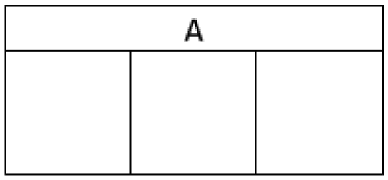
1 Garbage Collection (Continued)

We will now discuss an algorithm that we can use to perform garbage collection. This algorithm is known as the **mark-compact algorithm**.

1.1 Layout and Notation

In the following sections, we'll make use of the following box to represent heap space. Each square box represents one word in the heap (so each group of boxes represents three words of contiguous heap memory).



The (A) represents an address to this group of memory (like `0x100`).

The first box of each group is some metadata that we'll need for garbage collection. For our algorithm here, we will need to make use of this metadata. However, there are algorithms out there that don't require metadata.

In any case, the important thing to remember is that, under this representation, **each pair** requires 3 words in the heap.

1.2 The Algorithm

At a high level, our algorithm looks like

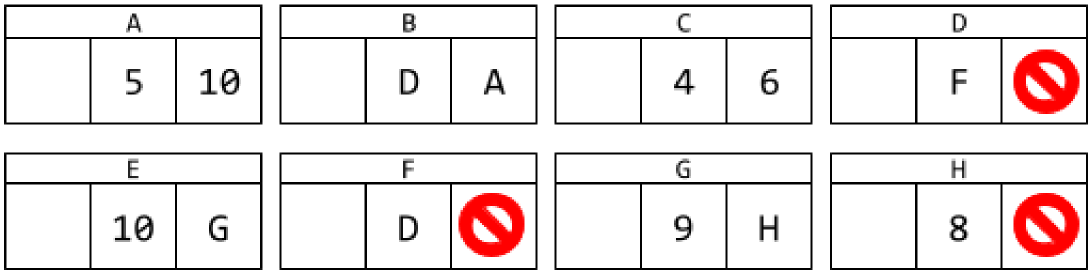
```
mark(roots):
    for ref in roots:
        mark_heap(ref)

mark_heap(r):
    if r.marked:
        return
    r.marked = true
    for (i, r') in r:
        if ispair(r')
            mark_heap(r')

fwd_headers():
    from = 0
    to = 0
    while move_from < HEAPEND:
        if from.marked:
            from.fwd = to
            to += 3 words/from.size
        from += 3 words/from.size

fwd_internal(roots):
    for ref in roots:
        update_fwd(ref)
        fwd_heap(ref)
fwd_heap(r):
    if r.fwded:
        return
    r.fwded = true
    for (i, r’) in r:
        if ispair(r’)
            r[i] = getfwd(r’)
            fwd_heap(r’)

compact():
    ...
```

Roughly speaking, we can break each of these methods into four groups:

1. `mark(roots)` and `mark_heap(r)`
2. `fwd_headers()`
3. `fwd_internal(roots)` and `fwd_heap(r)` (forwarding the internal pointers)
4. Compacting.

1.3 Motivating Example

Let’s consider the following heap structure.



Note that this example doesn’t correspond to any particular program.

1.3.1 Step 1: Finding the Root Set

First, we want to find all the references that are currently on the stack (or registers). These are called the **root set**.

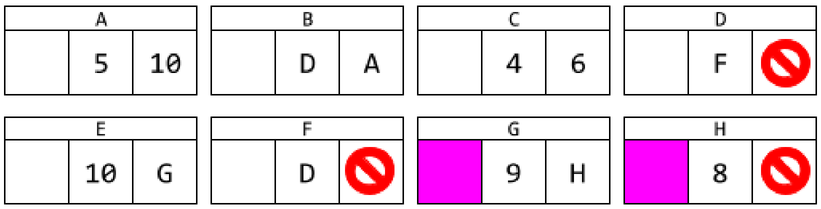
Right now, we don’t know what is considered garbage (since we don’t know what the stack or the registers look like). So, let’s suppose G and B are the only two references on the stack (or registers, depending on implementation). We call this the root set (the roots of your traversal into the heap). Let’s suppose we want to *clean* the heap.

1.3.2 Step 2: Marking Heap to Find Live Data

Now, we want to call the **mark** function with our root set. This is where we’re going to *mark* the memory in heap that are still in use (i.e., should not be garbage collected). This process is effectively depth-first search.

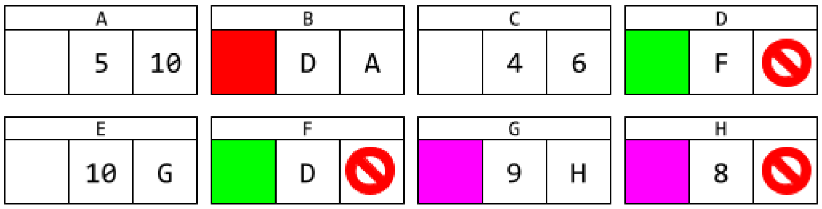
When we call **mark** with our root set, we’re iterating over each root, which in our example is G and B.

- We first call `mark_heap` with G. This will mark G, and then recursively call `mark_heap` with B and thus mark B as well. The result of marking is shown below:

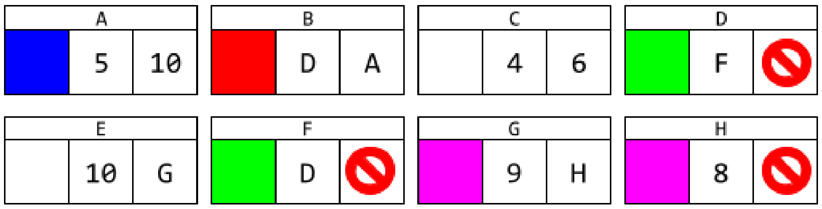


  Note that we're only considering non-`nil` pairs.

- Next, we call `mark_heap` with B. This will mark B, and then
  - recursively call `mark_heap` with D, marking it. Then, we'll recursively call `mark_heap` with F, marking it. Finally, we recursively call `mark_heap` with D, but since this has already been marked we don't need to do anything.



  - after that's done, we cal recursively call `mark_heap` with A, marking it. Note that, at A, there's no other pairs (only raw numbers), so we're done.

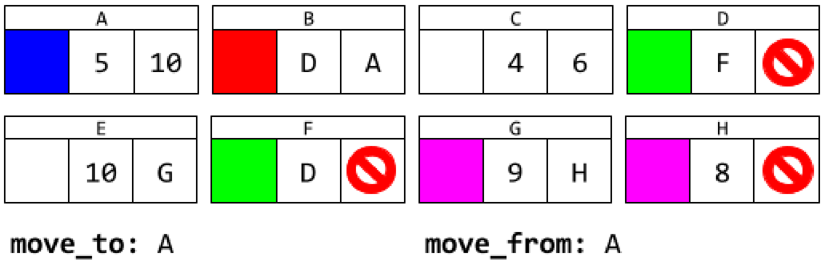


At this point, we're done. Notice how memory (C) and (E) haven't been marked; this means that they're garbage. Our goal, then, is to move `r15` to between F and G, and start allocation there! This, however, means we need to move everything back (compacting the heap).

1.3.3 Step 3: Forward Headers

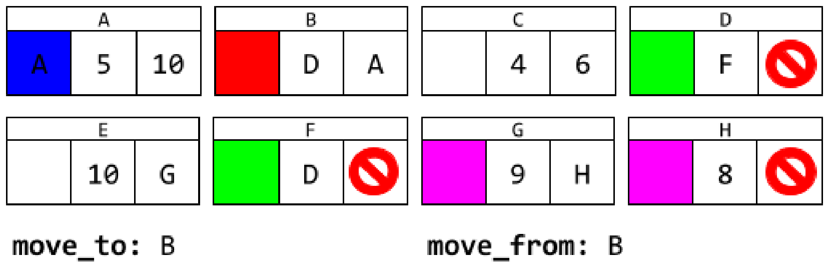
We'll make use of `fwd_headers`. For each marked pair, this function will set the new address (to store that pair after compating) in the pair's metadata (the first node).

Initially, we'll set `move_to` and `move_from` (`to` and `from` in the code, respectively) to `A` (the first memory location in heap).

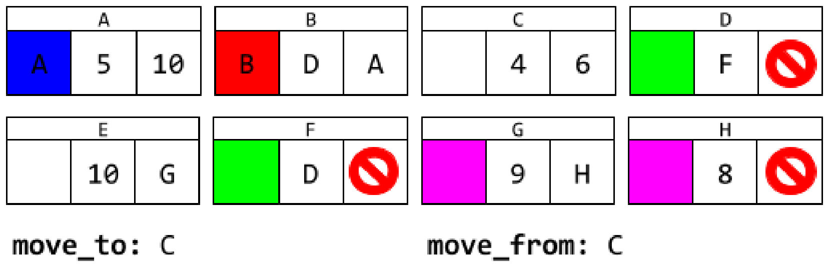


Iterating over each block of memory, we have

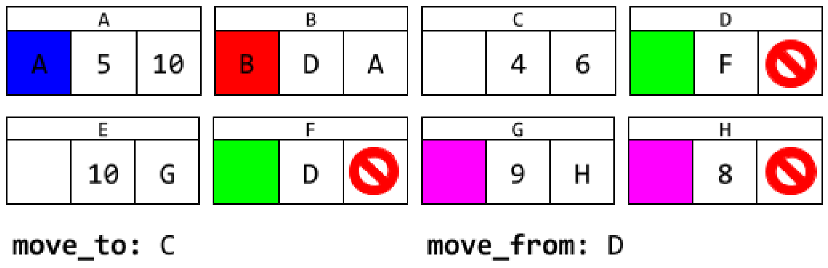
1. Since `move_from` $= A$ is marked, we set its metadata to `move_to` $= A$ (indicating that we'll move A to A after compacting). We also need to increment `move_to` and `move_from`. This gives us the following diagram:



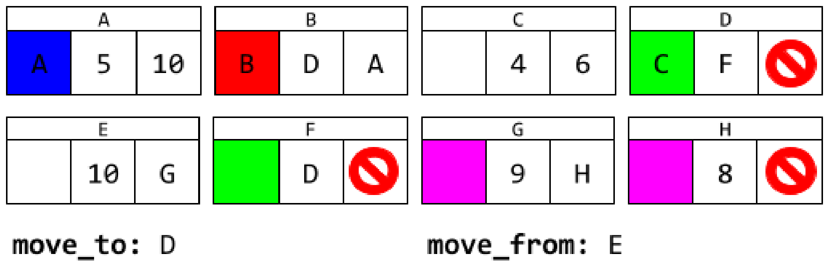
2. Since `move_from` $= B$ is marked, we set its metadata to `move_to` $= B$. We also need to increment `move_to` and `move_from`. This gives us the following diagram:



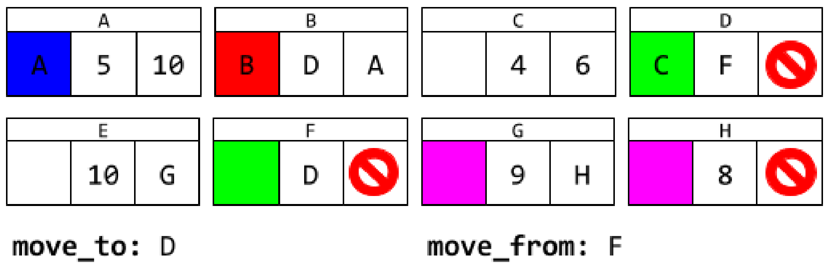
3. Since `move_from` $= C$ is **not** marked, we only increment `move_from`. This gives us the following diagram:



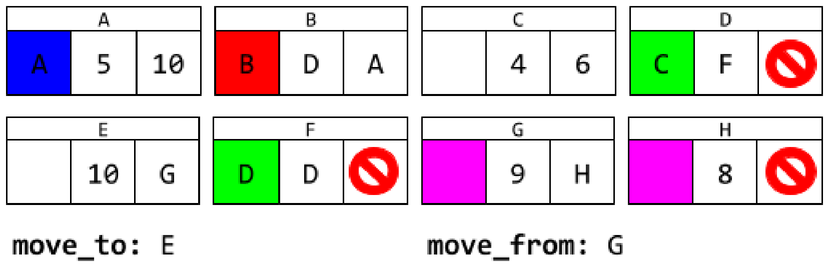
4. Since `move_from` $= D$ is marked, we set its metadata to `move_to` $= C$ (indicating that, after compating, D should be moved to C's location). We also need to increment `move_to` and `move_from`. This gives us the following diagram:



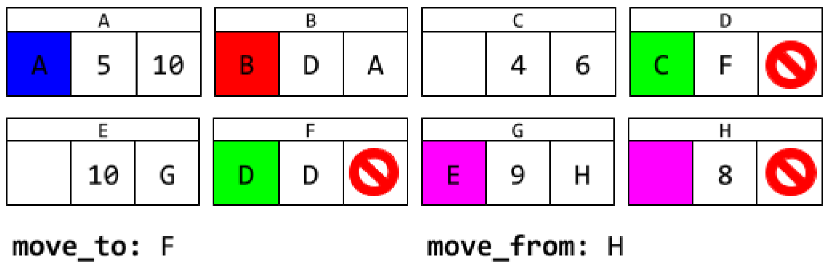
5. Since `move_from` $= E$ is not marked, we only increment `move_from`. This gives us the following diagram:



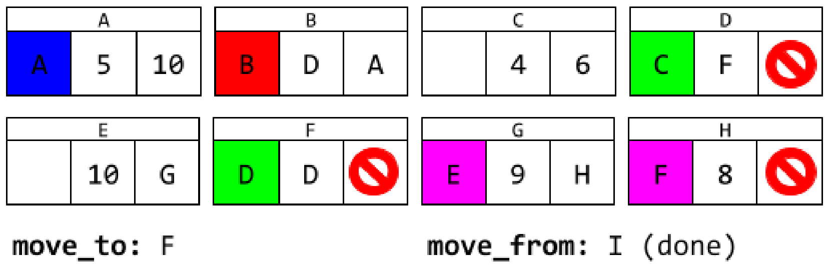
6. Since `move_from` $= F$ is marked, we set its metadata to `move_to` $= D$ (indicating that, after compating, F should be moved to D's location). We also need to increment `move_to` and `move_from`. This gives us the following diagram:



7. Since `move_from` $= G$ is marked, we set its metadata to `move_to` $= E$. We also need to increment `move_to` and `move_from`. This gives us the following diagram:



8. Since `move_from` $= H$ is marked, we set its metadata to `move_to` $= F$. We also need to increment `move_to` and `move_from`. This gives us the following diagram:



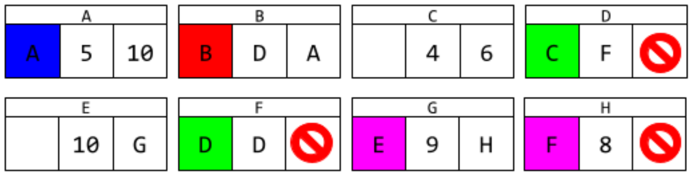
At this point, we're done. So, we know where each memory location should be moved to after compactness so that the heap is contiguous. *However*, we still need to update all the internal references within the heap!

1.3.4 Step 4: Forward Internal Addresses

Now that we've marked where everything should be moved to so we can maintain a contiguous heap structure, we still need to update all internal references so they point to the right places. For this, we'll use `fwd_internal`. The idea is that, for each block of memory, we want to consider each reference that the block has, if any. For each reference r:

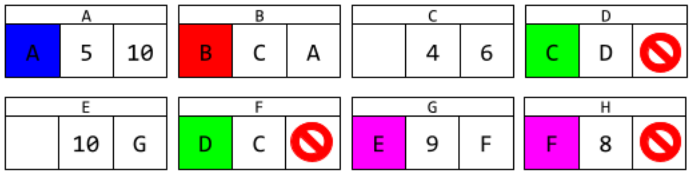
- Access the original memory block (before compating) at r.
- Get its forwarding address and use that as the new reference.

At the moment, this is what our memory diagram looks like:



1. A has no internal references; there's no changes that need to be made.
2. One of B's references need to change.
   - Consider reference D. Looking up reference D, we see that its forwarding address is C. Therefore, we replace D with C.
   - Consider reference A. Looking up reference A, we see that its forwarding address is A. Therefore, we can keep this as A.
3. C is garbage, so we don't need to consider it.
4. For D, one of its references need to change.
   - Consider reference F. Looking up reference F, we see that its forwarding address is D. Therefore, we replace F with D here.
   - `nil` is kept the same.
5. E is garbage, so we don't need to consider it.
6. One of F's references need to change.
   - Consider reference D. Looking it up, we see that its forwarding address is C. So, we replace D with C.
   - `nil` is kept the same.
7. One of G's references need to change.
   - 9 is a number, so we leave it alone.
   - Consider reference H. Looking it up, we see that its forwarding address is F. So, we replace H with F.
8. H has no internal references, so no changes needed.

The result of changing the internal references is



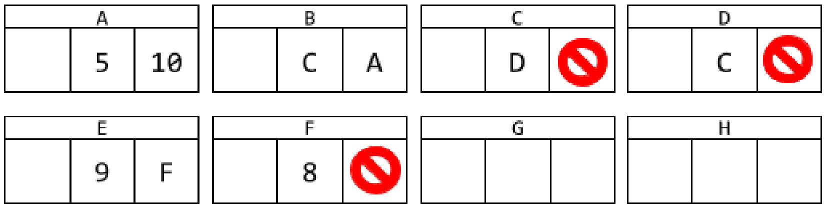
1.3.5 Step 5: Compacting the Heap

Finally, we want to compact the heap. Compacting will be *iteration* (like `fwd_headers`), **not** traversal (like `mark`). In particular, iteration involves iterating over each live data and copying it to its forwarding pointer.

In the compacting step, we want to

- copy each live data to the forwarding pointer, and
- unmark each data.

This gives us


A
5
10
B
C
A
C
D
D
C
E
9
F
F
8
G
H