Performance Evaluation of Extensible Arrays

by

Beilun Zhang

Student ID: 1156782

A research report submitted for the subject COMP30013

in the
School of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE

June 2023

THE UNIVERSITY OF MELBOURNE

Abstract

School of Computing and Information Systems

by Beilun Zhang

Student ID: 1156782

Arrays serve as one of the most commonly used data structures, providing a fundamental building block for numerous other data structures. The key characteristic of arrays lies in their ability to access and store elements in constant time. *Extensible arrays*, as variants of arrays, exhibit the capability to dynamically expand in size during runtime as elements are added. This flexibility is valuable in scenarios where the maximum storage requirement is unknown at instantiation. Standard implementations of extensible arrays offer constant-time store and access operations, either in the worst case or amortized case, albeit with varying space overheads. In this research project, we conduct a comprehensive performance evaluation of different types of extensible arrays through a series of experiments, primarily focusing on their time and space efficiency. Based on these results, we provide a general guide for when to use each type of array, allowing developers to make informed decisions regarding the adoption of extensible arrays in their projects.

Declaration of Authorship

I certify that:

- this thesis does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person where due reference is not made in the text;
- where necessary I have received clearance for this research from the University's Ethics Committee and have submitted all required data to the School; and
- the thesis is about 6200 words in length (excluding text in images, tables, bibliographies and appendices).

Signed: [illegible]

Date: 20/06/23

Acknowledgements

I would like to express my gratitude to my supervisor Professor Alistair Moffat, for his guidance and support throughout the research project. I would also like to thank the University of Melbourne for providing me with the opportunity to experience hands-on research as an undergraduate. Furthermore, I would like to express my gratitude to my friend Sandy Luo for providing valuable suggestions for this report.

Contents

List of Figures

List of Tables

Symbols

N	number of elements in an array
C	capacity of the array
v	index
k	growth multiplier of geometric array
K	length of a segment in the sliced array
B	variable maintained by HAT and multisliced array
r	number of handles in a multisliced array
x	an element
s	element size

Chapter 1

Introduction

1.1 Context

An array is a ubiquitous data structure in various programming applications, providing two fundamental functionalities: $store(v, x)$, which associates an element x with a positive integer index v, and $get(v)$, which returns the element x associated with index v. The maximum number of items an array can store is defined by its capacity C, and the number of elements it contains at some point in time is denoted by $N \leq C$, where $N = 0$ initially when the array is created.

In a *static* array, the capacity C is fixed and determined at initialization. On the other hand, an *extensible* array is created with an initial capacity C_0, which increases dynamically as the number of elements in the array grows. In general, an extensible array is expected to provide worst-case $O(1)$ $get()$ operation and amortized $O(1)$ $store()$ operation. There exist various implementations of extensible arrays, such as the geometric array, which increases the array capacity by a factor of k when the number of elements in it reaches the full capacity. The geometric array is used as the underlying data structure for C++'s std::vector [1]. In an extensible array, the index v of a $get(v)$ operation must be in the range $0 \leq v \leq N - 1$. For a $store(v, x)$ operation, the index v must be in the range $0 \leq v \leq N$

However, the geometric array may not always be the optimal choice for extensible arrays in certain contexts due to its significant flaw: it incurs a $O(N)$ space overhead, with $(k - 1)N$ space remaining unused after the capacity expansion. Alternative extensible array implementations, such as the sliced array, SQarray, hashed array tree, and the

[1] https://cplusplus.com/reference/vector/vector/, accessed 15th of May 2023

recently proposed array by Tarjan and Zwick [2], aim to mitigate this space overhead while maintaining efficient performance.

1.2 Motivation

The SQarray and the array proposed by Tarjan and Zwick [2] are two types of extensible arrays that require further exploration.

The SQarray is a data structure described by Brodnik et al. [3], and was recently rediscovered by Moffat and Mackenzie [1]. In their work, Moffat and Mackenzie conducted an evaluation of the SQarray's time and space efficiency, comparing it with other arrays such as the sliced array, geometric array, and monolithic array. The monolithic array, functioning as a static array, serves as the baseline for comparison, as it allocates all necessary memory at the start without any memory overhead. On the other hand, the SQarray has a $O(\sqrt{N})$ space overhead, while the space overhead of the sliced and the geometric array can be as large as $O(N)$.

Moffat and Mackenzie [1] investigate these arrays in the context of immediate access indexing, and documented the time and space efficiency of each type of extensible array. However, the performance of the extensible arrays in other contexts, apart from immediate access indexing, remains unclear. This is because the operations for each type of array have the same time complexity, such as amortized $O(1)$ for the $get()$ operation, but may have different constant overheads, cache friendliness, and other factors that can impact their performance.

Meanwhile, Tarjan and Zwick describe an extensible array that achieves $O(N^{\frac{1}{r}})$ persistent overhead, and $O(N^{1-\frac{1}{r}})$ instantaneous space overhead, where r is a constant parameter that the array maintains. The array still has an amortized $O(1)$ $store()$ operation and a worst-case $O(1)$ $get()$ operation. However, they only provide a description of the array and did not implement or test the actual performance of the array, leaving open many questions about its potential uses and limitations. This data structure is built as an extension of the hashed array tree (HAT), which is an extensible array that Sitarski and Alley [4] describes, and achieves $O(\sqrt{N})$ space overhead like the SQarray.

1.3 Aim and Scope

The aim of this study is to compare the performance of different types of extensible arrays, including the geometric array, sliced array, SQarray, Tarjan and Zwick's array, and HAT. While the space overhead of these arrays is different asymptotically, they all offer amortized $O(1)$ $store()$ and $get()$ operations. Therefore, our main focus is to perform an empirical evaluation of the speed of these operations and to identify any differences in constant overhead across the different array types.

1.4 Overview

In Chapter 2, the five extensible array types are introduced and explained in detail, including their structure, time, and space complexity. In Chapter 3, the implementation details of each array type in C are provided, followed by the results obtained when running them on various experiments. We will also analyze these results in detail. Finally, Chapter 4 discusses the significance of these findings and outlines potential avenues for future research.

Chapter 2

Background

This chapter provides an overview of different types of extensible arrays, including how they are implemented and their complexities.

2.1 Extensible array operation

The extensible array supports two main operations: $store(v, x)$ and $get(v)$. The number of elements N in an extensible array is a non-decreasing positive integer, that is initially zero. At any time, the range of valid index for $get(v)$ is $0 \leq v \leq N - 1$, and the range of valid index for $store(v)$ is $0 \leq v \leq N$. If $store(N, x)$ is performed, then the number of elements N is increased by one.

2.2 Geometric array

The geometric array is perhaps the most well-known extensible array. It contains a handle that points to a single segment with capacity C. When $N = C$ and a $store(N, x)$ operation is performed, a new segment of size $\lceil C \times k \rceil$ is created, and the N elements in the original segment are copied into this new segment. The parameter $k > 1$ is called the growth parameter of the geometric array. $\lceil C \times k \rceil$ would then become the new capacity of the geometric array, and the handle would point to the address of this new segment.

2.3 Sliced array

The sliced array has 3 main components:

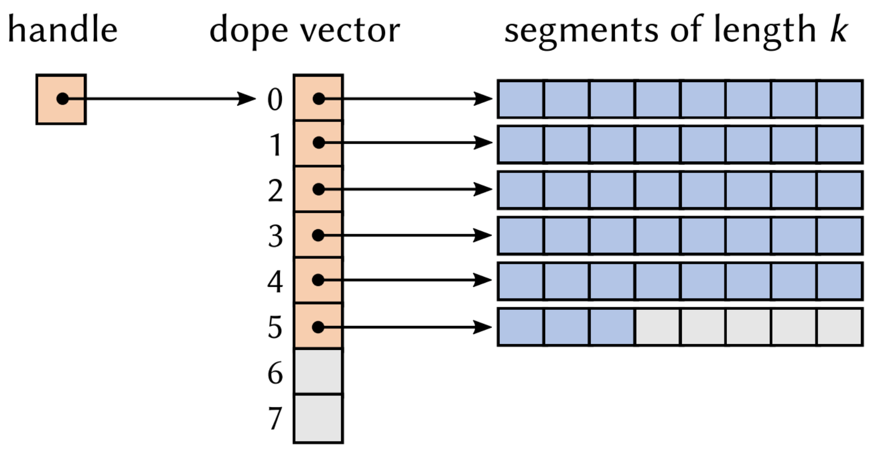


FIGURE 2.1: Structure of the sliced array. Source: Moffat and Mackenzie [1]

- *handle*: A pointer to the start of the dope vector
- *dope vector*: A contiguous array of pointers, each pointing to the start of a segment. The dope vector itself is a dynamic array that is allowed to grow in size
- *segment*: A contiguous array that stores the actual elements. In a sliced array, each segment has a fixed length of K

Given a handle D, the v-th element of the array can be accessed via $D[v/K][v\%K]$, where the "/ " symbol represents integer division and the "%" symbol represents the modulo operator. If K is a power of 2, then $v/K, v\%K$ can be efficiently computed with bit-wise operations.

To append an element, the element is added to the end of the last segment if it is not full. If the segment is full, a new segment of size K is allocated using $malloc()$, and the element is added to this new segment. A pointer to this new segment is created in the dope vector. If the dope vector is full, it is resized using $realloc()$ to a larger memory space, usually increasing geometrically, and the handle is updated to point to this new memory space. Overall, the time complexity of the $store()$ is amortized $O(1)$.

2.4 Hashed array tree (HAT)

The Hashed Array Tree (HAT) is an extensible array proposed by Sitarski [4]. It contains a handle, a dope vector, and several equal-sized segments, which are very similar to the sliced array. The HAT maintains a variable B, where $\sqrt{N} \leq B \leq 2\sqrt{N}$. The dope vector and segments are both of size B. When inserting a new element, we insert it into the last segment if it is not full. If it is full and there are less than B segments, we create

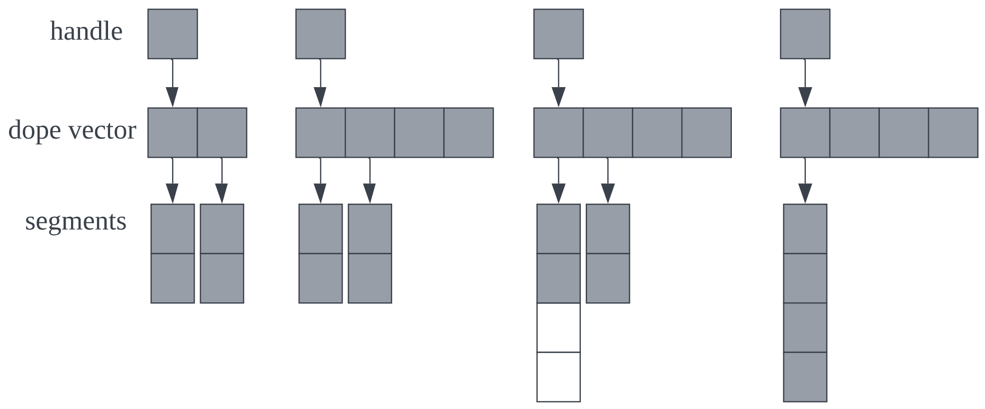


FIGURE 2.2: Rebuilding process when $B = 2$, going sequentially from left to right. Grey blocks are allocated, and white blocks are empty.

a new segment of size B and insert the element into it. If there are already B segments, we rebuild the data structure by doubling B.

The dope vector incurs a $O(B)$ space overhead, while a segment incurs a $O(B)$ space overhead as well when it is created. Since $B \leq 2\sqrt{N}$, the total space overhead is $O(\sqrt{N})$.

The rebuilding process can also be done with $O(\sqrt{N})$ space overhead. First, double the capacity of the dope vector with *realloc()*. Then, for every second segment (starting from the first), we double its capacity, and copy the next segment into the newly reallocated space of the previous segment. At any time, the space overhead required is $O(B) = O(\sqrt{N})$. Although the rebuilding operation is $O(N)$, it is rarely called, and hence the amortized time for the $store()$ operation remains $O(1)$.

Let D be the handle, then we can access the vth element of the array via $D[v/B][v\%B]$, which is similar to the sliced array, but with K replaced by B. Again, if B is always a power of 2, then this can be computed easily with bit-wise operations.

2.5 SQarray

The SQarray also has the 3 components that the sliced array has. However, its segments have a variable length as opposed to the constant length of K of the sliced array. The length of the i-th segment is determined by the following function:

$$L(i) = \begin{cases} 2 & \text{if } i < 2 \\ 2^{\lfloor \log_2 [(i+1)/3] \rfloor + 1} & \text{if } i \geq 2 \end{cases}$$

Algorithm 1 Computing the mapping from a one-dimensional array index v to a two-dimensional segment number and offset pair in SQarray. Taken from Moffat and Mackenzie [1]

```
function $mapping(v)$
    if $v = 0$ then
        set $b \leftarrow 1$
    else
        set $b \leftarrow (33 - \texttt{clz}(v)) \gg 1$
    set $segnum \leftarrow (v \gg b) + (1 \ll (b - 1)) - 1$
    set $offset \leftarrow v \,\&\, ((1 \ll b) - 1)$
    return $\langle segnum, offset \rangle$
```

Thus, the segment sizes follow the sequence $2, 2, 4, 4, 4, 8, 8, ...$

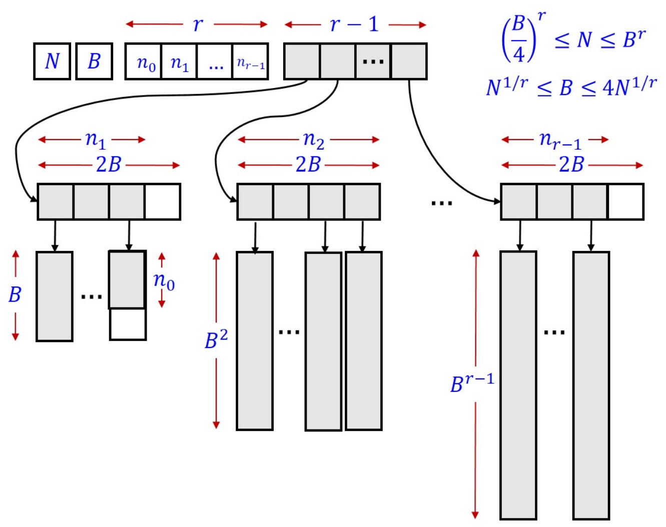
Given a handle D, the v-th element of the array can be accessed using $D[segnum][offset]$, where *segnum* and *offset* are computed with algorithm 1. Due to this complex mapping operation, the SQarray has a higher address processing overhead compared to the sliced and monolithic arrays.

2.6 Tarjan and Zwick's resizable array (Multisliced)

Tarjan and Zwick [2] proposed a novel way of constructing a *resizable* array with $O(N^{\frac{1}{r}})$ space overhead and $O(N^{1-\frac{1}{r}})$ instantaneous space overhead, where $r \geq 2$ is a constant parameter that the array maintains. A resizable array can *grow* or *shrink* in size, but for the purpose of this study, the focus is only placed on the growing aspect of the data structure. Thus, our implementation has slight differences from Tarjan and Zwick's original proposal. Since Tarjan and Zwick did not provide a name for their array, it is referred to as the *multisliced* array.

The array maintains a variable B with $N^{\frac{1}{r}} \leq B < 2N^{\frac{1}{r}}$, and it keeps $r - 1$ handles, pointing to dope vectors of size B. When the $r - 1$ handles are viewed separately, their structure is similar to $r - 1$ sliced arrays, with the handle i (0-indexed) having segments of length B^{i+1}. These sliced arrays are referred to as "blocks". When $r = 2$, the multisliced array is nearly identical to the HAT, with some slight nuances.

New elements are always inserted into the first block with segments of size B. The insertion logic is similar to that of the sliced array, but when the first block contains B full segments, we create a new segment of size B^2 in the second block and transfer the contents of the first block into this new segment. Then, we can continue to fill in the first block, which now becomes empty. When the second block is full, a segment of size B^3 is created in the third block, and the entire second block is copied into this segment,



r
r − 1
N
B
n_0 n_1 ... n_{r-1}
$\left(\frac{B}{4}\right)^r \leq N \leq B^r$
$N^{1/r} \leq B \leq 4N^{1/r}$
n_1
2B
n_2
2B
n_{r-1}
2B
B
n_0
B^2
B^{r-1}

Figure 2.3: General structure of the multisliced array as a resizable array. There are some differences between the figure and what we described, since we tweaked the structure to use it only as an extensible array. Source: Tarjan and Zwick [2]

and so on. Note, this meant that only the first block can contain segments that are partially filled, while the rest of the blocks contain segments that are always full. When $N = B^r$, we double B and rebuild the data structure.

Since the multisliced array has complex operations, we will discuss the specifics of each type of operation in the following sections.

2.6.1 Access operator

Let D be the pointer to the handles. For each $i \in [0, r-2]$, $D[i]$ represents handle i (or block), and for each $j \in [0, B-1]$, $D[i][j]$ points to the j-th segment in this block. Finally, for each $k \in [0, B^{i+1}-1]$, $D[i][j][k]$ represents the k-th element in this segment.

We define n_i as the number of full segments with size B^i, and let n_0 be the number of elements in $D[0][n_1]$ (i.e., the number of elements in the non-full segment of the first block). Using this definition, we can express N in base B as $(n_0, n_1, \ldots, n_{r-1})_B$. We also define $N_k = (0, 0, ..., n_k, ..., n_{r-1})_B = \sum_{j=k}^{r-1} n_j B^j$ (note that $N_r = 0$). We maintain $N_0, N_1, \ldots, N_r$ within the data structure. Additionally, assuming that B is a power of 2, we define $index = \log_2(B)$.

Assuming we record N_i $\forall i \in [0, r], n_1, index$, then we can access the v-th element of the array as $D[block][segnum][offset]$, where $block, segnum, offset$ are computed using algorithm 2.

Algorithm 2 Computing the mapping from a one-dimensional array index v to a three-dimensional output in the multisliced array

```
function $mapping(v)$
    set $k \leftarrow (31 \oplus \texttt{clz}(v))/index$
    set $B^k \leftarrow 1 \ll (index * k)$
    set $mask \leftarrow B^k - 1$
    if $k = 0$ then
        set $block \leftarrow 0$
        set $segnum \leftarrow n_1$
        set $offset \leftarrow (v - N_{k+1}) \gg (index * k)$
    else
        set $block \leftarrow k - 1$
        set $segnum \leftarrow (v - N_{k+1}) \gg (index * k)$
        set $offset \leftarrow (v - N_{k+1})$ & $mask$
    return $\langle block, segnum, offset \rangle$
```

The $mapping(v)$ function could be performed in constant time, hence the $get()$ operation can be performed at worst-case $O(1)$ time.

2.6.2 *combine_blocks*()

When a block becomes full, a process called *combine_blocks*() is initiated to move the elements to the next block. This process starts from the first block and recursively moves full blocks to the following block. Specifically, when block i reaches its maximum capacity of $B \times B^{i+1} = B^{i+2}$ elements, a new segment of size B^{i+2} is created in block $i+1$, and the contents of the block are transferred to this new segment. Throughout this process, the largest segment that can be generated is B^{r-1}, resulting in an instantaneous space overhead of $O(B^{r-1}) = O((N^{\frac{1}{r}})^{r-1}) = O(N^{1-\frac{1}{r}})$. In the scenario where the last block becomes full, indicating that the array has contained B^r elements, a *rebuild*() operation is triggered. The specifics of the *rebuild*() process will be discussed in Chapter 3.

2.7 Existing comparison

Previous research conducted by Katajainen [5] provides a relevant comparison to our own findings regarding the performance of the sliced array and the SQarray. Katajainen's experiments revealed that the sliced array is faster compared to the SQarray across various operations, including introsort (a hybrid of quicksort and heapsort), heapsort, random access, and repeated append.

2.8 Summary of complexity

The time complexity of $store()$ and $get()$ is shown in table 2.1. The superiority of the SQarray in terms of the worst-case complexity of the $store()$ operation is clearly observed.

Array type	$store()$, amortized	$store()$, worst-case	$get()$, worst case
Geometric	$O(1)$	$O(N)$	$O(1)$
Sliced	$O(1)$	$O(N)$	$O(1)$
SQarray	$O(1)$	$O(\sqrt{N})$	$O(1)$
HAT	$O(1)$	$O(N)$	$O(1)$
Multisliced	$O(1)$	$O(N)$	$O(1)$

TABLE 2.1: Time complexity of $store()$ and $get()$ in different type of array

To understand the space overhead of the extensible arrays, we will first distinguish between *instantaneous* and *persistent* space overhead. The instantaneous overhead is the peak space overhead that is incurred within the $store()$ operation. The persistent space overhead is the space overhead that remains even after the $store()$ call finishes. For example, in a geometric array, the largest instantaneous space overhead occurs when it needs to be reallocated to a bigger space. Since a segment of size $\approx N \times k$ is created during this process and is initially unused, the instantaneous space overhead at this instant is roughly $N \times k$. However, once the N elements are copied to this new segment, the original segment of size N is freed, and the amount of unused space in the new segment would be $(k-1) \times N$. This would be our persistent space overhead after the $store()$ operation. Table 2.2 highlights the space overheads of different types of extensible arrays.

Array type	instantaneous	persistent
Geometric	$O(N)$	$O(N)$
Sliced	$O(N)$	$O(N)$
SQarray	$O(\sqrt{N})$	$O(\sqrt{N})$
HAT	$O(\sqrt{N})$	$O(\sqrt{N})$
Multisliced	$O(N^{1-\frac{1}{r}})$	$O(N^{\frac{1}{r}})$

TABLE 2.2: Space overhead of different types of arrays

Chapter 3

Method and Results

This chapter discusses how each type of array is implemented in C, and the results we obtain when we run these extensible arrays against different types of experiments.

3.1 Array implementations

To compare the performance of different extensible arrays, we have implemented them in C code and created one .c file for each type of array. Each data structure (other than the base array) implements the "interface.h" interface, which contains general functions of the array including $store()$ and $get()$.

The specific structure of each type of array we will be using is:

- **Base**: A standard static C array with a size determined at compile time.
- **Monolithic**: Contains a handle that points to a single segment, with the segment's size determined at initialization. The implementation has about 70 lines of code.
- **Sliced**: Contains a handle that points to a dope vector, which in turn points to segments that have a fixed size of $K = 16$ elements. The size of the dope vector increases geometrically with a growth parameter of 2. The implementation has about 90 lines of code.
- **SQarray**: Contains a handle that points to a dope vector, which points to segments of variable size as described in Chapter 2. The size of the dope vector also increases geometrically with a growth parameter of 2. The implementation has about 140 lines of code.

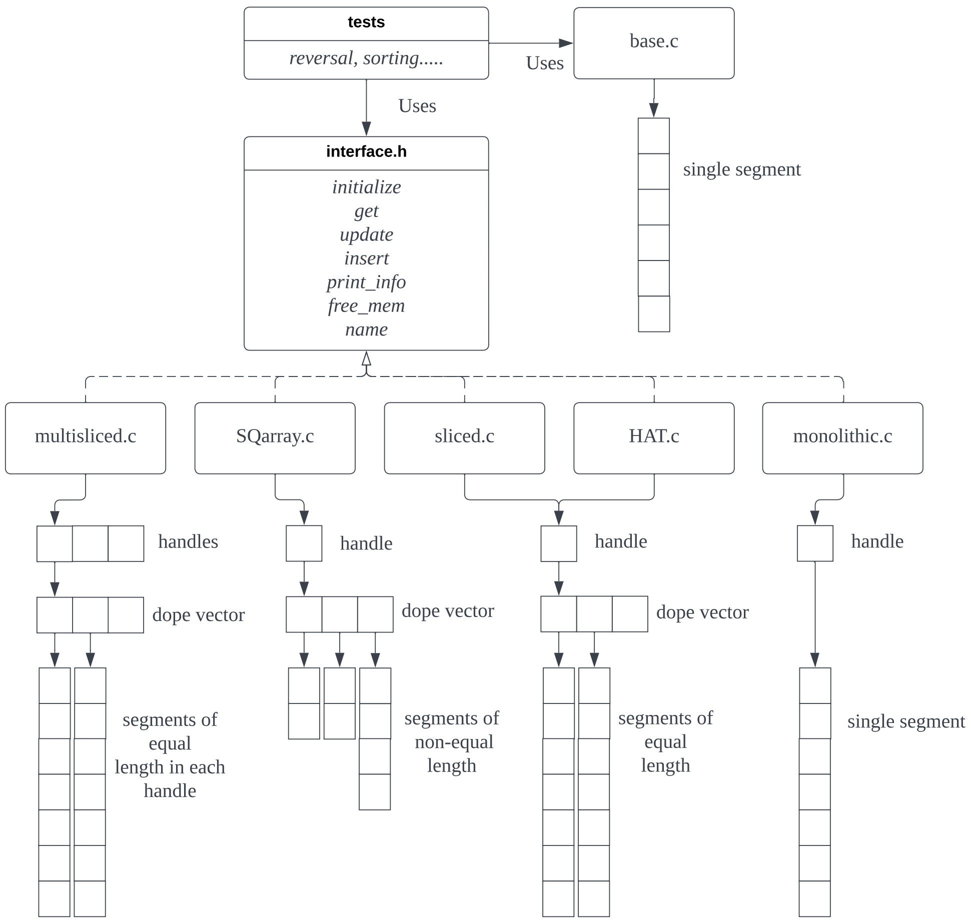


Figure 3.1: The structure of the code base

- **HAT**: Contains a handle that points to a dope vector and maintains a variable B. The dope vector has a length of B and points to segments of size B. It uses the $rebuild()$ function when the array is full. The implementation has about 120 lines of code.

- **Multisliced**: The multisliced array is initialized with a parameter r and maintains a variable B. It contains $r-1$ handles, each pointing to a dope vector, which in turn points to segments of equal size. It uses the $combine\_blocks()$ functions when a block is full, and the $rebuild()$ function when $N = B^r$. The implementation has about 230 lines of code.

It's worth noting that our implementation of the array presented by Tarjan and Zwick is simplified. Specifically, we set the initial B value to 8 and assumed that $B \geq 2^{r-1}$ at all times, greatly simplifying the rebuilding process, during which all segments would fit into the last block, making all other blocks empty. However, this assumption means that our implementation fails when $r = 5$, since initially $B = 8 < 2^{r-1} = 16$. As a result, we only evaluated the performance of the multisliced array when $r = 2, 3, 4$. We consider this simplification to be suitable since the persistent space overhead of a multisliced array with $r = 4$ is $O(N^{1/4})$, so it is already sufficient for many real-world applications, even when dealing with significantly large values of N.

3.2 Repeated Append

This section examines the performance of the $store(v, x)$ operation, specifically focusing on appending elements to the end of the array. This is achieved by setting $v = N$ and performing the $store(N, x)$ operation. In contrast to other tasks analyzed in the next section, appending involves adding elements to the array, while the other tasks utilize pre-populated arrays without introducing new elements. The objectives of this section are to address the following three questions:

- How does the running time of repeated append change for each array type, when the number of elements is kept fixed but the size of elements varies?
- How does the running time of repeated append change for each array type, when the size of elements is kept fixed but the number of elements varies?
- How does the space used by each array type change as more elements are appended to the array?

Experiments A and B are designed to answer the first question. The second question will be addressed with experiment C, and the third question will be explored through experiment D.

3.2.1 Experiment A: Making space

The $store(v, x)$ operation consists of two parts: *make_space*(v, s) and *copy*(v, x). The *make_space*(v, s) operation ensures that there is at least $s =$ *size_of(x)* bytes of allocated space at index v, possibly by using operations like *malloc*() or *realloc*(). The *copy*(v, x) operation copies the contents of element x into index v. When $v \neq N$, the *make_space*(v, s) operation is trivial because there is already an element at index v, and therefore the required space is already allocated (assuming all elements have the same size). However, when $v = N$, the *make_space*(N, s) operation becomes non-trivial. *make_space*(N, s) ensures that there is enough space to insert another element into the array and the specific logic depends on the array type. In this experiment, we will examine the running time of *make_space*(N, s) for each type of array, focusing on how it

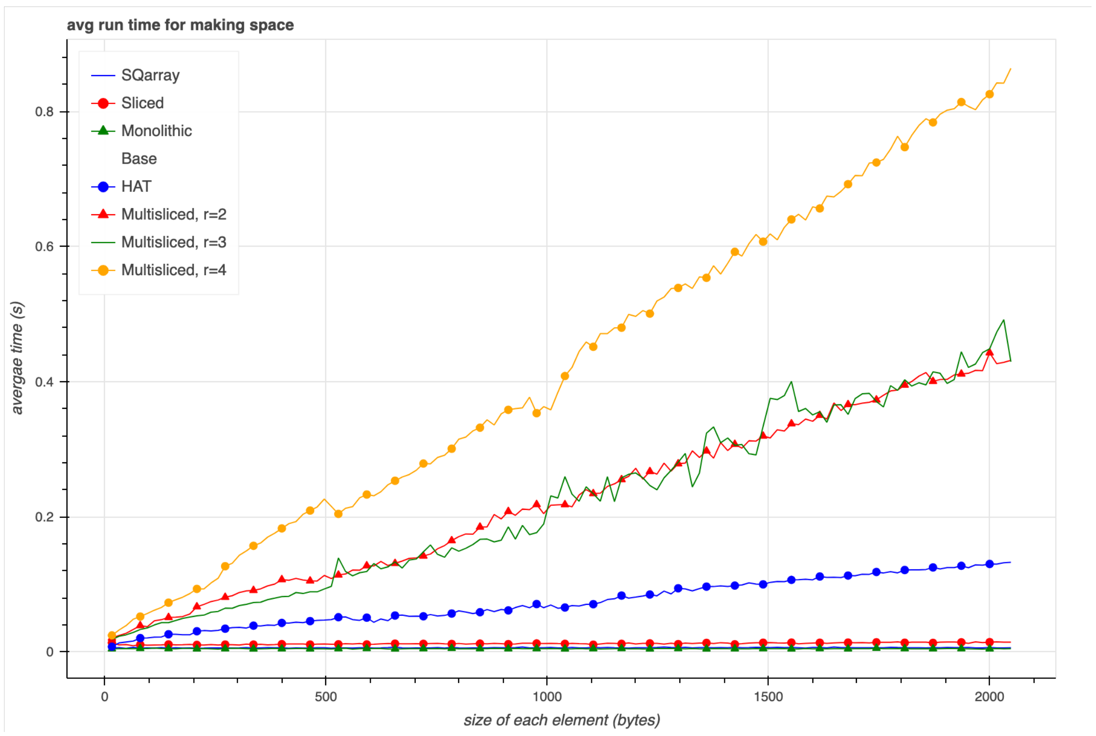


FIGURE 3.2: Average running time for 2^{20} *make_space*() operation, across 5 trials

varies with different element sizes.

Experimental Procedure:

1. Set the element size to be $s = 16$ bytes.
2. For each type of array, measure the time it takes to perform the $make\_space(N, s)$ operation 2^{20} times.
3. Increment the size of the elements by 16 bytes and repeat step 2. Stop when the size of the element reaches 2048.
4. Repeat steps 1-3 five times and record the average running time.

In this context, we define N as the number of times we called *make_space*(). Each $make\_space(N, s)$ operation can be seen as appending "nothing" to the array, incrementing the counter N by 1. By making space 2^{20} times, this allow $store(v, x)$ to be performed for index $0 \leq v \leq 2^{20}$.

Figure 3.2 shows that the sliced, SQarray, and monolithic arrays have a constant time for making space. This is because these array types use inexpensive *malloc*() operations to allocate space to segments and *realloc*() operations on the dope vector. The dope vector contains pointers with a constant size of 8 bytes, so the time for *make_space*()

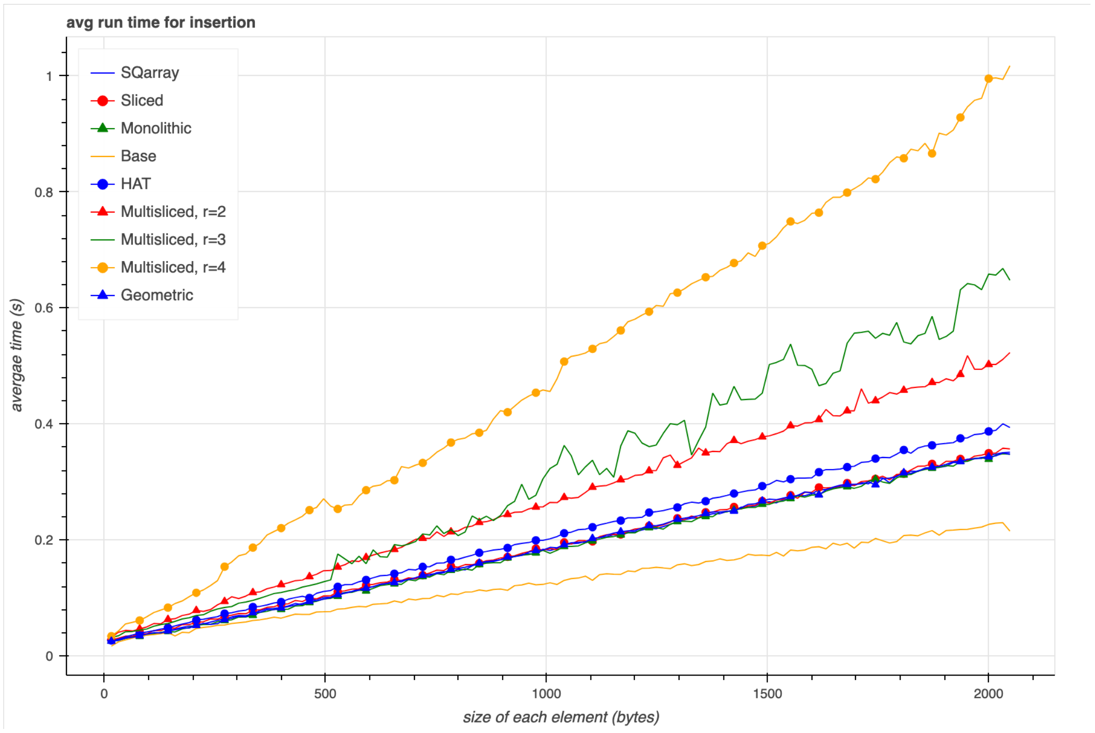


FIGURE 3.3: Average running time for inserting 2^{20} random elements into the array, across 5 trials

does not scale with the element size.

However, the HAT and multisliced arrays demonstrate a linear trend in running time as the size of each element increases. This is due to the occasional *rebuild*() process that both array types have to perform. Even though we did not actually insert any elements into the spaces that were made, the array assumes the presence of elements and performs the entire copying logic within the *rebuild*() operation. This operation scales linearly with the size of each element. Additionally, the multisliced arrays require the *combine_blocks*() operation, which also scales linearly with the element size.

3.2.2 Experiment B: Appending with fixed array size, variable element size

This experiment focuses on the running time of repeated $store(N, x)$ operations, which combines the $make\_space(N, s)$ and $copy(N, x)$ operations. Each element x consists of an 8-byte *key* and *data*. The procedure for this experiment is similar to Experiment A. However, instead of calling $make\_space(N, s)$ in step 2, we call $store(N, x)$, where the key of each element x is a random integer in the range $[0, 2^{20} - 1]$, generated using the *rand*() function with the seed 19284729. The remaining bits of x are filled with 1s.

Based on Figure 3.3, it can be observed that the running time generally follows a linear trend. The array types maintain a similar ranking in terms of performance in line with the observations made in experiment A. The base array, serving as the baseline, demonstrates the best performance. The geometric, monolithic, sliced, and SQarray arrays exhibit similar performances, with the sliced array performing slightly worse. The HAT array shows worse performance compared to these four array types, while the three multi-sliced arrays exhibit the poorest performance.

In summary, these results demonstrate that the multisliced array has a significantly higher constant overhead than other arrays when it comes to insertion. On the other hand, the geometric, sliced, and SQarray display behavior comparable to the monolithic array, and the HAT exhibits slightly worse overhead.

3.2.3 Experiment C: Appending with fixed element size, variable array size

This experiment examines the running time to append elements into the array, with varying array size, N.

N	2^{10}	2^{15}	2^{20}	2^{25}
base	0.012	0.345	11.0	352
monolithic	0.017	0.406	12.6	405
SQarray	0.020	0.401	12.6	403
geometric	0.022	0.607	12.9	405
HAT	0.019	0.432	13.1	447
sliced	0.021	0.450	14.4	462
multisliced, $r = 2$	0.028	0.463	16.4	468
multisliced, $r = 3$	0.033	0.678	15.5	482
multisliced, $r = 4$	0.037	0.924	20.1	578

TABLE 3.1: Average running time (ms) to append N elements of size 16 bytes, across 20 trials

Experimental Procedure:

1. Set the size of the array, $N = 2^{10}$
2. For each type of array, measure the time it takes to append elements of size 16 bytes into an array of size N
3. Repeat step 2 for $N = 2^{15}, 2^{20}, 2^{25}$
4. Repeat steps 1-3 for 20 times, and record the average running time.

Table 3.1 presents the results of the experiment. It is evident that the multisliced arrays perform the poorest, with their performance deteriorating as the value of r increases. On the other hand, the SQarray consistently demonstrates comparable or even better performance than the monolithic array, which serves as the baseline.

The relatively poor performance of the multisliced array and the HAT array can be attributed to their worst-case insertion time of $O(N)$, which involves costly operations like *rebuild*(). Similarly, the sliced array's underperformance can be attributed to its $O(N)$ worst-case *store*() time, specifically when using *realloc*() to increase the size of the dope vector. Furthermore, the sliced array requires a large number of *malloc*() calls, totaling $O(N)$, surpassing other array types. As *malloc*() is a system call with high overhead, it significantly impacts the sliced array's performance, causing it to perform even worse than the HAT. In contrast, the SQarray stands out as the top performer among all extensible arrays. It achieves this by having a worst-case insertion time of $O(\sqrt{N})$ and utilizing significantly fewer *malloc*() calls compared to the sliced array, amounting to only $O(\sqrt{N})$ calls.

3.2.4 Experiment D: Space overhead

To answer the third question regarding space overhead, we conducted an experiment to record the peak space overhead of each array type after appending elements with a size of 8 bytes. A total of 2^{14} elements were inserted into each array.

The observed space overhead curves of the sliced, geometric, and SQarray arrays align with the findings of Moffat and Mackenzie [1], as depicted in Figure A.1.

Figure 3.4 illustrates the space overhead curve of the multisliced array with $r = 4$, showcasing the clear distinction between the persistent and instantaneous space overhead. In the graph, the spikes that immediately decrease represent the instantaneous space overhead. Two types of spikes are evident: a larger spike occurs when the *combine_blocks*() operation is triggered upon filling the second block, leading to its relocation to a newly created segment of size B^3 in the third block. The smaller spike corresponds to the first block being filled, and relocated to a new segment of size B^2 in the second block. The saw-like steps in the curve reflect the gradual filling of the segment of size B in the first block, representing the persistent space overhead.

The experimental results showed that for multisliced arrays, the instantaneous overhead increases with higher values of r, while the persistent overhead decreases. This aligns with the theoretical expectations outlined by Tarjan and Zwick [2], where a multisliced

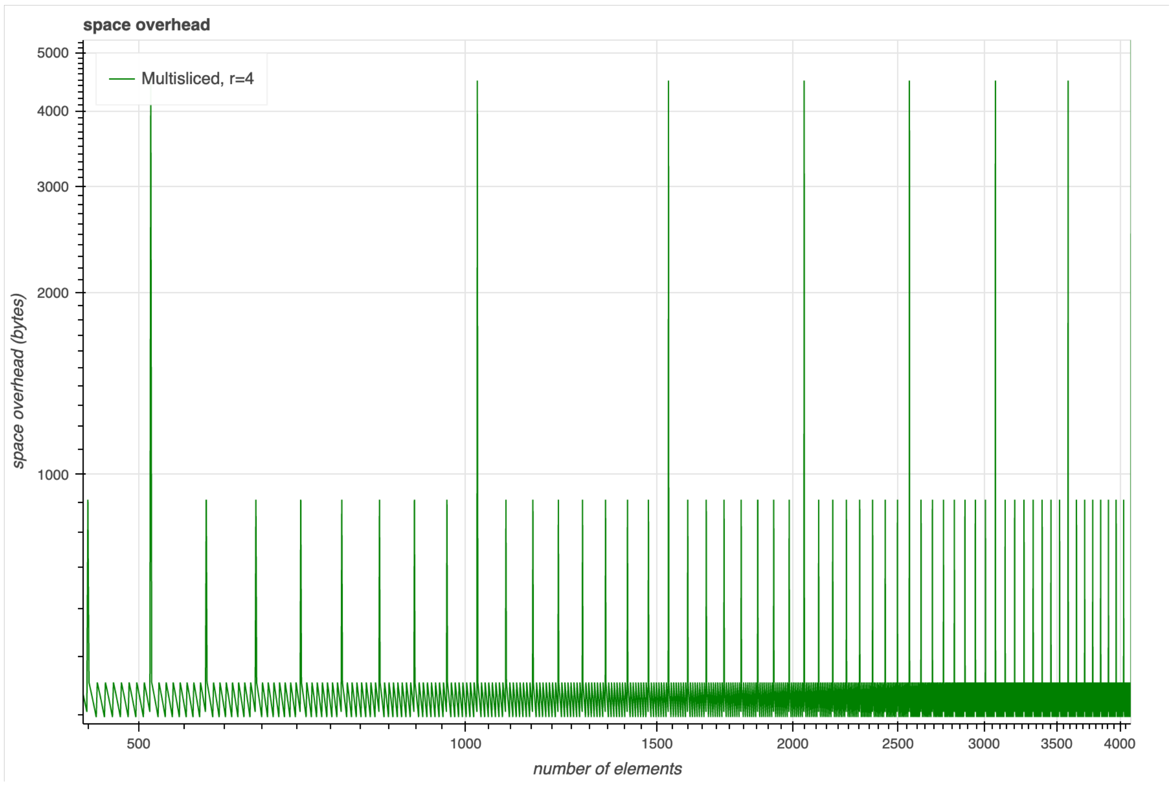


FIGURE 3.4: Space overhead of multisliced array with $r = 4$

array with parameter r should exhibit $O(N^{\frac{1}{r}})$ persistent space overhead and $O(N^{1-\frac{1}{r}})$ instantaneous space overhead. Further details can be found in Figure A.2.

Figure 3.5 presents a comparison of the space overhead between the SQarray, HAT, and multisliced arrays. We exclude the sliced and geometric arrays as they have a space overhead of $O(N)$, which is worse than the persistent and instantaneous space overhead of the other extensible arrays. The HAT and multisliced array with $r = 2$ have identical space overhead curves, so we do not include the latter in the graph. The SQarray and HAT demonstrate similar performance, with the HAT occasionally exhibiting lower space overhead. Both arrays have a higher persistent space overhead but a lower instantaneous space overhead compared to the multisliced arrays. Notably, the multisliced array with $r = 4$ showcases the lowest persistent space overhead but also has the highest instantaneous space overhead. These results are consistent with the theory presented in Table 2.2.

In conclusion, the multisliced array with $r = 4$ demonstrates the lowest persistent memory overhead, and we anticipate even lower persistent memory overhead as r increases. On the other hand, HAT, SQarray, and the multisliced array with $r = 2$ exhibit the lowest instantaneous space overhead.

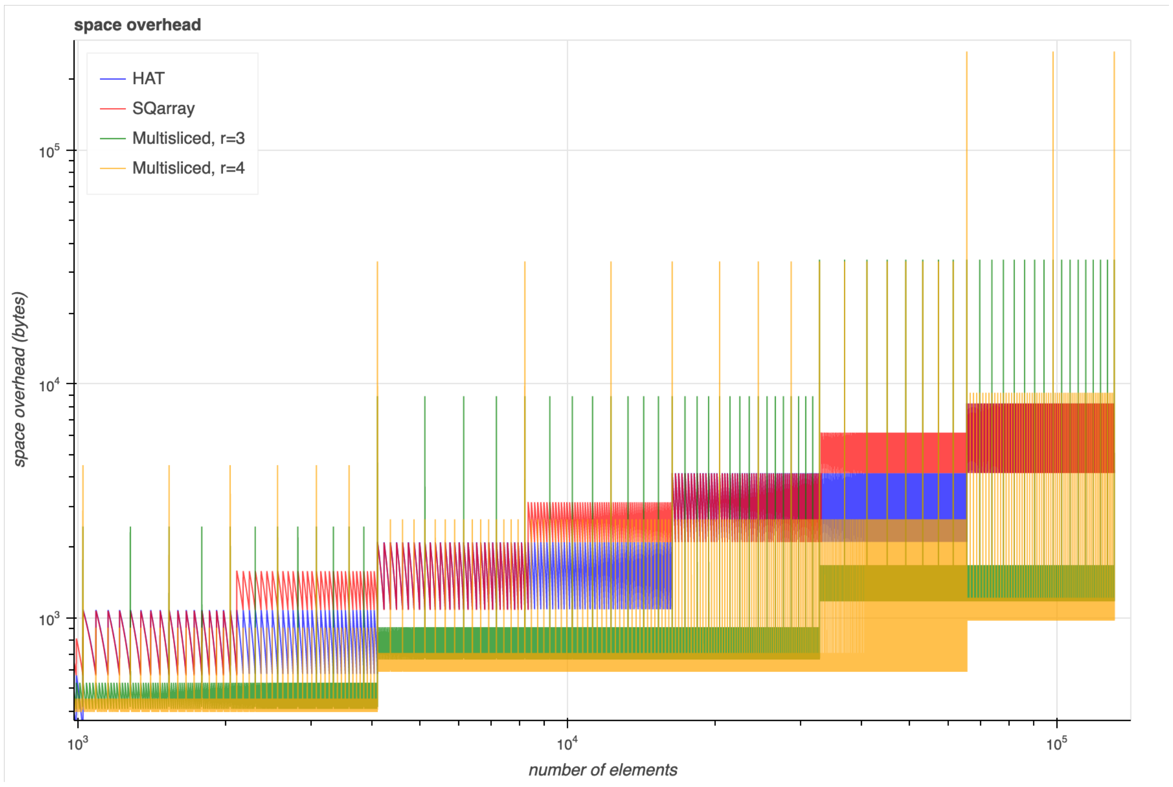


FIGURE 3.5: Comparison of space overhead between SQarray, HAT, multi-sliced arrays

3.3 Running applications

In addition to the repeated append experiments, we conducted a series of experiments to evaluate the performance of each data structure when it is already populated with elements. In the following experiments, we measure the time it takes for each type of array to run a specific task when the size of the array is fixed, but the size of the elements varies. The experimental procedure closely resembles that of experiments A and B, and the detailed process is outlined in Algorithm 3. The *fill_with_elements()* function populates the array with $N = 2^{20}$ elements, using the same procedure as described in experiment B. The *do(array, task)* function is responsible for executing specific tasks on the array. In the following experiments, we will delve into the details of the logic of how each task is performed.

Experiment E focuses on analyzing the running time of the $get()$ operation in isolation, whereas the remaining experiments involve combining the $get()$ operation with other operations, such as swapping two elements in the array. It is worth noting that no new elements are added to the array during these experiments. Consequently, the geometric array is not included in these experiments, as it is identical to the monolithic array when no new elements are inserted.

Algorithm 3 General procedure of running tasks and obtaining the run time of different types of arrays performing the task

```
function $run\_test()$
    for $task \in \{sort, reversal, ...\}$ do
        for $i = 1$ to 5 do
            for $element\_size = 16$ to 2048 step 16 do
                for $type \in \{base, monolithic, sliced, SQarray, HAT, multisliced\}$ do
                    set $array \leftarrow initialize(element\_size, type)$
                    $fill\_with\_elements(array)$
                    set $start\_time \leftarrow get\_time()$
                    $do(array, task)$
                    set $end\_time \leftarrow get\_time()$
                    $record(output\_file, end\_time\text{-}start\_time, element\_size, type, task)$
    return
```

3.3.1 Experiment E: random access

In this task, we measure the times it takes to perform 2^{20} $get(v)$ operations, where $0 \leq v \leq 2^{20} - 1$ is a random integer generated by the $rand()$ function with a seed of 19284729. We implemented the $get(v)$ operation such that it returns the memory address of the element at index v. After each $get(v)$ operation, we read the key of the element from the memory address and add it to a checksum. The checksum ensures the integrity of the experiment and prevents the $get(v)$ operations from being optimized out by the compiler.

From Figure 3.6, we observed the surprising result that the multisliced array with $r = 2$ (abbreviated as MS2) achieves the best performance. Additionally, while the curves of other array types increased linearly with increasing element size, the curve of MS2 remain constant. Both HAT and MS2 outperformed the base array, which is unexpected considering that the $get()$ logic for the base array is much simpler compared to these two extensible arrays.

The reason for the unexpectedly good performance of MS2 and HAT may be attributed to the CPU (Apple M1) used in our experiments. When we repeat this experiment on another computer with Intel i5 CPU, a different result can be observed in table 3.2, indicating that the performance of these two arrays can be hardware dependent. Both multisliced array with $r = 2$ and the HAT now performs worse than the base and monolithic array. The relative performance of the arrays in table 3.2 is in line with our expectations, as we anticipated that array types with more complex $get()$ logic would result in higher running times.

Apart from the MS2 and HAT, the performances of other types of arrays align with our findings in Figure 3.6: the multisliced arrays with $r = 3, 4$ exhibited the worst

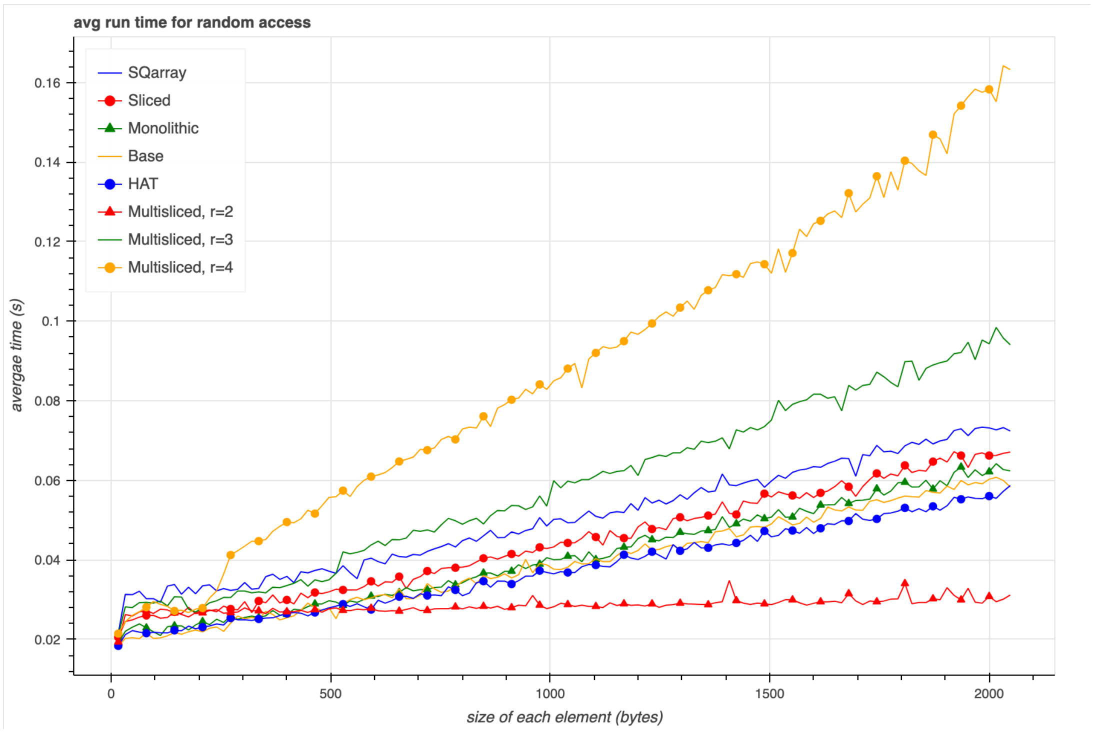


FIGURE 3.6: Average running time for randomly accessing 2^{20} elements in the array, across 5 trials

element size	2^7	2^8	2^9	2^{10}
base	26.5	32.4	45.5	71.0
monolithic	30.7	38.0	47.4	63.9
SQarray	46.8	51.7	61.7	79.0
HAT	37.0	44.5	54.3	76.4
sliced	43.0	48.1	55.8	72.7
multisliced, $r = 2$	53.0	62.0	77.5	105
multisliced, $r = 3$	50.3	62.1	98.8	141
multisliced, $r = 4$	60.0	82.8	122	240

TABLE 3.2: Average running time (ms) of random access across 20 trials, on Intel i5 processor

performance, the SQarray performed worse than the sliced array, which in turn was slightly worse than the monolithic array.

3.3.2 Experiment F: array reversal

In this task, we measure the time required to reverse an array of 2^{20} elements. The performance ranking of different array types is consistent with that observed in Experiment E, as shown in Figure 3.7. Once again, the multisliced array with $r = 2$ achieves the best performance, followed by the HAT. Both arrays outperform the base array in terms of running speed. The base, monolithic, sliced, and SQarray display similar performance levels, while the multisliced arrays with $r = 3, 4$ perform the worst.

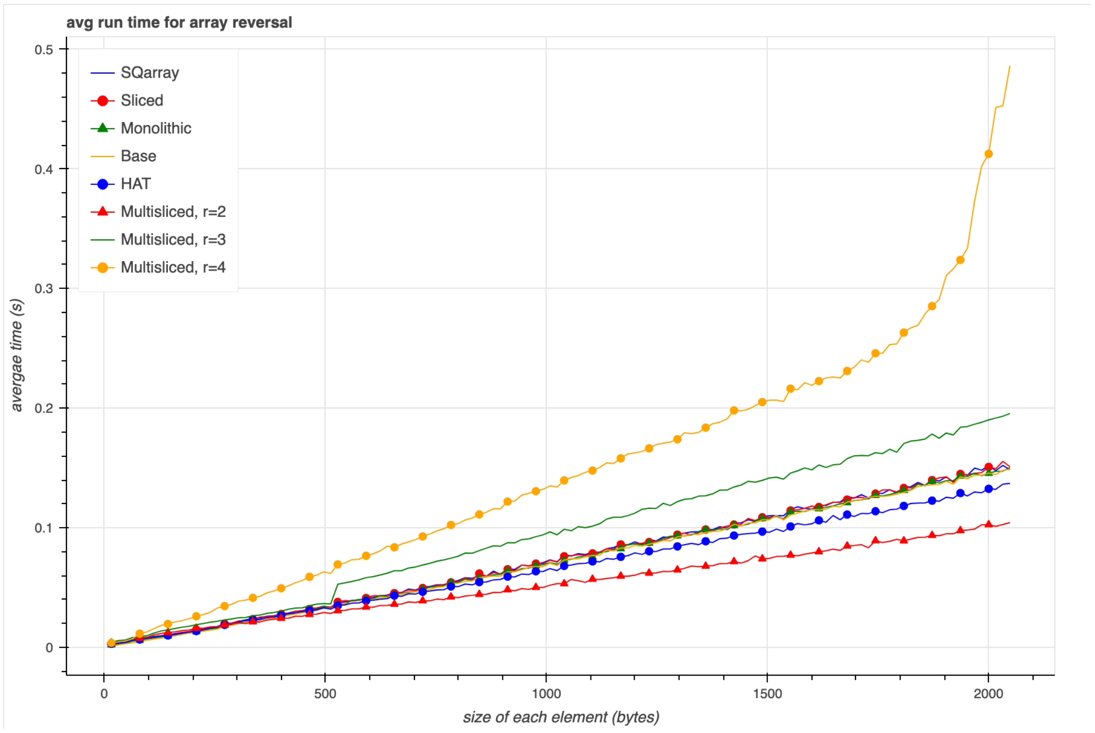


FIGURE 3.7: Average running time for reversing an array with 2^{20} elements, across 5 trials

The overall trend remains linear, with the performance of the multisliced array ($r = 4$) significantly deteriorating at larger element sizes. This similarity to Experiment E was expected since array reversal involves repeatedly using the *get*() operation to retrieve two elements and then swapping them, amounting to $N = 2^{20}$ *get*() operations in total. As a result, the overall logic of this experiment is similar to that of repeated random access. Once again, this result is heavily influenced by the hardware.

3.3.3 Experiment G: quicksort

In this task, we measured the time required to sort an array of 2^{20} elements using quicksort. This experiment mainly examines the efficiency of sequential access.

Figure 3.8 reveals that all three multisliced arrays exhibit the worst performance. The performance of the SQarray, sliced array, and HAT is quite similar. Notably, the anomaly observed in experiments E and F, where MS2 and the HAT outperformed all other arrays, has disappeared. As quicksort involves a significantly higher number of *get*() calls and element swaps compared to the previous two experiments, this suggests that in larger applications, the hardware optimizations may not provide the same advantages for MS2 and HAT, causing them to revert back to their expected behavior.

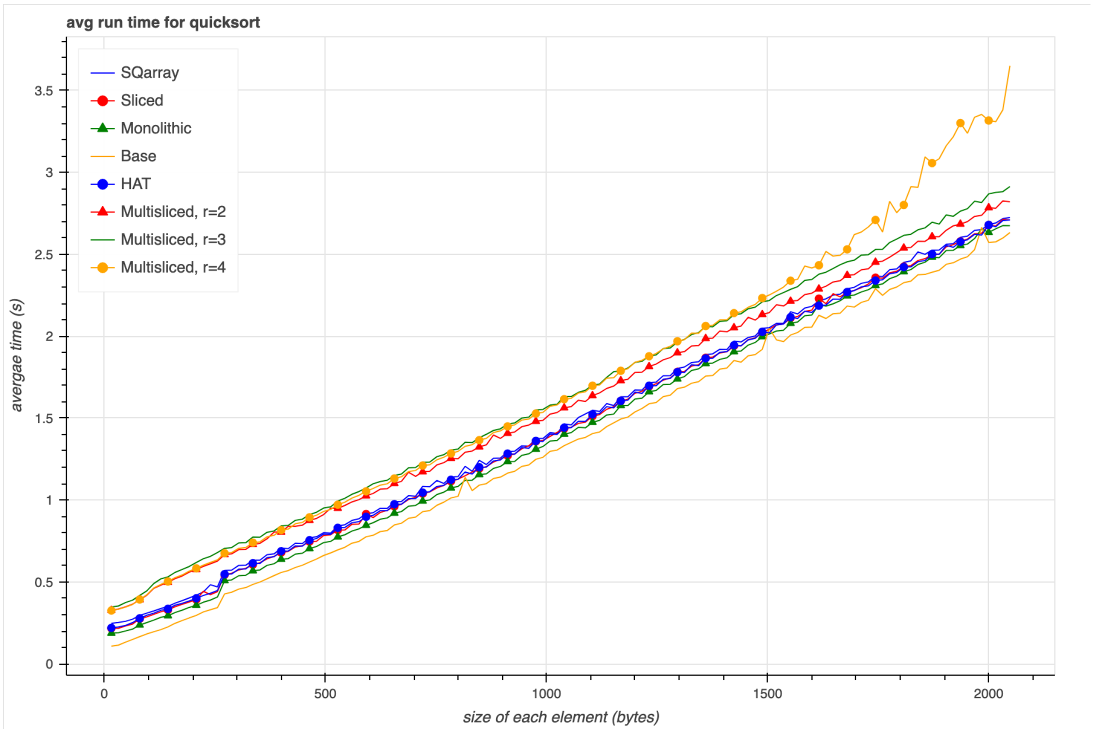


FIGURE 3.8: Average running time for sorting an array with 2^{20} elements with quicksort, across 5 trials

An overall linear trend is observed. However, contrary to the previous experiments, the curves in Figure 3.8 have the same gradient. This indicates that the running times of the different array types are only separated by a constant factor.

3.3.4 Experiment H: heapsort

In this task, we measured the time required to sort an array of 2^{20} elements using heapsort. This experiment mainly examines the efficiency of random access.

Figure 3.9 displays a similar trend compared to experiment G, where the multisliced arrays are still the worst performers, and the HAT still performed the best out of all extensible arrays. However, the SQarray performed better than the sliced array in this experiment.

3.4 Hardware and Software

The code was written in C and compiled with **gcc**, using -O3 optimization. The computer used to run the programs is MacBook Pro (13-inch, M1, 2020), with Apple M1 CPU and 8GB RAM. The code used to produce the experimental results can be found in https://github.com/zblwwwzbl/Extensible-array

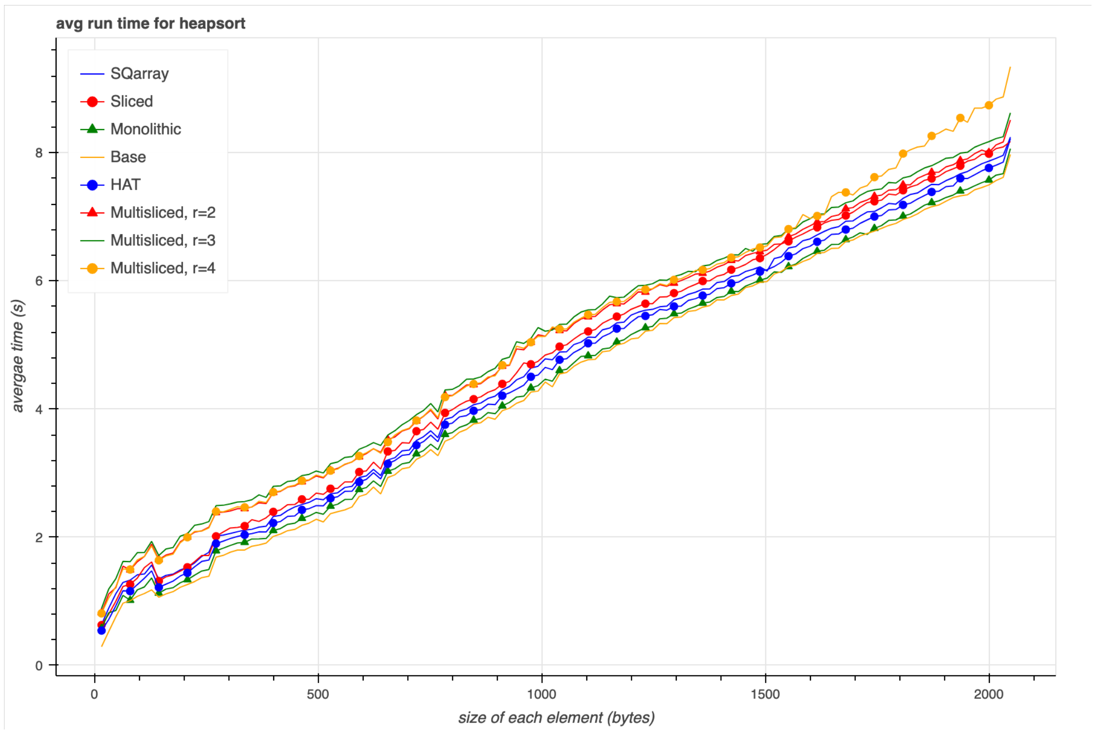


FIGURE 3.9: Average running time for sorting an array with 2^{20} elements with heapsort, across 5 trials

Chapter 4

Discussion

In this chapter, we will highlight the conclusions made during the experiment, and offer some paths for future work.

4.1 Comparison to previous work

Previous work by Katajainen [5] investigated various extensible arrays and compared their performance in experiments where the element size remained fixed, while the array size varied. Their findings showed that the sliced array consistently outperformed the SQarray in all experiments. However, our experimental results differ from Katajainen's observations.

In experiment C, we observed that the SQarray outperformed the sliced array across all array sizes, contrary to Katajainen's findings for the same experiment. This discrepancy can be attributed to several factors. First, we used larger elements of 16 bytes, whereas Katajainen used smaller elements of 4 bytes. Second, Katajainen used an Intel i5 CPU and 3.8GB RAM for their experiment, which is different from our hardware. Additionally, our experiment on heapsort demonstrated that the SQarray outperformed the sliced array. Again, this can be attributed to the use of different hardware and element sizes.

The variations in the results indicate that SQarray tends to perform better in applications where the element size is high. Additionally, these differences may also suggest that the SQarray exhibits improved performance on the newer hardware used in our experiments.

4.2 Recommendations

This section summarises the strength and weaknesses of different types of extensible arrays, and make suggestions as to when each type of array should be used.

4.2.1 Space

The geometric and sliced arrays should be avoided in applications where the size N of the array can be very large, as they have a space overhead of $O(N)$.

If there is ample available memory, it is recommended to use a multisliced array with a high value of the parameter r, such as $r = 4$. This choice minimizes the persistent space overhead, which is $O(N^{\frac{1}{r}})$, the lowest among all the extensible arrays mentioned.

However, if the available memory is limited, it is advisable to avoid multisliced arrays with a high r parameter, due to their high instantaneous space overhead being $O(N^{1-\frac{1}{r}})$. Using the multisliced array in this context may cause a significant amount of page faults, affecting the performance of the running programs. In such cases, the HAT or SQarray would be more suitable alternatives, having only $O(\sqrt{N})$ instantaneous space overhead.

4.2.2 Time

For applications that involve frequent $store()$ operations, the SQarray is recommended. It has demonstrated superior performance in experiments B and C, which specifically involve repeated append tasks with increasing element or array sizes. In those two experiments, the SQarray shows a really similar performance compared to the monolithic array, which is one of the baselines. Notably, the SQarray has a worst-case time complexity of $O(\sqrt{N})$ for $store()$, which is significantly better than the $O(N)$ worst-case complexity of other extensible arrays. Additionally, from experiment A it is observed that the cost of $rebuild()$ and $combine\_blocks()$ operations scales with the element size, so in applications where the element size is high, this further increases the worst-case $store()$ time of the HAT and the multisliced arrays. The low worst-case $store()$ time of the SQarray can alleviate choke points in the program, allowing faster array access after insertion and providing a smoother user experience.

On the other hand, in larger applications where no new elements are added and there are many $get()$ operations, the HAT is the preferred choice. It has shown superior performance in both quicksort (experiment G) and heapsort (experiment H), which measures the speed of sequential and random access. This performance advantage can be attributed to the HAT's simple *get*() logic and its cache-friendly nature, as each HAT segment contains a contiguous block of $O(\sqrt{N})$ elements. Although the HAT also shows a good performance in random access and array reversal, it's important to note that it may be hardware-dependent.

It's worth mentioning that the geometric array actually outperforms the HAT in the quicksort and heapsort experiments. When a task doesn't involve appending new elements, the geometric array behaves identically to the monolithic array, which outperforms all other extensible arrays in experiments G and H. Nonetheless, the geometric array's space overhead of $O(N)$ makes it impractical for large applications with significant memory requirements. Considering this significant trade-off, the geometric array is not recommended.

Based on the results of the previous experiments, it is evident that multisliced arrays consistently perform poorly due to their complex *get*() and *store*() logic. Therefore, in scenarios where speed is crucial, it is advisable to avoid using multisliced arrays. However, as observed in experiments G and H, the performance difference between multisliced arrays and other array types is only a constant factor. In both experiments, the multisliced arrays are less than 0.5 seconds slower than the other arrays, suggesting that the trade-off for a slightly longer running time but lower persistent space overhead might still be acceptable in larger applications. If a multisliced array is to be used, using parameter $r = 4$ tends to perform better than using $r = 3$ in experiments G and H, when the element size is below 1500 bytes. This suggested that choosing $r = 4$ may be a better option in complex applications when the element size is relatively low. The multisliced array with $r = 2$ is not recommended, as it is operationally identical to the HAT, but has a worse running time.

4.3 Future work

While we have explored and implemented various types of extensible arrays, there are still unexplored avenues for further improvement and investigation.

4.3.1 Hot start

One potential improvement for the SQarray is to address the issue of small initial segment sizes. Currently, the SQarray starts with segments of very low size, beginning from a size of 2. This can lead to a higher number of *malloc* operations when the array is small. To mitigate this issue, an enhancement could involve starting the segment size at a larger value, denoted as k, and doubling it according to the growth rules of the SQarray (resulting in segment sizes of $k, k, 2k, 2k, 2k, 4k, 4k, ...$). The traditional SQarray can be seen as utilizing $k = 2$. This approach may potentially improve both the performance and cache-friendliness of the SQarray by creating more contiguous blocks of memory within the array.

4.3.2 Multisliced array with $r = \log(N)$

Tarjan and Zwick [2] introduced another implementation of the multisliced array, where the parameter r is allowed to grow. In the previous sections, we've treated r as a constant. By letting $r = \log(N)$, Tarjan and Zwick demonstrated that we can achieve an $O(\log N)$ space overhead. Investigating the performance and characteristics of this array would be valuable.

4.3.3 More experiments

In our study, we focused solely on examining the performance of $store()$ and $get()$ operations in isolation. However, there is room for further exploration by conducting experiments that involve both adding new elements and extensive usage of $get()$. Additionally, we could repeat experiment E to H with fixed element size, but varying array size and observe any difference in results.

Furthermore, our work on extensible arrays could be expanded to include *resizable* arrays, which would introduce the functionality of *remove()*. This would enable our arrays to dynamically shrink in size as needed. By exploring the performance of resizable arrays, we can effectively assess their suitability and effectiveness as *deques*, which are data structures supporting efficient insertion and deletion at both ends.

4.4 Limitations

The results collected from all the experiments are dependent on the hardware, as evident from experiments E and F, and may not be generalizable to other hardware configurations. To address this issue, it would be beneficial to repeat the experiments on various hardware platforms and observe any variations. Furthermore, the C code written for the experiments has the potential for optimization, which could impact the performance of the arrays, and hence the results we collect.

4.5 Conclusion

In conclusion, our research project involved conducting a series of experiments to assess the performance of various extensible array types. Through these experiments, we identified the strengths and weaknesses of each array type. Leveraging these characteristics, we have provided practical recommendations on the most suitable extensible array for different contexts. By considering these findings, developers can make informed decisions regarding the adoption of extensible arrays in their projects.

Appendix A

An Appendix

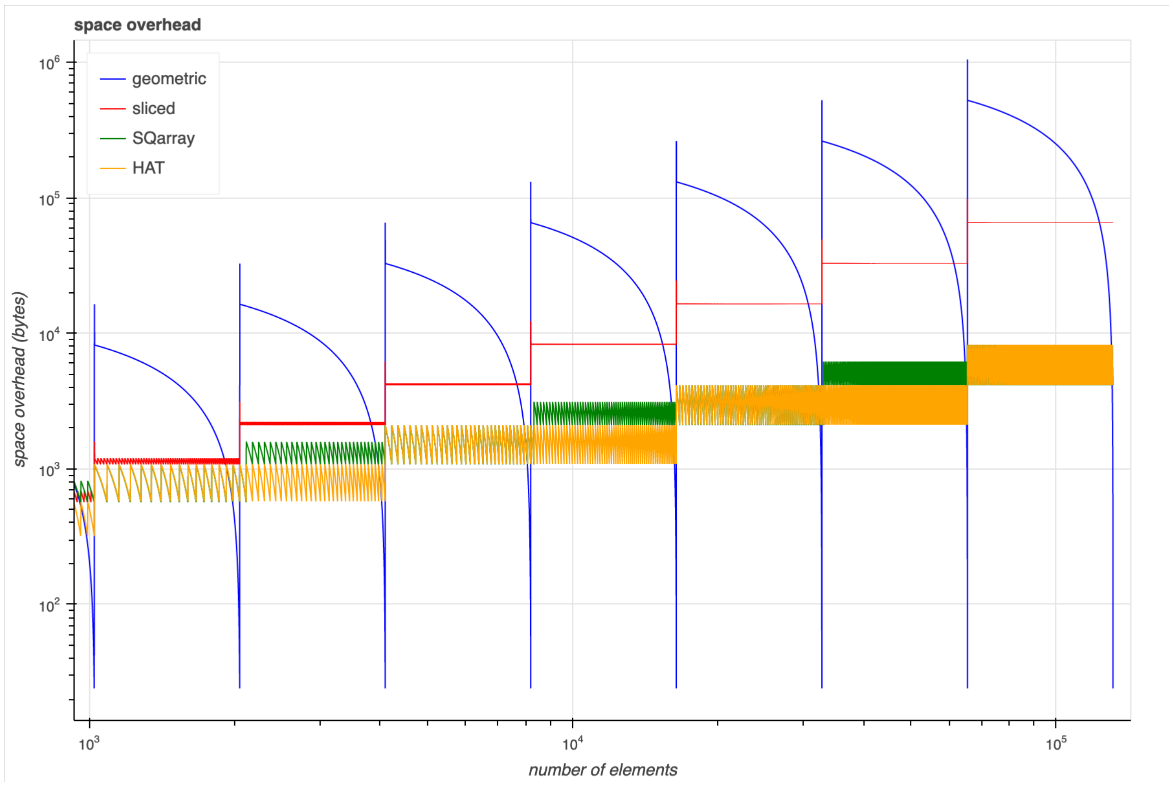


Figure A.1: Comparison of space overhead between geometric, sliced, SQarray and HAT

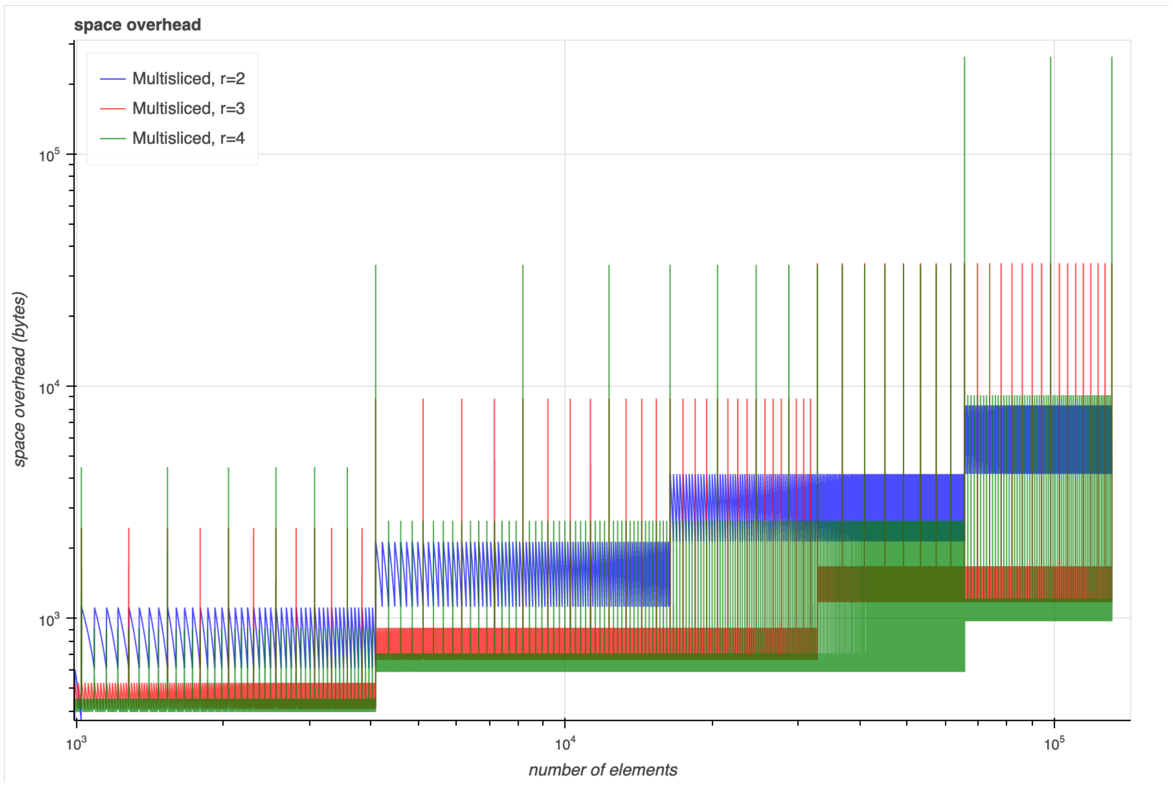


FIGURE A.2: Comparison of space overhead between multisliced array with r=2,3,4

Bibliography

[1] Alistair Moffat and Joel Mackenzie. Immediate-access indexing using space-efficient extensible arrays. In *Australasian Document Computing Symposium (ADCS '22), December 15-16, 2022, Adelaide, SA, Australia.* ACM, New York, NY, USA, 2022.

[2] Robert E Tarjan and Uri Zwick. Optimal resizable arrays. In *Symposium on Simplicity in Algorithms (SOSA)*, pages 285–304. SIAM, 2023.

[3] Andrej Brodnik, Svante Carlsson, Erik D Demaine, J Ian Ian Munro, and Robert Sedgewick. Resizable arrays in optimal time and space. In *Algorithms and Data Structures: 6th International Workshop, WADS'99 Vancouver, Canada, August 11–14, 1999 Proceedings 6*, pages 37–48. Springer, 1999.

[4] Edward Sitarski and Algorithm Alley. Hats: Hashed array trees. *Dr. Dobb's Journal*, 21(11), 1996.

[5] Jyrki Katajainen. Worst-case-efficient dynamic arrays in practice. In *Experimental Algorithms: 15th International Symposium, SEA 2016, St. Petersburg, Russia, June 5-8, 2016, Proceedings 15*, pages 167–183. Springer, 2016.