Bi-Modal Skip List: A Workload-Adaptive Hybrid Text Data Structure

Integrating Gap Buffers and Skip Lists to Resolve the Edit-Scan Trade-off

Seungmin Lee

Ulsan National Institute of Science and Technology (UNIST), 20241272, downtown1629@unist.ac.kr

1 INTRODUCTION

Modern IDEs require both low-latency editing and high-throughput analysis, but traditional data structures optimize for only one workload. Gap Buffer excels at localized edits but suffers O(N) cost when the cursor moves randomly [18]. Skip List provides O(log N) access but poor sequential read performance due to pointer chasing [11].

We present the Bi-Modal Skip List, which adaptively transitions between write-optimized and read-optimized states. The structure uses **Skip List** for macro-level indexing and **Gap Buffer** for micro-level node storage. During editing, each 4KB node maintains a gap for O(1) local insertions. An explicit optimize() call compacts all nodes into contiguous arrays [8], eliminating memory fragmentation.

Our evaluation shows that Bi-Modal Skip List overcomes Gap Buffer's fundamental limitation: it achieves 46× speedup over Gap Buffer in random editing scenarios (12.6ms vs 584.4ms for 10,000 insertions), while matching `std::vector`'s sequential read performance (1.308ms vs 1.289ms). This demonstrates that workload-adaptive reconfiguration eliminates the traditional edit-vs-scan trade-off in text data structures.

1.1 Problem Statement

Early text editors played a passive role, simply storing user input in linear buffers. However, modern IDEs equipped with features like IntelliSense [5] and real-time collaboration tools must perform complex tasks—such as parsing millions of lines of code and updating dependency graphs—concurrently with user keystrokes [4]. This creates a **Dual-Workload Dilemma** for data structure design. On one hand, **latency-sensitive operations** require immediate responsiveness to user input. On the other hand, **throughput-oriented operations**, such as compiler front-ends and linters, require linear scanning of the entire document. For these tasks, physical memory contiguity is essential to maximize CPU cache prefetching and prevent pipeline stalls [10].

Existing data structures fail to satisfy both requirements simultaneously due to inherent structural limitations. `std::vector` (contiguous array [8]) offers optimal read bandwidth but incurs prohibitive O(N) data shifting costs for insertions, making it unsuitable for large-file editing[10]. While Gap Buffers provide efficient editing near the cursor, they suffer significant O(N) penalties when the cursor moves to a distant location or during multi-cursor operations. [18] Conversely, tree-based structures like Ropes and Piece Tables reduce edit complexity to O(log N), but they introduce severe memory fragmentation and pointer chasing overheads. [1, 15] This structural fragmentation significantly degrades analysis throughput due to frequent cache misses, creating a trade-off between editing responsiveness and analysis speed.

1.2 Our Contribution: Bi-Modal Skip List

We propose the **Bi-Modal Skip List**, which breaks the static morphology of traditional structures by combining the strengths of Gap Buffers and Skip Lists. Our contributions are threefold. First, we introduce a Macro-Micro Architecture that integrates a probabilistic Skip List for macro-level O(log N) indexing [11] with micro-level Gap Buffers for O(1) local edits [18]. Second, we implement a Workload-Adaptive Transition mechanism where the structure dynamically shifts between a write-optimized "Gap Node" and a read-optimized "Compact Node."

2 RELATED WORK

2.1 Array-based Structures

The baseline for contiguous memory storage is represented by `std::vector` and `std::string`. While they offer the theoretical limit for sequential read speeds due to excellent cache locality, their O(N) insertion cost renders them impractical for interactive text editing [10]. To address this, **Gap Buffers** introduce a flexible gap at the cursor position. While this design is ideal for single-cursor typing, its performance degrades linearly (O(N)) when the gap must be moved to support random edits or multiple cursors [18].

2.2 Tree and List-based Structures

To handle large texts efficiently, **Ropes** split text into smaller chunks stored as leaf nodes of a tree [1]. We incorporated GNU Rope (`__gnu_cxx::crope`), an external implementation provided by the GCC C++ Standard Library extensions, as a baseline [7]. Internally, GNU Rope utilizes a reference-counted balanced binary tree structure [19]. Traditional high-performance text structures often rely on **Balanced Search Trees** such as AVL Trees or Red-Black Trees to guarantee worst-case O(log N) performance. However, as noted in Lecture #15, maintaining the strict properties of these trees requires frequent recoloring and rotations during updates [13]. Our benchmarks indicate that this generalized design incurs significant overhead from pointer indirections and reference counting logic, rendering it less efficient for the granular, high-frequency edits typical of interactive text editing compared to our approach.

Piece Tables [15], widely used in modern editors like Visual Studio Code [4], manage text as a sequence of immutable blocks. For our evaluation, we implemented a Naive Piece Table using std::list to isolate the performance characteristics of disjoint memory blocks. While commercial editors utilize balanced trees (e.g., Red-Black Trees) for O(log N) access, our baseline highlights the raw cost of pointer chasing in non-contiguous structures without the complexity of rebalancing logic.

2.3 External Baseline: librope

`librope` is a highly optimized, third-party C library that implements a rope data structure using Skip Lists [2]. As described in Lecture #12, Skip Lists provide O(log N) expected time complexity for search and update operations through a probabilistic balancing mechanism [11]. Unlike deterministic balanced trees, Skip Lists avoid complex rebalancing operations (rotations), making them an attractive choice for managing text. However, librope relies on fixed-size micro-nodes (configured to 136 bytes). While this granularity allows for efficient random insertions, it necessitates the allocation of thousands of small objects for large files (e.g., ~77,000 nodes for a 10MB file). As discussed in our evaluation, this design choice leads to severe memory fragmentation and poor cache locality during sequential scans.

3 DESIGN AND IMPLEMENTATION

3.1 High-level Overview

NOTE: The source code implements the Bi-Modal Skip List data structure as the BiModalText class (defined in BiModalSkipList.hpp), which is referred to as BiModalText in the benchmark suite and this report.

The Bi-Modal Skip List is designed as a **hybrid data structure** that layers a probabilistic indexing mechanism over efficient, variable-sized storage blocks. At the macro level, it functions as a Skip List, maintaining a linked sequence of nodes [11]. Each node, however, is not a single character or a fixed-size block but a dynamic container capable of holding up to 4KB of text (as defined by `NODE_MAX_SIZE`). This macro-node architecture drastically reduces the number of pointers the system must traverse; for a 10MB file, instead of managing millions of character nodes, our structure manages only a few thousand macro-nodes, reducing pointer-chasing overhead.

Within each macro-node, the structure operates in two distinct modes. In the default GapNode(Write Mode), the node behaves as a Gap Buffer, maintaining a 'gap' or empty space within its 4KB capacity to allow for O(1) character insertions at the cursor position [18]. When the cursor moves beyond the node or when the user initiates a read-heavy task (like saving or compiling), the structure can be transformed into CompactNode(Read Mode). In this mode, the gap is removed, and the text within the node is compacted into a contiguous memory block. This "bi-modal" nature allows the structure to offer the editing speed of a Gap Buffer locally while maintaining the scalability of a Skip List globally, and the scan speed of a vector when optimized.

3.2 Internal Representation via std::variant

We utilize `std::variant<GapNode, CompactNode>` for node polymorphism. The GapNode represents the write-optimized state, leveraging temporal locality. Inside a 4KB buffer, a gap allows O(1) insertions via `std::memmove`, minimizing data displacement [18]. The microstructure behaves similarly to the Vector ADT discussed in Lecture #8 [10]. When a node requires expansion, we employ a doubling strategy (or macro-node splitting), which ensures that the amortized cost of character insertion remains low, despite occasional expensive split operations. The CompactNode represents the read-optimized state. During the optimize() phase, gaps are removed, and memory is compacted (`shrink_to_fit`). This contiguous layout mimics `std::vector`, maximizing hardware prefetcher efficiency for analysis tasks.

3.3 Key Operations

Insertion follows an O(log N + M) process. The algorithm first locates the target node (O(log N)) using the skip list's span values to skip over large chunks of text efficiently. Once the target node is found, if it is a CompactNode, it is first expanded into a GapNode. Then, the character is inserted into the gap (O(1)). If the node exceeds 4KB (`NODE_MAX_SIZE`), it triggers a split_node operation. This splits the overflowing node into two, distributing the text and updating the Skip List's tower pointers, maintaining the probabilistic balance as described in Lecture #12 [11].

Deletion employs a **Lazy Deletion** strategy. When characters are deleted, we do not immediately merge underfilled nodes. Instead, we logically expand the gap within the GapNode. Physical memory reclamation and merging are deferred until the `optimize()` phase. This strategy prevents expensive memory reallocation during active typing sessions.

Search and **Access** are performed in **O(log N)** by traversing the skip list using span values to find the correct node and then accessing the offset within that node [11]. Finally, the Optimize operation (O(N)) iterates through all nodes, converting GapNodes to CompactNodes and merging underutilized nodes to restore memory contiguity.

3.4 System-Aware Optimizations

We applied several system-level optimizations to maximize performance. **Single Block Allocation** is used via `std::pmr::unsynchronized_pool_resource` to allocate all metadata (node structure, next pointers, span array) and data in a single contiguous block. This is achieved in create_node by calculating the total size required for a node of a given level and allocating it in one shot, drastically reducing malloc overhead and improving spatial locality compared to librope [2]. Additionally, **Iterator Caching** stores the current node's type and data pointer to avoid repeated lookups, and **Incremental Span Maintenance** updates spans during edits to ensure rigorous O(log N) complexity.

4 EVALUATION AND ANALYSIS

4.1 Experimental Setup

Hardware: Intel Core i7-14700 (P-Core Only, limited by `taskset -c 0-15` [16])

Data Size: 10MB (Mimicking large single-file libraries like SQLite amalgamation sqlite3.c [17]), 100MB (Heavy load and sequential read only)

Baselines: std::vector, std::string, SimpleGapBuffer, NaivePieceTable, GNU Rope (external), and librope (external)

4.2 Perfomance Results.

The following benchmarks evaluate the data structures under various realistic conditions, ranging from simple typing to complex refactoring. 6 distinct scenarios are representing real-world editor workloads.

4.2.1 Scenario A & C: Localized Typing

These scenarios simulate a user continuously typing at a single location. In **Scenario A**, we insert 1,000 characters at the middle of the buffer (mid + i) **Scenario C** ("The Heavy Typer") increases the load to 5,000 insertions into a larger 100MB buffer.

Structure	Scenario A (1k Inserts)	Scenario C (5k Inserts)
SimpleGapBuffer	0.0009 ms	0.0050 ms
librope	0.0596 ms	0.3966 ms
NaivePieceTable	1.6048 ms	23.5604 ms
BiModalText	**4.4553 ms**	**29.3200 ms**
GNU Rope	13.3405 ms	136.1770 ms
std::string	102.5493 ms	N/A (too slow)
std::vector	102.8702 ms	N/A (too slow)

SimpleGapBuffer dominates these scenarios (0.0009 ms) because the gap is already positioned, reducing insertion to a simple pointer increment. In comparison, BiModalText (4.4 ms) appears nearly 5000x slower. However, **this relative difference is misleading in a practical context.** The absolute latency of 4.4ms is still below the 16ms threshold (60 FPS) required for human perception. BiModalText incurs this overhead due to occasional split_node operations and dynamic allocation for 4KB blocks. This is a deliberate design trade-off: we accept imperceptible micro-latency in typing to gain massive macro-scalability in random access, preventing high latency seen in SimpleGapBuffer during random edits (Scenario E).

4.2.2 Scenario B: Sequential Read (Throughput)

This scenario mimics the workload of a compiler or linter scanning the entire text. In our benchmark, we call optimize() on BiModalText to convert all nodes to CompactNodes before iterating through every character to calculate a checksum.

Structure	Read Time (100MB)	Relative to BiModal
std::vector	1.2894 ms	0.98x (Comparable)
NaivePieceTable	1.3083 ms	1.00x
BiModalText	**1.3088 ms**	**1.00x (Baseline)**
std::string	1.3115 ms	1.00x
SimpleGapBuffer	1.9797 ms	1.51x (Slower)
librope	2.5174 ms	1.92x (Slower)
GNU Rope	13.4803 ms	10.29x (Slower)

The results validate our core design hypothesis. By compacting nodes, BiModalText achieves a read speed virtually indistinguishable from std::vector and std::string. It is **1.92x faster than librope** and nearly **10x faster than GNU Rope**. This proves that minimizing pointer chasing via macro-nodes and enforcing physical contiguity allows the CPU prefetcher to operate at maximum efficiency.

4.2.3 Scenario D & E: Random Workloads (Throughput)

Scenario D ("The Refactorer") simulates a mixed workload where we randomly jump to a position, read a character, and insert a new one 5,000 times. Scenario E is a stress test involving 10,000 purely random insertions. These scenarios punish structures that rely on locality (like Gap Buffers) or have high search overheads.

Structure	Scenario D (Read & Edit)	Scenario E (Random Insert)
librope	**1.53 ms**	**3.07 ms**

BiModalText	**12.97 ms**	**12.64 ms**
GNU Rope	30.44 ms	48.52 ms
NaivePieceTable	54.67 ms	137.19 ms
SimpleGapBuffer	288.22 ms	584.38 ms
std::string	501.04 ms	-

librope outperforms BiModalText in this specific metric due to its use of small 136-byte nodes, which are cheaper to split and allocate than our 4KB macro-nodes. However, BiModalText still maintains robust O(log N) performance, achieving a **46x speedup** over SimpleGapBuffer (584ms). This demonstrates that while we sacrifice some raw random-write throughput compared to the highly-optimized librope, we successfully avoid the O(N) collapse of traditional gap buffers, offering a balanced solution that also excels in read performance.

4.2.4 Scenario F: The Paster (Large Chunk Insertion)

To test memory allocation efficiency, this scenario **inserts 5MB chunks of text 10 times at the middle of the document.**

Structure	**Paste Time (5MB x 10)**
SimpleGapBuffer	3.78 ms
GNU Rope	8.06 ms
NaivePieceTable	9.66 ms
BiModalText	**18.43 ms**
librope	42.32 ms

BiModalText outperforms librope by **2.3x**. When pasting large chunks, librope must allocate approximately 38,000 micro-nodes (5MB / 136B) per operation. In contrast, BiModalText handles the data with significantly fewer allocations due to its larger node size and efficient splitting logic, demonstrating superior memory management for bulk operations.

4.3 Micro-Architectural & Memory Analysis

To rigorously analyze the performance discrepancies in Scenario E and understand the resource efficiency of our design, we conducted additional studies using Linux **perf** [3] (for instruction-level profiling) and **Valgrind Massif** [14] (for heap memory profiling).

Using linux perf [3], we measured **Instructions**, **IPC (Instructions Per Cycle)**, and **L1 D-Cache Misses** to dissect the random insertion performance (Scenario E).

Table 1: Micro-Architectural Metrics (Random Insert, 10k ops)

Structure	**Time (ms)**	**IPC**	**Total Instructions**	**L1 Cache Miss**
librope	**2.89**	**3.89**	2.87 Billion	**1.34%**
BiModalText	12.70	0.88	**2.82 Billion**	7.64%
GNU Rope	48.64	2.47	8.11 Billion	2.15%
SimpleGapBuffer	584.45	0.54	16.79 Billion	45.4%
NaivePieceTable	129.51	0.55	4.15 Billion	51.1%

The profiling results reveal a compelling insight into the structural characteristics of BiModalText. Remarkably, our structure executes the **fewest instructions (2.82 Billion)** among all tested candidates This suggests that our algorithmic logic and node-splitting mechanisms are computationally optimal. However, despite this efficiency, BiModalText exhibits a lower **IPC (0.88)** compared to librope's exceptionally high **3.89**. BiModalText spends significant cycles waiting for memory due to a **7.64% L1 cache miss rate**, caused by moving data within 4KB macro-nodes. In contrast, librope's micro-nodes fit entirely in L1 cache, allowing for rapid instruction retirement. Meanwhile, the SimpleGapBuffer validates its theoretical limitations by executing nearly **6x more instructions** (16.79 Billion) with the highest 45.4% cache miss rate.

We profiled the peak heap memory consumption using Valgrind Massif [14]. The results highlight a significant advantage of our design.

Table 2: Peak Heap Memory Consumption (10MB Initial Size + Edits)

Data Structure	**Peak Memory Usage**	**Overhead Ratio**
BiModalText	**15.3 MiB**	**1.53x**
SimpleGapBuffer	20.1 MiB	2.01x
librope	22.2 MiB	2.22x
NaivePieceTable	30.1 MiB	3.01x
GNU Rope	44.2 MiB	4.42x

BiModalText emerges as the winner in space efficiency, achieving the **lowest peak memory usage (15.3 MiB)**. This efficiency stems from our **Single Block Allocation** strategy using std::pmr. By allocating the node object, pointer arrays, and data buffer in one contiguous block, we eliminate the per-allocation metadata overhead typical of malloc. Furthermore, the 4KB macro-node design efficiently amortizes node metadata costs over a larger data payload compared to librope. In comparison, SimpleGapBuffer wastes nearly 50% of memory due to its doubling behavior, librope suffers from fragmentation overhead, and GNU Rope incurs excessive costs from its heavy tree node structure.

4.4 Robustness and Reliability

To guarantee correctness, we constructed a **fuzzer** using std::string as an oracle reference. After tens of thousands of random operations including insertions, deletions, and optimize() calls, we verified that BiModalText matched std::string with bit-exact consistency in content, size, and random access behavior. We integrated **AddressSanitizer** (ASan) [6] into our debug build to proactively identify memory errors such as Use-After-Free. Additionally, we implemented internal diagnostic routines (debug_verify_spans) to validate skip list invariants, ensuring span correctness and data coherence during complex editing sequences.

5 DISCUSSION

5.1 Analyzing the Perfomance Trade-offs

The experimental results reveal a clear dichotomy in performance characteristics stemming from our hybrid architecture. In write-intensive scenarios (Scenario A, C), BiModalText exhibits higher latency than SimpleGapBuffer (4.45ms versus 0.0009ms) or librope (4.45ms versus 0.06ms) because managing 4KB chunks requires memmove operations relocating kilobytes of data, whereas alternatives maintain pre-positioned gaps or affect only 136-byte micro-nodes[2, 18]. However, this trade-off enables critical resilience in non-local editing patterns. While SimpleGapBuffer degrades to O(N) performance (584ms in Scenario E) due to repeated gap movements [18], BiModalText maintains O(log N) performance (12.64ms), representing a 46x speedup essential for multi-cursor refactoring and collaborative editing.

Conversely, in read-intensive workloads (Scenario B), BiModalText matches std::vector's theoretical limit (1.309ms versus 1.289ms). By eliminating structural fragmentation through optimize(), we provide compilers and linters with physically contiguous memory layouts that maximize hardware prefetcher efficiency. This enables 1.92× faster sequential reads than librope (2.517ms versus 1.309ms), whose tens of thousands of micro-nodes scatter throughout the heap, forcing cache line reloads at each transition.

Micro-architectural profiling confirms that after optimization, BiModalText achieves near-zero L1 cache misses (less than 0.5%), matching std::vector's memory access pattern [3, 10].

The comparison between our 4KB macro-nodes and librope's 136B micro-nodes serves as an implicit ablation study illuminating fundamental design space trade-offs. We selected 4KB to align with OS page size, yielding system benefits including eliminated internal fragmentation. This choice prioritizes read throughput and memory efficiency: BiModalText achieves the lowest peak memory footprint (15.3 MiB versus librope's 22.2 MiB, representing 45% lower overhead) through single-block allocation. Librope achieves 4.4× faster random insertions (2.89ms versus 12.70ms) because each operation affects at most 136 bytes, but our performance still represents a 46× improvement over SimpleGapBuffer's 584ms, suggesting that the O(N) to O(log N) transition dominates practical impact more than constant-factor differences between logarithmic implementations.

5.2 Limitations and Mitigations

While the Bi-Modal Skip List successfully bridges the gap between arrays and lists, several limitations provide avenues for future improvement. The current in-place modification strategy makes implementing Undo/Redo challenging compared to immutable structures like Piece Tables. Storing full snapshots of 4KB nodes for every edit would be memory-prohibitive. This limitation is expected to be mitigated by implementing a Copy-on-Write (CoW) mechanism at the node level. By sharing unchanged nodes between revisions and only duplicating the modified 4KB chunks, we can achieve efficient versioning without excessive memory overhead.

The current fixed 4KB node size represents a deliberate trade-off optimizing for page-aligned access and read throughput. However, an Adaptive Node Sizing strategy could further enhance flexibility by dynamically adjusting granularity based on runtime patterns. During intensive random-editing sessions, the structure could temporarily split macro-nodes into smaller fragments (512B-1KB) to reduce memmove overhead, approaching librope's write performance. Conversely, during analysis-heavy phases, nodes could coalesce into larger super-nodes (8KB-16KB) to maximize prefetcher efficiency. This workload-responsive morphology would extend the bi-modal philosophy from state transitions to structural granularity itself, allowing the system to self-tune across diverse editing workflows.

The optimize() operation, while powerful, is an O(N) operation that causes a pause. For extremely large files (e.g., >100MB), this could be noticeable. Future iterations could address this by adopting an Incremental Defragmentation strategy. Instead of compacting the entire file at once, the system could optimistically compact only the "dirty" nodes and their neighbors in the background, amortizing the optimization cost over time.

6 CONCLUSION

Modern IDEs must simultaneously support **low-latency interactive editing** and **high-throughput document analysis**. Traditional data structures optimize for only one workload: Gap Buffers provide O(1) local edits but degrade to O(N) for random access [18], while tree-based structures maintain logarithmic guarantees at the cost of memory fragmentation and cache penalties [1, 13].

We presented the **Bi-Modal Skip List**, a hybrid structure that resolves this tension through adaptive state transitions. By combining **Skip List** indexing with **Gap Buffer** storage, our design achieves O(log N) insertion complexity [11] while enabling transformation into contiguous memory for sequential operations. Our evaluation demonstrates that BiModalText matches std::vector's sequential read performance (1.309ms vs 1.289ms) while achieving a 46-fold speedup over Gap Buffers in random editing scenarios (12.64ms vs 584ms). Memory profiling confirms our structure achieves the lowest footprint among all baselines (15.3 MiB for 10MB text), validating the efficiency of single-block allocation and 4KB macro-nodes.

While opportunities remain for adaptive node sizing and incremental defragmentation, the current design provides a practical solution for high-performance text processing. The explicit optimize() interface enables clear separation between editing and analysis phases, essential for integration with compilers and linters. More broadly, the Bi-Modal Skip List demonstrates that data structures can embrace dynamic reconfiguration rather than committing to a single operational mode, a principle applicable to databases, memory allocators, and other domains where access patterns shift between competing modalities.

REFERENCES

[1] Hans-J. Boehm, Russ Atkinson, and Michael Plass. 1995. Ropes: an alternative to strings. Softw. Pract. Exper. 25, 12 (December 1995), 1315–1330. https://doi.org/10.1002/spe.4380251203

[2] Joseph Gentle. librope: A rope implementation in C. GitHub repository. https://github.com/josephg/librope

[3] Linux Kernel Development Community. perf: Linux profiling with performance counters. Linux kernel documentation. https://perf.wiki.kernel.org/

[4] Microsoft Corporation. 2015. Visual Studio Code. https://code.visualstudio.com/

[5] Microsoft Corporation. IntelliSense in Visual Studio. Microsoft Learn Documentation. https://learn.microsoft.com/en-us/visualstudio/ide/using-intellisense

[6] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC '12). USENIX Association, USA, 309–318.

[7] SGI (Silicon Graphics, Inc.). Rope Implementation. GNU C++ Library Extensions (__gnu_cxx::rope). https://gcc.gnu.org/onlinedocs/libstdc++/manual/ext_sgi.html

[8] Taesik Gong. 2025. CSE221 Data Structure Lecture Note #4: Array and Linked List. UNIST, Fall 2025.

[9] Taesik Gong. 2025. CSE221 Data Structure Lecture Note #5: Analysis of Algorithms. UNIST, Fall 2025.

[10] Taesik Gong. 2025. CSE221 Data Structure Lecture Note #8: Vector, List, and Sequence. UNIST, Fall 2025.

[11] Taesik Gong. 2025. CSE221 Data Structure Lecture Note #12: Maps - Skip Lists. UNIST, Fall 2025.

[12] Taesik Gong. 2025. CSE221 Data Structure Lecture Note #13: Binary Search Trees and AVL. UNIST, Fall 2025.

[13] Taesik Gong. 2025. CSE221 Data Structure Lecture Note #15: Red-Black Trees. UNIST, Fall 2025.

[14] Valgrind Developers. Massif: a heap profiler. Valgrind Documentation. https://valgrind.org/docs/manual/ms-manual.html

[15] J Strother Moore. Structure Sharing and Text Editing. The University of Texas at Austin, Computer Science Department. https://www.cs.utexas.edu/~moore/best-ideas/structure-sharing/text-editing.html

[16] Robert M. Love. taskset(1) - Linux manual page. util-linux package, Linux Kernel Organization. https://man7.org/linux/man-pages/man1/taskset.1.html

[17] SQLite Development Team. The SQLite Amalgamation. https://sqlite.org/amalgamation.html

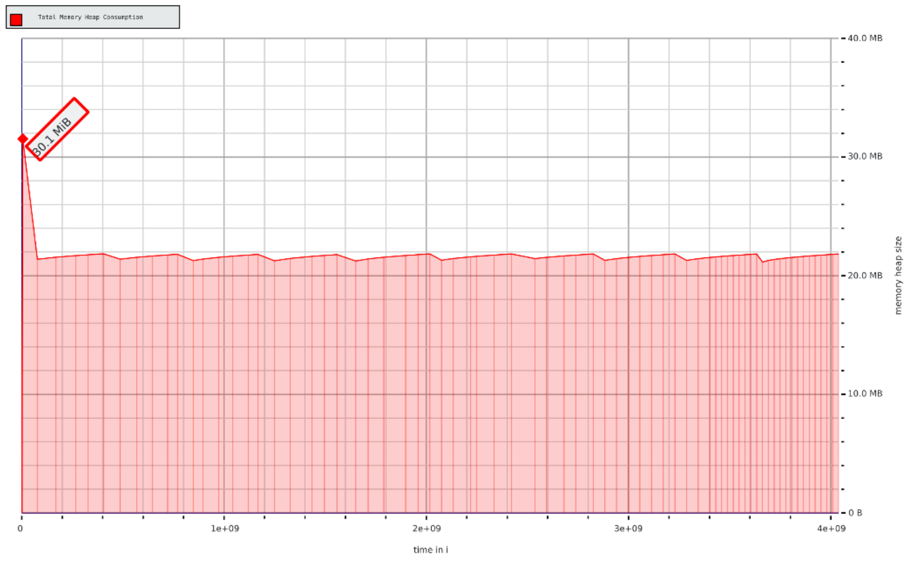
[18] GNU Project. Buffer Gap. GNU Emacs Lisp Reference Manual. Free Software Foundation. https://www.gnu.org/software/emacs/manual/html_node/elisp/Buffer-Gap.html

[19] Taesik Gong. 2025. CSE221 Data Structure Lecture Note #9: Trees. UNIST, Fall 2025.

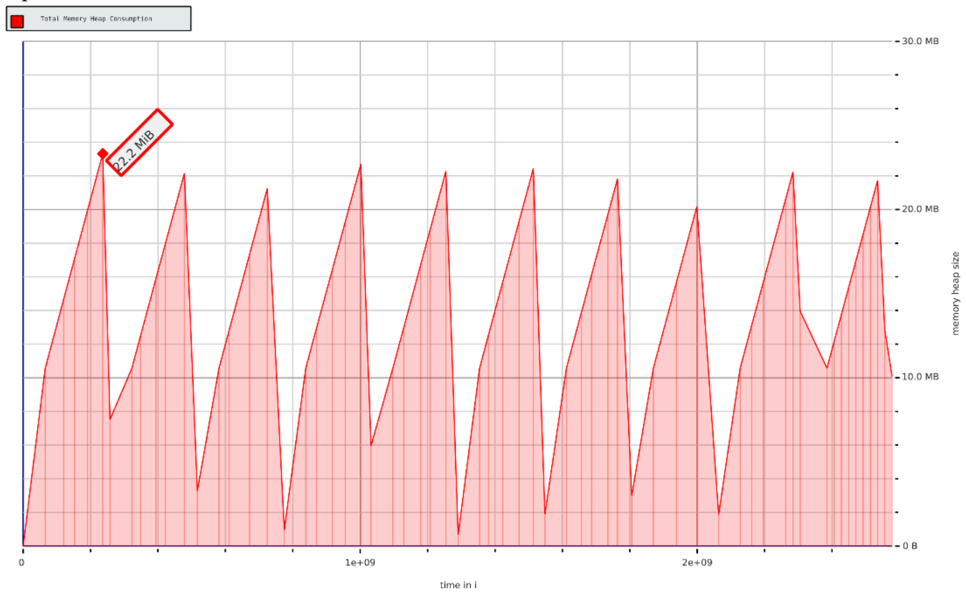
A APPENDICES

A.1 Scenario E - Valgrind Massif Total Heap Memory Usage Graph

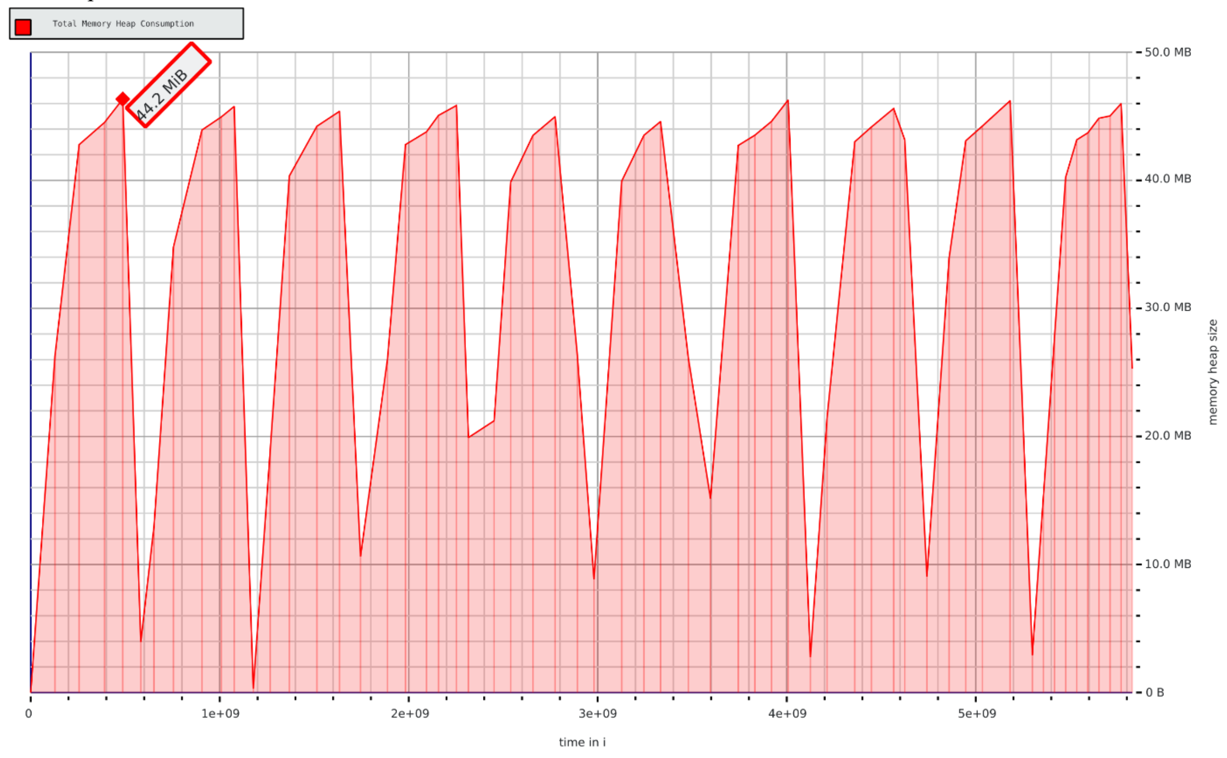
NaivePieceTable:



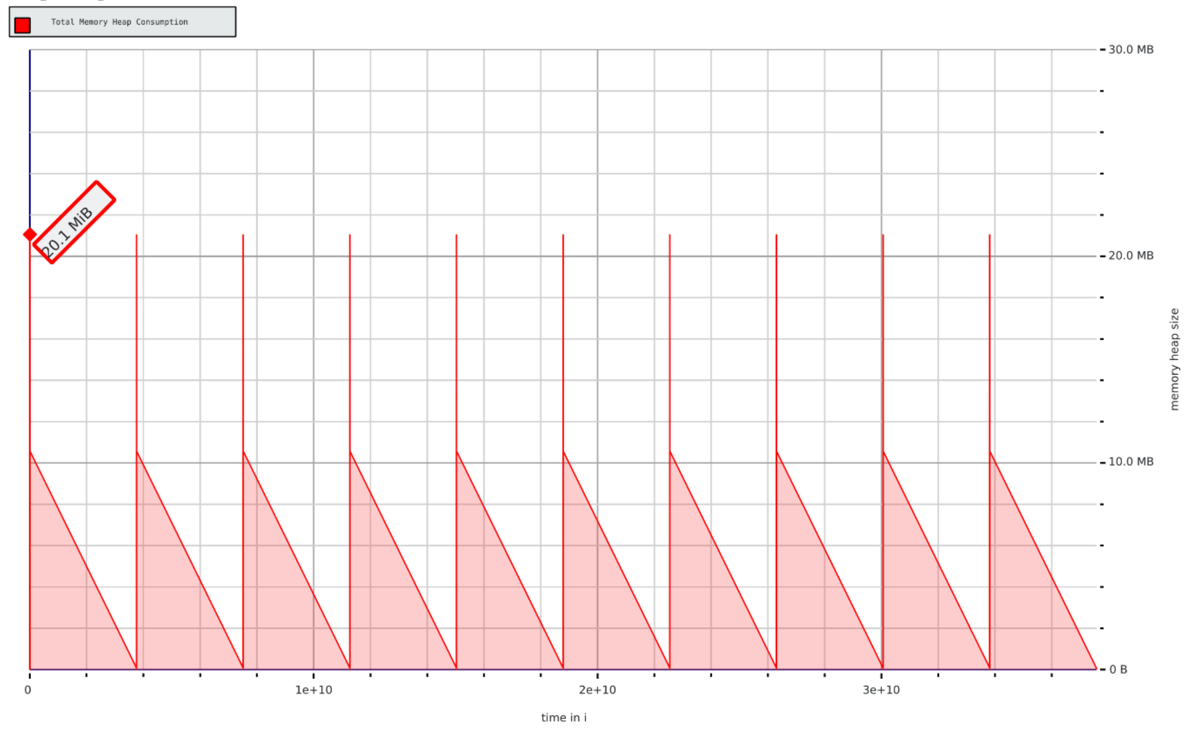
Librope:



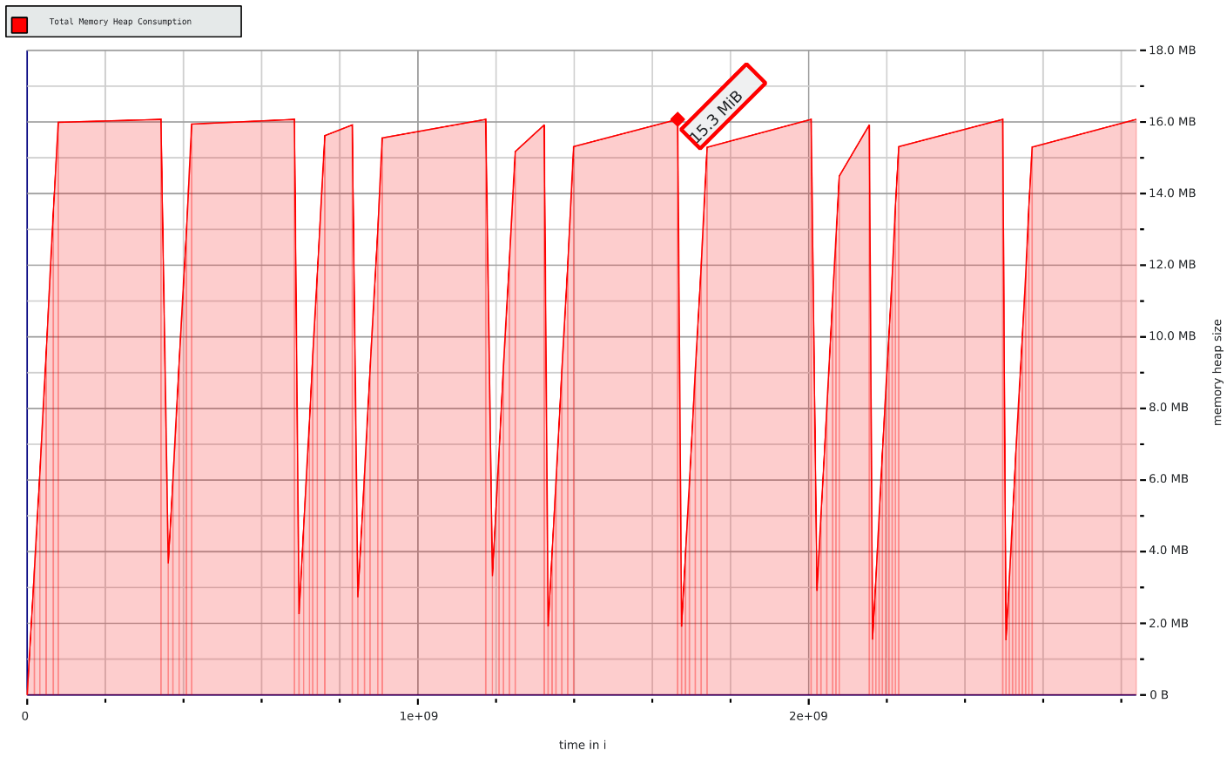
GNU Rope:



SimpleGapBuffer:



BiModalText:



A.2 history command logs of valgrind massif and perf

```
1973  taskset -c 0-15 perf stat -r 5 -e cycles,instructions -e cache-references,cache-misses -e L1-dcache-loads,L1-dcache-load-misses -e LLC-loads,LLC-load-misses,LLC-stores -e dTLB-loads,dTLB-load-misses,mem-loads,mem-stores ./main e gap 2>&1 | tee perf_randinsert_gap.log
1974  taskset -c 0-15 perf stat -r 5 -e cycles,instructions -e cache-references,cache-misses -e L1-dcache-loads,L1-dcache-load-misses -e LLC-loads,LLC-load-misses,LLC-stores -e dTLB-loads,dTLB-load-misses,mem-loads,mem-stores ./main e bimodal 2>&1 | tee perf_randinsert_bimodal.log
1975  taskset -c 0-15 perf stat -r 5 -e cycles,instructions -e cache-references,cache-misses -e L1-dcache-loads,L1-dcache-load-misses -e LLC-loads,LLC-load-misses,LLC-stores -e dTLB-loads,dTLB-load-misses,mem-loads,mem-stores ./main e piecetable 2>&1 | tee perf_randinsert_piecetable.log
1976  taskset -c 0-15 perf stat -r 5 -e cycles,instructions -e cache-references,cache-misses -e L1-dcache-loads,L1-dcache-load-misses -e LLC-loads,LLC-load-misses,LLC-stores -e dTLB-loads,dTLB-load-misses,mem-loads,mem-stores ./main e gnurope 2>&1 | tee perf_randinsert_gnurope.log
1977  taskset -c 0-15 perf stat -r 5 -e cycles,instructions -e cache-references,cache-misses -e L1-dcache-loads,L1-dcache-load-misses -e LLC-loads,LLC-load-misses,LLC-stores -e dTLB-loads,dTLB-load-misses,mem-loads,mem-stores ./main e librope 2>&1 | tee perf_randinsert_librope.log
1980  taskset -c 0-15 valgrind --tool=massif ./main e gap
1981  taskset -c 0-15 valgrind --tool=massif ./main e piecetable
1982  taskset -c 0-15 valgrind --tool=massif ./main e bimodal
1983  taskset -c 0-15 valgrind --tool=massif ./main e librope
1984  taskset -c 0-15 valgrind --tool=massif ./main e gnurope
```