EECS 483: Compiler Construction

Lecture 14:
Memory Management, Garbage Collection

March 10
Winter Semester 2025

Slides adapted from David Walker, Cornell University

Announcements

Midterm on Tuesday, March 18, 6-8pm.

- Topics: anything covered in assignments 1-3 and lecture material before spring break
- Rooms DOW1010, DOW1017, DOW1018

Midterm review in lecture **March 12** (previously said March 17) and in Discussion this week bring questions about course material.

Assignment 4 (dynamic typing, heap allocation) released after the midterm

Memory Deallocation and Reuse

- Every modern programming language allows programmers to allocate new storage dynamically
  - New records, arrays, tuples, objects, closures, etc.
- Every modern language needs facilities for **reclaiming and recycling** the storage used by programs
- It's usually the most complex aspect of the run-time system for any modern language.

Memory Deallocation and Reuse

- Memory used for an object can be reused if it is garbage
- What is garbage?
  - A value is garbage if it will not be used in any subsequent computation by the program
- How do we determine which objects are garbage?

Identifying Garbage

- How do we determine which objects are garbage?
- Stack-allocation:
  - when we return or tail call, all objects in the stack frame are garbage so that memory can be reused.
  - See: our implementation of return, branch with arguments
  - this is an **under-approximation.** Objects allocated on the stack may remain long after they are no longer used.
- Is it easy to determine which objects are garbage?
  - No. It's undecidable. Eg:

```
if long-and-tricky-computation then use v
else don't use v
```

Identifying Garbage

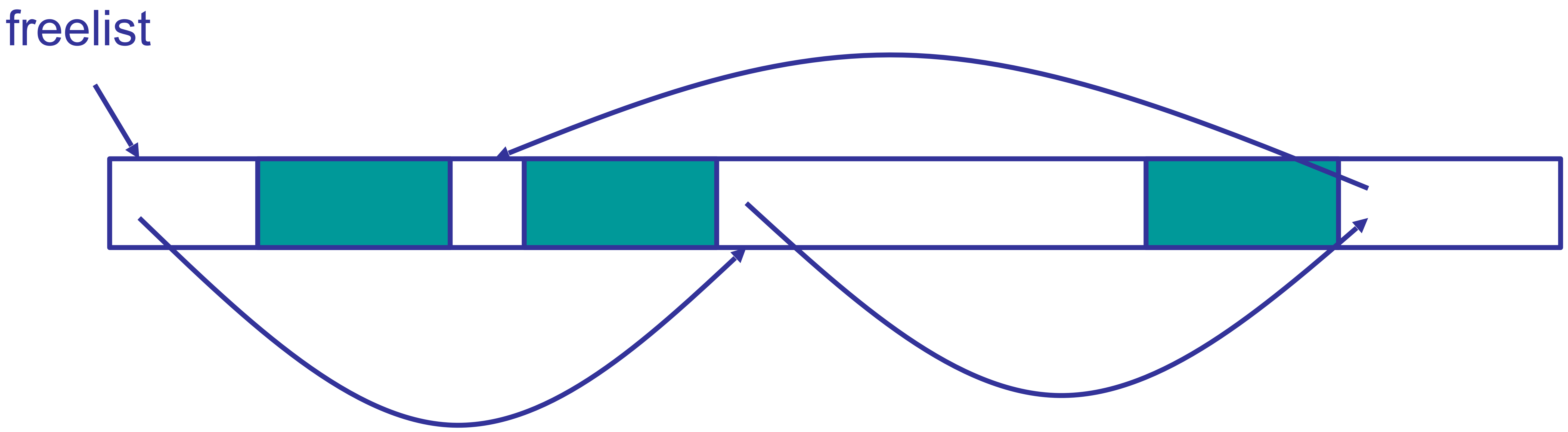
- Since determining which objects are garbage is tricky, people have come up with many different techniques
  - It’s the programmers problem:
    - Explicit allocation/deallocation
  - Reference counting
  - Tracing garbage collection
    - Mark-sweep, copying collection
    - Generational GC

Explicit MM

- User library manages memory; programmer decides when and where to allocate and deallocate
  - void* malloc(long n)
  - void free(void *addr)
  - Library calls OS for more pages when necessary
  - Advantage: if you work hard, you can free at the exact right time for their program
  - Disadvantage: people don't want to bother with such details if they can avoid it
  - Disadvantage: difficult to get right, dangerous when wrong

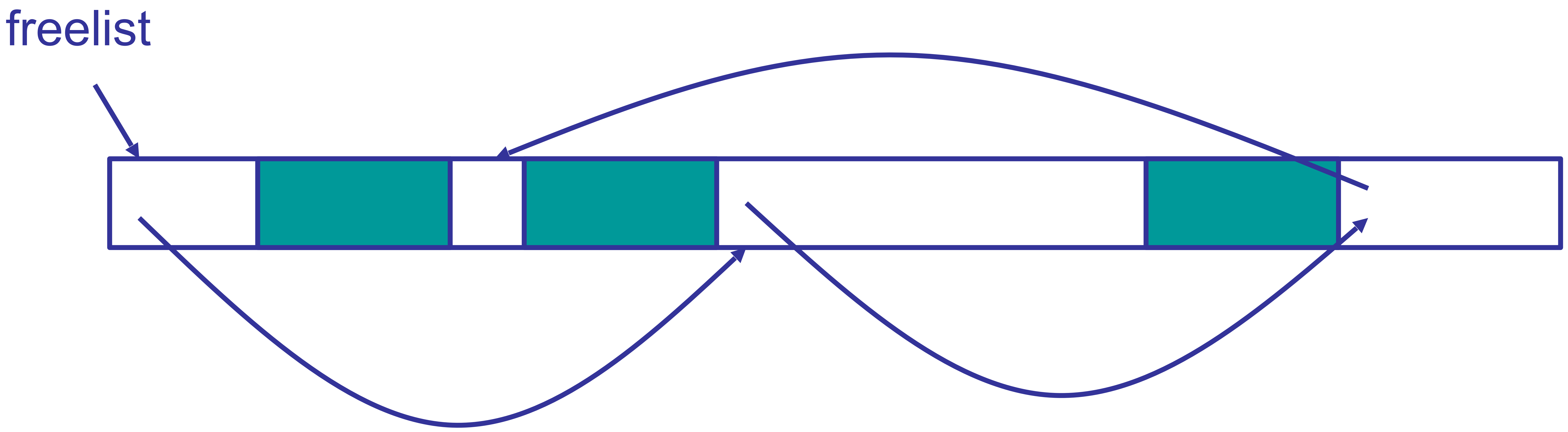
Explicit MM

- How does malloc/free work?
  - Blocks of unused memory stored on a freelist
  - malloc: search free list for usable memory block
  - free: put block onto the head of the freelist



Explicit MM

- Drawbacks
  - malloc is not free: we might have to do a significant search to find a big enough block
  - As program runs, the heap fragments leaving many small, unusable pieces



Explicit MM

- Solutions:
  - Use multiple free lists, one for each block size
    - Malloc and free become O(1)
    - But can run out of size 4 blocks, even though there are many size 6 blocks or size 2 blocks!
  - Blocks are powers of 2
    - Subdivide blocks to get the right size
    - Adjacent free blocks merged into the next biggest size
  - Still doesn't avoid fragmentation
    - 30% wasted space
    - No magic bullet: memory management always has a cost

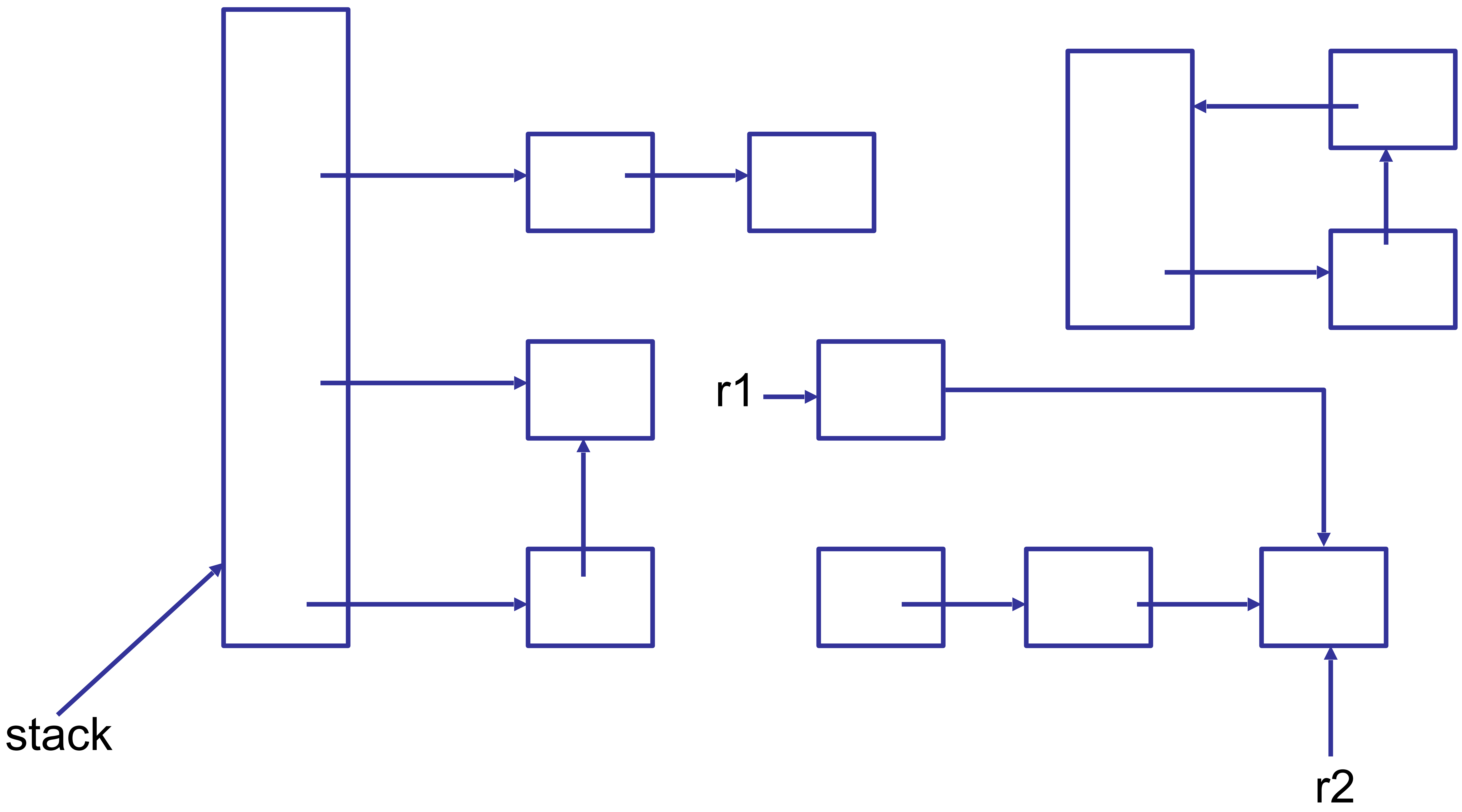
Automatic MM

- Languages with explicit MM are much harder to program than languages with automatic MM
  - Always worrying about dangling pointers, memory leaks: a huge software engineering burden with mistakes often leading to security vulnerabilities
  - Languages with unsafe, explicit MM are on the way out. Alternatives like Rust are being incorporated into high-performance settings like Linux kernel and web browsers.
  - Biden administration even instituted an executive order recommending new software be written in memory-safe languages!

Automatic MM

- Question: how do we decide which objects are garbage?
  - We conservatively approximate
  - Normal solution: an object is garbage when it becomes unreachable from the roots
    - The roots = registers, stack, global static data
    - If there is no path from the roots to an object, it cannot be used later in the computation so we can safely recycle its memory

Object Graph

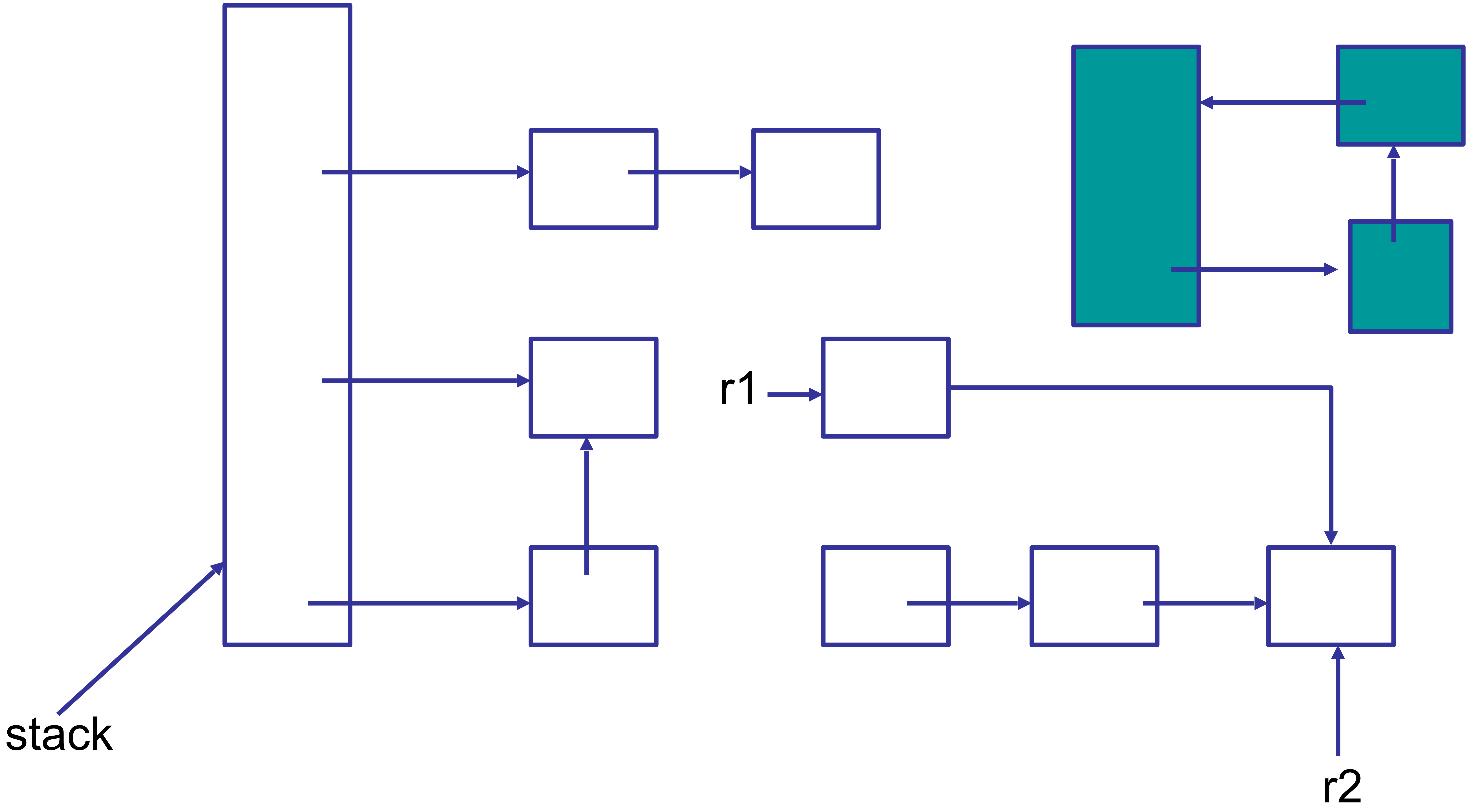


– How should we test reachability?

Reference Counting

- Keep track of the number of pointers to each object (the reference count).
- When the reference count goes to 0, the object is unreachable garbage

Object Graph



– Reference counting can't detect cycles

Reference Counting

– In place of a single assignment x.f = p:

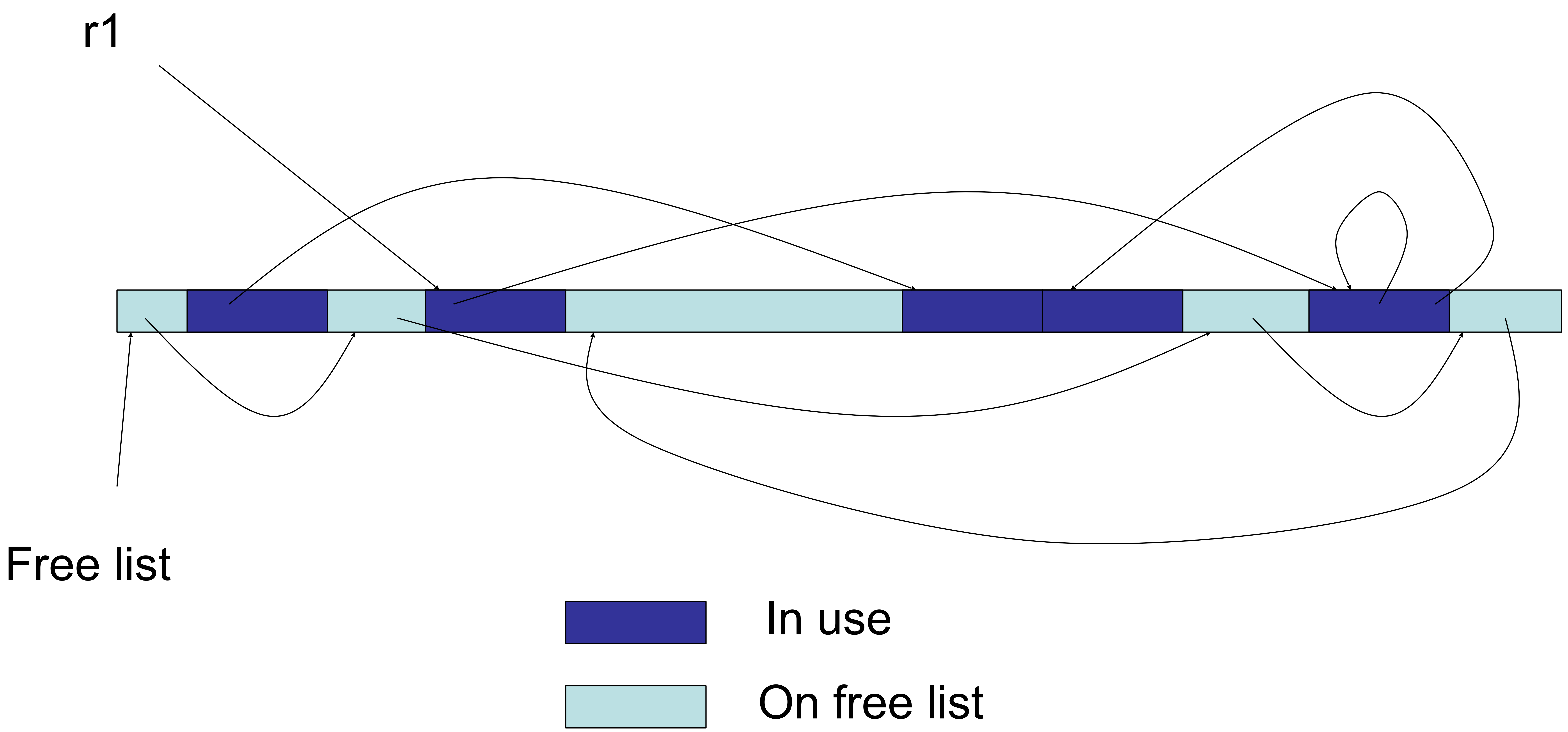
```
z = x.f
c = z.count
c = c – 1
z.count = c
If c = 0 call putOnFreeList(z)
x.f = p
c = p.count
c = c + 1
p.Count = c
```

- Ouch, that hurts performace-wise!
- Dataflow analysis can eliminate some increments and decrements, but many remain
- Reference counting used in some special cases but not usually as the primary GC mechanism in a language implementation

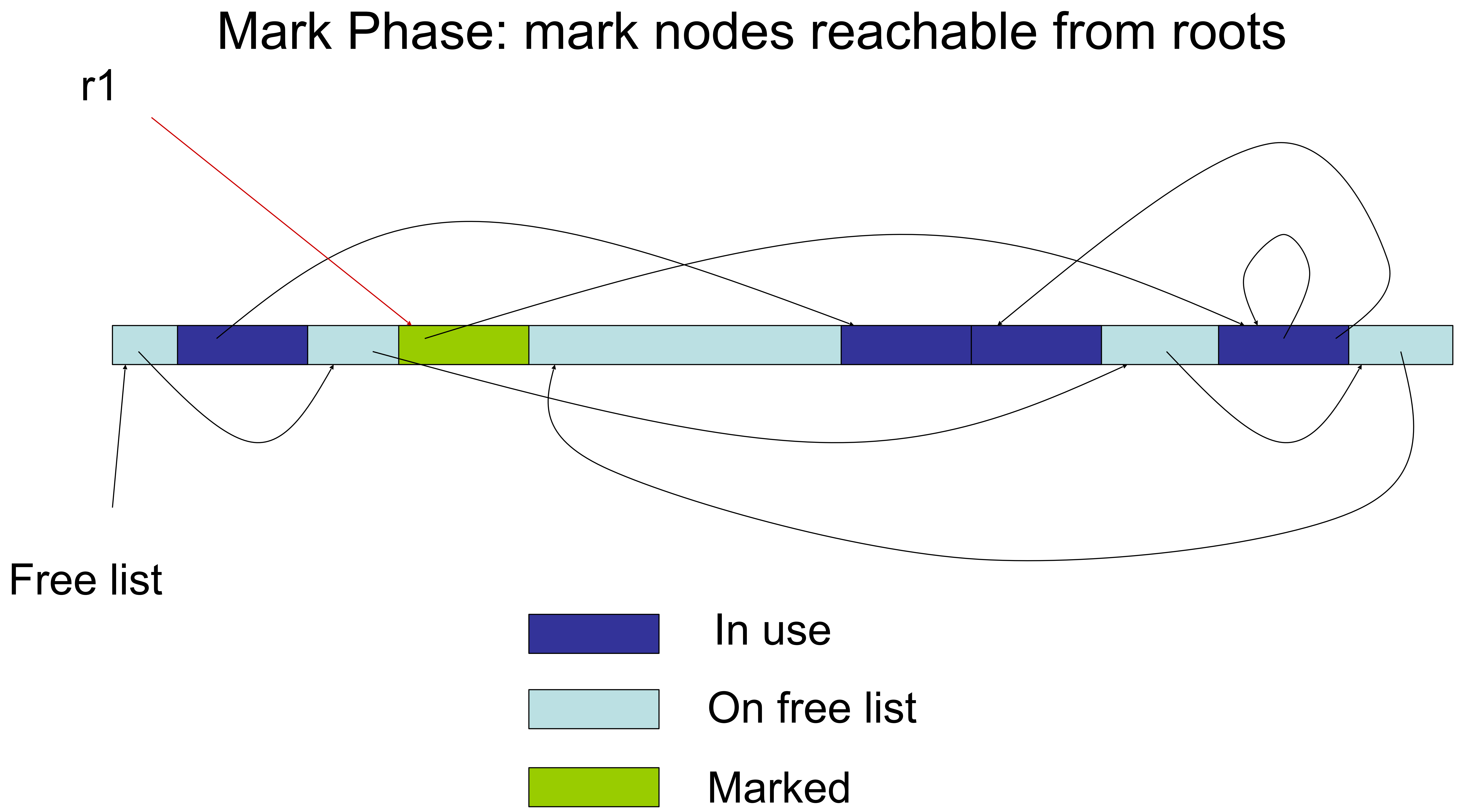
Mark-sweep

- A two-phase algorithm
  - Mark phase: Depth first traversal of object graph from the roots to mark live data
  - Sweep phase: iterate over entire heap, adding the unmarked data back onto the free list

Example



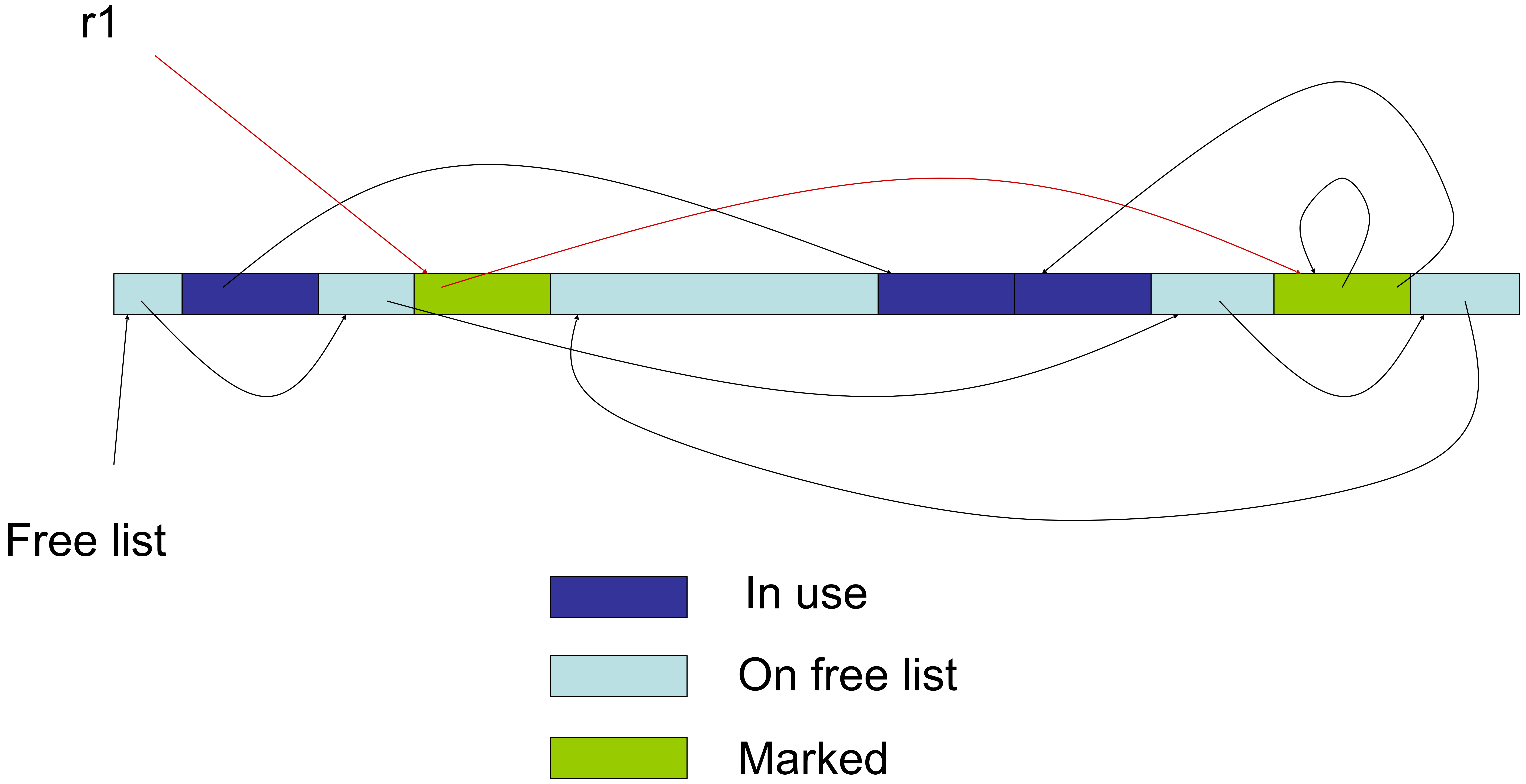
Example



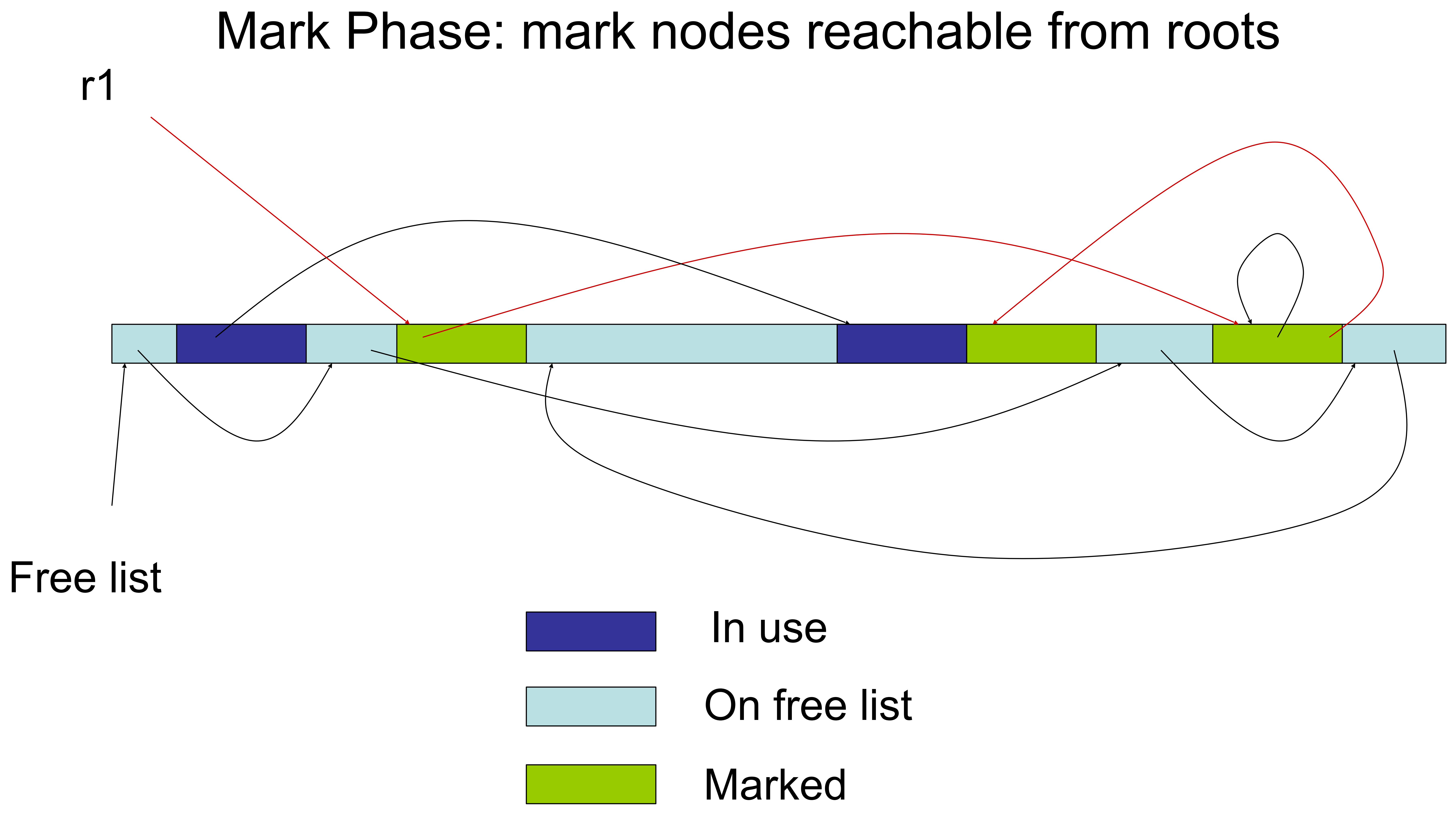
Mark Phase: mark nodes reachable from roots
r1
Free list
In use
On free list
Marked

Example

Mark Phase: mark nodes reachable from roots



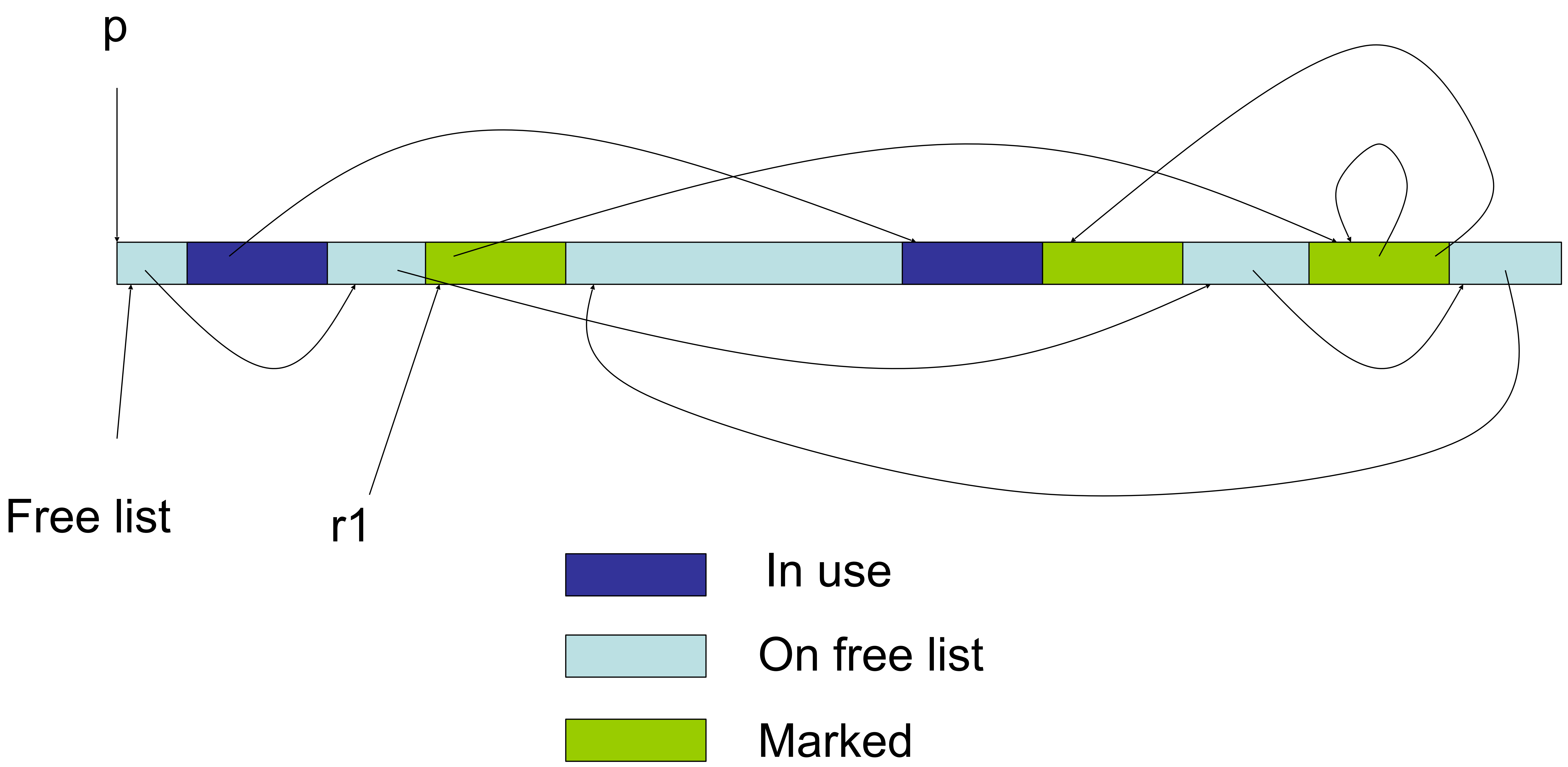
Example



Mark Phase: mark nodes reachable from roots
r1
Free list
In use
On free list
Marked

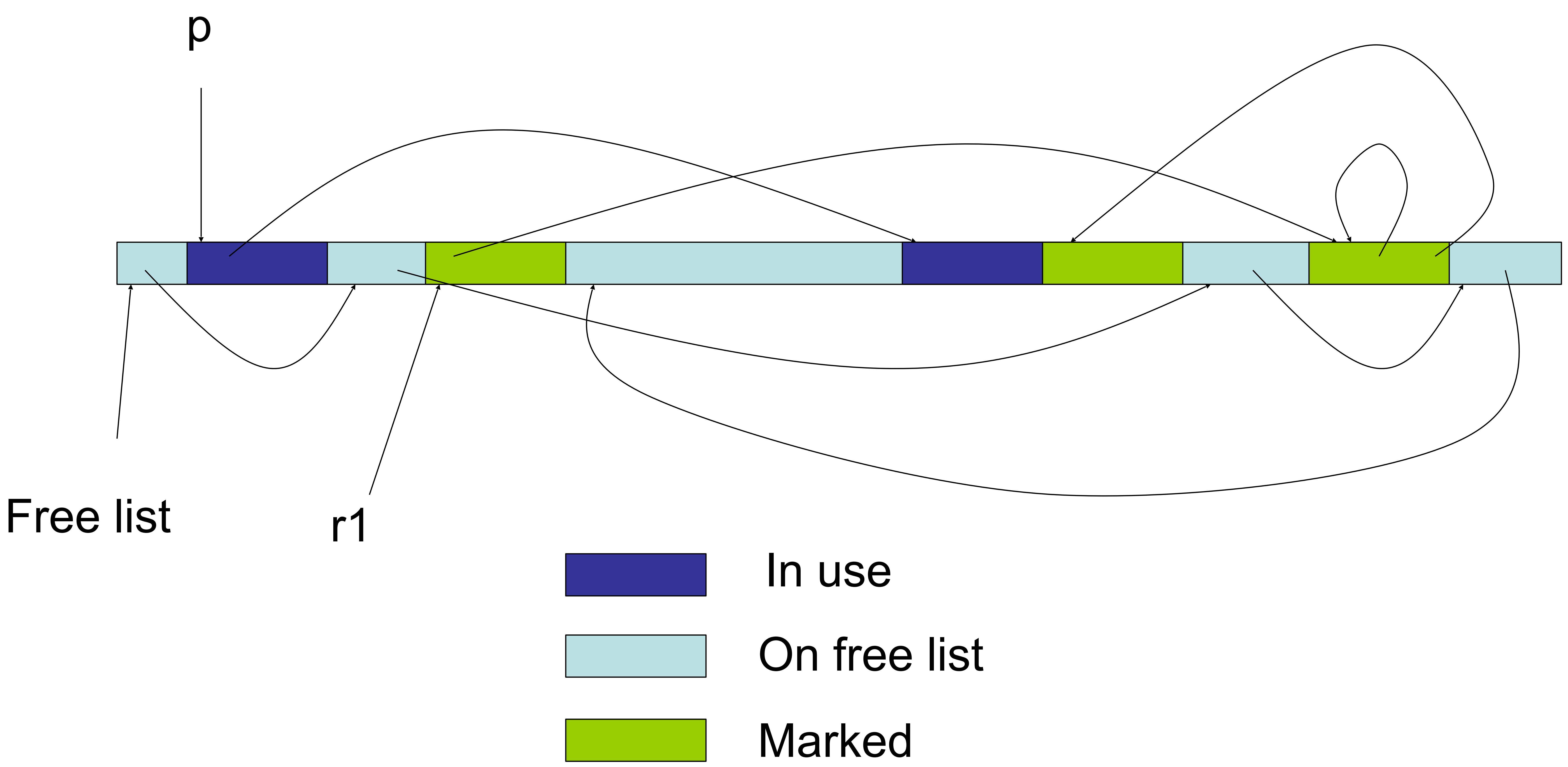
Example

Sweep Phase: set up sweep pointer; begin sweep



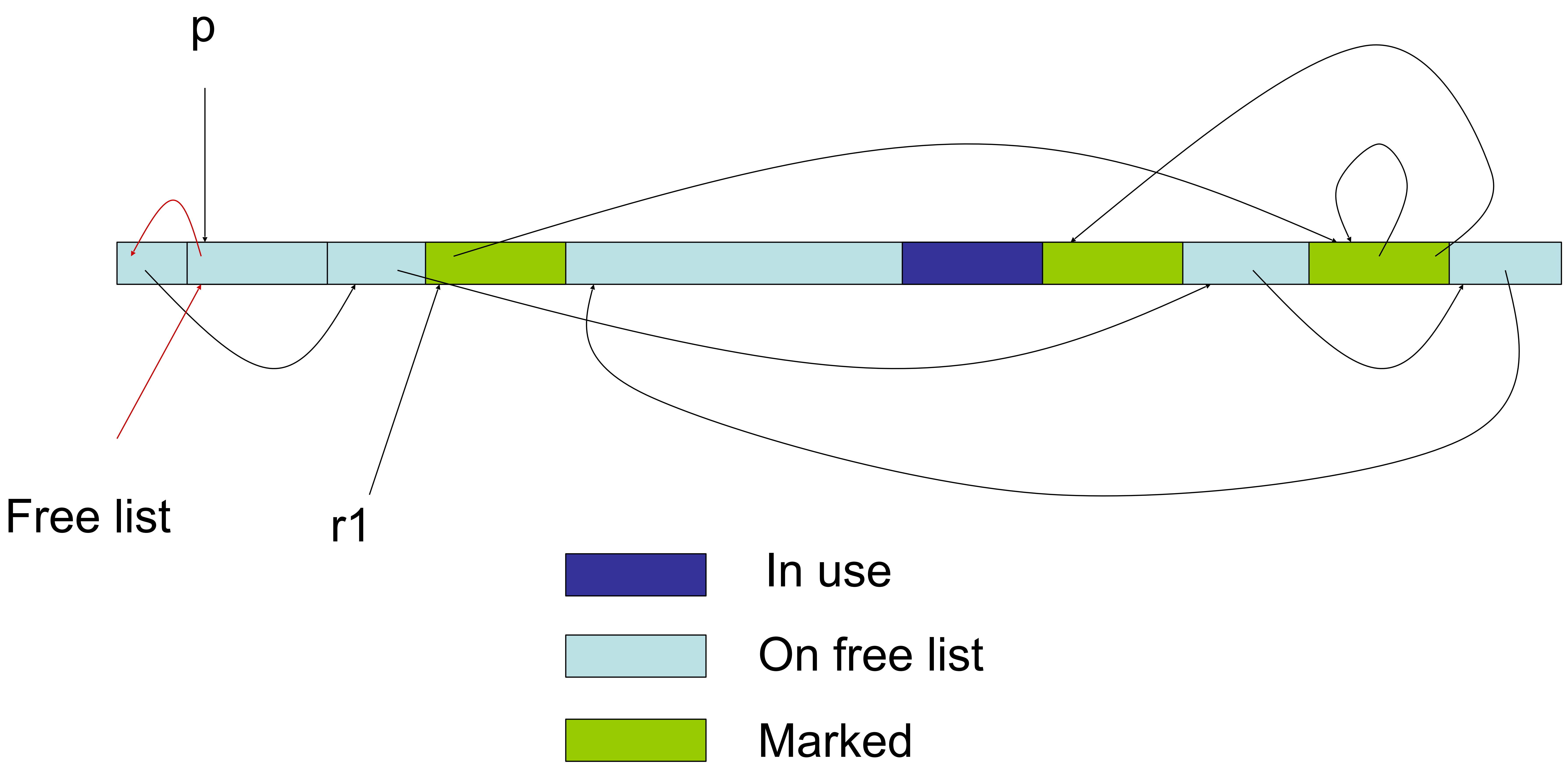
Example

Sweep Phase: add unmarked blocks to free list



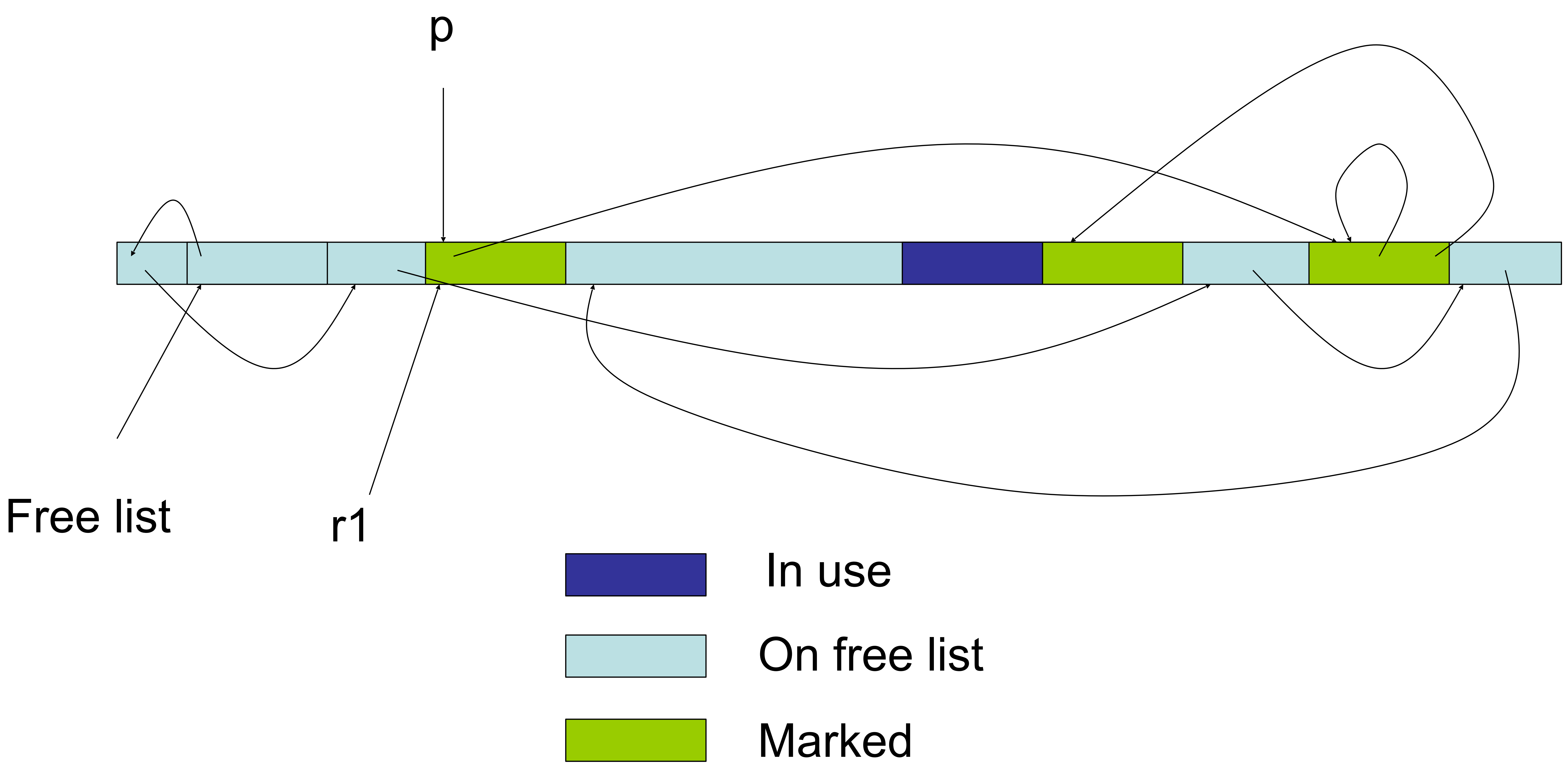
Example

Sweep Phase



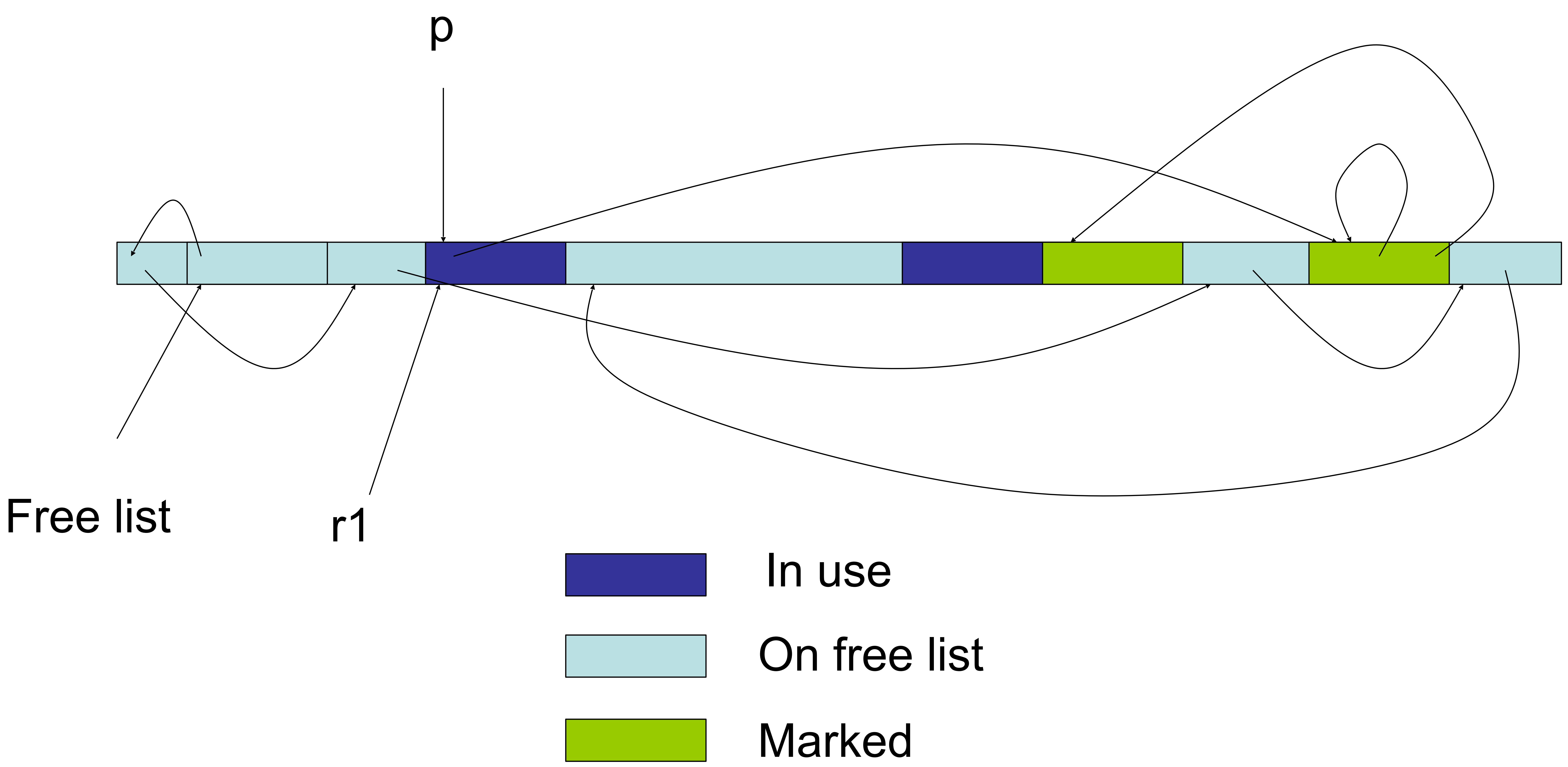
Example

Sweep Phase: retain & unmark marked blocks



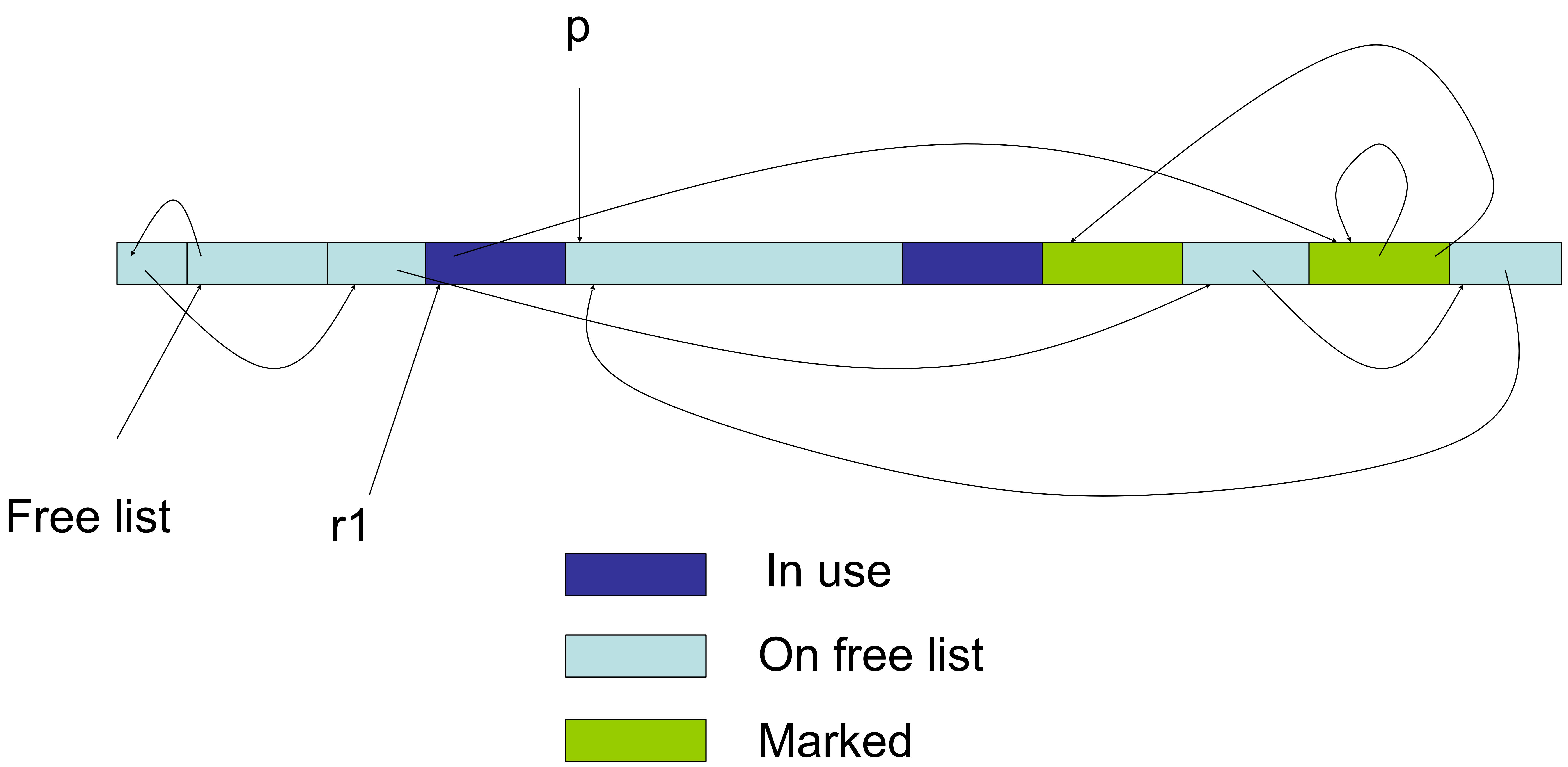
Example

Sweep Phase

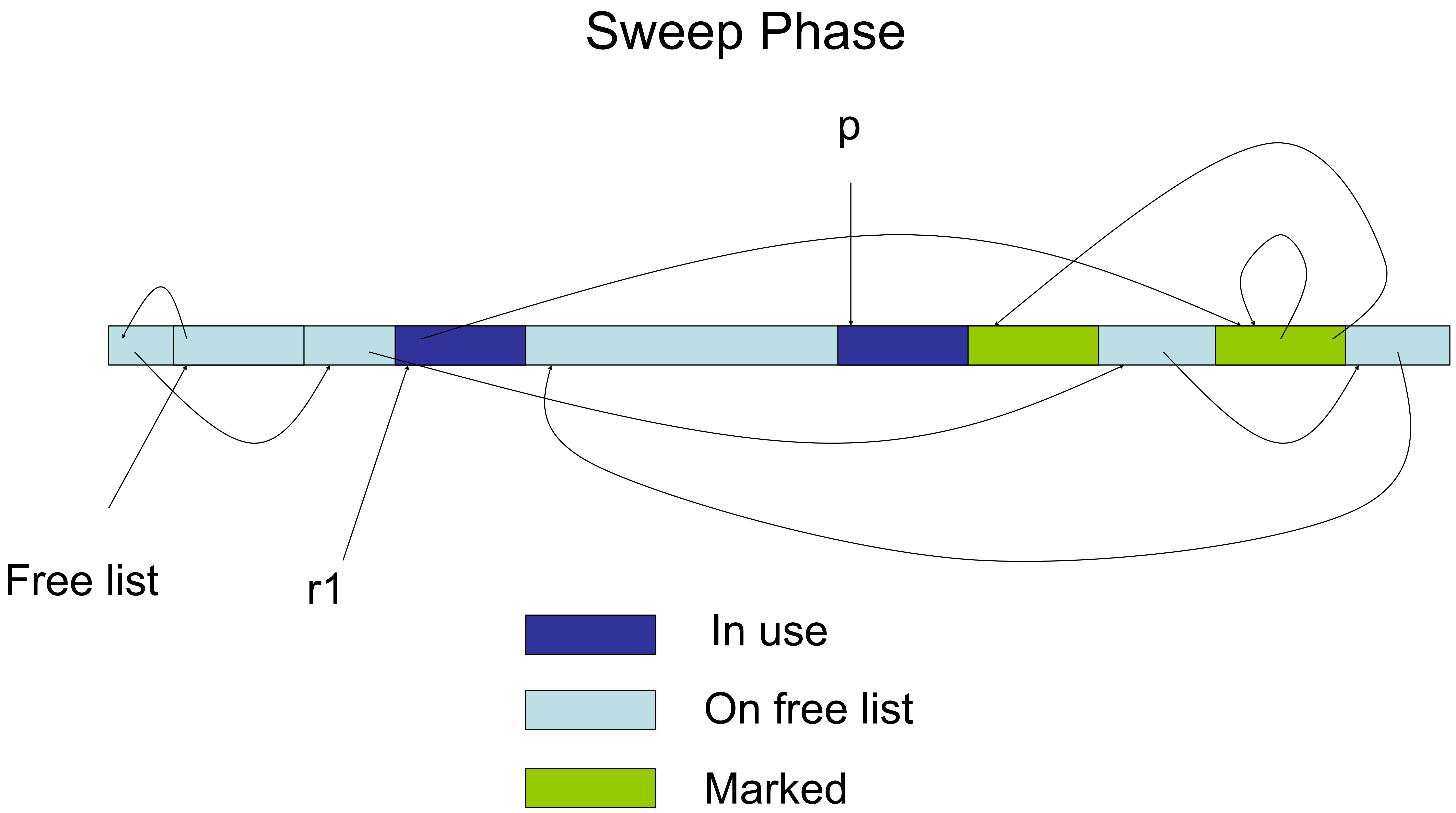


Example

Sweep Phase

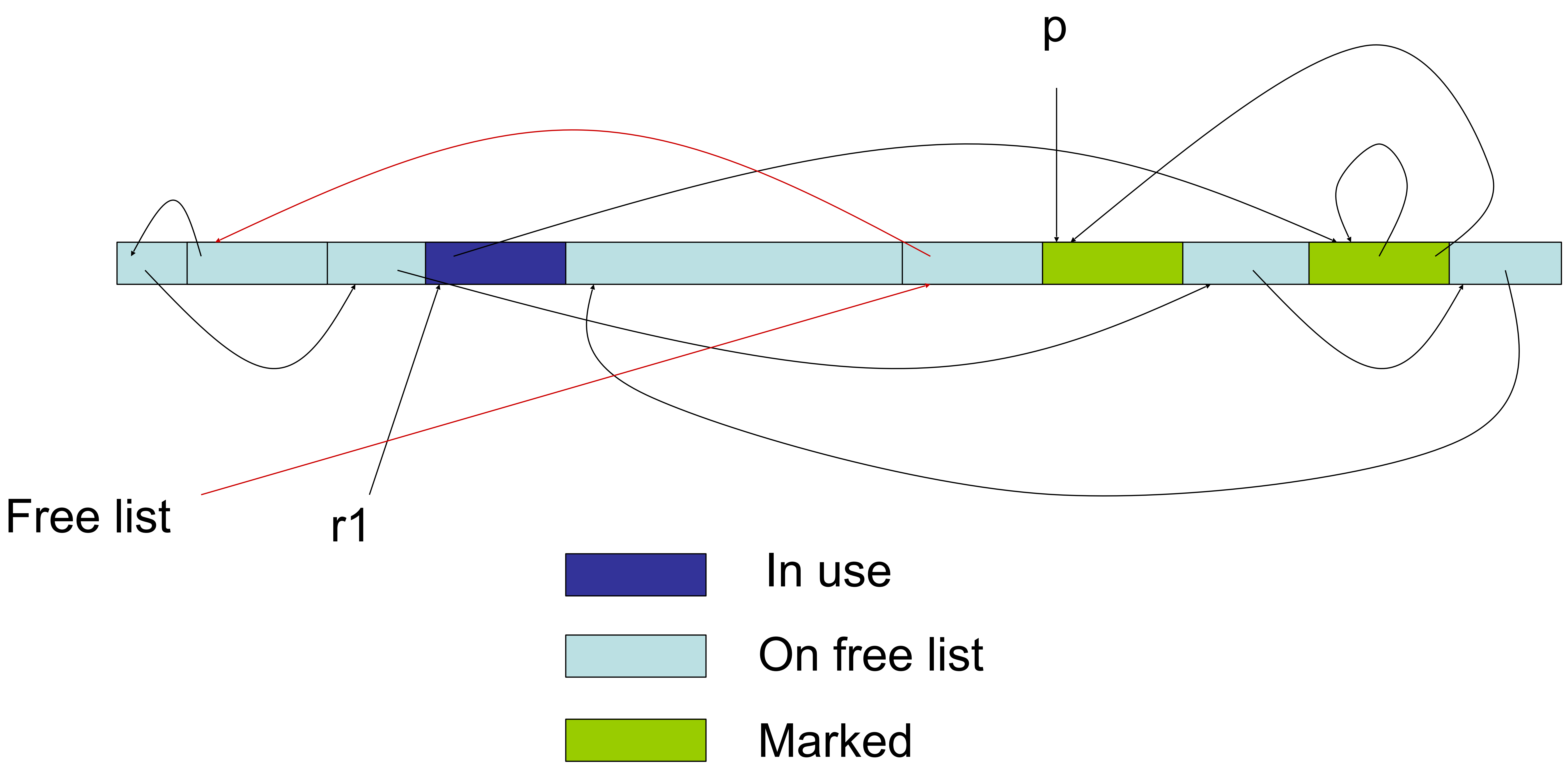


Example



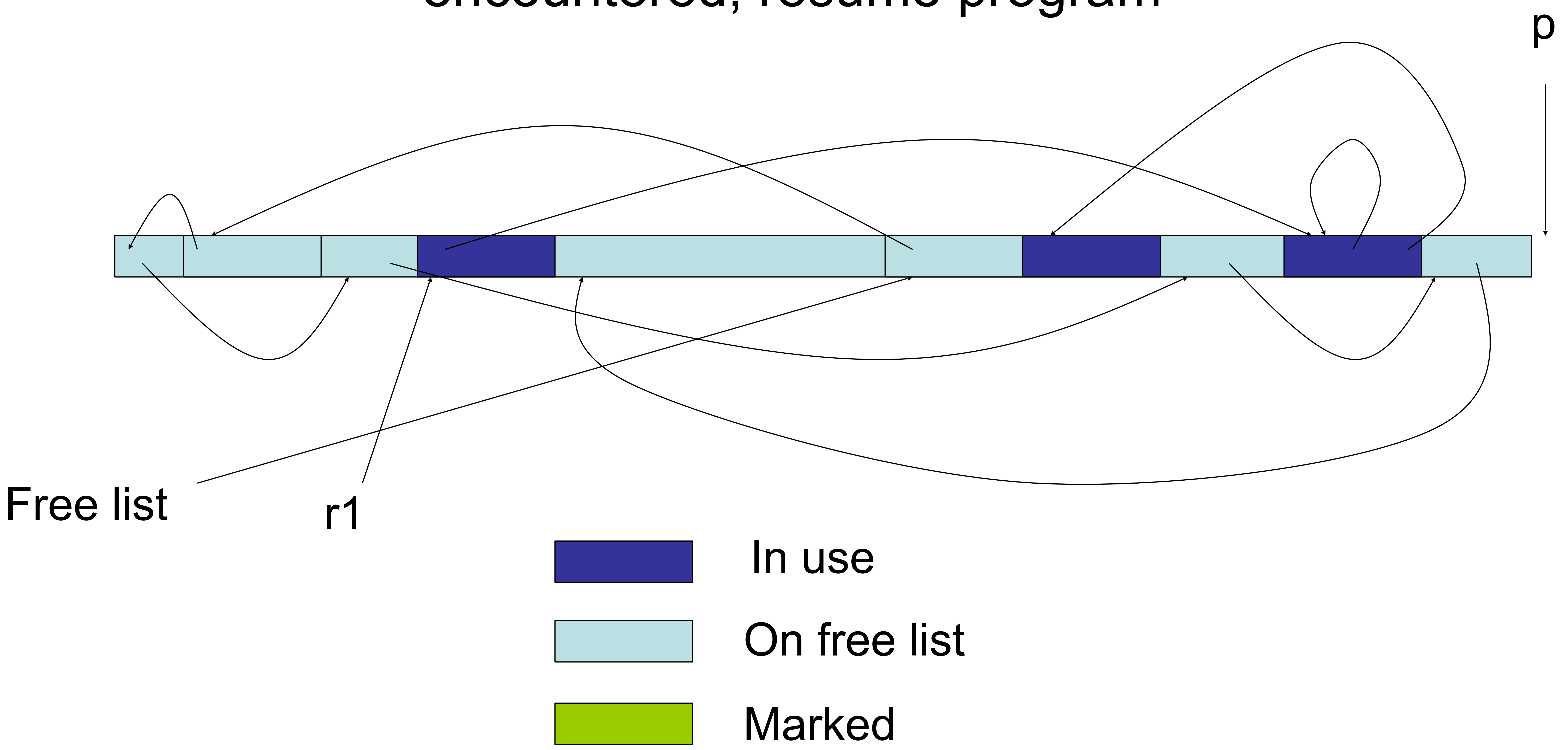
Example

Sweep Phase



Example

Sweep Phase: GC complete when heap boundary encountered; resume program



Cost of Mark Sweep

- Cost of mark phase:
  - O(R) where R is the # of reachable words
  - Assume cost is c1 * R (c1 may be 10 instr's)
- Cost of sweep phase:
  - O(H) where H is the # of words in entire heap
  - Assume cost is c2 * H (c2 may be 3 instr's)
- Analysis
  - Each collection returns H - R words
  - For every allocated word, we have GC cost:
    - ((c1 * R) + (c2 * H)) / (H - R)
  - R / H must be sufficiently small or GC cost is high
  - Eg: if R / H is larger than .5, increase heap size
    - Mark-sweep requires extra space like copying collection

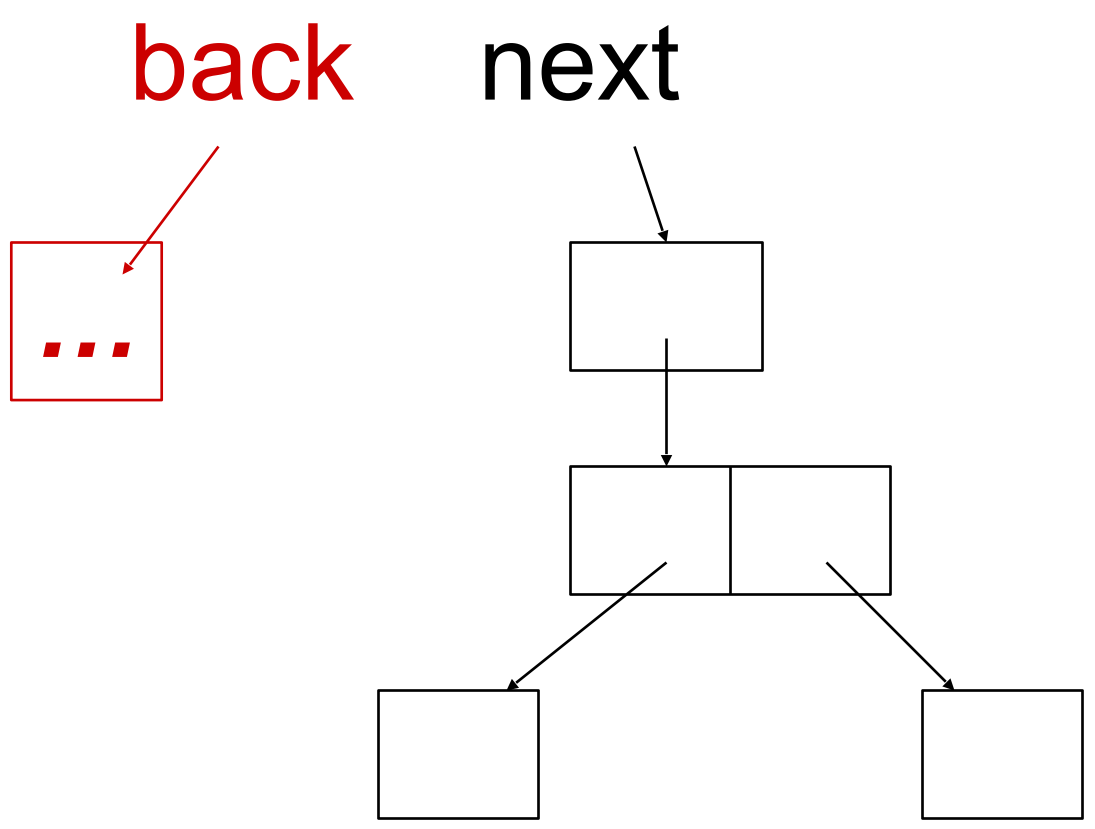
A Hidden Cost

- Depth-first search is usually implemented as a recursive algorithm
  - Uses stack space proportional to the longest path in the graph of reachable objects
    - one activation record/node in the path
    - activation records are big
  - If the heap is one long linked list, the stack space used in the algorithm will be greater than the heap size!!
  - What do we do?

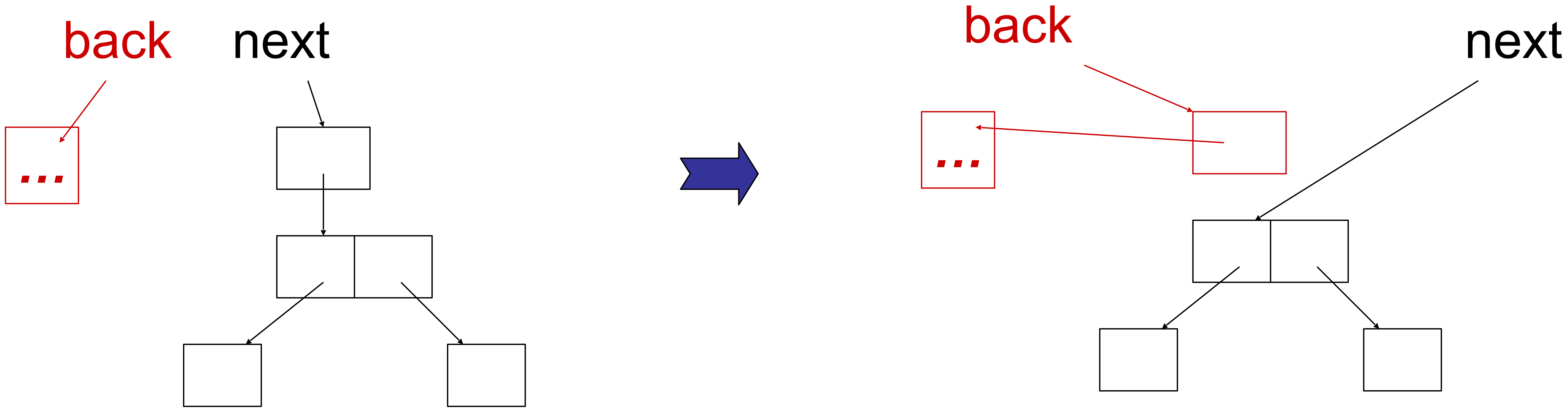
A nifty trick

- Deutsch-Schorr-Waite pointer reversal
  - Rather using a recursive algorithm, reuse the components of the graph you are traversing to build an explicit stack
  - This implementation trick only demands a few extra bits/block rather than an entire activation record/block
  - We already needed a few extra bits per block to hold the “mark” anyway

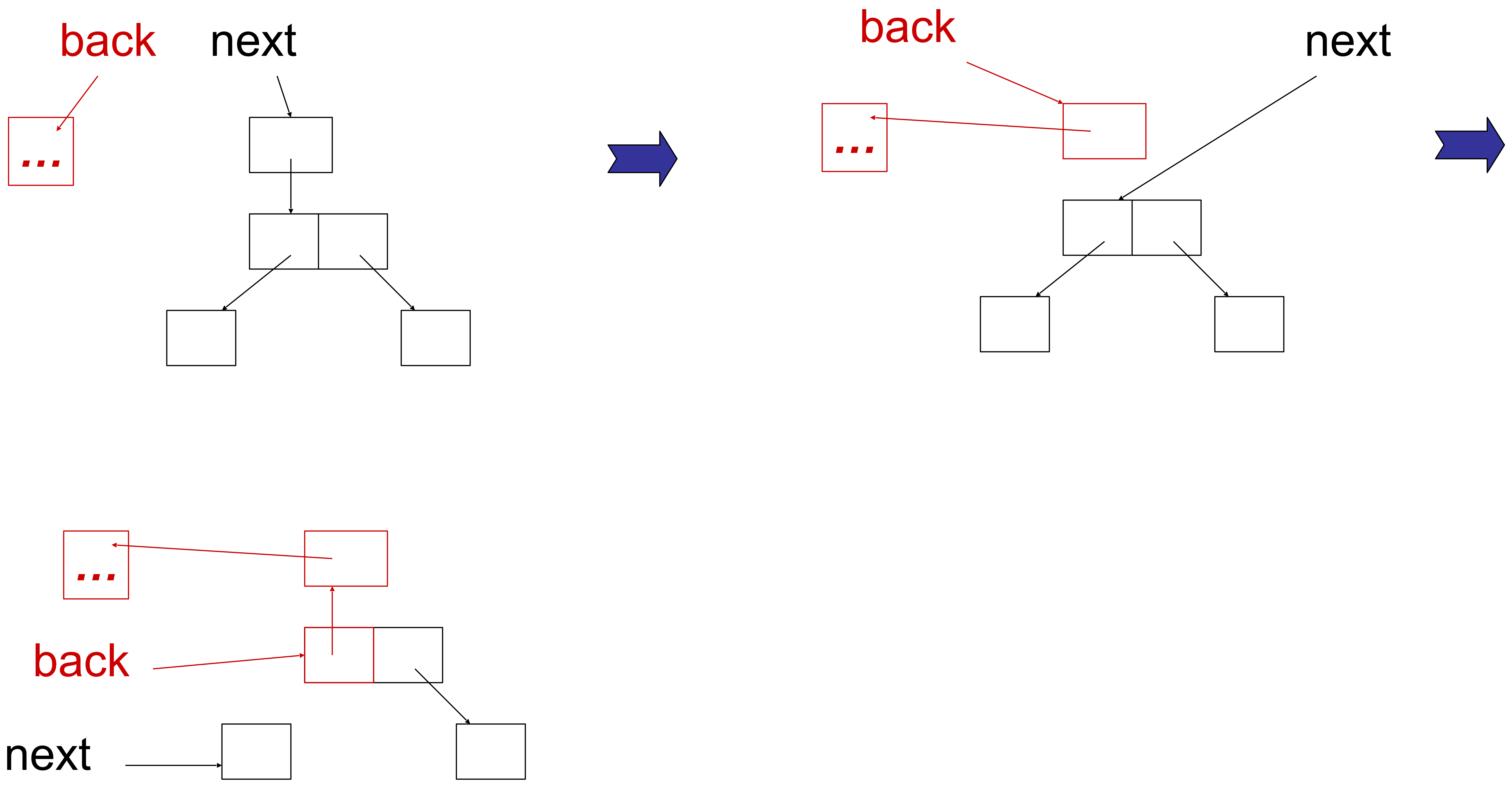
DSW Algorithm



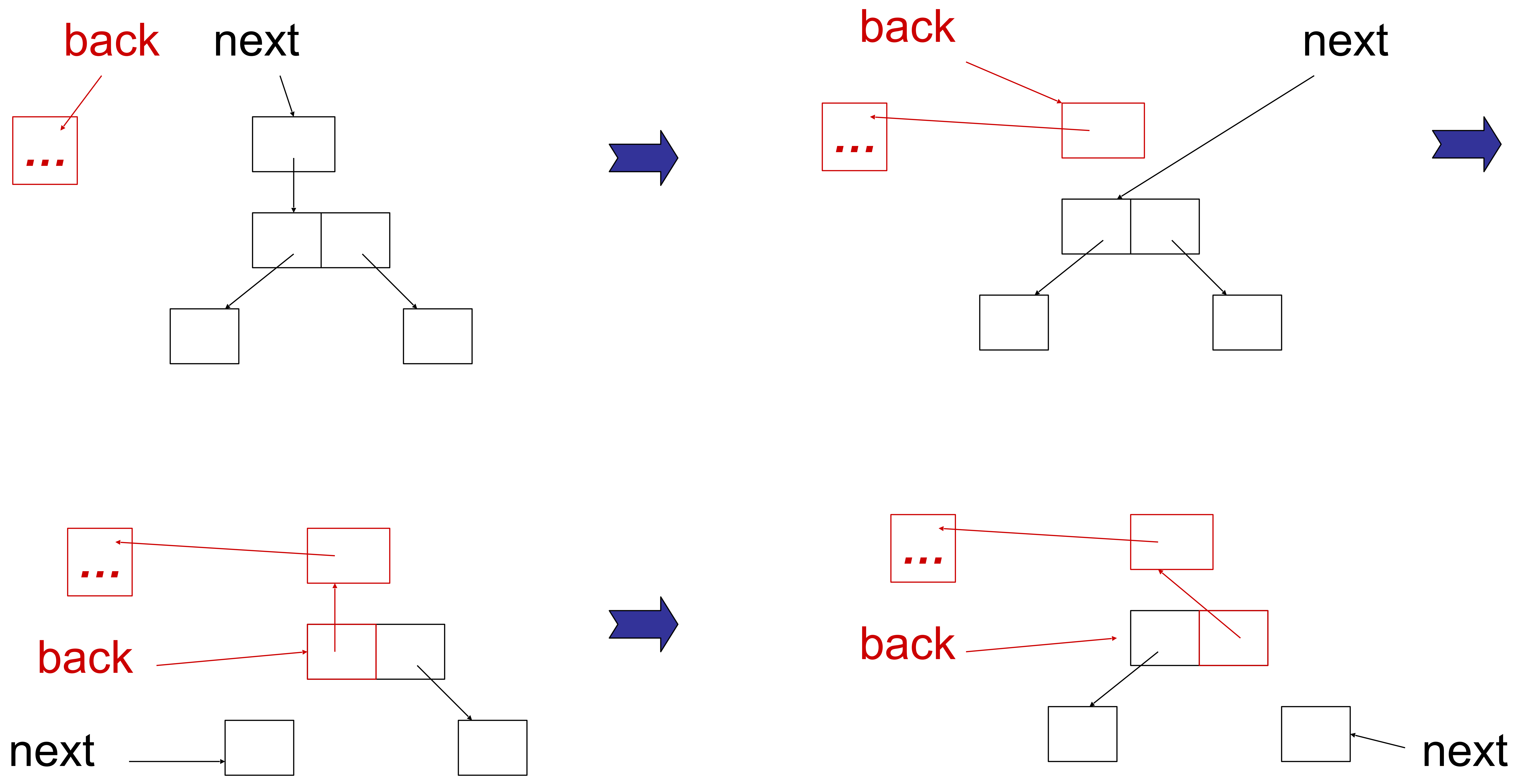
DSW Algorithm



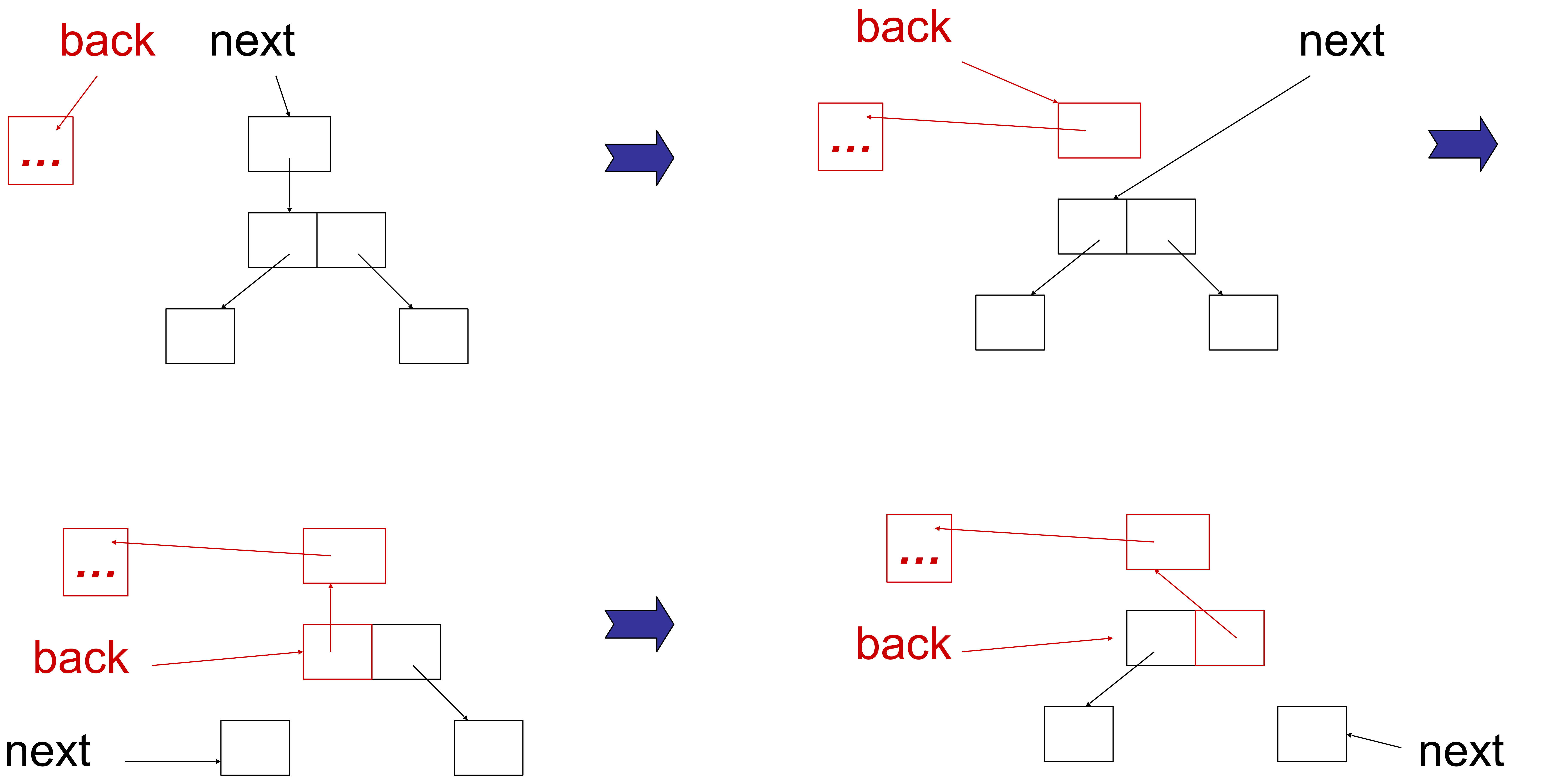
DSW Algorithm



DSW Algorithm



DSW Algorithm



• extra bits needed to keep track of which record fields we have processed so far

DSW Setup

- Extra space required for sweep:
  - 1 bit/record to keep track of whether the record has been seen (the "mark bit")
  - f log 2 bits/record where f is the number of fields in the record to keep track of how many fields have been processed
    - assume a field of each record x: x.done
- Functions:
  - mark x = sets x's mark bit
  - marked x = true if x's mark bit is set
  - pointer x = true if x is a pointer
  - fields x = returns number of fields in the record x

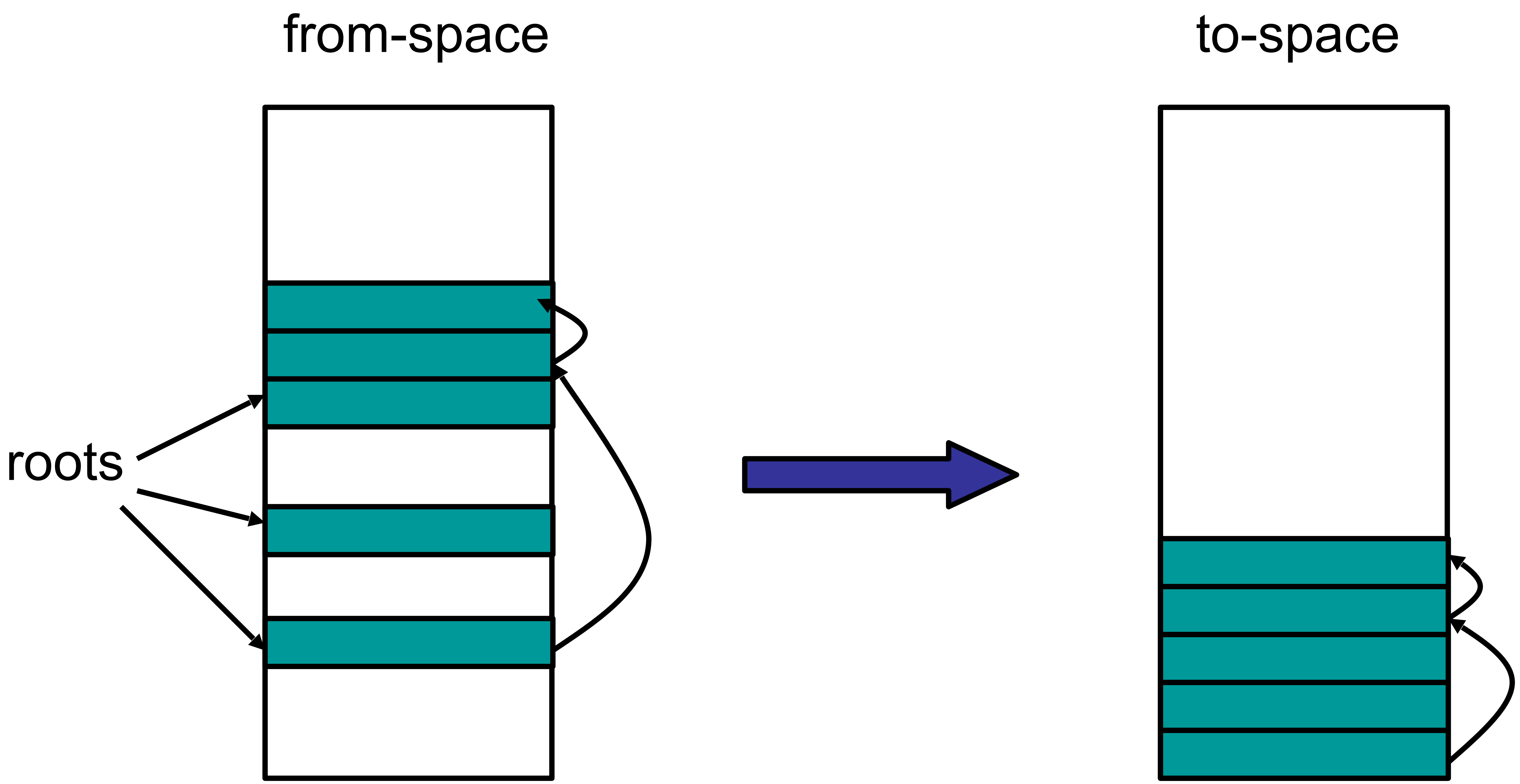
More Mark-Sweep

- Mark-sweep collectors can benefit from the tricks used to implement malloc/free efficiently
  - multiple free lists, one size of block/list
- Mark-sweep can suffer from fragmentation
  - blocks not copied and compacted like in copying collection
- Mark-sweep maximum space usage is the total heap size
  - but if the ratio of live data to heap size is too large then performance suffers

Copying Collection

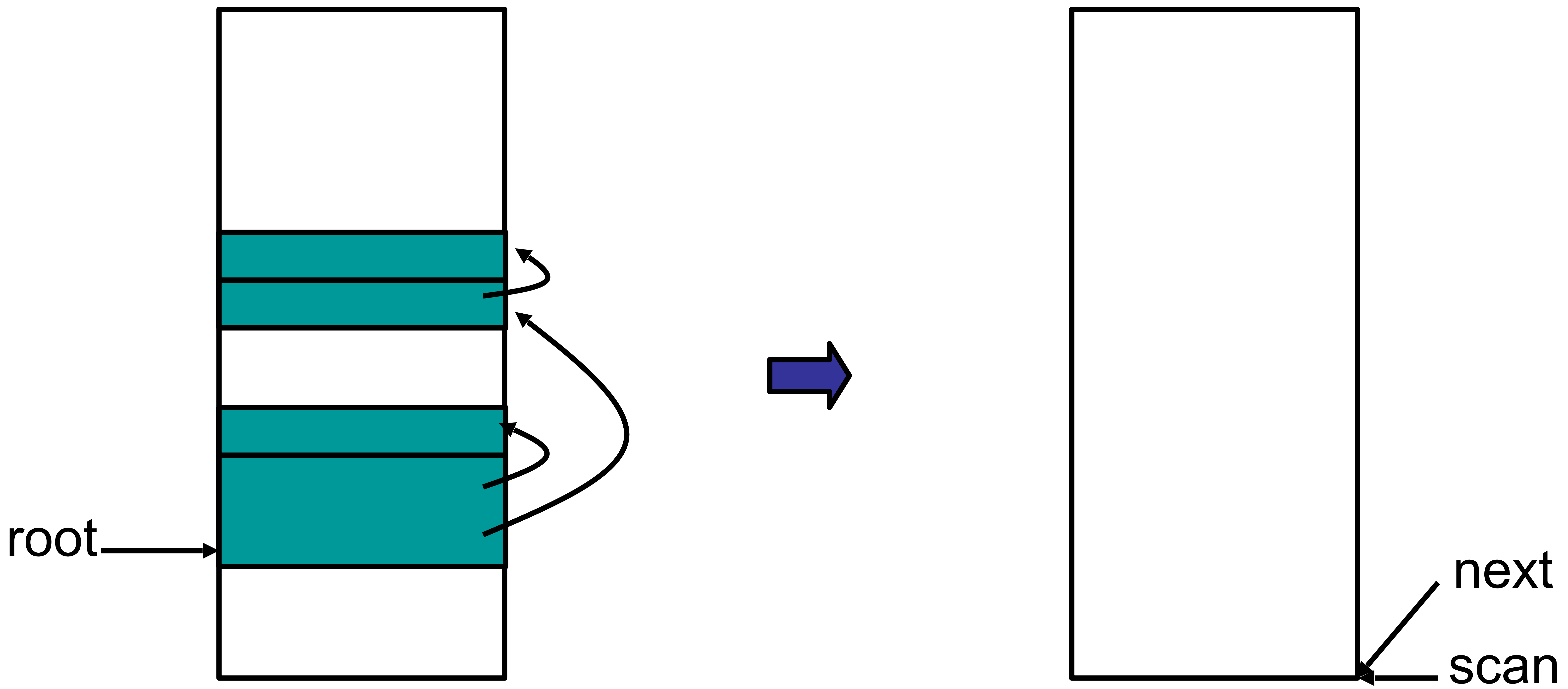
- Basic idea: use 2 heaps
  - One used by program
  - The other unused until GC time
- GC:
  - Start at the roots & traverse the reachable data
  - Copy reachable data from the active heap (from-space) to the other heap (to-space)
  - Dead objects are left behind in from space
  - Heaps switch roles

Copying Collection



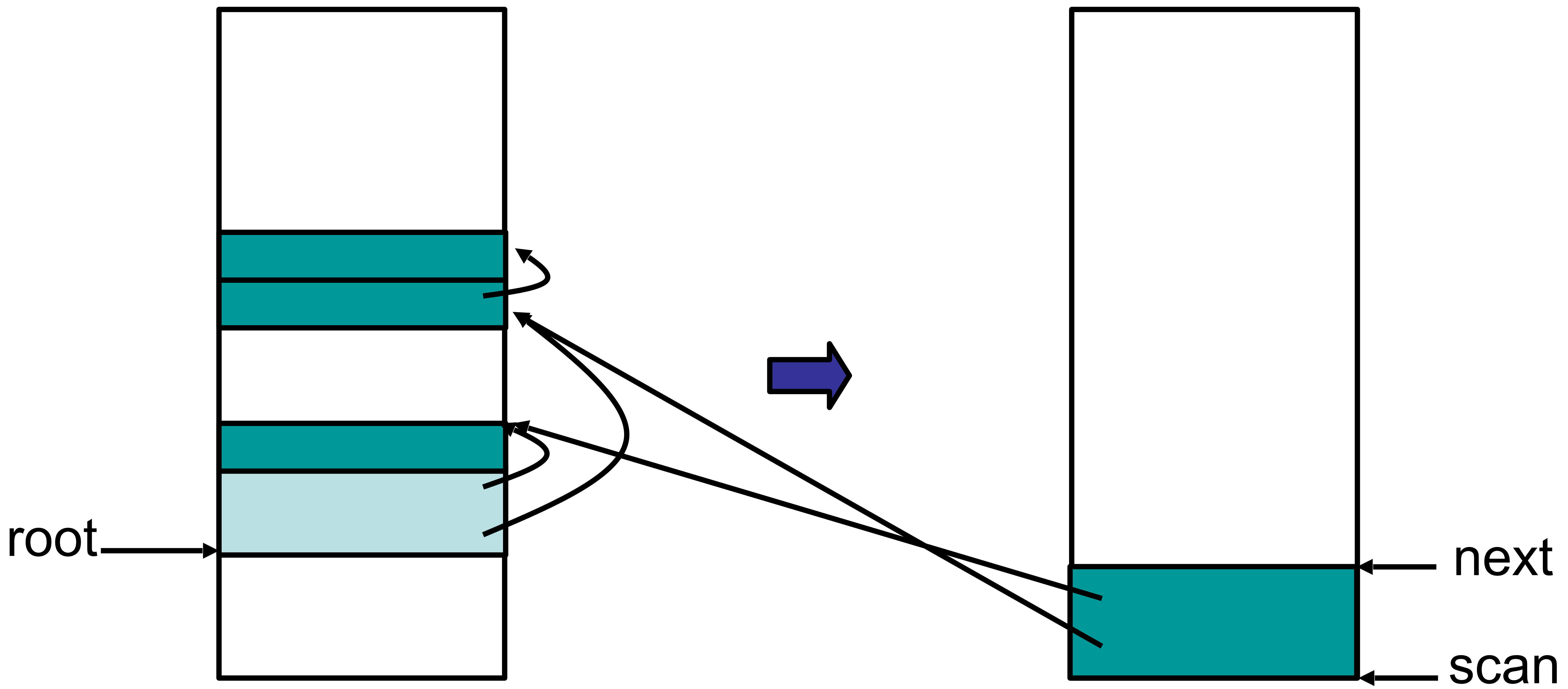
Copying GC

- Cheny's algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space



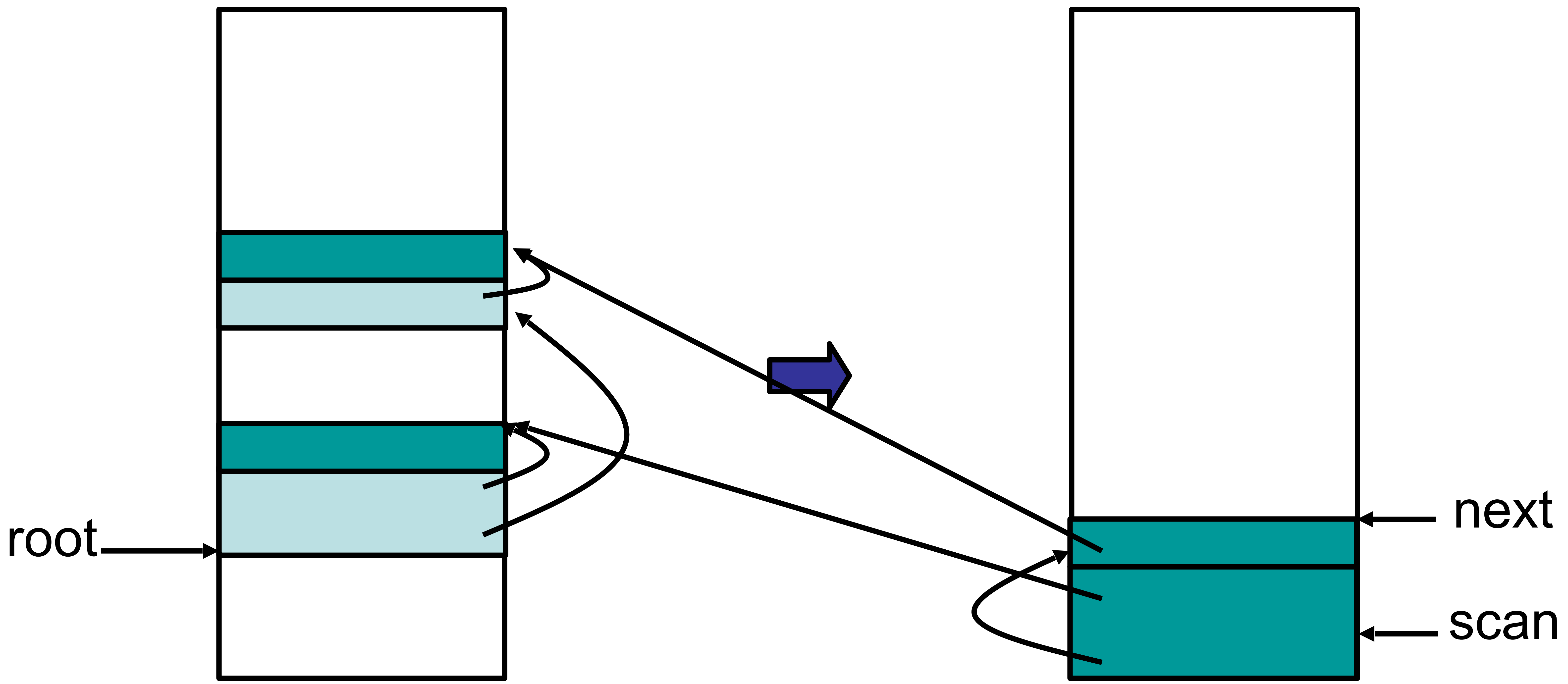
Copying GC

- Cheny's algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space



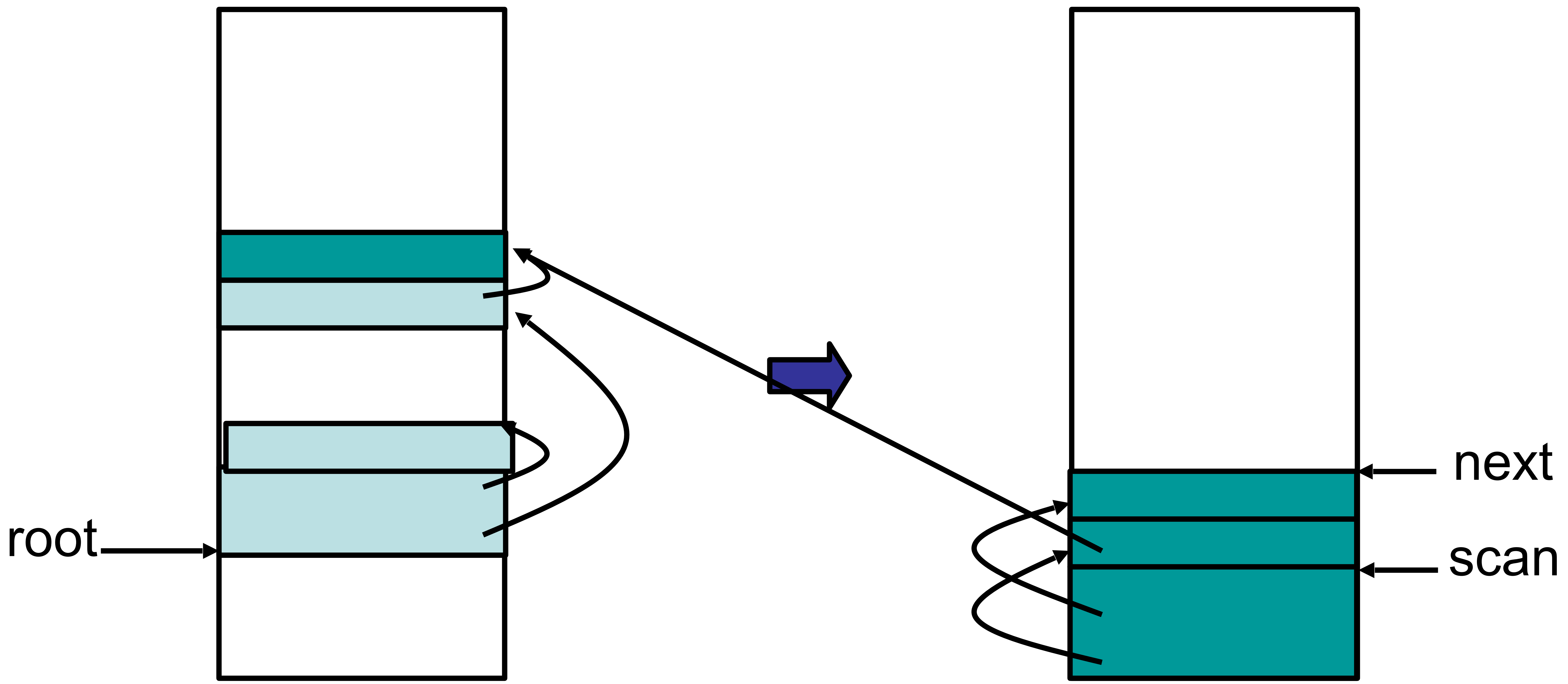
Copying GC

- Cheny's algorithm for copying collection
  – Traverse data breadth first, copying objects from from-space to to-space



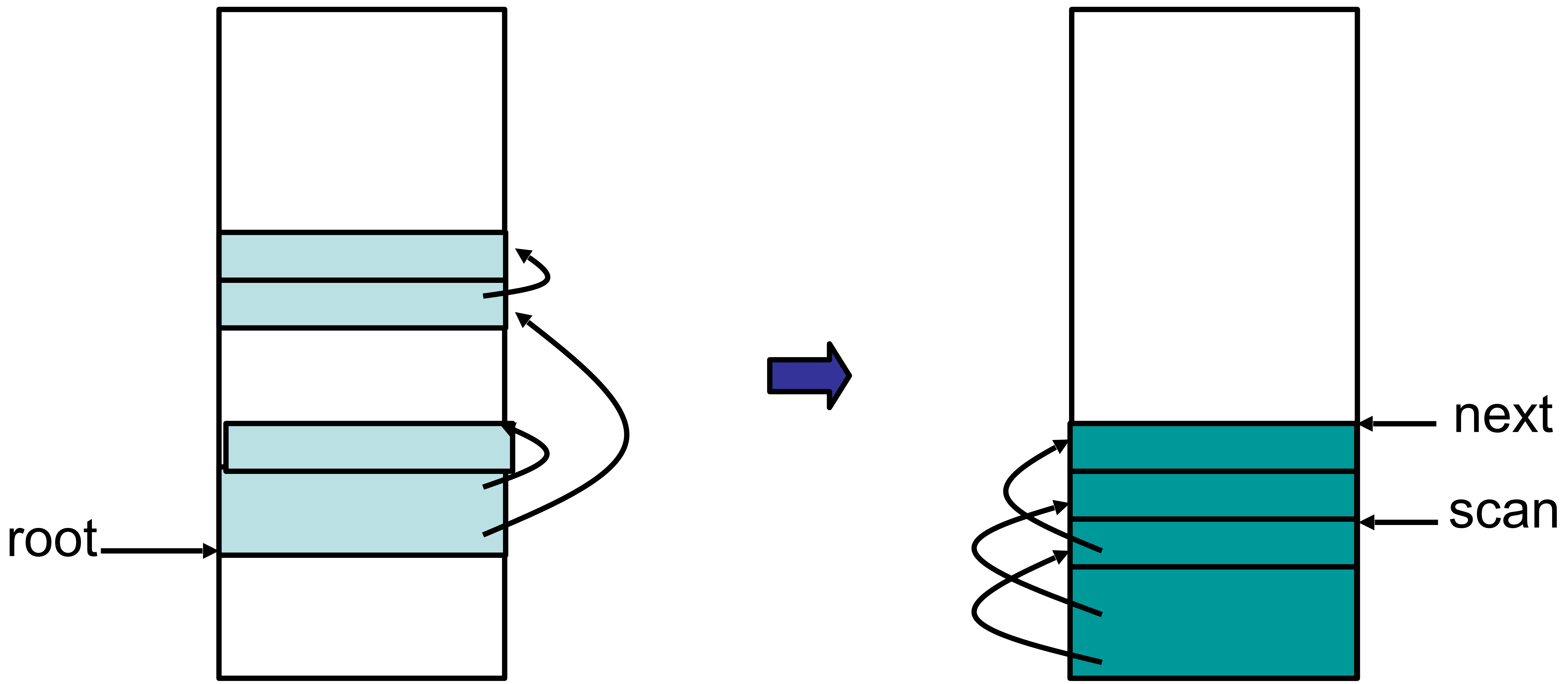
Copying GC

- Cheny's algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space



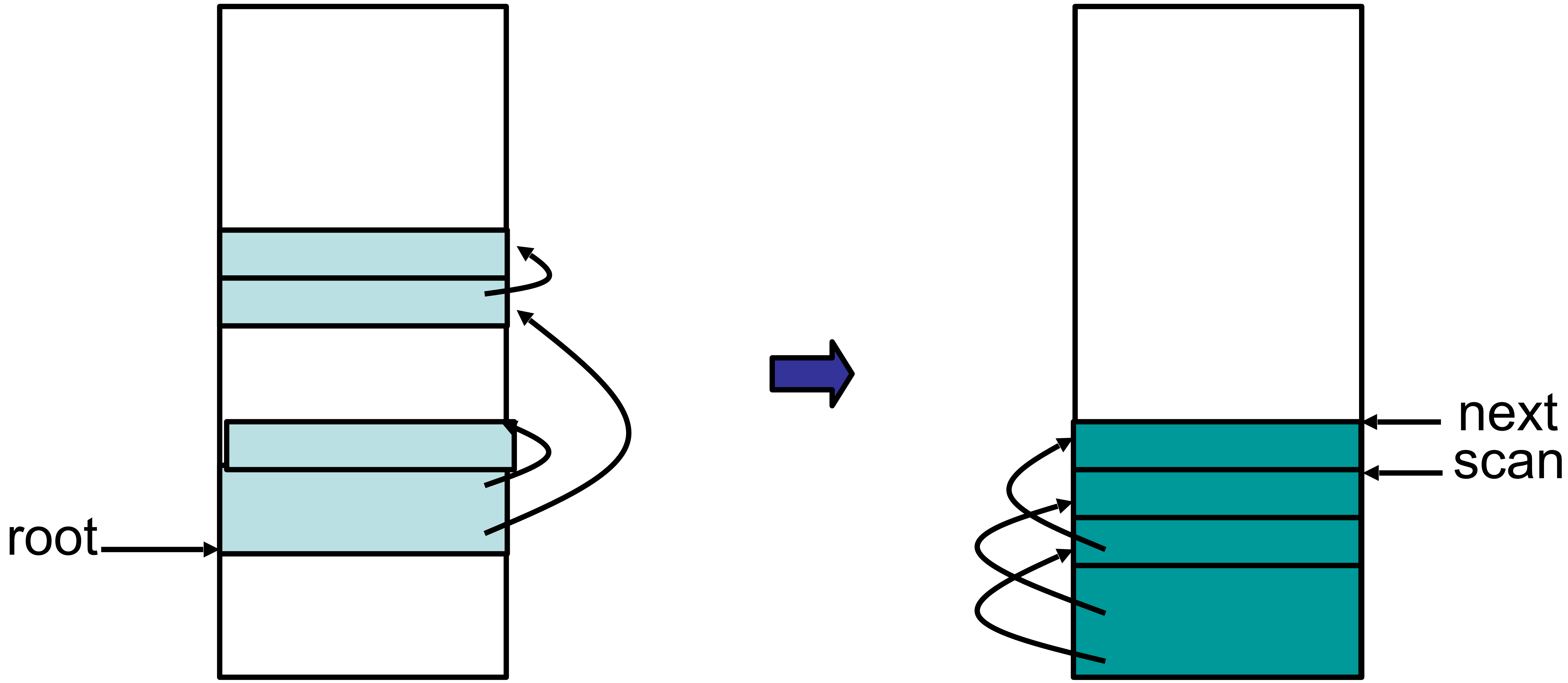
Copying GC

- Cheny's algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space



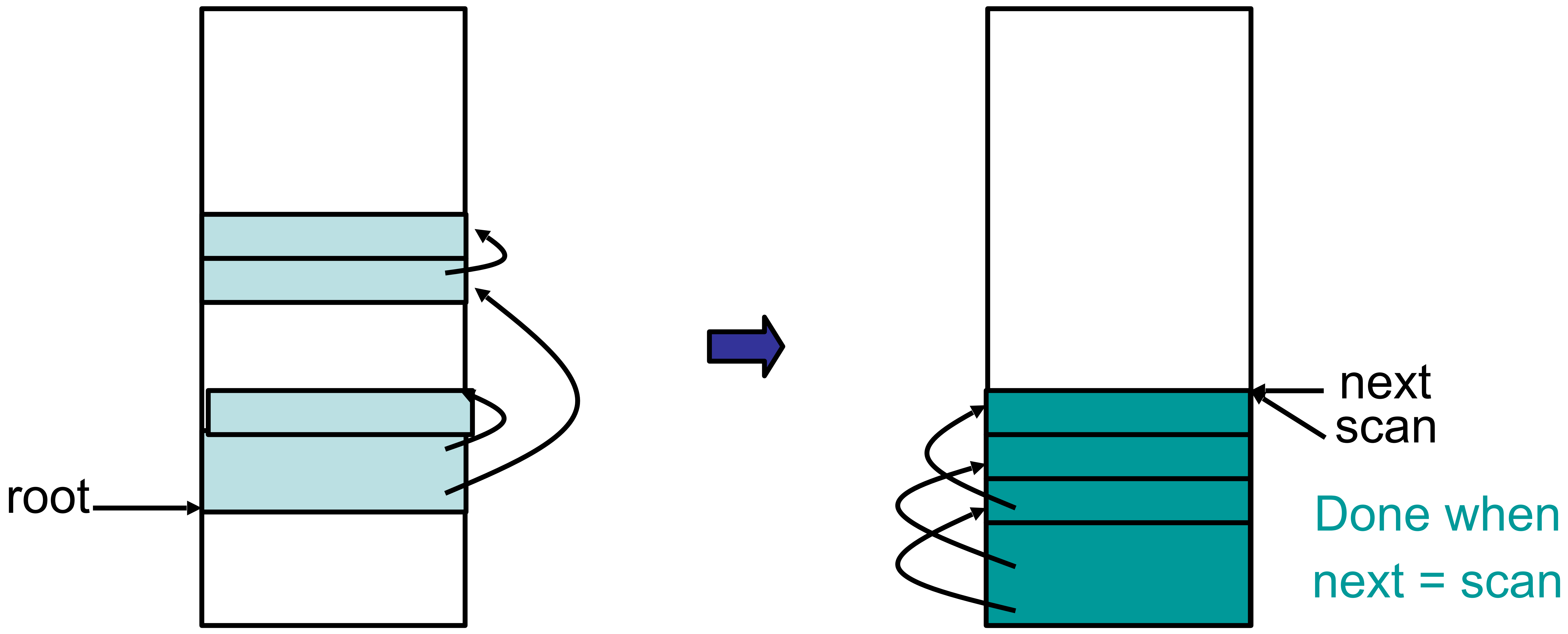
Copying GC

- Cheny's algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space



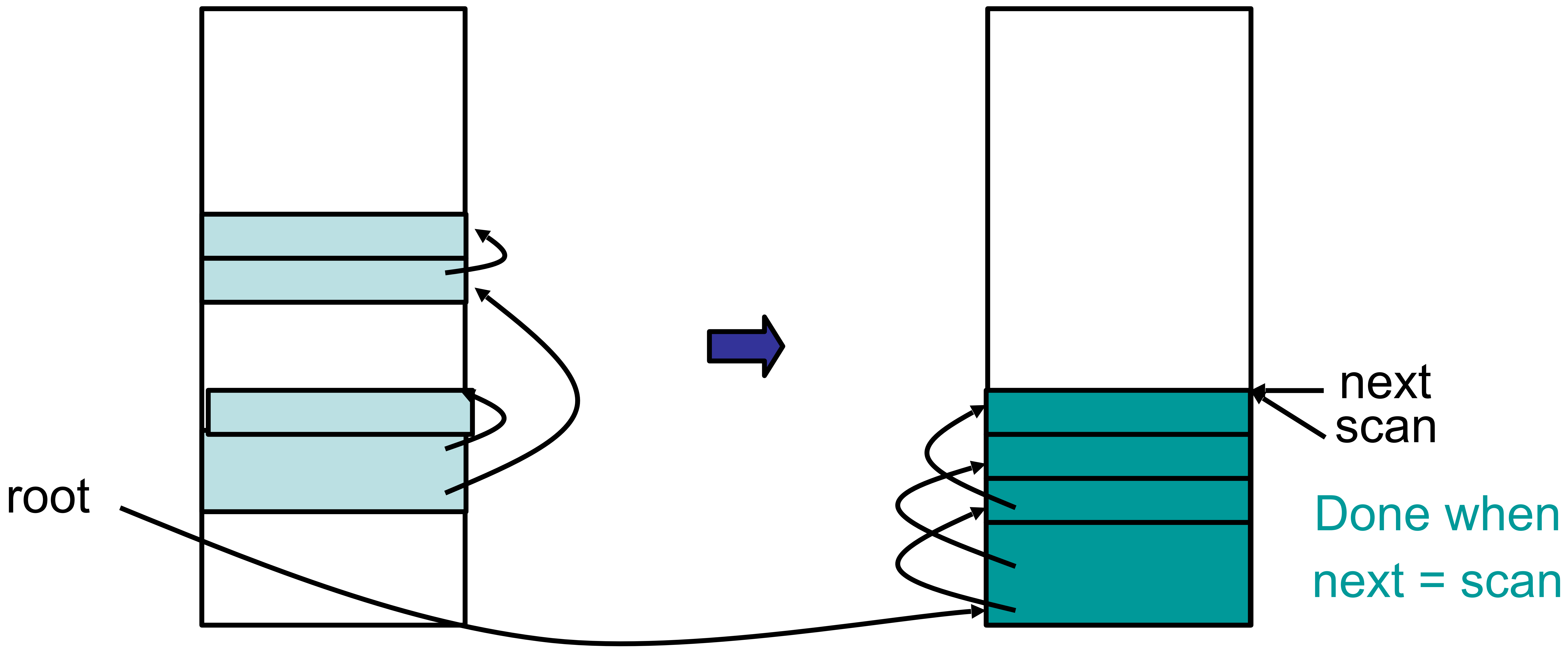
Copying GC

- Cheny's algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space



Copying GC

- Cheny’s algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space



Copying GC

- Pros
  - Simple & collects cycles
  - Run-time proportional to # live objects
  - Automatic compaction eliminates fragmentation
  - Fast allocation: pointer increment by object size
- Cons
  - Precise type information required (pointer or not)
    - Tag bits take extra space; normally use header word
  - Twice as much memory used as program requires
    - Usually, we anticipate live data will only be a small fragment of store
    - Allocate until 70% full
    - From-space = 70% heap; to-space = 30%
  - Long GC pauses = bad for interactive, real-time apps

Generational GC

- Empirical observation: if an object has been reachable for a long time, it is likely to remain so
- Empirical observation: in many languages (especially functional languages), most objects died young
- Conclusion: we save work by scanning the young objects frequently and the old objects infrequently

Generational GC

- Assign objects to different generations G0, G1,…
  - G0 contains young objects, most likely to be garbage
  - G0 scanned more often than G1
  - Common case is two generations (new, tenured)
  - Roots for GC of G0 include all objects in G1 in addition to stack, registers

Generational GC

- How do we avoid scanning tenured objects?
  - Observation: old objects rarely point to new objects
    - Normally, object is created and when it initialized it will point to older objects, not newer ones
    - Only happens if old object modified well after it is created
    - In functional languages that use mutation infrequently, pointers from old to new are very uncommon
  - Compiler inserts extra code on object field pointer write to catch modifications to old objects
  - Remembered set is used to keep track of objects that point into younger generation. Remembered set included in set of roots for scanning.

Generational GC

- Other issues
  - When do we promote objects from young generation to old generation
    - Usually after an object survives a collection, it will be promoted
  - How big should the generations be?
    - Appel says each should be exponentially larger than the last
  - When do we collect the old generation?
    - After several minor collections, we do a major collection

Generational GC

- Other issues
  - Sometimes different GC algorithms are used for the new and older generations.
    - Why? Because the have different characteristics
  - Copying collection for the new
    - Less than 10% of the new data is usually live
    - Copying collection cost is proportional to the live data
  - Mark-sweep for the old

Conservative Collection

- Even languages like C can benefit from GC
  - Boehm-Weiser-Demers conservative GC uses heuristics to determine which objects are pointers and which are integers without any language support
    - last 2 bits are non-zero => can't be a pointer
    - integer is not in allocated heap range => can't be a pointer
    - mark phase traverses all possible pointers
    - conservative because it may retain data that isn't reachable
      - thinks an integer is actually a pointer
      - since it does not copy objects (thereby changing pointer values), mistaking integers for pointers does not hurt
    - all gc is conservative anyway so this is almost never an issue (despite what people say)
    - sound if your program doesn't manufacture pointers from integers by, say, using xor (using normal pointer arithmetic is fine)

Compiler Interface

- The interface to the garbage collector involves two main parts
  - allocation code
    - languages can allocate up to approx 1 word/7 instructions
    - allocation code must be blazingly fast!
    - should be inlined and optimized to avoid call-return overhead
  - gc code
    - to call gc code, the program must identify the roots
    - to traverse data, heap layout must be specified somehow

Allocation Code

Assume size of record allocated is N:

1. Call alloc code
2. Test next + N < limit   (call gc on failure)
3. Move next into function result
4. Clear M[next], ..., M[next + N – 1]
5. next = next + N
6. Return from alloc code

Allocation Code

Assume size of record allocated is N:

1. Call alloc function
2. Test next + N < limit   (call gc on failure)
3. Move next into function result
4. Clear M[next], ..., M[next + N – 1]
5. next = next + N
6. Return from alloc function
7. Move result into computationally useful place
8. Store useful values into M[next],....,M[next + N - 1]

useful computation
not alloc overhead

Allocation Code

Assume size of record allocated is N:

1. Call alloc function
2. Test next + N < limit (call gc on failure)
3. Move next into function result
4. Clear M[next], ..., M[next + N – 1]
5. next = next + N
6. Return from alloc function
7. Move result into computationally useful place
8. Store useful values into M[next],....,M[next + N - 1]

inline alloc code

Allocation Code

Assume size of record allocated is N:

1. Call alloc function
2. Test next + N < limit (call gc on failure)
3. Move next into computationally useful place
4. Clear M[next], ..., M[next + N – 1]
5. next = next + N
6. Return from alloc function
7. Move next into computationally useful place
8. Store useful values into M[next],....,M[next + N - 1]

combine moves

Allocation Code

Assume size of record allocated is N:

1. Call alloc function
2. Test next + N < limit (call gc on failure)
3. Move next into computationally useful place
4. Clear M[next], ..., M[next + N – 1]
5. next = next + N
6. Return from alloc function
7. Move next into computationally useful place
8. Store useful values into M[next],....,M[next + N - 1]

eliminate useless store

Allocation Code

Assume size of record allocated is N:

1. Call alloc function
2. Test next + N < limit   (call gc on failure)
3. Move next into computationally useful place
4. Clear M[next], ..., M[next + N – 1]
5. next = next + N
6. Return from alloc function
7. Move next into computationally useful place
8. Store useful values into M[next],....,M[next + N - 1]

total overhead for allocation on the order of 3 instructions/alloc

Calling GC code

- To call the GC, program must:
  - identify the **roots**:
    - a GC-point, is an control-flow point where the garbage collector may be called
      - allocation point; function call
    - for any GC-point, compiler generates a pointer map that says which registers, stack locations in the current frame contain pointers
    - a global table maps GC-points (code addresses) to pointer maps
    - when program calls the GC, to find all roots:
      - GC scans down stack, one activation record at a time, looking up the current pointer map for that record

Calling GC code

- To call the GC, program must:
  - enable GC to determine data layout of all objects in the heap
    - every record has a header with size and pointer info
  - in object oriented languages like Java:
    - each object has an extra field that points to class definition
    - gc uses class definition to determine object layout including size and pointer info

Summary

- Garbage collectors are a complex and fascinating part of any modern language implementation
- Different collection algs have pros/cons
  - explicit MM, reference counting, copying, generational, mark-sweep
  - all methods, including explicit MM have costs
  - optimizations make allocation fast, GC time, space and latency requirements acceptable
  - additional reading: Appel Chapter 13