Lab 5: Implementation of a Mark-Sweep Garbage Collector

Jithendra Gannavarapu (2025MCS2743)
Manmadha Rao Kuttum (2025MCS2101)

January 16, 2026

Abstract

This report documents the extension of the Lab 4 Virtual Machine to support dynamic memory management. A "Stop-the-World" Mark-Sweep Garbage Collector (GC) was implemented to manage heap-allocated objects. The system supports root discovery from the stack and global memory, transitive reachability via recursive marking, and automatic reclamation of unreachable memory. Correctness was verified through a suite of 7 stress tests, confirming memory safety and the absence of leaks under high-load conditions.

1 Problem Statement

The objective was to transform the integer-only VM into a system capable of managing dynamic object lifecycles. The specific requirements included:

- **Heap Allocator:** A mechanism to allocate `Pair` objects dynamically.
- **Root Discovery:** Identifying live objects referenced by the Stack and Global Memory.
- **Mark Phase:** Recursively tracing object graphs to handle transitive reachability (e.g., Linked Lists).
- **Sweep Phase:** Reclaiming memory from objects that are no longer reachable.

2 System Design

2.1 Data Structures

To support mixed types (Integers vs. Pointers), we introduced a tagged union structure.

- **Object Header:** Every heap object inherits a common header containing a `marked` boolean flag and a `next` pointer to track allocations in a linked list.
- **Value Type:** The VM stack was upgraded from `int32_t` to a `Value` struct that can hold either a raw integer or an `Object*` pointer.

```
struct Object {
    ObjectType type;
    bool marked;         // GC Mark Bit
    struct Object* next; // Heap Tracking List
};

struct Value {
    ValueType type;      // VAL_INT or VAL_OBJ
    union {
        int32_t asInt;
        Object* asObj;
    };
};
```

Listing 1: The Object and Value Structures

2.2 Heap Organization

The "Heap" is implemented as a singly linked list maintained by the VM class (`Object* objects`).

- **Allocation:** New objects are inserted at the head of this list ($O(1)$ complexity).
- **Sweeping:** The GC traverses this list to find unmarked nodes and unlink them.

3 Garbage Collector Implementation

3.1 Phase 1: Mark (Reachability Analysis)

The marking process uses a Depth-First Search (DFS) approach.

1. **Root Scanning:** The GC iterates through the `stack` and `memory` vectors.
2. **Recursive Marking:** When an object is marked, the GC checks if it is a container (e.g., `OBJ_PAIR`). If so, it recursively marks the children (`left` and `right`).

This logic handles **Cyclic References** (e.g., $A \to B \to A$) by checking the `marked` flag before recursing, preventing infinite loops.

3.2 Phase 2: Sweep (Reclamation)

The sweep phase iterates through the global object list.

- If `marked == true`: The object is live. The flag is reset to `false` for the next cycle.
- If `marked == false`: The object is garbage. It is unlinked from the list and `delete` is called to free system memory.

4 Heap Observability & Liveness Tracking

A key challenge in this lab was determining the exact state of the heap (Living vs. Dead objects) for verification. We achieved this through the following mechanisms:

4.1 Tracking the "Living"

Liveness is determined by the **Mark Phase**. We implemented a `marked` bit in every object header.

- **Default State:** All objects are initialized with `marked = false`.
- **Discovery:** During GC, if an object is reachable from the stack, its bit is flipped to `true`.
- **Recursion:** If the living object is a `Pair`, the GC recursively visits its children (`left` and `right`), flipping their bits to `true` as well.

4.2 Tracking the "Dead"

Dead objects are identified implicitly during the **Sweep Phase**.

- The VM maintains a global linked list (`Object* objects`) of *every* allocated object.
- The GC traverses this list. Any node where `marked == false` is considered "Dead" (unreachable).
- These nodes are unlinked and immediately passed to the C++ `delete` operator to reclaim OS memory.

4.3 Heap Status Reporting

To verify the system, we implemented a `GCStats` structure. We calculate the heap status by counting nodes in the linked list before and after the GC cycle:

$$DeadObjects = Count_{Initial} - Count_{Post-Sweep}$$

This calculation provided the quantitative proof required for the "Stress Allocation" tests.

5 Requirements Fulfillment

The implementation satisfies all functional and non-functional requirements as defined in the Lab 5 Problem Statement.

5.1 Functional Requirements

1. **Heap Allocator:** *Achieved via:* The `allocatePair` function. It encapsulates C++ `new` and automatically inserts the new object into the VM's internal tracking list (`objects`), ensuring no allocation is lost.
2. **Root Discovery:** *Achieved via:* The `gc()` driver. It iterates through both the VM `stack` and the Global `memory` vector, treating every `VAL_OBJ` found as a root for the Mark phase.
3. **Mark Phase:** *Achieved via:* The `markObject` function. It implements a Depth-First Search (DFS). It correctly handles `OBJ_PAIR` types by recursively marking `pair->left` and `pair->right`, ensuring transitive reachability.

4. **Sweep Phase:** *Achieved via:* The `sweep()` function. It linearly traverses the heap linked list ($O(N)$). It performs two actions: deleting unmarked nodes (reclaiming memory) and resetting marked nodes (preparing for the next cycle).

5.2 Non-Functional Requirements

1. **Memory Safety:** *Achieved via:* The `VM` destructor (`~VM`). When the VM halts, the destructor walks the remaining heap list and frees all pointers, ensuring zero memory leaks in the host system.

2. **Correctness Under Stress:** *Achieved via:* The `testDeepGraph` and `testStress` cases. We validated the system with 10,000 linked objects to ensure the recursive marker does not crash and the allocator can handle high-frequency creation/deletion cycles without fragmentation or leaks.

6 Performance / Stress Evaluation

To satisfy the requirement for a brief evaluation of GC behavior under heavy allocation, we conducted a stress test involving the rapid allocation and subsequent reclamation of 50,000 objects in a single burst.

6.1 Methodology

The test measured the total wall-clock time required to:

1. Allocate 50,000 `Pair` objects (approx. 1.6 MB of heap metadata).

2. Trigger the Garbage Collector to identify all objects as unreachable (Mark Phase).

3. Sweep and `delete` all 50,000 objects (Sweep Phase).

This test validates the implementation's efficiency and ensures the recursion depth handling is robust enough for large datasets.

6.2 Results

The stress test was executed on a standard development environment. The results confirm the $O(N)$ linear time complexity of the sweep phase.

Metric	**Value**
Total Objects Processed	50,000
Objects Freed	50,000 (100%)
Memory Leaks	0
Execution Time	**4 ms**
Throughput	$\approx$ 12,500 objects/ms

Table 1: GC Stress Test Performance Metrics

6.3 Analysis

The system demonstrated high throughput with an average processing speed of 12,500 objects per millisecond. The total execution time of just 4 ms for a complete cycle of 50,000 nodes indicates that the overhead introduced by the tagged-union object headers and the mark-sweep algorithm is minimal. The successful reclamation of 100% of the allocated memory confirms the allocator and collector are memory-safe under high-load conditions.

7 Validation and Testing

The system was validated using a custom C++ test driver that manipulates the VM state directly. All 7 test cases defined in the requirements passed successfully.

7.1 Test Results Summary

Test Case	Description	Result
1.6.1 Basic Reachability	Object on stack survives GC.	**PASSED**
1.6.2 Unreachable Objects	Object not on stack is deleted.	**PASSED**
1.6.3 Transitive Reachability	$A \rightarrow B$ chain is preserved.	**PASSED**
1.6.4 Cyclic References	$A \leftrightarrow B$ cycle is preserved.	**PASSED**
1.6.5 Deep Object Graph	10,000-node linked list (Recursion).	**PASSED**
1.6.6 Closure Capture	Closure $\rightarrow$ Env/Func preserved.	**PASSED**
1.6.7 Stress Allocation	10,000 objects alloc/freed.	**PASSED**

Table 2: GC Verification Results

7.2 Stress Test Analysis

The `Stress Allocation` test created 10,000 objects (approx. 320KB of memory) without retaining references.

- **Pre-GC Heap:** 10,000 Objects (Active).
- **Post-GC Heap:** 0 Objects (Active).
- **Outcome:** The GC successfully reclaimed 100% of the garbage, confirming zero memory leaks.

8 Advanced Edge Case Validation

To verify the robustness of the Garbage Collector beyond the standard requirements, three advanced edge cases were implemented. These scenarios specifically target common failure modes in memory management systems, such as reference cycles and shared sub-graphs.

8.1 Test Scenarios

1. **Orphaned Cycle (The "Island of Isolation"):** Two objects refer to each other ($A \leftrightarrow B$) but are detached from the Root set. *Significance:* A simple Reference Counting collector would fail to reclaim this memory (resulting in a leak) because the reference counts would never drop to zero. Our Mark-Sweep GC correctly identifies them as unreachable from the roots and frees both.

2. **Diamond Graph (Shared Children):** A root points to objects B and C, and both B and C point to a shared child D. *Significance:* This stress-tests the recursion logic. The marker must visit D via B, mark it, and then correctly handle the second visit via C without infinite looping or double-counting.

3. **Self-Reference:** An object A points to itself ($A \rightarrow A$). *Significance:* Validates that the simplest form of cycle does not cause infinite recursion in the Mark phase.

8.2 Edge Case Results

All edge cases were verified using a dedicated test driver (`test_gc_edge.cpp`).

Edge Case	**Graph Structure**	**Outcome**
Orphaned Cycle	$A \leftrightarrow B$ (Unreachable)	**PASSED** (2 Freed)
Diamond Graph	Root $\rightarrow B, C \rightarrow D$	**PASSED** (4 Survived)
Self-Reference	$A \rightarrow A$ (Reachable)	**PASSED** (1 Survived)

Table 3: Advanced Edge Case Verification Results

9 Conclusion

This lab successfully extended the virtual machine with a robust memory management subsystem. The Mark-Sweep algorithm was chosen for its ability to handle cyclic references, a significant advantage over Reference Counting. By integrating the GC directly into the VM's `Value` system, we ensured backwards compatibility with integer arithmetic while enabling complex data structures. The successful completion of the "Deep Object Graph" test (10,000 nodes) demonstrates the implementation's stability and correctness under load.