Mark-and-Sweep: Garbage Collection Algorithm

Background

All the objects which are created dynamically (using new in C++ and Java) are allocated memory in the heap. If we go on creating objects we might get Out Of Memory error, since it is not possible to allocate heap memory to objects. So we need to clear heap memory by releasing memory for all those objects which are no longer referenced by the program (or the unreachable objects) so that the space is made available for subsequent new objects. This memory can be released by the programmer itself but it seems to be an overhead for the programmer, here garbage collection comes to our rescue, and it automatically releases the heap memory for all the unreferenced objects.
There are many garbage collection algorithms which run in the background. One of them is mark and sweep.

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:
1) Mark phase
2) Sweep phase

Mark Phase

When an object is created, its mark bit is set to 0(false). 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). Now to perform this operation we simply need to do a graph traversal, a depth first search approach would work for us. 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.

- 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.
- We can access the mark bit for an object by: markedBit(obj).

Algorithm -Mark phase:

```
Mark(root)
    If markedBit(root) = false then
        markedBit(root) = true
        For each v referenced by root
             Mark(v)
```

Note: If we have more than one root, then we simply have to call Mark() for all the root variables.

Sweep Phase

As the name suggests it “sweeps” the unreachable objects i.e. it clears the heap memory for all the

unreachable objects. 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 false.
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.

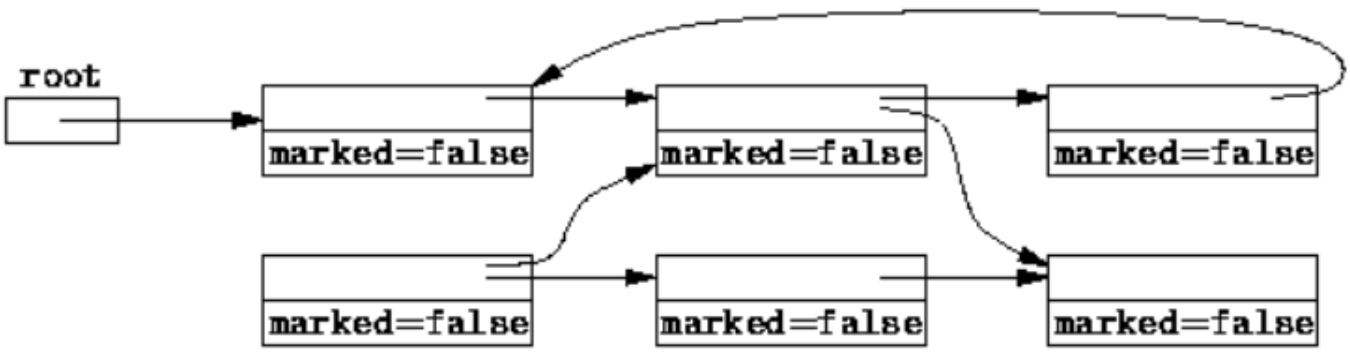
Algorithm – Sweep Phase

```
Sweep()
For each object p in heap
    If markedBit(p) = true then
        markedBit(p) = false
    else
        heap.release(p)
```

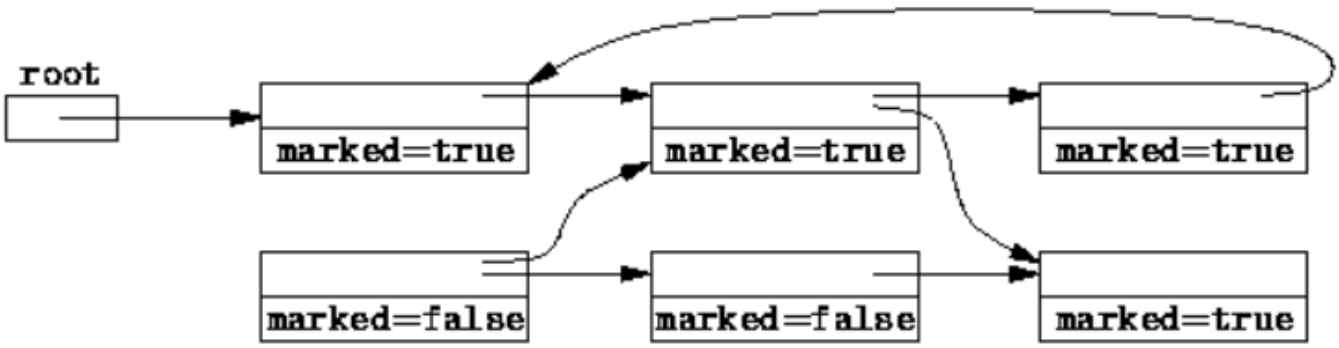
The mark-and-sweep algorithm is called a tracing garbage collector because is traces out the entire collection of objects that are directly or indirectly accessible by the program.

Example:

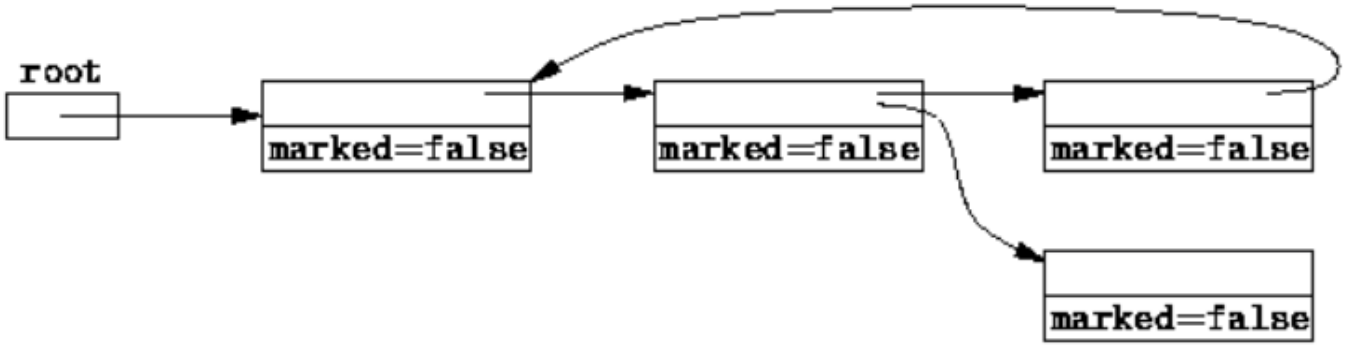
a) All the objects have their marked bits set to false.



b) Reachable objects are marked true



c) Non reachable objects are cleared from the heap.



Advantages of Mark and Sweep Algorithm

- It handles the case with cyclic references, even in case of a cycle, this algorithm never ends up in an infinite loop.
- There are no additional overheads incurred during the execution of the algorithm.

Disadvantages of Mark and Sweep Algorithm

- The main disadvantage of the mark-and-sweep approach is the fact that that normal program execution is suspended while the garbage collection algorithm runs.
- Other disadvantage is that, after the Mark and Sweep Algorithm is run several times on a program, reachable objects end up being separated by many, small unused memory regions. Look at the below figure for better understanding.
  Figure:

Heap Memory

Here white blocks denote the free memory, while the grey blocks denote the memory taken by all the reachable objects.
Now the free segments (which are denoted by white color) are of varying size let's say the 5 free segments are of size 1, 1, 2, 3, 5 (size in units).
Now we need to create an object which takes 10 units of memory, now assuming that memory can be allocated only in contiguous form of blocks, the creation of object isn't possible although we have an available memory space of 12 units and it will cause OutOfMemory error. This problem is termed as "Fragmentation". We have memory available in "fragments" but we are unable to utilize that memory space.
We can reduce the fragmentation by compaction; we shuffle the memory content to place all the free memory blocks together to form one large block. Now consider the above example, after compaction we have a continuous block of free memory of size 12 units so now we can allocate memory to an object of size 10 units.

References: