Reference counting

(is terrible)

The big idea

- GC: We’re pausing the mutator to scan all reachable references
- RC: Instead, have the mutator tell us when references change
- Essentially, “mark” in real time, “sweep” when an object when no references remain

The compiler part

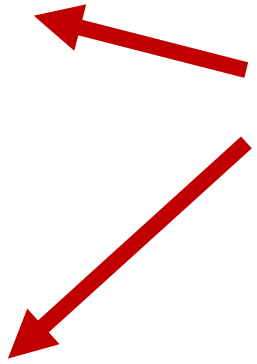
- Need a “write barrier”
- Every time the mutator writes a reference, it does some work for us
- Compiler’s job to promise this

Algorithm

```
writeReference(loc, newVal):
  if newVal != NULL:
    newVal->header.refCount += 1
  deref(*loc)
  *loc = newVal

deref(ref):
  if ref != NULL:
    ref->header.refCount -= 1
    if ref->header.refCount == 0:
      recFree(ref)

recFree(ref):
  foreach loc in ref->header.descriptor->ptrs:
    deref(*(ref+loc))
  free(ref)
```

Algorithm

```
writeReference(loc, newVal):
  if newVal != NULL:
    newVal->header.refCount += 1
  deref(*loc)
  *loc = newVal

deref(ref):
  if ref != NULL:
    ref->header.refCount -= 1
    if ref->header.refCount == 0:
      recFree(ref)

recFree(ref):
  foreach loc in ref->header.descriptor->ptrs:
    deref(*(ref+loc))
  free(ref)
```



Must keep reference count in object headers

Algorithm

```
writeReference(loc, newVal):
  if newVal != NULL:
    newVal->header.refCount += 1
  deref(*loc)
  *loc = newVal

deref(ref):
  if ref != NULL:
    ref->header.refCount -= 1
    if ref->header.refCount == 0:
      recFree(ref)

recFree(ref):
  foreach loc in ref->header.descriptor->ptrs:
    deref(*(ref+loc))
  free(ref)
```

Must keep reference count in object headers

Writing a reference is at least five instructions

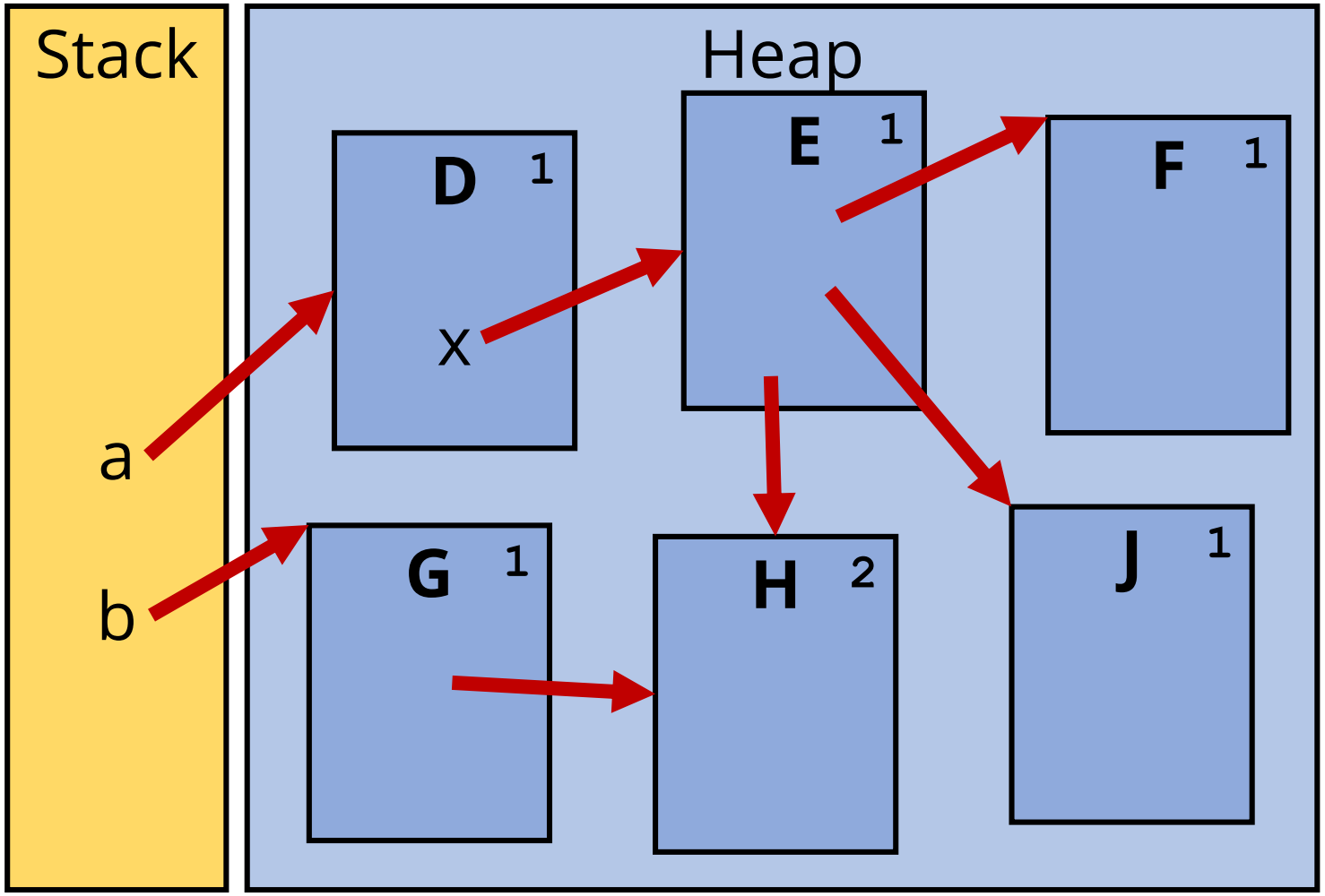
Algorithm

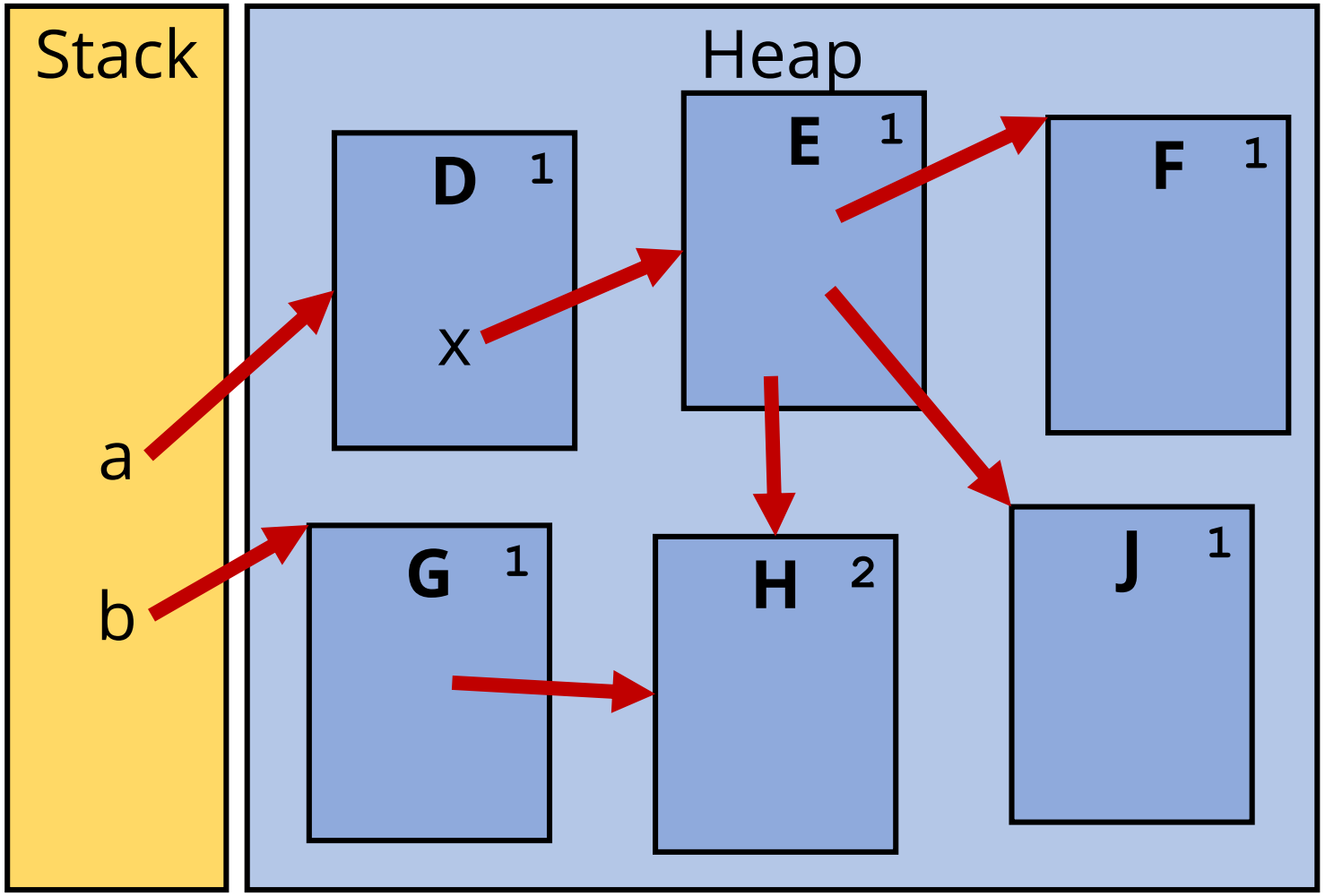
```
writeReference(loc, newVal):
  if newVal != NULL:
    newVal->header.refCount += 1
  deref(*loc)
  *loc = newVal

deref(ref):
  if ref != NULL:
    ref->header.refCount -= 1
    if ref->header.refCount == 0:
      recFree(ref)

recFree(ref):
  foreach loc in ref->header.descriptor->ptrs:
    deref(*(ref+loc))
  free(ref)
```

Must keep reference count in object headers

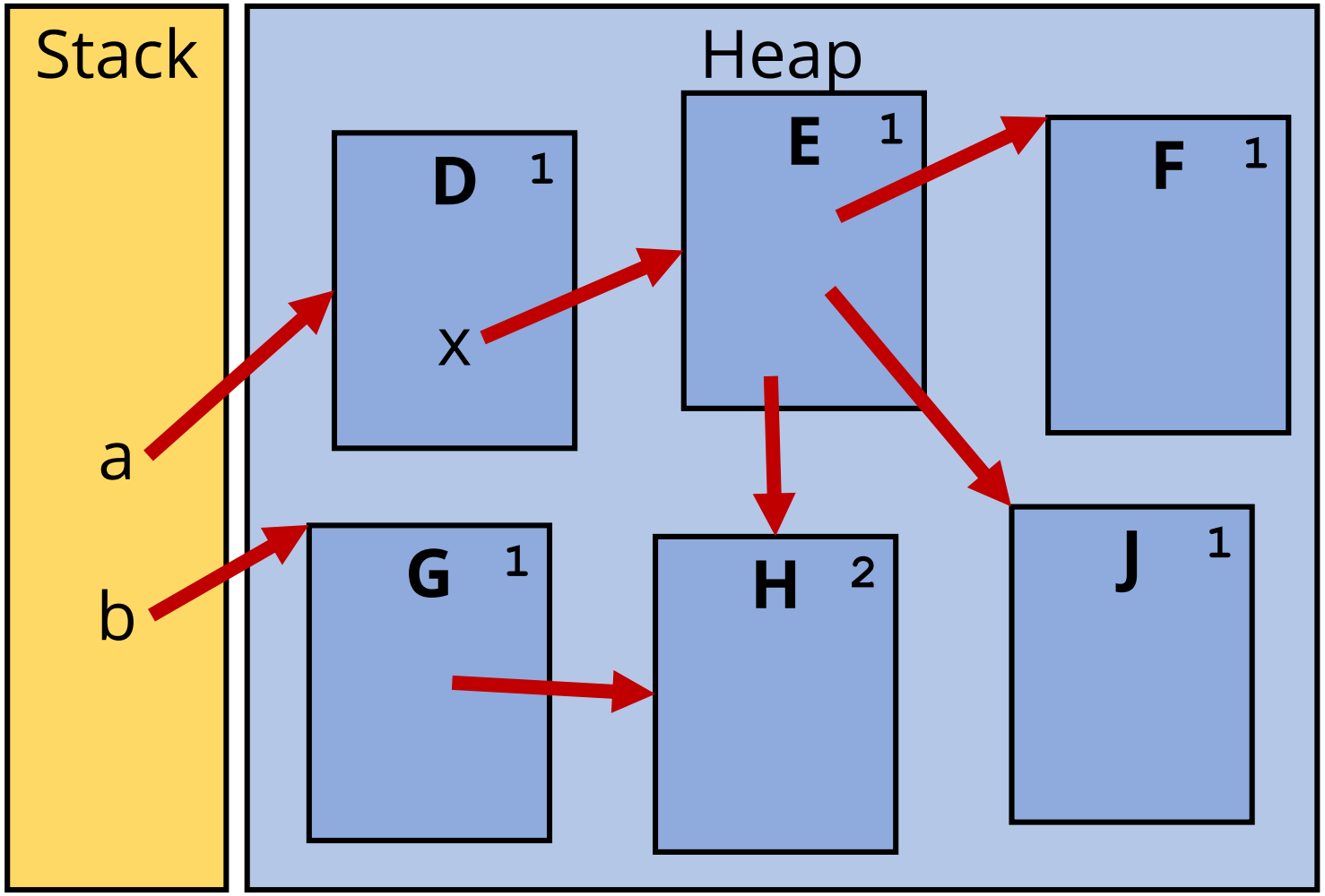
Writing a reference is at least five instructions

Simply free when count is zero

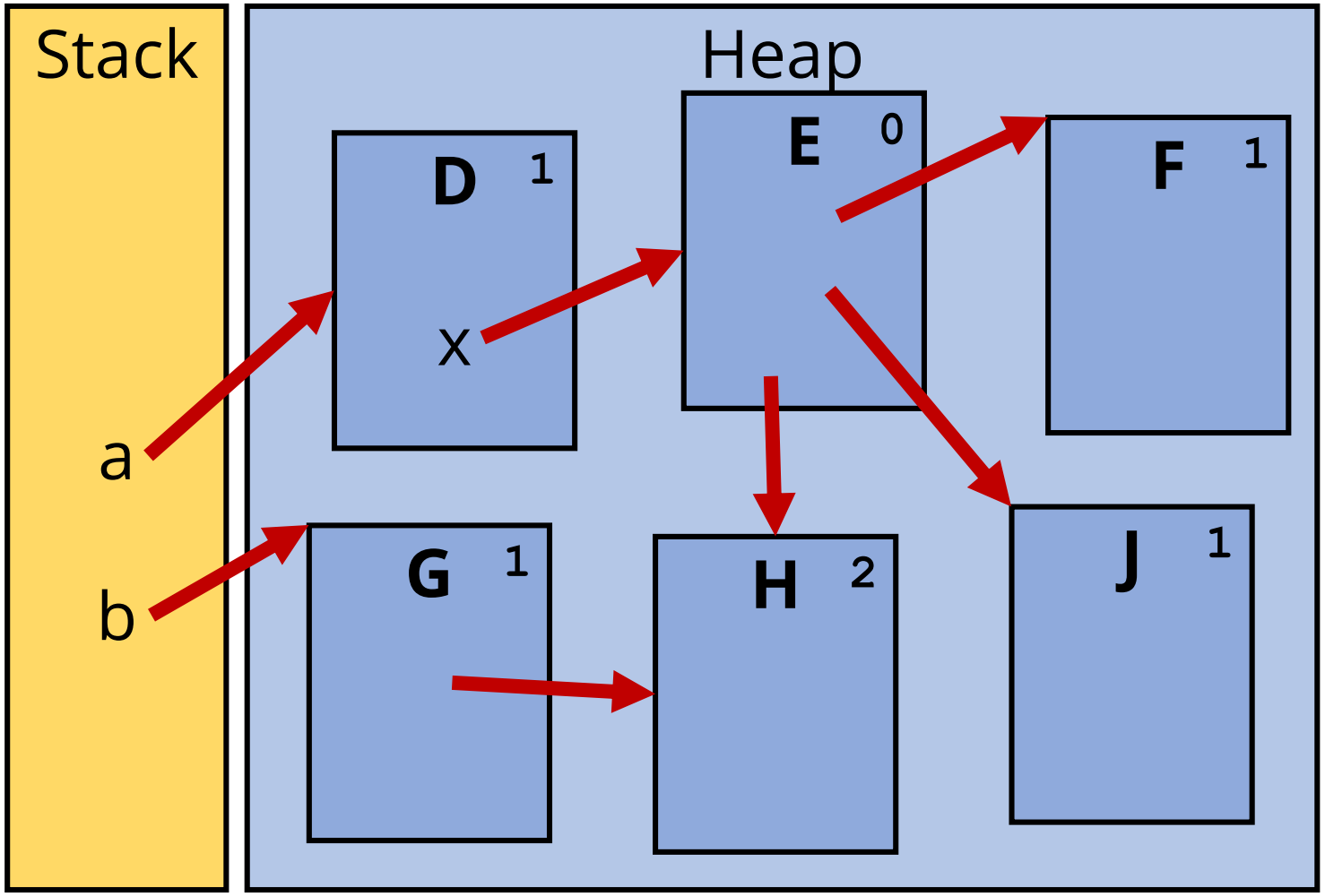

Stack
Heap
D 1
x
E 1
F 1
a
b
G 1
H 2
J 1



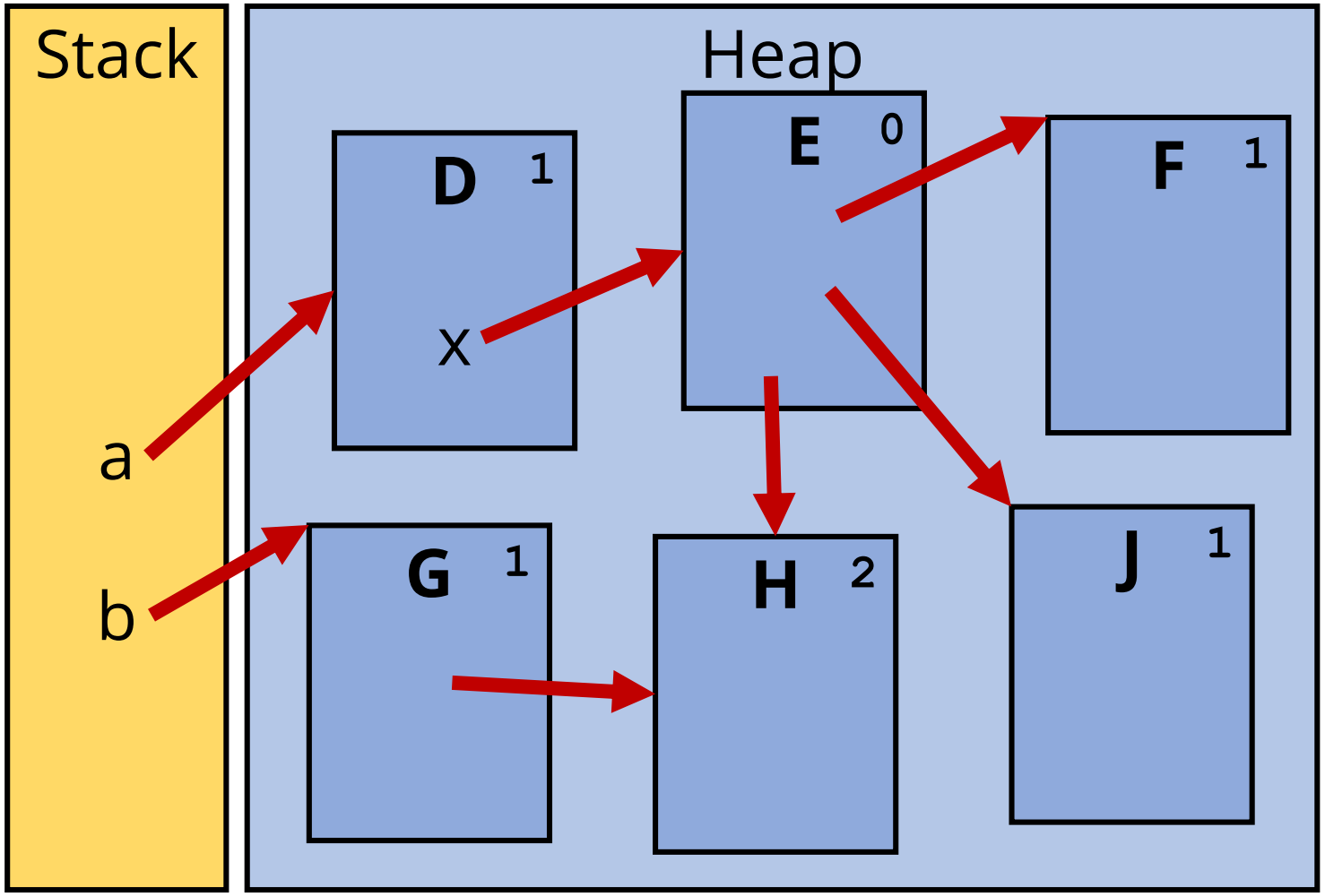

```
a->x = NULL
```



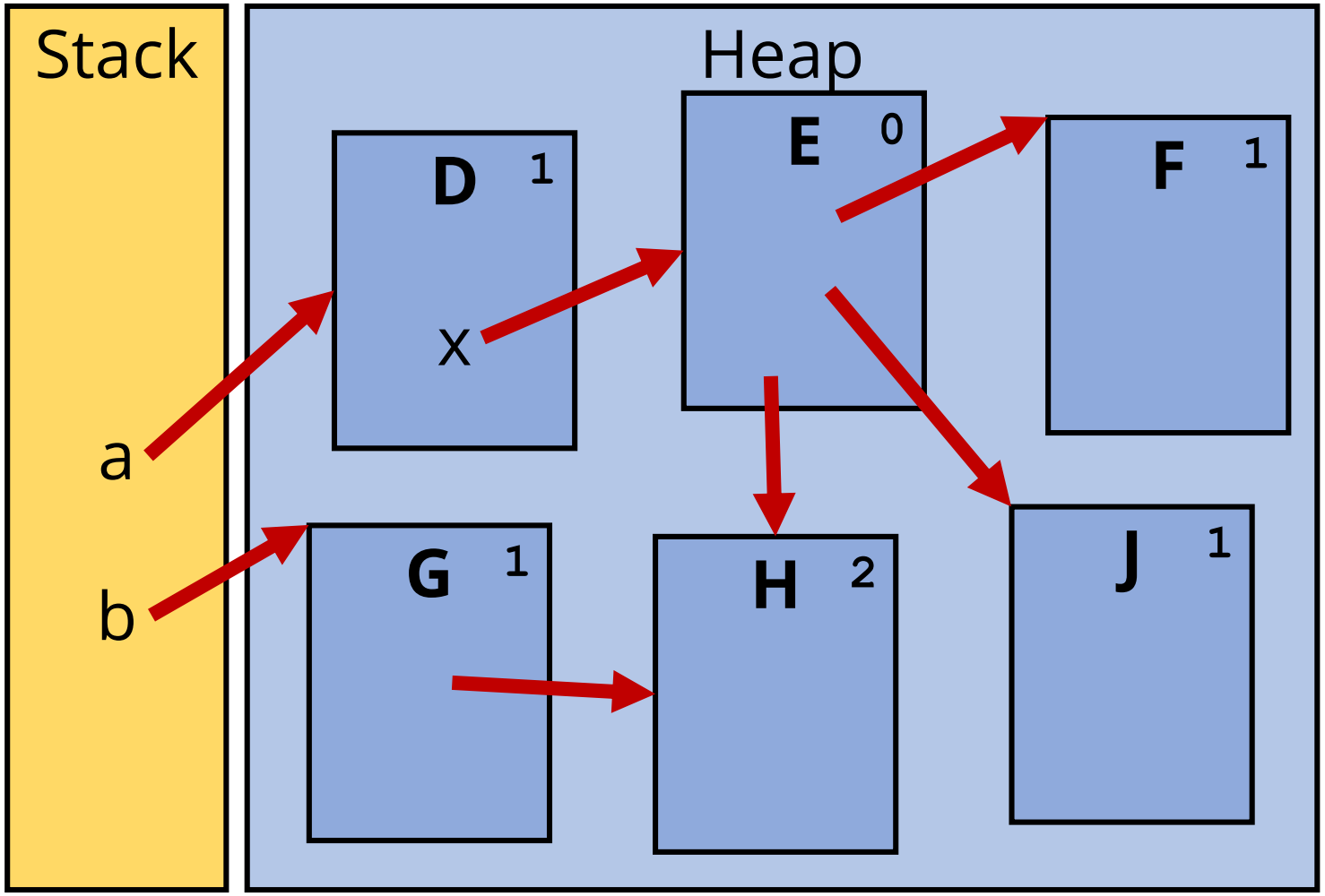

```
a->x = NULL
   deref(E)
```



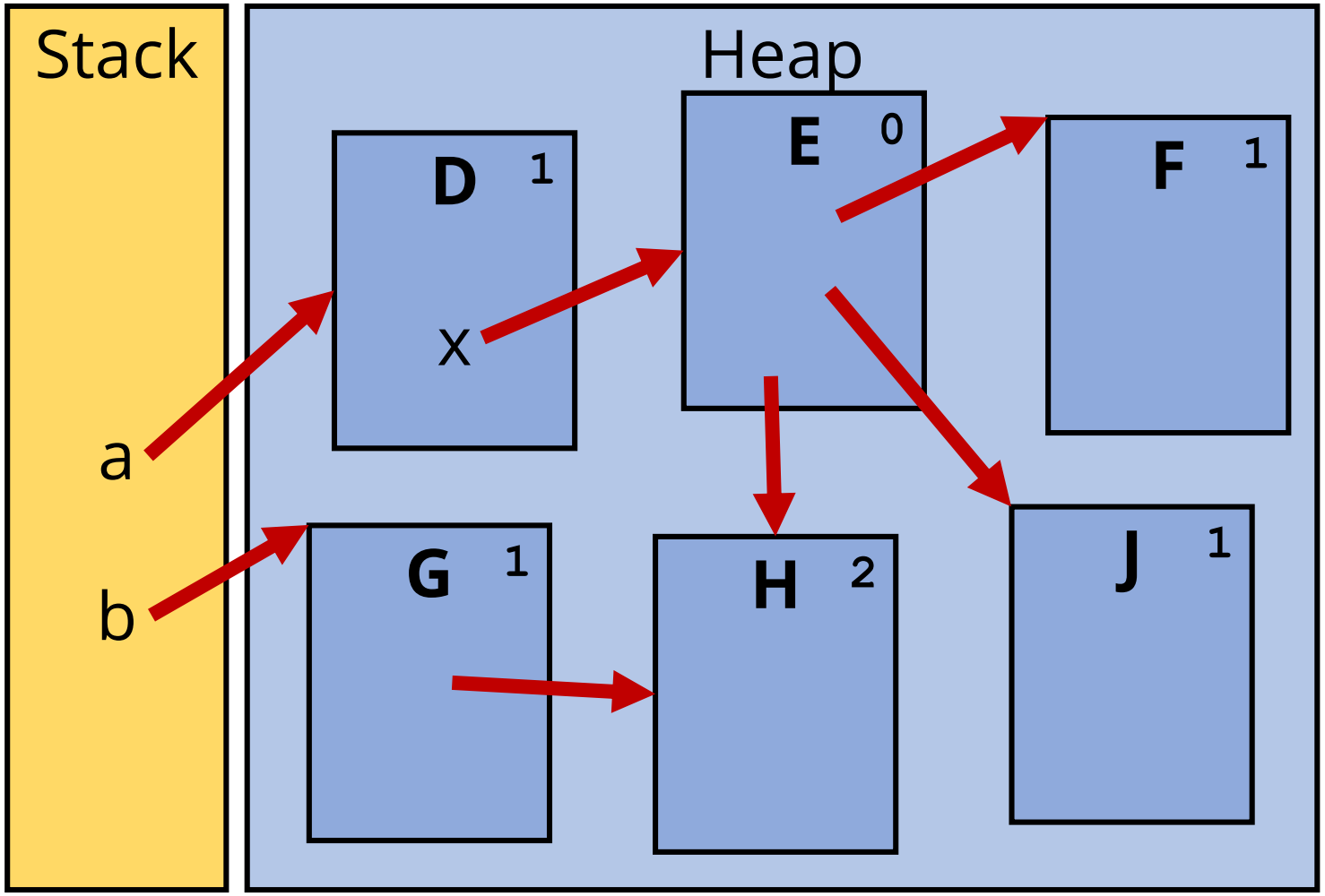

```
a->x = NULL
   deref(E)
```



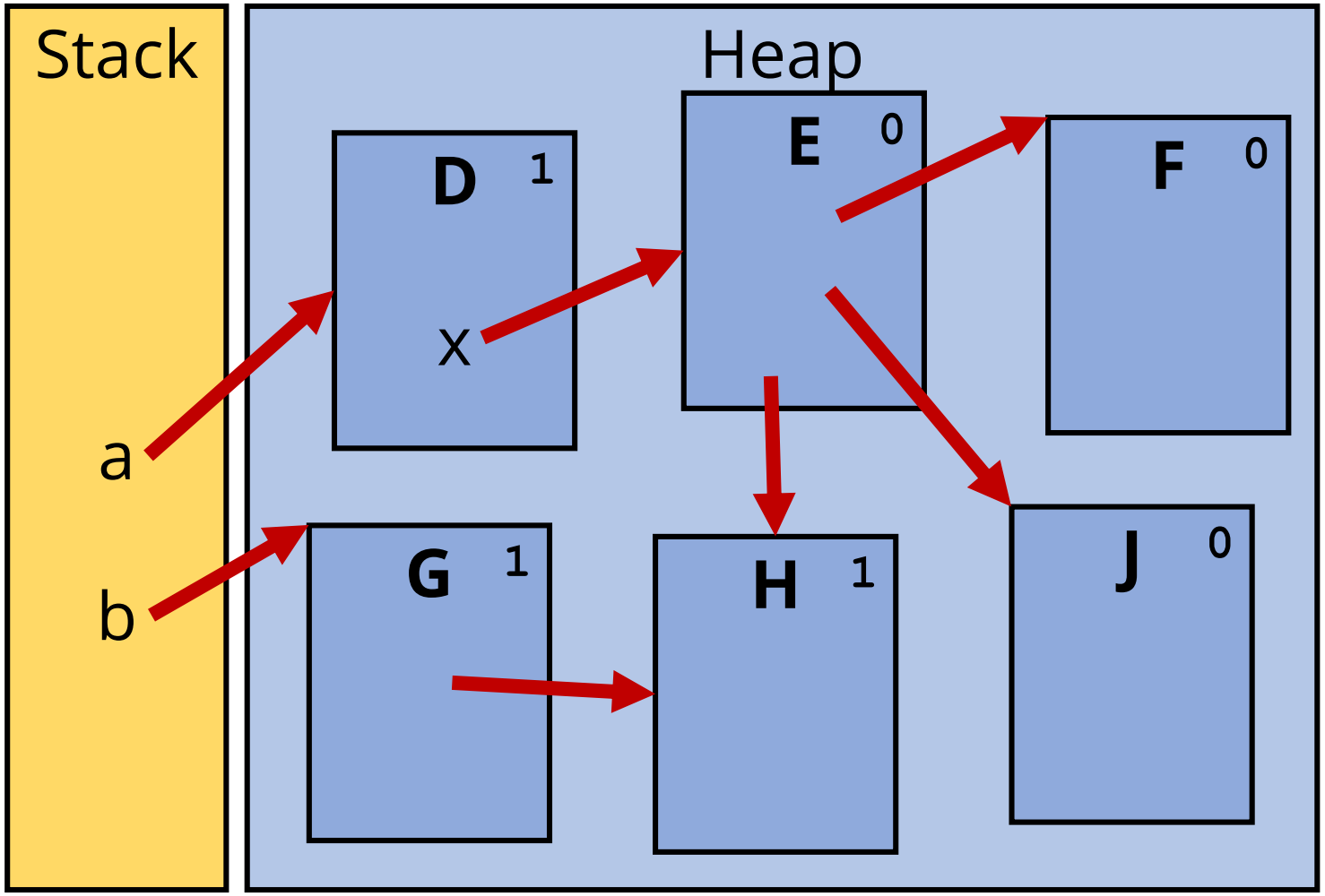

```
a->x = NULL
   deref(E)
      deref(F)
```



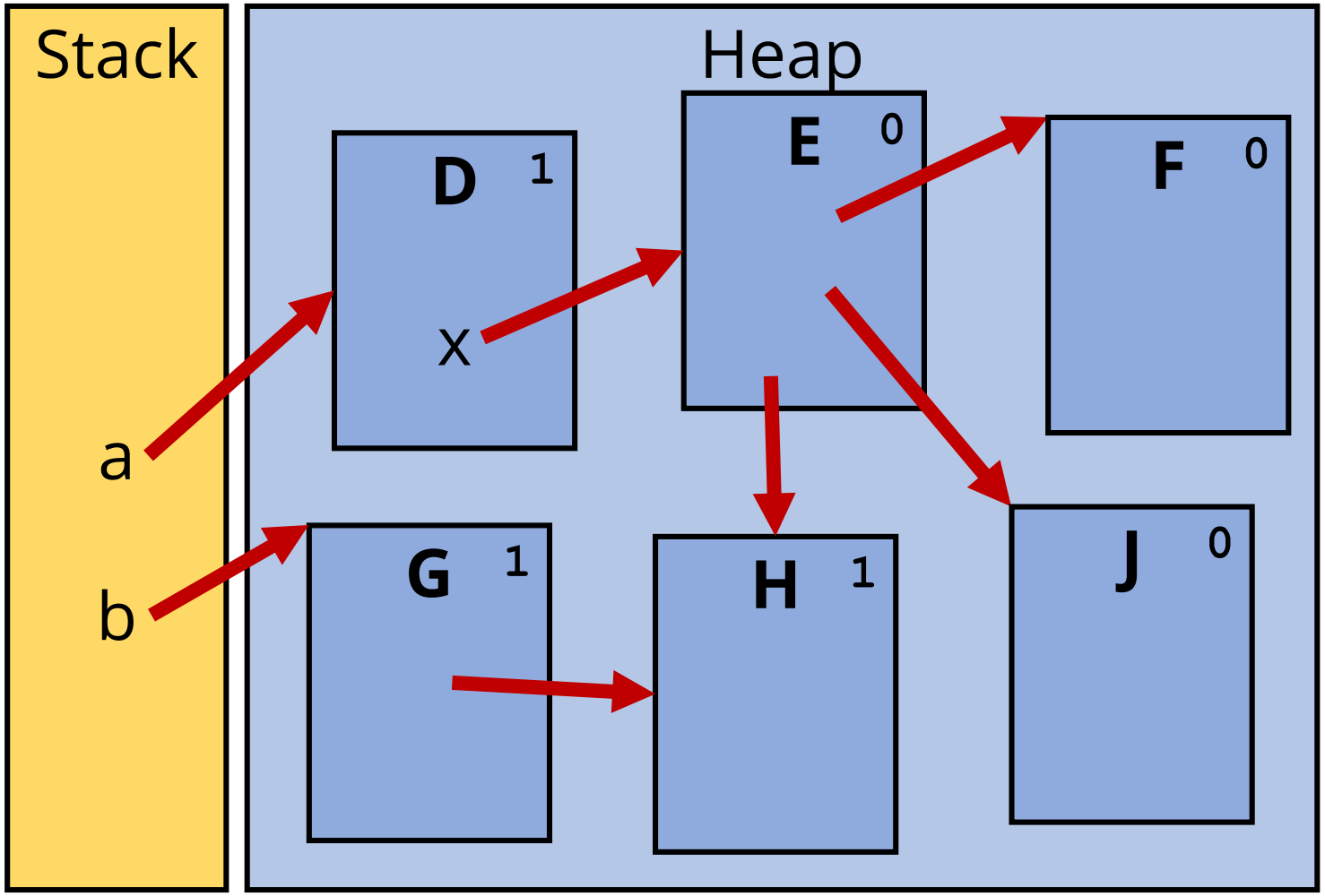

```
a->x = NULL
   deref(E)
      deref(F)

      deref(J)
```



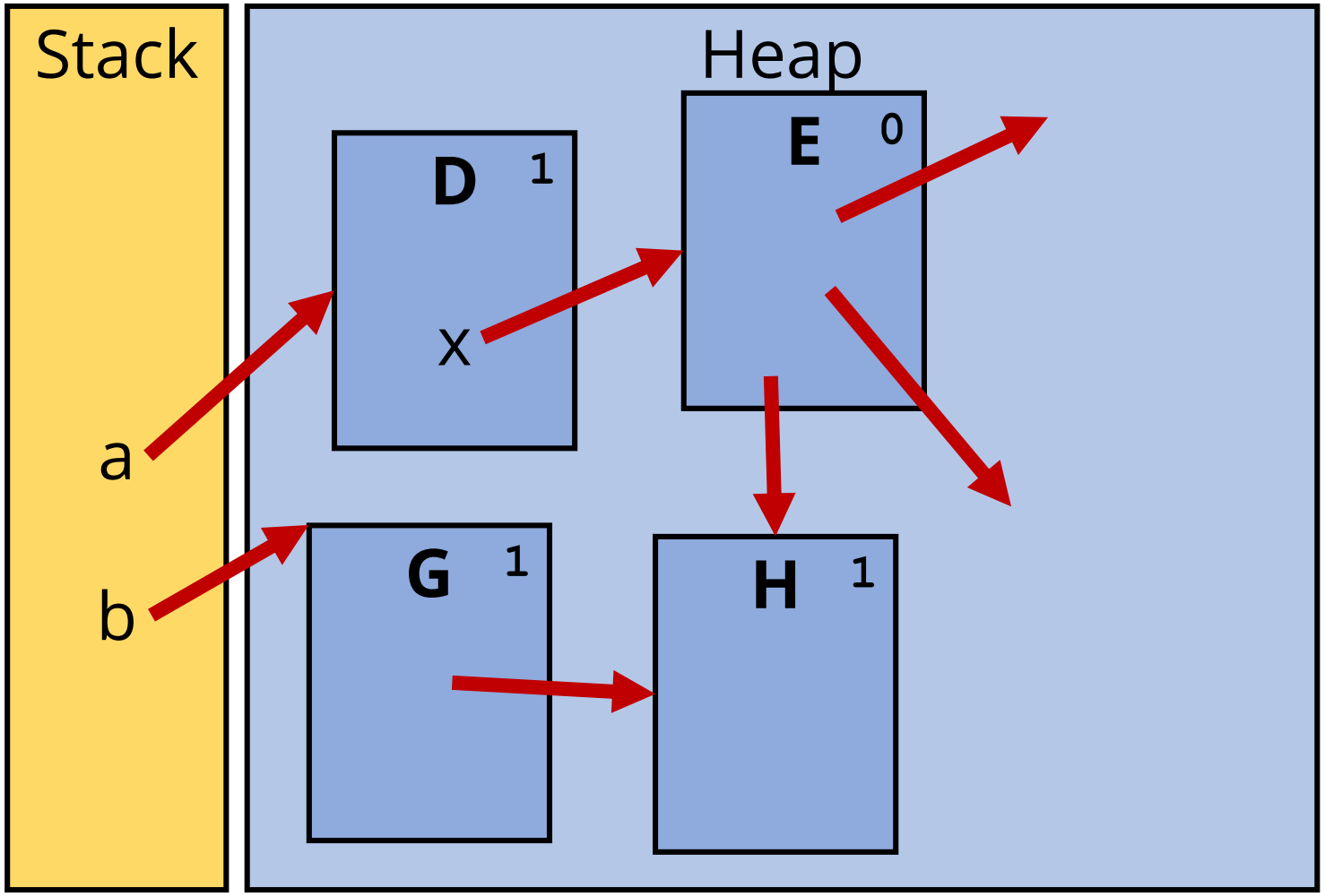

```
a->x = NULL
   deref(E)
      deref(F)

      deref(J)

      deref(H)
```



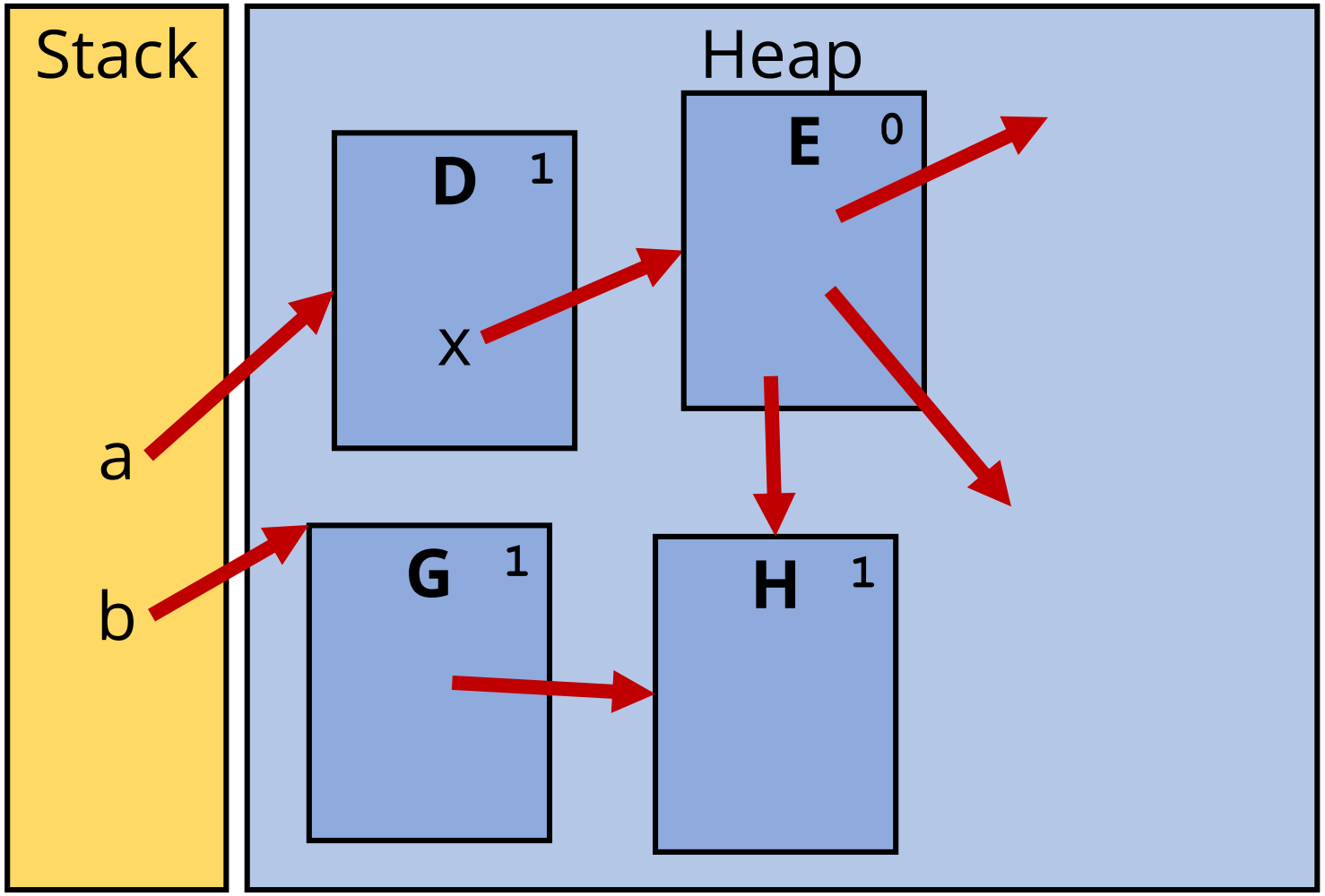

```
a->x = NULL
   deref(E)
      deref(F)

      deref(J)

      deref(H)
```



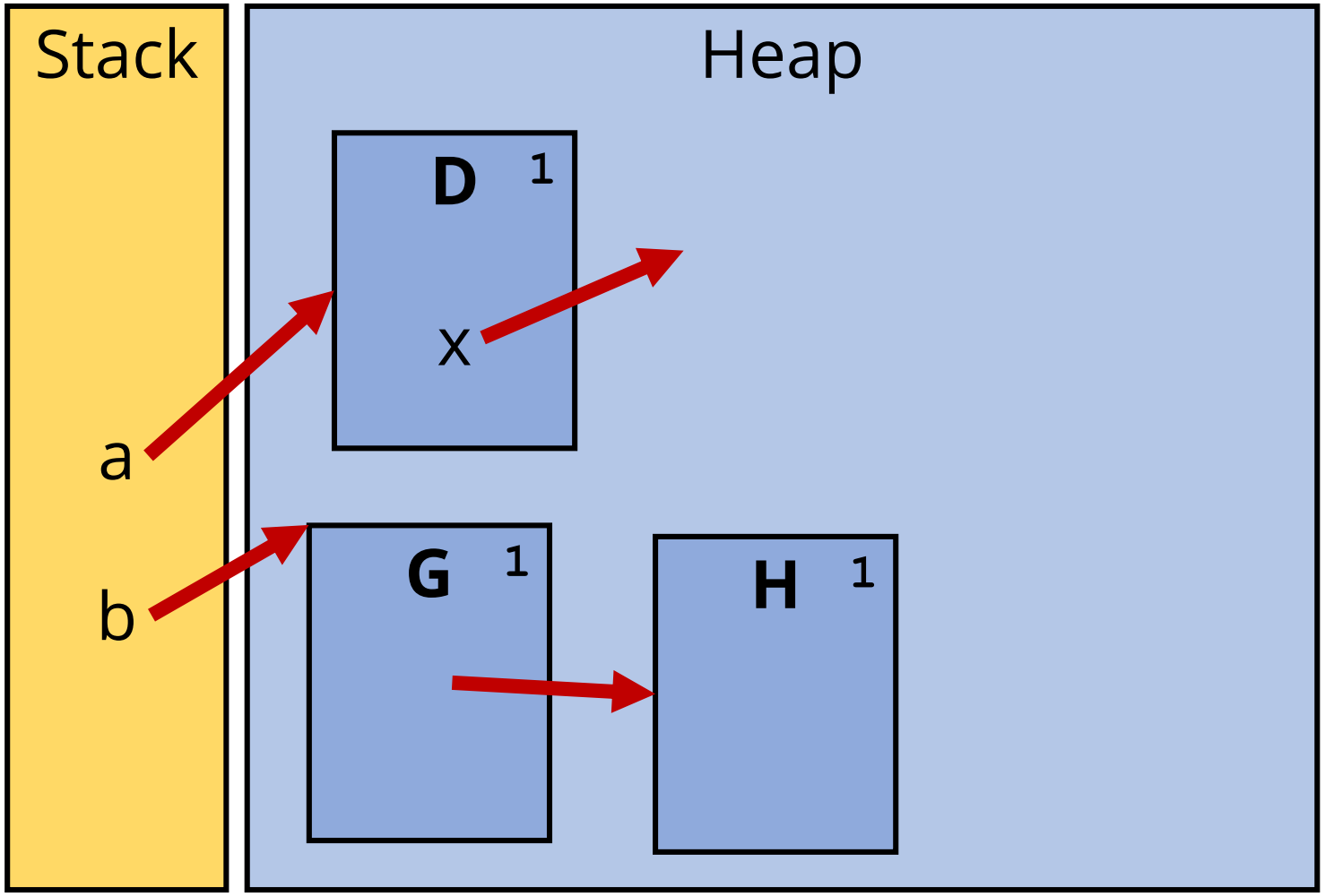

```
a->x = NULL
   deref(E)
      deref(F)
         free(F)
      deref(J)
         free(J)
      deref(H)
```



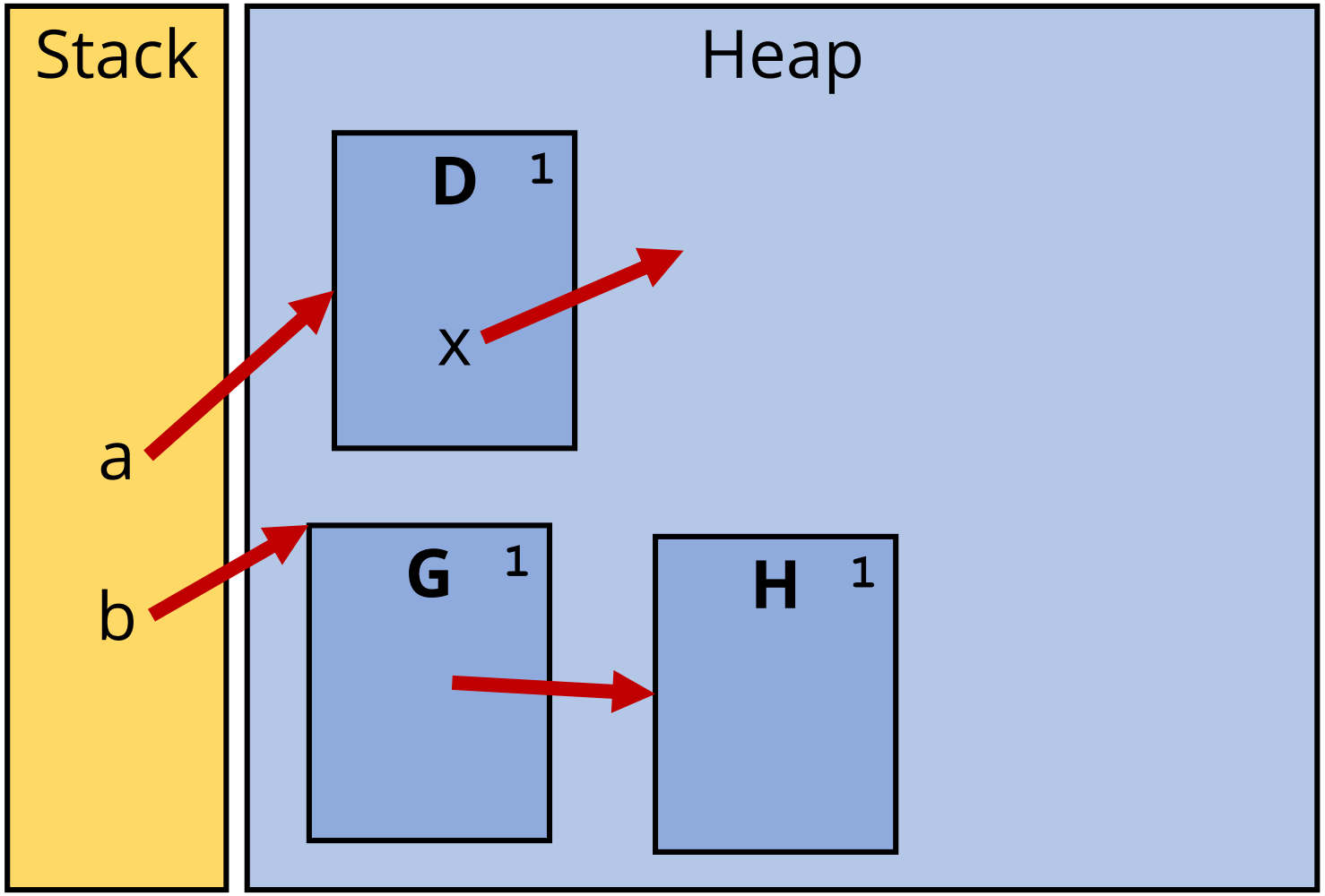

```
a->x = NULL
   deref(E)
      deref(F)
         free(F)
      deref(J)
         free(J)
      deref(H)
```



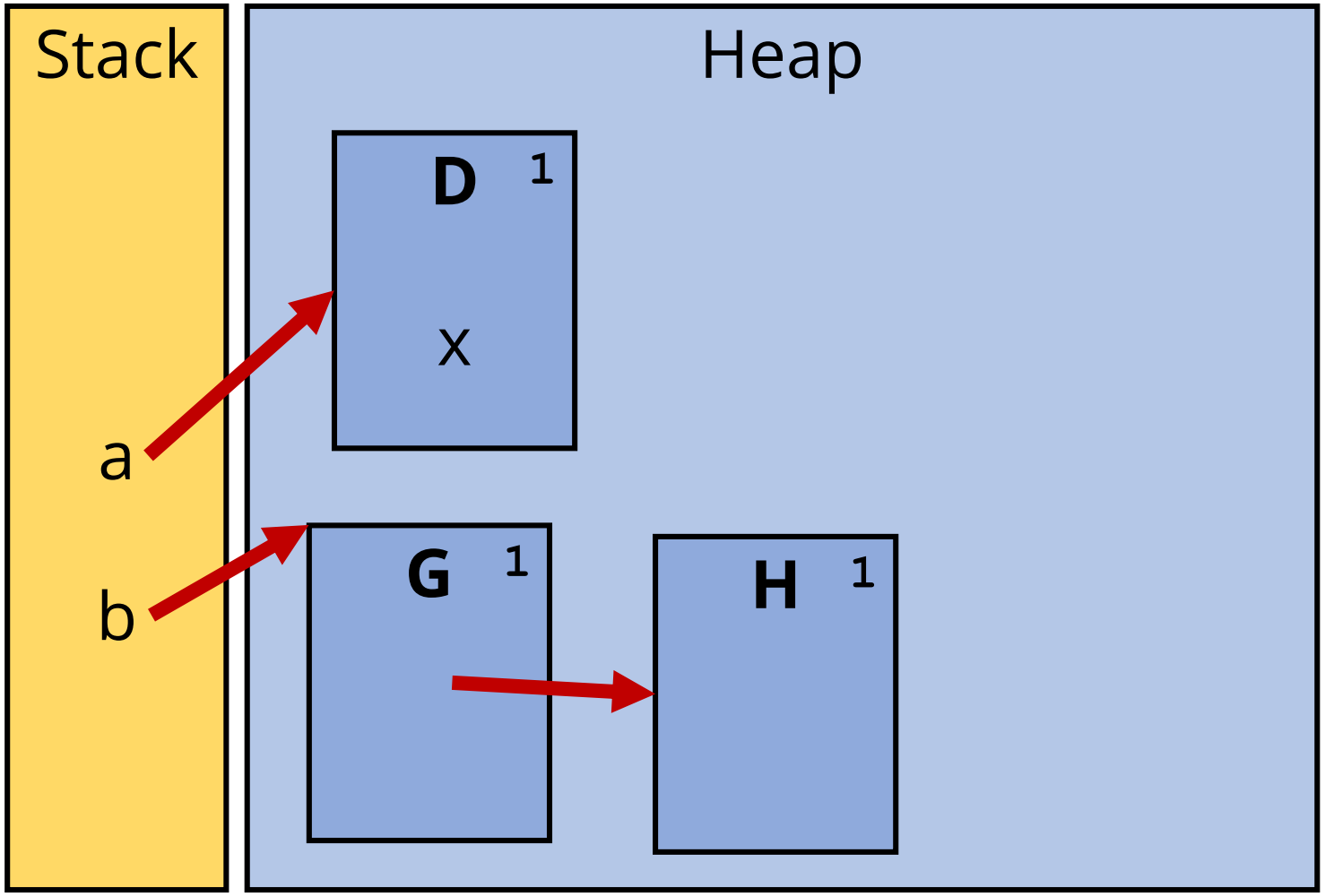

```
a->x = NULL
   deref(E)
      free(E)
```




```
a->x = NULL
   deref(E)
      free(E)
```




```
a->x = NULL
```




```
a->x = NULL
```

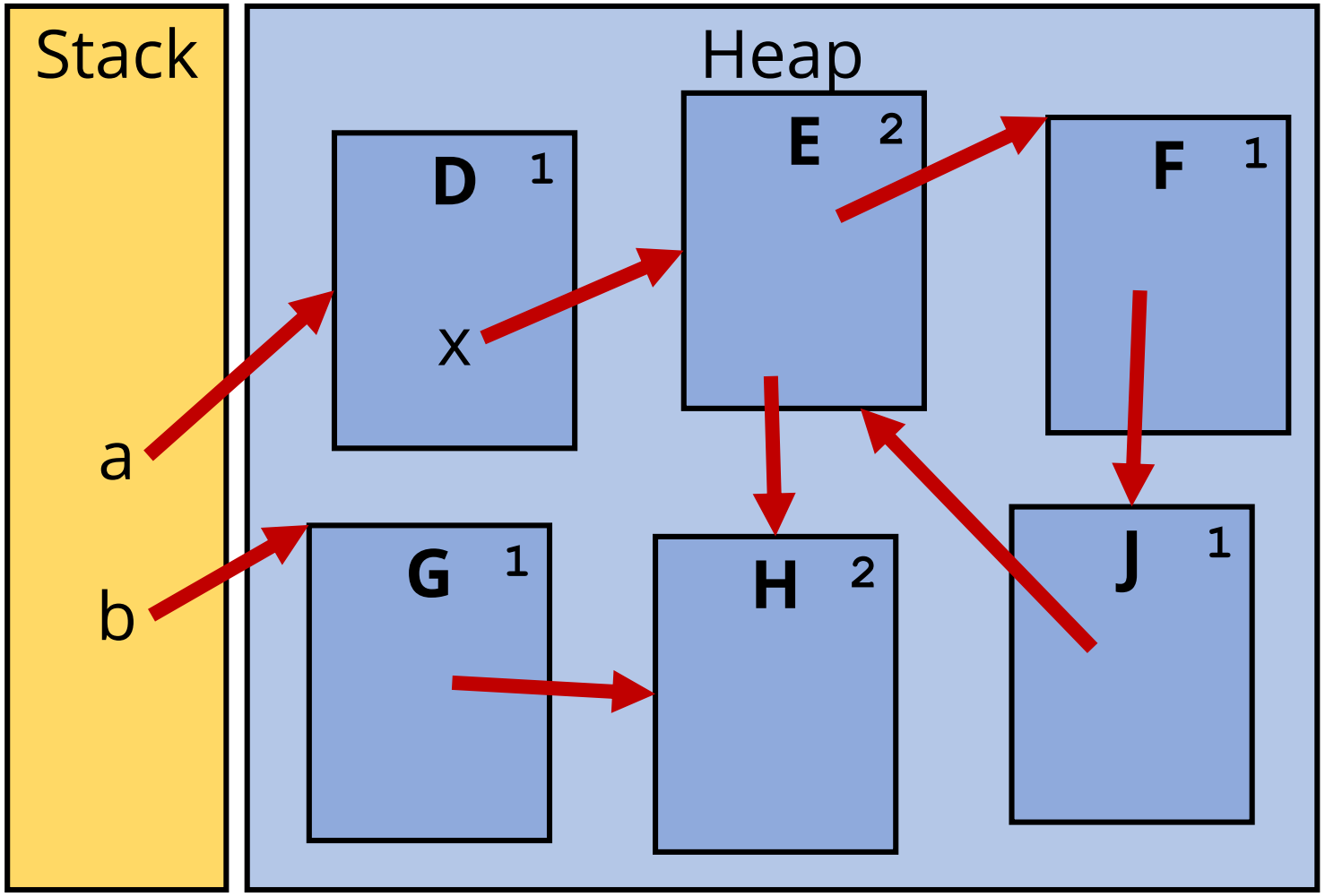
Advantages

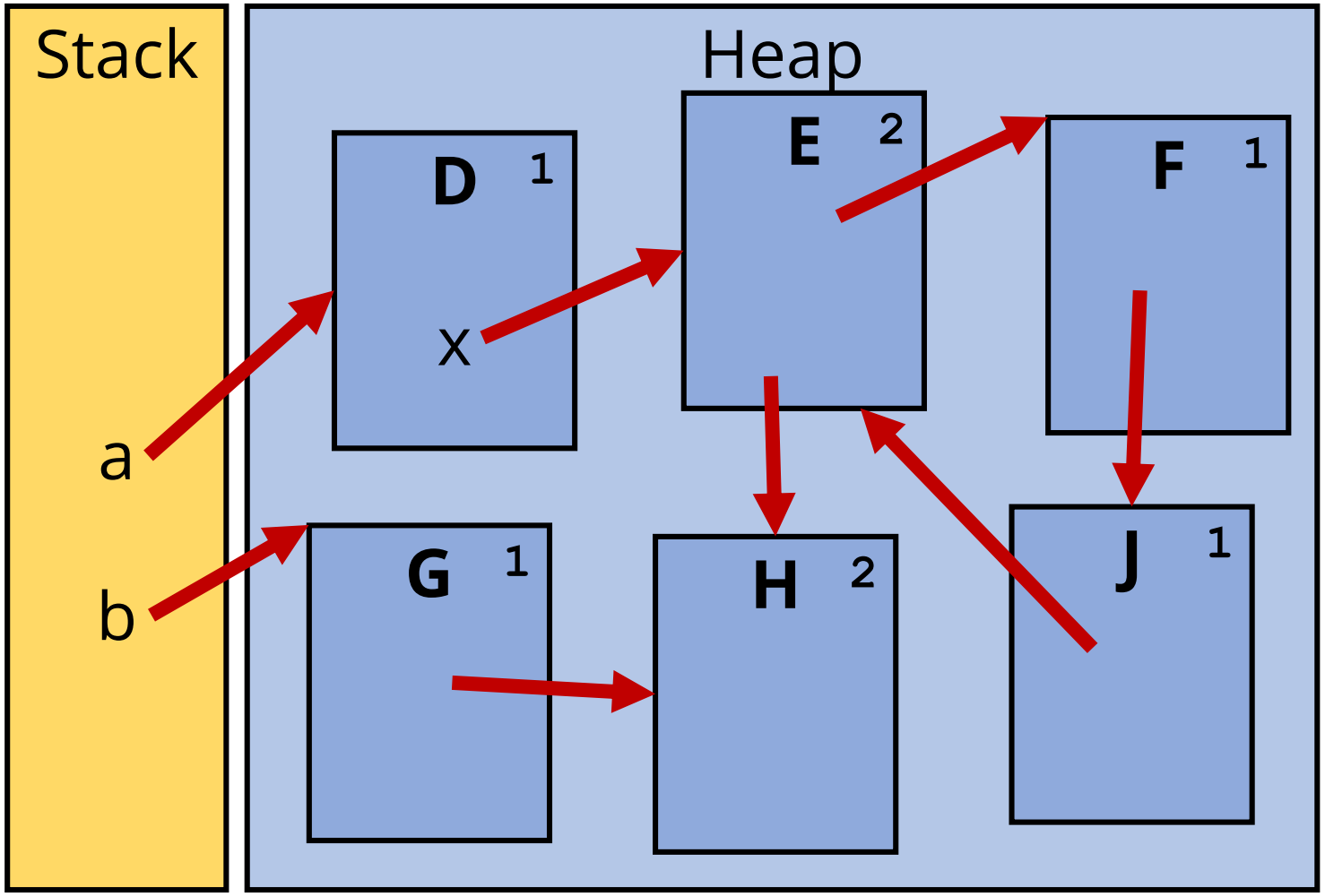
- Memory free’d immediately
- L ≈ H
- No stop-the-world GC pause[1]
- Intuitive connection to reachability

[1] Claim not actually true, as we’ll see soon

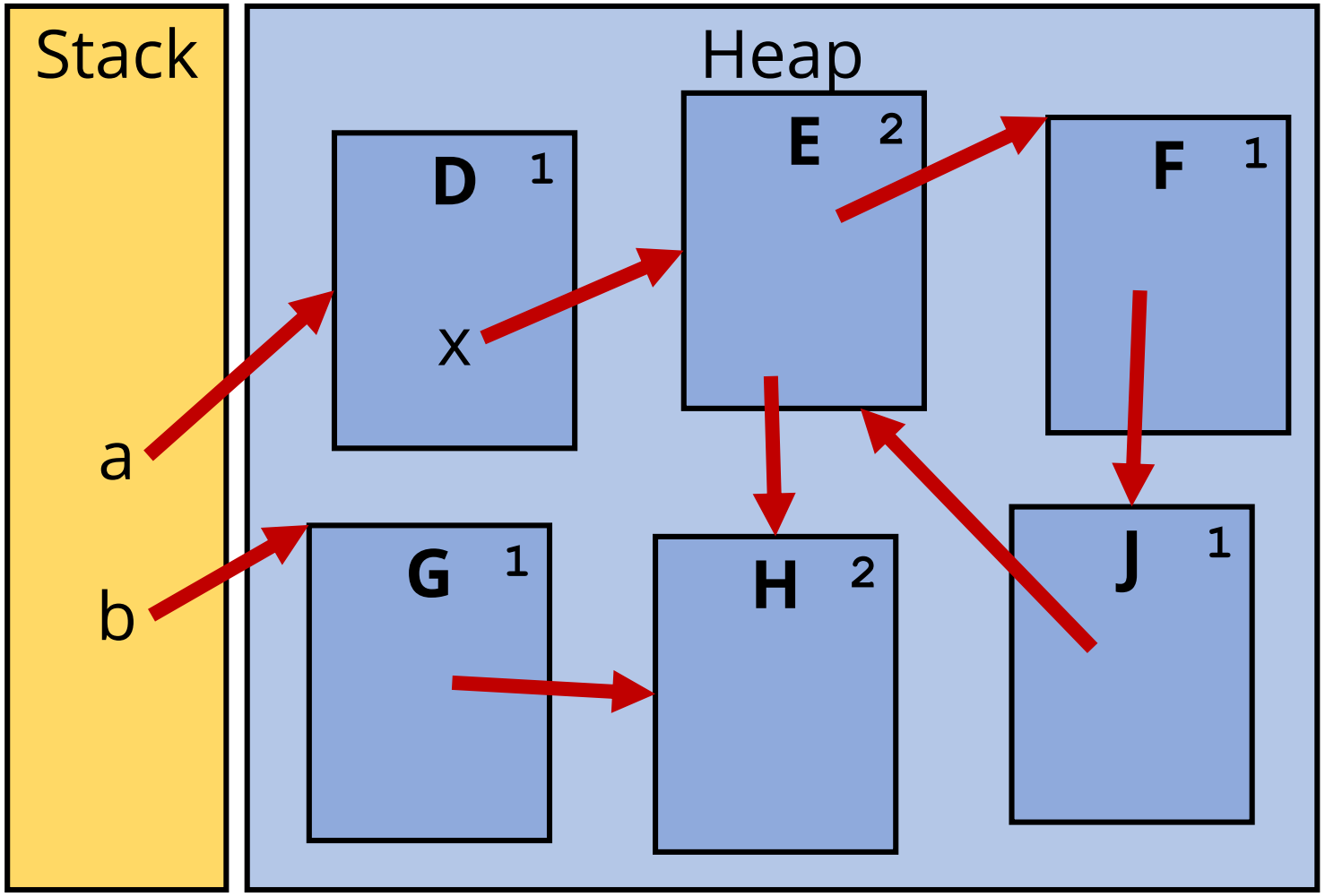
Disadvantages

- More than can fit on a slide
- Need *word* in header for ref count
- Must write to object to read from object
- Threads: Must lock reference count
- Recursive frees → unpredictable pauses

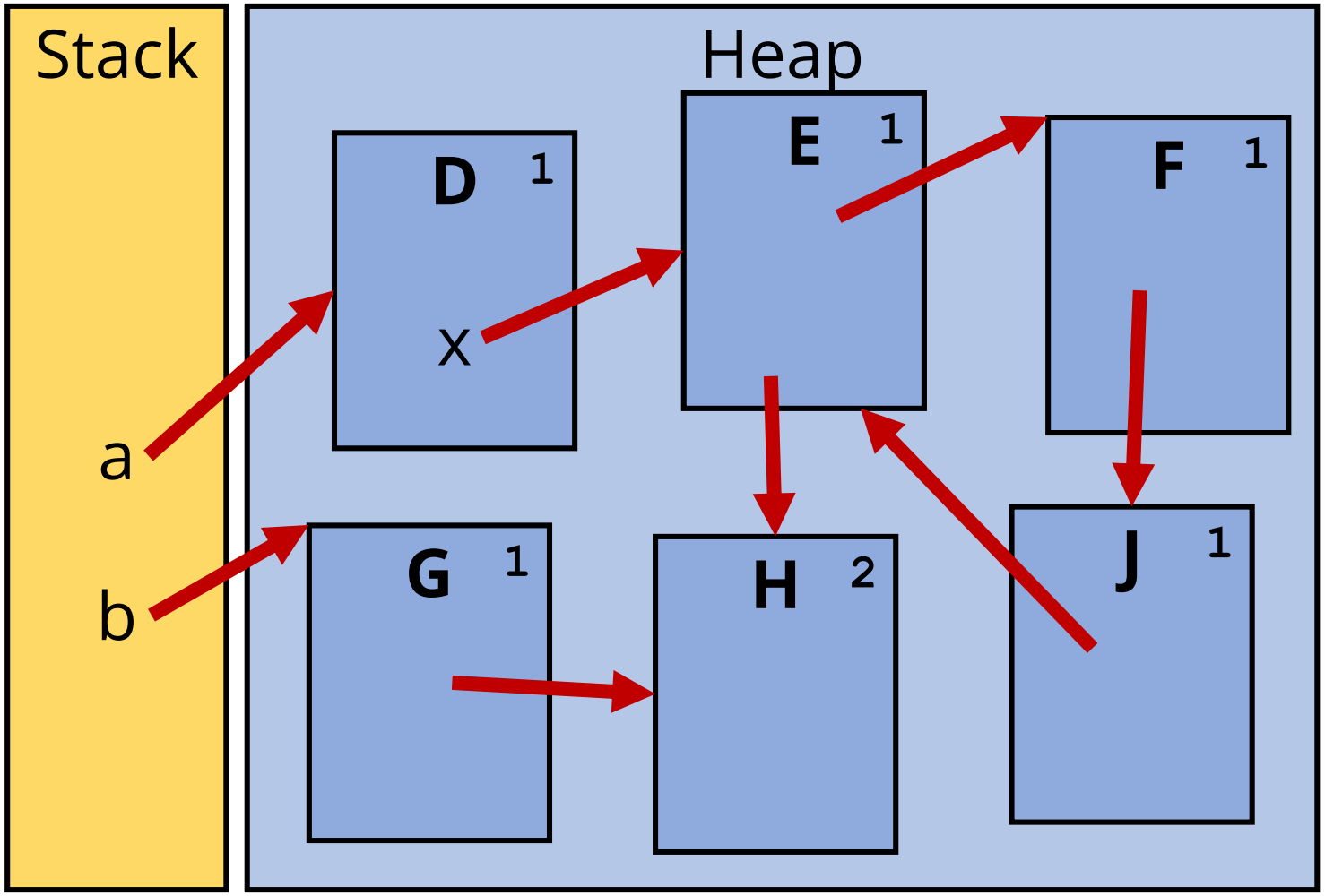

Stack
Heap
D 1
x
E 2
F 1
a
b
G 1
H 2
J 1



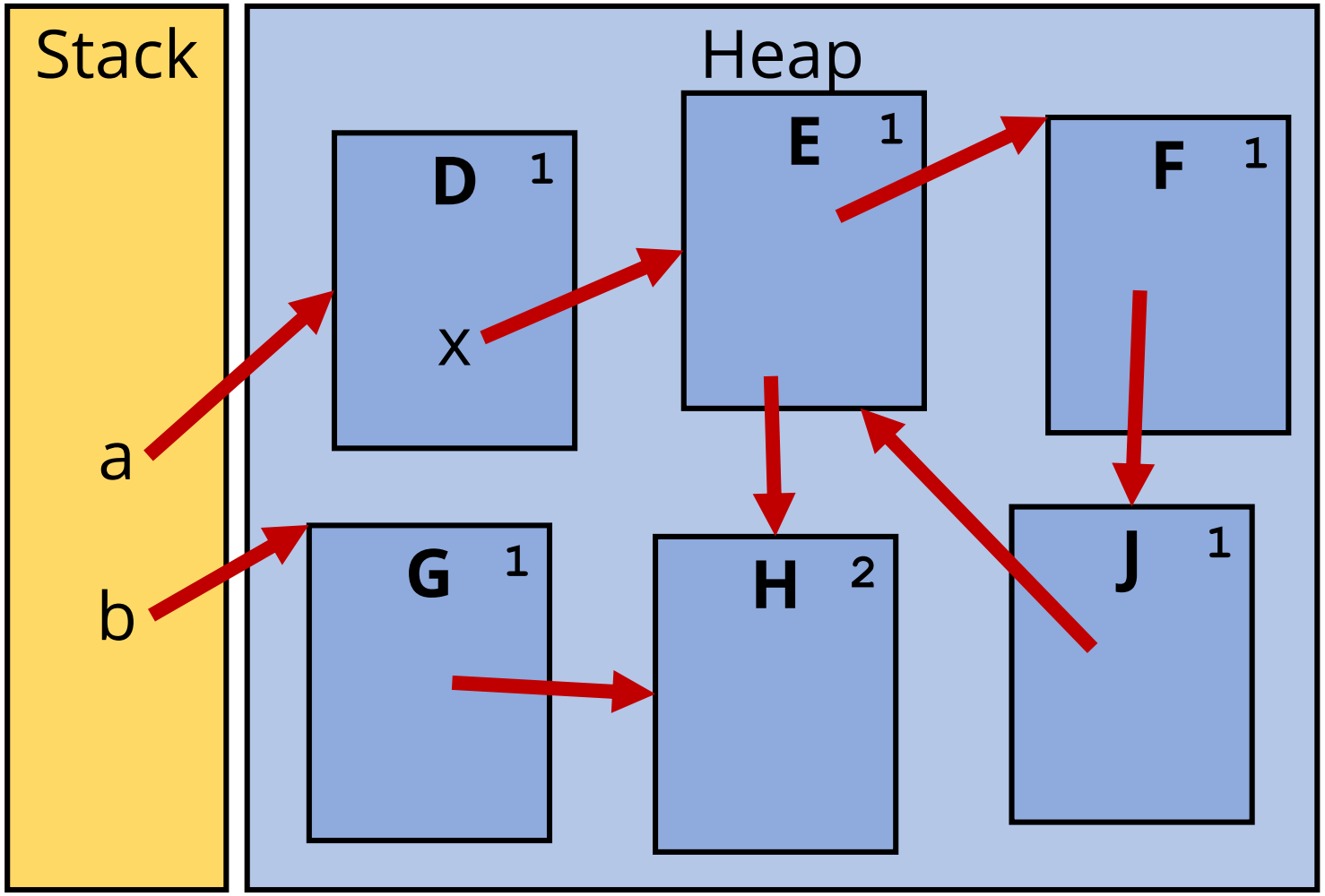

```
a->x = NULL
```



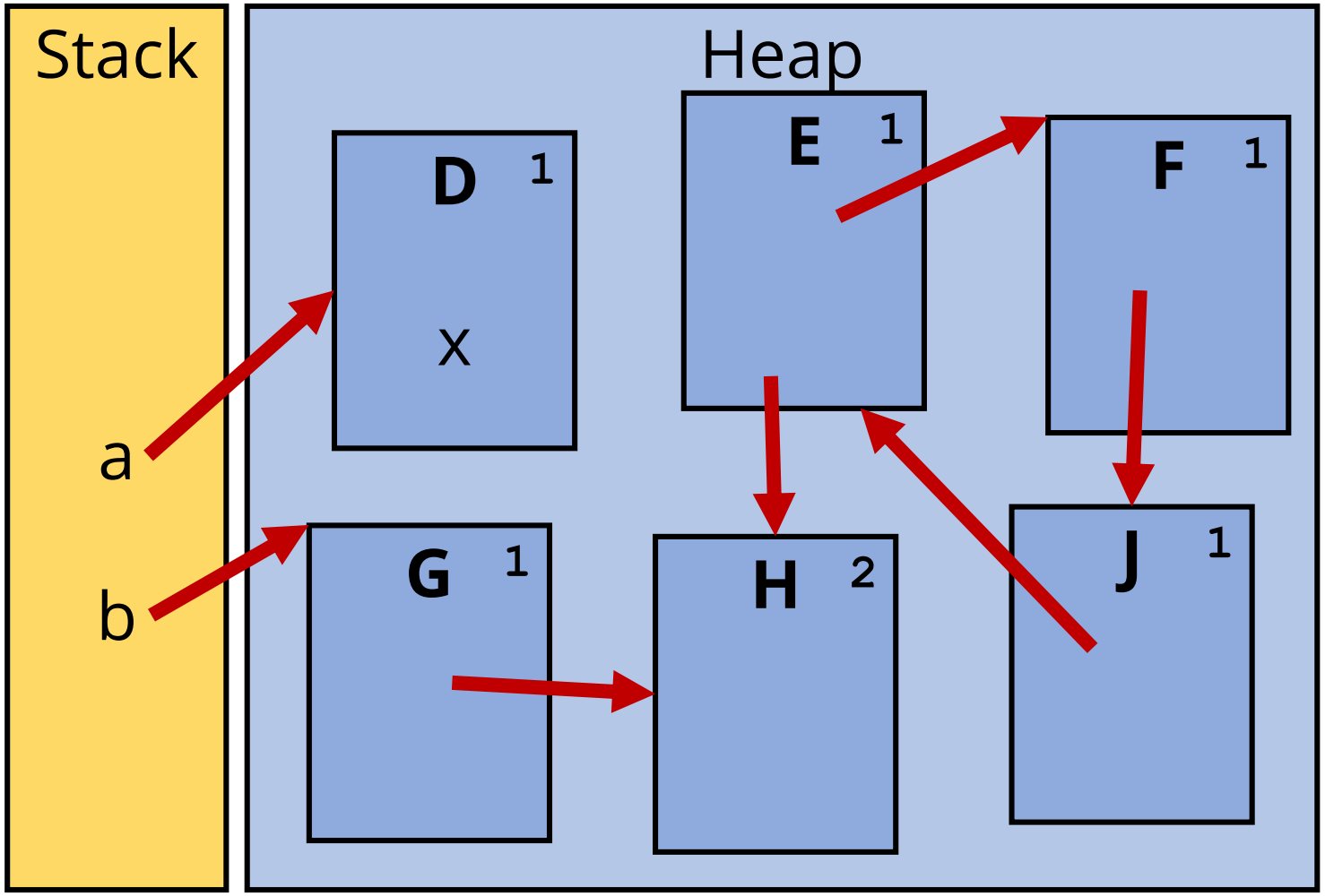

```
a->x = NULL
   deref(E)
```



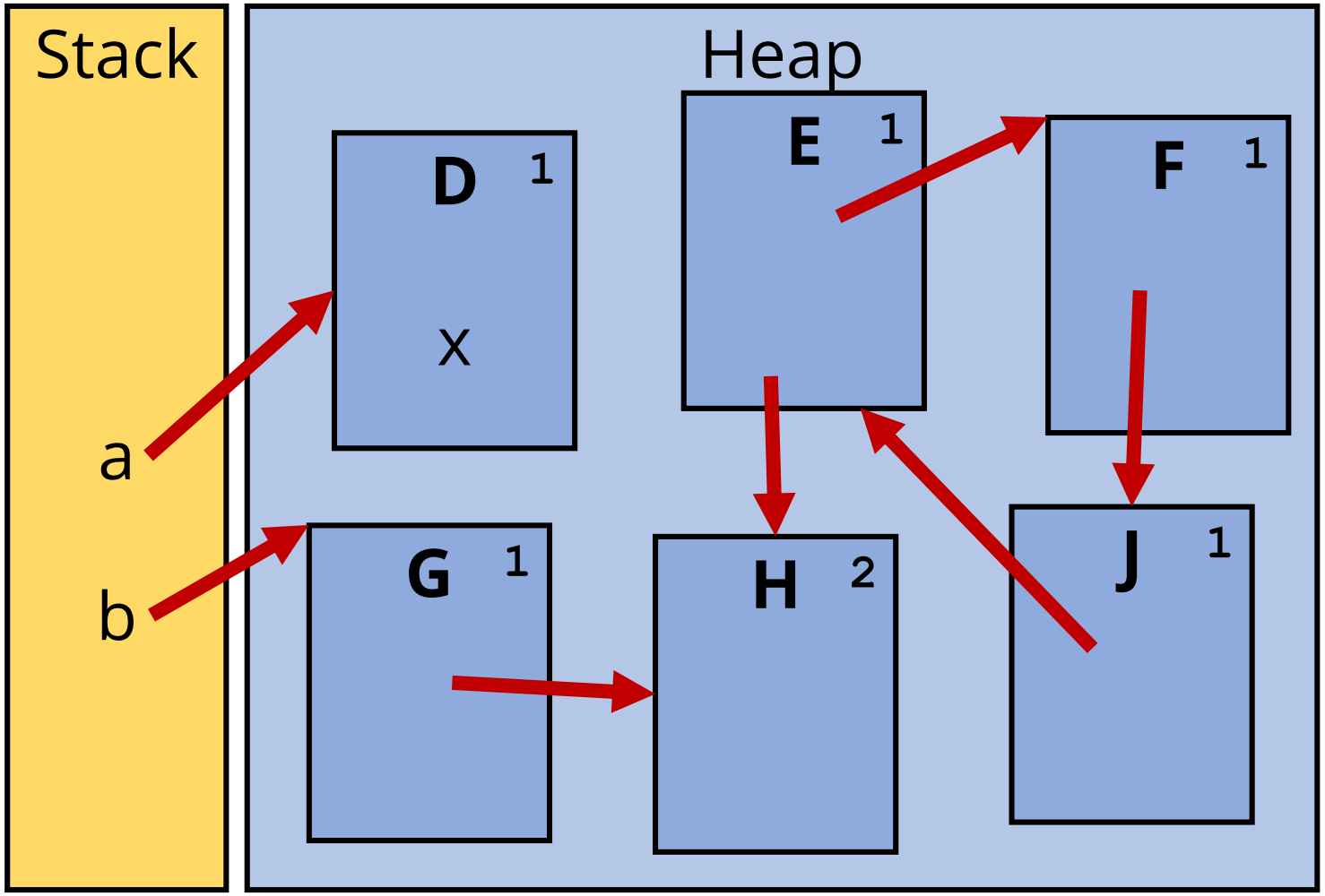

```
a->x = NULL
   deref(E)
```

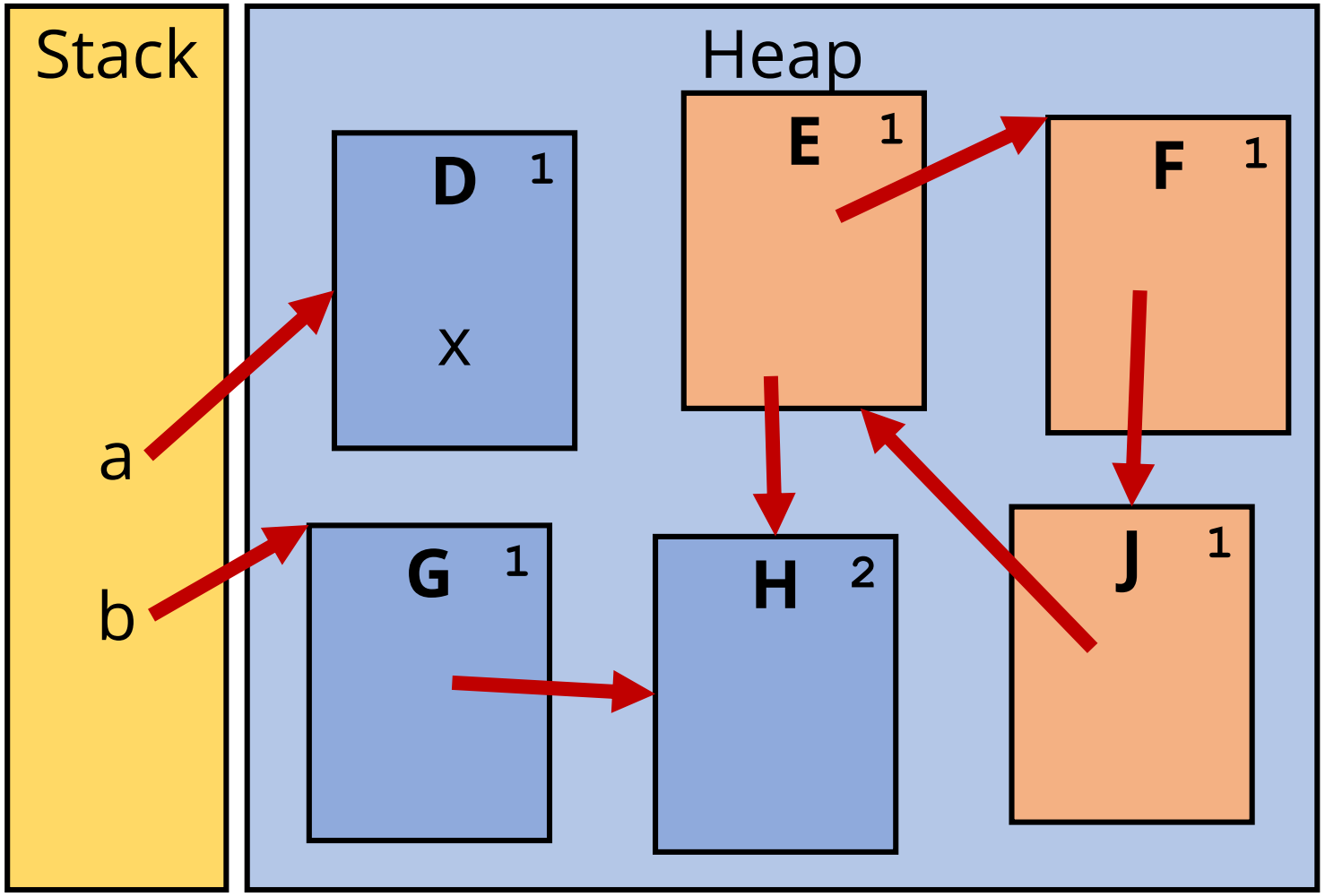



```
a->x = NULL
```




```
a->x = NULL
```


Stack
Heap
D 1
x
E 1
F 1
G 1
H 2
J 1
a
b


Stack
Heap
a
b
D 1
x
E 1
F 1
G 1
H 2
J 1

Cycles

- Cyclic data structures cannot be reclaimed
- Type system can guarantee non-cyclicness…
- but useful languages allow cycles

Breather

- Reference counting measures reachability through reference count
- High price on mutator: Write barriers
- Cannot collect cycles
- Can allow (L ≈ H)

Fixing reference counting

- Reference counting can be made better
- With fixes, *if* (L ≈ H) is crucial, it may be a good option
- *All* fixes involve GC pauses

Deferred reference counting

- Heap references change infrequently, stack references change frequently
- Intuition: Defer counting stack references
- If we don’t count some references, rc=0 doesn’t mean free!
- Store zero-count objects in a table for deferred processing

```
writeHeapReference(loc, newVal):
  incref(newVal)
  deref(*loc)
  *loc = newVal

incref(ref):
  if ref != NULL:
    newVal->header.refCount += 1

deref(ref):
  if ref != NULL:
    ref->header.refCount -= 1
    if ref->header.refCount==0:
      zct.push(ref)
```

```
collect():
  foreach loc in roots:
    incref(*loc)
  sweep()
  foreach loc in roots:
    deref(*loc)

sweep():
  while !zct.empty():
    ref := zct.pop()
    if ref->header.refCount==0:
      foreach loc in (ref ptrs):
        deref(*(ref+loc))
      free(ref)
```

When to collect

- Typically same as GC: When allocation cannot provide object without new pool
- “Collection” approx. O(D), max O(H)
- In practice, zct mostly correct

Improvement

- Cost of updating references slashed ~80%
- Threads still a problem
- Traded write time for pause time
- If D<L, pause time better than semispace

Coalesced reference counting

- Sequences of reference changes cancel each other out:
  - a->x = b; a->x = c; a->x = d;
  - b's, c's counts incremented then decremented
- So, defer counting fields too!

Coalesced reference counting

- Log which objects are changed
- Each thread gets its own log → avoid thread contention
- Use log to do only first decrement, last increment

```
writeHeapReference(ref, loc,
 newVal):
  if !dirty(ref):
    log(ref)
  *loc := newVal

log(ref):
  myLog.add([dup(ref), ref])
  setDirty(ref)
```

```
collect():
  collectLogs()
  foreach loc in roots:
    incref(*loc)
  sweep()
  foreach loc in roots:
    deref(*loc)

collectLogs():
  foreach entry in (all logs):
    oldObj := entry.first
    newObj := entry.second

    foreach loc in (obj ptrs):
      incref(*(newObj+loc))
      deref(*(oldObj+loc))

  (clear all logs)
  (mark all objects clean)
```

Improvement

- Need space in header for dirtiness
- Need logs for each thread
- Duplicate objects when dirtied
- Still need to sync on setDirty
- Most thread contention eliminated

Fixing cycles

- Usual solution: Always have a backup mark-and-sweep collector
  - Doesn’t that defeat the purpose?
  - Far less *frequent* collections
- Further solutions: Partial collection, trial deletion

Partial collection

- Dead cycles only occur after losing root reference
- Only objects which have had ref count reduced may be cycles
- In mark phase, ignore objects which have not been reduced since last GC

Trial deletion

- Alternative to GC
- Reduce ref count of potential cycle object to 0, recursively deref children
- If ref count goes *below* 0, cycle detected!
- Otherwise, restore ref counts
- Needs some form of deferral

Bringing it together

- All forms have a high mutator price
- Worst case pause O(D) regardless
- Secondary GC necessary
- One winning case: L ≈ H