Garbage Collection

Tobias Wrigstad

Slides (slightly mutilated) from Vitaly Shmatikov

Major Areas of Memory

- Static area
  - Fixed size, fixed content, allocated at compile time
- Stacken (one per thread/process)
  - Variable size, variable content (activation records)
  - Used for managing function calls and returns
- <u>Heap</u>
  - Fixed size, variable content
  - Dynamically allocated objects and data structures
    - Examples: ML reference cells, <u>malloc</u> in C, <u>new</u> in Java

Cells and Liveness

- Cell = data item in the heap
  - Cells are “pointed to” by pointers held in registers, stack, global/static memory, or in other heap cells
- Roots: registers, stack locations, global/static variables
- A cell is live if its address is held in a root or held by another live cell in the heap

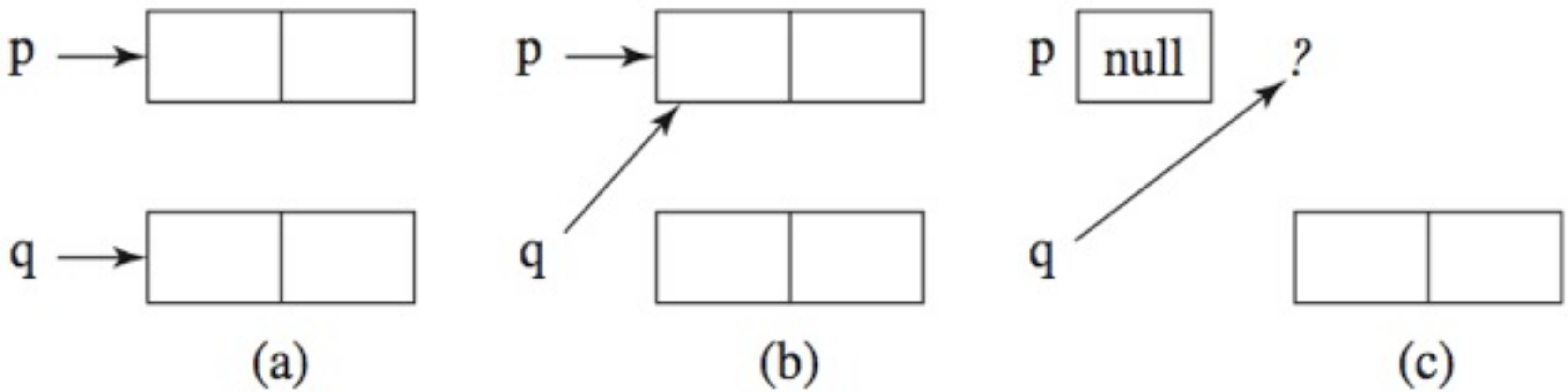
Garbage

- Garbage is a block of heap memory that cannot be accessed by the program
  - An allocated block of heap memory does not have a reference to it (cell is no longer “live”)
  - Another kind of memory error: a reference exists to a block of memory that is no longer allocated
- Garbage collection (GC) - automatic management of dynamically allocated storage
  - Reclaim unused heap blocks for later use by program

Example of Garbage

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

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



Why Garbage Collection?

- We have heaps of memory available to our programs, which they use
- … badly
  - Memory leaks, dangling references, double free, misaligned addresses, null pointer dereference, heap fragmentation
- **Also:** Explicit memory management breaks high-level programming abstraction
  - I must expose how a function works internally so that its users can manage memory correctly

GC and Programming Languages

- GC is not necessarily a language feature
- ”A pragmatic concern for automatic and efficient heap management”
  - GC’d PLs: Lisp, Scheme, Prolog, Smalltalk …
  - Manual MM: C, C++, Objective-C
    - But garbage collection libraries have been built for them
- GC revival started in the 90’s
  - Object-oriented languages: Java, Python, Ruby, C#
  - Functional languages: ML, Haskell, Erlang, Clojure

The Perfect Garbage Collector

- No visible impact on program execution
- Works with any program and its data structures
- Manages the heap efficiently
  - Always satisfies an allocation request and does not fragment

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

Reference Counting

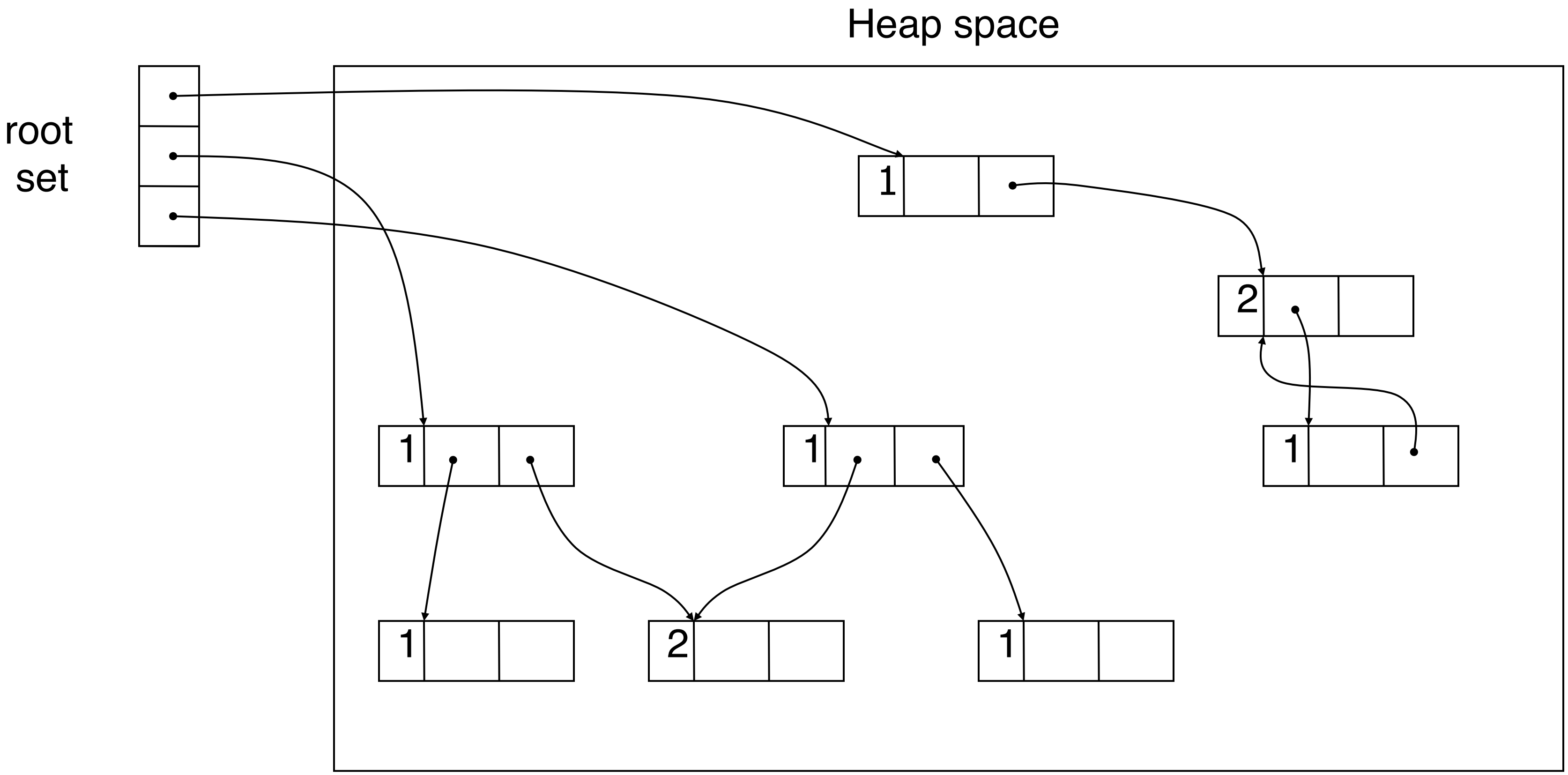
- Simply count the number of references to a cell
- Requires space and time overhead to store the count and increment (decrement) each time a reference is added (removed)
  - Reference counts are maintained in real-time, so no "stop-and-gag" effect
  - Incremental garbage collection
- Unix file system uses a reference count for files
- C++ "smart pointer" (e.g., auto_ptr) use reference counts

Spot the memory leak(s)

```
-(void) setName: (NSString*) _name {
    [name release];
    [_name retain];
    name = _name;
}


/* In the class Node */
-(void) connect: (Node*) n {
    if (node == nil) {
        [n retain];
        node = n;
        [node connect: self];
    }
}
```

Reference Counting: Example



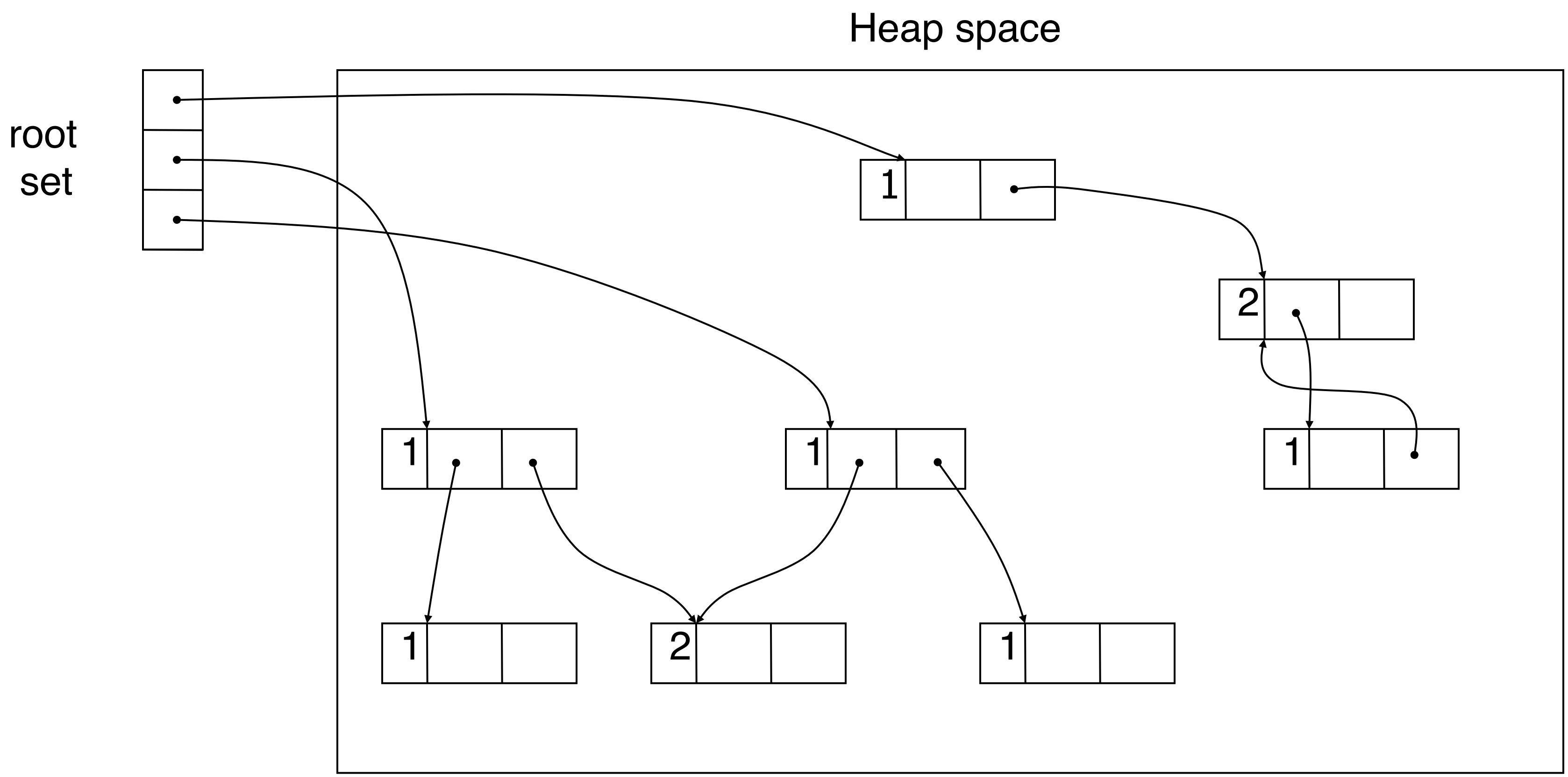
Reference Counting: Strengths

- Incremental overhead
  - Cell management interleaved with program execution
  - Good for interactive or real-time computation
- 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

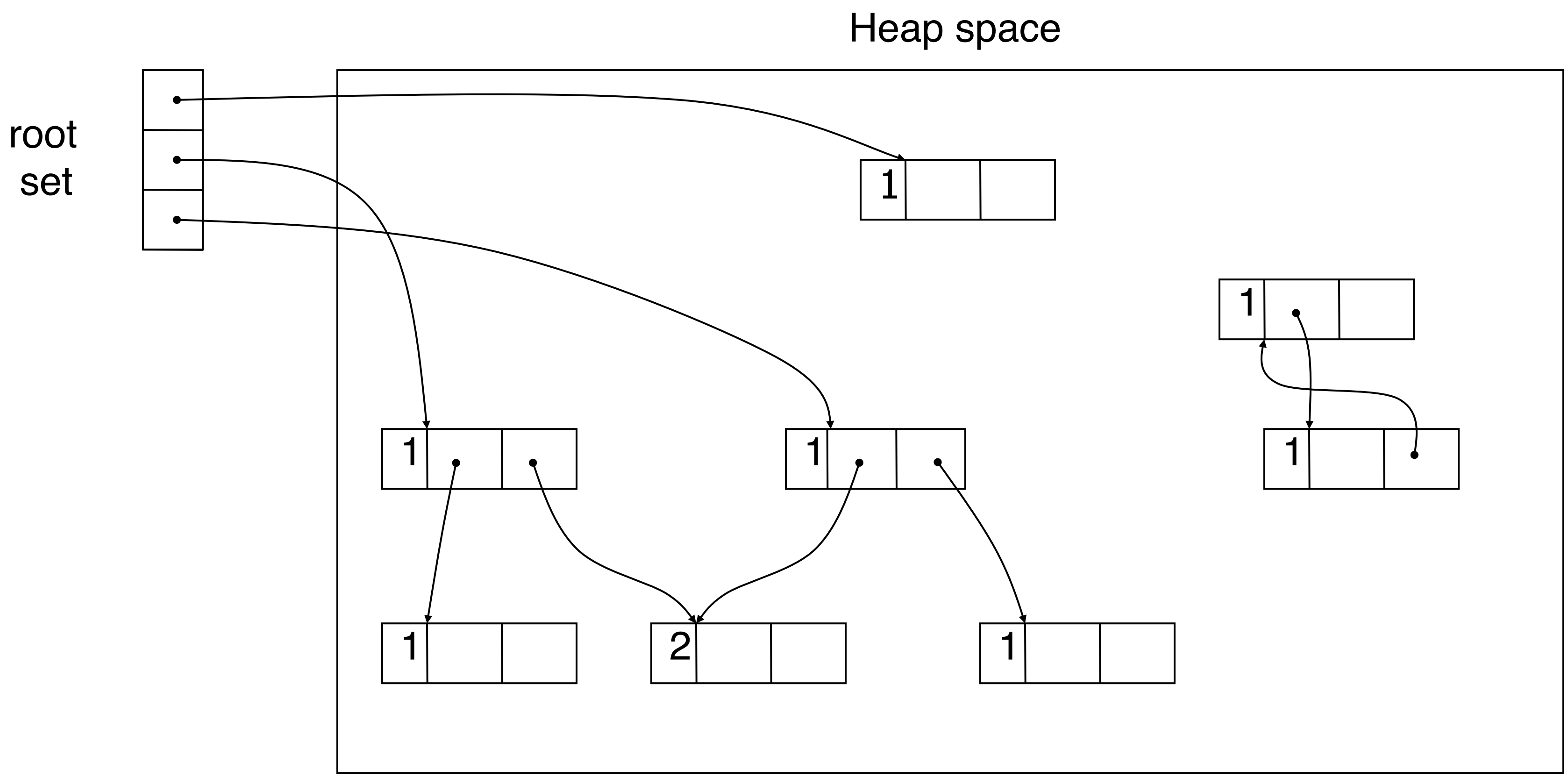
Reference Counting: Weaknesses

- Space overhead
  - 1 word for the count
- Time overhead
  - Updating a pointer to point to a new cell requires:
    - Check to ensure that it is not a self-reference
    - Decrement the count on the old cell, possibly deleting it
    - Update the pointer with the address of the new cell
    - Increment the count on the new cell
- One missed increment/decrement results in a dangling pointer / memory leak
- Cyclic data structures may cause leaks
- Very expensive in concurrent || parallel programs

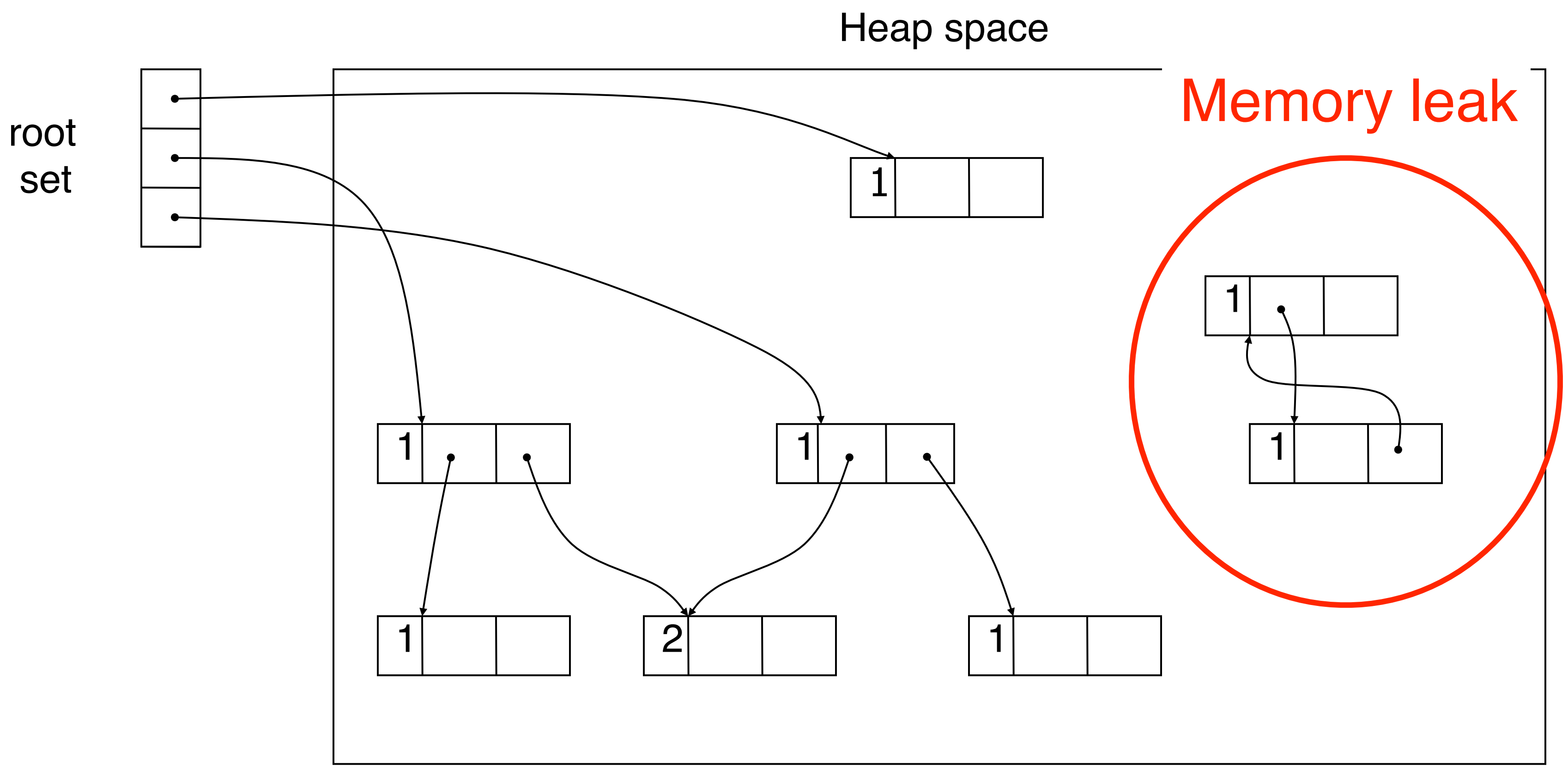
Reference Counting: Example



Reference Counting: Cycles



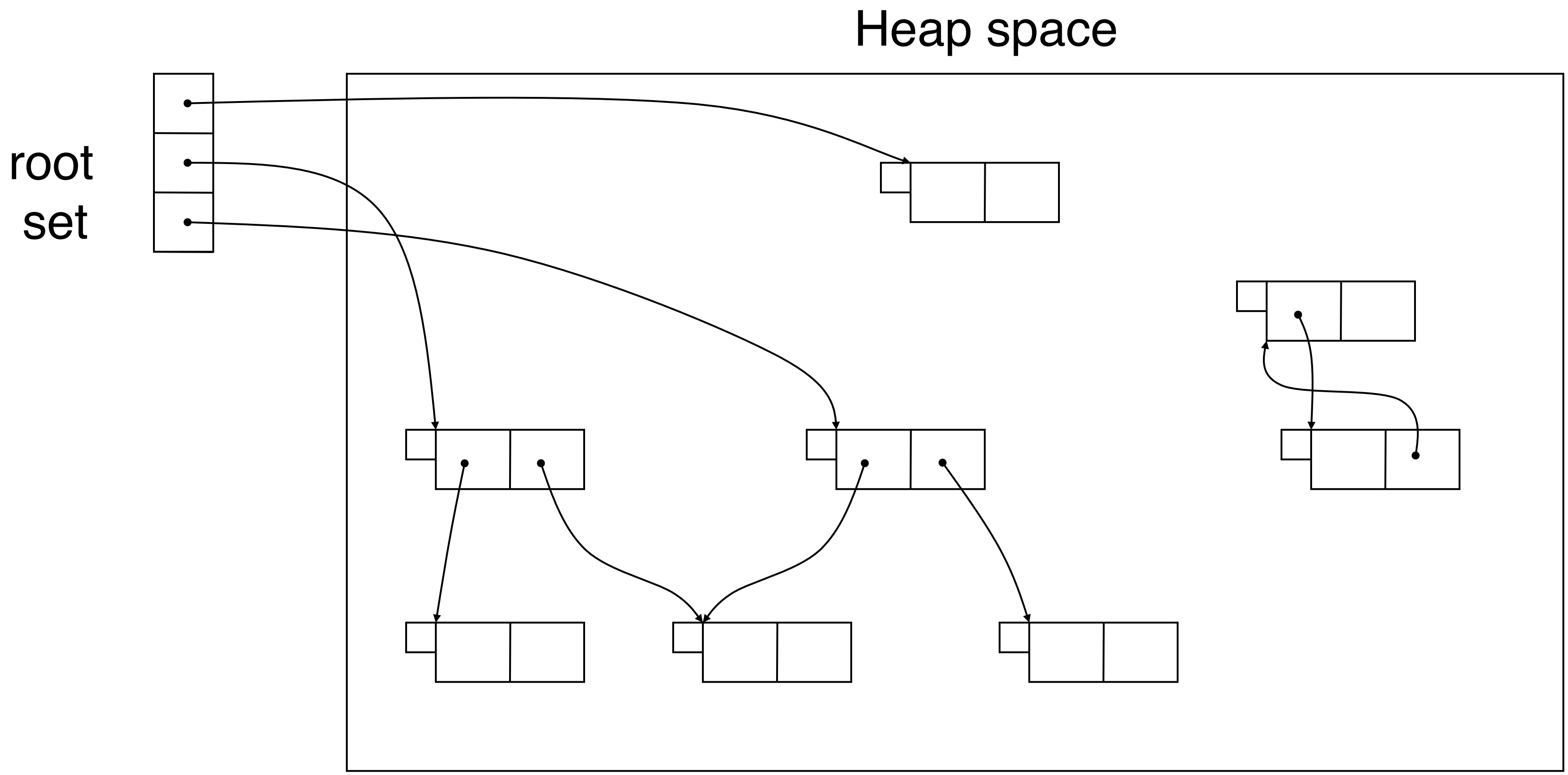
Reference Counting: Cycles



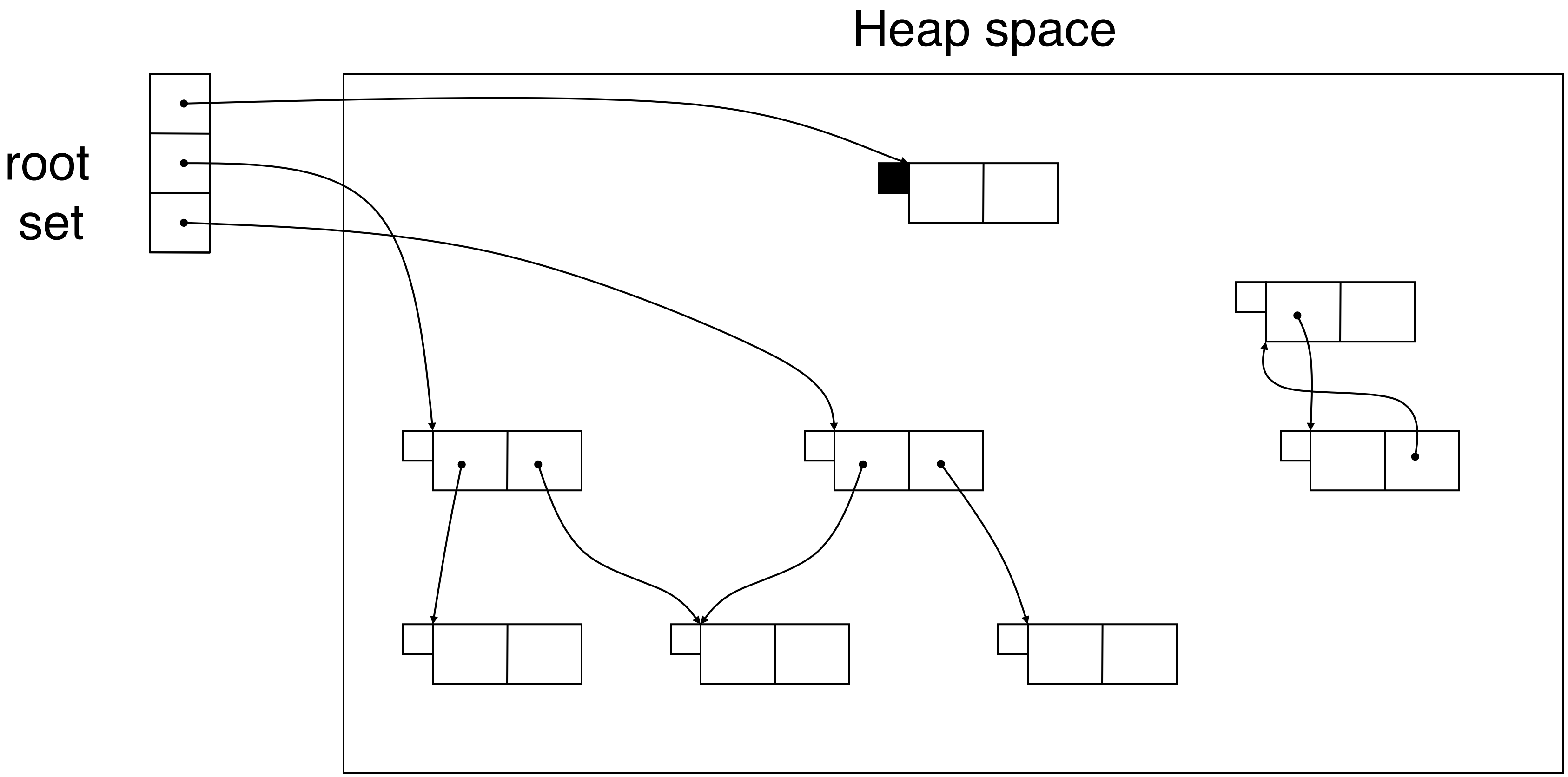
Mark-Sweep Garbage Collection

- Each cell has a <u>mark bit</u>
- Garbage remains unreachable and undetected until heap is used up; then GC goes to work, while program execution is suspended
- Marking phase
  - Starting from the roots, set the mark bit on all live cells
- Sweep phase
  - Return all unmarked cells to the free list
  - Reset the mark bit on all marked cells

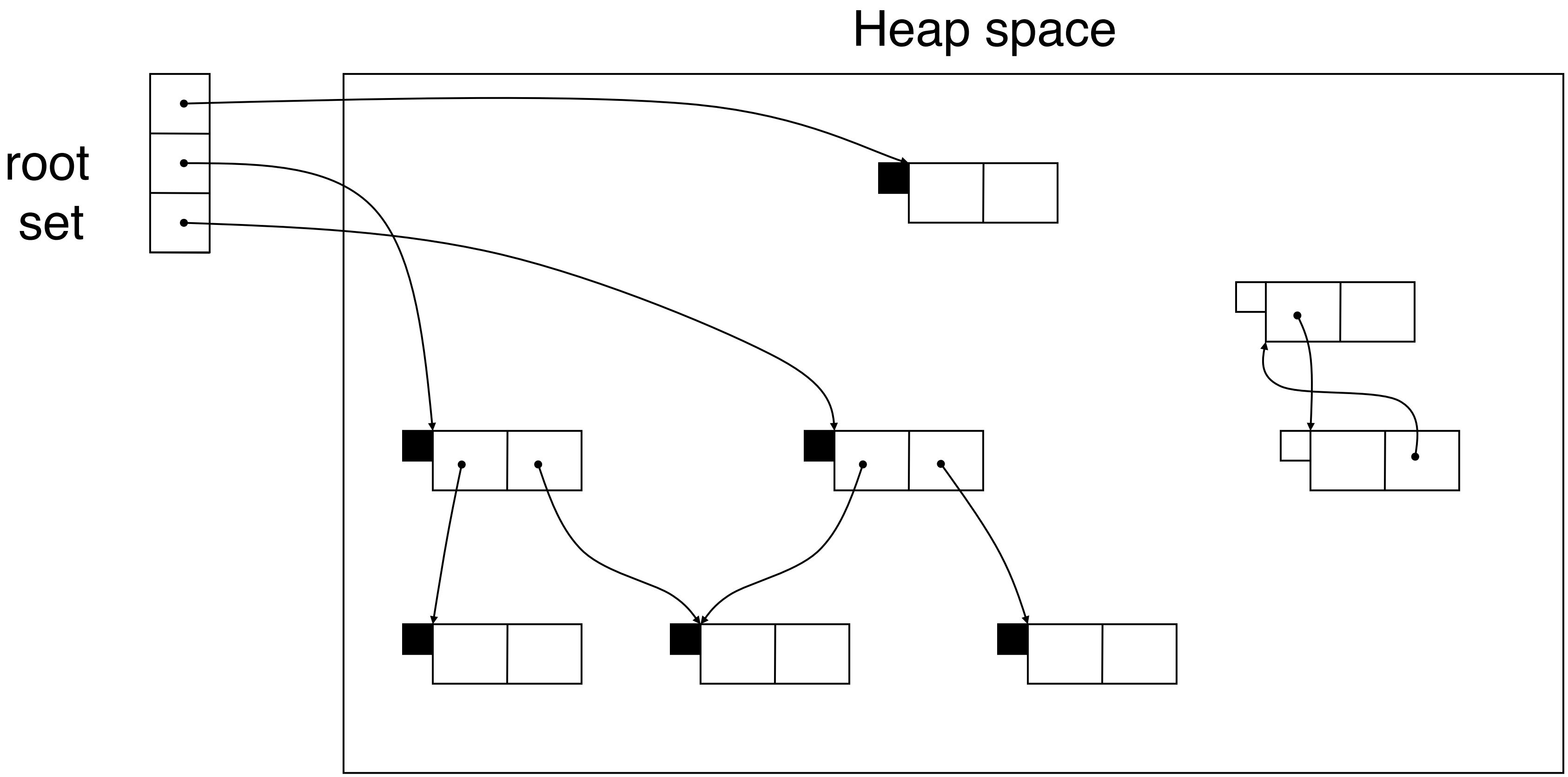
Mark-Sweep Example (1)



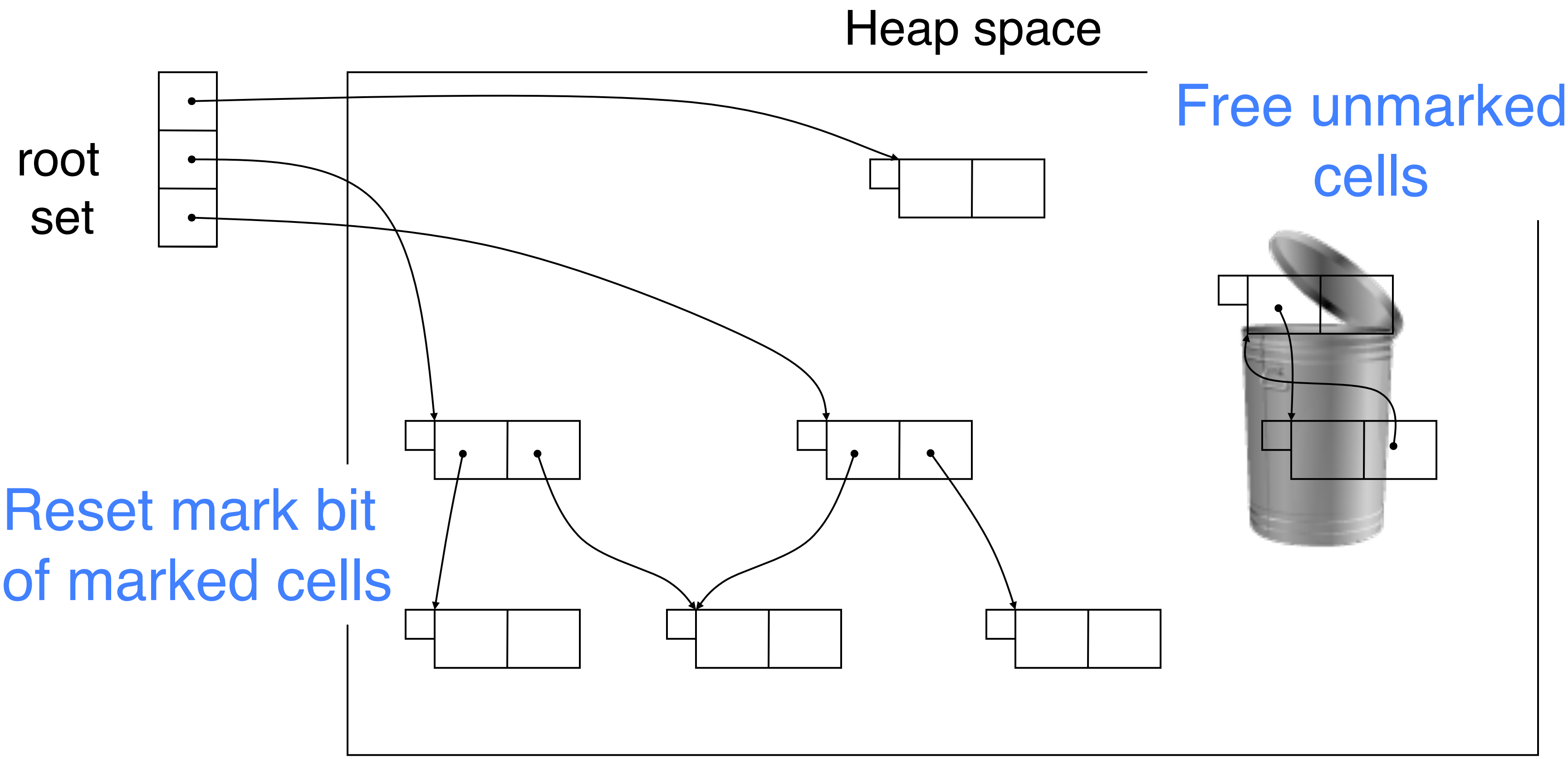
Mark-Sweep Example (2)



Mark-Sweep Example (3)



Mark-Sweep Example (4)



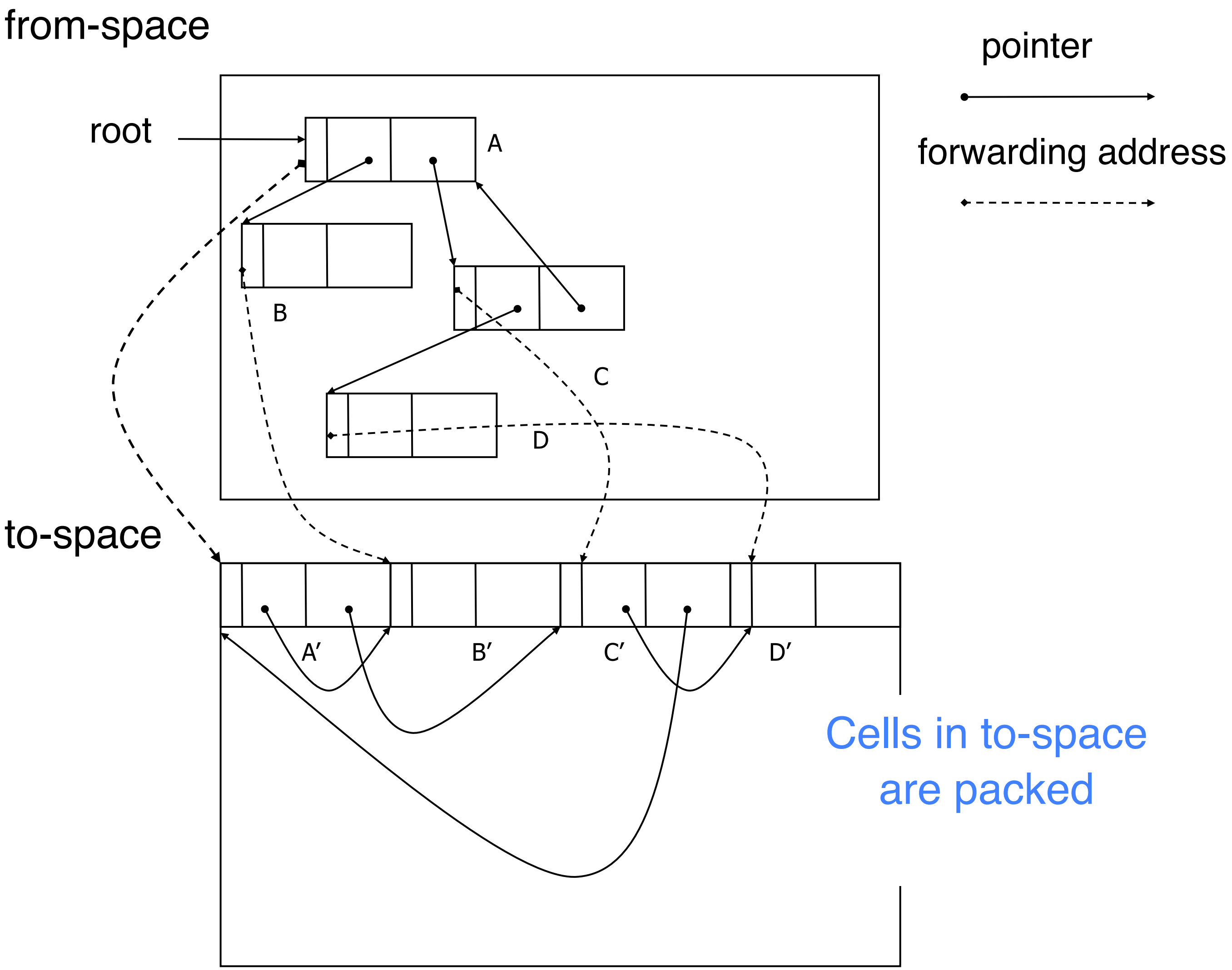
Mark-Sweep Costs and Benefits

- **Good**: handles cycles correctly
- **Good**: no space overhead
  - 1 bit used for marking cells may limit max values that can be stored in a cell (e.g., for integer cells)
- **Bad**: normal execution must be suspended
- **Bad**: may touch all virtual memory pages
  - May lead to excessive paging if the working-set size is small and the heap is not all in physical memory
- **Bad**: heap may fragment
  - Cache misses, page thrashing; more complex allocation

Copying Collector

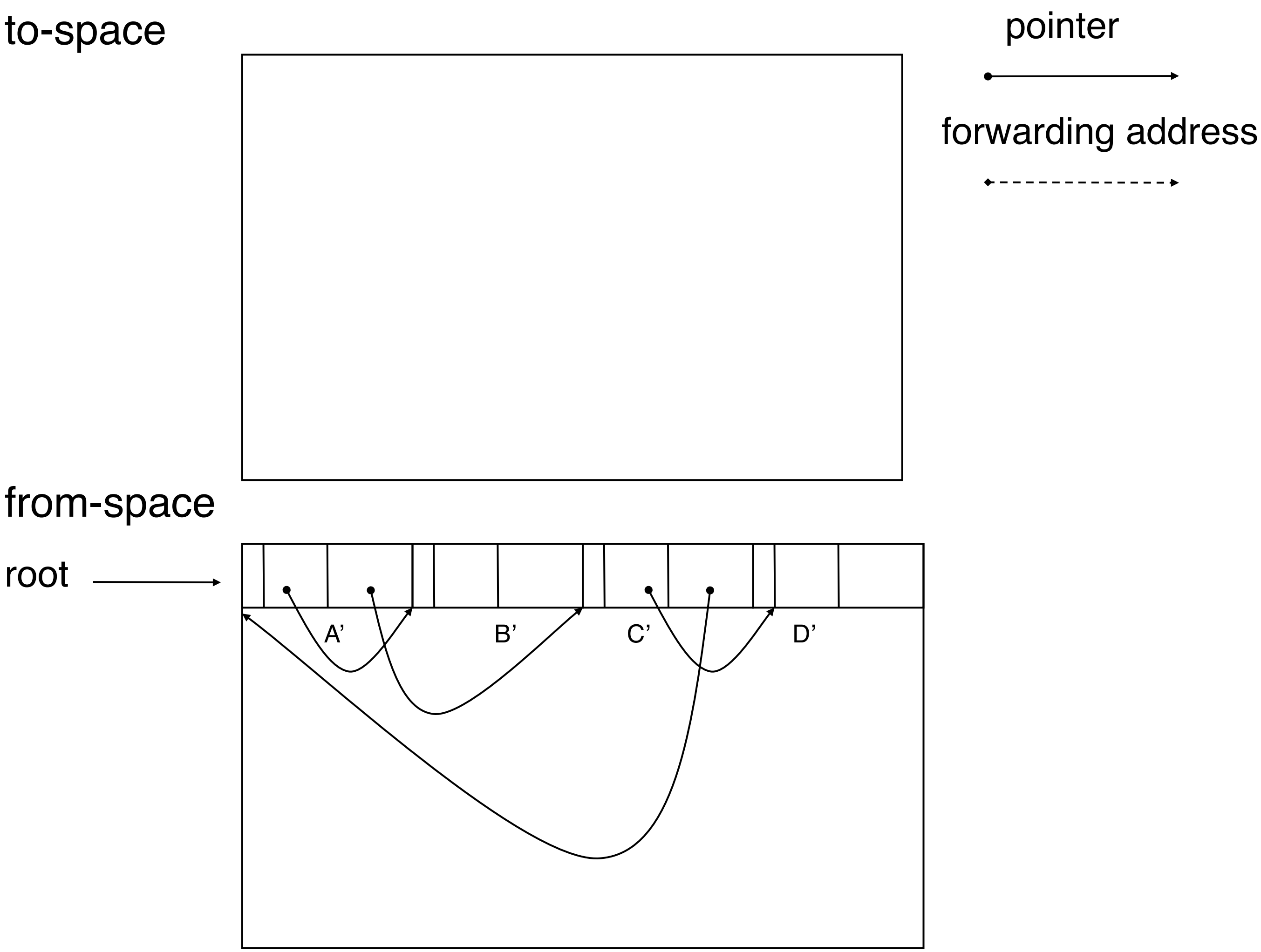
- Divide the heap into “from-space” and “to-space”
- Cells in from-space are traced and live cells are copied (“scavenged”) into to-space
  - To keep data structures linked, must update pointers for roots and cells that point into from-space
    - This is why references in Java and other languages are not pointers, but indirect abstractions for pointers
  - Only garbage is left in from-space
- When to-space fills up, the roles flip
  - Old to-space becomes from-space, and vice versa

Copying a Linked List [Cheney's algorithm]



Cells in to-space are packed

Flipping Spaces



Copying Collector Tradeoffs

- Good: very low cell allocation overhead
  - Out-of-space check requires just an addr comparison
  - Can efficiently allocate variable-sized cells
- Good: compacting
  - Eliminates fragmentation, good locality of reference
- Bad: twice the memory footprint
  - Probably Ok for 64-bit architectures (except for paging)
    - When copying, pages of both spaces need to be swapped in. For programs with large memory footprints, this could lead to lots of page faults for very little garbage collected
    - Large physical memory helps

Generational Garbage Collection

- Observation: most cells that die, die young
  - Nested scopes are entered and exited more frequently, so temporary objects in a nested scope are born and die close together in time
  - Inner expressions in Scheme are younger than outer expressions, so they become garbage sooner
- Divide the heap into generations, and GC the younger cells more frequently
  - Don't have to trace all cells during a GC cycle
  - Periodically reap the "older generations"
  - Amortize the cost across generations

Generational Observations

- Can measure “youth” by time or by growth rate
- Common Lisp: 50-90% of objects die before they are 10KB old
- Glasgow Haskell: 75-95% die within 10KB
  - No more than 5% survive beyond 1MB
- Standard ML of NJ reclaims over 98% of objects of any given generation during a collection
- C: one study showed that over 1/2 of the heap was garbage within 10KB and less than 10% lived for longer than 32KB

Frågor**?**

Genomgång projektarbete

Kommentarer efter TF:s föreläsning

- Gruppsammansättning
- Dagliga möten
- Låt inte saker ruttna
- Kontinuerlig integration
- Att alltid ha ett levande system
- T.ex. att själva använda ert eget system för att hitta buggar

Uppgiften

- Att skriva en egen minneshanterare i C

  Manuell minneshantering med explicit allokering och avallokering

  Referensräkning med explicita modifikationer av referensräknaren

  Tracing GC i med en enkel mark-sweep-algoritm

16-sidig specifikation i repot

  specifikation

  ”kurskrav”

Manuell minneshantering

- alloc
- free
- justering av allocs beteende

    snabb allokering som fragmenterar

    mindre fragmentering

    hög grad av lokalitet

Referensräkning

- alloc (refcount = 1)
- retain
- release
- count

Mark-sweep

- alloc
- automatisk gc vid slut på minne
- Hitta rotpekare på stacken (kod finnes)
- Scanna minnet efter möjliga pekare

Allokering med ”metadata”

- alloc(`"***d"`) = alloc(`3 * sizeof(void*) + sizeof(double)`)

  * = pekare

  c = char

  d = double

  f = float

  i = integer

  l = long integer

- Metadatat kan användas för att slippa scanna heapen

Hur arbetet skall utföras

- Ni är indelade i team om 6 personer

  Det kommer att vara viss flux denna vecka (alla är inte aktiva, etc.)

  Teamen beskrivs på kurswebben under ”grupper”

- Varje team delas in 3 programmeringspar

  Dessa skall rotera varje vecka så att man får jobba med (nästan) alla

  I vissa grupper kommer det att blir ett ”par” med tre programmerare

- Utgå från Scrum för skapandet av er process

  http://sv.wikipedia.org/wiki/Scrum

- **DEADLINE: 17/12**

Projektet

- All kod skall skrivas i C

    Testkod etc. får gärna skrivas i annat språk

- Testdriven utveckling skall användas (se bilder från Justin)
- Obligatoriska verktyg

    Trello (för planering och uppföljning av arbete)

    GitHub (för versionshantering och issue tracking)

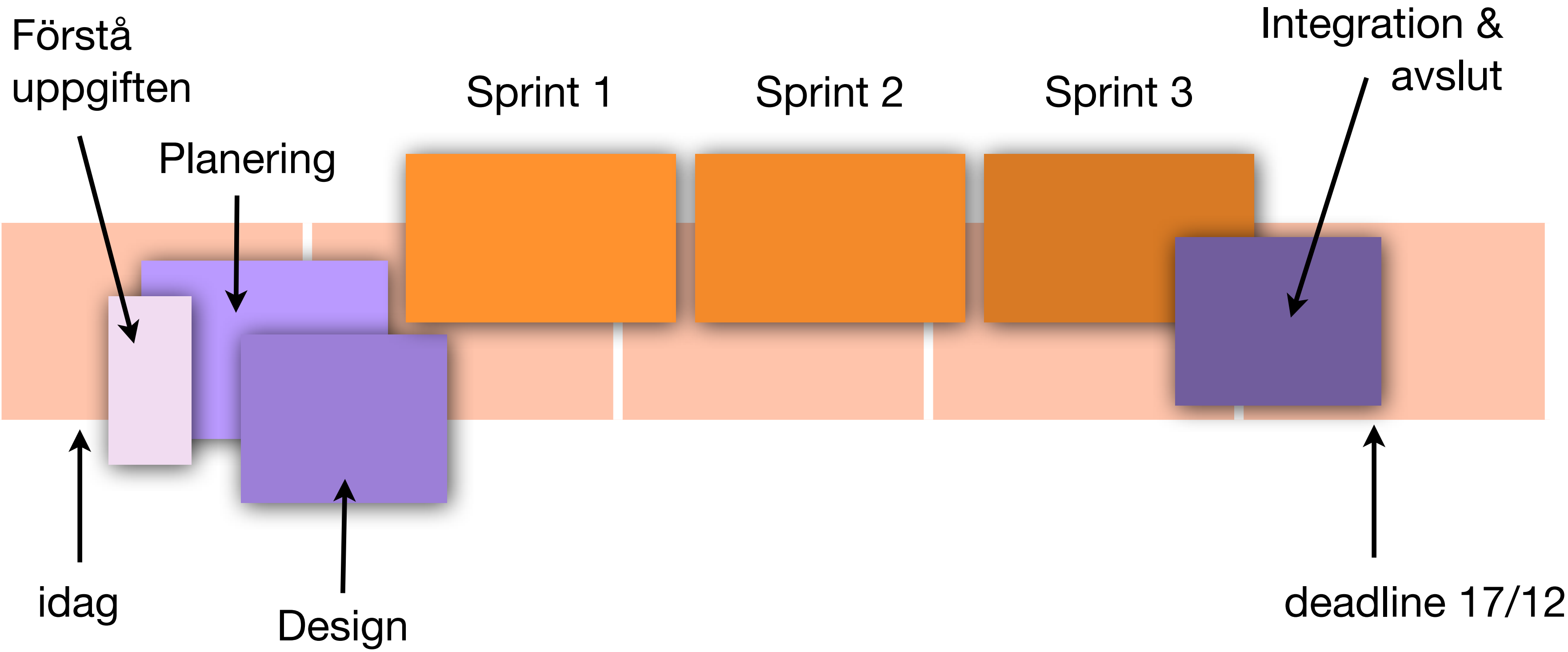
    git

    gcc

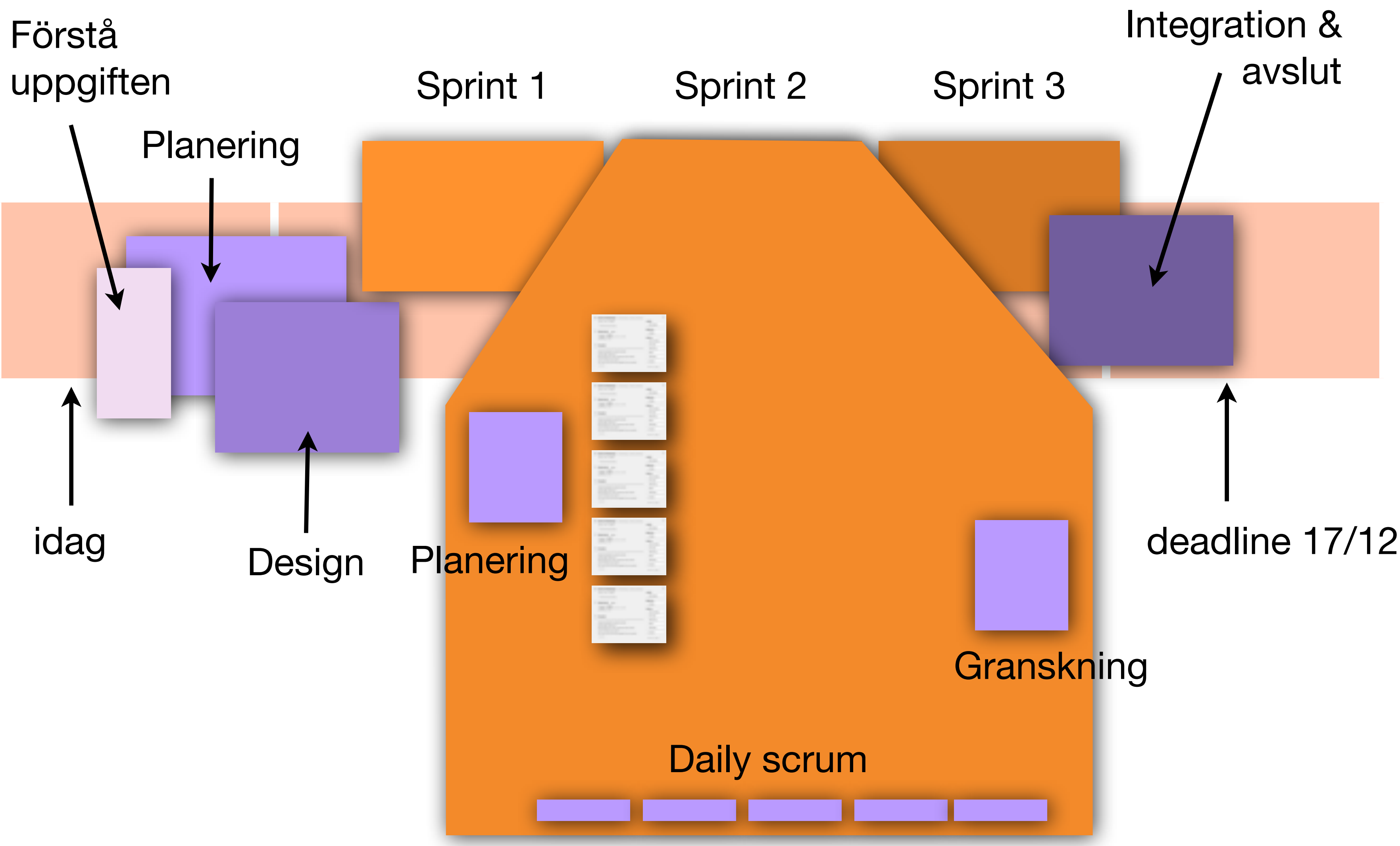
    Make

    valgrind

Tidplan (exempel)



Tidplan (exempel)



Vad skall lämnas in?

- Projektdagbok

  Vem har jobbat på vad, arbetstider per deluppgift, kort reflektion kring projektet, motivation av designen

- Övergripande designdokument

  Vilken uppdelning i delsystem, hur sitter delsystemen ihop?

- Koddokumentation på gränssnittsnivå

  Á la Javas API

- Gränssnitten mellan modulerna

  Det centrala dokumentet för parallell utveckling!

- Själva koden
- Tester

Plattformsoberoende

- Programmet som ni utvecklar skall vara plattformsoberoende

    Specifikt skall det fungera både på X86 och SPARC på IT

- Vissa aspekter av minneshanteringen är plattformsspecifika!

    T.ex. åt vilket håll stacken växer

    Pointer alignment?

    Spelar big/little endian någon roll?

Bedömning

- Den slutinlämnade kodens kvalitet och kompletthet
- Inlämnad dokumentation
- Kvalitet på egna testfall
- Aktivt deltagande i utvecklingsprocessen
- Projektdagboken
- **Beräknad arbetsåtgång: ca 133 arbetstimmar per person**

Vad händer om man inte är klar?

- Syftet med uppgiften är inte att producera ett färdigt system
- …*därmed inte sagt att det inte är viktigt att producera ett färdigt system*
- Om man inte är klar 17/12 kan tre saker hända:

  Man blir ändå **godkänd** eftersom man har dokumenterat sina brister, har skrivit bra kod i övrigt, och har en trovärdig plan för resten av systemet

  Man får **rest** och ett nytt sista inlämningdatum allra senast 10/1 –14

  Man har inte gjort ett seriöst försök och får därför **underkänt** och är välkommen igen HT2014

Får man redovisa mål som vanligt under projektet?

- JA! Det är en av poängerna!
- Vi byter nu till 1 labb/vecka (onsdagar 13–17)
- Syftet med dessa labbar är att man skall kunna redovisa mål som vanligt, samt få hjälp med projektet
- Notera att man får redovisa mål X65 även på labbarna

Konton

- trello.com — skapa ett eget konto
- github.com — skapa ett eget konto

- För att få tillgång till gruppens privata repo, maila Elias (elias.castegren@it.uu.se)

  Maila endast en gång per grupp!

  Ni skall använda de privata repo som vi ger ut — historiken där är viktig!