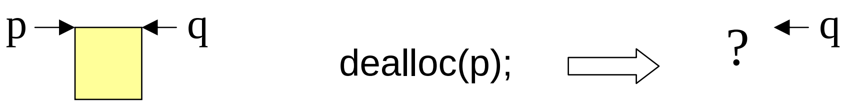
# 2. Garbage Collection

# *Motivation*

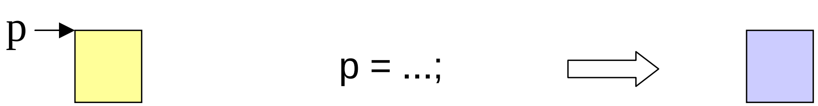
## Manual memory deallocation is error-prone (e.g. in C/C++)

### 1. deallocation too early




"dangling pointer"
q.f = ...;
destroys some other object

### 2. deallocation missing




"memory leak"
object becomes unreachable
but wastes space

## Garbage collection

A block is automatically reclaimed as soon as it is not referenced any more.


not referenced any more
=> reclaimed

first time 1960: Lisp
today in almost all lang.:
Java, C#,
Smalltalk, Eiffel,
Scheme, ...

+ safe (avoids too early or missing deallocation)
+ convenient (code becomes shorter and more readable)
- slower (run-time system must do bookkeeping about blocks)

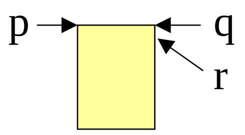
# *References (pointers)*

## When are new references created?

object creation: `p = new Person();`

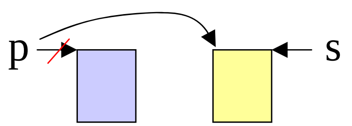
assignment: `q = p;`

parameter passing:
```
foo(p);
void foo(Object r) {...}
```

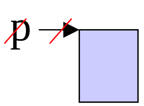



## When are references removed?

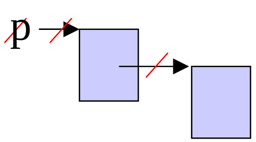
assignment: `p = s;`




death of local pointers
at the end of a method:
```
void foo(Object p) {
    ...
}
```




reclamation of an object
that contains pointers: `p = null;`



# *Garbage collection & object-orientation*

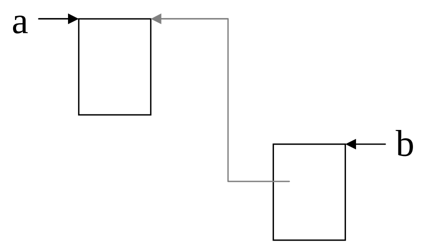
**Why is GC especially important in object-oriented languages?**

Due to information hiding one never knows by how many pointers an object is referenced.

**Example**

```
A a = new A();
B b = new B(a);
```

How many pointers reference the *A* object?




```
class B {
    private A p;

    public B(A a) { p = a; }
    ...
}
```

The *B* object has a private pointer to the *A* object.
Clients of *B* don't know that because *p* is private.

**Languages with GC**

Java, C#, Eiffel, Smalltalk,
Lisp, Scheme, Scala, Oberon, ...

**Languages without GC**

C, C++, Pascal, Fortran,
Cobol, ...

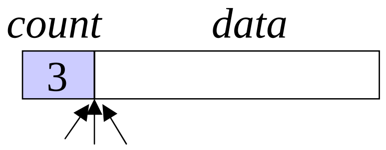
# 2. Garbage Collection

# *Reference Counting*

Oldest garbage collection technique (1960: Lisp)

## Idea

Every object has a counter holding the number of pointers that reference this object




*count* ... hidden counter field

As soon as the counter becomes 0 the object is deallocated

# *Counter management*

## Compiler generates code for updating the counters

**assignments**

```
p = q;
```

```
IncRef(q);
DecRef(p);
```

```
if (q != null) q.count++;
```

```
if (p != null) {
    p.count--;
    if (p.count == 0) dealloc(p);
}
```


**parameters & local variables**

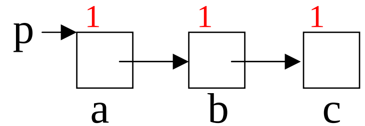
```
void foo(Object p) {
    ...
}
```

```
IncRef(p);
```

```
DecRef(p);
```


**deallocation of objects**

```
void dealloc(Object p) {
    for (all pointers q in *p) DecRef(q);
    ...
}
```




p = null; ⇨ a.count--;
b.count--;
c.count--;

# *Strengths*

+ Unreferenced blocks are immediately reclaimed
  (no delay like in other GC algorithms)

+ GC does not cause major pauses
  (GC overhead is evenly distributed over the whole program)

+ GC can be done incrementally

*DecRef(p)*

```
if (p != null) {
    p.count--;
    if (p.count == 0) worklist.add(p);
}
```

*background thread*

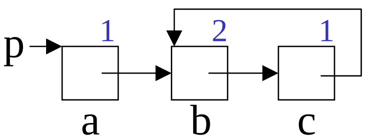
```
while (!worklist.empty()) {
    p = worklist.remove();
    dealloc(p);
}
```

# *Weaknesses*

- Pointer assignments and parameter passing impose some overhead
  (GC costs are proportional to the number of assigments, even if there is no garbage)

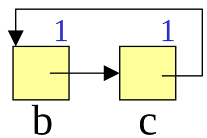
- Counters need space in every object (4 bytes)

- Does not work for cyclic data structures!




p = null;

⇒ a.count--;
b.count--;




these counters never become 0
⇒ objects are never deallocated

**Possibilities for dealing with cyclic data structures**

- ignore them (if there is sufficient memory)
- require the programmer to break the cycle (b.next = null;)
- try to detect the cycles (expensive)

# *Cycle detection* *(Lins92)*

decRef(p)

```
p.count--;
if (p.count == 0)
    dealloc(p);
else
    mark p for lazy garbage collection;
```

**Every node has one of the following colors:**

| | |
|---|---|
| **black** | still referenced => keep it |
| **white** | unreferenced => deallocate it |
| **grey** | under inspection by the GC |
| **red** | marked to be inspected by the GC |

```
void decRef(Obj p) {
    p.count--;
    if (p.count == 0) {
        p.color = white;
        for (all sons q) decRef(q);
        dealloc(p);
    } else if (p.color != red) {
        p.color = red;
        gcList.add(p);
    }
}
```

```
void incRef(Obj p) {
    p.count++;
    p.color = black;
}
```

From time to time do a garbage collection on the mark list

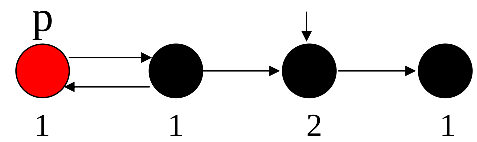
```
void gc() {
    do
        p = gcList.remove()
    until (p == null || p.color == red);
    if (p != null) {
        mark(p);
        sweep(p);
        collectWhite(p);
    }
}
```

# *Cycle detection (continued)*

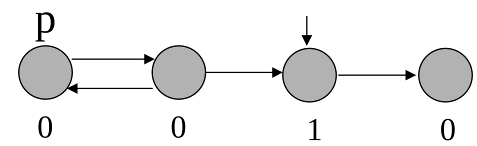
Make all referenced objects grey

```
void mark(Obj p) {
    if (p.color != grey) {
        p.color = grey;
        for (all sons q) { q.count--; mark(q); }
    }
}
```

before mark(p)



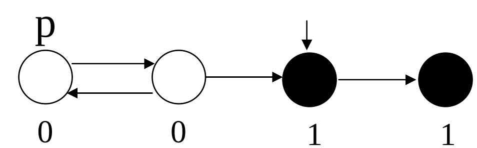

after mark(p)




Make all unreferenced objects white

```
void sweep(Obj p) {
    if (p.color == grey) {
        if (p.count == 0) {
            p.color = white;
            for (all sons q) sweep(q);
        } else restore(p);
    }
}
```

after sweep(p)




```
void restore(Obj p) {
    p.color = black;
    for (all sons q) {
        q.count++;
        if (q.color != black) restore(q);
    }
}
```

deallocate white objects

```
void collectWhite(Obj p) {
    if (p.color == white) {
        p.color = grey;          <-- to break cycles in collectWhite
        for (all sons q) collectWhite(q);
        dealloc(p);
    }
}
```

to break cycles in collectWhite

# *Where is reference counting used?*

- **In some interpreted languages where run time efficiency is not an issue**
  Lisp, PHP, ...

- **For managing references to distributed objects** (COM, CORBA, ...)

  e.g. COM

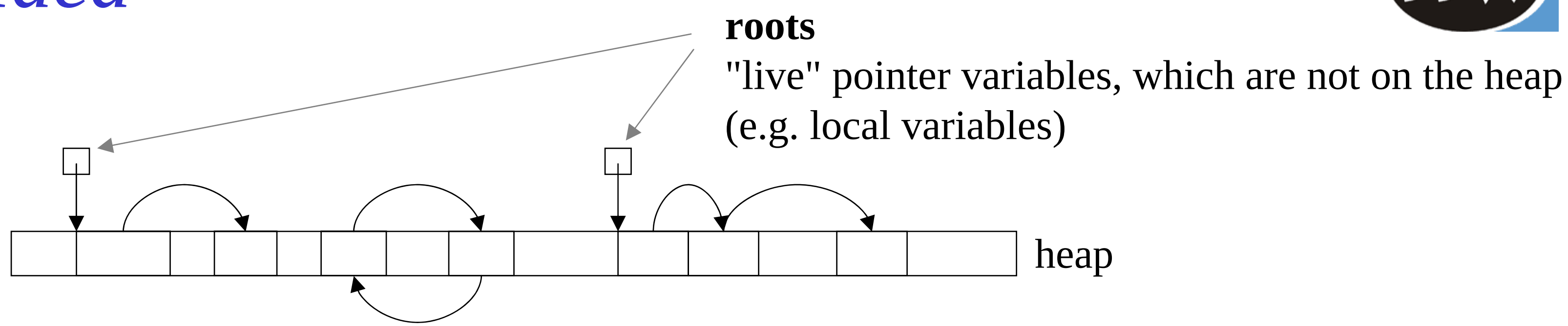
  | | |
  |---|---|
  | obj->AddRef(); | is called automatically by COM if a reference to an object (interface) is created |
  | obj->Release(); | must be called by the programmer if a reference is not needed any more |

- **For managing references (links) to files** (Unix, ...)

  A file must not be deleted if it is still referenced by a link

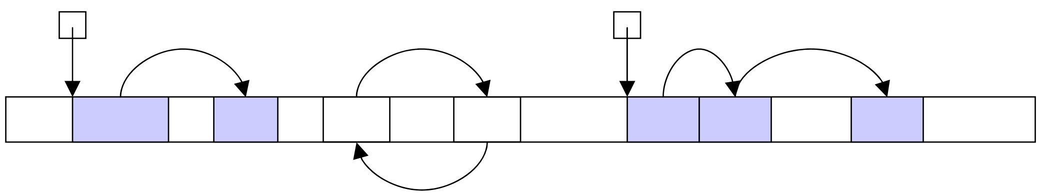
# 2. Garbage Collection

# *Idea*

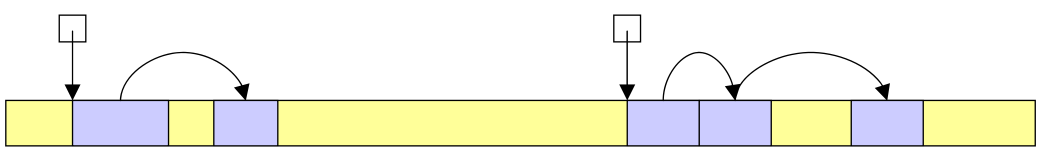



**2 phases** start when the heap is exhausted

**1. Mark**: Mark all objects that can be reached from roots (directly or indirectly)



**2. Sweep**: Traverse the heap sequentially and reclaim unmarked objects



# *Pros and Contras*

**Advantages**

+ No overhead in assignments and method calls.
  Compiler does not have to generate code for managing reference counters.
+ Needs only 1 mark bit per object (instead of a 4 byte counter field)
+ Cyclic data structures are handled correctly

**Disadvantages**

- Noticeable pause while the GC runs (problematic for real-time systems)
- Unfavorable for large heaps:
  - Sweep time is proportional to the heap size.
  - Mark time is proportional to the number of live objects
- No compaction (heap remains fragmented)
- Related objects are often spread over the whole heap (poor cache behavior).

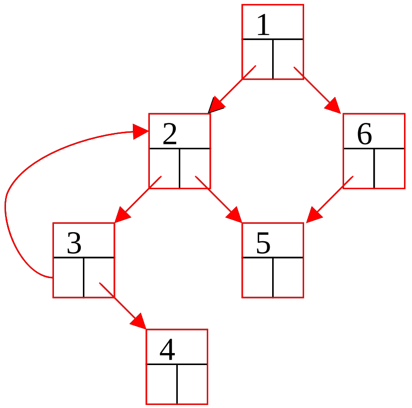
# *Naive implementation*

**Simplistic assumption**
all objects have the same type

```
class Block {
    boolean marked;
    Block left, right;
    ...  // data
}
```

**Mark algorithm**
recursive depth-first traversal

```
void mark (Block p) {
    if (p != null && !p.marked) {
        p.marked = true;
        mark(p.left);
        mark(p.right);
    }
}
```




Is this algorithm practical?

No!
Deep recursion might lead to stack overflow

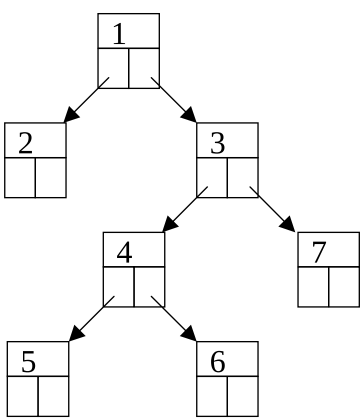
# *Deutsch-Schorr-Waite algorithm*

*Schorr, Waite: An efficient machine-independent procedure for garbage collection in various list structures, Communications of the ACM, Aug. 1967*

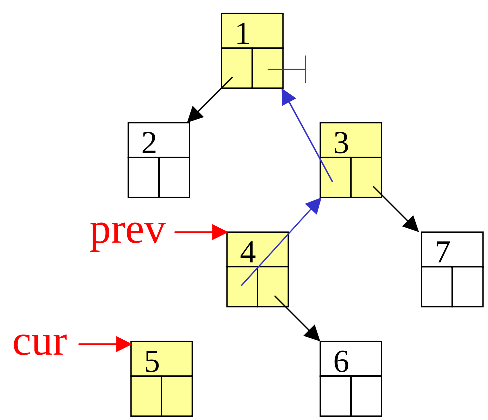
## Idea

- Pointers are followed iteratively not recursively
- The backward path is stored in the pointers themselves!

## Example




State while visiting node 5




cur: currently visited node
prev: predecessor of *cur*; node in which the backward chain starts

- No connection between *cur* and *prev*!
- One has to remember, whether the backward chain starts in *left* or in *right*

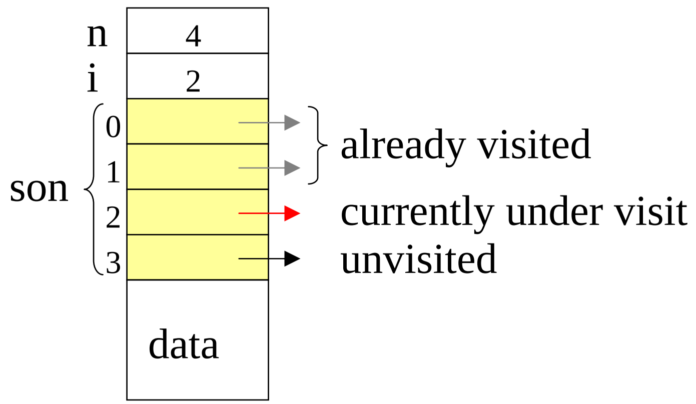
# *Objects with arbitrary number of pointers*

**Simplified assumption**

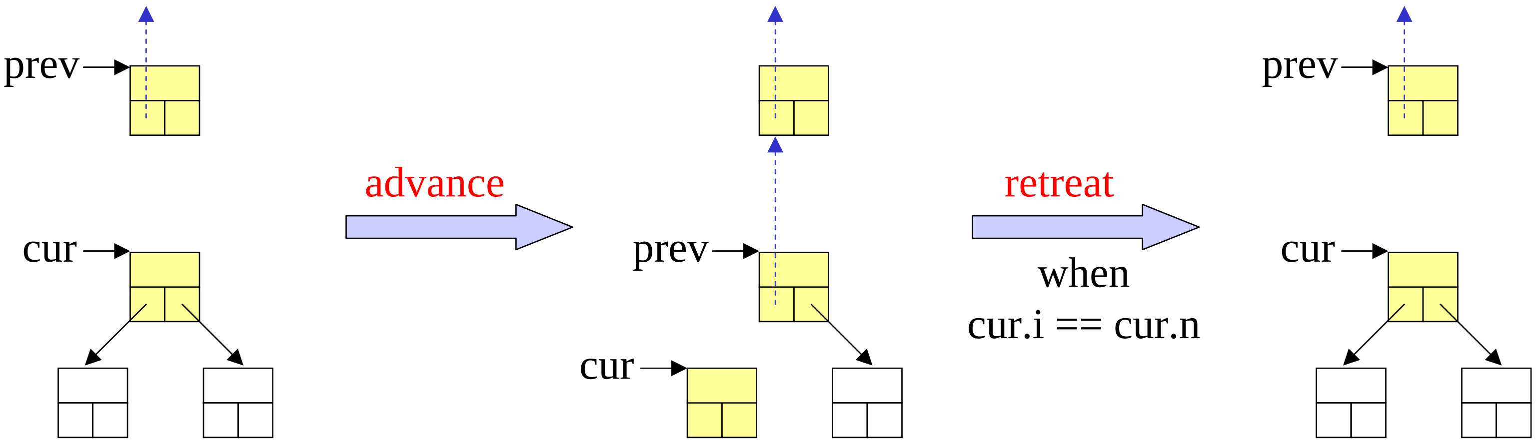
all pointers are stored in an array

```
class Block {
    int n;          // number of pointers
    int i;          // index of currently visited pointer
    Block[] son;    // pointer array
    ...             // data
}
```

← i == -1
⇒ block is still unvisited;
used for marking



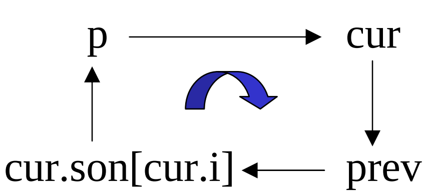
# Steps of the DSW algorithm



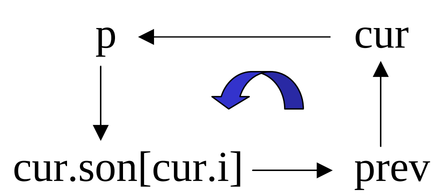

```
p = cur.son[cur.i];
cur.son[cur.i] = prev;
prev = cur;
cur = p;
```

```
p = cur;
cur = prev;
prev = cur.son[cur.i];
cur.son[cur.i] = p;
```

*pointer rotation*




*pointer rotation*



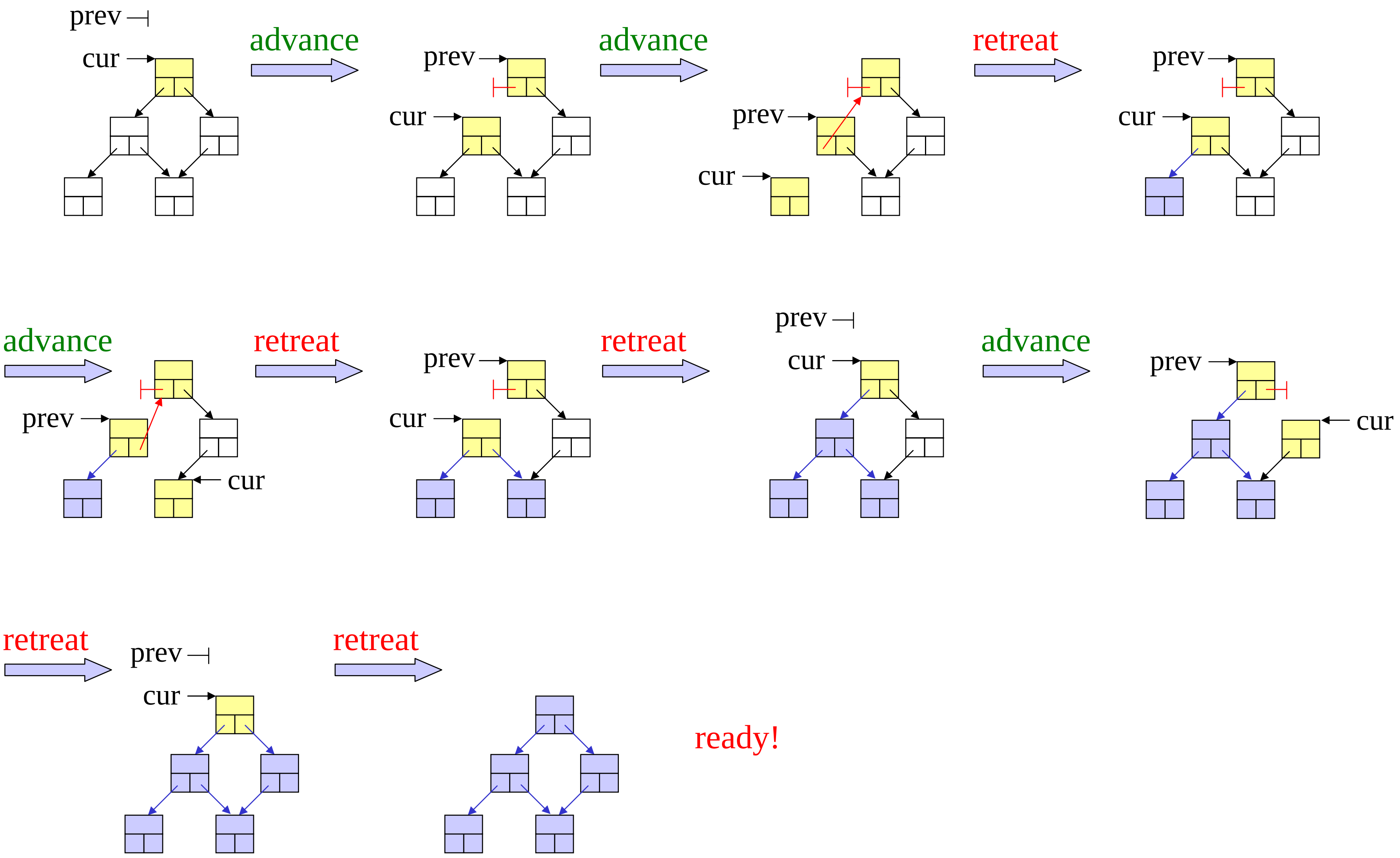
# *DSW algorithm*

```
void mark (Block cur) {
    // assert cur != null && cur.i < 0
    Block prev = null;
    for (;;) {
        cur.i++;  // mark
        if (cur.i < cur.n) { // advance
            Block p = cur.son[cur.i];
            if (p != null && p.i < 0) {
                cur.son[cur.i] = prev;
                prev = cur;
                cur = p;
            }
        } else { // retreat
            if (prev == null) return;
            Block p = cur;
            cur = prev;
            prev = cur.son[cur.i];
            cur.son[cur.i] = p;
        }
    }
}
```

*mark(p)* is called for every root pointer *p*

- Needs only memory for 3 local variables (*cur*, *prev*, *p*)
- No recursion
- Can traverse arbitrarily complex graphs

# *Example*

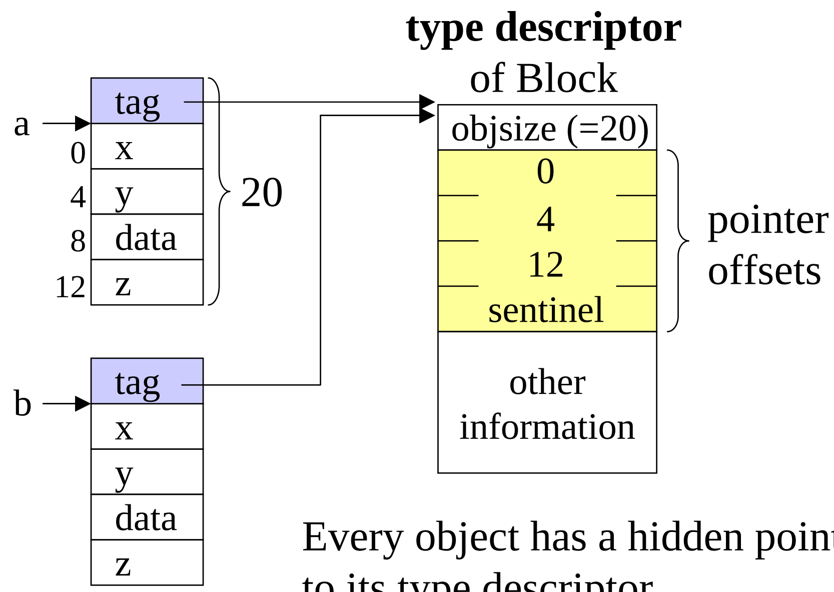

prev
cur
advance
prev
cur
advance
prev
cur
retreat
prev
cur
advance
prev
cur
retreat
prev
cur
retreat
prev
cur
advance
prev
cur
retreat
prev
cur
retreat
ready!

# Type descriptors

Allow pointers to be at arbitrary locations in an object

```
class Block {
    Block x;
    Block y;
    int data;
    Block z;
}
```

```
Block a = new Block();
Block b = new Block();
```

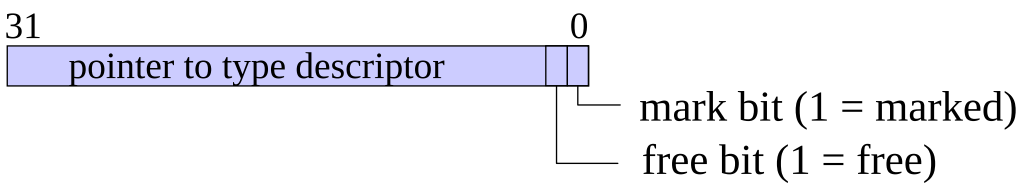



Every object has a hidden pointer (*tag*) to its type descriptor

- type descriptors are generated by the compiler for all classes; they are written to the object file
- the loader allocates the type descriptors on the heap
- *new Block()* allocates an object and installs in it a pointer to the corresponding type descriptor

# *Type tags*

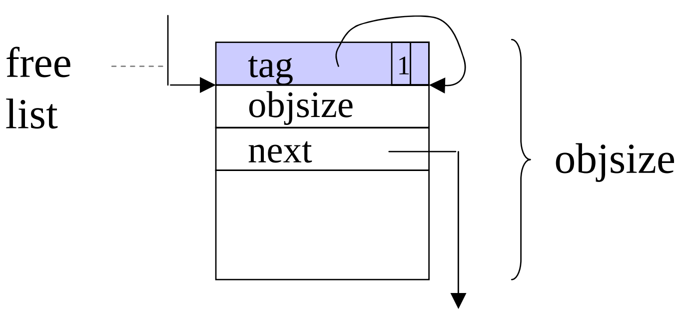
## Format of a type tag




- Type descriptors are 4 byte aligned (least significant 2 bits are 0)
- When GC is not running, the mark and free bits are guaranteed to be 0
- When GC is running, the mark and free bits have to be masked out

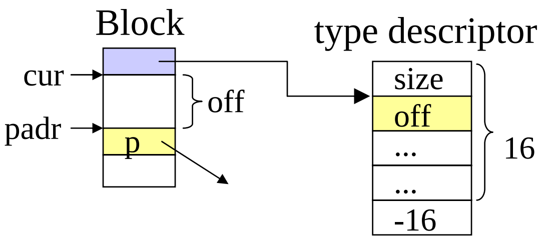
## Pseudo type descriptors for free blocks

are directly stored in the free block




In this way the block size of free and used objects can be uniformly accessed via *tag.objsize*.

# *Using the pointer offsets in mark()*




Tag is "abused" for pointing to the offset of the current son during *mark()*

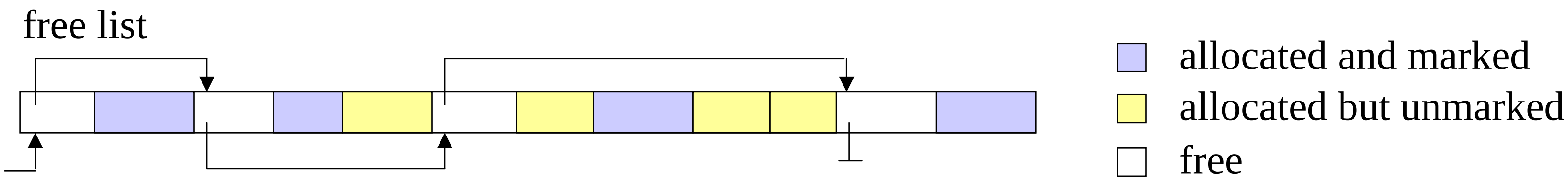
```
void mark (Pointer cur) { // assert: cur != null && !cur.marked
   prev = null; setMark(cur);
   for (;;) {
      cur.tag += 4;
      off = memory[cur.tag];
      if (off >= 0) {  // advance
         padr = cur + off; p = memory[padr];
         if (p != null && !p.marked) {
            memory[padr] = prev; prev = cur; cur = p;
            setMark(cur);
         }
      } else {  // off < 0: retreat
         cur.tag += off;  // restore tag
         if (prev == null) return;
         p = cur; cur = prev; off = memory[cur.tag]; padr = cur + off;
         prev = memory[padr]; memory[padr] = p;
      }
   }
}
```

This is the GC of Oberon

- 4 bytes overhead per object
- any number of pointers per object
- pointers may be at arbitrary positions
- fixed memory requirements

# *Sweep phase*

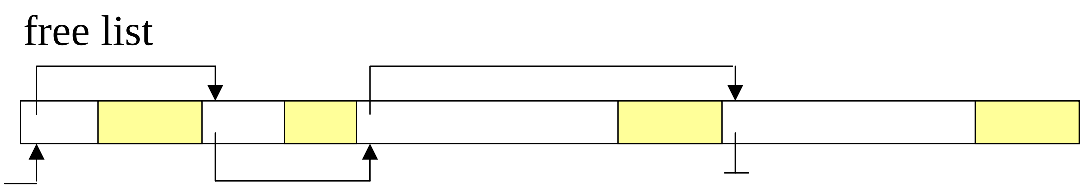
## Heap after the mark phase



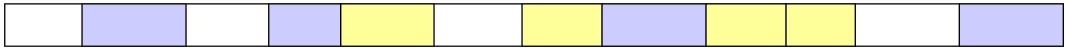

## Tasks of the sweep phase

- traverses all heap blocks sequentially
- merges adjacent free blocks
- builds a new free list
- clears the mark bits

## Heap after the sweep phase

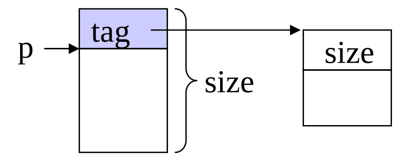


# Sweep algorithm

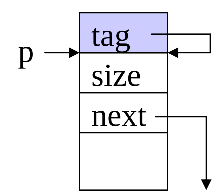


```
void sweep() {
    Pointer p = heapStart + 4;
    Pointer free = null;
    while (p < heapEnd) {
        if (p.marked) p.marked = false;
        else {  // free: collect p
            int size = p.tag.size;
            Pointer q = p + size;
            while (q < heapEnd && !q.marked) {
                size += q.tag.size; // merge
                q = q + q.tag.size;
            }
            p.tag = p; p.tag.size = size;
            p.tag.next = free; free = p;
        }
        p += p.tag.size;
    }
}
```

**allocated block**




**free block**




When *p.tag* is accessed the free bit must be masked to be 0 in free blocks

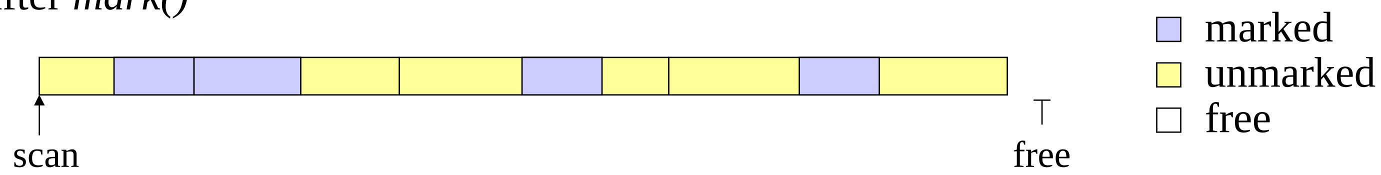
# *Lazy sweep*

## Problems

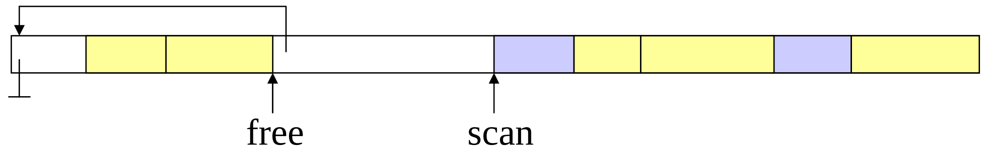
- Sweep must visit every block (takes some time if the heap is hundreds of megabytes large)
- In virtual memory systems any swapped pages must be swapped in and later swapped out again

**Lazy sweep** Sweep is done incrementally on demand

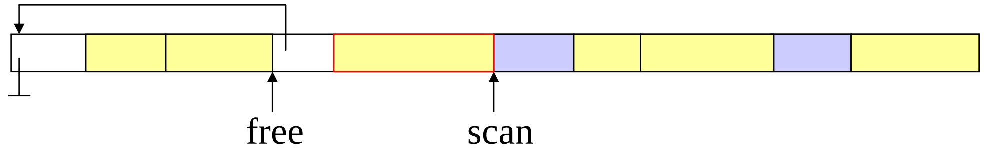
after *mark()*



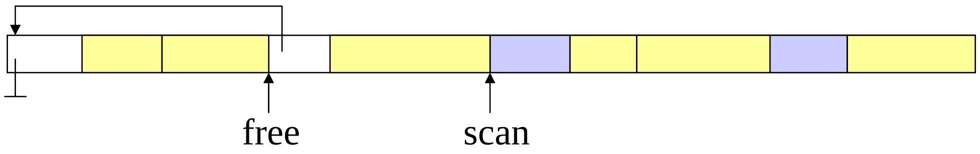

p = alloc(size); no block in free list ⇒ partial sweep



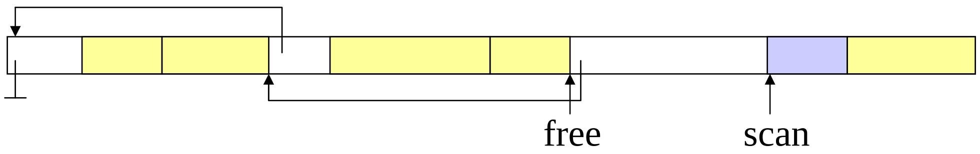

until a sufficiently large block is freed and *alloc()* can proceed



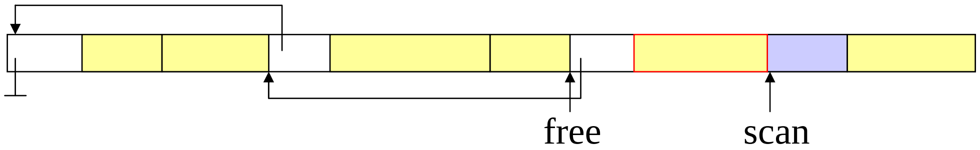
# *Lazy sweep (continued)*




p = alloc(size); no sufficiently large block in free list ⇒ partial sweep




*alloc()* can proceed




## Requirements

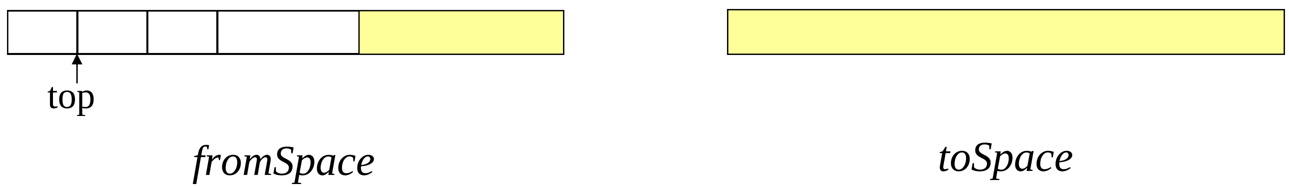
- Mark bits remain set while the program (the mutator) runs (they must be masked out when the type tag is accessed)
- *mark()* must only be restarted after the whole sweep has ended

# 2. Garbage Collection

# *Stop & Copy*

**The heap is divided into two parts: *fromSpace* and *toSpace***

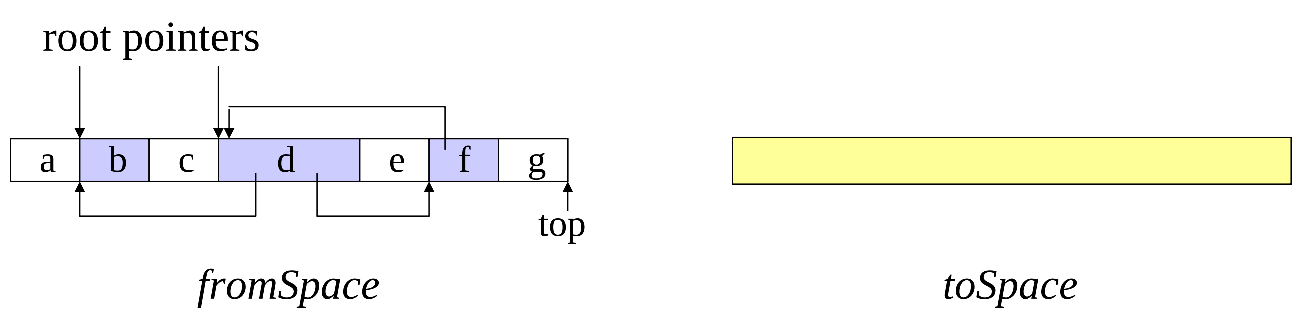
**New objects are allocated in *fromSpace***



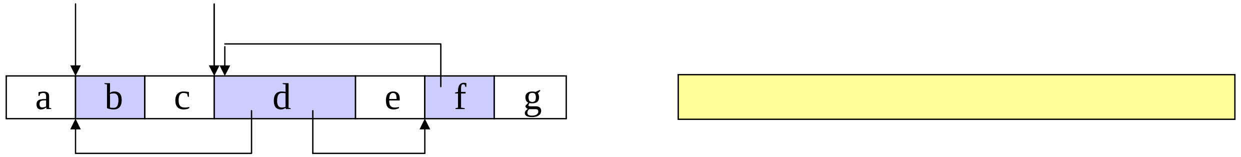

Simple sequential *alloc(size)*

top = top + size;

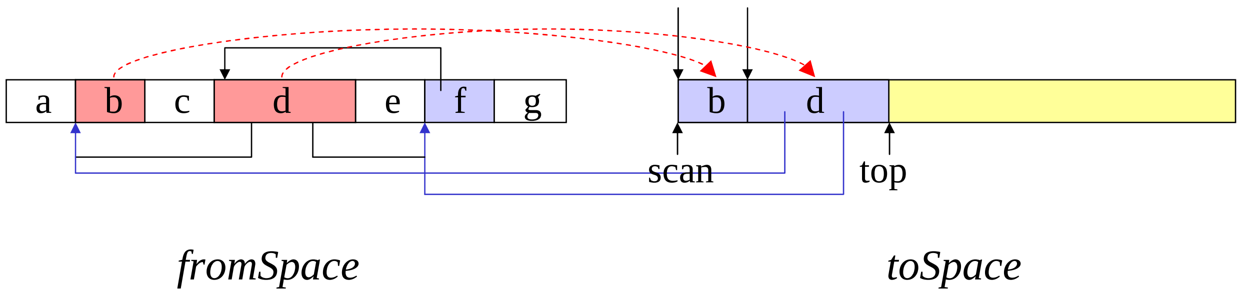
**If *fromSpace* is full all live objects are copied to *toSpace* (scavenging)**



# *Scavenging (phase 1)*



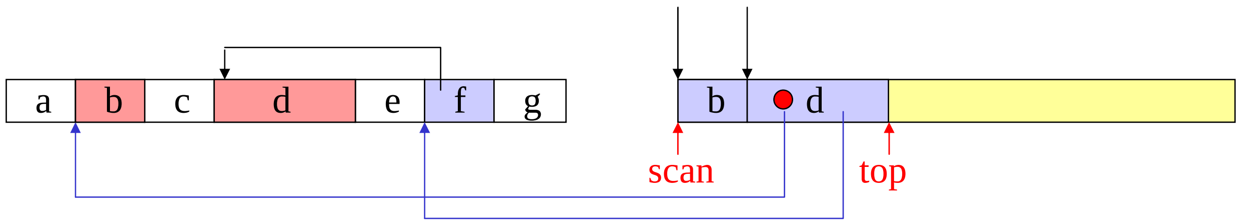

**1. Copy all objects that are directly referenced by roots**



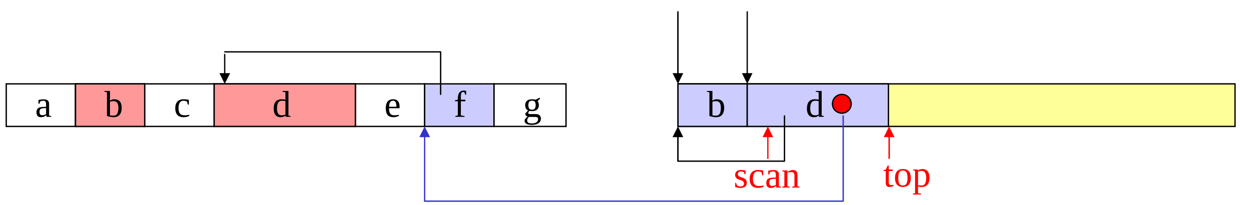

- Change root pointers to point to the copied objects
- Mark copied objects in *fromSpace* so that they are not copied again
  Install a forwarding pointer to the copy in *toSpace*
- Set a *scan* pointer to the start of *toSpace*
  Set a *top* pointer to the end of *toSpace*

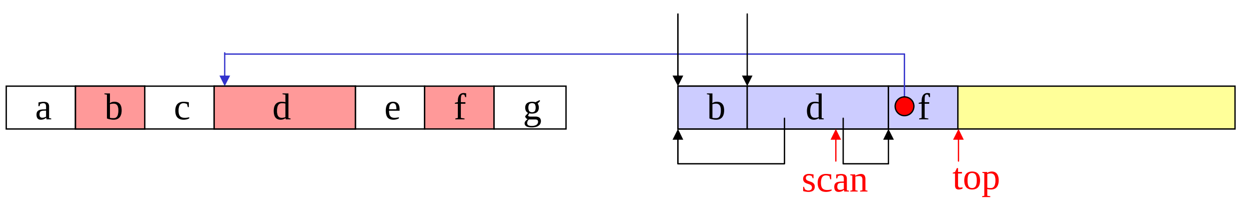
# *Scavenging (phase 2)*

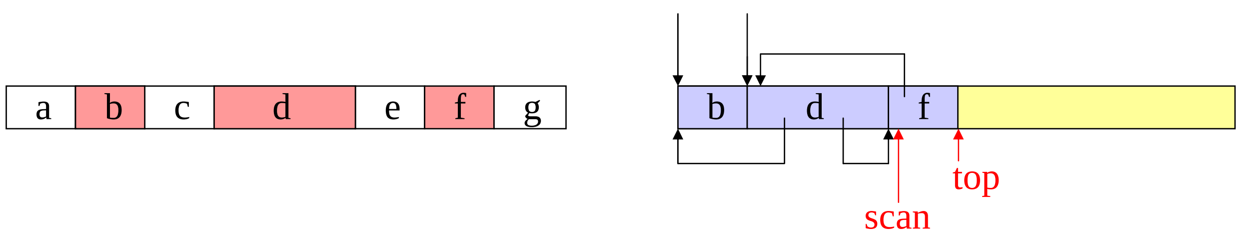
**2. Move the *scan* pointer through the objects in *toSpace***




- if *scan* hits a pointer:
  - copy the referenced object to *toSpace* (if not already copied)
    mark the copied object in *fromSpace* and install a forwarding pointer to the copy
  - change the pointer to point to the copy



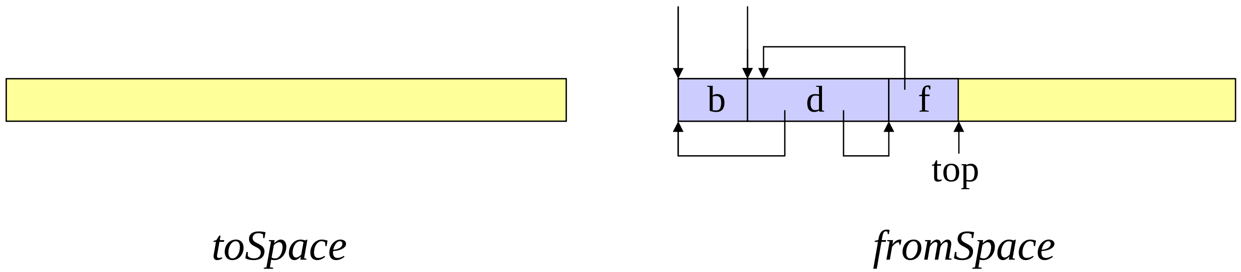




ready if
*scan* == *top*

# *Scavenging (phase 3)*

**3. Swap *fromSpace* and *toSpace***




New objects are allocated sequentially in *fromSpace* again

**Advantages**

- single-pass algorithm
  no *mark()*
  purely sequential; no graph traversal
- heap is compacted
  (no fragmentation, better locality)
- simple *alloc()*
- run time independent of heap size;
  depends only on the number of live objects

**Disadvantages**

- only half of the heap can be used
  for allocation
- copying costs time
- objects change their address
  => pointers have to be adjusted

# Comparison of the 3 basic techniques

## Run time performance

| | | |
|---|---|---|
| Reference Counting | time ≈ | number of pointer assignments + number of dead objects |
| Mark & Sweep | time ≈ | number of live objects + heap size |
| Stop & Copy | time ≈ | number of live objects |

## Overheads

| | RC | M&S | S&C |
|---|---|---|---|
| • for pointer assignments | *** | | |
| • for copying | | | *** |
| • for alloc() | * | * | |
| • for heap traversal | | *** | |

## GC and virtual memory

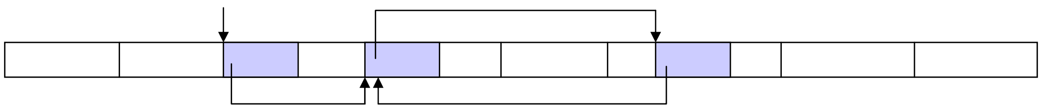
- Sweep swaps all pages in (others are swapped out)
- S&C can use big semi-spaces, because *toSpace* is originally on the disk anyway. While *toSpace* gets full its pages are swapped in and *fromSpace* gets swapped out

# 2. Garbage Collection

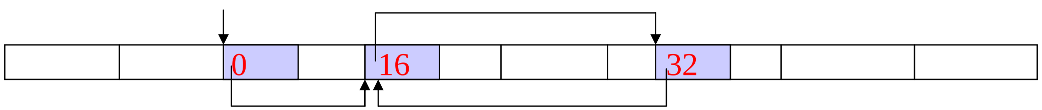
# *Mark & Compact*

Compacting variant of Mark & Sweep

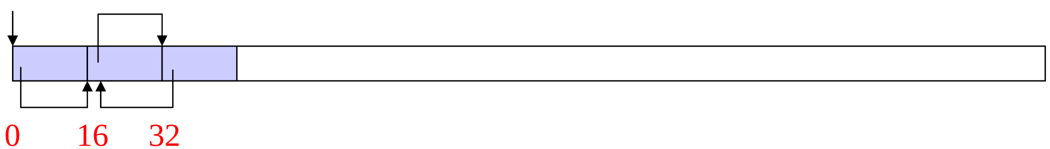
- Mark all reachable blocks



- Sweep 1: For every marked block compute its address after compaction




- Sweep 2: Adjust roots and pointer fields to point to the new addresses
- Sweep 3: Move blocks to the computed addresses




**Advantages**

+ removes fragmentation
+ simple sequential *alloc()*
+ order of objects on the heap is retained
+ good locality (virtual memory, caches)

**Disadvantages**

- objects must be copied (moved)
- needs an additional address field per block
- 3 sweeps necessary => slow

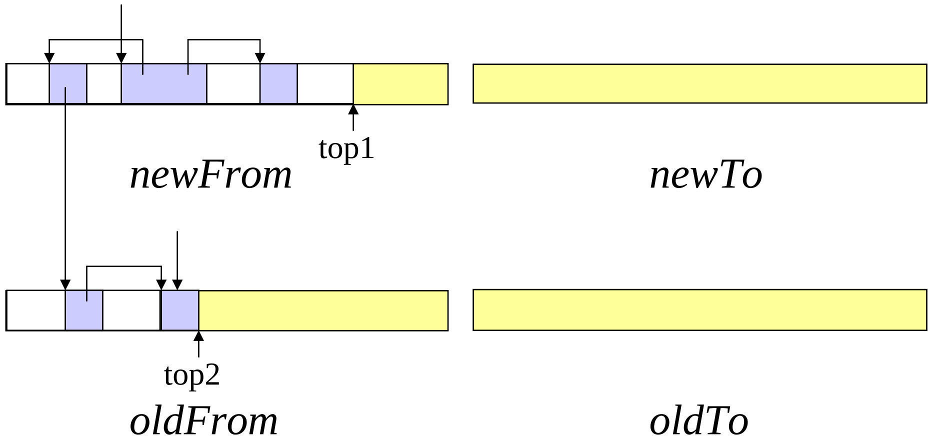
# 2. Garbage Collection

# *Generation scavenging -- idea*

Variant of Stop&Copy

**Problems** 1) Long-living objects must be copied in every GC run.
2) Most objects die young!

**Solution** Distinguish between short-living and long-living objects



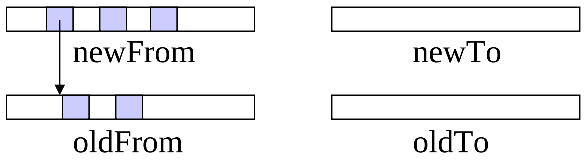

**Young generation**
(short-living objects)

**Old generation**
(long-living objects)

- New objects are always allocated in *newFrom*
- GC for *newFrom* is performed more frequently than for *oldFrom*;
  objects that have been copied *n* times get "tenured" (are copied to *oldFrom*)
- Good for languages with many short-living objects (e.g. Smalltalk)
  - only 5% of the objects survive the first collection
  - only 1% of the objects survive the second collection (but those live very long)
- *newFrom* often small (< 128 KB); managed with Stop&Copy
  *oldFrom* often large; managed with Mark&Compact or with an incremental GC

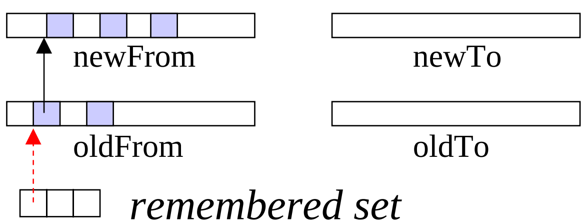
# *Cross-generation pointers*

## Pointers from *newFrom* to *oldFrom*




- Before every old-GC a new-GC is performed
- Pointer from new to old are detected and considered as roots for old-GC

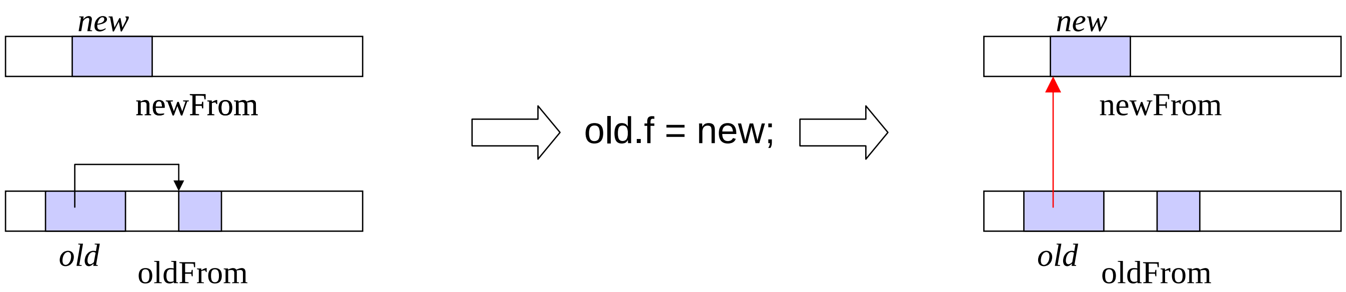
## Pointers from *oldFrom* to *newFrom*




- If an object is copied from *newFrom* to *oldFrom*, any pointers from old to new are stored in a so-called "remembered set"
- The remembered set is used as a root table for new-GC
- There is empirical evidence that there are more pointers from *newFrom* to *oldFrom* than vice versa

# *Write barriers*

## Problem: pointer assignments




Installs a pointer from *oldFrom* to *newFrom.*
How does the GC notice that?

## Compiler must generate "write barriers"

At every pointer assignment

```
old.f = new;
```

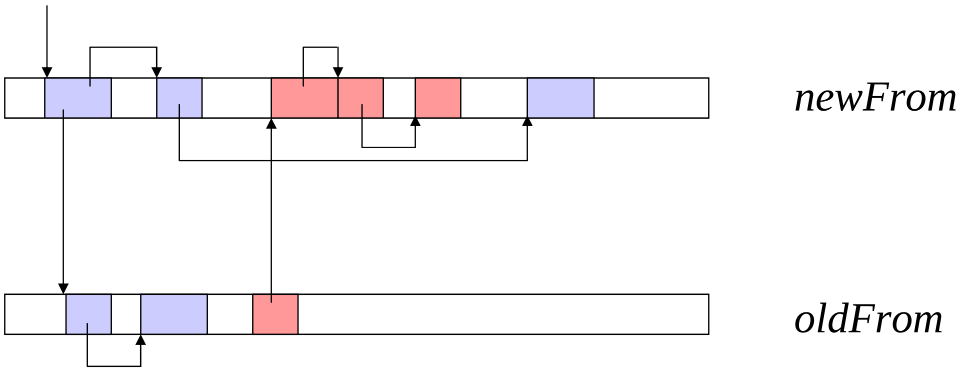
the following code must be generated:

```
if (old in oldFrom && new in newFrom)
    add Addr(old.f) to rememberedSet;
```

- Check can be implemented efficiently (see later)
- Such pointer assignments are rare
- Costs about 1% of run time in Java
- Optimizations possible, e.g.: assignments to *this.f* in a constructor cannot install a pointer from *oldFrom* to *newFrom*

# *Tenured garbage*

Dead objects in the old generation




Tenured garbage may keep dead objects in the young generation alive
=> should be avoided if possible

## Tenuring threshold

*n*: Number of copy runs until an object is copied to *oldFrom*

*Dilemma*

*n* small => much tenured garbage
*n* large => long-living objects remain in young generation longer than necessary
=> long GC times

# *Adaptive tenuring* *(Ungar 1988)*

Keep tenuring threshold flexible (depending on the amount of live objects in *newFrom*)

**Object age**

Every object remembers how often it was copied in the young generation (*age* field)

**Age table**

How many bytes of every age are in *newFrom*?

| age | bytes |
|---|---|
| 1 | 50000 |
| 2 | 9000 |
| 3 | 8000 |

i.e. 8000 bytes were already copied 3 times

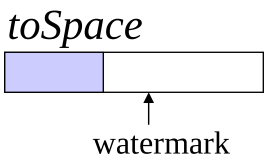
**Watermark**

Assume that the tolerated maximum GC pause is 1 ms
=> watermark = max. number of bytes, which can be copied in this time

# *Adaptive Tenuring (continued)*

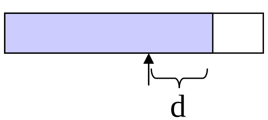
## Situation after copying

**survived objects < watermark**




=> *tenureAge* = ∞ i.e. during the next GC run no objects are copied to *oldSpace*

**survived objects > watermark**




=> *tenureAge* must be chosen such that at least *d* bytes are tenured in the next GC run

| | |
|---|---|
| 1 | 50000 |
| 2 | 9000 |
| 3 | 8000 |

} e.g. *d* = 10000 => all objects with *age* >= 2 are copied to *oldFrom* in the next GC run.

# 2. Garbage Collection

# *Motivation*

**Goal**

Avoid long GC pauses (pauses should be less than 1 ms)

**Application areas**

- For soft real-time systems (hard real-time systems should not use a GC at all)
- For the old generation in a generational GC
  GC pause is longer for the old generation because:
  - old generation is larger than young generation
  - there are more live objects in the old generation than in the young generation

**Idea**

GC (*collector*) and application program (*mutator*) run in parallel (interleaved)

a) collector runs continuously as a background thread
b) collector stops the mutator, but does only a partial collection

**Problem**

Mutator interferes with the collector!
Can change data structures, while the collector is visiting them

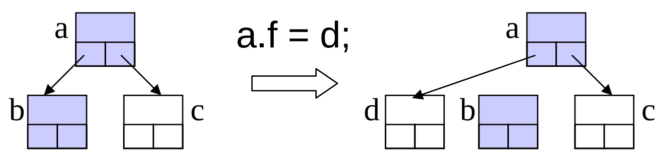
# *Suitability of basic techniques for incr. GC*

**Reference Counting** ⇒ yes

- Counter updates do not cause substantial pauses
- if counter == 0
  - object reference is written to a worklist
  - worklist is processed as a background thread
    (reclaiming objects and writing new references to the worklist)

**Mark & Sweep** ⇒ no

- Mutator may interfere with the mark phase
- Sweep phase can run in the background,
  if the mutator ignores the mark bits




b is kept alive
d is collected!

**Stop & Copy** ⇒ no

- Mutator may interfere

M&S and S&C must be modified in order to be able to run incrementally

# 2. Garbage Collection

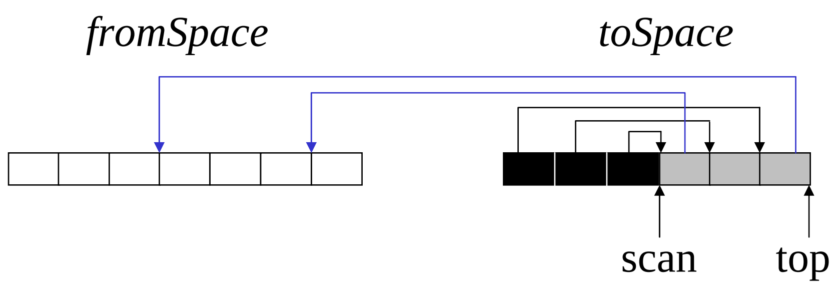
# *Tricolor marking*

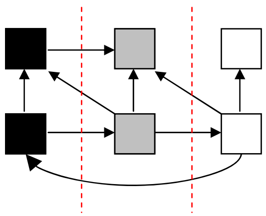
Abstraction of an incremental GC, from which various concrete algorithms can be derived

## Idea: all objects have one of the following colors

- **white** yet unseen objects (potentially garbage)
- **black** reachable and already processed objects
- **grey** objects that have been seen but not yet processed
  (pointers to them must still be followed)

## For example in Stop&Copy



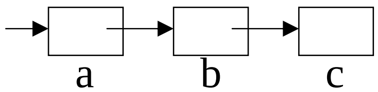



## Invariant

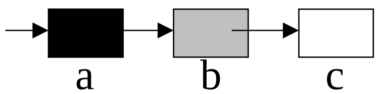
There must never be a pointer from a black object to a white object

# *What problem can arise?*

## Example

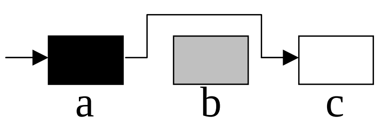



Collector starts running




Mutator interferes

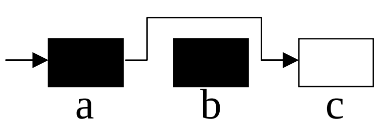
```
a.next = c;
b.next = null;
```




## Problem source

A pointer to a white object is installed into a black object

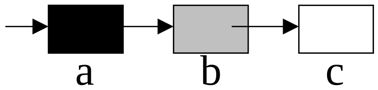
Collector continues




*c* is erroneously considered as garbage
*b* is erroneously kept alive

# *Problem solution*

We have to avoid that a black object points to a white object

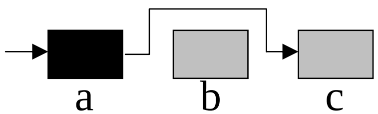



a.next = c;
b.next = null;

**This can be avoided in 2 ways**

## Read barrier

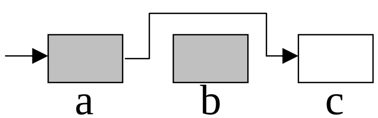
The mutator must not see white objects.
Whenever it reads a white object, this object is "greyed" (i.e. marked for processing)




a.next = **c**;

## Write barrier

If a white object is installed into a black object the black object is "greyed"
(i.e. it must be revisited)

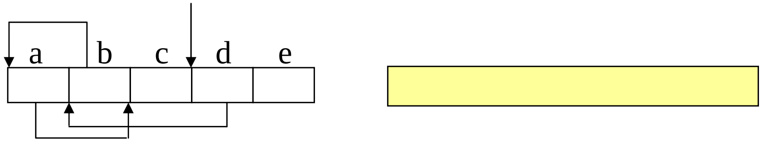



**a.next** = c;

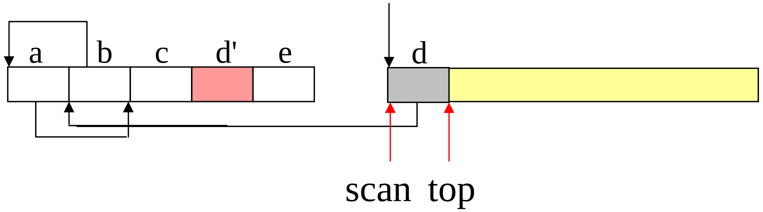
- Read barriers are more conservative, because the white object that is read need not be installed in a black object
- Read barriers are expensive if implemented in software.
- There are more pointer reads than pointer writes
- Read barriers usually for Stop&Copy
  Write barrier usually for Mark&Sweep

In both cases *b* is left over as "floating garbage"; but it is reclaimed in the next GC run.

# *Baker's algorithm (read barrier)*

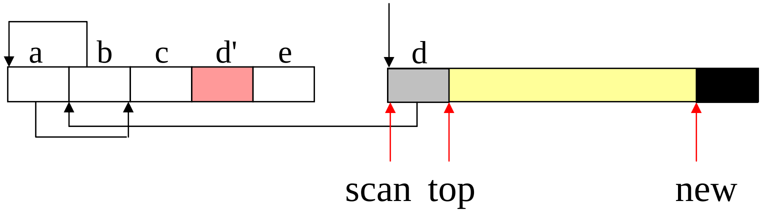



**1. Copy all objects that are directly referenced by roots**

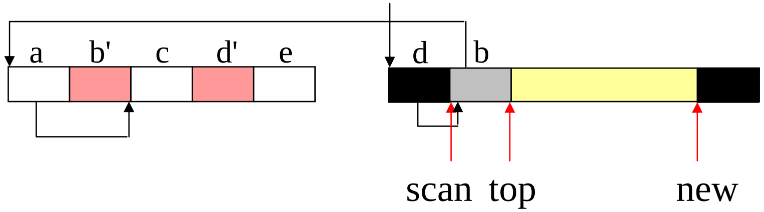



**2. New objects are allocated at the end of *toSpace***

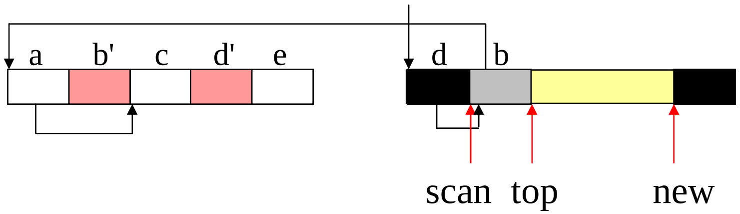
they are conceptually black (they do not contain pointers to white objects)




**3. At every *alloc()* do also an incremental scan/copy step**



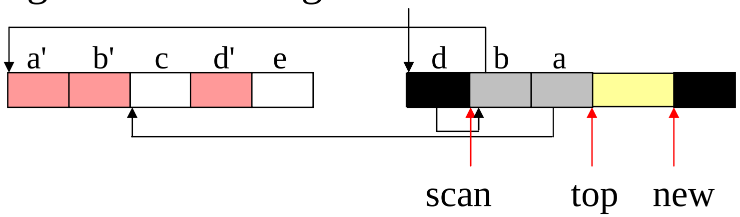
# Baker's algorithm (continued)




**4. If the mutator accesses a white object, this object is copied and becomes grey**

(read barrier)

e.g. after accessing *a*




```
Pointer get (Pointer p) {
    if (p points to fromSpace) {
        if (p not already copied)
            copy p to top;
            p.forward = top;
            top = top + p.size;
        }
        p = p.forward;
    }
    return p;
}
```

Read barrier: a = get(a);
Mutator sees only *toSpace* objects

Can also be implemented with virtual memory:
*fromSpace* is protected so that every read access causes a trap

## Problems

- Requires a read barrier for <u>every</u> read access to a white object (20% of the run time)
- Can only be implemented efficiently with special hardware or OS support (virtual memory)

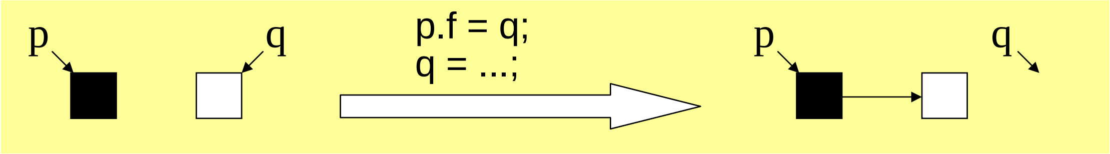
# *Write barrier algorithms*

Catch pointer <u>writes</u>, not pointer <u>reads</u>

+ Writes are less frequent than reads (5% vs. 15% of all instructions) => more efficient

- Works only for Mark&Sweep, not for Stop&Copy:
  Stop&Copy requires read barriers, because if an object that has already been copied is accessed again in *fromSpace* the access must be redirected to the copy in *toSpace*

## Problematic case




Two conditions must hold in order to cause a problem:

a) a white object is installed in a black object (p.f = q;)
b) all other pointers to the white object disappear (q = ...;)

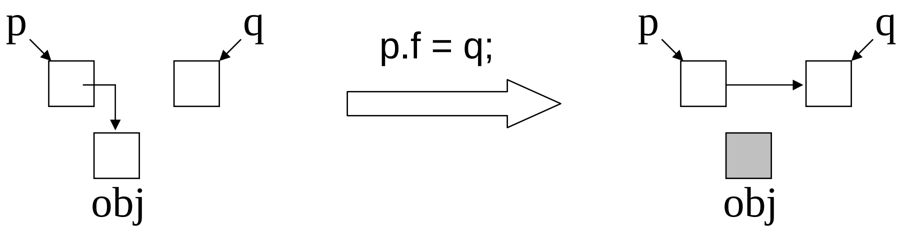
At least one of these conditions must be prevented

## 2 kinds of write barrier algorithms

- **Snapshot at beginning** (prevents condition b)
- **Incremental update** (prevents condition a)

# *Snapshot at beginning*

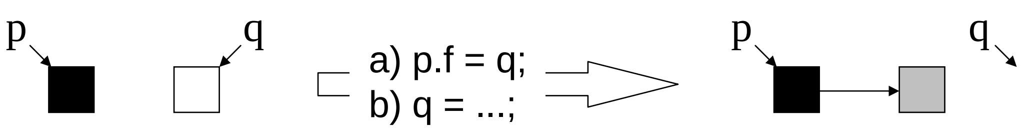
**Objects stay alive, if they were reachable <u>at the beginning</u> of the GC run**




*obj* was reachable and thus must stay alive

If a pointer is removed from an object the object is greyed

- Prevents that the last pointer to an object disappears (condition b)




- Implementation:

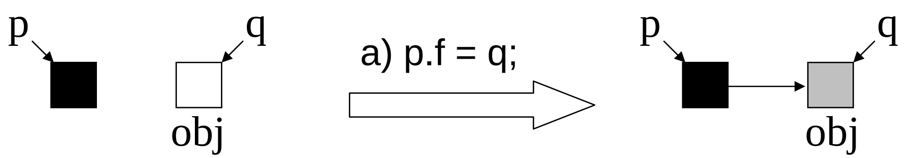
```
ptr = ...;   ⇒   worklist.add(ptr);
                 ptr = ...;
```

← Write barrier generated by the compiler; *worklist* is processed by the GC later

- Catches <u>all</u> pointer assignments (not only assignments to pointers in black objects)
- Very conservative!

# *Incremental update*

**Objects stay alive, if they are reachable <u>at the end</u> of the GC run**




if *q* disappears,
*obj* is visited nevertheless

if a pointer to a white object is
installed in a black object
the white object is greyed

- Prevents that white objects are installed in black ones (condition a)
- Implementation:

```
p.f = q;     ⇨     if (black(p) && white(q))
                       worklist.add(q);
                   p.f = q;
```

Write barrier generated by the compiler;
*worklist* is processed by the GC later

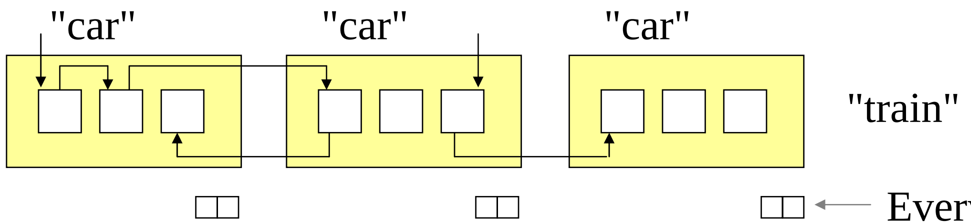
- Catches only assignments to pointers in black objects
  (more accurate than "snapshot at beginning")

# 2. Garbage Collection

# Idea

Incremental Stop&Copy algorithm with write barriers (primarily for old generation)

- **The heap is divided into segments ("cars") of fixed size** (e.g. 64 KBytes)




Every car has a *remembered set*: addresses of pointers from later cars to this car (= additional roots)

- **Every GC step collects exactly 1 car (the first one)**
  - copies live objects to other cars
  - releases the first car

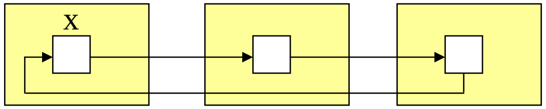
  GC of a single car is fast => no noticeable overhead

- **Objects that are larger than 1 car are managed in a separate heap** (large object area)

# Dealing with dead cycles

## Problem

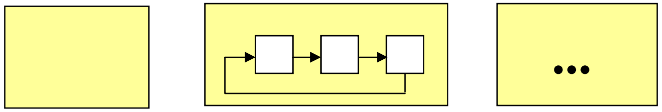
Dead cycle across several cars




If the first car is collected we don't see that *x* is dead, because it is referenced from a later car.

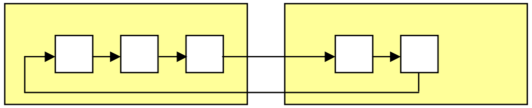
## Simple solution

All objects of a dead cycle must be copied into the same car




If this car is collected the whole cycle is released together with the car

## Does not always work ...

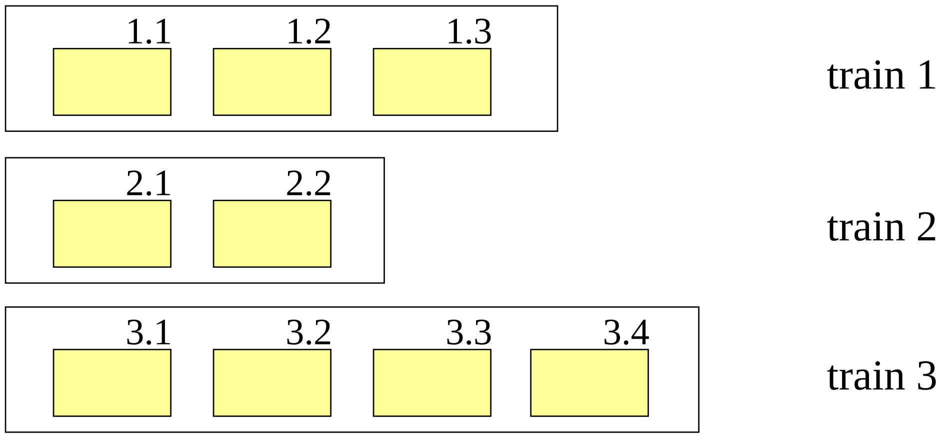


... because the objects of a cycle may not fit into a single car

=> This is where the train algorithm comes in

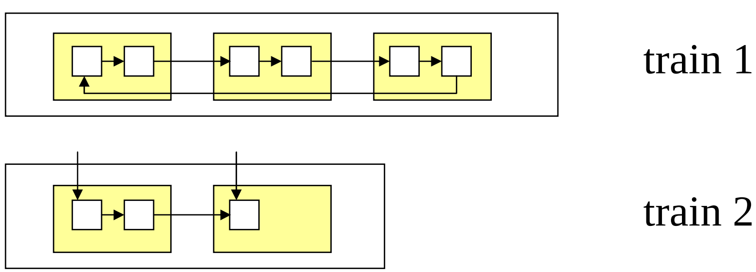
# *Train algorithm*

## Cars are grouped into several trains




Order of processing: 1.1 < 1.2 < 1.3 < 2.1 < ... < 3.3 < 3.4

## Our goal is to accumulate dead cycles in the first train



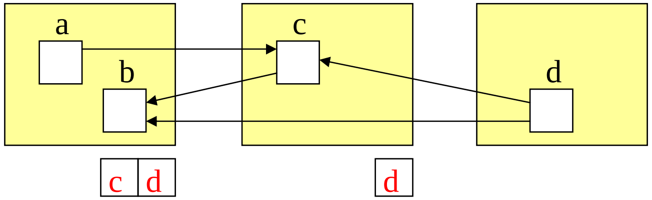

Objects that are referenced by roots or from other trains are evacuated to later trains

If no object in the first train is referenced from outside the first train
=> release the whole first train!

# *Remembered sets*

**Remembered set of a car *x***

Contains addresses of pointers from later cars to *x*

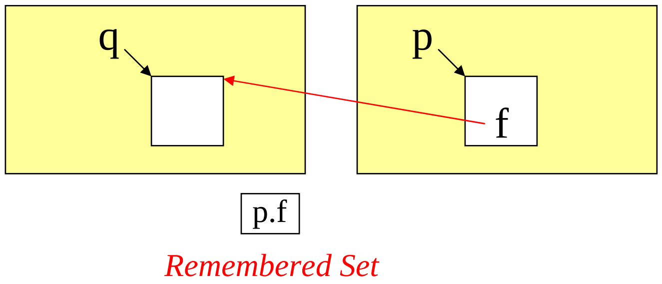



Additionally, there is a list of roots pointing from outside the heap into the cars

# *Updating remembered sets*

**Write barriers**

```
p.f = q;
```




If car(*q*) is before car(*p*) (i.e. if this is a pointer from back to front)
=> add address of *p.f* to remembered set of car(*q*)

# *Car ordering*

Cars are *logically* ordered!
Their *physical* addresses need not be in ascending order (cars may be anywhere in the heap)

| | | | | |
|---|---|---|---|---|
| *train 1* | 1.1 | 1.2 | 1.3 | |
| | 40000 | 10000 | 20000 | |
| *train 2* | 2.1 | 2.2 | | |
| | 50000 | 00000 | | |
| *train 3* | 3.1 | 3.2 | 3.3 | 3.4 |
| | 70000 | 90000 | 30000 | 80000 |

← physical addresses (hexadecimal)

**Car table**

maps physical address to car number

- e.g. car size $2^k$ (e.g. $2^{16}$ bytes = 64 KBytes)
- car index *n* = (*adr* - heapStart) >> *k*;
- *tab*[*n*] tells us which car is stored at *adr*

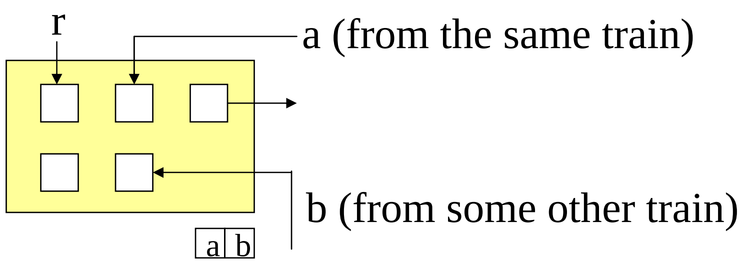
| n | tab |
|---|---|
| 0 0000 | 2.2 |
| 1 0000 | 1.2 |
| 2 0000 | 1.3 |
| 3 0000 | 3.3 |
| 4 0000 | 1.1 |
| 5 0000 | 2.1 |
| 6 0000 | - |
| 7 0000 | 3.1 |
| 8 0000 | 3.4 |
| 9 0000 | 3.2 |
| 10 0000 | ... |

**Example**

pointer from 30AF4 to 50082

- from *tab*[3] to *tab*[5]
- from 3.3 to 2.1
- from back to front

# *Incremental GCstep*

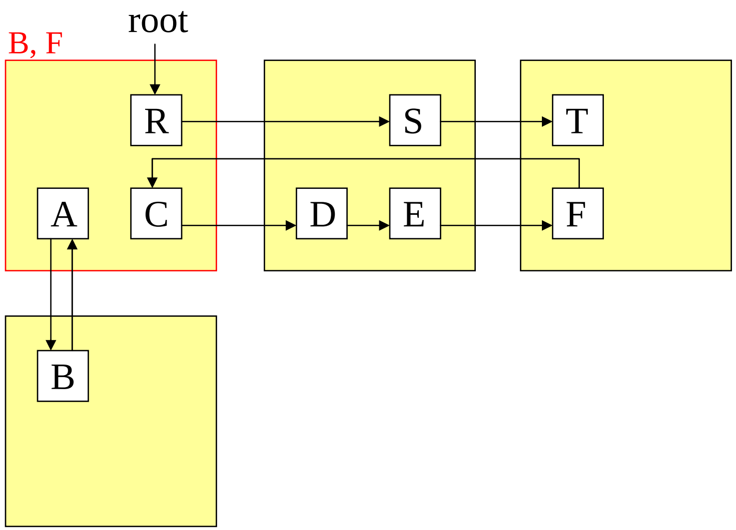



```
if (there are no pointers to the first train from outside this train)
   release the whole first train;
else {
   car = first car of first train;
   ---------------------------------------------- copy ----
   for (all p in rememberedSet(car)) {
      copy pobj to the last car of train(p);
      if full, start a new car in train(p);
   }
   for (all roots p that point to car) {
      copy pobj to the last car of the last train (not to the first train!);
      if full, start a new train;
   }
   ---------------------------------------------- scan ----
   for (all p in copied objects)
      if (p points to car) {
         copy pobj to last car of train(p);
         if full, start a new car in train(p);
      } else if (p points to a car m in front of car(p))
         add p to rememberedSet(m);
   release car;
}
```

**Additional considerations**

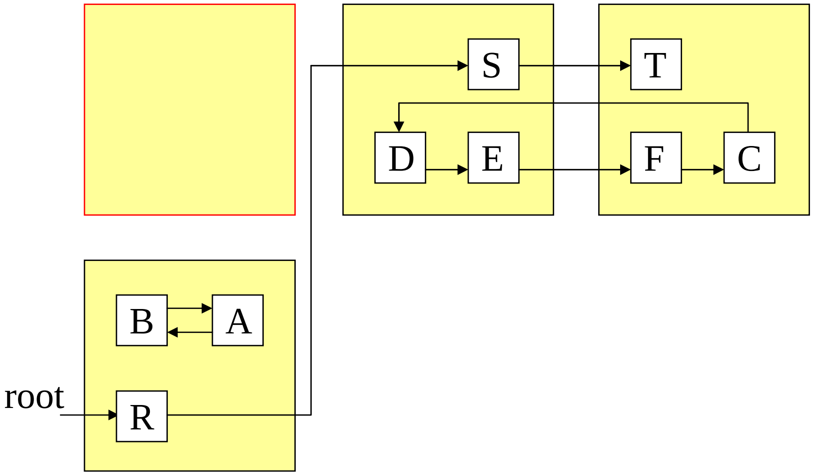
- How to find pointers from outside this train: inspect roots and remembered sets of all cars of this train.
- If there are multiple pointers to the same object => don't copy this object twice, but install a forwarding pointer.
- Cars and trains must be linked in order to find the first and the last car of a train.

# *Example*

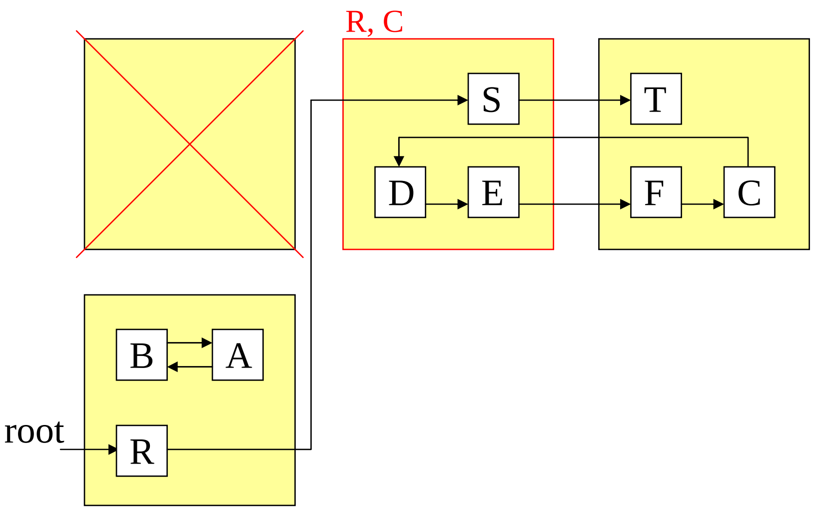



Assumption: our cars have only space for 3 objects

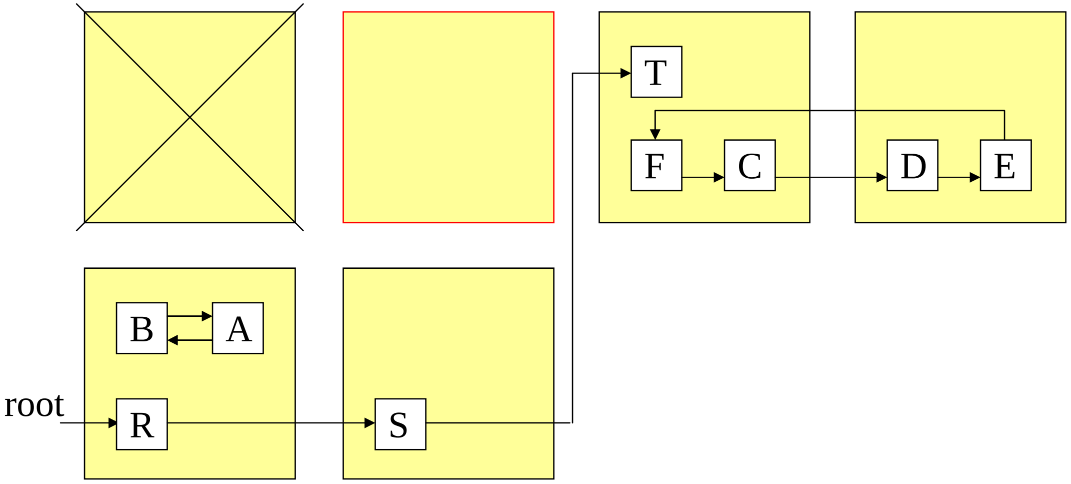
- copy *R* to the last car of the last train (because it is referenced from a root)
- copy *A* to the last car of train(*B*)
- copy *C* to the last car of train(*F*)



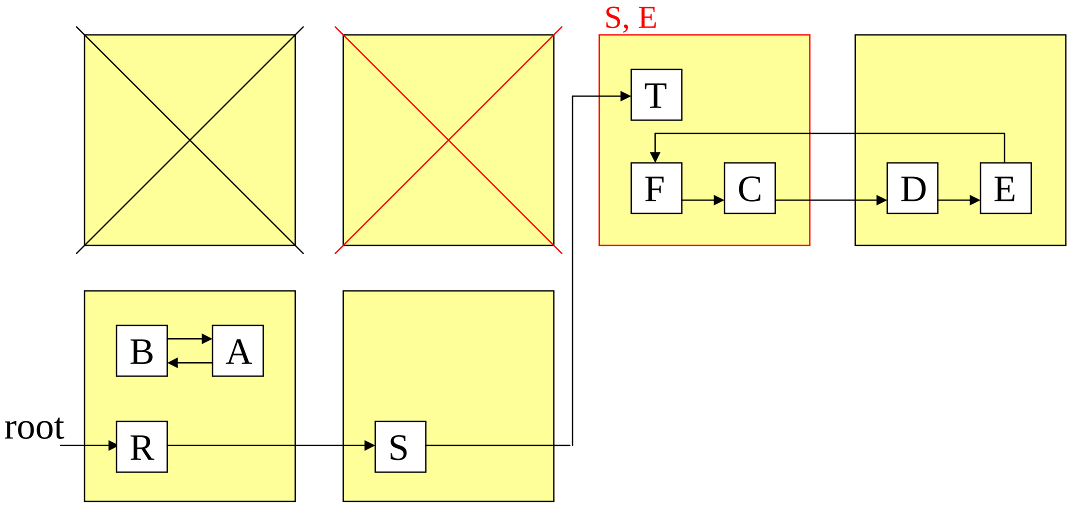
# *Example (continued)*



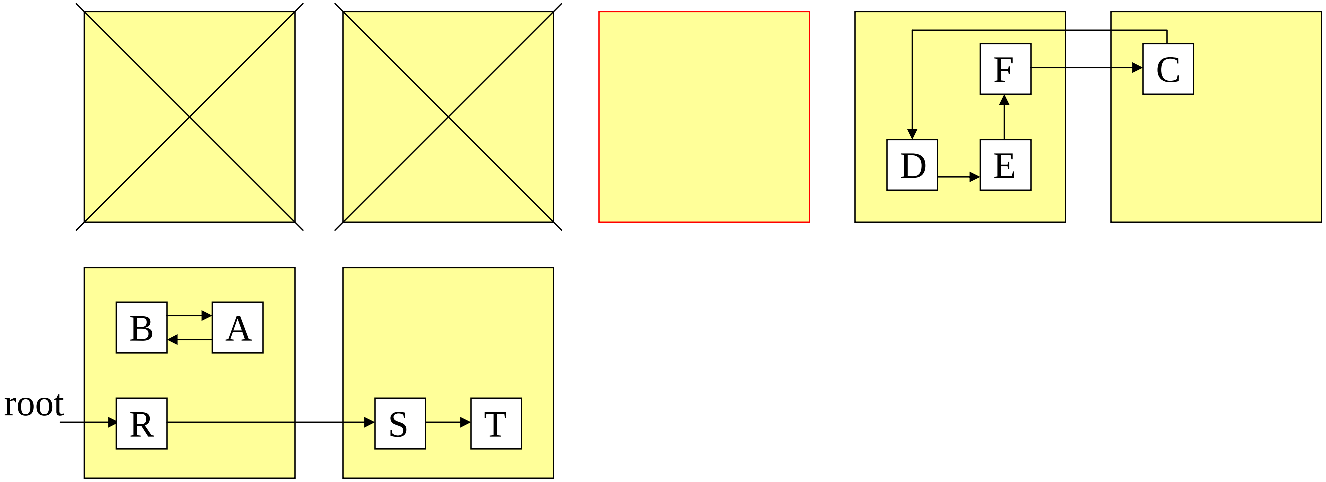

- copy *S* to the last car of train(*R*); no space => start a new car in train(*R*)
- copy *D* to the last car of train(*C*); no space => start a new car in train(*C*)
- copy *E* to the last car of train(*D*)



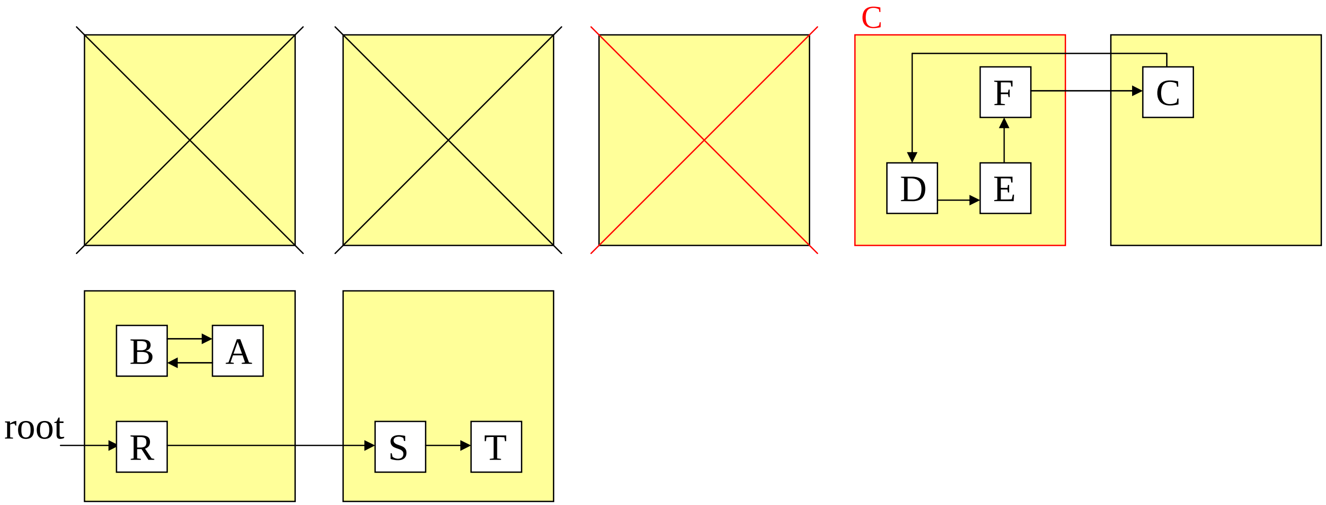
# *Example (continued)*




- copy *T* to the last car of train(*S*)
- copy *F* to the last car of train(*E*)
- copy *C* to the last car of train(*F*); no space => start a new car

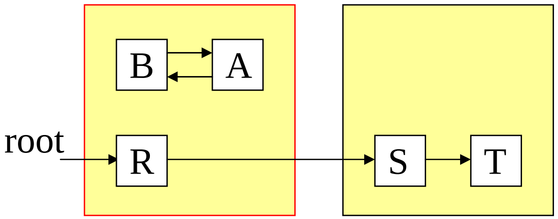


# *Example (continued)*

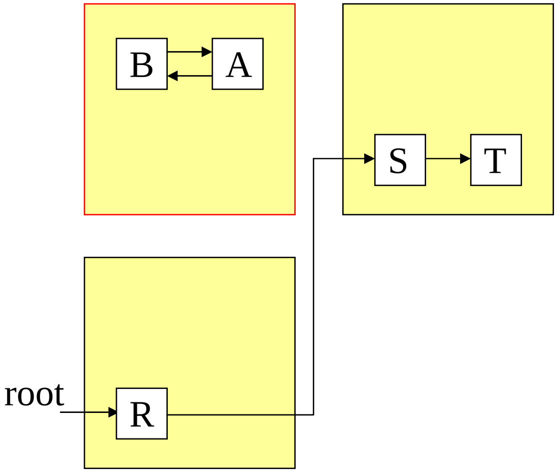



- no pointers to the first train from ouside this train => release the whole first train

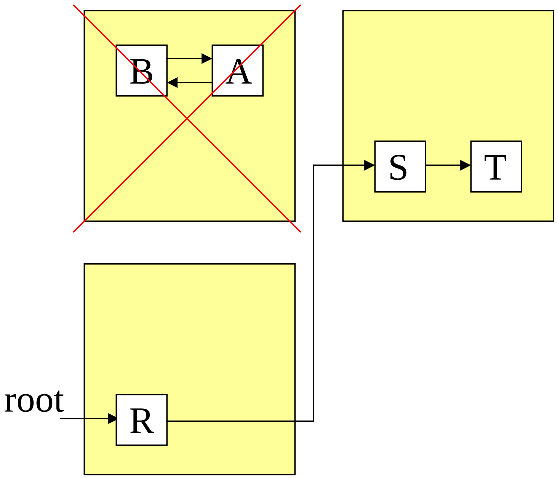
# *Example (continued)*




- copy *R* to the last car of the last train;
  Since there is only one train, start a new train

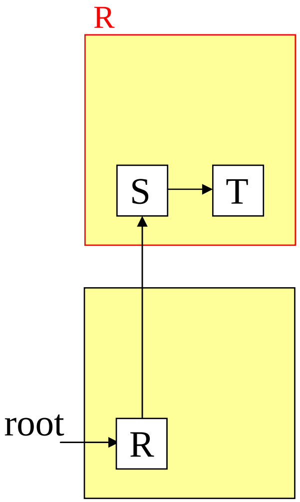


# *Example (continued)*

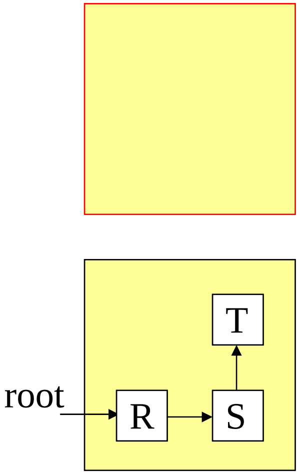



- no references into the first car => release first car

# *Example (continued)*




- copy *S* (and also *T*) to the last car of train(*R*)



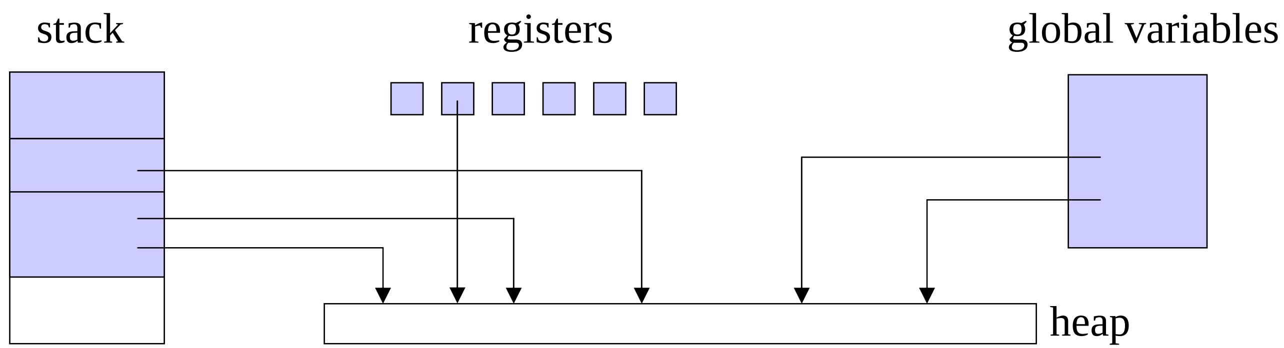

**ready!**
Only live objects survived
In every step at least 1 car was released => progress is guaranteed

# 2. Garbage Collection

# *Root pointers*

**Roots are all live pointers outside the heap**

- local variables on the stack
- reference parameters on the stack (can point to the middle of an object!)
- global variables in C, C++, Pascal, ...
  (static variables in Java are on the heap (in class objects))
- registers




**All objects that are (directly or indirectly) referenced from roots are live**

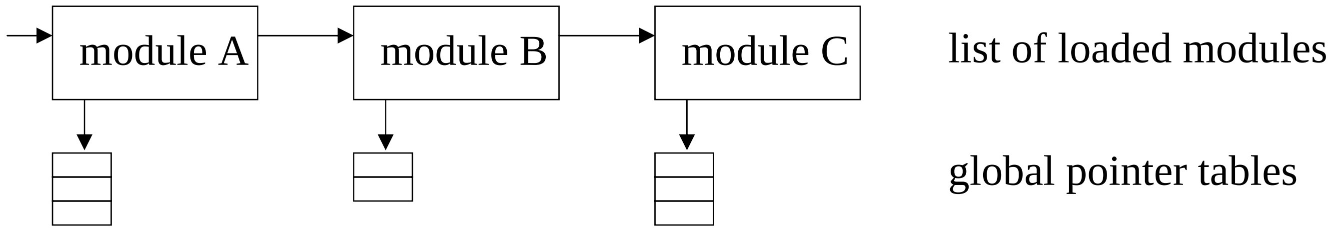
- *Mark & Sweep*:
  for (all roots p) mark(p);
  sweep();
- *Stop & Copy*:
  for (all roots p) copy referenced object to toSpace;
  scan();

# 2. Garbage Collection

# *Global pointers*

## Global pointer variables in Oberon

- For every module the compiler writes a list of global pointer addresses to the object file
- The loader creates a pointer offset table for every loaded module




```
for (all loaded modules m)
   for (all pointers p in m.ptrTab)
      if (p != null && *p not marked) mark(p);
```

## Static pointer variables in Java

- Fields of class objects (offsets stored in type descriptors)
- Loader creates class objects and stores their addresses in the roots table

# *Local pointers*

## For every method the compiler generates a table with pointer offsets

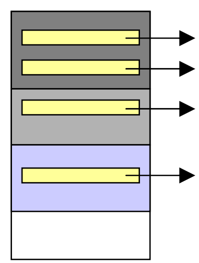
```
void foo(Obj a, int b) {    // a ... Offset 0
   int c;
   Obj d;                   // d ... Offset 3
   ...
}
```

| | fromPC | toPC | pointer offsets | registers with pointers |
|---|---|---|---|---|
| foo() | 1000 | 1250 | 0, 3 | |
| bar() | | | | |
| baz() | | | | |

- Compiler writes these tables to the object file
- Loader loads them into the VM

## Stack traversal in order to find local pointers

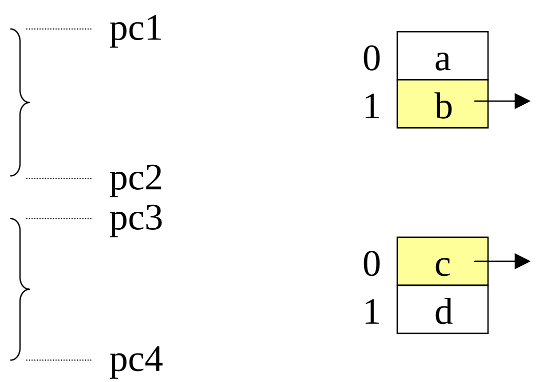
```
for (all stack frames f) {
   meth = method containing pc of f;
   for (all p in meth.ptrTab) mark(p);
   for (all r in meth.regTab) mark(r);
}
```



# *Blocks with different pointer offsets*

**Blocks of the same method can have different pointer offsets (in Java)**

```
if (...) {
    int a;
    Obj b;
    ...
} else {
    Obj c;
    int d;
}
```




Pointer offset table must have several regions per method

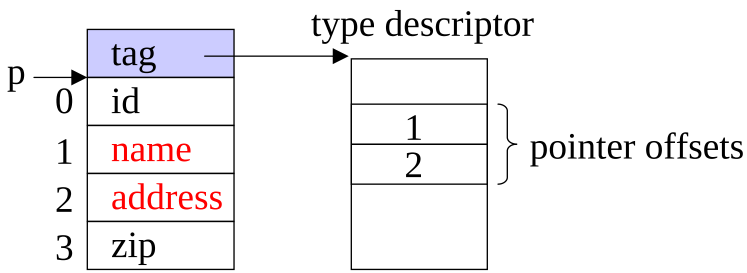
| from | to | pointer offsets |
|---|---|---|
| pc1 | pc2 | 1 |
| pc3 | pc4 | 0 |

In the Hotspot VM this is solved via safepoints (see later)
Also allows that a register may contain a pointer or a non-pointer at different locations

# *Pointers in objects*

```
class Person {
    int id;
    String name;
    String address;
    int zip;
}
```

Type descriptors contain pointer offsets




- Compiler writes type descriptor to the object file
- Loader loads type descriptor when the corresponding class is loaded
- *new Person()* installs the type descriptor in the *Person* object

# 2. Garbage Collection

# *Conservative garbage collection*

Used if the compiler does not generate pointer tables (e.g. in C/C++)

## "Guess" which memory locations contain pointers

- Check every word *w* (on the stack, in the global data area, in heap objects, in registers)
- *w* is a possible pointer if
  - *w* points to the heap (easy to check)
  - *w* points to the beginning of an object (difficult to check)

## Guessing must be done conservatively

If the guess is wrong (*w* is actually an int and not a pointer), no harm must occur

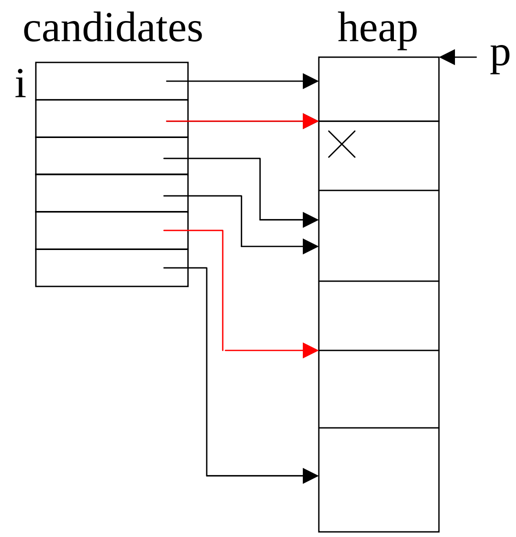
What if the guess was wrong?

- Mark&Sweep: an object is marked although it may be garbage
  => no harm
- Stop&Copy: the "wrong pointer" (which is actually an int) is changed to the new object location
  => destroys data
- Ref. Counting: only for searching pointers in deallocated objects;
  Counter is decremented, although the object was not referenced by *w*
  => counters become inconsistent

# *Implementation with a candidate list*

All possible pointers are collected in a list

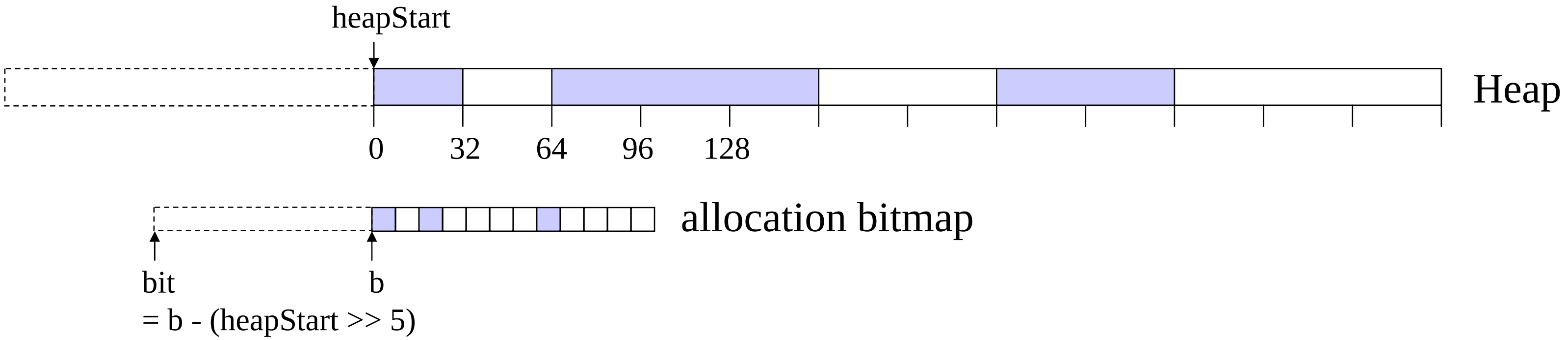
```
for (all words w in stack, global data and registers) {
    if (heapStart <= w < heapEnd) candidates.add(w);
}
candidates.sort();
i = 0; p = heapStart;
while (i < candidates.size() && p < heapEnd) {
    if (candidates[i] == p) {
        if (!p.marked) mark(p);
        i++; p = p + p.size;
    } else if (candidates[i] < p) {
        i++;
    } else {  // candidates[i] > p
        p = p + p.size;
    }
}
```




- Requires a full heap traversal to find the pointers
- In principle, *mark()* must inspect all words of an object in a similar way
- Sometimes a mixture: pointer tables for objects, conservative GC for stack etc. (e.g. in Oberon)

# *Implementation with an allocation bitmap*

- Blocks are allocated in multiples of 32 bytes (32, 64, 96, ...)
- There is a bitmap with 1 bit per 32 byte of heap area
- An address *a* is the beginning of a block ⇔ *bit*[*a* >> 5] == 1




- Bitmap requires 1 bit per 32 bytes (256 bits) => 1/256 = 0.4% of the heap
- *alloc()* must set the bits
- *sweep()* must reset the bits

```
for (all words w in stack, global data and registers) {
    if (heapStart <= w < heapEnd && bit[w >> 5] && !w.marked) mark(w);
}
```

\+ does not need a candidate list
\+ does not need an additional heap traversal
\- overhead for maintaining the bitmap
\- bit operations are expensive

# 2. Garbage Collection

# *GC in multi-threaded systems*

**Problems**

- Several mutator threads -- all must be stopped before GC runs
- GC is not allowed at all locations, e.g.

```
MOV  EAX, objAdr
ADD  EAX, fieldOffset
MOV  [EAX], value
```

← GC is not allowed here, because the object may be moved

**Safepoints (GC points)**

- Locations where GC is allowed
- Threads can only be stopped at safepoints
- For every safepoint there is
  - a table of pointer offsets in the current stack frame
  - a list of registers containing pointers

**Stopping threads at safepoints can be implemented in 2 ways**

- Safepoint polling
- Safepoint patching

# *Safepoint polling*

Mutator checks at safepoints if GC is pending

## Safepoints

- method entry (enter)
- method exit (return)
- system calls
- backward jumps

} guarantees that every thread reaches the next safepoint quickly

## Compiler emits the following code at every safepoint

```
if (gcPending) suspend();
```

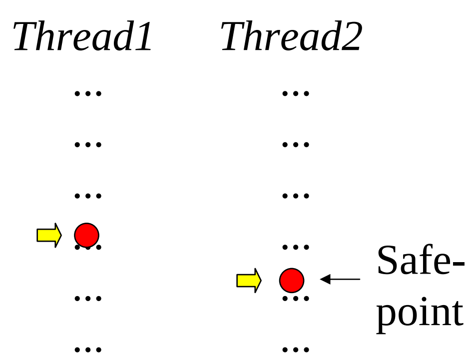
or:

```
MOV dummyAdr, 0
```

If *gcPending*, the memory page *dummyAdr* is made readonly
=> Trap => *suspend()*

## If the system runs out of memory ...

```
gcPending = true;
suspend all threads;
for (each thread t)
    if (not at a safepoint) t.resume();  // let it run to the next safepoint
// all threads are at safepoints
collect();
gcPending = false;
resume all threads;
```



# *Safepoint patching*

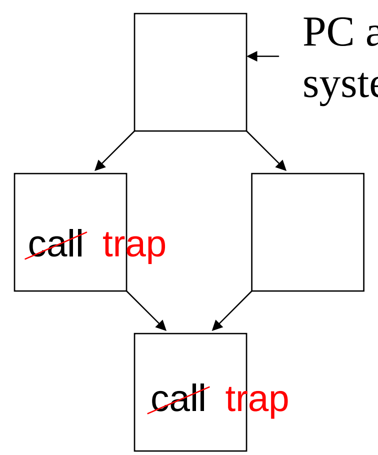
Safepoints are patched with instructions that cause a thread to be suspended

## Safepoints

- method call ← because of dynamic binding the invoked method is unknown => would require too many methods to be patched
- method exit (return)
- backward jump

## If the system runs out of memory ...

```
suspend all threads;
for (each thread t) {
    patch next safepoint with a trap instruction;
    t.resume();  // let it run until the next safepoint
}
// all threads are at safepoints
collect();
restore all patched instructions;
resume all threads;
```




PC at the time when the system runs out of memory

# 2. Garbage Collection

# *Goal*

**Cleanup work to be done before an object is reclaimed**

- closing open files
- terminating threads

e.g. in Java

```
class T {
    ...
    protected void finalize() throws Throwable {
        ... // cleanup work
        super.finalize();
    }
}
```

- is automatically called before a *T* object is reclaimed
- avoid finalizers if possible
  reclamation of finalized object is delayed (see later)

# *Finalization during GC* *(e.g. in Oberon)*

- Objects with a *finalize()* method are entered into a finalization list when they are allocated
  => *new()* takes a little longer
- Finalization list entries must not be considered as pointers,
  otherwise these objects would never be garbage collected

## Finalization during Mark&Sweep

- *mark();*
- for all unmarked objects in the finalization list:
  - *obj.finalize();*
  - remove *obj* from finalization list
- *sweep();*

## Finalization during Stop&Copy

- *scan();*
- for all objects in the finalization list, which have not been copied
  - *obj.finalize()*;
  - remove *obj* from finalization list

=> increases the GC pause!

# *Finalization in the background*
## *(e.g. in Java and .NET)*

- *mark();*
- for all unmarked objects *obj* in the finalization list
  - *worklist.add(obj);*
  - set mark bit of *obj*
- *sweep();*

- background thread processes *worklist*
  - calls *obj.finalize();* ← Finalization can happen at any time after GC! No guarantee that finalization will happen at all until the end of the program
  - clears mark bit
  - remove *obj* from finalization list
- object is collected during the next GC run

=> no substantial increase of GC pauses
but objects that have to be finalized are released with a delay

**Object resurrection**

```
protected void finalize() throws Throwable {
    ...
    globalVar = this;
}
```

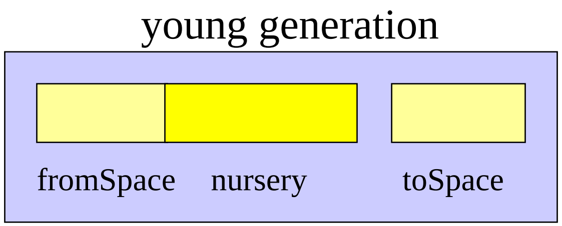
globalVar = this; ← brings finalized object to life again!

- next GC will detect this object as live => keep it
- if object finally dies => no finalizer call again!

# 2. Garbage Collection

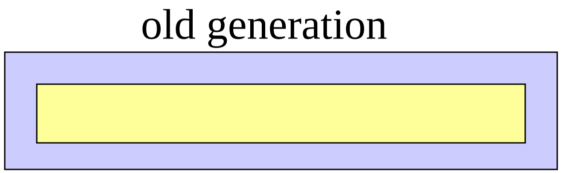
# *Heap layout*

## Two generations




**Stop&Copy**

- New objects are allocated in the *nursery*
- if full => copy *fromSpace* + *nursery* to *toSpace*
  advantage: less waste for *toSpace*
- if overflow => copy remaining objects to *old*
- after *n* copy passes an object is copied to *old*
- *n* is variable (adaptive tenuring)
  many new objects => faster "tenuring"




**Mark&Compact**

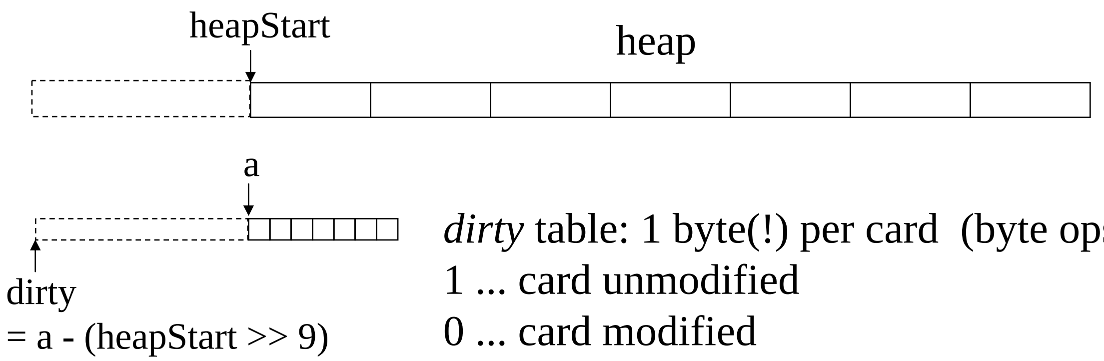
- is executed less frequently

## Remembered set

- List of all pointers from *old* to *young*
- An entry is made
  - if an object with pointers to other *young* objects is tenured
  - if a pointer to *young* is installed in an *old* object (detected by a write barrier)

# Write barriers

## Card-marking scheme




divided into
"cards" of 512 bytes

*dirty* table: 1 byte(!) per card  (byte ops are faster than bit ops)
1 ... card unmodified
0 ... card modified

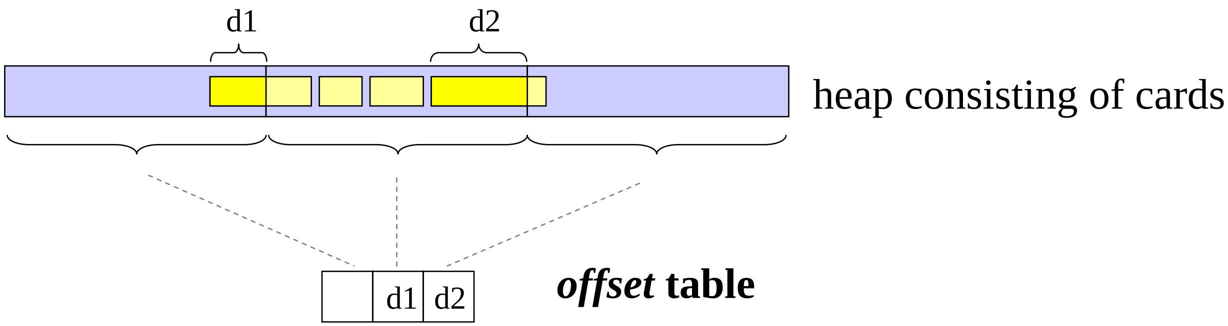
If a pointer is installed into an object: obj.f = ...; (no matter where it points to):

```
LEA   EAX, obj
MOV   EBX, EAX
SHR   EBX, 9
MOV   byte ptr dirty[EBX], 0
ADD   EAX, offset(f)
MOV   [EAX], ...
```

- 3 instructions per pointer write
- generated by the compiler (only for field assignments, not for assignments to pointer variables)
- write barriers cost about 1% of the run time
- GC searches all dirty cards in *oldSpace* for objects;
  any pointers in them that point to *newSpace* are entered into *rememberedSet*

# *Searching for objects in cards*

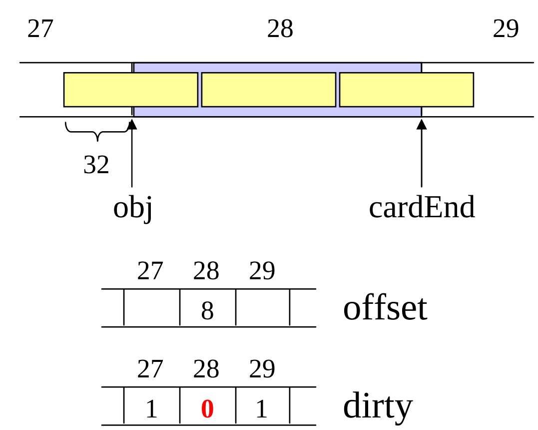
Objects may overlap card boundaries




***offset* table**

- 1 byte per card
- how far does the first object extend into the predecessor card?
- if it overlaps the whole predecessor card:
  *offset* = 255 => search one card before
- Objects are aligned to 4 byte boundaries
  => table holds *offset* / 4

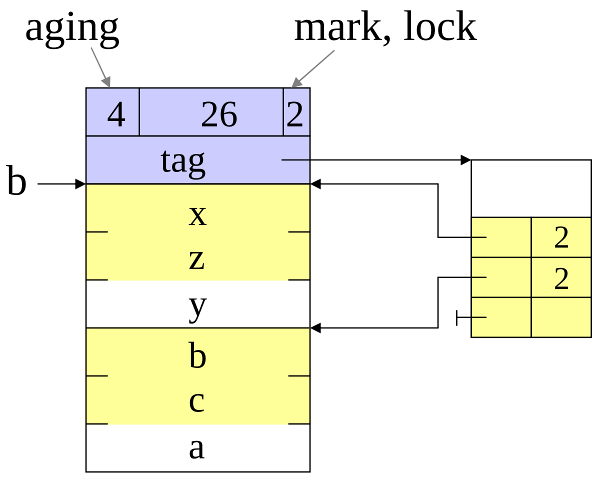
```
for (all dirty cards i in oldSpace) {
    obj = heapStart + 512 * i; cardEnd = obj + 512; j = i;
    while (offset[j] == 255) { obj = obj - 512; j--; }
    obj = obj - offset[j] * 4;
    while (obj < cardEnd) {
        for (all pointers p in obj.ptrTab)
            if (p points to newSpace) rememberedSet.add(p);
        obj += obj.size;
    }
}
```



# *Object layout*

```
class A {
    Obj x;
    int y;
    Obj z;
}
```

```
class B extends A {
    int a;
    Obj b;
    Obj c;
}
```




- All pointers of a class are stored in a contiguous area
- 2 words overhead per object
- First word is used for the new target address in Mark&Compact

# *Pointers on the stack*

Stack can hold frames of compiled or interpreted methods

**For interpreted methods**

Analyze the bytecodes to find out where the pointers are

**For compiled methods**

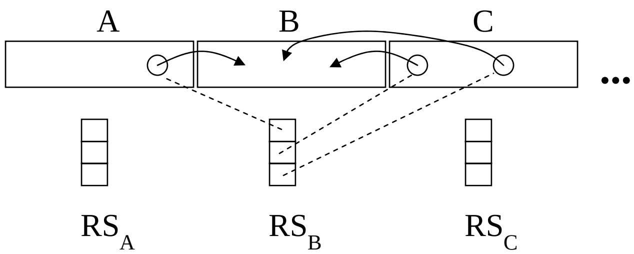
- Compiler generates a pointer table for every safepoint
  (call, return, backward branch and every instruction that can throw an exception)
- GC can only happen at safepoints
- Safepoint polling: at every safepoint there is the instruction:
  ```
  MOV dummyAdr, 0
  ```
  If GC is pending, the memory page *dummyAdr* is made readonly => trap => suspend()

# G1 -- Garbage-first collector

Alternative GC for server applications (large heaps, 4+ processors)
Since Java 6

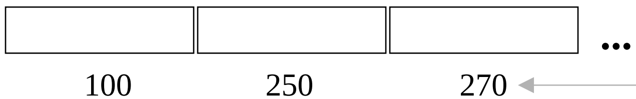
## Main ideas

1. Incremental GC (similar to train algo)




- Heap is divided into equally sized regions (~1MB)
- *Remembered set* per region (contain pointers from any region to this region; in contrast to train algo)

2. Collect regions with largest amount of garbage first




Regions are logically sorted by *collection costs*
- number of live bytes to be copied
- size of remembered set

3. Allocate new objects in "current region" (if full, start new current region)

# *G1 -- Computing live objects*

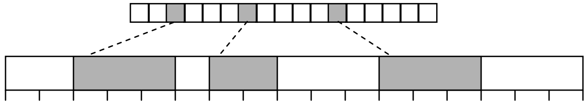
**Global marking phase** (started heuristically from time to time)

Mark all live objects (concurrently to the mutator)

```
foreach (root pointer r) mark(r);
while (!todo.empty()) {
    p = todo.remove();
    foreach (pointer q in *p) mark(q);
}
```

```
mark (p) {
    if (*p not marked) {
        mark *p;
        todo.add(p);
    }
}
```

Mark bits are kept in separate bitmap (1 bit per 8 bytes)



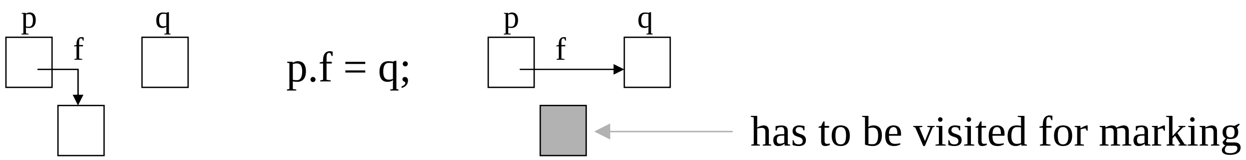
avoids synchronization between mutator and marker

# *G1 -- Building remembered sets*

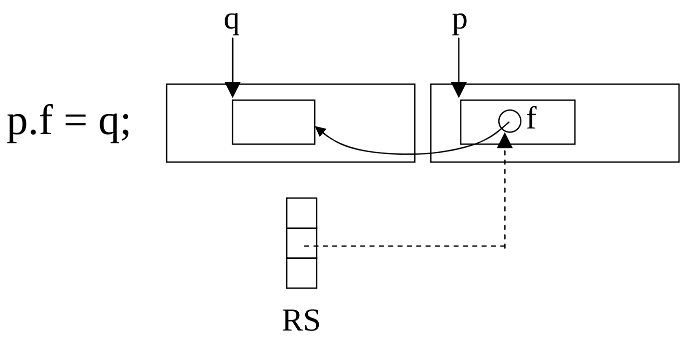
## Write barriers

Mutator threads use *write barriers* to catch pointer updates during marking.

*Snapshot at beginning*: make object grey if pointer to it is removed




Write barriers also build (update) the remembered sets



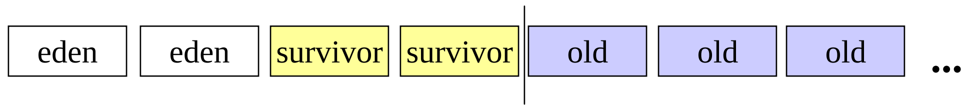
# *G1 -- Generations*

young:

| | |
|---|---|
| eden regions | those in which new objects have been allocated recently |
| survivor regions | contain objects with *age* < *tenureAge* |
| old regions | all other regions |

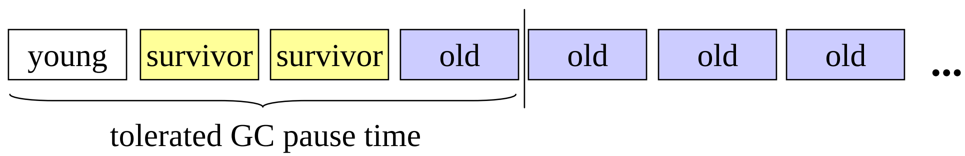
**Incremental evacuation step** (while all mutator threads are stopped)

Evacuate young regions to survivor regions or old regions




Adapt number of young regions such that evacuation does not exceed tolerated pause time

If time permits, evacuate also old regions with largest amount of garbage




For evacuation of region *R* use $roots_R$ and $RS_R$
After evacuation update remembered sets

For details see: David Detlefs et al.: Garbage-First Garbage Collection.
In Proc. Intl. Symp. on Memory Management (ISMM'04), Vancouver, Oct 24-25, 2004

# 2. Garbage Collection

2.1 Motivation

2.2 Basic techniques

- 2.2.1 Reference Counting
- 2.2.2 Mark & Sweep
- 2.2.3 Stop & Copy

2.3 Variants

- 2.3.1 Mark & Compact
- 2.3.2 Generation scavenging

2.4 Incremental garbage collection

- 2.4.1 Tricolor marking
- 2.4.2 Train algorithm

2.5 Root pointers

- 2.5.1 Pointer tables
- 2.5.2 Conservative garbage collection

2.6 Garbage collection in multi-threaded systems

2.7 Finalization

2.8 Case study: Java Hotspot VM

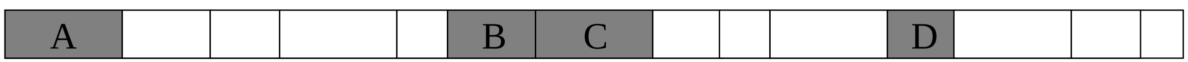
2.9 Case study: .NET

# *GC in .NET*

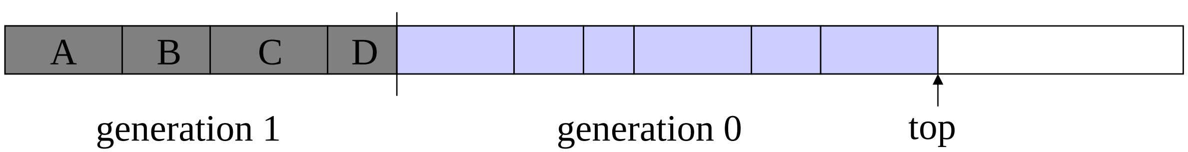
**Mark & Compact with multiple generations**

1. **Objects are allocated sequentially** (no free list)

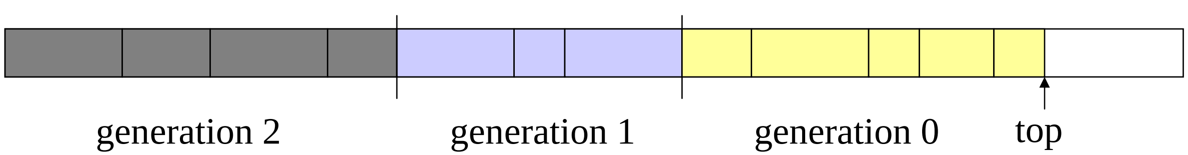

2. **If the heap is full => mark()**



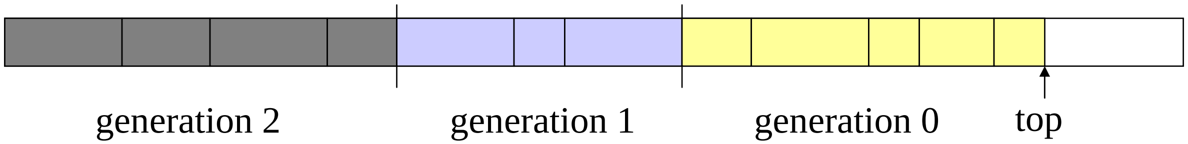

3. **compact()**
   New objects are allocated sequentially again




4. **If the heap is full => mark&compact only for generation 0!**
   faster; most dead objects are in generation 0



# GC in .NET




- Currently restricted to 3 generations
- From time to time there is a GC of generations 0+1 or generations 0+1+2 (heuristic)
- Pointers from generation 1+2 to generation 0 are detected with write barriers
  - *GetWriteWatch(..., oldGenArea, dirtyPages)* returns all dirty pages in *oldGenArea*
  - these must be searched for pointers to generation 0
- Objects larger than 20 KBytes are kept in a special heap (Mark&Sweep without compaction)
- GC of generation 0 takes less than 1 ms
- Threads are stopped at safepoints before the GC runs