Memory Management and Garbage Collection

Managing Memory

Managing Memory

- Storage management is still a hard problem in modern programming
- C and C++ programs have many storage bugs
  - forgetting to free unused memory
  - dereferencing a dangling pointer
  - overwriting parts of a data structure by accident
  - and so on...
- Storage bugs are hard to find
  - a bug can lead to a visible effect far away in time and program text from the source

Managing Memory

- This is an old problem:
  - studied since the 1950s for LISP

- There are well-known techniques for completely automatic memory management

- Became mainstream with the popularity of Java

Managing Memory

- When an object is created, unused space is automatically allocated

- After a while there is no more unused space

- Some space is occupied by objects that will never be used again
  - This space can be freed to be reused later

Managing Memory

- How do we know an object will “never be used again”?
- Observation: a program can use only the objects that it can find:

```
…
var x = new Object();
x = y;
```

Managing Memory

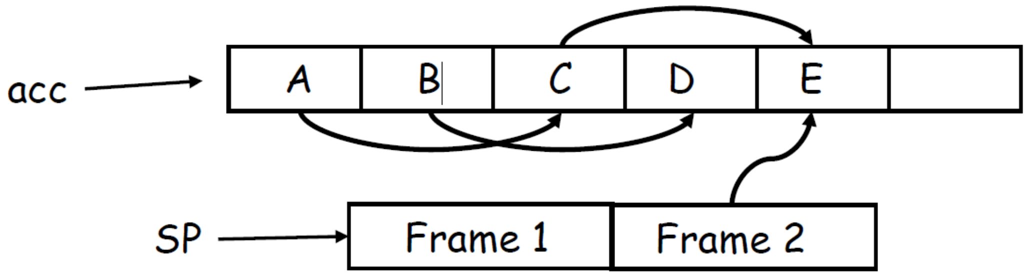
- An object x is reachable if and only if:
  - a register contains a pointer to x, or
  - another reachable object y contains a pointer to x
- You can find all reachable objects by starting from registers and following all the pointers
- An unreachable object can never be used
  - such objects are garbage

Managing Memory

- An accumulator
  - it points to an object
  - and this object may point to other objects, etc.

- And a stack pointer
  - each stack frame contains pointers
    - e.g., method parameters
  - each stack frame also contains non-pointers
    - e.g., return address
  - if we know the layout of the frame we can find the pointers in it

Managing Memory



- We start tracing from acc and stack
  - These are the roots

- Note B and D are unreachable from acc and stack
  - Thus we can reuse their storage

Managing Memory

- Every garbage collection scheme has the following steps
  1. Allocate space as needed for new objects
  2. When space runs out:
     a) Compute what objects might be used again (generally by tracing objects reachable from a set of “root” registers)
     b) Free the space used by objects not found in (a)

- Some strategies perform garbage collection before the space actually runs out

Mark and Sweep

Mark and Sweep

- When memory runs out, GC executes two phases
  - the mark phase: traces reachable objects
  - the sweep phase: collects garbage objects

- Every object has an extra bit: the mark bit
  - reserved for memory management
  - initially the mark bit is 0
  - set to 1 for the reachable objects in the mark phase

Mark and Sweep

```
// mark phase

let todo = { all roots }
while todo $\neq \emptyset$ do
    pick v $\in$ todo
    todo $\leftarrow$ todo - { v }
    if mark(v) = 0 then        // v is unmarked yet
        mark(v) $\leftarrow$ 1
        let $v_1,\ldots,v_n$ be the pointers contained in v
        todo $\leftarrow$ todo $\cup$ {$v_1,\ldots,v_n$}
    fi
od
```

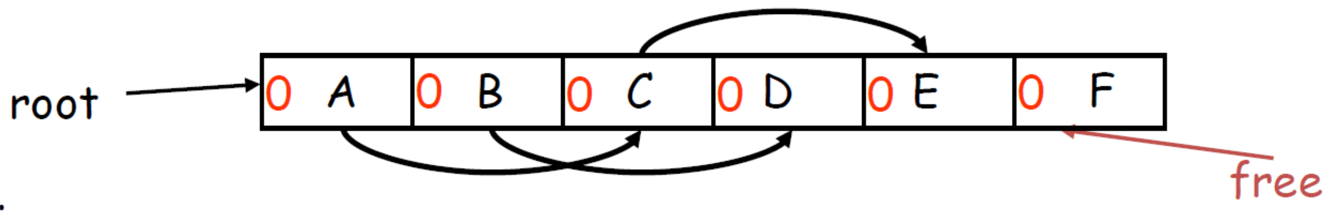
Mark and Sweep

- The sweep phase scans the heap looking for objects with mark bit 0
  - these objects were not visited in the mark phase
  - they are garbage
- Any such object is added to the free list
- Objects with a mark bit 1 have their mark bit reset to 0

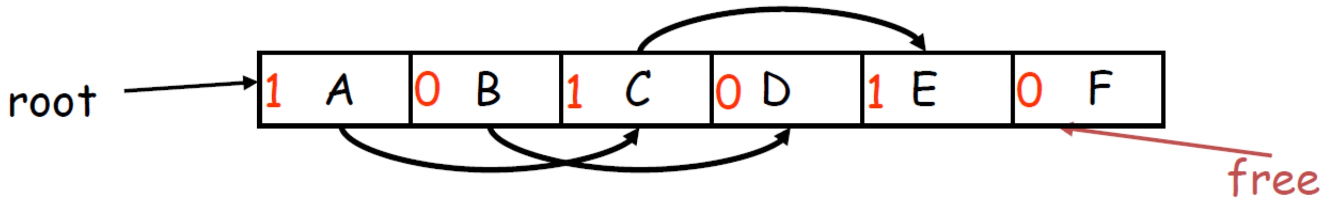
Mark and Sweep

```
// sweep phase
// sizeof(p) is the size of block starting at p

p ← bottom of heap
while p < top of heap do
  if mark(p) = 1 then
     mark(p) ← 0
  else
     add block p...(p+sizeof(p)-1) to freelist
  fi
  p ← p + sizeof(p)
od
```

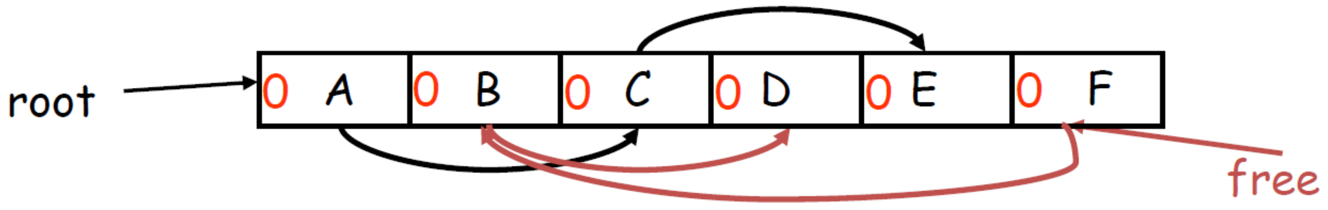
Mark and Sweep



After mark:

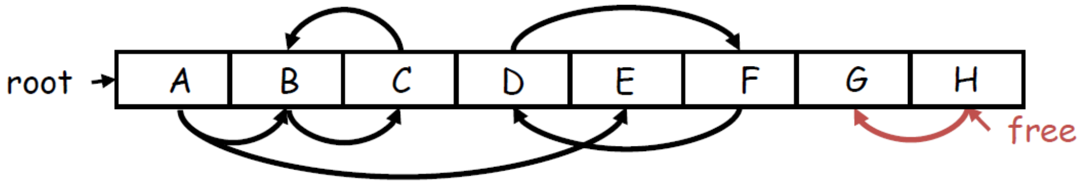


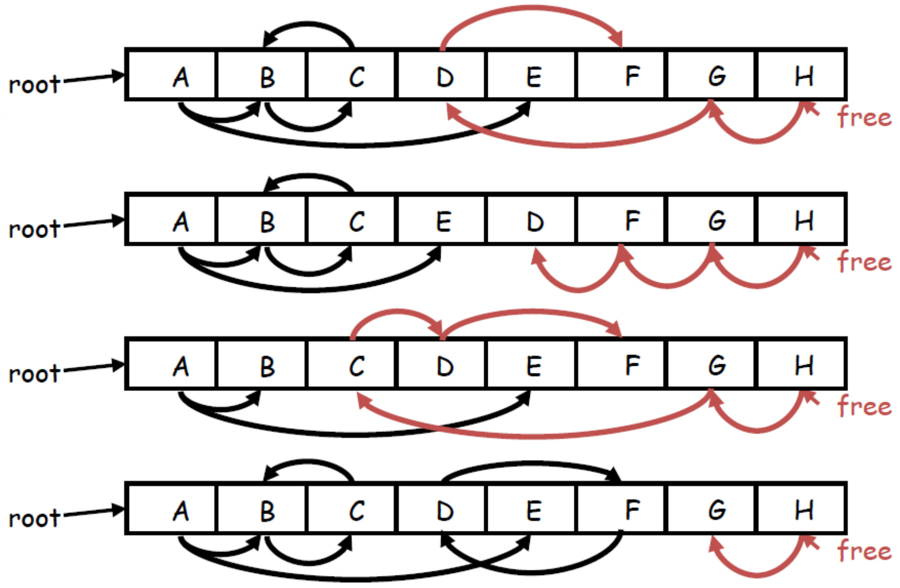
After sweep:



Mark and Sweep: Quiz

Choose the correct final heap after mark and sweep garbage collection.





Mark and Sweep

- While conceptually simple, this algorithm has a number of tricky details
  - typical of GC algorithms

- A serious problem with the mark phase
  - it is invoked when we are out of space
  - yet it needs space to construct the todo list
  - the size of the todo list is unbounded so we cannot reserve space for it a priori

Mark and Sweep

- The todo list is used as an auxiliary data structure to perform the reachability analysis

- There is a trick that allows the auxiliary data to be stored in the objects themselves
  - pointer reversal: when a pointer is followed it is reversed to point to its parent

- Similarly, the free list is stored in the free objects themselves

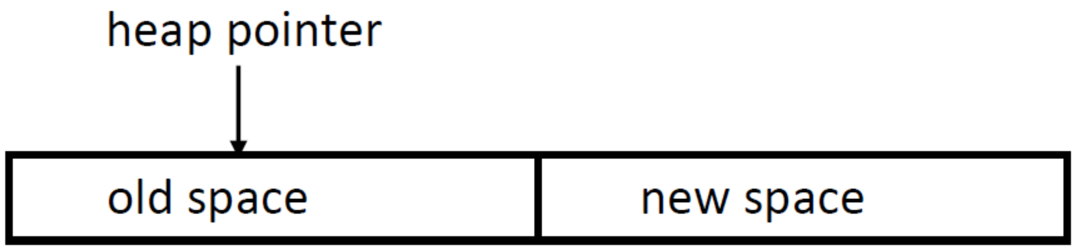
Mark and Sweep

- Space for a new object is allocated from the free list
  - a block large enough is picked
  - an area of the necessary size is allocated from it
  - the left-over is put back in the free list
- Mark and sweep can fragment the memory
  - merge free blocks possible
- Advantage: objects are not moved during GC
  - no need to update the pointers to objects
  - works for languages like C and C++

Stop and Copy

Stop and Copy

- Memory is organized into two areas
  - old space: used for allocation
  - new space: used as a reserve for GC



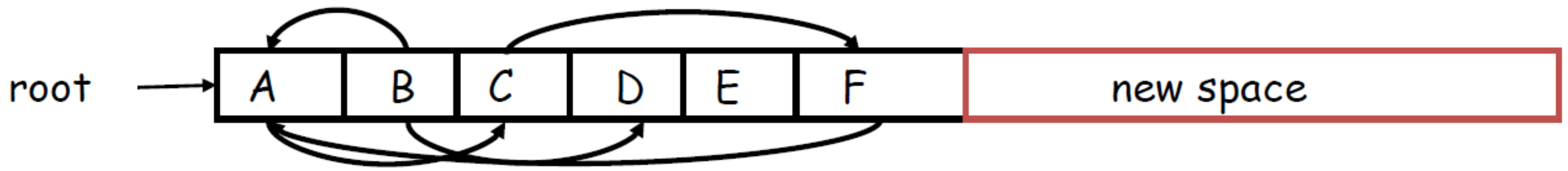
- The heap pointer points to the next free word in the old space
- Allocation just advances the heap pointer

Stop and Copy

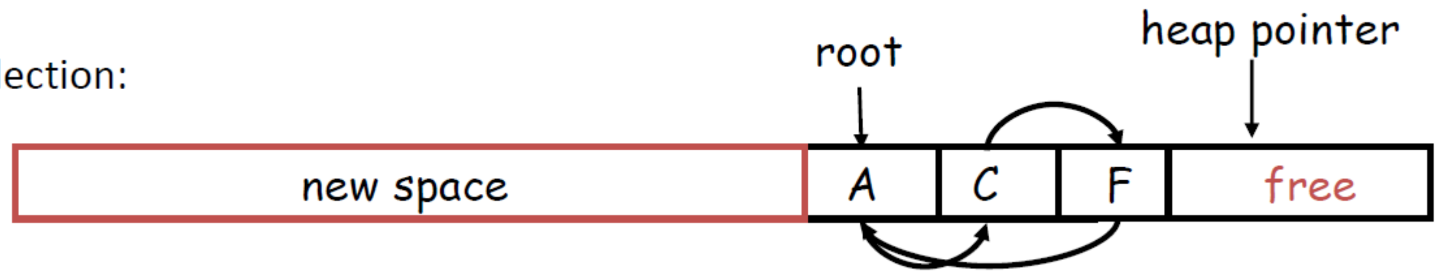
- Starts when the old space is full
- Copies all reachable objects from old space into new space
  - garbage is left behind
  - after the copy phase the new space uses less space than the old one before the collection
- After the copy the roles of the old and new spaces are reversed and the program resumes

Stop and Copy

Before collection:



After collection:

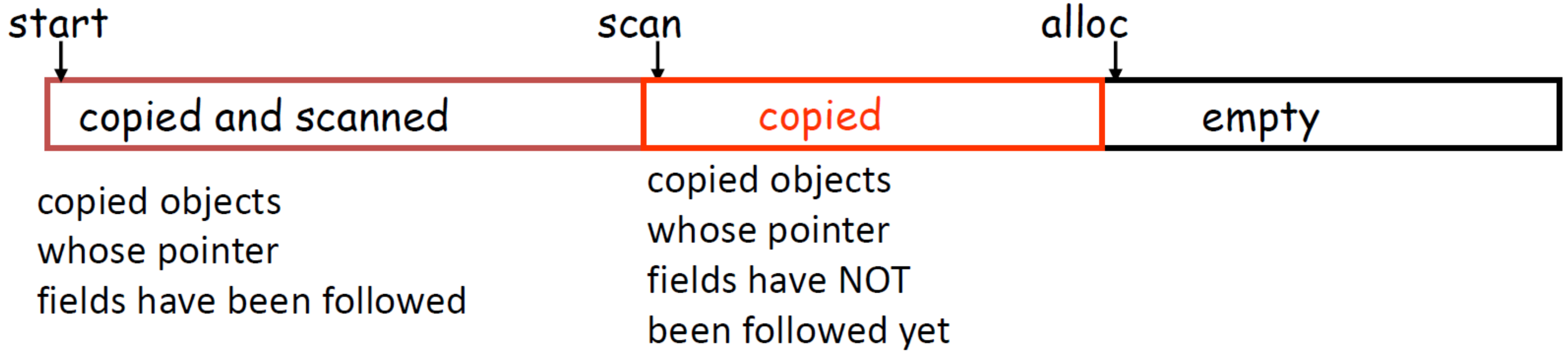


Stop and Copy

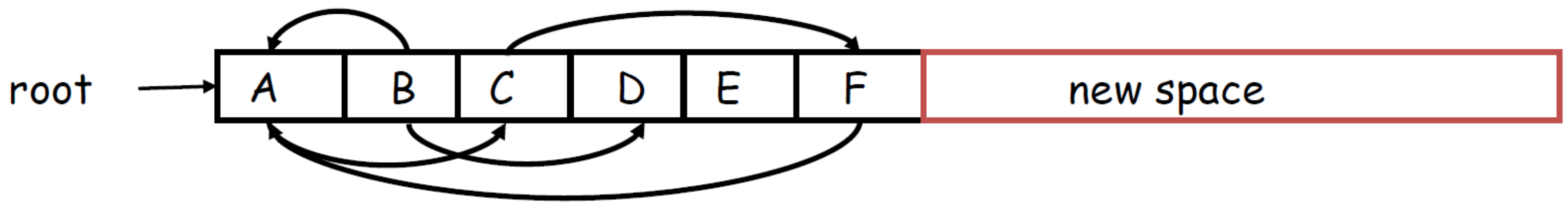
- We need to find all the reachable objects, as for mark and sweep
- As we find a reachable object we copy it into the new space
  - And we have to fix ALL pointers pointing to it!
- As we copy an object we store in the old copy a <u>forwarding pointer</u> to the new copy
  - when we later reach an object with a forwarding pointer we know it was already copied

Stop and Copy

- We still have the issue of how to implement the traversal without using extra space
- The following trick solves the problem:
  - partition the new space in three contiguous regions

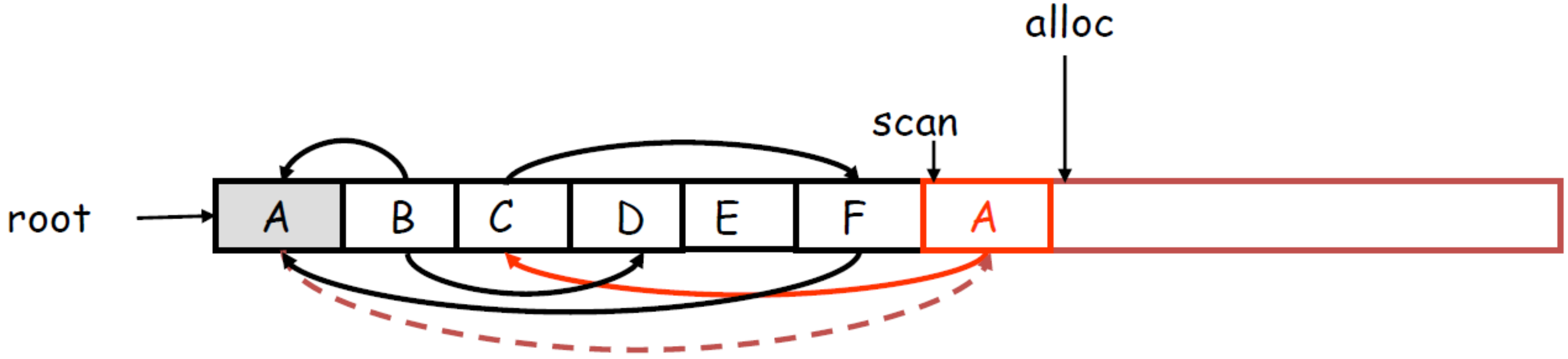

start
scan
alloc
copied and scanned
copied
empty
copied objects whose pointer fields have been followed
copied objects whose pointer fields have NOT been followed yet

Stop and Copy



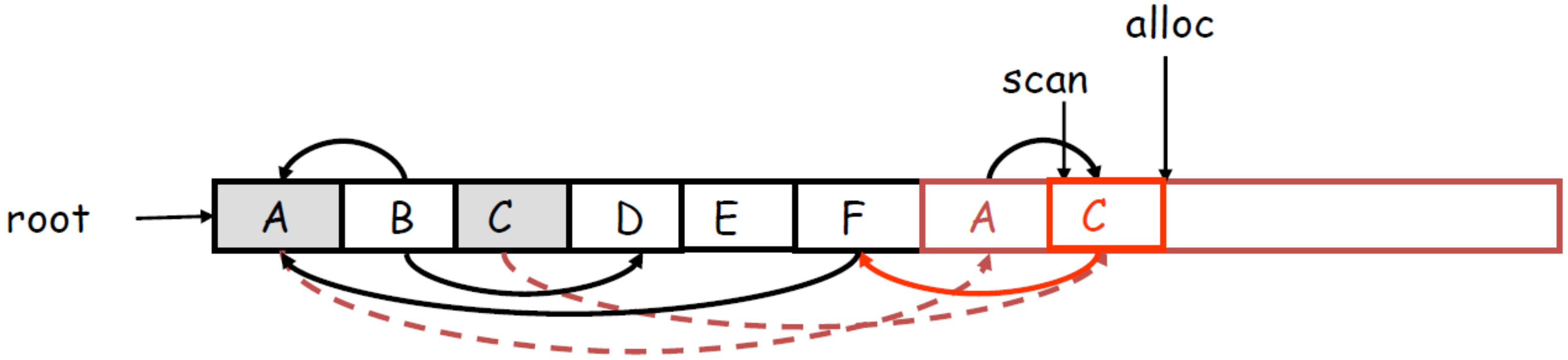
Stop and Copy

- Step 1: Copy the objects pointed to by roots and set forwarding pointers



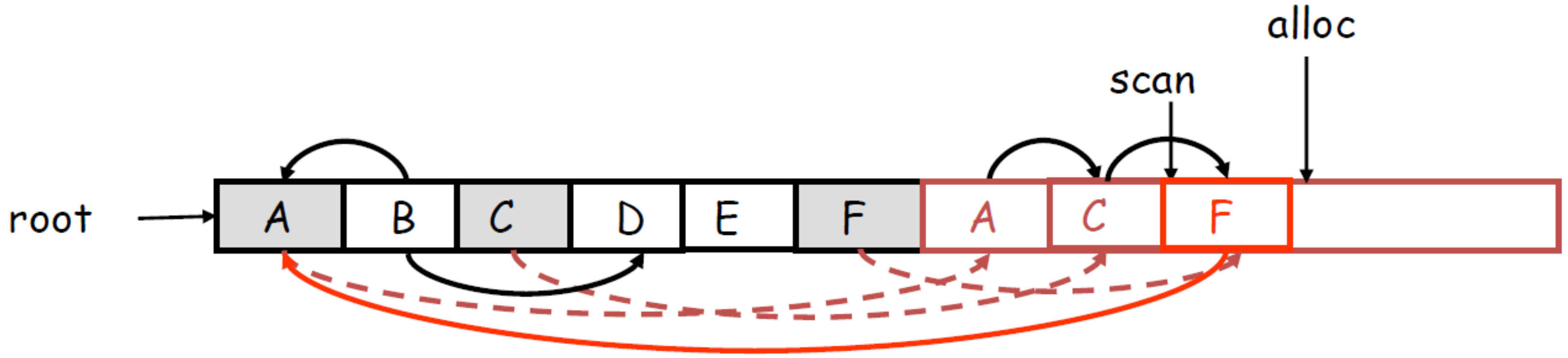
Stop and Copy

- Step 2: Follow the pointer in the next unscanned object (A)
  - copy the pointed-to objects (just C in this case)
  - fix the pointer in A
  - set forwarding pointer



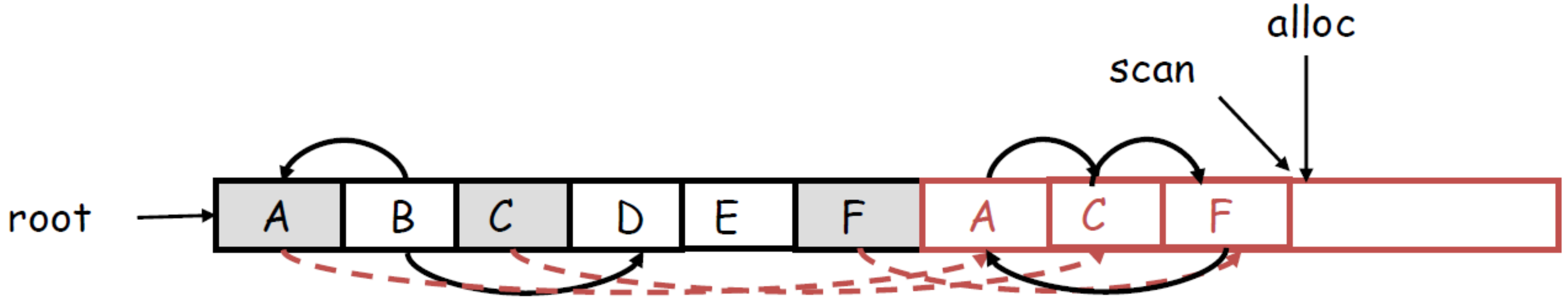
Stop and Copy

- Follow the pointer in the next unscanned object (C)
  - copy the pointed objects (F in this case)



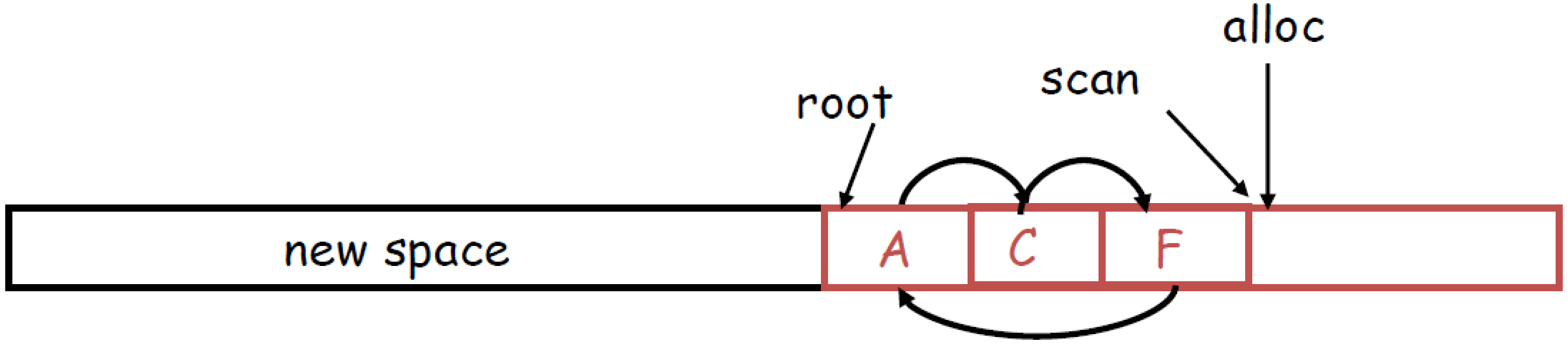
Stop and Copy

- Follow the pointer in the next unscanned object (F)
  - the pointed object (A) was already copied. Set the pointer same as the forwading pointer



Stop and Copy

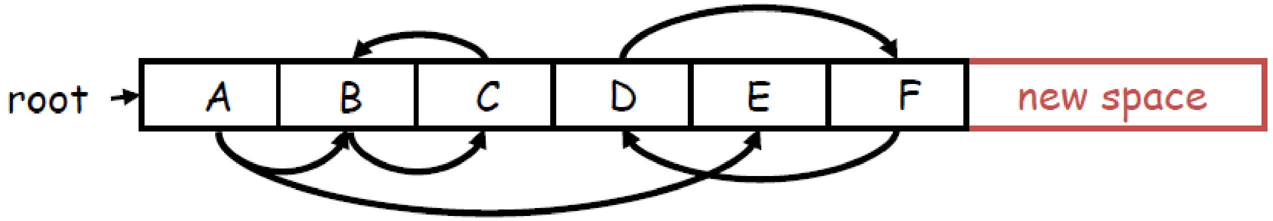
- Since scan caught up with alloc we are done
- Swap the role of the spaces and resume the program



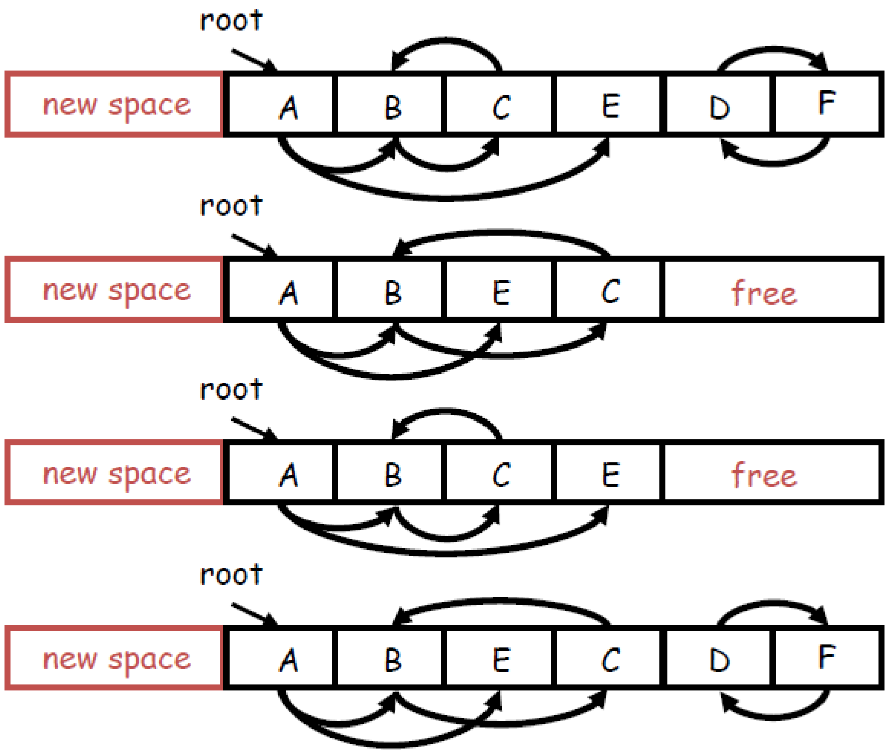
Stop and Copy

```
while scan <> alloc do
    let O be the object at scan pointer
    for each pointer p contained in O do
        find O’ that p points to
        if O’ is without a forwarding pointer
            copy O’ to new space (update alloc pointer)
            set 1st word of old O’ to point to the new copy
            change p to point to the new copy of O’
        else
            set p in O equal to the forwarding pointer
        fi
    end for
    increment scan pointer to the next object
od
```

Stop and Copy: Quiz



Choose the correct final heap after stop and copy garbage collection.



Stop and Copy

- As with mark and sweep, we must be able to tell how large an object is when we scan it
  - and we must also know where the pointers are inside the object

- We must also copy any objects pointed to by the stack and update pointers in the stack
  - this can be an expensive operation

Stop and Copy

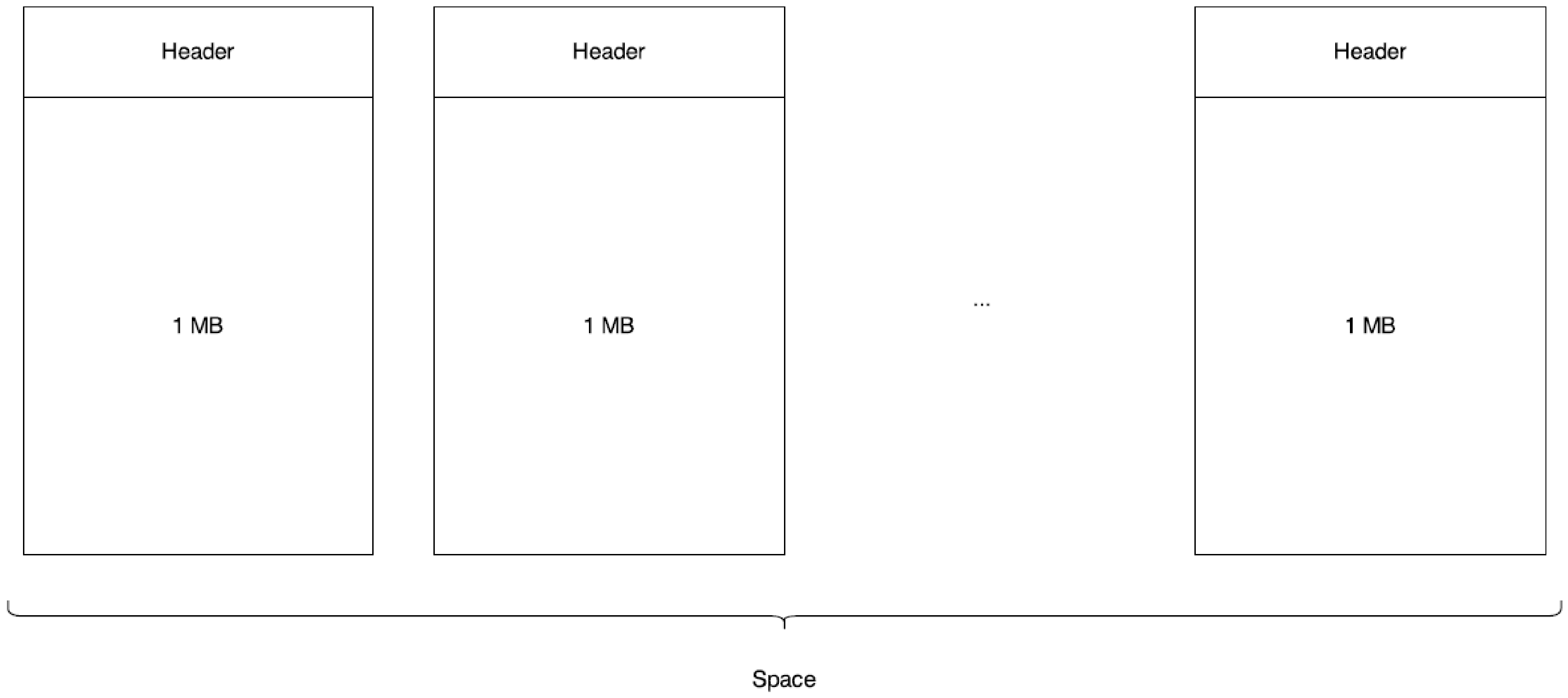
- Stop and copy is generally believed to be the fastest GC technique
- Allocation is very cheap
  - just increment the heap pointer
- Collection is relatively cheap
  - especially if there is a lot of garbage
  - only touch reachable objects
- But some languages do not allow copying
  - C, C++

V8’s GC

V8: Heap Organization

- New-space
  - Most objects are allocated here. New-space is small and is designed to be garbage collected very quickly, independent of other spaces.
- Old-pointer-space
  - Contains most objects which may have pointers to other objects. Most objects are moved here after surviving in new-space for a while.
- Old-data-space
  - Contains objects which just contain raw data (no pointers to other objects). Strings, boxed numbers, and arrays of unboxed doubles are moved here after surviving in new-space for a while.
- Large-object-space
  - This space contains objects which are larger than the size limits of other spaces. Each object gets its own mmap'd region of memory. Large objects are never moved by the garbage collector.
- Code-space
  - Code objects, which contain JITed instructions, are allocated here. This is the only space with executable memory (although Codes may be allocated in large-object-space, and those are executable, too).
- Cell-space, property-cell-space and map-space
  - These spaces contain Cells, PropertyCells, and Maps, respectively. Each of these spaces contains objects which are all the same size and has some constraints on what kind of objects they point to, which simplifies collection.

V8: Space Structure



V8’s GC: Discovering Pointers

- Distinguishing pointers and data on the heap is the first problem any garbage collector needs to solve.

- V8 uses tagged pointers
  - With this approach, we reserve a bit at the end of each word to indicate whether it is pointer or data. This approach requires limited compiler support, but it's simple to implement while being fairly efficient. V8 takes this approach. Some statically typed languages, such as OCaml, also take this approach.

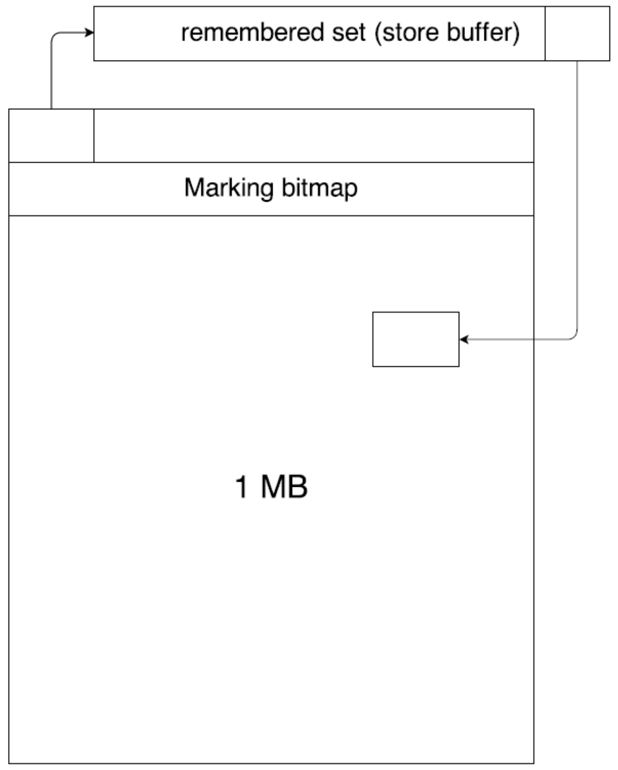
V8's GC: Generational Collection

- Objects tend to die young: most objects have a very short lifetime, while a small minority of objects tend to live much longer

- V8 divides the heap into two generations:
  - New-space – garbage collected by stop and copy algorithm (scavenge – minor garbage collection cycle)
  - Old-space – garbage collected by mark-sweep or mark-compact algorithm

V8’s GC: Write barriers

- What if a live object in old-space contains the only pointer to an object in new-space?
- Whenever a pointer to an object in new-space is written to a field of an object in old-space, we record the location of that field in the store buffer. In order to accomplish this, after most stores, we execute a bit of code called a write barrier, which detects and records these pointers.



V8’s GC: Marking

- There are three marking states for objects
  - white - it has not yet been discovered by the garbage collector.
  - grey - it has been discovered by the garbage collector, but not all of its neighbors have been processed yet.
  - black - it has been discovered, and all of its neighbors have been fully processed.

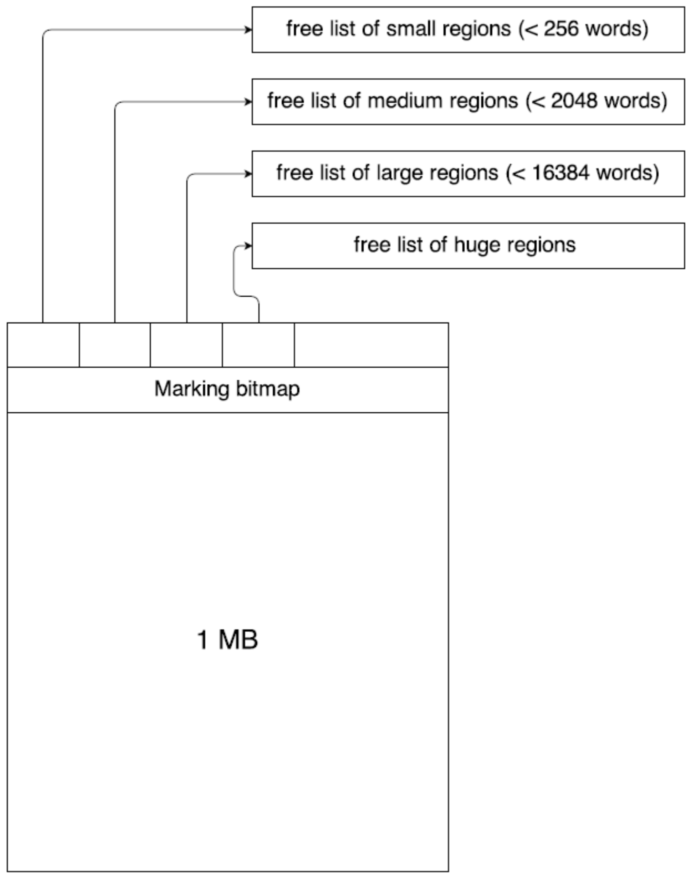
V8’s GC: Marking

- The marking algorithm is essentially a depth-first-search
- At the beginning of the marking cycle, the marking bitmap is clear, and all objects are white.
- Objects reachable from the roots are colored grey and pushed onto the marking deque.
- At each step, the GC pops an object from the deque, marks it black, marks neighboring white objects as grey, and pushes them onto the deque.
- The algorithm terminates when the deque is empty and all discovered objects have been marked black.

```
markingDeque = []
overflow = false

def markHeap():
  for root in roots:
    mark(root)

  do:
    if overflow:
      overflow = false
      refillMarkingDeque()

    while !markingDeque.isEmpty():
      obj = markingDeque.pop()
      setMarkBits(obj, BLACK)
      for neighbor in neighbors(obj):
        mark(neighbor)
  while overflow


def mark(obj):
  if markBits(obj) == WHITE:
    setMarkBits(obj, GREY)
    if markingDeque.isFull():
      overflow = true
    else:
      markingDeque.push(obj)

def refillMarkingDeque():
  for each obj on heap:
    if markBits(obj) == GREY:
      markingDeque.push(obj)
      if markingDeque.isFull():
        overflow = true
        return
```

V8’s GC: Sweep/Compact

- Once marking is complete, we can reclaim memory by either sweeping or compacting. Both algorithms work at a page level
- The sweeping algorithm scans for contiguous ranges of dead objects, converts them to free spaces, and adds them to free lists.
- The compacting algorithm attempts to reduce actual memory usage by migrating objects from fragmented pages (containing a lot of small free spaces) to free spaces on other pages.


free list of small regions (< 256 words)
free list of medium regions (< 2048 words)
free list of large regions (< 16384 words)
free list of huge regions
Marking bitmap
1 MB

V8’s GC: Incremental marking

- Incremental marking allows the heap to be marked in a series of small pauses, on other order of 5-10 ms each (on mobile).
- The object graph can change - we need to watch out for is for pointers being created from black to white objects.
- Write barriers come to our rescue. Not only do they record old→new pointers, but they also detect black→white pointers.
- When such a pointer is detected, the black object is changed to grey and pushed back onto the marking deque.

V8’s GC: Lazy/parallel sweeping

- Rather than sweeping all pages at the same time, the garbage collector sweeps pages on an as-needed basis until all pages have been swept.

- Parallel sweeping: since the main execution thread won't touch dead objects, pages can be swept by separate threads with a minimal amount of synchronization.