Java is a trademark of Sun Microsystems, Inc.

Java

JavaOne℠

The Ghost in the Virtual Machine
A Reference to References

Bob Lee

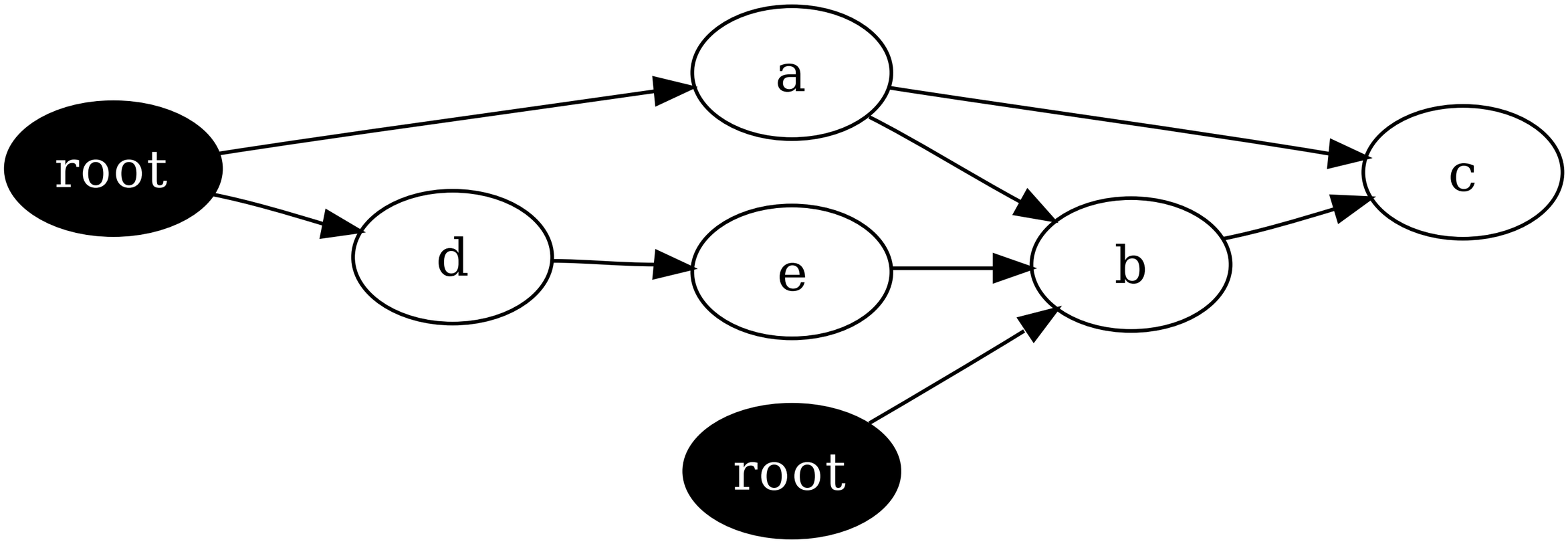
Google Inc.

Google

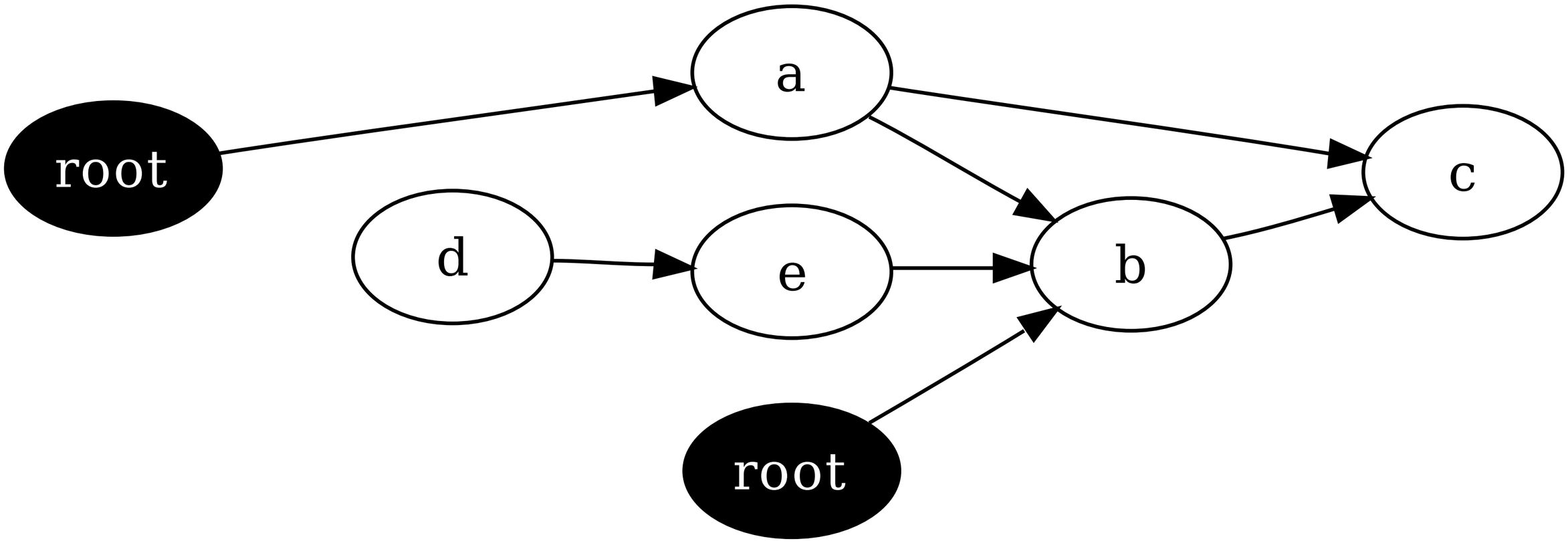
Goals

- Take the mystery out of garbage collection.
- Perform manual cleanup the Right way.
- Become honorary VM sanitation engineers.

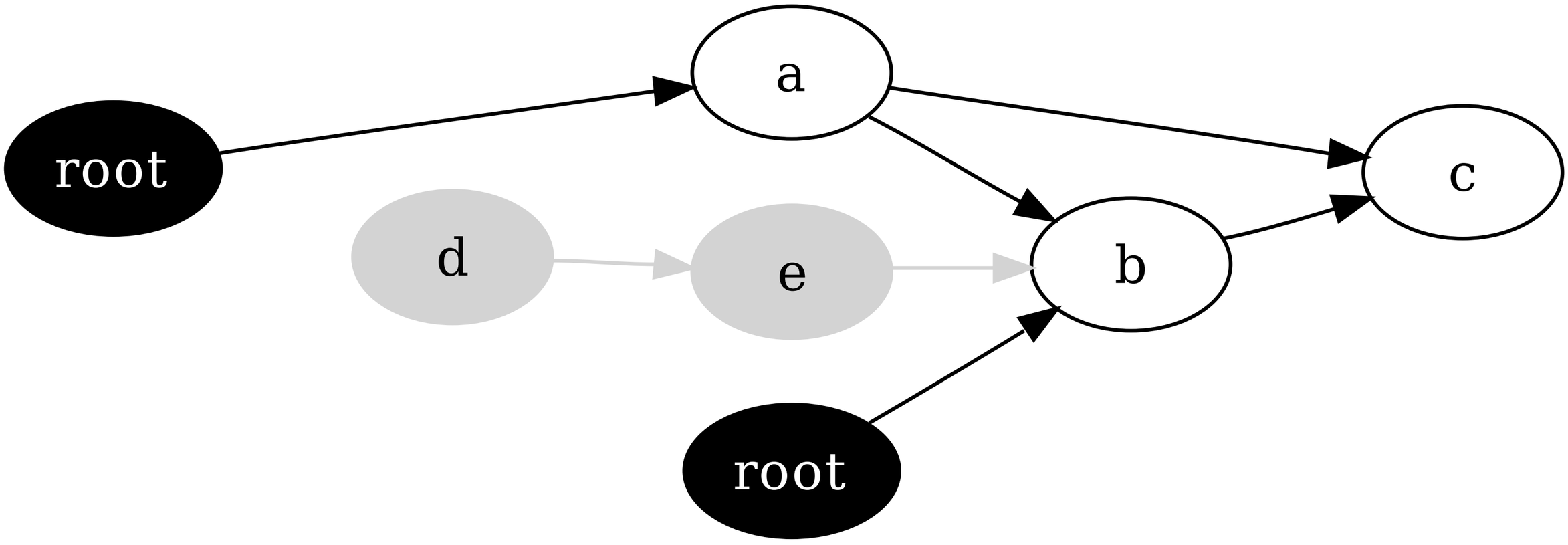
How does garbage collection work?



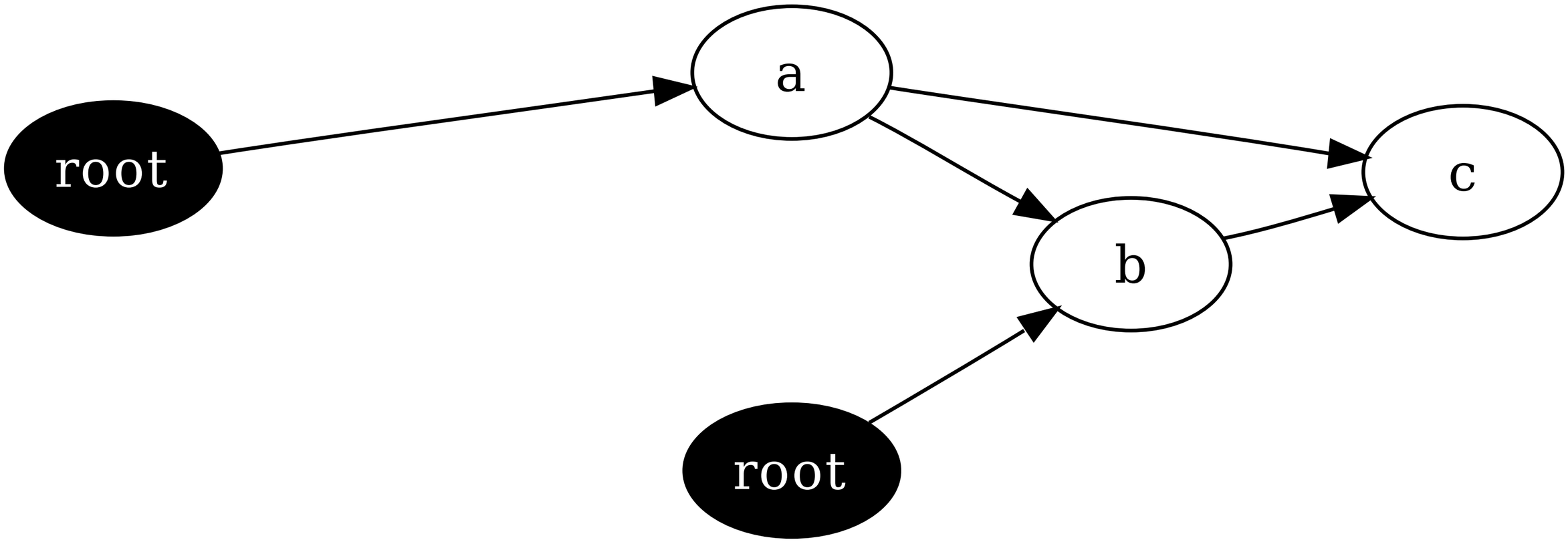
If the reference to D goes away...



We can no longer reach D or E.



So the collector reclaims them.



The GC can't do everything.

- Some things require manual cleanup.
  - Listeners
  - File descriptors
  - Native memory
  - External state (`IdentityHashMap`)
- Tools at your disposal:
  - `finally`
  - Overriding `Object.finalize()`
  - Reference queues

Try `finally` first.

> Reasons to not use `finally`:

- More work for programmers
- More error prone
- Cleanup happens in main thread

> ARM will help.

What is a finalizer?

```
public class Foo extends Bar {
  @Override protected void finalize() throws Throwable {
    try {
      ... // Clean up Foo.
    } finally {
      super.finalize(); // Clean up Bar.
    }
  }
}
```

Finalizers are seductively simple, but...

- They're not guaranteed to run, especially not timely.
- Avoid `System.runFinalizersOnExit()` and `runFinalization()`.
- Undefined threading model, can run concurrently!
- You must call `super.finalize()`.
- Exceptions are ignored (per spec).
- You can resurrect references.
- Keeps objects alive longer.
- Can make allocation/reclamation 430X slower (Bloch, Effective Java)

An external resource

```
public class NativeMemory {
  final int address = allocate();
  /** Allocates native memory. */
  static native int allocate();

  /** Writes to native memory. */
  public void write(byte[] data) {
    write(address, data);
  }
  static native void write(int address, byte[] data);

  /** Frees native memory. */
  @Override protected void finalize() {
    free(address);
  }
  static native void free(int address);
}
```

Let's play War!

`SegfaultFactory` can cause a segfault if its finalizer executes after `NativeResource`'s.

```
public class SegfaultFactory {
  private final NativeMemory nm;

  public SegfaultFactory(NativeMemory nm) {
    this.nm = nm;
  }

  @Override protected void finalize() {
    // 50/50 chance of failure
    nm.write("I'm taking the VM with me!".getBytes());
  }
}
```

Use protection.

```
public class NativeMemory {
  final int address = allocate();
  /** Allocates native memory. */
  static native int allocate();

  /** Writes to native memory. */
  boolean finalized;
  public synchronized void write(byte[] data) {
    if (!finalized) write(address, data);
    else /* do nothing? */;
  }
  static native void write(int address, byte[] data);

  /** Frees native memory. */
  @Override protected synchronized void finalize() {
    finalized = true;
    free(address);
  }
  static native void free(int address);
}
```

Finalizers are good for one thing.

Logging warnings

The alternative

An API-based approach:

```
public class WeakReference<T> {
  public WeakReference(T referent) {
     ...
  }
  public WeakReference(T referent,
                       ReferenceQueue<? super T> q) {
     ...
  }
  public T get() {
     ...
  }
  ...
}
```

What is a finalizer?

```
public class Button {
  private final List<WeakReference<Listener>> listeners
      = new ArrayList<WeakReference<Listener>>();
  public void add(Listener l) {
    listeners.add(new WeakReference<Listener>(l));
  }
  public void click() {
    Iterator<WeakReference<Listener>> i
        = listeners.iterator();
    while (i.hasNext()) {
      Listener l = i.next().get();
      if (l == null) i.remove();
      else l.onClick();
    }
  }
  public interface Listener {
    void onClick();
  }
}
```

The alternative

```
public class ReferenceQueue<T> {
  public T poll() {
    ...
  }
  public T remove() {
    ...
  }
  public T remove(long timeout) {
    ...
  }
}
```

Reachability

- An object is *reachable* if a live thread can access it.
- Examples of heap roots:
  - System classes (which have static fields)
  - Thread stacks
  - In-flight exceptions
  - JNI global references
  - The finalizer queue
  - The interned String pool
  - etc. (VM-dependent)

Making maps

```
public class BytecodeCache {
  final static Map<Class<?>, byte[]> cache = new MapMaker()
      .weakKeys()
      .softValues()
      .makeComputingMap(new Function<Class<?>, byte[]>() {
        public byte[] apply(Class<?> clazz) {
          ...
        }
      });

  public static byte[] bytesFor(Class<?> clazz) {
    return cache.get(clazz);
  }
}
```

Dante's Heap - The Levels of Reachability

- Strong
- Soft
- Weak
- Finalizer
- Phantom, JNI weak
- Unreachable

Dante's Heap - The Levels of Reachability

- **Strong**
- Soft
- Weak
- Finalizer
- Phantom, JNI weak
- Unreachable

Dante's Heap - The Levels of Reachability

- Strong
- **Soft**
- Weak
- Finalizer
- Phantom, JNI weak
- Unreachable

Dante's Heap - The Levels of Reachability

- Strong
- Soft
- **Weak**
- Finalizer
- Phantom, JNI weak
- Unreachable

Dante's Heap - The Levels of Reachability

- Strong
- Soft
- Weak
- **Finalizer**
- Phantom, JNI weak
- Unreachable

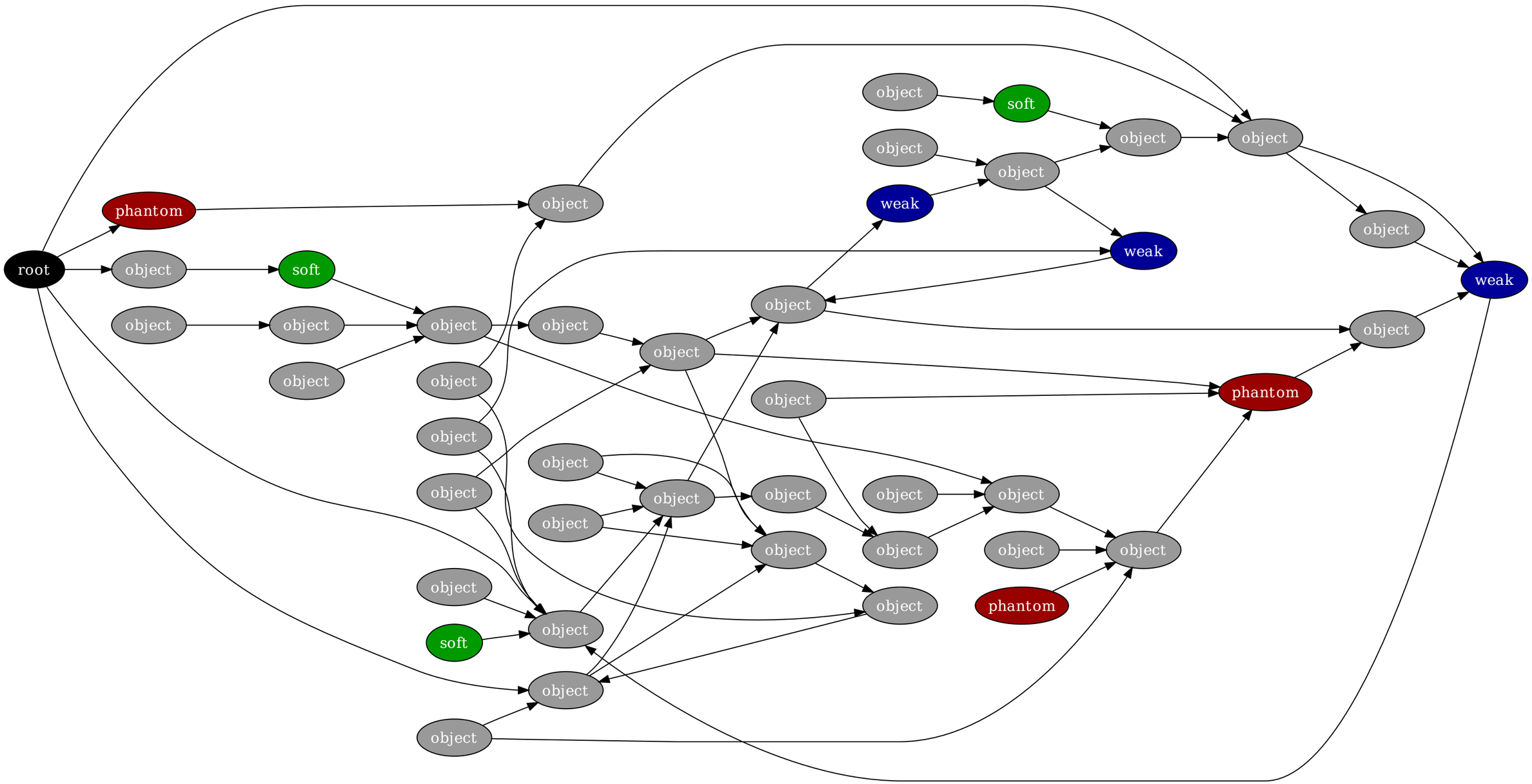
Dante's Heap - The Levels of Reachability

- Strong
- Soft
- Weak
- Finalizer
- **Phantom, JNI weak**
- Unreachable

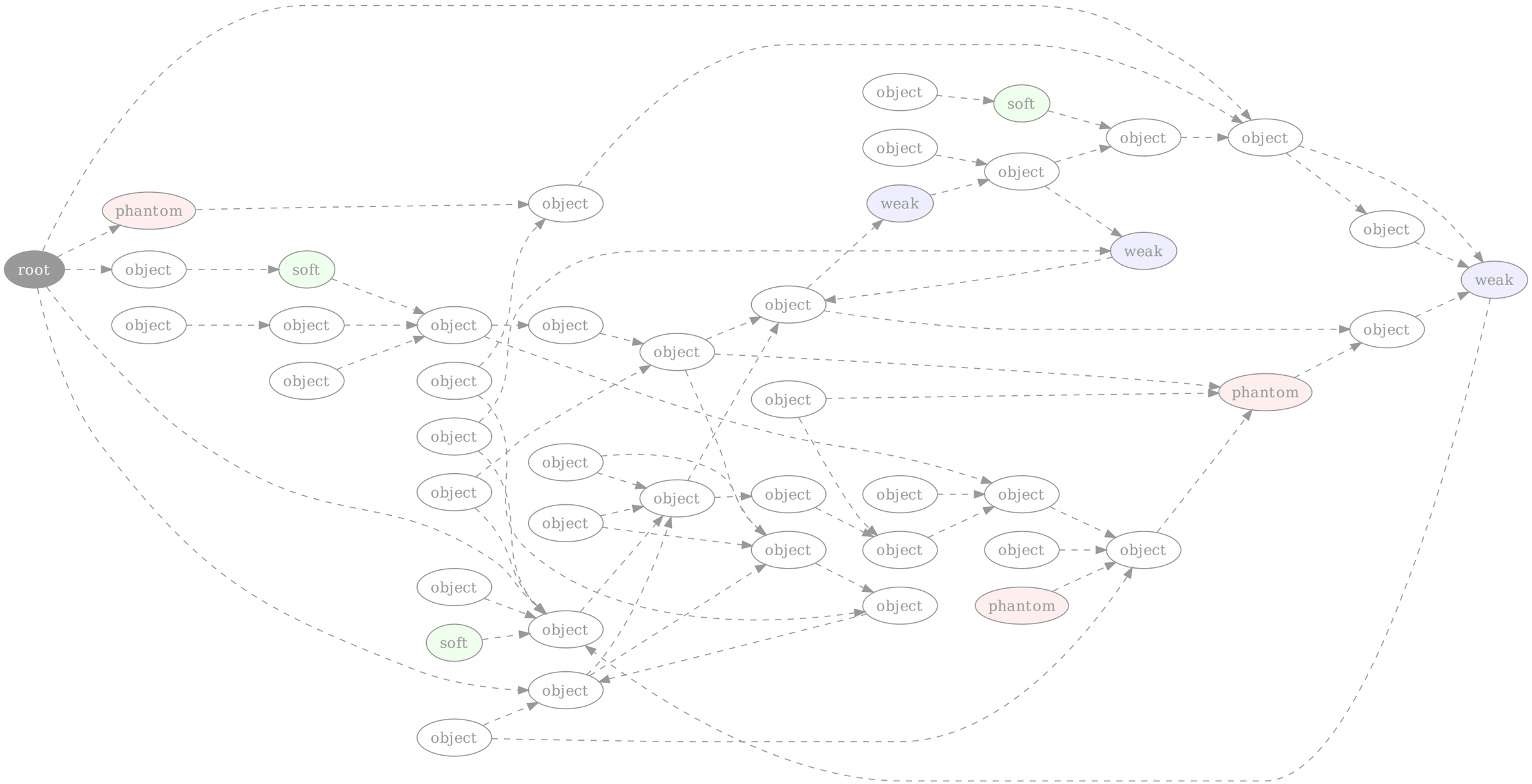
Dante's Heap - The Levels of Reachability

- Strong
- Soft
- Weak
- Finalizer
- Phantom, JNI weak
- **Unreachable**

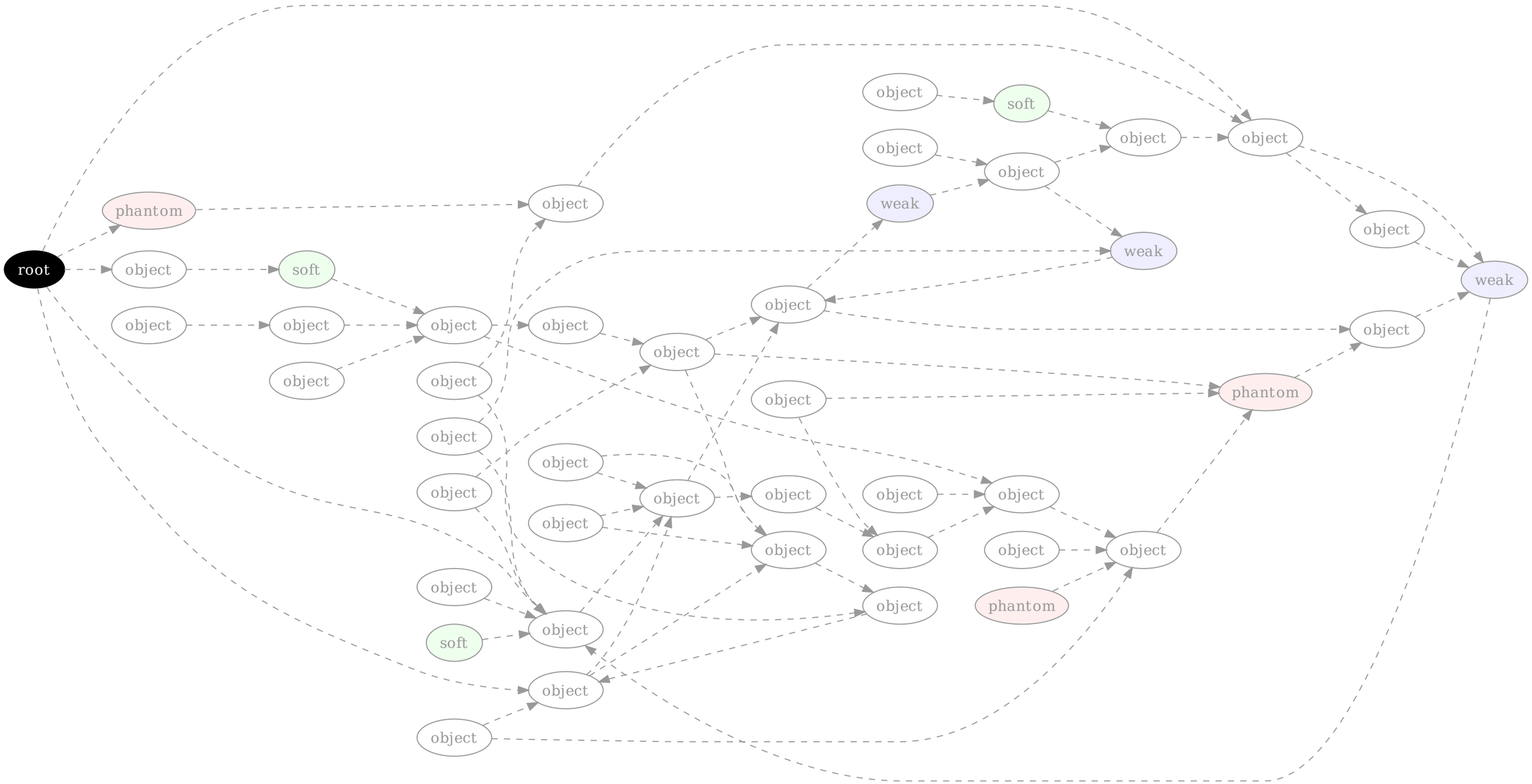
Let's mark and sweep a heap!



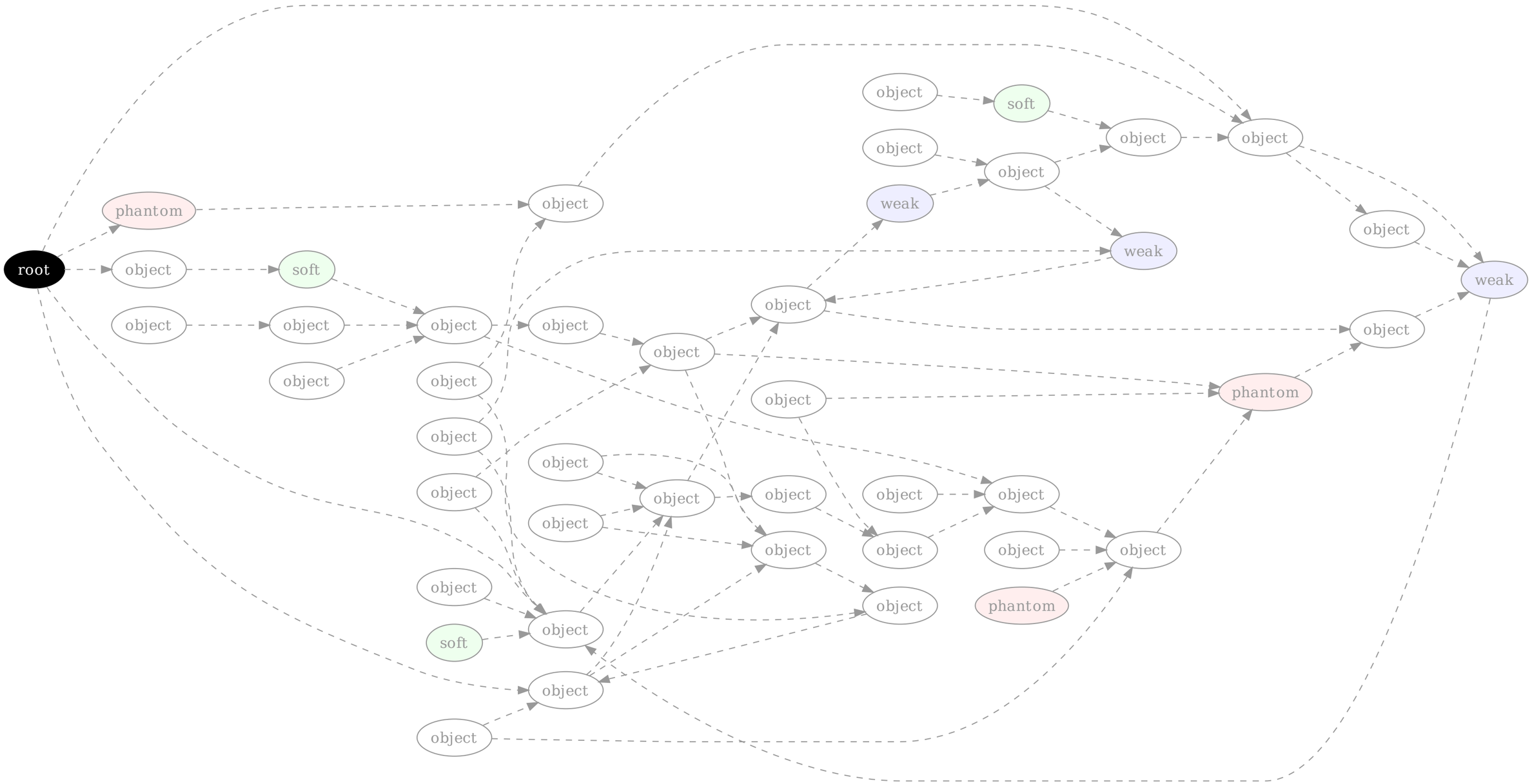
No objects are marked at first.



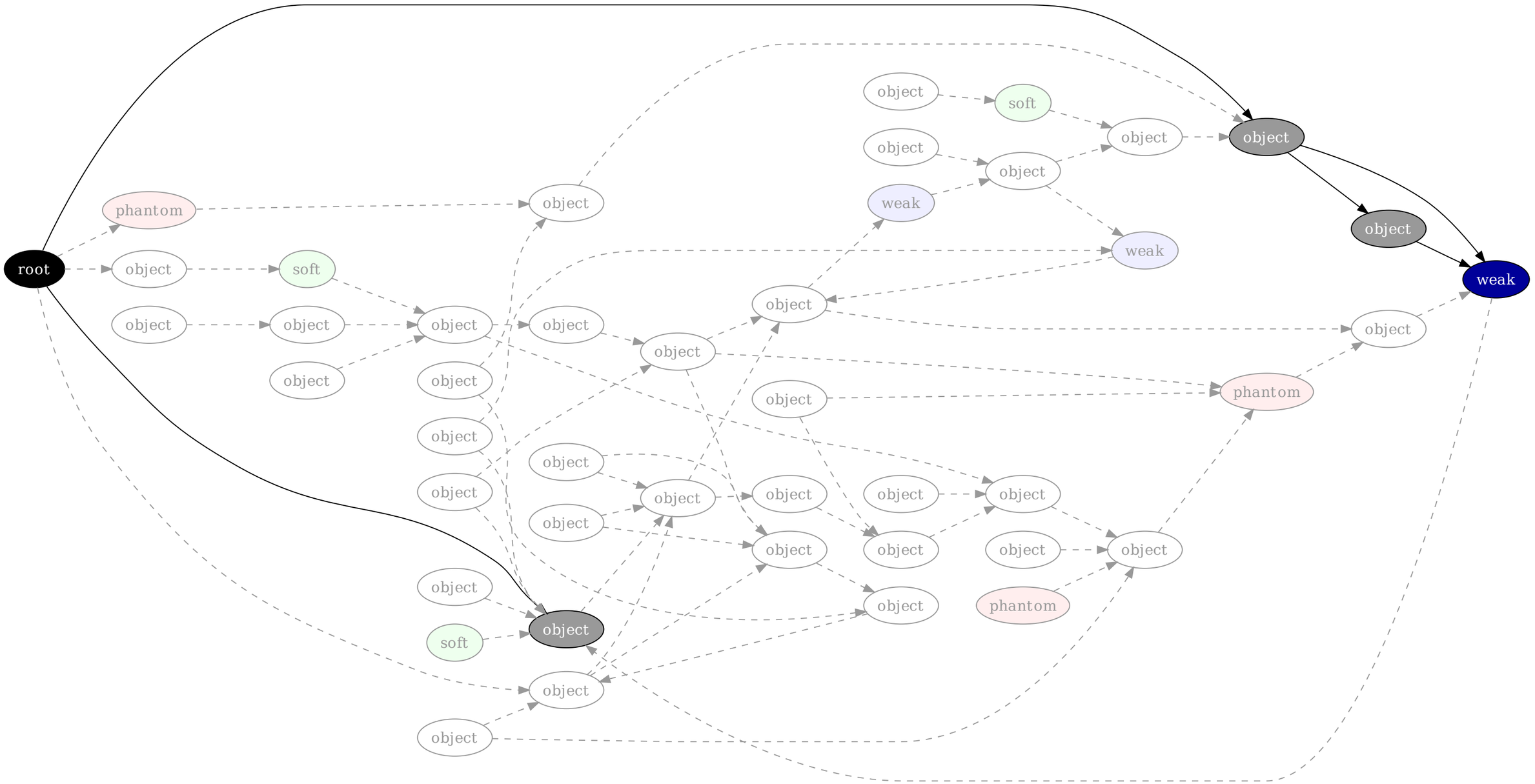
1. Start at a root.



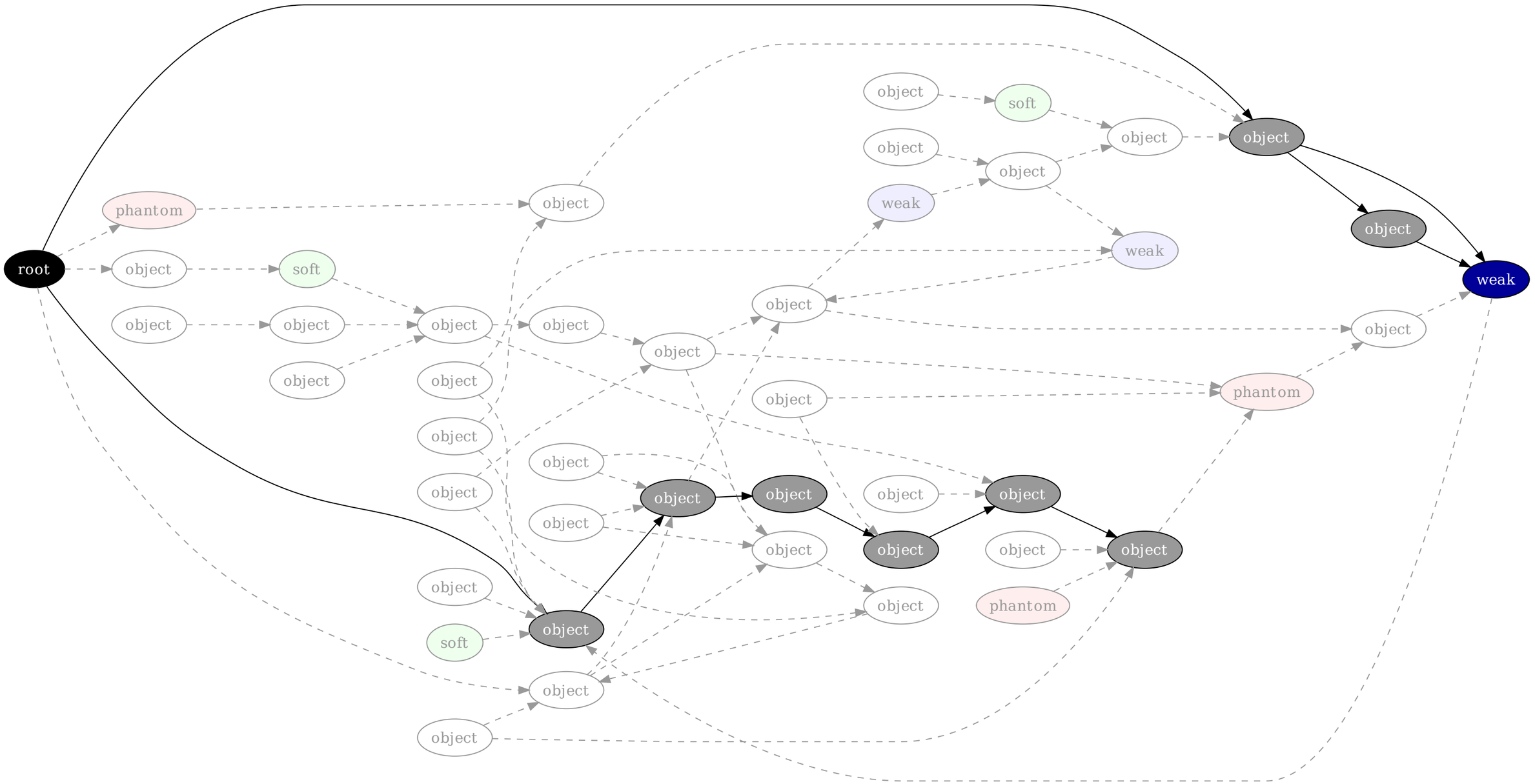
2. Trace and mark strongly-referenced objects.



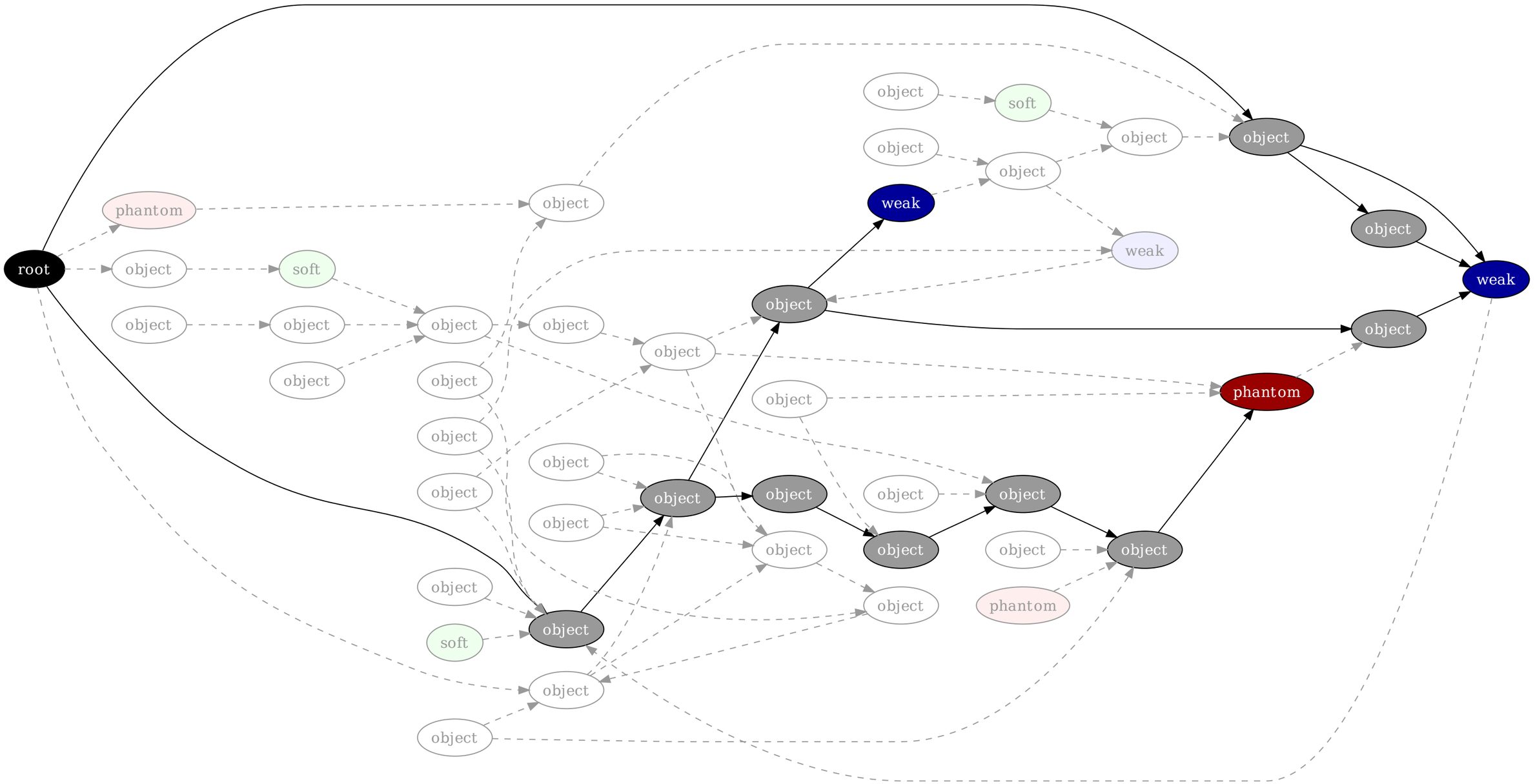
2. Trace and mark strongly-referenced objects.



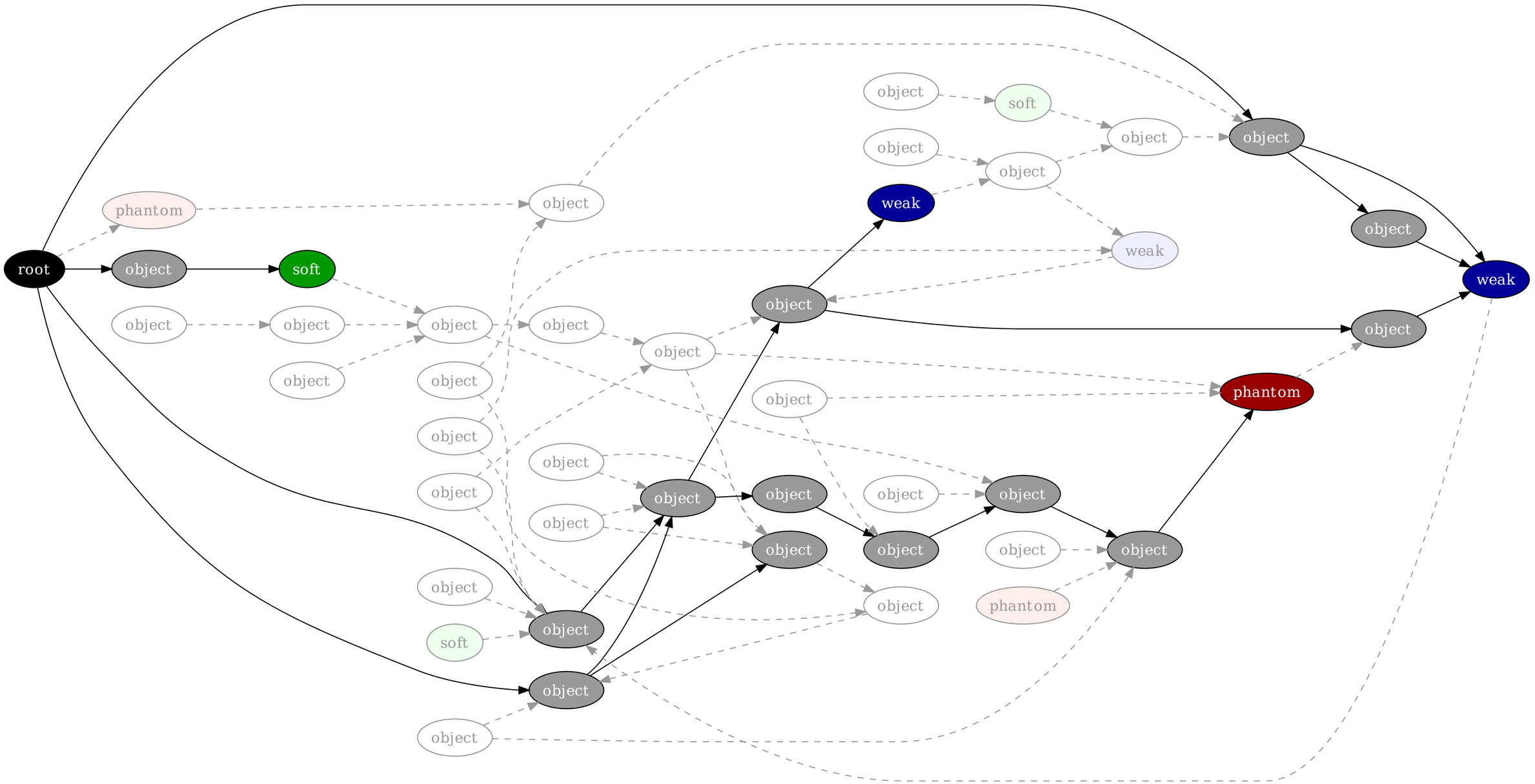
2. Trace and mark strongly-referenced objects.



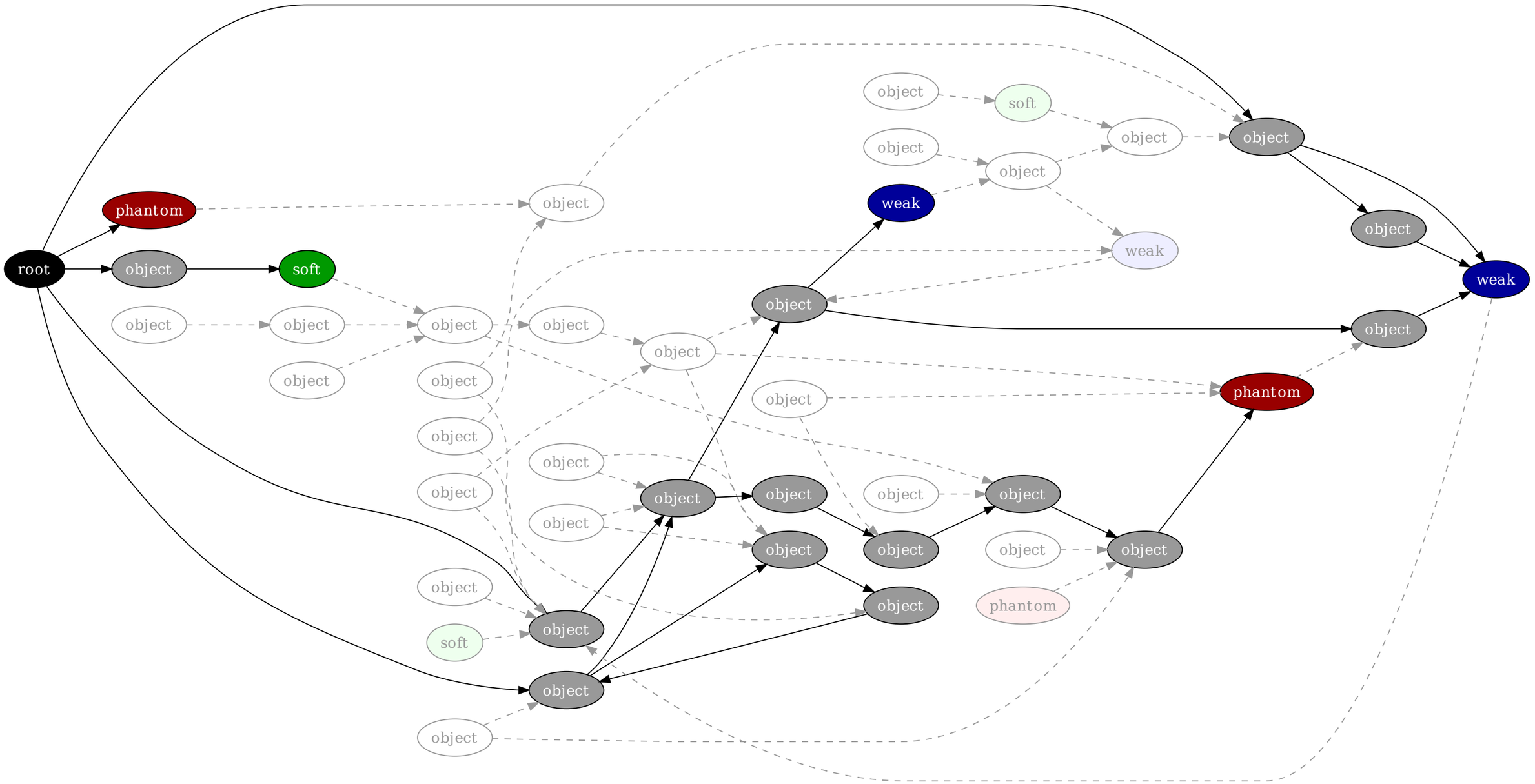
2. Trace and mark strongly-referenced objects.



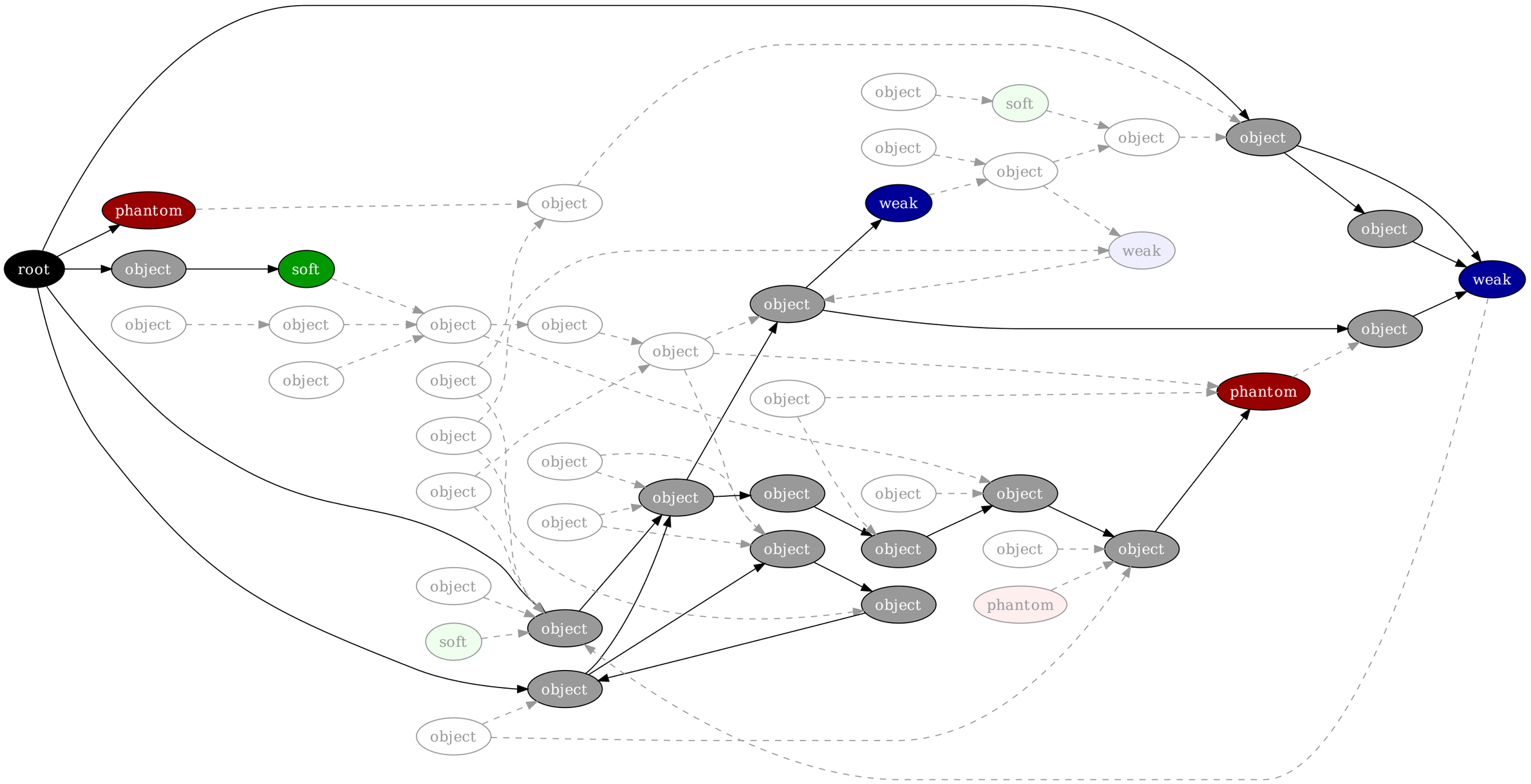
2. Trace and mark strongly-referenced objects.



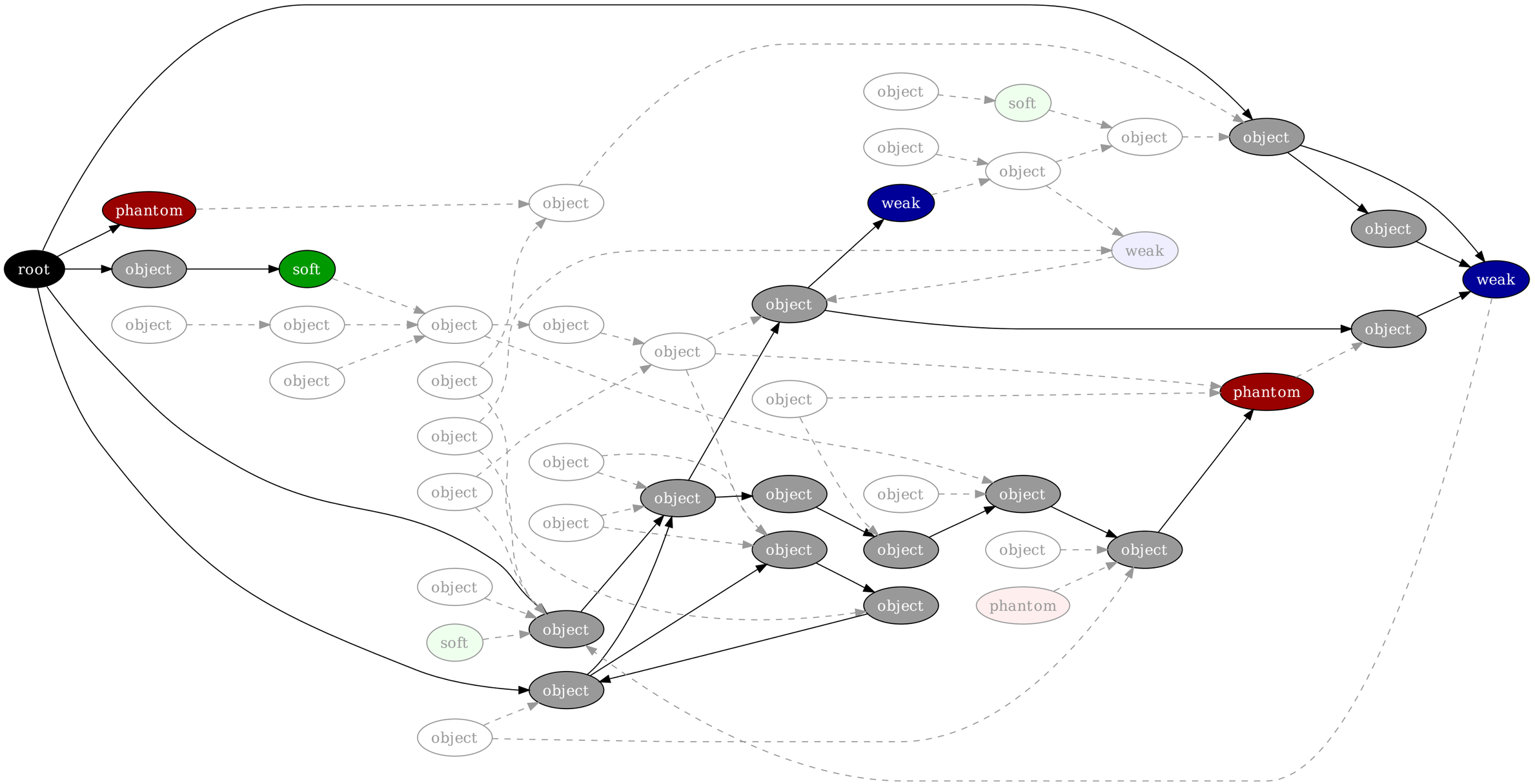
2. Trace and mark strongly-referenced objects.



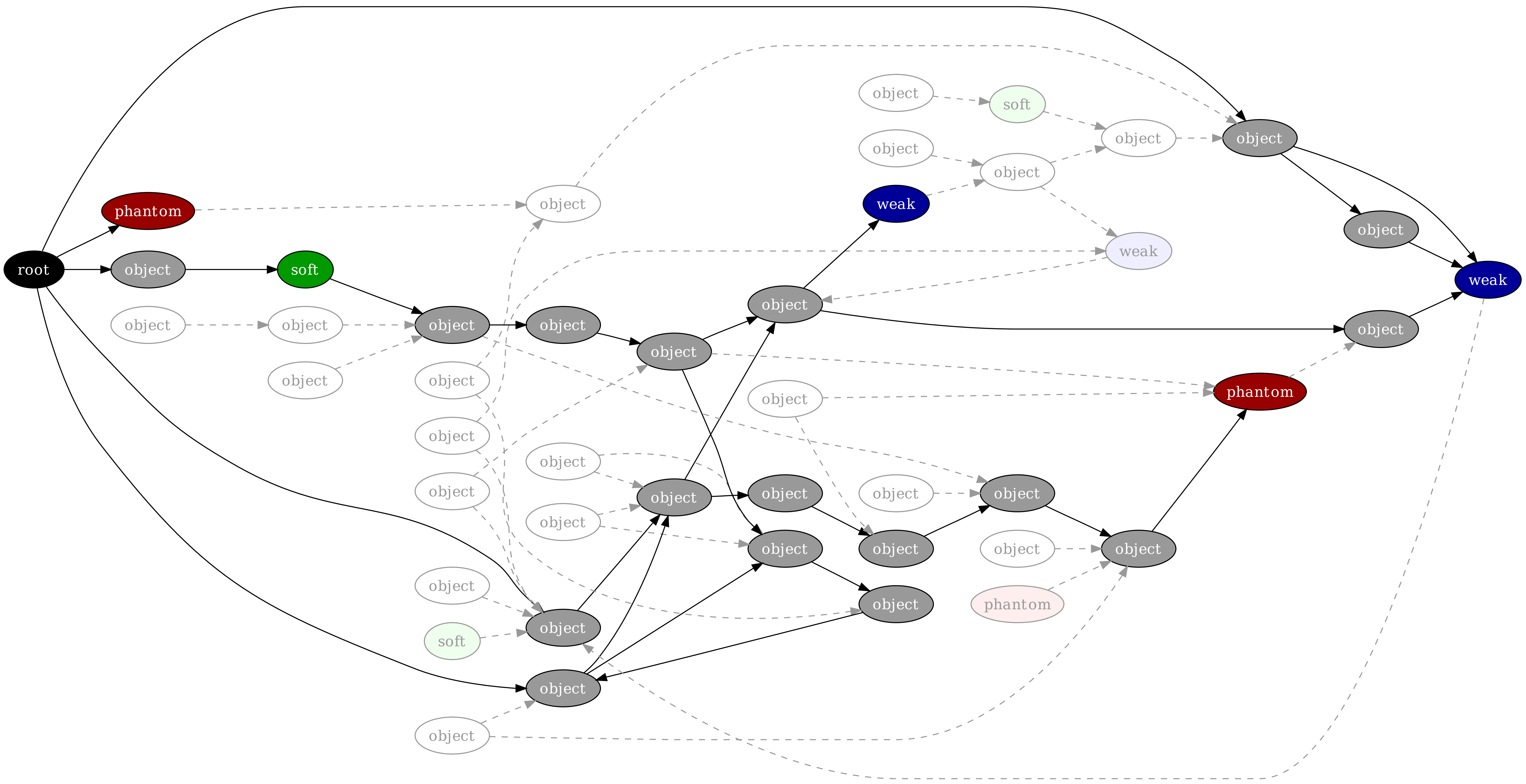
3. Optionally clear soft references.



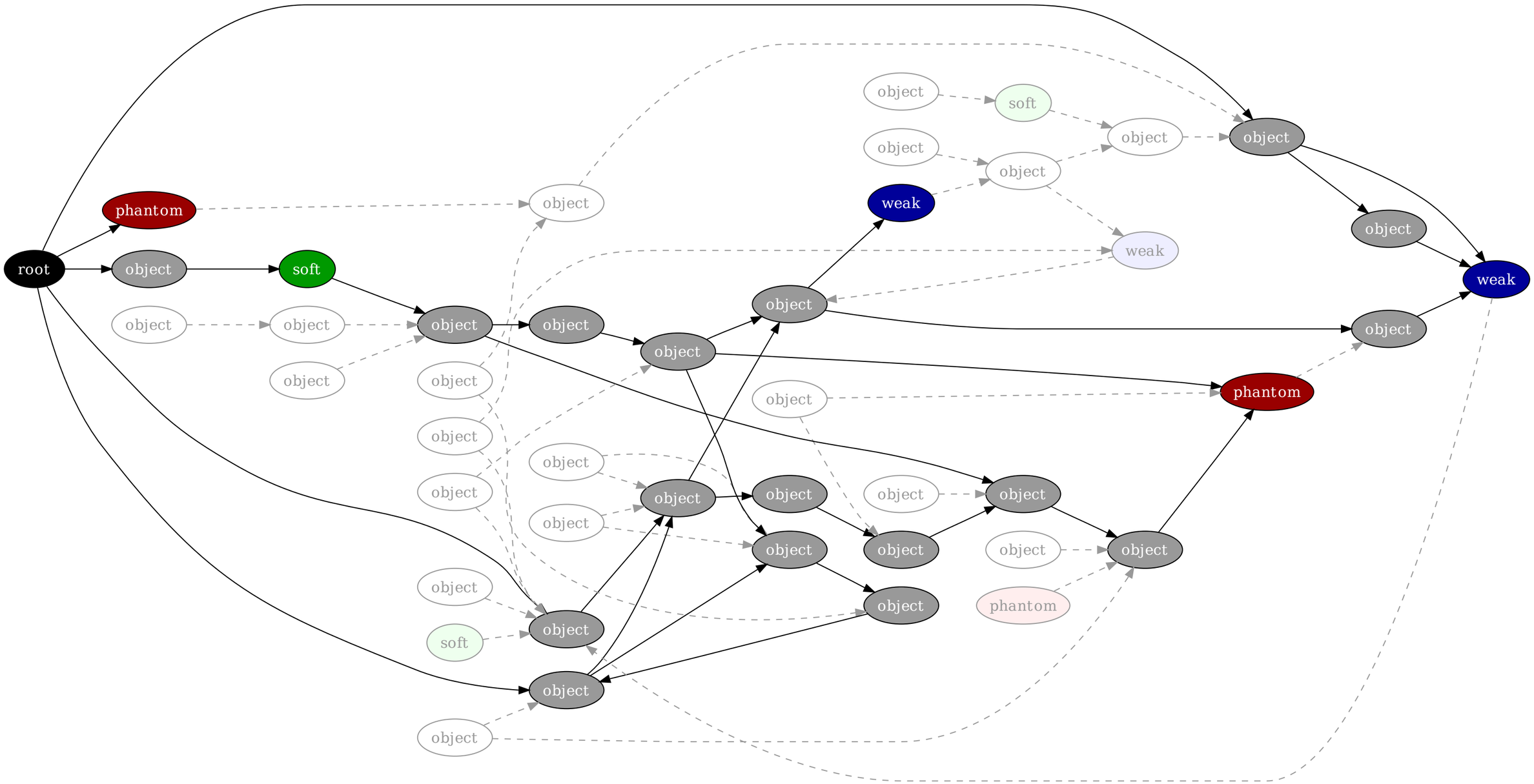
4. Trace and mark softly-referenced objects.



4. Trace and mark softly-referenced objects.



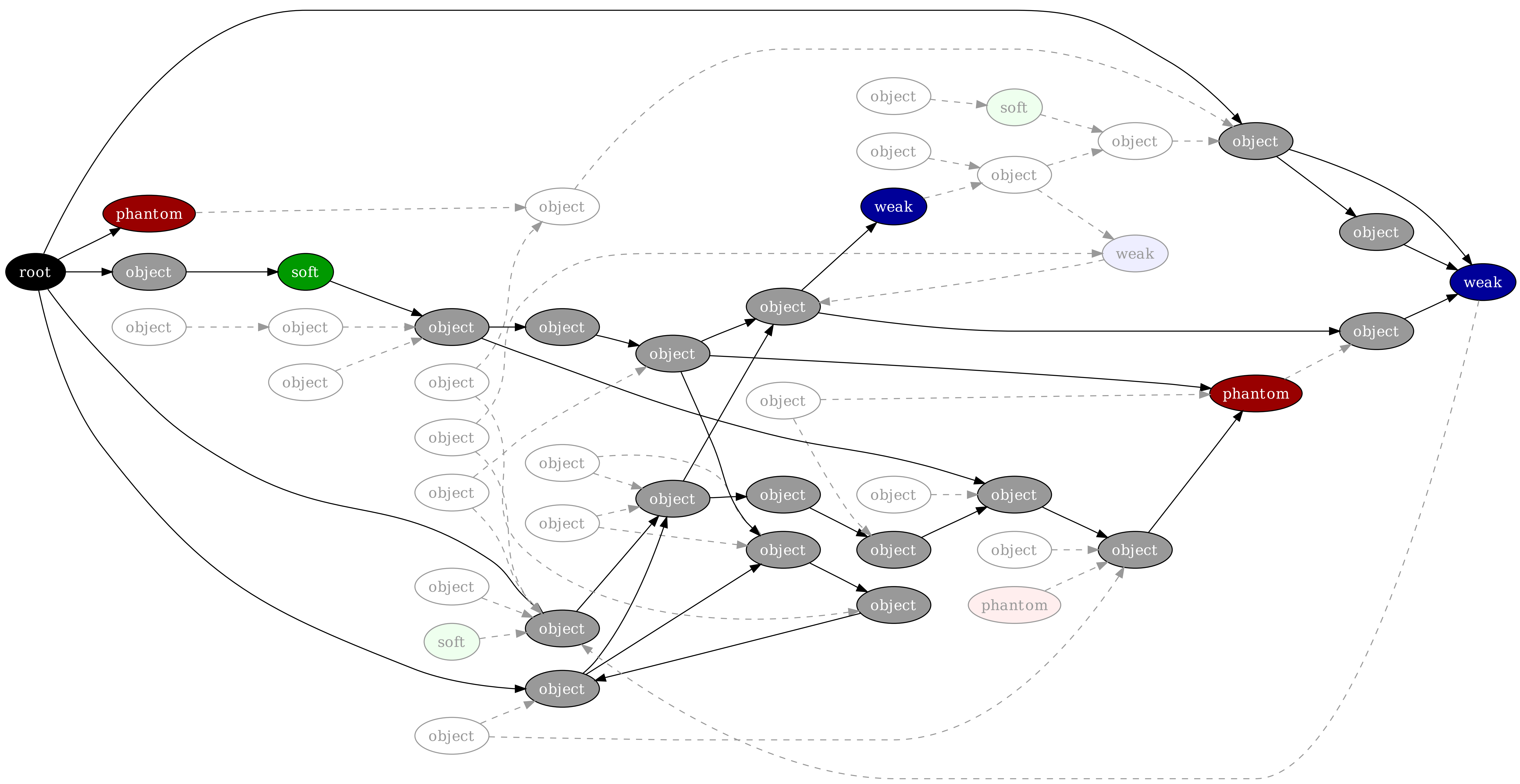
4. Trace and mark softly-referenced objects.



JavaOne

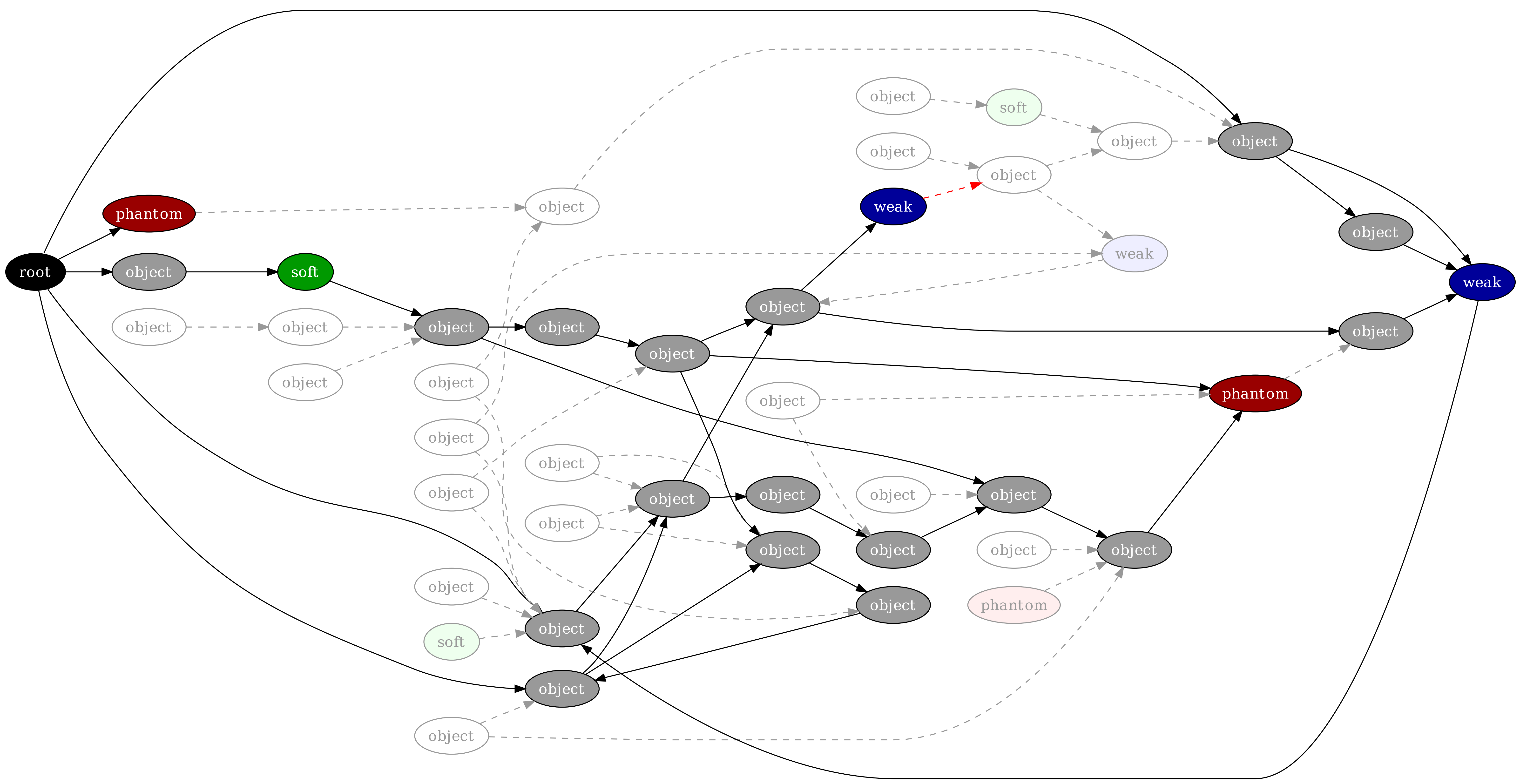
Java

5. Clear weak references.

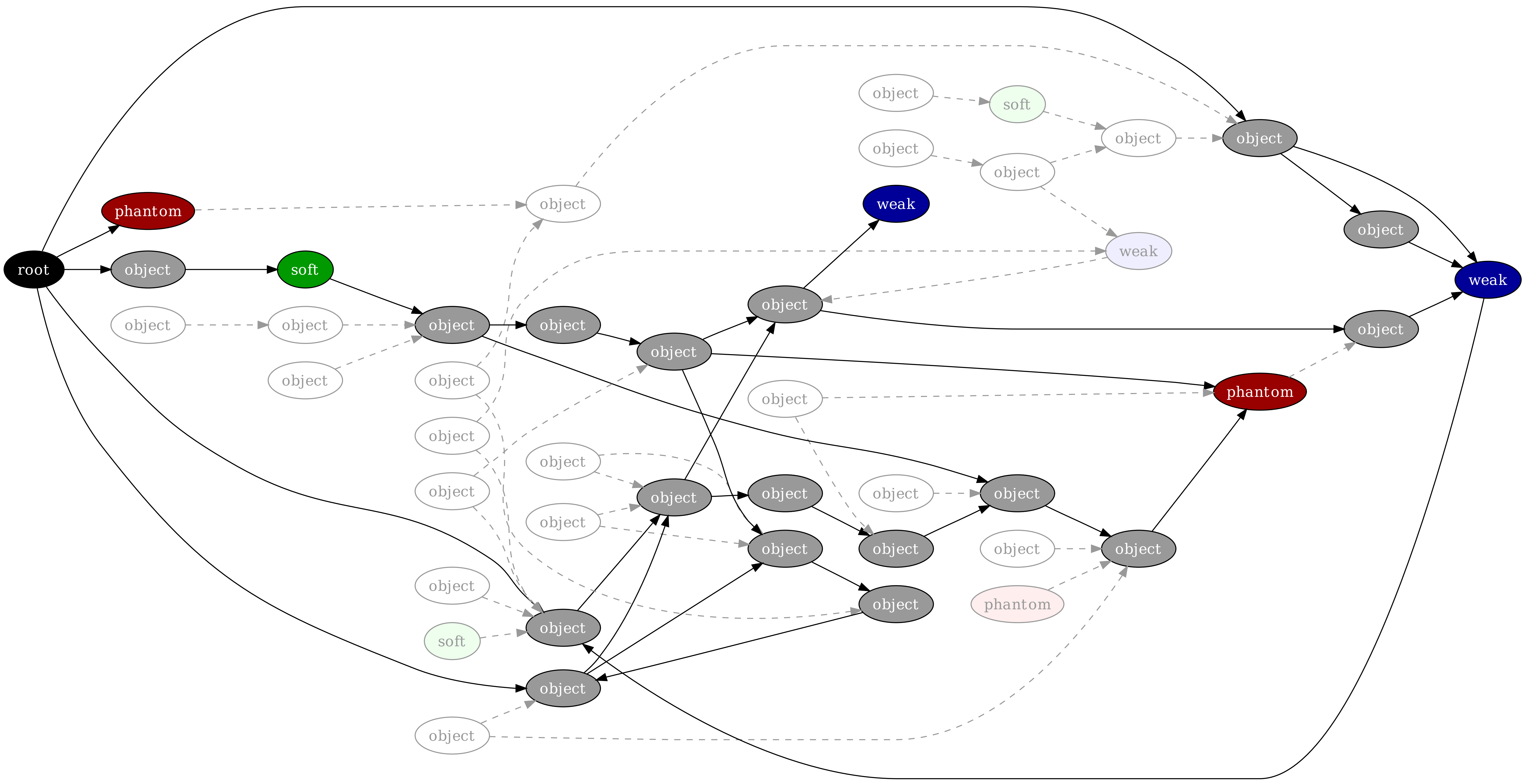


Sun microsystems

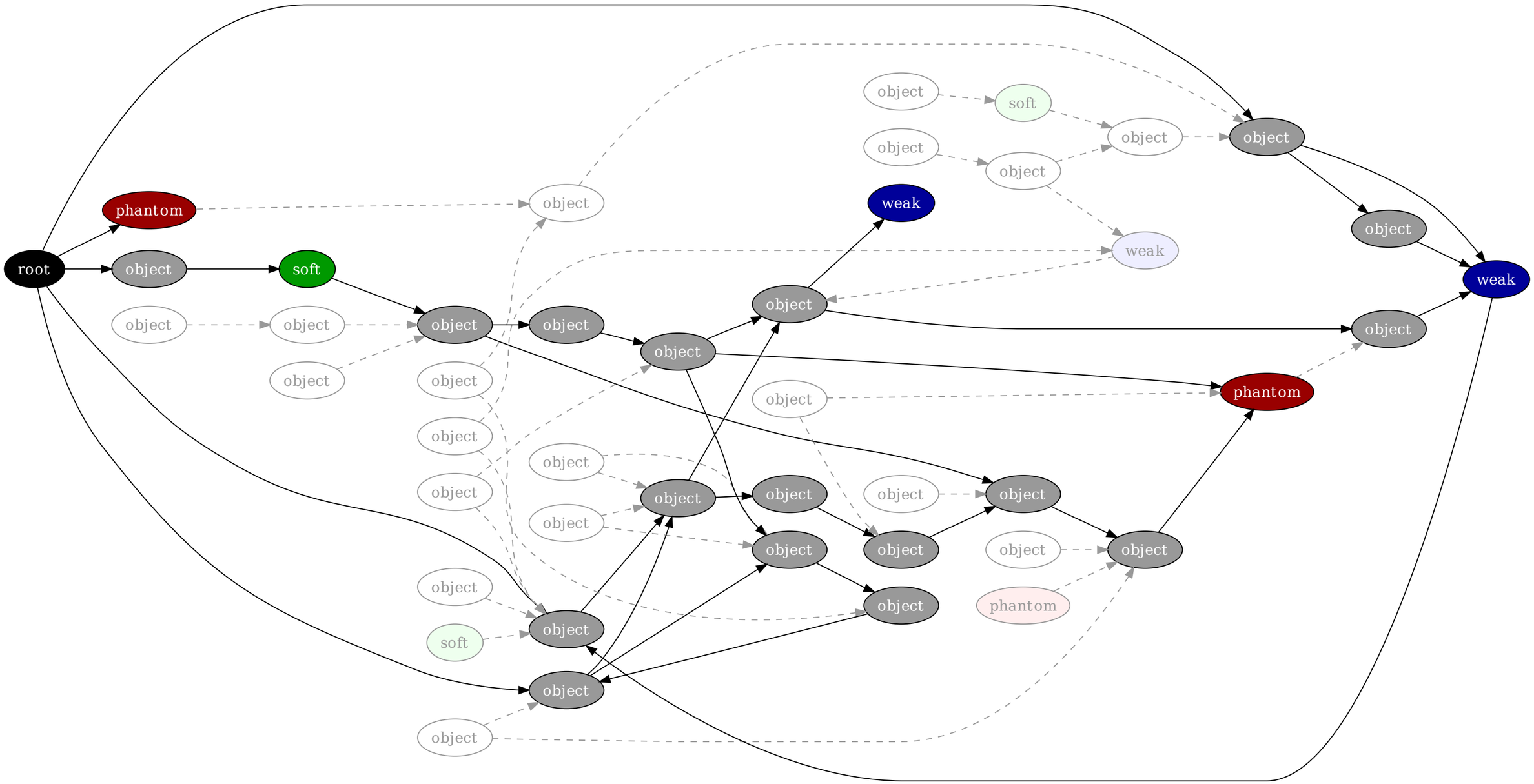
5. Clear weak references.



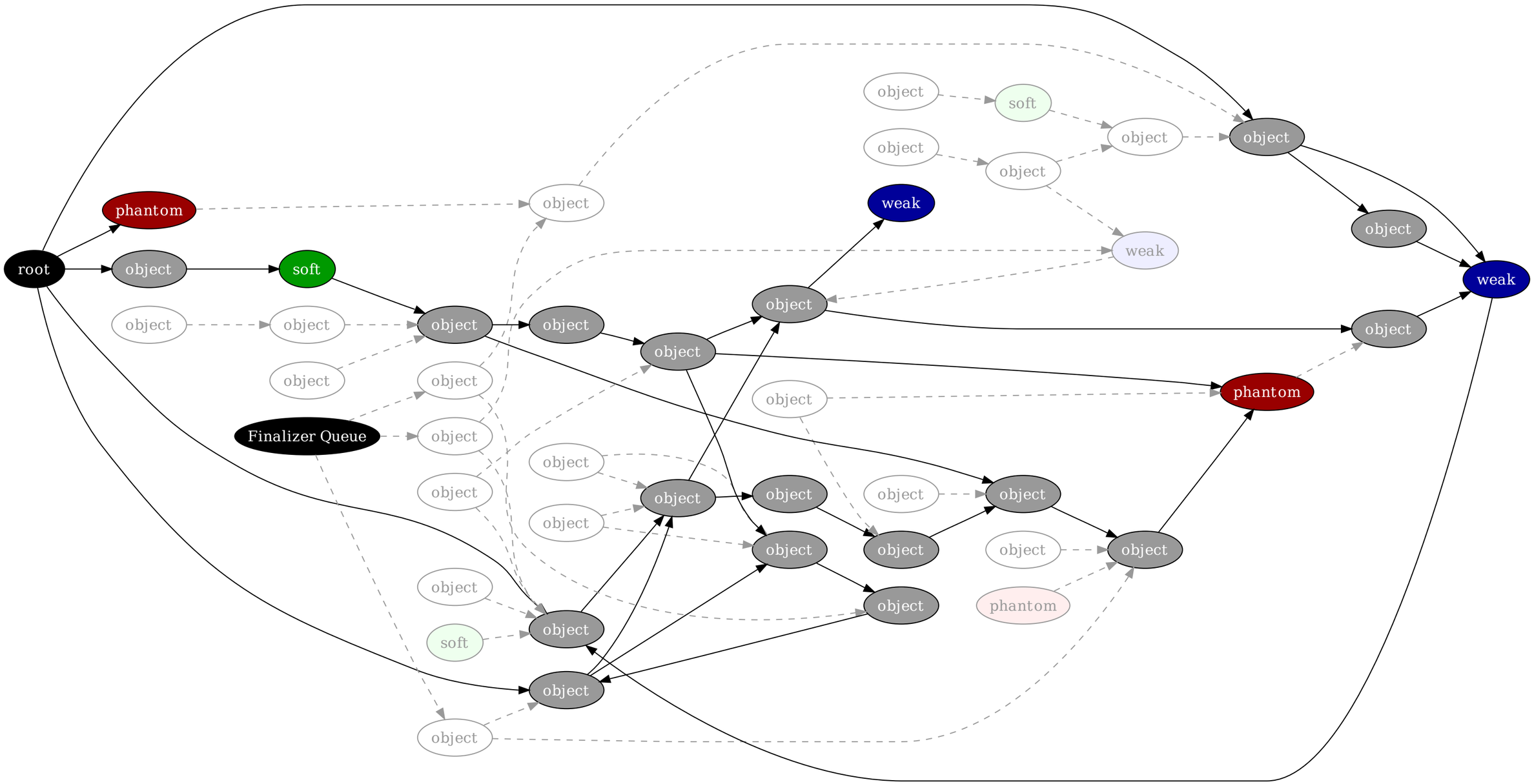
5. Clear weak references.



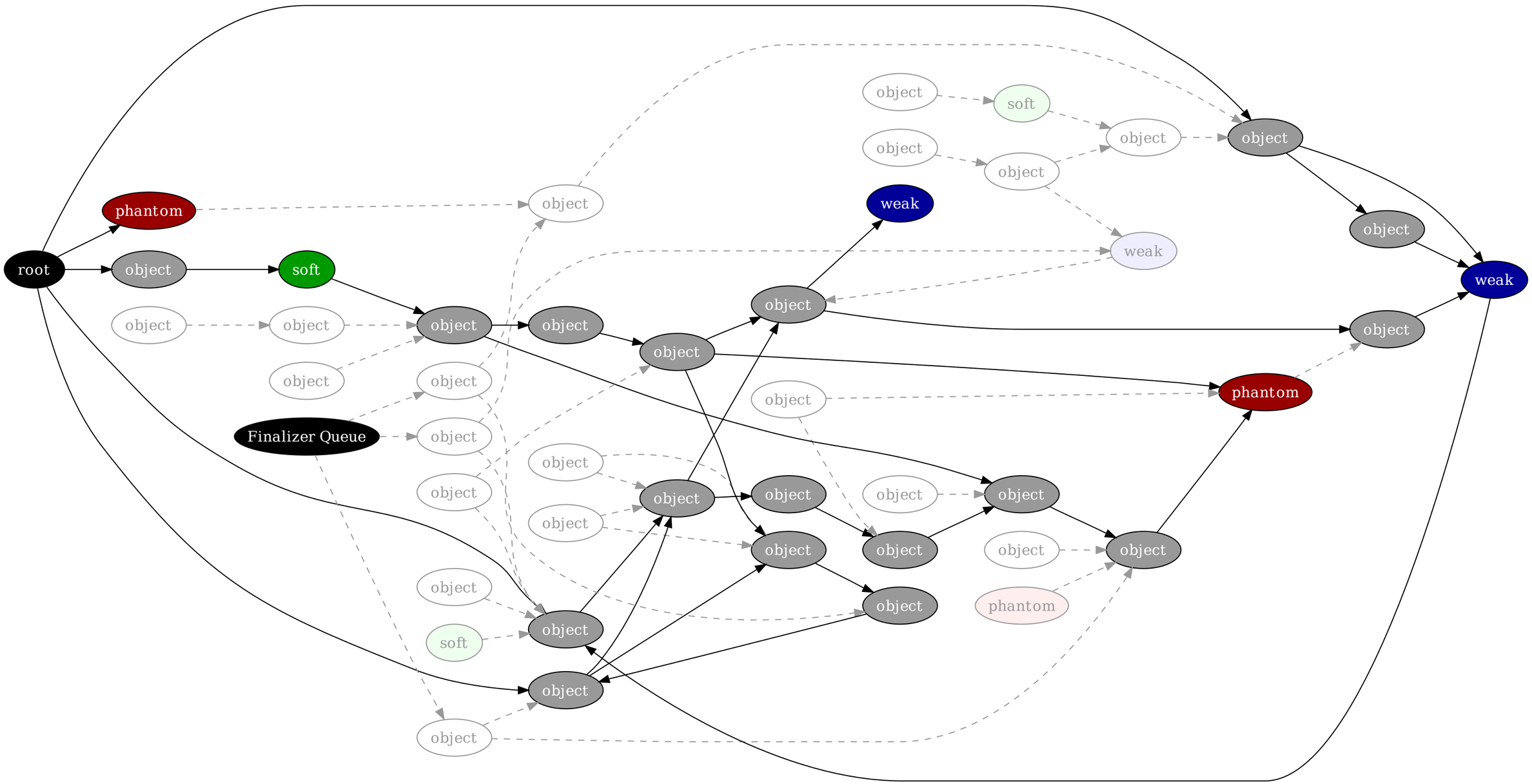
6. Enqueue finalizable objects.



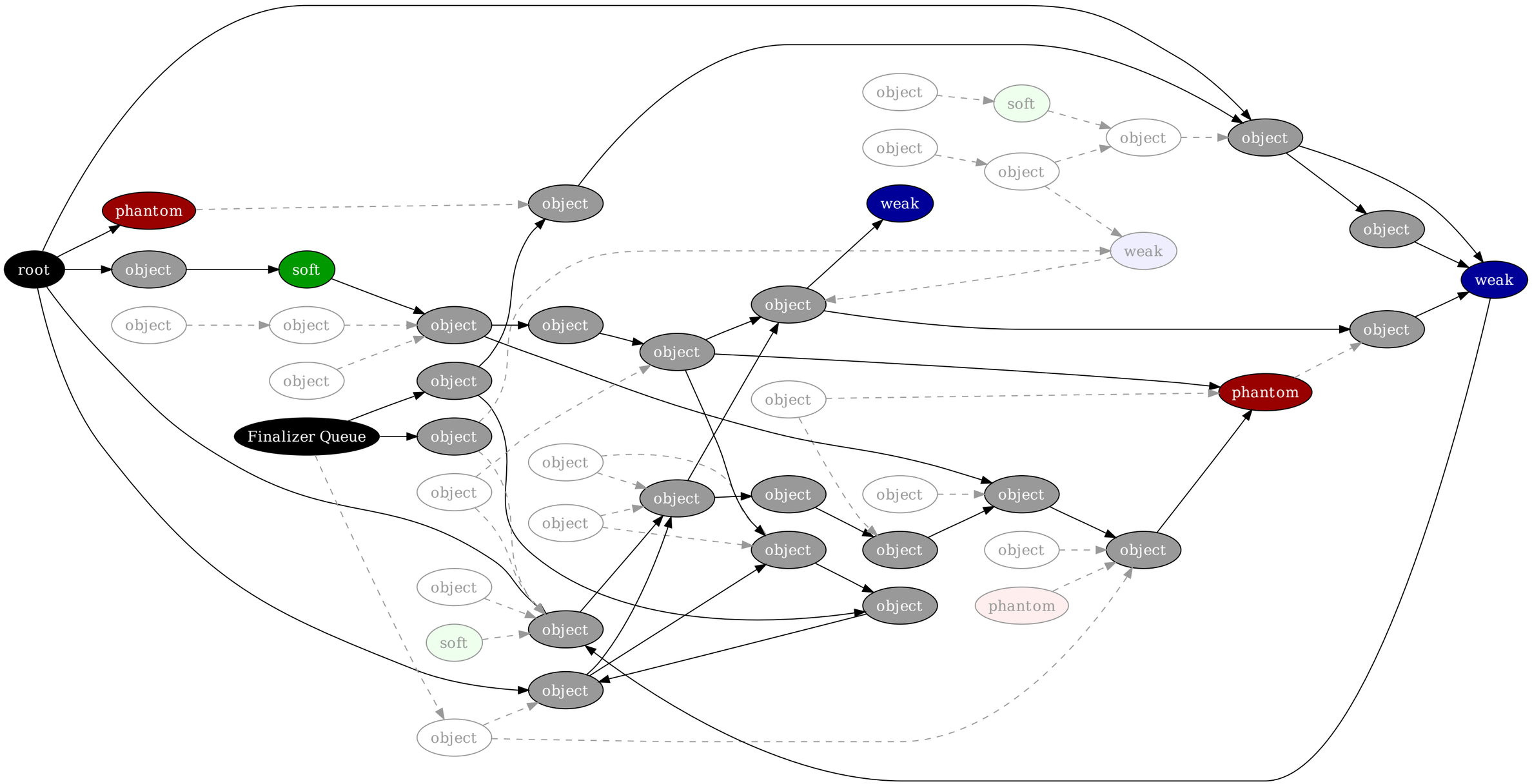
6. Enqueue finalizable objects.



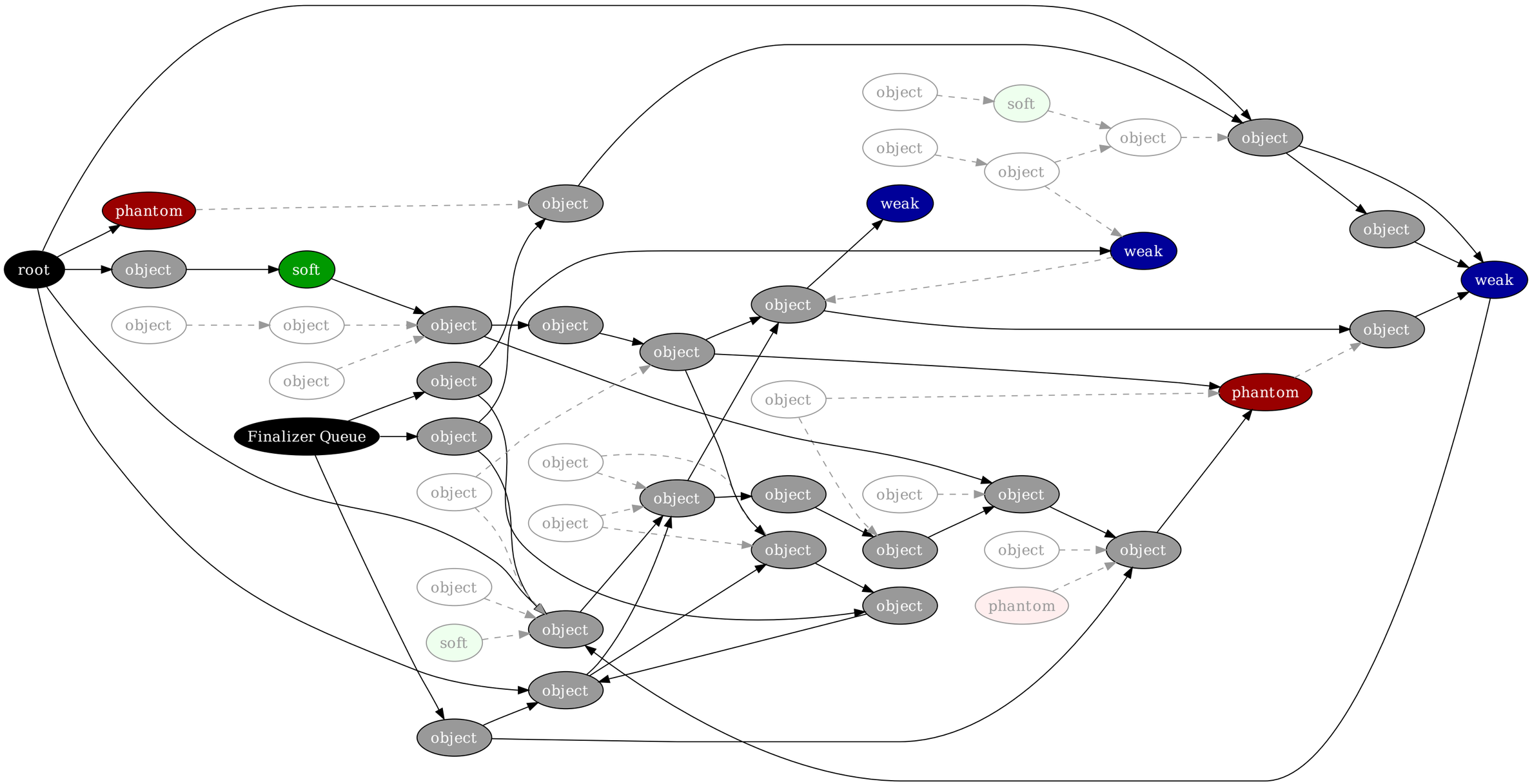
7. Repeat steps 1 through 5 for the queue.



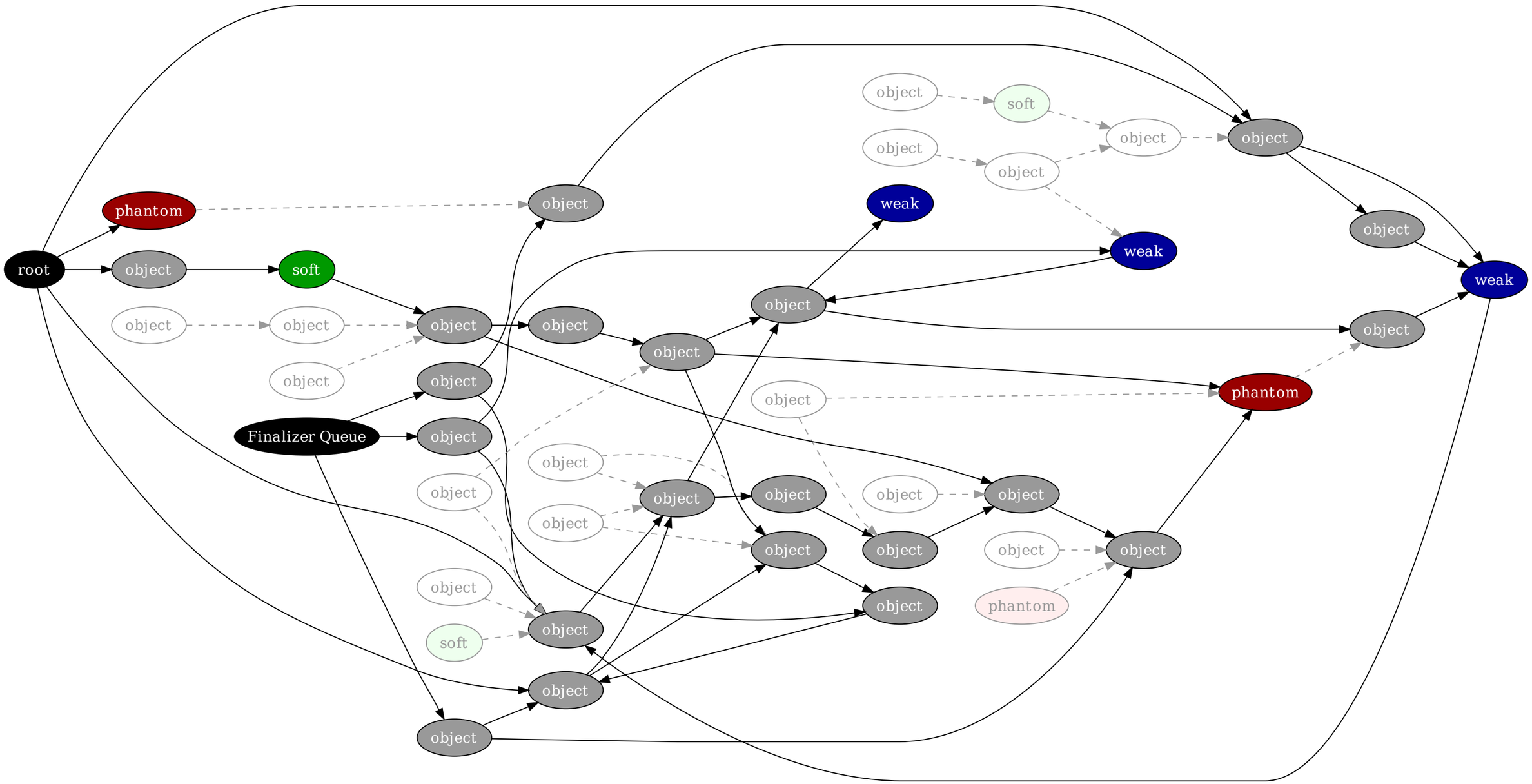
7. Repeat steps 1 through 5 for the queue.



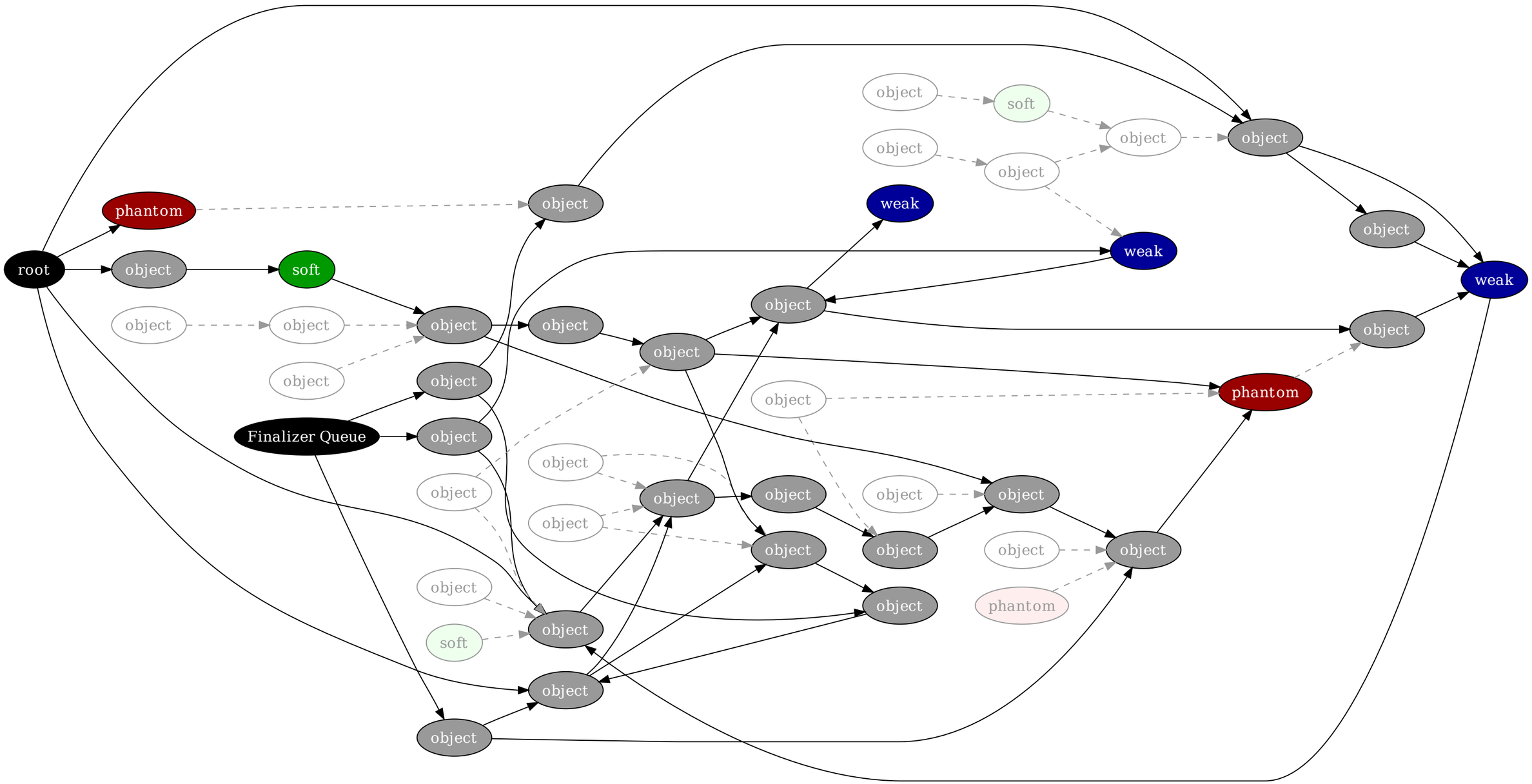
7. Repeat steps 1 through 5 for the queue.



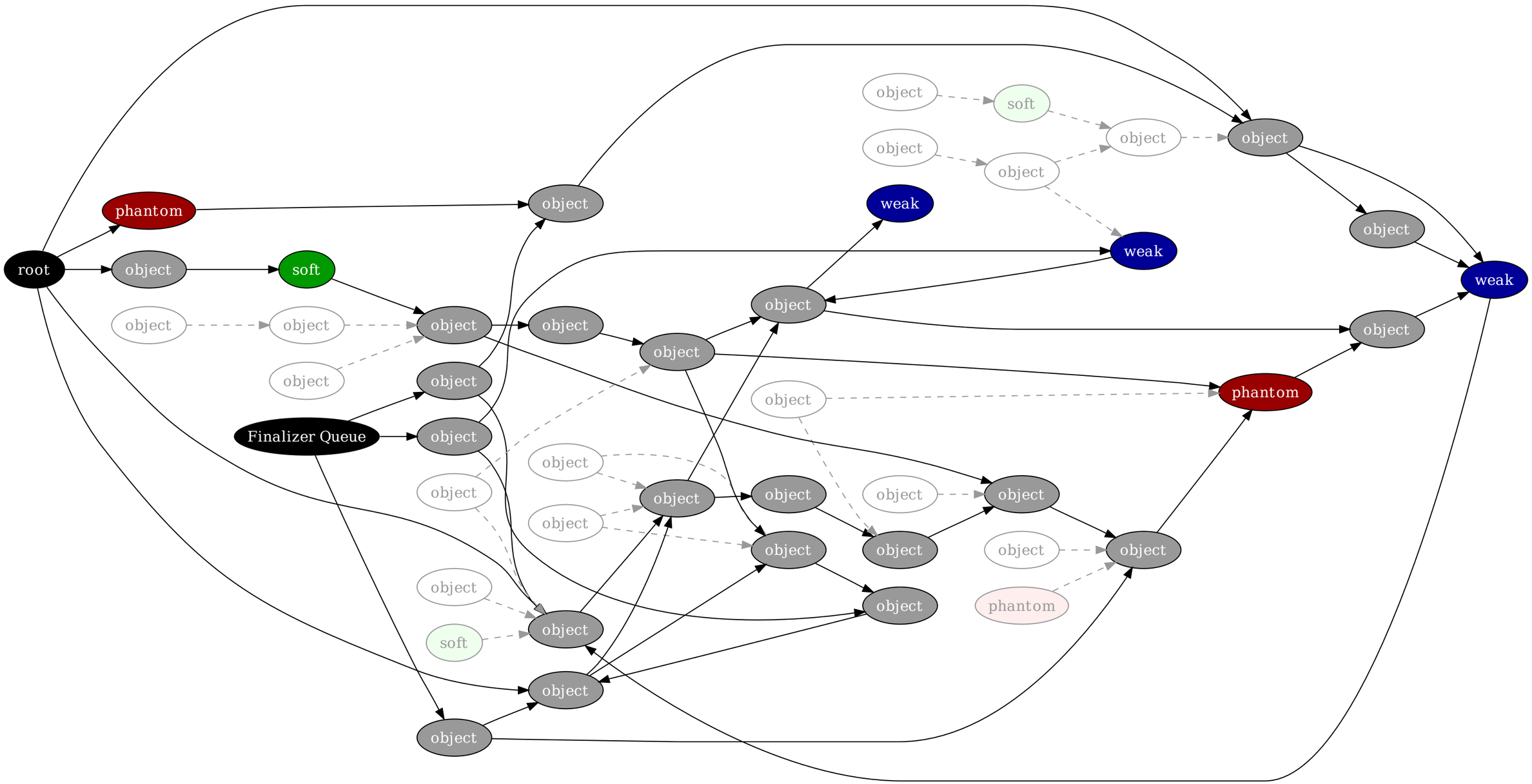
7. Repeat steps 1 through 5 for the queue.



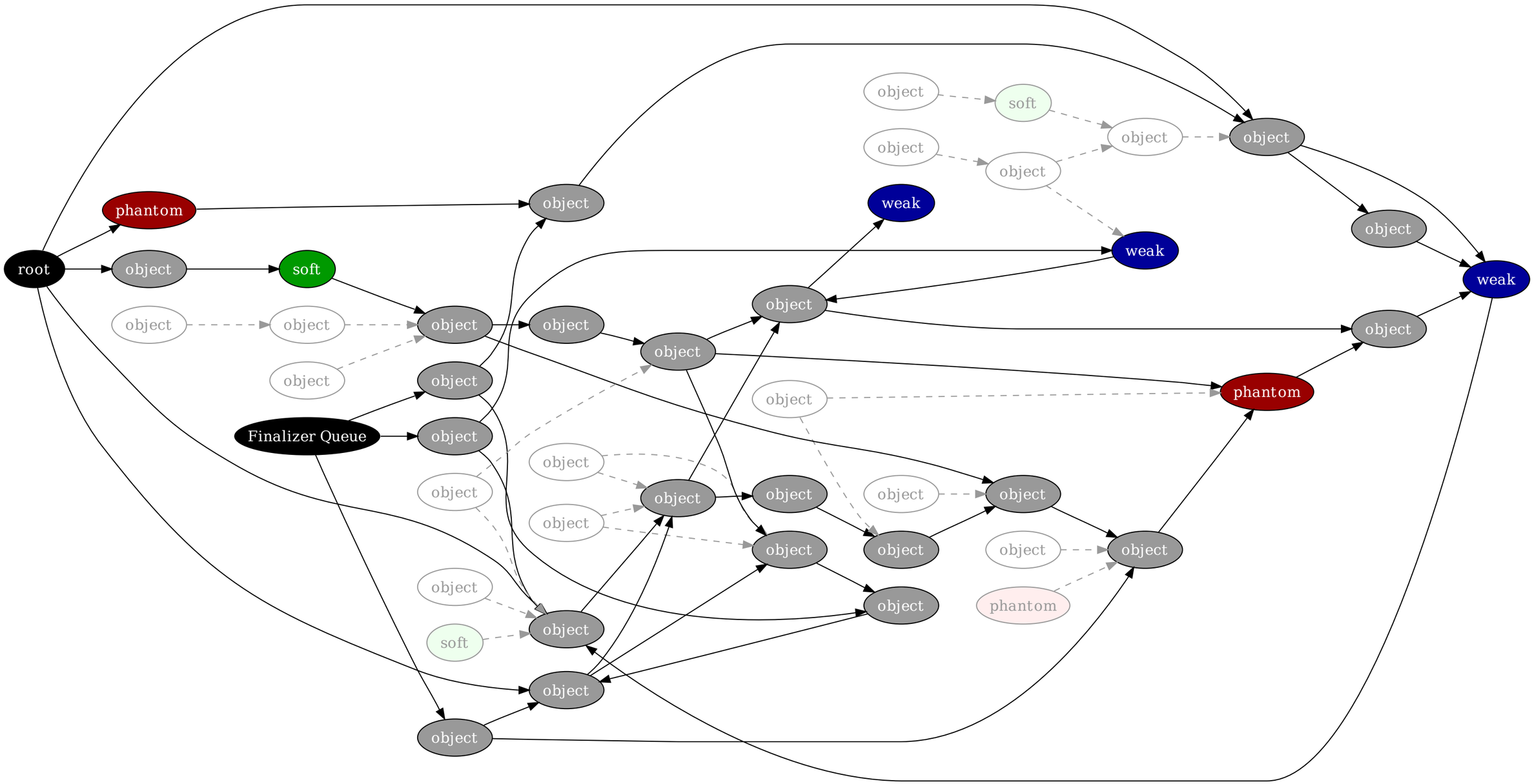
8. Possibly enqueue phantom references.



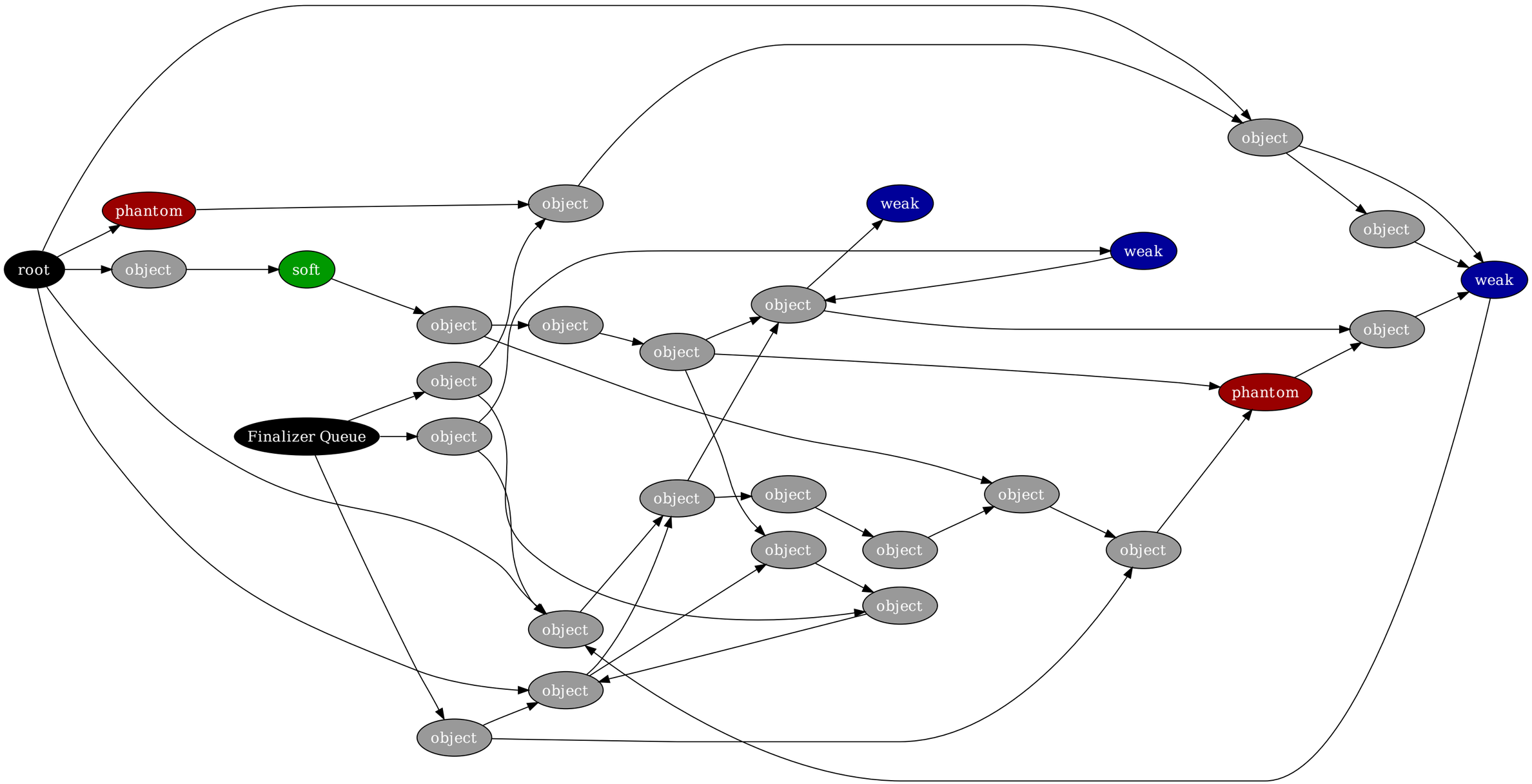
8. Possibly enqueue phantom references.



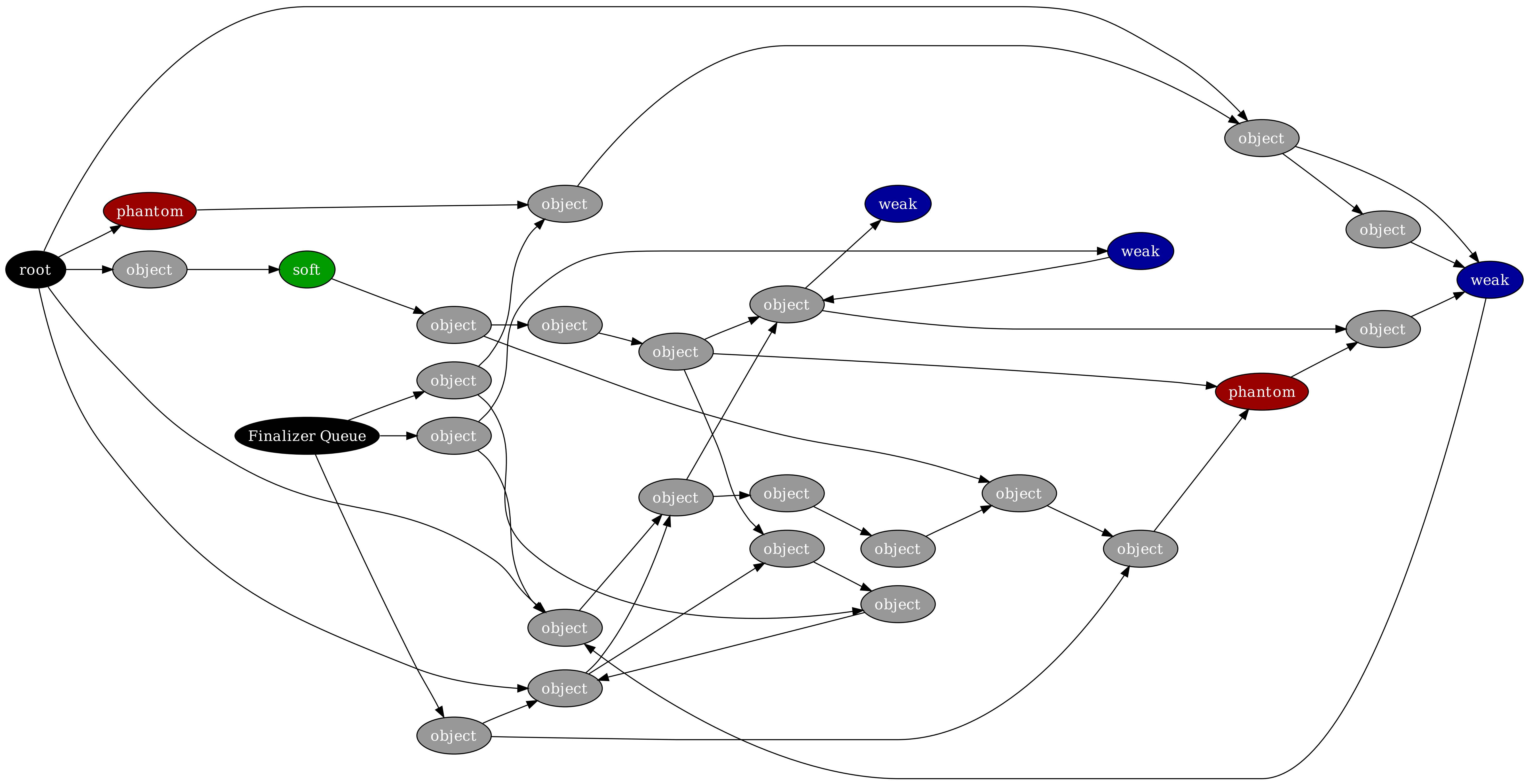
9. The remaining objects are dead.



9. The remaining objects are dead.



10. Repeat.


root
phantom
object
soft
object
object
object
weak
weak
object
weak
object
object
object
object
phantom
Finalizer Queue
object
object
object
object
object
object
object
object
object
object
object
object

Recap

1. Start at a root.

2. Trace and mark strongly-referenced objects.

3. Optionally clear soft references.

4. Trace and mark softly-referenced objects.

5. Clear weak references.

6. Enqueue finalizable objects.

7. Repeat steps 1 through 5 for the queue.

8. Possibly enqueue phantom references.

9. The remaining objects are dead.

10. Repeat.

Weak references aren't for caching!

- Many collectors will reclaim weak refs immediately.
- Use soft reference for caching, as intended:

“Virtual machine implementations are encouraged to bias against clearing recently-created or recently-used soft references.”

- The `SoftReference` documentation