Garbage Collection

Problem: When items are allocated from the heap, how do we know when to free them?

- **Solution 1**: The programmer explicitly frees the memory.
  - Pros: Easy for compiler
  - Cons: Hard for programmer
  - Ex: C/C++

- **Solution 2**: Free any variables that aren't live.
  - Actually, use a heuristic of freeing variables that aren't reachable.
  - This is garbage collection.

Mark-and-Sweep

- Can represent heap allocated records as a directed graph
- Step 1: Mark records with a DFS
- Step 2: Sweep the heap looking for unmarked nodes
- Garbage is put into the *freelist*

Algorithm

Mark phase

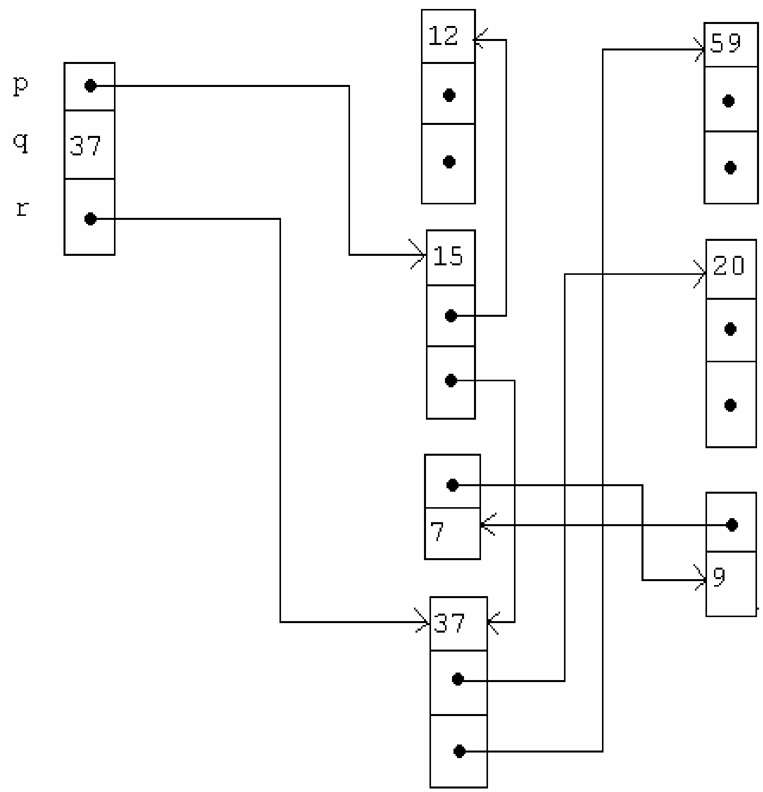
- For each root *x,* do DFS(*x*)

Sweep Phase

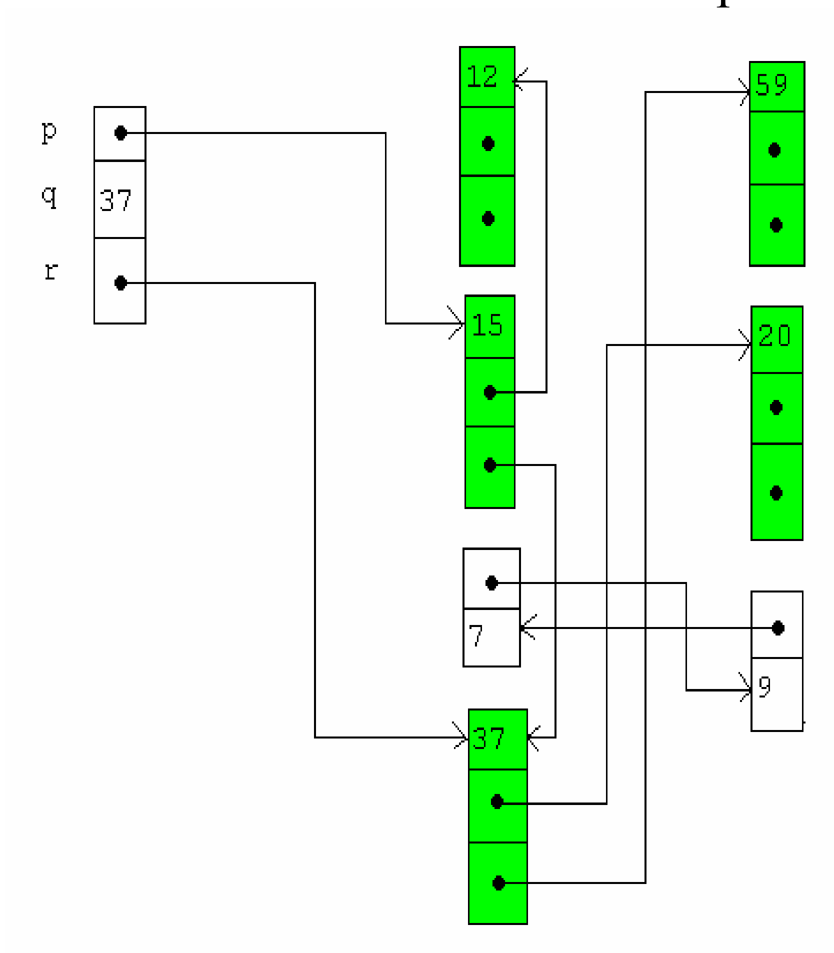
- *p* ← First address in heap
- While *p* < last address in heap
  1. If record *p* is marked, unmark *p*
  2. Else, let f_1 be the first field in *p*
     $p.f_1$ ← freelist
     freelist ← *p*
  3. *p* ← *p* + (size of record *p*)

Example

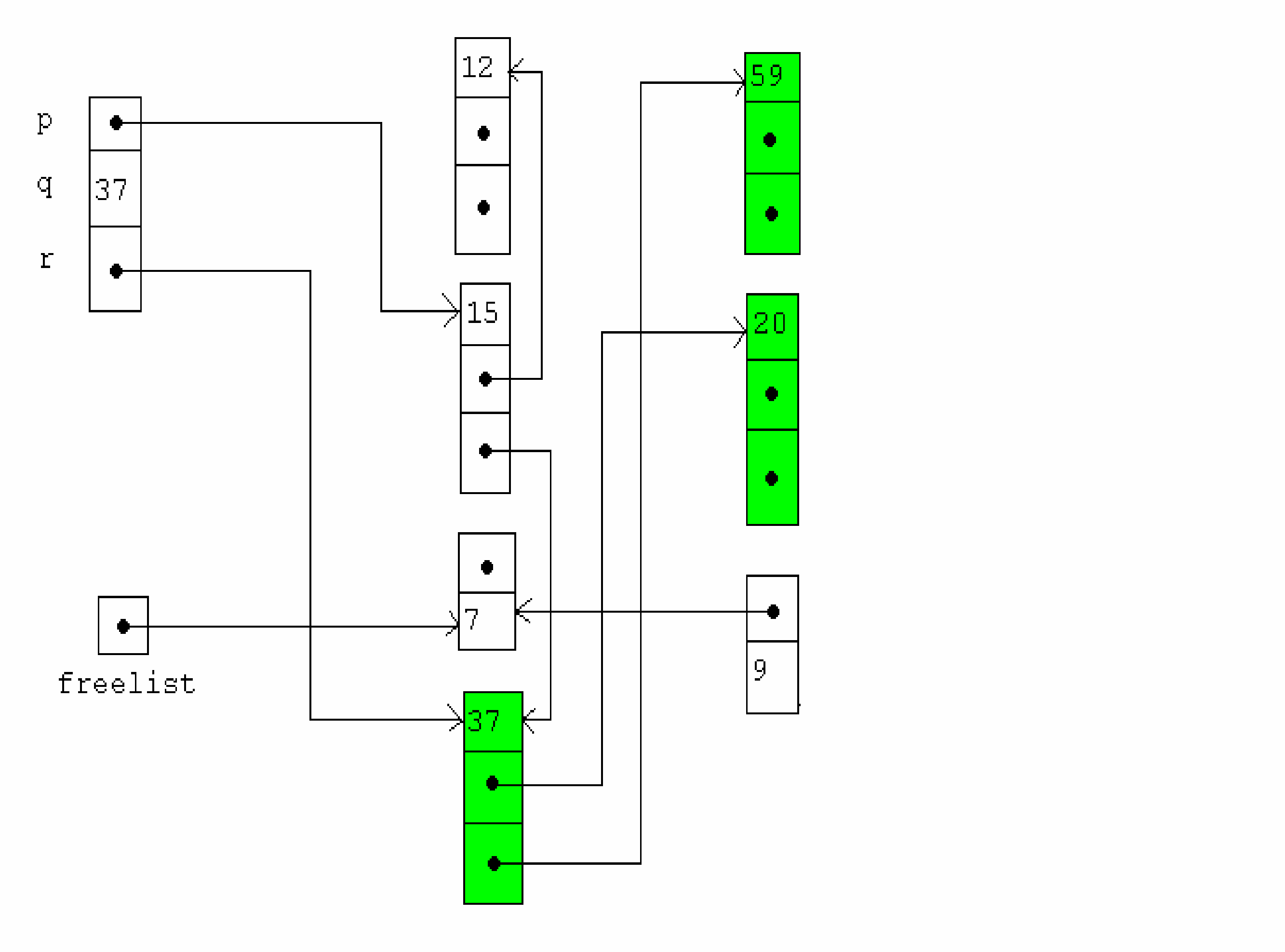
Beginning Heap



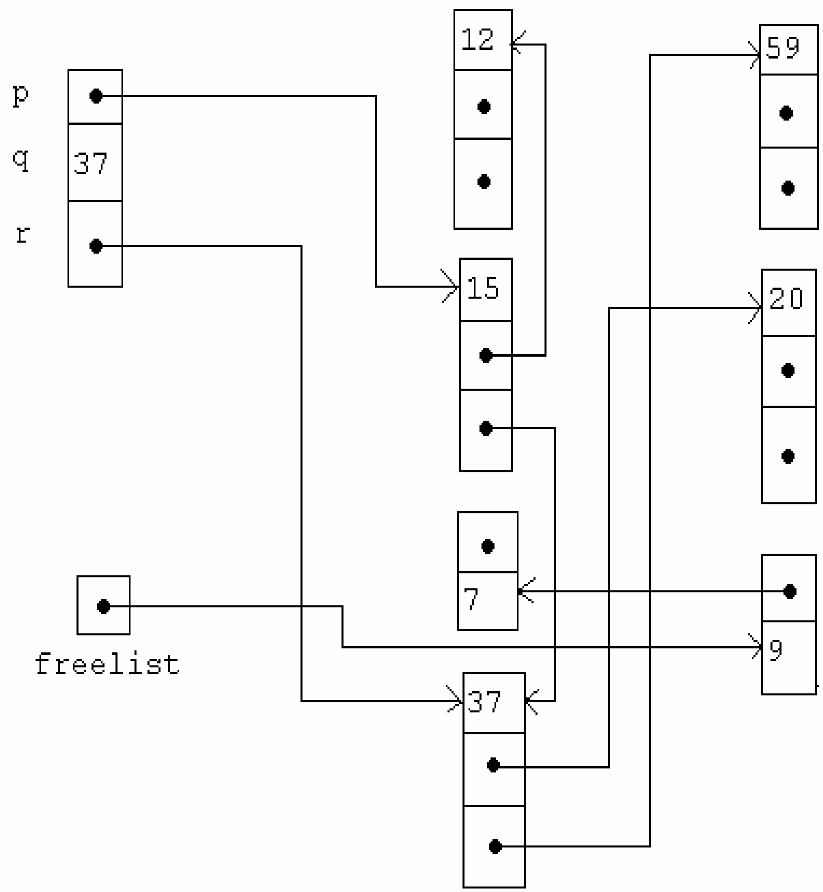
After mark phase



During Sweep phase



After Sweep phase



Complexity

- ***R*** = number of reachable records
- ***H*** = size of the heap
- Amortized cost: $(C_1\boldsymbol{R} + C_2\boldsymbol{H}) / (\boldsymbol{H} - \boldsymbol{R})$
- What does this mean?

Implementation Issues

- If we use recursion, the run-time stack could reach a size of ***H*** activation records!
- If we use an explicit stack, we could still have a stack of size ***H*** words!
- Pointer Reversal – Use the elements in the heap as the stack itself, reversing the pointers as you go
- Array of Freelists – freelist[*i*] stores records of size *i*
- Fragmentation – Internal and External

Copying Collection

- traverse graph

- need two heaps
  - from-space (working heap)
  - to-space (heap for garbage collection)

- redirect roots to to-space (new space)

- copy records from old space to new space
  - create isomorphic copy in to-space

- after all records moved, swap new and old space

- copy is contiguous – no external fragmentation

Advantage:

- simplicity - no stack or pointer reversal required

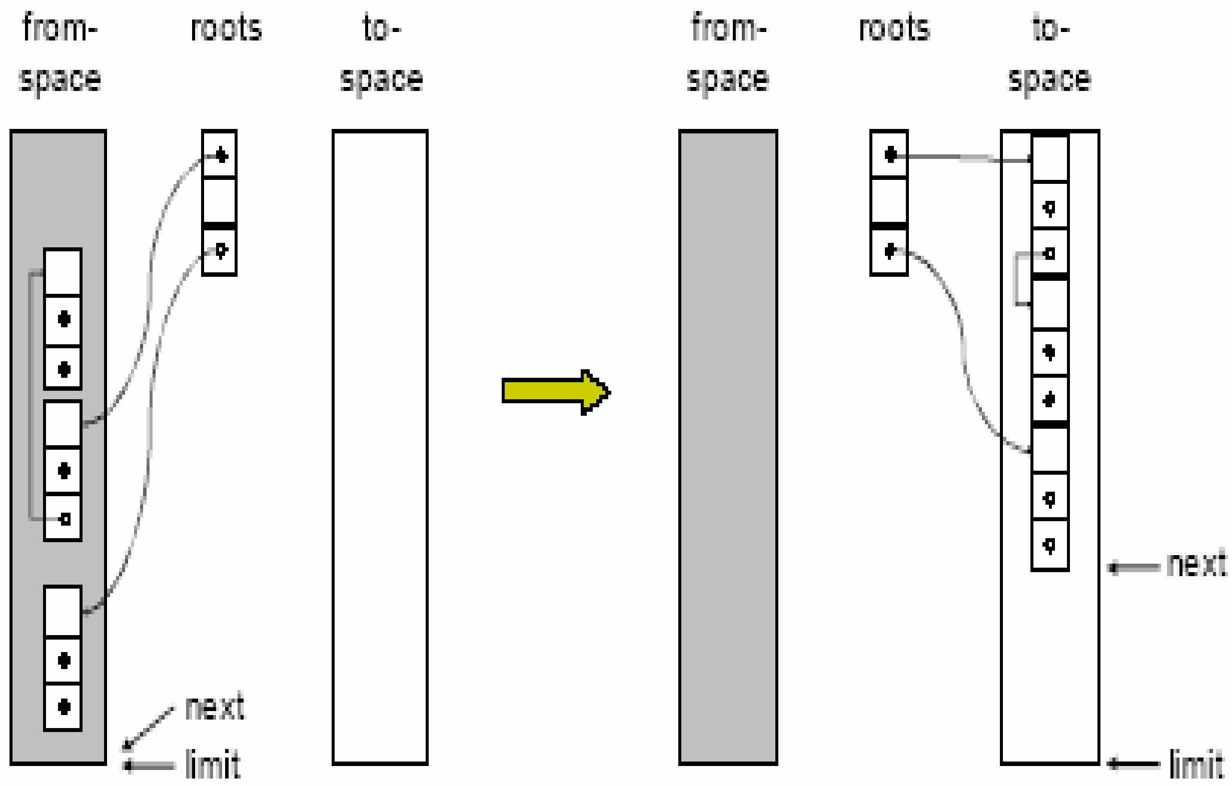
- doesn't move garbage

- makes free space contiguous,
  - allocation cheap
  - no freelist

Disadvantage:

- half of memory is wasted

- maintain accurate pointer
  - heap pointers (next, scan)
  - record pointer


from-
space
roots
to-
space
next
limit
from-
space
roots
to-
space
next
limit

Pointer Forwarding

Given pointer *p:*

Redirect record from from-space to to-space

Case 1:
If *p* points to already copied record, $p.f1$ is forwarding pointer that tells where copy is in to-space. Return forwarding pointer

Case 2:
If *p* points to record that has not been copied, copy record to the next free location in to-space and store forwarding pointer into $p.f1$. Return forwarding pointer

Case 3:
p points outside of from-space (to-space/not garbage collected arena)

```
forward (p) {
  if p points t from-space
    then if p.f1 points to to-space
            then return p.f1
    else for each field fi of p
              next.fi := p.fi
          p.f1 := next
          next := next + (size of *p)
          return p.f1
  else return p
```

Cheney's Algorithm

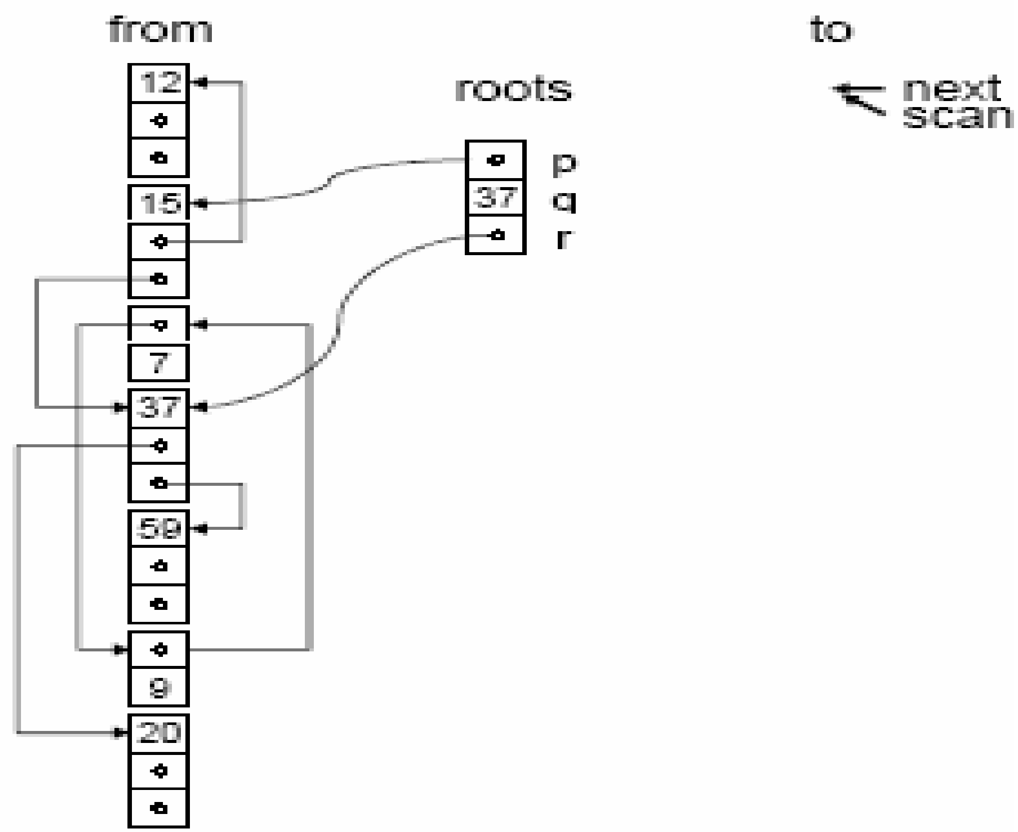
- Performs a breadth-first copy

1. Scan and Next points to start of to-space
   Roots are forwarded
   Records reachable from roots copied to to-space
   Next pointer incremented accordingly

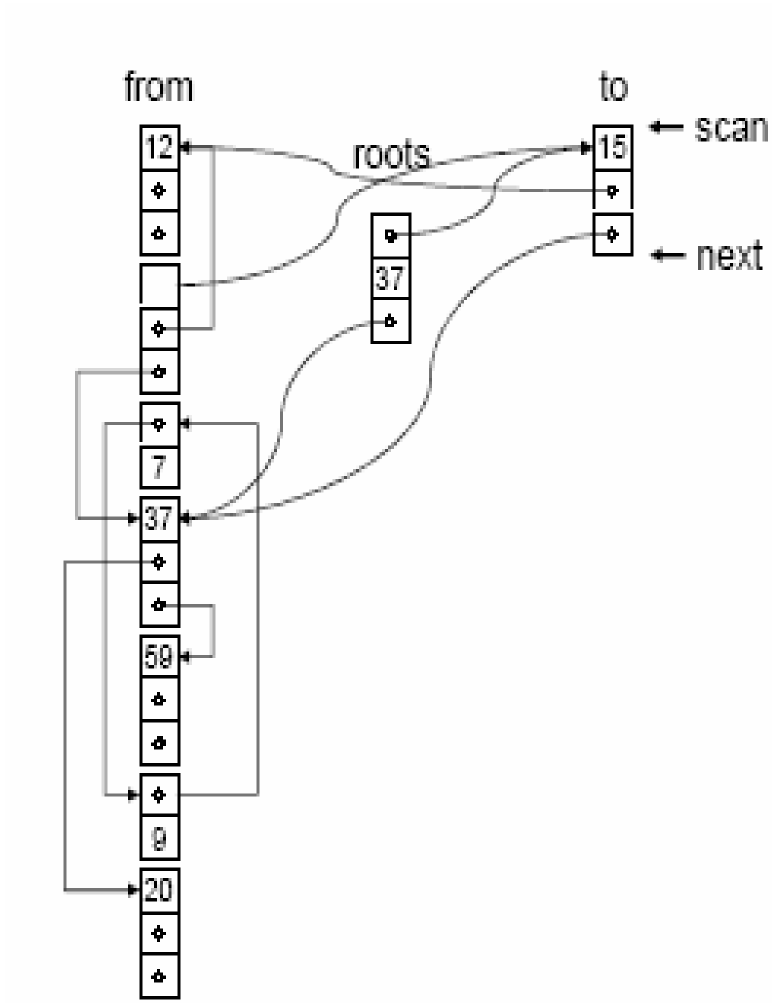
2. Scan ←→ Next contain records copied to to-space but fields not yet forwarded (ie fields point to from-space)
   - Scanning a record
     Forwards fields of each record not yet in to-space
     Both next and scan are incremented
     Garbage collection done when scan reaches next

```
scan := begin-of-to-space
next := scan
for each root r
    r := forward(r)
while scan < next
    for each field fi of *scan
        scan.fi := forward(scan.fi)
    scan := scan + (size of *scan)
```

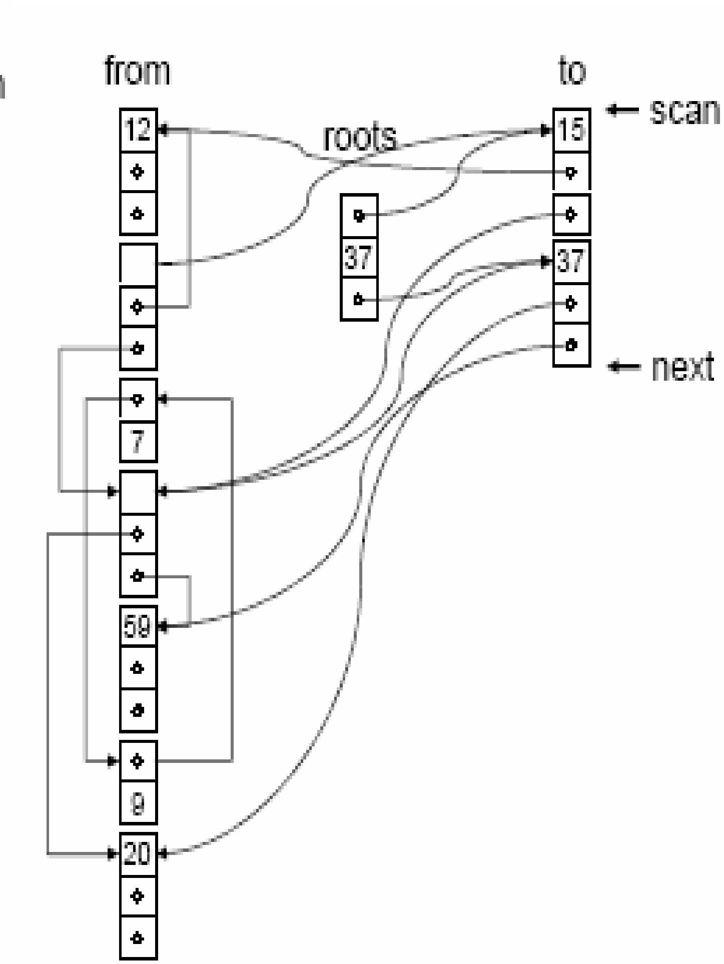
Example
Before



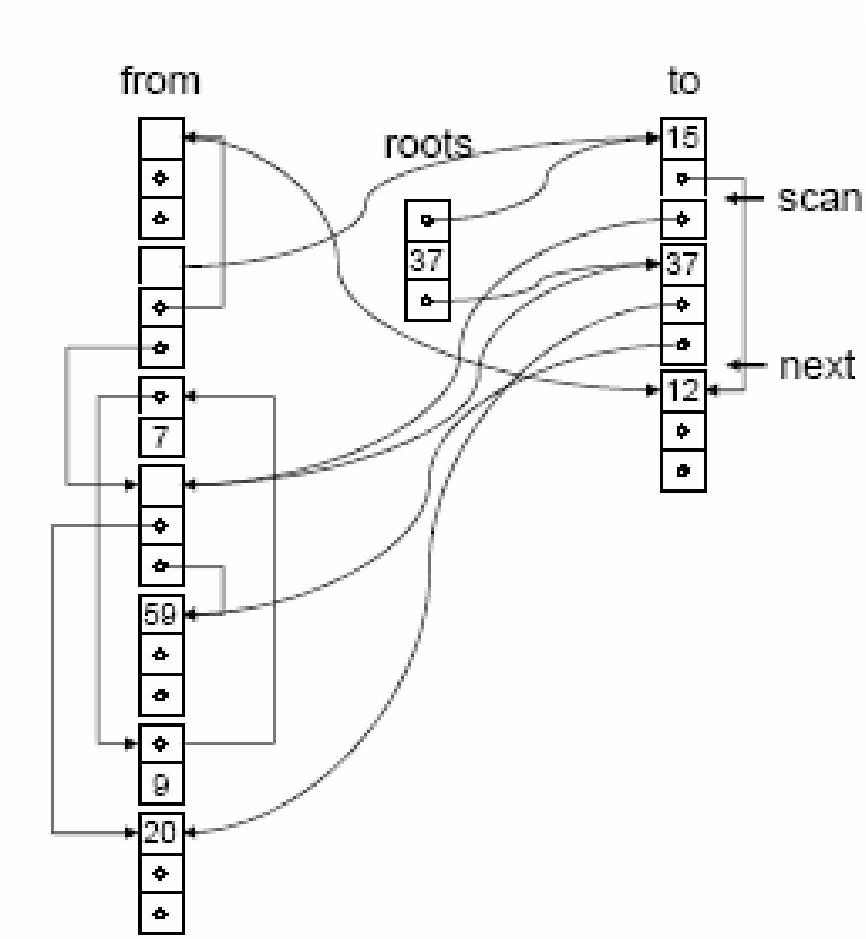
Forward Roots



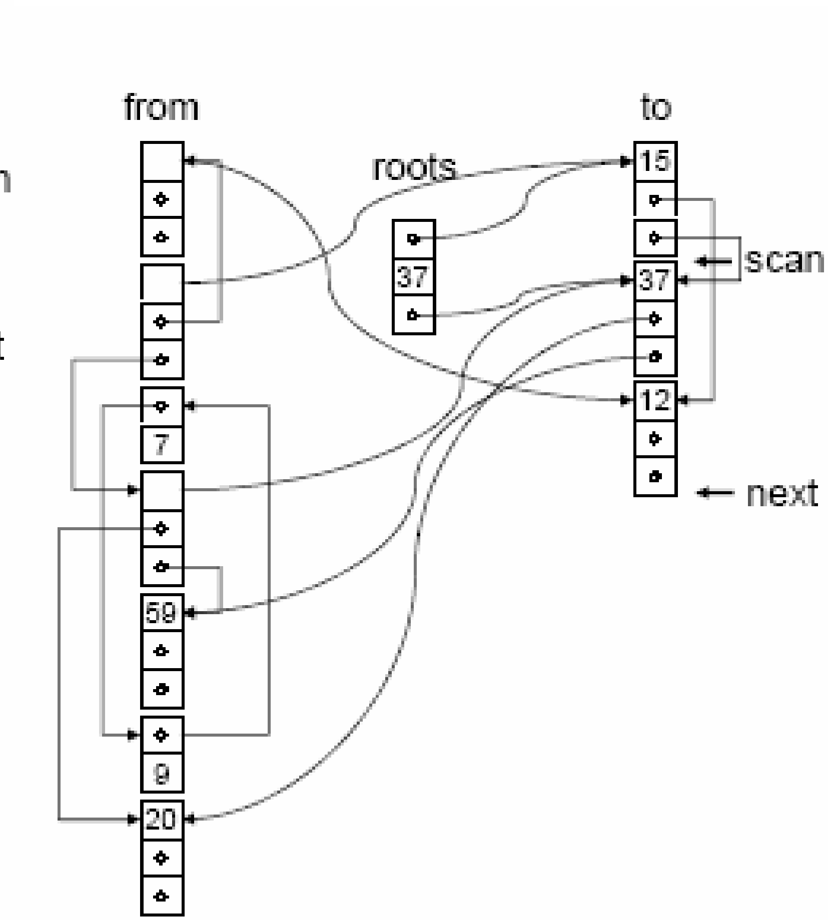
Forward Roots



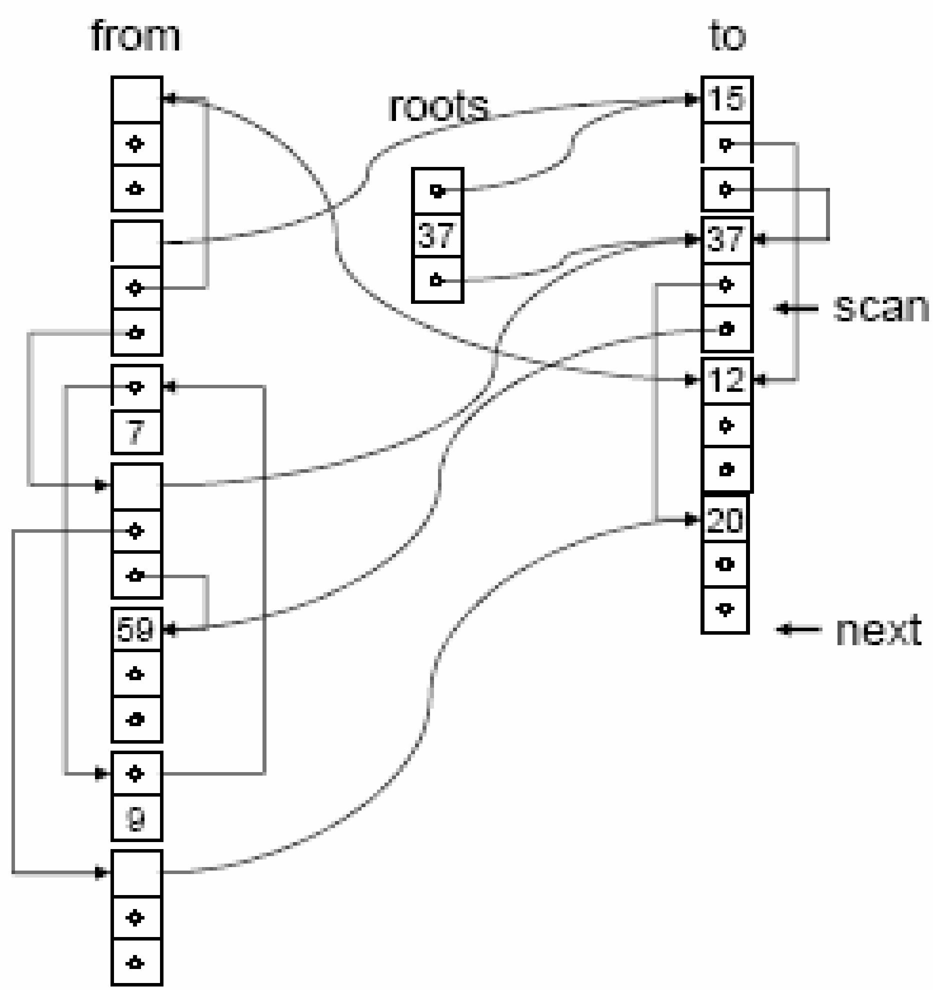
Scan and Forward

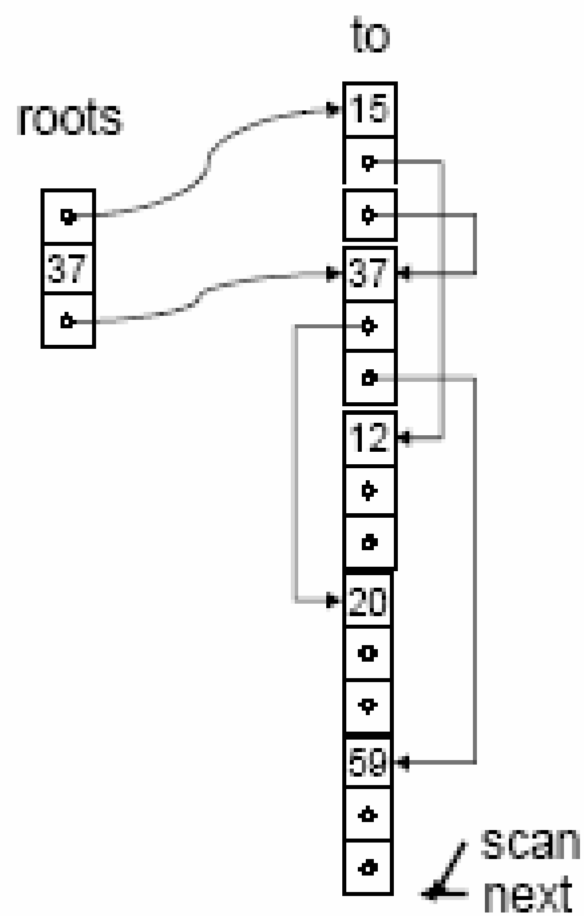


Scan and Forward



Scan and Forward **Done**





Bad locality of reference:

- breadth-first copy
    - records end far apart in memory
    - bad for virtual memory and caching

    Solution:

    Hybrid of breadth-first and depth-first

    Use breadth-first but forward the child of a node immediately, if possible

Cost

- breadth-first copying & hybrid

Amortized cost $c_3 \cdot R / (H/2 - R)$

C_3R = Total cost of collection based on number of records copied

$H/2 - R$ = heap divided by two – words/records to allocate before next collection

$H >> R$, cost approaches 0