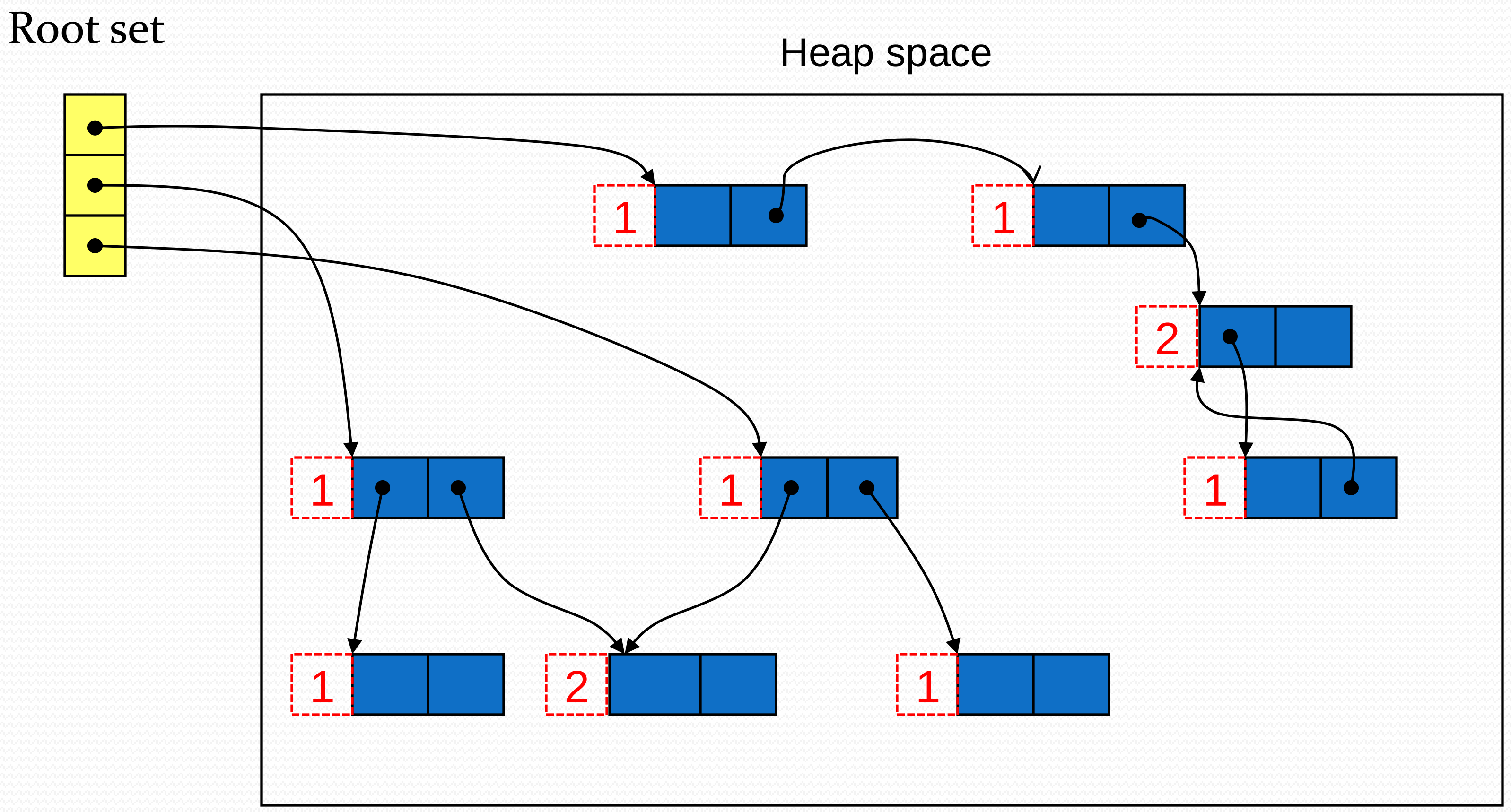
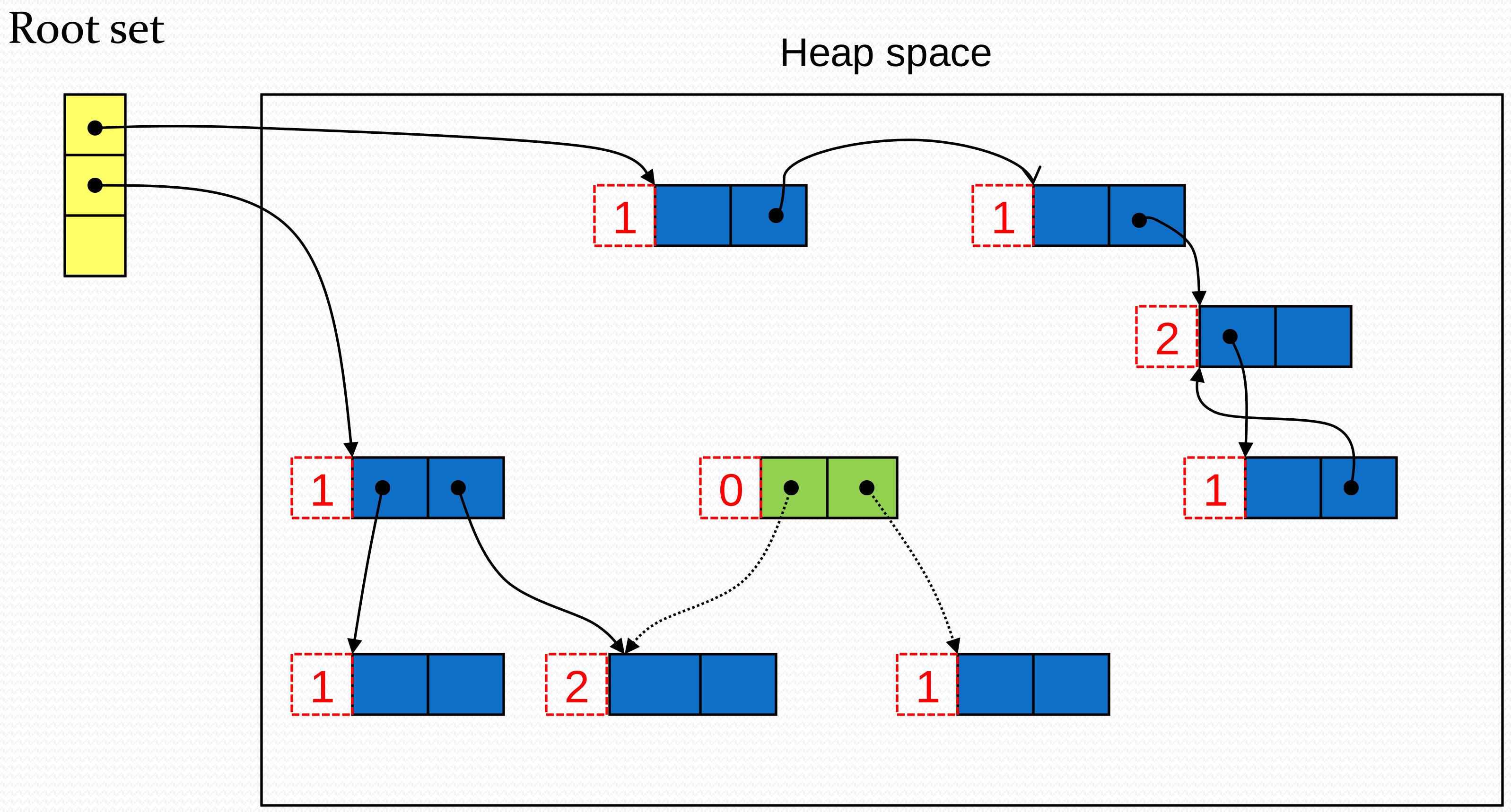
Advancements in RCGC

RCGC can be more attractive

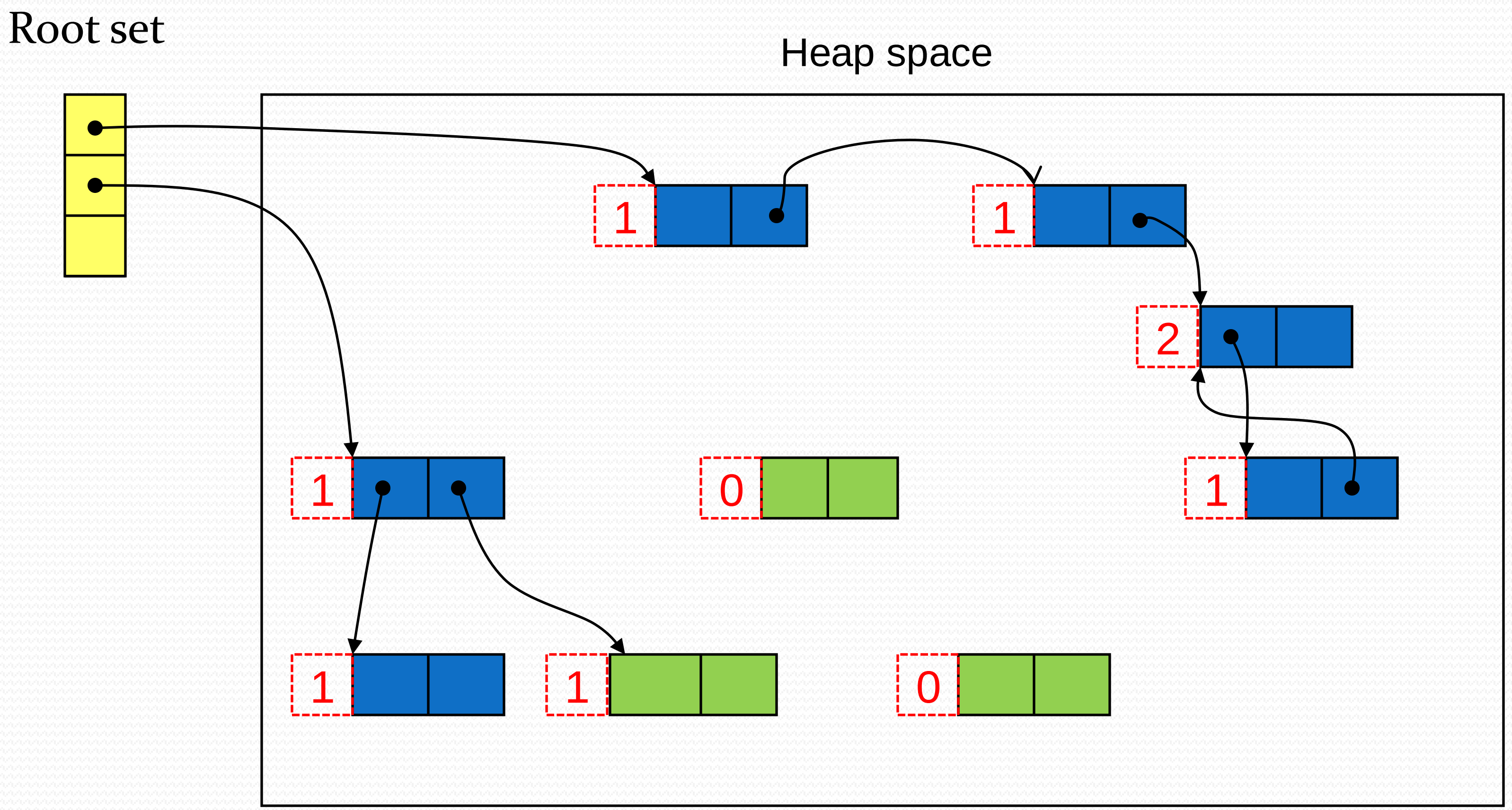
Reference counting example



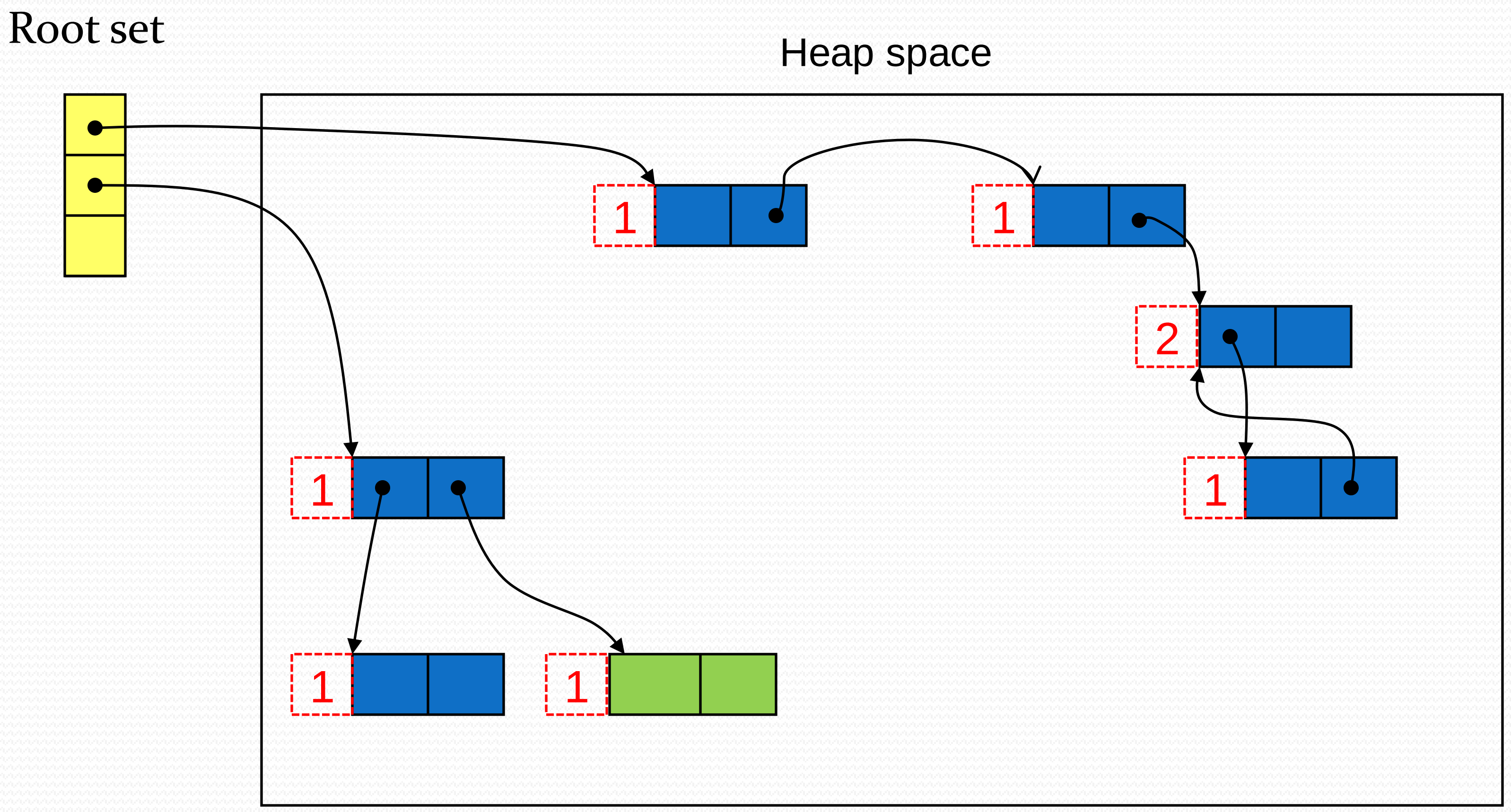
Reference counting example



Reference counting example



Reference counting example



Advantages of RCGC

- simple to implement
- Identify garbage as object dies
  - Immediate reuse of storage
- Good spatial locality of reference
  - Only objects in pointer reference need to be accessed
- Does not require additional heap storage to prevent GC from croaking
- Time overhead distributed throughout computation

Advantages of RCGC

- Adopted in several systems
  - Unix utilities awk and perl,
  - file systems,
  - Memory managemet in distributed systems
    - Reduced communication overhead

Deficiencies of RCGC

- Cost of removing last pointer unbounded
- Total overhead of adjusting RCs significantly greater than that of tracing collectors
- Substantial space overhead
- Inability to reclaim cyclic data structures

How do we overcome shortcomings?

- Problem
  - Cost of removing last pointer unbounded
    - Depends on size of sub-graph rooted at garbage object
- Solution
  - Non-recursive freeing
  - Weizenbaum: when last pointer to objet Q is deleted, simply push Q unto free-stack
    - Use free-list as a stack
    - RC field used to chain stack
    - Lazy deletion (update() unchanged)

Weizenbaum’s Algorithm

```
new() {
    if freeList == NULL
        abort “Memory exhausted”
    newcell = allocate() // pop stack
    for N in Children(newcell)
        delete(*N)
    RC(newcell) = 1
    return newcell
}
```

```
delete(N){
    if RC(N) == 1
        free(N)
    else decrementRC(N)
}

free(N){
    RC(N) = freeList // RC replace next
    freeList = N
} // push n unto the stack
```

Effects of Weizenbaum's Algorithm

- Less vulnerable to delays caused by cascading deletion
- If array is freed, all its pointers must still be deleted before its storage can be reused
  - May/not be noticeable
- Loses some benefits of immediacy
  - Memory inaccessible until data structure popped from stack
    - See new()

How do we overcome shortcomings?

- Problem
  - Total overhead of adjusting RCs significantly greater than that of tracing collectors
    - Overhead of maintain RC high on conventional hardware
      - Fetching counts may invalidate cache lines
      - Pages containing remote data may be paged in
    - ~ dozen instructions to adjust RC in both old & new *pointees*
    - What about iterating over a list?
- Solution
  - Deferred reference counting
    - Allow as few RC updates as possible
    - Deutsch-Bobrow Algorithm

Deutsch-Bobrow Algorithm

- Observation
  - Majority of pointer writes are made in local variables
  - Frequency of other pointer stores may be as low as 1%
    - True with modern optimizing compilers
- Deferred RC takes advantage of observation
  - Don't count references from local variables or stack
    - Use simple assignment
  - Only count references from heap objects

Implications of Deutsch-Bobrow

- Object no longer reclaimed as soon as it RC drops to 0
  - What about references from stack
- Objects with zero RC added to zero-count-table (ZCT) by delete()
- Periodically ZCT is reconciled
  - To remove and collect garbage
- Note: possible for other heap objects to store reference to entries in ZCT
  - Increment RC of entry
  - Remove entry from ZCT

Deutsch-Bobrow Algorithm

```
delete(N) {
    decrementRC(N)
    if RC(N) == 0
            add N to ZCT
}

update(R, S){
    incrementRC(S)
    delete(*R)
    remove S from ZCT
    *R = S
}
```

```
/* Three phase reconciliation */
reconcile(){
    for P in stack // mark the stack
            incrementRC(*P)
    for N in ZCT // reclaim garbage
            if RC(N) == 0
                    for  M in children(N)
                            delete(*M)
                    free(N)
    for P in stack S // unmark the stack
            decrementRC(*P)
}
```

Advantages of deferred RC

- Very effective at reducing cost of pointer writes
- Experience with Smalltalk implementation on Xerox Dorado in mid-eighties
  - Cut the cost of pointer manipulation by 80 %
  - Add small space overhead
  - Immediate vs deferred RC. [Ungar, 1984]

	Immediate	Deferred
Updates	15	3
Reconciliation		3
Recursive freeing	5	5
Total	20	11

Disadvantages of deferred RC

- Space overhead for ZCT
- ZCT can overflow
- Reduces RC advantage of immediacy

How do we overcome shortcomings?

- Problem
  - Substantial space overhead
    - Requires space in each object to store RC
    - Worst case: field large enough to hold total # of pointers
      - In heap and root set
- Solution
  - In practice objects don't have that many references
    - Typically each object receives just a few references at a time
  - Save space by using smaller RC field
    - Limited-field reference counting

Sticky reference counts

- The RC of an object cannot be allowed to exceed its maximum possible value
  - Its *sticky RC*
- Once a RC reaches this value, it is *stuck*
- It cannot be reduces since its true RC can be greater than its *sticky RC*
- It cannot be increased since it is limited by the size of the RC field

Adjusting *sticky* reference counts

```
incrementRC(N) {
    if RC(N) < sticky
        RC(N) = RC(N) + 1
}
```

```
decrementRC(N){
    if RC(N) < sticky
        RC(N) = RC(N) – 1
}
```

Restoring reference counts

- Why is this necessary?
  - An object cannot be reclaimed by RCGC once its RC reaches sticky
  - RC needs to be restored
  - Can use tracing collector
    - Can collect cycles

Tracing collection restores RC

```
mark_sweep () {
    for N in Heap
            RC(N) = 0
    for R in Roots
            mark(*R)
    sweep()
    if free_pool is empty
            abort "Memory exhausted"
}
```

```
mark(N){
    incrementRC(N)
    if RC(N) == 1
            for  M in children(N)
                    mark(*M)
}
```

Other RC Optimizations

- One-bit reference counting
  - Unique pointer *vs* shared pointer
- Using an 'Ought to be two' cache
  - A version of the one-bit RC
- Hardware reference counting
  - With other optimizations RC still more costly than tracing collectors
  - Need specialize hardware
    - Self-managing heap memory based on RC
    - Have not been successful commercially