

Dynamic Memory Allocation

CS31

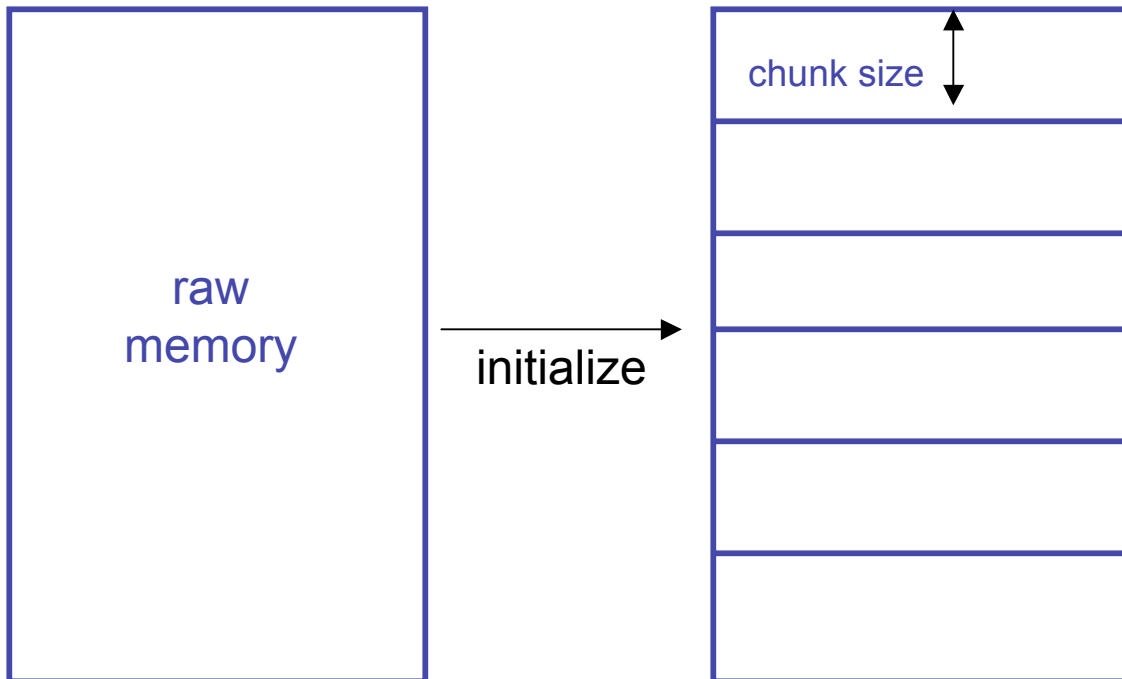
Pascal Van Hentenryck



Uniform Allocation

Imagine that all requested blocks are of the same type, hence the same size.

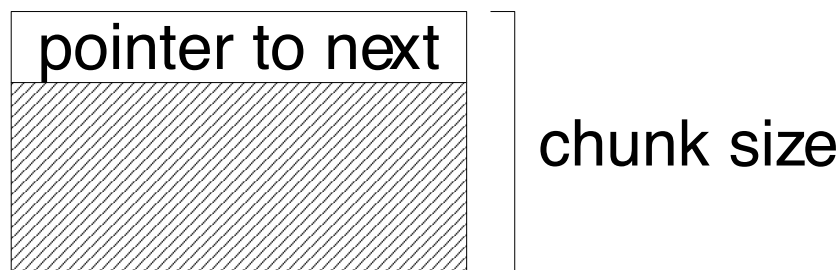
We can divide a pool of storage into chunks of the right size initially.



At any given time, some of the chunks will be in use by the program, and some will be free, available for use.

Free List

How do we keep track of which chunks are used, which are available? Keep a singly linked list of free (available) storage chunks. Keep the list in the memory itself -- it's free memory, so why not use it? Thus, nodes and elements are not separate objects. Each node of the list looks like

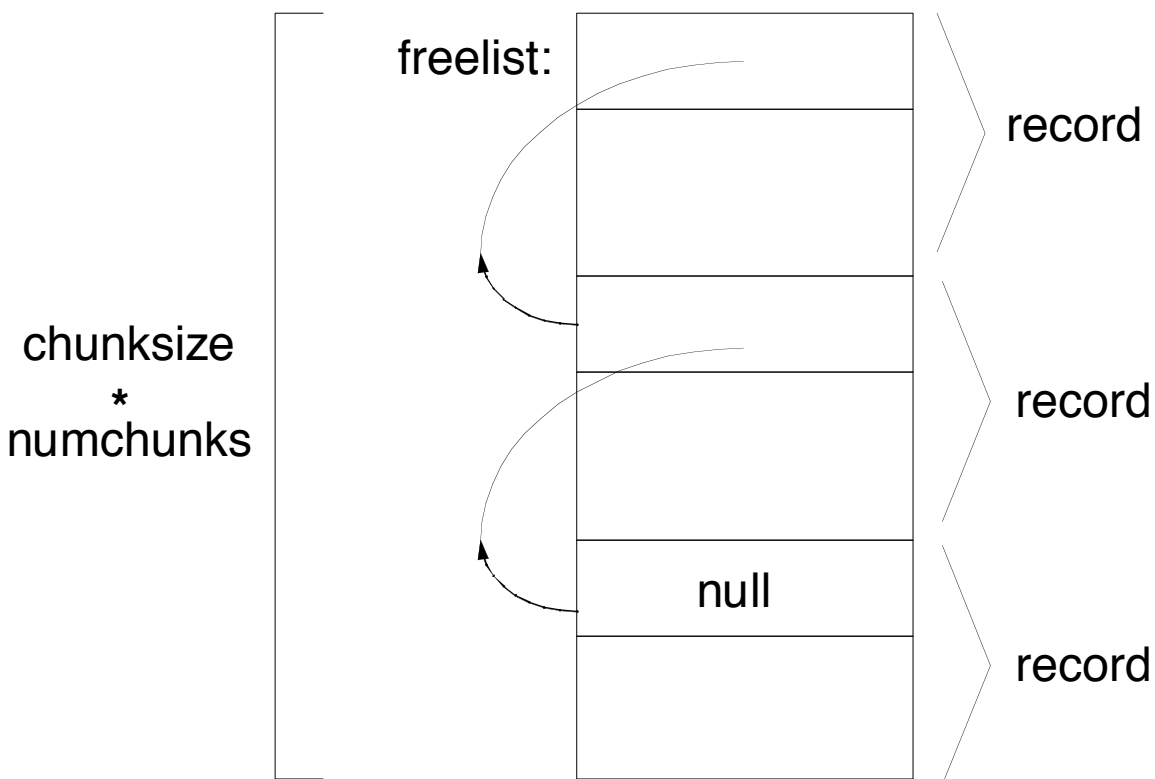


Before we initialize, the storage is just raw memory:

```
        .data
        chunksize =          26
        numchunks =         10
freelist: .word             0:260
```

Initial Configuration

After initialization, the freelist needs to look like this:



How can we make it so?

Initializing the Free List

```
# Params:          word chunksz, word numchunks, word
freelist
# Local variables:
# t0: a, the address of uninitialized memory
# t1: i, the number of chunks to init/loop count
# t2: size of chunks
# t3: offset to next-pointer (address of next chunk)

        .data
.eq      chunksz      16      # offsets from sp
.eq      numchunks    12      # of parameters
.eq      top          8

        .text
sub      $sp,$sp,4
sw      $ra,4($sp)      # push ra
lw      $t0,top($sp)    # a = freelist
lw      $t1,num($sp)    # i = numchunks
lw      $t2,size($sp)  # get chunksz
sub     $t1,$t1,1      # while (i>0)
add     $t3,$t2,$t0    #         offset=a+size
sw      $t3,next($t0)  #         a.next=offset
move    $t0,$t3        #         a=a.next
sub     $t1,$t1,1      #         --i
bnez    $t1,loop       # end while
sw      $0,next($t0)   # a.next = null
lw      $ra,4($sp)
add     $sp,$sp,4      # pop ra
jr      $ra            # return
```

New and Delete

To get a new object, just take it off the front of the free list.

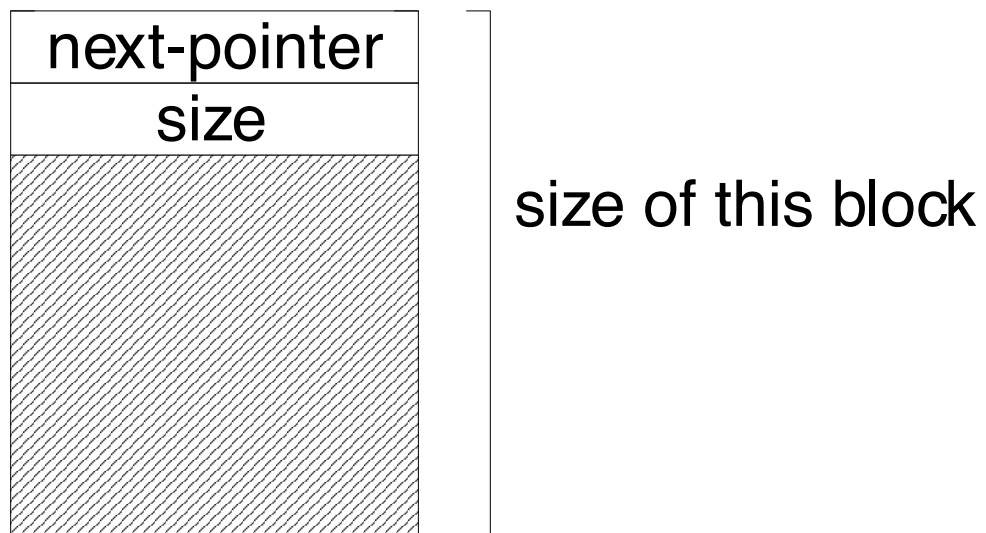
To delete an object, just put it back on the front of the free list.

Non-Uniform Allocation

Start with one big block of memory

- new has a size parameter
- on request for a small block, divide a block
- keep a freelist of available blocks

Each element of the freelist looks like this:



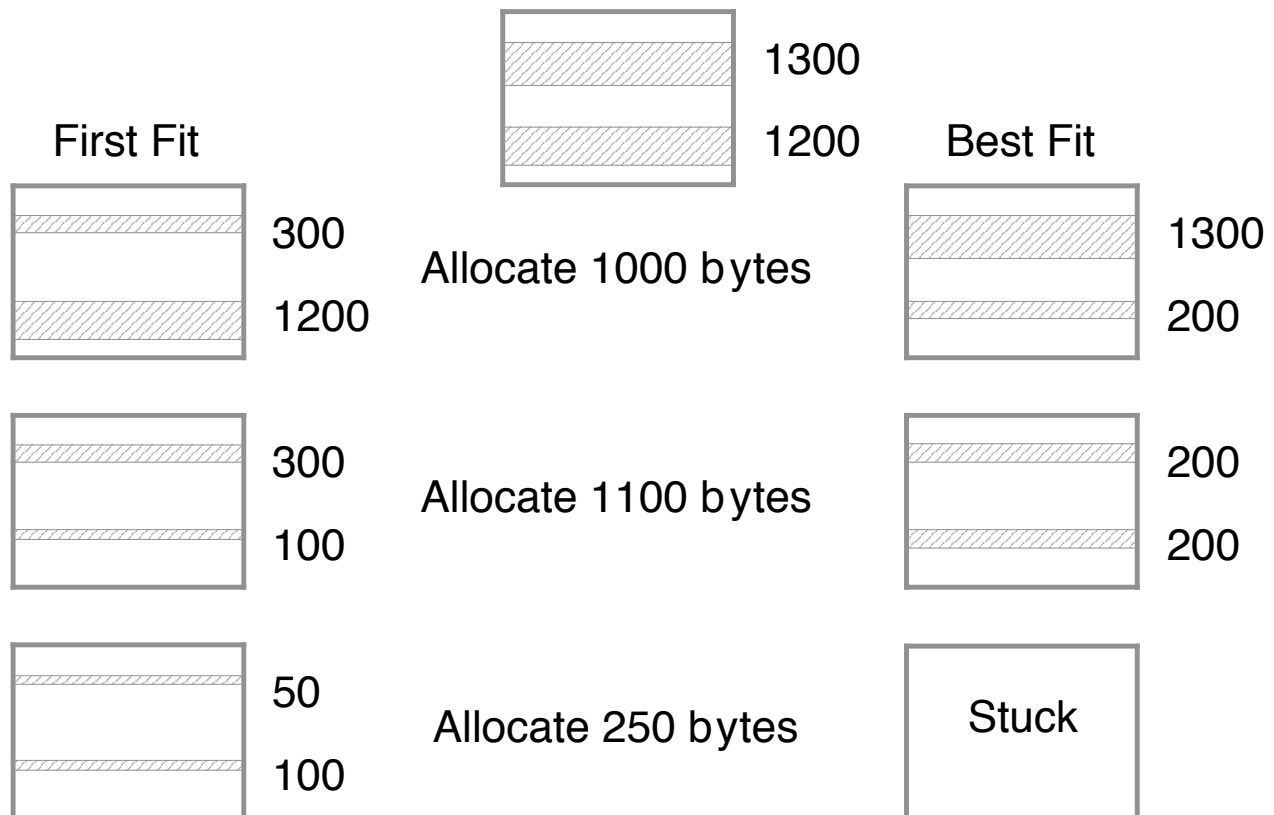
Blocks vary in size. How do we find one of the size we need?

Allocation Strategies

Two options: search the free list for...

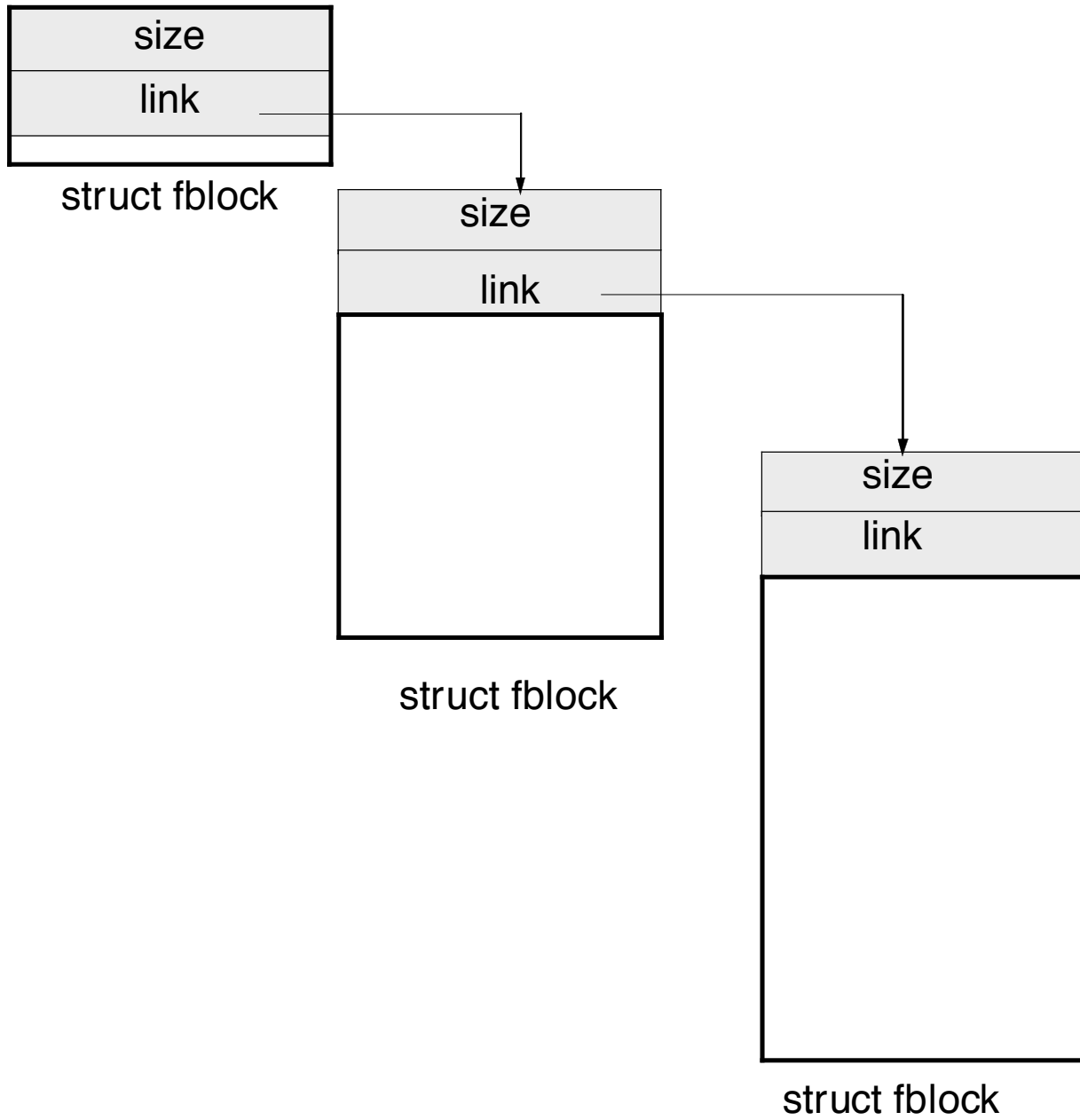
- the smallest block that's large enough ("best fit")
- any block that's large enough ("first fit")

Simulations show that first-fit is better. Example:



Counterexamples are also possible.

First Fit



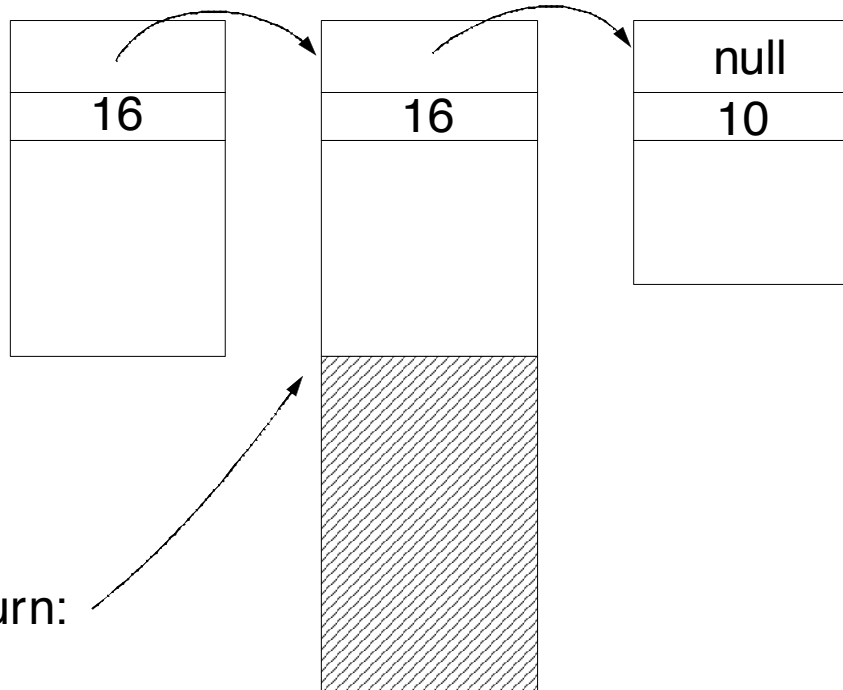
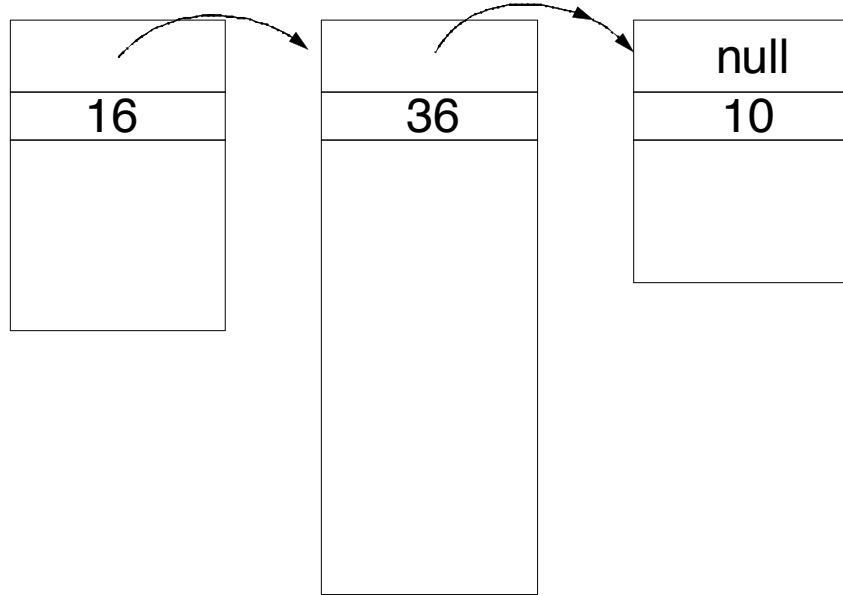
First-Fit Allocation

To allocate a block of size n ,

1. walk down free list until you find an element of size $\geq n$
2. break off a chunk of size n and give back a pointer to it
3. link the remainder, if any, back into the list

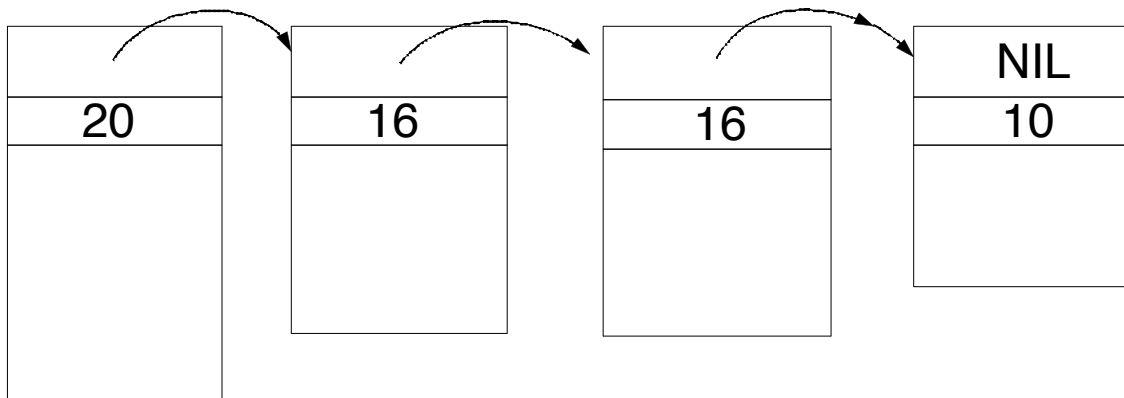
First Fit Example

new (20 words)



Deletion

What happens when we apply the simple algorithm?



We've got **FRAGMENTATION!**

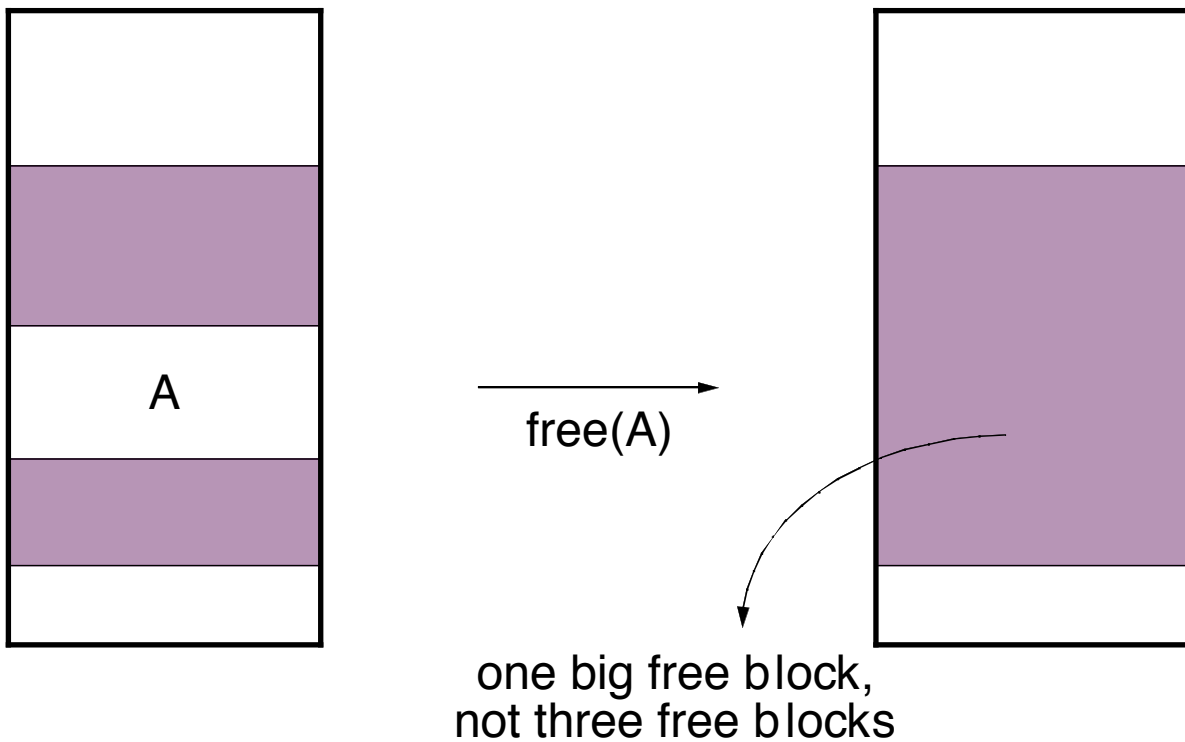
Two contiguous, free blocks of memory are separated in the list.

A request for 36 words of memory will fail, even though there are 36 contiguous words available.

Fixing Fragmentation

Basic Idea

- when storage is freed, see if it is contiguous to anything else on the freelist
- make freelist doubly linked, to allow insertion anywhere

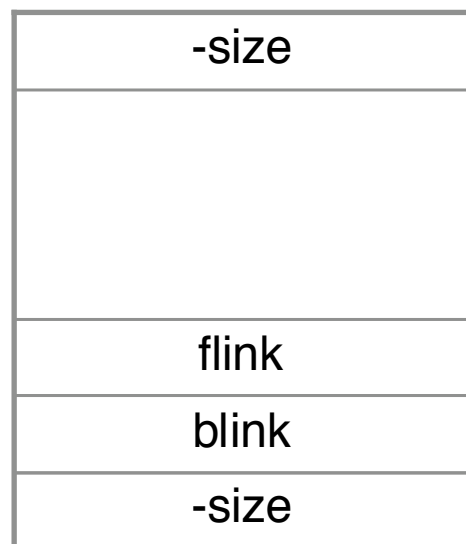


Fixing Fragmentation

Boundary Tags



Allocated Block



Free Block

Fixing Fragmentation

When deleting a block

- look if the block above it is in the free list (how?)
- look if the block below it is in the free list (how?)

If one is in the free list

- merge and update the pointers
- put in front of the freelist

Compaction

Compact all used cells

- at the beginning of the memory

How to do it?

- three-phase algorithm

Three phases

- Find the new address
- restore the pointers
- move the cells

Assumption

```
class Cell {  
    int marked;  
    Cell forwarding;  
    int size;  
    ...  
};
```

Compaction

Compact all used cells

- at the beginning of the memory

How to do it?

- three-phase algorithm

Three phases

- Find the new address
- restore the pointers
- move the cells

Assumption

```
class Cell {
    int marked;
    Cell forwarding;
    int size;
    ...
};
```

Compaction

Why is difficult?

- we cannot simply move a cell
- we need to update the pointers/references to it as well

How to find these addresses?

- that would take too much time
- we would like a linear time solution

Compaction

Intuition

- 3 phases

First Phase: New Addresses

- for each object, we compute its new address

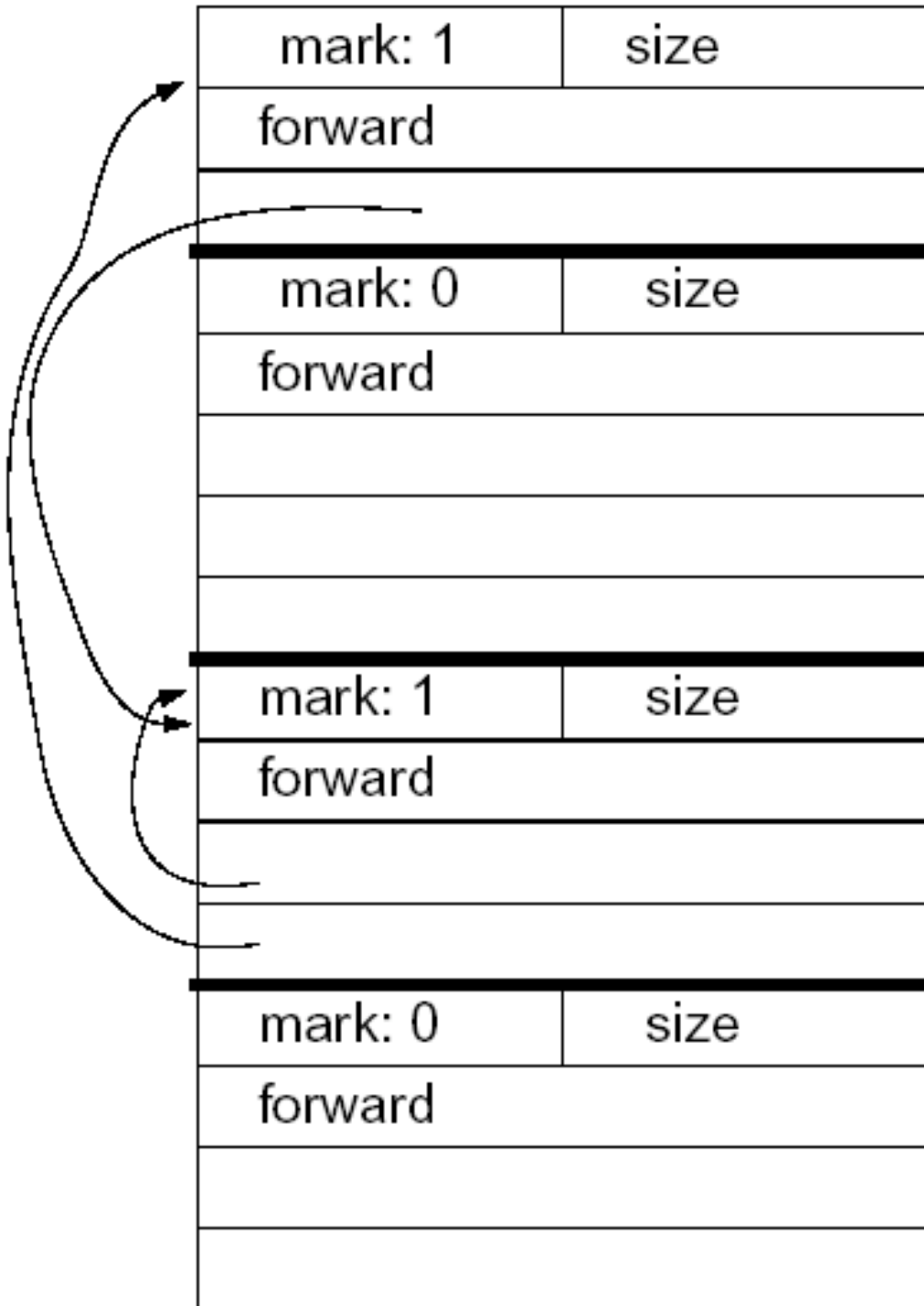
Second Phase: New Pointers

- for each object, we update its internal pointers

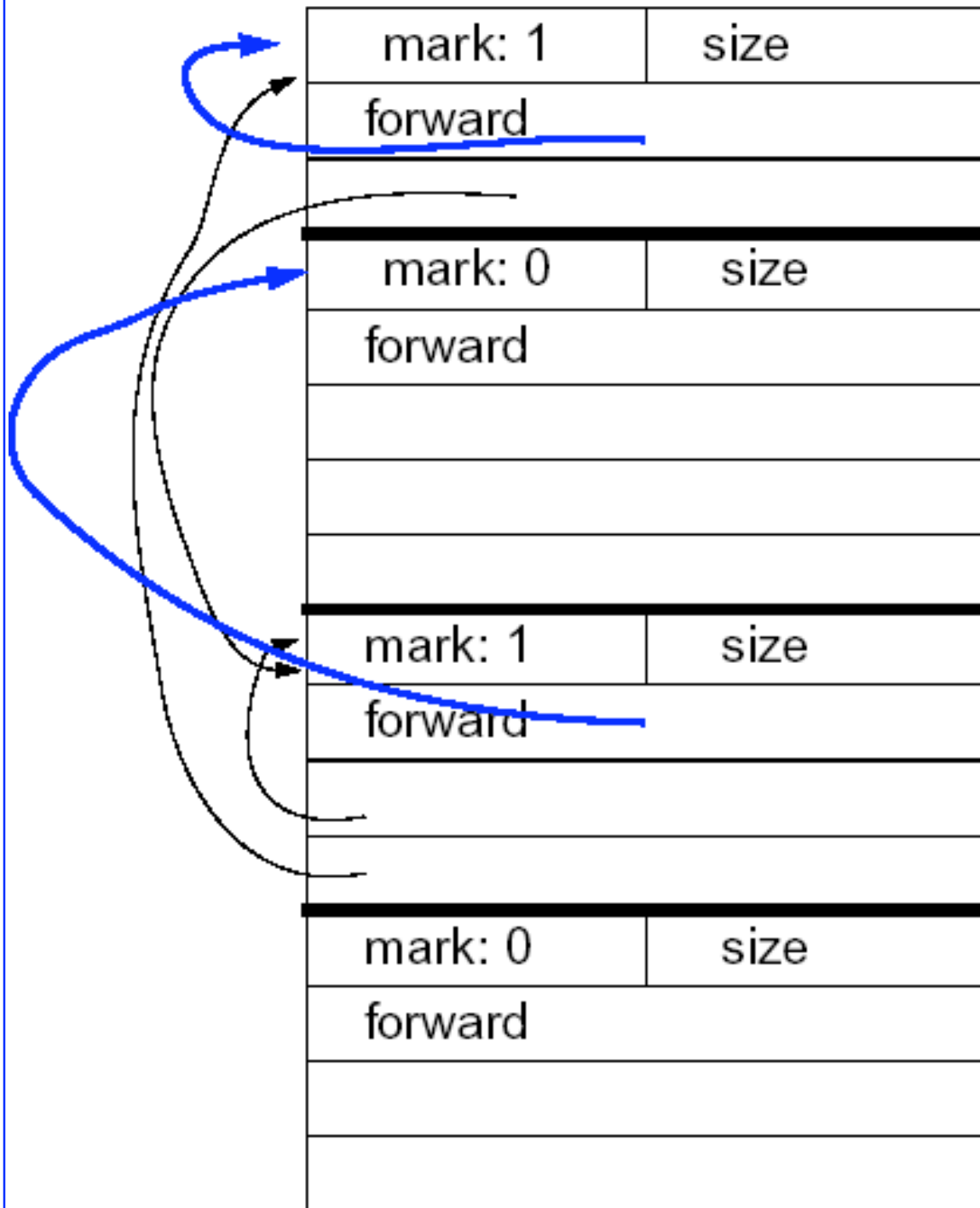
Third Phase: Compact

- we move each object at the right place

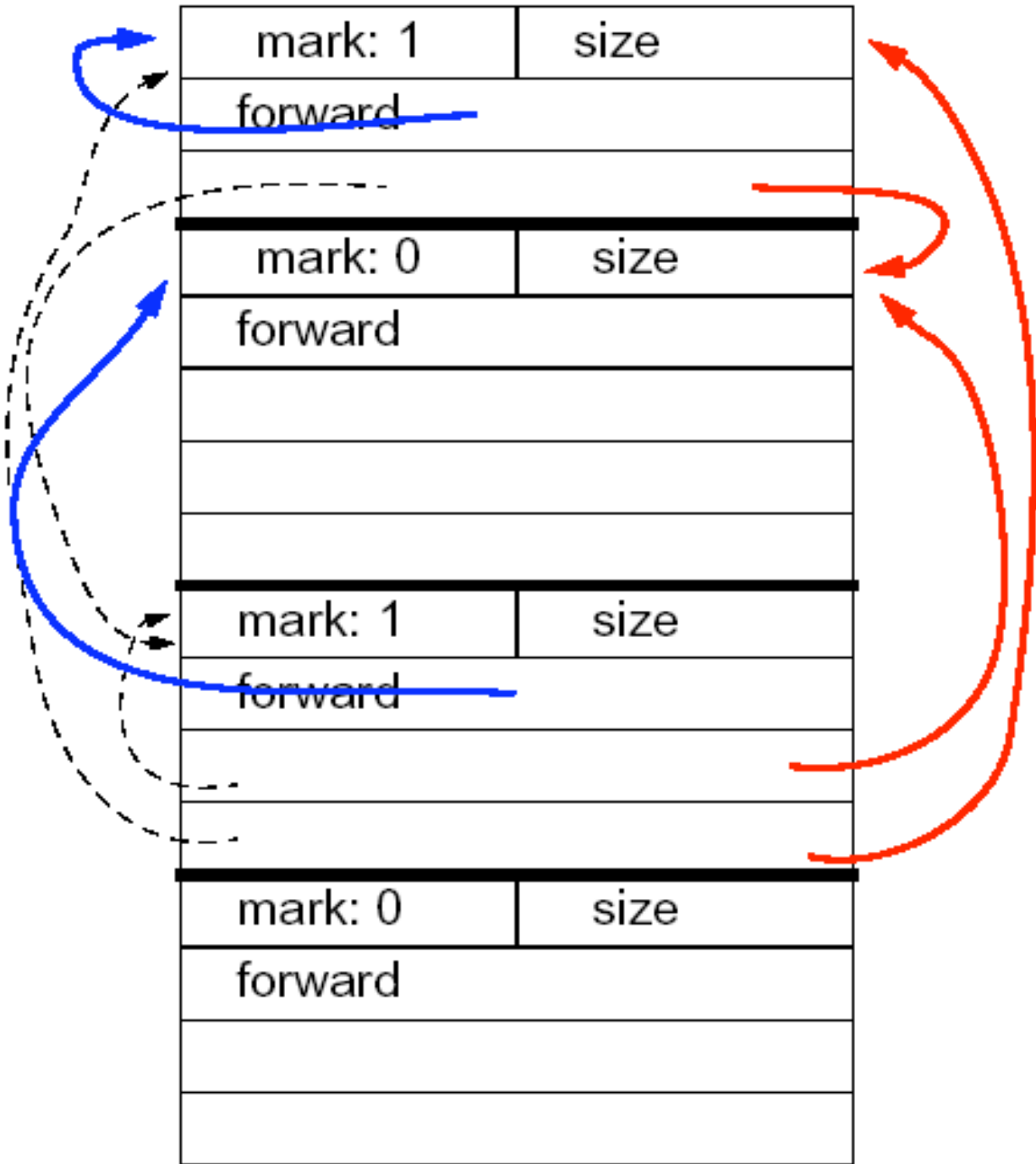
Before compaction



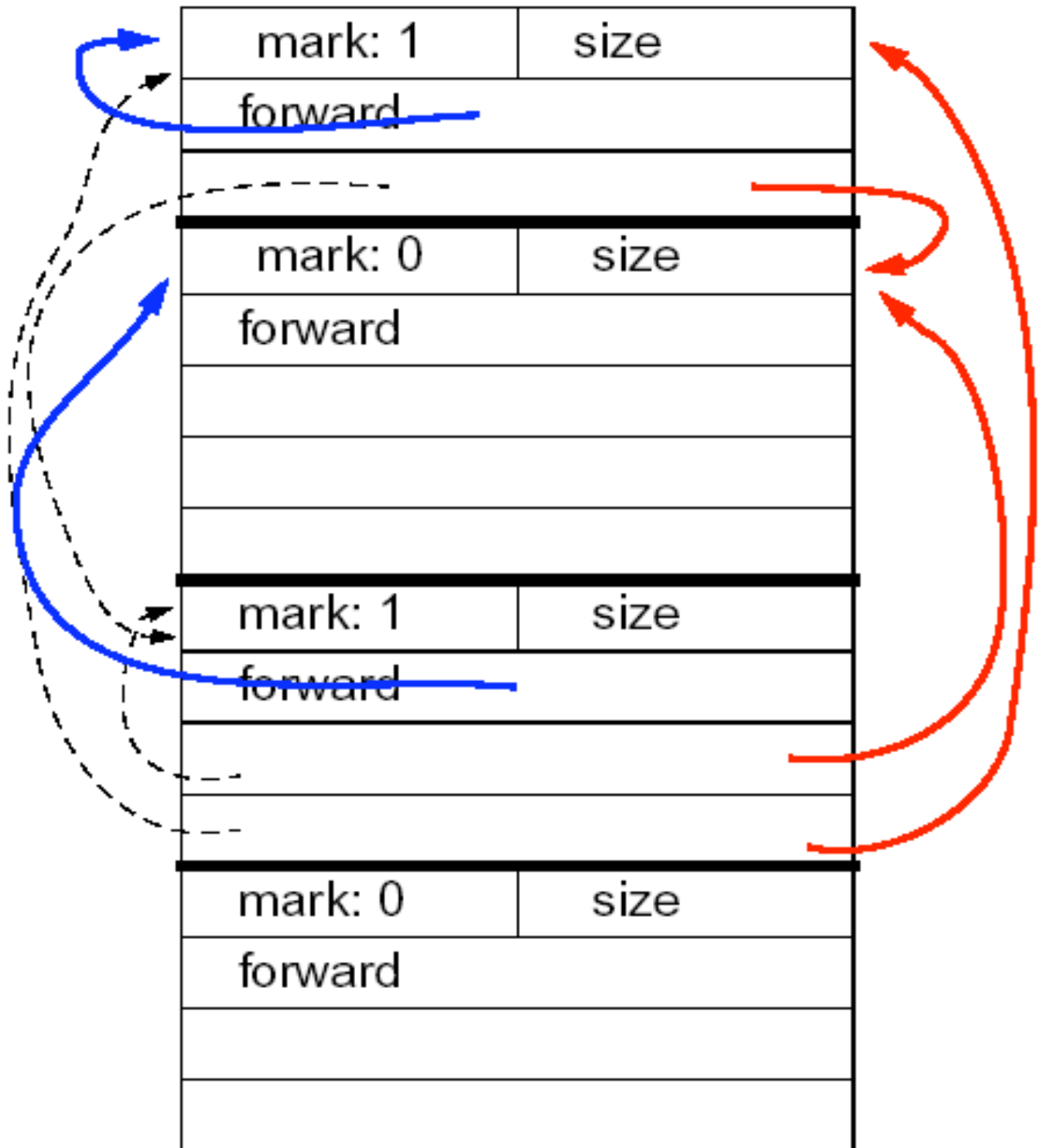
First Step: New Addresses



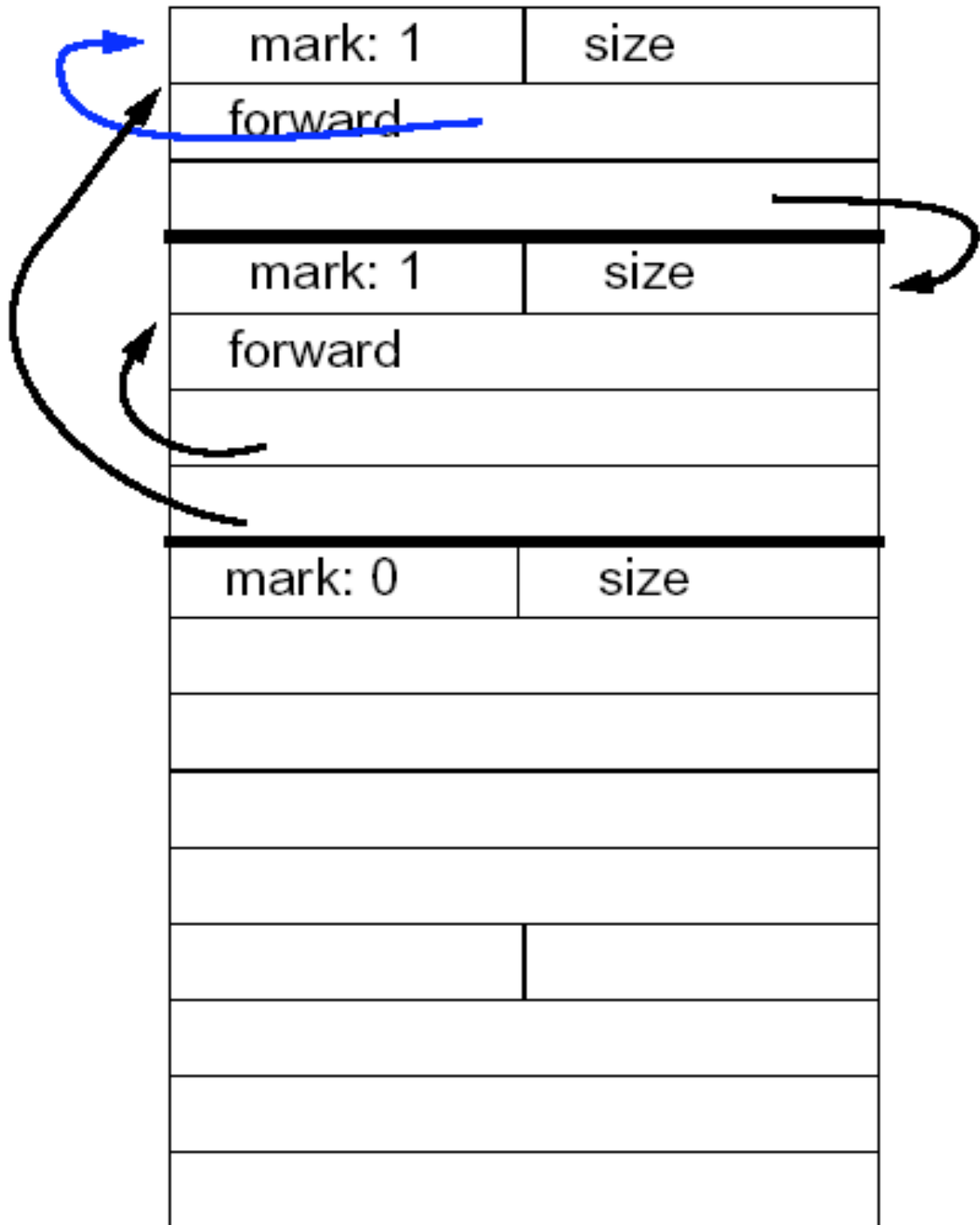
Second Step: Fix pointers



Third Step: Move objects



Third Step: move objects



Compaction

First phase

- Compute the new addresses
- Simply go over the memory and see what should be preserved

```
destination = first;
P = first;
while P <= last do
    if P.marked then
        P.forwarding = destination;
        destination += P.size;
    P += P.size;
```

This assumes that empty memory is organized as cells as well.

Compaction

Second phase

- Adjust the pointers

```
P = first;
while P <= last do
  if P.marked then
    for each pointer q in P do
      p.q := p.q.forwarding;
  P += P.size;
```

Note that the pointers are now correct but the cells are not at the right place

It is necessary to do so before moving the cells because otherwise the pointers would point to something irrelevant.

Compaction

Third Phase

- Move the cells

```
P = first;
while P <= last do
  if P.marked then
    copy P.size words
      from P
      to P.forwarding
  P += P.size;
```

The rest of the memory should also be organized as a big empty cell.