# CS100 Python
# Introduction to Programming

## Lecture 27. Garbage Collection

Fu Song

School of Information Science and Technology

ShanghaiTech University

# Learning Objectives

- Understand
  - Memory Management
  - Automatic Memory Mangagement
  - Garbage
  - Python Memory Management
  - Python Garbage Collection
  - Other common garbage collection

# Memory Management

In C/C++, 分配内存

- malloc/new is used to allocate memory
- free/delete is used to deallocate memory
- Memory bugs
  - forgetting to free unused memory (memory leak)
  - dereferencing a dangling pointer (use freed memory)
  - free freed memory
  - overwriting parts of a data structure by accident
- Memory bugs are hard to find: a bug can lead to a visible effect far away in time and program text from the source

# Automatic Memory Management

- **Automatically deallocate memory**
- This is an old problem: since the 1950s for LISP
- There are several well-known techniques for performing completely automatic memory management
  - Reference counting (runtime): Python, PHP, scripting languages
  - Mark-and-Sweep (runtime): Java, C#, Go
  - Object ownership + lifetime (compile time): Rust, C++11 Smart pointer
  - Region-based (compile time): Cyclone

# The Basic Idea

- When an object is created, unused space is automatically allocated
  - In C++, new objects are created by new className(...)
  - In Python, new objects are created by className(...)
- After a while there is **no more unused space**
- Some memory occupied by objects that will never be used again should be deallocated

How to determine whether objects will never be used again or not ?

# The Basic Idea

- How to determine whether objects will never be used again or not ?
- In general it is impossible to tell
  程序探试的
- Heuristic algorithms are used to find many (not all) objects that will never be used again
- **Key observation**: a program can use only the objects that it can find:

$$x = Class_1(\ldots)$$

$$x = Class_2(\ldots)$$

Instance object of $Class_1$ will never be used again after binding x to the instance object of $Class_2$

# Garbage

- An object x is reachable if and only if the object x is bound to a name
- Starting from all the names, we can find all the reachable objects
- An unreachable object can never be referred by the program
- All unreachable objects are called garbage
- All garbage can be deleted

Note: some objects which will never be used again may not be garbage

# Garbage

```
x = [1,2,3]
y = [4,5,6]
x.extend(y)
# y is never used later
```

After x.extend(y): there are two objects:

- [4,5,6]: bound to y, reachable, hence not garbage, but never used
- [1,2,3,4,5,6] : bound to x, reachable, hence not garbage

```
x = [1,2,3]
y = [4,5,6]
x.extend(y)
y = None
# y is never used later
```

After y=None: there are two objects:

- [4,5,6]: not reachable, hence garbage
- [1,2,3,4,5,6] : bound to x, reachable, hence not garbage

# Python Memory Management

- Python interpreter is implemented in C, called CPython
- In CPython, memory in low-level is managed via malloc/free
- Everything in Python is an object
- Dynamic Python's nature requires a lot of small memory allocations/deallocations
- To speed-up memory operations and reduce fragmentation, Python uses a special manager on top of the general-purpose allocator, called PyMalloc

存储残片

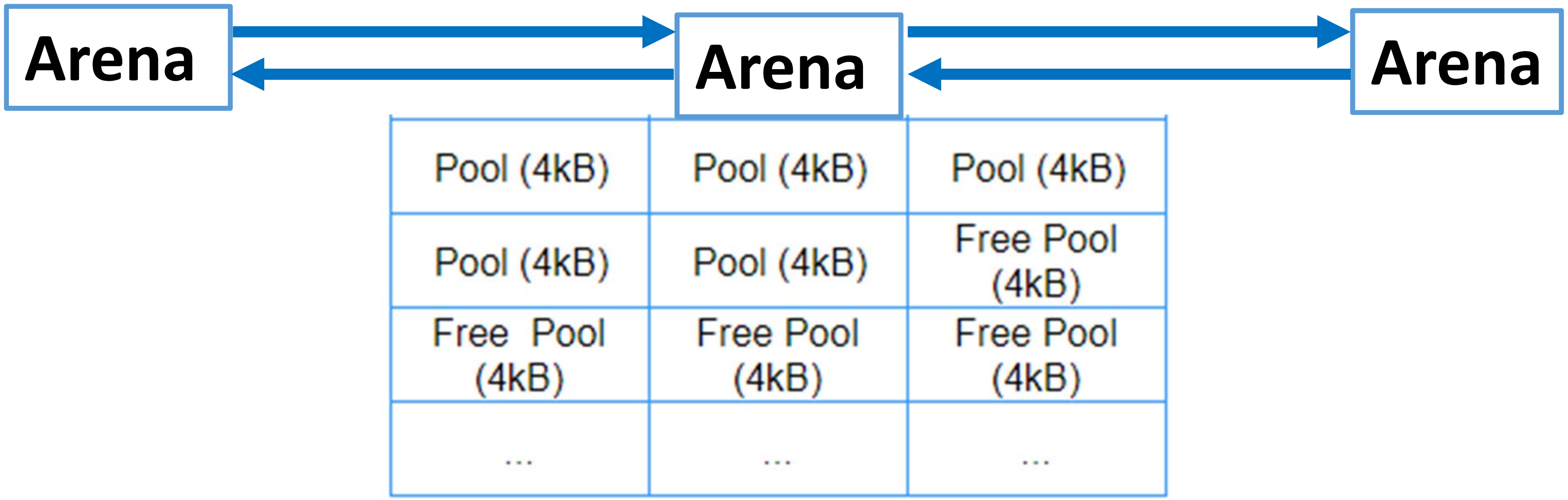
```
    _____   ______   ______       ________
   [ int ] [ dict ] [ list ] ... [ string ]       Python core         |
+3 | <----- Object-specific memory -----> | <-- Non-object memory --> |
    _______________________________       |                           |
   [   Python's object allocator   ]      |                           |
+2 | ####### Object memory ####### | <------ Internal buffers ------> |
    ______________________________________________________________    |
   [          Python's raw memory allocator (PyMem_ API)          ]   |
+1 | <----- Python memory (under PyMem manager's control) ------> |   |
    __________________________________________________________________
   [    Underlying general-purpose allocator (ex: C library malloc)   ]
 0 | <------ Virtual memory allocated for the python process -------> |
   =========================================================================
    _______________________________________________________________________
   [                OS-specific Virtual Memory Manager (VMM)               ]
-1 | <--- Kernel dynamic storage allocation & management (page-based) ---> |
    __________________________________   __________________________________
   [                                  ] [                                  ]
-2 | <-- Physical memory: ROM/RAM --> | | <-- Secondary storage (swap) --> |
```

# Small object allocation

- Larger objects (>SMALL_REQUEST_THRESHOLD) are routed to standard C allocator
- To reduce overhead for small objects less than SMALL_REQUEST_THRESHOLD (512 bytes), Python sub-allocates big blocks of memory
- Small object allocator uses three levels of abstraction
  - Arena
  - Pool
  - Block

# Arena

- **Arena is a chunk of 256kB memory allocated on the heap, which provides memory for 64 pools**
- **All arenas are linked using doubly linked list**
- **Pools are carved off on-demand**
- **An arena gets fully released iff all the pools in it are empty**



# Arena

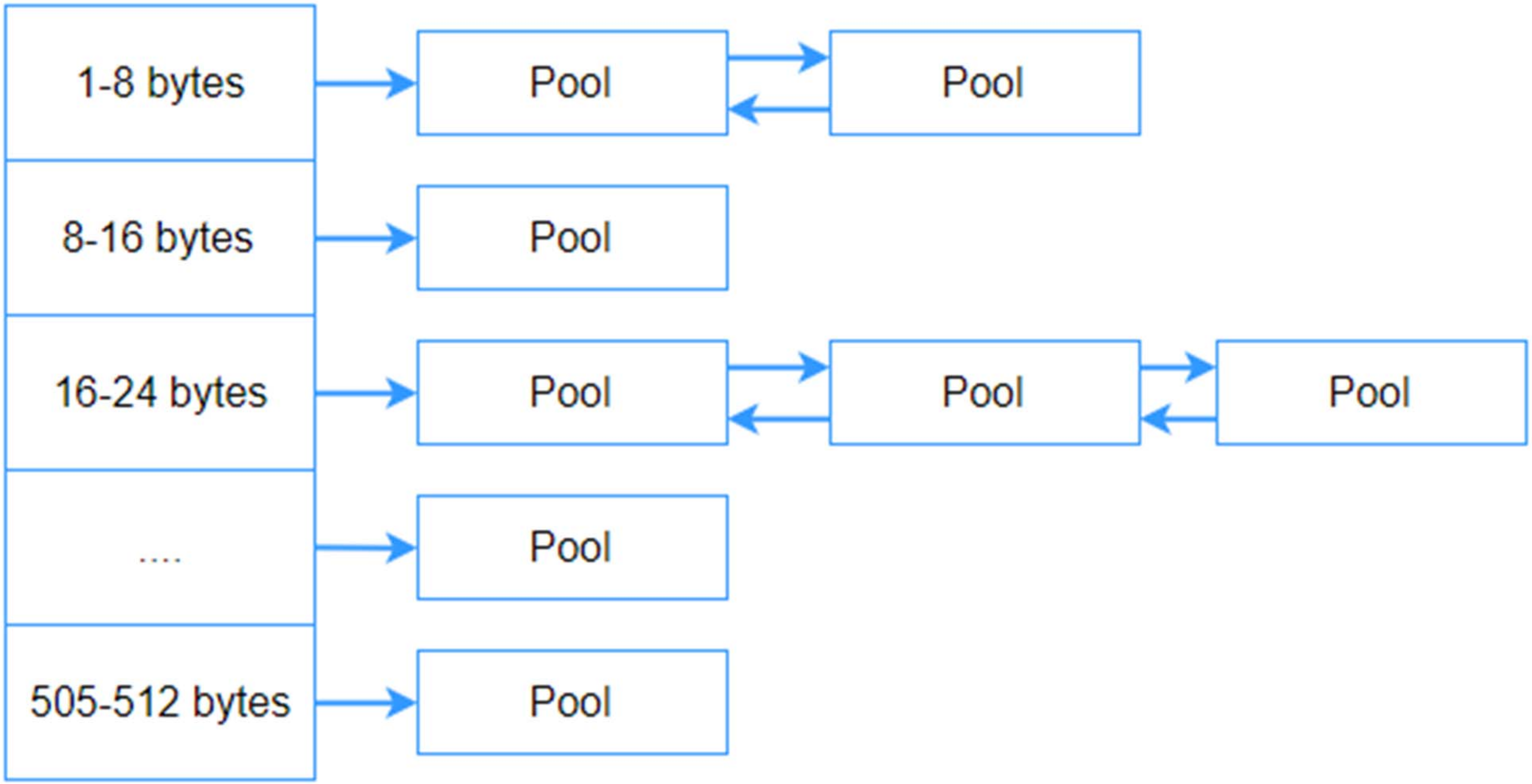
```
struct arena_object {
  uintptr_t address;  //address of the arena
  block* pool_address;
  uint nfreepools;    //number of available pools
  uint ntotalpools;   // total number of pools
  struct pool_header* freepools;
  struct arena_object* nextarena;
  struct arena_object* prevarena;
};
```

- **pool_address points to the next pool to be carved off**
- **freepools points to the singly linked list of available pools**
- **nextarena and prevarena are used to point to next and previous arena (doubly linked list)**

# Pool

- Pool is a collection of blocks of the same size
- Normally, the size of the pool is equal to the size of a memory page, i.e., 256Kb/64=4Kb
- Limiting pool to the fixed size of blocks helps with fragmentation
- If an object gets destroyed, the memory manager can fill this space with a new object of the same size
- Blocks within pools are again carved out as needed

# Pool




- **Pools** of the **same sized blocks** are linked together using **doubly linked list**

# Pool

```
struct pool_header {  //Pool for small blocks
  union {
     block *_padding;
     uint count; } ref;// number of allocated blocks
  block *freeblock; // pool's free list head
  struct pool_header *nextpool; // next pool
  struct pool_header *prevpool; // previous pool
  uint arenaindex; // index into arenas of base addr
  uint szidx;      // block size class index
  uint nextoffset; // offset (bytes) to free block
  uint maxnextoffset; // largest valid nextoffset
};
```

- **freeblock points to the start of a singly-linked list of free blocks within the pool**

# Pool

- **Each pool has three states:**
  - used: partially used, neither empty nor full
  - full: all the pool's blocks are currently allocated
  - empty: all the pool's blocks are currently available for allocation
- **When a pool is carved out as needed, the pool is enters used state**
  1. only "the first two" blocks are set up
  2. returning the first such block
  3. setting freeblock to a one-block list holding the second such block

# Pool

- **When a block is carved out from a pool (used state),**
  1. the freeblock points to next free block
  2. if all the pool's blocks are currently allocated, it enter full state
- **When a block is freed,**
  1. it's inserted at the front of its pool's freeblock list,
  2. its pool enters empty state, if all blocks in the pool are free
  3. the empty pool is added into freepool linked list of the arena

Note that pools and blocks are not allocating memory directly, they are using already allocated space from arenas

# Block

- **Block is a chunk of memory of a certain size, vary from 8 to 512 bytes and be a multiple of eight (i.e., 8-byte alignment)**
- **Each block can keep only one Python object of a fixed size**

| Request in bytes | Size of allocated block | size class idx |
|---|---|---|
| 1-8 | 8 | 0 |
| 9-16 | 16 | 1 |
| 17-24 | 24 | 2 |
| 25-32 | 32 | 3 |
| 33-40 | 40 | 4 |
| 41-48 | 48 | 5 |
| ... | ... | ... |
| 505-512 | 512 | 63 |

# Garbage collection in Python

- Usually, you do not need to worry about memory management when the objects are no longer needed Python automatically reclaims the memory from them
- However, understanding how garbage collector works can help you write better Python programs
- Standard CPython's garbage collector has two components
  - the reference counting collector
  - the generational garbage collector, known as gc module

# Note

- Python does not necessarily release the memory back to the Operating System.
- It has a specialized object allocator for small objects (smaller or equal to 512 bytes), which keeps some chunks of already allocated memory for further use in future
- Therefore, if a long-running Python process takes more memory over time, it does not necessarily mean that you have memory leaks.

# Reference Counting

- Reference counting is a simple technique to determine whether an object is garbage or not
- An object becomes garbage when there is no reference to them in a program
- Every object in Python has a reference count
- To keep track of references every object (even integer), reference count that is increased or decreased when a pointer to the object is copied or deleted

# Reference Counting

- In python, all classes (including built-in classes) are extension of PyObject

```
[object.h]
typedef struct _object {
  int ob_refcnt;                    // reference count
  struct _typeobject *ob_type;// type information
} PyObject;
```

# Reference Counting

- When an object is created and bound to some name, the reference count is 1

x = Class(…)

- When the object is copied, the reference count increases 1

1. assignment operator
2. argument passing
3. appending an object to a list
4. ……

e.g., x = y , x.append(y)

# Reference Counting

- When the object is deleted, the reference count decreases 1,
  1. Removed from a list
  2. An object having an attribute points to the object is destroyed
  3. Break of the name binding
  4. Name goes out of scope
  5. …….

     e.g., x= Class(); x = None

- The object becomes garbage when its the reference count reaches 0

# Reference Counting

- For each garbage object
  1. CPython automatically calls the object-specific deallocation function
  2. If the object contains references to other objects, then their reference count is decremented too
  3. Thus other objects may be deallocated in turn
  4. For example, when a list is deleted the reference count of all its items is decreased

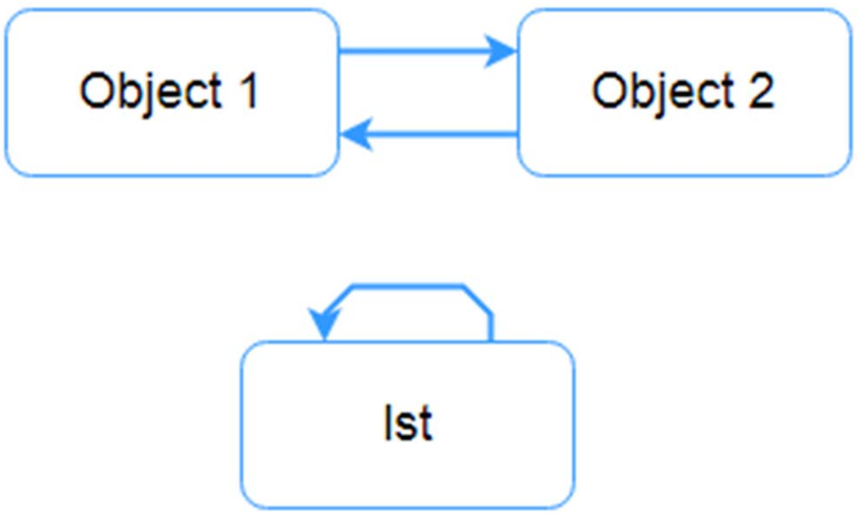
# Reference Counting

```
import sys
foo = []  # 2 refcnt, foo var and getrefcount
print(sys.getrefcount(foo))
def bar(a):
    print(sys.getrefcount(a)) # 4 refcnt, foo var,
                    # function argument, getrefcount
                    # and Python's function stack


bar(foo)
print(sys.getrefcount(foo)) # 2 refcnt,function
                            # scope is destroyed
x = [1,2,3]
foo.append(x)
print(sys.getrefcount(x)) # 3 refcnt, x var,
                          # foo list and getrefcount
```

# Reference cycles

- Classical reference counting has a fundamental problem
  - it cannot detect reference cycles
  - reference count for such objects is always at least 1
  - hence will never be deallocated
- Reference cycles can only occur in container objects (i.e., in objects which can contain other objects)



# Reference cycles

```
class Node:
    def __init__(self, node = None):
        self.next=node
n1 = Node()
n2 = Node(n1)
n1.next = n2
del n1,n2
```

After del n1, n2

- The objects are no longer accessible from Python code
- But, their reference count is 1
- Such objects are still sitting in the memory

# Generational garbage collector

- The reference counting technique handles all non-circular references
- The gc module is responsible for dealing with reference cycles
- Unlike the reference counting, the cyclic GC does not work in real-time and runs periodically
- To reduce the frequency of cyclic GC calls and pauses Cpython, uses various heuristics

# Generational garbage collector

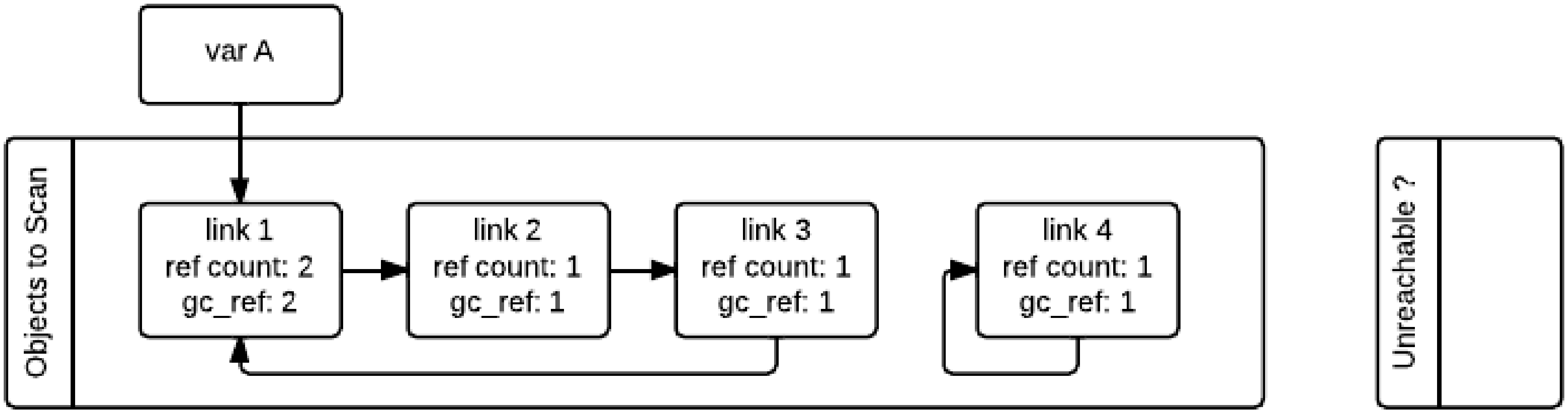
- The GC classifies container objects into 3 generations
- Every new object starts in the first generation
- If an object survives a garbage collection round, it moves to the older (higher) generation
- Lower generations are collected more often than higher
- Because most of the newly created objects die young, it improves GC performance and reduces the GC pause time

# Generational garbage collector

- In order to decide when to run, each generation has an individual counter and threshold
- Counter stores the number of object allocations minus deallocations since the last collection
- Every time you allocate a new container object, CPython checks whenever the counter of the first generation exceeds the threshold value, if so Python initiates the collection process
- If two or more generations exceed the threshold, GC chooses the oldest one
- The default threshold values are set to (700, 10, 10)

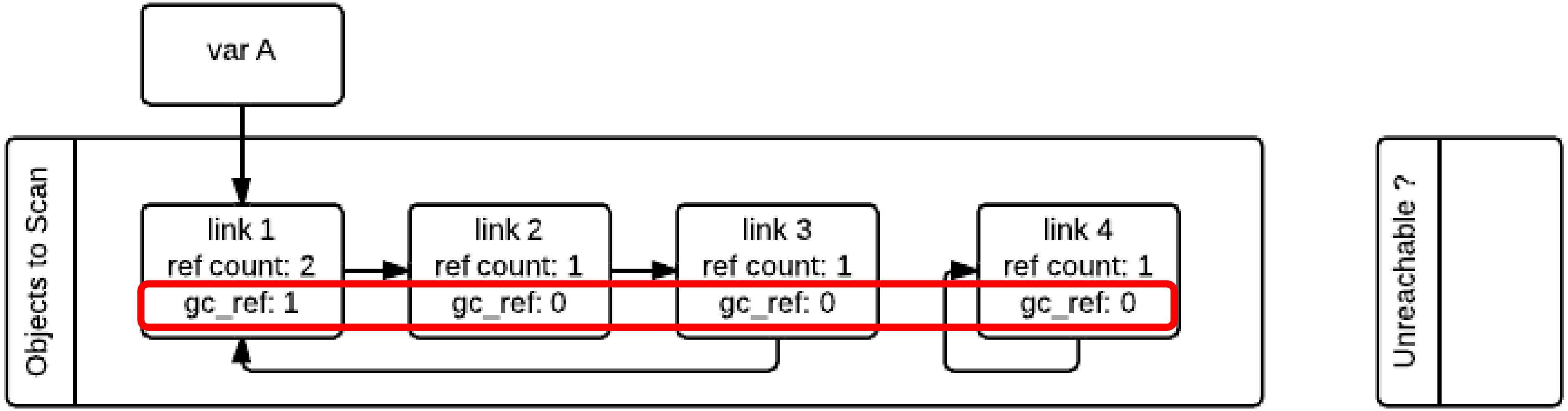
# How to find reference cycles

1. When GC starts, all the container objects to scan are in a list. As most objects are reachable, it is more efficient to assume all objects are reachable and move them to an unreachable list if needed
   - Each object container also has a gc_ref field initially set to the reference count



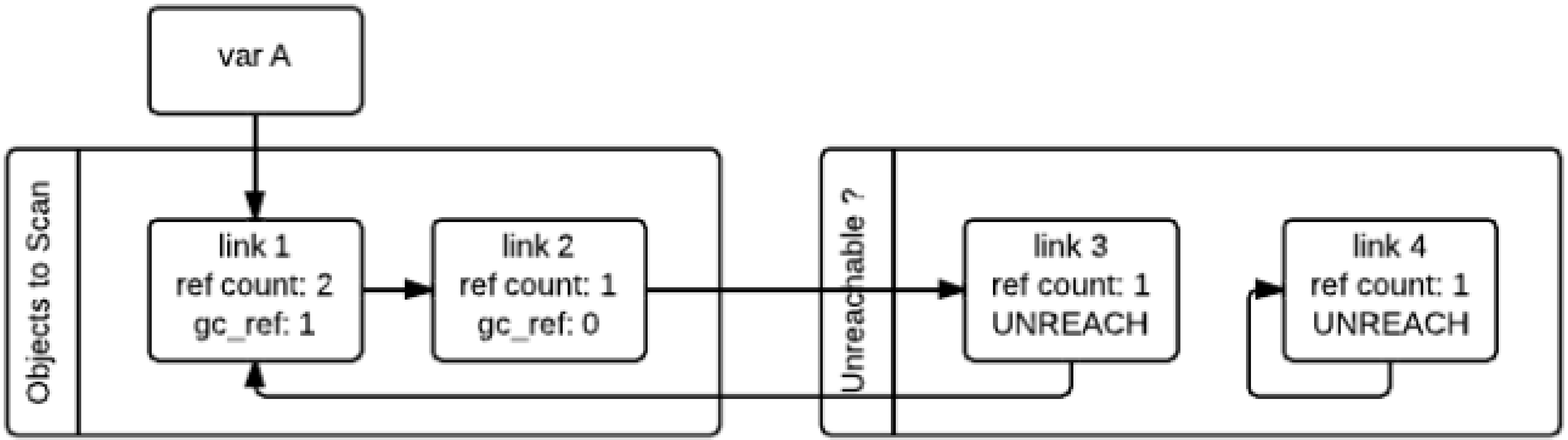
# How to find reference cycles

2. GC then goes through each container object and decrements by 1 the gc_ref of any other object it is referencing (first scan)



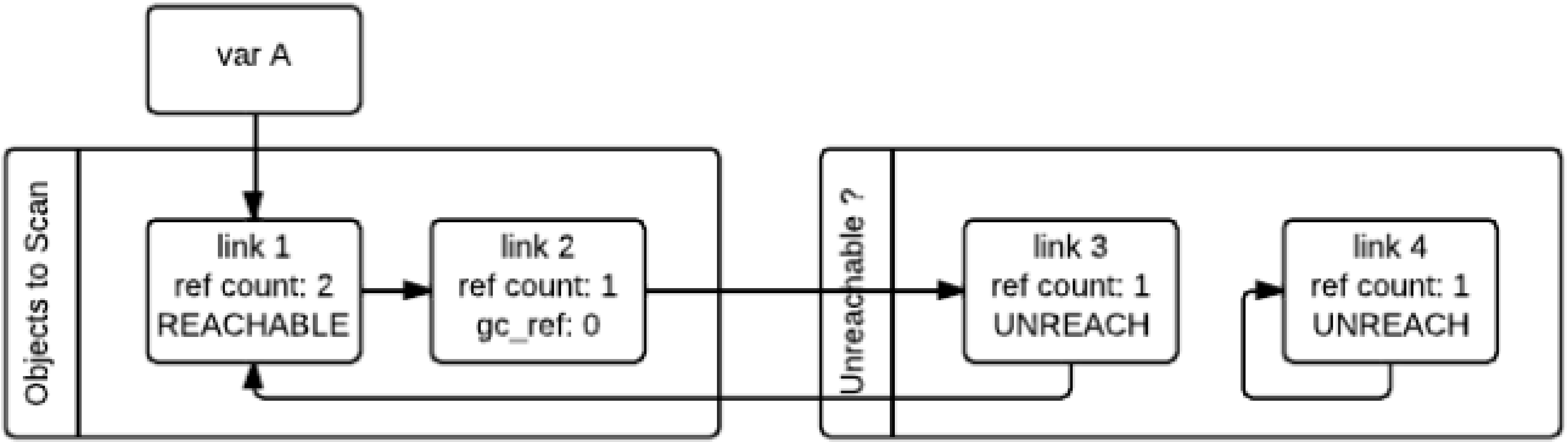
# How to find reference cycles

3. GC scans again the container objects. The objects whose gc_ref is zero are moved to the tentatively unreachable list (second scan)
   - GC processed the “link 3” and “link 4” objects but hasn’t processed “link 1” and “link 2” yet.



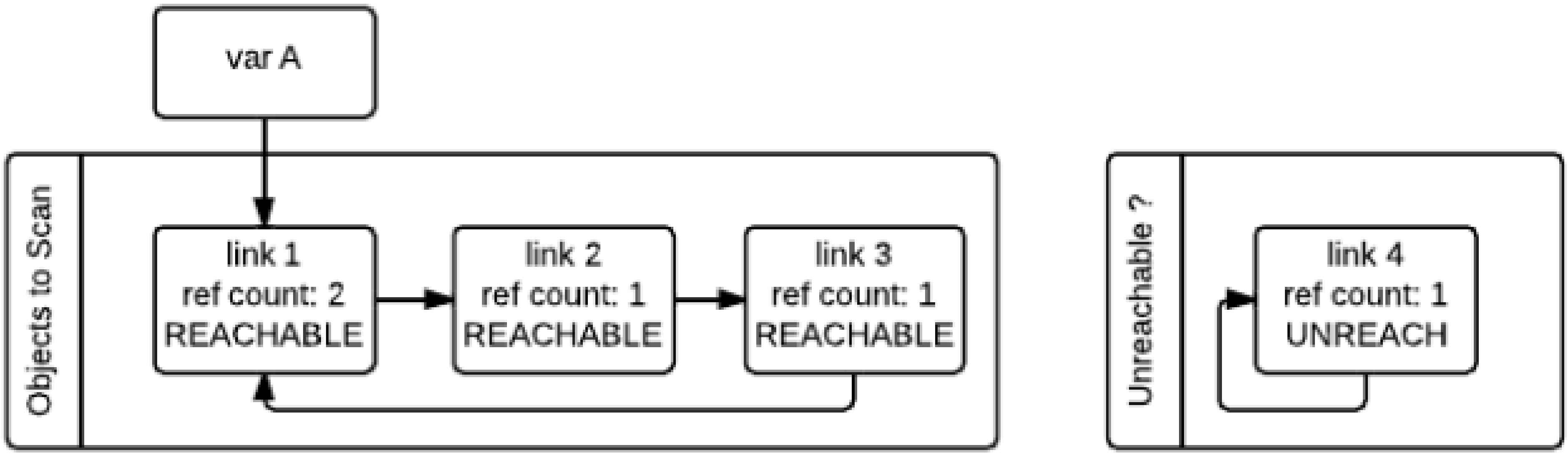
# How to find reference cycles

4. The objects whose gc_ref is non-zero are marked as GC_REACHABLE
   - assume that the GC scans next the “link 1” object.



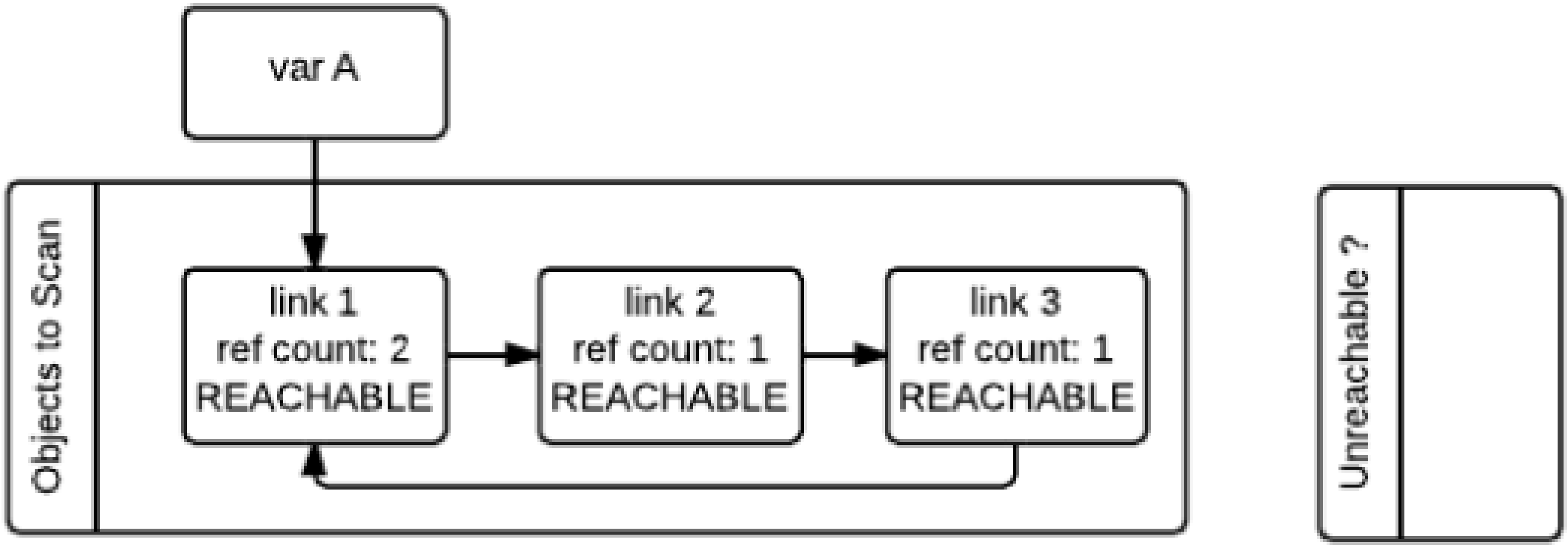
# How to find reference cycles

5. When an object is masked as GC_REACHABLE, it traverses its references to find all the objects that are reachable from it, marking them as GC_REACHABLE,
   - “link 2” and “link 3” below as they are reachable from “link 1”.  Because “link 3” is reachable after all, it is moved back to the original list.



# How to find reference cycles

6. When all the objects are scanned, the container objects in the unreachable list are really unreachable and can thus be garbage collected



# Performance tips

- Cycles can easily happen in real life, typically in graphs, linked lists or in structures, in which you need to keep track of relations between objects
- If your program has intensive workload and requires low latency, you should avoid reference cycles as possible
- To avoid circular references in your code, you need to use weak references, which are implemented in the weakref module
- Unlike the usual references, the weakref.ref doesn't increase the reference count and returns None if an object was destroyed

# gc module

- The standard gc module provides a lot of useful helpers that can help in
  1. disable GC, gc.disable()
  2. enable GC, gc.enable()
  3. use it manually, gc.collect()
  4. debugging, set debugging flags to DEBUG_SAVEALL, all unreachable objects found will be appended to gc.garbage list

```python
import gc
gc.set_debug(gc.DEBUG_SAVEALL)
class Node:
    pass

n1 = Node()
n2 = Node()
n3 = Node()
n1.next,n2.next,n3.next = n2,n3,n1

del n1,n2,n3
gc.collect()
for i in gc.garbage:
    if isinstance(i,Node):
        print(i)
```

```
<__main__.Node object at 0x02D28530>
<__main__.Node object at 0x02D65F10>
<__main__.Node object at 0x02D65ED0>
```

```python
import gc
gc.set_debug(gc.DEBUG_SAVEALL)
class Node:
    pass

n1 = Node()
n2 = Node()
n3 = Node()
n1.next,n2.next,n3.next = n2,n3,n1

#del n1,n2,n3
gc.collect()
for i in gc.garbage:
    if isinstance(i,Node):
        print(i)
```

No output

# Learning Objectives

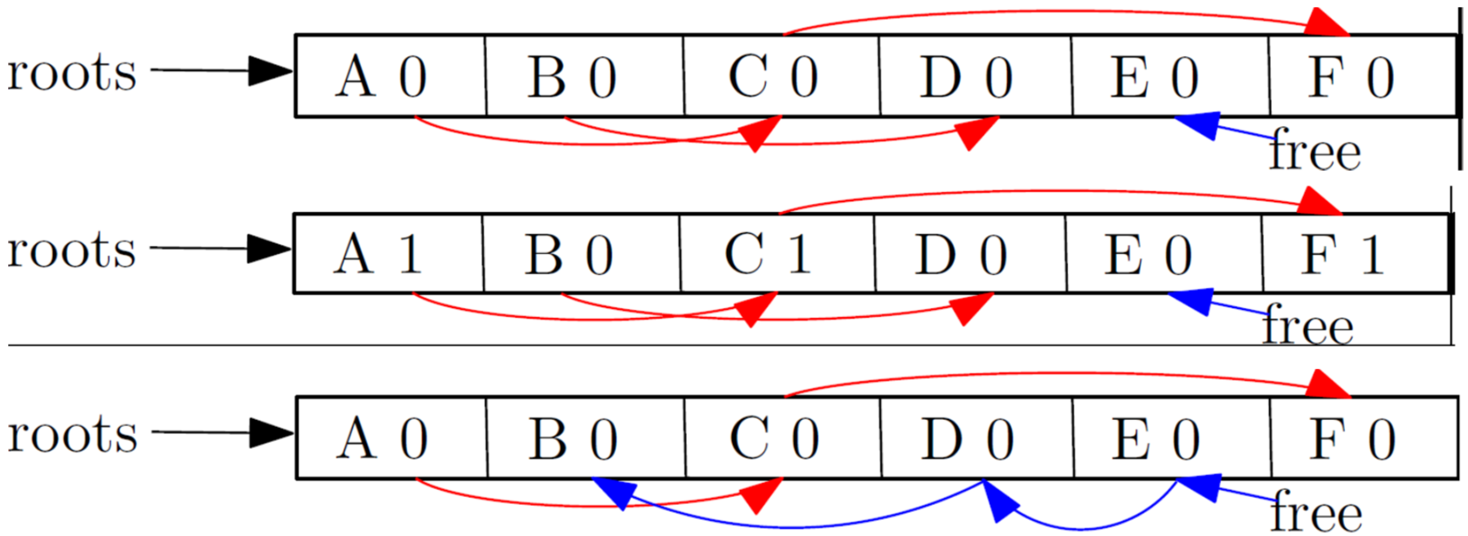
- **Understand**
  - **Memory Management**
  - **Automatic Memory Mangagement**
  - **Garbage**
  - **Python Memory Management**
  - **Python Garbage Collection**
  - **Other common garbage collection**
    - Mark and Sweep
    - Copying

# Mark and Sweep

- When memory runs out, GC executes two phases
  - the mark phase: traces reachable objects
  - the sweep phase: collects garbage objects
- Every object has an extra bit: the mark bit
  - reserved for memory management
  - initially the mark bit is 0
  - set to 1 for the reachable objects in the mark phase

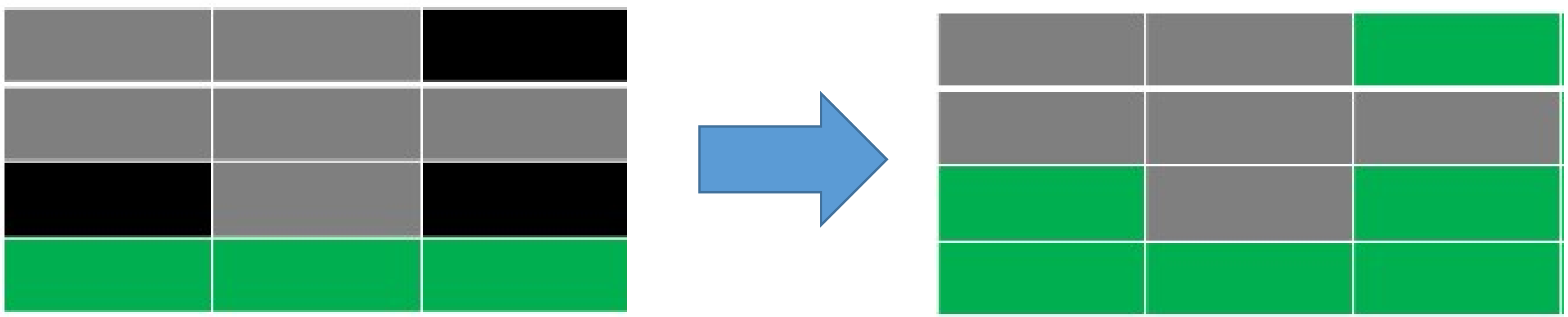
After mark phase

After sweep phase



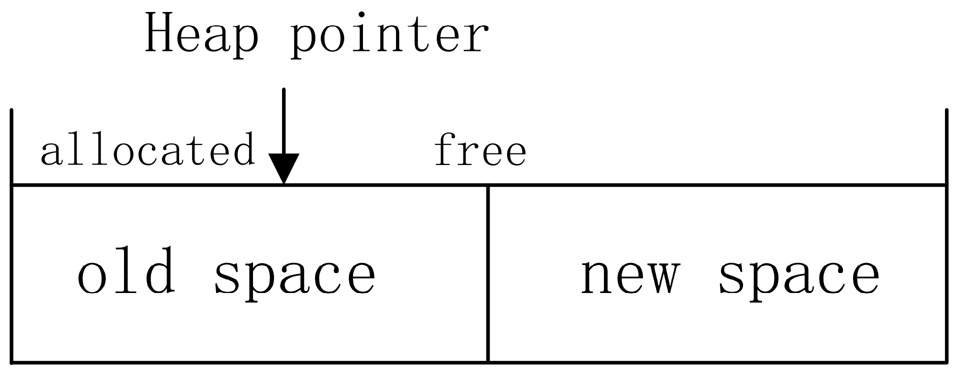
# Mark and Sweep: Evaluation

- Disadvantage
  - Mark and sweep can fragment the memory
- Advantage: objects are not moved during GC
  - no need to update the pointers to objects
  - works for languages like Java



# Copying

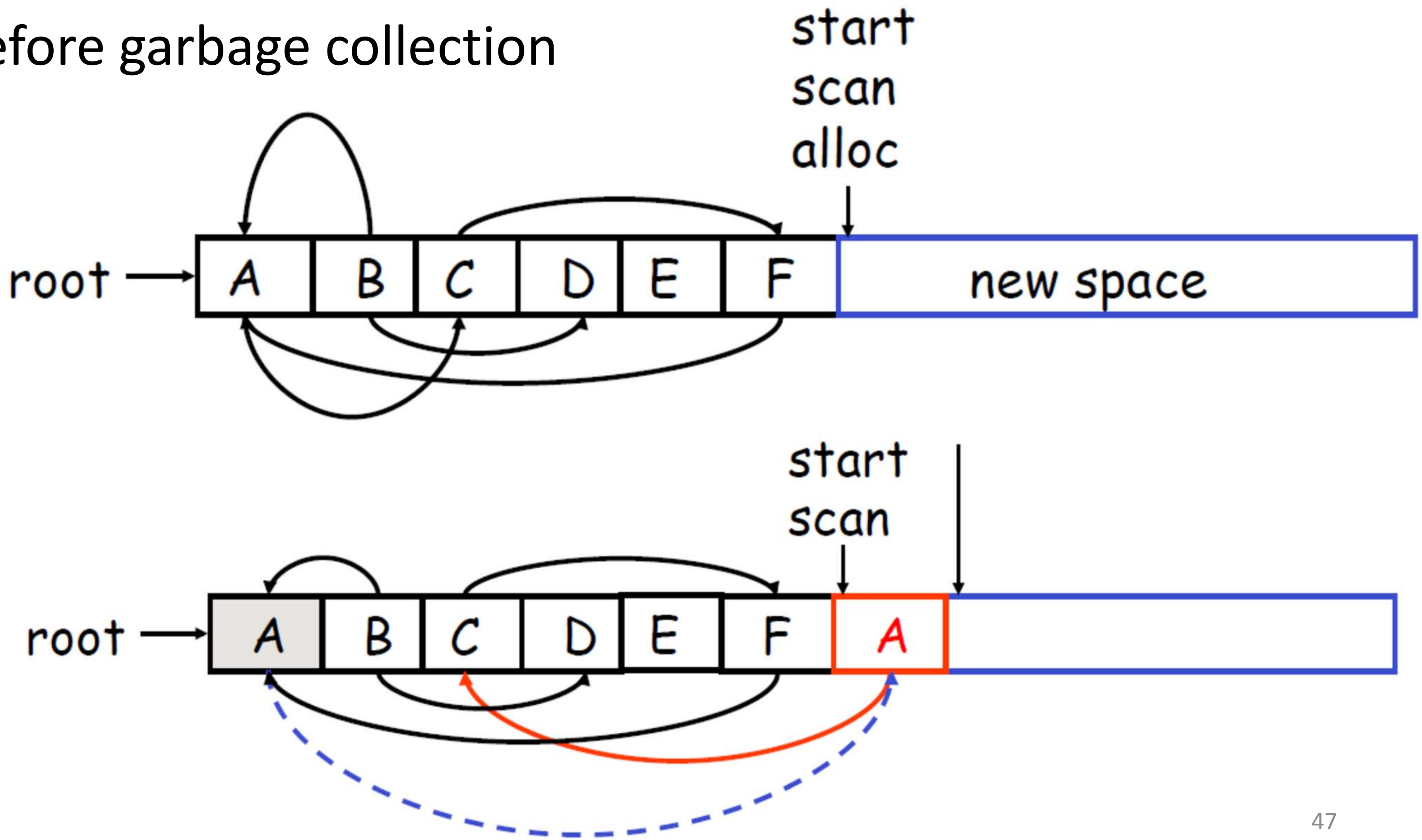
- Memory is organized into two equal areas
  - Old space: used for allocation
  - New space: used as a reserve for GC



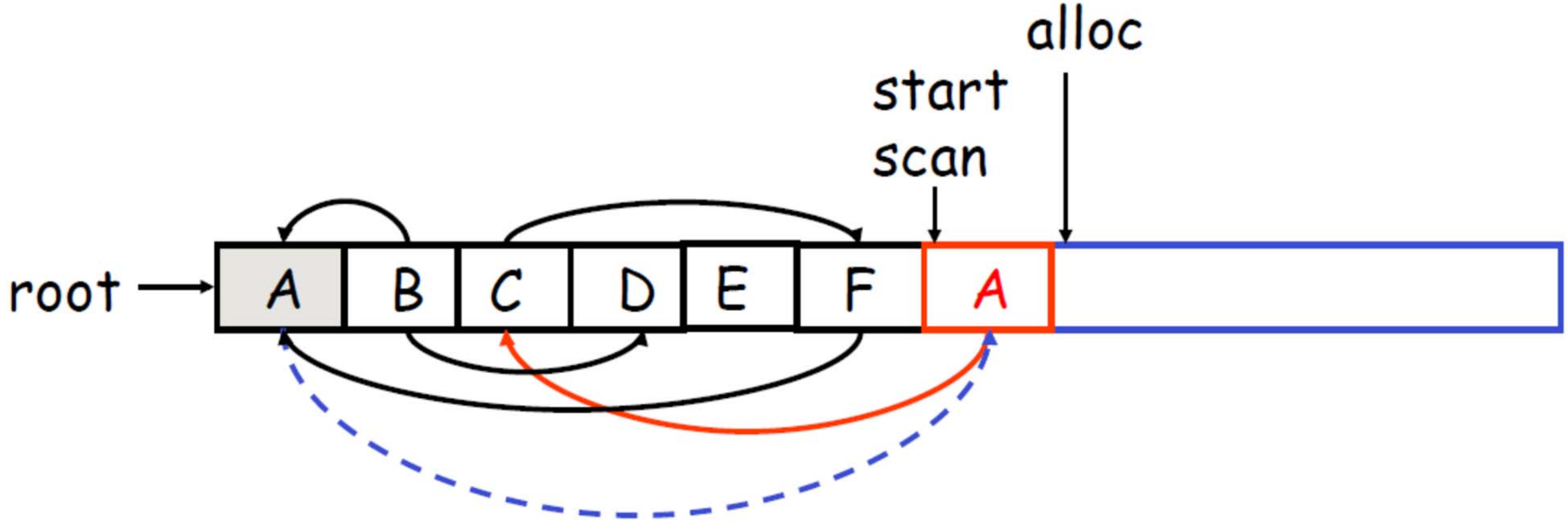

- The heap pointer points to the next free word in the old space
  - Allocation just advances the heap pointer

# Example

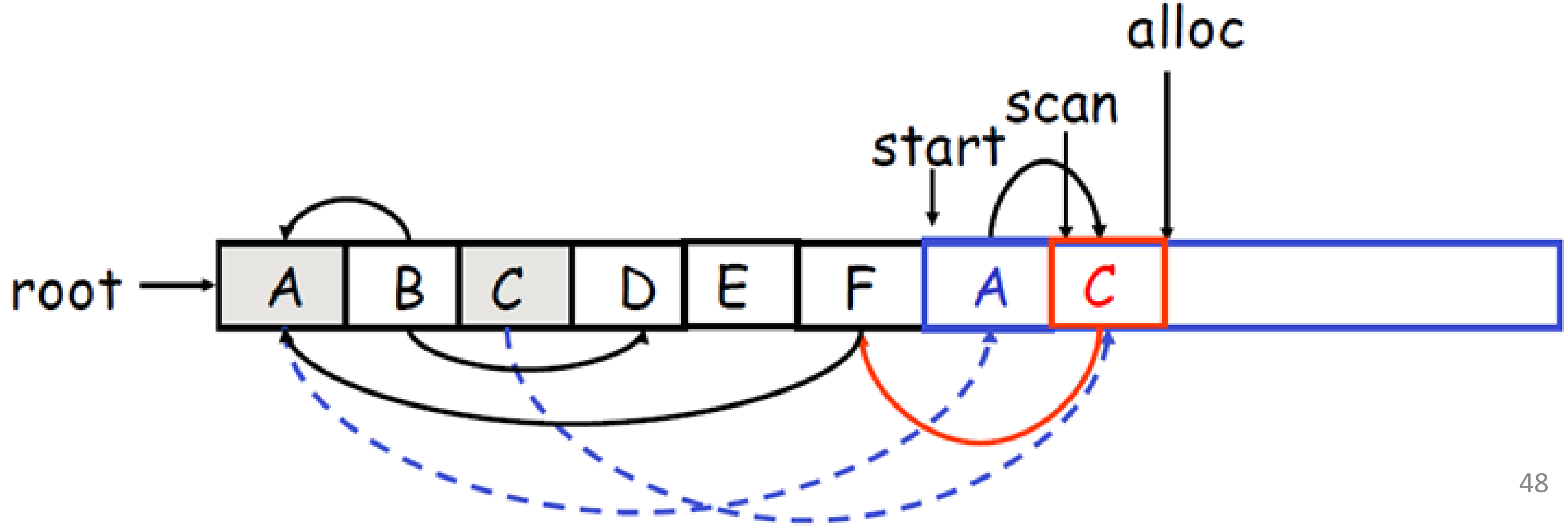
Before garbage collection



# Example

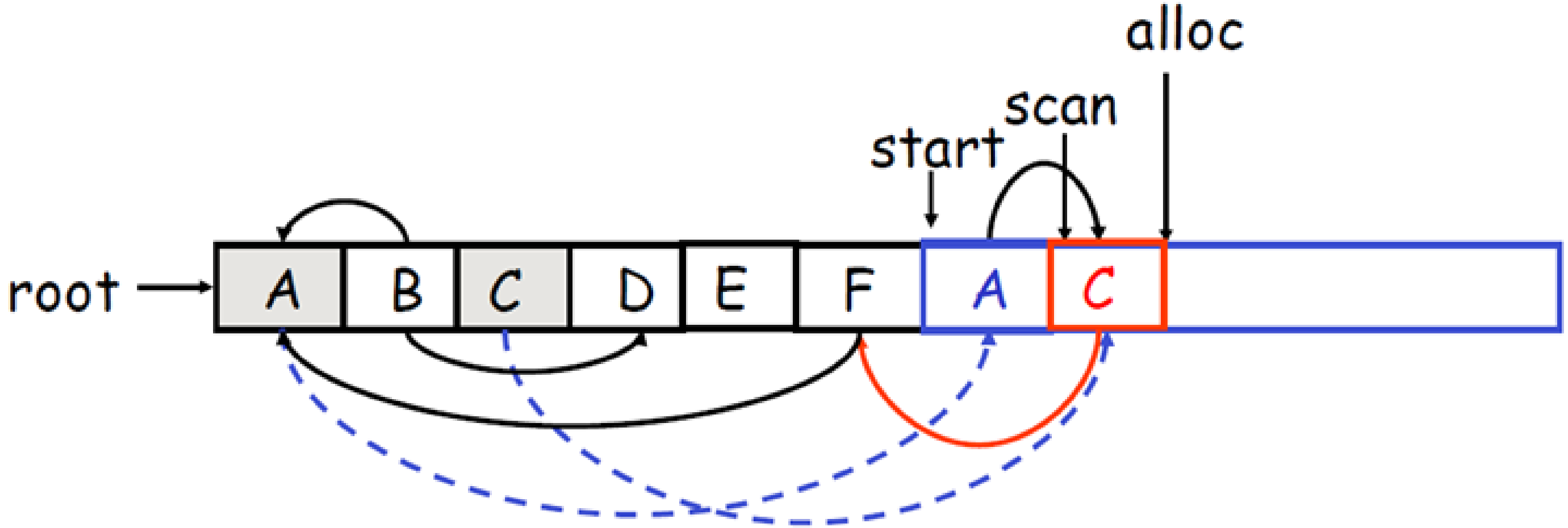


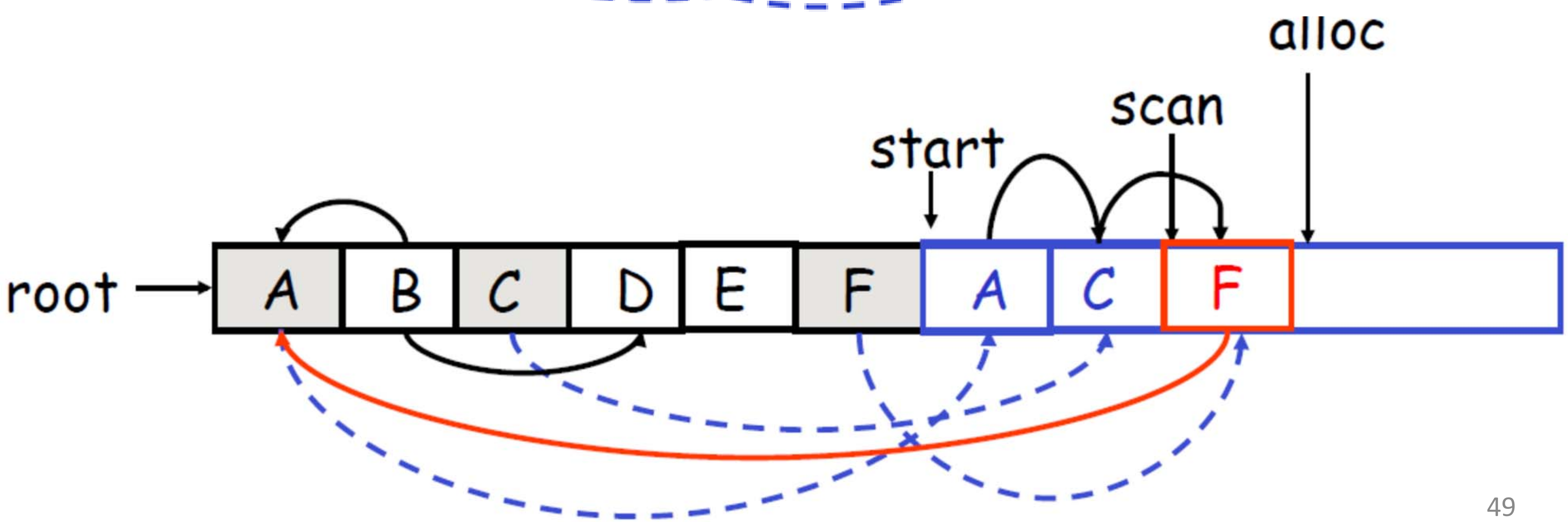
alloc
start
scan
root
A
B
C
D
E
F
A

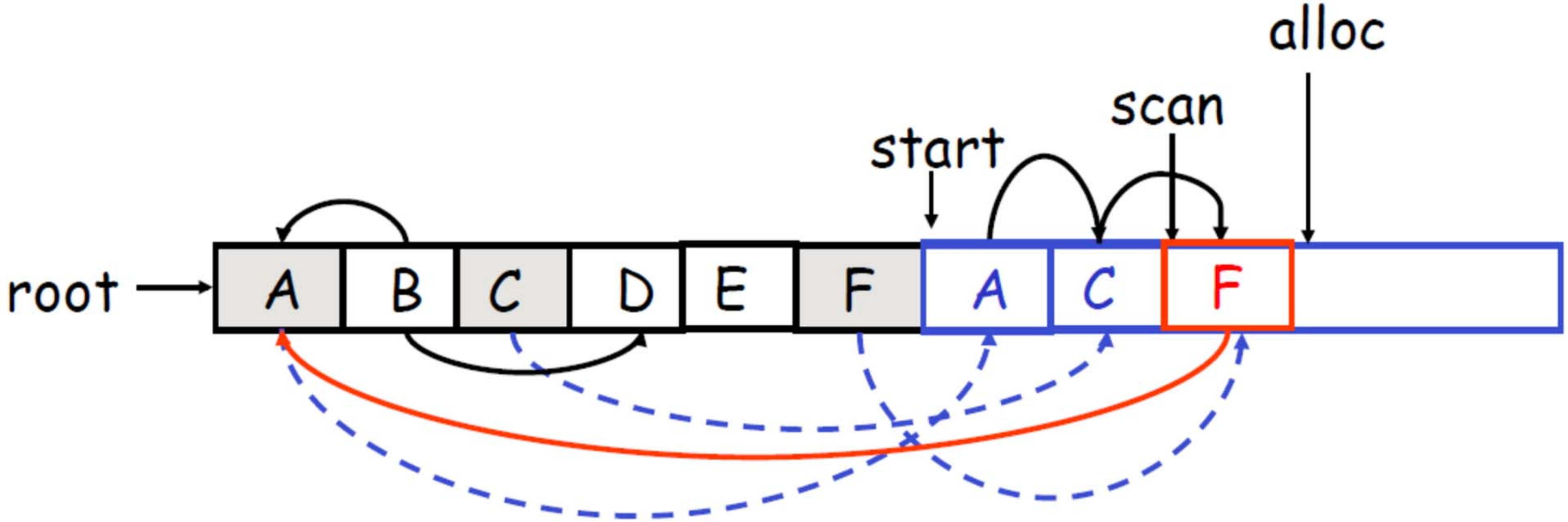
alloc
scan
start
root
A
B
C
D
E
F
A
C

# Example

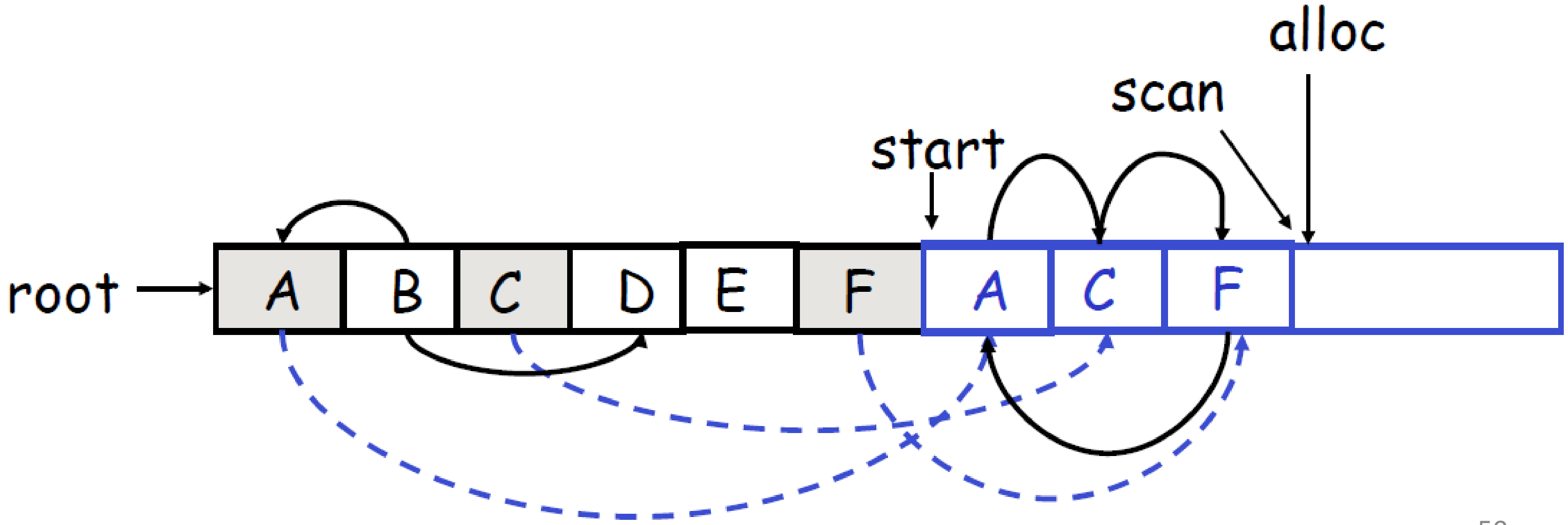

alloc
scan
start
root
A
B
C
D
E
F
A
C



alloc
scan
start
root
A
B
C
D
E
F
A
C
F

# Example

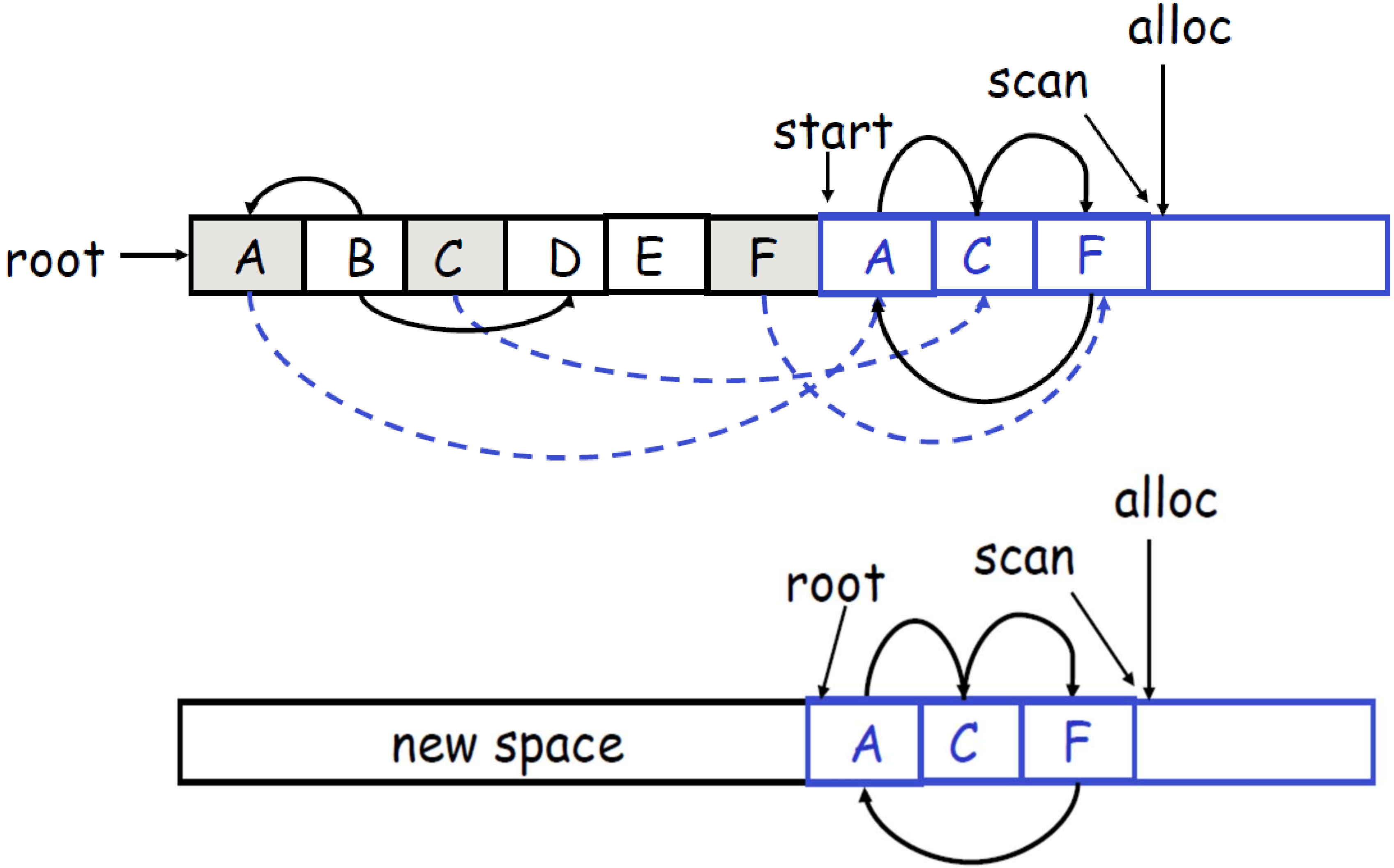


alloc
scan
start
root
A
B
C
D
E
F
A
C
F

alloc
scan
start
root
A
B
C
D
E
F
A
C
F

# Example


alloc
scan
start
root
A
B
C
D
E
F
A
C
F
alloc
scan
root
new space
A
C
F

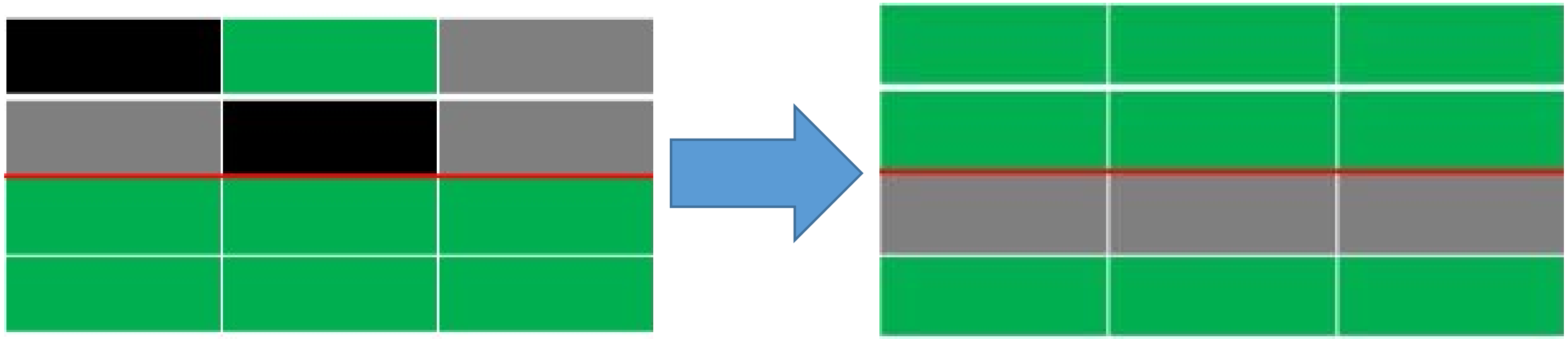
# Copying: Evaluation

**Advantages:**

1. Only touches live data, while mark and sweep touches both live and dead data
2. Copying is generally believed to be the fastest GC technique
3. No fragmentation

**Disadvantages:**

- Requires (at most) twice the memory space



# Recap

- Understand
  - Memory Management
  - Automatic Memory Mangagement
  - Garbage
  - Python Memory Management
  - Python Garbage Collection
  - Other common garbage collection