for the version of the `merge_set` using a balanced binary search tree of size $\Theta(n)$, as this structure has the most promising asymptotic bounds.

The `merge_set` was not conceived with efficient deletion in mind, and with the structure defined above it is difficult to support fast deletions. The simplest solution, and the one implemented, is to call the delete operations of both substructures, which takes linear time if the element resides in the main array. As most of the element are located in the main array, this is a slow operation. Due to initial dissapointing results of the overall performance of `merge_set`, we did not spend much effort trying to improve the deletion operation.

Running Time

Searching the `merge_set` involves searching both the main array and the input buffer, so total running time depends on which structure dominates. The sizes of the two structures are asymptotically equal, so in the RAM model they both have $\Theta(\log n)$ running time for search. Since the search tree makes less efficient use of the cache, it dominates the runnning time for search in the COB and UMH models. During random insertion, the search tree version only requires amortized $O(1)$ rotations.

Merging and reallocation is not cache-efficient for the search tree, thus iterating over the input buffer dominates the running time in the COB and UMH models. In the RAM model, iterating over both structures take linear time.

Operation	RAM	COB	UMH	Dominating factor
Search	$\log n$	$\log n$	$\sqrt{n} \log n$	Tree search
Insertion	$\log n$	$\log n$	$\sqrt{n} \log n$	Tree Search
Deletion	n	n/B	$n \log n$	Element Moves
Iteration	n	n	$n^{3/2}$	

Table 3.5: Running times for `merge_set`

Copying and moving

The `merge_set` needs no copying or moving when inserting into the input buffer, as this is just insertion into a `std::set`. During merging, however, all n elements are copied or moved. This cost is amortized over the size of the input buffer, call this βn, such that each insertion has the amortized cost $1/\beta$. For a reasonable β between 0.1 and 0.2, this means that insertion costs 5 to 10 element moves. Even though this is a constant factor, it can severely impact performance, especially for large objects.

Iterator Validity

Internally, the merging iterator stores four simpler iterators, thus the merging iterator can only stay valid if none of these iterators are invalidated. In addition to staying valid, the simpler iterators must retain their semantic properties. The four iterators are:

1. Iterator pointing to actual element (in main array or input buffer)
2. Iterator pointing to lower bounding element in opposite structure
3. Iterator pointing to the end of the main array
4. Iterator pointing to the end of the input buffer

If the input buffer is a balanced search tree with stable iterators, no iterators are invalidated unless a merging reallocation is triggered. This is because the main array is not modified during an insertion. The API of the `merge_set` may easily be extended to allow checking for this condition. Even though no simple iterators are invalidated, the lower bounding element may be changed by an insertion, because a new lower bounding element is inserted. This situation is shown in Figure 3.8. The upper figure shows an iterator pointing to the element 2 in the main array, while the lower bounding element in the input buffer is the element 3. The middle figure shows the insertion of the element 2.5, which subtly changes the meaning of the iterator. The iterator is no longer correct, as the lower bounding element is now 2.5. In the lower figure, the iterator is incremented, but since it has no knowledge of 2.5, it now points to 3. The insertion invalidated the iterator, causing it to behave incorrectly when incremented. As these problems are subtle, and not easily detectable by the user, no guarantees are made for iterator validity for insertion.

For deletion, there is the additional chance of modifying the main array, which will invalidate the merging iterator in less subtle ways, so there is certainly no chance of guaranteeing iterator validity for deletions.

Exception Guarantees

The `merge_set` can give the strong exception guarantee if the element type is copyable, or if it is `noexcept` movable. This is easy to see for normal insertion into the input buffer, which itself supports the strong guarantee. When merging is necessary, this is done by allocating a new memory chunk for the new main array. If the allocation fails, the old structure is not corrupted. If elements are copied over to the new array, then a failed copy also leaves the old structure uncorrupted. Moving the elements can only be done safely if the operation is `noexcept`, as the move operation usually changes the element moved from.

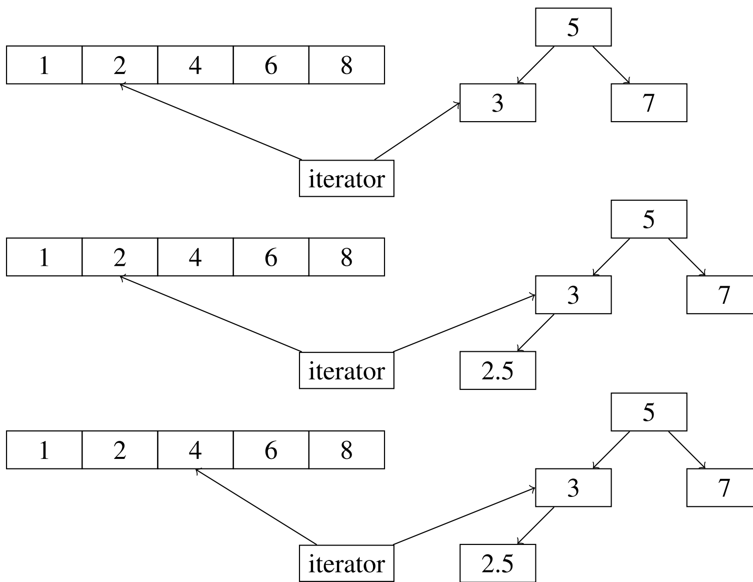

Figure 3.8: Invalidation of iterator in `merge_set`

Element Requirements

The `merge_set` only requires that element types support comparison.

3.5.1 Merge Iterator

It is sometimes useful to view a set of sorted ranges as a single merged range. The C++ standard provides the function `merge()`, but this function physically merges the two ranges into a new container. The purpose of the merge iterator presented in this section is simply to give a view of the merged range, without physically merging the underlying containers. We have only implemented merge iterators for pairs of ranges, but in theory there is no limit to the number of sets that can be merged.

Implementation

The merging iterator uses two iterators to continously keep track of one position for each range, in the same way one would in a merging algorithm. The smallest item pointed to by one of these iterators is the next element in the sequence. Incrementing the merging iterator is simply implemented by incrementing the iterator pointing two the smallest element.

In addition to the two iterators, the merging iterator stores the end iterator for both ranges. This is needed because one range will reach the end before the other. When this happens, one should stop comparing items and only increment the range that has not yet reached its end.

Multiple Ranges

The implementation of merging iterators above have not addressed the possibility of merging more than two sequences. If merges of more ranges are needed, this can be implemented by using the simple two-way merging iterator recursively. The number of comparisons used to increment such a merging iterator tree is linear in the size of the tree.

As each merging step in an N-way merge can be performed in $O(\log(N))$ time, this is not optimal. This problem can be solved in at least two ways:

1. Cache the current item pointed to by each merging iterator node, and make sure the merging tree is balanced. Dereferencing the iterator will be a constant time operation, because the item is cached in the root. Incrementing an iterator will only trigger an update in the subtree containing the smallest item, and recursively this leads to $O(\log(N))$ time increments, which is optimal.

2. Create an N-way merging iterator that internally uses a priority queue to efficiently read and update the positions. The N-way merging iterator could also be implemented by internally using the balanced tree of two-way mergers described above. The first method is flexible because it can be used without the programmer knowing whether the underlying iterators are merging iterators. The problem, however, is that this lack of knowledge means no guarantees on the height of the merge tree. The second solution is less flexible, requiring access to all basic, nonmerging iterators. The advantage of this solution is that the balance of the tree can be guaranteed. The advantage is evene greater if the explicit priority queue is shown to be faster.

Caching Iterator

The need for a caching merging iterator is outlined in Section 3.5.1. This could be implemented in several ways:

1. `merge_iterator` could be changed to always cache its current item. This would guarantee the performance of balanced merge trees, but the storage overhead of caching would be unavoidable.

2. `merge_iterator` could optionally cache its inputs, this guarantees that the iteration is no less efficient than possible given the effieciency of its underlying iterators.

3. A new `caching_iterator` class could be defined, that does nothing but cache the value of its underlying iterator. This adaptor could be wrapped around the underlying iterators for a merging iterator, or wrapped around the merging iterator itself, and model the previous two solutions. From a design perspective, the third solution is preferable, as it divides two unrelated concepts (merging and caching) into two simple classes. From a performance perspective, there is probably not much difference, as a good C++ compiler will be able to remove the apparent indirection.

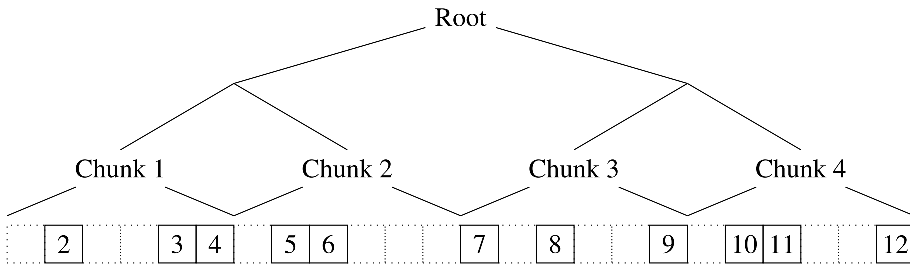

Figure 3.9: Conceptual view of a Packed Memory Array

3.6 Slack Set

Simple structures like `slide_set` and `boost::flat_set` gives optimal iteration speed because they store elements contiguously, but have to move $O(n)$ elements around during insertion. This cost is inevitable if the elements are to be kept in strict contiguous order at all times. This contiguous storage has two major benefits: The code required to iterate over the elements is very simple, resulting in low instruction overhead, few branch mispredictions and a very low chance of instruction cache misses. The iteration is also optimal in the cache-oblivious model, which means it will always efficiently use all levels of the memory hierarchy during iteration. In contrast, iterating over an unrolled linked list can only efficiently use levels of the memory hierarchy where cache line size is smaller than each node, as detailed in section 3.8.2.

The contiguous storage requirement is sufficient for cache-efficient iteration, but it is not necessary, if ignoring constant factors. One possible relaxation of the contiguous storage is to allow for gaps between elements. This is called a packed memory array (PMA) [17]. For a PMA with n elements, we require that the total size of the PMA is $\Theta(n)$. If the total size is βn, for some constant $\beta \geq 1$, iterating over the entire structure will cause $\beta n/B$ cache misses, which is $\Theta(n/B)$, and therefore optimal.

When inserting into a PMA, the insertion point is found by binary search. The only elements that need to move are the elements between the insertion point and the next gap. For a good distribution of gaps, this costs few moves. The tricky part is keeping the gaps well distributed across the PMA. This is done by breaking up the PMA into chunks of constant size k. When an insertion causes a chunk to be full, this triggers a redistribution of elements. The elements must be redistributed over a contiguous set of chunks containing the recently filled chunk.

To search for a suitable set of contiguous chunks, we organize the chunks into a virtual binary tree, where the chunks are leaf nodes. This tree structure is illustrated in Figure 3.9. Each internal node represents all chunks in its leaf nodes. From the leaf representing the recently filled chunk, we walk upwards in the virtual tree, and redistribute over the first viable internal node. An internal node is viable if the average occupancy of its chunks are within bounds. These bounds are given by:

$$\alpha - \frac{\alpha d}{h-1} \leq \text{average occupancy} \leq \beta + \frac{d(1-\beta)}{h-1} \,,$$

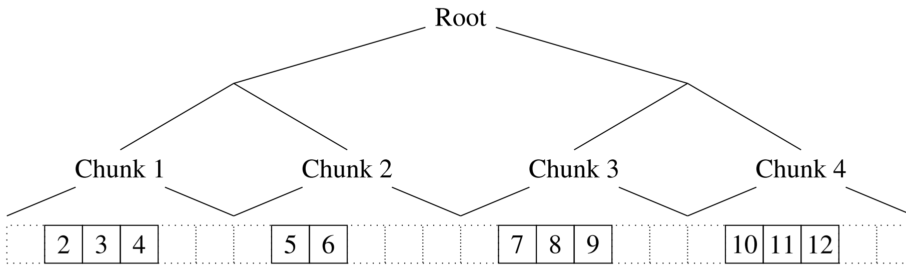

Figure 3.10: Structure of `slack_set`, using `static_slide_set` for chunks

where d is the depth of the node, h is the height of the virtual tree and α, β are parameters of the structure, fulfilling the requirement $0 < 2\alpha < \beta < 1$. The intuitive understanding of these bounds is that the leaves are bounded by $0, 1$, the root is bounded by α, β, and the levels in between have bounds given by a linear interpolation. If the root node fails meet the required bounds, this triggers a reallocation of the entire PMA, similar to a dynamic array.

The key to the amortized running time analysis of insertion is the linear interpolation of bounds. The difference in threshold between each level is $\frac{1-\beta}{h-1}$. At each level d in the tree, a redistribution will only happen when level $d+1$ is out of bounds. Let level d have capacity m, level $d+1$ then has capacity $\frac{m}{2}$. Since the last redistribution of level d, at least $\frac{m}{2} \cdot \frac{1-\beta}{h-1}$ elements have been added, in order to bring level $d+1$ out of bounds. The redistribution itself costs m, which means that the amortized cost of rebalancing level d is $\frac{2(h-1)}{1-\beta}$ which is $\Theta(h)$. Since this cost must be paid for each level of the virtual tree, the total amortized cost is $O(h^2)$.

Setting the chunk size k to a constant we get $h \in \Theta(\log\, n)$, and the amortized worst-case insertion time is $O(\log^2 n)$. A similar analysis is valid for deletions, which has the same amortized bounds. Clearly, the best case is $O(\log\, n)$ for insertion that includes search, and $O(1)$ for insertion at a known position. This is achieved if all inserts are spread perfectly in the array. When a chunk is first filled, this will then trigger a reallocation of the entire array, thus performing no more copies than a dynamic array.

The most direct implementation of a PMA maintains gaps by marking slots as used or unused. Chunks are only a virtual entity, representing an interval of the PMA. Searching in this gap structure requires a modified binary search algorithm, because the middle slot of the range might be unused. Similarly, the used/unused distinction must be checked for every step of an iteration. Fearing that this would lead to code complexity and frequent branch mispredictions, we instead opted for a class representing chunks explicitly. This is the basic idea of the `slack_set`.

The `slack_set` keeps a `std::vector` of chunks, where the class of the chunk is a template parameter of the `slack_set`. In order to retain the cache efficiency of the PMA, the chunk class is required to store elements directly inside the object, not through indirection, so basing the chunk implementation on `std::array` is a natural option. As the size of each chunk is constant, the asymptotic behaviour is not important, so the structure should most importantly be simple. When testing the `slack_set`, we have used a static size version of the

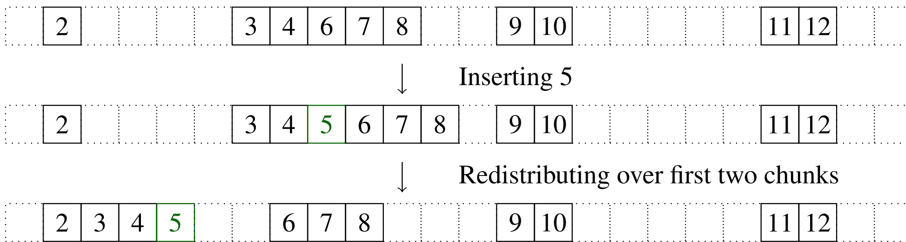

Figure 3.11: Insertion into `slack_set`, triggering redistribution

`slide_set`, termed `static_slide_set`, to represent chunks. The `static_slide_set` internally uses a `std::array` to store elements directly, as opposed to the indirect `std::vector` used by the `slide_set`. The resulting structure is illustrated in Figure 3.10.

Binary search on the `slack_set` is implemented by first searching on the last elements of each chunk, thus locating the chunk where the element must be if it exists. This chunk is then searched by ordinary binary search. This allows for a simple implementation of binary search. Iteration is achieved by using a two-level iterator, which internally keeps an iterator to a chunk and an iterator to an element inside that chunk.

Running Time

The search procedure of the `slack_set` is a slight modification of the simple binary search, but the running time is equal in all models. That is because the last elements of chunks are distributed evenly among all the elements, so the cache is essentially used in the same manner, although the spread nature of the elements will give a $O(1)$ factor increase in the number of cache misses.

Insertion has already been analyzed in the RAM model, where it has an amortized running time of $\Theta(\log^2 n)$. We have already shown that a node at level d we may amortize the m element moves over at least $\frac{m}{2} \cdot \frac{1-\beta}{h-1}$, or $\Theta\left(\frac{m}{\log n}\right)$ elements inserted. In the COB model, the m element moves cost m/B cache misses, so amortized running time is $O\left(\frac{\log^2 n}{B}\right)$.

What is the cost of iterating over m elements in the UMH model? In the worst case, the iteration might need to read a cache line from the slowest layer, which takes $\sqrt{\frac{n}{\alpha}}$ time. This cost dominates the cost of all layers with cache line sizes larger than m. This is because the cache lines are aligned, such that a read at the slowest level ensures that no more reads are necessary in the faster levels. This holds up to the layers with cache lines smaller than m. The first level with cache line size $< m$ requires $O(1)$ cache line reads, for a total cost of $\Theta(m)$. For each faster level beyond this, the number of reads are multiplied by ρ, but the time taken for each read is divided by ρ. For this reason, all layers with cache line sizes less than m have a cost of $\Theta(m)$. Since there are $\Theta\left(\log_\rho \frac{m}{\alpha}\right)$ such levels, the total cost of the m element moves is $O\left(\sqrt{\frac{n}{\alpha}} + m \log_\rho \frac{m}{\alpha}\right)$. The amortized cost for a level

with capacity m is then

$$O\left(\log n \cdot \left[\sqrt{\frac{n}{\alpha m}} + \log_\rho \frac{m}{\alpha}\right]\right) .$$

The $\sqrt{\frac{m}{\alpha m}}$ term is maximized in the leaf nodes, while the $\log_\rho \frac{m}{\alpha}$ term is largest in the root node. Since m has $O(1)$ size in the leaf nodes, and doubles in size for every level upward, the sum of the first term for all levels is $\Theta(\sqrt{\frac{n}{\alpha}})$. As the sum of the second terms for all levels can be at most the root level cost times the number of levels, or $O(\log^2_\rho \frac{n}{\alpha})$, the first term dominates. Thus the amortized running time for insertion is $O\left(\sqrt{n}\log n\right)$.

Operation	RAM	COB	UMH	Dominating factor
Search	$\log n$	$\log \frac{n}{B}$	$\sqrt{n}\log n$	Binary Search
Insertion	$\log^2 n$	$\frac{\log^2 n}{B}$	$\sqrt{n}\log n$	Redistribution
Insertion, random	$\log n$	$\log \frac{n}{B}$	$\sqrt{n}\log n$	Binary Search
Deletion	$\log^2 n$	$\frac{\log^2 n}{B}$	$\sqrt{n}\log n$	Redistribution
Deletion, random	$\log n$	$\log \frac{n}{B}$	$\sqrt{n}\log n$	Binary Search
Iteration	n	n/B	$n\log n$	

Table 3.6: Running times for `slack_set`

Copying and Moving

When insertions and deletions are spread very evenly, at most a constant number of copies/-moves are necessary for each insertion/deletion. However, when the distribution is more skewed, up to $\log^2 n$ copies/moves are necessary per insertion/deletion. Therefore we expect this structure to perform poorly for expensize copy/move types when distributions are skewed.

The current implementation does not support move-only element types. This is because the insertion operation is required to return an iterator to the inserted element. When a redistribution is triggered, a copy is made of the inserted element, in order to search for this element after redistribution is completed. It would certainly be possible instead to store the order of the element with repect to the set elements to be redistributed. A scan could then be made after redistribution, counting the number of elements in order to find the inserted element. We did not have the time to implementat this copy-free alternative, but we excpect is would be possible to do so without adding much overhead.

Iterator Validity

As a single insertion or deletion may cause any number of elements to move, the `slack_set` can not offer any iterator validity guarantees for insertions/deletions.

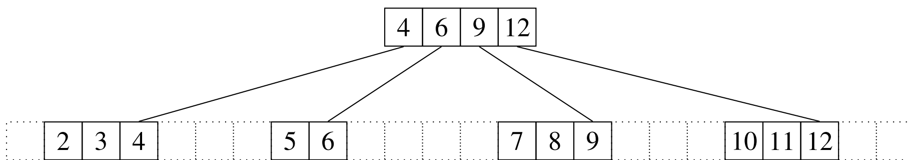

Figure 3.12: Structure of the `proxy_slack_set` searching cache

Exception Guarantees

In the current implementation of `slack_set`, redistribution is done by first copying/moving all elements in the redistribution range over to a temporary buffer. The elements are then copied/moved back in a second pass. If any of these passes are interrupted, the `slack_set` is in an invalid state, and it is impossible to restore validity without repeating element operations that may throw new exceptions. If all elements in the buffer are discarded, this would probably leaves some chunks empty, violating the distribution invariants. Because of this, the `slack_set` can only meet the weak exception guarantee if no `noexcept` copy or move operation exists for the element.

If a `noexcept` copy or move operation does exist for the element type, the slack set can give the strong exception guarantee. When an element is about to be inserted or deleted into a chunk, we can check if the operation will trigger a redistribution. If it will, the redistribution buffer may be allocated ahead of time. If this fails, the operation is aborted without having modified the `slack_set`.

Element Requirements

The `slack_set` only requires that element types support comparison.

3.6.1 Proxy Slack Set

Motivated by the fact that binary search dominates the running time for random insertions in `slack_set`, the `proxy_slack_set` is a variation of the `slack_set` attempting to speed up this binary search. The `proxy_slack_set` simply keeps a cache containing copies of all the largest elements in one contiguous array, as seen in Figure 3.12.

As the `proxy_slack_set` has to store copies of elements, it can only be used for copyable types. This restriction could be lifted somewhat by introducing the notion of a comparable proxy of a non-copyable type, discussed in Section 3.8.1.

3.7 Adaptive Slack Set

The PMA structure presented in Section 3.6 does not take insertion patterns into consideration when redistributing elements, but always redistributes elements evenly, as shown in Figure 3.11. If several insertions will occur in the same part of the PMA in the future, it would be beneficial to leave more gaps in this part during redistribution. This is the basic idea of the Adaptive Packed Memory Array (APMA), presented in [10]. The APMA

achieves the same $O(\log^2 n)$ amortized bound for arbitrary inserts, but can additionally insert contiguous elements in $O(\log n)$ time.

We first show how the APMA achieves the same bound for arbitrary inserts as the PMA. Consider a node in the virtual redistribution tree of the APMA. During redistribution, the node will not necessarily distribute elements equally between its two subtrees. It will, however, ensure that both subtrees have an occupancy that is within the upper and lower limits for this node, as described in Section 3.6. Because this property is satisfied, the analysis of the PMA insertion time is also valid for the APMA. This is called the *rebalance property* in [10].

If we assume that all insertions happen in the same leaf node of the redistribution tree, it is quite easy to see the optimal rebalancing algorithm that satisfies the balancing property. Without loss of generality, let all insertions happen at the start of the APMA. When a node a needs to be redistributed, the optimal policy is to put as many elements as possible in the subtree that will not have insertions in the future, the right subtree. The right subtree will now have occupancy at the upper limit of a. The next time the left subtree of a has occupancy above its bounds, so will a have. Its right subtree is already filled as much as possible, and its left subtree is above the upper limit for a. Therefore, each node will only redistribute once before delegating the task to its parent. The maximum number of times a node can redistribute before reallocation is then the number of redistributions in all levels above the node plus one time before the next reallocation. For the root node, there is only one redistribution. For a node at level d, the maximum number of redistributions is

$$R(d) = 1 + \sum_{i=0}^{d-1} R(i) = 2^d \,.$$

Because the work required to redistribute at level d is $\frac{n}{2^d}$, the total work at each level is $O(n)$. As there are $\Theta(\log n)$ levels in the tree, the total work is $O(n \log n)$. The number of elements inserted to force a reallocation is $\Theta(n)$, so the amortized cost of each insertion is $O(\log n)$.

The $O(\log n)$ bound is shown to hold for the APMA structure of [10]. We have developed a modification of the `slack_set`, inspired by the APMA, but we have not proved that the structure achieves the theoretical bounds. We have called this structure `adaptive_slack_set`↩
. It keeps track of hot spots by keeping an insertion counter for each chunk. When we insert a new element, all insertion counters decay by multiplication with a number ϕ, where $0 < \phi < 1$. Then the insertion counter of the active chunk is increased by one. This ensures that more recent insertions are given more weight. The insertion counters are not maintained at all times, but are only updated when their value is needed. When the value is needed, the time stamp of their last update, t_L, is compared to the a global time stamp of the latest insertion, t_G. The counter is then multiplied by $\phi^{t_G - t_L}$ and the local time stamp is set to the value of the global time stamp.

During redistribution, the insertion counters are used to decide how many elements to store in each subtree. Specifically, the ratio of insertion counter value to gap number should be close to equal for the two subtrees. When this would lead to a violation of the lower or upper bounds, these bounds are used instead.

Running Time

The running time for search is equal to that of `slack_set`. Arbitrary and random insertion/deletion patterns also give the same running time as for the `slack_set`, as random insertions are already dominated by search, and the `adaptive_slack_set` offers no improved redistribution scheme for arbitrary patterns. Indeed, it is impossible to adapt to an arbitrary pattern, as the pattern seen so far can be used to predict future patterns.

The only operation that is improved by `adaptive_slack_set` is repeated insertions in the same area of the `adaptive_slack_set`. The full range of insertion patterns that may benefit from the adaptive rebalance scheme is difficult to describe, but at least it includes insertion of contiguous values. By contiguous we mean that the elements are inserted in order (ascending or descending), and that the inserted elements do not interleave with any preexisting elements in the `adaptive_slack_set`.

If the above condition holds, we can analyze the amortized cost of redistributions between two reallocations. From before we know that a node at level d will need at most 2^d redistributions, each redistributing $\Theta(n/2^d)$ elements. For the RAM model, this means simply that cost for each level is $O(n)$, so the total cost for all levels is $O(n \log n)$. In the COB model, redistributing $\Theta(n/2^d)$ elements costs $\Theta(\frac{n}{2^d B})$ cache misses, so total work is $O((n \log n)/B)$. Thus the amortized cost of redistributions are $O(\log n)$ and $O((\log n)/B)$ for the RAM and COB models respectively.

In the UMH model, moving $\Theta(n/2^d)$ elements has the cost

$$O\left(\sqrt{\frac{n}{\alpha}} + \frac{n}{2^d} \log_\rho \frac{n}{\alpha 2^d}\right) ,$$

so each level d has the cost

$$O\left(2^d \sqrt{\frac{n}{\alpha}} + n \log_\rho \frac{n}{\alpha 2^d}\right) .$$

Starting at the leaf level, the first term may be summed up for all levels as follows:

$$n\sqrt{\frac{n}{\alpha}} + \frac{n}{2}\sqrt{\frac{n}{\alpha}} + \frac{n}{4}\sqrt{\frac{n}{\alpha}} + \cdots + \sqrt{\frac{n}{\alpha}} = O\left(n\sqrt{\frac{n}{\alpha}}\right) .$$

The second term is maximal at the root node, with the value $n \log_\rho \frac{n}{\alpha}$. Using this as an upper bound for all levels, it can easily be seen that the first term dominates the total cost. Amortizing this cost on the $\Theta(n)$ insertions between two reallocations, we get the amortized running time of $O(\sqrt{n})$.

Copying and Moving

Aside from copying/moving fewer elements during contiguous insertion/deletion, the `adative_slack_set`↩
 have the charateristics as the `slack_set`.

Iterator Validity

The `adaptive_slack_set` suffer the same problems as `slack_set`, an can not guarantee iterator validity for any insertion, deletion operation.

Operation	RAM	COB	UMH	Dominating factor
Search	$\log n$	$\log \frac{n}{B}$	$\sqrt{n} \log n$	Binary Search
Insertion	$\log^2 n$	$\frac{\log^2 n}{B}$	$\sqrt{n} \log n$	Redistribution
Insertion, random	$\log n$	$\log \frac{n}{B}$	$\sqrt{n} \log n$	Binary Search
Insertion, contiguous	$\log n$	$\log \frac{n}{B}$	$\sqrt{n} \log n$	Binary Search
Deletion	$\log^2 n$	$\frac{\log^2 n}{B}$	$\sqrt{n} \log n$	Redistribution
Deletion, random	$\log n$	$\log \frac{n}{B}$	$\sqrt{n} \log n$	Binary Search
Deletion, contiguous	$\log n$	$\log \frac{n}{B}$	$\sqrt{n} \log n$	Binary Search
Iteration	n	n/B	$n \log n$	

Table 3.7: Running times for `adaptive_slack_set`

Exception Guarantees

The exception guarantees offered by `adaptive_slack_set` are the same as for `slack_set`↩ . For element types with a `noexcept` copy or move operation, `adaptive_slack_set` can offer the strong exception guarantee, otherwise it may only offer the first level guarantee of not leaking resources.

Element Requirements

The `adaptive_slack_set` only requires that element types support comparison.

3.8 B^+ Set

The B-tree, first presented in [6], is well-known structure specifically designed to store and update large sets of records on permanent storage. Since permanent storage systems usually read and write data in large blocks, the B-tree is organized in a way that minimizes the number of blocks that are read and written. Memory hierarchies also have blocked reads and writes, so the blocked structure of B-trees may improve locality of reference.

A modification of the B-tree, called a B^+-tree [19], was implemented for this project. In a B^+-tree, elements are stored in leaf nodes, with copies stored in internal nodes. Internal nodes store a `std::array` of element copies and pointers to child nodes, while each leaf node stores a `slide_set` of elements. Two independent paramters specify the number of elements per internal node and the number of elements per leaf node.

Running Time

Even though the size of internal nodes and leaf nodes may be different, we simplify the analysis and use b for both. Searching the B^+-tree is done by searching through $\Theta(\log_b n)$ nodes. For each node, binary search is performed to find the correct child node. In the

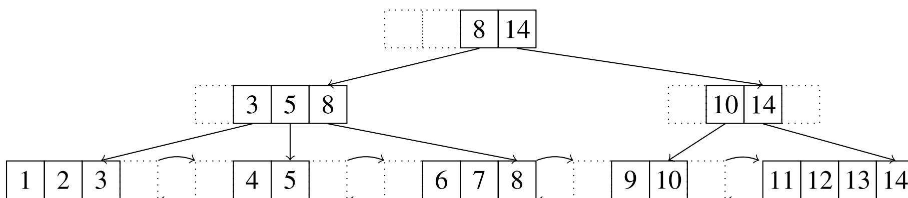

Figure 3.13: Structure of the `b_plus_set`

RAM model, searching a node takes $\log_2 b$ time, so the total running time is $\log_b n \cdot \log_2 b = \log_2 n$.

In the COB model, the search of each nodes costs $\log_2 \lceil \frac{b}{B} \rceil$ cache misses. Thus total running time is $\log_b n \cdot \log_2 \lceil \frac{b}{B} \rceil$. For $b < B$ this reduces to $\Theta(\log n)$. When b is close to B, the running time is $\Theta(\log_B n)$. As b goes toward n, the cost is $\Theta(\log \frac{n}{B})$, and becomes equal to ordinary binary search. This indicates that search is the fastest when b is roughly equal to B, which is not surprising. When B is unknown, or there are many levels of cache, it is difficult to use each level of the cache efficiently.

If we assume that $b < \sqrt{n/\alpha}$, each node fits in one or two cache lines in the slowest cache level in the UMH model. As the cost of reading these cache lines dominate the search time for a single node, each node costs $O(\sqrt{n})$ cache line reads. As there are $\log_b n$ nodes to traverse, total search time in the UMH model is $O(\sqrt{n} \log n)$.

The search procedure dominates the cost of all insertions/operations on the `b_plus_set`↩
.

Operation	RAM	COB	UMH	Dominating factor
Search	$\log n$	$\log \frac{n}{B}$	$\sqrt{n} \log n$	Binary Search
Insertion, random	$\log n$	$\log \frac{n}{B}$	$\sqrt{n} \log n$	Binary Search
Deletion	$\log n$	$\log \frac{n}{B}$	$\sqrt{n} \log n$	Redistribution
Iteration	n	n	$n^{3/2}$	

Table 3.8: Running times for `b_plus_set`

Copying and Moving

The amortized number of copies or moves per insertion/deletion is $O(1)$, but the constant factor depends on the maximum allowed sizes for leaf nodes and internal nodes. As larger nodes lead to more copies or moves, the optimal node size is likely to decrease with increasing copy/move cost for the element type.

Since the search structure keeps copies of elements in the internal nodes, elements must be copyable. However, for many move-only types, it is possible to create a proxy element that may only be used for comparison. This idea is discussed in more detail in Section 3.8.1.

Iterator Validity

The b_plus_set can only offer the weakest iterator validity guarantee.

Element Requirements

The b_plus_set only requires that element types support comparison.

3.8.1 Comparable Proxy

When implementing search structures for sets, it is often useful to store copies of the elements in the set. An example is a B^+-tree where the elements are stored in an iteration friendly way in the leaf nodes, while copies of the elements are used in the internal nodes to direct the search.

Storing copies directly would lead to a requirement that the element types are copyable, which is a non-trivial restriction. It is not unusual to store a set of resource-owning objects, and for unique ownership this means non-copyable. Even when objects are non-copyable, it might be safe to store some identifying information, as long as it is only used for comparison. Consider std::unique_ptr, which is the archetype of exclusive ownership. When storing a set of std::unique_ptrs, it would be safe to copy the pointer value if it is only used for comparisons, never dereferenced. To assure this, the raw pointer could be wrapped in a type which only supports the comparison operators.

Even for copyable types, storing copies in the search structure could be suboptimal. The type could be some large record type, which only uses a few data members for comparison operators. The search tree could then store only these necessary data members in the search structure. For large std::strings, the proxy type could store the first few characters directly in a std::array, to make comparisons for cache-efficient.

The above cases motivates the need for a general interface comparable_proxy<T>, which is a templated class overloading operator()(const T&). The return value is an object of a user-defined type which can be compared to other proxys and to the original elements. The class defaults to returning a simple copy for all copyable types, and in general has no default for non-copyable types. We do define a specialization for the smart pointer types, since this allows for sets of std::unique_ptrs, and avoids unnecessary reference counting overhead for std::shared_ptr.

An alternative to defining the comparable_proxy interface, is to create a map with the requirement that the key type is copyable. Then a set of non-copyable elememts can be created by creating a map from proxy to element. For example, a set of std::unique_ptr↩ <T>s can be implemented as a map from T* to std::unique_ptr. This alternative is arguably clunkier, but is possible without defining a new interface for elements. The main disadvantage of this approach is that since the types of key and value in a map are completely separate, both need to be stored for every element. In the std::unique_ptr example, this means doubling the memory usage.

3.8.2 Unrolled Linked List

The unrolled linked list is a hybrid of an array and a linked list. Instead of storing one element in each node as is done in a linked list, several elements are stored in an array inside each node. This structure has two major benefits over the linked list: Since node pointers are only stored once per node, the memory overhead of these pointers are amortized over the whole array of elements, making the overhead negligible even for moderate size arrays. In addition, since all elements are stored contiguously in each node, the node can be efficiently iterated over in the cache-oblivious model.

The cache efficiency of an unrolled linked list depends on the size of the cache line, B, and the size of each array. The optimal amount of cache misses for an iteration is $\lceil \frac{n}{B} \rceil$. The unrolled linked list has $\lceil \frac{n}{k} \rceil$ nodes. When iterating over the unrolled linked list, each of these nodes must be brought into cache, causing $\lceil \frac{n}{k} \rceil$ misses. In addition, each node will incur up to $\lceil \frac{k}{B} \rceil$ additional cache misses, resulting in a total of $\lceil \frac{n}{k} \rceil \cdot (1 + \lceil \frac{k}{B} \rceil) \approx \frac{n}{k} + \frac{n}{B} \approx \lceil \frac{n}{B} \rceil \cdot (\frac{B}{k} + 1)$ cache misses. For $B \approx k$, this is about twice the optimal. For $B \ll k$ the iteration is close to optimal. For $B \gg k$, however, the unrolled linked list is not able to efficiently utilize the cache.

In practice, there are reasons to limit k to a fairly small number, because typical operations on each array take $O(k)$ time. This means that the structure will effieciently use the top layers of the memory hierarchy, like caches. It will not, however, be able to utilize the lower levels, like memory paging.

3.9 Comparable Hash Set

Hash functions are powerful tools in creating efficient data structures, but can only be used for unordered collections. In this section we present the concept of a *comparable hash function* and its applications in creating efficient ordered collections. This notion of a comparable hash function is a generalization of the interger-only key values of Judy Arrays [25].

Given an ordered set of elements S, a comparable hash function from S to an unsigned integer must satisfy the following property: For elements $e_1 < e_2$ we must have $h(e_1) \leq h(e_2)$. In addition, the set of unsigned integers are represented by a number of bits, w, which is $\Theta(\log n)$. Furthermore, the values returned by the function should be distributed such that the chance of collision is low. A collision happens when $h(e_1) = h(e_2)$ for two elements $e_1 \neq e_2$. In particular, any element should collide with at most $O(1)$ other elements. Given such a function, it is possible to insert, search and delete in an ordered set very efficiently. Specifically, it is possible to do all these operations in a small fraction of w lookups. Even though w is a fixed number for any practical implementation, $w = 64$ in our case, it must be specified as $\log n$ in order to allow for the number of distinct representable integers to grow without bounds.

The demands on the comparable hash function are much stricter than for a normal hash function, since normal hash functions only need to worry about exact identity. A typical example of a problematic type for the comparable hash is a character string. A normal hash function can use the information in all the characters to form its value. The comparable hash function may only concatenate the first few characters. This means that

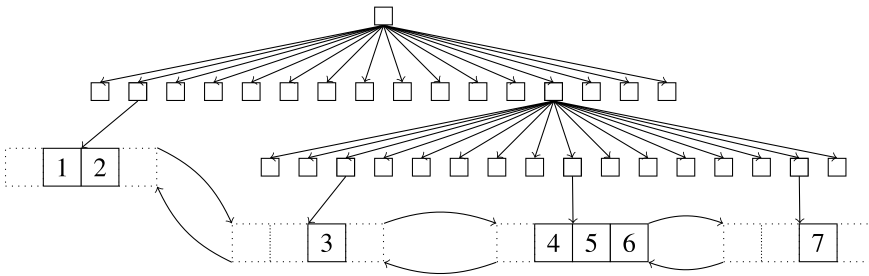

Figure 3.14: Structure of `comparable_hash_set`

collisions will be rampant when many strings begin with the same prefix. An example of such a problematic domain is web addresses, where the first 8 characters may be `http://w` for most of the strings. To accomodate for such types, the concept of comparable hash defined above is probably too narrow, and should be extended to return a value of variable width. However, we believe that the simpler description can lead to a simpler and faster implementation, so the element types that fit the simple description should not need to pay the cost of added complexity. For the same reasons mentioned here, a Judy Array supports two types of key values: integers and character strings.

We have implemented two slightly differing comparable hash structures, `comparable_hash_set` and `comparable_hash_set2`, shown in Figure 3.14 and 3.15, respectively. Both structures have a search tree of constant height connected to a linked list of `slide_set`s at the leaves. For all operations, the search tree can be traversed in $O(1)$ time. At the leaf, the tree is connected to a `slide_set`. The size of each `slide_set` is ideally bounded by a constant parameter, but may grow beyond this size if there are many collisions.

For both structures, each level of the search tree represents a set of possible values for the hash function. For each level, k bits are used to branch to the next level. The branches are created when the number of elements stored in the subtree grows above a constant threshold. The root node represents all possible values. When the number of elements grow above the threshold, 2^k children are created, representing all possible values of the first k bits. Each element is sent to the child node representing its first k bits, where the same operation might happen recursively at some later time.

Since there are only w bits there can only be $\lceil \frac{w}{k} \rceil$ levels in the search tree. At the lowest possible level, each node represents just one value of the hash function, so the number of elements in this node is bounded by the number of colliding elements with this value.

In the `comparable_hash_set`, each leaf node is connected to its own `slide_set`. When a node is split, a `slide_set` is created for each non-empty child node. Since each `slide_set` is part of the linked list, empty `slide_set`s would slow down iteration.

Even if there are no empty `slide_set`s in the `comparable_hash_set`, the linked list may consist of many very small `slide_set`s. This is addressed in `comparable_hash_set2`. Instead of keeping one `slide_set` for each leaf node, nodes may share a `slide_set`. We define the size of a node A to be the number of elements in the subtree rooted in A. When the size of a node grows beyond half the maximum size of a `slide_set`, the node is split, and all children point to the same `slide_set`. Each `slide_set` is also augmented with a `std::vector` of back pointers to all leaf nodes pointing to the `slide_set`. When the size of the

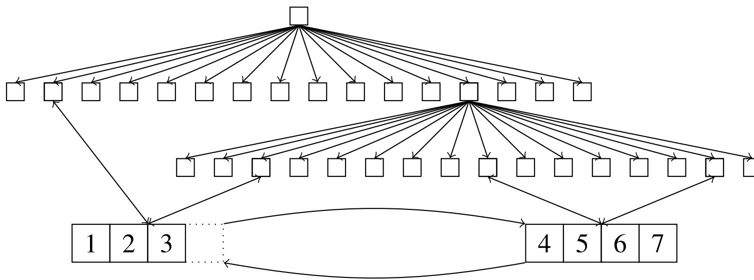

Figure 3.15: Structure of `comparable_hash_set2`

`slide_set` grows above the threshold, a new `slide_set` is created, and roughly half of the elements are moved. The nodes are divided into a left and right set in such a way that the difference in size is minimized. The worst split is achieved when the biggest node is exactly in the middle, because this would lead to a bad split no matter which set the node is put in. However, since no node is bigger than half the threshold, the split can be no worse than $(\frac{1}{4}, \frac{3}{4})$.

By using this sharing policy, the `comparable_hash_set2` can guarantee a lower threshold on the number of elements in each `slide_set`, at the cost of a more complicated insertion strategy.

Running Time

With integer sizes $w = \log n$, and using k bits for each node, the height of the search tree is at most $\frac{\log_2 n}{k}$. Each jump node access leads to a cache miss, as the nodes are located at arbitrary positions in memory. A simple amortization argument can be used to show that splitting and merging nodes do not contribute to the theoretical running time.

In the RAM and COB model, running time for all operations is simply $O(\log n)$, and the running time is $O(\sqrt{n} \log n)$ in the UMH model.

Copying and Moving

The hash-based structures only copy or move elements in single constant-size `slide_set`s during normal insertions/deletions, and the costs of splitting and merging nodes are amortized over many insertions and deletions. However, for elements that are expensive to copy or move, inserting into the `slide_set` may incure a large cost. Because of this, we expect the optimal maximum size for `slide_set`s to be smaller for expensive elements.

Iterator Validity

Because insertion into the `slide_set` may cause many elements to move, there hash-based structures can not offer any iterator validity guarantees beyond level one.

Exception Guarantees

Inserting elements into `slide_set`s means that exceptions during copying or moving may cause bad state. therefore, `noexcept` copying or moving is required for the hash-based structures to offer the strong guarantee, otherwise only the weak guarantee may be offered for insertions and deletions.

Element Requirements

In addition to the standard comparison operation, the hash-based structures require the elements to support the comparable hash-interface, and the elements must be distributed evenly to benefit from the hash-based structures.

Chapter 4

Use cases

For all of the structures compared in this report, the performance is measured for a common set of use cases. The uses cases consist of a set of operations along with a element type. The use cases are chosen to highlight the performance impact of different usage patterns, and to aid in choosing an appropriate structure for a given use. This section presents the different elements types used and the sets of operations performed.

4.1 Element Types

The element types used in this report are divided by two characteristics, size and indirection. For size, the most importatnt distiction is between elements small enough to fit several elements into a cache line, and elements larger than a cache line. The smaller types will benefit more from contiguous storage in memory, as several elements can be loaded by a single cache line read. Larger element types may still benefit from other cache effects of contiguous storage, like prefetching.

The second important distinction made in this report is between direct and indirect elements. Direct elements store their data in the same memory location as the element itself. In indirect elements, on the other hand, only a pointer to the data is stored in the same memory location as the element itself, and the data is stored in some other location. This has important implications for performance. Comparing indirect elements typically incur an extra cache line read, as the actual data has to be read. On the other hand, moving an indirect element can be achieved by simply moving a small pointer, which is typically cheap. Copying an indirect elements is expensive, as it usually involves allocating memory for the copied data, and then copying the data to the newly allocated memory.

To represent small, direct element types, `int` is used. Comparing, moving and copying are all cheap operations for the `int`, and accessing contiguous `int`s make effective use of the cache.

The representative of large, indirect elements is the `std::string`. This element type consist of a pointer to an array of characters. The number of characters is 50 for experiments

in this project. The `std::string` type is characterized by slow comparisons due to cache misses, fast moves and slow copies.

Finally a custom type has been implemented to represent large, direct object. This type has been dubbed `record`, and uses `std::array` to store 50 characters directly. This type does not suffer an extra cache miss during comparisons, but both moves and copies are slow.

4.2 Operation Sets

The operation sets used in this project have two basic motivations. One motivations is to simulate real usage of the structures, another motivation is to understand why structures differ in their performance. For example, deleting all elements in a structure one by one is not a realistic scenario, because it would be much quicker simply call `clear()`. On the other hand, a mix of equal amounts of insertions and deletions is a very realistic scenario, but is best understood if insertion only and deletion only scenarios are also tested.

4.2.1 Search

We have used two search scenarios, random order and ascending order. While random order search is perhaps the most general, the ascending search scenario is important to illustrate how the structures perform when access patterns are more structured. Searching in ascending order can model how searching performs for ordered insertions and deletions, and can also model the speedup gained when query items are localized in the structure. The scenarios are measured in the following ways:

- **Random order** The structure is filled with randomly generated elements, and copies of the elements are stored in a `std::vector`. The `std::vector` is shuffled, and used as queries for the search. The time taken to search for all n queries is then measured.

- **Ascending order** The structure is filled with randomly generated elements, and copies of the elements are stored in a `std::vector`. The `std::vector` is shuffled, and used as queries for the search. The time taken to search for all n queries is then measured.

4.2.2 Insertion

Because expected insertion performance of the different structures vary significantly with the insertion pattern, there are four different insertion only scenarios. In all these scenarios, n elements are first generated randomly and inserted into a `std::vector`. The elements are then ordered according to the scenario. The time taken to insert all n elements into an originally empty structure is then measured. The scenarios are:

- **Random order** Elements are shuffled randomly before insertion. This is perhaps the most realistic insertion pattern.

- **Ascending order** Elements are sorted, such that each inserted element is larger than previous elements.

- **Descending order** Elements are sorted in reverse order, such that each inserted element is smaller than previous elements.
- **Middle order** Elements are ordered such that each inserted is the new median value. This scenario is included to represent insertions that are highly localized, but not localized at the lower or upper end.

4.2.3 Deletion

Although the excpected performance for deletion vary with the deletion pattern as much as for insertion, we have not included as many different scenarios for deletion as for insertion. This is because, except for `merge_set`, deletion performance is excpected to be similar to insertion performance. So in order to reduce the amount of use cases, we have only included a random order deletion scenario.

In this scenario, the structure is filled with n randomly generated elements, and copies are stored in a `std::vector`. The `std::vector` is shuffled, and the time taken to delete all the elements of the `std::vector` from the structure is measured.

4.2.4 Iteration

The iteration scenario is important for ordered sets, because answering range queries efficiently is a feature not shared by unordered sets. In this scenario, the structure is filled with n ramdomly generated elements. Each element type is given a evaluation function, which converts the element to a number. The evaluation function is the identity function for `int`, while for `std::string` and `record` the sum of all characters is returned. The time is measured for iterating over all elements of the structure, and summing up the evalution of each element. The actual operation performed on each element is not very important, but an important requirement is that the operation should depend on the data. In that way, reading the data can not be left out by the compiler during optimization.

4.2.5 Mixed Insertion/Deletion

This scenario is included to measure the performance of insertions and deletions in the absence of large changes in structure size.

In this scenario, the structure is filled with n randomly generated elements, and copies are stored in a `std::vector` and shufffled. In addition, another `std::vector` is also filled with n randomly generated elements and shuffled. Deleting elements of the first `std::vector` from the structure is then interleaved with inserting elements of the second `std::vector` into the structure, and the time taken for these n insertions and n deletions is measured.

Chapter 5

Results

5.1 Experimental Setup

All experiments were run on an Intel ®Core ™i7-3517U @ 1.90GHz CPU. Cache sizes were 32KB, 256K and 4096KB for L1, L2 and L3, respectively. The source code was compiled with the g++ compiler, version 4.8.1, using the -O3 optimization option.

Each measurement was repeated a minimum of 50 times, and was further repeated until the standard deviation of the mean was a small fraction of the average measured value.

5.2 Tuning

Some of the structures tested have some tunable parameters. These include block sizes for all structures dividing elements into fixed-size blocks, occupancy limits for the PMA-based structures. These paramteres were tuned specifically to element types, but parameters are constant for all use cases. The motivation was that while the user will typically know the type of element stored, it is more difficult to know which use case presented here that most resembles an the users actual usage pattern. Furthermore, the actual usage pattern is likely to be a mix of several use cases, so results based on less specific tuning will probably generalize better to actual usage patterns.

The use of tuning parameters is nonetheless an inconvenience for the user, and especially when using these structures in a generic context, where the specific element type is not known. The optimal solution would be if parameters could be automatically tuned at compile-time, using no input from the user. This could be simple for direct types like `int` and `record`, because tuning could be based on element size, but it would be difficult for indirect types like string. As there is no simple way to know whether an object is direct or not, automatic compile-time tuning for general types seems impossible.

When tuning the `slack_set`, we found that the occupancy limits heavily influenced its ability to handle ordered inserts. To get reasonable performance on these insertion patterns, the occupancy limit at the root level had to be set to 50%, which means that the structure

uses up to $4n$ space after a reallocation. The `adaptive_slack_set` was able to handle much higher density (90%). This benefit of the `adaptive_slack_set` is not visible in the results below, but is important to note.

5.3 Search

5.3.1 Random Order

The measured running times for search is shown in Figure 5.1. For int, most structures perform equally well, with the exception of `std::set` and `merge_set`. For $n = 300\,000$, `std::set` uses 500 ns per query, `merge_set` uses 267 ns, while all other structures use between 120 ns and 170 ns. Although most structures have quite similar performance for search, the hash-based structures do have a small advantage for $1\,000 < n < 100\,000$, while the `b_plus_set` is at a slight disadvantage.

For sets of `std::string`s, the performance gap between `std::set` and the other structures is musch smaller, while `merge_set` performs worse than `std::set`. The performance of the hash-based structures is significantly better than all other structures, and especially the `comparable_hash_set`. For $n = 1000$, the `comparable_hash_set` search is about twice as fast as the fastest non-hash-based structure.

For sets of `record`s, `std::set` and `merge_set` again shows significantly worse performance compared to the other structures. The hash-based structures perform better than other structures, but the difference is less significant than for `std::string`. This use case also shows a small, but significant difference between the simple, array based structures, and the more advanced `slack_set` and `b_plus_set`. The exception to this pattern is `proxy_slack_set`, which is more similar to the simple array structures in performance.

5.3.2 Ascending Order

The measured running times for search is shown in Figure 5.2. As expected, all structures benefit from ordered search, as each mostly visits the same parts of the structure as the previous search, thus utilizing the cache very efficiently.

For sets of ints, all structures show a clear performance drop when the size of the structure grows larger then the L2 cache. Most structures show similar performance, except for very good performance shown by `comparable_hash_set`, and poor performance by `std::set` and `merge_set`.

The hash-based structures also perform well for sets of `std::string`s, while `merge_set` is significantly slower than other structures. All other structures show remarkably similar performance.

For sets of `record`s, the hash-based structures are less clearly faster than other structures, but `std::set` is yet again significantly slower than the other structures.

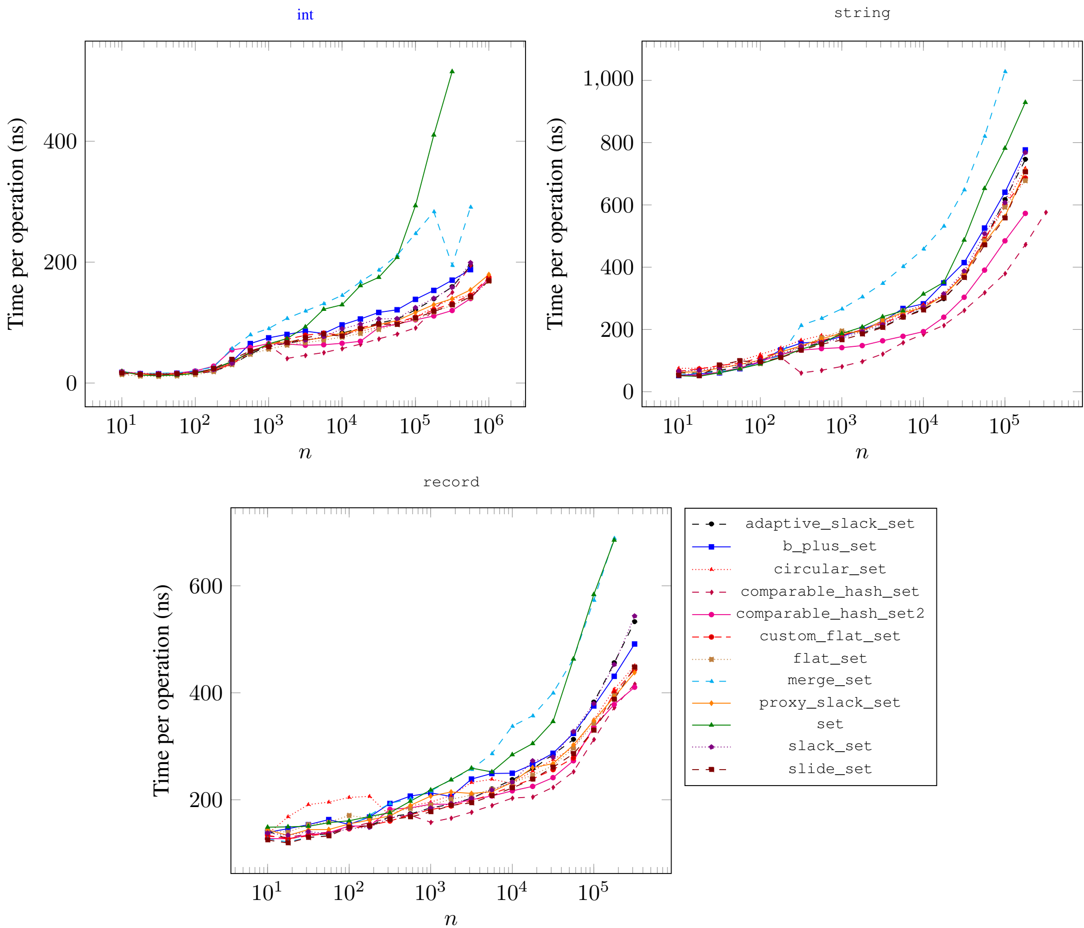

Figure 5.1: Searching for elements in strucure with size n

5.4 Insertion

5.4.1 Random Order

Results for random order insertion are shown in Figure 5.3. For sets of int s, the best overall performance is achieved by `b_plus_set`, `slack_set`, `proxy_slack_set` and `comparable_hash_set`↩ . The `comparable_hash_set2` has good performance for $n > 10\,000$, but performs eratically for small n. The performance of `std::set` is competitive for $n < 10\,000$, but for larger n the relative performance tapers off. The performance of `merge_set` is overall poor, and the `adaptive_slack_set` performs significantly worse than the other PMAs. The `boost:`↩ `flat_set` performs poorly for all n, but `custom_flat_set` shows that this is mostly an implementation issue. The `circular_set` is competitive for $300 < n < 4000$, and `slide_set` is the fastest structure for $n < 4000$.

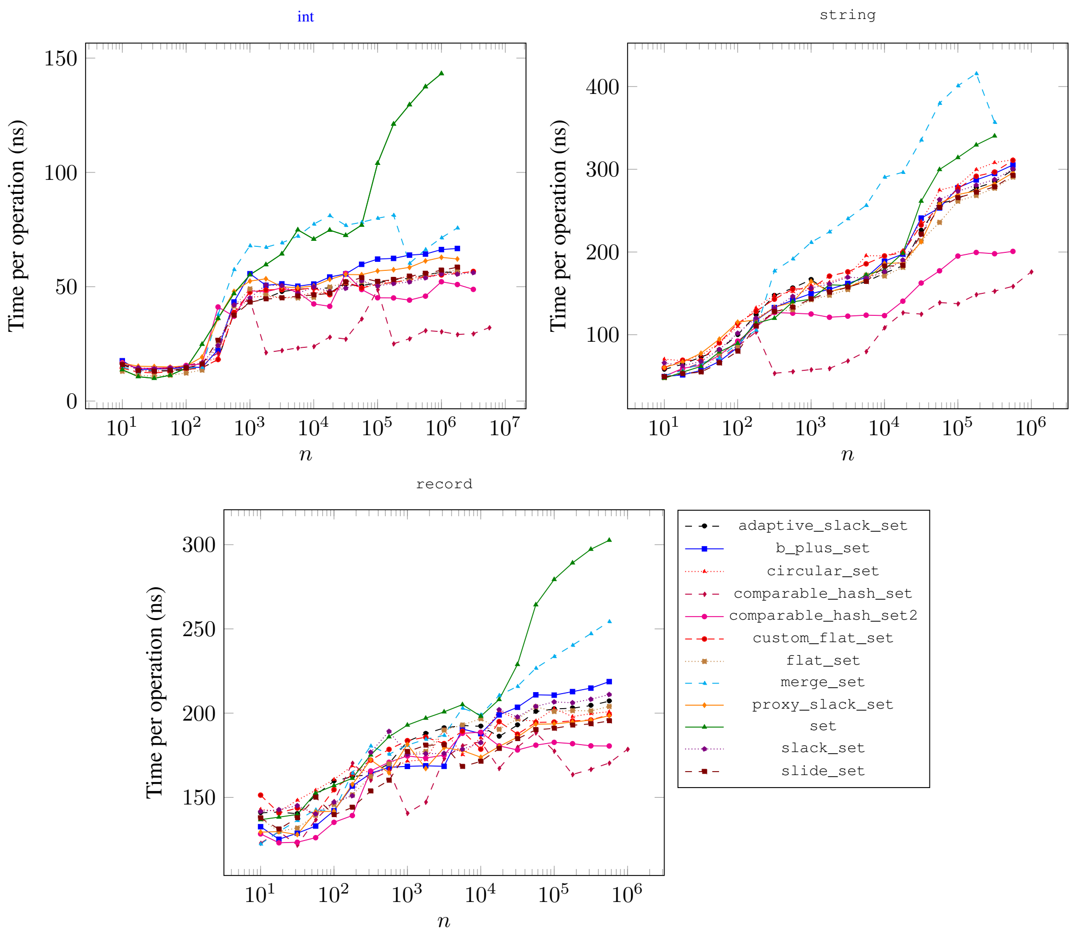

Figure 5.2: Searching for elements in strucure with size n

The situation is a bit different for sets of `std::string`s. While the best structures for `int`s also do well for `std::string`s, the `std::set` now outperforms many of them. The best structure, however, seems to be the `comparable_hash_set`. The array based structures are not competitive even for small n.

For `record`s, the `std::set` is the best performing structure for small n, while `comparable_hash_set`↩ performs best for $n > 2500$.

5.4.2 Ascending Order

Results for ascending order insertion are shown in Figure 5.4. For sets of `int`s there is no competition to the `slide_set` and `custom_flat_set`, which consistently inserts in < 9 ns for large n. The `circular_set` also have an essentially constant insertion time < 30 ns. Performance is also consistently good for `boost::flat_set` and `b_plus_set`. The hash-based

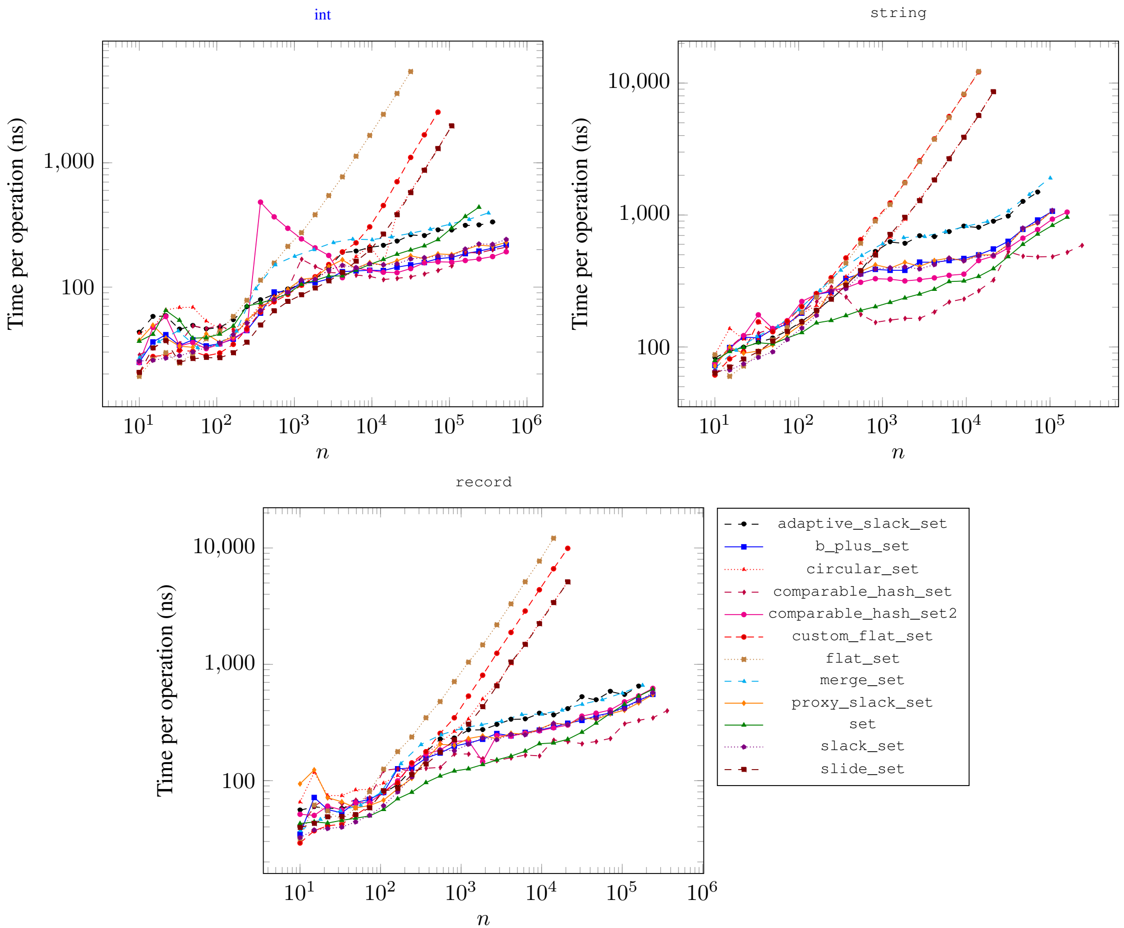

Figure 5.3: Inserting n elements in random order

structures are generally fast, but runnig time is very inconsistent for different n. The other structures also perform ordered insertions quite fast, ≈ 200 ns per insertion for $n \approx 1\,000\,000$, but is an order of magnitude slower than the fastest structures.

For sets of `std::string`s, the `slide_set` and `custom_flat_set` still performs best, but the gap is smaller. The hash-based structures are consistently good for different n, and competitive with `circular_set` for large n. The `std::set` outperforms the PMA-based structures, and perform almost as well as the `boost::flat_set` and `b_plus_set` for large n.

The same trends generally hold for sets of `record`s as for `std::string`s, except that `custom_flat_set` outperforms the `slide_set`.

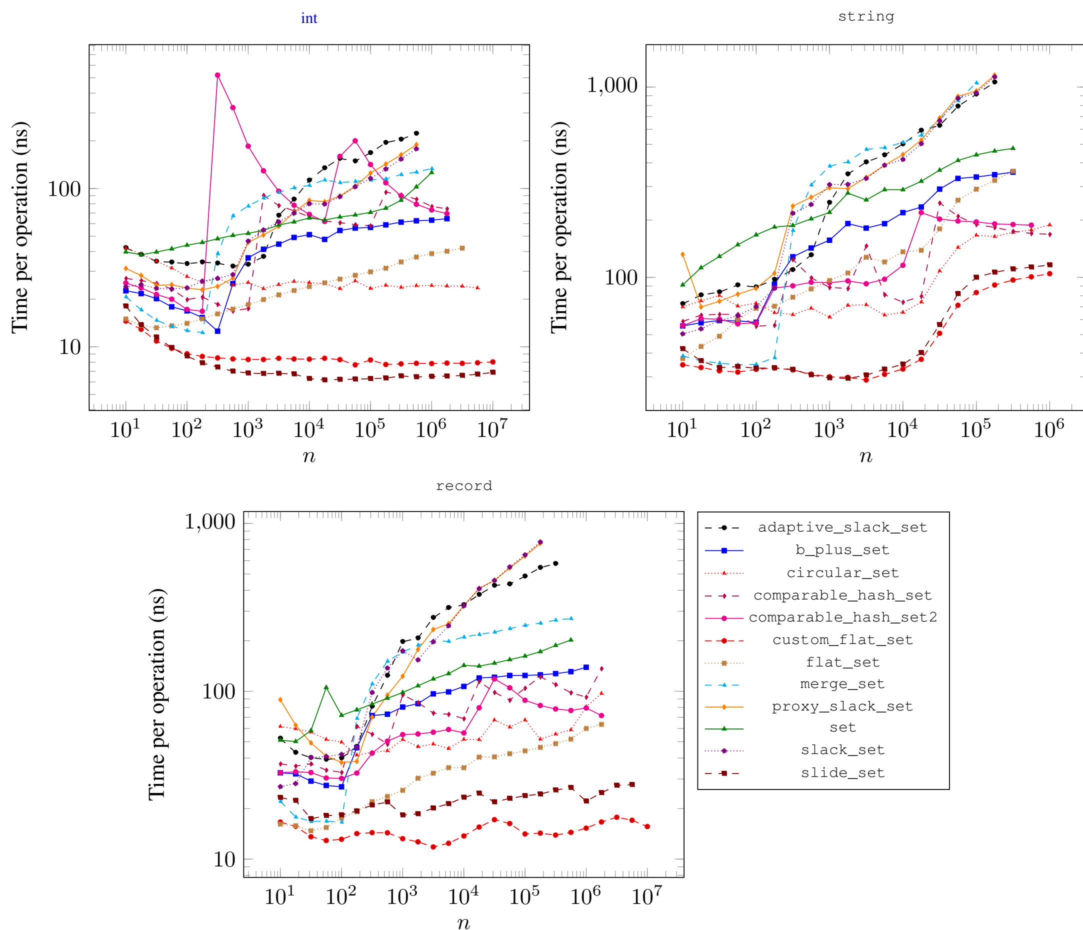

Figure 5.4: Inserting n elements in ascending order

5.4.3 Descending Order

Results for descending order insertion are shown in Figure 5.5. These results are similar to the results for ascending order insertion. The biggest exception is `boost::flat_set` and `custom_flat_set`, which now unsurprisingly show very poor performance for almost all element types and values of n. The only exception is that `custom_flat_set` is competitive for small sets of `int`s. Another difference is that the gap between `slide_set` and the other structures is smaller, especially for sets of `std::string`s.

5.4.4 Middle Order

Results for middle order insertion are shown in Figure 5.6. By middle order we mean that the nest element inserted always becomes the new median element of the set. For large val-

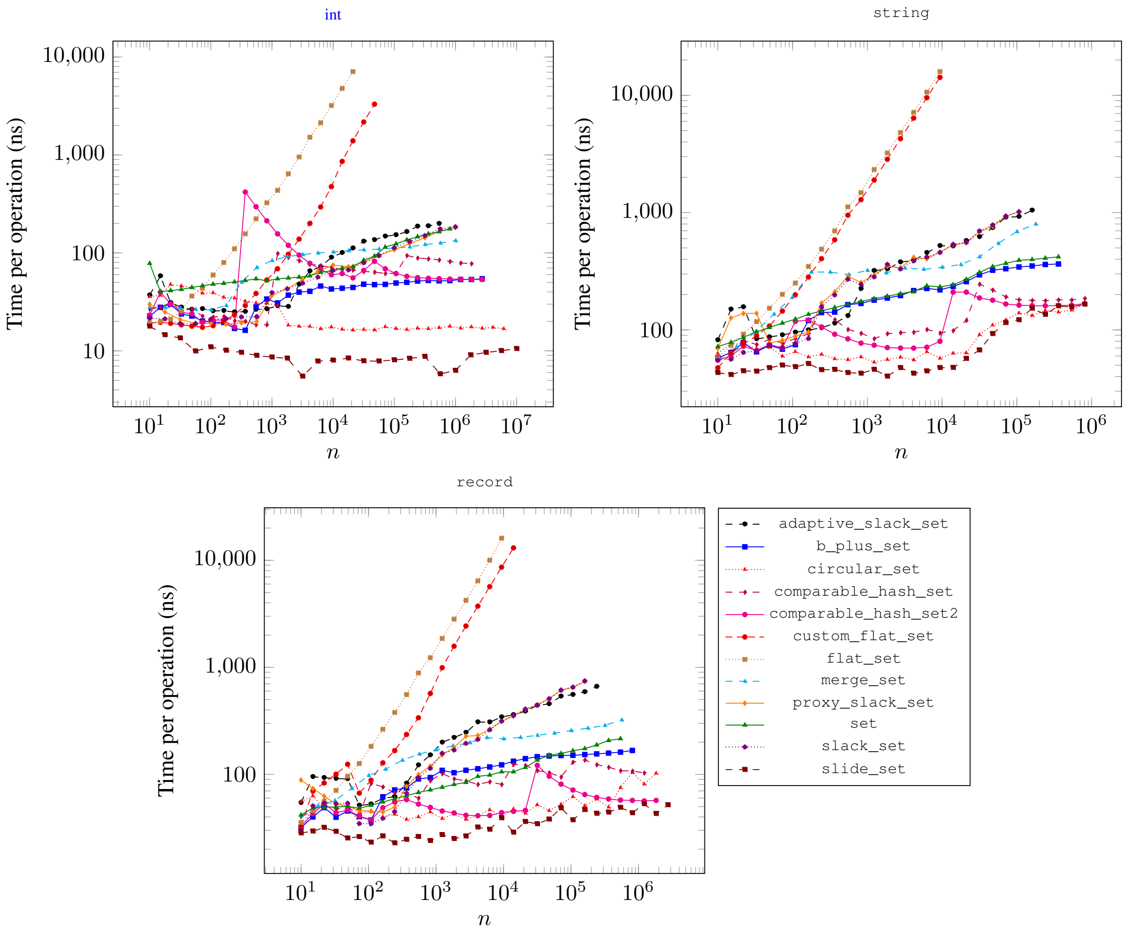

Figure 5.5: Inserting n elements in descending order

ues of n, no simple array based structures perform well, but `slide_set` is the best performer for small sets of `int`s. The `b_plus_set` is the best structure overall sets of `int`s, but the hash-based structures are equally good for large n. The hash-based structures do show the same inconsistent timings as for other insertion patterns, and especially `comparable_hash_set2` is very slow for small n.

For sets of `std::string`s, the best results are achieved by the `comparable_hash_set`, while `std::set` also achieve good overall results. The `b_plus_set` and `comparable_hash_set2` are competitive for large n, but are significantly slower than `std::set` for smaller n. The same trends continue for sets of `record`s.

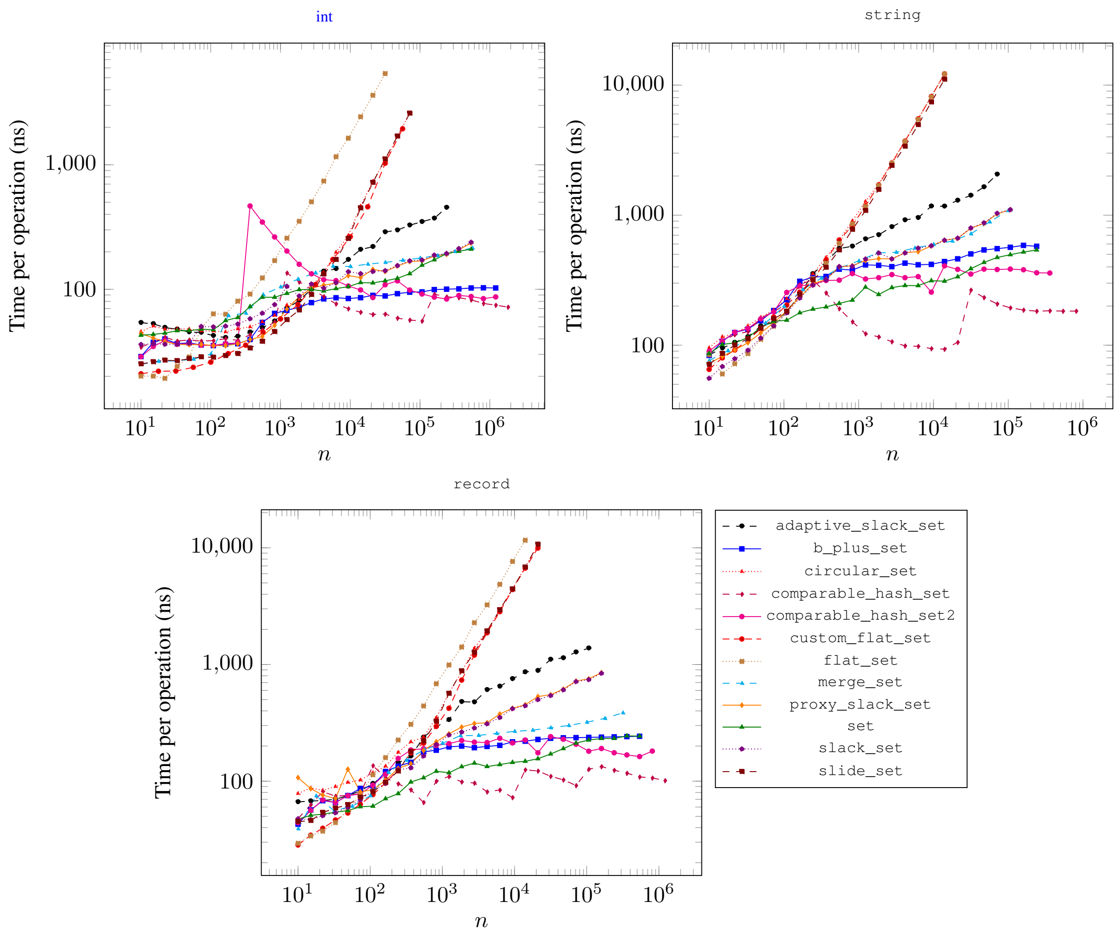

Figure 5.6: Inserting n elements in middle order

5.5 Deletion

Running times for deletion in random order are shown in Figure 5.7. The simple array based structures and the `merge_set` are much slower than other structures, although the `slide_set` and `merge_set` perform reasonably well for very small n. For sets of int↩ s, the `std::set` is significantly slower than the fastest structures for $n > 10\,000$. The `comparable_hash_set` outperforms the other structures for all element types, though the gap is more consistent for `std::string`s and `record`s.

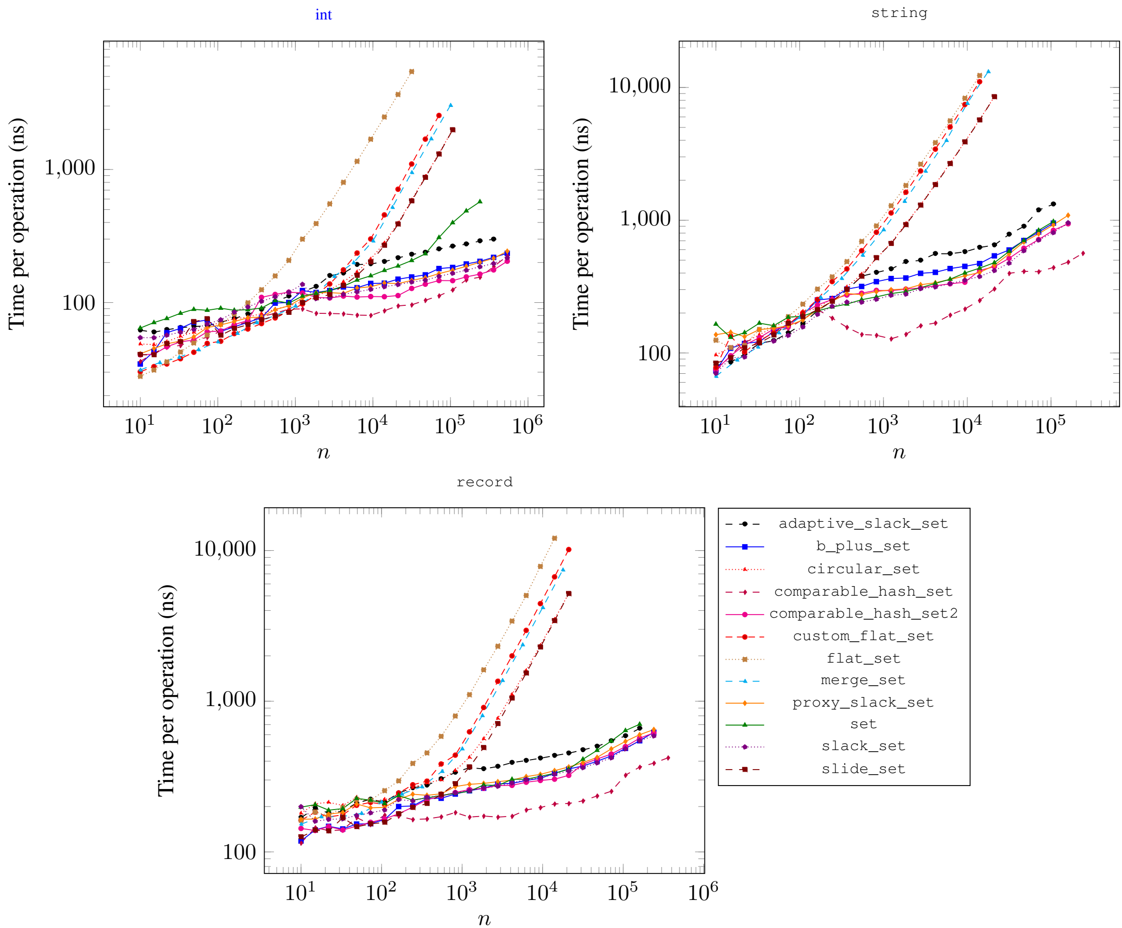

Figure 5.7: Deleting all n elements in random order

5.6 Iteration

Running times for iteration are shown in Figure 5.8. Iteration speed is almost identical for most structures, with a few notable exceptions. The `std::set` is much slower than all other structures, and for large sets of `int`s, it is up to 40 times slower than most other structures, and almost 70 times slower than `boost::flat_set`.

The three structures with fully contiguous elements, `boost::flat_set`, `custom_flat_set` and `slide_set`, are significantly faster than all other structures for sets of `int`s, probably benefiting from their simple iterator types.

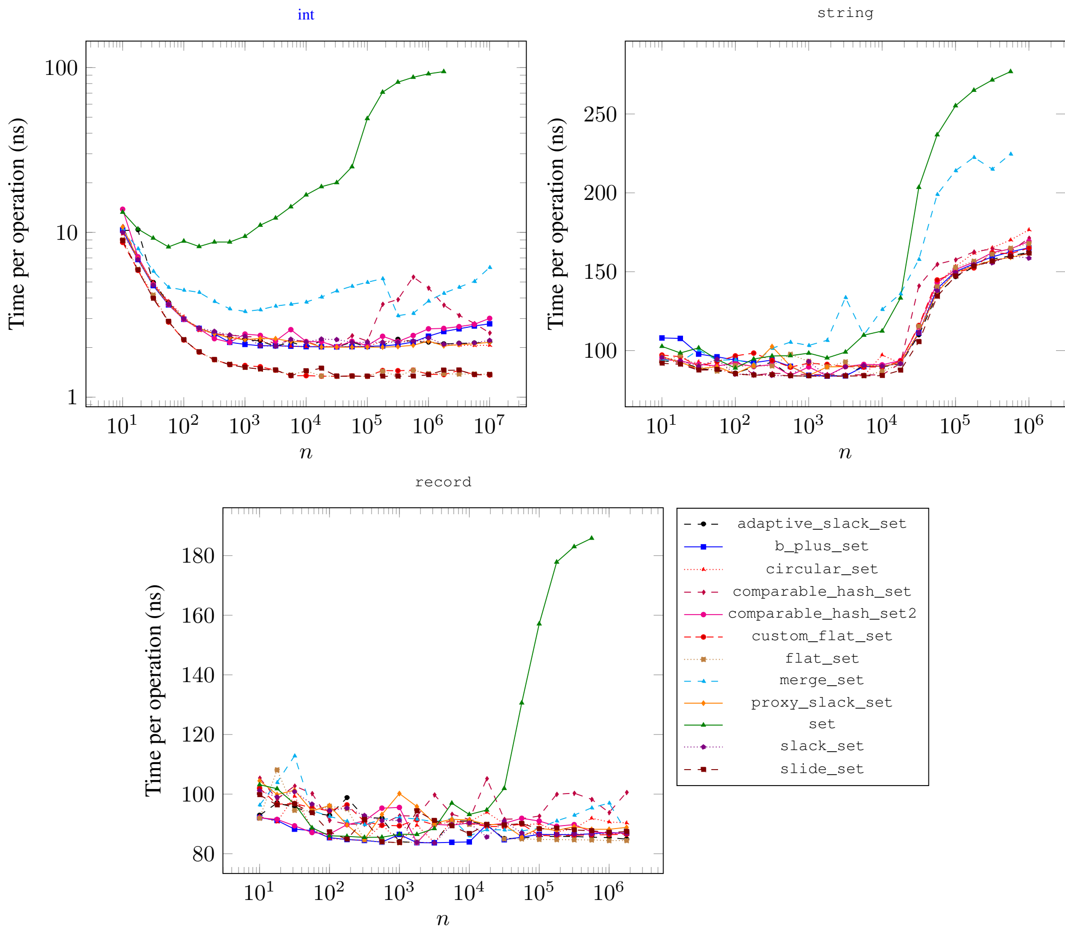

Figure 5.8: Iterating over structure with n elements

5.7 Mixed Inserton/Deletion

Running times for iteration are shown in Figure 5.9. Performance is remarkably similar for all non-quadratic time structures. The `std::set`, however, is significantly slower than other structures for sets of `int`s. The `adaptive_slack_set` is slower for all element types, while the `comparable_hash_set` is significantly faster than all other other structures for the heavier element types.

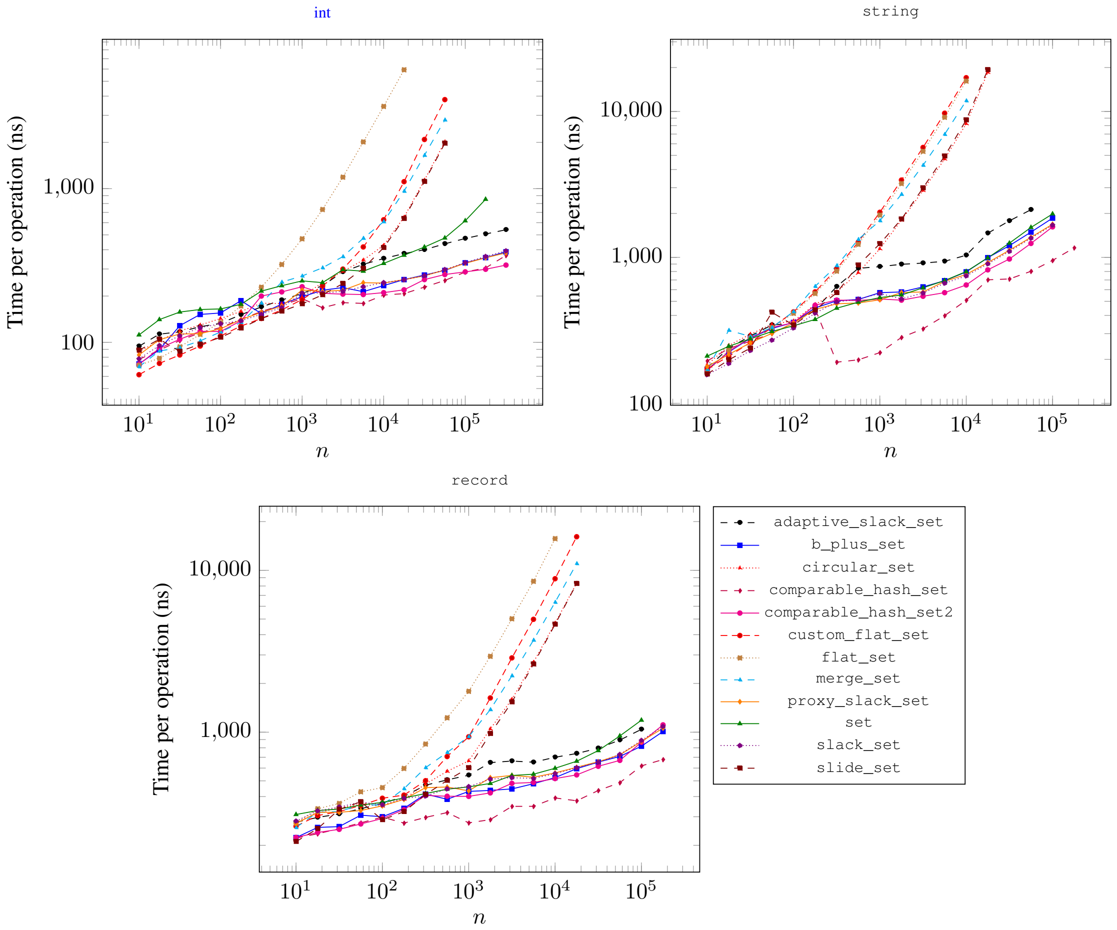

Figure 5.9: Mixed insertion/deletion in structure of size n

Chapter 6

Discussion

6.1 Data Structures

6.1.1 Set

When discussing the running times for `std::set` in comparison with other structures, it is important to remember the additional gurarantees given by `std::set`. It is is the only structure to give the strong exception guarantee, even when both the copy and move operation might throw an exception. It is also the only structure for which iterators are stable. This allows for many efficient usage patterns that are not tested in this report, as it can not be supported efficiently by the other structures. For example, consider a set of objects who want to register themselves as observers to another object, where the observed object may want to query on the set of observers. If every observer is given an iterator to its entry in the observer set, it may unregister itself in constant time, and this is only made possible by stable iterators.

That being said, the `std::set` is quite a bit slower than other structures for many important use cases. One important weakness of `std::set` is search, in particular for `int`s, and this has a large impact most of the other operations. Search times are similar to insertions times for all insertion patterns, meaning that search is probably the most costly part of the insertion procedure. This matches well with `std::set`s performance when inserting into sets of `std::string`s and `record`, where searching is not as slow compared to other structures. For these data types, `std::set` and `b_plus_set` are the best performing general purpose structures that do not require comparable hashing.

The second major weakness of `std::set` is iteration. As it potentially incurs a cache miss for every single element visited, performance is more than an order of magnitude worse than other structures for sets of `int`s. The gap is much smaller for the larger element types, and is quite competitive while the set of elements can still fit in the LLC.

6.1.2 Flat Set

The `boost::flat_set` is perhaps the simplest structure tested in this project, and in general it performs exactly as escpected. It cannot efficiently insert elements except at the upper end, but its simple memory layout means that binary search is fairly efficient, only beaten by the hash-based structures. The fact that elements are contiguous in memory also means that pointers to subranges may be passed directly to lower level C libraries.

For insertion in ascending order, the `boost::flat_set` is significantly slower than `slide_set` and `circular_set`, even though all these structures move $O(1)$ elements. This is because the `boost::flat_set` performs a full binary search for every insertion, while the other two structures include a check to see if the inserted element is larger than the last element. This is verified by the fact that `custom_flat_set` is just as fast as the `slide_set`.

In general, there are few uses cases where `boost::flat_set` ot `custom_flat_set` is among the best performing structures, and in all those use cases the `slide_set` performs about equally well. There might be an exception to this for insertion in middle order with very small n, but even then the difference is not very large.

6.1.3 Circular Set

In general, `circular_set` achieves the goals for which it was designed. Performance is very good for front and back insertion, but constant factors seem to be higher than for `custom_flat_set` and `slide_set`. There are no use cases in which `circular_set` performs better than `slide_set`, even though the `circular_set` moves fewer elements. In addition, the iterators for `circular_set` are slightly more complex to support the wrap-around, which leads to a noticeable slowdown when iterating over `int`s.

6.1.4 Slide Set

Although likely not to be completely novel, the `slide_set` was not based on any prior work known to the author, and is a structure with several benefits. The simple layout in memory makes iteration and binary search just as fast as for `flat_set`, and the ability to slide from side to side in a memory buffer makes insertions in both ends possible.

Even though the `slide_set` moves more elements than the `circular_set` during insertion in order, performance is much better, probably because of the simplicity of the structure.

For any sizeable n, however, the $O(n^2)$ running time makes it unusable for anything but very specific insertion patterns. For sets of `int`s, the performance is competitive up to a few thousand elements. For the heavier element types, however, `std::set` is faster for $n > 100$.

Even though the `slide_set` has $O(1)$ insertion at both ends, this performance benefit is brittle. If an element is to be inserted just one position from the end, a full binary search is needed. This could be mitigated by starting a search at both ends simultaneously. In each iteration, the distance from the search positions to their respective end of the set could be doubled, and thereby the search complexity would be logarithmic in the distance from the element to its nearest endpoint. This would be a more robust way of speeding up insertions and deletions close to one of the endpoints, but would probably add some unnecessary overhead when the element was not close to the end.

6.1.5 Merge Set

Even though the merge_set is able to achieve logarithmic insertion times while iterating over elements much faster than std::set, the running times for most operations are not impressive. It seems that the frequent global merging leads to very large constant factors, such that the merge_set is not very competitive for insertion, for any n. Performance of deletion is very poor, but this was expected due to the quadratic deletion complexity of the main linear array. Earlier on in the project, more advanced deletion mechanisms were considered, but the slow insertion times gave little motivation to add more complexity to the structure.

In general, merge_set was not found to be a suitable structure for any of the use cases considered. It is unable to give the same iterator validity and exception guarantees as std::set, so the faster iteration times comes at too high a price.

6.1.6 Slack Set

The starting point of this project was experiments showing the performance lead of random insertions into a slack_set compared to a std::set for large sets of ints. The slack_set is faster than std::set for several use cases, like search and iteration, but is not a real competitor to b_plus_set. Theoretically, the slack_set is more cache-efficient than b_plus_set for iteration, but in practice performance is very similar for the two structures.

This theoretical benefit was the prime motivation to build a structure with all elements sorted in one large array, as unrolled linked lists are much more flexible.

The proxy_slack_set did not perform significantly different from slack_set, which can be taken as a sign that the overhead during

6.1.7 Adaptive Slack Set

Instead of being faster than the slack_set for ordered insertions, the adaptive_slack_set is able to get similar performance with almost half the memory usage. This effect did not manifest for middle order insertions, which probably shows a weakness in the specific insertion counter strategy implemented. This could probably be refined to handle middle order insertion just as well as ascending/descending order.

For random insertions, the slack_set and adaptive_slack_set should perform about the same number of redistributions, so the low performance of the adaptive_slack_set is probably due to inefficiencies in the redistribution algorithm. Instead of redistributing elements in large blocks like the slack_set, the adaptive_slack_set must interleave the redistribution with extra logic dealing with the insertion counters. If this could be improved, there is a potential for the adaptive_slack_set to be a better performing and more general alternative to the slack_set.

Even if this is possible, however, it would probably not be able to outperform the b_plus_set.

6.1.8 B$^+$ Set

The b_plus_set is perhaps the most versatile of all structures tested. For sets of ints, it beats the std::set for all use cases, while for heavier element types none of the two structures consistently beats the other.

Perhaps most importantly, the b_plus_set has insertion/deletion performance comparable to std::set, while search and iteration are both far better. One of the main reasons to use ordered sets rather than unordered sets is support for range queries, and these consist of searching and iterating. For this reason, it might be desirable to include an alternative to std::set in the C++ standard, supporting the characteristics of b_plus_set.

It is not entirely clear, however, how a such structure could be specified by the C++ standard. The only obvious change is that the new structure cannot have stable iterators, or at least stable iterators must be in addition to faster, unstable iterators. A goal of the C++ standard is not to specify implementation details, and the asymptotic running times of the B$^+$ sets is equal to that of balanced binary search trees, when analyzed in the RAM model. The only precedent in the C++ standard for specifying cache-efficient layout is std::vector, in which elements are guaranteed to be contiguous in memory. One possible minimal specification is that the new structure introduces some notion of a partial contiguity of elements. That is, for some integer b, no fewer than b elements are stored contigously in memory. This property must be satisfied by all but one block of the structure, namely the root block. The structure must also be allowed to store up to $O(\log_n)$ copies of each element, but no more than $O(n)$ copies in total. Aside from this, the structure should have the same complexity guarantees as std::set.

6.1.9 Comparable Hash Set

The hash-based structures are among the fastest structures for all use cases tested, although they do show some erratic behaviour, especially comparable_hash_set2. It is remarkable that the they perform about as fast as circular_set and slide_set for descending order insertion into large sets of strings, when the latter two structures can do this with constant amortized running time.

In addition to the great performance achieved by the structures, there is a large potential for improvement, as several variants of these basic structures could be explored. Overall, the comparable_hash_set shows the best performance, probably because they avoid the added complexity of sharing leaf nodes. However, the fact that leaf nodes in comparable_hash_set may be arbitrarily small, leads to less efficient iteration. This is evident when iterating over large sets of ints.

Chapter 7

Conclusion

Based on the results and experience gathered during this project, the author has come to the following conclusions:

- The slide_set can be a useful alternative to boost::flat_set. In addition to being about as fast as the boost::flat_set for middle and ascending order insertion, the slide_set can insert in descending order in $O(1)$ time, as well as perform random insertions with half as many moves as the boost::flat_set. Experiments with more advanced data structures also show that slide_set and its static counterpart can be used efficiently as a basic building block of these structures.

- A new specification of ordered sets based on the characteristics of b_plus_set could be a useful addition to the C++ standard. This would make searching, iteration and specifically range queries much more efficient for small data types.

- Structures built upon sparse arrays like the packed memory array structures tested in this project show no significant benefits compared to structures using unrolled linked lists. Even though unrolled linked lists are theoretically inefficient, this effect does not seem to be an important concern for the set sizes tested in this project. Since unrolled linked lists are far more flexible, practical cache effiecient set structures should probably be based on these rather than on sparse arrays.

- The most promising approach to dramatically improving search performance for ordered sets is to explore hash-based structures. Even though the b_plus_set is theoretically able to exploit the cache better than simple binary search, actual search performance is remarkably similar. The hashed-based structures, on the other hand, has shown the potential to dramatically speed up search.

7.1 Future Work

As discussed in Section 6.1.9, the node sharing strategy of comparable_hash_set2 leads to significant overheads, while the small leaf nodes of comparable_hash_set can lead to less

consistent iteration speeds.

A possible strategy to combine the best of both worlds is for the bit ranges of each node to be dynamic. By being able to increase the bit range of a node in small increments, rather than jumping from 0 to 8 during a node split, it could be possible to avoid small leaf nodes without resorting to leaf node sharing. This strategy also has a possible benefit in the other direction: The bit range of the root node may be increased beyond the 8 bits tested in this project, to allow for constant time search for very evenly distributed element sets.

If the hash-based structures are to be generalized to less evenly distributed data sets, the simple node types used for this project are not suitable. Consider a set of 64-bit integers for which the most significant 32 bits are all zero. Small bit ranges would waste little space, but would take many meaningless jumps to get to the interesting bits. A large bit range would traverse fewer nodes, but waste lots of space for unused nodes. Although it is clear that another node type is needed, it is not clear what this node type should look like, or if indeed only one additional node type is needed. The addition of new node types must be accompanied by simple dynamic strategies to choose node types based on element distribution. The static decision of which composition of nodes are best for searching a set of elements is probably a difficult problem in itself. The fact that the decision must be taken dynamically, quickly, and taking into account the cost of changing the composition, possibly makes this a very challenging task.

Bibliography

[1] David Abrahams. *Exception-Safety in Generic Components*. Jan. 2014. URL: http://www.boost.org/community/exception_safety.html.

[2] M AdelsonVelskii and Evgenii Mikhailovich Landis. *An algorithm for the organization of information*. Tech. rep. DTIC Document, 1963.

[3] Bowen Alpern et al. "The uniform memory hierarchy model of computation". In: *Algorithmica* 12.2-3 (1994), pp. 72–109.

[4] Matt Austern. "Why you shouldn't use set-and what you should use instead". In: *C++ Report* (2000). URL: ftp://24.151.202.80/AiDisk_a1/Full/Completed/pdf/col1.pdf.

[5] Rudolf Bayer. "Symmetric binary B-trees: Data structure and maintenance algorithms". In: *Acta informatica* 1.4 (1972), pp. 290–306.

[6] Rudolf Bayer and Edward McCreight. *Organization and maintenance of large ordered indexes*. Springer, 2002.

[7] Pete Becker et al. *Working draft, standard for programming language C++*. Tech. rep. Technical Report, 2011.

[8] Michael A Bender, Richard Cole, and Rajeev Raman. "Exponential structures for efficient cache-oblivious algorithms". In: *Automata, Languages and Programming*. Springer, 2002, pp. 195–207.

[9] Michael A Bender, Erik D Demaine, and Martin Farach-Colton. "Cache-oblivious B-trees". In: *SIAM Journal on Computing* 35.2 (2005), pp. 341–358.

[10] Michael A Bender and Haodong Hu. "An adaptive packed-memory array". In: *ACM Transactions on Database Systems (TODS)* 32.4 (2007), p. 26.

[11] Michael A Bender et al. "A locality-preserving cache-oblivious dynamic dictionary". In: *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics. 2002, pp. 29–38.

[12] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. “Cache oblivious search trees via binary trees of small height”. In: *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics. 2002, pp. 39–48.

[13] Stephen A Cook and Robert A Reckhow. “Time bounded random access machines”. In: *Journal of Computer and System Sciences* 7.4 (1973), pp. 354–375.

[14] Robert M Corless et al. “On the LambertW function”. In: *Advances in Computational mathematics* 5.1 (1996), pp. 329–359.

[15] Ion Gaztanaga. *Boost Intrusive Library*. Jan. 2014. URL: `http://www.boost.org/doc/libs/1_55_0/doc/html/intrusive.html`.

[16] Ion Gaztanga. *Boost Container Library*. Jan. 2014. URL: `http://www.boost.org/doc/libs/1_55_0/doc/html/container.html`.

[17] Alon Itai, Alan G Konheim, and Michael Rodeh. *A sparse table implementation of priority queues*. Springer, 1981.

[18] Zardosht Kasheff. “Cache-oblivious dynamic search trees”. PhD thesis. Massachusetts Institute of Technology, 2004.

[19] Donald E Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching,” 1973*.

[20] Ig-hoon Lee et al. “Cst-trees: cache sensitive t-trees”. In: *Advances in Databases: Concepts, Systems and Applications*. Springer, 2007, pp. 398–409.

[21] Charles E Leiserson et al. *Introduction to algorithms*. The MIT press, 2001.

[22] Harald Prokop. “Cache-oblivious algorithms”. PhD thesis. Massachusetts Institute of Technology, 1999.

[23] William Pugh. “Skip lists: a probabilistic alternative to balanced trees”. In: *Communications of the ACM* 33.6 (1990), pp. 668–676.

[24] Riku Saikkonen and Eljas Soisalon-Soininen. “Cache-sensitive memory layout for binary trees”. In: *Fifth Ifip International Conference On Theoretical Computer Science–Tcs 2008*. Springer. 2008, pp. 241–255.

[25] Alan Silverstein and D Baskins. “Judy IV shop manual”. In: *h ttp://judy. sourceforge. net/doc/shop interm. pdf* (2002).

[26] Herb Sutter. *Exceptional C++: 47 engineering puzzles, programming problems, and solutions*. Addison-Wesley Professional, 2000.

[27] Robert Endre Tarjan. *Efficient top-down updating of red-black trees*. Princeton University, Department of Computer Science, 1985.

[28] Jeffrey Scott Vitter. “External memory algorithms”. In: *AlgorithmsESA98*. Springer, 1998, pp. 1–25.