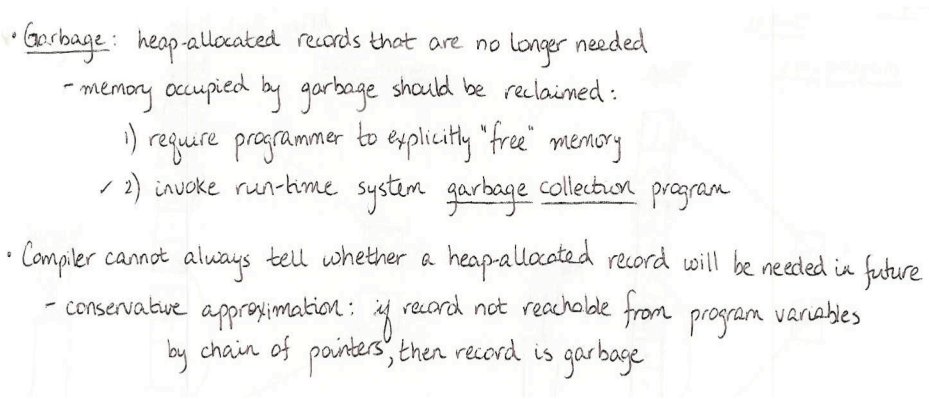
Garbage Collection

Deep Ghosh



- Garbage: heap-allocated records that are no longer needed
  - memory occupied by garbage should be reclaimed:
    1) require programmer to explicitly "free" memory
    ✓ 2) invoke run-time system garbage collection program
- Compiler cannot always tell whether a heap-allocated record will be needed in future
  - conservative approximation: if record not reachable from program variables by chain of pointers, then record is garbage

Manual Garbage Collection	**Automatic Garbage Collection**
+ High efficiency	+ Reduces programmer burden
+ Close programmer control	+ Eliminates sources of errors
− More code to maintain	− May hurt performance
− Correctness difficult	− Cannot determine all objects that won't be used in future

Overview

- Reference counting garbage collection
- Mark-and-sweep garbage collection
- Copying garbage collection

Reference Count Collection

- Mark and sweep collection identifies garbage by performing DFS
- Can identify garbage directly by keeping track of how many pointers point to each record
  → reference count of record, stored in record

Given: x ← p, x is program variable or record field

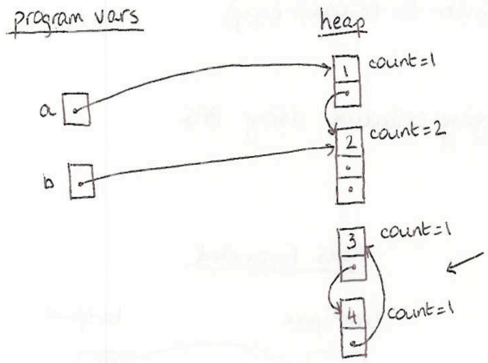
- Compiler emits code to perform following:
  1) increment reference count of p
  2) decrement reference count of record r that x previously pointed to
  3) if reference count of r=0, r is put on free-list, all records pointed to by r have their reference counts decremented

Two problems:

1) incrementing and decrementing reference counts expensive
   - instead of generating 'x←p', compiler must now generate

```
z ← x
c ← z.count
c ← c-1
z.count ← c
 if c=0 then putOnFreeList(z)
x ← p
c ← p.count
c ← c+1
p.count ← c
```

2) impossible to reclaim cycles of garbage



← these two records are garbage, but each has reference count of 1

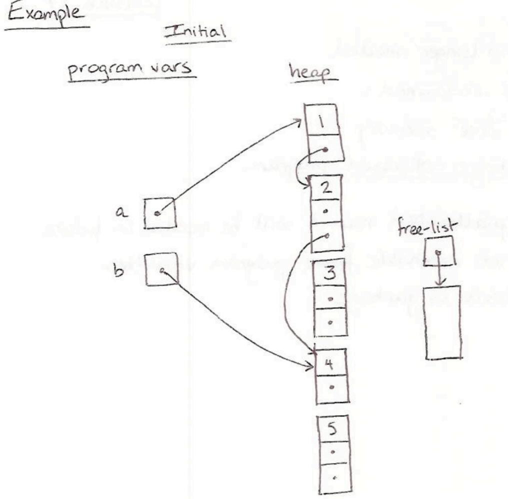
- two solutions to this problem:
  - require programmer to explicitly break all cycles when done
  - perform occasional mark-sweep collection to reclaim cycles of garbage

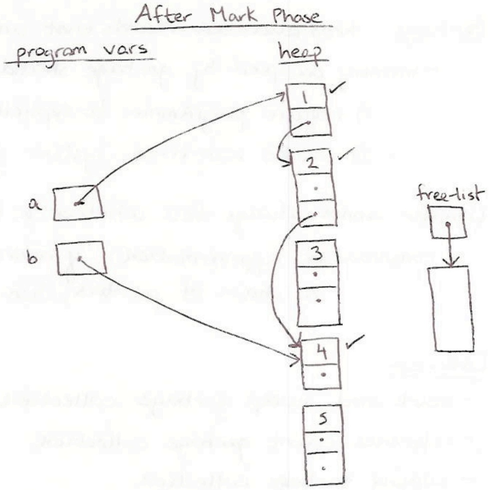
Mark and Sweep Collection

- program variables + heap-allocated records form directed graph
  - roots: program variables
  - internal nodes and leaves: heap-allocated records
  - edges: pointers
- mark phase: perform depth-first search (DFS) from each root node, marking all nodes reachable from root
  - heap-allocated record n reachable from root r if path $r \to n_1 \to n_2 \to \dots \to n$ exists
  - any node not marked must be garbage
- sweep phase: scan through entire heap, collecting onto a free-list all unmarked nodes
  - all nodes subsequently unmarked for next collection

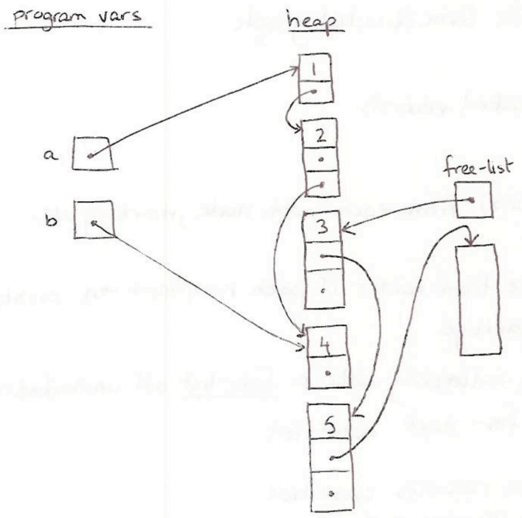
- After garbage collection, compiled program resumes execution
  - when run-time support function allocRecord called to allocate size n record, free-list checked for record of size n
    - if record exists, then return it
    - else replenish free-list by performing garbage collection

Example





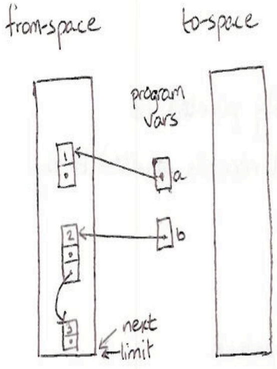
After Sweep Phase



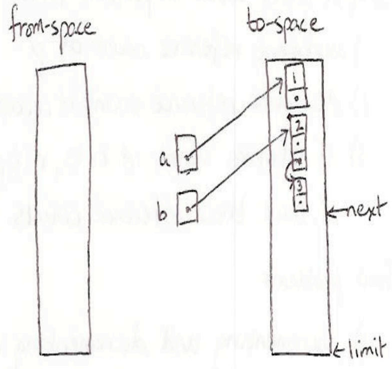
Copying Collection

- Heap divided into two regions: <u>from-space</u> and <u>to-space</u>
  - all records allocated in from-space
    - from-space is fragmented - garbage interspersed with reachable data
  - when from-space full, copying garbage collection copies all reachable records into to-space
    - to-space copy occupies contiguous memory → compact
  - roots point to to-space copy, entire from-space made unreachable
  - to-space becomes from-space, from-space becomes to-space

Before Collection



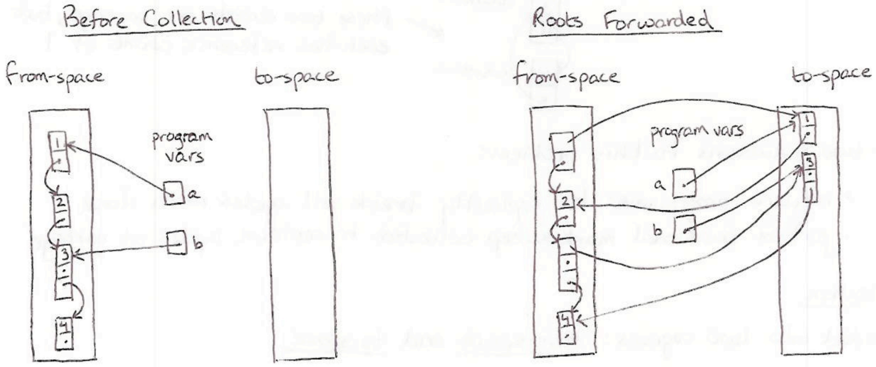
After Collection



- Forwarding: main operation used during copying collection
  - given a pointer that points to from-space, make it point to to-space
    1) if pointer p points to from-space record that hasn't been copied, then:
       - copy record into to-space
       - make first field of record pointed to by p (p.f1) point to copy
         ⇒ forwarding pointer
       - return pointer to to-space copy
    2) if pointer p points to from-space record that has been copied, then:
       - return p.f1 (pointer to to-space copy)

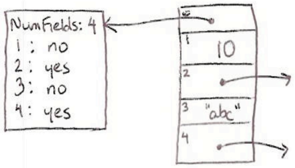
- Cheney's Algorithm: performs copying collection using BFS

Example



- How does compiler interact with garbage collector?
  1) generates code to invoke run-time support function allocRecord when record must be heap-allocated
     - allocRecord will invoke garbage collector, if necessary
  2) describes locations of directed graph roots, in preparation for next garbage collection
  3) describes layout of records on heap

- In order to determine which heap-allocated records are reachable, collector must know size of each record + location of pointer fields
  - let first word of every record be pointer to type-descriptor record



  - type-descriptor record generated by compiler during semantic analysis
    → pointer to record passed as argument to allocRecord

- Compiler must also identify all "active" program variables that are pointers
  - may be located on stack or register
  - used as roots during mark-sweep or copying garbage collection