Compiler construction in4303 – lecture 12

Memory Management

Chapter 5

Memory management

Why study memory management?

because it is there!

- Java
- Haskell
- Prolog

} modern programming languages

Implementation:

- runtime support system: garbage collection
- compiler generated hooks to identify pointers

Overview

- explicit deallocation
  - malloc() + free()
- implicit deallocation: garbage collection
  - reference counting
  - mark & scan
  - two-space copying

Data allocation with explicit deallocation

```
#include <stdlib.h>

void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

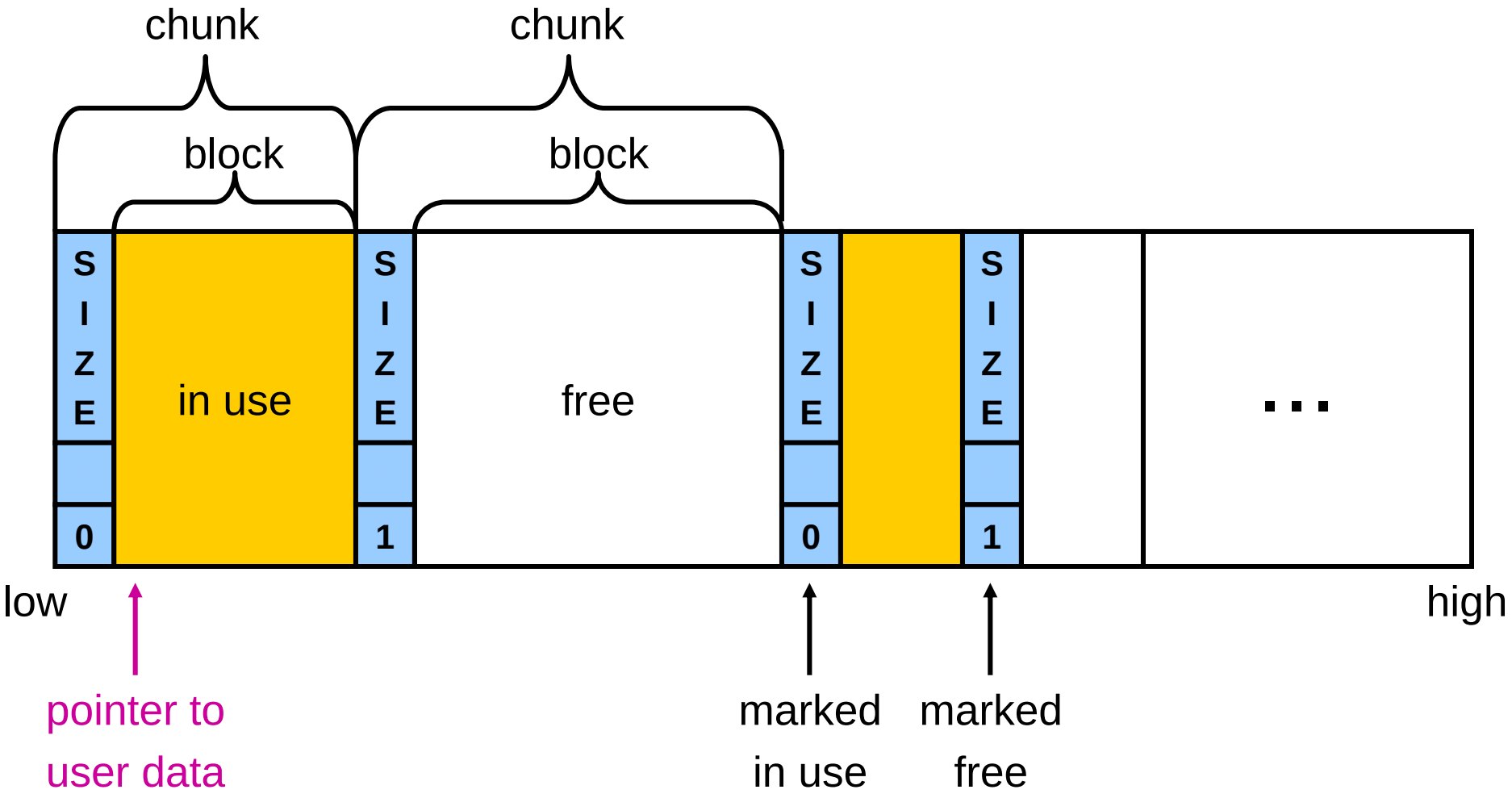
malloc()

- find free block of requested size
- mark it in use
- return a pointer to it.

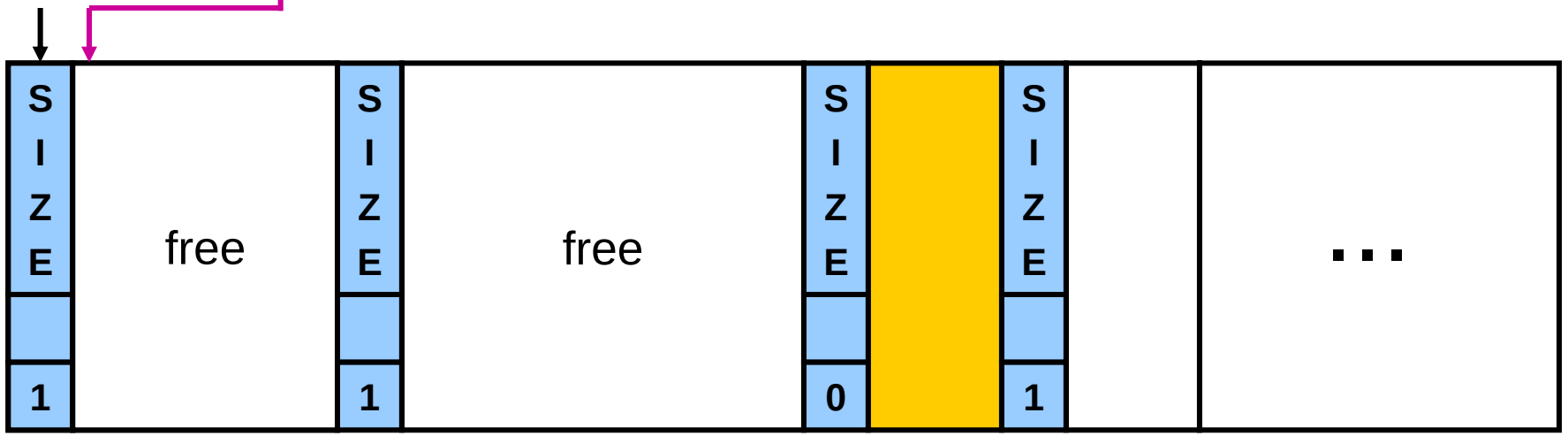
free()

- mark the block as free.

Heap layout

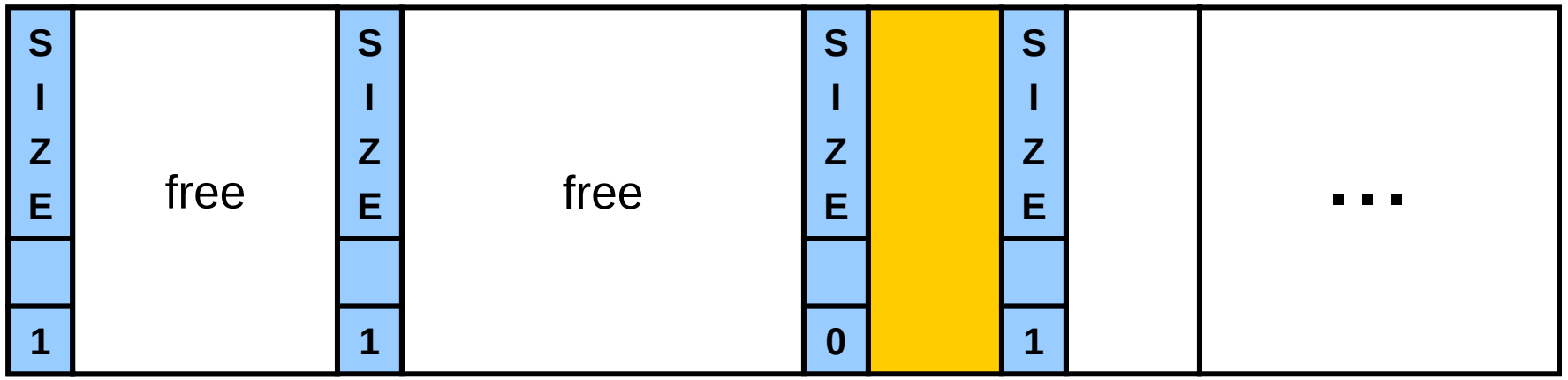

chunk
chunk
block
block
SIZE
in use
SIZE
free
SIZE
SIZE
...
0
1
0
1
low
high
pointer to
user data
marked
in use
marked
free

Free()




```
PROCEDURE Free (Block pointer):
   SET Chunk pointer TO Block pointer – Admin size;
   SET Chunk pointer .free TO True;
```

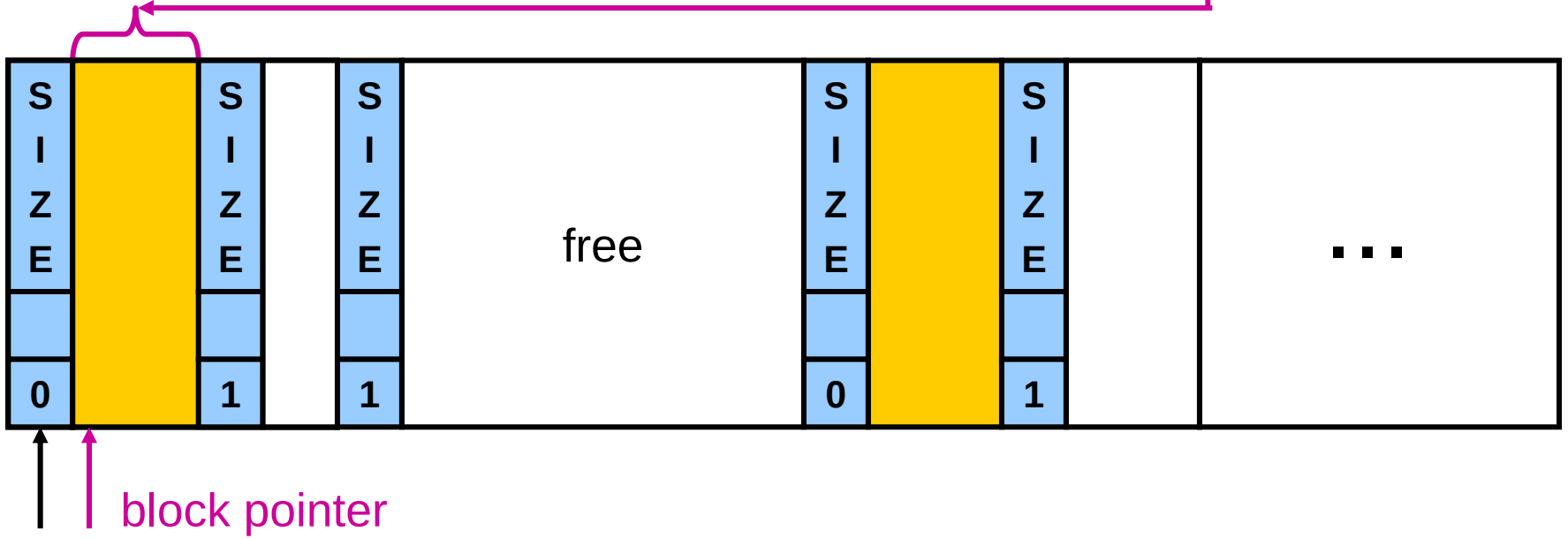
Malloc()




```
FUNCTION Malloc (Block size) RETURNS a generic pointer:
  SET Pointer TO Free block of size (Block size);
  IF pointer /= NULL: RETURN pointer;

  Coalesce free chunks ();
  RETURN Free block of size (Block size);
```

Free block of size (request)



- walk chunks from low to high
- check if chunk is free AND large enough
- if so, mark chunk in use AND return block pointer
- optimization: split chunk to free unused part

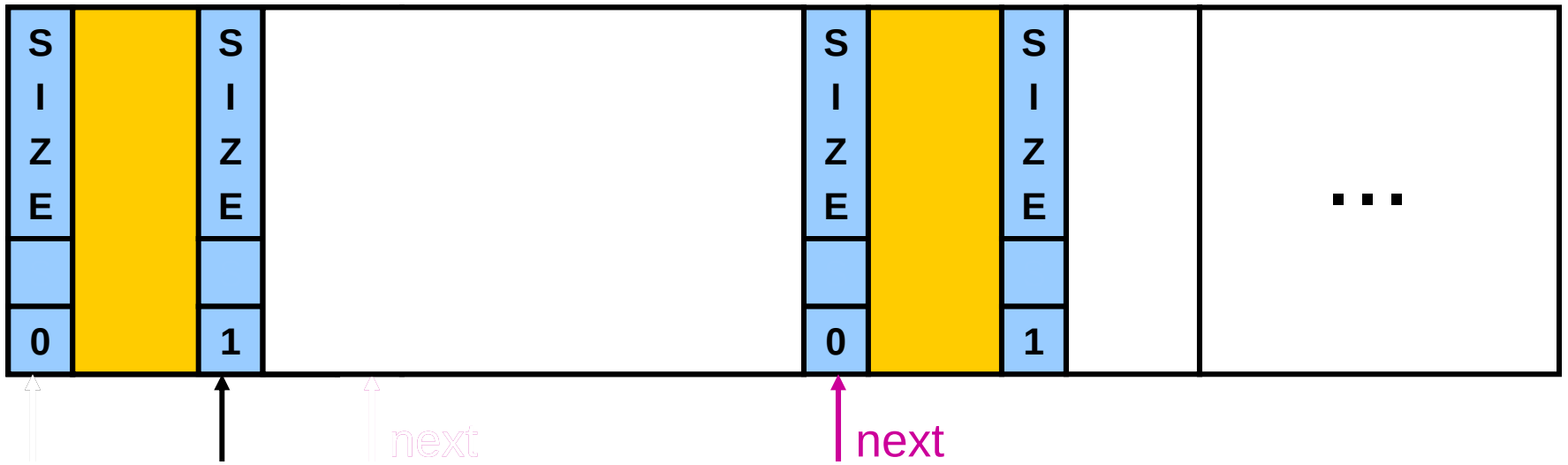
Free block of size

```
FUNCTION Free block of size (Block size)
                                  RETURNS a generic pointer:
  SET Chunk ptr TO Heap low;
  SET Request TO Block size + Admin size;

  WHILE Chunk ptr < Heap high:
    IF Chunk ptr .free AND Chunk ptr .size >= Request:
      Split chunk (Chunk ptr, Request)
      SET Chunk ptr .free TO False;
      RETURN Chunk ptr + Admin size;

    SET Chunk ptr TO Chunk ptr + Chunk ptr .size;
  RETURN NULL;
```

Coalesce free chunks ()



- walk chunks from low to high
- check if chunk is free
- if so, coalesce all subsequent free chunks

Coalesce free chunks

```
PROCEDURE Coalesce free chunks ():
  SET Chunk ptr TO Heap low;

  WHILE Chunk ptr < Heap high:
    IF Chunk ptr .free:
      SET Next TO Chunk ptr + Chunk ptr .size;
      WHILE Next < Heap high AND Next .free:
        SET Next TO Next + Next .size;
      SET Chunk ptr .size TO Next - Chunk ptr;

    SET Chunk ptr TO Chunk ptr + Chunk ptr .size;
```

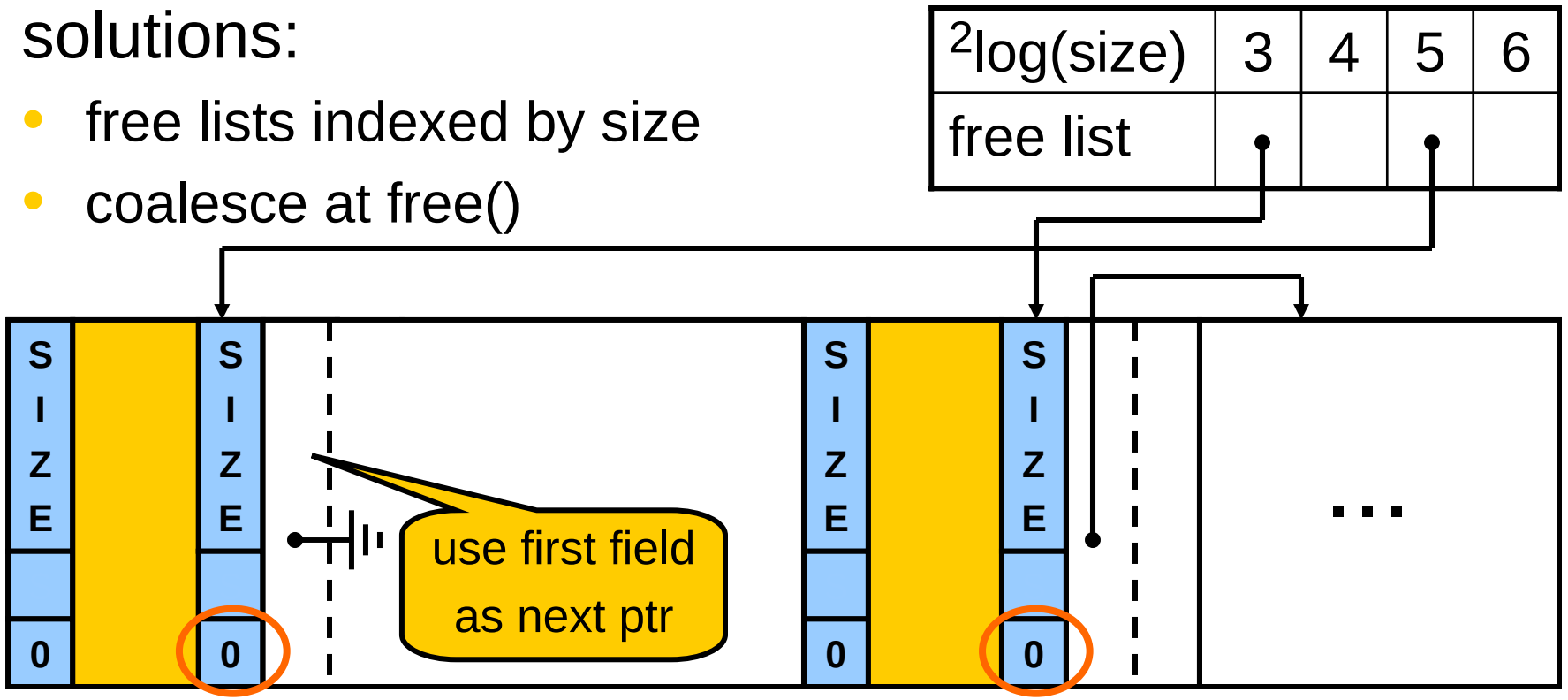
Optimizations

free block of size: poor performance (linear search)

malloc: irregular performance (coalesce phase)

solutions:

- free lists indexed by size
- coalesce at free()



Malloc() with free lists

```
FUNCTION Malloc (Block size) RETURNS a generic pointer:
  SET Chunk size TO Block size + Admin size;
  SET Index TO 2log(Chunk size);

  IF Index < 3:
    SET Index TO 3;

  IF Index <= 10 AND Free list[Index] /= NULL:
    SET Pointer TO Free list[Index];
    SET Free list[Index] TO Pointer .next;
    RETURN Pointer + Admin size;

  RETURN Free block of size (Block size);
```

Exercise (5 min.)

- give the pseudo code for free() when using free lists indexed by size.

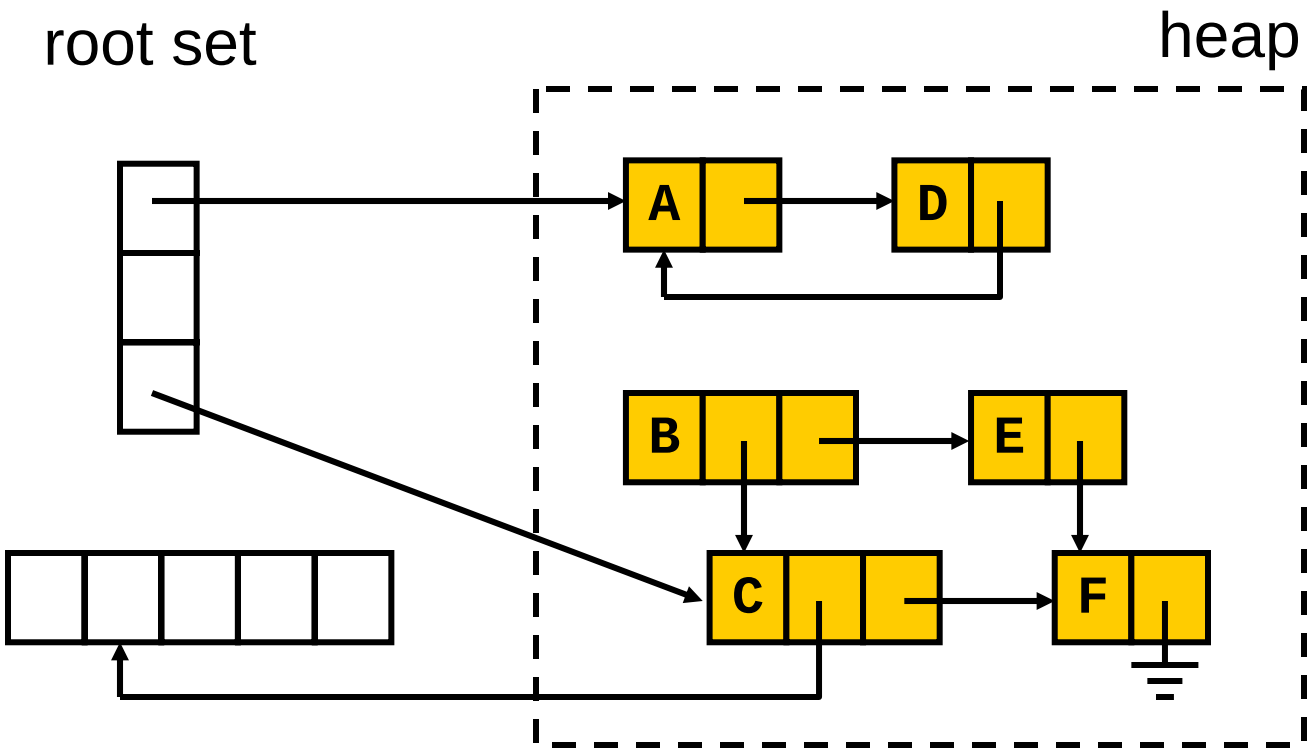
Answers

Garbage collection

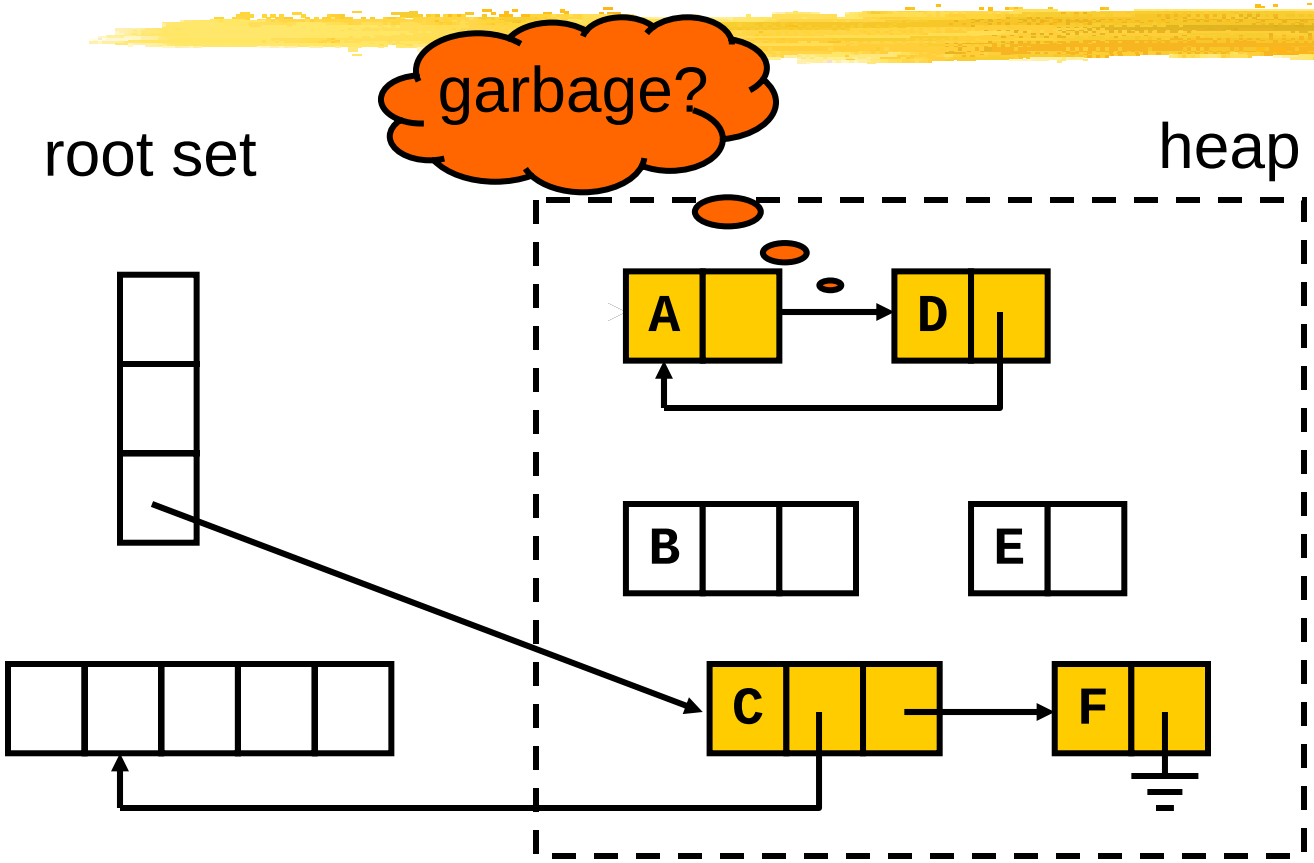
- memory allocation is explicit (new)
- memory deallocation is implicit

- garbage set: all chunks that will no longer be used by the program
  - chunks without incoming pointers
  - chunks that are unreachable from non-heap data

Example



Cyclic garbage



- “no-pointers”: NO
- “not-reachable”: YES

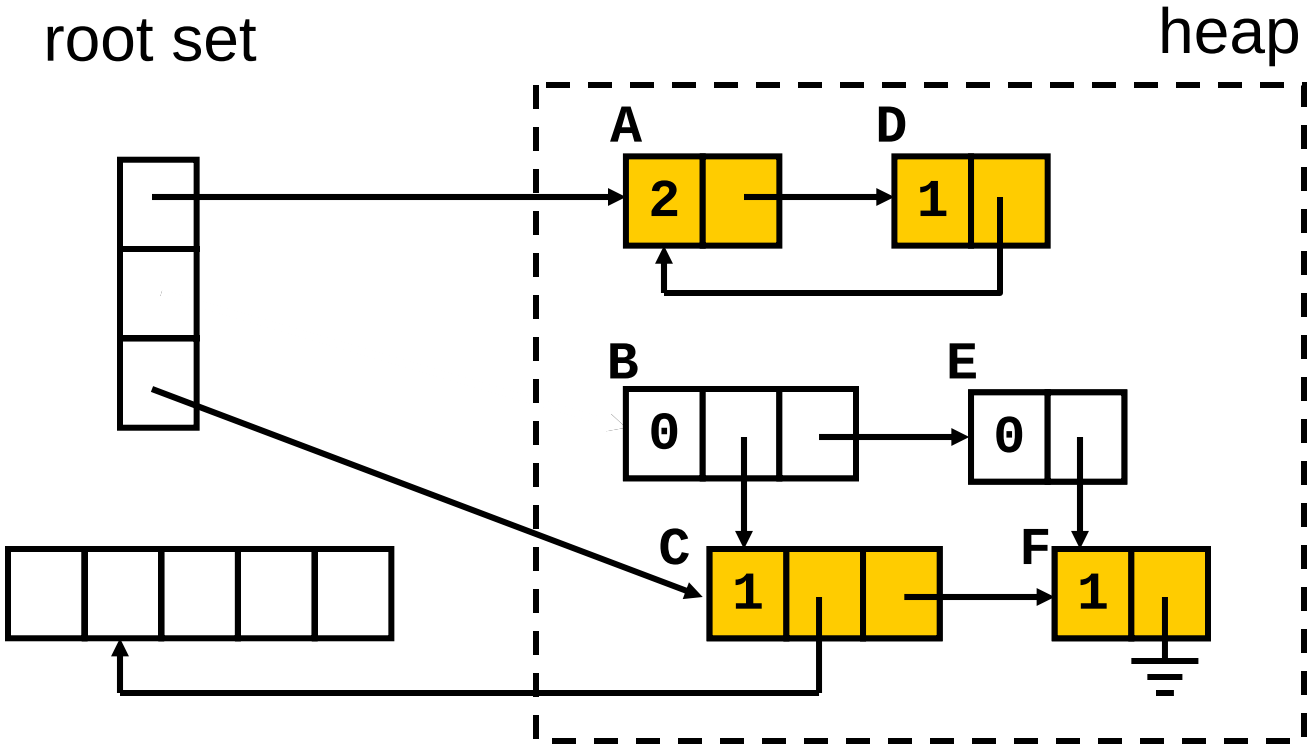
Compiler assistance: identifying pointers

- pointers inside chunks
  - user-defined data structures
  - compiler: generate self-descriptive chunks
- pointers located outside the heap (root set)
  - global data + stack
  - compiler: generate activation record descriptions

Self-descriptive chunks

- bitmap per data type
  - problem: overhead per chunk / interpretation
- compiler-generated routine per data type
  - calls GC for each pointer
  - problem: recursion
- organize data type to start off with *n* pointers
  - solution: *n* can be squeezed into chunk admin

Reference counting



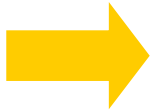
- record #pointers to each chunk
- reclaim when reference count drops to zero

Maintaining reference counts

pointer assignment:

```
VAR p, q : pointer;
...
p := q;
```

source

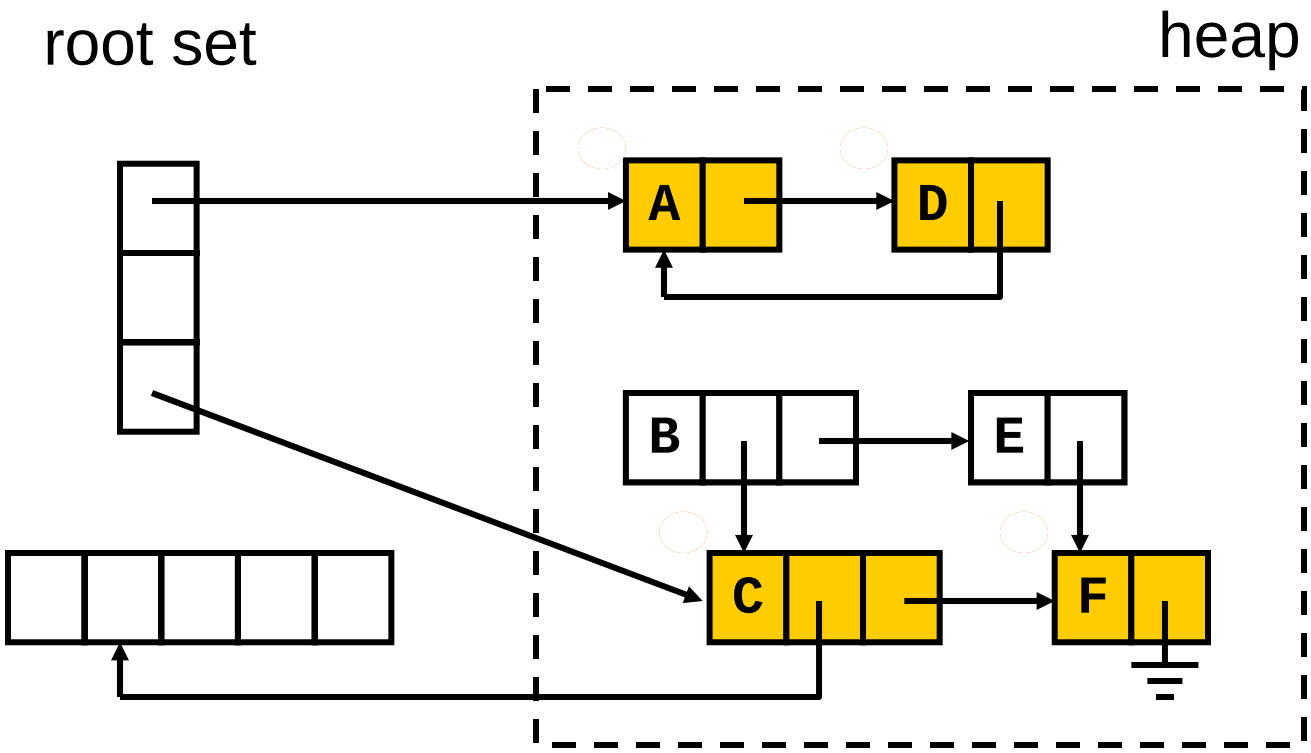


```
IF Points into the heap (q):
  Increment q .ref count;
IF Points into the heap (p):
  Decrement p .ref count;
  IF p .ref count = 0:
    Free recursively (p);
SET p TO q;
```

target

```
PROCEDURE Free recursively (Pointer):
  FOR each field f_i of record Pointer:
    IF Points into the heap (f_i):
      Decrement f_i .ref count;
      IF f_i .ref count = 0:
        Free recursively (f_i);
  Free chunk (Pointer);
```

Mark & scan



- mark all reachable chunks
- scan heap for unmarked chunks that can be freed

```
PROCEDURE Mark (Pointer):
  IF NOT Points into the heap (Pointer): RETURN;
  IF Pointer .marked: RETURN;
  SET Pointer .marked TO True;
  FOR each field f_i of record Pointer:
    Mark (f_i);
```

```
PROCEDURE Scan ():
  SET Chunk ptr TO Heap low;

  WHILE Chunk ptr < Heap high:
    IF Chunk ptr .marked:
      SET Chunk ptr .marked TO False;
    ELSE
      SET Chunk ptr .free TO True;

    SET Chunk ptr TO Chunk ptr + Chunk ptr .size;
```

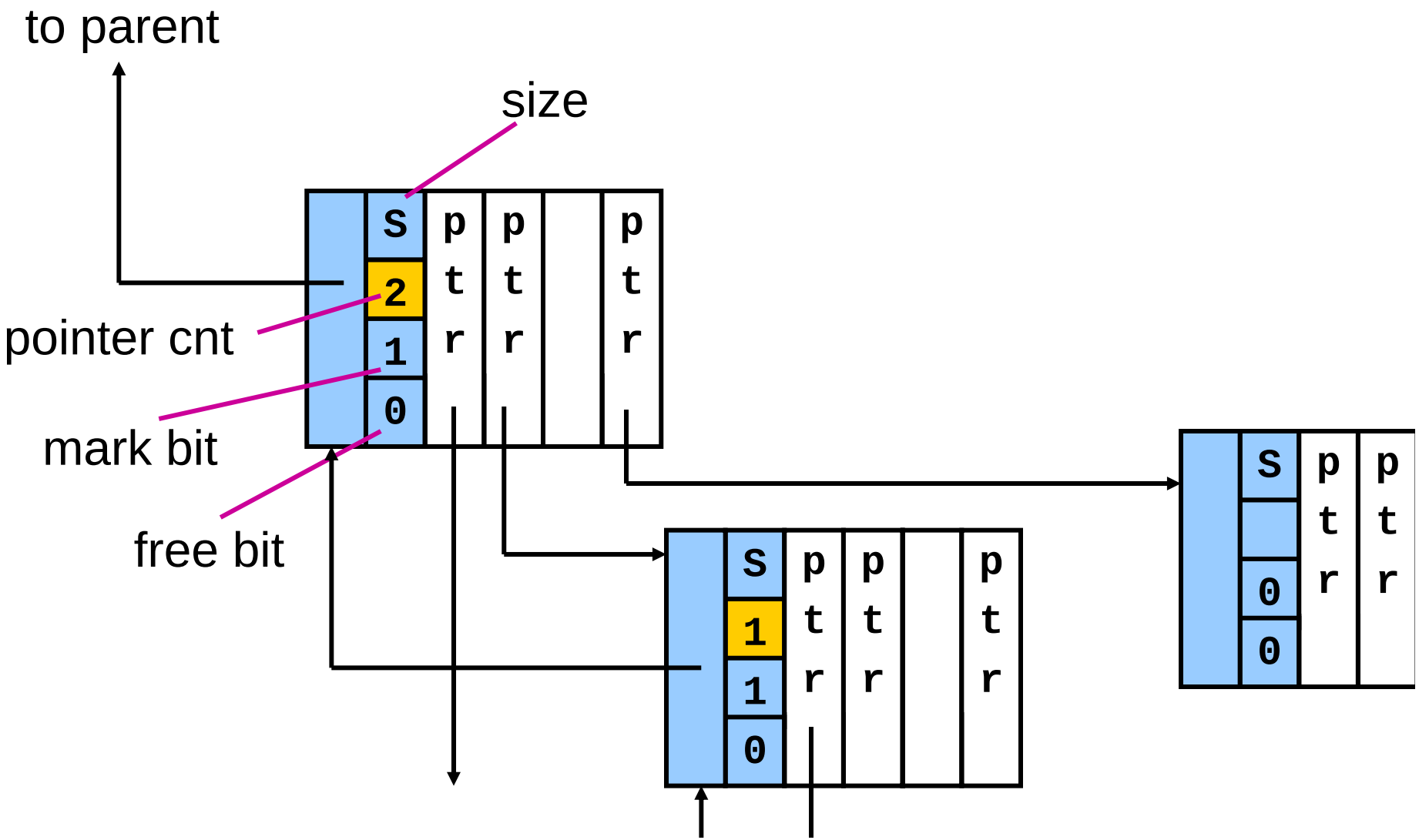
Advanced marking

- problem: mark() is recursive
- solution: embed stack in the chunks

each chunk records:

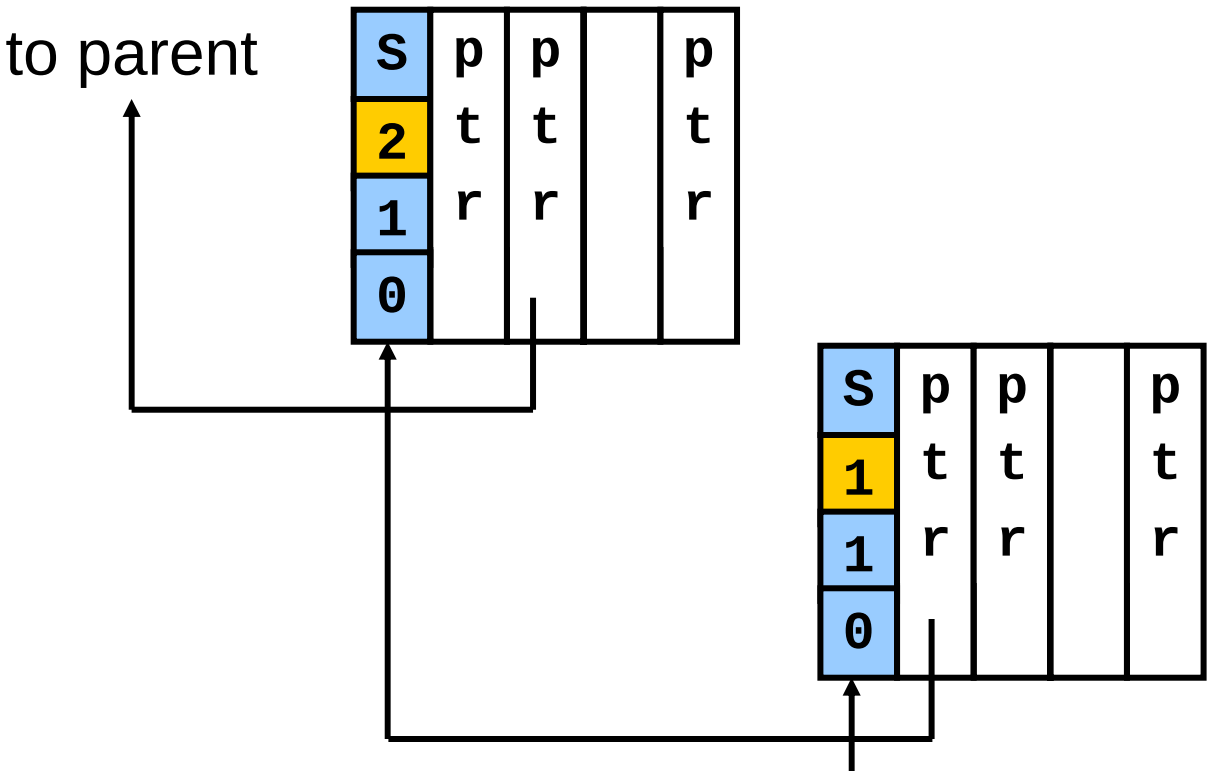
- a count denoting which child pointer is next
- a pointer to the parent node

Advanced marking



Pointer reversal [Schorr & Waite]

- avoid additional parent pointer
- use the *n*-th child pointer when visiting child *n*

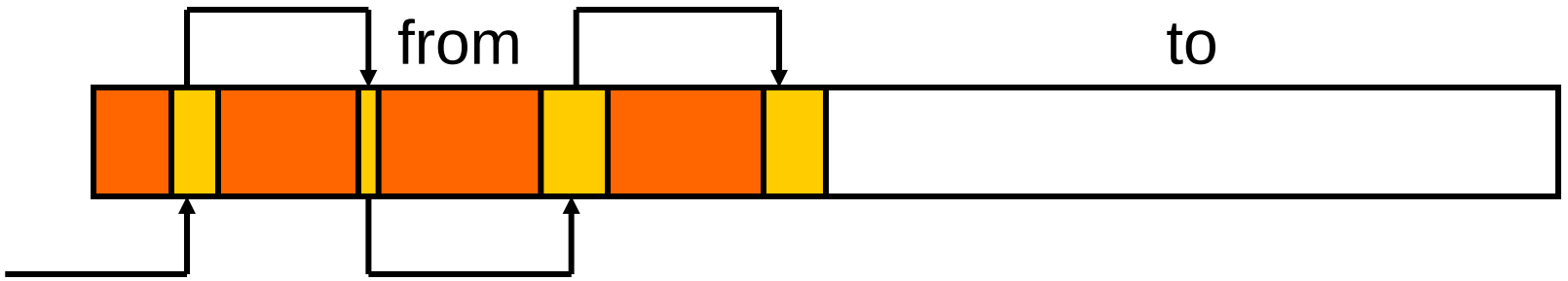


Two-space copying

- most chunks have a short live time
- memory fragmentation must be addressed

> copy all reachable chunks
> to consecutive locations

- partition heap in two spaces

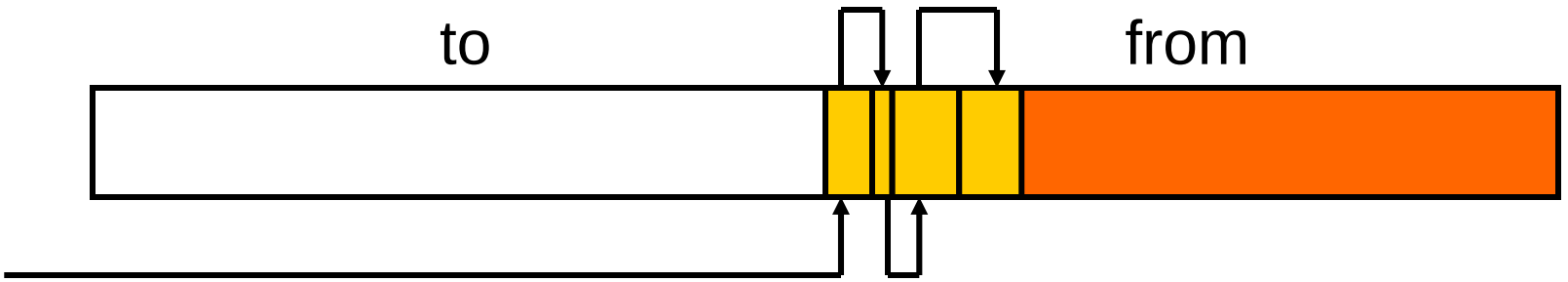


Two-space copying

- most chunks have a short live time
- memory fragmentation must be addressed

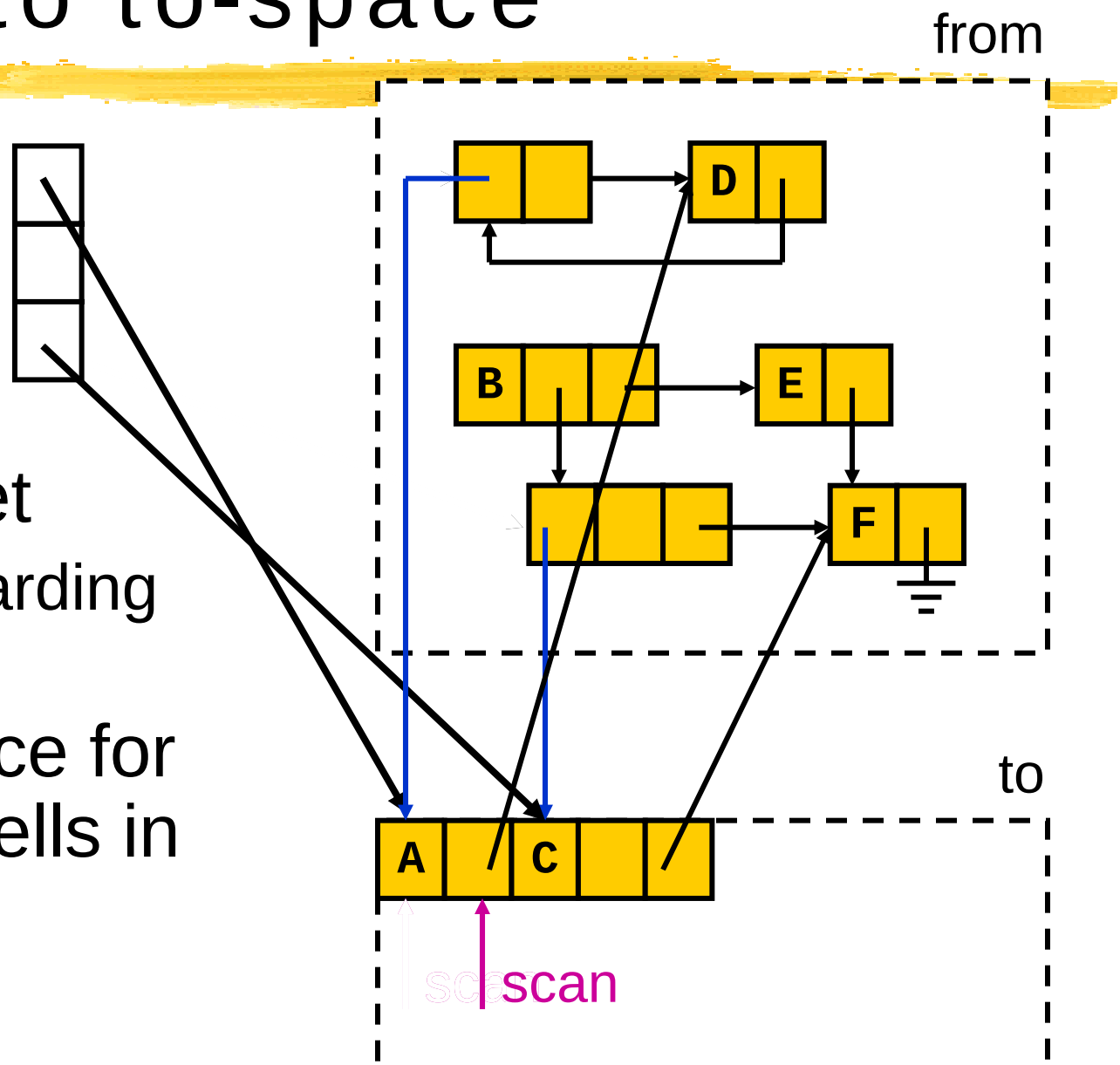
> copy all reachable chunks
> to consecutive locations

- partition heap in two spaces



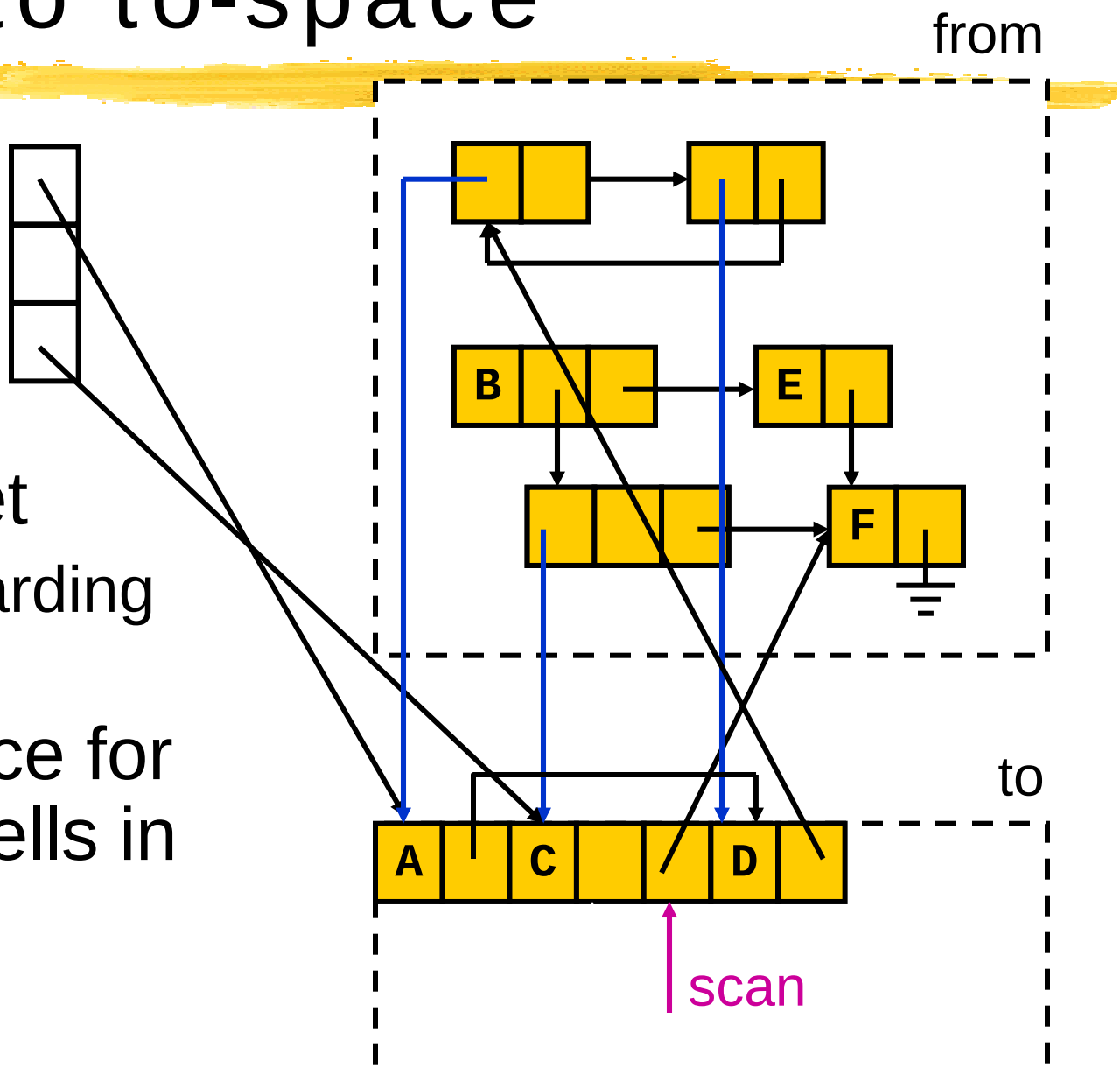
Copying to to-space

- copy root set
  - leave forwarding pointers
- scan to-space for reachable cells in from-space



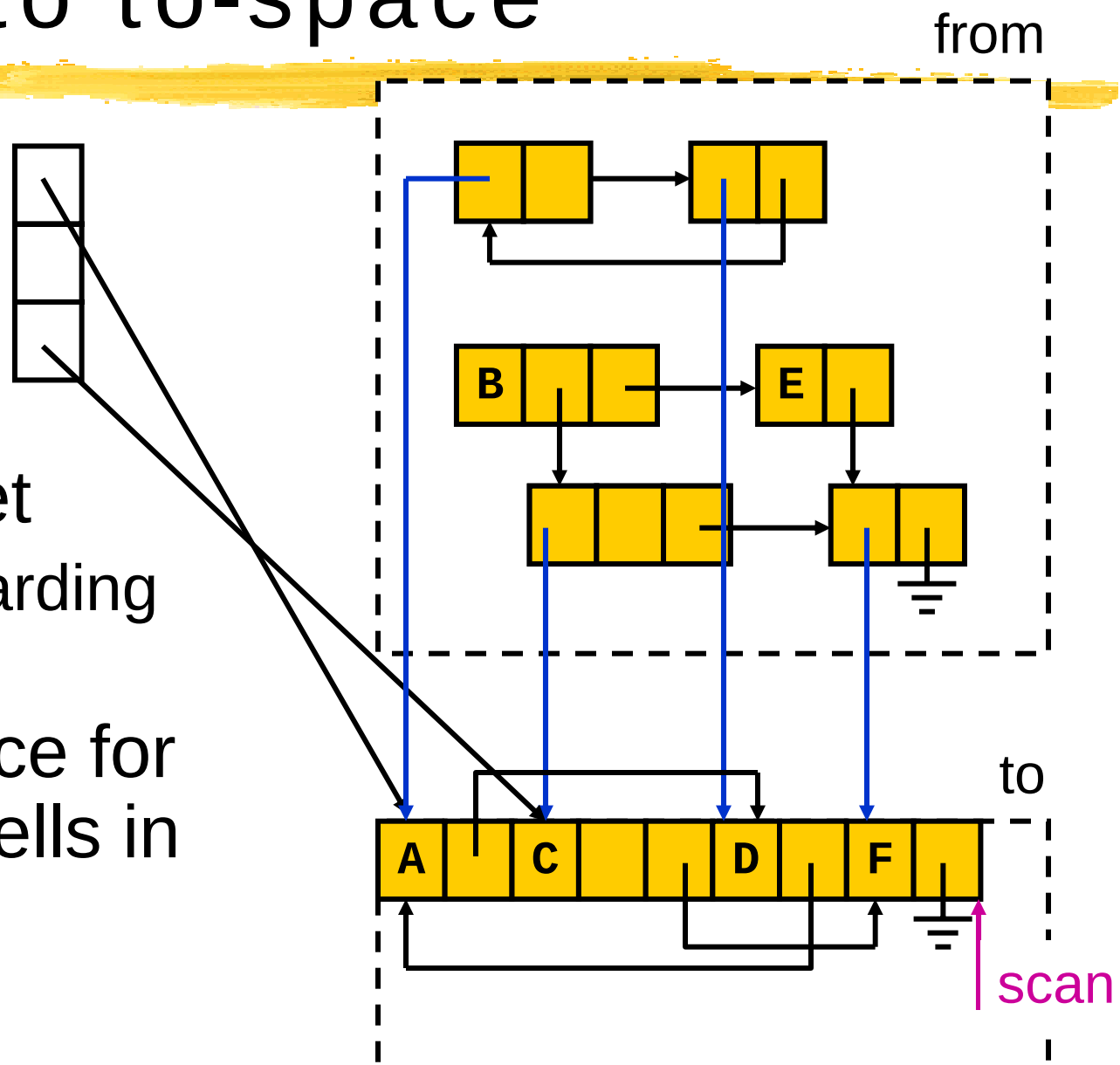
Copying to to-space

- copy root set
  - leave forwarding pointers
- scan to-space for reachable cells in from-space



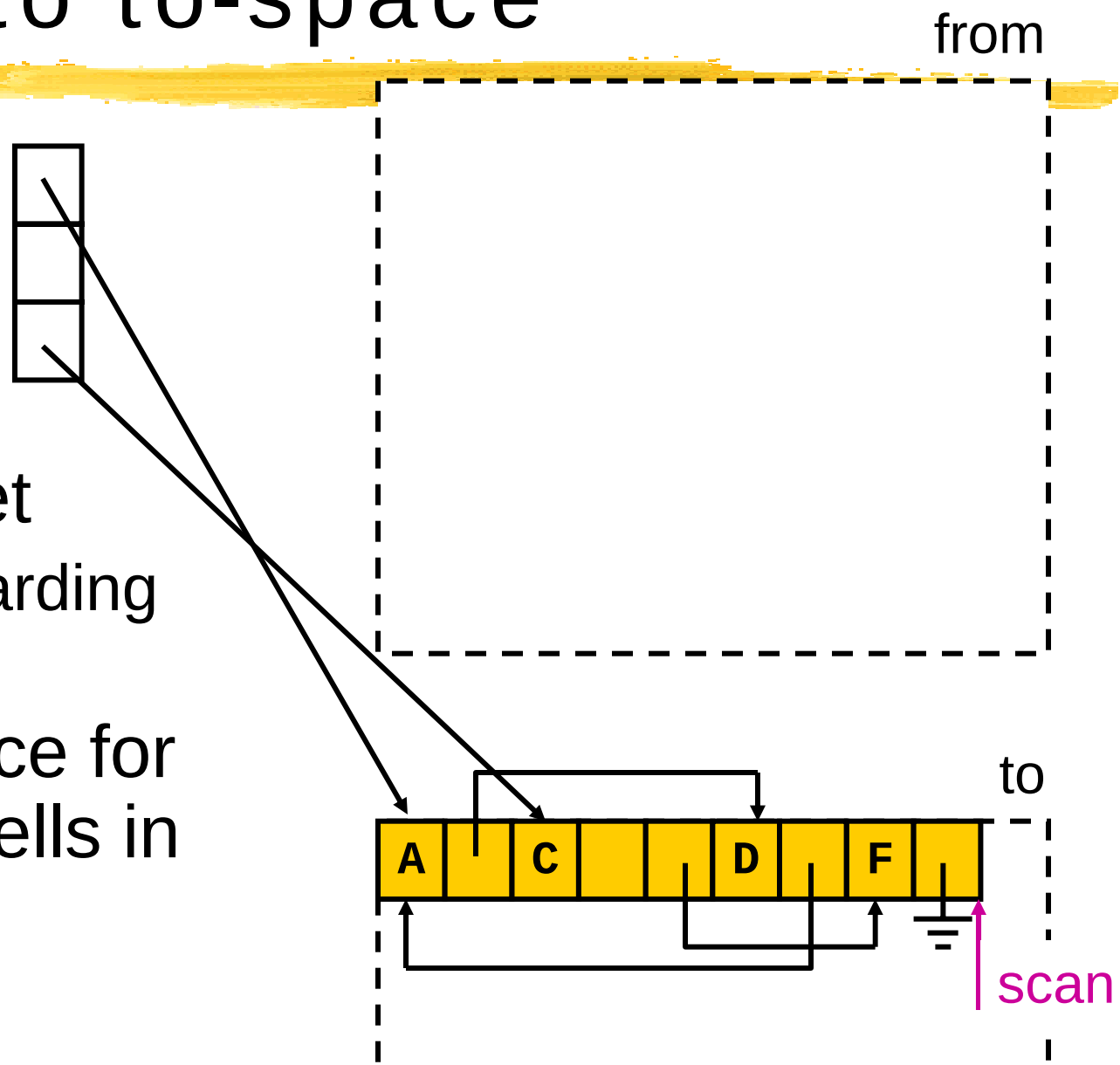
Copying to to-space

- copy root set
  - leave forwarding pointers
- scan to-space for reachable cells in from-space



Copying to to-space

- copy root set
  - leave forwarding pointers
- scan to-space for reachable cells in from-space



Garbage collection: summary

Tasks

- compiler: identify pointers (root set + chunk layout)
- runtime support: reclaim non-reachable nodes

Techniques

	pro	con
reference counting	- simple	- update per ptr manipulation - can't reclaim cyclic data
mark & scan	- overhead per run - reclaims cyclic data	- touches all nodes - memory fragmentation
two-space copying	- touches only live nodes - automatic compaction	- "wastes" halve of the heap