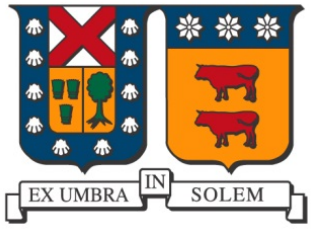
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
SANTIAGO - CHILE



“BRODNIK’S DATA STRUCTURE IN PRACTICE, REVISITED”

CARLOS CRISTÓBAL LAGOS CORTÉS

THESIS FOR THE DEGREE OF
CIVIL ENGINEER IN COMPUTER SCIENCE

Supervisor: ROBERTO JAVIER ASIN ACHA
Co-Supervisor: GABRIEL ALFREDO CARMONA TABJA
Co-Advisor: José Luis Martí

June - 2025

CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

Tipo de monografía (marcar una opción): ☒ Memoria o trabajo de título; ☐ Tesis de Postgrado;

Título del trabajo: Brodnik's Data Structure in Practice, Revisited

Nombre del candidato(a): Carlos Cristóbal Lagos Cortés

Carrera / Grado: Ingeniería Civil Informática

Campus: Santiago San Joaquin **;** **Departamento**: Informática

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Roberto Asín, en mi calidad de profesor(a) guía/director(a) del trabajo académico mencionado anteriormente **DEJO CONSTANCIA** que:

- He revisado esta versión del documento y corresponde a la versión final aprobada del trabajo.
- El trabajo cumple con los requisitos académicos y de formato establecidos por la institución

3.- EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL

☒ El trabajo **NO contiene información que amerite confidencialidad** y puede ser publicado de inmediato en repositorio con acceso abierto.

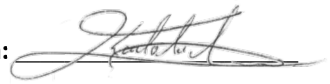
☐ El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (embargo) por:

☐ 6 meses; ☐ 12 meses; ☐ 2 años; ☐ 3 años; ☐ 5 años; ☐ 10 años

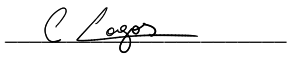
Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

4.- FIRMAS

Profesor(a) guía o director(a) de memoria o tesis:

Fecha: 08/20/2025 ; **Firma:** 

Estudiante o Candidato(a):

Fecha: 08/20/2025 ; **Firma:** 

Este formulario debe ser insertado como página 2 de la memoria o tesis, completado y firmado por estudiante y profesor(a) antes de la entrega en portal PRISMA de Biblioteca USM.

DEDICATION

I dedicate this work to all those who share the passion for algorithms, data structures, and problem-solving. This field awakens in me a deep enthusiasm, and I hope that this work can inspire that same passion in more people. I also dedicate it to my family, my partner, and my friends, who have been fundamental support during this process.

ACKNOWLEDGMENTS

I want to express my sincere gratitude to all the teachers who during my studies gave me experience, knowledge, and motivation. In particular, I thank Gabriel Carmona, who offered me invaluable support and knowledge in the field of competitive programming, awakening in me a passion for algorithms and data structures that I had never imagined. I also want to thank Professor Robert Asín, who helped me with his experience, resolving doubts and actively participating in the development of this work.

I extend my gratitude to my competitive programming teammates, Sebastián Torrealba and Abner Vidal, with who I have learned not only technical aspects, but also the value of collaborative work and joint growth. Their active participation and constant enthusiasm were key to keeping my motivation for algorithms and competitive programming alive.

Finally, I thank my family and friends for their constant companionship and encouragement in the most stressful moments. In a special way, I thank my mother for everything she has done for me over all these years, waiting for me every night and worrying day by day that everything would go well for me. Also to my partner, who always helped me clear my mind and provided me with the necessary moments of distraction throughout this process.

ABSTRACT

***Abstract*—** In 1999, Brodnik et al. proposed an alternative data structure to dynamic arrays for efficient data stream processing. Despite its theoretical promise, this structure has received limited implementation and evaluation on modern computer architectures. This work presents a novel C++-compatible implementation of Brodnik's structure, specifically designed as a container for use with the stack adapter. We evaluate the performance of this implementation across different application scenarios, including sorting and examining its effectiveness as the underlying container for priority_queue, and stack adapters. When employed as the underlying container for stack operations, our implementation demonstrates significant time performance advantages over the standard vector implementation while achieving substantial space savings. Compared to the default stack container, deque, Brodnik's structure matches the temporal performance while delivering improved space efficiency. These results establish the practical viability of Brodnik's theoretical framework in contemporary computing environments and highlight new opportunities for efficient data stream processing in performance-critical applications.

***Keywords*—** Brodnick, Data Structures, Stack, C++ Container.

TABLE OF CONTENTS

INDEX OF FIGURES

INDEX OF TABLES

INTRODUCTION

Data structures form the backbone of efficient algorithm design and implementation, with dynamic arrays being among the most fundamental and widely used constructs in modern programming. However, as computational demands have evolved and data processing requirements have become increasingly complex, the limitations of traditional dynamic arrays have become more apparent, particularly in scenarios involving continuous data streams and high-frequency operations.

In 1999, Brodnik et al. [Brodnik *et al.*, 1999] proposed an innovative alternative data structure designed specifically to address the performance bottlenecks associated with dynamic array operations in stream processing contexts. This structure introduced novel concepts such as superblock organization and logarithmic indexing schemes, which theoretically promised significant improvements in memory usage while maintaining time complexity. Due to its lower bound on memory usage, this work has been cited in several papers on succinct data structures [Raman y Rao, 2003, Raman *et al.*, 2001, Li *et al.*, 2023, Munro *et al.*, 2001, Arroyuelo *et al.*, 2016], dynamic data structures [Mehta y Sahni, 2004, Tarjan y Zwick, 2024], and others [Banerjee *et al.*, 2018].

Despite its theoretical promise, Brodnik's structure has remained largely unexplored in practical implementations. The gap between theoretical innovation and real-world application has persisted for over two decades, primarily due to implementation complexity and the lack of comprehensive evaluation on modern computer architectures. Contemporary systems feature sophisticated hardware characteristics such as multi-level cache hierarchies, branch prediction mechanisms, and vectorized instruction sets, which fundamentally alter the performance characteristics of data structure operations compared to the computational environments of Brodnik's era.

The C++ Standard Template Library (STL) [Stroustrup, 2013a] provides robust implementations of fundamental containers such as `std::vector`, `std::list`, and `std::deque`, which serve as the foundation for countless applications. These containers are often used as the underlying dynamic linear structures for different adaptors (i.e. abstract data types) like `std::stack`, `std::queue`, and `std::priority_queue`, which support specific types of operations including insertions, deletions, and size modifications.

In [Joannou y Raman, 2011], an empirical evaluation compares the Brodnik et al. data structure with `std::vector`, showing it to be impractical. They saw that even if this data structure uses less memory than `std::vector`, at the cost of being significantly slower and causing more fragmentation. Here we revisit the empirical evaluation and through more insights into this data structure.

In this work, we bridge the gap between Brodnik's theoretical contribution and practical application by presenting the first comprehensive implementation of this data structure, designed for integration with modern C++ development practices. Our implementation is specifi-

cally engineered to serve as a drop-in replacement for existing STL containers, maintaining API compatibility while providing enhanced performance for dynamic operations.

We conduct extensive empirical evaluations across multiple scenarios, with a particular focus on highly dynamic environments such as stack operations, where the frequency of push and pop operations puts significant stress on the underlying data structure. Our experimental results show that Brodnik's structure delivers substantial performance improvements and space efficiency gains compared to `std::vector` when used as the underlying container for stack operations, while achieving similar temporal performance to `std::deque` with better space utilization.

The contributions of this work are threefold: first, we provide the first modern implementation of Brodnik's data structure optimized for current hardware architectures; second, we demonstrate its practical viability through comprehensive performance evaluations across different STL adapter scenarios; and third, we establish a foundation for future research in alternative data structures for stream processing applications, opening new possibilities for efficient data processing in performance-critical environments.

CHAPTER 1
C++'s STL Containers and Adapters

The C++ Standard Template Library (STL) provides a comprehensive set of container classes that implement fundamental abstract data types, each offering distinct performance characteristics and memory layouts. Understanding these existing structures is essential for contextualizing our proposed data structure and identifying the specific gap it addresses in the current ecosystem.

1.1. Fundamental STL Containers

The `std::vector` [Dietzfelbinger *et al.*, 1988, Stroustrup, 2013b] container represents the simplest memory layout, storing elements contiguously in a dynamically allocated array. This design enables constant-time random access through pointer arithmetic and exhibits excellent cache locality, as adjacent elements reside in consecutive memory locations. However, vectors suffer from significant limitations in dynamic operations. Insertion and deletion at arbitrary positions require $O(n)$ time due to element shifting, making frequent modifications expensive.

More critically, `grow` operation presents substantial performance challenges. When a vector's capacity is exceeded, the container must allocate a new, larger array (typically doubling the previous size), copy all existing elements to the new location, and deallocate the original array. This reallocation process exhibits $O(n)$ complexity and invalidates all iterators, pointers, and references to vector elements. The amortized constant-time complexity of `push_back()` operations masks the periodic expensive reallocations, which can cause unpredictable performance spikes in time-sensitive applications.

Similarly, shrinking operations suffer from inefficiency. The `std::vector` specification does not guarantee automatic capacity reduction when elements are removed, meaning that memory usage may remain high even after significant data reduction. While the `shrink_to_fit()` function provides a mechanism to request capacity reduction, it involves potential reallocation with the same costly copying process, and the implementation may choose to ignore the request entirely. This produces that in the worst case the extra memory needed is $O(n)$.

At the opposite end of the spectrum, `std::list` [Knuth, 1997, Stroustrup, 2013b, Black, 2004] implements a node-based architecture where each element maintains pointers to its neighbors. This design facilitates $O(1)$ insertion and deletion at any position given an iterator, and growth occurs incrementally without relocating existing elements. The tradeoff manifests in poor cache performance due to scattered memory allocation and the overhead of storing pointers with each element. Random access degrades to $O(n)$ time, as traversal

requires following the chain of pointers sequentially.

The `std::deque` [Knuth, 1997, Stroustrup, 2013b] (double-ended queue) represents a hybrid approach that partially bridges these extremes. Internally, `std::deque` typically employs a two-level structure: a central array of pointers referencing fixed-size blocks that contain the actual elements. This architecture enables constant-time insertion and deletion at both ends while maintaining reasonable random access performance. Elements within each block benefit from spatial locality, though cache performance remains inferior to vectors due to the indirection layer and potential discontinuities between blocks.

Despite `std::deque`'s innovative design, it presents limitations that motivate further exploration. The fixed block size creates a rigid tradeoff between memory overhead and allocation frequency. Small blocks increase the pointer array overhead and fragmentation, while large blocks may waste memory for smaller collections. Furthermore, insertion and deletion at arbitrary positions still require $O(n)$ time, and the two-level indirection can introduce measurable overhead for simple access patterns.

The abstract data type perspective reveals that each STL container embodies specific design decisions balancing time complexity, space efficiency, and cache behavior. Vectors optimize for access and traversal but suffer from expensive growth and shrinkage operations, lists prioritize modification flexibility at the cost of access performance, and deques target double-ended operations while maintaining reasonable random access. However, modern applications increasingly demand data structures that can adapt to varying access patterns and scale efficiently across different collection sizes without the periodic performance penalties associated with bulk reallocations. This observation underscores the need for new approaches that can dynamically adjust their internal organization to match usage characteristics while maintaining predictable performance bounds.

1.2. STL Container Adapters

Beyond these fundamental containers, the STL demonstrates the power of abstraction through container adapters that implement higher-level abstract data types. The `std::queue`, `std::stack`, and `std::priority_queue` [Knuth, 1997, Stroustrup, 2013b, Vuillemin, 1978] adapters exemplify how specialized interfaces can be layered atop existing containers, provided the underlying structure supports the required operations with appropriate performance characteristics.

The `std::stack` adapter implements a Last-In-First-Out (LIFO) discipline by restricting access to a single end of the underlying container. By default, `std::stack` uses `std::deque` as its container, though `std::vector` and `std::list` are equally valid choices. The adapter requires only three operations from its underlying container: `back()` for accessing the top element, `push_back()` for insertion, and `pop_back()` for removal. This minimal interface allows any sequence container supporting these operations to serve as the implementation

basis, with the adapter ensuring that users cannot violate the LIFO semantics through direct element access or arbitrary position manipulation.

The choice of underlying container significantly impacts stack performance characteristics. When `std::vector` serves as the underlying container, stack operations benefit from excellent cache locality and minimal per-element overhead, but suffer from periodic reallocation costs that can introduce unpredictable latency spikes. The `std::deque` default provides more consistent performance by avoiding bulk reallocations, though it incurs additional memory overhead and indirection costs. The `std::list` option eliminates reallocation concerns entirely but sacrifices cache performance and increases memory usage due to pointer storage requirements.

Similarly, the `std::queue` adapter enforces First-In-First-Out (FIFO) behavior by exposing insertion at the back and removal from the front. The default `std::deque` container proves ideal for this purpose, as it provides constant-time operations at both ends. The adapter interface requires `front()`, `back()`, `push_back()`, and `pop_front()` operations. While `std::list` satisfies these requirements, `std::vector` cannot serve as a queue container due to its linear-time `pop_front()` complexity, illustrating how performance characteristics constrain container selection.

The `std::priority_queue` adapter presents a more complex case, implementing a heap-based priority queue typically backed by a `std::vector`. The adapter maintains the heap invariant through its `push()` and `pop()` operations, using the underlying container's random access iterators to efficiently navigate the implicit binary tree structure. The choice of `std::vector` reflects the heap's need for constant-time access to parent and child nodes through index arithmetic. Although `std::deque` could theoretically support these operations, `std::vector`'s superior cache locality and lower overhead make it the preferred choice for heap operations.

1.3. Design Implications

This adapter pattern reveals a crucial design principle: the separation of interface from implementation enables algorithmic flexibility without sacrificing type safety or performance. The adapters impose semantic constraints while delegating storage and basic operations to proven container implementations. This architecture suggests that our proposed data structure could similarly serve as an underlying container for these adapters, provided it exposes the necessary operations with competitive performance characteristics and addresses the reallocation issues that plague existing containers.

The relationship between containers and adapters also highlights the importance of iterator categories and operation complexity guarantees in the STL ecosystem. A container's suitability for a particular adapter depends not merely on the presence of required operations but on their asymptotic complexity, iterator stability guarantees, and the absence of unpre-

dictable performance characteristics such as bulk reallocations. This consideration becomes particularly relevant when designing new containers intended to integrate seamlessly with existing STL infrastructure while providing more predictable performance profiles for dynamic operations.

CHAPTER 2
Brodnick's Data Structure

Brodnik et al. [Brodnik *et al.*, 1999] proposed a data structure, which we refer to as `brodnik`, for single-resizable arrays that achieve optimal time and space complexity. They prove that any data structure supporting insertion and deletion of elements in arbitrary order requires $\Omega(\sqrt{n})$ extra space. This lower bound applies to resizable arrays, stacks, queues, randomized queues, priority queues, and deques.

The `brodnik` structure comprises two types of memory blocks: one *index block* and d *data blocks* (where d can grow or shrink during execution). The index block contains pointers to all the data blocks. The data blocks, denoted $DB_0, \ldots, DB_{d-1}$, store all the elements in the `brodnik` structure. A crucial difference between `brodnik` and `std::deque` is that the former's blocks do not have a fixed size.

Conceptually, data blocks are organized into s *superblocks*, denoted $SB_0, \ldots, SB_{s-1}$. Two data blocks belong to the same superblock if they have the same size. A superblock SB_k can contain up to $2^{\lfloor k/2 \rfloor}$ data blocks, each of size $2^{\lceil k/2 \rceil}$. When a superblock is full, it contains exactly 2^k elements. It is important to note that superblocks are not physically stored but serve as a conceptual framework to facilitate understanding of the structure's organization.

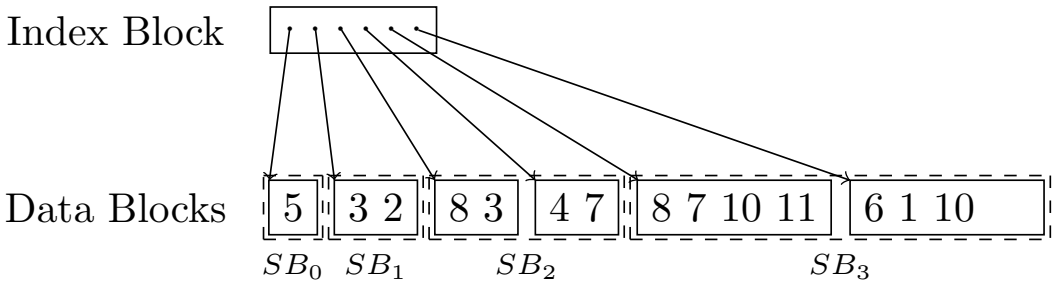


Figura 1: The corresponding index block, data blocks, and conceptual super blocks of the `bronik` structure after inserting the elements $\{5, 3, 2, 8, 3, 4, 7, 8, 7, 10, 11, 6, 1, 10\}$ in the given order. It is important to note that the last data block is not full, as it can store up to four elements.

As an example, Figure 1 illustrates the structure after inserting the elements $\{5, 3, 2, 8, 3, 4, 7, 8, 7, 10, 11, 6, 1, 10\}$ in that order. In this case, the `brodnik` structure consists of an index block with six pointers, six data blocks (with the last one not being full), and four conceptual superblocks.

Brodnik et al. demonstrate how to perform the following four fundamental operations for any resizable array:

- `grow()`: Increment n, creating a new element with index n.
- `shrink()`: Decrement n, discarding the element with index $n - 1$.
- `read(i)`: Return the element with index i, where $0 \leq i < n$.
- `write(i, x)`: Set the element with index i to x, where $0 \leq i < n$.

They replace `read(i)` and `write(i, x)` with the operation `access(i)`, which determines the location of the element with index i. It is clear that both `read` and `write` are nearly identical operations, differing only in their final instruction: return or replace, respectively.

The implementations of `grow()`, `shrink()`, and `access(i)` are described in Algorithms 1.3-1, 1.3-2, and 1.3-3, respectively. Algorithm 1.3-3 in the original paper contained an error in line 5, which was subsequently corrected by Joannou and Raman [Joannou y Raman, 2011]. In Algorithm 1.3-2, line 10, we added the condition that $s > 1$, because if $s = 1$ and we perform a `shrink()`, we will assume that there is a superblock 0 and reduce the number of elements to 0, which can lead to a segmentation fault if a `grow()` operation is performed afterwards.

Algorithm 1 Basic implementation of `grow()`

```
1:  if the last nonempty data block DB_{d-1} is full then
2:      if the last superblock SB_{s-1} is full then
3:          Increment s
4:          if s is odd then
5:              double the number of data blocks in a superblock
6:          else
7:              double the number of elements in a data block
8:          end if
9:          Set the occupancy of SB_{s-1} to empty
10:     end if
11:     if there are no empty data blocks then
12:         if the index block is full then
13:             Reallocate it to twice its current size
14:         end if
15:         Allocate a new last data block
16:         store a pointer to it in the index block
17:     end if
18:     Increment d and the number of data blocks occupying SB_{s-1}
19:     Set the occupancy of DB_{d-1} to empty
20: end if
21: Increment n and the number of elements occupying DB_{d-1}
```

Algorithm 2 Basic implementation of `shrink()`

Decrement n and the number of elements occupying the last nonempty data block DB_{d-1}.
if DB_{d-1} is empty **then**
  if there is another empty data block **then**
    Deallocate it.
  end if
  if the index block is a quarter full **then**
    Reallocate it to half its size.
  end if
  Decrement d and the number of data blocks occupying the last superblock SB_{s-1}.
  if $s > 1$ and SB_{s-1} is empty **then**
    Decrement s.
    if s is even **then**
      Halve the number of data blocks in a superblock.
    else
      Halve the number of elements in a data block.
    end if
    Set the occupancy of SB_{s-1} to full.
  end if
  Set the occupancy of DB_{d-1} to full.
end if

Algorithm 3 Basic implementation of `access(i)`

Let r denote the binary representation of $i + 1$, with all leading zeros removed.
Let $k = \lfloor \log_2(i + 1) \rfloor$, correspond to which superblock.
Let b be the $\lfloor k/2 \rfloor$ bits of r immediately after the leading 1-bit, correspond to which data block in superblock k.
Let e be the last $\lceil k/2 \rceil$ bits of r, correspond to which element in data block b.
Let $p = 2^{\lfloor k/2 \rfloor} + 2^{\lceil k/2 \rceil} - 2$.
Return the location of element e in data block DB_{p+b}.

In the original paper, Brodnik et al. proved that `brodnik` uses $O(\sqrt{n})$ extra space in the worst case and supports $O(1)$ time per operation on a random access machine. Additionally, they demonstrated that if `allocate` or `deallocate` is called when the number of elements is $n = n_0$, the next call to one of these operations will not occur until after $\Omega(\sqrt{n_0})$ further operations.

Brodnik et al. [Brodnik *et al.*, 1999] also presented a variation of `brodnik` adapted for alternative memory management systems. Some of these systems allocate memory in fixed-size blocks (e.g., of size h), causing total block sizes to be slightly larger than powers of two (e.g., $2^k + h$). In such cases, like the buddy system, which rounds the total block size to the next power of two, `brodnik` may end up using twice as much memory as needed, instead of the optimal $O(\sqrt{n})$ overhead.

Due to its optimal memory usage, `brodnik` is primarily studied in theoretical work related to succinct data structures, but it is rarely used in practice. However, one notable implementation was developed by Joannou and Raman [Joannou y Raman, 2011] in 2011. They created three variants of `brodnik`, and their experiments showed that `brodnik` was generally slower than `std::vector`, except for one variant. However, that variant required parameter tuning, making it impractical for general use. Their evaluation included both sequential and random access patterns, with the data structure growing up to a fixed target size.

In this work„ we show that for specific types of applications that use a dynamic stream of insertions and deletions, such as in stack problems, graph problems, etc. `brodnik` can be a practical option for large scale data, because the doubling technique used in `std::vector` can be costly. We focus on testing three main applications: stack simulation, random access, and graph simulation. In contrast to Joannou and Raman [Joannou y Raman, 2011], who mainly focused on dynamic arrays with random access. An important point to note is that our implementation is usable as a container in the C++ standard library, which makes it easy to use in practice.

CHAPTER 3
Experimental Evaluation

3.1. Performance Metrics

Our experimental evaluation focuses on two primary performance indicators that reflect the practical efficiency of dynamic array implementations in real-world computing environments:

- **CPU User Time**: Time spent executing user-mode instructions, measuring the computational efficiency of each data structure implementation.
- **Peak Memory Usage (RSS)**: Maximum Resident Set Size during execution, indicating memory efficiency and allocation overhead.

These metrics provide a comprehensive view of both computational and memory performance, which are critical factors in evaluating dynamic array implementations for performance-critical applications. The combination of these measurements allows us to assess the practical tradeoffs between time and space efficiency across different usage patterns.

3.2. Test Case Specifications

Our benchmark suite evaluates three distinct algorithmic scenarios, each designed to stress different aspects of dynamic linear containers' performance characteristics. These test cases represent common usage patterns found in real-world applications and provide insights into how each data structure performs under varying operational loads.

3.2.1. `std::stack` Operations

The stack operation benchmarks simulate realistic usage patterns encountered in applications such as expression evaluation, backtracking algorithms, and function call management. Our test suite encompasses five primary categories of stack behavior:

Random Operations (random): Initial insertion of n elements using $\texttt{push}$, followed by n randomized operations alternating between $\texttt{push}$ and $\texttt{pop}$. These tests evaluate performance under mixed workload scenarios where operation sequences vary unpredictably.

Sequential Push-Random Pattern (push-random): Initial insertion of n elements using `push`, followed by $n/2$ additional `push` operations, and concluding with $n/2$ random operations between `push` and `pop`. This pattern simulates applications with initial growth phases followed by mixed manipulation periods.

Sequential Random-Pop Pattern (random-pop): Initial insertion of n elements using `push`, followed by $n/2$ random operations (alternating `push` and `pop`), and concluding with $n/2$ consecutive `pop` operations. These tests measure efficiency during controlled shrinkage after mixed operation phases.

Sequential Push-Random-Pop Sequence (push-random-pop): Initial insertion of n elements using `push`, followed by $n/3$ additional `push` operations, then $n/3$ random operations between `push` and `pop`, and finally $n/3$ `pop` operations. This comprehensive pattern evaluates performance across complete stack lifecycle scenarios.

Sequential Cyclic Push-Pop Workloads (cyclic-push-pop): Initial insertion of n elements using `push`, followed by five cycles of variable-length operations, where each cycle performs a random number of `push` operations (between $n/10$ and n) followed by a corresponding number of `pop` operations limited by current stack size. These tests simulate high-stress scenarios with large insertion bursts followed by significant reductions, evaluating performance under intensive push-pop operations.

In all scenarios, the parameter $n \in \{10^7, 10^8, 10^9\}$, enabling evaluation across large-scale application regimes from ten million to one billion elements.

3.2.2. Sorting Algorithms

Sorting benchmarks consist of randomly generated arrays with sizes $n \in \{10^7, 10^8, 10^9\}$ elements, where n represents the array length, utilizing uniformly distributed integer values to ensure consistent comparison conditions. Standard sorting algorithms implemented in the STL's `std::algorithm` library were applied, evaluating the performance impact of using each dynamic container as the underlying storage mechanism for operations requiring frequent random access. Our implementation of `brodnik` provides the complete random access iterator interface required by STL algorithms, enabling seamless integration with `std::sort`. The `std::sort` function implements the introsort algorithm [Musser, 1997], a hybrid approach that adaptively selects between quicksort [Hoare, 1961] for general cases, heapsort [Williams, 2025, Floyd, 1964] for worst-case scenarios, and insertion sort [Knuth, 1998] for small subarrays. This algorithmic complexity stresses different aspects of container performance, from random access efficiency during partitioning to iterator stability during element swaps. The sorting benchmarks are particularly valuable for evaluating random access performance, as they require extensive element comparisons and movements throughout the container. These operations test both the theoretical complexity guarantees and the practical cache performance characteristics of each implementation.

3.2.3. `std::priority_queue` Operations

Priority queue benchmarks evaluate heap-based operations through two complementary testing methodologies that reflect common usage patterns in algorithmic applications.

Synthetic Workloads: We generate random sequences of `push` and `pop` operations on binary heaps, with `top` queries performed after specific modification intervals. The operation sequences maintain dynamically varying heap sizes throughout execution, thereby testing performance stability across different container states. Test instances comprise $n \in \{10^7, 10^8\}$ operations, where n represents the total number of operations. Operation ratios are carefully balanced to maintain heap sizes within realistic algorithmic bounds. These synthetic workloads isolate the performance characteristics of individual heap operations under controlled conditions.

Algorithmic Applications: To assess performance in realistic computational contexts, we implement Dijkstra's shortest path algorithm using each container as the underlying priority queue storage. Tests are conducted on randomly generated sparse and dense directed graphs. Sparse graphs comprise $n \in \{10^7, 10^8\}$ vertices, where $n = |V|$ represents the number of vertices, with $O(|V|)$ edge density (specifically $2|V|$ edges). These graphs are constructed through connected linear chains extended with additional random edges to ensure connectivity. Dense graphs comprise $n \in \{10^3, 10^4\}$ vertices with $O(|V|^2)$ edge density (up to $|V|^2/4$ edges), approaching maximum connectivity within practical memory constraints. This application-driven evaluation provides insights into how container performance translates to algorithmic efficiency in scenarios where priority queue operations are embedded within larger computational workflows requiring frequent distance comparisons and path updates.

The priority queue benchmarks present particularly demanding requirements for container implementations, as heap operations require efficient random access for parent-child navigation while maintaining competitive insertion and deletion performance. These evaluations effectively assess the balance between access efficiency and modification costs across different implementation strategies, revealing how theoretical complexity advantages manifest in practical algorithmic scenarios.

3.3. Experimental Environment

All experiments were conducted under controlled conditions to ensure reproducible results:

Hardware Specifications:

- CPU: Intel(R) Xeon(R) Gold 6330 CPU @ 2.00GHz
- Memory: 32GB

Software Environment:

- Operating System: Ubuntu 24.04 LTS
- Compiler: GCC (C++) version 14.2.0
- Compilation Flags: `-O3 -march=native`

3.4. Container Implementations Evaluated

- `brodnik`: Our implementation of Brodnik et al.'s structure
- `std::deque`: STL double-ended queue implementation
- `std::vector`: Baseline STL dynamic array implementation

All container implementations and test cases are available in the accompanying repository[1].

[1] https://github.com/CharlesLakes/dynamic-arrays/

3.5. Results and Analysis

3.5.1. Experimental Results for `std::stack`

First, Table 3.5.1 shows that for smaller values of n, `brodnik` is significantly slower than `std::vector` and `std::deque`, while for larger values of n, this gap is reduced. This can be explained by the nature of the experiment. Since this is a purely random `push`/`pop` sequence, even when $n = 10^9$, the number of elements in the data structure at any given time does not grow significantly. This means that `std::vector` rarely needs to perform costly memory copies due to resizing. Also, `std::deque` is slightly faster than `std::vector` for $n = 10^9$ and $n = 10^8$. A possible explanation is that the sequence of operations caused `std::vector` to perform several `grow`/`shrink` operations, while `std::deque` performed fewer. In terms of space, we can see that even with a purely random sequence of operations, `std::brodnik` consistently uses less peak memory across all values of n, with the largest difference being $6\,\%$ less than `std::vector` and $4\,\%$ less than `std::deque`.

N	Data Structure	CPU Time (s)	Memory (GB)
10^7	`brodnik`	7.890	0.044
	`std::deque`	7.100	0.045
	`std::vector`	6.810	0.071
10^8	`brodnik`	65.660	0.404
	`std::deque`	64.630	0.422
	`std::vector`	64.750	0.540
10^9	`brodnik`	641.650	4.005
	`std::deque`	640.590	4.191
	`std::vector`	640.990	4.299

Tabla 1: Performance Metrics for `std::stack` (random); table reports total execution time in seconds and peak memory usage in Gigabytes during the experiment of each n.

In contrast to the fully random sequence of operations, Table 3.5.1 shows that `brodnik` is faster than `std::deque` for $n = 10^7$ and $n = 10^9$, and faster than `std::vector` for $n = 10^8$ and $n = 10^9$. A possible explanation for this is that the initial mandatory sequence of `push` operations in this experiment forced `std::vector` to perform costly `grow` operations in the larger cases. After this initial phase, the random sequence that followed did not require many `grow`/`shrink` operations for any of the structures. In terms of space, again, `brodnik` consistently uses less peak memory across all values of n, with the largest difference being $30\,\%$ less than `std::vector` and $1\,\%$ less than `std::deque`. This can be explained by the fact that `deque` is a more space-efficient data structure than `std::vector`, thanks to its fixed block size.

Table 3.5.1 shows that in cases with a large number of `shrink` operations, all data structures exhibit similar performance without significant differences. However, in terms of space,

N	Data Structure	CPU Time (s)	Memory (GB)
10^7	`brodnik`	6.900	0.064
	`std::deque`	7.280	0.066
	`std::vector`	6.880	0.071
10^8	`brodnik`	68.780	0.604
	`std::deque`	68.560	0.632
	`std::vector`	69.240	1.077
10^9	`brodnik`	687.490	6.005
	`std::deque`	690.500	6.285
	`std::vector`	689.970	8.594

Tabla 2: Performance Metrics for `std::stack` (push-random); table reports total execution time in seconds and peak memory usage in Gigabytes during the experiment of each n.

`brodnik` uses less peak memory, with the largest difference observed at $n = 10^9$, where it uses $4\,\%$ less than `deque` and $6\,\%$ less than `vector`. This gap is smaller compared to other test cases because this experiment is designed to highlight the behavior of the `shrink` operation, resulting in a lower number of elements in the data structure compared to other experiments.

N	Data Structure	CPU Time (s)	Memory (GB)
10^7	`brodnik`	5.920	0.044
	`std::deque`	6.430	0.045
	`std::vector`	5.940	0.071
10^8	`brodnik`	60.690	0.404
	`std::deque`	59.880	0.422
	`std::vector`	59.200	0.540
10^9	`brodnik`	587.120	4.005
	`std::deque`	587.350	4.191
	`std::vector`	587.460	4.298

Tabla 3: Performance Metrics for `std::stack` (random-pop); table reports total execution time in seconds and peak memory usage in Gigabytes during the experiment of each n.

Finally, Tables 3.5.1 and 3.5.1 show similar behavior across the data structures, where `brodnik` is slower than `deque` and `vector`. A possible explanation is that forcing `brodnik` to grow and shrink multiple times and by large amounts can produce extra operations that end up being costly. However, in terms of space, `brodnik` always uses less peak memory compared to `deque` and `vector`, with their maximum differences being $4\,\%$ and $37\,\%$, respectively.

N	Data Structure	CPU Time (s)	Memory (GB)
10^7	`brodnik`	7.070	0.057
	`std::deque`	7.070	0.059
	`std::vector`	6.430	0.071
10^8	`brodnik`	64.070	0.537
	`std::deque`	64.040	0.562
	`std::vector`	64.580	0.540
10^9	`brodnik`	636.770	5.338
	`std::deque`	634.990	5.587
	`std::vector`	634.990	8.593

Tabla 4: Performance Metrics for `std::stack` (push-random-pop); table reports total execution time in seconds and peak memory usage in Gigabytes during the experiment of each n.

N	Data Structure	CPU Time (s)	Memory (GB)
10^7	`brodnik`	20.060	0.053
	`std::deque`	19.750	0.055
	`std::vector`	20.950	0.071
10^8	`brodnik`	160.200	0.459
	`std::deque`	160.420	0.479
	`std::vector`	160.680	0.540
10^9	`brodnik`	1003.590	6.459
	`std::deque`	1001.780	6.759
	`std::vector`	994.750	8.594

Tabla 5: Performance Metrics for `std::stack` (cyclic-push-pop); table reports total execution time in seconds and peak memory usage in Gigabytes during the experiment of each n.

3.5.2. Experimental Results for Sort

In Table 3.5.2, we can see that `brodnik` is slower than both `deque` and `vector`, with the latter always being the fastest. With $n = 10^9$, `brodnik` is 20 % slower than `vector`. This can be explained by the fact that access in `vector` is optimal, requiring only two operations: an addition (to compute the position) and a dereference (to retrieve the element), while `deque` and `brodnik` require additional operations to access their elements. In terms of space, `brodnik` and `vector` use very similar peak memory during execution, while `deque` uses slightly more. In this case, `brodnik` cannot take advantage of its optimal space usage to reduce peak memory, because `vector` can allocate n words of memory at the beginning, without needing to apply the doubling technique during growth.

N	Data Structure	CPU Time (s)	Memory (GB)
10^7	`brodnik`	5.020	0.044
	`std::deque`	4.470	0.045
	`std::vector`	4.330	0.044
10^8	`brodnik`	49.050	0.404
	`std::deque`	43.500	0.422
	`std::vector`	41.620	0.404
10^9	`brodnik`	510.250	4.005
	`std::deque`	442.070	4.191
	`std::vector`	423.330	4.004

Tabla 6: Performance Metrics for `std::sort`; table reports total execution time in seconds and peak memory usage in Gigabytes during the experiment of each n.

3.5.3. Experimental Results for Priority Queue

First, as a `priority_queue`, Table 3.5.3 reports that `brodnik` is faster than `deque`, but slower than `vector`, a possible explanation is that this TDA at the moment of applying a `push`/`pop` can possibly do more operations to set/erase an element, so even if the allocations are costly, the operations to move the elements during these operations, in vector is faster. But, in terms of space, `brodnik` consistently has lower peak memory usage. For instance, in the case of $n = 10^8$ it uses $25\,\%$ less than `vector` and $4\,\%$ less than `deque`.

N	Data Structure	CPU Time (s)	Memory (GB)
10^7	`brodnik`	7.340	0.023
	`std::deque`	7.360	0.024
	`std::vector`	6.830	0.037
10^8	`brodnik`	75.830	0.204
	`std::deque`	78.380	0.213
	`std::vector`	73.430	0.272

Tabla 7: Performance Metrics for `std::priority_queue` (random); table reports total execution time in seconds and peak memory usage in Gigabytes during the experiment of each n.

Regarding the usage of `brodnik` as the underlying container of `priority_queue` adapter in applications like the Dijkstra's algorithm, we notice that it significantly underperforms when compared to both `vector` and `deque`. A possible explanation for this is that during Dijkstra's execution, the `priority_queue` undergoes frequent movement of elements as smaller or larger values are inserted. This causes `brodnik` to perform many additional operations, including movement across different blocks, which can be considerably costly. However, in terms of space, `brodnik` continues to use less peak memory in both sparse and dense cases.

N	Data Structure	CPU Time (s)	Memory (GB)
10^7	`brodnik`	31.340	0.994
	`std::deque`	31.260	0.995
	`std::vector`	28.640	1.019
10^8	`brodnik`	395.780	9.904
	`std::deque`	393.650	9.923
	`std::vector`	363.170	10.023

Tabla 8: Performance Metrics for `std::priority_queue` (Dijkstra-sparse); table reports total execution time in seconds and peak memory usage in Gigabytes during the experiment of each n.

N	Data Structure	CPU Time (s)	Memory (GB)
10^3	`brodnik`	0.190	0.009
	`std::deque`	0.190	0.009
	`std::vector`	0.120	0.010
10^4	`brodnik`	16.090	0.599
	`std::deque`	15.890	0.599
	`std::vector`	15.940	0.600

Tabla 9: Performance Metrics for `std::priority_queue` (Dijkstra-dense); table reports total execution time in seconds and peak memory usage in Gigabytes during the experiment of each n.

CHAPTER 4
Conclusions and Future Work

This work presents the first modern, standards-compliant implementation of Brodnik's data structure, designed for seamless integration with the C++ Standard Template Library ecosystem. Our implementation provides a complete container interface that supports STL adapters and algorithms, demonstrating the practical viability of translating theoretical contributions into production-ready software components.

The comprehensive experimental evaluation reveals that Brodnik's structure exhibits a fundamental trade-off between computational efficiency and memory utilization. While generally slower than `std::vector` and `std::deque` in computationally intensive scenarios, `brodnik` consistently achieves superior memory efficiency across all evaluated workloads, with memory savings of up to 37 % compared to `std::vector`. Particularly noteworthy is its performance in specific stack operation patterns, where it can match or exceed standard containers while maintaining memory advantages, especially when expensive reallocations in traditional dynamic arrays become limiting factors.

These results establish `brodnik` as a viable alternative for memory-constrained environments, including embedded systems and large-scale data processing applications where memory availability represents a critical limitation. The trade-off of modest computational overhead for substantial memory savings represents a compelling value proposition that extends the practical applicability of dynamic data structures.

Several promising research directions emerge from this work. First, extending the implementation to support efficient `insert` and `erase` operations at arbitrary positions would create a fully dynamic data structure. Second, exploring integration with succinct data structures such as compressed bit vectors and succinct trees could leverage the optimal space usage for space-critical applications. Third, detailed memory fragmentation analysis compared to existing containers could reveal additional optimization opportunities. Finally, investigating adaptive block management strategies, concurrent access patterns, and hardware-aware optimizations could potentially close the performance gap with traditional containers while maintaining space efficiency advantages.

This work demonstrates that theoretical advances in data structures can be successfully translated into practical implementations when carefully adapted to modern computing environments, opening new opportunities for memory-efficient data processing in performance-critical applications.

REFERENCES

[Arroyuelo *et al.*, 2016] Arroyuelo, D., Davoodi, P., y Satti, S. R. (2016). Succinct dynamic cardinal trees. *Algorithmica*, 74(2):742–777.

[Banerjee *et al.*, 2018] Banerjee, N., Chakraborty, S., Raman, V., y Satti, S. R. (2018). Space efficient linear time algorithms for bfs, dfs and applications. *Theory of Computing Systems*, 62:1736–1762.

[Black, 2004] Black, P. E. (2004). Linked list. En Pieterse, V. y Black, P. E., editores, *Dictionary of Algorithms and Data Structures*. National Institute of Standards and Technology. Retrieved 2004-12-14.

[Brodnik *et al.*, 1999] Brodnik, A., Carlsson, S., Demaine, E. D., Ian Ian Munro, J., y Sedgewick, R. (1999). Resizable arrays in optimal time and space. En *Algorithms and Data Structures: 6th International Workshop, WADS'99 Vancouver, Canada, August 11–14, 1999 Proceedings 6*, pp. 37–48. Springer.

[Dietzfelbinger *et al.*, 1988] Dietzfelbinger, M., Karlin, A., Mehlhorn, K., auf der Heide, F., Rohnert, H., y Tarjan, R. (1988). Dynamic perfect hashing: upper and lower bounds. En *[Proceedings 1988] 29th Annual Symposium on Foundations of Computer Science*, pp. 524–531.

[Floyd, 1964] Floyd, R. W. (1964). Algorithm 245: Treesort. *Commun. ACM*, 7(12):701.

[Hoare, 1961] Hoare, C. A. R. (1961). Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321.

[Joannou y Raman, 2011] Joannou, S. y Raman, R. (2011). An empirical evaluation of extendible arrays. En *International Symposium on Experimental Algorithms*, pp. 447–458. Springer.

[Knuth, 1997] Knuth, D. E. (1997). *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, 3rd edición. Section 2.2.1: Stacks, Queues, and Deques, pp. 238–243.

[Knuth, 1998] Knuth, D. E. (1998). *5.2.1: Sorting by Insertion*, pp. 80–105. Addison-Wesley, 2nd edición.

[Li *et al.*, 2023] Li, T., Liang, J., Yu, H., y Zhou, R. (2023). Dynamic "succincter". En *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 1715–1733. IEEE.

[Mehta y Sahni, 2004] Mehta, D. P. y Sahni, S. (2004). *Handbook of data structures and applications*. Chapman and Hall/CRC.

[Munro *et al.*, 2001] Munro, J. I., Raman, V., y Storm, A. J. (2001). Representing dynamic binary trees succinctly. En *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pp. 529–536. Citeseer.

[Musser, 1997] Musser, D. R. (1997). Introspective sorting and selection algorithms. *Software: Practice and Experience*, 27(8):983–993.

[Raman *et al.*, 2001] Raman, R., Raman, V., y Rao, S. S. (2001). Succinct dynamic data structures. En *Algorithms and Data Structures: 7th International Workshop, WADS 2001 Providence, RI, USA, August 8–10, 2001 Proceedings 7*, pp. 426–437. Springer.

[Raman y Rao, 2003] Raman, R. y Rao, S. S. (2003). Succinct dynamic dictionaries and trees. En *International Colloquium on Automata, Languages, and Programming*, pp. 357–368. Springer.

[Stroustrup, 2013a] Stroustrup, B. (2013a). *The C++ Programming Language, Fourth Edition*. Addison Wesley.

[Stroustrup, 2013b] Stroustrup, B. (2013b). *The C++ Programming Language, Fourth Edition*, capítulo 31, pp. 885–925. Addison Wesley, Boston.

[Tarjan y Zwick, 2024] Tarjan, R. E. y Zwick, U. (2024). Optimal resizable arrays. *SIAM Journal on Computing*, 53(5):1354–1380.

[Vuillemin, 1978] Vuillemin, J. (1978). A data structure for manipulating priority queues. *Commun. ACM*, 21(4):309–315.

[Williams, 2025] Williams, J. (2025). Algorithm 232: Heapsort. *Commun. ACM*, 7(6):347–348.