10. Garbage Collection

- In contrast to explicit deallocation, garbage collection avoids dangling references.

- Some garbage collection algorithms must not be interrupted (e.g. Deutsch-Schorr-Waite). Thus response times for interrupts cannot be predicted.

- More involved <u>incremental</u> algorithms allow the garbage collection to proceed concurrently with the main program.

- Both explicit and implicit deallocation lead to the problem of <u>fragmentation</u> of the heap. Various strategies for minimizing fragmentation and for <u>compaction</u> of fragmented memory exists.

Mark-Sweep Garbage Collection

- Garbage collection works in two phases. We assume that each object on the heap has one spare bit for marking and unmarking it:
  - Initially all objects on the heap are unmarked.
  - For all declared pointer variables a depth-first search is made. All encountered objects are marked and all pointer fields are visited recursively. The recursion stops if a pointer field is nil or the object is already marked.
  - When this is finished, all unmarked objects on the heap are those which are not reachable. The memory they occupy can be reclaimed. Simultaneously, the marked object are unmarked.

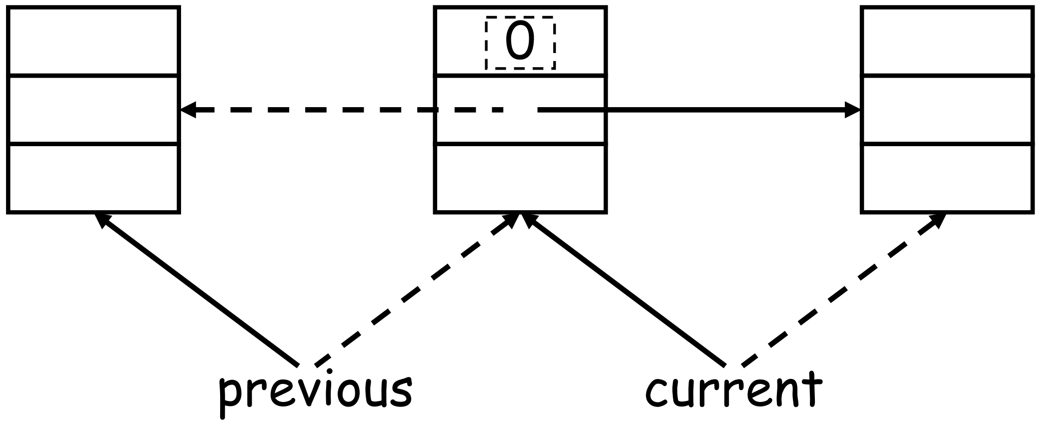
- The drawback of this algorithm is that for each recursive call, space on the stack for the activation frame is needed. In case we are reclaiming the memory of a singly linked list, that may need more memory space than the list itself!

In-Place Garbage Collection (Deutsch-Schorr-Waite) …

- A non-recursive depth-first search modifies the pointer fields while traversing such that from the current object a link to its parent is kept within the object. The algorithm uses two pointers, current and previous, in order to set the backward links. Assume that each object contains two pointer fields and one additional field for the index of the field pointing to the parent (0 or 1).

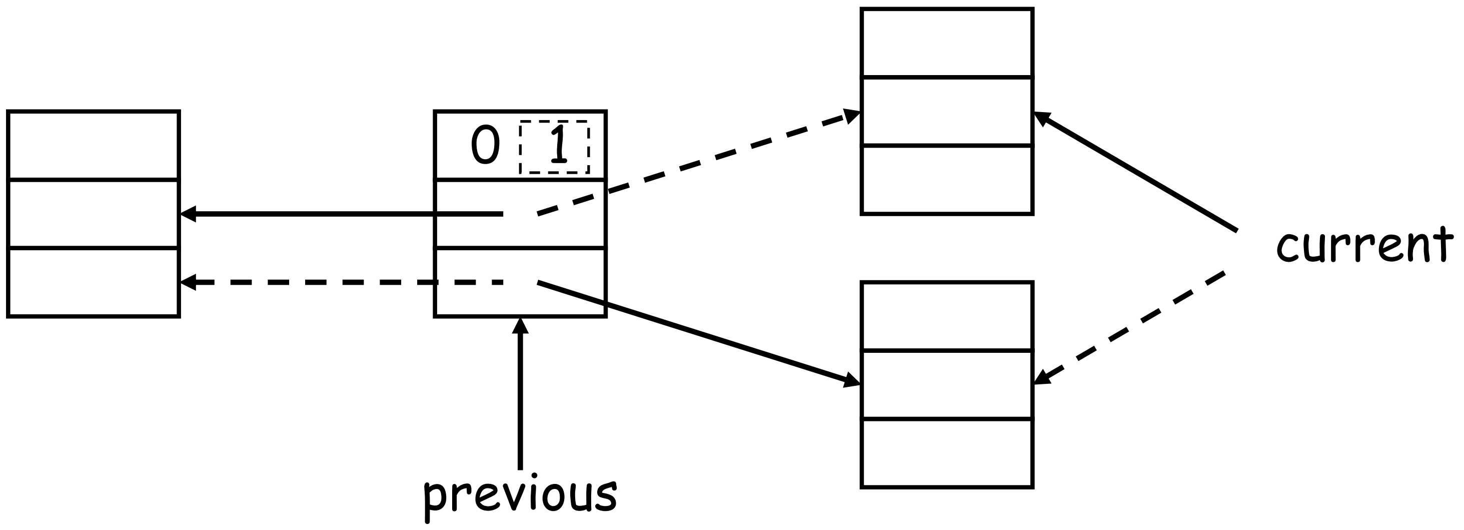
- Three operations form the core. Let the solid lines denote the current state and the dashed lines denote the next state.

- Advance: Increase the depth of the search:

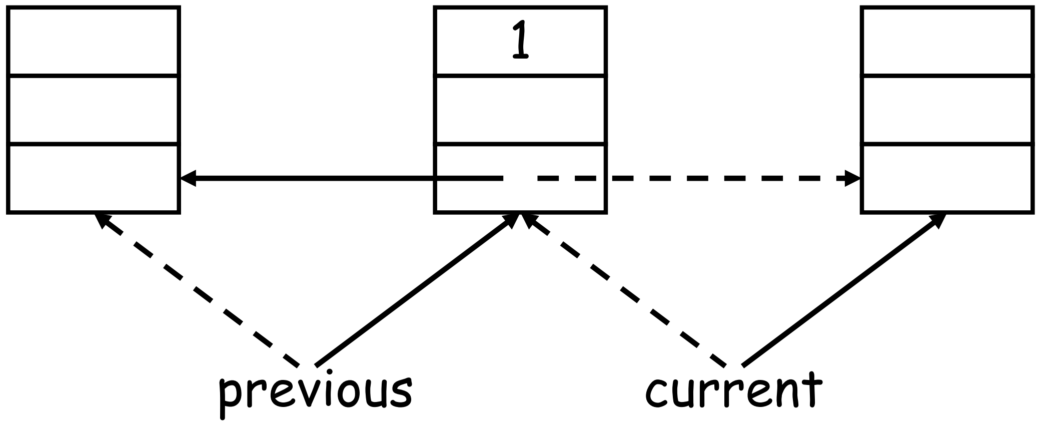


… In-Place Garbage Collection (Deutsch-Schorr-Waite)

- Switch: Visit the next object on the same depth:



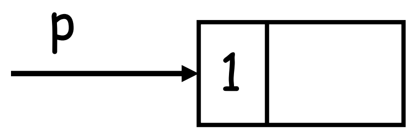
- Retreat: Decrease depth after visiting all objects from previous:



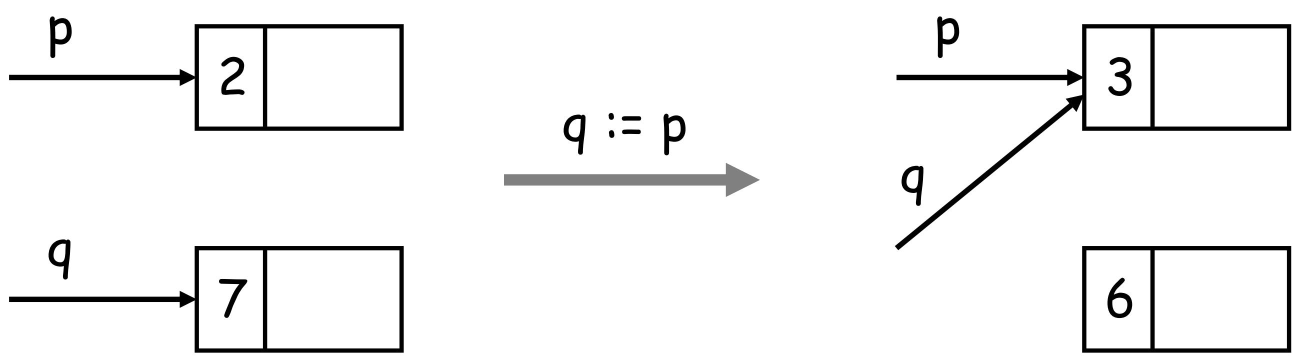
Garbage Collection by Referencing Counting …

- The mark-sweep algorithm collects garbage by <u>tracing</u>.
- An alternative is to use <u>counting</u>: every object on the heap keeps a count of the number of pointers to it. When an object is created, its count is set to 1:

new (p)

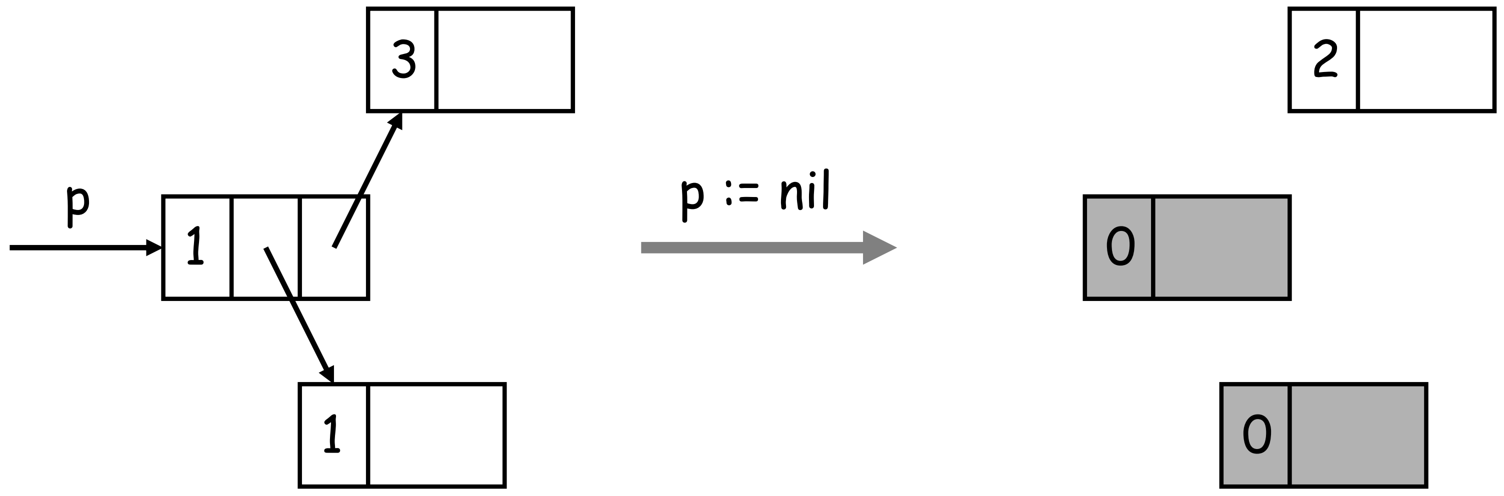


- With every assignment the counts are incremented and decremented:



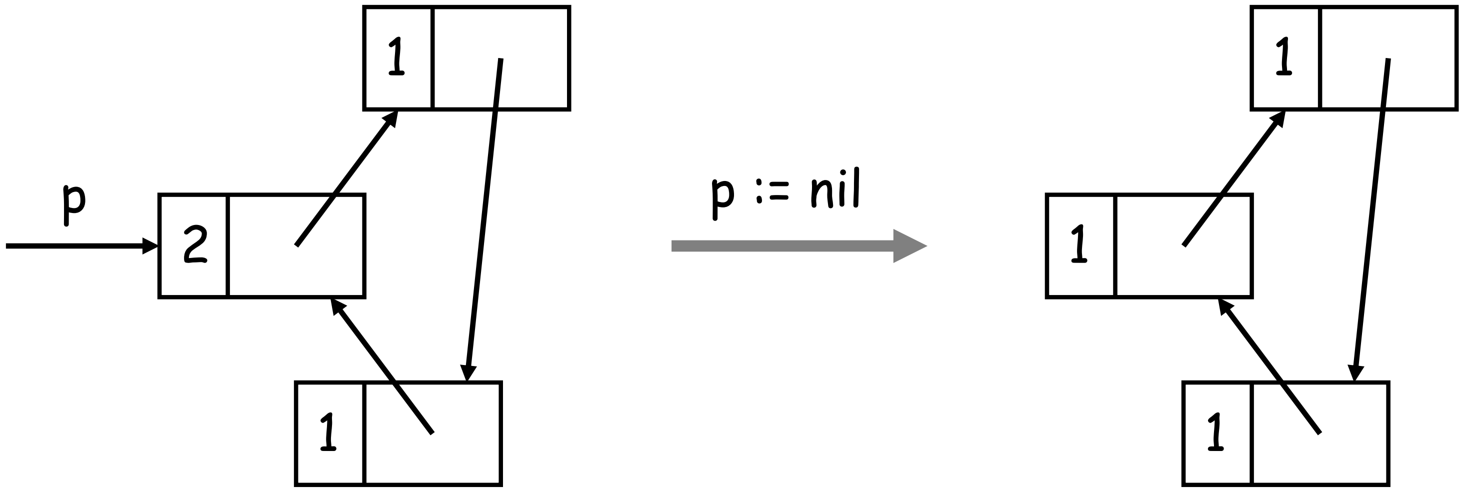
… Garbage Collection by Referencing Counting

- A function procedure call q := m (p) increments the count of p at entry and the count of q on return.
- Local variables and formal parameters cause decrements at the end of their lifetime. Assigning nil causes a decrement.
- When the count reaches 0, the object is reclaimed; objects referenced are visited recursively and their count is decremented as well:



Reference Counting vs. Mark-Sweep Garbage Collection …

- Reference counting cannot detect and reclaim cycles:



- Reference counting imposes a constant overhead with every pointer manipulation. In total, the overhead is larger than with mark-sweep, but is more equally distributed.

… Reference Counting vs. Mark-Sweep Garbage Collection

- The only delay caused by reference counting is when an object is reclaimed and all its children have to be visited, recursively, which can be at unpredictable places in the program and can be deep (e.g. a tree). This delay be further spread out by
  - placing an object with count 0 in the free list
  - postponing the recursive visits until the next memory request.

- Applications of Reference Counting:
  - better suited for real-time programming;
  - used in distributed systems, where tracing all pointers is impractical;
  - used in the Unix file system;
  - used in the Swift language: <u>strong</u>, <u>weak/unowned</u> references are distinguished.

Deferred Reference Counting

- Pointer assignment q := p is costly, i.e.

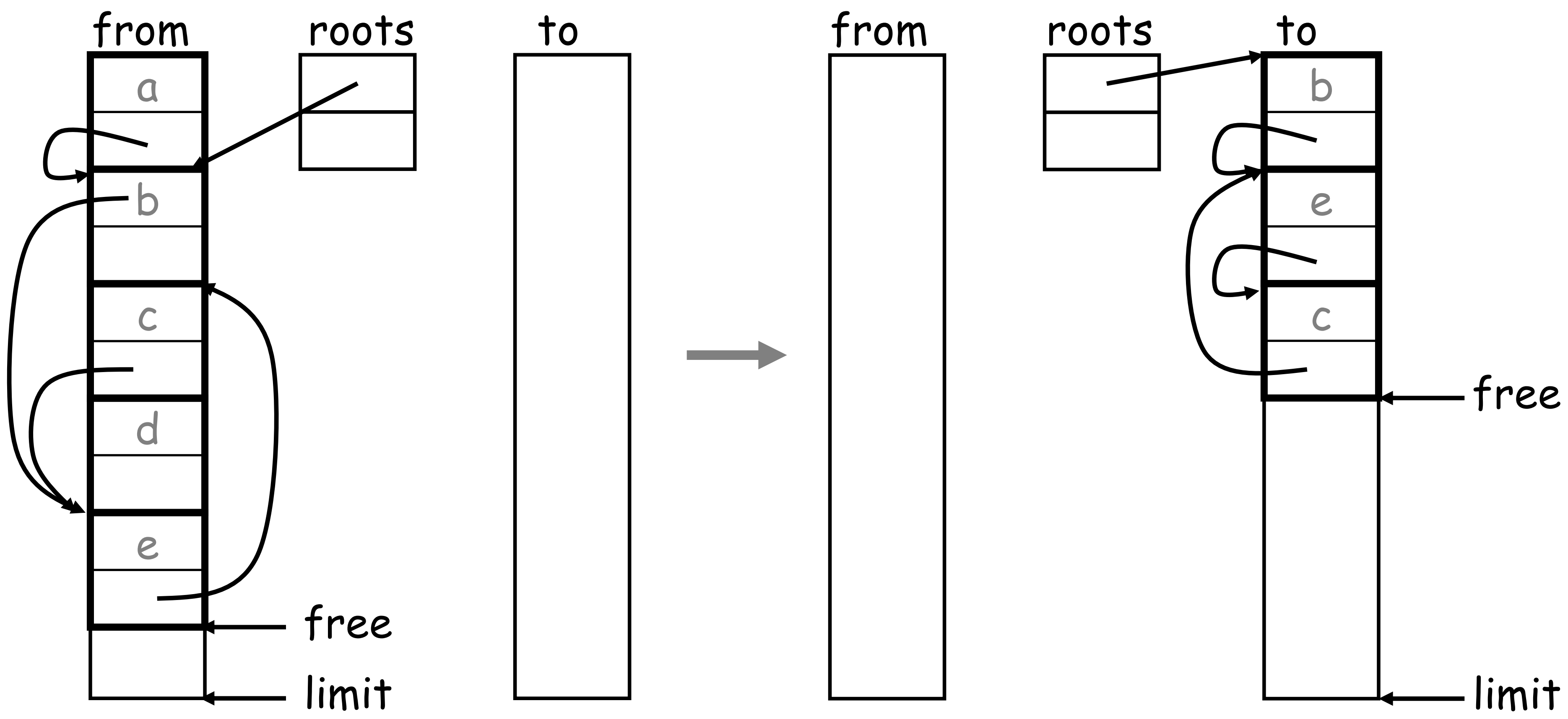
```
R1 := p       vs     R1 := q                       L:  R1 := p
q := R1              R2 := M[R1 + count]               q := R1
                     R2 := R2 - 1                      R2 := M[R1 + count]
                     M[R1 + count] := R2               R2 := R2 + 1
                     if R2 > 0 then L                  M[R1 + count] := R2
                     call putOnFreeList
```

- Most pointer stores are done to local variables. A way to reduce this overhead is to keep only an approximate count: pointer stores to local variables do not update the count, only stores to the heap do; the count reflects then the number of heap pointers only.

- If the count reaches zero, then the stack has to be scanned to check if local variables are pointing to the object before it can be recycled. This can be <u>deferred</u> by placing objects with zero count in a zero count table first and scanning that table periodically.

Copying Collection – 1

- The heap is divided into from-space and to-space. Local and global variables - the roots - point to the from-space:



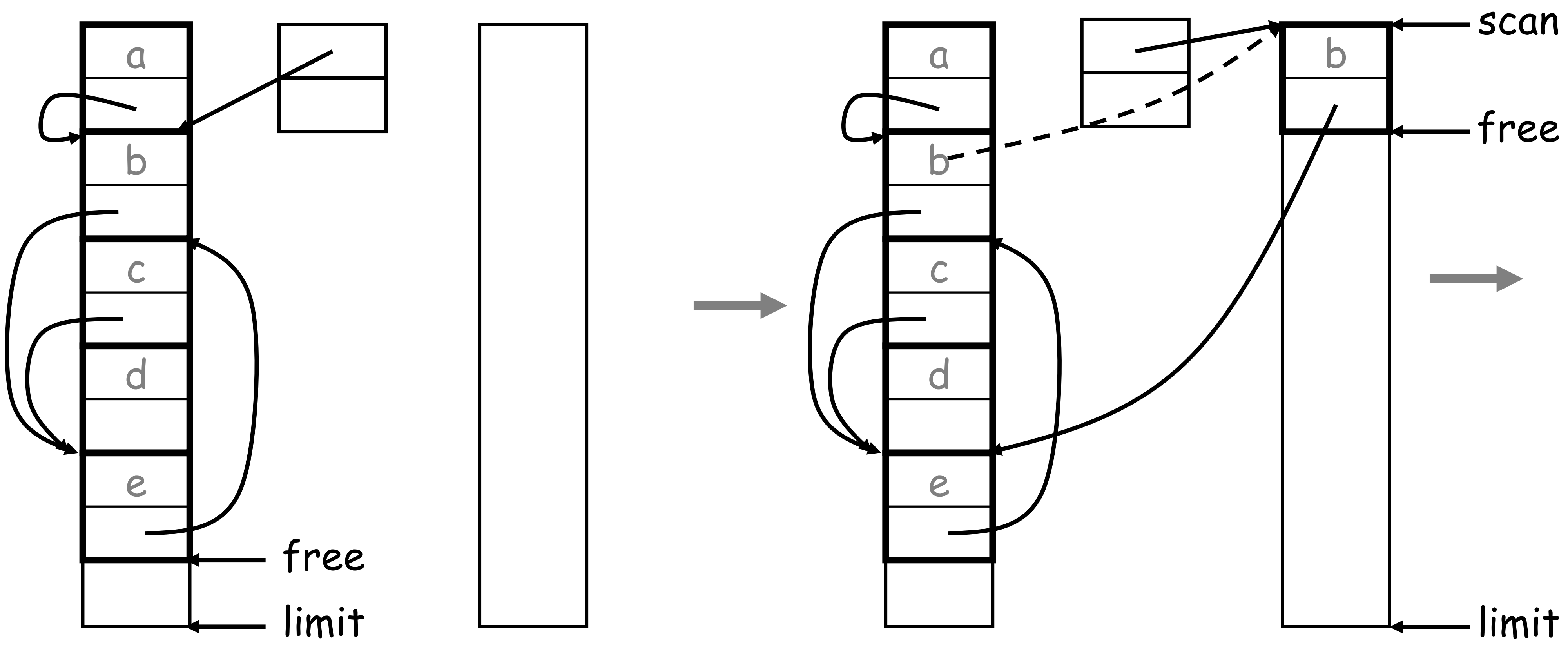
- free points the next free location; if it reaches limit, garbage collection is initiated, copying all reachable objects to the to-space.

Copying Collection – 2

- The key is how pointers in objects are being forwarded. Suppose p is a pointer (in the from-space) pointing to from-space:
- If p points to a from-space object that has already been copied to to-space, then p^.f is a special forwarding pointer that indicates where the copy is; p becomes p^.f in the to-space.
- If p points to a from-space object that has not been copied, then p^ is copied to free in the to-space, p^.f is set to free, p becomes free in the to-space, and free is incremented.
- If p points to an object elsewhere, then p is left unchanged in the to-space.

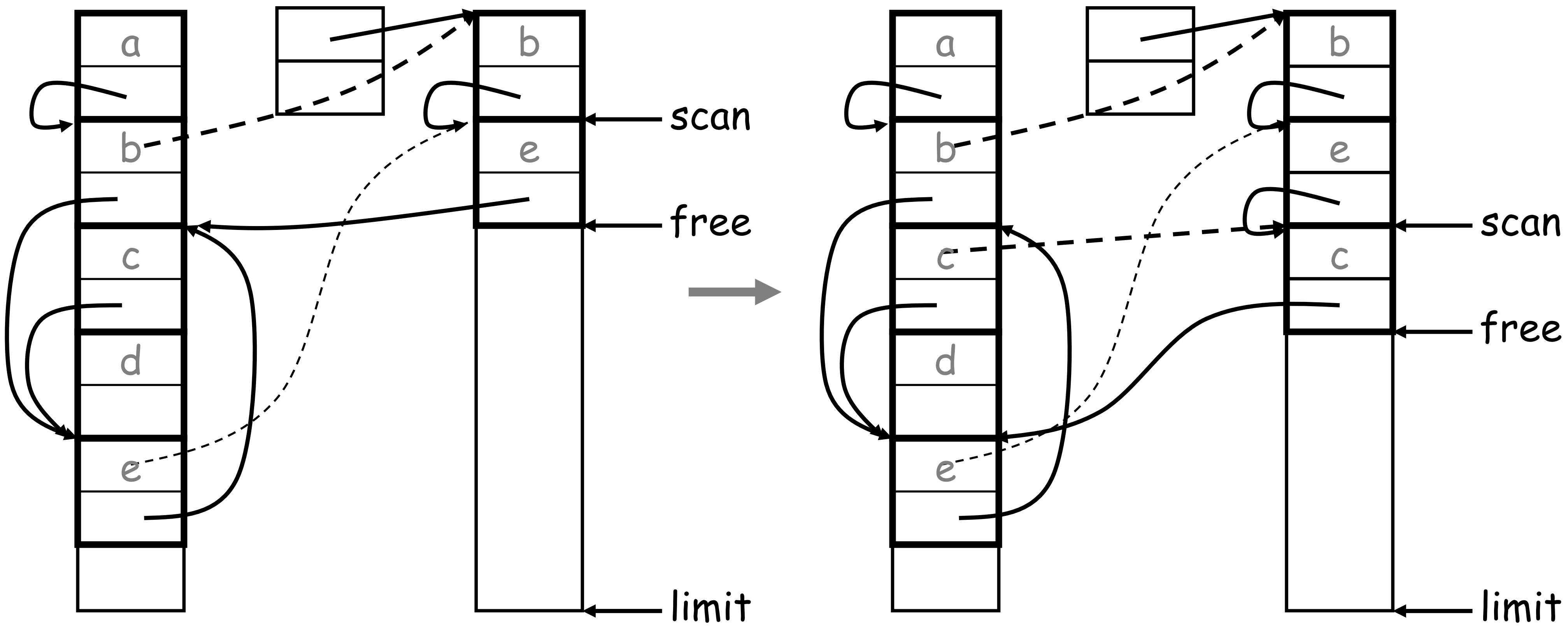
Copying Collection – 3

- Initially the objects reachable from the roots are copied and scan is set to the start of the to-space:



Copying Collection – 4

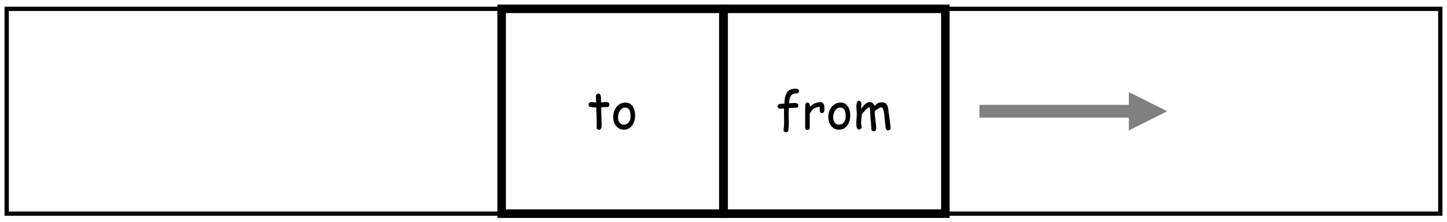
- Then all objects between scan and free are visited and all pointers in those forwarded; we assume f is stored in the first word:



- As the objects between scan and free are a queue of objects to be processed, this is a breadth-first traversal.

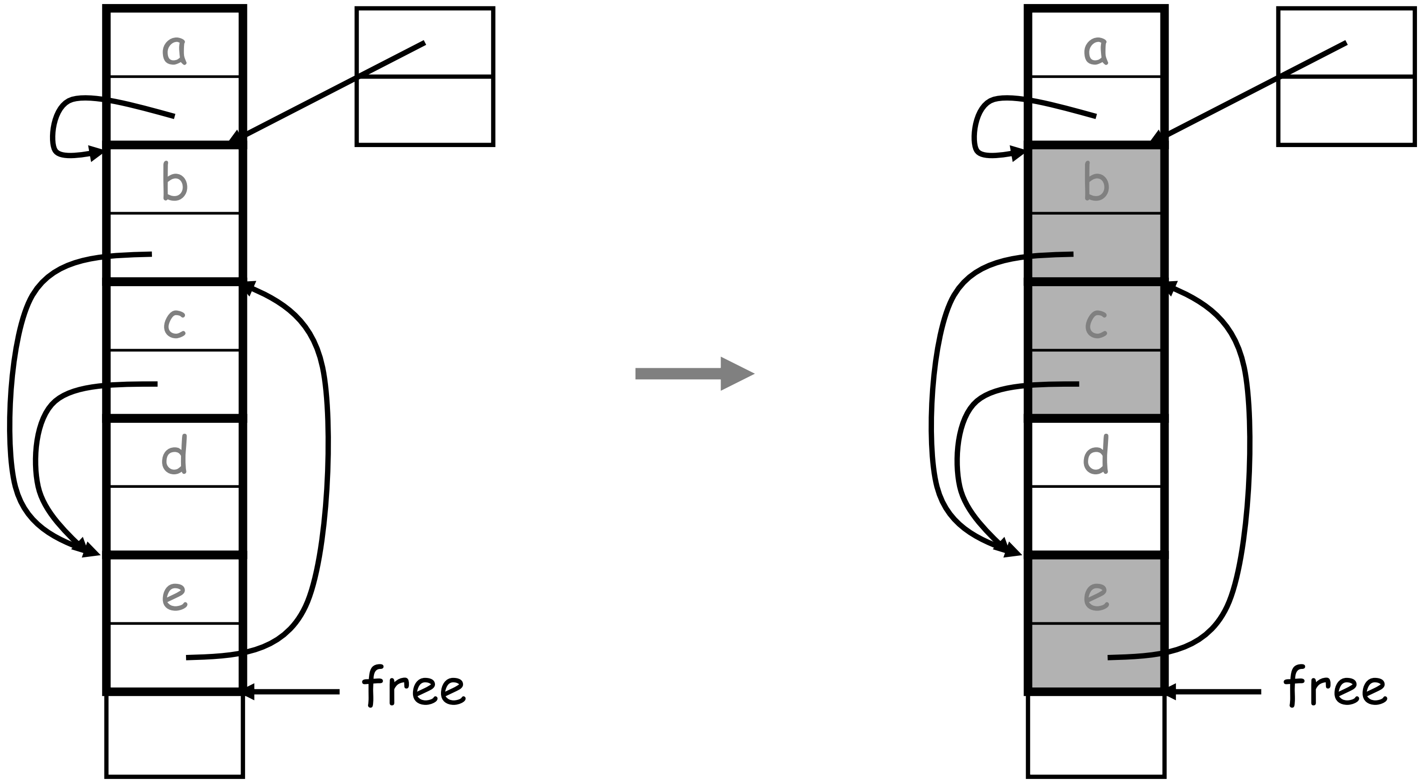
Copying Collection – 5

- Allocation of new objects only involves incrementing the free pointer; no need to maintain lists of free blocks.
- Copying collections also performs a <u>compaction</u>; there is no possibility for fragmentation.
- All reachable objects are copied, but depending on the programming language, many objects are not reachable.
- The main drawback is that only half of the memory can be used. This can be reduced by dividing the memory into n blocks, selecting one to-space and one from-space among those, and sliding the window. However, objects must be smaller than the block size:



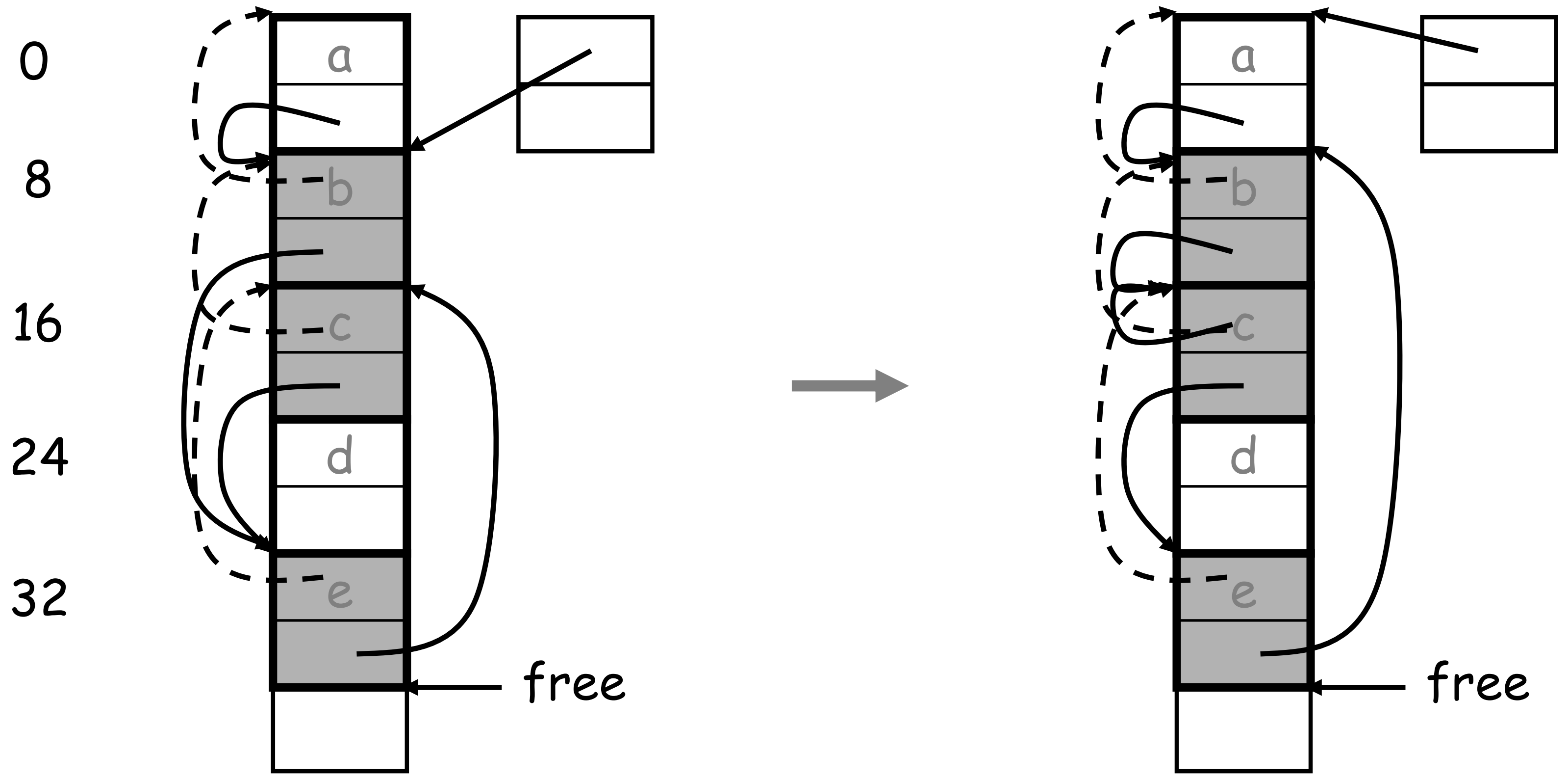
Mark-Compact Garbage Collection – 1

- Mark-compact collection is similar to mark-sweep, but does compaction like copying collection. Initially, starting from the root set, all live objects are marked, like in mark-sweep:



Mark-Compact Garbage Collection – 2

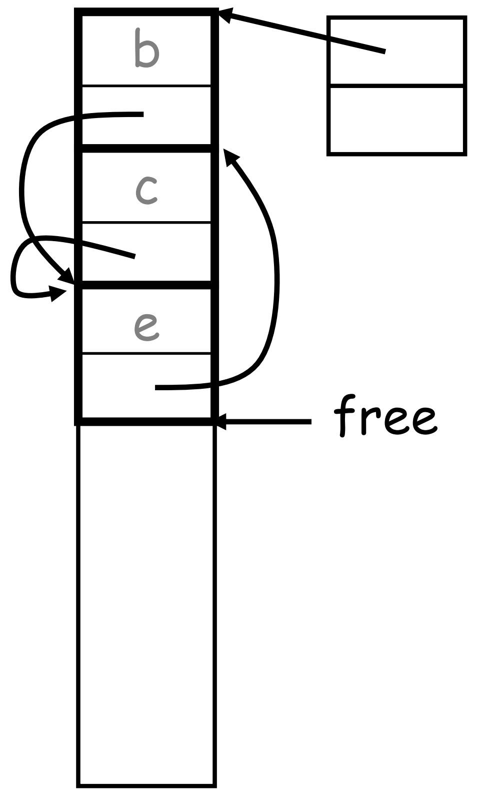
- In a sweep phase, the new address of each object is calculated and stored it is forwarding field; the new address is the sum of the sizes of all objects encountered so far.



- In the third phase, all pointers are updated to point to the new locations.

Mark-Compact Garbage Collection – 3

- In the fourth phase, the objects are moved to their new location; the pointers are left unchanged. On this occasion, all objects are unmarked and all forwarding pointers become unused again:



Mark-Compact Garbage Collection – 4

- Like copying collection, allocation of new objects only involves incrementing the free pointer; no need to maintain free lists.
- One extra pointer for forwarding is needed in each object; mark-sweep does not need extra space, copying collections needs twice as much space.
- Like with mark-sweep, the marking phase traverses the whole heap; if it is larger than main memory, swapping occurs. Copying collection only traverses live objects; sometimes only 5% are live.
- Compaction preserves the original order of objects, unlike copying collection; this supports locality and better caching.
- Mark-compact tends to accumulate long-lived objects at the bottom of the heap and would not move them. Copying collections always moves objects, whether small or large.

Generational Garbage Collection – 1

- Most objects die shortly after they are created: either an object is created as a "local" variable by the program or objects are created to hold the result of expression evaluation, e.g.:

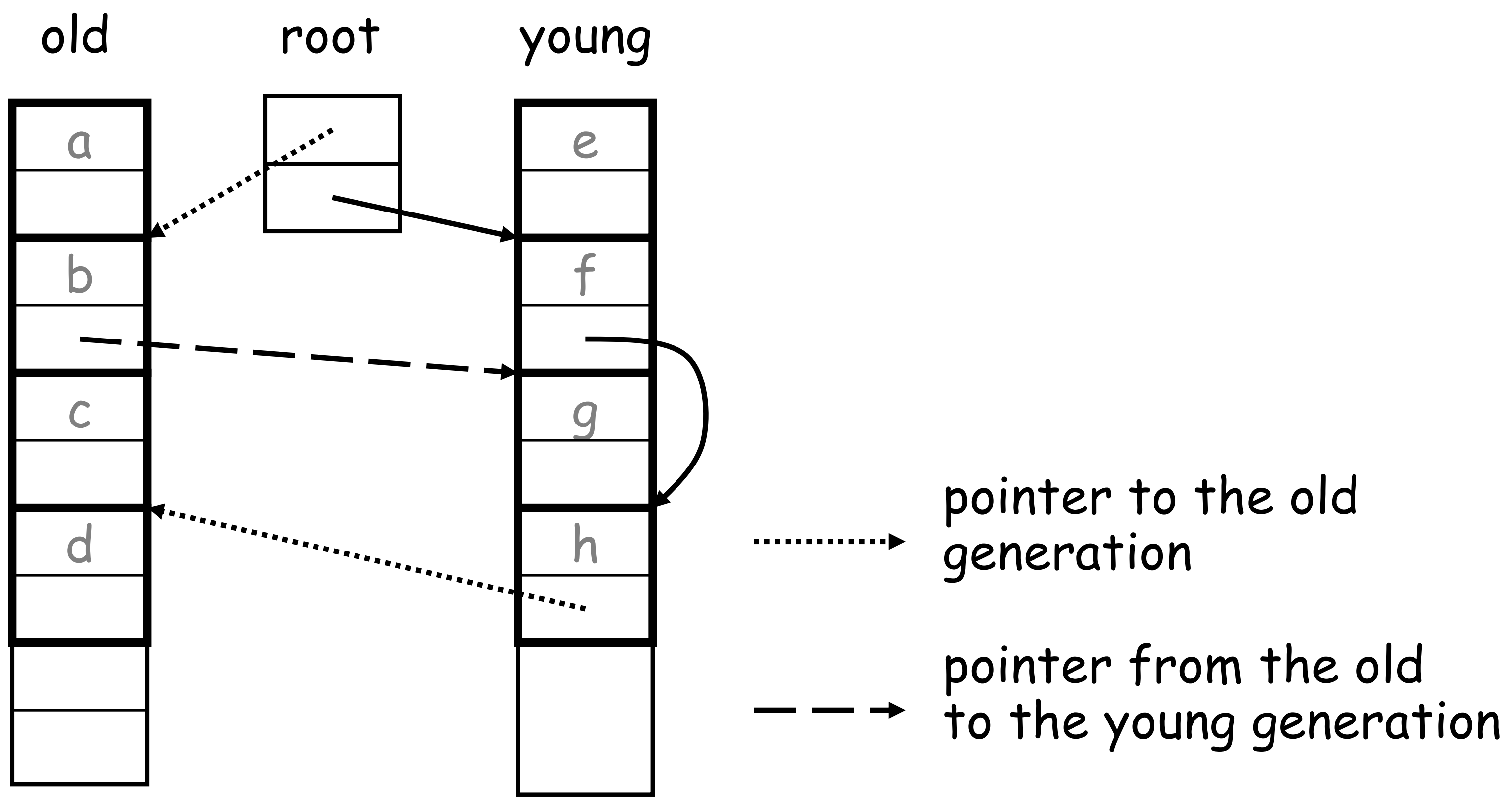
```
        System.out.println ("a = " + a);            assuming a is integer
```

- Almost all objects that are collected have been created since the last collection. On the other hand, once an object survives the first collection, it is likely to survive subsequent ones.

- Generational collection divides the heap into a number of generations, say an old and a young generation. Different strategies are used for different generations:
  - A tracing collection like mark-compact is used on the old generation.
  - A copying collector is used on the new generation.

Generational Garbage Collection – 2

- Once the heap is full, a minor collection on the youngest generation is performed; if that does not reclaim sufficient memory, the next older generation is collected.


old
root
young
a
b
c
d
e
f
g
h
pointer to the old generation
pointer from the old to the young generation

- The young generation is collected by following the pointers from the root set but ignoring those to the old generation.

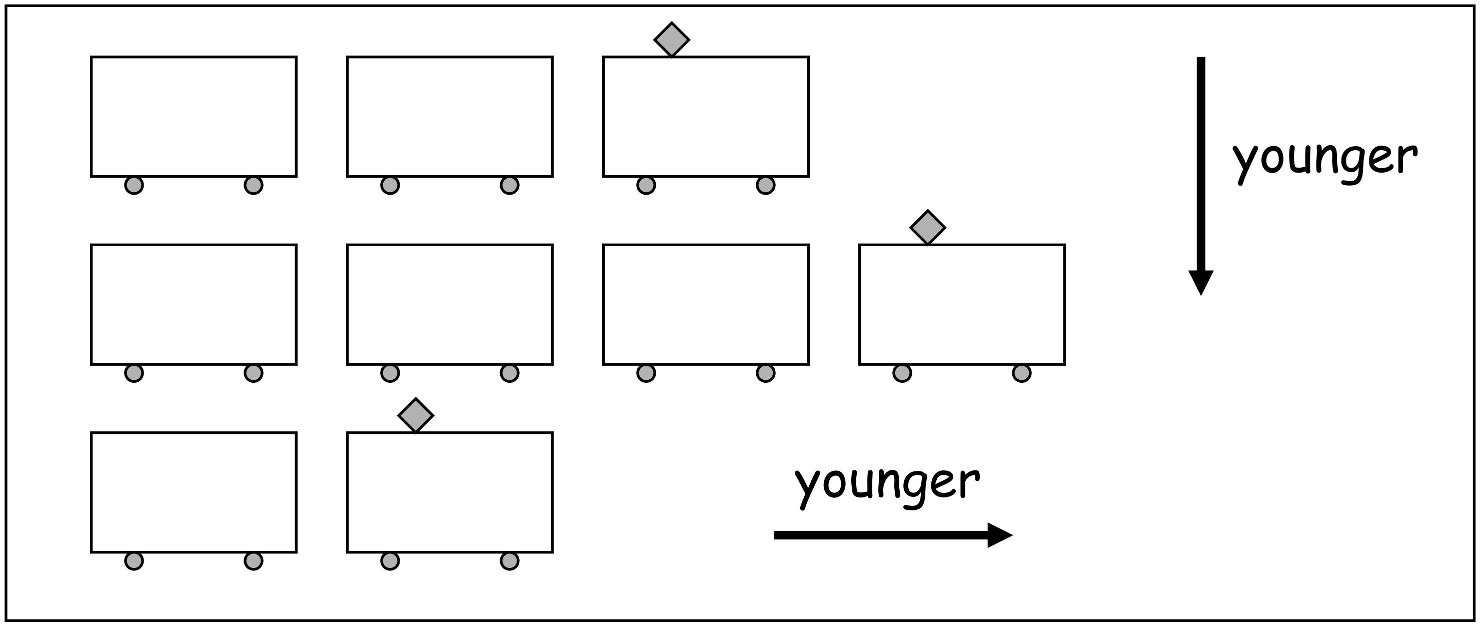
Generational Garbage Collection – 3

- However, pointers from the old generation to the new generation can exist and have to be treated as belonging to the root set. Several options exists:
  - Remembered set: whenever a pointer to a new object is stored in an old object, the pointer is added to a remembered set; whenever a new object becomes old, all its pointers to new objects are placed in the remembered set as well.
  - Card marking: the heap is divided into cards of equal size and a vector with a bit for every card is maintained. Whenever a pointer is stored in an object, the bit for the location of the pointer is set. A minor collection will check all the marked cards in the old generation. (Used by JDK 1.4.)
  - Page marking: if the card size is the page size of the virtual memory, the dirty bit of the page can be used for marking; this assumes that the dirty bit is available to user programs!

Incremental Collection of Older Generations - 1

- The problem with mark-compact is that it can be disruptive as the whole heap (of the older generation) is traversed and compacted. An idea is to split the heap into a number of blocks and collect garbage block by block. The train algorithm does allow that and also performs compaction.

- Blocks are called cars; cars of one generation are called a train. Cars within a train and trains are kept in lists according to their age (cars don't have to be allocated consecutively in memory).

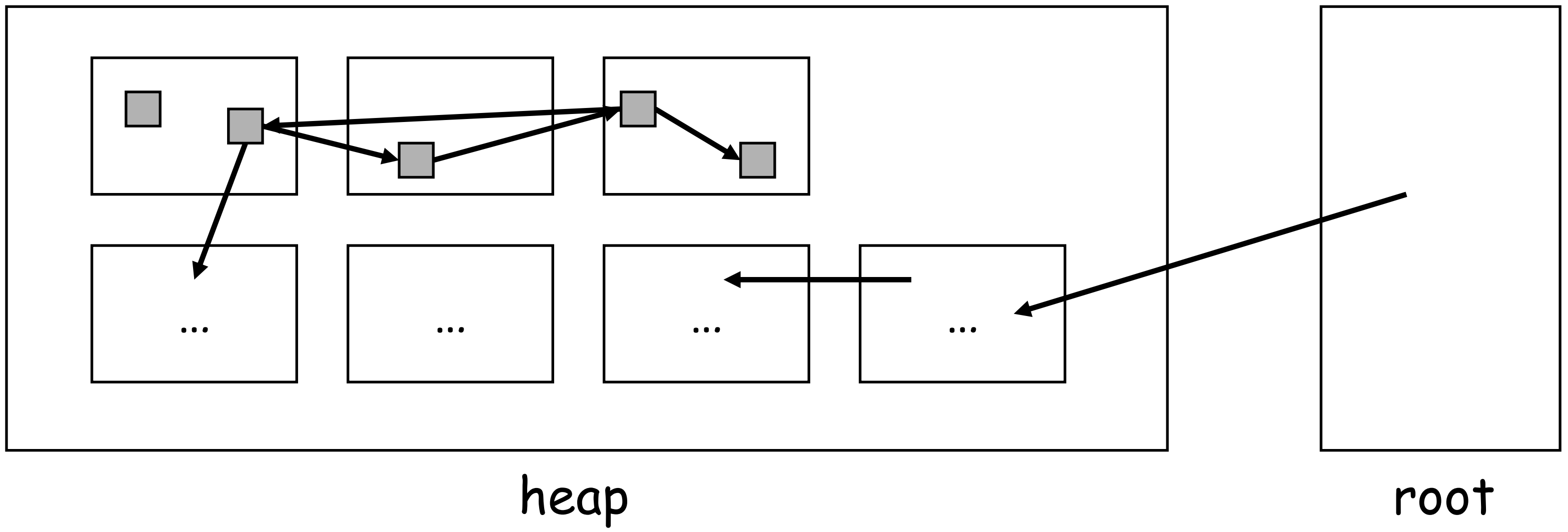


Incremental Collection of Older Objects - 2

- We assume that objects can fit into one car; objects larger than a car have to be put in a <u>large object space</u>, but can be treated there similarly.
- The root set includes pointers on the stack, global variables, and pointers in the young generation.
- Each car has a remembered set with cars pointing to it. We assume that, given a remembered set of a car, we can find all pointers within all objects pointing to objects in that car. For this, we have to know which fields in objects are pointers.
- The algorithm inspects the oldest train. First, a check is made if there are no pointers from the root set into that train and if all pointers to that train originate only from that train. For this, all entries of the remembered set of each car are inspected.

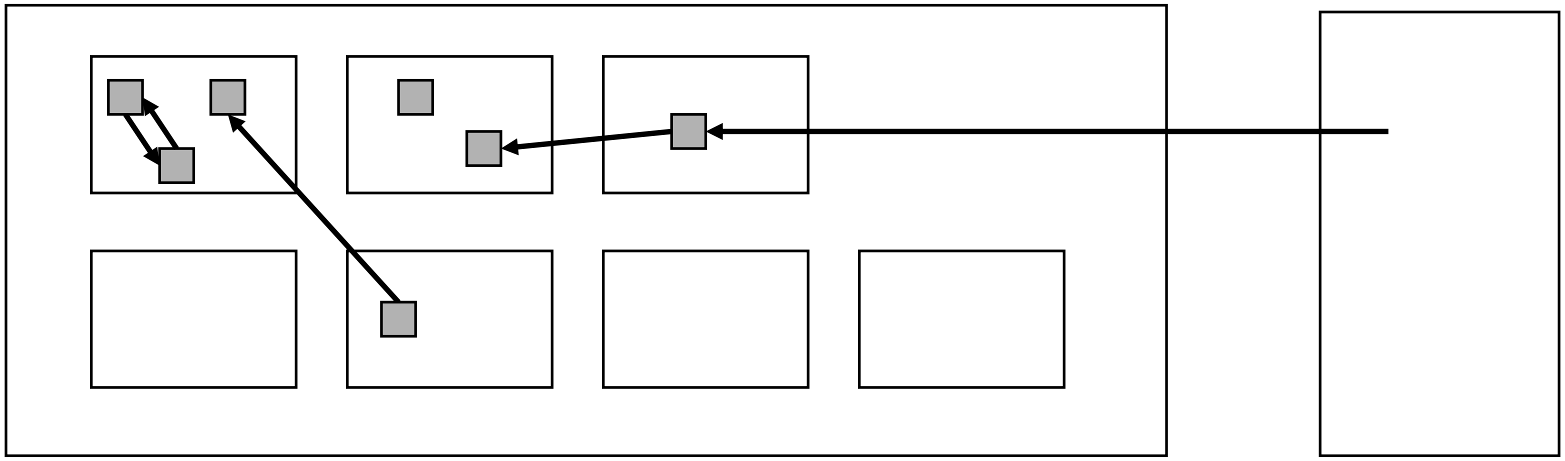
Incremental Collection of Older Objects - 3

- If there are no pointers from the root and from any other train to the oldest train, the whole train is garbage and is recycled.

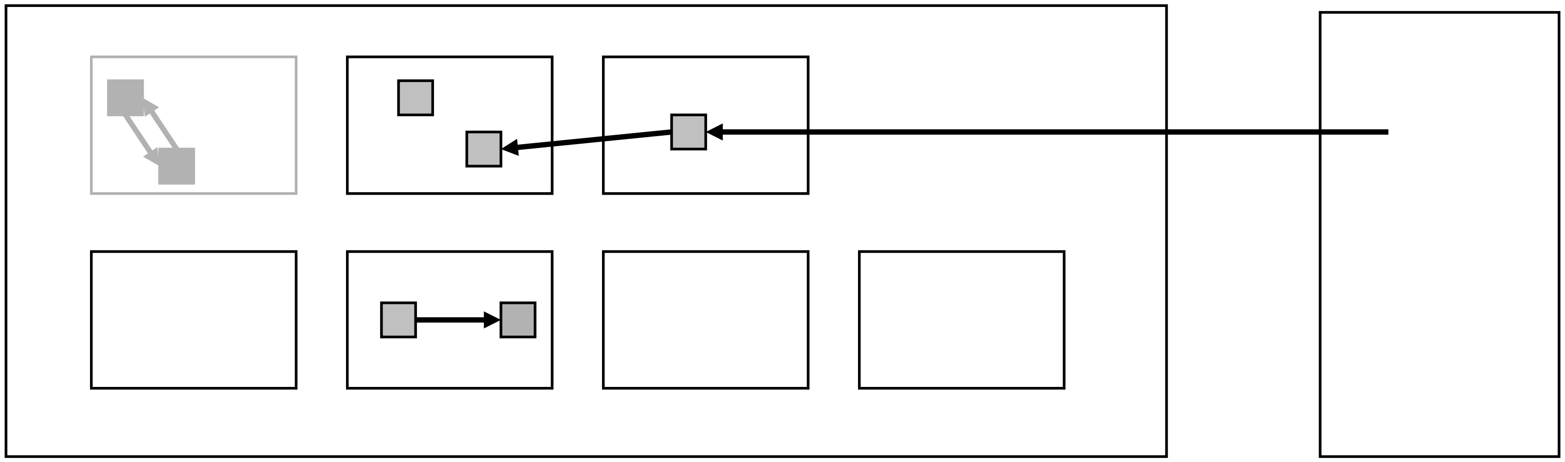


- Otherwise, the oldest car in the oldest train is inspected: each object is moved to a car from which there is a pointer to it; if that car is full, a new car is appended to that train. (In the pictures, we assume only three objects can fit in a car.)

Incremental Collection of Older Objects - 4



- If there are objects with no pointers to them in the oldest car, their space will be reclaimed with the car; this is also the case if there is "cyclic garbage" within a car.



Incremental Collection of Older Objects - 5

- Over time, connected garbage structures will migrate in one car, provided they fit all into a car, and that car will eventually be recycled. The key observation is how cyclic garbage structures that don't fit into a car are handled: they ultimately become the only objects of the oldest train, which is then reclaimed in the first step of the algorithm.

- The key in the implementation is how the remembered sets are updated: similar to card marking, each car has 2^n bytes and keeps a bitvector with the cars pointing to it. The bitvectors can be updated in constant time by shifting a pointer n bits to the right to obtain the index in the bitvector that needs to be set.

- New generations are added as they survive the copying collection of the youngest objects. (The algorithm is implemented in Java 1.4.)

Concurrent Copying Collection …

- Concurrent collection distinguishes between the mutator process – the main program – the collector process (or thread). Following form of copying collection is known as Baker's algorithm.

- Collection is initiated by flipping the roles of the from and to space and forwarding all root pointers. The mutator then resumes.

- Whenever the mutator requests a new object, that is placed at the end of the to-space by decrementing limit, and several objects at scan are forwarded.

- The mutator must only point to to-space. On each access to an object, a check is necessary if that object is in the to-space. If not, it is forwarded immediately.

… Concurrent Copying Collection

- The collector can process concurrently by forwarding objects at scan.
- In order to prevent that allocation uses up the to-space before the copying is finished, only half of the to-space (and from-space) is used; for each allocation by the mutator, one word must be scanned, such that in the worst case free, scan, and limit all meet in the middle of the to-space. Thus, only 1/4 of the heap can be used.
- The extra cost of this algorithm are the instructions that have to follow every access of an object to check whether it is already in the to-space; this can be a significant overhead if it is not supported by hardware.

Compiler Extensions for Garbage Collection

- The compiler must generate a type descriptor information for every type of objects on the heap and add it to the object file.
- The program loader must generate type descriptor objects from the information in the object file.
- The procedure new(p) must, after allocating memory, set the type tag appropriately.
- The run-time garbage collector must have a way of determining all objects on the heap and all possible pointers. The last point is easier for global pointer variables since the compiler can provide this information. It is more difficult for pointer variables within procedures. For example, the Oberon system does garbage collection only between procedure calls.