**GARBAGE COLLECTION ALGORITHMS**

**ALGORITHM 1: MARK AND SWEEP ALGORITHM**

* Any garbage collection algorithm must perform 2 basic operations.
* One, it should be able to detect all the unreachable objects and secondly, it must reclaim the heap space used by the garbage objects and make the space available again to the program.
* The above operations are performed by Mark and Sweep Algorithm in two phases:

**Mark** and **Sweep** respectively.

* **MARK PHASE:**

1. When an object is created, its mark bit is set to 0(false).
2. In the Mark phase, we set the marked bit for all the reachable objects (or the objects which a user can refer to) to 1(true)
3. Now to perform this operation we simply need to do a graph traversal, a depth first search approach would work for us.
4. Here we can consider every object as a node and then all the nodes (objects) that are reachable from this node (object) are visited and it goes on till we have visited all the reachable nodes.
5. Root is a variable that refer to an object and is directly accessible by local variable. We will assume that we have one root only.
6. We can access the mark bit for an object by: markedBit(obj).

* **SWEEP PHASE:**

1. It clears the heap memory for all the unreachable objects.
2. All those objects whose marked value is set to false are cleared from the heap memory, for all other objects (reachable objects) the marked bit is set to true.
3. Now the mark value for all the reachable objects is set to false, since we will run the algorithm (if required) and again we will go through the mark phase to mark all the reachable objects.

**MARK ALGORITHM**

mark\_sweep\_collect() = mark(root)  
sweep()

mark(o) =  
If mark-bit(o)=0

mark-bit(o)=1  
For p in references(o)

mark(p) EndFor

**SWEEP ALGORITHM**

sweep()  
o = 0  
While o < N

If mark-bit(o)=1 mark-bit(o)=0

Else free(o)

EndIf

o = o + size(o) EndWhile

* The mark-sweep algorithm operates in time linear in the size of the heap (i.e. O(N)).
* it must be invoked whenever an allocation fails
* In practice, the overhead, as well as the pause-time, of mark-sweep collectors is high compared to other algorithms.
* Mark-sweep does however have the advantage of freeing all unused memory, but this free memory easily becomes fragmented (limiting the availability of larger contiguous regions).

**ALGORITHM 2: COPYING GARBAGE COLLECTION**

* At an abstract level, all a copying collector does is start from a set of roots (in our case, the operand stack).
* Then, it traverses all of the reachable memory-allocated objects, copying them from one half of memory into the other half.
* The area of memory that we copy from is called old space and the area of memory that we copy to is called new space.
* When we copy the reachable data, we compact it so that it is in a contiguous chunk.
* So, in effect, we squeeze out the holes

in memory that the garbage data occupied.

After the copy and compaction, we end up with a compacted copy of the data in new space data and a (hopefully) large, contiguous area of memory in new space in which we can quickly and easily.

For example, suppose memory looks something like this, where the colored boxes represent different objects, and the thin black box in the middle represents the half-way point in memory.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Obj 1 | Obj 2 | Obj 3 | Obj 4 | Obj 5 |  |  |  |  |  |  |

At this point, we've filled up half of memory and so we initiate a collection. Old space is on the left and new space on the right. Suppose further that only the red and light-blue boxes (objects 2 and 4) are reachable from the stack. After copying and compacting, we would have a picture like this:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Obj 1 | Obj 2 | Obj 3 | Obj 4 | Obj 5 |  | Obj 2' | Obj 4' |  |  |  |

Notice that we copied the live data (the red and light-blue objects) into new space, but left the unreachable data in the first half. Now we can "throw away" the first half of memory (this doesn't really require any work):

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | Obj 2 | Obj 4 |  |  |  |

After copying the data into new space, we restart the computation where it left off. The computation continues allocating objects, but this time allocates them in the other half of memory (i.e., new space). The fact that we compacted the data makes it easy for the interpreter to allocate objects, because it has a large, contiguous hunk of free memory. So, for instance, we might allocate a few more objects:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | Obj 2 | Obj 4 | Obj 6 | Obj 7 | Obj 8 |

**ALGORITHM 3: GENERATIONAL GARBAGE COLLECTION**

* One major problem with a copying garbage collector is that it has to look at the entire memory every time a garbage collection must happen. Objects in memory have an important property of temporal persistance. Objects that have been in memory for a long time will likely continue to stay in memory, whereas objects that have recently entered memory will likely be discarded very soon.
* To exploit this principle, we can build what is known as a generational garbage collector. Objects will initially be allocated to a chunk of memory called the first generation, or G1. When G1 becomes full, we copy the live objects into another block of memory called the second generation, or G2, and free up the entire G1.
* When G2 becomes full, we continue the process to copy the live objects into G3. This continues for some number of generations n. At this nth generation, Gn, we use some other method such as copying garbage collection to reclaim memory space.
* NOTE: There is one problem with the generational approach. If an object in an older generation points to an object from a newer generation, when we garbage collect the newer generation, we will corrupt the older object. This can be fixed by keeping track of pointers across generations. Fortunately for us, we will restrict our garbage collector to work only for the functional set of MiniML. This means MiniML code with refs in it **will not work** under this garbage collector. We will leave it as a thought exercise to see why cross generational pointers are not an issue without side effects.