O'REILLY

oscon java

open source convention

oscon.com/java

The Ghost in the Virtual Machine

A Reference to References

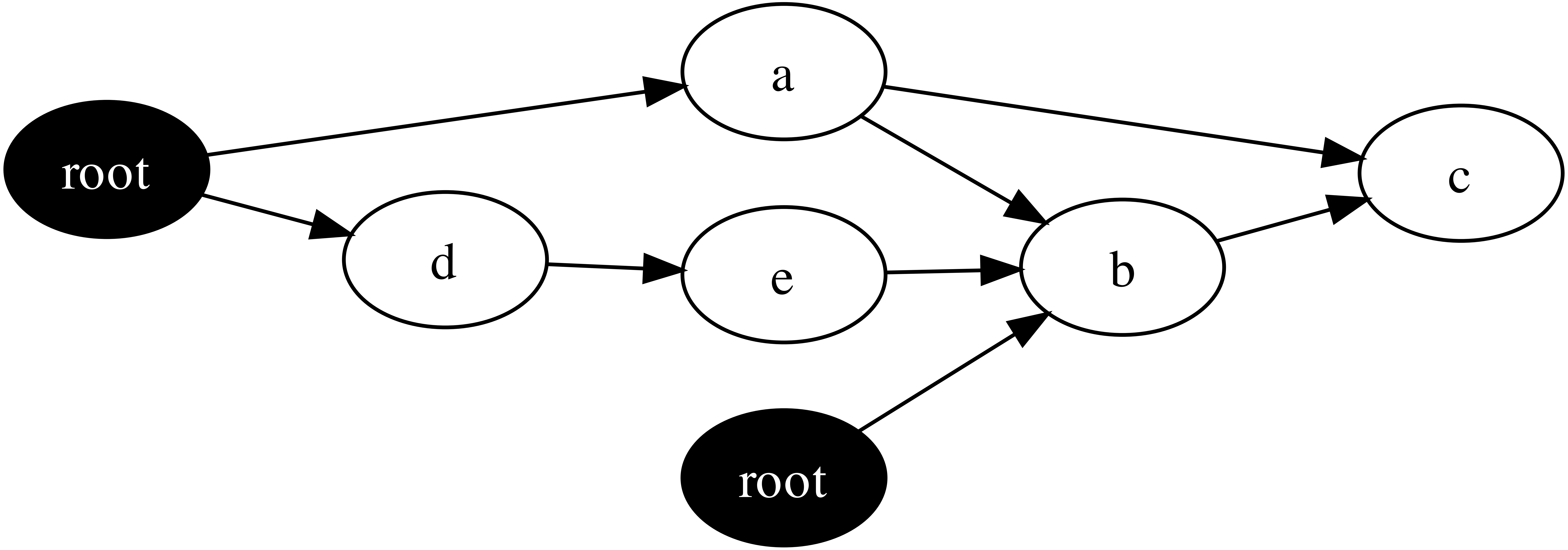
Bob Lee

Square Inc.

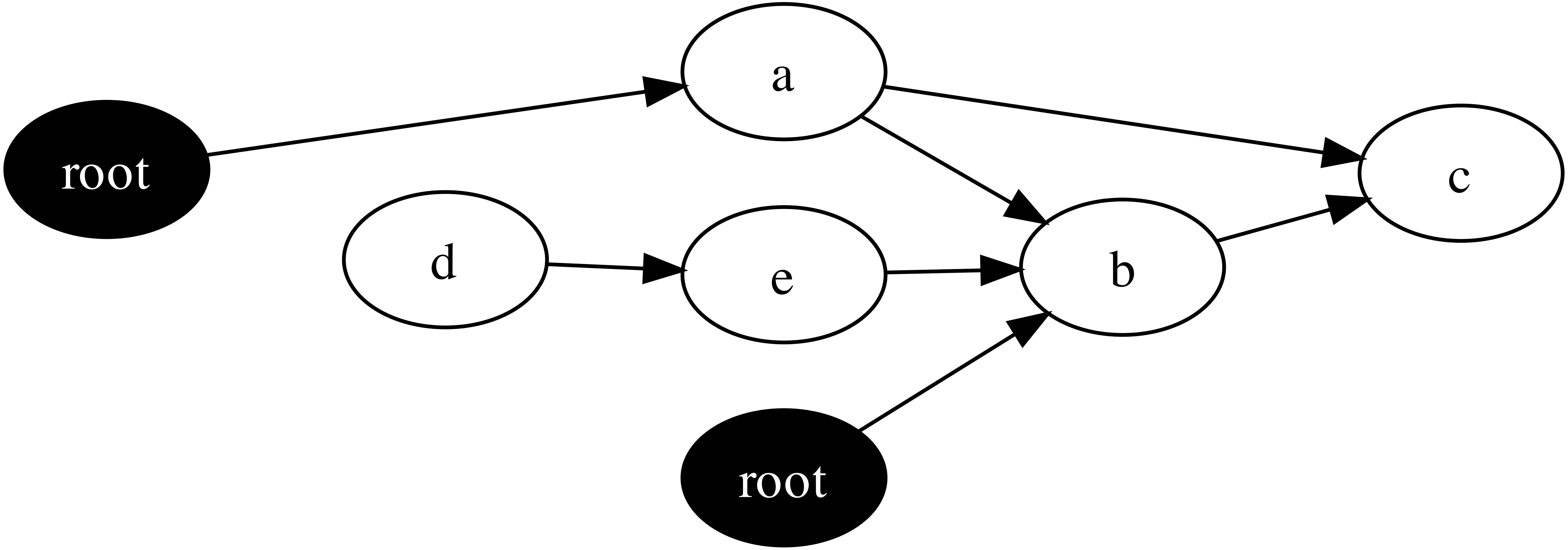
Goals

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

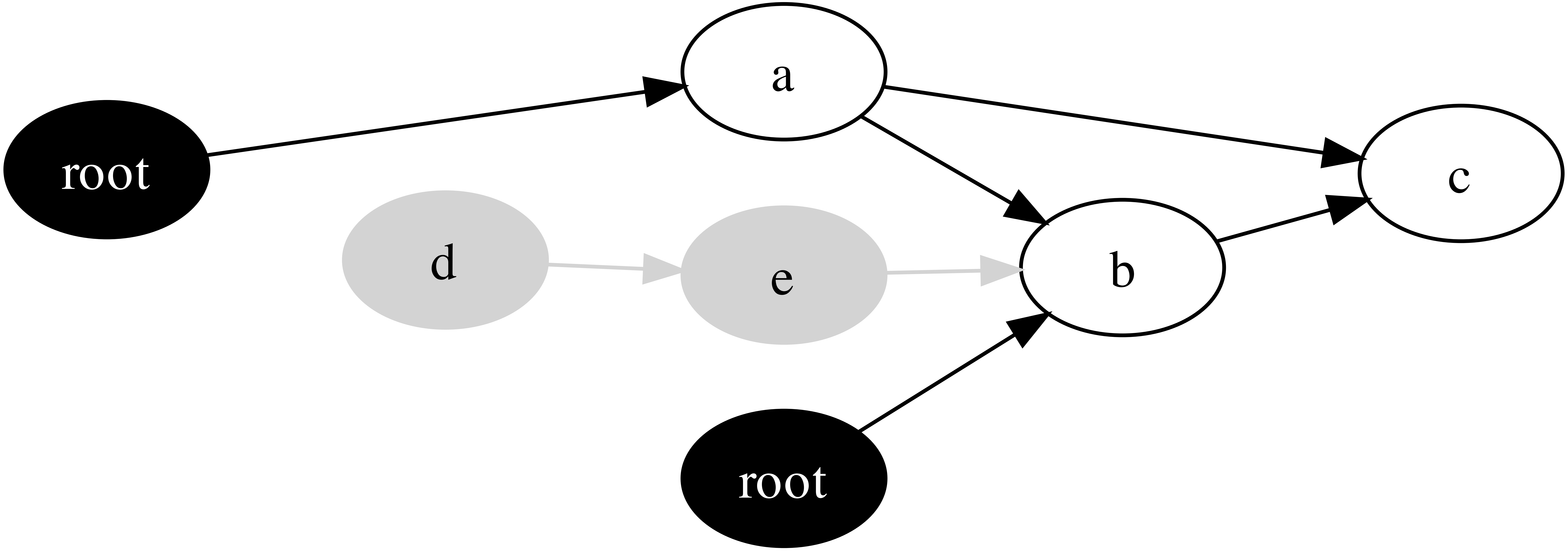
How does garbage collection work?



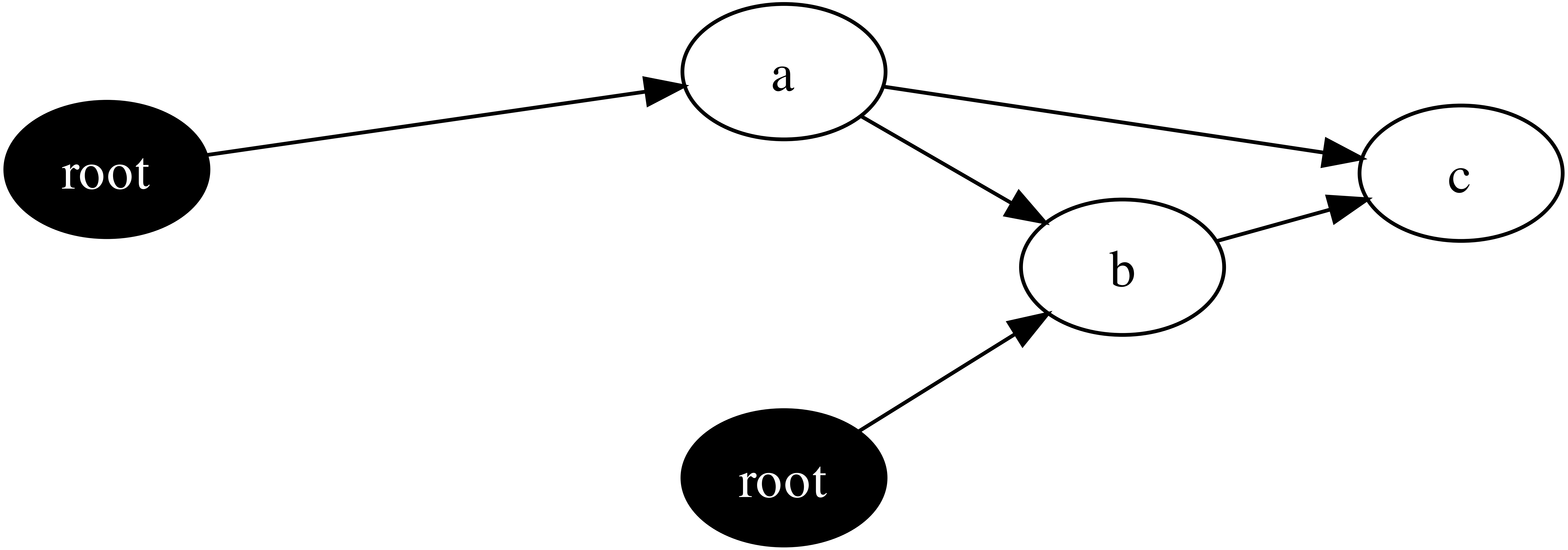
If the reference to D goes away...



We can no longer reach D or E.



So the collector reclaims them.



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)

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()`**
  - References (and reference queues)

Try `finally` first.

- Pros:
  - More straightforward
  - Handles exceptions in main thread
  - Ensures cleanup keeps pace
- Cons:
  - More work for programmers
  - More error prone
  - Cleanup happens in main thread
- ARM will help.

What is a finalizer?

A callback used by the garbage collector to notify an object when it is about to be reclaimed:

```
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.
- Undefined threading model; they can run concurrently!
- You must remember to call **`super.finalize()`**.
- Exceptions are ignored (per spec).
- You can resurrect references.
- They keep objects alive longer than necessary.
- They can make allocation/reclamation 430X slower!

  (Bloch, *Effective Java*)
- Worst of all, they messed up the reference API.

Example

```
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 **`NativeMemory`**'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());
  }
}
```

Always 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);
}
```

Basically, finalizers are good for one thing.

Logging warnings:

```
public class Connection {
  ...
  boolean closed;
  public synchronized void close() {
    reallyClose();
    closed = true;
  }
  private native void reallyClose();

  @Override protected synchronized void finalize() {
    if (!closed) {
      Logger.getLogger(Connection.class.getName())
          .warning("You forgot to close me!!!");
      close();
    }
  }
}
```

Basically, finalizers are good for one thing.

Logging warnings:

```
public class Connection {
  ...
  boolean closed;
  public synchronized void close() {
    reallyClose();
    closed = true;
  }
  private native void reallyClose();

  @Override protected synchronized void finalize() {
    if (!closed) {
      Logger.getLogger(Connection.class.getName())
          .warning("You forgot to close me!!!");
      close();
    }
  }
}
```

Unless you want to disable the warnings.

The alternative: The Reference API

- `@since 1.2`
- Reference types
  - **Soft:** for caching
  - **Weak:** for fast cleanup (pre-finalizer)
  - **Phantom:** for safe cleanup (post-finalizer)
- **Reference queues:** for notifications

package java.lang.ref

```
public abstract class Reference<T> {
  public T get() { ... }
}

public class SoftReference<T> extends Reference<T> {
  public SoftReference(T referent) { ... }
  public SoftReference(T referent, ReferenceQueue<? super T> q) { ... }
}

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

public class PhantomReference<T> extends Reference<T> {
  public PhantomReference(T referent, ReferenceQueue<? super T> q) { ... }
}

public class ReferenceQueue<T> {
  public ReferenceQueue() { ... }
  public Reference<? extends T> poll() { ... }
  public Reference<? extends T> remove() { ... }
}
```

Soft references

- Cleared when the VM runs low on memory
  - *Hopefully* in LRU fashion
- Tuned with **`-XX:SoftRefLRUPolicyMSPerMB`**
  - How long to retain soft refs in *ms per free MB of heap*
  - **Default:** 1000ms

Use soft references judiciously.

- For quick-and-dirty caching only
- Soft refs have no notion of *weight*:
  - Memory usage
  - Computation time
  - CPU usage
- Soft refs can exacerbate low memory conditions.

Caching a file

```
public class CachedFile {
  final File file;
  public CachedFile(File file) {
    this.file = file;
  }
  volatile SoftReference<byte[]> dataReference
      = new SoftReference<byte[]>(null);
  /** Gets file contents, reading them if necessary. */
  public byte[] getData() {
    byte[] data = dataReference.get();
    if (data != null) return data;
    data = readData();
    dataReference = new SoftReference<byte[]>(data);
    return data;
  }
  /** Reads file contents. */
  byte[] readData() {
    ...
  }
}
```

Weak references

- Cleared as soon as no strong or soft refs remain.
- Cleared ASAP, before the finalizer runs.
- **Not for caching!** Use soft references, as intended:

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

- The **`SoftReference`** documentation

Can you hear me now?

```
public class Button {
  public interface Listener {
    void onClick();
  }
  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();
    }
  }
}
```

Phantom references

- Enqueued after no other references remain, *post-finalizer*.
  - Can suffer similar problems to finalizers.
- Must be cleared manually, for no good reason.
- **`get()`** always returns **`null`**.
  - So you must use a reference queue.

Let's replace a finalizer!

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

  /** 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);
}
```

The reference

```
class NativeMemoryReference
    extends PhantomReference<NativeMemory> {
  final int address;
  NativeMemoryReference(NativeMemory referent,
      ReferenceQueue<NativeMemory> rq) {
    super(referent, rq);
    address = referent.address;
  }
}
```

The manager

```
public class NativeMemoryManager {
  private static final Set<Reference<?>> refs
      = Collections.synchronizedSet(new HashSet<Reference<?>>());
  private static final ReferenceQueue<NativeMemory> rq
      = new ReferenceQueue<NativeMemory>();
  public static NativeMemory allocate() {
    NativeMemory nm = new NativeMemory();
    refs.add(new NativeMemoryReference(nm, rq));
    cleanUp();
    return nm;
  }
  private static void cleanUp() {
    NativeMemoryReference ref;
    while ((ref = (NativeMemoryReference) rq.poll()) != null) {
      NativeMemory.free(ref.address);
      refs.remove(ref);
    }
  }
}
```

The manager *with Google Collections*

```
public class NativeMemoryManager {
  private static final Set<Reference<?>> refs
      = Collections.synchronizedSet(new HashSet<Reference<?>>());
  private static final FinalizableReferenceQueue frq
      = new FinalizableReferenceQueue();
  public static NativeMemory allocate() {
    NativeMemory nm = new NativeMemory();
    final int address = nm.address;
    refs.add(new FinalizablePhantomReference<NativeMemory>(nm, frq) {
      public void finalizeReferent() {
        NativeMemory.free(address);
        refs.remove(this);
      }
    });
    return nm;
  }
}
```

Tip: **accessing a phantom referent**

```
public class WeakPhantomReference<T> extends PhantomReference<T> {
  final WeakReference<T> weakReference;

  public WeakPhantomReference(T referent,
      ReferenceQueue<? super T> q) {
    super(referent, q);
    weakReference = new WeakReference<T>(referent);
  }

  /** Returns referent so long as it's weakly-reachable. */
  @Override public T get() {
    return weakReference.get();
  }
}
```

Don't forget...

The GC runs concurrently with your code:

```
public class RaceTheCollector {
  public <T> T dereference(WeakReference<T> referent) {
    T t = referent.get();
    if (t == null) {
      throw new NullPointerException("Reference is cleared.");
    }
    ... // The garbage collector runs.
    return referent.get(); // Can return null!!!
  }
}
```

java.util.WeakHashMap

- Useful for emulating additional fields
- Keeps weak refs to keys, strong refs to values
- Not concurrent
- Uses **equals()** when it should use **==**

Google Collections `MapMaker`

- Near drop-in replacement for **`WeakHashMap`**
- Strong, soft, or weak key and/or value references
- Concurrent, cleans up in background thread
- Uses **`==`** to compare weak and soft referents
- Supports on-demand computation of values

Google Collections `MapMaker`

```
public class GetterMethods {
  final static Map<Class<?>, List<Method>> cache = new MapMaker()
      .weakKeys()
      .softValues()
      .makeComputingMap(new Function<Class<?>, List<Method>>() {
        public List<Method> apply(Class<?> clazz) {
          List<Method> getters = new ArrayList<Method>();
          for (Method m : clazz.getMethods())
            if (m.getName().startsWith("get"))
               getters.add(m);
          return getters;
        }
      });
  public static List<Method> on(Class<?> clazz) {
    return cache.get(clazz);
  }
}
```

Usage: `List<Method> l = GetterMethods.on(Foo.class);`

Recap: The Levels of Reachability

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

Recap: The Levels of Reachability

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

Recap: The Levels of Reachability

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

Recap: The Levels of Reachability

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

Recap: The Levels of Reachability

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

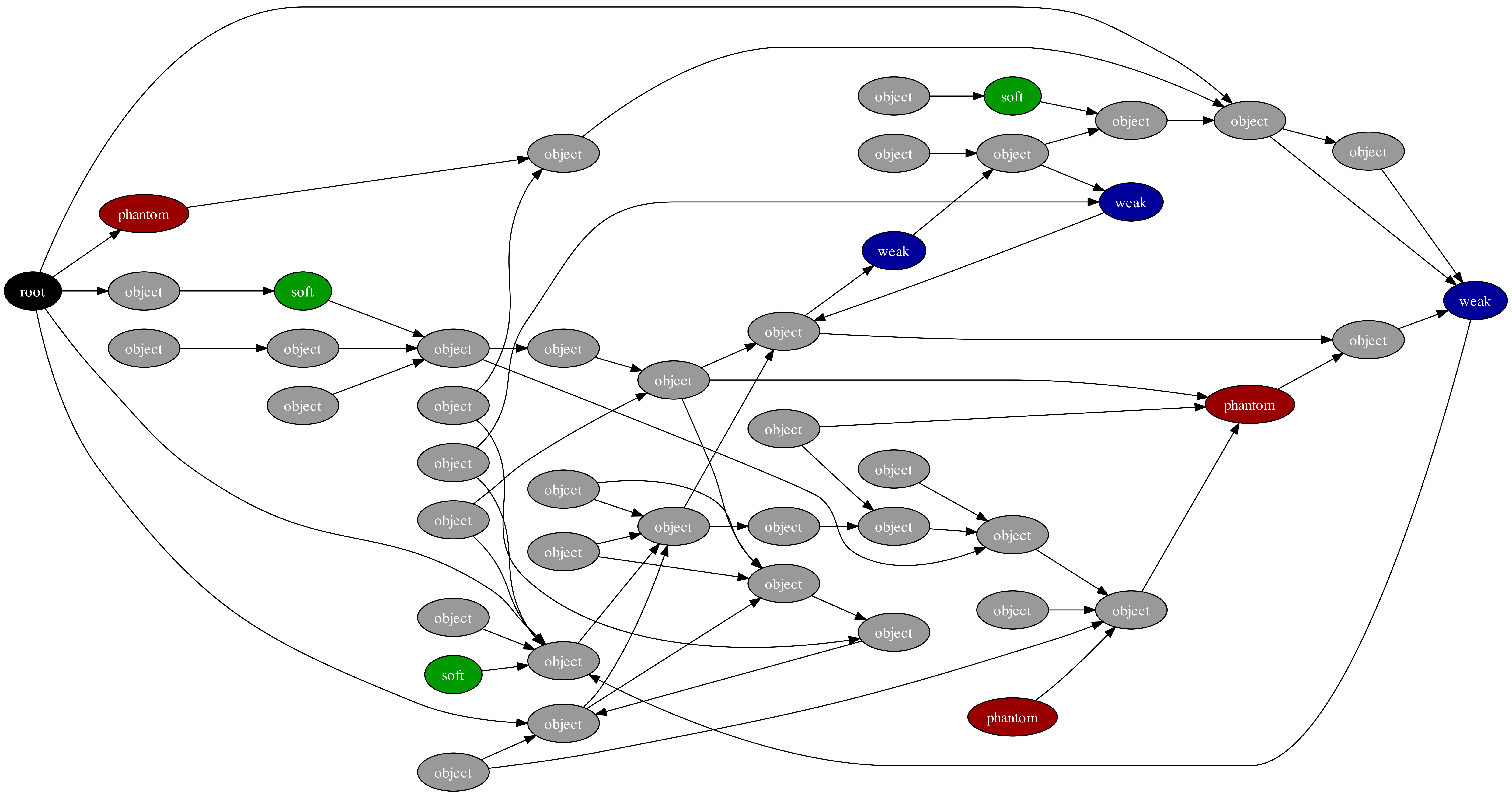
Recap: The Levels of Reachability

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

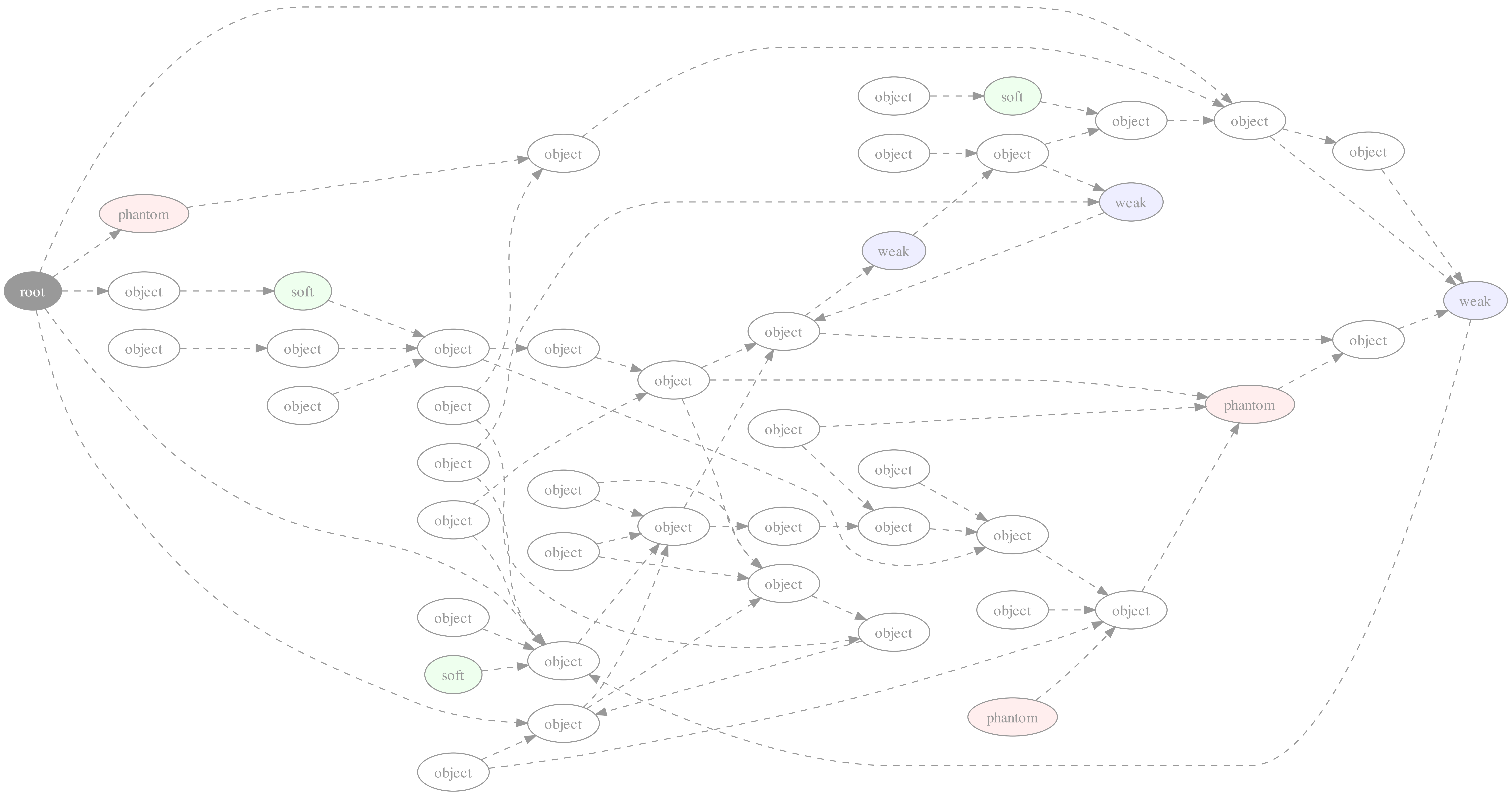
Recap: The Levels of Reachability

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

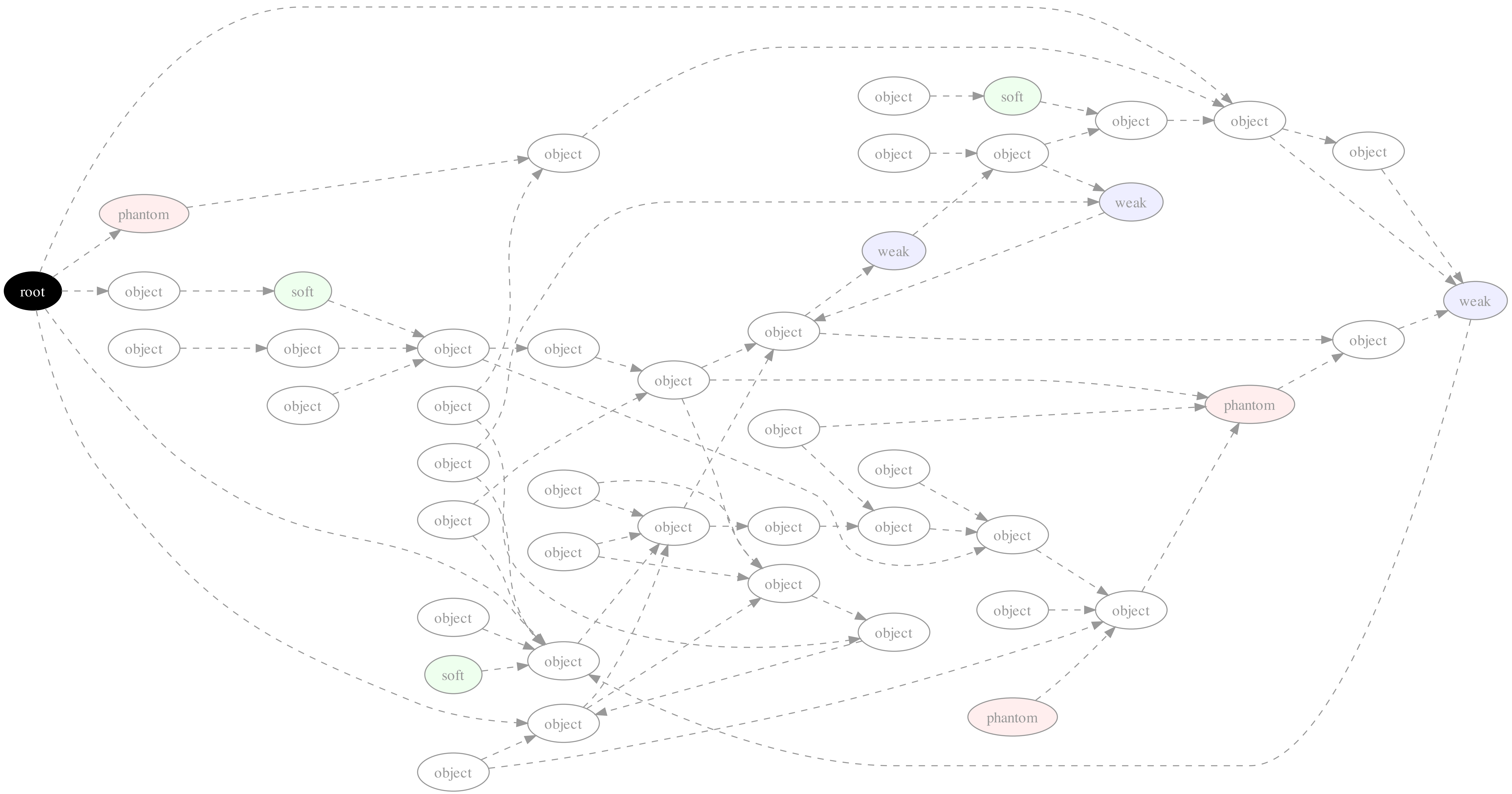
Let's mark and sweep a heap!



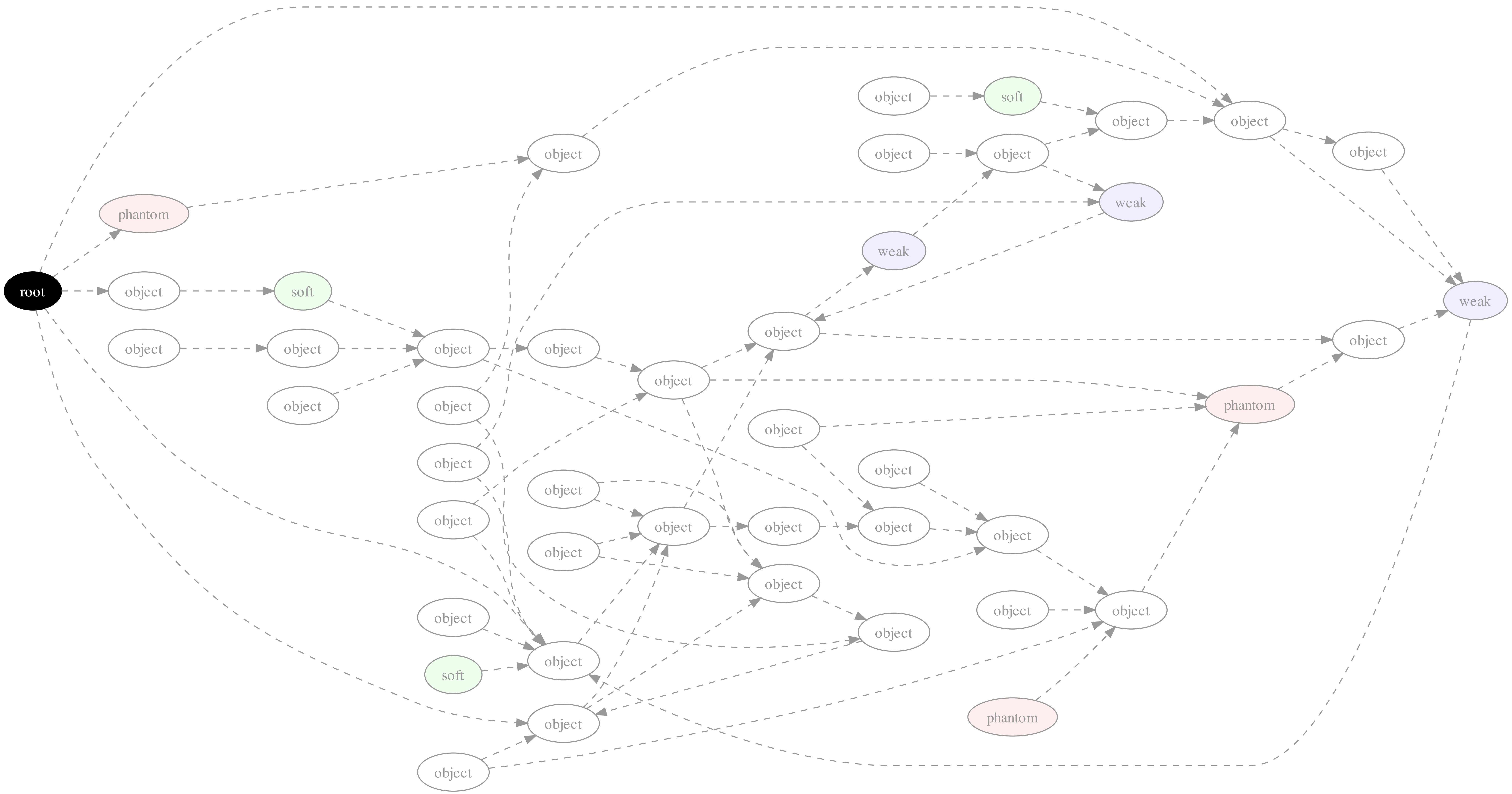
No objects are marked at first.



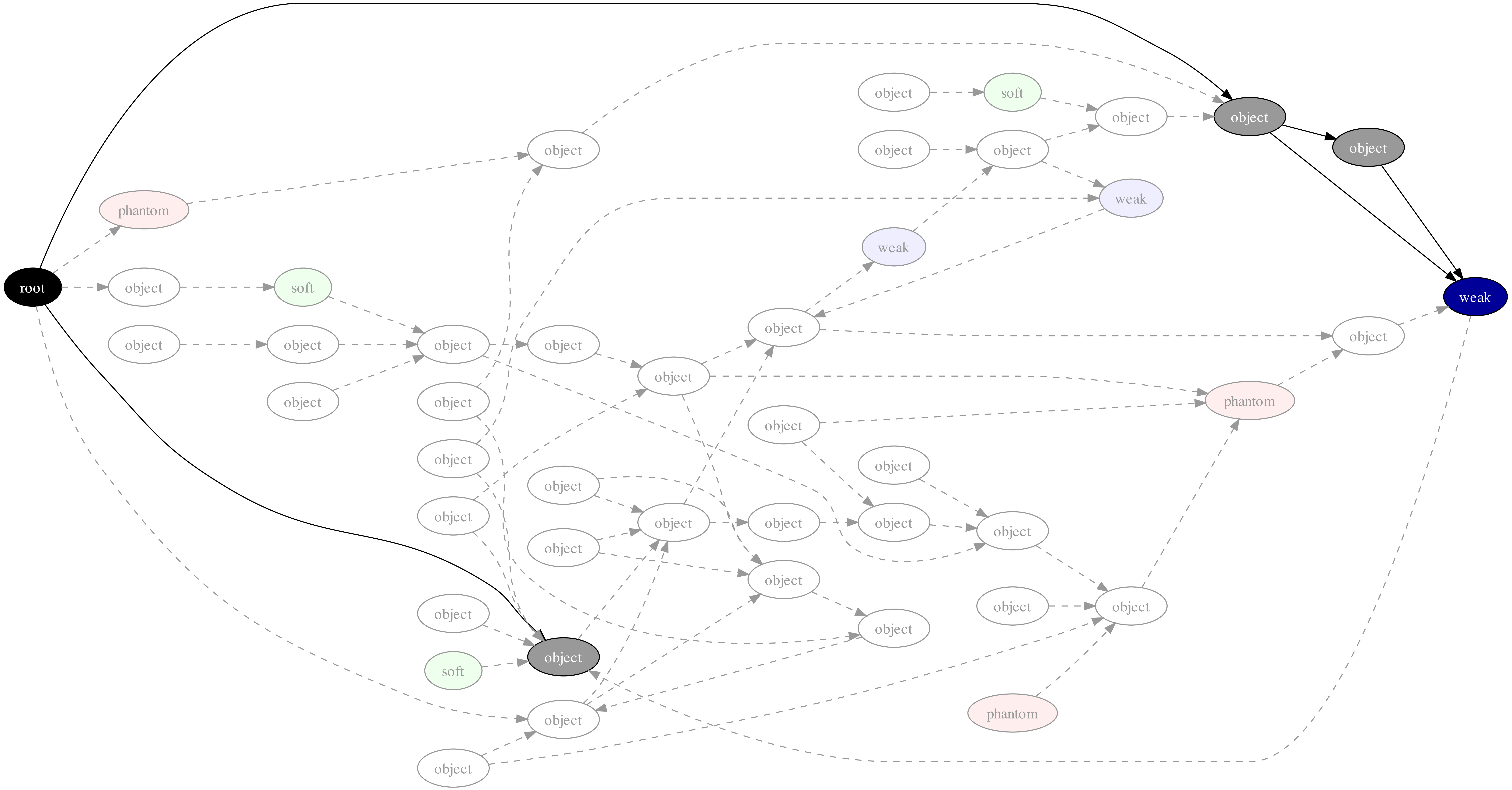
1. **Start at a root.**



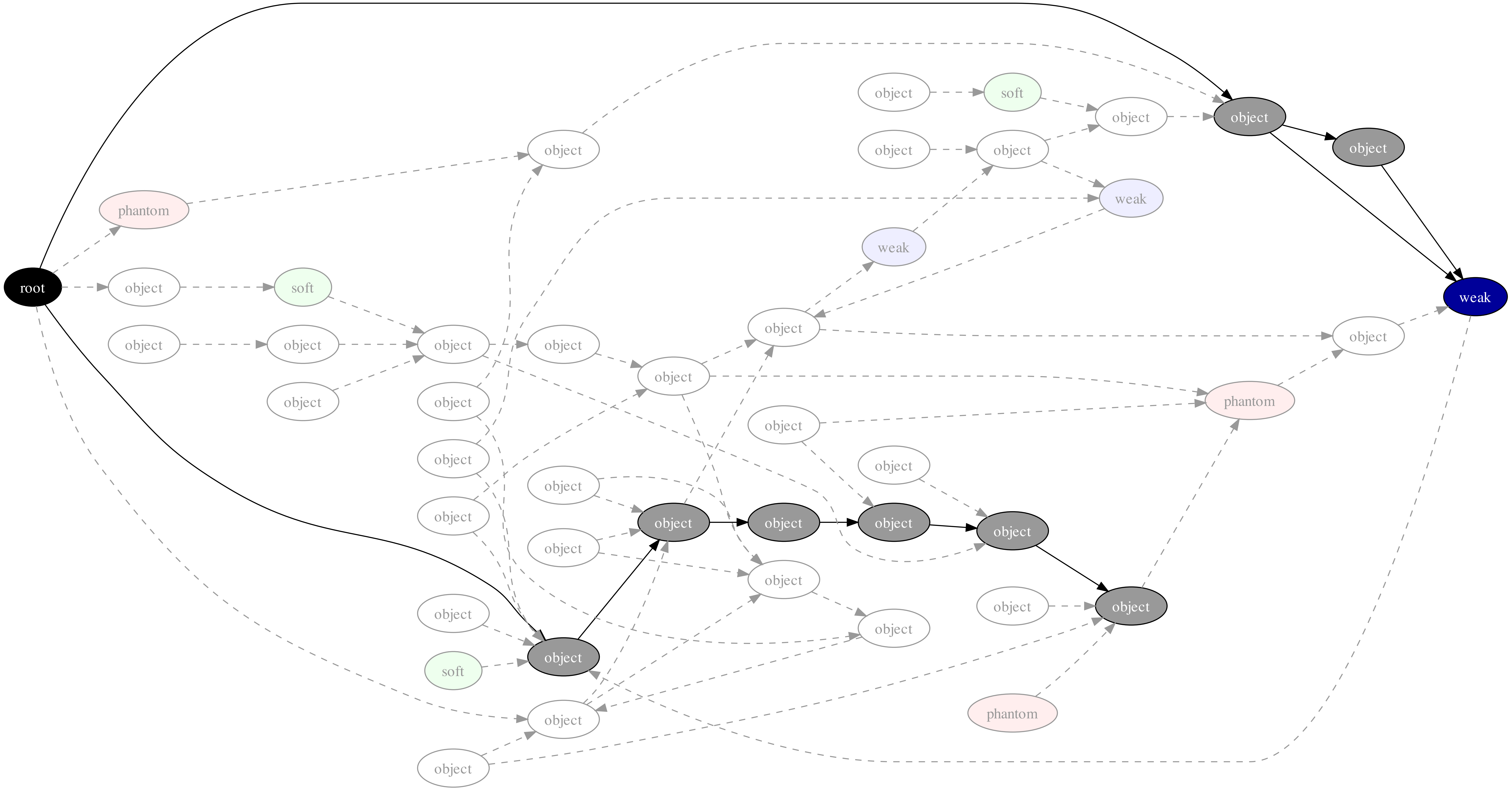
2. **Trace and mark strongly-referenced objects.**



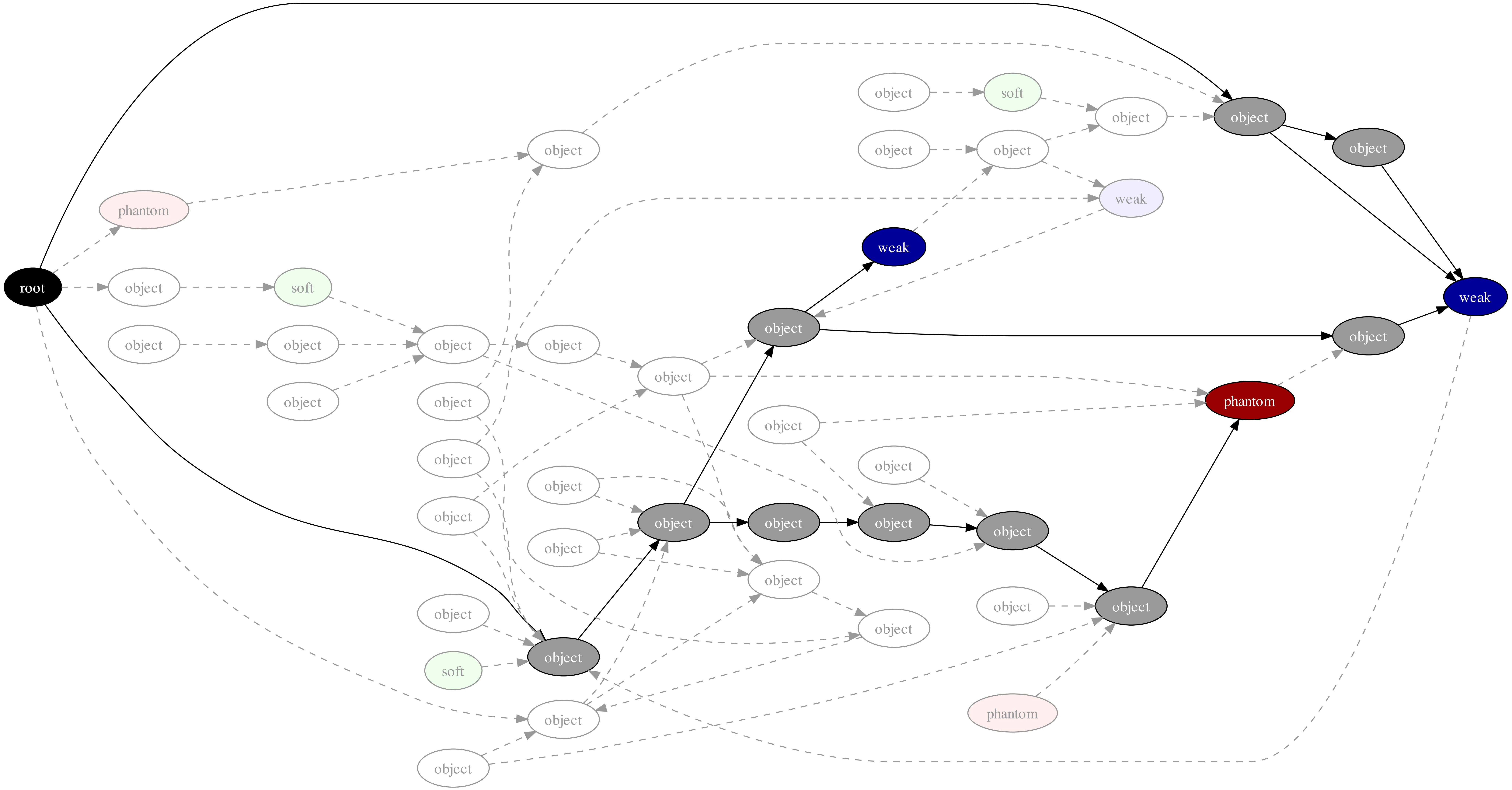
2. **Trace and mark strongly-referenced objects.**



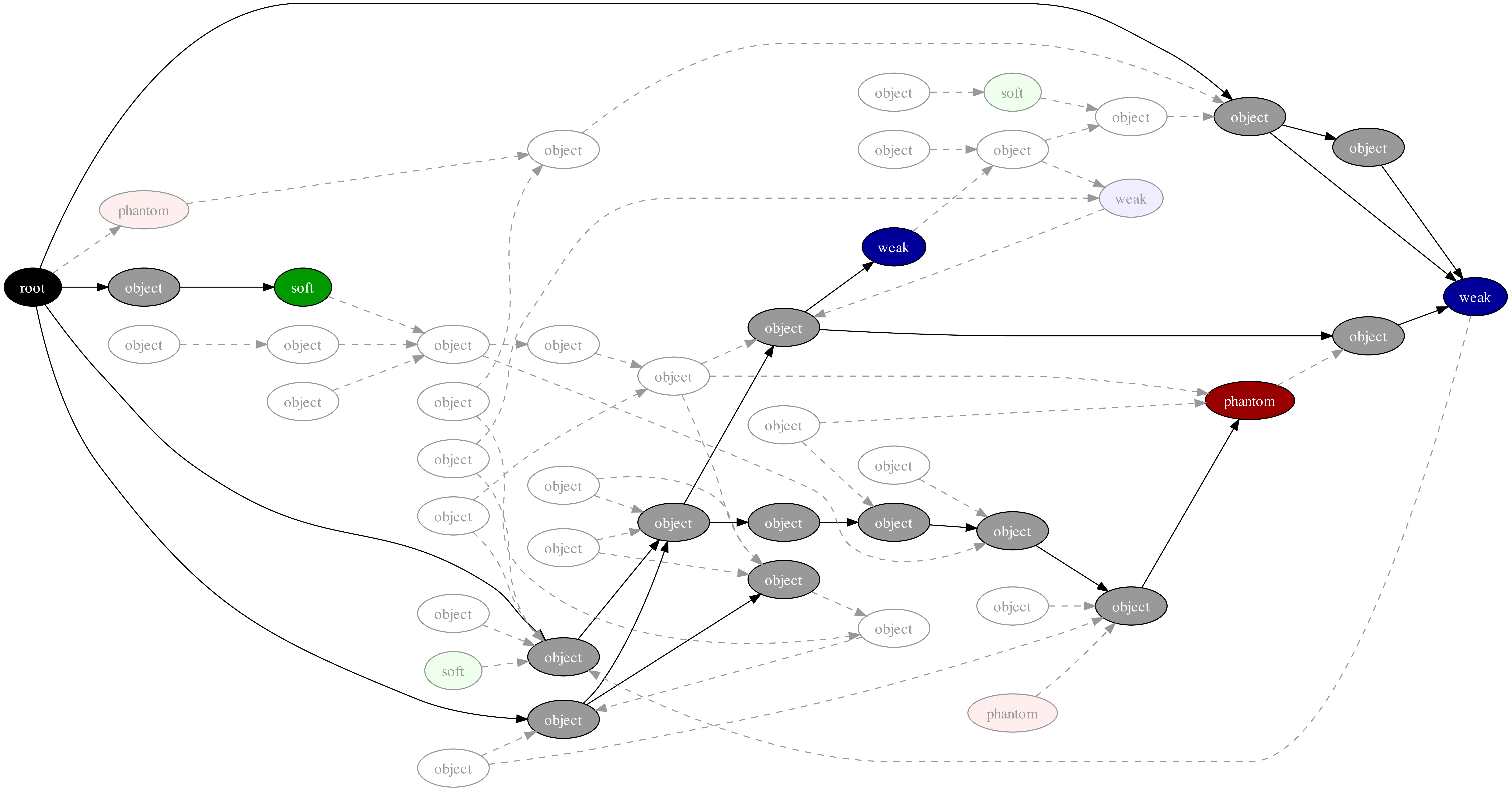
2. **Trace and mark strongly-referenced objects.**



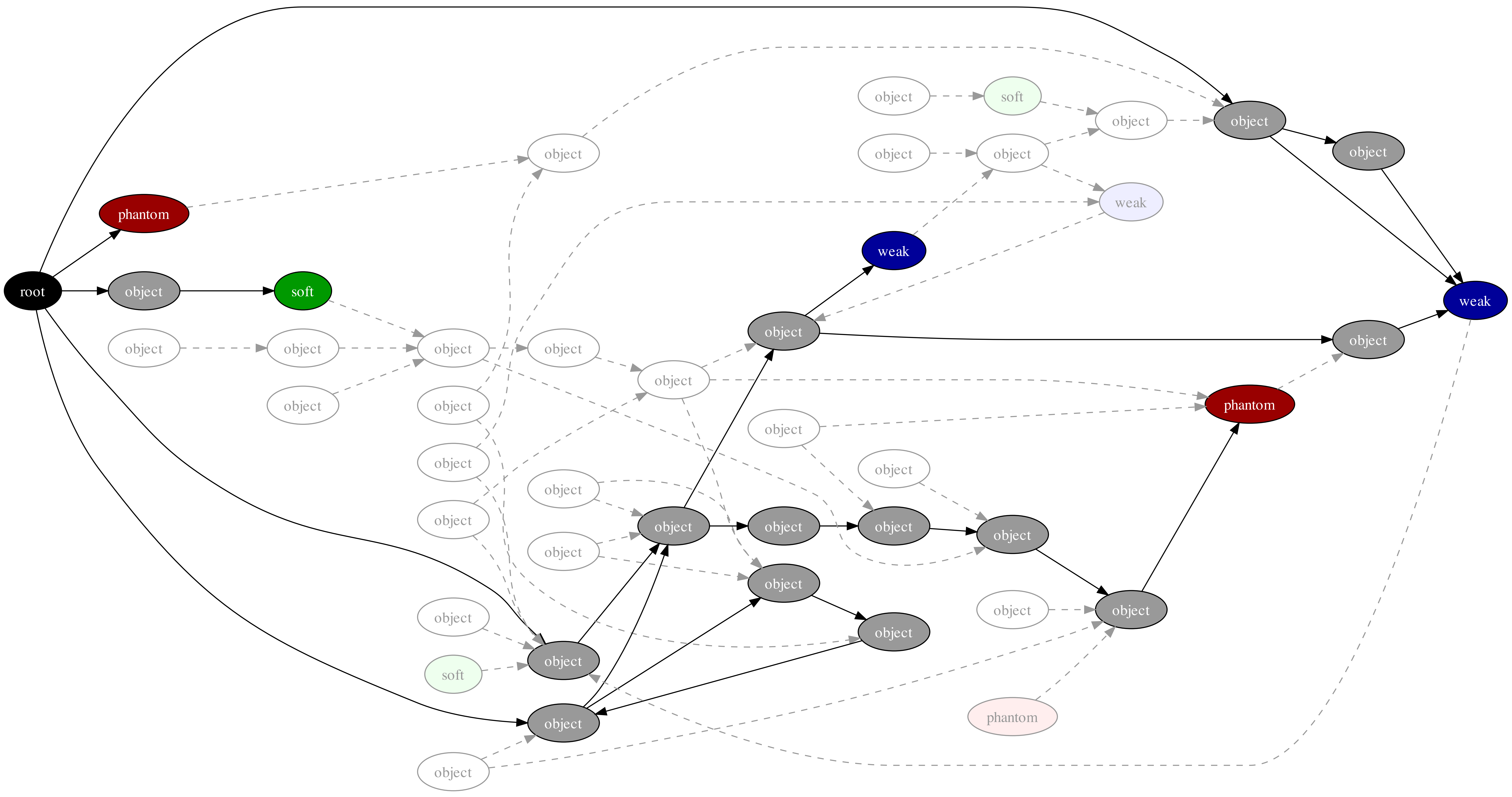
2. **Trace and mark strongly-referenced objects.**



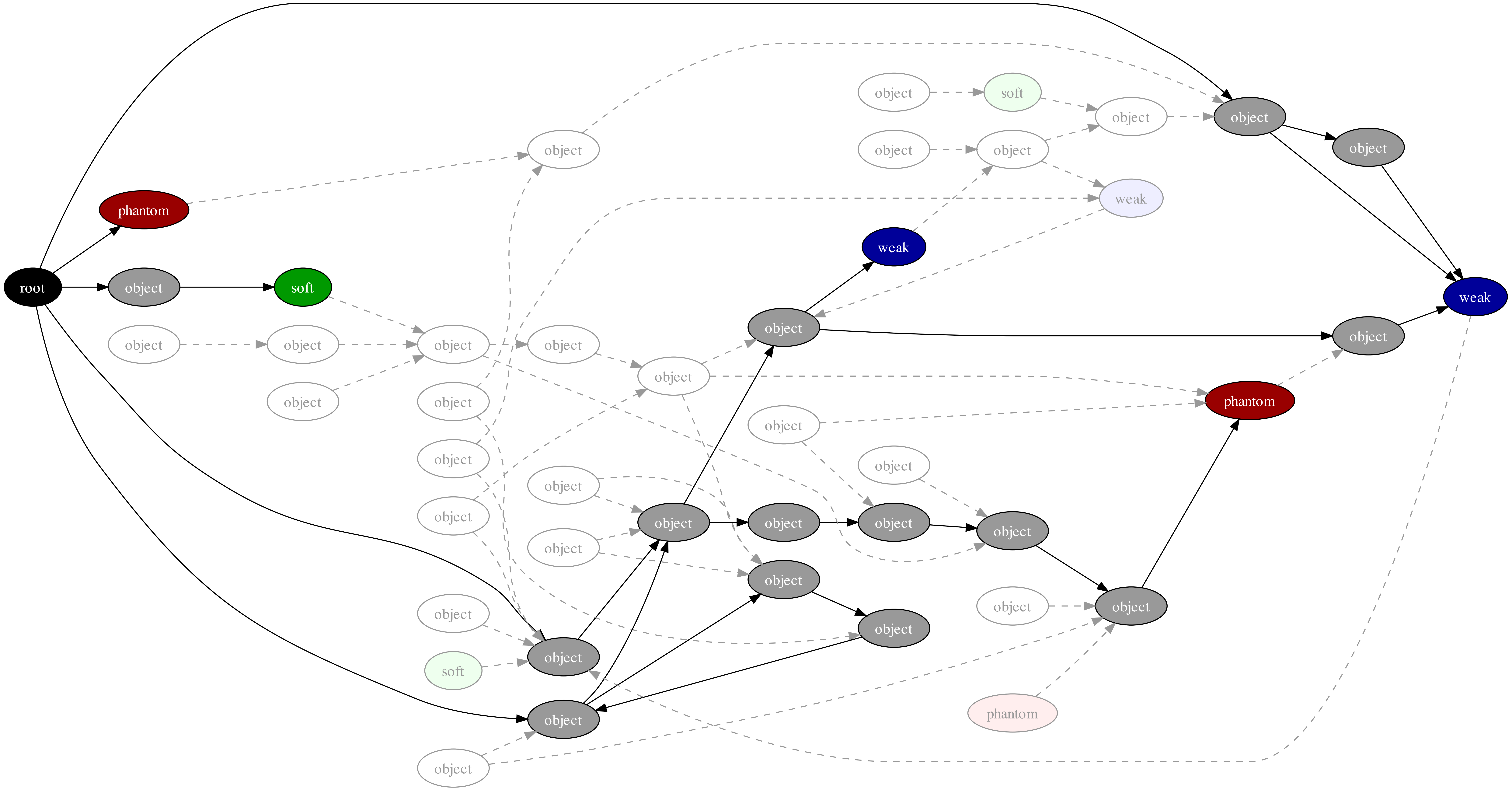
2. **Trace and mark strongly-referenced objects.**



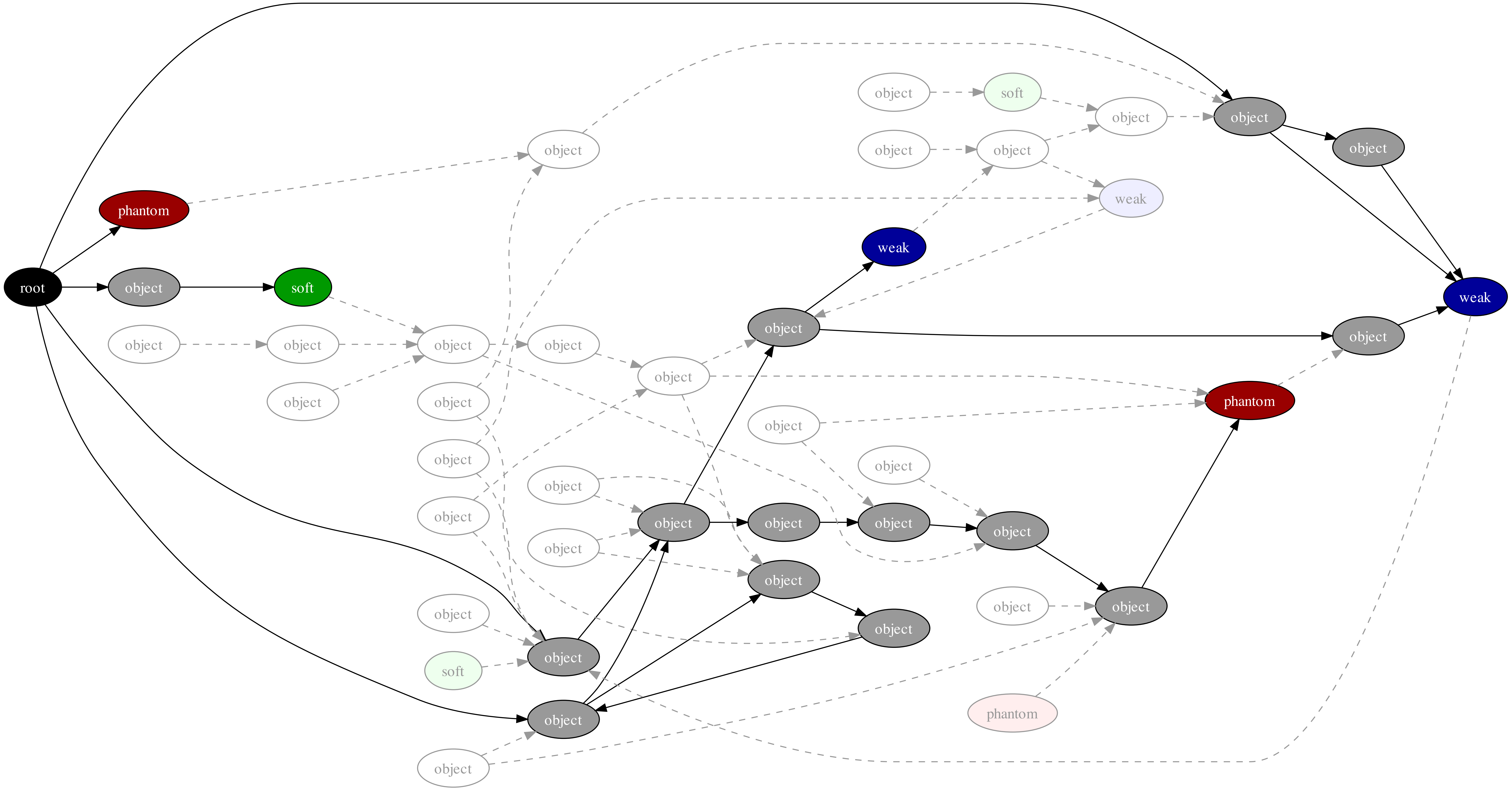
2. **Trace and mark strongly-referenced objects.**



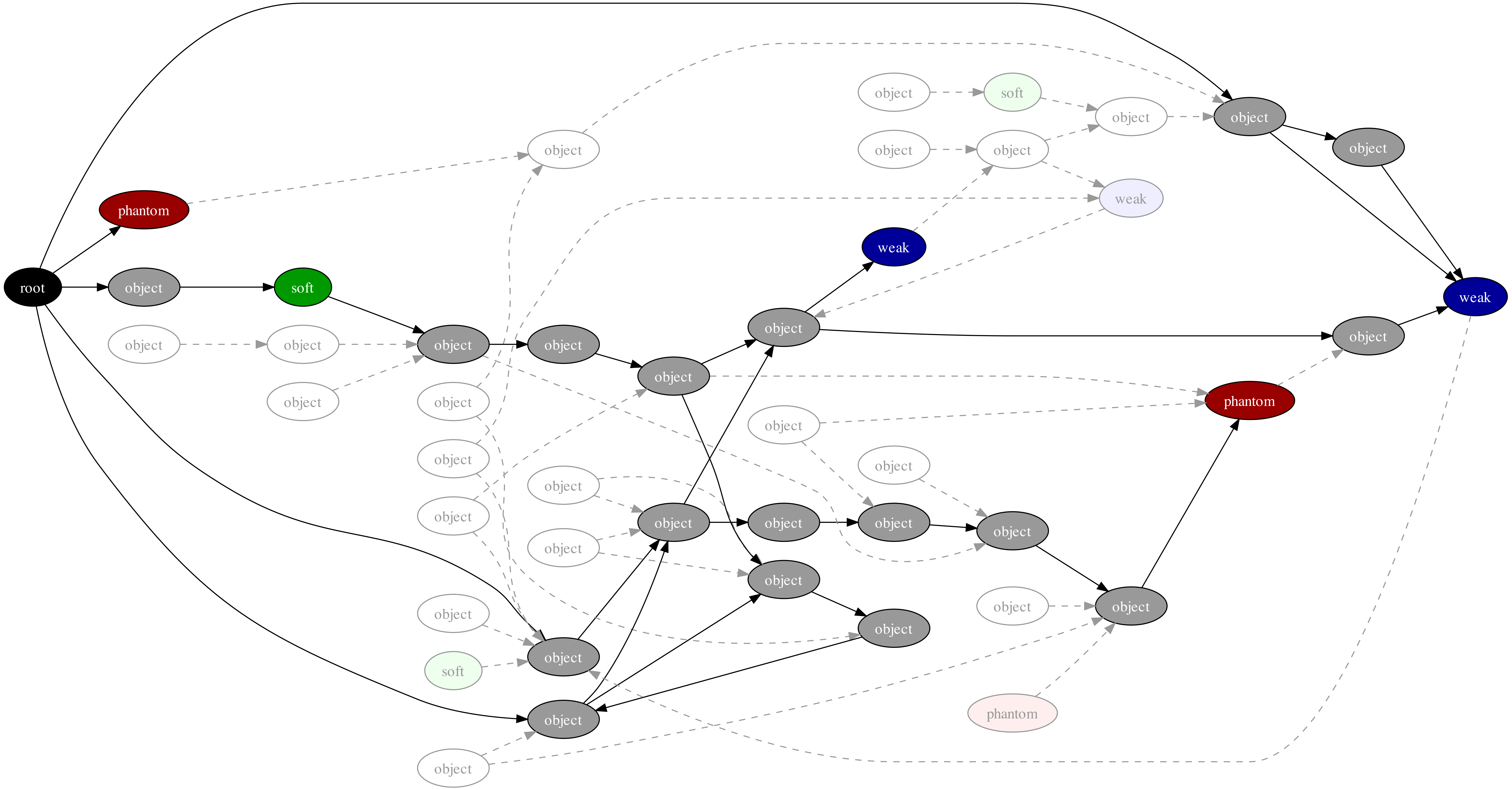
3. Optionally clear soft references.



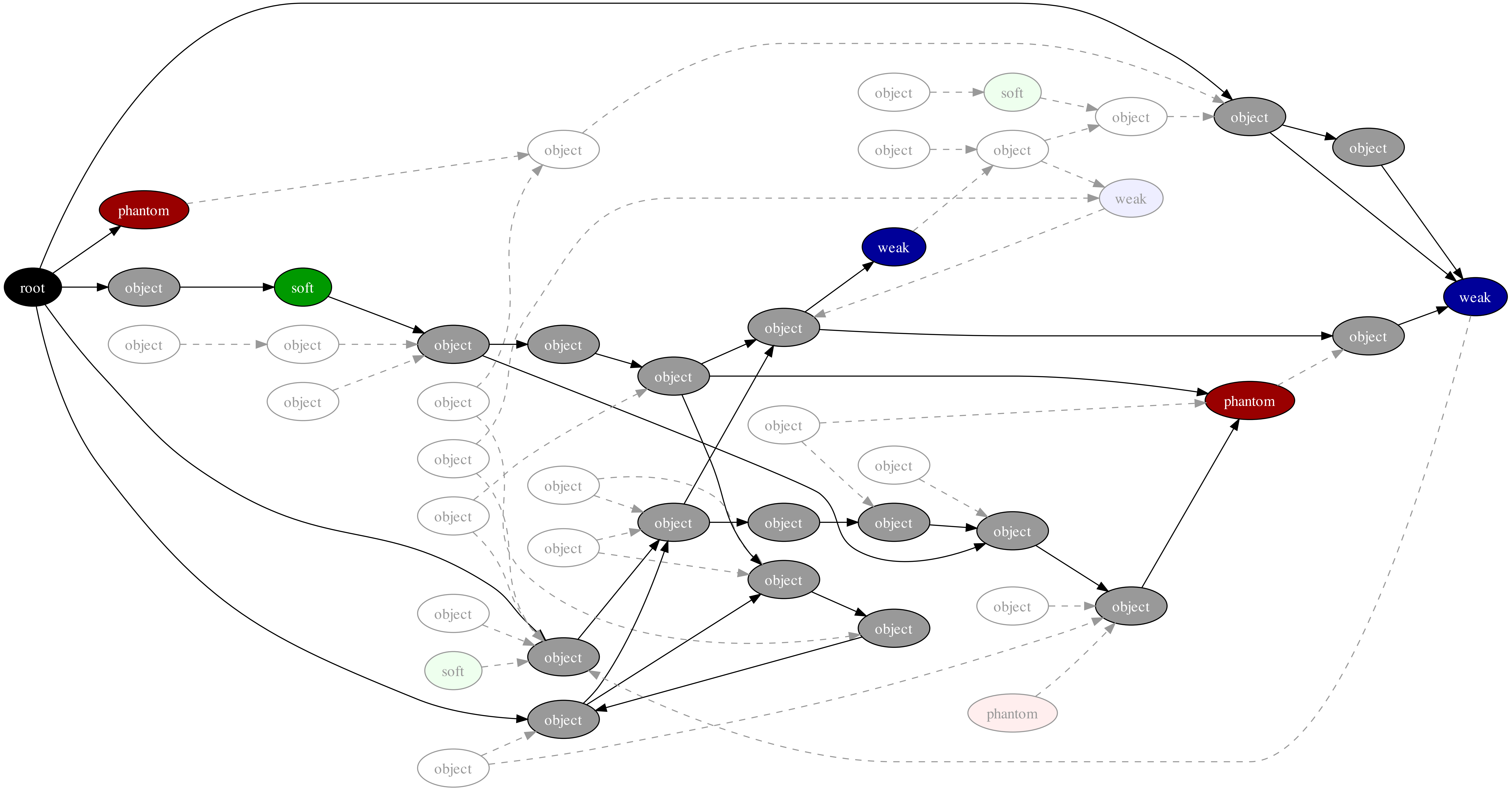
4. **Trace and mark softly-referenced objects.**



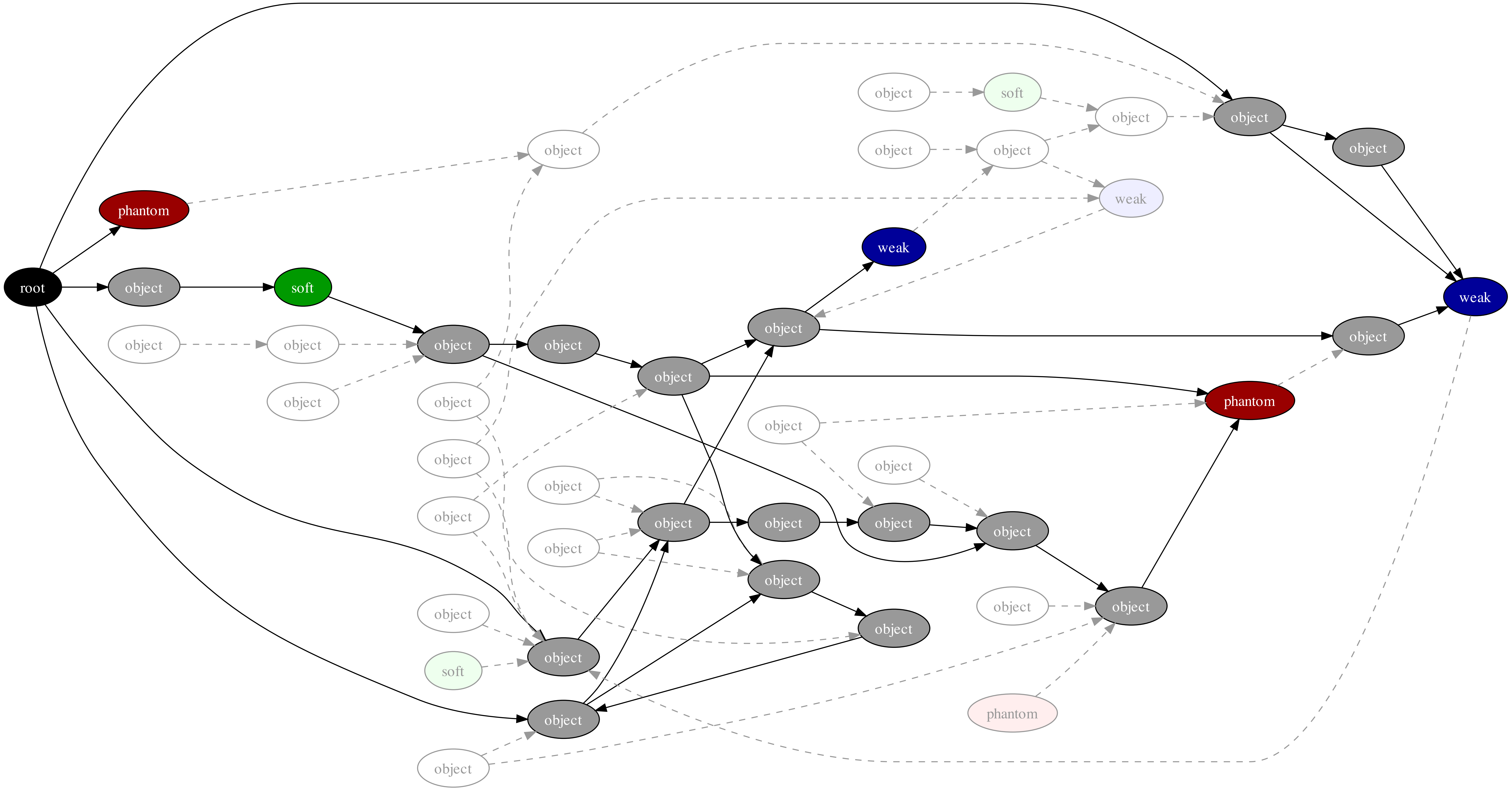
4. Trace and mark softly-referenced objects.



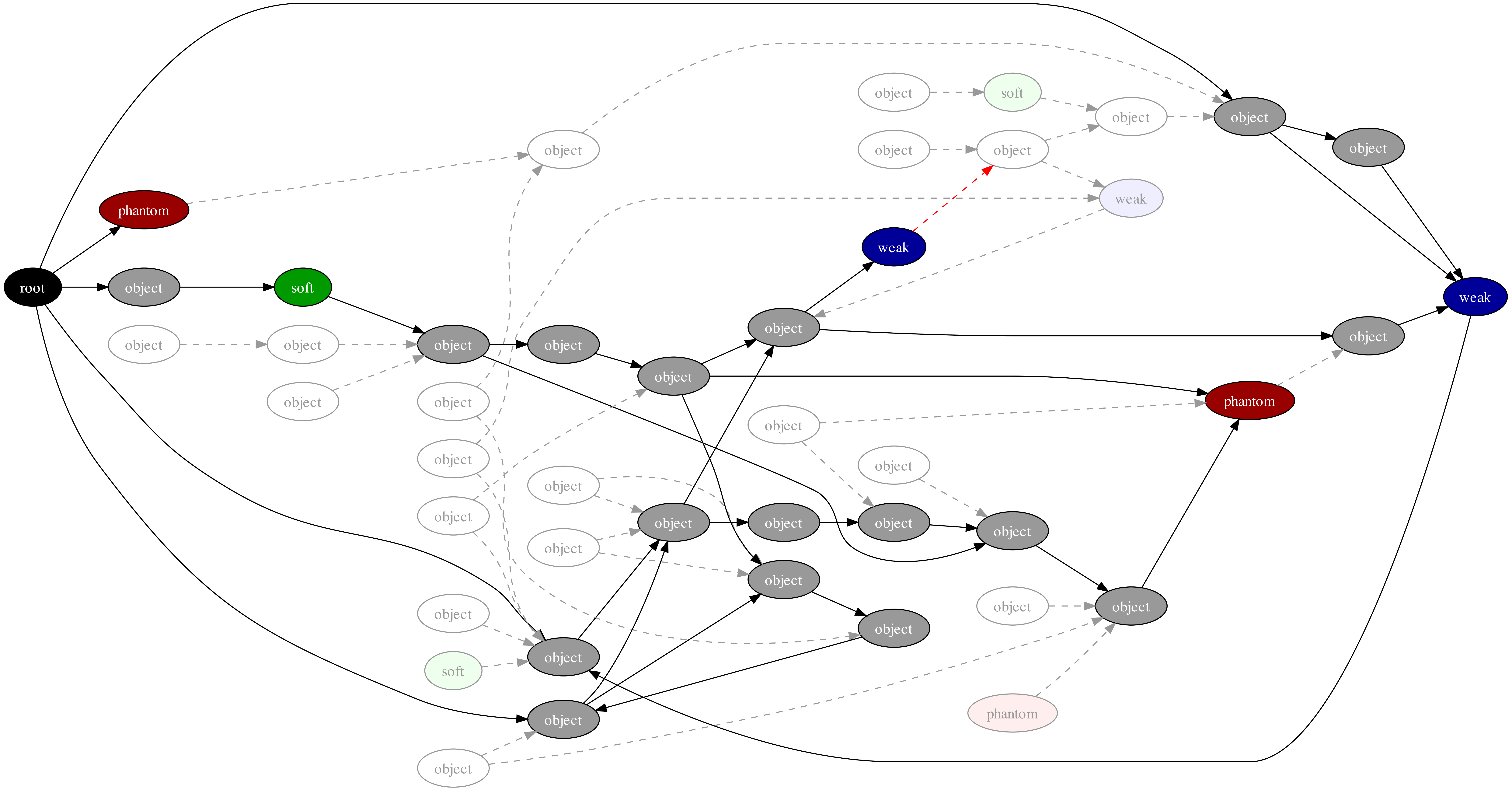
4. Trace and mark softly-referenced objects.



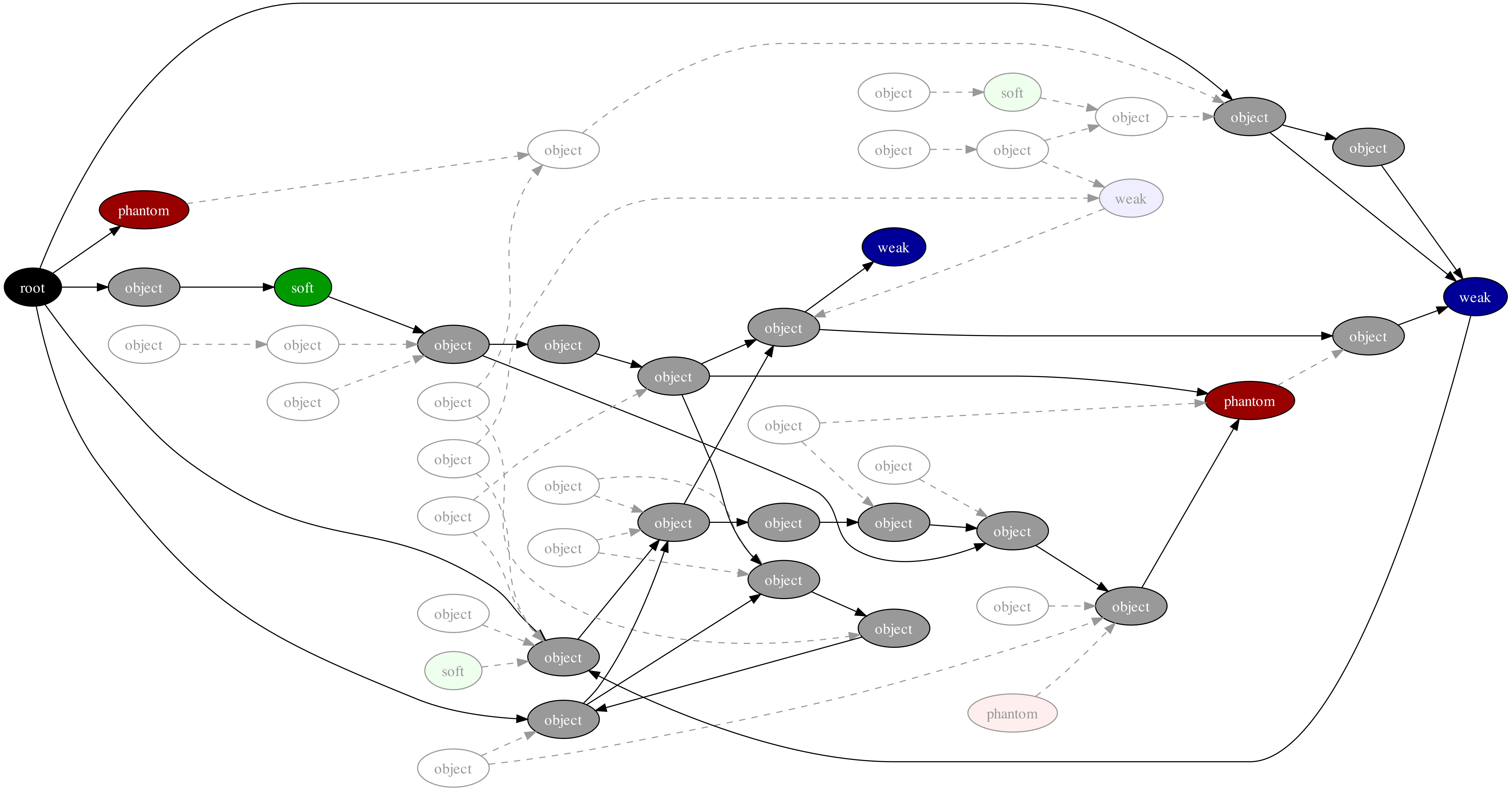
5. Clear weak references.



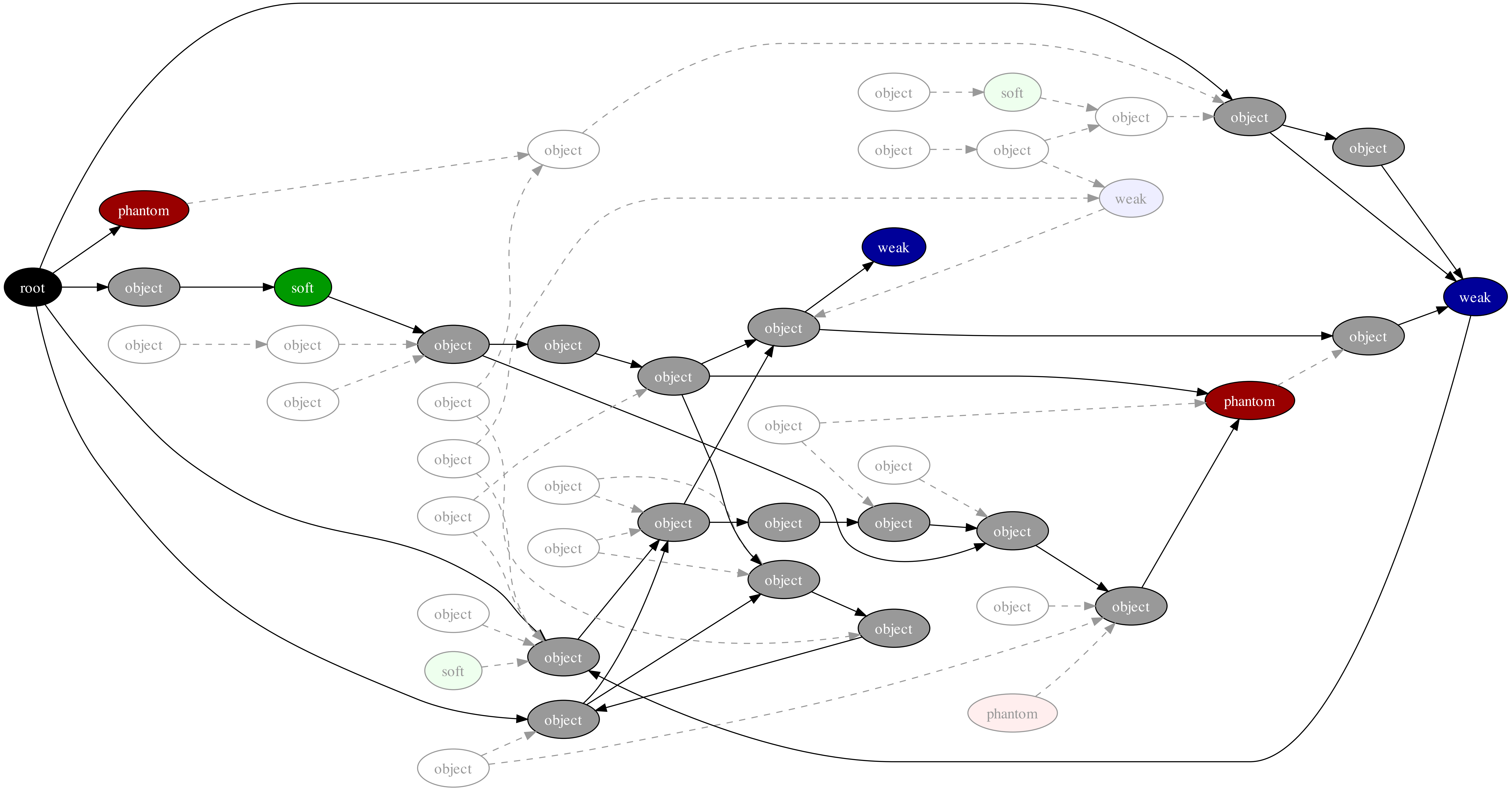
5. Clear weak references.



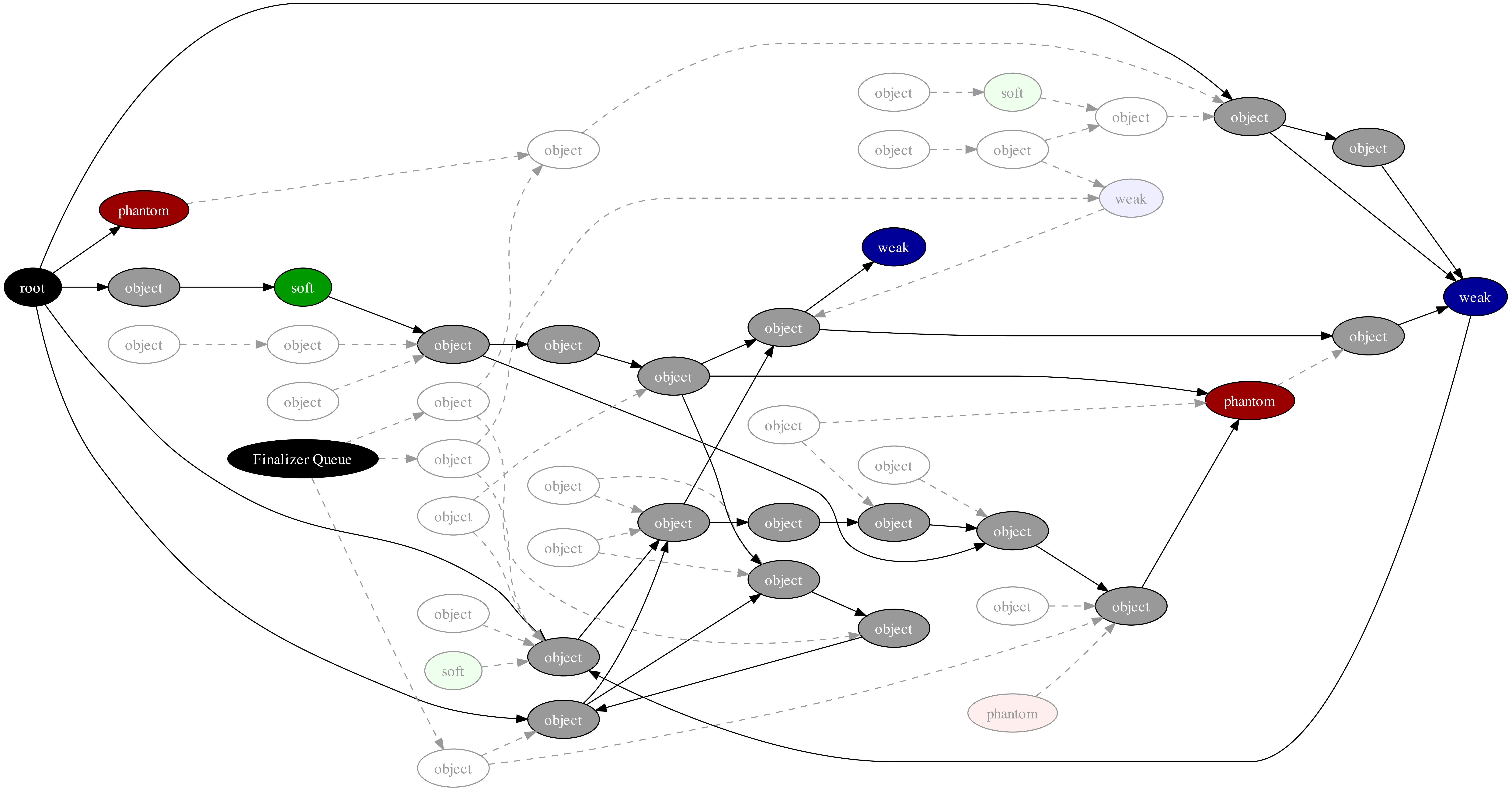
5. Clear weak references.



6. **Enqueue finalizable objects.**

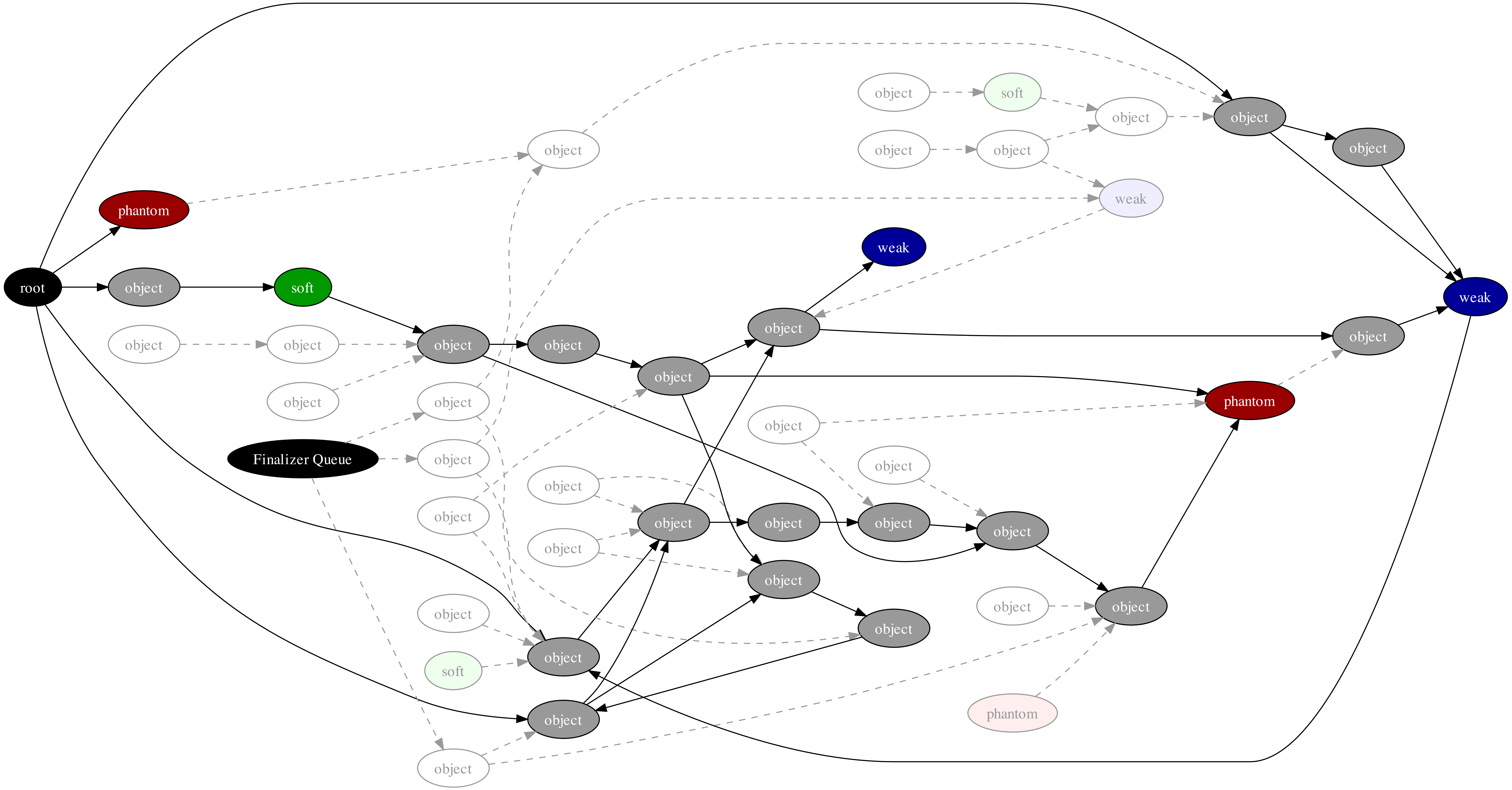


6. **Enqueue finalizable objects.**

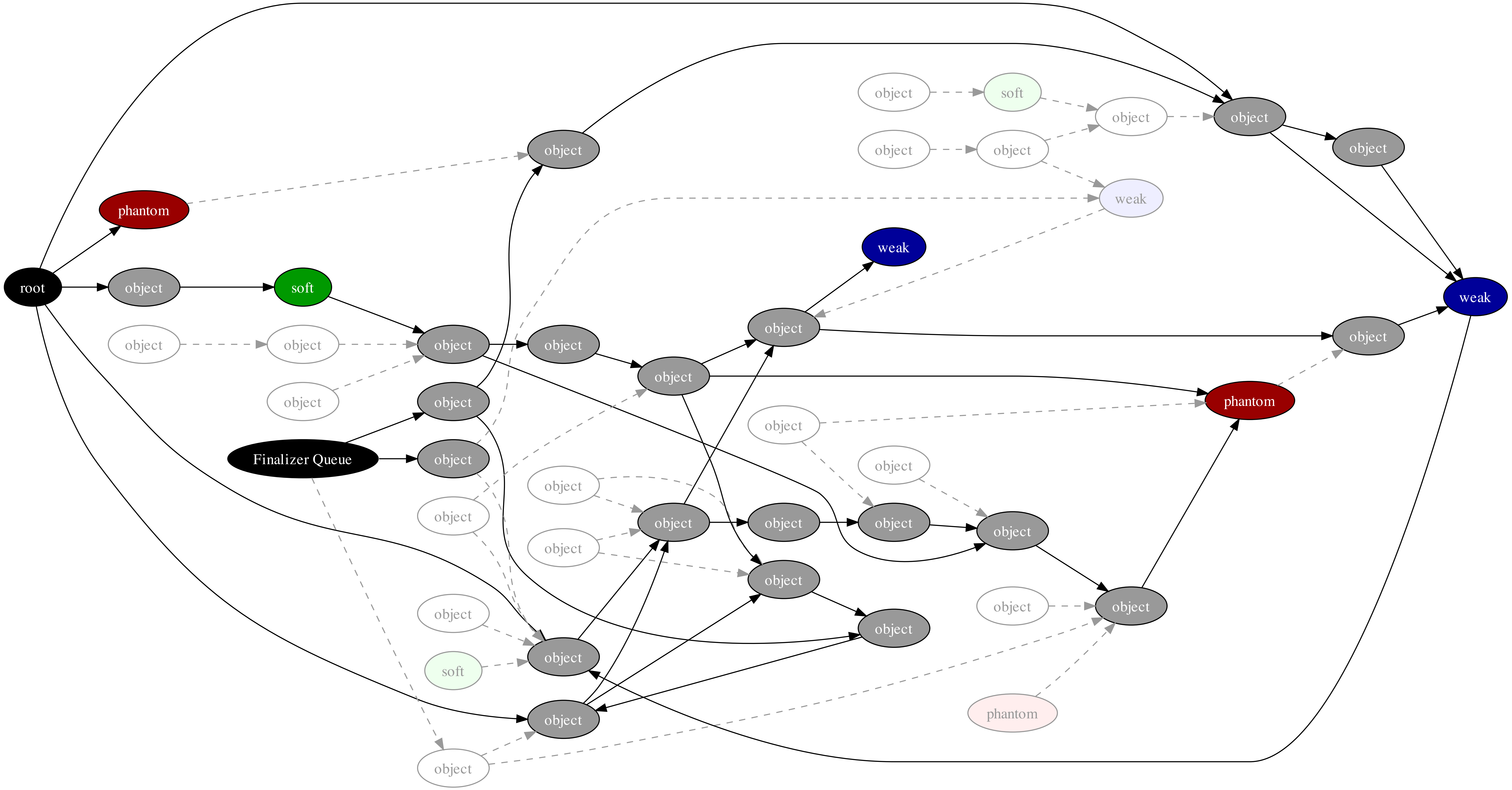


open source convention

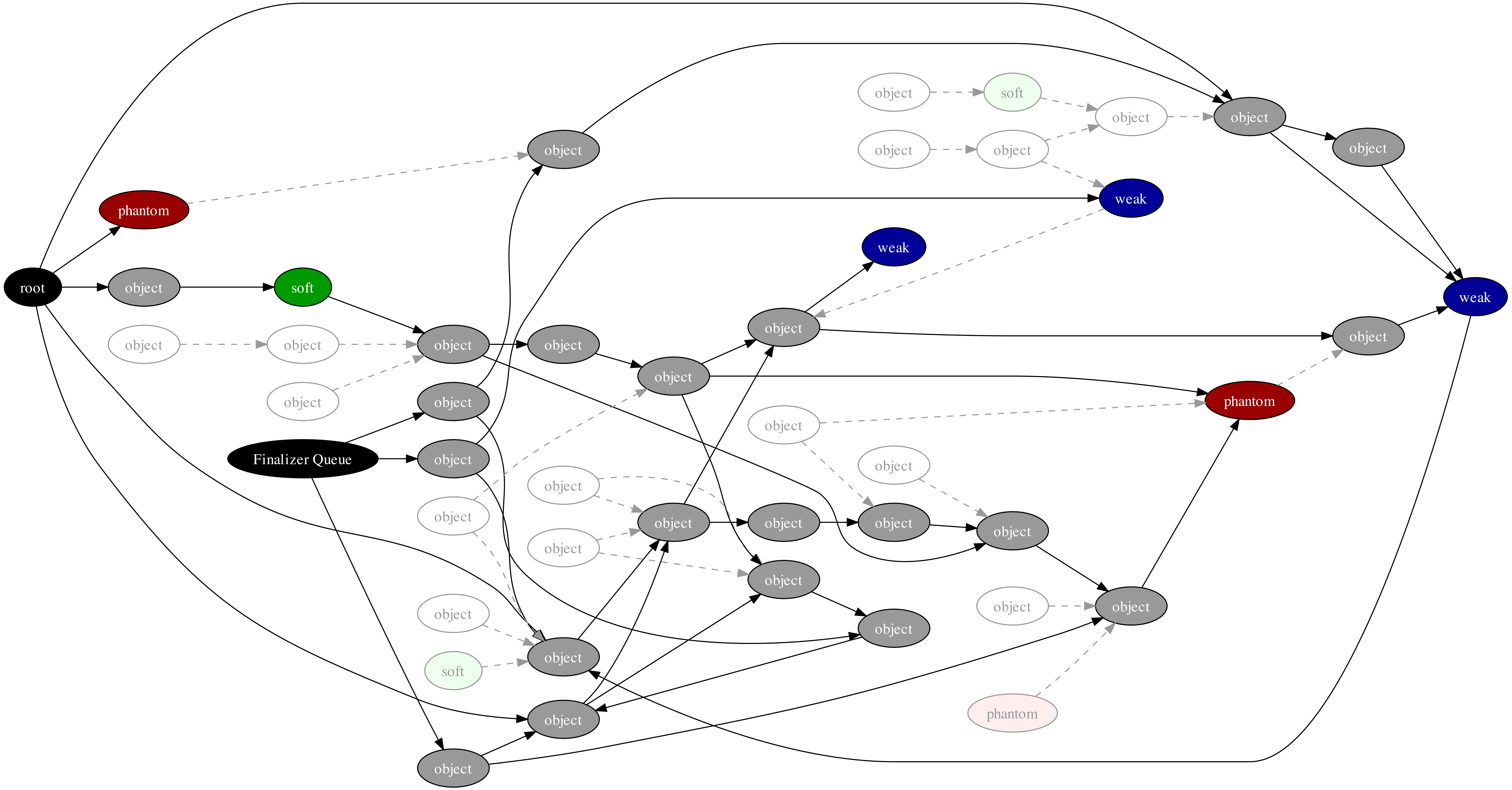
7. Repeat steps 1 through 5 for the queue.



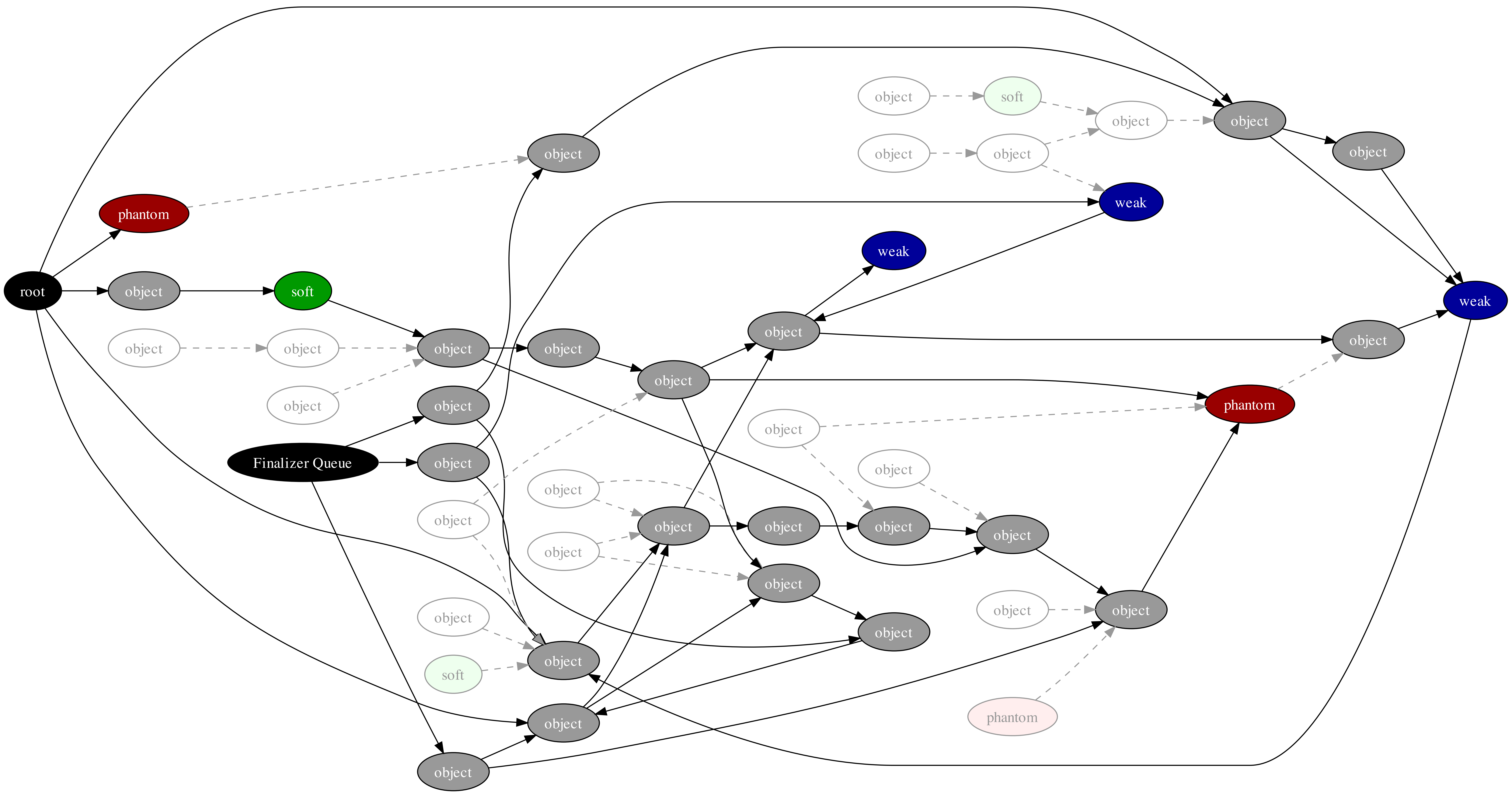
7. Repeat steps 1 through 5 for the queue.



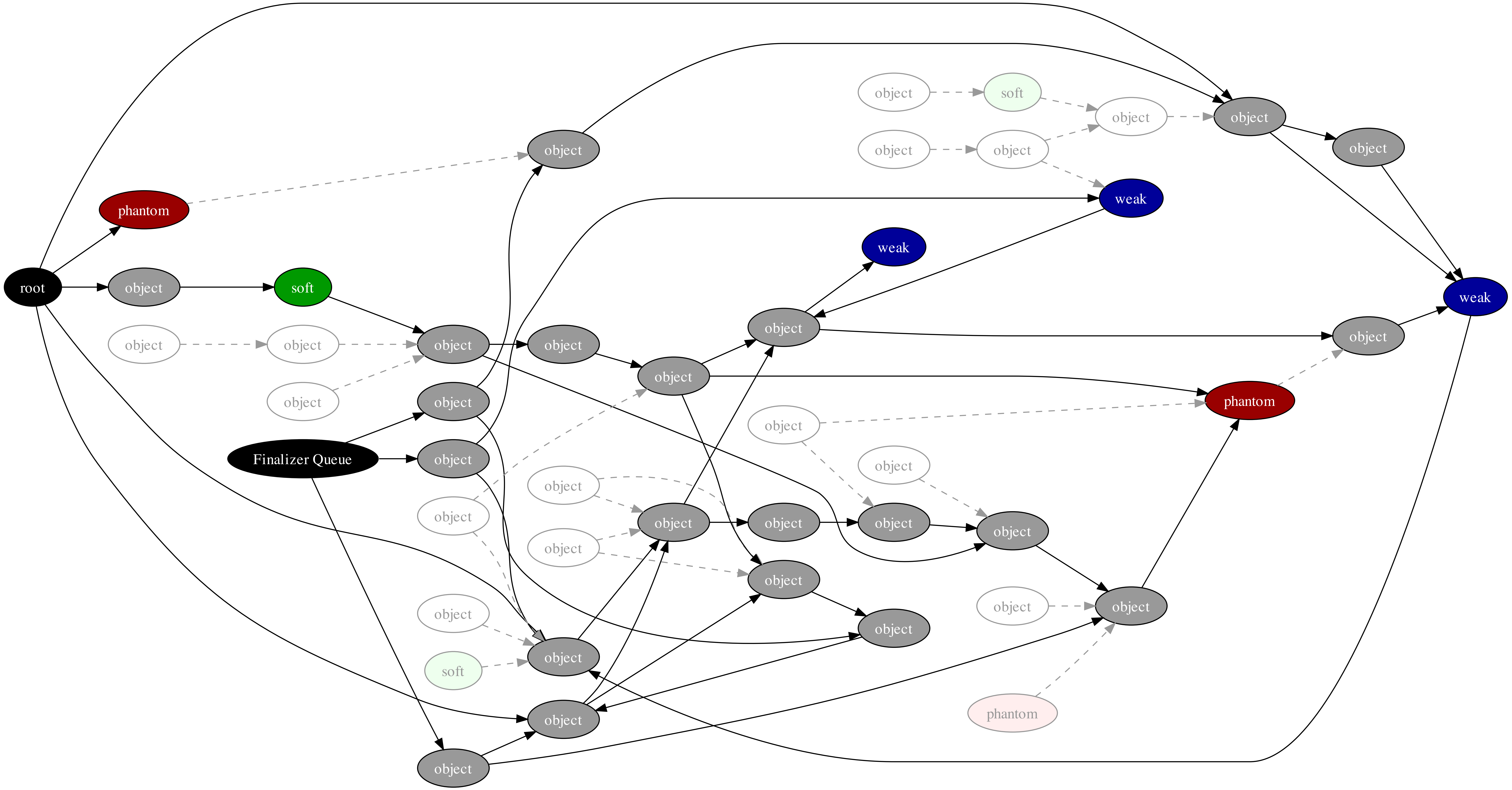
7. Repeat steps 1 through 5 for the queue.



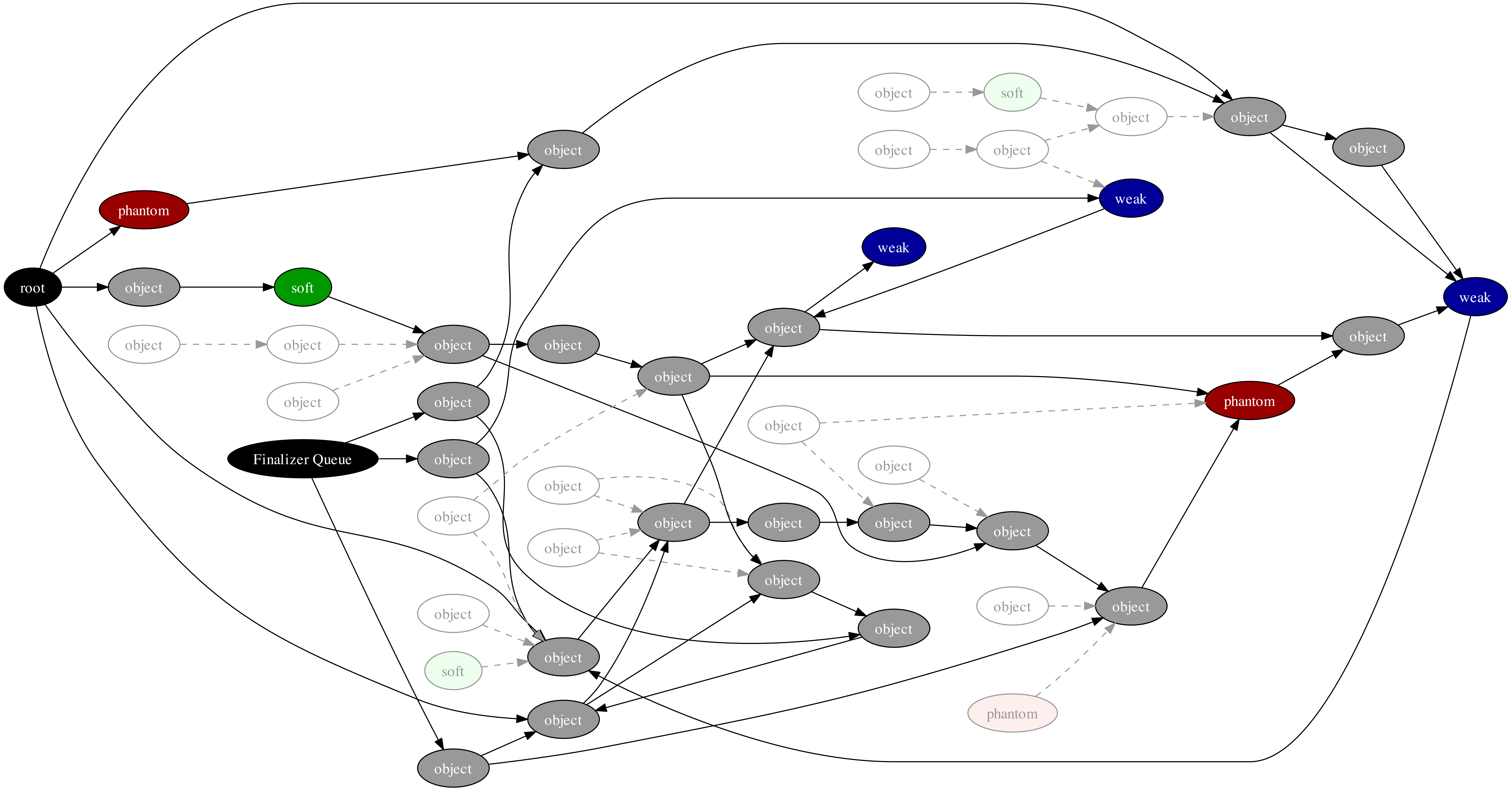
7. Repeat steps 1 through 5 for the queue.



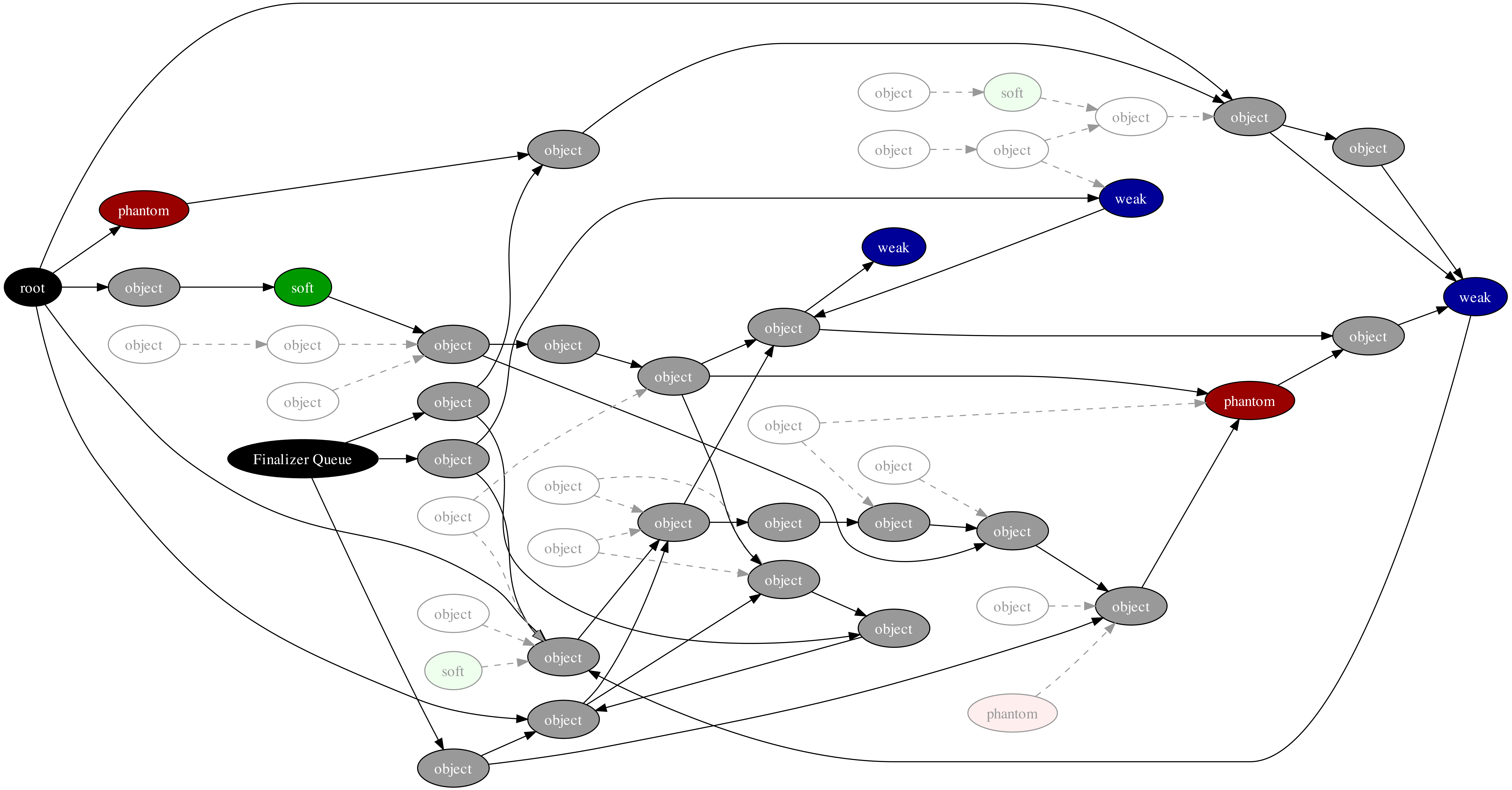
8. Possibly enqueue phantom references.



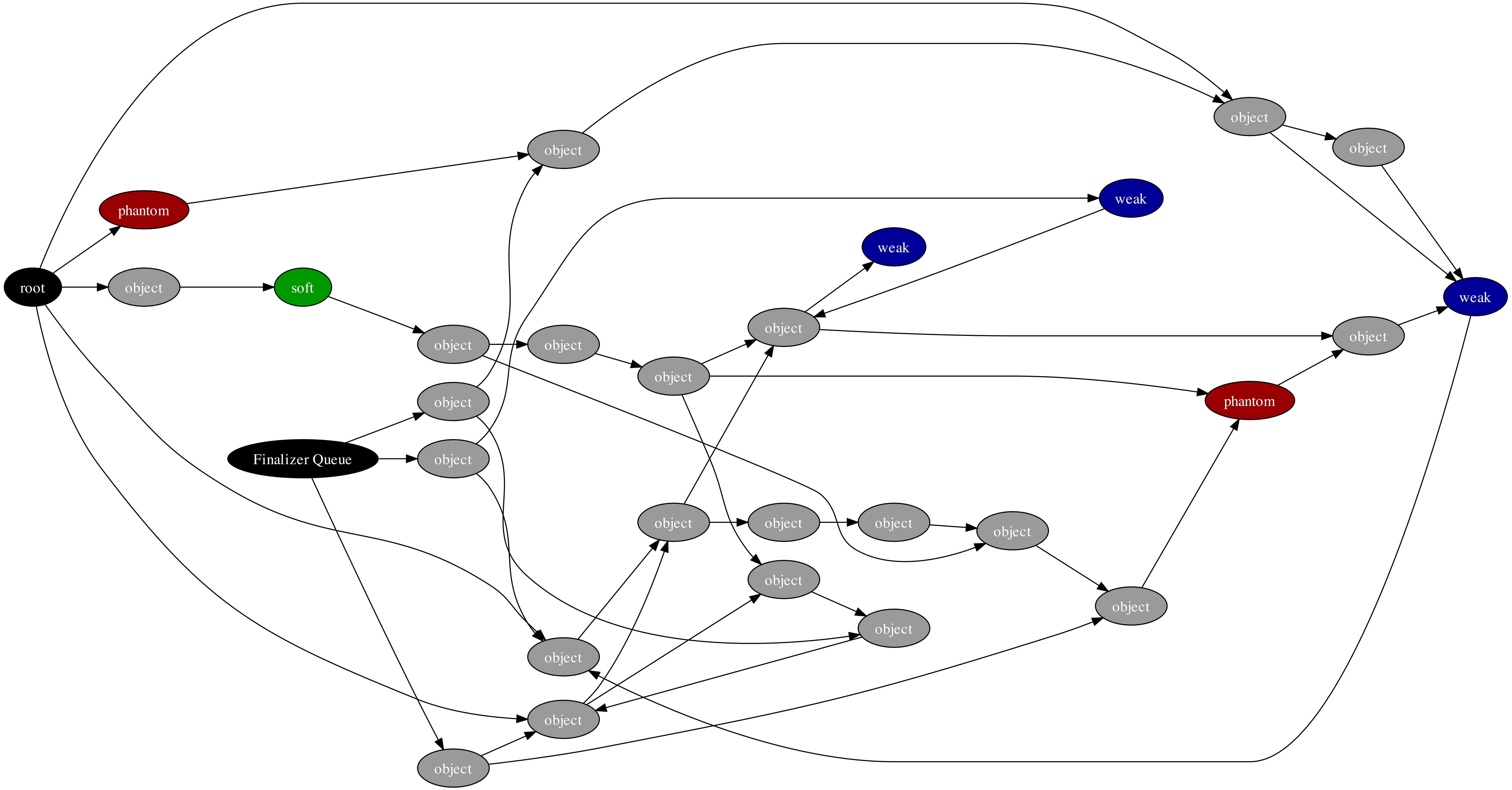
8. Possibly enqueue phantom references.



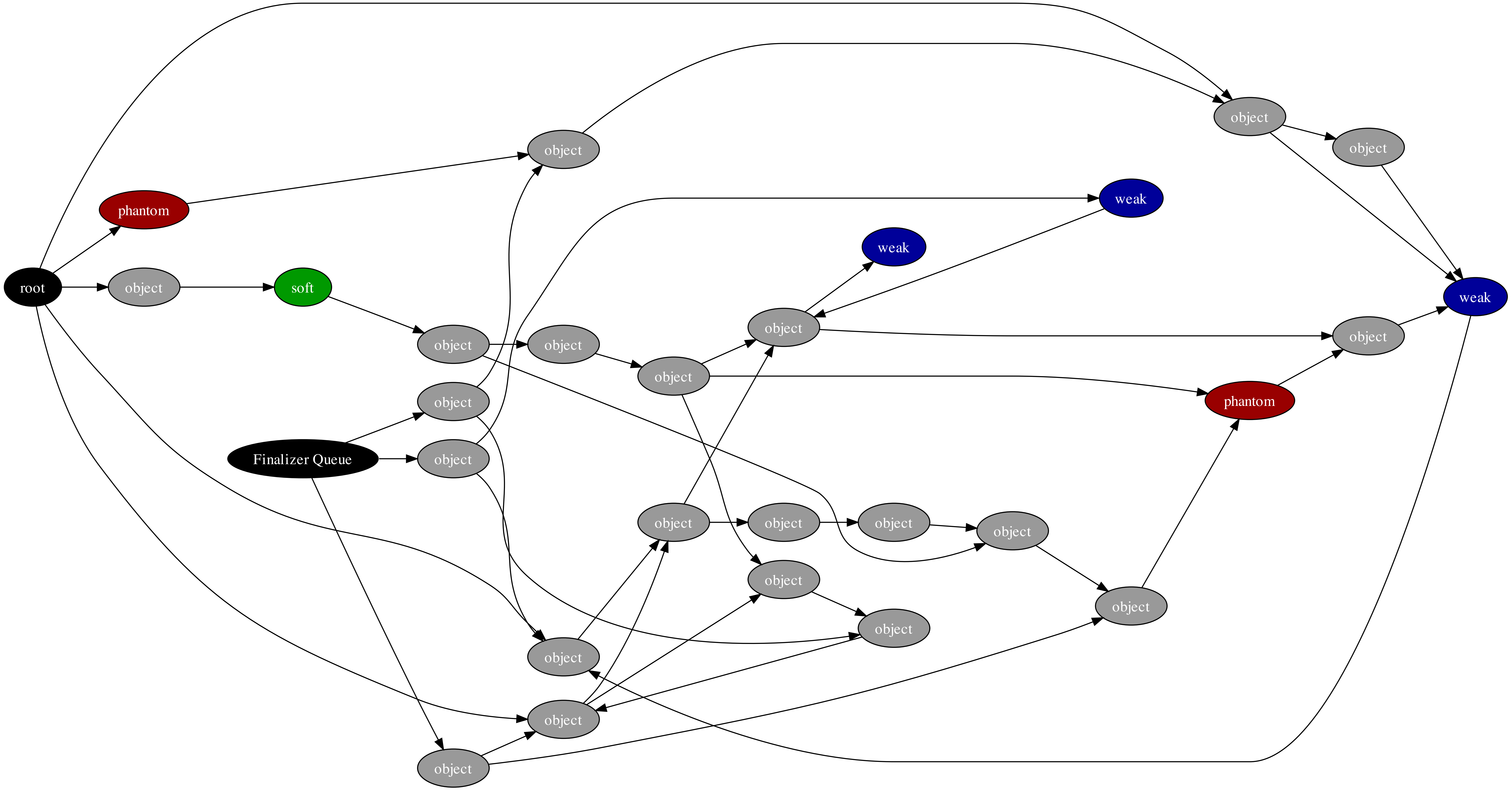
9. **The remaining objects are dead.**



9. The remaining objects are dead.



10. **Repeat.**



root
phantom
object
soft
Finalizer Queue
weak
weak
weak
phantom

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.