DTU Compute
Department of Applied Mathematics and Computer Science

Optimal Resizable Arrays

Master Thesis

Christian Rosenkilde Husted Kjær (s183943)
Victor Brevig (s184469)

Approval

Christian Rosenkilde Husted Kjær - s183943

.....

Signature

.....

Date

Victor Brevig - s184469

.....

Signature

.....

Date

Abstract

The resizable array, or dynamic array, is a data structure commonly used in computer programs. In this thesis, different solutions to the dynamic array problem are implemented, empirically tested and compared with regard to access time, insertion/deletion time and excess memory usage. Among the implemented solutions are the family of dynamic arrays presented by Tarjan and Zwick in their paper from 2022 [1], which we are the first to implement. This thesis includes additional implementation details, such as rebuilding the data structure that was not provided by Tarjan and Zwick. Based on the findings of the tests performed, several new solutions are presented. First, we present the Fast Combine data structure, which focuses on fast insertion and deletion times. Second, the Hybrid Sitarski data structure is presented, based on Sitarski's HAT data structure [2]. We show that this data structure is at least as good as HAT regarding excess memory, except when rebuilding the index block, which can at most happen every $\Omega(N)$ insertion or deletion. Our tests show that most of the time, Hybrid Sitarski is superior in terms of excess memory while being on par with HAT on access and insertion/deletion times. The Hybrid Sitarski data structure is also generalized to a $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$ data structure based on Tarjan and Zwick's $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$ family of data structure. This solution significantly lowers the excess memory usage while maintaining the same asymptotic bounds, although to a lesser degree for larger values of r.

Contents

1 Introduction

The array is one of the oldest and most widely used data structures. It is used by almost every program and is the foundation of many other data structures [3]. An array is a data structure which stores a sequence of elements at adjacent memory locations. Given an index in the array, one can get the element in that position in constant time using the random access property, i.e. the element indices can be computed at run time [3]. For that reason, the elements of the array are required to be of the same size and the same data representation. If the data to be stored is of fixed size, the array is highly memory efficient. However, if more data needs to be stored later, there is no guarantee that the array can be extended, as the subsequent memory locations might be occupied by something else. A resizable array, also known as a dynamic array, is another beneficial data structure. A resizable array, with a size of N, is a type of data structure that functions similarly to a standard array, but with the added ability to expand or contract its size. This means that it can add an element at the end, as well as remove the last element.

Common examples of dynamic array implementations are C++'s *Vector* and Java's *ArrayList* [4] [5]. In most programming languages, a standard library implementation of a dynamic array is included and is commonly implemented by the principle of geometric expansion. When the array is full, it is rebuilt to an array of the same size multiplied by some constant factor, for example, 2. This has a large amount of memory overhead of $\mathcal{O}(N)$ just to store the dynamic array with N elements. Several solutions have been proposed over the last decades, most of which have a common structure of splitting the array into smaller data blocks accessed through an index array of pointers. Even though most of these solutions use $\mathcal{O}(\sqrt{N})$ excess space to store the array, they have failed to gain broad adoption [6]. Ideally, a dynamic array data structure should minimize excess memory and support fast access and insertion times. However, these three parameters exist in a trade-off space, making choosing one implementation as the best non-obvious. This report aims to test various solutions to the dynamic array problem, even a few solutions that have not been implemented before and compare them according to this trade-off spectrum. New solutions are constructed and compared based on the test results and knowledge of the existing solutions.

The rest of the report is structured as follows: in section 2, the resizable array problem and the computational model considered in this report are covered. In section 3, previous work of the resizable array problem is described. Implementation details of the first implementation (to the best of our knowledge) of Tarjan and Zwick's [1] solution to the resizable array problem are covered in section 4. In section 5, empirical tests for excess memory, insertion time and access time are described, and the results are shown for all the solutions described in section 3. In section 6, new solutions to the resizable array problem and extensions to existing solutions are presented. The new solutions are also tested and compared to the previous solutions to the resizable array problem. Section 7 discusses the report results, our considerations and potential future work. Finally, the report is concluded in section 8.

2 Resizable array problem

Throughout this thesis we consider a dynamic array data structure supporting the following operations:

- $A \leftarrow Array()$ - Create and return an initially empty array.
- $A.Length()$ - Return the length of the array.
- $A.Get(i)$ - Returns the element at index i.
- $A.Set(i, x)$ - Set the element at index i to x.
- $A.Grow(x)$ - Increase the length of the array by one and insert the element x at the last position.
- $A.Shrink()$ - Decrease the length of the array by one by deleting the last element.

In particular, we will investigate how to implement a data structure supporting these operations using the least amount of additional memory as possible, while keeping fast *Get*, *Set*, *Grow* and *Shrink* operation run times both asymptotically and in practice.

We assume that the memory management of the system allows us to allocate and deallocate memory as fixed array chunks. Furthermore, it is assumed that allocation of memory takes constant time as is assumed in [1]. Even if it takes $\mathcal{O}(N)$ time to allocate and deallocate an array of size N the amortized bounds of the solutions explored and new solutions presented in this thesis will remain.

3 Previous Work

This section briefly describes existing solutions to the dynamic array problem. The bounds for the different data structures are compared and summarized at the end of the section.

3.1 Naive Data Structure

The most straightforward data structure for implementing resizable arrays uses only a single array of fixed size. For *Grow* operations, when the array is full, the data structure needs to allocate a larger array, copy over all the items from the old array to the new larger array and then deallocate the old array. Following a *Shrink* operation, the data structure may decide that the array is too empty and thus allocate a new smaller array, copy over all the items from the old array to the new one, and then deallocate the old array.

The simplest and most space-efficient data structure for resizable arrays will allocate a new array and copy over all elements for each *Grow* and *Shrink* operation. Thus, maintaining a fixed-sized array of the same size as the number of elements, i.e. its space complexity for storing the array is N. The cost of *Grow* and *Shrink* operations, however, are $\Theta(N)$. In addition, temporarily during any *Grow* or *Shrink* operation, $2N + \mathcal{O}(1)$ space is needed since two arrays are stored while copying takes place. Thus, this data structure should only be used if *Grow* or *Shrink* operations are extremely rare.

3.2 Geometric Expansion

A more commonly used data structure for the resizable array problem uses the geometric expansion/shrinking technique. It also uses a single fixed-sized array and a chosen parameter, $\alpha > 1$. For *Grow* operations, once the allocated array is filled, a new array is allocated of size αN, where all elements of the old array are copied into. The old array can then be deallocated. For *Shrink* operations, when the allocated array is, for example, $\frac{N}{\alpha^2}$ full, a new array of size $\frac{N}{\alpha}$ is allocated. All elements are copied into it from the old array, after which the old array is deallocated. Both *Grow* and *Shrink* operations take amortized $\mathcal{O}(1)$ time, and a suitable choice of α often proves to be satisfactory for many practical applications such as C++ vectors, Java ArrayLists, Python List Objects and more [6]. The geometric expansion / shrinking technique does, however, use $\mathcal{O}(N)$ excess memory to store the array in addition to a temporary $\mathcal{O}(N)$ additional excess memory when copying the elements of the array to the new array. As such, the geometric expansion technique, meanwhile efficient in terms of update and access operations, is not optimal regarding excess space.

3.3 Sitarski's HAT Data Structure

Sitarski describes in [2] a data structure named hashed arrays tree (HAT), a relatively simple implementation of a resizable array. The HAT data structure maintains an index array of size B, where $\sqrt{N} \leq B \leq 4\sqrt{N}$, and $\lceil N/B \rceil$ data blocks each of size B as seen in figure 3.1. The N items currently stored in the array are stored sequentially in the first $\lceil N/B \rceil$ data blocks, with the $\lceil N/B \rceil$-th data block only being partially full if N is not divisible by B. The elements in the index block are pointers to the data blocks such that the i-th element contains a pointer to the i-th data block. The HAT data structure thus only uses at most $N + 2B + \mathcal{O}(1)$ words of memory, where the $2B$ accounts for the index block and the one potentially empty data block.

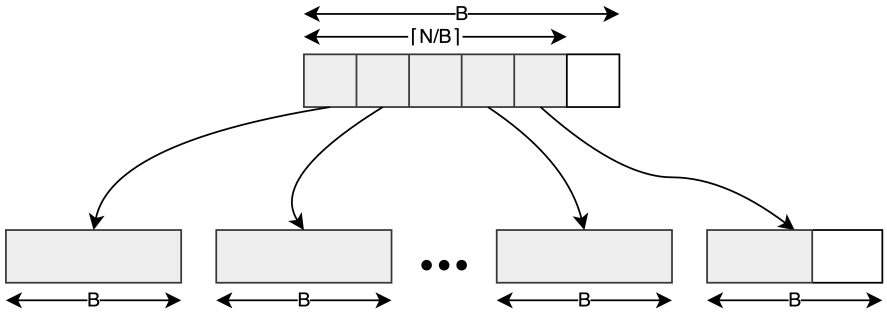


Figure 3.1: Sketch of Sitarski's HAT data structure.

Access operations are done in $\mathcal{O}(1)$ worst-case time, as the i-th item is stored in the $\lfloor N/B \rfloor$-th block at the $(i \mod B)$-th position. These can be computed fast in practice using bit-shifts if B is chosen as a power of two.

When $N < B^2$, performing a *Grow* operation is simple. If the $\lceil N/B \rceil$-th data block is not full, the element is added as the last item in the $\lceil N/B \rceil$-th data block. If the $\lceil N/B \rceil$-th data block is full, a new data block of size B is allocated, and the new element is added as the first item of this data block. This operation takes $\mathcal{O}(1)$ worst-case time, assuming that memory allocation takes constant time.

When N reaches B^2, the data structure is full and is rebuilt with B doubled. If B is chosen to be a power of two, it will stay a power of two, simplifying the indexing operations. When B is doubled, the capacity of the data structure is increased from N to $4N$. The amortized cost of a *Grow* operation is thus $\mathcal{O}(1)$ since the $\mathcal{O}(N)$ time to rebuild can be charged to the last $\frac{3}{4}N$ *Grow* operations that must have occurred before the previous rebuild.

The *Shrink* operation is also quite simple. If the element to delete is the first element of the last data block, the data block is deallocated. When $N = B^2/16$, the data structure is rebuilt with B halved. The amortized cost of a *Shrink* operation is also $\mathcal{O}(1)$ since $\mathcal{O}(N)$ operations must occur between two rebuild operations. Rebuild operations can be carried out while using only $\mathcal{O}(\sqrt{N})$ extra words of memory.

It is worth noting that there is a version of HAT that allows the $\lceil N/B \rceil$-th data block to be partially empty and the $(\lceil N/B \rceil + 1)$-th data block to be empty. Here the $(\lceil N/B \rceil + 1)$-th data block is only deallocated when both the $\lceil N/B \rceil$-th and $(\lceil N/B \rceil + 1)$-th data blocks are empty during a *Shrink* operation. This prevents sequential repeated deallocation and allocation for certain *Grow* and *Shrink* sequences. However, this variant comes at the cost of a more complex data structure and an extra B words of memory overhead.

3.4 Brodnik's Data structures

A few years after Sitarski developed the HAT data structure, Brodnik et al. [7] developed a solution in two variants with the same asymptotic bounds as HAT. The variant A data structure maintains an index array of size B with pointers to blocks of incrementally growing sizes starting from 1 as seen in figure 3.2. All data blocks except the last one are always full, and the last is potentially partially full. It is well known that the partial sum $1+2+3+\cdots+n$ is $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$. Since the index block has length B, at most B blocks can be full. Thus, we have $N \leq \sum_{k=1}^{B} k = \frac{B(B+1)}{2}$. It is easy to see that $B = \mathcal{O}(\sqrt{N})$. The last data block has size $\lceil \frac{\sqrt{8N+1}-1}{2} \rceil$ [7] which can potentially be empty. Thus, the potentially empty last data block and the index block give a total excess memory of $\mathcal{O}(\sqrt{N})$ used by this solution.

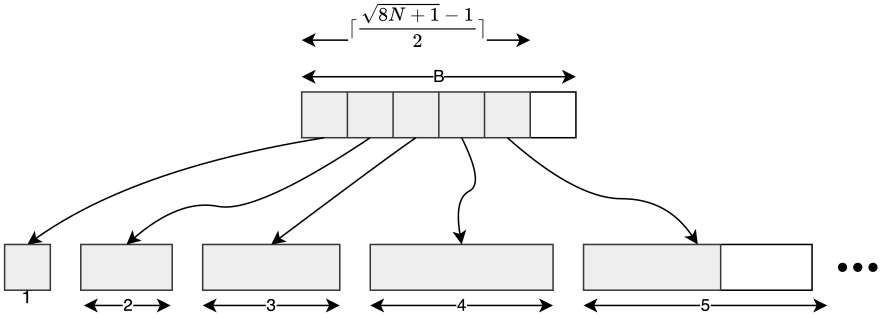


Figure 3.2: Sketch the variant A data structure by Brodnik et al.

Grow and *Shrink* operations are fairly simple in this setting. To perform a *Grow* operation, the last data block and the position within must be determined. If the last data block has space, the element is inserted as the last element of this block. Otherwise, the last data block is full, and a new one is allocated with a size 1 greater than the previous one. The element is inserted as the first element of the new data block, and a pointer to the data block is added to the index array. As for the *Shrink* operation, one has to check if the data block in which the last element resides only contains a single element. If this is the case, the data block is deallocated, and the pointer to it is removed from the index block. This procedure takes $\mathcal{O}(1)$ time, assuming allocation and deallocation of a memory block takes $\mathcal{O}(1)$ time. During both *Grow* and *Shrink* operations, the index array might have to increase or decrease in size. The index array is a dynamic array that can be implemented using geometric expansion, discussed in section 3.2 [7]. In section 3.2, it was shown that growing and shrinking a dynamic array implemented by geometric expansion takes amortized $\mathcal{O}(1)$ time. Thus, the *Grow* and *Shrink* operations of the solution by Brodnik et al. also take amortized $\mathcal{O}(1)$ time in total if locating the last element takes $\mathcal{O}(1)$ time. Rebuilding the index array also uses some excess memory temporarily depending on which method is used. When using the geometric expansion technique, it uses an additional $\mathcal{O}(\sqrt{N})$ excess space temporarily.

The i-th element is in the l-th data block, where $l = \lceil \frac{\sqrt{8i+1}-1}{2} \rceil$, at the $(i - \frac{l(l-1)}{2})$-th position [7]. The calculation to find the correct data block requires computing the square root of $8i + 1$. This takes $\Theta(\log \log i)$ worst-case time using Newton's method, which is the standard method for computing the square root fast [8]. This means that the access time is not the desired $\mathcal{O}(1)$ time. It also affects the *Grow* and *Shrink* operations, which use the same method to locate the last item. To reach optimal bounds, Brodnik et al. developed a data structure referred to here as version B, achieving the same asymptotic memory bounds and achieving faster access times by avoiding the need to calculate a square root [7]. The second version of the data structure is seen in figure 3.3. Here,

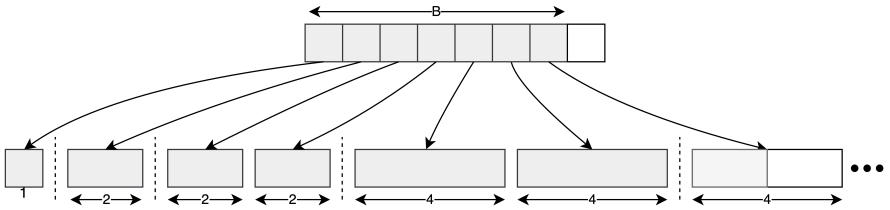


Figure 3.3: Sketch the variant A data structure by Brodnik et al.

the sizes of the data blocks are all a power of 2. Brodnik et al. introduce the concept

of super blocks, which are clusters of data blocks. These are not to be implemented but serve purposefully when analyzing the data structure and are marked with dashed lines in figure 3.3. The k-th super block consists of $2^{\lceil \frac{k}{2} \rceil}$ data blocks each of size $2^{\lfloor \frac{k}{2} \rfloor}$ [7]. Multiplying these, each super block contains 2^k elements. Brodnik et al. showed that the number of super blocks is $\lceil \log_2(n+1) \rceil$ [7]. To that extent, the i-th element is located in the $\lceil \log_2(i+1) \rceil$-th super block. This is exactly the position of the leading 1 bit in the binary representation of $i+1$. Not only is this a $\mathcal{O}(1)$ operation, but it is a very fast operation, as it is a low-level instruction supported by most modern computers. The full procedure for locating an element requires finding the relevant data block within the found super block and the specific position within. Brodnik et al. describe how to do this in worst case constant time using a few bit operations [7].

3.5 SQarray

A. Moffat and J. Mackenzie came up with an improvement to the data structures by Brodnik et al., but here arranging the data blocks differently [9]. In this data structure, named *SQarray*, the data blocks are all a power of two with two segments of length 2, three of length 4, six of length 8 and so on. In general, for any integer $l \geq 2$, there will be $3 \cdot 2^{l-2}$ data blocks of length 2^l [9]. Figure 3.4 shows a sketch of this configuration. Moffat and

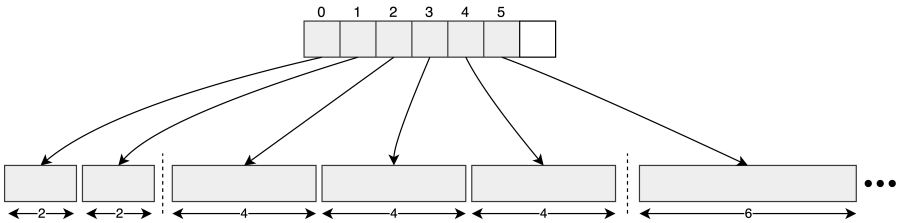


Figure 3.4: Sketch of the SQarray data structure.

Mackenzie proved that the longest segment cannot have more than $2\sqrt{N}$ empty array spots [9]. Furthermore, they showed that the index array must be strictly smaller than $3\sqrt{N} - 2$. Thus the total additional space usage is bound to $\mathcal{O}(\sqrt{N})$. Similar to previous solutions, a crucial part of access operations is locating which data block contains the element and the position within that data block. Both of these can be done using bit manipulation tricks since the data blocks all have sizes which are a power of 2, similar to the second structure by Brodnik et al. This yields a fast access time of $\mathcal{O}(1)$ worst case.

The index array is implemented using geometric expansion such that rebuilding this contributes an amortized $\mathcal{O}(1)$ time to the *Grow* and *Shrink* operations similarly to what was seen in section 3.3. Growing and shrinking the data structure locates the last element of the data structure and is simply a matter of checking the last data block and allocating or deallocating accordingly.

3.6 Data Structures by Tarjan and Zwick

Tarjan and Zwick [1] describes a family of data structures. They distinguish between the space needed for storing a resizable array and the temporary space occasionally needed while growing or shrinking the resizable array. The following definition is introduced to describe a data structure that distinguishes between space usage.

Definition 3.6.1. $(s(N), t(N))$*-implementation. Let* $s(N)$ *and* $t(N)$ *be two non-decreasing functions. A resizable array data structure is said to be an* $(s(N), t(N))$*-implementation if*

it uses at most $N + s(N)$ space to store a resizable array of size N and at most $N + t(N)$ space during a Grow or Shrink operation on a resizable array of size N [1].

3.6.1 Simple $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ data structure

The simple $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$-implementation is based on Sitarski's HAT data structure but maintains two index arrays of size B and $2B$ respectively, where $N^{1/3} \leq B \leq 4N^{1/3}$. The index array of size B contains pointers to large blocks of size B^2, and the index array of size $2B$ contains pointers to small data blocks of size B as seen in figure 3.5. The large data blocks are always full, and at most, one small block is partially full. Two counters, n_1 and n_2, are maintained to indicate how full the two index arrays are. The N items currently in the array are stored sequentially, starting with the large blocks and the elements added last stored in the small blocks. The access operations are similar to that of Sitarski's solution and thus support access operations in $\mathcal{O}(1)$ worst-case time. Since the elements are stored sequentially, it is easy to determine whether the element is in a large or small data block based on n_1 and n_2. If the element is located in a large block, it is in the i/B^2-th data block at the $(i \mod B^2)$-th position. If it is located in a small block, it is in the $(i - n_2B^2)/B$-th data block at the $((i - n_2B^2) \mod B)$-th position. Like Sitarski's HAT data structure, this solution benefits greatly from choosing B as a power of two, enabling fast bit operations.

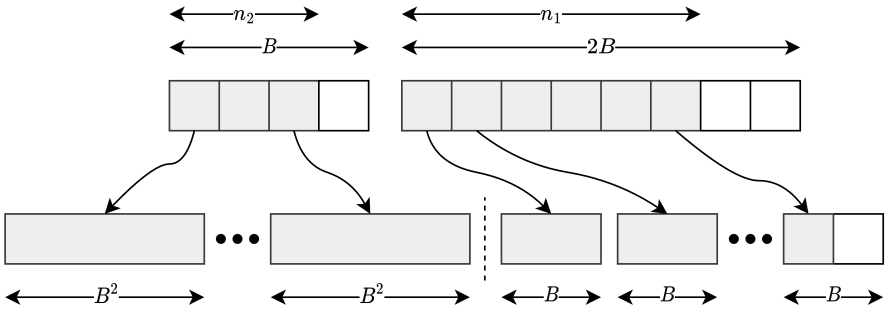


Figure 3.5: Sketch of the $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ version of Tarjan and Zwick's data structure.

For *Grow* operations, when $N = B^3$, the data structure is rebuilt with B doubled. If $N < B^3$, but all small data blocks are full, the first B of them are combined into a large data block, and the remaining B pointers are shifted B places back in the index array. If N is divisible by B, a new small data block is allocated, and the element is inserted as the first element of this new small data block. Otherwise, the element is inserted as the last element in the last small data block.

The shrink operation is similar. When $N = (\frac{B}{4})^3$ the data structure is rebuilt with B halved. If $N > (\frac{B}{4})^3$, but all small data blocks are empty, a large data block is split into B small data blocks. If the last element is the only element in the data block, the last small data block can be deallocated. This can easily be checked, for example, by checking if $(N \mod B) = 1$.

Since the data structure maintains the parameter B to be within the bound $N^{1/3} \leq B \leq 4N^{1/3}$, the data structure uses at most $N + \mathcal{O}(N^{1/3})$ space to store the array of size N. During a *CombineBlocks* or *Rebuild* operation at most $N + \mathcal{O}(N^{2/3})$ space is temporarily used, as at most a new large data block is allocated to which elements are copied into. The large data blocks are of size B^2, thus $\mathcal{O}((N^{1/3})^2) = \mathcal{O}(N^{2/3})$ extra space is temporarily used.

It is easy to see that *Grow* and *Shrink* operations take $\mathcal{O}(1)$ amortized time since the number of elements in the data structure increases from N to $8N$ when B is doubled and the data structure rebuilt. The $\mathcal{O}(N)$ time to rebuild can then be charged to the $\frac{7}{8}N$ *Grow* operations that must have occurred before the previous *Rebuild* operation. Furthermore, each element is moved maximally once from a small block to a large block before a new rebuild, giving a total of N copy operations, which only adds a constant factor compared to the rebuild cost. For a formal proof, see [1].

3.6.2 General $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$ data structure

The simple $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$-implementation can be generalized to a $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$-implementation for every $r \geq 2$. For a given $r \geq 2$, the data structure maintains a parameter B such that $N^{1/r} \leq B \leq 4N^{1/r}$. The generalized data structure maintains $r-1$ index arrays of size $2B$, each of which has a pointer to it from a top array of size $r-1$. The items are kept in data blocks of size $B, B^2, ..., B^{r-1}$, i.e. sizes that are powers of B (see figure 3.6).

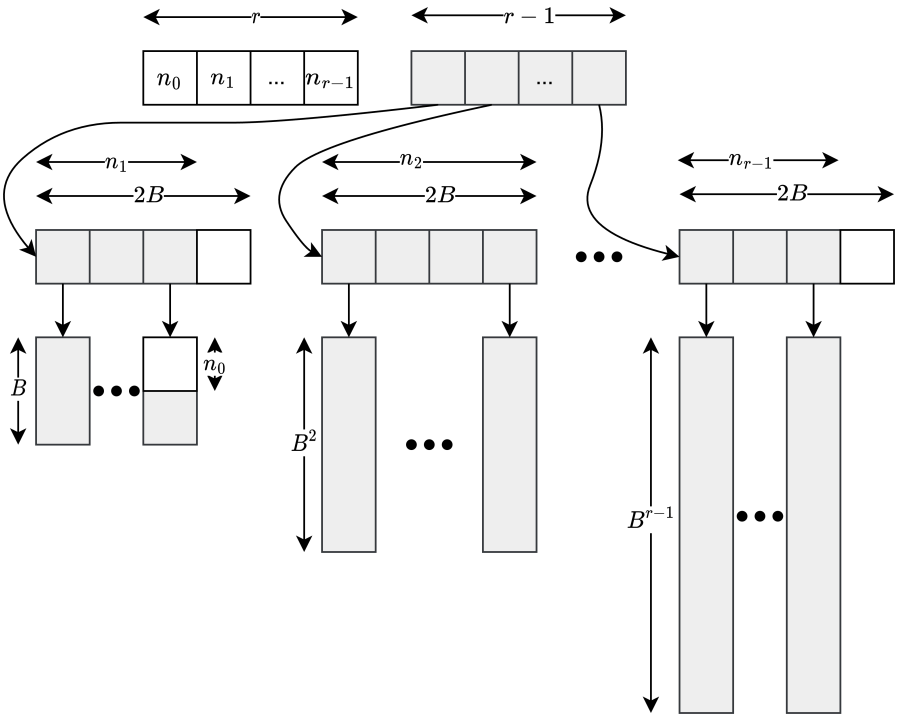


Figure 3.6: Sketch of the $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$ version of Tarjan and Zwick's data structure.

Data blocks of size $B^2, ..., B^{r-1}$ are always full, and at most, one block of size B is partially full. If the last block of size B is not full, then the *Grow* operation is easy. If it is full, but there are less than $2B$ data blocks in the first level, then a new block of size B is allocated, and the item is added as the first item of this data block. If $2B$ full blocks of size B^i exist, then a new block of size B^{i+1} is allocated, and B of these are copied over to the new block in the appropriate order. Then the B blocks of size B^i are deallocated. Thus, in the case that $2B$ full blocks of size B exist, B of them are copied over and deallocated, allowing space for a new block of size B to be allocated and the item to be inserted. When $N = B^r$, the data structure is rebuilt with B doubled.

Shrink operations are quite similar. If the last block of size B has more than 1 element, the

operation is easy. If the last block only has a single element, this block can be deallocated. If during a *Shrink* operation, no blocks of size B are available, a block of size B^i, for the smallest possible i, is split into $B-1$ blocks each of size $B^i, B^{i-1}, \ldots, B^2$ and B blocks of size B. The amortized cost of *Grow* and *Shrink* operations are $\mathcal{O}(r)$. A formal proof can be seen in [1].

An element can be accessed at a particular index in $\mathcal{O}(r)$ worst-case time. First, the correct level in which the element resides must be found. It is known that there are n_i full data blocks of size B^i at level i and that the higher levels contain the lower indices. The correct level can then be found by looping backwards through the levels and subtracting the number of elements in each level from the index until 0 or less is reached. This loop causes the bottleneck of the asymptotic running time. Once the correct level i is found, the index in the corresponding index block is found by dividing the remainder by B^i, and the index in the data block is found by calculating the remainder modulo B^i. These are both $\mathcal{O}(1)$ time operations. Thus, the $\mathcal{O}(r)$ time to find the correct level takes precedence. There are some tricks to make it possible to find the correct level in $\mathcal{O}(1)$ time (see section 4.3), which makes the total access time $\mathcal{O}(1)$ worst case.

As the data structure maintains $r-1$ index blocks each of size $2B$ and B is maintained such that $N^{1/r} \leq B \leq 4N^{1/r}$, the total space used to store an array containing N items is $N + r + (r-1) + 2B \cdot (r-1) + B = N + \mathcal{O}(rN^{1/r})$. The maximum amount of extra space that the data structure can temporarily use is $\mathcal{O}(N^{1-1/r})$, which is when a new block of size B^{r-1} is allocated. This is because $\mathcal{O}(B^{r-1}) = \mathcal{O}((\sqrt[r]{N})^{r-1}) = \mathcal{O}(N^{1-1/r})$.

3.7 Overview

An overview of the different solution's space and time complexities can be seen in table 3.1. As previously mentioned, *Grow* and *Shrink* operations are only at the end of the array. Inserting an element in the middle of the array is not covered.

	Access	Grow / Shrink	Excess space	Temp. excess space
Naive DA	$\mathcal{O}(1)$	$\mathcal{O}(N)$	0	$\mathcal{O}(N)$
Geometric Expansion	$\mathcal{O}(1)$	$\mathcal{O}(1)^+$	$\mathcal{O}(N)$	$\mathcal{O}(N)$
Sitarski (HAT)	$\mathcal{O}(1)$	$\mathcal{O}(1)^+$	$\mathcal{O}(\sqrt{N})$	$\mathcal{O}(\sqrt{N})$
Brodnik version A	$\mathcal{O}(1)$	$\mathcal{O}(1)^+$	$\mathcal{O}(\sqrt{N})$	$\mathcal{O}(\sqrt{N})$
Brodnik version B	$\mathcal{O}(1)$	$\mathcal{O}(1)^+$	$\mathcal{O}(\sqrt{N})$	$\mathcal{O}(\sqrt{N})$
SQarray	$\mathcal{O}(1)$	$\mathcal{O}(1)^+$	$\mathcal{O}(\sqrt{N})$	$\mathcal{O}(\sqrt{N})$
Simple Tarjan and Zwick	$\mathcal{O}(1)$	$\mathcal{O}(1)^+$	$\mathcal{O}(N^{1/3})$	$\mathcal{O}(N^{2/3})$
General Tarjan and Zwick	$\mathcal{O}(r)$	$\mathcal{O}(r)^+$	$\mathcal{O}(rN^{1/r})$	$\mathcal{O}(N^{1-1/r})$

Table 3.1: Comparison of time and space complexities of the explored solutions. The superscript $+$ means amortized bounds.

4 Implementation

We implemented each solution mentioned in section 3 in C++. The code can be seen at https://github.com/Christianrhk/OptimalResizableArrays. All solutions have been implemented as templates to allow the dynamic array solutions to store any data type. We attempted to optimize the code to the best of our ability, such as eliminating division and modulo operations and utilizing bit manipulation instead. The implementation details of the simple data structures, Sitarski's HAT, Brodnik's solutions and SQarray will not be discussed here as they have been implemented before. However, to the best of our knowledge, we are the first to implement the solutions by Tarjan and Zwick [1]. This will therefore be discussed in this section. Some implementation details were given in the paper. Other operations, such as rebuilding the data structures, were left open for interpretation and are thus our own. All tests were performed on a desktop PC running Windows 10, with a GTX 1660 Super graphics card, an Intel Core i5-4680K CPU and 16GB of RAM.

4.1 Implementation details of the simple $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ data structure

The pseudocode for *Grow* is found in algorithm 1 and for *Shrink* in algorithm 2. *CombineBlocks* and *SplitBlocks* are relatively straightforward and are seen in algorithm 3 and 4 respectively. In the pseudocode, A_1 refers to the index block with data blocks of size B^2, and A_2 refers to the index block containing data blocks of size B. It should be noted that many of the calculations can be optimized significantly by utilizing bit shifts instead. Line 1 in algorithm 1 can be optimized to be $N = (B << \log(B)) << \log(B)$, where $<<$ is the bitwise left shift operator and $>>$ is the bitwise right shift operator. It is also advantageous for the data structure to maintain $\log(B)$ as a variable since it can speed up several calculations. To maintain this variable, increment the variable by one instead of calculating $\log(B)$ when doubling B. Similarly, when halving B, decrement by the variable by one. Likewise line 1 in algorithm 2 can also be optimized to be $N = (1 << ((\log(B) - 2) \cdot 3))$.

Algorithm 1 *Grow(a)*

1: **if** $N = B^3$ **then**
2: $\quad Rebuild(2B)$
3: **else if** $n_1 = 2B$ & N % $B = 0$ **then**
4: $\quad CombineBlocks()$
5: **end if**
6: **if** N % $B = 0$ **then**
7: $\quad A_2[n1] \leftarrow Allocate(B)$
8: **end if**
9: $A_2[n_1 - 1][N$ % $B] \leftarrow a$
10: $N \leftarrow N + 1$

Algorithm 2 *Shrink()*

1: **if** $N = \frac{B}{4}^3$ **then**
2: $\quad Rebuild(\frac{B}{2})$
3: **end if**
4: **if** $n_1 = 0$ **then**
5: $\quad SplitBlocks()$
6: **else if** N % $B = 1$ **then**
7: $\quad Deallocate(A_2[n_1 - 1])$
8: $\quad n_1 \leftarrow n1 - 1$
9: **end if**
10: $N \leftarrow N - 1$

The *Rebuild* function, however, is a bit more complicated, and we have come up with several versions of it. The idea behind the first version was to simply call a modified *Get* to retrieve the items and insert them accordingly in the new data structure. The old allocations are deallocated as they are emptied. The pseudocode for the first version of

Algorithm 3 *CombineBlocks()*

1: $A_1[n_2] \leftarrow Allocate(B^2)$
2: **for** $i \leftarrow 0$ to B **do**
3: $\quad Copy(A_2[i], 0, A_1[n_2], i \cdot B, B)$
4: $\quad Copy(A_2[i+B], 0, A_2[i], 0, B)$
5: $\quad Deallocate(A_2[i+B])$
6: **end for**
7: $n_2 = n_2 + 1$
8: $n_1 = \frac{n_1}{2}$

Algorithm 4 *SplitBlocks()*

1: **for** $i \leftarrow 0$ to B **do**
2: $\quad A_2[n_1] \leftarrow Allocate(B)$
3: $\quad Copy(A_1[n_2 - 1], i \cdot B, A_2[i], 0, B)$
4: **end for**
5: $Deallocate(A_1[n_2 - 1])$
6: $n_2 = n_2 - 1$
7: $n_1 = B$

our *Rebuild* can be seen in figure 5. This *RebuildV1* version fills as many large blocks as possible before moving on to the small blocks and finally handling any potential left-overs. For each element during the rebuild, it calls the function *GetAndDelete* to retrieve the element and insert it accordingly in the newly allocated data block. If the element is the last element in the old block in which it resided, that data block is deallocated. The *GetAndDelete* function is not shown here, as it is very similar to the *get* described in section 3.6. The *GetAndDelete* function is $\mathcal{O}(1)$ worst-case time assuming that deallocation occurs in $\mathcal{O}(1)$ time. *GetAndDelete* is called exactly N times in the rebuild version, and as such, has a time complexity of $\mathcal{O}(N)$. Furthermore, it supports a temporary excess space bound of $\mathcal{O}(N^{2/3})$, since a *Rebuild* triggered by a *Grow* operation will at most use $2B + 4B + (2B)^2 = \mathcal{O}(B^2)$ temporary excess memory. As we are rebuilding when $N = B^3$, thus $B = N^{1/3}$, we have $\mathcal{O}(B^2) = \mathcal{O}(N^{1/3^2}) = \mathcal{O}(N^{2/3})$ temporary excess memory. For a *Rebuild* triggered by a *Shrink* operation, the data structure will at most use $B/2 + B + (B/2)^2 = \mathcal{O}(B^2)$ temporary excess memory. It is rebuilt when $N = \frac{B}{4}^3$, thus $B = 4N^{1/3}$ and by the same argument as for grow, we have $\mathcal{O}(N^{2/3})$ temporary excess memory.

The second *Rebuild* version does not utilize a modified *Get* to retrieve the elements and insert them accordingly. Instead, it is based on the fact that it takes 4 large blocks of size B^2 to create one large block of size $(2B)^2$ (since $\frac{(2B)^2}{B^2} = 4$). Likewise, for the small blocks, it takes two small blocks of size B to create one small block of size $2B$. If B is being halved instead of doubled, this works oppositely. Thus, this *Rebuild* version behaves differently if B is being doubled or halved. The pseudocode for *RebuildV2*, which have been separated for *Grow* and *Shrink*, can be seen in figure 6 and 7. Recall that the data structure maintains n_1 and n_2, which are the number of small and large blocks, respectively. For the *Grow* version of the *RebuildV2* operation, as many large blocks as possible are created of size $(2B)^2$. If n_2 is not divisible by 4, the remaining large blocks will be converted to small blocks of size $2B$. One large block of size B^2 can make $B/2$ small blocks of size $2B$ (since $\frac{B^2}{2B} = B/2$). Any small blocks are then converted. It takes two small blocks of size B to convert to one small block of size $2B$. There must always be an even number of small blocks, assuming we start at $B = 2$. There is no benefit to starting at $B = 1$, as we would have to rebuild to $B = 2$ already when $N = 1$. Any old large or small blocks are deallocated as soon as possible to keep the memory overhead as small as possible.

The *Shrink* version of the *RebuildV2* operation works oppositely to the *Grow* version of the *RebuildV2* operation. Each large block of size B^2 is converted into 4 large blocks of size $(B/2)^2$. There must be space for all of the new large blocks. Since we are rebuilding when $N = (B/4)^3$ if all elements were located in the large data blocks, there would be

Algorithm 5 *rebuildV1(B')*

1: $A_{1_{temp}} \leftarrow Allocate(B')$
2: $A_{2_{temp}} \leftarrow Allcoate(2B')$
3: $toRebuild \leftarrow N$
4: **for** $n_2 \leftarrow 0$ to $toRebuild/B'^2$ **do**
5: $\quad A_{1_{temp}}[n_2] \leftarrow Allocate(B'^2)$
6: $\quad$ **for** $j \leftarrow 0$ to B'^2 **do**
7: $\quad\quad A_{1_{temp}}[n_2][j] \leftarrow GetAndDelete(N - toRebuild)$
8: $\quad\quad toRebuild \leftarrow toRebuild - 1$
9: $\quad$ **end for**
10: **end for**
11: **for** $n_1 \leftarrow 0$ to $toRebuild/B'$ **do**
12: $\quad A_{2_{temp}}[n_1] \leftarrow Allocate(B')$
13: $\quad$ **for** $j \leftarrow 0$ to B' **do**
14: $\quad\quad A_{2_{temp}}[n_1][j] \leftarrow GetAndDelete(N - toRebuild)$
15: $\quad\quad toRebuild \leftarrow toRebuild - 1$
16: $\quad$ **end for**
17: **end for**
18: **if** $toRebuild > 0$ **then**
19: $\quad A_{2_{temp}}[n_1] \leftarrow Allocate(B')$
20: $\quad$ **for** $j \leftarrow 0$ to $toRebuild$ **do**
21: $\quad\quad A_{2_{temp}}[n_1][j] \leftarrow GetAndDelete(N - toRebuild - j)$
22: $\quad$ **end for**
23: **end if**
24: $A_1 \leftarrow A_{1_{temp}}$
25: $A_2 \leftarrow A_{2_{temp}}$
26: $B \leftarrow B'$

$\frac{(B/4)^3}{B^2} = \frac{B}{64}$ large data blocks. Each of these makes 4 new data blocks, yielding a total of $\frac{B}{64} \cdot 4 = \frac{B}{16}$ large data blocks with $\frac{B}{2}$ spots in the index block. Thus, since $\frac{B}{2} > \frac{B}{16}$, there will always be space. As many large blocks of size $(B/2)^2$ as possible are made from the small blocks of size B. It takes $B/4$ small blocks of size B to make one large block of size $(B/2)^2$ (since $\frac{B^2/4}{B} = B/4$). By the same argument for why there is always space for the large blocks after the rebuild, there will also be space for these large blocks. Lastly, any remaining small blocks are converted into small blocks. Each of the remaining small blocks of size B is converted into 2 small blocks of size $B/2$. There must also be space for all the new small blocks since we rebuild when $N = (B/4)^3$, and after the rebuild, we have a capacity of $(B/2)^3 + 2(B/2)^2$, which is greater than $(B/4)^3$. Since as many large blocks as possible have been created, the remaining must fit into the new small blocks. Both the *RebuildV2* for *Grow* and *Shrink* take $\mathcal{O}(N)$ time, assuming deallocation and allocation take constant time. For the *Grow* version, at most $2B + (2B)^2$ extra words of memory are temporarily allocated. For the *Shrink* version, at most $B/2 + (B/2)^2$ extra words of memory are temporarily allocated. Thus, by the same argument as for *RebuildV1*, the data structure uses at most $\mathcal{O}(N^{2/3})$ excess temporary memory.

Both of our versions of the rebuild function will, during a *Shrink* operation, fill up as many data blocks of size B^2 as possible and then fill up data blocks of size B with the remaining elements. There is no guarantee that any blocks of size B will be created in this process, leaving the A_2 index block empty. The *Shrink* operation first rebuilds and then removes

the element. It is, in this case, necessary to perform a *SplitBlocks* operation to get a data block of size B from which the element can be removed. This is a slight deviation from the pseudocode provided by Tarjan and Zwick for the *Shrink* operation. This is not always a problem, but it results from how we have chosen to implement the rebuilding of the data structure. Performing a *SplitBlocks* operation takes $\mathcal{O}(B^2) = \mathcal{O}(N)$ time, which is the same as rebuilding the data structure. Thus, performing *SplitBlocks* right after rebuilding does not change the amortized bounds of the *Shrink* operation.

Algorithm 6 *RebuildV2(B') // Grow*

$A_{1_{temp}} \leftarrow Allocate(B')$
for $i \leftarrow 0$ to $n_2/4$ **do**
    $A_{1_{temp}}[i] \leftarrow Allocate(B'^2)$
    $Copy(A_1[4i], 0, A_{1_{temp}}[i], 0, B'^2)$
    $Copy(A_1[4i+1], 0, A_{1_{temp}}[i], B'^2, B'^2)$
    $Copy(A_1[4i+2], 0, A_{1_{temp}}[i], 2B'^2, B'^2)$
    $Copy(A_1[4i+3], 0, A_{1_{temp}}[i], 3B'^2, B'^2)$
    $Deallocate(A_1[4i])$
    $Deallocate(A_1[4i+1])$
    $Deallocate(A_1[4i+2])$
    $Deallocate(A_1[4i+3])$
end for
$A_{2_{temp}} \leftarrow Allocate(2B')$
$n_{1_{temp}} \leftarrow 0$
for $i \leftarrow 0$ to n_2 % 4 **do**
    for $j \leftarrow 0$ to $B/2$ **do**
        $A_{2_{temp}}[n_{1_{temp}}] \leftarrow Allocate(B')$
        $Copy(A_1[n_2 - (n_2\%4) + i], j \cdot B', A_{2_{temp}}[n_{1_{temp}}], 0, B')$
        $n_{1_{temp}} \leftarrow n_{1_{temp}} + 1$
    end for
    $Deallocate(A_1[n_2 - (n_2 \% 4) + i])$
end for
$Deallocate(A_1)$
for $i \leftarrow 0$ to $n1/2$ **do**
    $A_{2_{temp}}[n_{1_{temp}}] \leftarrow Allocate(B')$
    $Copy(A_2[i], 0, A_{2_{temp}}[n_{1_{temp}}], 0, B)$
    $Copy(A_2[i+1], 0, A_{2_{temp}}[n_{1_{temp}}], B, B)$
    $Deallocate(A_2[i])$
    $Deallocate(A_2[i+1])$
    $n_{1_{temp}} \leftarrow n_{1_{temp}} + 1$
end for
$Deallocate(A_2)$
$A_1 \leftarrow A_{1_{temp}}$
$A_2 \leftarrow A_{2_{temp}}$
$B \leftarrow B'$
$n_1 \leftarrow n_{1_{temp}}$
$n_2 \leftarrow n_2/4 + n_2$ % 4

Algorithm 7 *RebuildV2(B') // Shrink*

$A_{1_{temp}} \leftarrow Allocate(B')$
for $i \leftarrow 0$ to n_2 **do**
    $A_{1_{temp}}[4i] \leftarrow Allocate(B'^2)$
    $A_{1_{temp}}[4i+1] \leftarrow Allocate(B'^2)$
    $A_{1_{temp}}[4i+2] \leftarrow Allocate(B'^2)$
    $A_{1_{temp}}[4i+3] \leftarrow Allocate(B'^2)$
    $Copy(A_1[i], 0, A_{1_{temp}}[4i], 0, B'^2)$
    $Copy(A_1[i], B'^2, A_{1_{temp}}[4i+1], 0, B'^2)$
    $Copy(A_1[i], 2 \cdot B'^2, A_{1_{temp}}[4i+2], 0, B'^2)$
    $Copy(A_1[i], 3 \cdot B'^2, A_{1_{temp}}[4i+3], 0, B'^2)$
    $Deallocate(A_1[i])$
end for
$Deallocate(A_1)$
for $i \leftarrow 0$ to $n_1/(B/4)$ **do**
    $A_{1_{temp}}[4n_2 + i] \leftarrow Allocate(B'^2)$
    for $j \leftarrow 0$ to $B/4$ **do**
        $Copy(A_2[i \cdot (B/4) + j], 0, A_{1_{temp}}[4n_2 + i], jB, B)$
        $Deallocate(A_2[i \cdot (B/4) + j])$
    end for
end for
$A_{2_{temp}} \leftarrow Allocate(2B')$
for $i \leftarrow n_1 - (n_1\%(B/4))$ to $n1$ **do**
    $A_{2_{temp}}[2i] \leftarrow Allocate(B')$
    $A_{2_{temp}}[2i+1] \leftarrow Allocate(B')$
    $Copy(A_2[i], 0, A_{2_{temp}}[2i], 0, B')$
    $Copy(A_2[i], B', A_{2_{temp}}[2i+1], 0, B')$
    $Deallocate(A_2[i])$
end for
$Deallocate(A_2)$
$A_1 \leftarrow A_{1_{temp}}$
$A_2 \leftarrow A_{2_{temp}}$
$n_2 \leftarrow 4 \cdot n_2 + n_1 - (n_1\%(B/4))$
$n_1 \leftarrow n_1$ % $(B/4)$
$B \leftarrow B'$

4.2 Implementation details of the general $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$ data structure

The *Grow* and *Shrink* operations for the $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$-implementation are quite similar to those of the $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$-implementation. They can be seen in algorithm 8 and 9, respectively. The pseudocode for *CombineBlocks* and *ShrinkBlocks* can also be seen in algorithm 10 and 11 respectively [1].

Algorithm 8 *Grow(a)*

1: **if** $N = B^r$ **then**
2: $Rebuild(2B)$
3: **else if** $n_1 = 2B$ & $n_0 = B$ **then**
4: $CombineBlocks()$
5: **end if**
6: **if** $n_1 = 0$ **or** $n_0 = B$ **then**
7: $A[1][n_1] \leftarrow Allocate(B)$
8: $n_1 \leftarrow n_1 + 1$
9: $n_0 = 0$
10: **end if**
11: $A[1][n_1 - 1][n_0] \leftarrow a$
12: $n_0 \leftarrow n_0 + 1$
13: $N \leftarrow N + 1$

Algorithm 9 *Shrink()*

1: **if** $N = (\frac{B}{4})^r$ **then**
2: $Rebuild(\frac{B}{2})$
3: **end if**
4: **if** $n_1 = 0$ **then**
5: $SplitBlocks()$
6: **end if**
7: $n_0 \leftarrow n_0 - 1$
8: $N \leftarrow N - 1$
9: **if** $n_0 = 0$ **then**
10: $Deallocate(A[1][n_1 - 1])$
11: $n_0 \leftarrow B$
12: $n_1 \leftarrow n1 - 1$
13: **end if**

Algorithm 10 *CombineBlocks()*

1: $k \leftarrow min\{i \in [1 : r - 1] | n_i < 2B\}$
2: **for** $i \leftarrow k - 1$ to 1 **do**
3: $A[i + 1][n_{i+1}] \leftarrow Allocate(B^{i+1})$
4: **for** $j \leftarrow 0$ to $B - 1$ **do**
5: $Copy(A[i][j], 0, A[i + 1][n_{i+1}], jB^i, B^i)$
6: $Deallocate(A[i][j])$
7: $A[i][j] \leftarrow A[i][j + B]$
8: **end for**
9: $n_i \leftarrow B$
10: $n_{i+1} \leftarrow n_{i+1} + 1$
11: **end for**

Algorithm 11 *SplitBlocks()*

1: $k \leftarrow min\{i \in [1 : r - 1] | n_i > 0\}$
2: **for** $i \leftarrow k - 1$ to 1 **do**
3: $n_{i+1} \leftarrow n_{i+1} - 1$
4: **for** $j \leftarrow 0$ to $B - 1$ **do**
5: $A[i][j] \leftarrow Allocate(B^i)$
6: $Copy(A[i + 1][n_{i+1}], jB^i, A[i][j], 0, B^i)$
7: **end for**
8: $Deallocate(A_1[n_2 - 1])$
9: **end for**

The data structure also maintains $\log(B)$ as a variable since this can be used to speed up some calculations significantly. Line 1 in algorithm 8 can then be calculated as $N = (1 << (\log(B) \cdot r))$. Any time there is a division or multiplication by a power of two, bit shifts should be utilized to optimize the code further. For example in line 3 of algorithm 8, $2B$ should be implemented as $B << 1$. Line 1 in algorithm 9 can also be optimized and should be implemented as $N = (1 << ((\log(B) - 2) \cdot r))$.

The *Rebuild* operation we implemented for the $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$-implementation were done in the same manner as the first version of the simple $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$-implementation's rebuild that was shown in algorithm 5. All elements are sequentially retrieved in increasing order and inserted into the new data structure with B halved or doubled using the *GetAndDelete* function, which deallocates the data block if it retrieves

the last element in a given data block. However, with a $\mathcal{O}(r)$ access time, this version of *Rebuild* takes $\mathcal{O}(rN)$ time. Although, rebuilding in $\mathcal{O}(rN)$ time does not change the amortized bounds of *Grow* and *Shrink* of $\mathcal{O}(r)$ as mentioned in section 3.6.2. With the constant time access described in section 4.3, this can be reduced to $\mathcal{O}(N)$ time for a *Rebuild* operation since the r factor from the previous solution stems from the fact that we are accessing N elements taking $\mathcal{O}(r)$ time each. Yet, we found this solution slower in practice and requires more excess memory, as later shown in section 5. An even simpler solution stems from realizing that we can iterate through the entire data structure in $\mathcal{O}(N)$ time. This can be done by storing three additional variables, one to keep track of the current level, data block and index within the data block. These can, of course, be updated in constant time by checking if the current data block is full and if we have visited all elements in the current level. Using this method reduces the *Rebuild* operation to $\mathcal{O}(N)$ time. The *Rebuild* operation could potentially be implemented even faster in practice by following a similar approach to that of the second *Rebuild* implementation in the $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ data structure by Tarjan and Zwick described in section 4.1. More on this is discussed in section 7. However, similarly to our rebuild implementations for the $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ version, we also pack the larger levels as much as possible before moving on to the smaller data blocks in lower levels. The first level is not guaranteed to contain any data block after a *Rebuild* operation during *Shrink*. In this case, a *SplitBlocks* operation is necessary before removing the element. Similarly to the argument while describing this problem for the $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ version in section 4.1, occasionally performing a *SplitBlocks* right after rebuilding the data structure is a deviation from the pseudocode by Tarjan and Zwick [1]. However, this does not change the amortized bounds of the *Shrink* operation.

Similar to the argument for the *Rebuild* version for the $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$-implementation, the *Rebuild* uses only $\mathcal{O}(N^{1-1/r})$ temporary excess memory. While rebuilding, $r-1$ new index blocks are created, contributing $(r-1) \cdot 2B$ temporary excess space. Since $N^{1/r} \leq B \leq 4N^{1/r}$, substituting this in for B shows that the temporary excess memory contributed by the index blocks are $\mathcal{O}(rN^{1/r})$. Asymptotically, this is less that $\mathcal{O}(N^{1-1/r})$. The largest temporary excess memory from copying elements into the new data blocks stems from creating the largest data block of size B^{r-1}. Using the bounds of B, the temporary excess memory is $\mathcal{O}(B^{r-1}) = \mathcal{O}((N^{1/r})^{r-1}) = \mathcal{O}(N^{1-1/r})$. Thus, the total temporary excess space following this implementation is $\mathcal{O}(N^{1-1/r})$.

4.3 Constant time access

If r is chosen to be fixed, the *Get* and *Set* functions will already be constant time. Yet, the constant access method, regardless of r, can still potentially speed up the lookup times and is thus still of interest. Tarjan and Zwick [1] outlined a theoretical way of achieving $\mathcal{O}(1)$ worst-case access time. After conversing with Mr. Zwick, he informed us of a second and simpler method of achieving $\mathcal{O}(1)$ worst-case access time.

The first method is the one outlined in the original paper [1] and requires a couple of minor modifications to the general $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$ data structure. The first change is that there are $2 \cdot (B-1)$ data blocks at each level instead of $2B$. The second change is that n_1 is now the number of full data blocks of size B and not the number of allocated data blocks. With these changes $N = (n_0, n_1, \ldots, n_{r-1})_B = \sum_{j=0}^{r-1} n_j B^j$ is a redundant base B representation (least significant digit listed first). Furthermore, $N_k = (0, 0, n_k \ldots, n_{r-1})_B = \sum_{j=k}^{r-1} n_j B^j$, where the numbers $N_0, N_1, \ldots, N_r$ are maintained by data structure. This can be done without affecting the asymptotic cost of the *Grow* and *Shrink* operations.

To access the i-th element, a unique $0 \leq k \leq r-1$ is found such that $N_{k+1} \leq i < N_k$.

This is, of course, easily found in $\mathcal{O}(r)$ time by simple traversal or even $\mathcal{O}(\log(r))$ time using binary search, but this can also be found in $\mathcal{O}(1)$ time. First, determine k by finding the largest k for which $i_k < n_k$. This is found by $k = \frac{l}{logB}$, where l is the leading 1 position in the binary representation of $n \oplus i$. Note that [1] says leading 1 position in the binary representation of $n - i$, but Mr. Zwick clarified that XOR is the correct operation. If $N_{k+1} \leq i < N_k$, then N happens to be of the non-redundant base B, and thus we have the correct index. Otherwise we have $N_k \leq i \leq N_k + B^k$ and there must exist an $0 \leq l < k$ such that $n_l \geq B$ and $n_{l+1} = \cdots = n_{k-1} = B - 1$. The proof is explained in detail in Tarjan and Zwick [1]. Since there are exactly $B - 1$ blocks in the levels $n_{l+1}, \ldots, n_{k-1}$, if i is in one of these levels, the base B counter $i - N_k$ can be used to determine which level the element is in. The position of the largest index that is not $B-1$ will be the level in which the item resides. For any number, this can be found in constant time by counting the number of leading significant digits in a base B number, which is equal to $B - 1$, and subtracting it from the leading position. This calculation does not work if i is in level l. Thus, a method of determining if i is in l in constant time is required. We were unable to determine a method of doing this before being contacted by Mr. Zwick with the second method of obtaining constant time access. We then stopped attempting to find l in constant time. One solution would, however, be to maintain l_k for each k much like how N_k for $0 \leq k \leq r$ is maintained by the data structure at no additional cost to the asymptotic amortized time. However, this comes with the cost of storing an additional $r - 1$ integers.

The second method of achieving constant time access requires only one of the two minor modifications that the first method required. Namely, that n_1 is now the number of full data blocks of size B, i.e. it does not include a potentially partially full data block if it exists. This method essentially "searches" through the data structure from the other direction. We now have that $N = \sum_{j=0}^{r-1} n_j B^j = (n_{r-1}, \ldots, n_1, n_0)_B$, where $0 \leq n_j \leq 2B$ and $0 \leq j \leq r - 1$. The numbers $n_{r-1}, \ldots, n_1, n_0$ can be stored compactly in two words, $N^0 = \left(n^0_{r-1}, \ldots, n^0_1, n^0_0\right)_B$ and $N^1 = \left(n^1_{r-1}, \ldots, n^1_1, n^1_0\right)_B$, where $n_j = n^0_j + B n^1_j, 0 \leq n^0_j < B$ and $0 \leq n^1_j \leq 2$, for $0 \leq j < r$ [10].

As we now "search" in the other direction, we have that $N_k = (0, 0, n_{k-1} \ldots, n_1)_B = \sum_{j=0}^{k-1} n_j B^j$ ($N_r = N$ and $N_0 = 0$, opposite the first method). The data structure maintains the numbers N_k, which can also be computed in constant time instead [10].

To access the i-th item in the resizable array, it is again necessary to find a unique $0 \leq k < r$, but now obeying $N - N_{k+1} \leq i < N - N_k$. The i-th item resides in position $(i - (N - N_{k+1})) \mod B^k$ of the $\left\lfloor \frac{i-(N-N_{k+1})}{B^k} \right\rfloor$-th data block of size B^k. If $k = 0$, the i-th item resides in position $i \mod B$ of the partially filled data block of size B [10].

Let $x = (N-1)-i$. We need to find k such that $N_k \leq x < N_{k+1}$. Let $x = (x_{r-1}, \ldots, x_1, x_0)_B$. We then need to find the largest index l for $x_l > 0$. This can be done by finding the most significant set bit in x. More specifically, $l = \frac{msb(x)}{\log(B)}$. Now, if $x < N_l$ then $k = l - 1$. If $N_l \leq x < N_{l+1}$ then $k = l$. Otherwise $N_{l+1} \leq x$ and $k = l'$, where $l' > l$ is the smallest index for which either $n^0_{l'} > 0$ or $n^1_{l'} > 0$. This is found as $l' = \frac{lsb((N^0 \vee N^1) \wedge (\neg M_{l+1}))}{\log(B)}$, where $lsb(x)$ is the least significant set bit in x and M_k is a mask of $k \cdot \log(B)$ 1's, i.e. $M_k = 2^{k \cdot \log(B)} - 1$ [10].

However, this method only works for $B \geq 4$, leaving it unusable for $B = 2$. The easy solution is to start with $B = 4$, as it will only affect very short arrays. Furthermore, this solution assumes that $msb(x)$ and $lsb(x)$ can be computed in constant time.

5 Tests & results

The data structures from section 3 were tested based on the three parameters listed below:

1. **Excess memory**, i.e. the additional memory used in excess to the N words used for storing the elements. This includes the excess memory required to store the array and the temporary excess memory.
2. **Insertion time**, i.e. the time it takes to perform a *Grow* operation, thus inserting an element at the end of the resizable array.
3. **Random access time**, i.e. the time it takes to perform a *Get* operation of an element in the resizable array.

We decided not to consider testing the *Shrink* and *Set* operations as *Shrink* is very similar to *Grow* and *Set* is almost identical to the *Get* operation.

Although all implementations are made to support all data types, all tests were exclusively made using the primitive data type *int* in C++. The primitive integer type in C++ is always 4 bytes, which makes it easy to work with compared to, for example, strings where the size depends on the length of the string.

5.1 Excess memory

To test the excess memory used by the different data structures as the number of elements increase, we incrementally inserted one million elements and measured the excess memory usage after each insertion. The excess space was measured by counting and summing up the bytes used by all the data structure's components, i.e. arrays, pointers, variables, etc. To only get the excess memory used, we subtracted the total number of elements N used times the number of bytes used per element, which is 4 in the case of integers in C++.

All resizable array data structures require temporary excess space while copying elements from an old array to a new one. Tarjan and Zwick [1] even differentiated between the excess memory used for storing the array versus the excess memory temporarily used at certain times. During these tests, we only consider the maximum excess space required for performing a single *Grow* operation. This was done by keeping a counter throughout the operation, which would be increased every time a new array was allocated and similarly decreased when an array was deallocated by the appropriate amount. The counter's largest value throughout the operation is the one recorded as the maximum excess space.

Using this method, the data structures described in section 3, namely the naive dynamic array, geometric expansion, Sitarski's HAT as well as Brodnik's solutions, SQarray and the solutions by Tarjan and Zwick were tested, and the results plotted in a double logarithmic plot seen in figure 5.1. The general solution by Tarjan and Zwick was tested with $r \in [3, 10]$. This range was chosen with the reasoning that it is required to calculate B^r, and much higher values of r can cause numeric overflow. In figure 5.1, the results for $r = 3$, $r = 6$ and $r = 10$ are included.

It is immediately seen in figure 5.1 that the naive solution and the geometric expansion solution use significantly more memory than the other solutions most of the time. As mentioned in section 3.1, the naive data structure temporarily uses N excess space for

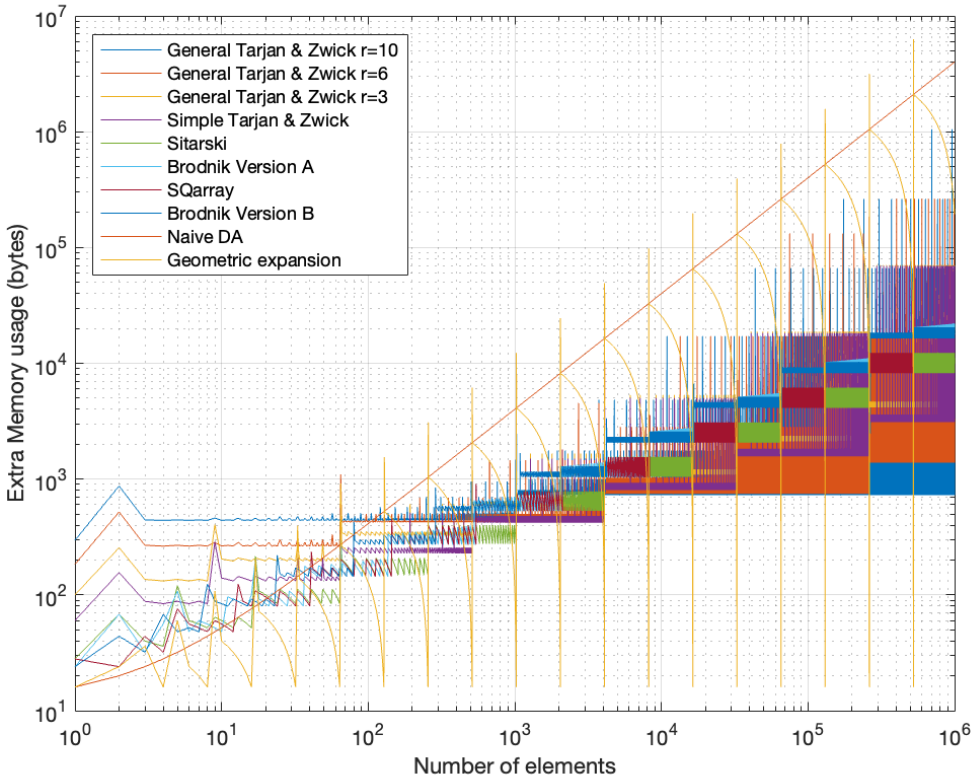


Figure 5.1: Double log plot of excess memory of all solutions combined.

every *Grow* operation. Thus we see a linear memory profile as the amount of extra space increases by one integer or 4 bytes for every element inserted. The geometric expansion solution temporarily uses approximately $2N$ excess space during rebuilds since it allocates a new array of size $2N$ while still having to copy from the old array of size N. Between rebuilds, the excess space goes down linearly as the array allocated gradually is filled. This is seen as an exponential fall in the double logarithmic plot in figure 5.1.

To further inspect the other solutions' memory profiles and highlight the differences between the solutions, figure 5.2 is zoomed in on a particular region in a regular plot. Figure 5.2 shows that Brodnik's data structures and the SQarray's excess memory profile are very similar. Brodnik's version B and SQarray's excess memory profiles are identical in the zoomed-in range.

For the general version of Tarjan and Zwick's solution, it is seen that a higher r value does indeed correspond to lower excess memory while not rebuilding. In fact, the version with $r = 10$ achieves around 1kB of overhead memory for arrays containing just over half a million integers, as seen in figure 5.2. The simple version of Tarjan and Zwick behaves similarly to the general version with $r = 3$ but consistently uses a bit less memory. These two solutions are exactly the same except that the index block in the simple version has size B, whereas it has size $2B$ in the general version with $r = 3$, thus leading to a slight increase in excess space. Additionally, the version with $r = 3$ has the top-level array with pointers to the index arrays, leading to a further increase in excess space.

Finally, Sitarski's HAT data structure uses considerably less memory than all other solutions tested except for the data structures by Tarjan and Zwick. However, Sitarski's

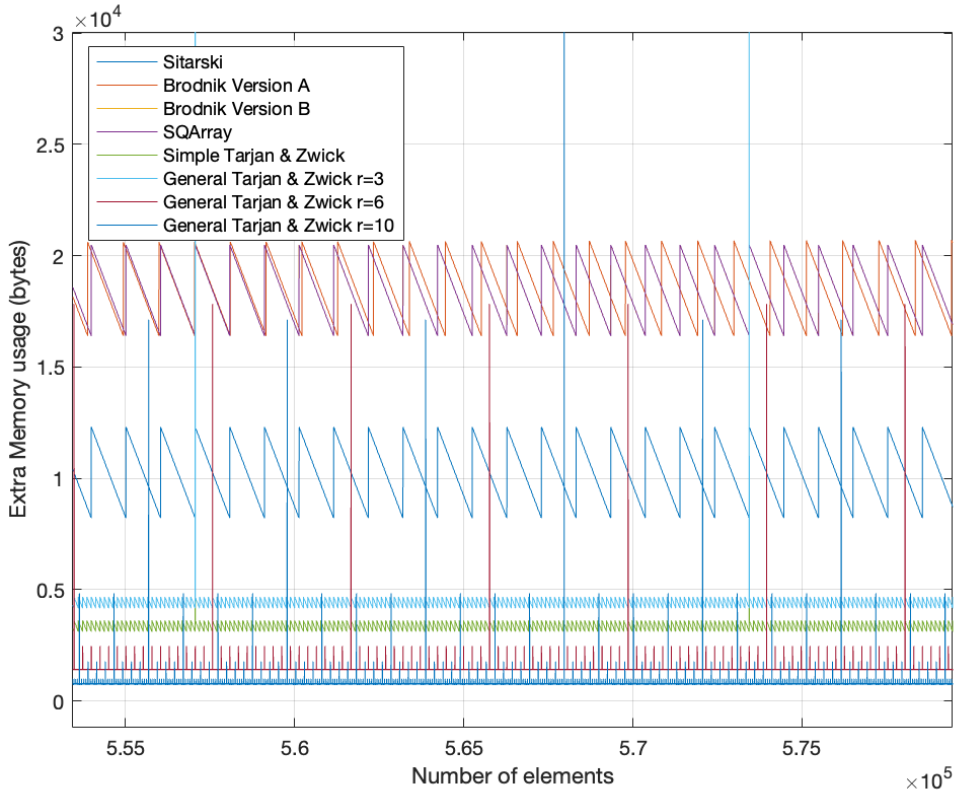


Figure 5.2: Excess memory of all solutions in a zoomed plot.

solution has fewer spikes as it does not merge blocks but only rebuilds with B doubled when $N = B^2$. This does not lead to very large spikes, as only an additional $2B$ overhead is required to copy two blocks of size B into one new block of size $2B$. It is also seen in figure 5.3, where the simple data structure by Tarjan and Zwick, Sitarski, SQarray and Brodnik's version B are compared. Of the $\mathcal{O}(\sqrt{N})$ excess memory solutions, Sitarski's solution uses the least amount of excess memory to store the dynamic array.

Figure 5.4 shows the general version of Tarjan and Zwick's solution in a double logarithmic plot for $r = 3$, $r = 6$ and $r = 10$. Similarly to figure 5.2, figure 5.4 shows that larger r values correspond to less excess memory needed to store the array. This is, however, not true for very small values of N since there are more non-utilized index blocks. There are, of course, more optimized solutions for small arrays. Figure 5.4 also shows the spikes from rebuilding the data structures and combining blocks. The spikes for the solutions with $r = 6$ and $r = 10$ have larger spikes than that with $r = 3$. We do not see much larger excess memory spikes for $r = 10$ because the data structure does not contain enough elements to utilize the larger data blocks. For example, with $r = 10$, the largest data block has size B^9, much larger than the 10^6 elements inserted.

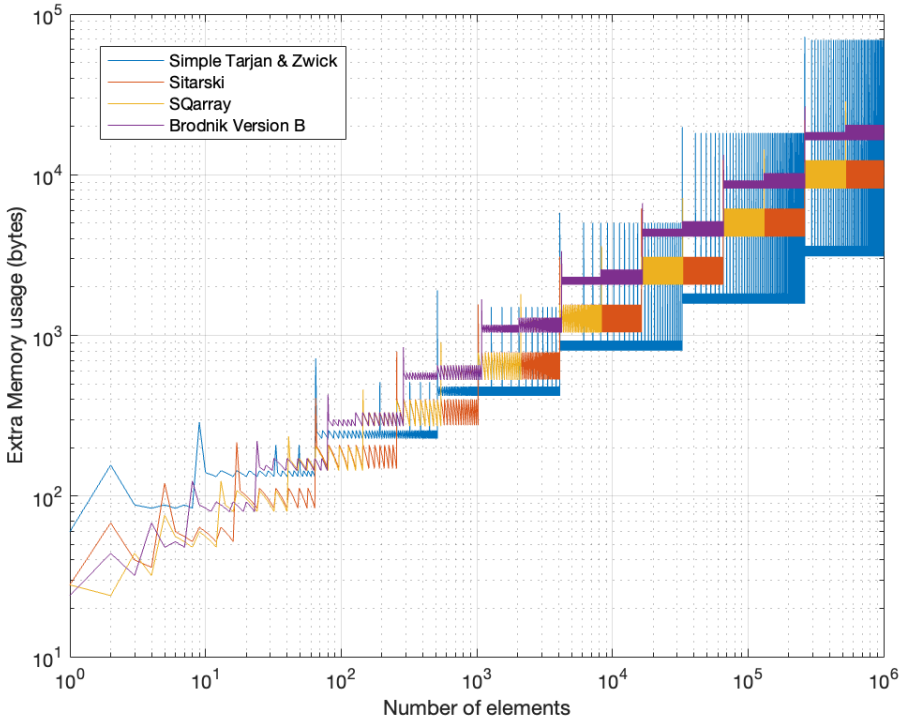


Figure 5.3: Double logarithmic plot of excess memory of simple version by Tarjan and Zwick, Sitarski's HAT, Brodnik's version B and SQarray.

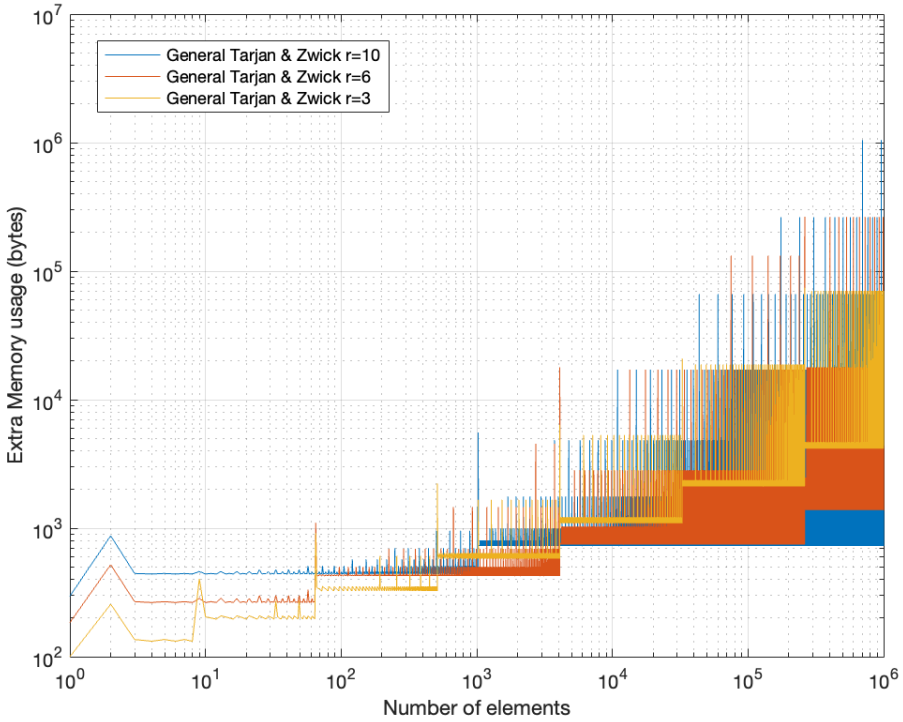


Figure 5.4: Double log plot of excess memory of the general solution by Tarjan and Zwick for $r = 3$, $r = 6$ and $r = 10$.

5.2 Insertion time

To perform run-time tests of the *Grow* operation for the different data structures, we inserted 1 million elements into an initially empty data structure. We recorded the cumulative time spent after calling the *Grow* function, i.e. the total time spent since the first insertion in milliseconds. This was done 5 times for each data structure, and each timestamp is averaged over these 5 runs. Figure 5.5 shows the test results of all previous solutions discussed in section 3. All other solutions except for the naive dynamic array are barely visible in comparison to it. This is in accordance with the $\mathcal{O}(N)$ insertion time for the naive dynamic array, as discussed in section 3.1.

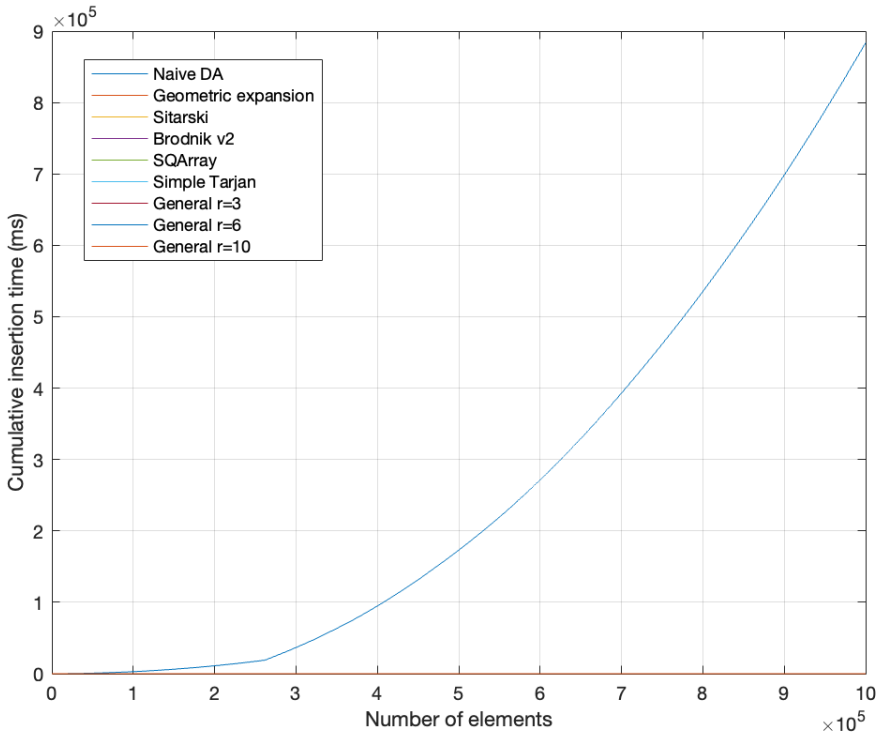


Figure 5.5: Cumulative insertion time of the previous data structures.

To inspect the behaviour of the other data structures, figure 5.6 shows the same results but without the naive dynamic array. Figure 5.6 shows that the cumulative insertion time of the remaining solutions lies very close to each other, especially when not considering the data structures of Tarjan and Zwick. The geometric expansion solution seems to be performing best, followed by Sitarski's HAT data structure, although the insertion time is very quick, and the differences between the solutions are minor.

To investigate the effect of varying the r value in the general version of Tarjan and Zwick's solution, figure 5.7 shows the test results for $r = 3$ to $r = 10$. Figure 5.7 shows an increasing cumulative insertion time as r increases. Especially, the data structure with $r = 10$ takes a significantly longer time to grow the resizable array by one million elements. This is expected as the *Grow* operation has amortized $\mathcal{O}(r)$ insertion time.

Table 5.1 shows the time to insert one element averaged over inserting one million elements. Again, the solutions except the naive dynamic array have similar average insertion times, with a slight increase for larger r values in the general version by Tarjan and Zwick.

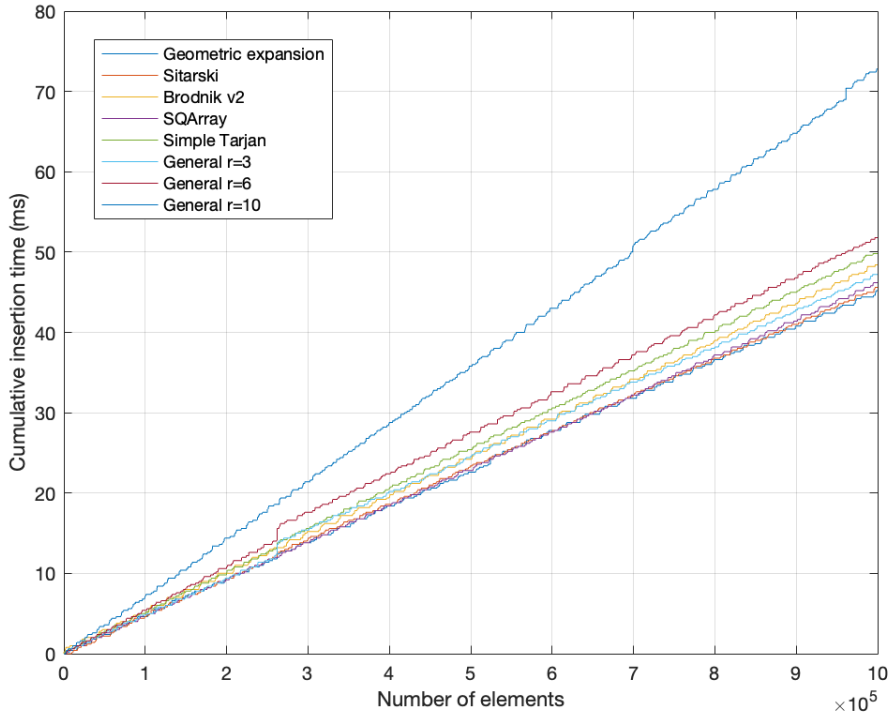


Figure 5.6: Cumulative insertion time of the previous data structures without the naive dynamic array.

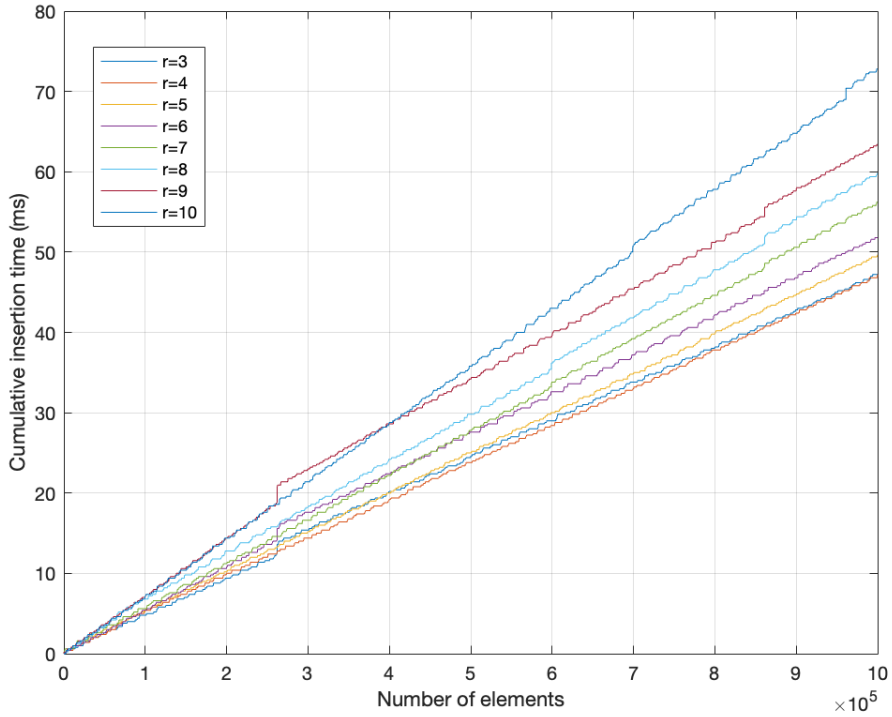


Figure 5.7: Cumulative insertion time of the general version of Tarjan and Zwick's data structure.

Data structure	Average Insertion time
Naive	0.8835 ms
Geometric Expansion	4.52e^{-5} ms
Sitarski	4.56e^{-5} ms
Brodnik v2	4.84e^{-5} ms
SQArray	4.62e^{-5} ms
Simple Tarjan and Zwick	4.98e^{-5} ms
General r=3	4.72e^{-5} ms
General r=6	5.18e^{-5} ms
General r=10	7.28e^{-5} ms

Table 5.1: Table of average insertion times averaged over 1 million insertions.

5.3 Access time

When testing the access time of the different resizable array data structures it is important to test across a wide spectrum of array sizes and access at various indices. The test was set up such that access time was tested as the array grew in size allowing it to be tested across various array sizes. Due to access being a generally fast operation and in order to get a readable time measurement, the time measurements are the sum of 10 million access operations. A time measurement was made for every 100 elements inserted up to array sizes of 10^5. Since we are comparing the access tests across several data structures, the same sequence of access operations has to be made across the different data structures. As such, a random seed was set and a predefined list of 10 million random indices in the interval $[0, A.Length() - 1]$ was made prior to each time measurement using the C++ *rand()* function. All resizable arrays discussed in section 3 were tested using this method and compared. The plotted access times can be seen in figure 5.8.

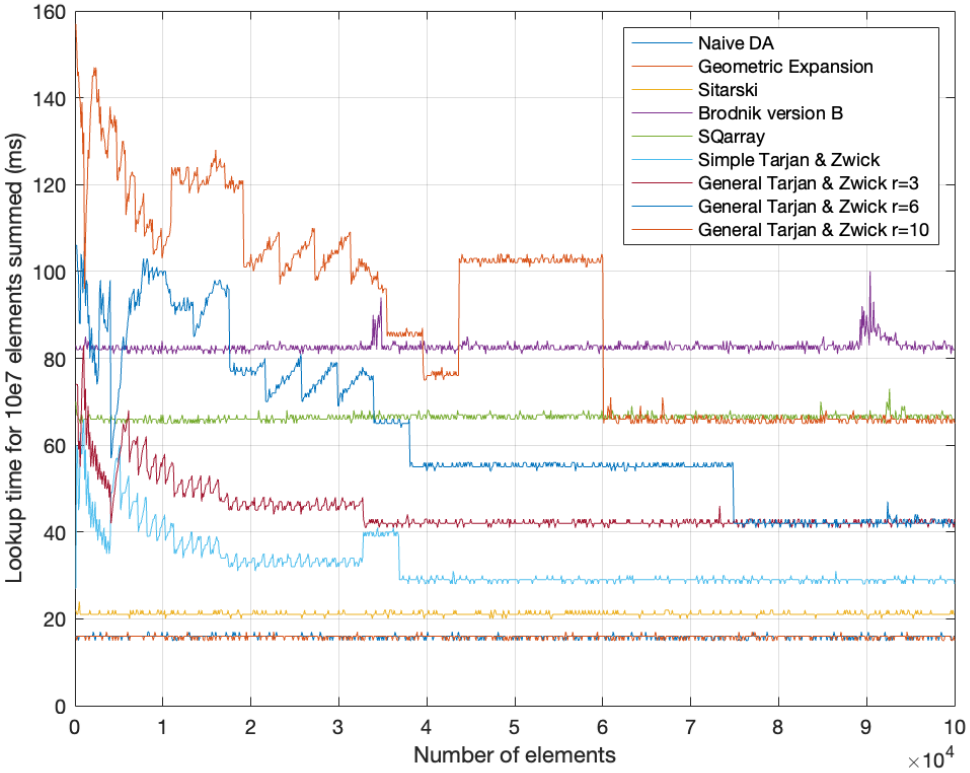


Figure 5.8: Times to perform 10 million lookups for the previous solutions.

As expected, the basic data structures utilizing the naive and geometric expansion techniques, are the fastest as they are just performing a fixed-sized array lookup. Sitarski's HAT solution also performs fast lookups, 2.1 nanoseconds on average per lookup compared to 1.6 nanoseconds on average per lookup for the basic data structures as seen in table 5.2. The simple $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ data structure also performs fast access operations with 3.2 nanoseconds on average per lookup, yet still takes twice as long as the basic data structures.

Interestingly, Brodnik's solution and SQarray perform quite a bit worse in terms of access time compared to HAT, despite achieving the same bounds. They are respectively about 4 and 3 times slower at performing access operations as compared to Sitarski's solution

Data structure	Access time
Naive	15.879 ms
Geometric Expansion	15.832 ms
Sitarski	21.126 ms
Brodnik Version B	82.699 ms
SQarray	66.434 ms
Simple Tarjan and Zwick	32.320 ms
General r=3	44.731 ms
General r=6	62.577 ms
General r=10	89.991 ms

Table 5.2: Time to perform 10 million lookups.

with Brodnik using 8.2 nanoseconds per lookup and SQArray about 6.6 nanoseconds.

Figure 5.9 shows the lookup times for performing 10^7 random access operations using the general version of Tarjan and Zwick's solution for $r = 3$ to $r = 10$. As described in section 3.6.2, the access time is $\mathcal{O}(r)$, so we should expect to see higher lookup times for larger values of r. The results in figure 5.9 seem to systematically vary for the first number of elements inserted and then flatten out to and stabilize at a particular level. The initial variance could be due to the compiler finding optimizations which eventually stabilize to the fastest performance. The results do, however, show that the data structures with larger r performs slower than those with lower values of r. This is especially seen in the first part with a large variance. It is, however, a bit odd that they stabilize into three distinct levels. The access times of $r = 3$, $r = 6$ and $r = 10$ are also plotted with the other previous solutions in figure 5.8 for comparison. Here $r = 3$ uses 4.4 nanoseconds per lookup, $r = 6$ uses 6.3 nanoseconds and $r = 10$ uses 9 nanoseconds per lookup. Even for $r = 10$, it is a comparable lookup time to that of SQarray and version B by Brodnik.

The constant time access general $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$ solution was also tested and compared to that of the $\mathcal{O}(r)$ access time solution. The solution was tested like the other solutions and tested for $r = 3, \ldots, 10$. The results of the test can be seen in figure 5.10. The lookup times have a higher variance for about the first third of the inserted elements, much like we saw for the $\mathcal{O}(r)$ access time solution in figure 5.9. Contrary to the $\mathcal{O}(r)$ access time solution, the constant time lookup version does not flatten into three distinct levels. However, we see some odd systematic patterns.

In general, the constant time access is much slower than the $\mathcal{O}(r)$ time access due to the small values of r. For the $\mathcal{O}(r)$ solution, with $r = 3$, we see a 4.5 nanosecond access time versus an 11.5 nanosecond access time for the $\mathcal{O}(1)$ solution. Likewise, for $r = 6$, an average of 6.3 nanosecond lookup versus an 13.6 nanosecond access time for the $\mathcal{O}(r)$ and constant time solutions respectively. For $r = 10$, we see average lookup times of 9 nanosecond and 17.3 nanosecond respectively. These times can also be seen in table 5.3. Interestingly, the constant time access solution also increases for larger values of r, where this should be expected to be the same for all values of r. This could also be influenced by the high variance and odd systematic patterns we see in figure 5.10.

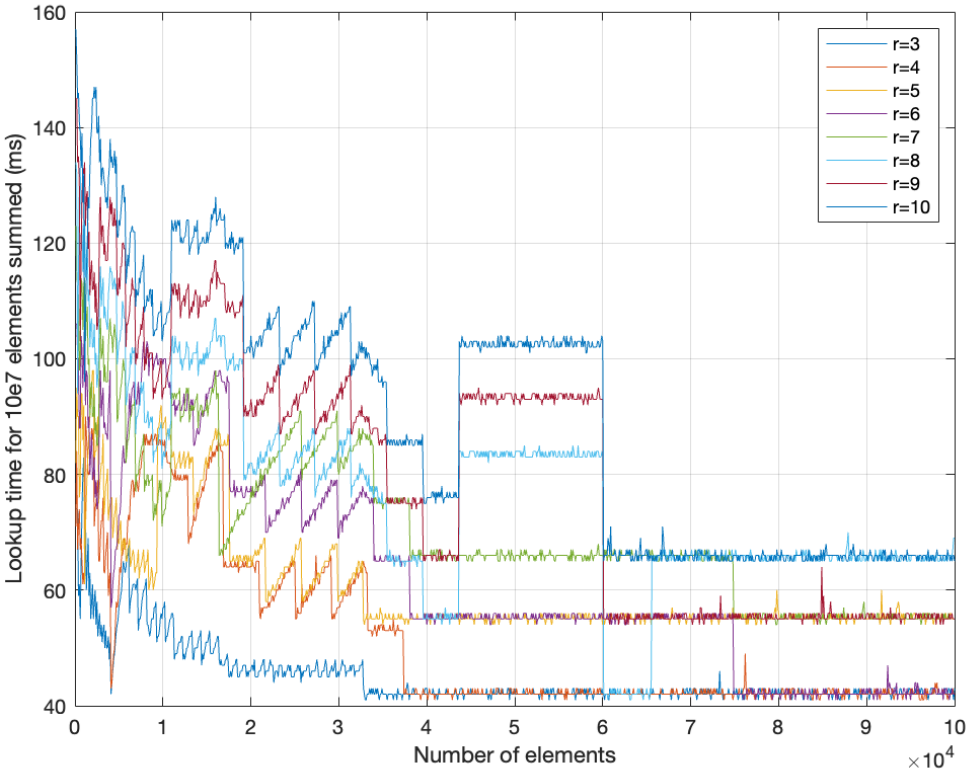


Figure 5.9: Times to perform 10 million lookups for the general version of Tarjan and Zwicks data structure for $r = 3$ to $r = 10$.

Data structure	Access time
General r=3 $\mathcal{O}(1)$ access	114.781 ms
General r=6 $\mathcal{O}(1)$ access	135.840 ms
General r=10 $\mathcal{O}(1)$ access	173.085 ms
General r=3	44.731 ms
General r=6	62.577 ms
General r=10	89.991 ms

Table 5.3: Time to perform 10 million lookups for the general version of Tarjan and Zwick's data structure with $\mathcal{O}(r)$ and $\mathcal{O}(1)$ access time.

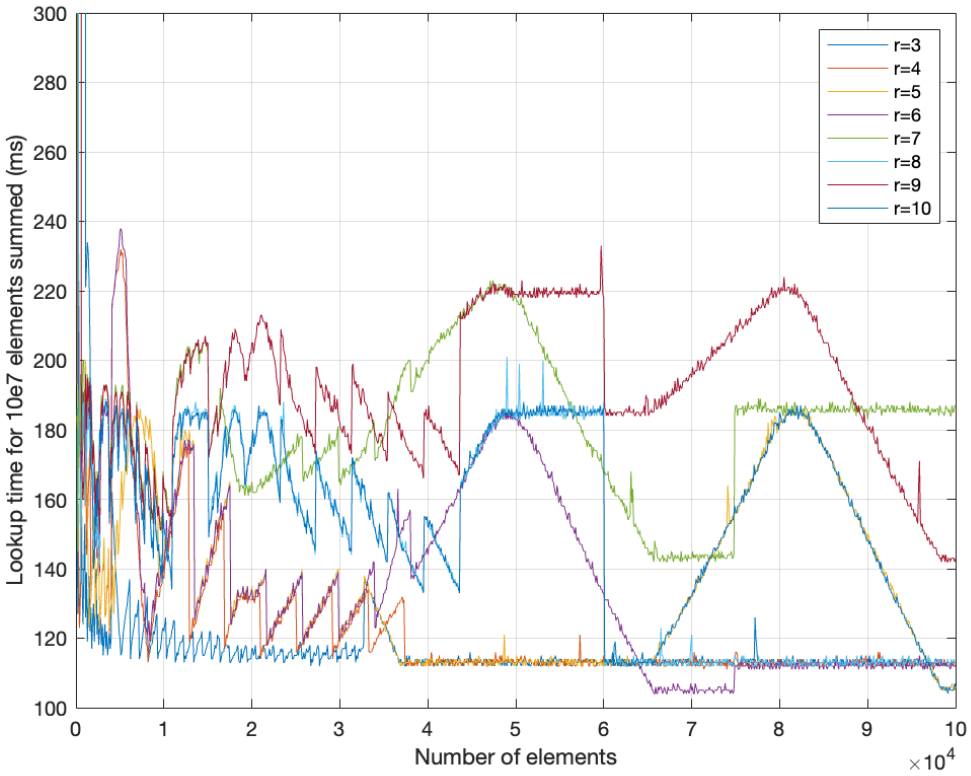


Figure 5.10: Times to perform 10 million lookups for the constant time access version of Tarjan and Zwick's general solution.

6 New solutions

This section explores new solutions and extensions to previous solutions explored in section 3 based on the results shown in section 5. Designing new solutions for a space-efficient dynamic array is a very constrained problem with little room for algorithmic creativity. To sketch up some of these constrains, it was shown by Tarjan and Zwick that for any $r \geq 2$, it holds that $N + \mathcal{O}(N^{\frac{1}{r}})$ is sufficient to store the dynamic array. It further holds that $N + \mathcal{O}(N^{1-\frac{1}{r}})$ space is used occasionally during *Grow* and *Shrink* operations [1]. In other words, we cannot create a new solution with smaller excess memory to store the array without sacrificing the size of the temporary memory occasionally used by the data structure.

Furthermore, it was also shown by Tarjan and Zwick that any dynamic array data structure using $N+\mathcal{O}(N^{\frac{1}{r}})$ space to store the array has an amortized *Grow* cost of $\Omega(r)$ [1]. In other words, reducing the excess memory of such a data structure increases the insertion times asymptotically. This also says we must have some constant r to get amortized constant *Grow* operations.

Because of the constraints described above, we realised that we should not strive to improve the theoretical bounds of existing solutions. Instead, we looked at the previous solutions that performed best during our tests in section 5. With knowledge about the data structures, the aim is to tweak these to improve performance in practice while keeping the theoretical bounds. Since the dynamic array problem is so constrained but with such wide practical use, even small optimizations can have a profound impact.

6.1 Fast Combine

The idea for the Fast Combine data structure came from trying to make the *CombineBlocks* operation faster in the $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ version of Tarjan and Zwick's solution. As mentioned in section 3.6.1, when B data blocks of size B are full, they are combined into one larger block of size B^2. This operation requires copying B^2 elements. The idea behind Fast Combine is to eliminate the need to combine blocks entirely, thus avoiding the need to copy the elements. Not only does this remove the time required to copy the elements, but it also reduces the temporary excess memory required while combining blocks. This can be achieved by changing the length of the right index block A_1 to B instead of $2B$. Furthermore, instead of pointing to data blocks of size B^2 from the left index block A_2, add an additional layer of index blocks of size B in between as seen in figure 6.1. As such, the left index block A_1 and its data blocks contain B^2 elements when full and are identical to the subtrees pointed to by A_2. When the A_1 structure is full, it is possible to add a pointer from A_2 to A_1 and allocate a new empty A_1. Similarly, when the A_1 structure is empty, corresponding to the $SplitBlocks$ operation in Tarjan and Zwick's solution, the empty A_1 can be deleted and set to the rightmost index block pointed to by A_2. These operations are just a matter of adding or deleting a pointer and allocating or de-allocating a memory block, both of which are assumed to be constant time operations.

Since the Fast Combine data structure is largely based on the $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ version of Tarjan and Zwick's solution, the logic and the pseudocode of the *Grow* and *Shrink* operations are exactly the same and can be seen in algorithm 1 and algorithm 2 found in section 4.1. However, the *CombineBlocks*, *SplitBlocks* and *Rebuild* operations are different. The pseudocode for *CombineBlocks* and *SplitBlocks* can be seen in algorithm 12

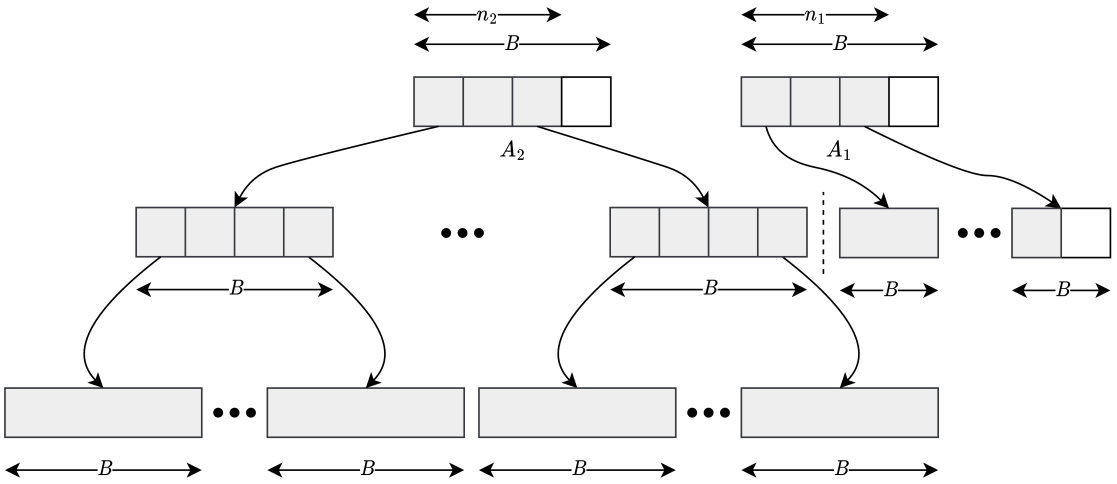


Figure 6.1: Sketch of the Fast Combine data structure.

and 13, respectively. The *Rebuild* is done similarly as the $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ solution by Tarjan and Zwick, and is achieved by filling A_2 up as much as possible and then inserting the remaining into A_1.

Algorithm 12 *CombineBlocks*

1: $A_2[n_2] \leftarrow A_1$
2: $A_1 \leftarrow Allocate(B)$
3: $n_1 = 0$
4: $n_2 = n_2 + 1$

Algorithm 13 *SplitBlocks*

1: $A_1 \leftarrow Deallocate(B)$
2: $A_1 \leftarrow A_2[n_2 - 1]$
3: $A_2[n_2 - 1] \leftarrow Deallocate(B)$
4: $n_1 = 0$
5: $n_2 = n_2 - 1$

Compared to the *CombineBlocks* and *SplitBlocks* used in the $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ version of Tarjan and Zwick's solution described in algorithm 3 and algorithm 4 respectively, it is easy to see that both of these are much faster and $\mathcal{O}(1)$ time worst case.

A crucial disadvantage of this model is that we store up to B^2 pointers in the leftmost structure. Similar to Tarjan and Zwick's solution, $B = \mathcal{O}(N^{\frac{1}{3}})$. Thus storing $\mathcal{O}(B^2)$ gives an additional space of $\mathcal{O}(N^{\frac{2}{3}})$ which is asymptotically more than $\mathcal{O}(N^{\frac{1}{3}})$ achieved by Tarjan and Zwick. However, Tarjan and Zwick's solution requires $\mathcal{O}(N^{\frac{2}{3}})$ additional space when rebuilding since at least one additional block of size B^2 must be stored while copying. Since the maximal size block in the Fast Combine data structure is B, only $\mathcal{O}(N^{\frac{1}{3}})$ temporary excess space is required. Thus, this data structure achieves $\mathcal{O}((N^{2/3}), \mathcal{O}(N^{1/3}))$ memory bounds which is the opposite of Tarjan and Zwick's solution with $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ memory bounds.

Another downside of this solution is that in the *Get* method if the element lies in the leftmost structure, it is necessary to follow two pointers. As the left structure has two layers, it also requires a bit more calculation to find the correct index in the corresponding data block. Specifically, if $i < n_2 B^2$, the element must be located in the leftmost structure and can be found as $A_2[i/B^2][(i \bmod B^2)/B][i \bmod B]$. In practice, this can be implemented fast using bit manipulation as $A_2[i >> (logB + logB)][(i \& ((B << logB) - 1)) >> logB][i \& (B - 1)]$. log$(B)$ can stored as a variable and updated during *Grow* and *Shrink* operations simply by incrementing or decrementing by one since B is a power of 2. If $i \geq n_2 B^2$,

the element must lie in the right-most structure and can be accessed as follows: $A_1[(i - n_2B^2)/B][i \bmod B]$. This can, in a similar fashion, be implemented efficiently using bit manipulation as $A_1[(i - (n_2 << (logB + logB)) >> logB][i\&(B-1)]$.

We tested the Fast Combine similarly to the data structures described in section 3. Figure 6.2 shows our memory test plotted together with the $\mathcal{O}(N^{1/3}), \mathcal{O}((N^{2/3}))$ data structure by Tarjan and Zwick. We see that Fast Combine consistently uses significantly more excess memory, and the gap is growing with N, which was expected as the asymptotic bound is $\mathcal{O}(N^{2/3})$ versus $\mathcal{O}(N^{1/3})$ from Tarjan and Zwick's solution. On the other hand, the asymptotic bound for temporary memory spikes is $\mathcal{O}(N^{1/3})$. We see very small memory spikes in practice relative to the consistent excess memory use.

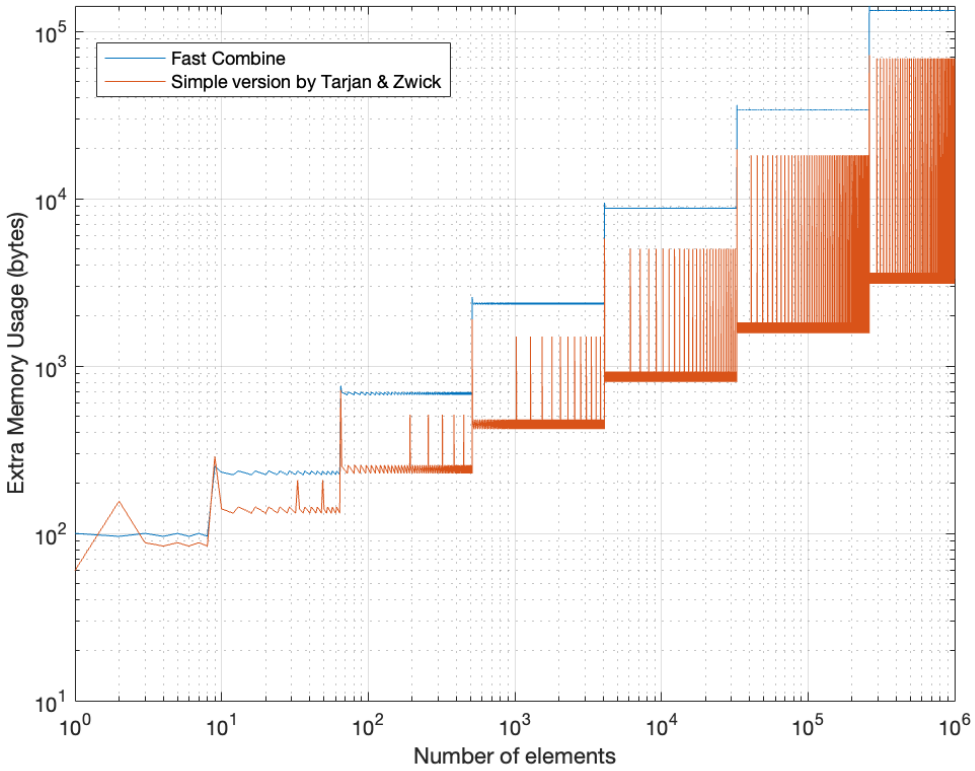


Figure 6.2: Double logarithmic plot of excess memory of Fast Combine and the $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ data structure by Tarjan and Zwick.

Figure 6.3 shows the cumulative lookup time for 10 million *Get* operations performed for every 100 insertions. Figure 6.3 shows that the lookup time is worse compared to that of, for example, the $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ version of Tarjan and Zwick's solution. This is due to the deeper data structure that must be traversed to find the wanted element. More specifically, two pointers have to be followed instead of one.

The Fast Combine solution's strength is its fast insertion and deletions. However, our tests showed an average insertion time of $5.02e^{-5}$ milliseconds which is interestingly slightly more than the average insertion time of Tarjan and Zwick's $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ solution which had an average insertion time of $4.98e^{-5}$ milliseconds. This could be due to some operations during *Rebuild* that differ and take longer. However, this shows that performing fast *CombineBlocks* does not give much in terms of the overall insertion time. The biggest

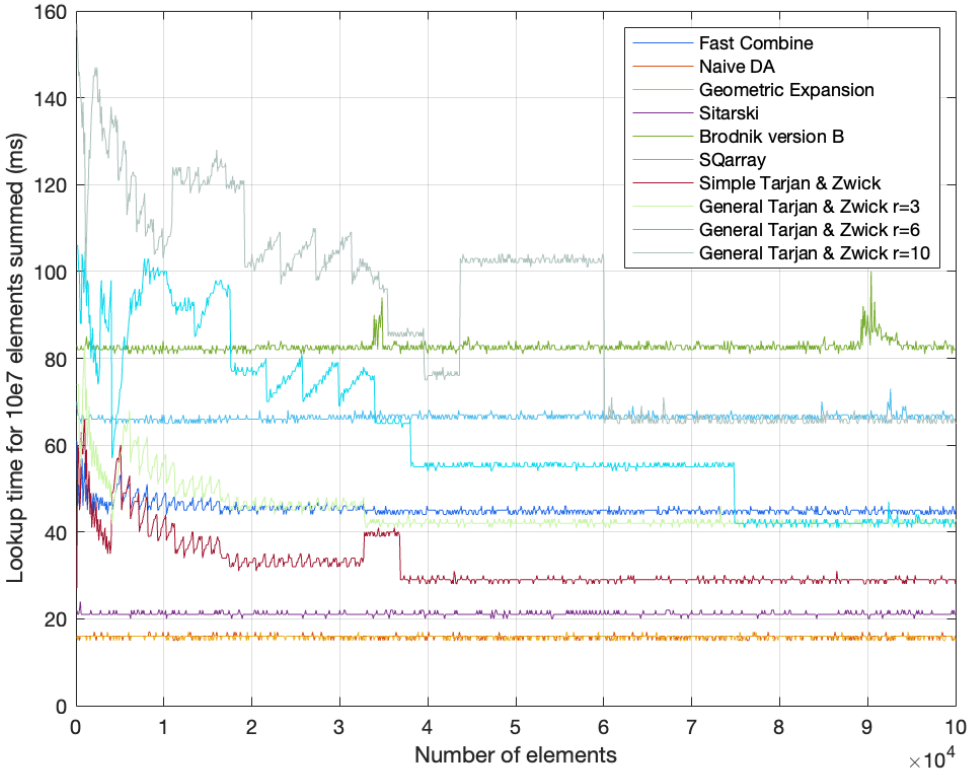


Figure 6.3: Times to perform 10 million lookups for the previous solutions and Fast Combine.

flaw is, however, its memory overhead, leaving this solution far from optimal. While it was an interesting idea, this solution will not be considered further.

6.2 Hybrid Sitarski

After looking at the test results of the previous solutions to the resizable array problem, it became clear that Sitarski's HAT data structure is the most favourable in terms of looking at space overhead, access time and insertion time. As such, we wanted to see if it was possible to tweak the solution in any way in order to improve the data structure. Looking at Sitarski's solution, memory overhead comes from two sources: the index array and the last block, which may be partially full. With everything else being the same size, reducing the size of the index block thus reduces the memory overhead but requires more frequent rebuilds since the data structure reaches capacity faster. For example, halving or doubling the length of the index array directly implies that the capacity of the data structure is halved or doubled, respectively. When the indexed array is halved or doubled, rebuilding the data structure happens when $N = \frac{B^2}{2}$ or $N = 2B^2$ respectively instead of $N = B^2$ as in Sitarski's HAT data structure.
To test the effect that halving or doubling the index array has on excess space, we implemented these and compared them to the HAT solution. This is seen in the double logarithmic plots in figure 6.4. We have labelled the solution with the index array halved as Sitarski Half and the solution with the index array doubled as Sitarski Double.

From figure 6.4, we see that both solutions alternate with Sitarski's HAT data structure in terms of having the lowest excess memory. However, Sitarski Half uses strictly less

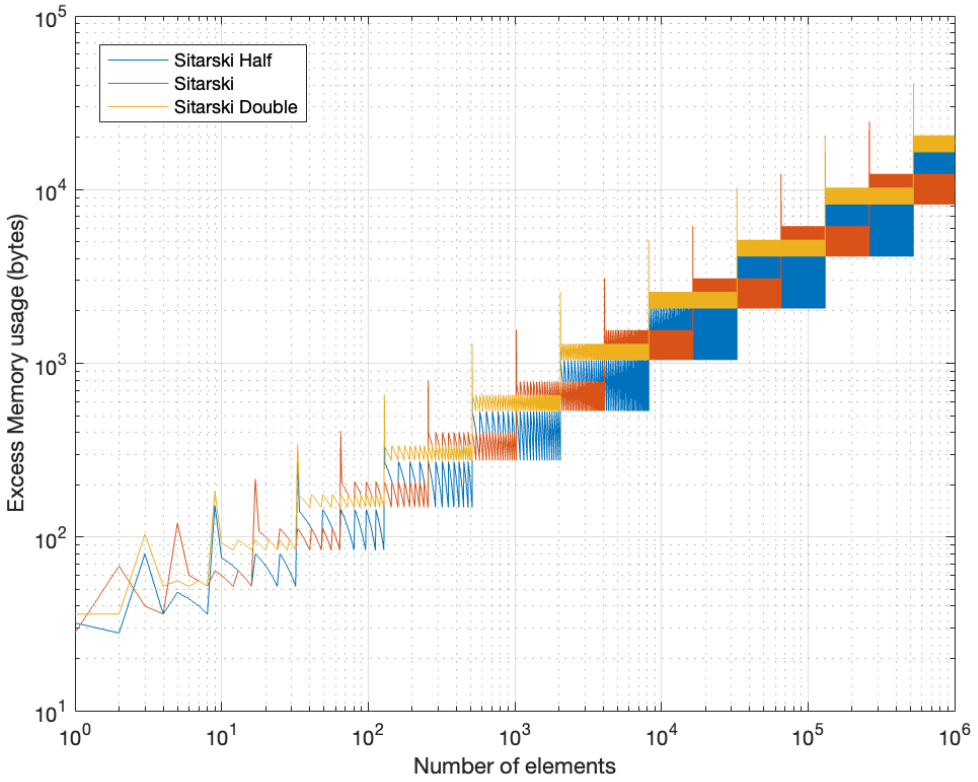


Figure 6.4: Double logarithmic plot of Sitarski's HAT data structure with $\frac{B}{2}$ index block size and $2B$ index block size.

excess memory than Sitarski Double. Thus, we will not investigate Sitarski Double any further.

Due to the alternating lowest excess memory between Sitarski's HAT and Sitarski Half, naturally, we thought about combining the two solutions to always achieve the lowest excess memory possible for any array size. A sketch of such a solution is seen in figure 6.5. This solution we have labelled Hybrid Sitarski.

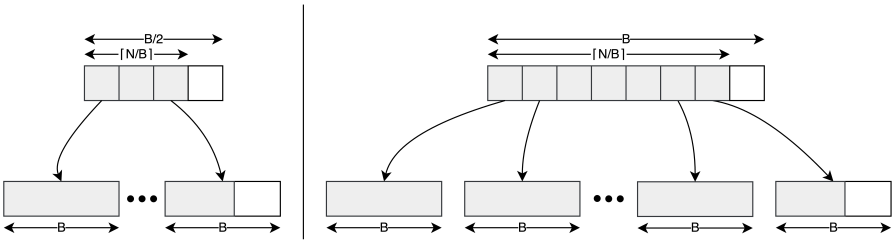


Figure 6.5: Sketch of the Hybrid Sitarski data structure.

The idea is to sometimes behave like the HAT data structure when it is most beneficial and sometimes behave like Sitarski Half when it is most beneficial. This can be achieved by starting with the Sitarski Half solution, and when it is full, instead of rebuilding it with B doubled, simply double the length of the index array, effectively transforming the data structure to the Sitarski's HAT data structure. As only the index block is rebuilt, we call this an index block rebuild. When the standard HAT structure is full, rebuild the structure back

to the Sitarski Half structure with B doubled. Since the index block already has size B, there is no need to rebuild it, thus, we are only rebuilding the data blocks. This is referred to as a data block rebuild.

Regarding the *Shrink* operation, we started by comparing the excess memory of Sitarski's HAT solution to the Sitarski Half solution. Figure 6.6 shows the excess memory of the two solutions when performing 10^6 *Shrink* operations with both data structures initially containing 10^6 elements in a double logarithmic plot. Note that the x-axis is "flipped", meaning that chronologically, we go from 10^6 elements down to 0, but here it is shown in increasing order such that it matches the monotonically increasing property of the log function applied to the plot. From the results shown in figure 6.6, Sitarski Half seems to

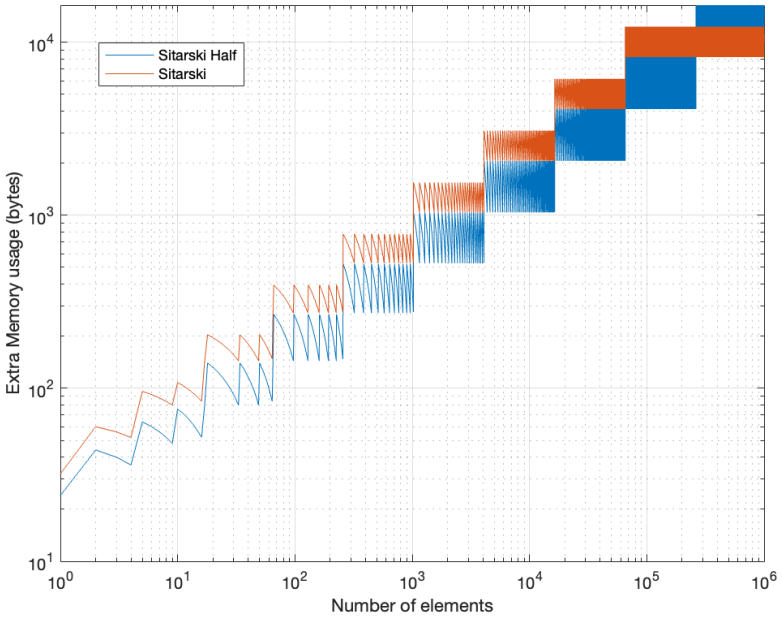


Figure 6.6: Double logarithmic plot of excess memory of Sitarski's HAT data structure and Sitarski Half during consecutive calls of *Shrink*.

perform strictly better than the original Sitarski solution except before the first rebuild. We know from our tests of the *Grow* operation seen in figure 6.4 that while growing the data structure, the two versions are altering with regards to which performs best in terms of excess memory. Thus, since we start this test with 10^6 elements, the HAT data structure is coincidentally better here before the first rebuild. Intuitively, it makes sense that if we are only shrinking the data structure, there is never a need for having an index block larger than $\frac{B}{2}$. This is because we rebuild when $N = (\frac{B}{4})^2$ and $(\frac{2B}{4})^2 = \frac{B^2}{4}$. Thus only $\frac{1}{4}$-th of the data blocks are full after rebuilding the data structure during a *Shrink* operation. Hence an index block of size $\frac{B}{4}$ is sufficient in this case. However, we do wish to design a data structure which handles both *Grow* and *Shrink* operations well.

In a similar way to how the *Rebuild* during the *Grow* operation is split into rebuilding the index block and data blocks individually, the same is done for *Shrink* but with one key difference. When $N = (\frac{B}{4})^2$ during a *Shrink* operation, a data block rebuild is triggered, just as in the standard HAT solution. However, it also rebuilds the index block to have size $\frac{B}{2}$ with the new B that was just halved. This has two purposes: 1) it decreases the memory overhead, as we have just seen that the data structure with less than $\frac{B^2}{2}$ ele-

ments performs better when the index block has size $\frac{B}{2}$. 2) it simplifies the data structure significantly as both *Grow* and *Shrink* data block rebuilds leave the index block at size $\frac{B}{2}$ and data blocks of size B. Furthermore, if a *Shrink* operation is called when $N = \frac{B^2}{4}$ and the length of the index array is B, the index block is rebuilt to size $\frac{B}{2}$. Note that it is necessary to check that the length of the index array is B as there are scenarios where we hit $N = \frac{B^2}{4}$, but the index is already of size $\frac{B}{2}$. An example is if we only shrink the data structure after a *Grow* data block rebuild. The purpose of the index block rebuild is to regulate the index block back to size $\frac{B}{2}$ when we no longer need a full-sized index block. It is essentially reverting the index block rebuild happening during *Grow* operations.

The pseudocode for *Grow* and *Shrink* operations is shown in algorithm 14 and 15, respectively. Note that rebuilding the data structure when $N = B^2$ is fairly simple and is included directly in the *Grow* operation and similarly for *Shrink* when $N = \frac{B^2}{4}$. The $Copy(A, x, B, y, l)$ function copies l elements from the array A starting at index x into the array B starting at the index y. Hybrid Sitarski only differs from Sitarski's HAT solution during rebuilding in which *Grow* is split up into two checks, namely on line 1 and 11. We also do not need to allocate a new index block during the data blocks rebuild as this will have the same size and can therefore be reused. Similarly, during *Shrink*, we split the rebuilding into two checks happening on lines 1 and 12. One problem with *Shrink* is that during the data block rebuild, it is necessary to allocate a whole new index block also, giving larger spikes in excess memory. The reason is that we have to split one data block into two of half size. However, since there are two of them, the second data block will overwrite a data block in the index block, which has not yet been split. Thus it is necessary to fill up a temporary index block as seen in line 3 to 8.

Algorithm 14 *Grow(a)*

1: **if** $N = B^2$ **then**
2: $B \leftarrow B \cdot 2$
3: **for** $i = 0$ to $\frac{B}{2}$ **do**
4: $A_{temp} \leftarrow Allocate(B)$
5: $Copy(A[2i], 0, A_{temp}, 0, \frac{B}{2})$
6: $Copy(A[2i+1], \frac{B}{2}, A_{temp}, 0, \frac{B}{2})$
7: $Deallocate(A[2i])$
8: $Deallocate(A[2i+1])$
9: $A[i] \leftarrow A_{temp}$
10: **end for**
11: **else if** $N = \frac{B^2}{2}$ & $len(A) = \frac{B}{2}$ **then**
12: $A_{temp} \leftarrow Allocate(B)$
13: $Copy(A, 0, A_{temp}, 0, \frac{B}{2})$
14: $Deallocate(A)$
15: $A \leftarrow A_{temp}$
16: **end if**
17: **if** $\lceil \frac{N}{B} \rceil \neq \lfloor \frac{N}{B} \rfloor$ || $N = 0$ **then**
18: $A[\lfloor \frac{N}{B} \rfloor][N \% B] \leftarrow a$
19: **else**
20: $A[\lceil \frac{N}{B} \rceil] \leftarrow Allocate(B)$
21: $A[\lceil \frac{N}{B} \rceil][0] \leftarrow a$
22: **end if**
23: $N \leftarrow N + 1$

Algorithm 15 *Shrink*

1: **if** $N = (\frac{B}{4})^2$ **then**
2: $B \leftarrow \frac{B}{2}$
3: $A_{temp} \leftarrow Allocate(B)$
4: **for** $i = 0$ to $\lceil \frac{N}{B} \rceil$ **do**
5: $A_{temp}[2i] \leftarrow Allocate(B)$
6: $A_{temp}[2i+1] \leftarrow Allocate(B)$
7: $Copy(A[i], 0, A_{temp}[2i], 0, B)$
8: $Copy(A[i], B, A_{temp}[2i+1], 0, B)$
9: **end for**
10: $Deallocate(A)$
11: $A \leftarrow A_{temp}$
12: **else if** $N = \frac{B^2}{4}$ & $len(A) = B$ **then**
13: $A_{temp} \leftarrow Allocate(\frac{B}{2})$
14: $Copy(A, 0, A_{temp}, 0, \frac{B}{2})$
15: $Deallocate(A)$
16: $A \leftarrow A_{temp}$
17: **else if** $N - 1 \% B = 0$ **then**
18: $Deallocate(A[\lceil \frac{N}{B} \rceil])$
19: **end if**
20: $N \leftarrow N - 1$

The Hybrid Sitarski solution was tested and compared to Sitarski's HAT and the Sitarski Half solution in a similar fashion to the tests done in section 5.1. Figure 6.7 shows the excess memory in a double logarithmic plot of the three different data structures while performing 10^6 *Grow* operations. It can be seen in figure 6.7 that the excess memory of

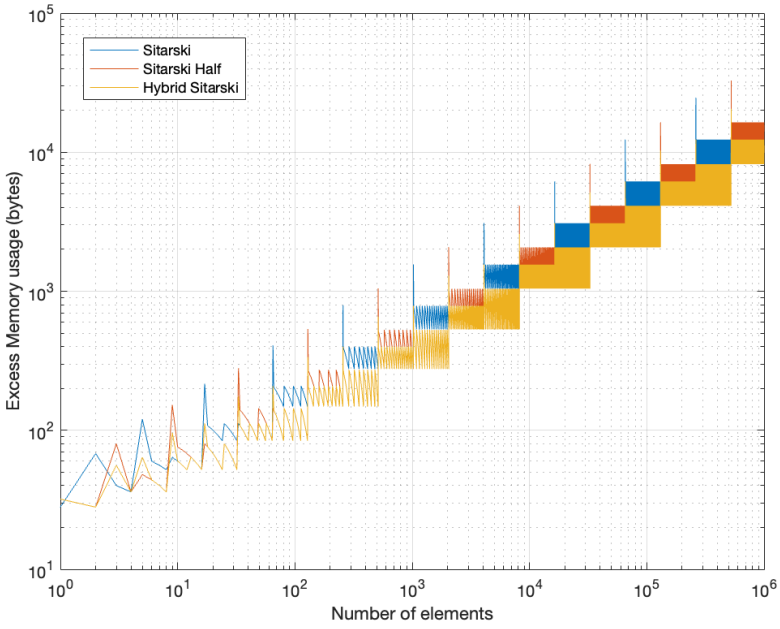


Figure 6.7: Double logarithmic plot of excess memory of the Hybrid Sitarski data structure in comparison to its component solutions.

Hybrid Sitarski lies below that of either of its component solutions except for the temporary excess memory used during the rebuilds shown by the spikes. The temporary excess memory spikes of the hybrid solution are smaller than the spikes of the corresponding component solution when rebuilding. The Hybrid Sitarski data structure does rebuild twice as often, namely both at $N = B^2$ and $N = \frac{B^2}{2}$. However, the reason we do not see large spikes is that this solution essentially splits the rebuilding process into two separate operations. At $N = \frac{B^2}{2}$ the index block is rebuilt, and the $\frac{B}{2}$ pointers in the index block have to be copied into a new index block of size B. This incurs temporary excess space of B times the size of the pointers. When $N = B^2$, only the data blocks and not the index block has to rebuilt. Note that this was already rebuilt to size B in the previous case. To rebuild a data block to size $2B$, the new block has to be allocated while the elements are copied into it. This causes a temporary excess memory of $2B$ times the size of the elements being copied, and this is why a larger spike is seen in this case. In Sitarski's HAT solution, larger spikes are seen since both the index array and the data blocks are rebuilt at the same time. Furthermore, the new index block has to exist while the data blocks are rebuilt, yielding $2B$ excess memory from the index array and $2B$ excess memory from the current data block being rebuilt. Thus, even though the Hybrid Sitarski solution spikes in memory twice as often, the spikes are lower, and the total excess memory for both spikes combined is smaller than the corresponding excess memory of either the HAT solution or the Sitarski Half solution.

Figure 6.8 shows the memory test of the three solutions in a double logarithmic plot. For the test, all solutions started with 10^6 elements and where after 10^6 *Shrink* operations were performed. Note that similar to in figure 6.6, the x-axis is "flipped" with regards to

the chronological order. Figure 6.8 shows that while only shrinking the data structure,

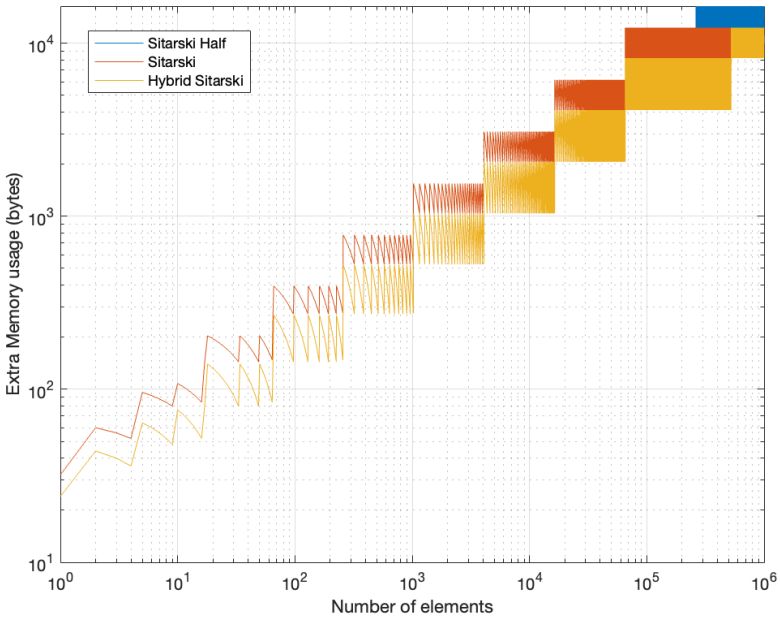


Figure 6.8: Double logarithmic plot of excess memory of the Sitarski's HAT, Sitarski Half and Hybrid Sitarski during consecutive calls of *Shrink*.

Hybrid Sitarski behaves exactly like the Sitarski Half data structure. This is expected since the index block is never rebuilt to size B. As mentioned earlier, after growing the data structure 10^6 times, Sitarski's HAT uses less excess memory. Hybrid Sitarski will, in this case, have an index block of size B, thus using the minimum amount of excess memory between its two component solutions.

So far, we have only tested sequences of consecutive *Grow* and *Shrink* operations. However, in practice, the solution should perform well for any sequence of *Grow* and *Shrink* operations. To get a glimpse of the performance during a varied sequence of operations, we performed a test where we initially inserted 10^4 elements. Then we created a sequence of 100 coin flips uniformly at random. For each coin flip, we then performed a sequence of 1000 consecutive *Grow* or *Shrink* operations depending on the outcome of the coin flip. The results of this test are shown in figure 6.9. Figure 6.9 shows that during this random sequence, the hybrid version performed at least as well as the standard HAT data structure, except for a few spikes when rebuilding the index block either at $N = \frac{B^2}{4}$ or $N = \frac{B^2}{2}$. These results hint that this is true for all sequences of *Grow* and *Shrink* operations, which we later prove in Lemma 3.

The Hybrid Sitarski data structure was tested in terms of access and insertion times similar to the previous data structures in section 3. Figure 6.10 shows the lookup time performed every 100-th insertion for 10 million lookups. Figure 6.10 shows that the lookup time of Hybrid Sitarski is on par with that of Sitarski's HAT data structure. Both have very fast access times compared to other solutions using $\mathcal{O}(\sqrt{N})$ excess memory to store the array.

In the following, properties of the Hybrid Sitarski data structure are formally proven.

Lemma 1. *The Hybrid Sitarski data structure uses at most $N + \mathcal{O}(\sqrt{N})$ memory to store the array and at most $N + \mathcal{O}(\sqrt{N})$ memory temporarily during a Grow or Shrink operation.*

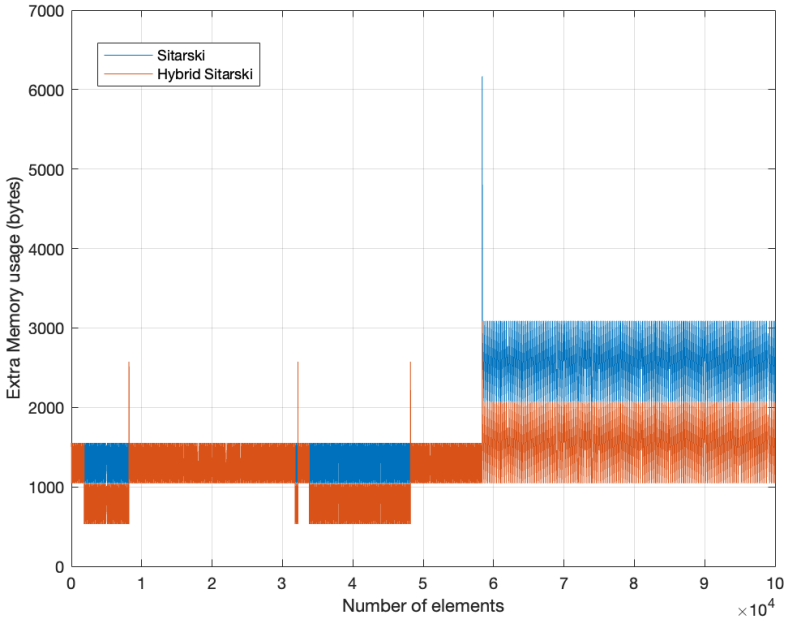


Figure 6.9: Plot of excess memory of Sitarski's HAT data structure and Hybrid Sitarski when performing a random sequence of *Grow* and *Shrink* operations on the data structures with 10^4 initial elements.

Proof. The Sitarski Half data structure has the same memory bounds as Sitarski's HAT data structure of $(\mathcal{O}(\sqrt{N}), \mathcal{O}(\sqrt{N}))$ since the only difference is that the index block has half the size which is still $\mathcal{O}(B) = \mathcal{O}(\sqrt{N})$. Because the Hybrid Sitarski data structure is always in a state of Sitarski's HAT or Sitarski Half, the excess memory bounds must necessarily be the same. □

Lemma 2. *The Grow and Shrink operations of the Hybrid Sitarski data structure have amortized $\mathcal{O}(1)$ run time.*

Proof. First, not considering the index block rebuilds at $N = \frac{B^2}{4}$ and $N = \frac{B^2}{2}$, the amortized cost of *Grow* and *Shrink* operations are $\mathcal{O}(1)$ for the exact same reasons as for Sitarski's HAT structure presented in section 3.3. Now including the index block rebuilds, performing such a rebuild takes $\mathcal{O}(B)$ time as the index block sizes are $\mathcal{O}(B)$. It is thus sufficient to argue that at least $\mathcal{O}(B)$ *Grow* or *Shrink* operations must take place between a data block rebuild and an index block rebuild, as well as between two index block rebuilds.

Between the lower bound data block rebuild at $N = (\frac{B}{4})^2$ and the lower bound index block rebuild at $N = \frac{B^2}{4}$ there is a factor B^2. Immediately after rebuilding during a *Grow* operation, it follows that $N = (\frac{B}{2})^2 = \frac{B^2}{4}$. The same is true immediately after a rebuild during a *Shrink* operation as $N = (\frac{B}{4})^2 = \frac{B^2}{4}$. However, these will not trigger the index block downsizing, which also happens at $N = \frac{B^2}{4}$, since the index block already has size $\frac{B}{2}$. The index block will, however, be doubled in size at $N = \frac{B^2}{2}$ which requires at least a factor B^2 *Grow* or *Shrink* operations after the data block rebuild. From an index block rebuild at $N = \frac{B^2}{2}$ to a data block rebuild at $N = B^2$, there must also be a factor B^2 *Grow* or *Shrink* operations in between. Finally, the index block rebuilds happen at $N = \frac{B^2}{4}$ and $\frac{B^2}{2}$, in which there must also be a factor of B^2 *Grow* and *Shrink* operations in between.

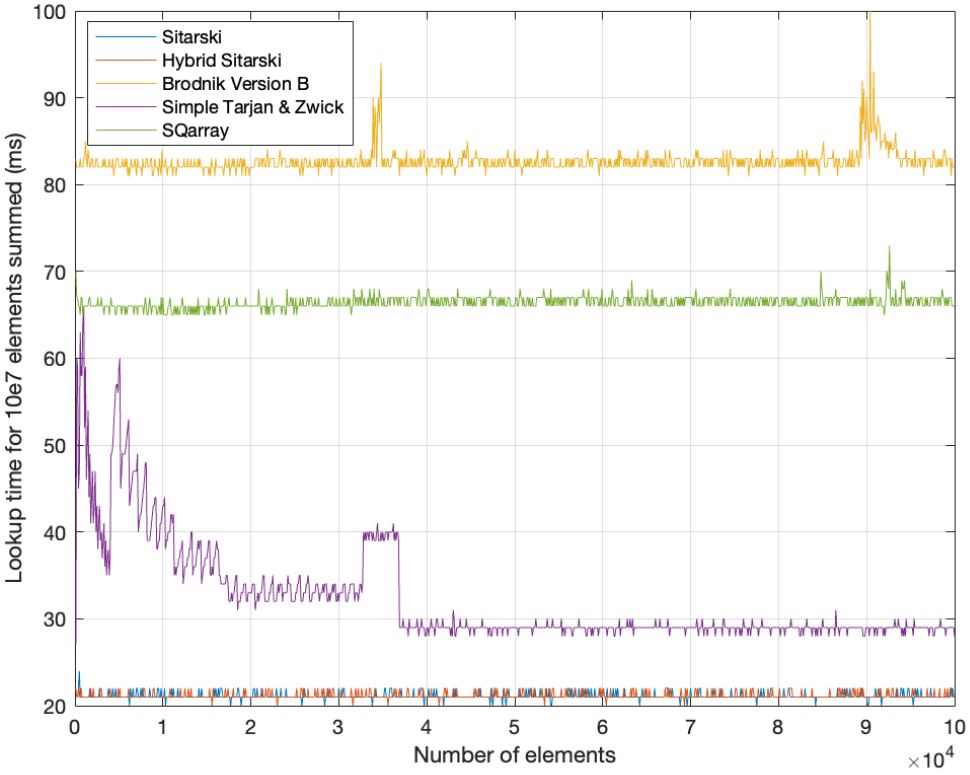


Figure 6.10: Times to perform 10 million lookups for the previous $\mathcal{O}(\sqrt{N})$ excess memory solutions discussed and Hybrid Sitarski.

Thus, since there must be $\mathcal{O}(B^2)$ *Grow* and *Shrink* operations between any pair of data block rebuilds and index block rebuilds, the amortized cost must be $\mathcal{O}(1)$. □

Lemma 3. *After any sequence of Grow and Shrink operations, the Hybrid Sitarski data structure uses at most the same amount of excess memory as Sitarski's HAT data structure except when* $N = \frac{B^2}{2}$ *or* $N = \frac{B^2}{4}$ *where* B *is a positive power of two.*

Proof. Both data structures rebuild data blocks when $N = B^2$ during *Grow* and $N = (\frac{B}{4})^2$ during *Shrink*. Thus, the data blocks will have the same size B after any sequence of *Grow* and *Shrink* operations, implying that the excess memory from the last partially full data blocks is exactly the same for both data structures. Therefore, it is sufficient to investigate the excess memory from the index block and the temporary excess memory caused by data block rebuilds. To do this, we proceed by proving the said property when $N = (\frac{B}{4})^2$, $N = B^2$ and in the open intervals $N \in ((\frac{B}{4})^2, \frac{B^2}{4})$, $N \in (\frac{B^2}{4}, \frac{B^2}{2})$ and $N = (\frac{B^2}{2}, B^2)$. These are all possible values of N, which are not explicitly neglected in the lemma.

- $N = (\frac{B}{4})^2$: Here, the data structure is rebuilt caused by a *Shrink* operation. Both data structures have a temporary memory overhead from rebuilding the data blocks and the index block. Rebuilding the data blocks causes exactly the same memory overhead for both solutions, as the data blocks have the same size. The HAT data structure rebuilds the index block from size B to $\frac{B}{2}$ (using the B value when the rebuild is triggered). In the hybrid version, the index block must be of size $\frac{B}{2}$ at the point of rebuilding, since both *Grow* and *Shrink* rebuild the index array to $\frac{B}{2}$ (using

the B value after the rebuild). Additionally, the index block is only increased in size to B when $N = \frac{B^2}{2}$. Since we now are at $N = (\frac{B}{4})^2$ we must have passed $N = \frac{B^2}{4}$ which shrinks the index block back to size $\frac{B}{2}$. Thus the hybrid version rebuilds the index block from $\frac{B}{2}$ to $\frac{B}{4}$ (with the old B value), implying that the temporary excess memory is smaller than that of the HAT data structure.

- $N = B^2$: Here, the data structure is rebuilt caused by a *Grow* operation. Similar to the *Shrink* rebuild, the temporary excess memory caused by the data block rebuild is equal for both solutions. Since the Hybrid Sitarski solution does not rebuild the index block, as opposed to Sitarski's HAT solution, Hybrid Sitarski must use less temporary excess memory than Sitarski's HAT data structure.
- $N \in ((\frac{B}{4})^2, \frac{B^2}{4})$: In this interval, the index block of the hybrid version can only be of size $\frac{B}{2}$ since the only time the index block is increased to B is when $N = \frac{B^2}{2}$. To be in this interval after increasing the index block, we must have shrunk past $\frac{B^2}{4}$, which reduces the index block back to size $\frac{B}{2}$. Since the index block of the HAT structure has size B, the hybrid version is always better in terms of excess memory in this interval.
- $N \in (\frac{B^2}{4}, \frac{B^2}{2})$: Here, the hybrid solution is in a state in between two index block rebuilds. The index block can be of size $\frac{B}{2}$, e.g. if there is a data block rebuild followed by a sequence of *Grow* operations. The index block can also have size B if, for example, a sequence of *Grow* operations caused an index block rebuild at $N = \frac{B^2}{2}$ followed by some *Shrink* operations bringing N back into the considered interval. Again, the HAT data structure always has an index block of size B. Thus, there is no guarantee that the hybrid version is better in terms of excess memory, but it is at least as good in this interval.
- $N = (\frac{B^2}{2}, B^2)$: In this interval, the index block of the hybrid data structure must be of size B. This is because both *Grow* and *Shrink* rebuild operations leave the index block at size $\frac{B}{2}$. Thus, there must have been a sequence of a majority of *Grow* operations since the last data block rebuild. These trigger the index block rebuild to size B at $N = \frac{B^2}{2}$. The index block is only halved at $N = \frac{B^2}{4}$, which is out of this interval and this state is therefore irrelevant. Since the index block of the HAT structure also has size B, the excess memory of the hybrid version is exactly equal to that of the HAT data structure.

It has now been proven that for all possible values of N except for $N = \frac{B^2}{4}$ and $N = \frac{B^2}{2}$, the Hybrid Sitarski solution uses at most the same memory as the HAT data structure. This is independent of B and, therefore, also holds for any B, which is a power of two.

□

6.2.1 Varying the rebuilding range

The choice of halving the index block can seem arbitrary, and a natural question to ask is what happens if this factor is changed, e.g. to $\frac{1}{4}$. The problem here is that when the data structure rebuilds to the structure with an index block of size $\frac{B}{4}$ with B doubled, the data structure becomes immediately full and has to rebuild again. This is because the rebuild happens when $N = B^2$ and we rebuild to a structure with an index block of length $\frac{2B}{4}$ and data blocks of length $2B$. Multiplying these together, the capacity is B^2. Thus, the data structure is exactly full. It follows that there can, at most, be a factor of 2 between the sizes of the two index blocks. This, of course, also implies that we cannot construct a solution which rebuilds the index blocks to sizes of the following sequence: $1, 2, 4, \ldots, \frac{B}{4}, \frac{B}{2}, B$ and

the rebuild back to an index block of size 1 with B doubled. However, another solution that may be of interest is when we go between index blocks of size $\frac{B}{4}$ and $\frac{B}{2}$. Going beyond B, for example, by going from B to $2B$ is already ruled out, as we saw from figure 6.4. Here, Sitarski Double performed significantly worse. Thus, we will only test going from $\frac{B}{4}$ to $\frac{B}{2}$ and compare this method to the Hybrid Sitarski solution. The results are shown in figure 6.11. From figure 6.11, we see that the version going from $\frac{B}{4}$ to $\frac{B}{2}$ sometimes uses

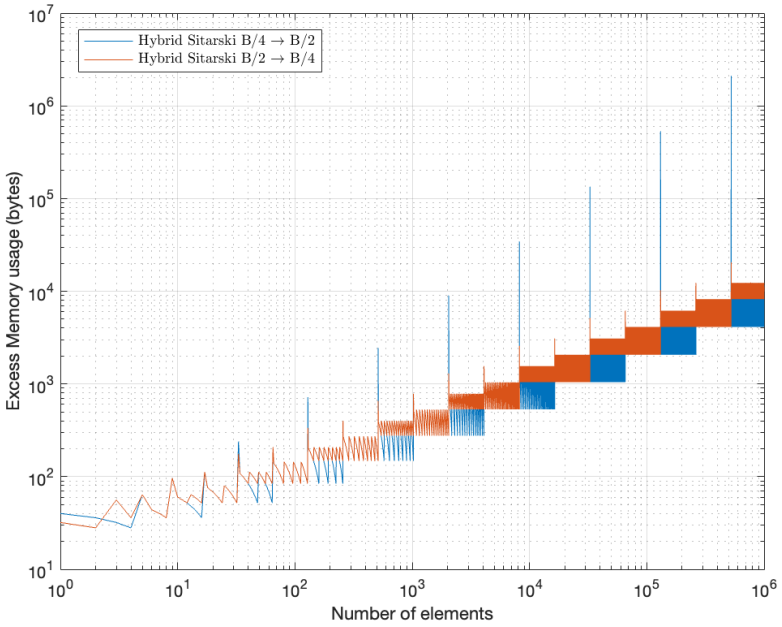


Figure 6.11: Excess memory of Hybrid Sitarski solution with index block size going from $\frac{B}{4}$ to $\frac{B}{2}$.

less additional memory than the Hybrid Sitarski solution previously described. However, we also see that the spikes are significantly larger. During the interval where it sometimes uses less memory, i.e. where the index block has size $\frac{B}{4}$, the range in which the excess memory oscillates is twice as large. This is because going from $\frac{B}{4}$ to $\frac{B}{2}$ is essentially the same as going from $\frac{B}{2}$ to B but with data blocks of size $2B$. Varying the size of the data blocks will be discussed further in section 6.2.3. We could have continued with the model that takes index blocks from size $\frac{B}{4}$ to $\frac{B}{2}$ and designed a fully functional data structure supporting both *Grow* and *Shrink* operations in amortized constant time. This would require changing the *Grow* rebuild to when $N = \frac{B^2}{2}$ and the *Shrink* rebuild to when $N = (\frac{B}{8})^2$. However, we decided not to continue this path due to the large spikes that come at the cost of only minor additional excess memory savings.

6.2.2 The incremental index block approach

Since having a smaller index block leads to less additional memory, another idea is to start the data structure with an index block of size $\frac{B}{2}$. Then one could incrementally increase the size of the index array by 1 as needed, i.e. when the last data block is full. The data blocks are still rebuilt with B doubled when $N = B^2$. Note that this incremental approach implies that the index block has to be rebuilt every B insertion leading to $\frac{B}{2}$ index block rebuilds between each data block rebuild, and the memory spikes grow incrementally accordingly when considering a sequence of *Grow* operations.

Figure 6.12a shows the excess memory of both the Hybrid Sitarski and the Hybrid Sitarski

using the incremental approach.

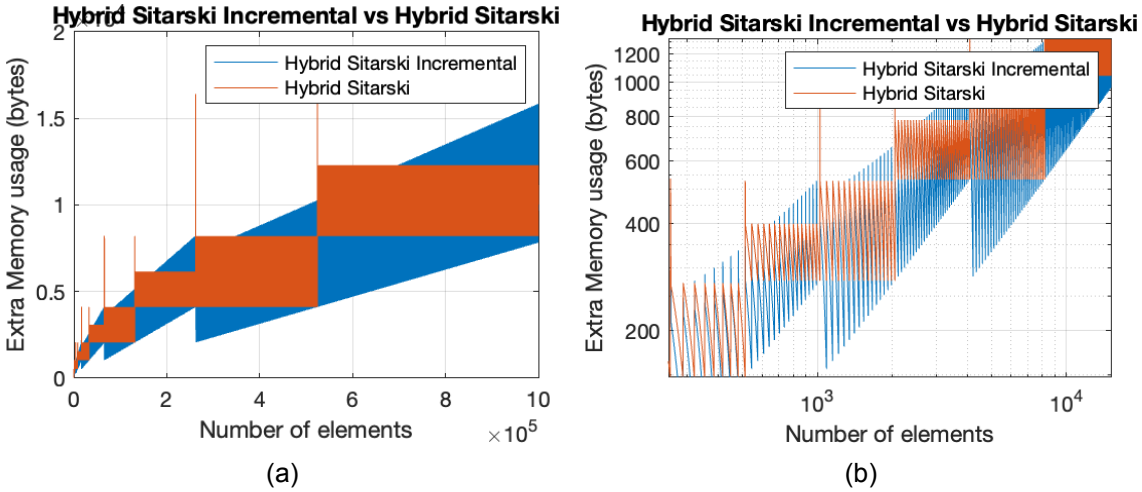


Figure 6.12: Excess memory of Hybrid Sitarski and Hybrid Sitarski using the incremental approach. (b) is a double log chart of a zoomed section of (a).

The results in figure 6.12a show that we do not see sudden jumps in excess memory from rebuilding, but rather a gradual increase as the index block incrementally grows from $\frac{B}{2}$ to B. Where the Hybrid Sitarski solution has a fixed range of excess memory between rebuilds, the width of excess memory of Hybrid Sitarski using the incremental approach gets wider. It's important to note that the graph doesn't show the spikes in the incremental data structure during data block rebuilds, which are comparable to the ones shown. While figure 6.12a gives good insight into how the data structures work in comparison, it is hard to determine which is better regarding excess memory as the lines merge. As previously described, the index block has to be rebuilt after every B *Grow* operation leading to spikes. To see this, figure 6.12b shows a zoomed version in a double logarithmic plot. Here we see that the darker areas of the blue line corresponding to the incremental approach are where the excess memory lies most of the time. These oscillations are where the last data block gets filled up. The spikes above correspond to the B index block rebuilds between each data block rebuild. We see that the incremental approach's concentrated area perfectly aligns with the non-incremental approach right before a data block rebuild and the index block rebuild in the non-incremental approach. At all other times, the concentrated area of the incremental approach lies below that of the non-incremental approach. Thus, even though the incremental approach is worse at certain times, the worse part only stems from the spaced-out index block rebuilds and is not the state of the data structure most of the time.

A fairly simple calculation of the total excess memory used between two data block rebuilds for a sequence of *Grow* operations shows that the incremental uses less total excess memory than the Hybrid Sitarski solution during this sequence. First, we only consider the excess memory used by the index block throughout the sequence since the excess memory used by the partially empty data blocks is the same for both data structures. Even further, we only have to consider the part of the index block that is not used since both data structures will have the same amounts of elements in the index block after the same number of *Grow* operations. Starting with the incremental approach, the index block is always full, meaning that the excess memory difference due to the index block stems from index block rebuilds which happen every B insertions. The first time

this happens, the excess memory is $\frac{B}{2}$, then it is $\frac{B}{2}+1$ and so on up to B. The sum $\frac{B}{2}+(\frac{B}{2}+1)+(\frac{B}{2}+2)+\cdots+(\frac{B}{2}+\frac{B}{2})$ with $\frac{B}{2}$ terms in total can be simplified to $\frac{1}{8}B^2+\frac{3}{4}B$. Now looking at the Hybrid Sitarski solution, from just after a data block rebuild until $N=\frac{B^2}{2}$, i.e. while the index block of size $\frac{B}{2}$ goes from being half full to full, the excess space from the partially full index block decreases by one or every B *Grow* operations. Thus the total contribution to the excess memory is $B\cdot(\frac{B}{4}+(\frac{B}{4}-1)+(\frac{B}{4}-2)+\cdots+2+1)=B\cdot\frac{\frac{B}{4}\cdot(\frac{B}{4}+1)}{2}$. Similarly, the contribution from the second part after the rebuild of the index array is $B\cdot(\frac{B}{2}+(\frac{B}{2}-1)+(\frac{B}{2}-2)+\cdots+2+1)=B\cdot\frac{\frac{B}{2}\cdot(\frac{B}{2}+1)}{2}$. We also have to add the additional memory from the rebuild of the index array, which is $\frac{B}{2}+(\frac{B}{2}+1)$. In total this gives $\frac{5}{32}B^3+\frac{3}{8}B^2+B+1$. Thus, the non-incremental approach is asymptotically worse than the incremental approach when considering the difference in total excess memory usage between two rebuilds following *Grow* operations.

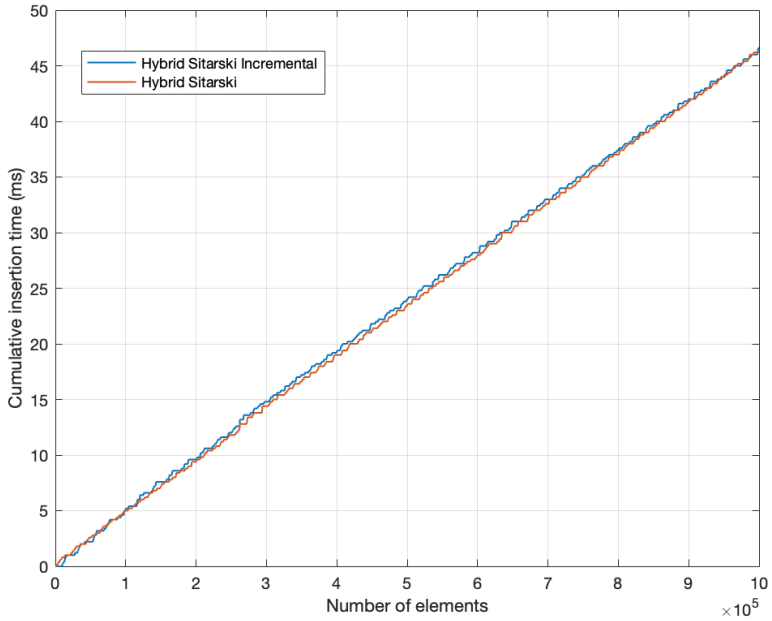
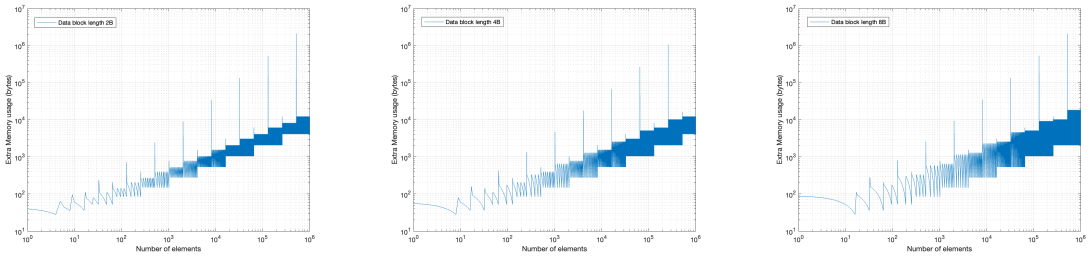


Figure 6.13: Cumulative insertion times for the Hybrid Sitarski Incremental solution and Hybrid Sitarski solution.

Intuitively, the incremental approach should be slower on average since the index block is rebuilt more often. However, figure 6.13 shows no significant difference in cumulative insertion time. Furthermore, the average insertion time of the incremental approach is $4.66e^{-5}$ ms versus $4.62e^{-5}$ ms of the Hybrid Sitarski. If only *Grow* operations were allowed and no *Shrink* operations, the amortized time of *Grow* would be $\mathcal{O}(1)$, since rebuilding the index array takes $\mathcal{O}(B)$ time (as the length of the index block ranges from $\frac{B}{2}$ to B), and rebuilding of the index array happens after B *Grow* operations. When introducing *Shrink* operations, however, the issue with the incremental approach is that we can get into a sequence of *Grow* and *Shrink* operations that would break the $\mathcal{O}(1)$ amortized bounds of *Grow* and *Shrink* operations. To see this, consider the data structure just after incrementing the length of the index array. Here, the last data block will have exactly one element. If a *Shrink* operation is called, we must rebuild the index array again and decrease the length. Now it only requires one *Grow* operation to rebuild the index array again, thus giving us $\mathcal{O}(B)$ time *Grow* and *Shrink* operations. A solution to this problem



(a) Hybrid Sitarski with data block lengths of $2B$.

(b) Hybrid Sitarski with data block lengths of $4B$.

(c) Hybrid Sitarski with data block lengths of $2B$.

Figure 6.14: Hybrid Sitarski with various data block lengths

would be to allow the last data block to be empty and the second to last data block partially full or full, similar to the version of Sitarski's HAT solution explained at the bottom of section 3.3. This would, however, add an extra B words of excess memory, thus defeating the purpose of the data structure, rendering the incremental approach worse than Hybrid Sitarski when aiming to maintain $\mathcal{O}(1)$ time *Grow* and *Shrink* operations.

6.2.3 Varying the size of data blocks

As mentioned earlier, excess memory in the Sitarski solution is a combination of the index block and the last data block, which may be partially full. The effect on excess memory resulting from varying the index block size has been thoroughly investigated, and we have yet to see the effect of varying the size of the data blocks. This was tested using the Hybrid Sitarski data structure since this was proven to be better in terms of excess memory than the standard Sitarski solution by lemma 3. Figure 6.14 shows the Hybrid Sitarski data structure with data block sizes of $2B$, $4B$ and $8B$, respectively, from left to right. The graphs show that the centre at which the segments between rebuilds oscillate is the same. However, the oscillations' width increases with the data blocks' size. That is, between the rebuilds, the peak excess memory becomes larger, and the lowest excess memory becomes lower. Intuitively, this makes sense since each oscillation corresponds to the current last data block being filled up. When it is full, the excess memory is the smallest, after which a new data block is allocated, yielding the smallest excess memory. By increasing the size of the data blocks, this span becomes wider. We cannot conclude that increasing the data block size makes the data structure better or worse in terms of excess memory, but merely that it increases the volatility of the excess memory. Some applications might favour more stable memory usage, but this is up to the developer to decide.

6.3 Hybrid Tarjan and Zwick

With the Hybrid Sitarski solution discussed in section 6.2 in mind and the fact that Tarjan and Zwick's [1] solution is based on Sitarski's solution [2] it made sense to test if a similar hybrid structure could benefit the $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$ family of data structures.

6.3.1 Hybrid $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ data structure

Initially, we decided to test if the hybrid method is feasible with the simple $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ data structure. Recall that the simple $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ data structure has an index array of size B for large data blocks and an index array of size $2B$ for small data blocks. If we were to halve the size of both index arrays in the data structure as we do for Hybrid Sitarski, the index array for the small data blocks would be of size B. Since the small data

blocks are of size B and the large data blocks of size B^2, this means that if we split a large block into small blocks, all small blocks would immediately be full. If it was followed by a *Grow* operation, a *CombineBlocks* operation is needed immediately following the *SplitBlocks* operation to make space for the new element. Likewise, if a *Shrink* operation follows a *Grow* operation that caused a *CombineBlocks* operation, all small data blocks would be empty, thus requiring a *SplitBlocks* operation. Thus, if there is a series of *Grow* and *Shrink* operations immediately following a *Grow* or *Shrink* operation causing either a *CombineBlocks* or a *SplitBlocks* operation to occur respectively, each of the following *Grow* and *Shrink* operations would take $\mathcal{O}(B^2)$ time. As such, we need $\mathcal{O}(B^2)$ *Grow* or *Shrink* operations to occur between two *CombineBlocks* or *SplitBlocks* operations in order to maintain the amortized $\mathcal{O}(1)$ time for *Grow* and *Shrink* operations. This is also likely why Tarjan and Zwick chose the size of the small index array to be of size $2B$. It becomes clear that the small index array cannot be of length B. It is possible however to maintain the amortized $\mathcal{O}(1)$ *Grow* and *Shrink* operations by choosing the size of this index array to be say $\frac{3}{2}B$ instead, as it is then of size $B + \mathcal{O}(B)$, allowing $\mathcal{O}(B^2)$ *Grow* or *Shrink* operations between two *CombineBlocks* or *SplitBlocks* operations. The size of the index array pointing to large data blocks doesn't influence the $\mathcal{O}(1)$ amortized *Grow* and *Shrink* operations like the size of the index array for the small blocks does, as only one block is either moved into or out of this index array. The size of this can then be changed to $B/2$ with no complications. As such, the changes compared to the standard $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ implementation is that the size of the index array for the large blocks is changed to $B/2$ and the size of the index array for the small blocks is changed to $\frac{3}{2}B$ as seen in figure 6.15.

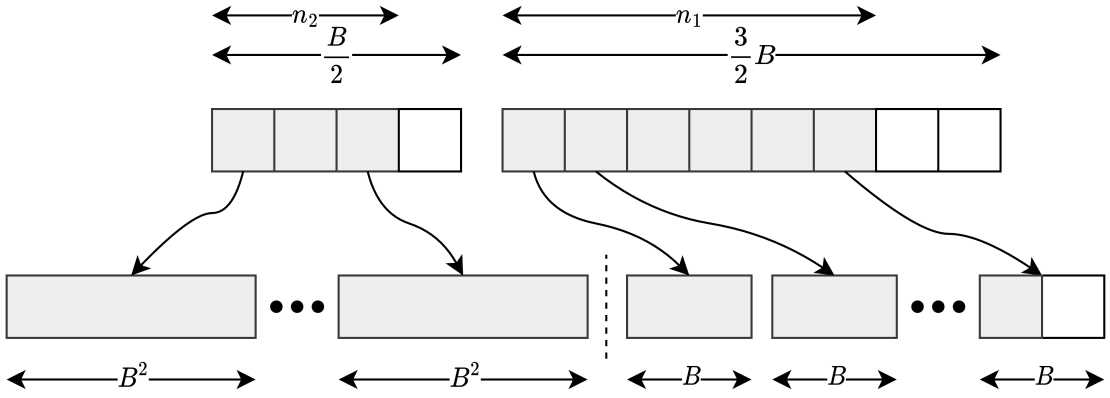


Figure 6.15: Sketch of the hybrid version of Tarjan and Zwick's simple solution

When $N = \frac{B^3}{2}$, we rebuild the index blocks be of size B and $2B$ respectively. When $N = B^3$, the data structure is rebuilt with B doubled and the index blocks with size $B/2$ and $\frac{3}{2}B$ respectively again. A *Shrink* operation works similarly to how it worked for Sitarski Hybrid. When $N = (\frac{B}{4})^3$, the data blocks are rebuilt with B halved. However, the index blocks are also rebuilt to size $\frac{B}{2}$ and $\frac{3}{2}B$, respectively, for the two index blocks. Furthermore, when $N = \frac{B^3}{4}$ and if the length of the index block for the large data blocks is B and the length of the index block for the small data blocks is $2B$, the index blocks are rebuilt to size $\frac{B}{2}$ and $\frac{3}{2}B$ respectively. Here, the small index block must only be $3/4$ full, otherwise, there isn't space to rebuild it. We can only reach $N = \frac{B^3}{4}$ when the index block for the small data block is either full or half full. We can only reach $N = \frac{B^3}{4}$ and have the index block for the small data blocks be full if we had a *Grow* operation prior to the current *Shrink* operation.

Yet if the previous operation was a *Grow* operation, then $N < \frac{B^3}{4}$, which cannot occur. Therefore, the index block for the small data blocks will only ever be half full when $N = \frac{B^3}{4}$.

The hybrid $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ solution was implemented, tested and compared to the $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ solution from Tarjan and Zwick, which can be seen in figure 6.16. Figure 6.16 shows the total excess memory of the two solutions, i.e. both the excess space used to store the resizable array of size N and the temporary space used to resize during a *Grow* or *Shrink* operation as per definition 3.6.1.

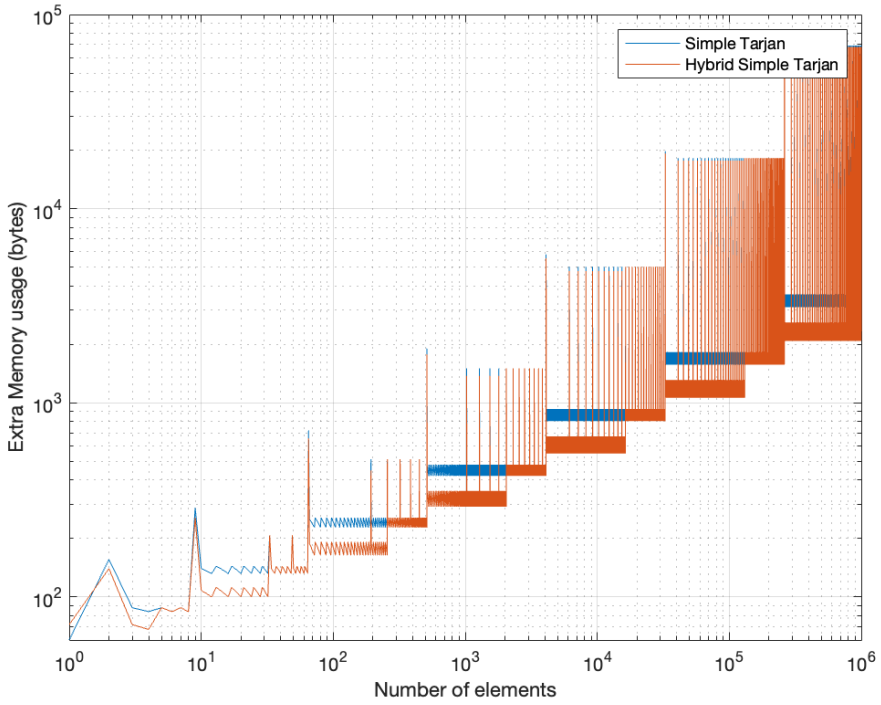
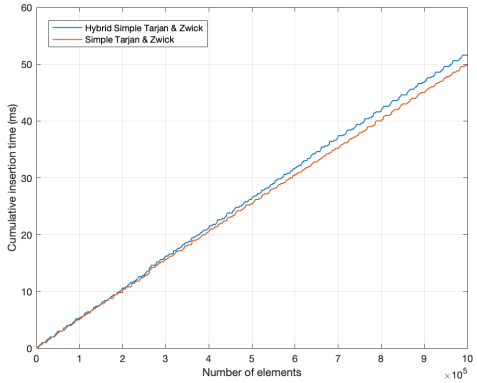


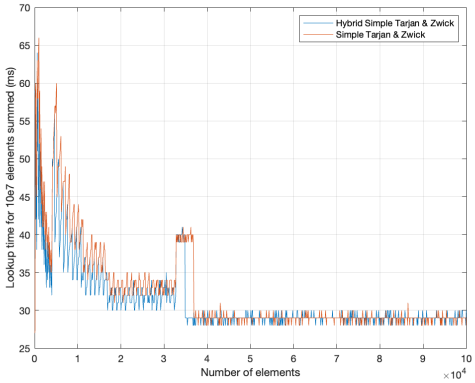
Figure 6.16: Excess memory of Hybrid and standard Tarjan and Zwick considering total excess memory and excess memory used to store N elements.

As depicted in Figure 6.16, the temporary excess memory (the spikes) appears to be nearly identical. It is quite clear to see that the excess memory used to store a resizable array of size N is considerably lower when using the hybrid $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ solution discussed here compared to that of the standard $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ solution by Tarjan and Zwick.

Figure 6.17 shows cumulative insertion and access times for the hybrid version compared to the standard version by Tarjan and Zwick. The figure shows that the hybrid version seems to perform slightly worse for insertion time and about the same for access time.



(a) Cumulative insertion time of hybrid and standard version of Tarjan and Zwick's simple solution.



(b) Times to perform 10 million lookups for the simple version of Tarjan and Zwick's data structure and the hybrid version.

Figure 6.17: Access and insertion time tests for the Hybrid $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ data structure.

6.3.2 Hybrid $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$ data structure

The principle of the hybrid $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$ family of data structures is very similar to that of the simpler Hybrid $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ data structure. To ensure amortized $\mathcal{O}(r)$ *Grow* and *Shrink* bounds, the index arrays can not be of size B. Instead, like in the previous section, the size of the index blocks must be $\frac{3}{2}B$, except index block $r-1$, which can be of size $B/2$. As level $r-1$ is the last level, there are no data blocks at a higher level that can be split into $B-1$ blocks and $B-1$ blocks cannot be combined into a block at level r as it does not exist. The choice of the index blocks having size $\frac{B}{2}$ in level $r-1$ is somewhat arbitrary but is consistent with the ideas of halving the index block in the Hybrid Sitarski solution in section 6.5.

The original solution by Tarjan and Zwick rebuilds the data structure when $N = B^r$ during a *Grow* operation as described in section 3.6. At this point, the data structure is not at maximum capacity - it only corresponds to the last level being half full. In general, it is beneficial to rebuild the data structure during a *Grow* operation when it is as close to being full as possible. This delays the rebuild operation as much as possible, thus minimizing excess memory by keeping smaller index blocks and data blocks for as long as possible. In addition, delaying the rebuild operation is also beneficial in terms of the insertion time, as it is a costly operation. For the hybrid solution, we consider two rebuilds that can happen during the *Grow* operation, namely rebuilding the index block and rebuilding the data blocks. Delaying the rebuilding of the index blocks as much as possible is also beneficial for the exact same reasons. For both types of rebuilds, we can calculate the maximum capacity of the data structure at both points. Starting with the index block, the sizes are $\frac{3}{2}B$ for levels 1 to $r-2$ and $\frac{1}{2}B$ for level $r-1$. The capacity, C, for a given B and r before a rebuild of the index blocks is then given by equation 6.1.

$$C = \frac{3}{2}B \cdot \sum_{i=1}^{r-2} B^i + \frac{1}{2}B \cdot B^{r-1} = \frac{B^r}{2} + \frac{3B^r}{2B-2} - \frac{3B^2}{2B-2} \tag{6.1}$$

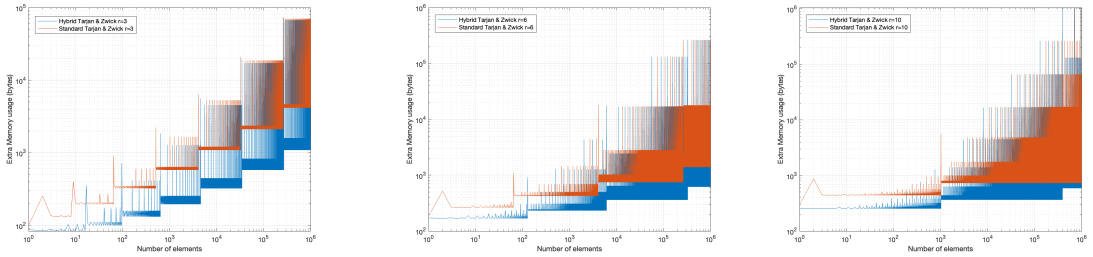
To fully minimize excess space, we should rebuild the index blocks when N is equal to the value of equation 6.1. However, this check has to be done for every *Grow* operation and should therefore be as fast as possible to calculate. Thus, if an approximation of the expression can be computed faster, it could be advantageous without losing much. It is important not to go above the calculated capacity, as the data structure would run out of capacity before a rebuild would occur. Looking at the terms of equation 6.1, we have that $\frac{3B^r}{2B-2} = \frac{3}{2}\frac{B^r}{B-1} > \frac{3}{2}\frac{B^r}{B} = \frac{3}{2}B^{r-1} > B^{r-1}$ since $B \geq 2$. We know that the last term $\frac{3B^2}{2B-2}$ must be subtracted, however, this is actually covered in the first inequality of the previous reduction. The amount we loose in this reduction is $\frac{3}{2}\frac{B^r}{B-1} - \frac{3}{2}\frac{B^r}{B} = \frac{3}{2}\frac{B^rB-(B-1)B^r}{(B-1)B} = \frac{3}{2}\frac{B^{r+1}-B^{r+1}+B^r}{(B-1)B} = \frac{3}{2}\frac{B^{r-1}}{(B-1)}$. As $B \geq 2$ and $r \geq 3$, the amount we lose in the previous simplification is enough to cover for the $\frac{3B^2}{2B-2}$ elements we should subtract. Thus, we can rebuild the index block when $N = \frac{B^r}{2} + B^{r-1}$. This expression is less but close to full capacity and can be computed fast using bit manipulation as follows: $((1 << \log(B) \cdot r) >> 1) + (1 << \log(B) \cdot (r-1))$. Again, $\log(B)$ can be maintained by the data structure to further speed up this calculation.

Now looking at when to rebuild the data blocks, we know that level 1 to $r-2$ contains index blocks of size $2B$ and level $r-1$ has an index block of size B since we must have rebuilt the index blocks between two data block rebuilds. Thus, the total capacity, C, for a given B and r prior to a data block rebuild is given by equation 6.2.

$$C = 2B \cdot \sum_{i=1}^{r-2} B^i + B^r = B^r + \frac{2B^r}{B-1} - \frac{2B^2}{B-1} \tag{6.2}$$

Similar to before, we are interested in an expression which is lower but close to equation 6.2, which can be computed efficiently. Again, we must subtract the last term $\frac{2B^r}{B-1}$. This is covered by the simplification done in the following step. By a similar calculation, the difference is $2\frac{B^r}{B-1} - 2\frac{B^r}{B} = 2\frac{B^r B-(B-1)B^r}{(B-1)B} = 2\frac{B^{r+1}-B^{r+1}+B^r}{(B-1)B} = 2\frac{B^{r-1}}{(B-1)}$. Since $B \geq 2$ and $r \geq 3$, this is at least as large as $\frac{2B^r}{B-1}$. Thus, we rebuild the data blocks when $N = B^r + 2B^{r-1}$ which can be checked fast in practise as follows: $(1 << \log(B) \cdot r) + ((1 << \log(B) \cdot (r-1)) << 1)$. Moving the rebuilds closer to capacity is, of course, not limited to this hybrid version but can also be utilized in the standard $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$ data structure by Tarjan and Zwick. Here, the rebuild can actually be pushed even further since the index blocks have a size of $2B$.

For the *Shrink* operation, it is slightly different than that of Hybrid Sitarski and the Hybrid $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ data structure. It still halves the size of the data blocks when $N = (\frac{B}{4})^r$ and rebuilds all index blocks to be of size $\frac{3}{2}B$, except for the index block in level $r-1$ which is rebuilt to size $B/2$. Here, however, it is not possible to do an index block rebuild when $N = \frac{B^r}{4}$, as there is no guarantee that the index blocks for levels 1 to $r-2$ are only $3/4$ full and that the index block for level $r-1$ is only half full because they are not considered individually. This was also the reason that the second rebuild version for the $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$-implementation potentially doesn't work for the $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$-implementation, as mentioned in section 4.



(a) (b) (c)

Figure 6.18: Excess memory of Hybrid $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$ for $r = 3$, $r = 6$ and $r = 10$

Lemma 4. *The Hybrid $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$ data structure uses $N + \mathcal{O}(rN^{1/r})$ excess memory to store the array and $N + \mathcal{O}(N^{1-1/r})$ memory temporarily during a Grow or Shrink operation.*

Proof. The Hybrid $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$ only has two states. 1) being in a state of the $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$-solution by Tarjan and Zwick or 2) being in a state of having smaller index arrays. The data structure maintains that $N^{1/r} \leq B \leq 4N^{1/r}$. When in the first state, at most $N + r + (r-1) + (r-2) \cdot (2B-1) + B + B = N + \mathcal{O}(rN^{1/r})$ space is used to store the array. When in the second state at most $N + r + (r-1) + (r-2) \cdot (\frac{3}{2}B - 1) + B/2 + B = N + \mathcal{O}(rN^{1/r})$ space is used to store the array. In any case, it always uses, at most, $N + \mathcal{O}(rN^{1/r})$ space to store the resizable array.

For an index block rebuild, at most $2B + (r-1)$ excess space is temporarily used. This is clearly $\mathcal{O}(N^{1-1/r})$. For a data block rebuild, the maximum temporary excess memory used is when a data block of size $B^(r-1)$ is rebuilt. This is also $\mathcal{O}(N^{1-1/r})$, since $N^{1/r} \leq B \leq 4N^{1/r}$. Thus, $\mathcal{O}((N^{1/r})^{r-1}) = \mathcal{O}(N^{1-1/r})$. □

Lemma 5. *The Hybrid* $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$ *data structure supports Grow and Shrink operations in amortized* $\mathcal{O}(r)$ *time.*

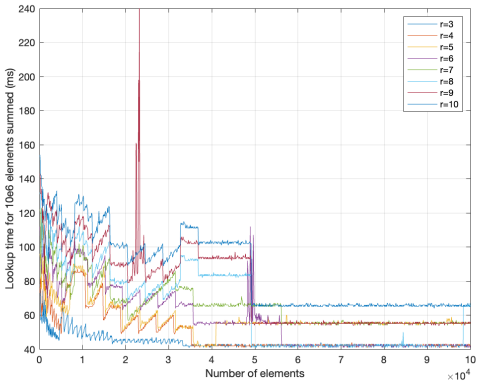
Proof. Tarjan and Zwick showed that their $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$ data structure supports *Grow* and *Shrink* in amortized $\mathcal{O}(r)$ time [1]. One difference from the hybrid version to Tarjan and Zwick's data structure is that when $N < \frac{B^r}{2} + B^{r-1}$, the first $r-2$ levels have index block of size $\frac{3B}{2}$ and the last level has an index block of size B. In Tarjan and Zwick's version, all index blocks have a size of $2B$. The sizes $\frac{3B}{2}$ and B in the hybrid version are still $\mathcal{O}(B)$, which doesn't influence the amortized analysis by Tarjan and Zwick [1]. When $N \in [\frac{B^r}{2} + B^{r-1}, B^r + 2B^{r-1})$, all index blocks have size $2B$ exactly like Tarjan and Zwick's version. Thus, in this case, the amortized bounds hold as well. The last difference from Tarjan and Zwick's data structure is that the hybrid version rebuilds the index arrays when $\frac{B^r}{2} + B^{r-1}$ to have size $2B$. Rebuilding all index arrays takes $\mathcal{O}(rB)$ time. However, since we do not shrink the index arrays again, the index array rebuild can, at most, happen once between two data block rebuilds. Clearly, this does not affect the amortized bounds of $\mathcal{O}(r)$. □

The family of Hybrid $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$ data structures were implemented, tested and compared to that of the standard $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$ solutions by Tarjan and Zwick. The results of these tests and comparisons can be seen in figure 6.18. Figure 6.18 shows the total excess memory, i.e. both the excess space used to store a resizable array of size N and the temporary excess space used during a *Grow* or *Shrink* operation for the three data structures where $r = 3$, $r = 6$ and $r = 9$. It is clear that there are significant memory savings to be had by using the hybrid version. The temporary excess memory seems not to be any worse than that of the general version by Tarjan and Zwick, and the memory required to store the resizable array is clearly lower. It is, however, noticeable that there is less of a gain for larger values of r. This is expected since, for lower values of r, the more significant gain from the last level isn't diluted as much.

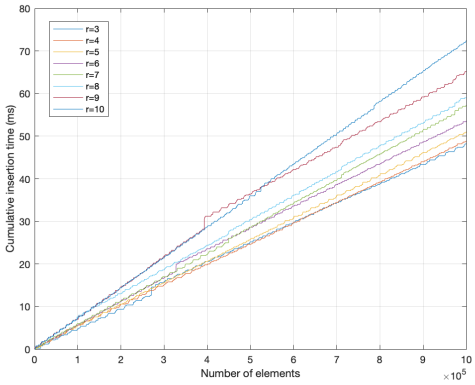
Figure 6.19 shows the results of performing access and insertion tests. Figure 6.19 shows very similar results to that of the general version by Tarjan and Zwick seen in section 5.2 and 5.3. The average access and insertion times are shown together with those of the general version by Tarjan and Zwick in table 6.1 and 6.2.

Data structure	Access time
Hybrid General r=3	44.192 ms
Hybrid General r=6	57.747 ms
Hybrid General r=10	85.811 ms
General r=3	44.731 ms
General r=6	62.577 ms
General r=10	89.991 ms

Table 6.1: Time to perform 10 million lookups for the general version of Tarjan and Zwick's data structure and the hybrid version.



(a) Times to perform 10 million lookups for the hybrid general version of Tarjan and Zwick's data structure.



(b) Cumulative insertion time for the hybrid general version of Tarjan and Zwick's data structure for $r = 3$ to $r = 10$.

Figure 6.19: Insertion and access time for the hybrid general version of Tarjan and Zwick's data structures

Data structure	Average Insertion time
Hybrid General r=3	$4.82e^{-}5$ ms
Hybrid General r=6	$5.36e^{-}5$ ms
Hybrid General r=10	$7.24e^{-}5$ ms
General r=3	$4.72e^{-}5$ ms
General r=6	$5.18e^{-}5$ ms
General r=10	$7.28e^{-}5$ ms

Table 6.2: Table of average insertion time averaged over 1 million insertions for the general version of Tarjan and Zwick's data structure and the hybrid version.

As seen in table 6.1 and 6.2, the average access and insertion times are very similar. Thus, the hybrid method does not seem to influence these significantly.

7 Discussion

Designing a dynamic array data structure requires understanding and navigating the trade-off space between access time, insertion time and memory usage. Memory usage has its own trade-off spectrum between excess memory to store the array and temporary excess memory. Increasing the complexity of the data structure can minimize the excess memory since the memory can be managed in a more fine-tuned manner. However, the complexity implies slower access times and typically also slower resizing of the data structure. These are two crucial functionalities of a dynamic array. An example of a more complex data structure is the $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$ data structure by Tarjan and Zwick [1]. Increased complexity can also work in the opposite direction, where additional excess memory is sacrificed for faster insertion or lookup times. An example of the former is the Fast Combine data structure which was proposed in section 6.1. On the other end of the spectrum, reducing the complexity of the data structure often means very fast access and modification times but naively uses more space than necessary as it can only use a limited amount of logic. Data structures such as the commonly used geometric expansion technique lie on this side of the spectrum. A more versatile dynamic array that can be useful in many programs, instead of the current geometric expansion array, may be found somewhere in between. Solutions in this range could be those by Brodnik et al. and Sitarski as well as the SQarray and the $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ version of Tarjan and Zwick's data structure. However, it is important to consider special use cases. Some applications might not require many access and modification operations and could go for a more complex data structure to reduce excess memory. On the other hand, an application might not need to keep the excess memory at a minimum but prioritizes fast access and resizing operations.

Regarding the memory spectrum, since computer memory comes in limited quantities, it is important to consider that programmers of applications have different preferences or even requirements. The tests for excess memory in section 5.1 demonstrate that Tarjan and Zwick's $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$ solution uses consistently low excess memory. However, it experiences occasional large memory spikes. This can be regulated by varying the r parameter. Some applications might have strict memory limitations, and thus these large spikes may be a problem. From our results, the $r = 10$ version jumps almost 16 times the excess base memory during a rebuild just after inserting a bit more than 10^5 elements. This gap grows asymptotically as $\mathcal{O}(N^{1-1/r})$ and is seen to be growing fast in practice. These spikes will be too much for many applications with a tight memory budget. However, it is important to note that the majority of the major programming languages use the geometric expansion with some growth factor, which the programmer can manually set [6]. Compared to geometric expansion, the spikes from the $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$ version of Tarjan and Zwick's solution are still an order of magnitude smaller for $r = 10$. Thus, in terms of memory, the general solution by Tarjan and Zwick can be a great option for a programmer who wants a low excess memory most of the time but has enough memory overhead to deal with the occasional spikes. Going to the less extreme ends of the spectrum, we found the HAT data structure by Sitarski to be a solution with considerably low excess memory to store the array compared to the similar solutions developed around that time. This is while only spiking a factor of 2 while rebuilding the data structure and keeping very competitive access and insertion times. Therefore, the HAT data structure is a great option for a dynamic array for a programmer who wants a simple solution

with much smaller and more consistent excess memory than the currently used geometric expansion array.

We presented the Hybrid Sitarski solution in section 6.2, which was proven to always be on par or better in terms of excess memory when storing the array compared to Sitarski's HAT data structure. Even the temporary excess memory spikes from rebuilding the data blocks are smaller in the hybrid version. One thing to consider is that it has temporary excess memory spikes when rebuilding the index block at $N = \frac{B^2}{4}$ and $N = \frac{B^2}{2}$. However, these spikes are limited to a factor of 2 compared to the excess memory used to store the array. Furthermore, there must necessarily be $\frac{B^2}{4} = \Omega(N)$ *Grow* or *Shrink* operations between such spikes. Even though the insertion and access times are similar to that of the HAT data structure, the different memory profile makes it impossible to conclude that the hybrid version is strictly better. However, most applications will benefit from spending significantly less excess memory on storing the array most of the time, with only a few occasional relatively small memory spikes. The hybrid version especially shines in memory-restricted systems since the memory spikes, albeit slightly more often, are much smaller, thus allowing for a tighter memory cap.

The hybrid technique was applied to the $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ and $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$ versions of Tarjan and Zwick's data structures. We saw that using the hybrid version can save a significant amount of excess memory, up to around a factor of 2 for the latter, considering the excess memory to store the array as was shown in section 6.3.2. Regulating the index blocks could be a viable solution for dynamic array data structures with multiple layers, as we could apply this technique to other data structures, such as Tarjan and Zwick's simple and general versions. On the other hand, designing a dynamic array to perform well when only undergoing *Grow* operations is easy. As we saw in section 6.2, considering all sequences of *Grow* and *Shrink* operations for the hybrid approach can easily complicate an otherwise simple data structure such as Sitarski's HAT. Another example of how *Shrink* operations complicate resizable arrays is the need for size $B + \mathcal{O}(B)$ index blocks in Tarjan and Zwick's data structures, but index blocks of size B are sufficient if only *Grow* operations are allowed. Not only must the run time amortized bounds be maintained, but the logic of the operations becomes more complex as the total number of states that the data structure can be in increases. This also implies that constructing proofs of run time and memory properties such as Lemma 2 and 3 becomes increasingly complicated. This is true for applying the hybrid approach described in the report and is a natural consequence of making any data structure increasingly complex.

For the access time tests, it was decided to perform 10 million lookups for every time stamp. This was required to get a readable time stamp as a smaller number of lookups occasionally led to a timestamp of zero. Since we are performing 10 million lookups in a row, spatial and temporal locality plays a large role in the access times. As also seen in section 5.3 (table 5.2), the data structures which only maintain a single fixed-sized array have the fastest lookup time. This is expected as both naive and geometric expansion techniques maintain a single fixed-sized array, i.e. contiguous memory blocks. Large chunks are likely to be loaded into the cache after the first cache hit, making it comparatively quicker to access subsequent elements. The other data structures have far more separated data blocks, which may not be located sequentially in memory, and thus may not be loaded in the cache simultaneously. This potentially leads to more cache misses and, thus, increased lookup times. To which extent this affects the access times is unknown. We do, however, see in table 5.3 quite different lookup times between the constant time access and $\mathcal{O}(r)$ time access versions of the general Tarjan and Zwick solution, despite it being exactly the same data structure. As such, the access function implementation has a much

larger influence on the lookup time compared to the temporal and spatial locality of the implementation, even though we are performing 10 million lookups in a row.

As was first seen in section 3.6.2, a simple loop-based method yields $\mathcal{O}(r)$ worst-case access time for the general version of Tarjan and Zwick's data structure. Later in section 4.3, it was seen that using an additional $\mathcal{O}(r)$ excess space allows a constant worst-case access time utilizing the linkage between the structural properties and the base B representation. Even though many of the calculations used in this method were optimized using bit manipulation, it is still a much more complex computation than the first solution taking $\mathcal{O}(r)$ time. In section 5.3, our tests revealed that for $r = 3$ to $r = 10$, the $\mathcal{O}(1)$ method gave average lookup times closer to each other but were still slower in practice than those using the $\mathcal{O}(r)$ method, even for $r = 10$. Interpolating these results, one could imagine that for an r value large enough, the second method will be faster in practice. However, a practical limit exists to how large r can be. As described in section 3.6.2, a B^r calculation is performed during every *Grow* operation. The maximum number for the integer type in C++ is 2147483647. This can easily be surpassed with, for example, $B = 4$ and $r = 16$. Thus, the data structure cannot rebuild from $B = 2$ to $B = 4$ without causing integer overflows. Although, for $N = B^r$ to happen, it requires a huge number of elements to be inserted. For the most part, only the first few levels will be utilized, leaving larger excess memory from the top-level block. Since the value of r is so limited, we may say that the first solution is $\mathcal{O}(1)$ in practice and is simply faster than the second method for all values of practically applicable values of r as not to mention the extra space required. This is part of the reason we chose the $\mathcal{O}(r)$ method for the *Get* operation in the hybrid version of the data structure. The $\mathcal{O}(1)$ technique is merely a theoretically interesting method to achieve $\mathcal{O}(1)$ worst-case bounds. Another method that could be fast in practice is using binary search to find the corresponding level. However, it is necessary to keep the N_k array of size $r - 1$ similarly to the constant time solution since it is necessary to check whether the index of interest is above or below $N_k[i]$ where i is the current level during the binary search. Again, since r is so limited in practice, it might still be faster to loop through every level.

In the $(\mathcal{O}(N^{1/3}), \mathcal{O}(N^{2/3}))$ version, we know that it takes exactly 4 data blocks of size B^2 to fill one data block of size $(2B)^2$. If there are less than 4 data blocks or if there are residual data blocks after combining as many as possible, we know they can all fit into the small blocks of size $2B$ since B^2 is always divisible by $2B$ for any B that is a positive power of 2. The same does not always hold in the general case when dealing with the $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$ data structure. A simple example to illustrate this is when the data structure with $r = 6$ is rebuilt from $B = 2$ to $B = 4$. The data block sizes B^i for the levels $i = 1$ to $i = 5$ before the rebuild are $2, 4, 8, 16, 32$. Following the rebuild with $B = 4$, the resulting sequence is $4, 16, 64, 256, 1024$. If we, before the rebuild, for example, have one data block of size 32, we would either have to combine this with smaller data blocks to create one new data block of size 64 or split it up into, for example, two new blocks of size 16. In general, this becomes a problem of ”packing” the elements into blocks with the extra constraint that the index order must be correctly maintained. This should be solvable since we are only packing into $r - 1$ bins. However, we chose not to follow this path as we already have a simpler $\mathcal{O}(N)$ solution for the *Rebuild* operation as described earlier, and it is not the main focus of this report. Nonetheless, seeing if such a method could bring the *Rebuild* time down in practice could be interesting.

One of the benefits of the $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$ data structure by Tarjan and Zwick and, to the same extent, our hybrid version of it presented in section 6.3.2, is that it can easily be incorporated into a programming language where it is parameterized by r. A programmer

can then choose this value customized to how the programmer values consistent low excess memory versus smaller occasional memory spikes for that particular application. One can then ask the following two questions: 1) Is there an optimal value for r that can minimize the excess memory needed to store an array and the occasional use of temporary memory for every given value of N? 2) If such exists, can we design the data structure which automatically regulates the r value to follow the optimal one? Intuitively, having a data structure with a large r value but containing very few elements, i.e. we are far from capacity, means that only a few of the $r-1$ levels will be used. Therefore the excess memory will be consistently worse than using a lower r value. Similarly, using a data structure with a small r value to store many elements will also negatively impact excess memory compared to using a larger r value. This is because level i has data blocks of size B^i, so having more levels means we can pack more elements into larger data blocks, requiring fewer pointers. This implies that there must be some sweet spot for the best r value as a function of N, i.e. $r = r(N)$. In their paper [1], Tarjan and Zwick assumes that $r = \mathcal{O}(\log(N))$ since $N^{\frac{1}{\mathcal{O}(\log(N))}} = \mathcal{O}(1)$. They then go on to show that if $r(N) \leq \frac{1}{2}\frac{\log(N)}{\log\log(N)}$, there exists an $(\mathcal{O}(N^{1/r}), \mathcal{O}(N^{1-1/r}))$ implementation of a dynamic array with worst case access time $\mathcal{O}(1)$ and *Grow* and *Shrink* operations in amortized $\mathcal{O}(r)$ time. Essentially, this is achieved during the process of copying elements from several existing data blocks into one larger, freshly allocated block, which happens both during the *CombineBlocks* and *Rebuild* operations. The temporary memory overhead can be reduced by copying the elements into several smaller blocks of a specific length to make the bounds hold instead of one large block. This adds an additional layer and is similar to our Fast Combine solution. Practically the r value can be changed during rebuilds, which will not add any additional overhead since the data structure is rebuilt anyway. Thus, the constraint $r(N) \leq \frac{1}{2}\frac{\log(N)}{\log\log(N)}$ should be easy to maintain in practice and might be worth exploring.

8 Conclusion

Although the resizable array problem has been a problem thought to have been completely solved, we show in this project that there are still significant improvements to be made, albeit not asymptotically. We have implemented and compared several previous solutions to the resizable array problem. We found that Sitarski's HAT data structure performed best when considering all parameters, namely excess memory for storing the array, temporary excess memory, access time and insertion time. If excess memory for storing the resizable array at a given time is the primary concern, then Tarjan and Zwick's family of data structures performed best and is parameterized by r to control the ratio between consistent and temporary excess memory. Their data structure with $r = 2$ is essentially the same as Sitarski's HAT data structure, and increasing r shifts the data structure to use less excess memory to store the array and use more during internal reorganization. We found that large r values caused the data structure to spike significantly in memory, even to the point where it may be a problem for certain applications in practice. Increasing r also increases the access time of the data structure. Utilizing the fact that elements are stored in data blocks whose size are powers of B allows for a base B counter, further allowing a $\mathcal{O}(1)$ worst-case access time. However, this was worse in practice for most practically useful values of r.

Based on the findings of our tests, we came up with new solutions and extensions to existing solutions. We presented Fast Combine in section 6.1, a $\mathcal{O}((N^{2/3}), \mathcal{O}(N^{1/3}))$ solution, although an interesting idea, the memory overhead is too large without reducing the insertion/deletion times significantly for this to be a practical solution. This showed how constrained the dynamic array problem is and how increasing the complexity of the data structure slightly can have significant drawbacks. The Hybrid Sitarski data structure was then introduced, a $\mathcal{O}((\sqrt{N}), \mathcal{O}(\sqrt{N}))$ solution using less or equal excess memory to store the array compared to Sitarski's HAT data structure. Periods of less excess memory usage, however, come with the cost of additional memory spikes occasionally. Several index block and data block sizes were tested, but it was found that the sizes used in Hybrid Sitarski resulted in a more consistent and low memory profile. Furthermore, there was no significant difference in access and insertion times between Hybrid Sitarski and Sitarski's HAT data structure. The generalized Hybrid $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$-solution was also presented. This solution showed significantly less excess memory in practice compared to the $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$-solutions by Tarjan and Zwick [1], although this effect decreases with higher values of r. Once again, with no significant differences to the access and insertion times compared to that of Tarjan and Zwicks $(\mathcal{O}(rN^{1/r}), \mathcal{O}(N^{1-1/r}))$-solution. Although it has been proven that the bounds of the problem cannot be improved, significant improvements can still be made to the resizable array problem, as seen by the new solutions presented in this thesis.

Bibliography

[1] Robert E. Tarjan and Uri Zwick. “Optimal resizable arrays”. In: (2023). arXiv: 2211.11009 [cs.DS].

[2] Edward Sitarski. “Algorithm alley: HATs: Hashed array trees.” In: *Dr. Dobb’s Journal of Software Tools* 21.9 (1996), 107–??

[3] Wikipedia contributor. *Array (data structure)*. Online, Accessed: June 30th 2023. URL: https://en.wikipedia.org/wiki/Array_(data_structure).

[4] Online, Accessed: June 30th 2023. URL: https://web.archive.org/web/20150806162750/http://www.gahcep.com/cpp-internals-stl-vector-part-1/.

[5] Online, Accessed: June 30th 2023. URL: https://hg.openjdk.org/jdk6/jdk6/jdk/file/e0e25ac28560/src/share/classes/java/util/ArrayList.java?fbclid=IwAR24YyUKGZ427r0yf0xAPkYc7toyQqJk
pks3yxuYKgGrYiebgsSmdlHT4.

[6] Wikipedia contributor. *Dynamic array*. Online, Accessed: June 30th 2023. URL: https://en.wikipedia.org/wiki/Dynamic_array.

[7] Andrej Brodnik et al. “Resizable Arrays in Optimal Time and Space”. In: *Proc. of 6th WADS* (1999), pp. 37–48.

[8] W. T. Vetterling B. P. Flannery S. A. Teukolsky. *Numerical Recipes in C: The Art of Scientific Computing*. 2nd ed. Camb. Univ. Press, 1992.

[9] Alistair Moffat and Joel Mackenzie. “Immediate-Access Indexing Using Space-Efficient Extensible Arrays”. In: ().

[10] Robert E. Tarjan and Uri Zwick. “Optimal resizable arrays (Unpublished version)”. In: (2023).

Technical
University of
Denmark

Anker Engelunds Vej 1, Building 101
2800 Kgs. Lyngby
Tlf. 4525 1700

www.dtu.dk