

# 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 collector
  - Read barrier - program only accesses objects in to-space.



# Homework

---

- Will be available on Tuesday on the course web-page.
- To be delivered two weeks later.

# Recall: incremental Collection

- Partition the collection work to "increments" ..
- Baker's copying collector [1977]:
  - During each allocation do some GC work.
- Uniprocessor world (not good for SMP).

Stop the world:



Incremental:





# Not so simple!

---

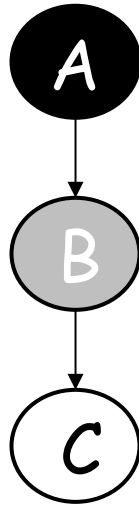
- The heap changes while we collect.
- For example,
  - The collector scans an object B, but before marking its children, the collector is stopped and the program resumes.
  - When the collector returns, the children may have been modified.

Incremental:

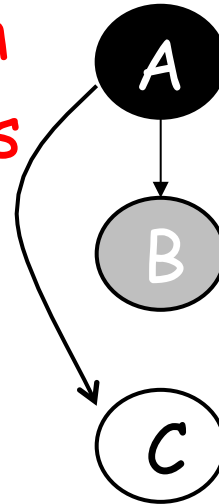


# Example

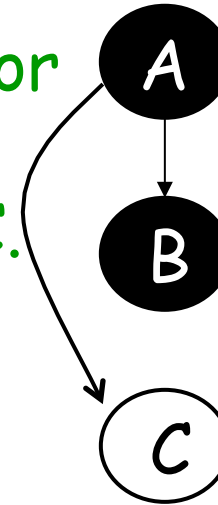
GC marks  
A's  
children  
and  
makes A  
black.



Program  
modifies  
pointer.



Collector  
fails to  
trace C.



Time



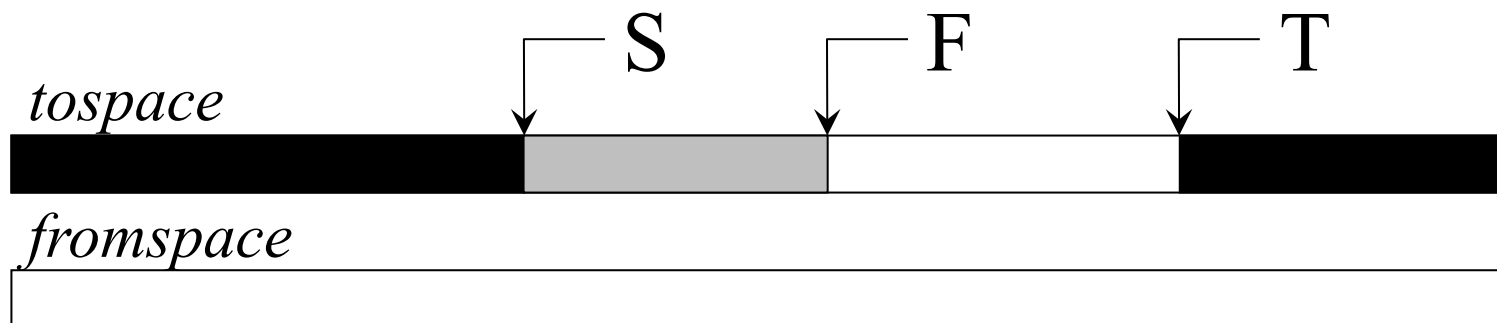
# Baker's incremental copying

---

- Goal: never let a black to white pointer be created.
- Idea: 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.

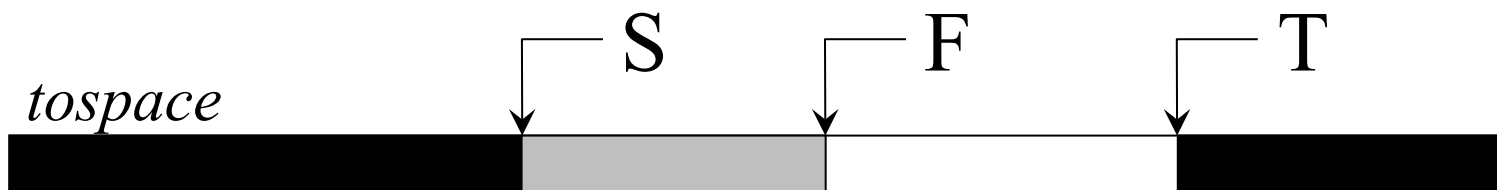
# During collector work

- 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  $S=F$ .
- OutOfMemory if copy/alloc fails between F and T



# Cooperation of Program

- **Allocation** via the T pointer downwards.
- A **read-barrier**: any read of a pointer-slot is trapped and checked.
- If it points to to-space, the read continues as usual.
- If not and referent copied - use forwarding pointer to update the pointer and use the new one.
- Otherwise: copy; install forwarding ptr; update pointer; use updated pointer.





# 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 - use updated pointer.
- Otherwise - copy, set forwarding address, use updated 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**.
  - Need a word for forwarding pointer in each object.



# 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.



## 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.

# 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.



# More Specifically,

---

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



# Number of Generations

---

- Some implementations had 7 generations.
- Some had a dynamically changing number.
- The simplest and probably most popular implementations has two generations, denoted:
  - The **young** generation and
  - The **old** generation (or 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.



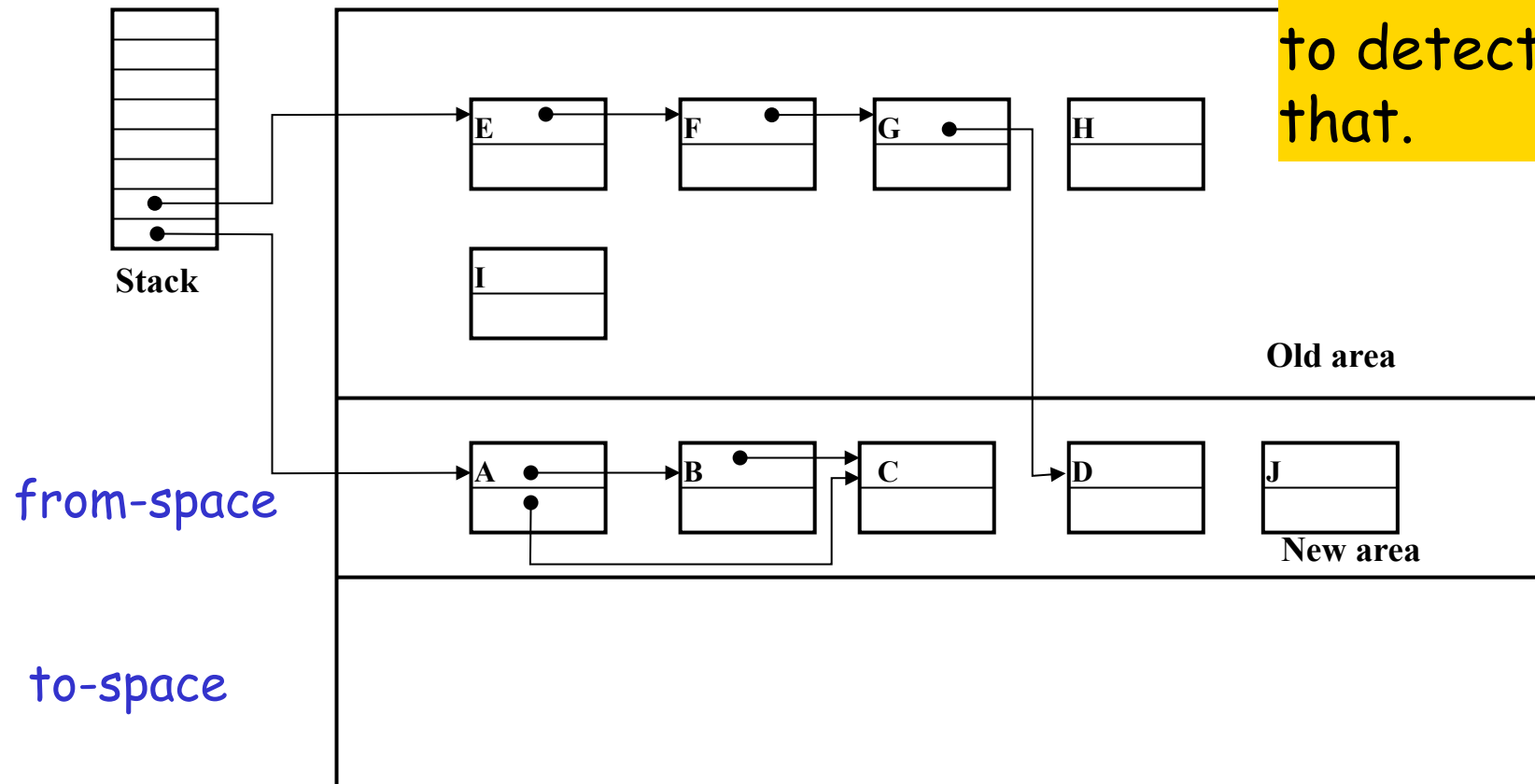
# Mark-Sweep or Copying ?

---

- 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.

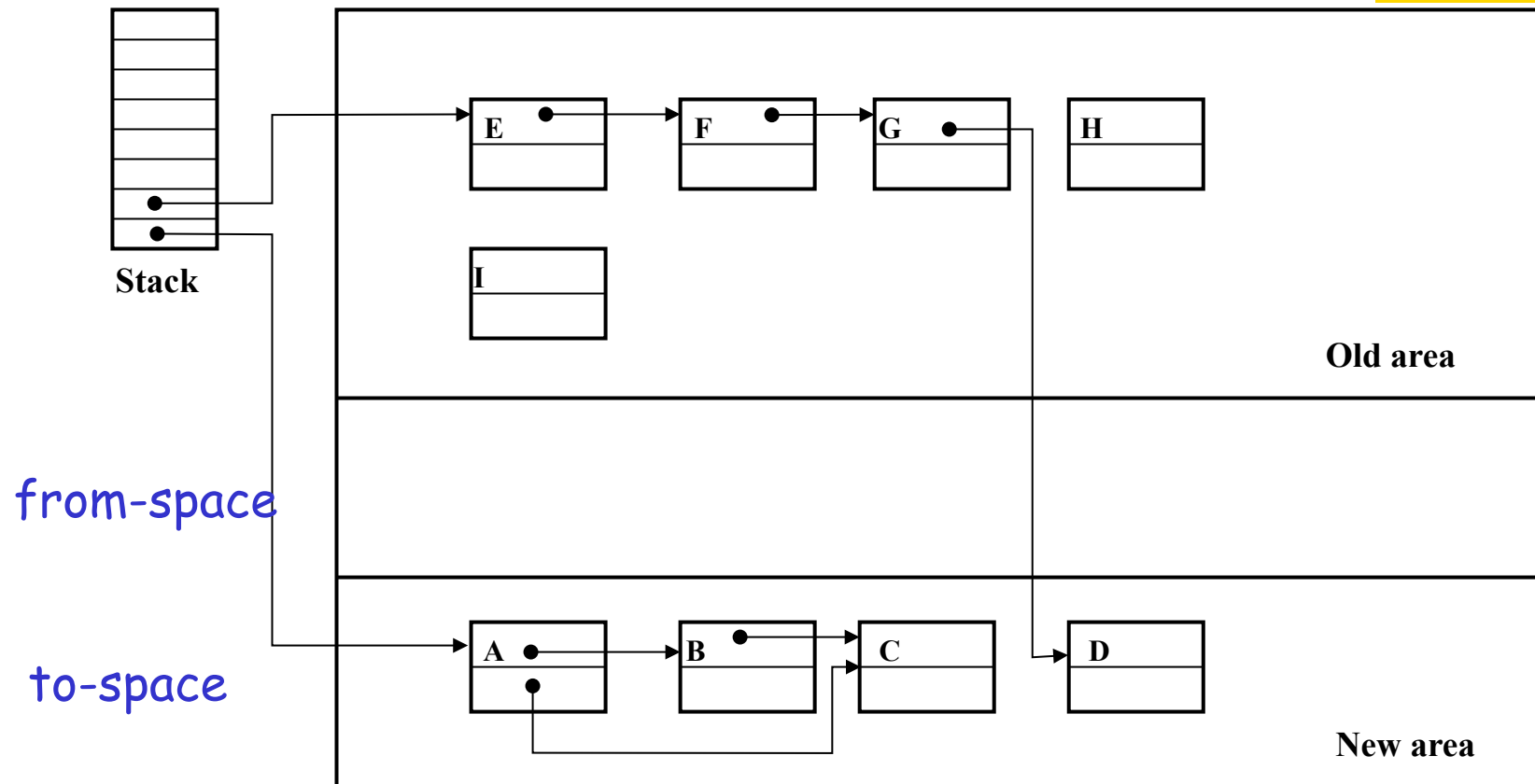
# A Simple Example

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

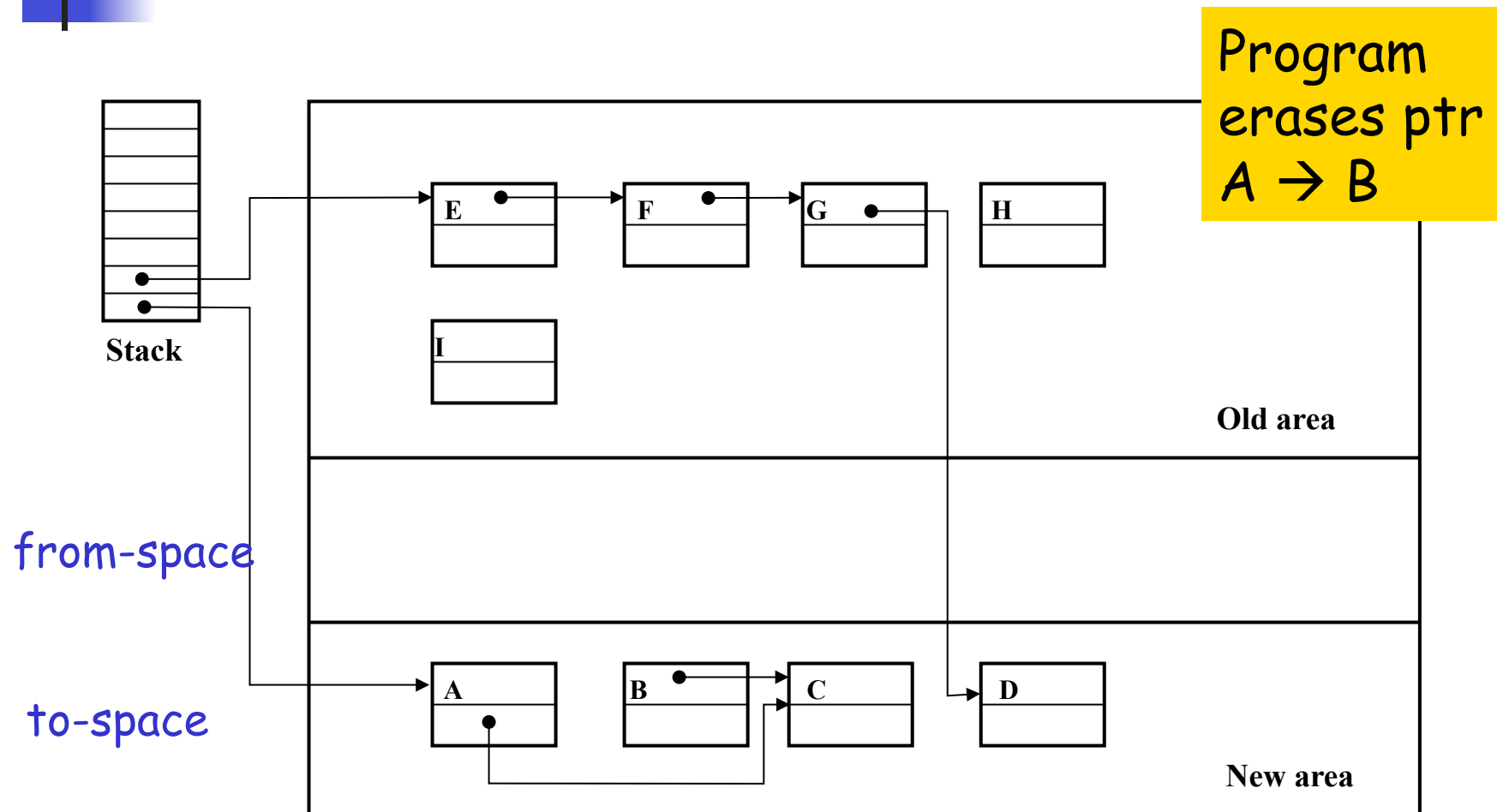


# Generational GC - Example

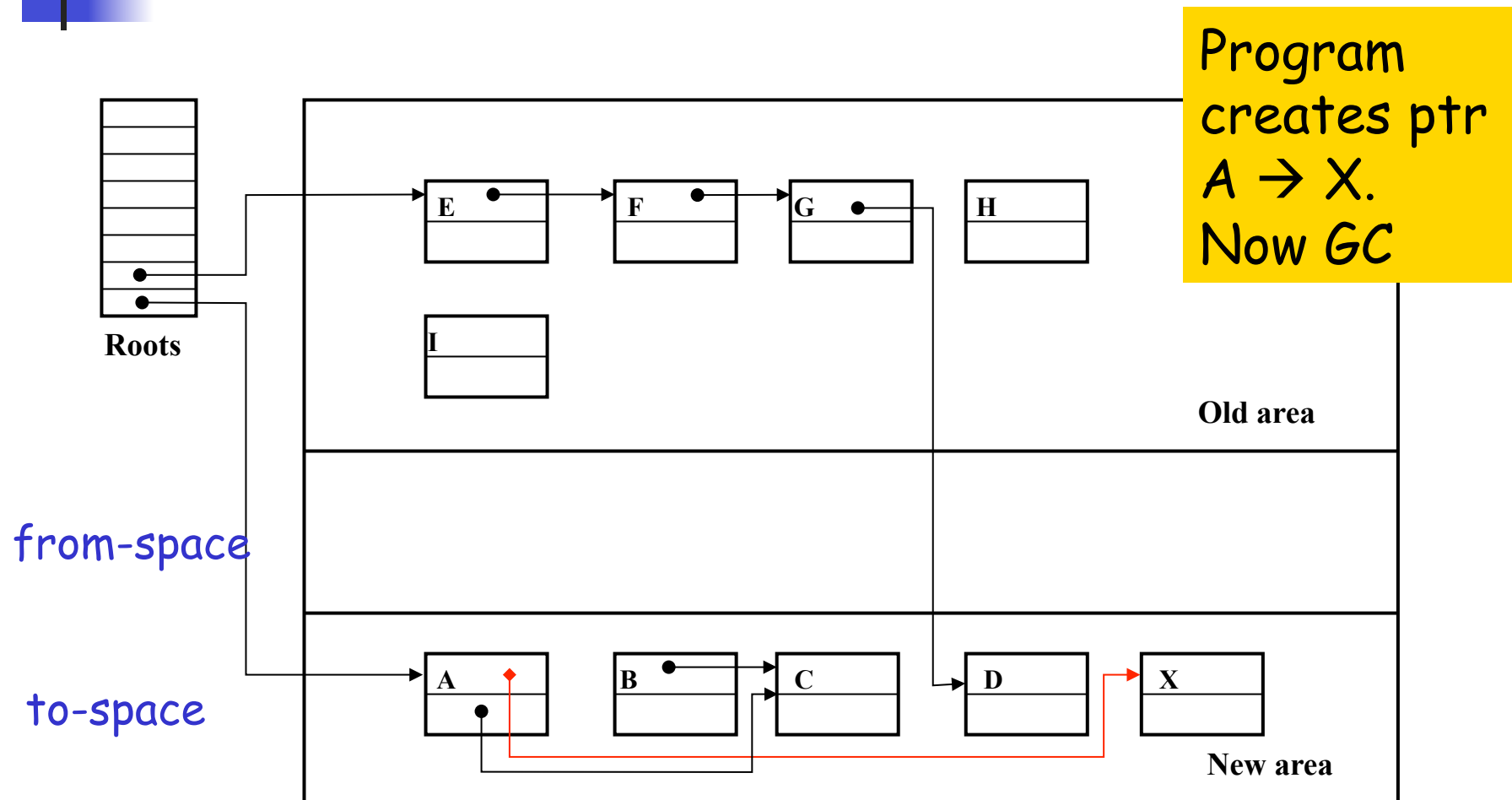
After GC



# Generational GC - Example

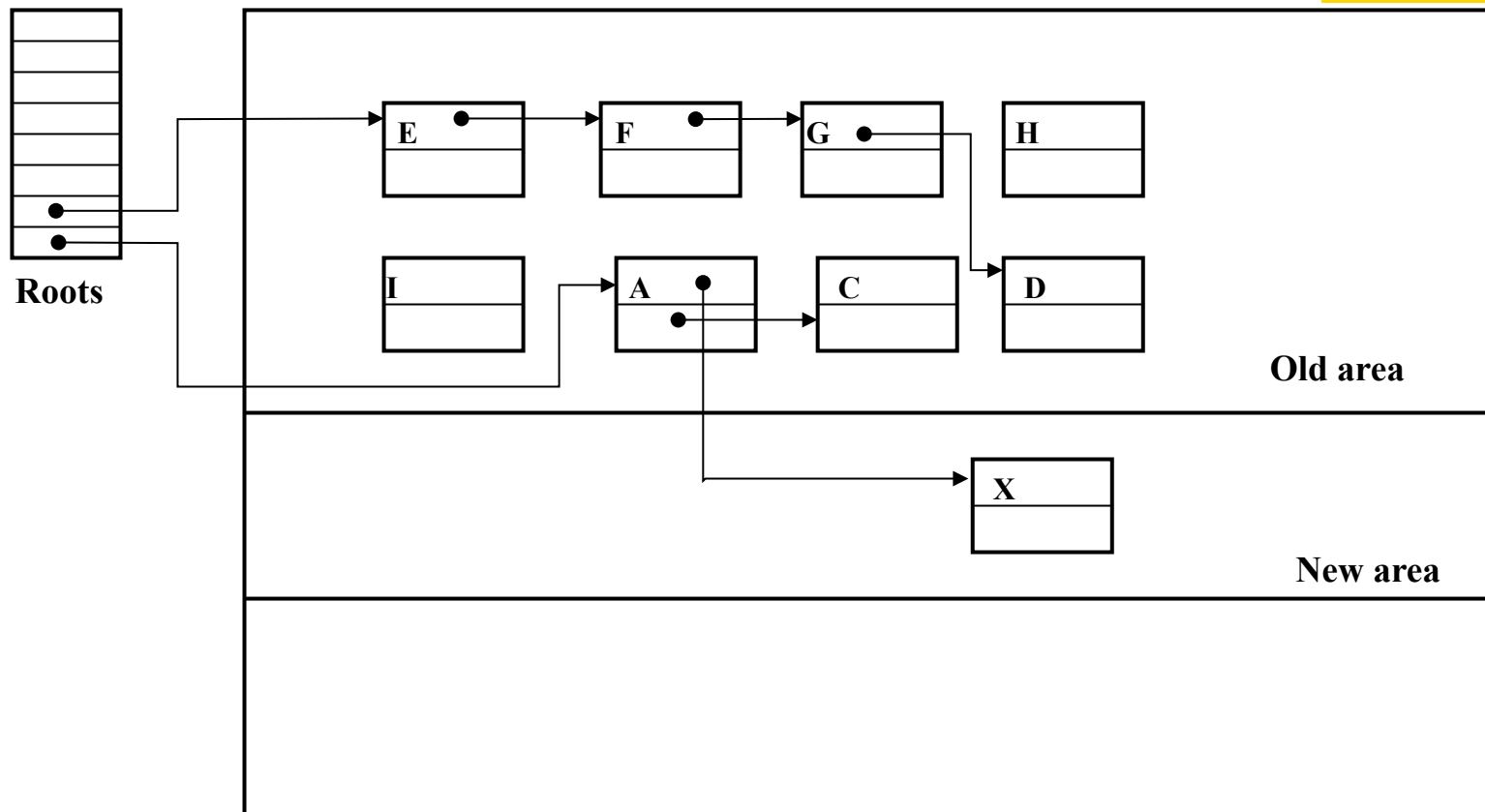


# Generational GC - Example



# Generational GC - Example

After GC





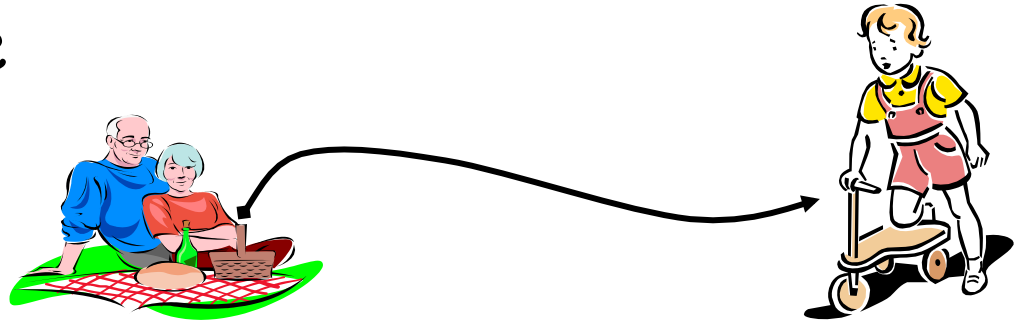
# Two Issues

---

1. **Inter-generational pointers** (main difficulty):
  - Pointers from objects in the old generation to objects in the young generation may witness the liveness of an object.  
If we naively trace the old generation we miss most of the benefits.
2. **Promotion policies**: when is an object promoted to the next generation?

# Issue I: Inter-Generational Pointers

- We will not trace through the old generation.
- The idea:
  - "Keep a list" of all inter-generational pointers.
  - Assume (conservatively) the parent (old) object is alive.
  - Treat these pointers as additional roots.
- "Typically": most pointers are from young to old (few from old to young).





# Old Collection = Full Collection

---

- When collecting the old generation we also collect the young generation.
  - Thus - we do not need to record (the many) pointers from young to old.
  - Cost is negligible since the young area is small and since full collections are infrequent.



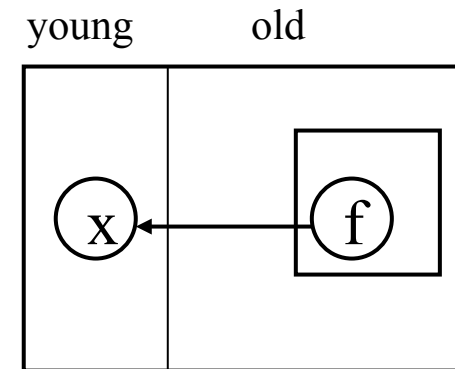
# Inter-Generational Pointers

---

- Inter-generational pointers are created:
  - When objects are promoted to old generation
  - When pointers are modified in the old generation.
- The first can be monitored by the collector during promotion.
- 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  $f \rightarrow x$ ;  
     $f := x$   
}
```





# 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 only; collect old and young generations together.



# 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



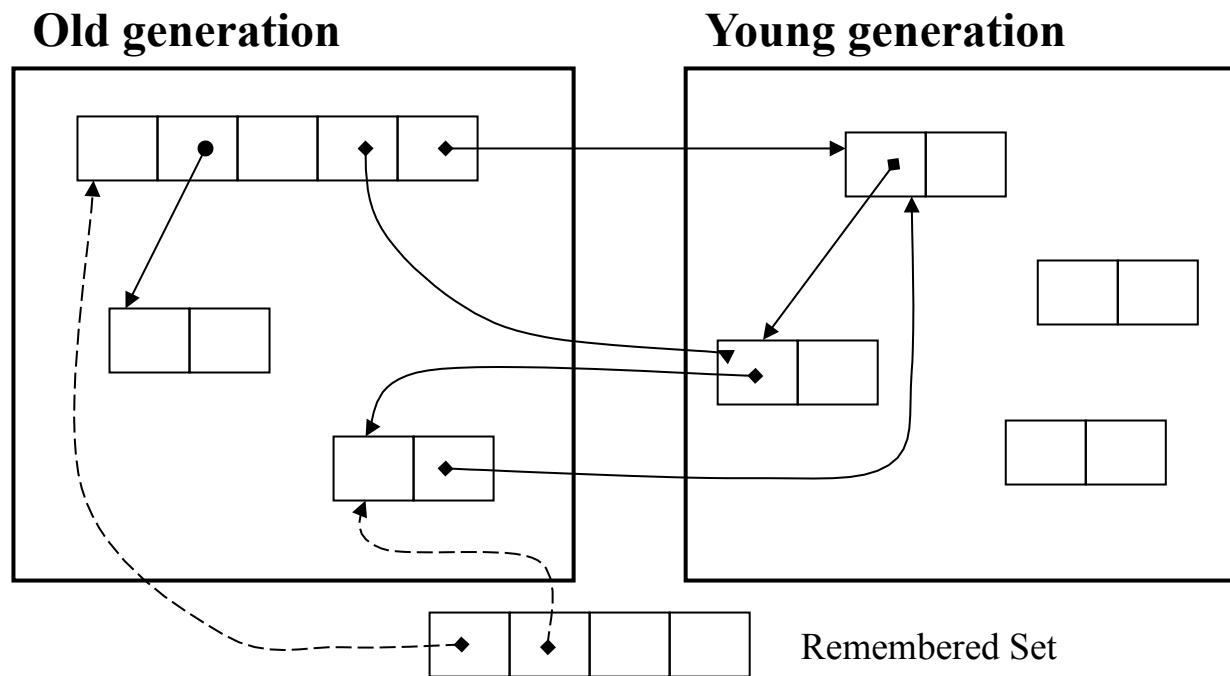
# Remembered Sets

---

- 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" → program suffers.



# Card Marking

---

- Heap divided into small regions called **cards**.
- Each card has a bit signifying whether an inter-generational pointer might exist on this card. (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.
- **Good for the program:**  
Write barrier is light → program runs faster.
- Can we combine the two methods and gain something from both?



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

---

- Program still does the same: a modified card is marked dirty by program.
- But the collector does not scan the same cards repeatedly.
- Instead: collector scans dirty cards, updates remembered sets & clears all dirty bits.
- Afterwards, remembered sets can be used for fast collection.
- Advantage:
  - Program uses fast write barrier.
  - Collector scans a card (at most) once per modification.



# Sequential Store Buffer (SSB)

---

- Idea: let collector do most of the work.
- Whenever the program modifies a pointer, the write barrier unconditionally adds the pointer address to the end of a buffer: 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

- Object Lifetimes:
  - Clock time.
    - Machine dependent.
    - Implementation dependent.
  - Count bytes of heap allocated.
    - Machine independent.
    - Better reflects the demand made upon the memory management sub-system.





# Tradeoffs: Size and Policies

---

- **Tradeoff 1 - size of the young generation:**
  - A small young generation means short pauses.
  - But, a small generation means more collections, (decreasing efficiency).
- **Tradeoff 2 - promotion policy:**
  - Promoting fast means more space in young generation and less collections.
  - But, promoting fast means more tenured dead objects that will stay in the heap for long.



# Dealing With The Tradeoff

---

- Multiple generations
- Promotion threshold
- Adaptive tenuring

# Multiple Generations



## + Pros:

- + Keep young generation small & promote fast.
- + Intermediate gen's lets 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 the generations, keeping remembered sets, etc.
- Pause time for collecting intermediate generations may still be disruptive



# Promotion Threshold

---

- Promotion by the number of times an object has survived a collection.
- 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

---

- We want to tenure **enough** objects so that the young generation gets enough free space, but **not too many** so that objects are not promoted young.
- A typical choice: promote objects so that 80% of the young generation space becomes free.
- **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]

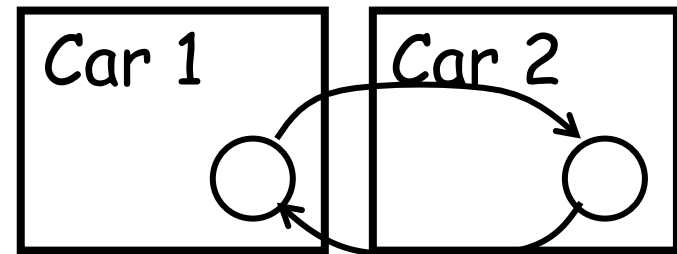
---

- 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!





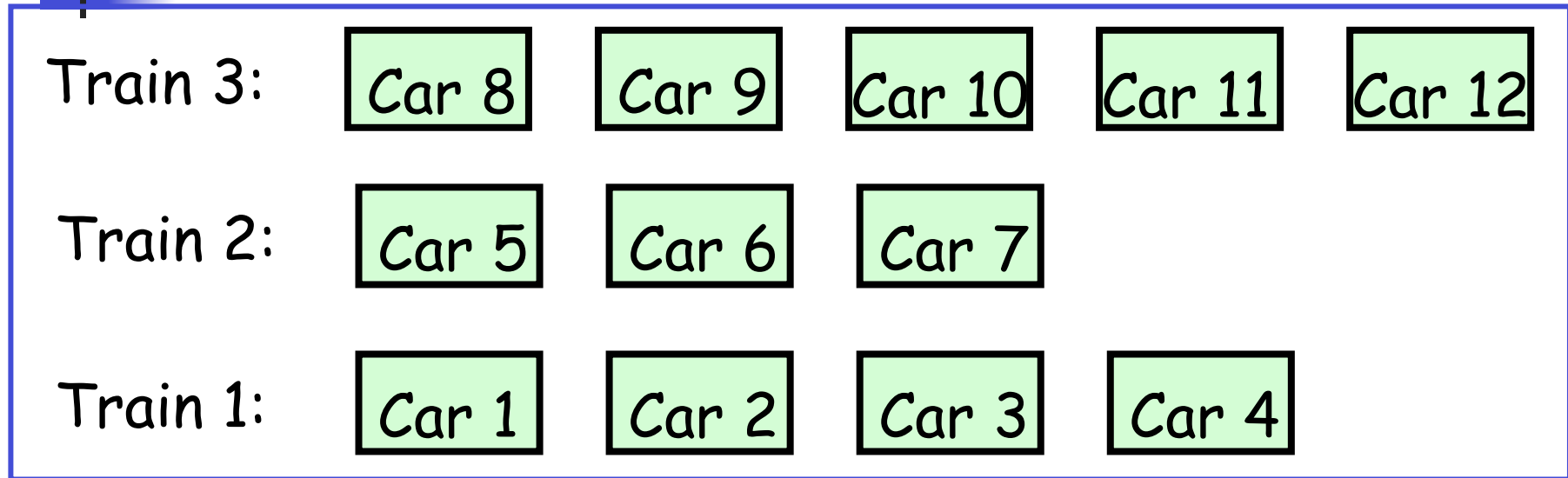
# Using Cars and Trains

---

- 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 1<sup>st</sup> car of 1<sup>st</sup> 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.



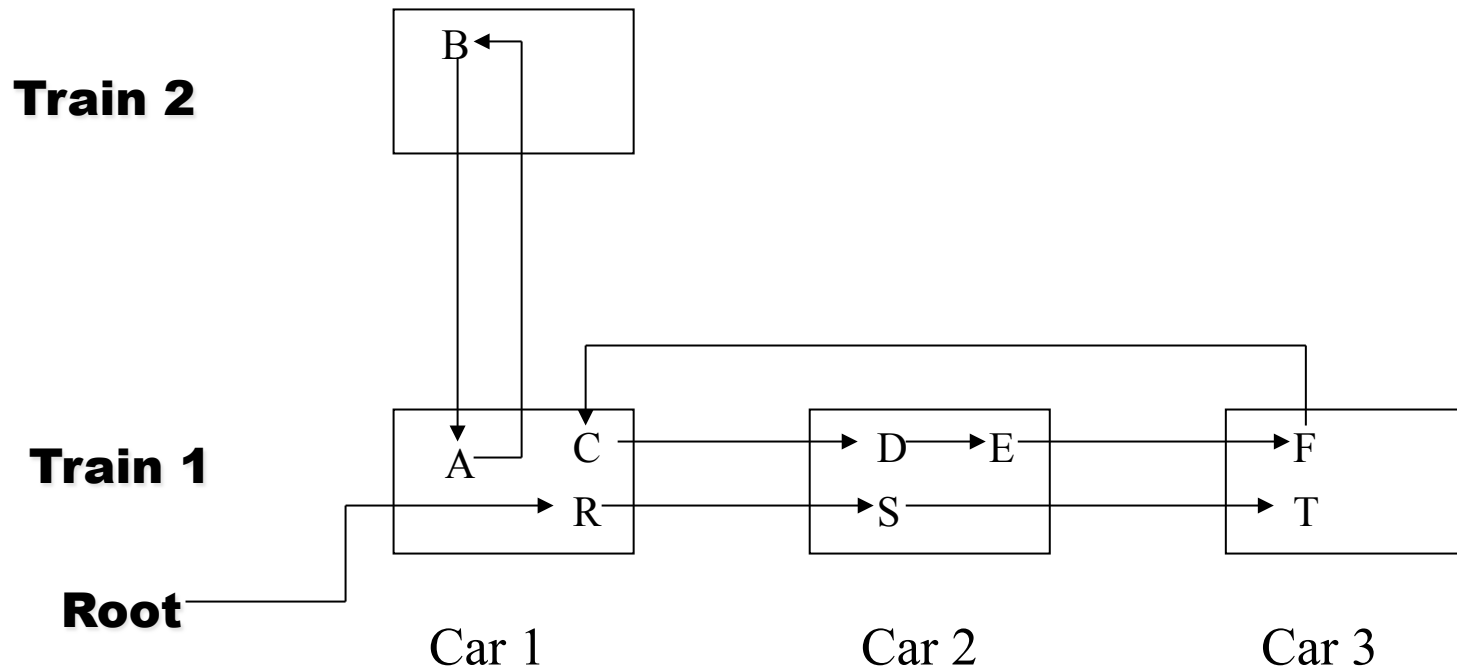
## Collection of Current Car - Cont'd

---

- 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.

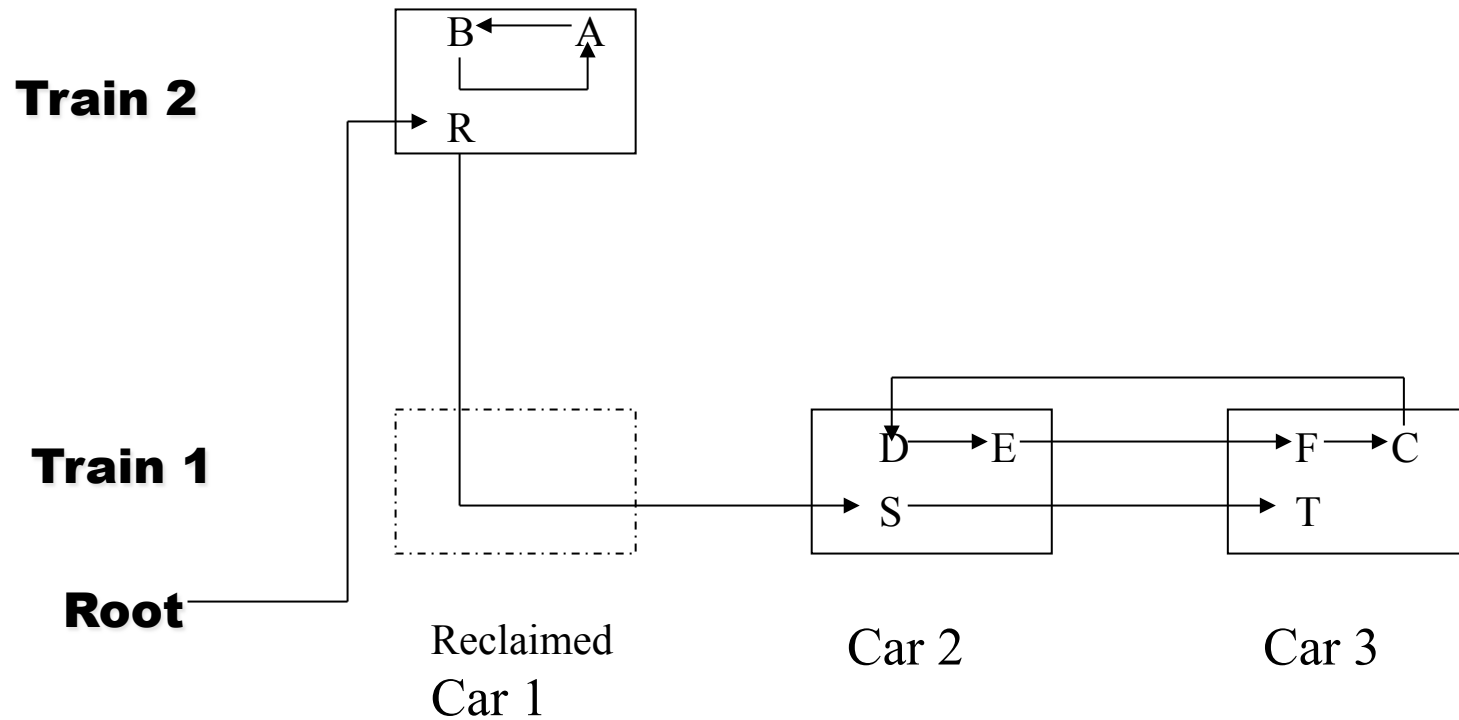
# Example

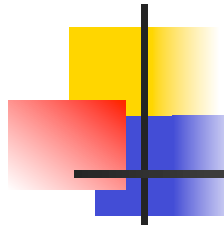
Say, max num of objects that can fit in the car is 3



# Example(continue)

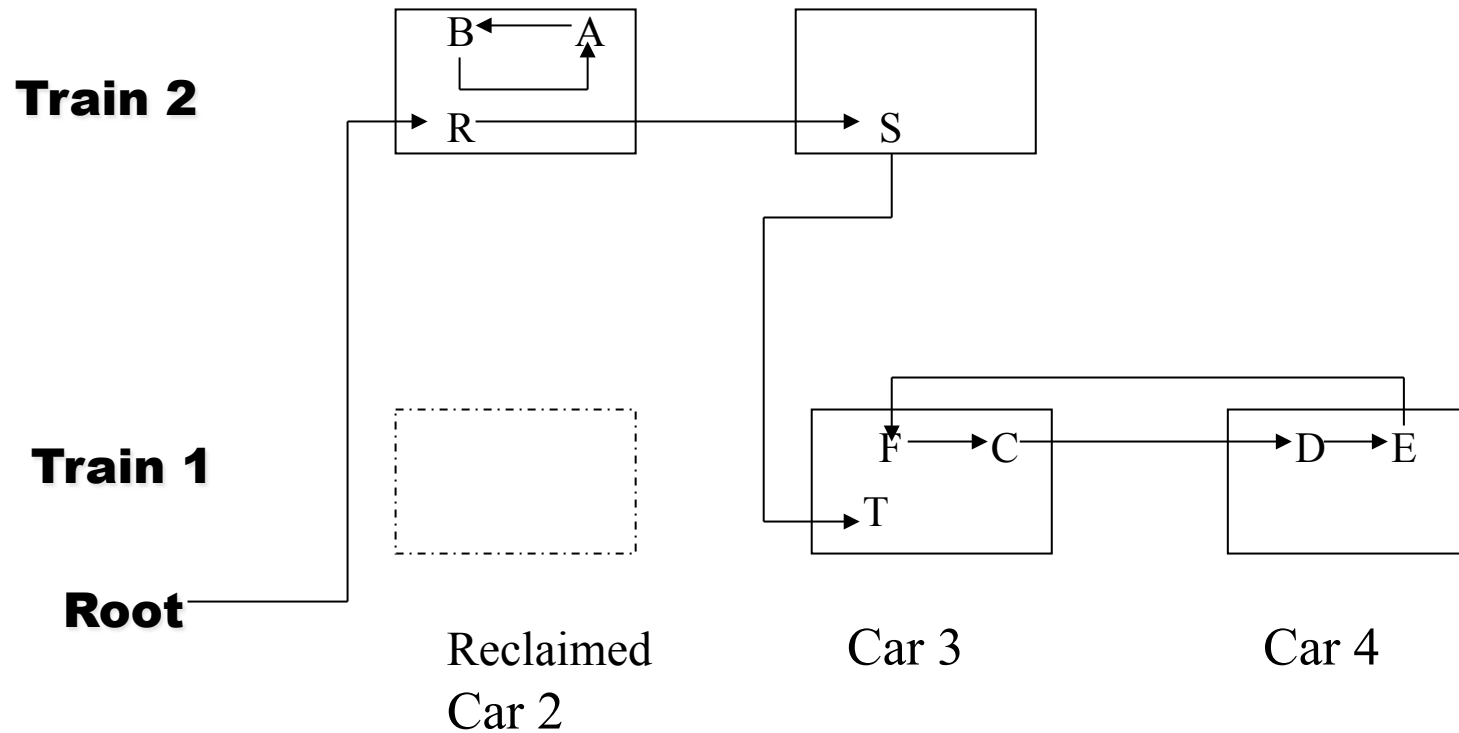
After first invocation of the algorithm





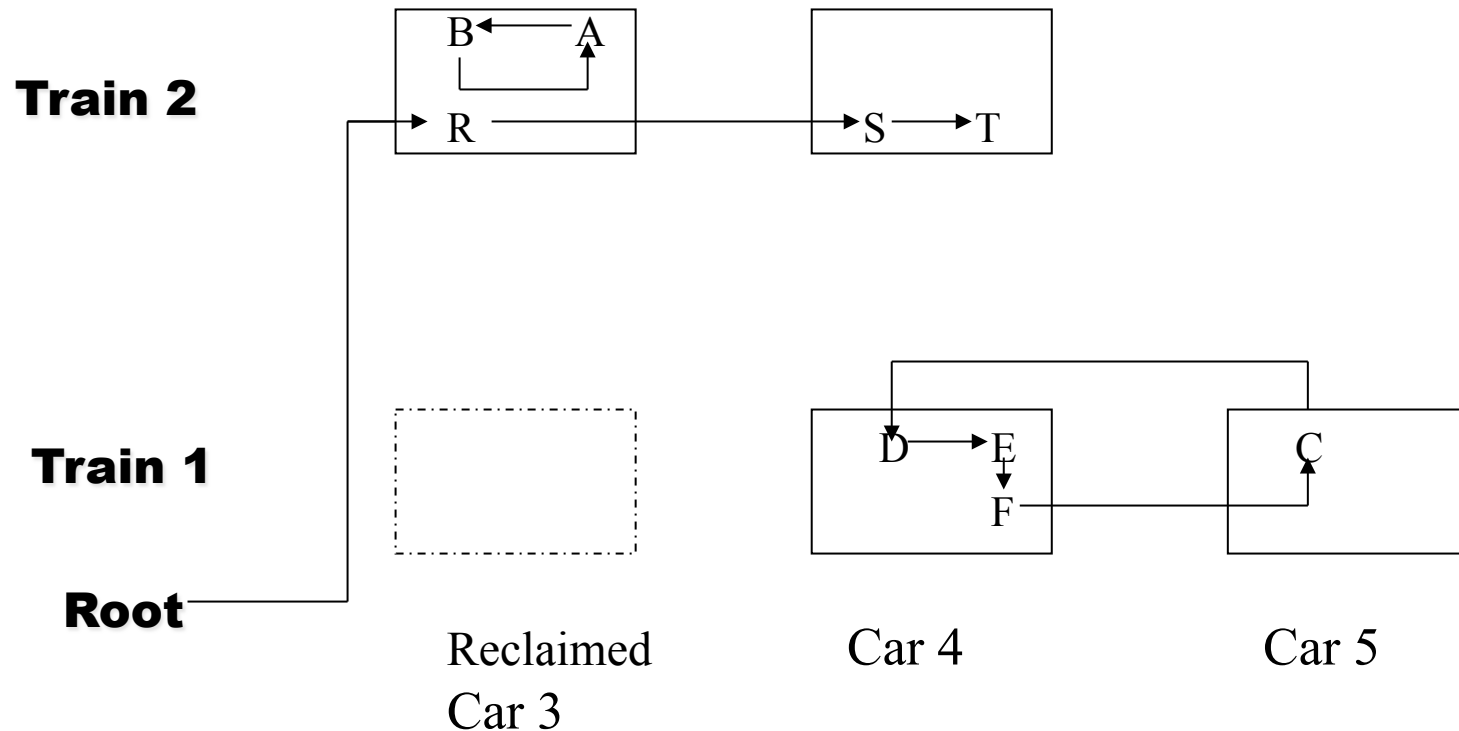
# Example (Continue)

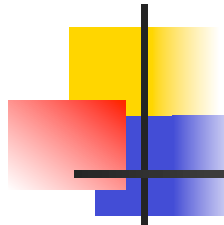
After second invocation of the algorithm



# Example (Continue)

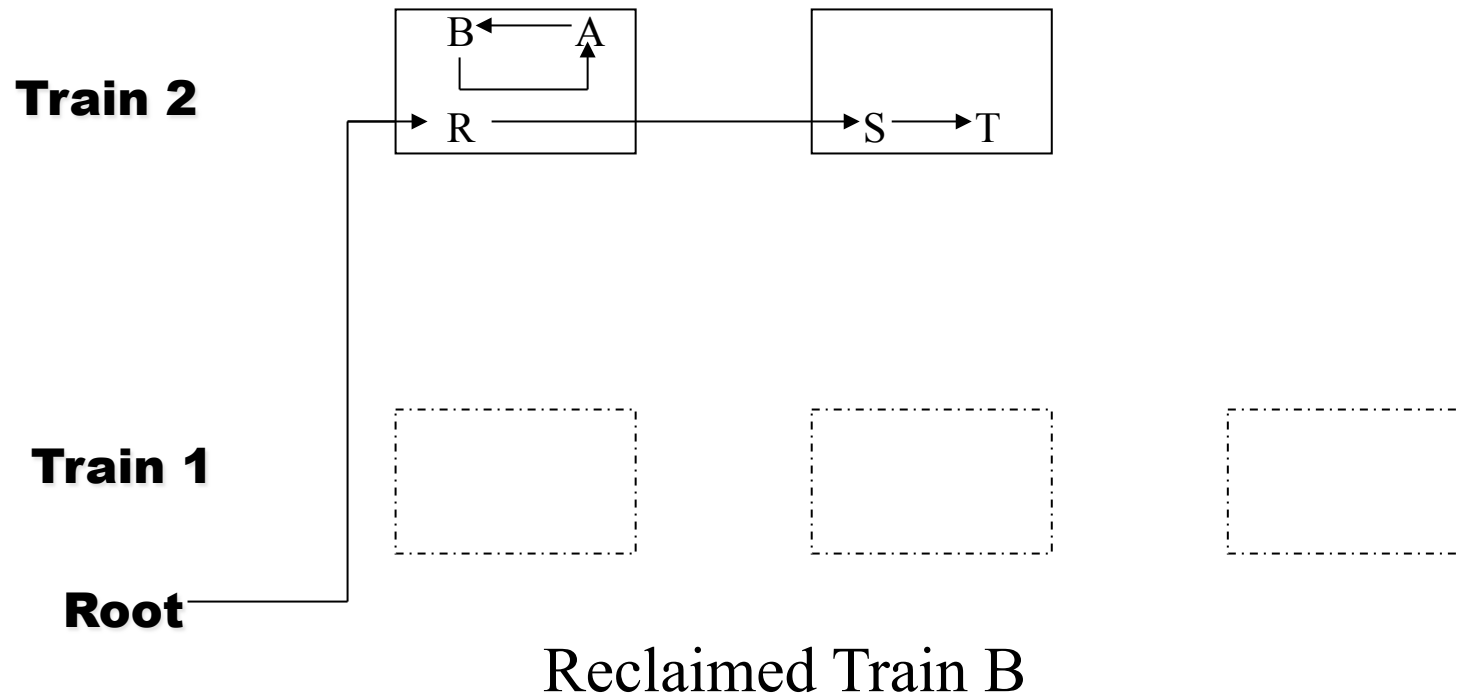
After third invocation of the algorithm

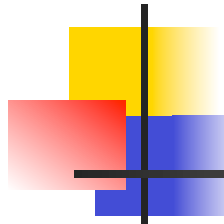




# Example (Continue)

In the middle of fourth invocation of the algorithm

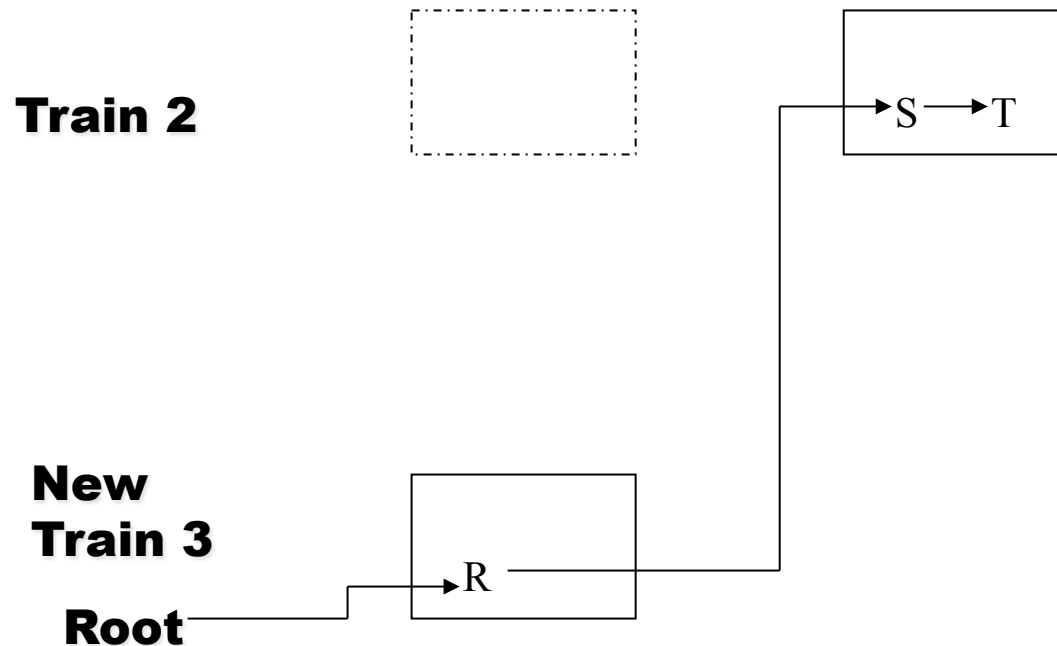


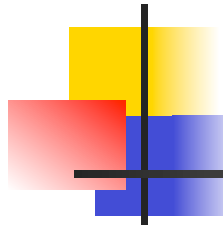


# Example (Continue)

---

After fourth invocation of the algorithm





# Example (Continue)

---

After fifth invocation of algorithm

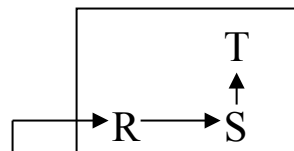
**Train 2**



Freed Train A

**New  
Train 3**

**Root**





# Properties

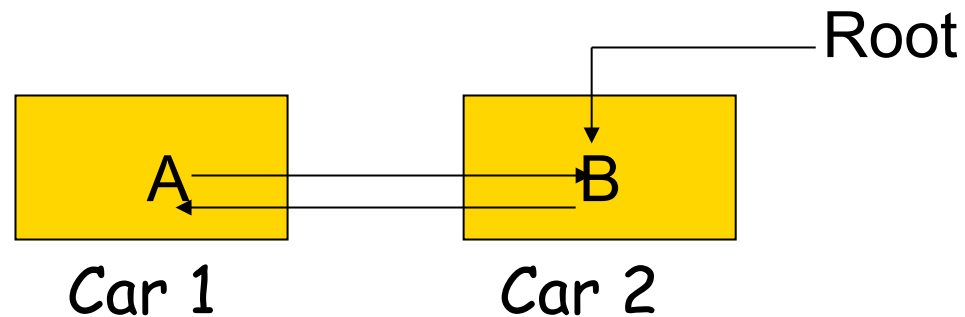
---

- 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]

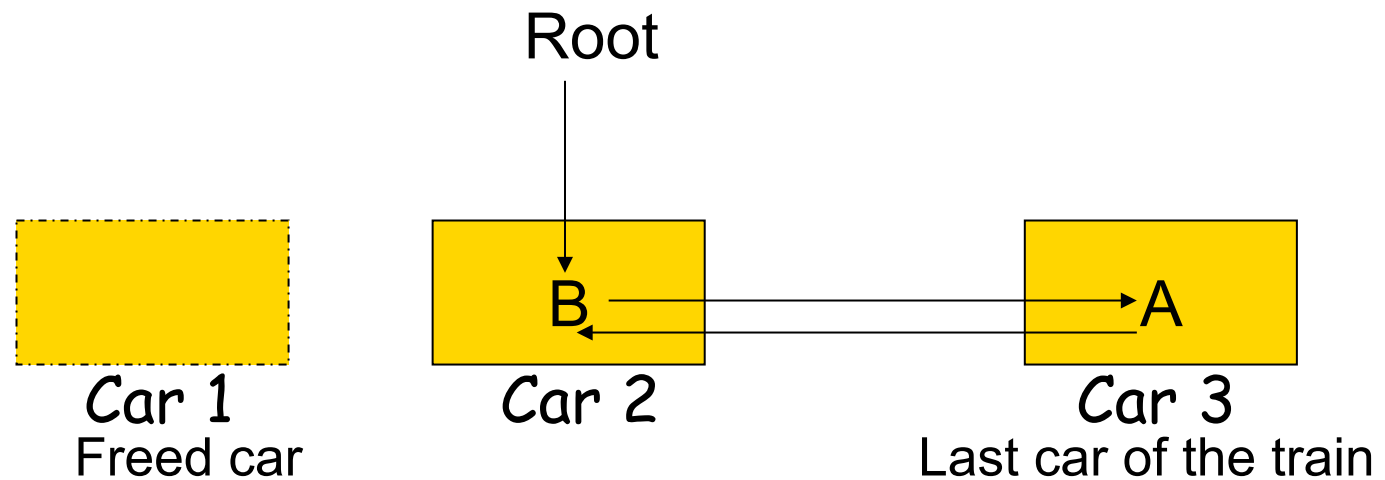
There is a (rare) scenario in which the algorithm never finishes handling a train.  
Consider the following situation:



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

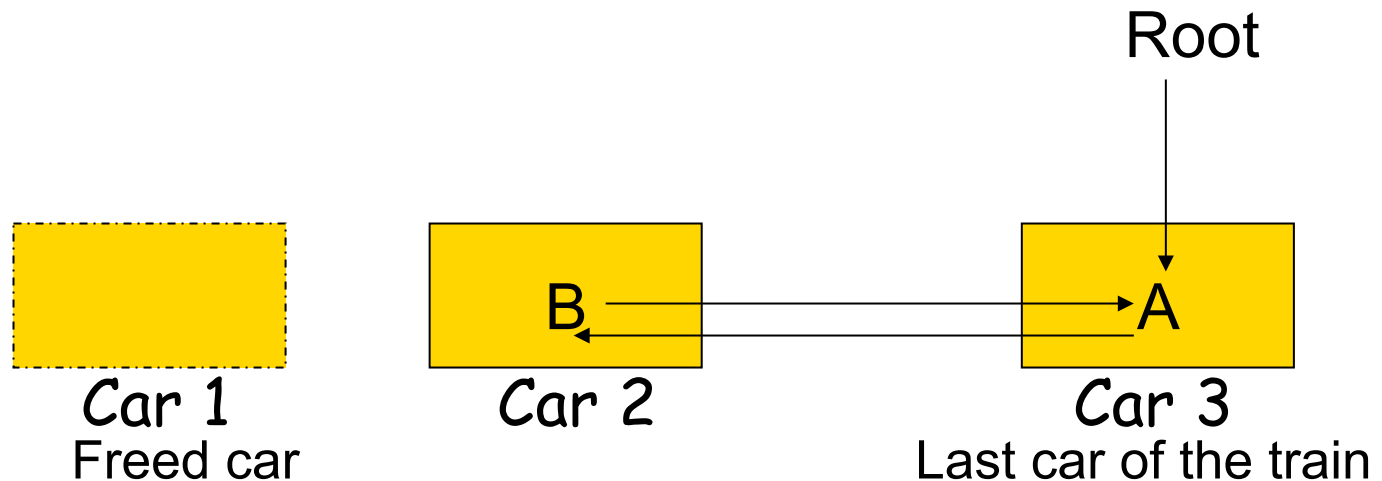
# A Termination Flaw - Cont'd

After collecting car 1, car 3 is created.



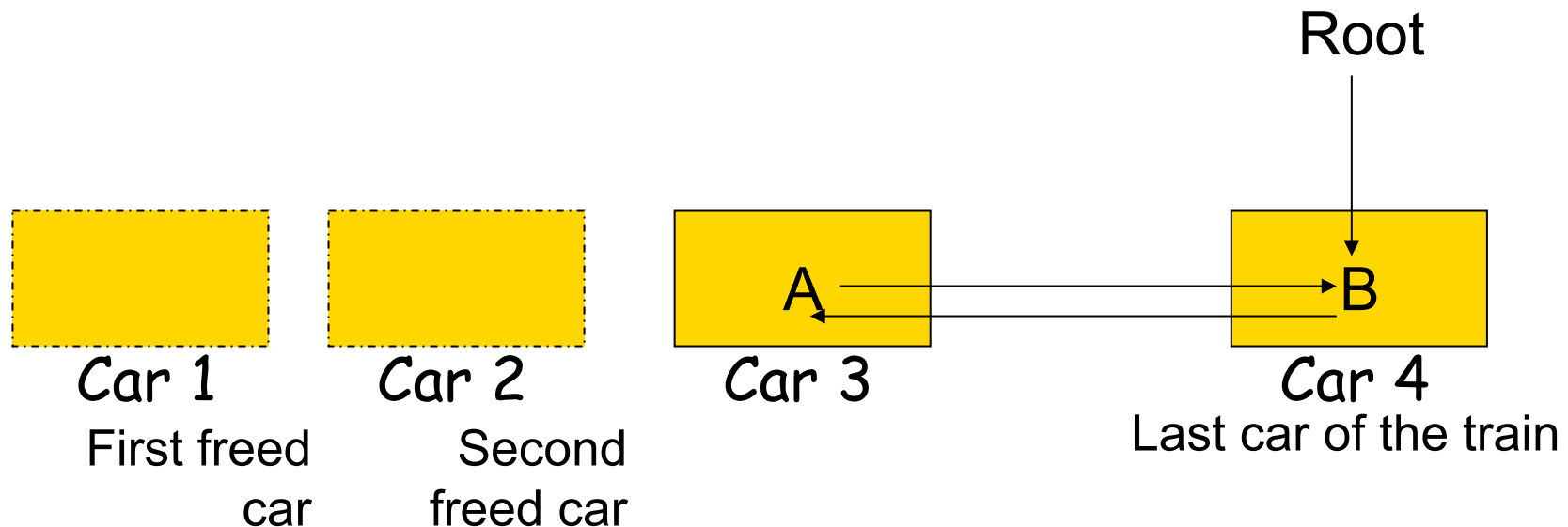
# A Termination Flaw - Cont'd

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.