Garbage collection

David Walker

CS 320

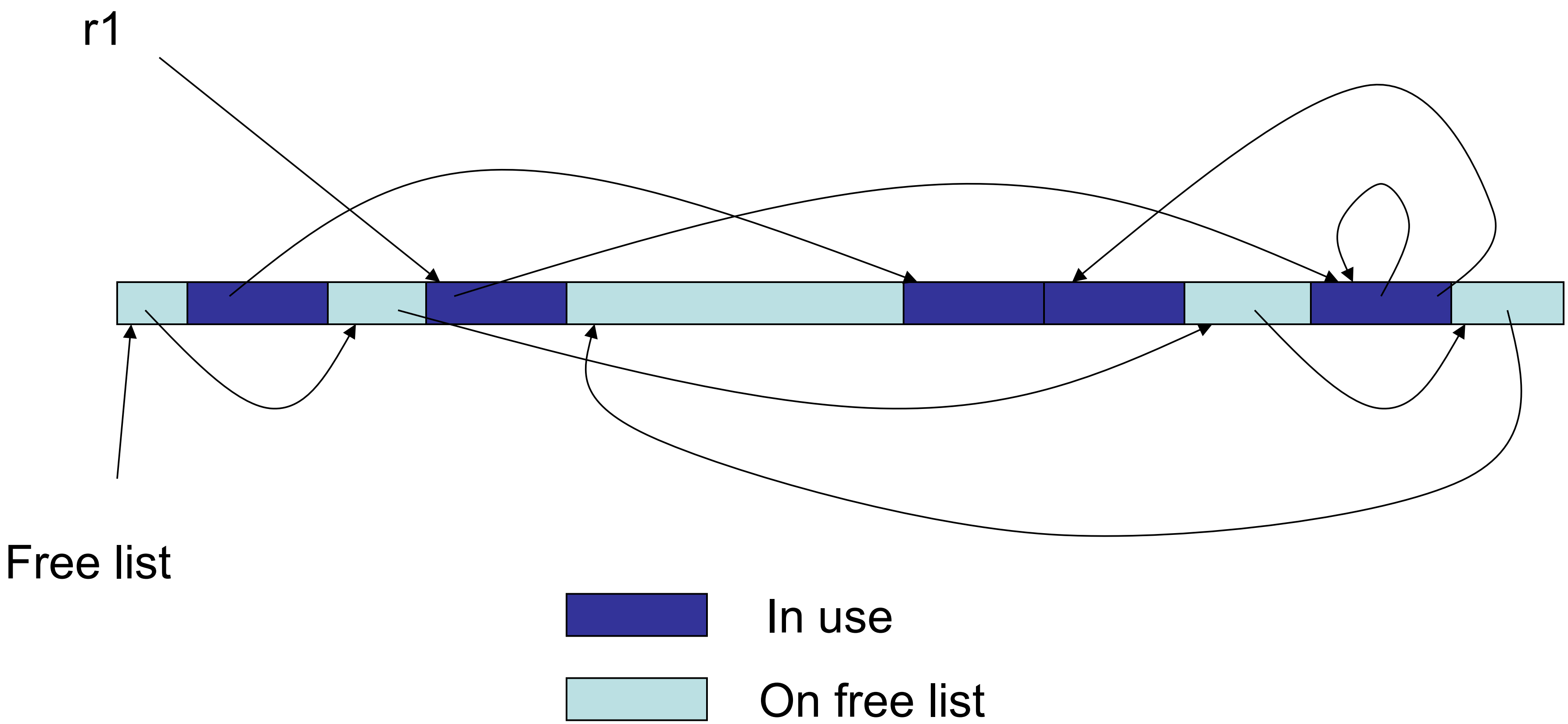
Where are we?

- Last time: A survey of common garbage collection techniques
  - Manual memory management
  - Reference counting (Appel 13.2)
  - Copying collection (Appel 13.3)
  - Generational collection (Appel 13.4)
  - Baker's algorithm (Appel 13.6)
- Today:
  - Mark-sweep collection (Appel 13.1)
  - Conservative collection
  - Compiler interface (13.7)

Mark-sweep

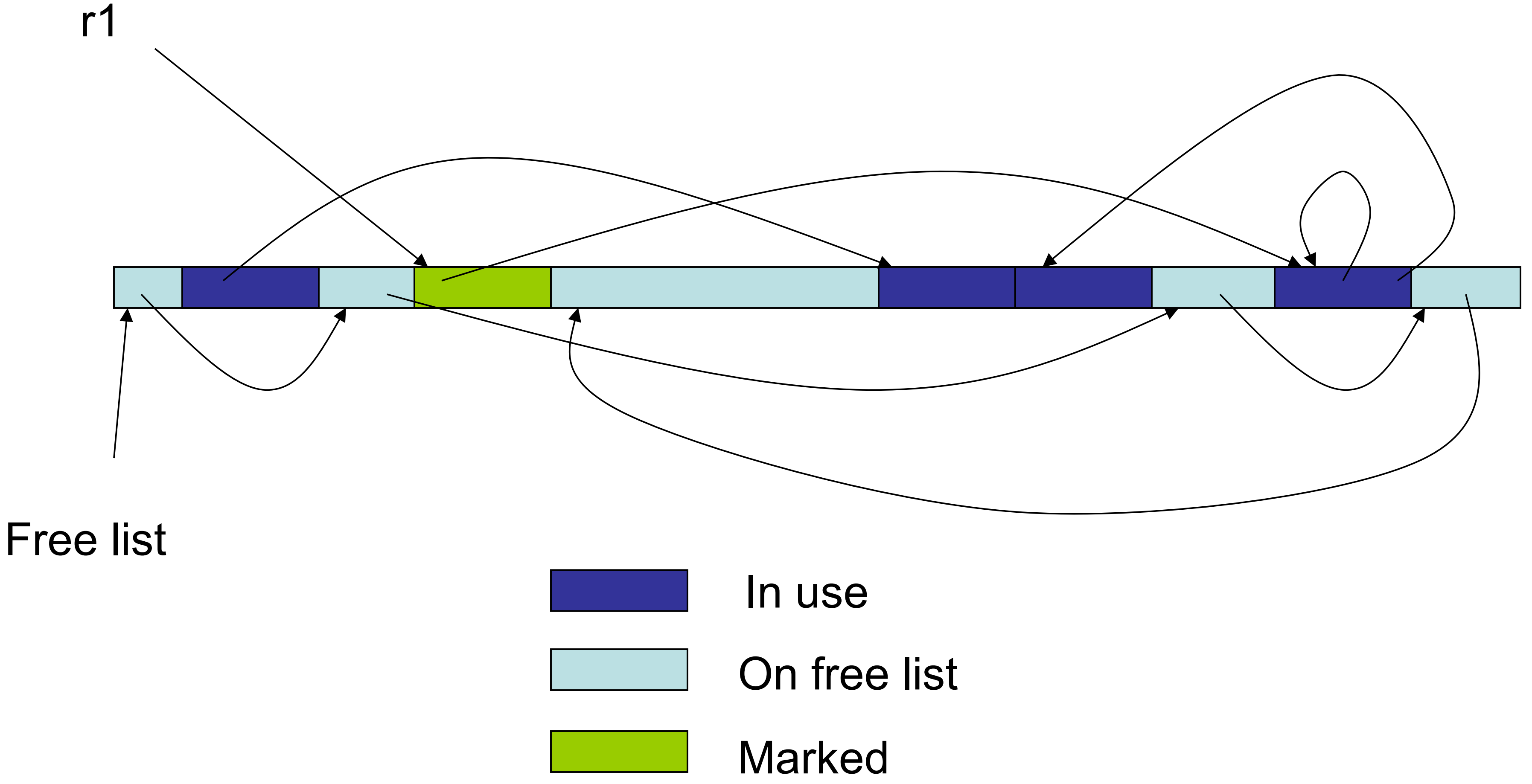
- A two-phase algorithm
  - Mark phase: Depth first traversal of object graph from the roots to mark live data
  - Sweep phase: iterate over entire heap, adding the unmarked data back onto the free list

Example



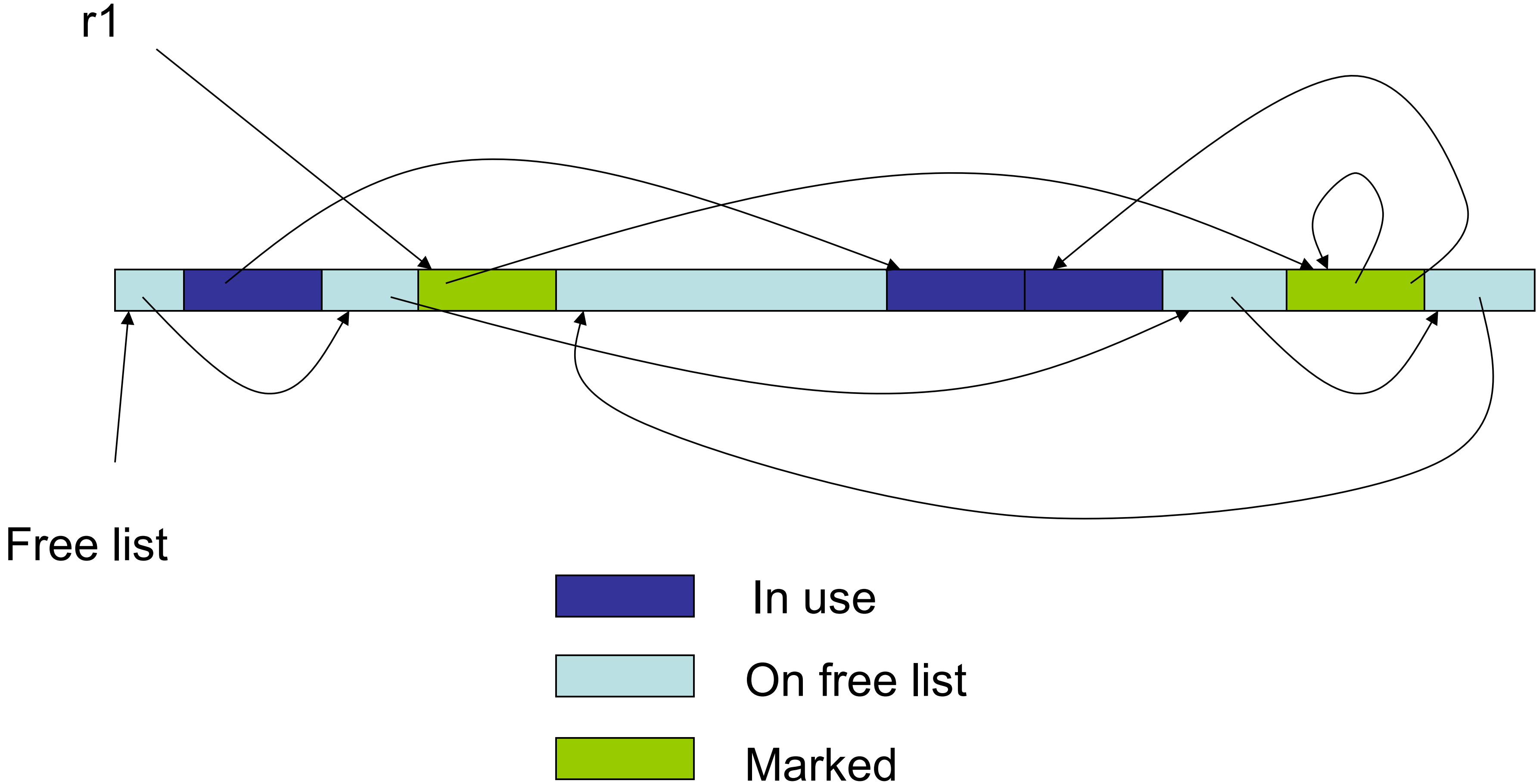
Example

Mark Phase: mark nodes reachable from roots



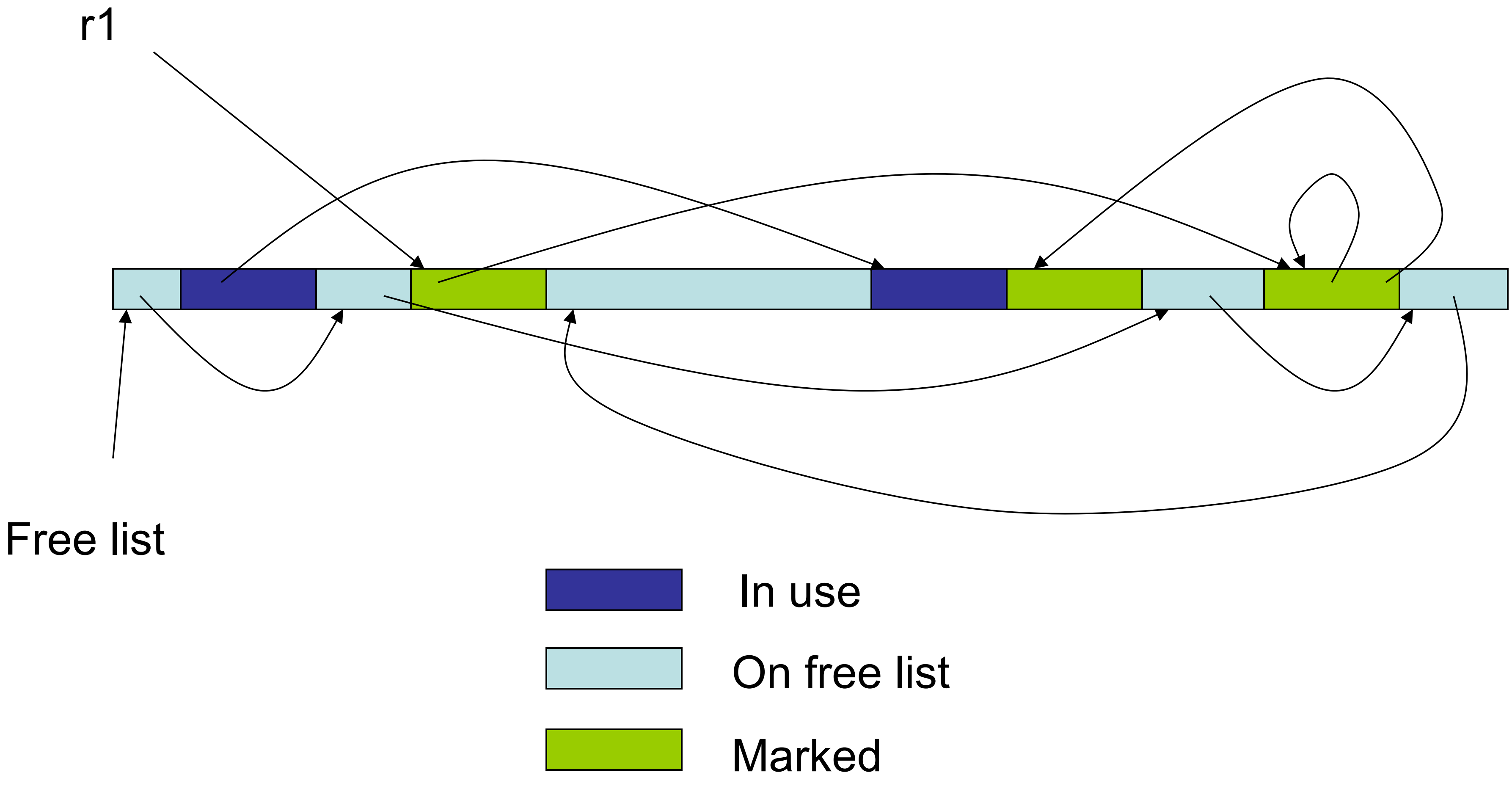
Example

Mark Phase: mark nodes reachable from roots



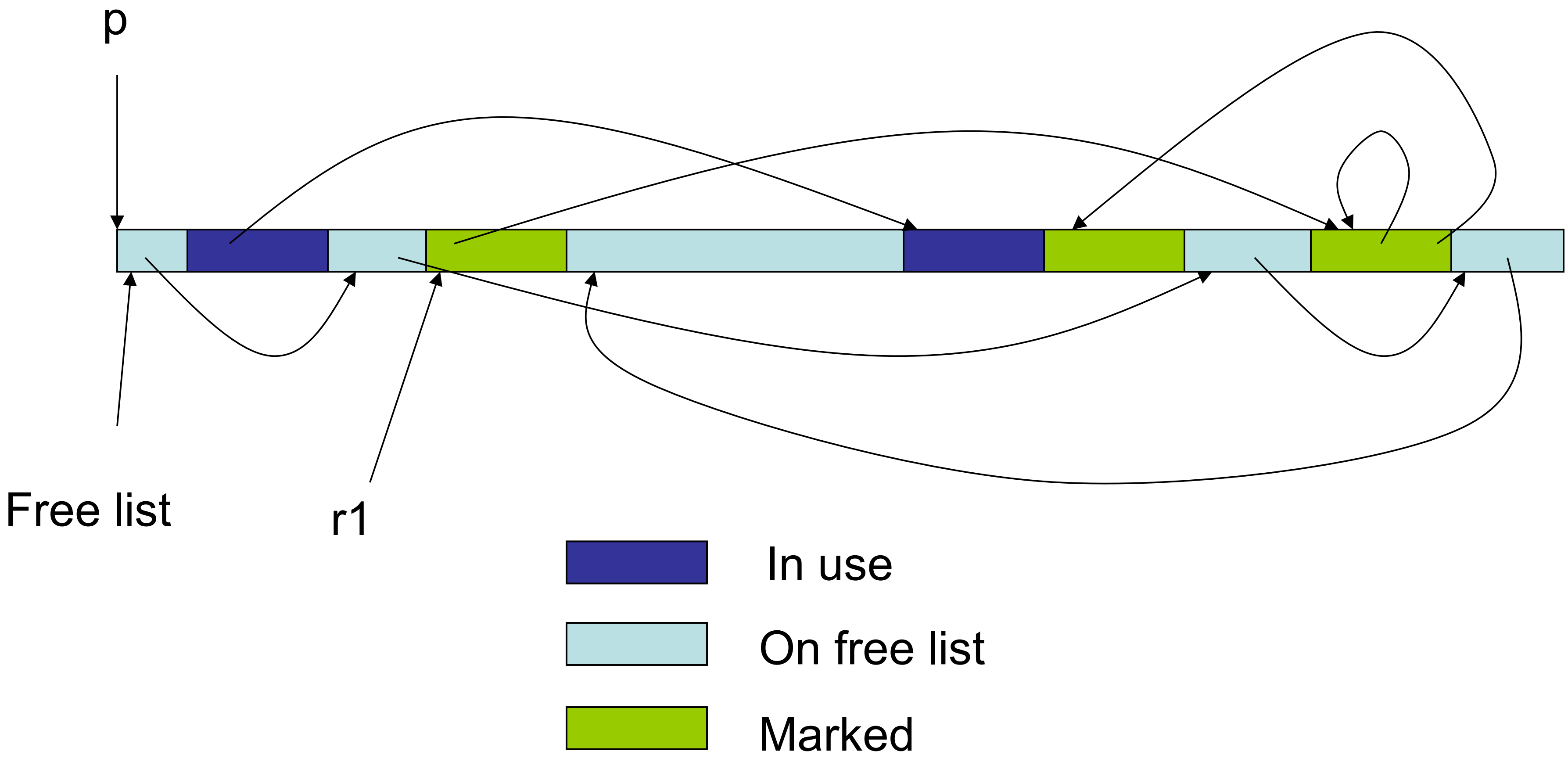
Example

Mark Phase: mark nodes reachable from roots



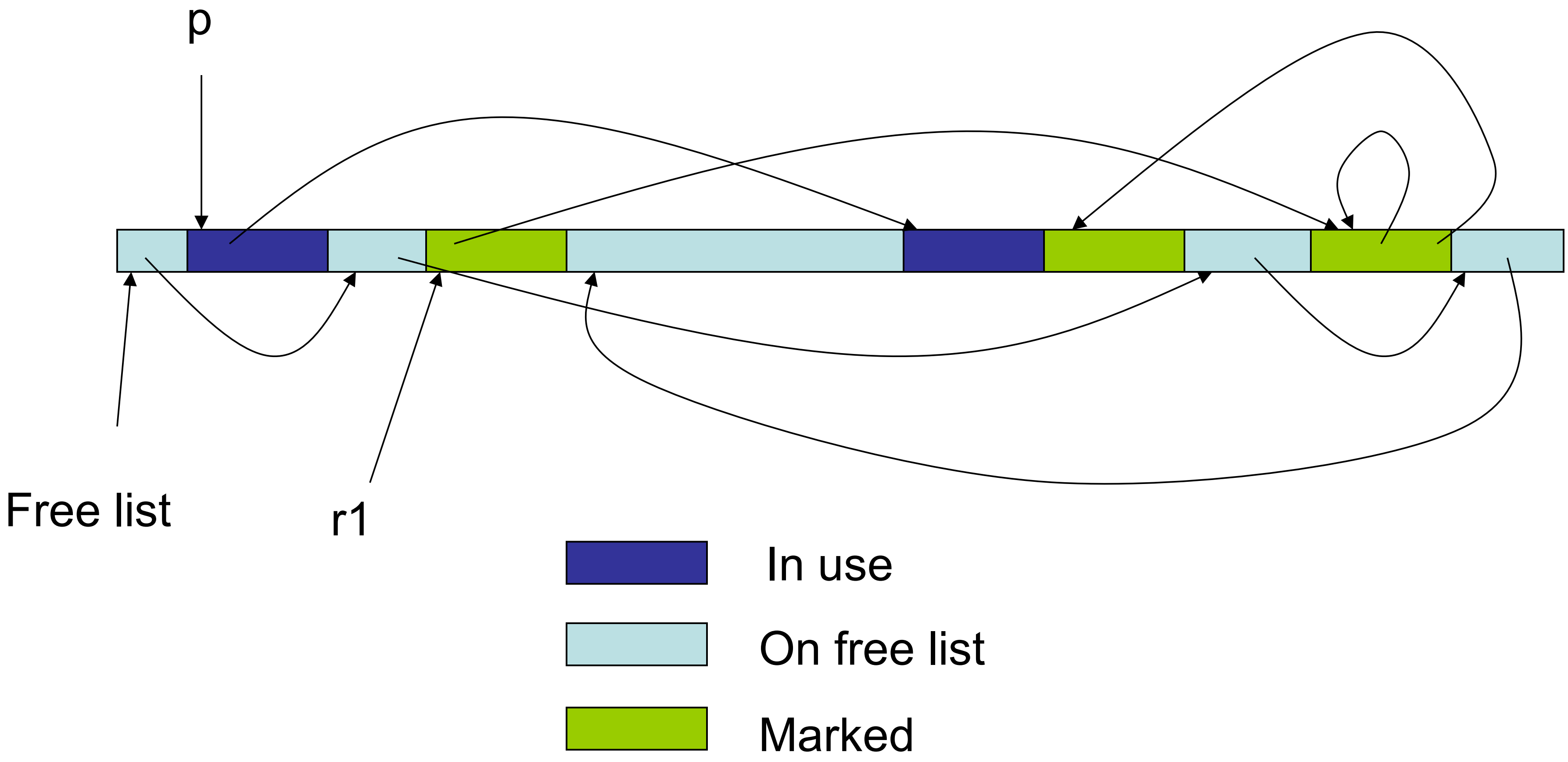
Example

Sweep Phase: set up sweep pointer; begin sweep



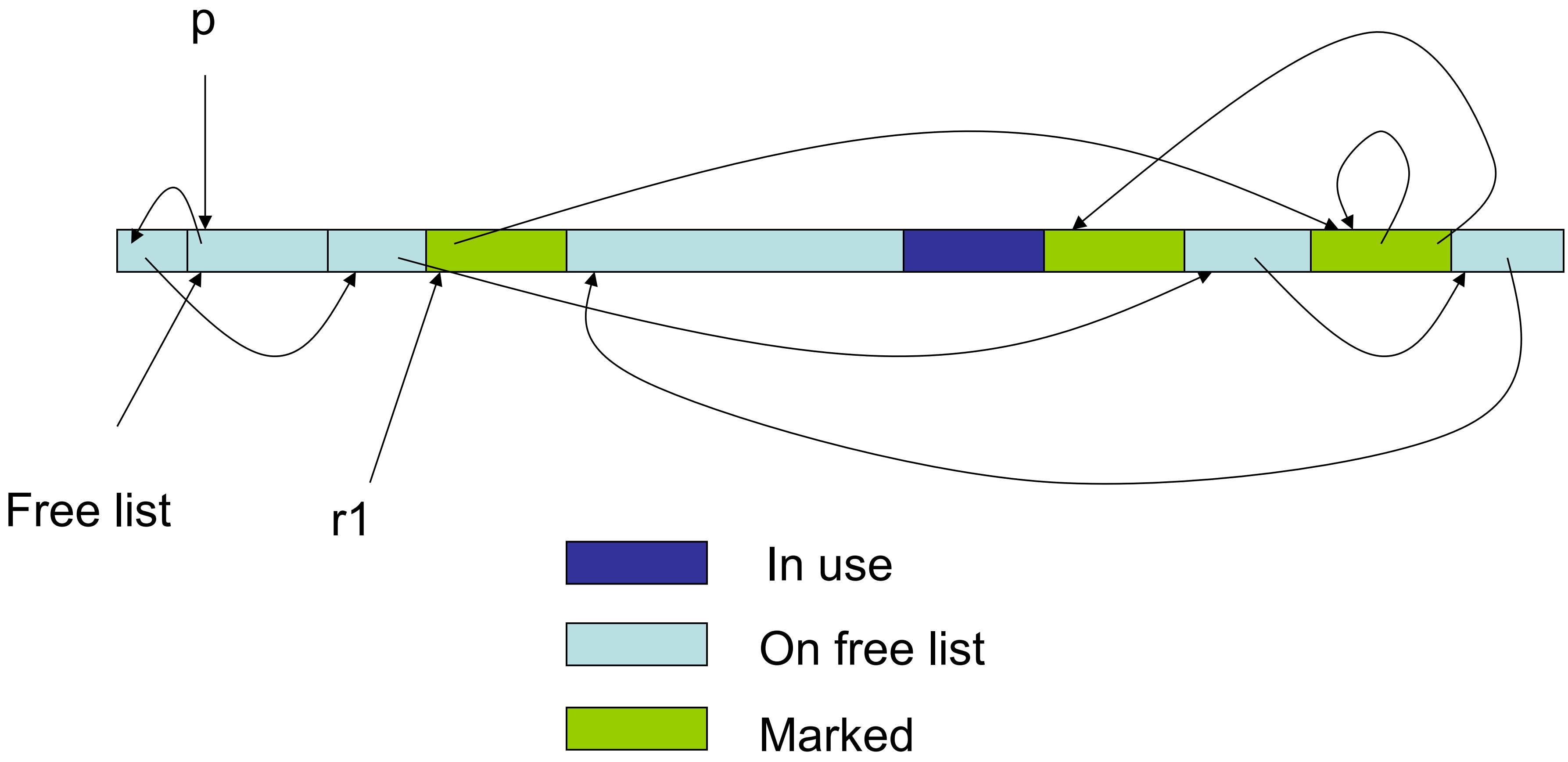
Example

Sweep Phase: add unmarked blocks to free list



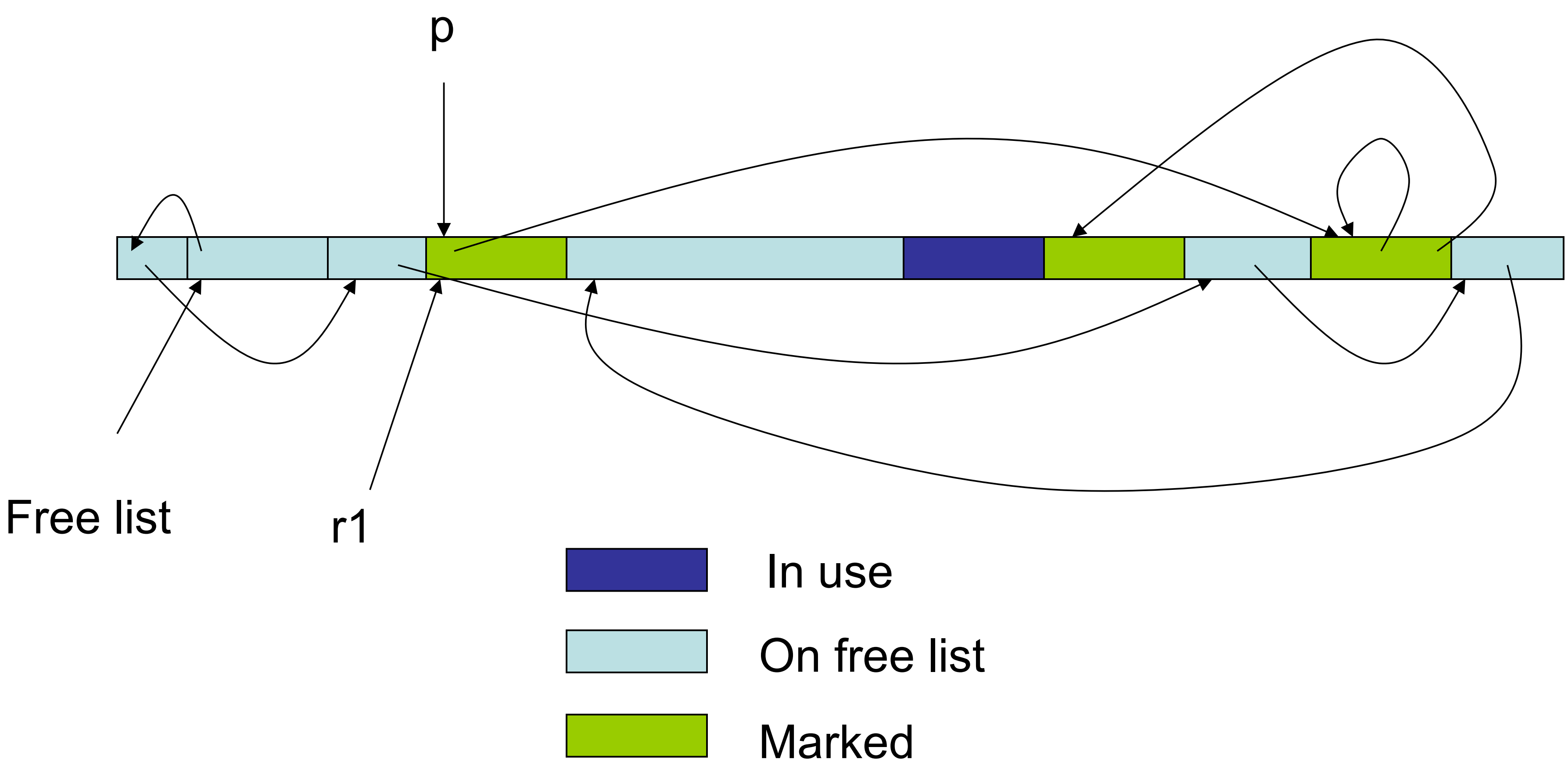
Example

Sweep Phase



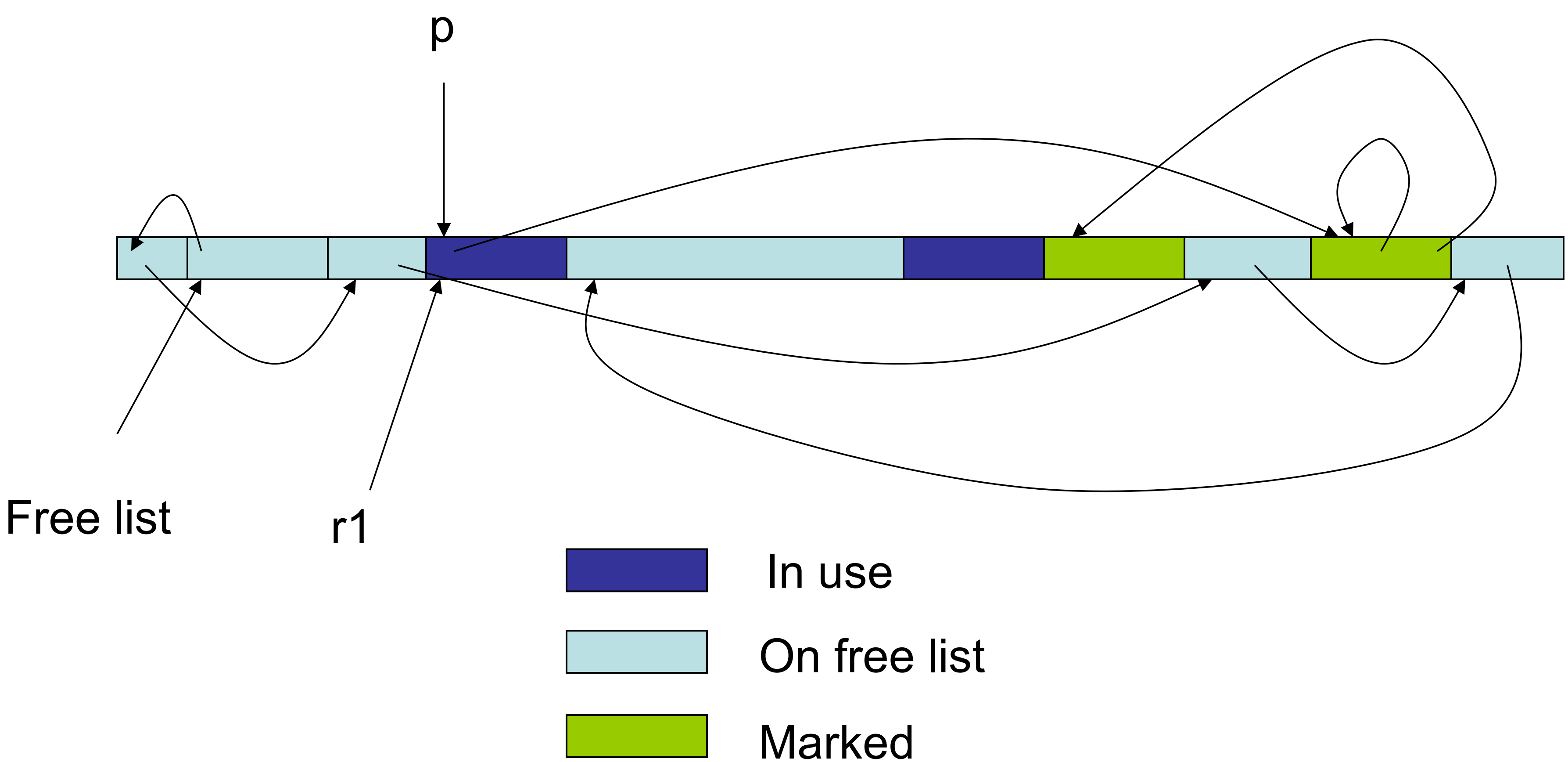
Example

Sweep Phase: retain & unmark marked blocks



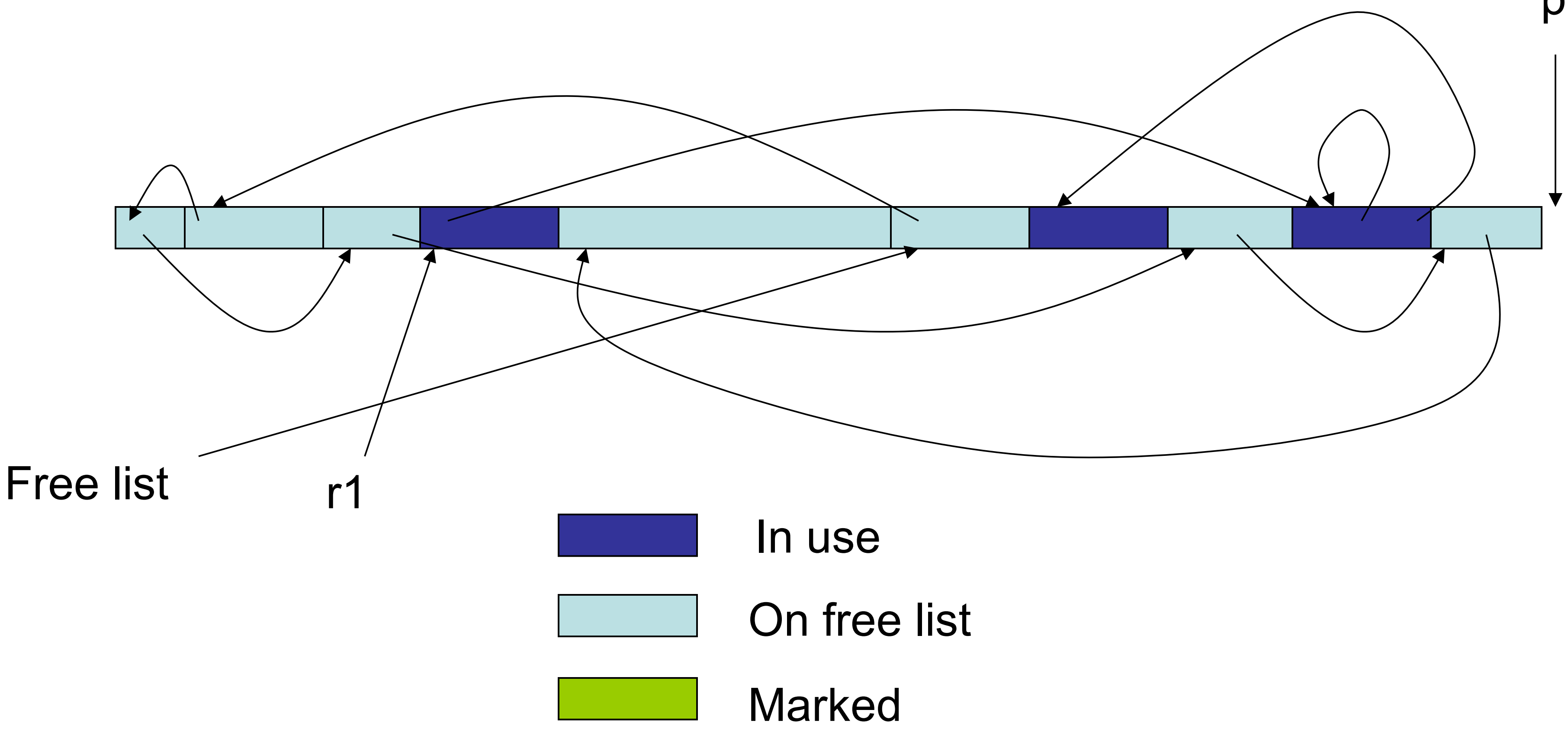
Example

Sweep Phase



Example

Sweep Phase: 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)
- Amortized 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

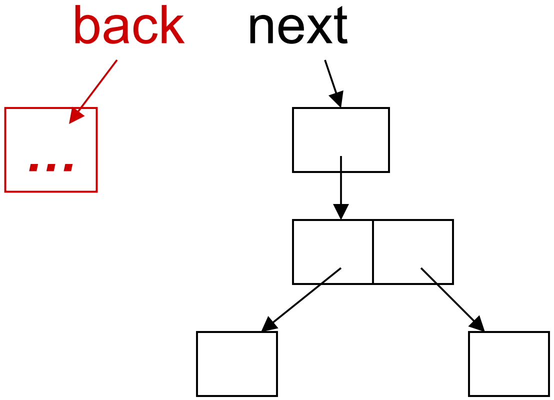
A Hidden Cost

- 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?

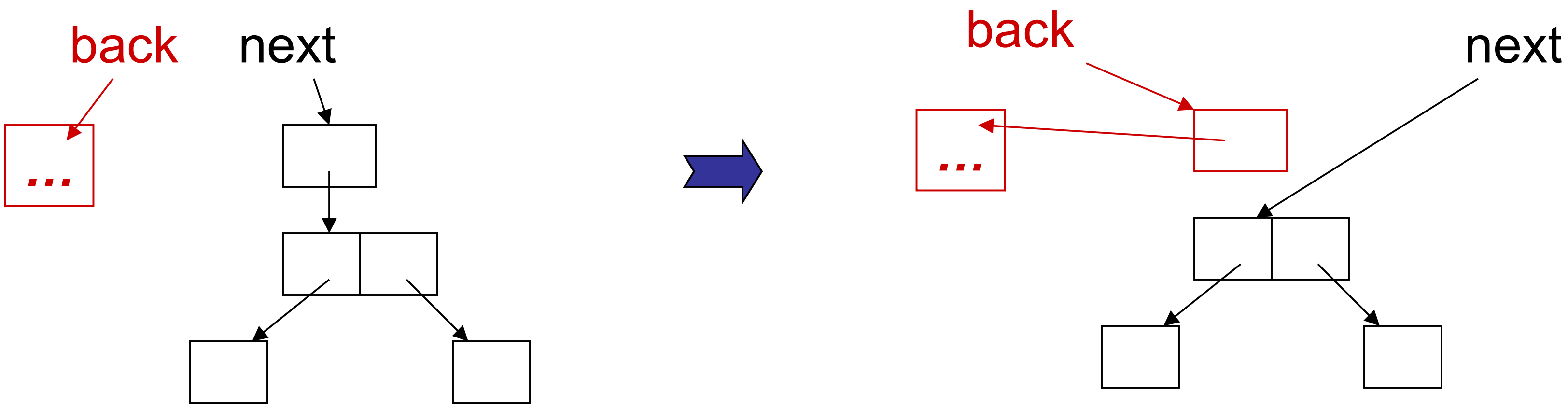
A nifty trick

- 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

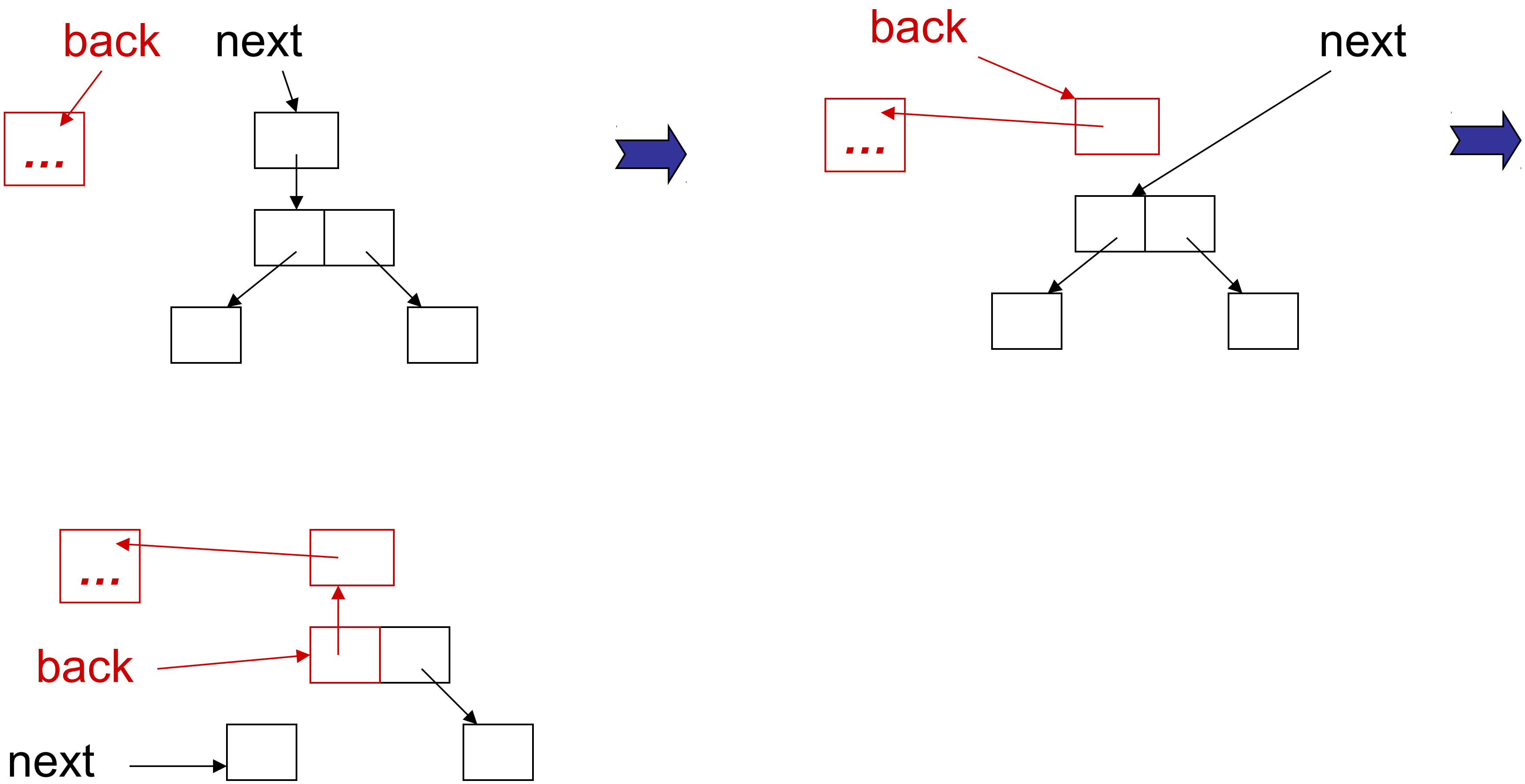
DSW Algorithm



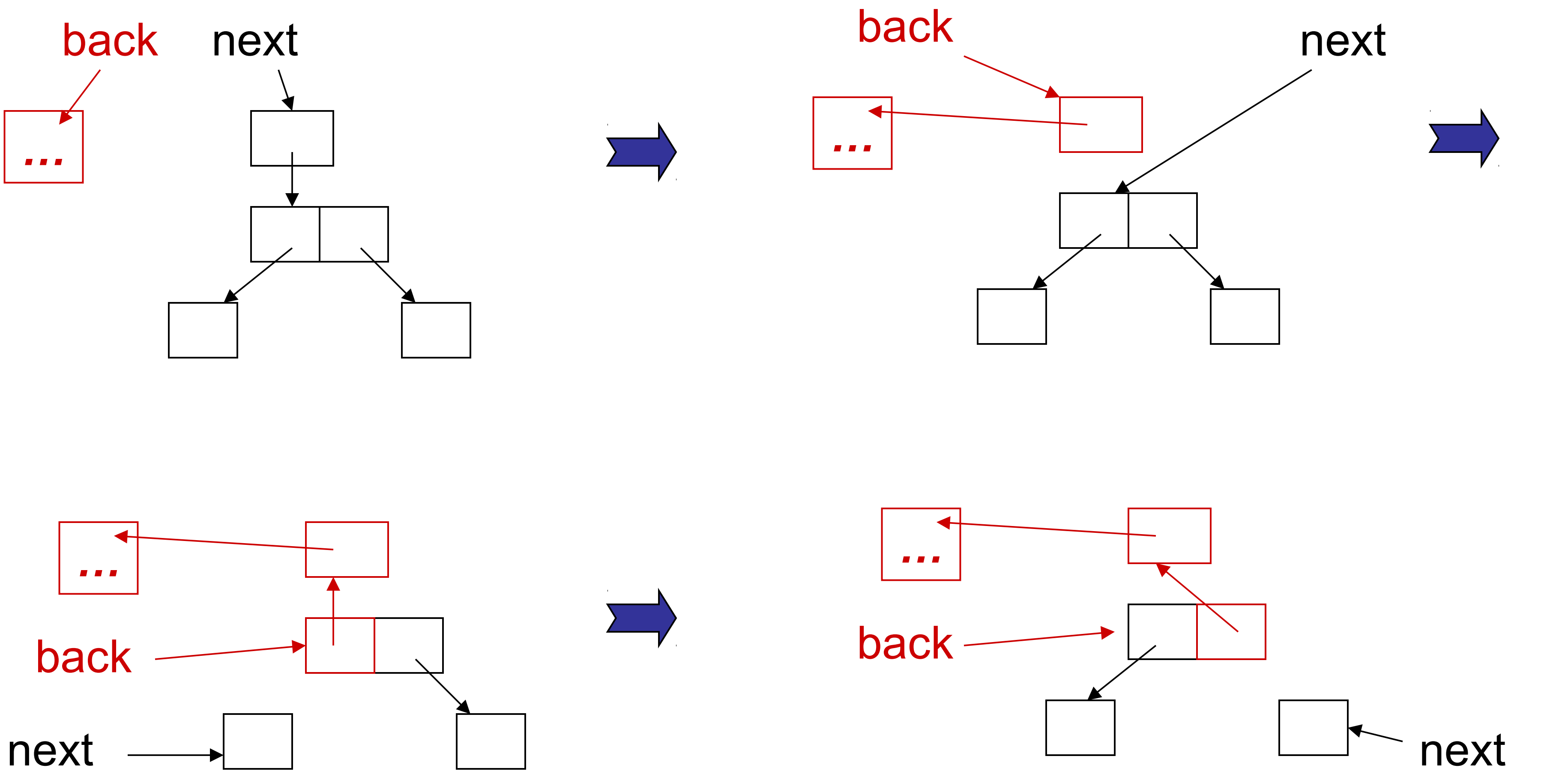
DSW Algorithm



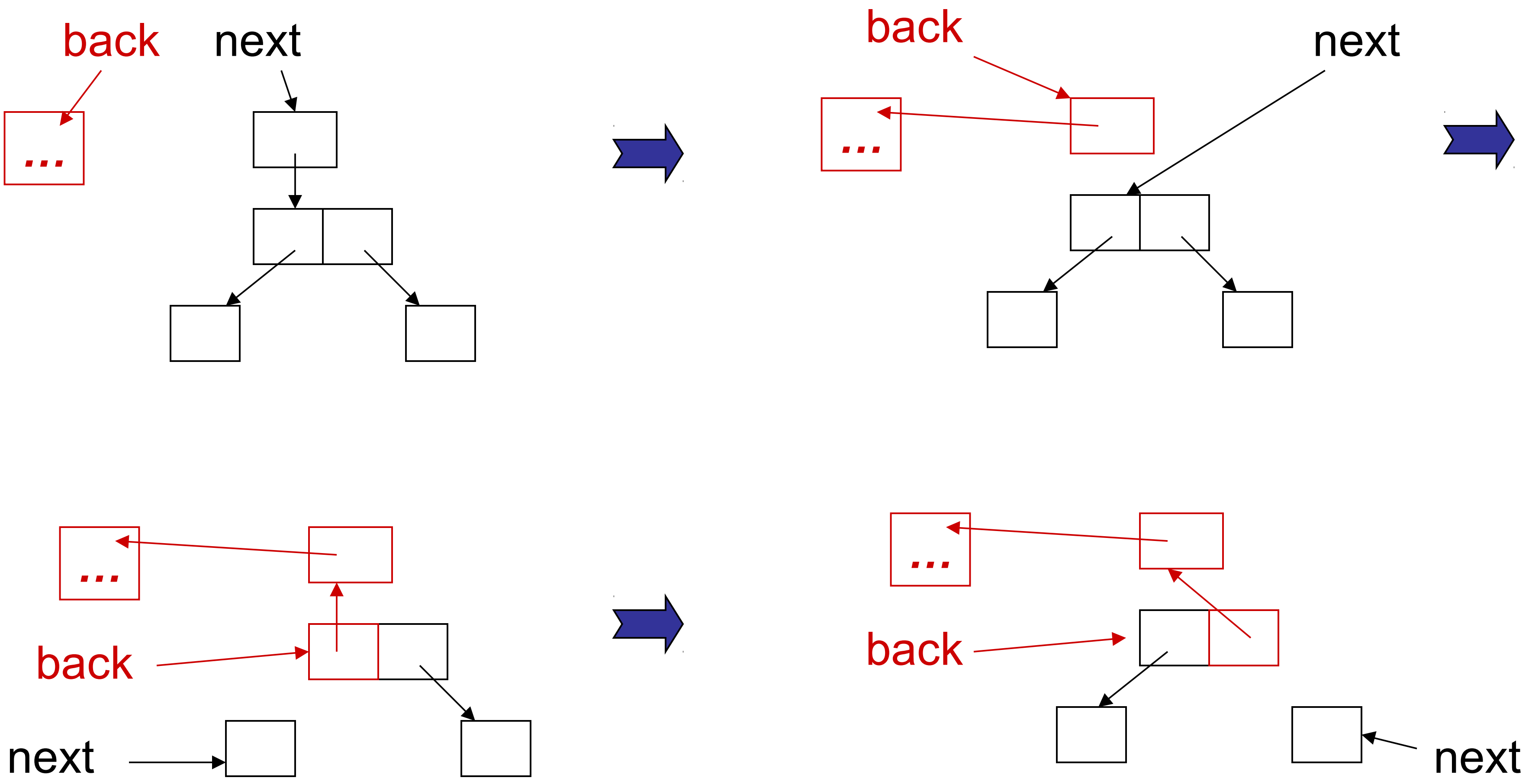
DSW Algorithm



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 vector: done[x]
- 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

DSW Algorithm

```
fun dfs(next) =
if (pointer next) &
  not (marked next) then

  (* initialization *)

  while true do

    i = done[next]
    if i < (fields next) then

      (* process ith field *)

    else

      (* back-track to previous
         record *)
```

(* depth-first search in
constant space *)

(* next is object being processed *)

(* done[next] is field being processed *)

DSW Algorithm

```
fun dfs(next) =                    (* depth-first search in
if (pointer next) &                   constant space *)
  not (marked next) then

  (* initialization *)

  while true do

    i = done[next]
    if i < (fields next) then

      (* process ith field *)

    else

      (* back-track to previous
         record *)
```

```
back = nil;
mark next;
done[next] = 0;
```

DSW Algorithm

```
fun dfs(next) =                          (* depth-first search in
if (pointer next) &                         constant space *)
   not (marked next) then

 (* initialization *)

 while true do

   i = done[next]
   if i < (fields next) then

     (* process ith field *)

   else

     (* back-track to previous
        record *)
```

```
y = next.i
if (pointer y) & not (marked y) then
  next.i = back;
  back = next;
  next = y;
  mark next;
  done[next] = 0;
else
  done[next] = i + 1
```

reuse field to store back ptr

DSW Algorithm

```
fun dfs(next) =                    (* depth-first search in
if (pointer next) &                   constant space *)
   not (marked next) then

 (* initialization *)

 while true do

   i = done[next]
   if i < (fields next) then

     (* process ith field *)

   else

     (* back-track to previous
        record *)
```

```
y = next.i
if (pointer y) & not (marked y) then
  next.i = back;
  back = next;
  next = y;
  mark next;          initialize for
  done[next] = 0;     next iteration
else
  done[next] = i + 1
```

DSW Algorithm

```
fun dfs(next) =
if (pointer next) &
   not (marked next) then

 (* initialization *)

 while true do

   i = done[next]
   if i < (fields next) then

     (* process ith field *)

   else

     (* back-track to previous
        record *)
```

(* depth-first search in constant space *)

```
y = next.i
if (pointer y) & not (marked y) then
  next.i = back;
  back = next;
  next = y;
  mark next;
  done[next] = 0;
else
  done[next] = i + 1
```

field is done

DSW Algorithm

```
fun dfs(next) =
if (pointer next) &
    not (marked next) then

  (* initialization *)

  while true do

    i = done[next]
    if i < (fields next) then

      (* process ith field *)

    else

      (* back-track to previous
         record *)
```

(* depth-first search in constant space *)

```
y = next;
next = back;

if next = nil then return;

i = done[next];
back = next.i;
next.i = y;
done[next] = i + 1;
```

dfs complete

DSW Algorithm

(* depth-first search in constant space *)

```
fun dfs(next) =
if (pointer next) &
   not (marked next) then

  (* initialization *)

  while true do

    i = done[next]
    if i < (fields next) then

      (* process ith field *)

    else

      (* back-track to previous
         record *)
```

```
y = next;
next = back;

if next = nil then return;

i = done[next];
back = next.i;
next.i = y;
done[next] = i + 1;
```

advance to next field

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 doesn’t require 2x live data size to operate
  - but if the ratio of live data to heap size is too large then performance suffers

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
    - 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 allocated 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 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

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
    - for ML, Tiger, Pascal:
      - every record has a header with size and pointer info
    - for Java, Modula-3:
      - 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
  - read Appel Chapter 13 and be able to analyze, compare and contrast different GC mechanisms