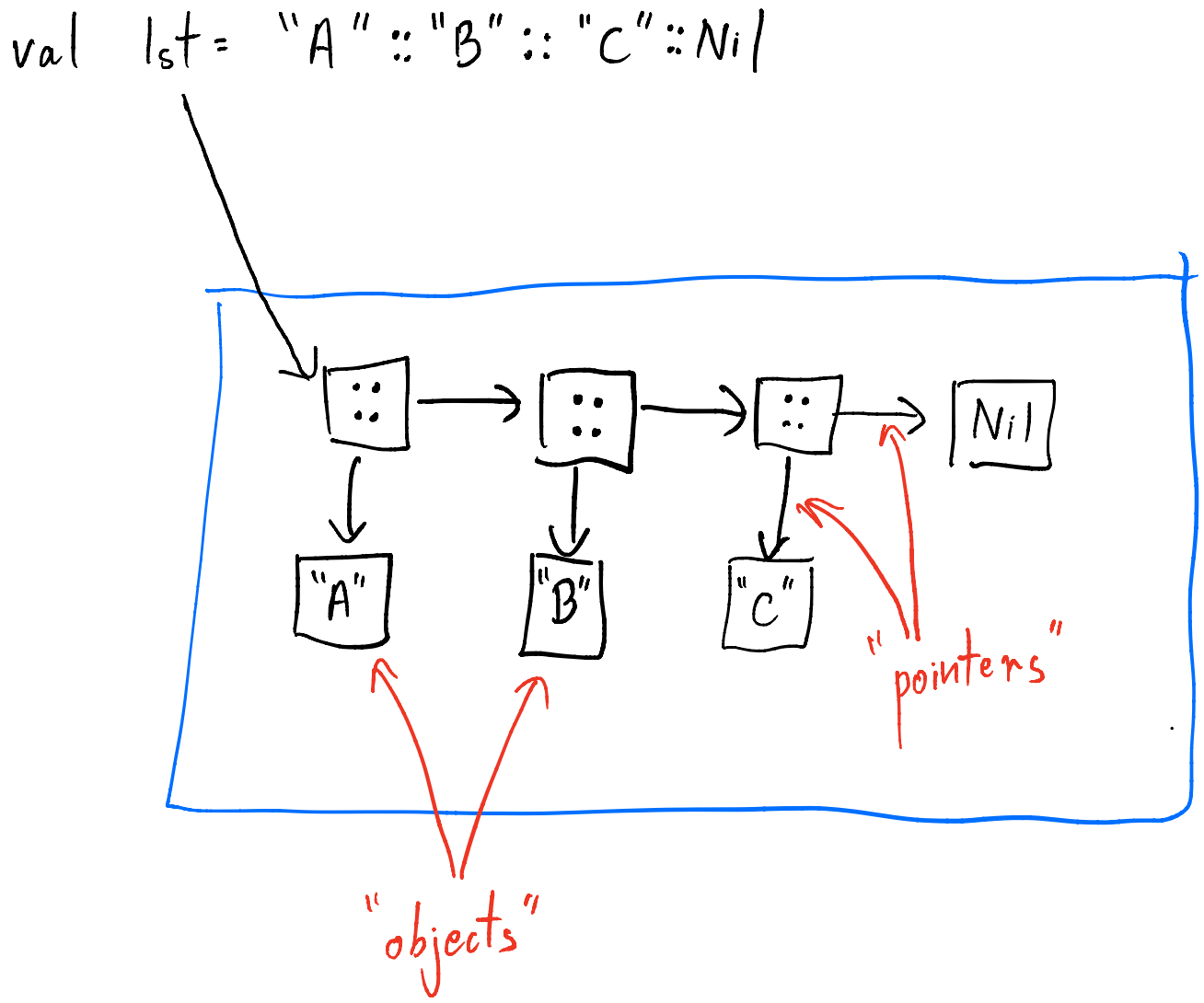
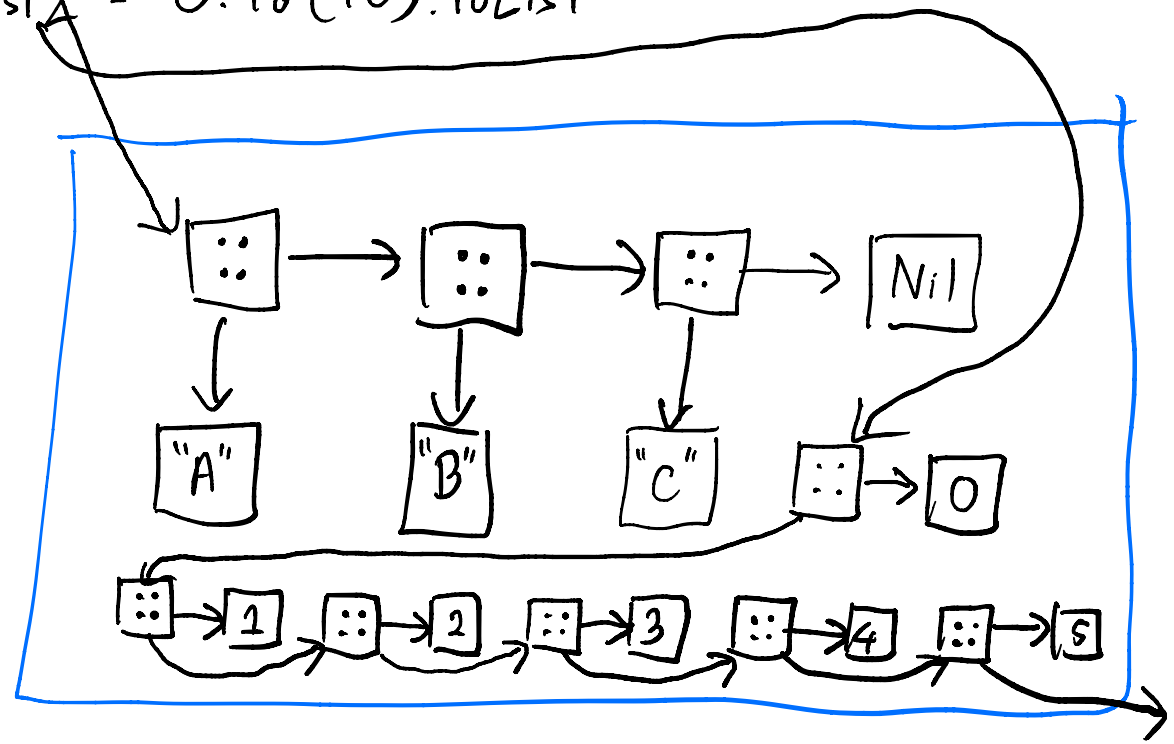
```
def makeAdder(x:Int):Int=>Int={ def add(y:Int)={x+y}; add }
val addTen = makeAdder(10) ; addTen(50)
= val addTen={def add(y:Int)={10+y}; add } ; addTen(50)
= val addTen={val add = (y:Int)=>{10+y}; add } ; addTen(50)
= val addTen={(y:Int)=> 10+y } ; addTen(50)
= def addTen(y:Int)={10+y}; addTen(50)
= 10+50
= 60
```

Equational Reasoning

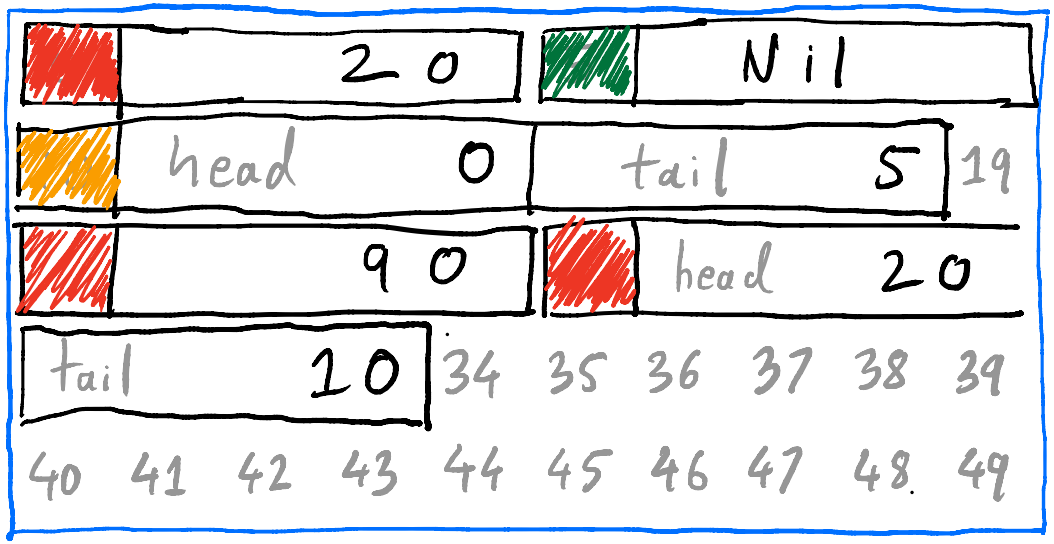

val lst = "A" :: "B" :: "C" :: Nil
Nil
"A"
"B"
"C"
"pointers"
"objects"

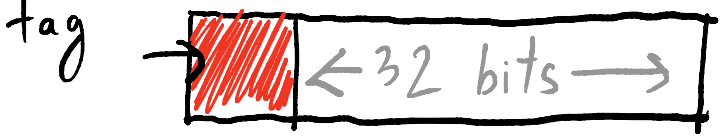
```
val lst = "A" :: "B" :: "C" :: Nil
val lst2 = 0.to(10).toList
```



0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49

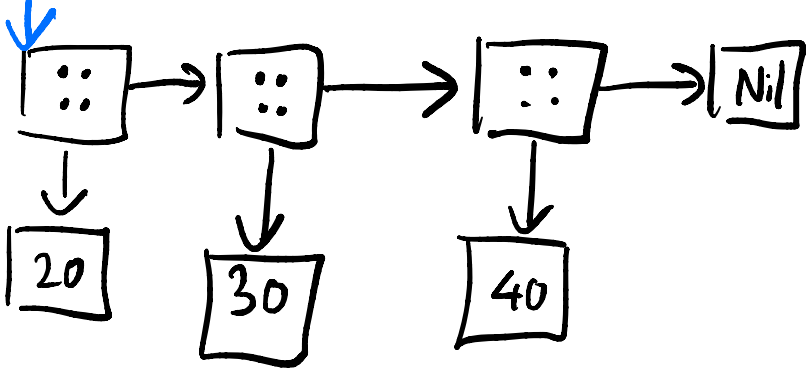
```
val n = 20
val lst = n :: Nil
val lst2 = 90 :: lst
```





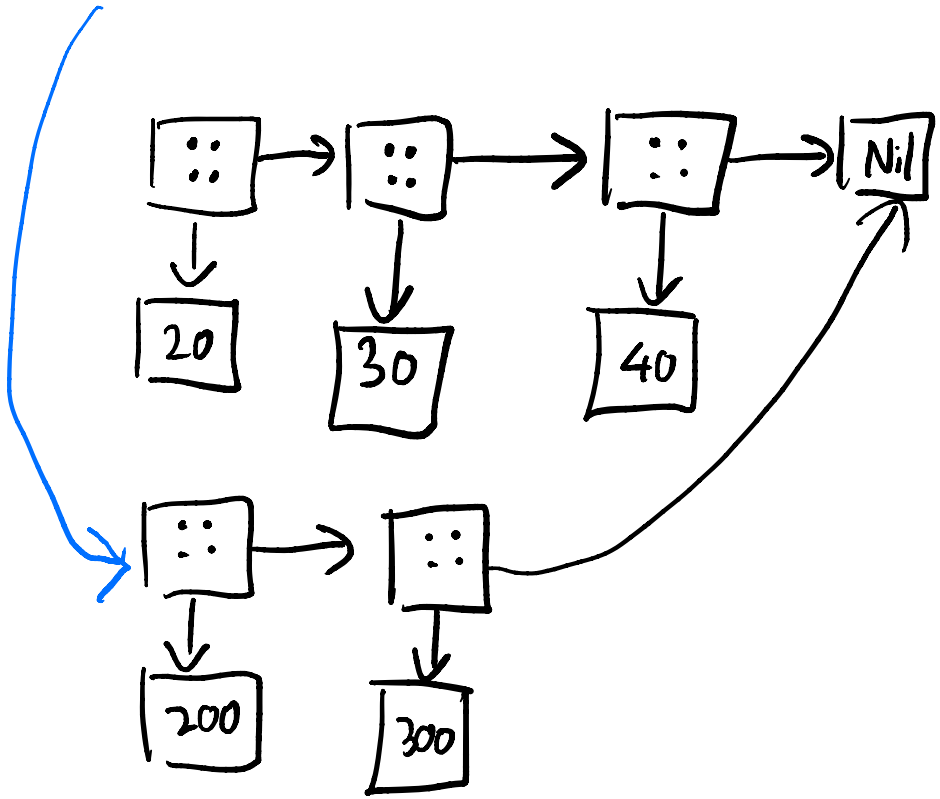
⇒ var lst = 20 :: 30 :: 40 :: Nil

lst = 200 :: 300 :: Nil



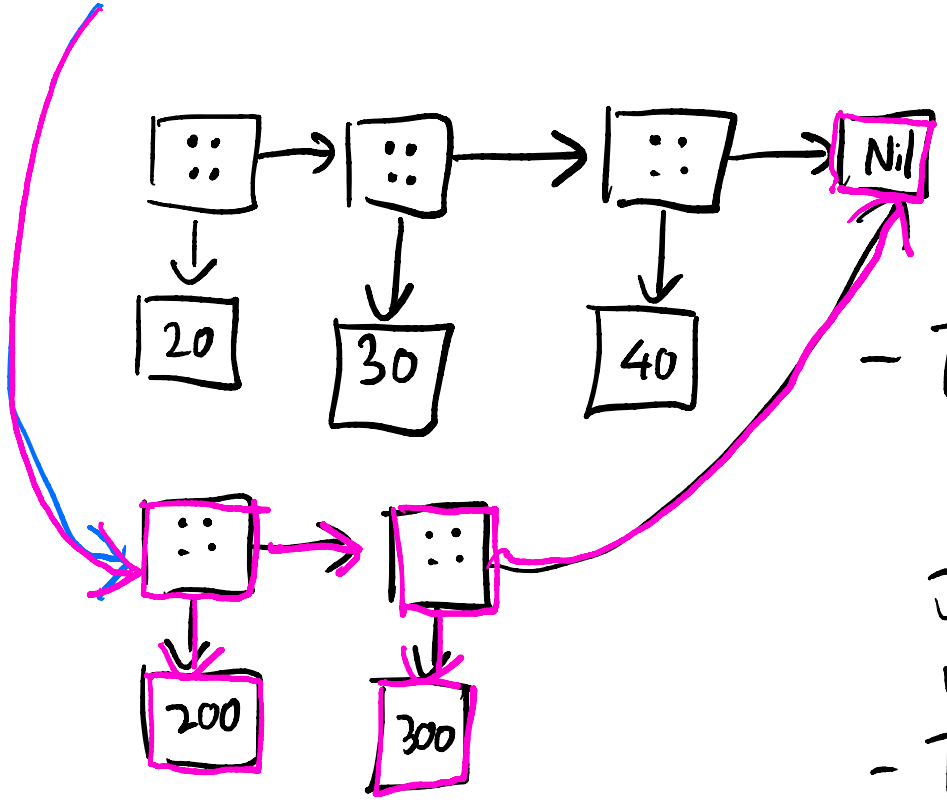
var lst = 20 :: 30 :: 40 :: Nil

⇒ lst = 200 :: 300 :: Nil



var lst = 20 :: 30 :: 40 :: Nil

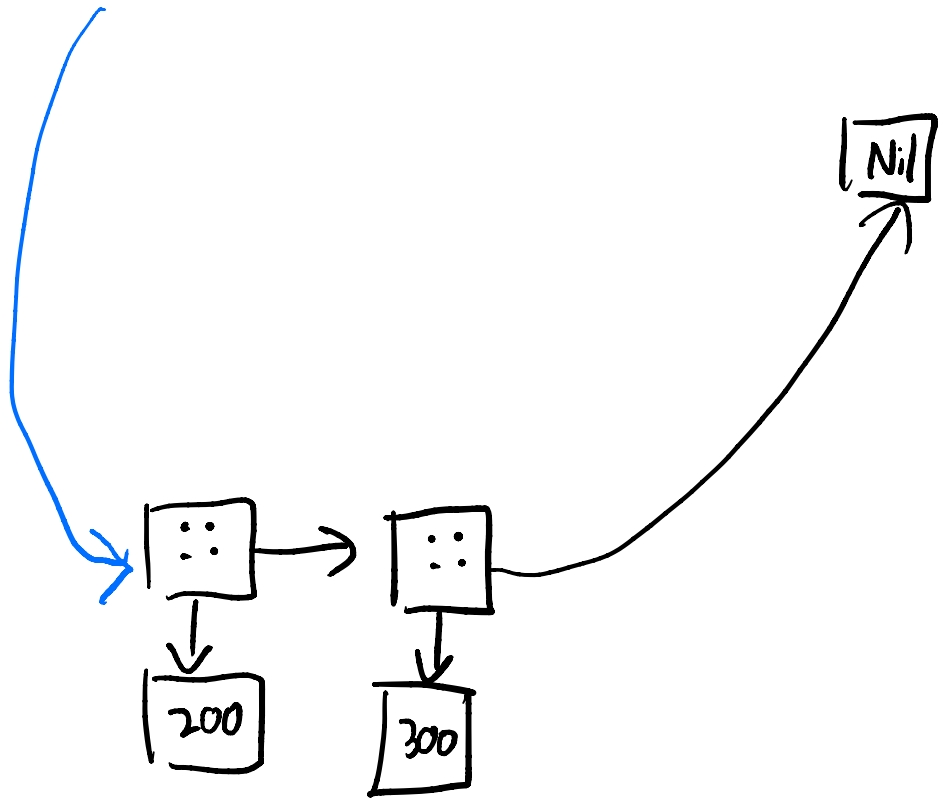
⇒ lst = 200 :: 300 :: Nil



- The pink objects are reachable from the lst variable.
- The other objects are not reachable.

var lst = 20 :: 30 :: 40 :: Nil

⇒ lst = 200 :: 300 :: Nil



- The pink objects are reachable from the lst variable.
- The other objects are not reachable.

↓

So, we can just delete them!

The "Mark and Sweep" Algorithm

1. **Mark Phase.** Starting from the variables in the program (specifically, the variables on the stack), mark (in pink) the objects in memory that are reachable by following pointers in the object graph.
2. **Sweep Phase.** Delete the objects that were not marked.
3. Unmark remaining objects

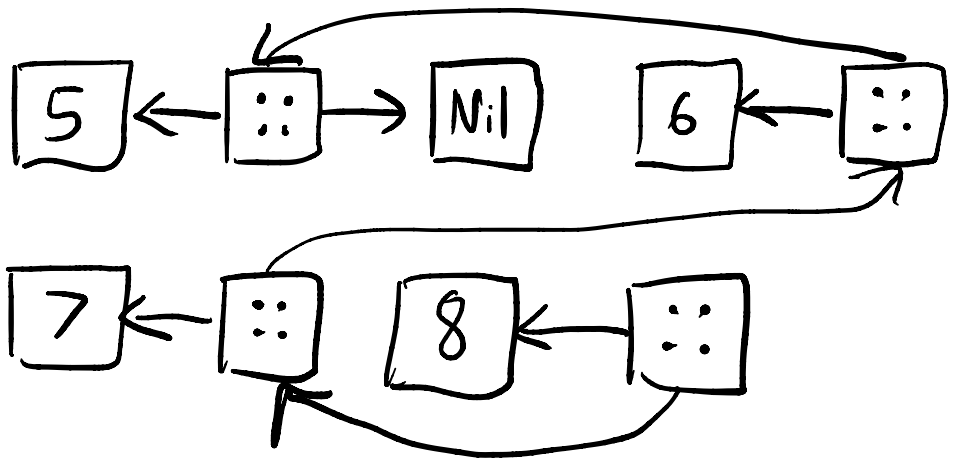
The "Mark and Sweep" Algorithm is slow on large heaps

O(n) — n = number of objects that are "alive"

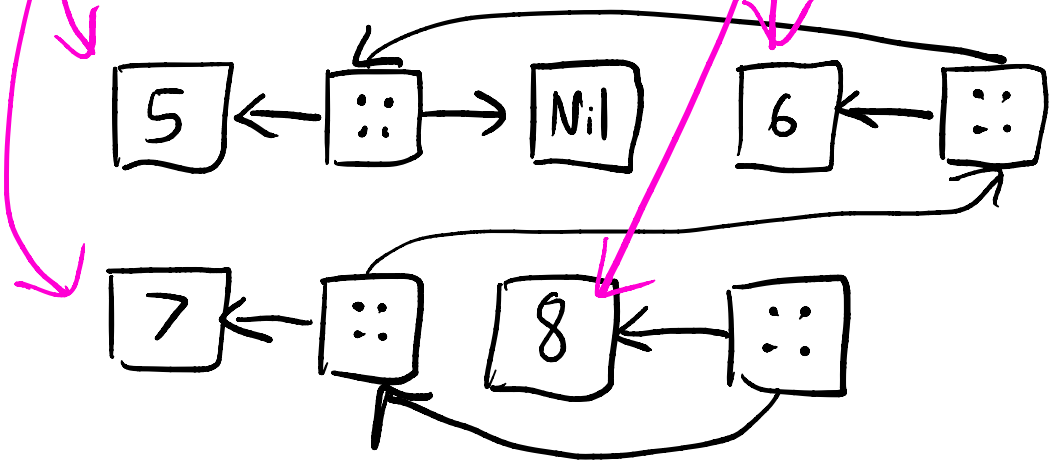
1. **Mark Phase.** Starting from the variables in the program (specifically, the variables on the stack), mark (in pink) the objects in memory that are reachable by following pointers in the object graph.
2. **Sweep Phase.** Delete the objects that were not marked.
3. Unmark remaining objects ← O(n)

O(m) — where m is the size of memory

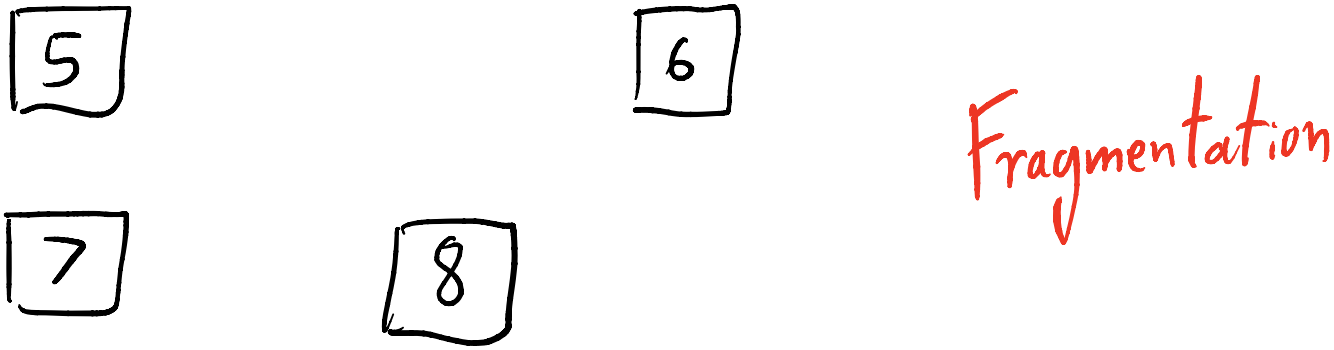
```
val x = 5 ; var l1 = x :: Nil ; val y = 6 ; var l2 = y :: l1 ;
val z = 7 ; var l3 = z :: l2 ; val w = 8 ; var l4 = w :: l2 ;
```



```
val x = 5 ; var l1 = x::Nil; val y = 6; var l2 = y:: l1;
val z = 7; var l3 = z:: l2; val w = 8; var l4 = w:: l2;
l1 = null; l2 = null; l3 = null; l4 = null;
```



```
val x = 5 ; var l1 = x :: Nil ; val y = 6 ; var l2 = y :: l1 ;
val z = 7 ; var l3 = z :: l2 ; val w = 8 ; var l4 = w :: l2 ;
l1 = null ; l2 = null ; l3 = null ; l4 = null ;
```

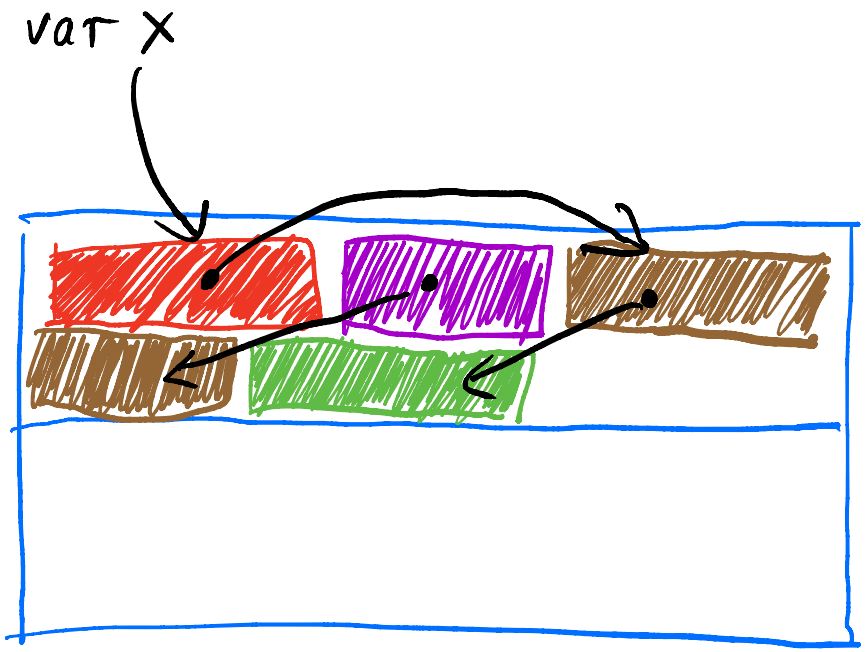


Fragmentation

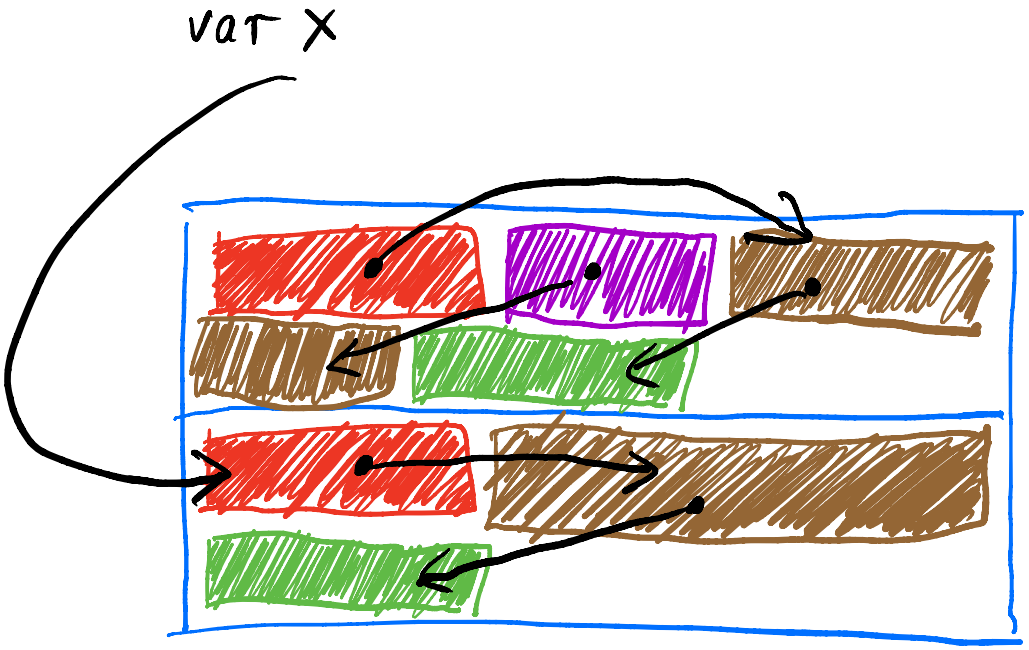
Low-level Details

1. Need a "free list" of blocks of unused memory.
2. To allocate a new object, need to search for a large enough block in the free list.
3. If no block can be found, may need to "compact" memory by moving objects closer together.

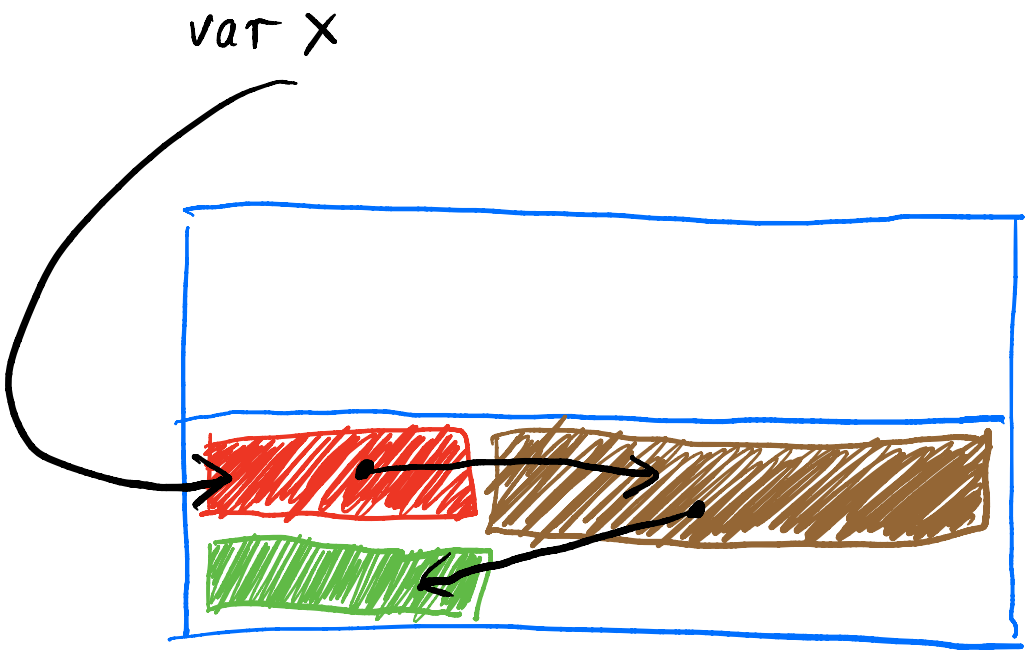
The "Stop-and-Copy" Algorithm



The "Stop-and-Copy" Algorithm



The "Stop-and-Copy" Algorithm



Compaction "for free"

Stop and Copy Algorithm

1. Divide the heap into two halves ("semispaces").
2. Allocate objects in one half.
3. When one half is full, copy reachable objects to the other half. (Update pointers to point to new copies).
4. Allocate objects in the new half.
5. Keep alternating between the two halves.

Half of memory is lost!

No need for a free-list, just need 1 pointer to end of heap

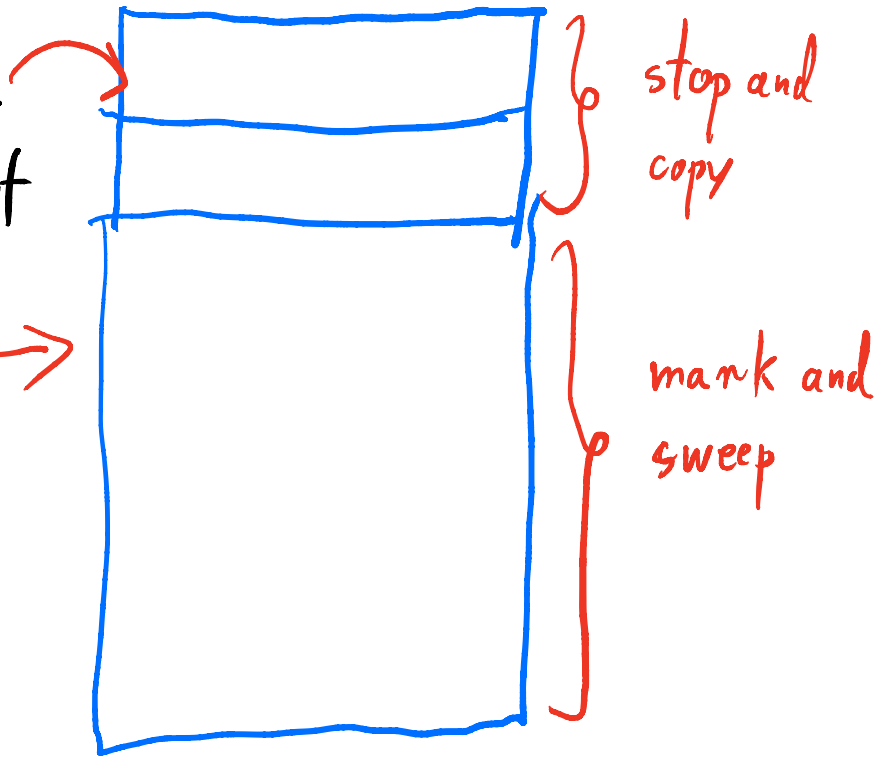
Copying large objects...

Stop and Copy Algorithm

1. Divide the heap into two halves ("semispaces")
2. Allocate objects in one half.
3. When one half is full, copy reachable objects to the other half. (Update pointers to point to new copies).
4. Allocate objects in the new half.
5. Keep alternating between the two halves.

Generational Garbage Collector

1. Allocate new objects in the nursery.
2. If an object survives n rounds of stop-and-copy, move it to the old generation.
3. Allocate large objects directly in the old generation. (To avoid moving them.)



Generational Hypothesis: New objects tend to die young. (High infant mortality.)

Old objects tend to live for a long time.

Why does GC work?

- No pointer arithmetic.
- No way to directly address memory.

contrast to C/C++