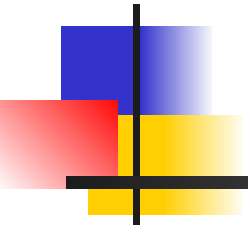
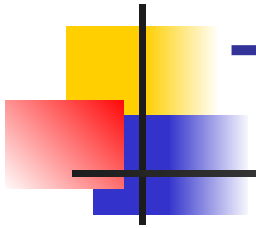
Algorithms for Dynamic Memory Management (236780)



Lecture 4

Lecturer: Erez Petrank

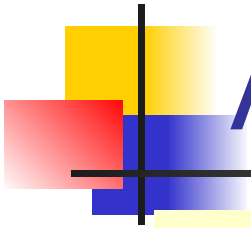


Topics last week

- The Copying Garbage Collector algorithm:
  - Basics
  - Cheney's collector
- Additional issues:
  - Large objects, Lange & Dupont, (Mark-Copy).
- Baker's incremental copying collection:
  - Baker's read barrier: program only accesses objects in to-space.
  - Replication copying with a write barrier

Generational Garbage Collection

[Lieberman-Hewitt83 , Ungar84]



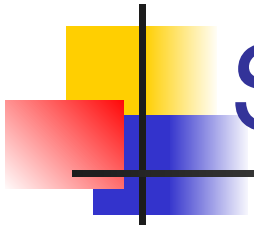
A Property of Programs

"The weak generational hypothesis": most objects die young.

"The strong generational hypothesis": the older the object is the less likely it is to die.

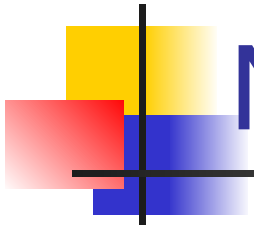
Using the hypothesis:

separate objects according to their ages and collect the area of the young objects more frequently.



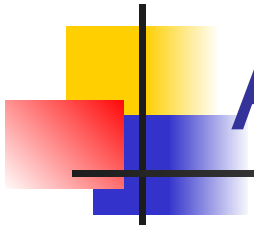
Specifically,

- Heap divided into two or more areas (generations).
- Objects allocated in 1st (youngest) generation.
- The youngest generation is collected frequently.
- Objects that survive the young generation "long enough" are promoted to the 2nd generation.
- The 2nd generation (together with the 1st) is collected less frequently than the 1st.
- Objects surviving the 2nd generation "long enough" are promoted to the 3rd generation, etc.



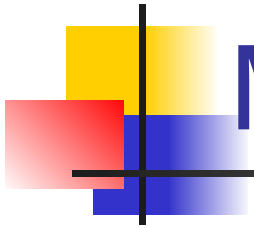
Number of Generations

- Some implementations had 7 generations.
- Some had a dynamically changing number.
- The simplest and probably most popular implementations have two generations, denoted:
  - The young generation and
  - The old generation (a.k.a. mature objects space).



Advantages

- Short pauses: the young generation is kept small and so most pauses are short.
- Efficiency: collection efforts are concentrated where garbage exists.
- Locality:
  - Collector: mostly concentrated on a small part of the heap
  - Program: allocates (and mostly uses) young objects in a small part of the memory.

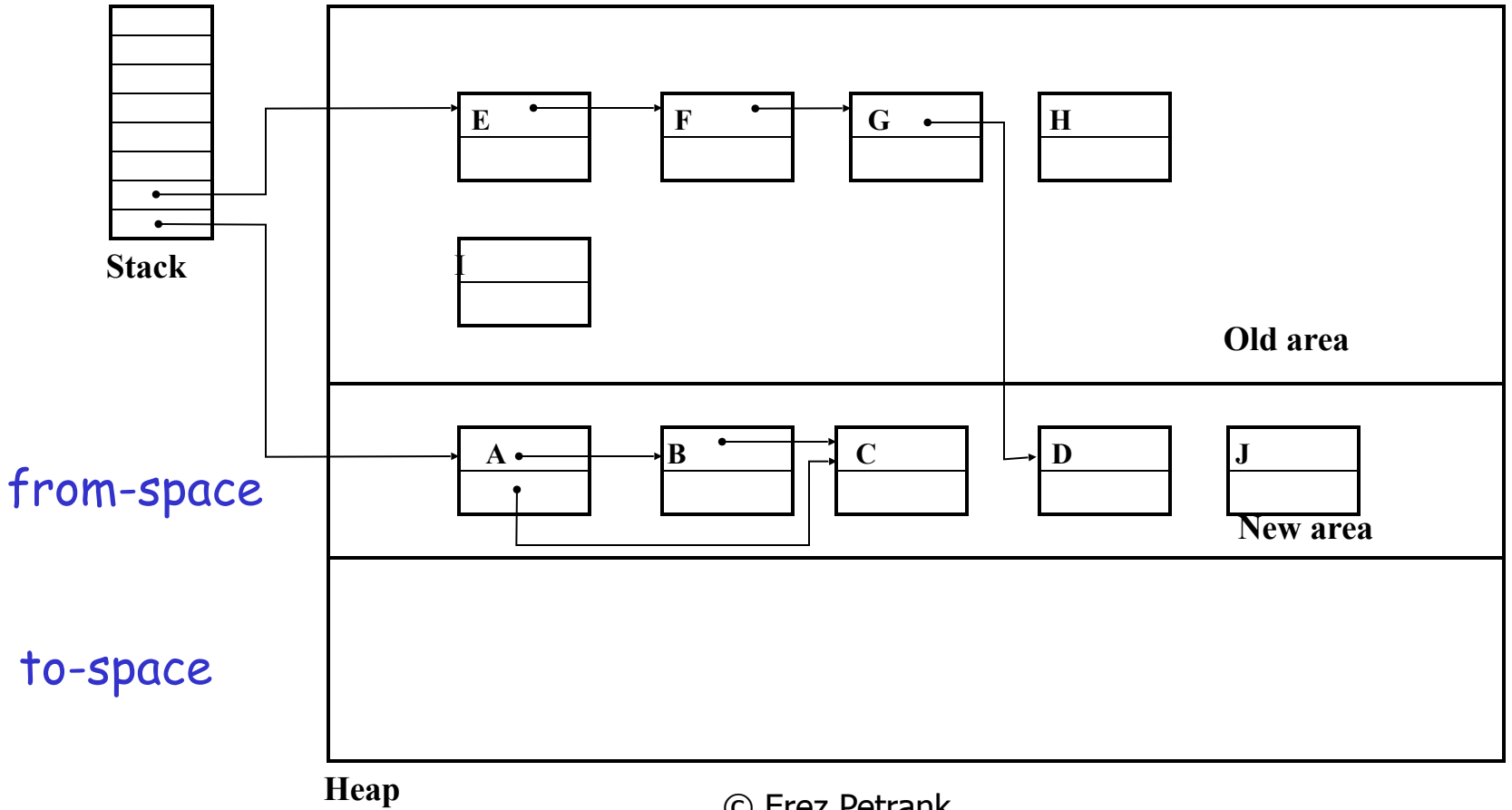


Mark-Sweep or Copying ?

- Copying is good when live space is small (time) and heap is small (space).
- A popular choice:
  - Copying for the (small) young generation.
  - Mark-and-sweep for the full collection.
- A small waste in space, high efficiency.

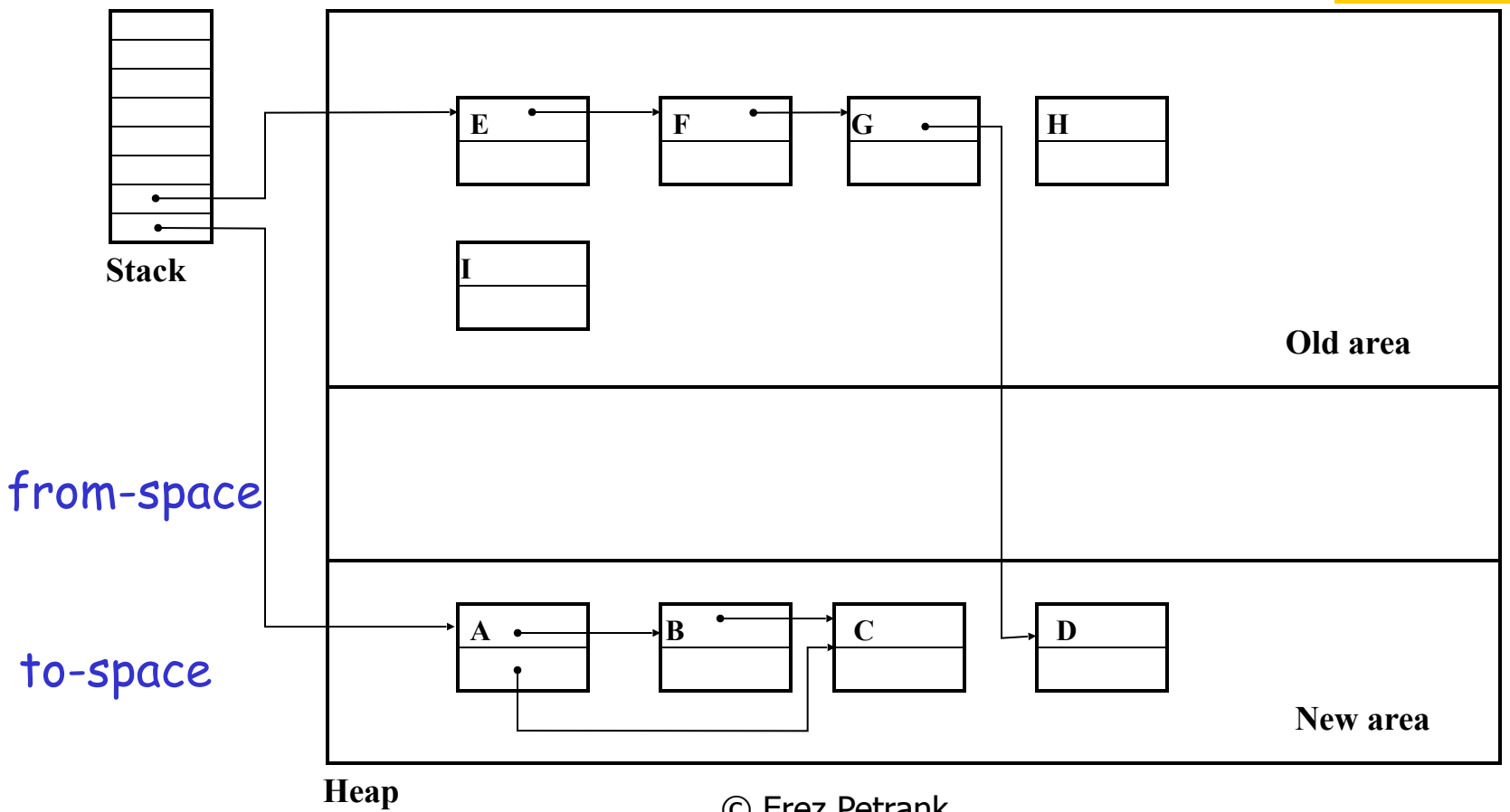
A Simple Example

Note that D is alive! We must be able to detect that.



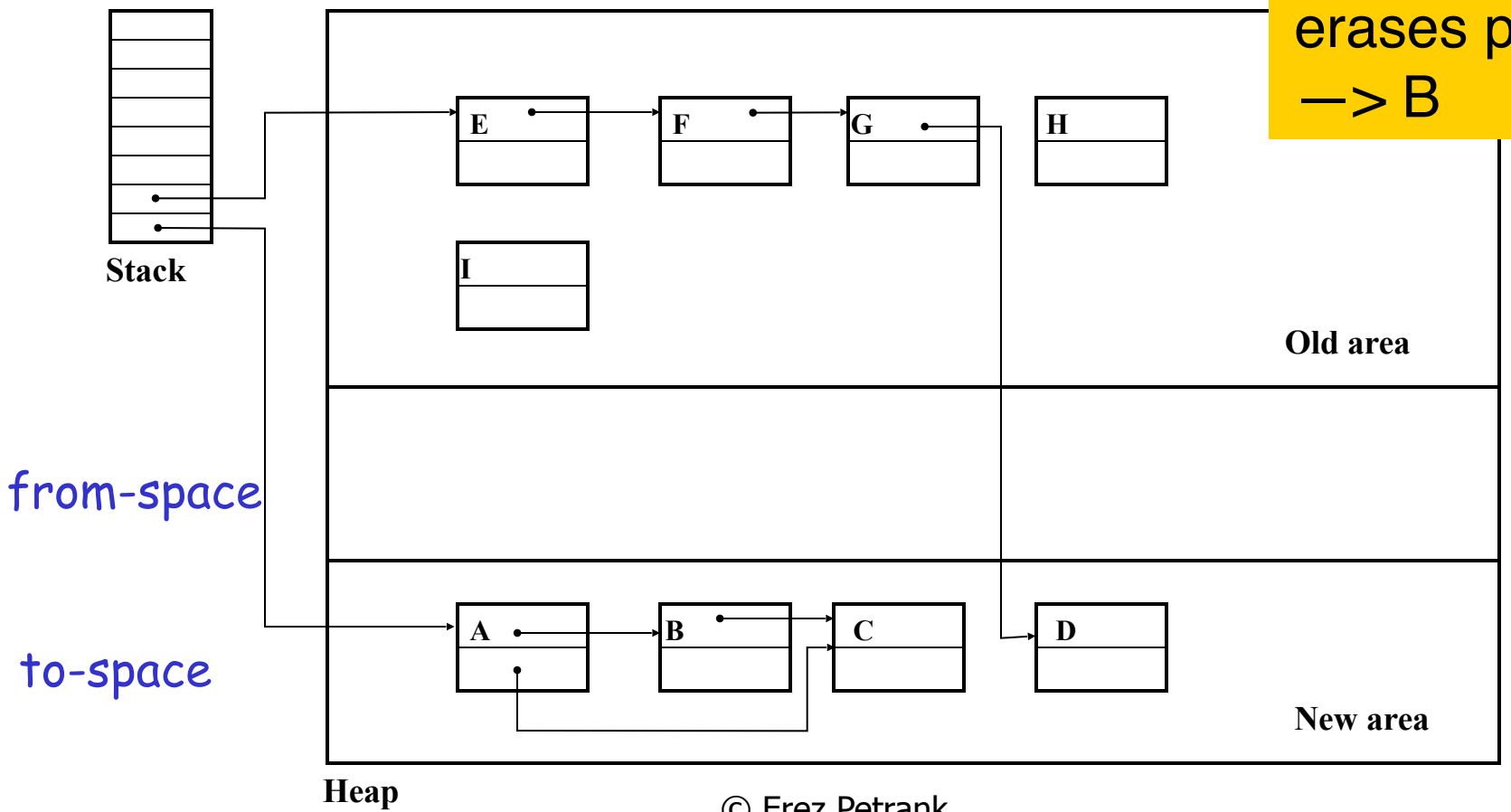
Generational GC - Example

After GC

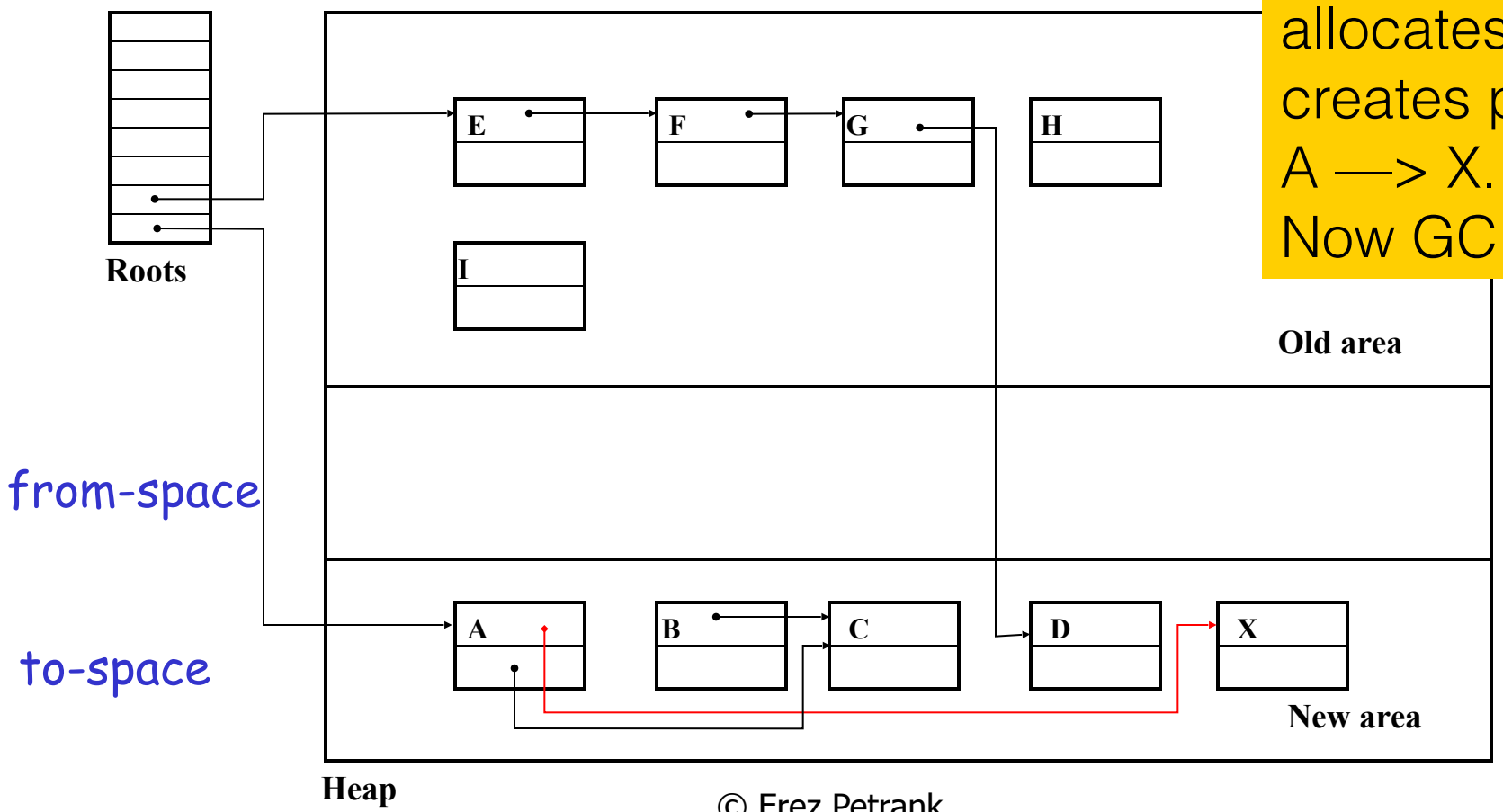


Generational GC - Example

Program erases ptr A —> B



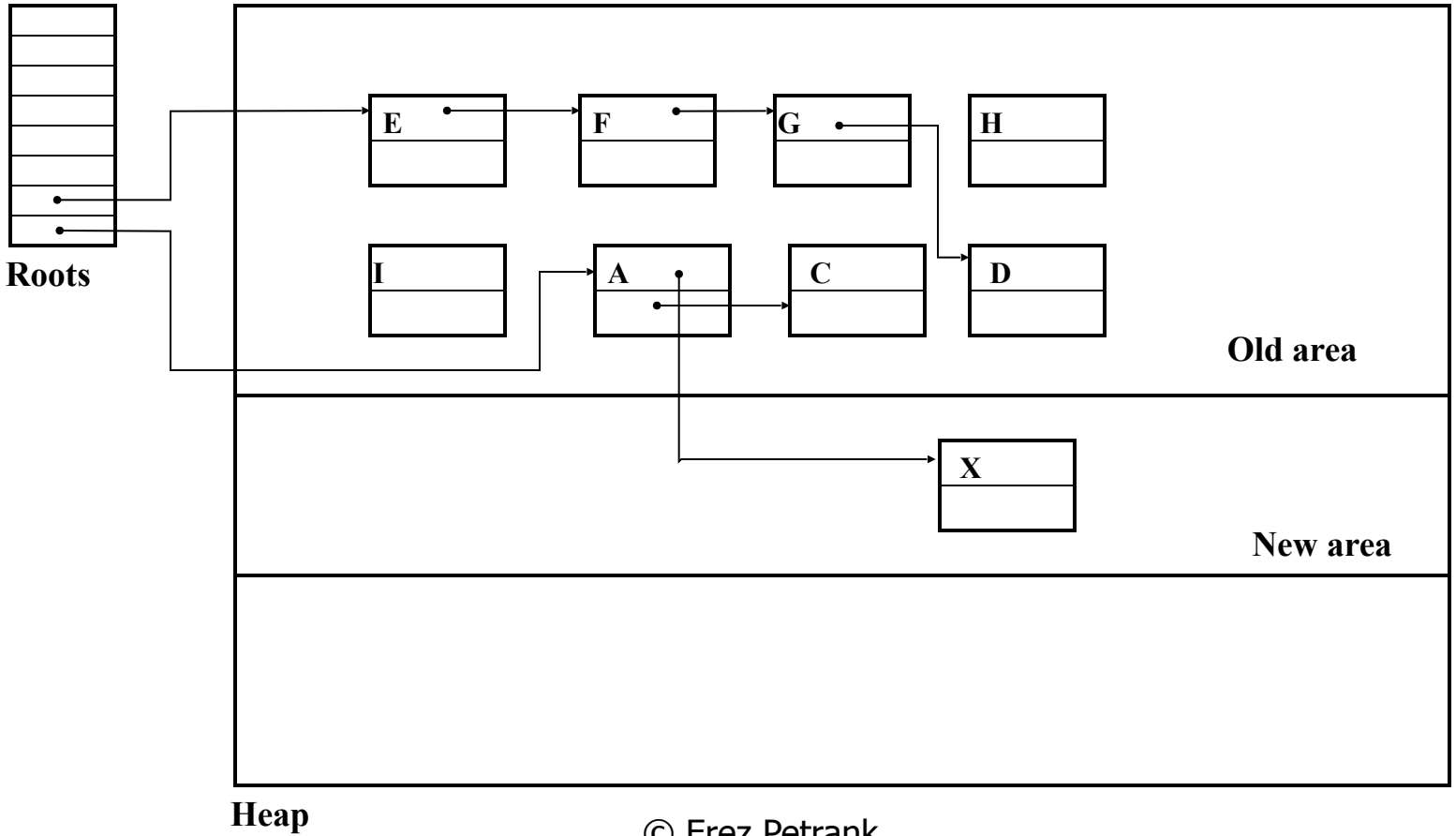
Generational GC - Example

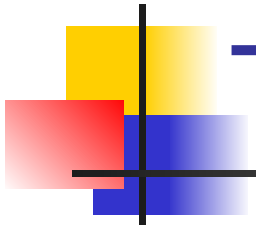


Program allocates X, creates ptr A —> X. Now GC

Generational GC - Example

After GC



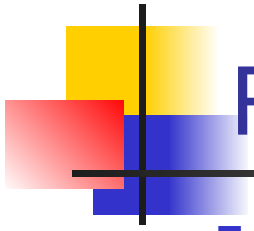


Two Issues

1. Inter-generational pointers (main difficulty):
   - Pointers from old to young objects may demonstrate liveness of (a young) object.
   - Finding them by tracing the old generation misses most of the benefits.
2. Promotion policies: when is an object promoted to the next generation?

Issue I: Inter-Generational Pointers

- The idea:
  - Record all inter-generational pointers.
  - Assume (conservatively) the parent (old) object is alive.
  - Therefore, treat these pointers as additional roots.
- Typically: most pointers are from young to old.
  - Few from old to young.
- When collecting the old generation collect young generation too.
  - Do not record (the many) pointers from young to old.
  - Additional cost is negligible: small young area, infrequent full collections.

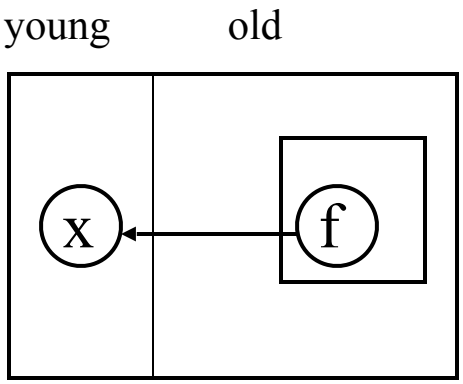


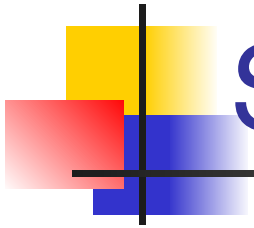
Recording Inter-Generational Pointers

- Inter-generational pointers are created during:
  - Promotion to old generation
  - Pointer Modification in the old generation.
- The first can be monitored by the collector.
- The second requires a write barrier.

Write Barrier Example

```
Update(field f, object x)
{
    if f is in old space and x is in young
            remember f->x;
    f:=x
}
```

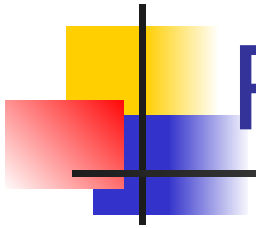




Saving in Costs

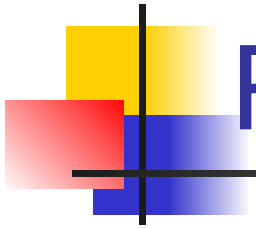
- Saving 1: do not record pointer modifications on the stack. (Major saving)
- Saving 2: do not record pointers from young to old generation.
  - Implication: never collect old generation alone.

We always try to avoid a write-barrier on local variables.



Records

- There are several standard ways to record inter-generational pointers.
- We will mention:
  - Remembered sets.
  - Card marking.
  - “Remembered sets can play cards”.
  - Sequential Store Buffers

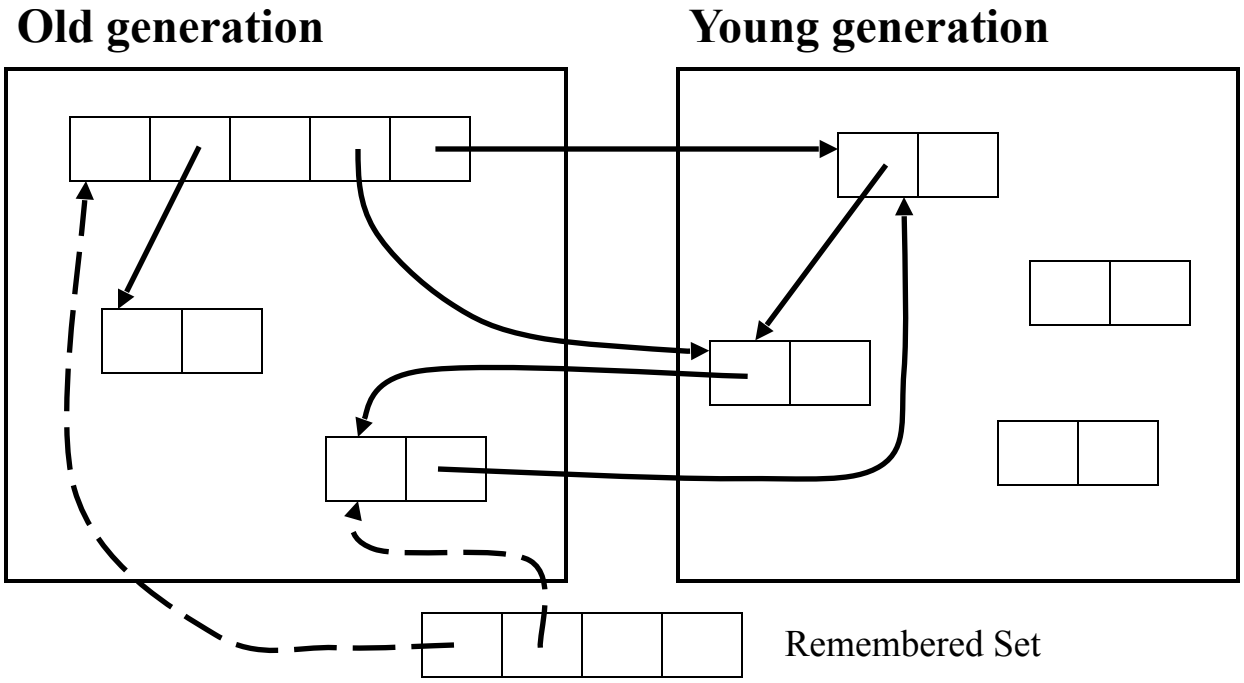


Remembered Sets

- Record locations of pointers (or objects with pointers) to the young generation.
- Updated by write barrier and while collector promoting.

- Avoid duplicates by keeping a bit in object header.
- Remove irrelevant pointers during the collection when irrelevance is discovered.

An Example



Properties (Rem'ed Sets)

- Good for the collector:

  Scanning during the collection is easy.

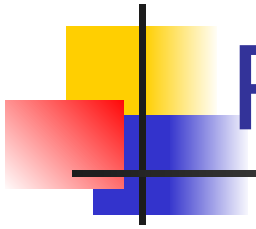
  - Large objects may still create a problem.

- Bad for the program:

  Write barrier is "heavy" $\Longrightarrow$ program suffers.

Card Marking

- Heap divided into small regions called cards.
- Each card has a bit signifying whether it might contain inter-generational pointers
  - These cards are called “dirty”.
- Collector scans dirty cards after the roots.
- Maintenance:
  - Collector clears dirty bit if card has no i.g.p.
  - Collector sets dirty bit during promotion if necessary.
  - Mutator sets dirty bit when modifying a pointer on the card.



Properties (Card Marking)

- Bad for the collector:

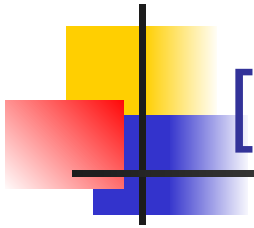
  Scanning during the collection is more difficult.
  - Must scan the whole card to find one pointer.
  - And scan the card again next time as well.

- Good for the program:

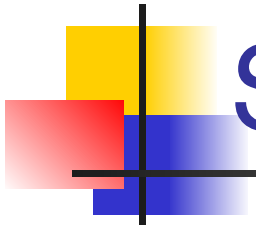
  Write barrier is light $\Longrightarrow$ program runs faster.

- Can we combine the two methods and gain something from both?

"Remembered sets can play cards" [Hosking & Hudson 93]

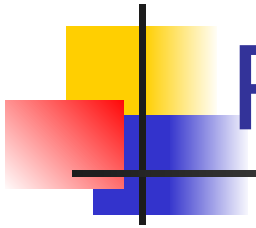


- Program marks modified cards dirty (as in card marking).
- But the collector does not scan cards repeatedly.
- Instead: collector scans dirty cards, updates remembered sets & clears all dirty bits.
- Afterwards, remembered sets used for fast collection.
- Advantage:
  - Program uses fast write barrier.
  - Collector scans a card (at most) once per modification.



Sequential Store Buffer (SSB)

- Idea: collector does most of the work.
- When the program modifies a pointer, it adds its address to the end of a buffer, called the **sequential store buffer**.
- At collection time, entries in the SSB are distributed into remembered sets.
- Addresses that may contain i.g.p.'s must appear in the sequential store buffer or in the remembered set.

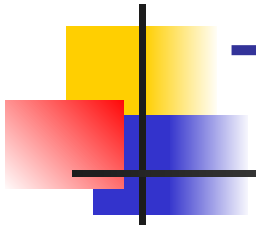


Properties (SSB)

- Similar to remembered sets, except that the work is postponed and not done when the program run.
- Faster program, slower collection.

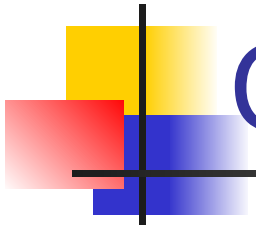
Issue II: Promotion policies

- Measuring Object Lifetimes:
- Clock time.
  - Machine dependent.
  - Implementation dependent.
- Count bytes of heap allocated.
  - Machine independent.
  - Better reflects the memory demands.



Tradeoffs: Size and Policies

- Tradeoff 1 – size of the young generation:
  - Small $\Longrightarrow$ short pauses.
  - But, small $\Longrightarrow$ more collections (decreasing efficiency).
- Tradeoff 2 – promotion policy:
  - Promoting fast $\Longrightarrow$ more space in young generation —> fewer collections.
  - But, promoting fast $\Longrightarrow$ more tenured dead objects that will stay in the heap for long.

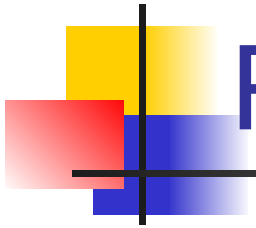


Controlling the Tradeoff

- Multiple generations
- Promotion threshold
- Adaptive tenuring

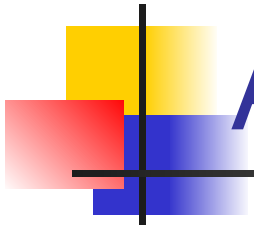
Multiple Generations

+ Pros:
  + Keep young generation small & promote fast.
  + Intermediate gen's let prematurely promoted objects die before getting to the old generation.
  + Intermediate generations fill more slowly and need to be collected less frequently.
- Cons:
  - More overhead: managing generations, recording inter-generational pointers, etc.
  - Pause time for intermediate generations may be disruptive



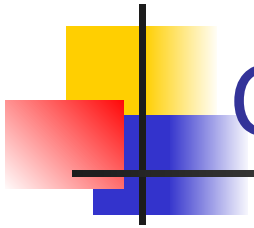
Promotion Threshold

- Promotion by the number of times an object survived collections.
- Must use a counter for each object, or maintain several spaces, each for a different age.
- Tuning the promotion number is complex. Depends on the application.
  - An alternative: change the promotion policy dynamically.



Adaptive Tenuring

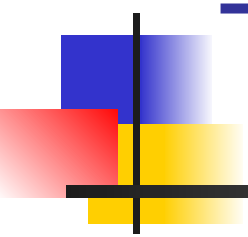
- Goal: tenure enough to free space in young generation, but not too much - don't promote young objects.
- A typical choice: aim for 80% free young space.
- Problem: we know only after collecting & promoting!
- Solution: assume behavior is steady.
  - If not enough objects promoted – reduce promotion threshold for next collection.
  - Else – increase it.



Generational Collection Properties

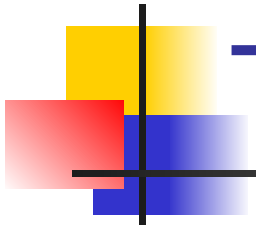
- Mostly short pauses
- Good locality (for both program and collector).
- Improved efficiency (in spite of overhead).
- Widely used…

- One problem: major collections are still long.



The Train Algorithm

[Hudson & Moss 92]



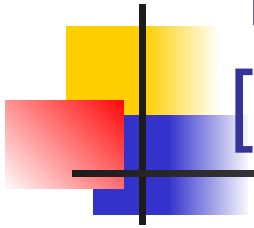
The Train Algorithm

Collect the old generation (mature object space) incrementally.

With each young generation collection, perform part of the old generation collection.

Employed by Sun's Hotspot JVM.

The Train Algorithm [Hudson-Moss 92]

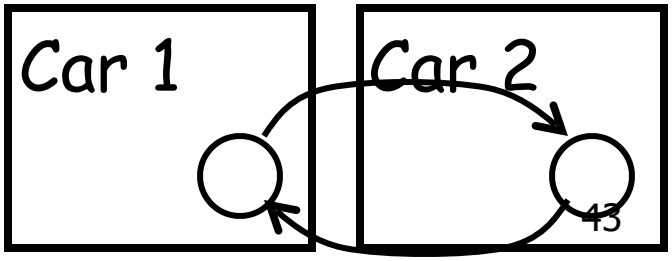


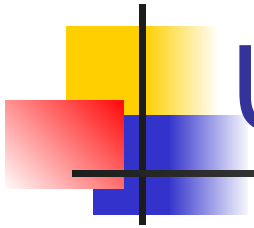
- Structure of heap:
  - Young generation
  - Mature Object Space (MOS) = old generation.
  - Large Object Space (LOS)
    - Each large object has a representing record in the normal heap that keeps age and gets tenured.

Mature Object Space

- Goal: collect old generation (MOS) with short pauses.
- First try: partition the MOS into cars and collect one car at a time.
- Keep a remembered set for each of the cars.
- A collection: collect young generation plus one car of the old generation.
- Pauses are short, never need to collect the entire heap.

Problem: cyclic structures!

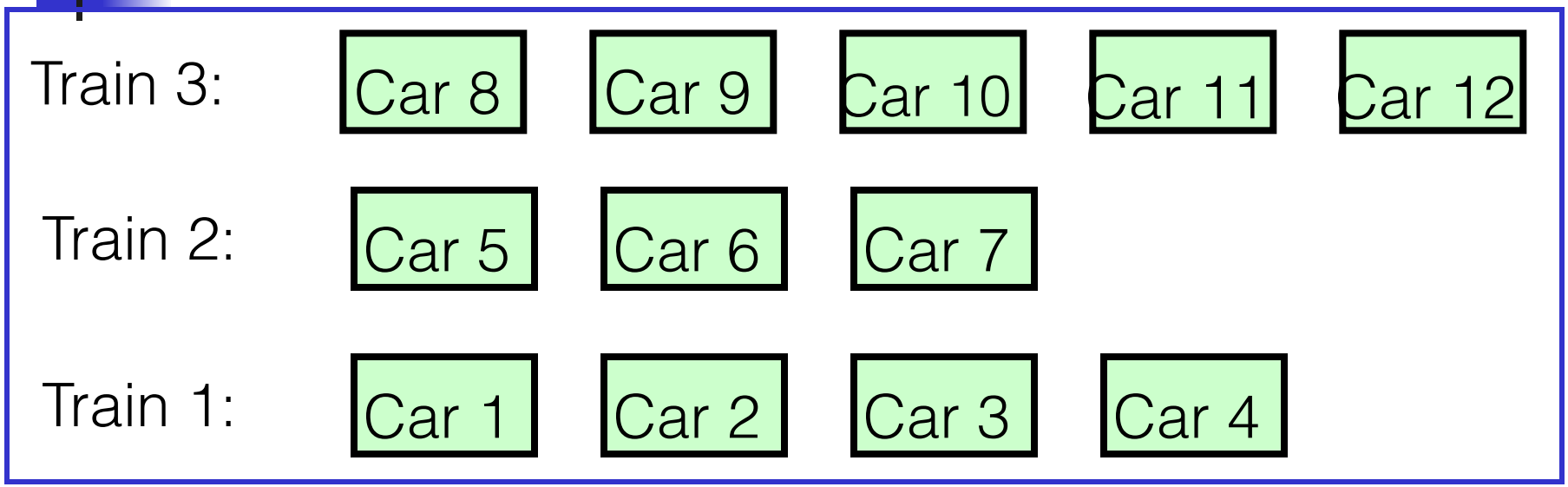




Using Cars and Trains

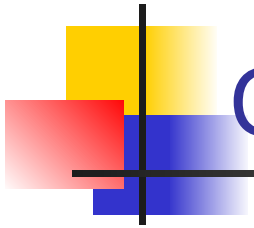
- Solution: partition old generation into cars, but group cars into trains.
  - Remember set for train -- union for its cars minus intra-train references
- Goal: move a cyclic garbage structure into a single train and collect the full train.
- Avenue: when collecting a car, move all live objects out “wisely”. Then - reclaim the car.
- Idea: when an object moves out of a car, place it on another train that may contain objects of its cluster.

Structure



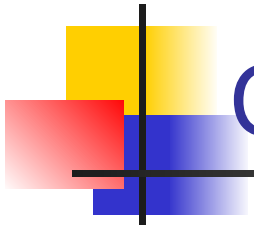
Order: collect 1st car of 1st train and then reclaim it.

Thus – record only pointers from higher to lower cars.



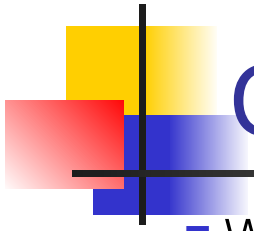
Collection of Current Car

- Mark objects directly reachable from the roots.
- If there are no root pointers to the train and train's remembered set empty, then reclaim entire train.
- If no root pointers to the car and car's remembered set empty, then reclaim entire car.
- Otherwise…



Collection of Current Car – Cont'd

- An object referenced by roots (or young generation) is moved to the last train
  - Or a new one if too long.
- An object A referenced from train k is moved to train k (last car). Other objects in the car reachable from A are moved to train k too.
- An object referenced from another car (same train) is moved to the last car (with descendants).
- For each moved object, keep a forwarding pointer, and update all pointers via rememb'd set or roots.



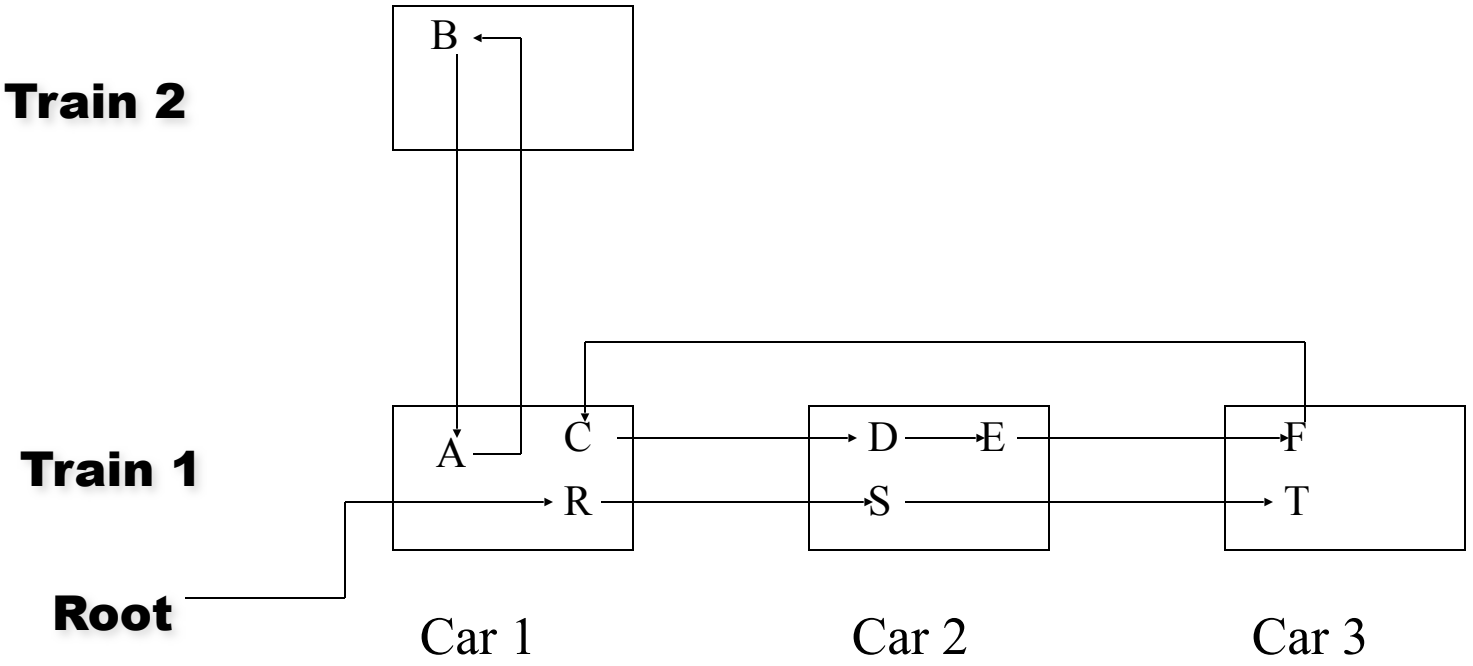
Collection of Current Car – Cont'd

- When all objects have been copied out, the current car is recycled.

- Note:
  - If the current train has a cyclic garbage cluster, it will never leave the train.
  - If the current train has live objects they will eventually leave the train
  - Thus, a cyclic structure entirely in this train is reclaimed eventually.

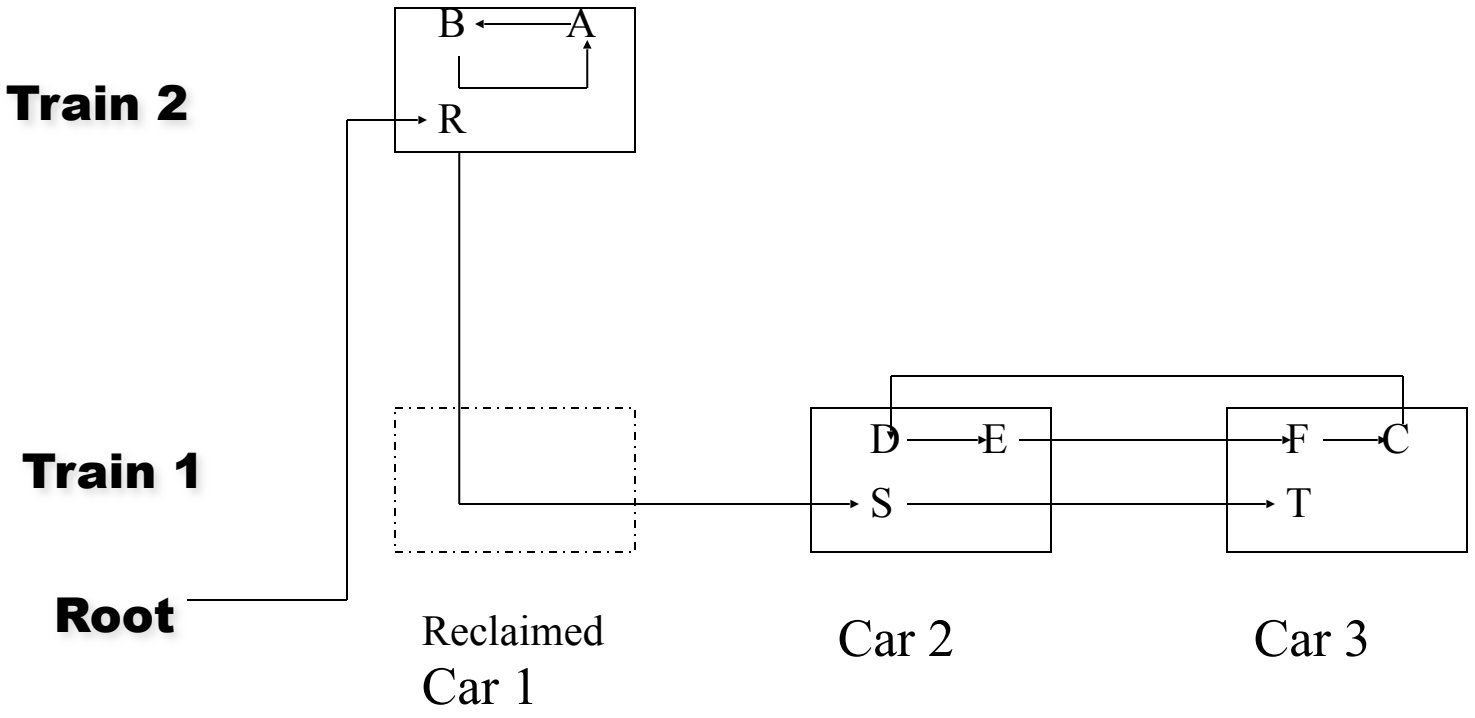
Example

Say, max num of objects that can fit in the car is 3



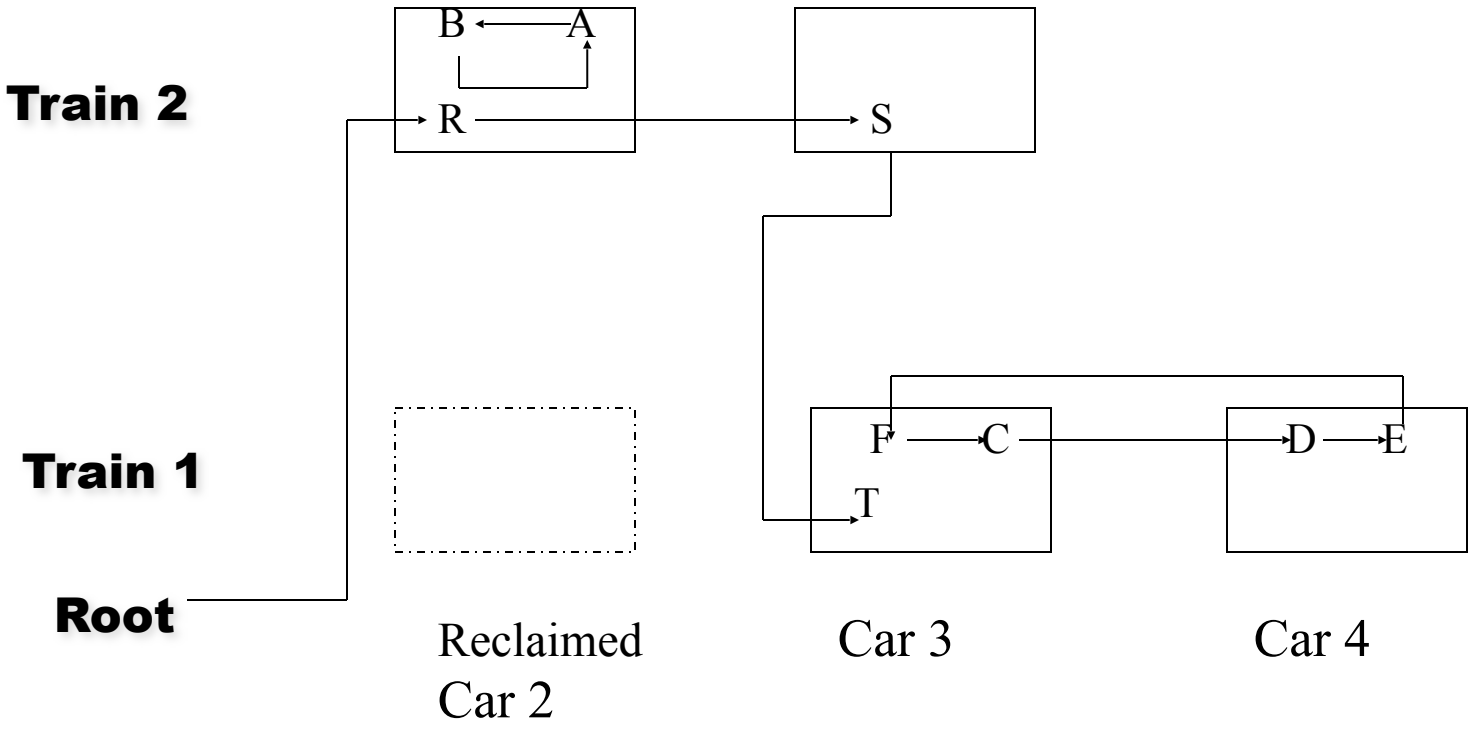
Example(continue)

After first invocation of the algorithm



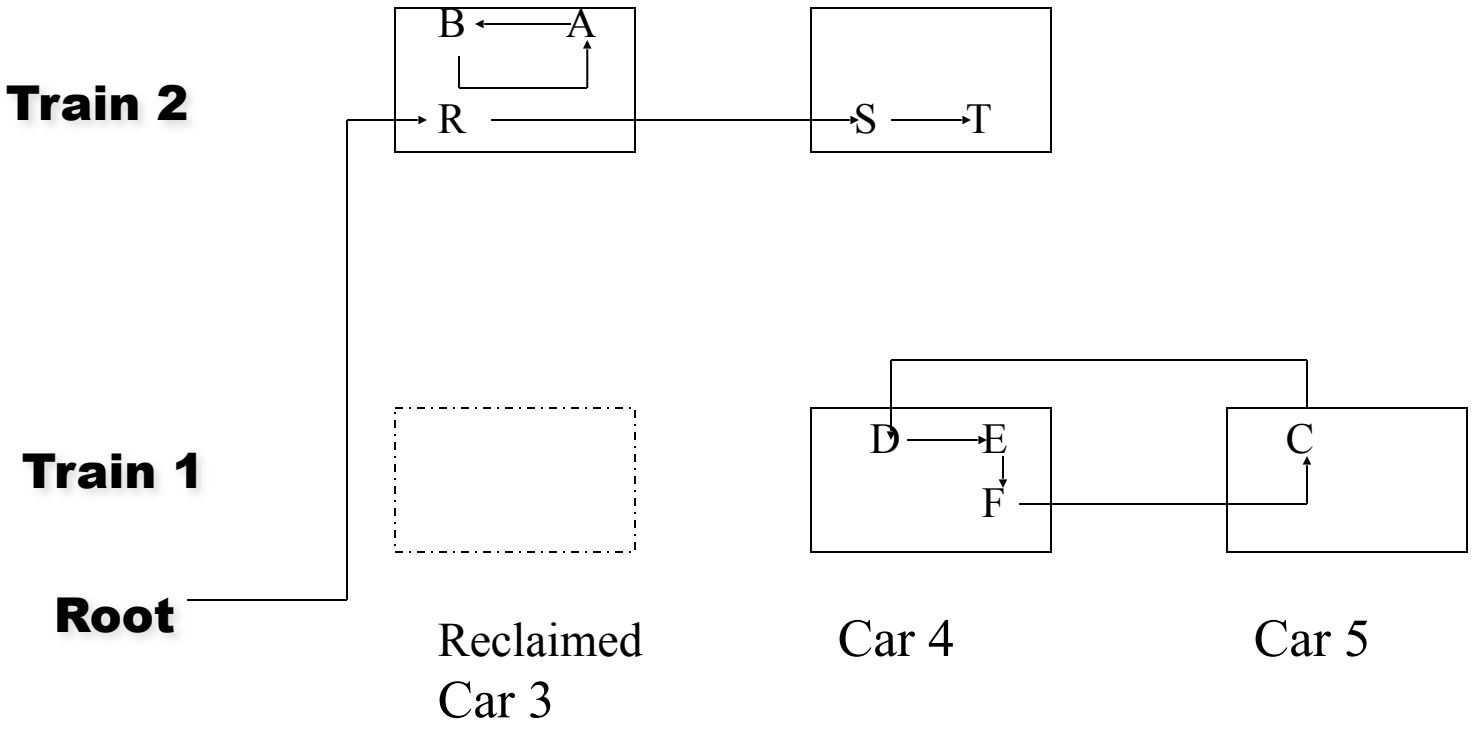
Example (Continue)

After second invocation of the algorithm



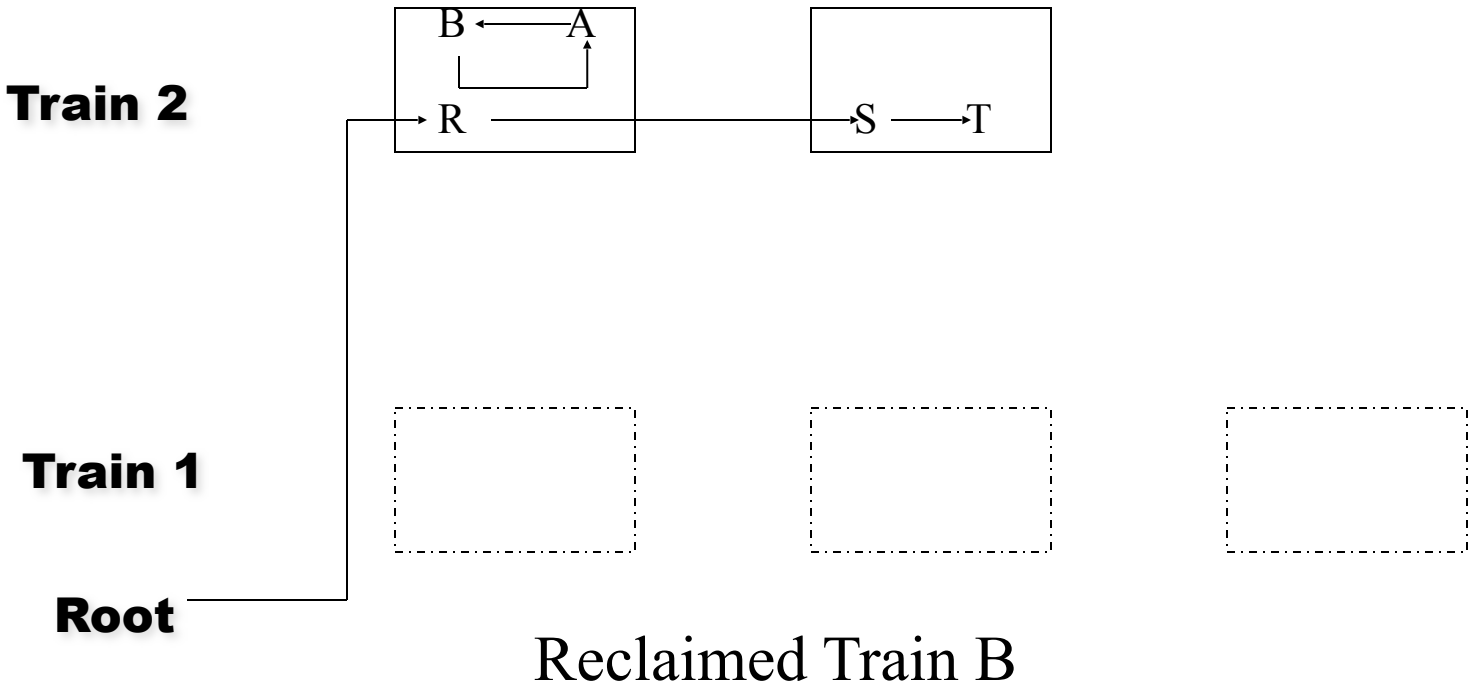
Example (Continue)

After third invocation of the algorithm



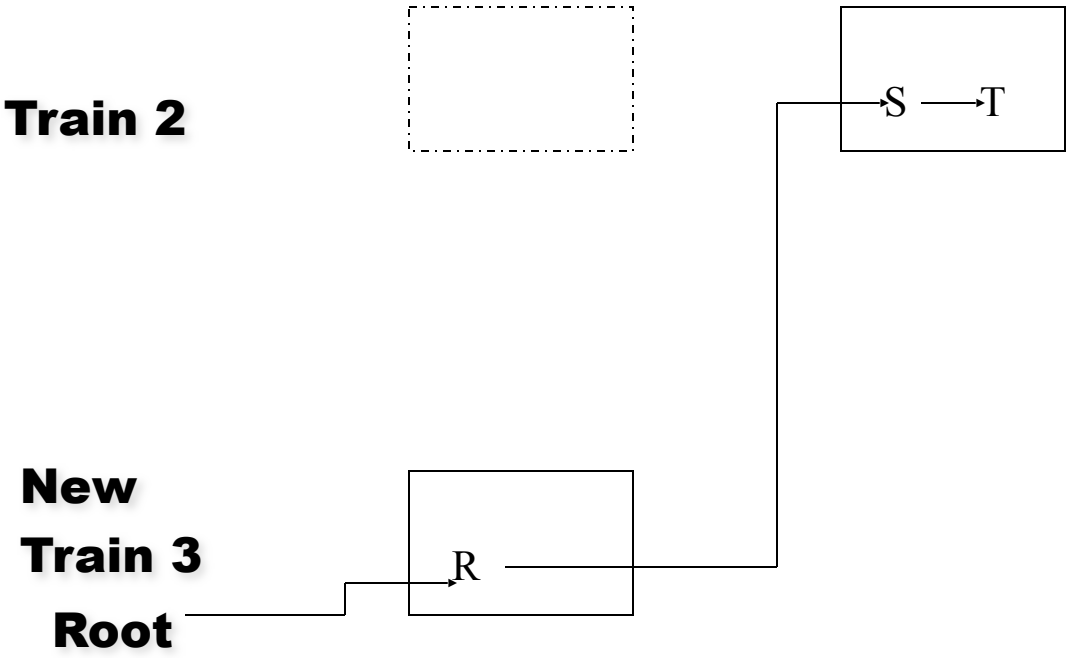
Example (Continue)

In the middle of fourth invocation of the algorithm



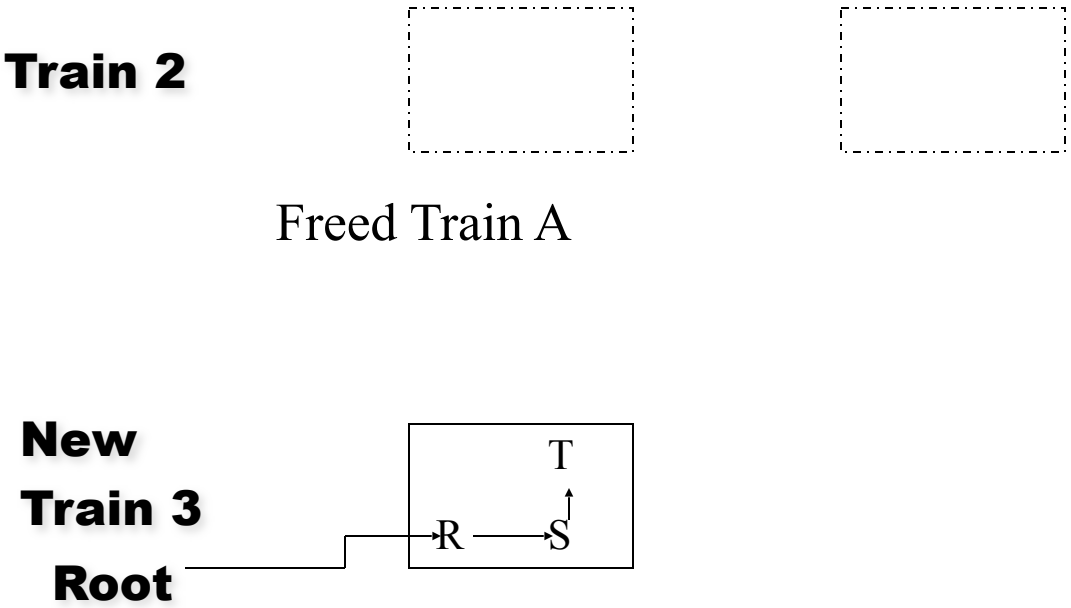
Example (Continue)

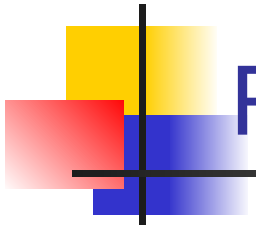
After fourth invocation of the algorithm



Example (Continue)

After fifth invocation of algorithm





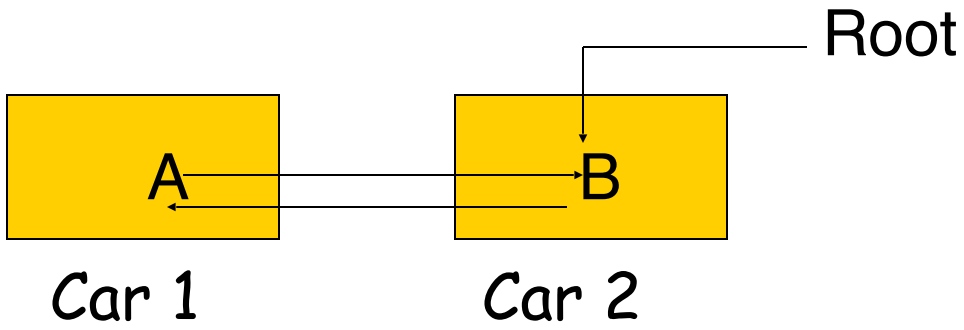
Properties

- Old generation is collected incrementally.
- The algorithm re-clusters live objects with no additional cost.
- The algorithm periodically copies all reachable objects.

A Termination Flaw [Seligman-Grarup 95]

There is a (rare) scenario in which the algorithm never finishes handling a train.

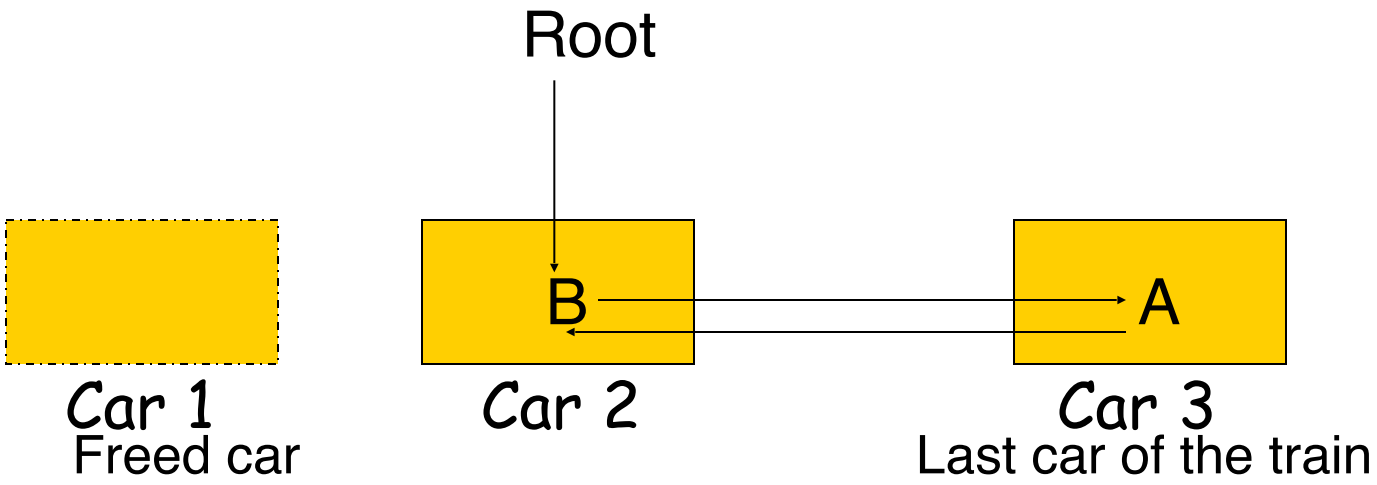
Consider the following situation:



Assume A and B are too large to reside on the same car

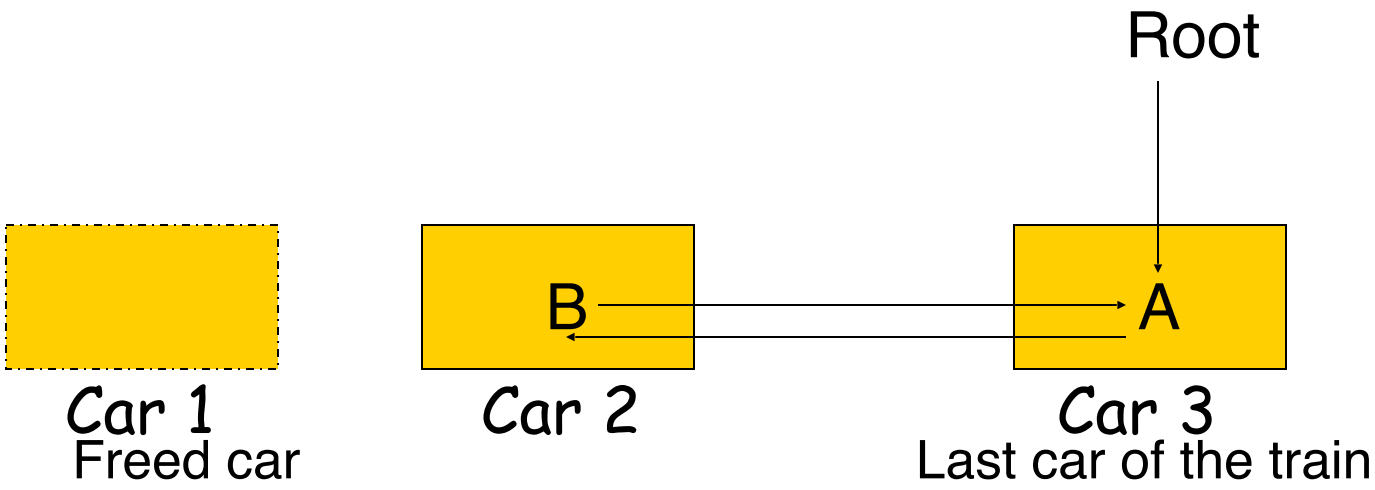
A Termination Flaw - Cont'd

After collecting car 1, car 3 is created.



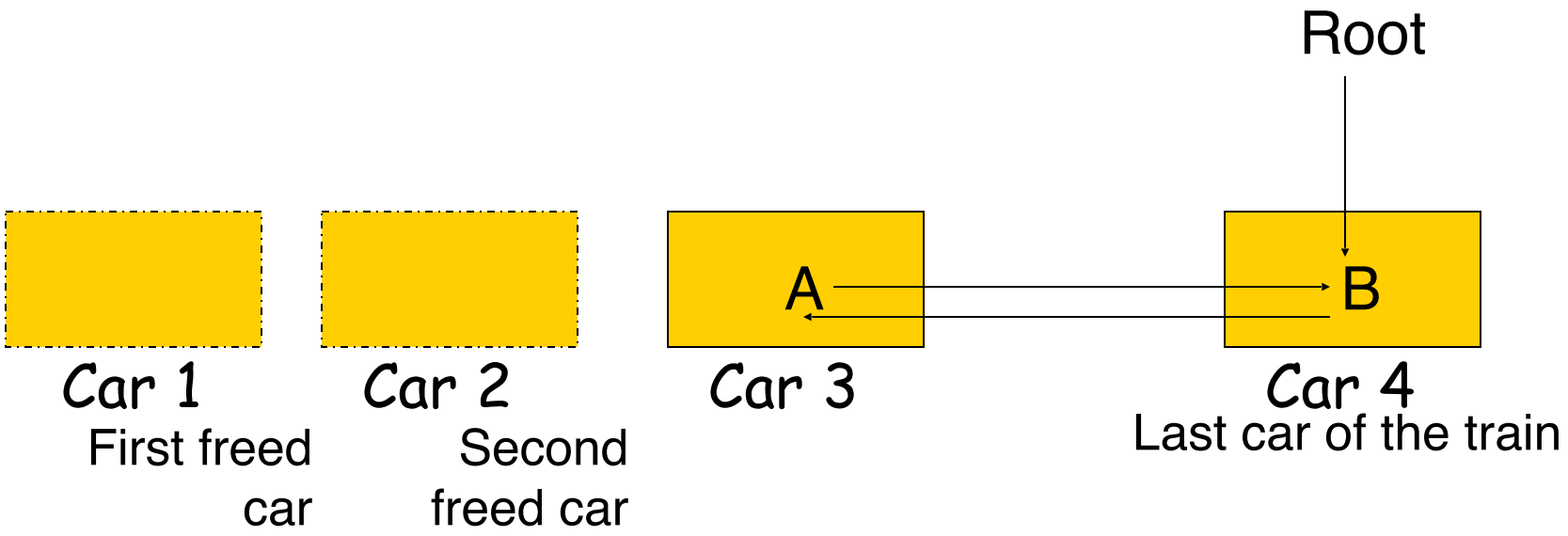
A Termination Flaw - Cont'd

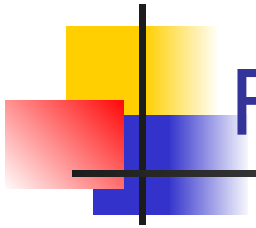
After collecting car 1, program resumes. Now, mutator modifies the root to point to A. Next collector works on Car 2.



A Termination Flaw - Cont'd

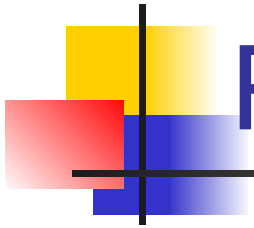
- B copied to last car of current train. But when program resumes, mutator changes the root **again**, to point to B. And so the train is never reclaimed...





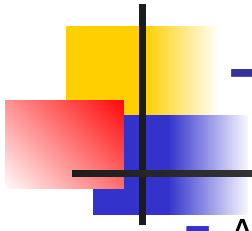
Fixing by Special Monitoring

- If no object is moved out of the train then collection denoted futile.
- Whenever a futile collection occurs, one of the external references to the train is recorded:
  - It must exist (otherwise, reclaim train)
  - It is not in first car (otherwise, not futile).
- The recorded reference is an additional root for subsequent collections on this train.
- Referenced object is evacuated according to the standard evacuation heuristic.



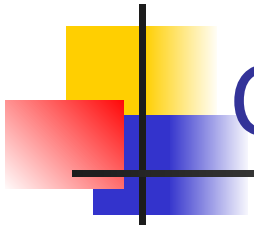
Popular Objects

- Moving popular objects may be disruptive!
  - Large remembered sets to update.
- Solution:
  - cars with popular objects are not collected.
  - Non-popular objects are evacuated as usual.
  - car is “logically” moved to end of highest train from which popular objects are referenced.



Train Algorithm - Properties

- Advantages:
  - Incremental (short pauses).
  - Partial compaction and clustering.
  - Easy implementation (on stock hardware with no special operating system features).
  - Copying, but not much space must be reserved.
- Disadvantages:
  - Time overhead for repeated copying.
  - Space overhead: car management & rem'ed sets



Generational Collection - Summary

- Reduces most of the pause times (all of them with the Train algorithm).
- Improves efficiency by concentrating the efforts where garbage exists.
- Improves cache and paging behavior of collector and program.
- Widely used.