Garbage Collection Techniques

Mainack Mondal
Sandip Chakraborty

CS 60203
Autumn 2024

Outline

- Garbage Collection Techniques:
  - Reference Counting
  - Tracing
  - Modern Techniques
- Case Study: GC in Java
- Drawbacks of GC

Garbage Collection Techniques

Garbage Collection Techniques: Overview

- **Reference counting:**
  - Directly keeps track of live cells
  - GC takes place whenever heap block is allocated
  - Doesn’t detect all garbage

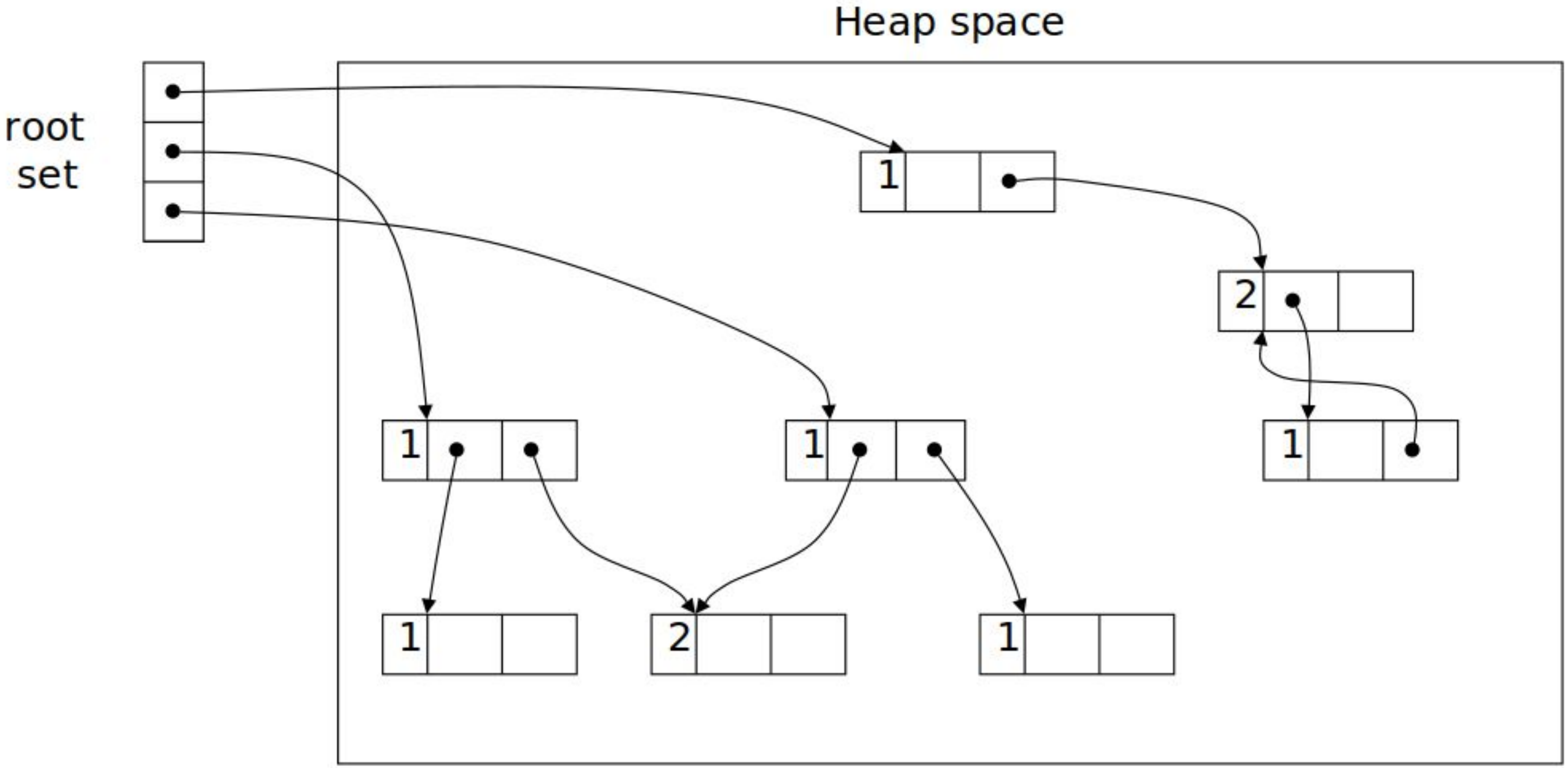
- **Tracing:**
  - GC takes place and identifies live cells when a request for memory fails
  - Mark-sweep

- **Modern techniques:** Generational GC

Reference Counting

- Counting the number of references to a live cell
- Requires space and time overhead to store the count and increment (decrement) each time a reference is added (removed)
  - Reference counts are maintained in real-time, so no *“stop-and-gag”* effect
  - Incremental garbage collection
- Unix file system uses a reference count for files
- Eg: Smart Pointers in C++: `std::shared_ptr`

Reference Counting - Example



Example - Kernel Reference Counting

- Adding reference counters to kernel objects
- Kernel ref counters: *krefs*

How do you use krefs ?

- Add a *kref* object to your data structure

```
struct my_data
{
     .
     .
     struct kref refcount;
     .
     .
};
```

Initialization

```
struct my_data *data;

data = kmalloc(sizeof(*data), GFP_KERNEL);
if (!data)
       return -ENOMEM;
kref_init(&data->refcount);
```

kref rules:

- When making a non-temporary copy of a pointer, especially if it's passed to another thread, increment the refcount with *kref_get()*:

  ```
  kref_get(&data->refcount);
  ```

  Note: If you already have a valid pointer to a kref-ed structure (the refcount cannot go to zero) you may do this without a lock.

- When you are done with a pointer, call *kref_put()*:

```
kref_put(&data->refcount, data_release);
```

Note: If this is the last reference to the pointer, the release routine will be called. If the code never tries to get a valid pointer to a kref-ed structure without already holding a valid pointer, it is safe to do this without a lock.

- To safely gain a reference to a kref-managed structure without a valid pointer, serialize access to ensure *kref_put()* isn't called during *kref_get()*, keeping the structure valid.

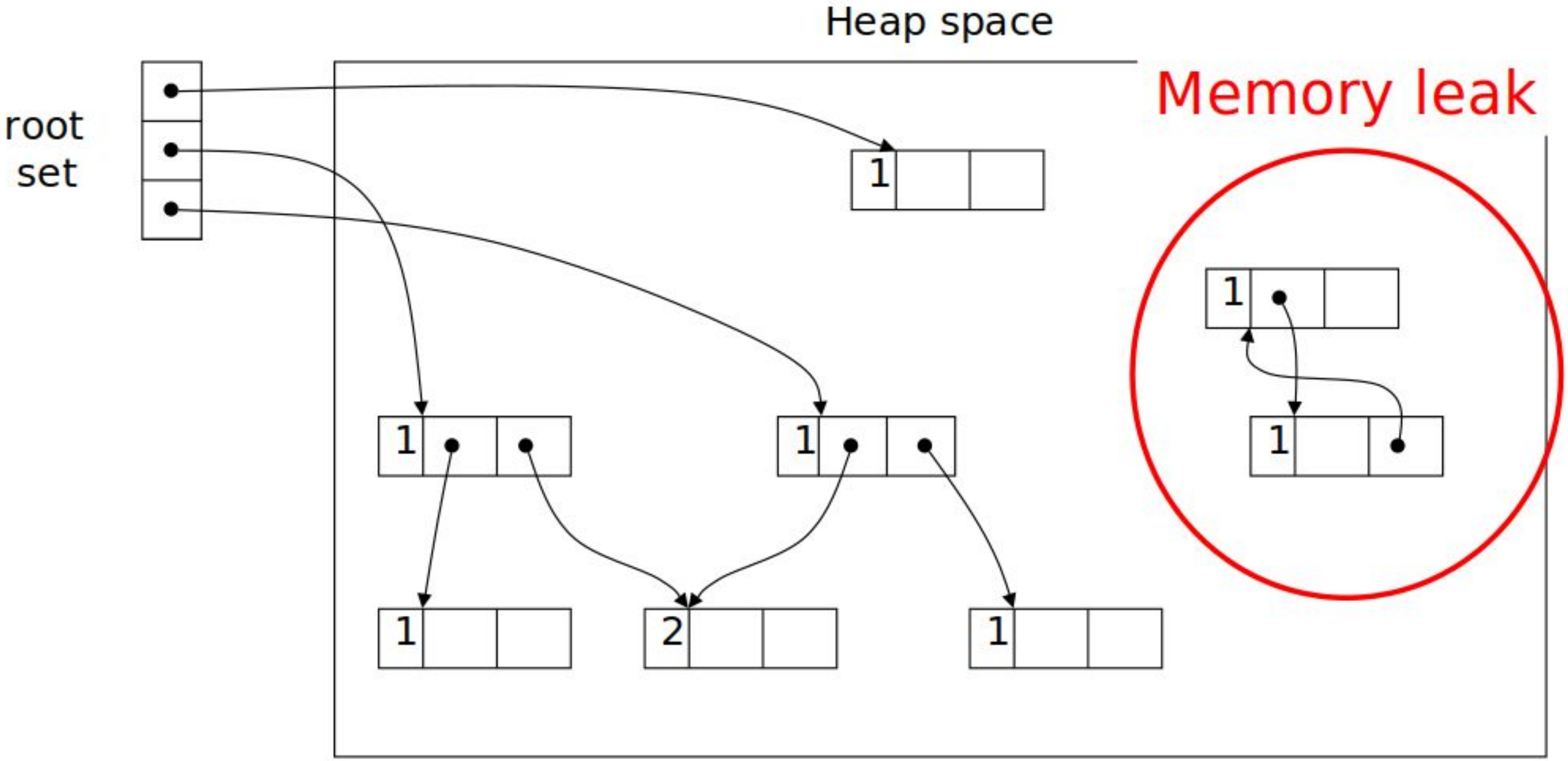
Reference Counting - Pros

- Incremental overhead
  - Cell management interleaved with program execution
  - Good for interactive or real-time computation
- Relatively easy to implement
- Can coexist with manual memory management
- Spatial locality of reference is good
  - Access pattern to virtual memory pages no worse than the program, so no excessive paging
- Can re-use freed cells immediately
  - If RC == 0, put back onto the free list

Reference Counting - Cons

- Space overhead
  - 1 word for the count
- Time overhead
  - Updating a pointer to point to a new cell requires:
  - Check to ensure that it is not a self-reference
  - Decrement the count on the old cell, possibly deleting it
  - Update the pointer with the address of the new cell
  - Increment the count on the new cell
- Ref. counting and pointer load-store operations should be atomic
- Cannot reclaim cyclic data structures

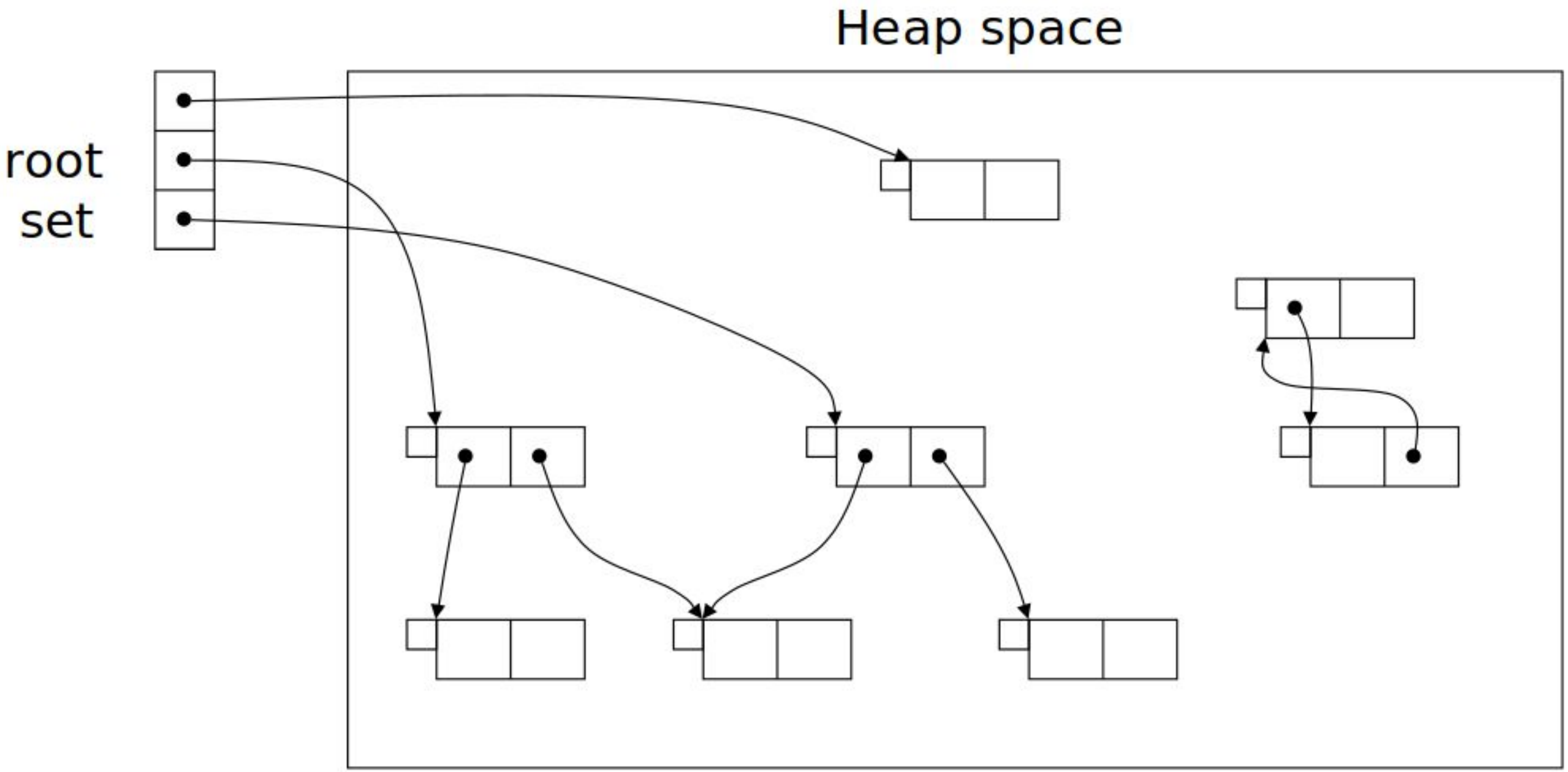
Reference Counting - Cyclic Data Structures



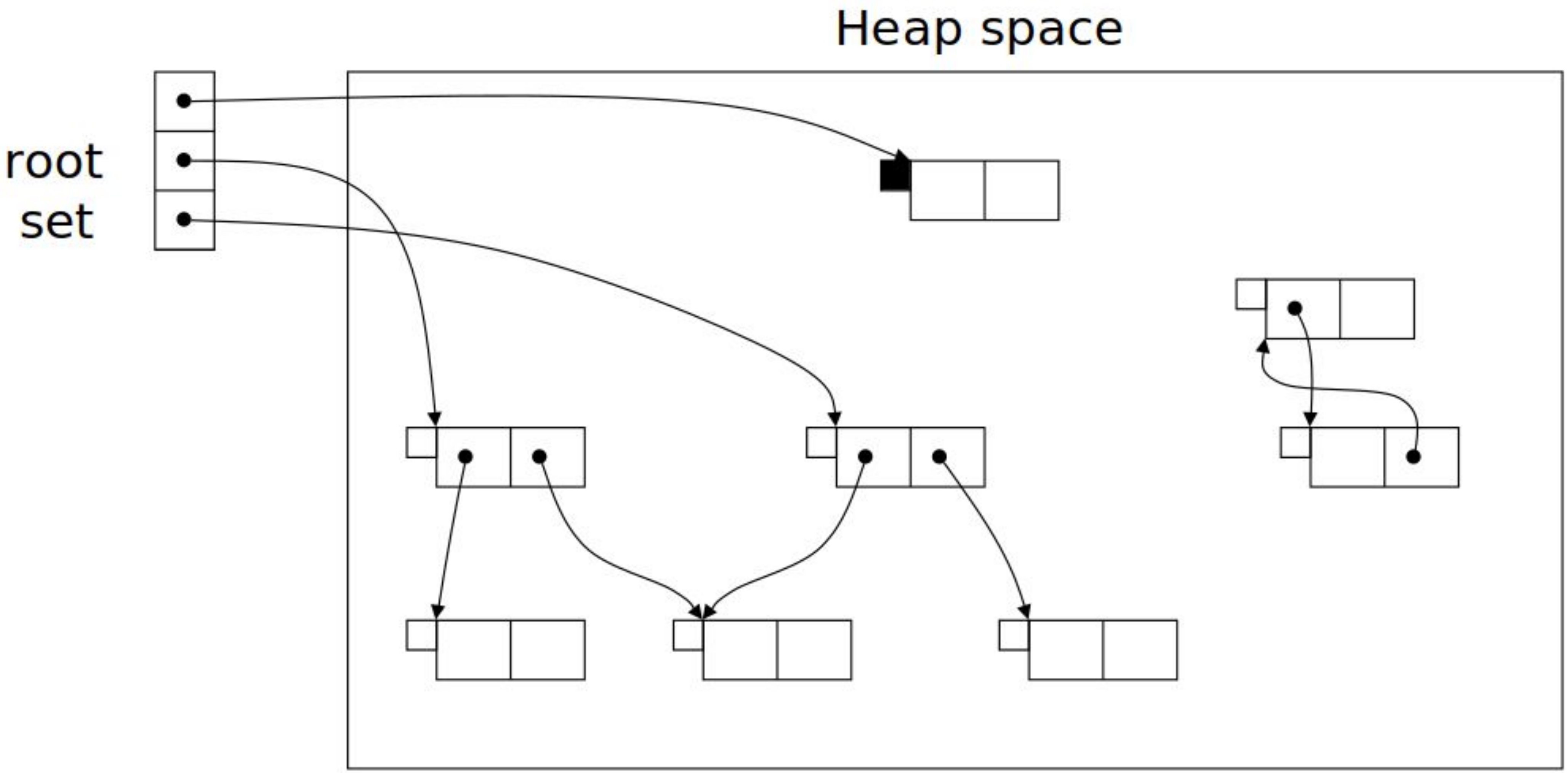
Tracing - Mark and Sweep GC

- Each cell has a mark bit
- Garbage remains unreachable and undetected until heap is used up; then GC goes to work, while program execution is suspended
- Marking phase
  - Starting from the roots, set the mark bit on all live cells
- Sweep phase
  - Return all unmarked cells to the free list
  - Reset the mark bit on all marked cells

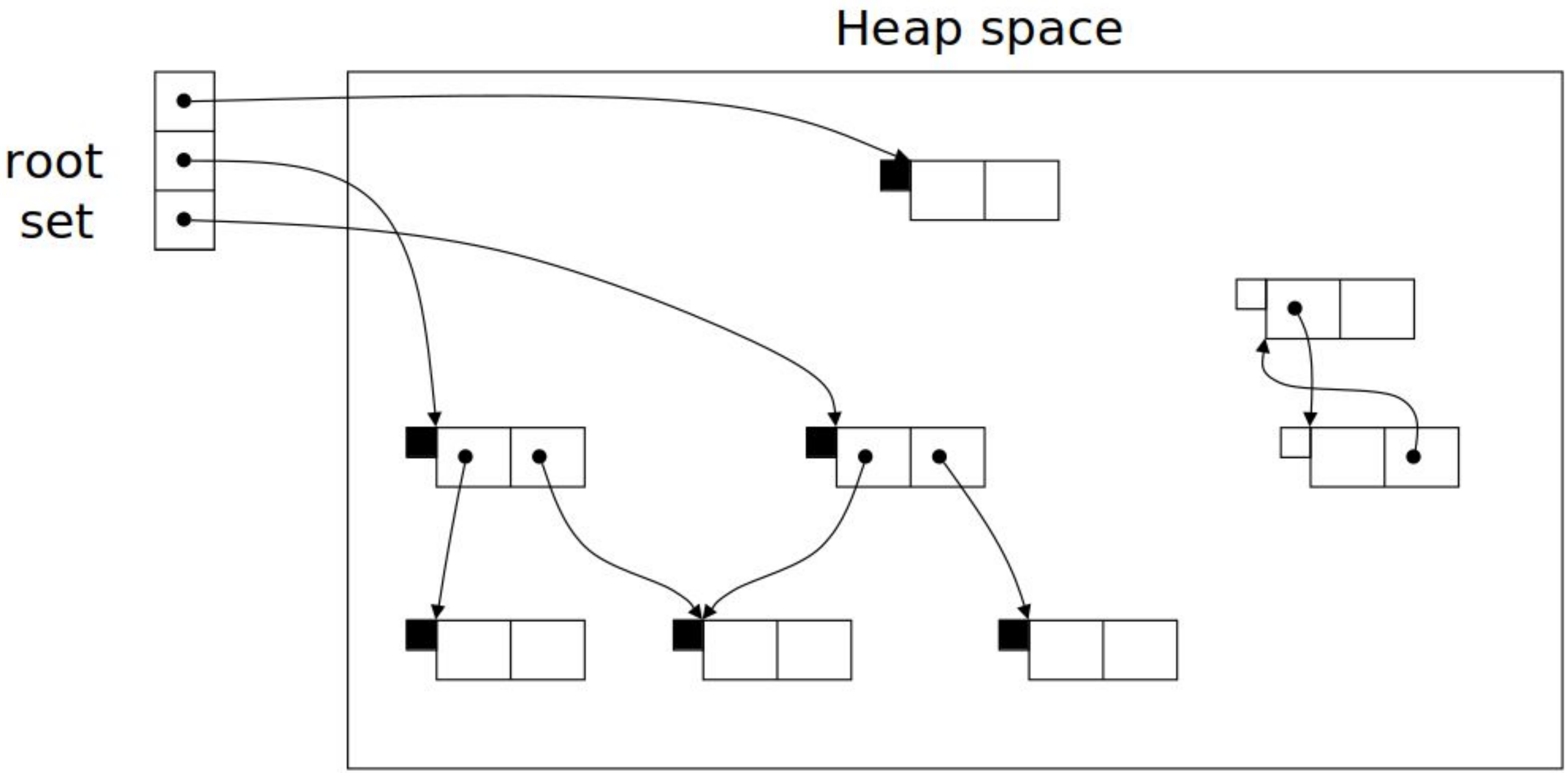
Mark and Sweep GC (Example)



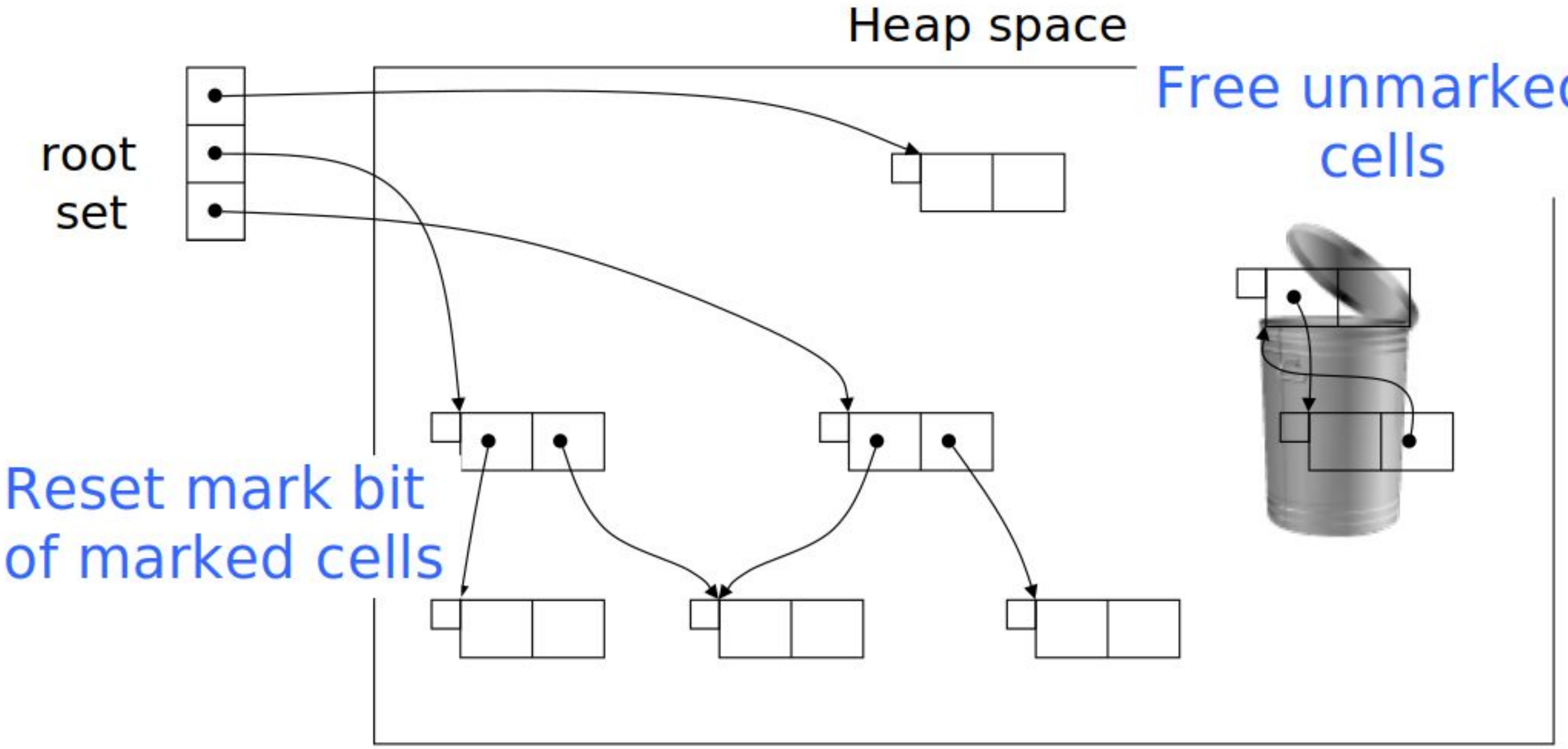
Mark and Sweep GC (Example)



Mark and Sweep GC (Example)



Mark and Sweep GC (Example)



Mark and Sweep GC - Pros and Cons

Pros:

- Handles cyclic data structures correctly
- Minimal space overhead
  - 1 bit used for marking cells may limit max values that can be stored in a cell (e.g., for integer cells)

Cons:

- Normal execution is suspended
- May touch all virtual memory pages
  - May lead to excessive paging if the working-set size is small and the heap is not all in physical memory
- Heap may fragment
  - Cache misses, page thrashing; more complex allocation

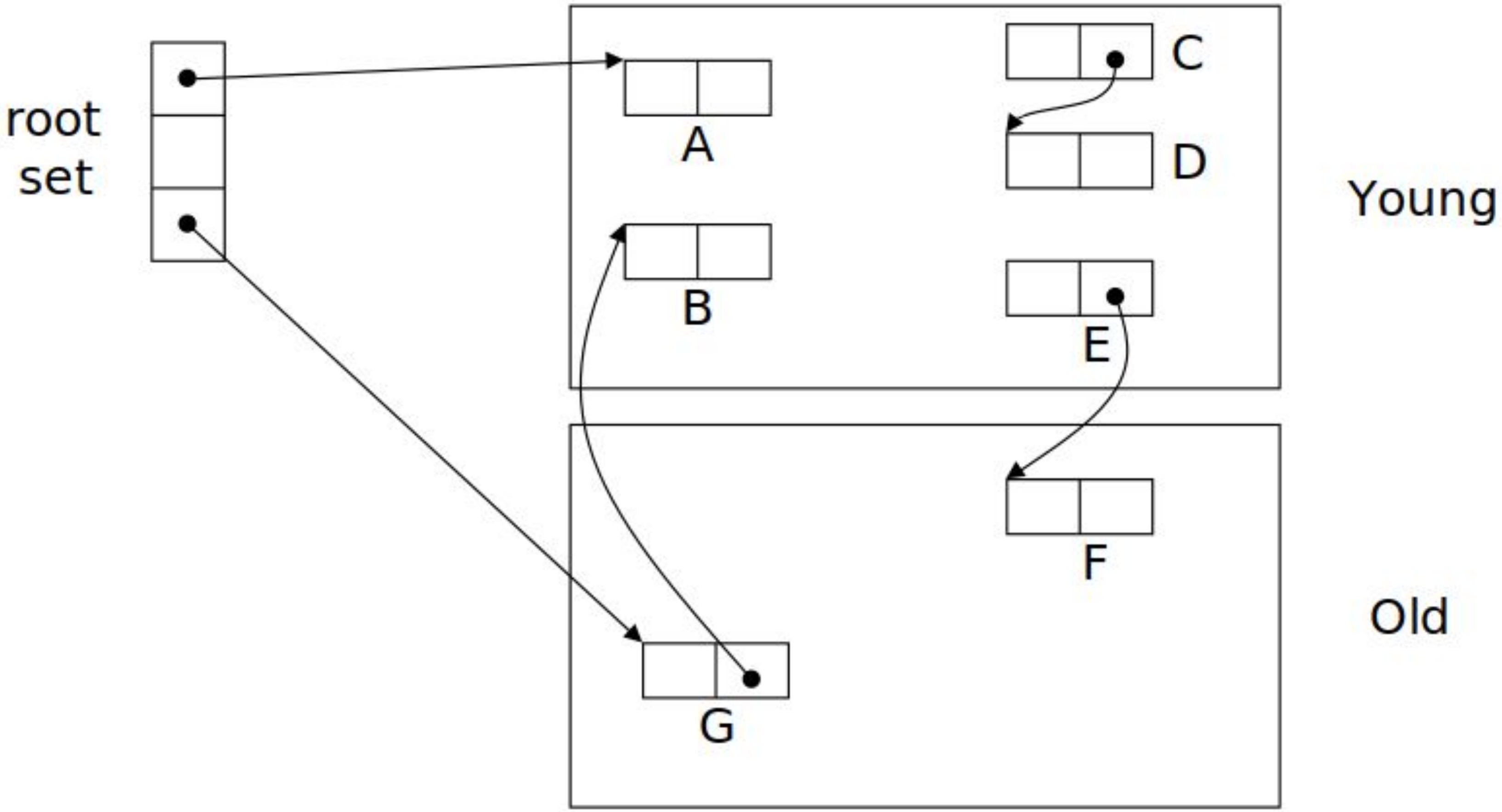
Modern Techniques - Generational GC

Observation: "Most objects die young" - Ungar (1984)

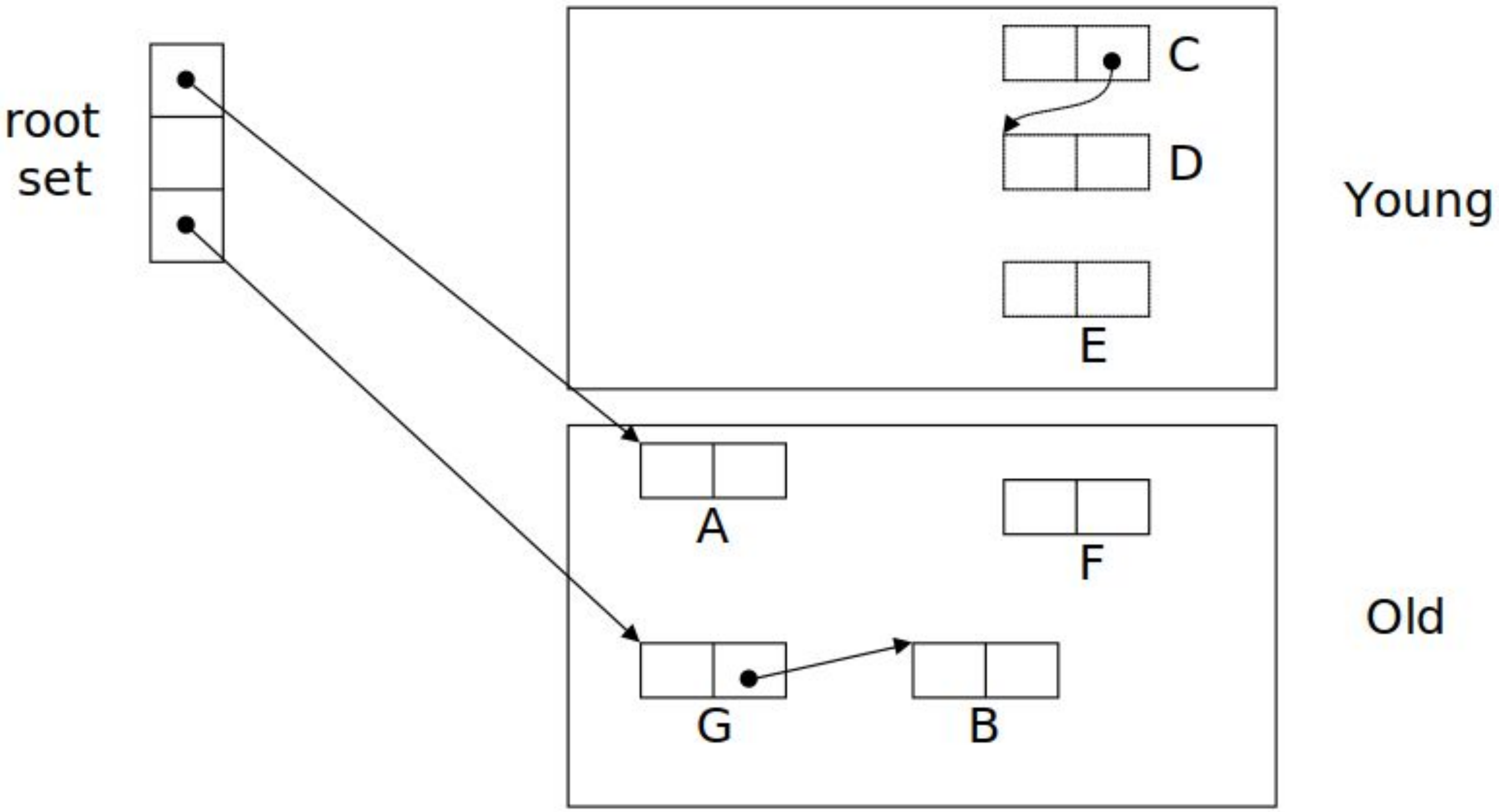
- Nested scopes are entered and exited more frequently, so temporary objects in a nested scope are born and die close together in time
- Inner expressions in Scheme are younger than outer expressions, so they become garbage sooner

- Divide the heap into generations, and GC the younger cells more frequently
  - Don't have to trace all cells during a GC cycle
  - Periodically reap the "older generations"
  - Amortize the cost across generations
  - Surviving young objects promoted into old generation

Generational GC - Example



Generational GC - Example



Generational GC - Pros

- Efficiency
    - The GC efficiently collects short-lived objects, minimizing the need for expensive major collections.
- Reduced Pause Times
    - Minor collections in the young generation are quick and have minimal performance impact. Major collections in the old generation are infrequent, leading to shorter pause times.
- Optimized for Common Use Cases
    - Usually, most objects are short lived. Eg: Haskell: 75-95% die within 10KB, C: Over 50% is garbage within 10 KB, less than 10% lived longer than 32KB

Generational GC - Cons

- Increased Complexity
  - Managing multiple memory regions (young and old generations) adds complexity to the GC implementation.
- Promotion Overhead
  - Frequent promotion of objects from the young to the old generation can increase overhead, especially if many objects are promoted prematurely.
- Fragmentation
  - The old generation may suffer from fragmentation, leading to inefficient use of memory and potentially higher GC costs for defragmentation.

Case Study: GC in Java

GC in Java (JVM)

- Implementation in JVM
- Uses a collection of GC algorithms like Mark and Sweep, Generational etc.
- Uses GC roots to identify live and dead objects
  - Classes loaded by system class loader (not custom class loaders)
  - Live threads
  - Local variables and parameters of the currently executing methods
  - Objects used as a monitor for synchronization etc.
- Traverses the whole **object graph** in memory, starting from those GC Roots and following references from the roots to other objects.

Mark and Compact GC

- A variant of Mark and Sweep GC
- Occurs in three phases:
  - Mark
    - mark objects as alive

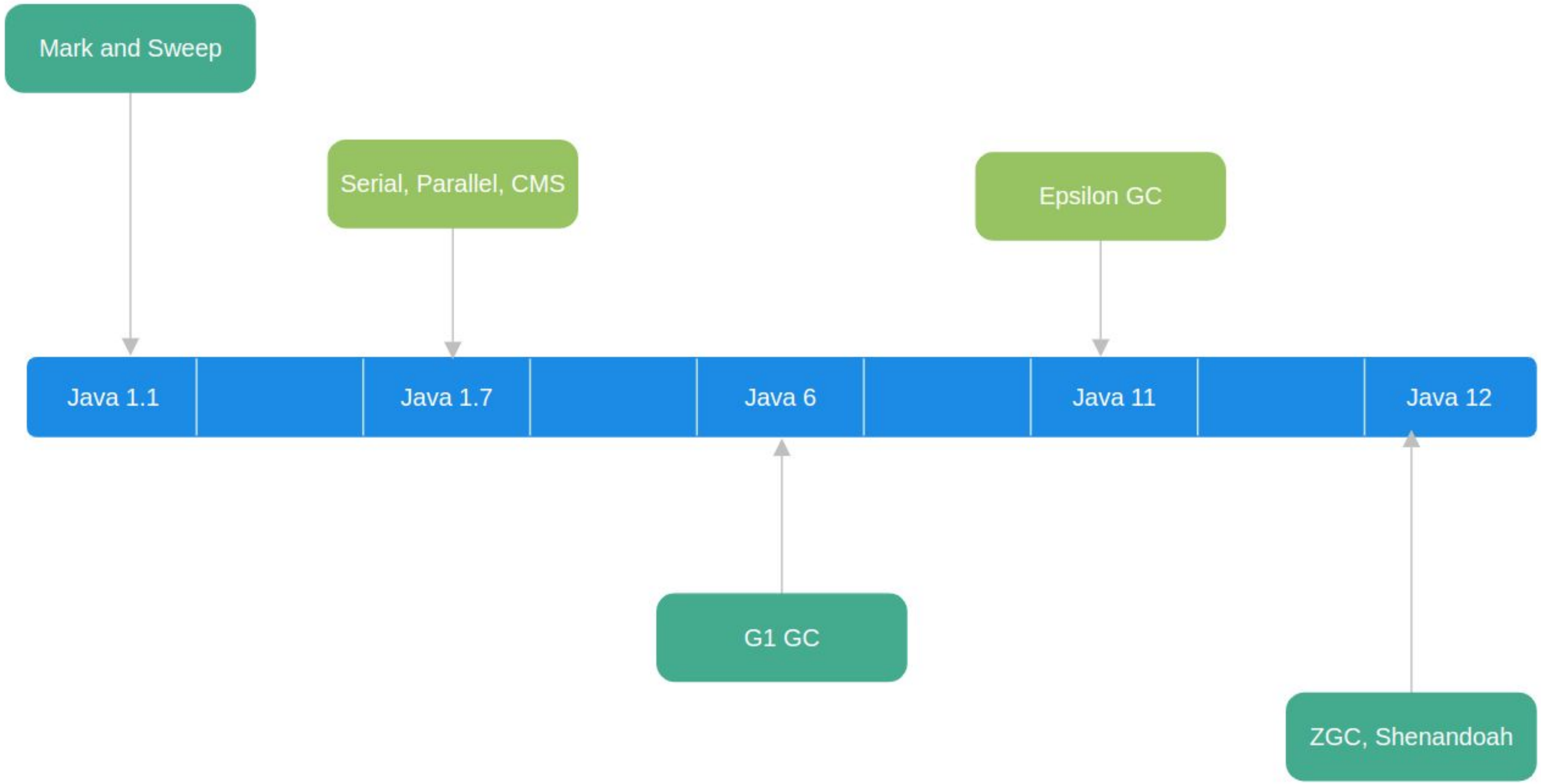
  - Sweep
    - release the memory fragments associated with dead objects

Mark and Compact GC

- Compact
  - compacts the remaining(live cells) memory together

- Reduces **fragmentation**
- In addition Java uses Generational GC
  - Divided into two generations:
    - *Young Generation* (Minor GC)
      - Eden Space
      - Survivor Spaces (FromSpace and ToSpace)
    - *Old Generation* (Major GC)

Evolution of GC in Java



Types of GC in Java

Serial GC:

- Simplest type of GC
- Stops all application threads to run GC
- **Pros:** simple to use, good for single threaded applications
- **Cons:** high pause times, not for multithreaded applications

Parallel GC:

- Designed to work with multiple threads and processors
- **Pros:** Better performance than Serial GC in multithreaded environments
- **Cons:** Still causes application pauses which can be more than serial GC

CMS (Concurrent Mark and Sweep) GC:

- Minimizes pauses by doing most of the work concurrently with appl. threads
- **Pros:** Good for programs with high memory requirement but strict pause time
- **Cons:** CPU intensive, can lead to fragmentation

G1(Garbage First) GC (Default in current Java):

- Generational GC
- Divides the heap into regions
- Prioritizes collections of regions containing most garbage
- **Pros:** Suitable for large heaps, minimizes pause times, compacts free spaces without lengthy pauses
- **Cons:** Can be more CPU intensive, not suitable for programs with less than 50% heap usage

Epsilon GC (No-op GC):
- Handles memory allocation but has no reclamation mechanism
- Once Java heap is used up, JVM shut down
- **Pros:** Used for Ultra-low-latency systems with known memory footprint
- **Cons:** No reclamation leading to *OutOfMemoryErrors*

Shenandoah:
- G1 GC on steroids
- Can concurrently relocate objects with application (G1 could not do that)
- **Pros:** Suitable for programs with large heap and minimum pause times

ZGC:
- Extremely low pause times
- **Pros:** For ultra low latency systems (less than 10ms of pause) and high memory based systems (Multi terabytes)
- **Cons:** More CPU intensive

Drawbacks of GC

Problems with automatic garbage collection:

- **Unpredictable Performance:**
  - GC pauses the program at any point
- **Scalability:**
  - As heap sizes increase, performance drops
- **Heavy Resource Usage:**
  - GC-languages tend to use 10X more memory than non-GC languages
- **Degraded performance:**
  - GC scan the entire heap which takes time as well as contaminates the cache

Interesting Point: Discord moved from Go to Rust for its degraded performance due to GC (Read more: Link)

Solution?

Solutions:

- Use Rust
  - Rust has a unique memory ownership transfer model
  - Similar to move semantics in C++ but it is present by default

Example:

```
fn main() {
    let a = String::from("hello");
    let b = a; // copy the value a into b (This transfers ownership, a is
unusable now)
    println!("{}", a) // This will throw an error because a has been moved or
ownership has been transferred
    println!("{}", b) // hello
}
```

- Use C++ smart pointers like `std::unique_ptr`, `std::shared_ptr` etc.
  Example:

```
#include <iostream>
#include <memory>

struct Person{
    std::string name;
    int age;

    Person(std::string _name, int _age):
name(_name), age(_age) {}
};

int main() {
    std::unique_ptr<Person> p = new
Person("John", 20);

    // Perform some tasks
    //  No memory leak
    return 0;
}
```

```
#include <iostream>


struct Person{
    std::string name;
    int age;

    Person(std::string _name, int _age):
name(_name), age(_age) {}
};

int main() {
    Person *p = new Person("John", 20);

    // Perform some tasks
    // Memory leak as p is not freed
    return 0;
}
```

To know more, read: Link1, Link2, Link3, CppCon video

Still want to use GC?

How to decide which GC to use?

Criteria	Comments
Heap Size	G1, ZGC and Shenandoah can handle large heap size
Pause time	For strict pause time requirements, use G1, CMS, ZGC or Shenandoah
Throughput	Parallel GC for maximizing throughput
Memory Overhead	ZGC and Shenandoah can consume more memory
CPU overhead	ZGC and Shenandoah consume more memory than others

References

- Kernel Reference Counting: https://docs.kernel.org/core-api/kref.html
- The Garbage Collection Handbook, Richard Jones, Antony Hosking, Eliot Moss : https://gchandbook.org/
- GC in Java: Link
- Papers:
  - Concurrent GC:
    - E. W. Dijkstra, On-the-Fly Garbage Collection: An Exercise in Cooperation, CACM '78
  - Generational GC:
    - Andrew W. Appel, Simple Generational Garbage Collection and Fast Allocation, Software Practice and Experience '89
  - Reference Counting:
    - Bavid F. Bacon, V.T. Rajan, Concurrent Cycle Collection in Reference Counted Systems, ECOOP '01
    - H. Azatchi and E. Petrank, Integrating Generations with Advanced Reference Counting Garbage Collectors, CC '03
  - Mark Compacting:
    - F. Lockwood Morris, A time- and space-efficient garbage compaction algorithm, CACM '78