CC Lecture 15

Prepared for: 7th Sem, CE, DDU

Prepared by: Niyati J. Buch

Basic abstraction of **trace based algorithm**

(e.g mark and sweep)

- All **trace-based algorithms** compute the set of reachable objects and then take the complement of this set.

- Memory is therefore recycled as follows:
  a) The program or mutator runs and **makes allocation** requests.
  b) The garbage collector **discovers reachability** by tracing.
  c) The garbage collector **reclaims the storage** for unreachable objects.

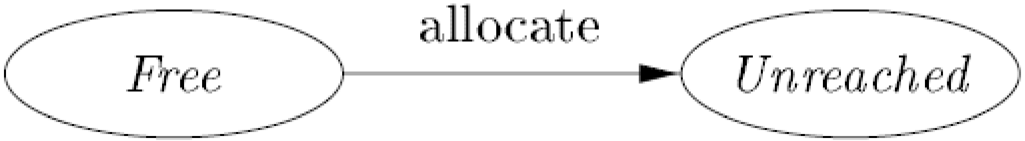
Four states for chunks of memory

1. **Free state**
   - A chunk is in the Free state if it is ready to be allocated.
2. **Unreached state**
   - A chunk is in the Unreached state at any point during garbage collection if its reachability has not yet been established.
3. **Un-scanned state**
   - A chunk is in the Un-scanned state if it is known to be reachable, but its pointers have not yet been scanned.
4. **Scanned state**
   - Every Un-scanned object will eventually be scanned and transition to the Scanned state.

Scanned state

- Every Un-scanned object will eventually be scanned and transition to the Scanned state.
- To scan an object, we examine each of the pointers within it and follow those pointers to the objects to which they refer.
- If a reference is to an Unreached object, then that object is put in the Un-scanned state.
- When the scan of an object is completed, that object is placed in the Scanned state.
- A Scanned object can only contain references to other Scanned or Un-scanned objects, and never to Unreached objects.

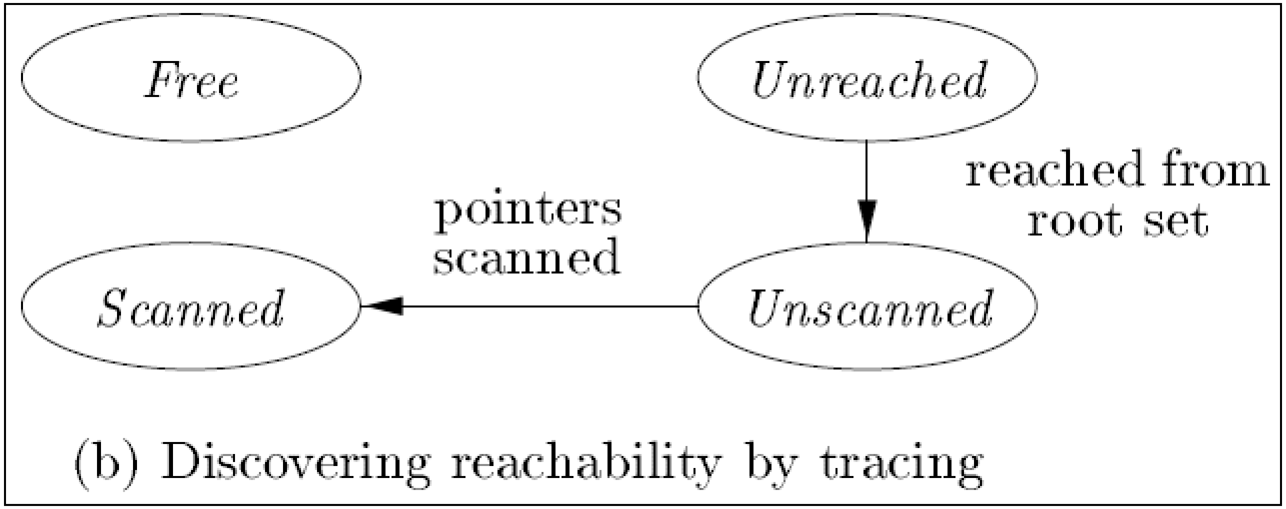
States of memory in a garbage collection cycle



(a) Before tracing: action of mutator

- Whenever a chunk is allocated by the memory manager, its state is set to Unreached as in Fig. (a)

States of memory in a garbage collection cycle

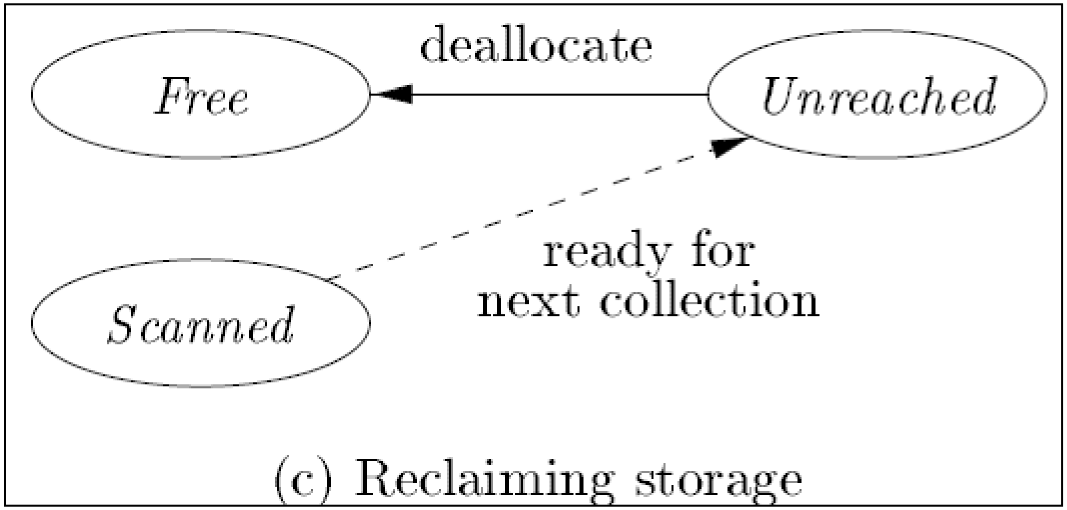


(b) Discovering reachability by tracing

- The transition to Un-scanned from Unreached occurs when we discover that a chunk is reachable as in Fig. (b).

- When the scan of an object is completed, that object is placed in the Scanned state.

States of memory in a garbage collection cycle



(c) Reclaiming storage

- When no objects are left in the Un-scanned state, the computation of reachability is complete.
- Objects left in the Unreached state at the end are truly unreachable.
- The garbage collector reclaims the space they occupy and places the chunks in the Free state.(the solid transition in Fig. (c))
- To get ready for the next cycle of garbage collection, objects in the Scanned state are returned to the Unreached state.(the dashed transition in Fig. (c))

Mark-and-Sweep Algorithm

- **Mark-and-Sweep garbage-collection** algorithm(s) are straightforward, stop-the-world algorithm(s) that find all the unreachable objects, and put them on the list of free space.

- The algorithm has **two** phases
  - visits and "**marks**" all the reachable objects in the first tracing step
  - then "**sweeps**" the entire heap to free up unreachable objects.

Mark-and-Sweep Algorithm

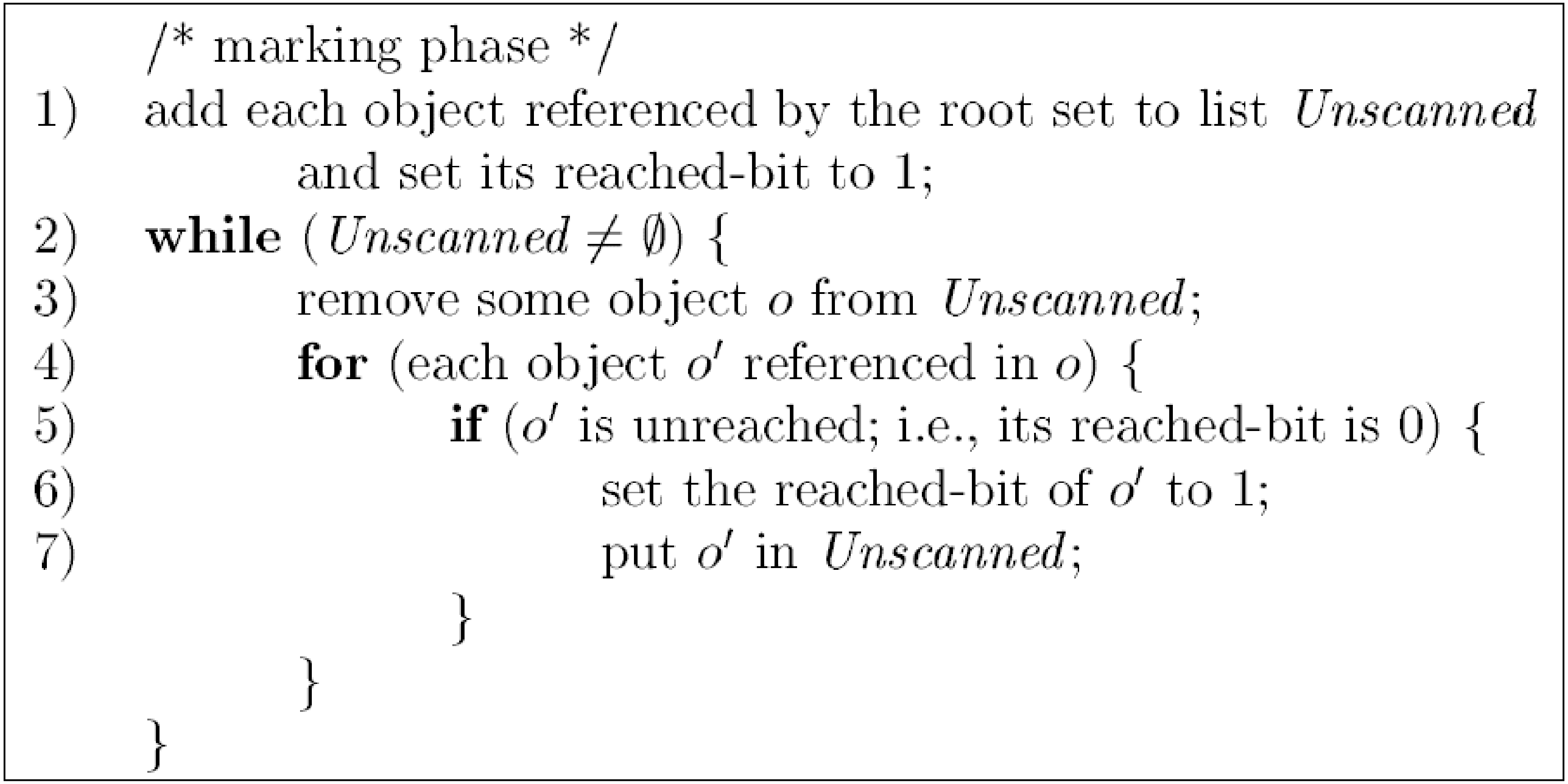
- **INPUT:**
  - A **root set** of objects, a **heap**, and a free **list**, called **Free**, with all the unallocated chunks of the heap.
  - All chunks of space are marked with boundary tags to indicate their free/used status and size.

- **OUTPUT:**
  - A modified **Free** list after all the garbage has been removed.

Mark-and-Sweep Algorithm

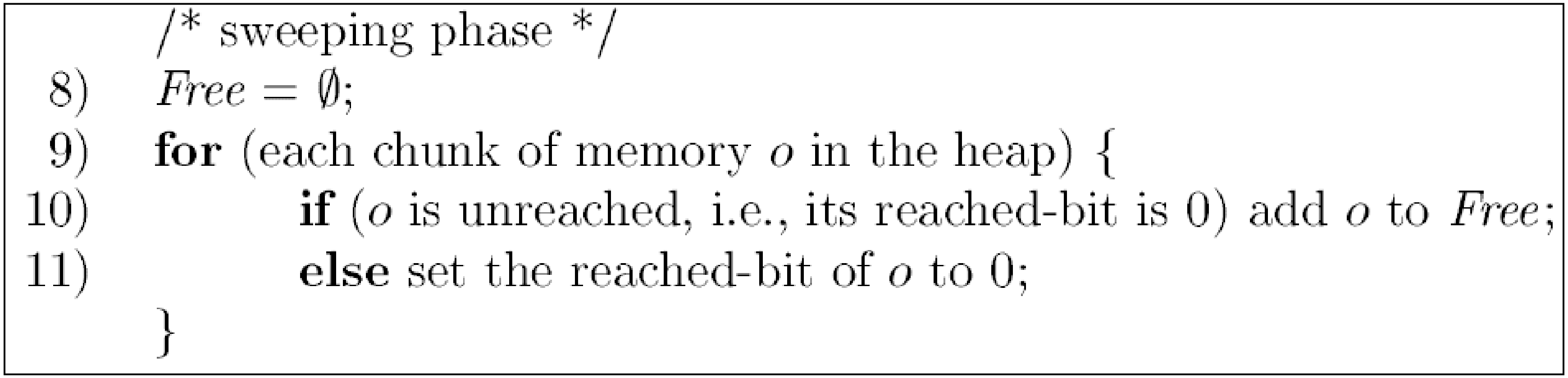
- The algorithm uses several simple data structures.
  - List **Free** holds objects known to be free.
  - A list called **Unscanned**, holds objects that we have determined are reached, but whose successors(other objects can be reached through them) have not yet been considered.
  - The **Unscanned** list is empty initially.
  - Additionally, each object includes a **bit** to indicate whether it has been reached(the **reached-bit**).
  - Before the algorithm begins, all allocated objects have the **reached-bit** set to **0**.

Mark-and-Sweep Algorithm - Mark



```
/* marking phase */
add each object referenced by the root set to list $Unscanned$
        and set its reached-bit to 1;
while ($Unscanned \neq \emptyset$) {
        remove some object $o$ from $Unscanned$;
        for (each object $o'$ referenced in $o$) {
                if ($o'$ is unreached; i.e., its reached-bit is 0) {
                        set the reached-bit of $o'$ to 1;
                        put $o'$ in $Unscanned$;
                }
        }
}
```

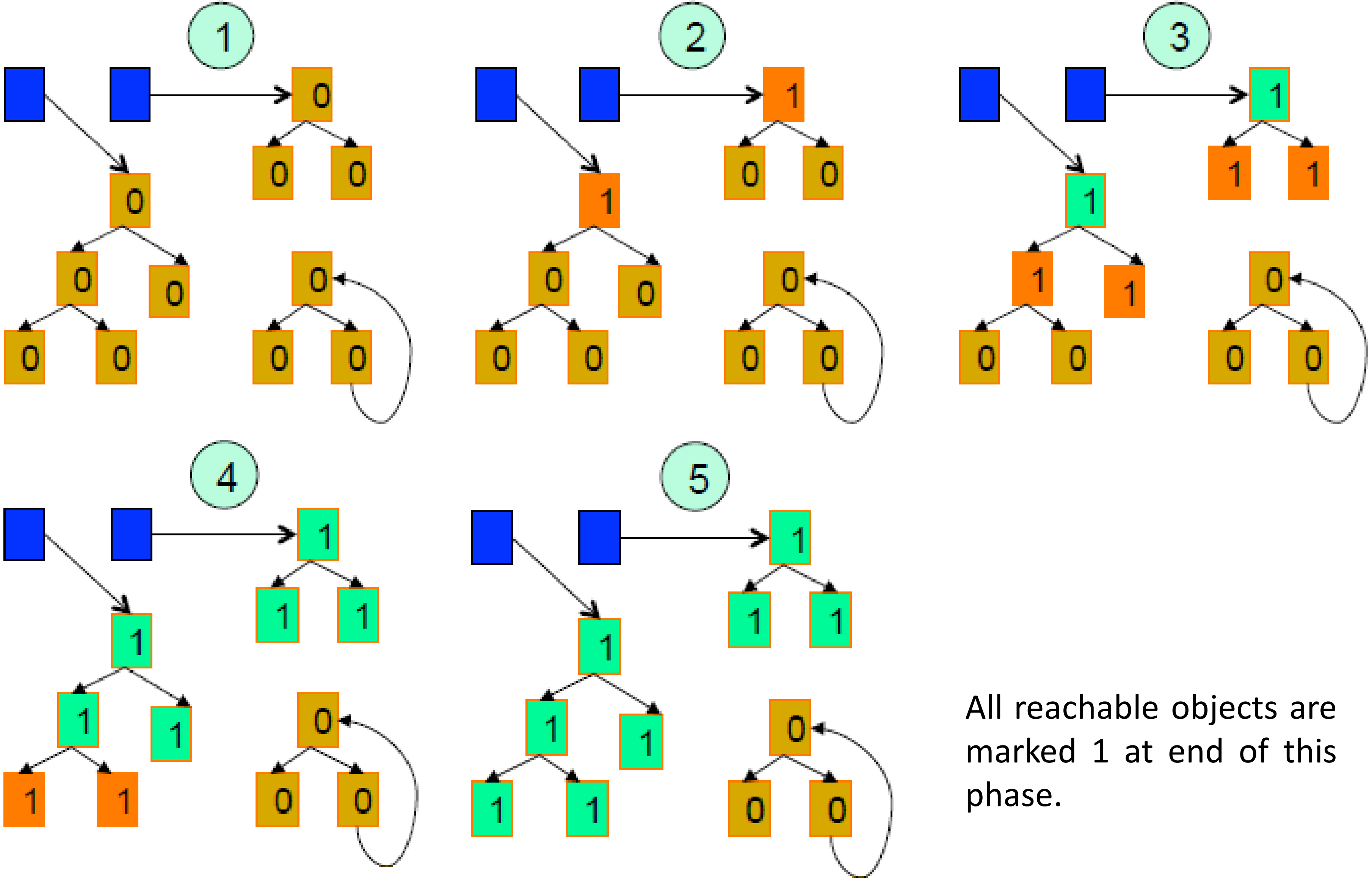
Mark-and-Sweep Algorithm - Sweep



```
/* sweeping phase */
$Free = \emptyset$;
for (each chunk of memory $o$ in the heap) {
        if ($o$ is unreached, i.e., its reached-bit is 0) add $o$ to $Free$;
        else set the reached-bit of $o$ to 0;
}
```

Mark-and-Sweep – Example

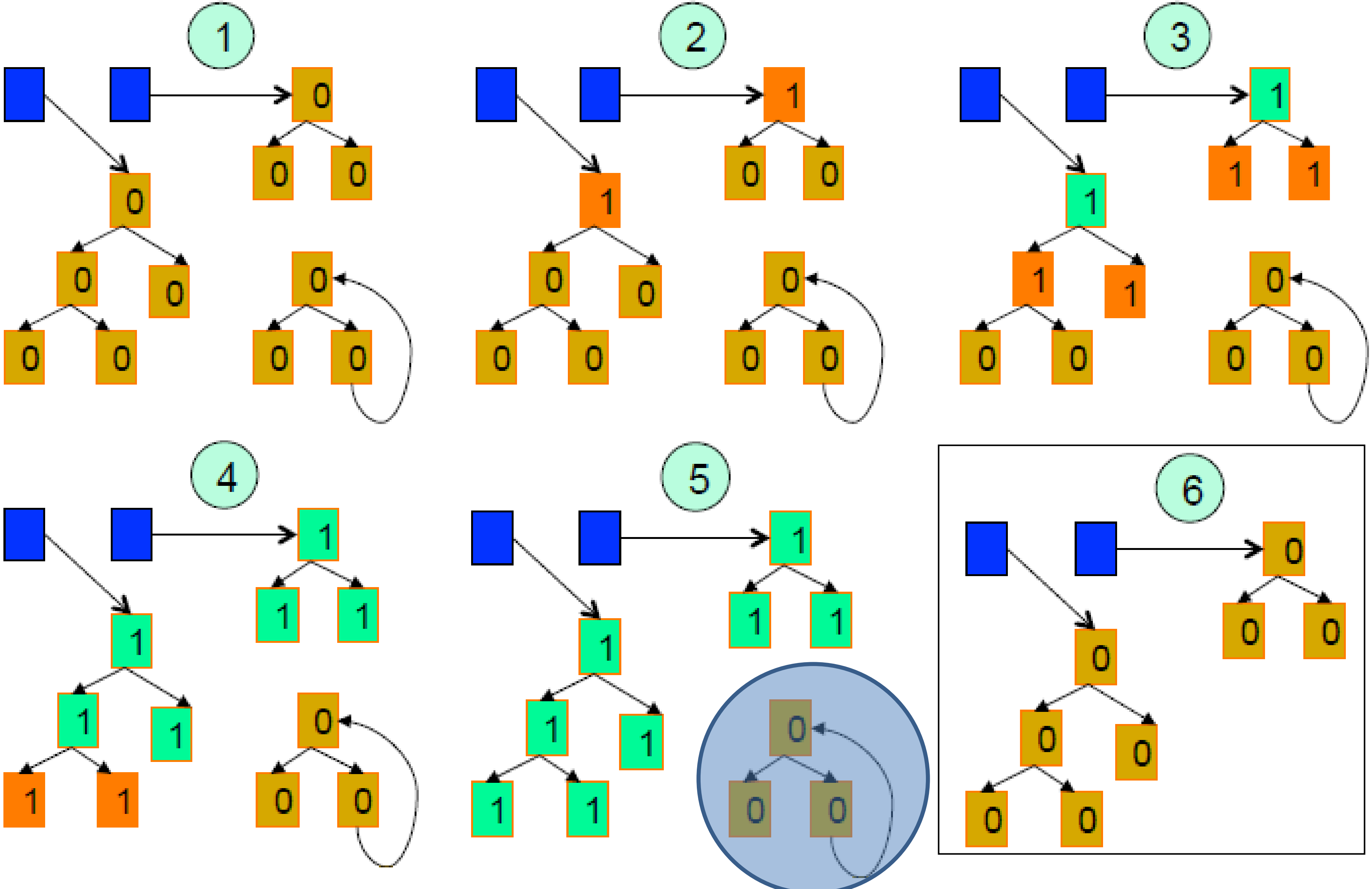
(NPTEL Course on Principles of Compiler Design by Y.N. Srikant)



All reachable objects are marked 1 at end of this phase.

Mark-and-**Sweep** – Example

(NPTEL Course on Principles of Compiler Design by Y.N. Srikant)



Optimizing Mark-and-Sweep

- The final step in the basic mark-and-sweep algorithm is expensive because there is no easy way to find only the unreachable objects without examining the entire heap.
- So the time taken is proportional to the size of the heap.

- An improved algorithm, by **Baker**, keeps a list of all allocated objects.

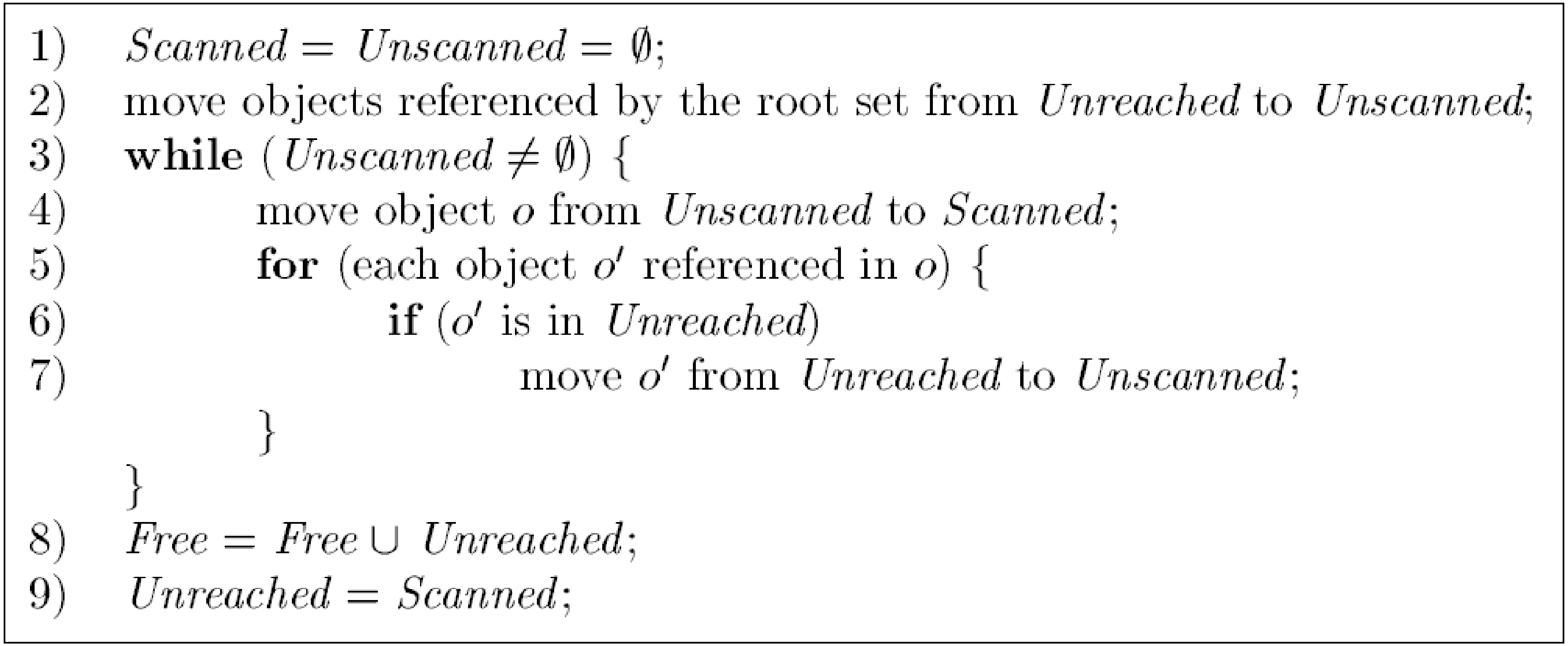
- To find the set of unreachable objects, which we must return to free space, we take the set difference of the allocated objects and the reached objects.

Baker's Mark-and-Sweep collector

- **INPUT**:
  - A root set of objects, a heap, a free list Free, and a list of allocated objects, which we refer to as Unreached.

- **OUTPUT**:
  - Modified lists Free and Unreached, which holds allocated objects.

Baker's Mark-and-Sweep collector



```
$Scanned = Unscanned = \emptyset$;
move objects referenced by the root set from $Unreached$ to $Unscanned$;
while ($Unscanned \neq \emptyset$) {
        move object $o$ from $Unscanned$ to $Scanned$;
        for (each object $o'$ referenced in $o$) {
                if ($o'$ is in $Unreached$)
                        move $o'$ from $Unreached$ to $Unscanned$;
        }
}
$Free = Free \cup Unreached$;
$Unreached = Scanned$;
```

Relocating Collectors

- **Relocating collectors** move reachable objects around in the heap to eliminate memory fragmentation.
- It is common that the space occupied by reachable objects is much smaller than the freed space.
- Instead of freeing the holes individually, relocate all the reachable objects into one end of the heap, leaving the entire rest of the heap as one free chunk.
- As GC already analyzed every reference within the reachable objects
- So this and references in root set is required to be changed.

Advantages

- Having all the reachable objects in contiguous locations reduces fragmentation of the memory space.
- Also, by making the data occupy fewer cache lines and pages, relocation improves a program‘s temporal and spatial locality, since new objects created at about the same time are allocated nearby chunks.
- Objects in nearby chunks can benefit from prefetching if they are used together.
- Further, the data structure for maintaining free space is simplified; instead of a free list, all we need is a pointer free to the beginning of the one free block.

Types of Relocating Collectors

- **Relocating collectors** vary in whether they relocate in place or reserve space ahead of time for the relocation:

1. A **Mark-and-Compact collector**, described in this section, compacts objects in place.
    - Relocating in place reduces memory usage.

2. The more efficient and popular **Copying Collector** moves objects from one region of memory to another.
    - Reserving extra space for relocation allows reachable objects to be moved as they are discovered.

3 phases of Mark-and-Compact collector

1. First is a **marking** phase, similar to that of the mark-and-sweep algorithms described previously.
2. Second, the algorithm **scans** the allocated section of the heap and **computes a new address** for each of the reachable objects.
    - New addresses are assigned from the low end of the heap, so there are no holes between reachable objects.
    - The new address for each object is recorded in a structure called NewLocation.
3. Finally, the algorithm **copies** objects to their new locations, updating all references in the objects to point to the corresponding new locations.
    - The needed addresses are found in NewLocation.