Heap Heap Hooray: Memory Management

Tyler Gutowski, Trevor Schiff,
Dr. Ryan Stansifer (client)

Task	Description	Tyler	Trevor
Implementation of root detection	Implement “root detection”, or the process of finding root objects by walking the call stack and inspecting registers/locals.	0.0	1.0
Implementation of the Mark Phase algorithm	Implement the mark phase, which traverses the object graph and marks all allocations that can be reached.	0.5	0.5
Implementation of the Sweep Phase algorithm	Implement the sweep phase, which frees objects that haven’t been marked.	0.8	0.2
Mark-and-Sweep testing	Write test scripts to ensure the mark-sweep algorithm works.	0.5	0.5
GC parameterization	Ensure that different garbage collectors can be selected when running the code.	0.4	0.0
Implementation/testing of the Copying Algorithm	Implement the new sweep phase, which consists of moving all “marked” objects to a new heap. This defragments the memory. Write test scripts to ensure the copying algorithm works.	N/A	N/A
Comparing metrics between different GC algorithms	Compare the garbage collecting algorithms. On hold until parameterization is completed.	N/A	N/A
Dividing the heap into multiple parts for defragmentation	Split the heap into multiple parts to ensure that we can implement the “copying” algorithm.	N/A	N/A

Basic Overview: Mark-and-Sweep

When we try to allocate memory, but none is available, perform:

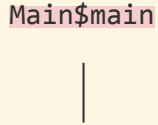
1. Mark Phase
    a. Starting from the roots, the garbage collector traverses the graph of all objects.
        i. “Roots” of the graph are anything reachable without touching the heap (local variables, CPU registers).
    b. Each object that can be reached from the roots is marked.
        i. Same object pointer detection heuristic, as seen in decrementing refcount of children
2. Sweep Phase
    a. Iterate over all live heap allocations.
    b. Objects that are not marked are considered garbage (unreachable).
    c. Free the memory for these objects.
    d. Unmark all objects to reset GC state.

Implementation: Mark-and-Sweep

Program

```
class Main {
    public static void main(String[] a) {
        new Test().execute();
    }
}

class Test {
    public int execute() {
        Node dummy = new Node();
        makeCycle();
    }

    public void makeCycle() {
        Node one = new Node();
        Node two = new Node();

        one.setNext(two);
        two.setNext(one);
    }
}
```

Graph

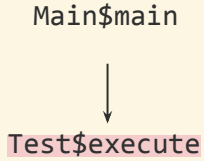


Implementation: Mark-and-Sweep

Program

```
class Main {
    public static void main(String[] a) {
        new Test().execute();
    }
}

class Test {
    public int execute() {
        Node dummy = new Node();
        makeCycle();
    }

    public void makeCycle() {
        Node one = new Node();
        Node two = new Node();

        one.setNext(two);
        two.setNext(one);
    }
}
```

Graph

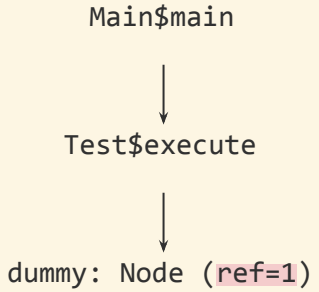


Implementation: Mark-and-Sweep

Program

```
class Main {
    public static void main(String[] a) {
        new Test().execute();
    }
}

class Test {
    public int execute() {
        Node dummy = new Node();
        makeCycle();
    }

    public void makeCycle() {
        Node one = new Node();
        Node two = new Node();

        one.setNext(two);
        two.setNext(one);
    }
}
```

Graph

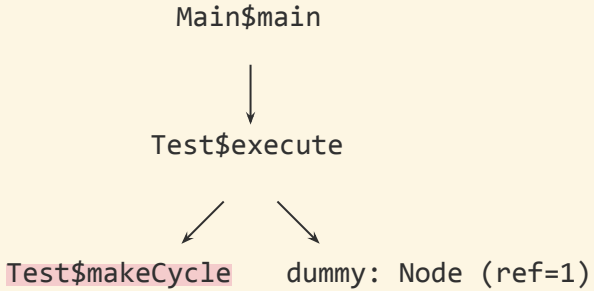


Implementation: Mark-and-Sweep

Program

```
class Main {
    public static void main(String[] a) {
        new Test().execute();
    }
}

class Test {
    public int execute() {
        Node dummy = new Node();
        makeCycle();
    }

    public void makeCycle() {
        Node one = new Node();
        Node two = new Node();

        one.setNext(two);
        two.setNext(one);
    }
}
```

Graph

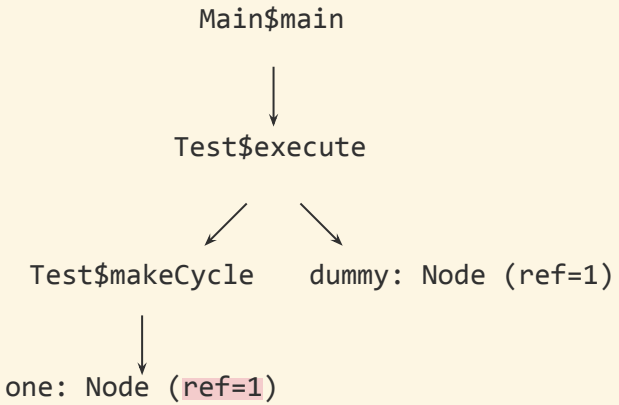


Implementation: Mark-and-Sweep

Program

```
class Main {
    public static void main(String[] a) {
        new Test().execute();
    }
}

class Test {
    public int execute() {
        Node dummy = new Node();
        makeCycle();
    }

    public void makeCycle() {
        Node one = new Node();
        Node two = new Node();

        one.setNext(two);
        two.setNext(one);
    }
}
```

Graph

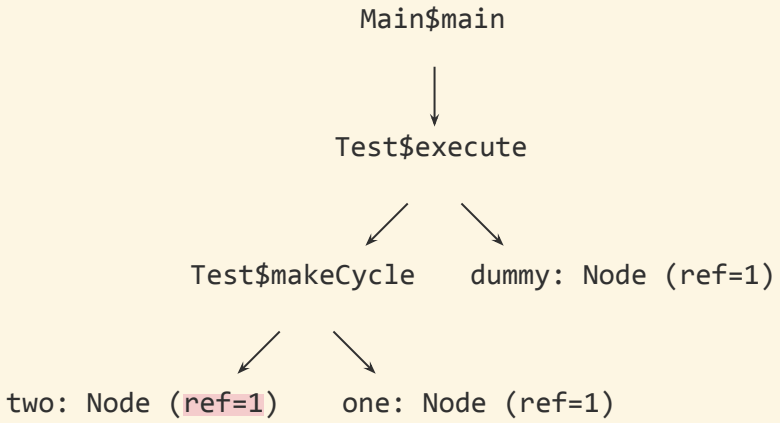


Implementation: Mark-and-Sweep

Program

```
class Main {
    public static void main(String[] a) {
        new Test().execute();
    }
}

class Test {
    public int execute() {
        Node dummy = new Node();
        makeCycle();
    }

    public void makeCycle() {
        Node one = new Node();
        Node two = new Node();

        one.setNext(two);
        two.setNext(one);
    }
}
```

Graph

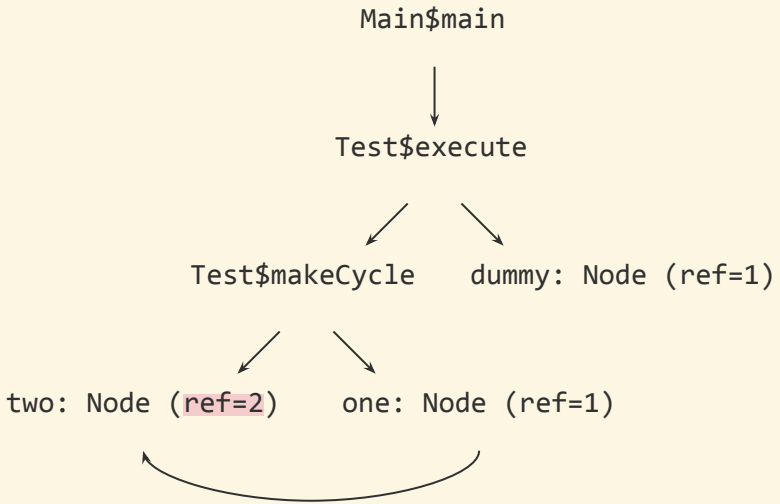


Implementation: Mark-and-Sweep

Program

```
class Main {
    public static void main(String[] a) {
        new Test().execute();
    }
}

class Test {
    public int execute() {
        Node dummy = new Node();
        makeCycle();
    }

    public void makeCycle() {
        Node one = new Node();
        Node two = new Node();

        one.setNext(two);
        two.setNext(one);
    }
}
```

Graph

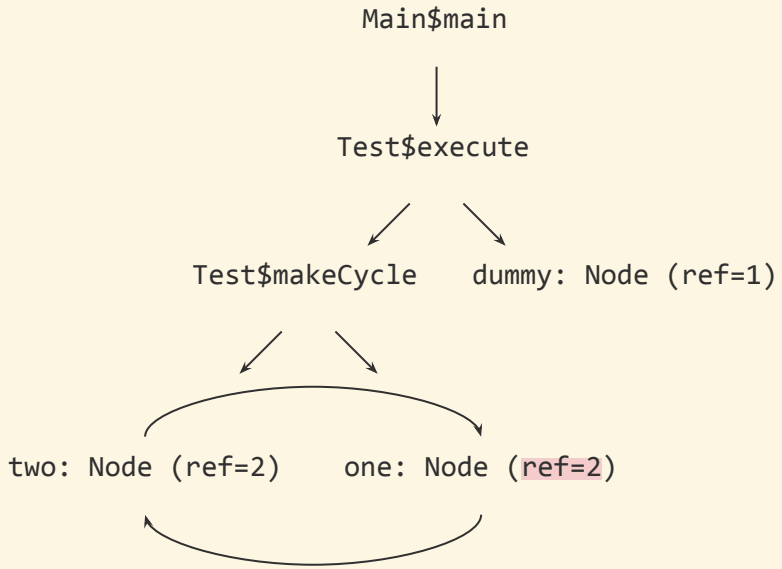


Implementation: Mark-and-Sweep

Program

```
class Main {
    public static void main(String[] a) {
        new Test().execute();
    }
}

class Test {
    public int execute() {
        Node dummy = new Node();
        makeCycle();
    }

    public void makeCycle() {
        Node one = new Node();
        Node two = new Node();

        one.setNext(two);
        two.setNext(one);
    }
}
```

Graph

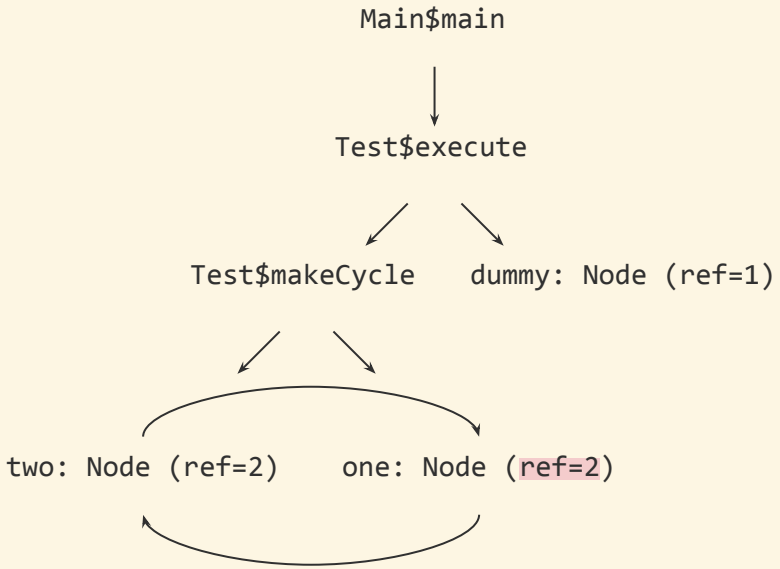


Implementation: Mark-and-Sweep

Program

```
class Main {
    public static void main(String[] a) {
        new Test().execute();
    }
}

class Test {
    public int execute() {
        Node dummy = new Node();
        makeCycle();
    }

    public void makeCycle() {
        Node one = new Node();
        Node two = new Node();

        one.setNext(two);
        two.setNext(one);
    }
}
```

Graph



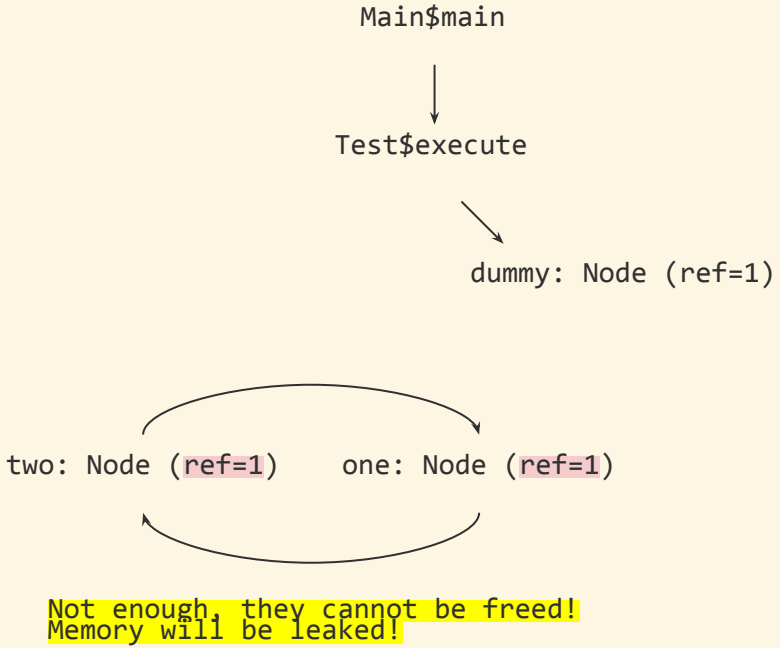
End of scope, decrement refcount of 'one' & 'two'

Implementation: Mark-and-Sweep

Program

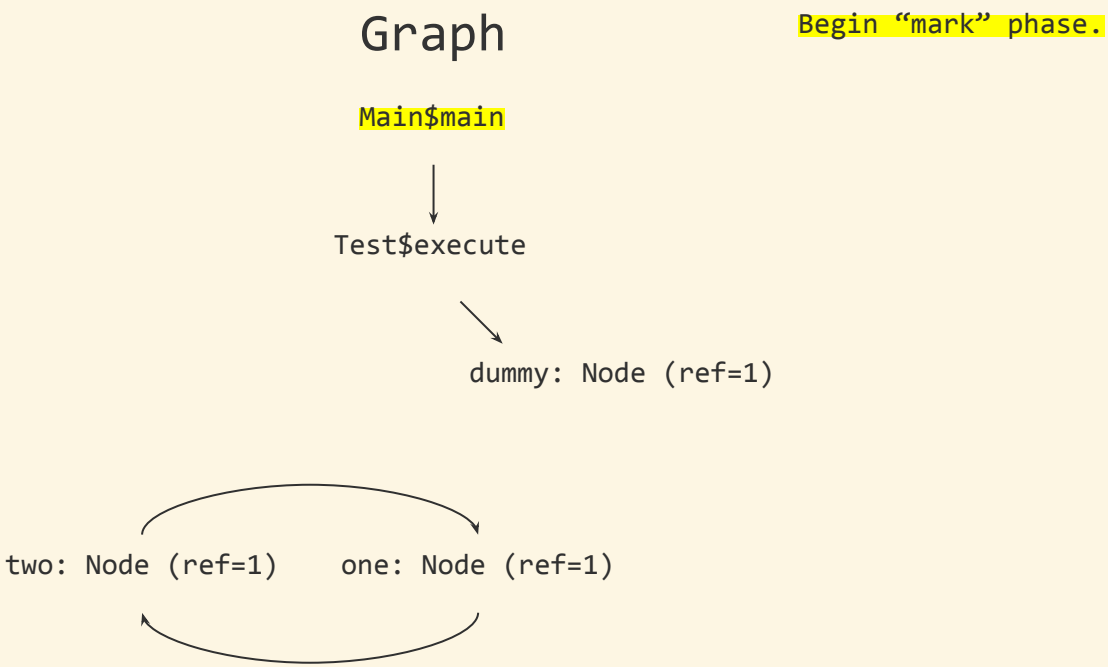
```
class Main {
    public static void main(String[] a) {
        new Test().execute();
    }
}

class Test {
    public int execute() {
        Node dummy = new Node();
        makeCycle();
    }

    public void makeCycle() {
        Node one = new Node();
        Node two = new Node();

        one.setNext(two);
        two.setNext(one);
    }
}
```

Graph

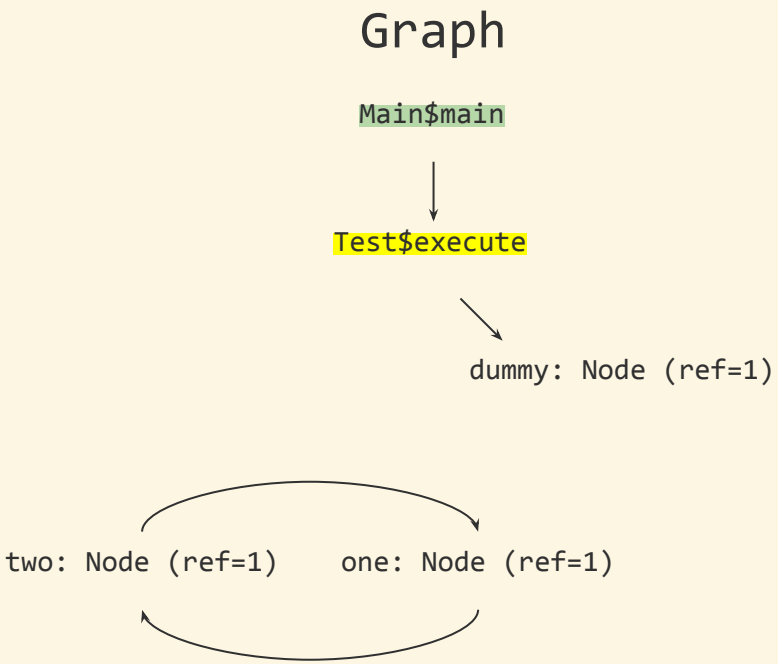


Not enough, they cannot be freed!
Memory will be leaked!

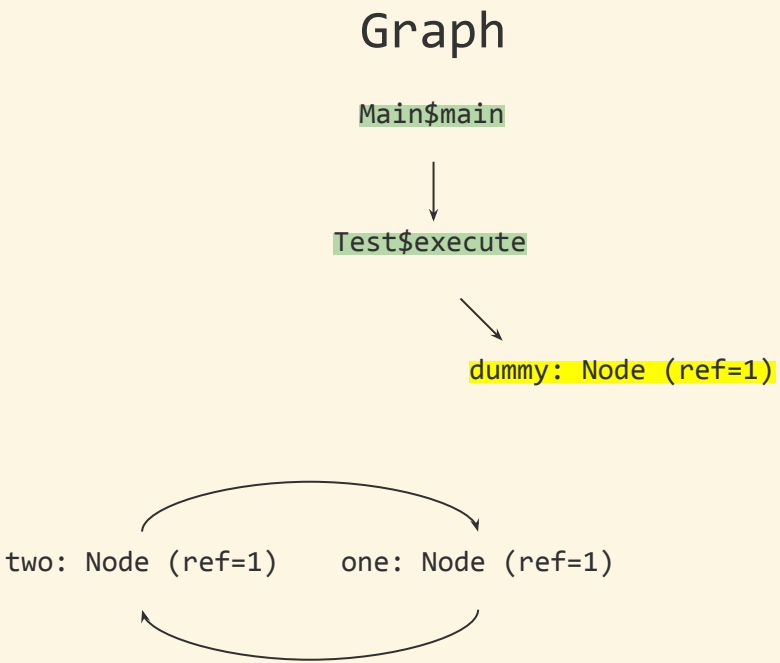
Implementation: Mark-and-Sweep



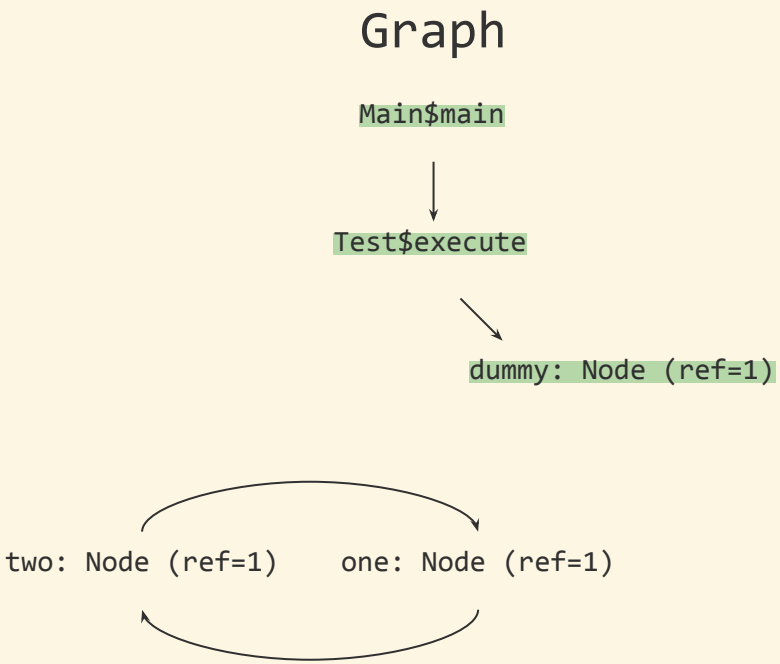
Implementation: Mark-and-Sweep



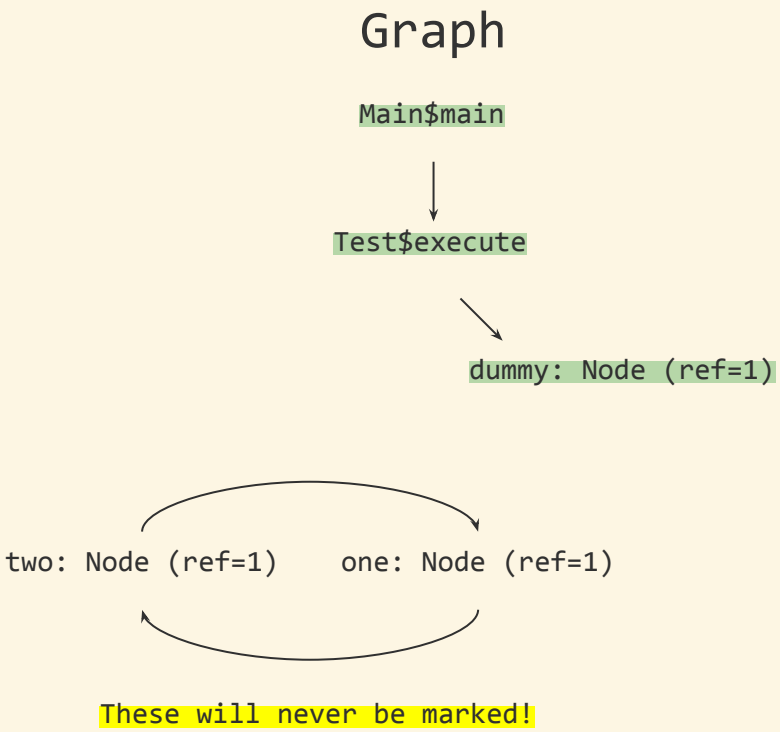
Implementation: Mark-and-Sweep



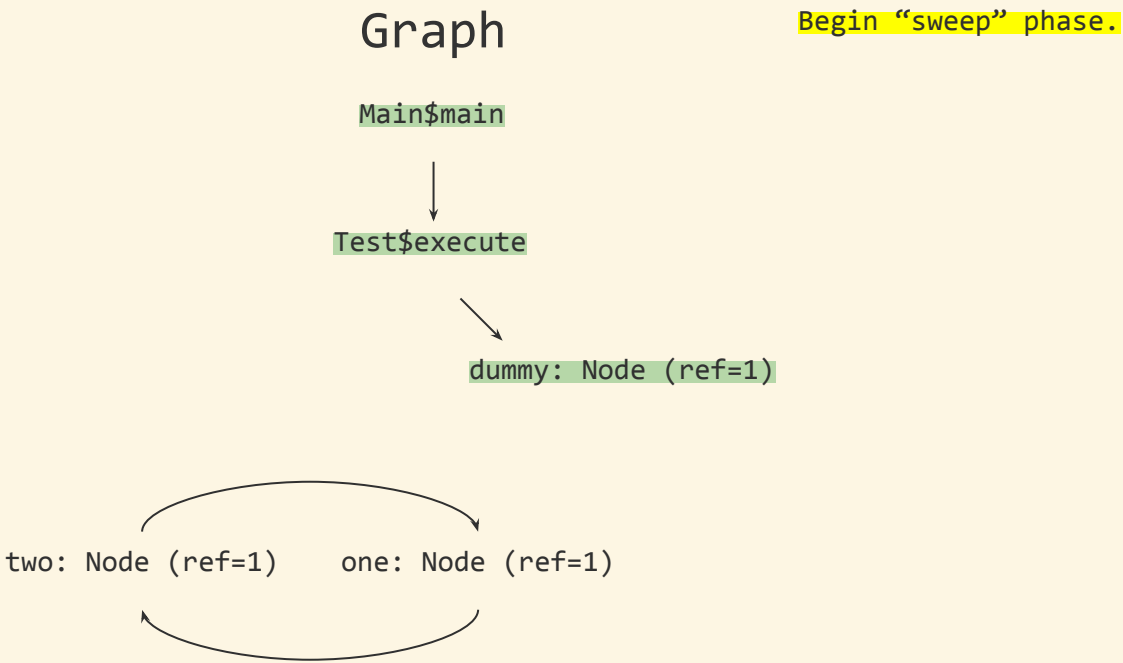
Implementation: Mark-and-Sweep



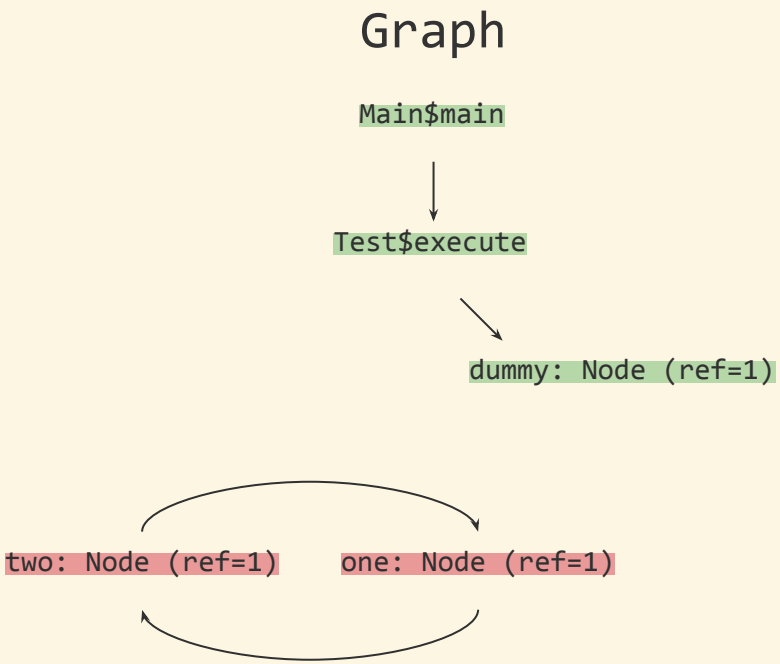
Implementation: Mark-and-Sweep



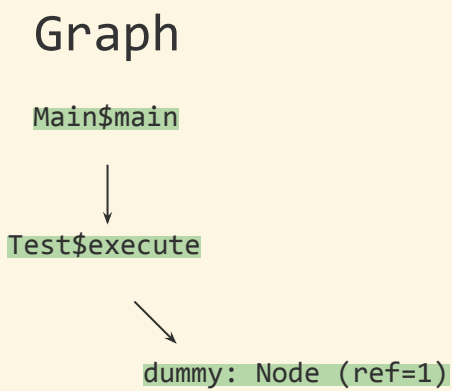
Implementation: Mark-and-Sweep



Implementation: Mark-and-Sweep

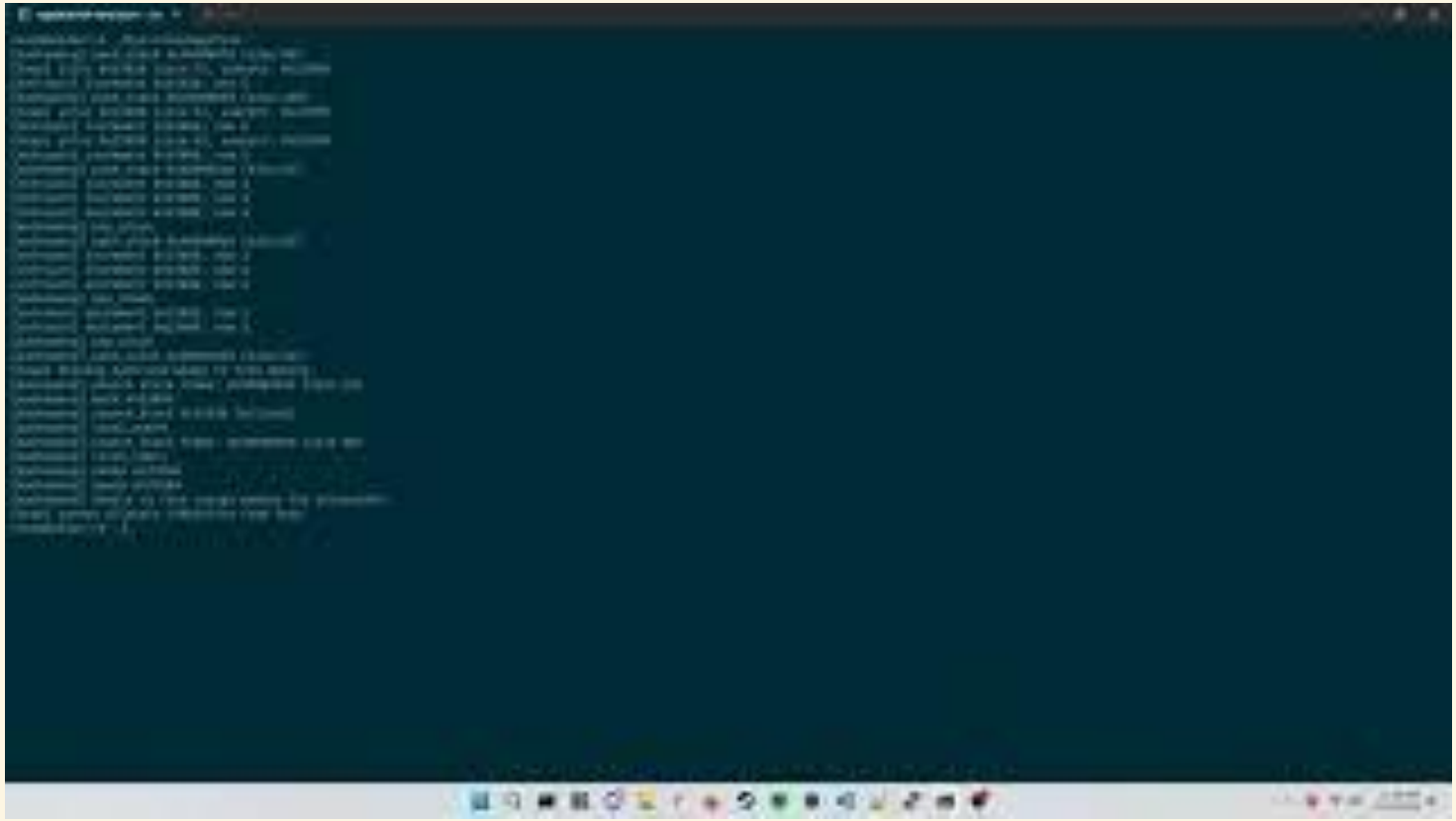


Implementation: Mark-and-Sweep



Unreachable objects released.

Demo: Mark-and-Sweep



Implementation: Mark-and-Sweep

```
root@debian:~# ./CyclicGarbageTest
[marksweep] push_stack 0x40800668 (size:100)
[heap] alloc 0x25028 (size:4), userptr: 0x25030
[refcount] increment 0x25028, now 1
[marksweep] push_stack 0x40800600 (size:100)
[heap] alloc 0x25068 (size:4), userptr: 0x25070
[refcount] increment 0x25068, now 1
[heap] alloc 0x25088 (size:4), userptr: 0x25090
[refcount] increment 0x25088, now 1
[marksweep] push_stack 0x408005a0 (size:92)
[refcount] increment 0x25088, now 2
[refcount] increment 0x25088, now 3
[refcount] decrement 0x25088, now 2
[marksweep] pop_stack
[marksweep] push_stack 0x408005a0 (size:92)
[refcount] increment 0x25068, now 2
[refcount] increment 0x25068, now 3
[refcount] decrement 0x25068, now 2
[marksweep] pop_stack
[refcount] decrement 0x25068, now 1
[refcount] decrement 0x25088, now 1
[marksweep] pop_stack
[heap] Running mark-and-sweep to free memory.
```

```
class CyclicGarbageTest {
    public int execute() {
        Node dummy;
        int[] buffer;

        dummy = new Node();
        this.leak();
        buffer = new int[2000000000];

        return 0;
    }

    public int leak() {
        Node one;
        Node two;

        one = new Node();
        two = new Node();

        one.setNext(two);
        two.setNext(one);

        return 0;
    }
}
```

Implementation: Mark-and-Sweep

```
[marksweep] search stack frame: 0x40800668 (size:100)
  sp->lreg[0]=408006C8
  sp->lreg[1]=00000000
  sp->lreg[2]=00000000
  sp->lreg[3]=00000000
  sp->lreg[4]=00000000
  sp->lreg[5]=00000000
  sp->lreg[6]=00000000
  sp->lreg[7]=3FFFEF70
  sp->ioreg[0]=408006C8
  sp->ioreg[1]=00000000
  sp->ioreg[2]=00000000
  sp->ioreg[3]=00000000
  sp->ioreg[4]=00000000
  sp->ioreg[5]=00000000
  local_num=2
  sp->locals[0 (align:2)] = 00025030
[marksweep] mark 0x25028
[marksweep] search alloced block 0x25030
  sp->locals[1 (align:3)] = 000102D4
[marksweep] sweep 0x25068
[marksweep] sweep 0x25088
[heap] cannot allocate 3705032716 from heap
[heap] alloced:
[heap]    addr:0x25028 size:4 ref:1 ← 0x25030 - sizeof(HeapHeader)
```

```
class CyclicGarbageTest {
    public int execute() {
        Node dummy;
        int[] buffer;

        dummy = new Node();
        this.leak();
        buffer = new int[2000000000];

        return 0;
    }

    public int leak() {
        Node one;
        Node two;

        one = new Node();
        two = new Node();

        one.setNext(two);
        two.setNext(one);

        return 0;
    }
}
```

Milestone 4 Goals

- Copying GC as primary implementation
  - Mark-and-Sweep fragments the memory
    - Leaves a lot unusable
  - Defragments heap by copying live allocations into new heap
    - Then releases unreachable allocations
  - Similar to mark-sweep
    - “Copy to new heap” process replaces traditional “sweep” process
- Add parameterization to run specific GC implementations
  - Don’t have to work off specific branches.
- Run tests with all 3 GC implementations
  - Compare tests with vital metrics.