Imperativ och objektorienterad programmeringsmetodik

Föreläsning 5 av många

Tobias Wrigstad

Lite om länkade strukturer — notation, insättning, frigöra…

De två viktigaste minnesareorna

Stacken

Håller funktionernas minne, dvs. variabler och parametrar

Hanteras automatiskt av C

LIFO-ordning, vår ”tallriksmodell”

Heapen

Allt minne som allokeras dynamiskt mha malloc, calloc och realloc — och frigörs med free

Måste hanteras manuellt

En 1-dimensionell array, vårt ”rutade papper”

```
int foo(int *x, int y)
{
  int *z = malloc(42);
  ...
  free(z);
  ...
  return -7;
}
```

De två viktigaste minnesareorna

Stacken

Håller funktionernas minne, dvs. variabler och parametrar

Hanteras automatiskt av C

LIFO-ordning, vår ”tallriksmodell”

Heapen

Allt minne som allokeras dynamiskt mha malloc, calloc och realloc — och frigörs med free

Måste hanteras manuellt

En 1-dimensionell array, vårt ”rutade papper”

Variablerna x, y och z ligger på stacken

```
int foo(int *x, int y)
{
  int *z = malloc(42);
  ...
  free(z);
  ...
  return -7;
}
```

De två viktigaste minnesareorna

Stacken

Håller funktionernas minne, dvs. variabler och parametrar

Hanteras automatiskt av C

LIFO-ordning, vår ”tallriksmodell”

Heapen

Allt minne som allokeras dynamiskt mha malloc, calloc och realloc — och frigörs med free

Måste hanteras manuellt

En 1-dimensionell array, vårt ”rutade papper”

Variabeln z pekar på ett minnesutrymme allokerat på heapen

```
int foo(int *x, int y)
{
  int z = malloc(42);
  ...
  free(z);
  ...
  return -7;
}
```

De två viktigaste minnesareorna

Stacken

Håller funktionernas minne, dvs. variabler och parametrar

Hanteras automatiskt av C

LIFO-ordning, vår ”tallriksmodell”

Heapen

Allt minne som allokeras dynamiskt mha malloc, calloc och realloc — och frigörs med free

Måste hanteras manuellt

En 1-dimensionell array, vårt ”rutade papper”

Om variabeln x pekar på ett minnesutrymme allokerat på heapen eller stacken ”vet inte foo”

```
int foo(int * x int y)
{
  int *z = malloc(42);
  ...
  free(z);
  ...
  return -7;
}
```

De två viktigaste minnesareorna

Stacken

Håller funktionernas minne, dvs. variabler och parametrar

Hanteras automatiskt av C

LIFO-ordning, vår ”tallriksmodell”

Heapen

Allt minne som allokeras dynamiskt mha malloc, calloc och realloc — och frigörs med free

Måste hanteras manuellt

En 1-dimensionell array, vårt ”rutade papper”

Variabeln x pekar på ett minnesutrymme allokerat på stacken

```c
int foo(int *x int y)
{
  int *z = malloc(42);
  ...
  free(z);
  ...
  return -7;
}
```

```c
int bar = 18;
foo(&bar, 1337);
```

De två viktigaste minnesareorna

Stacken

Håller funktionernas minne, dvs. variabler och parametrar

Hanteras automatiskt av C

LIFO-ordning, vår ”tallriksmodell”

Heapen

Allt minne som allokeras dynamiskt mha malloc, calloc och realloc — och frigörs med free

Måste hanteras manuellt

En 1-dimensionell array, vårt ”rutade papper”

Variabeln x pekar på ett minnesutrymme allokerat på heapen

```
int foo(int * x int y)
{
  int *z = malloc(42);
  ...
  free(z);
  ...
  return -7;
}
```

```
int *bar = calloc(16, sizeof(int));;
foo(bar, 1337);
```

De två viktigaste minnesareorna

Stacken

När en funktion anropas ”**pushar** vi en ny stack-frame” (dvs. vi lägger en ny tallrik för den funktionen överst på stapeln)

När en funktion returneras ”**poppar** vi dess stack-frame” (dvs. vi kastar bort dess tallrik och alla dess värden går förlorade som inte *kopieras* bort från den)

```
int fak(int *n)
{
  return n == 1 ? n : n * fak(n - 1);
}
```

```
fak(5);
```

De två viktigaste minnesareorna

Stacken

När en funktion anropas ”**pushar** vi en ny stack-frame” (dvs. vi lägger en ny tallrik för den funktionen överst på stapeln)

När en funktion returneras ”**poppar** vi dess stack-frame” (dvs. vi kastar bort dess tallrik och alla dess värden går förlorade som inte *kopieras* bort från den)

```
int fak(int *n)
{
  return n == 1 ? n : n * fak(n - 1);
}
```

```
fak(5);
```

n = 5

De två viktigaste minnesareorna

Stacken

När en funktion anropas ”**pushar** vi en ny stack-frame” (dvs. vi lägger en ny tallrik för den funktionen överst på stapeln)

När en funktion returneras ”**poppar** vi dess stack-frame” (dvs. vi kastar bort dess tallrik och alla dess värden går förlorade som inte *kopieras* bort från den)

```
int fak(int *n)
{
  return n == 1 ? n : n * fak(n - 1);
}
```

```
fak(5);
```

n = 4
n = 5

De två viktigaste minnesareorna

Stacken

När en funktion anropas ”**pushar** vi en ny stack-frame” (dvs. vi lägger en ny tallrik för den funktionen överst på stapeln)

När en funktion returneras ”**poppar** vi dess stack-frame” (dvs. vi kastar bort dess tallrik och alla dess värden går förlorade som inte *kopieras* bort från den)

```
int fak(int *n)
{
  return n == 1 ? n : n * fak(n - 1);
}
```

```
fak(5);
```

n = 3
n = 4
n = 5

De två viktigaste minnesareorna

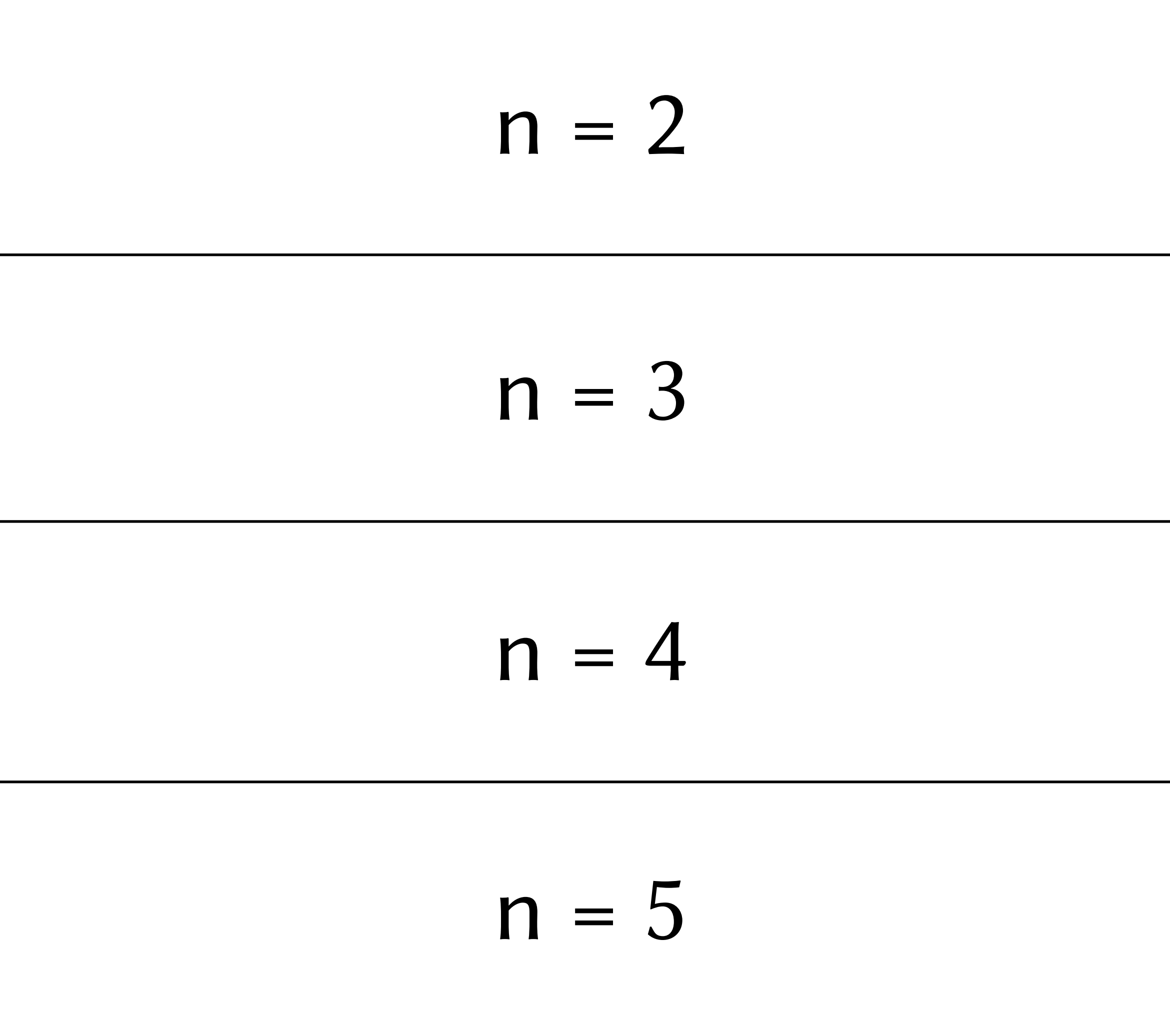
Stacken

När en funktion anropas ”**pushar** vi en ny stack-frame” (dvs. vi lägger en ny tallrik för den funktionen överst på stapeln)

När en funktion returneras ”**poppar** vi dess stack-frame” (dvs. vi kastar bort dess tallrik och alla dess värden går förlorade som inte *kopieras* bort från den)

```
int fak(int *n)
{
  return n == 1 ? n : n * fak(n - 1);
}
```

```
fak(5);
```

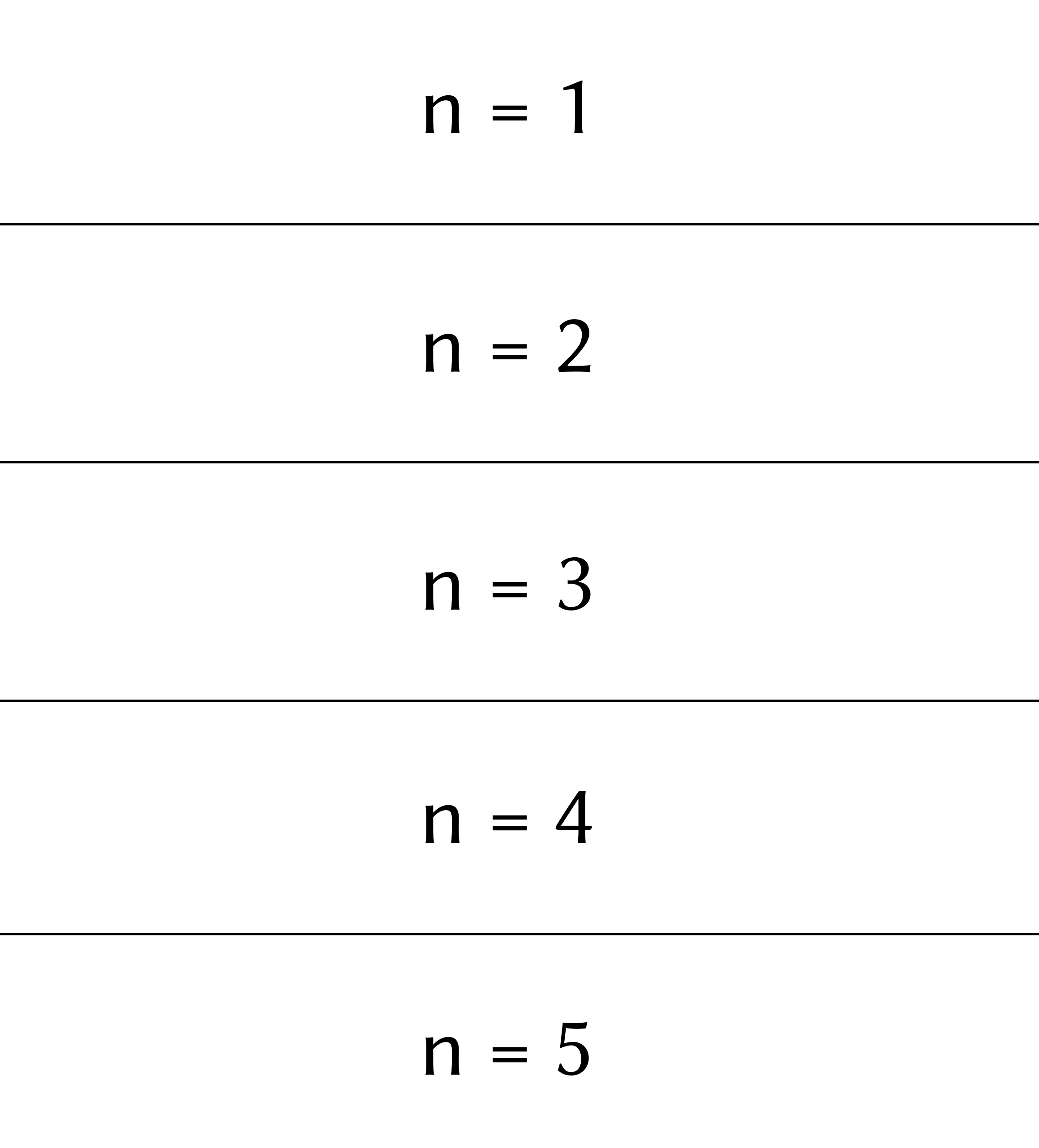


De två viktigaste minnesareorna

Stacken

När en funktion anropas ”**pushar** vi en ny stack-frame” (dvs. vi lägger en ny tallrik för den funktionen överst på stapeln)

När en funktion returneras ”**poppar** vi dess stack-frame” (dvs. vi kastar bort dess tallrik och alla dess värden går förlorade som inte *kopieras* bort från den)




```
int fak(int *n)
{
  return n == 1 ? n : n * fak(n - 1);
}
```

```
fak(5);
```

De två viktigaste minnesareorna

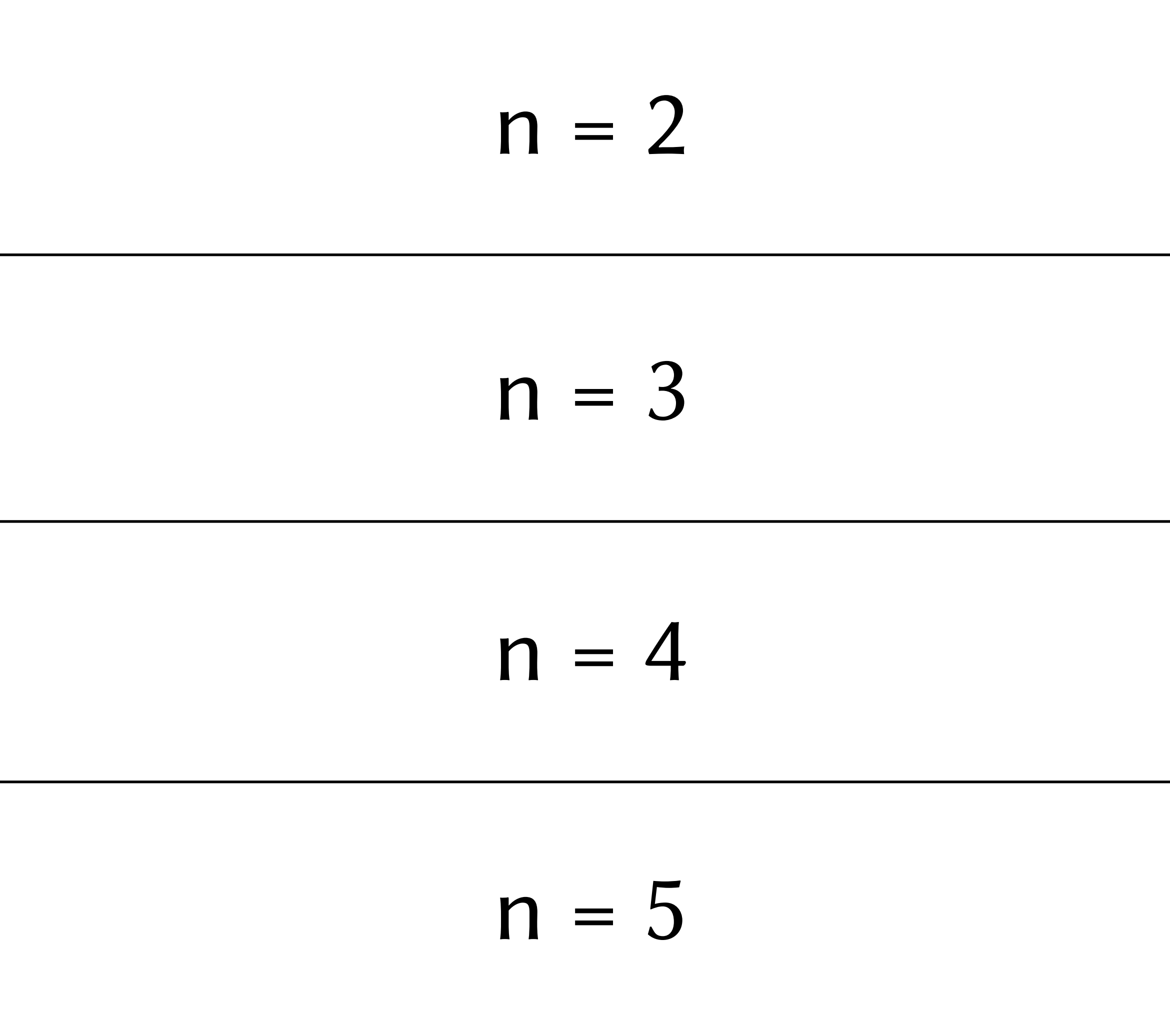
Stacken

När en funktion anropas ”**pushar** vi en ny stack-frame” (dvs. vi lägger en ny tallrik för den funktionen överst på stapeln)

När en funktion returneras ”**poppar** vi dess stack-frame” (dvs. vi kastar bort dess tallrik och alla dess värden går förlorade som inte *kopieras* bort från den)

```
int fak(int *n)
{
  return n == 1 ? n : n * fak(n - 1);
}
```

```
fak(5);
```



De två viktigaste minnesareorna

Stacken

När en funktion anropas ”**pushar** vi en ny stack-frame” (dvs. vi lägger en ny tallrik för den funktionen överst på stapeln)

När en funktion returneras ”**poppar** vi dess stack-frame” (dvs. vi kastar bort dess tallrik och alla dess värden går förlorade som inte *kopieras* bort från den)

```
int fak(int *n)
{
  return n == 1 ? n : n * fak(n - 1);
}
```

```
fak(5);
```

n = 3
n = 4
n = 5

De två viktigaste minnesareorna

Stacken

När en funktion anropas ”**pushar** vi en ny stack-frame” (dvs. vi lägger en ny tallrik för den funktionen överst på stapeln)

När en funktion returneras ”**poppar** vi dess stack-frame” (dvs. vi kastar bort dess tallrik och alla dess värden går förlorade som inte *kopieras* bort från den)

```
int fak(int *n)
{
  return n == 1 ? n : n * fak(n - 1);
}
```

```
fak(5);
```

n = 4
n = 5

De två viktigaste minnesareorna

Stacken

När en funktion anropas ”**pushar** vi en ny stack-frame” (dvs. vi lägger en ny tallrik för den funktionen överst på stapeln)

När en funktion returneras ”**poppar** vi dess stack-frame” (dvs. vi kastar bort dess tallrik och alla dess värden går förlorade som inte *kopieras* bort från den)

```
int fak(int *n)
{
  return n == 1 ? n : n * fak(n - 1);
}
```

```
fak(5);
```

n = 5

De två viktigaste minnesareorna

Stacken

När en funktion anropas ”**pushar** vi en ny stack-frame” (dvs. vi lägger en ny tallrik för den funktionen överst på stapeln)

När en funktion returneras ”**poppar** vi dess stack-frame” (dvs. vi kastar bort dess tallrik och alla dess värden går förlorade som inte *kopieras* bort från den)

```
int fak(int *n)
{
  return n == 1 ? n : n * fak(n - 1);
}
```

```
fak(5);
```

De två viktigaste minnesareorna

Heapen

Anrop till malloc, calloc eller realloc reserverar ett sammanhängande utrymme i minnet som vi kan komma åt via dess adress (dvs. en pekare)

Vi måste frigöra (hela) utrymmet manuellt

Funktionen realloc låter oss krympa ett utrymme från dess högsta adress

Vi kan även växa ett utrymme från dess högsta adress, vilket ev. medför att utrymmet måste flyttas får att få plats

```c
int *p = calloc(2, sizeof(int));
```

Storleken på en int

upptaget

upptaget

upptaget

De två viktigaste minnesareorna

Heapen

Anrop till malloc, calloc eller realloc reserverar ett sammanhängande utrymme i minnet som vi kan komma åt via dess adress (dvs. en pekare)

Vi måste frigöra (hela) utrymmet manuellt

Funktionen realloc låter oss krympa ett utrymme från dess högsta adress

Vi kan även växa ett utrymme från dess högsta adress, vilket ev. medför att utrymmet måste flyttas får att få plats

```
int *p = calloc(2, sizeof(int));
```

p

Storleken på en int

0 0

upptaget

upptaget

upptaget

De två viktigaste minnesareorna

Heapen

Anrop till malloc, calloc eller realloc reserverar ett sammanhängande utrymme i minnet som vi kan komma åt via dess adress (dvs. en pekare)

Vi måste frigöra (hela) utrymmet manuellt

Funktionen realloc låter oss krympa ett utrymme från dess högsta adress

Vi kan även växa ett utrymme från dess högsta adress, vilket ev. medför att utrymmet måste flyttas får att få plats

```
*p = 42;
p[1] = 4711;
```

p

Storleken på en int

42

4711

upptaget

upptaget

upptaget

De två viktigaste minnesareorna

Heapen

Anrop till malloc, calloc eller realloc reserverar ett sammanhängande utrymme i minnet som vi kan komma åt via dess adress (dvs. en pekare)

Vi måste frigöra (hela) utrymmet manuellt

Funktionen realloc låter oss krympa ett utrymme från dess högsta adress

Vi kan även växa ett utrymme från dess högsta adress, vilket ev. medför att utrymmet måste flyttas får att få plats

Storleken på en int

```
int *q = p; // aliasering!
*q = 9001;
```

De två viktigaste minnesareorna

Heapen

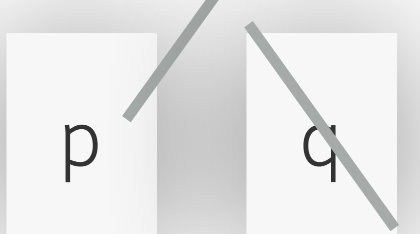
Anrop till malloc, calloc eller realloc reserverar ett sammanhängande utrymme i minnet som vi kan komma åt via dess adress (dvs. en pekare)

Vi måste frigöra (hela) utrymmet manuellt

Funktionen realloc låter oss krympa ett utrymme från dess högsta adress

Vi kan även växa ett utrymme från dess högsta adress, vilket ev. medför att utrymmet måste flyttas får att få plats

```
int *q = p; // aliasering!
*q = 9001;
q = NULL;
```



De två viktigaste minnesareorna

Heapen

Anrop till malloc, calloc eller realloc reserverar ett sammanhängande utrymme i minnet som vi kan komma åt via dess adress (dvs. en pekare)

Vi måste frigöra (hela) utrymmet manuellt

Funktionen realloc låter oss krympa ett utrymme från dess högsta adress

Vi kan även växa ett utrymme från dess högsta adress, vilket ev. medför att utrymmet måste flyttas får att få plats

```
int *q = p; // aliasering!
*q = 9001;
q = p+1;
```

Storleken på en int

9001 4711 upptaget

upptaget

upptaget

p q

De två viktigaste minnesareorna

Heapen

Anrop till malloc, calloc eller realloc reserverar ett sammanhängande utrymme i minnet som vi kan komma åt via dess adress (dvs. en pekare)

Vi måste frigöra (hela) utrymmet manuellt

Funktionen realloc låter oss krympa ett utrymme från dess högsta adress

Vi kan även växa ett utrymme från dess högsta adress, vilket ev. medför att utrymmet måste flyttas får att få plats

```
q = realloc(p, 3 * sizeof(int));
```

```
p
```

De två viktigaste minnesareorna

Heapen

Anrop till malloc, calloc eller realloc reserverar ett sammanhängande utrymme i minnet som vi kan komma åt via dess adress (dvs. en pekare)

Vi måste frigöra (hela) utrymmet manuellt

Funktionen realloc låter oss krympa ett utrymme från dess högsta adress

Vi kan även växa ett utrymme från dess högsta adress, vilket ev. medför att utrymmet måste flyttas får att få plats

```
q = realloc(p, 3 * sizeof(int));
```

De två viktigaste minnesareorna

Heapen

Anrop till malloc, calloc eller realloc reserverar ett sammanhängande utrymme i minnet som vi kan komma åt via dess adress (dvs. en pekare)

Vi måste frigöra (hela) utrymmet manuellt

Funktionen realloc låter oss krympa ett utrymme från dess högsta adress

Vi kan även växa ett utrymme från dess högsta adress, vilket ev. medför att utrymmet måste flyttas får att få plats

```
free(q);
```

C-pussel!

Finn tre fel!

```
void example()
{
  char *buf = calloc(6, sizeof(char));

  strcpy(buf, "Hello!");

  printf("My string: '%s'\n", buf);
}
```

buf

0 0 0 0 0 0

upptaget

upptaget

upptaget

Finn tre fel!

```
void example()
{
  char *buf = calloc(6, sizeof(char));

  strcpy(buf, "Hello!");

  printf("My string: '%s'\n", buf);
}
```

buf

H e l l o ! \0

upptaget

upptaget

upptaget

Finn tre fel!

```c
void example()
{
  char *buf = calloc(6, sizeof(char));

  strcpy(buf, "Hello!");

  printf("My string: '%s'\n", buf);
}
```

Invalid write!

buf

Finn tre fel!

```
void example()
{
  char *buf = calloc(6, sizeof(char));

  strcpy(buf, "Hello!");

  printf("My string: '%s'\n", buf);
}
```

Invalid write!

Invalid read!

Finn tre fel!

```
void example()
{
  char *buf = calloc(6, sizeof(char));

  strcpy(buf, "Hello!");

  printf("My string: '%s'\n", buf);
}
```

Invalid write!

Invalid read!

Memory leak!

(iom att buf*-pekaren inte längre går att nå från programmet och därmed inte heller att anropa* free *på!)*

Struktar är minneskartor

Sammasatta datatyper

Låter oss bygga komplexa datatyper från andra datatyper

Ordningen i strukten speglar minneslayouten

```
struct point
{
  int x;
  int y;
};
```

```
struct rectangle
{
  struct point upper_left;
  struct point lower_right;
};
```

```
calloc(1, sizeof(struct point));
```

```
calloc(2, sizeof(struct point));
```

Storleken på en int

Typalias

Vi kan bygga hjälpsamma alias till typnamn

Vi kan sedan använda aliaset istället för att mena samma sak

Dessa kan användas för att…

…abstrahera, ex:

`typedef uint8_t age;` — en ålder är ett heltal 0–255

…förenkla, ex:

`typedef bool(*eq_test)(point_t *, point_t *);` — `eq_test` är en funktion som jämför två punkter

~ inkapsling, ex:

`typedef struct point point_t;` — när inga andra detaljer om strukten ”point” ges

Länkade datastrukturer

Ritnotation

Structen…

```
typedef struct link link_t;

struct link
{
  int value;
  link_t *next;
};
```

…ritar vi så här

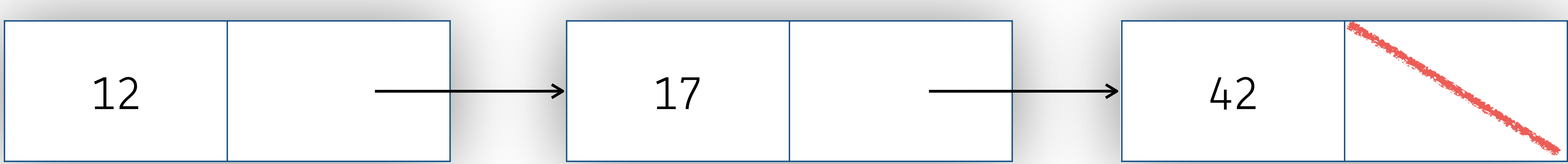
value	next

…eller i bland så här (värden istället för posternas namn)

42	

strecket betyder NULL — **ibland skriver vi ut** NULL

Länkarna i en länkad lista ritar vi så här



Dessa avser samma strukt som föregående bild, dvs. `link_t`

Struktarna som bygger upp en länkad lista

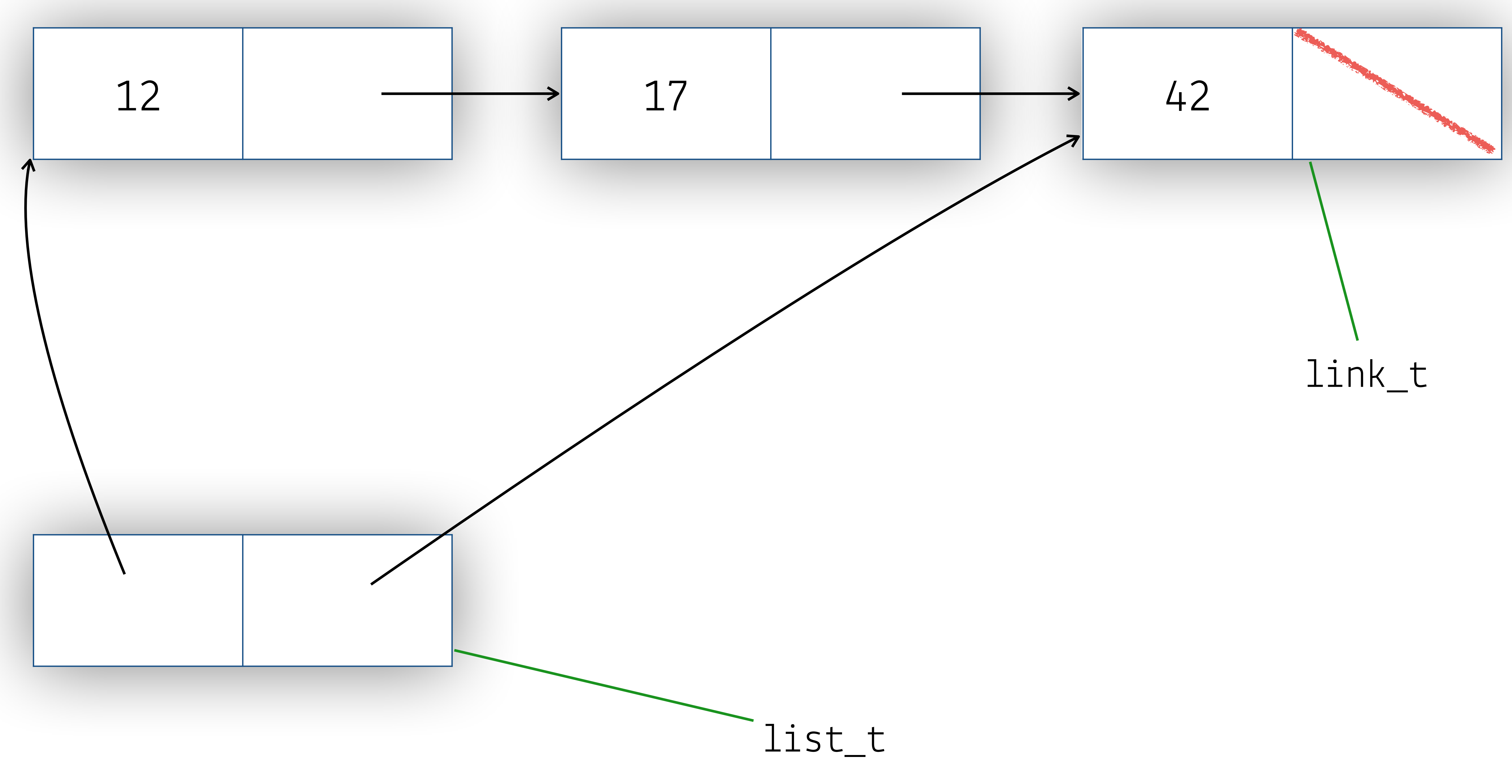
”Själva listan”

```
typedef struct list list_t;

struct list
{
  link_t *first;
  link_t *last;
};
```

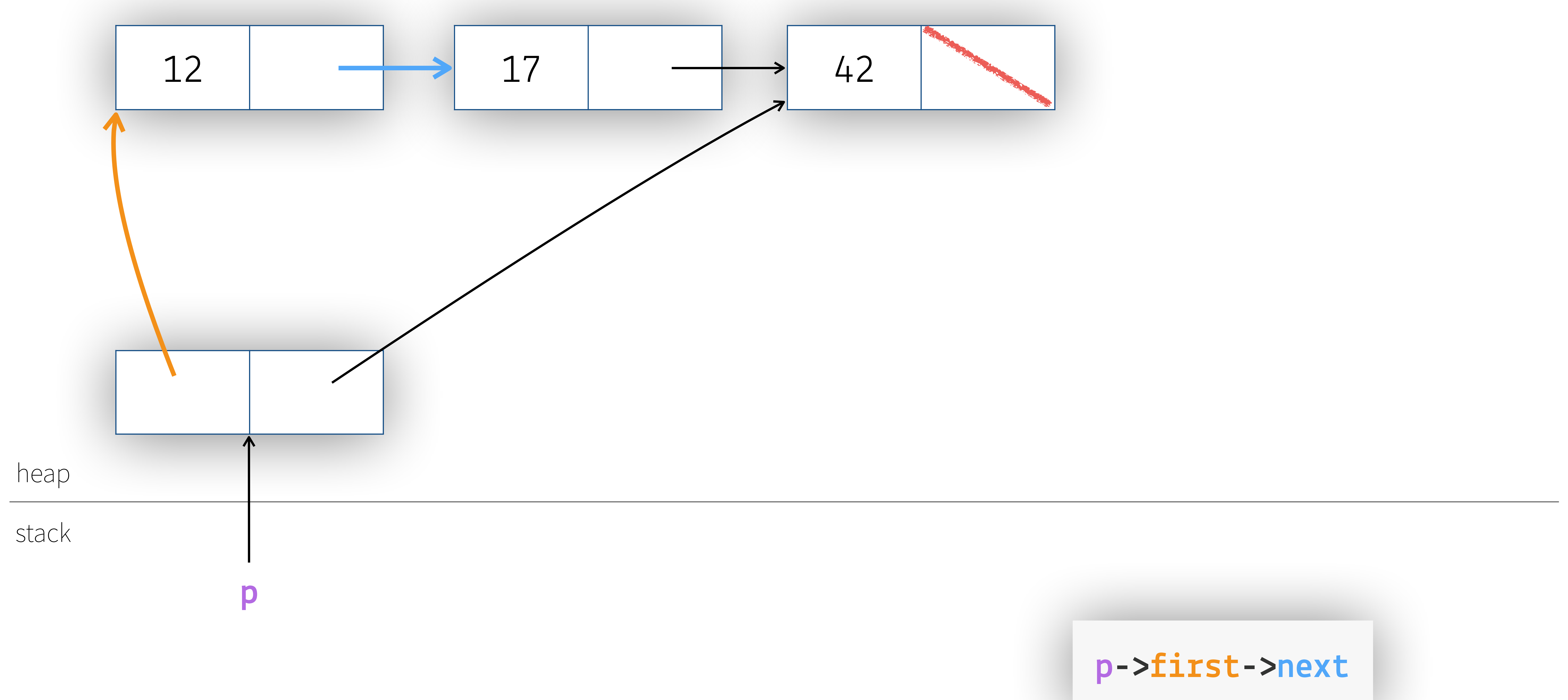
Länkarna

```
typedef struct link link_t;

struct link
{
  int value;
  link_t *next;
};
```

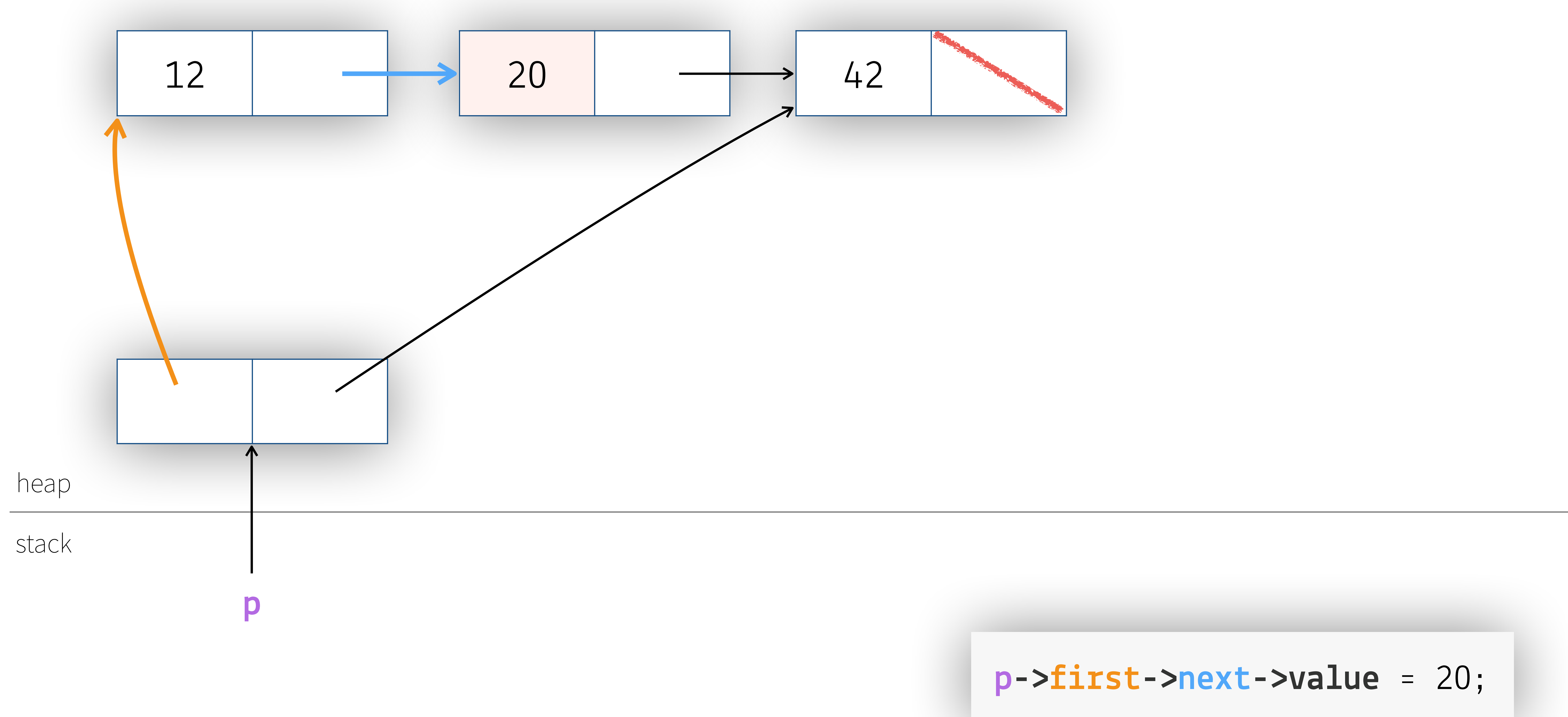
En komplett länkad lista



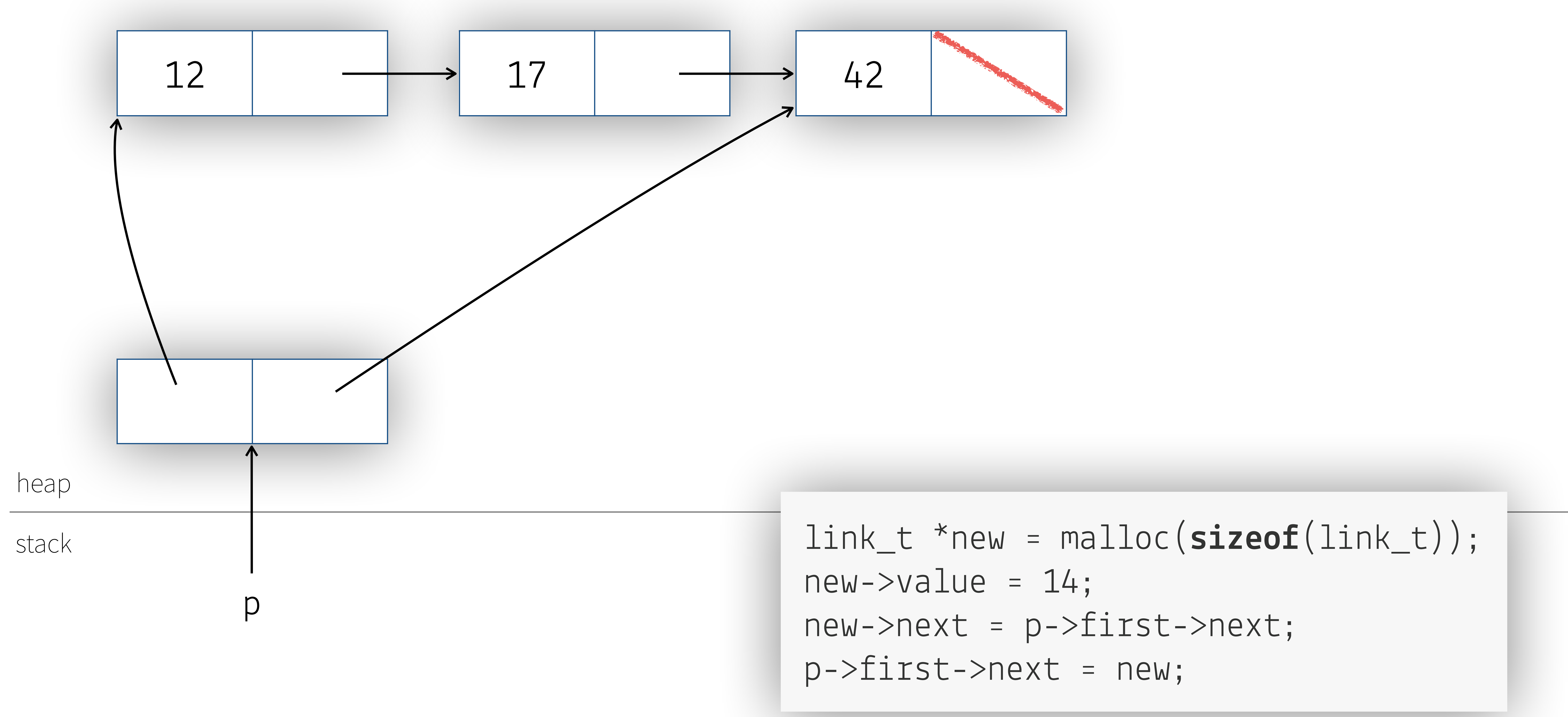
De olika delarna och hur man kommer åt dem



De olika delarna och hur man kommer åt dem

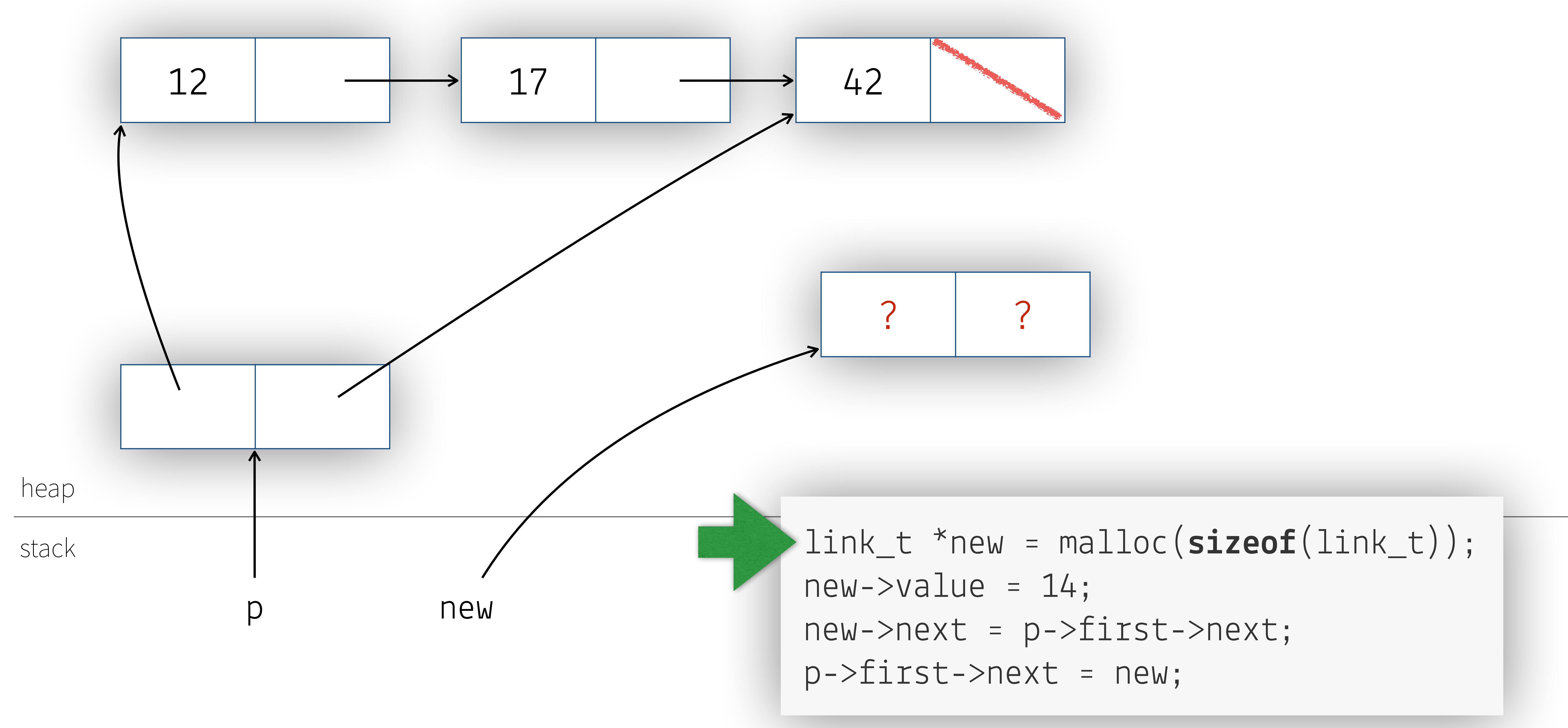


Lägg till ett element i listan

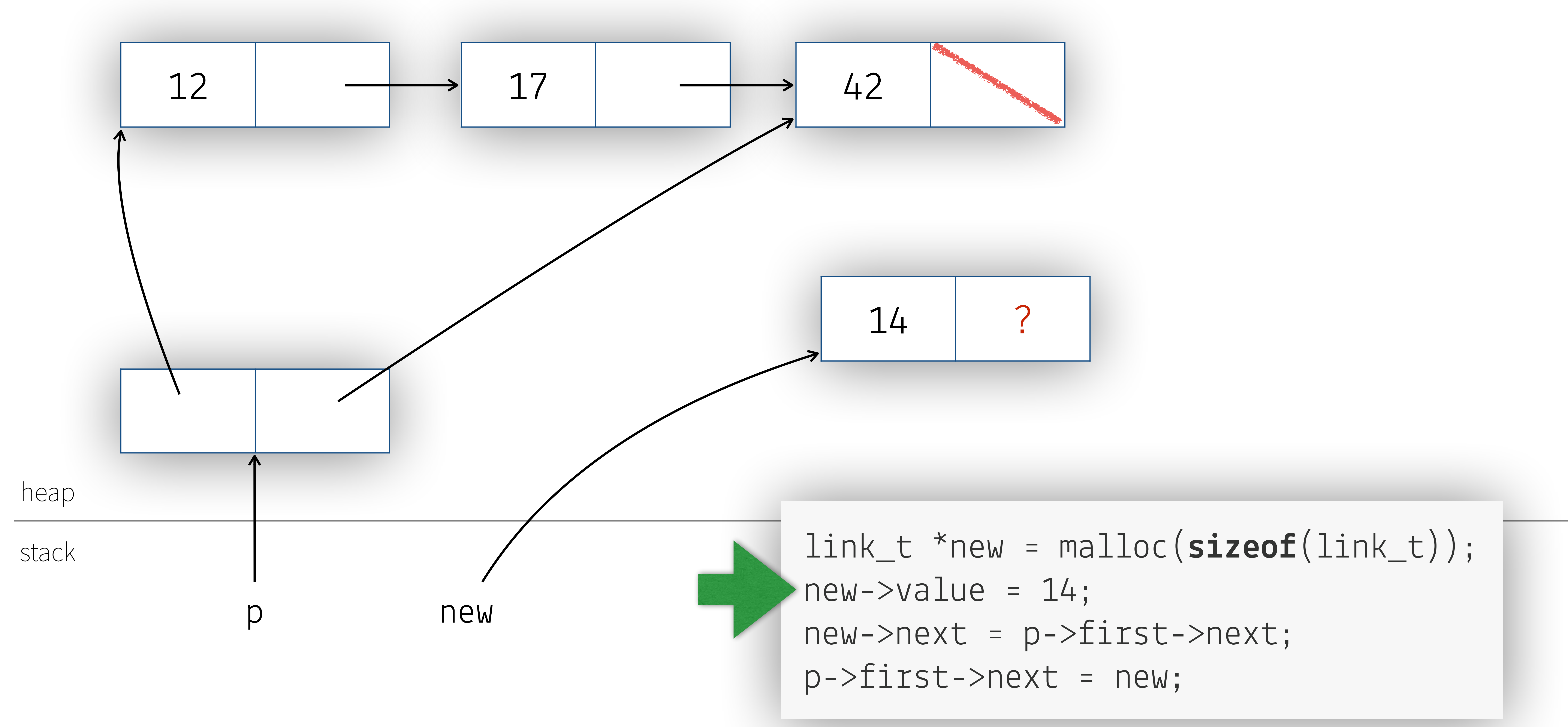



```
link_t *new = malloc(sizeof(link_t));
new->value = 14;
new->next = p->first->next;
p->first->next = new;
```

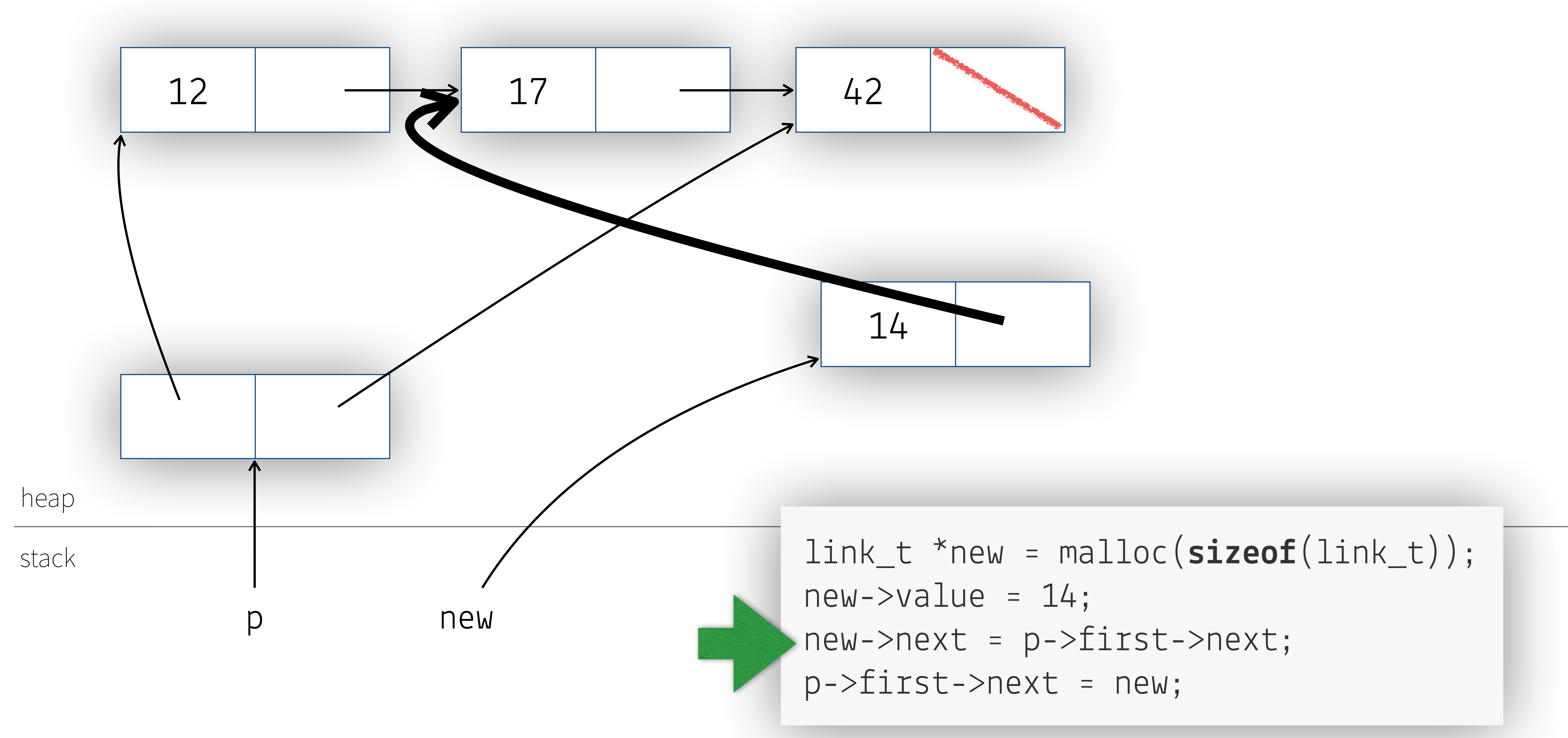
Lägg till ett element i listan [steg 1]



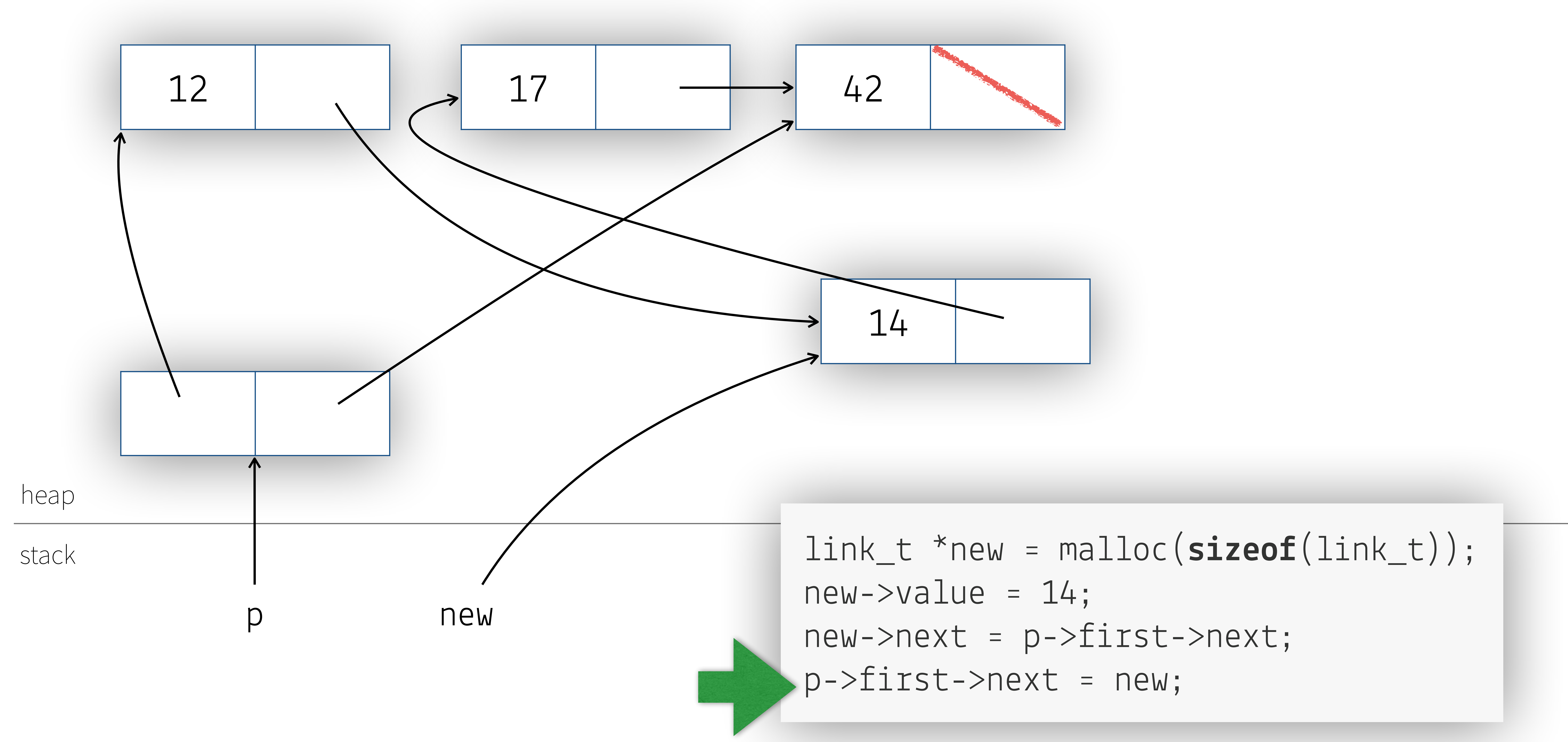
Lägg till ett element i listan [steg 2]



Lägg till ett element i listan [steg 3]

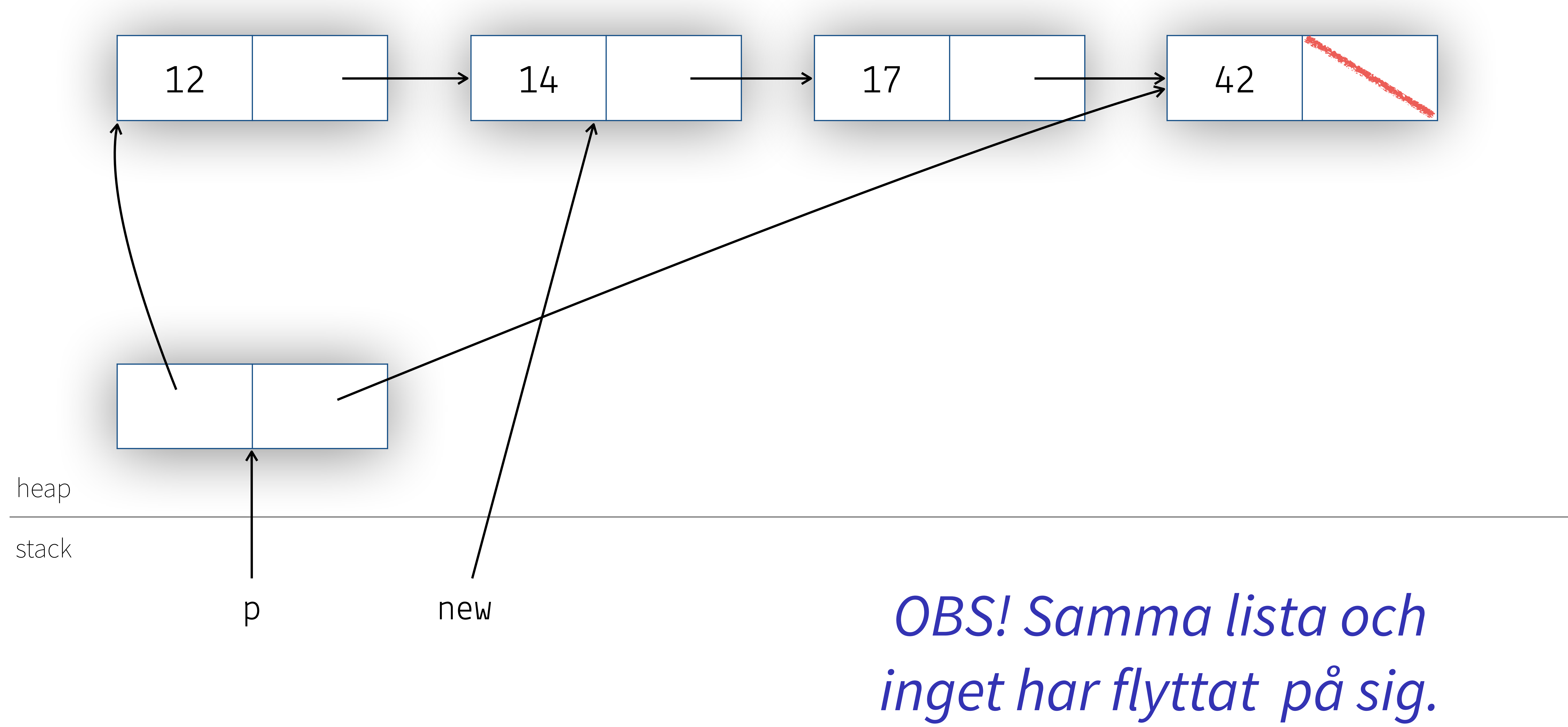


Lägg till ett element i listan [steg 4]




```
link_t *new = malloc(sizeof(link_t));
new->value = 14;
new->next = p->first->next;
p->first->next = new;
```

Listan mindre rörigt ritad



OBS! Samma lista och inget har flyttat på sig.

Förslag till övning på kammaren

- Skriv ett program som tar in strängar som kommandoradsargument och som stoppar in dem i en länkad lista och skriver ut dem

  Utgå från programmet nedan som loopar genom kommandoradsargumenten och skriver ut dem (utan en länkad lista)

- Skriv en funktion `list_append` som stoppar in sista med hjälp av `last` i `list_t`
- Skriv en funktion `list_prepend` som stoppar in sista med hjälp av `first` i `list_t`
- Skriv en funktion `list_insert` som använder `strcmp` för att stoppa in strängarna i sorteringsordning

```
int main(int argc, char *argv[])
{
  for(int i = 0; i < argc; ++i) puts(argv[i]);
  return 0;
}
```

Att frigöra en länkad lista

- Ny notation:

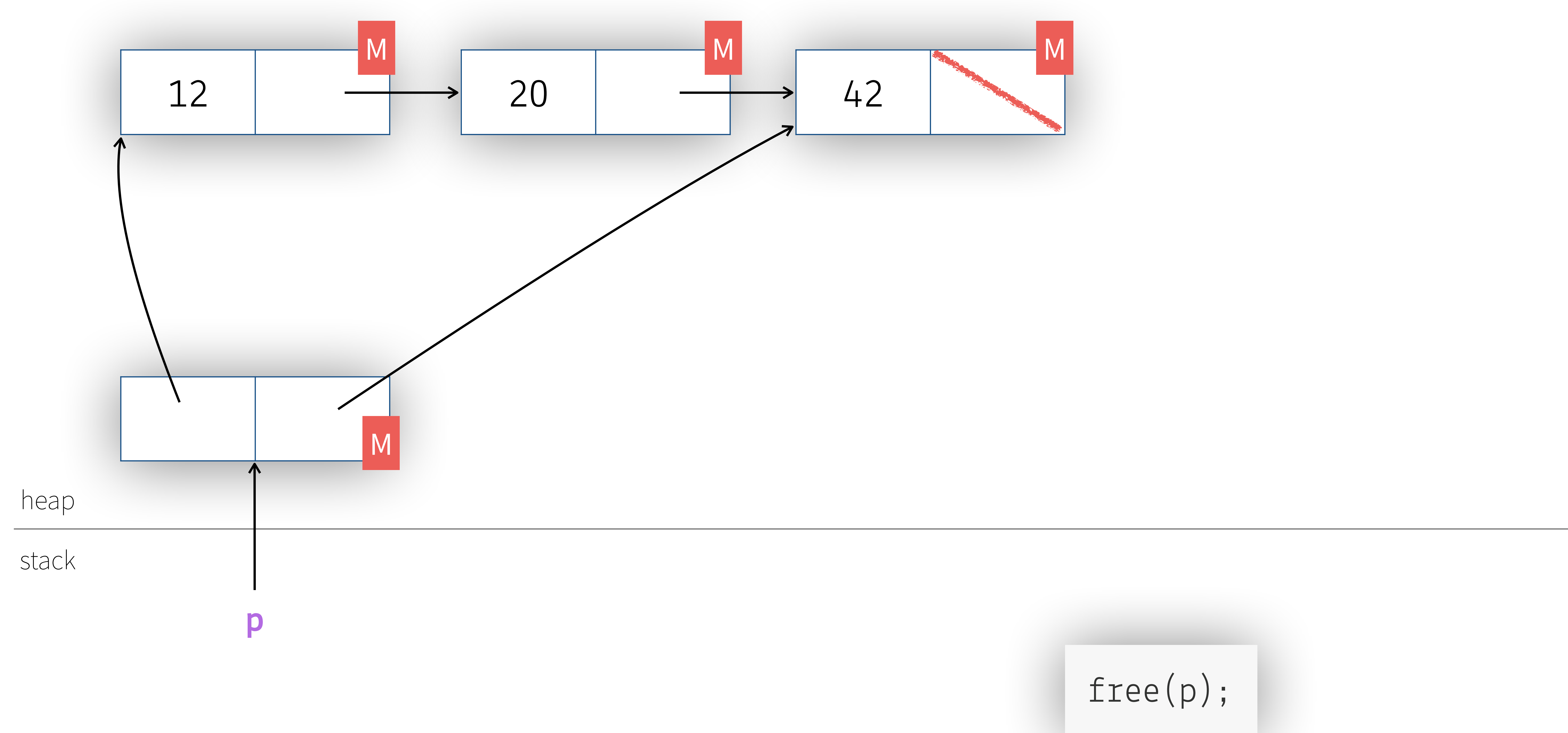
M betyder att `malloc` anser att strukten används

F betyder att `malloc` anser att strukten inte används och är fri att återanvända dess minne

OBS! Vi får inte läsa struktar vars minne är F — för de kanske inte finns längre

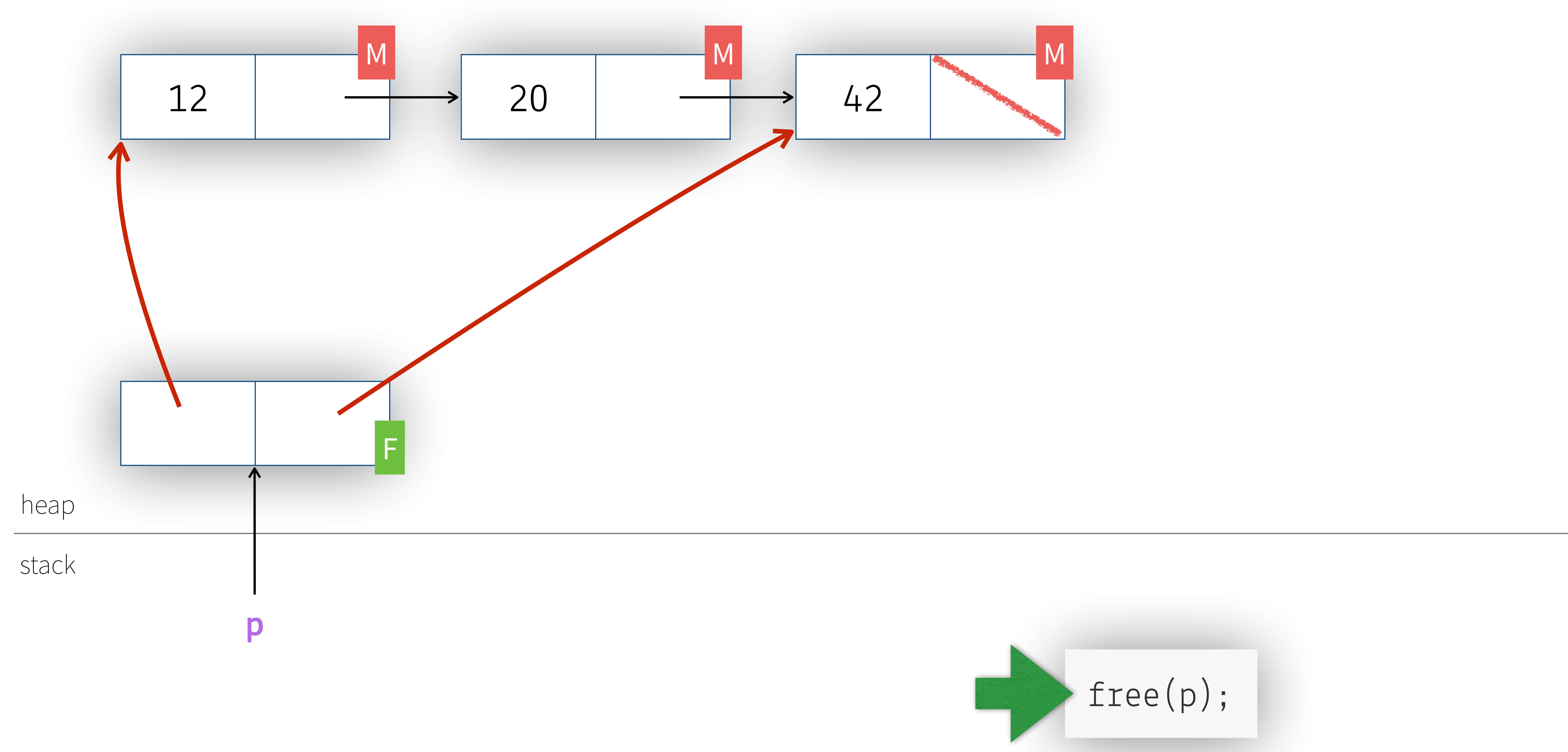
Röda värden i strukten betyder att de kanske inte finns längre

Frigöra en länkad lista — försök 1 (1/2) [OBS! Fel]




```
free(p);
```

Frigöra en länkad lista — försök 1 (2/2) [OBS! Fel]



Observationer

- Så fort vi gör `free(p)` så kan malloc återanvända det minne som p pekar på
- När sker det? Generellt omöjligt att veta.
- Därför:

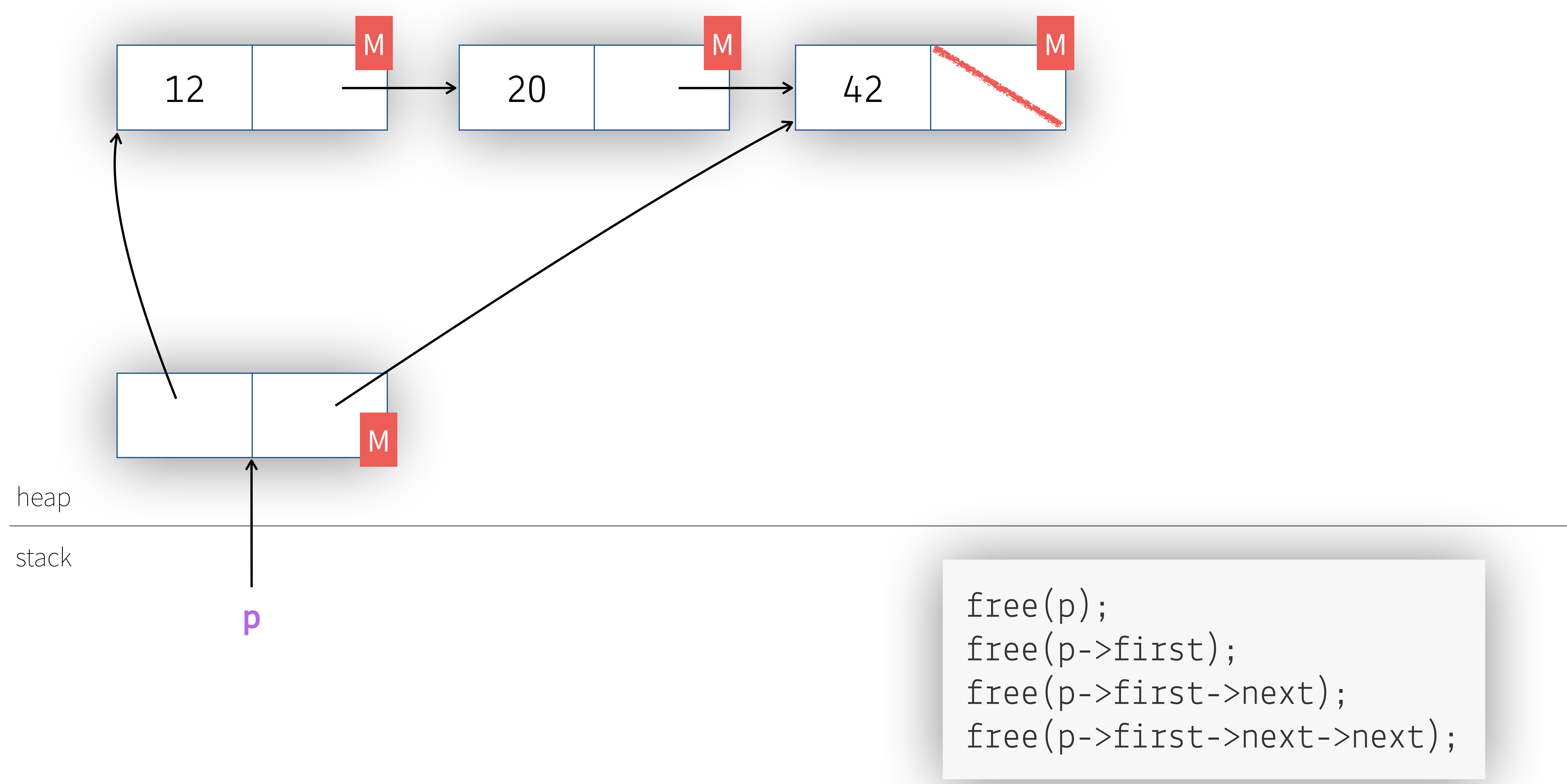
  efter att du gjort `free(p)` får inte inte använda p igen förrän du har pekat om `p` att peka på något data som du vet är validt

- Konsekvens av `free(p)` blir därför

  vi förlorar möjligheten att komma åt `p->first`, `p->first->next`,
  och `p->first->next->next`

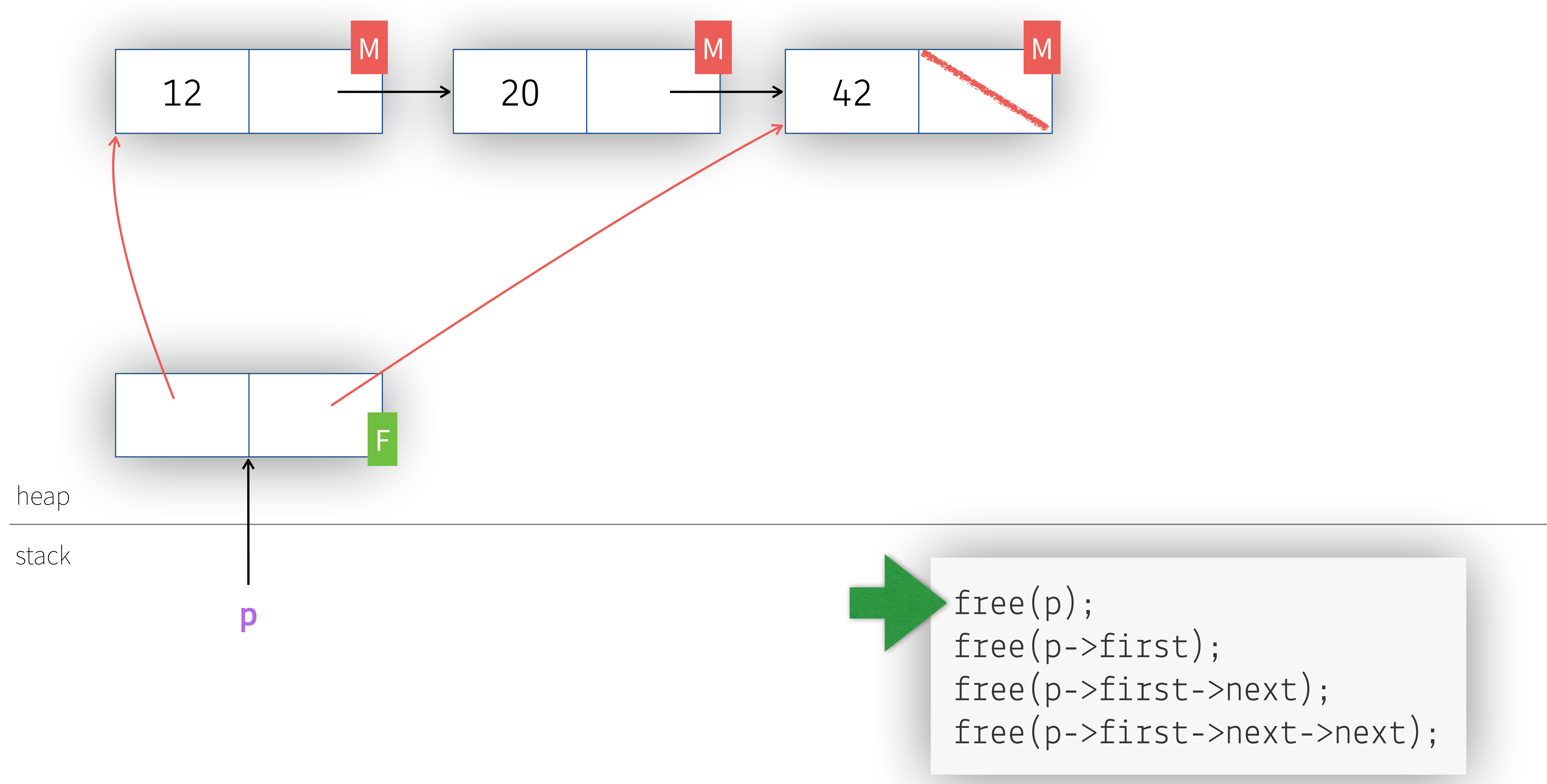
  …vilket i förlängningen betyder att vi läcker minne (3 `link_t`-struktar för att vara exakt)

Frigöra en länkad lista — försök 2 (1/3) [OBS! Fel igen]

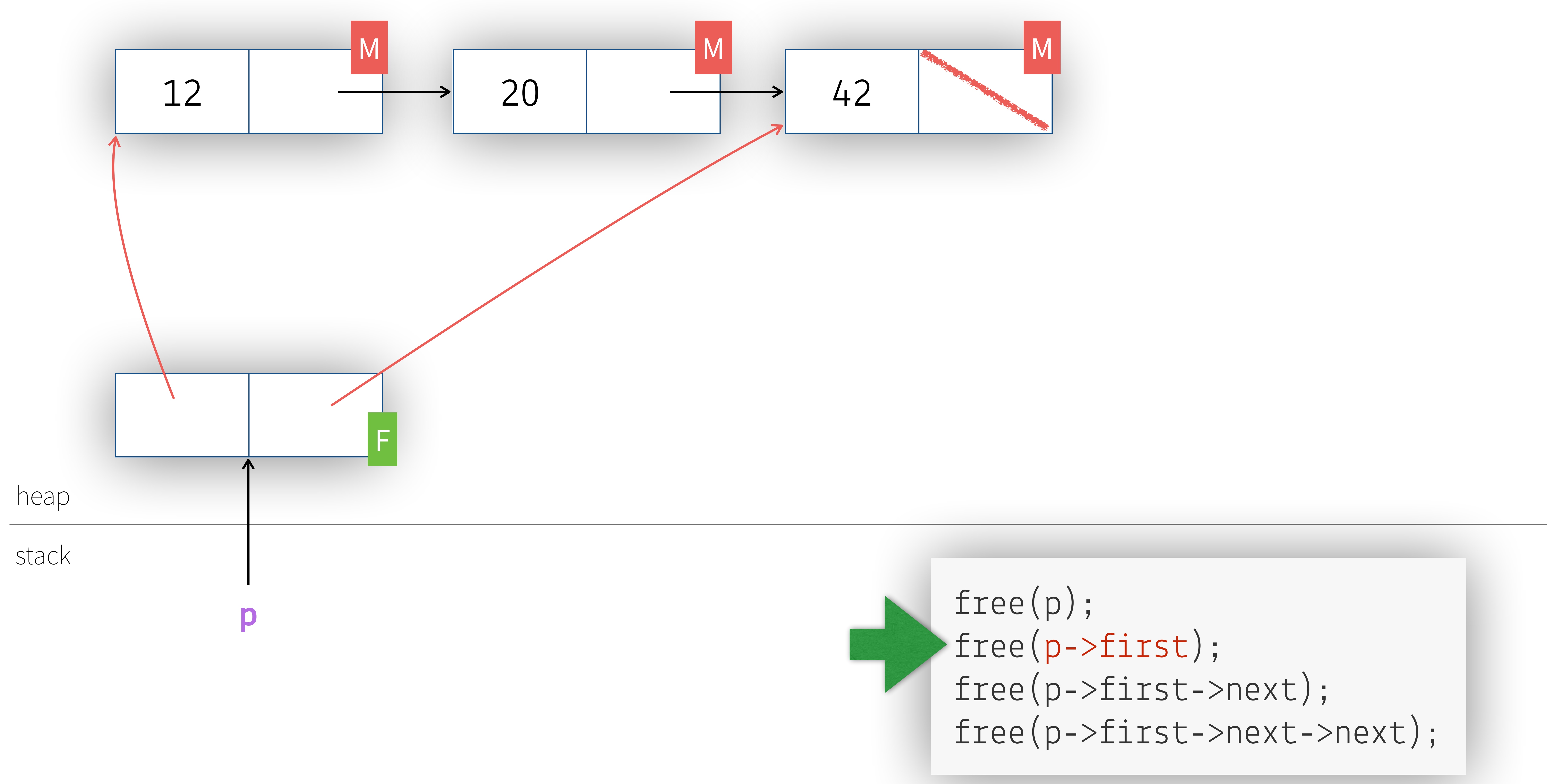



```
free(p);
free(p->first);
free(p->first->next);
free(p->first->next->next);
```

Frigöra en länkad lista — försök 2 (2/3) [OBS! Fel igen]



Frigöra en länkad lista — försök 2 (3/3) [OBS! Fel igen]




```
free(p);
free(p->first);
free(p->first->next);
free(p->first->next->next);
```

Observationer

- Vi gör free i omvänd ordning

  Vi måste börja i slutet av listan så att vi inte hela tiden frigör det minne som vi sedan vill läsa för att komma åt next-pekaren

- Det finns risk att program som gör så här fungerar ändå

  T.ex. för att inget minne återanvänds mellan `free(p);` och `free(p->next);`

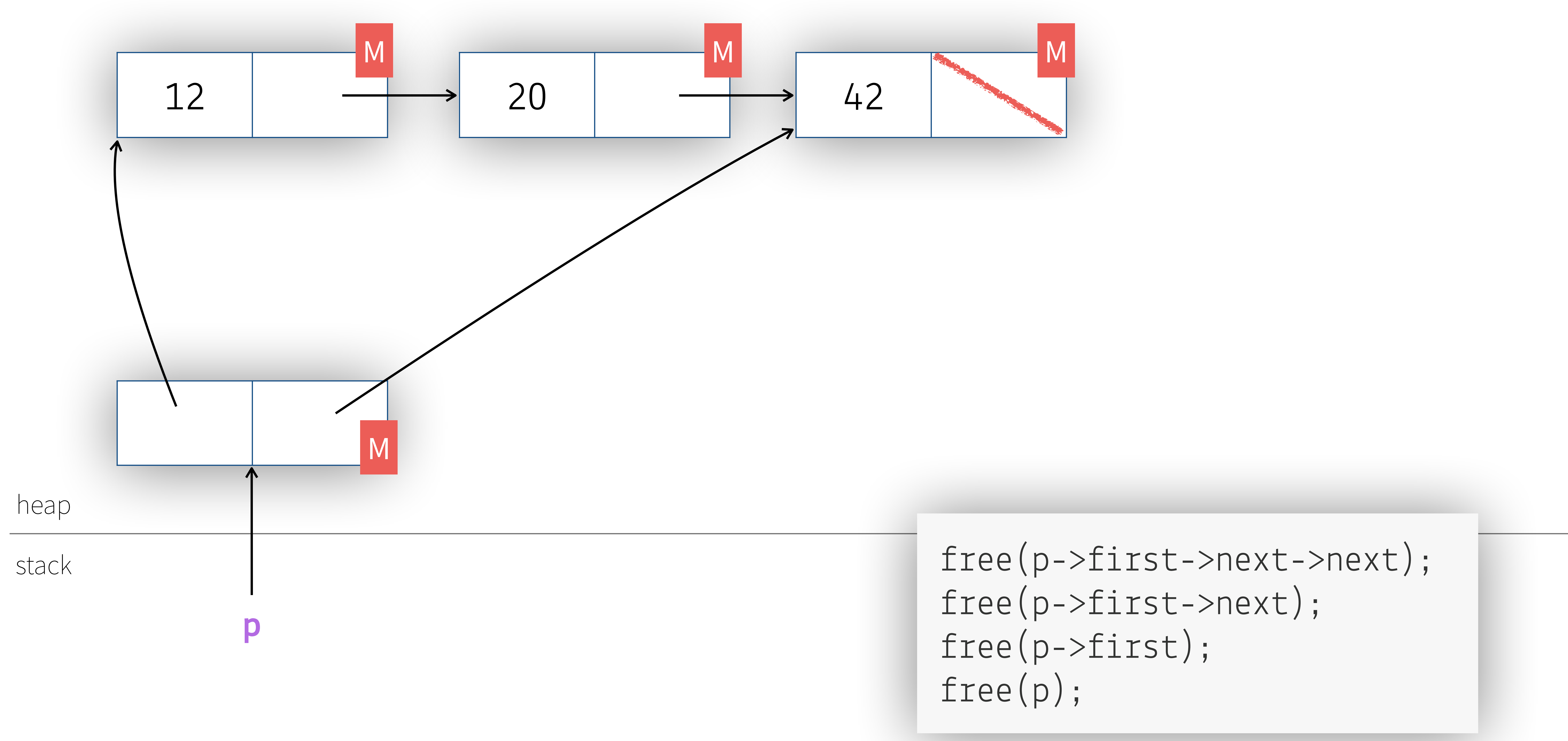
  ...men det är ändå ett felaktigt program

- Ta hjälp av **valgrind** för att hitta denna typ av fel

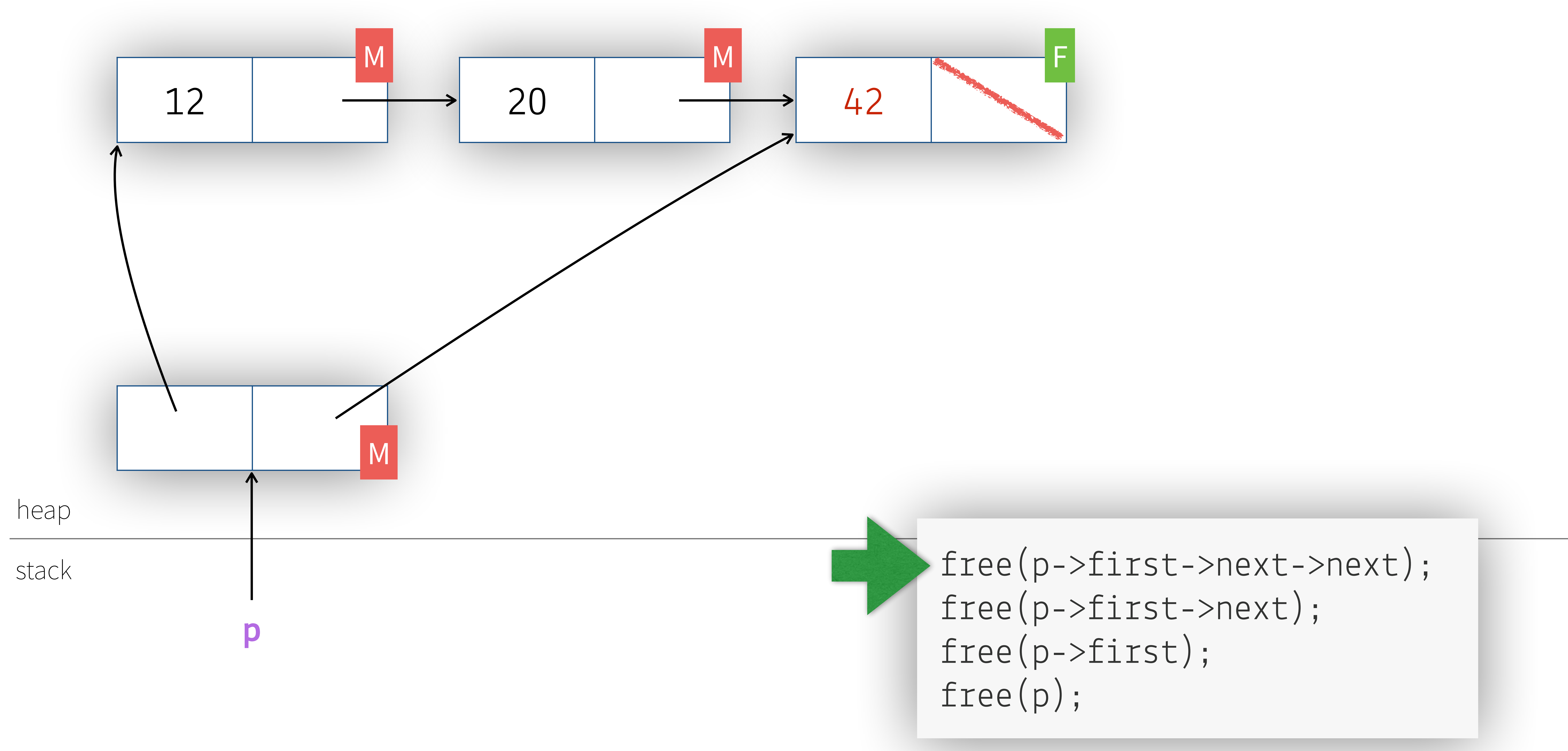
  ”Invalid read of size 8” — du läser 8 bytes av minne som inte är i en strukt som är M

- Efter `free(p)` blir p en s.k. **dangling pointer** — en pekare till ett minnesblock som inte längre finns

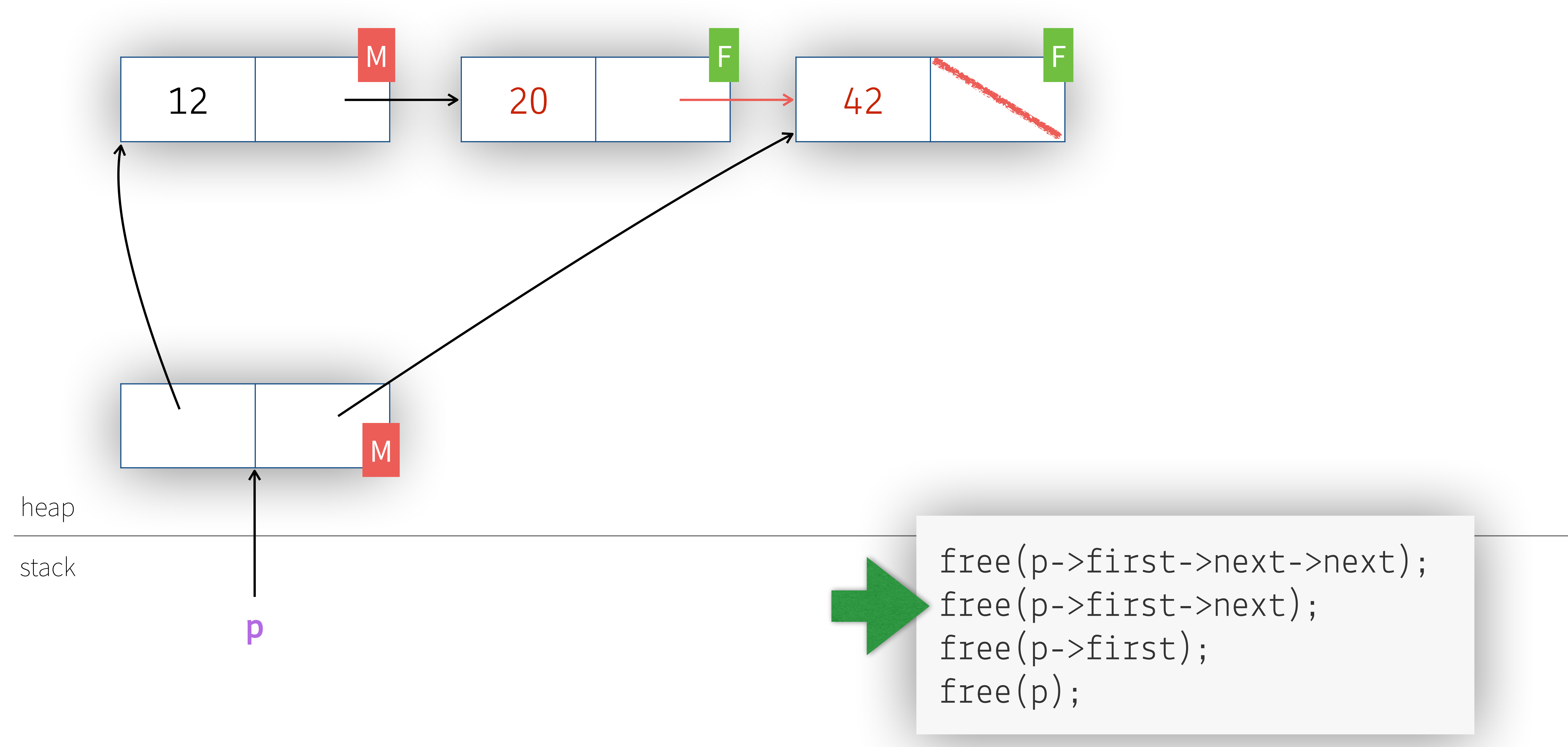
Frigöra en länkad lista — försök 3 [OBS! Korrekt]



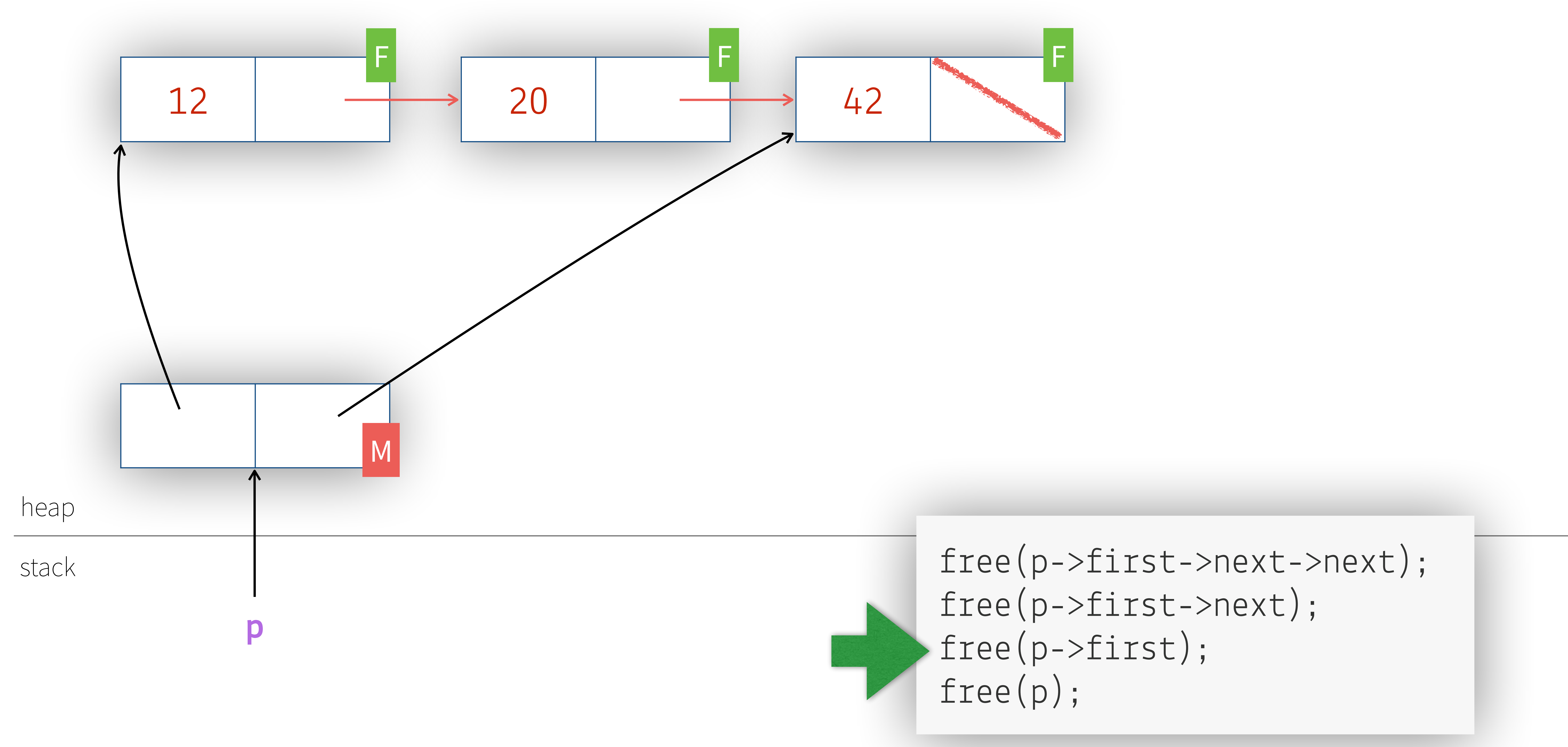
Frigöra en länkad lista — försök 3 [OBS! Korrekt]



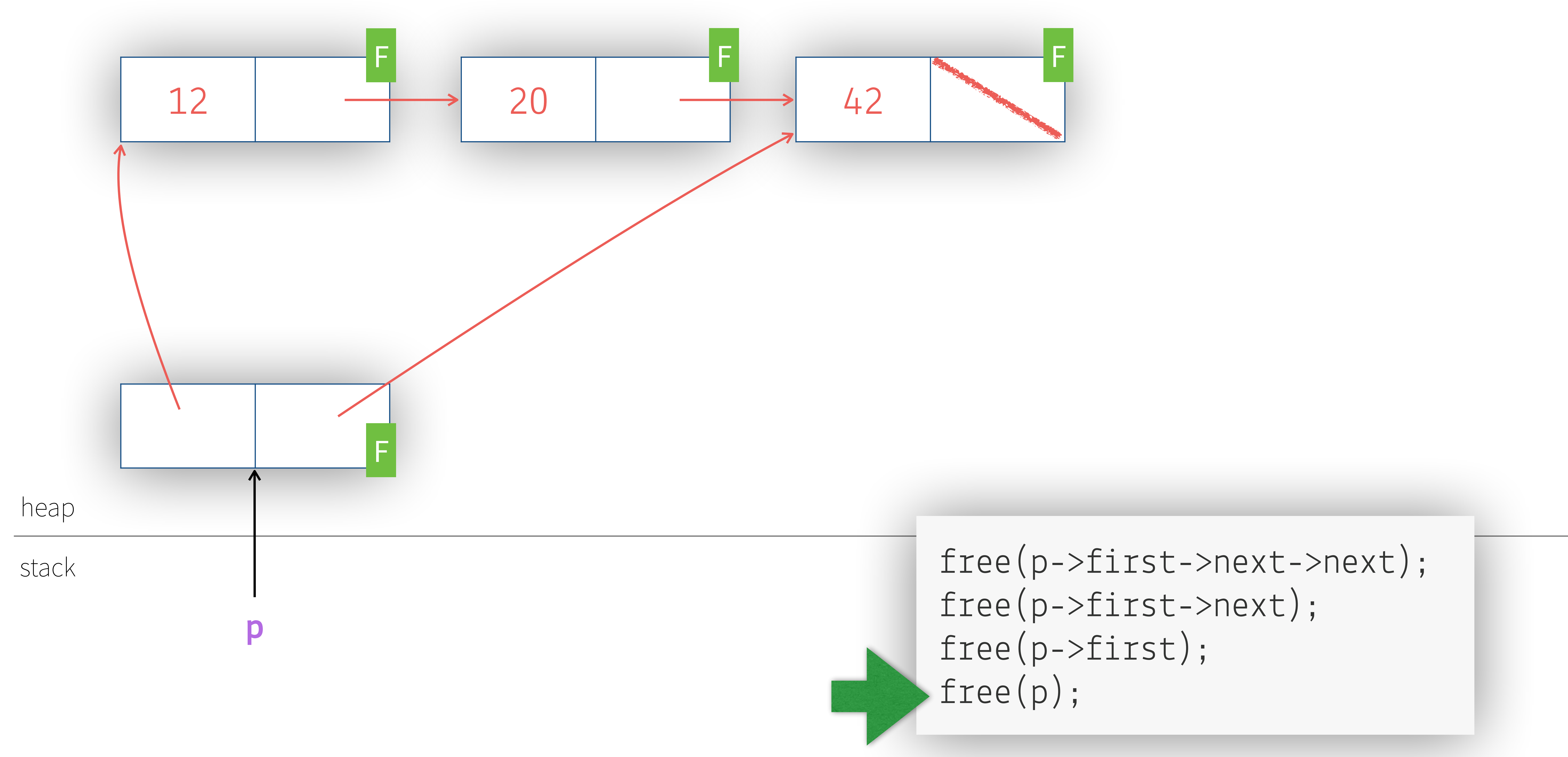
Frigöra en länkad lista — försök 3 [OBS! Korrekt]



Frigöra en länkad lista — försök 3 [OBS! Korrekt]



Frigöra en länkad lista — försök 3 [OBS! Korrekt]



Observationer

- Observera att minnet inte ”suddas” när man gör `free`
- Det är därför som det finns skräpdata överallt när man allokerar med `malloc`
- Om du har **otur** kan du lyckas frigöra minne och sedan använda det utan att märka det

Ett litet tips för att undvika detta problem — använd nedanstående istället för `free`

```
#define Free(ptr) { free(ptr); ptr = NULL; }
```

Med `Free(p);` sätts p till NULL som sido-effekt vilket kommer att få kod som gör felet i försök två att krascha med ett **segfault**

(det hade stått `Free(p); Free(p->next); ...` — den andra `Free` hade kraschat)

Förslag till övning på kammaren

- Utöka programmet från föregående övning med stöd för att ta bort den länkade listan
- Implementera `list_free_rec` som tar bort listan med hjälp av `link_free_rec` som är en rekursiv hjälpfunktion
- Implementera `list_free_iter` som tar bort listan med hjälp av en loop och utan att anropa några hjälpfunktioner
- Använd `Free`-makrot från föregående bild för att undvika att du använder **dangling pointers** av misstag
- Använd valgrind för att verifiera att ditt program inte läcker minne

Vad valgrind gör

- Åtkomst till minne utanför en levande allokering — invalid reads och writes

  `p[2] = x;` — invalid write till utanför p

  `x = p[2];` — invalid read från utanför p

  `free(p); *p = x; x = *p;` — invalid write/read

- Allt minne som är allokerat vid programmets slut rapporteras

  Skiljer mellan nåbart och icke nåbart

- Allt minne som tappas bort (och därför ger upphov till läckor)

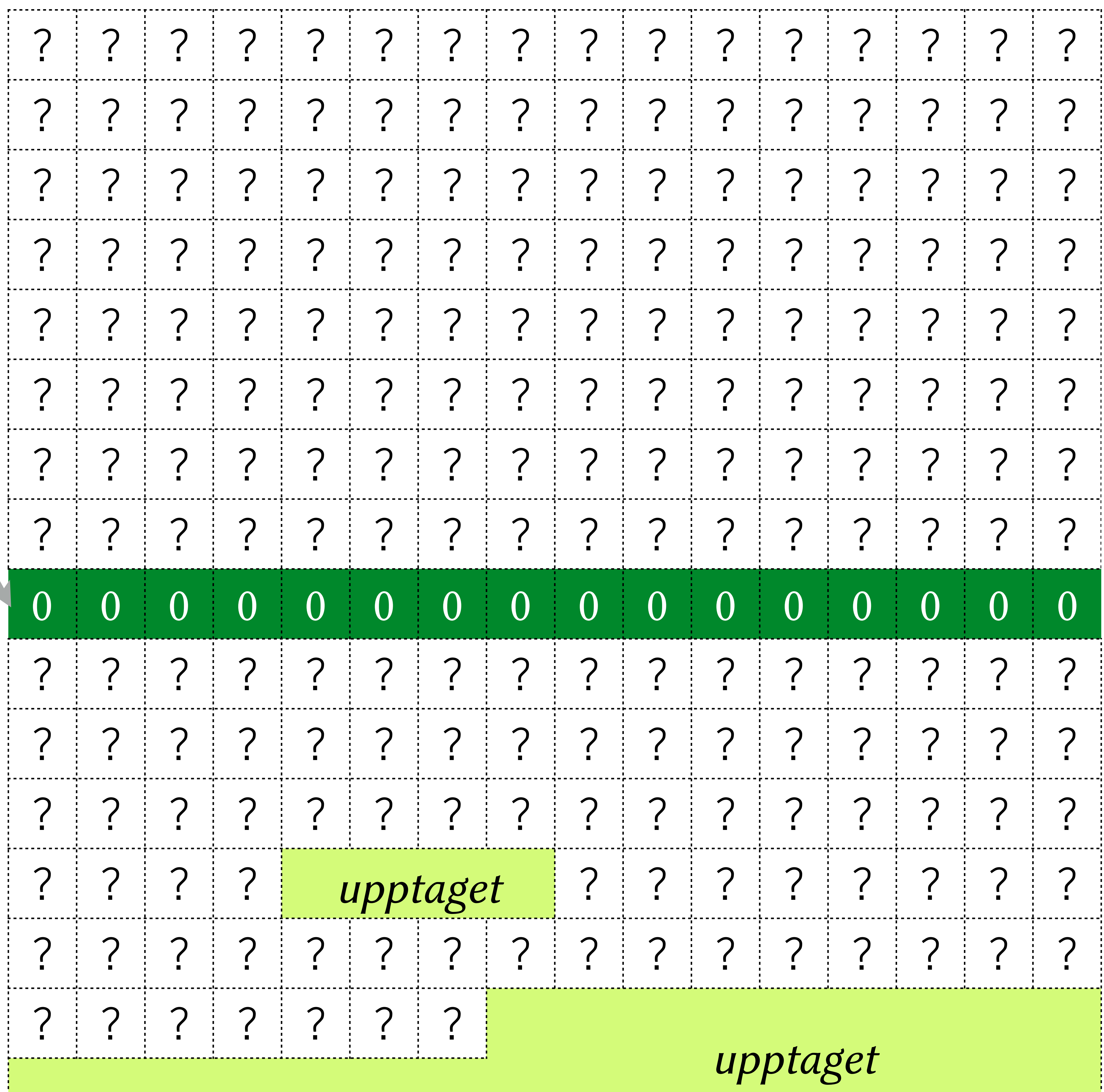
  `p = NULL;` — nu kan vi inte göra `free(p)` längre!

Program med invalid reads och writes och minnesläckage är inte acceptabla program, även om de brukar passera testerna!

C är svagt typat

- Vad är innebörden av nedanstående program? (Som kompilerar utan varningar.)

```
void *p = calloc(16, sizeof(char));

int *an_integer = p;
char *a_string = p;

strcpy(a_string, "42");

printf("Answer: %d\n", *an_integer);
```



C är svagt typat

- Vad är innebörden av nedanstående program? (Som kompilerar utan varningar.)

```
void *p = calloc(16, sizeof(char));

int *an_integer = p;
char *a_string = p;

strcpy(a_string, ”42”);

printf(”Answer: %d\n”, *an_integer);
```

”Answer: 12852”

p

?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	*upptaget*				?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	*upptaget*								

Väl mött på labben!