

MANAGING MEMORY

OUTLINE

- Memory Organization
- Garbage Collection
 - Reference counting
 - Mark-and-sweep
 - Copy collection

MEMORY ORGANIZATION

- *Memory management* is the process of *binding* values to memory locations.
- A *process* is a program in execution.
- All the memory used by a process must reside in the process's *address space*.
- How the address space is organized depends on the operating system and the programming language being used.
- We are primarily concerned with imperative languages (such as C++/Java) in this lecture.
- Techniques developed here applies to all paradigms.

MAJOR AREAS OF MEMORY

○ *Static area:*

- Storage requirements known in advance and remain constant
- allocated at compile time (static or const)

○ *Run-time stack:*

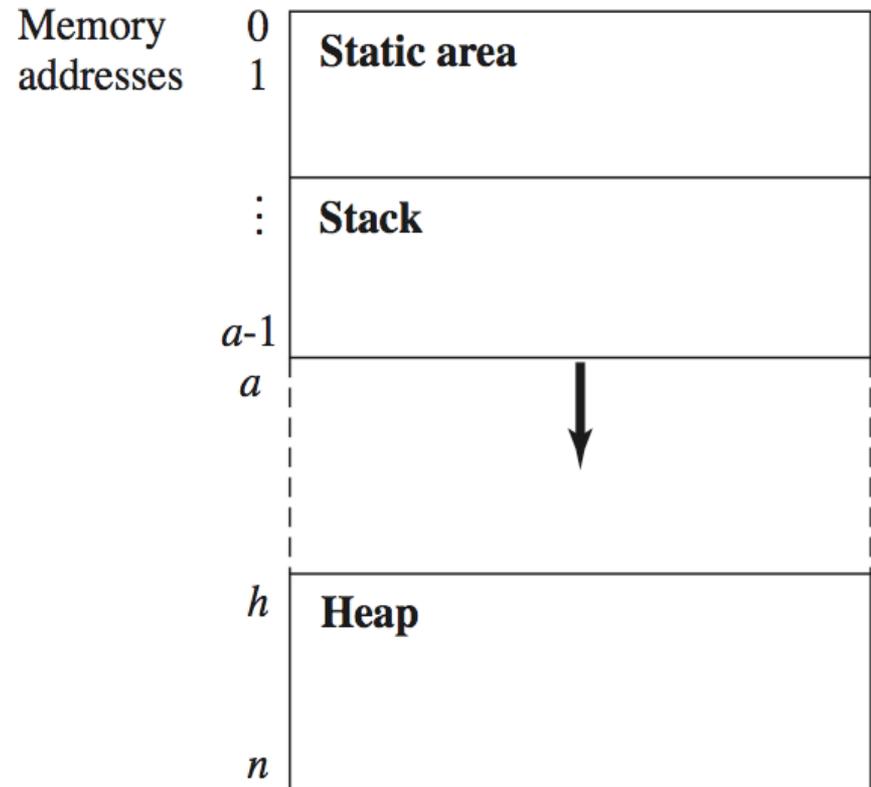
- local variables that get allocated each time a function is called (a.k.a. call stack)
- center of control for function call and return

○ *Heap:*

- dynamically allocated objects and data structures
- recall the memory store M in last lecture
- the least organized and most dynamic storage area
- Easily fragmented – needs *garbage collection*

STRUCTURE OF RUNTIME MEMORY

- $0 \leq a \leq h \leq n$
- Each memory word can be:
 - Unused
 - Undef
 - An elementary value



STATIC MEMORY

- Global variables that can be statically allocated get placed in the *static area*.
- Constants may also be placed in the static area depending on their type.
- The static area may be split into different parts for variables and for constants.
 - Data segment: static and global variables/constants
 - text segment: executable instructions
- Values that can be statically bound (e.g. at compile time) can be placed here.
 - String literals: “hello world!”

RUNTIME STACK

- The stack is a contiguous region of memory that grows and shrinks as a process runs.
- It is used to hold *local environments (closures)* or *activation records* for functions and procedures. These are also called *stack frames*.
- When a function is called (activated), storage for its local variables, the calling parameters, and return linkage is allocated by growing the stack.
- When control is returned from the function, the stack frame is de-allocated and the stack shrinks.
- A function's stack frame exists as long as the function is active.

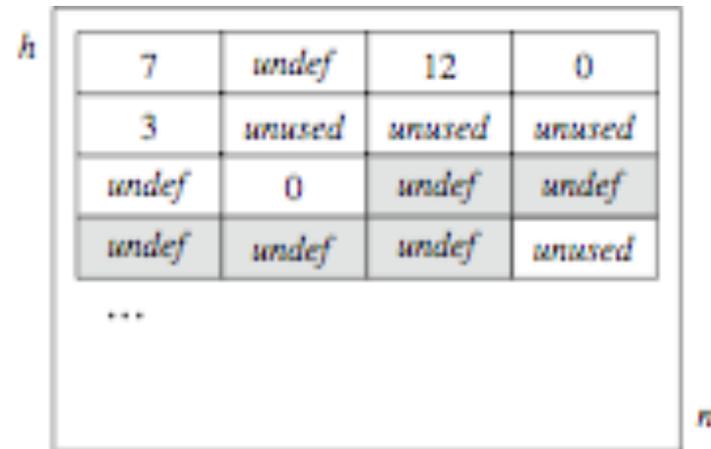
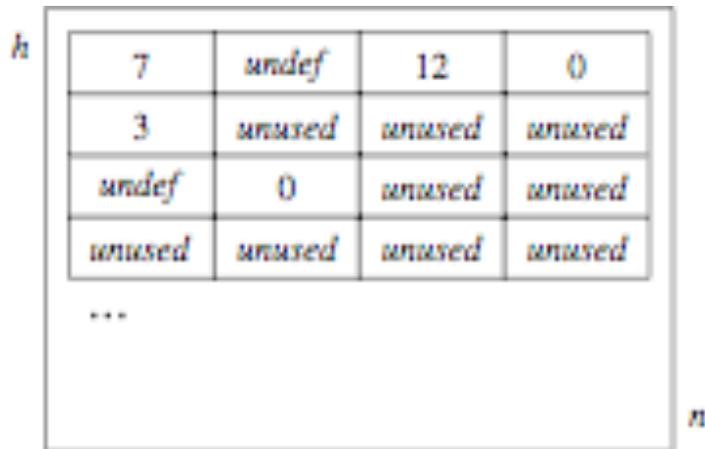
HEAP

- Variable storage that is dynamically allocated at runtime is placed in the heap.
- The heap is managed by dividing it into blocks.
 - In many real implementations, a tree structure (binary heap).
- As a process runs space is allocated to new variables from heap space (malloc, new).
- When a variable's lifetime expires its space may be returned to the heap (*deallocated*). This can leave holes in the heap causing *fragmentation*.
- Some languages leave managing the heap in the hands of the programmer (C, C++, etc.).
- Others do *heap management* (Java, Python, etc.).

ALLOCATING HEAP BLOCKS

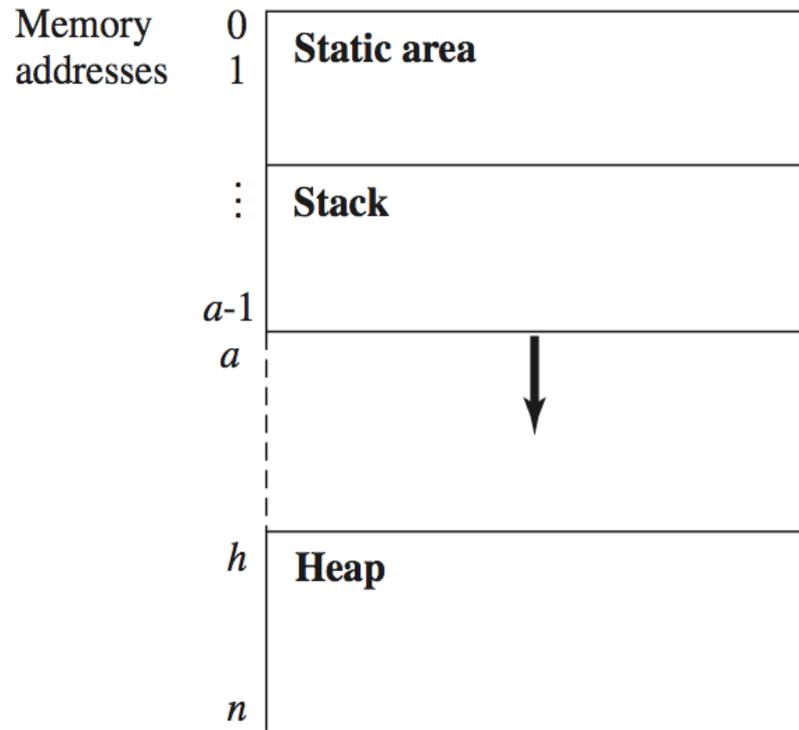
- The function *new* allocates a *contiguous block* of heap space to the program.

E.g., `new(5)` returns the address of the next block of 5 words available in the heap:



STACK AND HEAP OVERFLOW

- **Stack overflow** occurs when the top of stack, a , would exceed its (fixed) limit, h .
 - Stack can also go *underflow*.
- **Heap overflow** occurs when a call to *new* occurs and the heap does not have a large enough block available to satisfy the call.



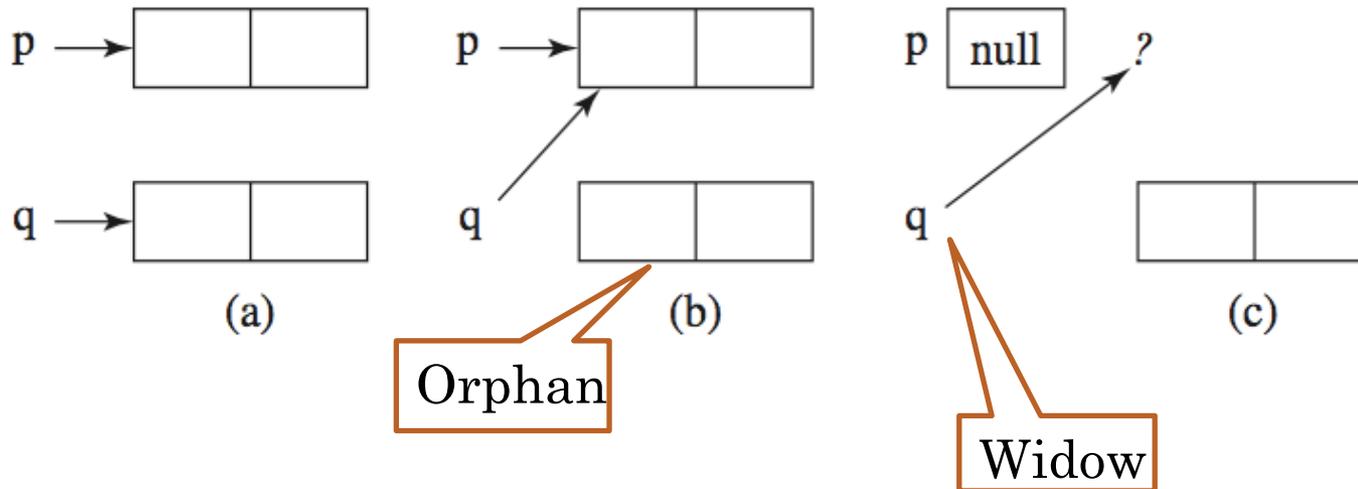
GARBAGE COLLECTION

- *Garbage* is a block of heap memory that cannot be accessed by the program.
- Garbage can occur when either:
 1. An allocated block of heap memory has no reference to it (an “orphan”), or
 2. A reference exists to a block of memory that is no longer allocated (a “widow”).

GARBAGE EXAMPLE

```
class node {  
    int value;  
    node next;  
}  
node p, q;
```

```
p = new node();  
q = new node();  
q = p;  
delete p;
```



WHY GARBAGE COLLECTION?

- Today's programs consume storage freely
 - 8GB laptops, 16-32 GB desktops, 512GB servers
 - 64-bit address spaces (x64, SPARC, Itanium, Opteron)
- ... and mismanage it
 - Memory leaks, dangling references, double free, misaligned addresses, null pointer dereference, heap fragmentation
 - Poor use of reference locality, resulting in high cache miss rates and/or excessive demand paging
- Explicit memory management breaks high-level programming abstraction



GC AND PROGRAMMING LANGUAGES

- GC is not a language feature
- GC is a pragmatic concern for automatic and efficient heap management
 - Cooperative langs: Lisp, Scheme, Prolog, Smalltalk ...
 - Uncooperative languages: C and C++
 - But garbage collection libraries have been built for C/C++
- Recent languages have GC built-in:
 - Object-oriented languages: Modula-3, Java, C#, Python
 - In Java, runs as a low-priority thread; `System.gc` may be called by the program
 - Functional languages: ML and Haskell



THE PERFECT GARBAGE COLLECTOR

- No visible impact on program execution
- Works with any program and its data structures
 - For example, handles cyclic data structures
- Collects garbage (and only garbage) cells quickly
 - Incremental; can meet real-time constraints
- Has excellent spatial locality of reference
 - No excessive paging, no negative cache effects
- Manages the heap efficiently
 - Always satisfies an allocation request and does not fragment

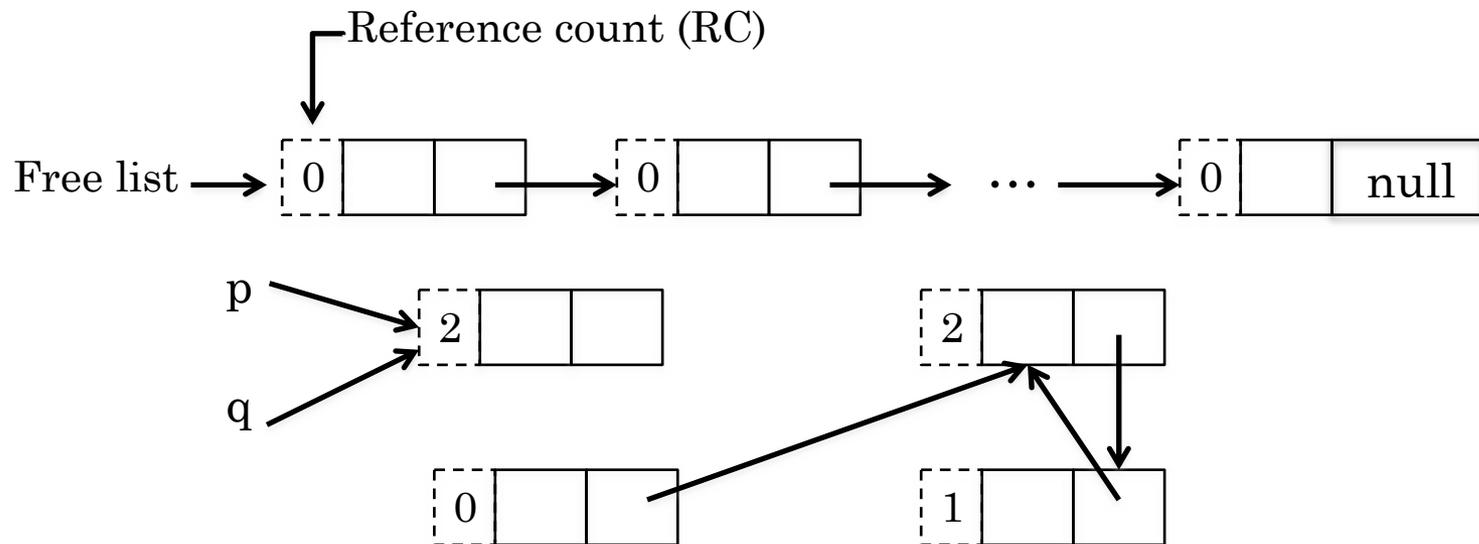


GARBAGE COLLECTION ALGORITHMS

- *Garbage collection* is any strategy that reclaims unused heap blocks for later use by the program.
- Three classical garbage collection strategies:
 - Reference Counting
 - occurs whenever a heap block is allocated, but doesn't detect all garbage.
 - Mark-and-Sweep
 - Occurs only on heap overflow, detects all garbage, but makes two passes on the heap.
 - Copy Collection
 - Faster than mark-sweep, but reduces the size of the heap space.

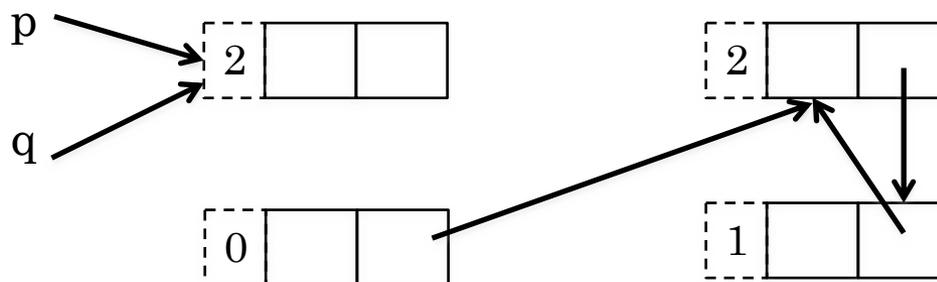
REFERENCE COUNTING

- The heap is a chain of nodes (the *free_list*).
- Each node has a reference count (RC).
- For an assignment, like $q = p$, garbage can occur:



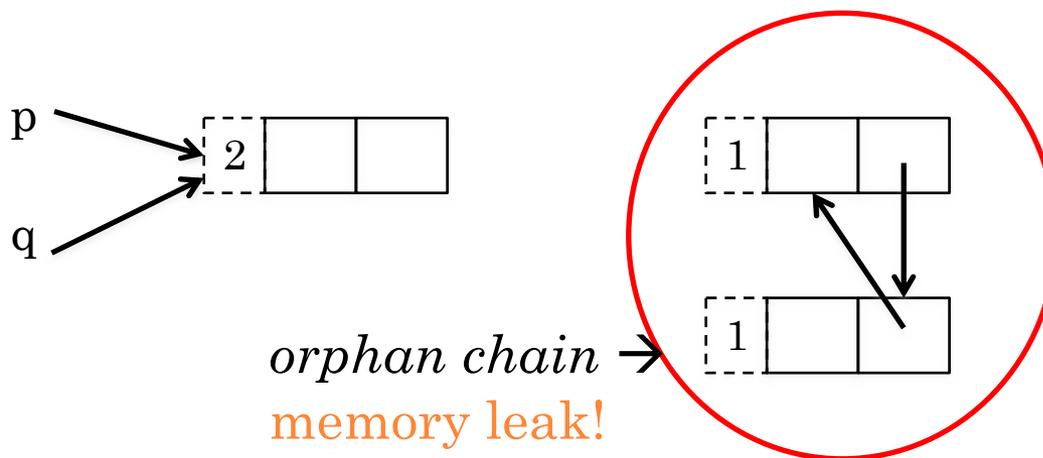
BUT NOT ALL GARBAGE IS COLLECTED...

- Since q's node has $RC = 0$, the RC for each of its children is reduced by 1, it is returned to the free list, and this process repeats for its descendants, leaving:



BUT NOT ALL GARBAGE IS COLLECTED...

- Since q's node has $RC = 0$, the RC for each of its children is reduced by 1; it is returned to the free list, and this process repeats for its descendants, leaving:



ADVANTAGES OF REFERENCE COUNTING

- Occurs dynamically, overhead of garbage collection is spread over time
- Relatively easy to implement
- Can coexist with manual memory management
- Spatial locality of reference is good
 - Access pattern to virtual memory pages no worse than the program, so no excessive paging
 - No long jumps.
- Can re-use freed cells immediately
 - If $RC == 0$, put back onto the free list

DISADVANTAGES OF REFERENCE COUNTING

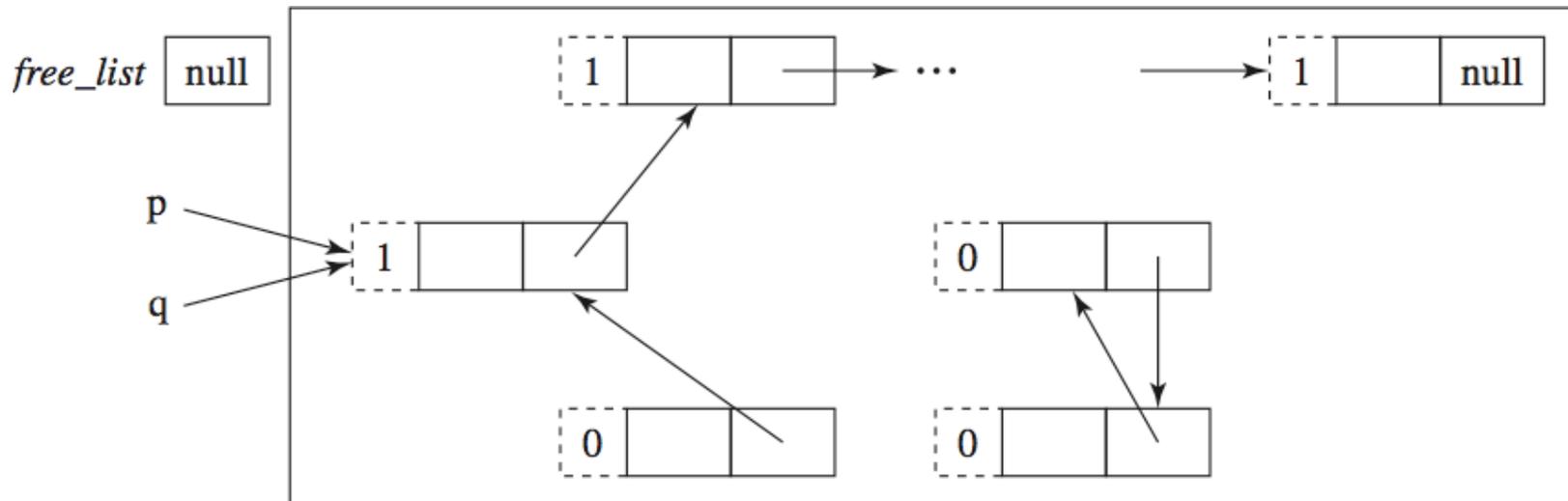
- Failure to detect inaccessible circular structure and hence the GC is incomplete
- Space overhead by appending an integer number to every node in the heap
- Performance overhead created by the book-keeping done during pointer assignment or when a heap block is allocated/de-allocated:
 - Check to ensure that it is not a self-reference
 - Decrement the count on the old cell, possibly deleting it
 - Update the pointer with the address of the new cell
 - Increment the count on the new cell

MARK-AND-SWEEP

- Each node in the *free_list* has a mark bit (MB) initially 0.
- Called only when heap overflow occurs:
 - Pass I: Mark all nodes that are (directly or indirectly) accessible from the stack by setting their MB=1.
 - Pass II: Sweep through the entire heap and return all unmarked (MB=0) nodes to the free list.
- *Note: all orphans are detected and returned to the free list.*

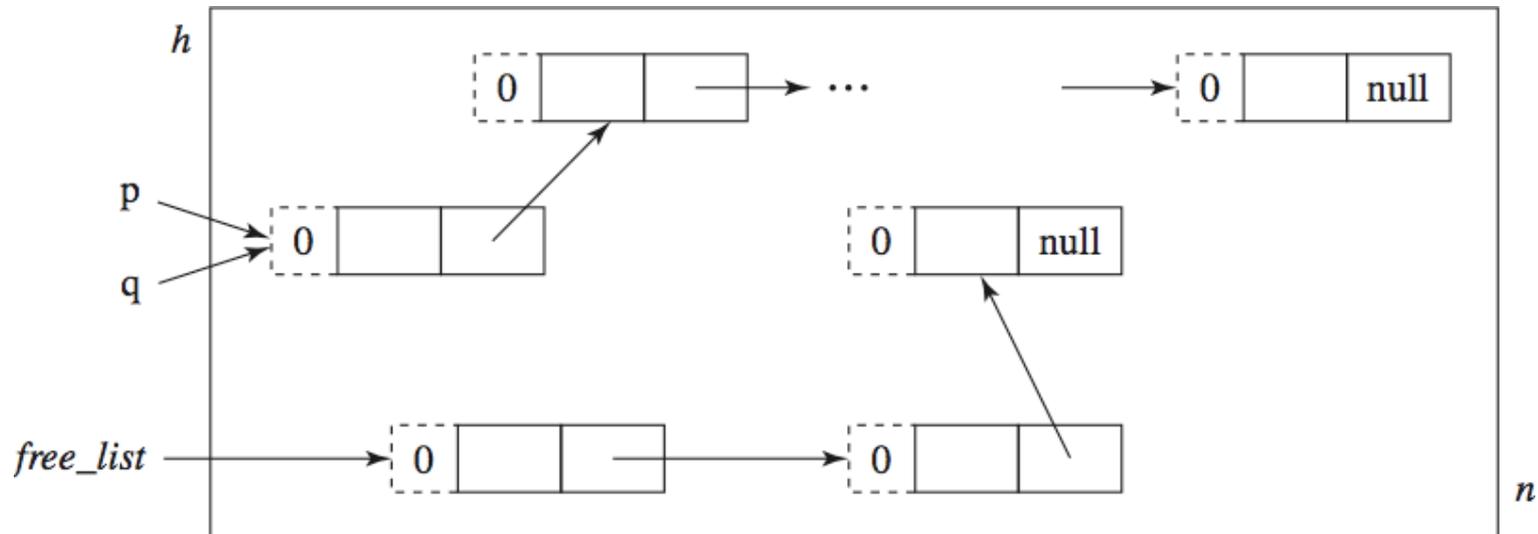
HEAP AFTER PASS I OF MARK-AND-SWEEP

- Triggered by `q=new node()` and `free_list = null`.
- All accessible nodes are marked 1.



HEAP AFTER PASS II OF MARK-AND-SWEEP

- Now *free_list* is restored and
- the assignment $q = \text{new node}()$ can proceed.



PROS AND CONS OF MARK-AND-SWEEP

○ Pros:

- handles cycles correctly
- very little space overhead
 - 1 bit used for marking cells may limit max values that can be stored in a cell (e.g., for integer cells)

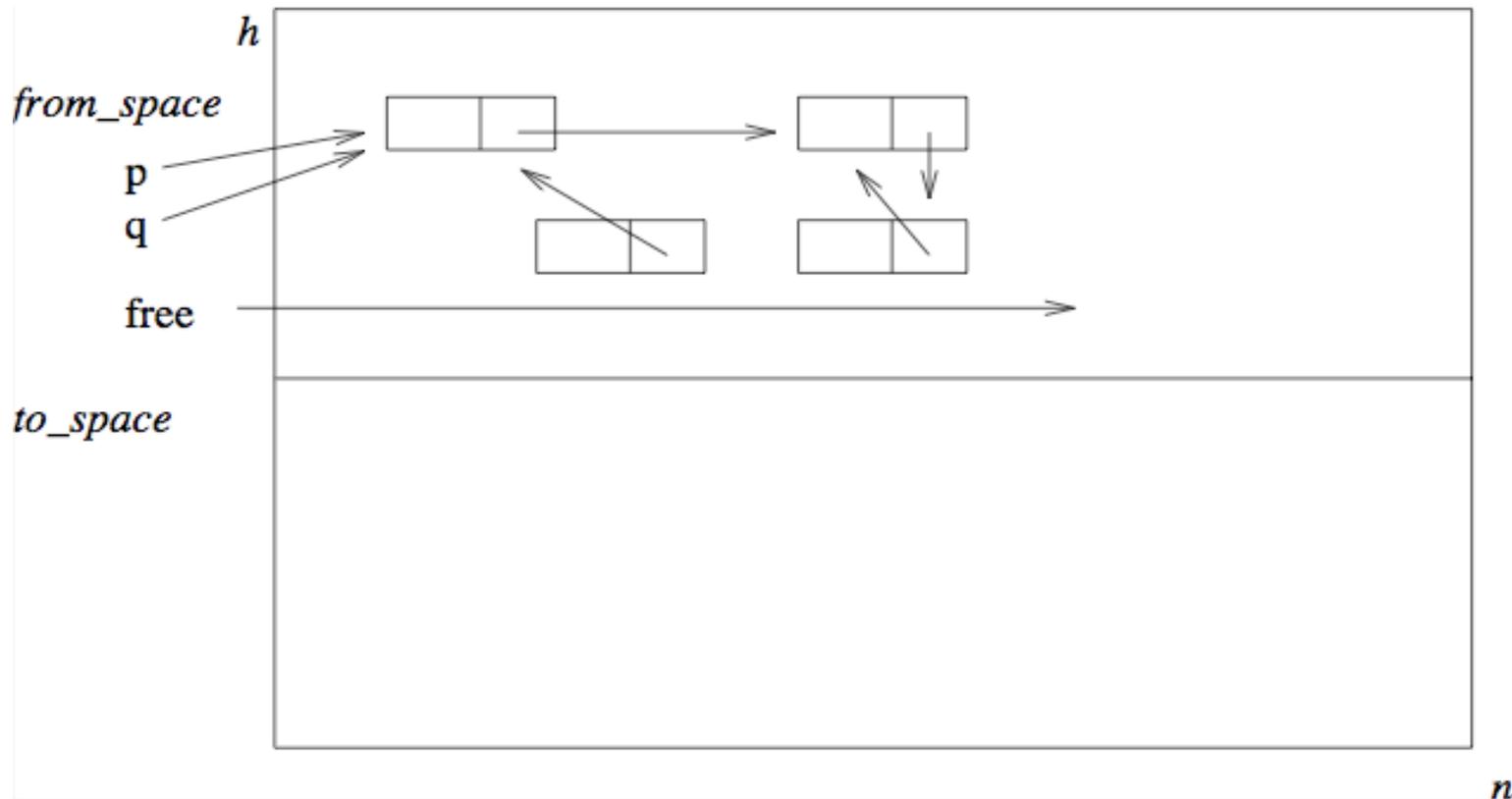
○ Cons:

- normal execution must be suspended (noticeable pause)
- may touch all virtual memory pages
 - May lead to excessive paging if the working-set size is small and the heap is not all in physical memory
- heap may fragment
 - Cache misses, page thrashing; more complex allocation



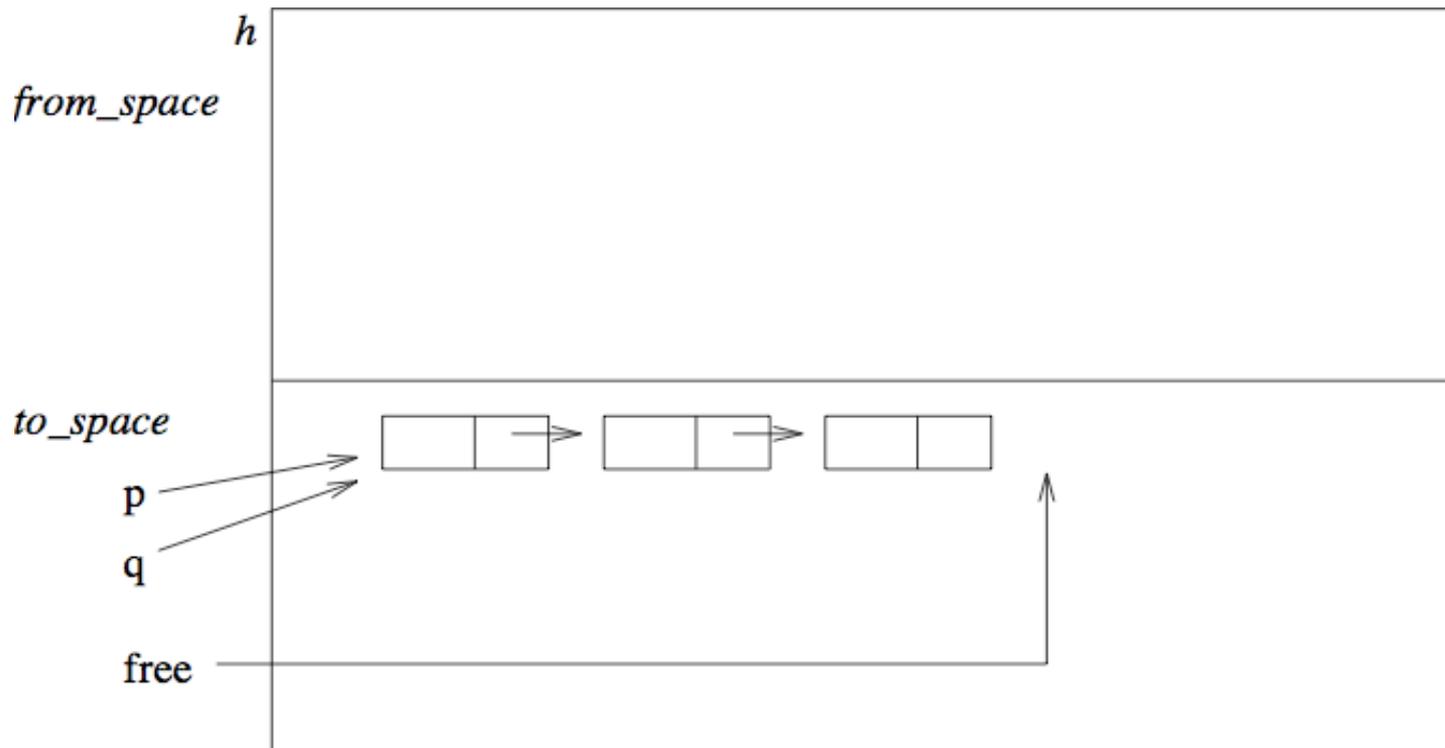
COPY COLLECTION

- Heap partitioned into two halves; only one is active.
- Triggered by `q=new node()` and *free_list* outside the active half:

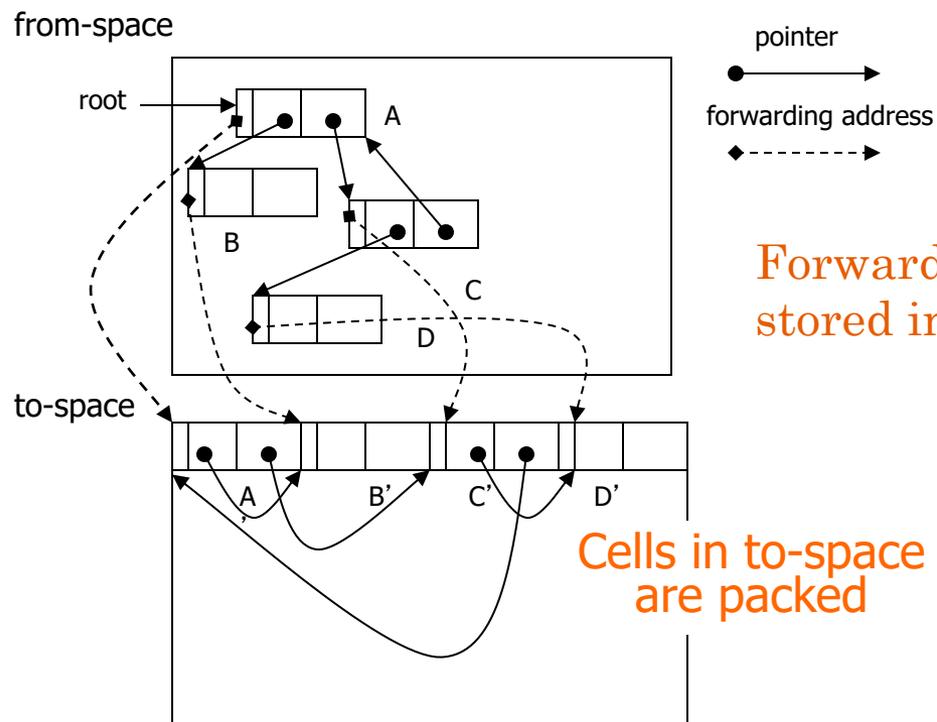


ACCESSIBLE NODES COPIED TO OTHER HALF

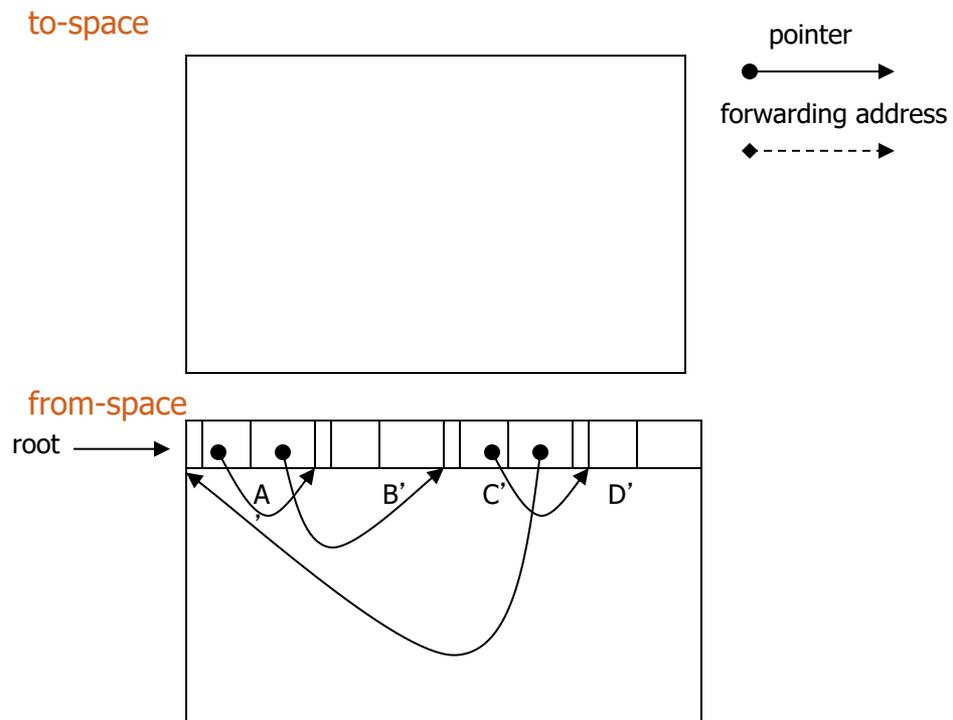
- Note: The accessible nodes are packed, orphans are returned to the free_list, and the two halves reverse roles.



CHENEY'S ALGORITHM



CHENEY'S ALGORITHM



PROS AND CONS OF COPY COLLECTION

○ Pros:

- very low cell allocation overhead
 - Out-of-space check requires just an addr comparison
 - Can efficiently allocate variable-sized cells
- compacting
 - Eliminates fragmentation, good locality of reference

○ Cons:

- Twice the memory footprint
 - Probably Ok for 64-bit architectures (except for paging)
 - When copying, pages of both spaces need to be swapped in. For programs with large memory footprints, this could lead to lots of page faults for very little garbage collected
 - Large physical memory helps

GARBAGE COLLECTION SUMMARY

- Modern algorithms are more elaborate.
 - Most are hybrids/refinements of the above three.
 - E.g., generational garbage collection
 - Nodes that die, die young
 - Divide the heap into generations, and GC younger generations more often
 - Doesn't reclaim all free space – may need mark & sweep or copy collection occasionally
 - Java/.NET: GC a few recent generations only
- In Java, garbage collection is built-in.
 - runs as a low-priority thread.
 - Also, `System.gc` may be called by the program.
- Functional languages have garbage collection built-in.
- C/C++ default garbage collection to the programmer.