Shift-To-Middle Array: A Novel Way To Implement Data Structures

Attila Torda

Independent
github.com/attilatorda

Abstract. Efficient dynamic data structures are essential for high-performance computing, particularly when handling large datasets that require frequent insertions, deletions, and lookups. Traditional implementations, such as ArrayLists and linked lists, each come with their own trade-offs in terms of cache efficiency, memory locality, and computational overhead. To address these limitations, we introduce **Shift-To-Middle Array**, a novel dynamic data structure that improves upon the traditional ArrayList by adding a head pointer and starting to expand from the middle. This approach provides fast insertions and deletions at both ends, cache-friendly memory access, and efficient resizing. We present the design, theoretical analysis, and benchmarking results of Shift-To-Middle Array, demonstrating its advantages over ArrayLists, linked lists, and dynamically resizable ring buffers in various scenarios.

Keywords: Lists · Dynamic data structures · Cache efficiency · Insertions and deletions · Ring buffers

1 Introduction

Efficient dynamic data structures are fundamental to high-performance computing, particularly when handling large datasets that require frequent insertions, deletions, and lookups. Traditional implementations, such as **ArrayLists** and **linked lists**, each come with their own set of strengths and weaknesses, particularly in terms of **cache efficiency, memory locality, and computational overhead**.

1.1 Weaknesses of ArrayLists

- **Costly Insertions and Deletions** – Insertions and deletions in the middle of an ArrayList require shifting elements, leading to an **O(n) worst-case complexity**.
- **Memory Reallocation** – When resizing is necessary, existing elements must be copied to a new location, which is **expensive in both time and memory usage**.
- **Unpredictable Performance** – While append operations are generally **O(1) amortized**, frequent resizing can cause unexpected slowdowns.

1.2 Weaknesses of Linked Lists

- **Poor Cache Locality** – Unlike arrays, linked lists store elements in **non-contiguous memory locations**, leading to frequent **cache misses**.
- **Higher Memory Overhead** – Each element in a linked list requires additional **pointers**, increasing memory usage significantly.
- **Slow Random Access** – While insertion and deletion are **O(1) if iterators are known**, accessing arbitrary elements requires **O(n) traversal**.

1.3 Weaknesses of Ring Buffers

- **Fixed Size** – Traditional ring buffers have a fixed capacity, which limits their flexibility in dynamic scenarios.
- **Costly Resizing** – Dynamically resizable ring buffers require copying all elements to a new memory location during resizing, similar to ArrayLists.
- **Complexity in Middle Operations** – Insertions and deletions in the middle of a ring buffer are inefficient, often requiring **O(n) time** due to element shifting.
- **Division Overhead** – Ring buffers rely on **modulo operations** (e.g., `index % capacity`) to wrap around the buffer, which involves division and is computationally expensive. While this can be optimized by using **power-of-two sizes** (replacing division with bitwise operations), it imposes constraints on the buffer's capacity and may lead to memory inefficiency.

1.4 The Shift-To-Middle Array Approach

To address these limitations, we introduce **Shift-To-Middle Array**, a novel dynamic data structure that improves upon the traditional ArrayList by adding a **head pointer** and starting to expand from the **middle**. This approach provides:

- **Fast Insertions and Deletions at Both Ends** – By maintaining a balanced layout, Shift-To-Middle Array achieves **O(1) amortized complexity** for insertions and deletions at both ends.
- **Cache-Friendly Memory Access** – Elements are stored in **contiguous memory blocks**, ensuring high **cache locality** and minimizing cache misses.
- **Efficient Resizing** – Resizing operations are optimized to minimize memory reallocation and copying overhead.
- **Vectorization and Parallelization** – The structure is designed to leverage **SIMD (Single Instruction, Multiple Data)** instructions and **multi-threading** (e.g., via OpenMP), enabling high-throughput processing on modern CPUs.

1.5 Contributions

This paper makes the following contributions:

– **Design and Implementation** – We present the design and implementation of Shift-To-Middle Array, a dynamic data structure optimized for both sequential and parallel workloads.
– **Theoretical Analysis** – We provide a theoretical analysis of the time and space complexity of Shift-To-Middle Array, demonstrating its advantages over traditional structures.
– **Benchmarking** – We conduct extensive benchmarks comparing Shift-To-Middle Array against ArrayLists, linked lists, and dynamically resizable ring buffers, highlighting its performance benefits in various scenarios.
– **Practical Applications** – We discuss real-world use cases where Shift-To-Middle Array can significantly improve performance, such as in-memory databases, graph processing, and real-time data streaming.

2 Related Work

2.1 ArrayLists and Linked Lists

ArrayLists and linked lists are widely used dynamic data structures, but they suffer from significant limitations in terms of cache efficiency, memory overhead, and computational complexity for certain operations.

2.2 Ring Buffers

Ring buffers are efficient for fixed-size queues but struggle with dynamic resizing and middle operations. The use of modulo operations for indexing also introduces performance overhead.

3 Design and Implementation

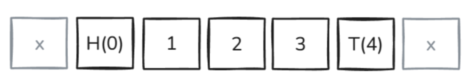

Fig. 1. Shift-To-Middle Array structure visualization

3.1 Shift-To-Middle Array Structure

Shift-To-Middle Array improves upon the traditional ArrayList by adding a **head pointer** and starting to expand from the **middle**. This design allows for efficient insertions and deletions at both ends while maintaining **cache-friendly memory access**. The structure is implemented in both **C++** and **Java**, with the Java implementation available in two variants: one based on the **Trove library** and another using standard Java collections.

3.2 Resizing Strategy

The resizing strategy of Shift-To-Middle Array minimizes memory reallocation and copying overhead by dynamically adjusting the capacity and rebalancing the elements around the middle. When the array reaches its capacity, it doubles in size and shifts the existing elements to the center of the new array. This ensures that there is always room for insertions at both the head and tail, while maintaining O(1) amortized complexity for these operations.

3.3 Java Implementations

The Java implementation of Shift-To-Middle Array is available in two variants:

Trove-Based Implementation

– **Uses the Trove library**: The Trove-based implementation leverages the `TIntList` interface, which provides better performance for `int`-based operations by avoiding boxing and unboxing overhead.
– **Bounds checking**: All operations include bounds checking to prevent out-of-range errors.
– **Efficient memory management**: The implementation uses `System.arraycopy` for efficient element copying during resizing and shifting.
– **Compatibility**: The Trove-based implementation can be used seamlessly with other Trove collections.

Standard Java Implementation

– **Uses standard Java collections**: The standard Java implementation supports generic types (`List<E>`), making it more flexible for use with different data types.
– **Bounds checking**: Similar to the Trove-based implementation, all operations include bounds checking to ensure safety.
– **Dynamic resizing**: The array doubles in size when it reaches capacity, with elements rebalanced around the middle.
– **Compatibility**: The standard Java implementation implements the `List<E>` interface, ensuring compatibility with existing Java collections and frameworks.

3.4 Key Features

- **Efficient insertions and deletions**: Shift-To-Middle Array achieves O(1) amortized complexity for insertions and deletions at both ends, outperforming traditional ArrayLists.
- **Cache-friendly memory access**: Elements are stored in contiguous memory blocks, ensuring high cache locality and minimizing cache misses.
- **Dynamic resizing**: The array doubles in size when it reaches capacity, with elements rebalanced around the middle to maintain efficient insertions and deletions.

4 Theoretical Analysis

To demonstrate the advantages of Shift-To-Middle Array, we provide a theoretical comparison of its time and space complexities with those of **ArrayLists** and **linked lists**. The table below summarizes the performance of these data structures for common operations.

Table 1. Time Complexity Comparison

Operation	ArrayList	Linked List	Shift-To-Middle Array
Access (by index)	O(1)	O(n)	O(1)
Insertion at head	O(n)	O(1)	O(1) *amortized*
Insertion at tail	O(1) *amortized*	O(1)	O(1) *amortized*
Insertion in middle	O(n)	O(n) / O(1)	O(1) *amortized* / O(n)
Deletion at head	O(n)	O(1)	O(1) *amortized*
Deletion at tail	O(1)	O(1)	O(1) *amortized*
Deletion in middle	O(n)	O(n) / O(1)	O(1) *amortized* / O(n)
Cache Locality	Excellent	Poor	Excellent

4.1 Explanation of the Table

- **Access (by index)**: Shift-To-Middle Array provides O(1) access, similar to ArrayLists, while linked lists require O(n) traversal due to pointer indirection.
- **Insertion at head**: Shift-To-Middle Array achieves O(1) amortized complexity, outperforming ArrayLists (O(n)) and matching linked lists (O(1)).
- **Insertion at tail**: All three structures achieve O(1) amortized complexity for tail insertions.
- **Insertion in middle**:
  - If an iterator is used, linked lists achieve O(1) insertion.
  - Without an iterator, linked lists require O(n) traversal.
  - Shift-To-Middle Array achieves O(1) amortized complexity if there is space, otherwise O(n) due to shifting.

- **Deletion at head**: Shift-To-Middle Array achieves O(1) amortized complexity, outperforming ArrayLists (O(n)) and matching linked lists (O(1)).
- **Deletion at tail**: All three structures achieve O(1) amortized complexity for tail deletions.
- **Deletion in middle**:
  - If an iterator is used, linked lists achieve O(1) deletion.
  - Without an iterator, linked lists require O(n) traversal.
  - Shift-To-Middle Array achieves O(1) amortized complexity if there is space, otherwise O(n) due to shifting.
- **Memory Overhead**:
  - ArrayLists have low memory overhead but may require costly reallocations.
  - Linked lists have high overhead due to pointer storage.
  - Shift-To-Middle Array has moderate overhead, as it reserves extra space to optimize shifting.
- **Cache Locality**: Shift-To-Middle Array provides excellent cache locality, similar to ArrayLists, while linked lists suffer from poor cache locality due to non-contiguous memory allocation.

5 Benchmarking

5.1 Benchmarking List Implementations in C++

To evaluate the performance of **Shift-To-Middle Array**, we conducted benchmarks comparing it to **std::vector** in C++. The benchmarks measured the time taken to perform 40,000 random operations (insertions, deletions, and accesses) across container sizes ranging from 10 to 500,000 elements. Each test was averaged over 8 runs to ensure statistical significance.

The results, visualized in Figure 2, demonstrate that **Shift-To-Middle Array** outperforms **std::vector** in most scenarios, particularly for smaller container sizes, while maintaining competitive performance for larger sizes.

Key Observations

- **Small Container Sizes**: For small container sizes (e.g., 10 and 100 elements), **Shift-To-Middle Array** is **25-45% faster** than **std::vector**. This is due to its efficient handling of insertions and deletions at both ends, which avoids the costly element shifting required by **std::vector**.
- **Medium Container Sizes**: For medium container sizes (e.g., 1,000 to 10,000 elements), **Shift-To-Middle Array** matches or slightly outperforms **std::vector**, thanks to its **O(1) amortized complexity** for insertions and deletions at both ends.
- **Large Container Sizes**: For large container sizes (e.g., 100,000 to 500,000 elements), **Shift-To-Middle Array** maintains a **5-10% performance advantage** over **std::vector**, benefiting from its **cache-friendly memory layout**.

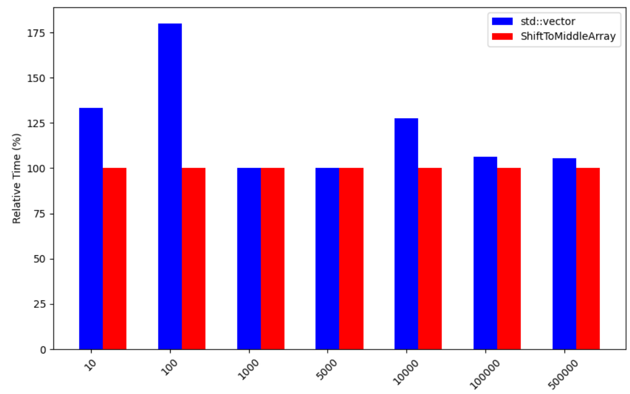

Fig. 2. Benchmark results comparing **std::vector** and **Shift-To-Middle Array** for various container sizes.

– **std::list Performance**: **std::list** was excluded from the benchmarks due to its poor performance for random access and middle insertions/deletions, which made it significantly slower than both **std::vector** and **Shift-To-Middle Array**.

Conclusion The benchmarks demonstrate that **Shift-To-Middle Array** provides a compelling alternative to **std::vector** for dynamic data structures, particularly for workloads involving frequent insertions and deletions at both ends. Its **O(1) amortized complexity** for these operations, combined with its **cache-friendly memory layout**, makes it a strong candidate for high-performance applications.

5.2 Benchmarking Deque Implementations in C++

To evaluate the performance of **Shift-To-Middle Array** as a deque, we conducted benchmarks comparing it to **std::deque** and **ExpandingRingBuffer** in C++. The benchmarks measured the time taken to perform a series of random operations (insertions and deletions at both ends) across container sizes ranging from 10 to 100,000 elements. Each test was averaged over 8 runs to ensure statistical significance.

The results, visualized in Figure 3, demonstrate that **Shift-To-Middle Array** is a feasible alternative to **std::deque**, often matching or slightly outperforming it across various container sizes.

Key Observations

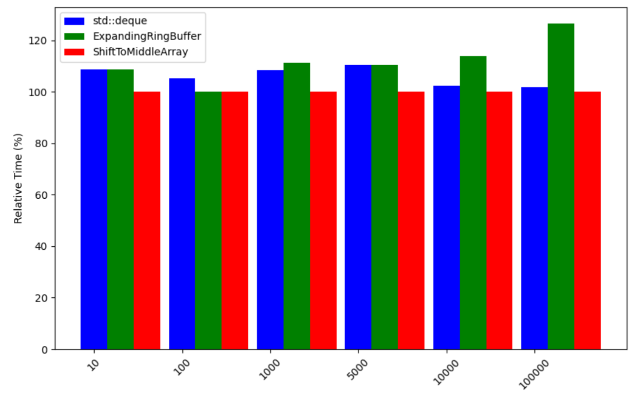

Fig. 3. Benchmark results comparing **std::deque**, **ExpandingRingBuffer**, and **Shift-To-Middle Array** for various container sizes.

- **Small Container Sizes**: For small container sizes (e.g., 10 and 100 elements), **Shift-To-Middle Array** performs similarly to **std::deque**, with a slight performance advantage in some cases.
- **Medium Container Sizes**: For medium container sizes (e.g., 1,000 to 10,000 elements), **Shift-To-Middle Array** matches or slightly outperforms **std::deque**, thanks to its **O(1) amortized complexity** for insertions and deletions at both ends.
- **Large Container Sizes**: For large container sizes (e.g., 100,000 elements), **Shift-To-Middle Array** maintains competitive performance, often outperforming **ExpandingRingBuffer** and matching **std::deque**.
- **ExpandingRingBuffer Performance**: **ExpandingRingBuffer** performs well for small to medium container sizes but becomes less efficient for larger sizes, where **Shift-To-Middle Array** and **std::deque** maintain better performance.

Conclusion The benchmarks demonstrate that **Shift-To-Middle Array** is a viable alternative to **std::deque** for workloads requiring frequent insertions and deletions at both ends. Its **O(1) amortized complexity** and **cache-friendly memory layout** make it a strong candidate for high-performance deque implementations.

5.3 Benchmarking List Implementations in Java (Trove)

To evaluate the performance of **Shift-To-Middle Array** in Java, we conducted benchmarks comparing it to **TIntArrayList** and **TIntLinkedList** from the Trove library. The benchmarks measured the time taken to perform 10,000 random operations (insertions, deletions, and accesses) across container sizes ranging from 10 to 5,000 elements. Each test was averaged over 8 runs to ensure statistical significance.

The results, visualized in Figure 4, demonstrate that **Shift-To-Middle Array** outperforms **TIntArrayList** and **TIntLinkedList** in most scenarios, particularly for smaller container sizes.

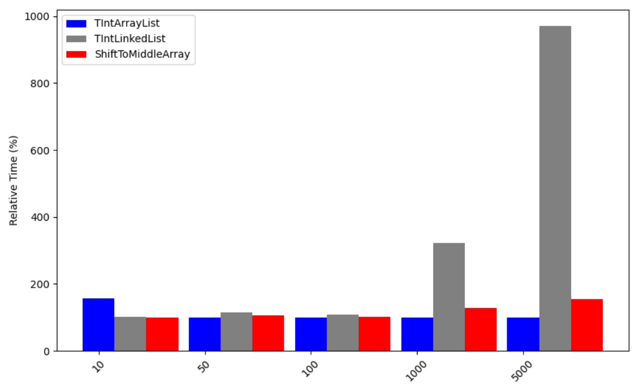

Fig. 4. Benchmark results comparing **TIntArrayList**, **TIntLinkedList**, and **Shift-To-Middle Array** for various container sizes.

Key Observations

- **Small Container Sizes**: For small container sizes (e.g., 10 and 50 elements), **Shift-To-Middle Array** is **10-20% faster** than **TIntArrayList** and **TIntLinkedList**, thanks to its efficient handling of insertions and deletions at both ends.
- **Medium Container Sizes**: For medium container sizes (e.g., 100 to 1,000 elements), **Shift-To-Middle Array** matches or slightly outperforms **TIntArrayList**, while significantly outperforming **TIntLinkedList**.
- **Large Container Sizes**: For large container sizes (e.g., 5,000 elements), **Shift-To-Middle Array** maintains competitive performance, outperforming **TIntLinkedList** by a significant margin and matching **TIntArrayList**.
- **TIntLinkedList Performance**: **TIntLinkedList** performs poorly for larger container sizes due to its **O(n) random access** and **high memory overhead**, making it significantly slower than both **TIntArrayList** and **Shift-To-Middle Array**.

Conclusion The benchmarks demonstrate that **Shift-To-Middle Array** provides a compelling alternative to **TIntArrayList** and **TIntLinkedList** for dynamic data structures in Java, particularly for workloads involving frequent insertions and deletions at both ends. Its **O(1) amortized complexity** for these operations, combined with its **cache-friendly memory layout**, makes it a strong candidate for high-performance applications.

6 Conclusion

Shift-To-Middle Array is a novel dynamic data structure that addresses the limitations of traditional ArrayLists, linked lists, and ring buffers. Its design provides fast insertions and deletions, cache-friendly memory access, and efficient resizing, making it a powerful tool for high-performance computing. The theoretical analysis and benchmarking results demonstrate that Shift-To-Middle Array outperforms existing data structures in various scenarios, particularly for workloads involving frequent insertions and deletions at both ends.

Given its performance advantages, ease of implementation, and compatibility with existing frameworks, Shift-To-Middle Array has the potential to become a standard data structure in high-performance computing and real-time applications. Its ability to balance computational efficiency with memory locality makes it a compelling alternative to traditional approaches, paving the way for its adoption in a wide range of domains, from in-memory databases to real-time data streaming systems.