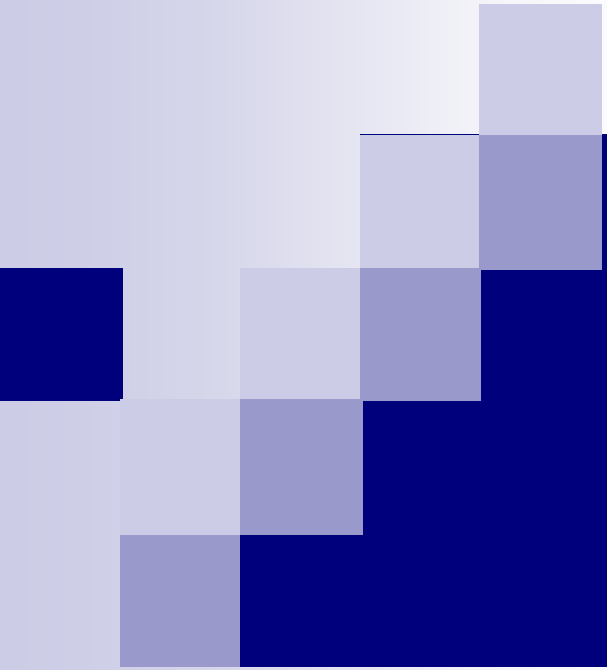


Advanced algorithms

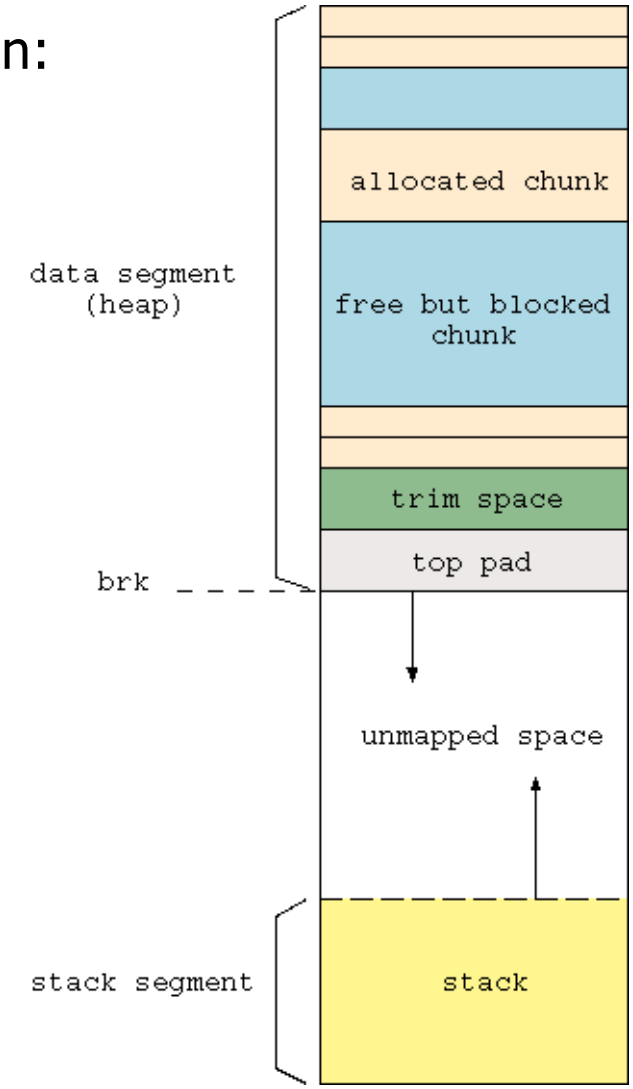
dynamic data structures,
garbage collectors

Jiří Vyskočil, Radek Mařík

2011

Dynamic Data Structures

Program data memory representation:



Dynamic Data Structures

- The main goals of a memory allocator:

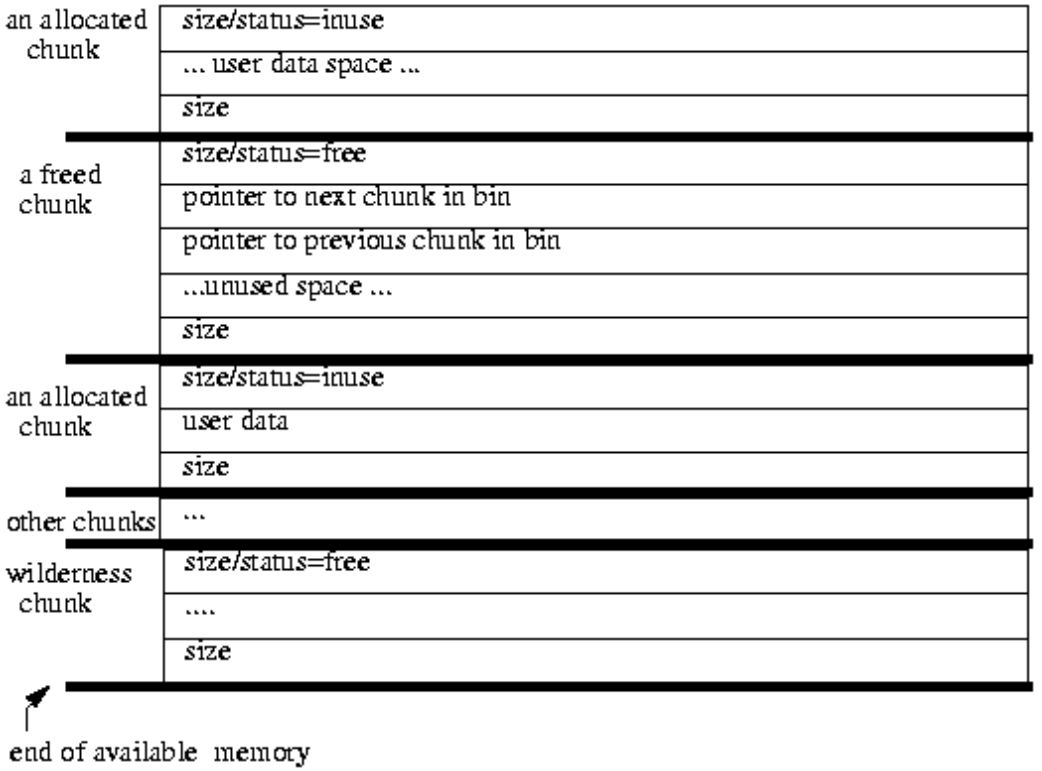
  - **minimizing time**
    - The `malloc`, `free` and `realloc` routines should be as fast as possible in the average case.

  - **minimizing space**
    - The allocator should not waste space: It should obtain as little memory from the system as possible, and should maintain memory in ways that minimize *fragmentation* -- ``holes'' in contiguous chunks of memory that are not used by the program.

  - **maximizing locality**
    - Allocating chunks of memory that are typically used together near each other. This helps minimize page and cache misses during program execution.

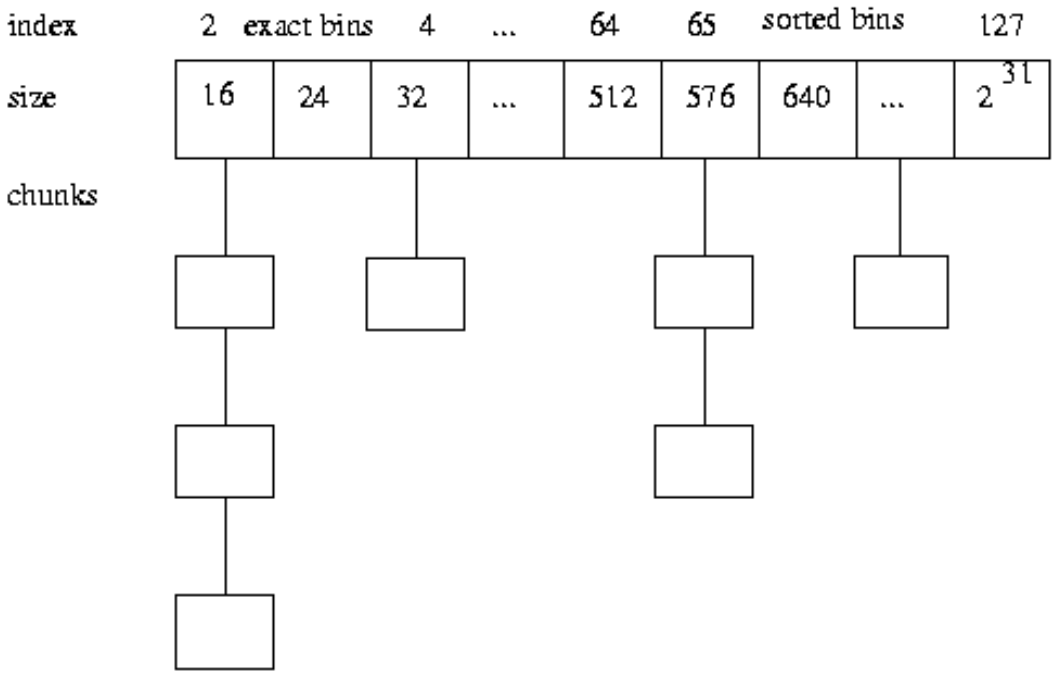
Dynamic Data Structures

- **dynamic data structures**
  - Two bordering unused chunks can be coalesced into one larger chunk. This minimizes the number of unusable small chunks.
  - All chunks can be traversed starting from any known chunk in either a forward or backward direction.
  - Free chunks are connected (see further description).



Dynamic Data Structures

- Available chunks are maintained in bins, grouped by size.
- There are some number (here 128) of fixed-width bins, approximately logarithmically spaced in size.
- Bins of sizes less than some constant size (here 512 bytes) hold each only exactly one size (spaced 8 bytes apart, simplifying enforcement of 8-byte alignment).
- Searches for available chunks are processed in smallest-first, *best-fit* order.



Dynamic Data Structures

- deallocation algorithm:
  1. Look to the previous and the next chunk from the current chunk for the deallocation.
  2. If there is at least one of this chunk free then connect all of these free chunks to one free chunk (Of course you need to disconnect these free chunks from their correspondent groups). This operation tends to have a relatively low fragmentation.
  3. The result chunk is marked as free and connected to the right group.

- Because both size information and bin links must be held in each available chunk, the smallest allocable chunk is 16 bytes in systems with 32-bit pointers and 24 bytes in systems with 64-bit pointers.
- This can lead to significant wastage for example in applications allocating many tiny linked-list nodes.

Dynamic Data Structures - Locality

- If locality were the *only* goal, an allocator might always allocate each successive chunk as close to the previous one as possible. However, this *nearest-fit* (often approximated by *next-fit*) strategy can lead to a very bad fragmentation.

- In general, the problem of fragmentation can be solved by allocation via an operating system with paging. Typically, this is too slow and the size of the allocation is rounded by the size of the page which increases vastly the memory requirements.
- These techniques are used mainly in debugging (e.g. library Electric Fence). It helps you to detect two common programming bugs:
  - software that overruns the boundaries of a `malloc()` memory allocation, and
  - software that touches a memory allocation that has been released by `free()`.
- Such a system allocates at least two pages for every allocated chunk and mark one of this page as inaccessible. When software reads or writes this inaccessible page, the hardware issues a segmentation fault, stopping the program at the offending instruction. It is then trivial to find the erroneous statement using a standard debugger. In a similar manner, memory that has been released by `free()` is made inaccessible, and any code that touches it will get a segmentation fault.

Garbage Collector

- **garbage collector**
  - garbage collection is a form of automatic memory management. The *garbage collector* (GC) attempts to reclaim garbage (= memory occupied by objects that are no longer in use) by the program.

- **the basic principles are:**

1. Find data objects in the program that cannot be accessed in the future.

2. Reclaim the resources used by those objects.

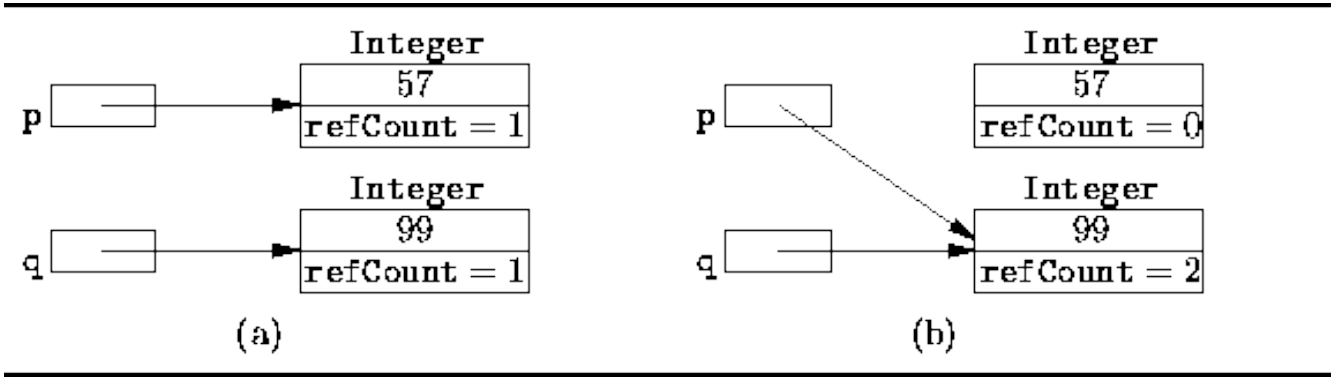
Garbage Collector

- **benefits**
  - Frees the programmer from manually dealing with memory deallocation.
  - certain categories of bugs are eliminated or reduced:
    - **Dangling pointer bugs**, which occur when a piece of memory is freed while there are still pointers to it, and one of those pointers is then used. **=> unpredictable results**
    - **Double free bugs**, which occur when the program tries to free a region of memory that has already been freed, and perhaps already been allocated again. **=> sometimes undefined results**
    - Certain kinds of **memory leaks**, in which a program fails to free memory occupied by objects that will not be used again. **=> memory exhaustion**
- **disadvantages**
  - **Additional computing resources.**
    - GC consumes computing resources in deciding which memory to free, reconstructing facts that may have been known to the programmer.
    - GC needs some additional information about the places of dynamic variables and for its own run.
  - **Unpredictable delays.** The moment when the garbage is actually collected can be unpredictable, resulting in delays throughout a session. This is unacceptable in real-time environments such as device drivers, interactive programs,...
  - A semantic GC is so called **undecidable problem** (same as the halting problem) (the present GCs are based only on a syntactic level).

Garbage Collector – Reference Counting

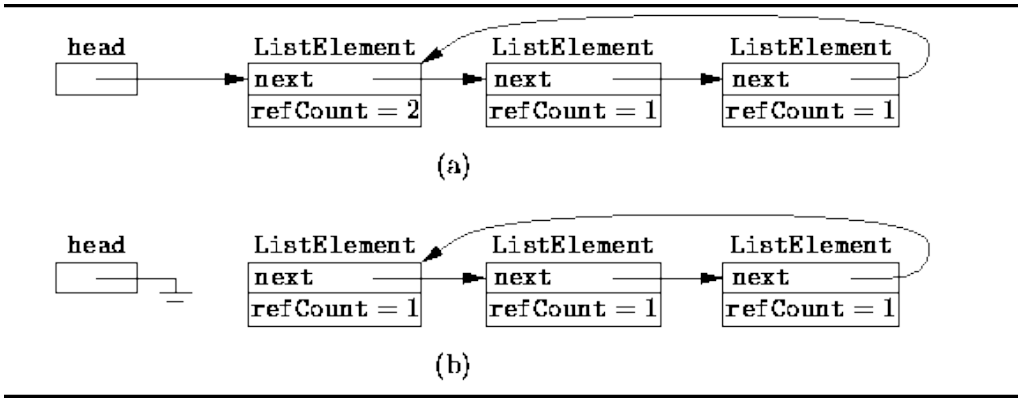
- **reference counting**
  - Each object has a count of the number of references to it.
  - An object's reference count is incremented when a reference to it is created, and
  - decremented when a reference is destroyed.
  - The object's memory is reclaimed when the count reaches zero.

Reference counts before and after the assignment `p=q`:



Garbage Collector – Reference Counting

- **major disadvantages**
  - If two or more objects refer to each other, they can create a cycle whereby neither will be collected as their mutual references never let their reference counts become zero.



  - In naive implementations, each assignment of a reference and each reference falling out of scope often require modifications of one or more reference counters.

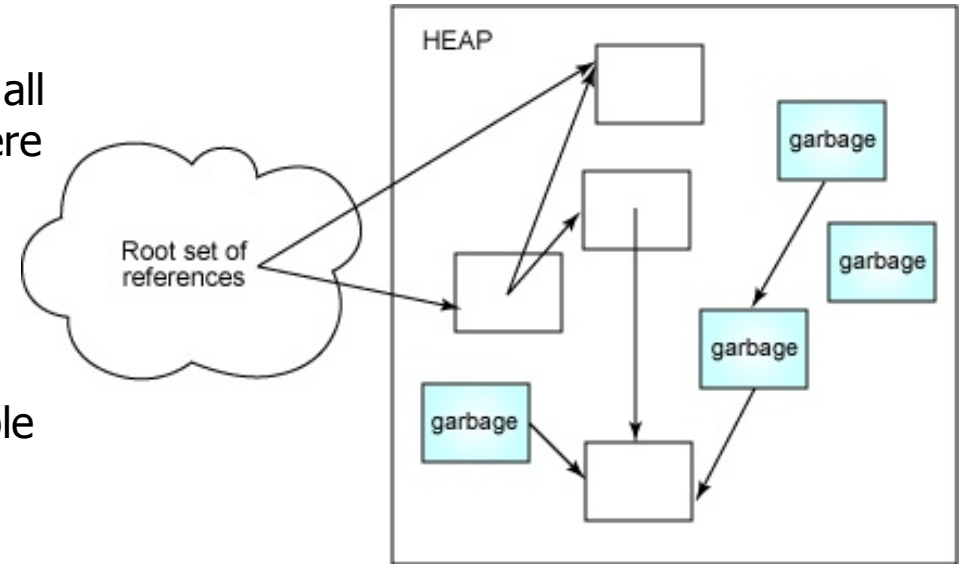
- One **important advantage** of reference counting is that it provides deterministic garbage collection (unlike tracing GC).
- Therefore, reference counting by itself is not a suitable garbage collection scheme for arbitrary objects. But it is often used in more specific situations e.g. in modern Pascals for strings manipulations.

Garbage Collector - Tracing GCs

- *Tracing* GCs are the most common type of GC. They first determine which objects are *reachable* (or potentially reachable), and then discard all remaining objects.

- **syntactic reachability of an object**
  - Informally, an object (or cell) is reachable (or live) if it is referenced by at least one variable in the program, either directly or through references from other reachable objects. More precisely, objects can be reachable in only two ways:

1. A distinguished set of objects are assumed to be reachable: these are known as the *roots (or root set of references)*. Typically, these include all the objects referenced from anywhere in the call stack (that is, all local variables and parameters in the functions currently being invoked), and any global variables.
2. Anything referenced from a reachable object is itself reachable; more formally, reachability is a *transitive closure*.



Garbage Collector - Naïve Mark & Sweep

- **naïve mark & sweep**
  - Its so called *tracing collector*. Tracing collectors are so called because they trace through the working set of memory. These garbage collectors perform collection in cycles.
  - each object in memory has a flag (typically a single bit) reserved for garbage collection use only.
  - This flag is always *cleared*, except during the collection cycle.
- **algorithm**
  1. The first stage of collection does a tree traversal of the entire 'root set', marking each object that is pointed to as being 'in-use‘ flag.
  2. All objects that those objects point to, and so on, are marked ‘in-use’ as well, so that every object that is ultimately pointed to from the root set is marked ‘in-use’.
  3. Finally, all memory is scanned from start to finish, examining all free or used blocks; those with the ‘in-use’ flag still cleared are not reachable by any program or data, and their memory is freed. (For objects which are marked ‘in-use’, the ‘in-use’ flag is cleared again, preparing for the next cycle.)

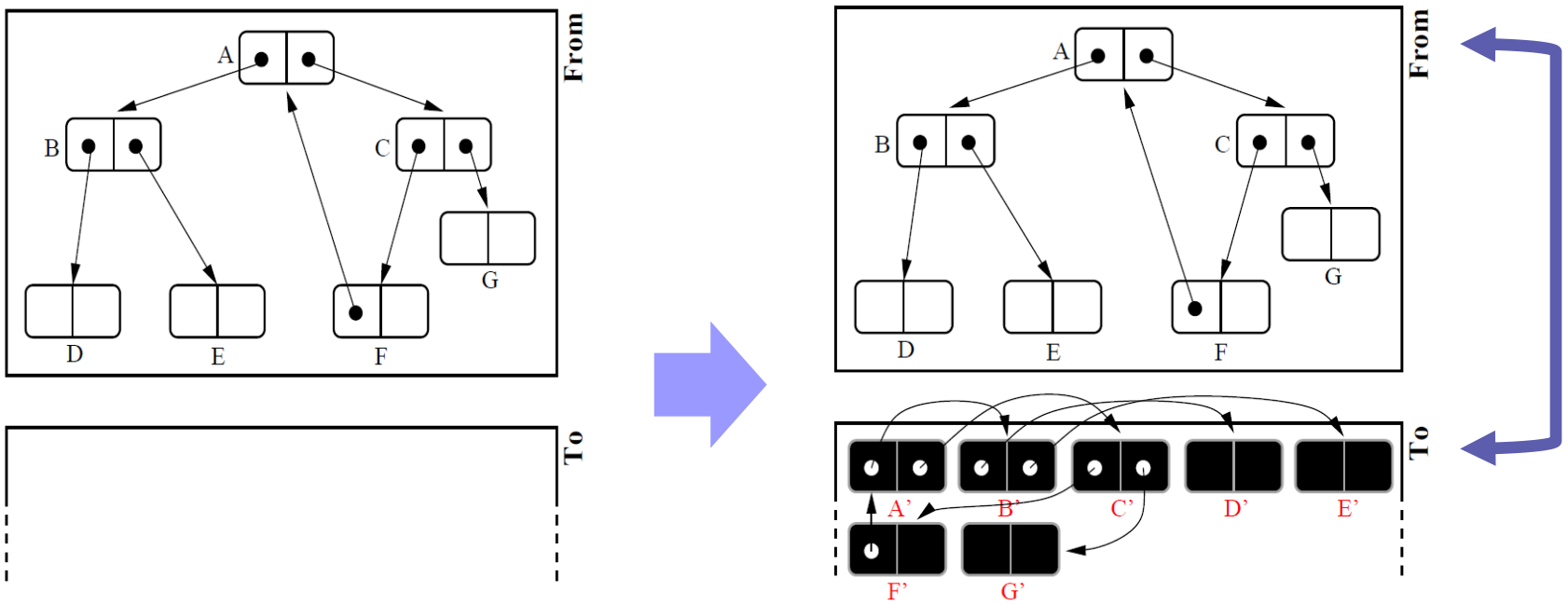
Garbage Collector - Naïve Mark & Sweep

- **benefits**
  - it can be technically implemented in constant memory
    - 1 bit used for marking cells and during traversing structures, directions of pointers can be swapped for storing the backtracking path.
  - it handles cycles correctly
- **disadvantages**
  - the entire system must be suspended during collection
  - no mutation of the working set can be allowed
  - This will cause programs to 'freeze' periodically (and generally unpredictably), making real-time and time-critical applications impossible.

Garbage Collector - Semi-Space Collector

- **semi-space or copying collector (Cheney's algorithm)**
  - the heap is divided into two equal halves, only one of which is in use at any one time.
  - Garbage collection is performed by copying live objects from one semispace (the **from-space**) to the other (the **to-space**), which then becomes the new heap. The entire old heap is then discarded in one piece.
- **algorithm**
  - **Object references on the stack.** Object references on the stack are checked. One of the two following actions is taken for each object reference that points to an object in from-space:
    - If the object has not yet been moved to the to-space, this is done by creating an identical copy in the to-space, and then replacing the from-space version with a forwarding pointer to the to-space copy. Then update the object reference to refer to the new version in to-space.
    - If the object has already been moved to the to-space, simply update the reference from the forwarding pointer in from-space.
  - **Objects in the to-space.** The garbage collector examines all object references *in the objects that have been migrated to the to-space*, and performs one of the above two actions on the referenced objects.
  - Once all to-space references have been examined and updated, garbage collection is complete.

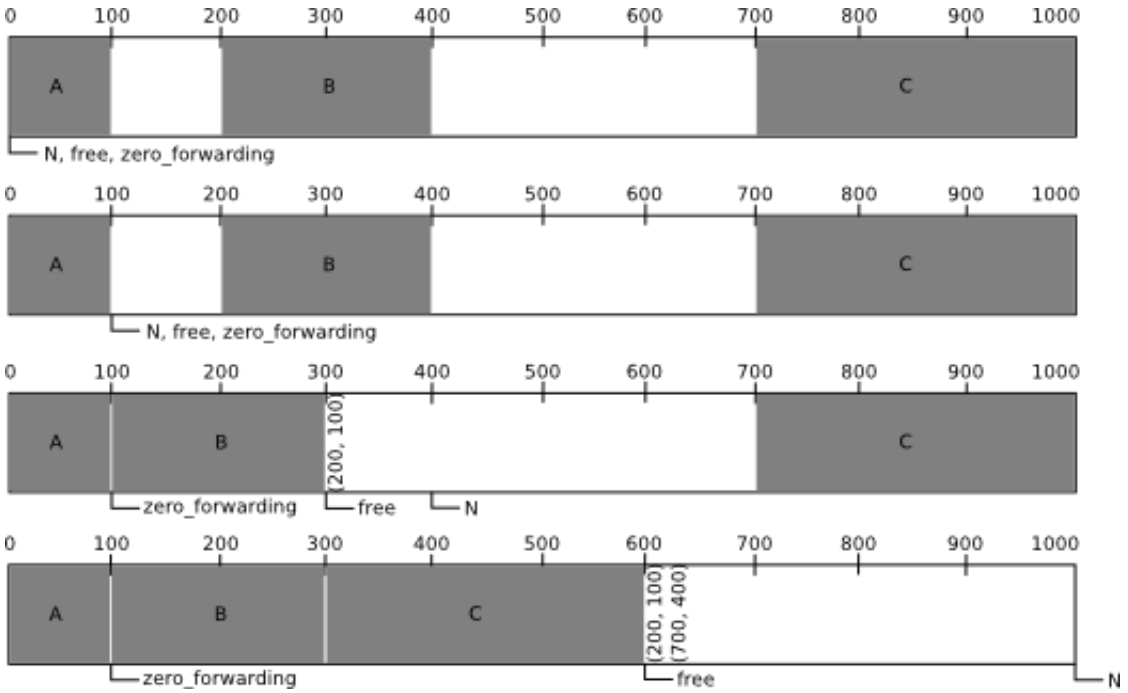
Garbage Collector - Semi-Space Collector



- **benefits**
  - does a complete defragmentation of memory (on contrary to Mark & Sweep)
  - allocation can be done in constant time
- **disadvantages**
  - similar as mark & sweep.
  - very large contiguous region of free memory is necessarily required on every collection cycle.

Garbage Collector – Mark & Compact

- **algorithm Mark & Compact**
  - Mark & compact algorithms can be regarded as a combination of the mark-sweep algorithm and Cheney's copying algorithm.
    - does a defragmentation of memory (on contrary to Mark & Sweep)
    - Has lower memory consumptions (on contrary to Cheney's algorithm)
    - Does not copy big "long living" objects as frequently as Cheney's algorithm.

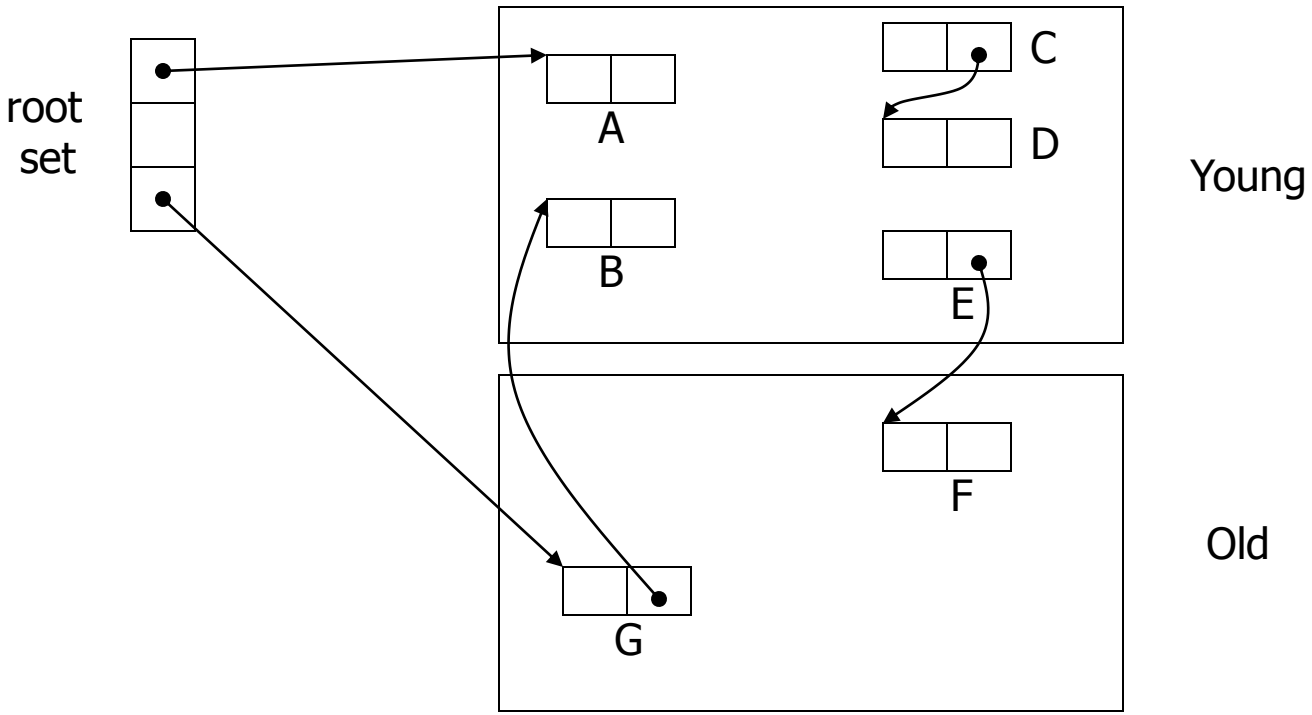


Garbage Collector - Generational

- **generational GC (ephemeral GC)**
  - Generational garbage collection is a heuristic approach.
  - It has been empirically observed that:
    - *generational hypothesis .* In many programs, the most recently created objects are also those most likely to become unreachable quickly.
    - only a very small part of references from „old“ objects is pointing to younger objects.
  - A generational GC divides objects into generations and, on most cycles, will place only the objects of a subset of generations into the initial white (condemned) set.
  - Furthermore, the runtime system maintains knowledge of when references cross generations by observing the creation and overwriting of references.
  - When the garbage collector runs, it may be able to use this knowledge to prove that some objects in the initial white set are unreachable without having to traverse the entire reference tree.
  - If the generational hypothesis holds, this results in much faster collection cycles while still reclaiming most unreachable objects.
  - generational garbage collectors use separate memory regions for different ages of objects.
  - When a region becomes full, those few objects that are referenced from older memory regions are promoted (copied) up to the next highest region, and the entire region can then be overwritten with fresh objects.

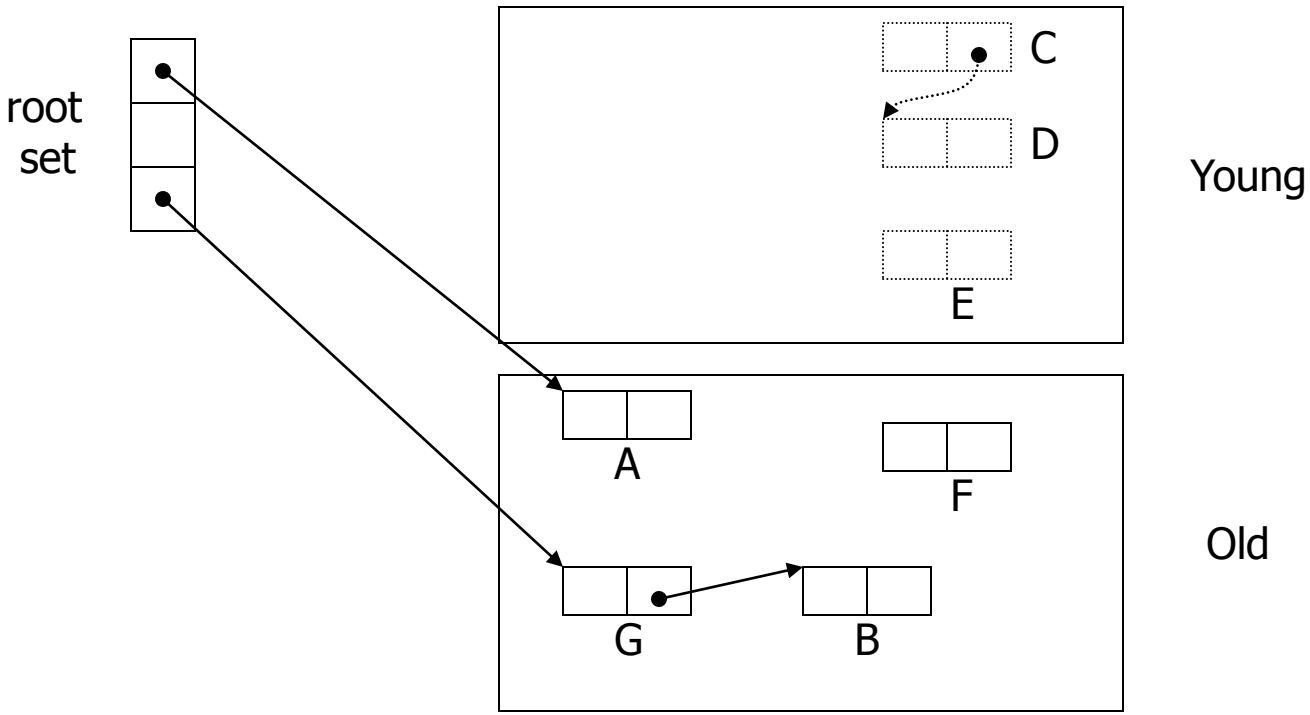
Garbage Collector - Generational

- **generational GC, example of "aging" 1/2:**



Garbage Collector - Generational

- **generational GC, example of "aging" 2/2:**



References

- Lee, D. *A Memory Allocator.* http://g.oswego.edu/dl/html/malloc.html
- Preiss, B. R. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java.* Wiley 1999. ISBN-13: 9780471346135.
- Cheney, C. J. *A Nonrecursive List Compacting Algorithm*. Communications of the ACM 13 (11): 677–678. (November 1970).
- Wikipedia: http://en.wikipedia.org/wiki/Garbage_collection_(computer_science)