Lecture 24

Public Service Announcement: "Hackers@Berkeley is hosting BEARHACK this Saturday at 11AM in the Wozniak Lounge. It's a 24 hour hackathon - there'll be tons of good food (Cheeseboard, sushi & boba!), activities (including massages!), and prizes (including an Occulus Rift).

Administrivia:

- There will be a homework assignment due next Wednesday night (but of course you can do it early!).
- Project 3 to be posted today or tomorrow.

Storage Management

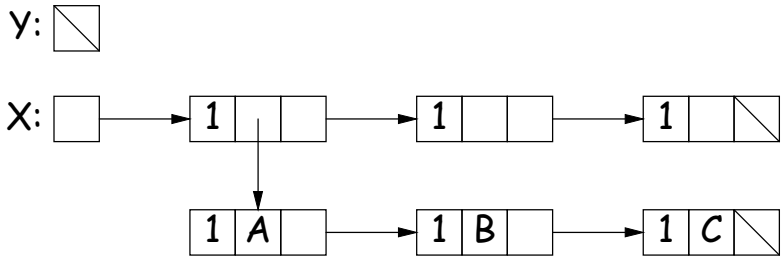
- Java has no means to free dynamic storage.
- However, when no expression in any thread can possibly be influenced by or change an object, it might as well not exist:

```
IntList wasteful ()
{
  IntList c = new IntList (3, new IntList (4, null));
  return c.tail;
  // variable c now deallocated, so no way
  // to get to first cell of list
}
```

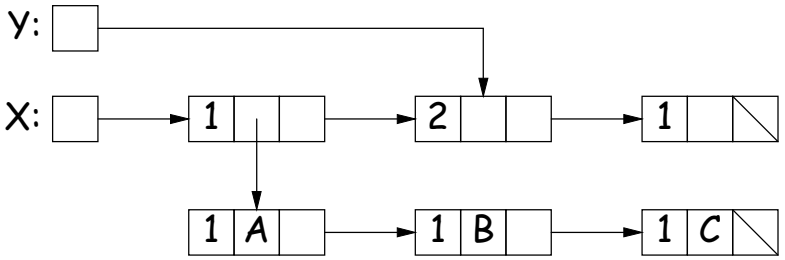
- At this point, Java runtime, like Scheme's, recycles the object `c` pointed to: *garbage collection*.

Garbage Collection: Reference Counting

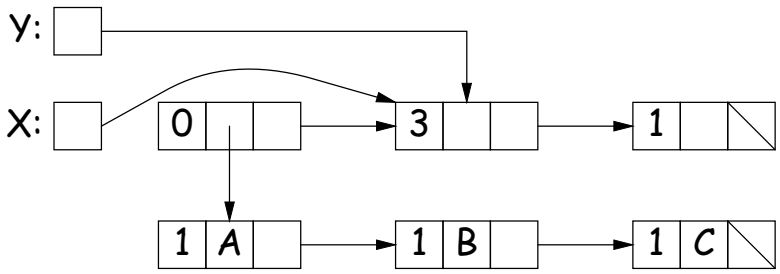
- Idea: Keep count of number of pointers to each object.

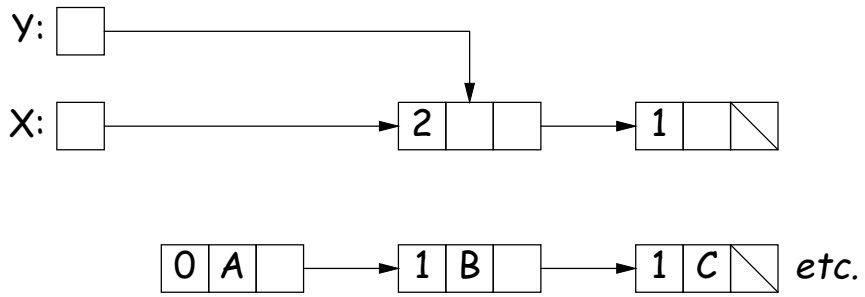



```
Y = X.tail;
```

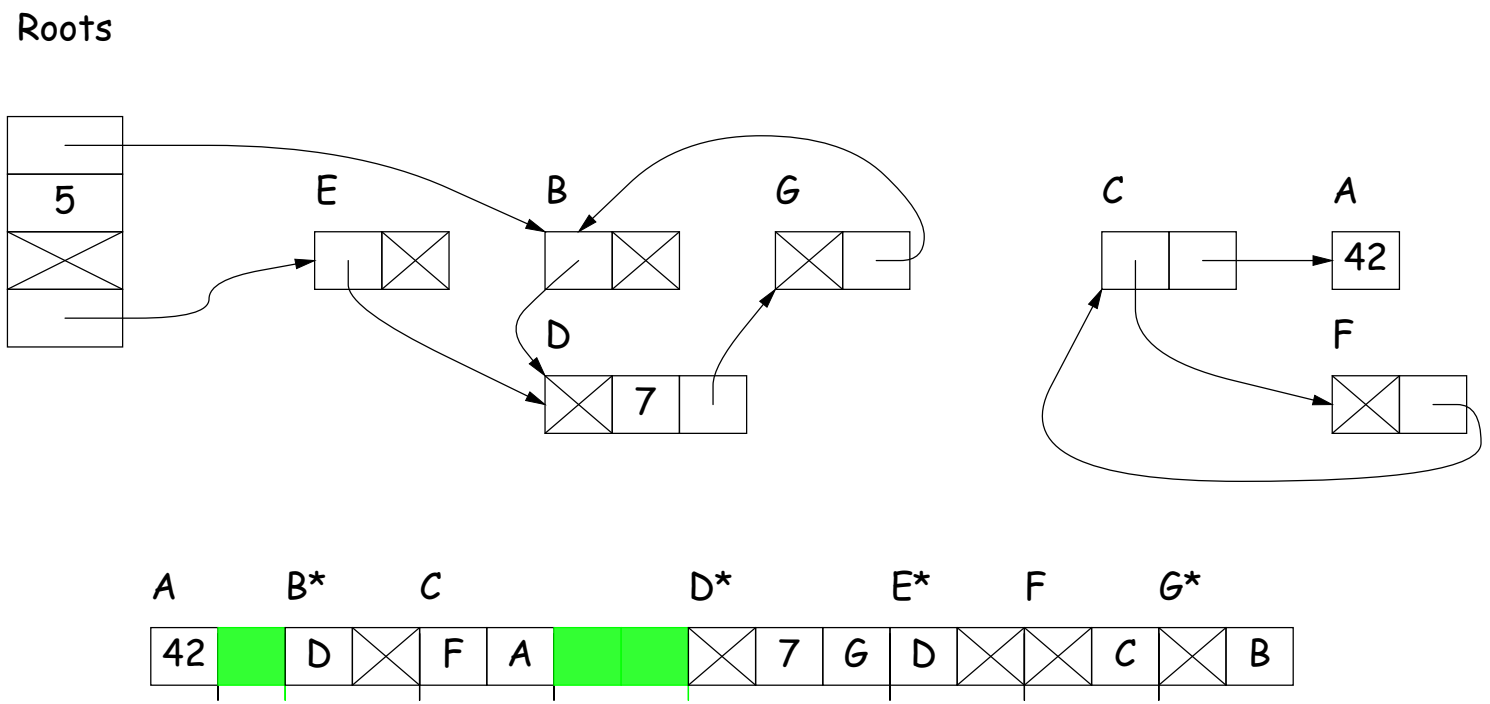



```
X = Y;
```



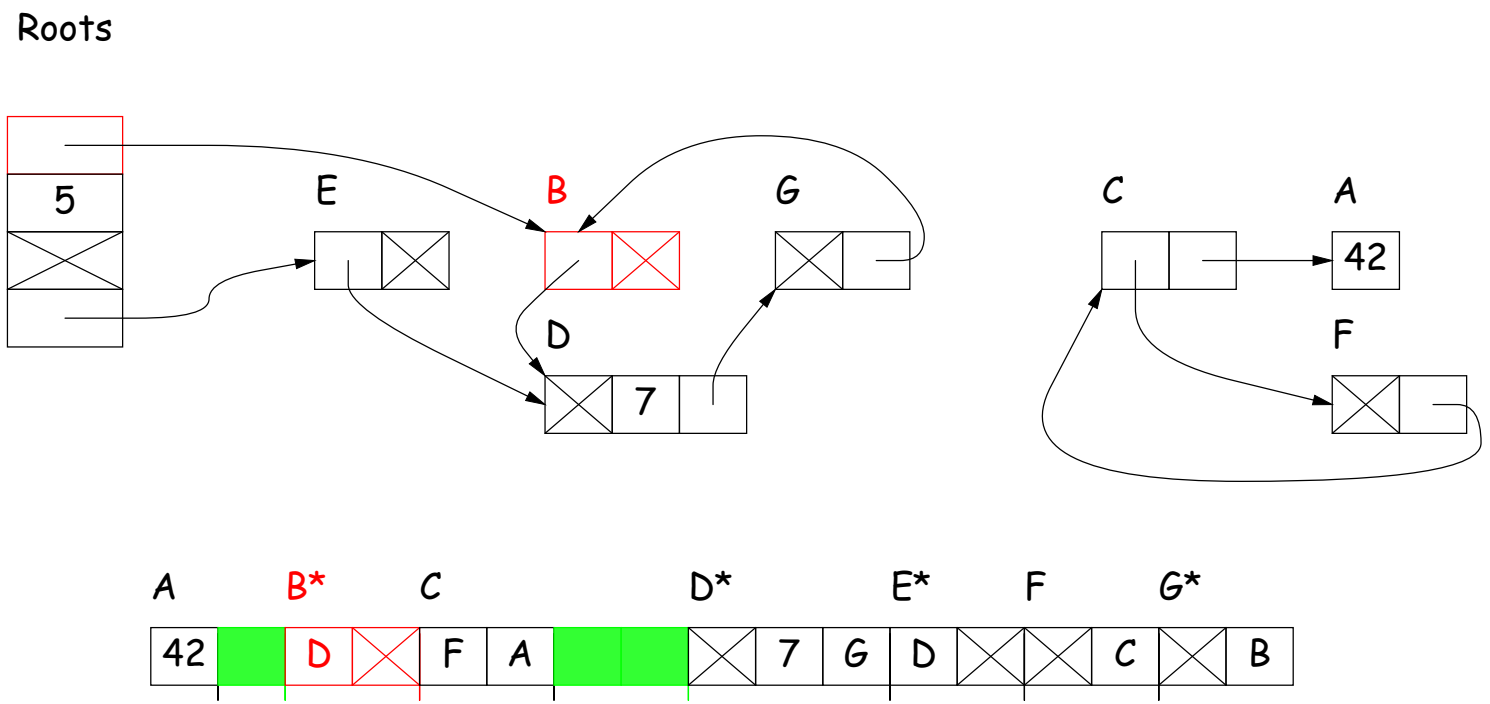


Garbage Collection: Mark and Sweep



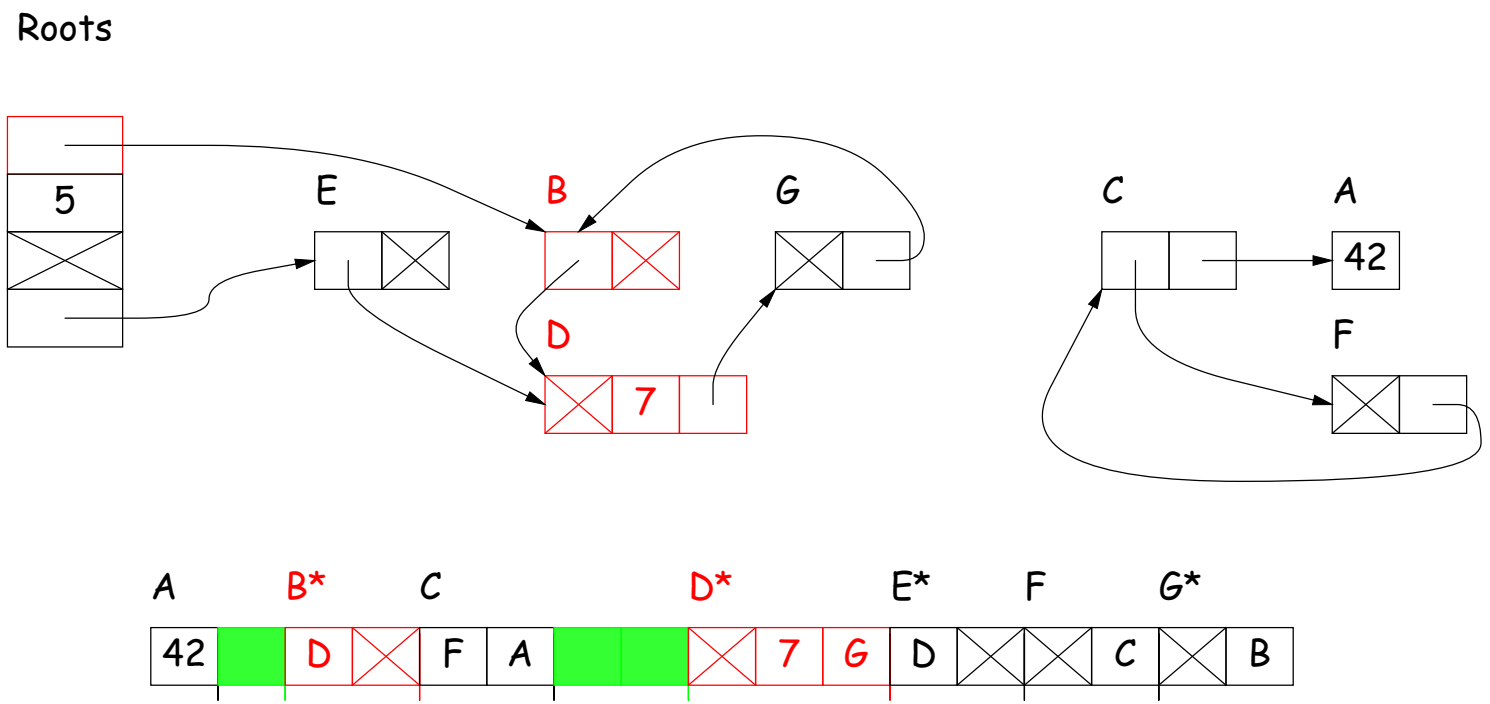
- Start at roots (named variables, static and on stack)
- Perform graph traversal to find and mark all reachable storage.
- Sweep over memory, adding all unmarked storage to free list.

Garbage Collection: Mark and Sweep



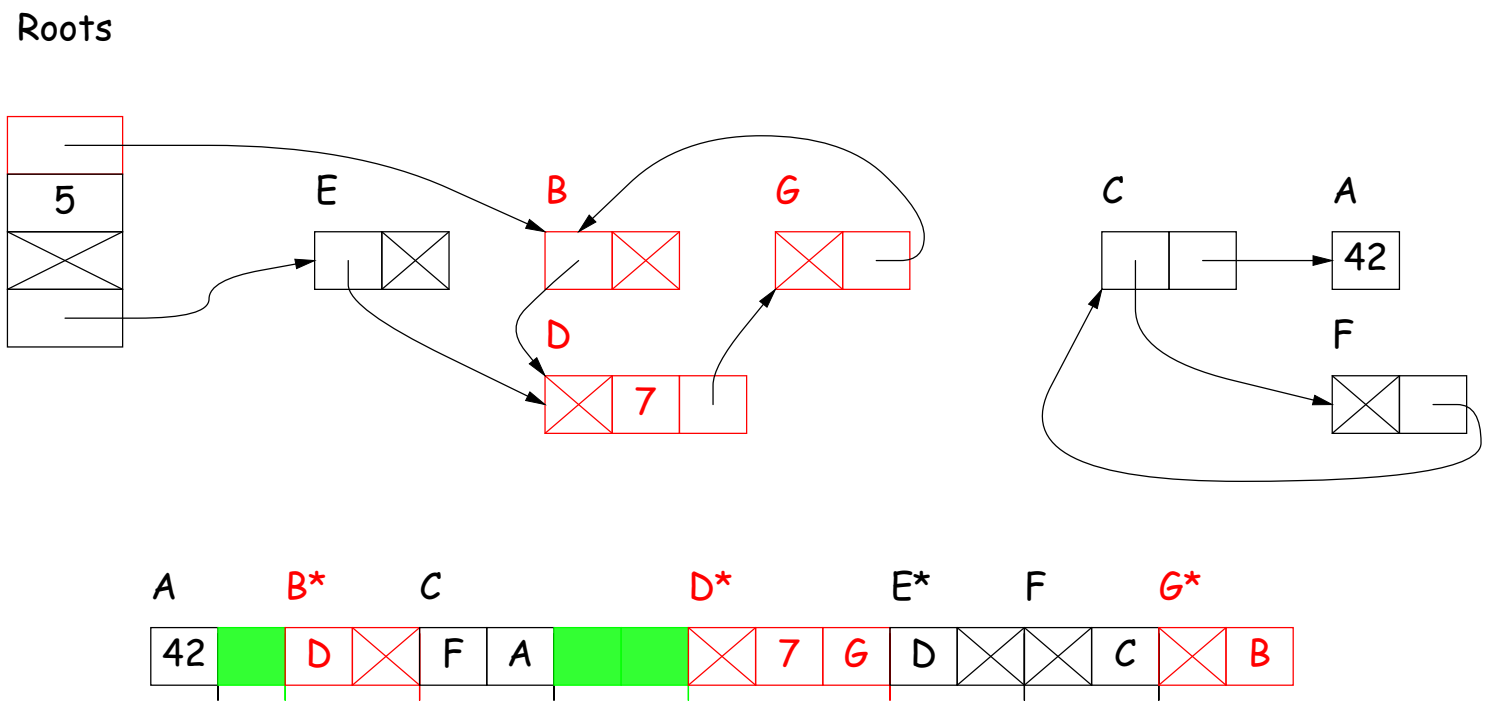
- Start at roots (named variables, static and on stack)
- Perform graph traversal to find and mark all reachable storage.
- Sweep over memory, adding all unmarked storage to free list.

Garbage Collection: Mark and Sweep



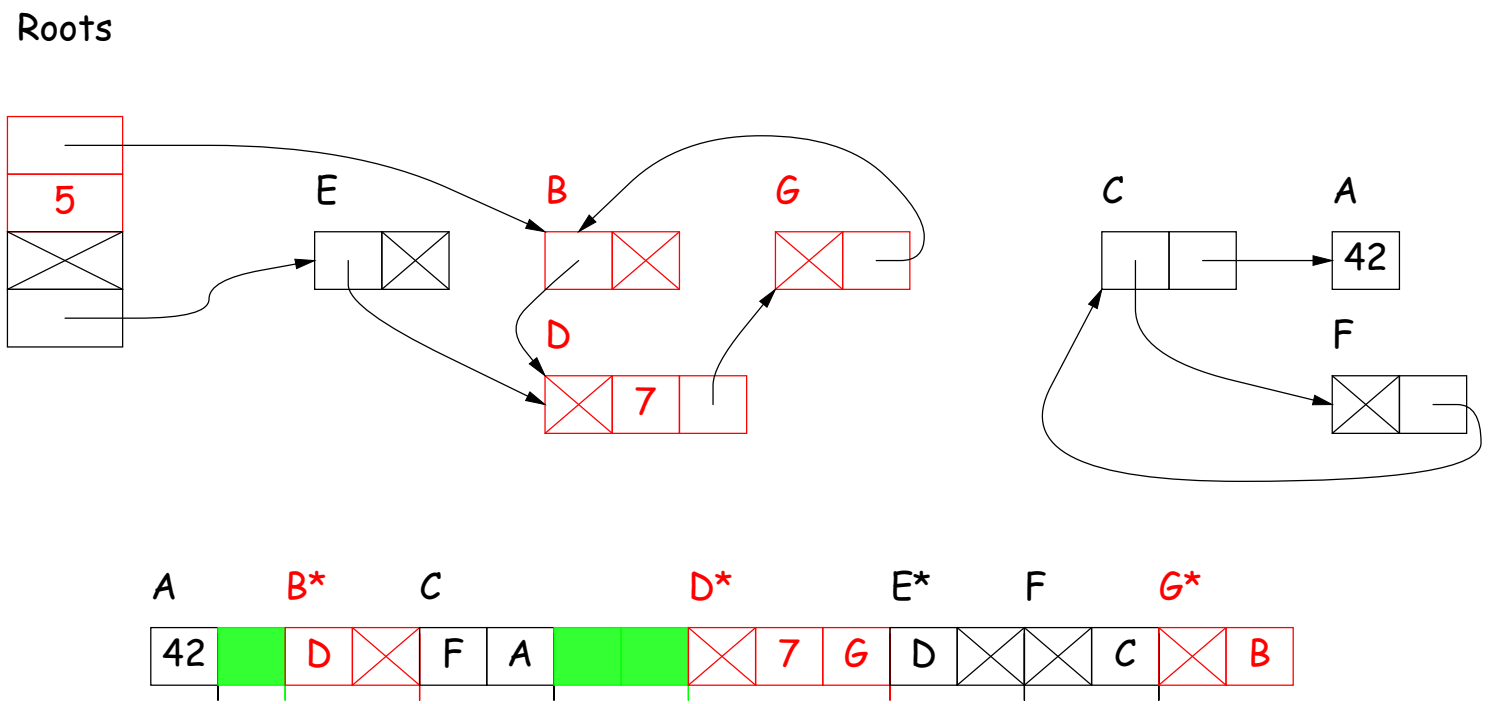
- Start at roots (named variables, static and on stack)
- Perform graph traversal to find and mark all reachable storage.
- Sweep over memory, adding all unmarked storage to free list.

Garbage Collection: Mark and Sweep



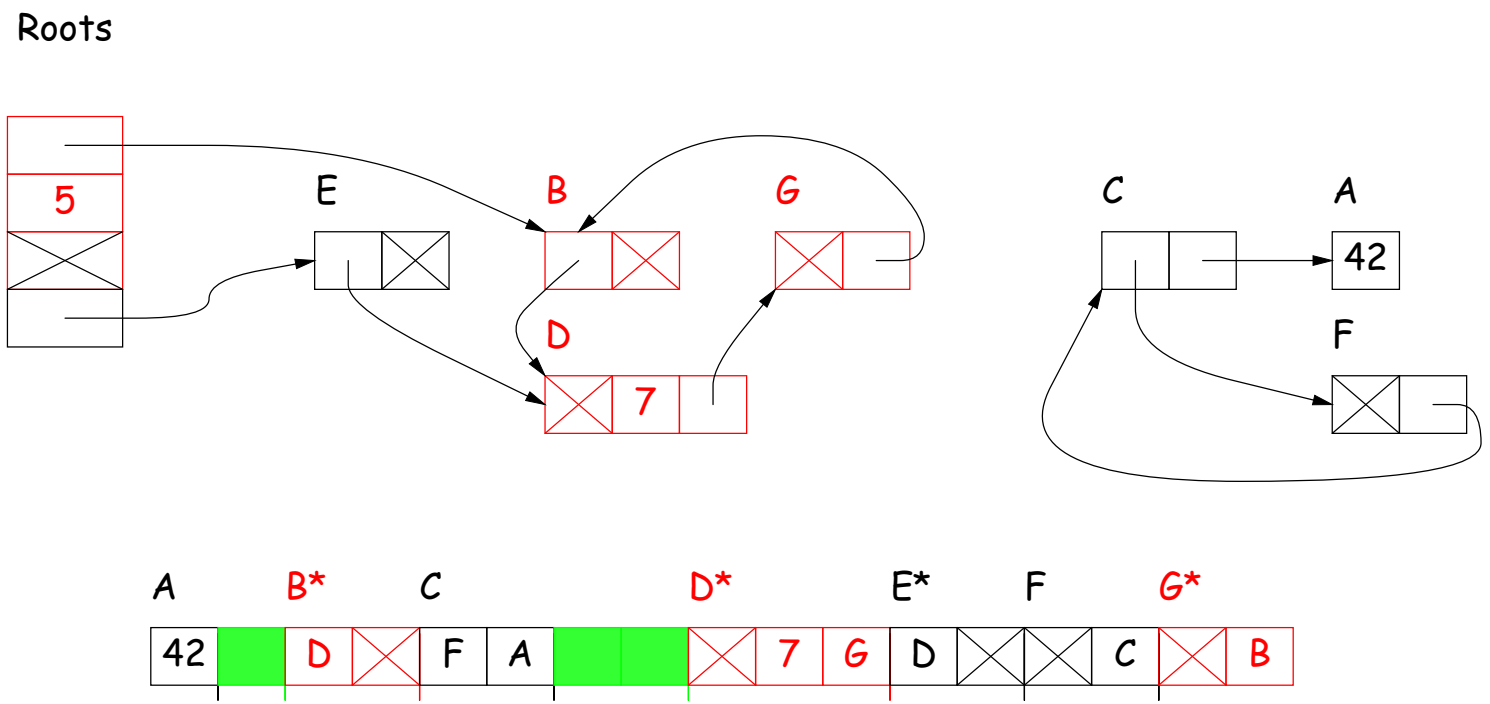
- Start at roots (named variables, static and on stack)
- Perform graph traversal to find and mark all reachable storage.
- Sweep over memory, adding all unmarked storage to free list.

Garbage Collection: Mark and Sweep



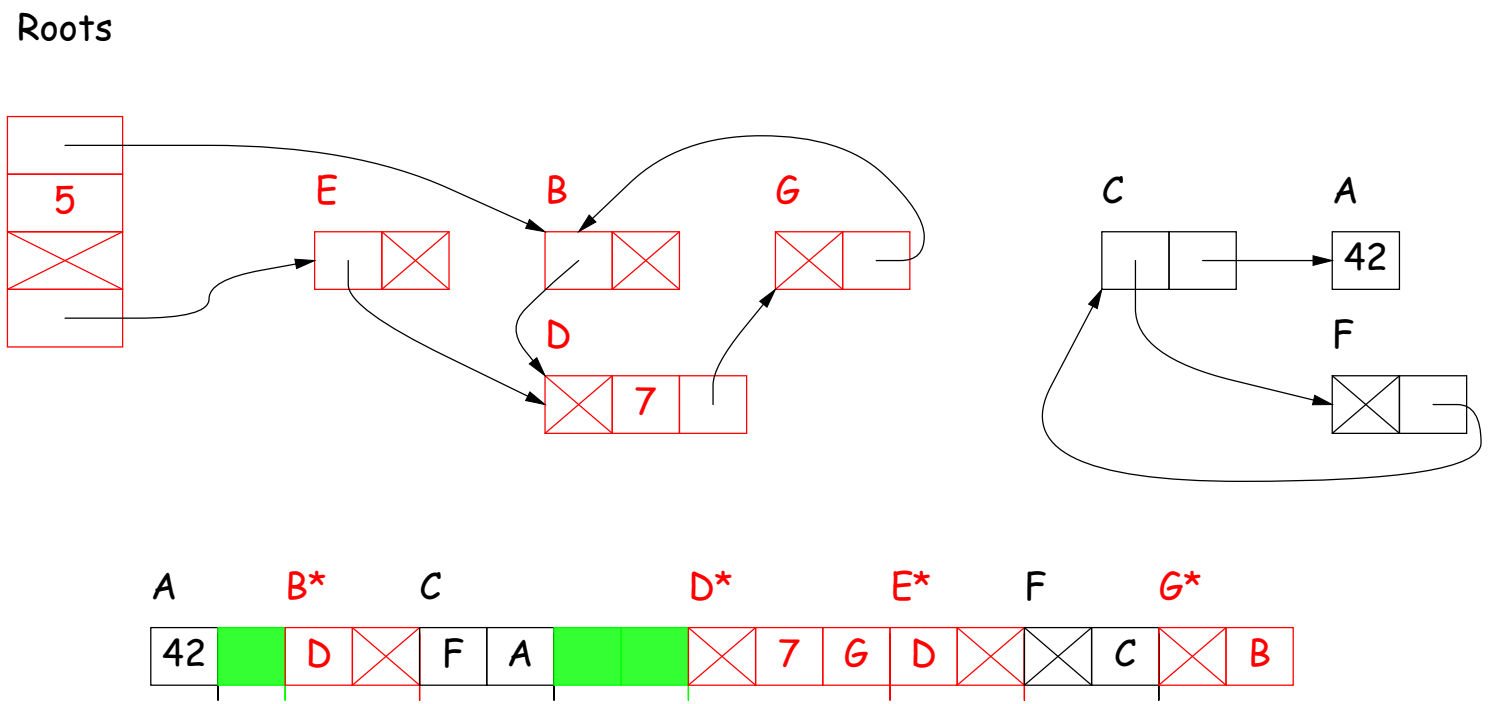
- Start at roots (named variables, static and on stack)
- Perform graph traversal to find and mark all reachable storage.
- Sweep over memory, adding all unmarked storage to free list.

Garbage Collection: Mark and Sweep



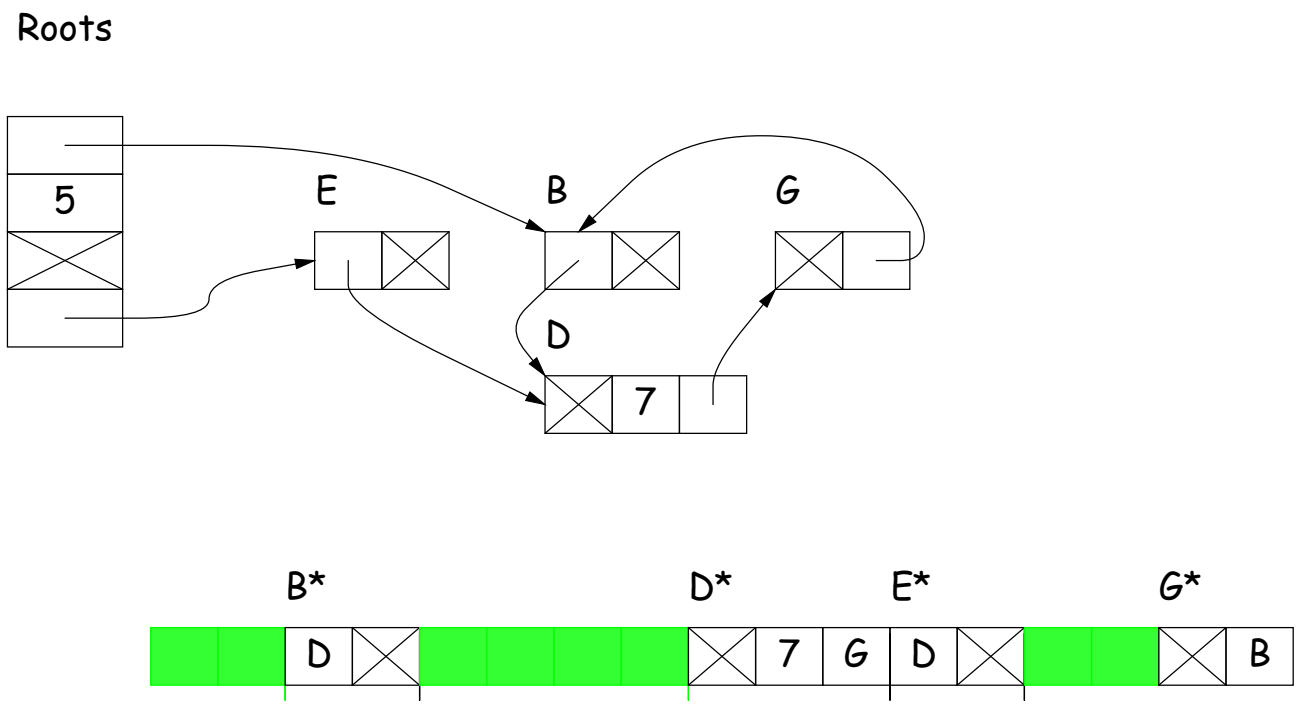
- Start at roots (named variables, static and on stack)
- Perform graph traversal to find and mark all reachable storage.
- Sweep over memory, adding all unmarked storage to free list.

Garbage Collection: Mark and Sweep



- Start at roots (named variables, static and on stack)
- Perform graph traversal to find and mark all reachable storage.
- Sweep over memory, adding all unmarked storage to free list.

Garbage Collection: Mark and Sweep

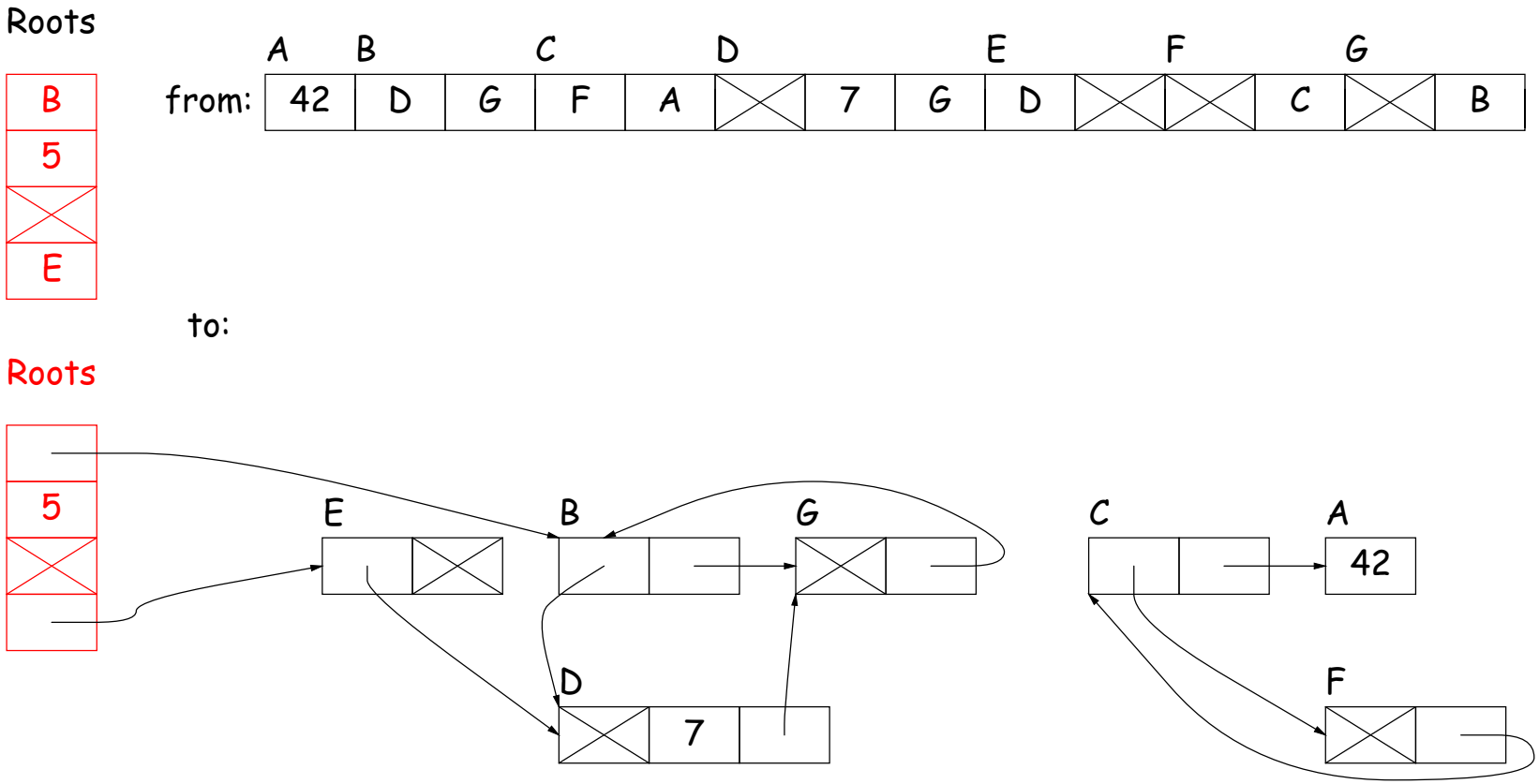


- Start at roots (named variables, static and on stack)
- Perform graph traversal to find and mark all reachable storage.
- Sweep over memory, adding all unmarked storage to free list.

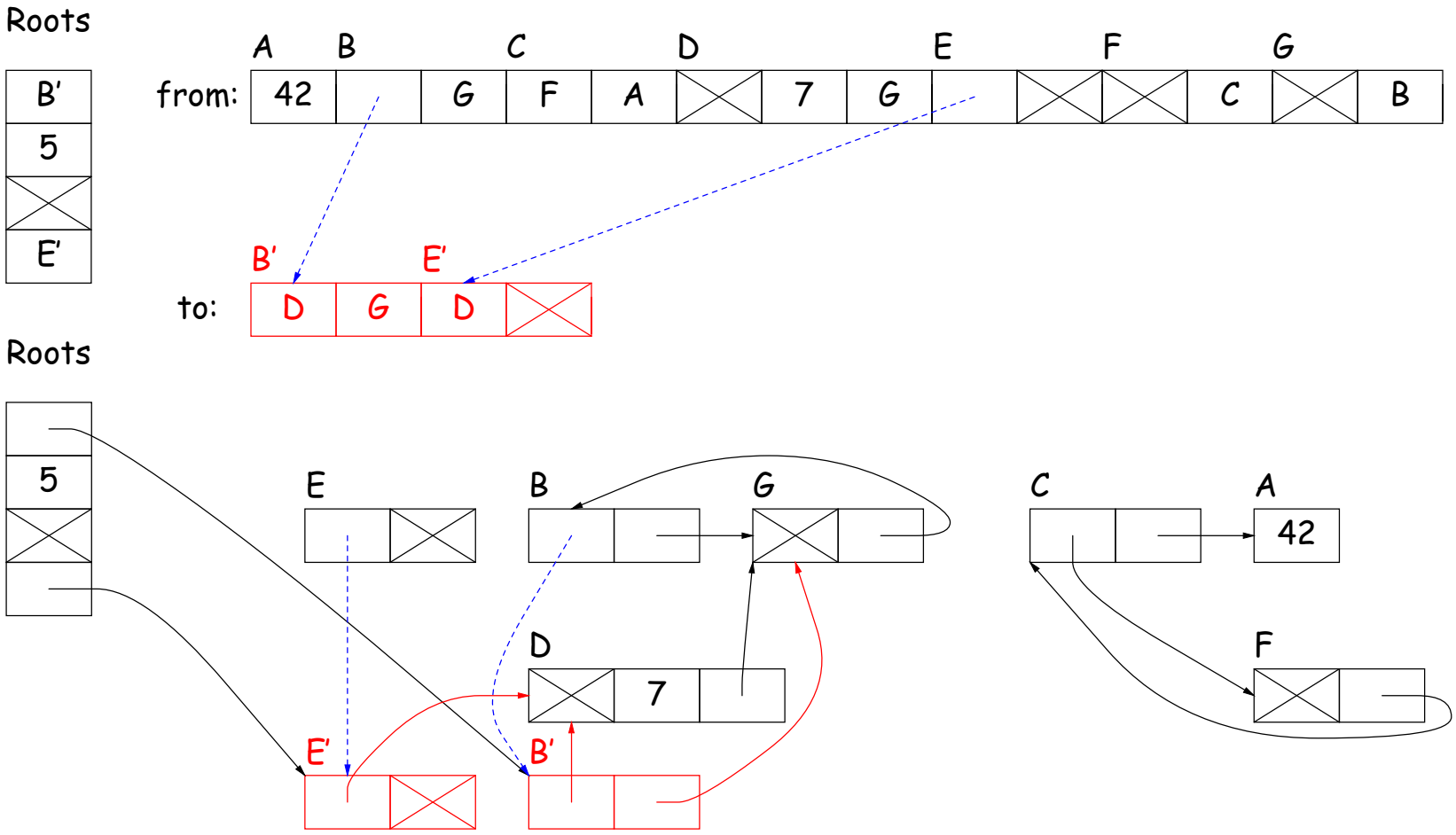
Copying Garbage Collection

- Copy (and move) only reachable (useful) storage from 'from' space to 'to' space.
- The 'from' and 'to' areas are called *semispaces*. Need twice the virtual memory you actually use.
- As you copy, mark 'from' storage as moved, and leave behind a *forwarding pointer* that tells how to translate other references to the old storage.
- At end of algorithm, 'from' and 'to' swap roles, and the old 'from' area is freed *en masse*.
- Copied storage is *compacted* (gaps squeezed out) with possible advantages for memory access.

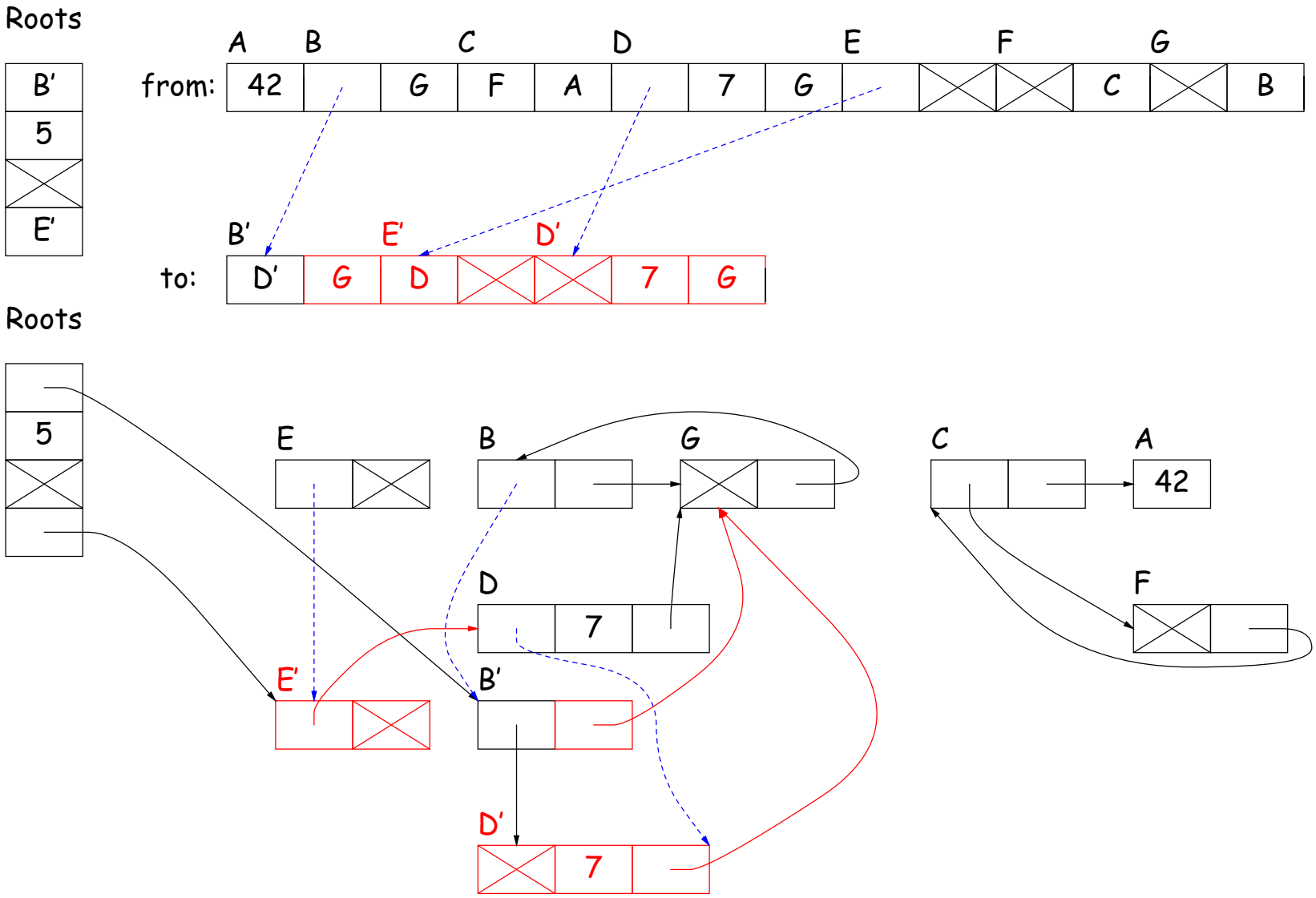
Copying Garbage Collection, Illustrated



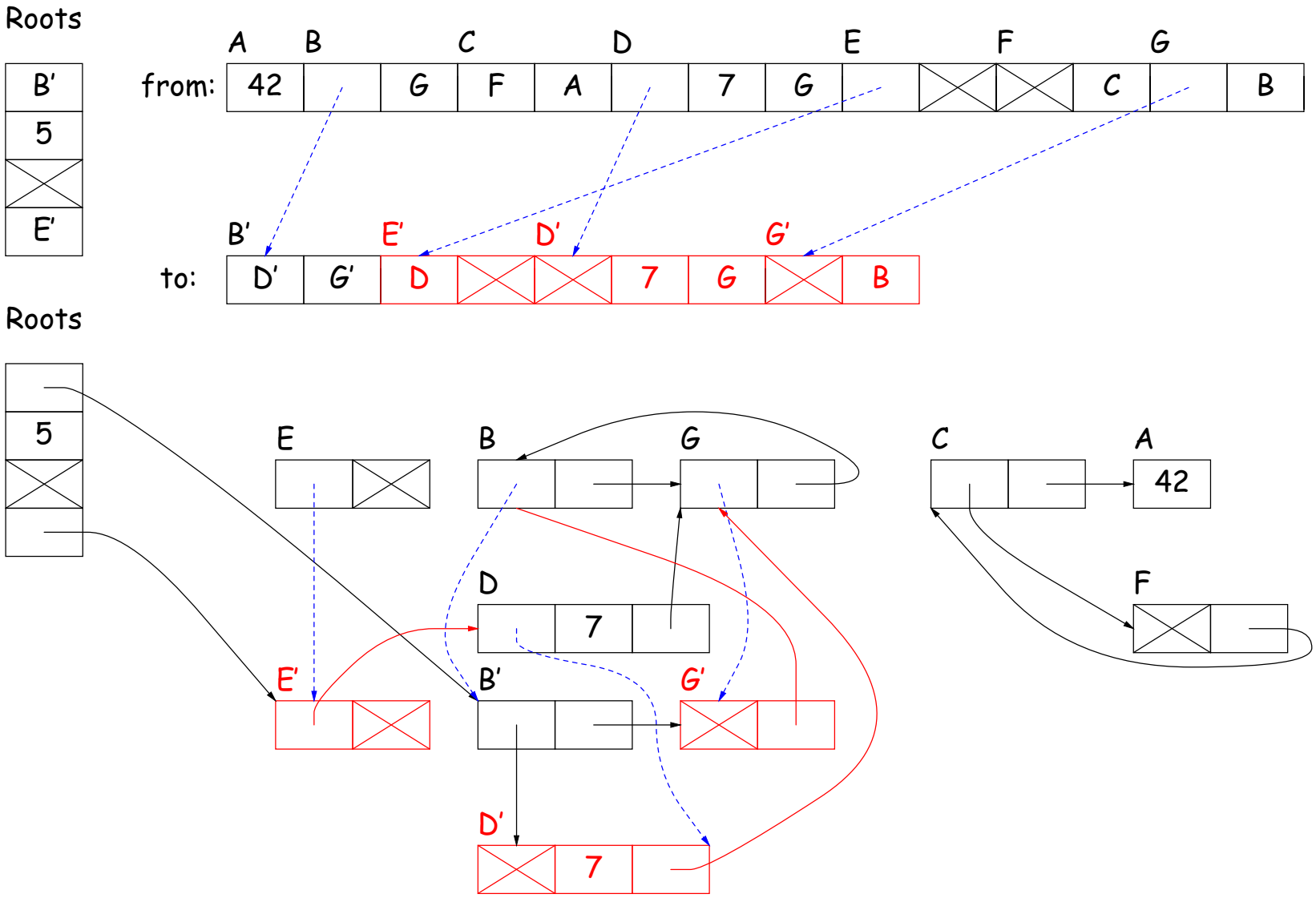
Copying Garbage Collection, Illustrated



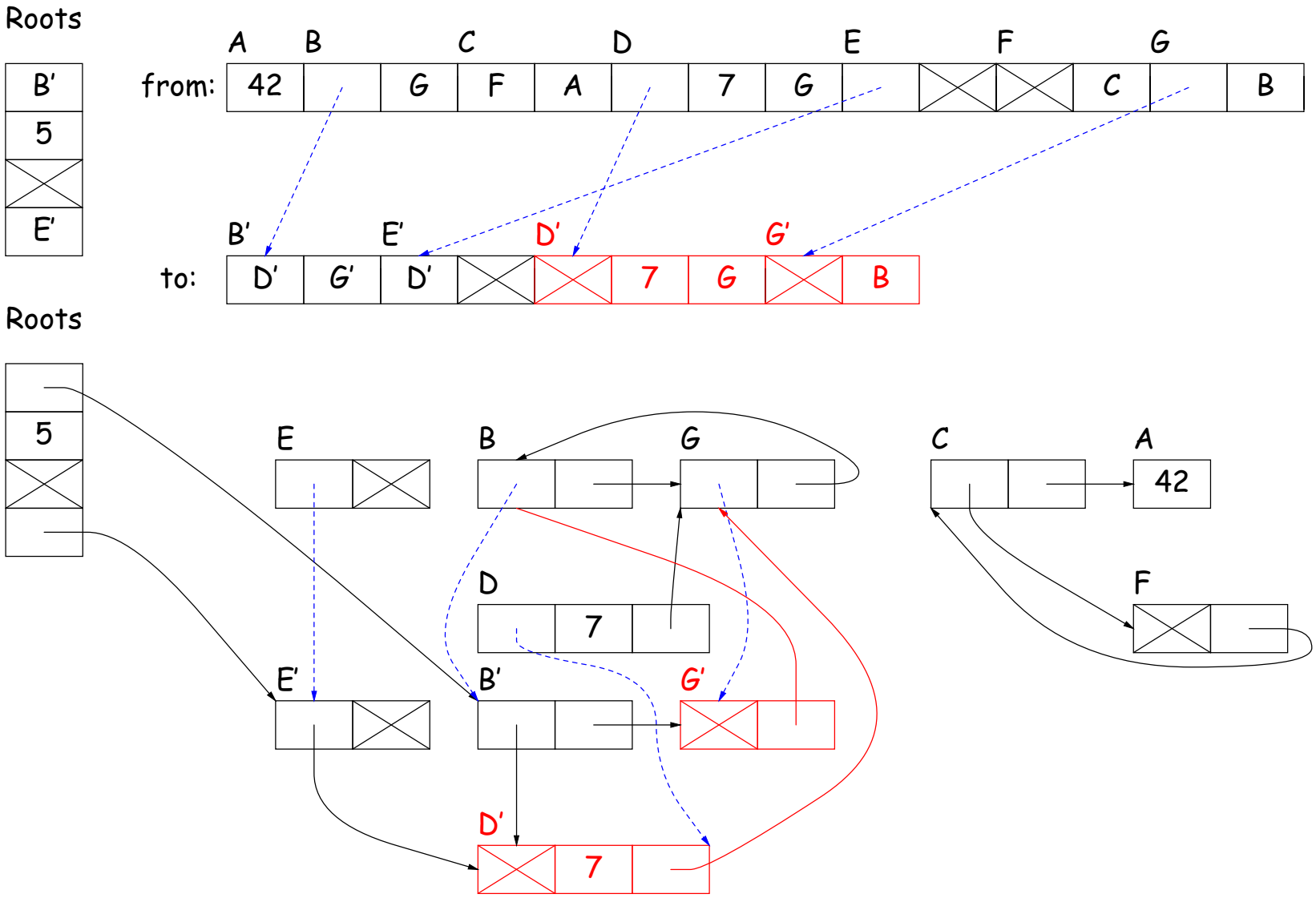
Copying Garbage Collection, Illustrated



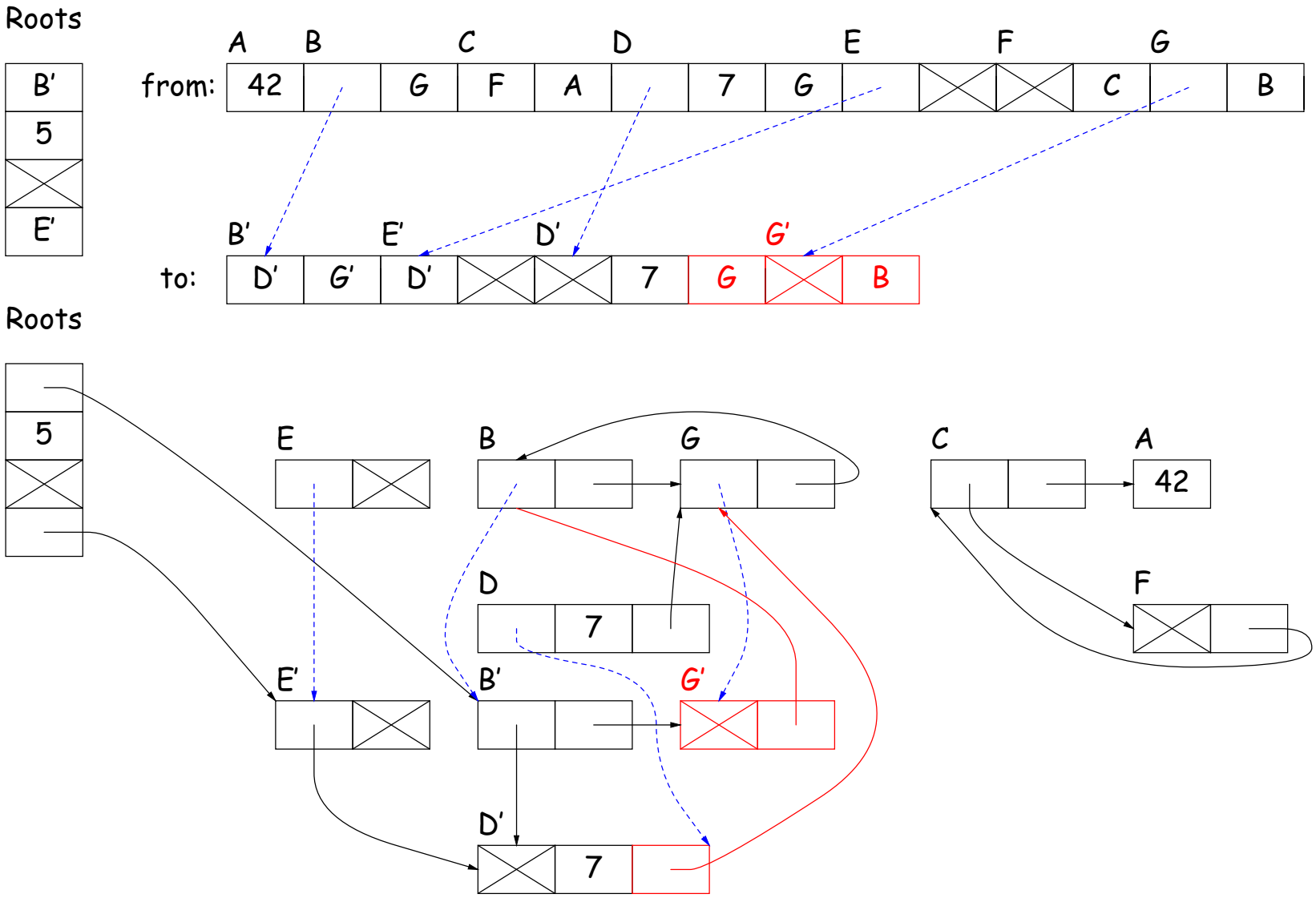
Copying Garbage Collection, Illustrated



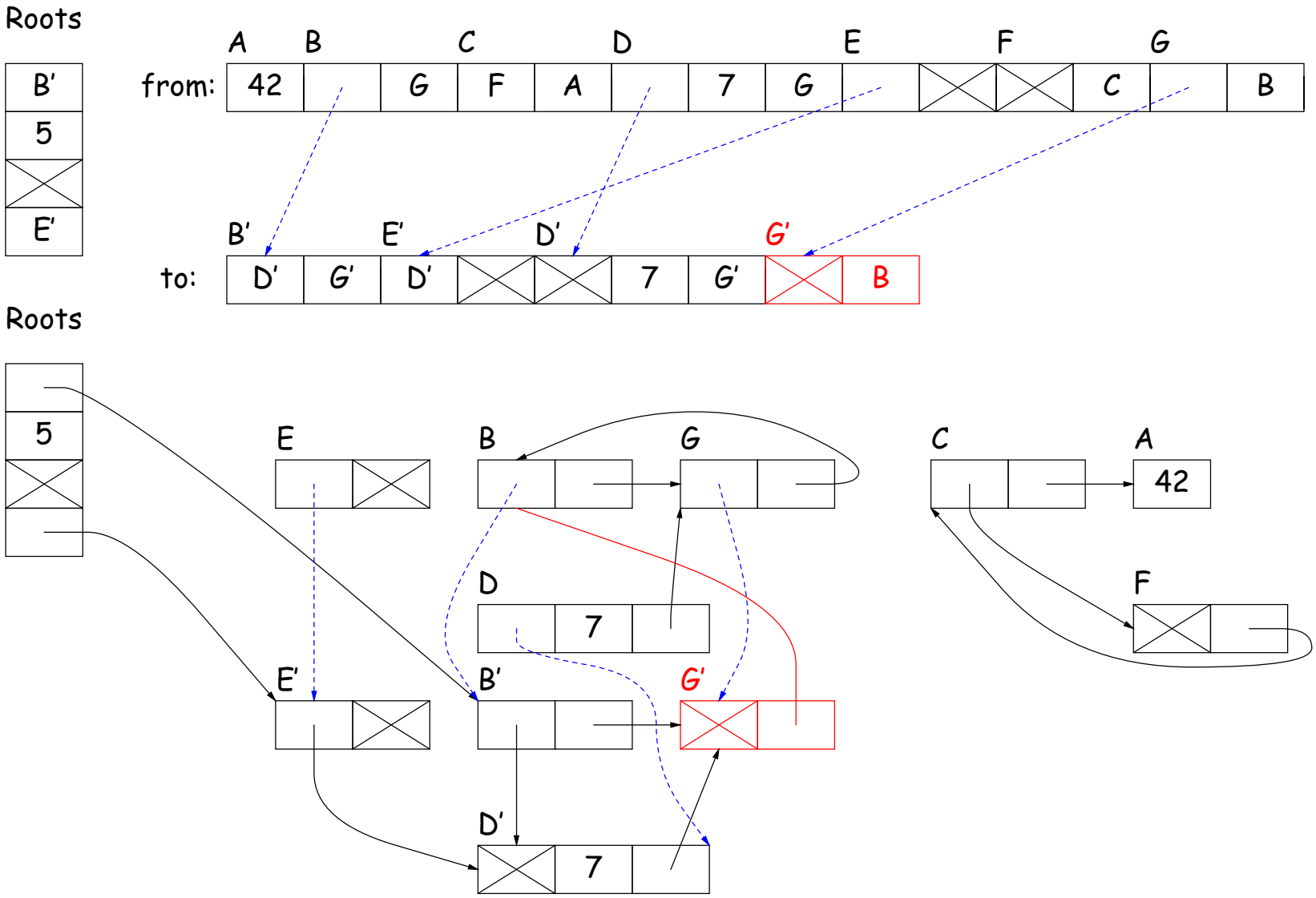
Copying Garbage Collection, Illustrated



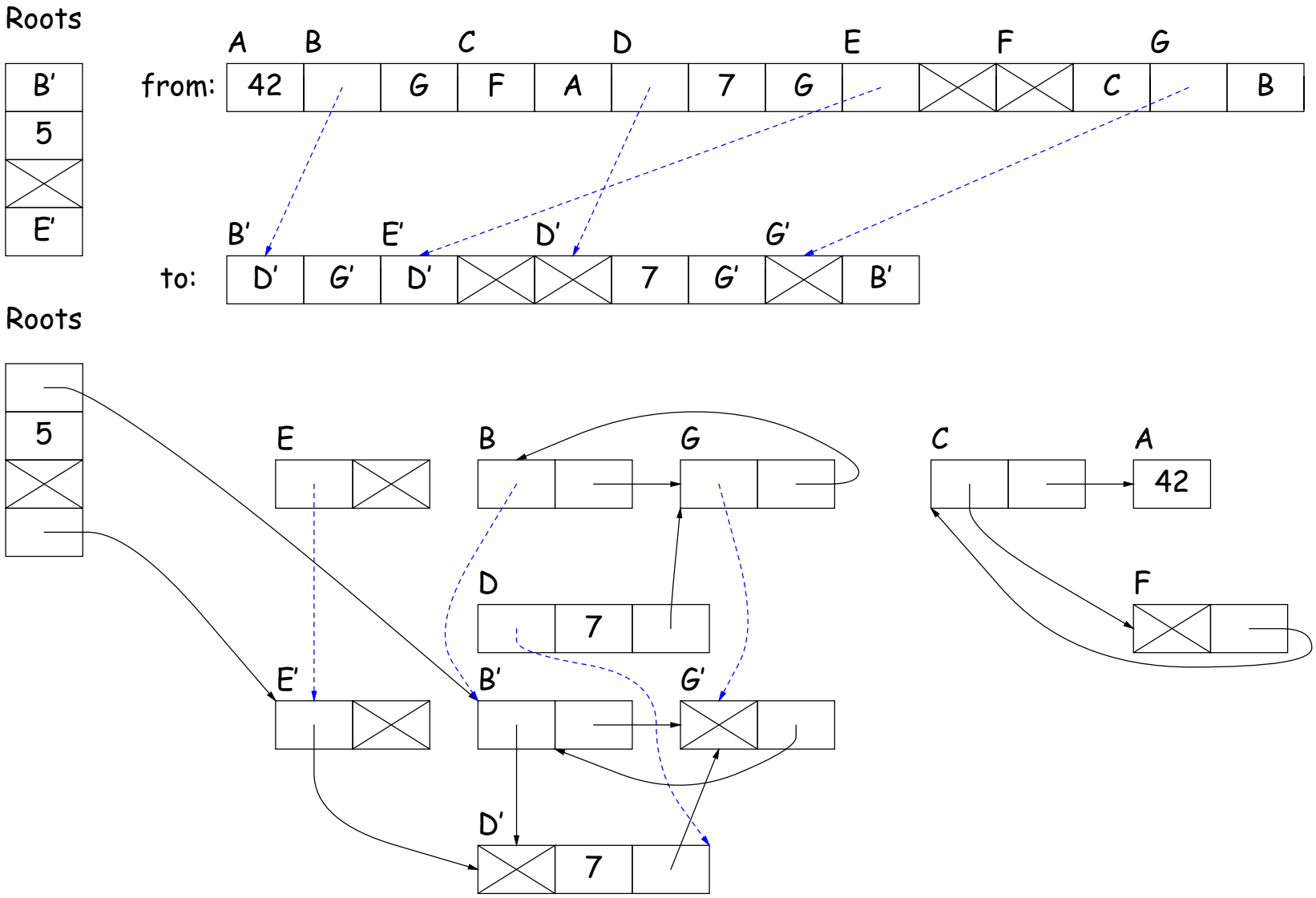
Copying Garbage Collection, Illustrated



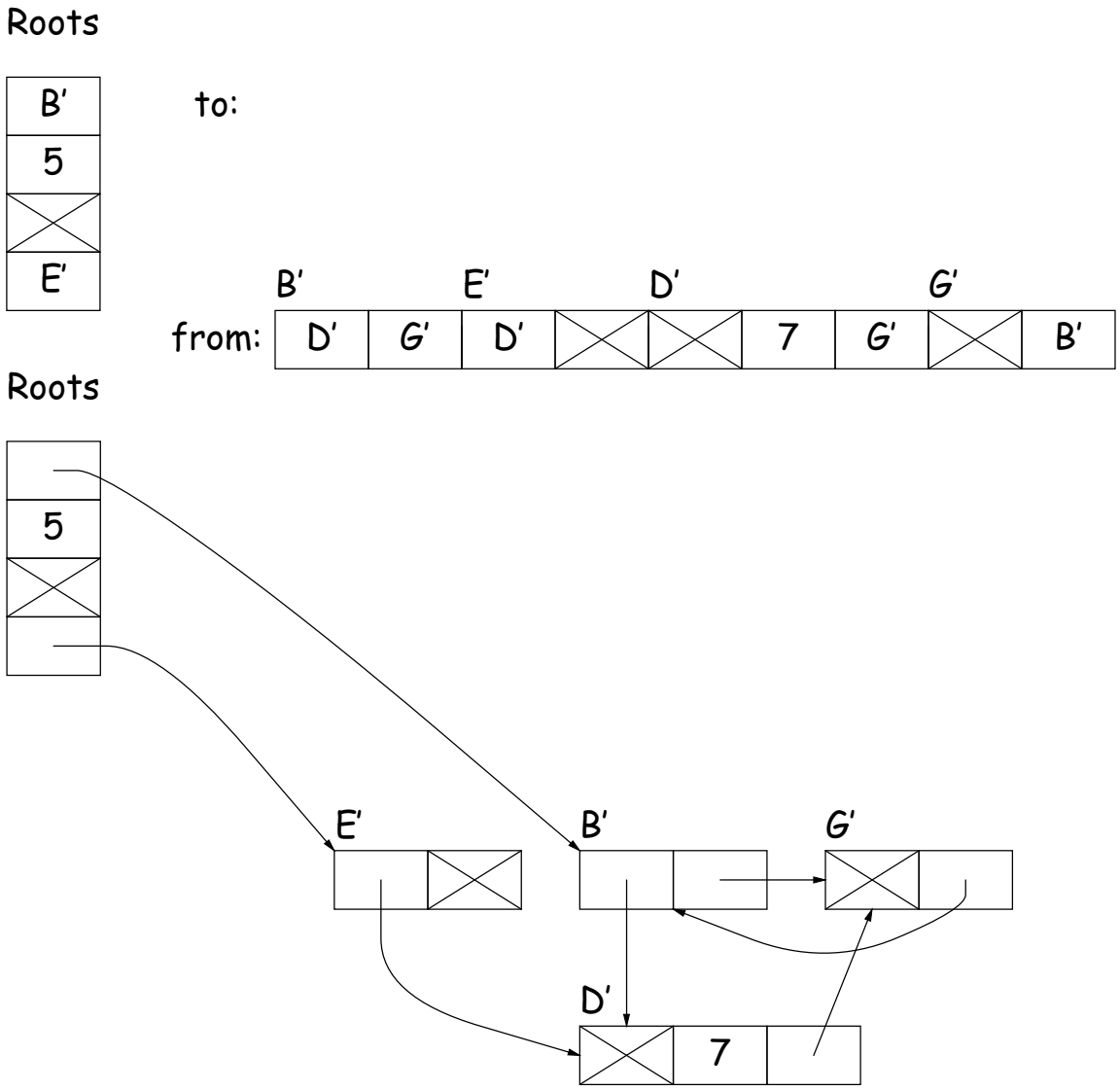
Copying Garbage Collection, Illustrated



Copying Garbage Collection, Illustrated



Copying Garbage Collection, Illustrated



Roots and Other Pointers

- Above methods require that we know locations of roots and of pointer fields in objects.
- Positions of some roots change during execution.
- Compiler keeps tables mapping PC to where roots are.
- Runtime type information (virtual tables) keep information of where pointer fields are.
- Implementation must guarantee that fields are initialized.

Conservative Garbage Collection

- With C, you have none of the needed information.
- But easy to know the addresses of allocated storage, and sizes of allocated objects (allocator keeps them around).
- So, guess that any word that looks like an address of allocated storage is a valid address.
- Do mark-and-sweep on this assumption (look at whole stack and static storage for roots).
- Marks some garbage, but can be surprisingly effective.

Generational Garbage Collection

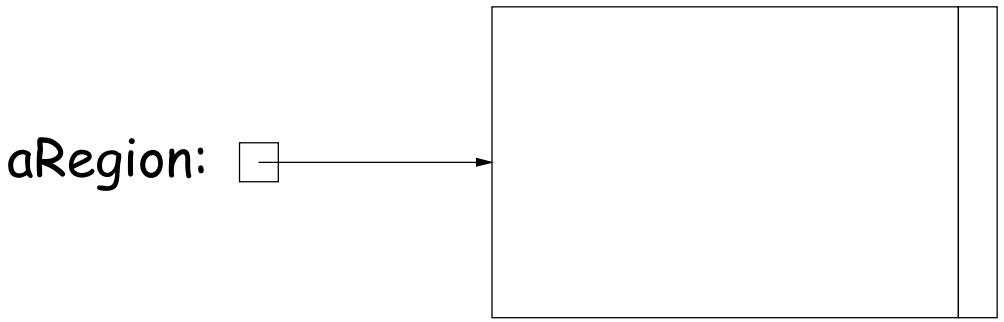
- Heap storage tends to "die young."
- So divide memory into young and old storage, and do copying only on young storage.
- Must add old storage that points to young storage to roots.
- When young storage survives a GC (or two), move it to old storage.
- Every now and then, stop the world and do a full garbage collection.
- This technique significantly speeds up GC.

Region-Based Allocation

- Garbage collection (all forms) does incur overheads, which can be unpredictable,
- While manual freeing is prone to error and inconvenient.
- One compromise is *region-based allocation*.
- Idea:
  - Create a data structure known as a region (or zone, or arena, or various other names).
  - Provides two operations: allocate object, and free *all* objects.
- Thus, to perform calculation that creates lots of temporary heap objects,
  - Create region (a local variable).
  - Allocate all the temporary storage in this region.
  - Delete whole region at end.

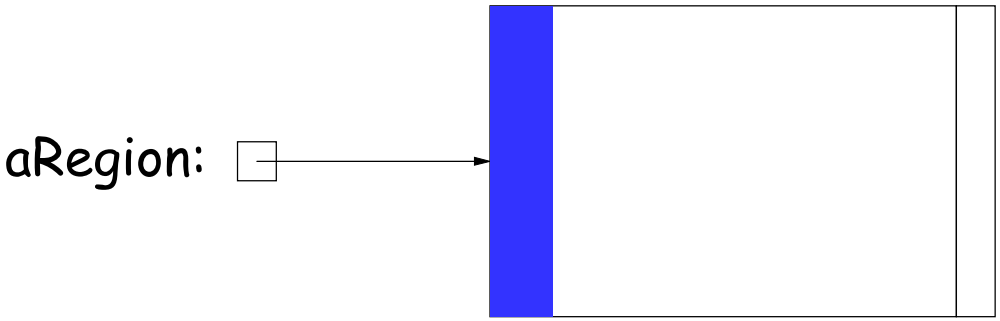
Region Implementation

- Simple implementation: allocate storage in big blocks, and allocate objects sequentially in the blocks.
- Freeing all blocks frees all the objects quickly.



Region Implementation

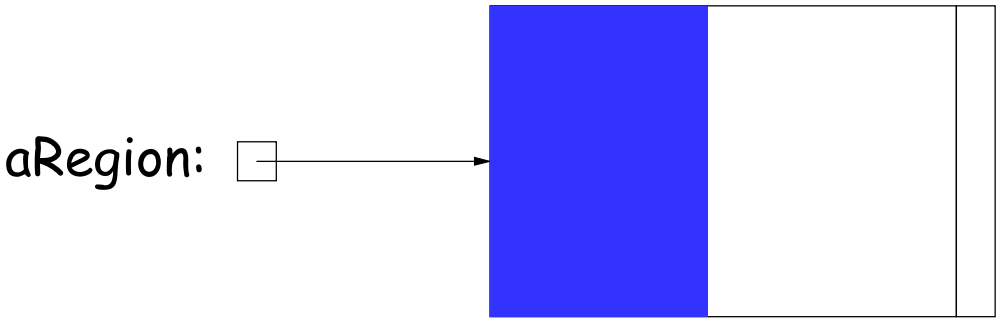
- Simple implementation: allocate storage in big blocks, and allocate objects sequentially in the blocks.
- Freeing all blocks frees all the objects quickly.




```
x = aRegion.alloc (40);
```

Region Implementation

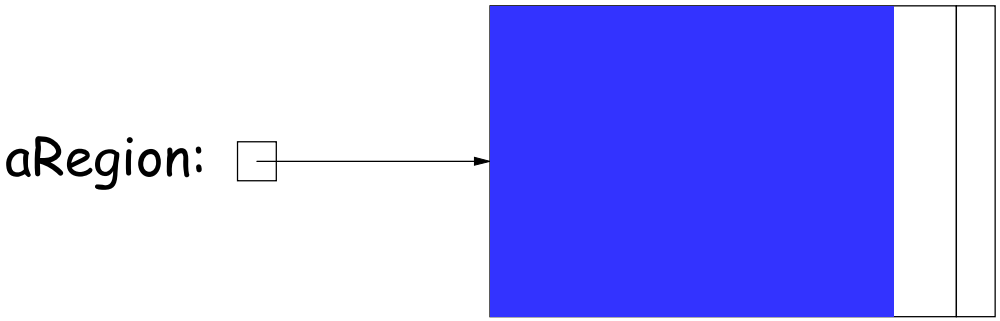
- Simple implementation: allocate storage in big blocks, and allocate objects sequentially in the blocks.
- Freeing all blocks frees all the objects quickly.




```
x = aRegion.alloc (40);
y = aRegion.alloc (100);
```

Region Implementation

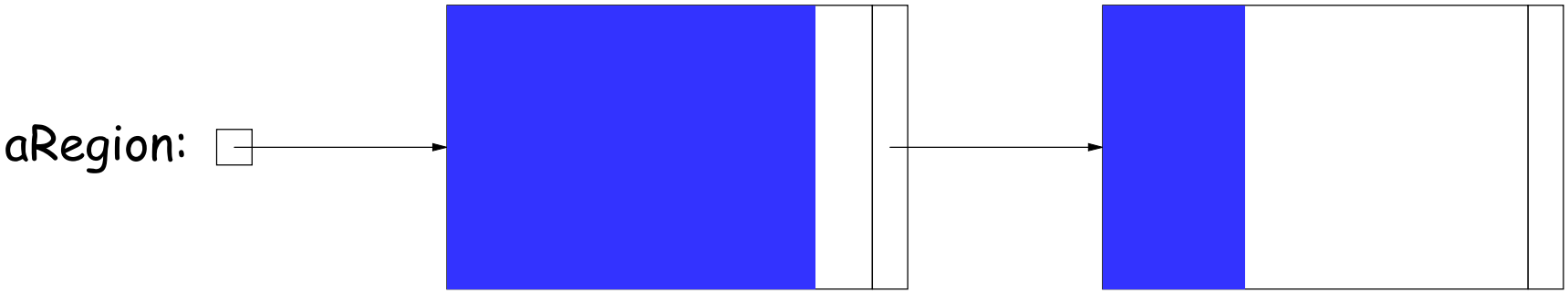
- Simple implementation: allocate storage in big blocks, and allocate objects sequentially in the blocks.
- Freeing all blocks frees all the objects quickly.




```
x = aRegion.alloc (40);
y = aRegion.alloc (100);
z = aRegion.alloc (120);
```

Region Implementation

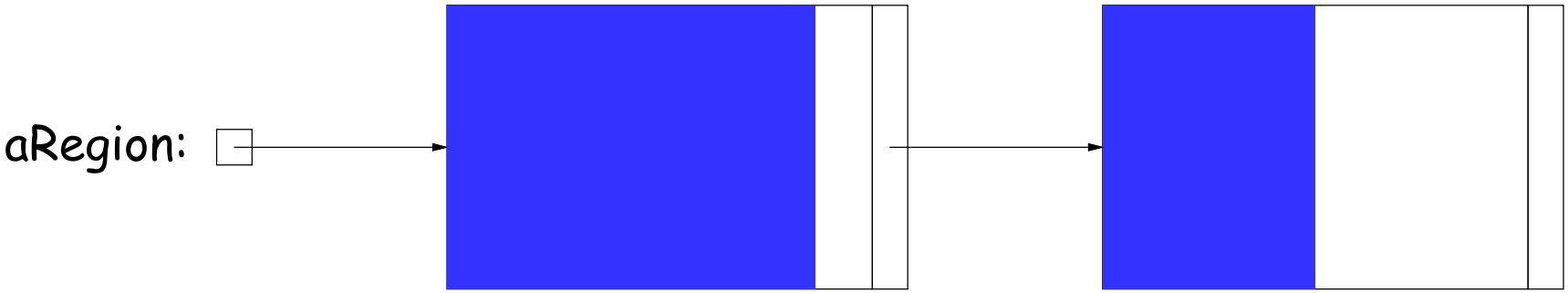
- Simple implementation: allocate storage in big blocks, and allocate objects sequentially in the blocks.
- Freeing all blocks frees all the objects quickly.




```
x = aRegion.alloc (40);
y = aRegion.alloc (100);
z = aRegion.alloc (120);
v = aRegion.alloc (100);
```

Region Implementation

- Simple implementation: allocate storage in big blocks, and allocate objects sequentially in the blocks.
- Freeing all blocks frees all the objects quickly.




```
x = aRegion.alloc (40);
y = aRegion.alloc (100);
z = aRegion.alloc (120);
v = aRegion.alloc (100);
w = aRegion.alloc (50);
```

Region Implementation

- Simple implementation: allocate storage in big blocks, and allocate objects sequentially in the blocks.
- Freeing all blocks frees all the objects quickly.

aRegion: 

```
x = aRegion.alloc (40);
y = aRegion.alloc (100);
z = aRegion.alloc (120);
v = aRegion.alloc (100);
w = aRegion.alloc (50);
aRegion.freeAll ();
```

Region Implementation

- Simple implementation: allocate storage in big blocks, and allocate objects sequentially in the blocks.
- Freeing all blocks frees all the objects quickly.

```
x = aRegion.alloc (40);
y = aRegion.alloc (100);
z = aRegion.alloc (120);
v = aRegion.alloc (100);
w = aRegion.alloc (50);
aRegion.freeAll ();
```

- Potential problem: using x, y, z, . . . after freeAll.