# On-the-Fly Garbage Collection: An Exercise in Cooperation

Edsget W. Dijkstra, Leslie Lamport,
A.J. Martin and E.F.M. Steffens
Communications of the ACM, 1978

Presented by Almog Benin
25/5/2014

# Outline of talk

- Introduction
- Problem formulation
- The first coarse-grained solution
- The second coarse-grained solution
- The fine-grained solution
- Related work
- Conclusions
- My own conclusions

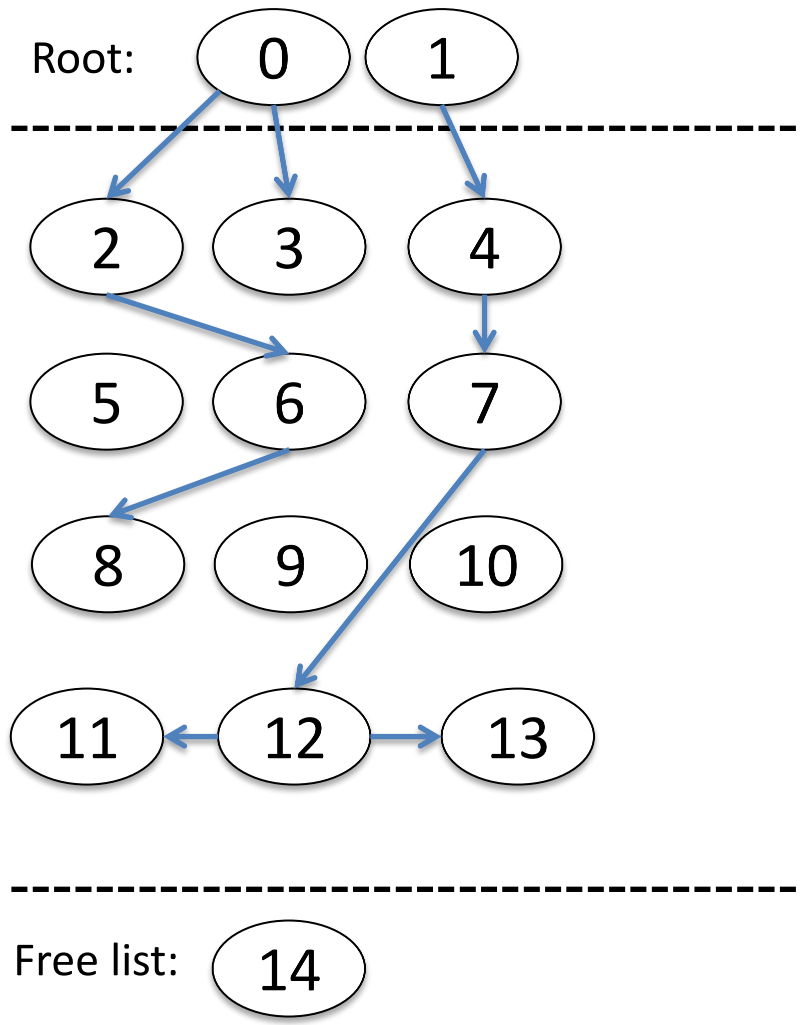
# INTRODUCTION

# Dynamic Memory

- Operations:
  - Allocate (malloc)
  - Release (free)
- The programmer is responsible for releasing the memory

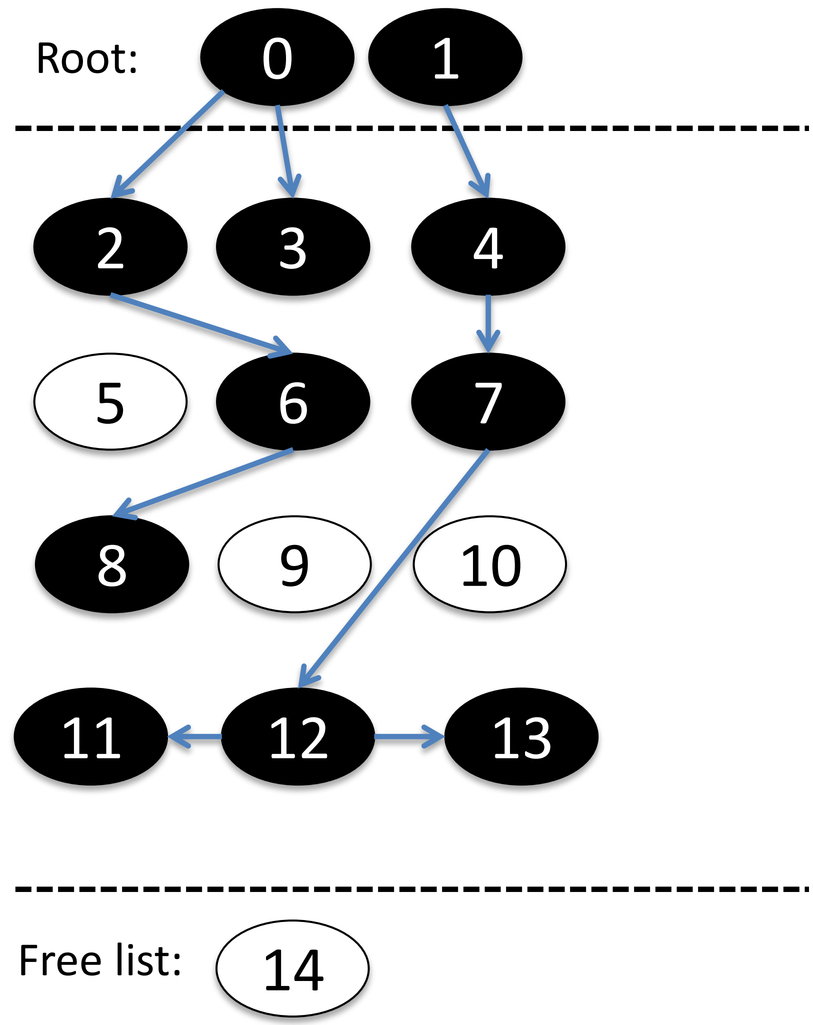
# Garbage Collector (Mark & Sweep)

- Responsible for determining which data is not in use (garbage)
- Generally, consists of 2 phases:
  - Marking phase
  - Appending phase (Sweeping phase)



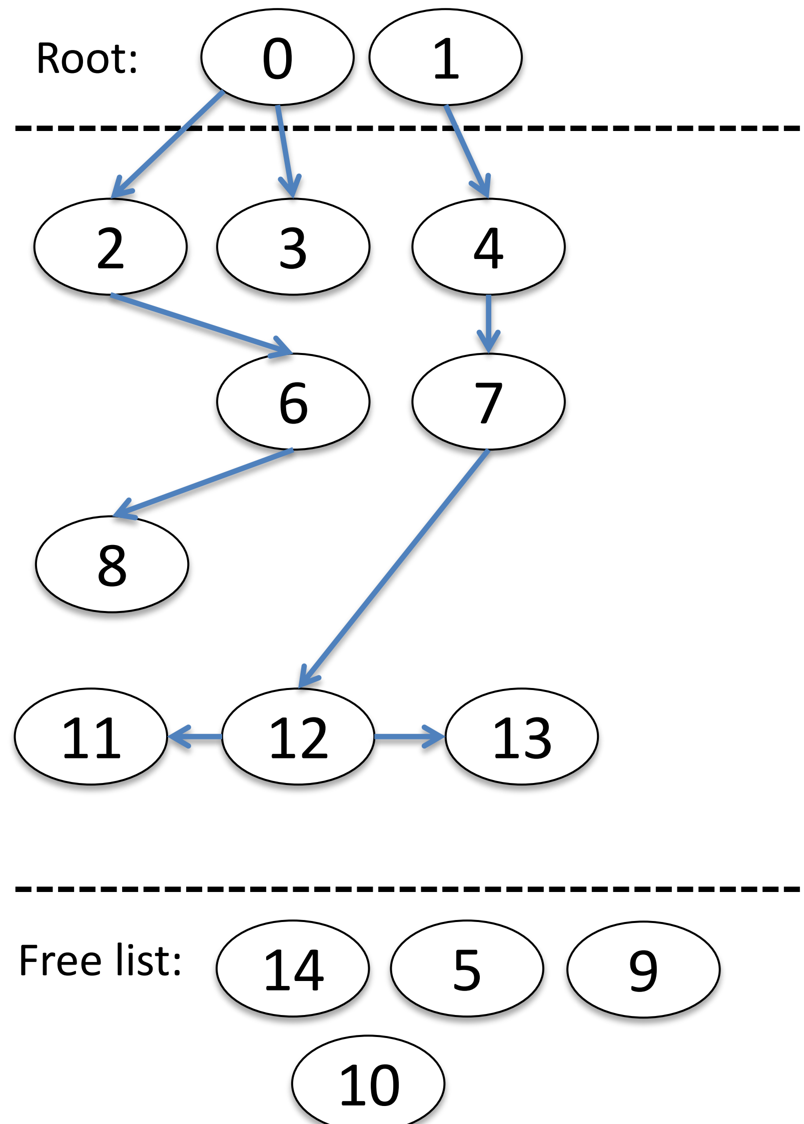
# Garbage Collector – cont’

- Responsible for determining which data is not in use (garbage)
- Generally, consists of 2 phases:
  - **Marking phase**
  - Appending phase (Sweeping phase)



# Garbage Collector – cont’

- Responsible for determining which data is not in use (garbage)
- Generally, consists of 2 phases:
  - Marking phase
  - **Appending phase (Sweeping phase)**

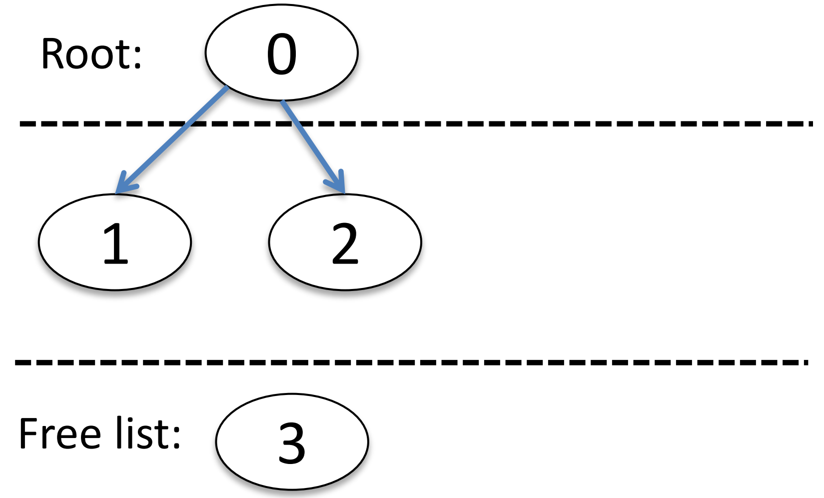


# Motivation

- Sequential garbage collection:
  - Suspend all threads
  - Execute garbage collection
  - Resume all threads
- Thread suspension may not be suitable to some applications:
  - Real Time
  - User experience
- **Can we maintain the garbage collection in a separated thread?**

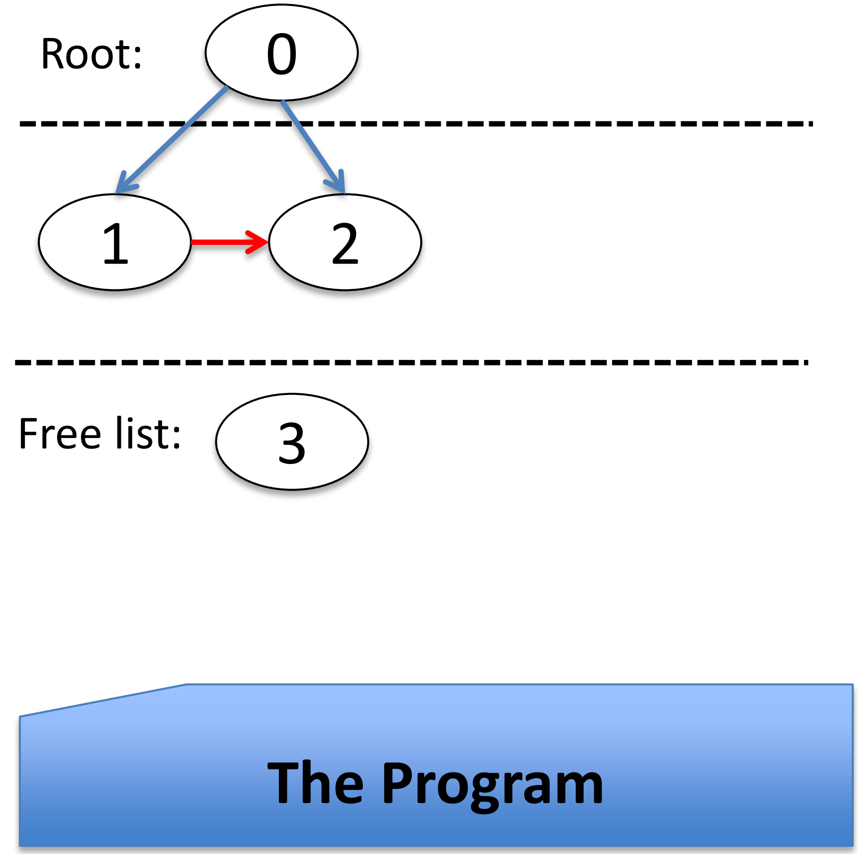
# The Challenge - Why it’s a problem?

- Node #2 is always reachable!
- The collector observes nodes one at a time.



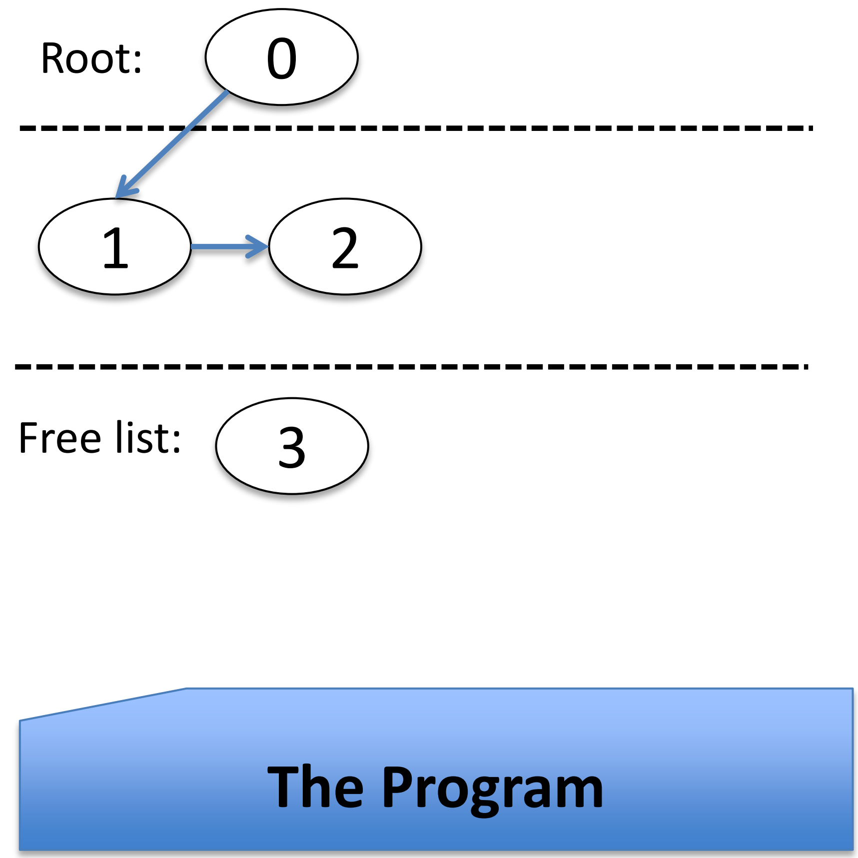
# The Challenge - Why it's a problem?

- Node #2 is always reachable!
- The collector observes nodes one at a time.



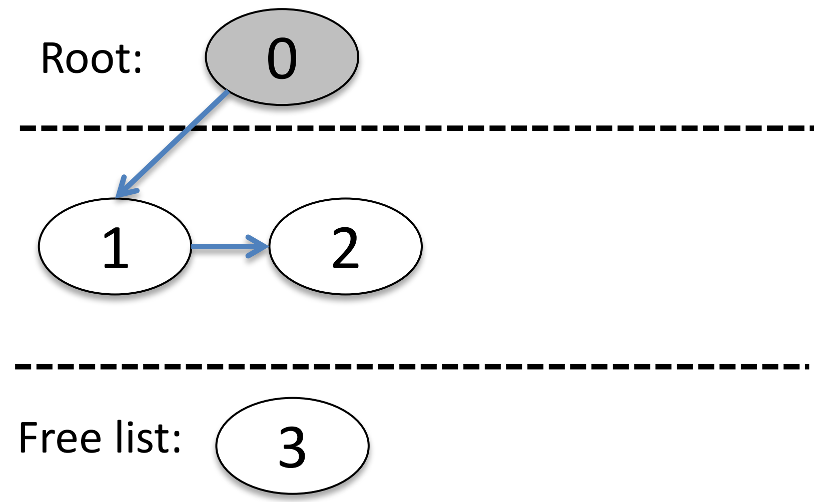
# The Challenge - Why it's a problem?

- Node #2 is always reachable!
- The collector observes nodes one at a time.



# The Challenge - Why it's a problem?

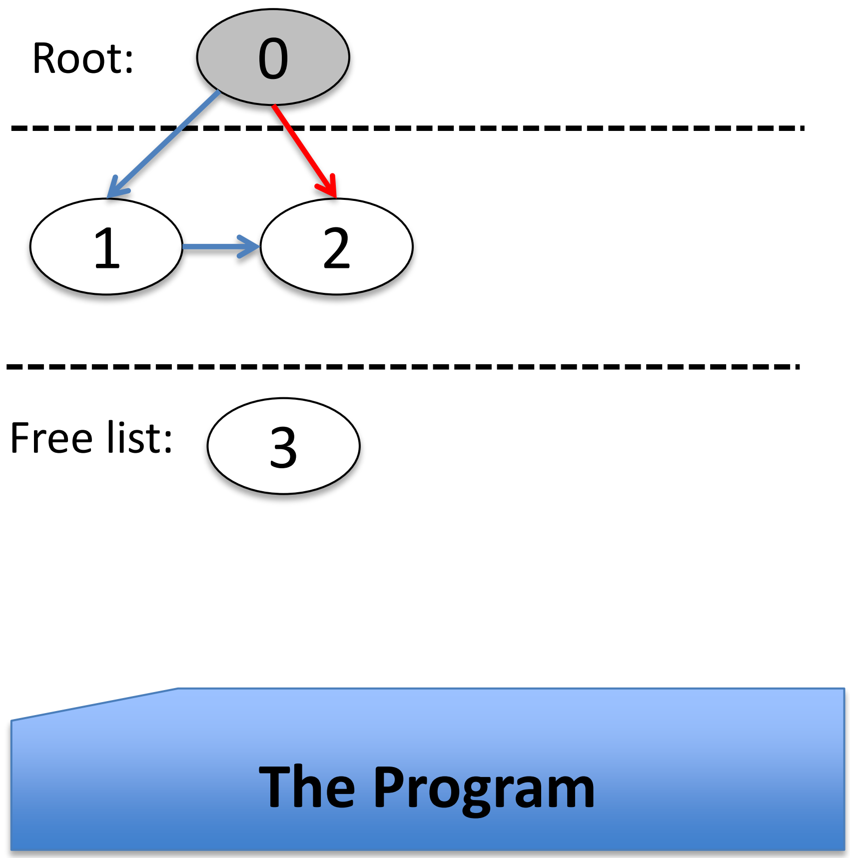
- Node #2 is always reachable!
- The collector observes nodes one at a time.




**Now the collector observes node #0 and <u>its successors</u>.**

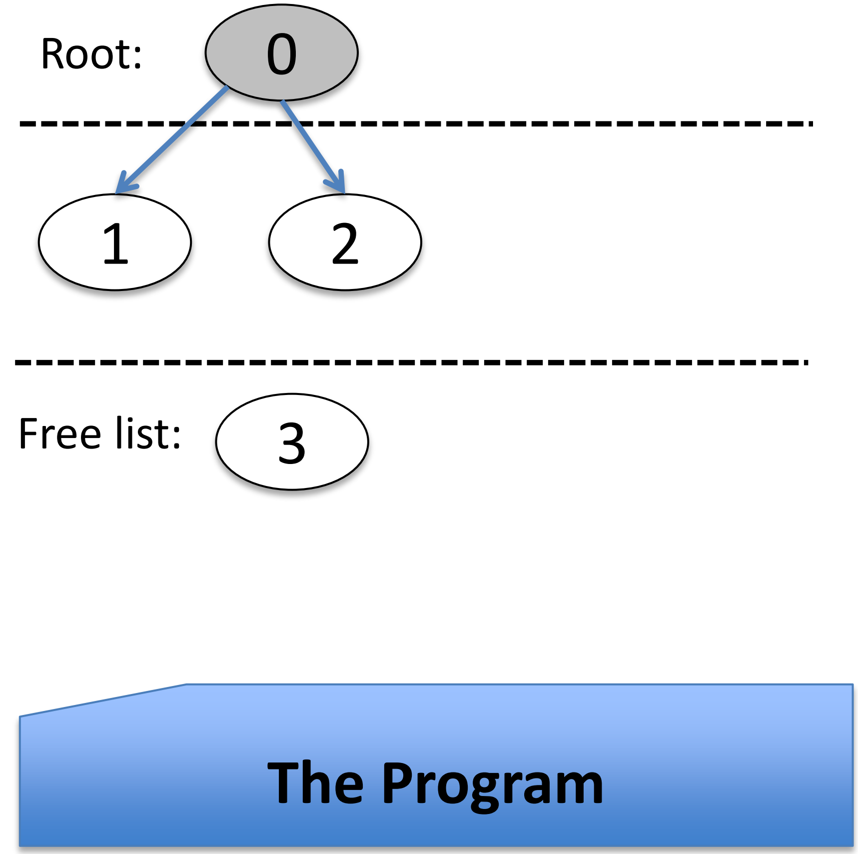
# The Challenge - Why it's a problem?

- Node #2 is always reachable!
- The collector observes nodes one at a time.



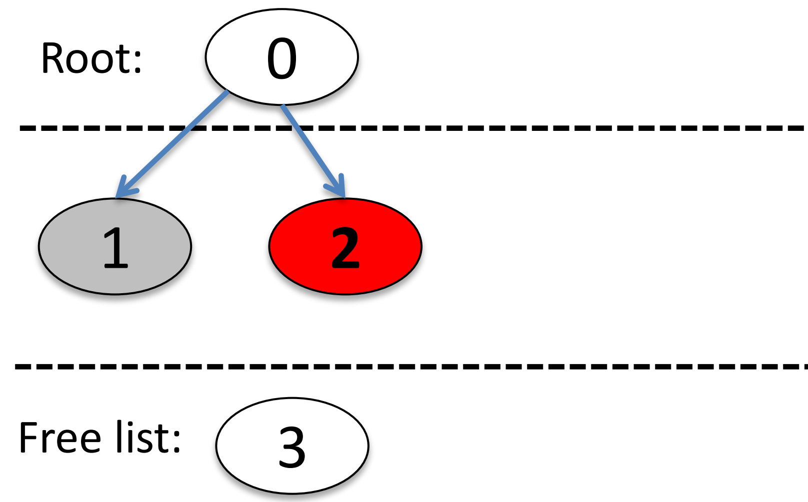
# The Challenge - Why it's a problem?

- Node #2 is always reachable!
- The collector observes nodes one at a time.



# The Challenge - Why it's a problem?

- Node #2 is always reachable!
- The collector observes nodes one at a time.
- **The collector may not notice that node #2 is reachable!**




**Now the collector observes node #1 and its successors.**

# Concurrent Garbage Collector

- Collecting the garbage concurrently to the computation proper.
  - Mutator thread
  - Collector thread
- We set the following constraints:
  - Minimal synchronization
  - Minimal overhead for the mutator
  - Collect the garbage “regardless” of the mutator activity

# Granularity – The Grain of Action

- We use <….> to denote an atomic operation.
- Coarse-grained solution uses large atomic operations.
- Fine-grained solution uses small atomic operations.

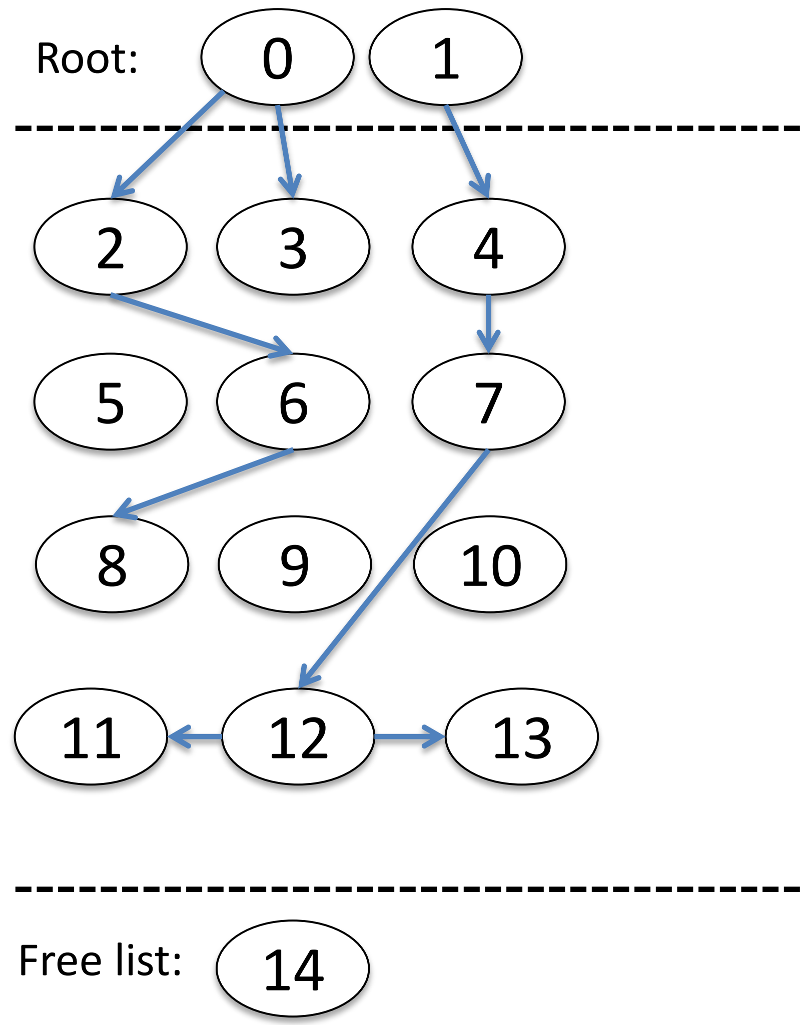
# PROBLEM FORMULATION

# The Threads

- Mutator thread(s)
  - Represents the computation proper.
- Collector thread
  - Responsible of identifying and recycling the not-used memory.

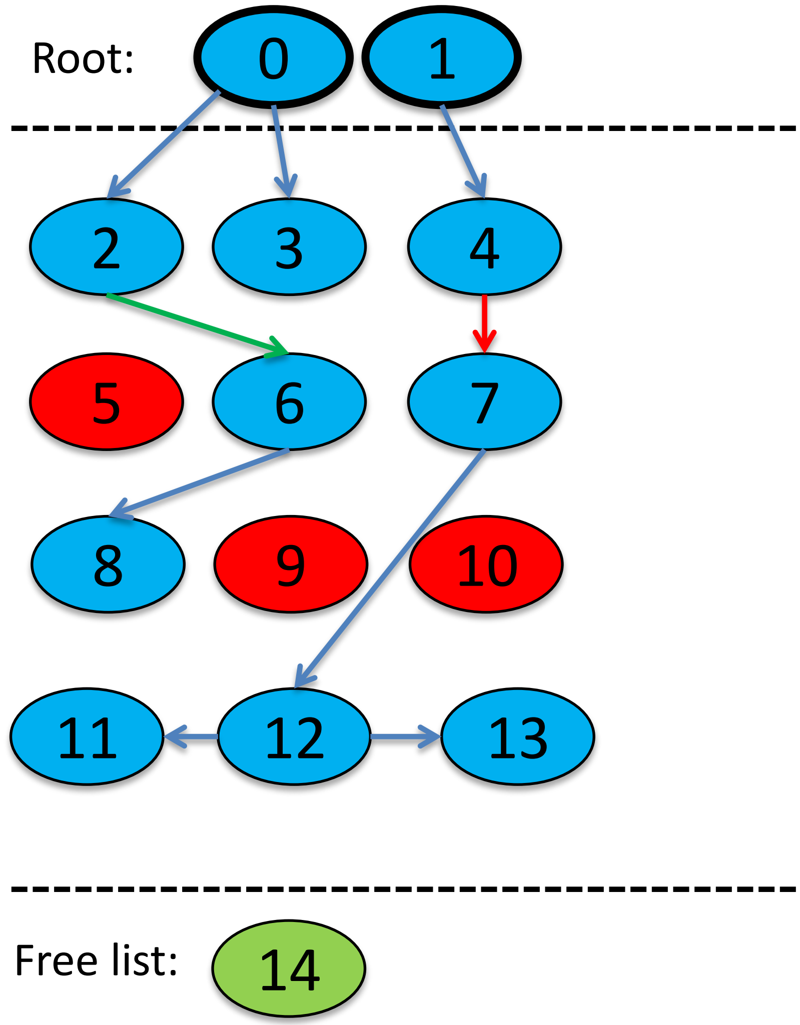
# Memory Abstraction

- Directed graph of constant nodes (but varying edges).
- Each node represents a memory block.
- Each node may have 2 outgoing edges (for the relation “point to”).



# Memory Abstraction – cont'

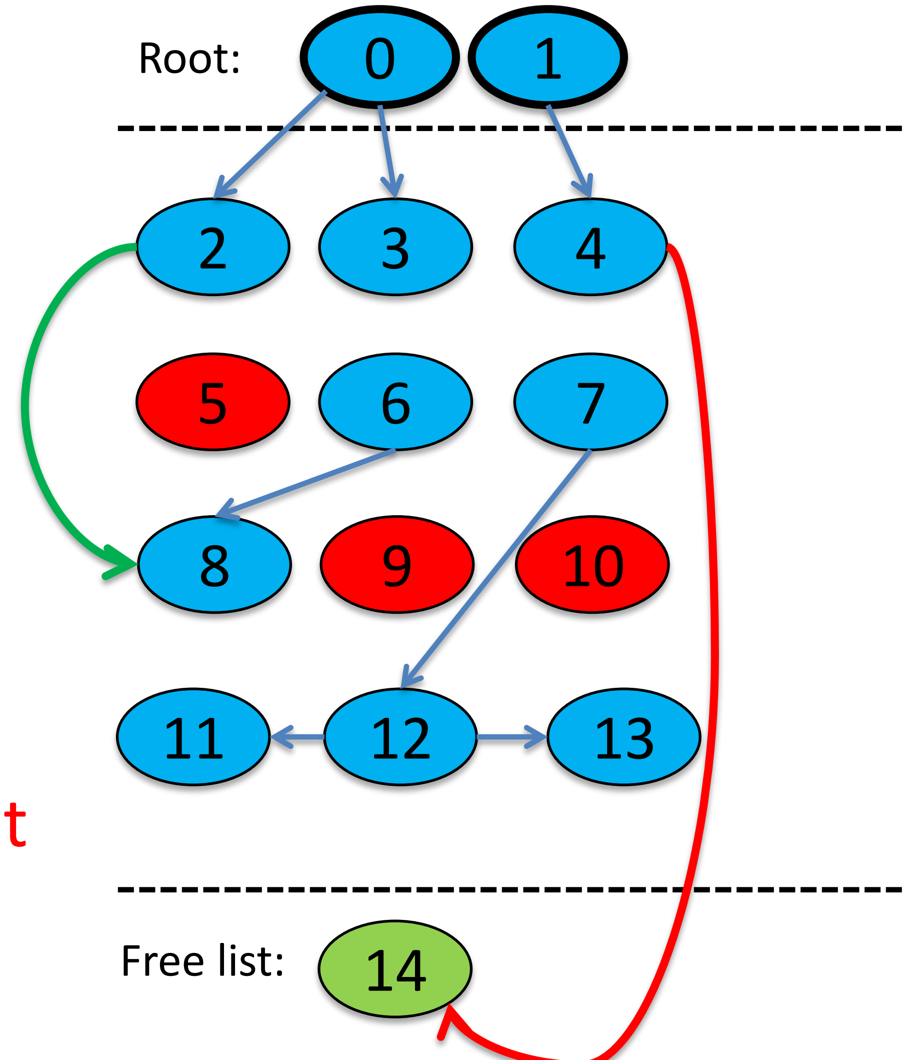
- **Root nodes** – a fixed set of nodes that cannot be garbage.
- **Reachable node** – a node that is reachable from at least one root node.
- **Data structure** – the residual sub-graph of the reachable nodes.
- **Garbage nodes** – nodes that are not reachable but are not in the free list.
- **Free list** – a list of nodes found to be garbage.



# Action Types

**1: Redirects R->R.** **2: Redirects R->NR.** 3: Add R->R. 4: Add R->NR. 5:delete

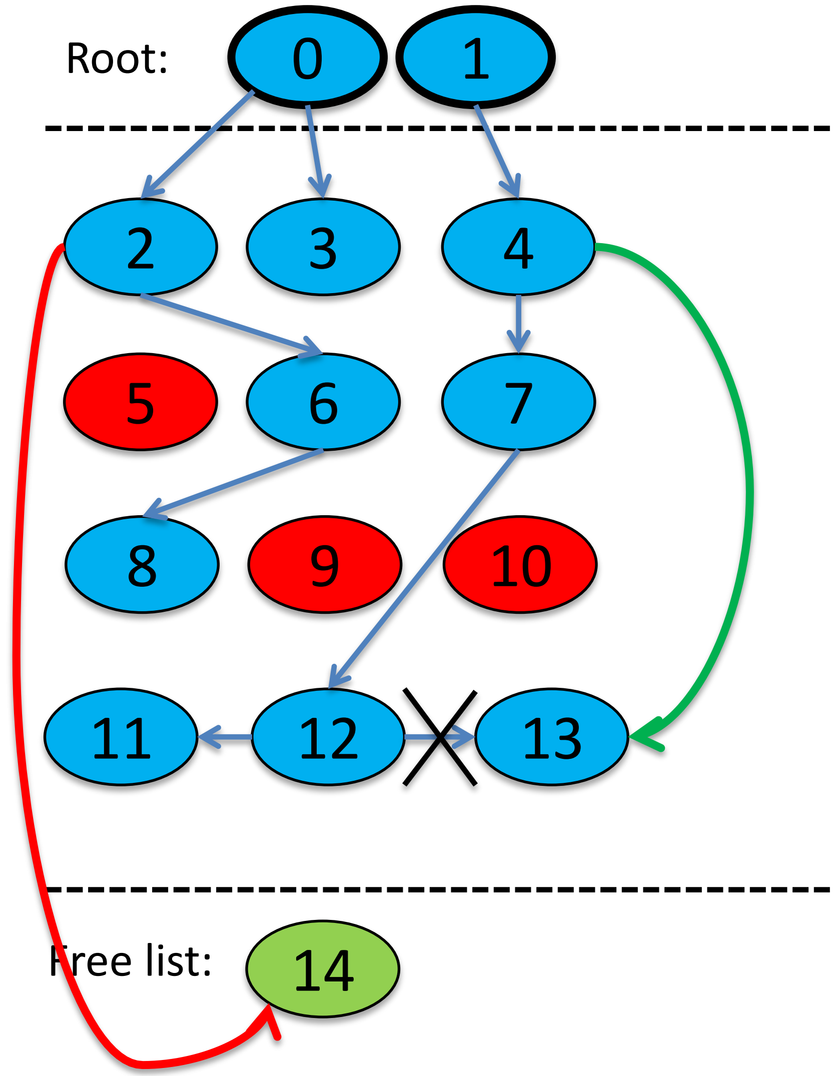
1. **Redirecting** an outgoing edge of a reachable node towards an already reachable one.
2. **Redirecting** an outgoing edge of a reachable node towards a not yet reachable one without outgoing edges.



# Action Types – cont'

1: Redirects R->R. 2: Redirects R->NR. **3: Add R->R.** **4: Add R->NR.** **5:delete**

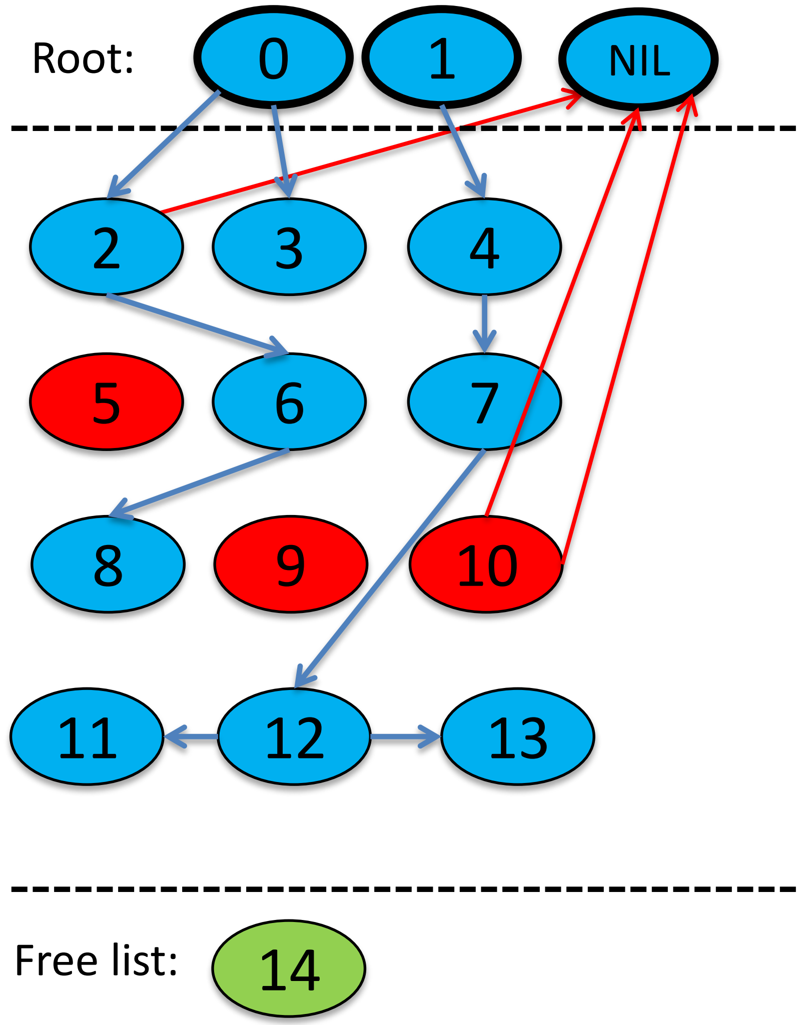
3. **Adding** an edge pointing from a reachable node towards an already reachable one.
4. **Adding** an edge pointing from a reachable node towards a not yet reachable one without outgoing edges.
5. **Removing** an outgoing edge of a reachable node.



# First Simplification

1: Redirects R->R. 2: Redirects R->NR. **3: Add R->R. 4: Add R->NR. 5:delete**

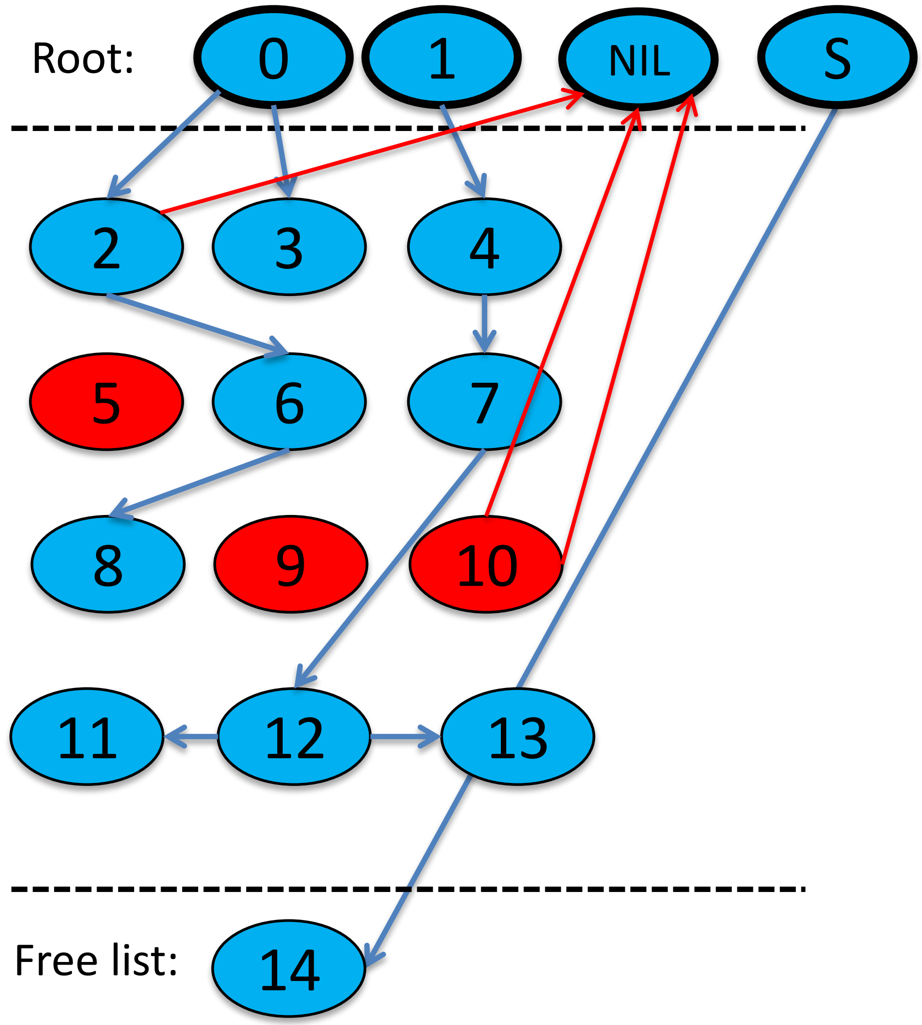
- Use special root node called "NIL".
- Pointing to such node represents a missing edge.
- Allows us to reduce the action types:
  - Action type 3 & 5 can be translated to type 1.
  - Action type 4 can be translated to type 2.



# Second Simplification

1: Redirects R->R. **2: Redirects R->NR.** 3: Add R->R. 4: Add R->NR. 5:delete

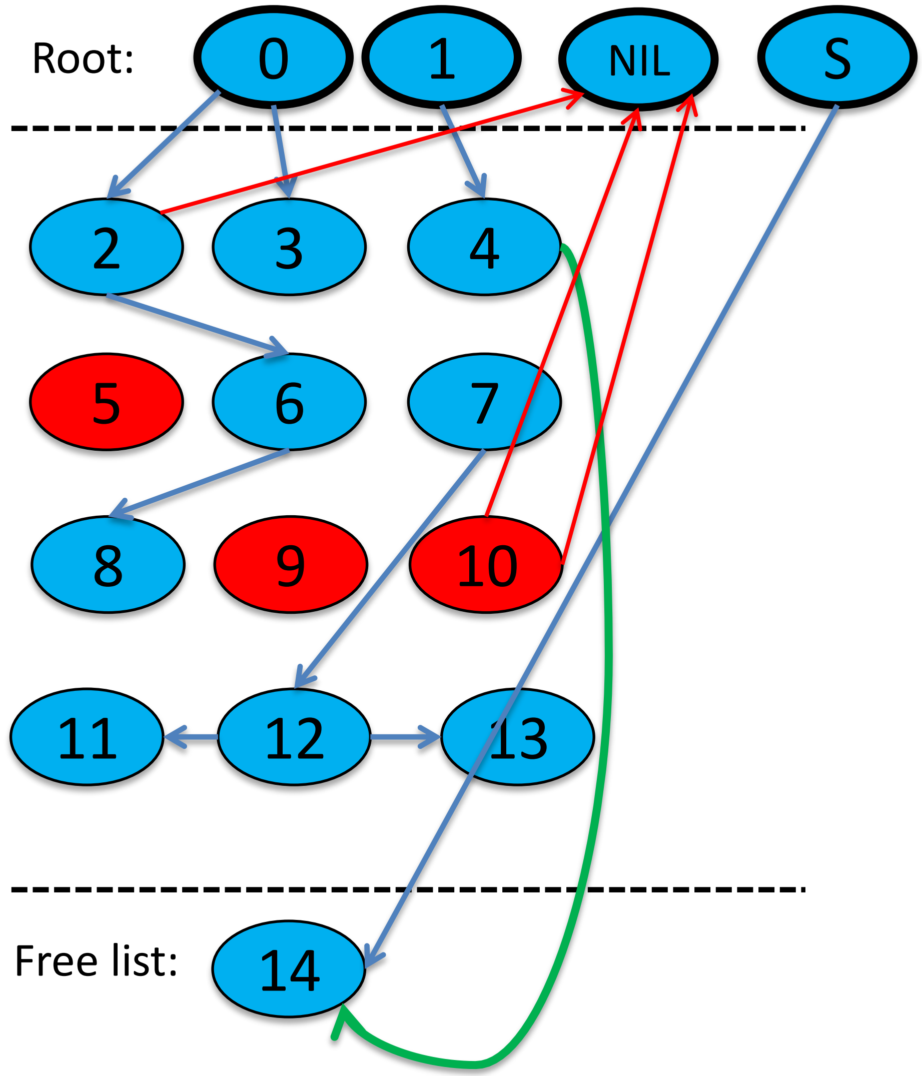
- Introducing (some) special *root* nodes and linking to them NIL and all of the free nodes.
- Making the nodes of the free list as part of the data structure.
- Allows us to reduce the action types:
  - Action type 2 can be translated to two actions of type 1.



# Second Simplification

1: Redirects R->R. **2: Redirects R->NR.** 3: Add R->R. 4: Add R->NR. 5:delete

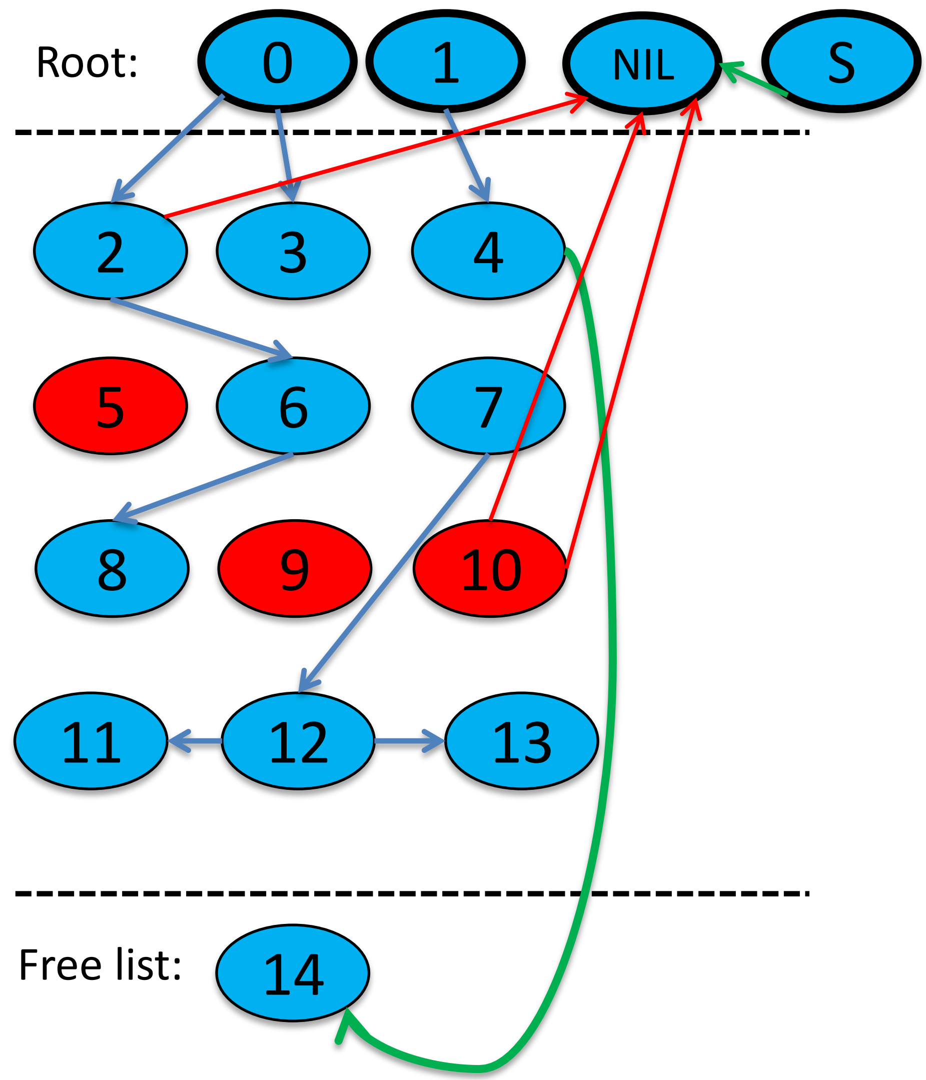
- Introducing (some) special *root* nodes and linking to them NIL and all of the free nodes.
- Making the nodes of the free list as part of the data structure.
- Allows us to reduce the action types:
  - **Action type 2 can be translated to two actions of type 1.**



# Second Simplification

1: Redirects R->R. **2: Redirects R->NR.** 3: Add R->R. 4: Add R->NR. 5:delete

- Introducing (some) special *root* nodes and linking to them NIL and all of the free nodes.
- Making the nodes of the free list as part of the data structure.
- Allows us to reduce the action types:
  - **Action type 2 can be translated to two actions of type 1.**



# The Resulting Formulation

- There are 2 thread types:
  - Mutator(s):
    - “Redirect an outgoing edge of a reachable node towards an already reachable one” (Action type 1)
  - Collector:
    - Marking phase: “Mark all reachable nodes”
    - Appending phase: “Append all unmarked nodes to the free list an clear the marking from all nodes”
- **For simplifying the description, we hide the new edges\nodes from the subsequent slides.**

# Correctness Criteria

- CC1: Every garbage node is eventually appended to the free list.
- CC2: Appending a garbage node to the free list is the collector’s only modification of the shape of the data structure.

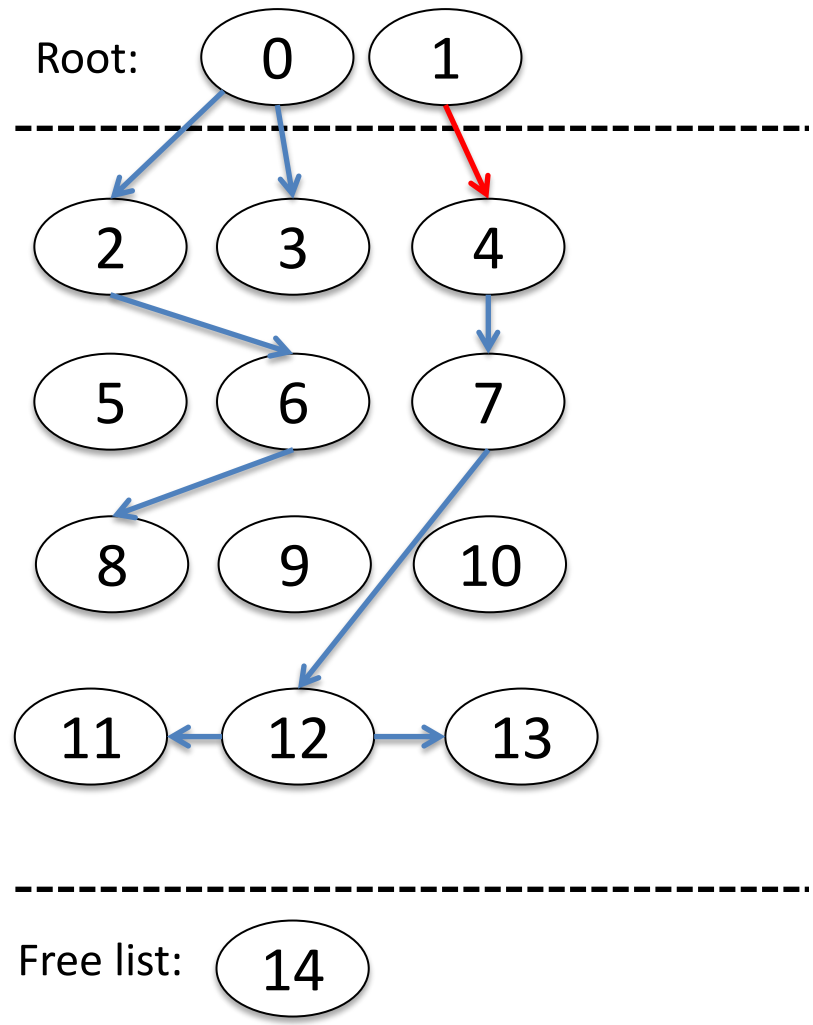
# THE FIRST COARSE-GRAINED SOLUTION

# Using Colors for marking

- 2 Basic colors:
  - White: Not found to be reachable yet.
  - Black: Found to be reachable.
- The monotonicity invariant "P1":
  - "No edge points from a black node to a white one"
- Need an intermediate color:
  - Gray

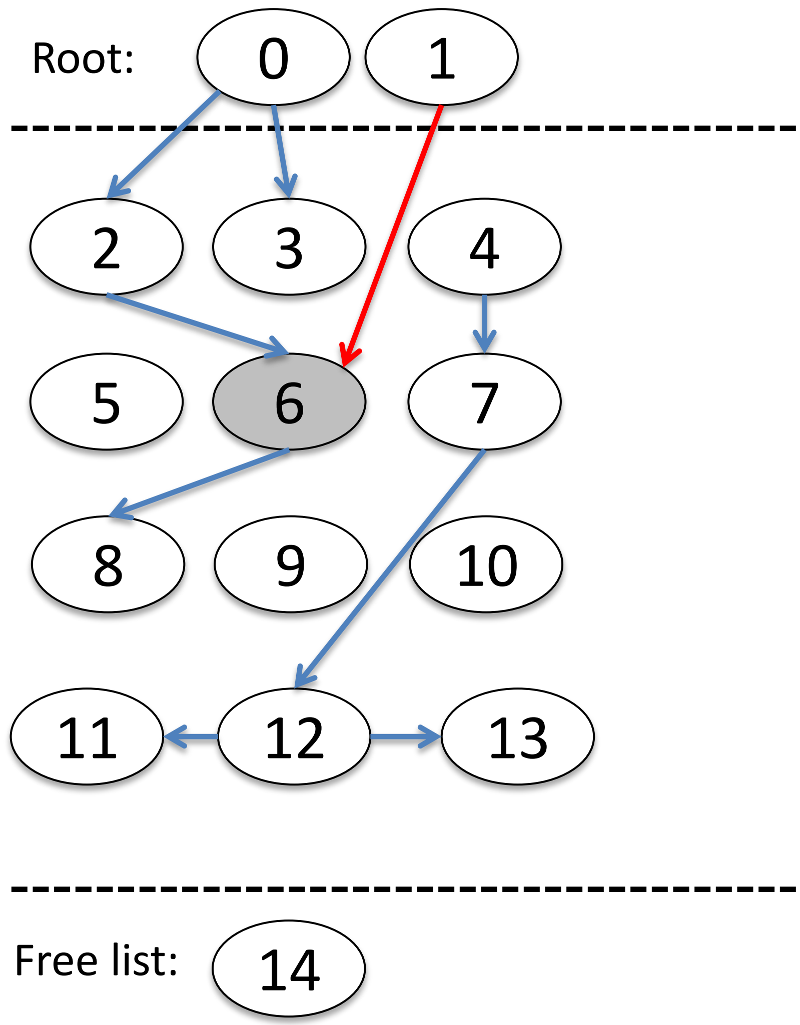
# The Mutator

- The "Shade" operation on a node:
  - If the node is white, make it gray.
  - Otherwise (gray\black), leave it unchanged.
- The mutator operation "M1":
  - <Redirect an outgoing edge of a reachable node towards an already reachable one, and shade the new target>



# The Mutator

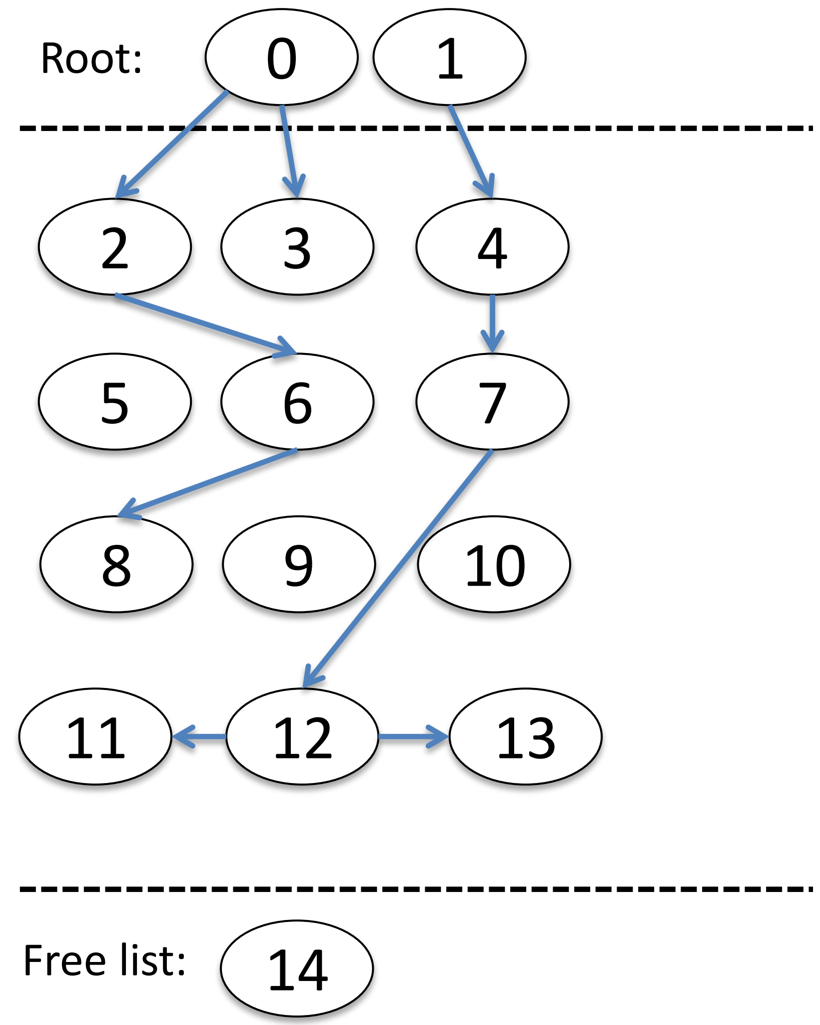
- The “Shade” operation on a node:
  - If the node is white, make it gray.
  - Otherwise (gray\black), leave it unchanged.
- The mutator operation “M1”:
  - **<Redirect an outgoing edge of a reachable node towards an already reachable one, and shade the new target>**



# The Collector: The Marking Phase

1. Shade all roots.
2. $i \leftarrow 0$
3. $k \leftarrow M$
4. While $k > 0$
    1. < $c$ ←color of node #$i$>
    2. If $c$ ==Gray then
        1. “C1”: <Shade the successors of node #$i$ and make node #$i$ black>
    3. Else
        1. $k \leftarrow k - 1$
    4. $i \leftarrow (i + 1)\%M$

$M$ is the nodes count in the graph



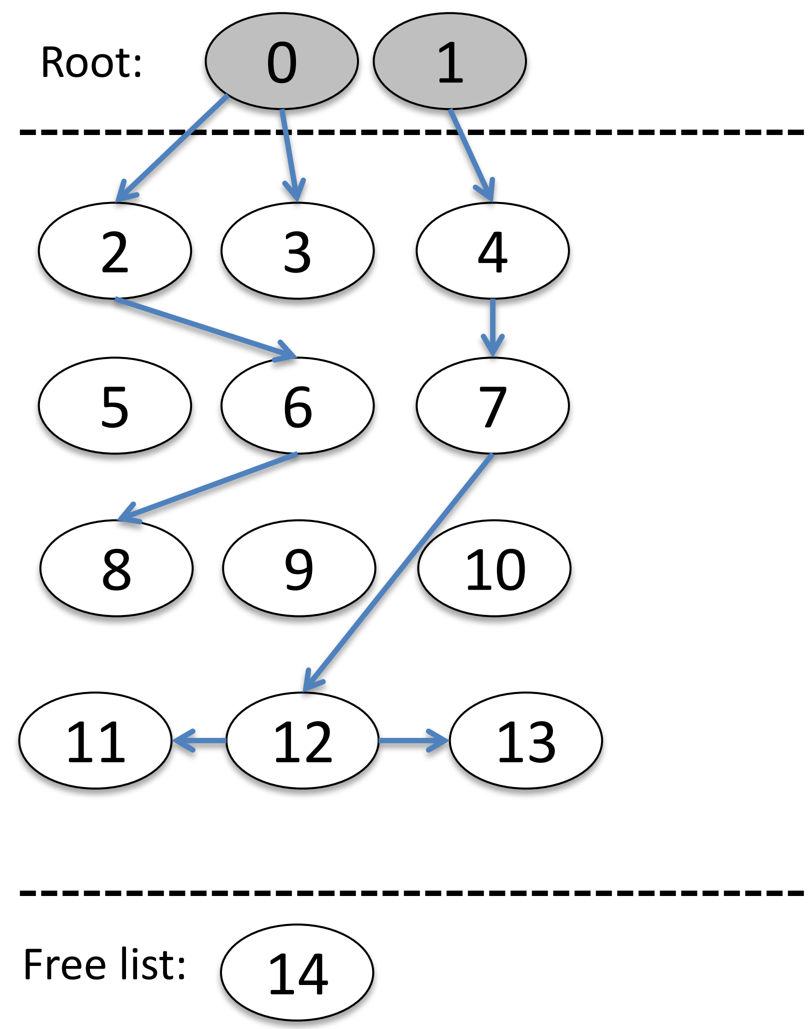

Green simple atomic operations.

We will try to break the red.

# The Collector: The Marking Phase

1. **Shade all roots.**
2. $i \leftarrow 0$
3. $k \leftarrow M$
4. While $k > 0$
    1. < $c$ ←color of node #$i$>
    2. If $c$ ==Gray then
        1. “C1”: <Shade the successors of node #$i$ and make node #$i$ black>
    3. Else
        1. $k \leftarrow k - 1$
    4. $i \leftarrow (i + 1)\%M$

$M$ is the nodes count in the graph



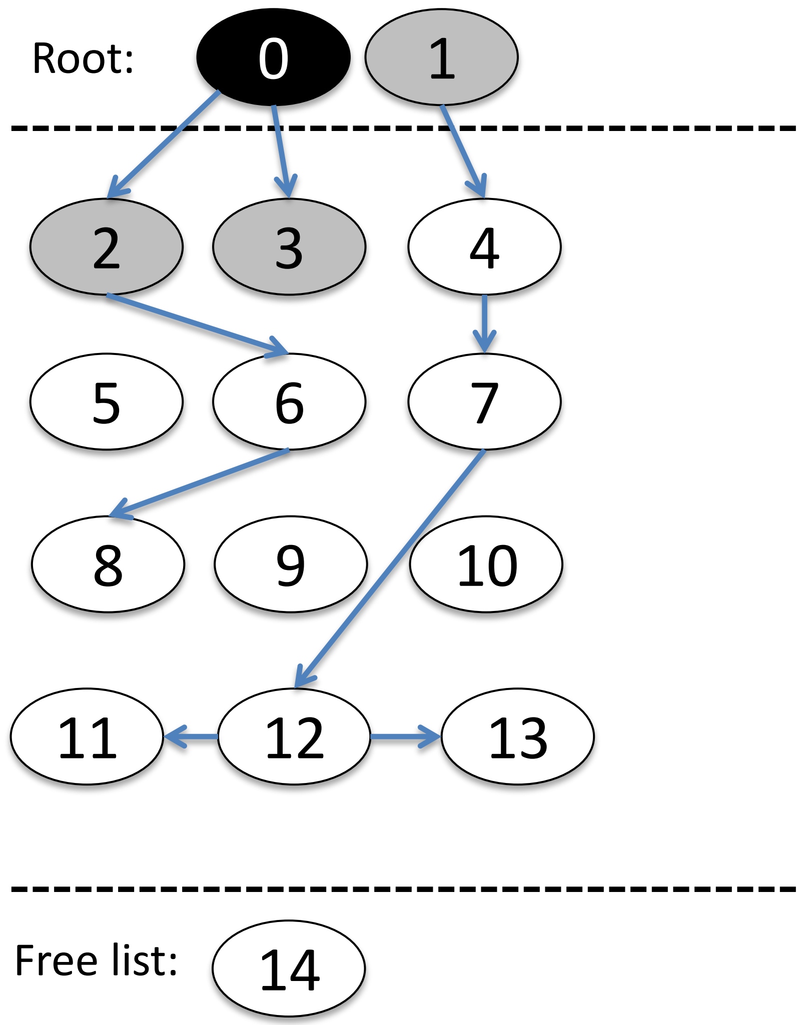

Green simple atomic operations.

We will try to break the red.

# The Collector: The Marking Phase

1. Shade all roots.
2. $i \leftarrow 0$
3. $k \leftarrow M$
4. **While $\boldsymbol{k > 0}$**
   1. **< $\boldsymbol{c}$ ←color of node #$\boldsymbol{i}$>**
   2. **If $\boldsymbol{c}$ ==Gray then**
      1. **“C1”: <Shade the successors of node #$\boldsymbol{i}$ and make node #$\boldsymbol{i}$ black>**
   3. **Else**
      1. $\boldsymbol{k \leftarrow k - 1}$
   4. $\boldsymbol{i \leftarrow (i + 1)\%M}$

$M$ is the nodes count in the graph



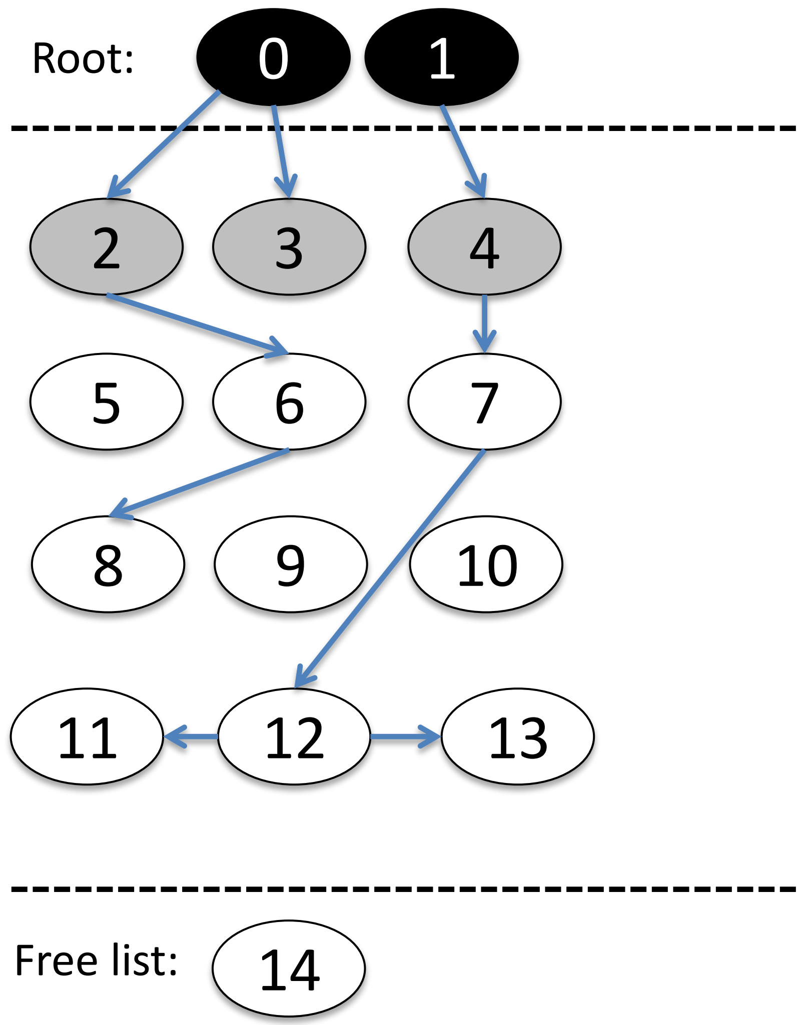

Green simple atomic operations.

We will try to break the red.

# The Collector: The Marking Phase

1. Shade all roots.
2. $i \leftarrow 0$
3. $k \leftarrow M$
4. **While $k > 0$**
    1. **< $c$ ←color of node #$i$>**
    2. **If $c$ ==Gray then**
        1. **"C1": <Shade the successors of node #$i$ and make node #$i$ black>**
    3. **Else**
        1. $k \leftarrow k - 1$
    4. $i \leftarrow (i + 1)\%M$

$M$ is the nodes count in the graph



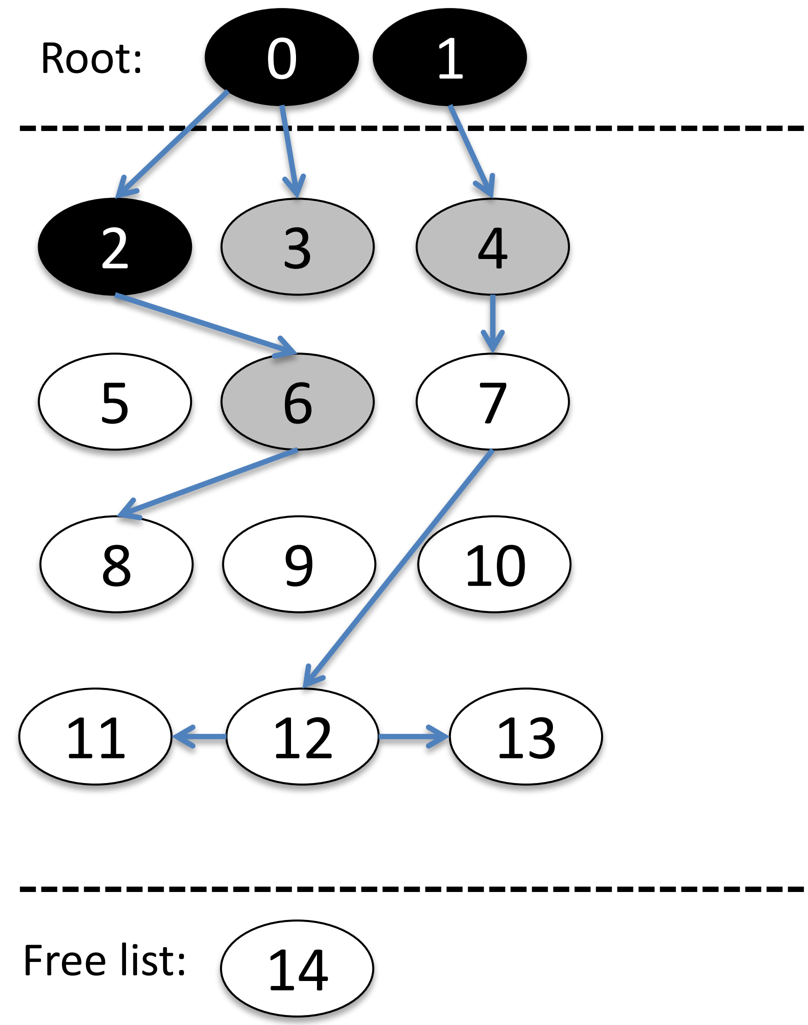

Green simple atomic operations.

We will try to break the red.

# The Collector: The Marking Phase

1. Shade all roots.
2. $i \leftarrow 0$
3. $k \leftarrow M$
4. **While $k > 0$**
    1. **< $c$ ←color of node #$i$>**
    2. **If $c$ ==Gray then**
        1. **“C1”: <Shade the successors of node #$i$ and make node #$i$ black>**
    3. **Else**
        1. $k \leftarrow k - 1$
    4. $i \leftarrow (i+1)\%M$

$M$ is the nodes count in the graph




Green simple atomic operations.

We will try to break the red.

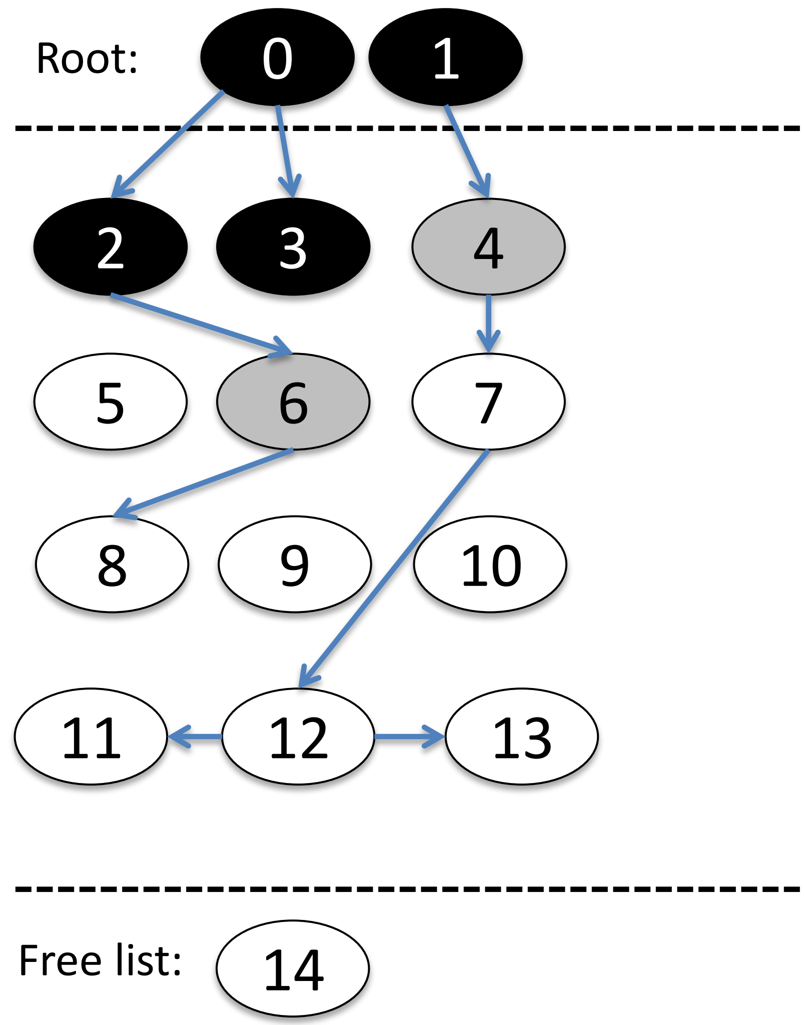
# The Collector: The Marking Phase

$M$ is the nodes count in the graph

1. Shade all roots.
2. $i \leftarrow 0$
3. $k \leftarrow M$
4. **While $k > 0$**
   1. **< $c$ ←color of node #$i$>**
   2. **If $c$ ==Gray then**
      1. **“C1”: <Shade the successors of node #$i$ and make node #$i$ black>**
   3. **Else**
      1. $k \leftarrow k - 1$
   4. $i \leftarrow (i + 1)\%M$

Green simple atomic operations.

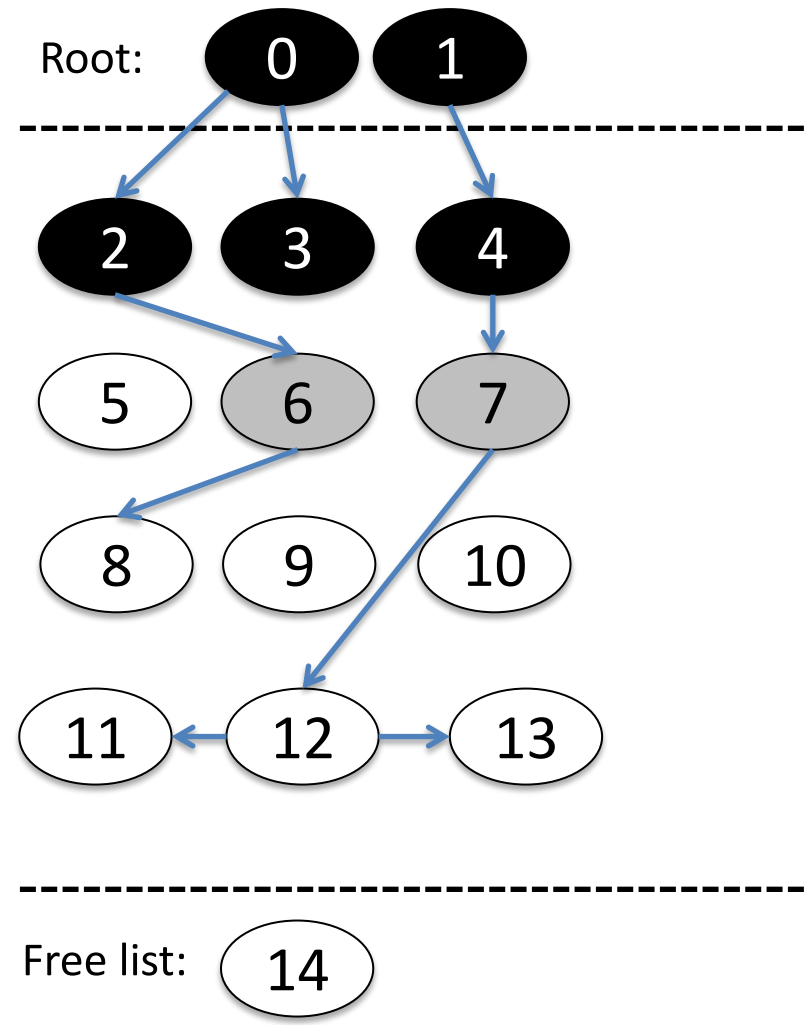
We will try to break the red.



# The Collector: The Marking Phase

1. Shade all roots.
2. $i \leftarrow 0$
3. $k \leftarrow M$
4. **While $k > 0$**
    1. **< $c$ ←color of node #$i$>**
    2. **If $c$ ==Gray then**
        1. **"C1": <Shade the successors of node #$i$ and make node #$i$ black>**
    3. **Else**
        1. $\boldsymbol{k \leftarrow k - 1}$
    4. $\boldsymbol{i \leftarrow (i + 1)\% M}$

$M$ is the nodes count in the graph



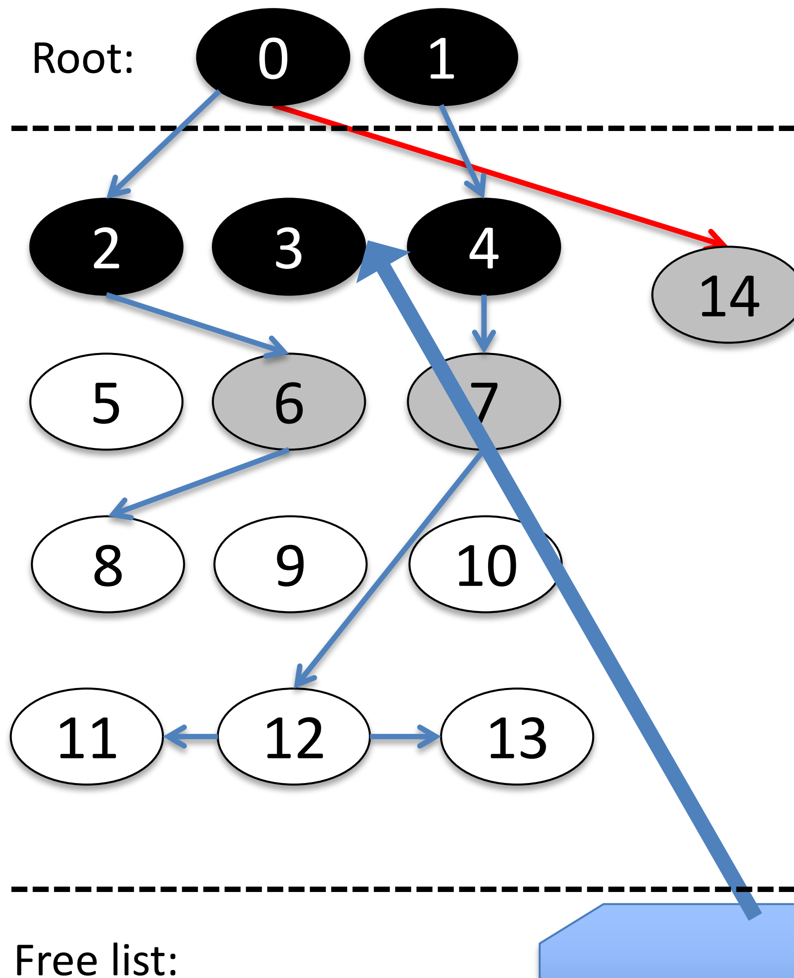

Green simple atomic operations.

We will try to break the red.

# The Collector: The Marking Phase

1. Shade all roots.
2. $i \leftarrow 0$
3. $k \leftarrow M$
4. While $k > 0$
    1. < $c$ ←color of node #$i$>
    2. If $c$ ==Gray then
        1. “C1”: <Shade the successors of node #$i$ and make node #$i$ black>
    3. Else
        1. $k \leftarrow k - 1$
    4. $i \leftarrow (i + 1)\%M$

$M$ is the nodes count in the graph




Mutator interleaved!

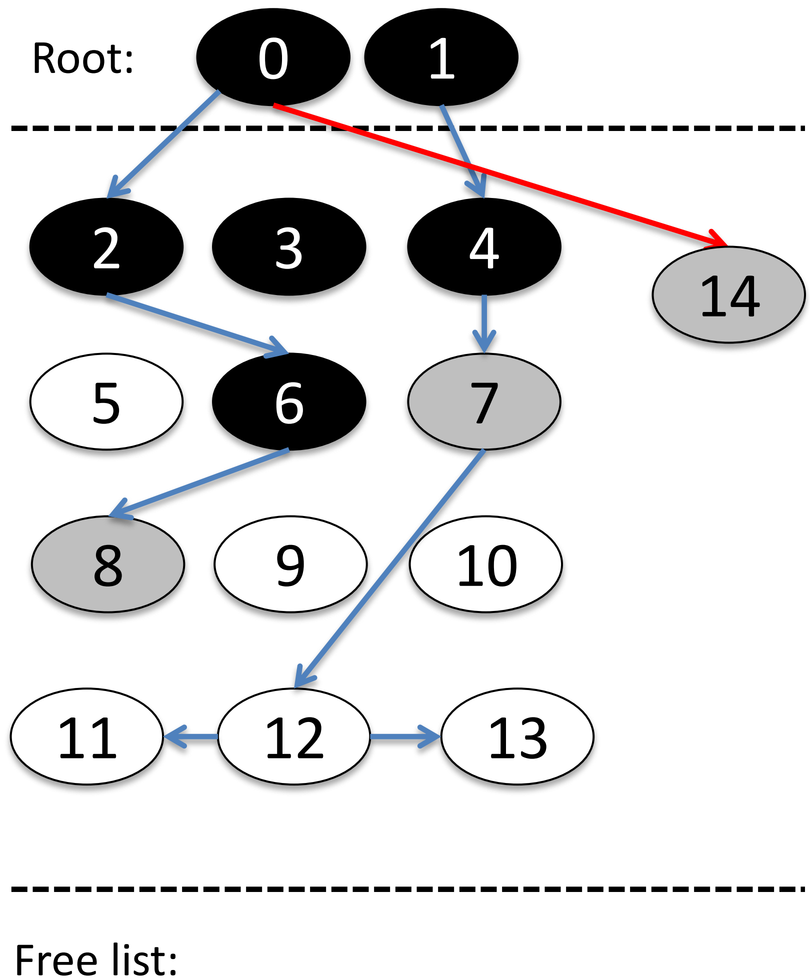
Green simple atomic operations.

We will try to break the red.

# The Collector: The Marking Phase

1. Shade all roots.
2. $i \leftarrow 0$
3. $k \leftarrow M$
4. **While $k > 0$**
    1. **< $c$ ←color of node #$i$>**
    2. **If $c$ ==Gray then**
        1. **"C1": <Shade the successors of node #$i$ and make node #$i$ black>**
    3. **Else**
        1. $\boldsymbol{k \leftarrow k - 1}$
    4. $\boldsymbol{i \leftarrow (i+1)\%M}$

$M$ is the nodes count in the graph



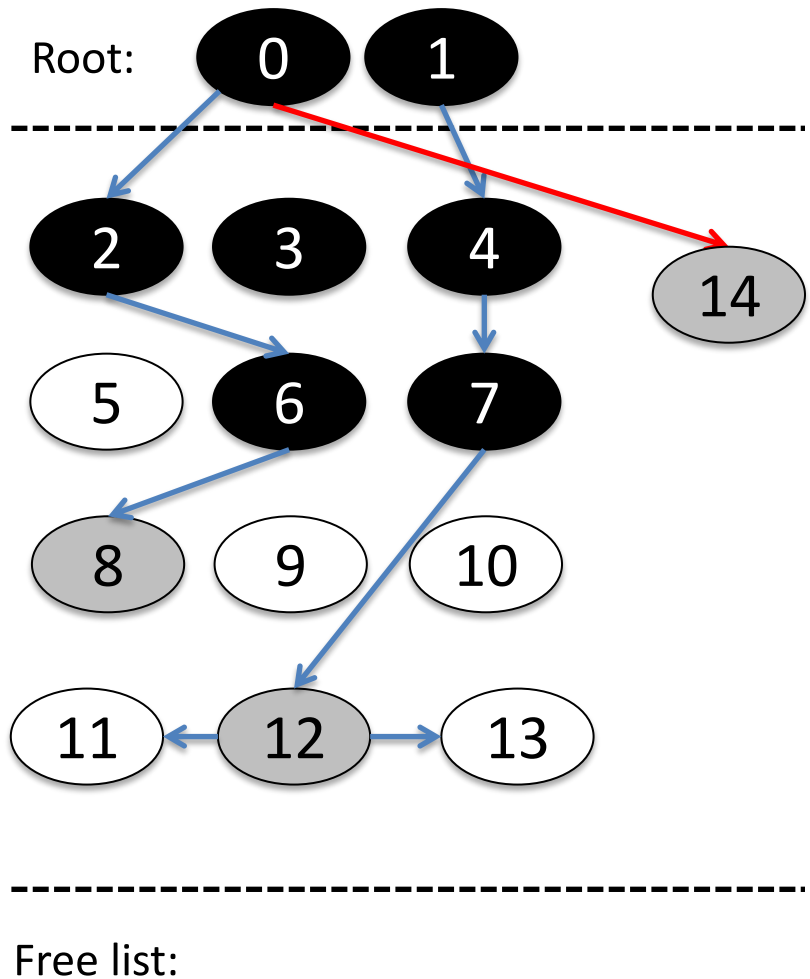

Green simple atomic operations.

We will try to break the red.

# The Collector: The Marking Phase

1. Shade all roots.
2. $i \leftarrow 0$
3. $k \leftarrow M$
4. **While $k > 0$**
    1. < $c$ ←color of node #$i$>
    2. **If $c$ ==Gray then**
        1. **“C1”: <Shade the successors of node #$i$ and make node #$i$ black>**
    3. **Else**
        1. $\boldsymbol{k \leftarrow k - 1}$
    4. $\boldsymbol{i \leftarrow (i + 1)\%M}$

$M$ is the nodes count in the graph



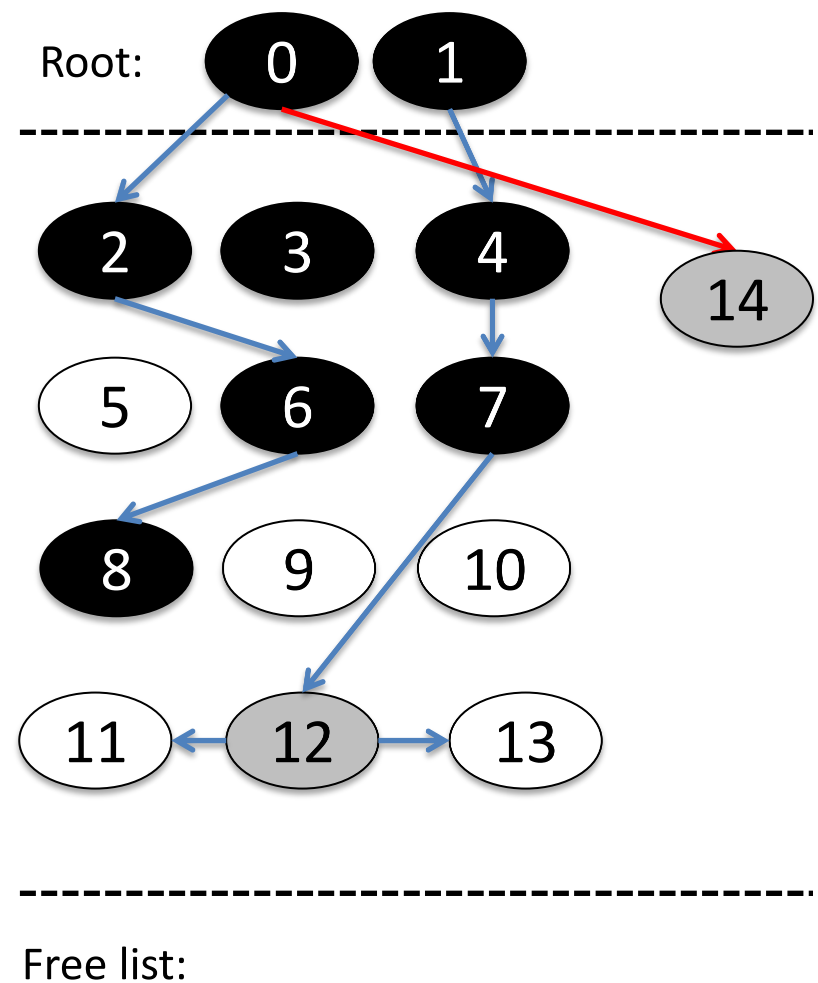

Green simple atomic operations.

We will try to break the red.

# The Collector: The Marking Phase

1. Shade all roots.
2. $i \leftarrow 0$
3. $k \leftarrow M$
4. **While $k > 0$**
   1. < $c$ ←color of node #$i$>
   2. **If $c$ ==Gray then**
      1. **"C1": <Shade the successors of node #$i$ and make node #$i$ black>**
   3. **Else**
      1. $k \leftarrow k - 1$
   4. $i \leftarrow (i + 1)\%M$

$M$ is the nodes count in the graph



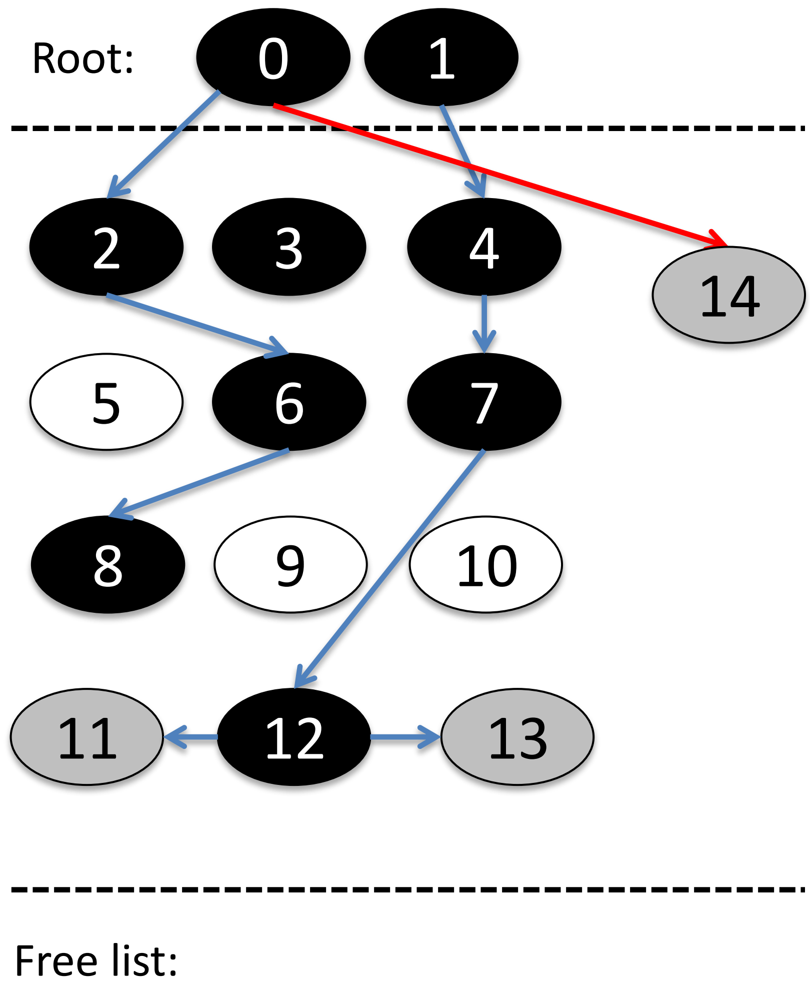

Green simple atomic operations.

We will try to break the red.

# The Collector: The Marking Phase

1. Shade all roots.
2. $i \leftarrow 0$
3. $k \leftarrow M$
4. **While $k > 0$**
    1. **< $c$ ←color of node #$i$>**
    2. **If $c$ ==Gray then**
        1. **"C1": <Shade the successors of node #$i$ and make node #$i$ black>**
    3. **Else**
        1. $\boldsymbol{k \leftarrow k - 1}$
    4. $\boldsymbol{i \leftarrow (i + 1)\%M}$

$M$ is the nodes count in the graph



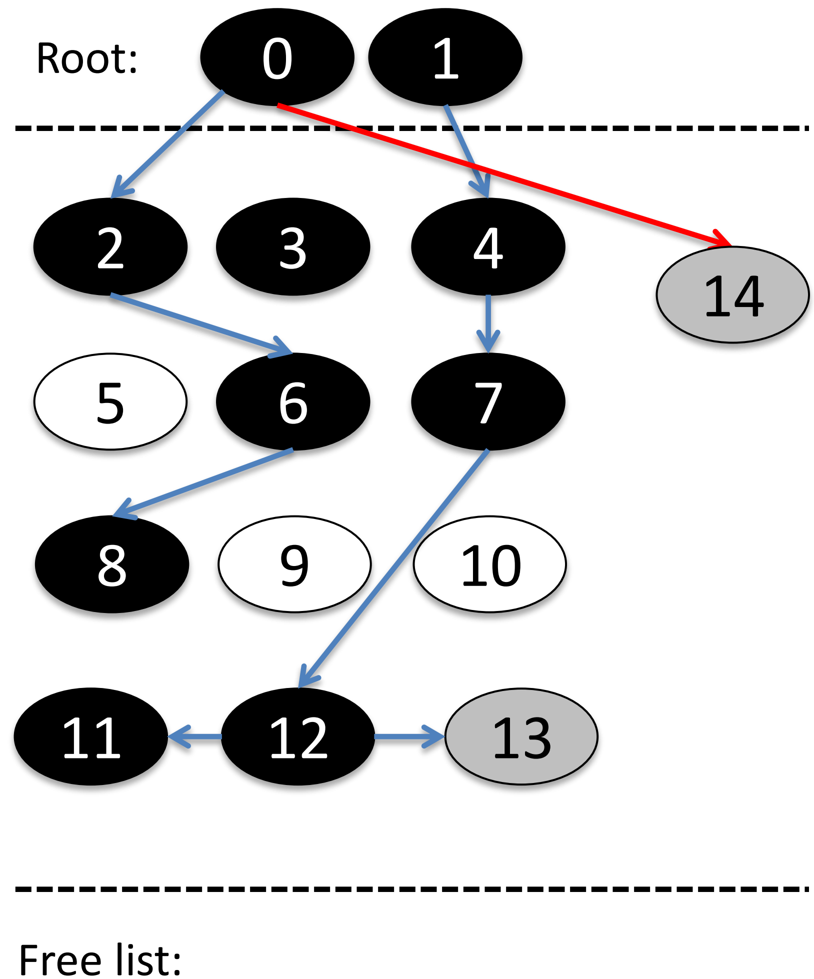

Green simple atomic operations.

We will try to break the red.

# The Collector: The Marking Phase

1. Shade all roots.
2. $i \leftarrow 0$
3. $k \leftarrow M$
4. **While $k > 0$**
   1. **< $c$ ←color of node #$i$>**
   2. **If $c$ ==Gray then**
      1. **"C1": <Shade the successors of node #$i$ and make node #$i$ black>**
   3. **Else**
      1. $k \leftarrow k - 1$
   4. $i \leftarrow (i + 1)\%M$

$M$ is the nodes count in the graph



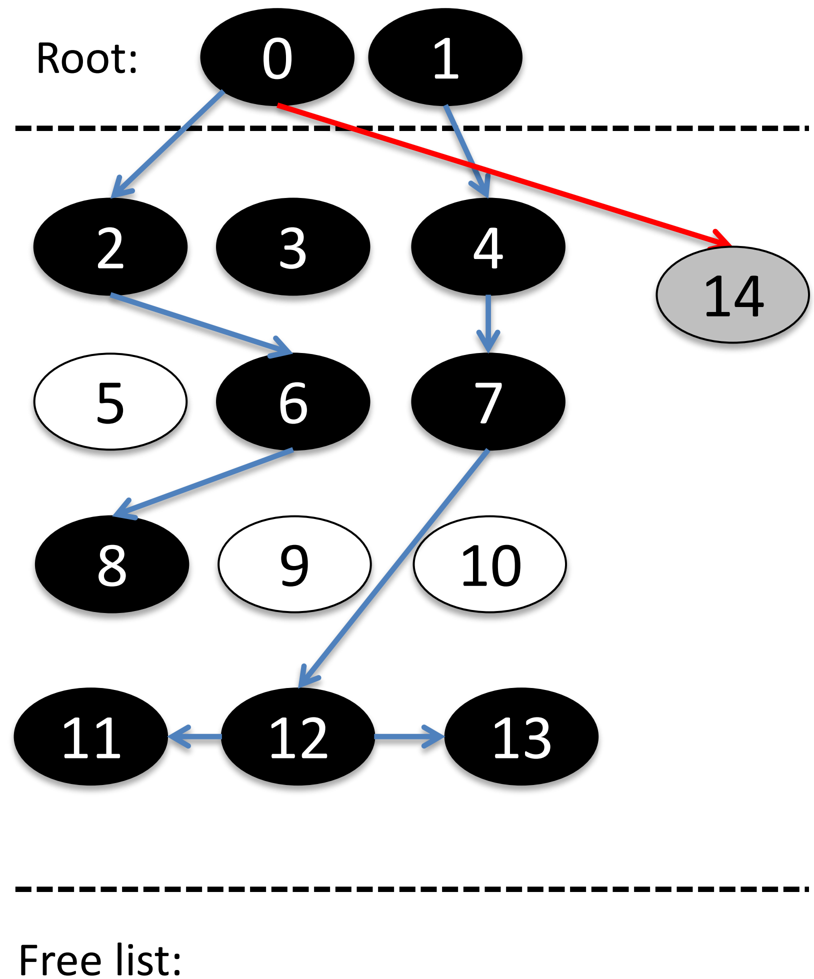

Green simple atomic operations.

We will try to break the red.

# The Collector: The Marking Phase

1. Shade all roots.
2. $i \leftarrow 0$
3. $k \leftarrow M$
4. **While $k > 0$**
    1. **< $c$ ←color of node #$i$>**
    2. **If $c$ ==Gray then**
        1. **“C1”: <Shade the successors of node #$i$ and make node #$i$ black>**
    3. **Else**
        1. $k \leftarrow k - 1$
    4. $i \leftarrow (i + 1)\%M$

$M$ is the nodes count in the graph



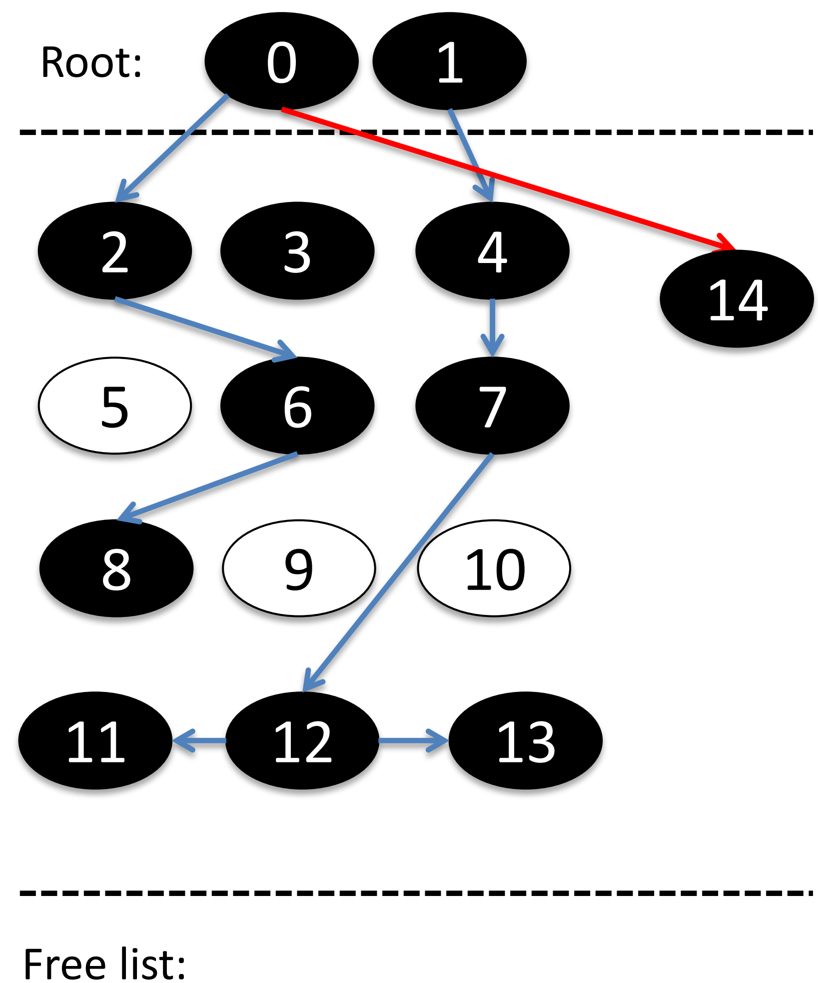

Green simple atomic operations.

We will try to break the red.

# The Collector: The Marking Phase

1. Shade all roots.
2. $i \leftarrow 0$
3. $k \leftarrow M$
4. **While $k > 0$**
    1. < $c$ ←color of node #$i$>
    2. **If $c$ ==Gray then**
        1. **“C1”: <Shade the successors of node #$i$ and make node #$i$ black>**
    3. **Else**
        1. $k \leftarrow k - 1$
    4. $i \leftarrow (i + 1)\%M$

$M$ is the nodes count in the graph




Green simple atomic operations.

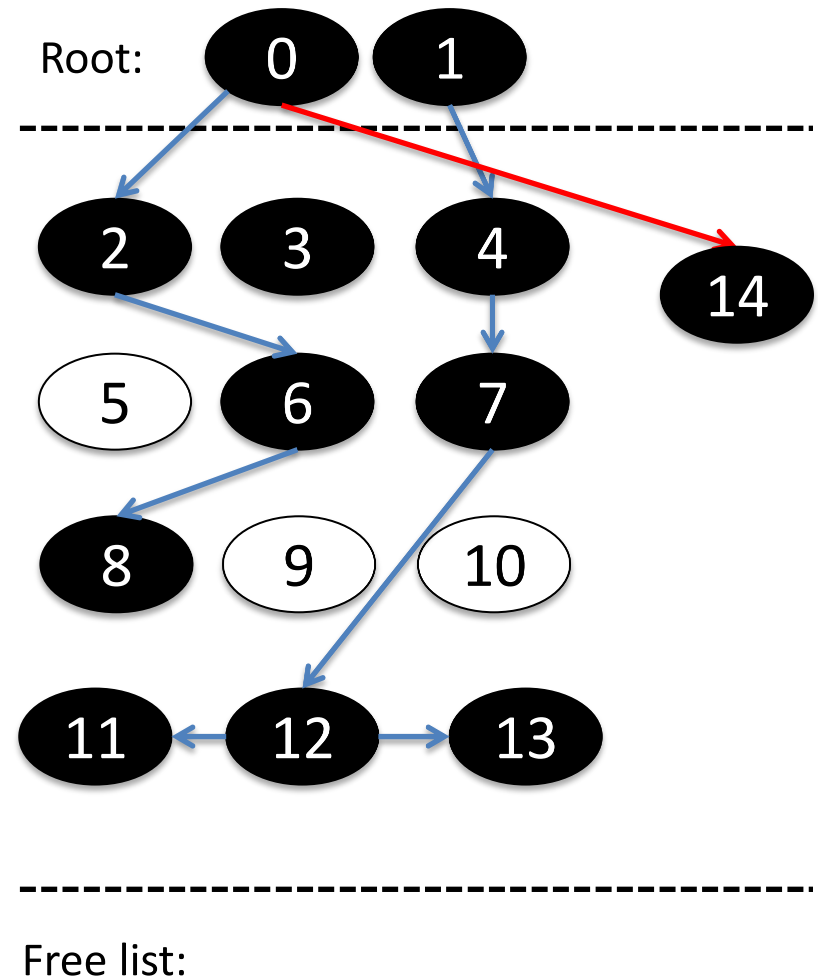
We will try to break the red.

# The Collector: The Appending Phase

1. While $i < M$
   1. < $c$ ←color of node #$i$>
   2. If $c$ ==WHITE then
      1. <Append node #$i$ to the free list>
   3. Else if $c$ == BLACK then
      1. <make node #$i$ white>
   4. $i++$

Green simple atomic operations.

We will try to break the red.

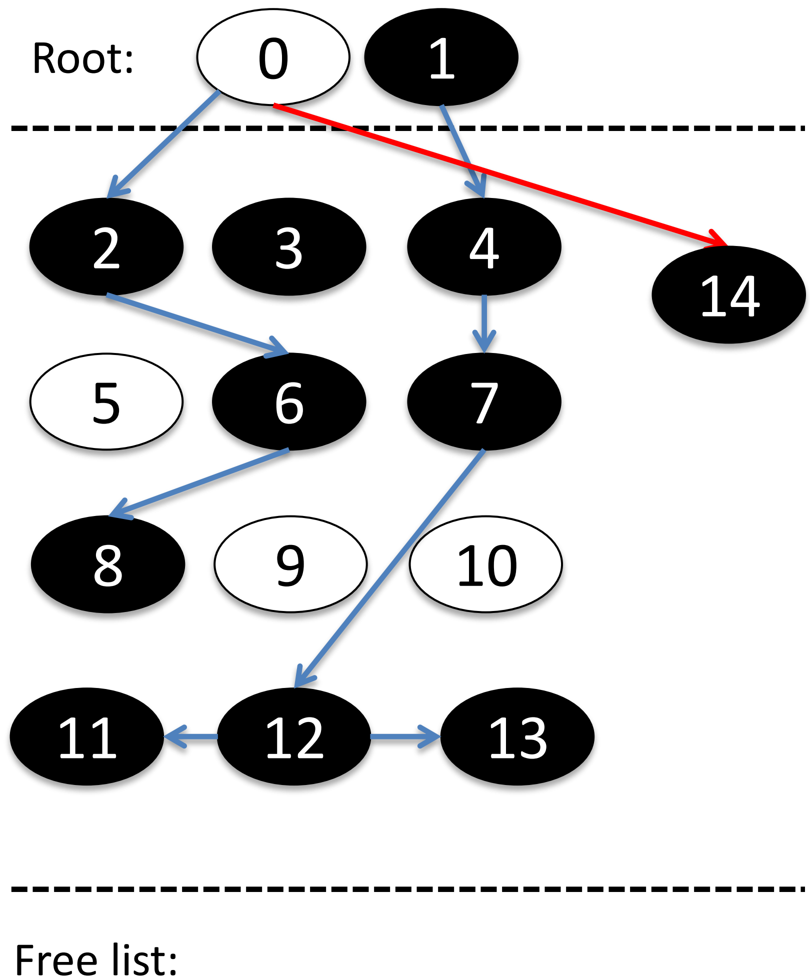


# The Collector: The Appending Phase

1. While $i < M$
   1. < $c$ ←color of node #$i$>
   2. If $c$ ==WHITE then
      1. <Append node #$i$ to the free list>
   3. Else if $c$ == BLACK then
      1. **<make node #$i$ white>**
   4. $i++$

Green simple atomic operations.

We will try to break the red.

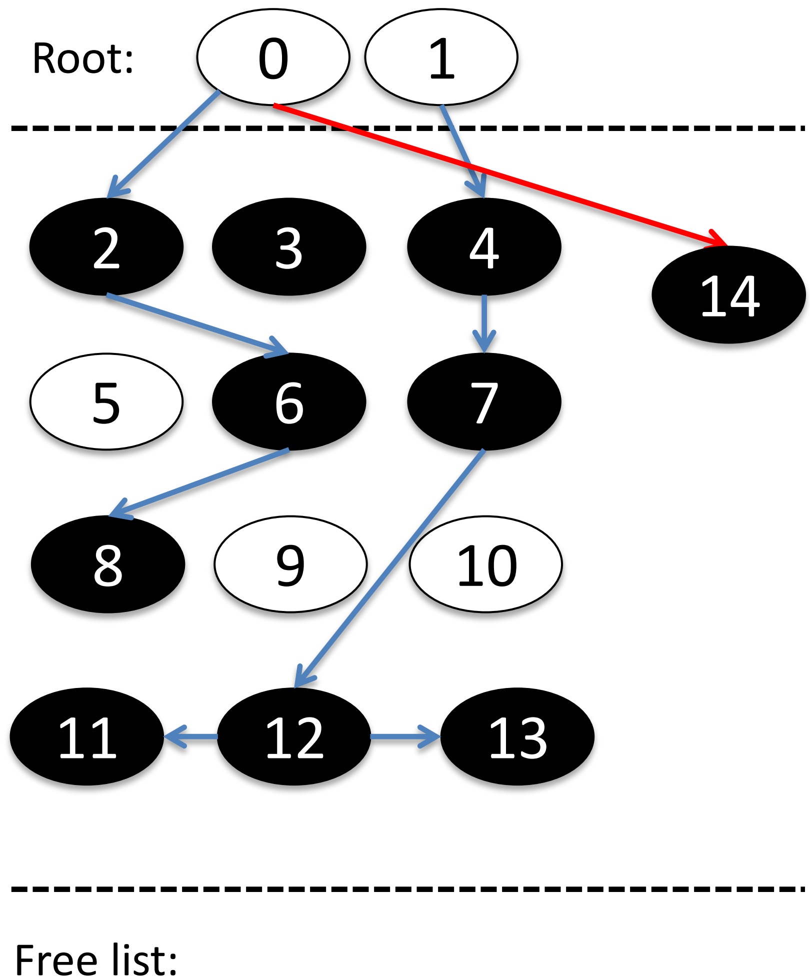


# The Collector: The Appending Phase

1. While $i < M$
    1. < $c$ $\leftarrow$color of node #$i$>
    2. If $c$ ==WHITE then
        1. <Append node #$i$ to the free list>
    3. Else if $c$ == BLACK then
        1. **<make node #$i$ white>**
    4. $i++$

Green simple atomic operations.

We will try to break the red.

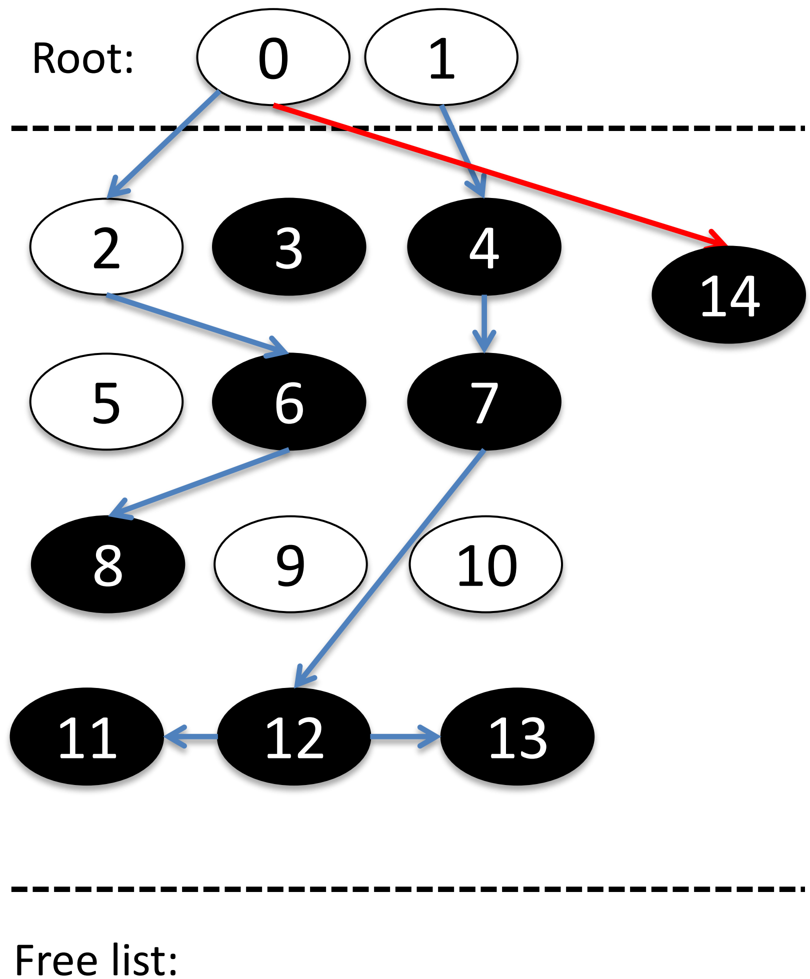


# The Collector: The Appending Phase

1. While $i < M$
   1. $< c \leftarrow$color of node $\#i>$
   2. If $c ==$WHITE then
      1. <Append node $\#i$ to the free list>
   3. Else if $c ==$ BLACK then
      1. **<make node $\#i$ white>**
   4. $i++$

Green simple atomic operations.

We will try to break the red.

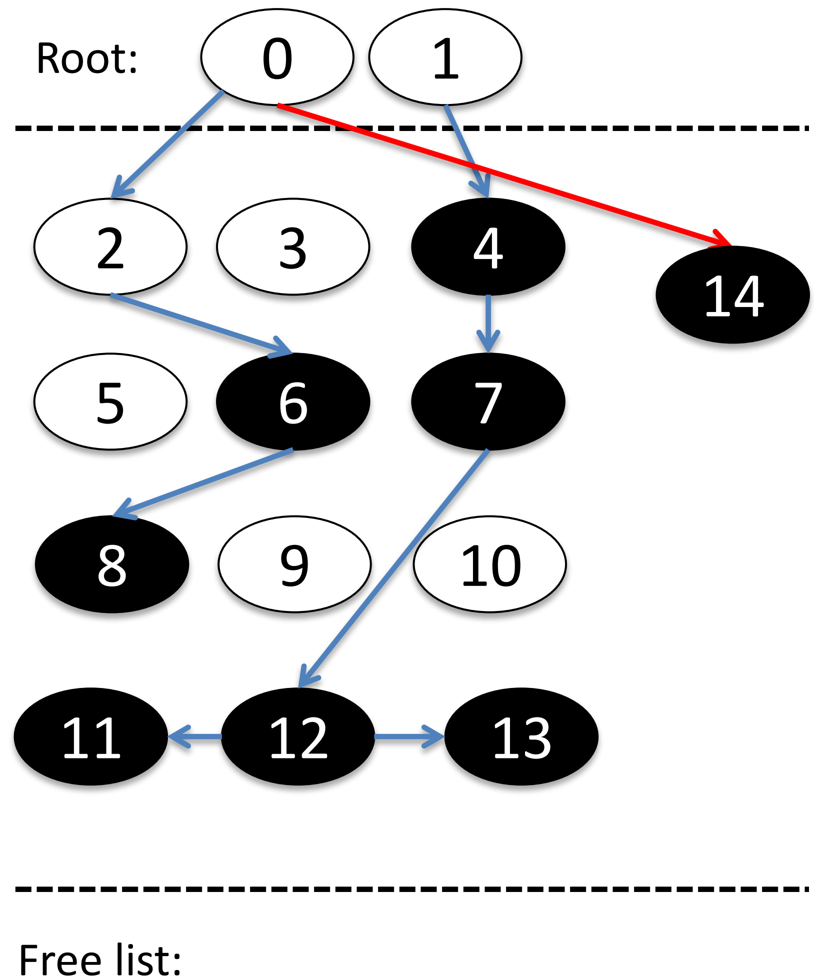


# The Collector: The Appending Phase

1. While $i < M$
   1. < $c$ ←color of node #$i$>
   2. If $c$ ==WHITE then
      1. <Append node #$i$ to the free list>
   3. Else if $c ==$ BLACK then
      1. **<make node #$i$ white>**
   4. $i++$

Green simple atomic operations.

We will try to break the red.

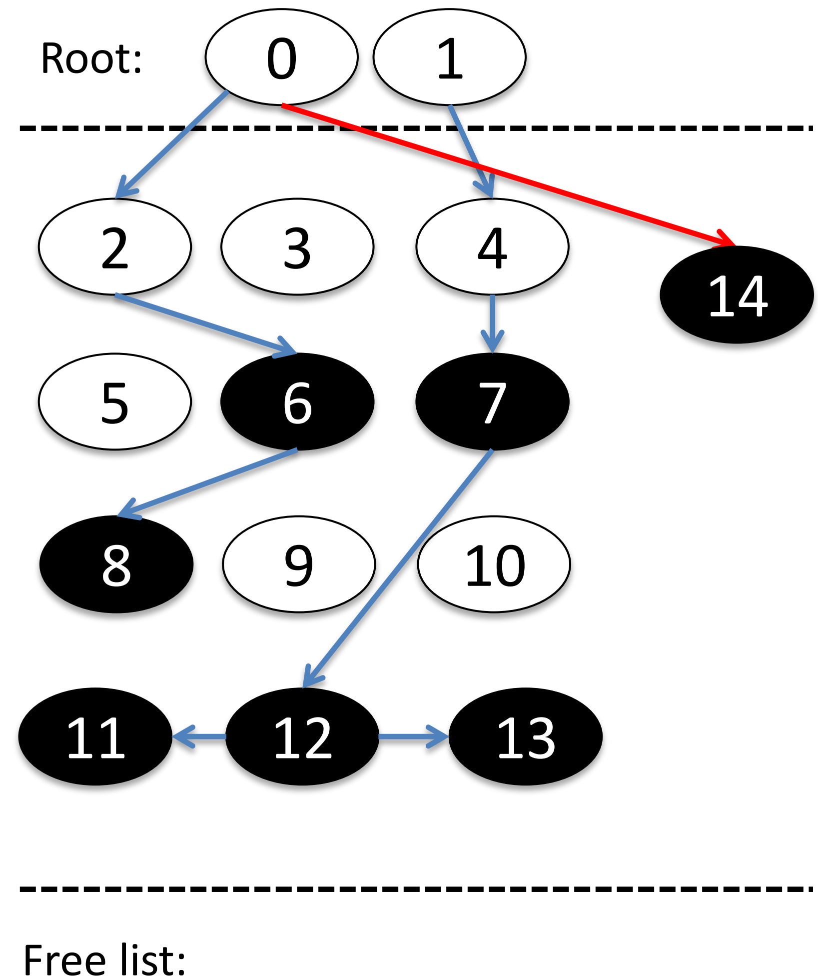


# The Collector: The Appending Phase

1. While $i < M$
   1. < $c \leftarrow$color of node #$i$>
   2. If $c$ ==WHITE then
      1. <Append node #$i$ to the free list>
   3. Else if $c$ == BLACK then
      1. **<make node #$i$ white>**
   4. $i++$

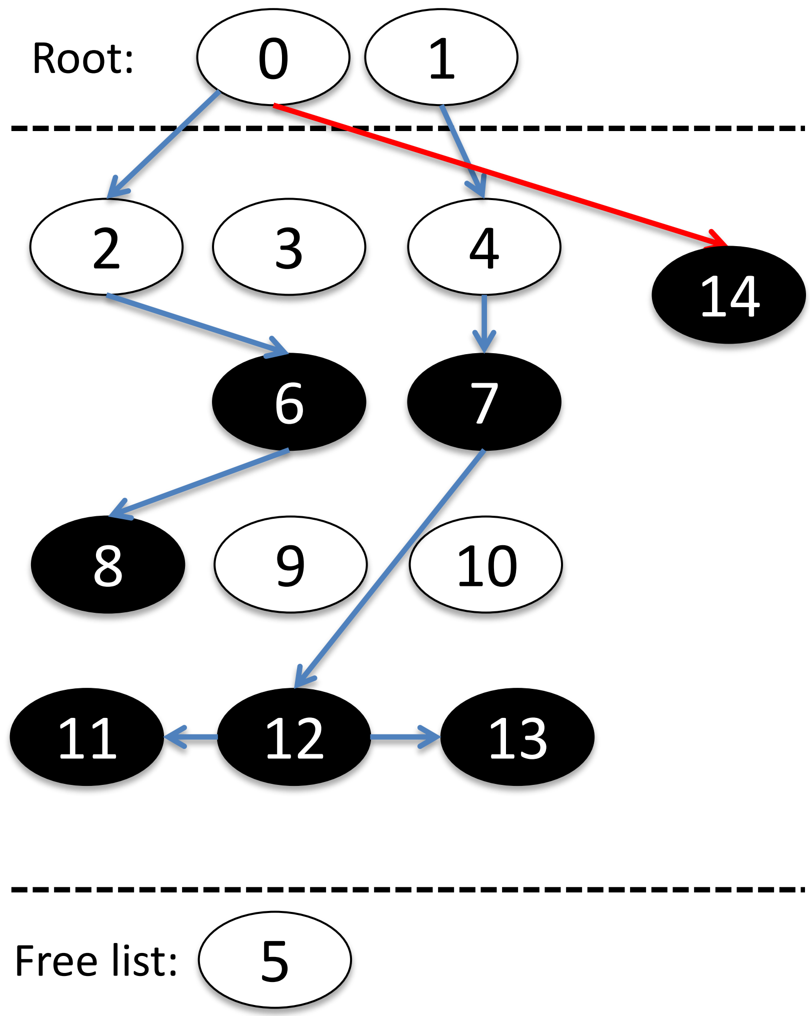
Green simple atomic operations.

We will try to break the red.



# The Collector: The Appending Phase

1. While $i < M$
   1. $< c \leftarrow$color of node $\#i>$
   2. If $c$ ==WHITE then
      1. **$<$Append node $\#i$ to the free list$>$**
   3. Else if $c ==$ BLACK then
      1. $<$make node $\#i$ white$>$
   4. $i++$




Green simple atomic operations.

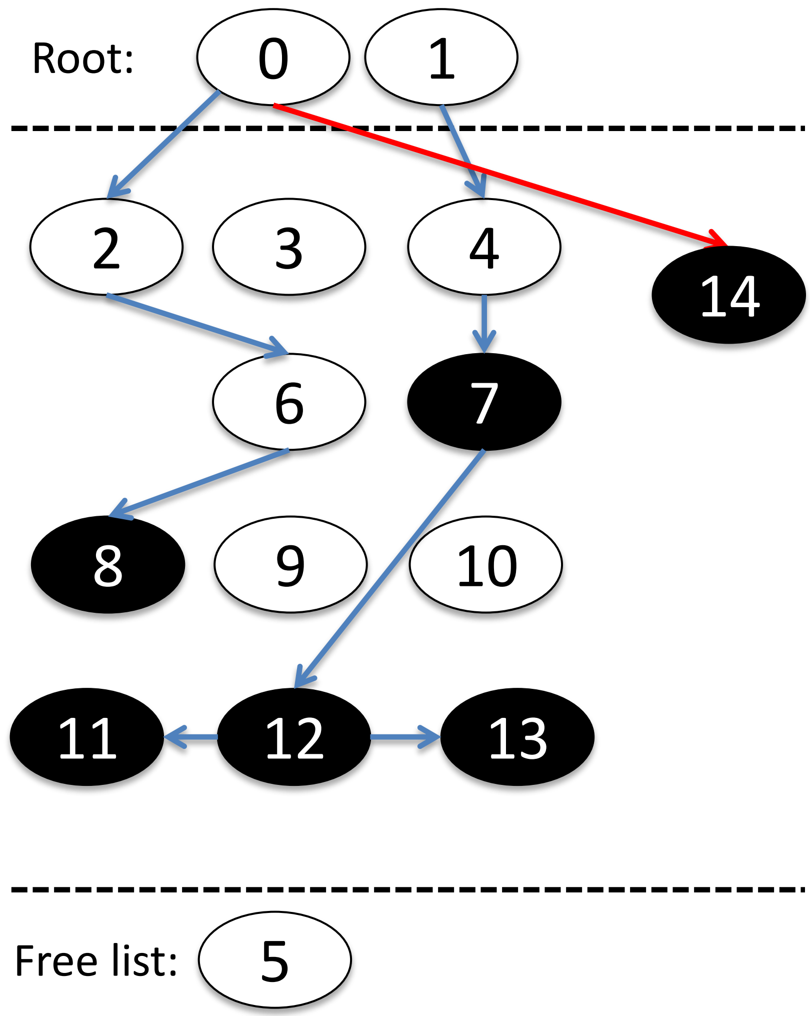
We will try to break the red.

# The Collector: The Appending Phase

1. While $i < M$
   1. < $c$ ←color of node #$i$>
   2. If $c$ ==WHITE then
      1. <Append node #$i$ to the free list>
   3. Else if $c$ == BLACK then
      1. **<make node #$i$ white>**
   4. $i++$

Green simple atomic operations.

We will try to break the red.

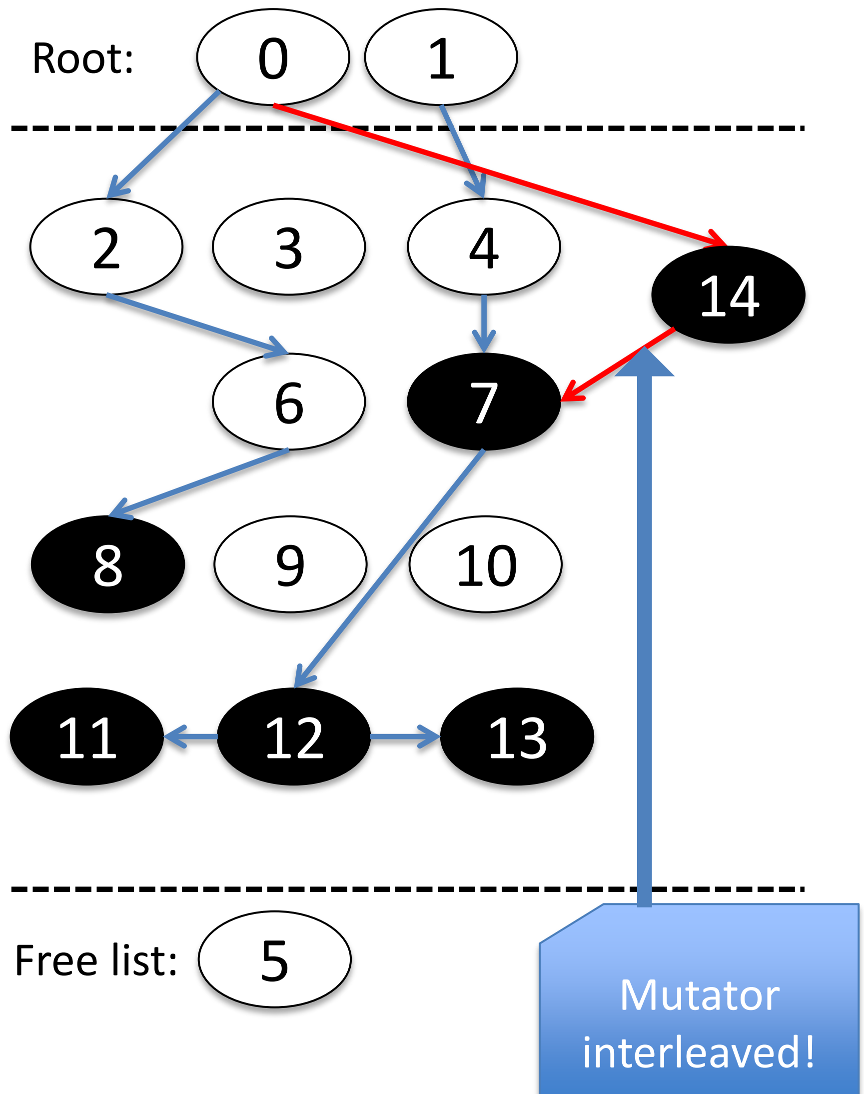


# The Collector: The Appending Phase

1. While $i < M$
   1. $< c \leftarrow$color of node $\#i>$
   2. If $c$ ==WHITE then
      1. <Append node $\#i$ to the free list>
   3. Else if $c ==$ BLACK then
      1. <make node $\#i$ white>
   4. $i++$

Green simple atomic operations.

We will try to break the red.

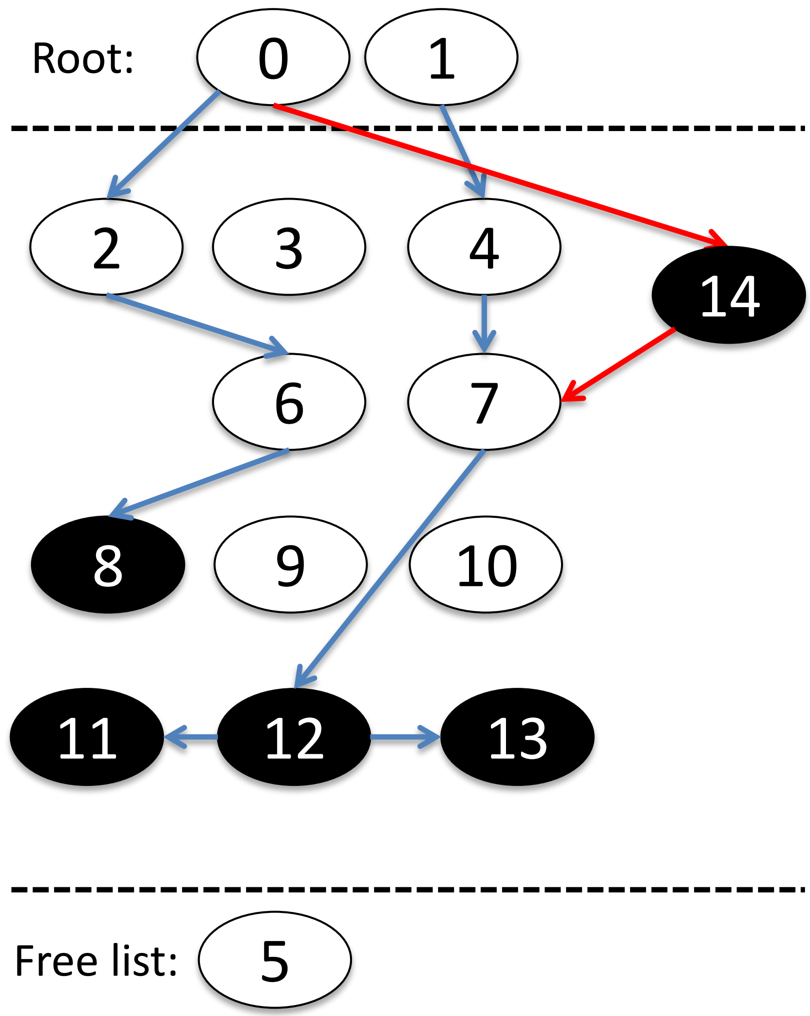


# The Collector: The Appending Phase

1. While $i < M$
   1. < $c$ ←color of node #$i$>
   2. If $c$ ==WHITE then
      1. <Append node #$i$ to the free list>
   3. Else if $c$ == BLACK then
      1. **<make node #$i$ white>**
   4. $i++$

Green simple atomic operations.

We will try to break the red.

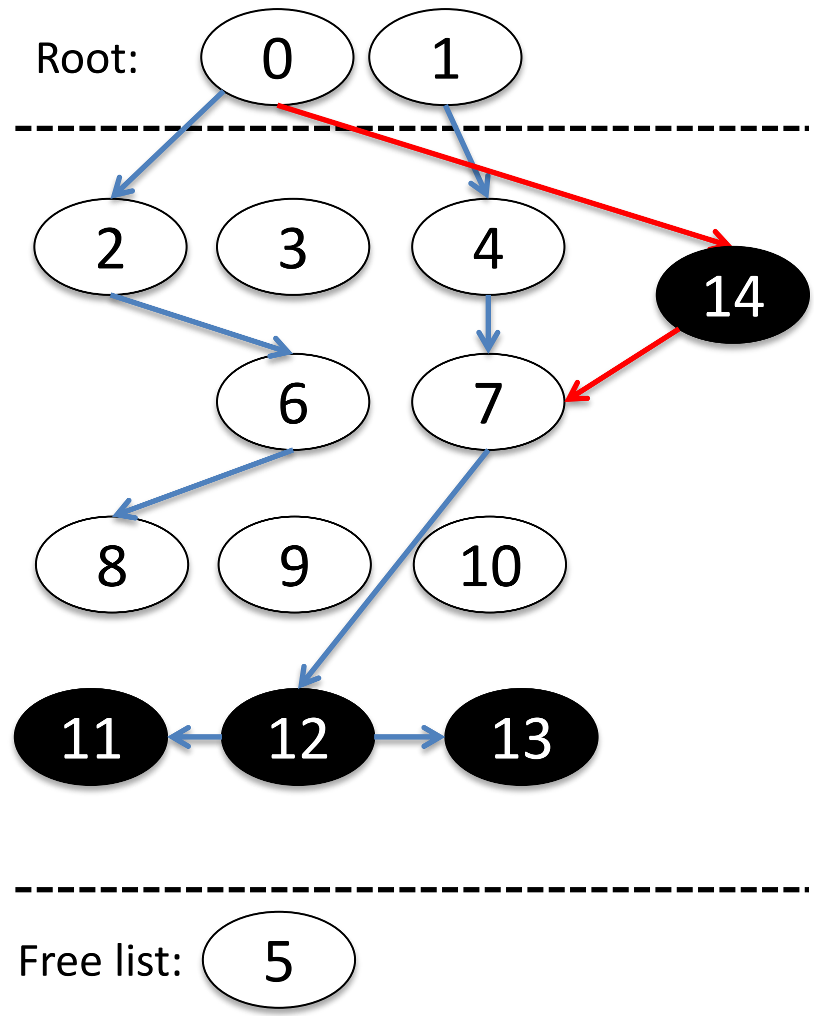


# The Collector: The Appending Phase

1. While $i < M$
   1. < $c \leftarrow$color of node #$i$>
   2. If $c ==$WHITE then
      1. <Append node #$i$ to the free list>
   3. Else if $c ==$ BLACK then
      1. **<make node #$i$ white>**
   4. $i++$

Green simple atomic operations.

We will try to break the red.

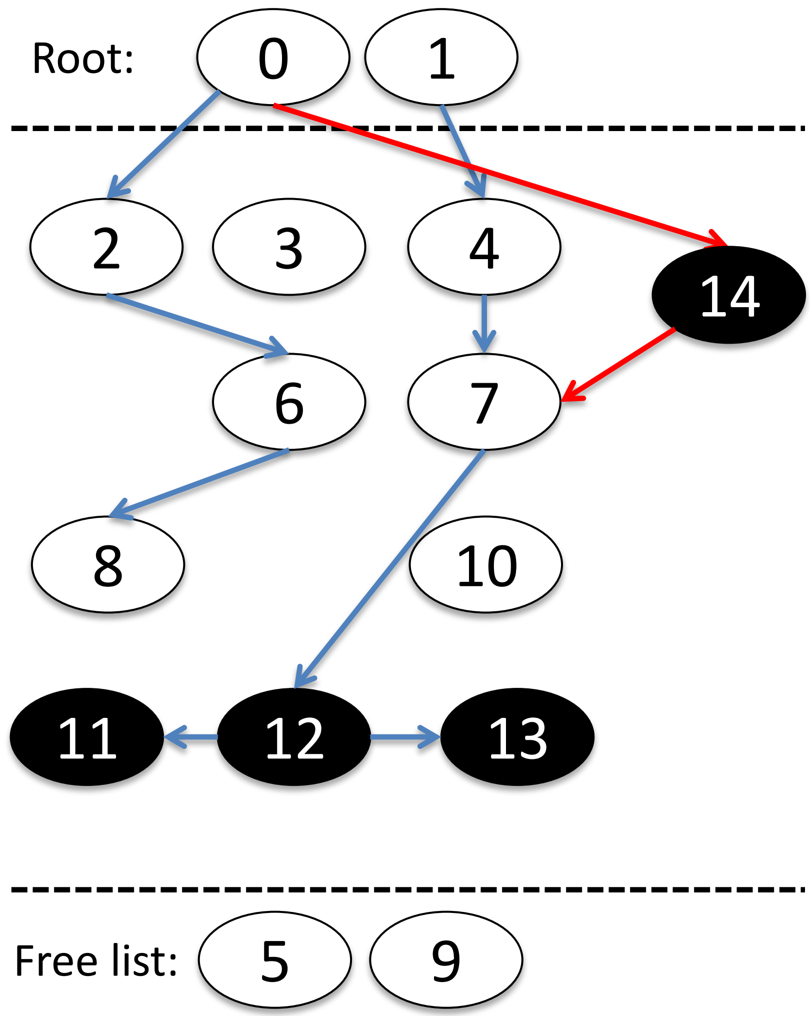


# The Collector: The Appending Phase

1. While $i < M$
    1. < $c$ $\leftarrow$color of node #$i$>
    2. If $c$ ==WHITE then
        1. **<Append node #$i$ to the free list>**
    3. Else if $c$ == BLACK then
        1. <make node #$i$ white>
    4. $i++$

Green simple atomic operations.

We will try to break the red.

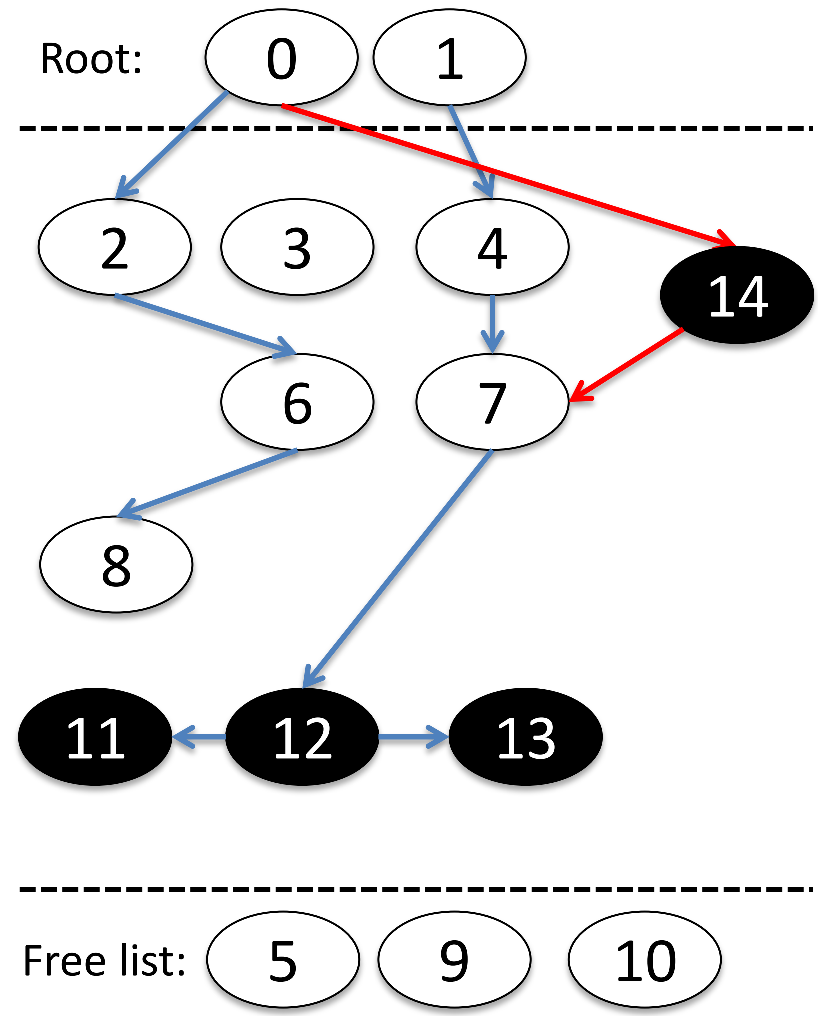


# The Collector: The Appending Phase

1. While $i < M$
    1. < $c$ ←color of node #$i$>
    2. If $c$ ==WHITE then
        1. **<Append node #$i$ to the free list>**
    3. Else if $c$ == BLACK then
        1. <make node #$i$ white>
    4. $i++$

Green simple atomic operations.

We will try to break the red.

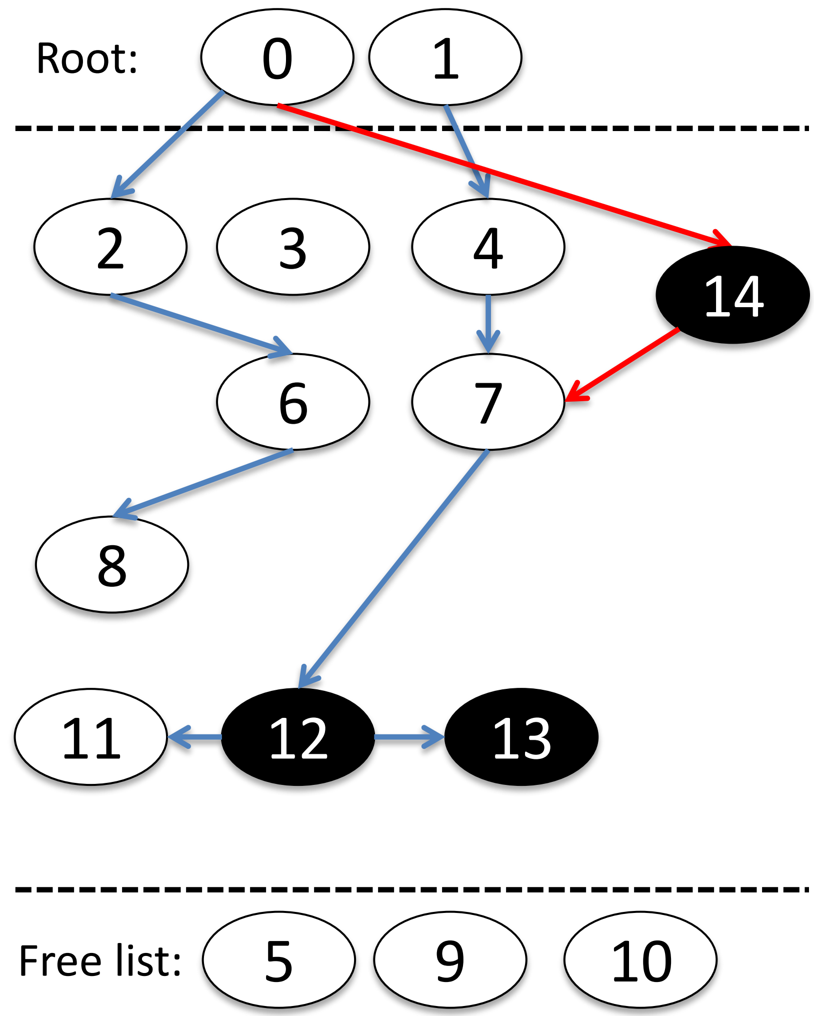


# The Collector: The Appending Phase

1. While $i < M$
   1. < $c \leftarrow$color of node #$i$>
   2. If $c$ ==WHITE then
      1. <Append node #$i$ to the free list>
   3. Else if $c ==$ BLACK then
      1. **<make node #$i$ white>**
   4. $i++$

Green simple atomic operations.

We will try to break the red.

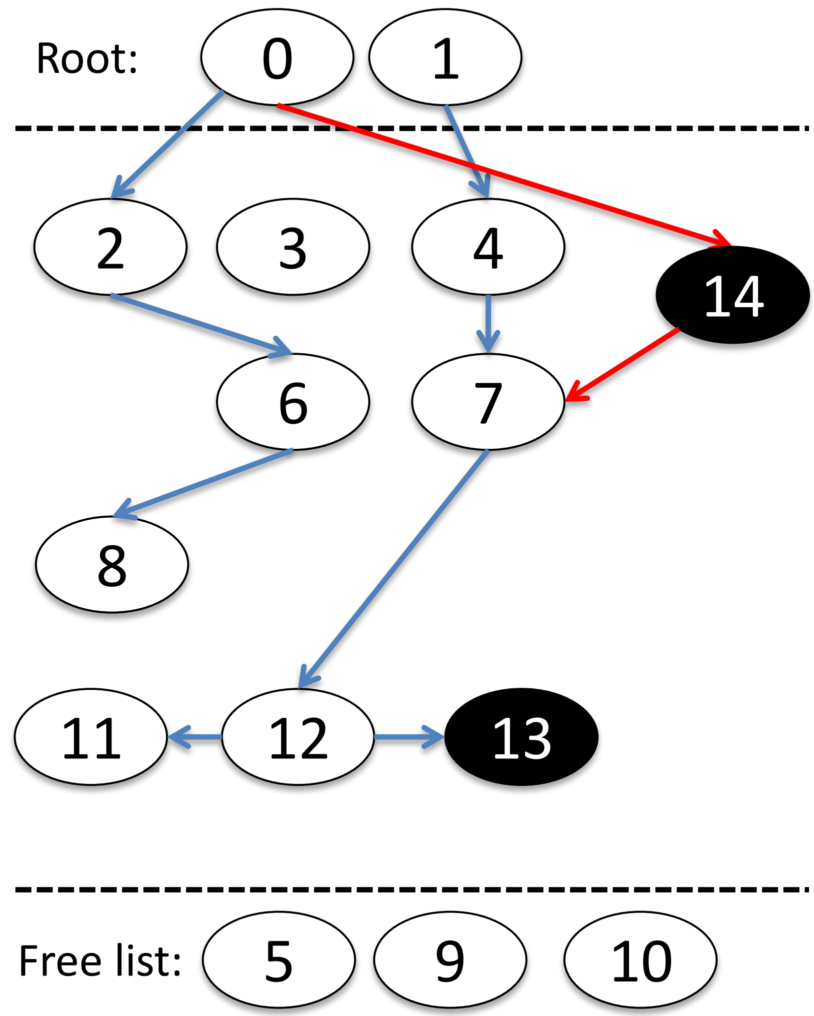


# The Collector: The Appending Phase

1. While $i < M$
    1. < $c \leftarrow$ color of node #$i$>
    2. If $c ==$ WHITE then
        1. <Append node #$i$ to the free list>
    3. Else if $c ==$ BLACK then
        1. **<make node #$i$ white>**
    4. $i++$

Green simple atomic operations.

We will try to break the red.

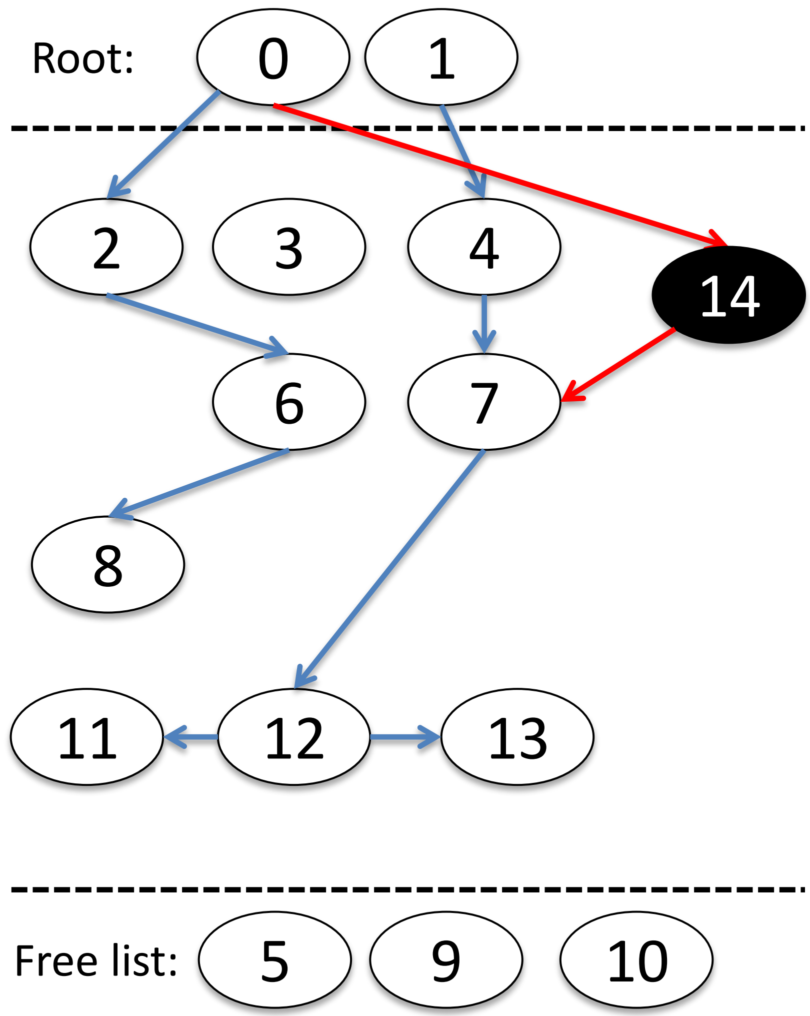


# The Collector: The Appending Phase

1. While $i < M$
    1. < $c$ ←color of node #$i$>
    2. If $c$ ==WHITE then
        1. <Append node #$i$ to the free list>
    3. Else if $c$ == BLACK then
        1. **<make node #$i$ white>**
    4. $i++$

Green simple atomic operations.

We will try to break the red.

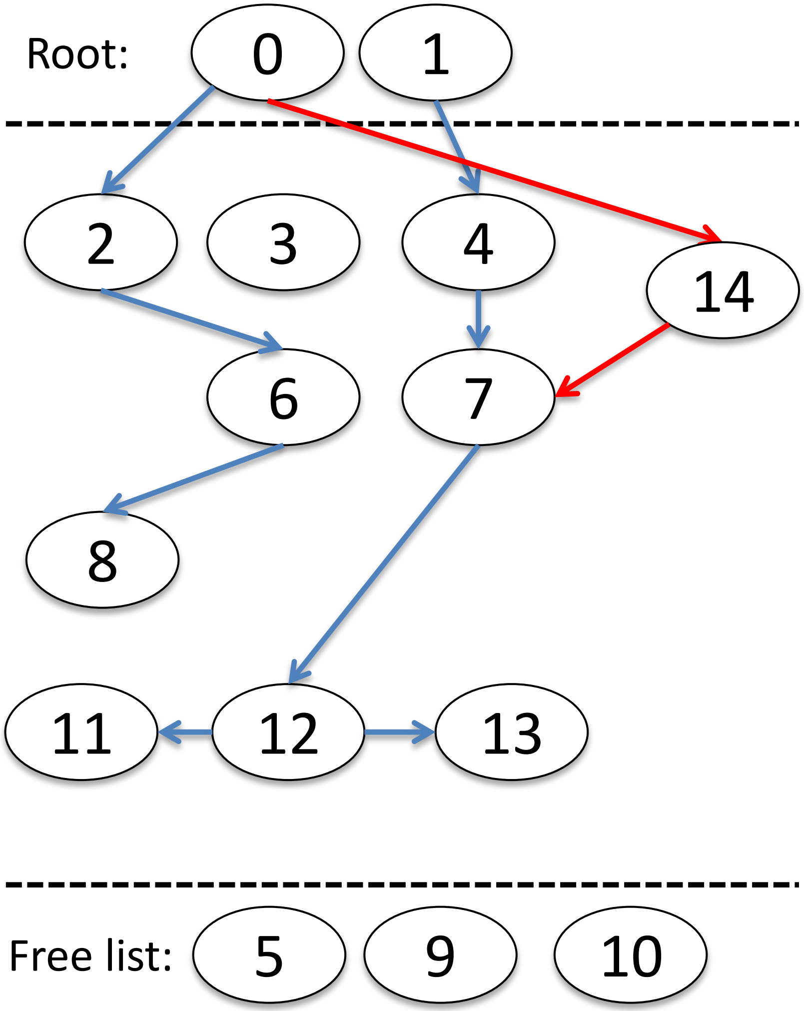


# The Collector: The Appending Phase

1. While $i < M$
    1. < $c$ ←color of node #$i$>
    2. If $c$ ==WHITE then
        1. <Append node #$i$ to the free list>
    3. Else if $c$ == BLACK then
        1. **<make node #$i$ white>**
    4. $i++$

Green simple atomic operations.

We will try to break the red.

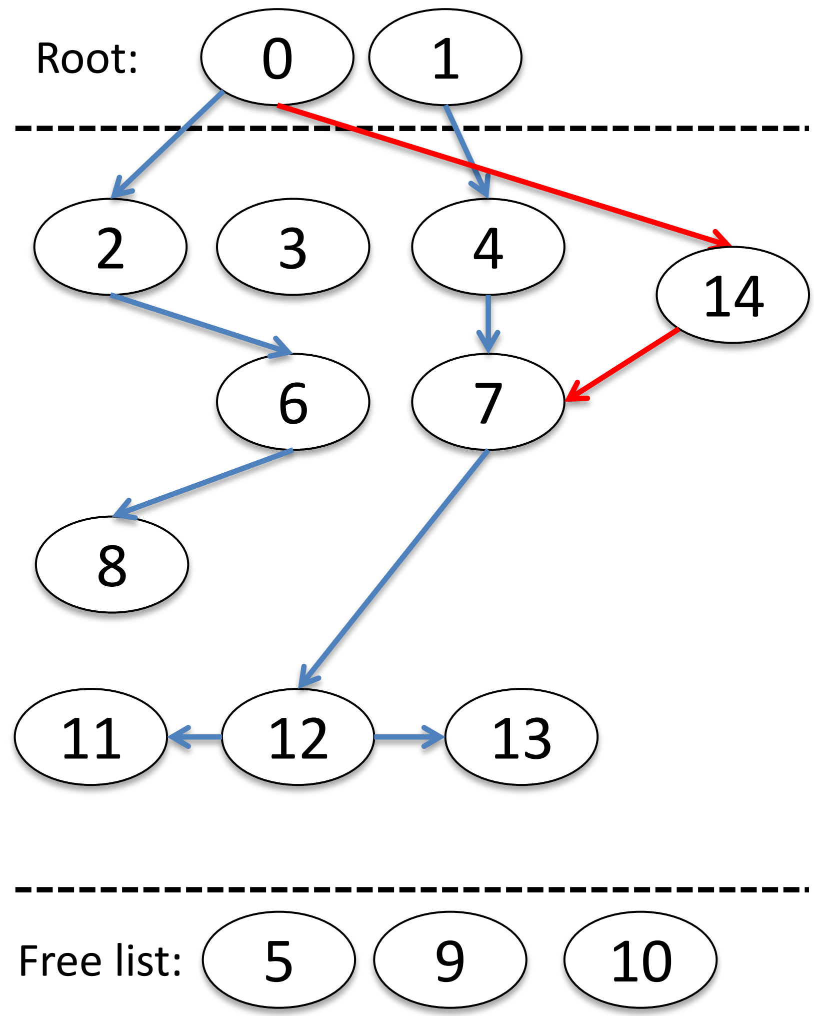


# The Collector: The Marking Phase (2)

1. Shade all roots.
2. $i \leftarrow 0$
3. $k \leftarrow M$
4. While $k > 0$
    1. < $c$ ←color of node #$i$>
    2. If $c$ ==Gray then
        1. “C1”: <Shade the successors of node #$i$ and make node #$i$ black>
    3. Else
        1. $k \leftarrow k - 1$
    4. $i \leftarrow (i + 1)\%M$

Green simple atomic operations.

We will try to break the red.

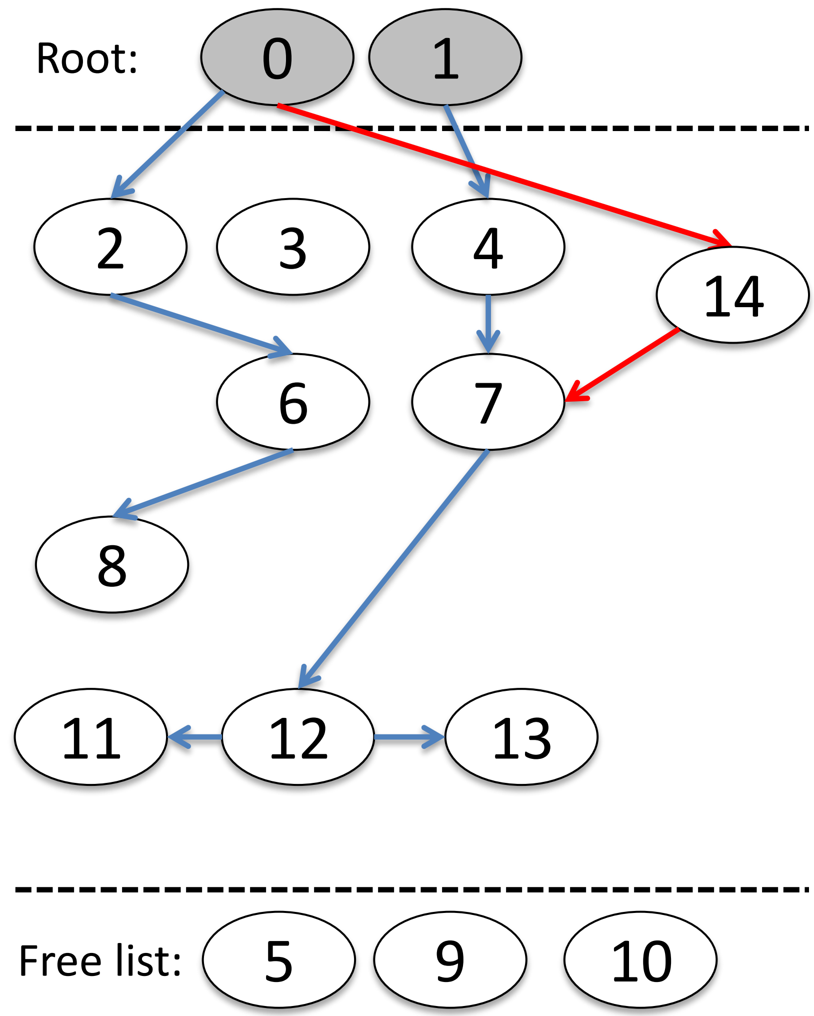


# The Collector: The Marking Phase (2)

1. **Shade all roots.**
2. $i \leftarrow 0$
3. $k \leftarrow M$
4. While $k > 0$
    1. < $c$ ←color of node #$i$>
    2. If $c$ ==Gray then
        1. "C1": <Shade the successors of node #$i$ and make node #$i$ black>
    3. Else
        1. $k \leftarrow k - 1$
    4. $i \leftarrow (i + 1)\%M$

Green simple atomic operations.

We will try to break the red.

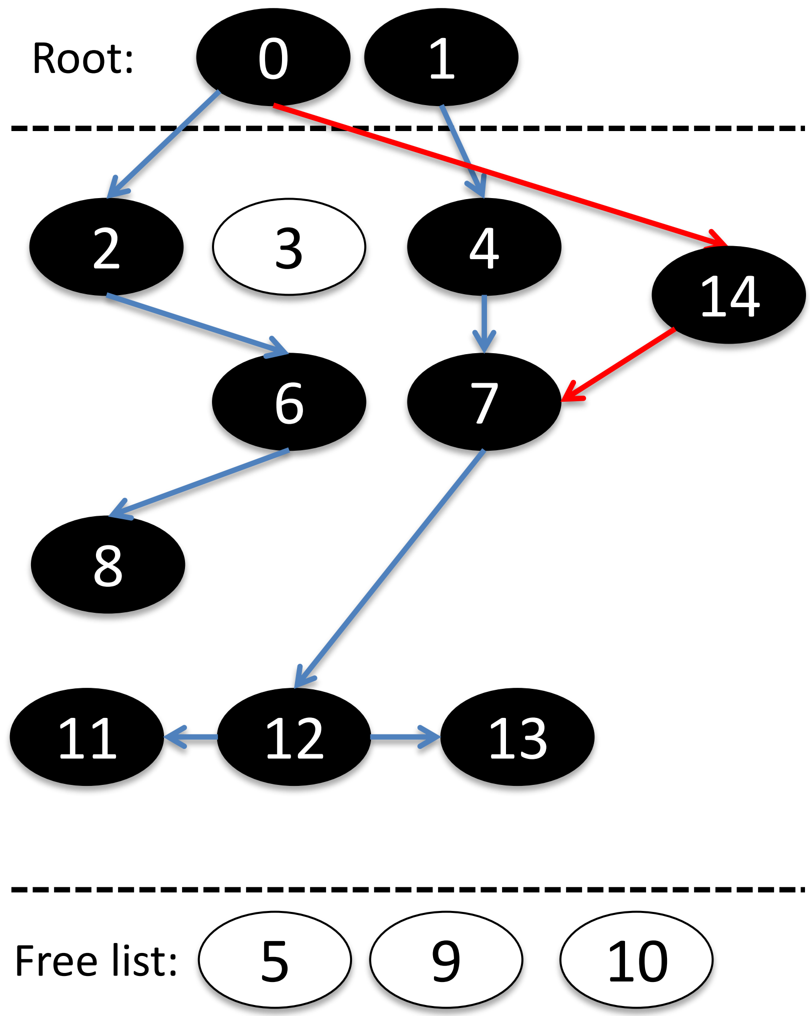


# The Collector: The Marking Phase (2)

1. Shade all roots.
2. $i \leftarrow 0$
3. $k \leftarrow M$
4. **While $k > 0$**
   1. **< $c$ ←color of node #$i$>**
   2. **If $c$ ==Gray then**
      1. **"C1": <Shade the successors of node #$i$ and make node #$i$ black>**
   3. **Else**
      1. $k \leftarrow k - 1$
   4. $i \leftarrow (i + 1)\%M$

Green simple atomic operations.

We will try to break the red.

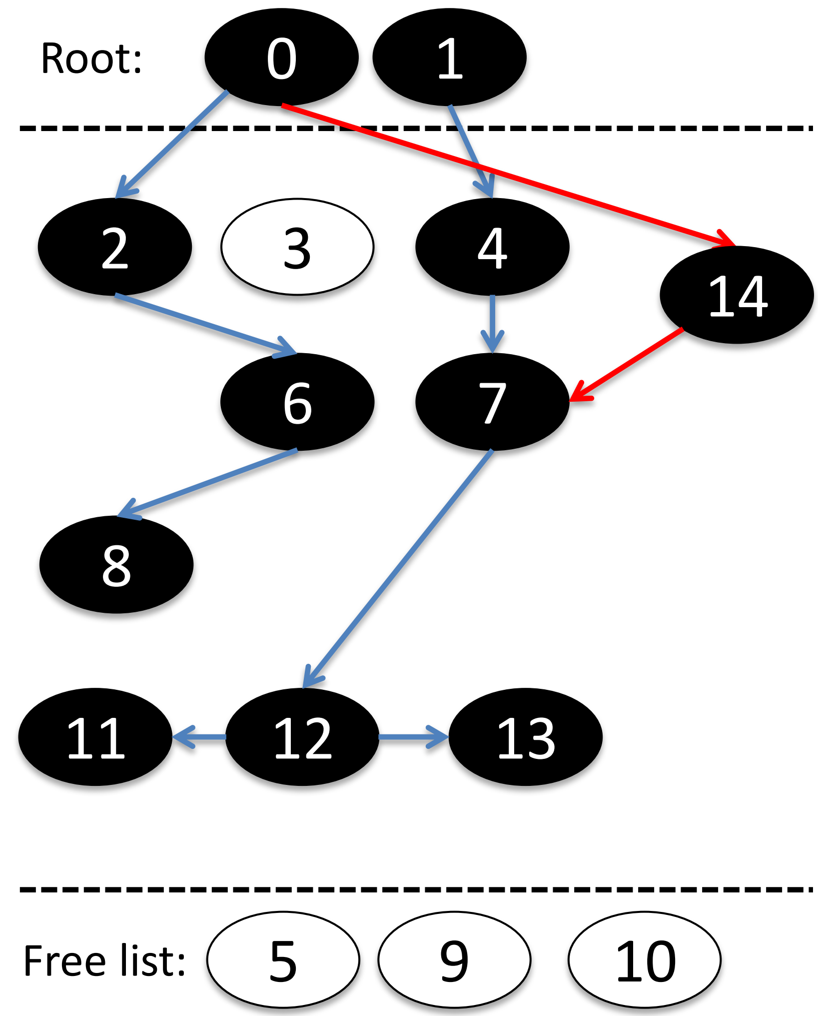


# The Collector: The Appending Phase(2)

1. While $i < M$
   1. < $c$ ←color of node #$i$>
   2. If $c$ ==WHITE then
      1. <Append node #$i$ to the free list>
   3. Else if $c$ == BLACK then
      1. <make node #$i$ white>
   4. $i++$

Green simple atomic operations.

We will try to break the red.

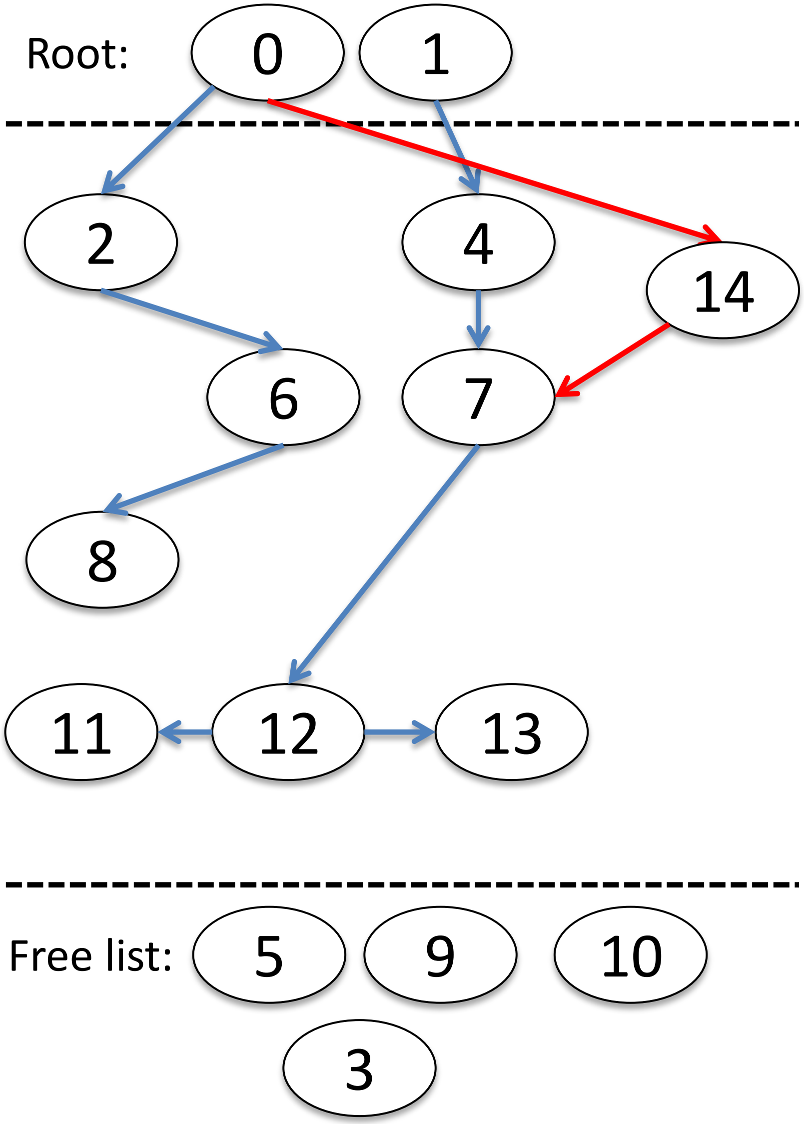


# The Collector: The Appending Phase(2)

1. While $i < M$
    1. < $c$ ←color of node #$i$>
    2. If $c$ ==WHITE then
        1. <Append node #$i$ to the free list>
    3. Else if $c$ == BLACK then
        1. <make node #$i$ white>
    4. $i++$

Green simple atomic operations.

We will try to break the red.



# Proof for Correction Criteria 2

- Reminder for CC2: “Appending a garbage node to the free list is the collector’s only modification of the shape of data structure”.
- The marking phase doesn’t change the data structure.
- Prove the rest by showing that the following invariant holds after the marking phase completes:
  - “A white node with a number $\geq i$ is garbage”

# Proof for Correction Criteria 2 – cont'

- For the first iteration ($i = 0$), this derives from the following observations:
  - The marking phase terminates when there is no gray node.
  - The absence of gray nodes is stable once reached.
  - At the end of the appending phase, there is no black nodes.

# Proof for Correction Criteria 2 – cont’

- For the other iterations ($i > 0$), this derives from the following observations:
  - There are 2 ways to violate the invariance:
    - Making a non-garbage node white.
    - Making a (white) garbage node into non-garbage.
  - The mutator
    - doesn’t convert nodes to white.
    - don’t deal with to white garbage nodes.
  - The collector
    - For the $i$-th iteration, only the $i$-th node may change the color.

# Proof for Correction Criteria 1

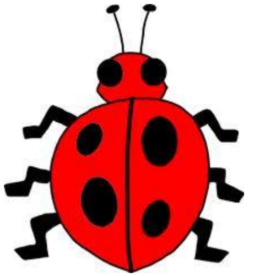
- Reminder for CC1: “Every garbage node is eventually appended to the free list”.
- First we need to prove that both phases terminates correctly.
  - The marking phase terminates because the quantity $\boldsymbol{k} + \boldsymbol{M} * (\boldsymbol{X})$, where $X$ is non-black nodes, decreases by at least 1 for each iteration.
  - The appending phase terminate obviously, and the mutator cannot change the color of the nodes.

# Proof for Correction Criteria 1 – cont'

- At the beginning of the appending phase, the nodes can be partitioned into 3 sets:
  - The set of reachable nodes
    - They are black
  - The set of white garbage nodes
    - Will be appended to the free list in the first appending phase to come
  - The set of black garbage node
    - Will be appended to the free list in the next appending phase to come

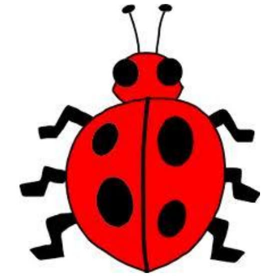
# THE SECOND COARSE-GRAINED SOLUTION

# The BUGGY Proposal

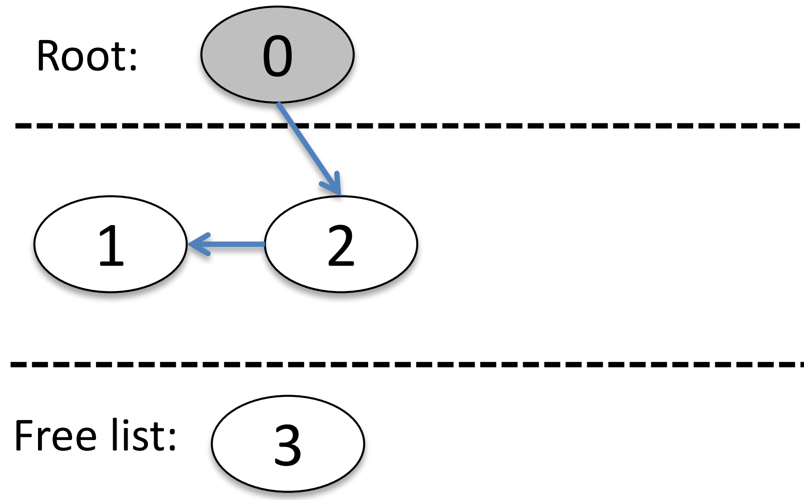


- An attempt to break M1 into 2 atomic operations:
  - <Redirect an outgoing edge of a reachable node towards an already reachable one>
  - <Shade the new target>
- Shading must be the first in order to keep P1!
- A bug was found by Stenning & Woodger.

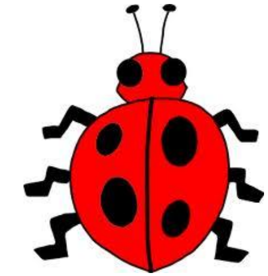
# The BUGGY Proposal - Demo



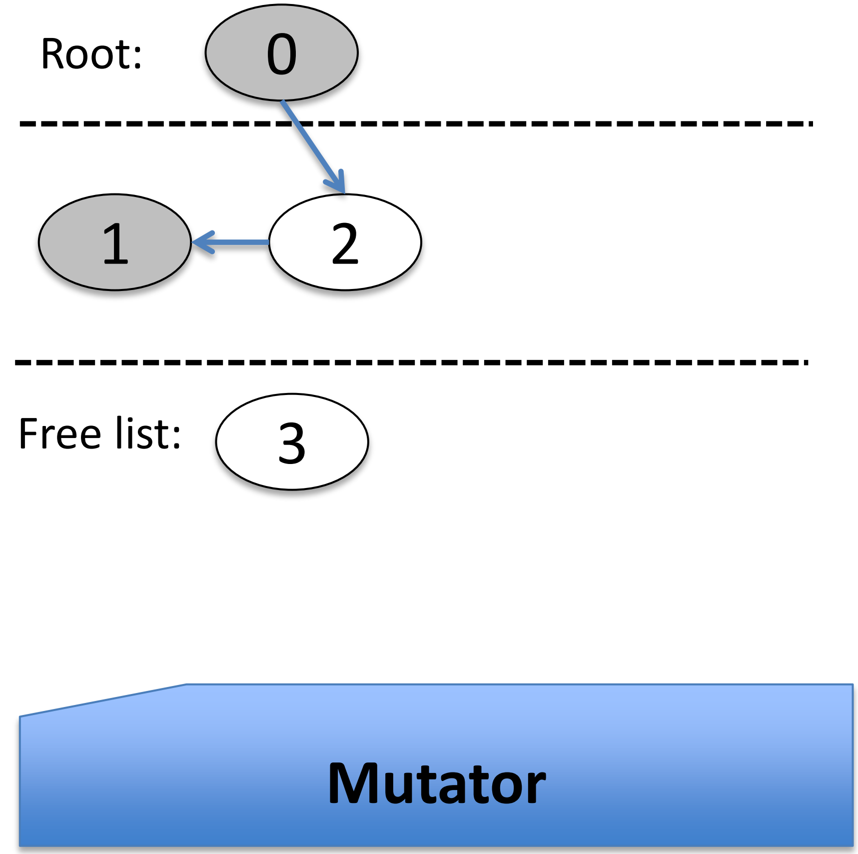
- An attempt to break M1 into 2 atomic operations:
  - <Redirect an outgoing edge of a reachable node towards an already reachable one>
  - <Shade the new target>
- Shading must be the first in order to keep P1!



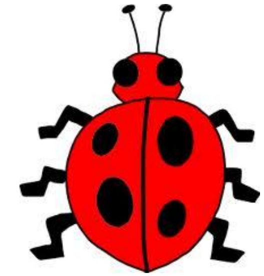
# The BUGGY Proposal - Demo



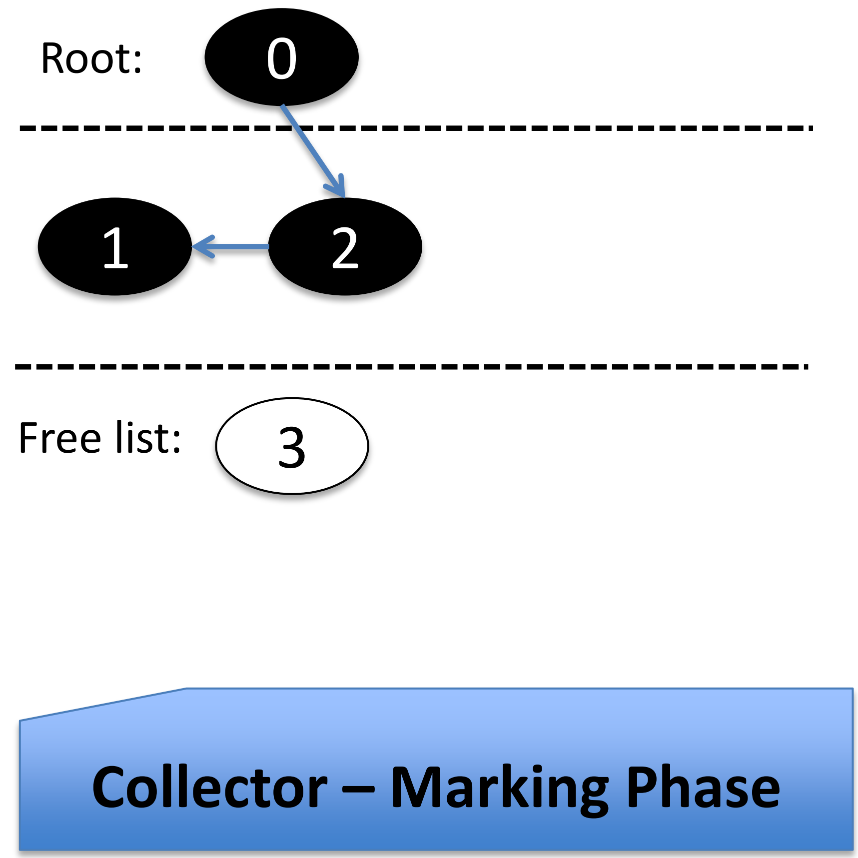
- An attempt to break M1 into 2 atomic operations:
  - <Redirect an outgoing edge of a reachable node towards an already reachable one>
  - **<Shade the new target>**
- Shading must be the first in order to keep P1!



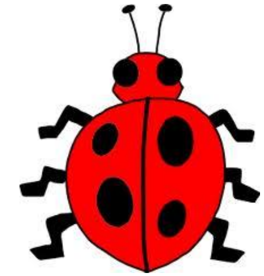
# The BUGGY Proposal - Demo



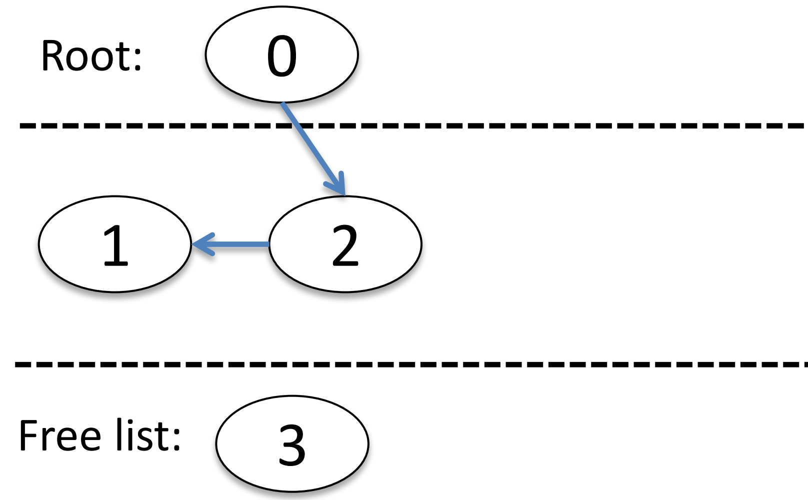
- An attempt to break M1 into 2 atomic operations:
  - <Redirect an outgoing edge of a reachable node towards an already reachable one>
  - <Shade the new target>
- Shading must be the first in order to keep P1!



# The BUGGY Proposal - Demo

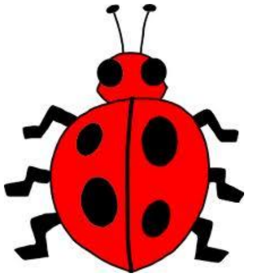


- An attempt to break M1 into 2 atomic operations:
  - <Redirect an outgoing edge of a reachable node towards an already reachable one>
  - <Shade the new target>
- Shading must be the first in order to keep P1!

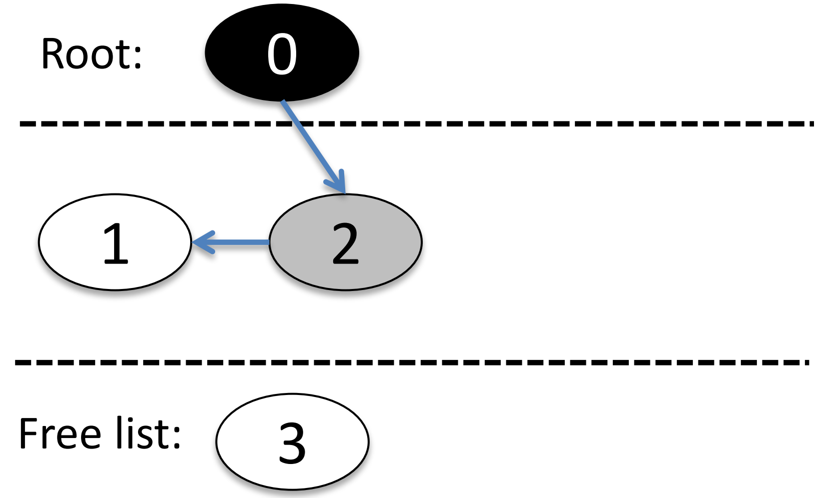



**Collector – Appending Phase**

# The BUGGY Proposal - Demo

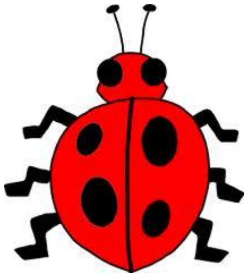


- An attempt to break M1 into 2 atomic operations:
  - <Redirect an outgoing edge of a reachable node towards an already reachable one>
  - <Shade the new target>
- Shading must be the first in order to keep P1!

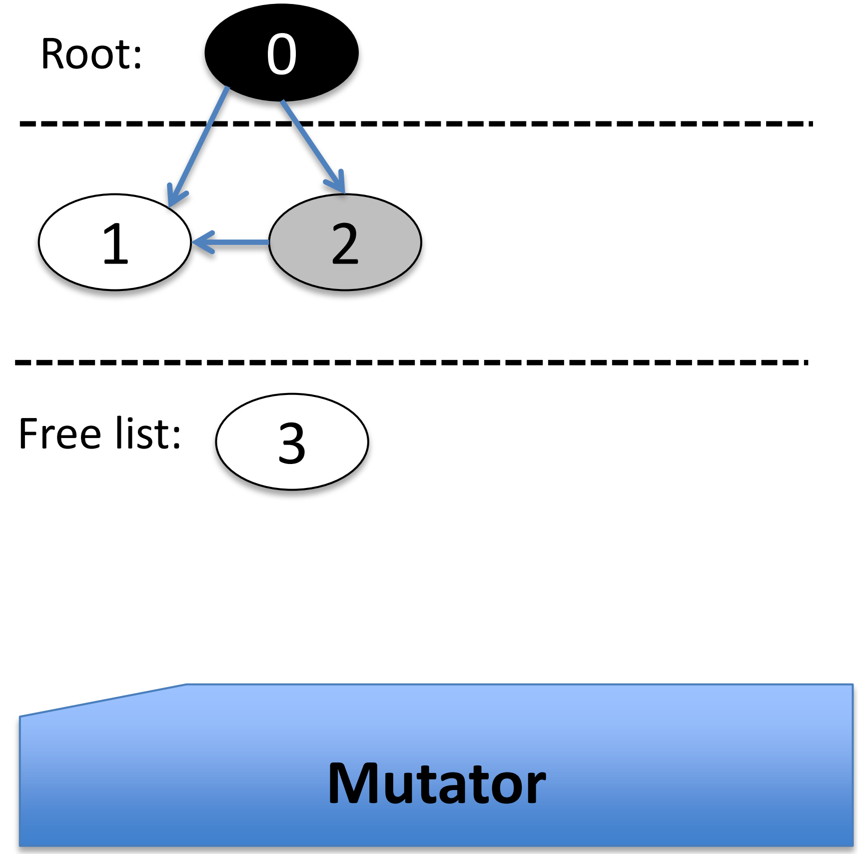



**Collector – Marking Phase**

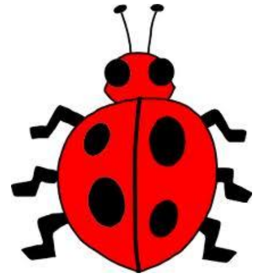
# The BUGGY Proposal - Demo



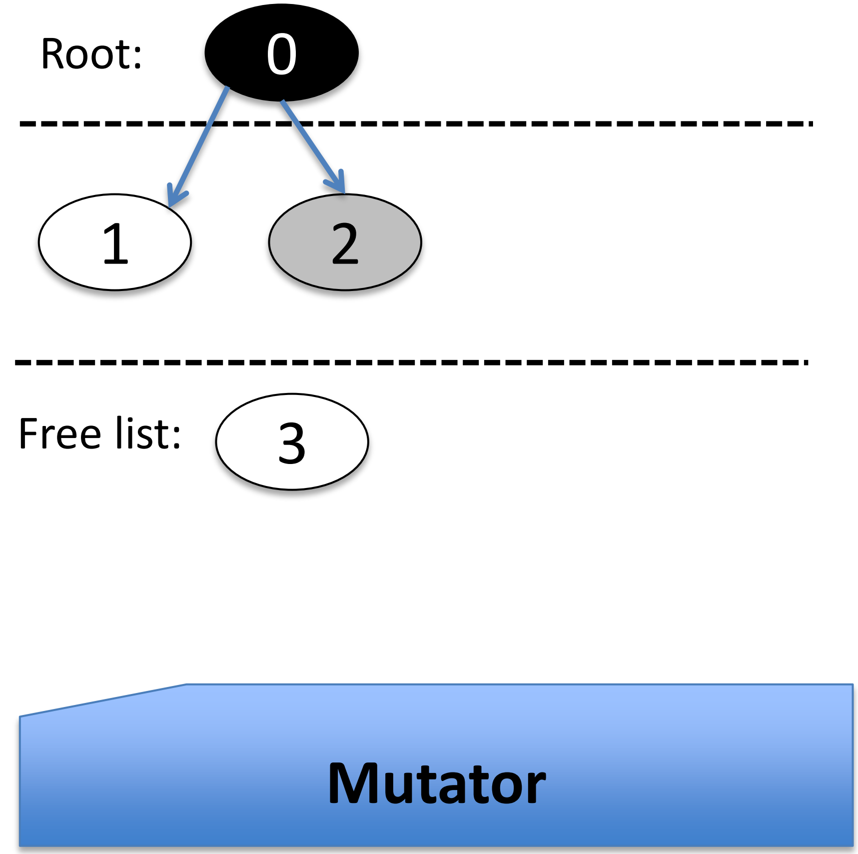
- An attempt to break M1 into 2 atomic operations:
  - **<Redirect an outgoing edge of a reachable node towards an already reachable one>**
  - <Shade the new target>
- Shading must be the first in order to keep P1!



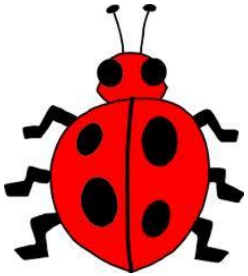
# The BUGGY Proposal - Demo



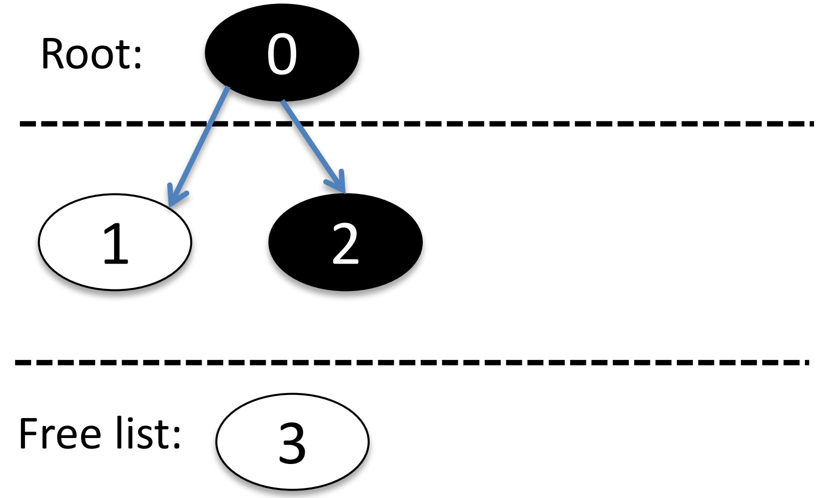
- An attempt to break M1 into 2 atomic operations:
  - **<Redirect an outgoing edge of a reachable node towards an already reachable one>**
  - **<Shade the new target>**
- Shading must be the first in order to keep P1!



# The BUGGY Proposal - Demo

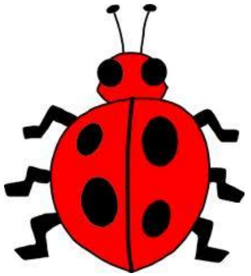


- An attempt to break M1 into 2 atomic operations:
  - <Redirect an outgoing edge of a reachable node towards an already reachable one>
  - <Shade the new target>
- Shading must be the first in order to keep P1!

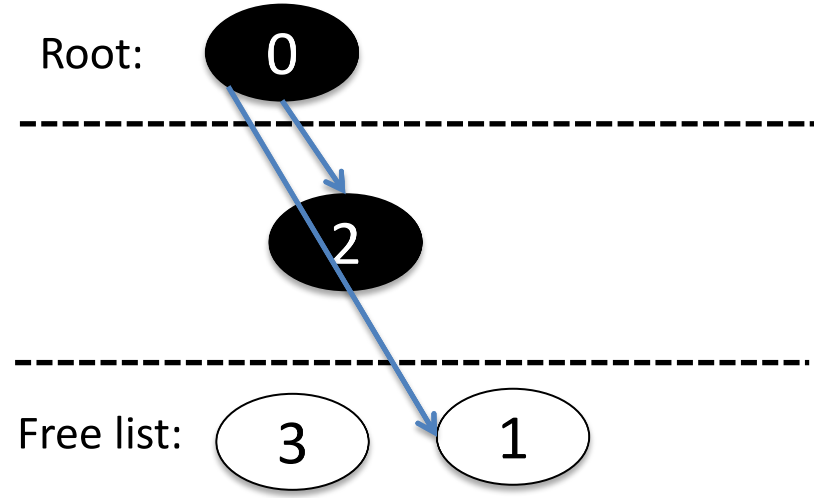



**Collector – Marking Phase**

# The BUGGY Proposal - Demo



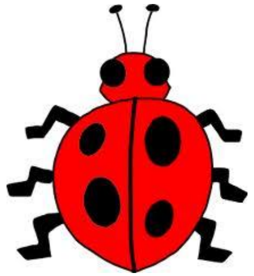
- An attempt to break M1 into 2 atomic operations:
  - <Redirect an outgoing edge of a reachable node towards an already reachable one>
  - <Shade the new target>
- Shading must be the first in order to keep P1!



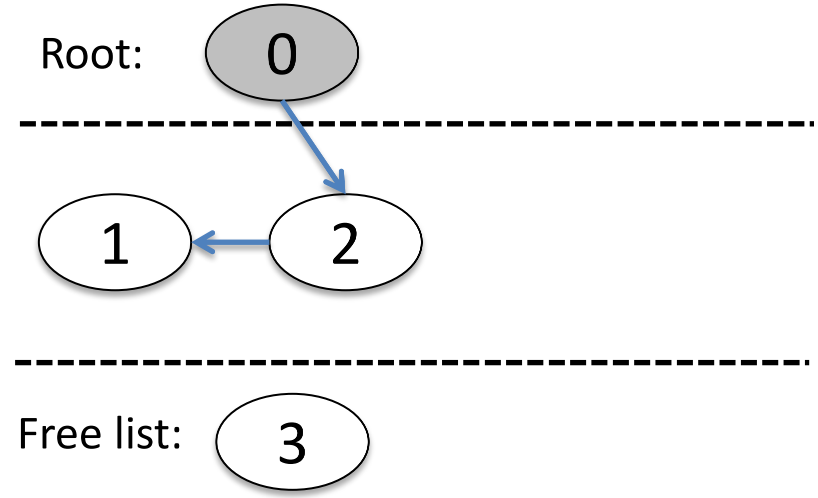

**A Reachable Node in The Free List!**

**Collector – Appending Phase**

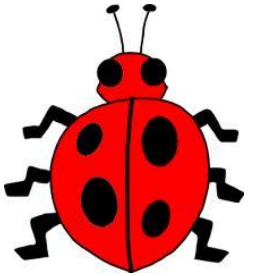
# The BUGGY Proposal - Reply



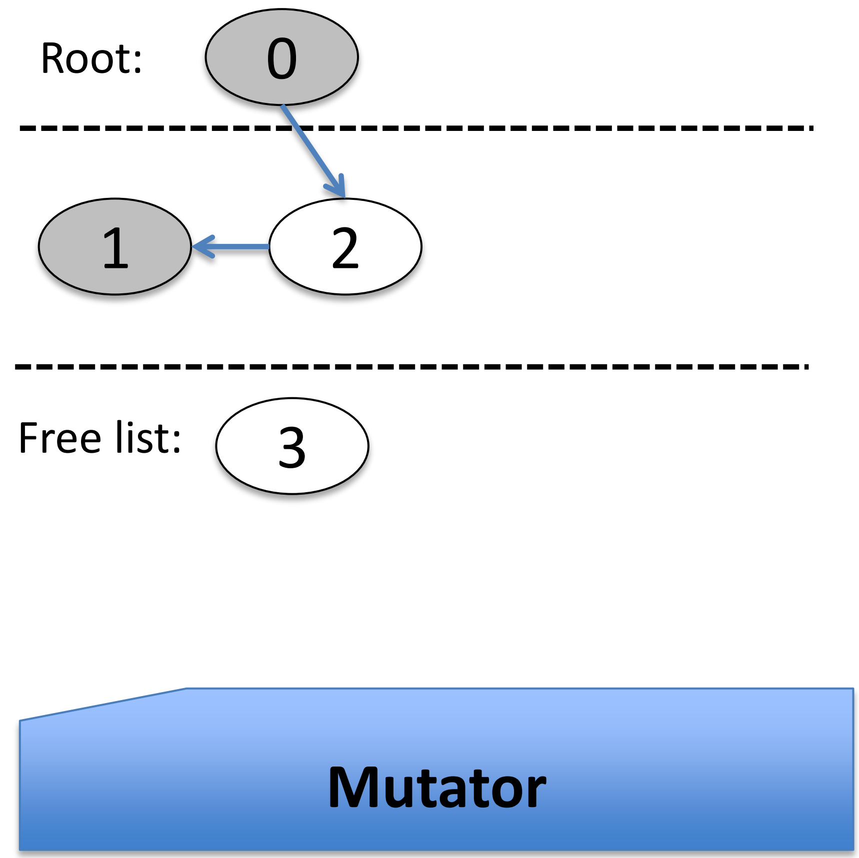
- An attempt to break M1 into 2 atomic operations:
  - <Redirect an outgoing edge of a reachable node towards an already reachable one>
  - <Shade the new target>
- Shading must be the first in order to keep P1!



# The BUGGY Proposal - Reply

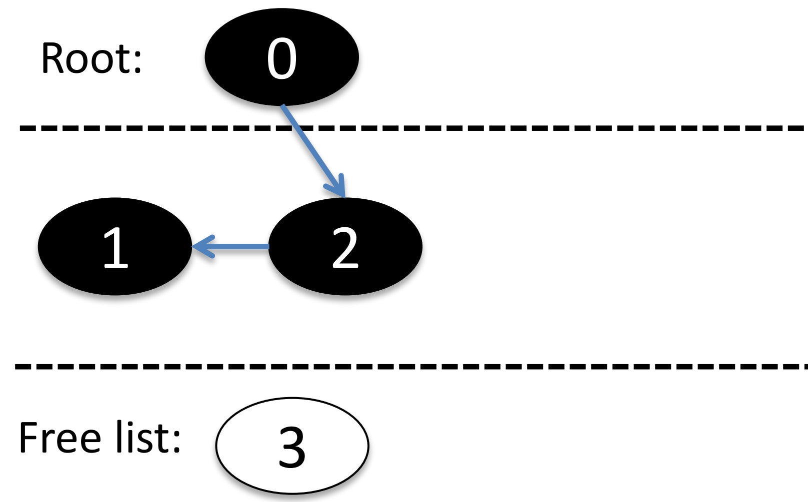


- An attempt to break M1 into 2 atomic operations:
  - <Redirect an outgoing edge of a reachable node towards an already reachable one>
  - **<Shade the new target>**
- Shading must be the first in order to keep P1!



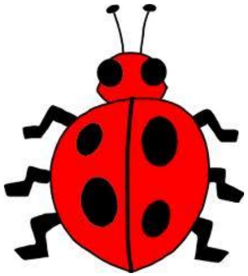
# The BUGGY Proposal - Reply

- An attempt to break M1 into 2 atomic operations:
  – <Redirect an outgoing edge of a reachable node towards an already reachable one>
  – <Shade the new target>
- Shading must be the first in order to keep P1!

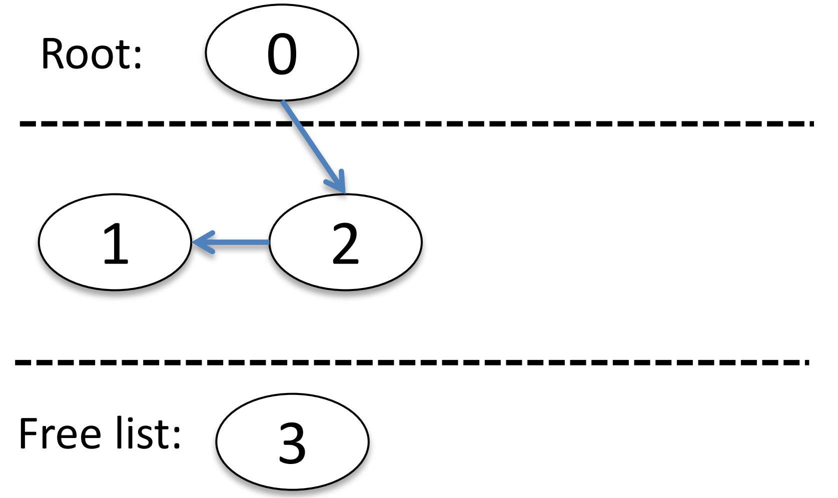



**Collector – Marking Phase**

# The BUGGY Proposal - Reply

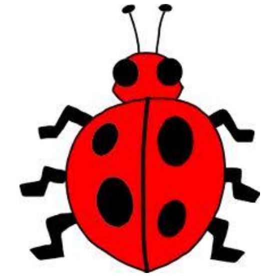


- An attempt to break M1 into 2 atomic operations:
  – <Redirect an outgoing edge of a reachable node towards an already reachable one>
  – <Shade the new target>
- Shading must be the first in order to keep P1!

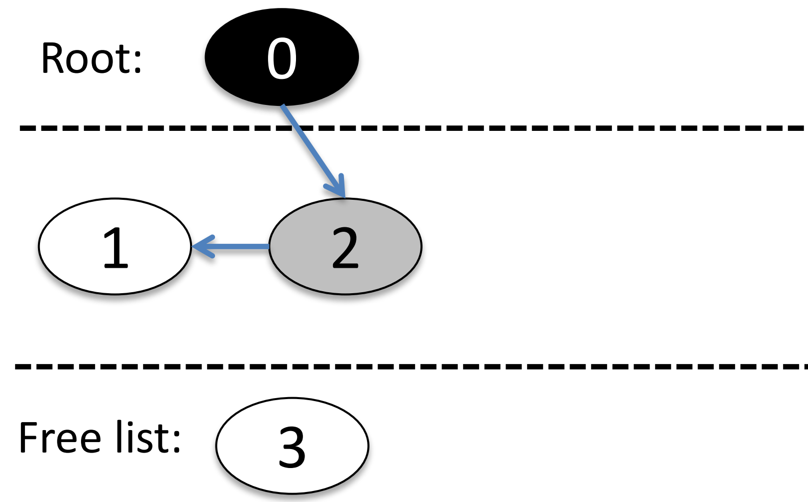



**Collector – Appending Phase**

# The BUGGY Proposal - Reply

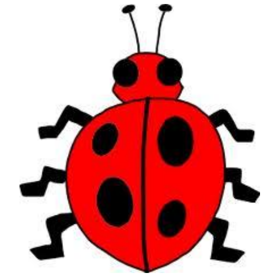


- An attempt to break M1 into 2 atomic operations:
  - <Redirect an outgoing edge of a reachable node towards an already reachable one>
  - <Shade the new target>
- Shading must be the first in order to keep P1!

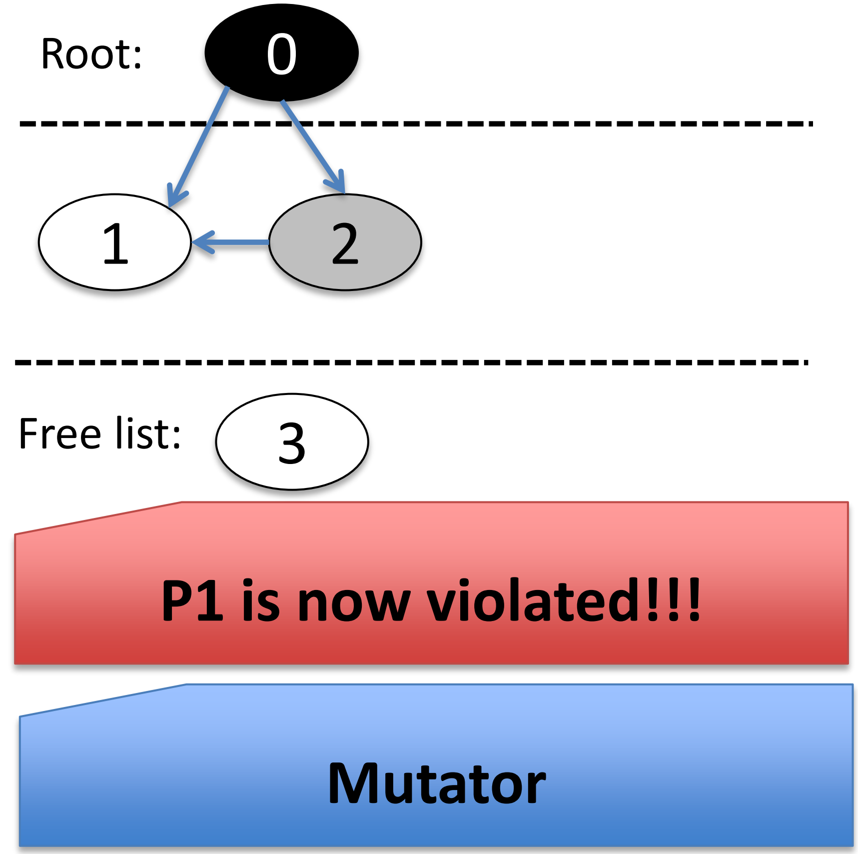



**Collector – Marking Phase**

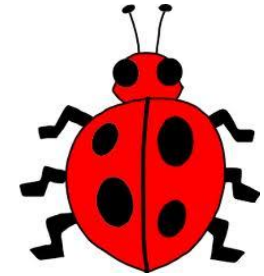
# The BUGGY Proposal - Reply



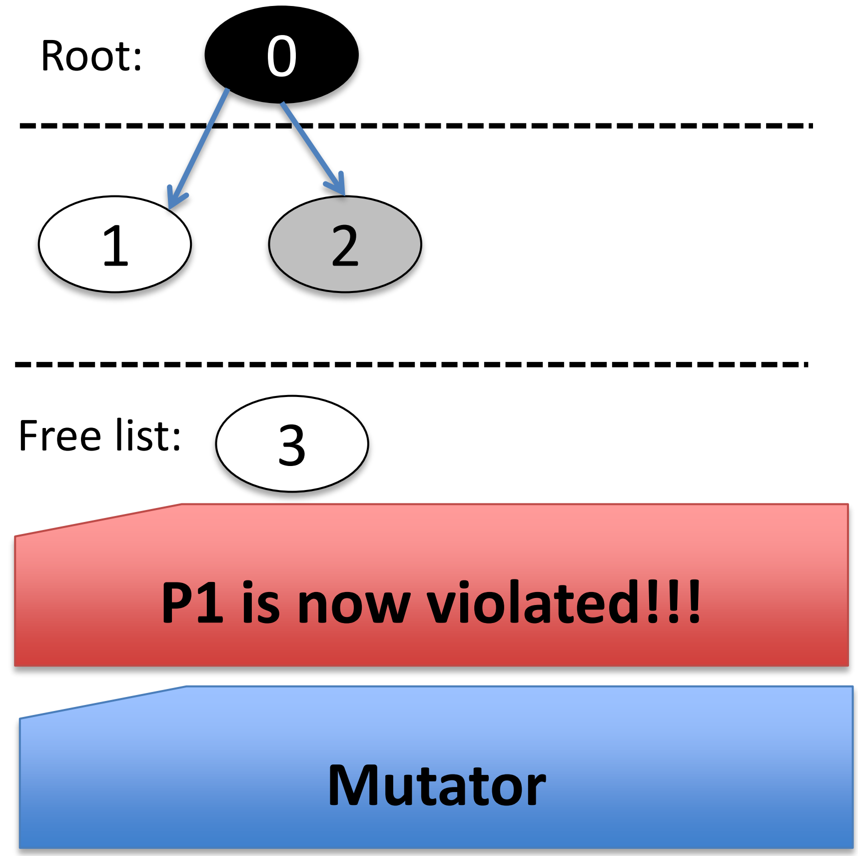
- An attempt to break M1 into 2 atomic operations:
  - **<Redirect an outgoing edge of a reachable node towards an already reachable one>**
  - <Shade the new target>
- Shading must be the first in order to keep P1!



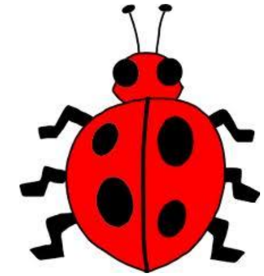
# The BUGGY Proposal - Reply



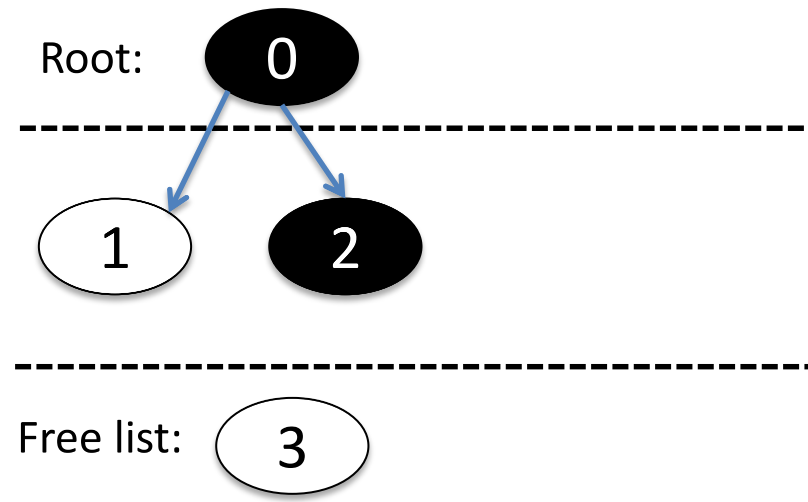
- An attempt to break M1 into 2 atomic operations:
  - <Redirect an outgoing edge of a reachable node towards an already reachable one>
  - <Shade the new target>
- Shading must be the first in order to keep P1!



# The BUGGY Proposal - Reply



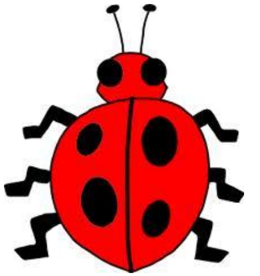
- An attempt to break M1 into 2 atomic operations:
  - <Redirect an outgoing edge of a reachable node towards an already reachable one>
  - <Shade the new target>
- Shading must be the first in order to keep P1!



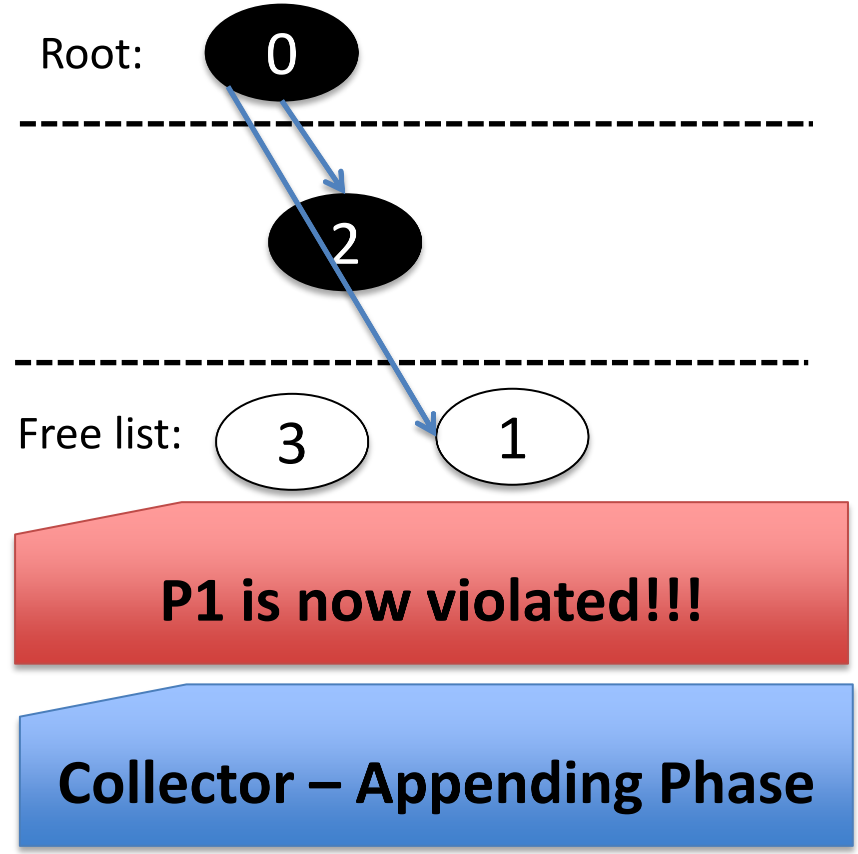

**P1 is now violated!!!**

**Collector – Marking Phase**

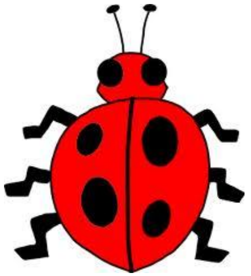
# The BUGGY Proposal - Reply



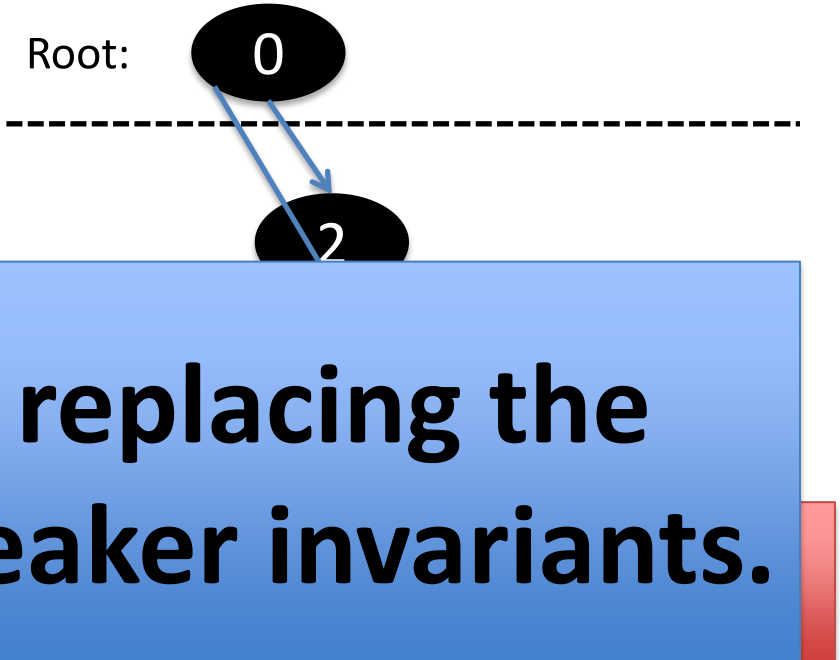
- An attempt to break M1 into 2 atomic operations:
  - <Redirect an outgoing edge of a reachable node towards an already reachable one>
  - <Shade the new target>
- Shading must be the first in order to keep P1!



# The BUGGY Proposal - Reply



- An attempt to break M1 into 2 atomic operations:




**The new idea – replacing the invariant P1 by weaker invariants.**

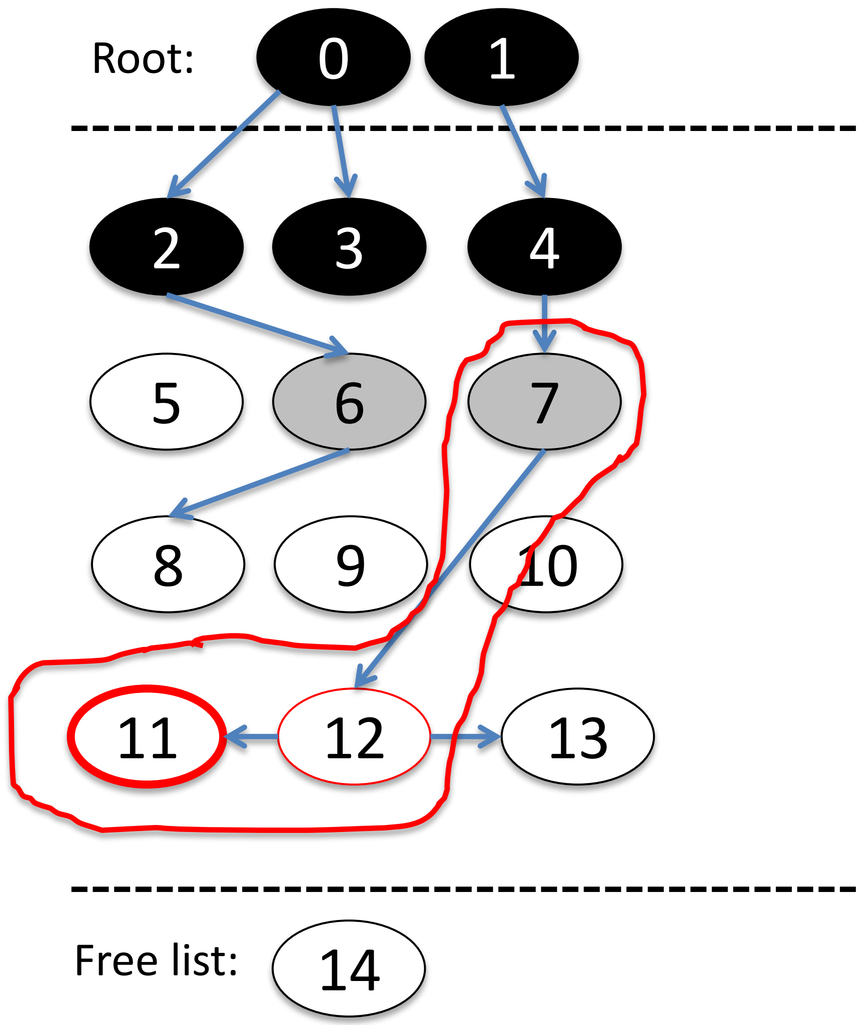
  - <Shade the new target>

- Shading must be the first in order to keep P1!

**Collector – Appending Phase**

# New Invariant: P2

- Propagation path: A path of consisting solely of edges with white targets, and starting from a gray node.
- P2: “For each white reachable node, there exists a propagation path leading to it”

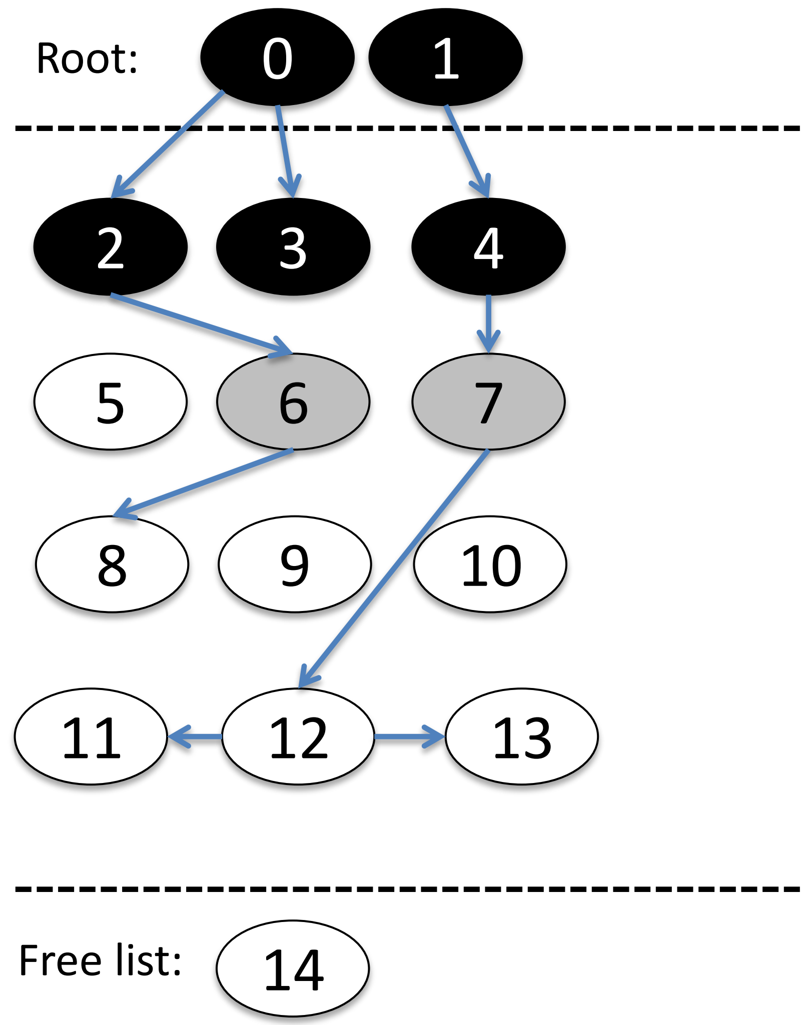


# New Invariant: P3

- P3: “Only the last edge placed by the mutator may lead from a black node to a white one”

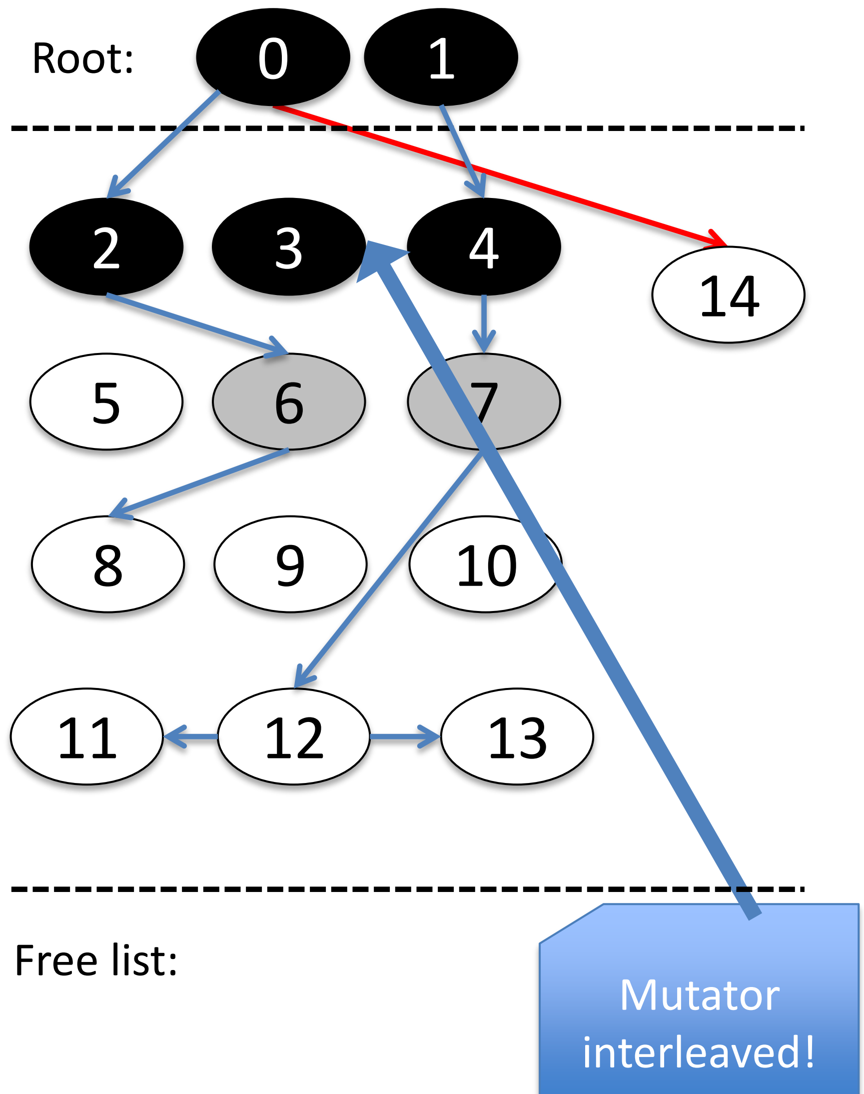
# The New Algorithm

- The collector remains the same!
- The mutator's new operation is the following M2:
  - <Shade the target of the previously redirected edge, and redirect an outgoing edge of a reachable node towards a reachable node>



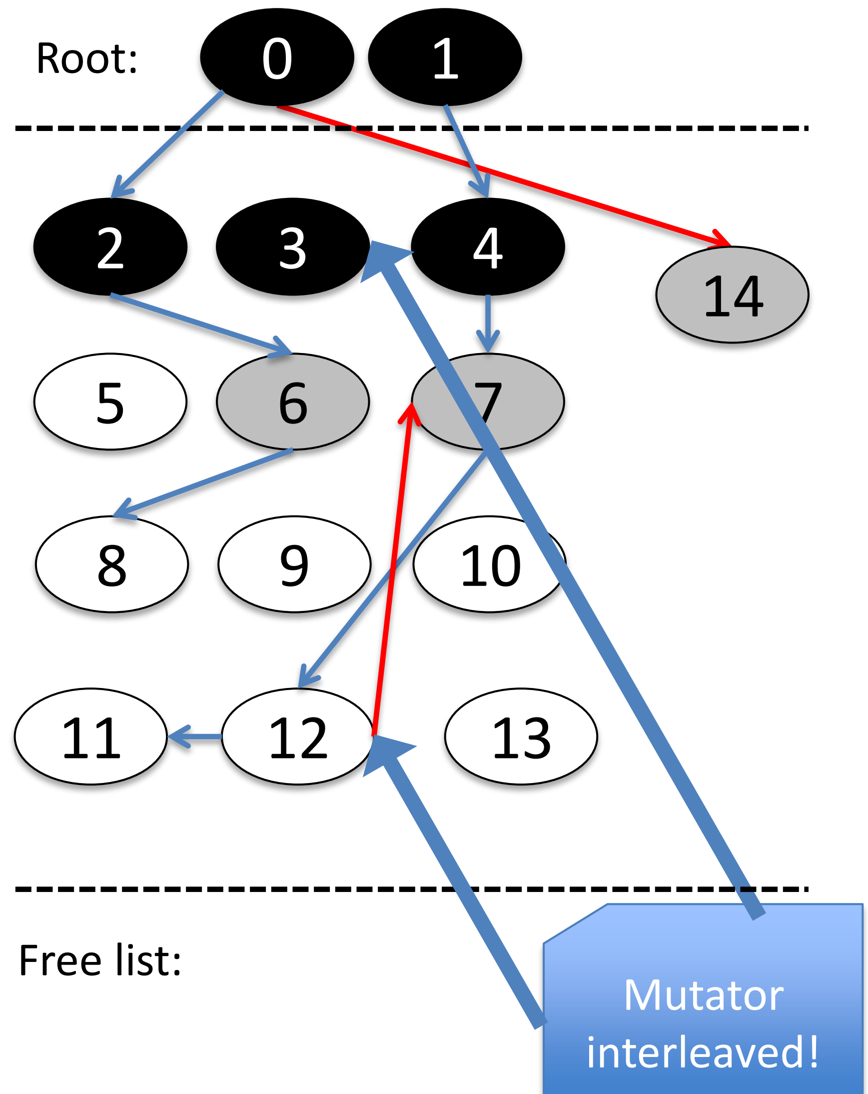
# The New Algorithm

- The collector remains the same!
- The mutator's new operation is the following M2:
  - <Shade the target of the previously redirected edge, and redirect an outgoing edge of a reachable node towards a reachable node>



# The New Algorithm

- The collector remains the same!
- The mutator's new operation is the following M2:
  - <Shade the target of the previously redirected edge, and redirect an outgoing edge of a reachable node towards a reachable node>



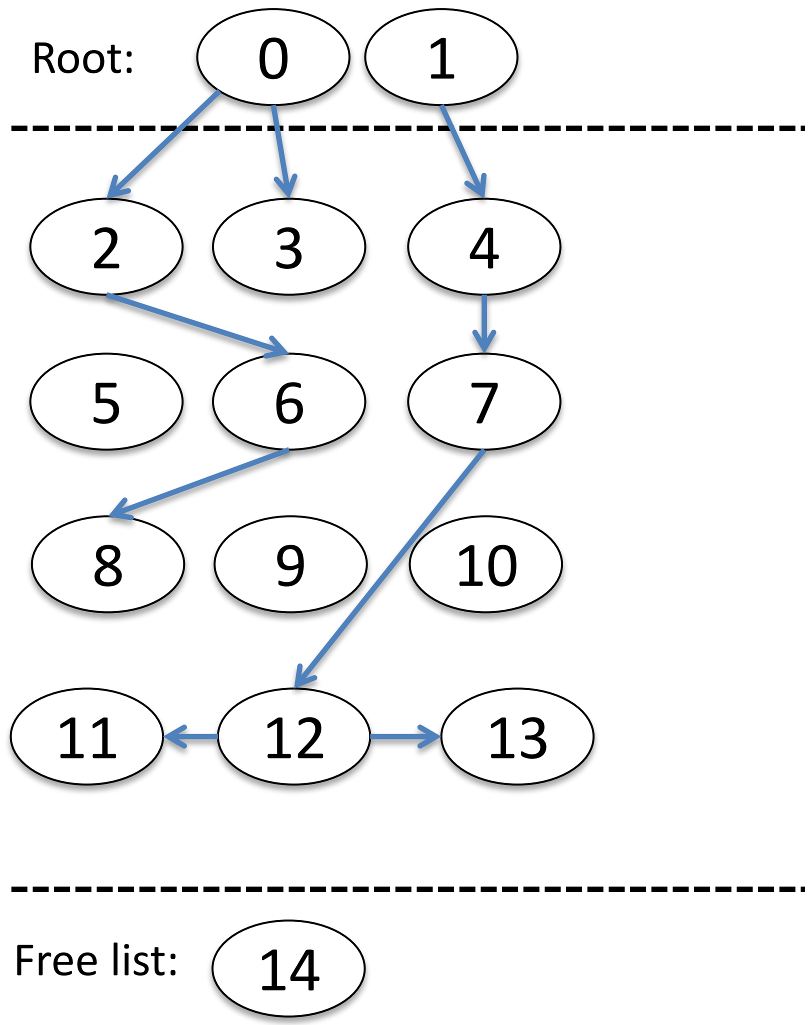
# Correction proof

- P2 & P3 are invariants for this algorithm.
- By using these invariants we can proof the correctness of the second algorithm in the same manner of the first one.

# THE FINE-GRAINED SOLUTION

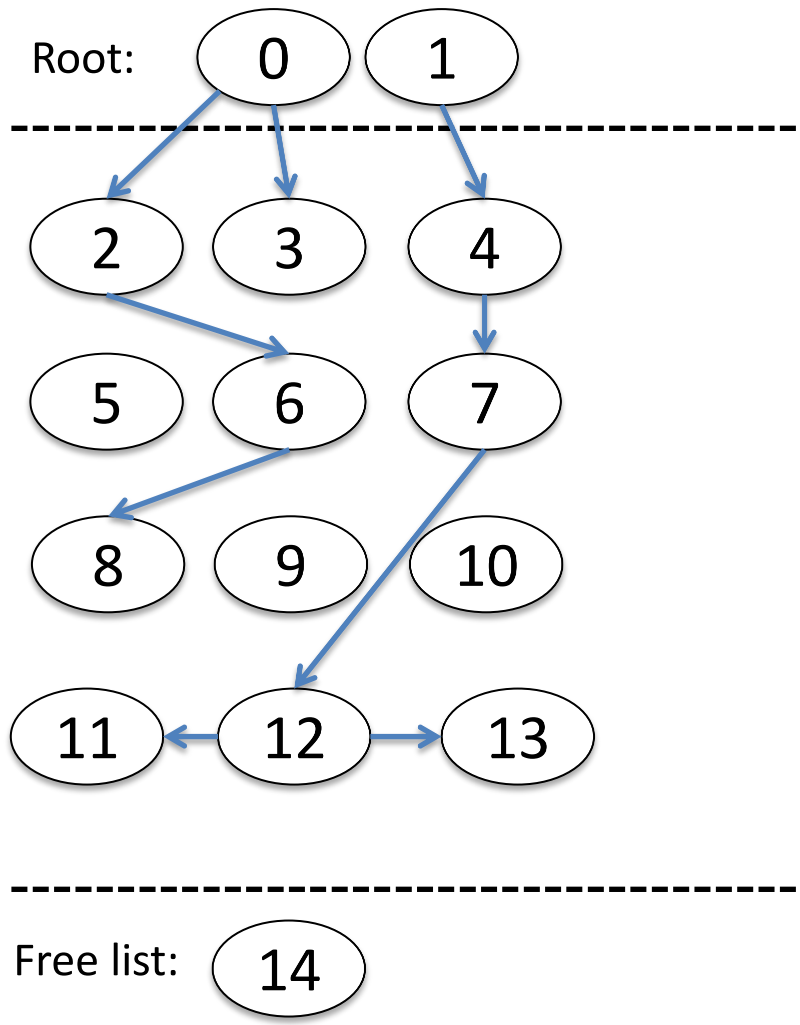
# The New Mutator

- M2.1:
  - <Shade the target of the previously redirected edge>
- M2.2:
  - <Redirect an outgoing edge of a reachable node towards a reachable node>



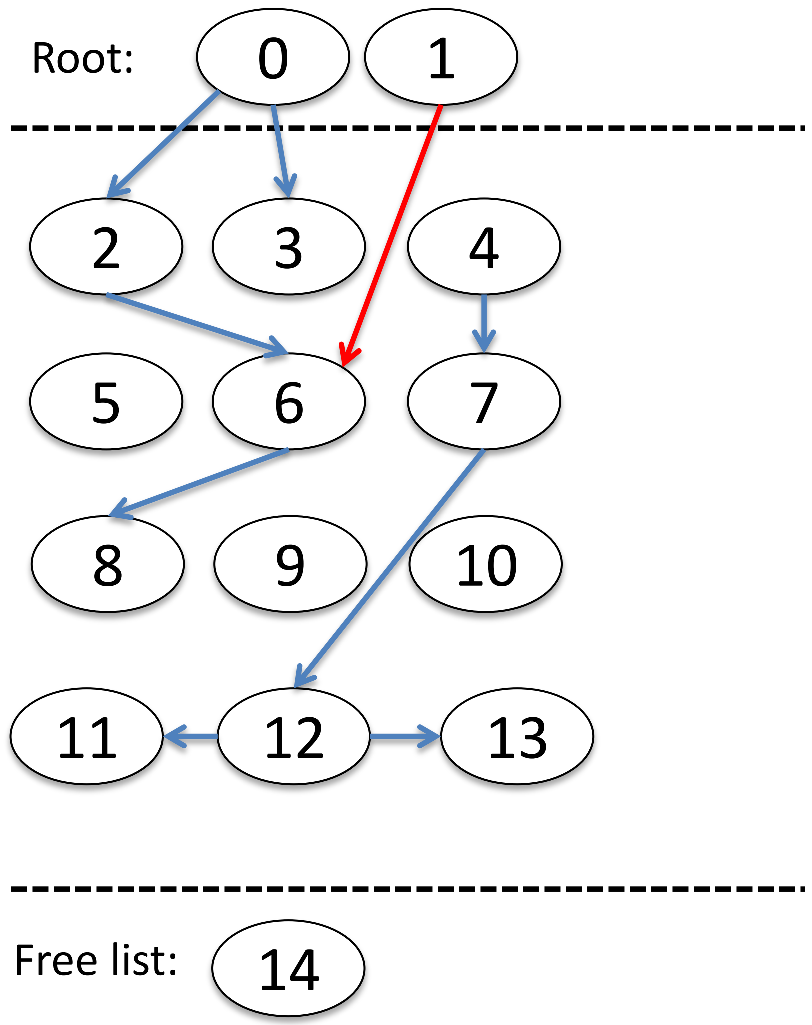
# The New Mutator

- M2.1:
  - **<Shade the target of the previously redirected edge>**
- M2.2:
  - <Redirect an outgoing edge of a reachable node towards a reachable node>



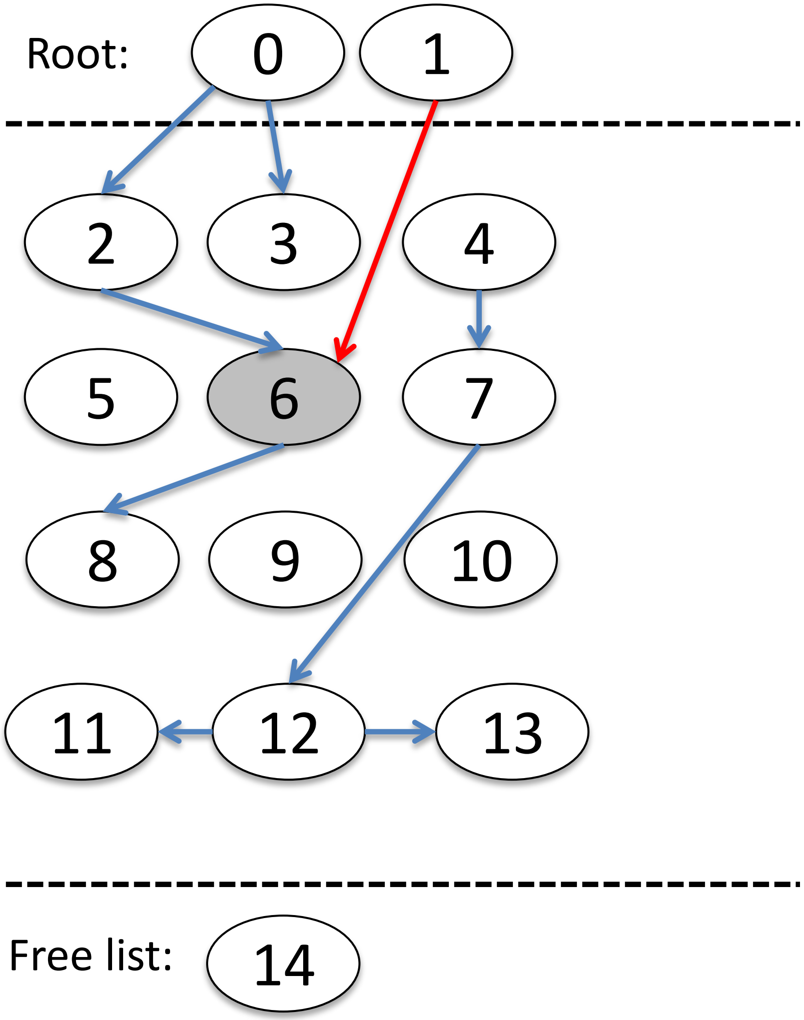
# The New Mutator

- M2.1:
  - <Shade the target of the previously redirected edge>
- M2.2:
  - **<Redirect an outgoing edge of a reachable node towards a reachable node>**



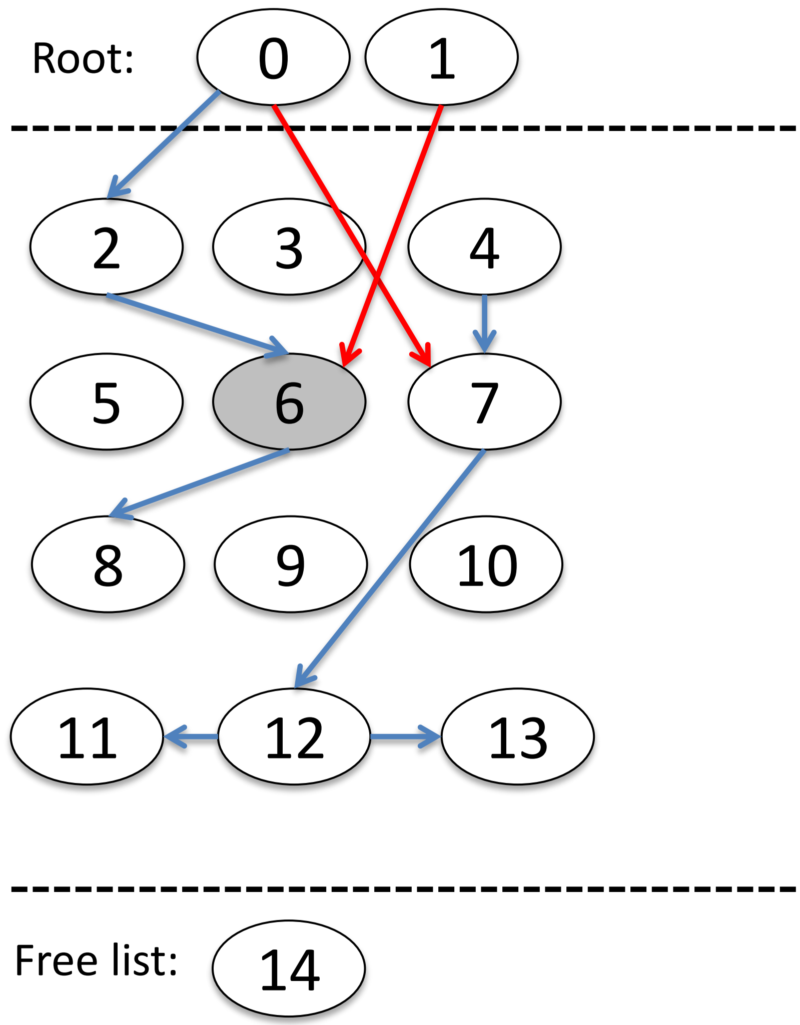
# The New Mutator

- M2.1:
  - **<Shade the target of the previously redirected edge>**
- M2.2:
  - <Redirect an outgoing edge of a reachable node towards a reachable node>



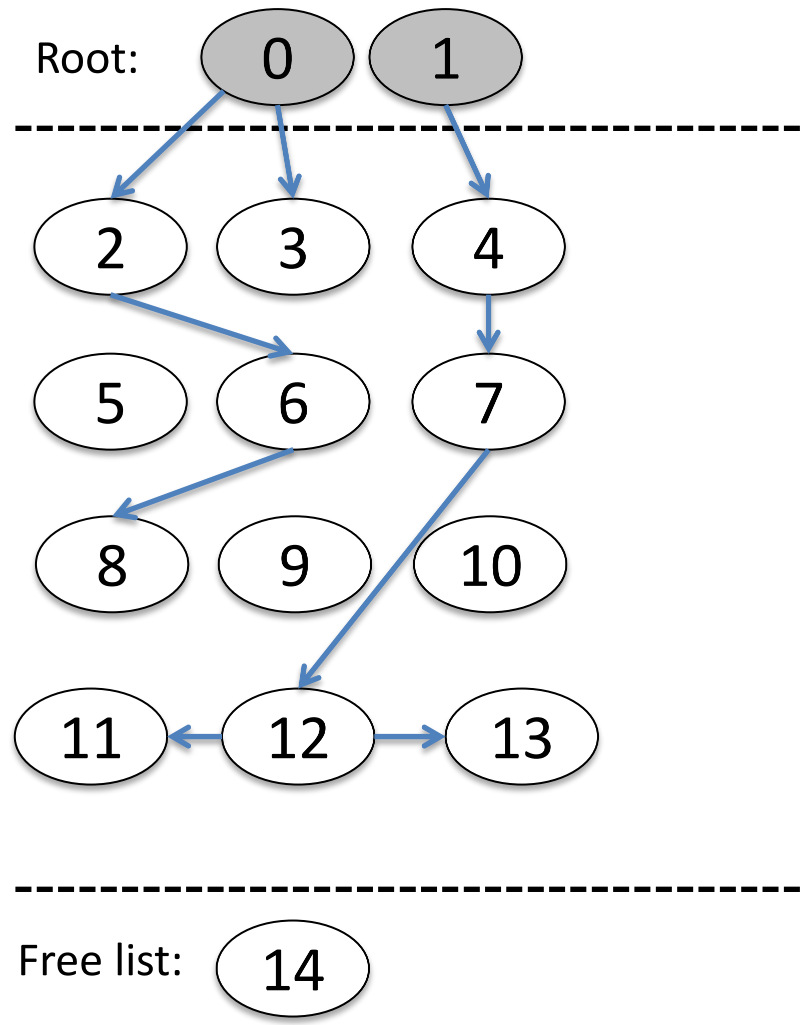
# The New Mutator

- M2.1:
  - <Shade the target of the previously redirected edge>
- M2.2:
  - **<Redirect an outgoing edge of a reachable node towards a reachable node>**



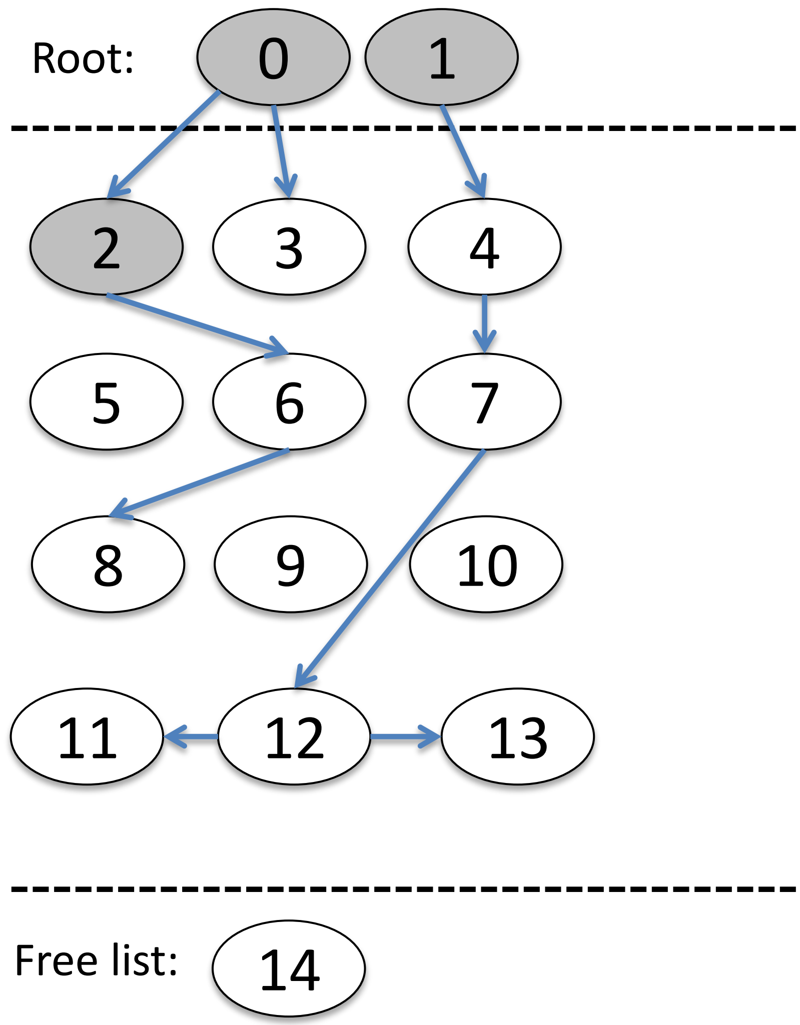
# The New Collector

- Basically the same, but with finer operations.
- C1.1a:
  - C1.1: <m1 := number of the left-hand successor of node #i>
  - C1.2: <shade node #M1>
- C1.3a:
  - C1.3: <m2:= number of the right-hand successor of node #i>
  - C1.4: <shade node #M2>
- CI.5: <make node #i black>



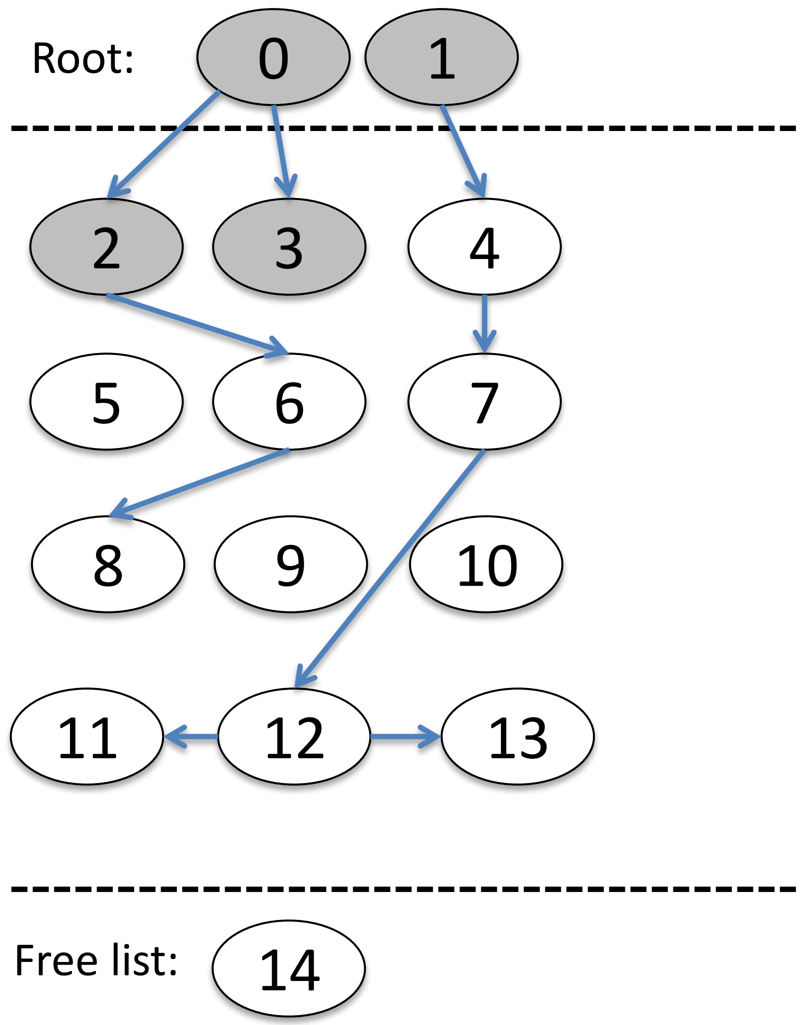
# The New Collector

- Basically the same, but with finer operations.
- **C1.1a:**
  - **C1.1: <m1 := number of the left-hand successor of node #i>**
  - **C1.2: <shade node #M1>**
- C1.3a:
  - C1.3: <m2:= number of the right-hand successor of node #i>
  - C1.4: <shade node #M2>
- Cl.5: <make node #i black>



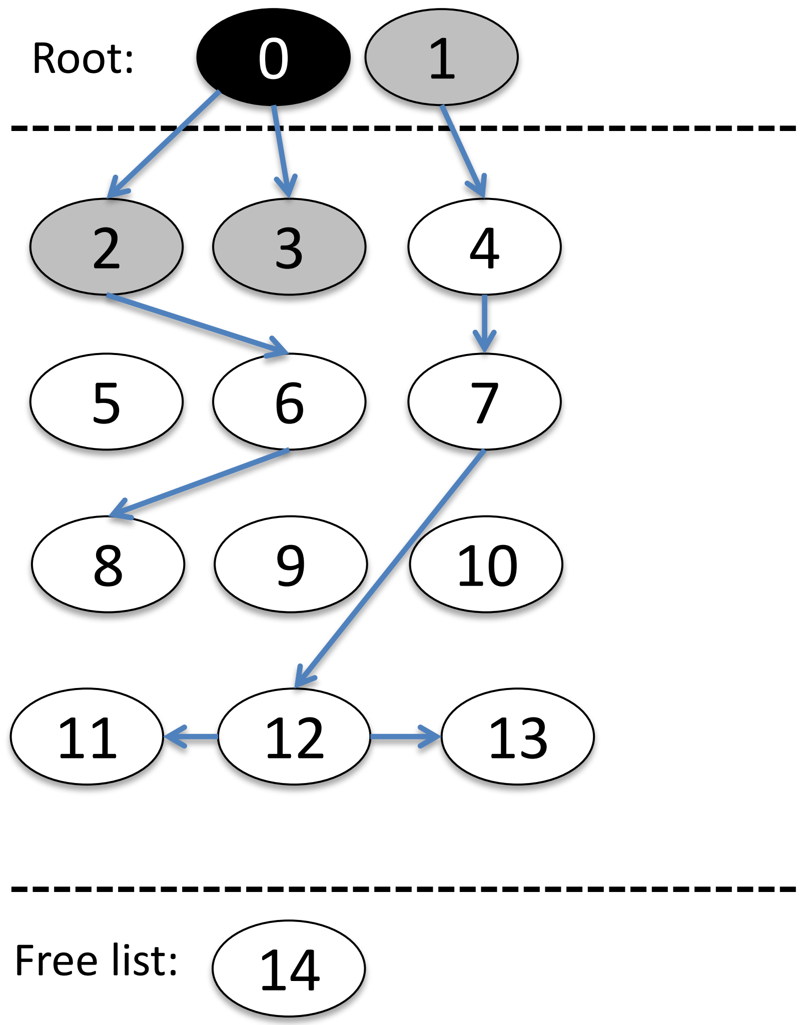
# The New Collector

- Basically the same, but with finer operations.
- C1.1a:
  - C1.1: <m1 := number of the left-hand successor of node #i>
  - C1.2: <shade node #M1>
- **C1.3a:**
  - **C1.3: <m2:= number of the right-hand successor of node #i>**
  - **C1.4: <shade node #M2>**
- CI.5: <make node #i black>



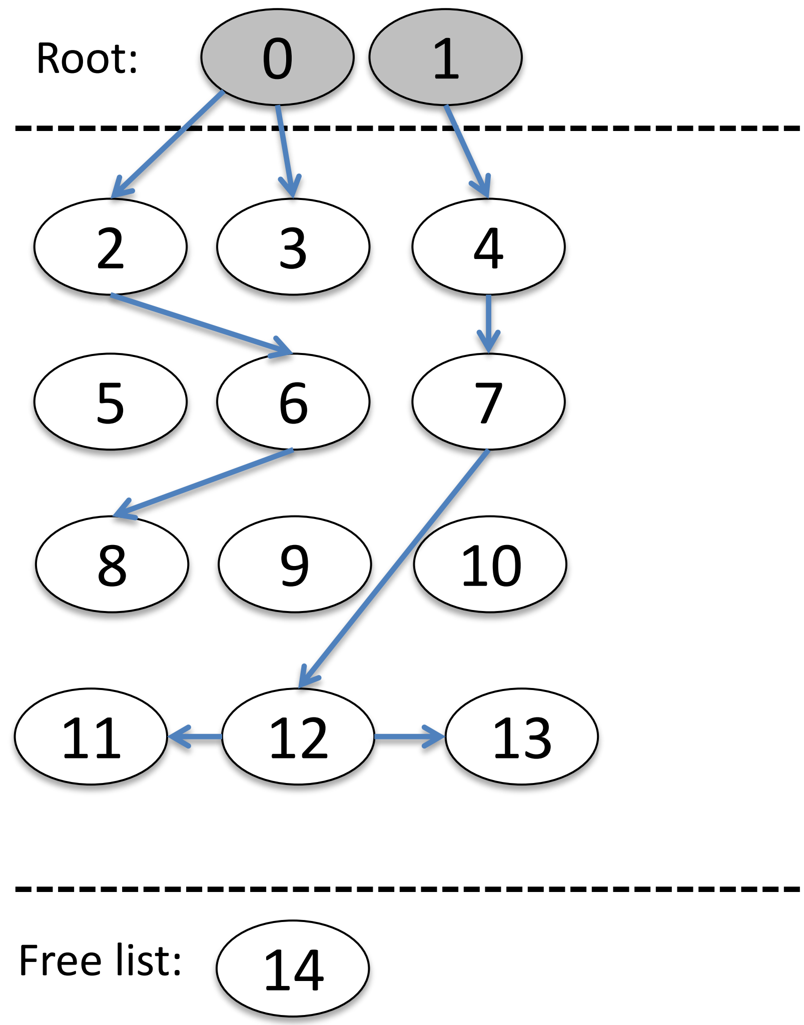
# The New Collector

- Basically the same, but with finer operations.
- C1.1a:
  - C1.1: <m1 := number of the left-hand successor of node #i>
  - C1.2: <shade node #M1>
- C1.3a:
  - C1.3: <m2:= number of the right-hand successor of node #i>
  - C1.4: <shade node #M2>
- **CI.5: <make node #i black>**



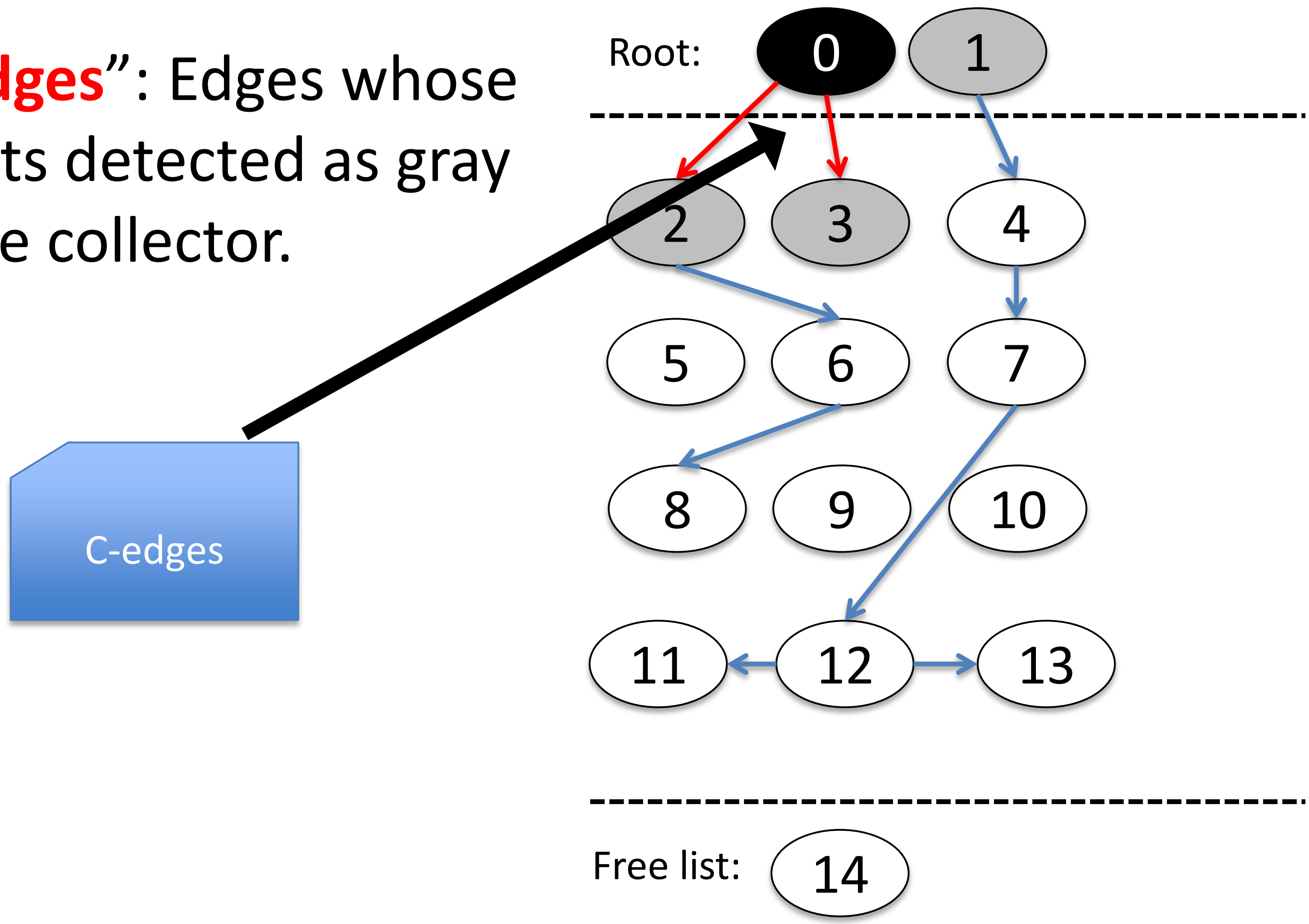
# C-edges

- "C-edges": Edges whose targets detected as gray by the collector.



# C-edges

- "**C-edges**": Edges whose targets detected as gray by the collector.



# The New Invariants

- P2a: “Every root is gray or black, and for each white reachable node there exists a propagation path leading to it, containing no C-edges.
- P3a: “There exists at most one edge E satisfying that ‘(E is a black-to-white edge) or (E is C-edge with a white target)’.
  - The existence of E implies that the mutator is between action M2.2 and the subsequent M2.1, and that E is identical with the edge most recently redirected by the mutator.

# Correction Proof

- P2a & P3a are invariants for this algorithm.
- By using these invariants we can proof the correctness of the fine-grained algorithm in the same manner of the coarse-grained ones.

# Related work

- This is the first paper for concurrent GC.
- *"Real-Time Garbage Collection on General-Purpose Machines",* Yuasa, 1990
  - *Designed for single core systems.*
- *"Multiprocessing compactifying garbage collection"*, Steele, 1975
  - Contained a bug.
  - Fixed in 1976.

# Gries's proof

$\{Mfree \wedge Mgraph\}$
*mutator* : **do true** $\Rightarrow$
  $\{Mfree \wedge Mgraph\}$
  Let $k, j$ be indices of nodes reachable from $ROOT$;
  $\{Mfree \wedge Mgraph \wedge reachR(k) \wedge reachR(j)\}$
  **if true** $\Rightarrow$ $\{Mfree \wedge Mgraph \wedge reachR(k)\}$
    $m[k].left := 0$
    $\{Mfree \wedge Mgraph \wedge reachR(k) \wedge \mathscr{L}k = 0\}$
  ▯ **true** $\Rightarrow$ $\{Mfree \wedge Mgraph \wedge reachR(k) \wedge reachR(j)\}$
    $addleft(k, j)$
    $\{Mfree \wedge Mgraph \wedge reachR(k) \wedge \mathscr{L}k = j\}$
  ▯ **true** $\Rightarrow$ *Take first free node as k's left successor*:
      $\{Mfree \wedge Mgraph \wedge reachR(k)\}$
      $f := M[FREE].left$;
      $\{Mfree \wedge Mgraph \wedge reachR(k) \wedge \mathscr{L}FREE = f \neq 0\}$
      $addleft(k, f)$;
      $\left\{\begin{array}{l} Ifree \wedge Mgraph \wedge reachR(k) \wedge \\ \quad \mathscr{L}Free = \mathscr{L}k = f \neq 0 \wedge \\ \text{every path from } ROOT \text{ to free list uses} \\ \quad \text{edge } (k, \mathscr{L}k) \end{array}\right\}$
      **do** $f = ENDFREE \Rightarrow$ **skip od**;
      $\left\{\begin{array}{l} Ifree \wedge Mgraph \wedge reachR(k) \wedge \\ \quad \mathscr{L}FREE = \mathscr{L}k = f \neq 0 \wedge \\ \text{every path from } ROOT \text{ to free list uses} \\ \quad \text{edge } (k, \mathscr{L}k) \end{array}\right\}$
      $addleft(FREE, m[f].left)$;
      $\left\{\begin{array}{l} Ifree \wedge Mgraph \wedge reachR(k) \wedge \\ \quad \mathscr{L}FREE = \mathscr{L}f \wedge \mathscr{L}k = f \wedge \mathscr{R}f = 0 \\ \text{every path from } ROOT \text{ to free list uses} \\ \quad \text{edge } (f, \mathscr{L}f) \end{array}\right\}$
      $m[f].left := 0$
      **fi**
      $\{Mfree \wedge Mgraph\}$
    **od**

*Collect*:
$\{Cfree \wedge \neg\, mark \wedge \text{all white nodes are unreachable}\}$
$\{Cfree \wedge Ccoll(-1)\}$
**for** $i := 0$ **step** 1 **until** $N$ **do**
  $\{Cfree \wedge Ccoll(i-1)\}$
  **if** $m[i].color = white \Rightarrow$ (4.3.1)
    $\{Cfree \wedge Ccoll(i) \wedge \neg\, reach(i)\}$
    $m[i].left := 0;\ m[i].right := 0;$
    $\{Cfree \wedge Ccoll(i) \wedge \neg\, reach(i) \wedge \mathscr{L}i = \mathscr{R}i = 0\}$
    $m[ENDFREE].left := i;$
    $\{Ifree \wedge Ccoll(i) \wedge \mathscr{L}ENDFREE = i \neq 0 \wedge \mathscr{L}0 = \mathscr{R}0 = 0\}$
    $ENDFREE := i$
    $\{Cfree \wedge Ccoll(i) \wedge ENDFREE = i\}$
  ▯ $m[i].color = black \Rightarrow \{Cfree \wedge Ccoll(i-1) \wedge i \text{ black}\}$
    $whiten(i)$
    $\{Cfree \wedge Ccoll(i)\}$
  ▯ $m[i].color = gray \Rightarrow$ **skip** $\{Cfree \wedge Ccoll(i)\}$
  **fi**
  $\{Cfree \wedge Ccoll(i)\}$;
$\{Cfree \wedge Ccoll(N)\}$
$\{Cfree \wedge \neg\, mark \wedge \text{no black nodes}\}$

# Conclusions

- Started by defining the problem
- Presented a fine-grained solution by 3 milestones:
  - The first coarse-grained solution
  - The second coarse-grained solution
  - The fine-grained solution

# My Own Conclusion

- Very interesting idea.
- Applying these techniques on modern OS with multiple processes may raise some challenges
  - A Collector thread per process may lead to a serious performance impact.
  - Sharing the same collector thread between processes may lead to serous security issues to deal with.

# Questions?

?