

Algorithms for Dynamic Memory Management

Lecture 3

Lecturer: Erez Petrank

Copying Garbage Collection

3 classical collectors:

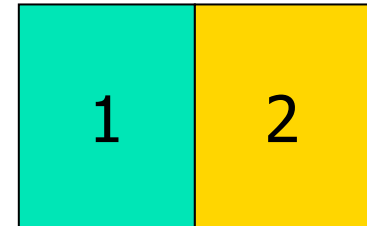
- Mark & Sweep (Compact)
- Copying
- Reference Counting

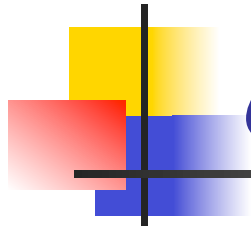
} Tracing



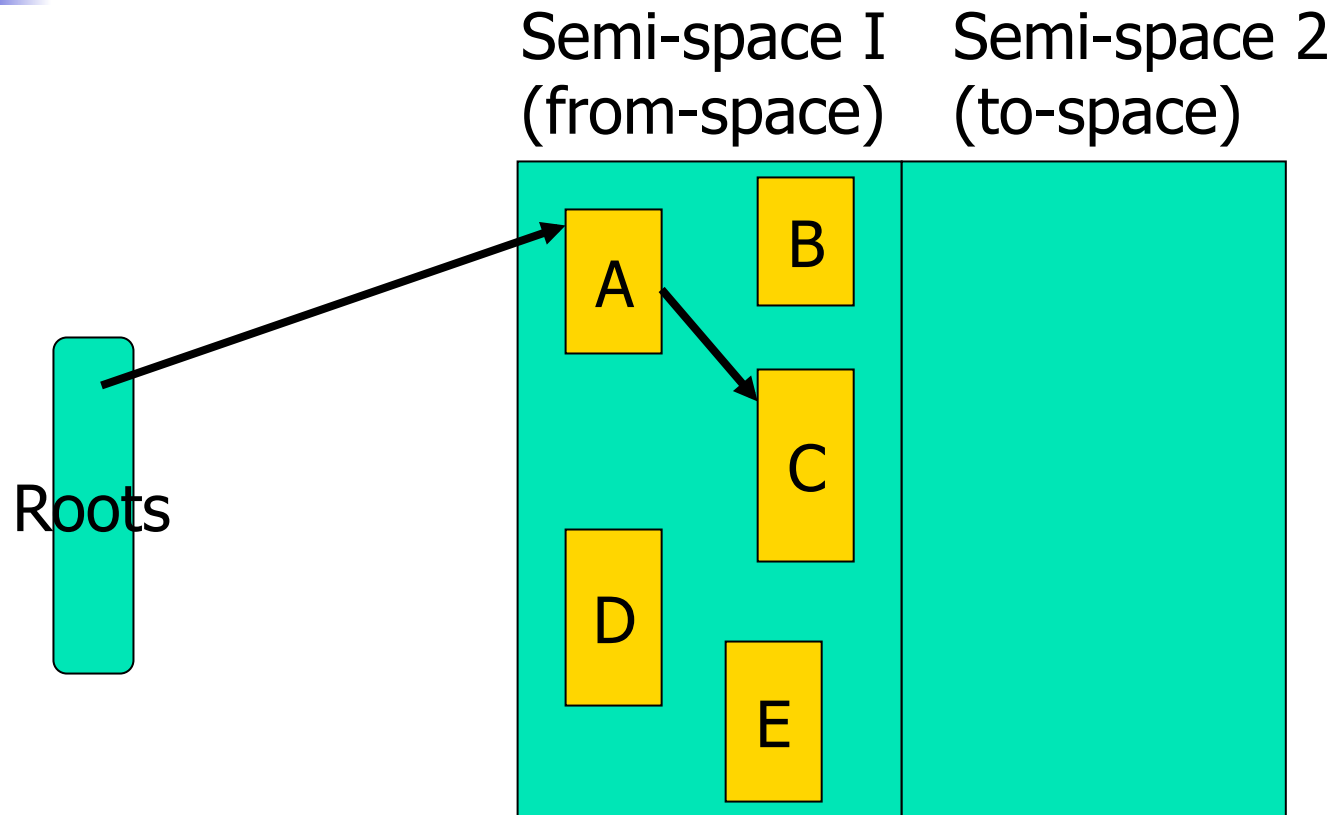
The Idea

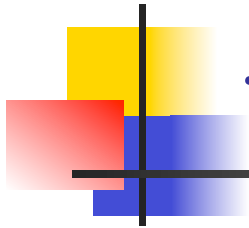
- **Heap** partitioned into two semi-spaces.
- Semi-space 1 takes all allocations.
- Semi-space 2 is reserved.
- During *GC*, the collector traces all reachable objects and copies them to the semi-space 2.
- After copying, activity goes to semi-space 2. Semi-space 1 is reserved till next collection.



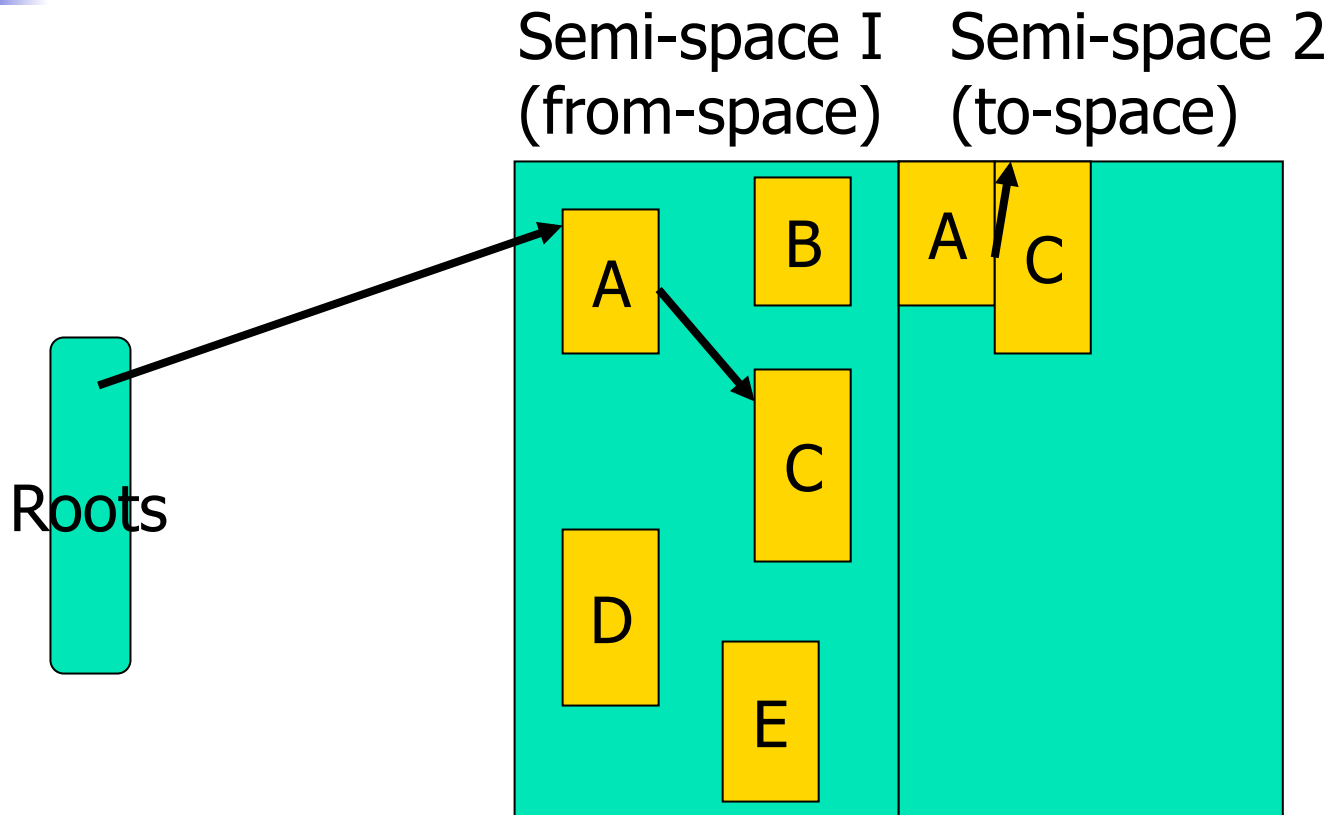


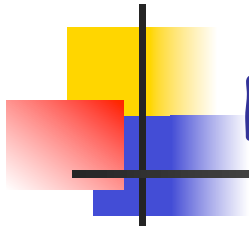
Copying garbage collection



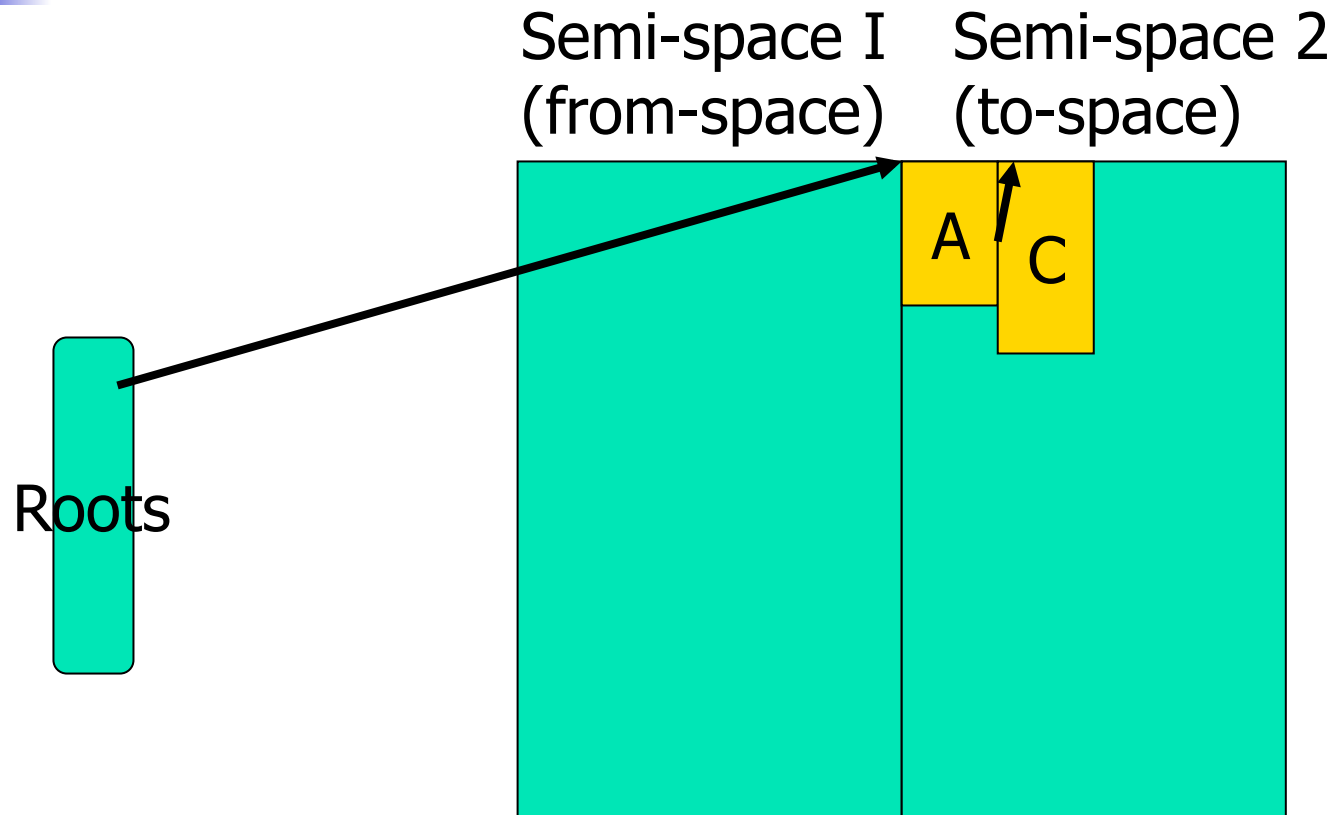


The collection copies...





Roots are updated; space I reclaimed.





Update Pointers

- Copied objects have wrong pointers!
- Keep “forwarding pointers” inside objects in from-space.
- Scan all to-space and fix pointers.
- Naive implementation employs a mark-stack. However, Cheney's algorithm does not.



The idea

- **Start** by copying the objects reachable directly from the roots into to-space.
- **Continue** by repeatedly copying objects reachable from the objects currently in to-space (and fixing the pointers).
- When all pointers in to-space point to to-space, we know that **all objects reachable from roots have been copied**.



Copying - Collection Cycle

- Stop all threads
- Flip sub-spaces; i.e., change roles of from-space and to-space
- Copy objects directly reachable from roots to to-space
- Scan to-space and fix from-space pointers.
 - Fix: copy object if not yet copied & fix pointer.
- Collection complete when all objects in to-space scanned.



Basic Action: Scan a pointer p

- **If** p points to to-space **then** do nothing.
- **Elseif** from-space referent has a forwarding ptr, **then** update pointer-slot.
- **Otherwise** (this is a reachable object that has not yet been copied) -
 - copy object into to-space,
 - record forwarding pointer,
 - update pointer-slot.



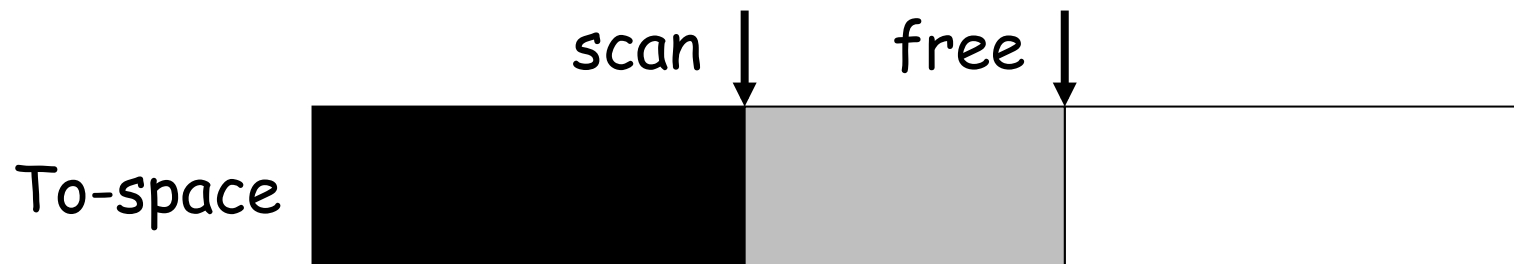
Overall Algorithm

- Flip sub-spaces
- Scan roots
- Scan all pointers in to-space.

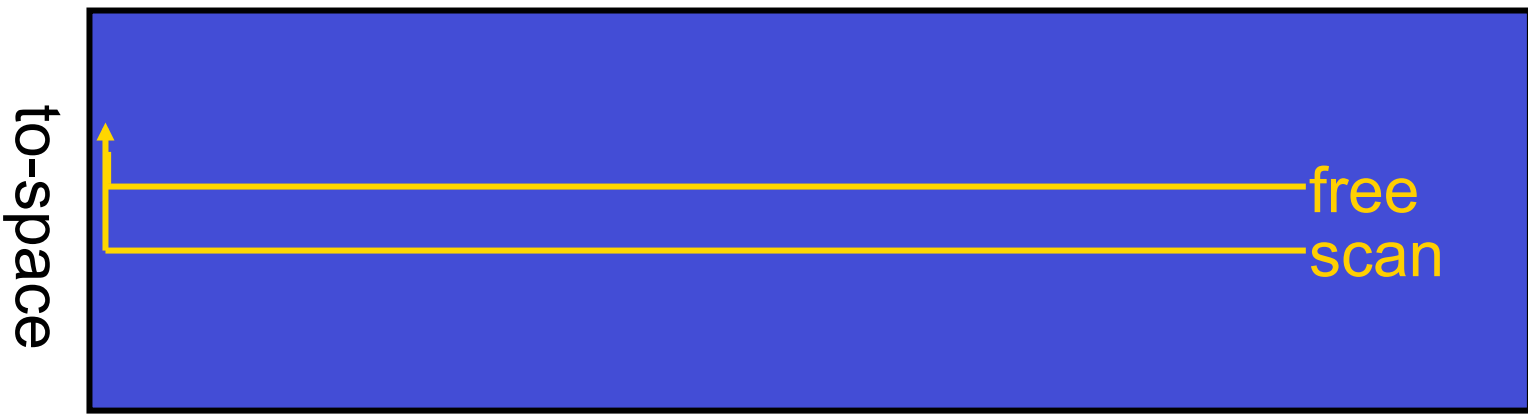
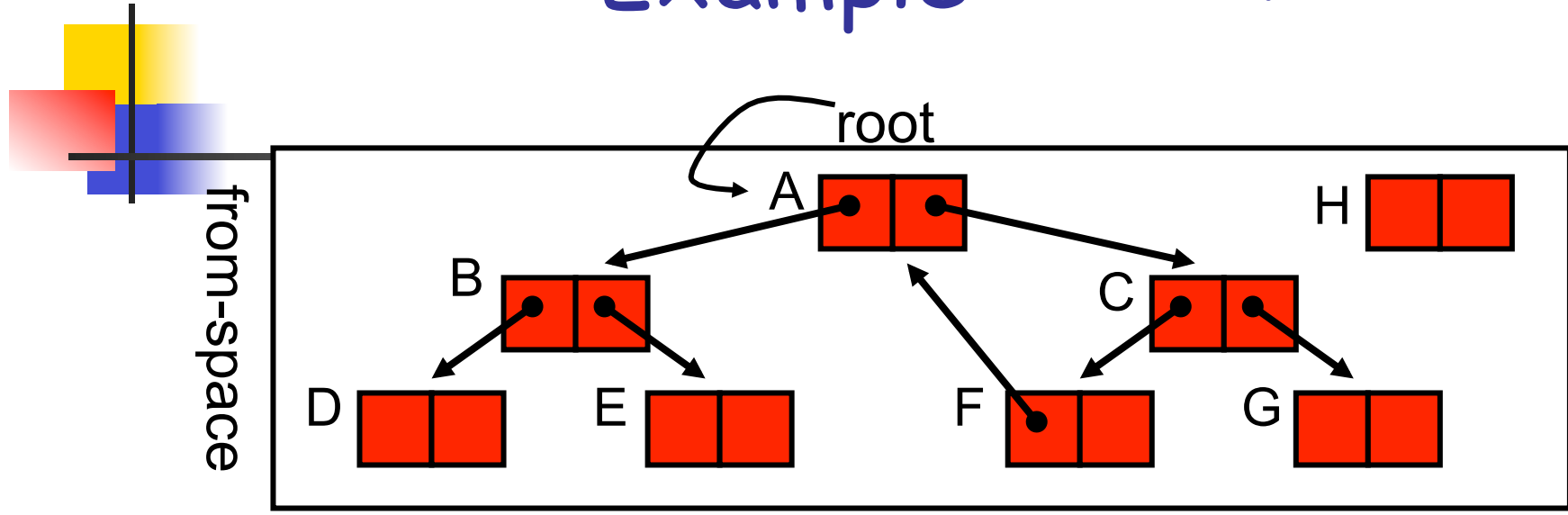
Cheney's Algorithm: Two pointers (and no stack)

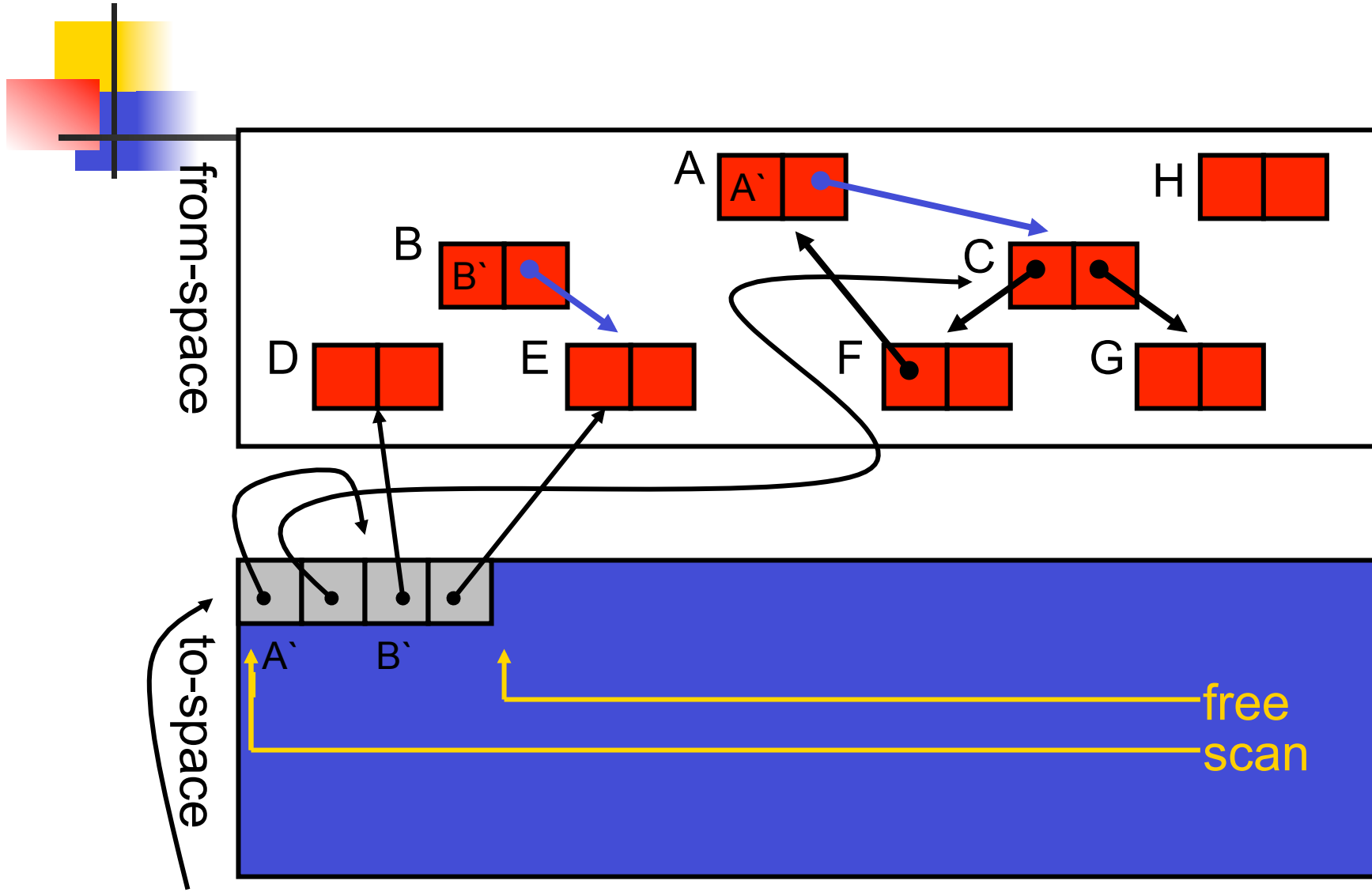
The idea to keep two pointers to monitor the collection progress is by Cheney:

- **Scan**: points beyond scanned objects.
- **Free**: points beyond copied (gray) objects and before the free space.

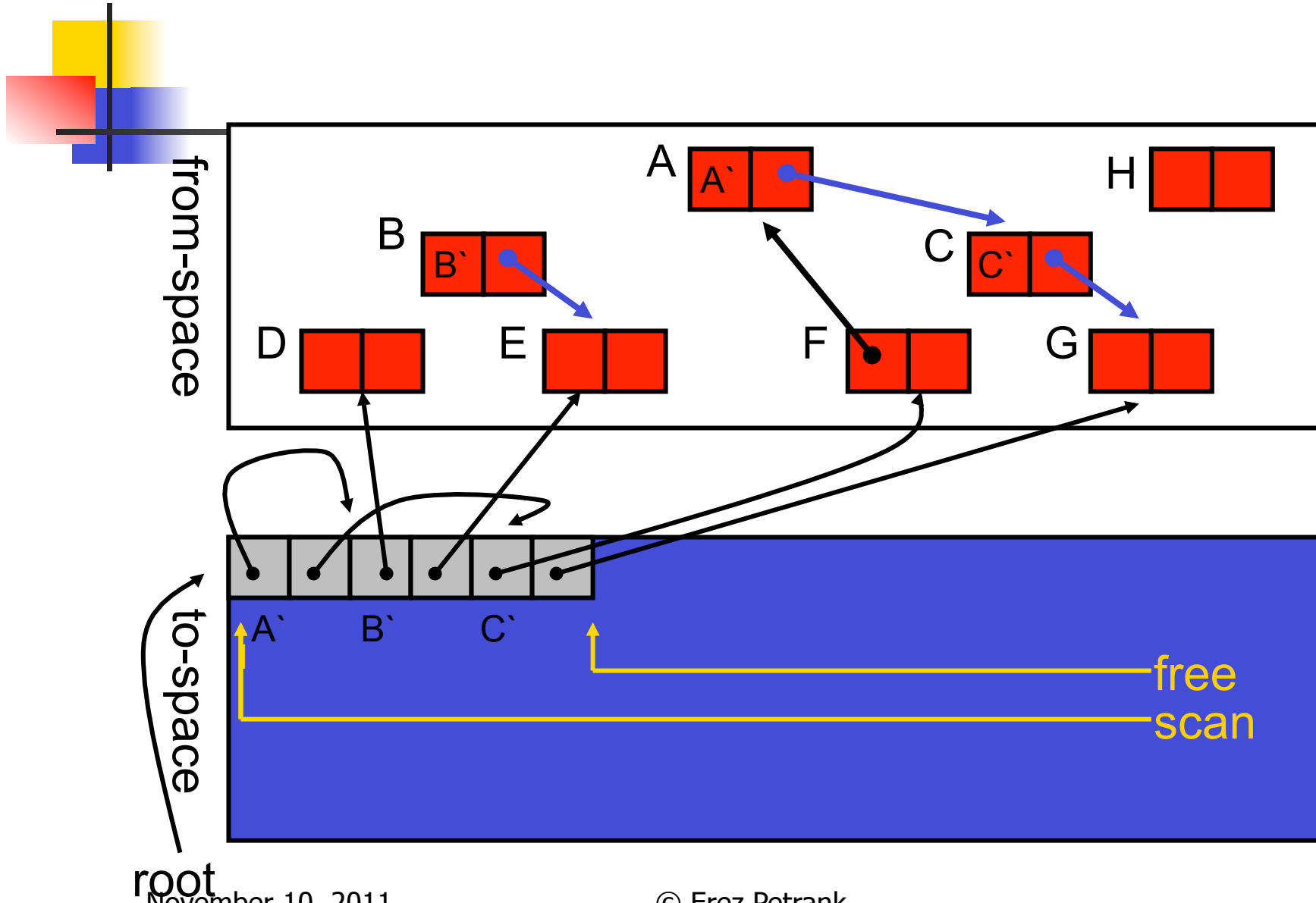


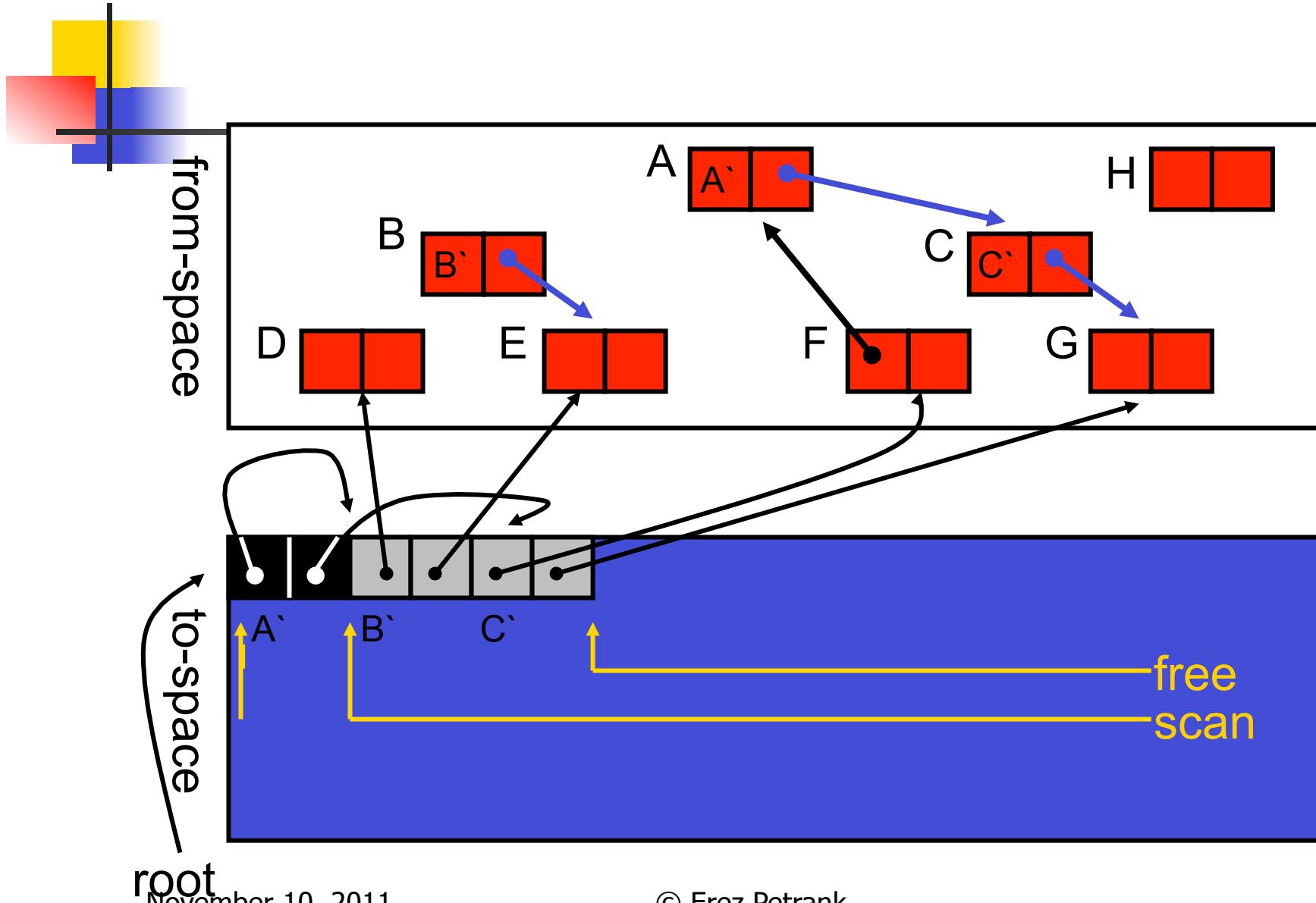
Example :



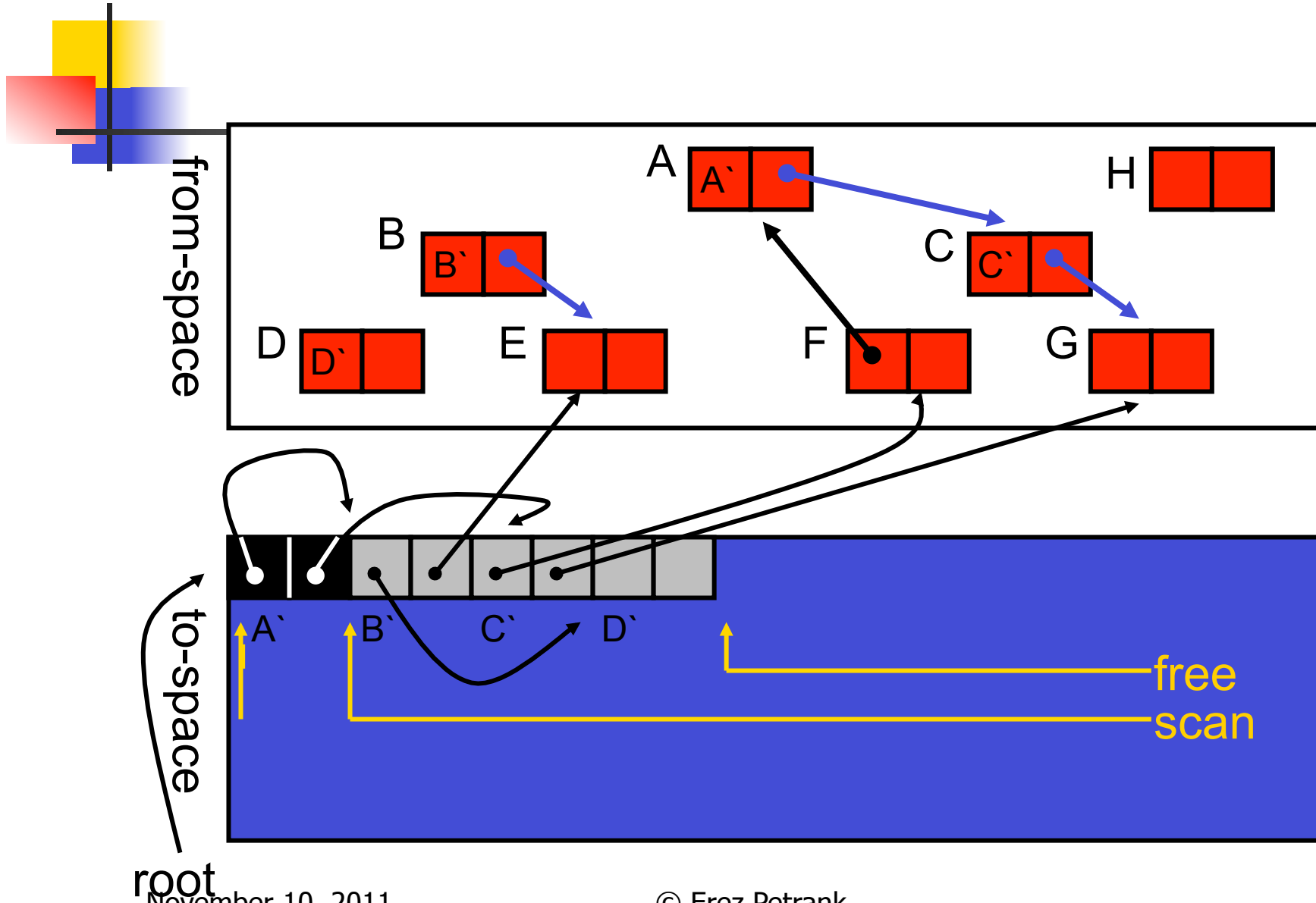


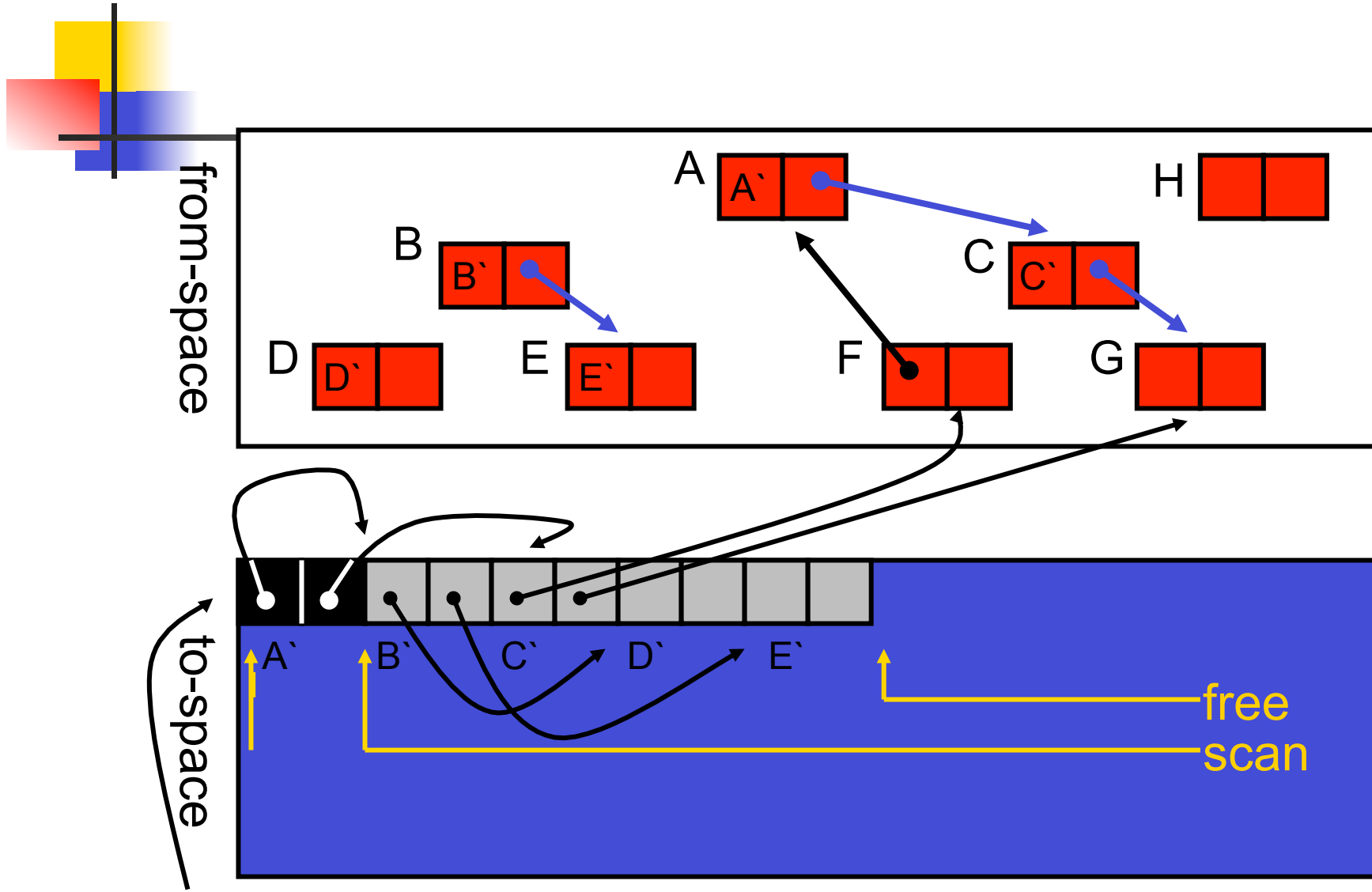
root

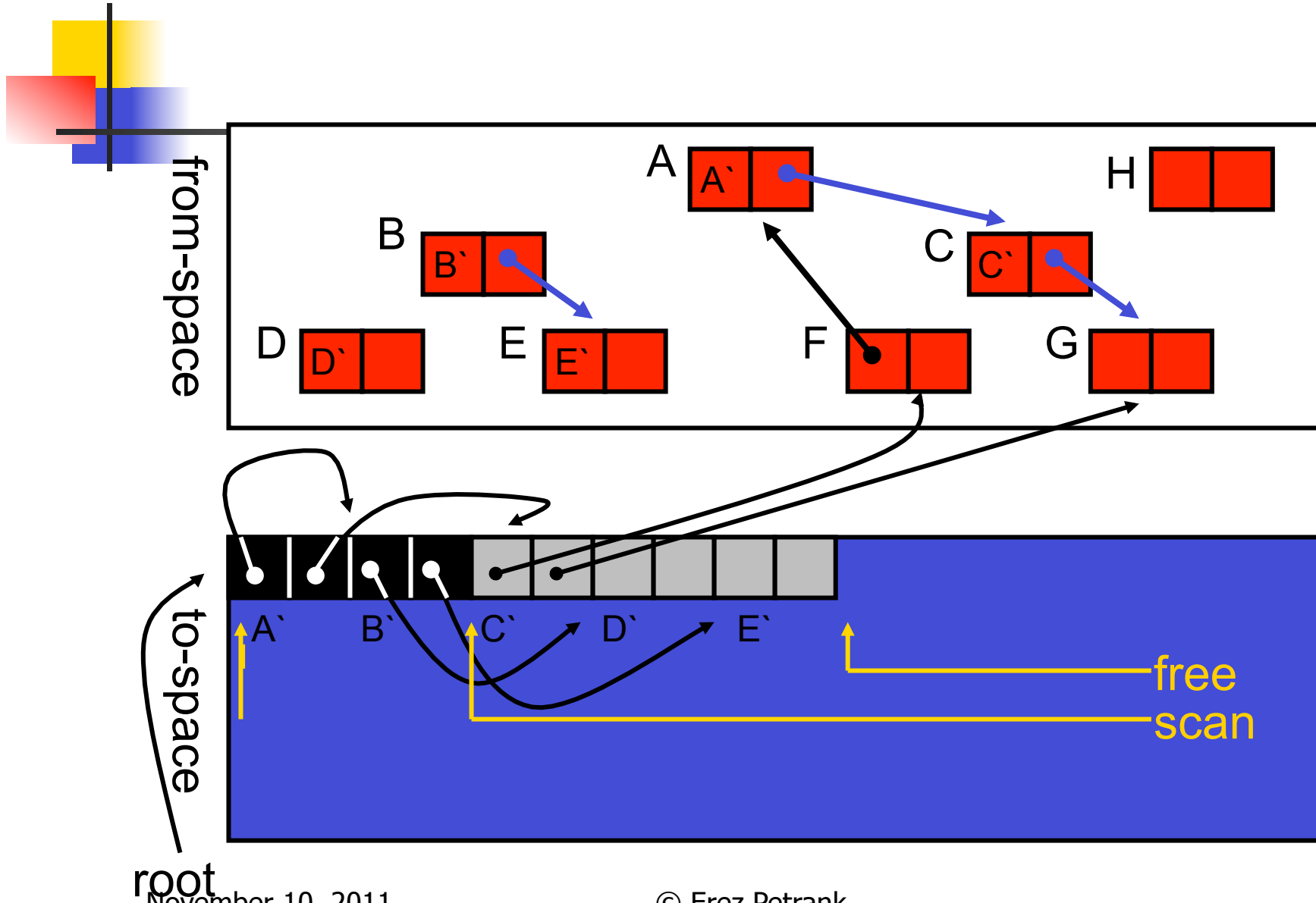


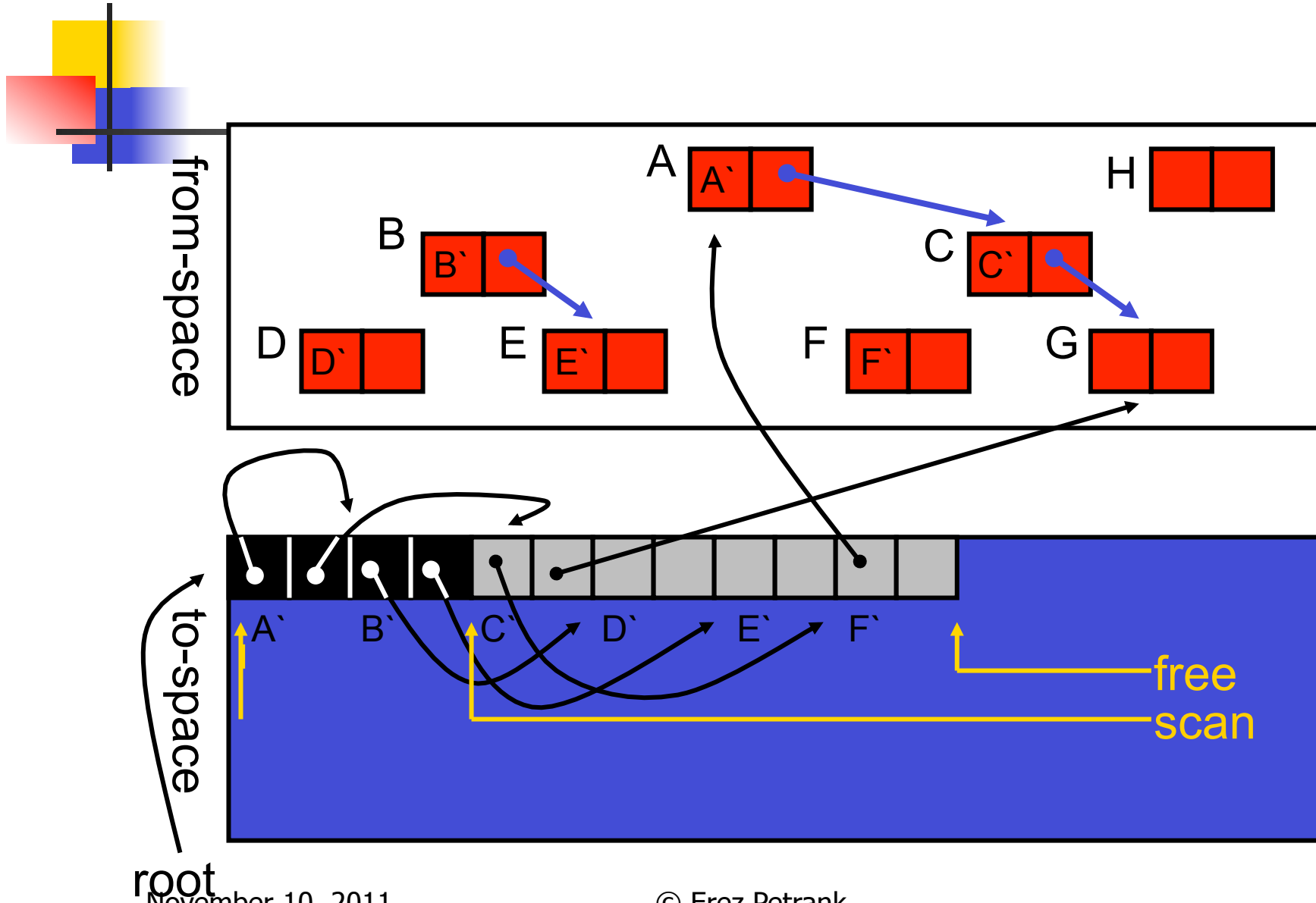


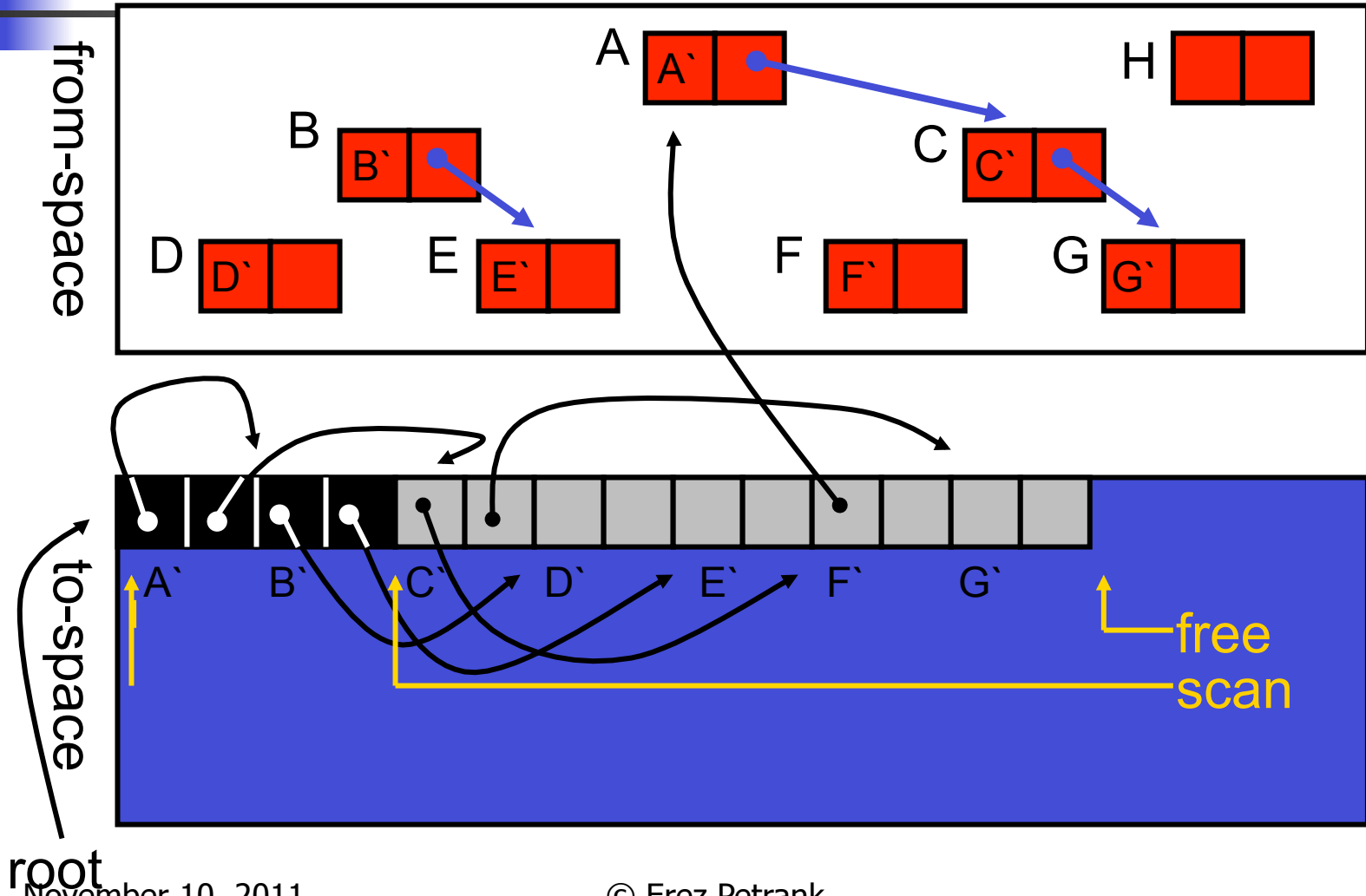
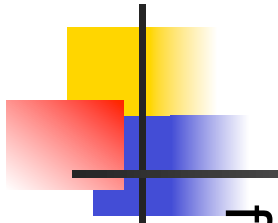
root

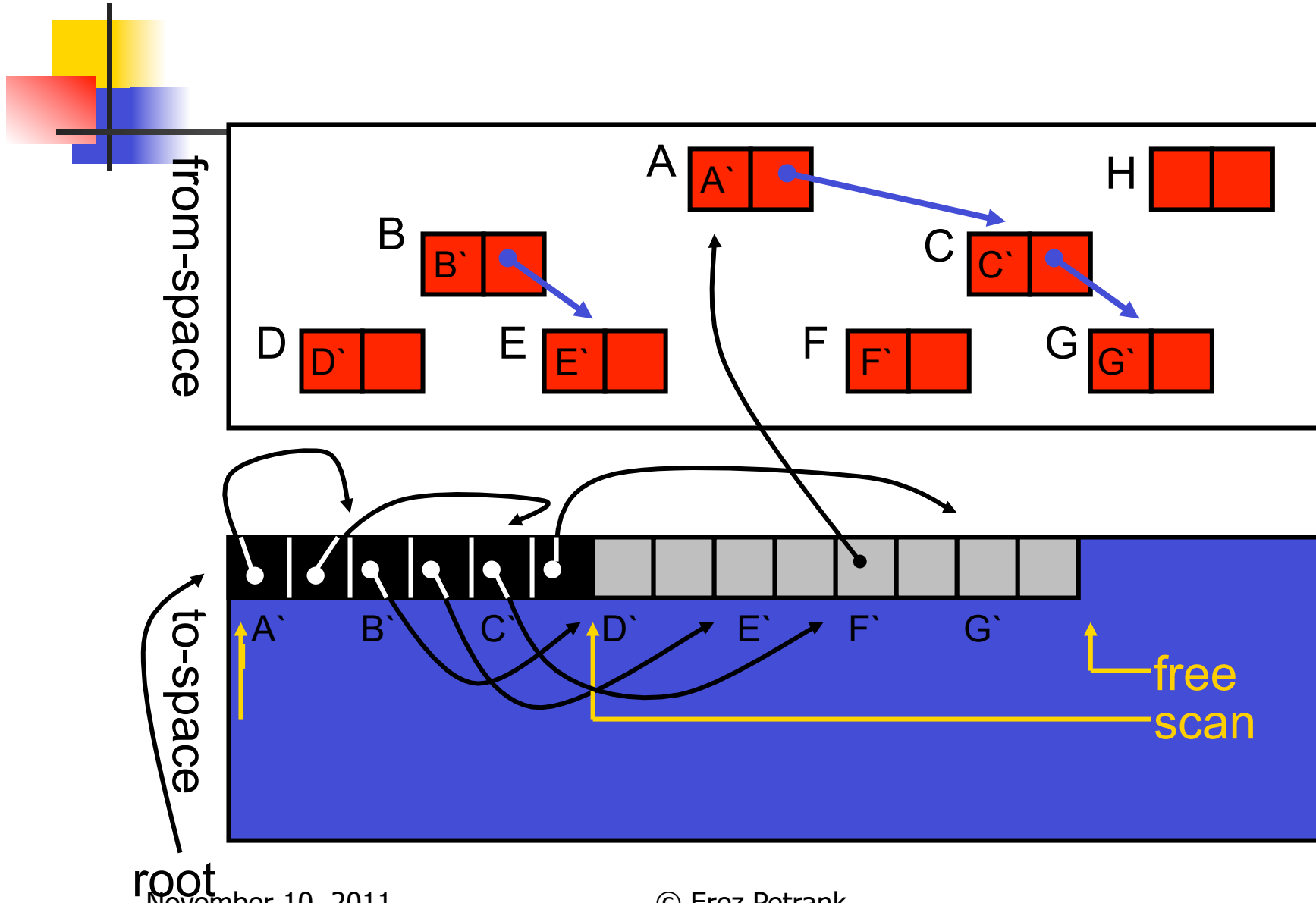


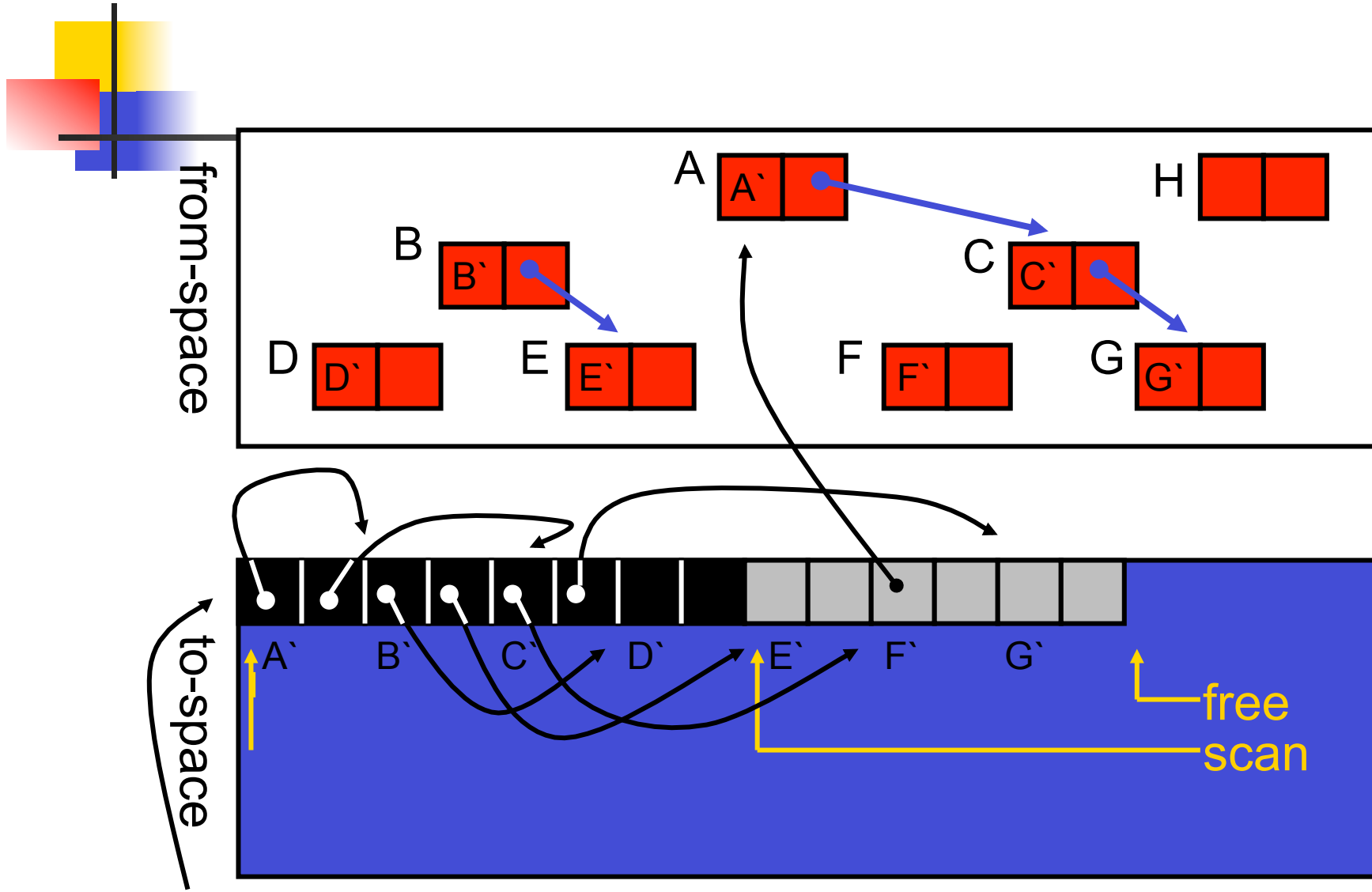


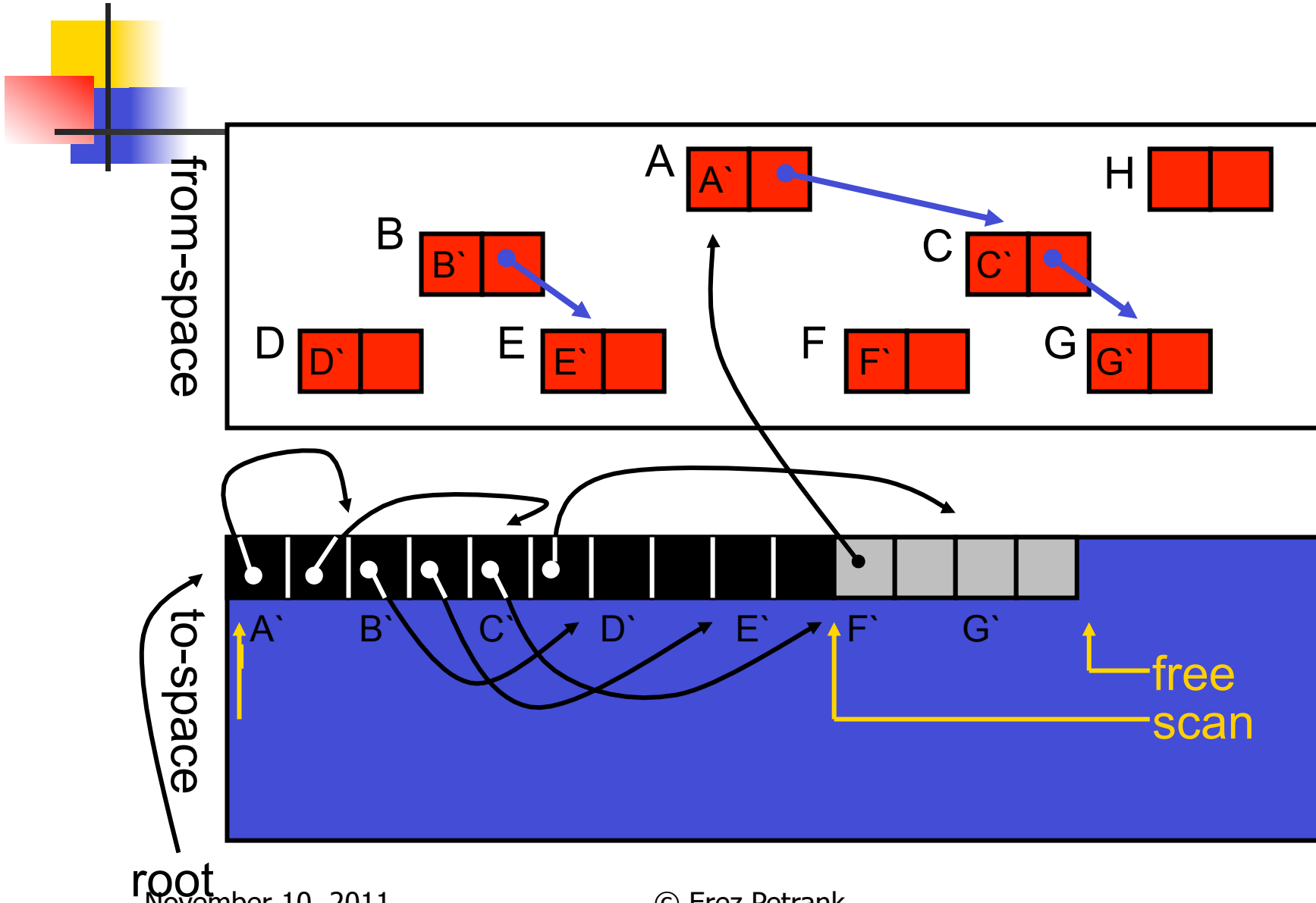




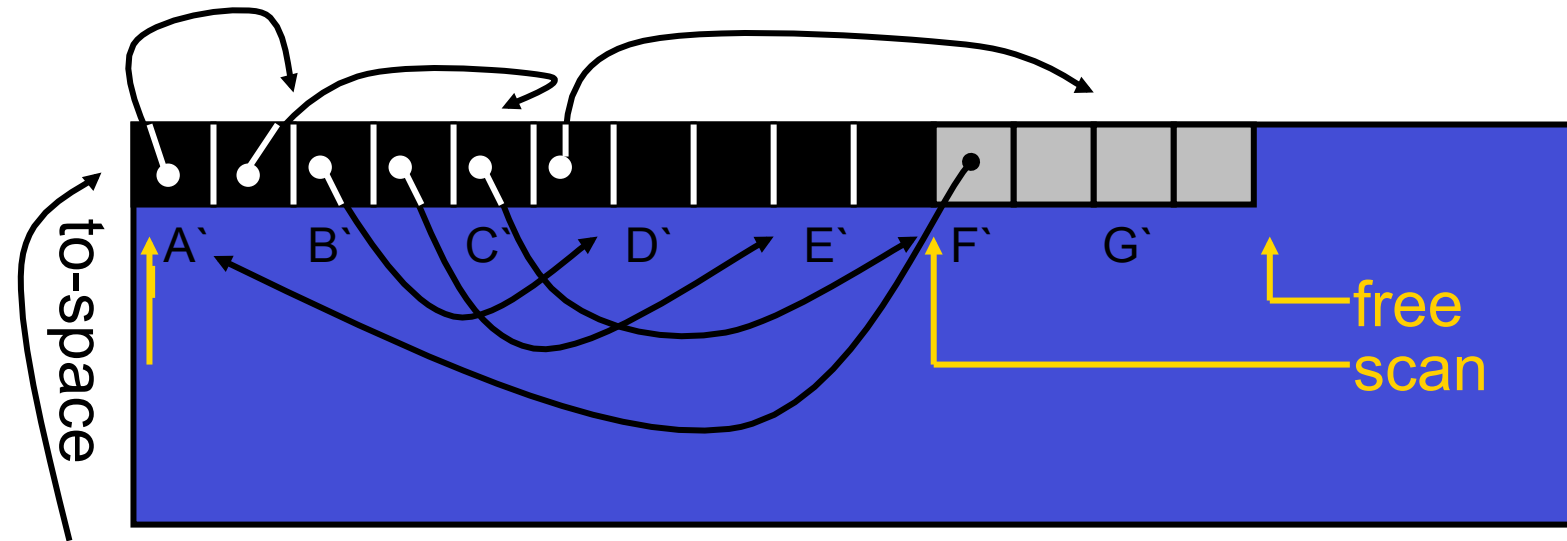
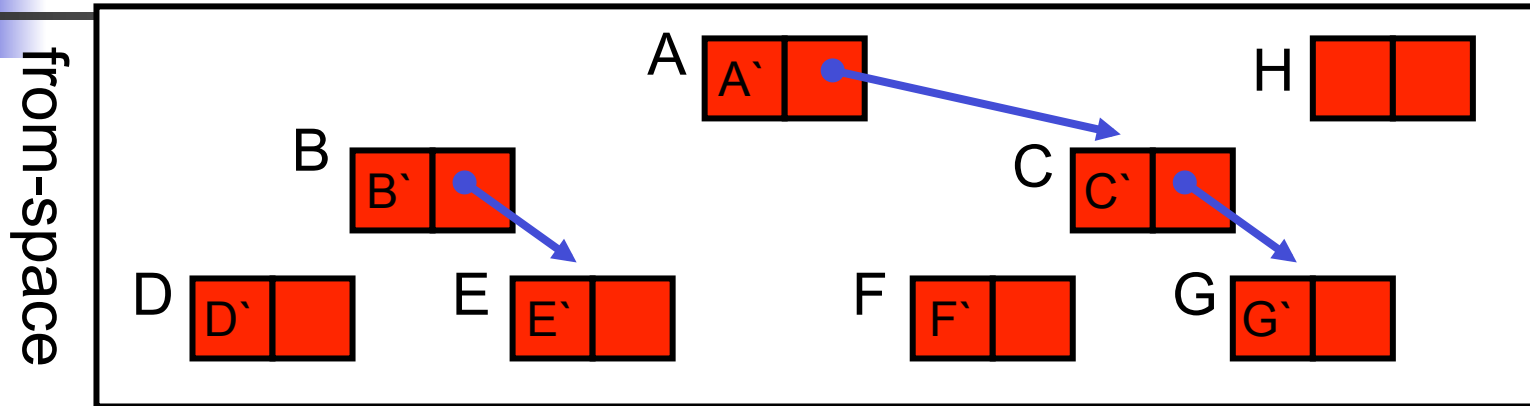
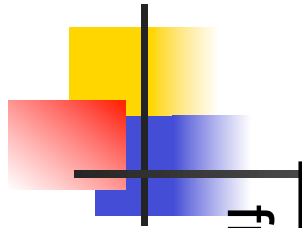




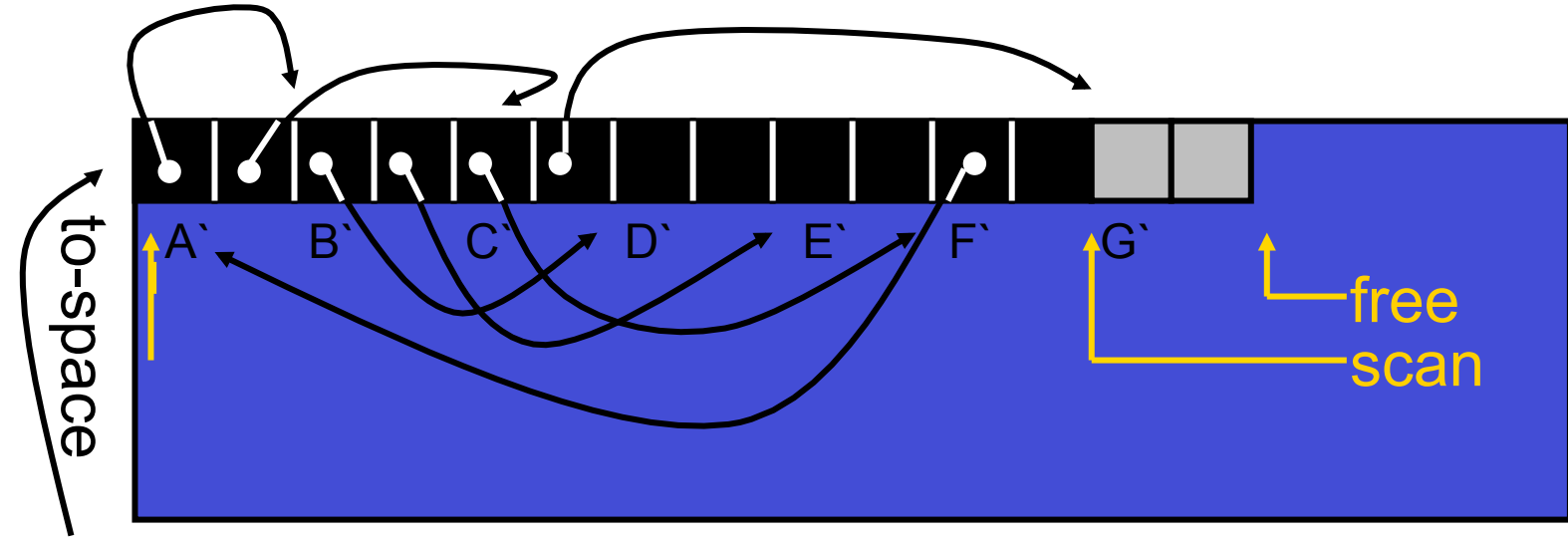
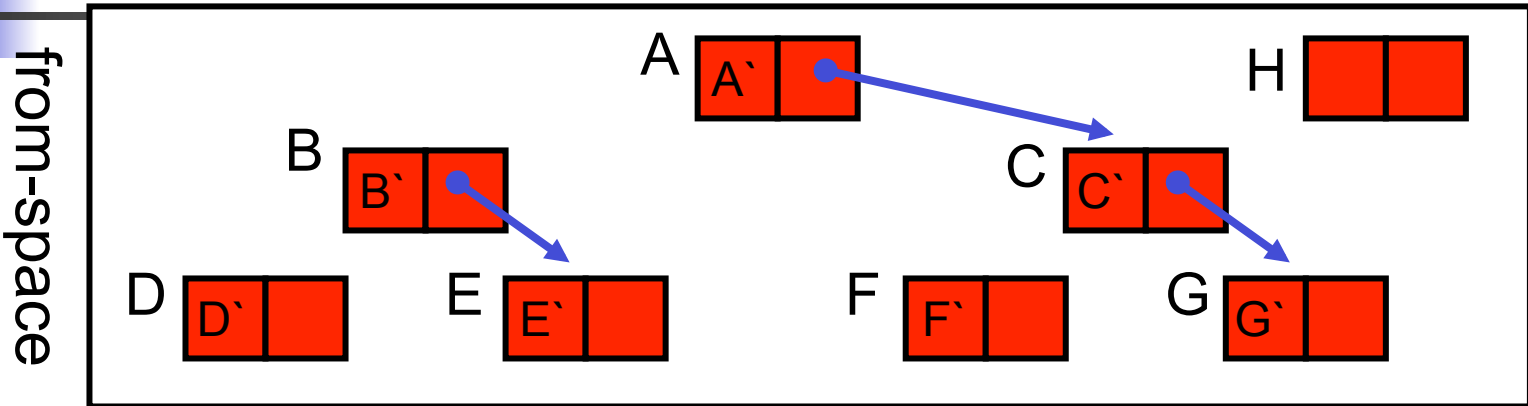
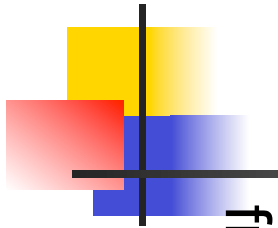




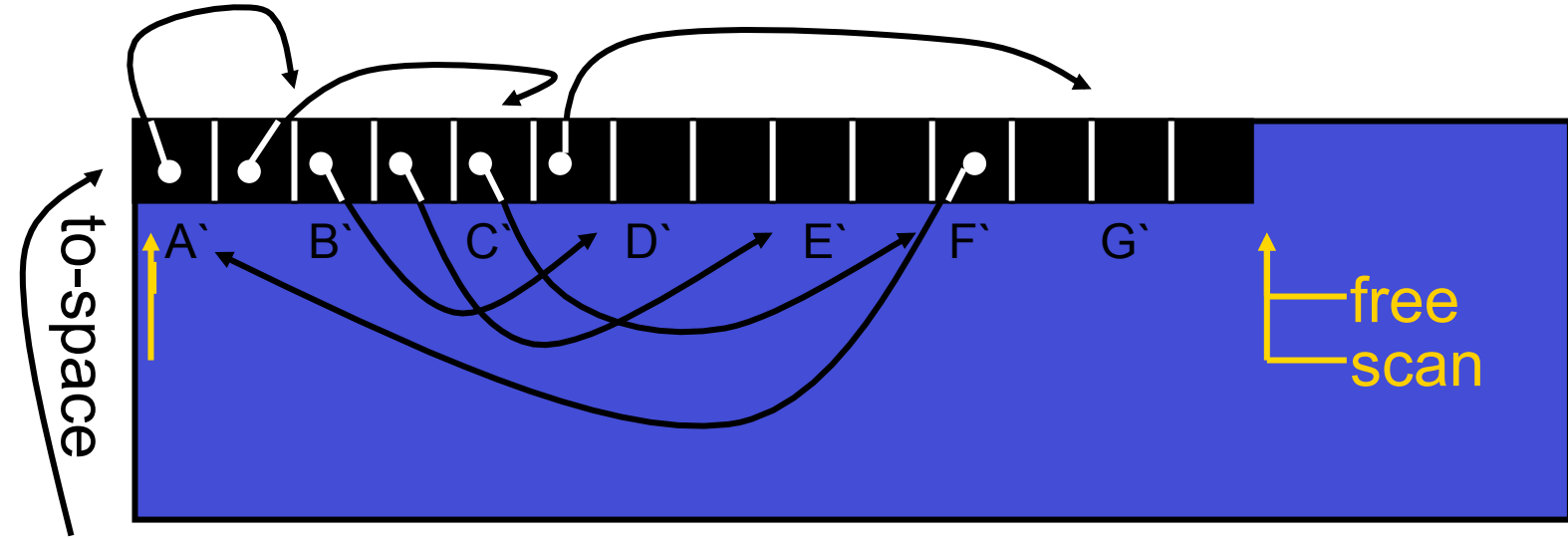
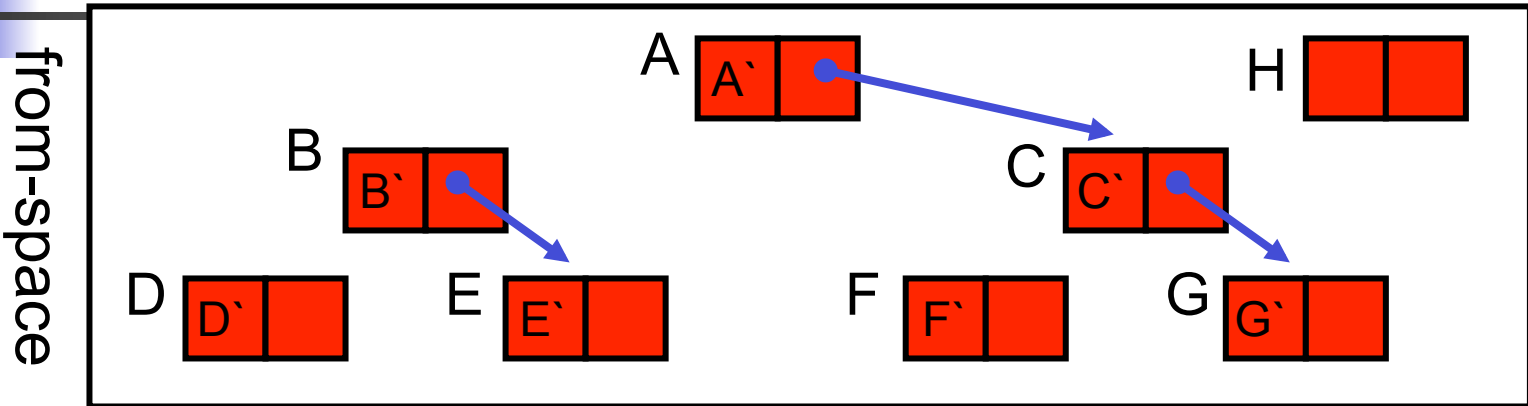
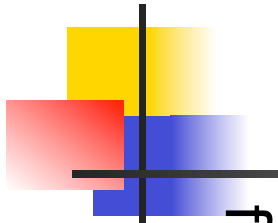
root



root



root



root



Properties



- **Space overhead:**
 - Half the heap reserved !
 - But: no auxiliary structures (no mark-stack, no bitmaps) and compaction for free: fragmentation space saved.
- **Complexity proportional to live data**
 - Recommended when survival rate is low.
 - Yet, cost of copying is larger than marking the object.
- Simple and **efficient allocation** ("linear").
- **Locality:**
 - Bound to page out all live memory during each collection



The algorithm (init)

Init() =

Tospace = Heap_bottom

space_size = Heap_size / 2

top_of_space = Tospace + space_size

Fromspace = top_of_space + 1

free = Tospace



The algorithm (Allocation)

```
New(n) =  
  if free + n > top_of_space  
    Collect()  
    if free + n > top_of_space  
      abort "Memory exhausted"  
  new-object = free  
  free = free + n  
  return (new-object)
```



The algorithm (collect)

```
collect() =  
    from-space, to-space =  
        to-space, from-space //swap  
    scan = free = Tospace  
    top_of_space = Tospace + space_size  
    for R in Roots  
        R = copy(R)  
    while scan < free  
        for P in children(scan)  
            *P = copy(P)  
        scan = scan + size (scan)
```



Handling a pointer

```
copy(P) =  
  if forwarded(P)  
    return forwarding_address(P)  
  else  
    addr = free  
    mem-copy(P, free)  
    free = free + size(P)  
    forwarding_address(P) = addr  
    return (addr)
```



Garbage collector efficiency

- R = Number of reachable objects, then Cheney copying requires cR operations.
- c depends on time to copy an object and average number of pointers per object.



Garbage collector efficiency

- R = Number of reachable objects, then Cheney copying requires cR operations.
- M = size of each semi-space, s = average size of an object. Thus, number of objects allocated between gc's (& reclaimed during) is $M/s - R$.
- g = CPU cost of GC per object reclaimed:

$$g = \frac{cR}{M/s - R} = \frac{c}{M/(sR) - 1}$$



Garbage collector efficiency

- Recall: R = number of reachable objects, M = size of each semi-space, s = average size of an object, g = CPU cost of GC per object reclaimed.
- R & s are application dependent.
- M may be determined by the system
- g , the cost, can be made arbitrarily small by increasing M . [Appel]: with sufficient memory (14 times the size of live space), it is cheaper to garbage collect than to free explicitly

$$g = \frac{cR}{M/s - R} = \frac{c}{M/(sR) - 1}$$



Example:

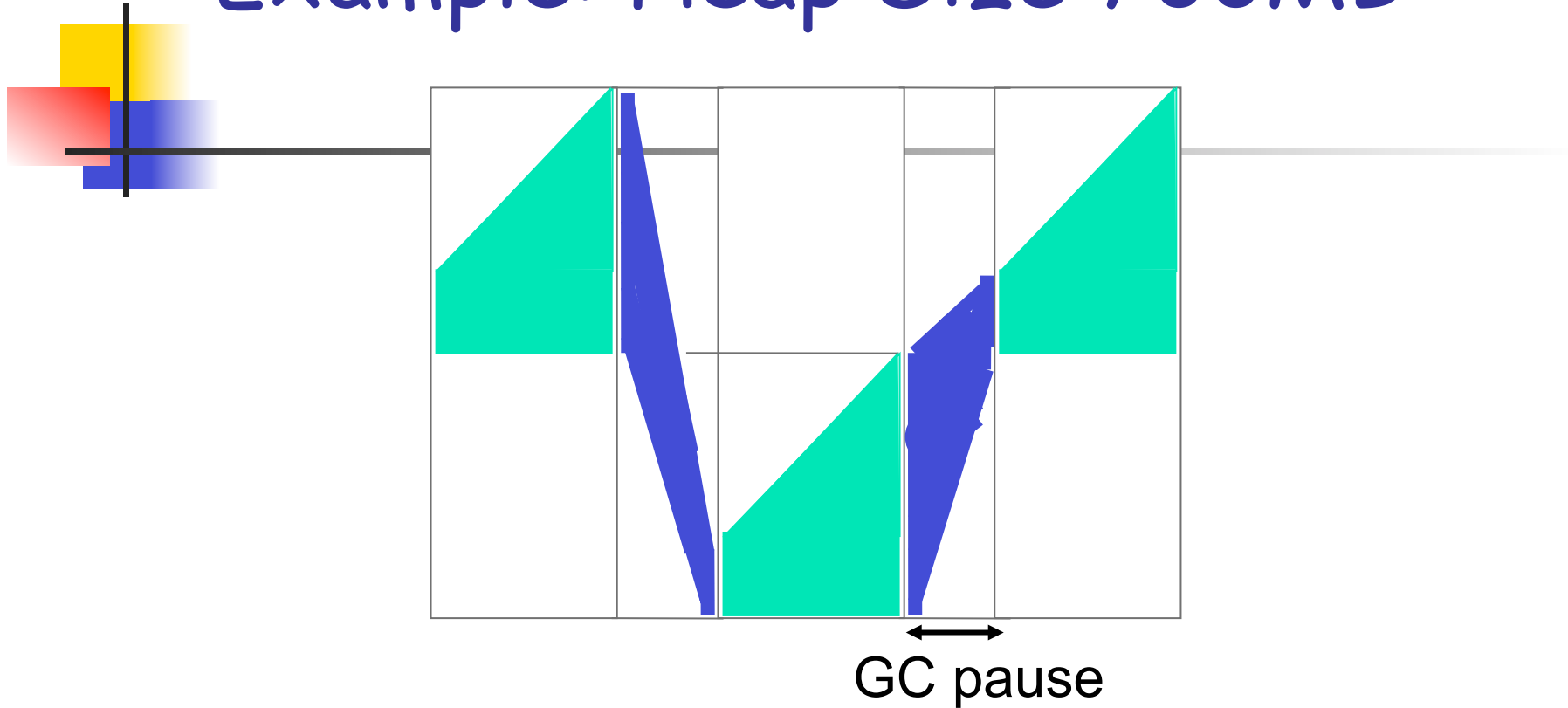
- Imagine that a program is run twice, once with semi-spaces of **350MB** and then with semi-spaces of **700MB**.
- Properties:
 - Suppose the amount of active memory used ($s \cdot R$) is **100MB** (and is stable).
 - The program allocates **1800MB** in total.

Example: Heap Size 350MB



To complete the run with the smaller heap, program must collect 6 times, copying $6 \cdot 100 = 600\text{MB}$ of data.

Example: Heap Size 700MB



To complete, the run with the larger heap must garbage collect only twice, copying $2 \times 100 = 200\text{MB}$ of data.



Historical review

- [Minsky 63] garbage collector for Lisp 1.5: secondary tape storage for other space.
- [Bobrow-Murphy 67] combined mark-sweep as the primary method *GC* with a variant of Minsky's copying collector to compact the heap.
- [Fenichel-Yochelson 69] copying algorithm with recursion for Lisp.
- [Cheney 70] efficient iterative copying garbage collection with the scan and free pointers.
- Lots of other implementations since then.



Improvements and Ideas

- Large objects
- Incremental compaction
- Mark & Copy



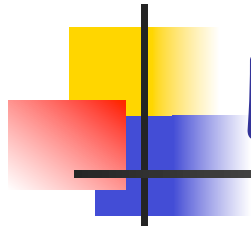
Large & Long-Lived Objects

- Copying GC copies surviving objects from one semi-space to the other.
- Good case: short lived small objects.
- Bad case: long lived large objects.
- Ideas:
 - Put large objects in a separate region collected by mark & sweep.
 - Put long lived objects in a separate region. Either collect infrequently or use mark & sweep there.
 - Main point: divide the heap into separately managed regions.

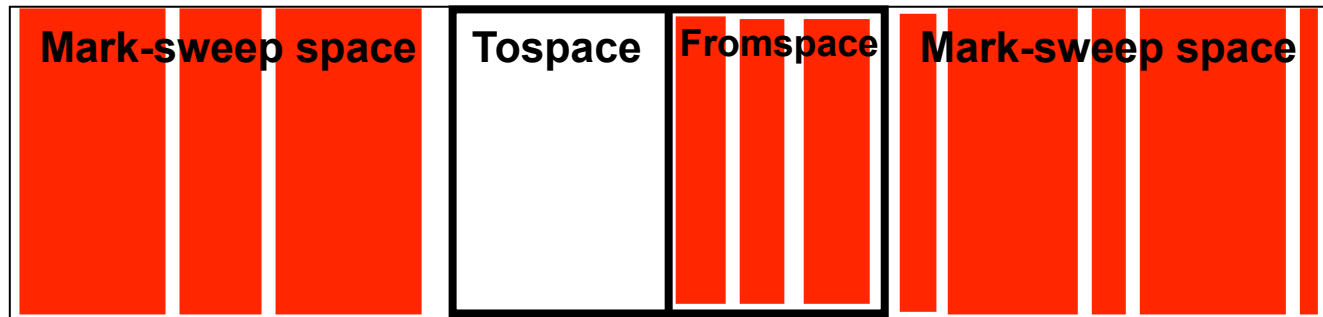


Incremental Compaction

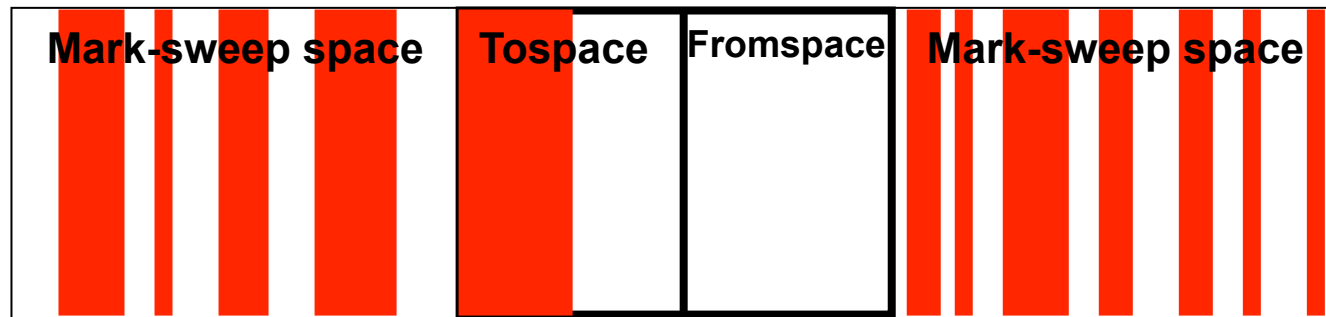
- Compaction is time consuming, whereas copying is space consuming.
- [Lang and Dupont 87]: let's use copying to obtain **incremental** compaction.
- Divide the heap into $n+1$ equally sized areas. One area is unused and n areas are used for allocation.
- During collection, two of these areas are treated as a pair of semi-spaces and managed by copying collector while the rest of the heap is mark-swept.
- The pair of segments chosen as semi-spaces rotates through the address space at each collection, incrementally compacting the heap.



Lang & Dupont - Heap Partition



Before collection



After collection



The L&D Collector

- Segment i free at start (current **to-space**).
- Segment $i+1$ serves as from-space (and to-space of the next collection).
- It all happens simultaneously:
 - When visiting the descendants of an object:
 - If descendant **not in from-space** then **mark** it.
 - Otherwise, (**in from-space**) "scan":
 - If not copied yet - copy to to-space, leave a forwarding pointer, and update the pointer.
 - If copied - update pointer using forwarding pointer.
- Finally, sweep all segments except $i, i+1$.



Properties

- A small space overhead (depending on n)
- A small payment in time (extra "if" in the marking procedure, plus copying is a bit more expensive.)
- No extra passes are required.
- After n collections the whole heap has been compacted incrementally (speed depends on n).
- Compaction quality is low (depending on n).
- Allocation cannot use "bump pointer" style.

Lang-Dupont Collector Versus Traditional

Compared to mark-compact

- A small space overhead.
- Much more efficient.
- Compaction is slow and its quality is low (depending on n).

Compared to copying

- Much more space-efficient (depending on n).
- Collection efficiency similar to mark-sweep.
- Allocation cannot use "bump pointer" style.



MC (Mark-Copy) and MC²

- Narendran Sachindran, Eliot Moss, Emery Berger



Virtual Memory Reminder

- A process has a large **virtual** (memory) address space.
- This space is divided into virtual pages (typically 4KB)
- Some of the virtual pages are kept on physical memory while the rest reside on the hard disk.
- The system has a map from virtual to physical pages.
- Due to locality of computation, **page faults** (pages that are not available in the memory and must be retrieved from the disk) are seldom.



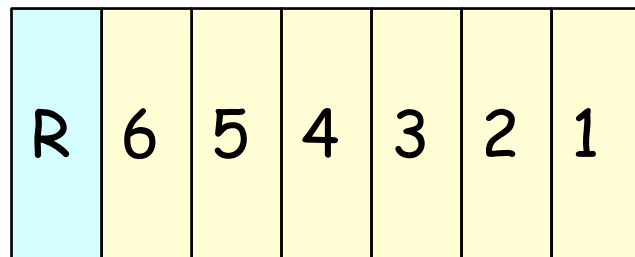
Virtual Memory Primitives

- A virtual page is *mapped* into a physical page when a space on that page is allocated.
- Virtual pages consume “real” resources (such as space on disk and memory) when mapped.
- A virtual page can be *released* or *unmapped* when the process does not need the data on it anymore.



MC (Mark-Copy) and MC²

- Sachindran, Moss, and Berger
- Problem: copying requires double space.
- Solution: add some work, but save space.
- Partition the heap H into n small sub-heaps.
- Keep only one extra space of size H/n .

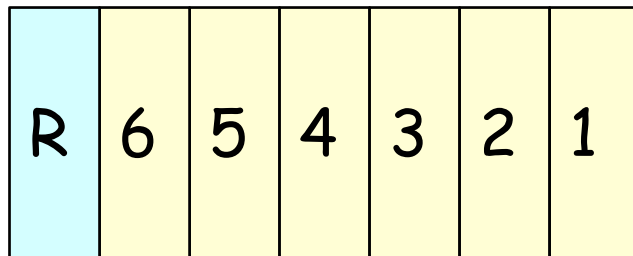


In the example, 6 areas, and an additional reserved 1/6 (wasted 1/7)



MC (Mark-Copy)

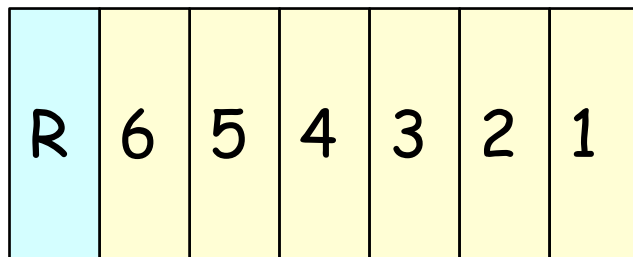
- Naively, when heap full, stop the program and:
 - Collect area 1 into area R,
 - Collect area 2 into area 1,
 - ...
 - Collect area n into area n-1.
- MC extends this simple scheme.





MC (Mark-Copy)

- Naively, when heap full, stop the program and:
 - Collect area i into area $i+1$...
- Problem 1: low-quality compaction,
- Problem 2: how do we know which object is alive in space i ?
- Problem 3: how do we update the pointers?



An Initial Marking Phase

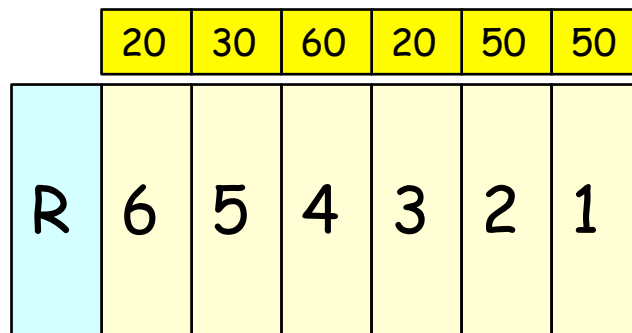
- Add a marking phase that goes over all live objects
 - Now we know which objects are alive.
 - Also, create *remembered sets*. One for each area recording pointers referencing it from higher areas (for pointer updates.)
 - Finally, sum up space in each area.

	20	70	60	20	50	50
R	6	5	4	3	2	1

Say, each area
can hold 100MB

Copy Passes

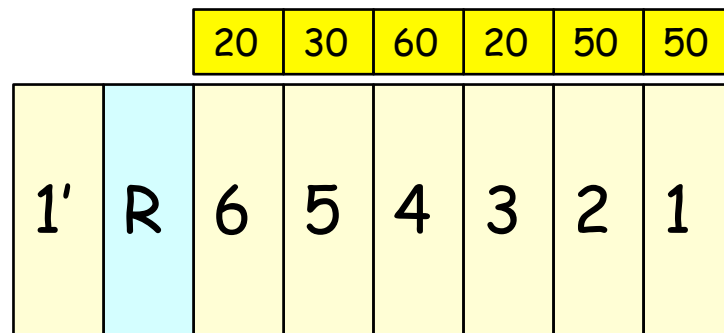
- Pass I: Compute How many spaces (1,2,...) can fit into R.
- Copy the live elements in these spaces into R using a copying collector, using roots + remembered set.
- “Unmap” areas 1 and 2; Map a new area 1’.



In the example,
Pass I: 1 & 2 into R;

Copy Passes

- Pass I: Compute How many spaces (1,2,...) can fit into R.
- Copy the live elements in these spaces into R using a copying collector, using roots + remembered set.
- "Unmap" areas 1 and 2; Map a new area 1'.
- Pass II: Compute the next spaces that can be copied into area 1', copy live objects, un-map them, map 2', ...



In the example,
Pass I: 1 & 2 into R;
Pass II: 3 & 4 into 1'
Pass III: 5 & 6 into 1'
and 2'.



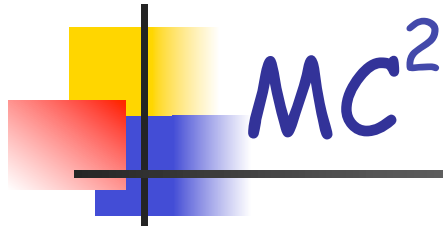
Tuning

- A larger number of areas means
 - Smaller space overhead
 - More overhead on remembered sets, and collection passes.



Properties

- Advantages of a copying collector:
 - Fast allocation, with good locality, heap compaction, touch only live objects.
 - Much smaller space overhead.
- Disadvantages:
 - An additional tracing phase (with mark-stack space overhead),
 - An additional remembered-set space and time overhead.
 - Several (extended) root scanning,



- Improvements:
 - Do not use virtual memory services,
 - Do not copy all objects in old space,
 - Reduce space overhead of remembered sets.

- We will not discuss:
 - Integrate with generations,
 - Increase incrementality.



Changes from MC

- Areas are maintained by the collector, each area has a logical number determining its order.
- High occupancy areas: objects are never copied from them. Instead, their logical number is made high.
- The marking phase creates the remembered sets.
- The collector computes groups of areas whose accumulated live set fits into a single empty area.
- (We ignore changes related to generations and incrementality.)



Integration with Generations

- The original papers put it all in a generational setting, to be discussed next week.
- The old generation is managed with MC (or MC^2).
- The trigger for a full collection: after promoting live young objects, remaining space is below the young generation size (ignoring the reserved area).



Discussion: Mark & Sweep vs. Copying

- Efficiency
 - Marking vs. copying
 - Sweeping
- Space usage
 - Wasted semi-space vs. mark-stack and mark-bits
- Fragmentation
- Caching

- No real conclusion!
Mark-and-sweep probably wins in popularity.
Both serve in a standard generational collector.



Summary

- Cheney's collector
- Large objects
- Incremental compaction
- Mark-Copy



Incremental Garbage Collection



Problem

- **Long pauses disturb the user.**
 - Real time cannot be guaranteed
 - Slow response time - lost clients
 - Sometimes even communication time-outs
- **Response time** may be bad for a long period after a pause: held transactions must wait for queue to be handled.
- An important measure for the collection: **length of pauses** (average, maximum, distribution).

Solution: incremental Collection

- partition the collection work to "increments" ..
- Baker's copying collector [1977]:
 - During each allocation do some GC work.

Stop the world:



Incremental:





Not so simple!

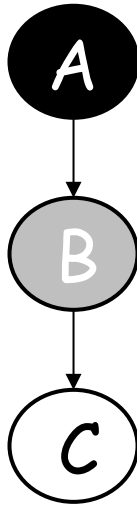
- The heap changes while we collect.
- For example,
 - we scan object B and before marking its children, the collector is stopped and the program resumes.
 - When the collector returns, the children may have been modified.

Incremental:

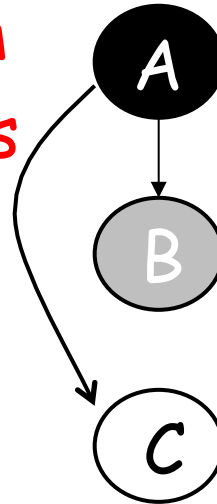


Example

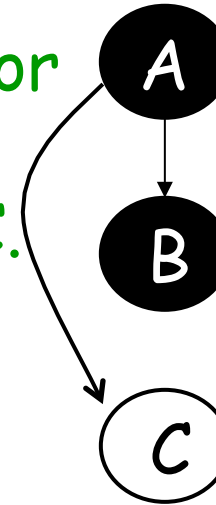
GC marks
A's
children
and
makes A
black.



Program
modifies
pointer.



Collector
fails to
trace C.



Time



Three-Color Abstraction

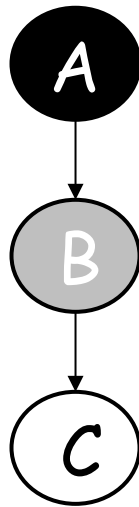
- Black - objects in to-space that have been scanned (their children have been copied to to-space).
- Gray - objects have been copied to to-space, but have not been fully scanned yet.
- White - objects that have not yet been copied into to-space.

(A similar abstraction for mark and sweep exists.)

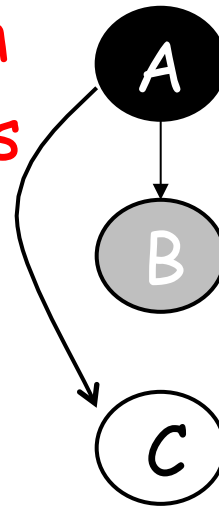


Problem: a pointer from black to white!

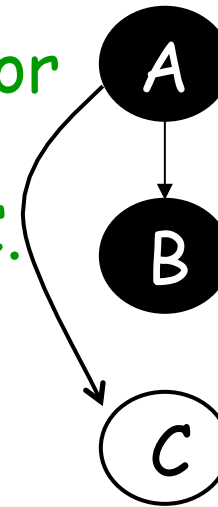
GC marks
A's
children
and
makes A
black.



Program
modifies
pointer.



Collector
fails to
trace C.



Time





Possible solutions

- “Aggressive” solution: do not allow black to white edges ([Baker], today).
- “Finer” solution: make sure that:
for each white object O there is always a chain of references starting at a gray object, continuing with white objects ending in O ([Dijkstra et al], later in this course).

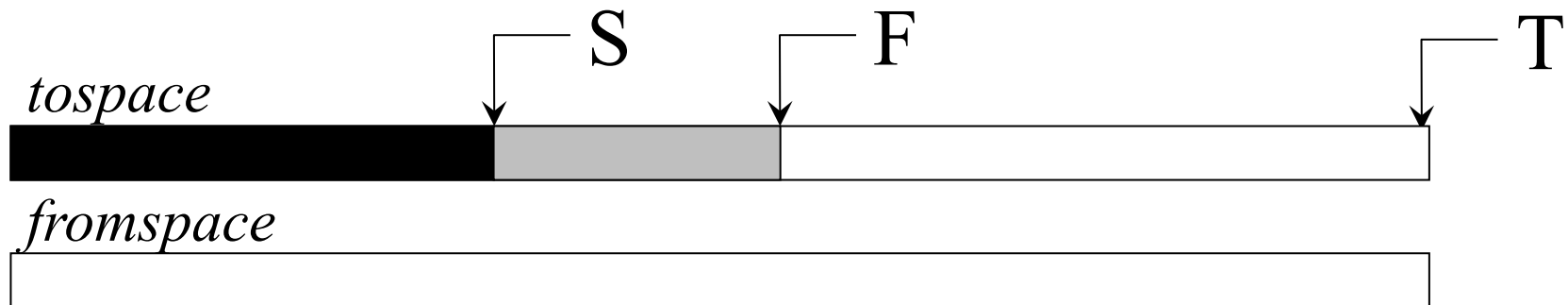


Baker's incremental copying

- Never let a black to white pointer be created.
- Program may only access gray/black objects (that have already been copied).
- A read barrier on each pointer read.
- When the program tries to access a from-space object *A*, *A* is first copied to to-space (becoming gray) and only then "given" to program.
- Program never holds a pointer to a white object, it can never write a black-to-white pointer.

Recall Cheney's

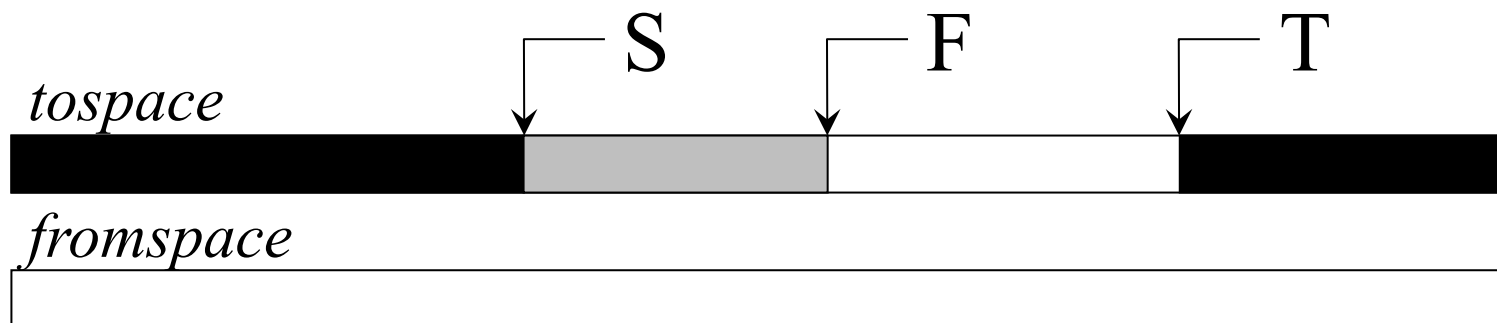
- To-space is composed of
 - black objects (have been copied & scanned)
 - Gray objects (have been copied)
 - Free space (for allocation and copying)
- 3 pointers (scan, free, top) separate the semi-space.





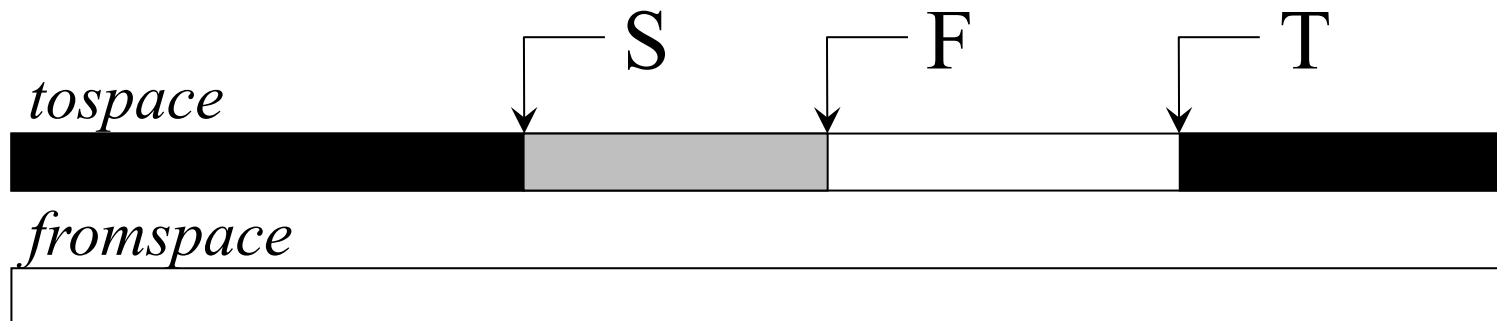
During collector work

- Program allocates objects black (no pointers there) via pointer T.
- Collector copies objects via pointer F.
- Collector scans objects via pointer S.
- Collection terminates when $S=F$.



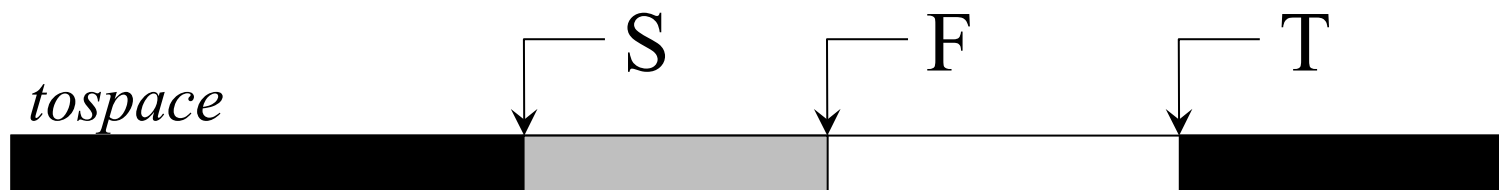
Modifying Cheney's Collector

- Collector first flips from-space & to-space.
- Objects pointed by roots copied to to-space; $S \rightarrow$ beginning, $F \rightarrow$ after copied objects, $T \rightarrow$ end of to-space. Program (root) pointers are now pointing to to-space only.
- Collector small incremental work: scan an object at location s . For any pointer to from-space
 - If object already copied - update pointer.
 - Otherwise - copy, leave forwarding pointer, update pointer.
 - Increment s .



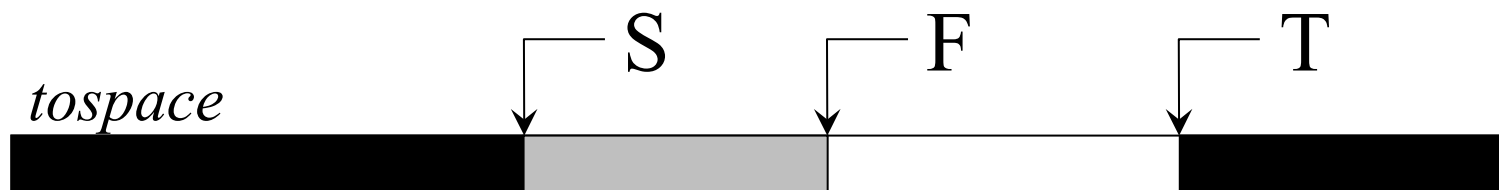
Cooperation of Program

- Allocation via the T pointer downwards.
- Any read operation of the program of a pointer-slot in the heap is trapped and causes invocation of the read barrier.
- Read barrier checks if pointer points to to-space.
- If yes - the read continues as usual.



Cooperation of Program

- Allocation via the T pointer downwards.
- Any read operation of the program of a pointer-slot in the heap is trapped and causes invocation of the read barrier.
- Read barrier checks if pointer points to to-space.
- If not - scan referred object: the referred object is copied into to-space (via pointer F), a forwarding pointer is set, and the pointer in the heap is updated. Only then the read operation continues.





Who Runs the Collector?

- Collector should make sufficient progress to prevent out-of-memory.
- A good collaboration: collect during each allocation.
- Program runs some collector code during allocation, length of time may be tuned.
- (Some more work by the read barrier.)

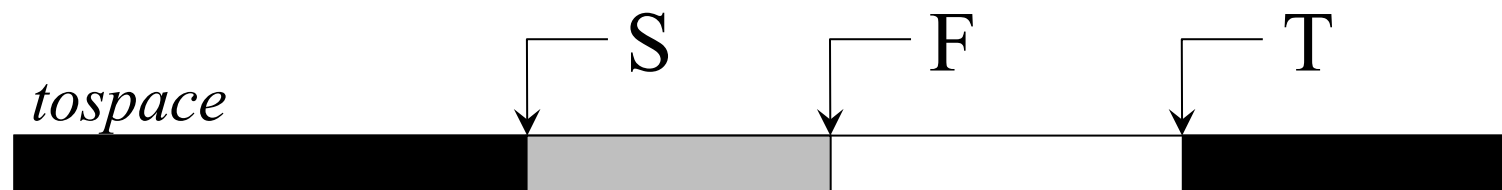


Making Enough Progress

- R = reachable space (to be traced) when collector starts.
- K = number of words scanned with allocation of each word.
- Copying terminates after R/k words are allocated.
- (New objects do not need to be traced.)
- Space required in to-space: $M \geq R(1+1/k)$.
- Tradeoff: small $k \rightarrow$ short pauses, but larger heap.
- Given a program that has R live words, and if M is the size of a semi-space, we may set k to
 $k \geq R/(M-R)$.

Allocation Failure

- If not enough space to allocate a new object, program switches from incremental mode to stop-the-world mode.
- Collector finishes the scan and a new incremental collection starts immediately.
- If there is no free space to finish the collection → an out-of-memory exception.





Properties

- **Good:**
 - No long pauses
 - Garbage is collected when necessary (during new allocations).
- **Bad:**
 - Read barrier! (may add 30% to running time)
- **Real time not fully obtained:**
 - flipping (and scanning roots) time is not bounded.
 - Time for copying a large object is unbounded.
 - Maybe many short pauses occur frequently, not letting the program make any progress.



Replication Copying

Nettles and O'Toole [1992]

- A read barrier is too expensive, use a write barrier.
- Program threads access only from-space (white) objects.
- Copying is done "in the background".
- After copying completed, roots are modified to point to to-space replicas.



Replication Copying

Nettles and O'Toole [1992]

- **A problem: program may modify from-space after a replica has been created in to-space.**
- **Solution: a write barrier.**
- All modifications are recorded in a mutation-log.
- Collector applies modifications to to-space.
- Modifications to a data field are easily copied.
- Modifications to a pointer require care: pointer is in from-space.
- Collector checks if pointed object already copied.
- If yes - update the pointer.
- Otherwise - copy, set forwarding address, update pointer.



Termination

- Collection is complete when all objects in to-space are scanned and mutation log is empty.
- But, as the collector is incremental, mutation log keeps growing.
- When “not much” work is left, the collector is run alone to finish the scanning and mutation log.



Properties

- Good:
 - No long pauses
 - Write barrier instead of read-barrier
- Bad:
 - Each modification is done twice (and recorded)
 - An expensive write barrier - modern write barriers **only trap pointer modifications.**



Going on...

- We will get into more sophisticated incremental and concurrent collectors later in the course.
- But next we present a different type of solution for shortening pause times:
generational garbage collection.