CS 24: Introduction to Computing Systems

Garbage Collection

Previously, we've only considered *explicit* dynamic memory allocators that require the user to deallocate memory. We now consider *implicit* dynamic memory allocators which are responsible for automatically reclaiming memory for the user.

From Pointers to References

In languages like Java and Python, pointers are wrapped in **references**. A **reference** is like a pointer (in that it refers to a particular value), but it cannot be changed directly. The language runtime is responsible for mapping between references and pointers for the user. Consider the following short program in a Python-like language:

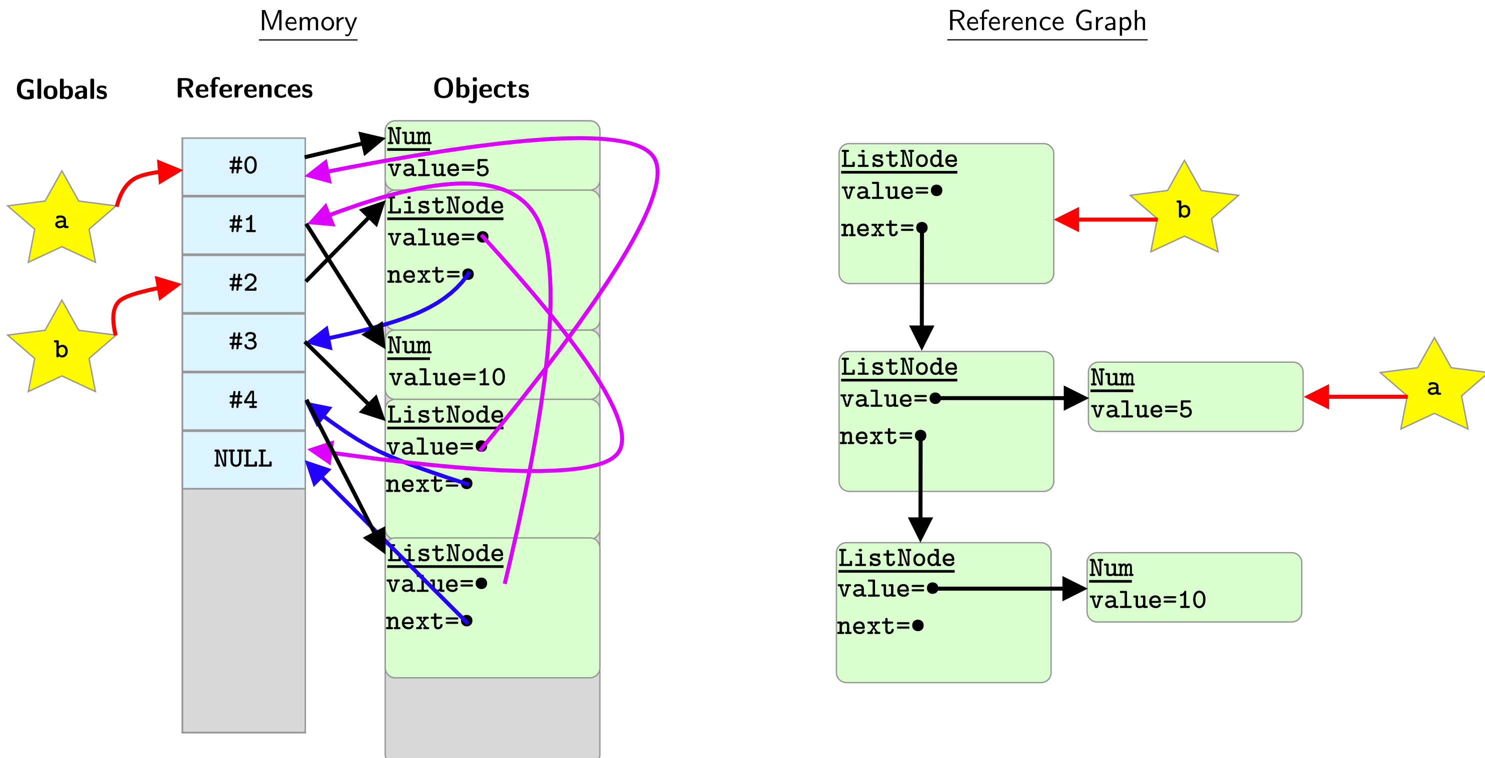
Example 1

```
1  a = 5
2  b = [a, 10]
```

This snippet has two **global variables** and five **values**. In a language with references, the full program state after line 2 might look like the following:

- References: #0, #1, #2, #3, #4
- Values:
  - $\#0 \mapsto \text{Num}(5)$
  - $\#1 \mapsto \text{Num}(10)$
  - $\#2 \mapsto \text{ListNode}(\texttt{value} = \texttt{NULL}, \texttt{next} = \#3)$
  - $\#3 \mapsto \text{ListNode}(\texttt{value} = \#0, \texttt{next} = \#4)$
  - $\#4 \mapsto \text{ListNode}(\texttt{value} = \#1, \texttt{next} = \texttt{NULL})$
- Globals: a is #0, b is #2

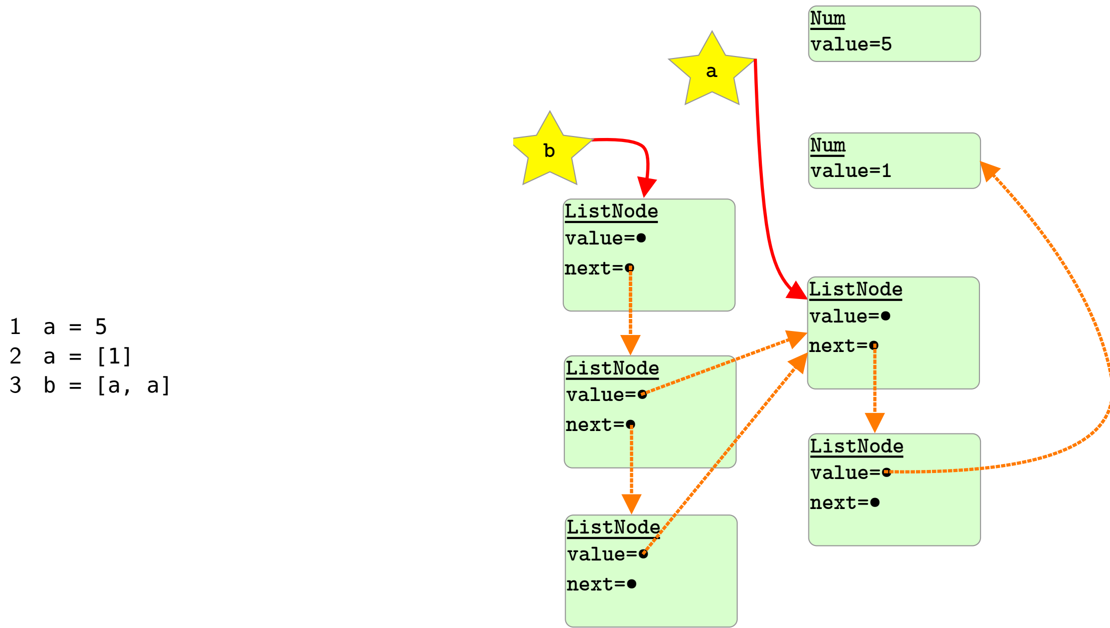
It can help to visualize this state by looking at a *memory-centric* diagram or a *reference graph* diagram:



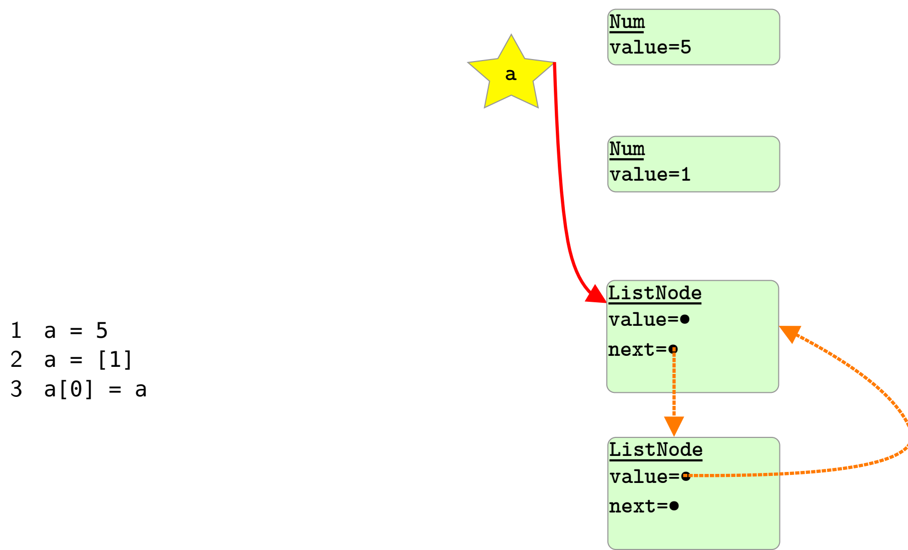
The memory-centric diagram, although useful, can get very cluttered; so, we primarily view the garbage collection problem via the reference graph. In particular, a value is *live* (i.e., not garbage) if and only if it is reachable from one of the root nodes (i.e., global variables).

You can check your understanding of the *reference graph* using the two following examples.

Example 2



Example 3



Reference Counting

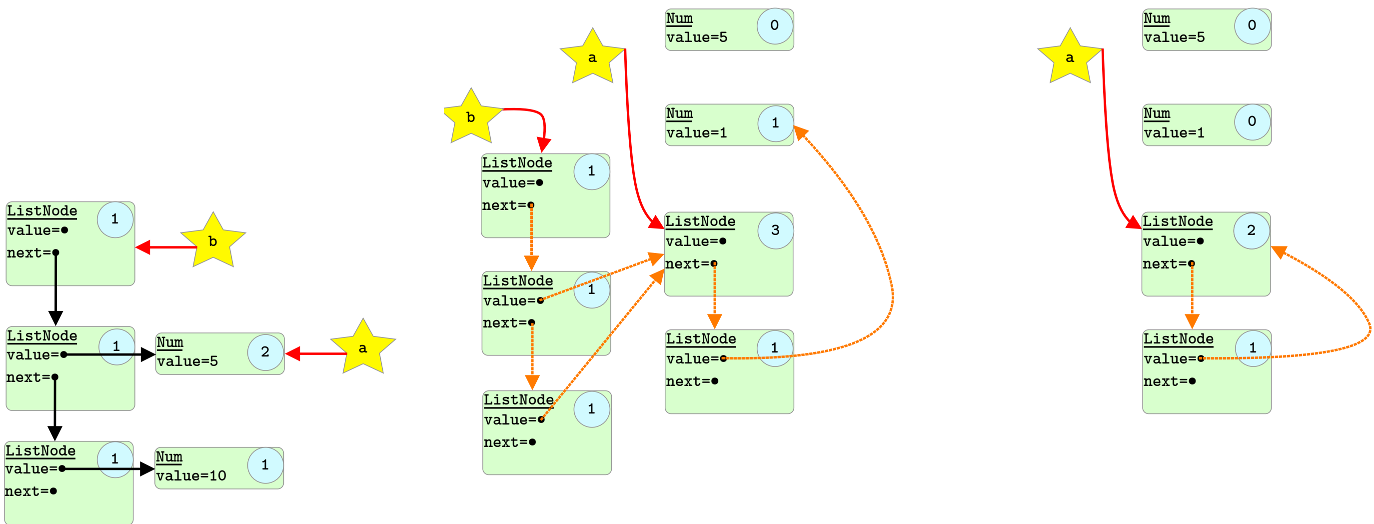
One algorithm to implement garbage collection is called "reference counting". The idea is that every value has an associated "reference count" which keeps track of the number of references to it. These counts are updated every time a new reference is created or an old reference is destroyed. Consider the following statements and expressions:

- **Assignment Statement** (e.g., `a = b`). If `a` previously pointed to something, then the reference count on that old value is decreased. Then, the reference count on the new value is increased.

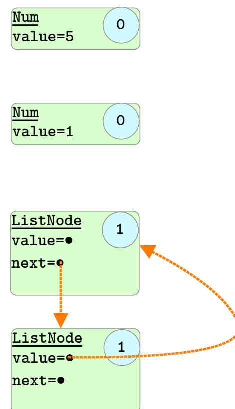
- **(Linked) List Constructor Expressions** (e.g., `[1, 2, 3]`). We encode our linked lists with a dummy node at the beginning with a value of `NULL`. Then, every node in the list except the dummy one has a reference to the next node in the list; so, they all increase their reference counts by one.

When a reference count reaches zero, that value is no longer used and is ready to be collected. Note that collecting one value might trigger another count to change, which might make more values to collect.

Consider the following three example graphs which correspond to our original three examples. The blue circles contain the value reference counts.



Notice that there is no garbage to collect in the first example, and definitely garbage to collect in the second and third examples. Now, imagine that we run "`del a`" on the third example. The resulting graph would be:



Since there are no global variables left, we should be able to reclaim everything! But we can't because there is a *cycle* in the reference graph. This is a major drawback of reference counting. There are several ways to fix this problem, but the easiest is to just rely on another algorithm to pick up the leftover garbage.

Tracing Garbage Collection

Unlike a reference counting garbage collector, a *tracing* garbage collector traverses the reference graph from the root using a bfs or dfs. To avoid program state changing while the algorithm is running, these algorithms tend to "stop the world" while running. Because this takes a long time, they can often create long pauses in the program execution which can be less than ideal.

Mark and Sweep

This time, we will traverse the reference graph to try to discover which nodes are unreachable. All we need to store is an "`is_reachable`" bit attached to each value. All of these bits will initially be set to zero (i.e., "we have not discovered them yet").

Mark and Sweep is made up of two phases, creatively called "mark" and "sweep". In the mark phase, the algorithm does a depth-first search from each root node (e.g., global variables), setting `is_reachable` to true for every value it finds. Then, in the sweep phase, the algorithm goes through all values and collects the ones that have `is_reachable` set to false (e.g., the values not reachable from the root nodes).

Here's some pseudocode:

```
1  def mark(value):
2     if value.is_reachable:
3        return
4     value.is_reachable = true
5     for child in value.children():
6        mark(child)
7
8  def sweep(pool):
9     for ref in pool.references():
10       value = dereference(ref)
11       if value.is_reachable:
12          value.is_reachable = false
13       else:
14          dereference(ref) = NULL
15
16 for value in pool.get_roots():
17    mark(value)
18 sweep(pool)
```

Optionally, Mark and Sweep can *compact* the pool which means moving all the values to contiguous memory at the beginning of the pool. In a compacting version, all the references must be updated to the new value locations. Notably, this is *exactly what we weren't allowed to do* in an explicit allocator! Because all pointers are accessed through references, we can now edit them and move the objects in memory!

Stop and Copy

Another strategy for garbage collection is to split the memory space into several sub-spaces with different characteristics. The simplest version of this kind of algorithm is called *stop and copy* or *semi-space* collector. In *stop and copy*, the memory space is split into two equally sized halves: the from-space and the to-space. Values are initially *only* allocated in the from-space until there is no memory in it left. Then, Stop and Copy runs and transfers all live values to the to-space. Because these two spaces are disjoint, all the remaining values in the from-space can be collected. After Stop and Copy runs, the from and to spaces are switched.

```
1  def move(value):
2     if value.visited:
3        return
4     value.visited = true
5     for child in value.children():
6        move(child)
7     move_value(value, to_space)
8
9  for value in pool.get_roots():
```

```
10     move(value)
11 for ref in pool.references():
12     value = dereference(reference)
13     if value in from_space:
14         dereference(ref) = NULL
```

Lifetimes and Performance

Both Mark and Sweep and Stop and Copy have to traverse the entire reference graph, but they differ in which types of values they focus on. Mark and Sweep works best when values last a long time, because the bulk of the work in the sweep phase is in collecting (and compacting) the garbage. In contrast, Stop and Copy works best when values are short-lived, because only live values are copied to the to-space.

Generational Garbage Collection

The *generational hypothesis* (which is generally accepted) says that (1) most values die quickly, and (2) those that don't tend to stay around for a very long time.

This observation leads to the idea of a "generational" garbage collector where have multiple distinct spaces in memory which values "tenure" into. Then, the spaces with young values get collected more frequently than those with older values. Furthermore, we can mix-and-match using any/all of the other garbage collection algorithms for each of the spaces.