

Copying GC

AKA the scavenger

Copying garbage collection algorithm

- Divides the heap into two equal parts
 - *to-space* & *from-space*
 - Use one part at a time (*to-space*)
- When *to-space* fills up, flip the roles
 - Old *to-space* becomes *from-space*
 - Old *from-space* becomes *to-space*
- Copy live objects from *from-space* to *to-space*
 - Begin by following pointer from the root set

Copying garbage collection

- Also known by the following names
 - Stop-and-copy garbage collection
 - Scavenger
 - Garbage objects are simply abandoned in old space
 - Picks out worthwhile objects amidst garbage & take them away

Advantages of Copying Collection

- Live data structures are compacted in *to-space*
 - No fragmentation
- Allocate object fast and efficiently
 - No free list
 - Out-of-space check is a simple pointer comparison
 - New memory allocated by simply incrementing free space pointer

Advantages of Copying Collection

- Collects in time proportional to live data
- Avoids stack and/or pointer reversal
- Imposes no overhead on mutator operations
 - *e.g. pointer updates*
- Advantages over RCGC and MSGC have lead to its widespread adoption

Disadvantages of copying collection

- Wastes half your memory
- Copying takes time

Allocation in copying collector

```
init() {  
    to_space = Heap_bottom  
    space_size = Heap_size / 2  
    top_of_space = to_space + space_size  
    from_space = top_of_space + 1  
    free = to_space  
}  
new(n){  
    if (free + n > top_of_space) {flip()}  
    if (free + n > top_of_space) { abort "Memory exhausted"}  
    new_cell = free // allocate()  
    free = free + n  
    return new_cell  
}
```

Flipping the spaces

```
flip() {  
    to_space, from_space = from_space, to_space  
    top_of_space = to_space + space_size  
    free = to_space  
    for R in Roots  
        R = copy(R)           // Root pointer now points to copy of R  
}
```


Copying for variable-sized objects

// P points to memory location, not an address

copy(P) {

 if (atomic(P) or P == nil) return P // P is not a pointer

 if not forwarded(P) // P stores a forwarding address after copied

 n = size(P)

 P = free // reserve space in to_space for copy of P

 free = free + n

 temp = P[o] // P[o] holds forwarding address

 forwarding_address(P) = P

 P [o] = copy(temp) // Restore P[o]

 for i = 1 to i = n - 1 // Copy each field of P in to P

 P [i] = copy(P[i])

 return forwarding_address(P) // Stored in P[o]

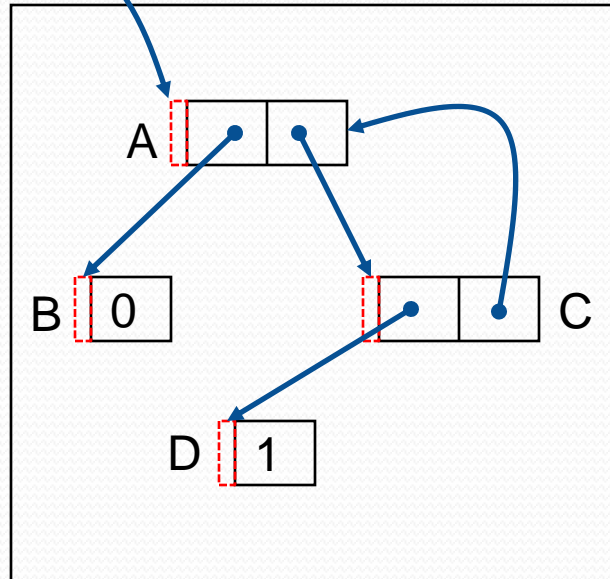
}

Example of wow algorithm works

Root set



To-space



From-space

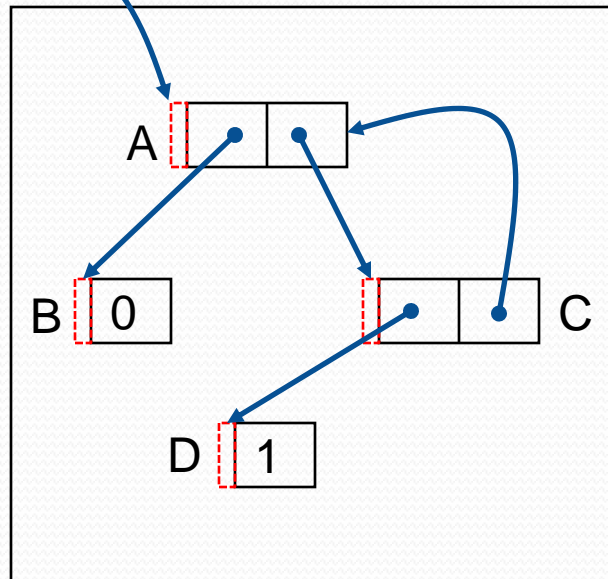


Flip the spaces

Root set



From-space



To-space

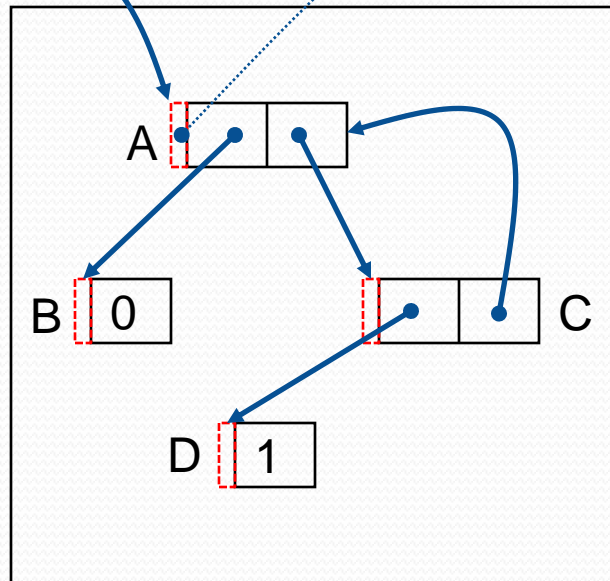


Copy root object first

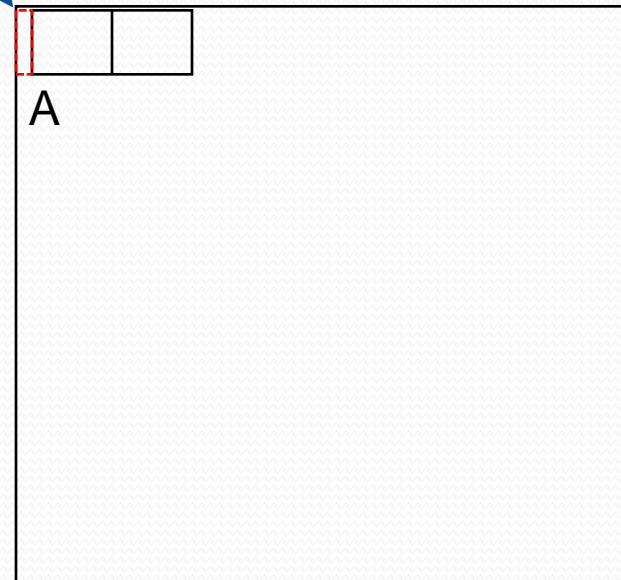
Root set



From-space



To-space

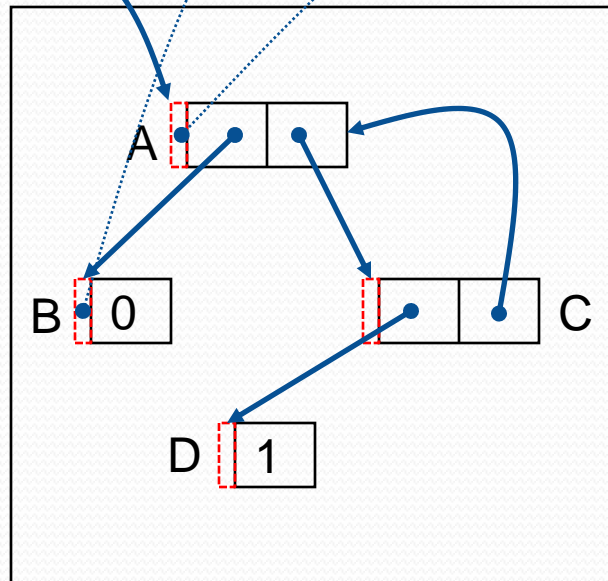


Then copy B: left child of A

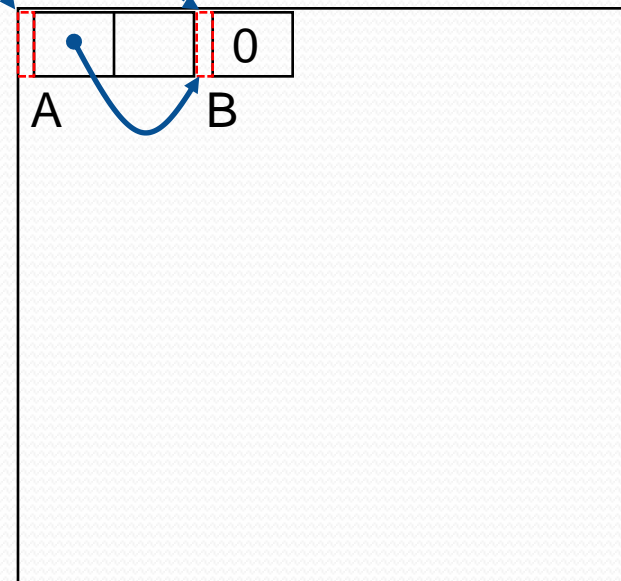
Root set



From-space



To-space

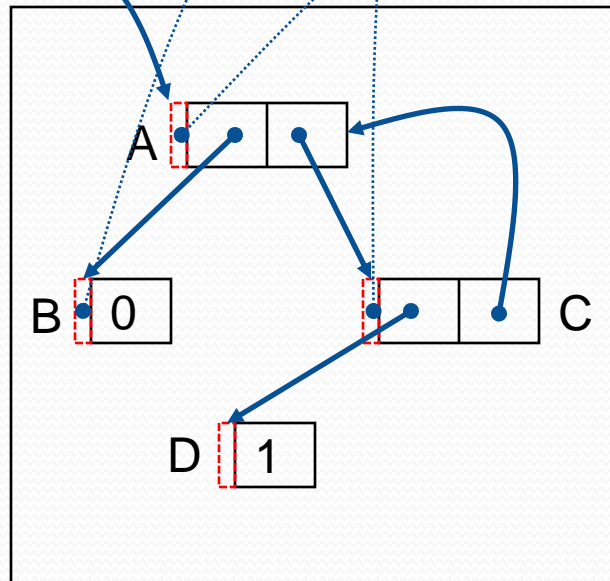


Now copy C: right child of A

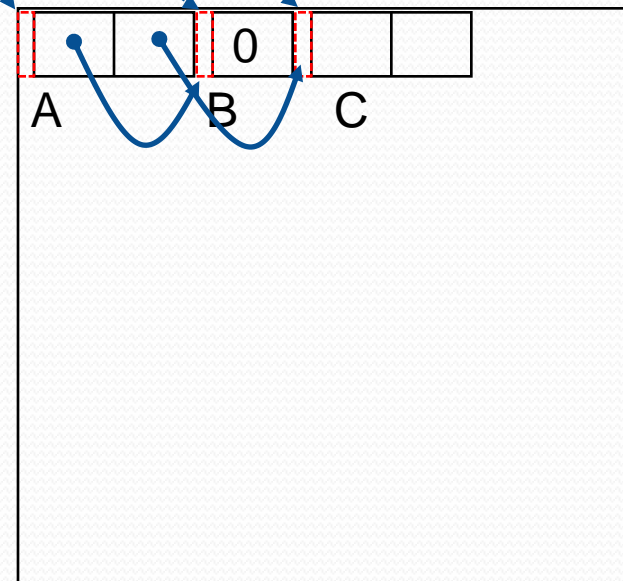
Root set



From-space

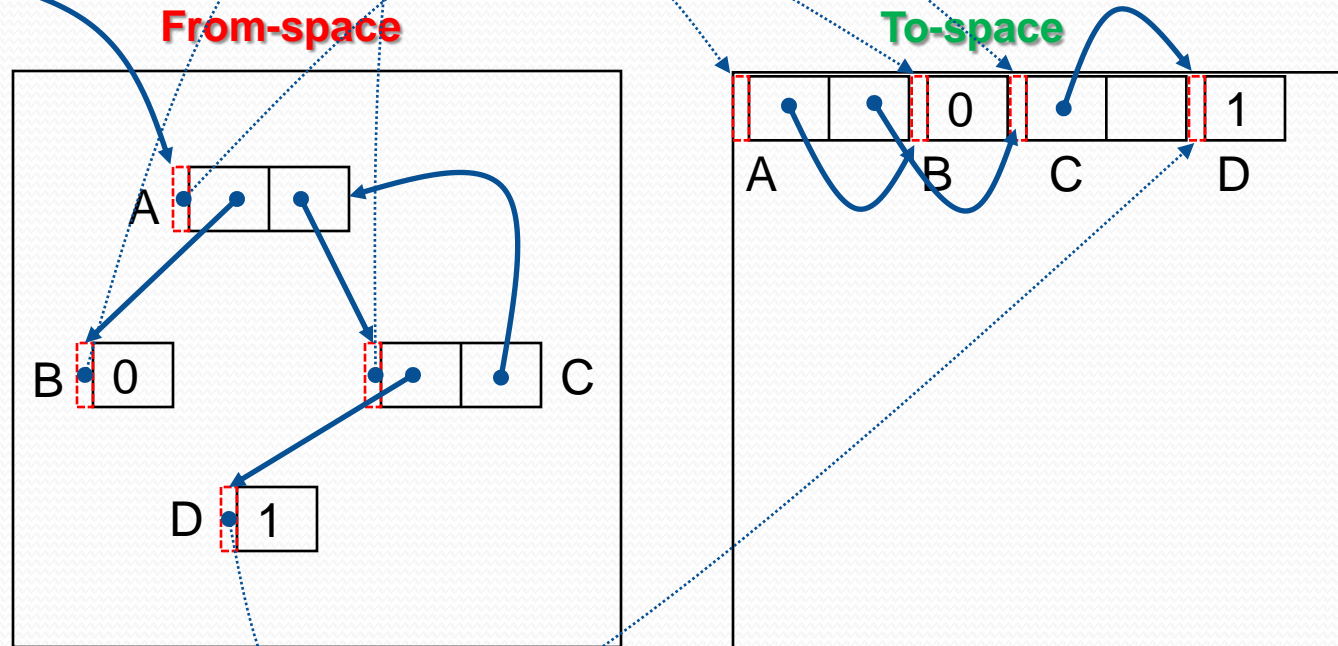


To-space



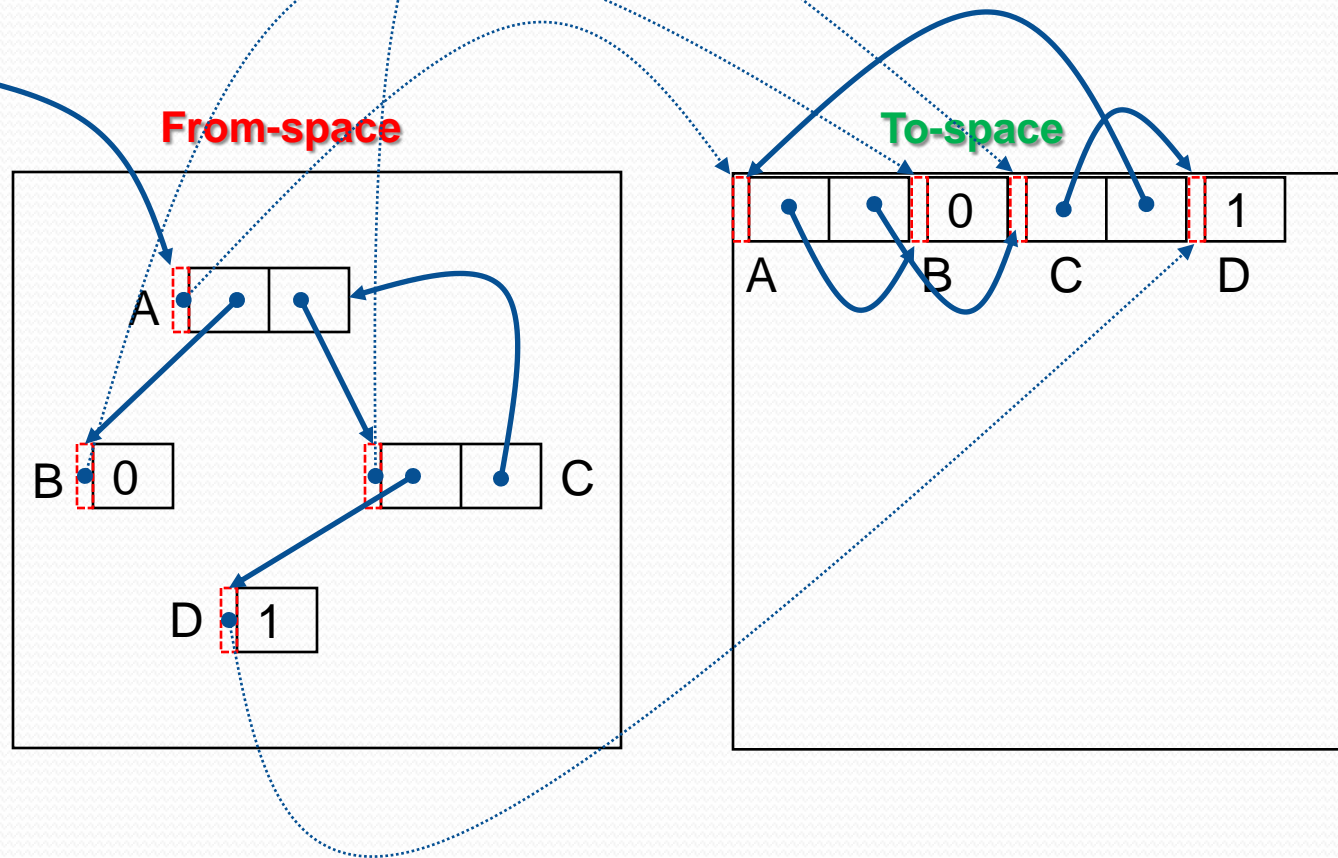
Copy D: left child of C

Root set



Copy A: right child of C

Root set



Analysis of copying collection

- Assume there are L words of live data in heap of size H words
- Cost of GC is kL
 - Realistic value is $10L$ ($k = 10$)
- Cost per reclaimed word

$$\frac{kL}{\left(\frac{H}{2} - L\right)}$$

- This has no lower bounds as H grows
- If $H = 4L$ then $k \sim 10$