编译原理

13. 垃圾回收

rainoftime.github.io
浙江大学
计算机科学与技术学院

Content

Outline

1. Introduction
2. Mark-and-Sweep
3. Reference Count
4. Copying Collection
5. Interface to the Compiler

Runtime System

- Runtime system: the stuff that the language implicitly assumes and that is not described in the program
  - Handling of POSIX signals
    - POSIX = Portable Operating System Interface
  - Automated core management (e.g., work stealing)
  - Virtual machine execution (just-in-time compilation)
  - Class loading
  - **Automated memory management (e.g., garbage collection)**
  - …
- Also known as “language runtime” or just “runtime”

Memory Management

- **Problems with manual management**
  - Memory leaks, double frees, use-after frees,…
- **Storage bugs are hard to find**
  - A bug can lead to a visible effect far away in time and program text from the source
- **How to manage the memory**?
  - For performance, productivity, safety & security

Major Areas of Memory

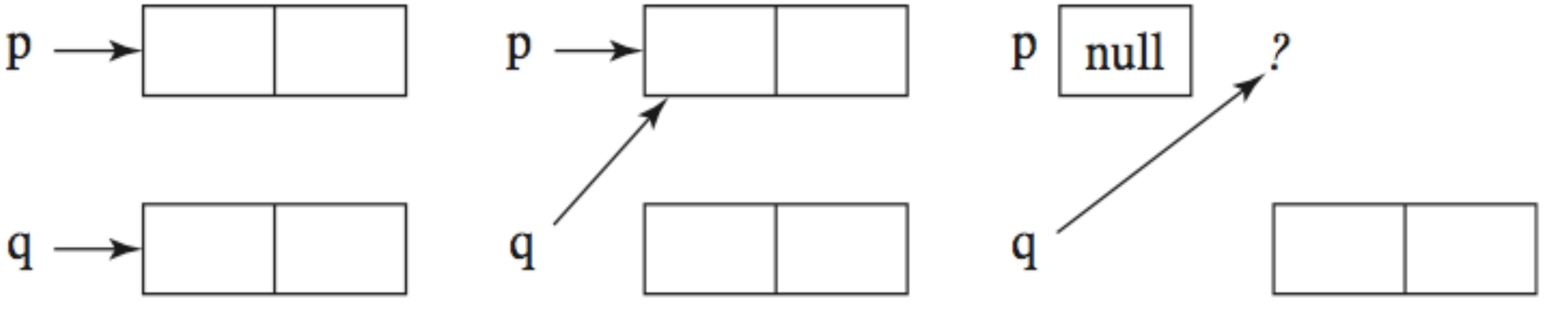
- Static area
  - Allocated at compile time
- Run-time stack
  - Activation records
  - Used for managing function calls and returns
- Heap
  - Dynamically allocated objects and data structures
    - Examples: malloc in C, new in Java

Garbage Collection: What

- Garbage: Allocated but no longer used heap storage

```
class node {
    int value;
    node next;
}
node p, q;
```

```
p = new node();
q = new node();
q = p;
delete p;
```



(a) (b) (c)

Garbage Collection: What

- Garbage: Allocated but no longer used heap storage
- **Garbage collection:** automatically frees storage which is not used by the program any more
  – Is part of the runtime system
  – First application in LISP (McCarthy 1960)

John McCarthy
1971 Turing Award

Garbage Collection: What

- Garbage: Allocated but no longer used heap storage
- **Garbage collection:** automatically frees storage which is not used by the program any more
  - Is part of the runtime system
  - First application in LISP (McCarthy 1960)
- A garbage collector has two phases
  - Garbage detection: finds which objects are alive and which dead
  - Garbage reclamation: deallocates dead objects

The Perfect Garbage Collector

- An ideal garbage collector would have the attributes
  - **Safe**: only garbage is reclaimed
  - "**Complete**": almost all garbage is collected
  - **Low overhead** in time and sped
  - Short **pause time** (the program waists for the collector)
  - **Parallel**: able to utilize additional cores
  - **Simple** 😄
  - **Easy to use collected free space**
  - **…?**

These are difficult and often conflicting requirements!

Garbage Collection: How

- Garbage: Allocated but no longer used heap storage

- **Question**: When is memory cell M not any longer used?
  - Let P be any program not using M
  - New program sketch:

    Execute P; Use M;
  - Hence:

    M used $\Leftrightarrow$ P terminates
  - We are doomed: halting problem!
- So “last use / non longer used” is undecidable!

Garbage Collection: How

- Garbage: Allocated but no longer used heap storage
- In general, it is undecidable whether an object is garbage
  - Need to rely on a conservative approximation
  - So that the GC is SAFE (only garbage is reclaimed)

Garbage Collection: How

- Garbage: Allocated but no longer used heap storage
- In general, it is undecidable whether an object is garbage
  - Need to rely on a conservative approximation
  - So that the GC is SAFE (only garbage is reclaimed)
- **Idea**: Use reachability information as "approximation"
  - Heap-allocated records unreachable by any chain of pointers from program variables are garbage
  - By "conservative approximation", we mean

Unreachable → not live (no longer used)

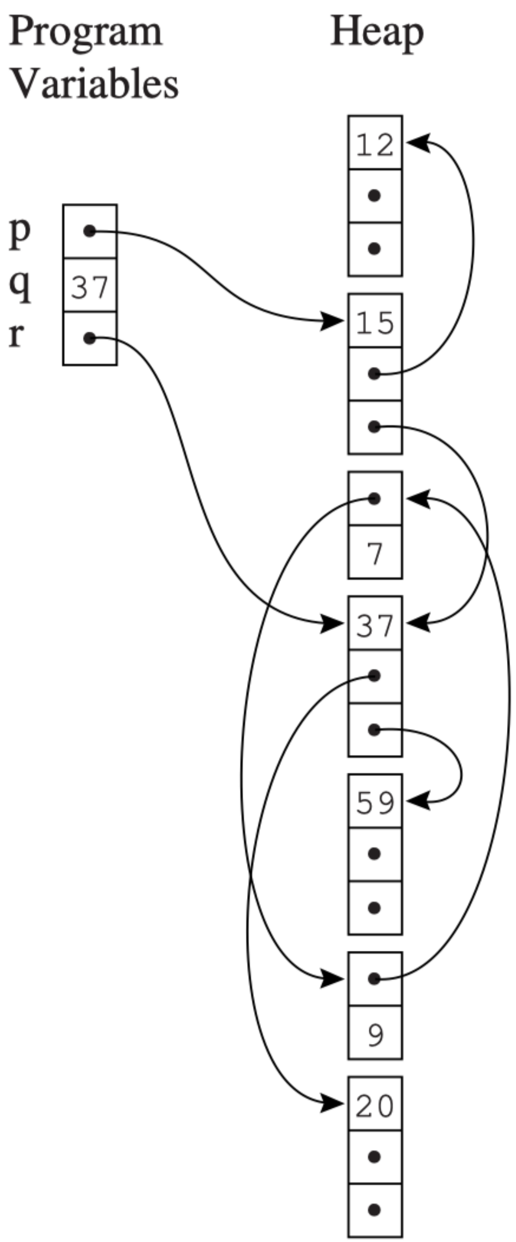
Garbage Collection: How

- Garbage: Allocated but no longer used heap storage
- In general, it is undecidable whether an object is garbage
  – Need to rely on a conservative approximation
  – So that the GC is SAFE (only garbage is reclaimed)
- **Idea**: Use reachability information as "approximation"
- **Key problem**: How to decide/check reachability?

Summary of GC Techniques

- **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
  - Copy collection
- **Modern techniques**: generational GC, etc.

Basic Data Structure: Directed Graph

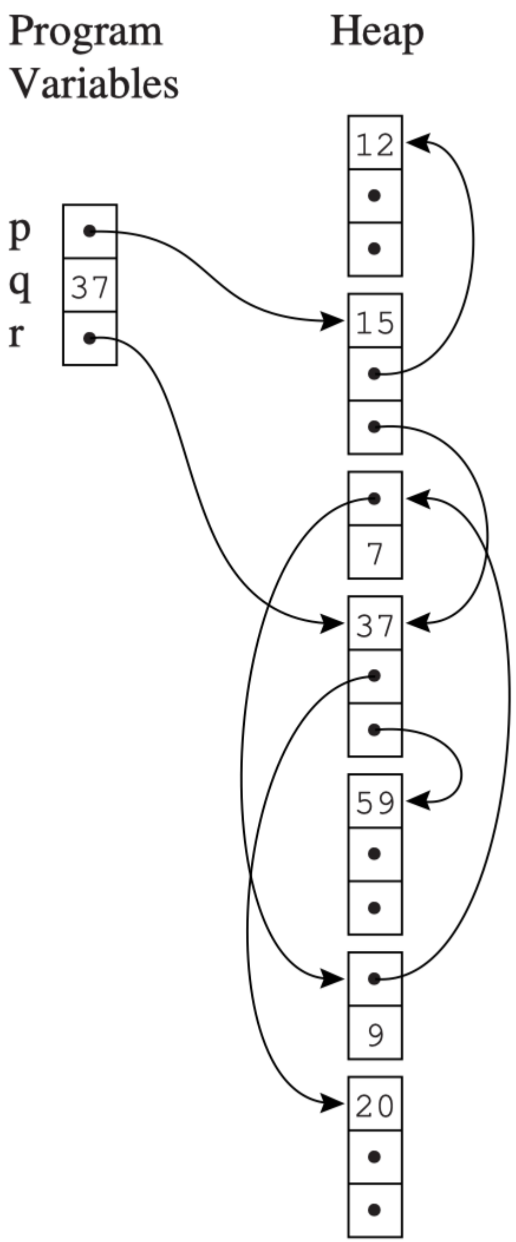


- **Directed graph**
  - nodes: program variables and heap-allocated records
  - edges: pointers

- **Root**: the program variables are roots of this graph.
  - Registers
  - Local vars/formal parameters on stack
  - global variables

- A node is **reachable** if there is a path of directed edges r -> ... -> n
  - r: some root

More about the Directed Graph



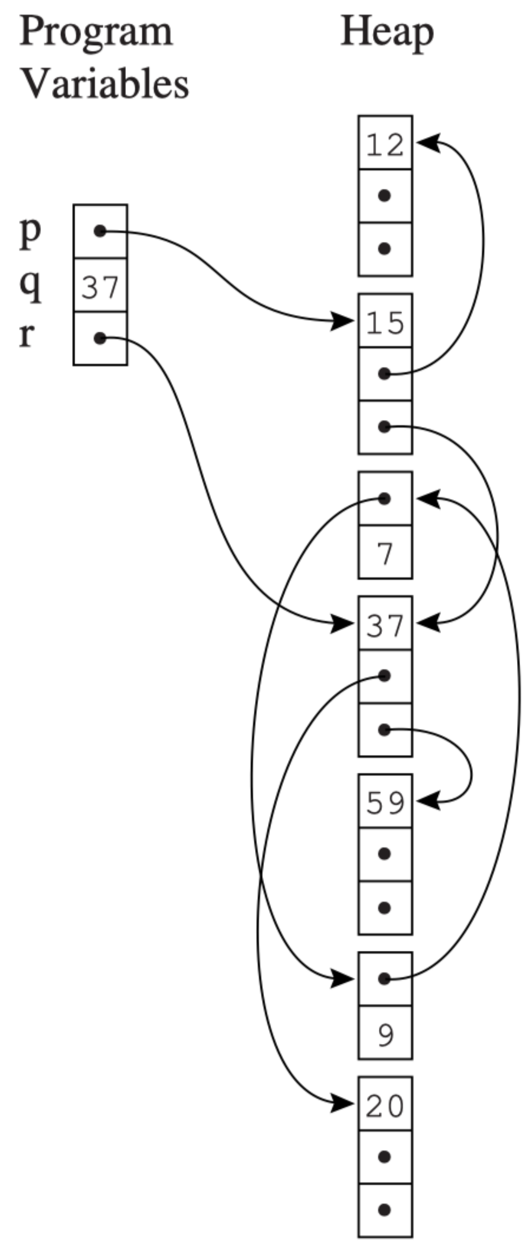
- **Directed graph**
  - nodes: program variables and heap-allocated records
  - edges: pointers

- On understanding the "pointers"
  - p points to a record y: we mean the value of p is the address of y (let y be the name /identifier of the record)

In this lecture, we may use "p" to refer to either the pointer **OR** the record it points to (as in Tiger book…)

Garbage Collection: Example

```
let
  type list = {link: list, key: int}
  type tree = {key: int, left: tree,
right: tree}
  function maketree() = ···
  function showtree(t: tree) = ···
in
  let var x := list{link=nil,key=7}
          var y := list{link=x,key=9}
  in x.link := y
  end;
  let var p := maketree()
      var r := p.right
      var q := r.key
  in garbage-collect here
    showtree(r)
  end
end
```



1. Mark-and-Sweep

- Mark-and-Sweep
- Explicit Stack
- Pointer Reversal

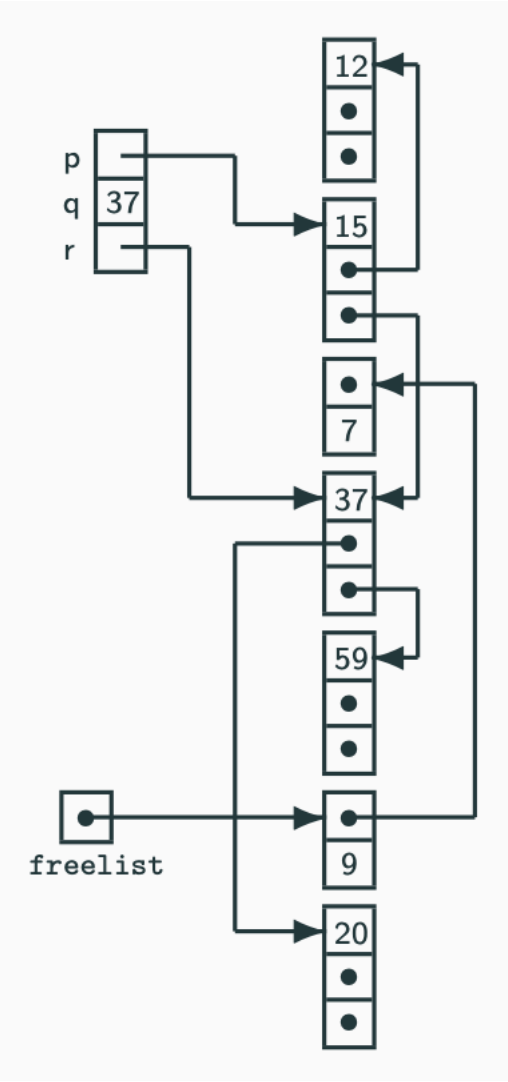
Mark-and-Sweep: Overview

- **Mark**
  - Search the graph from the roots (program variables)
  - Mark all the nodes searched
- **Sweep**
  - Sweep the entire heap by a linear scan, putting the unmarked nodes to a freelist
  - Unmark marked nodes

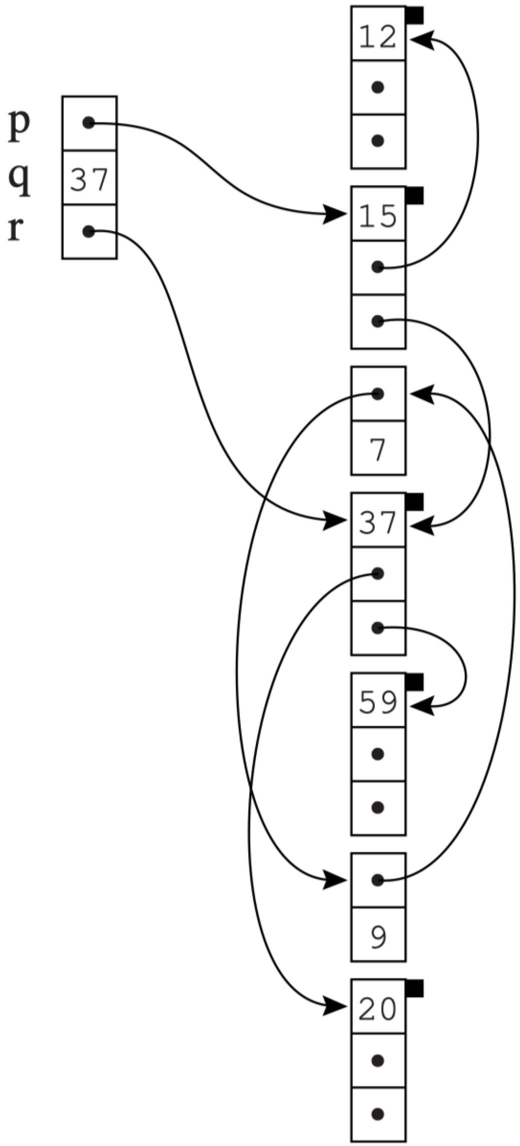
Additional Basic Data Structures: Freelist

freelist:

- The memory manager must know which parts of the heap are free and allocated
- Free blocks are stored in a free list (a linked list of heap blocks)
- Used by several GC algorithms (e.g., marked-and-sweep)



Mark-and-Sweep: Mark



(a) Marked

- A graph-search alg. such as depth-first search can mark all the reachable nodes.

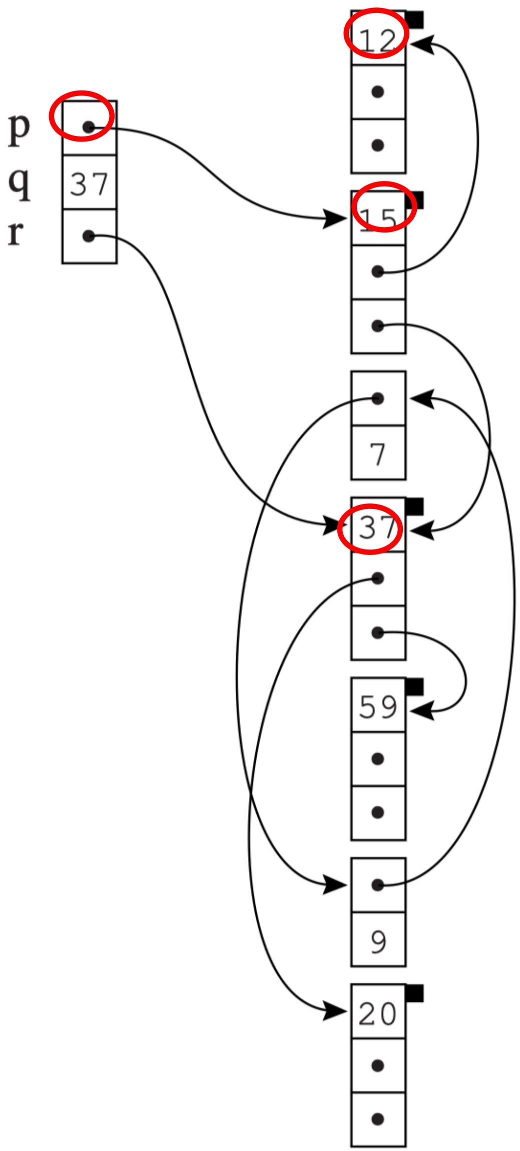
for each root v
  DFS(v)

function DFS(x)
if *x* is a pointer into the heap
  if record *x* is not marked
    mark *x*
    for each field *fi* of record *x*
      DFS(x.fi)

??x到底是pointe
还是record

- x的值是某record的地址，换句话说它指向该record
- 如果给该record一个名字(比如y), 就相当于说"x is a pointer and it points to record y"

Mark-and-Sweep: Mark



(a) Marked

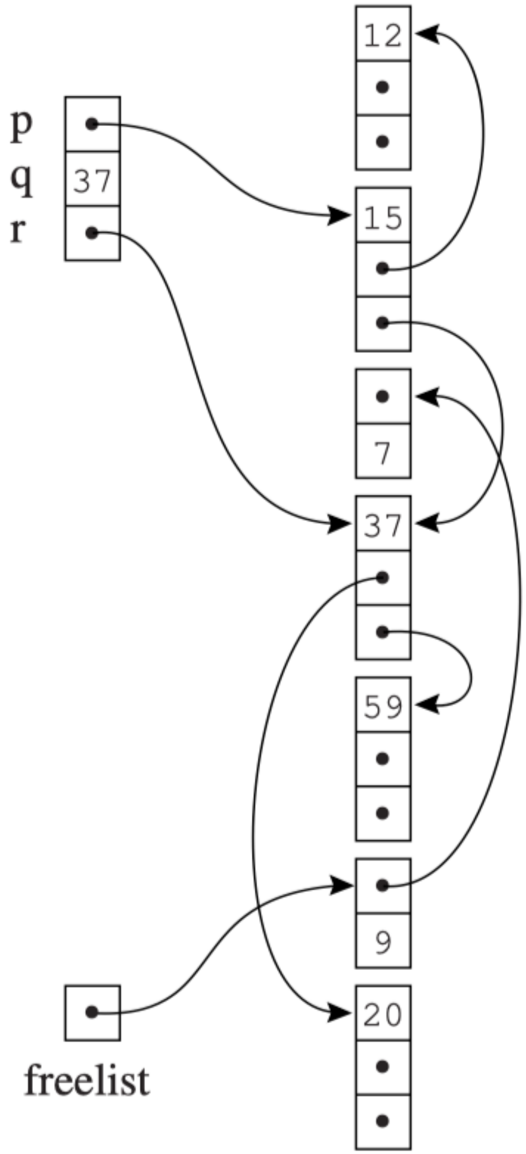
- A graph-search alg. such as depth-first search can mark all the reachable nodes.

for each root v
    DFS(v)

function DFS(x)
if *x* is a pointer into the heap
    if record *x* is not marked
        mark *x*
        for each field *fi* of record *x*
            DFS(x.fi)

E.g., 7 and 9 are not reachable!

Mark-and-Sweep: Sweep



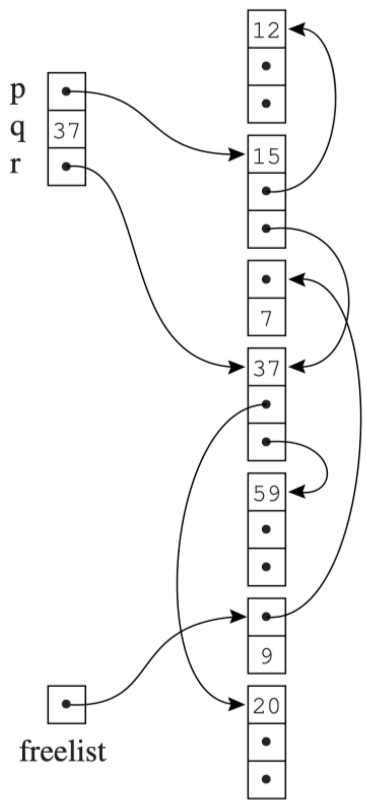
(b) Swept

- **Sweep** the entire heap, from its first address to its last
  - Find unmarked nods (garbage)
  - Link them together in *freelist*.

$p \leftarrow$ first address in heap
while $p <$ last address in heap
  if record p is marked
    unmark p
  else let f_1 be the first field in p
    $p.f_1 \leftarrow freelist$
    $freelist \leftarrow p$
  $p \leftarrow p +$ (size of record p)

E.g., 7 and 9 are added to the freelist

Mark-and-Sweep: Sweep



(b) Swept

$p \leftarrow$ first address in heap
while $p <$ last address in heap
if record p is marked
unmark p
else let f_1 be the first field in p
$p.f_1 \leftarrow freelist$
$freelist \leftarrow p$
$p \leftarrow p +$ (size of record p)

Mark-and-Sweep Collection

Mark phase:

for each root v
    DFS(v)

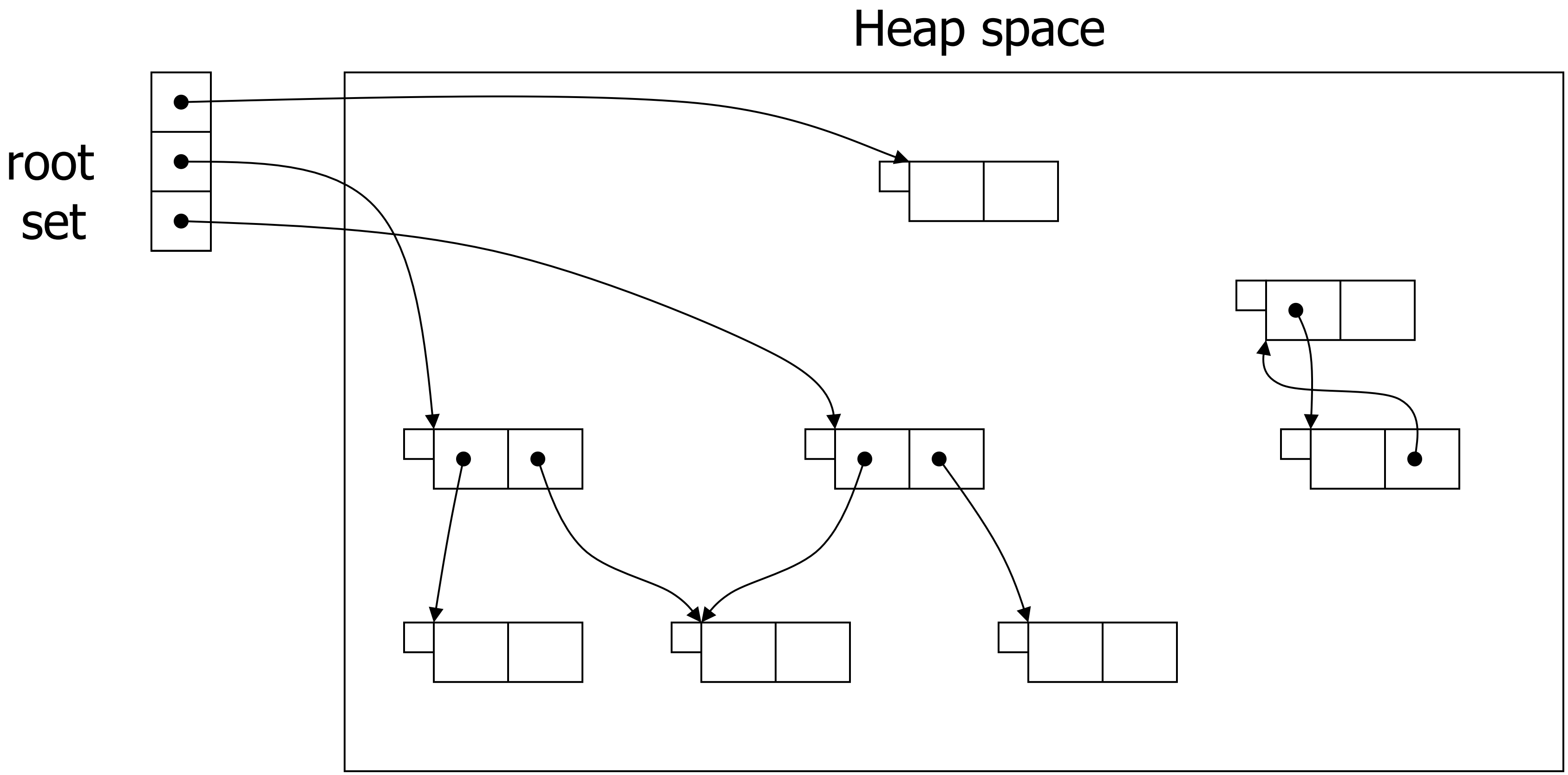
function DFS(x)
    if x is a pointer into the heap
        if record x is not marked
            mark x
            for each field *fi* of record x
                DFS(x.fi)

Sweep phase:

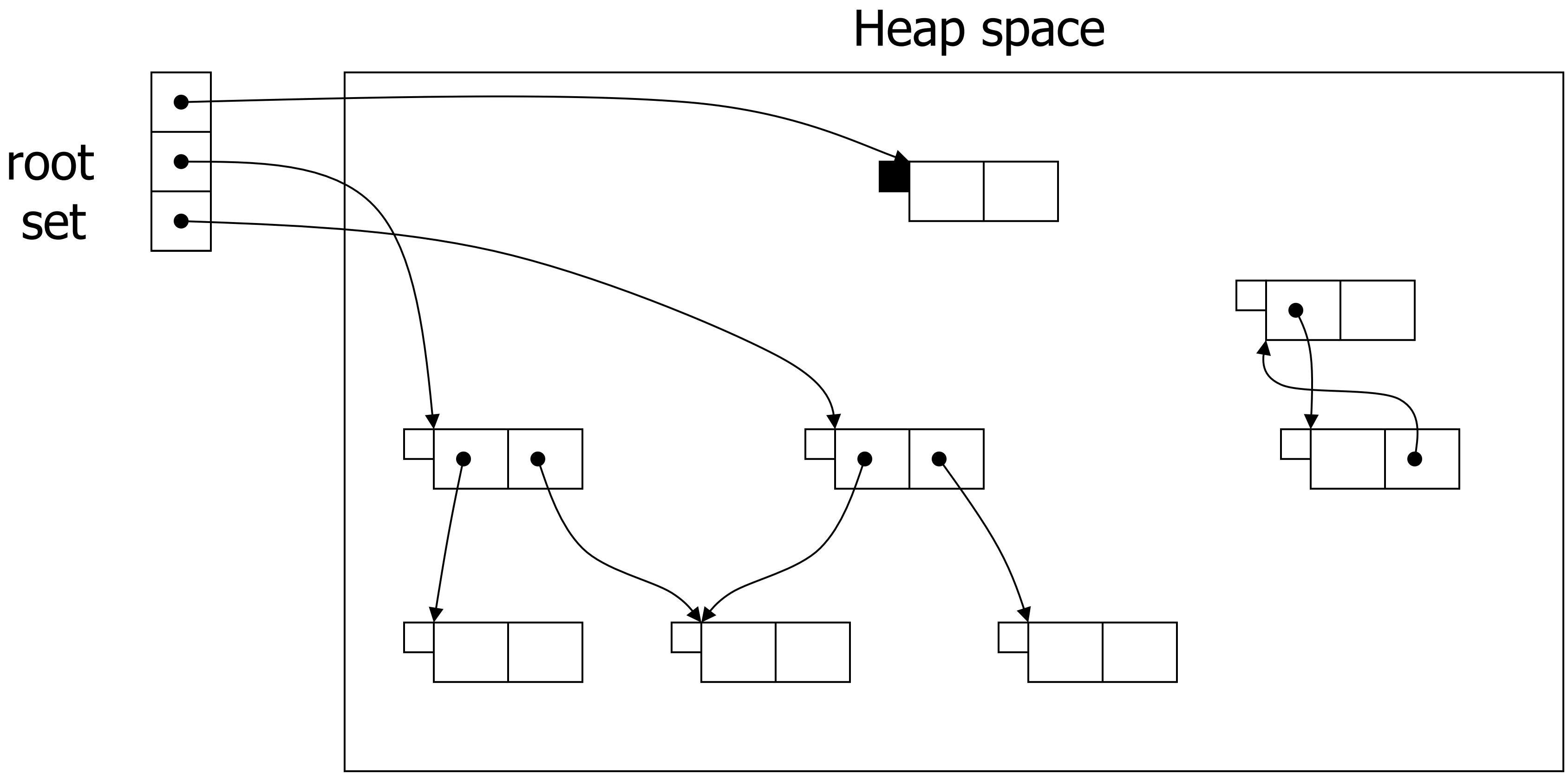
$p \leftarrow$ first address in heap
while $p <$ last address in heap
    if record p is marked
        unmark p
    else let f_1 be the first field in p
        $p.f_1 \leftarrow \mathit{freelist}$
        $\mathit{freelist} \leftarrow p$
    $p \leftarrow p +$ (size of record p)

- After the garbage collection, the program resumes execution.
- Whenever it wants to heap-allocate a new record, it gets a record from the freelist
- Do garbage collection again when freelist is empty !

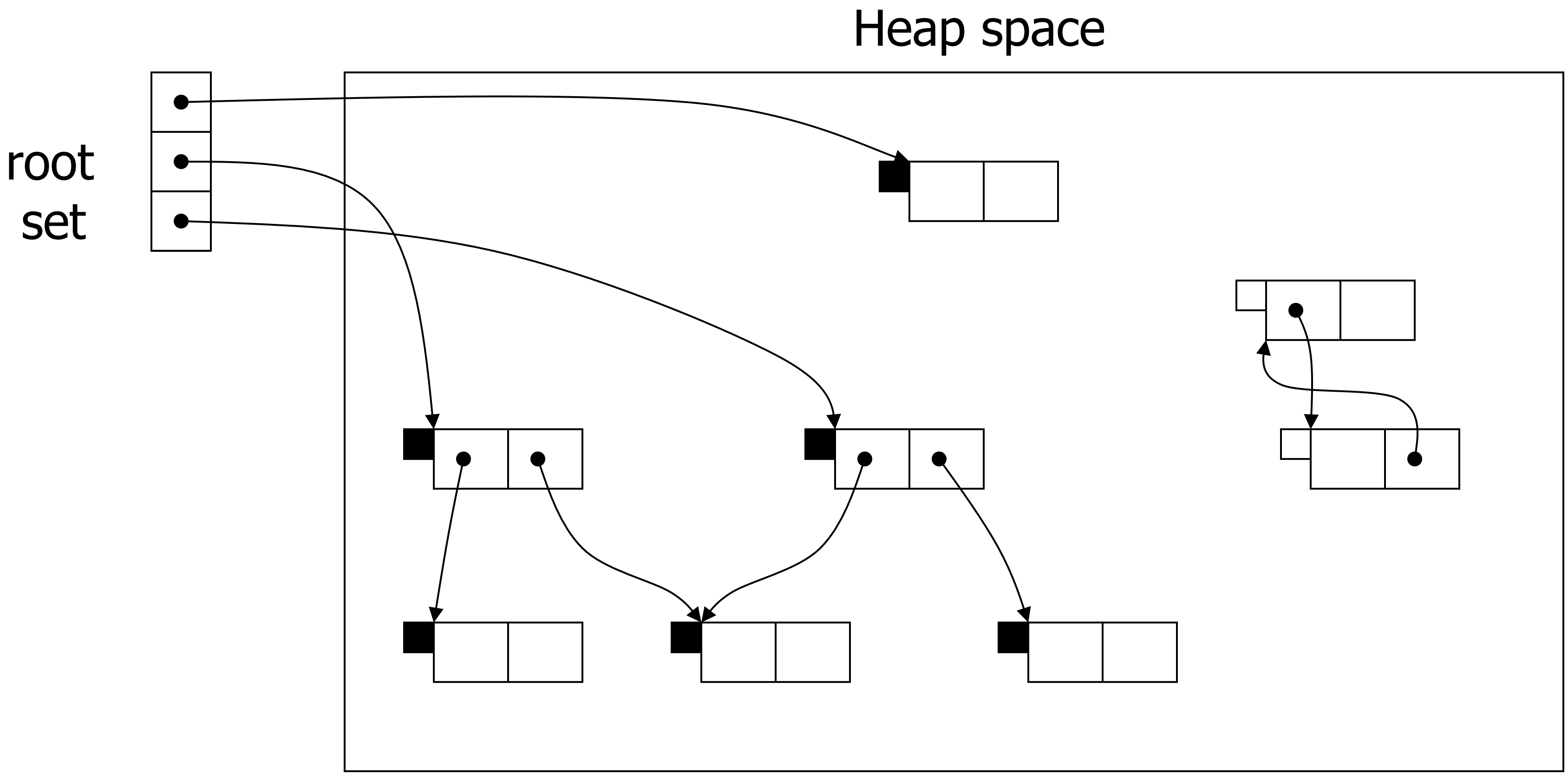
Example: Mark-and-Sweep



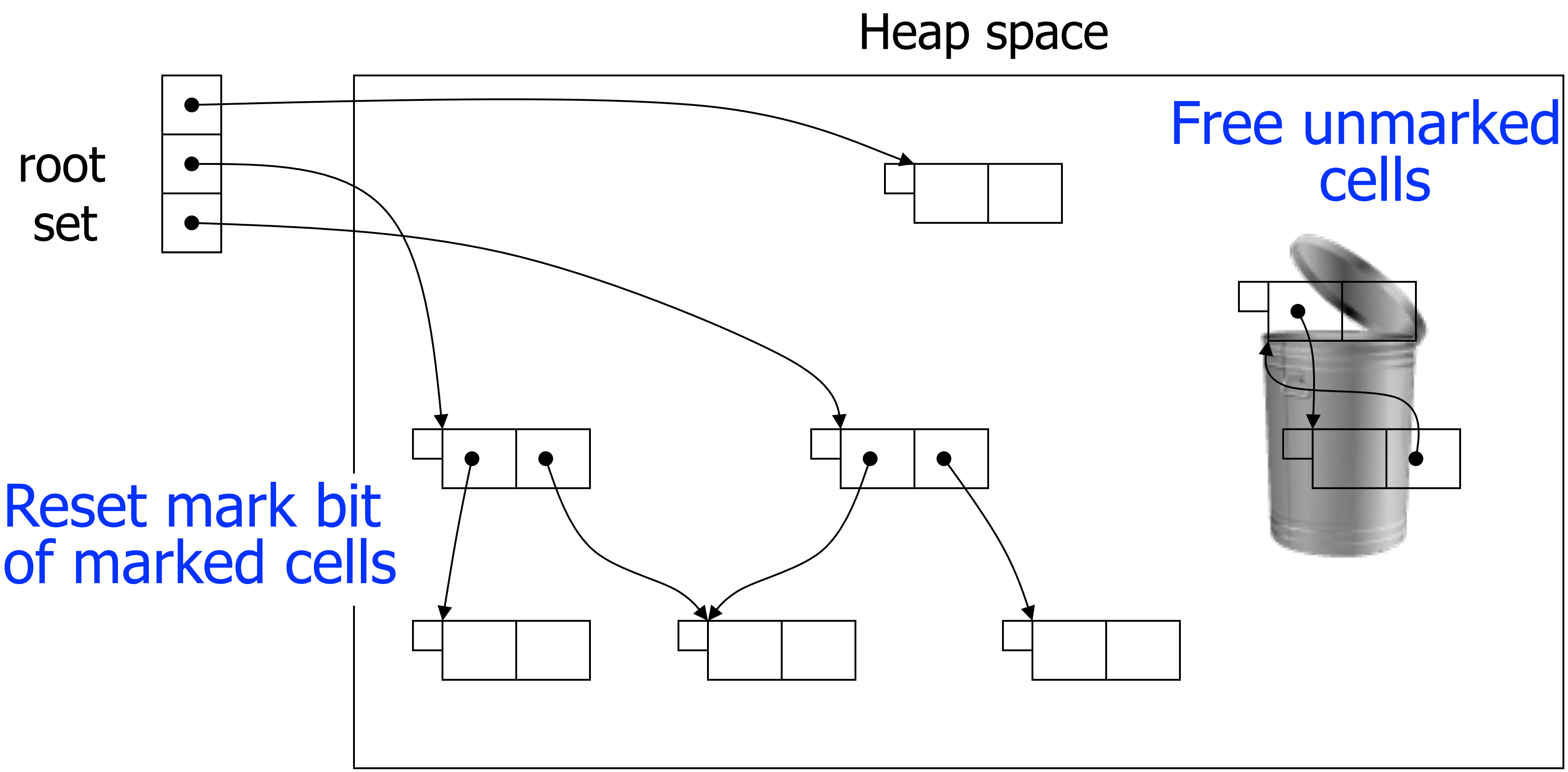
Example: Mark-and-Sweep



Example: Mark-and-Sweep



Example: Mark-and-Sweep



Cost of Mark-and-Sweep

- **H**: heap size; **R**: reachable data
- **Time of GC**
  – Mark: proportional to the amount of reachable data
  – Sweep: proportional the size of the heap
  – Total time: $c_1R + c_2H$

- GC replenish the freelist with H-R words
- Amortized cost $(c_1R + c_2H) / (H - R)$
  – If R is closed to H, the cost is very high

1. Mark-and-Sweep

- Mark-and-Sweep
- Explicit Stack
- Pointer Reversal

Implementation Issue of Mark-and-Sweep

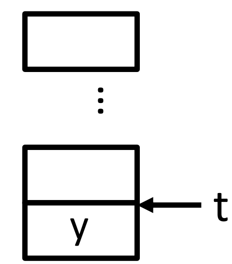
- The DFS algorithm is recursive
  - Extreme case: N stack frames for an N-elem linked list
  - The length of the stack of activation records would be larger than the entire heap
  - Easy to cause stack overflow!

- **Solution**: Use an explicit stack instead of recursion

Using Explicit Stack

- Benefit: H words instead of H activation records !

function DFS(x)
    if x is a pointer and record x is not marked
        mark x
        $t \leftarrow 1$
        stack[t] $\leftarrow x$ *//把深搜的起点加入*
        while $t > 0$
            $x \leftarrow$ stack[t]; $t \leftarrow t - 1$ *//取出栈顶元素*
            for each field f_i of record x
                if $x.f_i$ is a pointer and record $x.fi$ is not marked
                    //指向了一个没有标记过的record
                    mark $x.fi$ *// 可以理解为"被x.fi指向的record"*
                    $t \leftarrow t + 1$; stack[t] $\leftarrow x.fi$ *// 加入栈中*



stack

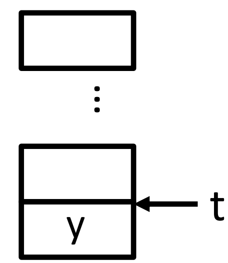
- t: the top of the stack
- stack: a worklist

Using Explicit Stack

- Benefit: H words instead of H activation records !

function DFS(x)
  if x is a pointer to record y which is not marked
    mark y
    $t \leftarrow 1$
    stack[t] $\leftarrow y$ //把深搜的起点加入
    while $t > 0$
      $y \leftarrow$ stack[t]; $t \leftarrow t - 1$ //取出栈顶元素
      for each field f_i of record y
        if $y.f_i$ is a pointer to record z which is not marked
          //指向了一个没有标记过的record
          mark z
          $t \leftarrow t + 1$; stack[t] $\leftarrow z$ // *加入栈中*

- t: the top of the stack
- stack: a worklist



stack

However, it is still unacceptable to require auxiliary stack memory as large as the heap being collected!

1. Mark-and-Sweep

- Mark-and-Sweep
- Explicit Stack
- Pointer Reversal

Pointer Reversal

- **Problem**: Depth-first search needs a stack
  - Stack depth could be as big as the graph
- **Solution:** Deutsch-Schorr-Waite (DSW) pointer reversal
  - Don't use an explicit stack for DFS
  - Reuse the graph components to assist backtracking

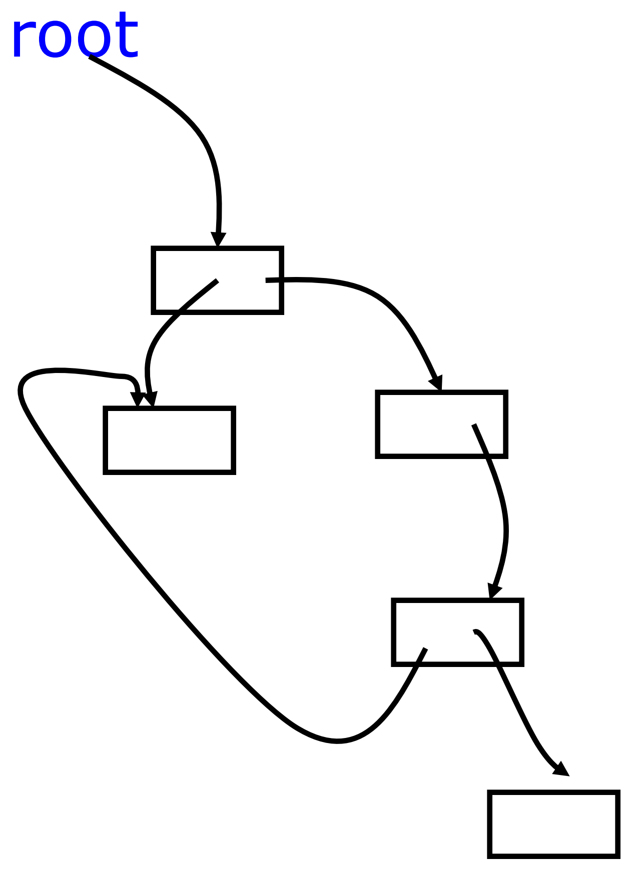
Developed independently by Schorr and Waite (1967)
and by Deutsch (1973)

Pointer Reversal

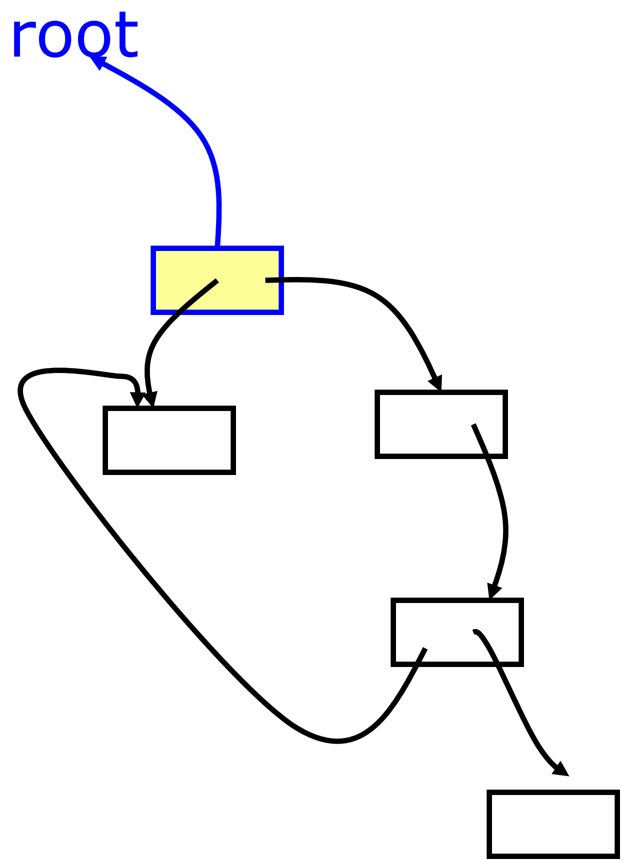
- **The basic idea**: store the DFS stack in the graph itself.

- When a new record is encountered during the search
  – Mark the record
  – Change a pointer in the record to point back to the DFS parent record
  – When we can go no deeper, return, following the back links, restoring the links.

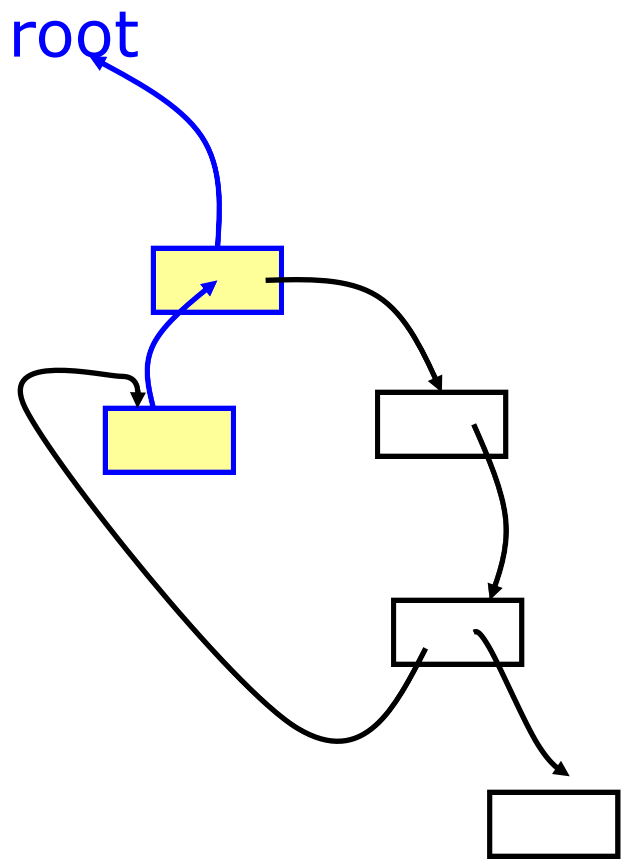
Example: Pointer Reversal



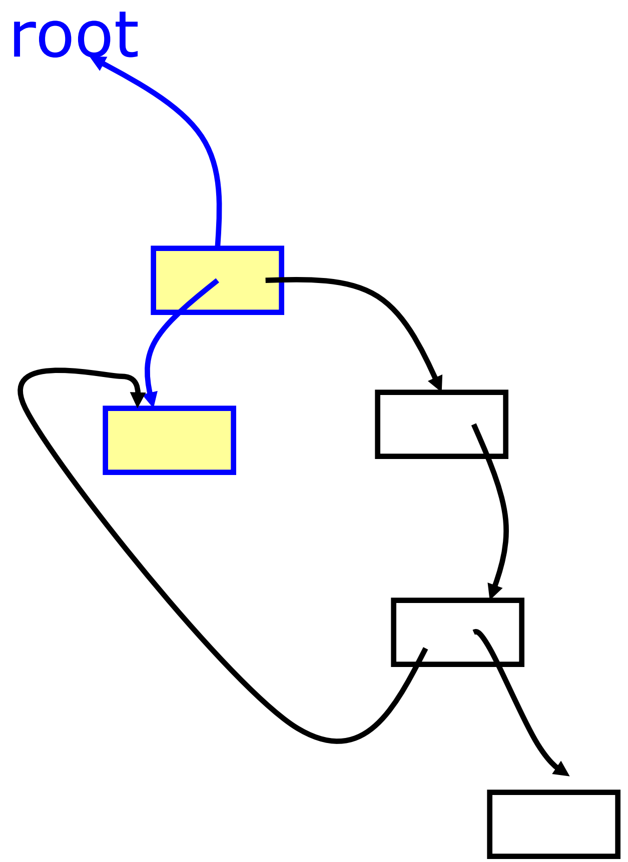
Example: Pointer Reversal



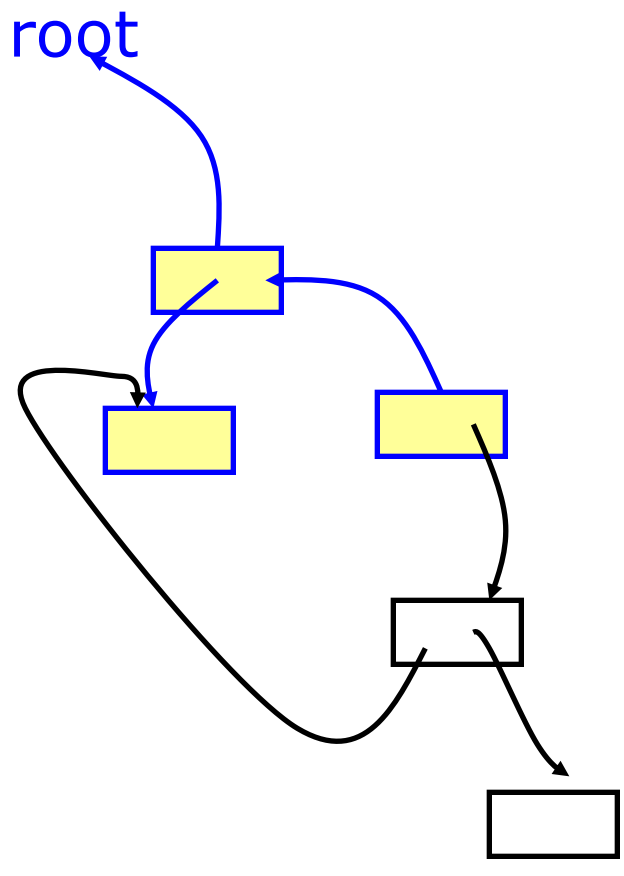
Example: Pointer Reversal



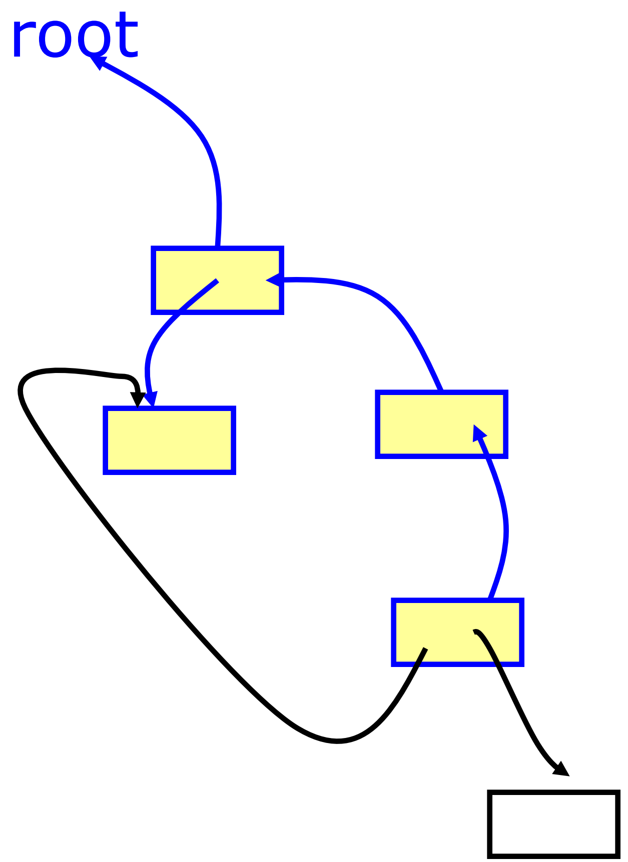
Example: Pointer Reversal



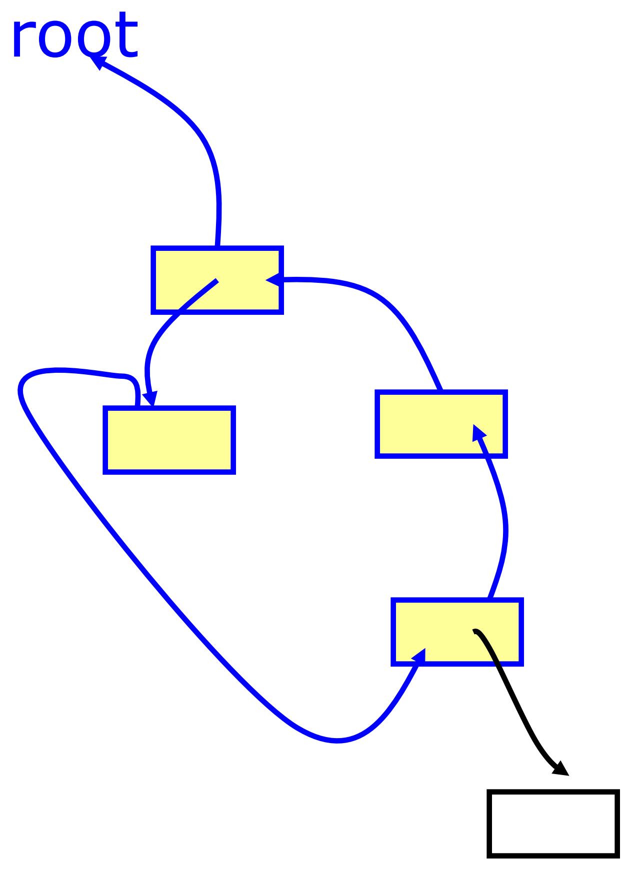
Example: Pointer Reversal



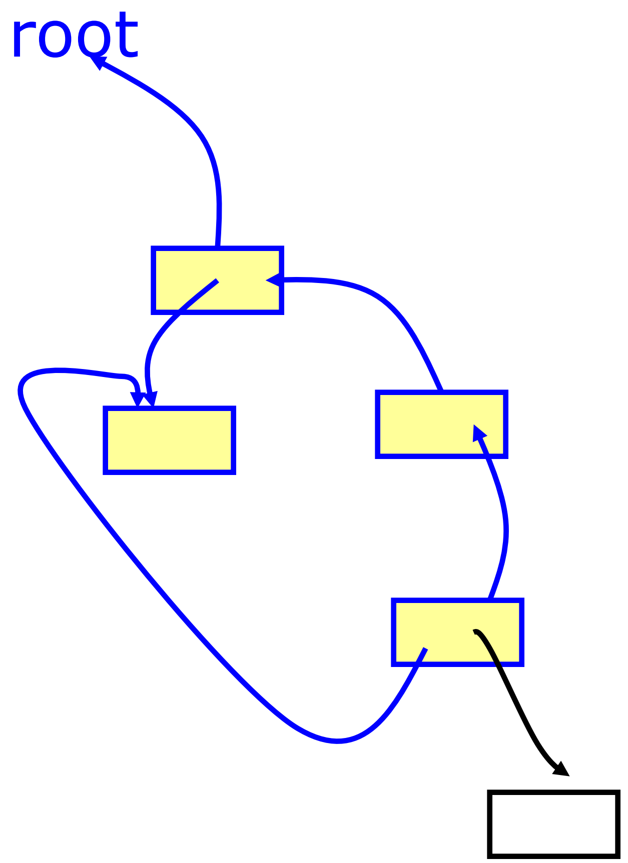
Example: Pointer Reversal



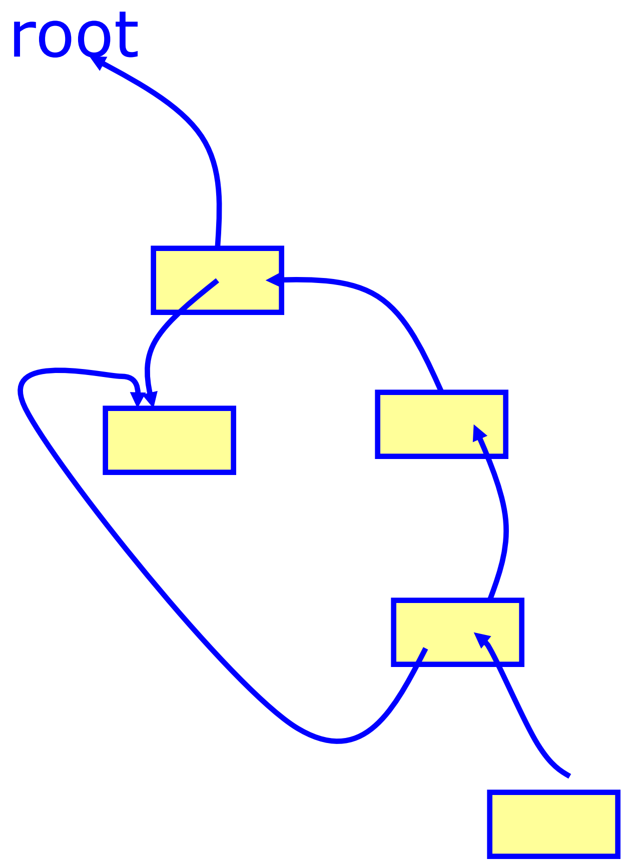
Example: Pointer Reversal



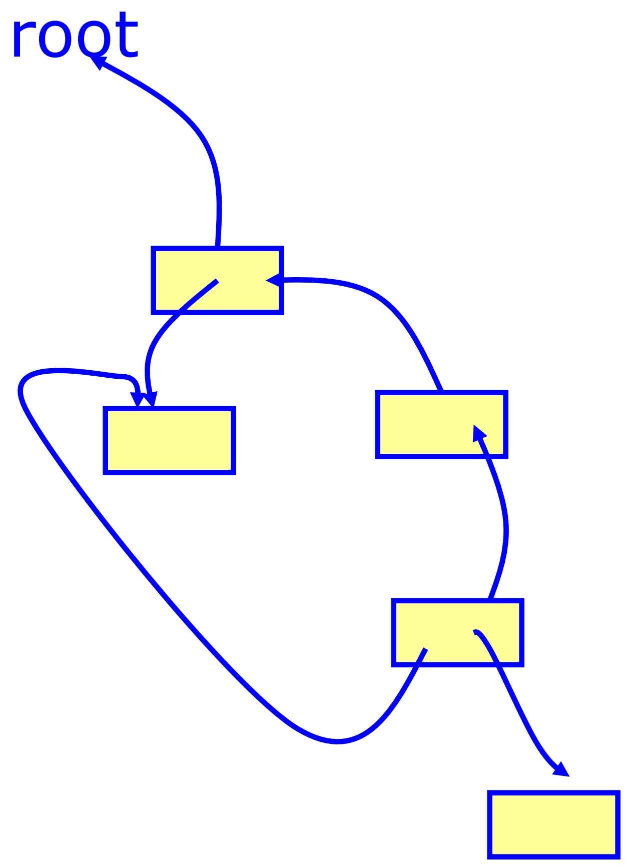
Example: Pointer Reversal



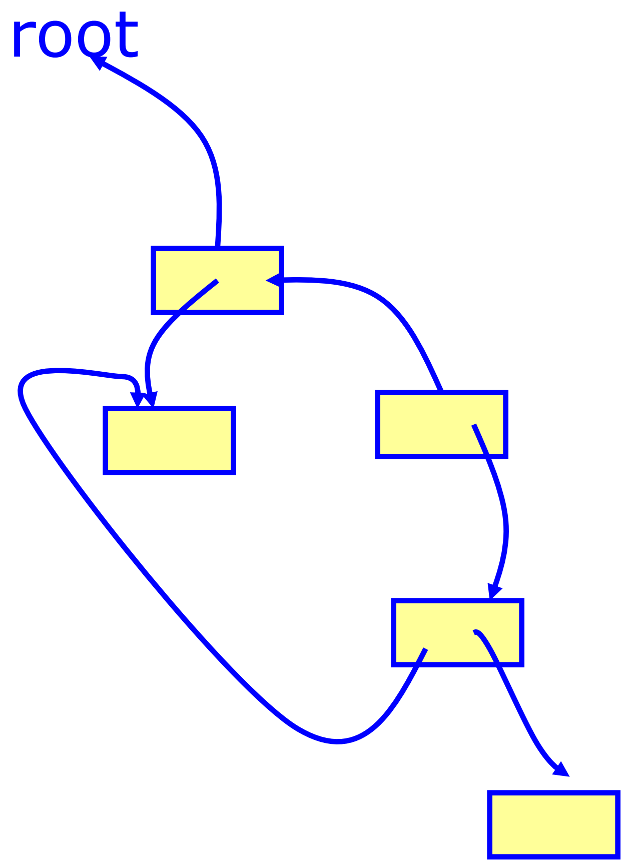
Example: Pointer Reversal



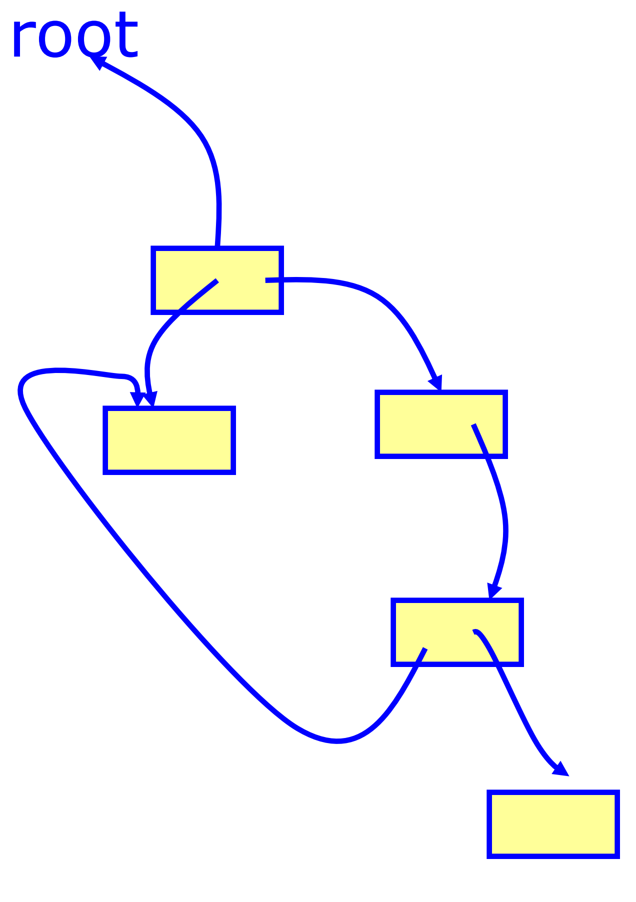
Example: Pointer Reversal



Example: Pointer Reversal



Example: Pointer Reversal



Pointer Reversal

- **Which to reuse?**
- **Observation**
  - After the contents of field *x.fi* has been pushed on the stack, the algorithm will never again look the original location *x.fi*.
  - *x.fi* can be used to store one element of the stack
  - When the stack is popped, restore the original value of x.fi

Mark phase:

for each root v
    DFS(v)

function DFS(x)
  if *x* is a pointer into the heap
    if record *x* is not marked
      mark *x*
      for each field *fi* of record *x*
        DFS(x.fi)

Pointer Reversal Setup

function DFS(x)
  if x is a pointer and record x is not marked
    (* initialization *)

Call dfs passing each root as next

    while true
        …
        if i < # of fields in record x

Object being processed

        (* process ith field *)

        else
          (* back-track to previous record *)

Back-track during the DFS
Decide termination here

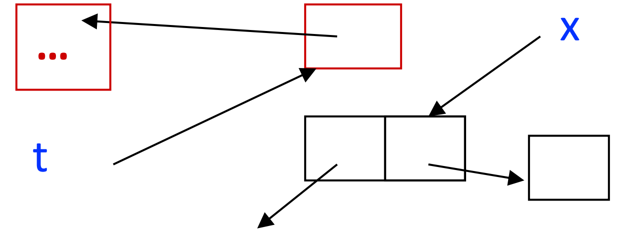
Pointer Reversal

```
function DFS(x)
 if x is a pointer and record x is not marked
   t ← nil
   mark x; done[x] ←  0
   while true
          i ← done[x]
          if i < # of fields in record x
            y ← x. fi
            if y is a pointer and record y is not marked
              x. fi ← t; t ← x; x ← y
              mark x; done[x] ← 0
            else
              done[x] ← i + 1
          else
            y ← x; x ← t
            if x = nil then return
            i ← done[x]
            t ← x. fi; x. fi ← y
            done[x] ← i + 1
```

// **done**: track whether the fields in each record have been processed

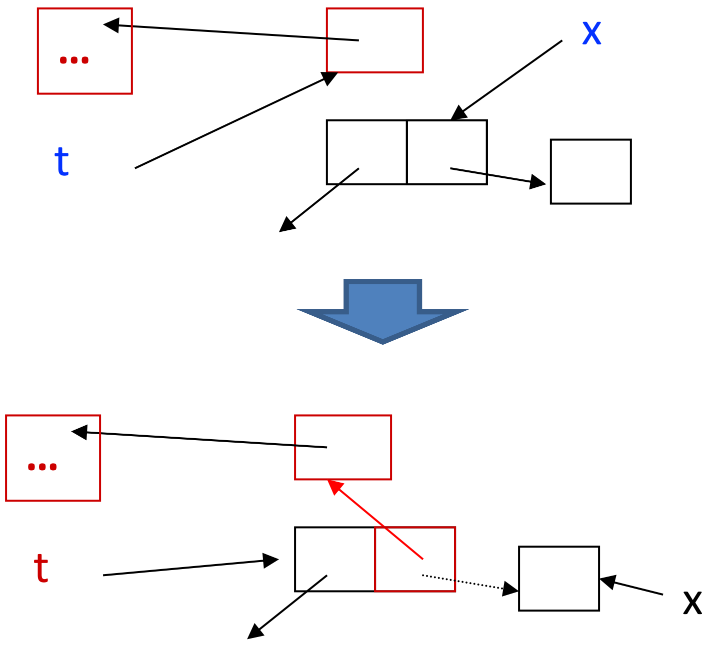
- Keep updating global pointers t and x
- Reuse filed x.fi to store the value of t

Pointer Reversal



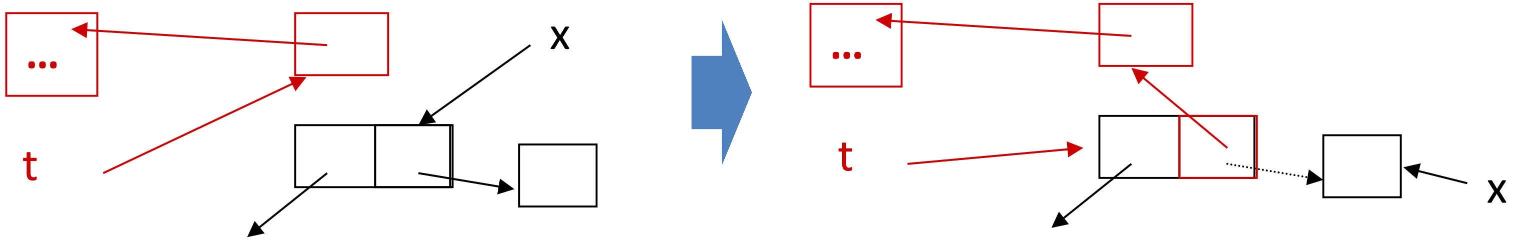

```
function DFS(x)
 if x is a pointer and record x is not marked
  t ← nil
  mark x; done[x] ← 0
  while true
        i ← done[x]
        if i < # of fields in record x
          y ← x. fi
          if y is a pointer and record y is not marked
            x. fi ← t; t ← x; x ← y      // reuse field x.fi  to store t!
            mark x; done[x] ← 0
          else
            done[x] ← i + 1
        else
          y ← x; x ← t
          if x = nil then return
          i ← done[x]
          t ← x. fi; x. fi ← y
          done[x] ← i + 1
```

Pointer Reversal




```
function DFS(x)
  if x is a pointer and record x is not marked
    t ← nil
    mark x; done[x] ← 0
    while true
          i ← done[x]
          if i < # of fields in record x
             y ← x. fi
             if y is a pointer and record y is not marked
                x. fi ← t; t ← x; x ← y       // reuse field x.fi  to store t!
                mark x; done[x] ← 0
             else
                done[x] ← i + 1
          else
             y ← x; x ← t
             if x = nil then return
             i ← done[x]
             t ← x. fi; x. fi ← y
             done[x] ← i + 1
```

Pointer Reversal



Pointer Reversal

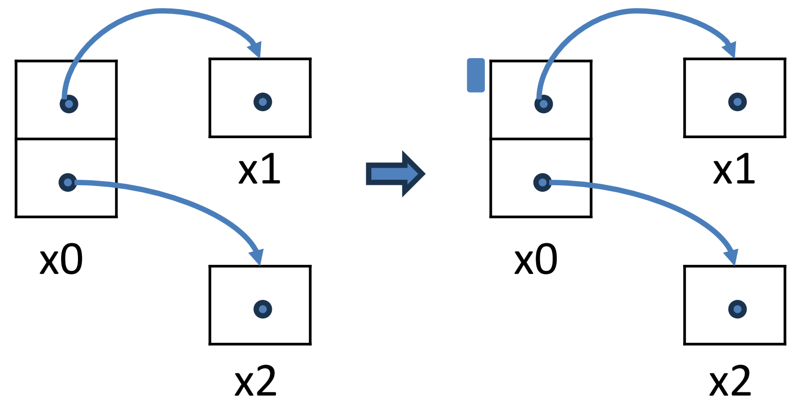
function DFS(x)
  if x is a pointer and record x is not marked
    $t \leftarrow$ nil
    mark x; done[x] $\leftarrow$ 0
    while true
        $i \leftarrow$ done[x]
        if $i <$ # of fields in record x
          $y \leftarrow x.\,f_i$
          if y is a pointer and record y is not marked
            $x.\,f_i \leftarrow t$; $t \leftarrow x$; $x \leftarrow y$
            mark x; done[x] $\leftarrow$ 0
          else
            done[x] $\leftarrow i + 1$
        else (* back-track to previous record *)
          $y \leftarrow x$; $x \leftarrow t$
          if $x =$ nil **then return** // DFS completes
          $i \leftarrow$ done[x]
          $t \leftarrow x.\,f_i$; $x.\,f_i \leftarrow y$
          done[x] $\leftarrow i + 1$

Pointer Reversal

```
function DFS(x)
 if x is a pointer and record x is not marked
  t ← nil
  mark x; done[x] ← 0
  while true
        i ← done[x]
        if i < # of fields in record x
          y ← x. fi
          if y is a pointer and record y is not marked
            x. fi ← t; t ← x; x ← y
            mark x; done[x] ← 0
          else
            done[x] ← i + 1
        else  (* back-track to previous record *)
          y ← x; x ← t
          if x = nil then return          // DFS completes
          i ← done[x]
          t ← x. fi; x. fi ← y            // When popping the stack, x.fi is restored
          done[x] ← i + 1                 // to its original value
```

Example: Pointer Reversal

function DFS(x)
  if x is a pointer and record x is not marked
    $t \leftarrow$ nil
    mark x; done[x] $\leftarrow$ 0
    while true
        $i \leftarrow$ done[x]
        if $i <$ # of fields in record x
          $y \leftarrow x.fi$
          if y is a pointer and record y is not marked
            $x.fi \leftarrow t$; $t \leftarrow x$; $x \leftarrow y$
            mark x; done[x] $\leftarrow$ 0
          else
            done[x] $\leftarrow i + 1$
        else
          $y \leftarrow x$; $x \leftarrow t$
          if x = nil **then return**
          $i \leftarrow$ done[x]
          $t \leftarrow x.fi$; $x.fi \leftarrow y$
          done[x] $\leftarrow i + 1$



After:

t = nil

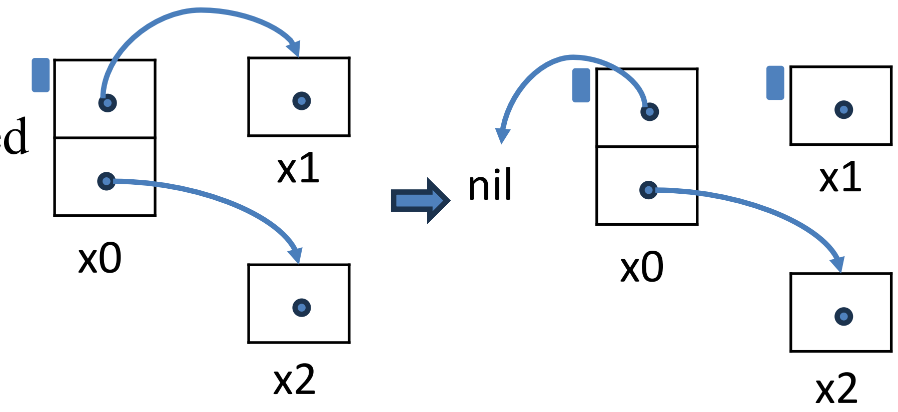
x = x0

mark x0

done[x0] = 0

Example: Pointer Reversal

function DFS(x)
  if x is a pointer and record x is not marked
    $t \leftarrow$ nil
    mark x; done[x] $\leftarrow$ 0
    while true
      $i \leftarrow$ done[x]
      if $i <$ # of fields in record x
        $y \leftarrow x.fi$
        if y is a pointer and record y is not marked
          $x.fi \leftarrow t$; $t \leftarrow x$; $x \leftarrow y$
          mark x; done[x] $\leftarrow$ 0
        else
          done[x] $\leftarrow i + 1$
      else
        $y \leftarrow x$; $x \leftarrow t$
        if $x =$ nil **then return**
        $i \leftarrow$ done[x]
        $t \leftarrow x.fi$; $x.fi \leftarrow y$
        done[x] $\leftarrow i + 1$

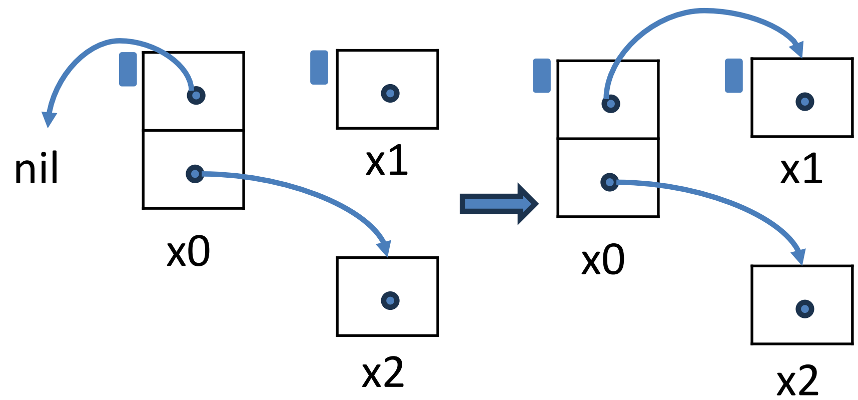


Before:
t = nil; x = x0;
done[x0] = 0

After:
i = done[x0] = 0
y = x0.f0 = x1
x0.f0 = t = nil
t = x = x0
x = y = x1
mark x1
done[x1] = 0

Example: Pointer Reversal

function DFS(x)
  if x is a pointer and record x is not marked
    $t \leftarrow$ nil
    mark x; done[x] $\leftarrow$ 0
    while true
      $i \leftarrow$ done[x]
      if $i <$ # of fields in record x
        $y \leftarrow x.\,fi$
        if y is a pointer and record y is not marked
          $x.\,fi \leftarrow t$; $t \leftarrow x$; $x \leftarrow y$
          mark x; done[x] $\leftarrow$ 0
        else
          done[x] $\leftarrow i + 1$
      else
        $y \leftarrow x$; $x \leftarrow t$
        if $x =$ nil **then return**
        $i \leftarrow$ done[x]
        $t \leftarrow x.\,fi$; $x.\,fi \leftarrow y$
        done[x] $\leftarrow i + 1$



Before:

t = x0

x = x1

done[x1] = 0

After:

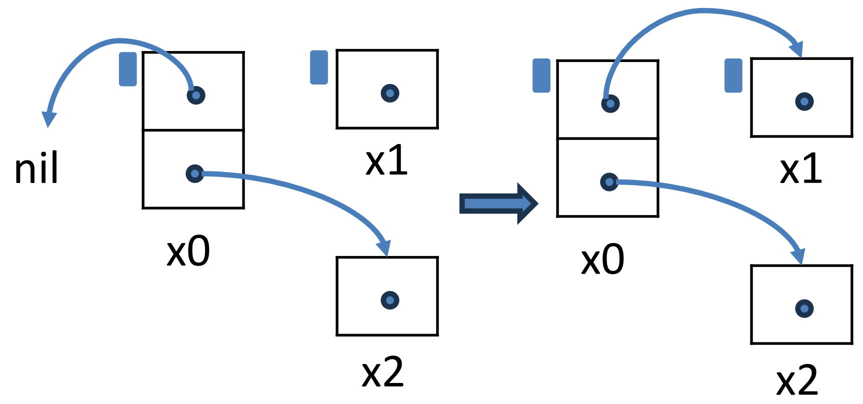
i = done[x1] = 0

y = x1.f0 = not pointer

done[x1] = i+1 = 1

Example: Pointer Reversal

function DFS(x)
  if x is a pointer and record x is not marked
    $t \leftarrow$ nil
    mark x; done[x] $\leftarrow$ 0
    while true
        $i \leftarrow$ done[x]
        if $i <$ # of fields in record x
          $y \leftarrow x.fi$
          if y is a pointer and record y is not marked
            $x.fi \leftarrow t$; $t \leftarrow x$; $x \leftarrow y$
            mark x; done[x] $\leftarrow$ 0
          else
            done[x] $\leftarrow i + 1$
        else
          $y \leftarrow x$; $x \leftarrow t$
          if x = nil **then return**
          $i \leftarrow$ done[x]
          $t \leftarrow x.fi$; $x.fi \leftarrow y$
          done[x] $\leftarrow i + 1$



i = done[x1] = 0

y = x1.f0 = not pointer

done[x1] = i+1 = 1

i = done[x1] = 1

y = x1, x = x0 // back to parent (x0)

i = done[x0] = 0

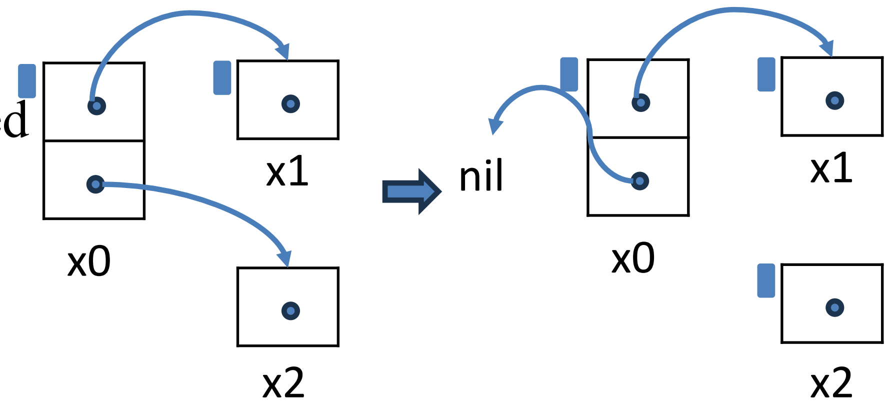
t = x0.f0 = nil // update stack top

x0.f0 = x1 // restore

done[x0] = 1

Example: Pointer Reversal

function DFS(x)
  if x is a pointer and record x is not marked
    $t \leftarrow$ nil
    mark x; done[x] $\leftarrow$ 0
    while true
        $i \leftarrow$ done[x]
        if $i <$ # of fields in record x
          $y \leftarrow x.\,fi$
          if y is a pointer and record y is not marked
            $x.\,fi \leftarrow t$; $t \leftarrow x$; $x \leftarrow y$
            mark x; done[x] $\leftarrow$ 0
          else
            done[x] $\leftarrow i + 1$
        else
          $y \leftarrow x$; $x \leftarrow t$
          if x = nil **then return**
          $i \leftarrow$ done[x]
          $t \leftarrow x.\,fi$; $x.\,fi \leftarrow y$
          done[x] $\leftarrow i + 1$



t = nil; x = x0

done[x0] = 1

i = done[x0] = 1

y = x0.f1 = x2

x0.f1 = t = nil

t = x = x0

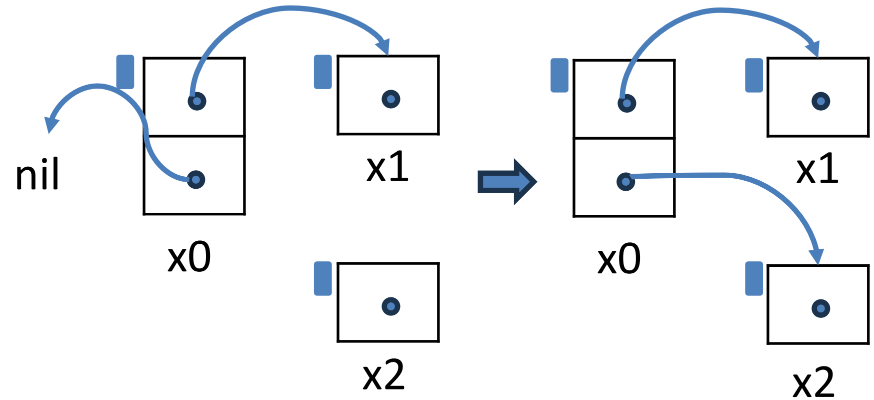
x = x2

mark x2

done[x2] = 0

Example: Pointer Reversal

function DFS(x)
  if x is a pointer and record x is not marked
    $t \leftarrow$ nil
    mark x; done[x] $\leftarrow$ 0
    while true
        $i \leftarrow$ done[x]
        if $i <$ # of fields in record x
          $y \leftarrow x.\,fi$
          if y is a pointer and record y is not marked
            $x.\,fi \leftarrow t$; $t \leftarrow x$; $x \leftarrow y$
            mark x; done[x] $\leftarrow$ 0
          else
            done[x] $\leftarrow i + 1$
        else
          $y \leftarrow x$; $x \leftarrow t$
          if x = nil **then return**
          $i \leftarrow$ done[x]
          $t \leftarrow x.\,fi$; $x.\,fi \leftarrow y$
          done[x] $\leftarrow i + 1$



t = x0; x = x2

i = done[x2] = 0

y = x2.f0 = not a pointer

done[x2] = 1

i = done[x2] = 1

y = x = x2; x = t = x0

i = done[x0] = 1

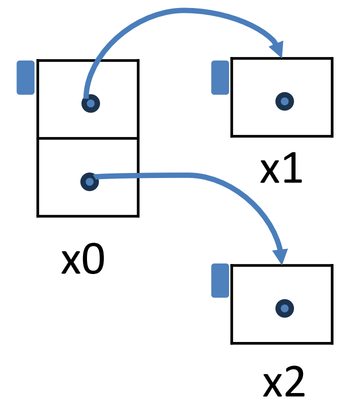
t = x0.f1 = nil

x0.f1 = y = x2 // restore

done[x0] = 2

Example: Pointer Reversal

function DFS(x)
  if x is a pointer and record x is not marked
    $t \leftarrow$ nil
    mark x; done[x] $\leftarrow$ 0
    while true
        $i \leftarrow$ done[x]
        if $i <$ # of fields in record x
          $y \leftarrow x.\,fi$
          if y is a pointer and record y is not marked
            $x.\,fi \leftarrow t$; $t \leftarrow x$; $x \leftarrow y$
            mark x; done[x] $\leftarrow$ 0
          else
            done[x] $\leftarrow i + 1$
        else
          $y \leftarrow x$; $x \leftarrow t$
          if $x =$ nil **then return**
          $i \leftarrow$ done[x]
          $t \leftarrow x.\,fi$; $x.\,fi \leftarrow y$
          done[x] $\leftarrow i + 1$



t = nil

x = x0

i = done[x0] = 2

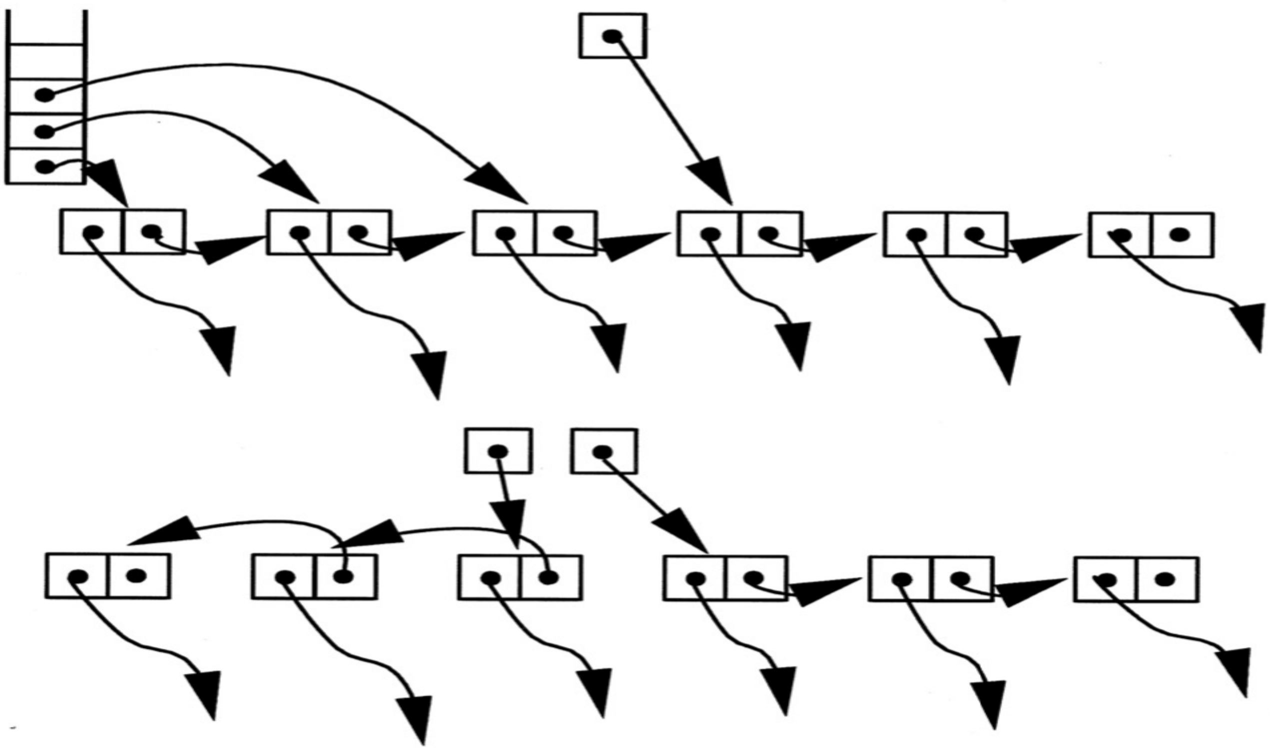
y = x = x0

x = t = nil

return

Pointer Reversal

- **Problem**: Depth-first search needs a stack
  - Stack depth could be as big as the graph
- **Solution**: Chain the stack inside the graph!



Summary: Mark-and-Sweep

- **Pros**
  - High efficiency if little garbage exist.
  - Be able to collect cyclic references.
  - Objects/records are not moved during GC
- **Cons**
  - Low efficiency with large amount of garbage
  - Normal execution must be suspended
  - Leads to fragmentation in the heap
    - Cache misses, page thrashing; more complex allocation

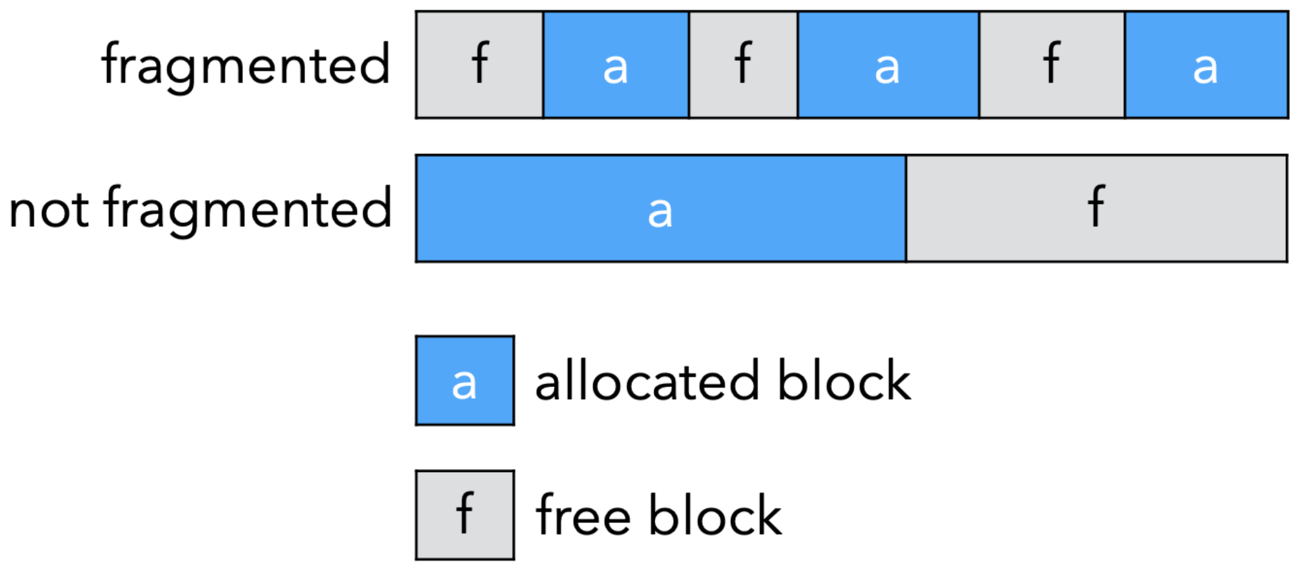
About the Fragmentation Problem

- External fragmentation
  - The program wants to allocate a record of size n, and there are many free records smaller than n but none of the right size

- Internal fragmentation
  - The program uses a too-large record without splitting it, so that the unused memory is inside the record instead of outside

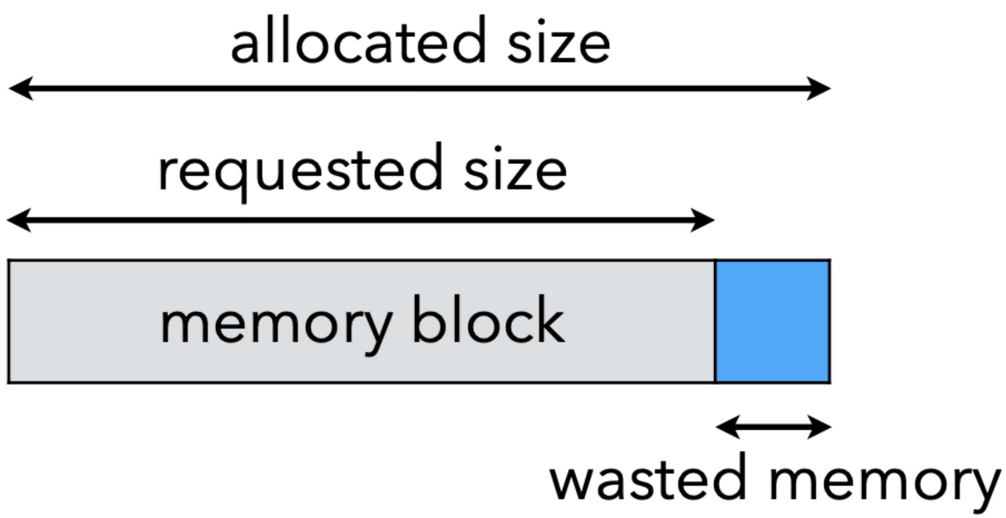
Example: External Fragmentation

- The two heaps below have the same amount of free memory, but the first suffers from **external fragmentation** while the second does not.
- Therefore, some requests can be fulfilled by the second but not by the first



Example: Internal Fragmentation

- The memory manager sometimes allocates more memory than requested, e.g. to satisfy alignment constraints.
- This results in small amounts of wasted memory scattered in the heap, and is called **internal fragmentation.**



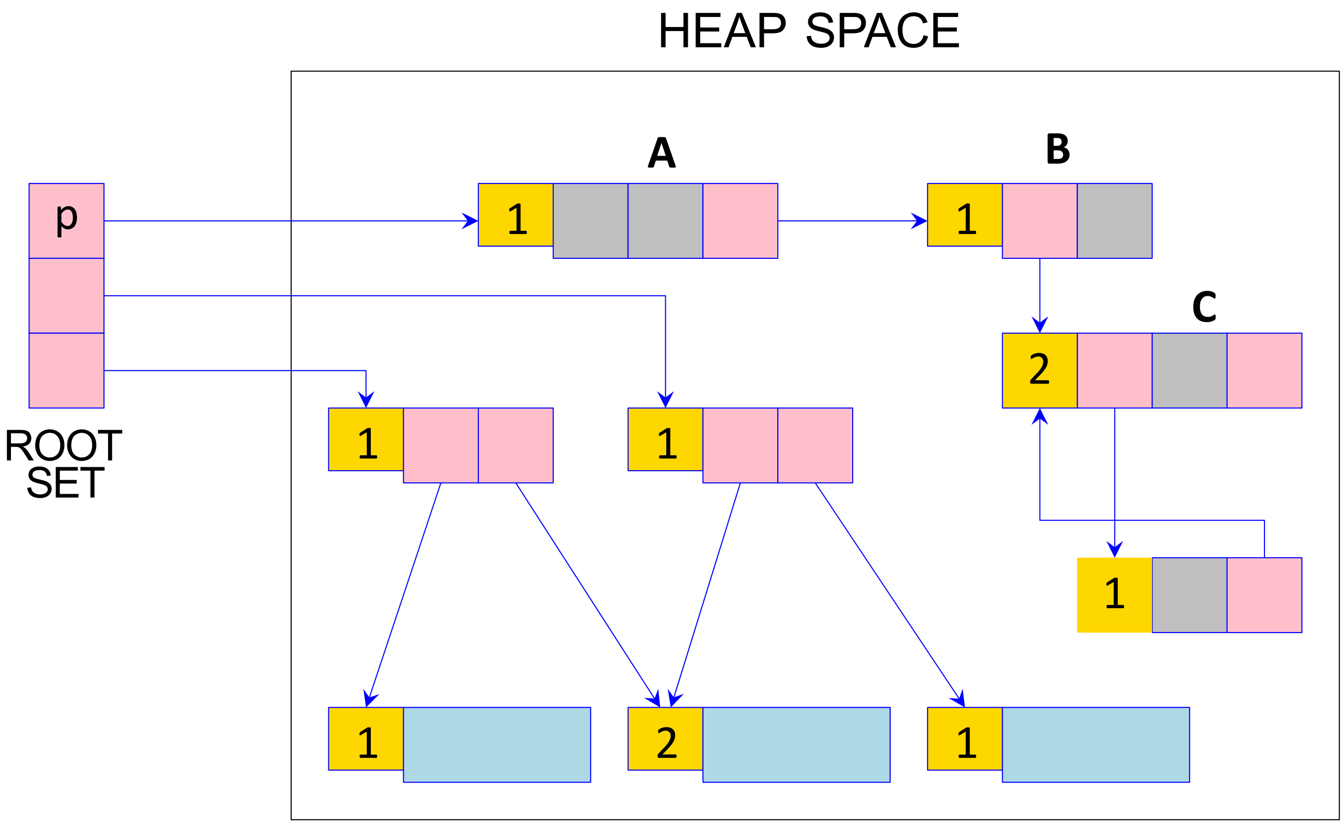
2. Reference Counting

Reference Counting

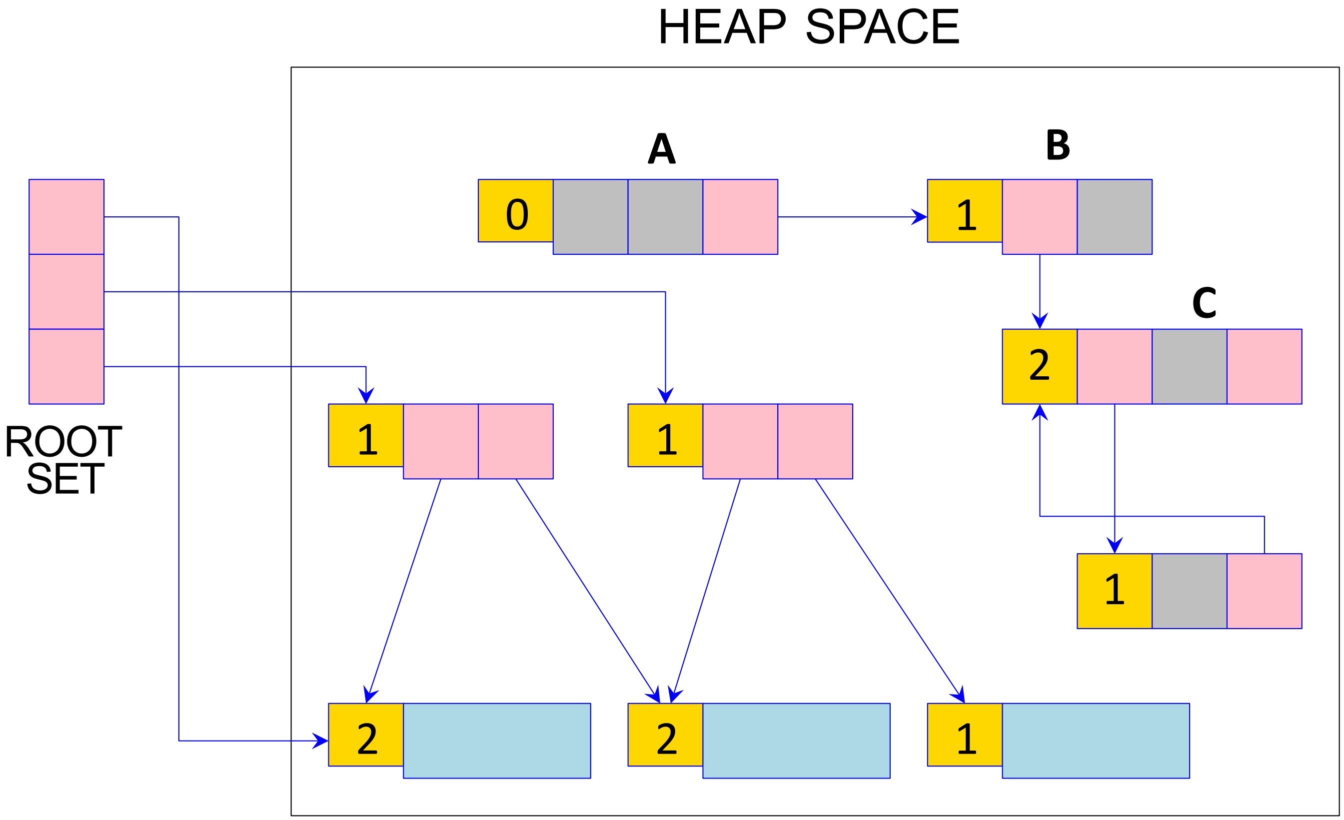
- **Idea**: rather than wait for memory to be exhausted, try to collect an object when there are no more pointers pointing to it (not reachable)
  - Keep track of the number of pointers to each object (the reference count)
  - Whenever a new reference to the data structure is established, increment the reference count
  - When the reference count goes to 0, the object is unreachable garbage

reference_count(x) = 0 ➔ x is unreachable

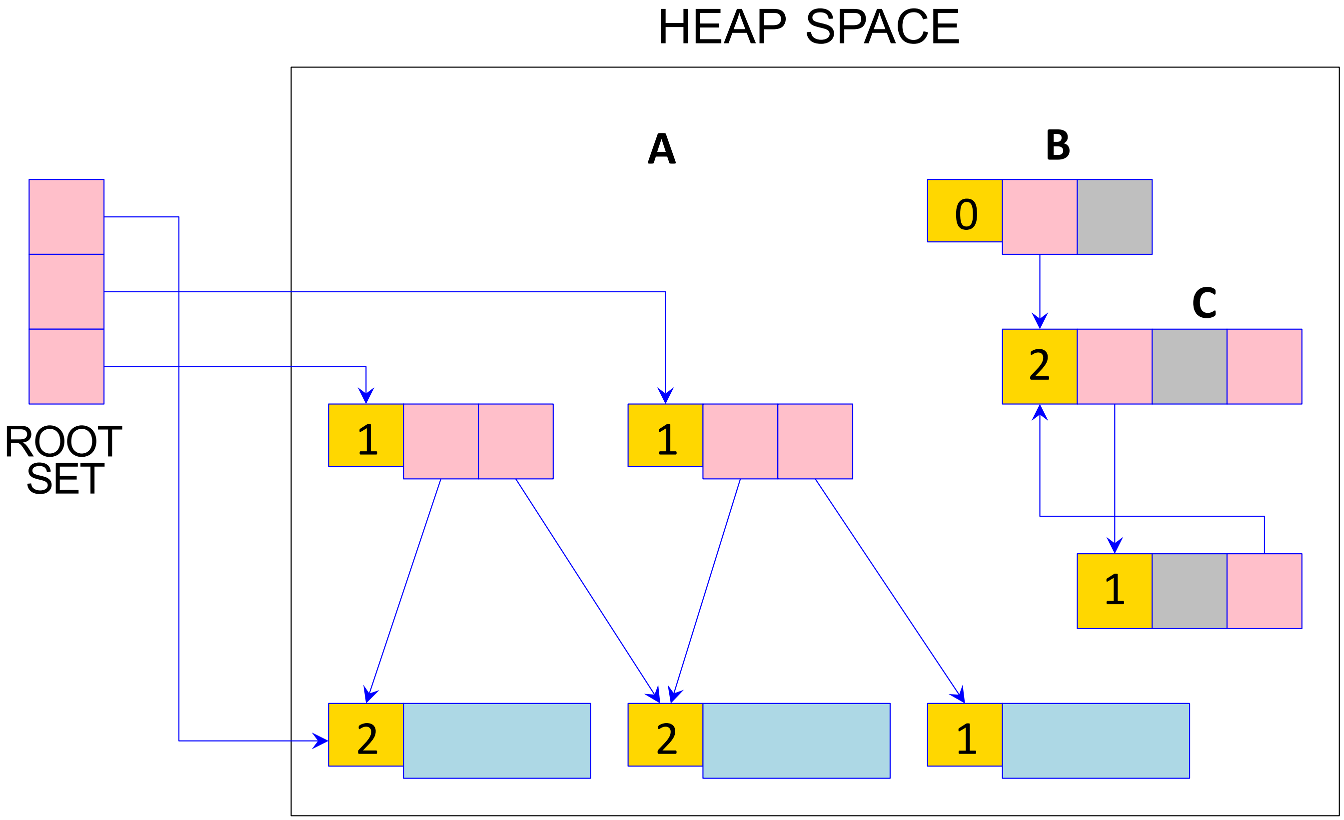
Example: Reference Counting



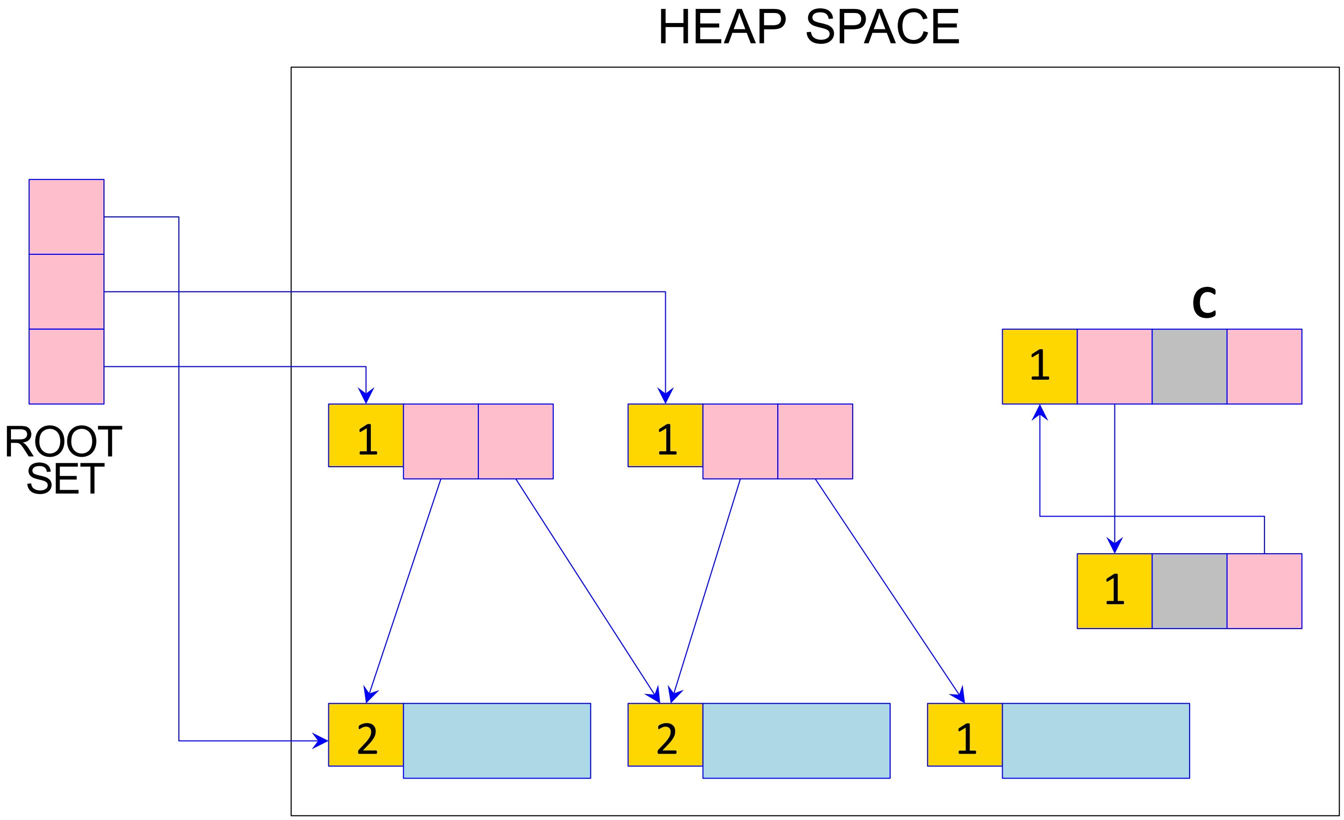
Example: Reference Counting



Example: Reference Counting



Example: Reference Counting



Reference Counting

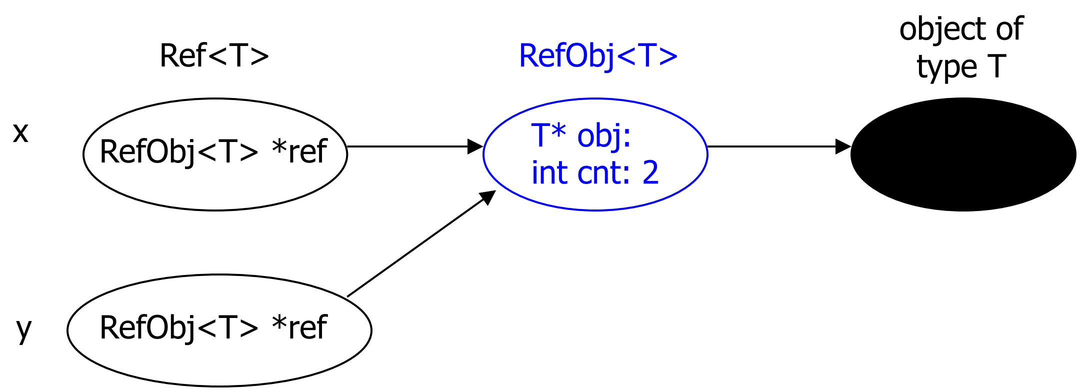
- How to keep track? Each assignment operation manipulates the reference counts.
- **Whenever p is stored into x.fi (i.e., x.fi = p)**
  - The reference count of p is incremented, and
  - The reference count of what x.fi previously pointed to is decremented.

Reference Counting

- How to keep track? Each assignment operation manipulates the reference counts.
- **Whenever p is stored into x.fi (i.e., x.fi = p)**
  - The reference count of p is incremented, and
  - The reference count of what x.fi previously pointed to is decremented.
- **If the reference count of some record r reaches zero**
  - r is put on the *freelist*, and
  - all the other records that r points to have their reference counts decremented.

Example: “Smart Pointer” in C++

- Similar to std::auto_ptr<T> in ANSI C++



sizeof(RefObj<T>) = 8 bytes of overhead per reference-counted object

sizeof(Ref<T>) = 4 bytes

Fits in a register

Easily passed by value as an argument or result of a function

Takes no more space than regular pointer, but much “safer” (why?)

Example: “Smart Pointer” in C++

```
template<class T> class Ref {
     RefObj<T>* ref;
     Ref<T>* operator&() {}
public:
     Ref() : ref(0) {}
     Ref(T* p) : ref(new RefObj<T>(p)) { ref->inc();}
     Ref(const Ref<T>& r) : ref(r.ref) { ref->inc(); }
     ~Ref() { if (ref->dec() == 0) delete ref; }

     Ref<T>& operator=(const Ref<T>& that) {
        if (this != &that) {
           if (ref->dec() == 0) delete ref;
               ref = that.ref;
               ref->inc(); }
         return *this; }
     T* operator->() { return *ref; }
     T& operator*() { return *ref; }
};
```

```
template<class T> class RefObj {
      T* obj;
      int cnt;
public:
     RefObj(T* t) : obj(t), cnt(0) {}
     ~RefObj() { delete obj; }

     int inc() { return ++cnt; }
     int dec() { return --cnt; }

     operator T*() { return obj; }
     operator T&() { return *obj; }
     T& operator *() { return *obj; }
};
```

Reference Counting: Strengths

- Incremental overhead
  - Cell management interleaved with program execution
  - So, no “stop-and-collection” effect
- 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: Problems

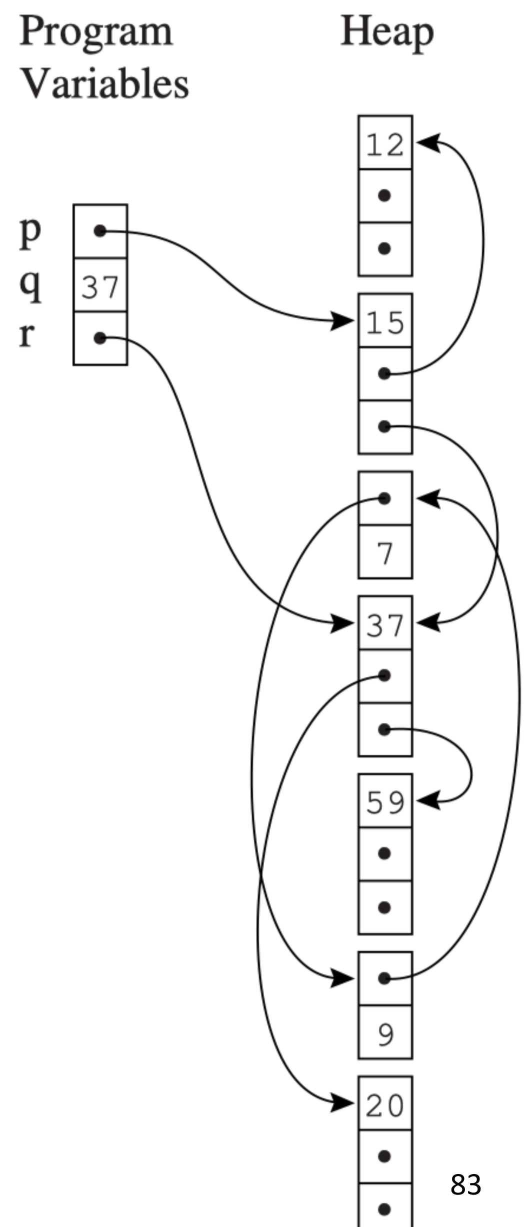
- Reference counting seems simple and attractive.
- But there are two major problems:

1. Cycles of garbage cannot be claimed
2. Incrementing the reference counts is very expensive

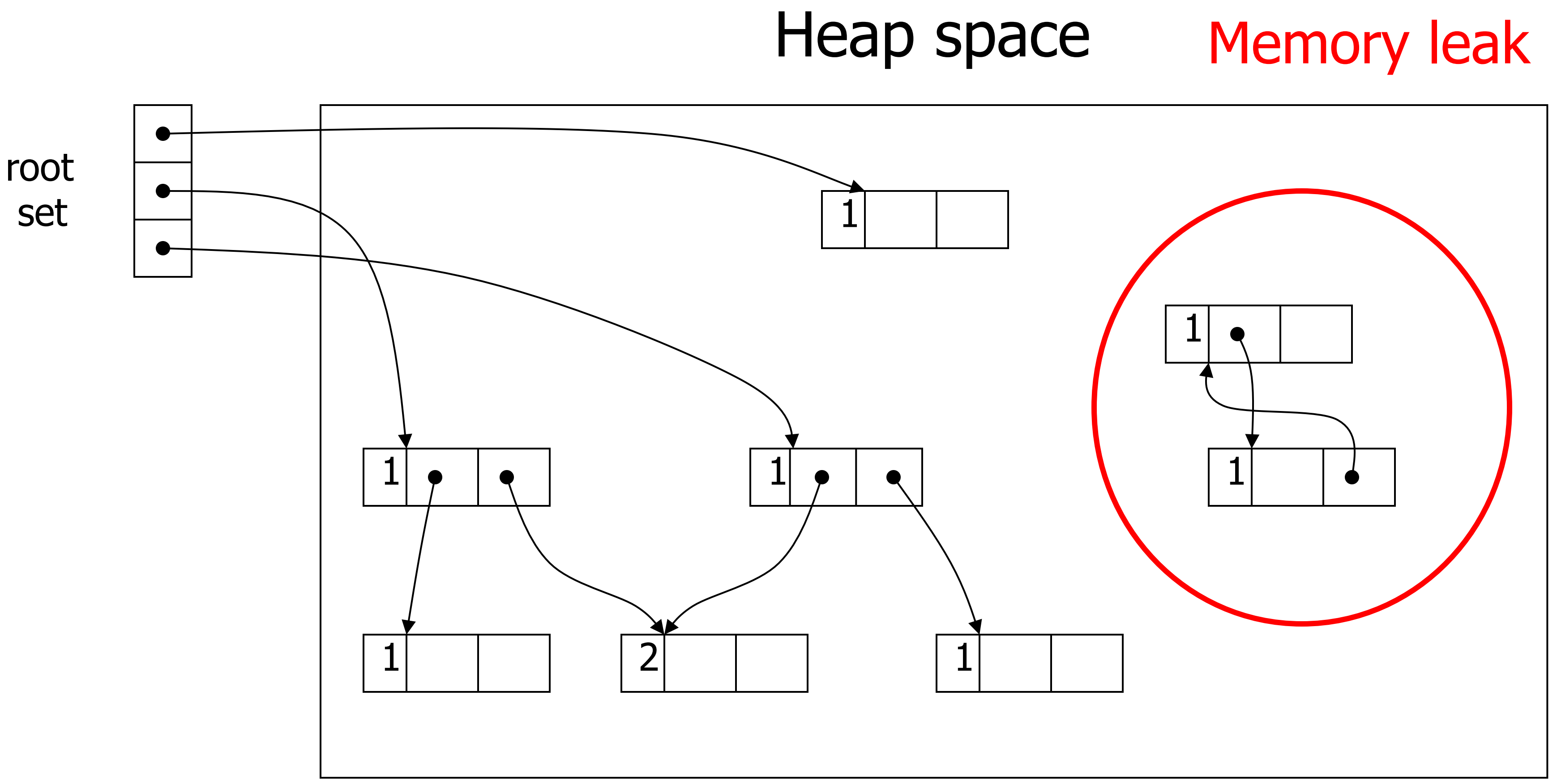
Problem 1: Reference Cycle

- A reference cycle is a set of objects that cyclically refer to one another.
  - Example: the record storing 7 and the record storing 9

- Reference count tracks the number of references, not number of reachable references.



Example: Reference Counting: Cycles



Problem 2: Too Many Emitted Codes (x.fi ← p)

- Incrementing the reference counts is very expensive.
- In place of the single machine instrution $x.\,fi \leftarrow p$, the program must execute:

$$
\begin{array}{ll}
z & \leftarrow x.f_i \\
c & \leftarrow z.\text{count} \\
c & \leftarrow c - 1 \\
z.\text{count} & \leftarrow c \\
\multicolumn{2}{l}{\text{if } c = 0 \text{ call } \mathit{putOnFreelist}} \\
x.f_i & \leftarrow p \\
c & \leftarrow p.\text{count} \\
c & \leftarrow c + 1 \\
p.\text{count} & \leftarrow c
\end{array}
$$

// The ref. count of what x.fi previously pointed to is decremented

// The reference count of p is incremented

Dataflow analysis can eliminate some increments and decrements, but many remain

Summary: Reference Counting

- **Advantages**
  - Immediate collection: (i.e., reduce the time between the object becoming garbage and its reclamation)
  - Incremental collection
  - Simple to implement
- **Disadvantages**
  - Can not collect unreachable cyclic data structures
  - Relatively inefficient

3. Copying Collection

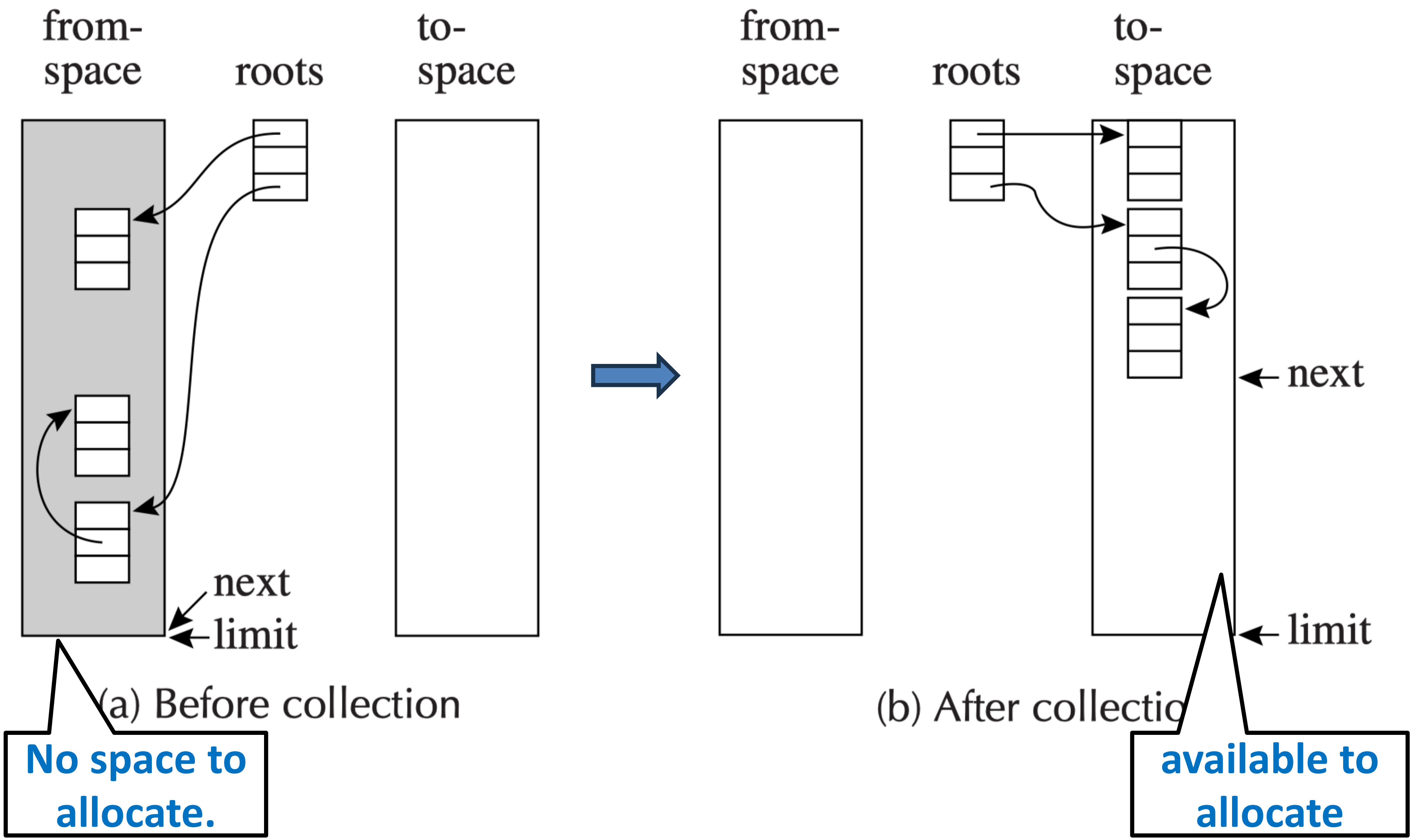
- Overview
- Pointer Forwarding
- Cheney's Algorithm

Copying Collection

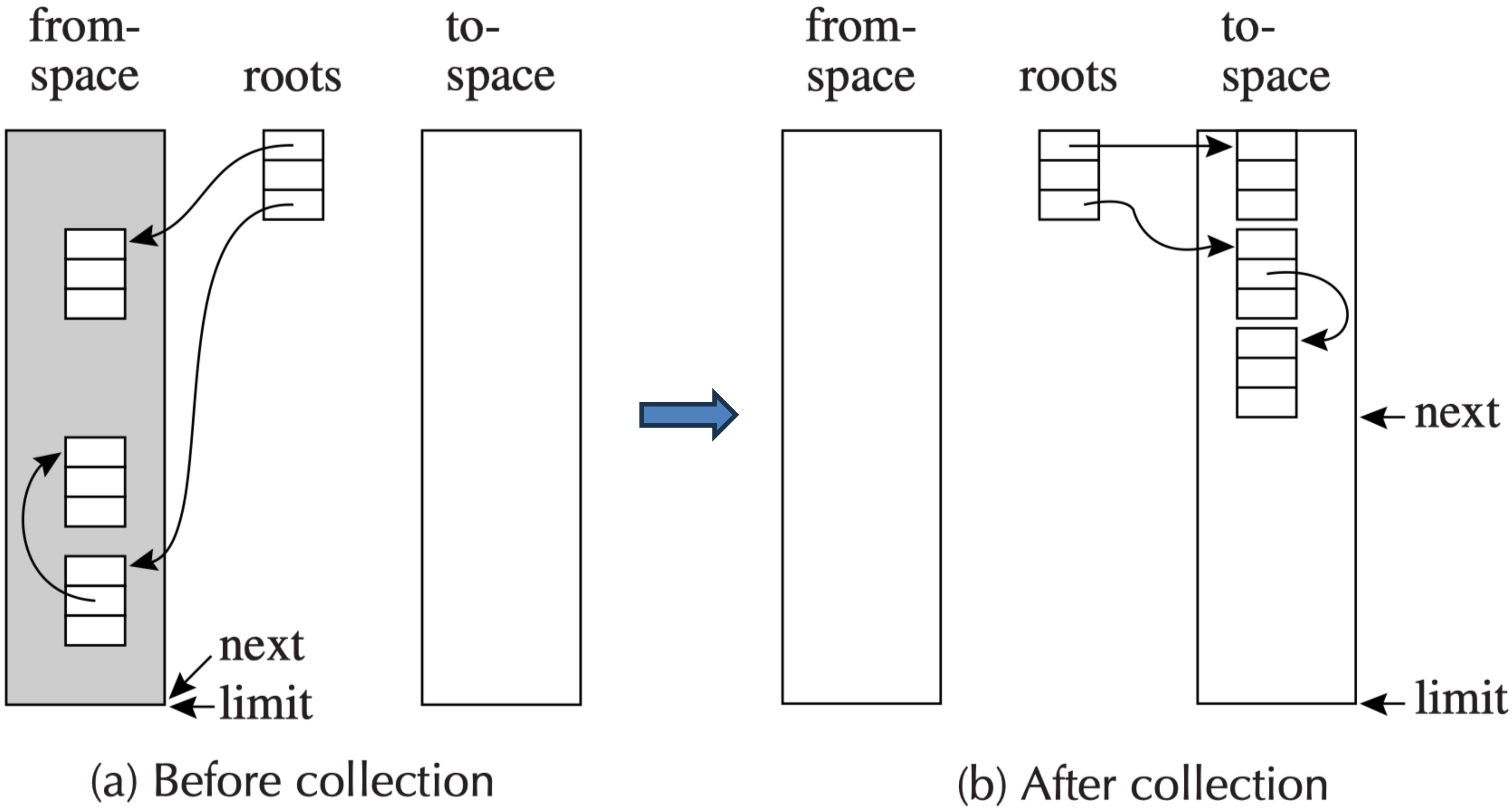
- **Basic idea: use 2 heaps**
  - from-space: the one used by program
  - to-space: the one unused until GC time

- **Garbage collection:**
  - When from-space is exhausted, traverse the from-space, and copy all **reachable nodes** to to-space
    - Garbage is left behind
  - When next reaches the limit, change the role of from-space and to-space

Example: Copying Collection



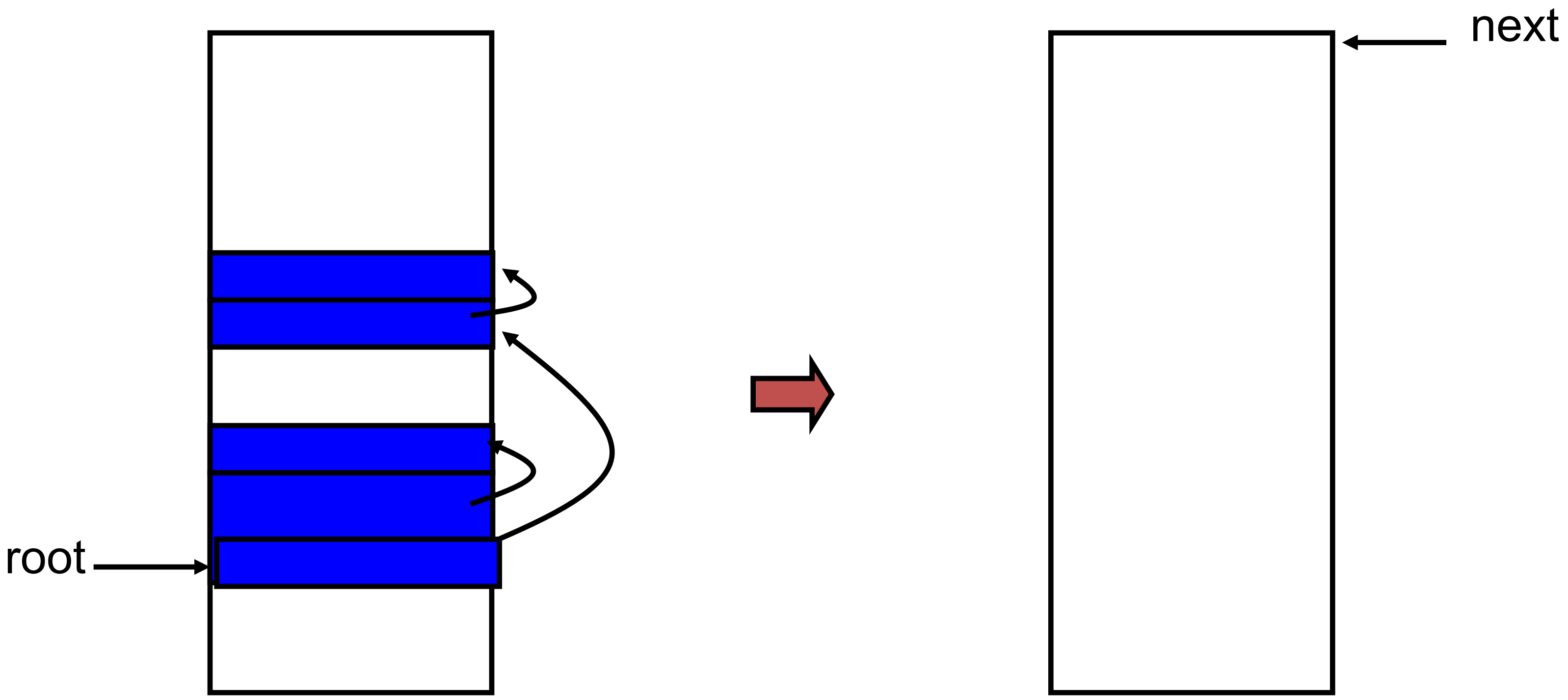
Example: Copying Collection



- Allocating a new record in the to-space is easy: p=next, next=next + n.
- All the reachable nodes are around the beginning of the to-space
（Does not have a fragmentation problem)

Initiating a Collection

- The pointer next is initialized to point at the beginning of the to-space
- As each reachable record in from-space is found, it is copied to to-space at position next, and next incremented by the size of the record.



3. Copying Collection

- Overview
- Pointer Forwarding
- Cheney’s Algorithm

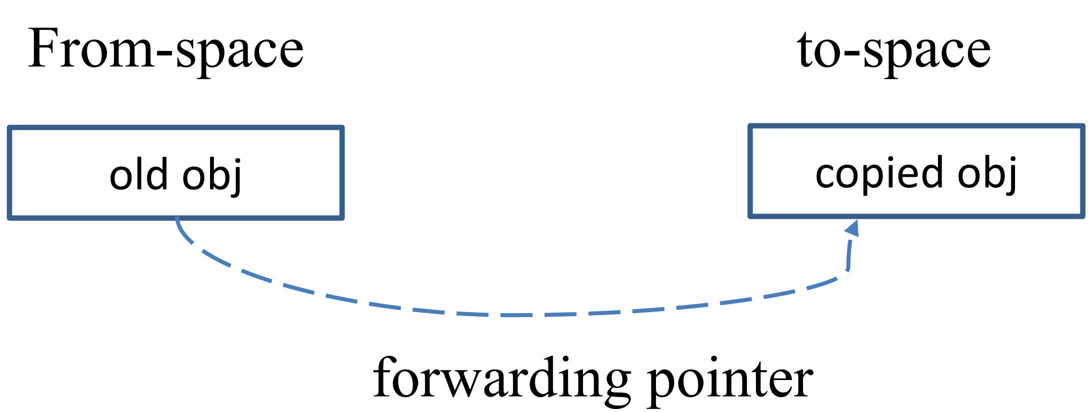
Pointer Forwarding: Why?

- We need to find all the reachable records, as for the mark-and-sweep approach
- As we find a reachable record, we copy it into the new space (to-space)

- **Besides,** we have to fix **ALL** pointers pointing to it (preserve the points-to relations)
- But how can we know such information when performing the DFS/BFS (in the runtime)?!

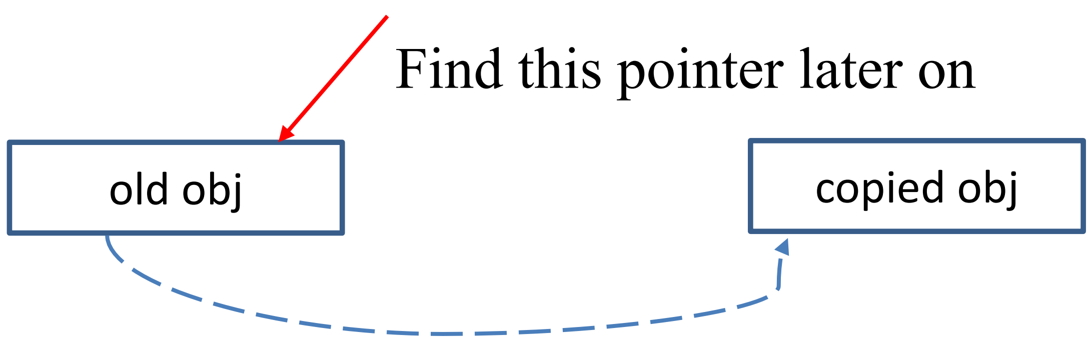
Pointer Forwarding: Insight

- As we find a reachable record, we copy it into the new space (to-space)
  - **Problem**: We have to fix ALL pointers pointing to it
- As we copy a record, we store in the old copy (from-space) a forwarding pointer to the new copy
  - When we later reach a record with a forwarding pointer, we know it was already copied



Pointer Forwarding: Insight

- As we find a reachable record, we copy it into the new space (to-space)
  - **Problem**: We have to fix ALL pointers pointing to it
- As we copy a record, we store in the old copy (from-space) a forwarding pointer to the new copy
  - When we later reach a record with a forwarding pointer, we know it was already copied



Use the forwarding pointer
to find where the new record is!

Pointer Forwarding

- **Pointer forwarding:** Given a pointer p that points to from-space, make p point to to-space

```
function Forward(p)
    if p points to from-space
    then if p. f1 points to to-space
        then return p. f1
        else for each field fi of p
                next. fi ← p. fi
            p. f1 ← next
            next ← next + size of record p
            return p. f1
    else return p
```

1. p points to a from-space record that has already been copied: **p.f1 is a *forwarding pointer*** that indicates where the copy is

Pointer Forwarding

- **Pointer forwarding:** Given a pointer p that points to from-space, make p point to to-space

```
function Forward(p)
    if p points to from-space
    then if p. f1 points to to-space
        then return p. f1
        else for each field fi of p
                next. fi ← p. fi
            p. f1 ← next // forwarding ptr
            next ← next + size of record p
            return p. f1
    else return p
```

1. p points to a from-space record that has already been copied: **p.f1 is a *forwarding pointer*** that indicates where the copy is

2. p points to a from-space record that has not yet been copied

此时写from-space中原来那个record的域f1是合法的，因为所有数据都已经复制到了to-space！

Pointer Forwarding

- **Pointer forwarding:** Given a pointer p that points to from-space, make p point to to-space

```
function Forward(p)
    if p points to from-space
    then if p. f1 points to to-space
        then return p. f1
        else for each field fi of p
                next. fi ← p. fi
            p. f1 ← next
            next ← next + size of record p
            return p. f1
    else return p
```

1. p points to a from-space record that has already been copied: **p.f1 is a *forwarding pointer*** that indicates where the copy is

2. p points to a from-space record that has not yet been copied

3. p is not a pointer or points outside *from-space*

3. Copying Collection

- Overview
- Pointer Forwarding
- Cheney’s Algorithm

Cheney's Algorithm

- Partition the to-space in three contiguous regions
  - Copied: the record is copied, but we haven't yet looked at pointers inside the record
  - Copied and scanned: the record is copied and we have proposed all pointers in the record

Copied records whose pointer fields were followed

Copied records whose pointer fields have NOT been followed yet

Cheney's Algorithm

- Partition the to-space in three contiguous regions
  - Copied: the record is copied, but we haven't yet looked at pointers inside the record
  - Copied and scanned: the record is copied and we have proposed all pointers in the record

Copied records whose pointer fields were followed

Copied records whose pointer fields have NOT been followed yet

The worklist (breadth-first search queue!)

Cheney's Algorithm

- **Cheney's algorithm**: using breadth-first search to traverse the reachable data, copying from from-space to to-space

scan ← *next* ← beginning of to-space
for each root r
    r ← Forward(r)
while *scan* < *next*
    for each field *fi* of record at *scan*
        scan.fi ← Forward(*scan.fi*)
    scan ← *scan* + size of record at *scan*

ALGORITHM 13.9: Breadth-first copying garbage collection

Cheney's Algorithm

- **Cheney's algorithm**: using breadth-first search to traverse the reachable data, copying from from-space to to-space

> *scan* ← *next* ← beginning of to-space
> **for** each root r
>   r ← Forward(r)
> **while** *scan* < *next*
>   **for** each field *fi* of record at *scan*
>   *scan.fi* ← Forward(*scan.fi*)
>   *scan* ← *scan* + size of record at *scan*

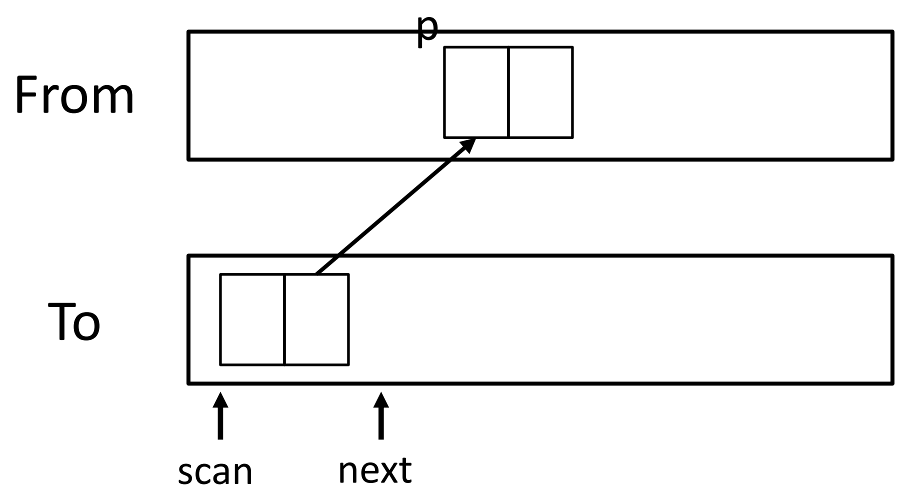
ALGORITHM 13.9: Breadth-first copying garbage collection

- The BFS queue: area between scan and next
- When scan catches up with next, then done!

Example: Cheney's Algorithm

```
function Forward(p)
    if p points to from-space
    then if p.f1 points to to-space
        then return p.f1
        else for each field fi of p
                next.fi ← p.fi
             p.f1 ← next
             next ← next + size of record p
             return p.f1
    else return p
```

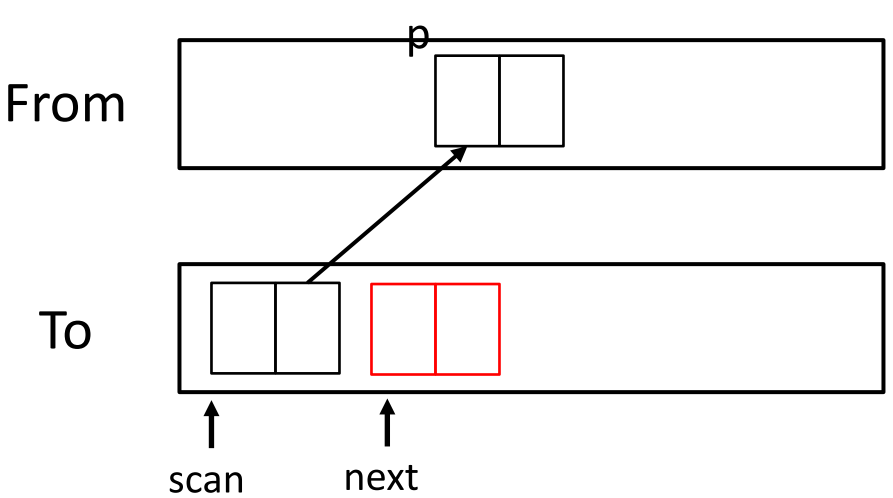
```
scan ← next ← beginning of to-space
for each root r
    r ← Forward(r)
while scan < next
    for each field f_i at scan
        scan.f_i ← Forward(scan.f_i)
    scan ← scan + size of record at scan
```



Example: Cheney's Algorithm

function Forward(p)
  if p points to from-space
  then if p. f1 points to to-space
    then return p. f1
    else for each field fi of p
      next. fi ← p. fi
    p. f1 ← next
    next ← next + size of record p
    return p. f1
  else return p

scan ← next ← beginning of to-space
for each root r
  r ← Forward(r)
while scan < next
  for each field f_i at scan
    scan.f_i ← Forward(scan.f_i)
  scan ← scan + size of record at scan

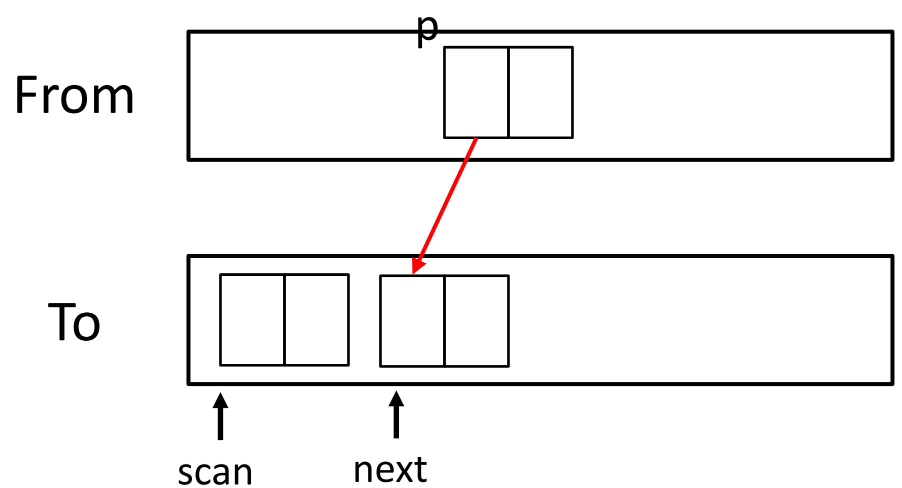


Example: Cheney's Algorithm

function Forward(p)
  if p points to from-space
  then if p. f1 points to to-space
    then return p. f1
    else for each field fi of p
        next. fi ← p. fi
      p. f1 ← next
      next ← next + size of record p
      return p. f1
  else return p

scan ← next ← beginning of to-space
for each root r
  r ← Forward(r)
while scan < next
  for each field f_i at scan
    scan.f_i ← Forward(scan.f_i)
  scan ← scan + size of record at scan

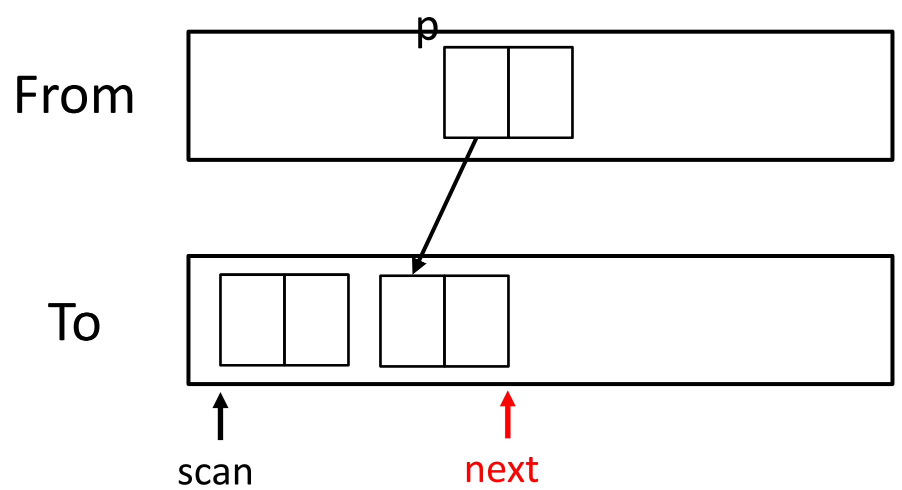
p.f1:
the forwarding pointer!



Example: Cheney's Algorithm

```
function Forward(p)
    if p points to from-space
    then if p. f1 points to to-space
        then return p. f1
        else for each field fi of p
                next. fi ← p. fi
            p. f1 ← next
            next ← next + size of record p
            return p. f1
    else return p
```

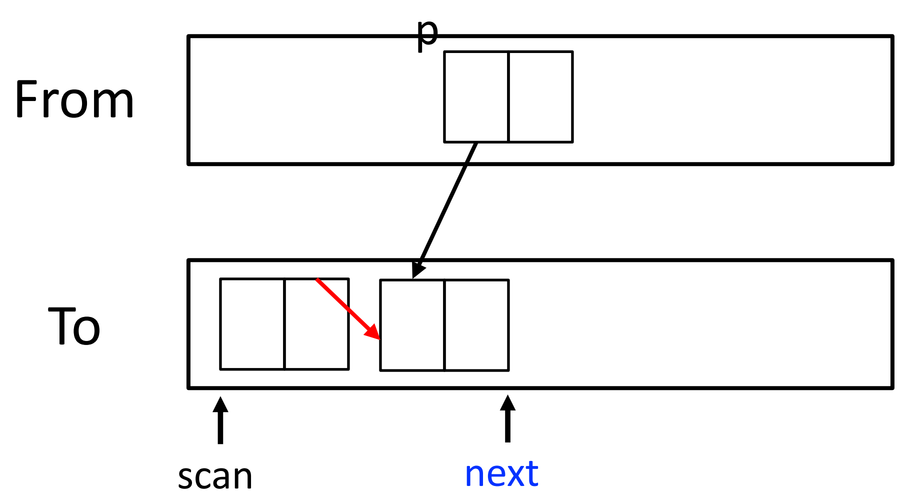
```
scan ← next ← beginning of to-space
for each root r
    r ← Forward(r)
while scan < next
    for each field f_i at scan
        scan.f_i ← Forward(scan.f_i)
    scan ← scan + size of record at scan
```



Example: Cheney's Algorithm

function Forward(p)
    if p points to from-space
    then if p. f1 points to to-space
        then return p. f1
        else for each field fi of p
                next. fi ← p. fi
            p. f1 ← next
            next ← next + size of record p
            return p. f1
    else return p

scan ← next ← beginning of to-space
for each root r
    r ← Forward(r)
while scan < next
    for each field f_i at scan
        scan.f_i ← Forward(scan.f_i)
    scan ← scan + size of record at scan

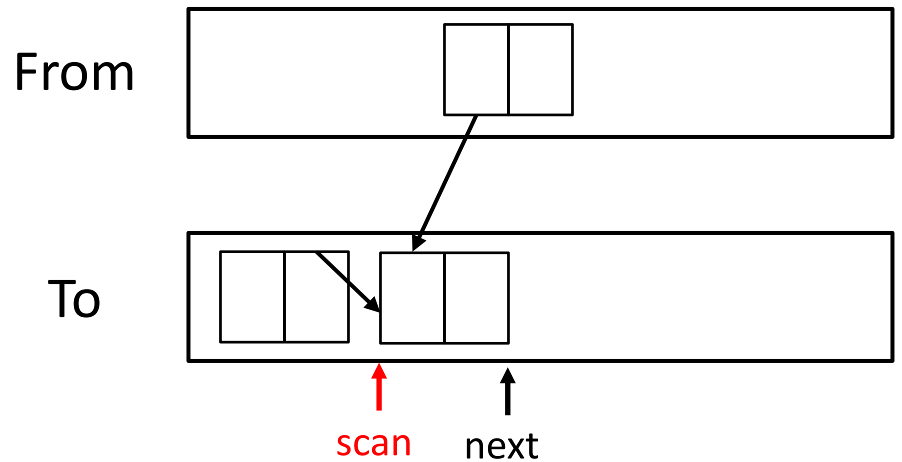


Example: Cheney's Algorithm

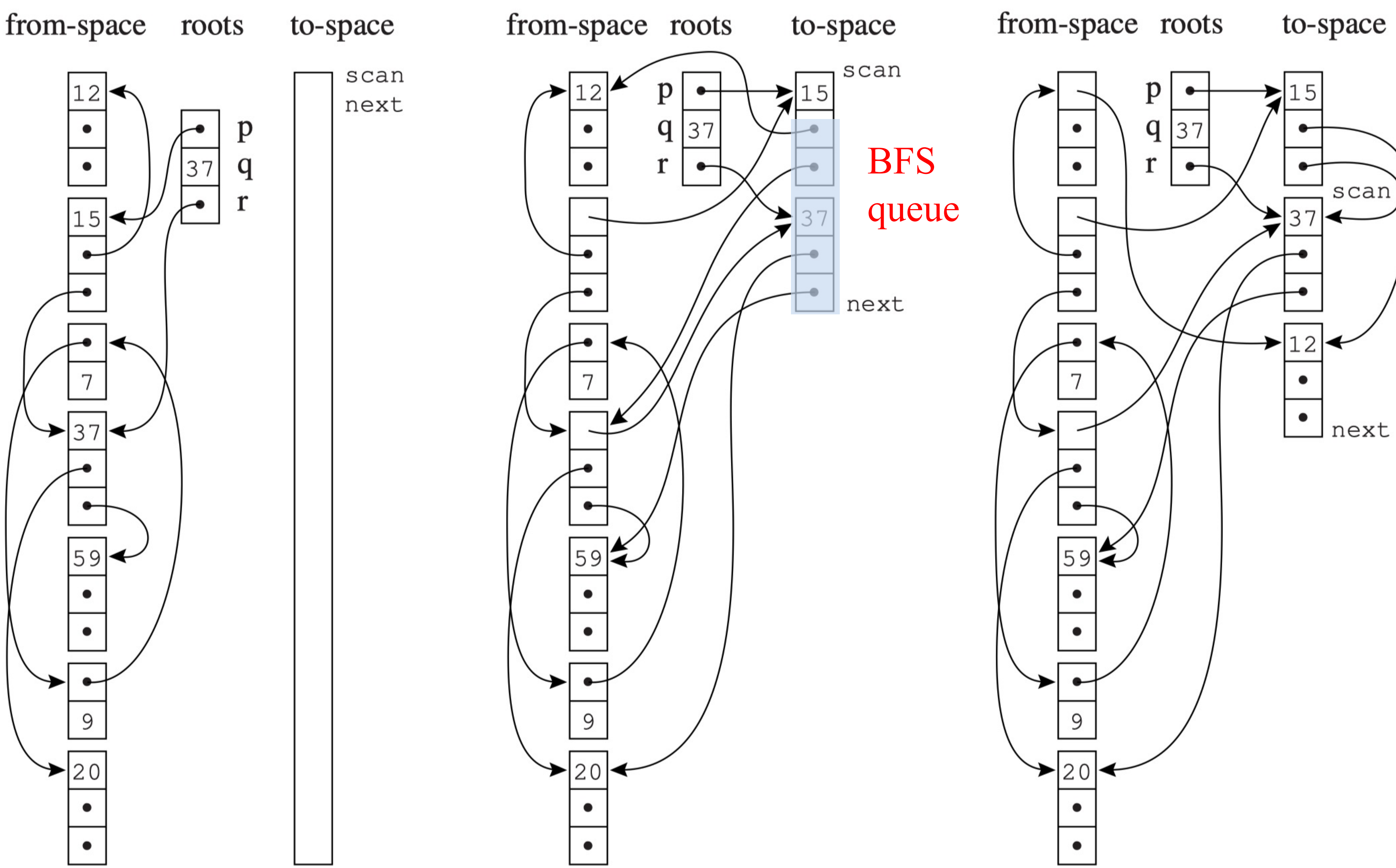
```
function Forward(p)
    if p points to from-space
    then if p. f1 points to to-space
        then return p. f1
        else for each field fi of p
                next. fi ← p. fi
            p. f1 ← next
            next ← next + size of record p
            return p. f1
    else return p
```

```
scan ← next ← beginning of to-space
for each root r
    r ← Forward(r)
while scan < next
    for each field f_i at scan
        scan.f_i ← Forward(scan.f_i)
    scan ← scan + size of record at scan
```

Area between scan and next: the BFS queue!

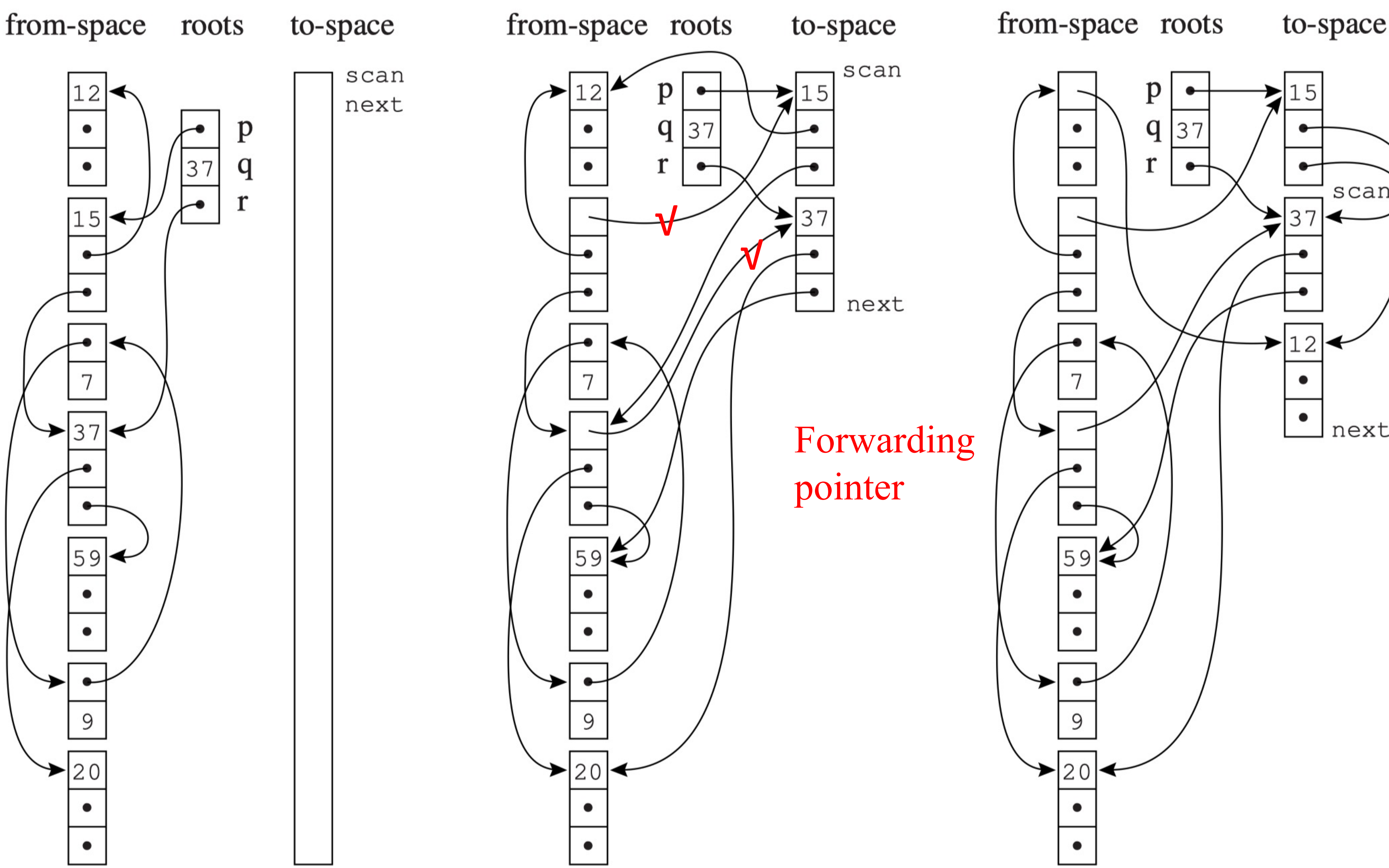


Example: Breadth-first Copying Collection



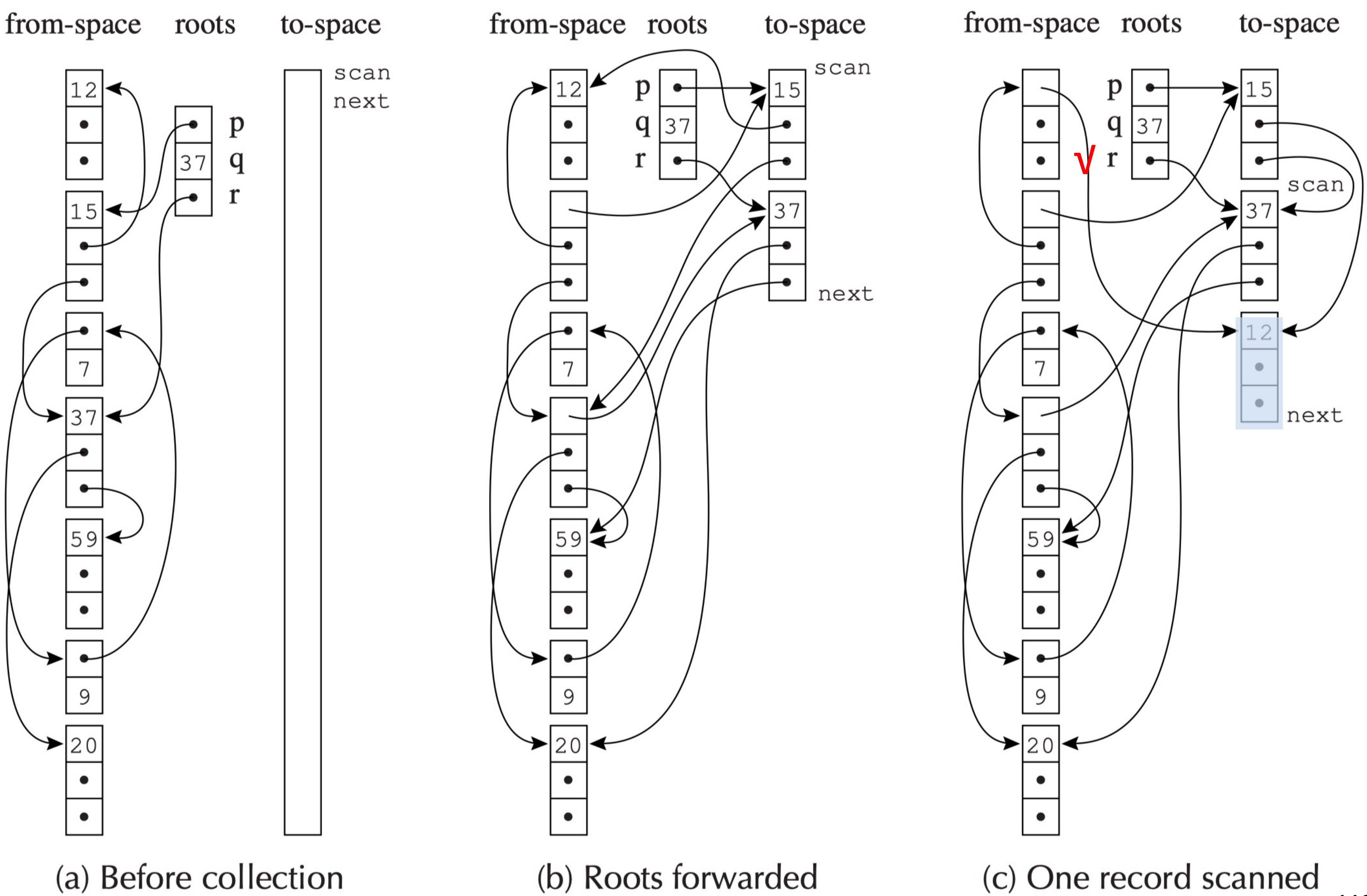
(a) Before collection (b) Roots forwarded (c) One record scanned

Example: Breadth-first Copying Collection



(a) Before collection (b) Roots forwarded (c) One record scanned

Example: Breadth-first Copying Collection



Limitation of Cheney's Algorithm: Locality of Reference

- In a computer system with virtual memory, or with a memory cache, good locality of reference is important :
  - After the program fetches address a, then the memory subsystem expects addresses near a to be fetched soon.
- Pointer data structures copied by breadth-first have poor locality of reference.
  - If a record at address *a* points to another record at address *b*, it is likely that *a* and *b* will be far apart.
- Depth-first copying gives better locality
  - But depth-first copy requires pointer-reversal, which is inconvenient and slow.

A Hybrid Algorithm

- A hybrid, partly depth-first and partly breadth-first algorithm can provide acceptable locality.

```
function Forward(p)
    if p points to from-space
    then if p.f1 points to to-space
            then return p.f1
            else Chase(p); return p.f1
    else return p
```

- The basic idea is to use breadth-first copying, but whenever an object is copied, see if some child can be copied near it.

```
function Chase(p)
    repeat
        q ← next
        next ← next+ size of record p
        r ← nil
        for each field fi of record p
            q.fi ← p.fi
            if q.fi points to from-space and
q.fi.f1 does not point to to-space
            then r ← q.fi
        p.f1 ← q
        p←r
    until p = nil
```

Summary: Copying Collection

- **Advantage:**
  - Simplicity - no stack or pointer reversal required
  - Run-time proportional to # live objects
  - Leve free-space contiguous
    - Automatic compaction eliminates fragmentation
  - Form basis of many later algorithms

- **Disadvantage:**
  - Half of memory is wasted
  - Poor locality (at least the Cheney's algorithm)
  - Precise type information required (pointer or not)

4. Interface to the Compiler

- Fast Allocation
- Describing Data Layouts
- Describing Roots (Pointer Map)
- Derived Pointers

Interface to the Compiler

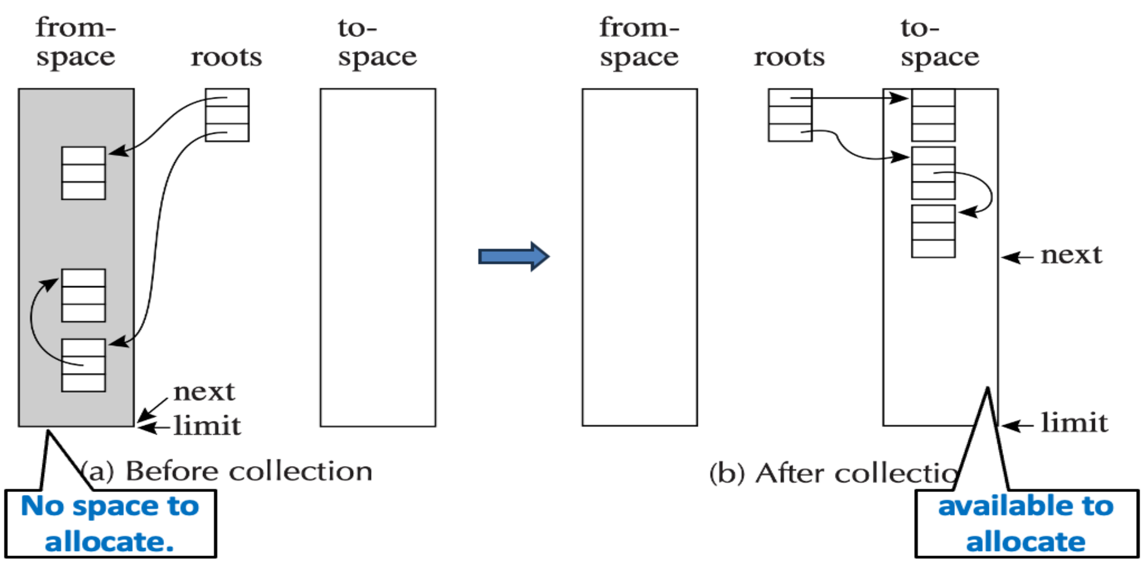
- Although the garbage collector is a part of “runtime”,
- The compiler for a garbage-collected language interacts with the garbage collector by:
  1. Generating code that allocate records
  2. Describing locations of roots for each garbage collection cycle
  3. Describing the layout of data records on the heap
  4. Generating instructions to implement a read or write barrier (for some versions of incremental collection)
  5. …

Allocation Matters for Programs

- Especially for:
  - Functional languages (updating is discouraged)
  - Memory-intensive applications (access once for each)
- Empirical measurements: one in seven instructions is a store!
  - We have at most 1/7 word of allocation per instruction

Fast Allocation (for Copying Collection)

- There is a considerable cost to create the heap records.
- Copying collection should be used (as it is fast)
  - The allocation space is contiguous free region
  - The next free location is next and
  - The end of the region is limit



Fast Allocation (for Copying Collection)

The steps to allocate a record of size N:

1. Call the allocate function
2. Test *next + N < limit* ? (If the test fails, call GC)
3. Move *next* to *result*
4. Clear *M[next]*, *M[next+1]*, …, *M[next + N - 1]*
5. *next <- next + N*
6. Return from the allocate function

A. Move result into some computationally useful place

B. Store useful values into the record

(Steps A and B are not allocation overhead)

Fast Allocation

Step 1 and 6 should be eliminated by *inline expanding* the allocate function

The steps to allocate a record of size N:

1. ~~Call the allocate function~~
2. Test $next + N < limit$? (If the test fails, call GC)
3. Move *next* to *result*
4. Clear $M[next]$, $M[next+1]$, …, $M[next + N - 1]$
5. $next \leftarrow next + N$
6. ~~Return from the allocate function~~

A. Move result into some computationally useful place

B. Store useful values into the record

(Steps A and B are not allocation overhead)

Fast Allocation

Step 3 can often be eliminated by combining it with Step A.

The steps to allocate a record of size N:

1. ~~Call the allocate function~~
2. Test *next* + *N* < *limit* ? (If the test fails, call GC)

A. Move *next* into some computationally useful place

4. Clear *M[next]*, *M[next+1]*, ..., *M[next + N - 1]*
5. *next <- next + N*
6. ~~Return from the allocate function~~

~~A. Move result into some computationally useful place~~

B. Store useful values into the record

(Steps A and B are not allocation overhead)

Fast Allocation

Step 4 can be eliminated by in factor of Step B.

The steps to allocate a record of size N:

1. ~~Call the allocate function~~
2. Test *next* + *N* < *limit* ? (If the test fails, call GC)

A. Move *next* into some computationally useful place

4. ~~Clear *M[next]*, *M[next+1]*, …, *M[next + N - 1]*~~
5. *next* <- *next* + *N*
6. ~~Return from the allocate function~~

A. ~~Move result into some computationally useful place~~

B. Store useful values into the record

(Steps A and B are not allocation overhead)

Fast Allocation

Step 2 and 5 cannot be eliminated. But they can be shared among multiple allocations.

The steps to allocate a record of size N

~~1. Call the allocate function~~

2. Test *next + N < limit* ? (If the test fails, call GC)

A. Move *next* into some computationally useful place

~~4. Clear *M[next]*, *M[next+1]*, …, *M[next + N - 1]*~~

5. *next <- next + N*

~~6. Return from the allocate function~~

~~A. Move result into some computationally useful place~~

B. Store useful values into the record

(Steps A and B are not allocation overhead)

Fast Allocation (for Copying Collection)

- Step 2: Test *next* + *N* < *limit* ? (If the test fails, call GC)
- Step 5: *next* <- *next* + *N*

- By keeping next and limit in registers, steps 2 and 5 can be done in a total of three instructions

- By this combination of techniques, the cost of allocating a record – and then eventually garbage collecting it - can be brought down to about four instructions

4. Interface to the Compiler

- Fast Allocation
- Describing Data Layouts
- Describing Roots (Pointer Map)
- Derived Pointers

Describing Data Layouts

- The collector needs to handle records of different types
  - Different length: used when adding *scan*

```
scan ← next ← beginning of to-space
for each root r
   r ← Forward(r)
while scan < next
   for each field f_i at scan
      scan.f_i ← Forward(scan.f_i)
   scan ← scan + size of record at scan
```

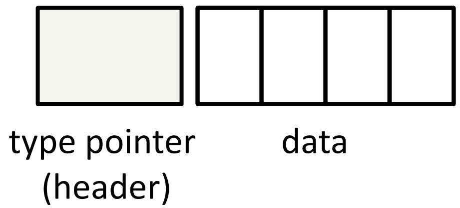
Describing Data Layouts

- The collector needs to handle records of different types
  - Different length: used when adding *scan*
  - Field type: used by *Forward*
    - Only pointers need to be processed

```
scan ← next ← beginning of to-space
for each root r
    r ← Forward(r)
while scan < next
    for each field f_i at scan
        scan.f_i ← Forward(scan.f_i)
    scan ← scan + size of record at scan
```

Describing Data Layouts

- For statically typed language, such as Tiger or Pascal, or for object-oriented languages, such as Java:
  - have the first word of every object point to a special type- or class-descriptor record.
- **Type- or class-descriptor**
  - The total size of the object
  - the location of each pointer field
- Type- or class-descriptor is generated by the compiler
  - In which phase? (semantic analysis)

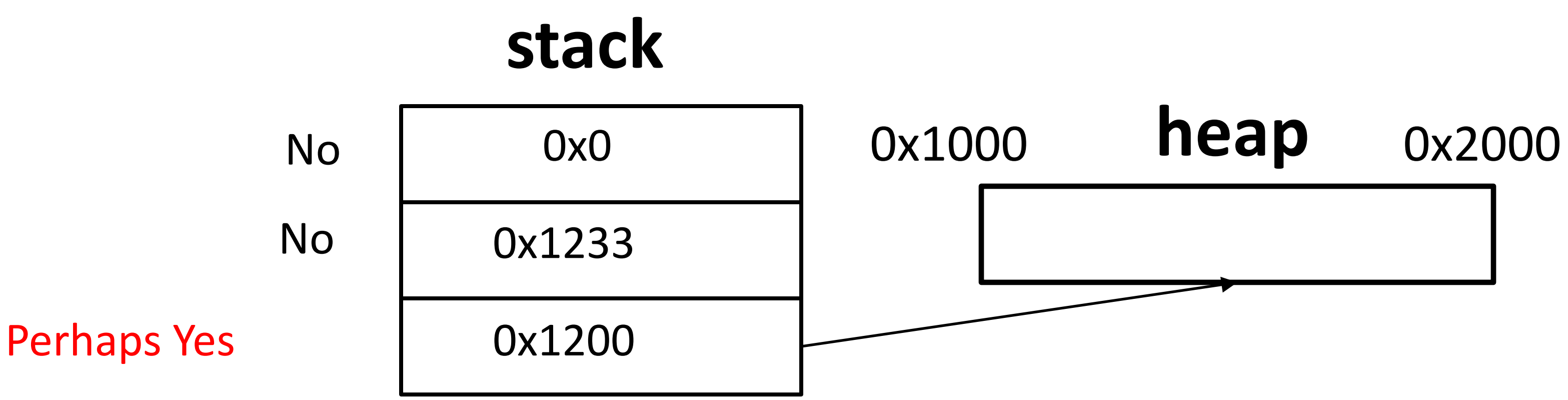


4. Interface to the Compiler

- Fast Allocation
- Describing Data Layouts
- Describing Roots (Pointer Map)
- Derived Pointers

Root Description

- GC starts from roots for tracing
  - But where are them?
- A straightforward design: guess!
  - Scanning stacks/registers
  - Finding all looking like pointers



Exact Root Description

- The guess-based solution is known as an approximate GC
  - Some integers might be treated as pointers
  - But how to implement an exact one?

- Tiger's solution: building a pointer map
  - All maps are generated by compilers
    - Compilers know which temp is a pointer during compilation

Pointer Maps

- A pointer map should consist:
  - Pointers on stack
  - Pointers in callee-saved registers
  - GC threads use those pointers to traverse
- Where should we insert a pointer map?
  - It depends on when GC is triggered
    - When allocation: inserting before *alloc_record*

```
gen_pointer_map
call alloc_record
```

```
alloc_record:
…(allocation)
```

Pointer Maps

- A pointer map should consist:
  - Pointers on stack
  - Pointers in callee-saved registers
  - GC threads use those pointers to traverse
- Where should we insert a pointer map?
  - It depends on when GC is triggered
    - When allocation: inserting before *alloc_record*
    - For recursive invocation: inserting for all function calls

```
gen_pointer_map
call xxx
```

```
gen_pointer_map
call alloc_record
```

```
alloc_record:
…(allocation)
```

Pointer Maps

- To find all the roots, the collector starts at the top of the stack and scans downward
  - Each return address keys the pointer-map entry that describes the next frame
  - In each frame, the collector marks (or forwards, if copying collection) from the pointers in that frame

Pointer Maps

- Callee-save registers need special handling
- Suppose function f calls g, which calls h.
- The pointer map for g must describe which of its callee-save registers contain pointers at the call to h and which are "inherited" from f.

4. Interface to the Compiler

- Fast Allocation
- Describing Data Layouts
- Describing Roots (Pointer Map)
- Derived Pointers

Derived Pointers

- Sometimes a compiled program has a pointer that points into the middle of a heap record, or that points before or after the record.
- For example:
  - a[i-2000] can be calculated internally as M[a-2000+i]

$$\begin{aligned} t1 &\leftarrow a\text{-}2000 \\ t2 &\leftarrow t1 + i \\ t3 &\leftarrow M[t2] \end{aligned}$$

- If a[i-2000] occurs inside a loop, the compiler might choose to hoist t1 ← a – 2000 outside the loop to avoid recalculating it in each iteration.
- if the loop also contains an *alloc*, and a garbage collection occurs while t1 is live: t1 does not point the the beginning of an object or (worse yet) points to an unrelated object.

Derived Pointers

$$t1 \leftarrow a\text{-}2000$$
$$t2 \leftarrow t1 + i$$
$$t3 \leftarrow M[t2]$$

- We say that t1 is *derived* from the *base* pointer a.
- The collector will be confused by t1
- How to handle this problem?

- The pointer map must identify each *derived pointer* and tell its base pointer.
- When the collector relocates *a* to address *a'*, it must adjust t1 to point to address t1 + a' - a
- a must remain live as long as t1 is live. For example:

Derived Pointers

```
let
  var a := int array[100] of 0
in
  for i := 1930 to 1990
      do f(a[i-2000])
end
```

$r1 \leftarrow 100$
$r2 \leftarrow 0$
call alloc
$a \leftarrow r1$
$t1 \leftarrow a - 2000$
$i \leftarrow 1930$
$L1 : r1 \leftarrow M[t1 + i]$
call f
$L2$: if $i \leq 1990$ goto L1

- The temporary a appears dead after the assignment to t1
- But then the pointer map associated with the return address L2 would not be able to "explain" t1 adequately.
- Therefore, a derived pointer implicitly keeps its base pointer live.

Summary

- Mark-and-sweep collection
- Reference counting
- Copying collection
- Generational Collection
- Incremental Collection
- Interface to the Compiler

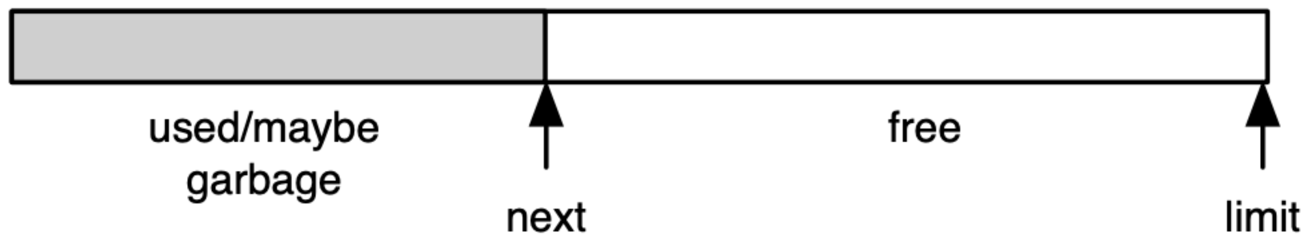
Summary

- Garbage collectors are a complex and fascinating part of any modern language implementation
- Different collection algs have pros/cons
  – explicit MM, reference counting, copying, 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

Thank you all for your attention

Allocation

- Linear allocation



- Freelist allocation