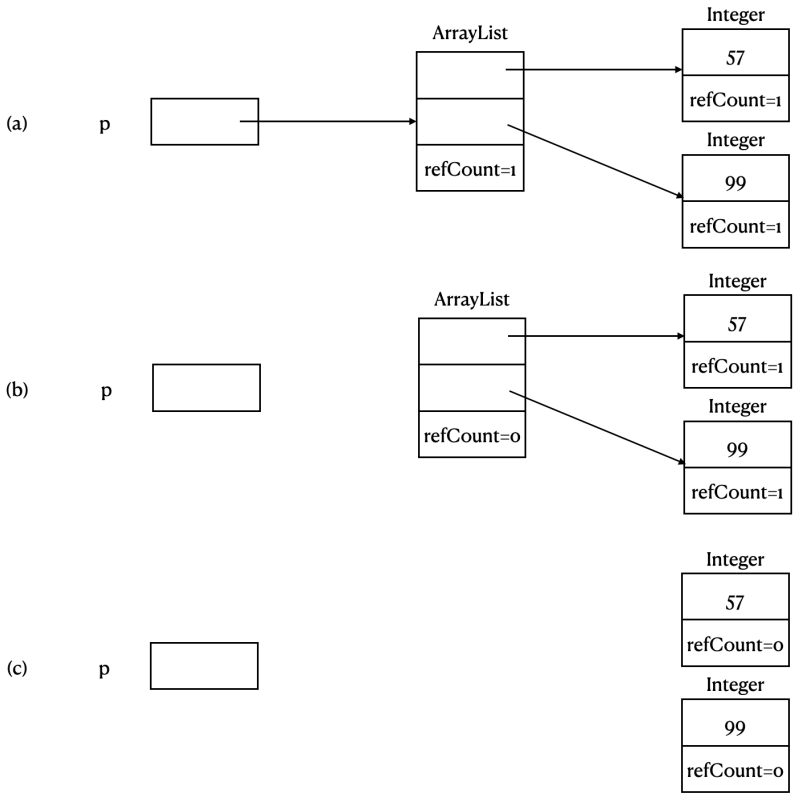
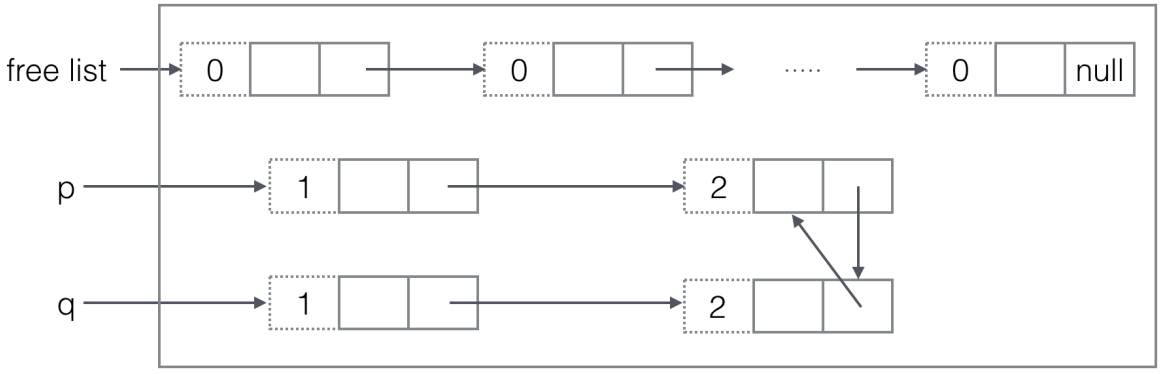
Memory Management (III)

Reference Counting (cont.)

- Example: `ArrayList<Integer> p = new ArrayList<Integer>(); p.add(new Integer(57)); p.add(new Integer(99));` What the memory for l looks like? What if I assign `p = null`? what the memory looks like?



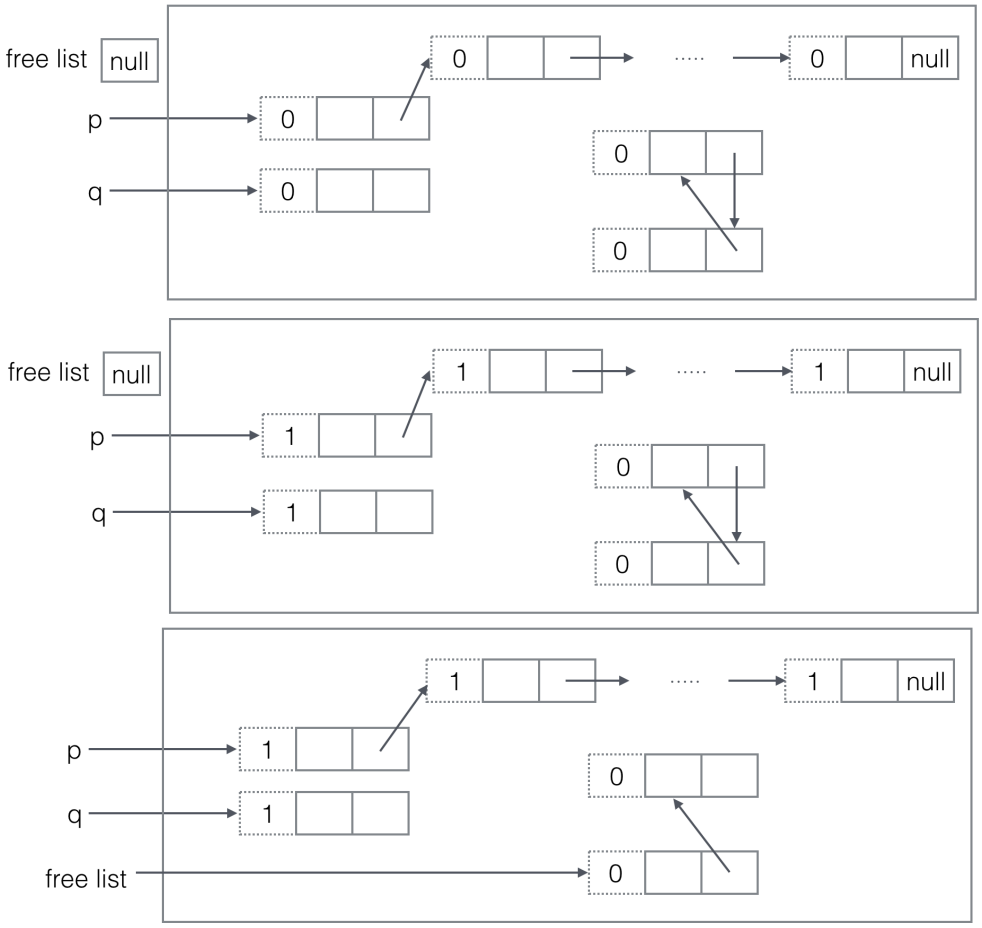
- Anytime the program **deletes a reference** to a node, **or the reference's lifetime ends**, the program has to **decrement the reference counter** of the node.
- If a **counter becomes zero**, because of either an assignment or a deletion, then the **program must travers any children it references** and **see if their counters are now also zero**. Any **nodes with zero counters go back to the free list**.
- So far, reference counting looks like a good idea. However, it does not always work.



- Advantage: Does not require time in large chunks to clean up memory.
- Disadvantage: Cannot detect inaccessible rings of nodes.

Mark-sweep

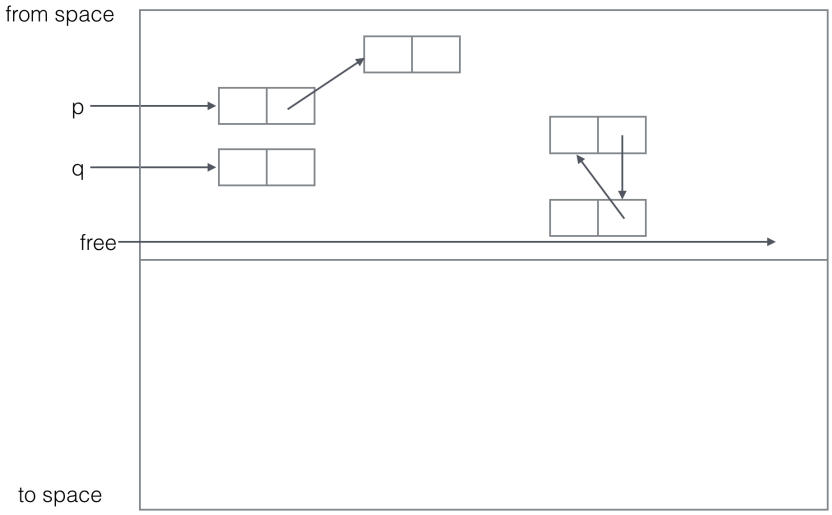
- **Mark-sweep** was the **first garbage collection** algorithm to be developed that is **able to reclaim cyclic data structure**.
- When using marking-sweep, **unreferenced memory blocks are not reclaimed immediately**. Instead, garbage is allowed to **accumulate until all variable memory has been exhausted**.
- Mark-sweep consists of **two phases**: the **mark phase** and the **sweep phase**.
- Instead of a counter, **each node has a single boolean that is initially zero, which mean unmarked**.
- The **mark phase starts with each reference in the active symbol table, follow that reference through memory, setting each visited node's boolean marker to 1**.
- **Once the reference traversal is complete, run through the entire heap and out any node who's marker is 0 back to the free list**. This is the sweep phase.

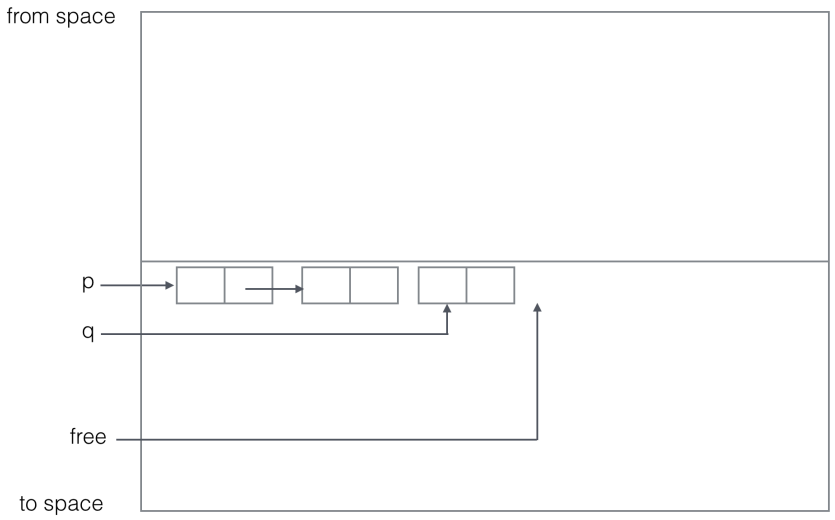


- Mark-sweep **trace out the memory blocks that are in use**. So it is **able to** correctly **identify and collect garbage** even in the presence of **reference cycles**.
- The **issue** of mark-sweep is that when the algorithm is triggered, the **execution of the program is suspended temporarily**. Once all unreferenced objects have been reclaimed, the normal execution of the program can resume.

Copy Collection

- Copy collection is also called **defragment collection**.
- When using copy collection, the **heap is divided into two separate regions**.
- At any point in time, all dynamically **allocated blocks reside in only one of the two regions**. We call this region **active region**. the other region is unoccupied, which is called **inactive region**.
- When the **memory in the active region is exhausted**, the **program is suspended and garbage collection is invoked**.
- Copy collection algorithm **copies** the memory block of all **active references from the active region to the inactive region**.
- **As** each memory block is **copied**, the **reference is updated to the new location**.
- **After** the **copying is completed**, the **active and inactive region exchange their roles**.
- **Since** the copy collection algorithm **copies only the memory block of active references**, **the garbage blocks are left behind**.
- The **memory space occupied by the garbage is reclaimed all at once when the active region becomes inactive**.
- **As** the copy collection algorithm **copies blocks from active region to inactive region**, **it stores these blocks in contiguous memory locations**. Therefore, this algorithm **automatically defragments the heap**.





- The copy collection algorithm can detect inaccessible rings of nodes and can defragment. However, if an application program has a large memory footprint, the time required to copy all objects can be quite significant. Besides, it requires twice as much memory as the program actually uses and need to suspend the current running program while execution.