Memory Management
Chapter 5

Mooly Sagiv

Announcements

- Sample exams end of the week
- Hazara class & advanced topics next week
- Next week recitation solve sample exams
- Extra office hours TBD

Topics

- Heap allocation
- Manuel heap allocation
- Automatic memory reallocation (GC)

Limitations of Stack Frames

- A local variable of P cannot be stored in the activation record of P if its duration exceeds the duration of P

- Example: Dynamic allocation
int * f() { return (int *) malloc(sizeof(int));
}

Currying Functions

```
int (*)() f(int x)
{
  int g(int y)
   {
      return x + y;
    }
    return g ;
 }


int (*h)() = f(3);
int (*j)()  = f(4);

int z = h(5);
int  w = j(7);
```
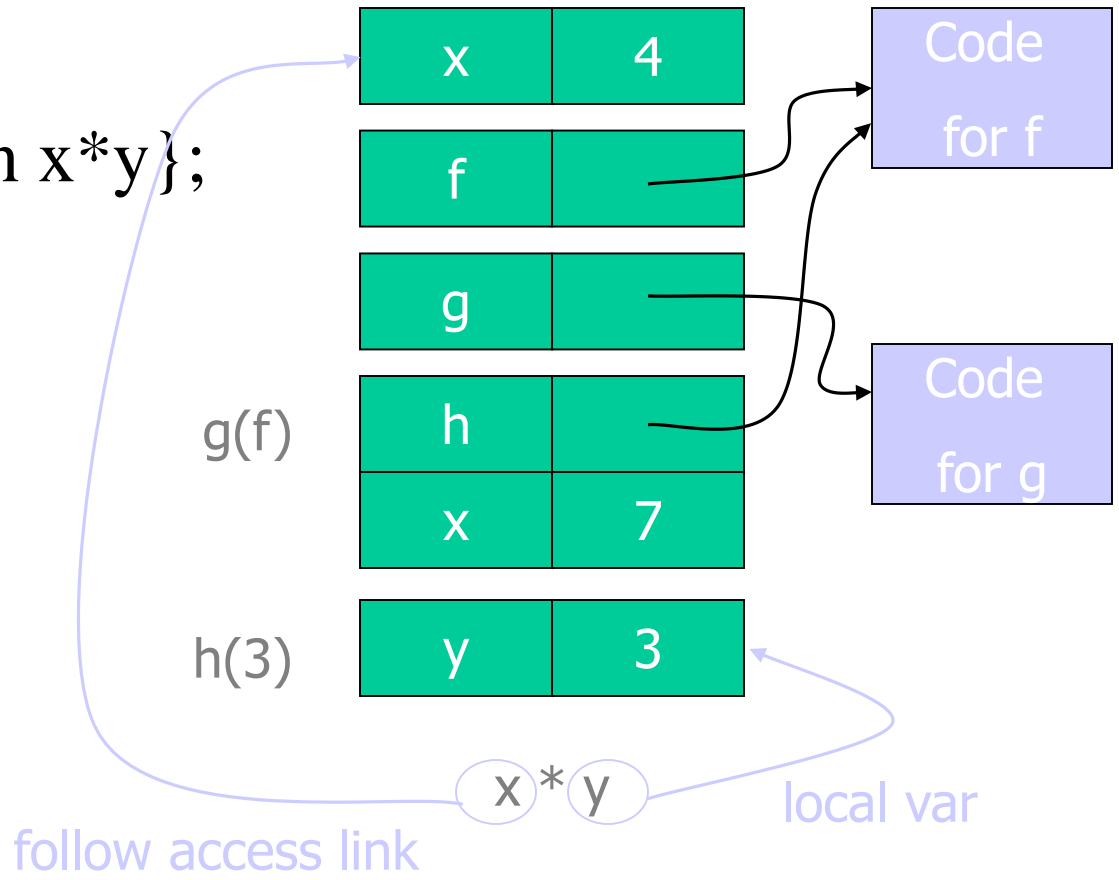
Browser events (Javascript)
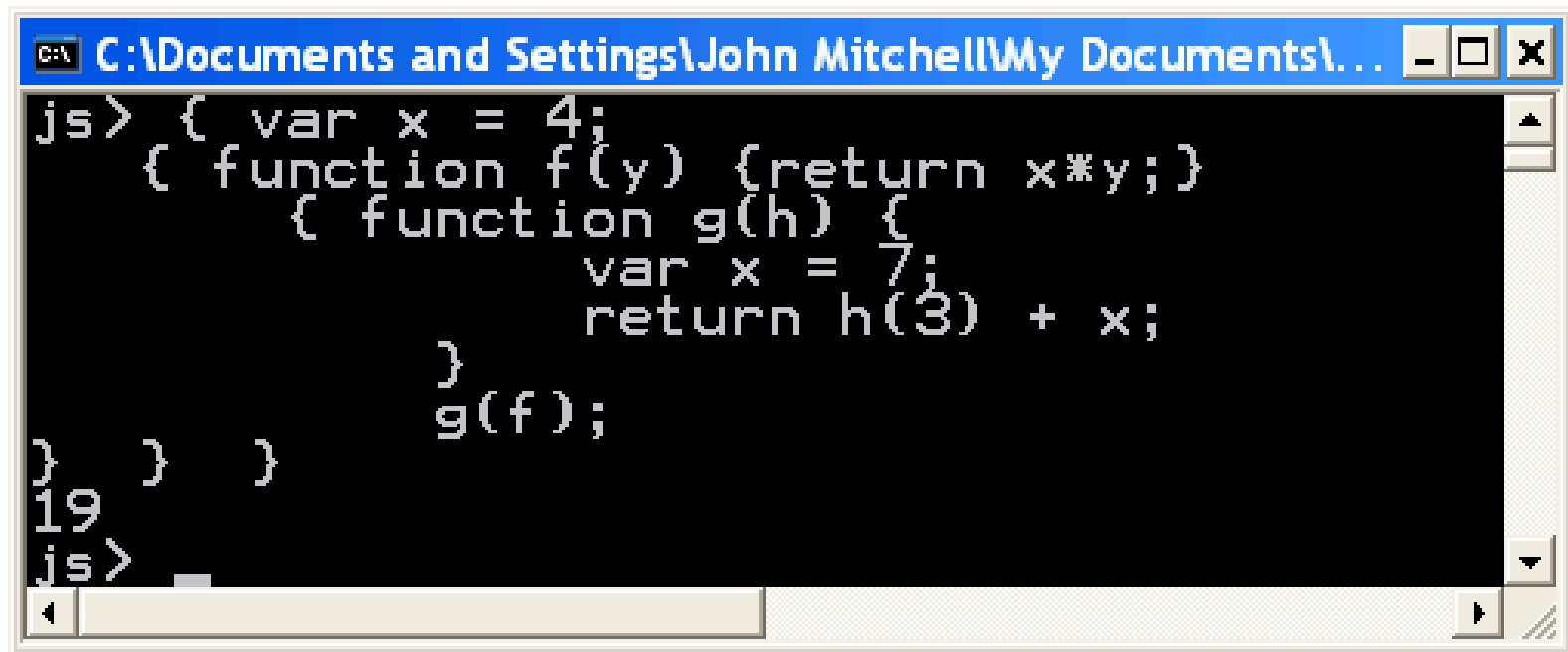
```
<script type="text/JavaScript">
    function whichButton(event) {
      if (event.button==1) {
                alert("You clicked the left mouse button!") }
      else {
                alert("You clicked the right mouse button!")
      }}
</script>
…
<body onmousedown="whichButton(event)">
…
</body>
```

Static Scope for Function Argument

```
{ var x = 4;
   { function f(y) {return x*y};
      { function g(h) {
           int x=7;
           return h(3) + x;
        };
      g(f);
} } }
```

x	4

f	

g	

g(f)

h	
x	7

h(3)

y	3

Code for f

Code for g

x * y

follow access link

local var

Result of function call
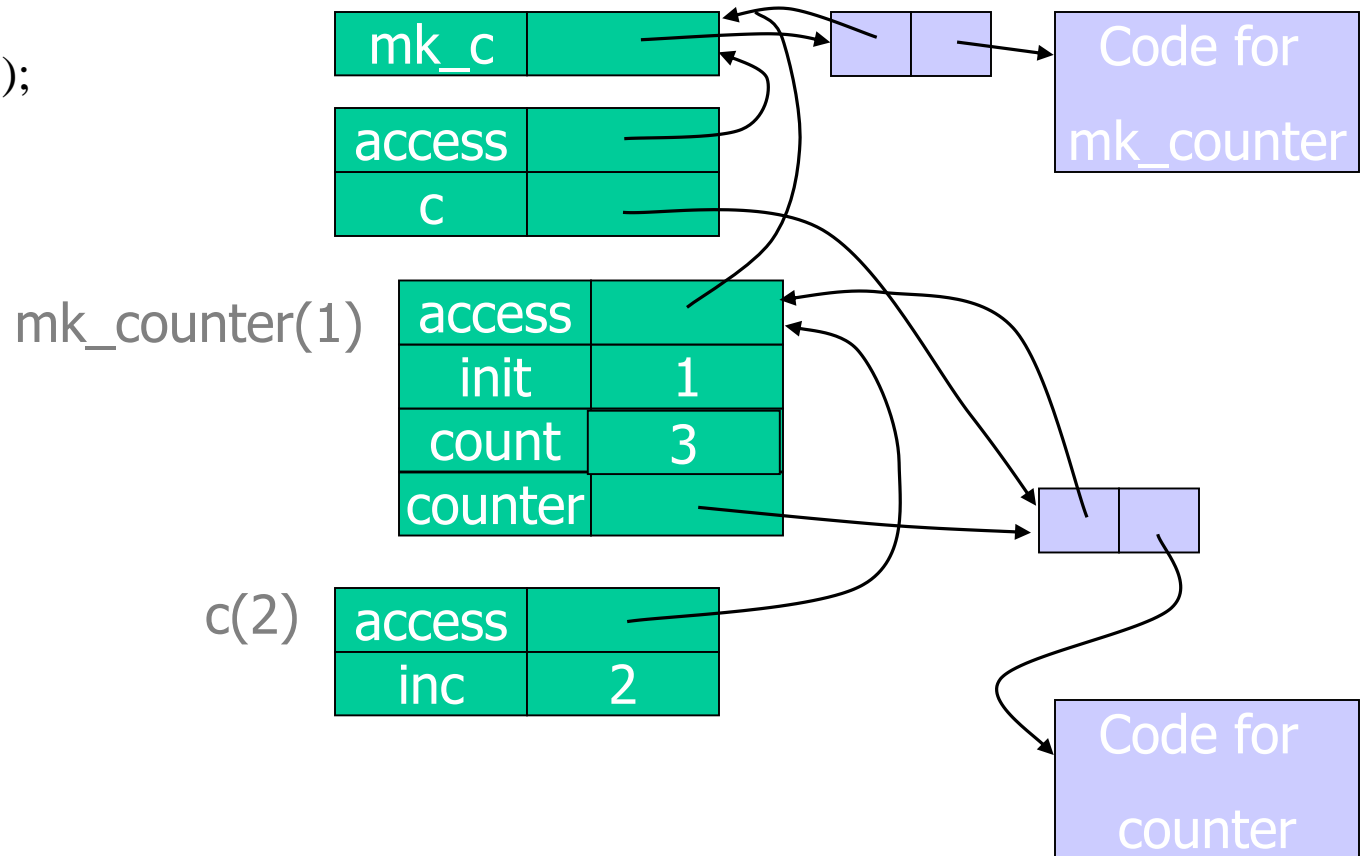
```
js> { var x = 4;
    { function f(y) {return x*y;}
      { function g(h) {
          var x = 7;
          return h(3) + x;
        }
        g(f);
      } } }
19
js>
```

Closures

- Activation records in the heap
- Function value is pair $closure = \langle env, code \rangle$
- When a function represented by a closure is called,
  - Allocate activation record for call (as always)
  - Set the access link in the activation record using the environment pointer from the closure

Function Results and Closures

```
function mk_counter (init) {
    var count = init;
    function counter(inc) {count=count+inc; return count};
    return counter};
var c  = mk_counter(1);
c(2) + c(2);
```

mk_c	
access	
c	

Code for mk_counter

mk_counter(1)

access	
init	1
count	3
counter	

c(2)

access	
inc	2

Code for counter

Duration

- The duration of a variable is the interval of time in which its value persist

- Examples
  - Automatic variables in C

    [block entry, block exit]

  - Frames in C
  - Frames in JS

Program Runtime State

Code segment

Stack segment

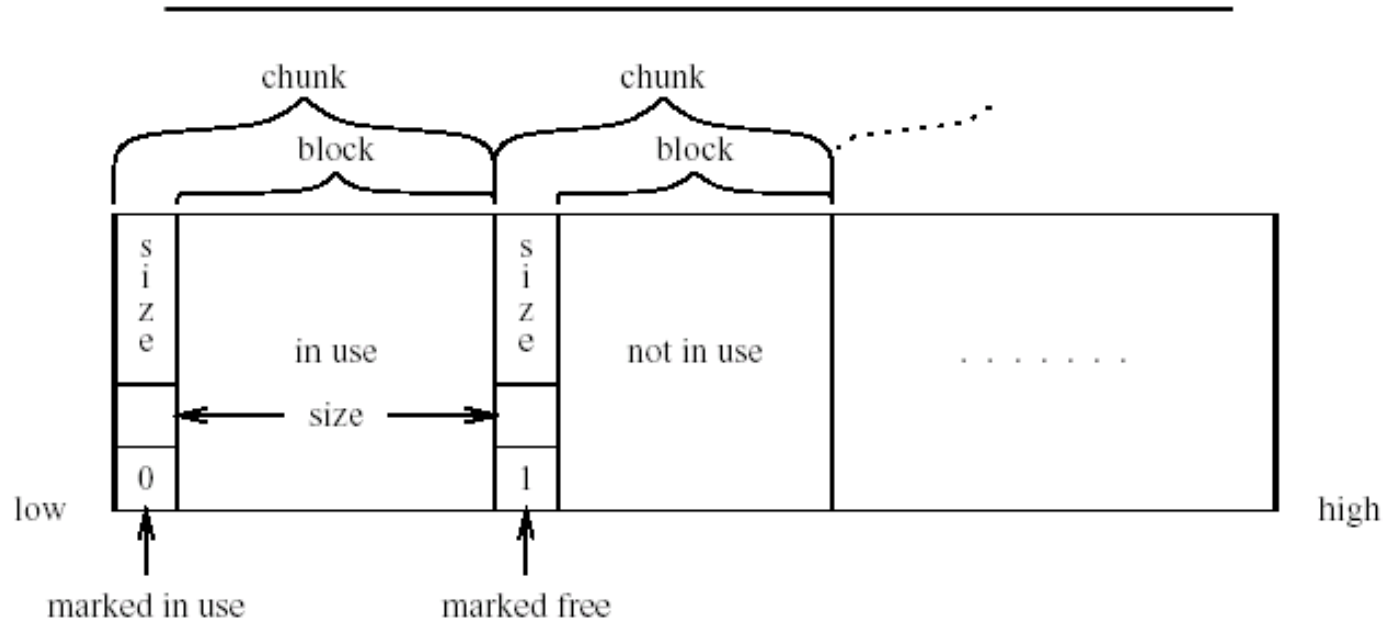Data Segment

Machine Registers

fixed heap
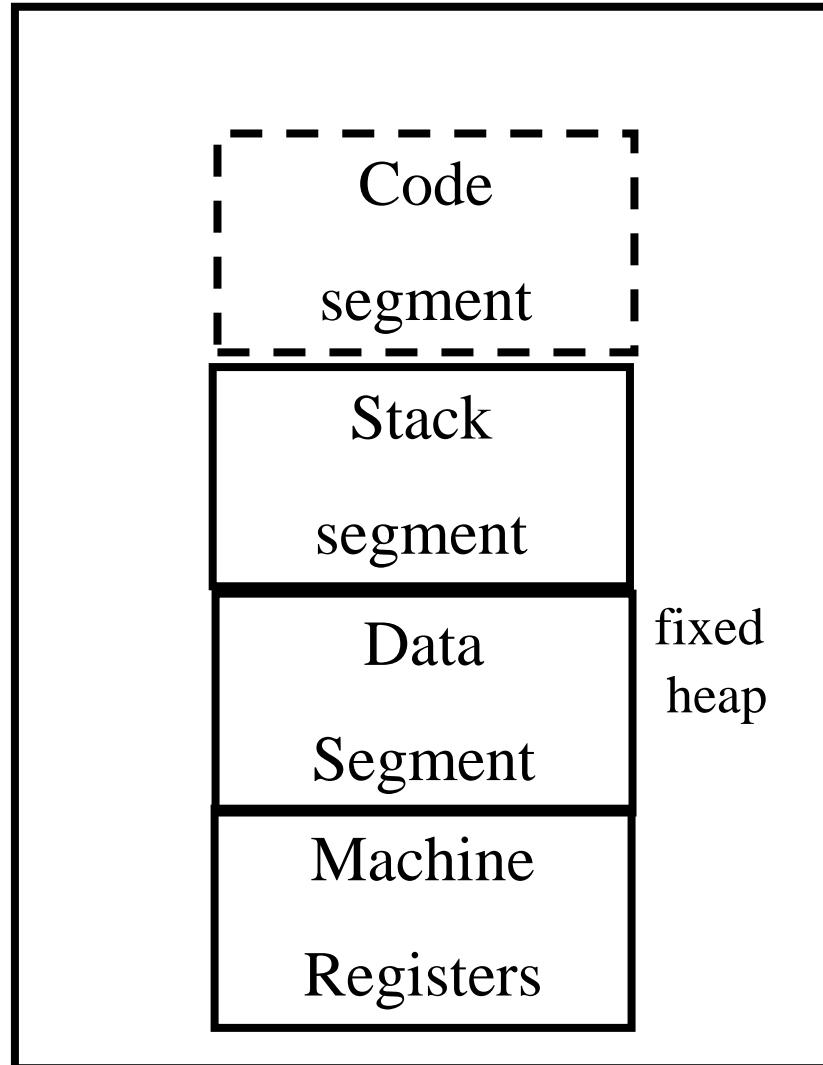
Data Allocation Methods

- Explicit deallocation
- Automatic deallocation

Explicit Deallocation

- Pascal, C, C++
- Two basic mechanisms
    - void * malloc(size_t size)
    - void free(void *ptr)
- Part of the language runtime
- Expensive
- Error prone
- Different implementations

Memory Structure used by malloc()/free()

Memory Structure used by malloc()/free()

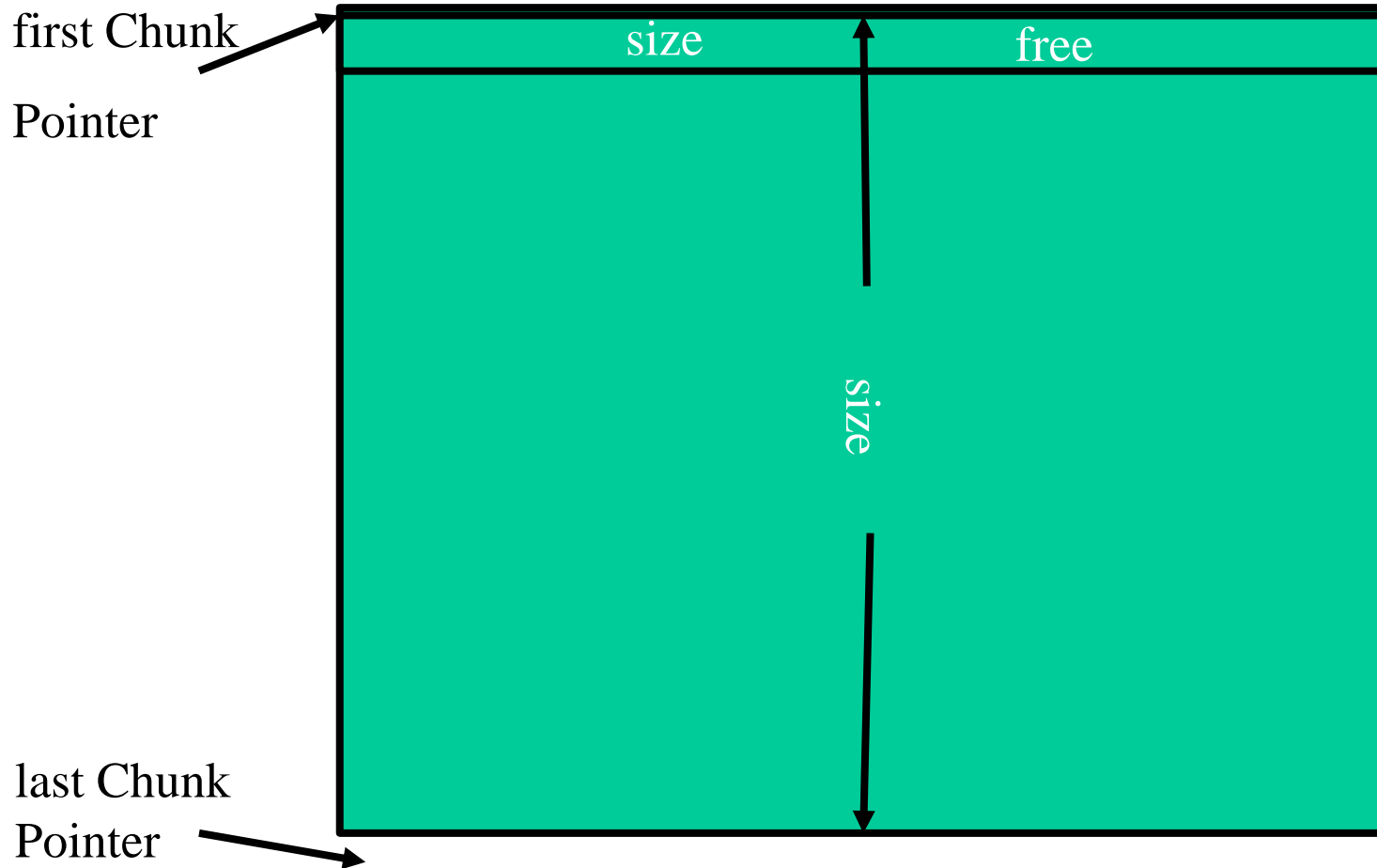| 1KB/0	

p= malloc(100)

100/1		896/0	

q= malloc(100)

100/1		100/1		792/0	

free(p)

100/0		100/1		792/0	

Simple Implementation (Init)

first Chunk
Pointer

size   free

size

last Chunk
Pointer

malloc implementation

function malloc(size) returning a polymorphic
block pointer

  pointer = next_free_block(size)

   if pointer ≠ null return pointer

   coalesce_free_chunks()

   pointer = next_free_block(size)

   if pointer ≠ null return pointer

   return a solution to out of memory with size

Next Free Block

function next_free_block(size) returning a polymorphic
block pointer

   pointer = first_chunk_pointer

   requested_size = size + administration_size;

   while pointer $\leq$ last_chunk_pointer do

     if pointer.size $\geq$ requested_size

       split(pointer, requested_size)
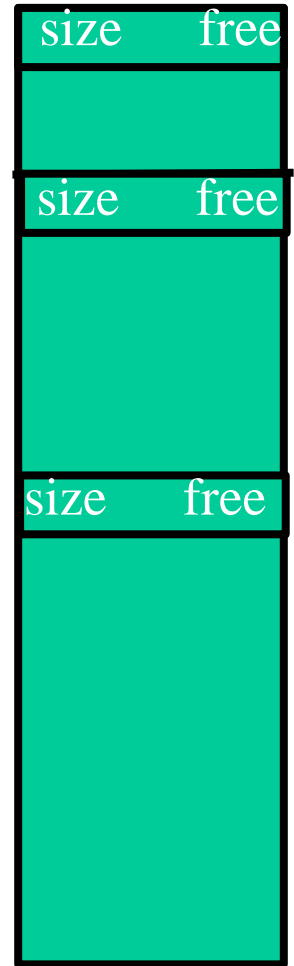
       pointer.free = false;

       return pointer + administrative_size

     pointer = pointer + pointer.size

  od

  return null

size	free
size	free
size	free

Splitting Chunks

split(pointer, requsted_size)

  leftover_size = pointer.size – requested_size
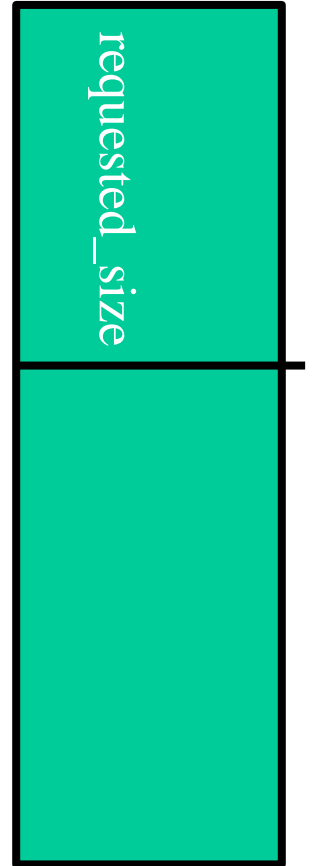
  if leftover_size > administrative_size

    pointer.size = requested_size

    leftover_pointer = pointer + requested_size

    leftover_pointer.free = true

    leftover_pointer.size = leftover_sizs

requested_size

Coalescing Chunks

coalesce_free_chunks

  pointer = first_chunk_pointer

  while pointer ≤ last_chunk_pointer do

    if pointer.free
        coelsce_with_followers(pointer)
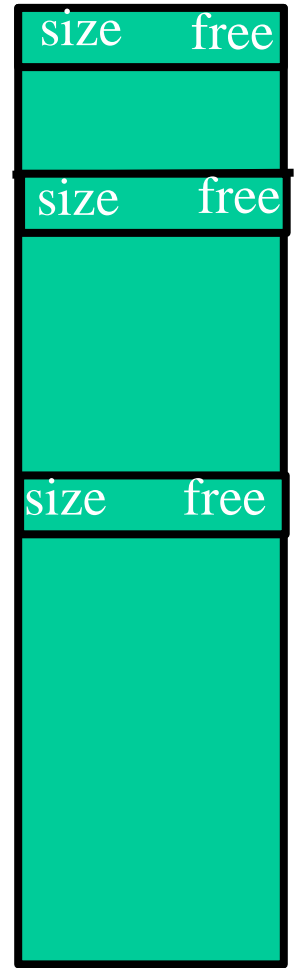
  pointer = pointer + pointer.size

coalesce_with_followers(pointer)

  next_pointer = pointer +pointer.size

  while next_pointer ≤ last_chunk_pointer

    and next_pointer.free do

    pointer.size = pointer_size + next_pointer_size

    next_pointer = next_pointer + next_pointer.size

size   free

size   free

size   free

Implementing Free

free(pointer)

    chunk_pointer = pointer – administrative_size

    chunk_pointer.free = true

Drawbacks of the simple implementation

Fragmentation

- External
 - Too many small chunks
- Internal
  - A use of too big chunk without splitting the chunk
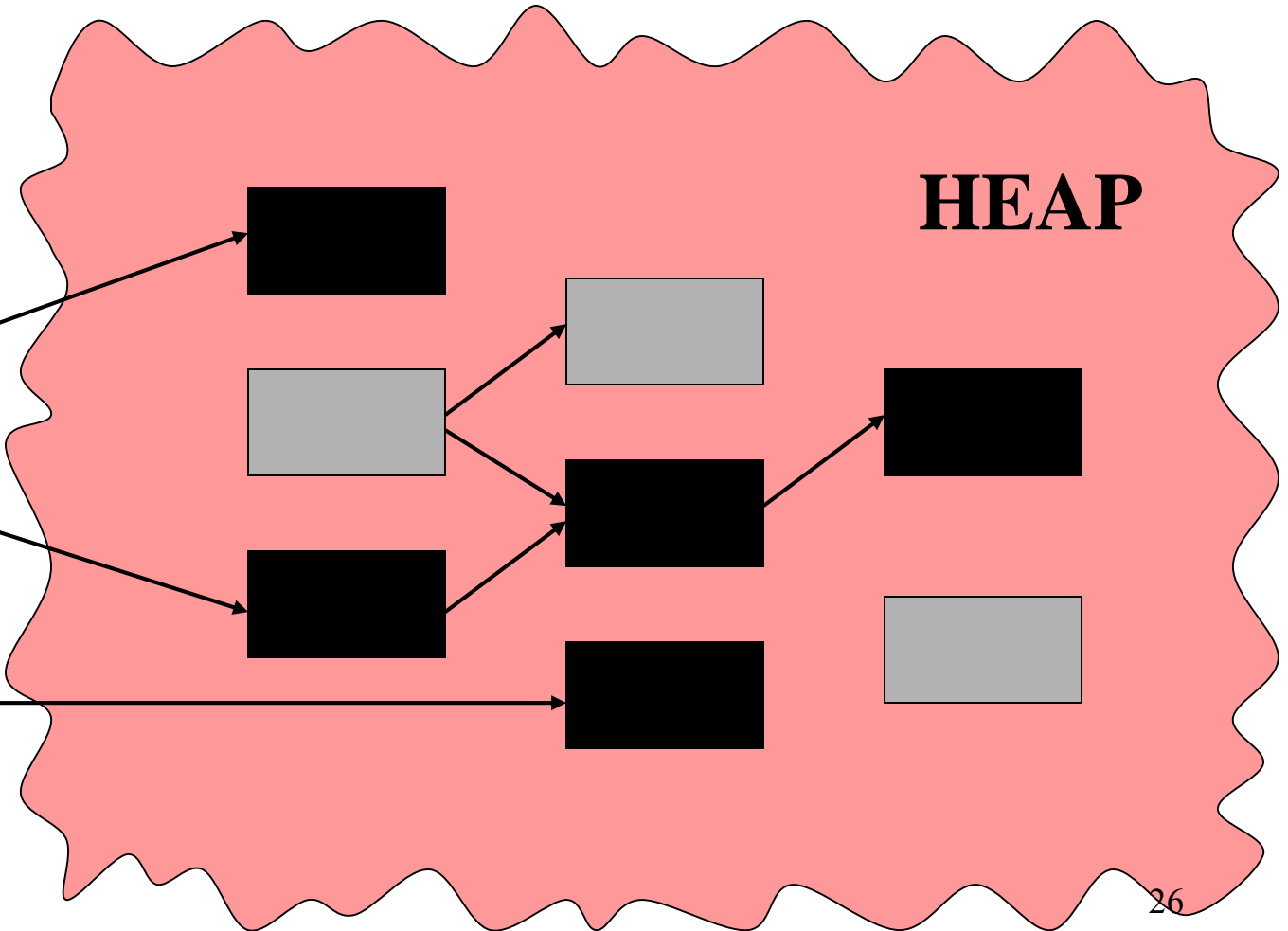- Freelist may be implemented as an array of lists

Summary Explicit Allocation/Free

- Considerable overhead

- Sophisticated implementations
  - Fragmentation
  - Locality of references

Garbage Collection

ROOT SET

HEAP

a
b
c
d
e
f

Stack +Registers

26

Garbage Collection

ROOT SET

HEAP

a
b
c
d
e
f
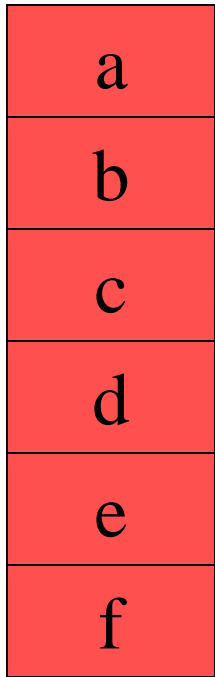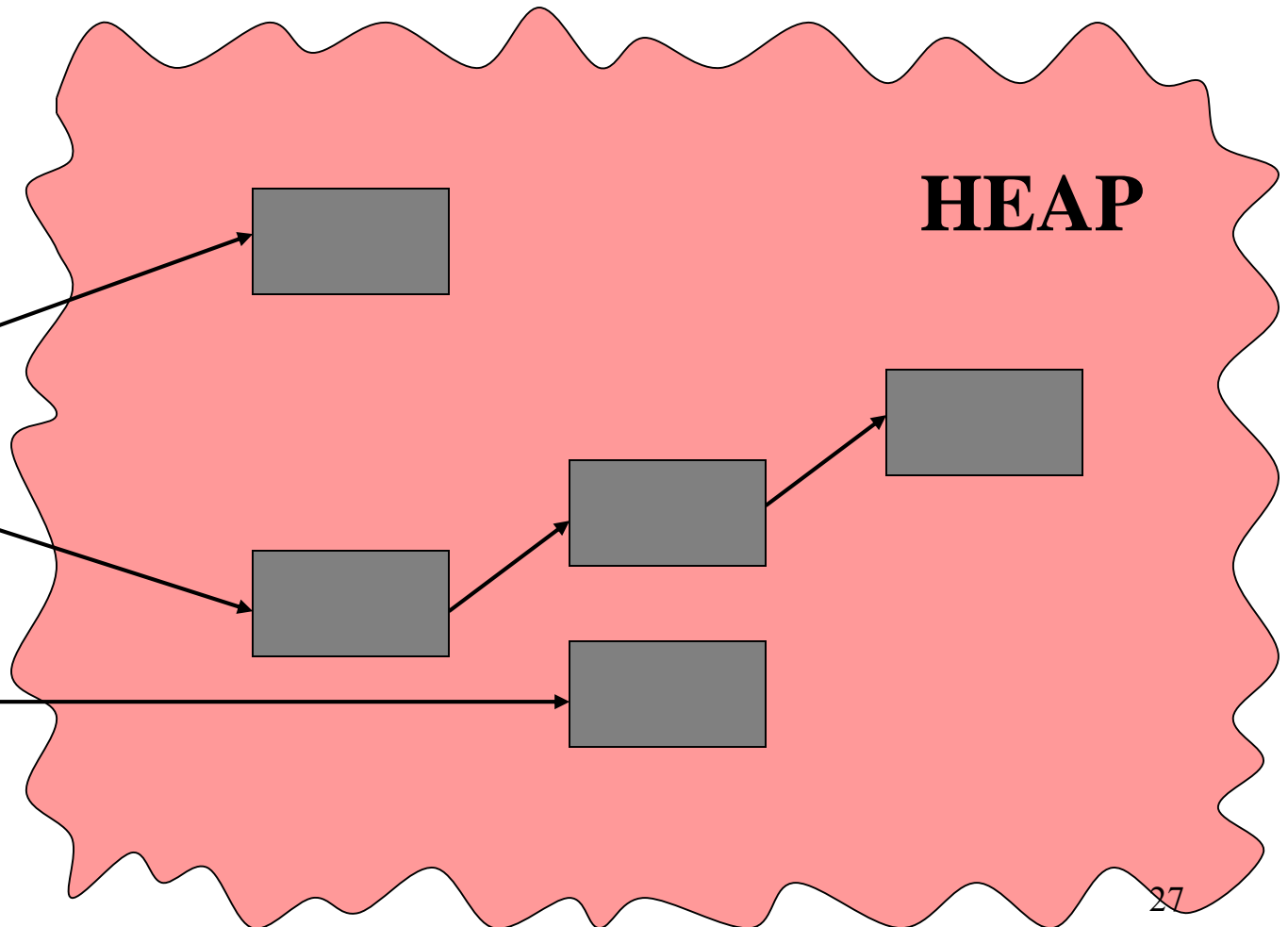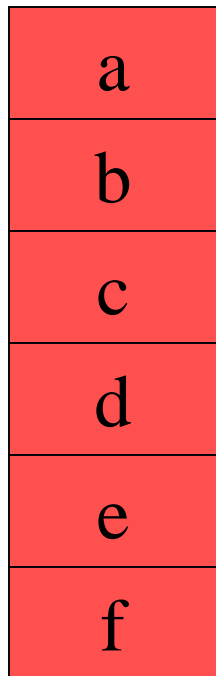
Stack +Registers

What is garbage collection

- The runtime environment reuse chunks that were allocated but are not subsequently used

- garbage chunks
  - not live

- It is undecidable to find the garbage chunks:
  - Decidability of liveness
  - Decidability of type information

- conservative collection
  - every live chunk is identified
  - some garbage runtime chunk are not identified

- Find the reachable chunks via pointer chains

- Often done in the allocation function

typedef struct list {struct list *link; int key} *List;

typedef struct tree {int key;

　　　　　　　　struct tree *left:

　　　　　　　　struct tree *right} *Tree;

foo() {　List x = cons(NULL, 7);

　List y = cons(x, 9);

　x->link = y;

　}

void main() {

　Tree p, r; int q;

　foo();

　p = maketree();　r = p->right;

　q= r->key;

　showtree(r);}

stack　　　　heap

p

q

r

x

y

link

7

link

9

```
typedef struct list  {struct list *link; int key} *List;

typedef struct tree {int key;

                struct tree *left:

                struct tree *right} *Tree;

foo() {    List x = cons(NULL, 7);

    List y = cons(x, 9);

    x->link = y;

     }

void main() {

  Tree p, r; int q;

  foo();

  p = maketree();   r = p->right;

  q= r->key;

  showtree(r);}
```

stack   heap

p

q

r

x

y

link

key   7

link

key   9
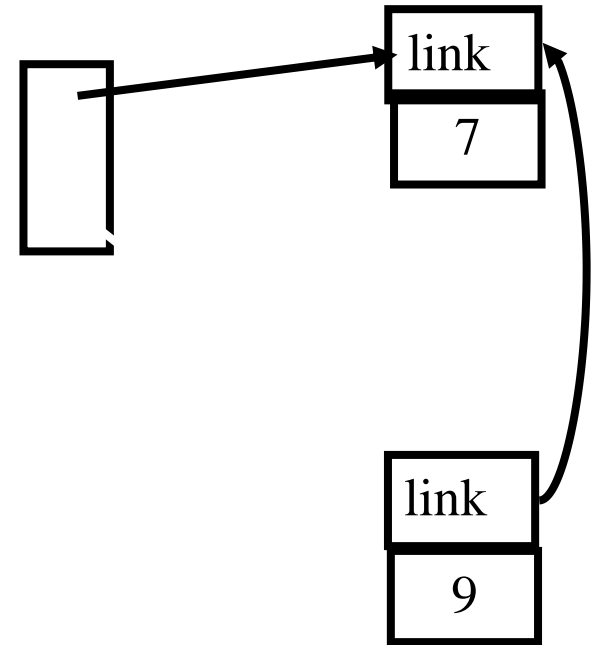
```
typedef struct list  {struct list *link; int key} *List;

typedef struct tree {int key;

                   struct tree *left:

                   struct tree *right} *Tree;

foo() {    List x = create_list(NULL, 7);

    List y = create_list(x, 9);

   x->link = y;

      }

void main() {

  Tree p, r; int q;

  foo();

  p = maketree();   r = p->right;

  q= r->key;

  showtree(r);}
```


p

q   37

r

12
left
right
15
left
right
link
7
37
left
right
59
left
right
link
31
9

20
left
right
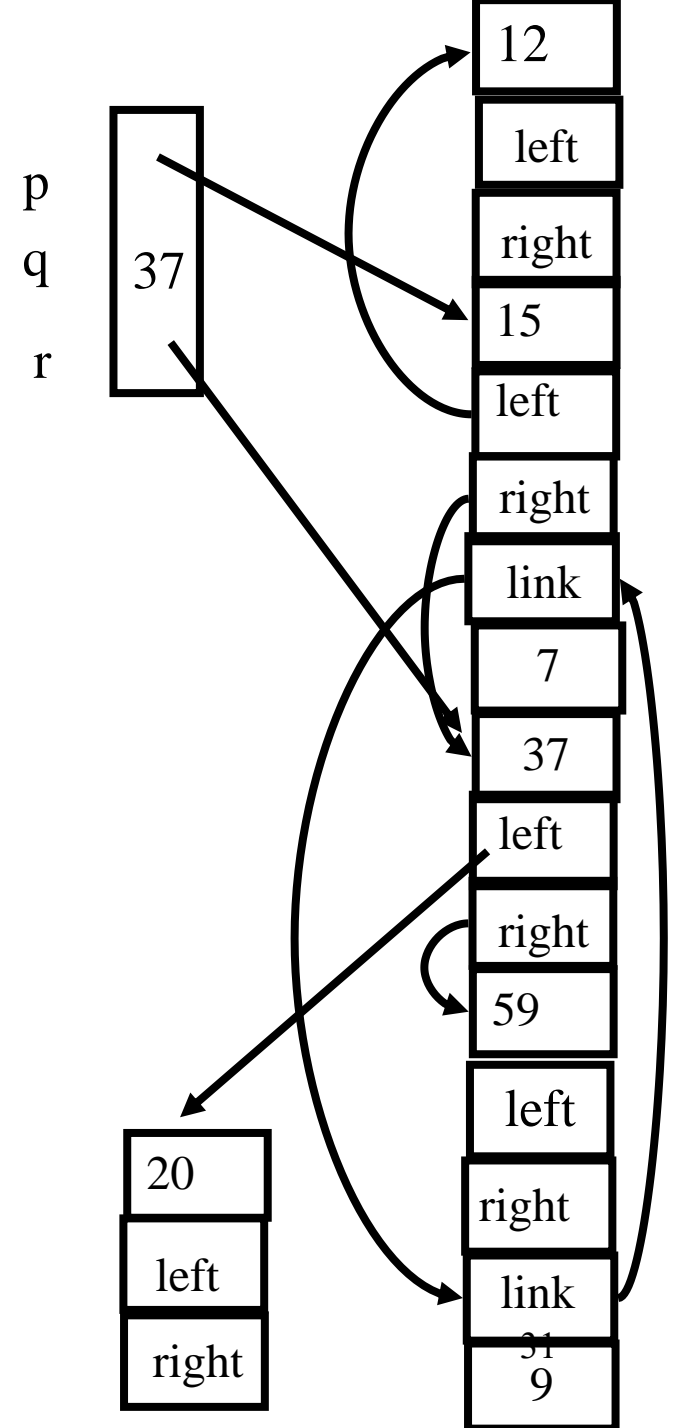
Outline

- Why is it needed?

- Why is it taught?

- Reference Counts

- Mark-and-Sweep Collection

- Copying Collection

- Generational Collection

Tracing

- Incremental Collection

- Interfaces to the Compiler

A Pathological C Program

a = malloc(…) ;

b = a;

free (a);

c = malloc (…);

if (b == c) printf("unexpected equality");

Garbage Collection vs. Explicit Memory Deallocation

- Faster program development
- Less error prone
- Can lead to faster programs
  - Can improve locality of references
- Support very general programming styles, e.g. higher order and OO programming
- Standard in ML, Java, C#
- Supported in C and C++ via separate libraries

- May require more space
- Needs a large memory
- Can lead to long pauses
- Can change locality of references
- Effectiveness depends on programming language and style
- Hides documentation
- More trusted code

Interesting Aspects of Garbage Collection

- Data structures
- Non constant time costs
- Amortized algorithms
- Constant factors matter
- Interfaces between compilers and runtime environments
- Interfaces between compilers and virtual memory management

Reference Counts

- Maintain a counter per chunk
- The compiler generates code to update counter
- Constant overhead per instruction

p
q
r

37

12   1
left
right
15   1
left
right
link   1
7
37   2
left
right
59   1
left
right
link   1

1   20
left
right

9

37
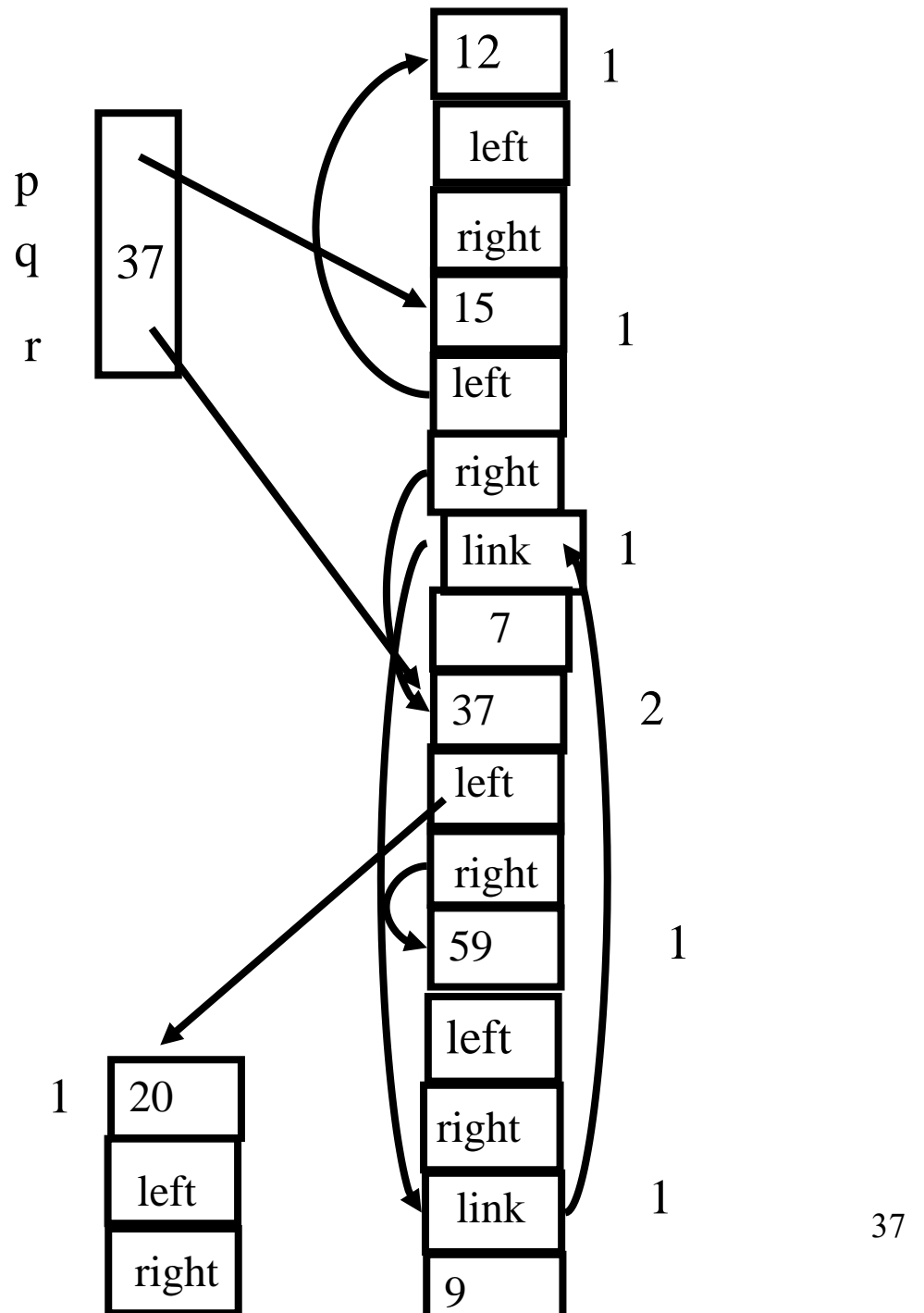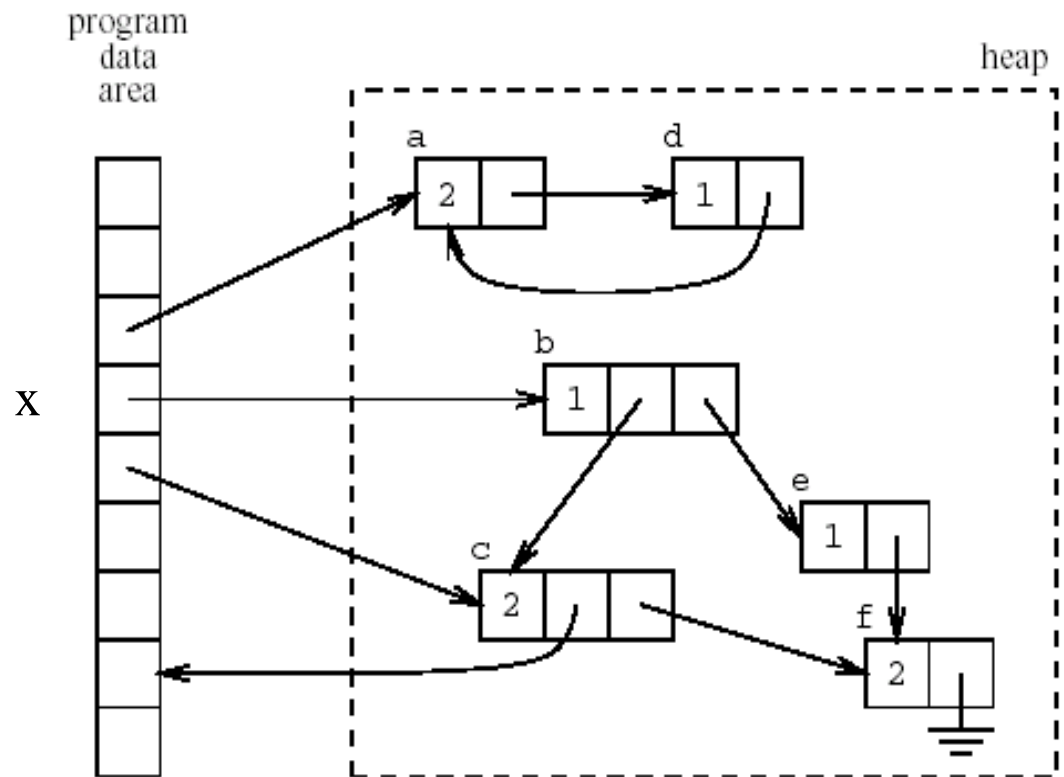
Another Example

Another Example (x→b=NULL)
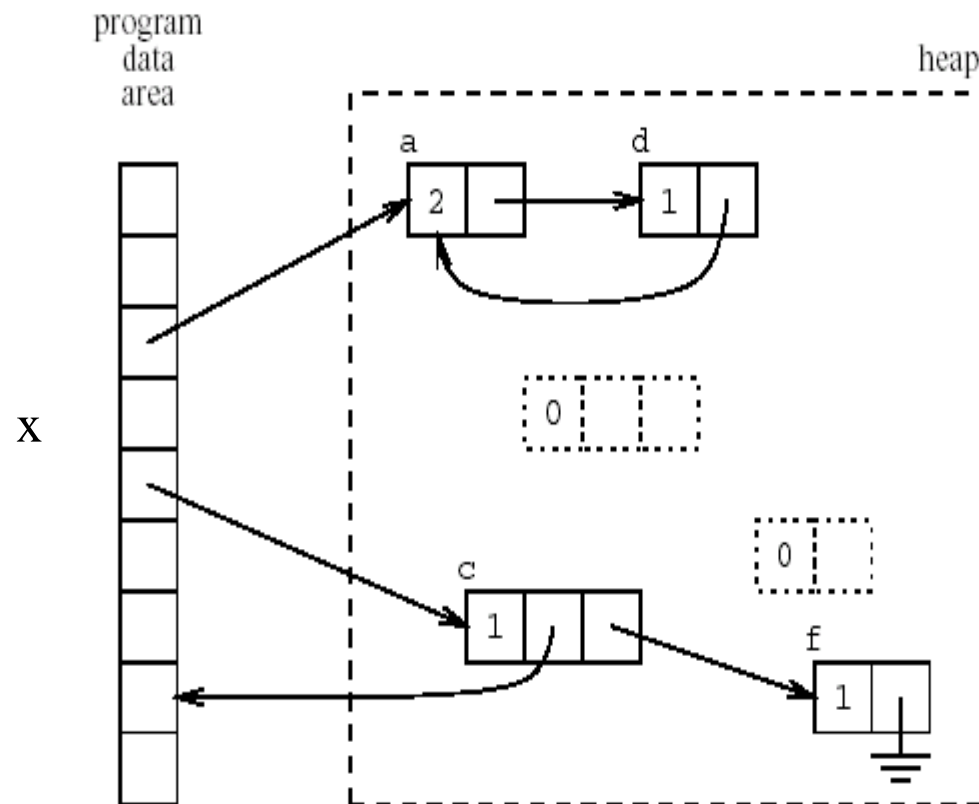
Code for p := q

if points to the heap q increment q's reference count

if points to the heap p
    decrement p's reference count

    if p's reference count becomes zero then recursively free

Recursive Free

```
PROCEDURE Free recursively depending on reference counts(Pointer);
    WHILE Pointer /= No chunk:
        IF NOT Points into the heap (Pointer): RETURN;
        IF NOT Pointer .reference count = 0: RETURN;

        FOR EACH Index IN 1 .. Pointer .number of pointers - 1:
            Free recursively depending on reference counts
                (Pointer .pointer [Index]);

        SET Aux pointer TO Pointer;
        IF Pointer .number of pointers = 0:
            SET Pointer TO No chunk;
        ELSE Pointer .number of pointers > 0:
            SET Pointer TO
                Pointer .pointer [Pointer .number of pointers];
        Free chunk(Aux pointer);    // the actual freeing operation
```

Asymptotic Complexity

- Reference counting can be implemented with constant overhead

- How?

Lazy Reference Counters

- Free one element

- Free more elements when required

- Constant time overhead

- But may require more space

Reference Counts (Summary)

- Fixed but big constant overhead
- Fragmentation
- Cyclic Data Structures
- Compiler optimizations can help
- Can delay updating reference counters from the stack
- Implemented in libraries and file systems
  - No language support
- But not currently popular
- Will it be popular for large heaps?

Mark-and-Sweep(Scan) Collection

- **Mark** the chunks reachable from the roots (stack, static variables and machine registers)

- **Sweep** the heap space by moving unreachable chunks to the freelist (Scan)

The Mark Phase

for each root v

  DFS(v)

function DFS(x)

  if x is a pointer and chunk x is not marked

       mark x

       for each reference field f_i of chunk x

          DFS(x.f_i)

The Sweep Phase

p := first address in heap

while p < last address in the heap

    if chunk p is marked

        unmark p

    else let f_1 be the first pointer reference field in p

        $p.f_1$:= freelist

        freelist := p

    p := p + size of chunk p

Mark

p
q   37
r

12

left

right

15

left

right

link

7

37

left

right

59

left

right

link

20

left

right

9

48

Sweep

p
q   37
r

freelist

12

left

right

15

left

right

link

7

37

left

right

59

left

right

link

9

20

left

right

49

p
q   37
r

freelist

12

left

right

15

left

right

link

7

37

left

right

59

left

right

link

9

20

left

right

50
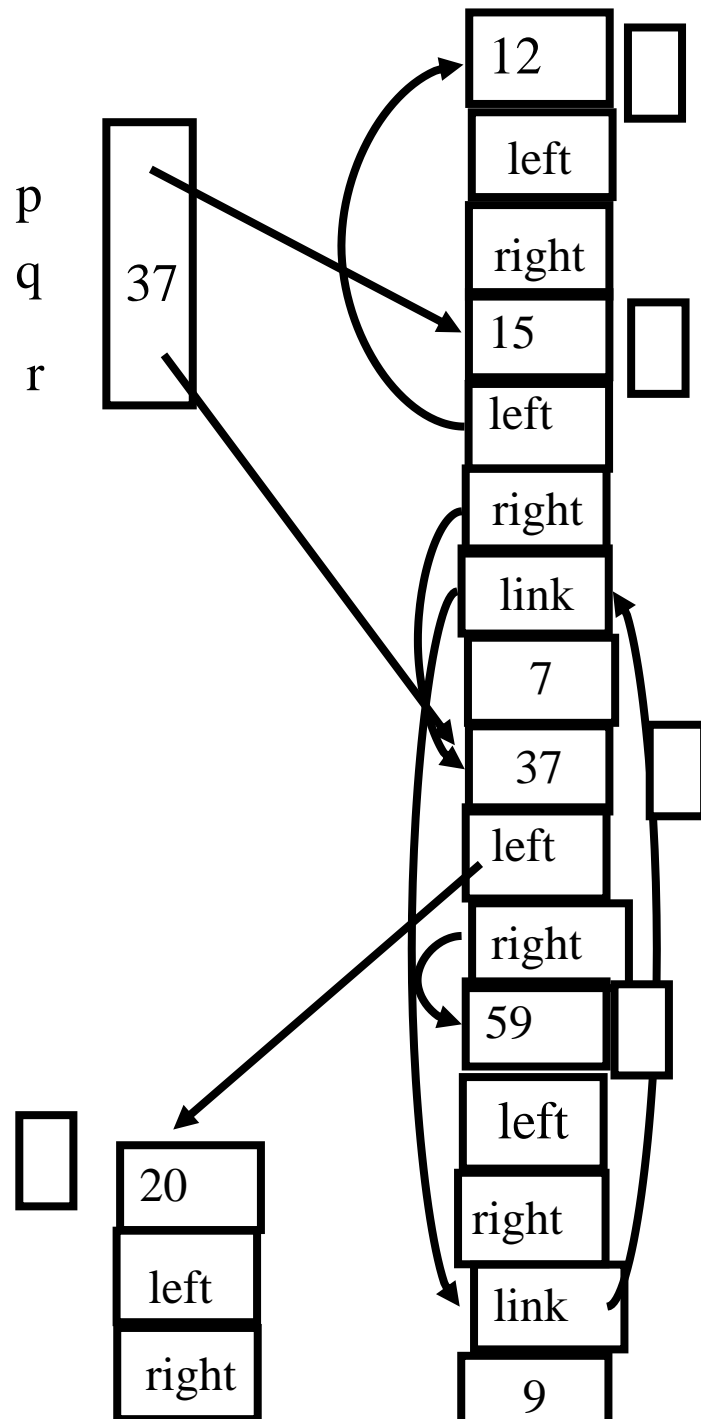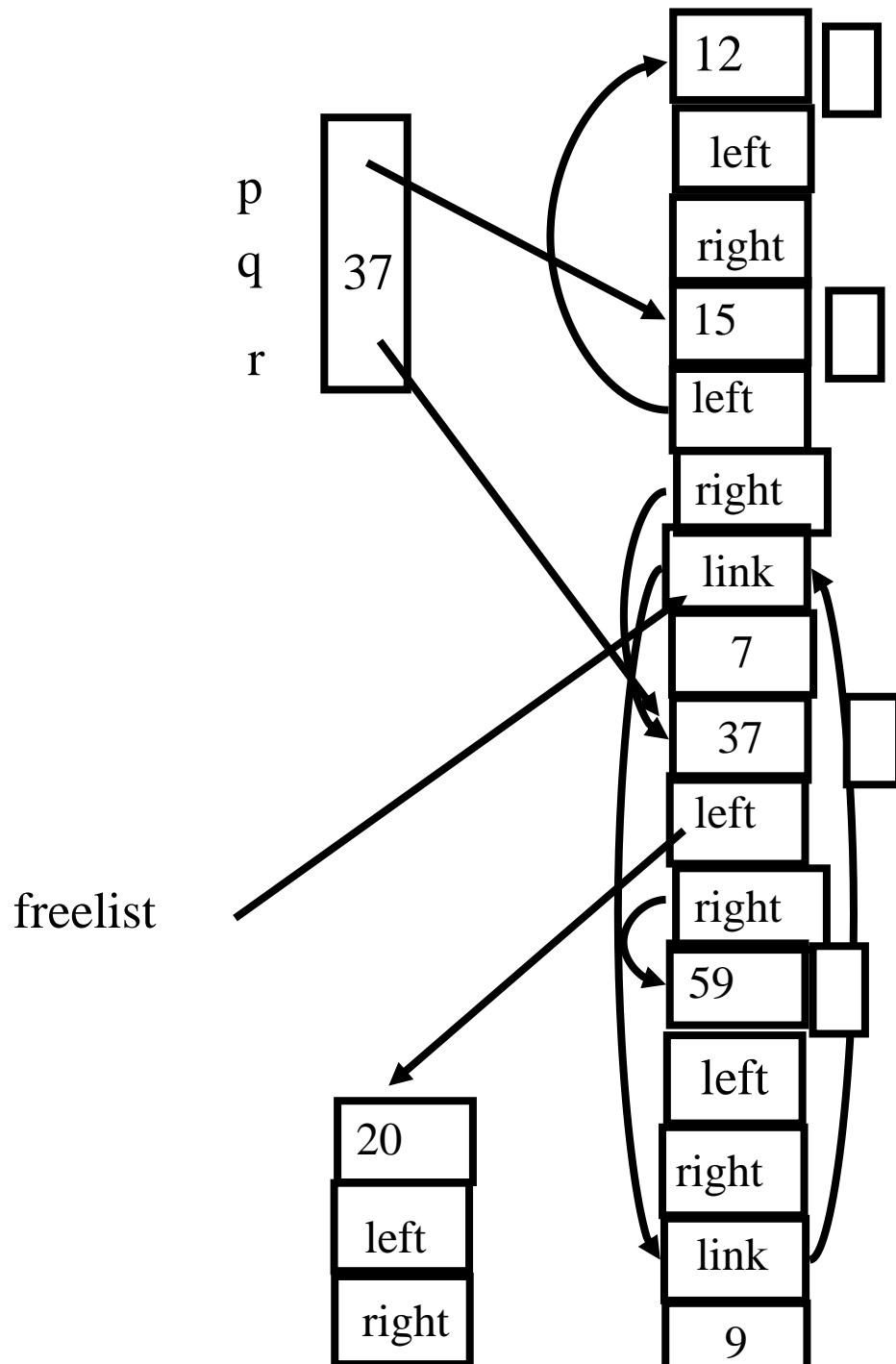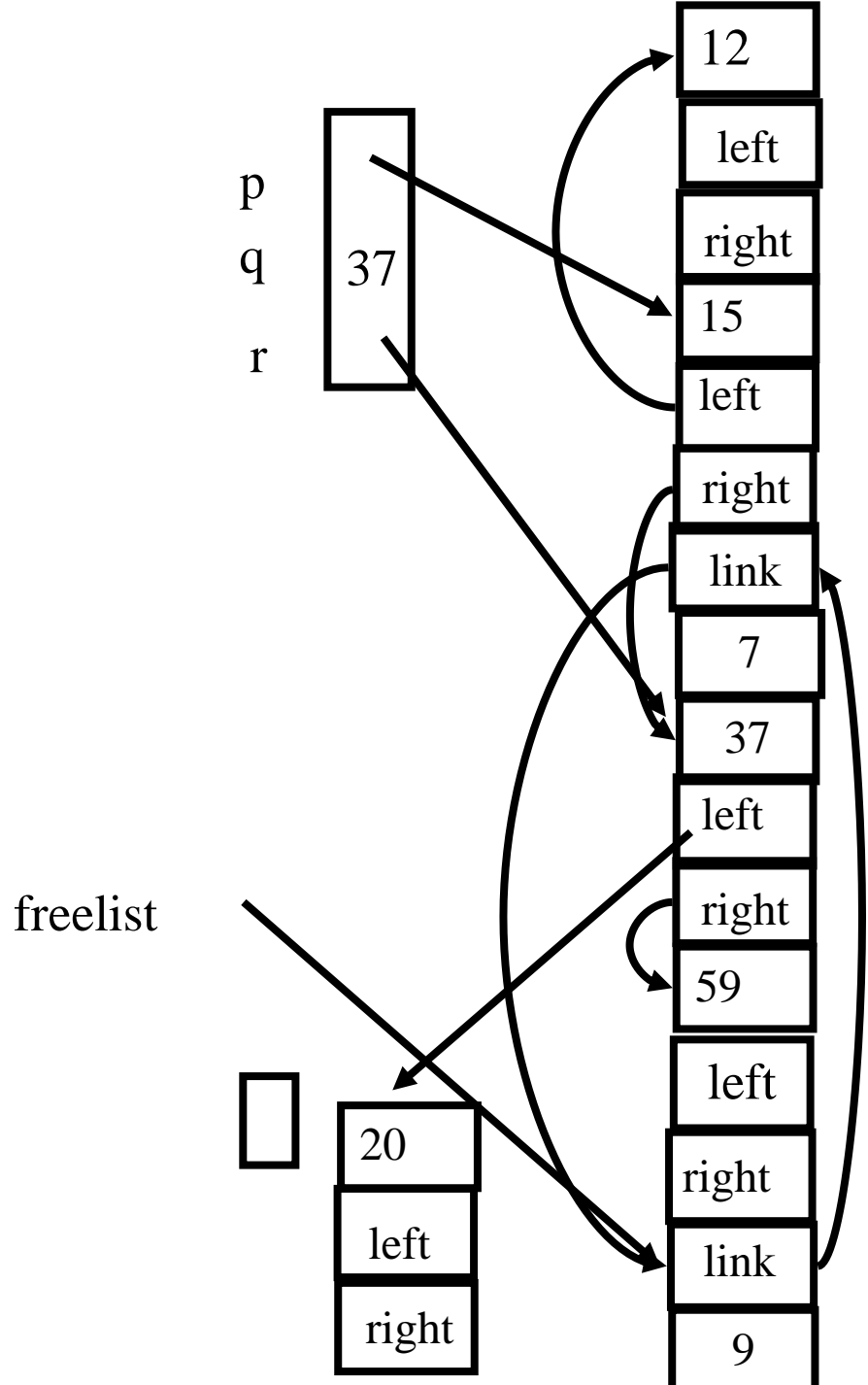
Cost of GC

- The cost of a single garbage collection can be linear in the size of the store
  - may cause quadratic program slowdown
- Amortized cost
  - collection-time/storage reclaimed
  - Cost of one garbage collection
    - $c_1 R + c_2 H$
  - H - R Reclaimed chunks
  - Cost per reclaimed chunk
    - $(c_1 R + c_2 H)/ (H - R)$
  - If R/H > 0.5
    - increase H
  - if R/H < 0.5
    - cost per reclaimed word is $c_1 + 2c_2$ ~16
  - There is no lower bound

The Mark Phase

for each root v

  DFS(v)

function DFS(x)

   if x is a pointer and chunk x is not marked

        mark x

        for each reference field f_i of chunk x

          DFS(x.f_i)

Efficient implementation of Mark(DFS)

- Explicit stack
- Parent pointers
- Pointer reversal
- Other data structures

Adding Parent Pointer

to parent

size

marked bit

pointer counter

free bit

1 2 3 4

1

3

0

pointer pointer pointer pointer

to parent

3rd child

. . . .
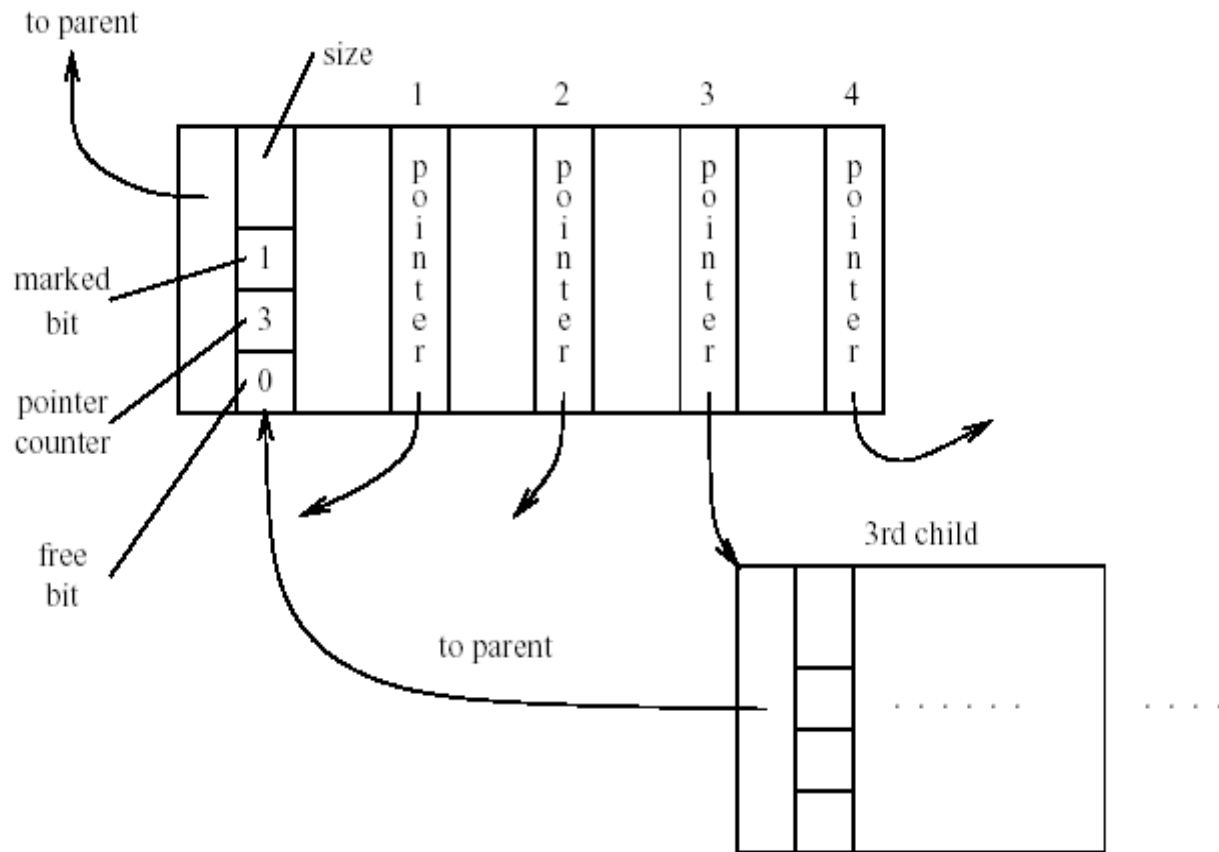
Avoiding Parent Pointers (Deutch-Schorr-Waite)

- Depth first search can be implemented without recursion or stack

- Maintain a counter of visited children

- Observation:

  - The pointer link from a parent to a child is not needed when it is visited

  - Temporary store pointer to the parent (instead of the field)

  - Restore when the visit of child is finished

Arriving at C
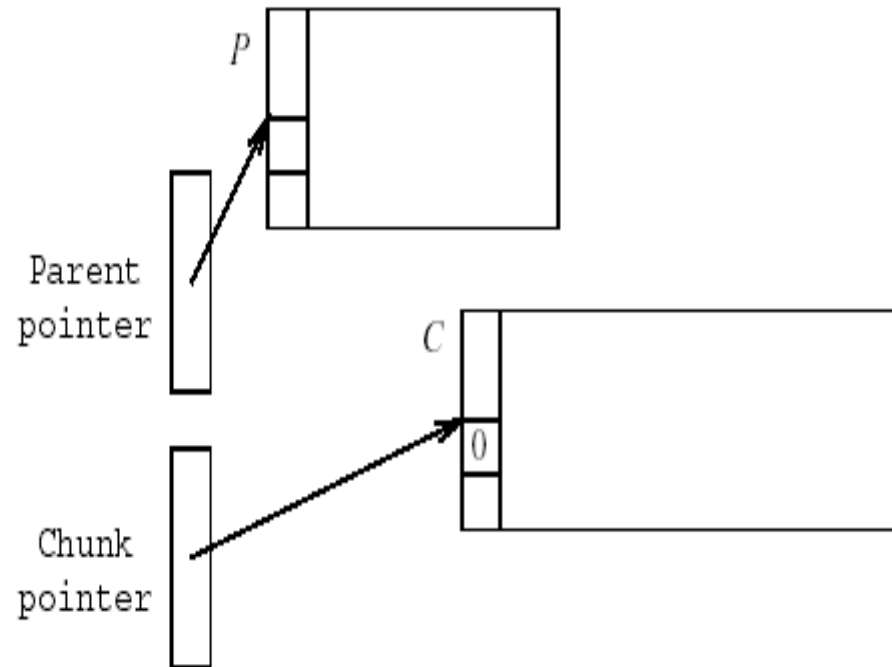
Visiting n-pointer field D

SET old parent pointer TO parent pointer ;

SET Parent pointer TO chunk pointer ;

SET Chunk pointer TO n-th pointer field of C;

SET n-th pointer field in C TO old parent pointer;
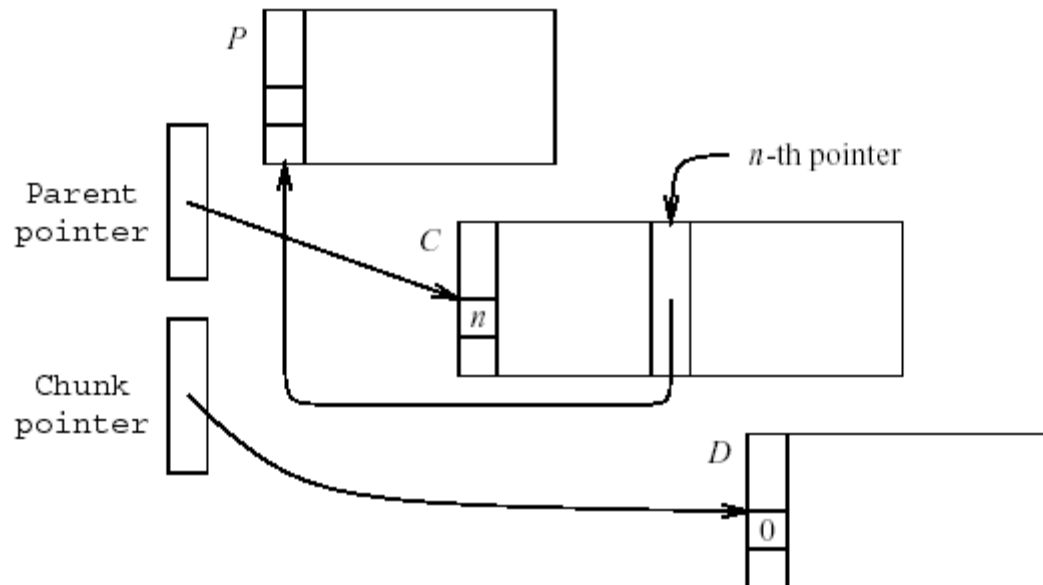
About to return from D
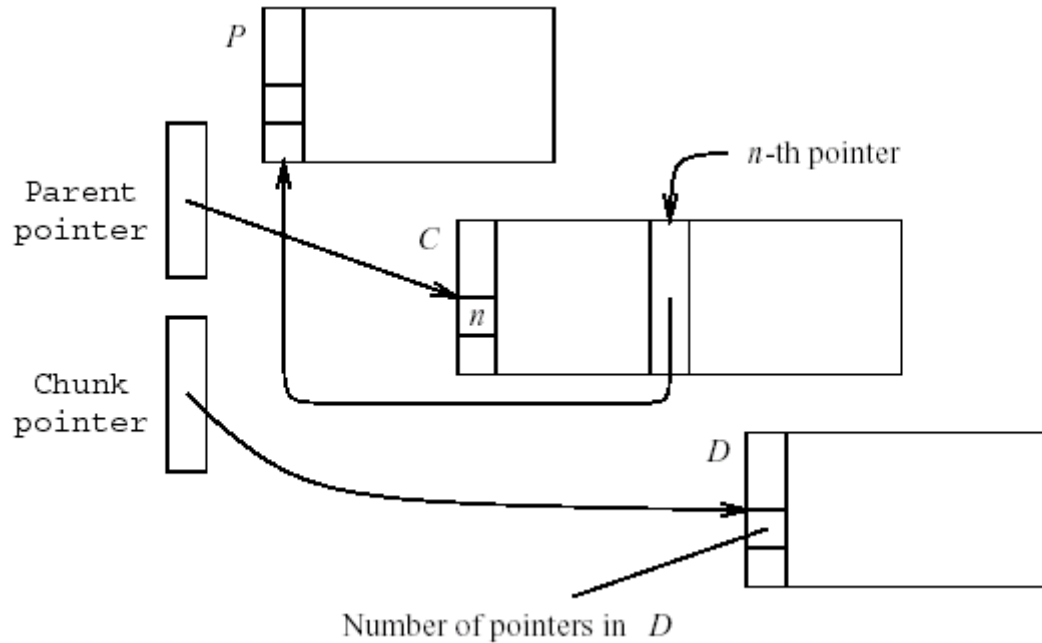
SET old parent pointer TO Parent pointer ;

SET Parent pointer TO n-th pointer field of C ;

SET n-th pointer field of C TO chunk pointer;

SET chunk pointer TO old parent pointer;

58

Compaction

- The sweep phase can compact adjacent chunks

- Reduce fragmentation

Copying Collection

- Maintains two separate heaps
  - from-space
  - to-space
- pointer next to the next free chunk in from-space
- A pointer limit to the last chunk in from-space
- If next = limit copy the reachable chunks from from-space into to-space
  - set next and limit
  - Switch from-space and to-space
- Requires type information

To-Space

next

limit

Breadth-first Copying Garbage Collection

next := beginning of to-space

scan := next

for each root r

    r := Forward(r)

while scan < next

        for each reference field f_i of chunk at scan

            $scan.f_i$:= Forward($scan.f_i$)

       scan := scan + size of chunk at scan

The Forwarding Procedure

function Forward(p)

if p points to from-space

then if p.f_1 points to to-space

return p.f_1

else for each reference field f_i of p

next.f_i := p.f_i

p.f_1 := next

next := next size of chunk p

return p.f_1

else return p

A Simple Example

stack

f400 | 17 | ← | f400 |
       | f800 |
       | 0 |

b

f

f800 | 13 |
       | 0 |
       | f400 |
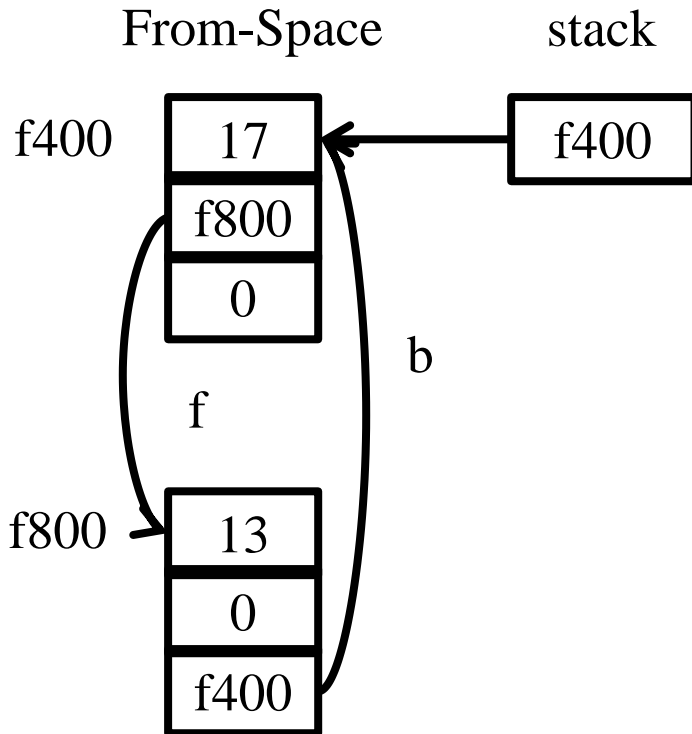
struct DL{

  int data;

   struct DL* f;

   struct DL *b

    }

Before Forward(f400)

From-Space   stack   to-Space

f400

| 17 |
| f800 |
| 0 |

f400

scan

t600

next

b

f

f800

| 13 |
| 0 |
| f400 |
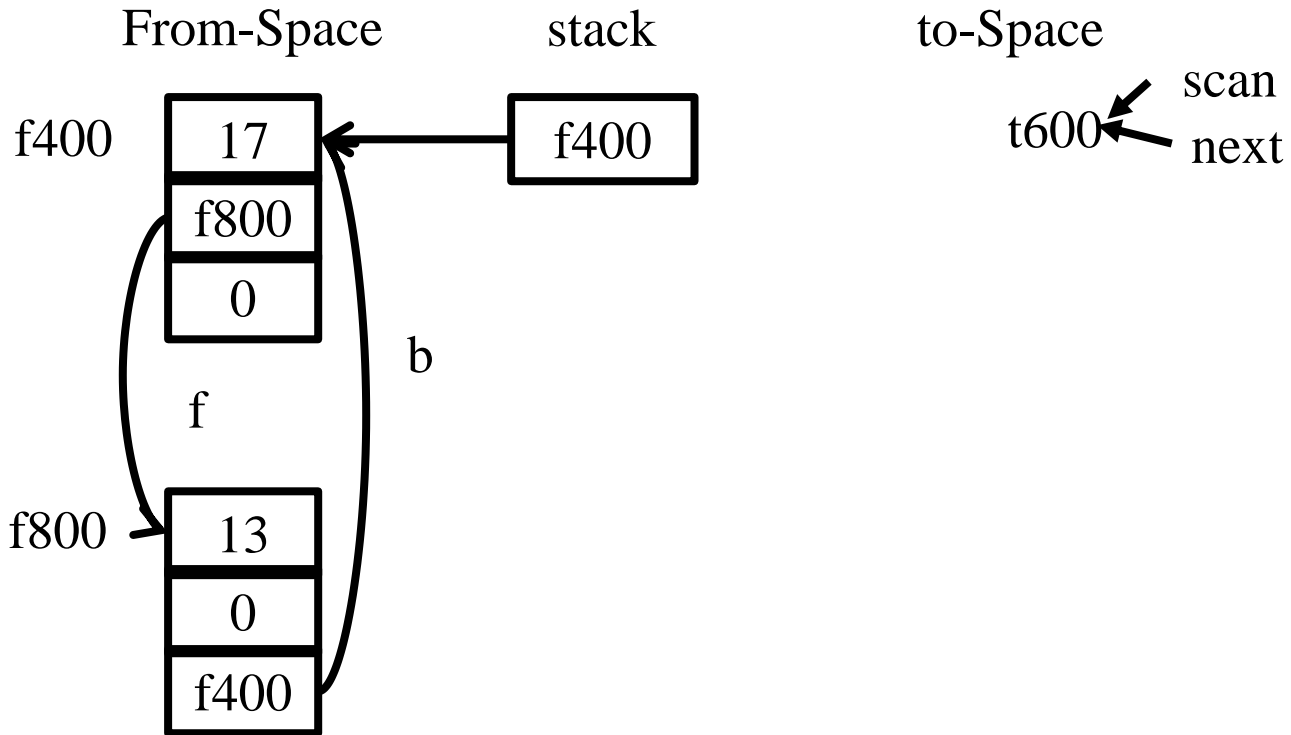
After Forward(f400) before Forward(f800)

From-Space   stack   to-Space

scan

f400 | 17 | | t600 | | 17 | t600

t600   f

0   f800

b   f   0

next

f800 | 13

0
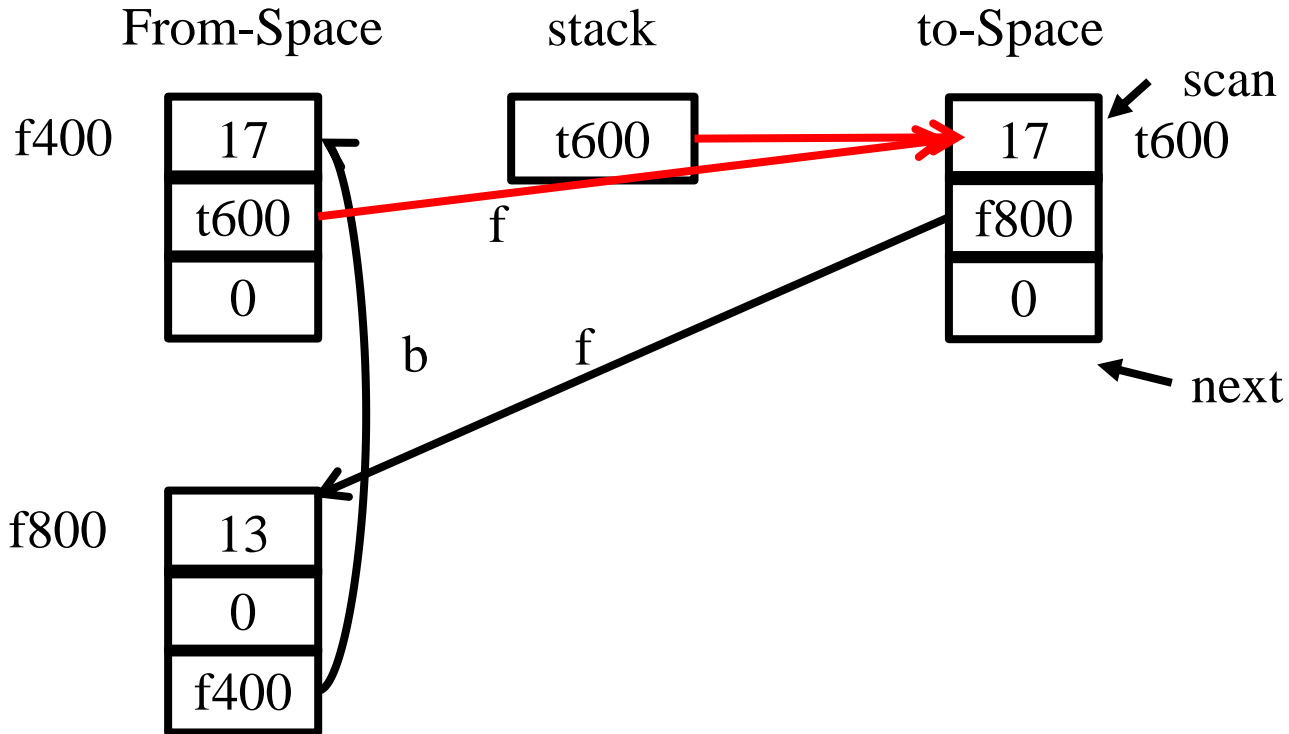
f400

After Forward(f800)
Before Forward(0)

From-Space   stack   to-Space

f400 | 17 | f | t600 | | 17 | scan t600
       | t600| | t612|
       | 0 | | 0 | f
                         b | 13 | t612
                     b
f800 | 13 | | 0 |
       | t612| | f400|
       | f400| next

After Forward(0)
Before Forward(0)

From-Space   stack   to-Space

f400 | 17 | | t600 | | 17 | t600

       | t600 | f | t612 |

       | 0 | b | 0 | f scan

                        b | 13 | t612

f800 | 13 | | 0 |

       | t612 | | f400 |

       | f400 | ← next

After Forward(0)
Before Forward(f400)

From-Space   stack   to-Space

f400

| 17 |
| t600 |
| 0 |

| t600 |

| 17 |
| t612 |
| 0 |

t600

f

b

f
scan
t612

b

f800

| 13 |
| t612 |
| f400 |

| 13 |
| 0 |
| f400 |

next

After Forward(f400)

From-Space   stack   to-Space

f400 | 17 | | t600 | | 17 | t600
        | t600 | f | t612 |
        | 0 | | 0 | f
                        b   b | 13 | t612
                                               | 0 |
f800 | 13 | | t600 | scan
        | t612 | next
        | f400 |

69

12

left

right

15

left

right

link

7

37

left

right

59

left

right

link

9

p

q 37

r

20

left

right

70

12

left

right

left

right

link

7

37

left

right

59

left

right

link

9

15

left

right

p

q 37

r

20

left

right

scan

next

71

12

left

right

left

right

link

7

left

right

59

left

right

link

9

20

left

right

15

left

right

37

left

right

scan

next

p
q 37
r

scan

left

right

left

right

15

left

right

37

left

right

12

left

right

p

q 37

r

37

link

7

37

left

right

59

left

right

link

9

20

left
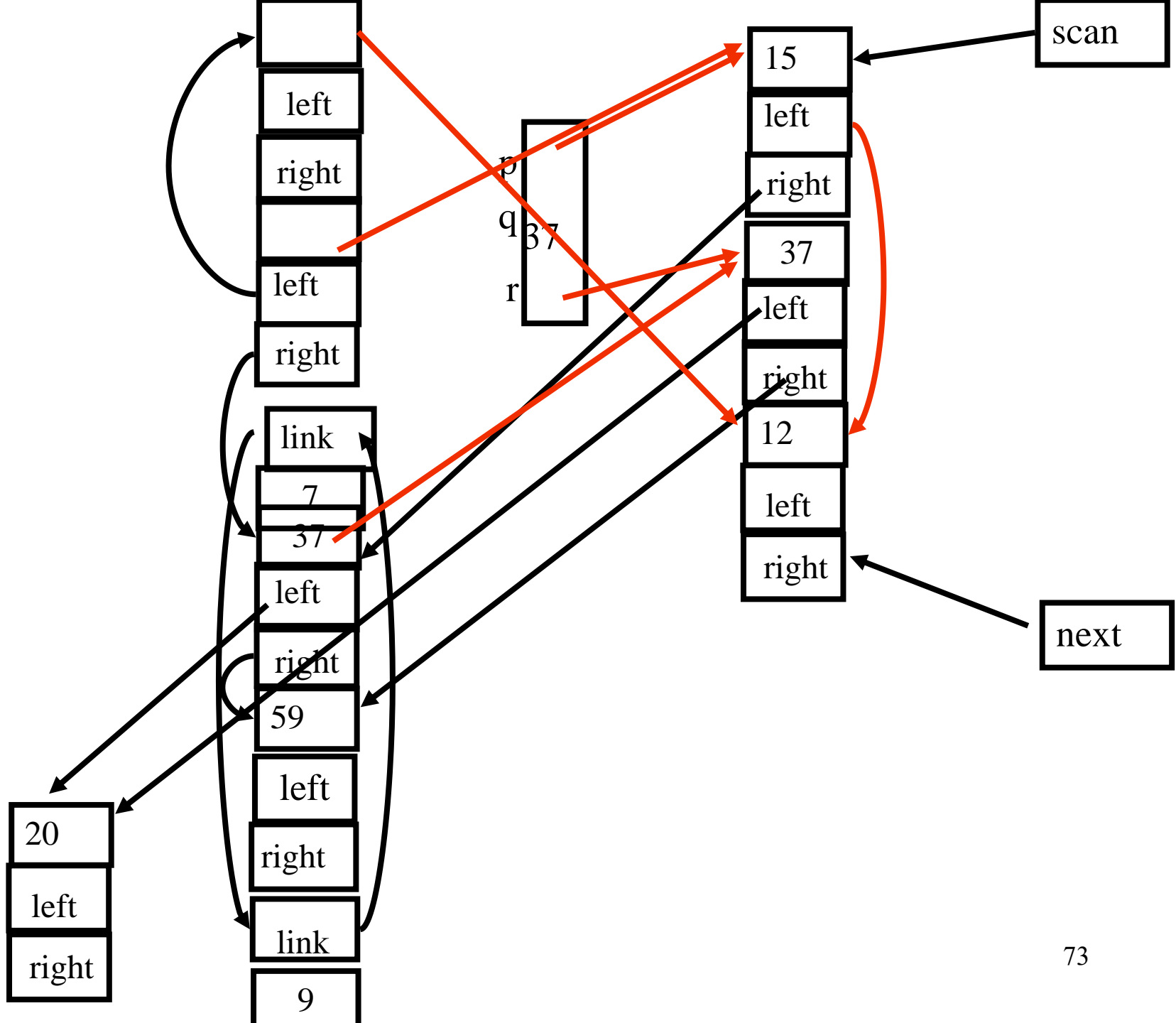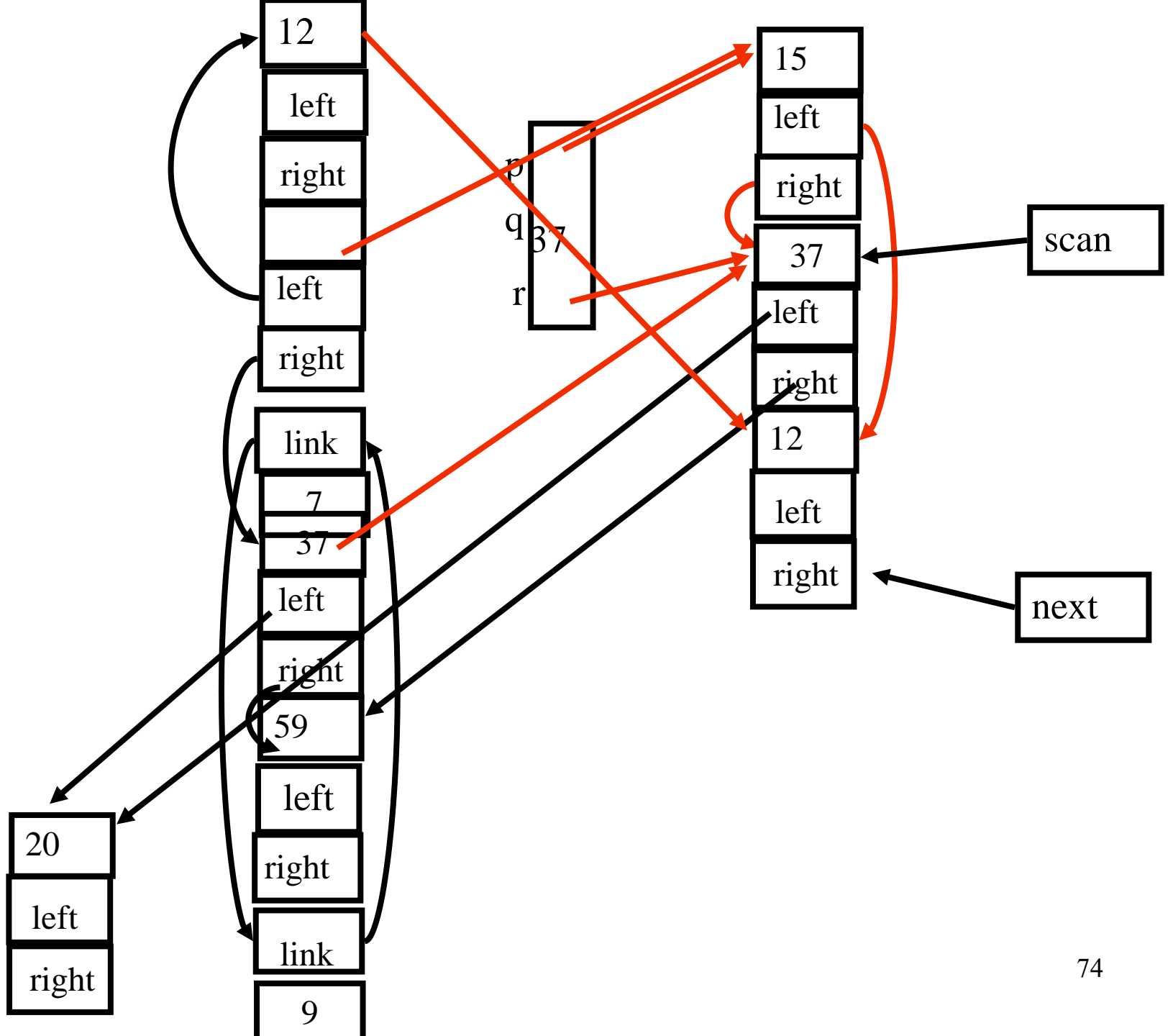
right

next

73

Amortized Cost of Copy Collection

$c_3 R / (H/2 - R)$

Locality of references

- Copy collection does not create fragmentation
- Cheney's algorithm may lead to subfields that point to far away chunks
  - poor virtual memory and cache performance
- DFS normally yields better locality but is harder to implement
- DFS may also be bad for locality for chunks with more than one pointer fields
- A compromise is a hybrid breadth first search with two levels down (Semi-depth first forwarding)
- Results can be improved using dynamic information

The New Forwarding Procedure

function Chase(p)

function Forward(p)   repeat

  if p points to from-space $q := next$

     then if $p.f_1$ points to to-space $next := next + $ size of chunk p

      return $p.f_1$ $r := null$

     else Chase(p); return $p.f_1$ for each reference field f_i of p

  else return p $q.f_i := p.f_i$

             if $q.f_i$ points to from-space and
                   $q.f_i.f_1$ does not point to to-space

                      then $r := q.f_i$

          $p.f_1 := q$

          $p := r$

  until $p = null$

77

Summary Copy Garbage Collection

Pros

- Compact

- Can improve memory locality

- Cost proportional to reachable heap

  - Especially good when large amounts of garbage exist when gc is called

Cons

- Requires type information

- May affect memory locality

Generational Garbage Collection

- Newly created objects contain higher percentage of garbage
- Partition the heap into generations G_1 and G_2
- First garbage collect the G_1 heap
  - chunks which are reachable
- After two or three collections chunks are promoted to G_2
- Once a while garbage collect G_2
- Can be generalized to more than two heaps
- But how can we garbage collect in G_1?

Scanning roots from older generations

- **remembered list**
  - The compiler generates code after each destructive update $b.f_i := a$
    to put b into a vector of updated objects scanned by the garbage collector

- **remembered set**
  - remembered-list + "set-bit"

- **Card marking**
  - Divide the memory into 2^k cards

- **Page marking**
  - k = page size
  - virtual memory system catches updates to old-generations using the dirty-bit

Incremental Collection

- Even the most efficient garbage collection can interrupt the program for quite a while
- Under certain conditions the collector can run concurrently with the program (mutator)
- Need to guarantee that mutator leaves the chunks in consistent state, e.g., may need to restart collection
- Two solutions
  - compile-time
    - Generate extra instructions at store/load
  - virtual-memory
    - Mark certain pages as read(write)-only
    - a write into (read from) this page by the program restart mutator

Tricolor marking

- Generalized GC
- Three kinds of chunks
  - White
    - Not visited (not marked or not copied)
  - Grey
    - Marked or copied but children have not been examined
  - Black
    - Marked and their children are marked

Basic Tricolor marking

while there are any grey objects

  select a grey chunk p

 for each reference field f_i of chunk p

    if chunk $p.f_i$ is white

      color chunk $p.f_i$ grey
color chunk p black

Invariants

•No black points to white

•Every grey is on the collector's (stack or queue) data structure

Establishing the invariants

- Dijkstra, Lamport, et al
  - Mutator stores a white pointer a into a black pointer b
    - color a grey (compile-time)
- Steele
  - Mutator stores a white pointer a into a black pointer b
    - color b grey (compile-time)
- Boehm, Demers, Shenker
  - All black pages are marked read-only
  - A store into black page mark all the objects in this page grey (virtual memory system)
- Baker
  - Whenever the mutator fetches a pointer b to a grey or white object
    - color b grey (compile-time)
- Appel, Ellis, Li
  - Whenever the mutator fetches a pointer b from a page containing a non black object
    - color every object on this page black and children grey (virtual memory system)

Interfaces to the Compiler

- The semantic analysis identifies chunk fields which are pointers and their size
- Generate runtime descriptors at the beginning of the chunks
  - Can employ different allocation/deallocation functions
- Pass the descriptors to the allocation function
- The compiler also passes pointer-map
  - the set of live pointer locals, temporaries, and registers
- Recorded at ?-time for every procedure

Summary

- Garbage collection is an effective technique
- Leads to more secure programs
- Tolerable cost
- But is not used in certain applications
  - Realtime
- Generational garbage collection works fast
  - Emulates stack
- But high synchronization costs
- Compiler can allocate data on stack sometimes
  - Escape analysis
- May be improved