Memory management

Lecture 4

Formal Languages and Compilers 2011

Nataliia Bielova

Memory management

- Main operations that need the memory allocation:
  - segments of the user running code
  - constants and static user data
  - dynamic structures of user data
  - temporal values in the evaluation of the expressions
  - transmission of the parameters and return values
  - buffer I/O
- Operations that ask for dynamic allocation/reallocation of the memory:
  - call/return of subprograms
  - creation/destruction of data structures
  - inserting/removing components of the dynamic data structures
  - temporal values in the expressions and commands

Memory management (cont)

- Methods of management:
  - static
  - stack
  - heap

- Phases of the memory management:
  - allocation
  - release
    - explicit (dispose, free,…)
    - implicit (garbage collection)
  - compacting

Memory management (cont)

- The user should be responsible for memory management?
  - **YES** ->
    - precise knowledge about the needed memory for the data
    - efficiency
  - **NO** ->
    - possible loss of information
    - interference with the system
  - **Mix** of the two options

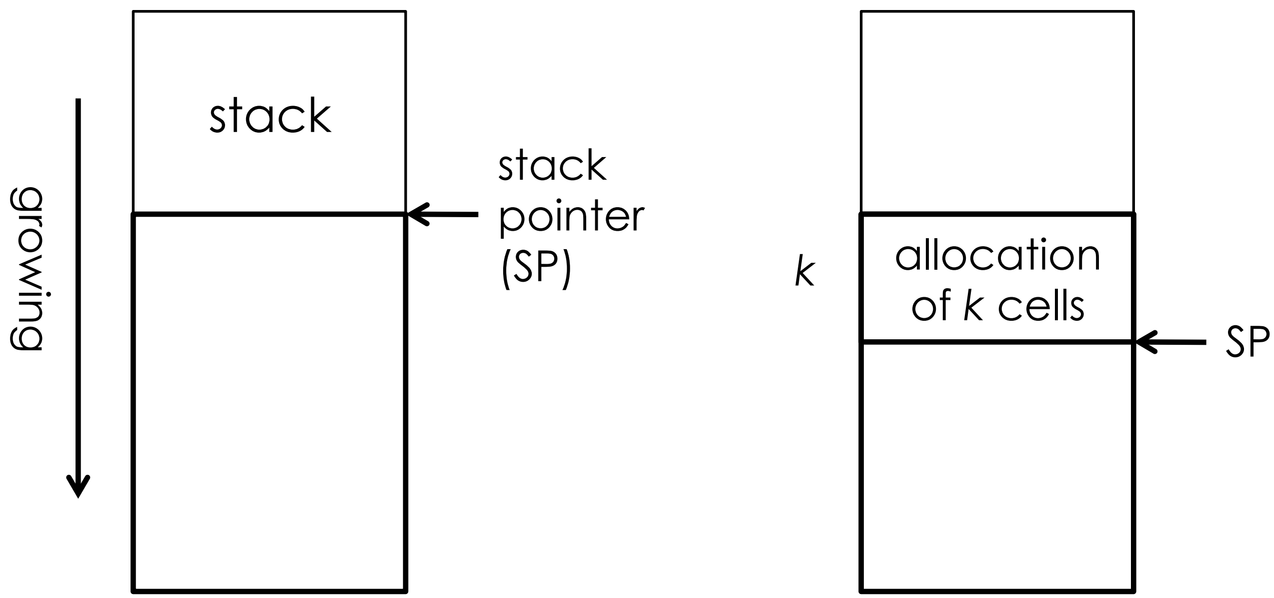
- Mechanisms for memory allocation:
  - declaration
  - explicit allocation using pointers
  - primitive operations that ask for memory (e.g. cons in LISP)

Static management

- Principal characteristics:
  - allocation at compile time
  - memory management is never done at run-time
  - no problem of the memory recovery
  - efficiency at run-time
  - impossibility of having recursion
  - impossibility of managing data structures that have non-fixed size (that are asking according to their process state or to some input)
- All the modern languages have some type of dynamic memory management

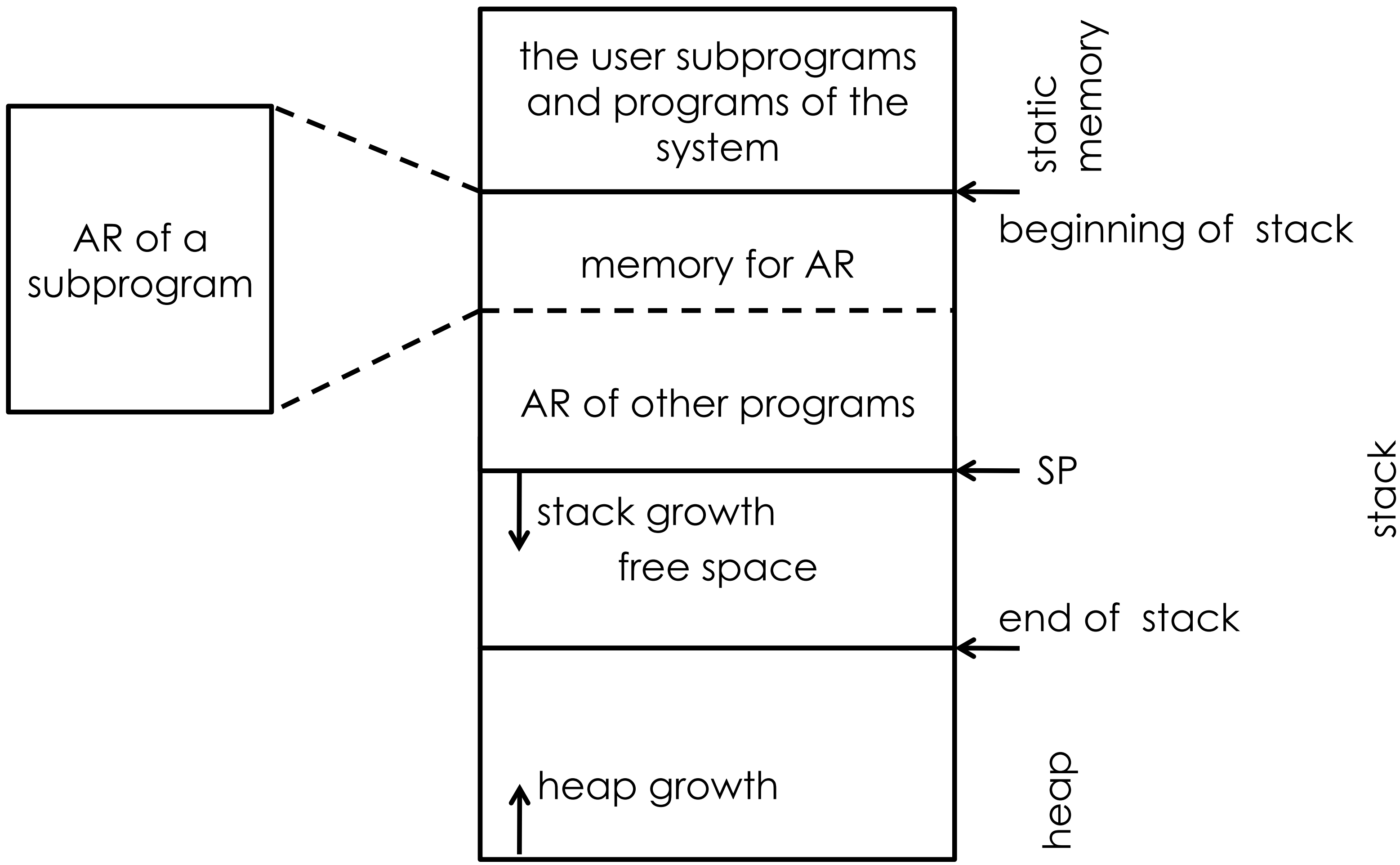
Stack

- It's the simplest form of the memory management at run-time.



- allocation and deallocation is a simple move of the SP
- technique that suits the last-in first-out calls of subprograms: applied to allocation/deallocation of the activation record

How memory is organized?

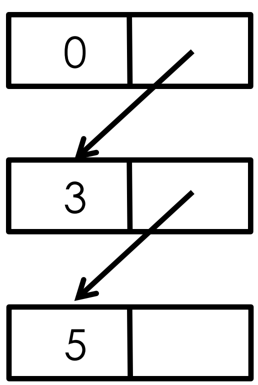


Heap

- Heap:
  - part of the memory for dynamic data types
  - “free list” management
  - is used for the operations:
    - malloc/free (C)
    - new/dispose (Pascal)
    - operations over the lists (Lisp/ML/OCaml)

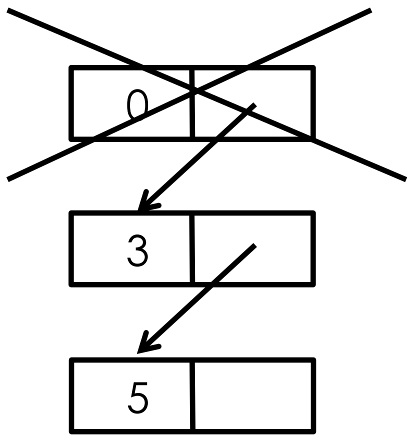
Heap management: examples

```
let f l = match l with
  | [] -> []
  | hd::tl -> if hd=0 then tl
                 else (hd+1)::tl
```



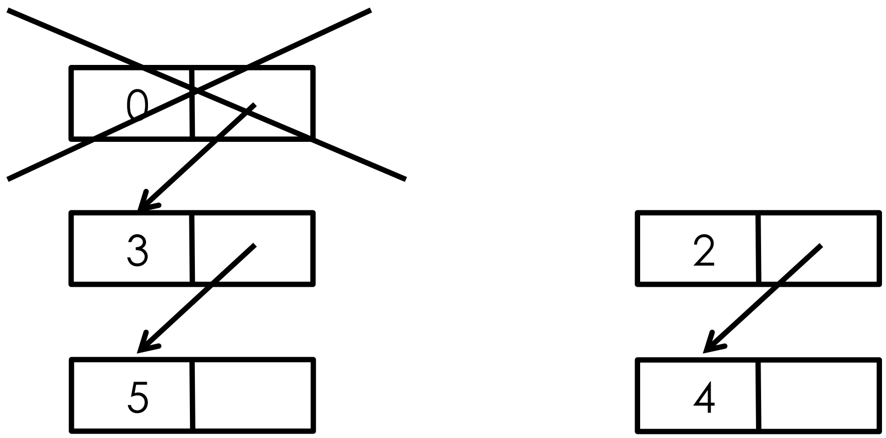
Heap management: examples

```
let f l = match l with
    | [] -> []
    | hd::tl -> if hd=0 then tl
                else (hd+1)::tl
```



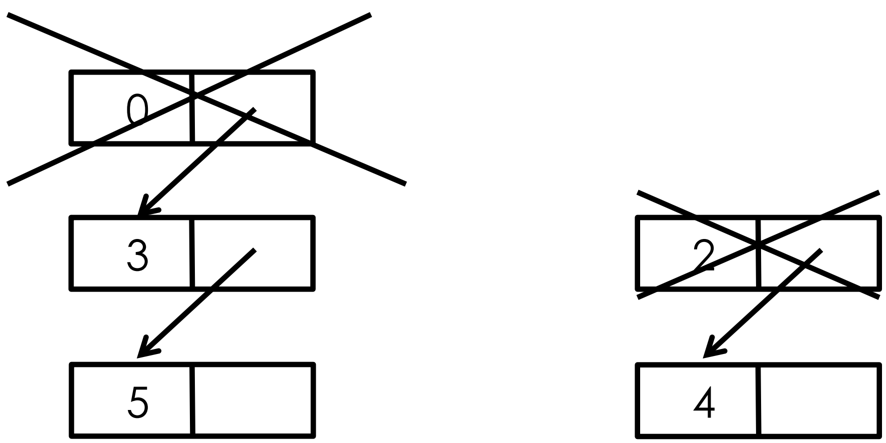
Heap management: examples

```
let f l = match l with
  | [] -> []
  | hd::tl -> if hd=0 then tl
              else (hd+1)::tl
```



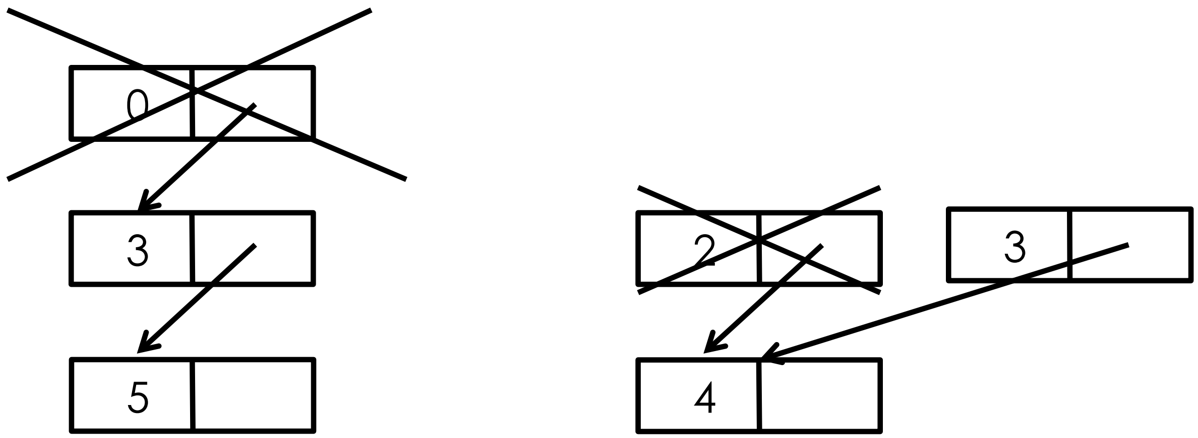
Heap management: examples

```
let f l = match l with
  | [] -> []
  | hd::tl -> if hd=0 then tl
              else (hd+1)::tl
```



Heap management: examples

```
let f l = match l with
    | [] -> []
    | hd::tl -> if hd=0 then tl
                else (hd+1)::tl
```



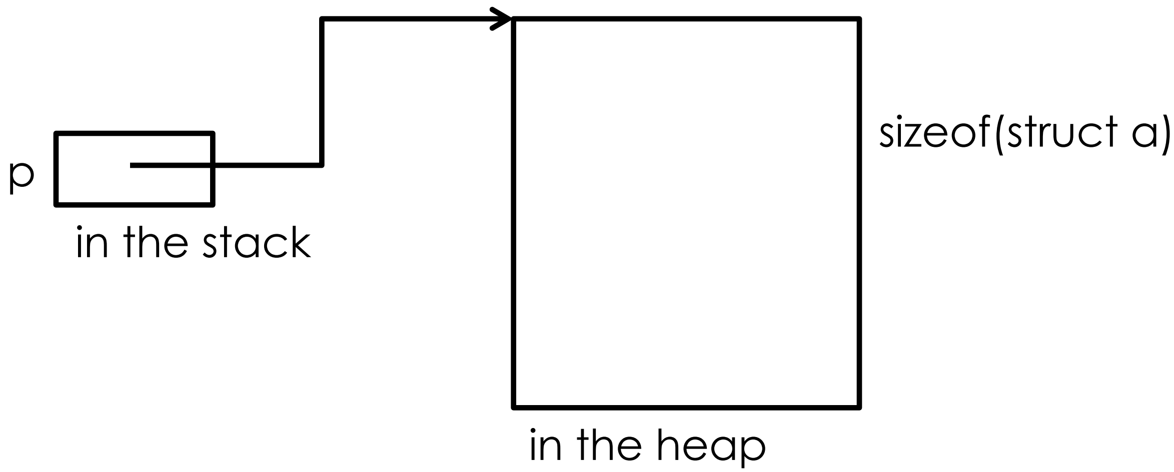
Heap management: examples

C:

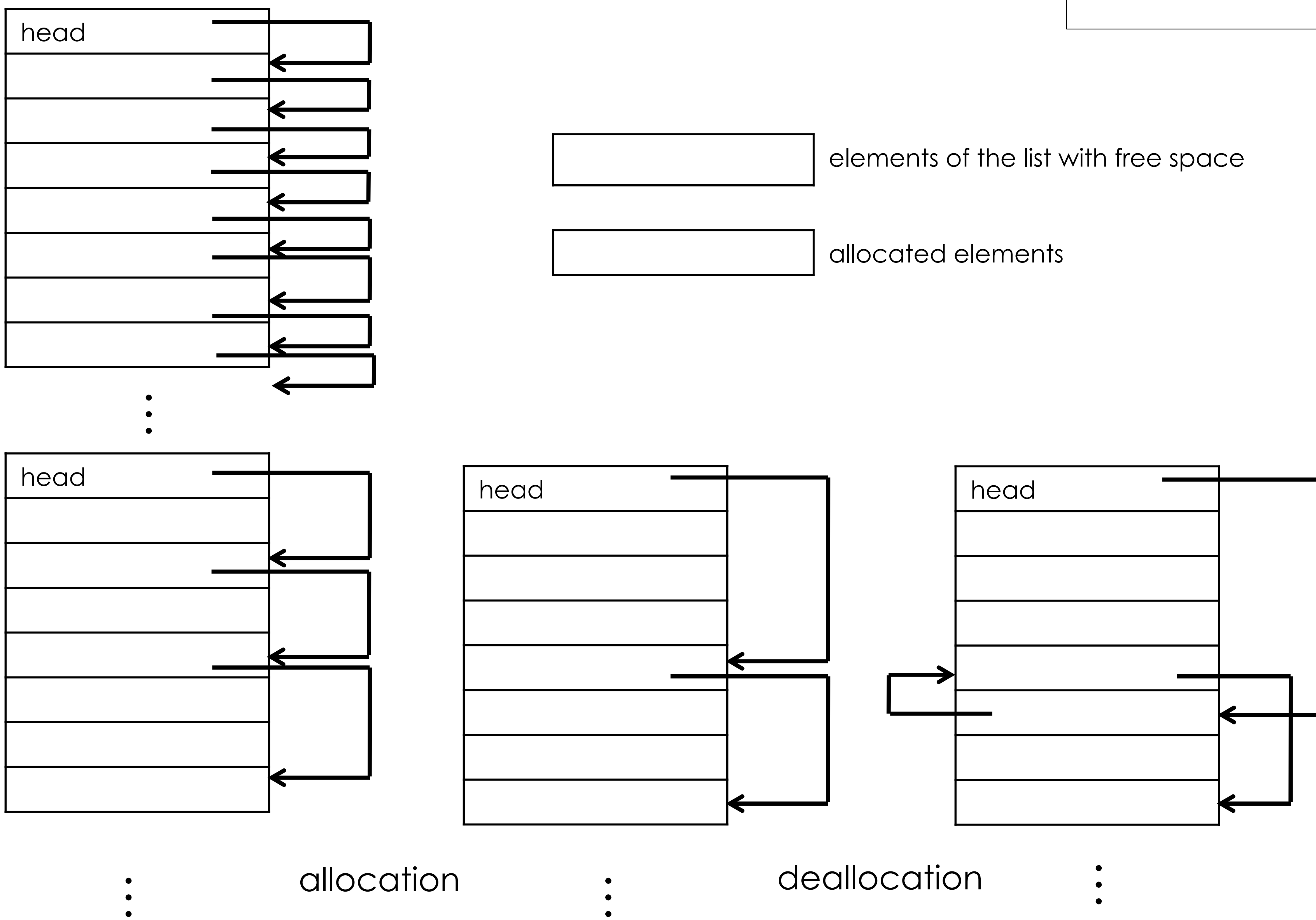
struct a { int x; int y; };

struct a *p;

p = (struct a*) malloc(sizeof (struct a));



Heap with elements of fixed length

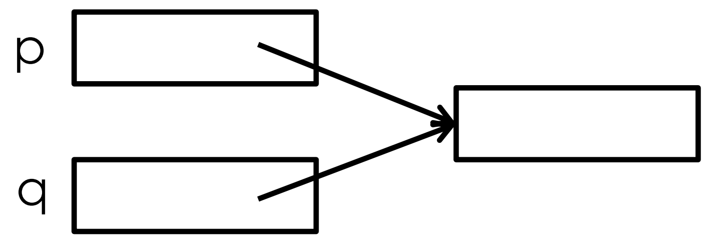


Memory release

When there are two pointers to one piece of memory in the heap:

p = &l;

q = &l;



Problems:

Garbage: the data exist (allocated) but all the pointers to it are destructed

Dangling References: pointers to access piece of memory continue to exist when the lifetime of the associated data is over

Memory release (cont.)

Example

int *p, *q;

…

p = (int *) malloc(sizeof(int));

- p = NULL   /* garbage: we risk to run out of dynamic memory*/
- q = p; free(p) /* dangling reference: serious errors in the execution*/

Solutions:

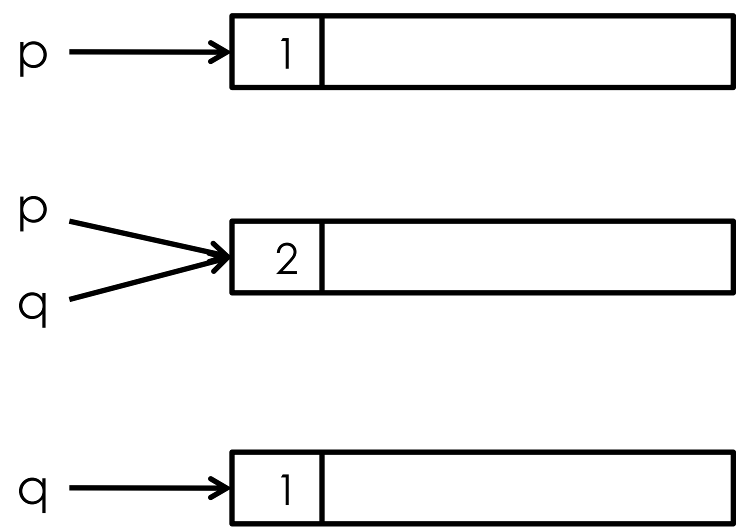
- **reference counter:**
  - explicit memory release
  - mechanism of counting the pointers
- **garbage collection:**
  - admitting garbage but not dangling reference
  - garbage collector (when we run out of heap)

Reference counter

element of the heap

K: number of pointers to an element => memory is released only if K=0

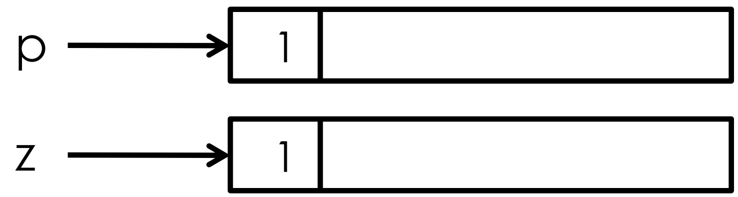
```
int *p, *q;
p = (int *) malloc(sizeof(int));

q = p;

free(p);
```



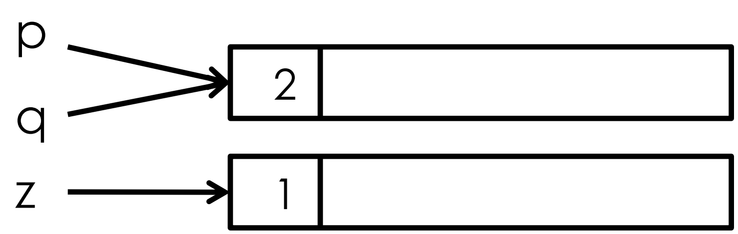
Reference counter (cont.)

Defect: simple operations (e.g. q = p) become much more expensive

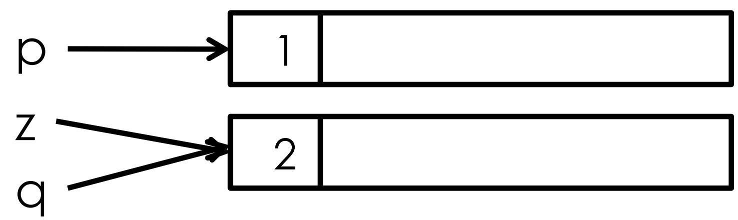
```
int *p, *q, *z;

p = (int *) malloc(sizeof(int));

z = (int *) malloc(sizeof(int));
```



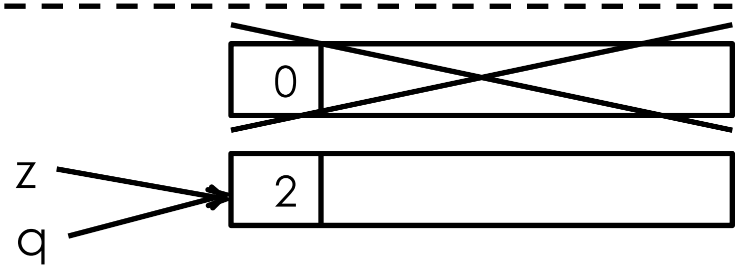

```
q = p;
```




```
q = z;
```




```
p  = NULL;
```



Reference counter (cont.)

- p = malloc(…)
  1. memory allocation and initiate counter with 1
  2. assignment of the allocated structure to p
- free(p) (or p = NULL)
  1. decrease the counter of the structure pointed by p
  2. recover memory in case the counter = 0
  3. delete p
- p = q
  1. decrease the counter of the structure pointed by p
  2. release the memory in case the counter = 0
  3. increase the counter of the structure pointed by q

Note: the access to the structure pointed by p is possible only if the counter is ≠ 0

Garbage collection

Idea:

- allows creation of garbage
  - avoids dangling references
  - doesn't have to manage the reference counter
- collects the garbage only when the memory is run out
  1. interruption of program computation
  2. control of "garbage collector"
  3. recovery of program computation

Note: garbage collection can be an expensive mechanism

Garbage collection (cont.)

- Garbage collection operates in two phases: mark and sweep
- Every element in the heap has a bit M for marking:
  - 0/OFF
  - 1/ON (initial marking)

An element is **active** when it is a part of the allocated structure.

1. **mark**: every active element is marked as OFF
2. **sweep**: all the elements ON are returned to the heap

Note:

- **sweep**: simple linear scanning of the heap
- **mark**: difficult!

Garbage collection (cont.)

- Showing the example of marking and sweeping.

Garbage collection: Active Elements

What does it mean that an element is **active**?

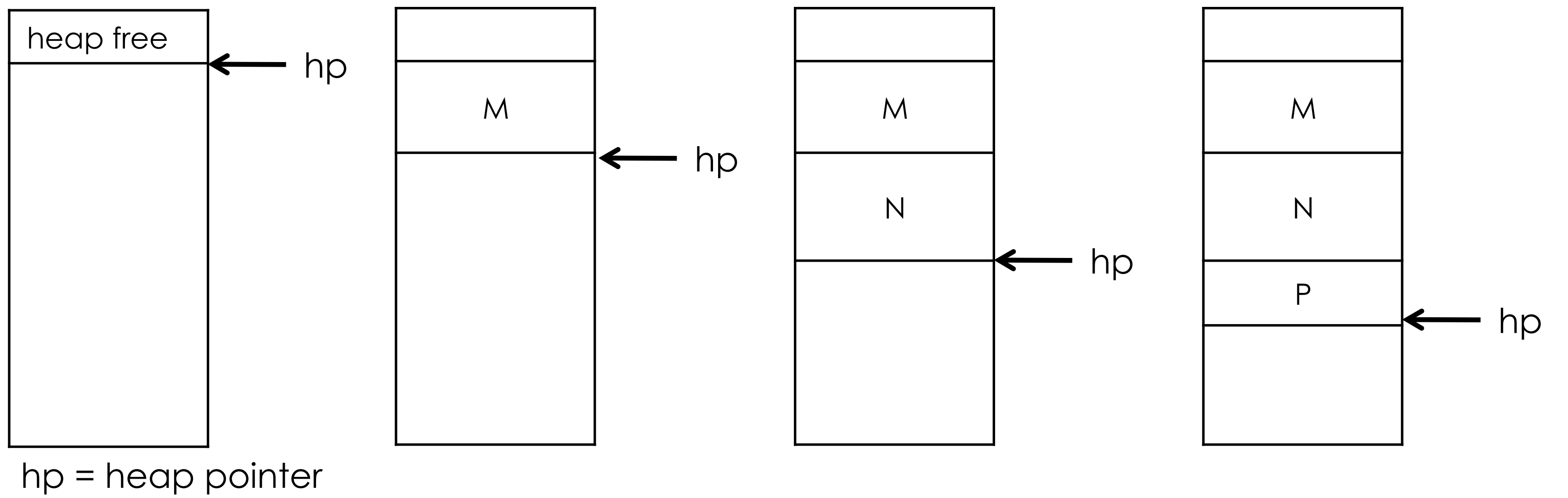
It is active if

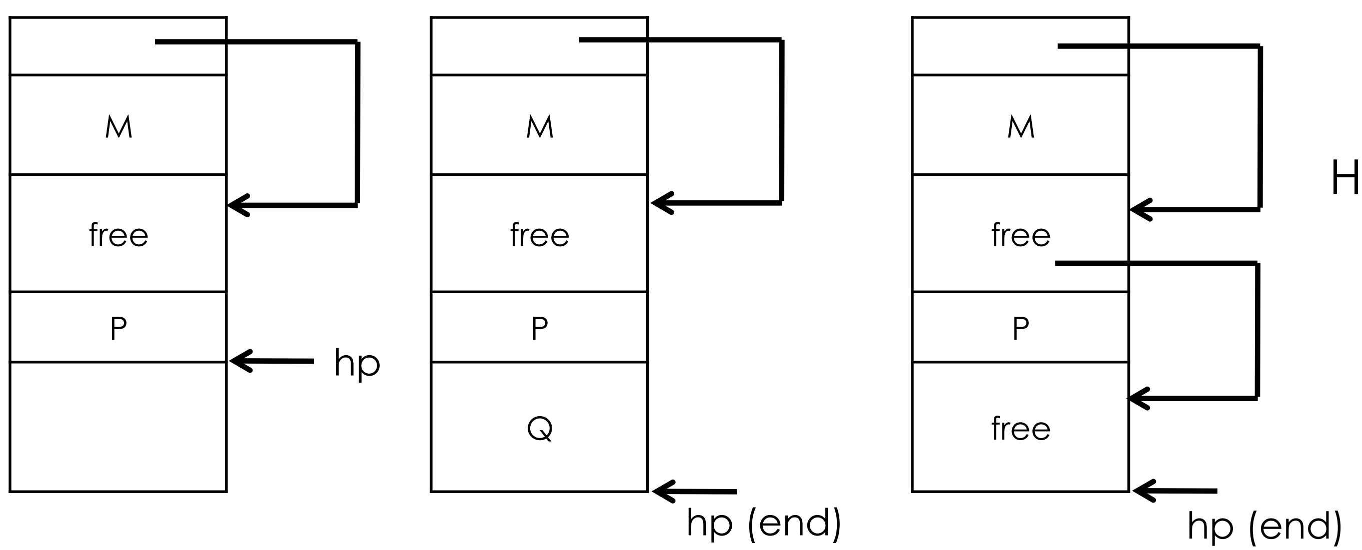
- it is pointed from an element outside of the heap
- It is pointed from an active element of the heap

To ensure that the release is possible the following conditions should be met:

- every active element should be accessible through the chain of pointers that start outside of the heap
- it should be possible to identify all the pointers outside of the heap that point to the elements of the heap
- it should be possible to identify in every active element of the heap whether it contains pointers to other elements of the heap

Heap with elements of non-fixed length





How to proceed?

Heap with elements of non-fixed length (cont.)

Two techniques:

1. using the list that is dynamically created:
   a) If block of length *m* is needed, it looks for the block B of the length *n* >= *m*
   b) cuts the block B (if *n*>*m*)

Two ways of choosing the block B:

- first fit - first in the free memory list that fits.
- best fit - the smallest free block in the list that fits.

⇒problem of fragmentation

2. compaction technique
   - all the free space is compacted in one block and moved in front of the heap, with the corresponding update of the pointers in the heap