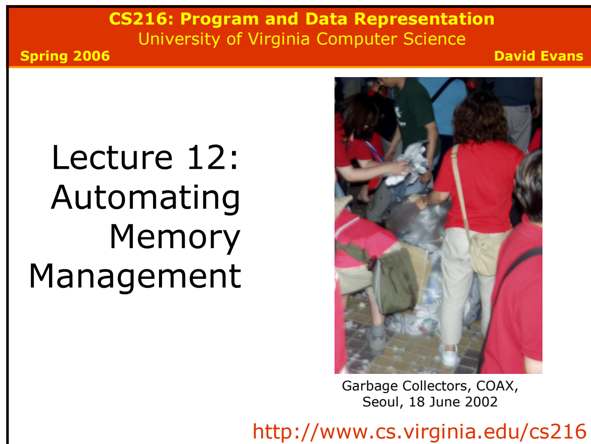
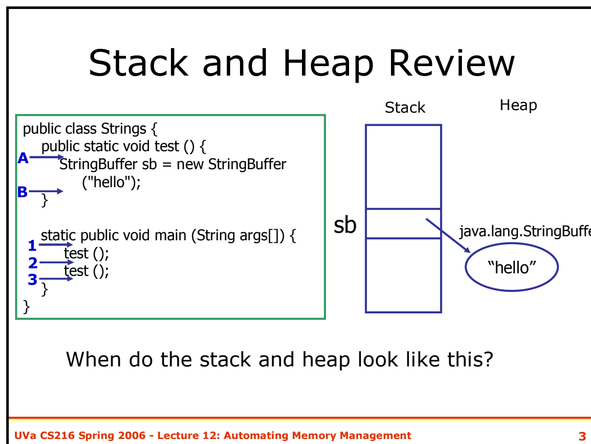
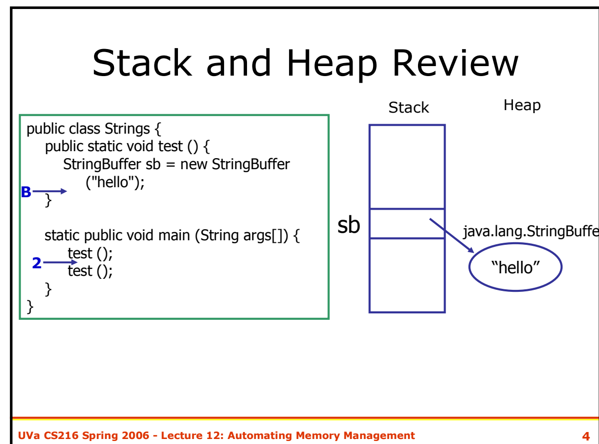
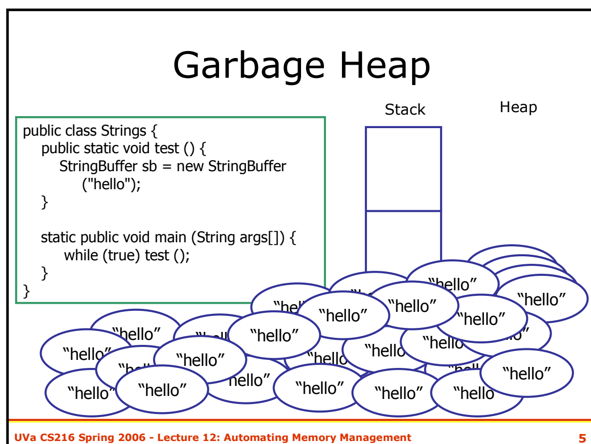


Menu

- Stack and Heap
- Mark and Sweep
- Stop and Copy
- Reference Counting
- Java's Garbage Collector
- Python's Solution







Garbage Collection

- System needs to reclaim storage on the heap used by garbage objects
- How can it identify garbage objects?
- How come we don't need to garbage collect the stack?

Mark and Sweep

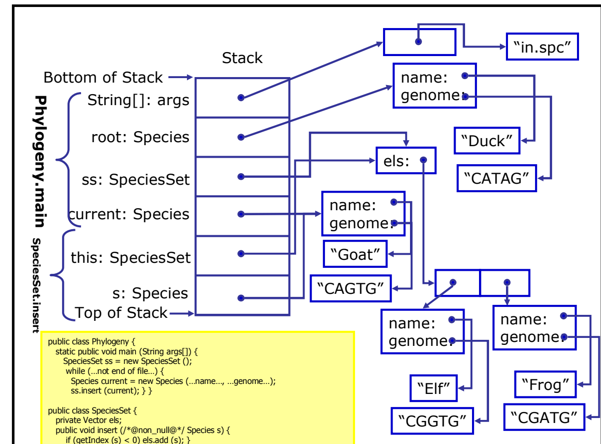
Mark and Sweep

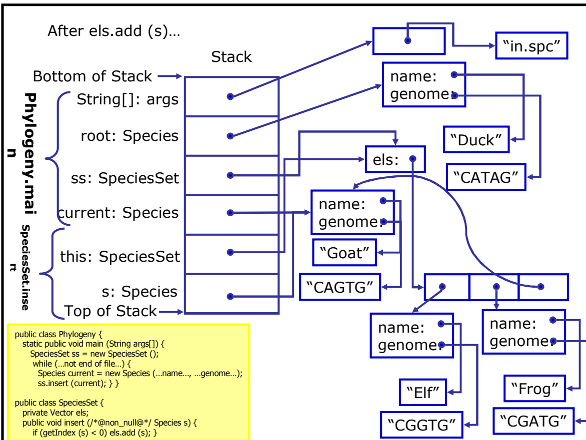
- John McCarthy, 1960 (first LISP implementation)
- Start with a set of root references
- Mark every object you can reach from those references
- Sweep up the unmarked objects

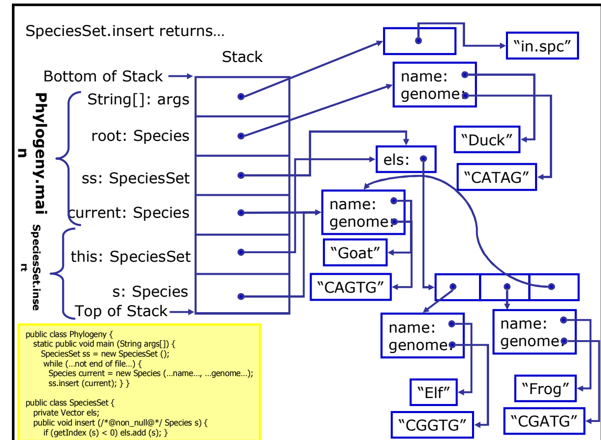
What are the root references?

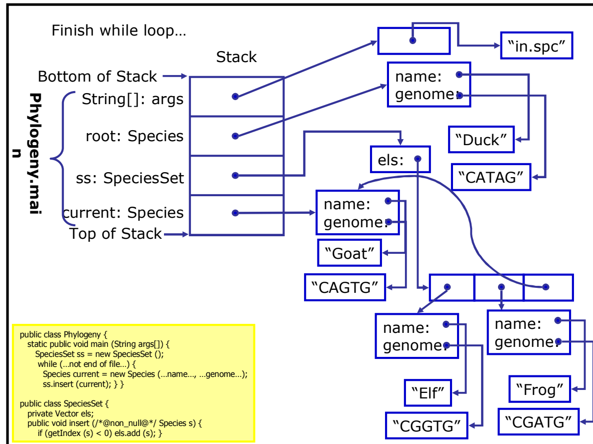
References on the stack.


```
public class Phylogeny {
   static public void main (String args[]) {
      SpeciesSet ss = new SpeciesSet ();
      … (open file for reading)
      while (…not end of file…) {
         Species current = new Species (…name from file…,
                                        …genome from file…);
         ss.insert (current);
   }
}

public class SpeciesSet {
   private Vector els;
   public void insert (/*@non_null@*/ Species s) {
      if (getIndex (s) < 0) els.add (s); }
```

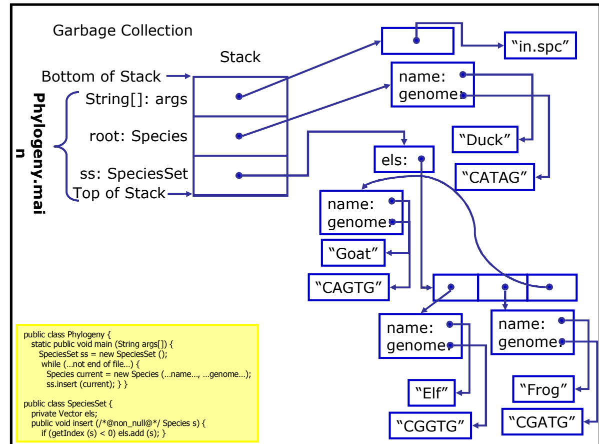





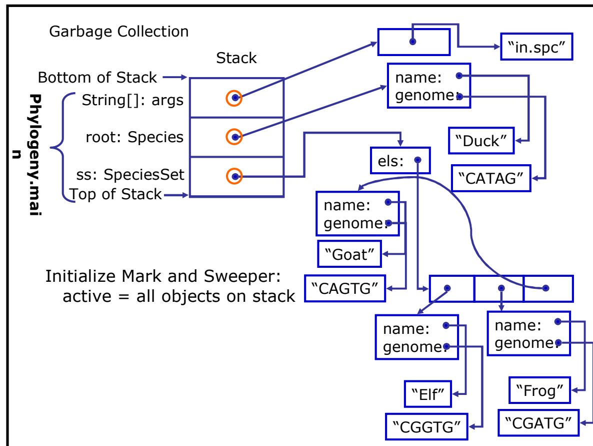




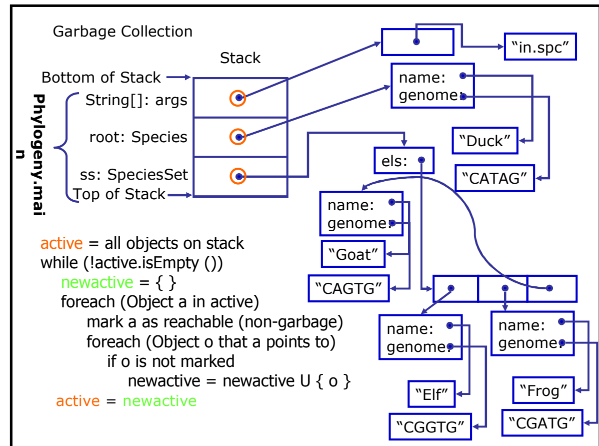
Finish while loop…
Stack
Bottom of Stack
String[]: args
root: Species
ss: SpeciesSet
current: Species
Top of Stack
Phylogeny.main
"in.spc"
name:
genome
"Duck"
"CATAG"
els:
name:
genome
"Goat"
"CAGTG"
name:
genome
name:
genome
"Elf"
"Frog"
"CGGTG"
"CGATG"
public class Phylogeny {
static public void main (String args[]) {
SpeciesSet ss = new SpeciesSet ();
while (…not end of file…) {
Species current = new Species (…name…, …genome…);
ss.insert (current); } }
public class SpeciesSet {
private Vector els;
public void insert (/*@non_null@*/ Species s) {
if (getIndex (s) < 0) els.add (s); }



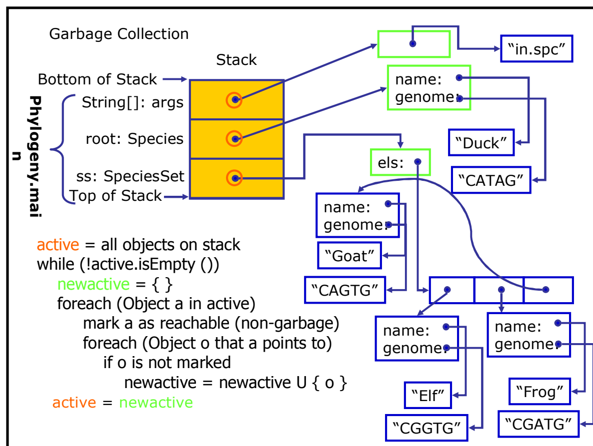
Garbage Collection
Stack
Bottom of Stack
String[]: args
root: Species
ss: SpeciesSet
Top of Stack
Phylogeny.main
"in.spc"
name:
genome
"Duck"
"CATAG"
els:
name:
genome
"Goat"
"CAGTG"
name:
genome
name:
genome
"Elf"
"Frog"
"CGGTG"
"CGATG"
public class Phylogeny {
static public void main (String args[]) {
SpeciesSet ss = new SpeciesSet ();
while (…not end of file…) {
Species current = new Species (…name…, …genome…);
ss.insert (current); } }
public class SpeciesSet {
private Vector els;
public void insert (/*@non_null@*/ Species s) {
if (getIndex (s) < 0) els.add (s); }



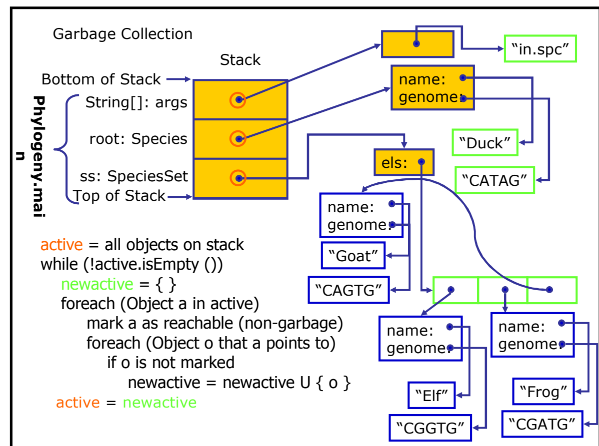
Garbage Collection
Stack
Bottom of Stack
String[]: args
root: Species
ss: SpeciesSet
Top of Stack
Phylogeny.main
"in.spc"
name:
genome
"Duck"
"CATAG"
els:
name:
genome
"Goat"
"CAGTG"
Initialize Mark and Sweeper:
active = all objects on stack
name:
genome
name:
genome
"Elf"
"Frog"
"CGGTG"
"CGATG"



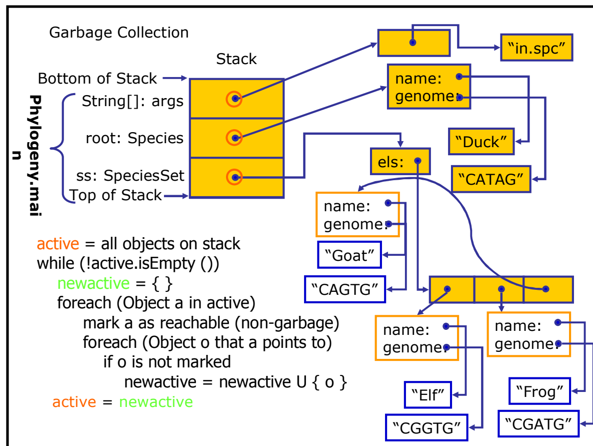
Garbage Collection
Stack
Bottom of Stack
String[]: args
root: Species
ss: SpeciesSet
Top of Stack
Phylogeny.main
"in.spc"
name:
genome
"Duck"
"CATAG"
els:
name:
genome
"Goat"
"CAGTG"
active = all objects on stack
while (!active.isEmpty ())
newactive = { }
foreach (Object a in active)
mark a as reachable (non-garbage)
foreach (Object o that a points to)
if o is not marked
newactive = newactive U { o }
active = newactive
name:
genome
name:
genome
"Elf"
"Frog"
"CGGTG"
"CGATG"

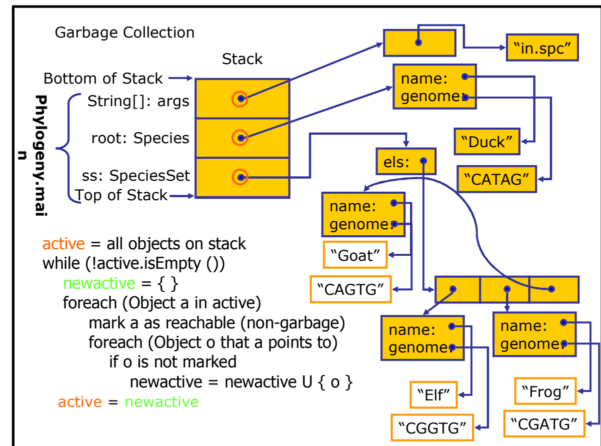


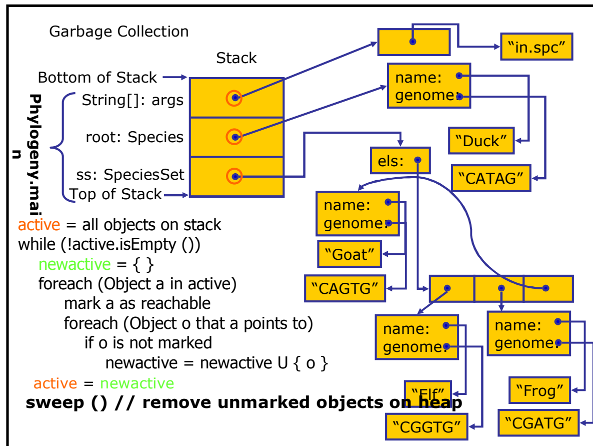
Garbage Collection
Stack
Bottom of Stack
String[]: args
root: Species
ss: SpeciesSet
Top of Stack
Phylogeny.main
"in.spc"
name:
genome
"Duck"
"CATAG"
els:
name:
genome
"Goat"
"CAGTG"
active = all objects on stack
while (!active.isEmpty ())
newactive = { }
foreach (Object a in active)
mark a as reachable (non-garbage)
foreach (Object o that a points to)
if o is not marked
newactive = newactive U { o }
active = newactive
name:
genome
name:
genome
"Elf"
"Frog"
"CGGTG"
"CGATG"

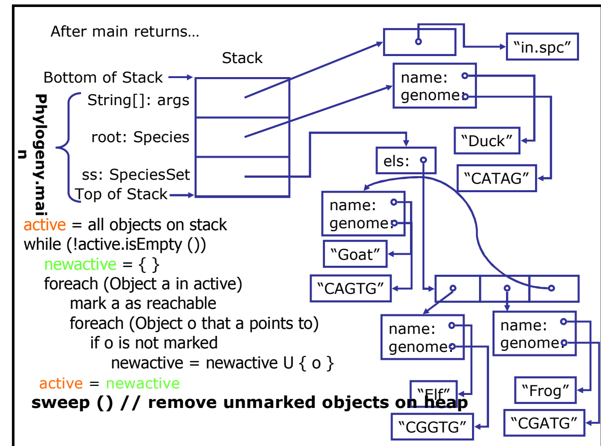


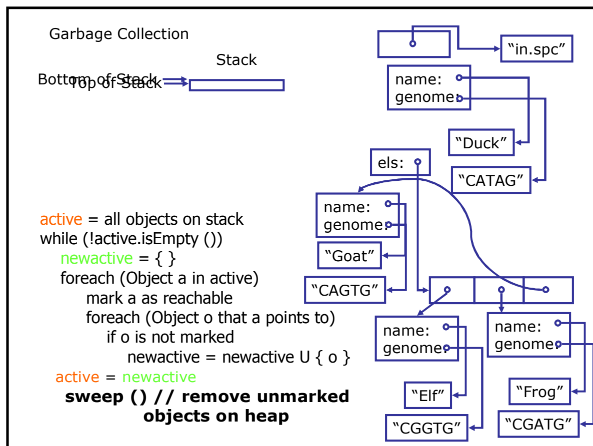
Garbage Collection
Stack
Bottom of Stack
String[]: args
root: Species
ss: SpeciesSet
Top of Stack
Phylogeny.main
"in.spc"
name:
genome
"Duck"
"CATAG"
els:
name:
genome
"Goat"
"CAGTG"
active = all objects on stack
while (!active.isEmpty ())
newactive = { }
foreach (Object a in active)
mark a as reachable (non-garbage)
foreach (Object o that a points to)
if o is not marked
newactive = newactive U { o }
active = newactive
name:
genome
name:
genome
"Elf"
"Frog"
"CGGTG"
"CGATG"











Problems with Mark and Sweep

- Fragmentation: free space and alive objects will be mixed
  - Harder to allocate space for new objects
  - Poor locality means bad memory performance
    - Caches make it quick to load nearby memory
- Multiple Threads
  - One stack per thread, one heap shared by all threads
  - All threads must stop for garbage collection

Stop and Copy

- Solves fragmentation problem
- Copy all reachable objects to a new memory area
- After copying, reclaim the whole old heap
- Disadvantages:
  - More complicated: need to change stack and internal object pointers to new heap
  - Need to save enough memory to copy
  - Expensive if most objects are not garbage

Generational Collectors

- Observation:
  - Many objects are short-lived
    - Temporary objects that get garbage collected right away
  - Other objects are long-lived
    - Data that lives for the duration of execution
- Separate storage into regions
  - Short term: collect frequently
  - Long term: collect infrequently
- Stop and copy, but move copies into longer-lived areas

Java's Garbage Collector (Sun implementation)

- Mark and Sweep collector
- Before collecting an object, it will call finalize

```
protected void finalize() throws Throwable
```

- Can call garbage collector directly: System.gc ()

Reference Counting

What if each object kept track of the number of references to it?

If an object has zero references, it is garbage!

Referencing Counting

T x = new T ();

The x object has one reference

y = x;

The object x references has 1 more ref

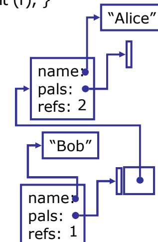
The object y_{pre} references has 1 less ref

} Leave scope where x is declared

The object x references has 1 less ref

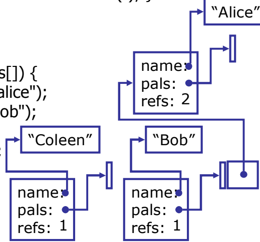
Reference Counting

```
class Recycle {
   private String name; private Vector pals;
   public Recycle (String name) { this.name = name; pals = new Vector (); }
   public void addPal (Recycle r) {  pals.addElement (r); }
}

public class Garbage {
   static public void main (String args[]) {
      Recycle alice = new Recycle ("alice");
      Recycle bob = new Recycle ("bob");
      bob.addPal (alice);
      alice = new Recycle ("coleen");
      bob = new Recycle ("dave");
   }
}
```



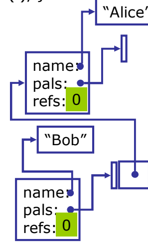
Reference Counting

```
class Recycle {
   private String name; private Vector pals;
   public Recycle (String name) { this.name = name; pals = new Vector (); }
   public void addPal (Recycle r) {  pals.addElement (r); }
}

public class Garbage {
   static public void main (String args[]) {
      Recycle alice = new Recycle ("alice");
      Recycle bob = new Recycle ("bob");
      bob.addPal (alice);
      alice = new Recycle ("coleen");
      bob = new Recycle ("dave");
   }
}
```



Reference Counting

```
class Recycle {
   private String name; private Vector pals;
   public Recycle (String name) { this.name = name; pals = new Vector (); }
   public void addPal (Recycle r) {  pals.addElement (r); }
}

public class Garbage {
   static public void main (String args[]) {
      Recycle alice = new Recycle ("alice");
      Recycle bob = new Recycle ("bob");
      bob.addPal (alice);
      alice = new Recycle ("coleen");
      bob = new Recycle ("dave");
   }
}
```



Python's Implementation

- Automatic reference counting to manage objects
- ...but optional additional garbage collector (we'll see why soon)

app1

```
static int
app1(PyListObject *self, PyObject *v)
{
  Py_ssize_t n = PyList_GET_SIZE(self);

  assert (v != NULL);
  if (n == INT_MAX) {
    PyErr_SetString(PyExc_OverflowError,
                    "cannot add more objects to list");
    return -1;
  }

  if (list_resize(self, n+1) == -1)
    return -1;

  Py_INCREF(v);
  PyList_SET_ITEM(self, n, v);
  return 0;
}
```


Python Objects

```
#define _Py_NewReference(op) ( \
   (op)->ob_refcnt = 1)

#define Py_INCREF(op) ( \
   (op)->ob_refcnt++)

#define Py_DECREF(op) \
   if (--(op)->ob_refcnt != 0) \
      _Py_CHECK_REFCNT(op) \
   else \
      _Py_Dealloc((PyObject *)(op))

#define _Py_CHECK_REFCNT(OP) \
 { if ((OP)->ob_refcnt < 0) \
    _Py_NegativeRefcount(__FILE__, __LINE__, \
                         (PyObject *)(OP)); }
```

Note: I have removed some debugging macros from these.

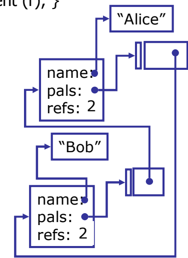
list_dealloc

```
static void list_dealloc(PyListObject *op) {
  Py_ssize_t i;
  PyObject_GC_UnTrack(op);
  Py_TRASHCAN_SAFE_BEGIN(op)
  if (op->ob_item != NULL) {
      /* Do it backwards, for Christian Tismer. There's a simple test case
         where somehow this reduces thrashing when a *very* large list
         is created and immediately deleted. */
      i = op->ob_size;
      while (--i >= 0) {
        Py_XDECREF(op->ob_item[i]);
      }
      PyMem_FREE(op->ob_item); }
  if (num_free_lists < MAXFREELISTS && PyList_CheckExact(op))
      free_lists[num_free_lists++] = op;
  else op->ob_type->tp_free((PyObject *)op);
  Py_TRASHCAN_SAFE_END(op)
}
```

Can reference counting ever fail to reclaim unreachable storage?

Circular References

```
class Recycle {
   private String name; private Vector pals;
   public Recycle (String name) { this.name = name; pals = new Vector (); }
   public void addPal (Recycle r) {  pals.addElement (r); }
}

public class Garbage {
   static public void main (String args[]) {
      Recycle alice = new Recycle ("alice");
      Recycle bob = new Recycle ("bob");
      bob.addPal (alice);
      alice.addPal (bob);
      alice = null;
      bob = null;
   }
}
```



Reference Counting Summary

- Advantages
  - Can clean up garbage right away when the last reference is lost
  - No need to stop other threads!
- Disadvantages
  - Need to store and maintain reference count
  - Some garbage is left to fester (circular references)
  - Memory fragmentation

Python has both!

- Reference counting:
  - To quickly reclaim most storage
- Garbage collector (optional, but on by default):
  - To collect circular references

Charge

- In Java: be happy you have a garbage collector to clean up for you
- In Python: by happy you have a reference counter managing your objects (but worry it might miss things!)
- In C:
  - Why is it hard to write a garbage collector for C?
- Return Exam 1