Cheney's Copying GC

Iterative copying cheaper than recursive copying

Allocation in copying collector

```
init() {
    to_space  = Heap_bottom
    space_size = Heap_size / 2
    top_of_space = to_space + space_size
    from_space = top_of_space + 1
    free = to_space
}
new(n){
    if (free + n > top_of_space) {flip()}
     if (free + n > top_of_space) { abort "Memory exhausted"}
    new_cell = free                              // allocate()
    free = free + n
    return new_cell
}
```

Flipping the spaces

```
flip() {
    to_space, from_space  = from_space, to_space
    top_of_space = to_space + space_size
    free = to_space
    for R in Roots
            R = copy(R)                    // Root pointer now points to copy of R
}
```

Copying for variable-sized objects

```
// P points to memory location, not an address
copy(P) {
    if (atomic(P) or P == nil) return P   // P is not a pointer
    if not forwarded(P)                   // P stores a forwarding address after copied
            n = size(P)
            P  = free                     // reserve space in to_space for copy of P
            free = free + n
            temp = P[0]                   // P[0] holds forwarding address
            forwarding_address(P) = P
             P [0] = copy(temp)           // Restore P[0]
            for i = 1 to i = n – 1        // Copy each field of P in to P
                        P [i] = copy(P[i])
    return forwarding_address(P)          // Stored in P[0]
}
```

Basic copying collector

- Uses recursive call to copy
  - Recursive calls costs CPU time
  - Recursion stack occupies precious space
- Alternative:
  - Cheney's iterative copying collector
  - Just 2 pointers are needed: scan and free
  - Remember branch points of active graph as a queue
  - scan and free point to opposite ends of queue
    - Stored in new semi-space in objects already copied
  - Use tricolor abstraction

Cheney's copying collector

- Immediately reachable objects form initial queue of objects for a breadth-first traversal
- scan pointer is advanced from first object location by location
- Each encounter of pointer into from-space, pointee is copied to the end of the queue (in to-space) and the pointer to the object is updated

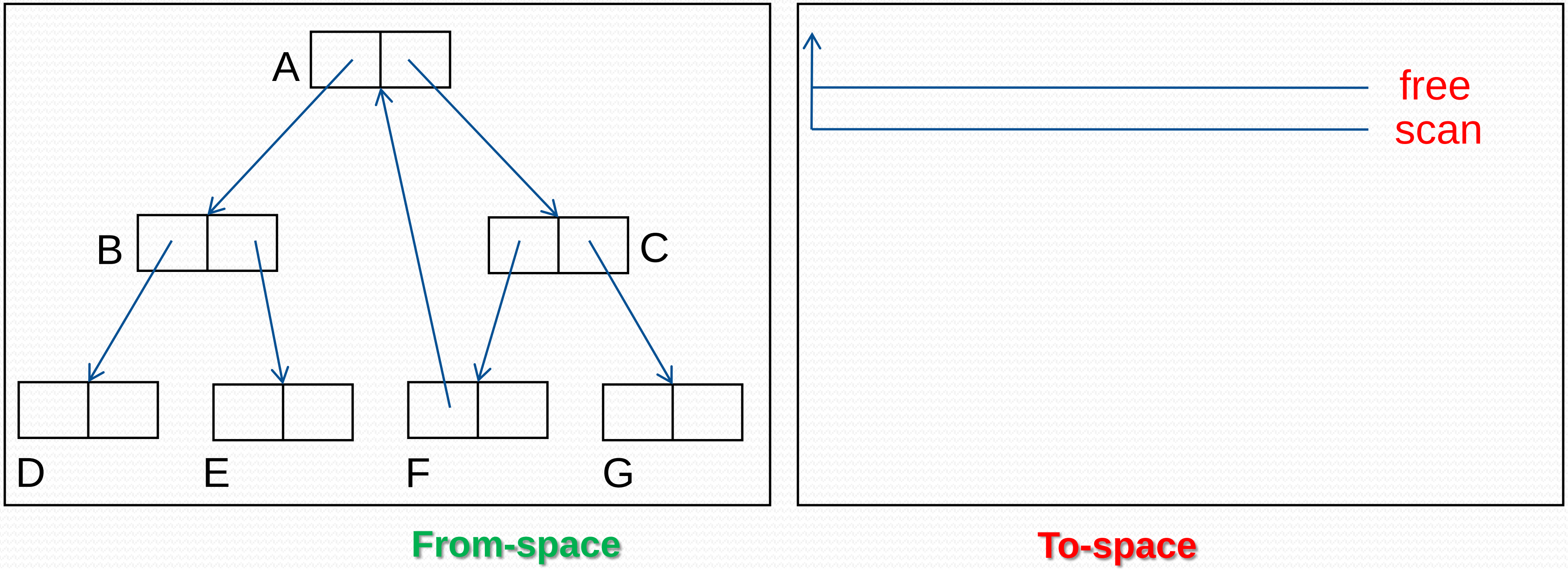
Cheney's copying collector

- When an object is copied to to-space, a forwarding pointer is installed in the old version of the object
- The forwarding pointer signifies that the old version of object is obsolete and indicates where to find replica

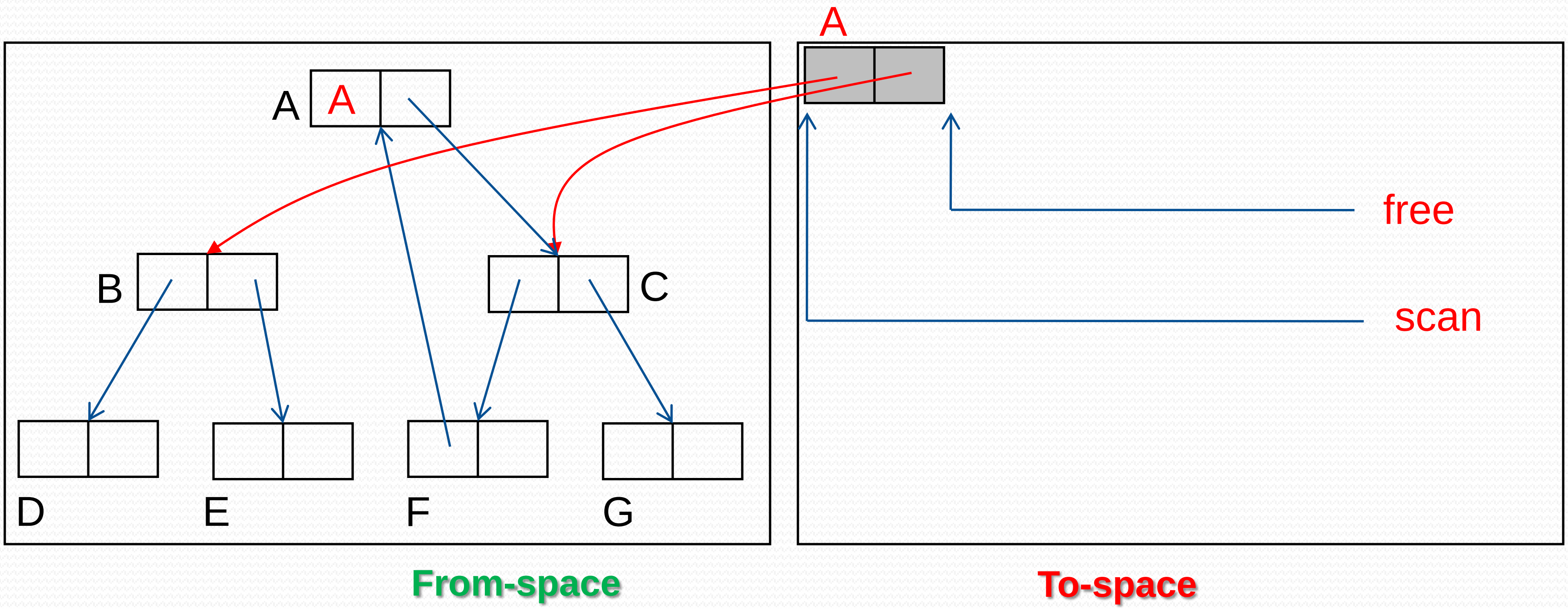
Cheney's tricolor abstraction

- Black:
  - Object and immediate descendents visited.
  - GC finished with black objects and need not visit them again
- Grey:
  - Object is visited but its descendents may not have been scanned
  - Collector must visit it again
- White
  - Object not visited and is garbage at end of tracing phase

Cheney's algorithm after the flip

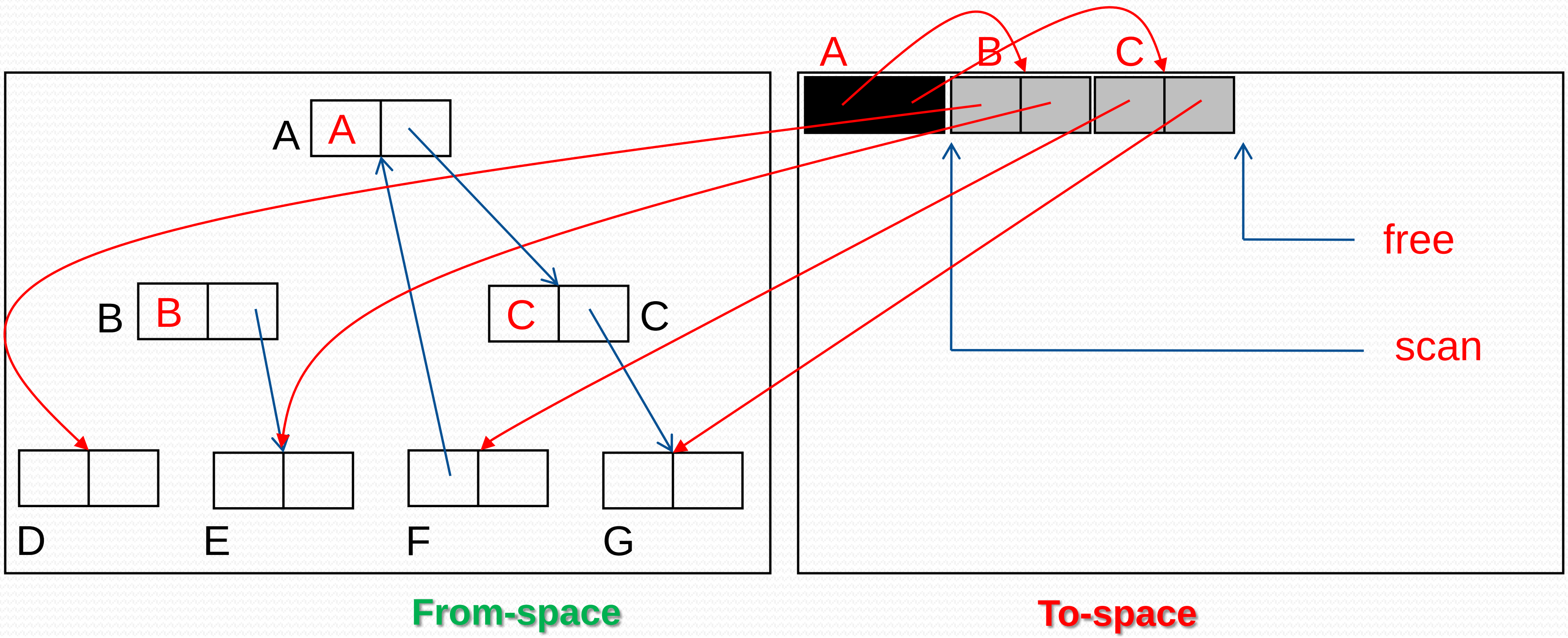


Roots of structure copied



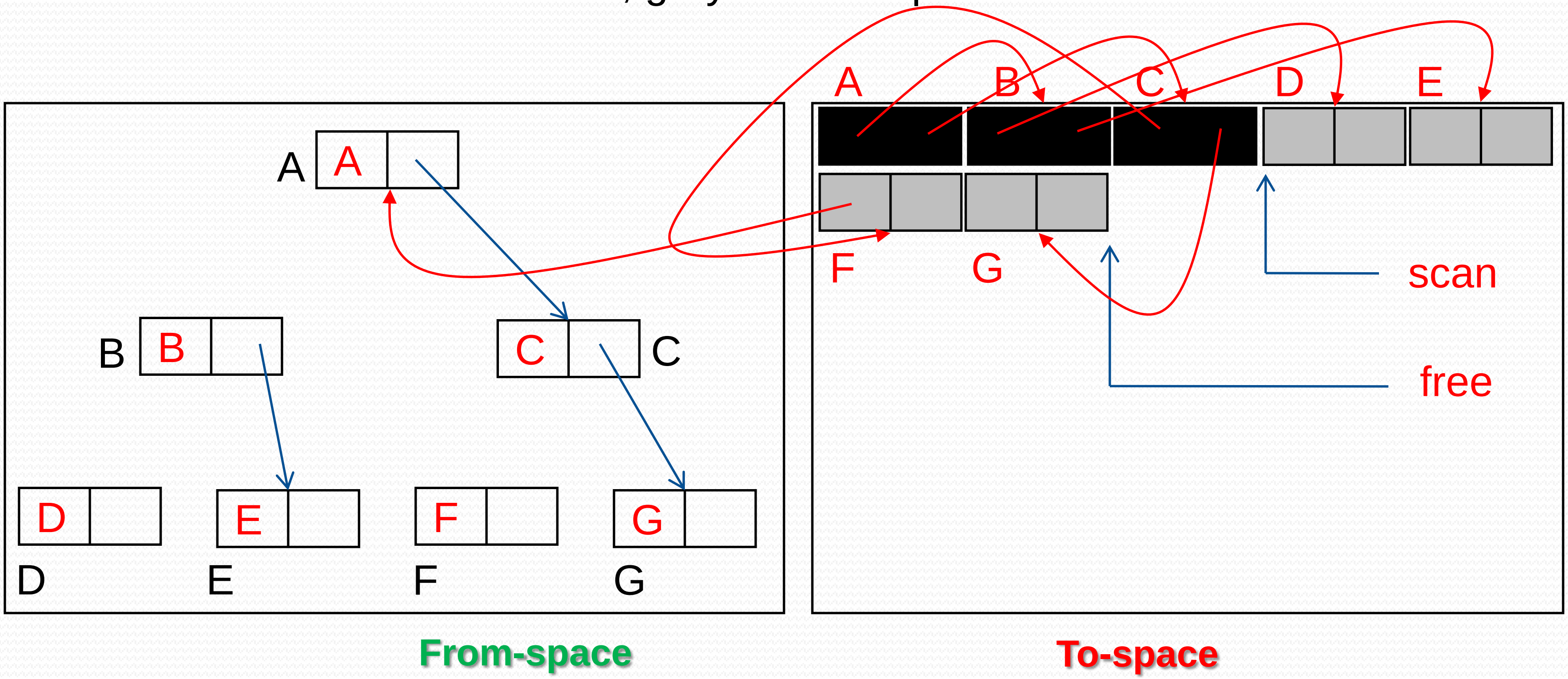
A scanned, copying B and C

Black nodes have been scanned; grey nodes copied but not scanned



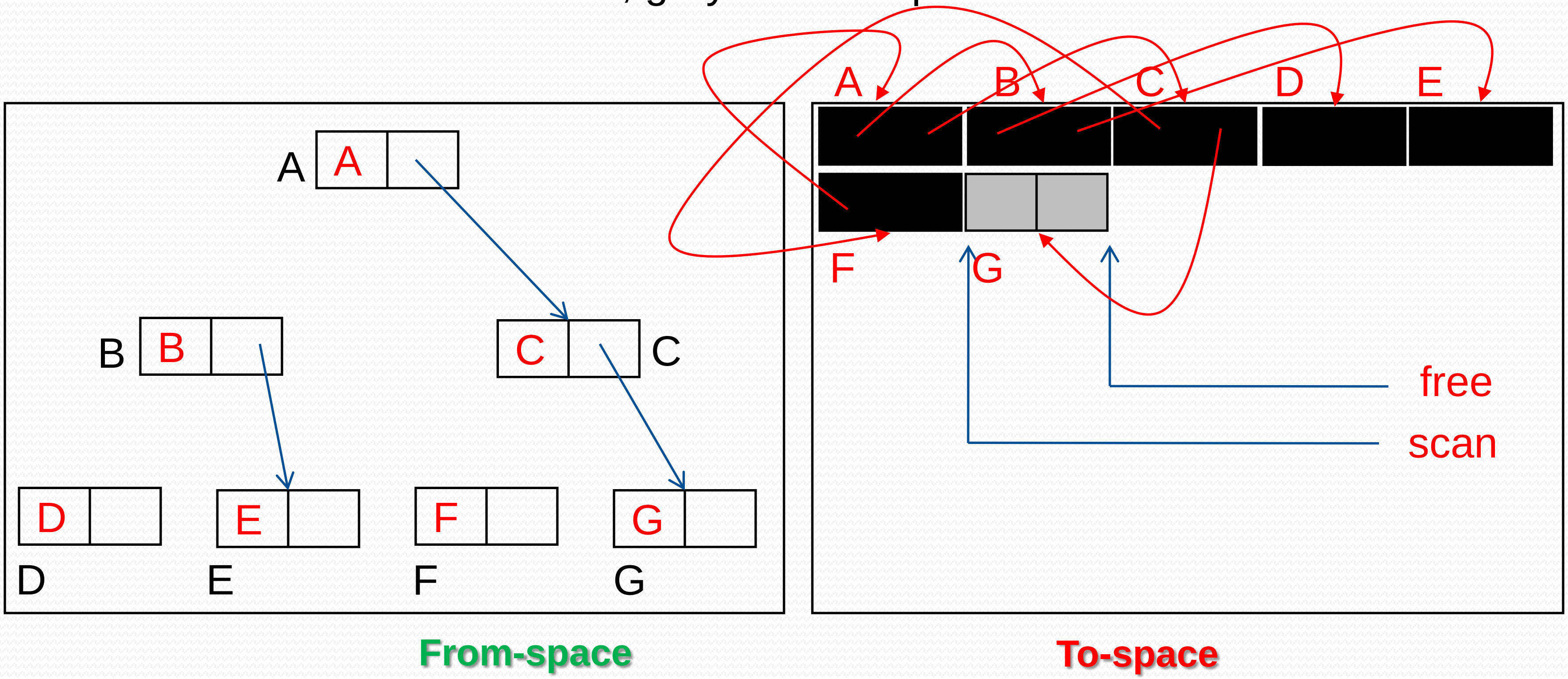
All from-space objects copied

Black nodes have been scanned; grey nodes copied but not scanned

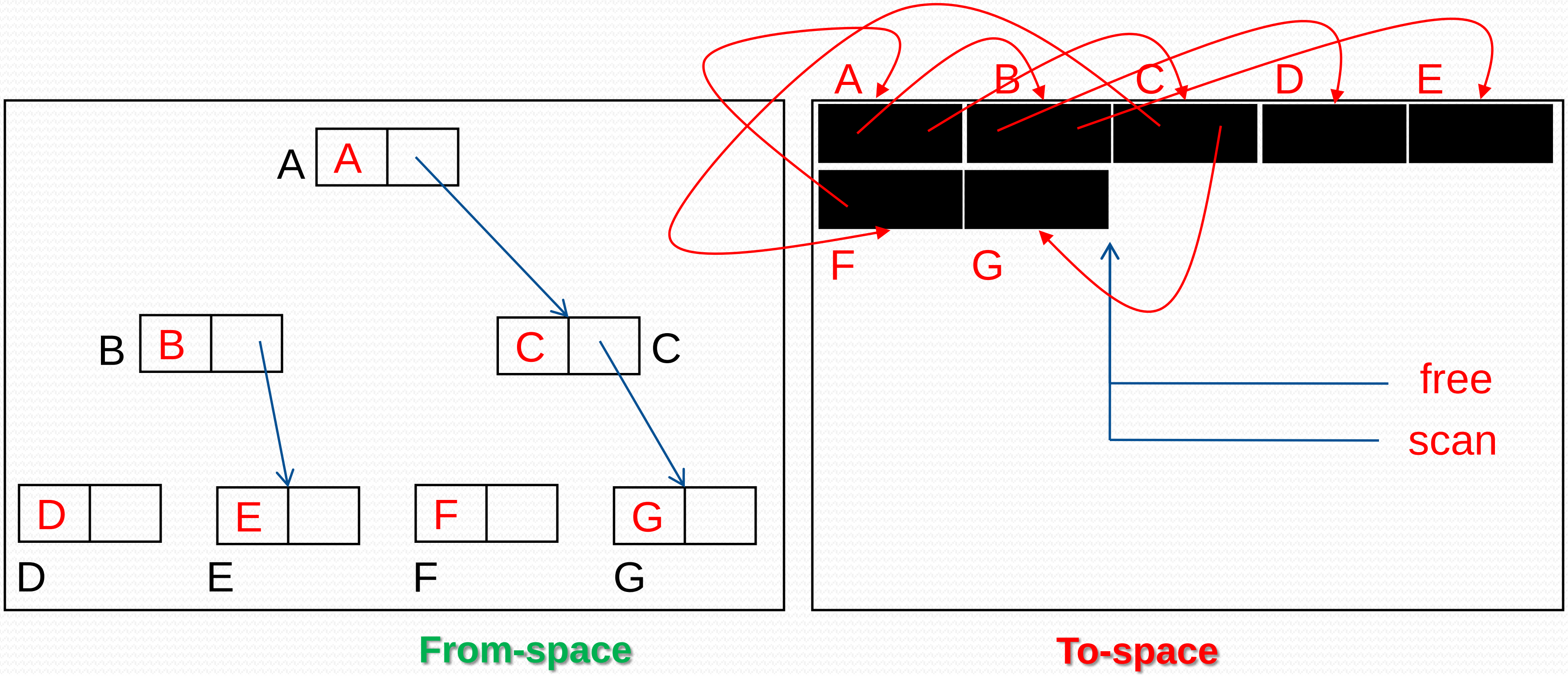


left(F) updated

Black nodes have been scanned; grey nodes copied but not scanned



Algorithm terminates



Cheney's algorithm

```
flip() {
    to_space, from_space  = from_space, to_space
    top_of_space = to_space + space_size
    scan = free = to_space
    for R in Roots
            R = copy(R)                    // Root pointer now points to copy of R
    while scan < free
            for P in children(scan)
                    *P = copy(*P)
            scan = scan + size(scan)
}
```

Cheney's algorithm

```
copy() {
    if forwarded(P)
            return forwarding_address(P)
    else
            addr = free
            move(P free)
            free = free + size(P)
            forwarding_address(P) = addr
            return addr
}
```

Multiple-area collection

- Problem:
  - CPU cost of scavenging depends in part on size of objects
    - Copying small objects no more expensive than marking with bitmap
    - Cost of copying large objects may be prohibitive
      - Typically contains bitmaps and strings (atomic)
- Solution:
  - Use large object space (separate memory region)
  - Assume objects have header and body
    - Keep header in semi-space
    - Keep body in large object space (use mark-sweep)

Multiple-area collection

- Problem:
  - Some objects may have some permanence
  - Repeatedly copying such objects is wasteful
- Solution:
  - Use separate static area
  - Do not garbage collect such region
  - Trace region for pointers to heap object outside static area
- Preview for generational garbage collection

Incrementally compacting collector

- Divide heap into multiple separately managed regions
  - Allows compacting of parts of the heap
  - Use mark-sweep or other approach on other regions
- Lang and Dupont:
  - Divide heap into n + 1 equally sized segments
  - At each GC cycle:
    - Choose 2 regions for copying GC
    - Mark-sweep other regions
    - Rotate regions used for copying GC
  - Collector chooses which transition to take next
    - Give preference to mark-sweep to limit growth of stack

Effects of incremental compactor

- Compact small fragments into single piece
- Compactor will pass through every segment of the heap in n collection cycle
- Small cost: extra segment used for a semi-space