CC Lecture 14

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

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

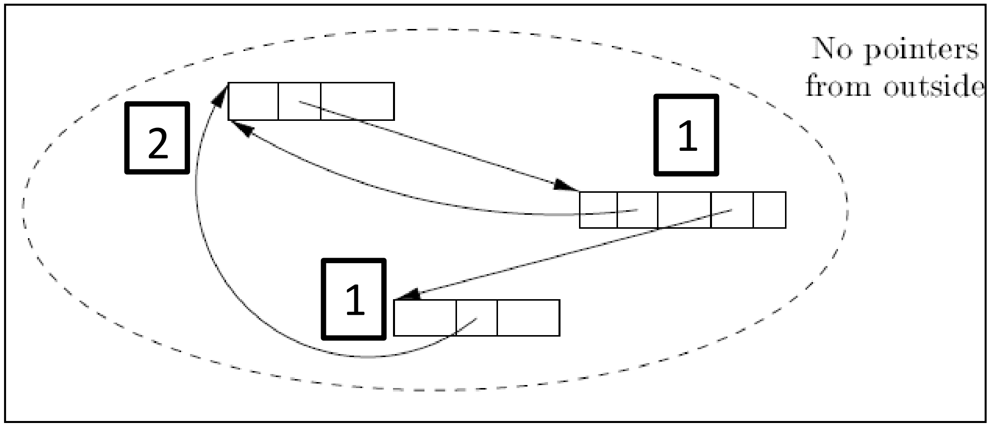
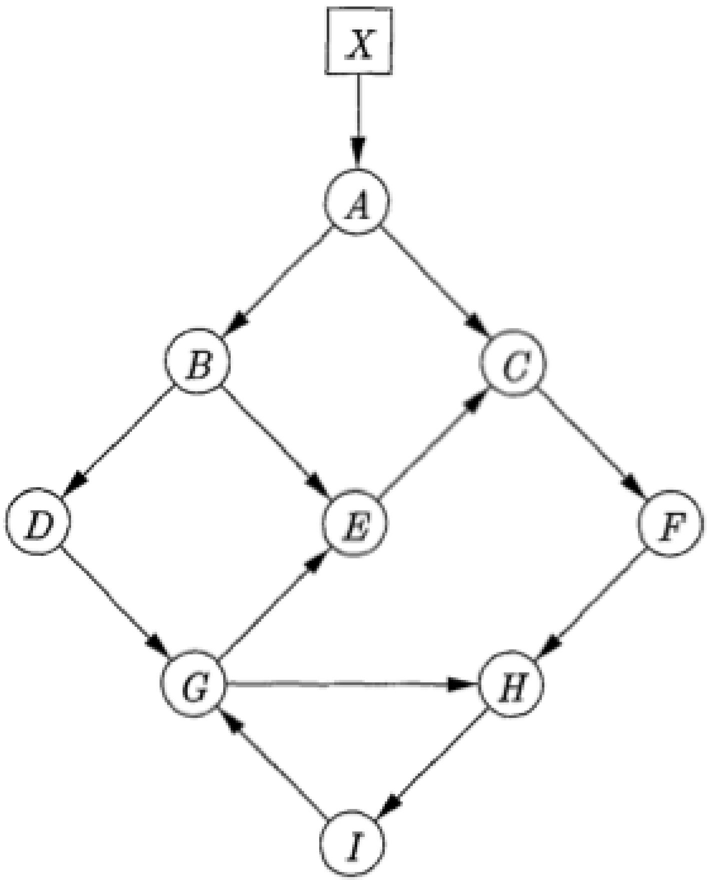


Figure shows three objects with references among them, but no references from anywhere else.

- 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.

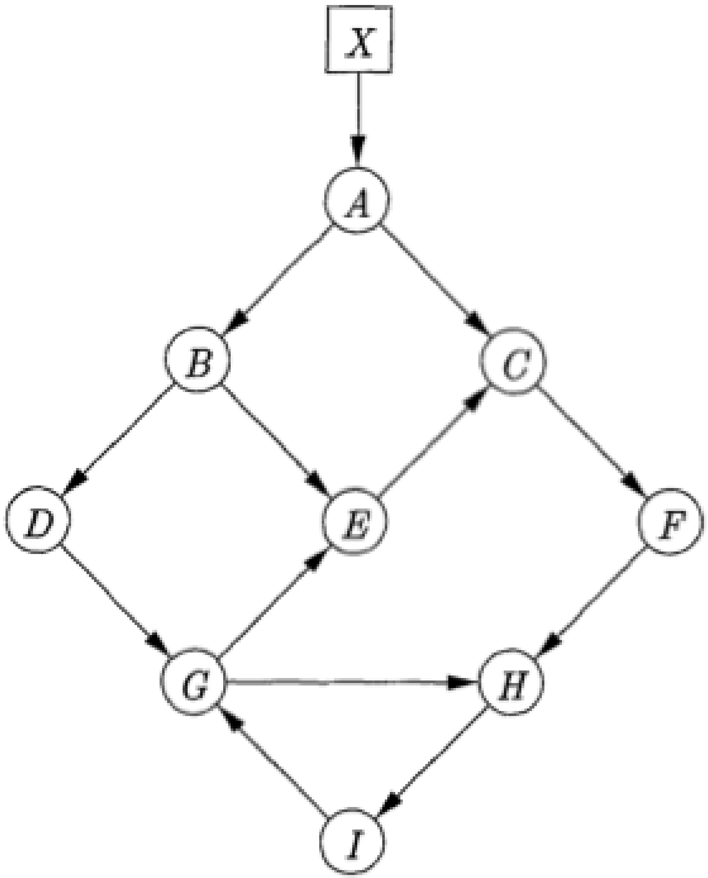
- Such a situation is equivalent to to a memory leak if we use reference counting for garbage collection, since then this garbage and any structures like it are never deallocated.

Example
(from Aho Ullman book)

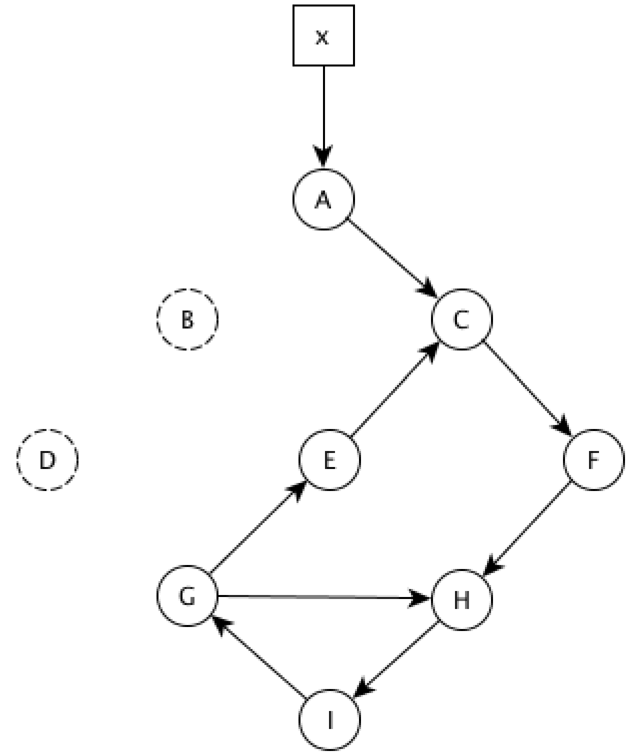


- What happens to the reference counts of the objects if the pointer from A to B is deleted?

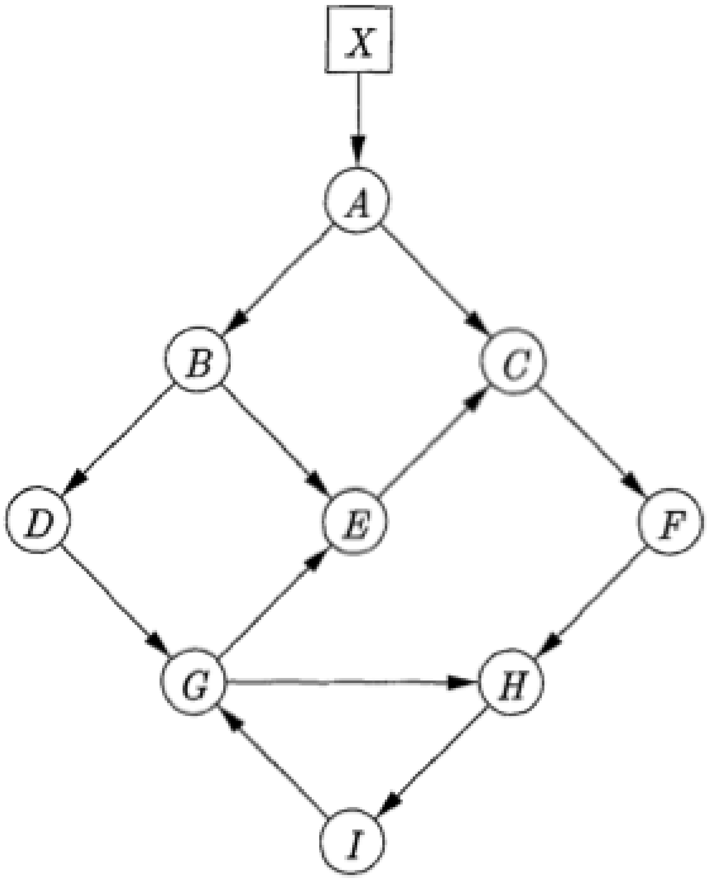
Example
(from Aho Ullman book)



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

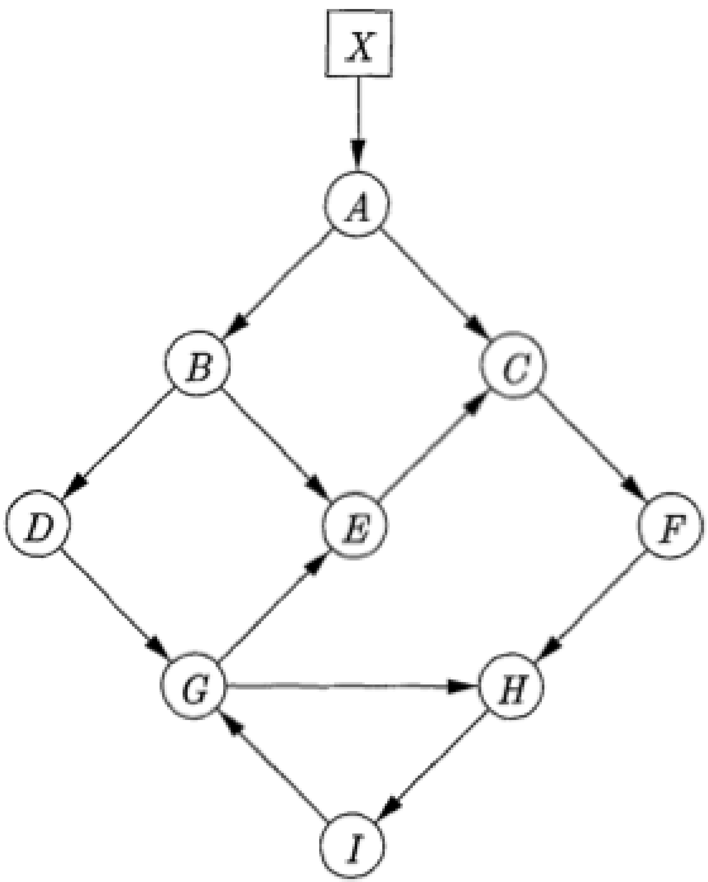


Example
(from Aho Ullman book)

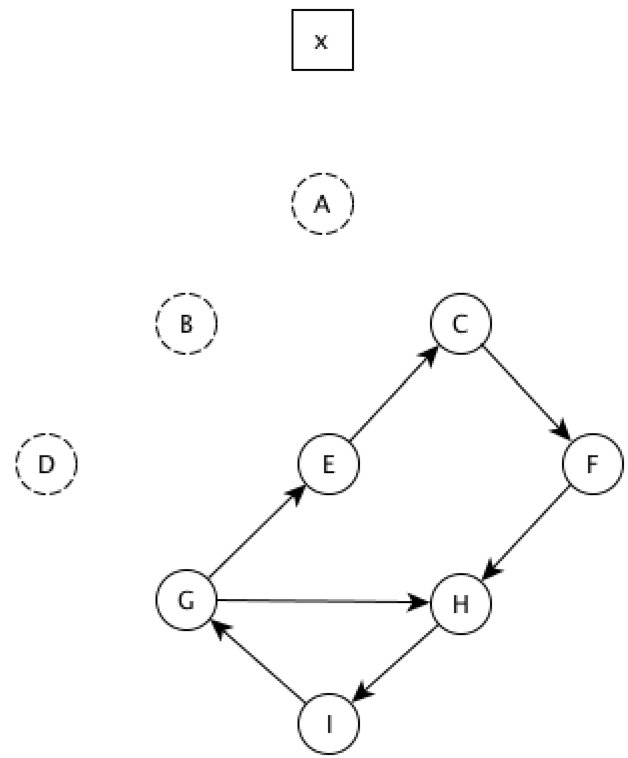


- What happens to the reference counts of the objects if the pointer from X to A is deleted?

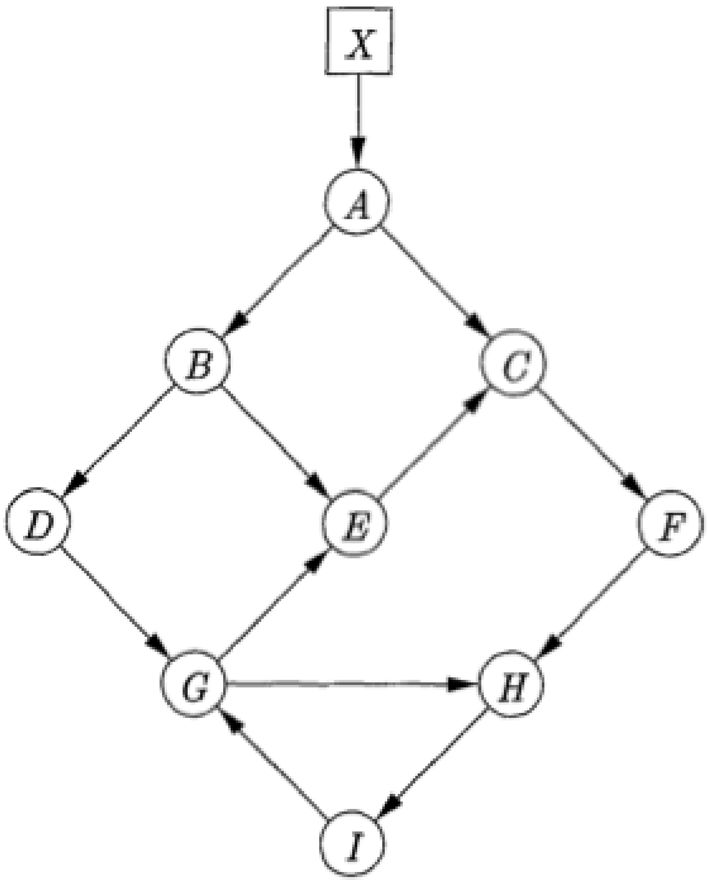
Example
(from Aho Ullman book)



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

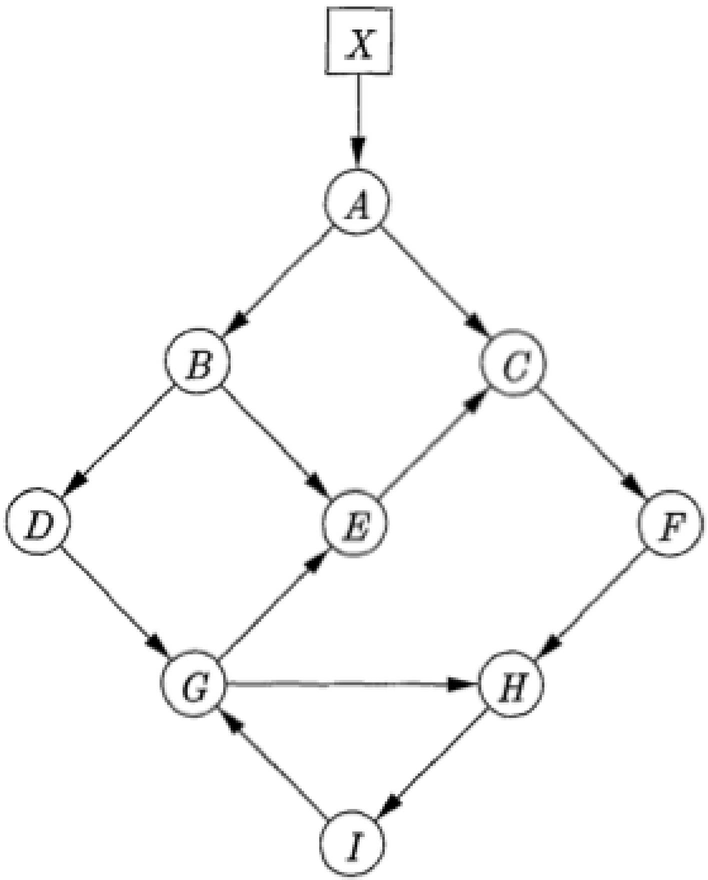


Example
(from Aho Ullman book)



- 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?