stuff: Storage management.

text to which it applies (is *visible*).

- Need not be contiguous.
- In Java, is static: independent of data.

- *Lifetime* or *extent* of storage is portion of program execution during which it exists.
  - Always contiguous
  - Generally dynamic: depends on data
- Classes of extent:
  - *Static:* entire duration of program
  - *Local* or *automatic:* duration of call or block execution (local variable)
  - *Dynamic:* From time of allocation statement (**new**) to deallocation, if any.

storage.

- However, when no expression in any thread can possibly be influenced by or change an object, it might as well not exist:

```
IntList wasteful()
{
  IntList c = new IntList(3, new IntList(4,
null));
  return c.tail;
  // variable c now deallocated, so no way
  // to get to first cell of list
}
```

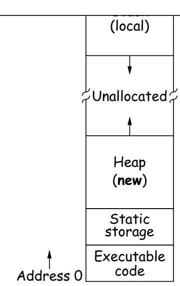
- At this point, Java runtime, like Scheme's, recycles the object c pointed to: *garbage collection*.

as integer addresses.

- Corresponds to machine's own practice.
- In Java, cannot convert integers $\leftrightarrow$ pointers,
- But crucial parts of Java runtime implemented in C, or sometimes machine code, where you can.
- Crude allocator in C:

```
char store[STORAGE_SIZE];  // Allocated array
size_t remainder = STORAGE_SIZE;

/** A pointer to a block of at least N bytes
of storage */
void* simpleAlloc(size_t n) { // void*: pointer
to anything
   if (n > remainder) ERROR();
   remainder = (remainder - n) & ~0x7;  // Make
multiple of 8
   return (void*) (store + remainder);
```

- OS gives way to turn chunks of unallocated region into heap.
- Happens automatically for stack.

because of

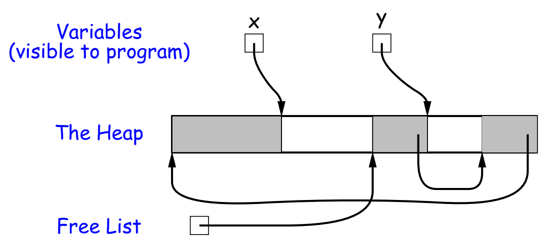
- Lack of run-time information about what is array
- Possibility of converting pointers to integers.
- Lack of run-time information about *unions:*

```
union Various {
  int Int;
  char* Pntr;
  double Double;
} X;  // X is either an int, char*, or double
```

- Java avoids all three problems; automatic collection possible.
- Explicit freeing can be somewhat faster, but rather error-prone:
  - Memory corruption

from OS and gives to applications.

- Or gives recycled storage, when available.
- When storage is freed, added to a *free list* data structure to be recycled.
- Used both for explicit freeing and some kinds of automatic garbage collection.

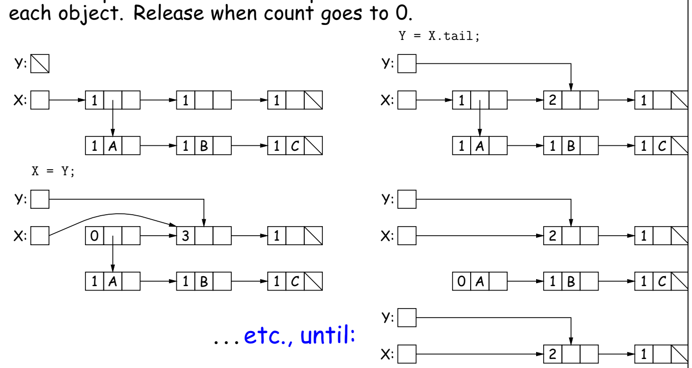


sizes.

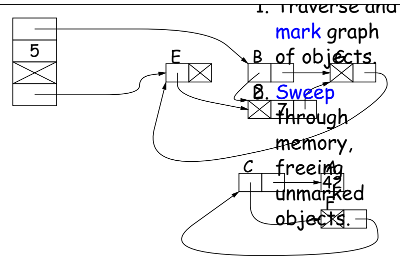
- Not all chunks on the free list are big enough, and one may have to search for a chunk and break it up if too big.
- Various strategies to find a chunk that fits have been used:
  - *Sequential fits:*
    - ∗ Link blocks in LIFO or FIFO order, or sorted by address.
    - ∗ Coalesce adjacent blocks.
    - ∗ Search for *first fit* on list, *best fit* on list, or *next fit* on list after last-chosen chunk.
  - *Segregated fits:* separate free lists for different chunk sizes.
  - *Buddy systems:* A kind of segregated fit where some newly adjacent free blocks

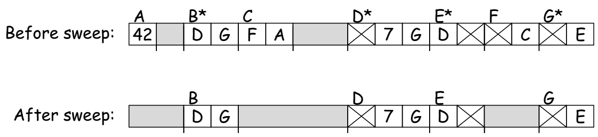
memory into lots of little scattered chunks.

- Idea: Keep count of number of pointers to each object. Release when count goes to 0.



1. Traverse and mark graph of objects.
2. Sweep through memory, freeing unmarked objects.



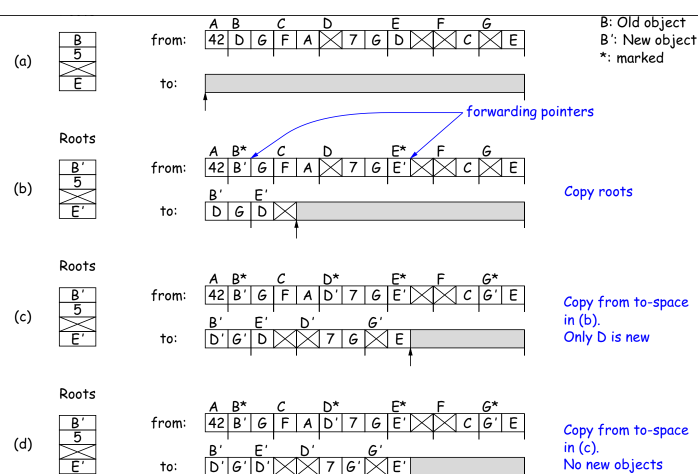


exisiting objects—pointers stay the same.

- The total amount of work depends on the amount of memory swept—i.e., the total amount of active (non-garbage) storage + amount of garbage. Not necessarily a big hit: the garbage had to be active at one time, and hence there was always some "good" processing in the past for each byte of garbage scanned.

tion takes time proportional to amount of active storage:

- Traverse the graph of active objects breadth first, *copying* them into a large contiguous area (called "to-space").
- As you copy each object, mark it and put a *forwarding pointer* into it that points to where you copied it.
- The next time you have to copy an already marked object, just use its forwarding pointer instead.
- When done, the space you copied from ("from-space") becomes the next to-space; in effect, all its objects are freed in constant time.



- Most older objects stay active, and need not be collected.
- Would be nice to avoid copying them over and over.
- *Generational garbage collection* schemes have two (or more) from spaces: one for newly created objects (*new space*) and one for "tenured" objects that have survived garbage collection (*old space*).
- A typical garbage collection collects only in new space, ignores pointers from new to old space, and moves objects to old space.
- As roots, uses usual roots plus pointers in old space that have changed (so that they might be pointing to new space).
- When old space full, collect all spaces.

- Lots of work on how to implement these ideas efficiently.
- *Distributed garbage collection:* What if objects scattered over many machines?
- *Real-time collection:* where predictable pause times are important, leads to *incremental* collection, doing a little at a time.