CS 2002D PROGRAM DESIGN

Garbage Collection #1

CSED, NIT CALICUT

14.11.2022

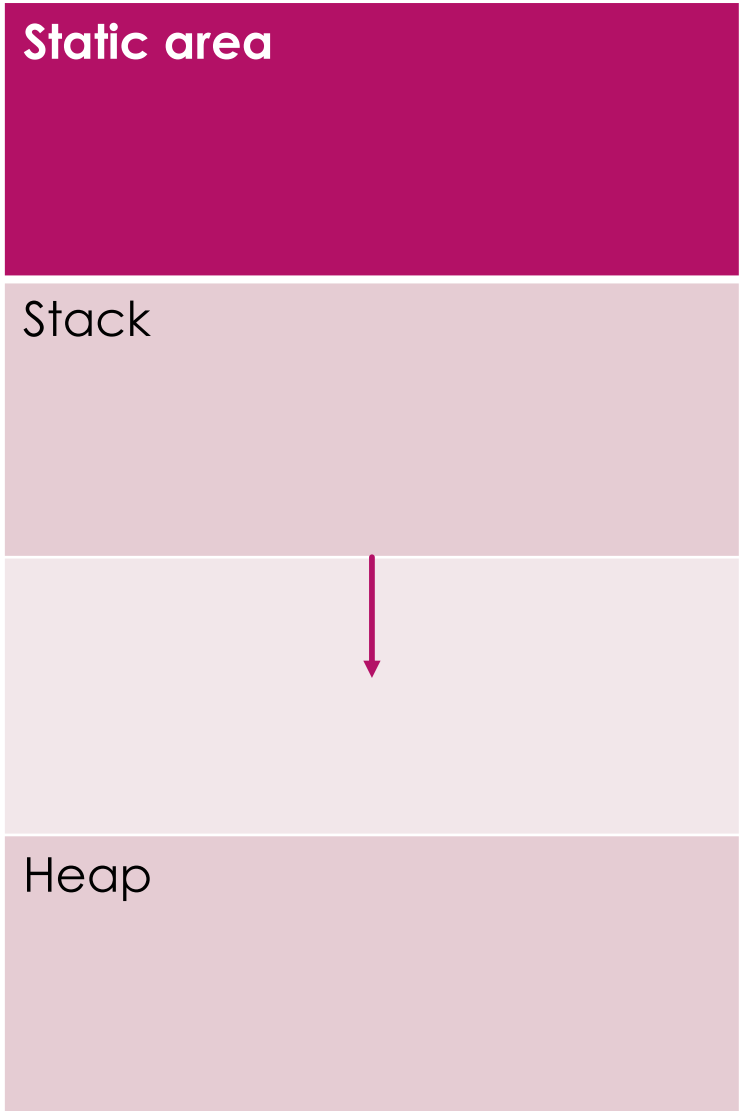
Courtesy : Dr. SALEENA N

Overview

- Runtime management of dynamic memory
  - Tricky and error-prone
  - C, C++ - assumed to be either by the programmer or by the system
  - JAVA – Only the system can be responsible
  - Automation of runtime memory management

Run time memory

- In languages like C, C++ and JAVA memory at run time can be viewed as having three parts
  - The static area
  - The run-time stack
  - The heap


Static area
Stack
Heap

The Heap

- **Dynamic allocation and deallocation of storage blocks of different sizes**
  - ***new* and *delete* calls (malloc, free)**
  - **Heap can eventually become fragmented**

The Heap

- **Heap Overflow**
  - **A call to new occurs and the heap does not contain a contiguous block of unused words of the required size**

Garbage Collection

- Garbage Collection algorithms to manage heap memory
  - To utilise the available space efficiently

Garbage

- **Garbage**
  - Any block of heap memory that cannot be accessed by the program
  - No pointer accessible to the program to reference the block

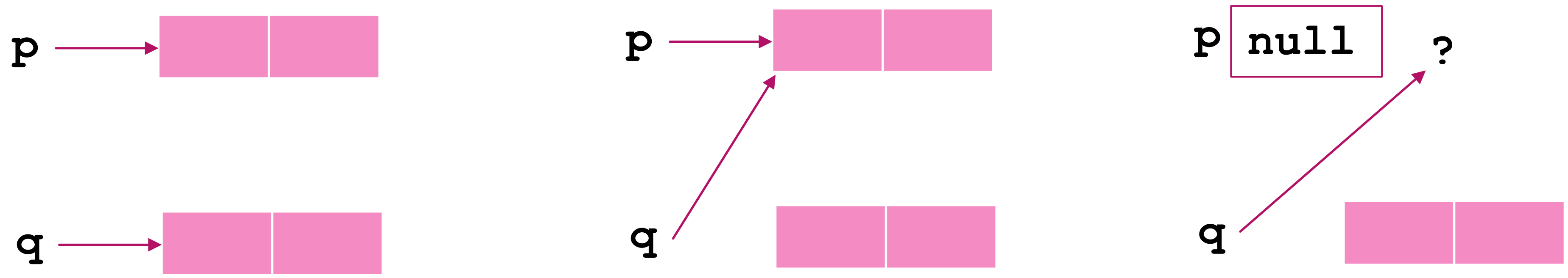
Memory Leak

```
p = new node();
q = new node();
q = p;    //creates a memory leak
```

Allocated area, but not
accessible, garbage (orphan)

Dangling Reference

```
p = new node();
q = new node();
q = p;
delete (p); // where does q point to?
```



Allocated area,
but not
accessible,
garbage

Dangling
Reference

Garbage

- Garbage
  - Inactive objects in heap
  - Blocks that are allocated earlier but no longer needed

Garbage Collection (GC)

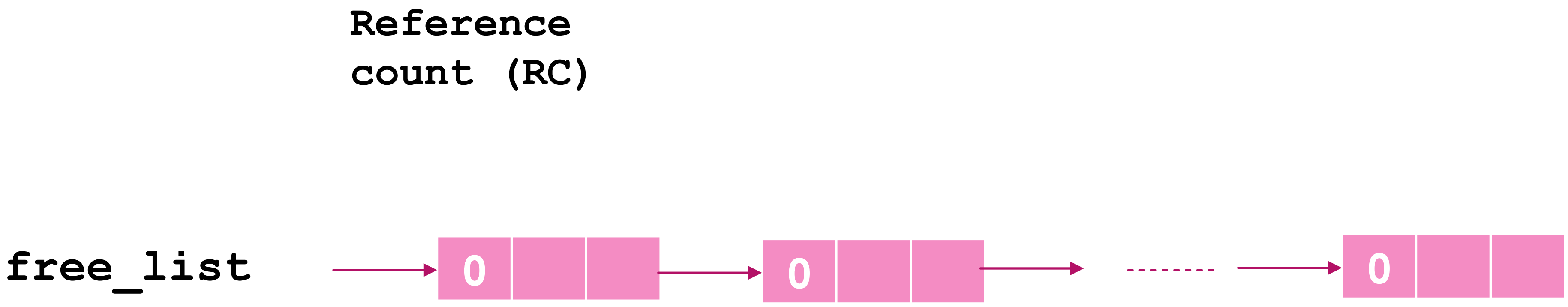
- Reclaim blocks that are garbage, so that these blocks can be reused
- Started with functional languages like Lisp
  - List Processing, John McCarthy, AI
- Automatic Garbage Collection - JAVA, Lisp
- Programmer's responsibility – C, C++
  - Explicitly free blocks
  - Supporting tools

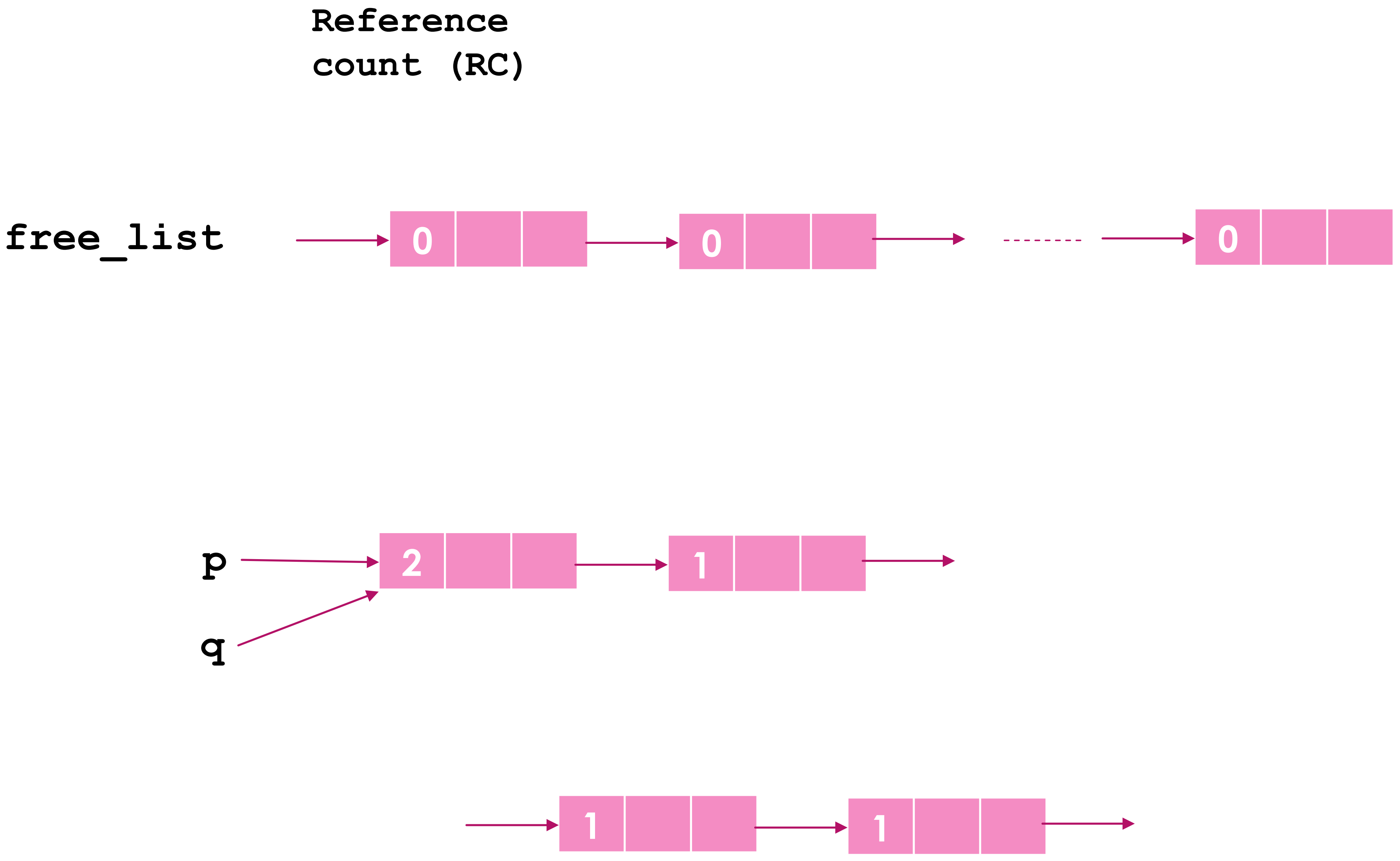
Garbage Collection

- Three major strategies
  - Reference Counting
  - Mark-sweep
  - Copy collection

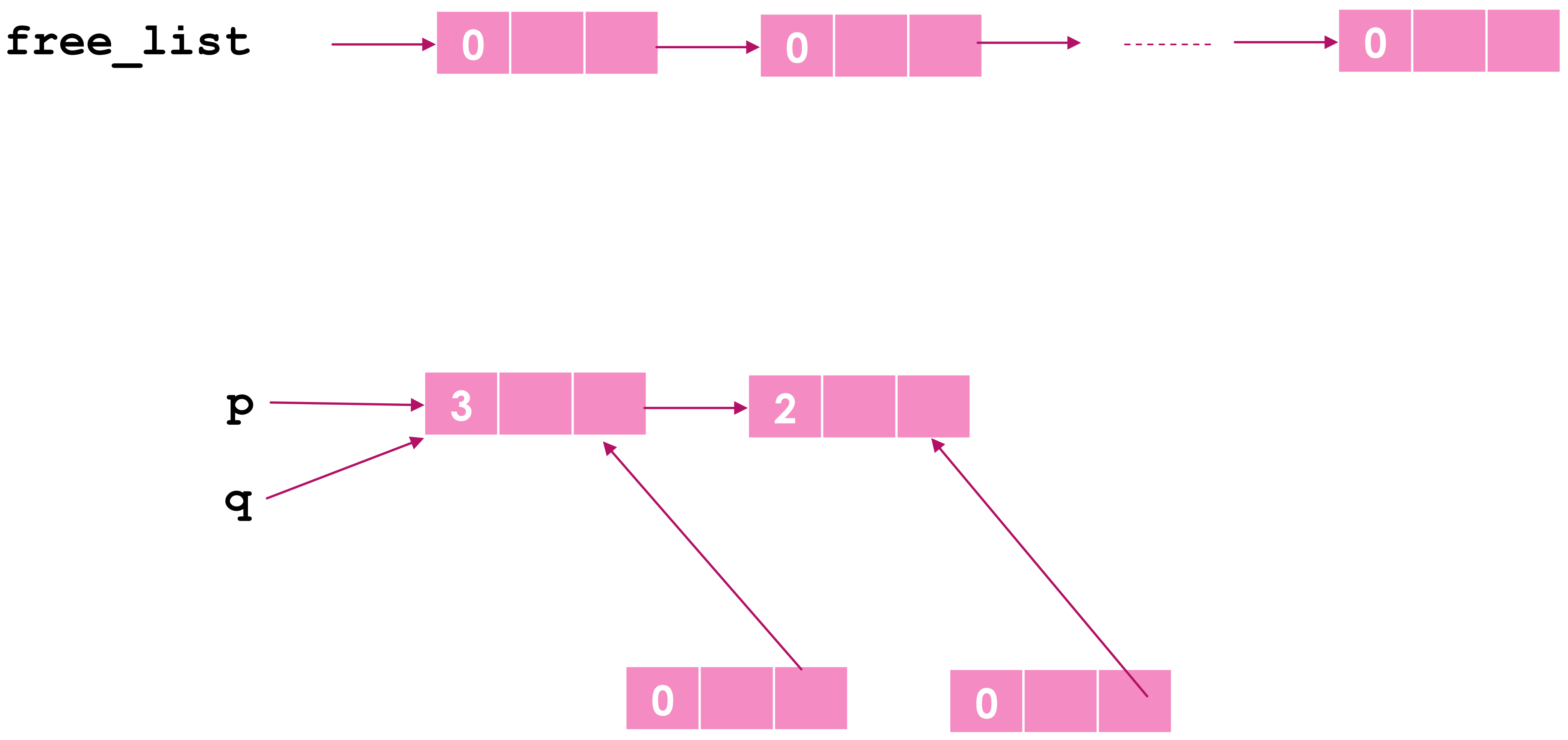
Reference Counting

- Assumes -initial heap is a continuous chain of nodes, *free_list*
- Each node keeps a count of the number of pointers referencing that node – Reference Count (RC)
- RC is initially set to 0
- RC incremented when the node is allocated (removed from *free_list*)
- RC updated whenever *new* or *delete* is called

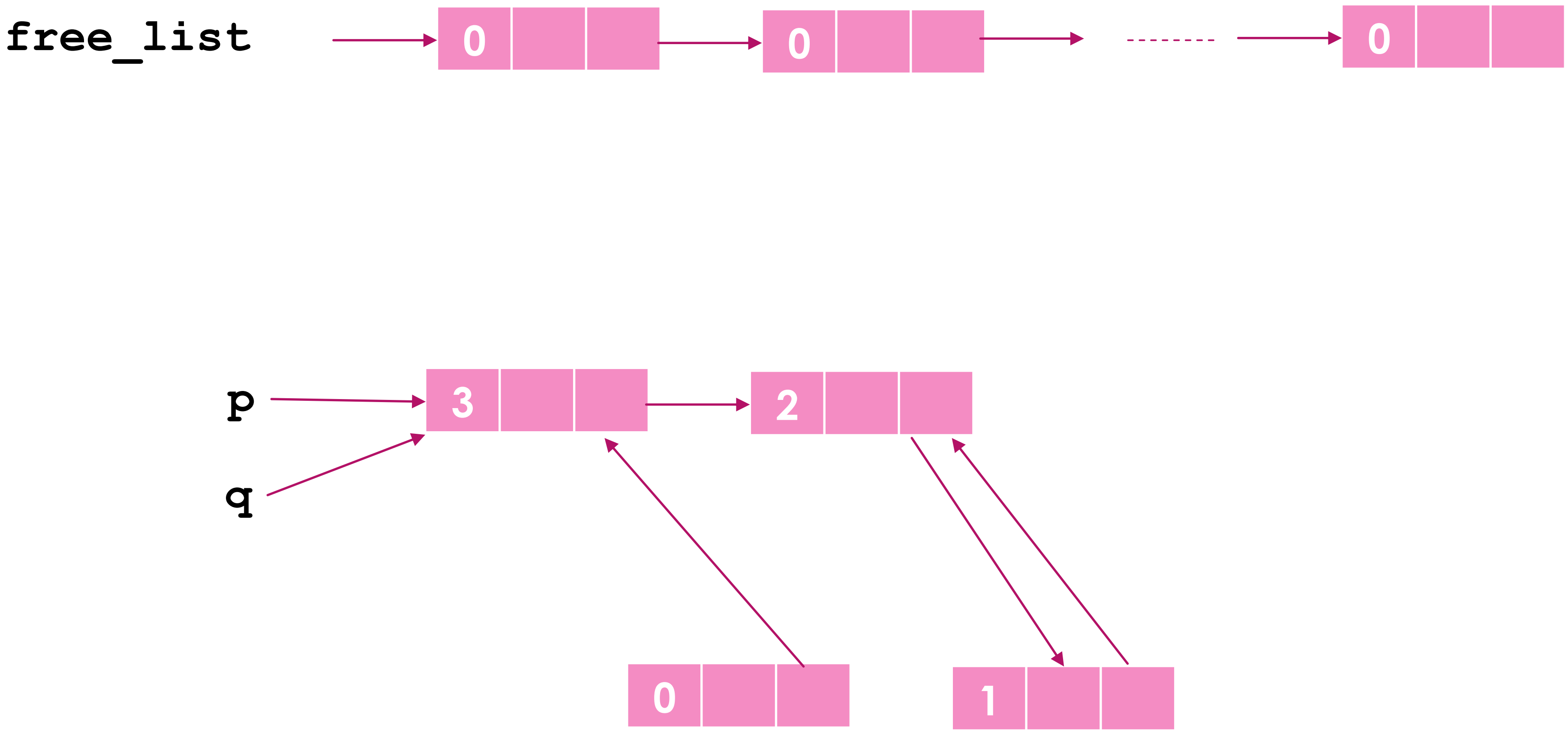

Reference
count (RC)
free_list
0
0
0


Reference
count (RC)
free_list
0
0
0
p
q
2
1
1
1

Reference
count (RC)



Reference count (RC)



Reference Counting

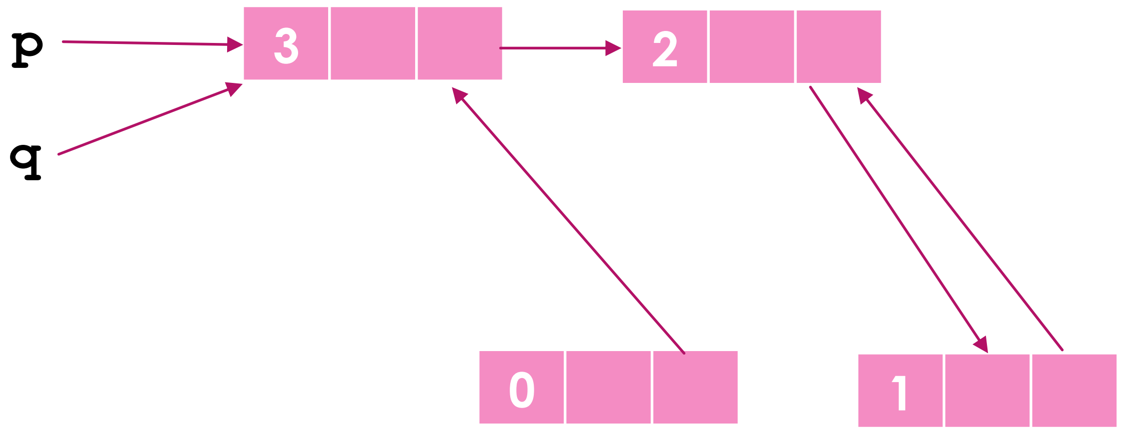
- *new* – allocates a node from *free_list* and sets its RC to 1
- *delete* – RC set to 0 and node returned to *free_list*

Reference Counting

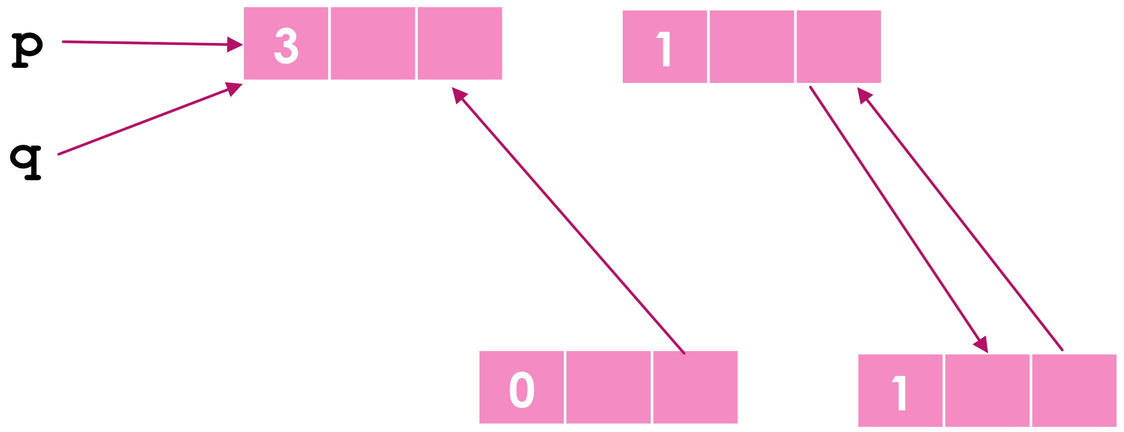
- For pointer assignment q=p
- RC of p increased by 1
- RC of node earlier pointed to by q, say node x, is decreased by 1
- if RC of node x becomes 0, return x to *free_list* ?

Reference Counting

- For pointer assignment q = p
  - RC of p is increased by 1
  - RC of node earlier pointed to by q, say node x, is decreased by 1
  - if RC of node x becomes 0, RC of each descendent of x is decreased by 1 and x is returned to *free_list*, and this step is repeated for each descendent of x
  - The pointer q is assigned the value of p



`p.next = null ?`



Two isolated nodes with RC =1 pointing to each other
Algorithm fails to return these to *free_list*

Reference Counting

- Invoked dynamically - whenever a pointer assignment or other heap action is triggered by the program
- Overhead associated with GC is distributed over the run-time life of the program
- Fails to detect inaccessible circular chains
- Storage overhead for keeping RC
- Performance overhead

Garbage Collection

- Three major strategies
  - Reference Counting ✔
  - Mark-sweep
  - Copy collection

Mark-Sweep

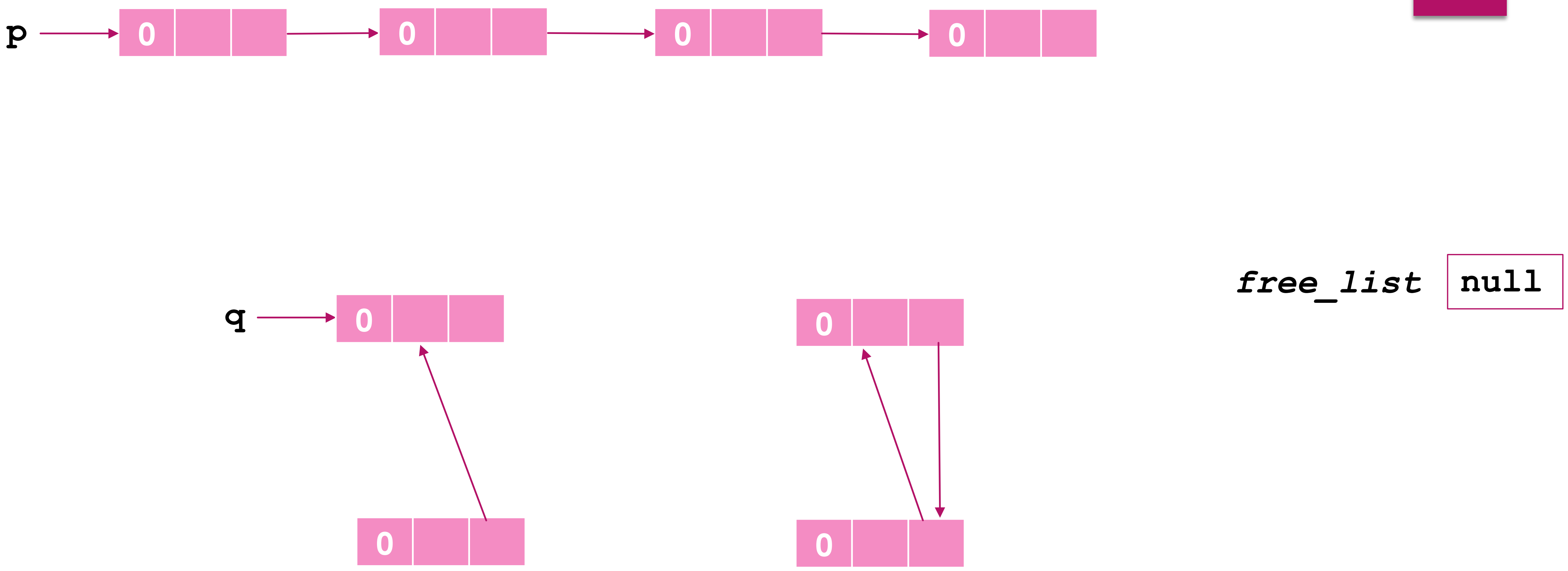
- Called into action only when the heap is full
- Allocation, deallocation, pointer assignments can be without GC overhead
- Once invoked GC is time-consuming

Mark-Sweep

- Invoked when a new node is requested and the heap is full
- Makes two passes on the heap
  - Mark pass – every reachable heap block is marked
  - Sweep pass – returns all unmarked nodes to the free list

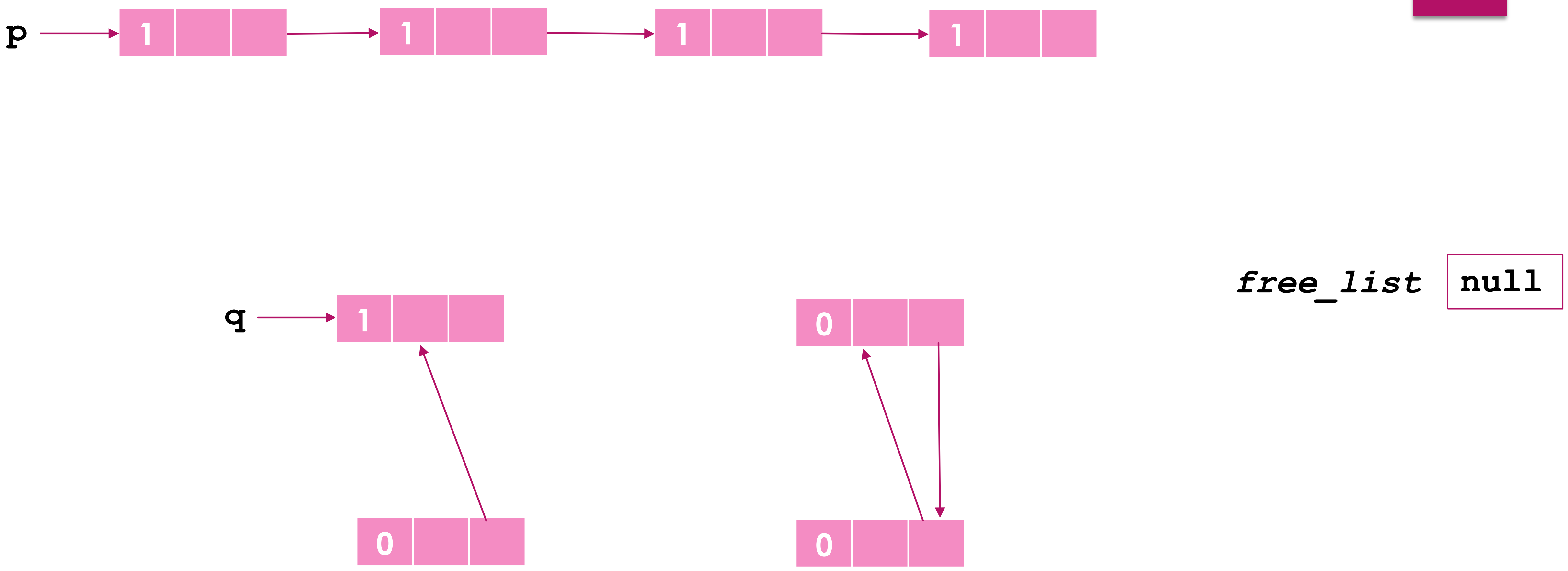
Mark bit

- A mark bit attached to each heap node
  - initialized to 0
  - set to 1 if node is reachable

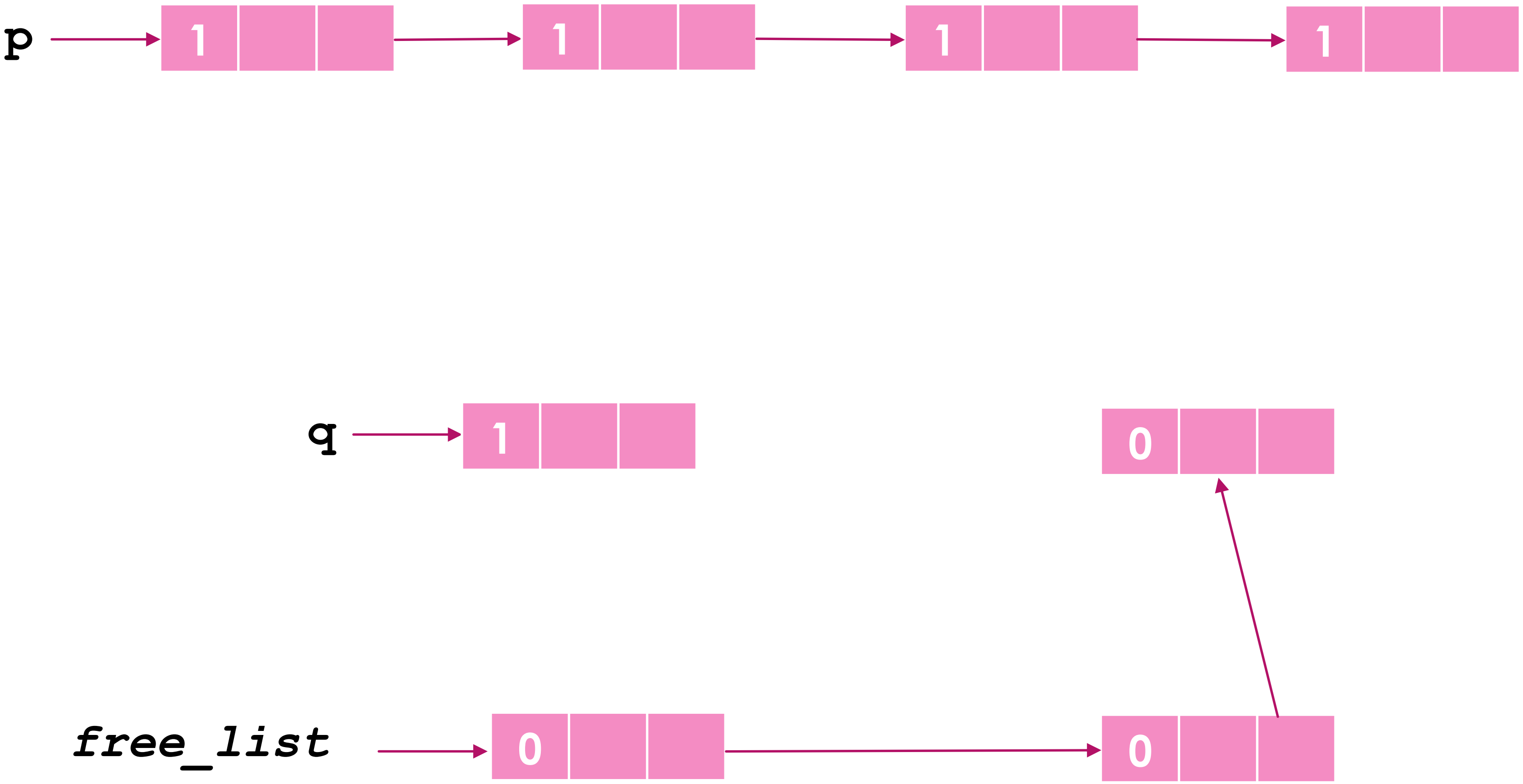

p
0
0
0
0
q
0
0
0
0
free_list
null

Mark pass

- Every reachable heap block is marked
- Reachable?
  - Reachable by following a chain of pointers originating in the run-time stack
    - Local variables in the activation record



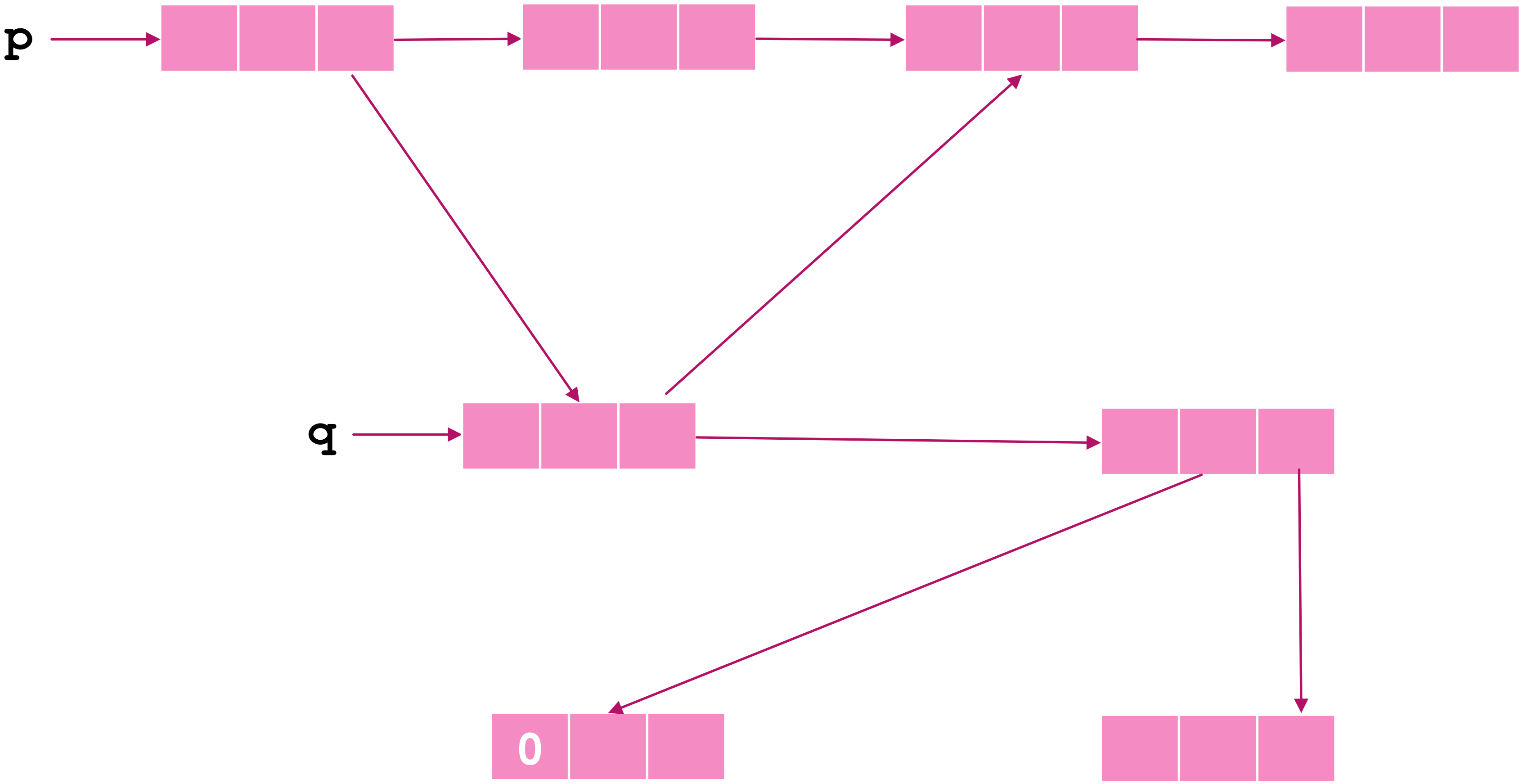
Heap after Pass I



Heap after Pass II

Mark and Sweep

- Cells with more than one pointer field
  - Tree node with left and right child links
- Reachable nodes
  - Reachable through any of the links



Nodes reachable from p and q to be explored

Mark and Sweep

- Depth First Search (DFS)
  - Recursive – space overhead
  - Non recursive algorithms preferred
- From which node to start DFS?
- Graph Search Algorithms

Garbage Collection

- Three major strategies
  - Reference Counting ✔
  - Mark-sweep ✔
  - Copy collection

Copy Collection

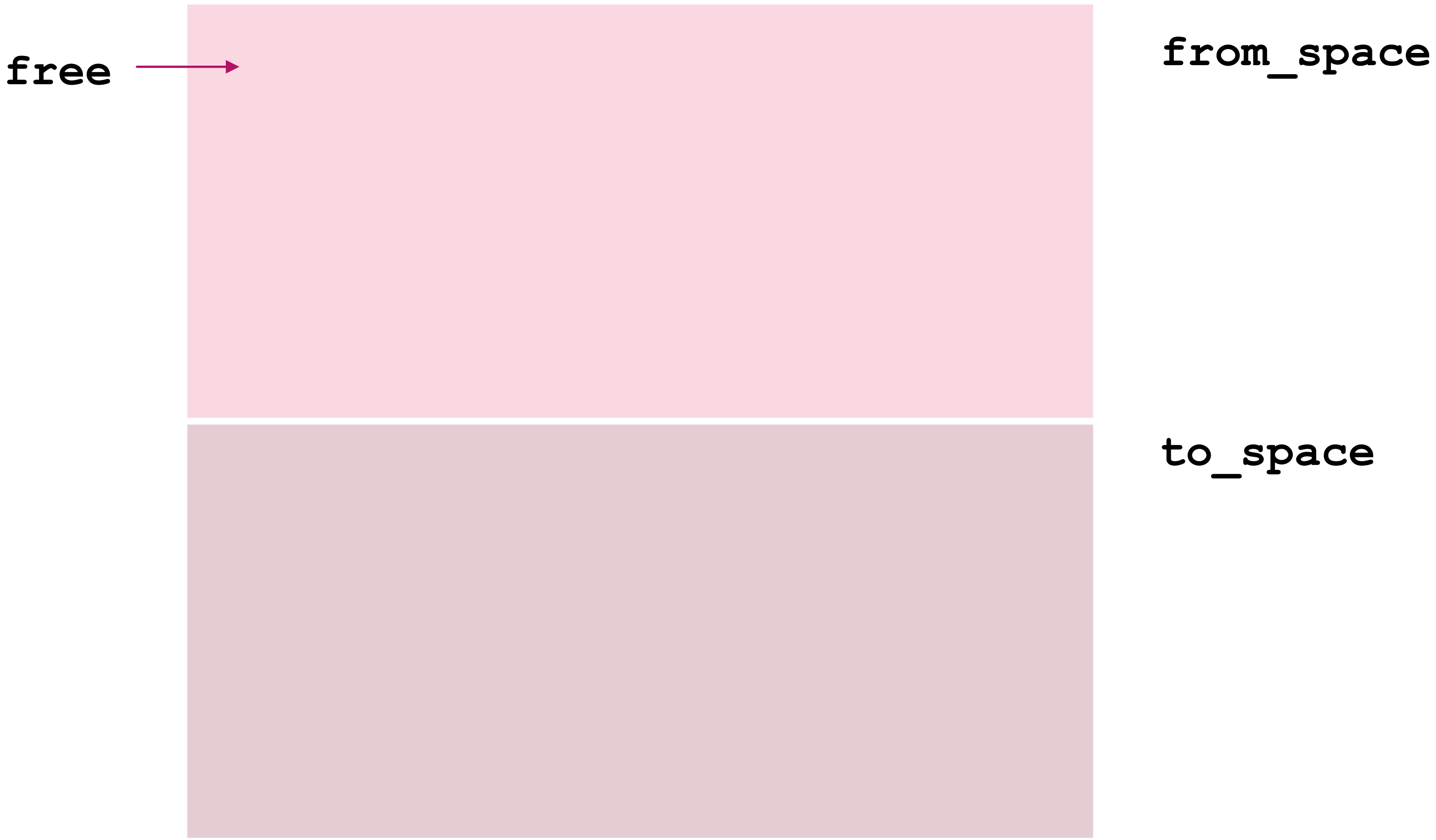
- Called only when the heap is full
- Makes only one pass in the heap

Copy Collection

- Heap divided into two identical blocks
  - From-space
  - To_space
- No extra mark field
- No free_list

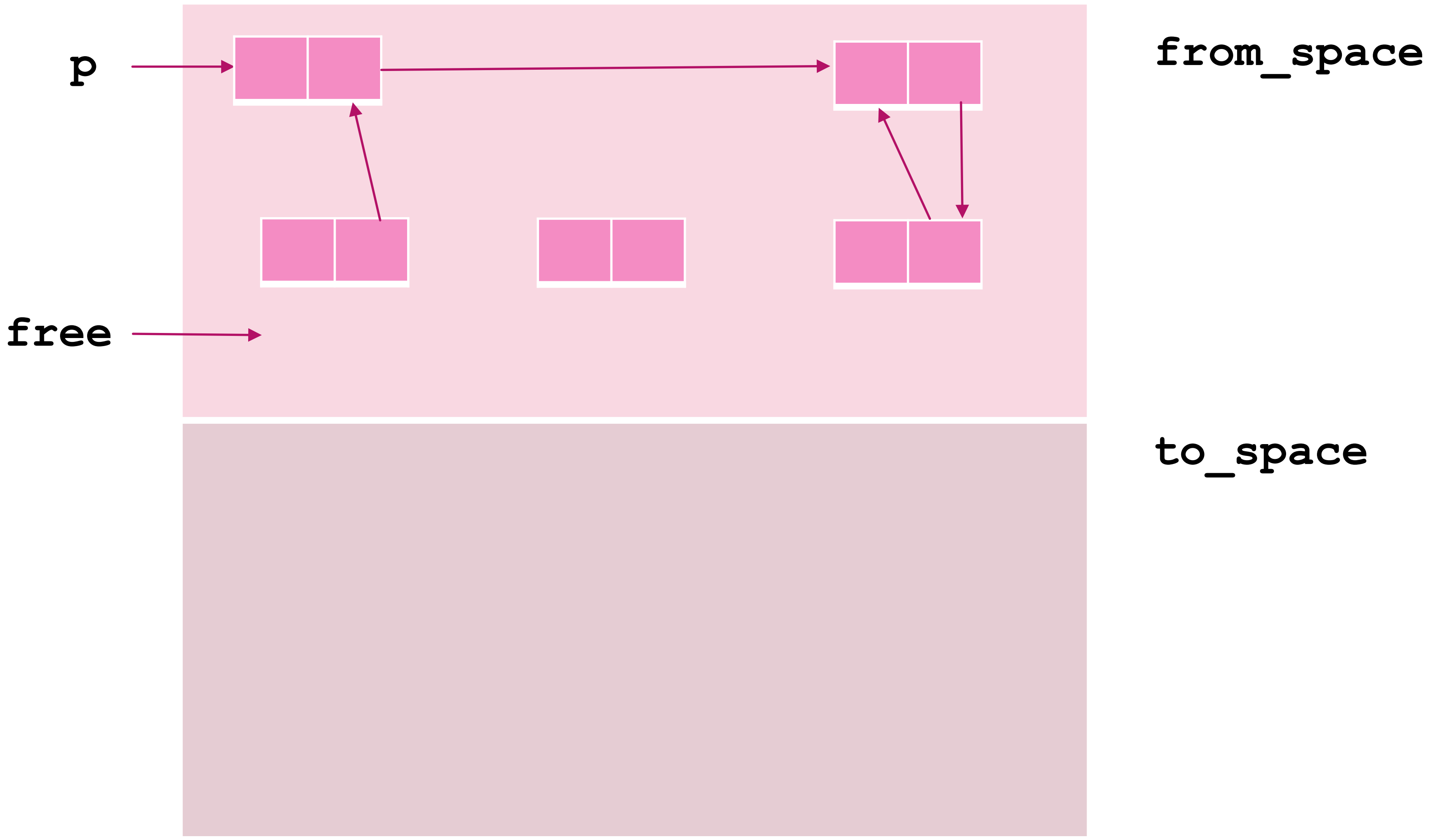
Copy Collection

- Initially heap is divided into two identical blocks
  - *from-space*
  - *to_space*
- All active nodes in *from_space*
- *to_space* is unused
- A pointer *free* points to the end of the allocated area in the *from_space*


free
from_space
to_space

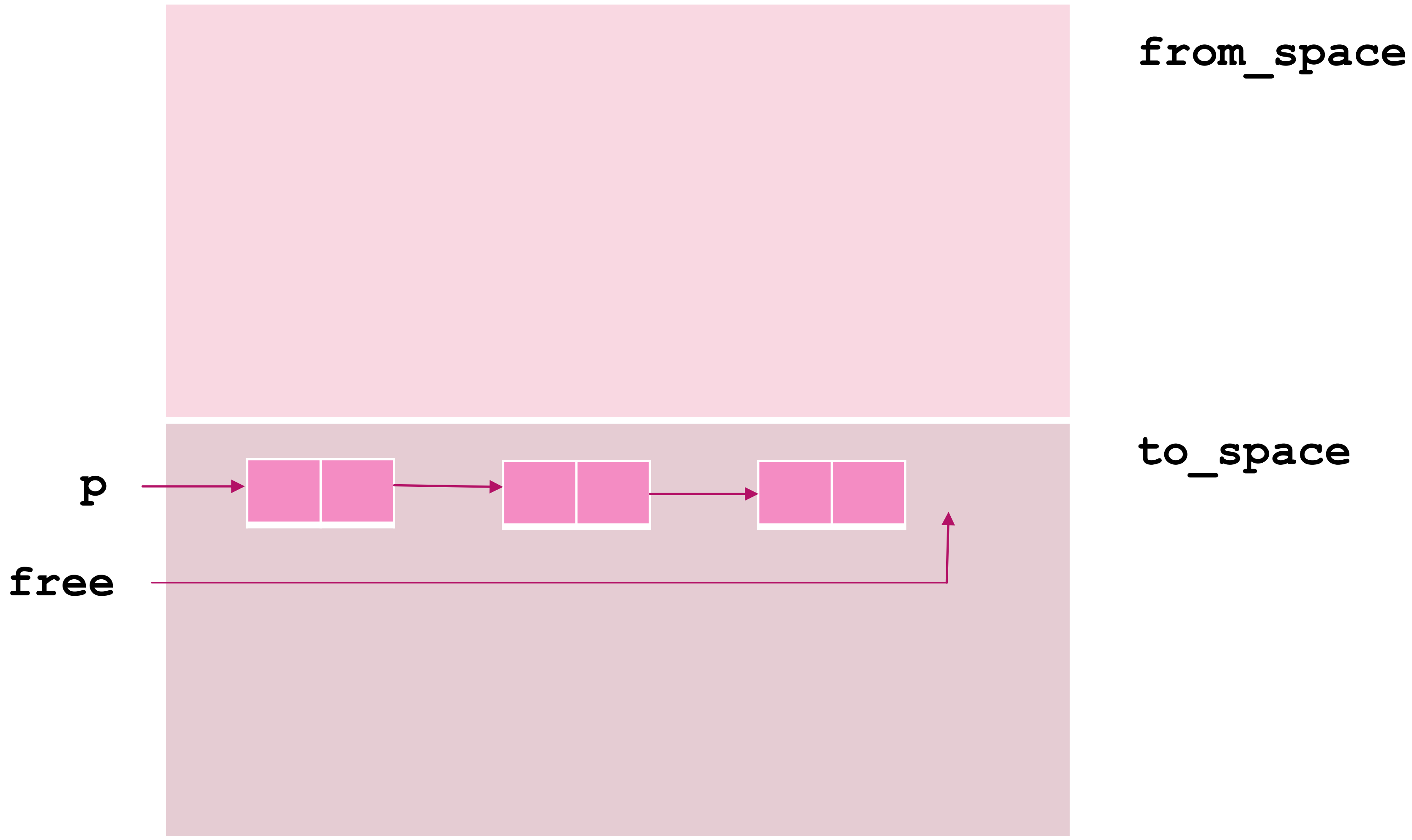
Copy Collection

- Allocating new node
  - Next available block referenced by free is allocated
  - Free is updated


p
free
from_space
to_space

Copy Collection

- No free memory available for allocating new node
  - Invoke copy collection
    - Allocated blocks copied to the *to_space*
    - Roles of *to_space* and *from_space* are reversed
    - Eliminates all inaccessible nodes in *from_space* and tightly repacks the active nodes in *to_space*


from_space
to_space
p
free

Copy Collection

- **Faster**
- **No extra bit for reference count or marking**
- **No *free_list* to be maintained**

Conclusion

- Other more complex GC algorithms
  - Generational GC – heap divided into generations
- Hybrid approaches – selecting between mark and sweep and copy collection depending on certain parameters
- Run GC as a low priority process (thread)- executes whenever demand on processing time is low
  - Reduces GC calls during peak processing time
- Explicit calls to GC as in Java

Memory Management

Managing heap memory

- Allocation
  - Algorithms
    - First-fit, Best-fit, Buddy System
- Compaction

Memory

- Static Area
- Stack Area
- Heap Area

Heap

- Blocks of memory can be allocated and deallocated dynamically (during program execution)
- Allocation as a block of the required size (or greater) from the available memory space
- Memory Management Algorithms
  - Allocation, deallocation, compaction

Free list

- List of heap blocks that are currently not in use (free)
- Initially the entire heap is free
  - a single element in the list
- Allocation algorithm searches the list for a block of appropriate size
  - Principal Concerns – Speed and Space
    - Space concerns related with fragmentation

Heap Memory Management

- Initially single chunk of free memory
- Eventually divided into
  - Allocated blocks
  - Free blocks
- Free blocks – linked together as free list
  - Whenever a block becomes free it can be added to the front of the free list
  - Fragments in the free list

Fragmentation

- Internal Fragmentation
  - Allocates a block that is larger than requested – extra space within the block unused
    - Block of size 500 allocated for a request of size 400 – 100 Bytes unused
- External Fragmentation
  - Free space is scattered into multiple blocks but not a single one is large enough to satisfy a request
    - Free blocks of size 32, 64, 32, 16 available – request for 128 can not be satisfied

Allocation - Algorithms

- First-fit : Find the first block that is large enough to satisfy the request
  - For a request of size *d* allocate the first available block whose size is >=*d*
  - *1000, 2000, 600, 526, 800, 64*
  - *Request for 400 – block of size 1000 allocated being the first block*
  - *Internal Fragmentation*

Allocation - Algorithms

- **Best-fit : Find the smallest block that is large enough to satisfy the request**
  - **search the entire free list to find the best fitting block whose size >=*d***
  - **To reduce fragmentation**
  - **Slower than first-fit**
  - **Free list : 1000, 2000, 600, 526, 800, 64**
  - **Request for 400 – block of size 526 allocated**

To minimise Internal Fragmentation

- If the block to be allocated is sufficiently large
  - divide the block into two before allocation
  - If the left over portion is large return to free list
  - otherwise (if size is below some minimum threshold) keep it in the allocated block

Compaction of empty blocks

- **Whenever a block becomes free check the possibility of combining with an adjacent block**
- **Block beginning at p of size s is returned**
  - **Look for a block beginning at p+s (block to the right)**
  - **If right adjacent block is empty, both can be combined**
    - **Remove the right adjacent block from the linked list**
      - **Efficient algorithms?**
  - **Finding left adjacent free block is more complicated**
  - **Alternatively, Initiate Merge only when there is a request that can not be satisfied**

Multiple free lists

- Cost of allocation with a single free list
  - Linear in the number of free blocks
- Reduce search time by maintaining separate free lists for blocks of different sizes (standard size)
  - Eg : A list for blocks of size 32, another for blocks of size 64 etc..
- Request for size d rounded up to the nearest standard size
  - Search and allocate from the appropriate list
- Heap is divided into *pools* – one for each standard size

Dividing the heap

- **Heap is divided into *pools* – one for each standard size**
- **Division may be static or dynamic**
- **Buddy System – a common mechanism for dynamic pool adjustment**

Buddy System

- Blocks come only in certain sizes
- Say $s_1 < s_2 < s_3 < \ldots . < s_k$ are all the sizes
- Common choices for size
  - 1, 2, 4, 8, …. (block sizes are powers of 2)
    - Exponential buddy system ($s_{i+1} = 2 * s_i$)
  - 1, 2, 3, 5, 8, 13….
    - Fibonacci buddy system ($s_{i+1} = s_i + s_{i-1}$)

Buddy System

- All empty blocks of size s_i are linked in a list
- Array of headers to each list
- A block of size d required
- Choose the smallest permitted sized block
  - block of size s_i such that $s_i >= d$ but $s_{i-1} < d$

Exponential Buddy System

- **Each block of size s_{i+1} may be viewed as consisting of two blocks of size s_i**
- **A block of size 64 may be viewed as consisting of two buddies of size 32 each**

Exponential Buddy System

- A block of size 28 is requested
- Search for a block of size 32 (next 2^k size) and allot if available
- If not, search for block of size 64
- if available, split into two blocks of size 32
  - Creates two buddies
  - Allocate one, add the buddy to the free list for 32
- If not, look for a block of size 128
- ……

Exponential Buddy System

- A block of size 2^k is not available
- Search for a block of size 2^{k+1}
- if available, split into two blocks of size 2^k each
  - Creates two buddies
  - Allocate one, add the buddy to the free list for 2^k

Buddy System

- If no free block of size s_i exists
  - Find a block of size s_{i+1} and split into two
  - One of size s_i and the other of size s_{i+1} - s_i
  - Buddy system constraints that s_{i+1} - s_i must be some s_j for $j <= i$

Exponential Buddy System –returning blocks

- A block of size 32 is freed
- If its buddy is also free, combine together into a single block of size 64
- Can repeat with the resulting block of size 64, if its buddy is also free

Conclusion

- Heap Memory Management
  - Maintaining the free list
    - Appropriate data structures
  - Allocation, freeing, compaction
    - Algorithms
  - Time, Space trade-off

Reference

1. **A V Aho, J E Hopcroft, J D Ullman *Data Structures and Algorithms,* Pearson Education Asia, 2000**
2. **M L Scott *Programming Language Pragmatics*, Second Edition, Elsevier, 2006**
3. **A B Tucker, R E Noonan *Programming Languages-Principles and paradigms, Tata Mc GrawHill , 2007***

External Sorting

External Sorting

- Sort data residing in secondary memory
- Huge amount of data
  - Primary Memory of limited capacity
    - can not hold all the records simultaneously
- Merge Sort Algorithm

Secondary Storage

- Assume the list to be sorted resides on a disk
- Block – unit of data that is read from or written to a disk at one time

External Merge Sort

- Most popular method
- Two distinct phases

  Phase 1: Sorting small segments separately

  Phase 2: Merging the sorted segments

External Merge Sort

- Phase 1: Sorting small segments separately
  - Segments of the list are sorted using internal sorting
  - Sorted segments, known as runs are written to external storage
- Phase 2: Merging the runs
  - The entire runs need not be present in main memory
  - Keep only the leading records of the runs needed for merge

External Merge Sort - Example

- A list of 4500 records to be sorted
- Main memory can hold only 750 records at a time
- Divide into 6 segments
- Sort each segment separately using internal sorting
  - Generates 6 runs, each run written to disk

External Merge Sort - Example

4500 records to be sorted

Run 1 : 1-750

Run 2 : 751-1500

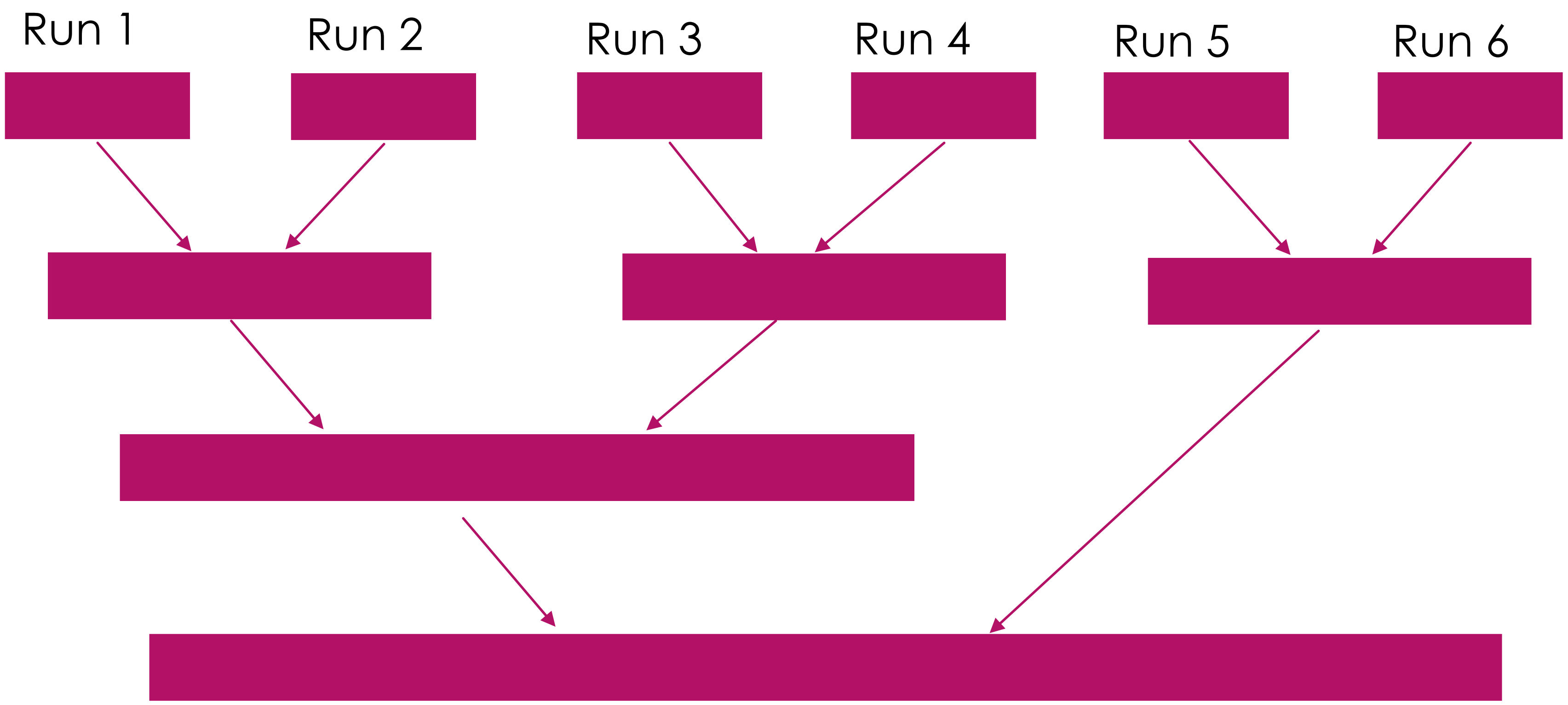
Run 3: 1501 - 2250

Run 4: 2251- 3000

Run 5: 3001 - 3750

Run 6: 3751-4500

Merging of runs



Size of a run?

- Based on block size, length of list, primary memory capacity
- Assuming block size is 250 records (enough to hold 250 records)
- Each run composed of 3 blocks

Merging the runs

- Merge two runs of size 750 records each
  - Primary memory can hold only maximum 750 records at a time ?
- Merging does not require all records together
- Only the leading records of the two runs brought to main memory

Merging the runs

- Use 3 blocks of internal memory
  - Each can hold 250 records
- 2 blocks as input buffers, third as output buffer
- One block from each run brought to input buffer
- Merged records written to output buffer

- When output buffer is full?
- When an input buffer is empty ?

Merging the runs

- One block from each run brought to input buffer (maximum 250 + 250)
- Merged records written to output buffer (maximum size 250)
- When output buffer is full ?
  - write to disk
- Whenever an input buffer is empty ?
  - refill with another block from the same run

External Sort – Time Complexity

- Time to access disk blocks
  - Seek Time
  - Latency Time
- Time to read/write one block from/to disk
- Time to internally sort segments
- Time to Merge

External Sort – Improvements

- Parallelising I/O and internal merging
    - Two disks – one for input and the other for output
    - Proper buffer handling

External Sort – Improvements

- 2-way merge with initial *m* runs
  - How many passes ?
- Higher order merge (k-way merge k>2)
  - Reduce the number of passes required for merging
  - More buffers required
  - How many passes for a k-way merge on *m* runs ?
  - Number of key comparisons in each step?

Reference

1. A V Aho, J E Hopcroft, J D Ullman *Data Structures and Algorithms,* Pearson Education Asia, 2000
2. E Horowitz, S Sahni, D Mehta *Fundamentals of Data Structures in C++*, Second Edition, Universities Press, 2008