CC WEEK 9

Prepared for: 7th Sem, CE, DDU

Prepared by: Niyati J. Buch

Introduction to Garbage Collection

- Data that cannot be referenced is generally known as **garbage**.
- Many high-level programming languages remove the burden of manual memory management from the programmer by offering **automatic garbage collection**, which deallocates unreachable data.
- Languages supporting GC
  - Lisp(since 1958), Java, C#, Perl, Python, Prolog, etc .

Design Goals for GC

- **Garbage collection** is the reclamation of chunks of storage holding objects that can no longer be accessed by a program.
- Assumptions for GC
  - **Type** of object must be determined by GC at runtime (**Type safety**)
    - **Size** and **pointer fields** can be determined by GC
  - **References** to objects are always to the address of the **beginning** of the object
  - All **references** to an object have the **same value** and can be identified easily

Mechanism

- A user program, **the mutator**, modifies the collection of objects in the heap.
- The mutator creates objects by acquiring space from the memory manager, and the mutator may introduce and drop references to existing objects.
- Objects become garbage when the mutator program cannot "**reach**" them.
- The garbage collector finds these **unreachable objects** and reclaims their space by handing them to the memory manager, which keeps track of the free space.

Requirements/Performance Metrics

1. **Overall Execution time**
   - As Garbage collection can be very slow, it is important that it not significantly increase the total run time of an application.

2. **Space usage**
   - It is important that garbage collection avoid fragmentation and make the best use of the available memory.

Requirements/Performance Metrics (cont.)

3. **Pause time**
   - Simple garbage collectors are notorious for causing programs (the mutators) to pause suddenly for an extremely long time, as garbage collection kicks in without warning.
   - Thus, besides minimizing the overall execution time, it is desirable that the maximum pause time be minimized.

4. **Program locality**
   - It can improve a mutator's <u>temporal locality</u> by freeing up space and reusing it.
   - It can improve the mutator's <u>spatial locality</u> by relocating data used together in the same cache or pages.

Reachability of Objects

- All the data that can be accessed (reached) directly by a program without having to dereference any pointer is referred as the **root set**.
- Recursively, any object whose reference is stored in a field of a member of the root set is also reachable.

- **New objects** are introduced through object allocations and add to the set of reachable objects.
- **Parameter passing** and **assignments** can propagate reachability.
- **Assignments** and **ends of procedures** can terminate reachability.

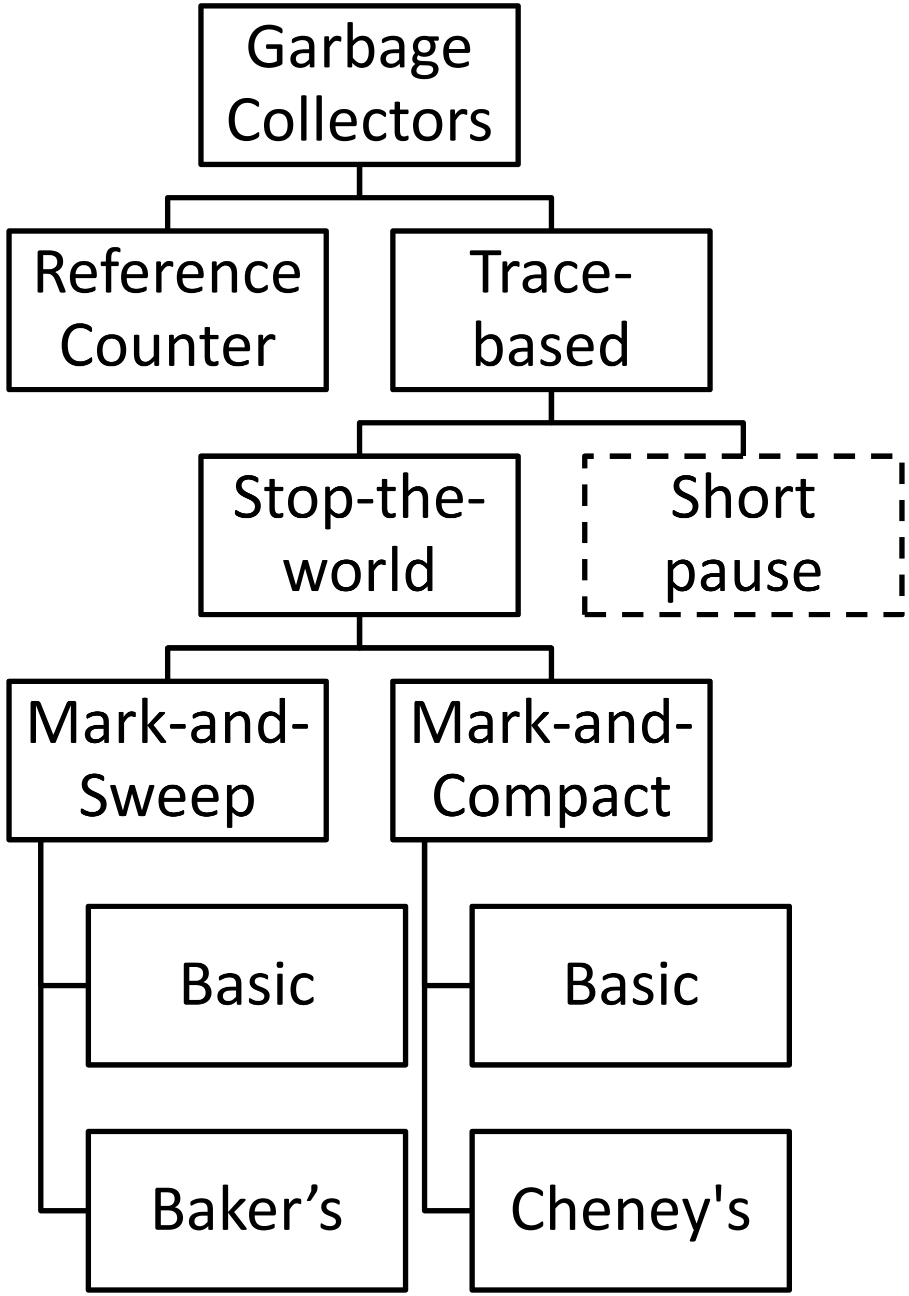
- Similarly, an object that becomes *unreachable can cause more objects to* become unreachable.

How to find unreachable objects?

- A garbage collector periodically **finds** all unreachable objects by one of the two methods

1. Catch the transitions as reachable objects become unreachable

2. Or, periodically locate all reachable objects and infer that all other objects are unreachable



http://infolab.stanford.edu/~ullman/dragon/w06/lectures/gc.pdf

Reference Counting Garbage Collector

"Catch the transitions as reachable objects become unreachable"

- This approach is used by Reference Counting GC.

- A **count of the references** to an object is maintained, as the mutator (program) performs actions that may change the reachability set.
- When the **count** becomes zero, the object becomes **unreachable**.

- **Reference count** requires an **extra field** in the object.

Maintaining Reference Counts

1. **Object Allocation.**
   - The reference count of the new object is set to 1.
   - ref_count = 1
2. **Parameter Passing.**
   - The reference count of each object passed into a procedure is incremented.
   - ref_count++
3. **Reference Assignments.**
   - For statement u = v, where u and v are references, the reference count of the object referred to by v goes up by one, and the count for the old object referred to by u goes down by one.
   - For u, ref_count--
   - For v, ref_count++

Maintaining Reference Counts

4. **Procedure Returns.**
   - As a procedure exits, objects referred to by the local variables in its activation record have their counts decremented.
   - If several local variables hold references to the same object, that object's count must be decremented once for each such reference.
   - ref_count--

5. **Transitive Loss of Reachability.**
   - Whenever the reference count of an object becomes zero, we must also decrement the count of each object pointed to by a reference within the object.
   - Transitively, ref_count--

Advantages of Reference Counting GC

- Garbage collection is incremental
  - overheads are distributed to the mutator's operations
  - are spread out throughout the life time of the mutator
- Garbage is collected immediately and hence space usage is low
- Useful for real-time and interactive applications, where long and sudden pauses are unacceptable

Disadvantages of Reference Counting GC

- High overhead due to reference maintenance
  - additional operations are introduced with each reference assignment, and at procedure entries and exits.
  - This overhead is proportional to the amount of computation in the program, and not just to the number of objects in the system.

- Cannot collect unreachable cyclic data structures
  - E.g. circularly linked lists
  - since the reference counts never become zero

Unreachable Cyclic Data Structure

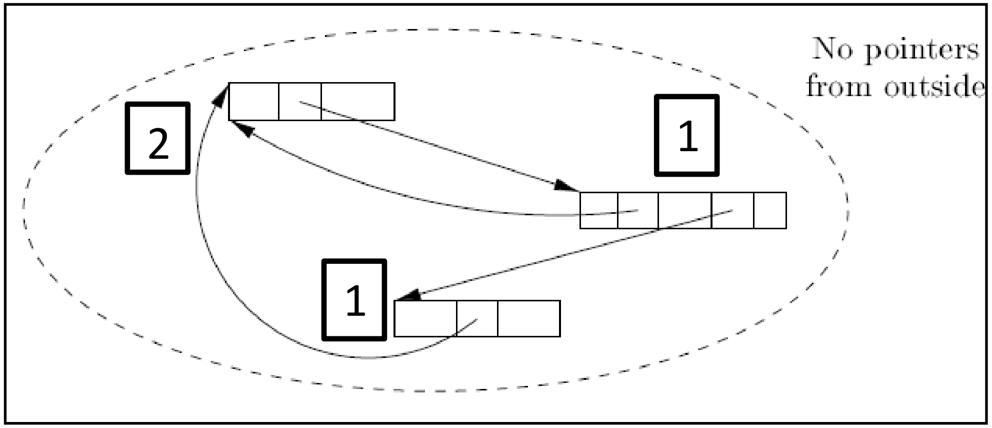
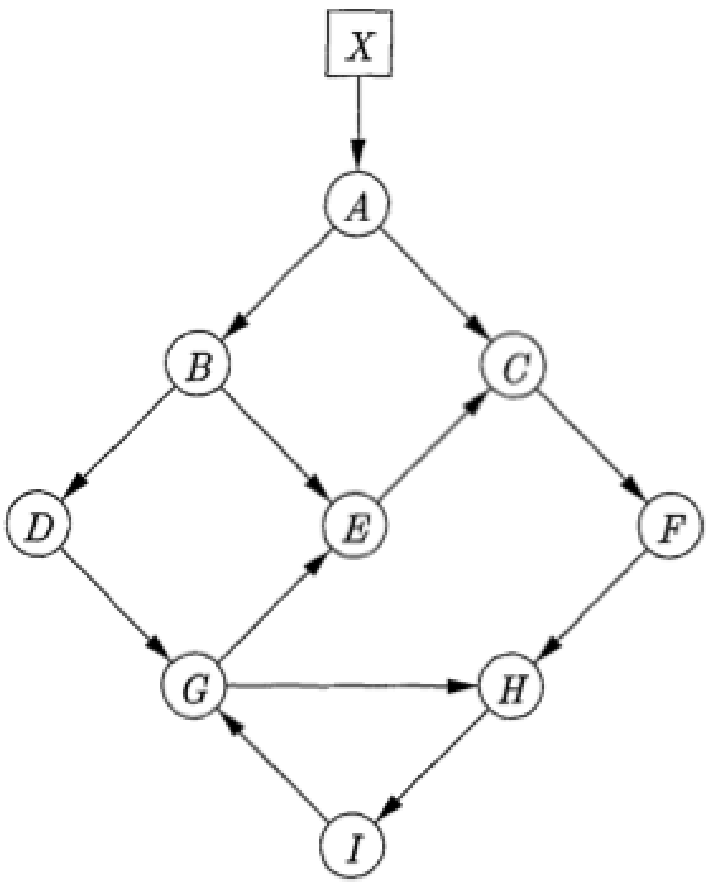


Figure shows three objects with references among them, but no references from anywhere else.

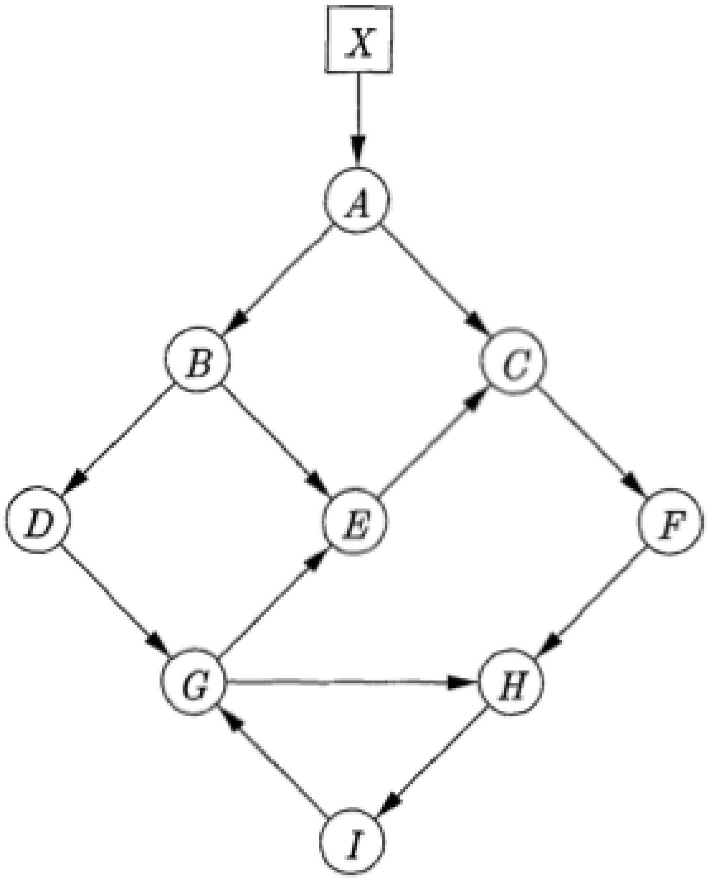
- If none of these objects is part of the root set, then they are all garbage, but their reference counts are each greater than 0.
- Such a situation is equivalent to to a memory leak if we use reference counting for garbage collection, since then this garbage and any structures like it are never deallocated.

Example
(from Aho Ullman book)

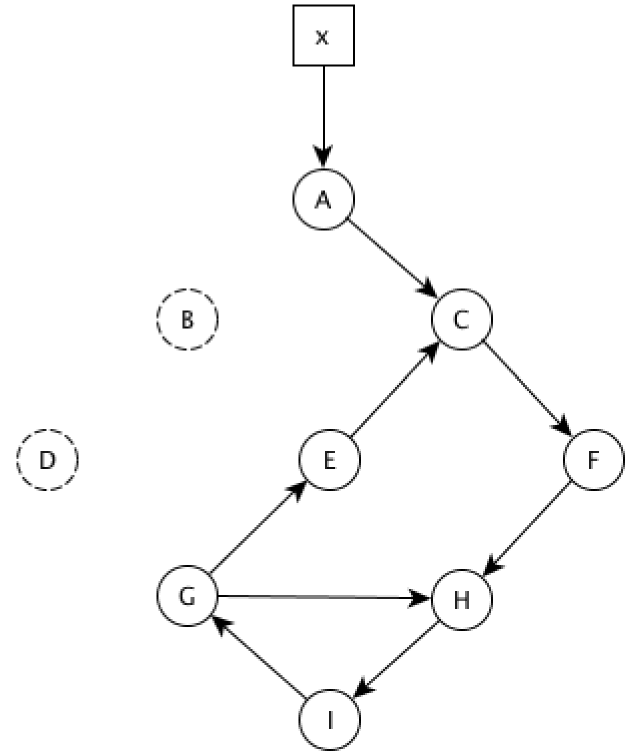


- What happens to the reference counts of the objects if the pointer from A to B is deleted?

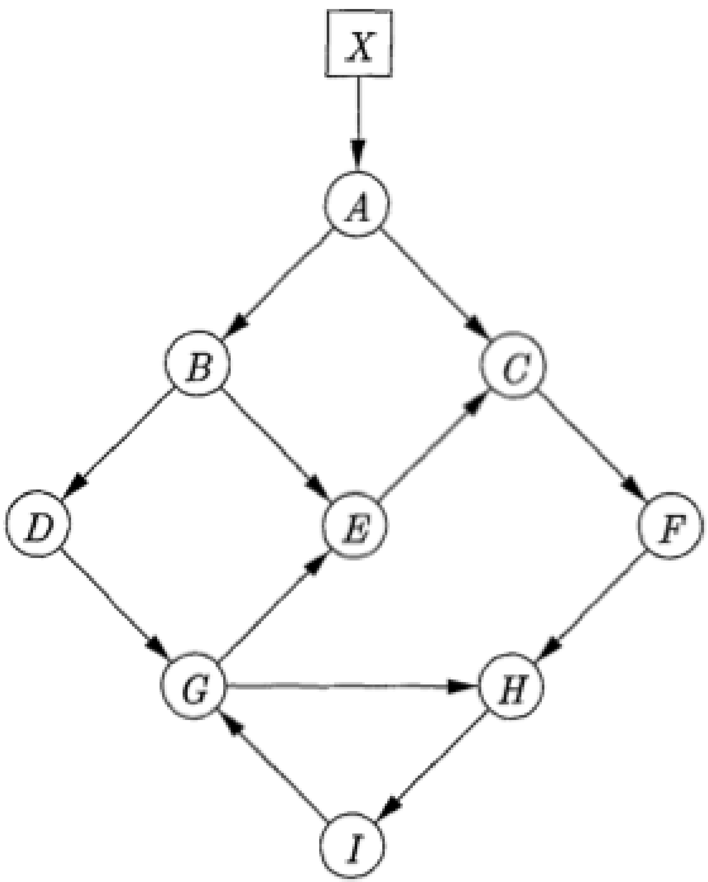
Example
(from Aho Ullman book)



- What happens to the reference counts of the objects if the pointer from A to B is deleted?

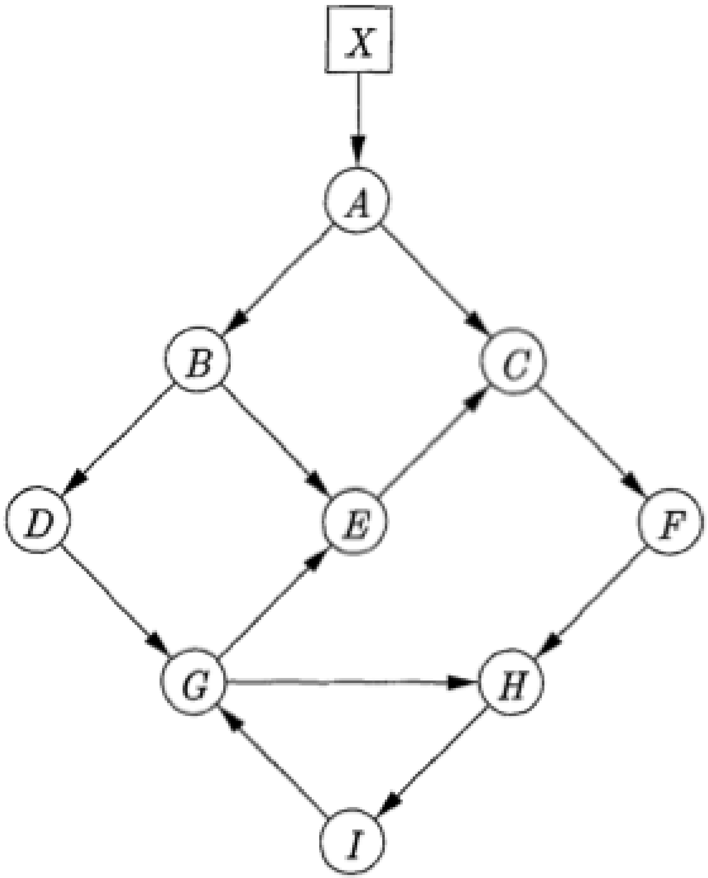


Example
(from Aho Ullman book)

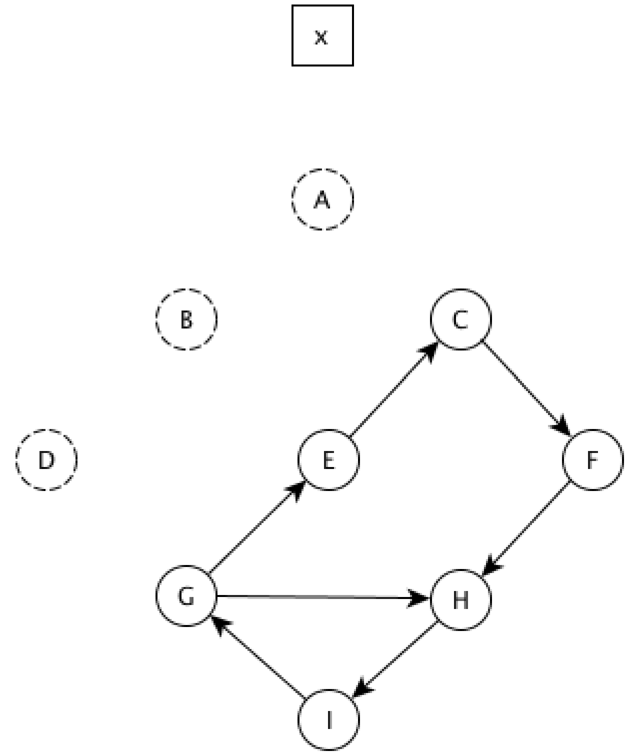


- What happens to the reference counts of the objects if the pointer from X to A is deleted?

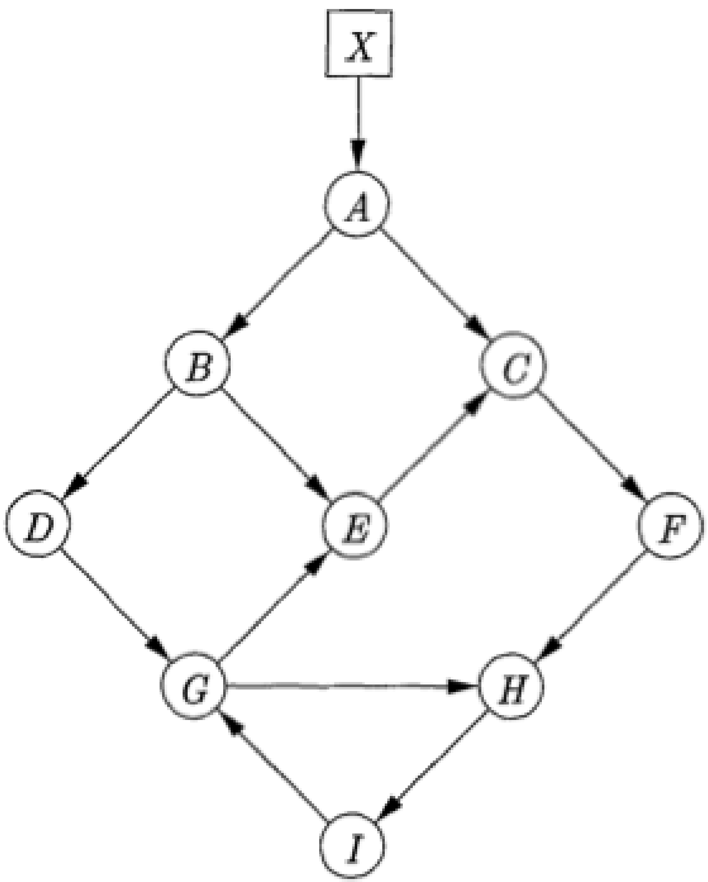
Example
(from Aho Ullman book)



- What happens to the reference counts of the objects if the pointer from X to A is deleted?

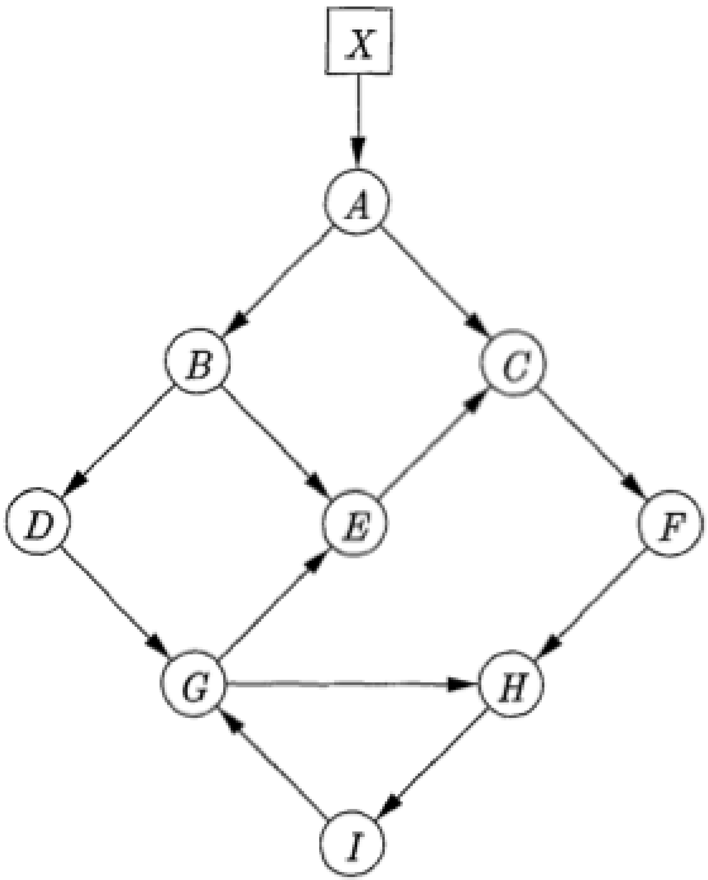


Example
(from Aho Ullman book)

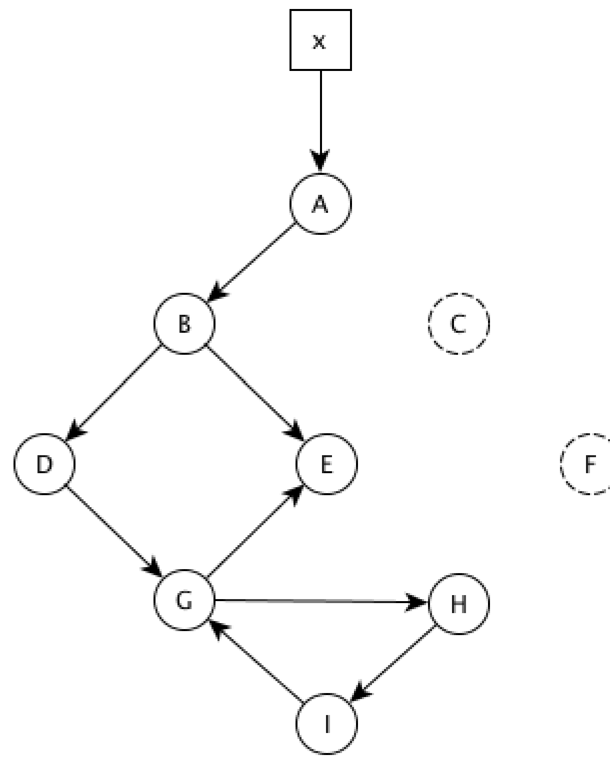


- What happens to the reference counts of the objects if the node C is deleted?

Example
(from Aho Ullman book)



- What happens to the reference counts of the objects if the node C is deleted?



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.

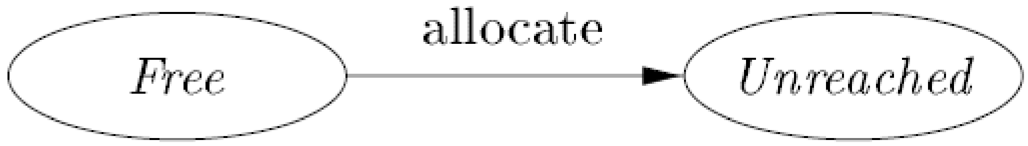
Four states for chunks of memory

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.

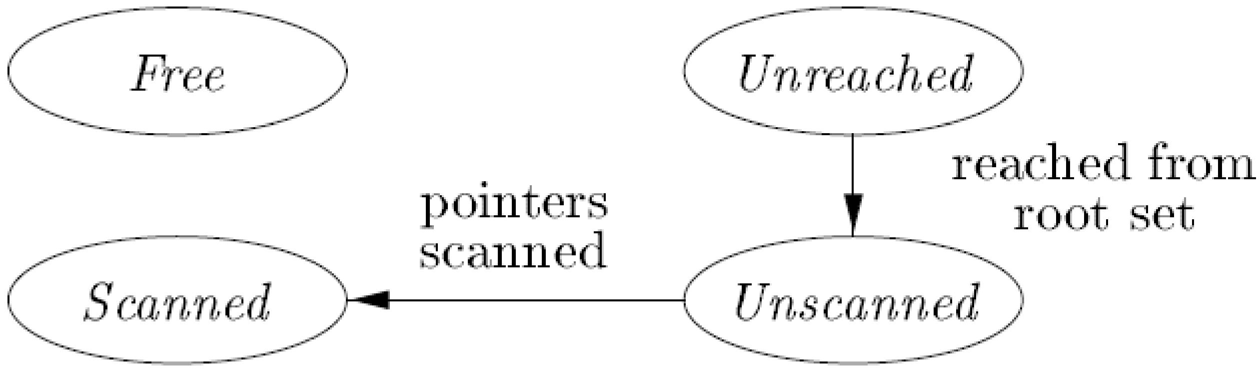
States of memory in a garbage collection cycle



(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)

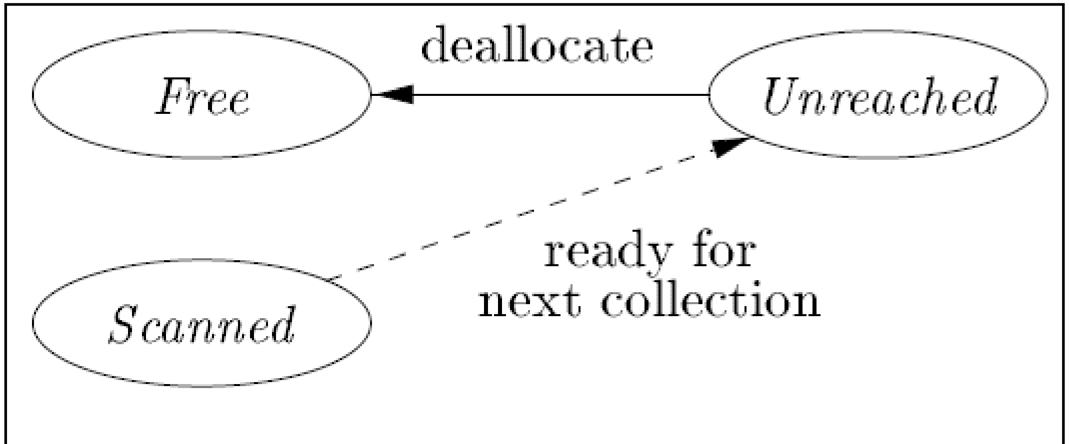
States of memory in a garbage collection cycle



(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 Unscanned 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**.

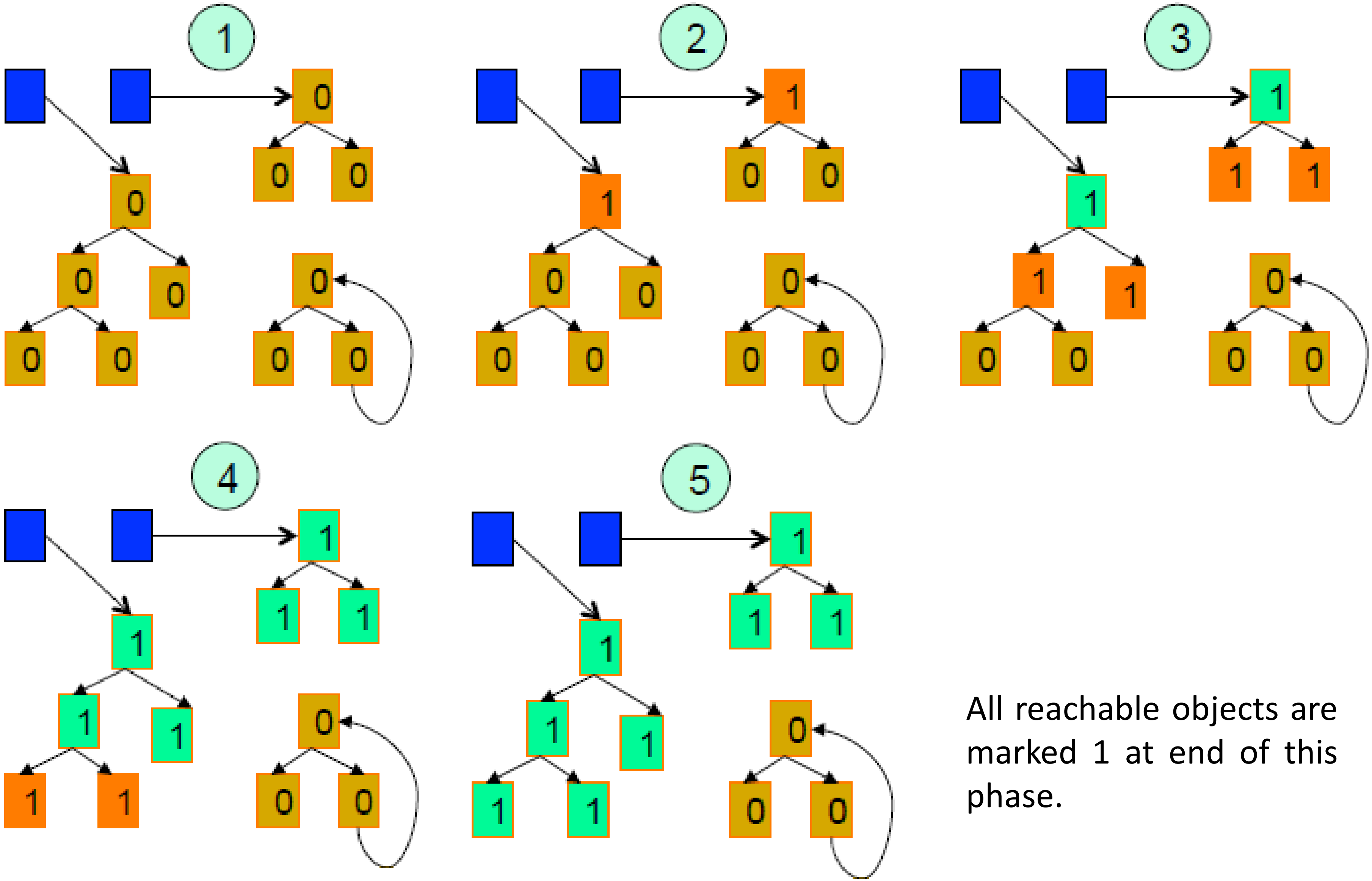
Mark-and-Sweep Algorithm - Mark

```
    /* marking phase */
1)  add each object referenced by the root set to list Unscanned
            and set its reached-bit to 1;
2)  while (Unscanned ≠ ∅) {
3)          remove some object o from Unscanned;
4)          for (each object o′ referenced in o) {
5)                  if (o′ is unreached; i.e., its reached-bit is 0) {
6)                          set the reached-bit of o′ to 1;
7)                          put o′ in Unscanned;
                    }
            }
    }
```

Mark-and-Sweep Algorithm - Sweep

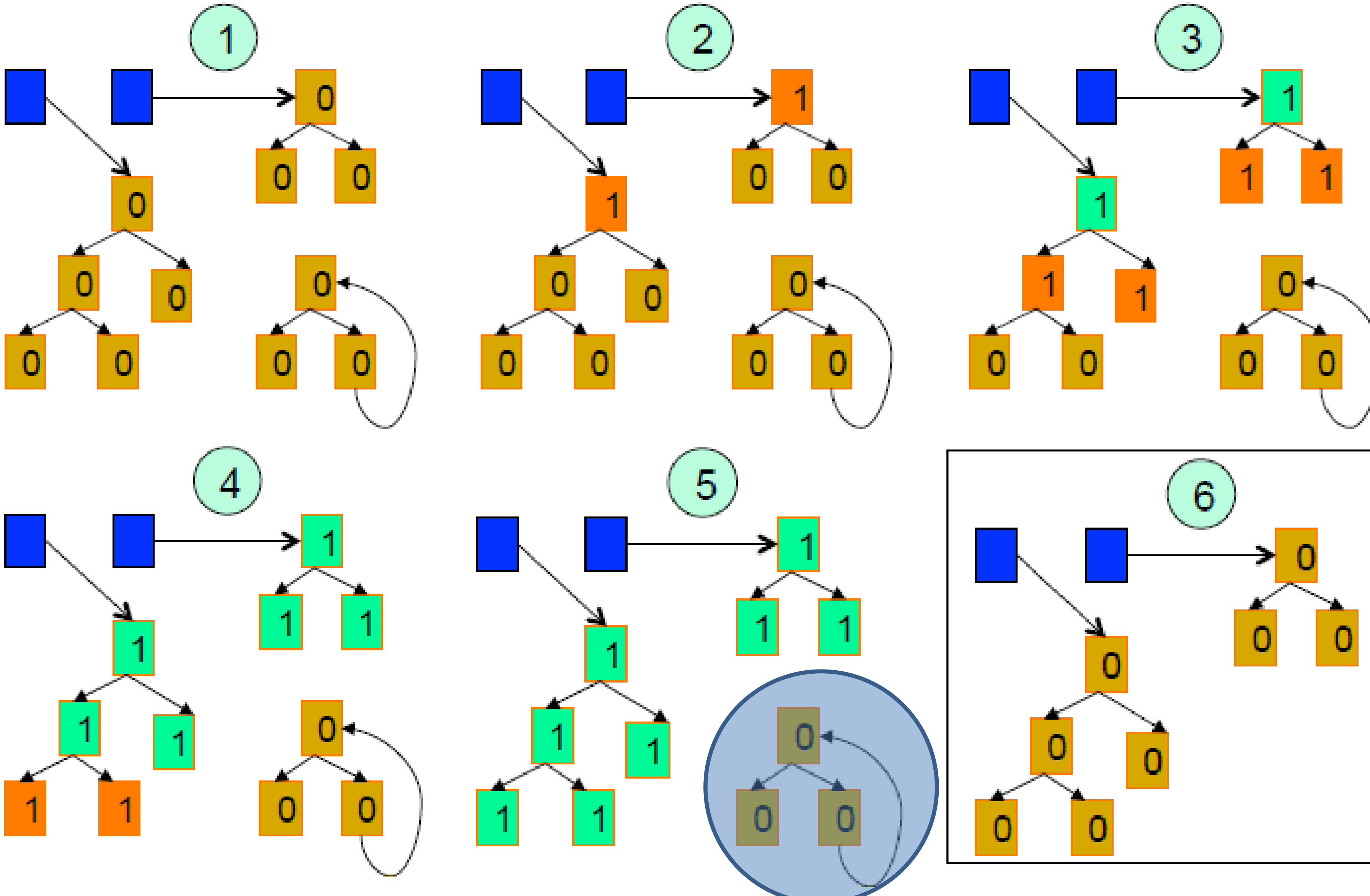
```
/* sweeping phase */
Free = ∅;
for (each chunk of memory o in the heap) {
        if (o is unreached, i.e., its reached-bit is 0) add o to Free;
        else set the reached-bit of o to 0;
}
```

Mark and Sweep - Example



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

Mark and **Sweep** - Example



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.
- 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 = ∅;
move objects referenced by the root set from Unreached to Unscanned;
while (Unscanned ≠ ∅) {
        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 ∪ 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.

Phase 1: Mark-and-Compact collector

```
/* mark */
Unscanned = set of objects referenced by the root set;
while (Unscanned ≠ ∅) {
    remove object o from Unscanned;
    for (each object o′ referenced in o) {
        if (o′ is unreached) {
            mark o′ as reached;
            put o′ on list Unscanned;
        }
    }
}
```

This is just like mark phase in mark and sweep algorithm.

Phase 2: Mark-and-Compact collector

```
/* compute new locations */
free = starting location of heap storage;
for (each chunk of memory o in the heap, from the low end) {
        if (o is reached) {
                NewLocation(o) = free;
                free = free + sizeof(o);
        }
}
```

Maintain a table (hash?) from reached chunks to new locations for the objects in those chunks.
Scan chunks from low end of heap.
Maintain pointer free that counts how much space is used by reached objects so far.

Phase 3: Mark-and-Compact collector

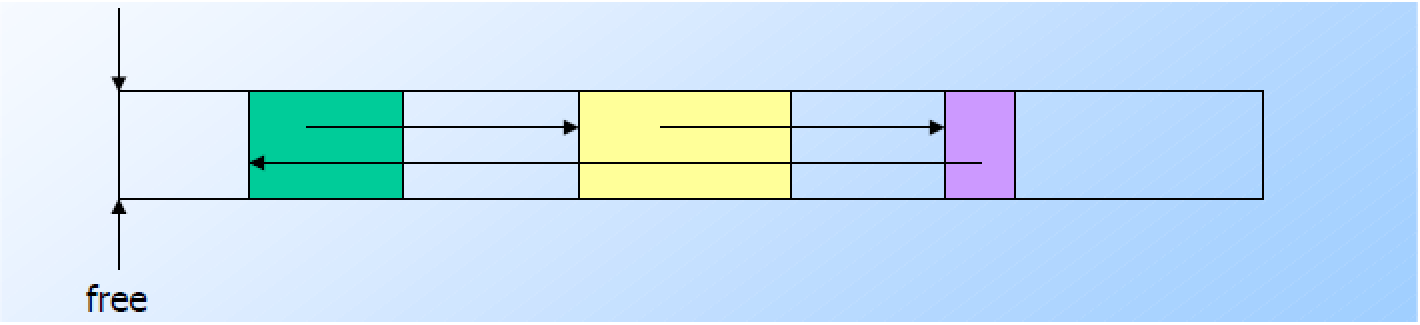
```
/* retarget references and move reached objects */
for (each chunk of memory o in the heap, from the low end) {
        if (o is reached) {
                for (each reference o.r in o)
                        o.r = NewLocation(o.r);
                copy o to NewLocation(o);
        }
}
for (each reference r in the root set)
        r = NewLocation(r);
```

Move all reached objects to their new locations, and also retarget all references in those objects to the new locations.

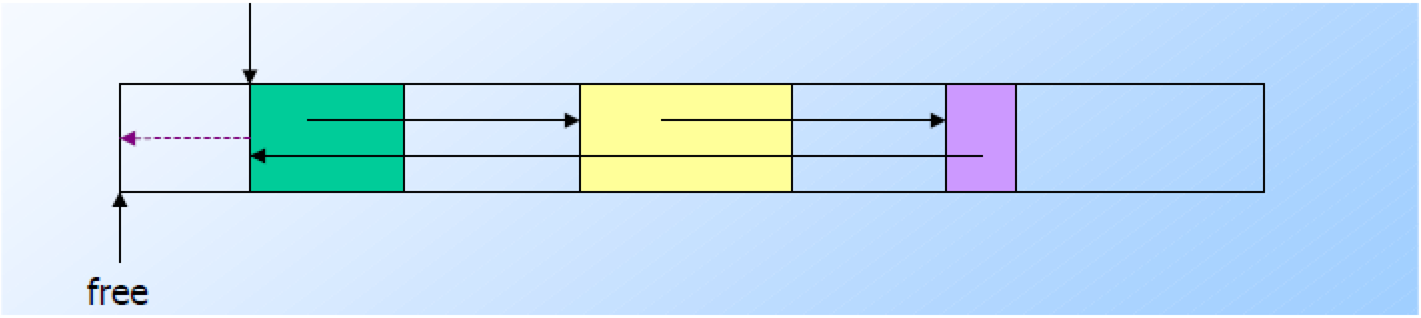
Use the table of new locations.

Retarget root references.

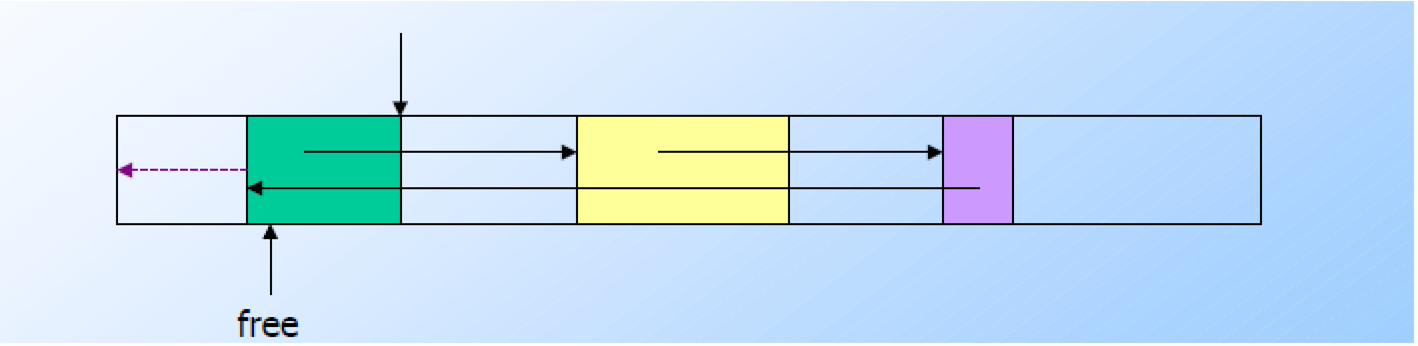
Example



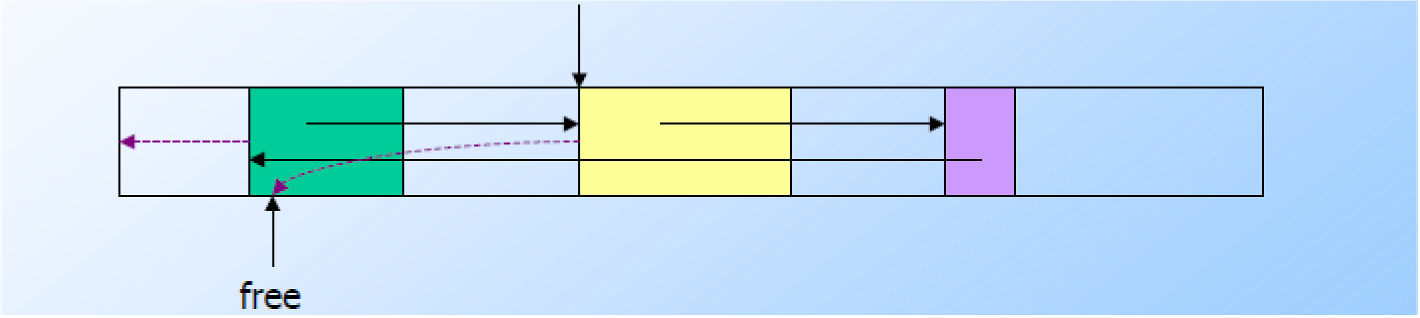
Example



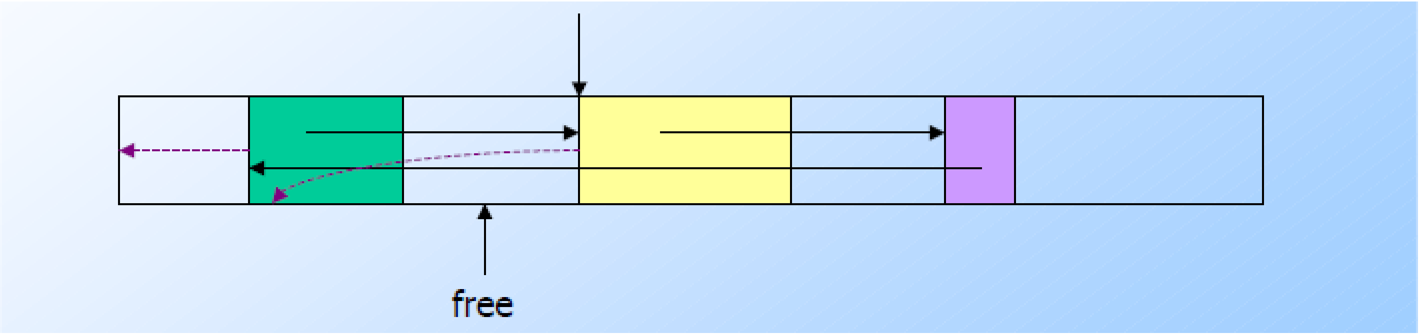
Example



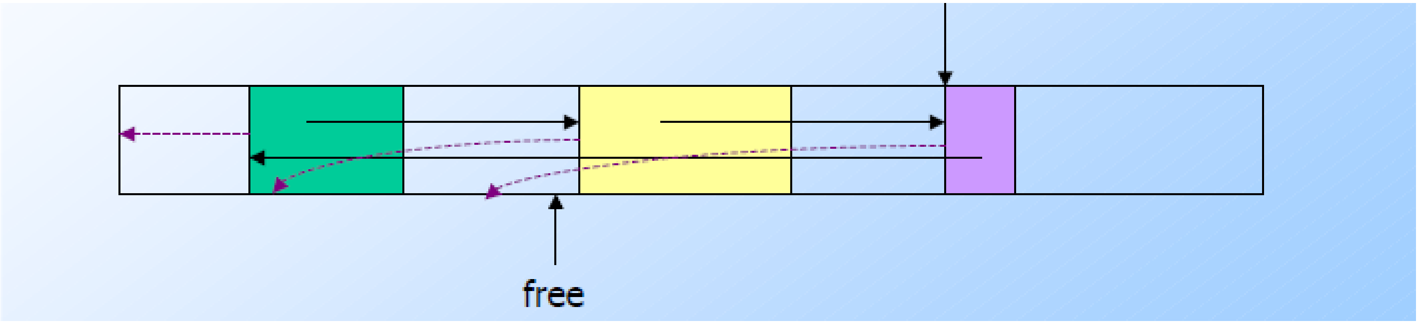
Example



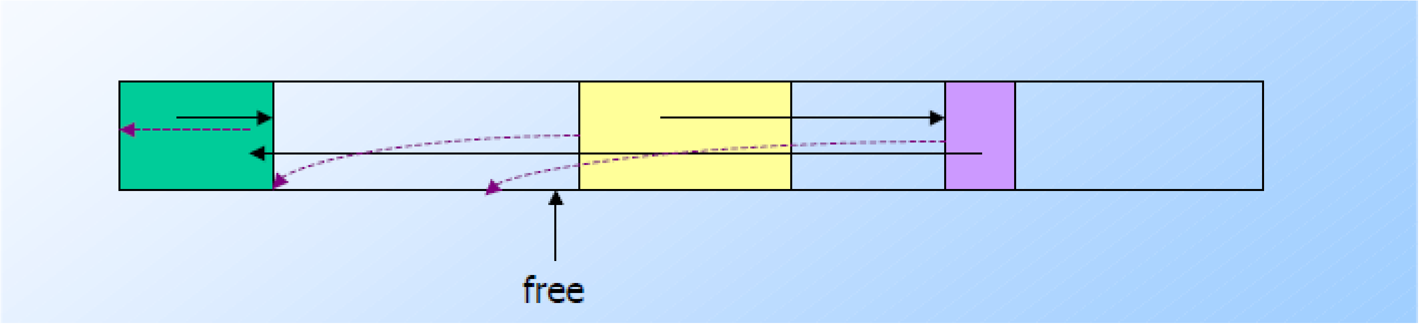
Example



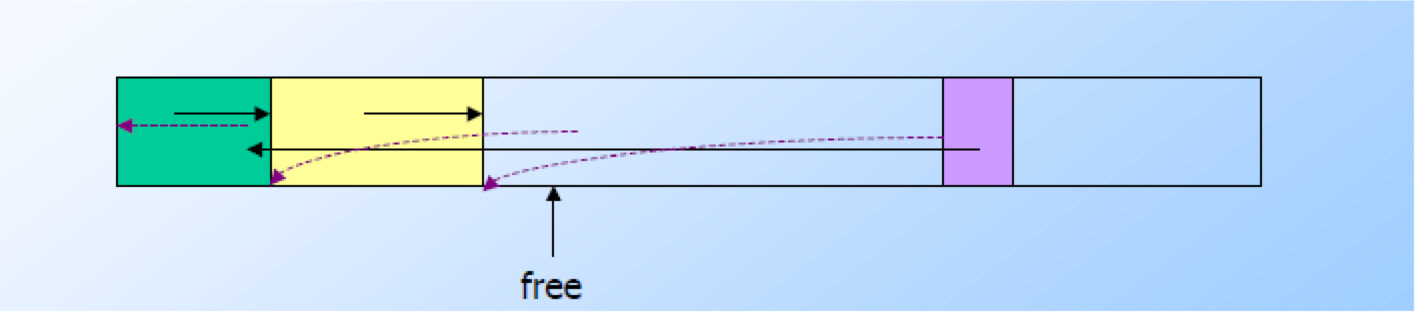
Example



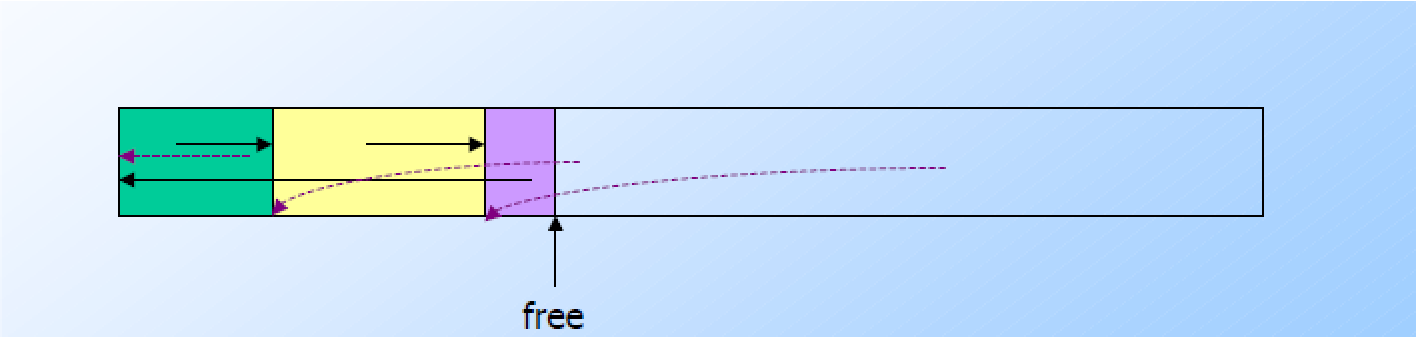
Example



Example



Example



Comparison

- **Basic Mark-and-Sweep**:
  - Cost is proportional to the number of chunks in the heap
- **Baker's Mark-and-Sweep**:
  - Cost is proportional to the number of reached objects
- **Basic Mark-and-Compact**:
  - Cost is proportional to the number of chunks in the heap plus the total size of the reached objects

- Cheney's Copying collector:
  - Cost proportional to the total size of the reached objects (it does not touch any of the unreachable object)