Memory Management

Advanced Compiler Techniques
2005
Erik Stenman
EPFL

Memory Management

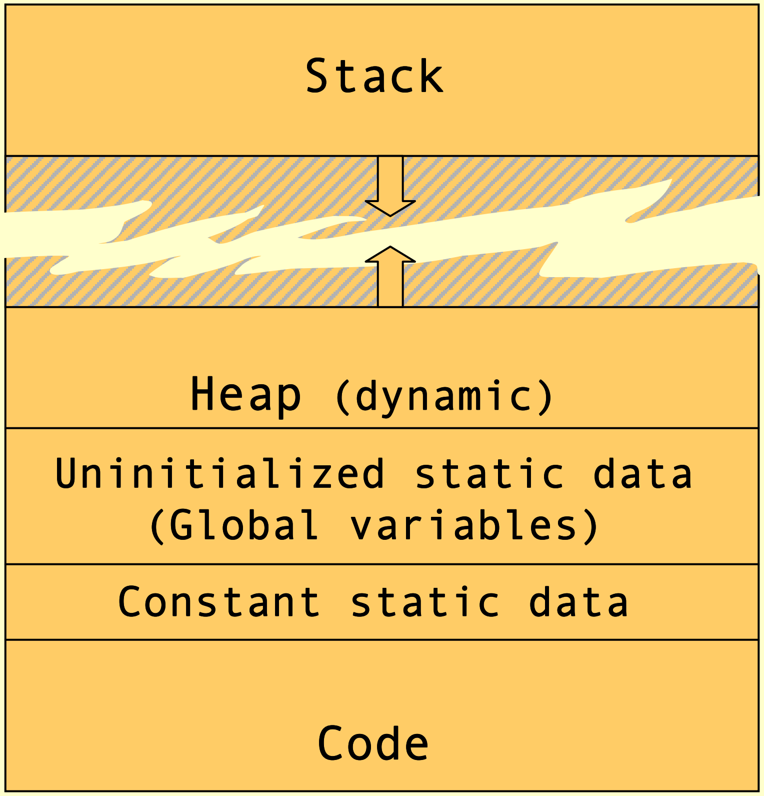
- The computer memory is a limited resource so the memory use of programs has to be *managed* in some way.
- The *memory management* is usually performed by a *runtime system* with help from the compiler.
  - The runtime system is a set of system procedures linked to the program.
  - For C programs it can be as simple as a small library for interacting with the operating system.
  - For Erlang programs the runtime system implements almost all the functionality normally provided by the OS.

Memory Management

- In a language such as C there are three ways to allocate memory:
  1. *Static allocation*. The size of memory needed by global variables (and code) is decided at compile time.
  2. *Stack allocation*. Activation records are allocated on the stack at function calls.
  3. *Heap allocation*. **Dynamically** allocated by the programmer e.g. by the use of `malloc`.

Memory Organization

- A typical layout of the memory of a C program looks like:



Dynamic Memory Management

- Heap allocation is necessary for data that lives longer than the function which created it, and which is passed by reference, e.g., lists in eins.
- Two design questions for the heap:
  - How is space for data allocated on the heap?
  - How and when is the space deallocated?
- Considerations in memory management design:
  - *Space leaks* & *dangling pointers*.
  - The runtime cost for *allocation* and *deallocation*.
  - *Space overhead* of the memory manager.
  - *Fragmentation*.

Fragmentation

- The memory management system should try to avoid *fragmentation*, i.e. when the free memory is broken up into several small blocks instead of few large blocks.
- In a fragmented system memory allocation may fail because there is no free block that is large enough even though the total free memory would be large enough.
- We distinguish between:
  - *Internal fragmentation* – the allocated block is larger than the requested size (the waste is in the allocated data).
  - *External fragmentation* – all free blocks are too small (the waste is in the layout of the free data).

Memory Allocation

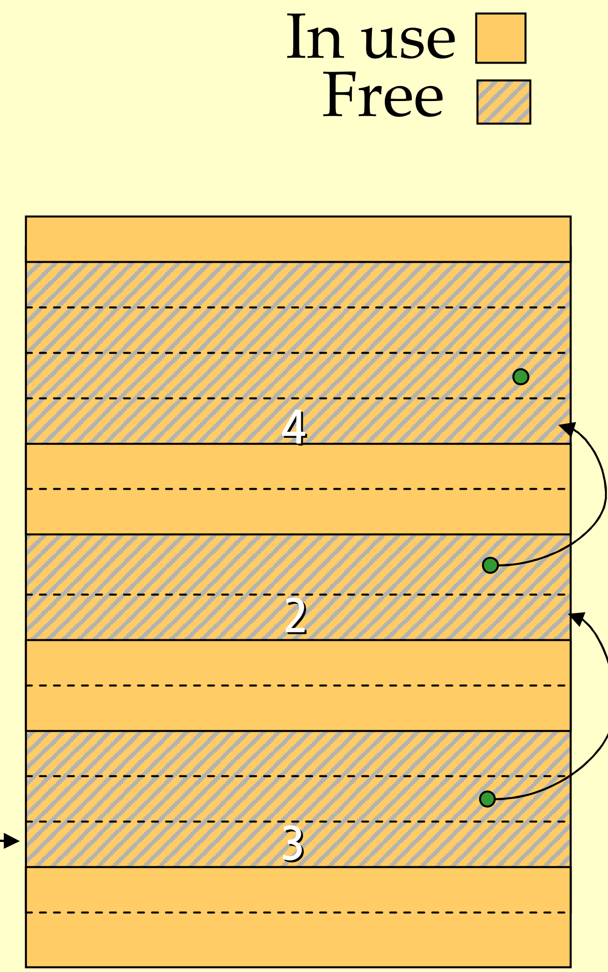
- The use of a *free-list* is a common scheme.
- The system keeps a list of unused memory blocks.
- To allocate memory the free-list is searched to find a block which is large enough.
- The block is removed from the free-list and used to store the data. If the block is larger than the need, it is split and the unused part is returned to the free-list (to avoid internal fragmentation).
- When the memory is freed it is returned to the free-list. Adjacent memory blocks can be merged (or coalesced) into larger blocks (to avoid external fragmentation).

Free-list

- The free-list can be stored in the free memory since it is not used for anything else. (We assume, or ensure, that each memory block is at least two words).

Free list:

This can be stored as a static global variable.



Free-list

- Note that we need to know the size of a block when it is deallocated. This means that even allocated blocks need to have a size field in them.
- Thus the space overhead will be at least one word per allocated data object. (It might also be advantageous to keep the link.)
- The cost (time) of allocation/deallocation is proportional to the search through the free-list.

Free-list

- There are many different ways to implement the details of the free-list algorithm:
  - Search method: first-fit, best-fit, next-fit.
  - Links: single, double.
  - Layout: one list, one list per block size, tree, buddy.

Deallocation

- Deallocation can either be *explicit* or *implicit*.
- Explicit deallocation is used in e.g., Pascal (`new/dispose`), C (`malloc/free`), and C++ (`new/delete`).
- Implicit deallocation is used in e.g., Lisp, Prolog, Erlang, ML, and Java.

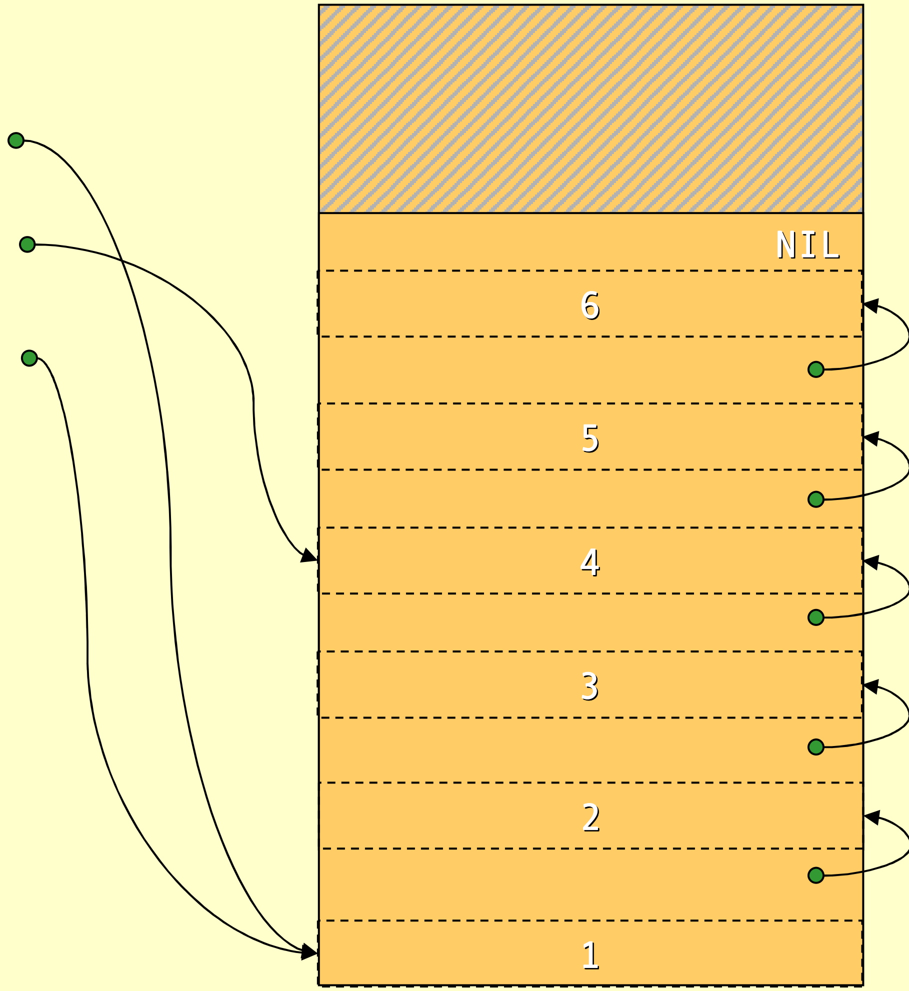
Explicit Deallocation

- Explicit deallocation has a number of problems:
  - If done too soon it leads to dangling pointers.
  - If done too late (or not at all) it leads to space leaks.
  - In some cases it is almost impossible to do it at the right time. Consider a library routine to append two mutable lists:

```
c = append(a,b);
```

Explicit Deallocation

```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
printList(b);
```



Explicit Deallocation

```
list a = new List(1,2,3);
list b = new List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
printList(b);
free(c);
```

- The programmer now has to ensure that a, b, and c are all deallocated at the same time. A mistake would lead to dangling pointers.
- If b is in use long after a, and c, then we will keep a live too long. A space leak.

Implicit Deallocation

- With *implicit deallocation* the programmer does not have to worry about when to deallocate memory.
- The runtime system will *dynamically* decide when it is **safe** to do this.
- In some cases, and systems, the compiler can also add static dealloctions to the program.
- The most commonly used automatic deallocation method is called *garbage collection* (GC).
- There are other methods such as *region based* allocation and deallocation.

Garbage Collection (GC)

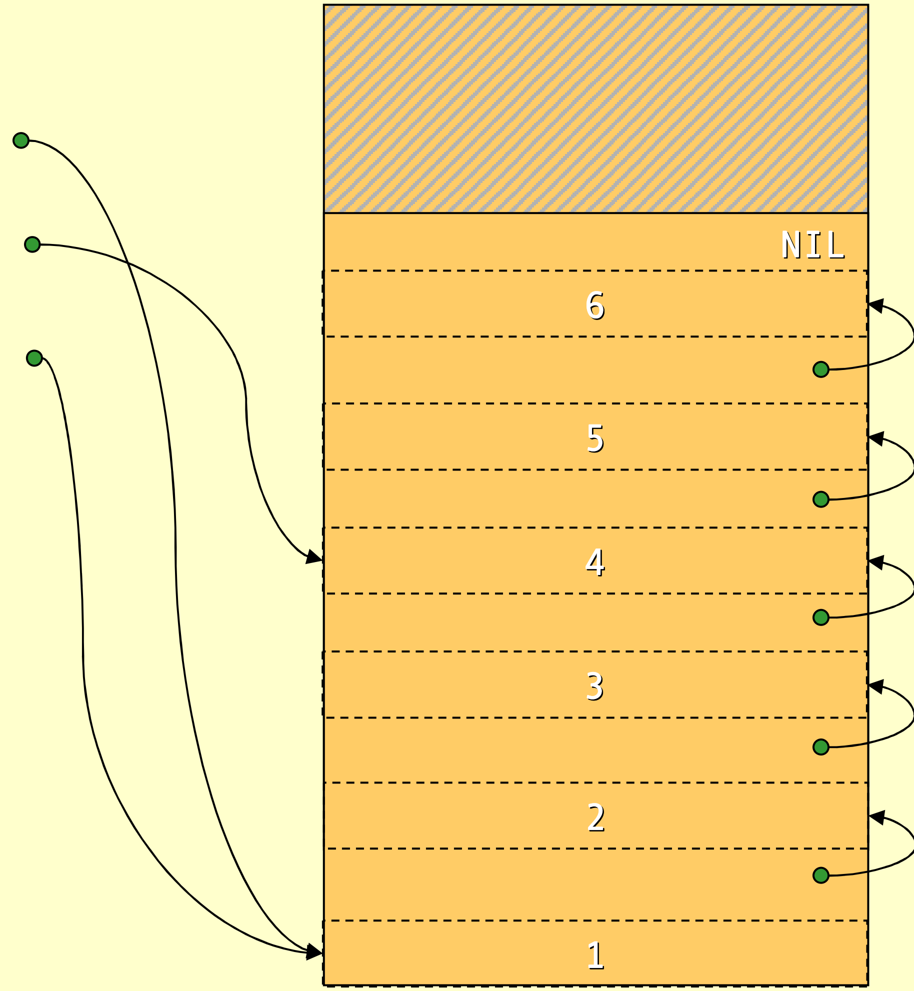
- *Garbage collection* is a common name for a set of techniques to deallocate heap memory that is unreachable by the program.
- There are several different base algorithms: *reference counting*, *mark & sweep*, *copying*.
- We can also distinguish between how the GC interferes or interacts with the program: *disruptive*, *incremental*, *real-time*, *concurrent*.

The Reachability Graph

- The data reachable by the program form a directed graph, where the edges are pointers.
- The *roots* of this graph can be in:
  1. global variables,
  2. registers,
  3. local variables & formal parameters on the stack.
- Objects are *reachable* iff there is a path of edges that leads to them from some root. Hence, the compiler must tell the GC where the roots are.

The Reachability Graph

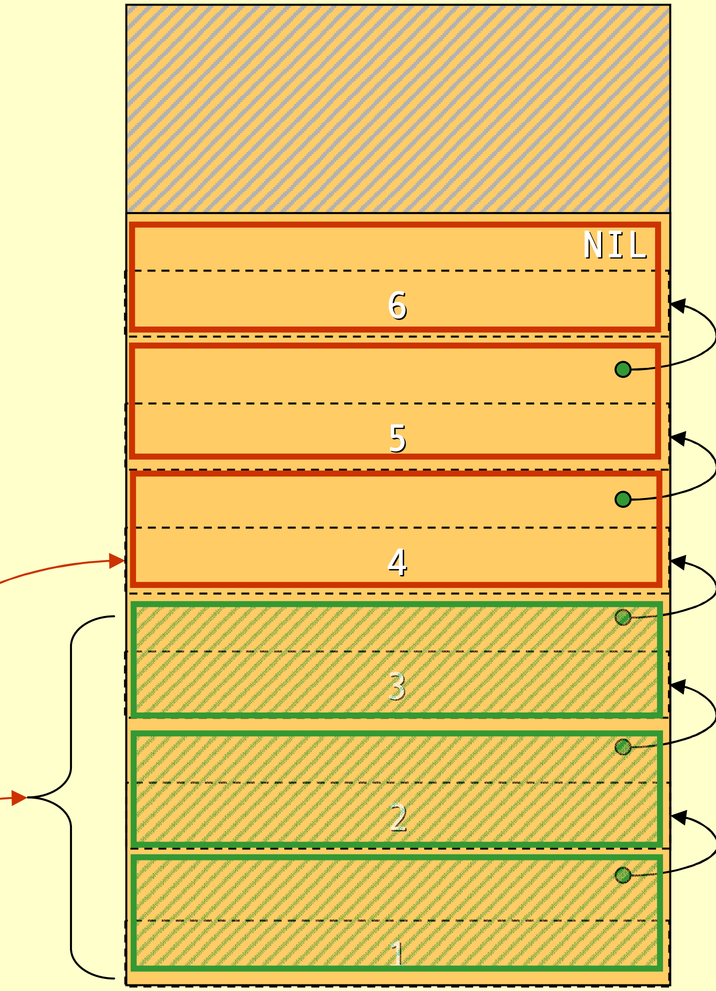
```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
return b;
```



The Reachability Graph

```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
return b;
```

roots: b

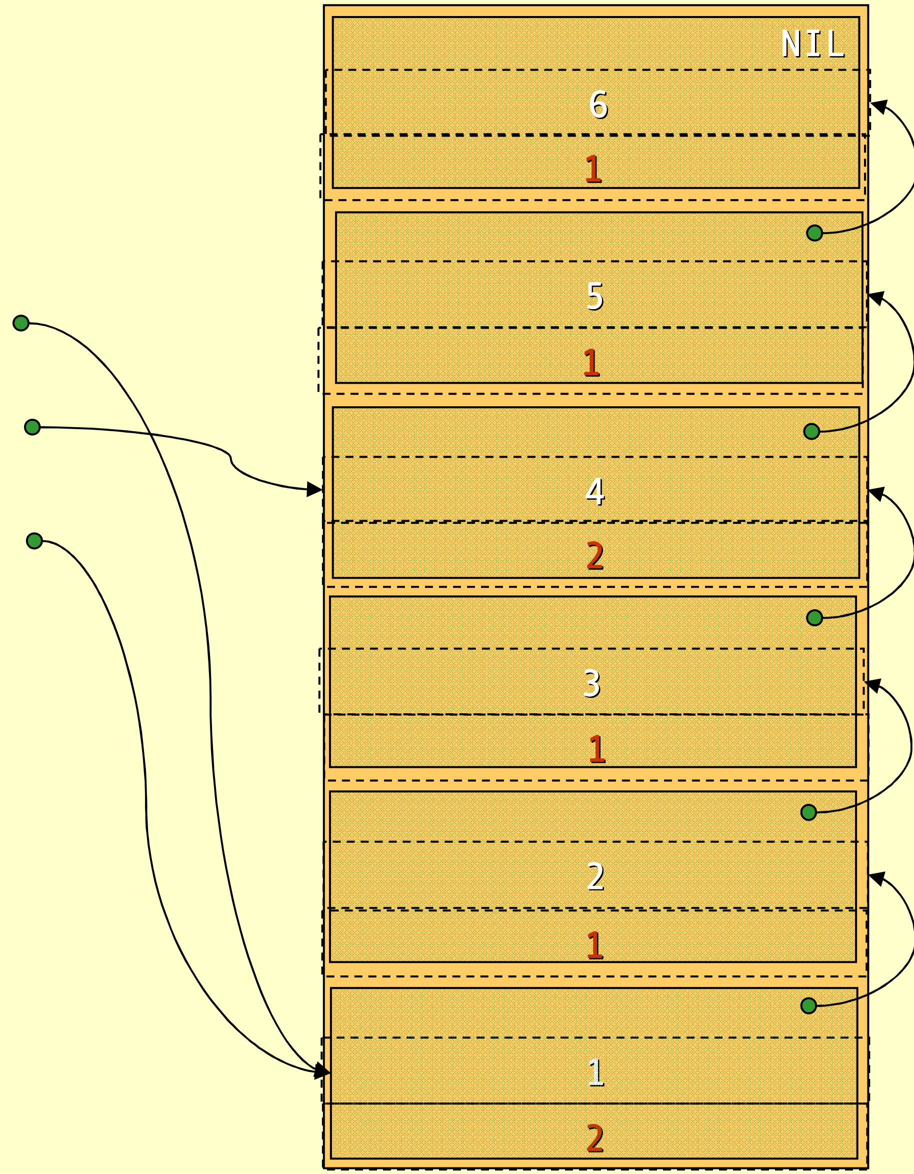


The goal with the GC is to deallocate these:

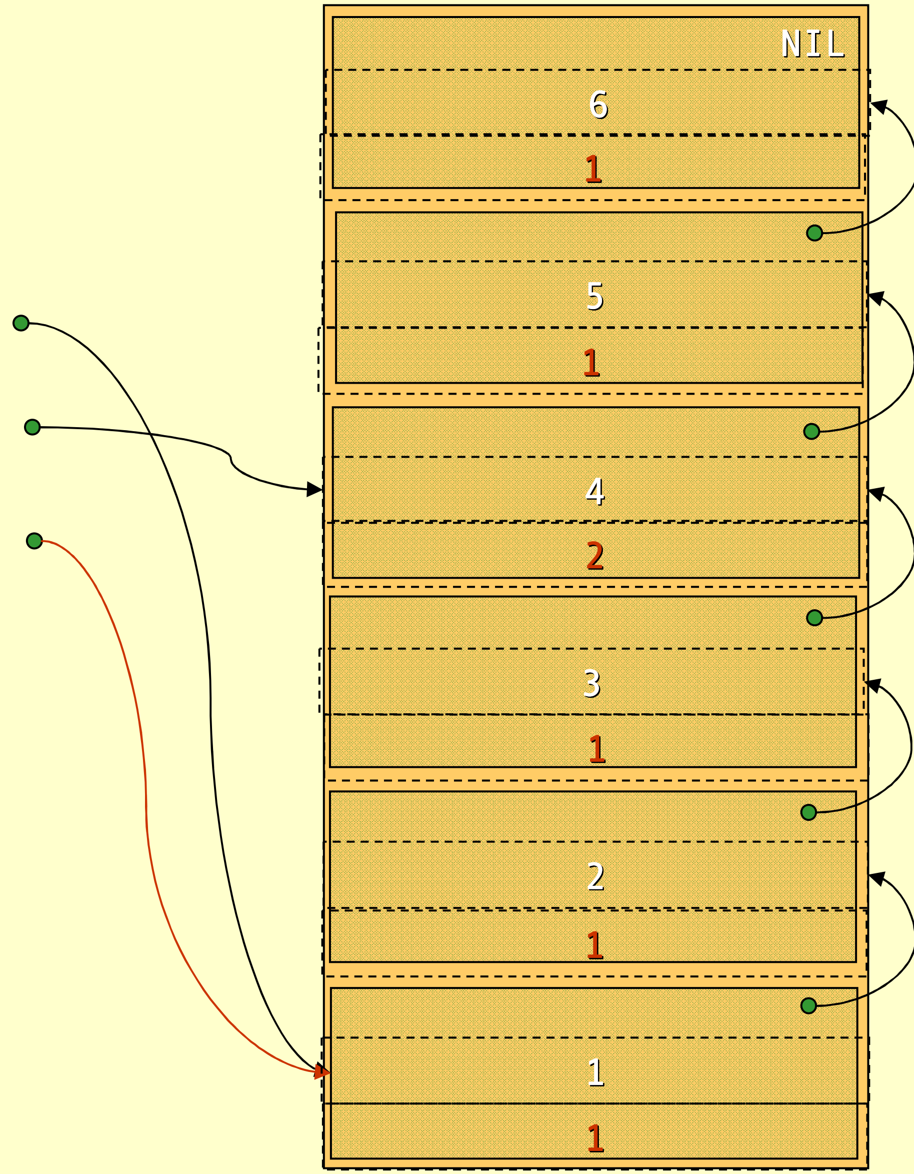
Reference Counting

- Idea: Keep track of how many references there are to each object.
- If there are 0 references deallocate the object.
- The compiler must add code to maintain the reference count (refcount).
  - Set the count to 1 when created.
  - For an assignment x = y:
    - if (x != null) x.refcount—;
    - if (y!=null) y.refcount++;
  - When a stack frame is deallocated decrease the refcount of each object pointed to from the frame.
  - When refcount reaches 0 deallocate the object and decrease refcount of each child.

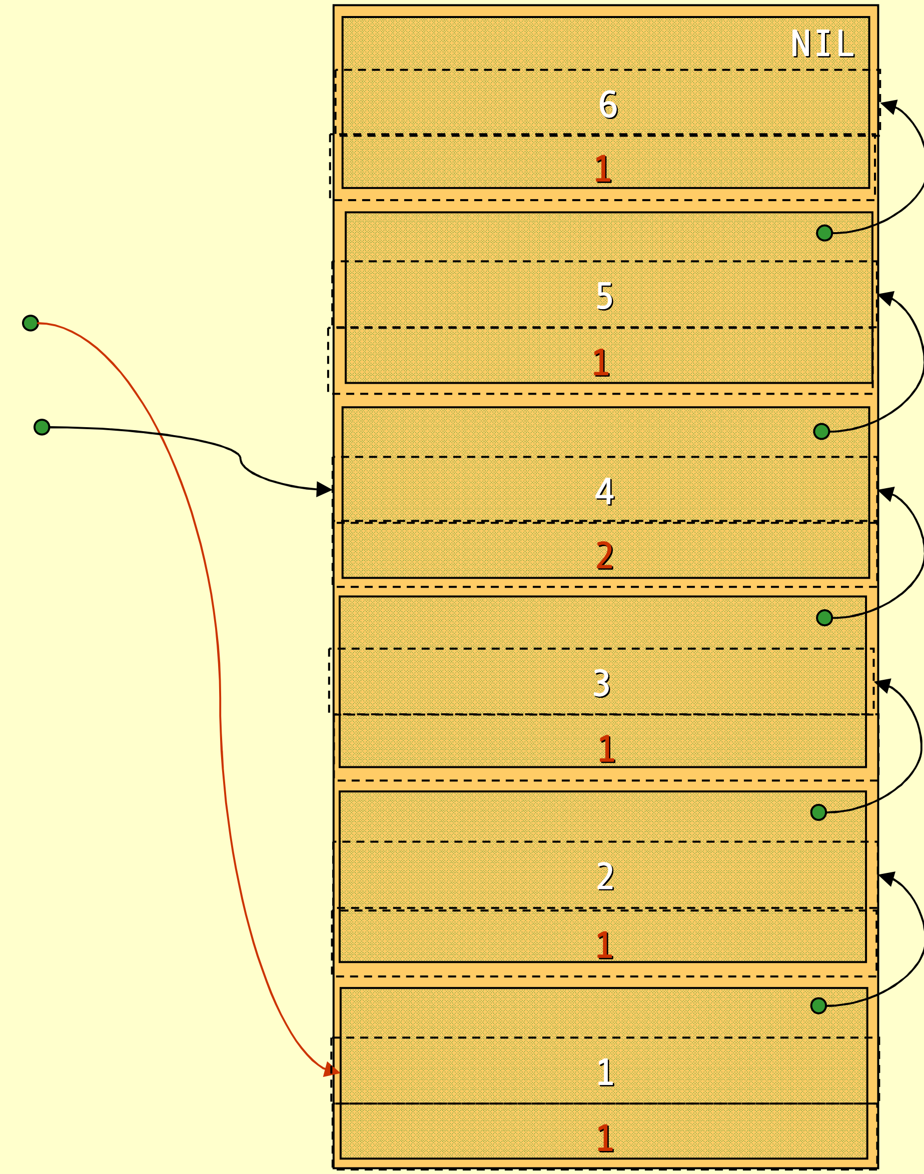
```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
decRefCount(c);
decRefCount(a);
doLotsOfStuff();
return b;
```



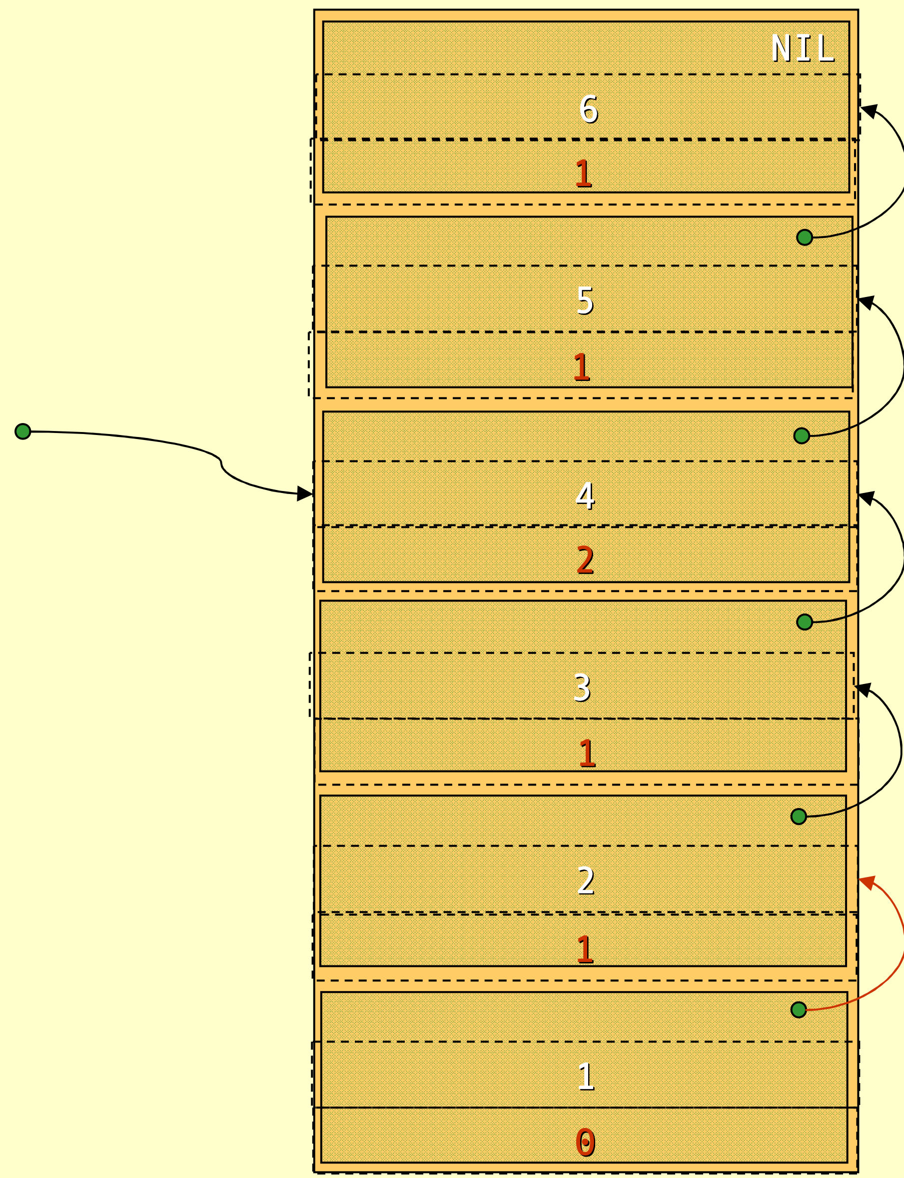
```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
decRefCount(c);
decRefCount(a);
doLotsOfStuff();
return b;
```



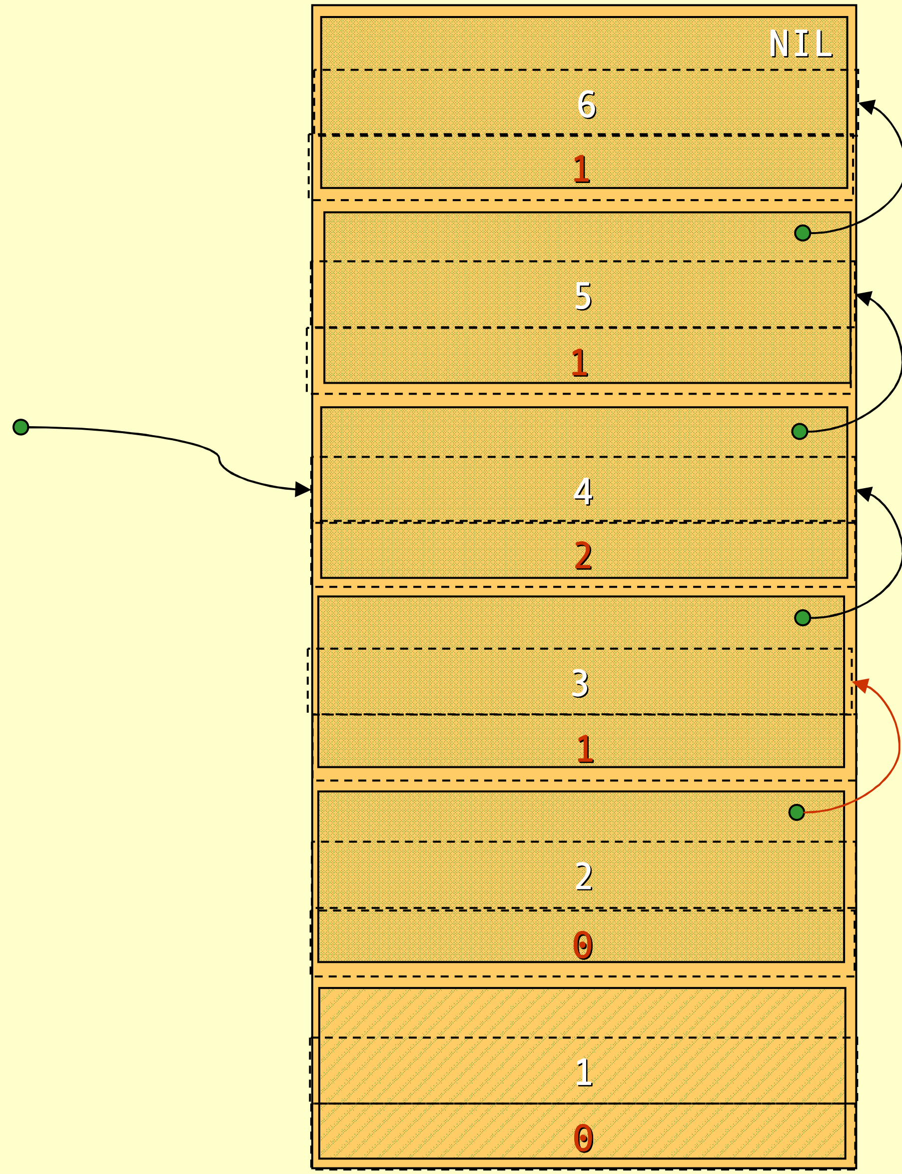
```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
decRefCount(c);
decRefCount(a);
doLotsOfStuff();
return b;
```



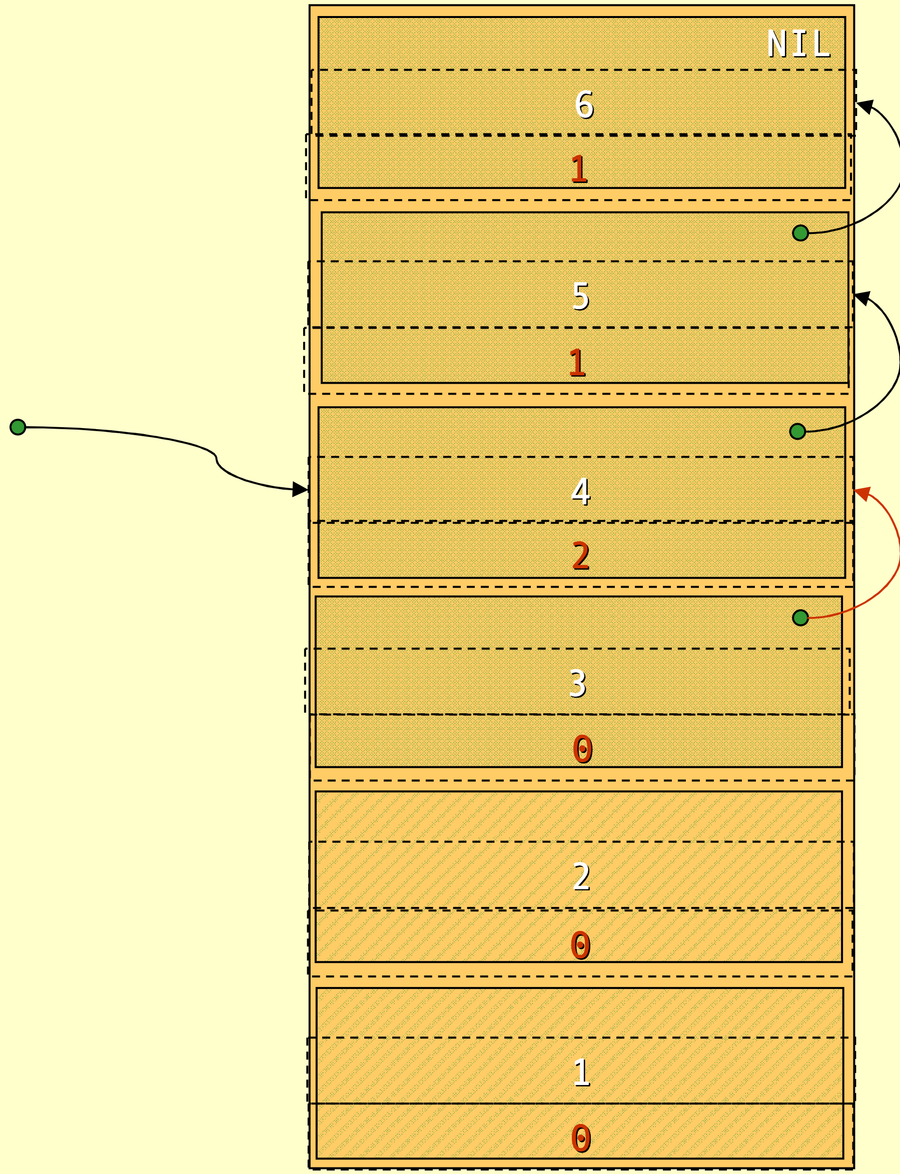
```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
decRefCount(c);
decRefCount(a);
doLotsOfStuff();
return b;
```



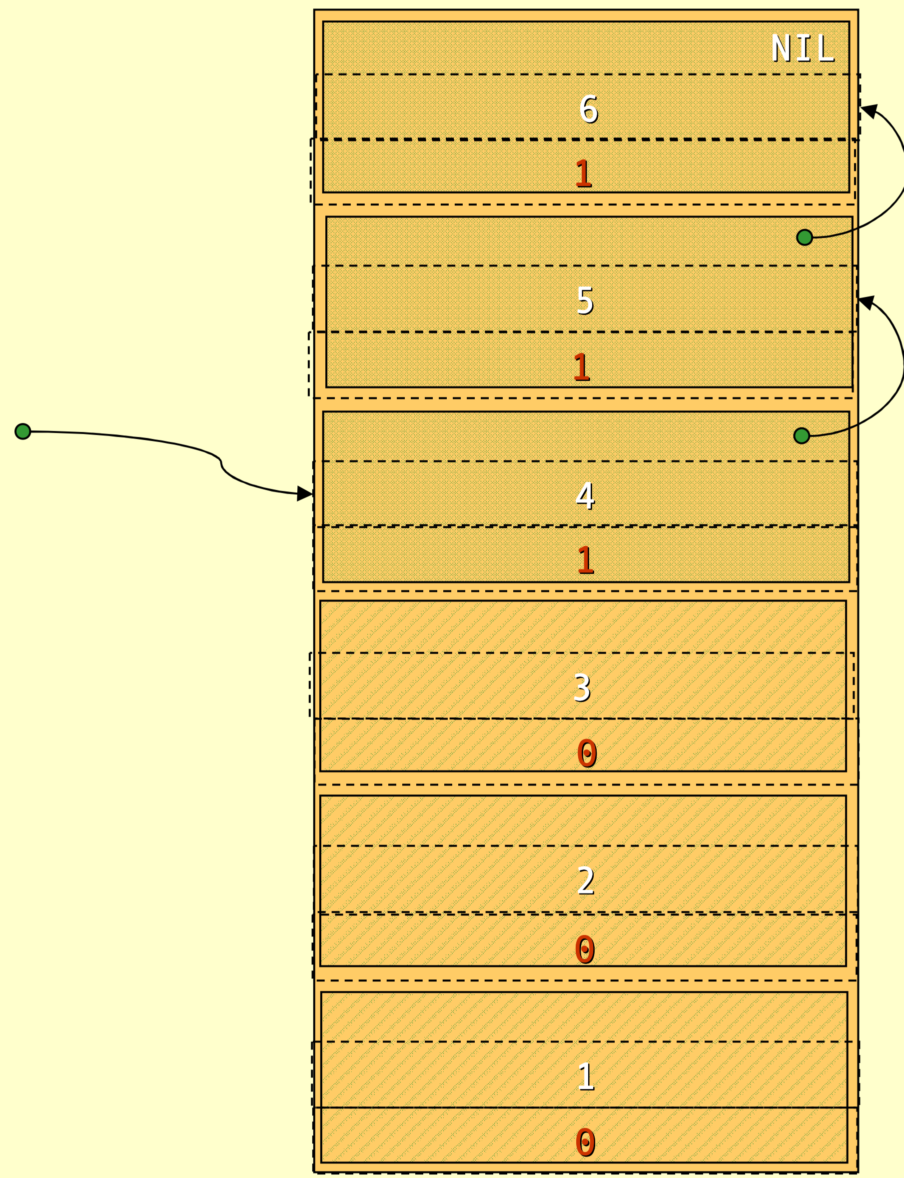
```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
decRefCount(c);
decRefCount(a);
doLotsOfStuff();
return b;
```



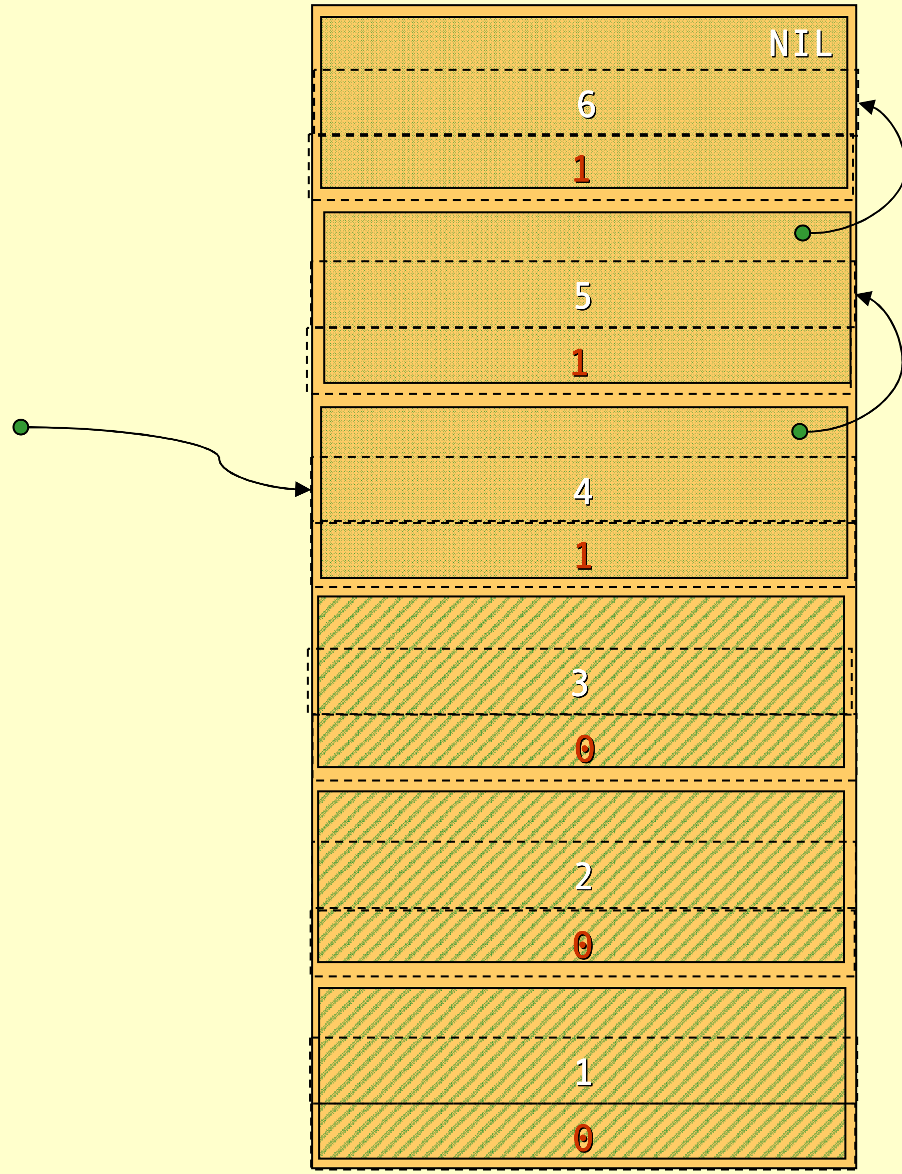
```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
decRefCount(c);
decRefCount(a);   ⇦
doLotsOfStuff();
return b;
```



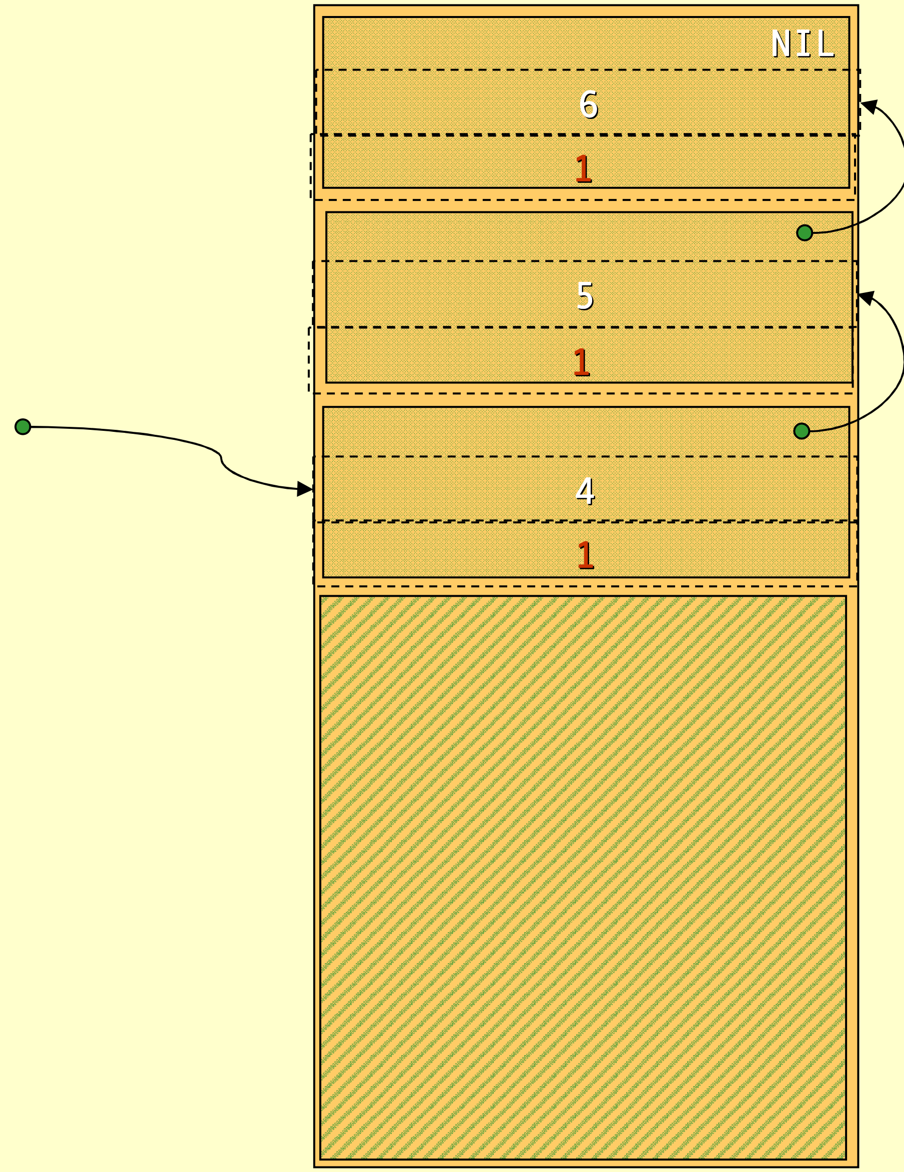
```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
decRefCount(c);
decRefCount(a);
doLotsOfStuff();
return b;
```



```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
decRefCount(c);
decRefCount(a);
doLotsOfStuff();
return b;
```



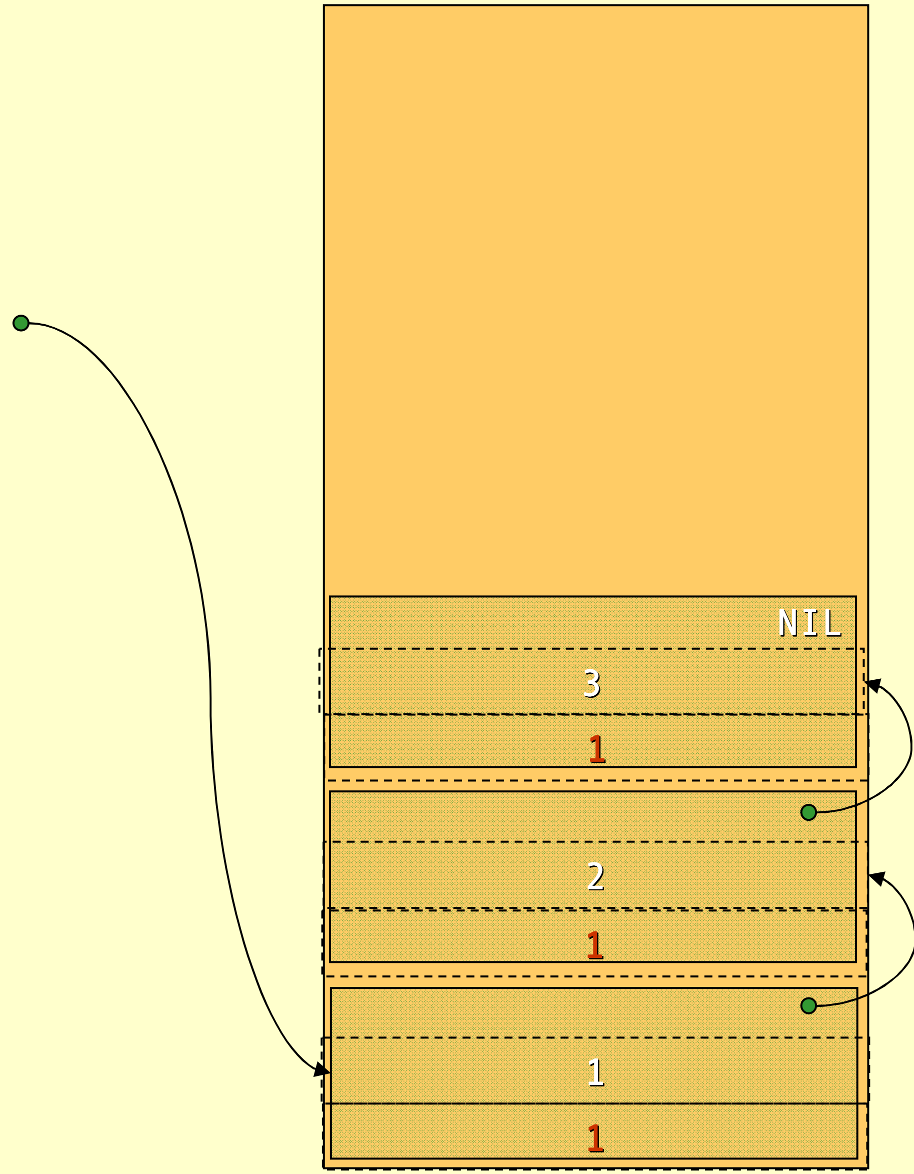
```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
decRefCount(c);
decRefCount(a);
doLotsOfStuff();
return b;
```



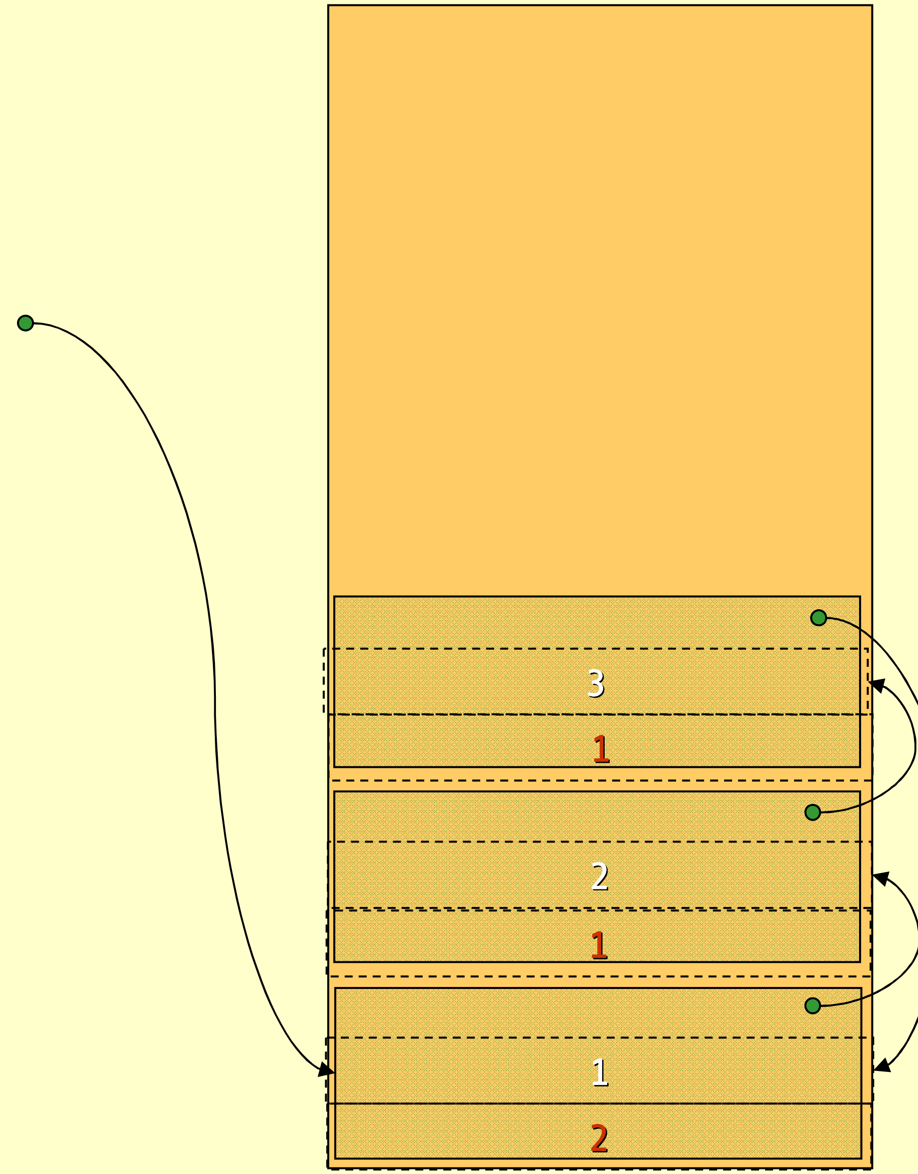
Reference Count

- Advantages of reference count:
  - Rather easy to implement.
  - Storage reclaimed **immediately**.
- Disadvantages of reference count:
  - Space overhead: 1 word per object.
  - Keeping track of the reference counts is **very expensive**. (Each simple pointer copy becomes several instructions.)
  - There is one more problem…

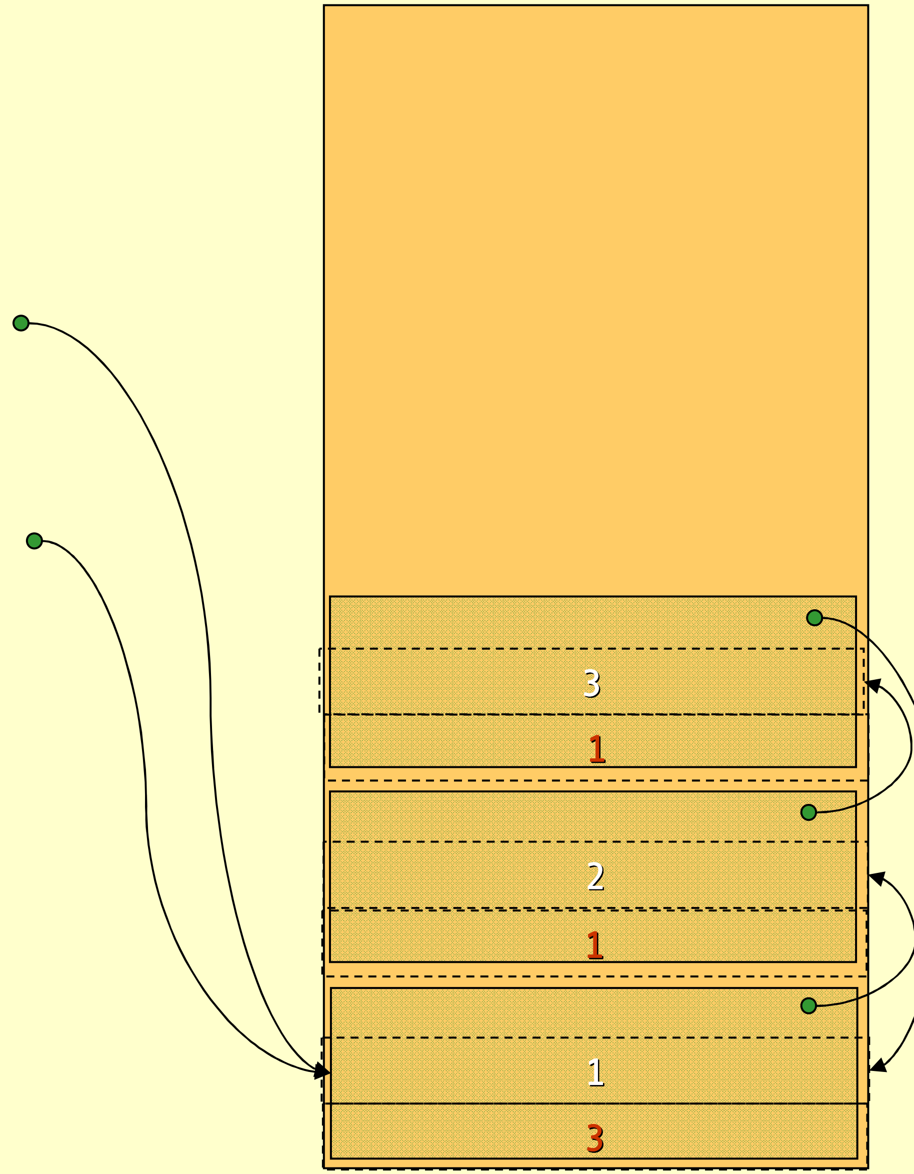
```
list a = List(1,2,3);
list b = NIL;
list c = append(a,a);
printList(c);
decRefCount(c);
decRefCount(a);
doLotsOfStuff();
return b;
```



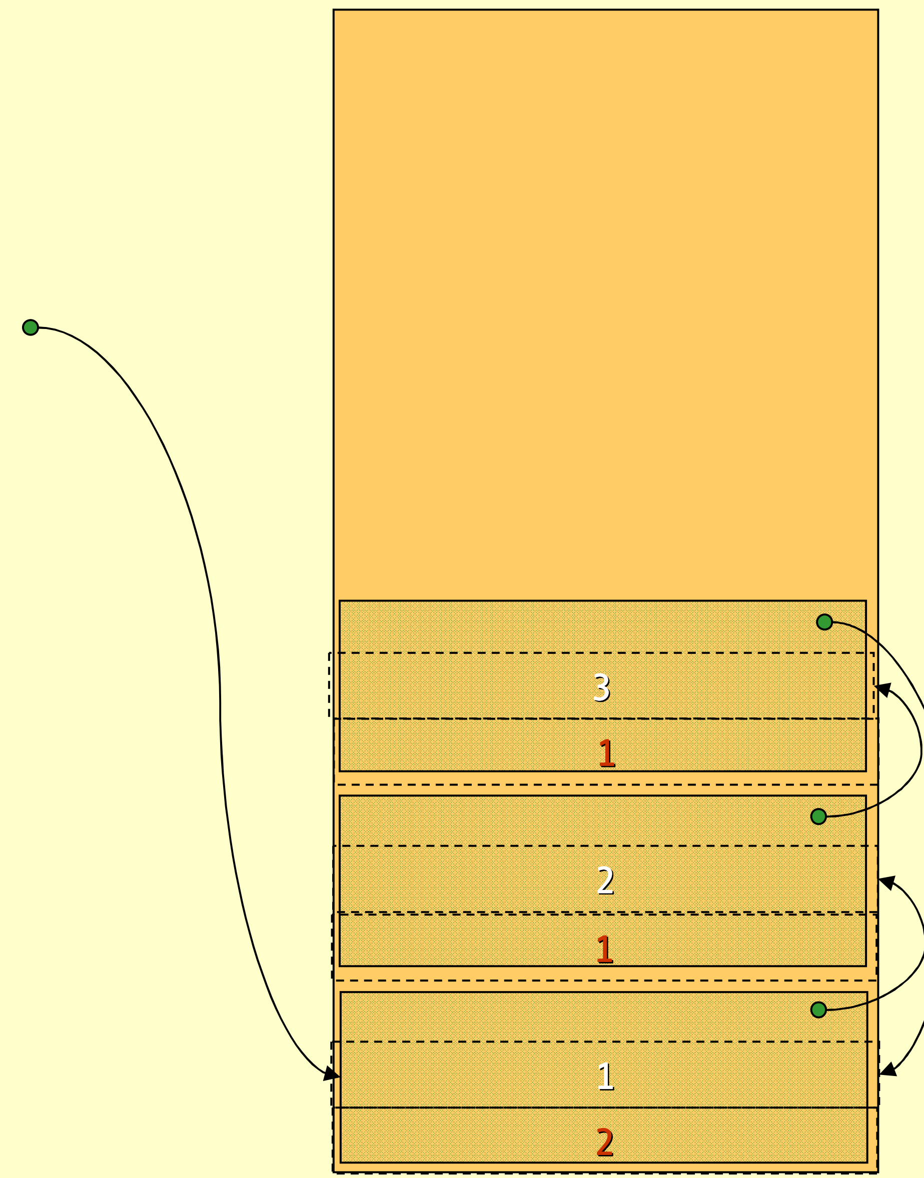
```
list a = List(1,2,3);
list b = NIL;
list c = append(a,a);
printList(c);
decRefCount(c);
decRefCount(a);
doLotsOfStuff();
return b;
```



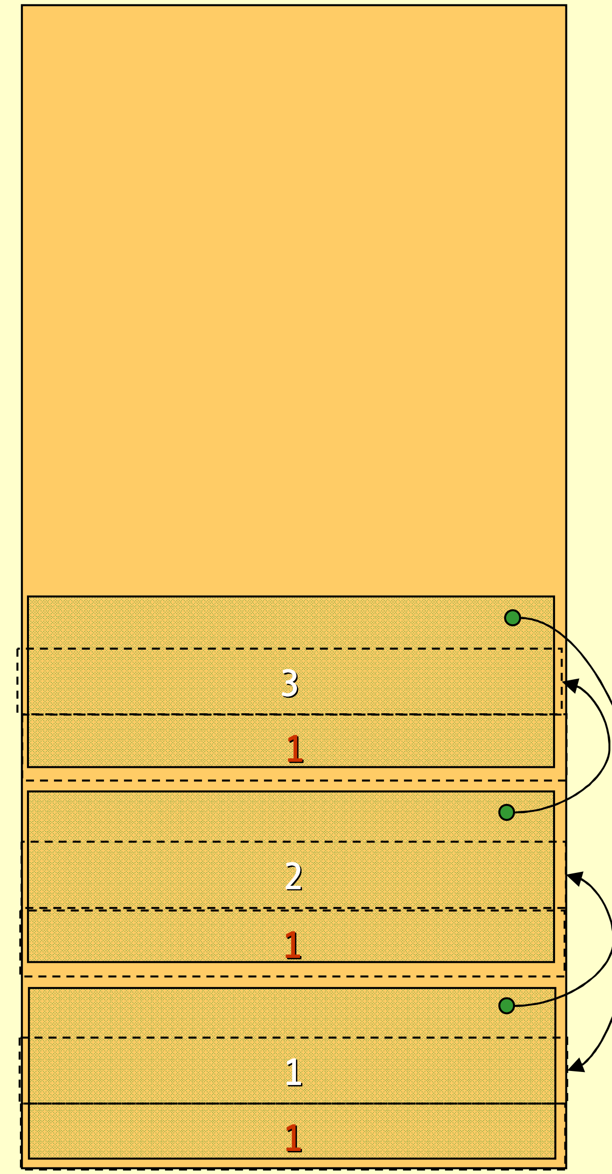
```
list a = List(1,2,3);
list b = NIL;
list c = append(a,a);
printList(c);
decRefCount(c);
decRefCount(a);
doLotsOfStuff();
return b;
```



```
list a = List(1,2,3);
list b = NIL;
list c = append(a,a);
printList(c);
decRefCount(c);
decRefCount(a);
doLotsOfStuff();
return b;
```



```
list a = List(1,2,3);
list b = NIL;
list c = append(a,a);
printList(c);
decRefCount(c);
decRefCount(a);
doLotsOfStuff();
return b;
```



Reference Count

- Big disadvantage with reference count:
  - The refcount of *cyclic structures* never reaches zero!
- There are ways to solve this, but they are very complicated.
- Due to this fact reference count is *very seldom* used in practice. There is one nice use, as we shall see later...
- In a pure language or a language without destructive updates there are no cyclic structures, making reference counting a viable option.

Mark & Sweep

- A *mark & sweep* GC is made up of two *phases*:
  1. First all reachable objects are *marked*.
  2. Then the heap is *swept* clean of dead objects.
- The mark phase is done by a *depth first search* through the reachability graph starting from the roots.

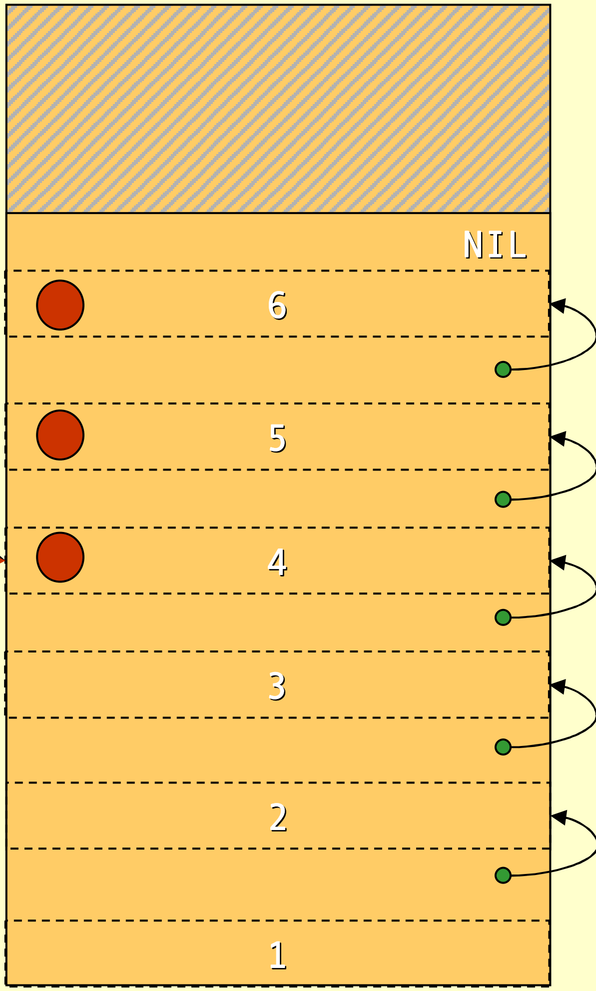
Depth First Mark Algorithm

```
mark(x) {
  if(! marked(x)) {
    setMark(x);
    for each field f of x
      mark(*f)
  }
}
```

Example: Mark

```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
return b;
```

mark(b)



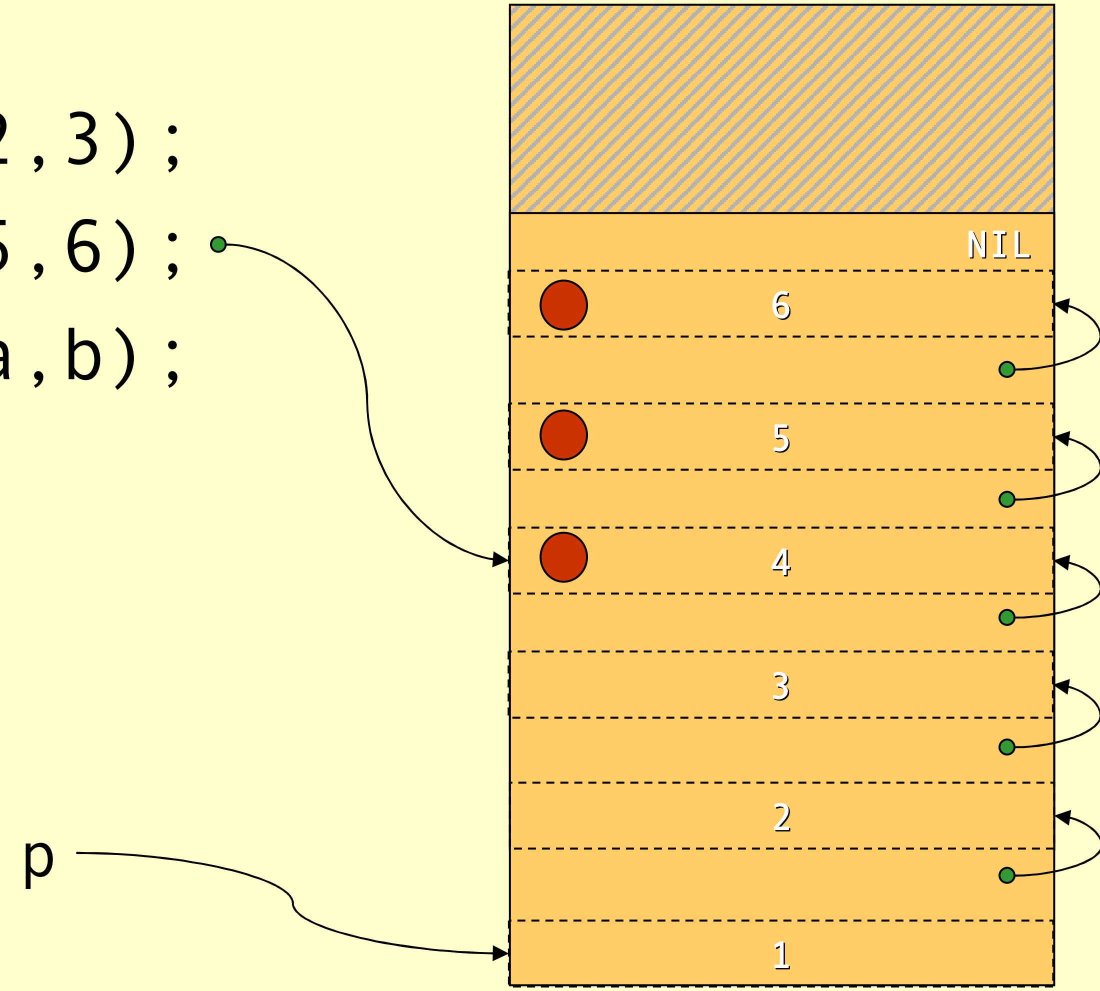
The Sweep

- The Sweep phase goes through the whole heap from start to finish and adds unmarked objects to the free-list.

```
p = heapStart;
while (p<heapEnd) {
  if(marked(*p)) clearMark(*p);
  else free(p);
  p += size(*p);
}
```

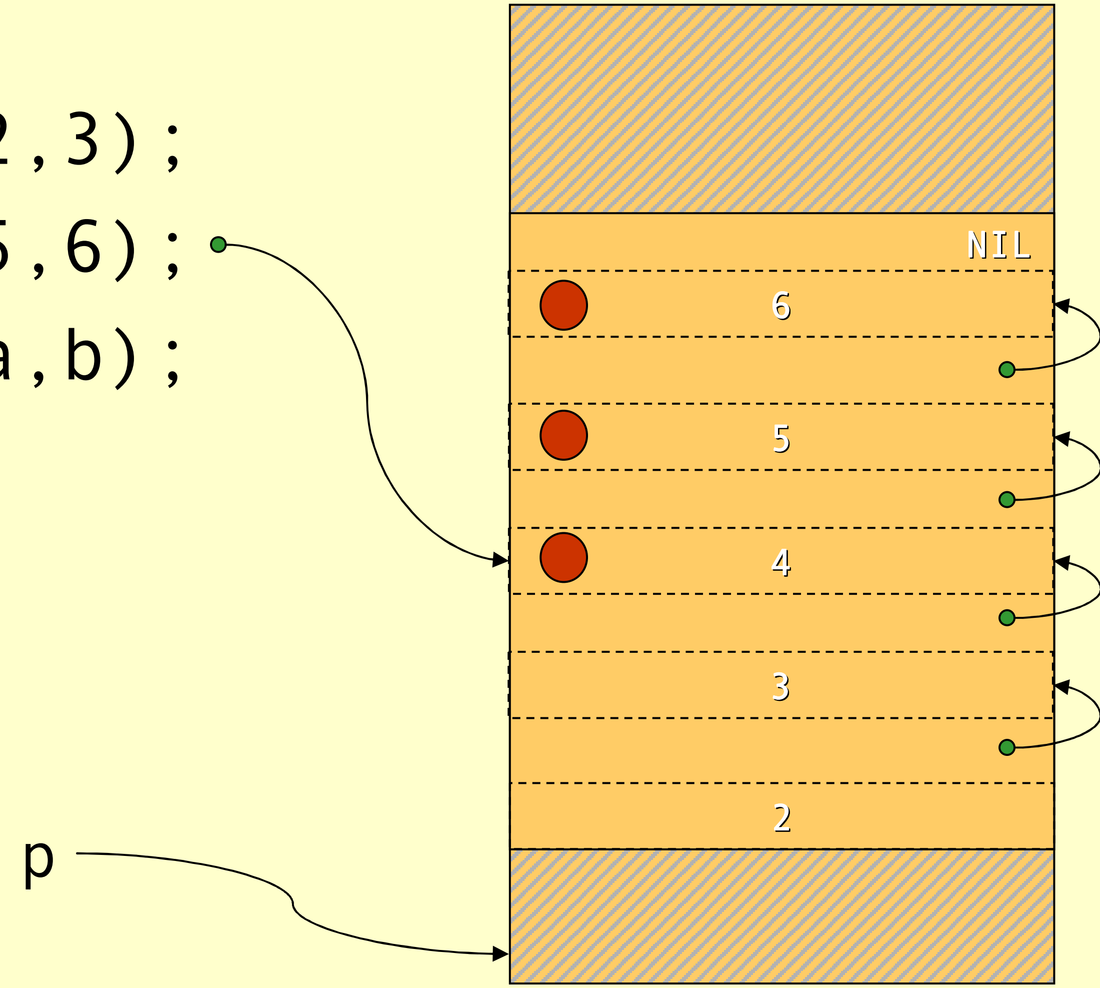
Example: Sweep

```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
return b;
```



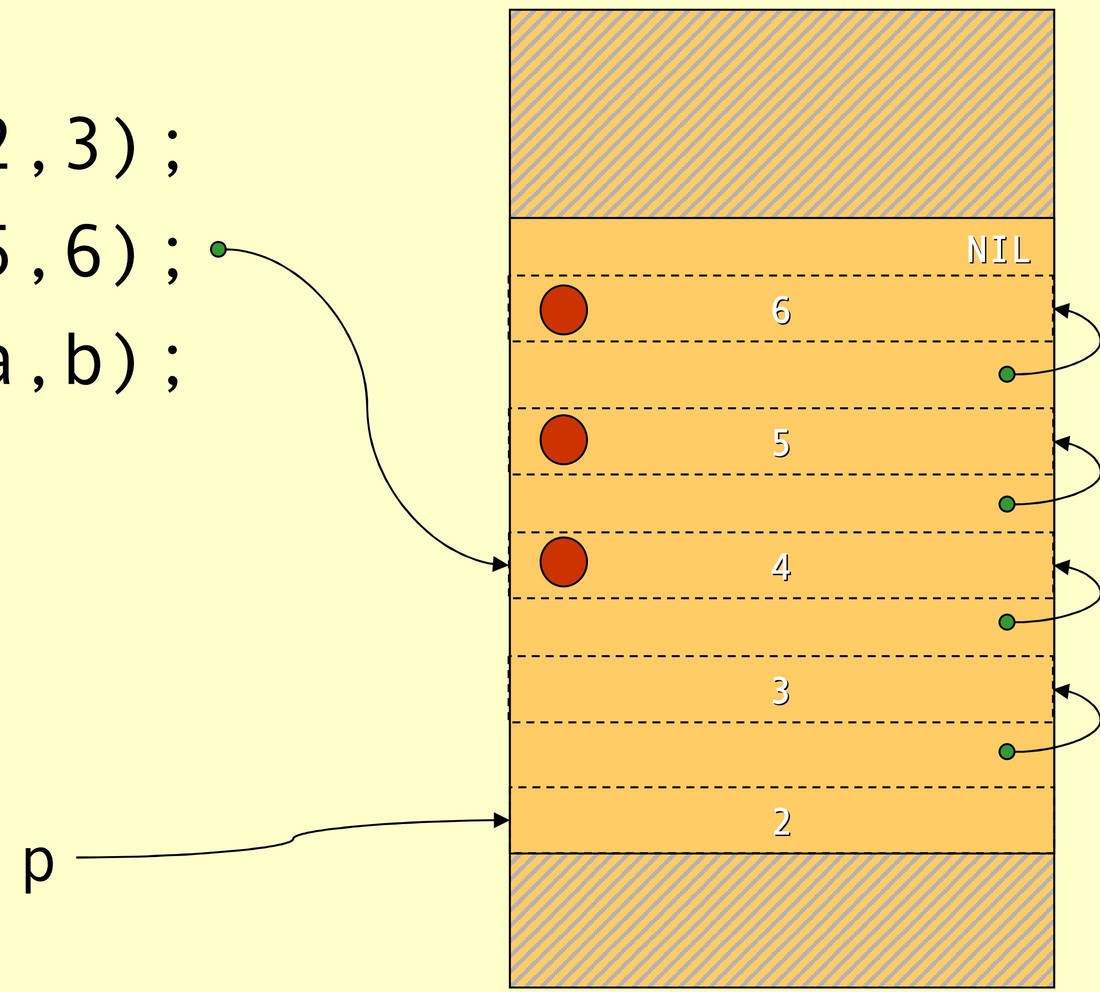
Example: Sweep

```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
return b;
```



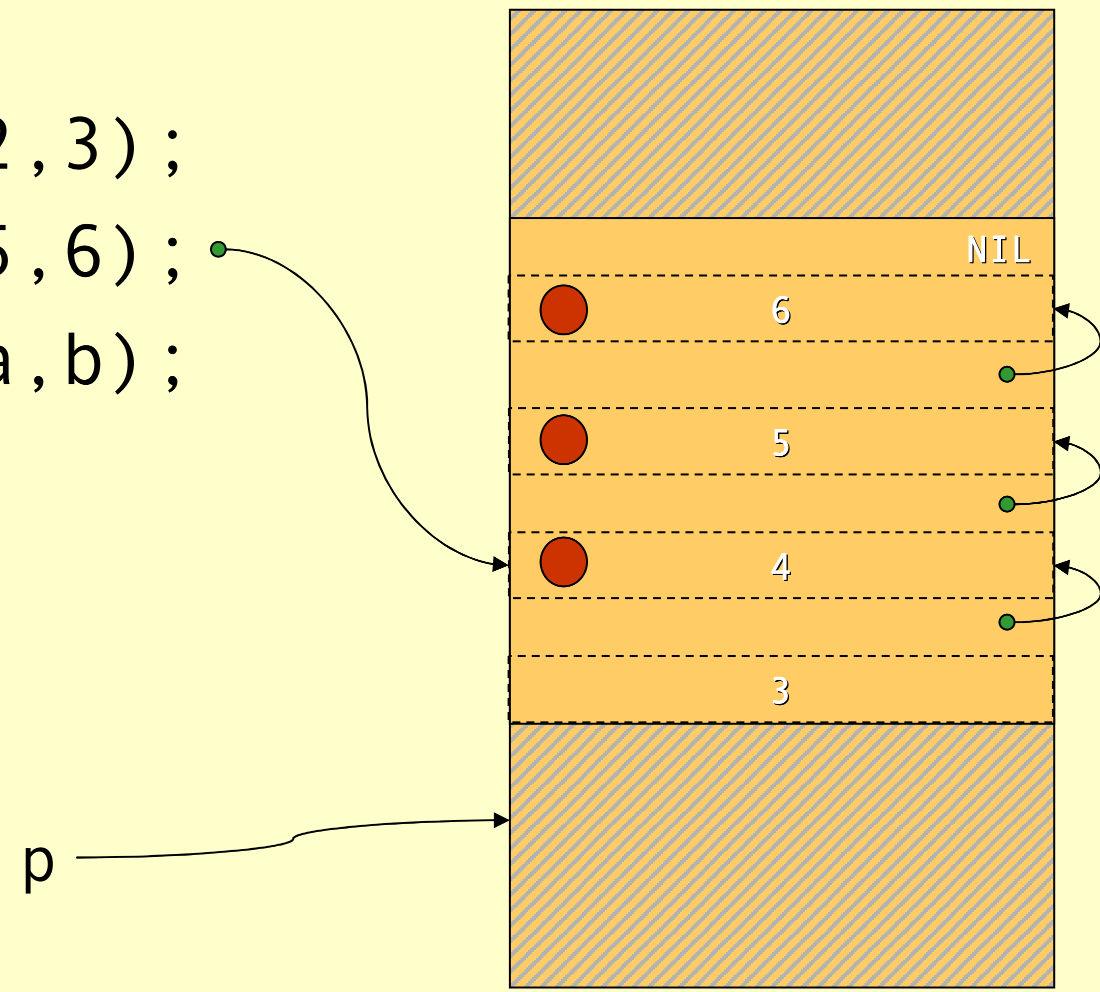
Example: Sweep

```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
return b;
```



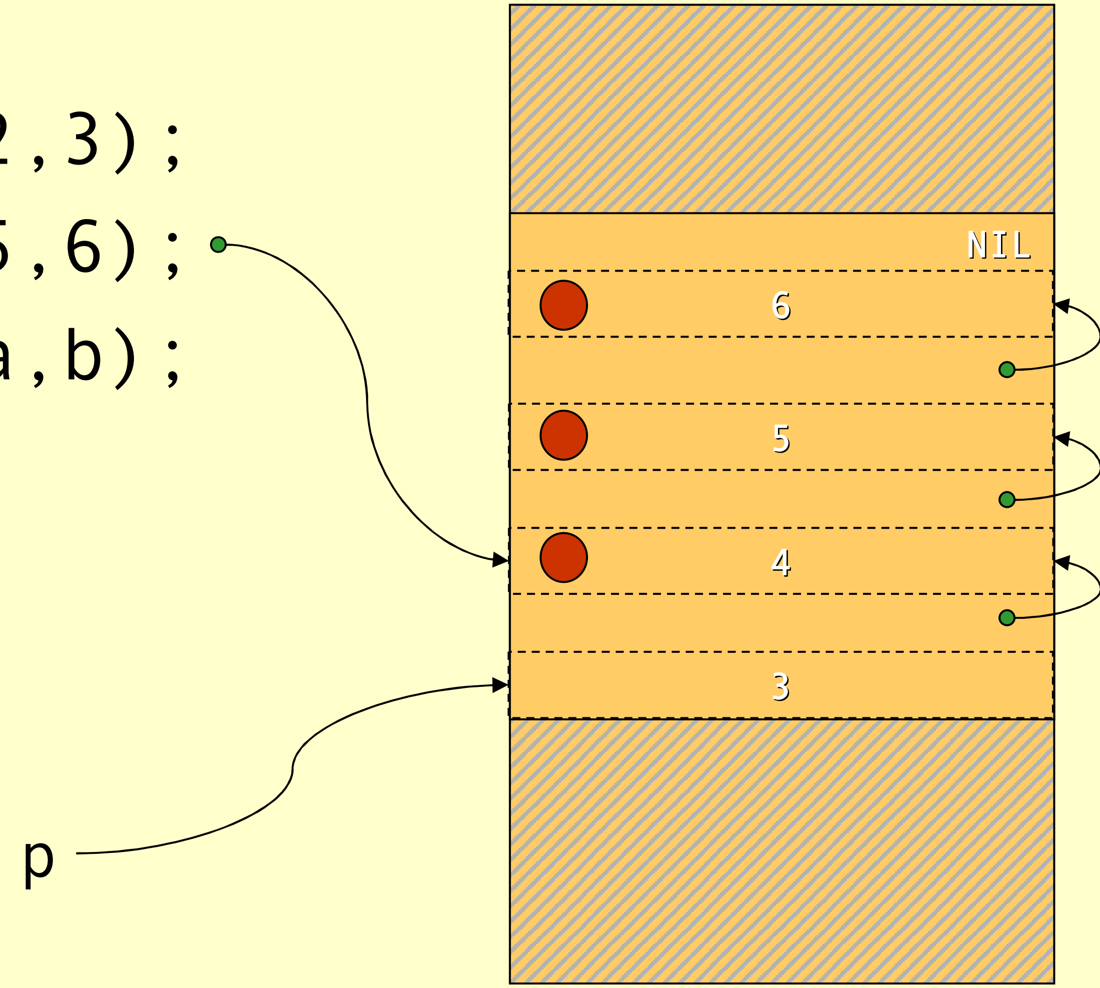
Example: Sweep

```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
return b;
```



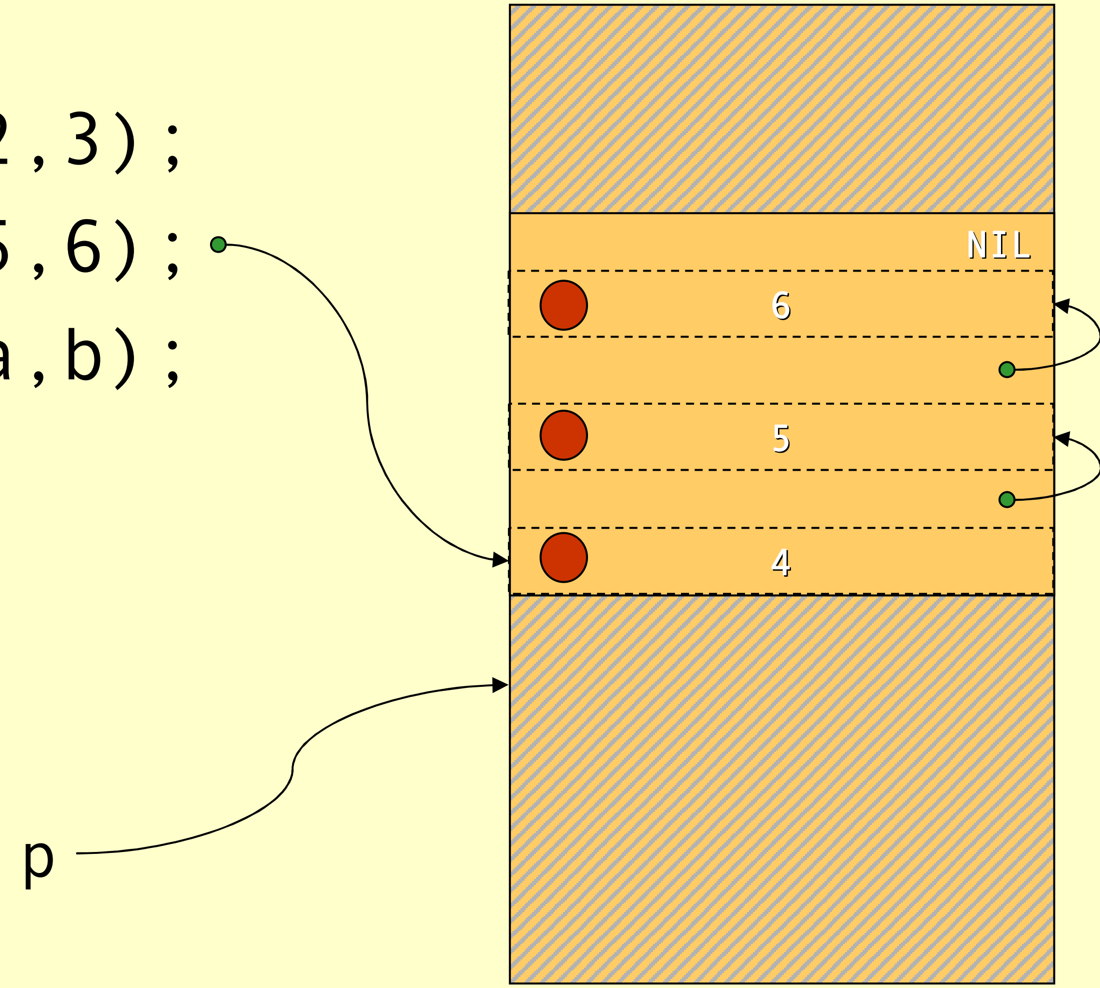
Example: Sweep


```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
return b;
```



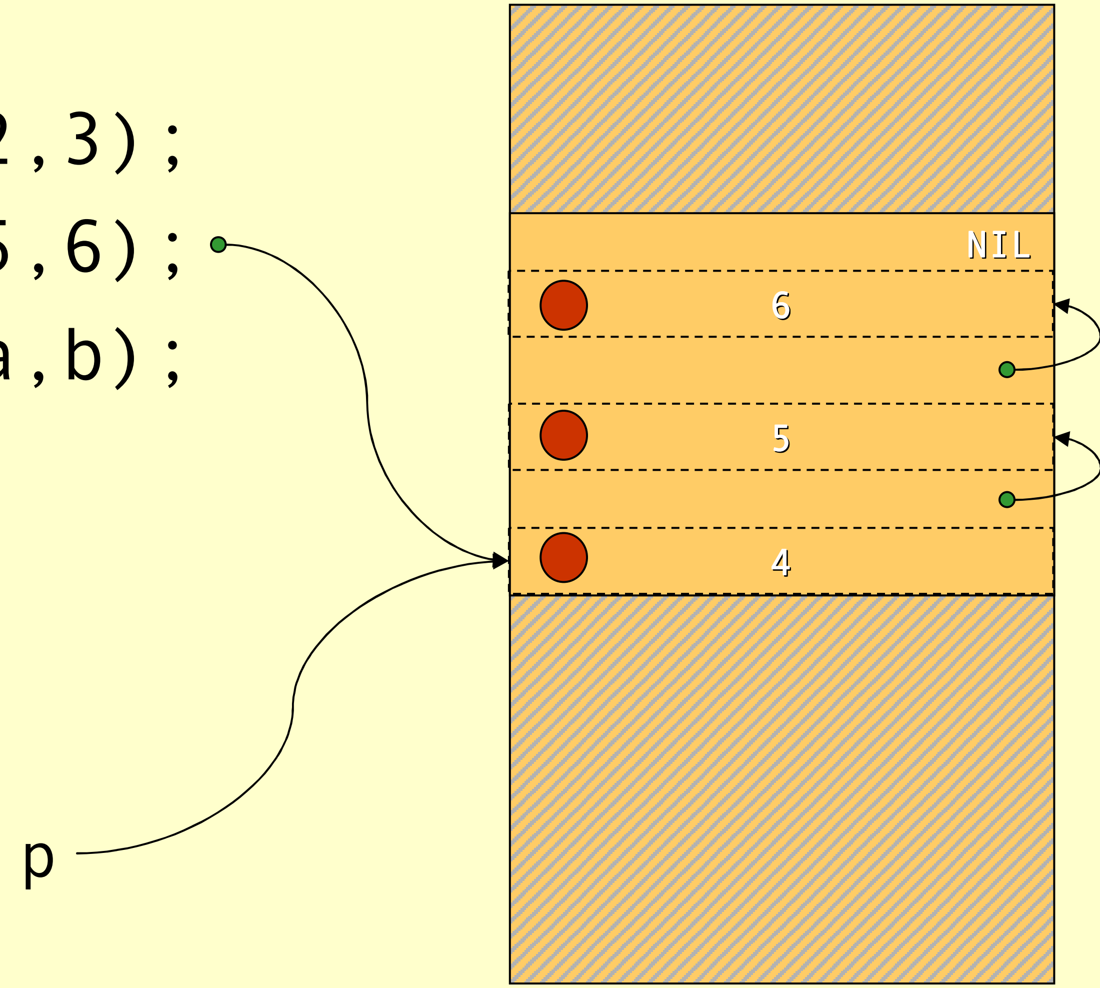
Example: Sweep

```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
return b;
```



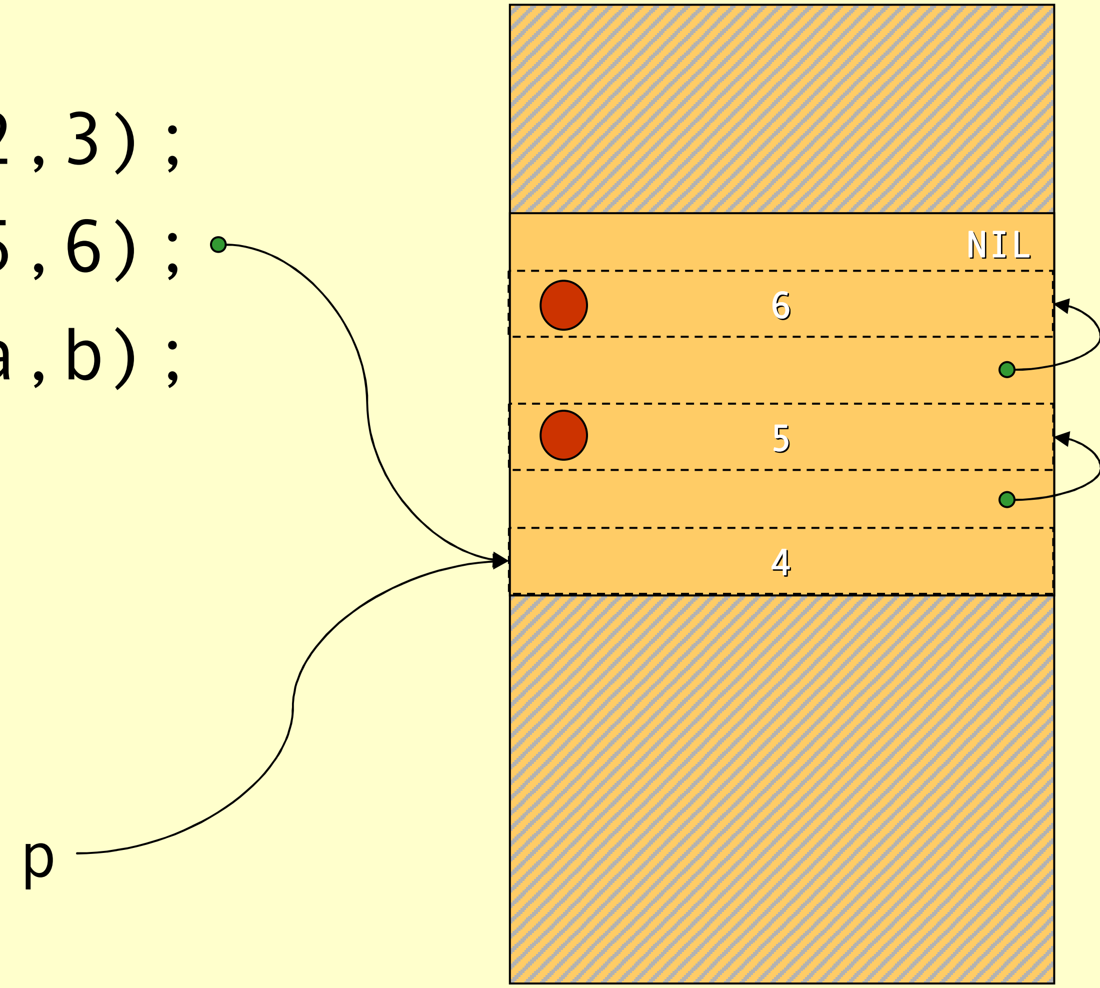
Example: Sweep

```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
return b;
```



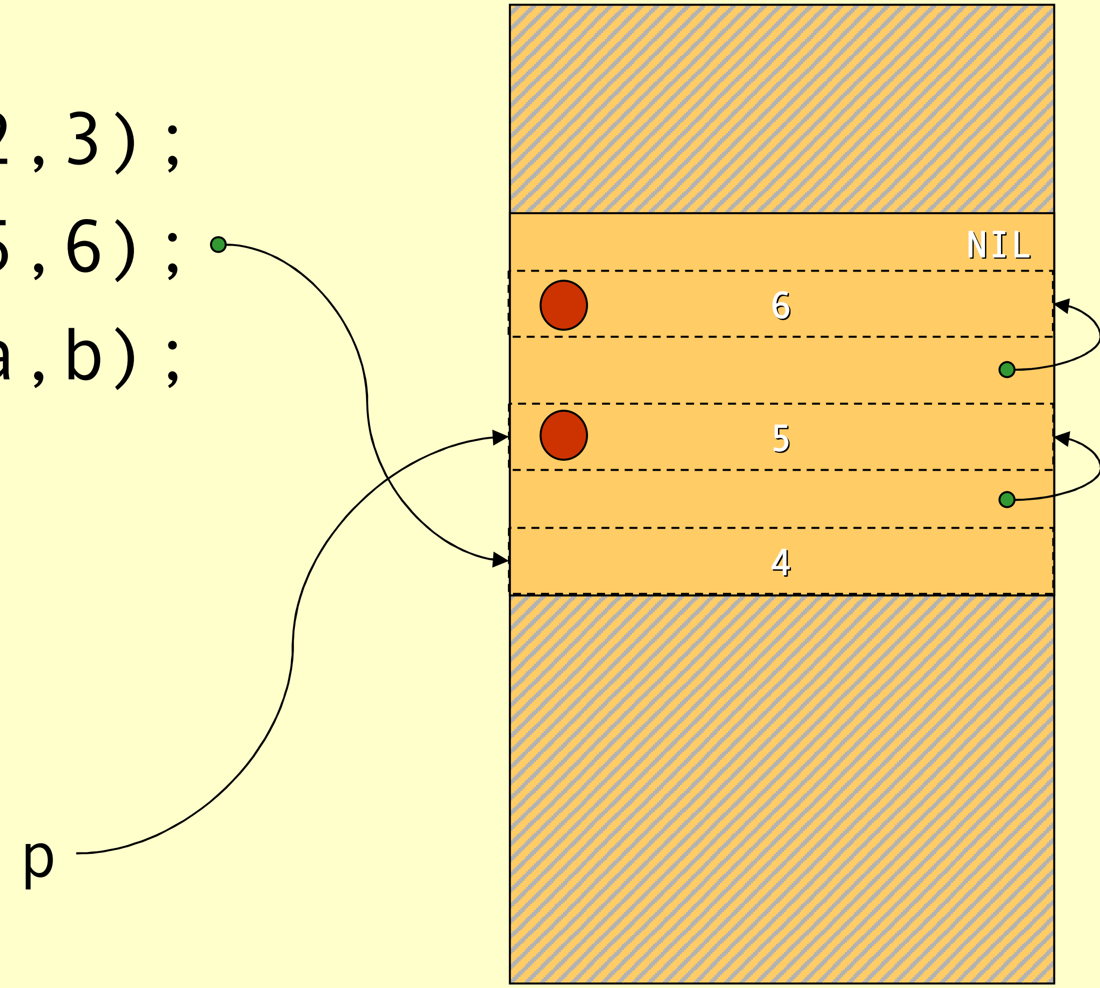
Example: Sweep

```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
return b;
```



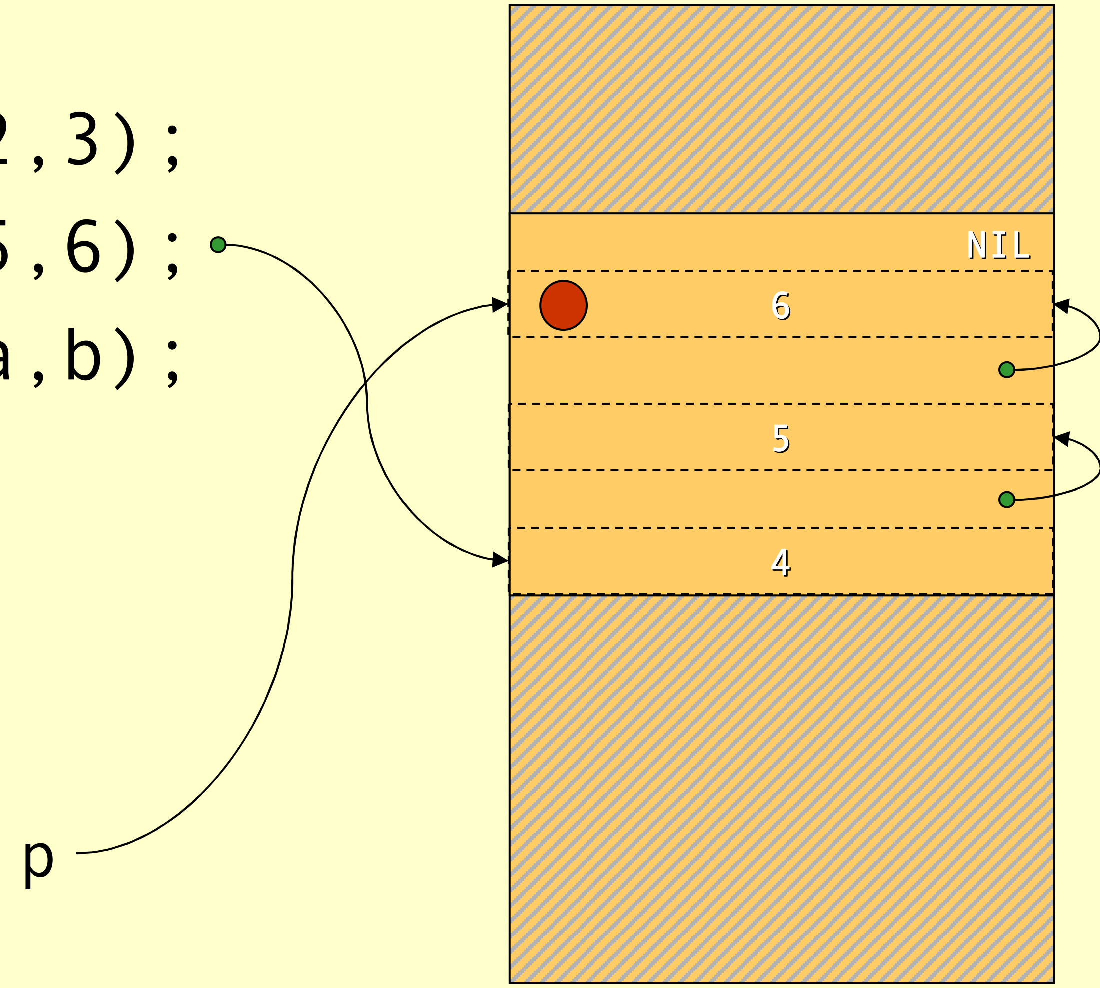
Example: Sweep

```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
return b;
```



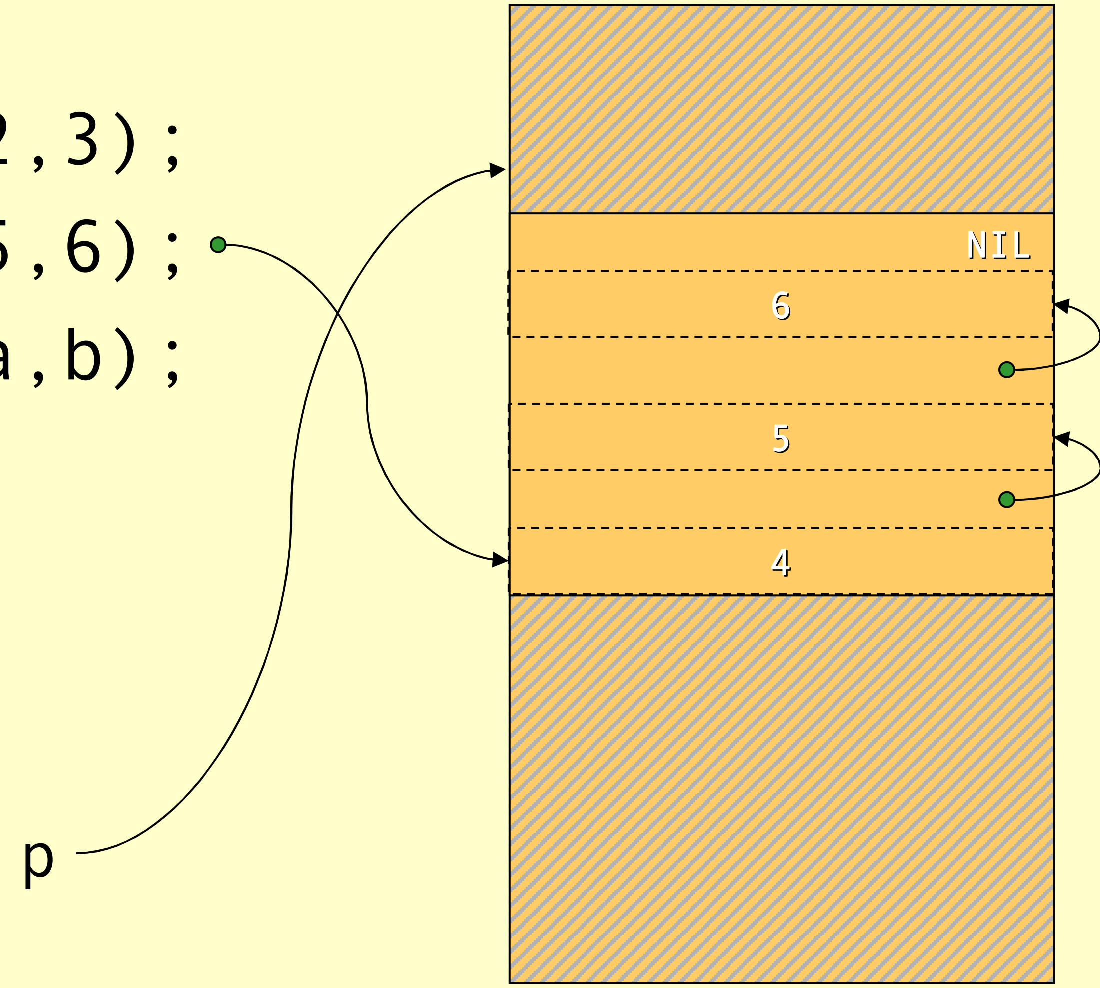
Example: Sweep

```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
return b;
```



Example: Sweep

```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
return b;
```



Cost of Mark & Sweep

- The mark phase takes time proportional to the amount of reachable data (R).
- The sweep phase takes time proportional to the size of the heap (H).
- The work done by the GC is to recover H-R words of memory.
- Them *amortized cost* of GC (overhead/allocated word) is:

$$\frac{c_1 R + c_2 H}{H - R}$$

- If $R \approx H$ the cost is very high. The cost goes down as the number of dead words increases.

Mark & Sweep

- Where do we store the mark bits?
  - We will discuss data representation a bit more at the end of the lecture. With some representations there will always be a tag or a header word in each heap object where the mark bit can be stored.
- They can be stored in a separate bitmap table:
  - If we have a 32-bit architecture and the smallest heap object is 2 words. (The three least significant bits == 0)
  - Then we can have 536,870,911 objects and need 67,108,863 bytes to store these bits.
  - This might seem to be a lot, but it is *only* **1.562%** of the total heap.

Tuning Mark & Sweep

- There is one problem with the mark phase:
  - While doing the depth first search we need to keep track of other paths to search.
  - If this is done with recursive calls we will need one allocation record **for each** level we descend in the reachability graph.
  - **Solutions**: Explicit stack or pointer reversal.

Mark & Sweep

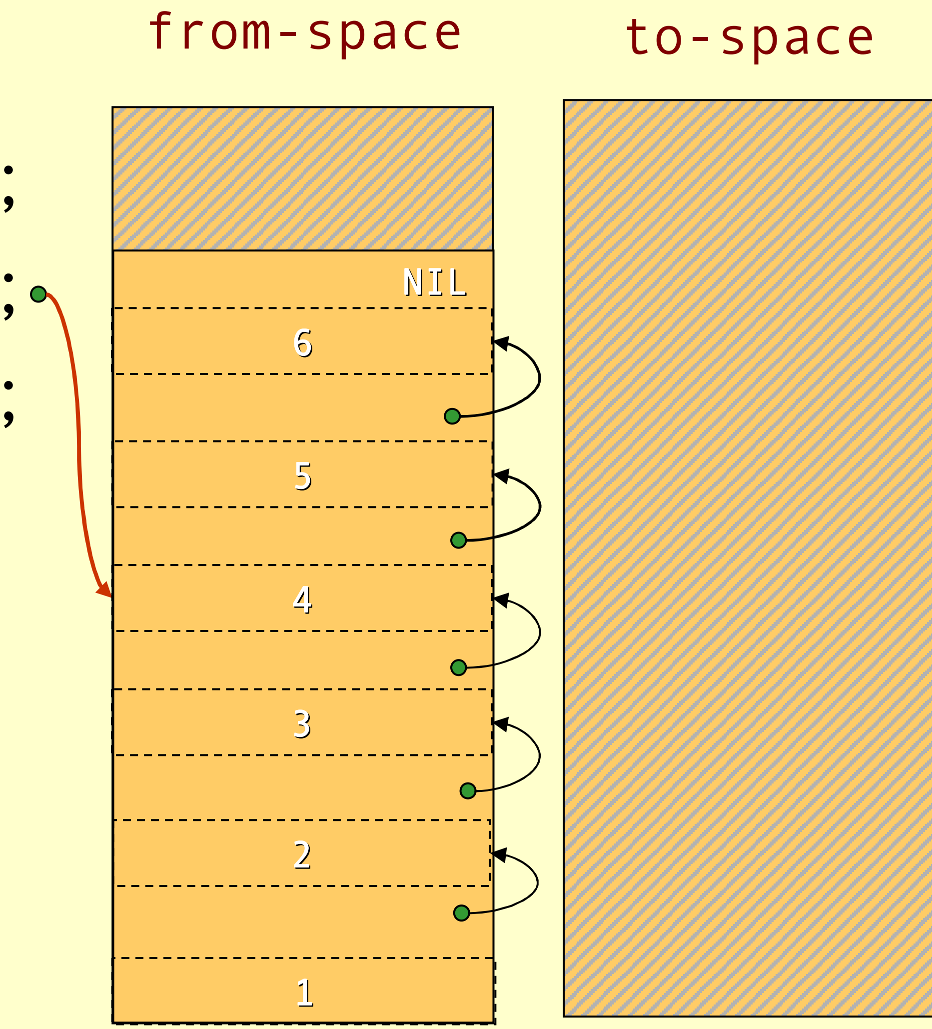
- Advantages with mark & sweep:
  - Can reclaim cyclic structures.
  - Standard version is easy to implement.
  - Can have relatively low space overhead.
- Disadvantages:
  - Fragmentation can become a problem.
  - Allocation from a free-list can be costly.

Copying Collector

- The idea of a copying garbage collector is to divide the memory space in two parts.
- Allocation is done linearly in one part (*from-space*).
- When that part is full all reachable objects are copied to the other part (*to-space*).

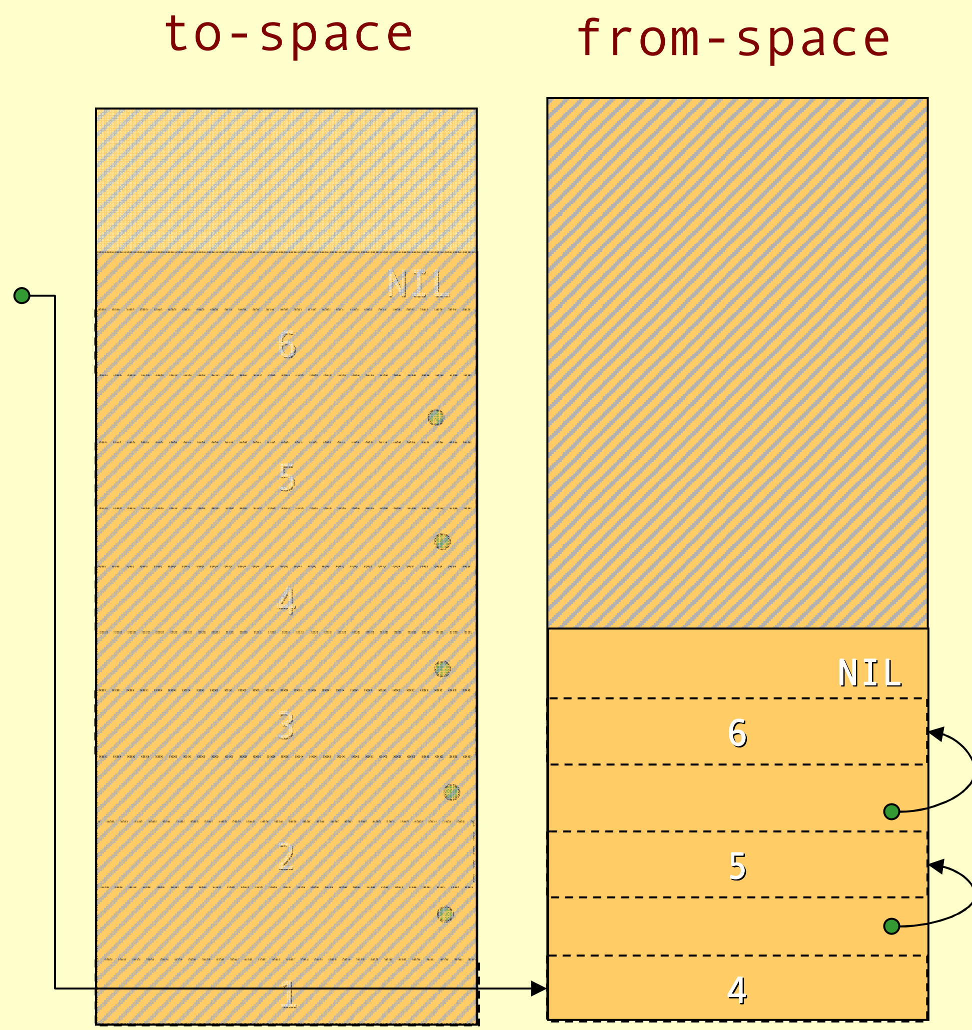
Before GC

```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
return b;
```



After GC

```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
return b;
```



Forwarding Pointers

- Given a pointer p that point to from-space make it point to to-space:
  - If p points to a from-space record that contains a pointer to to-space, then *p is a *forwarding-pointer* that indicates where the copy is. Set p=*p.
  - If *p has not been copied, copy *p to location `next`, `*p=next`, `p=next`, `next+=size(*p)`.

Cheney's Copying Collector

- Cheney's algorithm uses breadth-first to traverse the live data.
- The algorithm is non-recursive, requires no extra space or time consuming tricks (such as pointer reversal), and it is very simple to implement.
- The disadvantage is that breadth-first does not give as good locality of references as depth-first.

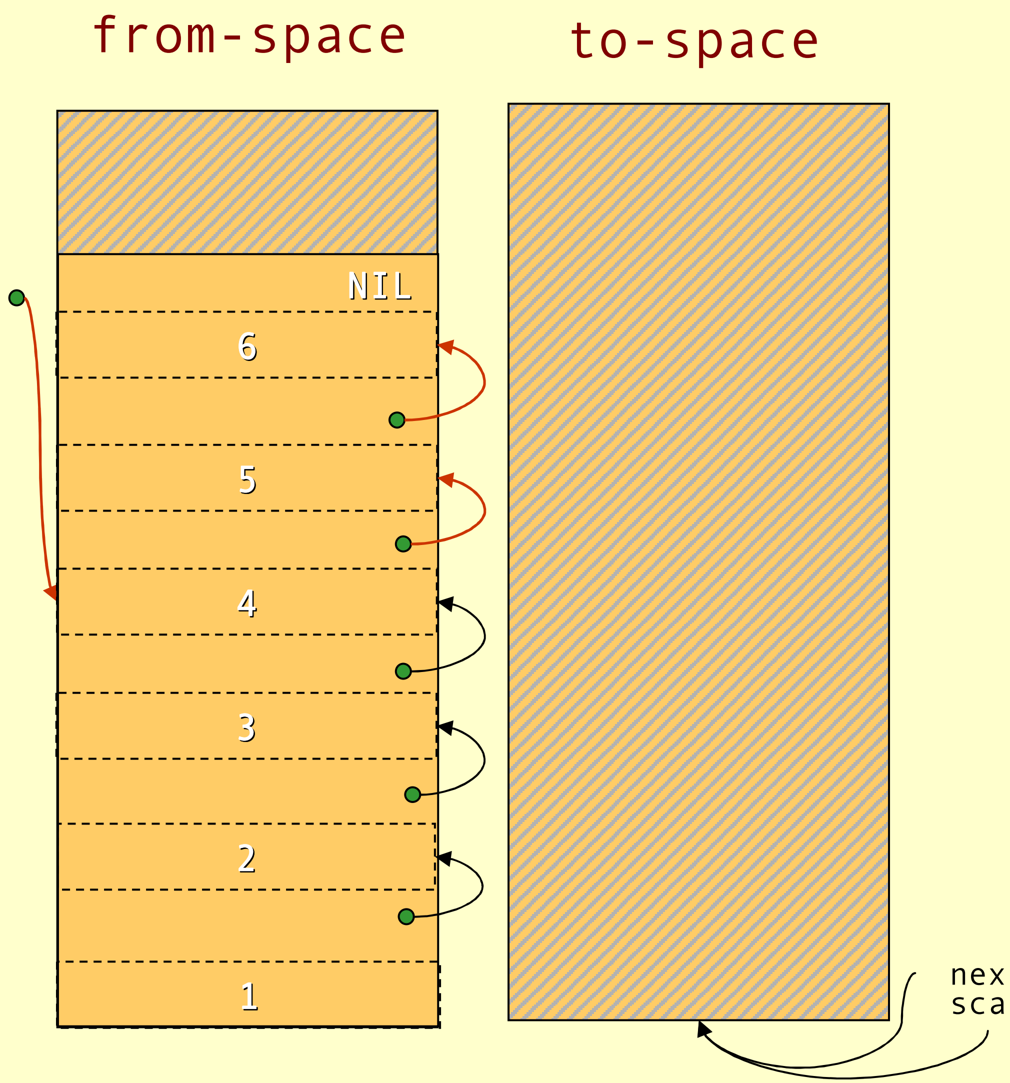
Cheney's Copying Collector

- The algorithm:
  1. Forward all roots.
  2. Use the area between `scan` and `next` as a queue for copied records whose children has yet not been forwarded.

```
scan = next = start of to-space
for each root r { r = forward(r); }
while scan < next {
  for each field f of *scan
   scan->f = forward(scan->f)
  scan += size(*scan)
}
```

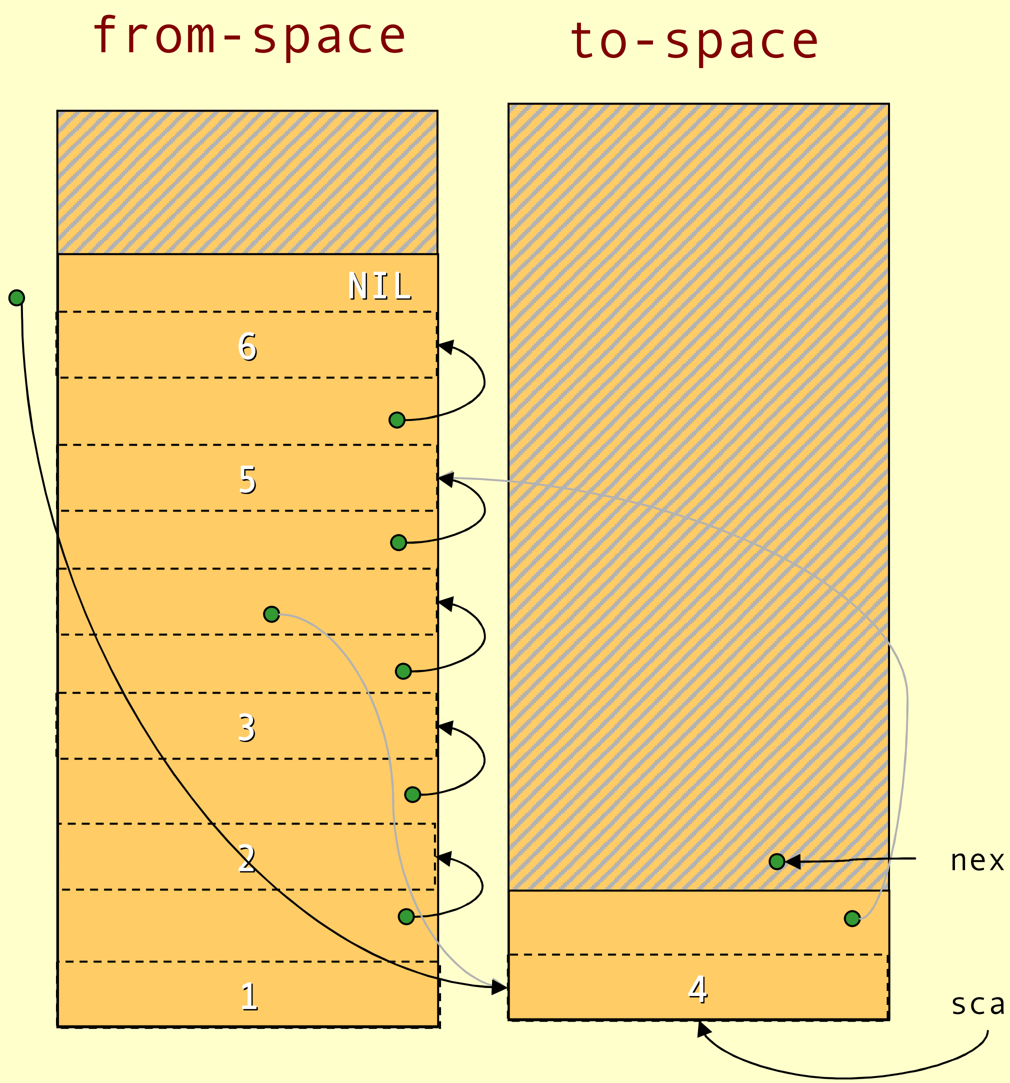
Before GC

```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
return b;
```



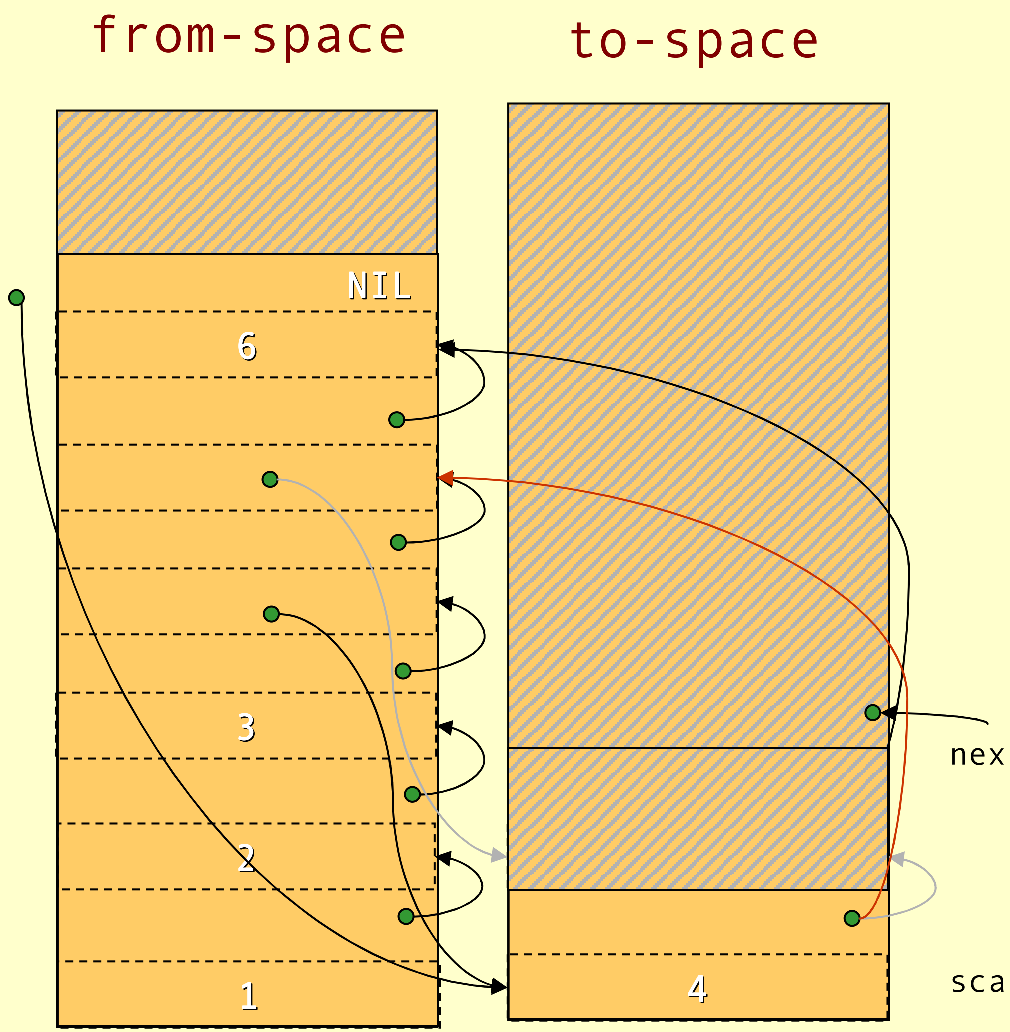
Forward Roots

```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
return b;
```



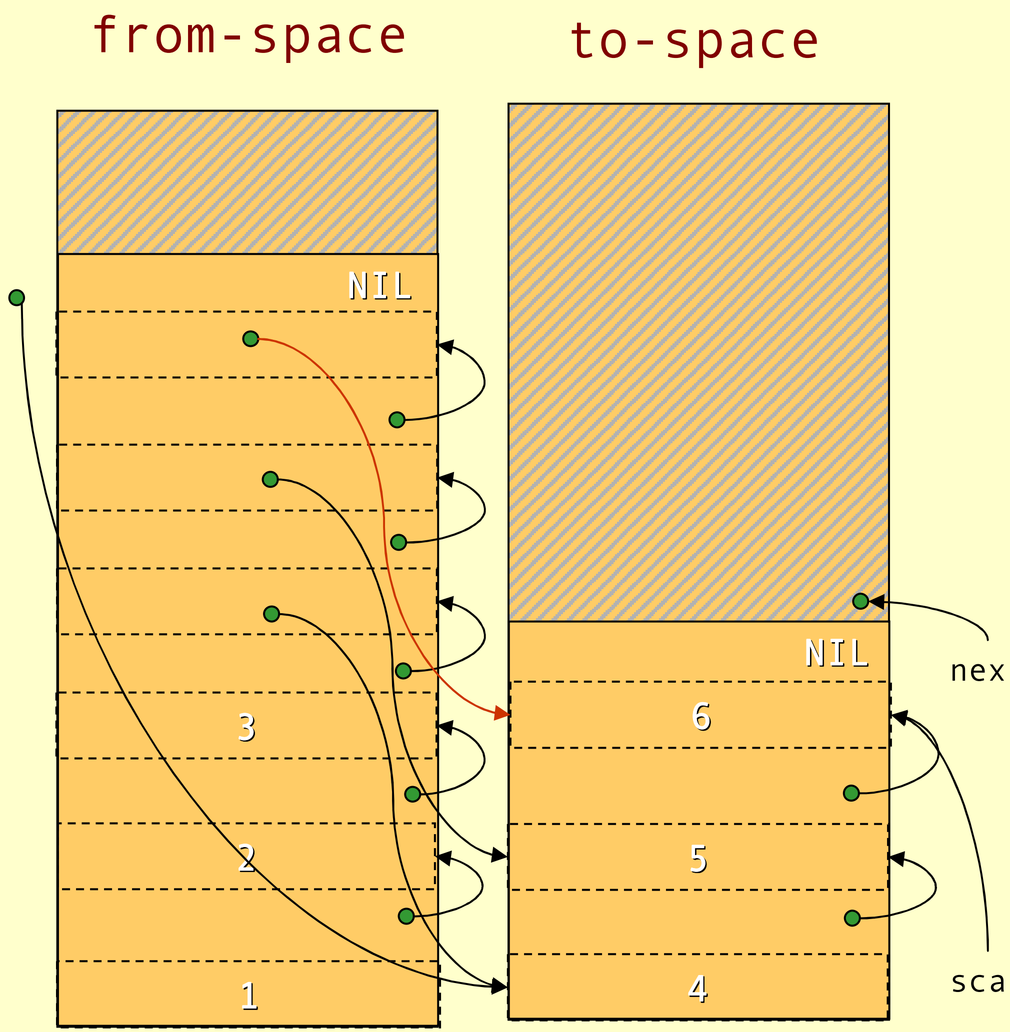
Scanning

```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
return b;
```



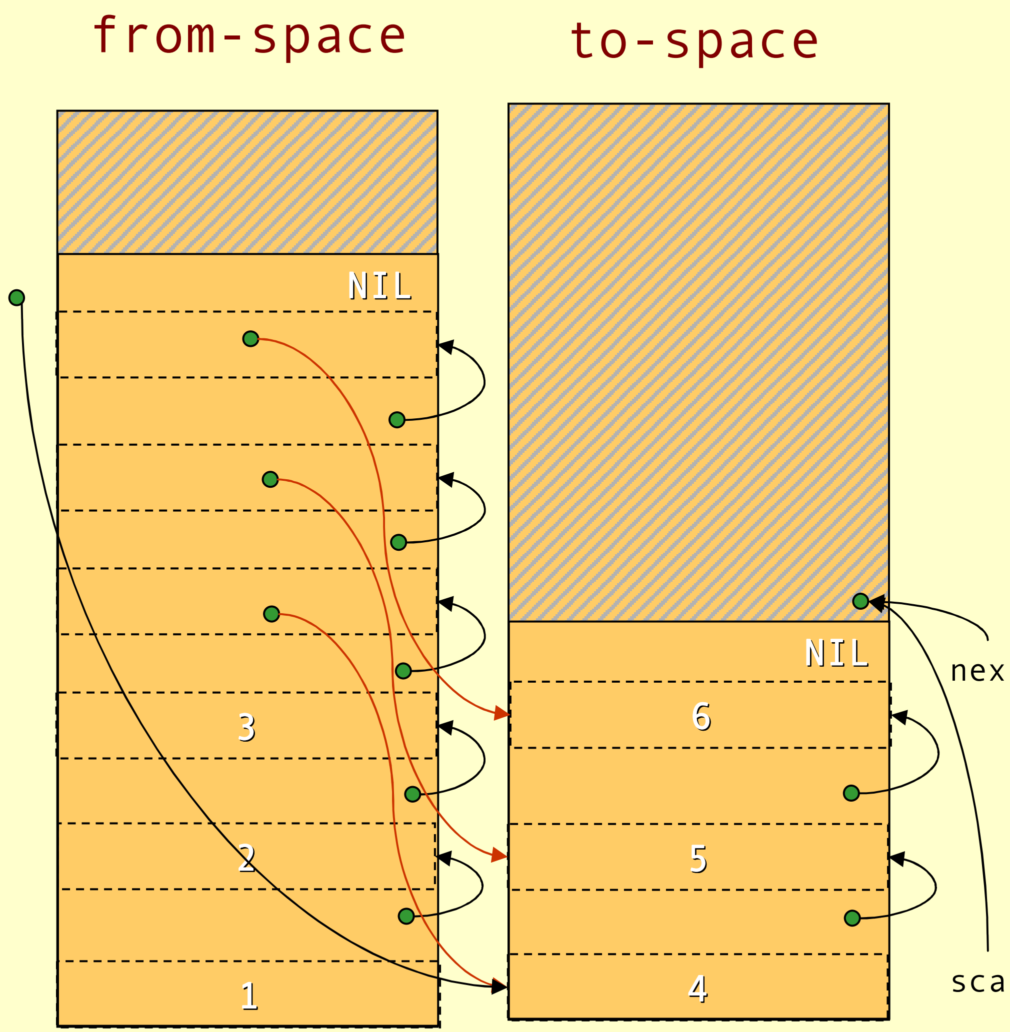
Scanning

```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
return b;
```



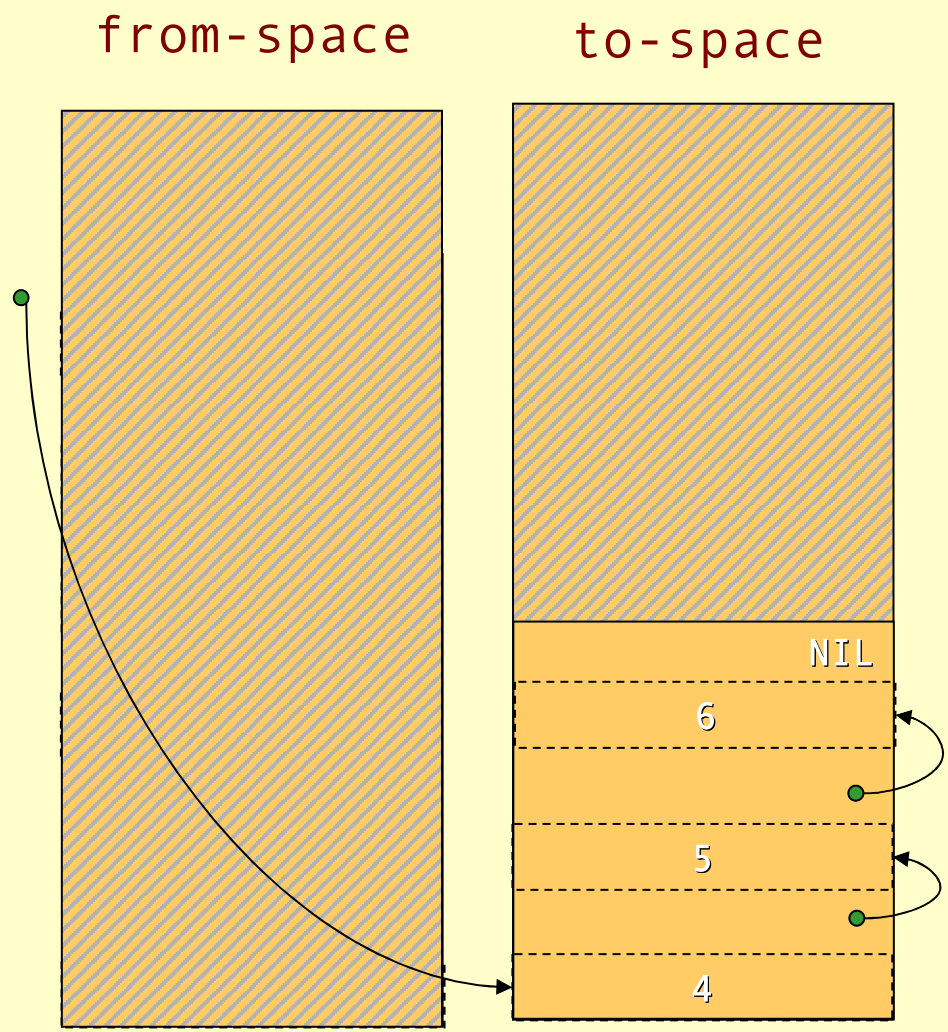
Scanning

```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
return b;
```



Scanning

```
list a = List(1,2,3);
list b = List(4,5,6);
list c = append(a,b);
printList(c);
doLotsOfStuff();
return b;
```



Cost of Copying GC

- The GC takes time proportional to the amount of reachable data (R).
- The work done by the GC is to recover H/2 - R words of memory.
- The *amortized cost* of GC (overhead/allocated word) is:

$$\frac{c_1 R}{(H/2) - R}$$

- If H is much larger than R then the cost approaches zero.
- The GC is often self-tuning (by changing the size of H) so that H = 4R, giving a GC cost of c_1 per allocated word.

Copying GC

- Advantages of copying GC:
  - Can handle cyclic structures.
  - Very easy to implement.
  - Extremely fast allocation (no free-list) just a check and heap pointer increment.
  - Automatic compaction: no fragmentation.
  - Only visits live data – time only proportional to live data.
- Disadvantages of copying GC:
  - Double the space overhead since two heaps are needed.
  - Long lived live data might be copied several times.
  - Copying all the live data might lead to long stop times.

Generational GC

- Empirical observation: most objects die young. The longer an object lives the higher the probability it will survive the next GC.
- The benefit of GC is highest for young objects.
- Idea: Keep young objects in a small space (called *nursery*) which is GC-ed more often than the whole heap.
- With such a *generational GC* each collection takes less time and yields proportionally more space.

Generational GC

- In a generational GC we want to collect the younger generation without having to look at older generations.
- But we have to consider all pointers from older generations to younger generations as roots.
  - (In a language without destructive updates this is not a problem, since there are no such pointers.)
- These inter-generational references must be remembered (e.g., by keeping a *remembered set*). The compiler has to ensure that all store operations in an older generation are checked.

Cost of Generational GC

- It is common for the youngest generation to have less than 10% live data.
- With a copying collector H/R = 10 in this generation.
- The *amortized cost* of a *minor* collection is:

$$\frac{c_1 R}{(10\ R) - R}$$

- Performing a major collection can be very expensive.
- Maintaining the remembered set also takes time. If a programs does many updates of old objects with pointers to new objects a generational GC can be more expensive than a non-generational GC.

Incremental GC

- An *incremental* (or *concurrent*) GC keeps the stop-times down by interleaving GC with program execution.
  - The *collector* tries to free memory while the program, called the *mutator* changes the reachability graph.
- An incremental GC only operates at request from the mutator.
- A concurrent GC can operate in between any two mutator instructions.

Data Layout

- The compiler and the runtime system should agree on a *data layout*.
  - The GC needs to know the size of records, and which fields of a record contains pointers to other records.
- Another approach is to not give any information to the collector about which fields are pointers.
  - The collector must then make a *conservative guess*, and treat all words that **looks like** pointers to the heap as such.
  - Since it is unsafe to change such pointers a *conservative collector* has to be non-moving.

Data Layout

- In statically typed or OO languages, each record can start with a *header word* that points to a description of the type or class.
- In many functional languages the set of data types can not be extended; for such languages one can use a *tagging scheme* where unused bits in a pointer indicate what data type it points to.

The Root Set

- The set of registers and stack slots that contain live data can be described by a *pointer map* (*stack map*).
- For each pointer that is live after a function call the pointer map identifies its register or stack slot.
- The *return address* can be used as a key in a hash map to find the pointer map.
- To mark/forward the roots the GC starts at the top of the stack and scans downwards frame by frame. (In a generational collector the stack scan can also be made generational.)

Finalizers

- Some languages (notably OO) have *finalizers*, that is, some code that should be executed before some data is deallocated.
- This is, e.g., useful to make sure that an object frees all resources (open files, locks, etc) before dying.
- With a copying collector the handling of finalizers becomes more difficult. Such a GC does not normally visit the dead data. So all finalizers have to be remembered and after GC a check has to be done to see if any freed data triggers a finalizer.
- A mark & sweep collector does not have this problem, but just as with a copying collector it might take a long time after the last use before garbage is actually collected.
- If one wants to ensure that a finalizer is executed as soon as the object dies then one has to use reference counting.

Summary

- Manual allocation is unsafe and **should not be used**. (It also comes at a cost, maintaining a free-list is not for free.)
- Garbage collection solves the problem of automatic memory management.
- In most cases a generational copying collector will be the most efficient solution.