Formal Languages and Compilers - Exercises
Lecture 4
Memory Management

28/03/2012

Outline

Memory management overview - 1

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

Memory management overview - 2

Operations that ask for dynamic allocation/reallocation of 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 overview - 3

Methods of management

- Static
- Heap
- Stack

Phases of memory management

- Allocation
- Compacting
- Release
  - Explicit (`dispose`, `free`)
  - Implicit (garbage collection)

Memory management overview - 3

Methods of management

- Static
- Heap
- Stack

Phases of memory management

- Allocation
- Compacting
- Release
  - Explicit (`dispose`, `free`)
  - Implicit (garbage collection)

Memory management overview - 4

Should the user be responsible for memory management?

Yes:
- Precise knowledge about the memory needed
- 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

Memory management overview - 4

Should the user be responsible for memory management?

Yes:
- Precise knowledge about the memory needed
- 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

Memory management overview - 4

Should the user be responsible for memory management?

Yes:
- Precise knowledge about the memory needed
- 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

Outline

Static management

Good things

- Allocation at compile time
- Memory management is never done at run-time
- No problem of the memory recovery
- Efficiency at run-time

Bad things

- 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

Static management

Good things

- Allocation at compile time
- Memory management is never done at run-time
- No problem of the memory recovery
- Efficiency at run-time

Bad things

- 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

Static management

Good things

- Allocation at compile time
- Memory management is never done at run-time
- No problem of the memory recovery
- Efficiency at run-time

Bad things

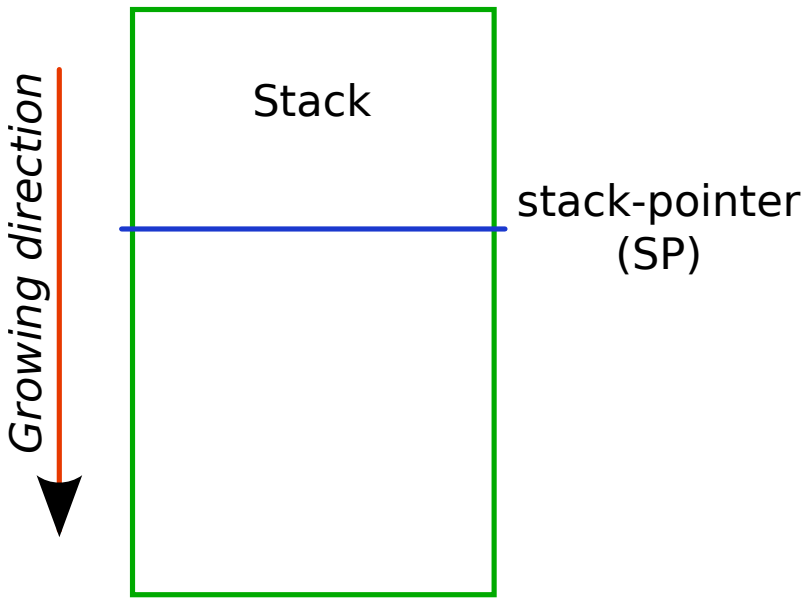
- Impossibility of having recursion
- Impossibility of managing data structures that have non-fixed size (that are asking according to their process state or to some input)

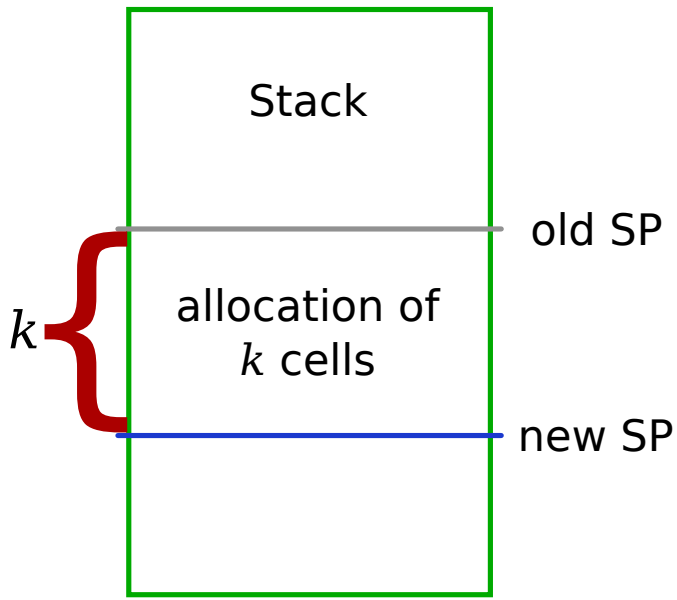
All the modern languages have some type of dynamic memory management

Outline

Stack

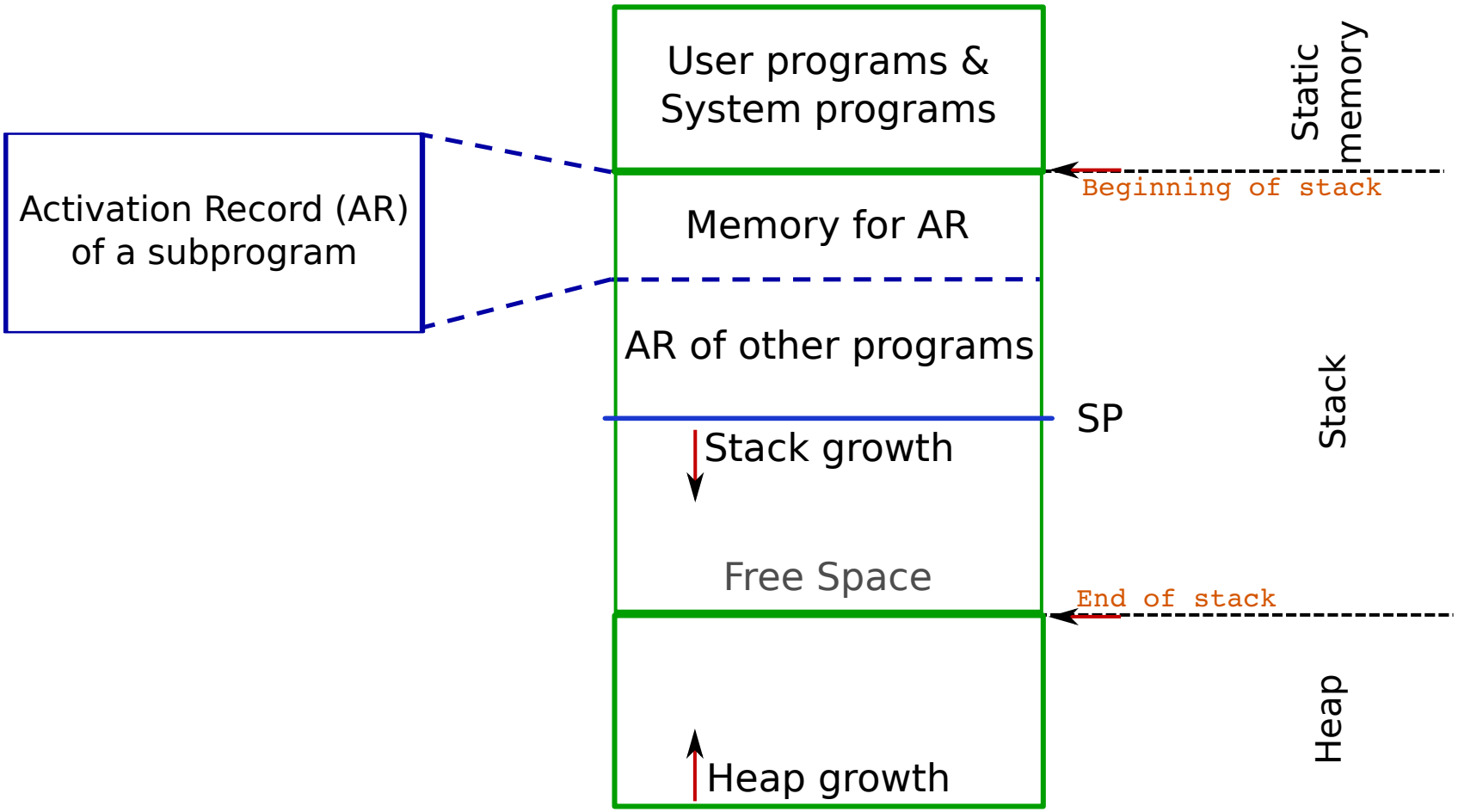
- It s the simplest form of the memory management at run-time





- Allocation and deallocation are simple moves of the SP
- Technique that suits the LIFO calls of subprograms: applied to allocation/deallocation of the activation record

How is memory organized?



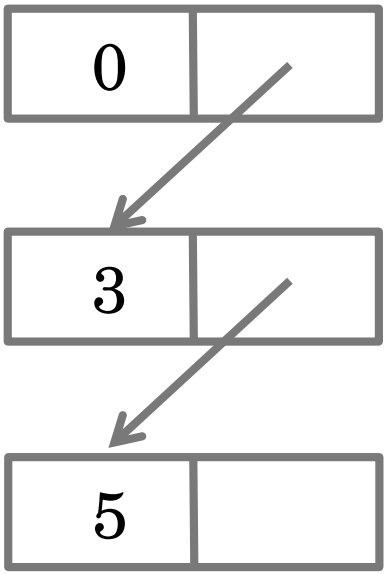
Outline

Heap

- Part of the memory for dynamic data types
- Free list management
- Is used for the operations `malloc/free` (in C) or `new/dispose` (in Pascal)
- Operations over the lists (Lisp/ML/O'CaML)

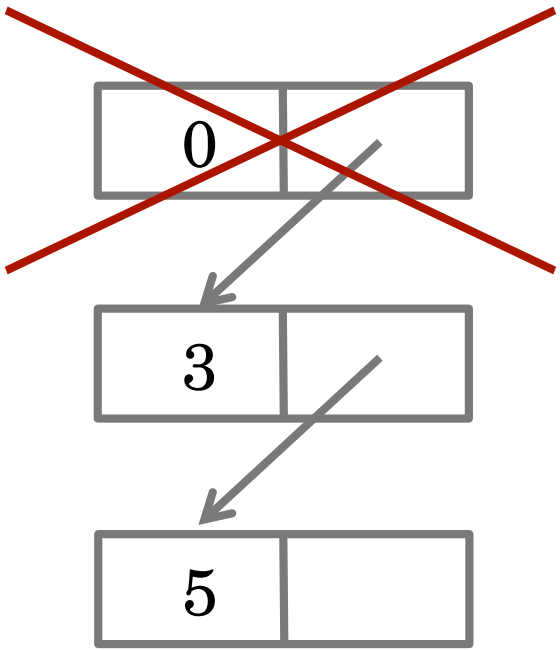
Heap management: examples

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



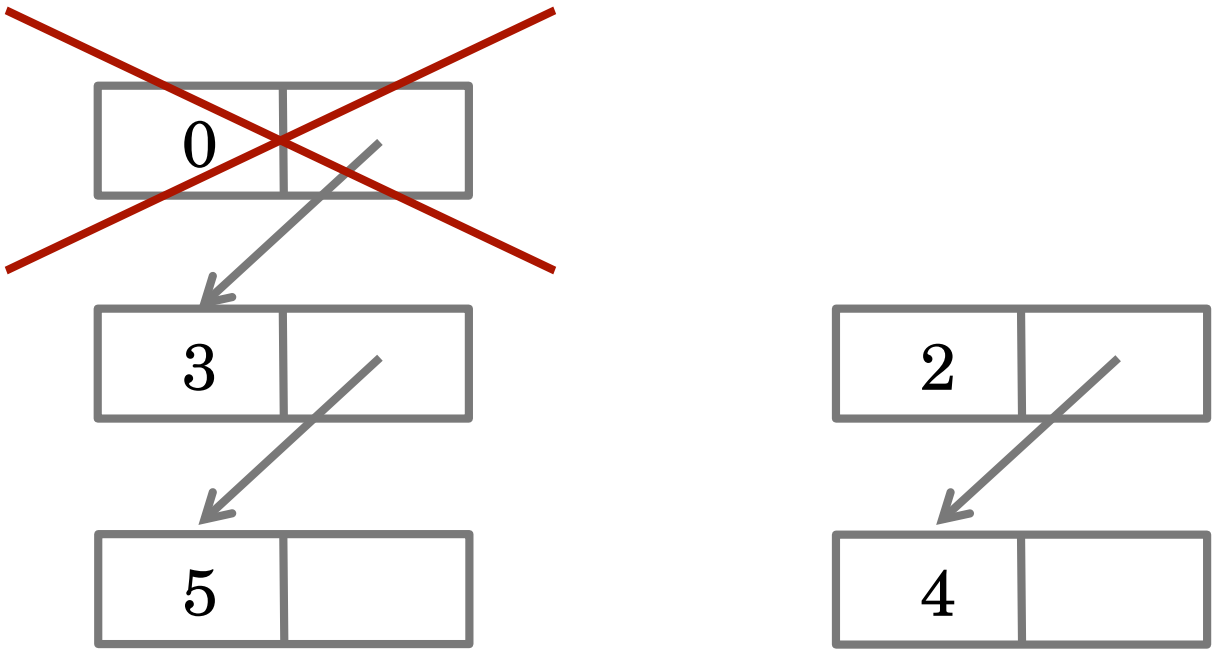
Heap management: examples

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



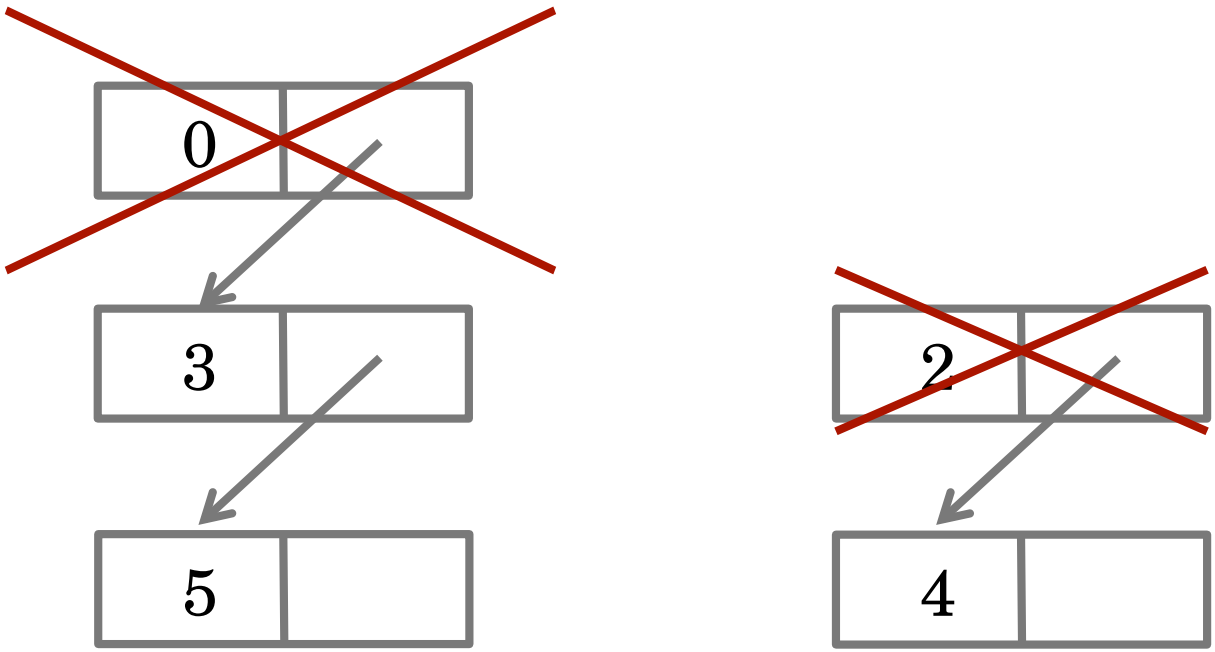
Heap management: examples

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



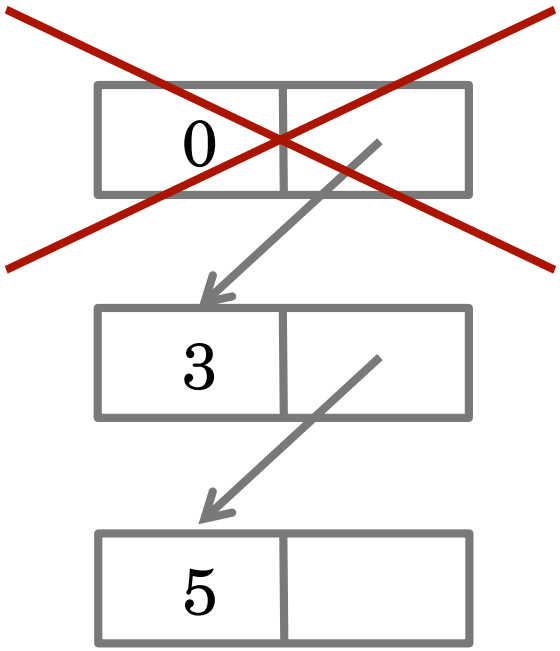
Heap management: examples

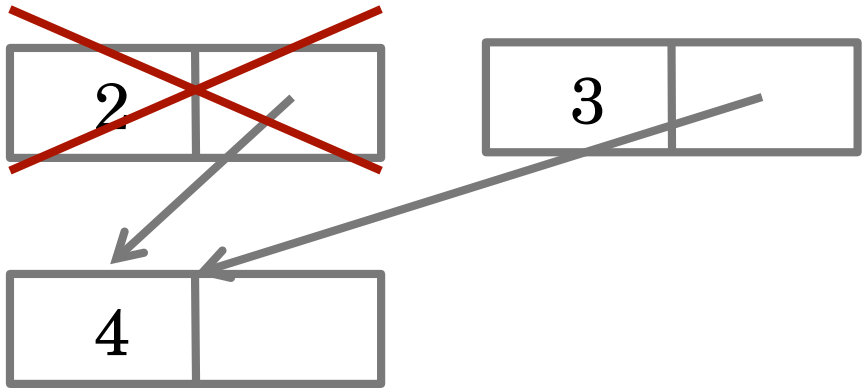
```
let f l = match l with
    [] -> []
  | hd::tl -> if hd=0 then tl else (hd+1)::tl
```



Heap management: examples

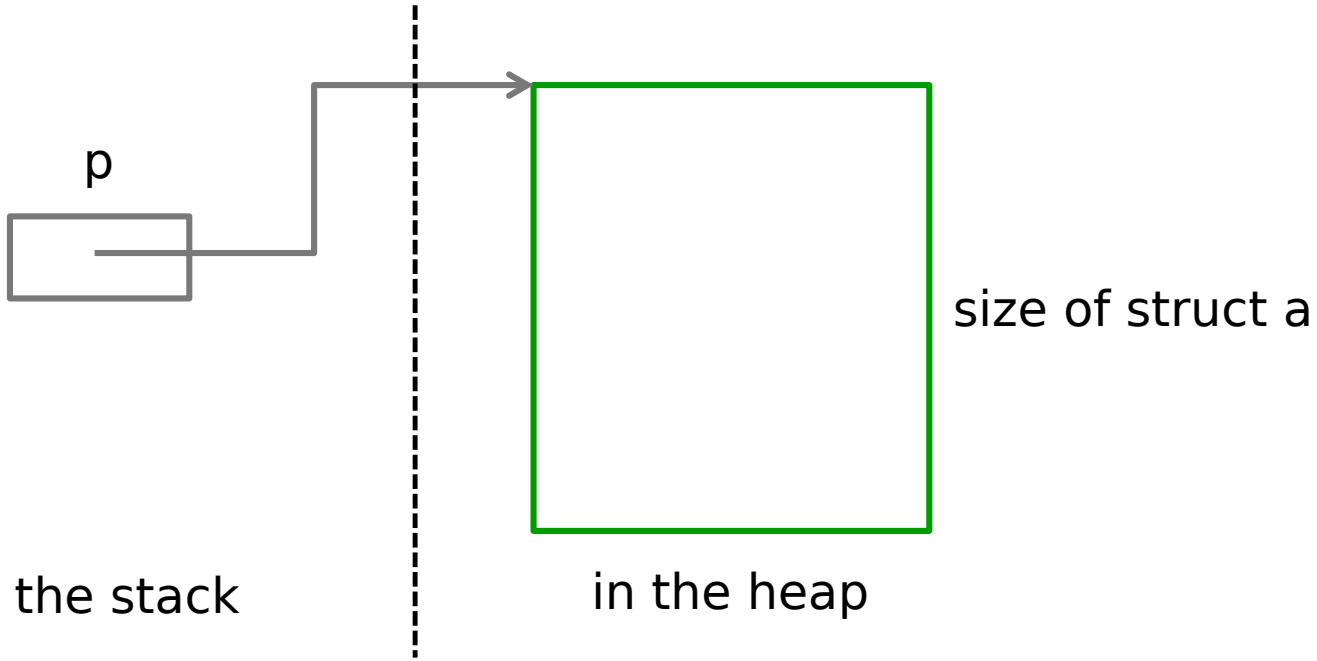
```
let f l = match l with
    [] -> []
  | hd::tl -> if hd=0 then tl else (hd+1)::tl
```



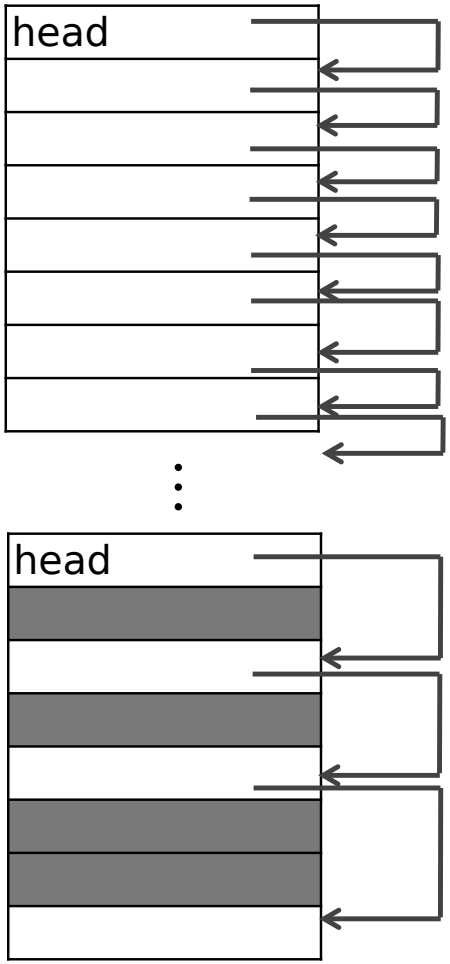


Heap management: examples - 2

```
struct a { int x; int y; };
struct a *p;
p = (struct a*) malloc(sizeof (struct a));
```



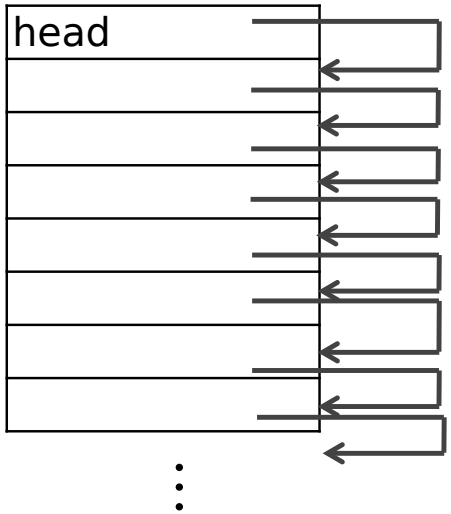
Heap with elements of fixed length

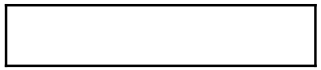


Elements of the list with free space

Allocated elements

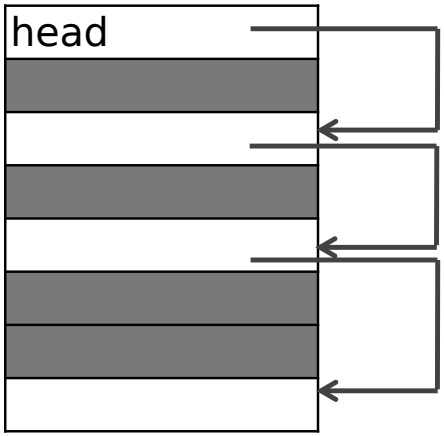
Heap with elements of fixed length

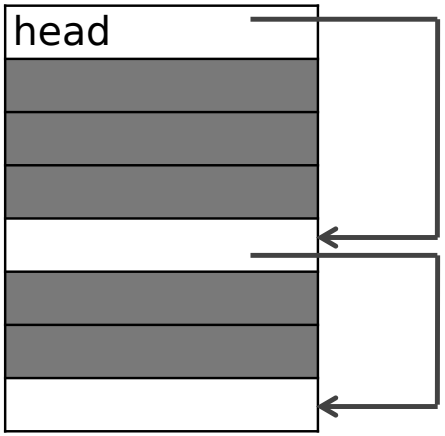




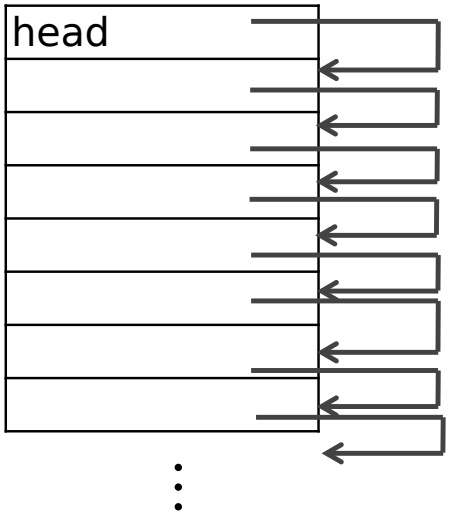
Elements of the list with free space

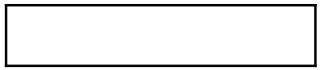
Allocated elements





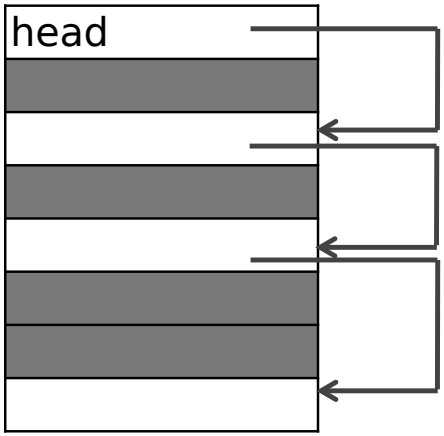
Heap with elements of fixed length

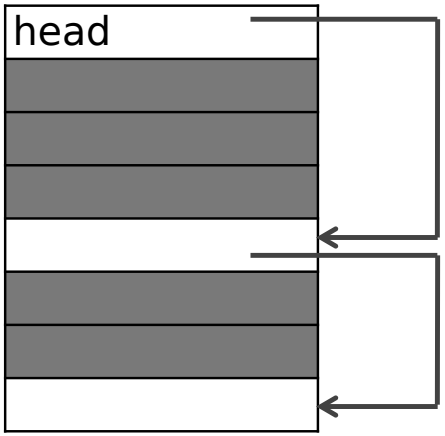


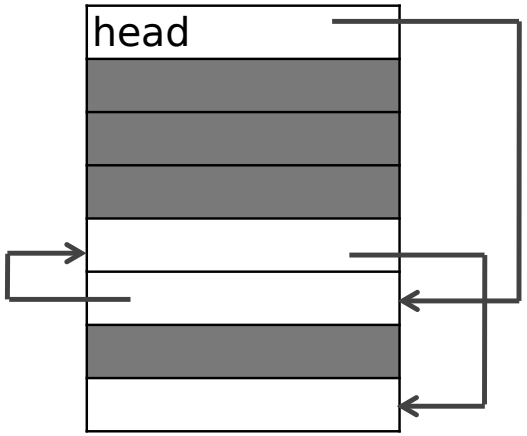


Elements of the list with free space

Allocated elements





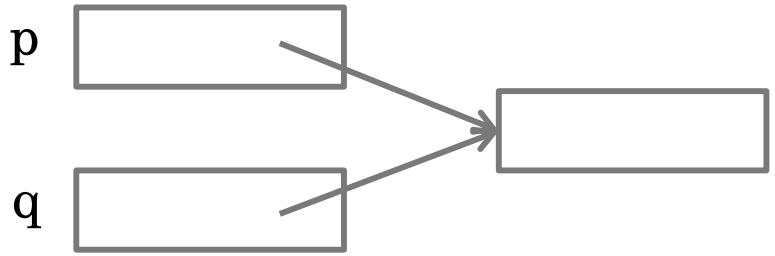


Outline

Memory release - 1

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

```
p = &i;
q = &i;
```



Problems

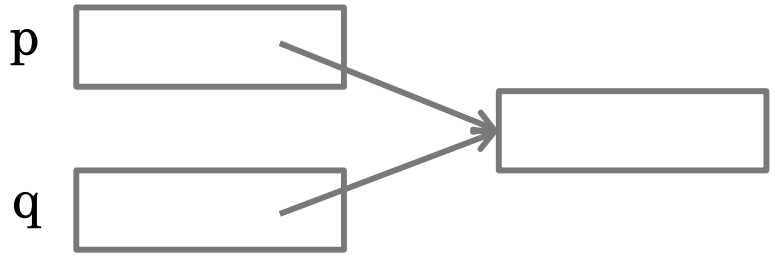
Garbage: the data exist (allocated) but all the pointers to it are destructed

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

Memory release - 1

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

```
p = &i;
q = &i;
```



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 - 2

```
int *p;
p = (int *) malloc(sizeof(int));
p = NULL  /* garbage */

int *p, *q;
p = (int *) malloc(sizeof(int));
free(p); /* p still equal previous address */
q = p; /* Dangling reference */
```

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)

Memory release - 2

```
int *p;
p = (int *) malloc(sizeof(int));
p = NULL  /* garbage */

int *p, *q;
p = (int *) malloc(sizeof(int));
free(p); /* p still equal previous address */
q = p; /* Dangling reference */
```

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)

Memory release - 2

```
int *p;
p = (int *) malloc(sizeof(int));
p = NULL  /* garbage */

int *p, *q;
p = (int *) malloc(sizeof(int));
free(p); /* p still equal previous address */
q = p; /* Dangling reference */
```

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)

Outline

Reference counter - 1

Element of the heap

K is the number of pointers to the element; memory is released only if $K = 0$

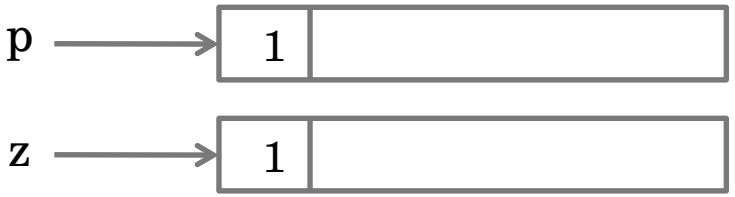
```
int *p, *q;
p = malloc(sizeof(int));
```


```
q = p;
```


```
free(p);
```

Reference counter - 2

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



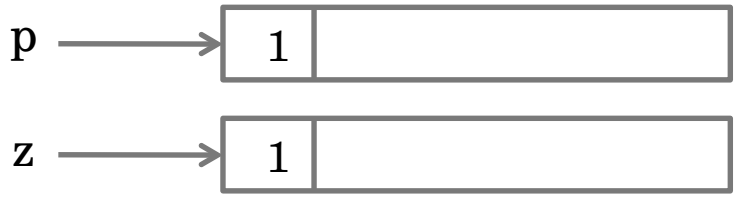
Reference counter - 2

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

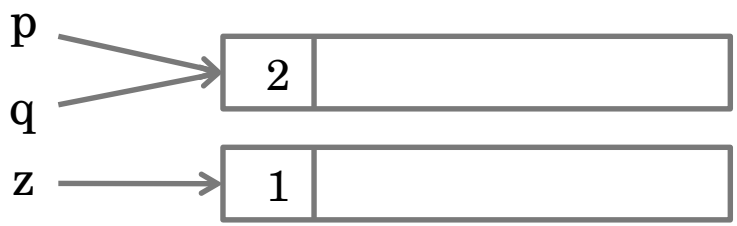
```
q = p;
```

Reference counter - 2

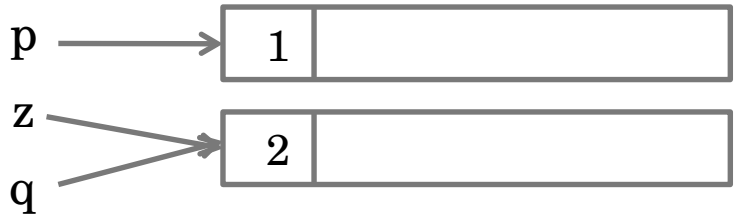
```
int *p, *q, *z;
p = (int *) malloc (sizeof(int));
z = (int *) malloc (sizeof(int));
```




```
q = p;
```

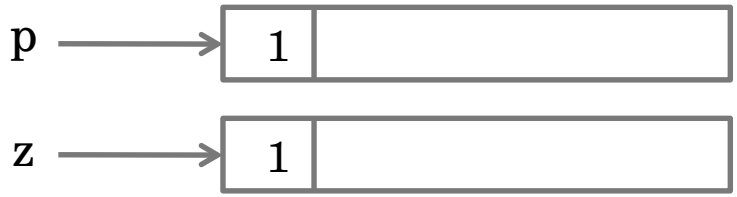



```
q = z;
```

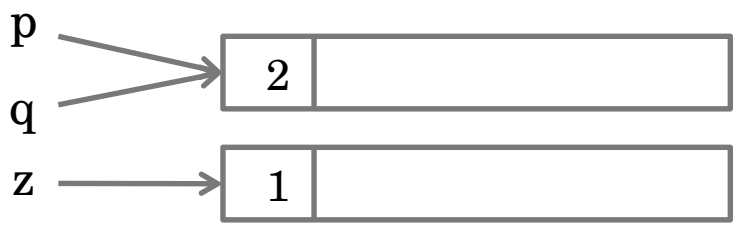


Reference counter - 2

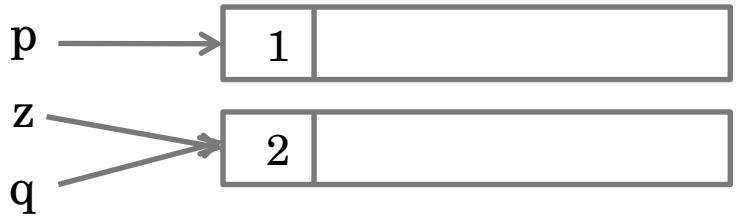
```
int *p, *q, *z;
p = (int *) malloc (sizeof(int));
z = (int *) malloc (sizeof(int));
```



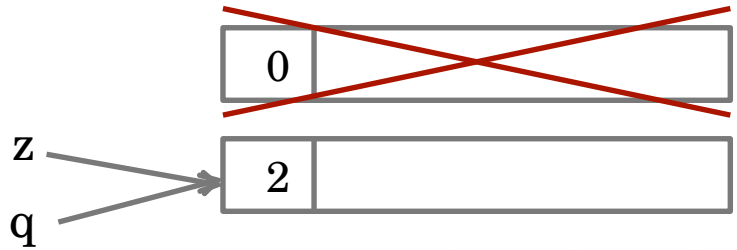

```
q = p;
```




```
q = z;
```




```
p = NULL;
```



Reference counter - 3

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

- 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 $\neq 0$

Reference counter - 3

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

- 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 $\neq 0$

Reference counter - 3

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

- 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 $\neq 0$

Reference counter - 3

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

- 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 $\neq 0$

Reference counter - 3

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

- 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 $\neq 0$

Outline

Garbage collection - 1

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

Phases of GC

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 - 1

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

Phases of GC

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 - 1

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

Phases of GC

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 - 2

- 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.

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

Note

sweep: simple linear scanning of the heap

mark: difficult!

Garbage collection: Active Elements

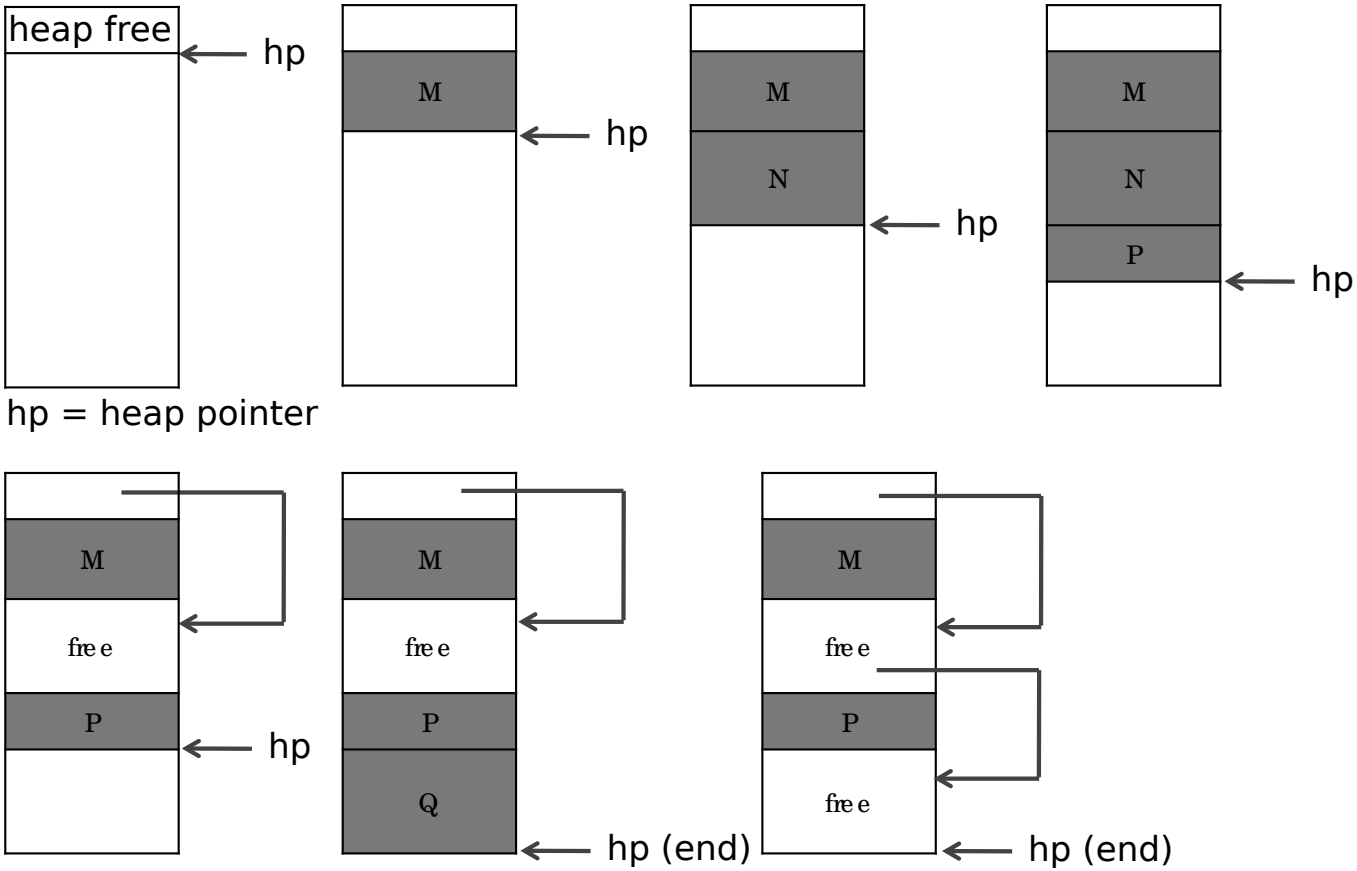
What does it mean that an element is active?

- 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 - 1



How to proceed?

Heap with elements of non-fixed length - 2

Using the list that is dynamically created

1. If a block of length `m` is needed, it looks for the block B of the length $n \geq m$
2. 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

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

Heap with elements of non-fixed length - 2

Using the list that is dynamically created

1. If a block of length `m` is needed, it looks for the block B of the length $n \geq m$
2. 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

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