Implicit Memory Management: Garbage Collection

- ***Garbage collection:*** **automatic reclamation of heap-allocated storage—application never has to free**


```
void foo() {
    int *p = mall
    return; /* p
}
```

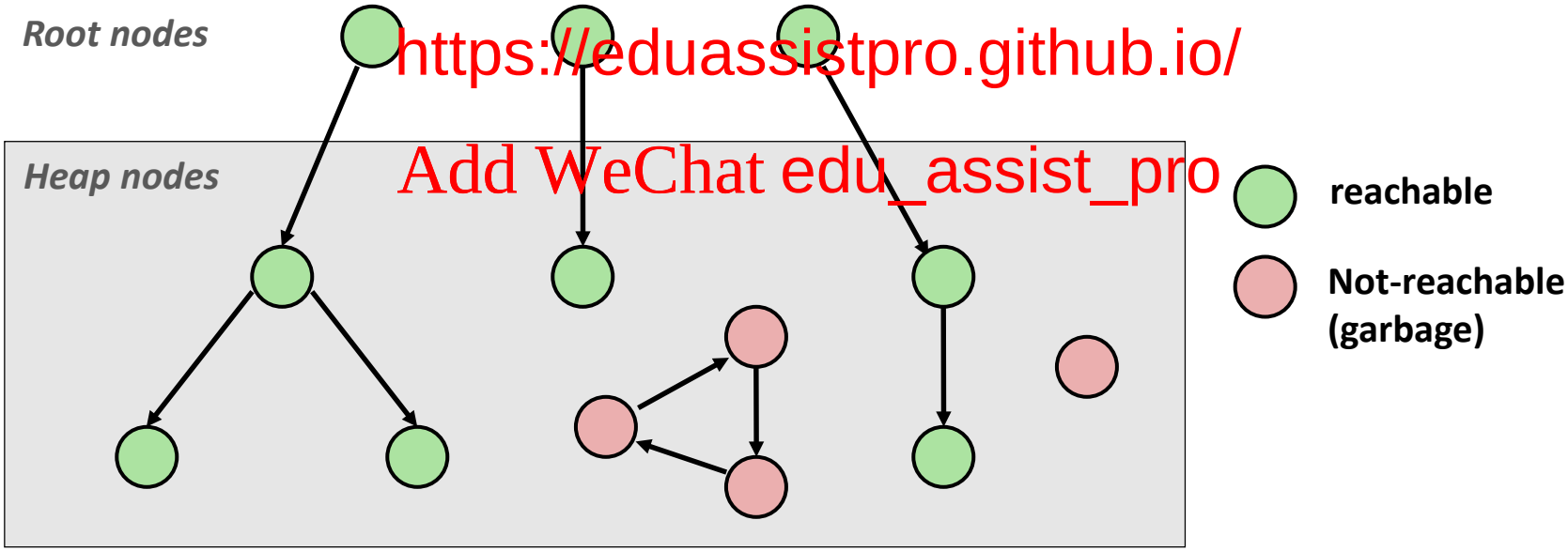

- **Common in functional languages, scripting languages, and modern object oriented languages:**
  - Python, Lisp, ML, Java, Perl, Mathematica
  - Requires a runtime environment (interpreter)
- **Variants ("conservative" garbage collectors) exist for C and C++**
  - However, cannot necessarily collect all garbage

Garbage Collection

- **How does the memory manager know when memory can be freed?**
  - In general we cannot know what is going to be used in the future since it depends on conditionals
  - But we can tell that certain blocks cannot be used **if there are no pointers to the**
- **Must make certain assumptions** **ters**
  - Memory manager can distinguish p on-pointers
  - All pointers point to the start of a block
  - Cannot hide pointers
    (e.g., by coercing them to an **`int`**, and then back again)

Memory as a Graph

- **We view memory as a directed graph**
  - Each block is a node in the graph
  - Each pointer is an edge in the graph
  - Locations not in the heap that contain pointers into the heap are called ***root*** nodes (e.g. register locations, the stack, global variables)



A node (block) is *reachable* if there is a path from any root to that node.

Non-reachable nodes are *garbage* (cannot be needed by the application)

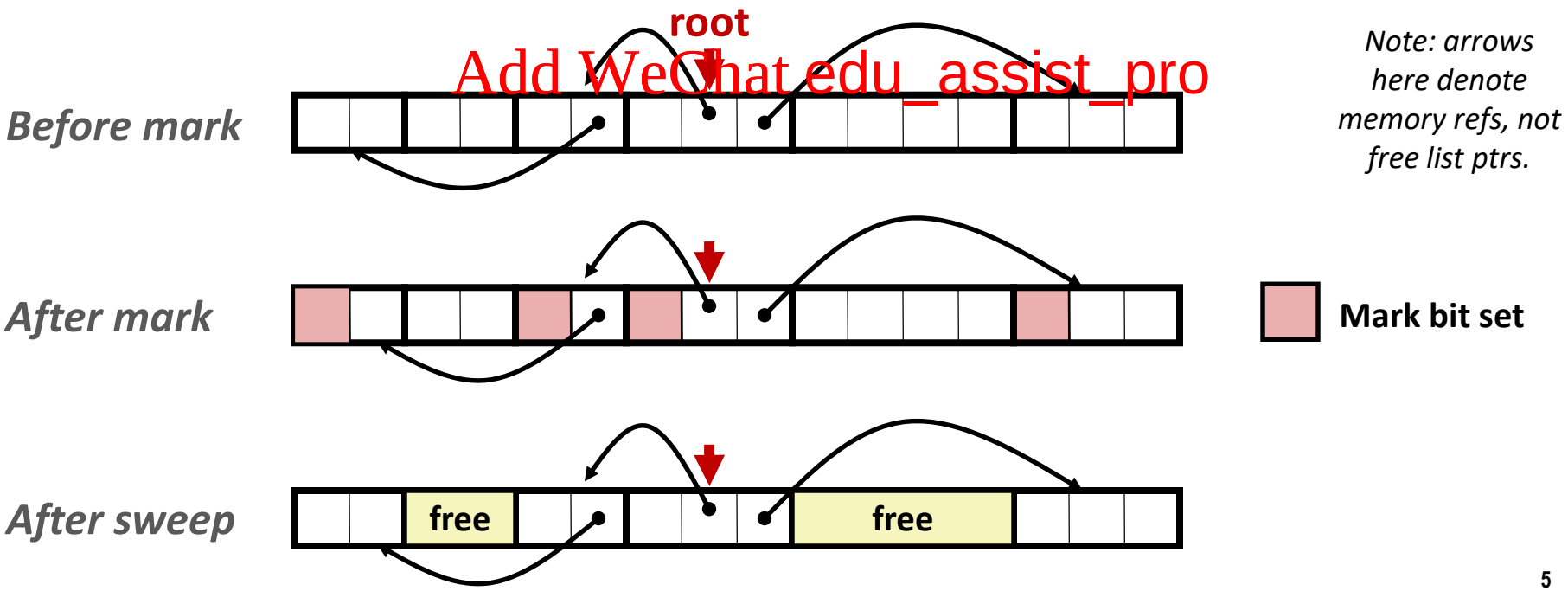
Reachable Blocks

```
class myclass:
    x = 5

m1 = myclass()

def foo():
    m2 = mycl

foo()
```


- m1 is a root node (global var)
- m2 is a root node, but only while foo() executes
- After foo() is done, m2's object is non-reachable

Mark and Sweep Collecting

- **Can build on top of malloc/free package**
  - Allocate using `malloc` until you "run out of space"
- **When out of space:**
  - Use extra ***mark bit*** in the head of each block
  - ***Mark:*** Start at r eachable block
  - ***Sweep:*** Scan all not marked



Mark and Sweep (cont.)

Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // do nothing if not pointer
   if (markBitSet(p)) return;       // check if already marked
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)    // call mark on all words
     mark(p[i]);                                 block
   return;
}
```


Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {
      if markBitSet(p)
         clearMarkBit();
      else if (allocateBitSet(p))
         free(p);
      p += length(p);
}
```