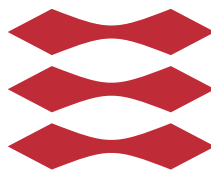


# Data Structures for Arrays

Thomas Søren Henney

DTU



Kongens Lyngby 2017

Technical University of Denmark  
Department of Applied Mathematics and Computer Science  
Richard Petersens Plads, building 324,  
2800 Kongens Lyngby, Denmark  
Phone +45 4525 3031  
[compute@compute.dtu.dk](mailto:compute@compute.dtu.dk)  
[www.compute.dtu.dk](http://www.compute.dtu.dk)

# Summary (English)

---

This thesis investigates different ways to implement data structures for indexable dynamic sets such as arrays that provide efficient element retrieval, insertion, and removal. The traditional array has constant time retrieval, but insertion and removal have  $O(n)$  time complexities, which can be improved by a constant factor with dynamic arrays. This paper introduces the tiered vector and analyses and discusses 1- and 2-tiered vectors for a total of seven different implementations. Insertions and removals in 2-tiered vectors have an amortised time complexity of  $O(\sqrt{n})$ . This time complexity can also be deamortised at the cost of extra space consumption. Element retrievals run in  $O(1)$  time. It is shown that 2-tiered vectors can achieve very fast running times when implemented cleverly by utilising bit tricks. Retrieval is slower than in the standard dynamic array implementation in C++ by a factor 1.6 to 6.5. The insertion and removal times in the 2-tiered vectors immediately outperform any other implementation. Until the 2-tiered vectors contain upwards of a million elements, their insertion and removal times are also notably faster than those of balanced binary search trees whose asymptotic running times are  $O(\log(n))$ .



# Preface

---

This thesis was prepared at DTU Compute in fulfilment of the requirements for acquiring a B.Sc. in Engineering within Software Technology.

The thesis deals with data structures for representing ordered dynamic sets of elements with efficient element retrieval, insertion, and removal.

Copenhagen, 16-June-2017



Thomas Søren Henney



# Acknowledgements

---

I would like to thank my advisors, Inge Li Gørtz and Philip Bille for all their help and guidance throughout the entire project period, always pointing me in the necessary direction whenever I was in doubt or on the wrong track. The knowledge they have provided in their several courses at DTU has also been crucial to writing this thesis. I would also like to thank Andreas Halkjær From for providing big academical support and inspiration as well as friendship throughout the past three years. A special thanks goes out to my girlfriend, Adelina Yafasova, who has supported me through the difficult and less entertaining times for many of the past years, but especially during the past semester.





# Contents

---

<b>Summary (English)</b>	<b>i</b>
<b>Preface</b>	<b>iii</b>
<b>Acknowledgements</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Existing Data Structures for Arrays</b>	<b>3</b>
2.1 The Array . . . . .	3
2.2 The Dynamic Array . . . . .	4
2.2.1 Element insertion . . . . .	4
2.2.2 Element removal . . . . .	4
2.2.3 Element retrieval . . . . .	5
2.3 Binary Search Trees . . . . .	5
2.3.1 Element retrieval . . . . .	5
2.3.2 Element insertion . . . . .	6
2.3.3 Element removal . . . . .	6
2.3.4 Balanced Binary Search Trees . . . . .	6
<b>3 Tiered Vectors</b>	<b>9</b>
3.1 The 1-Tiered Vector . . . . .	9
3.1.1 Element retrieval . . . . .	10
3.1.2 Element insertion . . . . .	10
3.1.3 Element removal . . . . .	11
3.2 The Simple 2-Tiered Vector . . . . .	11
3.2.1 Element insertion . . . . .	12
3.2.2 Element removal . . . . .	14
3.2.3 Element retrieval . . . . .	16

3.3	2-Tiered Vector using a Deque . . . . .	16
3.3.1	Element retrieval . . . . .	16
3.3.2	Element insertion . . . . .	17
3.3.3	Element removal . . . . .	19
3.4	Improving the Tiered Vectors with Bit Tricks . . . . .	19
3.5	Improving Worst Case Performance . . . . .	20
3.5.1	Deamortising Insertion and Removal Costs . . . . .	21
3.5.2	Generalising . . . . .	23
<b>4</b>	<b>Implementation</b>	<b>25</b>
4.1	Data Structures . . . . .	25
4.2	Unit Testing . . . . .	28
4.3	Benchmarks . . . . .	29
<b>5</b>	<b>Analysis</b>	<b>31</b>
5.1	Asymptotic Time Complexities . . . . .	31
5.2	Analysis of Benchmarks . . . . .	32
5.2.1	Element retrieval . . . . .	33
5.2.2	Element insertion . . . . .	35
5.2.3	Element removal . . . . .	37
5.2.4	Deamortisation . . . . .	37
<b>6</b>	<b>Conclusion</b>	<b>41</b>
<b>A</b>	<b>Project Plan</b>	<b>43</b>
A.1	Description . . . . .	43
A.2	Teaching Goals . . . . .	44
A.3	Self-evaluation of the Project Process . . . . .	44
	<b>Bibliography</b>	<b>45</b>

# CHAPTER 1

## Introduction

---

This thesis focuses on different possible ways to implement array-like data structures. Arrays are essential for many computational problems and essential for most non-trivial programs. Therefore, optimising them can provide significant benefits.

The problem at hand is to find an efficient data structure to be used for arrays. This data structure must represent an arbitrarily large dynamic set of elements in a certain order and provide the following operations:

1. *getElementAt( $i$ )*: Known as element retrieval. It returns the element stored at index  $i$  of the data structure.
2. *insertElementAt( $i, e$ )*: Inserts the element  $e$  at index  $i$  such that the element currently at that index and any other elements to the right have their indices shifted up once.
3. *removeElementAt( $i$ )*: Removes (and possibly returns) the element at the specified index  $i$  and shifts all elements at higher indices once to the left so as to not leave any gap.

This thesis will briefly discuss the traditional array used in many programming languages and its limitations. Arrays are often replaced by dynamic arrays as these are essentially arrays with much improved asymptotic runtime for insertion and deletion at the end of the array. After introducing this well-known data

structure, this thesis will describe the tiered vector originally introduced by Goodrich and Kloss[5]. The tiered vector is a general data structure which can be implemented in several ways, providing slightly different runtime trade-offs. This project will explore possibilities for creating new state-of-the-art data structures on top of the described ones.

After introducing each data structure, the data structures will be analysed and compared with one another in both a theoretical and empirical setting.

## CHAPTER 2

# Existing Data Structures for Arrays

---

The following chapter will cover the data structures that are most commonly used when fast retrieval, insertion, and deletion are desired.

## 2.1 The Array

Arrays are commonly used to perform retrieval, insertion, and deletion operations. Element retrieval in arrays obviously takes  $O(1)$  time.

Insertion and deletion, however, take order of the number of elements in the array to perform. The length of the array must correspond to the number of elements stored. Whenever a new element is inserted, a new array with capacity for one extra element must be allocated, and the elements of the old array have to be moved one at a time to the newly allocated array, clearly taking linear time. Therefore, the array is not a very efficient solution.

## 2.2 The Dynamic Array

The less naive approach to using an array is to make it dynamic. Dynamic arrays are arrays with slight overhead. The underlying array has a given length,  $m$ . Along with the length, we also store the number of elements in the data structure,  $n$ , separately.

### 2.2.1 Element insertion

Inserting into the dynamic array at index  $i$  is done by shifting the element currently at the position and every following element once toward the end of the underlying array. This makes room to insert the element, increasing  $n$  by one. The running time thus becomes  $O(n - i)$  since all elements at index  $i$  and above must be moved.

**Increasing capacity** If the underlying array is full, however, we allocate a new array of double the original size and move every element to their respective indices before inserting the new element. This does not change the amortised running time of the operation, which we show with the accounting method. If we only allocate a new array once the size has doubled, this means that we must have inserted  $\frac{n}{2}$  elements since the size was last increased. As such, we must have performed  $O(\frac{n}{2} \cdot (n - i))$ . Adding onto that the cost of allocating and moving all current elements the total cost becomes  $O(\frac{n}{2} \cdot (n - i) + n)$ . Thus, the average cost for operations becomes  $\frac{\frac{n}{2} \cdot (n - i) + n}{\frac{n}{2}} = n - i + 2 = O(n - i)$ . However, the worst case running time remains  $O(n)$ . This can be bypassed by having the second array readily allocated, and whenever a new element is added, the new element, along with one of the old ones, is added to the second array. This requires a constant factor more memory, which becomes insignificant asymptotically and still maintains the worst case  $O(n - i)$  insertion time.

### 2.2.2 Element removal

Removal from index  $i$  in the dynamic array is basically the opposite process of insertion. We simply shift every element at index  $i + 1$  and above once down and decrement  $n$ . In order to minimise space consumption, it is also possible – but not strictly necessary – to halve the size of the array once a certain minimum threshold for the number of elements in the array is reached. We will let this

threshold be  $n \leq \frac{m}{4}$ . The running time for removal thus becomes  $O(n - i)$  by the same argument as for insertion, including for the analysis of the amortised running time.

For both element insertion and removal it should be noted that unless  $i$  a number lower than  $n$  by some constant, both operations will run in  $O(n)$  time. For instance if the dynamic array is used as a stack, elements will always be inserted and removed at the end, making these operations  $O(1)$ .

### 2.2.3 Element retrieval

Element retrieval is performed trivially by indexing directly into the underlying array with a running time of  $O(1)[1]$ .

## 2.3 Binary Search Trees

Binary search trees (BST's) are commonly used to represent items that have some inherent ordering such as sets or multisets.

The regular BST is defined as a binary tree where each node has a left and a right child along with a key and possibly an associated value. The left child of any node in the BST must have a key which is lesser than or equal to the node's own key, while the key of the right child must be greater than the node's own key.

### 2.3.1 Element retrieval

Since the nodes of the BST are ordered by their keys, we can search down through the tree starting at the root node to find the element with a given key. If the key we are searching for is lower than the current node, we recursively search the left subtree. If it is higher, we search the right subtree. Once the key of the current node matches the requested key, we can return the value of this node.

Since every recursive call goes a level down the tree, and all operations within the call take constant time, the running time of retrieval becomes  $O(h)$  where  $h$  denotes the height of the tree.

### 2.3.2 Element insertion

Elements in the BST are inserted in a recursive manner, most easily described by the following pseudocode:

```
function INSERTBST(node, element)
  if element.key ≤ node.key then
    if node.left = NIL then
      node.left ← element
    else
      InsertBST(node.left, element)
    end if
  else
    if node.right = NIL then
      node.right ← element
    else
      InsertBST(node.right, element)
    end if
  end if
end function
```

Exactly like for element retrieval, insertion requires searching through the height of the tree, making the time complexity  $O(h)$ .

### 2.3.3 Element removal

When removing an element from the BST, we find the element by searching like we did for retrieval. The node,  $x$ , containing the key can have zero, one, or two children, and we do slightly different operations dependent on this. If  $x$  has no children, we can safely remove it. If it has one child, this child can be made the child of  $x$ 's parent instead, effectively removing the node. If  $x$  has two children, find the node,  $y$ , that is the successor to  $x$ .  $y$  cannot have a left child since that would have to be the successor of  $x$  instead. Thus, we can make  $y$ 's child the direct child of its parent and insert  $y$  in  $x$ 's place instead. This total time for removal remains  $O(h)$ .

### 2.3.4 Balanced Binary Search Trees

The above description does not guarantee a very efficient data structure. All operations take  $O(h)$  time, but  $h$  is only upper bound by the number of elements



when insertion and removal are implemented as described. It is, however, possible to balance the BST while performing each operation as done in 2-3-4 trees and red-black trees. Balancing the tree limits the its tree to  $O(\log(n))$ , effectively ensuring a worst case running time for all three operations of  $O(\log(n))[1]$ .



# Tiered Vectors

---

The tiered vector is a state-of-the-art data structure introduced by Goodrich and Kloss in their paper “Tiered Vector: Efficient Dynamic Arrays for Rank-Based Sequences”[5]. In this they explain how to theoretically create the tiered vector data structure and achieve  $O(k)$  worst case time performance for retrieving elements, and  $O(n^{\frac{1}{k}}) = O(\sqrt[k]{n})$  amortised running time for element insertion and deletion for any fixed  $k \in \mathbb{N}$ . The generalised tiered vector can be thought of as a layered structure, in which  $k$  denotes its number of tiers. Goodrich and Kloss give a recursive definition of the data structure. This section will instead provide a detailed explanation of several ways to create the tiered vectors with one or two tiers. As it turns out, tiered vectors of arbitrary amounts of layers quickly become non-trivial to describe and implement.

## 3.1 The 1-Tiered Vector

The 1-tiered vector is a deque (double-ended queue). The deque is a queue data structure, and therefore it implements the *pop* and *push* operations. Aside from this, it also supports the operations *retrieve(index)*, *insert(index, element)*, and *remove(index)*. The deque is implemented in an array,  $a$ , with a length of  $m$ . The index of the head element,  $h$ , is maintained, as well as the number of elements,  $n$ .

### 3.1.1 Element retrieval

Retrieving an element at a specified index,  $i$ , is done by returning the element at index  $(h + i) \bmod m$  in  $a$ , clearly a constant time operation.

### 3.1.2 Element insertion

All insertions are followed by incrementing  $n$  by 1.

Element insertions at either end correspond to the *push* operations supported by queues. Elements are added at the front by decrementing the head once (that is  $h = (h + m - 1) \bmod m$  such that the array functions as a torrus for the deque) followed by inserting at index  $h$ . Elements are added at the end by simply inserting the element at index  $(h + n) \bmod m$ . Insertions at either end thus take  $O(1)$  time.

Inserting an element at an arbitrary position  $i$  requires existing elements to be moved. There are two cases:

1.  $i < \frac{n}{2}$ : We insert from the front. The head is decremented as described above, and elements from the head until position  $i$  are moved to the previous position in the deque.
2.  $i \geq \frac{n}{2}$ : We insert from the back. Elements from the end down to position  $i$  are moved to the next position in the deque.

In either case, the new element is inserted at index  $(h + i) \bmod m$ . This insertion amounts to  $O(\min\{i, n - i\})$  operations because we choose to insert from the end that requires fewest element to be moved.

A special case occurs when  $n = m$  such that the deque is full, and we wish to insert a new element. In this case, we use techniques similar to those of dynamic array. That is, if we wish to insert an element we first double the size of the deque. This is done by creating a new array of double capacity, which will be our new  $a$ , and inserting the elements from head to tail into the new array at their respective indices. Upon doing this, we double  $m$  and reset  $h$  to 0, followed by performing a regular insertion of the new element. The amortised running time analysis for this is essentially the same as for the corresponding dynamic array doubling technique in section 2.2.1. Expanding upon this, it is also possible to maintain a second deque of double the size alongside the current one in order to secure a worst case running time of  $O(\min\{i, n - i\})$  rather than letting this be the amortised cost.

An example of insertion with resizing can be seen on figure 3.1a.

### 3.1.3 Element removal

All removals are followed by decrementing  $n$  by 1.

Removals at either end correspond to the *pop* operations of a queue. Element removals are similar to insertions in that we can easily remove from the head by incrementing  $h$  by one. Removals from the end require nothing but the usual decrement of  $n$  since we do not mind what is stored in the array outside of the limits of the deque.

Elements removed from any other position  $i$  follow a similar pattern to insertion:

1.  $i < \frac{n}{2}$ : The elements from the head until position  $i - 1$  are moved to the next position in the deque, and then  $h$  is incremented.
2.  $i \geq \frac{n}{2}$ : Elements from the end down to position  $i + 1$  are moved to the previous position in the deque.

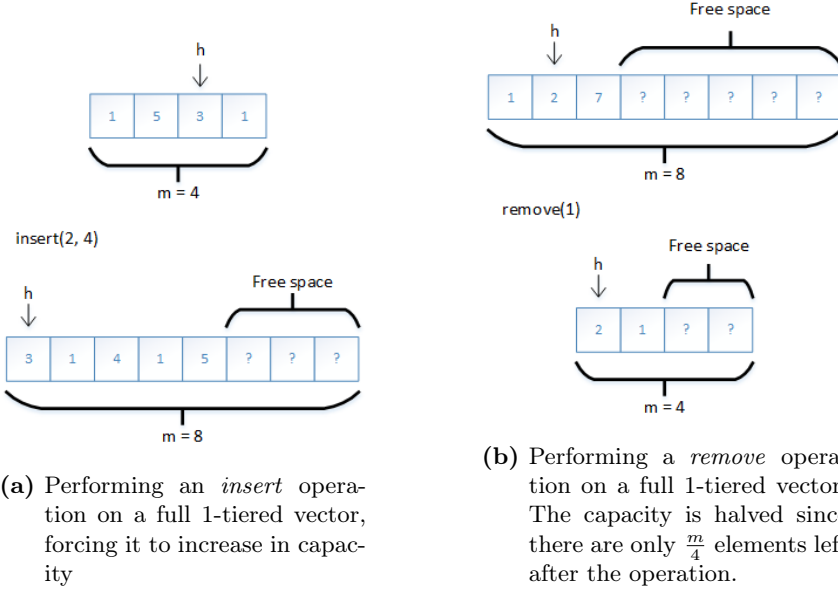
In either case, the element that we wished to remove is overwritten. Either operation requires  $O(\min\{i, n - i\})$  by the same reasoning as for insertion.

Just like for the dynamic array, we can also decide to decrease the size of the deque once there are too few elements left to justify the space consumption. Once only  $\frac{m}{4}$  of the deque contains elements, we create a new array of half the size and move the remaining elements into this instead.

An example of element removal with resizing can be seen on figure 3.1b.

## 3.2 The Simple 2-Tiered Vector

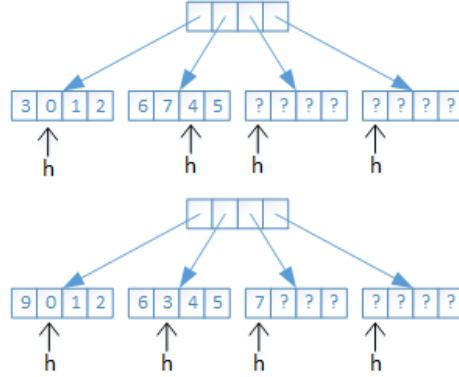
The initial non-trivial tiered vector is one for  $k = 2$ . The 2-tiered vector can be thought of as a tree with a height of 1 where the root has  $m$  children that each correspond to a 1-tiered vector. Each of the 1-tiered vectors have exactly capacity  $m$ . They are children in the tree, which are maintained by an array,  $a$ , in the root node. The actual elements of the data structure are contained within these children.



### 3.2.1 Element insertion

When we wish to insert an element at index  $i$ , we can calculate the index,  $i_c$ , of the child that currently contains the element at index  $i$  as  $i_c = \frac{i}{m}$ . If this child is full, we must *pop* its last element and *push* it onto the next child, which is found at index  $i_c + 1$  in  $a$ . Should this child also be full, there is once again no room, and we continue to pop its last element and push it onto the next child — and so on until we reach the first non-full child. Once this is done, there should finally be room for the new element in the child that we started at. We then recursively insert the element into the child at its index  $i' = i - i_c \cdot m$ .

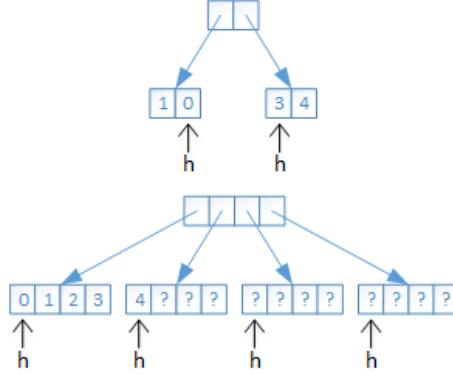
Recall that each *push* and *pop* operation in the 1-tiered vectors has a running time of  $O(1)$ . The total amount of *push* and *pop* operations is  $2 \cdot \frac{n-i}{m}$ .  $m$  is upper bound to  $O(\sqrt{n})$  (as we shall see later in the description of element removal in section 3.2.2), making the number of operations  $O(\sqrt{n} - \frac{i}{\sqrt{n}})$ . After that, inserting into a child with at most  $m = O(\sqrt{n})$  elements at index  $i' < \frac{\sqrt{n}}{2}$  takes  $O(\sqrt{n} + \frac{\sqrt{n}}{2}) = O(\sqrt{n})$  time, resulting in a total running time for insertion of  $O(\sqrt{n} - \frac{i}{\sqrt{n}} + \sqrt{n}) = O(\sqrt{n})$ . Also note that inserting elements at the end of the data structure is a constant time operation. An example insertion can be seen on figure 3.2.



**Figure 3.2:** Example of an insertion into the simple 2-tiered vector. 3 is inserted at index 3, causing 7 to be moved to the next 1-tiered vector, and  $h$  to be decreased once to make room.

**Increasing capacity** The initial value of  $m$  clearly creates an upper bound for  $n$  of  $m^2$  since there are exactly  $m$  children of the root node, each with a capacity of  $m$ . Therefore, we must expand the 2-tiered vector once  $n$  reaches  $m^2$  in order to accommodate for more elements on the next insertion. We do this by increasing  $m$  to  $m' = 2m$  and thus deliberately doubling the capacity of each of the children as well as the amount of children. Afterwards, each of the initial children are only half full, but we wish to ensure that they are filled from left to right. Starting at the second child, we then *pop* its first element and *push* it onto the end of the first child until the second child contains no more elements. At this point, the first child must also necessarily be filled, making the next children containing elements the third and fourth ones and so on in pairs. The remaining pairs can be combined in the same manner as described above followed by rearranging the children to move the filled ones to the front of the array. This ensures that  $a$  is only filled from the start. An example of one such operation can be seen on figure 3.3.

It takes  $m$  operations to move the contents of one entire child, and we end up moving the contents of  $m$  children (because all elements are moved to new underlying arrays in the 1-tiered vectors even if new 1-tiered vectors are not created) followed by rearranging  $\frac{m}{2}$  of them. This causes the increase of capacity to have a running time of  $m^2 + \frac{m}{2} = O(m^2) = O(n)$ . However, since it increases the capacity of the 2-tiered vector by a factor 4, we only increase the capacity every time at least  $O(\frac{3n}{4})$  elements have been inserted. Alternately, if we at some point decrease the size upon removal (see section 3.2.2), it will at that point only contain  $\frac{m^2}{2}$  elements such that at least  $\frac{n}{2}$  elements must be inserted before the next capacity increase. Since a regular insertion takes  $O(\sqrt{n})$  time,



**Figure 3.3:** Example of an insertion into the simple 2-tiered vector with no more available space. The element 2 is to be inserted at index 2. The capacities of each of the children are doubled as well as the amount of children. Every element is then moved to the beginning of the new 1-tiered vectors, and then 2 is inserted just like in the regular insertion.

we will have a total of  $O(\frac{n}{2} \cdot \sqrt{n} + n) = O(n^{1.5})$  operations between capacity increases, letting the average time for insertion be  $O(\frac{n^{1.5}}{n}) = O(n^{0.5}) = O(\sqrt{n})$ . This shows that we can have an amortised insertion time of  $O(\sqrt{n})$ , but there is a worst case performance of  $O(n)$  in specific, rare cases.

A different approach to increasing the capacity is simply iteratively *popping* the first element of each non-full child (aside from the first) and *pushing* them onto the first non-full child until all elements are in the first  $\frac{m}{4}$  children. This also takes  $O(n)$  time and is essentially the same process as the one described above, so we can still expect an amortised cost of insertion of  $O(\sqrt{n})$ .

### 3.2.2 Element removal

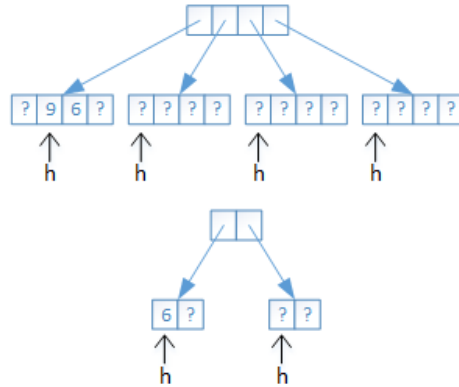
Removal at a given index  $i$  in the simple 2-tiered vector works just opposite of insertion. The element we wish to remove can be found in the child at index  $i_c$  (calculated exactly like in the above) in  $a$ , and we can also calculate its index within this child,  $i'$ , like we did for insertion. The element is removed from the child, leaving space for an extra element. If the child was full before the operation, there are possibly several non-empty children in the upper part of  $a$ . We iterate through the remaining children, starting with the one from which our element was removed. In each iteration, the first element of the next child is



*popped* and instead *pushed* onto the end of the current child – a process which continues until the next child is empty, or we have reached the final child of the root.

By an identical argument to the one in the above section 3.2.1, the amortised running time for element removal is  $O(\sqrt{n})$ , and it is possible to remove elements from the back in  $O(1)$  time.

**Decreasing capacity** Just as we have done for several of the other array data structures, we can decide to shrink the capacity of the 2-tiered vector to ensure a space complexity of  $O(n)$  and sustain the  $O(\sqrt{n})$  amortised time performance for insertion and removal. Once there are fewer than  $\frac{m^2}{8}$ , we will decrease  $m$  to  $m' = \frac{m}{2}$ , thus halving the capacity of each deque as well as halving the amount of deques. In order to perform the capacity decrease, each deque is split through the middle into two deques of size  $m'$ . The split deques are essentially already in order and therefore just need to be added in the same order into  $a$ . At the end of the operation, the 2-tiered vector will contain exactly  $\frac{m^2}{2}$  elements. As we argued for in the above section, we have now shown that no sequence of insertions and removals from the 2-tiered vector can cause the amortised running time of insertion or removal to exceed  $O(\sqrt{n})$ . A small example of element removal with a capacity decrease is displayed on figure 3.4.



**Figure 3.4:** Example of a removal from the simple 2-tiered vector that causes it to shrink. 9 is removed from index 0, causing  $n$  to fall below  $\frac{m^2}{8}$ , causing  $m$  to be reduced from 4 to 2.

### 3.2.3 Element retrieval

As we have already seen, it is very simple to find the element at index  $i$  by first finding the child containing it at index  $i_c = \frac{i}{m}$ . In this 1-tiered vector, the element is then located at index  $i' = i - i_c \cdot m$ . An alternate way of calculating  $i'$  is  $i \bmod m$ .<sup>1</sup>

## 3.3 2-Tiered Vector using a Deque

The deque 2-tiered vector acts much like the aforementioned simple 2-tiered vector. The main difference is that the children are not kept track of by using an array, but instead by a deque with some alterations. The deque is implemented in an array, but always contains exactly  $m$  children. However, the first element in the deque is defined as the first child that contains any elements. We will maintain a pointer to this child,  $h$ .

Using a deque for the 2-tiered vector may seem like it adds a lot of unnecessary overhead to a problem already solved by the simple 2-tiered vectors. It is, however, possible to construct tiered vectors of three tiers and more which require all the lower tiered vectors to have  $O(1)$  time element insertions and removals from either end. The deque 2-tiered vector thus lays the groundwork for the general higher tiered vectors mentioned at the beginning of this chapter.

### 3.3.1 Element retrieval

Because this data structure uses a deque to handle children rather than an array, all operations are complicated slightly compared to the previously described tiered vectors. They break down to the same essentials even so.

The index,  $i_c$ , of the child currently containing the element at index  $i$  is calculated as  $i_c = \lceil \frac{i+1-n_0}{m} \rceil$ . Here,  $n_0$  denotes the number of elements currently contained in the first child since it can be anything in the interval  $[1, m]$ . In this child, we retrieve the element at index  $i' = (i - n_0) \bmod m$ .

Retrieval in the 1-tiered vector has a  $O(1)$  time complexity, and the other calculations are also only  $O(1)$ , making the total running time for retrieval constant.

---

<sup>1</sup>When implemented, this can safely be assumed to be slightly slower because division in most CPU's is slower than multiplication.[4]

### 3.3.2 Element insertion

Firstly, the element currently at index  $i$  is found like in element retrieval. If  $i_c$  is 0, we insert the element at exactly index  $i$  in the first child. In any other case, we must insert the element at index  $i' = (i - n_0) \bmod m$  in the child we just found.

While we in the previously described 2-tiered vectors always *pop* and *push* elements from the found child towards the end of the array, the deque representation of the children allows us to move the elements towards either the front or the back. We wish to insert from the front if  $i < n - i$ , meaning that we have to perform fewest operations if we move the elements toward the front, much like in the 1-tiered vectors; otherwise, we insert from the back. The steps to these are described in the following.

**Inserting from the front** If the current child has no remaining capacity,  $h$  must be decremented. Recalling that the children are contained in a deque,  $h$  is decremented by  $h = (h - 1 + m) \bmod m$ . If we are not inserting our element into the first child, there are a few corrections to be made because the nature of moving elements towards the front displaces many elements influencing both  $i_c$  and  $i'$ . It is most easily described by the following piece of pseudocode:

```

if  $i_c > 0$  then
     $i' \leftarrow i' - 1$ 
    if  $i' < 0$  then
         $i_c \leftarrow i_c - 1$ 
        if  $i_c = 0$  then
             $i' \leftarrow \text{children}[h].\text{size}()$ 
        else
             $i' \leftarrow m - 1$ 
        end if
    end if
end if

```

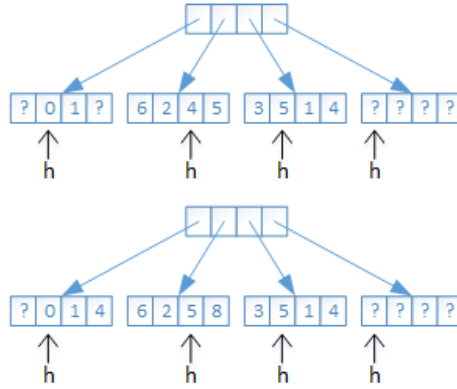
$i_c$  and  $i'$  now have the correct values for us to be able to simply *pop* the first element of each child from the front until the child at index  $i_c$  and *push* them onto the end of the preceding child.

**Inserting from the back** This is done by simply *popping* the last element of each child from the child at index  $i_c$  until the last child containing elements and *pushing* them onto the beginning of the subsequent child. If the last child

with elements is full, its last element is also moved to the next child, which must necessarily be empty.<sup>2</sup>

The total amount of *pop* and *push* operations is  $2 \cdot \min\{i_c, m - i_c\}$ . The time complexity for an insertion depends on this as well as the time to insert into the appropriate deque, which takes  $O(m - i')$  time. This amounts to  $O(\min\{i_c, m - i_c\} + m - i') = O(\min\{\frac{i}{\sqrt{n}}, \sqrt{n} - \frac{i}{\sqrt{n}}\} + \sqrt{n}) = O(\sqrt{n})$ , which turns out to be no different from the simple 2-tiered vector.

Figure 3.5 shows how one should think of the data structure and the manner in which it is possible to insert elements from the front.



**Figure 3.5:** Example of an insertion into the deque 2-tiered vector. 8 is to be inserted at index 4. Since  $n = 10$ , and  $4 < 10 - 4$ , the element is inserted from the front by *pop*ping 4 from the second child and *push*ing it to the end of the first. 5 is then moved, making room for 8 at the desired index.

**Increasing capacity** As for the simple 2-tiered vector,  $n$  is limited to  $m^2$  in the deque 2-tiered vector. We increase the size in an almost identical manner to how we did in the simple 2-tiered vector as well. That is, we reset  $h$  and double  $m$ , thus doubling the number of children as well as each of their capacities followed by moving every element into the new children, making sure they are filled from the left. However, this size increase occurs as soon as the closest child to the end we are inserting from has reached its capacity, and there are also elements in the child at the opposite end. This means that we possibly already increase the capacity when  $n$  equals  $m^2 - m + 1$ . Since this is still

<sup>2</sup>Otherwise, the capacity of the tiered vector would have been increased as described in the following paragraph.

$O(m^2)$ , we can use the same argument as for the simple 2-tiered vector to show that the running time can be amortised to  $O(\sqrt{n})$  on average for insertions.

### 3.3.3 Element removal

Performing removals can also be done from one end or the other, depending on the index from which we remove. Initially, the element at the index  $i$  is retrieved as described above. Whether *pops* and *pushes* are performed from the front or back of the tiered vector depends on whether  $i < n - i$ . When removing from the front, the last element of the first child is *popped* and *pushed* on to the next child and so on until the child that contained the requested element. If the first child no longer contains any elements, the pointer to the first child will be increased once. When removing from the back, elements are moved from the back toward the removal index exactly like in the simple 2-tiered vector.

The total amount of operations is analogous to the ones for insertion, and the running time analysis is identical, making the total time complexity for removal  $O(\sqrt{n})$ .

**Decreasing capacity** Like we did for the simple 2-tiered vector, the capacity of the deque 2-tiered vector must also be decreased once  $n$  goes below  $\frac{m^2}{8}$ , and this can ensure an amortised removal time of  $O(\sqrt{n})$  which was already explained in section 3.2.2.

## 3.4 Improving the Tiered Vectors with Bit Tricks

The three tiered vectors described in this chapter require several computationally slow calculations for each of the three operations. In the 2-tiered vectors, all three operations require the same calculation of  $i_c$  and  $i'$ . At the hardware level, these calculations presumably take a *relatively* long time because of the divisions and multiplication. The 1-tiered vector also requires modulo operations.

We can work around the comparatively slow performance of integer multiplication and division if we initially let  $m$  be a power of 2, e.g. 1, 2, or 4 and utilise this by performing simple bit tricks to presumably greatly increase performance. Alongside  $m$  we store a shift,  $s$ , which corresponds to the power of 2 that  $m$  currently is.

This results in us being able to simplify several calculations as shown in table 3.1:

Operation	Bit trick	Performed when
$i \bmod m$	$i \& (m - 1)$	Indexing in 1-tiered vectors
$m \cdot 2$	$m \ll 1$	Increasing capacity
$\frac{m}{2}$	$m \gg 1$	Decreasing capacity
$i_c \cdot m$	$i_c \ll s$	Calculating $i'$ in 2-tiered vectors
$\frac{m^2}{8}$	$m \ll s \gg 3$	Calculating whether to decrease capacity in 2-tiered vectors
$\frac{i}{m}$	$i \gg s$	Calculating $i_c$ in simple 2-tiered vectors

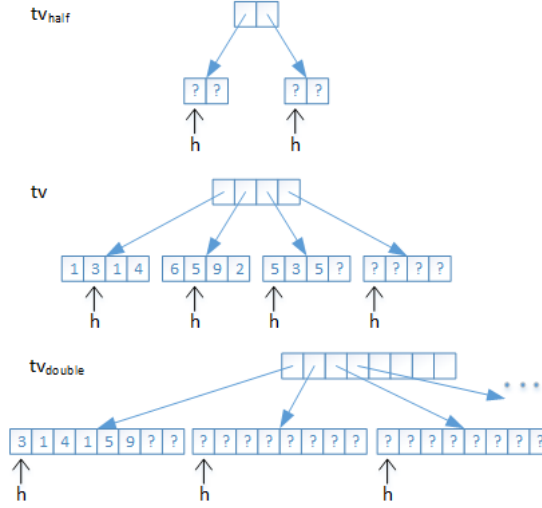
**Table 3.1:** Bit tricks

Obviously these bit tricks do not improve the asymptotic time complexity of the insertion, removal, and retrieval operations. Nonetheless, as will be shown in chapter 5, they prove to be significant performance boosts. This is, of course, somewhat expected, given the fact that multiplications and divisions are not trivial operations on most CPU's. Most modern CPU's are, however, highly optimised to perform operations such as bitshifts and bitwise ANDs[4]. The shifts are well-known ways of multiplying or dividing with powers of two. The modulus operation with any power of two can easily be recognised to work once you realise that the mask  $2^x - 1$  has a bit representation with exactly the least significant  $x$  bits toggled on.

Utilising these bit tricks on each of the data structures described in this chapter leaves us with three additional data structures to analyse in the following chapters.

### 3.5 Improving Worst Case Performance

So far, all the time complexities for insertion and removal of elements in the 2-tiered vectors have only been an amortised time of  $O(\sqrt{n})$ . This means that once the 2-tiered vectors must increase or decrease in size, the insertion and removal times for that single operation becomes  $O(n)$ . Thus the question becomes whether this can be combated if we want to guarantee  $O(\sqrt{n})$  insertion and removal. It turns out that it is possible to do by using a strategy similar to that of the dynamic array explained in section 2.2.1. The following will describe how to do this with the simple 2-tiered vectors. The description is of substantial length, so we can refer to figure 3.6 to visualise the idea.



**Figure 3.6:** Example of a simple 2-tiered vector with deamortised insertion and removal times. The bottom figure of  $tv_{double}$  consists of 5 not shown 1-tiered vectors as children, indicated by the loose arrow and the three dots. Some insertions have been performed since the latest size increase as indicated by the fact that  $tv_{double}$  contains some of the elements of  $tv$ .  $tv_{half}$  contains no elements since  $n > \frac{m^2}{4}$ .

### 3.5.1 Deamortising Insertion and Removal Costs

We maintain two additional arrays of 1-tiered vectors alongside  $a$ , essentially representing two additional 2-tiered vectors,  $tv_{double}$  and  $tv_{half}$ . The first of these,  $a_{double}$ , will initially have a capacity of  $m_{double} = 2m$  with each element being a pointer to a 1-tiered vector also of capacity  $m_{double}$ . Additionally, we will keep track of the total amount of elements in  $tv_{double}$  and call this  $n_{double}$ . We use similar variables for  $tv_{half}$ .  $a_{half}$  represents the array,  $n_{half}$  is the number of elements, and  $m_{half} = \frac{m}{2}$ . We will refer to the 2-tiered vector being operated on as  $tv$ .

**Element insertion** Our goal is that once  $n$  reaches the total capacity of  $m^2$  in the original 2-tiered vector, we do not have to move every element into  $2m$  new 1-tiered vectors of capacity  $2m$  each because this data structure is already at our disposal —  $tv_{double}$ . This allows us to simply replace  $a$  with  $a_{double}$ .

Let us begin with assuming that we have just increased the capacity of the 2-tiered vector by replacing  $a$  with  $a_{double}$ . During the capacity increase, the value of  $m$  is also doubled. Now we need a new  $a_{double}$ , so we allocate a new array of  $m_{double}$  1-tiered vectors, each with a capacity of  $m_{double}$ . This is essentially allocating  $2m$  arrays with some additional overhead, so the time complexity is  $O(2m) = O(2\sqrt{n}) = O(\sqrt{n})$ . Additionally, we reset  $n_{double}$  to 0. The only change for  $tv_{half}$  is that its array,  $a_{half}$ , will have its pointer replaced by one to the old  $a$ . This upholds the property of  $tv_{half}$  having a total capacity of  $m_{half}^2$  because  $\frac{m^2}{4} = (\frac{m}{2})^2$ . These operations only amount to a total running time of  $O(\sqrt{n})$  for the insertion operation that initiated the capacity increase.

Whenever a new element is inserted in  $tv$  at index  $i$ , we add it to the appropriate 1-tiered vector in  $tv_{double}$  at the correct index, provided that  $n_{double} > i$ . Both indices are calculated exactly like for  $tv$ , only using  $m_{double}$  instead of  $m$ . Aside from inserting the newest element into  $tv_{double}$ , two additional elements are inserted at the very end of  $tv_{double}$ . We know that inserting at the end of a 2-tiered vector can be done in  $O(1)$  time, so doing it twice remains a constant time operation. The element insertion at the arbitrary index  $i$  has a  $O(m_{double}) = O(\sqrt{n})$  running time. By inserting two elements at a time in  $tv_{double}$ , once the number of elements in  $tv$  has been doubled since the last increase in capacity, we will have added at least twice as many elements to  $tv_{double}$ . At this point,  $n$  is exactly  $\frac{m^2}{2}$ , so  $a_{double}$  will be available before the next increase in capacity at  $n = m^2$ .

If  $n > \frac{m^2}{4}$ ,  $n_{half}$  is reset to 0, and we do not insert anything into  $tv_{half}$ . As long as  $n \leq \frac{m^2}{4}$  and  $i < n_{half}$ , we insert every new element inserted into  $tv$  into  $tv_{half}$ .

Summarising, insertion requires two  $O(1)$  insertions and possibly one or two  $O(\sqrt{n})$  insertions in addition to the original insertion, achieving the desired  $O(\sqrt{n})$  total time complexity.

**Element removal** Deamortising the time complexity for removals is crucial since the general  $O(\sqrt{n})$  running times cannot be ensured if  $m$  is not scaled to  $O(\sqrt{n})$ . Whenever we remove an element at index  $i$ , the element must also be removed in  $tv_{half}$  if  $i < n_{half}$ . Moreover, every element removal must be followed by inserting two elements of  $tv$  at the end of  $tv_{half}$  as long as  $n_{half} < n$ . The total capacity of  $tv_{half}$  is  $\frac{1}{4}$  of the total of  $tv$ . Recall that we decrease the capacity and thus halve  $m$  when  $n = \frac{m^2}{8}$ . At this point we must be certain that  $tv_{half}$  contains the same amount of elements as  $tv$ . If  $n_{half} = 0$  when  $n = \frac{m^2}{4}$ , and we start inserting two elements for every removal, by the time we



have removed  $\frac{m^2}{8}$  elements, we will have inserted at least  $\frac{m^2}{4}$  elements in  $tv_{half}$ . Even though we could potentially have removed half of these elements in the following removal operations, this still leaves us with at least  $\frac{m^2}{8}$  elements. None of these operations change the running time of removal, since we only add an extra removal from  $tv_{half}$  ( $O(\sqrt{n})$ ) and two constant-time insertions at the end of  $tv_{half}$ .

Handling removal of the element is simpler in  $tv_{double}$ . The only requirement is that the element is removed in both  $tv$  and  $tv_{double}$ . The latter is only done if  $i < n_{double}$  since the element will not be there in the first place if that is not the case. This does not change our guarantee of  $tv$  and  $tv_{double}$  containing the exact same elements before the next increase in capacity because either 1 element is removed from both or nothing is removed from  $tv_{double}$ .

When the capacity of  $tv$  is decreased,  $a_{double}$  will point to the old  $a$  and  $tv_{double}$  will thus contain all the elements it should in an array of suitable capacity. Similarly,  $a$  will point to the old  $a_{half}$ , which will be reinitialised as  $m_{half}$  new 1-tiered vectors, each of capacity  $m_{half}$ . Remember that  $m$  has been decreased at this point.

This shows that removals, like insertions, can also be guaranteed a worst case running time of  $O(\sqrt{n})$ .

### 3.5.2 Generalising

The above description goes for the simple 2-tiered vector with or without bit tricks. It should definitely be possible to do something similar for the deque 2-tiered vectors, but I have refrained from further considering the details of doing this. The double-ended nature of the data structure could easily prove to be a big obstacle, greatly complicating each operation.



## CHAPTER 4

# Implementation

---

This thesis is accompanied by a piece of C++ code written to test the described data structures in practice. Everything is written in Visual Studio using the Visual C++ compiler and amounts to roughly 3 000 lines of code, not including any empty lines.

### 4.1 Data Structures

All data structures mentioned in chapter 2 have standard implementations in most programming languages including C++ with the Visual C++ compiler. The dynamic array is implemented in the `std::vector`, and the balanced binary search tree is implemented in `std::set`, although its exact implementation is not given[8][7].

The remaining data structures were implemented as subclasses of the following abstract class:

```
class ArrayDataStructure {
public:
    virtual uint32_t size() = 0;
```

```

const int32_t operator[](const int32_t index)
{ return getElemAt(index); }

virtual int32_t getElemAt(int32_t) = 0;
virtual void setElemAt(int32_t, int32_t) = 0;
virtual void insertElemAt(int32_t, int32_t) = 0;
virtual int32_t removeElemAt(int32_t) = 0;

virtual void insertLast(int32_t) = 0;
virtual int32_t removeLast() = 0;

virtual string toString() = 0;
virtual string toStringPretty() = 0;
};

```

Virtual function calls introduce some overhead[3], but using the abstract class allows for much easier testing. In an eventual complete implementation inheritance should not be used.

I decided to simplify everything by restricting the element type to 32-bit integers. This could easily be refactored to be generic, allowing for the creation of any type of elements. Several methods in the above are not strictly necessary either. The `toString` and `toStringPretty` method were purely meant for debugging purposes, but they are nice to have in a final implementation in any case. The `insertLast` and `removeLast` methods are also not necessary, but using them ended up providing much cleaner code as they are more legible than calling `insertElemAt(n, x)` or `removeElemAt(n - 1)`.

The specifics of each implementation are most easily inspected by looking over the code provided alongside this thesis. The following will only cover some implementation details.

**Deque and BitTrickDeque** These are the 1-tiered vectors. I decided to wrap the array in the C++ `std::vector` class and reallocating the needed capacity when the dequeues shrank or grew. The implementations are fairly straightforward. The `BitTrickDeque` cannot be initialised with a capacity that is not a power of 2, whereas the user can decide on any capacity in the regular `Deque`.

**Simple2TieredVector and BitTrickSimple2TieredVector** In both these implementations I used the `std::vector` to store the children. Whenever increasing the capacity of the tiered vector, I only added  $\frac{m}{2}$  new children of capac-

ity of the new  $m$  and called the `increaseCapacity()` method on the already existing 1-tiered vectors as can be seen by the following code stump:

```
int32_t oldM = m;
m *= 2;

vector<Deque> newChildren(m);
for (int32_t i = 0; i < oldM; i++) {
    newChildren[i] = children[i];
    newChildren[i].increaseCapacity();
}
for (int32_t i = oldM; i < m; i++) {
    newChildren[i] = Deque(m);
}
children = newChildren;
```

The `BitTrickSimple2TieredVector` of course uses the `BitTrickDeque` class rather than the `Deque` class as well as utilising several bit tricks within its own member methods.

**DeamortisedBitTrickSimple2TieredVector** I decided to only implement a deamortised version of the bit trick simple 2-tiered vector. This implementation is mainly designed for proof-of-concept of the deamortisation as it for the time being only implements  $tv_{double}$  from the description in section 3.5. Because my implementation wraps the array of 1-tiered vectors in the `std::vector` class, I decided to not reallocate  $2m$  new 1-tiered vectors after every capacity increase. Instead, I allocated them one at a time in the following insertions until they were all created. New and existing elements of the tiered vector were still inserted into  $tv_{double}$  as described in section 3.5. This made this method for increasing capacity much simpler as well, but introduces a new variable  $missing_{double}$ , denoting how many of the new children that have been initialised:

```
void increaseCapacity() {
    int32_t oldM = m;
    m = m << 1;
    children.swap(children_double);
    n_double = 0;
    missing_double = 0;
    shift++;
}
```

**Deque2TieredVector** and **BitTrickDeque2TieredVector** These classes are implemented much like the simple 2-tiered vectors. The key difference is the way in which the children are stored. Given that they are represented in a sort of deque, this of course adds some extra overhead. To reduce some redundancy of calculating the index in the array at which a given child is located, I added the short `child` method:

```
int32_t child(int32_t i) {  
    return (h + i) % m;  
}
```

Again, the bit trick version doesn't perform a modulo operation, but uses a bitwise AND instead.

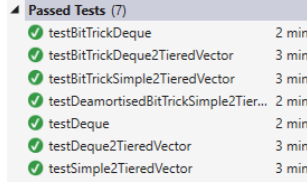
## 4.2 Unit Testing

I have created unit tests with Visual Studio's native C++ unit testing framework to test retrieval, insertion, and removal in each of the tiered vector data structures described in chapter 3. All data structures are initialised with their default constructors, meaning that their capacities start at  $4^k$ . Randomisation in the unit tests is achieved using the `std::linear_congruential_engine`, which is simply a common pseudorandom number generator and fine for this application[2][6].

1 000 random elements are created. These are the elements to be inserted into each data structure. A random order of indices to insert into is then generated, always ensuring that the next index is in the interval  $[0, n]$ . Using `std::vector`, the expected order of elements after each insertion is stored. When inserting into each data structure, the current order of elements is verified with the expected order. Once all 1 000 elements have been inserted, their order is verified one last time before they are removed one at a time in a new random order, and the element order is confirmed after each removal.

I decided to unit test using randomised black box tests and verifying the contents of the data structures after each operation because each operation has several edge cases, which I might accidentally miss if testing in a white box manner. Using a high degree of randomness (i.e. several seeds and 32-bit numbers) and many operations, all edge cases are highly likely to be reached.

The results of the tests are displayed in figure 4.1.



Passed Tests (7)	
testBitTrickDeque	2 min
testBitTrickDeque2TieredVector	3 min
testBitTrickSimple2TieredVector	3 min
testDeamortisedBitTrickSimple2Tier...	2 min
testDeque	2 min
testDeque2TieredVector	3 min
testSimple2TieredVector	3 min

Figure 4.1: All tests passed in unit testing

### 4.3 Benchmarks

In order to perform a detailed analysis of each data structure, I created a small benchmarking framework for each of the retrieval, insertion, and removal operations. I used the same linear congruential generator for randomness as in section 4.2. All times were measured using wall time rather than CPU time because of the increased accuracy of wall time measurements. This can cause fluctuations at random because the operating system does not guarantee that the thread is run continuously. These fluctuations were minimised by shutting down as many processes as possible while running the tests.

I performed each test with the same maximum data structure size of 500 000 in intervals of 500, resulting in each operation being tested for 1 000 different sizes for each data structure. For retrieval I benchmarked the time it took to retrieve 100 000 elements at each interval. The retrieval tests were run twice, one regular run and once without any optimisation because array accesses were likely being optimised out since the values were not used for anything. Insertion was tested by repeatedly timing the insertion of 500 elements at random indices until the data structure reached a size of 500 000. For removal, 500 elements were removed at a time until the data structure became empty.

I did an additional test of the `BitTrickSimple2TieredVector` against the `DeamortisedBitTrickSimple2TieredVector`, in which I only tested insertion. I only looked at the interesting data points, which are just when tiered vectors must increase their capacities and the following insertions. More precisely, I fetched data for index  $i$  if  $\lfloor \log_2(i) \rfloor = 2^k \wedge i \leq 2^{\lfloor \log_2(i) \rfloor} + 2^{\lfloor \log_2(\sqrt{i}) \rfloor}$  for some integer  $k$ .

The results of the benchmarks are shown in section 5.2.





# Analysis

---

In the following, we will be examining the theoretical time complexities of each of the described data structures and comparing them. However, based on the implementations and tests presented in chapter 4, we will also be looking at the actual running times of each of the retrieval, insertion, and deletion operations.

## 5.1 Asymptotic Time Complexities

Table 5.1 outlines the time complexity of each of the data structures described in chapters 2 and 3.

The takeaway from this table is that the BST should have far superior performance for insertion and removal, followed by all the 2-tiered vectors. In the rare cases described previously, the worst case performance for 2-tiered vectors is linear unless they are deamortised. Everything but the BST have constant time element retrieval.

Data structure	Retrieval	Average insertion/removal	Worst case insertion/removal
Array	$O(1)$	$O(n)$	$O(n)$
Dynamic array	$O(1)$	$O(n)$	$O(n)$
Balanced binary search tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$
Deque	$O(1)$	$O(n)$	$O(n)$
Bit trick deque	$O(1)$	$O(n)$	$O(n)$
Simple 2-tiered vector	$O(1)$	$O(\sqrt{n})$	$O(n)$
Bit trick simple 2-tiered vector	$O(1)$	$O(\sqrt{n})$	$O(n)$
Deque 2-tiered vector	$O(1)$	$O(\sqrt{n})$	$O(n)$
Bit trick deque 2-tiered vector	$O(1)$	$O(\sqrt{n})$	$O(n)$
Deamortised simple 2-tiered vector	$O(1)$	$O(\sqrt{n})$	$O(\sqrt{n})$

**Table 5.1:** Asymptotic time complexities for the operations on different data structures.

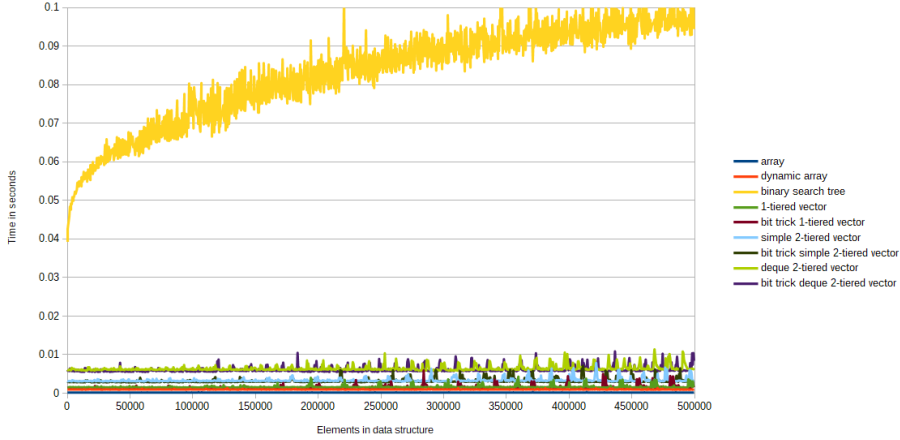
## 5.2 Analysis of Benchmarks

I have tested of every one of my implemented data structures against the standard array implementations in the C++ `std` library mentioned in chapter 4. Everything was tested on a computer running 64-bit Windows 10 with an Intel®Core™i7-4720HQ 2.60 GHz CPU and 8 GB RAM. Results are subject to differ slightly on different CPU architectures.

In the following I will provide a detailed analysis of how the data structures in my implementations compare to the standard implementations as well as to each other. The results are obviously limited by the efficiency of my implementations.

### 5.2.1 Element retrieval

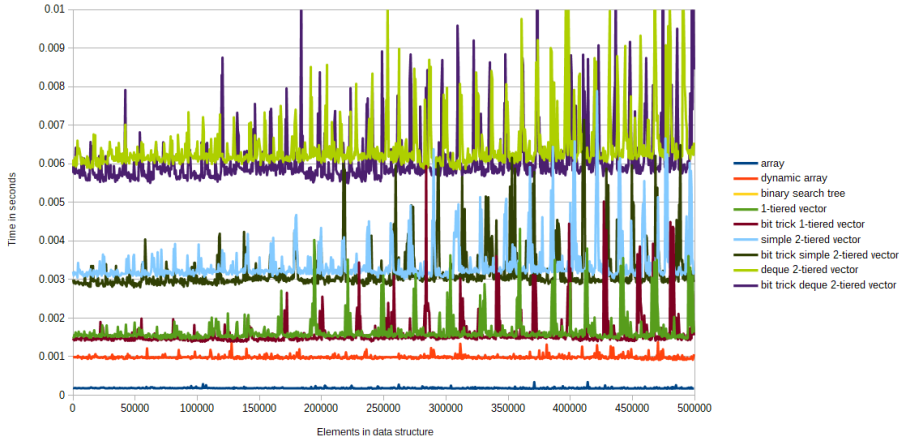
As mentioned in section 4.3 element retrieval benchmarks were run both with and without code optimisation. Let us first take a look at the results without any optimisation enabled. They can be seen on figure 5.1.



**Figure 5.1:** Time to retrieve 100 000 randomly selected elements in different data structures

Quite evidently and to no surprise from our knowledge of the data structures, the binary search tree performs much worse than the others due to its  $O(\log(n))$  time complexity while the others have constant time performance for element retrieval. We can also see that the 2-tiered vector implemented using a deque both with and without bit tricks performs significantly worse than the array, dynamic array, and other tiered vectors. In order to distinguish the remaining elements, we take a more detailed look as depicted on figure 5.2 as well as their average times in table 5.2.

There are seemingly very large fluctuations in the running times for every data structure aside from the regular array. The `std::vector` is also not as influenced by these as the others. The sizes of the fluctuations seems to correlate somewhat with the number of elements in the data structures. This could suggest that they are the result of cache misses since the amount of misses should increase when we actively use greater amounts of memory. It is also possible that they are the result of increasingly computationally difficult operations for the data structures that do not utilise any bit tricks. Seeing as the bit trick implementations display the same exact pattern, the fluctuations are probably not a direct result of divisions or multiplications.



**Figure 5.2:** Detailed view of retrieving 100 000 elements in all the data structures aside from the binary search tree

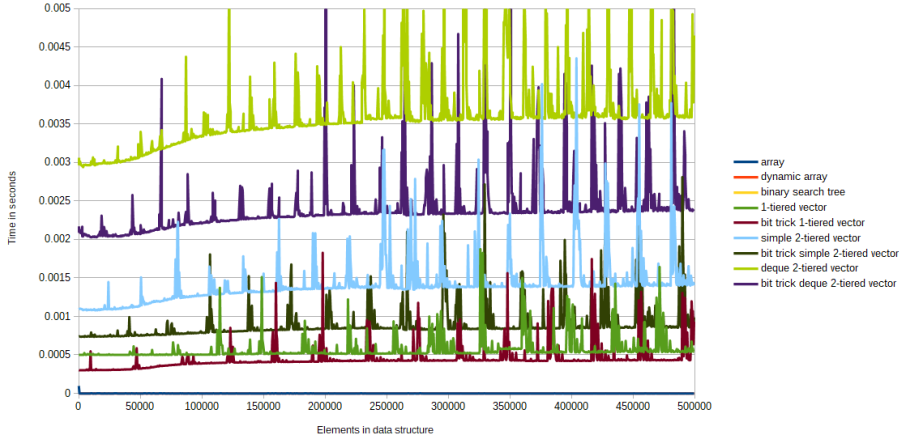
Data structure	Average time for retrieval
Array	0.19 ms
Vector	0.99 ms
1-tiered vector	1.65 ms
Bit trick 1-tiered vector	1.59 ms
Simple 2-tiered vector	3.36 ms
Bit trick simple 2-tiered vector	3.16 ms
Deque 2-tiered vector	6.44 ms
Bit trick deque 2-tiered vector	6.09 ms

**Table 5.2:** Average times for retrieval of 100 000 elements

The regular array is remarkably faster and more consistent than any of the other data structures, and the `std::vector` is also a fair bit faster than the tiered vectors, at least by a factor 1.59 compared to the best 1-tiered vector and 3.19 compared to the best 2-tiered vector. Aside from that, we can clearly see the similar tiered vectors grouping together with the implementations that utilise bit tricks only being ever so slightly faster.

Despite some of the data structures being less efficient, it is still worth noting that all their constants are quite low as there is no more than a factor 6.5 difference between the dynamic array and the deque 2-tiered vector, and even 6.47 milliseconds to retrieve 100 000 elements is reasonably fast.

Now let us analyse how these benchmarks look when all code optimisation is enabled instead. I will only provide the detailed view since the `std::set` is rather uninteresting. The results can be seen on figure 5.3.



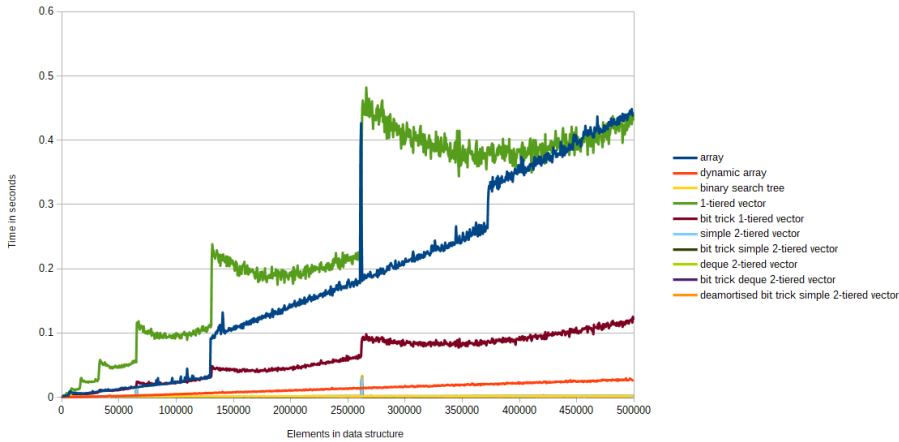
**Figure 5.3:** Detailed view of retrieving 100 000 elements in all the data structures aside from the binary search tree with code optimisation enabled

In this figure it becomes quite evident that retrieval from the array as well as the `std::vector` are completely optimised out. Therefore we cannot reliably analyse their data. We do, however, notice that the compiler was able to generally optimise the data structures since the retrieval times are notably lower than without optimisation. Especially the bit trick data structures have had notable performance boosts, and we can no longer see the clear distinction between the three main types of tiered vectors. The bit trick simple 2-tiered vector is particularly impressive as its running time is almost on par with the 1-tiered vectors’.

### 5.2.2 Element insertion

The results of the insertion tests are shown on figure 5.4.

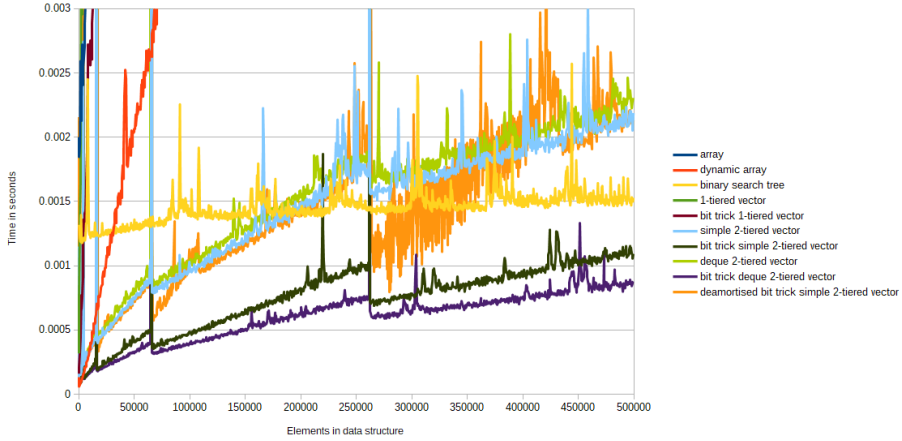
The initial graph is somewhat dull as we can only make out the slower data structures. We do, however, clearly see the efficiency of the dynamic array as well as how much the bit tricks affect the 1-tiered vector. It is also easy to make out when the L2 and L3 caches cannot be used to store the array anymore. A more detailed view can be seen on figure 5.5.



**Figure 5.4:** Time to insert 500 elements at random positions in different data structures

We know that the BST should be much more efficient than any of the other data structures with its insertion time of  $O(\log(n))$ . Surprisingly, even the `std::vector` is more efficient than the `std::set` until they contain around 32 000 elements although it has linear time insertion. Until more than roughly 100 000 elements have been inserted, all the 2-tiered vectors also remain more efficient. This shows that for most intents they are likely to be preferred to using BST's. It should be noted that both bit trick implementations are significantly better than their regular counterparts while the deque 2-tiered vectors also display slightly better performance.

Despite the variations on the graphs, we can still notice the somewhat jagged nature of the graphs for both the 1- and 2-tiered vectors. The sudden dips in running times are when the data structures increase in size, which is very noticeable at sizes 16 384 ( $= 2^{14}$ ), 65 536 ( $= 2^{16}$ ), and 262 144 ( $= 2^{18}$ ). The graph for the only deamortised 2-tiered vector is fairly inconsistent compared to the others, but its speed is very similar to those of the other 2-tiered vectors, suggesting that the extra operations do not add too much overhead. Its shape has the same jaggedness of the others, but as we shall see in section 5.2.4 there is no single operation running (comparatively) slowly.



**Figure 5.5:** Detailed view of inserting 500 elements at random positions in different data structures

### 5.2.3 Element removal

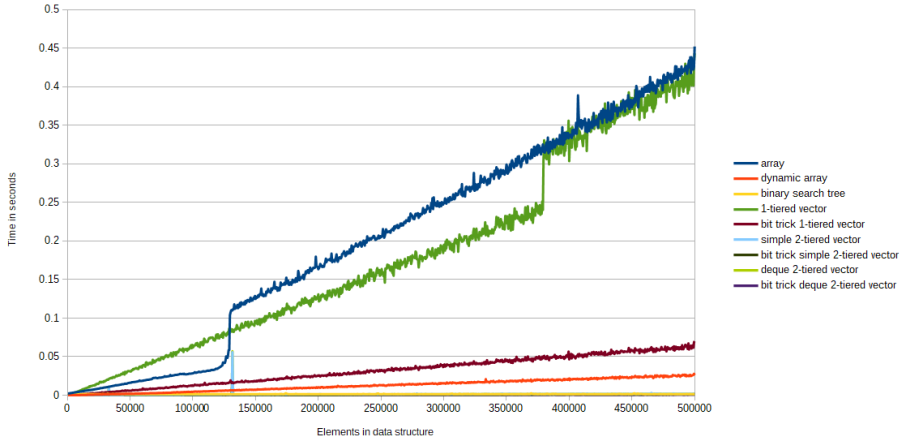
The performance of the different data structures for removal seen on figure 5.6 shows that, much like for insertion, the single tier structures perform significantly worse than the 2-tiered vectors and the binary search tree. There is a quite notable and sudden jump in performance for the array at a size of around 130 000 and later for the 1-tiered vector. These are probably the points at which the cache is no longer large enough to store the arrays. The interesting part of the data can be seen on figure 5.7.

This data shows that removals in the 2-tiered vectors are more efficient than in the BST until there are at least several hundred thousand elements. It is also worth a mention that the deque 2-tiered vectors are better for removal than insertion compared to the simple 2-tiered vectors. However, the bit trick deque 2-tiered vector is clearly far more efficient than any of the other data structures.

### 5.2.4 Deamortisation

I pitted the bit trick simple 2-tiered vector against the prototype implementation of its deamortised version. The results can be seen on figure 5.8.

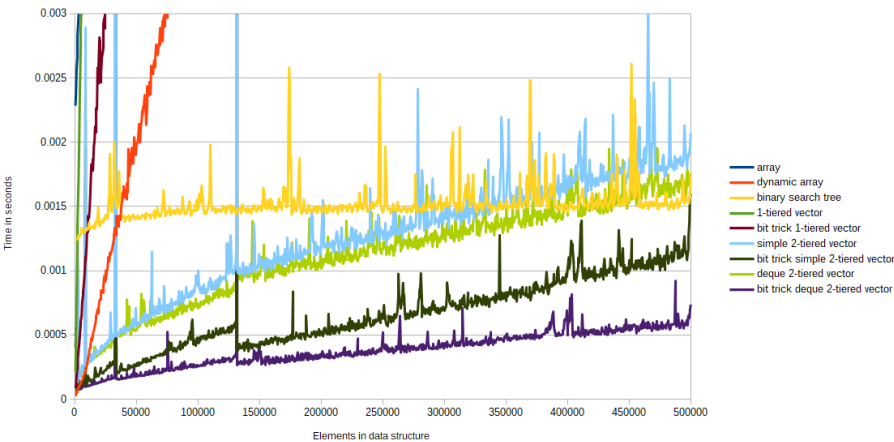
The figure shows each that inserting a new element whenever  $n = 2^x$  for even  $x$



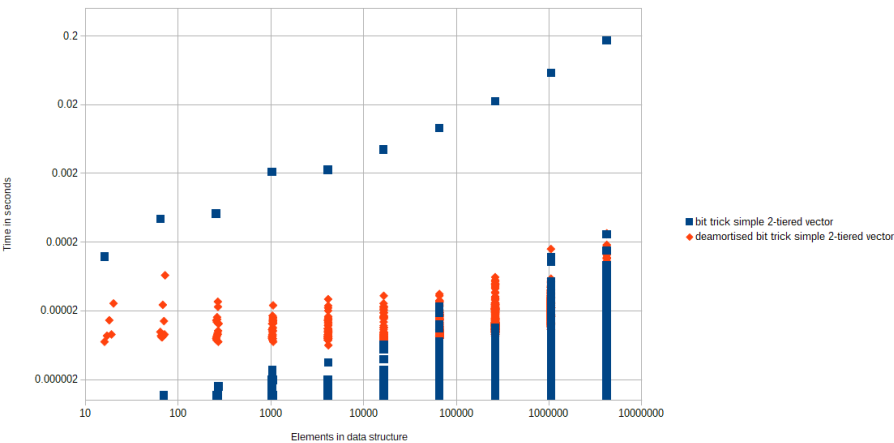
**Figure 5.6:** Time to remove 500 elements from random positions in different data structures

takes linear time in the regular implementation, whereas the cost is spread out amongst the next  $\sqrt{n}$  insertions for the deamortised version. This verifies our theoretical analysis, and as we can see at around four million ( $2^{22}$ ) elements, the time difference for that single insertion from the rest is a around a factor 1000, which is quite significant. With more elements being inserted, the time used to spread out the capacity increase becomes less significant.





**Figure 5.7:** Detailed view of removing 500 elements from random positions in different data structures



**Figure 5.8:** Time to insert a single elements at a random position in the regular and deamortised versions of the bit trick simple 2-tiered vector. Data is only present for the chokepoints. That is, when the tiered vectors must increase in size. Double-logarithmic scale.



# Conclusion

---

This thesis has been about describing, analysing, and testing existing and state-of-the-art data structures for arrays. More precisely, the challenge has been to find data structures that can represent ordered, indexable dynamic sets of elements and handle element insertion, removal, and retrieval efficiently. The array data structure can thus be thought of as an abstract data type supporting the mentioned operations. The currently used data structures are primarily the regular array and the dynamic array. These allow for efficient  $O(1)$  running times for retrieving elements, but insertions and removals require linear time in the amount of elements currently in the data structure. Binary search trees have also been assessed, although their application requires the elements to have some inherent ordering. They provide a slightly different trade-off for the three operations, namely  $O(\log(n))$  time complexity for each of them if implemented as balanced binary search trees. Tiered vectors were introduced as a new means of performing the requested operations. The report has shown how 1-tiered vectors have the same asymptotic time complexities as arrays and dynamic arrays and in fact are similar to dynamic arrays in some senses. They are a stepping stone to the 2-tiered vectors, which have been the primary focus. A 2-tiered vector is a layered structure, in which the upper layer has pointers to a number of 1-tiered vectors. The layers allow us to perform simple calculations in order to determine in which 1-tiered vector a given element belongs, and where it should be put within that 1-tiered vector. The sequence of 1-tiered vectors can be represented either in a regular array or a deque-like structure. By using 2-tiered vectors

it becomes possible to guarantee constant time element retrieval and achieve  $O(\sqrt{n})$  amortised time element insertion and removal. Furthermore, by cleverly choosing the capacity of the 2-tiered vectors one can utilise several bitwise operations that can seriously speed up execution time for all three operations. Finally it is shown how it becomes possible to deamortise the  $O(\sqrt{n})$  cost for insertion and removal by using a constant factor extra space, thus guaranteeing  $O(\sqrt{n})$  time complexity while maintaining constant time element retrieval.

All the described data structures have been implemented in C++ except for the array, dynamic array, and binary search tree, all of which have standard implementations. This has provided a grand total of 10 different data structures to compare. Taking a look at the performance for element retrieval, the benchmarks in this thesis show that the logarithmic time complexity of the binary search trees are by far inferior to all the other data structures. Regular arrays are also notably faster than the remaining constant time array data structures. Interestingly, all the tiered vector data structures had retrieval times similar to those of the standard dynamic array implementation. The 1-tiered vectors were slightly faster while the 2-tiered vectors were slower only by a factor 1.6 to 6.5 on average. The 2-tiered vectors greatly outperformed all the other data structures on benchmarks for insertion and removal. The one that has been called the deque 2-tiered vector was able to perform both operations faster than any other data structure when implemented using bit tricks, including the binary search tree when both were filled with upwards of a million elements. This is despite the binary search tree having a much better asymptotic time complexity. On the other hand, the simple 2-tiered vector, as it has been called, had a slightly worse performance for insertion and removal while being slightly faster for element retrieval instead. The deamortised 2-tiered vector was also tested. When run against its regular counterpart, it was clearly shown that there were no choke-points such that no single operation was slow despite the data structure in general being slower by a small constant factor.

Summarising, this thesis has shown how several different ways of implementing array data structures can challenge the current conventions for this abstract data type. In general it can be said that 2-tiered vectors should be used as they allow for  $O(1)$  time element retrieval as well as  $O(\sqrt{n})$  insertion and removals. The exact implementation can depend on the needs of the user.

Higher-tiered vectors have been described by others[5], and further work along the lines of this thesis should examine how they fare compared to the ones that have been studied here. This also opens the question as to whether the insertion and removal operations can be deamortised to a worst case time performance of  $O(\sqrt[k]{n})$  in any  $k$ -tiered vector. It is also possible that there are entirely different ways to rethink the abstract type data explored in this thesis, or that some of what has been described could be improved.

## APPENDIX A

# Project Plan

---

The project plan can be seen below and remains the same as the original.

<b>Title:</b>	Data Structures for Arrays
<b>Type:</b>	BSc
<b>Student:</b>	s144439 Thomas Søren Henney
<b>Advisors:</b>	Inge Li Gørtz and Philip Bille
<b>ETCS:</b>	20
<b>Evaluation:</b>	7 point scale
<b>Period:</b>	13.02.2017 - 16.06.2017

### A.1 Description

The main topic of this project is tiered vectors.<sup>1</sup> The goal is to survey and implement selected state-of-the-art data structures for arrays with the goal of optimising indexing, insertion, and deletion. Based on the experiences, the aim is to design and develop new solutions.

---

<sup>1</sup>Goodrich, MT and Kloss, JG; Tiered vectors: Efficient dynamic arrays for rank-based sequences; Lecture Notes in Computer Science; <http://www.ics.uci.edu/~goodrich/pubs/wads99.pdf>; Published: 1999

## A.2 Teaching Goals

- Survey existing data structures for arrays.
- Implement and compare common and selected state-of-the art data structures for list indexing, insertion, and deletion, such as arrays, dynamic arrays, skip lists, and tiered vectors.
- Design new data structures and algorithms based on experiences with existing state-of-the-art data structures and algorithms.
- Implement a prototype of the new data structures and algorithms.
- Analyze and evaluate the efficiency of the solutions from a theoretical and practical perspective.
- Document key relevant aspects of the work in a concise manner.

## A.3 Self-evaluation of the Project Process

The contents of this thesis reflect very well on the original project plan, and there has been no need to alter it.<sup>2</sup> Initially, I had a slow start with implementing the examined data structures. Part of this was caused by my somewhat lacking experience with C++, resulting in several bumps on the road with memory management and scaling the project with addition of new data structures, unit tests, and benchmarks. Once I had passed these initial hurdles, the structure of the code became more manageable. I would have liked to describe and implement the general k-tiered vector described by Goodrich and Kloss[5], but my full comprehension of how it worked came at a too late stage. At this point I decided to limit myself and instead independently investigate new data structures.

---

<sup>2</sup>Minor words have been changed without any actual influence on meaning.

# Bibliography

---

- [1] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to algorithms*. eng. 2nd ed. MIT Press, 2001, 1180 s. ISBN: 0070131511, 0262032937.
- [2] cppreference.com. *std::linear\_congruential\_engine*. 2016. URL: [http://en.cppreference.com/w/cpp/numeric/random/linear\\_congruential\\_engine](http://en.cppreference.com/w/cpp/numeric/random/linear_congruential_engine) (visited on 06/15/2017).
- [3] Karel Driesen and Urs Hölzle. “The direct cost of virtual function calls in C++”. eng. In: *Sigplan Notices (acm Special Interest Group on Programming Languages)* 31.10 (1996). <http://www.cs.ucsb.edu/~urs/oocsb/papers/oopsla96.pdf>, pp. 306–323. ISSN: 15581160, 03621340. DOI: 10.1145/236338.236369.
- [4] Agner Fog. *Instruction tables*. 2017. URL: [http://www.agner.org/optimize/instruction\\_tables.pdf](http://www.agner.org/optimize/instruction_tables.pdf) (visited on 06/15/2017).
- [5] MT Goodrich and JG Kloss. “Tiered vectors: Efficient dynamic arrays for rank-based sequences”. eng. In: *Lecture Notes in Computer Science* 1663 (1999), pp. 205–216. ISSN: 16113349, 03029743.
- [6] Stephen K. Park, Keith W. Miller, and Paul K. Stockmeyer. “Technical Correspondence”. <http://www.firstpr.com.au/dsp/rand31/p105-crawford.pdf>.
- [7] Colin Robertson et al. *set Class*. 2017. URL: <https://docs.microsoft.com/da-dk/cpp/standard-library/set-class> (visited on 06/07/2017).
- [8] Colin Robertson et al. *vector Class*. 2017. URL: <https://docs.microsoft.com/da-dk/cpp/standard-library/vector-class> (visited on 06/07/2017).