Chapter 1

Garbage Collection

It's called a Garbage Can, not a Garbage Cannot

1.1 Introduction

When we are implementing a language, there may be times when we need more information than just what is given to us and our operational semantics. We saw this when we added variables to our language: we needed an environment to keep track of variables and what they were bound to. This environemnt acted as our memory, allowing us to make a variable and store a value with it. When we were done with the scope of the variable, then the binding was automatically dropped (thanks to the immutability of OCaml). This automatic memory management is analogous to the stack, where items are pushed and popped automatically, as opposed to the heap, where either the user has to management their own memory (C) or a garbage collector will manage the allocation and deallocation (OCaml, Java, etc).

Since we will be good people, we will not push the burden of memory management to the user. We will make the garbage collector. To do so, we need to consider what makes a good garbage collector.

A good garbage collector must take a conservative approach to deallocating memory. Consider:

Memory	Don't free	Free
In use	Good	Really Bad
Not in use	Eh	Good

Notice that we want to free memory that is no longer in use, and not free memory that is in use. If we have memory that we don't use and do not free it, all that really happens is we lose useable memory. This could mean we run out of memory, or things take longer to lookup in memory (decreasing time performance). Ultimately, the program will not be *wrong* per se, rather it will just be unoptimized.

Alternatively, we could have a garbage collector that frees things that are in use. This can cause the program to fail and potentially have dangerous side effects[1].

Determining if something is in use or not to be freed is the goal of garbage collection. However, while some things are easy to figure out if they are in use or not, some things are more ambiguous. To be a conservative garbage collector means to err on the side of caution, and only free things that we are guaranteed to be no longer in use. So how do we know if something is in use or not?

We will see three basic ideas to determining the answer to this question. However, it is important to note that modern garbage collector typically will use a modified or combination of the following ideas.

[1] Technically not freeing things that need to can also lead to security vulnerabilities too

1.2 Reference Counting

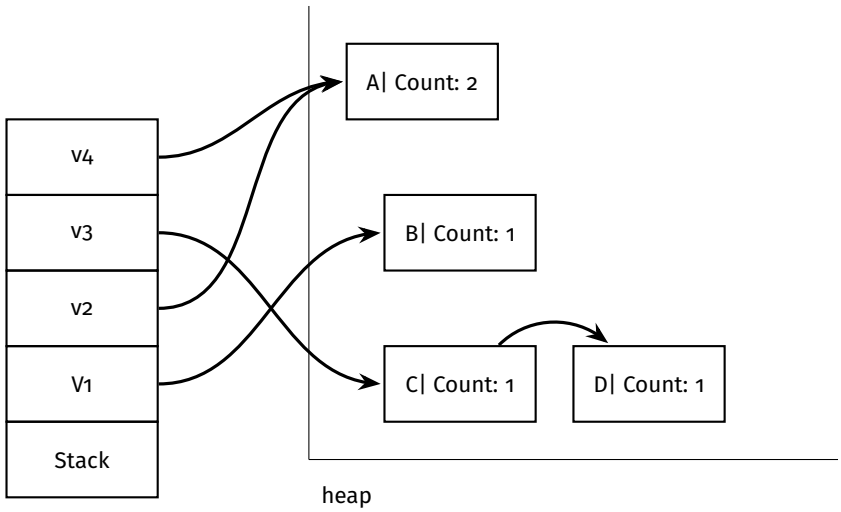
We say that a piece of memory is in use if we can reach that piece of data. If we loose a reference to a segment of memory, then we can't really use what's stored there so we can free it. One idea is to keep track of how many pointers (references) point to a segment of memory, and deallocate (free) when that counter reaches 0. Consider:

```
1 {
2   int* v1 = malloc(sizeof(int)); //say memory address 0xfff
3   // one thing points to 0xfff
4   {
5   int* v2 = v1; //now 2 things point to 0xfff
6   }
7   //now 1 thing points to 0xfff
8 }
9 //now nothing points to 0xfff
```

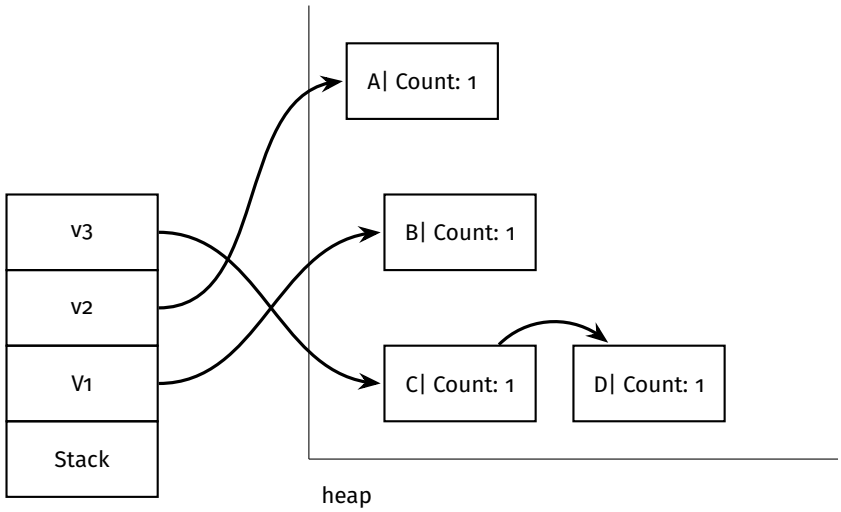
Recall that variables are valid until the end of their scope, and you can create a temporary scope with curly braces({}). Thus, `v1` is valid until line 6, while `v2` is valid until line 5. If we consider how many things are pointing to memory address `0xfff` at each line, we can see that the count increases when we allocate, or set a pointer to a variable. The count then decrements when the pointer goes out of scope.

It is important to note that the counter is incremented everytime a pointer to that segment of memory if updated (added or deleted). While this is a constant time operation, this typically ends up being called quite a lot of times. Additionally, space for the counter needs to be allocated with each item on the heap so there is some added space complexity to consider.

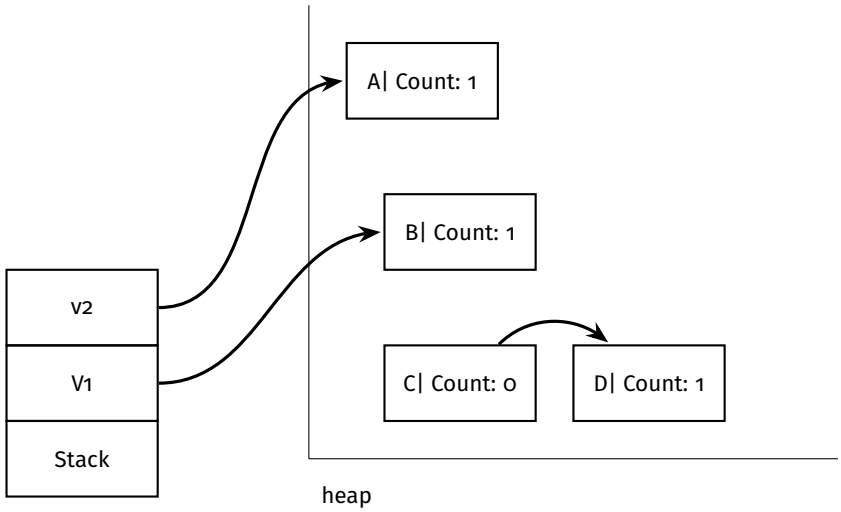
Consider the memory map:



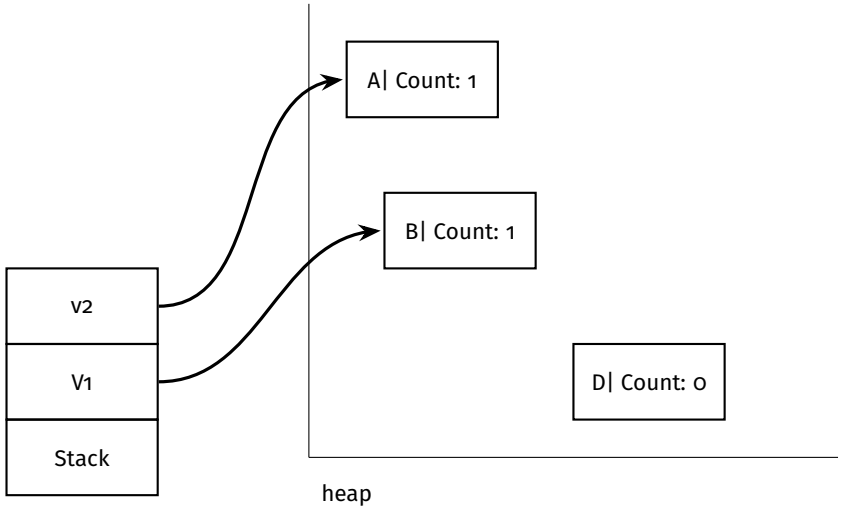
If v4 went out of scope, then the reference count of item A would be decremented to 2.



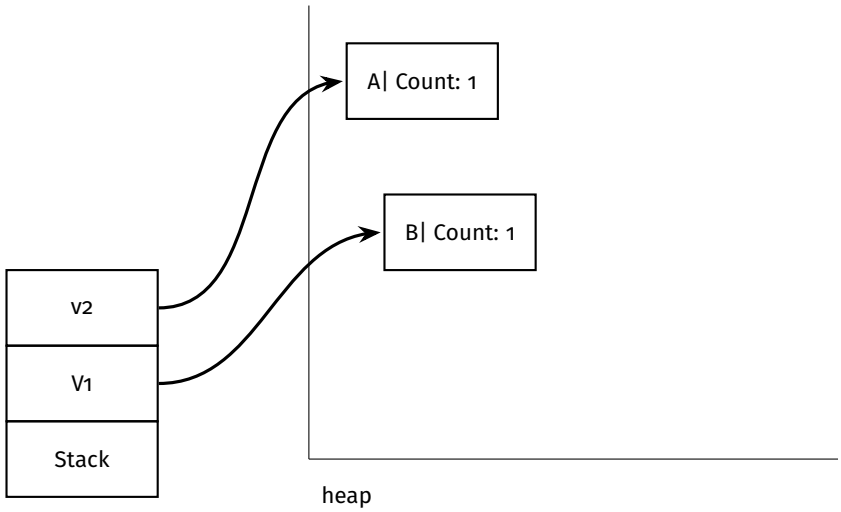
If v3 was popped off, then C would be at count 0.



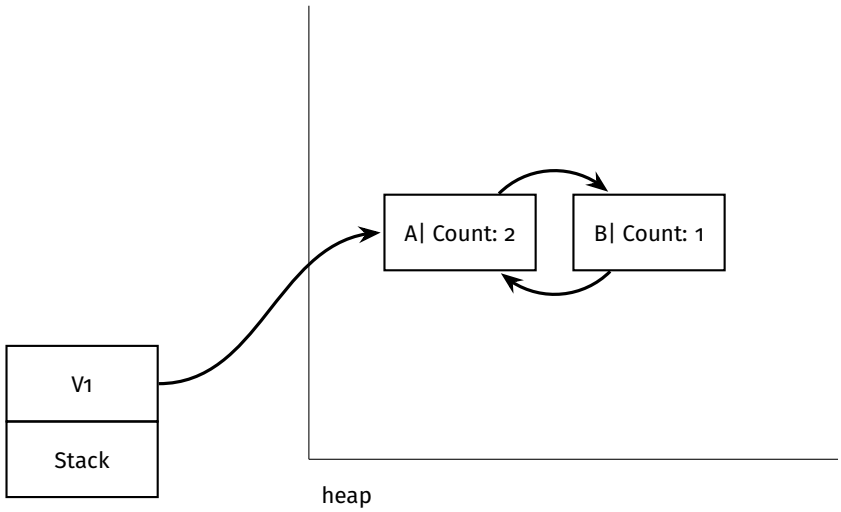
Since the count of C hits zero, then we free that item.



Now since the freeing of C caused the counter of D to decrement to zero, then D has to be freed as well.



One issue to consider is the idea of cyclic data. What happens when v1 is popped off the stack in the following memory diagram?



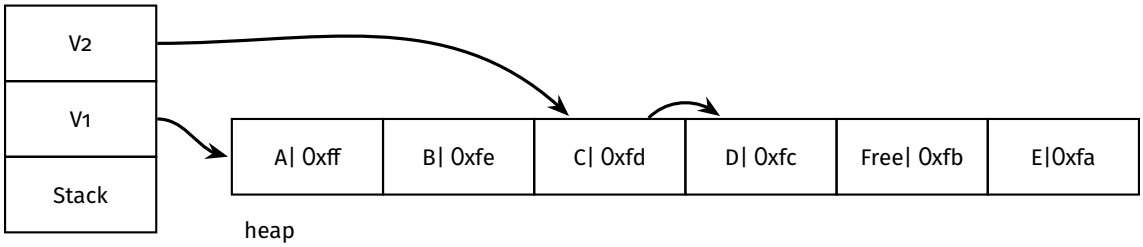
1.3 Mark and Sweep

Mark and Sweep is the next garbage collection strategy we will talk about, and it is a form of tracing garbage collection. There are various variations of Mark and Sweep and we will talk about one of the most basic forms.

In Mark and sweep we consider what is reachable based off what you can get to via the stack. However, we also need to go through and actually free everything that should be freed. In order to do so, we need to go through the entire heap, and figure out if what we are looking at is reachable from the stack or not[2]

The heap is just a linear piece of memory so let's restructure the picture of the heap:

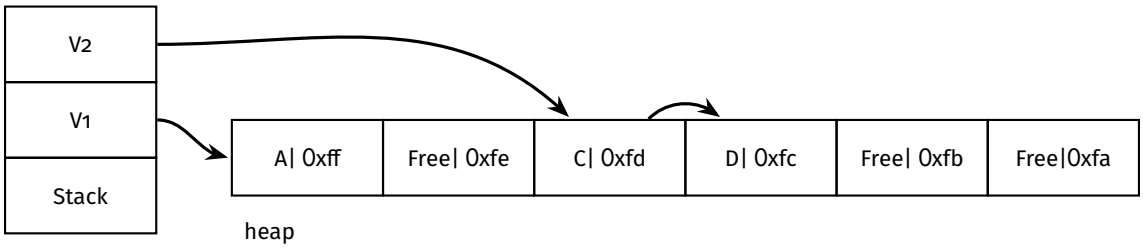
[2] some implementations have you go through stack then heap in O(n+m) fashion. Here we will be doing O(m*n).



In order to figure out what is in use and what is not in use, we can iterate through the entire heap, figure out if we can reach where we are looking via the stack.

- We look at `0xff` and we check if anything on the stack can reach it or it is already free.
- We look at `0xfe` and we check if anything on the stack can reach it or it is already free.
- We look at `0xfd` and we check if anything on the stack can reach it or it is already free.
- We look at `0xfc` and we check if anything on the stack can reach it or it is already free.
- We look at `0xfb` and we check if anything on the stack can reach it or it is already free.
- We look at `0xfa` and we check if anything on the stack can reach it or it is already free.

After we do all of this, the only places of memory that fail this check are `0xfe` and `0xfa`. We then know we can free these two places in memory.



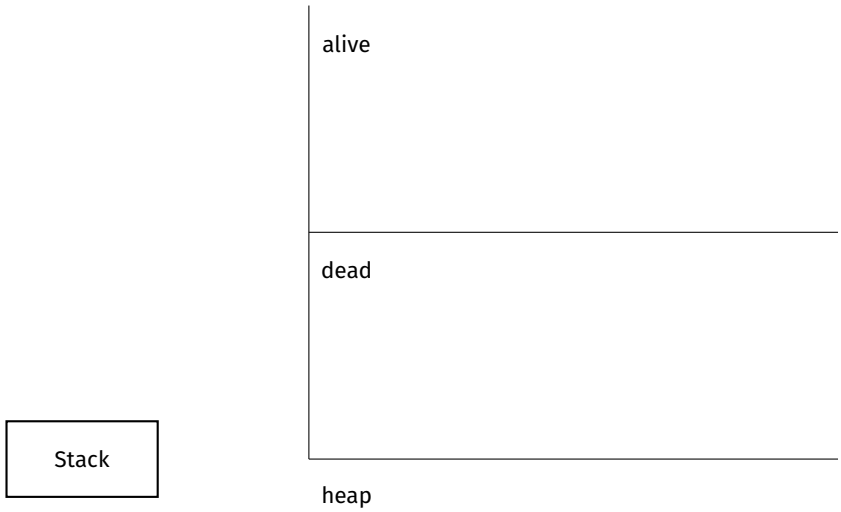
Note: `0xfc` is reachable from the stack, just not directly.

One thing to note is that the program must be paused while this is happening since we do not want things to be allocated or freed while this is occurring.

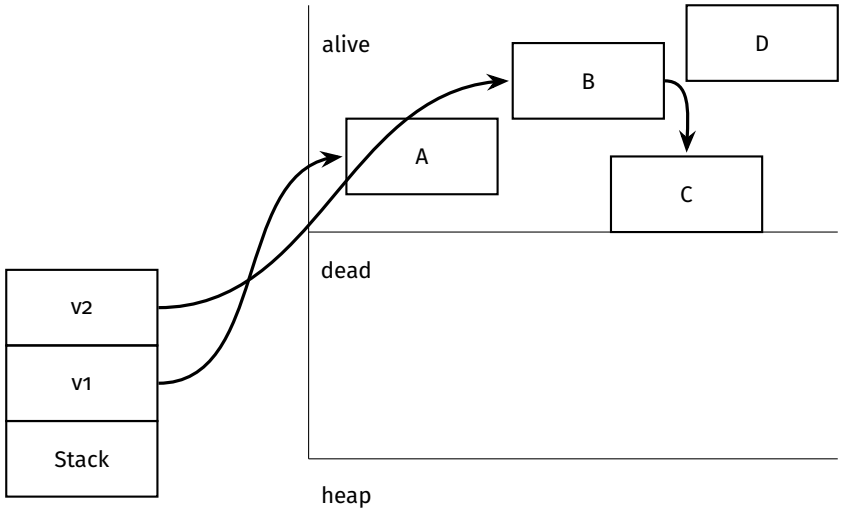
1.4 Stop and Copy

In the same vein of Mark and Sweep, have another tracing garb age collector: stop and Copy. Much like Mark and Sweep, the program must stop while this is occurring.

In Stop and Copy, we must first partition the Heap into an alive partition and a dead partition.

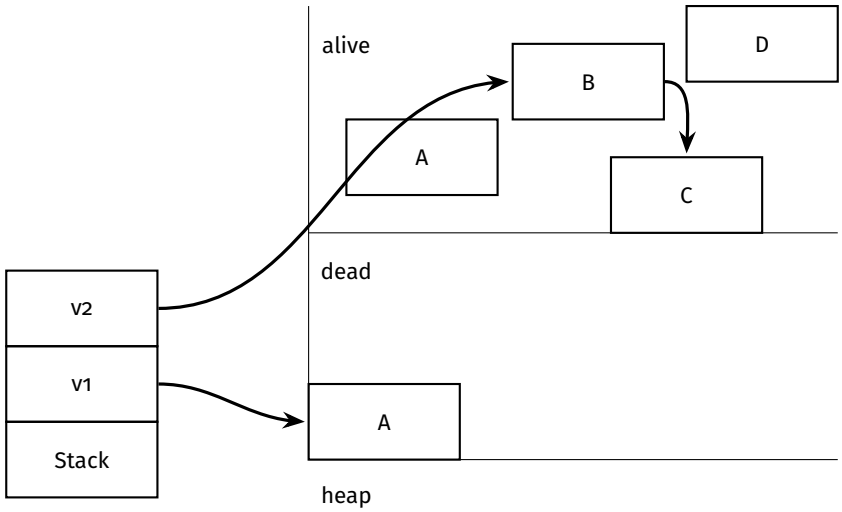


Then whenever we need to allocate something, we do so in the alive part.

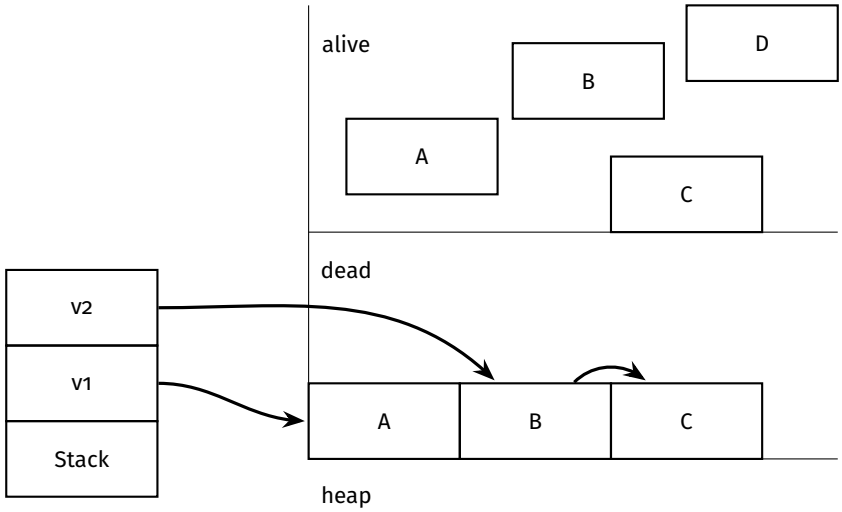


Then, when running garbage collection, we go through the entire stack, and copy over everything that is reachable to the dead partition.

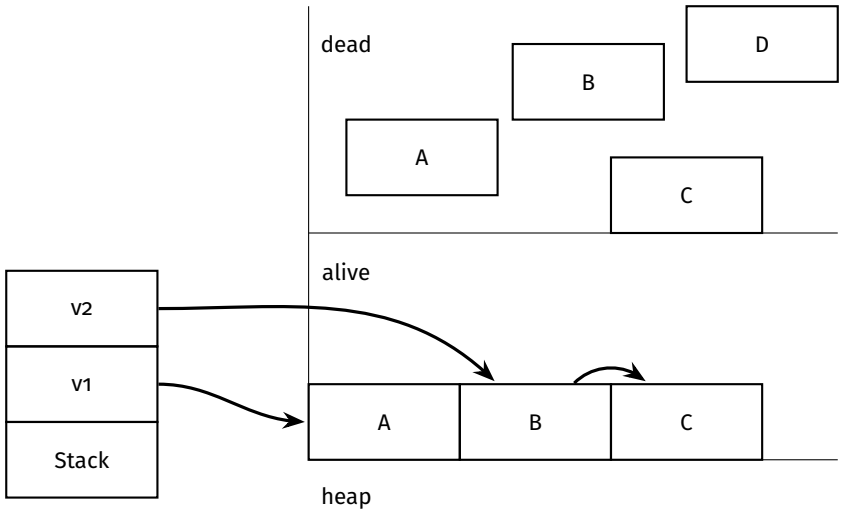
So first we look at say, v1 and copy whatever it points to to the dead partition.



We then do so for all other items on the stack:



Then we swap partitions and continue.



We will continue to allocate in the alive section only, and when garbage collection happens again, we will copy every in-use memory over and swap partitions again. Due to the fact that this will copy over data, we can actually defragment our memory to optimize later memory allocation. This is important since we start off by halving our useable memory space to begin with.

Notice that when this happens again, we will just copy right over our past memory usage.