Cornell Bowers C·IS
Computer Science

CS3410: Computer Systems and Organization

LEC26: Memory Safe Languages

Dr. Kevin Laeufer
Monday, December 1, 2025

Credits: Laeufer, Sampson, **Susag**, Weatherspoon

Love Systems? Love Teaching?

TA CS 3410.

Apply By December 3rd →

Computer System Organization and Programming

https://bowers-student-hiring.coecis.cornell.edu/position.cfm?recid=835

Final 3 Lectures

- **Today** - Memory Safety
- **Wed, 12/03** - MapReduce
- **Mon, 12/08** - Review

Important: Today is the last day to declare conflicts and take an alternative final! *(see ed post for details)*

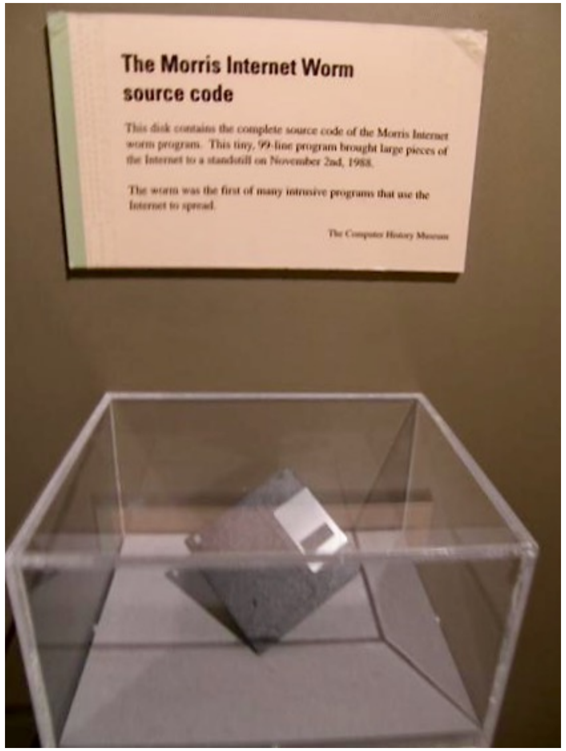
Plan for today.

- **Motivation:** Memory Safety Issues in C and C++
- Buffer Overflow
- Uninitialized Values
- **Automatic Memory Management**
    - Garbage Collection
    - Reference Counting
- **Rust**
    - Ownership and References

Manual Memory Management
is a
Calamitous Failure

Morris Worm

- One of the oldest **computer worms** distributed via the Internet (1988)
- Exploited several vulnerabilities, including a **buffer overflow**
- Viral denial-of-service attack
    - U.S. Government Accountability Office estimates economic impact was between $100K-$10M
- Named after author Robert Tappan Morris
    - First felony conviction under Computer Fraud and Abuse Act
    - 3 years probation, 400 hours of community service, fine of $10,050.



A disk containing the 99-line "Morris Worm" source code used to be on display at the Computer History Museum. (*Wikimedia Commons*)

Robert Tappan Morris

Memory Unsafety Leads to Vulnerabilities

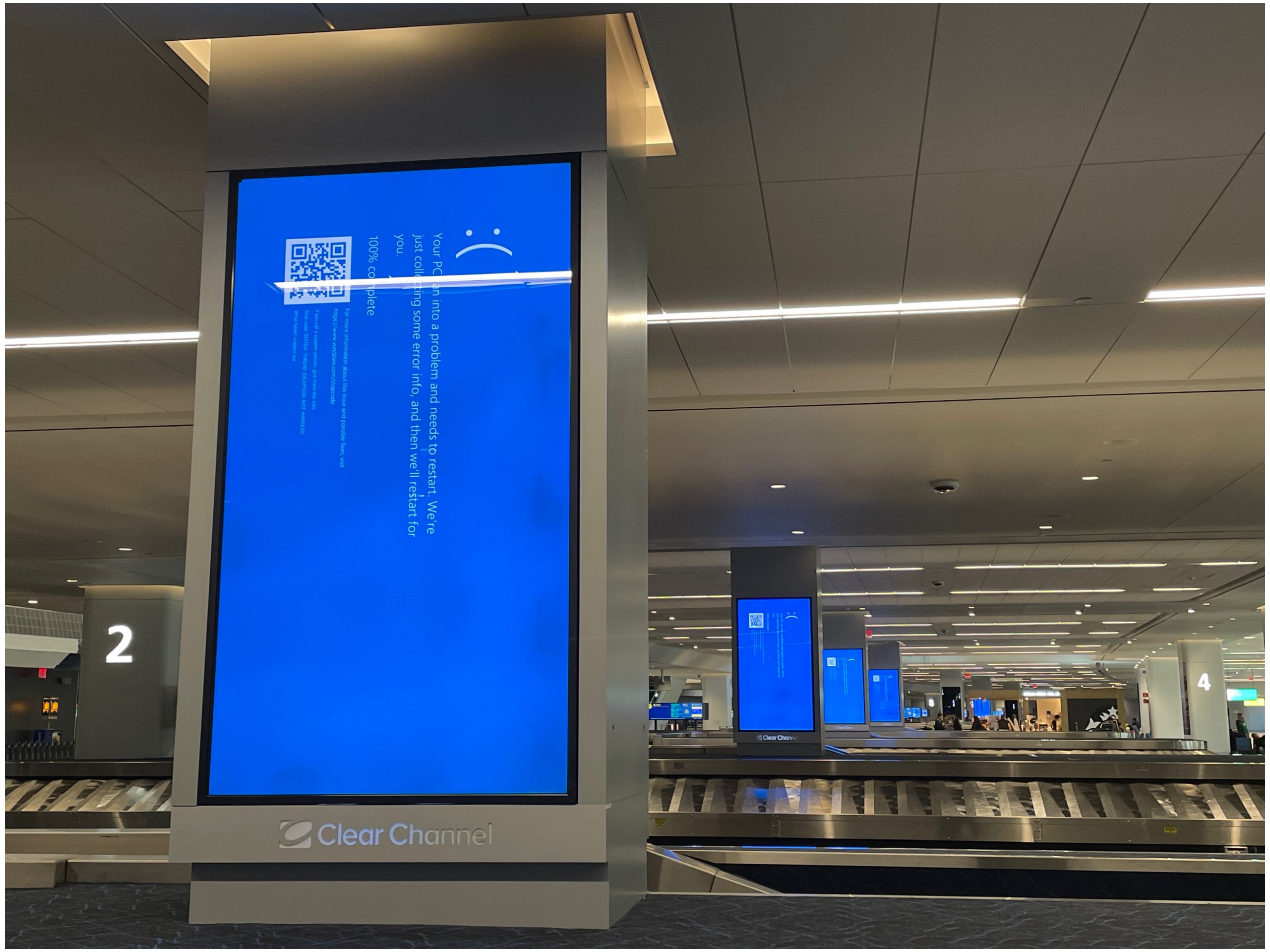
- Heartbleed: security bug in OpenSSL in 2014
  - Directly caused by a **buffer over-read**
  - Allowed attackers to receive passwords, session cookies, etc.
- In 2019, Microsoft found that 70% of all security vulnerabilities were **memory safety violations**
- In 2020, Google reported that 70% of all "severe security bugs" in Chromium were caused by memory safety bugs

Heartbleed logo
(https://heartbleed.com)

Microsoft

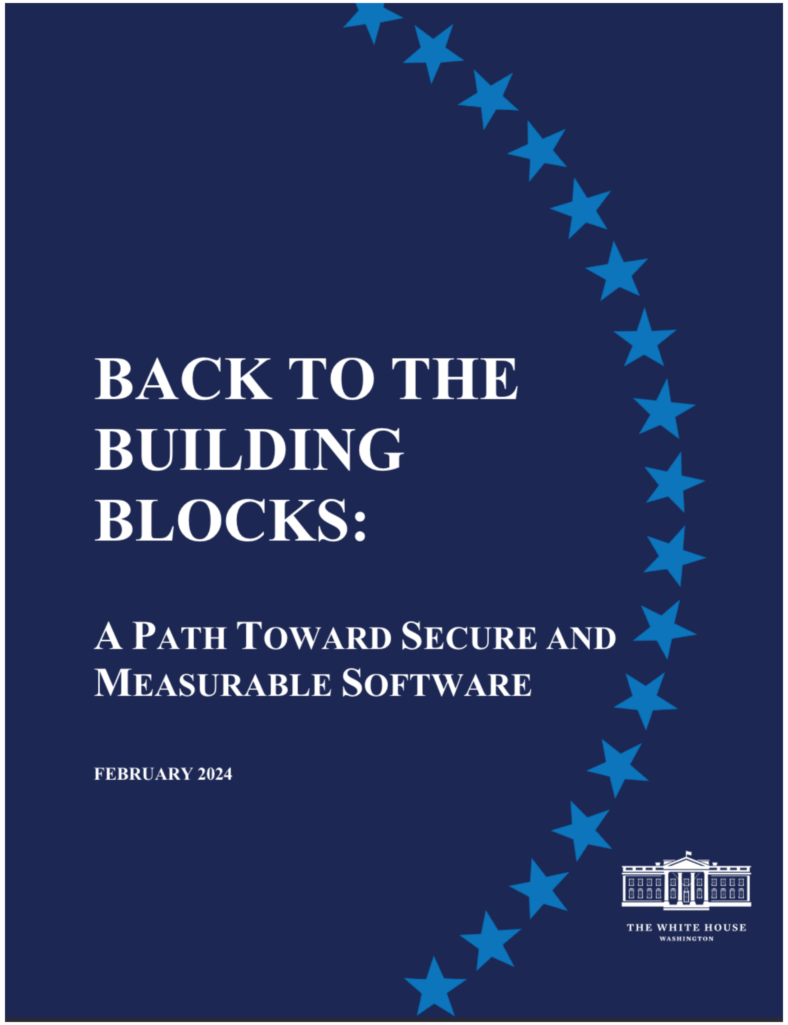
2024 CrowdStrike Outage

- ~8.5 million PCs crashed and were unable to restart across the planet
- Estimated to cost ~$10 billion
- Ultimately due to an **out-of-bounds access**!



Back to the Building Blocks

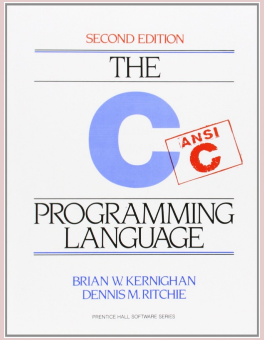
- Biden Administration report encouraging widespread adoption of:
  - **Memory Safe languages**
  - Memory Safe hardware (esp. for use in space)
  - Formal methods



Memory Safe Languages

- Memory **un**safety is a direct product of manual memory management (i.e., breaking the **Laws of the Heap**)
- A **memory safe** language handles memory management *automatically* on the behalf of the developer

Memory Management Styles

Manual	Automated (Dynamic)	Automated (Static)
 C++	Swift, C#, Java, GO, OCaml, Haskell	R

What are memory safe languages?

Languages that prevent the following bugs are called memory safe:

- **Buffer Overflow** (remember A8?)
- **Uninitialized variables**
- **Use after free:** After you `free` memory, you can't use it. *(see Lecture 5)*
- **Double free:** You can only `free` memory once. *(see Lecture 5)*

Note: This list is not exhaustive. There are a couple of less important bugs that we skipped.

How can we prevent Buffer Overflows?

- **“Fat” Pointers** store additional information besides the memory address.
- For an array, we would want to store the address of the first element, as well as the number of elements, allowing for an out-of-bounds check at runtime.
- The biggest downside is that this generally doubles the size of our pointer. However, most often, the length would be stored in a separate variable anyway.
- The bounds check is surprisingly cheap on modern hardware.

Uninitialized Variables

- C allows us to declare variables without assigning them a value.
- Reading such a variable is undefined behavior, since the space it occupies might contain arbitrary data.
- Instead, other languages will require the user to assign a value when declaring the variable, or initialize with a well defined default value (often null or 0).

Use After Free & Double Free

- If we stop using `free`, then we cannot use it incorrectly!
- However, without `free`, we can never release memory that isn't needed anymore.
- Can we build a system that automatically frees memory once it is no longer needed?

Dynamic, Automated Memory Management

Automated Memory Management

- **Goal:** the *language* is responsible for freeing memory that is no longer needed
- Language should **abstract away** memory as an implementation detail (e.g., no raw pointers)
- Two approaches:
  1. **Dynamic**: when memory is freed is decided at **run-time** (i.e., when the program is running)
  2. **Static**: when memory is freed is decided at **compile-time** (i.e., when the program is compiled)

Garbage Collectors (GCs)

Invented by John McCarthy in ~1959 for the **LISP** language

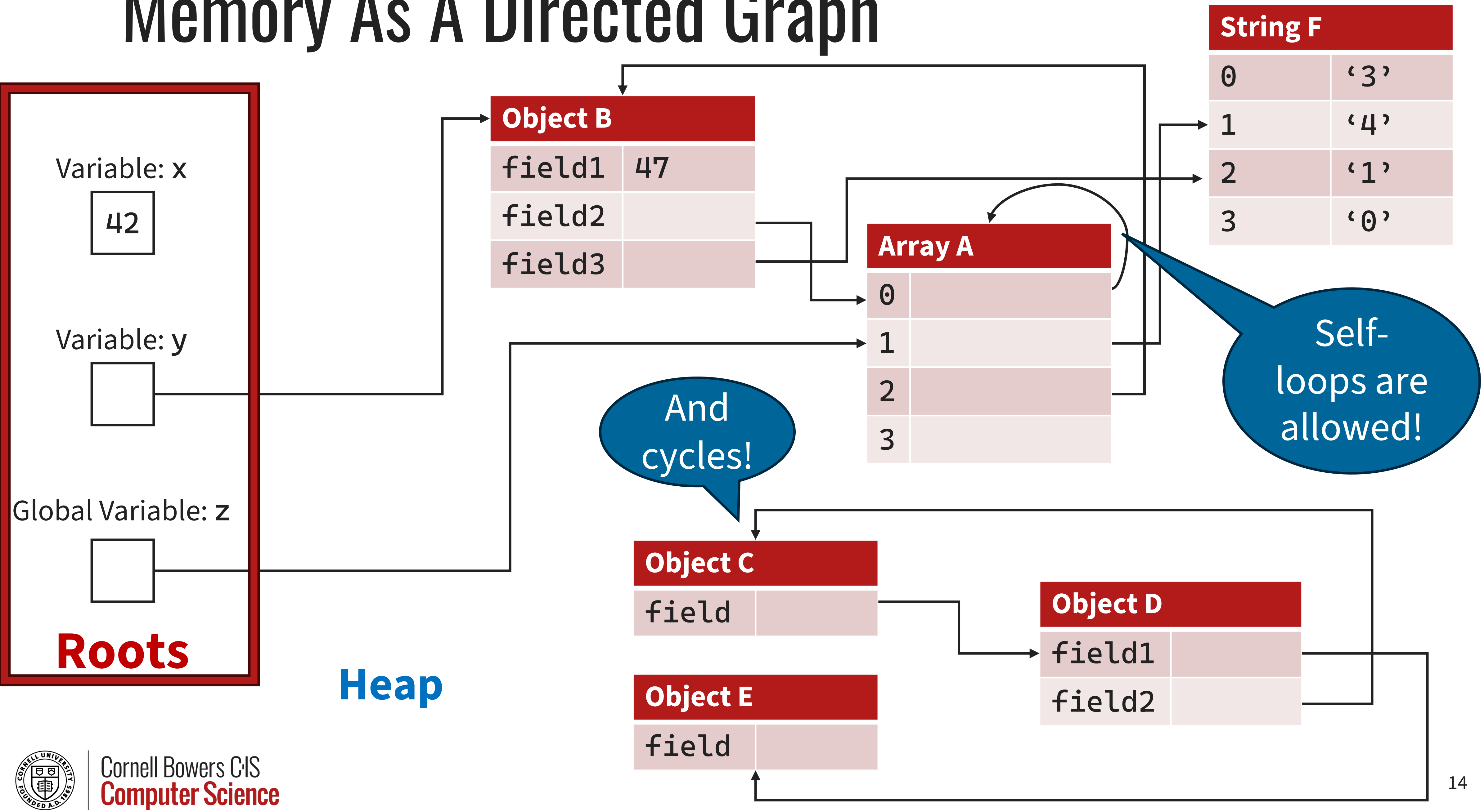
John McCarthy (1927-2011)

A **garbage collector** is a system that searches the heap for memory blocks that were allocated by the program, but are no longer used (i.e., **unreachable**)

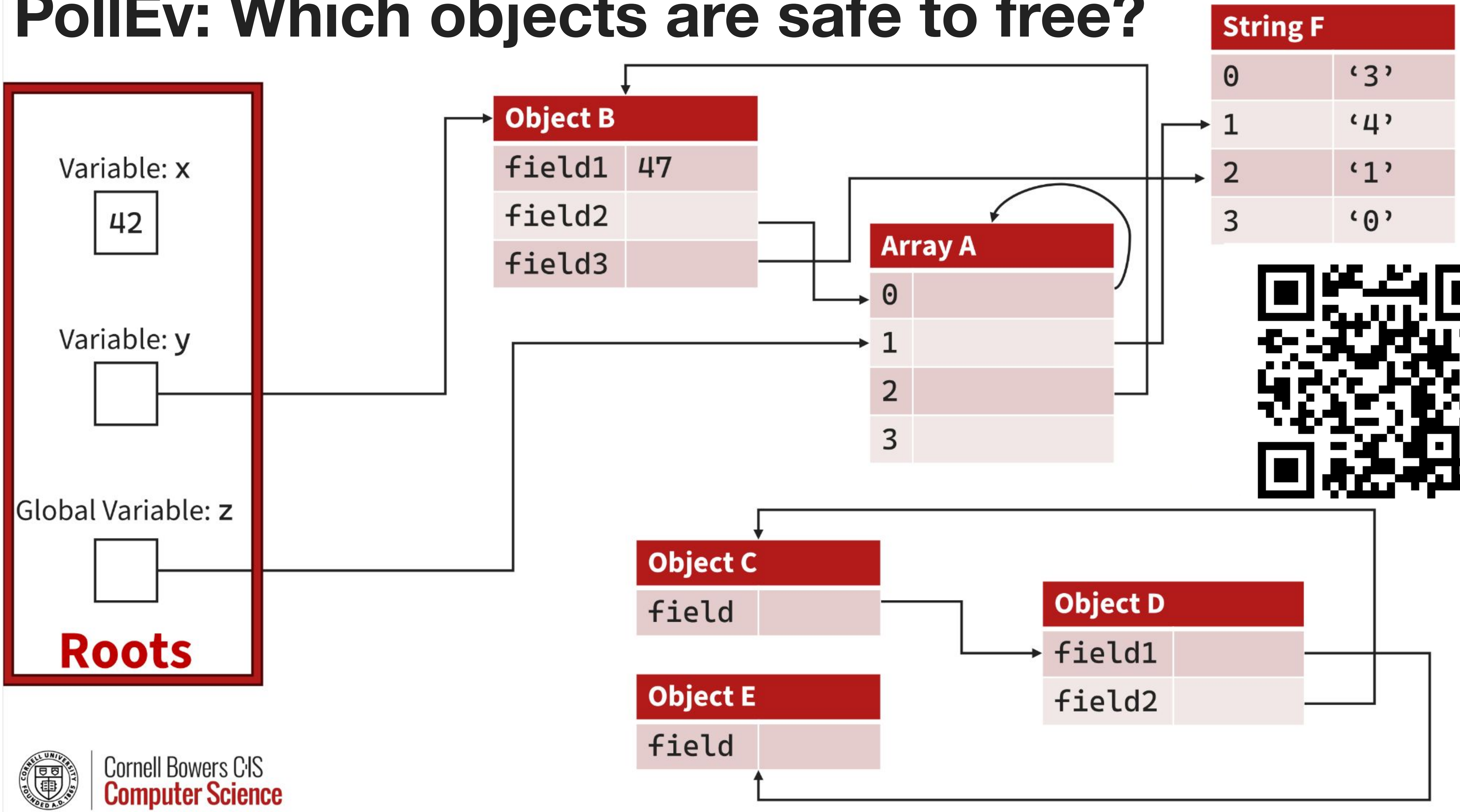
Key Idea: view memory as a ***directed graph***

- Nodes are blocks of memory
- Edges are pointers/references between blocks

Memory As A Directed Graph



PollEv: Which objects are safe to free?



Mark-and-Sweep or Tracing GCs

Most common type of garbage collector

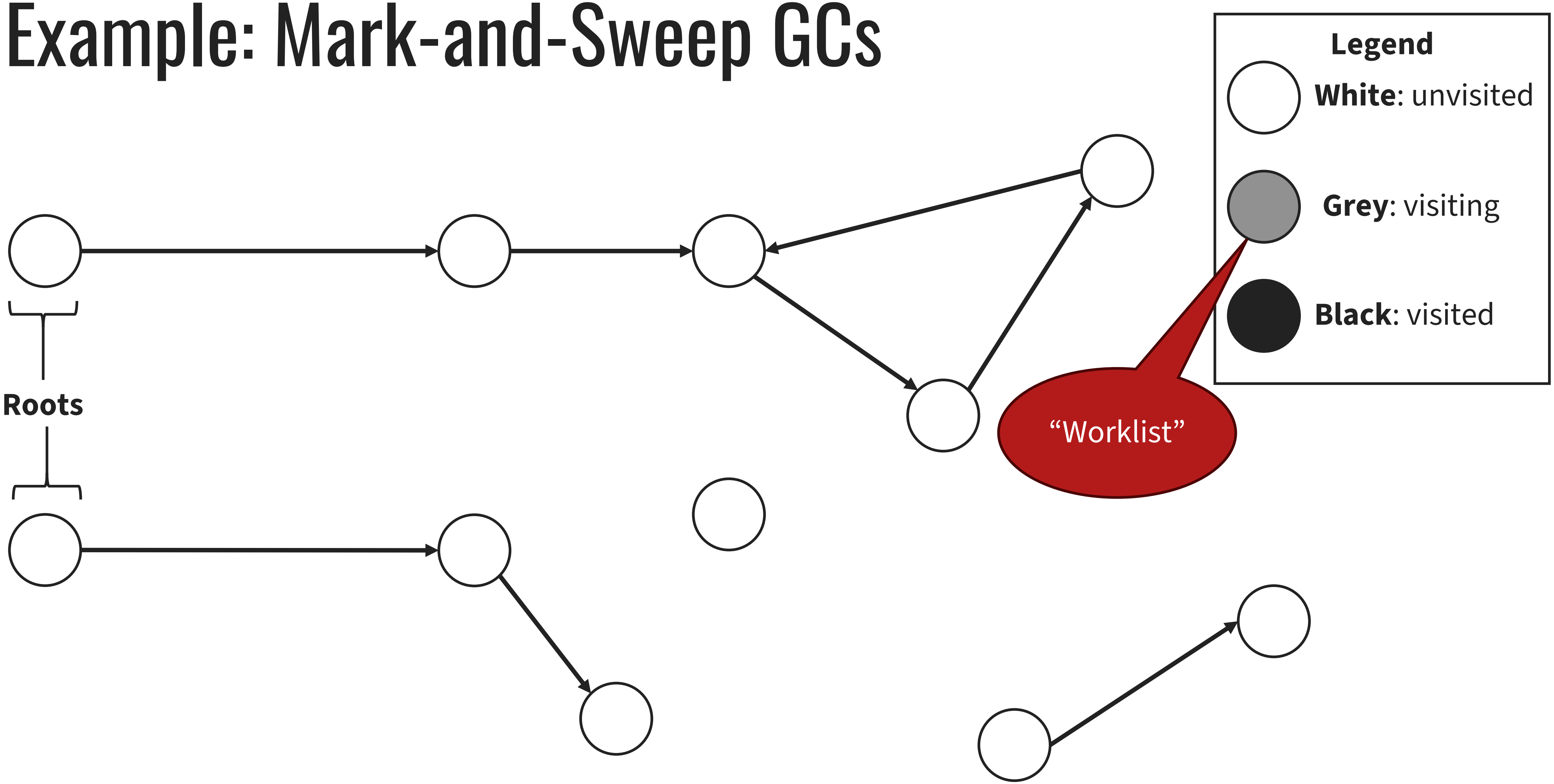
A memory block is **reachable** if it is either:

1. in the root set, or
2. referenced by a block of memory that is **reachable**

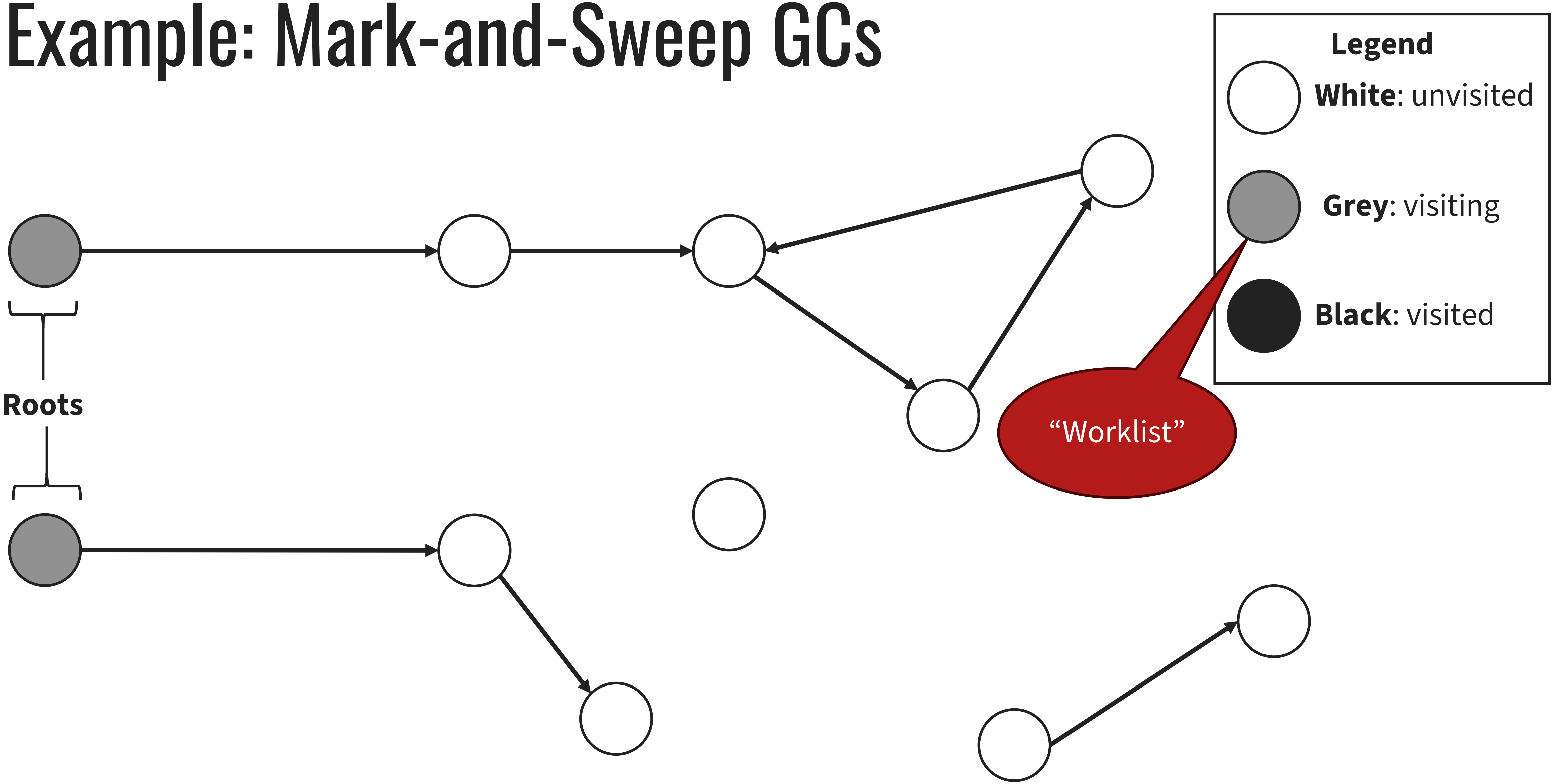
Two Phases:

1. **Mark**: traverse entire root set and **mark** each object that is **reachable**
2. **Sweep**: free all memory *not* marked as reachable

Example: Mark-and-Sweep GCs

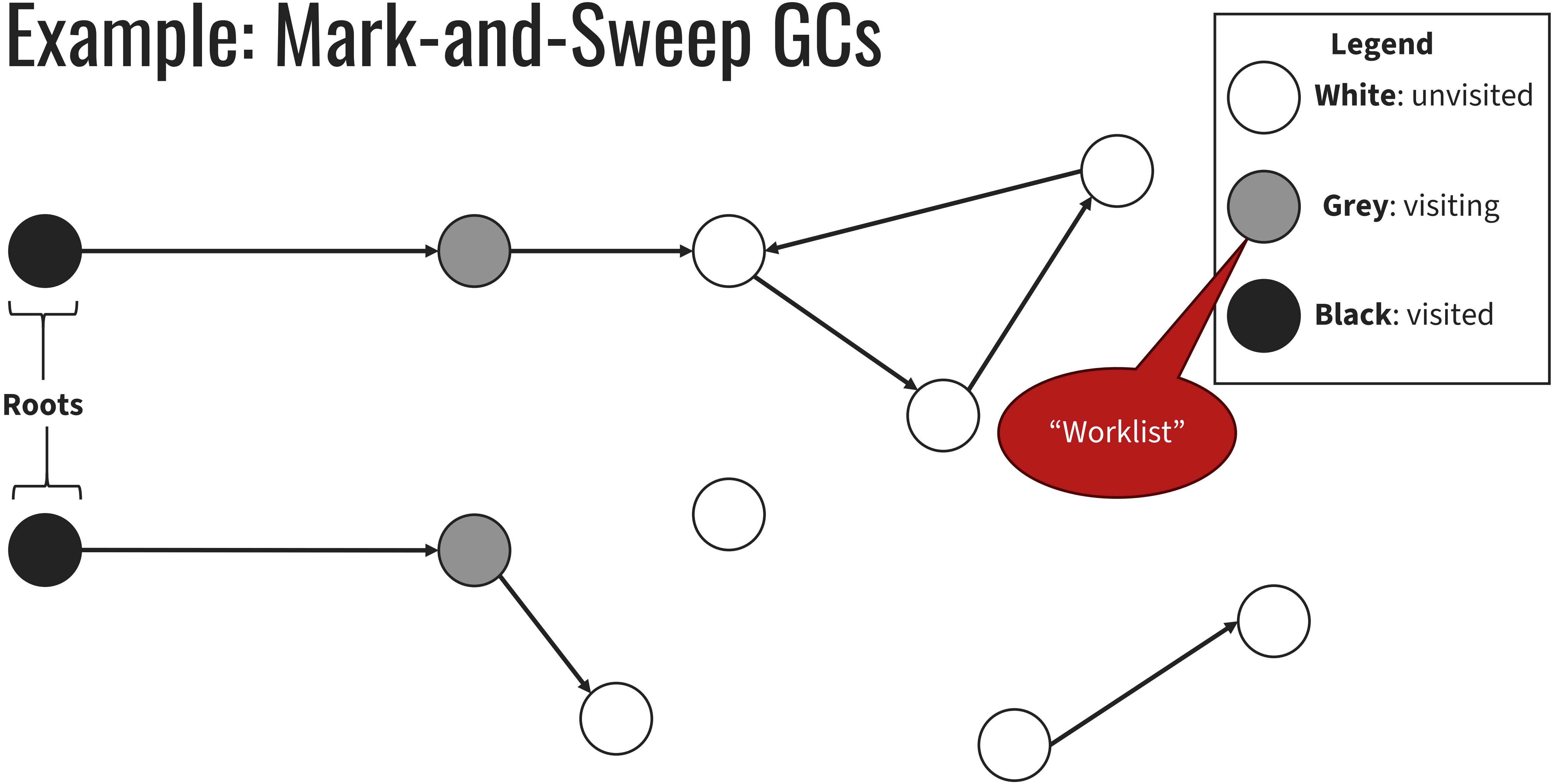


Example: Mark-and-Sweep GCs



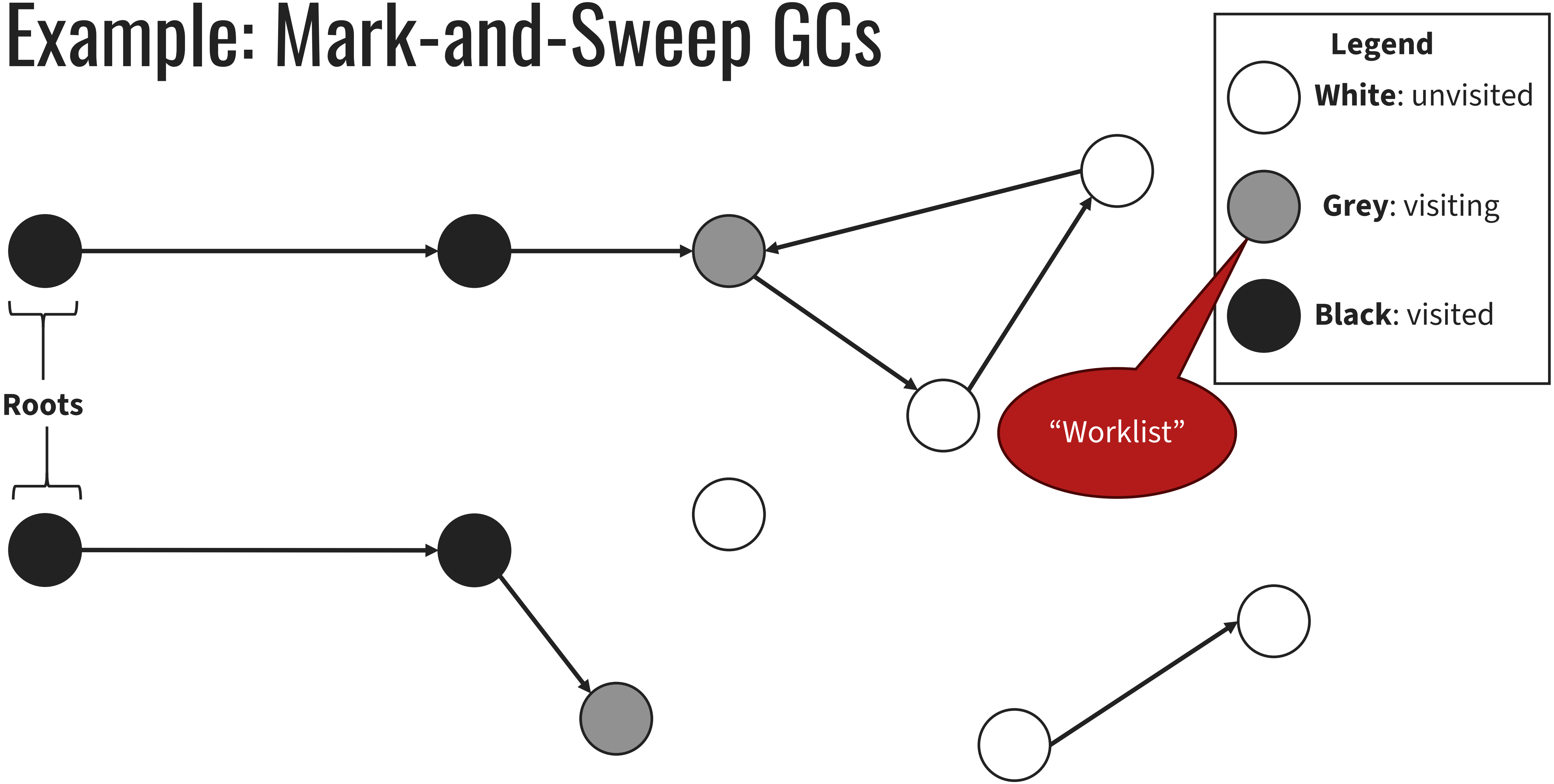
1. Color the root set as grey

Example: Mark-and-Sweep GCs



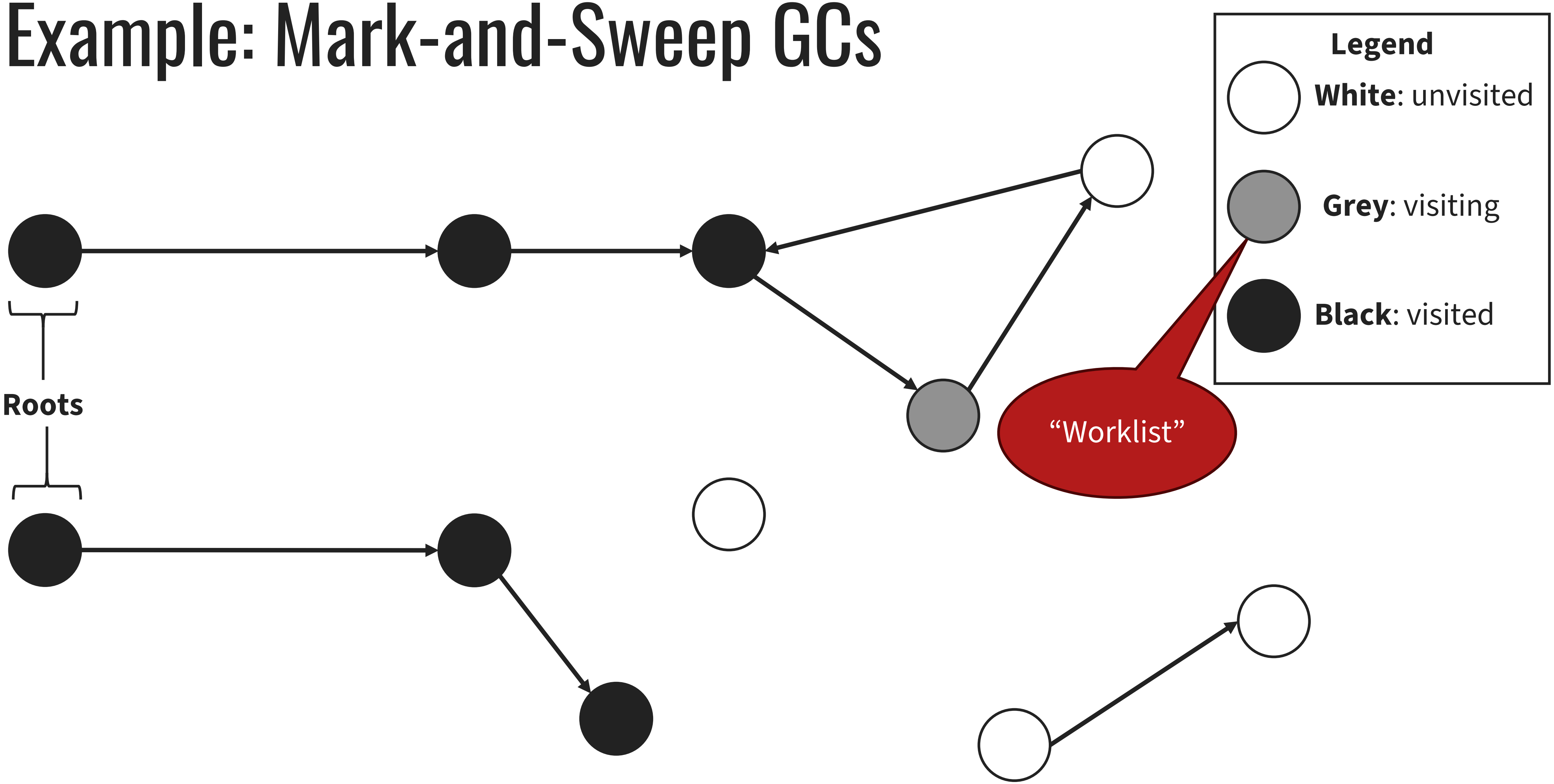
2. Explore references from grey nodes

Example: Mark-and-Sweep GCs



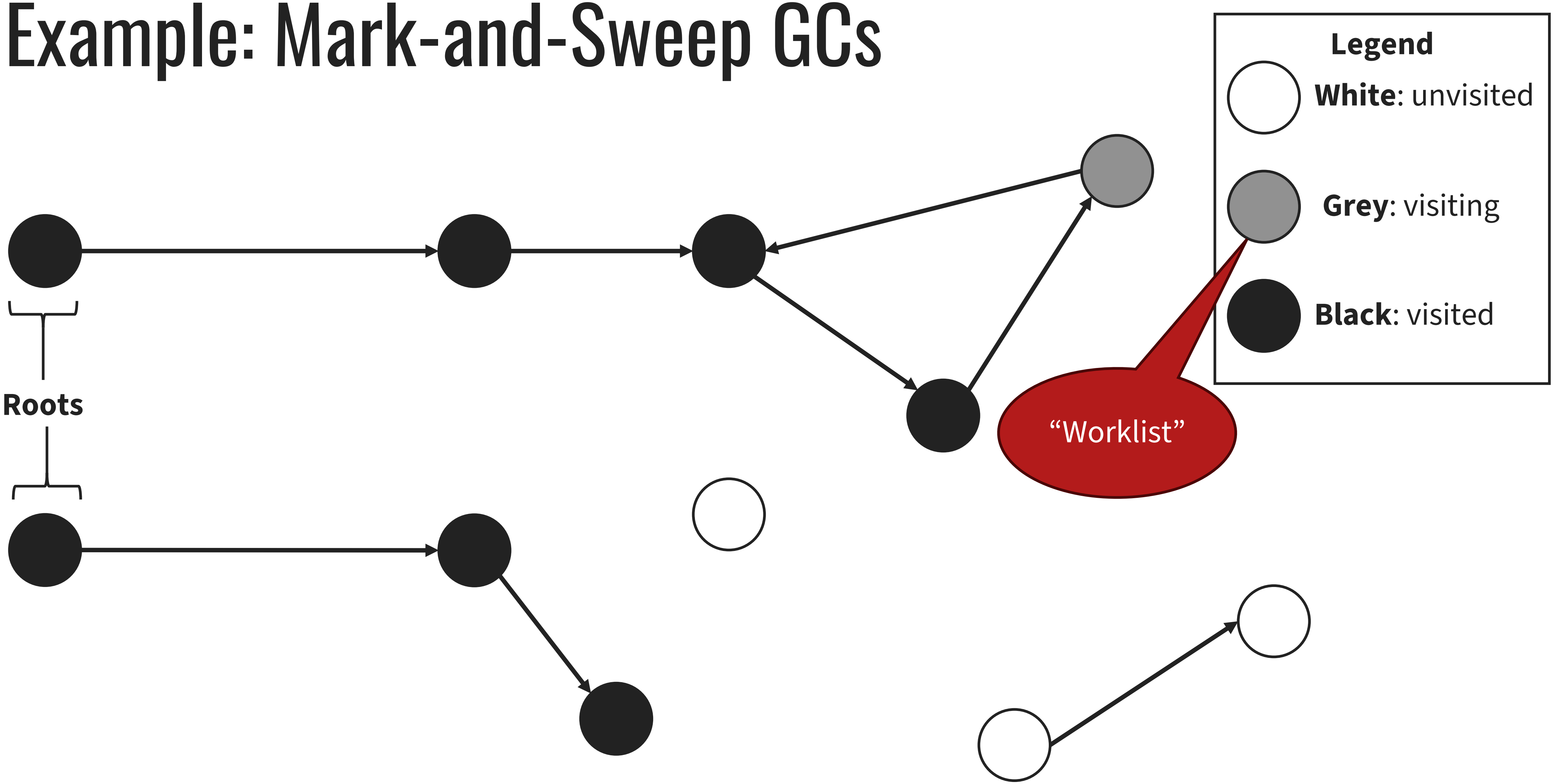
2. Explore references from grey nodes

Example: Mark-and-Sweep GCs



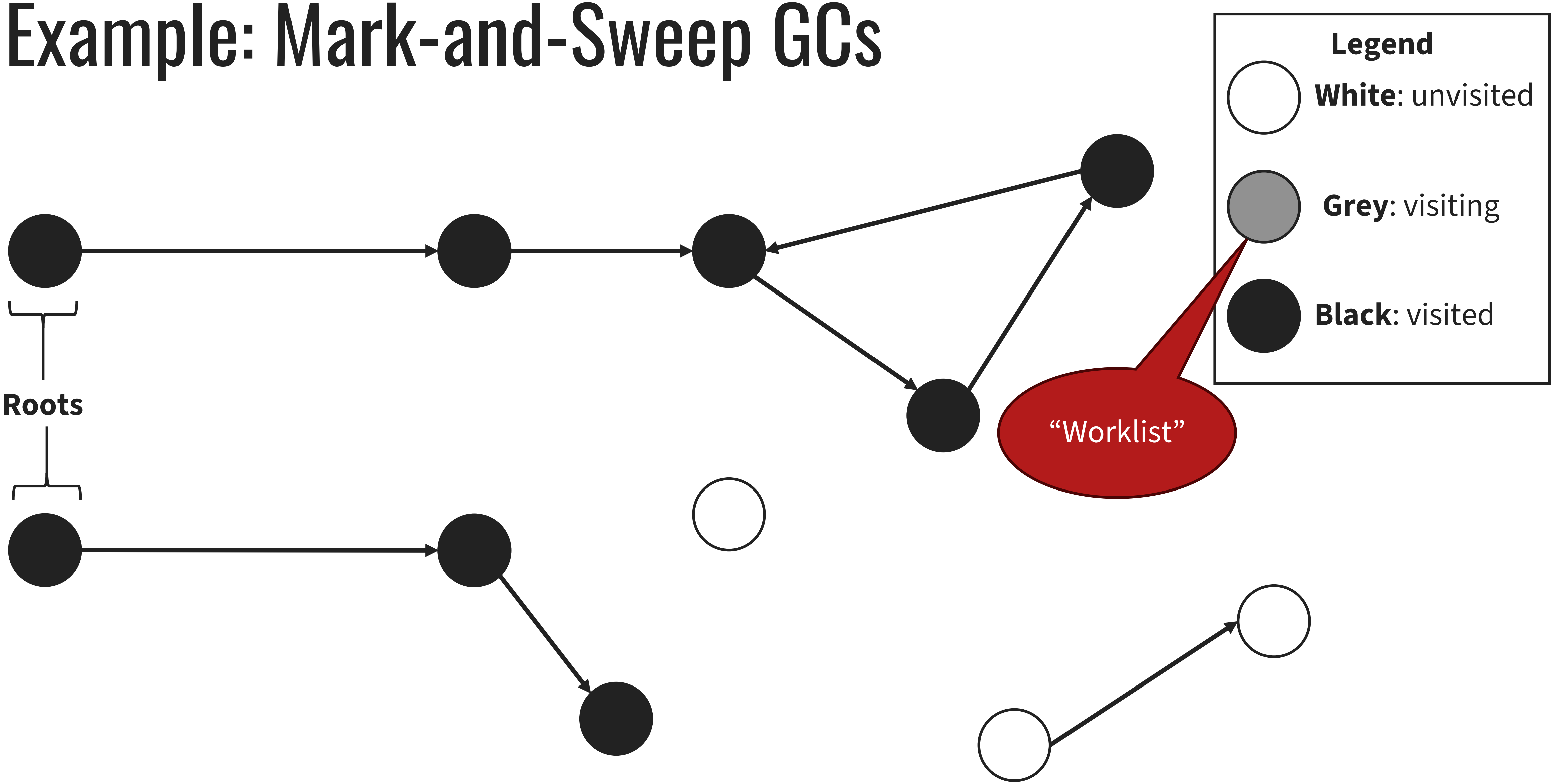
2. Explore references from grey nodes

Example: Mark-and-Sweep GCs



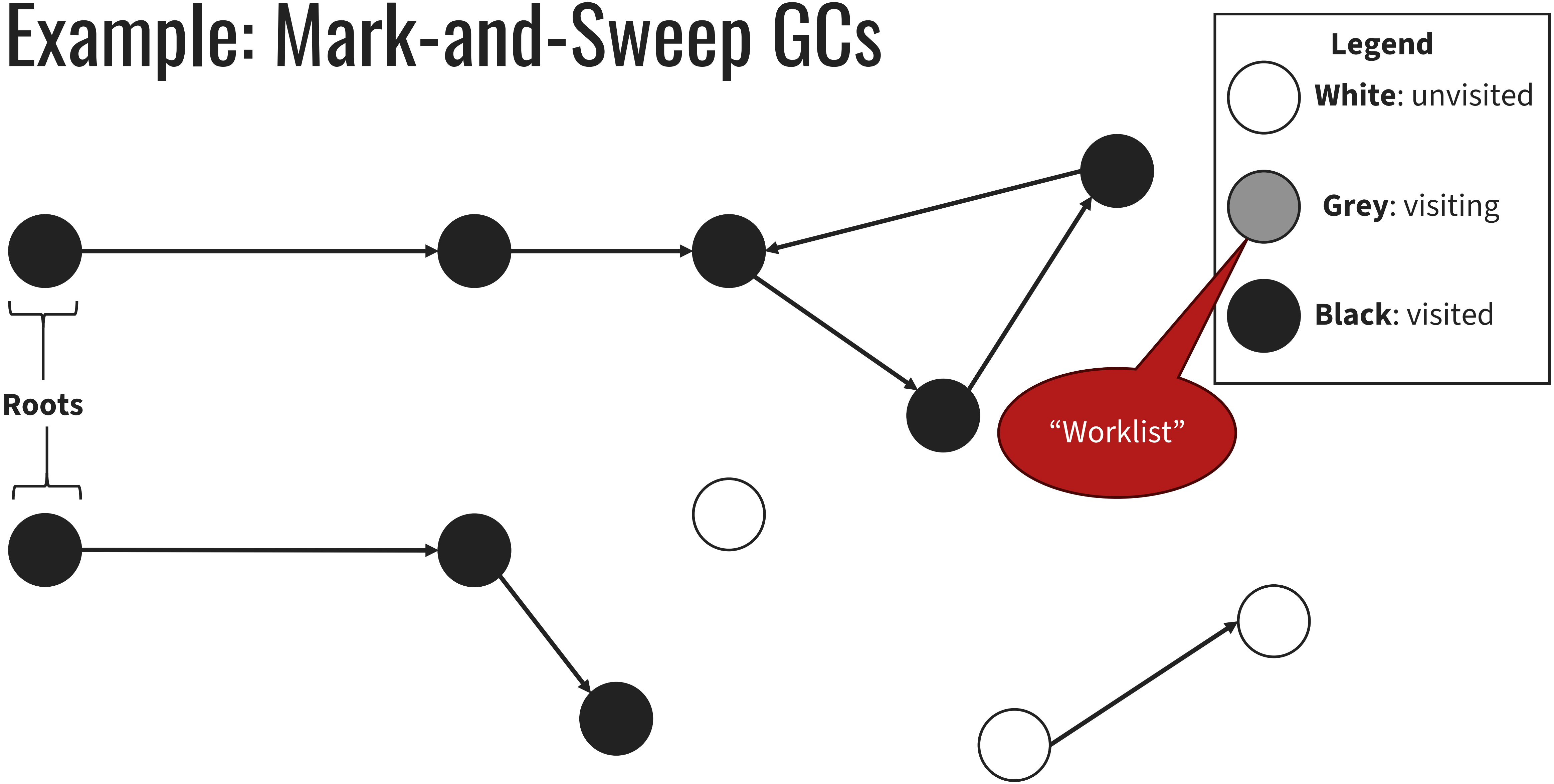
2. Explore references from grey nodes

Example: Mark-and-Sweep GCs



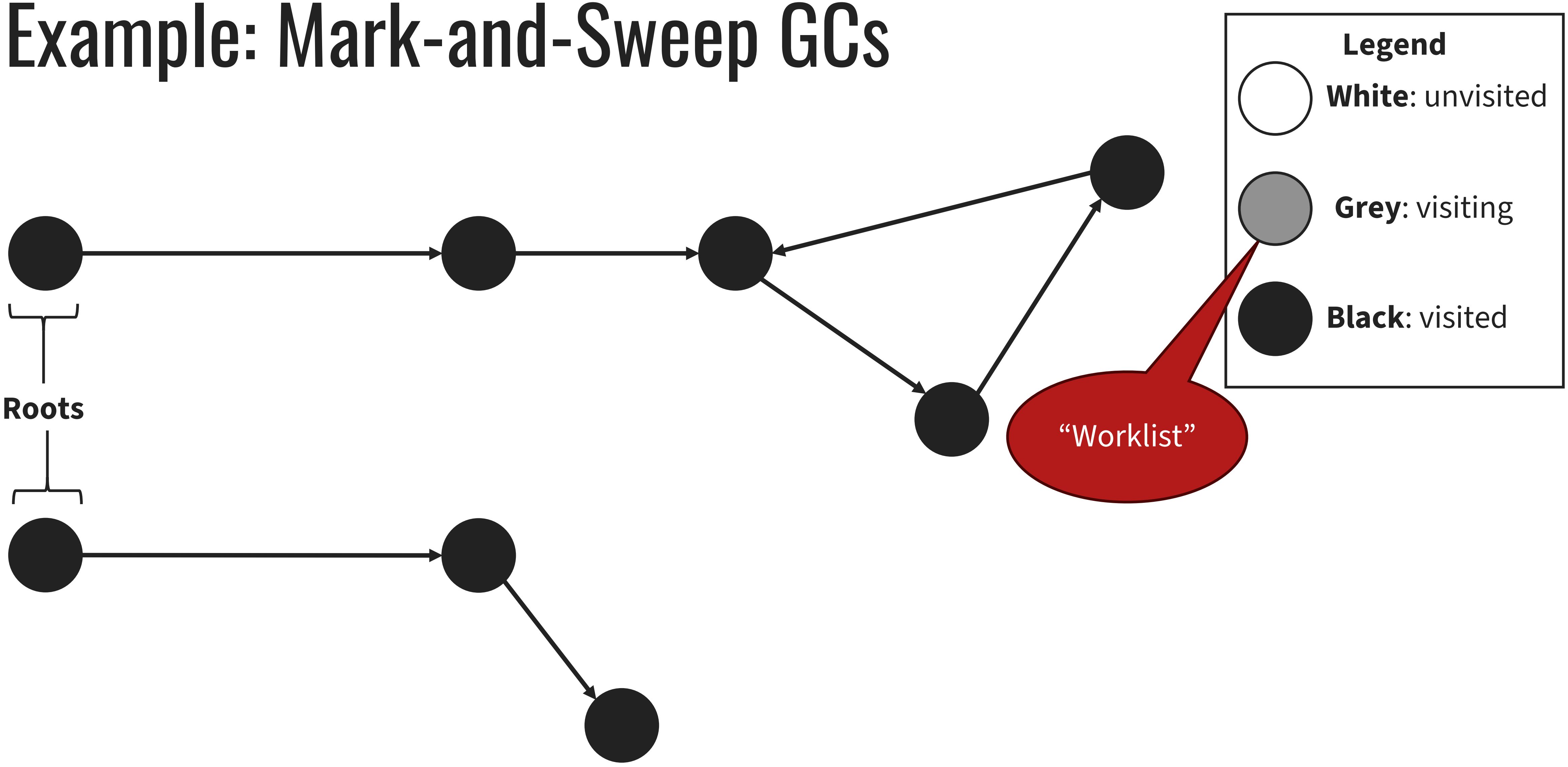
2. Explore references from grey nodes

Example: Mark-and-Sweep GCs



3. After exhausting all grey nodes, free white nodes

Example: Mark-and-Sweep GCs



4. Clear all marks for next GC run

When should the GC run?

Many, *many* options:

- When the system is low on memory
- Manually triggered
- Periodically on a schedule

Hard to predict when GC will run

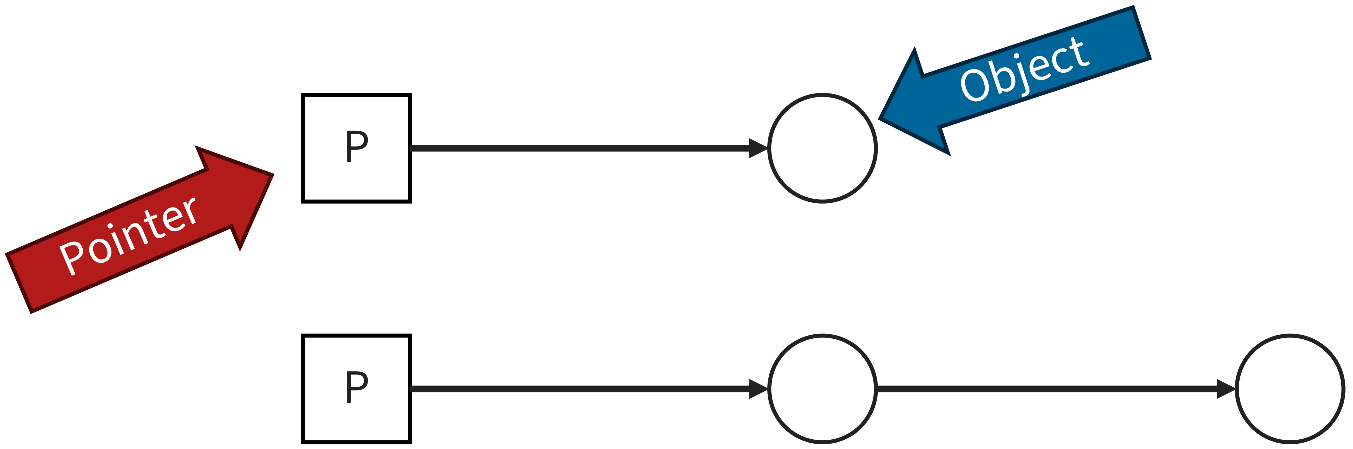
Program (generally) cannot modify memory while GC is running

- Program commonly needs to be paused (**stop-the-world**)

Reference Counting

Key Idea:

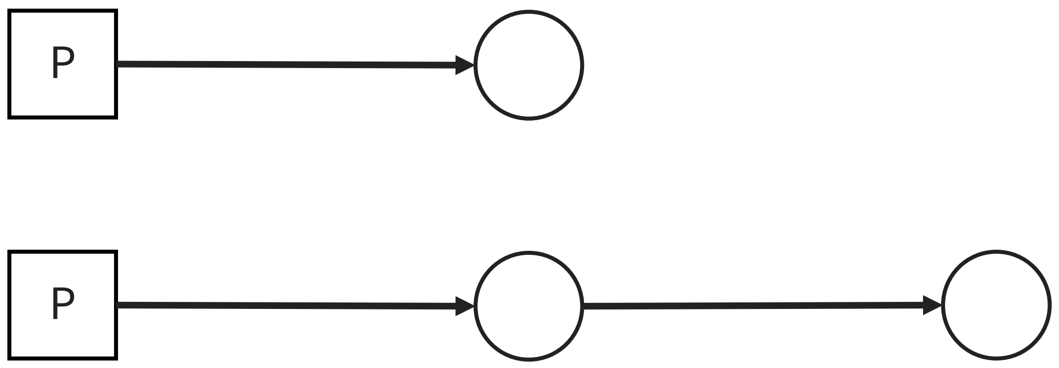
1. Keep track of how many pointers to each object exist
   - Increment when a new pointer is made
   - Decrement when a pointer is deleted
2. If reference count reaches 0, free object
3. Recursively reference count all objects that the freed object referenced



Reference Counting

Key Idea:

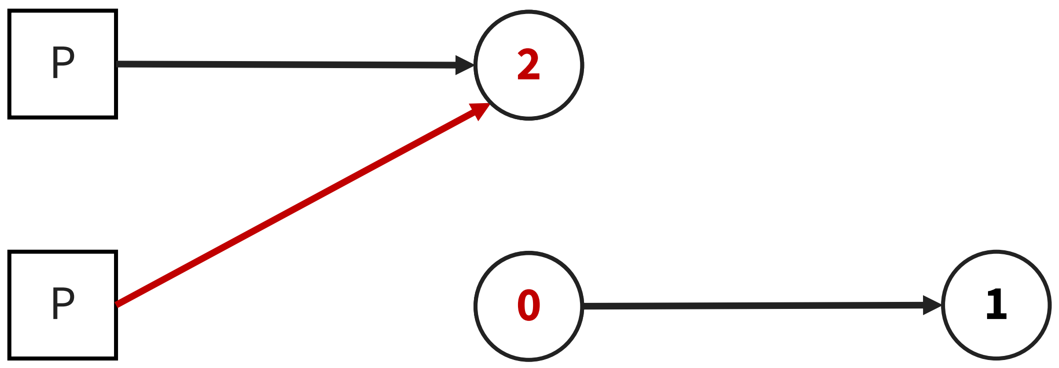
1. Keep track of how many pointers to each object exist
   - Increment when a new pointer is made
   - Decrement when a pointer is deleted
2. If reference count reaches 0, free object
3. Recursively reference count all objects that the freed object referenced



Reference Counting

Key Idea:

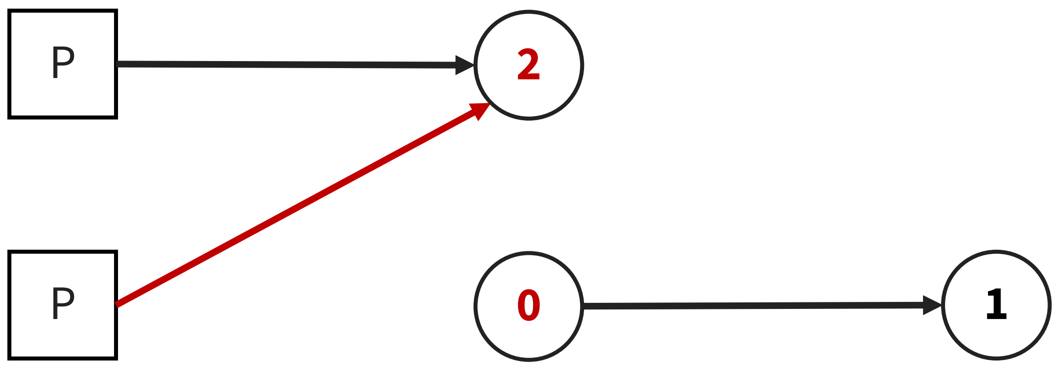
1. Keep track of how many pointers to each object exist
   - Increment when a new pointer is made
   - Decrement when a pointer is deleted
2. If reference count reaches 0, free object
3. Recursively reference count all objects that the freed object referenced



Reference Counting

Key Idea:

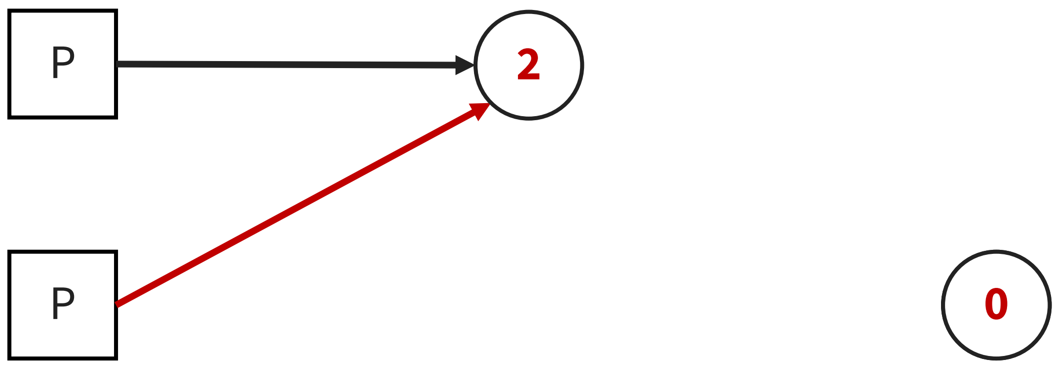
1. Keep track of how many pointers to each object exist
   - Increment when a new pointer is made
   - Decrement when a pointer is deleted
2. If reference count reaches 0, free object
3. Recursively reference count all objects that the freed object referenced



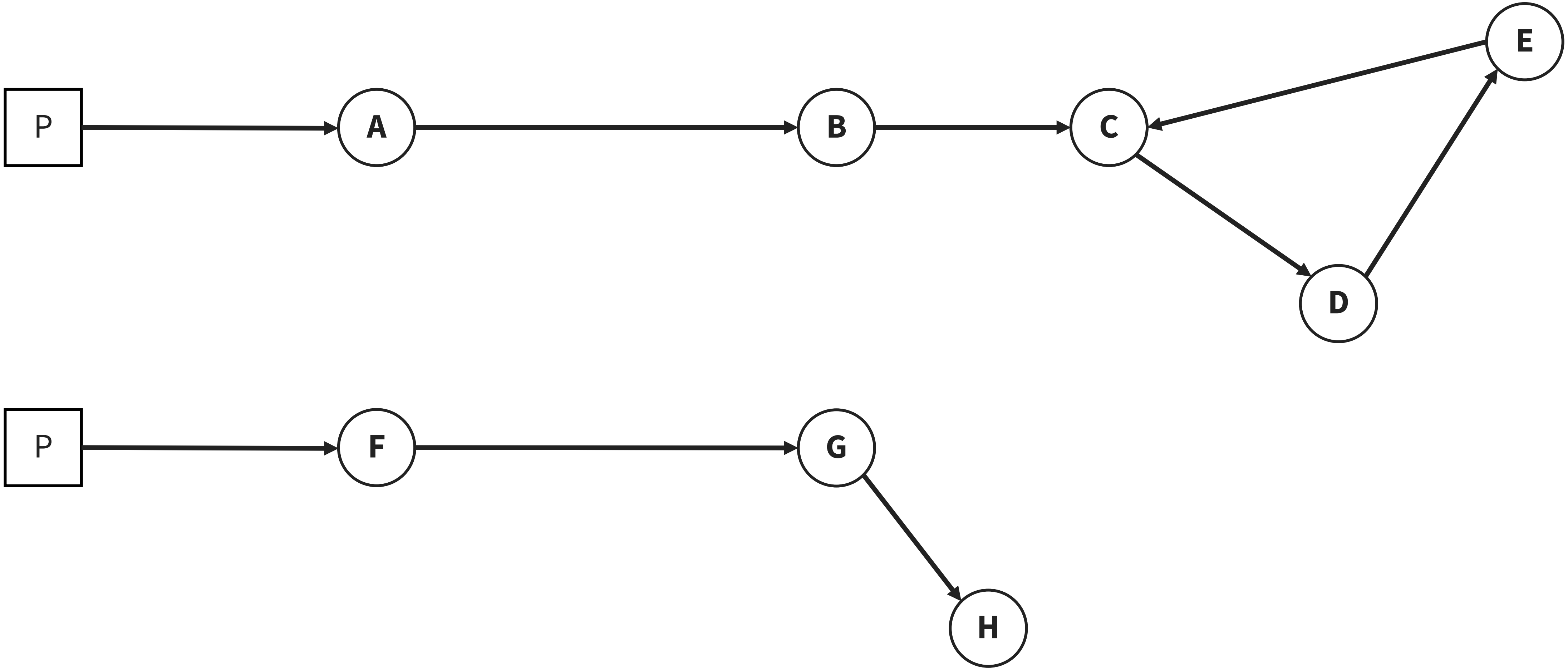
Reference Counting

Key Idea:

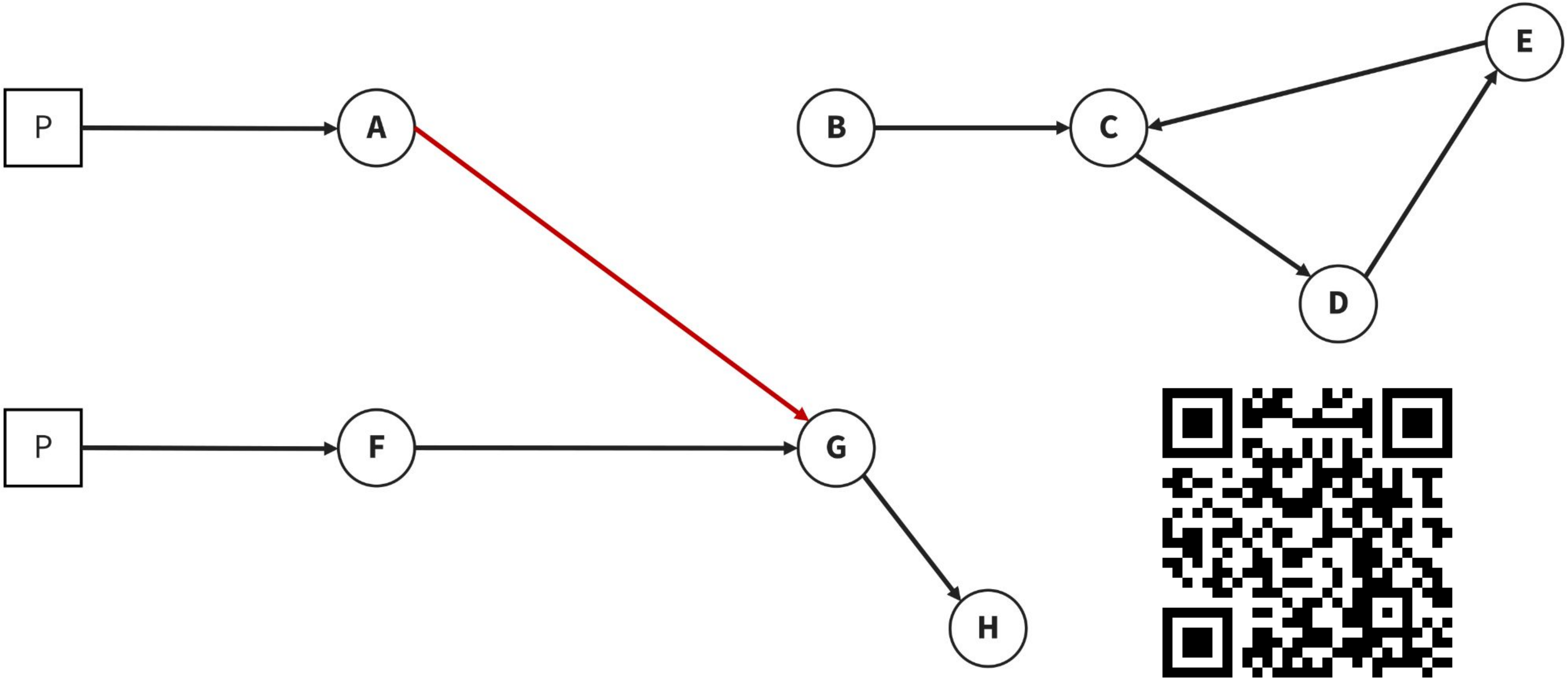
1. Keep track of how many pointers to each object exist
   - Increment when a new pointer is made
   - Decrement when a pointer is deleted
2. If reference count reaches 0, free object
3. Recursively reference count all objects that the freed object referenced



PollEv: Reference Counting

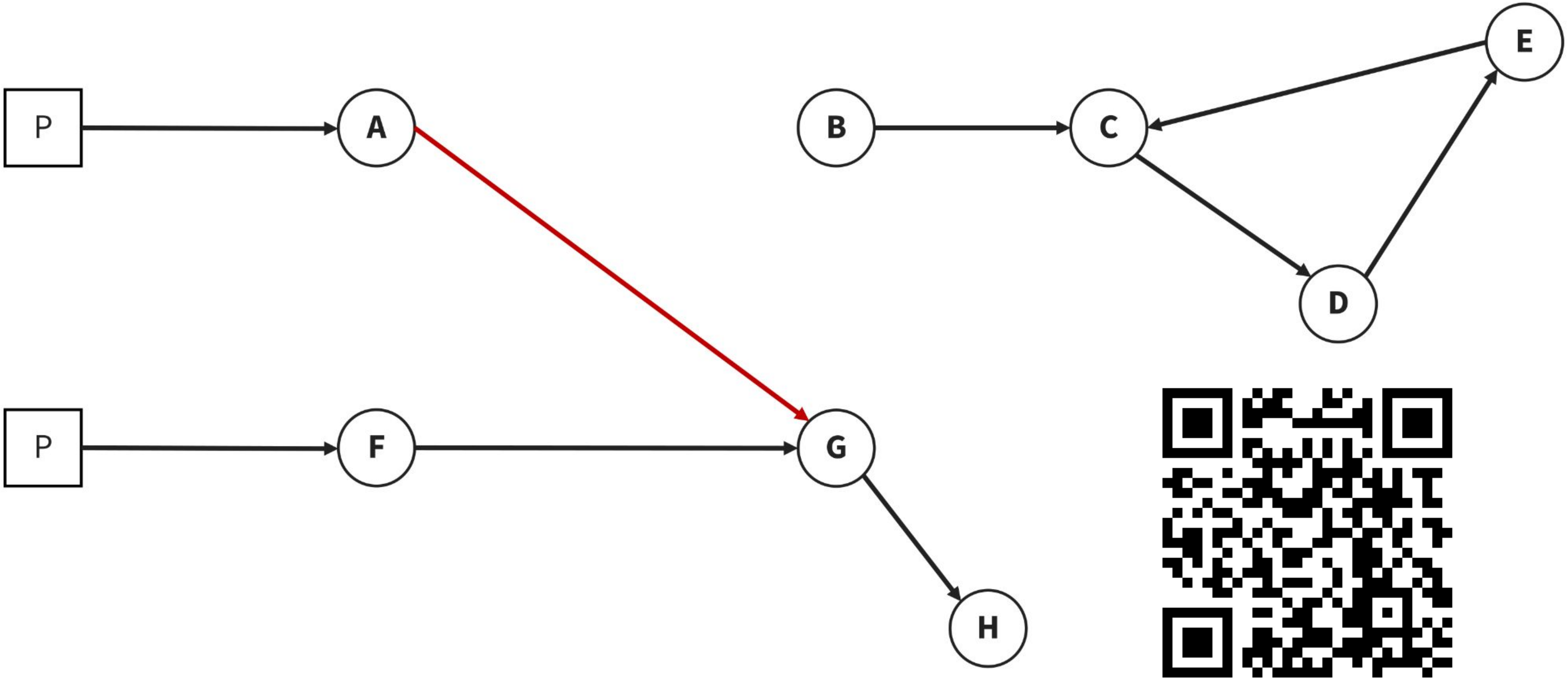


PollEV: Which objects are <u>safe</u> to free?



PollEv.com/cs3410

PollEV: Which objects will be freed by the reference counting?



PollEv.com/cs3410

Compare & Contrast

Garbage Collectors	Reference Counting

Compare & Contrast

Garbage Collectors	Reference Counting
Runs *periodically*	Update counts on *every* pointer update

Compare & Contrast

Garbage Collectors	Reference Counting
Runs *periodically*	Update counts on *every* pointer update
Perform BFS/DFS/etc. starting at root sets to find all *reachable* memory	Increment/decrement counter on the old/new pointed-to object

Compare & Contrast

Garbage Collectors	Reference Counting
Runs *periodically*	Update counts on *every* pointer update
Perform BFS/DFS/etc. starting at root sets to find all *reachable* memory	Increment/decrement counter on the old/new pointed-to object
Little to no metadata	Reference counts for each object

Compare & Contrast

Garbage Collectors	**Reference Counting**
Runs *periodically*	Update counts on *every* pointer update
Perform BFS/DFS/etc. starting at root sets to find all *reachable* memory	Increment/decrement counter on the old/new pointed-to object
Little to no metadata	Reference counts for each object
Handles cycles	**Struggles with cycles**

Why is C Still Used?

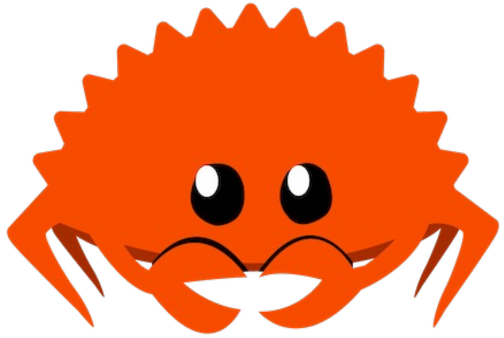
	Energy
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

Energy Efficiency across Programming Language
R Pereira, M Couto, F Ribeiro, R Rua, J Cunha, JP Fernandes, J Saraiva. ACM SIGPLAN International Conference on Software Language Engineering (SLA). Pages 256–267. October 2017.
https://doi.org/10.1145/3136014.3136031

Rust

Rust

- A **strongly typed**, **compiled**, **memory safe**, **systems-oriented** programming language
  - High-level ergonomics from functional programming
  - Low-level control from C/C++
  - **Safe + Fast!**
- Designed by Graydon Hoare (Mozilla); began as personal project
- **Wildly popular**: “Most desired PL” in Stack Overflow’s annual developer survey for the past **9 years** (2024: 83%)
- **Rapid adoption**: first language other than C and assembly to be supported in development of Linux kernel



Ferris

How does memory management work in Rust?

- **Compiler** knows where to insert de-allocation calls
  - Perfect memory management without GC!
- But… programmer needs to follow certain *ownership* rules
  - Rules are checked by the compiler
- Consequence: ***undefined behavior*** is caught at compile-time instead of runtime!
  - Improved reliability and stability
  - Improved **performance**

Variables Live in the Stack

```rust
fn increment(x: i32) -> i32 {
  x + 1
}


fn main() {
  let n = 5;
  let y = increment(n);
  println!("The value of y is: {y}");
}
```

Boxes Live in the Heap

```
let a = [0; 10_000_000];
let b = a;
```

Array a is *copied* into b

```
let a = Box::new([0; 10_000_000]);
let b = a;
```

a and b are pointers to the array

Array lives on the heap

The Owner Manages Deallocation

```
fn make_and_drop() {
  let a_box = Box::new(5);
}

fn main() {
  let a_num = 4;
  make_and_drop();
}
```

5 is placed on heap

`a_box` is the owner of the `Box`

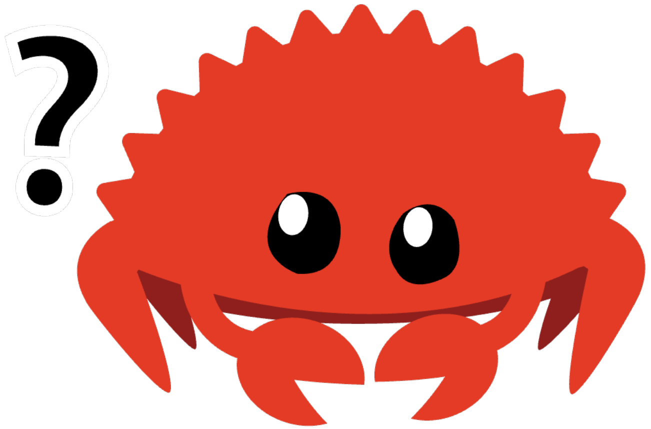
Key Idea:
When a variable goes out of scope, the heap data it **owns** is deallocated (dropped)

Moving

```rust
fn greet(mut name: String) -> String {
    name.insert_str(0, "Hi, ");
    name.push_str("!");
    name
}

fn main() {
    let name = String::from("Zach");
    let greeting = greet(name);
    println!("{}", greeting);
}
```

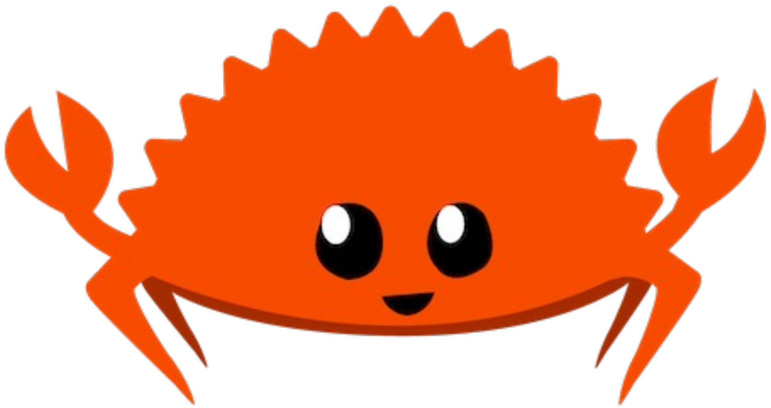
Moving

```rust
fn greet(mut name: String) -> String {
    name.insert_str(0, "Hi, ");
    name.push_str("!");
    name
}

fn main() {
    let name = String::from("Zach");
    let greeting = greet(name);
    println!("{}", greeting);
    println!("Bye, {name}!");
}
```



Cloning

```rust
fn greet(mut name: String) -> String {
  name.insert_str(0, "Hi, ");
  name.push_str("!");
  name
}

fn main() {
  let name = String::from("Zach");
  let name_clone = name.clone();
  let greeting = greet(name_clone);
  println!("{}", greeting);
  println!("Bye, {name}!");
}
```



Ownership

A discipline of heap management:

1. All data on the heap must be owned by **exactly one** variable
   - Variable is referred to as **owner**
2. Rust deallocates heap data once its owner goes out of scope
3. Ownership can be transferred by **moves** (e.g., assignment, function calls)

Moving is Cumbersome; Cloning seems Inefficient

```rust
fn greet(name: String) {
    println!("Hi, {name}!");
}

fn main() {
    let name = String::from("Zach");
    let name_clone = name.clone();
    greet(name_clone);
    println!("Bye, {name}!");
}
```

Do we absolutely **need** to clone here?

References

Ampersand!

```
fn greet(name: &String) {
    println!("Hi, {name}!");
}

fn main() {
    let name = String::from("Zach");
    greet(&name);
    println!("Bye, {name}!");
}
```

Ampersand!

References are non-owning pointers

References are like pointers!

```rust
let mut x: Box<i32> = Box::new(1);
let a: i32 = *x;
*x += 1;

let r1: &Box<i32> = &x;
let b: i32 = **r1;

let r2: &i32 = &*x;
let c: i32 = *r2;
```

Aliases & Mutation

- Rust distinguishes between **immutable** and **mutable** variables and references

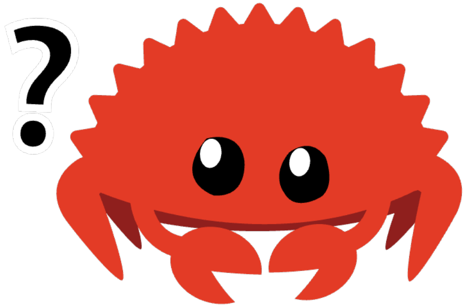
Mutable

```
let mut v: Vec<i32> = vec![1, 2, 3];
let num: &i32 = &v[2];
v.push(42);
println!("Third element is {}", *num);
```

Alias (read-only)

Need to resize & relocate vector

Undefined behavior!



Takeaway

Data should never be aliased and mutated **at the same time!**

In any scope, there can be either:

1. *any number* of **immutable** references, or
2. *at most one* **mutable** reference

referring to the same variable

Checked by the **borrow checker**!

References are Pointers with Permissions

In Rust, all variables can be seen as having **read**, **write**, and/or **own** permissions

- All variables are **read-only** by default
- Add **write** permission by adding `mut` keyword

- Immutable references have just **read** permission
- Mutable references have **read** and **write** permissions
  - However, **write** permission is on loan from the variable

Borrow checker finds permission violations

Hungry for more Rust?

- Read **The Rust Programming Language** online textbook!
- Rust by Example
- Website: https://www.rust-lang.org/

Recap

- **Motivation**: What are memory safe languages and why do we need them?
- **Dynamic** automated memory management
  - Garbage Collectors (GCs)
  - Reference Counting
- **Static** automated memory management
  - Rust!
  - **Key Feature:** Ownership
    - References