Languages and Compilers (SProg og Oversættere)

Heap allocation and Garbage Collection

Heap allocation and Garbage Collection

- Why may we need heap allocation?
- Garbage collection strategies (Types of GCs)

Terminology

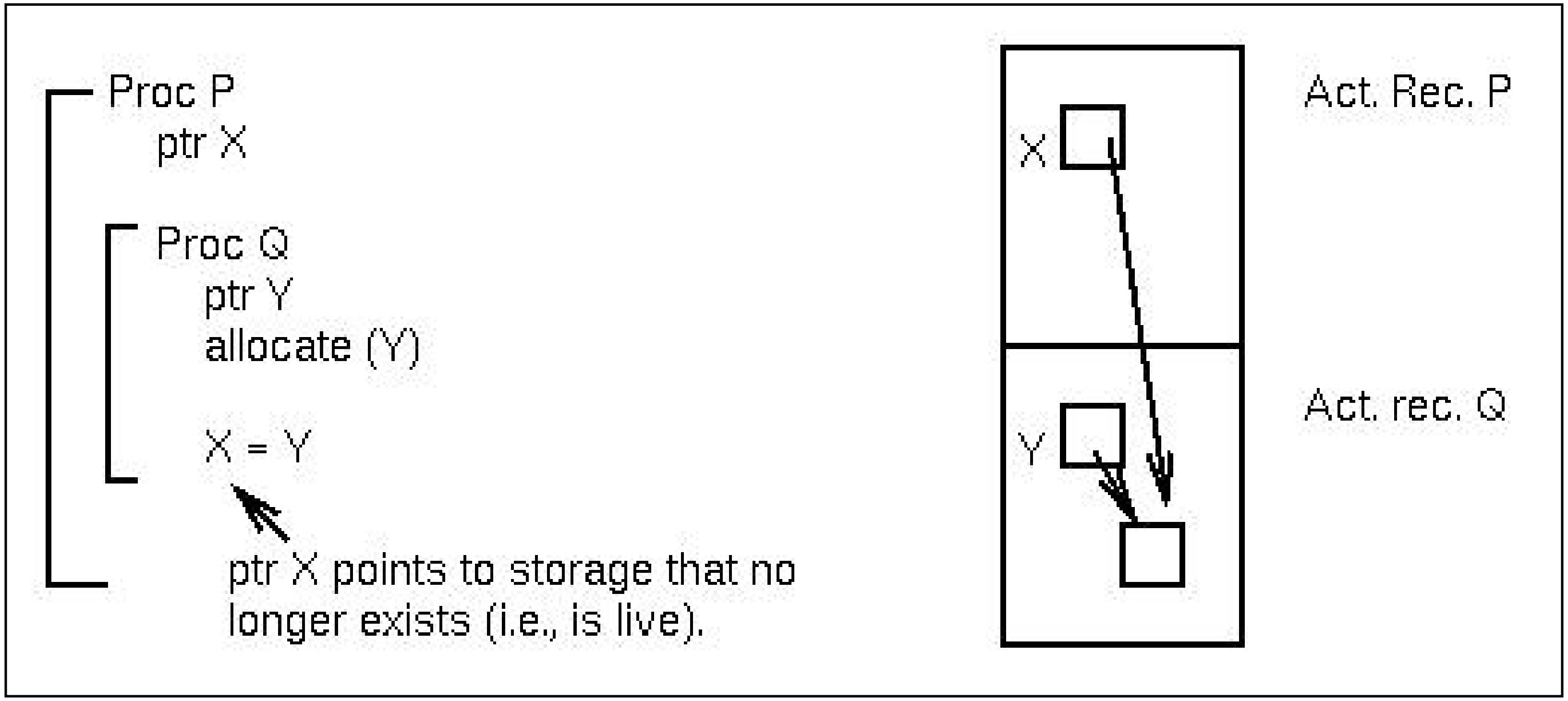
- **Roots:** values that a program can manipulate directly (i.e. values held in registers, on the program stack, and global variables.)
- **Node/Cell/Object:** an individually allocated piece of data in the heap.
- **Children Nodes:** the list of pointers that a given node contains.
- **Live Node:** a node whose address is held in a root or is the child of a live node.
- **Garbage:** nodes that are not live, but are not free either.
- **Garbage collection:** the task of recovering (freeing) garbage nodes.
- **Mutator:** The program running alongside the garbage collection system.

Heap Storage

- Memory allocation under explicit programmatic control
  - C malloc, C++ / Pascal / Java / C# new operation.
- Memory allocation implicit in language constructs
  - Lisp, Scheme, Haskell, SML, … most functional languages
  - Autoboxing/unboxing in Java 1.5 and C#
- Deallocation under explicit programmatic control
  - C, C++, Pascal
- Deallocation implicit
  - Java, C#, Lisp, Scheme, Haskell, SML, …

Stacks and dynamic allocations are incompatible

Why can't we just do dynamic allocation within the stack?



How does things become garbage?

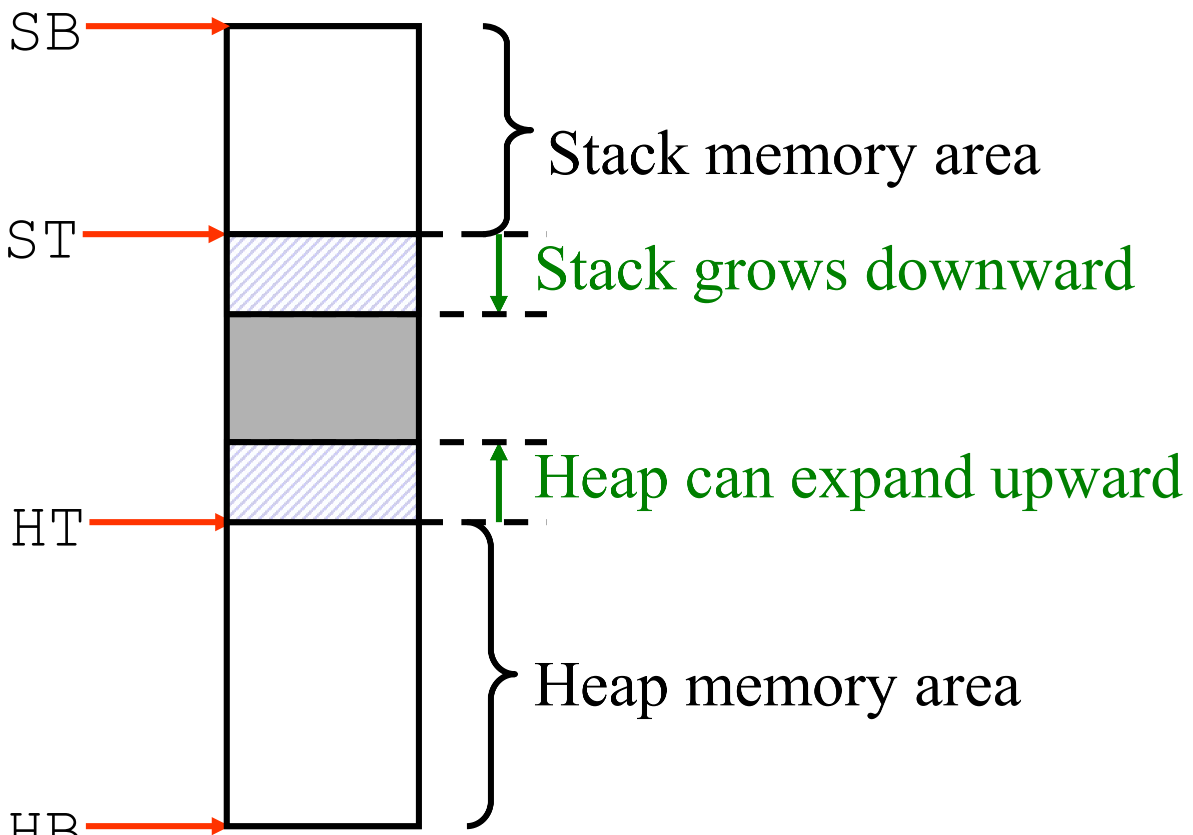
```
int *p, *q;
…
p = malloc(sizeof(int));
p = q;
```

Newly created space becomes garbage

```
for(int i=0;i<10000;i++){
    SomeClass obj= new SomeClass(i);
    System.out.println(obj);
}
```

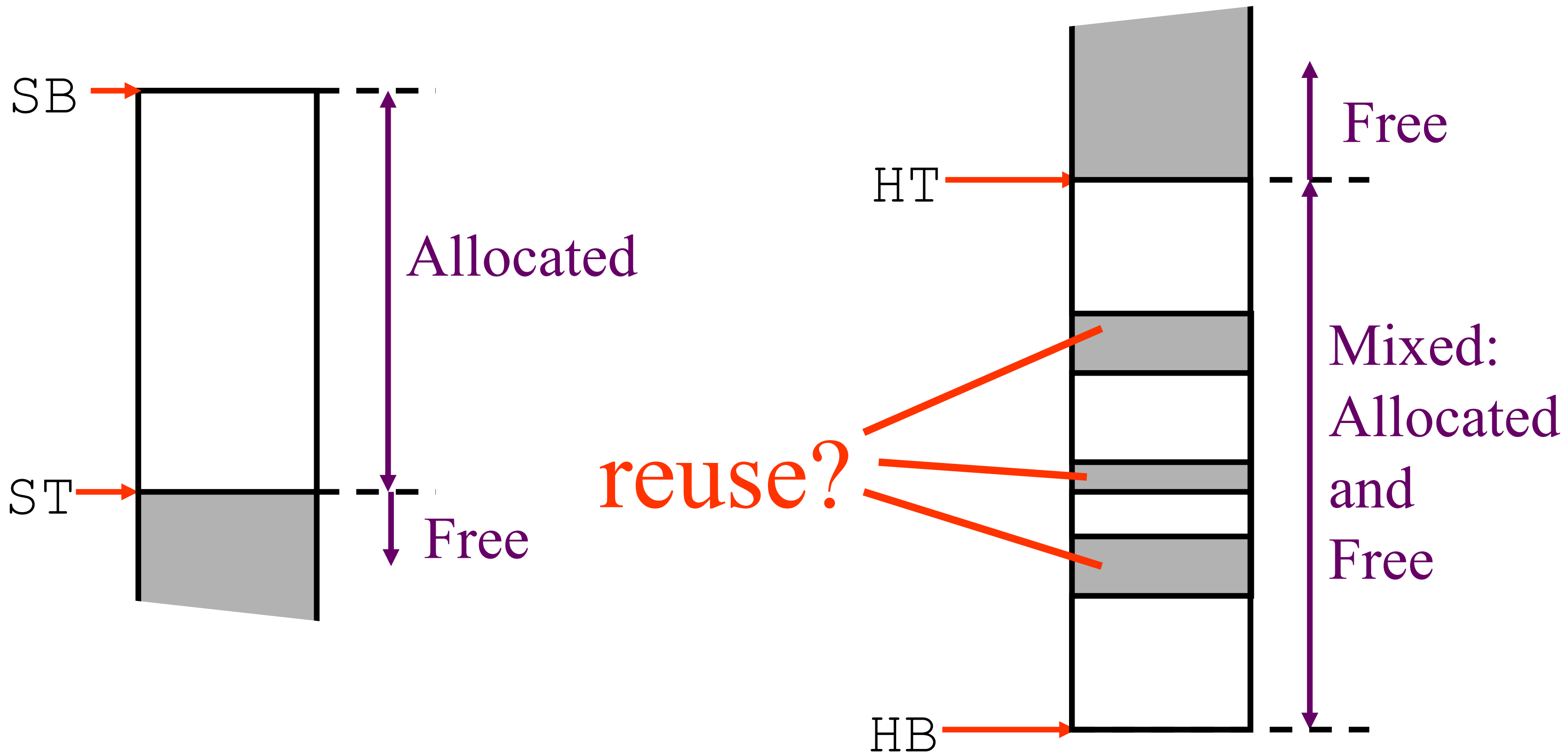
Creates 10000 objects, which becomes garbage just after the print

Where to put the heap?



How to keep track of free memory?

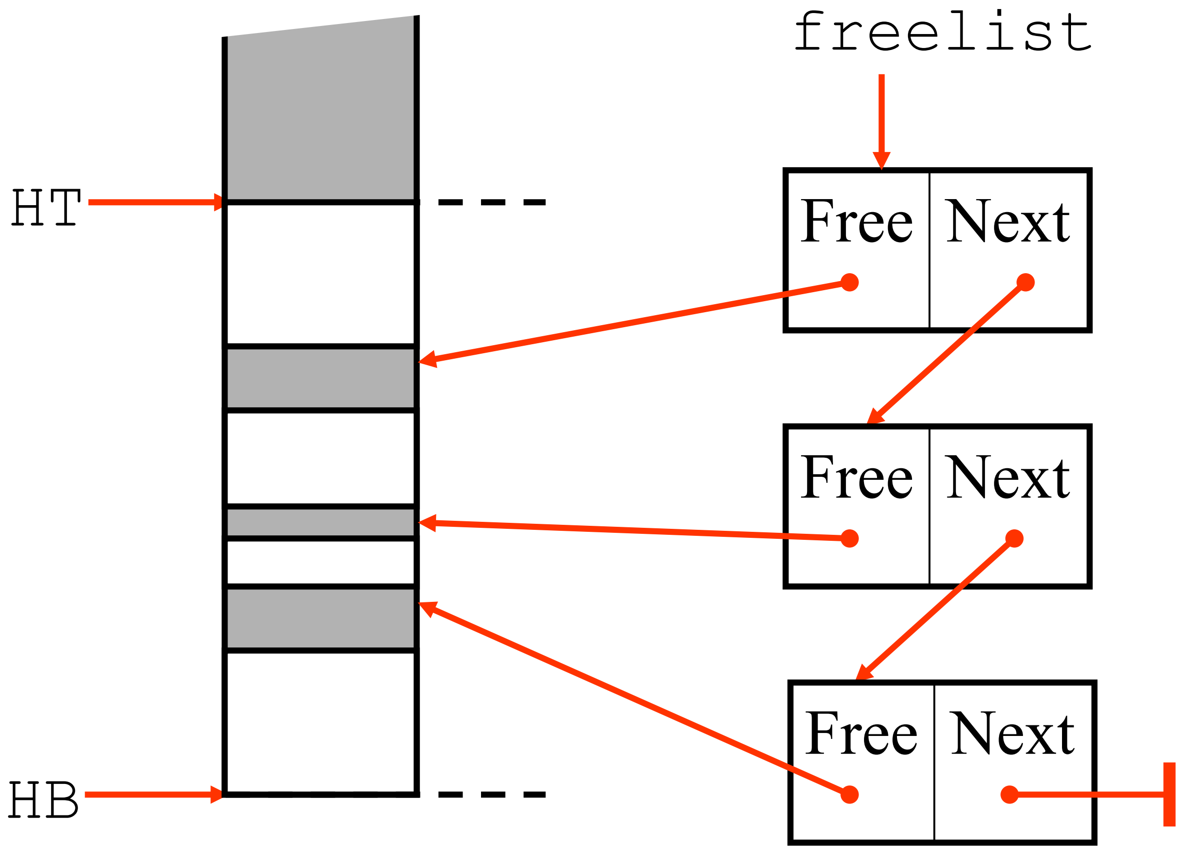
Stack is LIFO allocation => ST moves up/down everything above ST is in use/allocated. Below is free memory. This is easy! But …
Heap is not LIFO, how to manage free space in the "middle" of the heap?



How to keep track of free memory?

How to manage free space in the "middle" of the heap?

=> keep track of free blocks in a data structure: the "free list". For example we could use a linked list pointing to free blocks.



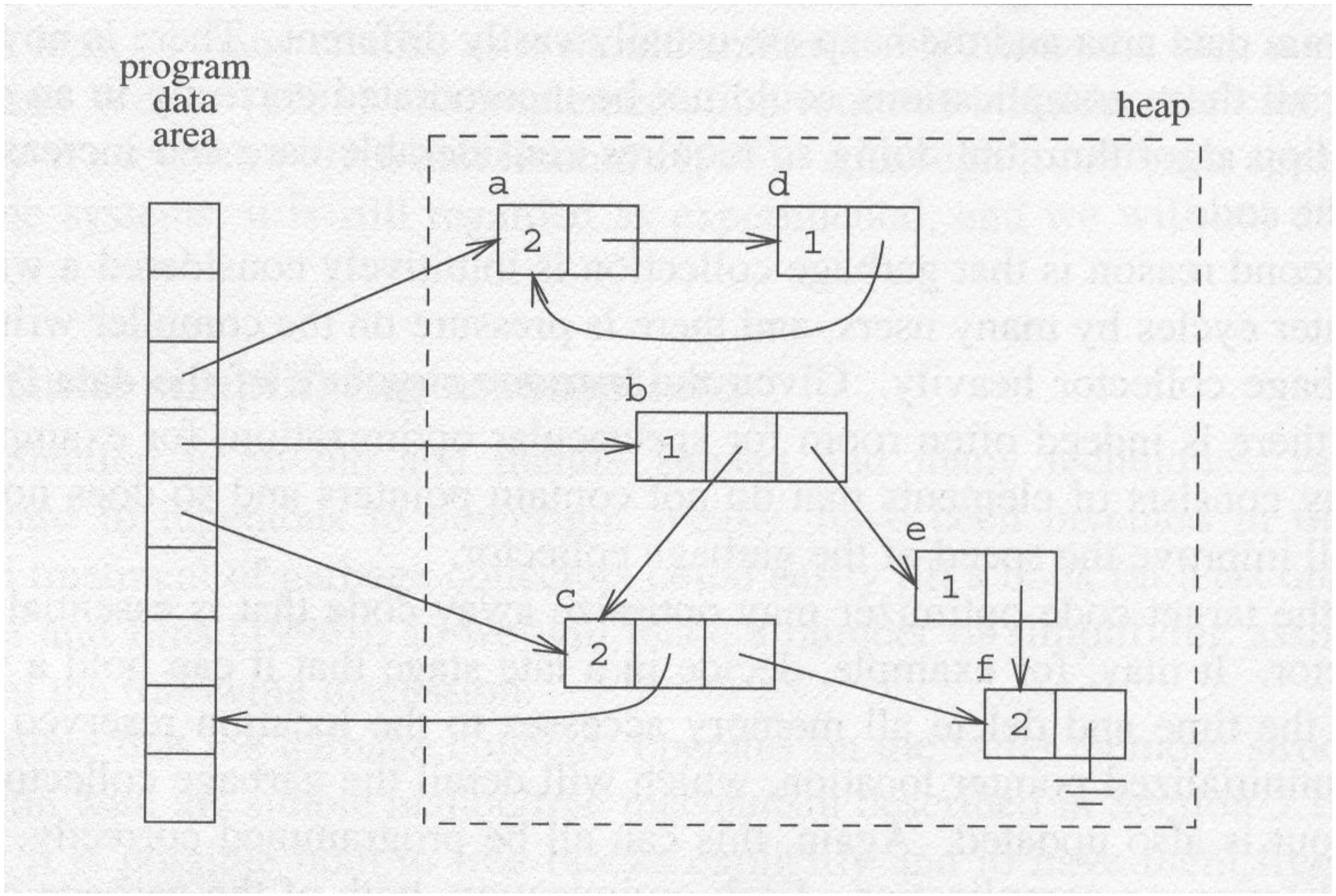
A freelist!
Good idea!

But where do we find the memory to store this data structure?

Types of garbage collectors

- The “Classic” algorithms
  - Reference counting
  - Mark and sweep
- Copying garbage collection
- Generational garbage collection

Reference Counting



Mark-Sweep

- The first tracing garbage collection algorithm
- Garbage cells are allowed to build up until heap space is exhausted (i.e. a user program requests a memory allocation, but there is insufficient free space on the heap to satisfy the request.)
- At this point, the mark-sweep algorithm is invoked, and garbage cells are returned to the free list.
- Performed in two phases:
  - **Mark:** identifies all live cells by setting a mark bit. Live cells are cells reachable from a root.
  - **Sweep:** returns garbage cells to the free list.
  - **Compaction**: we push live cells to one end of the heap
    - We can add a compaction phase as shown in Fig. 12.17.

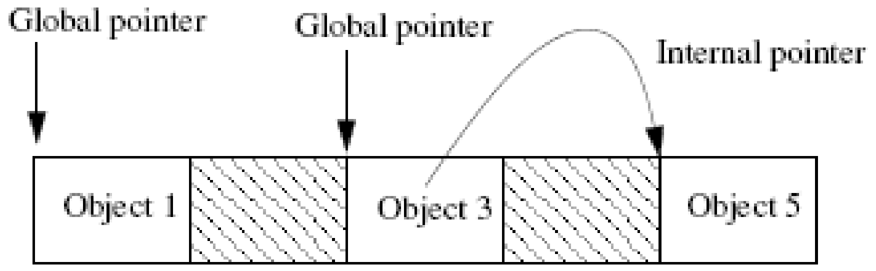


Figure 12.16: Mark-Sweep Garbage Collection

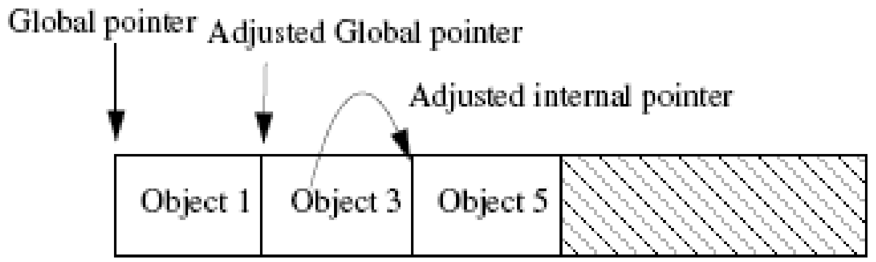


Figure 12.17: Mark-Sweep Garbage Collection with Compaction

Copying Garbage Collection

- Like mark-compact, copying garbage collection does not really "collect" garbage.
- Rather it moves all the live objects into one area and the rest of the heap is known to be available.
- Copying collectors integrate the traversal and the copying process, so that objects need only be traversed once.
- The work needed is proportional to the amount of live data (all of which must be copied).

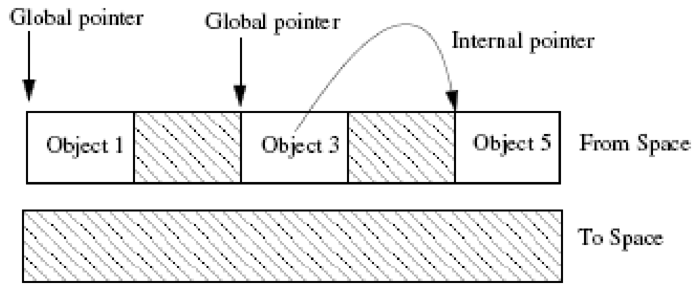


Figure 12.18: Copying Garbage Collection (a)

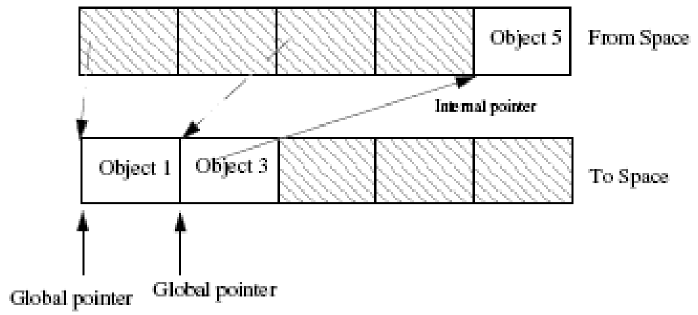


Figure 12.19: Copying Garbage Collection (b)

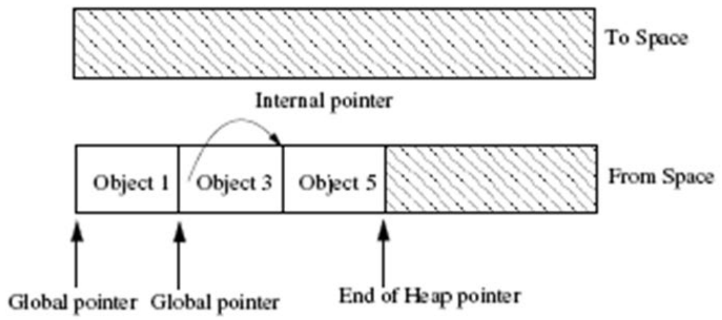


Figure 12.20: Copying Garbage Collection (c)

Generational Garbage Collection

- Attempts to address weaknesses of simple tracing collectors such as mark-sweep and copying collectors:
  - All active data must be marked or copied.
  - For copying collectors, each page of the heap is touched every two collection cycles, even though the user program is only using half the heap, leading to poor cache behavior and page faults.
  - Long-lived objects are handled inefficiently.
- Generational garbage collection is based on the *generational hypothesis*:

Most objects die young.

- As such, concentrate garbage collection efforts on objects likely to be garbage: young objects.