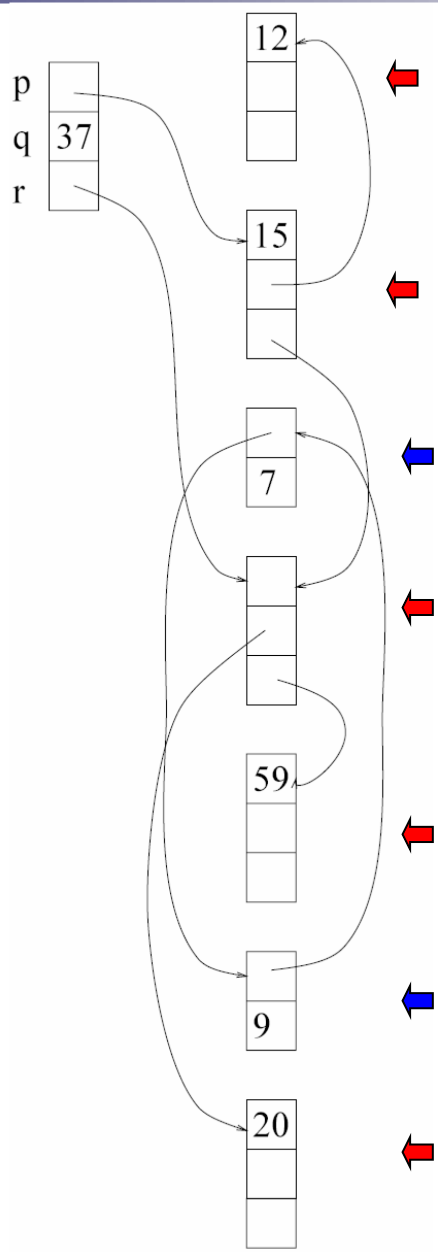
Garbage Collection

Garbage

- Heap-allocated objects not reachable by any chain of pointers from program variables
  - `Car c; c= new car(); …. c= new truck();`
- Garbage collection (GC) is reclaiming the memory space occupied by garbage
  - GC is performed not by the compiler but by the run-time system (the support program linked with the compiled code or by virtual machine)
  - GC recently enjoys its renewed popularity due to Java
    - Reduced time-to-market, improved S/W reliability…

Garbage Collection

- How can we identify garbage?
  - When we need GC (i.e., run out of memory), we can view **program variables** and **heap-allocated, live objects** form a **DAG** where variables are **roots**
    - Why? a live object *n* is reachable in the DAG since there will be a path starting from some root *r* to *n*
    - An object is garbage if it is not reachable in the DAG



Reachable objects

Garbage

Root Sets

- How to identify roots when GC occurs?
  - In Java, local variables (including parameters) and temporaries are located in stack
  - If JIT compiler is employed, some of them can reside in registers as well
  - So, we need an information on locations of variables in order to identify roots precisely
    - Who knows this information? Compiler does
- Two approaches: precise GC and conservative GC

Precise GC

- Compiler generates a map of (variables, locations) for all places in the code where GC can possibly occur
  - At object allocation point: new()
  - Some blocking operations in some VM
    - Function calls, synchronizations, loop back edges…
  - When GC occurs, GC gets roots using the map

Conservative GC

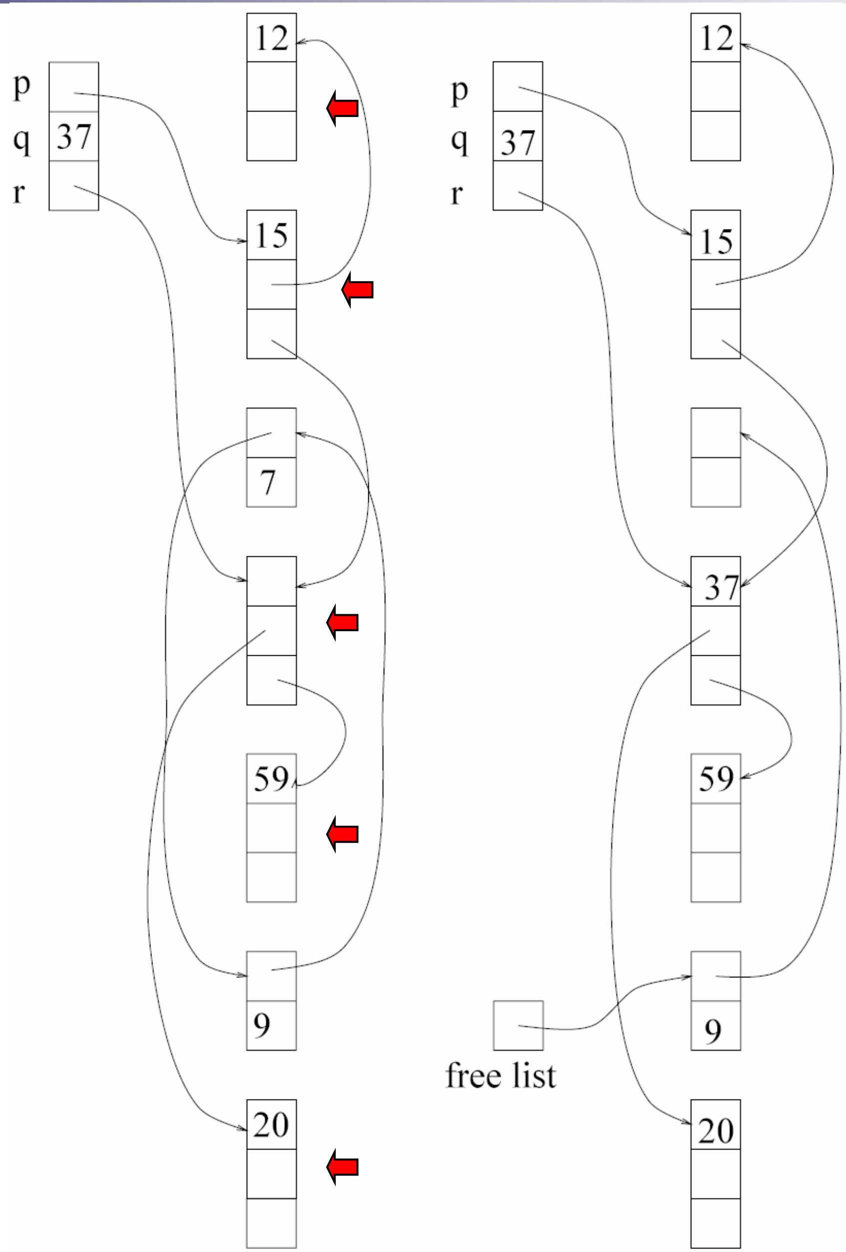
- Presume all locations that can potentially have pointers as roots
  - If they are really pointers, then everything is fine
  - If not, then we might regard a dead object as a live one, so GC might not be able to collect it
- Simple but not-effective GC
- Not used in a real VM

GC Techniques

- Mark-and-Sweep
- Copying
- Generational

Mark-and-Sweep Collection

- Simplest garbage collection algorithm
- Mark phase
  - Mark all reachable nodes by graph traversal
    - E.g, depth-first traversal
- Sweep phase
  - Scan through the entire heap, looking for nodes that are not marked; these nodes will be reclaimed into a linked list (`freelist`)



(a) Marked (b) Swept

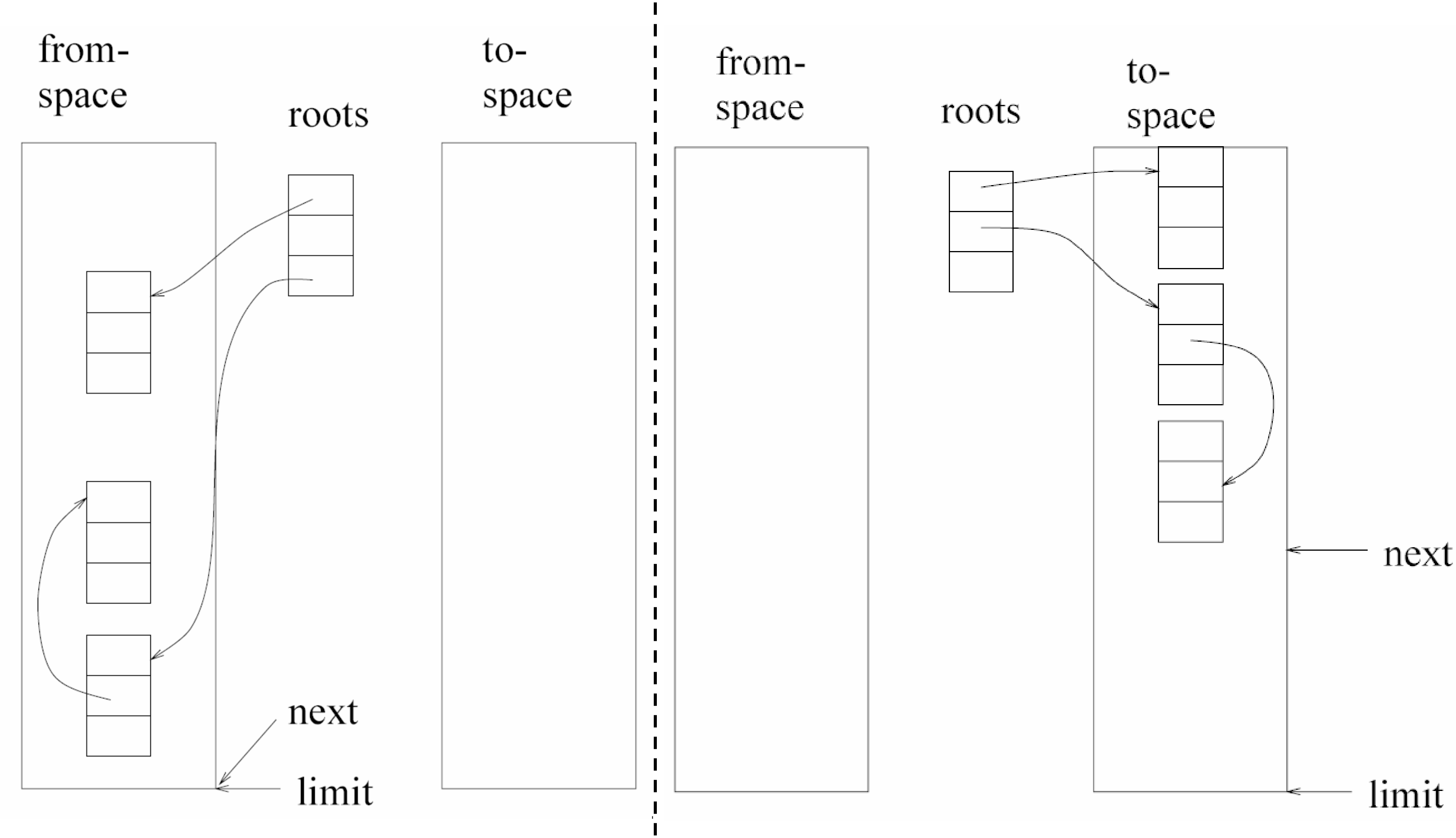
```
function DFS(x)
    if x is a pointer into the heap
        if record x is not marked
            mark(x)
            for each field f of record x
                DFS(x.f)

Sweep phase:
    p = first address in heap
    while ( p < last address in heap )
        if record p is marked
            unmark p
        else let f1 be the first field in p
            p.f1     = freelist
            freelist = p
        p = p + size_of_record(p)
```

An Array of freelist

- For efficient allocation, an array of freelists is used so that freelist[i] is a list of all records of size i
  - Can allocate a node of size i by taking freelist[i]
  - If attempt to allocate from an empty freelist[i], it can try to grab a larger record from freelist[j] (j > i) and split it, putting unused portion back on freelist[j-i] (if this fails, we need GC).

Copying Collection

- Heap is divided into **from-space** and **to-space**
  - Memory is allocated only from the from-space initially
  - The collector traverses the DAG in the from-space, building an **isomorphic copy** in the fresh to-space
  - The to-space copy is compact, occupying contiguous memory without fragmentation
    - Incrementing the next pointer contiguously
  - The roots are made to point at the to-space copy
  - Then, the entire from-space is collected
  - Change the role and continue



(a) Before collection

(b) After collection

Generational Collection

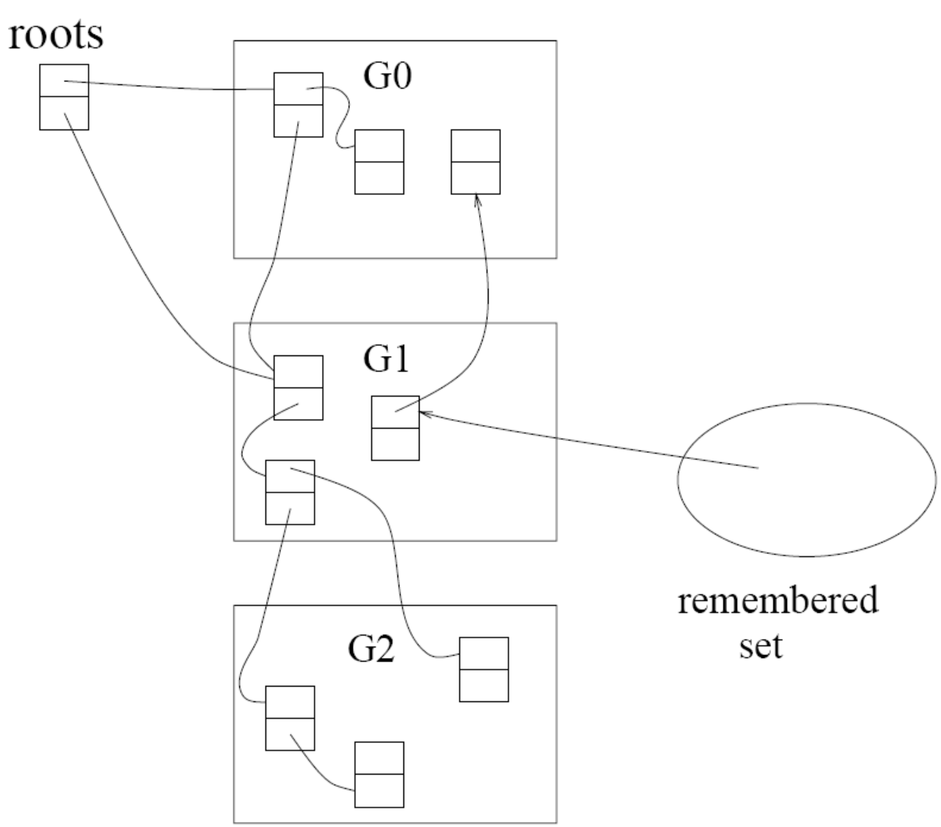
- In many programs, newly created objects are likely to die soon while objects that are still reachable after many collections will survive more collections
- GC should focus its efforts more on "young" data
- Generational GC: divide the heap into generations
  - With the youngest objects in generation G0; G1 objects are older than G0, G2 objects are older than G1, and so on
  - Collection of G0 just starts from its roots
    - Can use either mark-and-sweep or copying collection
  - After several collections of G0, G1 may have enough garbage, so both G0 and G1 are collected altogether.
  - If an object at G_i survives two or three collections, it promots to G_{i+1}

Problem of Generational GC

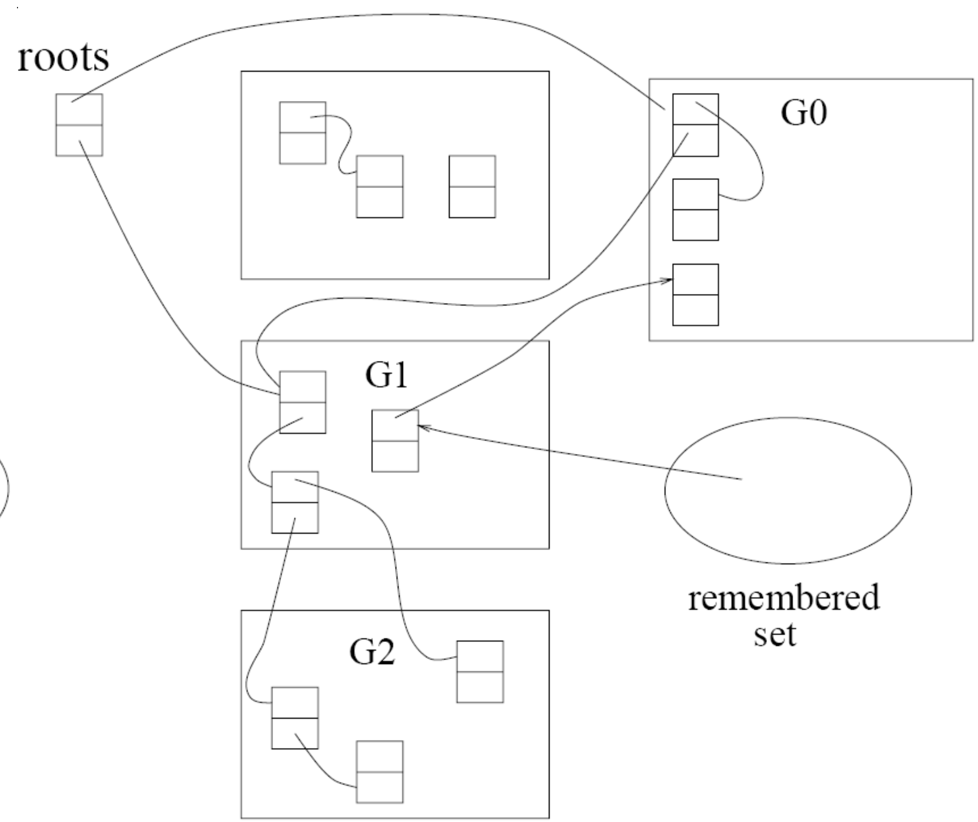
- Roots for G0 are not just program variables; it can includes any pointer within G1/G2 that points into G0
  - If too many of these “old” roots. Searching time for roots might be longer than traversal of G0
  - Fortunately, it is rare for an old record to point to a much younger object; an object is initialized when created by pointing other older objects
  - An old object **b** can point to a newer object if some field of **b** is updated long after **b** is created

Remembered List and Set

- In order to the search of all G1, G2, ... for roots of G0, we make the program **remember** where there are pointers from older objects to new objects
  - Remembered list: when there is an update `b.f = a`, (generate code to) put `b` into a vector of updated objects. At each GC, the collector scans the list looking for old objects that point to G0
  - Remembered set: use a bit within b to record that b is already in the vector; then the code can check this bit to avoid duplicate references to b in the vector



(a) Before collection



(b) After collection

GC Summary

- Most modern programming languages are equipped with GC for faster development
- Compiler for GC language generally interact with the collector by generating code that describes locations of root
- Generational copying collection is most popular, with some incremental collection