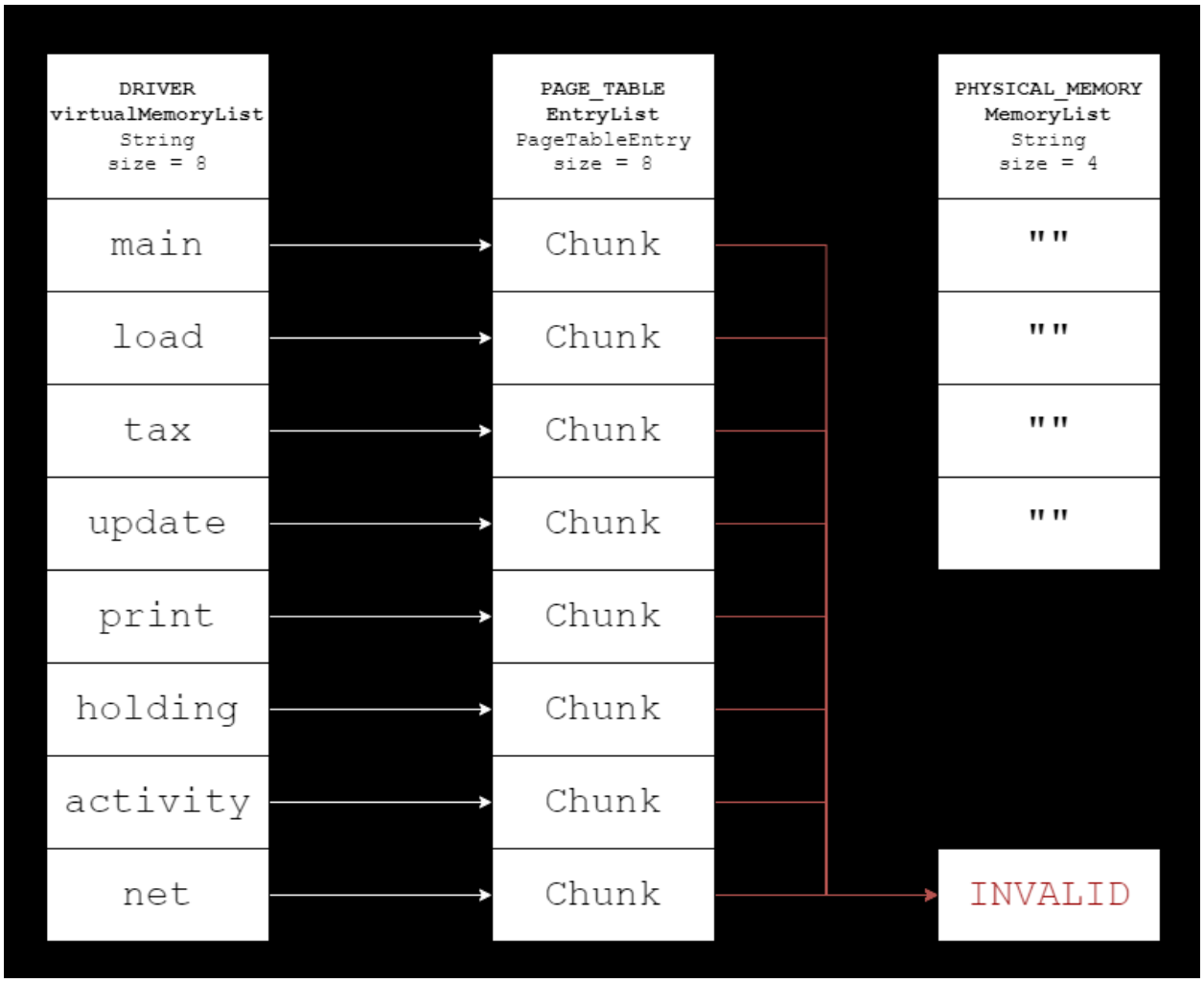
Please read the assignment write up and the programming hints before reading this document!

At the beginning of your program (After everything has been initialized but BEFORE the driver starts running down its *runList*), your objects will look like so:

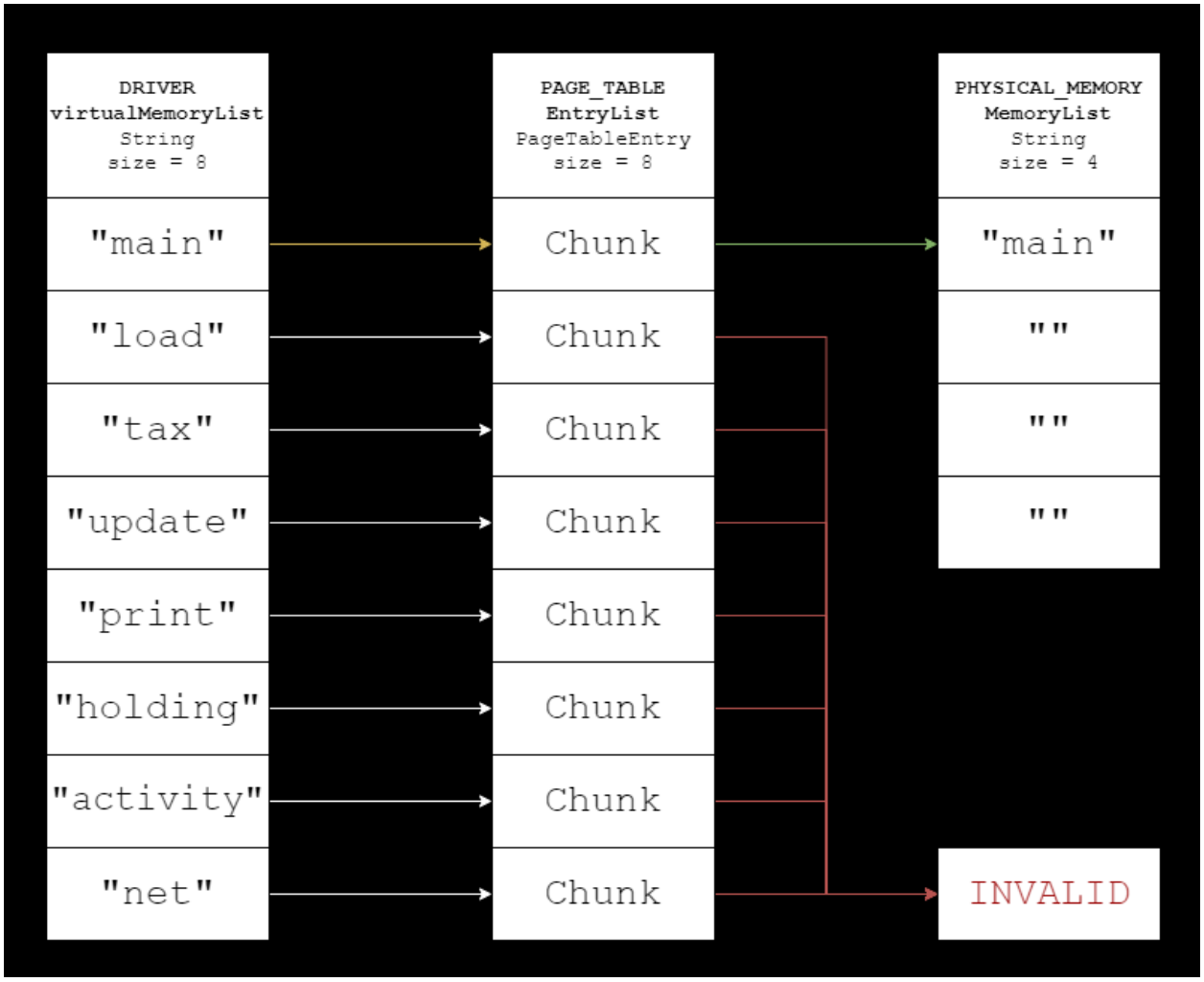


- Each item in *virtualMemoryList* corresponds to a chunk in *EntryList*. Each of those chunks start off *invalid*.
- The *memoryList* contains only empty strings.

The Driver's first item from the *runList* is *"main"*. It will attempt to reference page 0 in the *entryList*. However, since that chunk is invalid (as it should

be when initialized), a **page fault** occurs, and *"main"* will need to be stored in the *memoryList*.

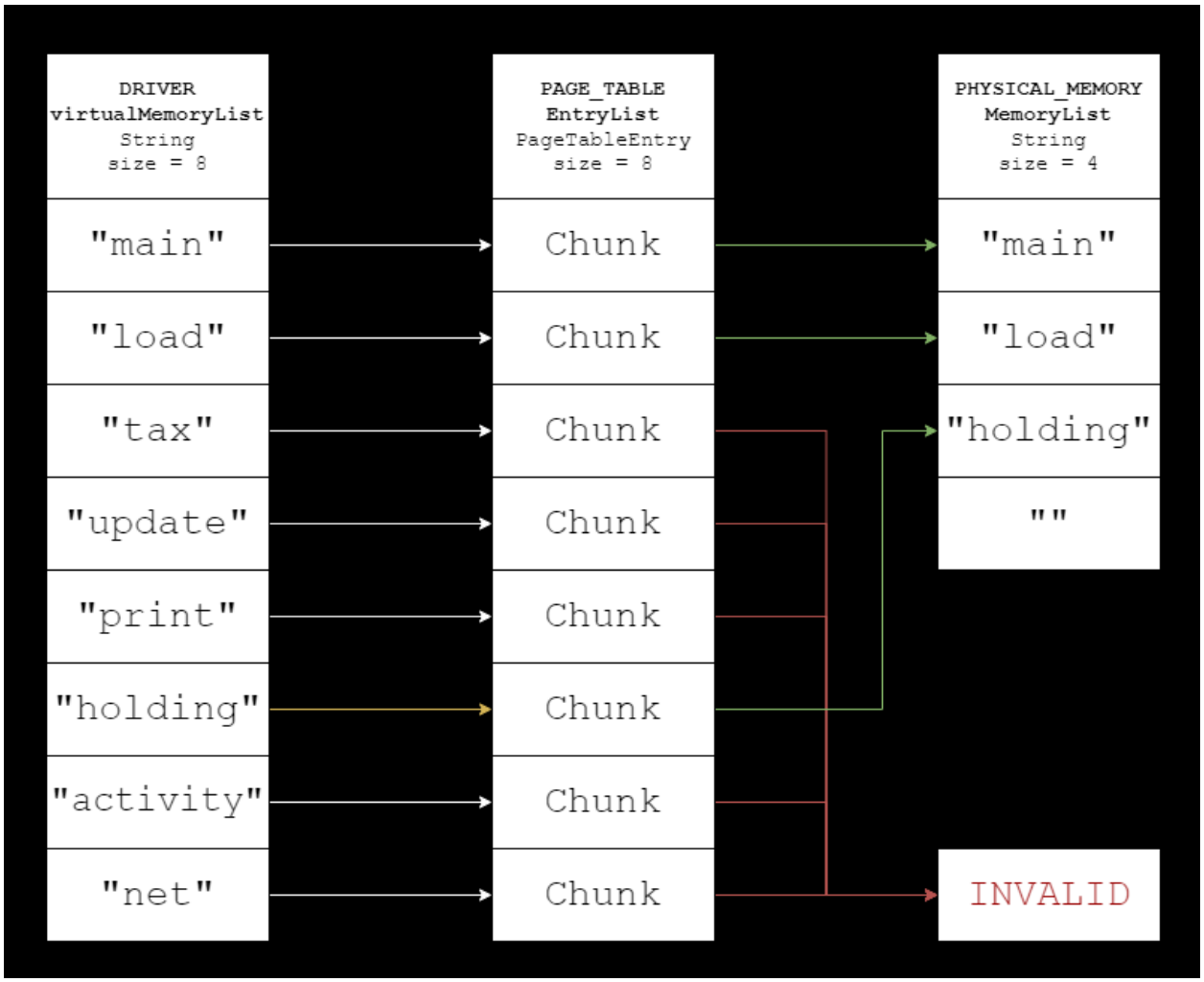
Since the *memoryList* is not full yet, the replacement algorithm in the *physicalMemory* class should just pick the next empty slot to assign a page to (frame 0):



- *memoryList[0]* is filled with *"main"*.
- *entryList[0]*'s *physicalMemoryIndex* now points at index 0 in the *memoryList*. Since the *physicalMemoryIndex* now correctly points to a frame containing the chunk's corresponding *item "main"*, it is now valid.

We're all set for the next run, which is "load". Similarly to the last reference, the corresponding chunk is invalid, so we'll need to do a page fault, find the next *memoryList* index to load to (1, since it is still not full), and update the Chunk to point there.

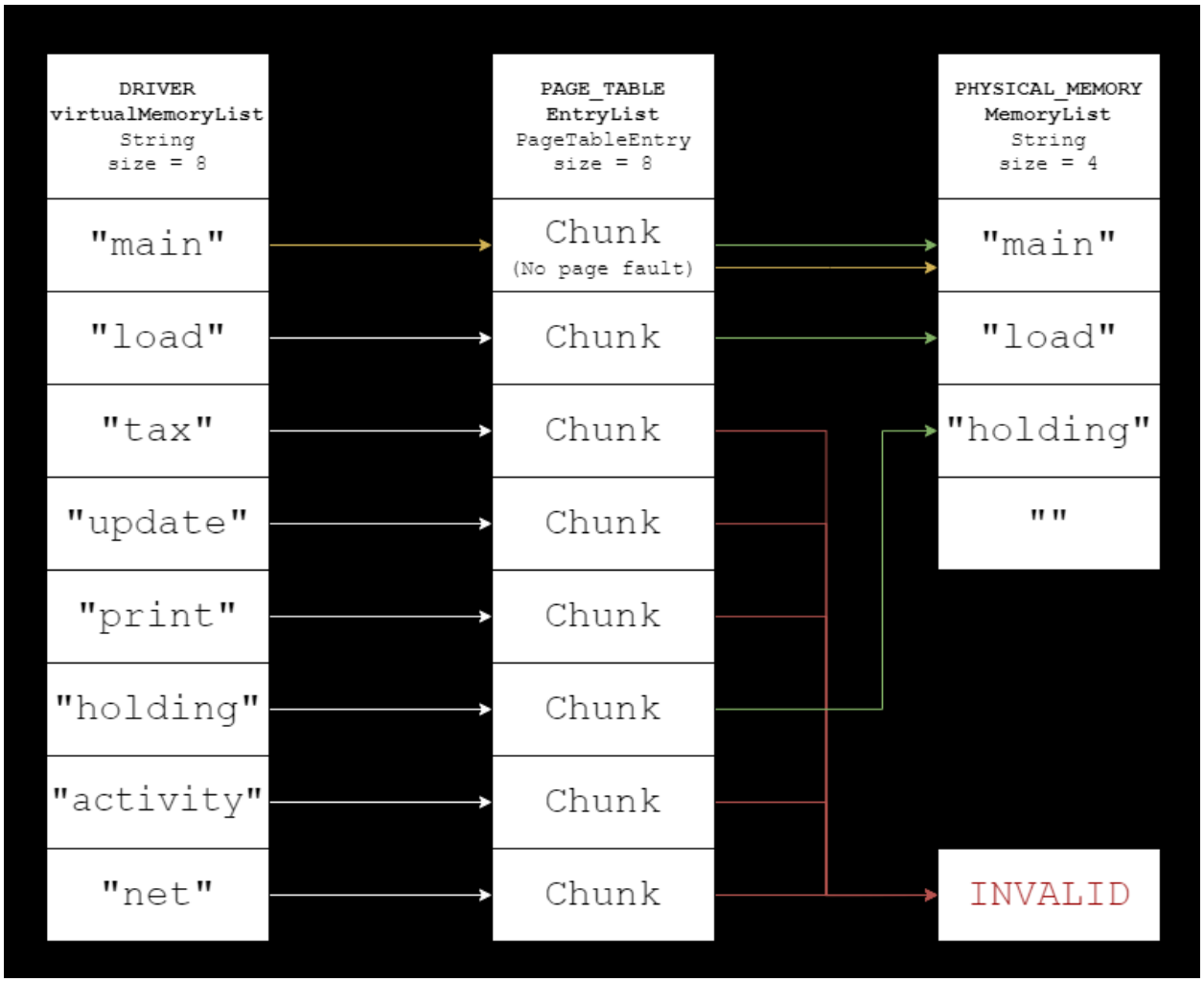
Since the exact same thing will happen for the next run (“holding”), we’ll skip to that state:



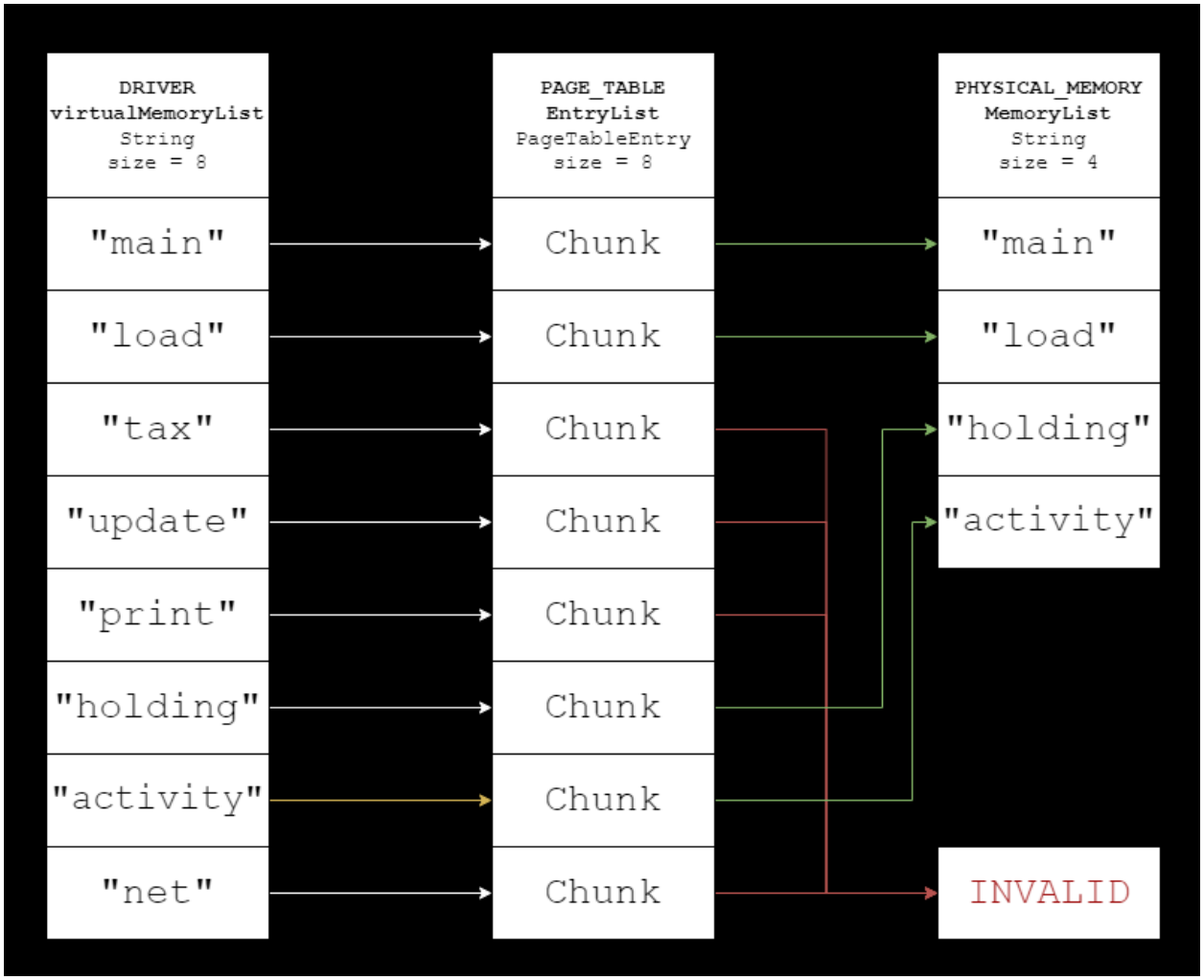
- Note that *entryList[5]* **points** to *memoryList[2]*!

Our next run is actually *“main”* again. However, since the chunk corresponding with *“main”* **IS** valid, we can just access the physicalMemoryIndex

it points to immediately and continue to the next run!



Next up is *"activity"*. The corresponding chunk for it is invalid, so we'll get a page fault again, but we've still got one empty slot, so just fill it like the last three times:

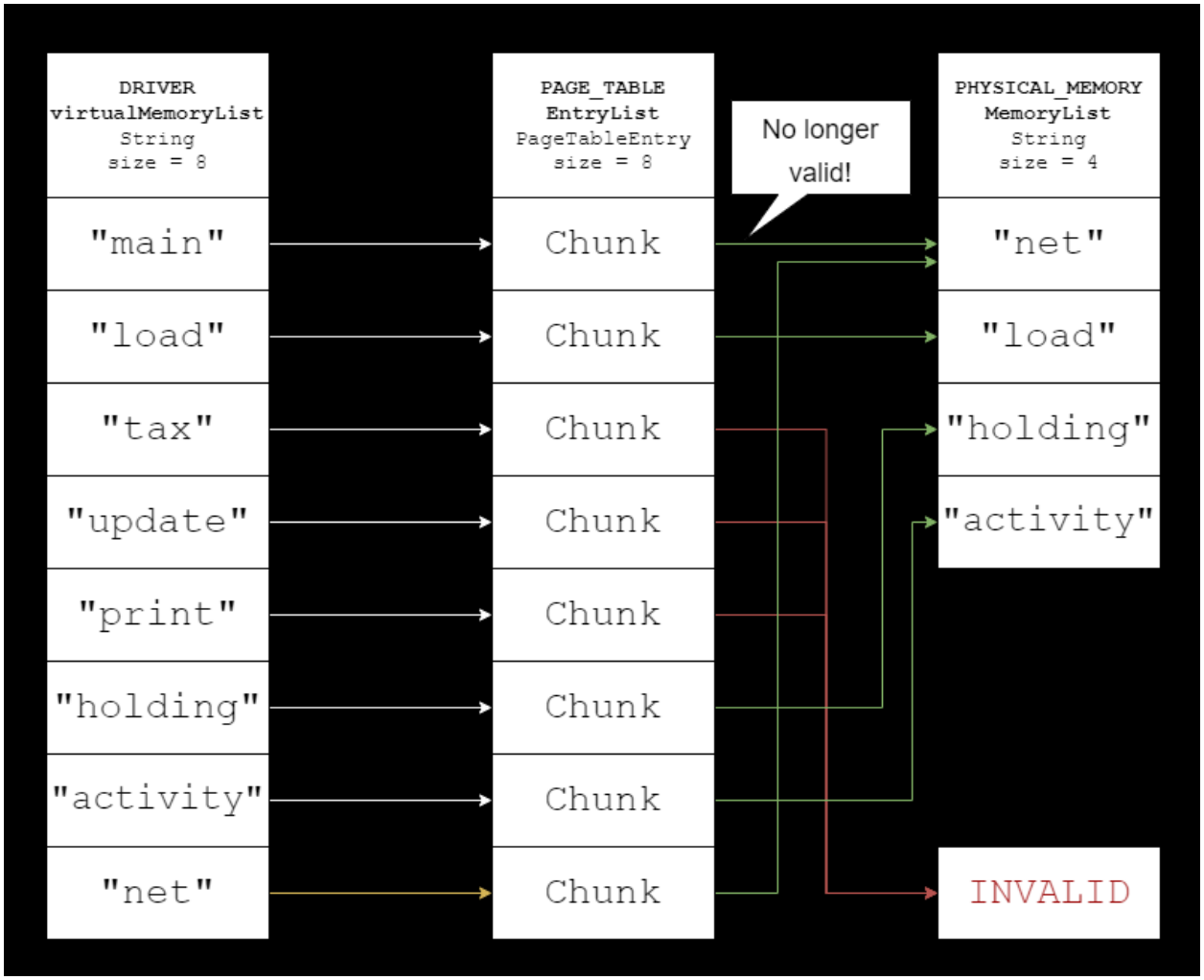


- Our *memoryList* is now **full**. We'll need to do page replacement if any new *items* are needed!

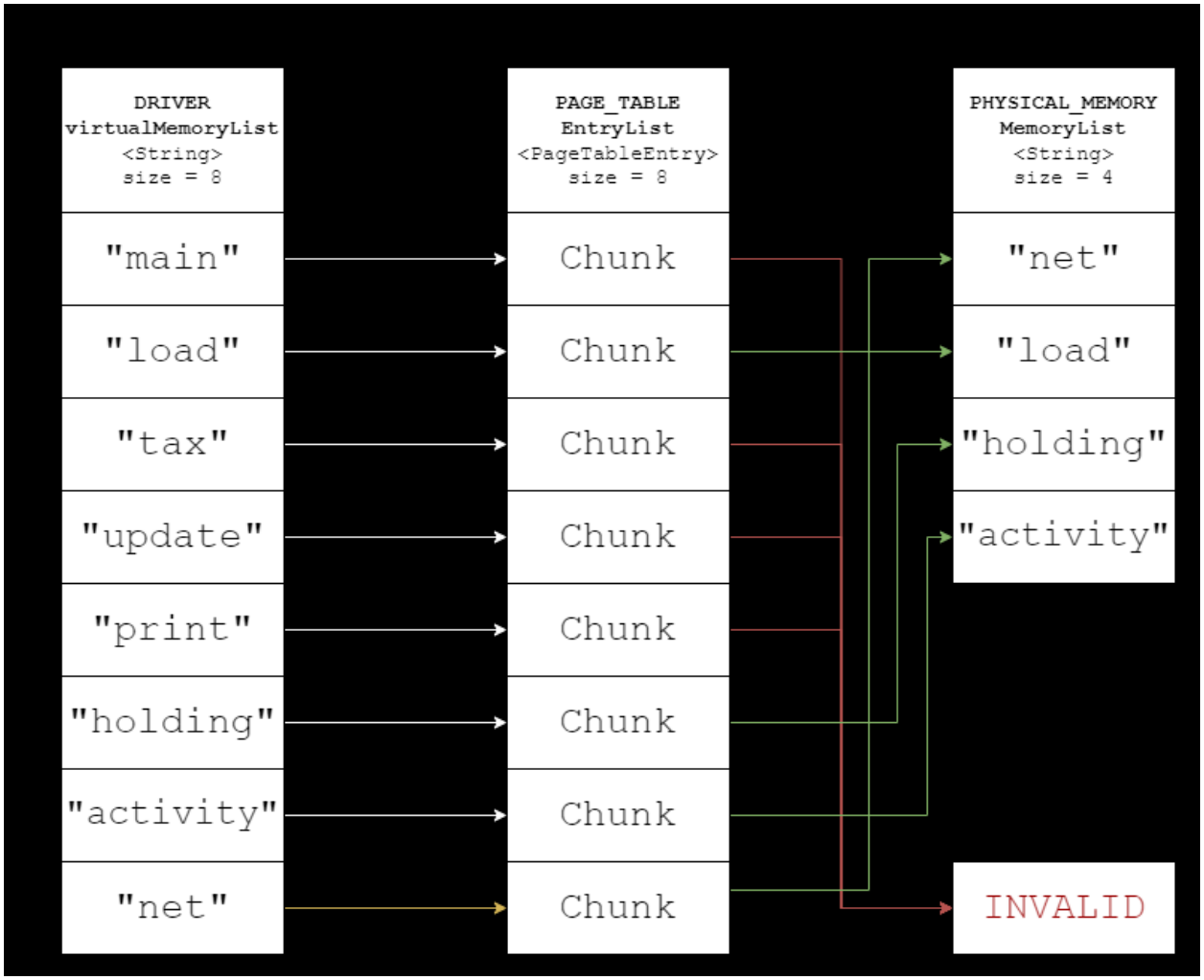
Now we come to our first page replacement, since the next item to run is *"net"*. *"net"* is currently invalid AND our *memoryList* is full, so when the page fault occurs, we are going to have to sacrifice one of the *items* in *memoryList* and give its spot to the incoming *item*, *"net"*.

Our replacement algorithm tells us which index in *memoryList* will be replaced (known as the 'victim'). FIFO makes this decision easy- the first index filled to will be the first to be replaced: index 0. Therefore, *"net"* will get stored there and the corresponding chunk will point to index 0.

However, we have a problem:



- *entryList[0]*'s *physicalMemoryIndex* is **NOT** correct anymore! Since it's corresponding *item* no longer exists in the *memoryList* we must **invalidate** it!



- e*ntryList[0]*, the chunk that was pointing to the replaced index, is invalidated, so our diagram is correct now.

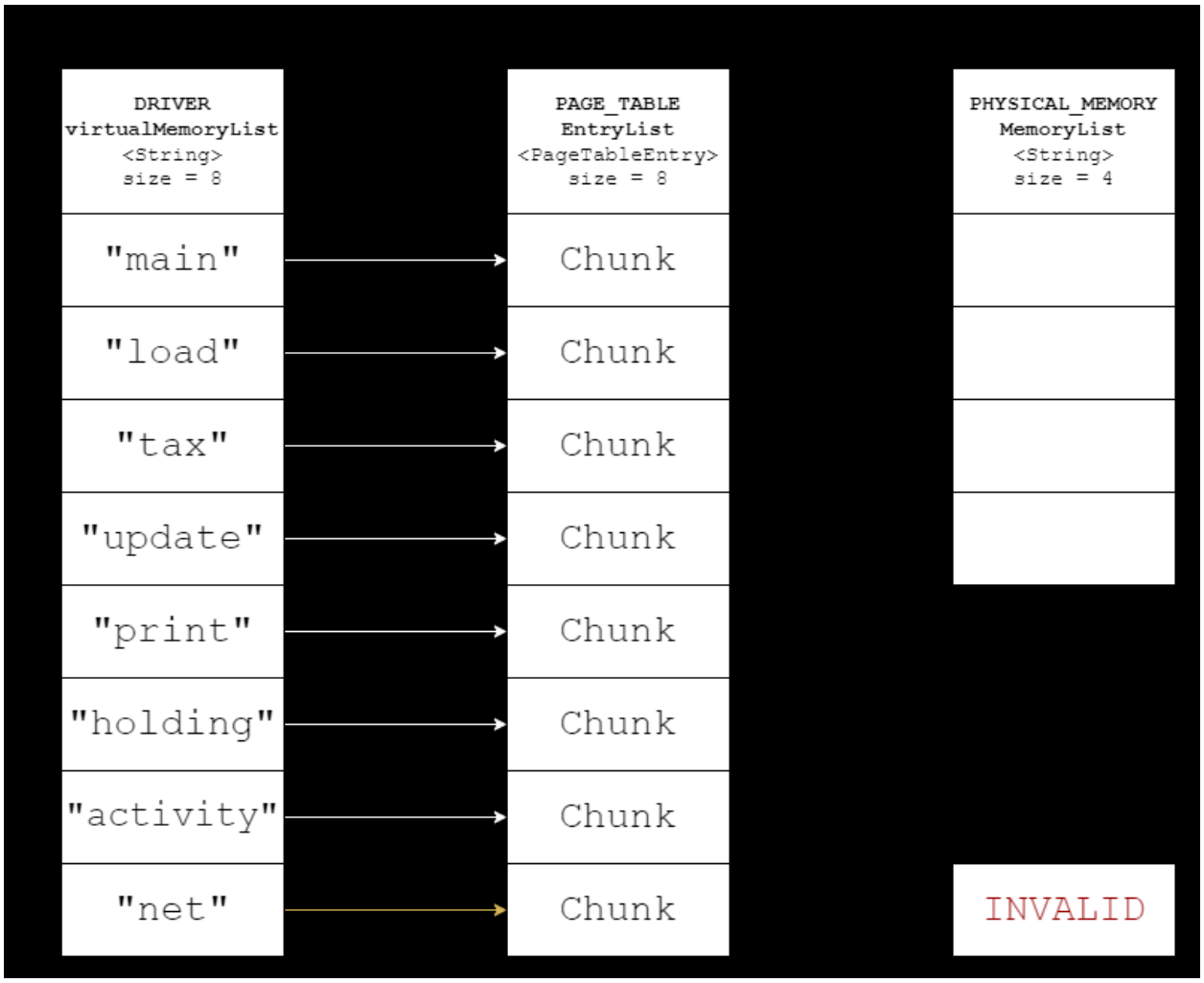
When an *item* in the *memoryList* gets replaced, we need to ensure that the chunk that was pointing to that *memoryList* index is invalidated (Consider this: What would happen if we **forgot** to invalidate replaced indices?).

Need Practice?

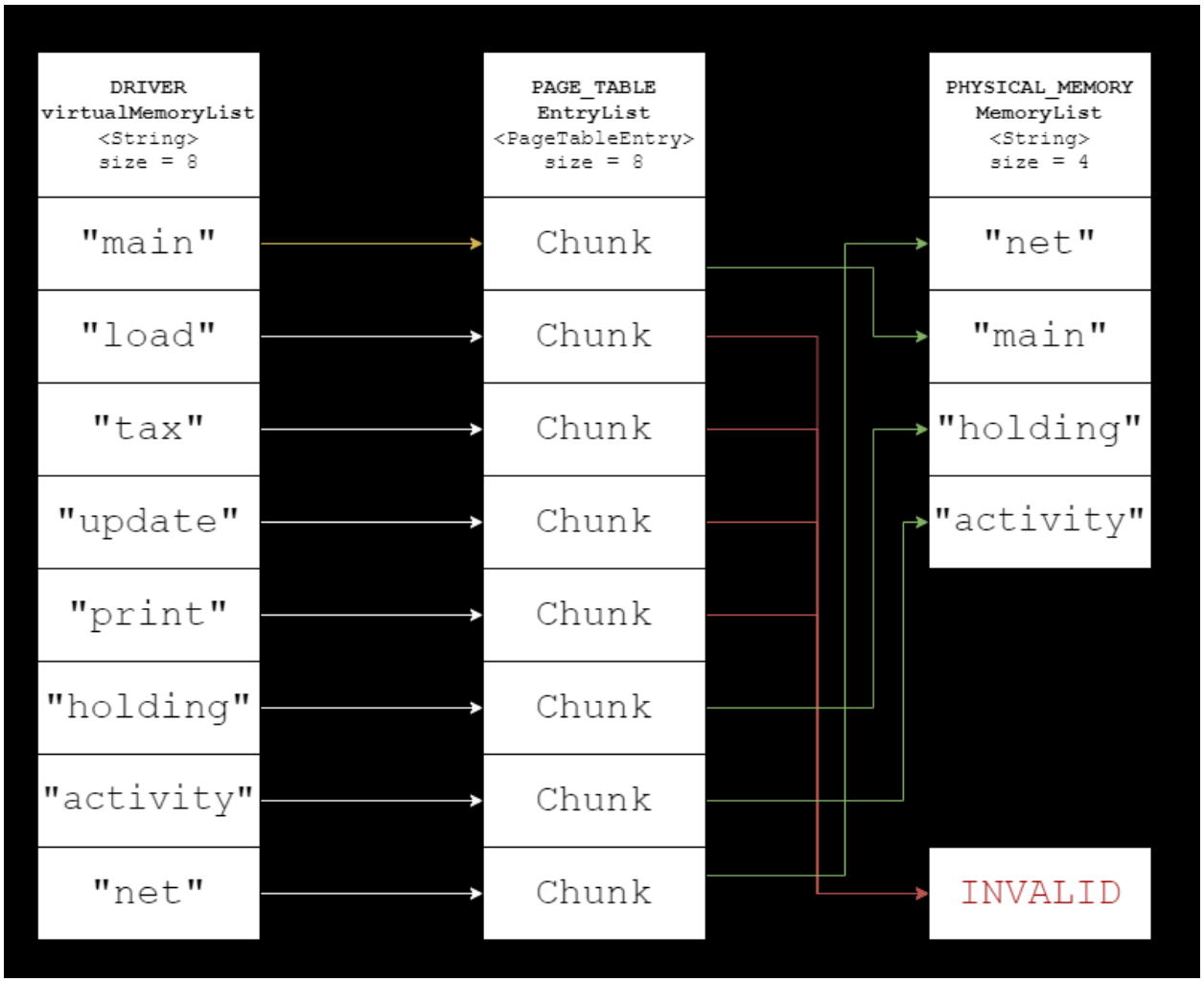
The next two items to run are “main” again (as we know, one of the weaknesses of FIFO is that commonly used data can wind up getting pulled out only to be needed again shortly after!), with a new *item “update”* coming right after it. Below is an incomplete version of this diagram, if you would like practice,

draw in the connections the Chunks will have after another *"main"* and an *"update"* call are made, similarly to the diagrams above.

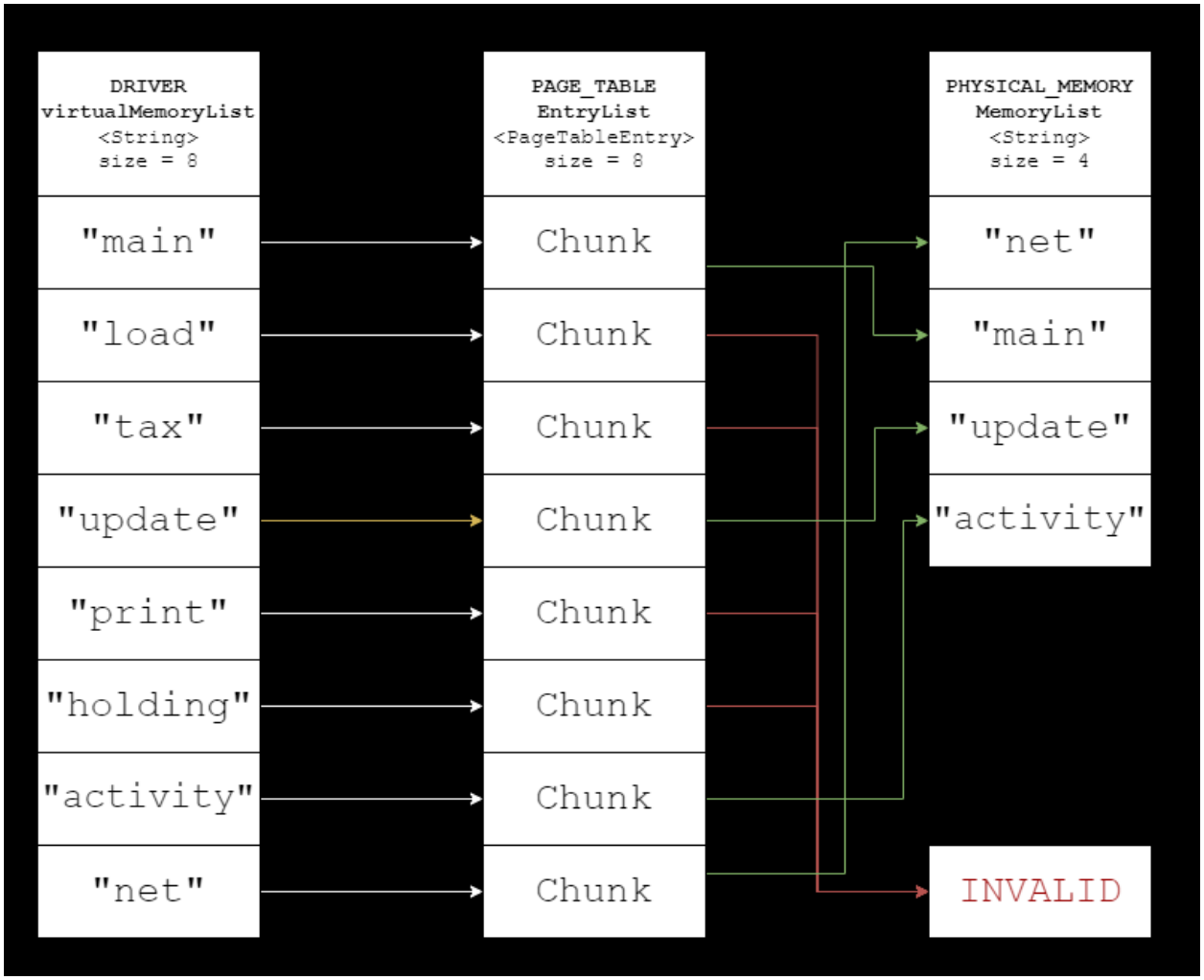
The correct results will be separated by a blank page.



"main" is ran:



"update" is ran:



Implementation Notes

How you invalidate a chunk / PageTableEntry is up to you. The programming hints state you can use a bool in the struct to describe validity (valid = true), or you could just set physicalMemoryIndex to a sensible number value to indicate when it is invalid (You could use -1, which has the added "benefit" of causing an out of range exception if your program accidentally tries to access invalid indices, which will certainly draw your attention to errors!)

Remember, FIFO is going to choose the replacement indices in the order they were first assigned to. In this case, it is a simple, repeating 0, 1, 2, 3 pattern.

Consider how you might implement this in a program, what operators can mimic this pattern?