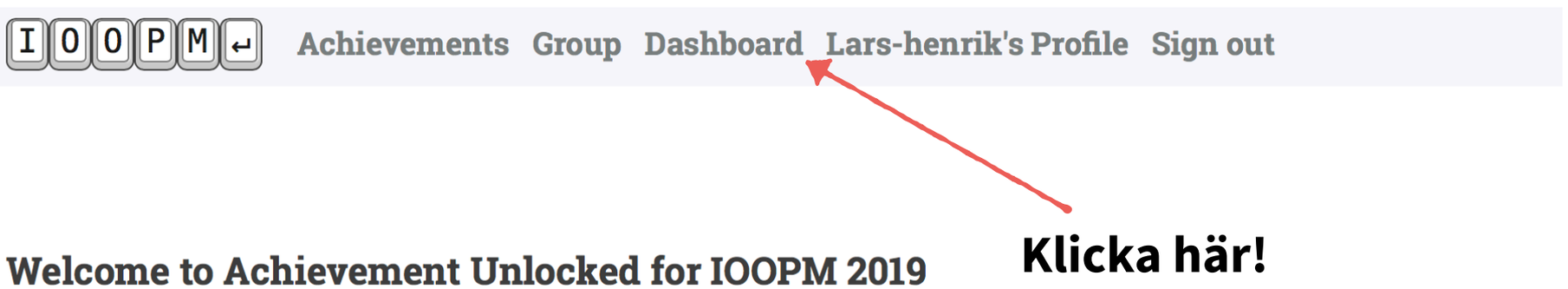
Föreläsning 5

Tobias Wrigstad

Lite om länkade strukturer — notation, insättning, frigöra…

Redovisning av mål (1)



Redovisning av mål (2)

IOOPM↵ Achievements Group Dashboard Lars-henrik's Profile Sign out

- Pending demonstration
- All demonstrations
- My exam results
- My Progress

Request a demonstration

* Partner

Goals to demonstrate

Collapse Passed To be passed Grade 3 Grade 4 Grade 5

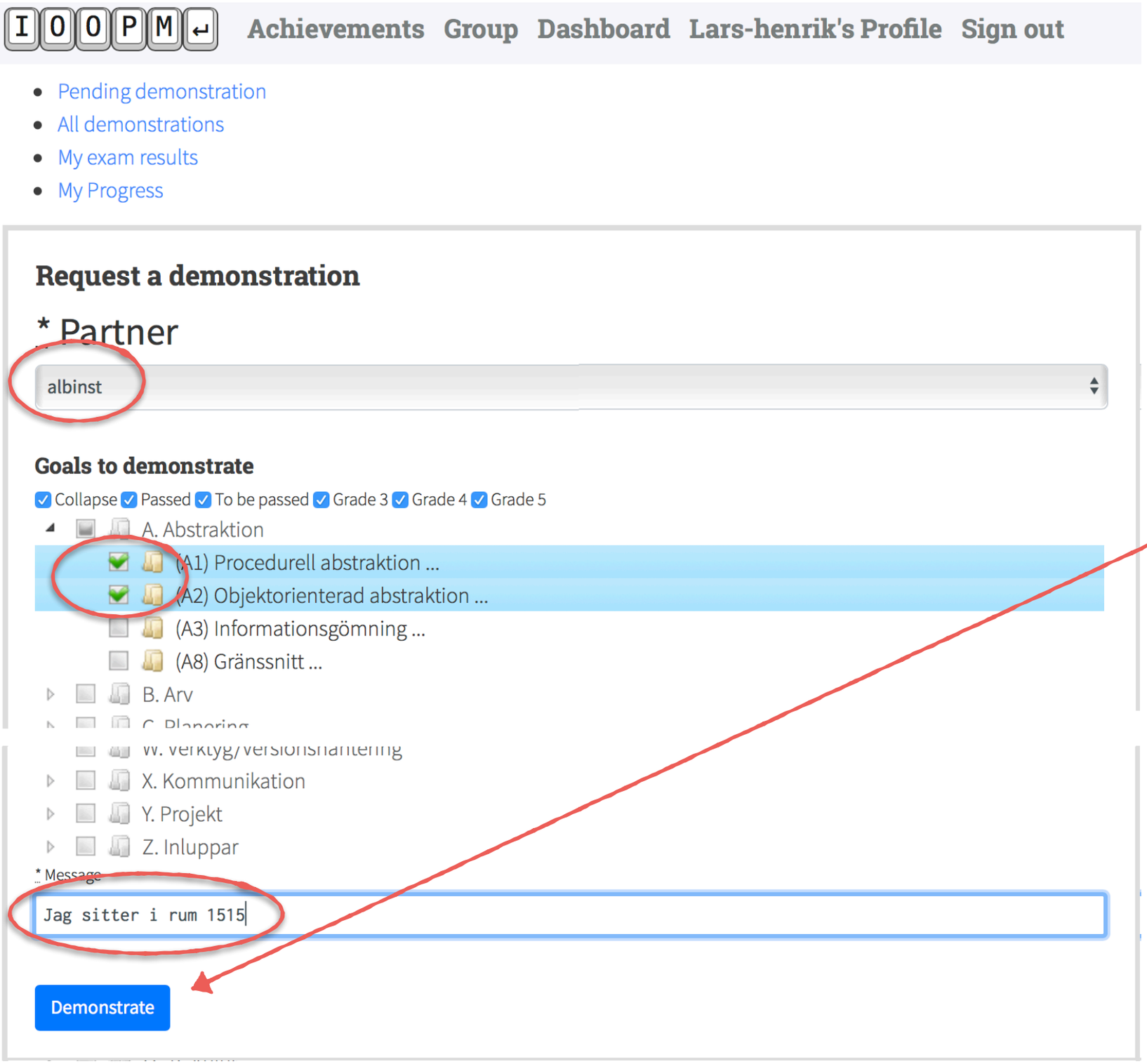
- A. Abstraktion
- B. Arv
- C. Planering
- W. Verktyg/versionshantering
- X. Kommunikation
- Y. Projekt
- Z. Inluppar

* Message

I am doing this demo at room xxx

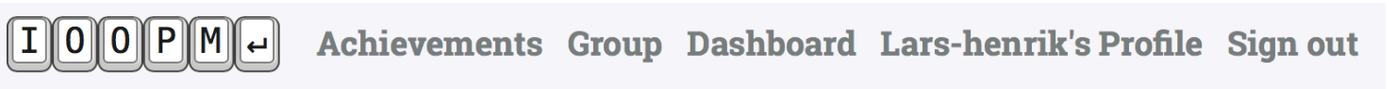
Demonstrate

Redovisning av mål (3)

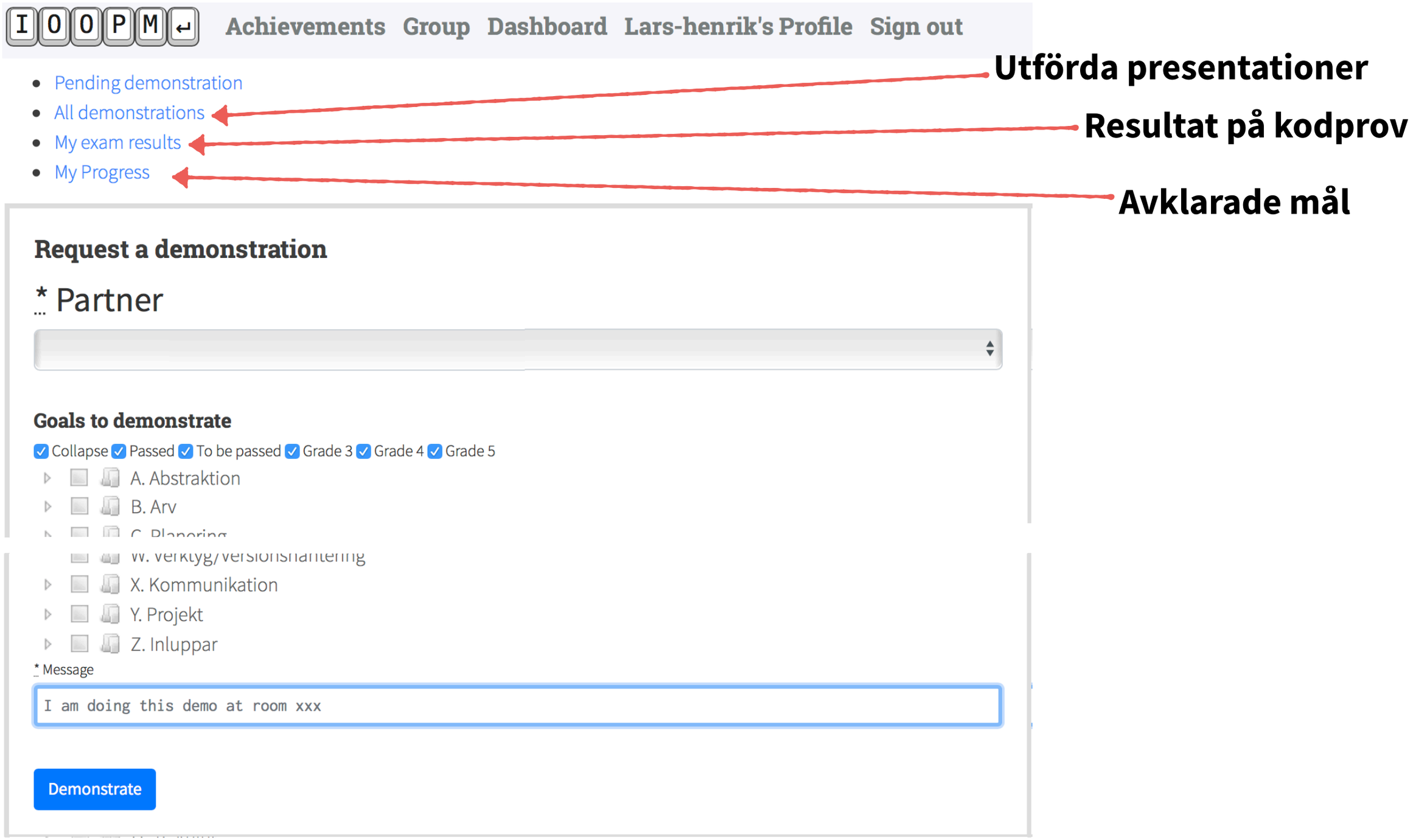


Klicka här för att be om att redovisa!

Redovisning av mål (4)



Kolla status



Utförda presentationer

Resultat på kodprov

Avklarade mål

Utförda presentationer

Achievements Group Dashboard Lars-henrik's Profile Sign out

A Complete Account of your Demonstrations

Color coding: Passed Not passed Not passed with push-back

1. Demonstration created at 15 Sep 15:12 (Ablin Sjertna and Lars-henrik Eriksson)
   Examined by Lhe.. at 15 Sep 15:17
   - (A1) Procedurell abstraktion ...
   - (A2) Objektorienterad abstraktion ...

Sammanställning av avklarade mål

IOOPM↵ Achievements Group Dashboard Lars-henrik's Profile Sign out

Lars-henrik.Eriksson.

Progress to Grade 3

2%

Detailed Transcript

Name	Status
A1	G

L3 passed

Name	Status
A2	U
B4	U
B5	U
C7	U
D9	U
E10	U
E11	U
F13	U
G15	U
G16	U

Name	Status

L4 passed

Name	Status
A3	U
B6	U
A8	U
E12	U
F14	U
G17	U
I24	U
I25	U
J28	U
K31	U

Name	Status

L5 passed

Name	Status
H18	U
H21	U
J29	U
K32	U
O44	U
P48	U
Q51	U
X62	U

L5 remaining

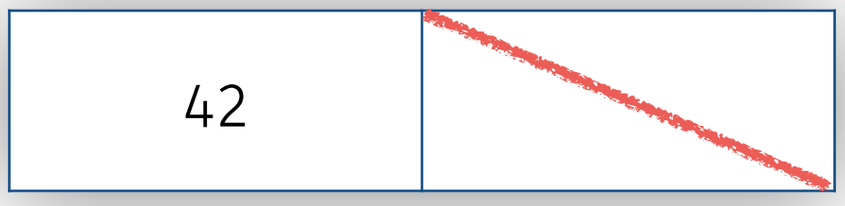
Länkade listor: notation

Structen…

```
typedef struct link link_t;

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

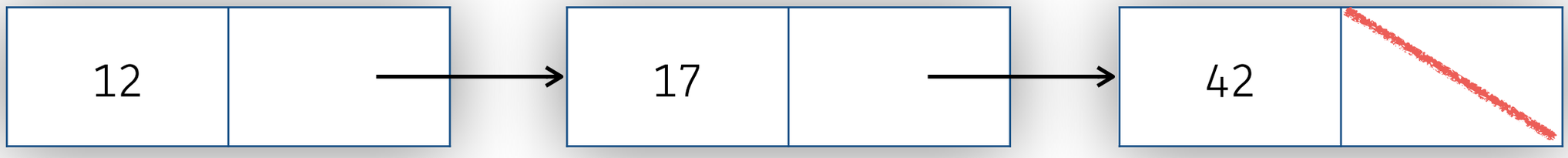
…ritar vi så här

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



strecket betyder NULL — ibland skriver vi ut NULL

Länkarna i en länkad lista ritar vi så här




```
typedef struct link link_t;

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

Struktarna som bygger upp en länkad lista

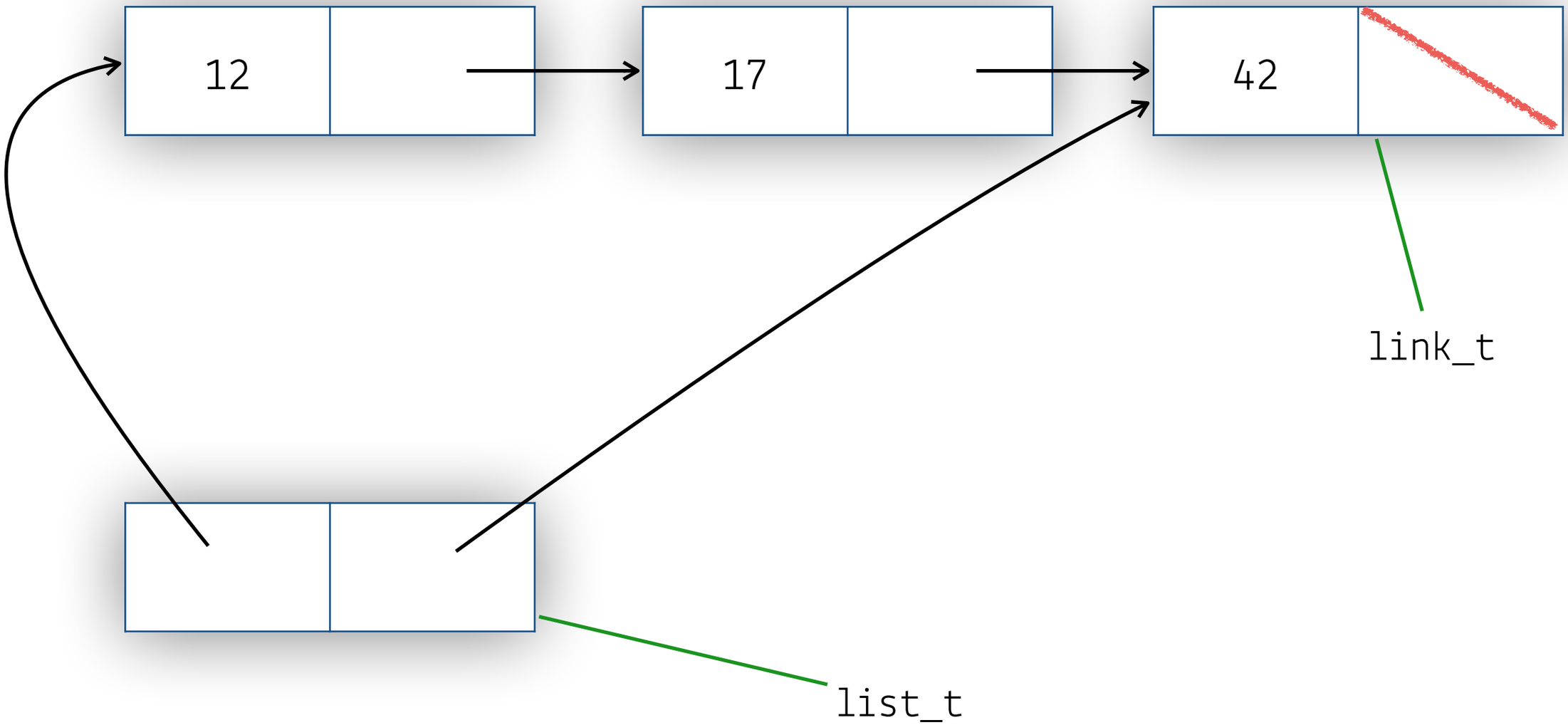
”Själva listan”

```
typedef struct list list_t;

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

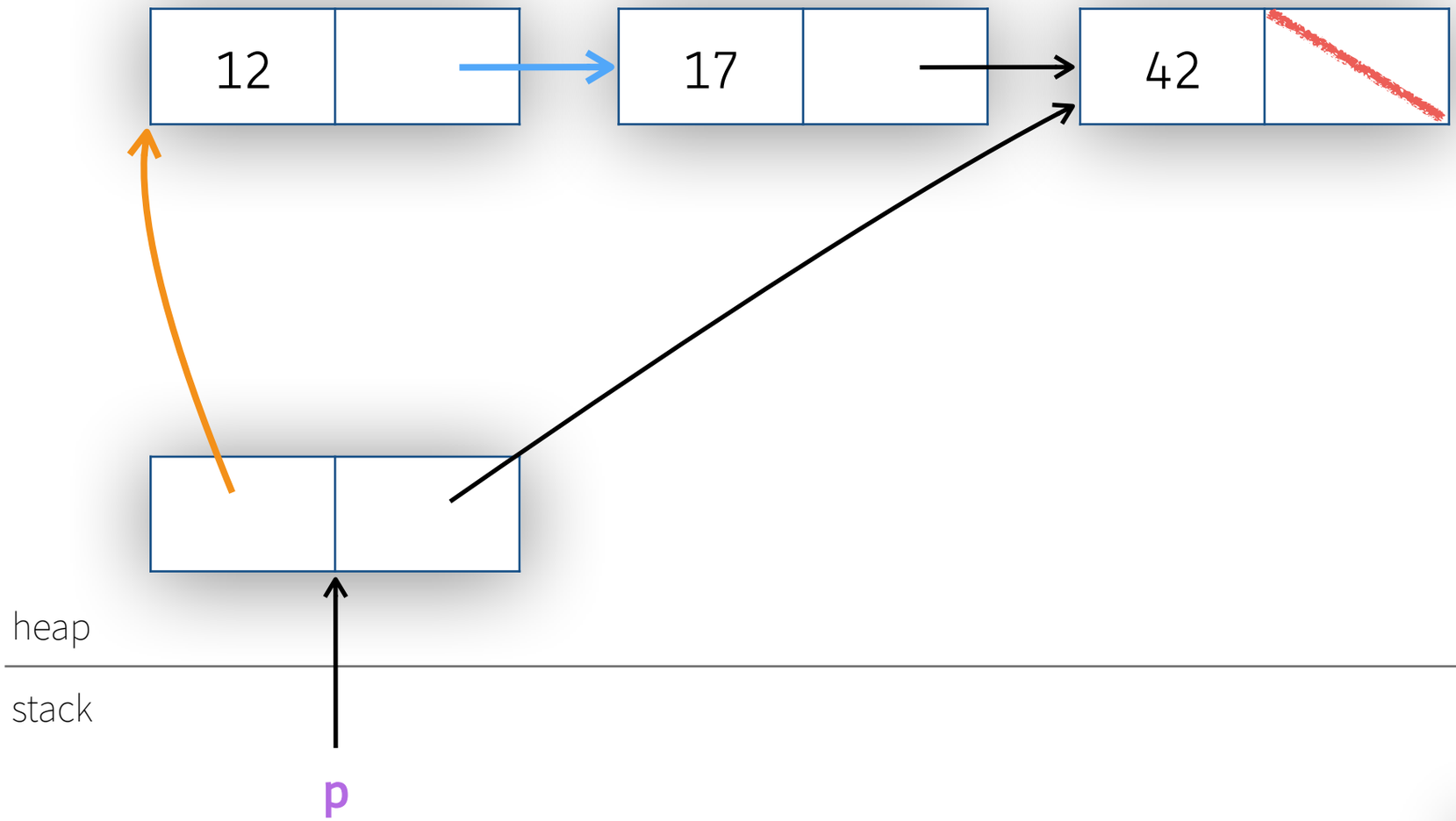
Länkarna

```
typedef struct link link_t;

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

En komplett länkad lista

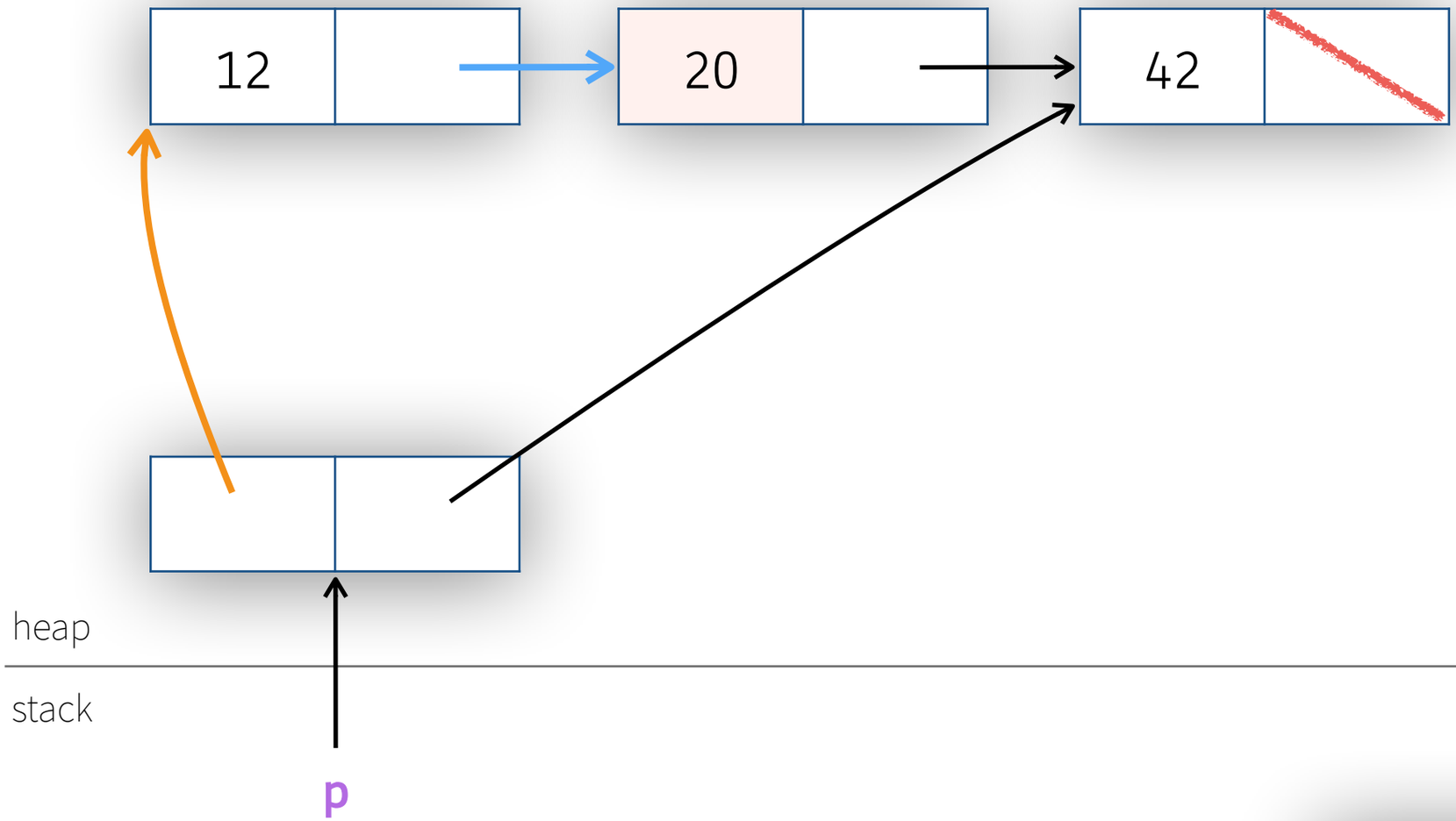


De olika delarna och hur man kommer åt dem



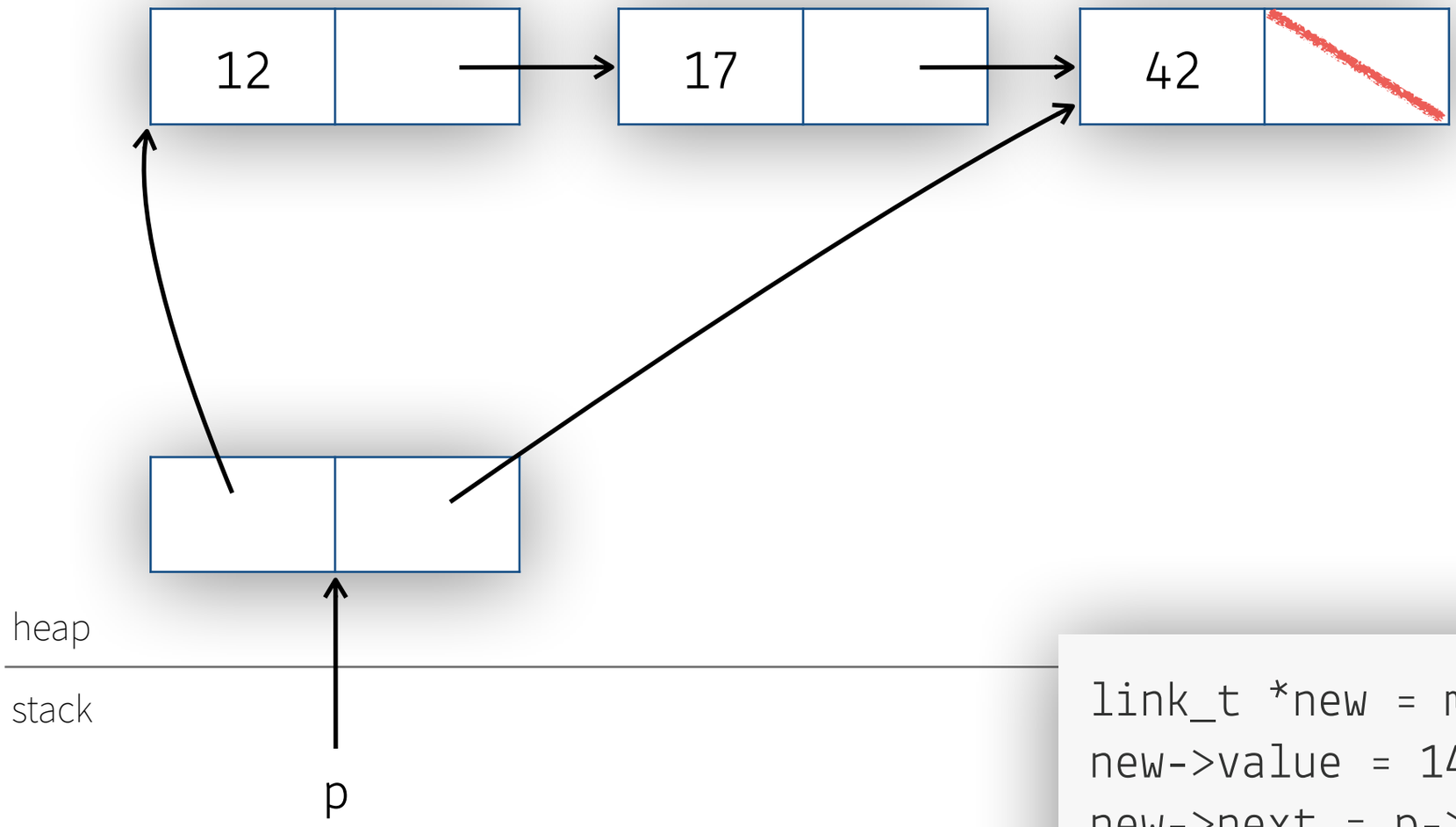

```
p->first->next
```

De olika delarna och hur man kommer åt dem



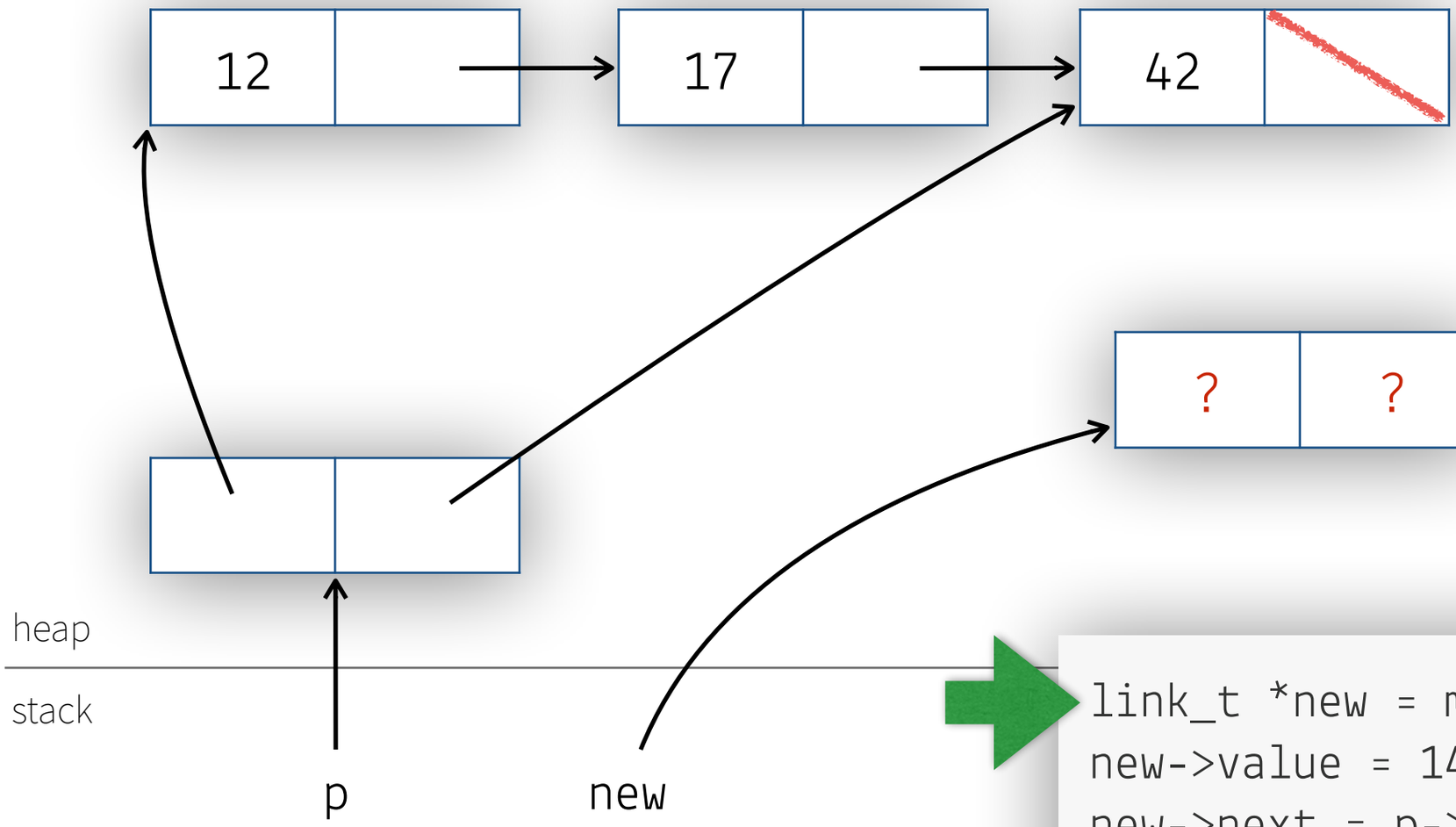

```
p->first->next->value == 20;
```

Lägg till ett element i listan



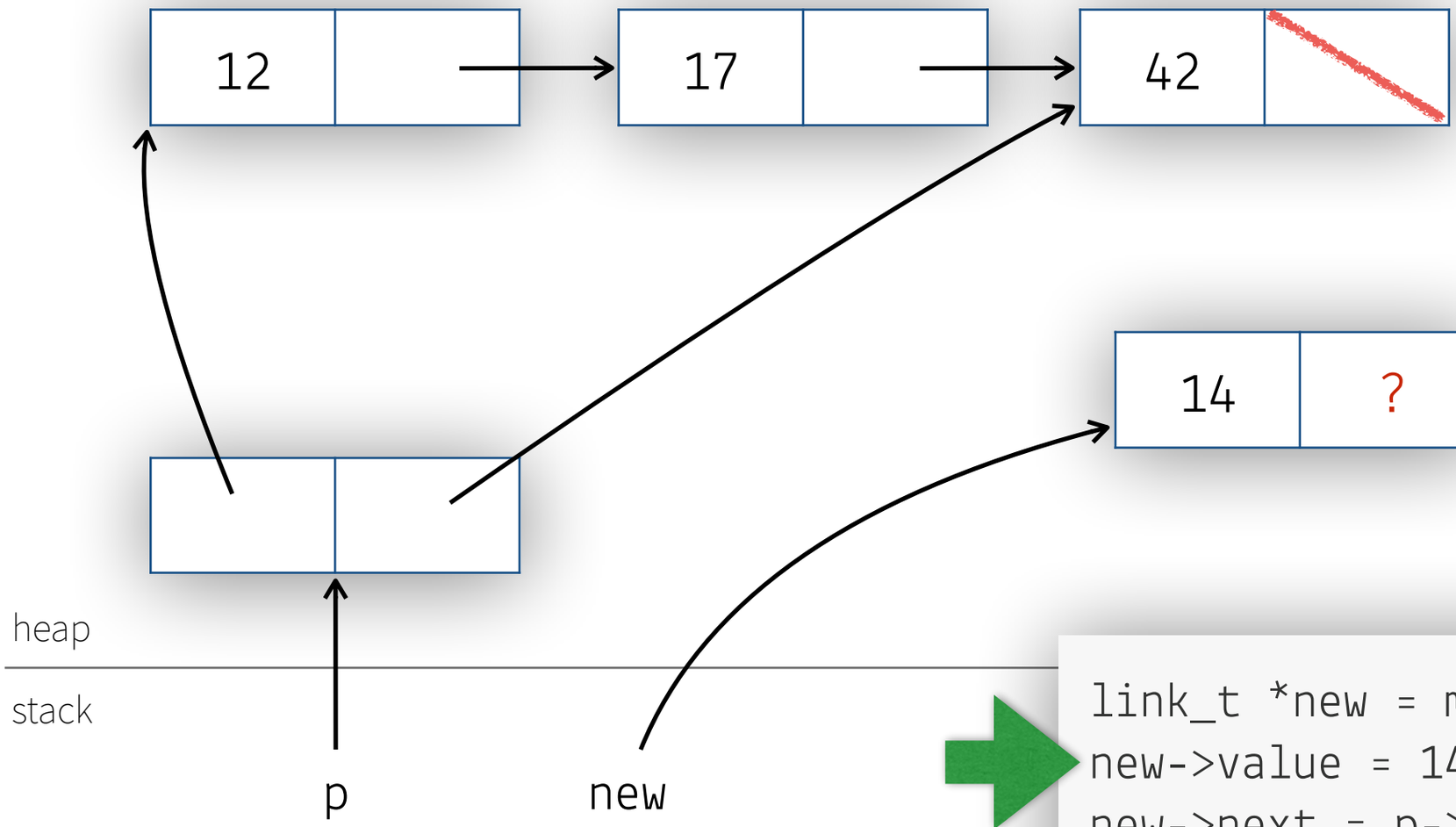

```
link_t *new = malloc(sizeof(link_t));
new->value = 14;
new->next = p->first->next;
p->first->next = new;
```

Lägg till ett element i listan [steg 1]



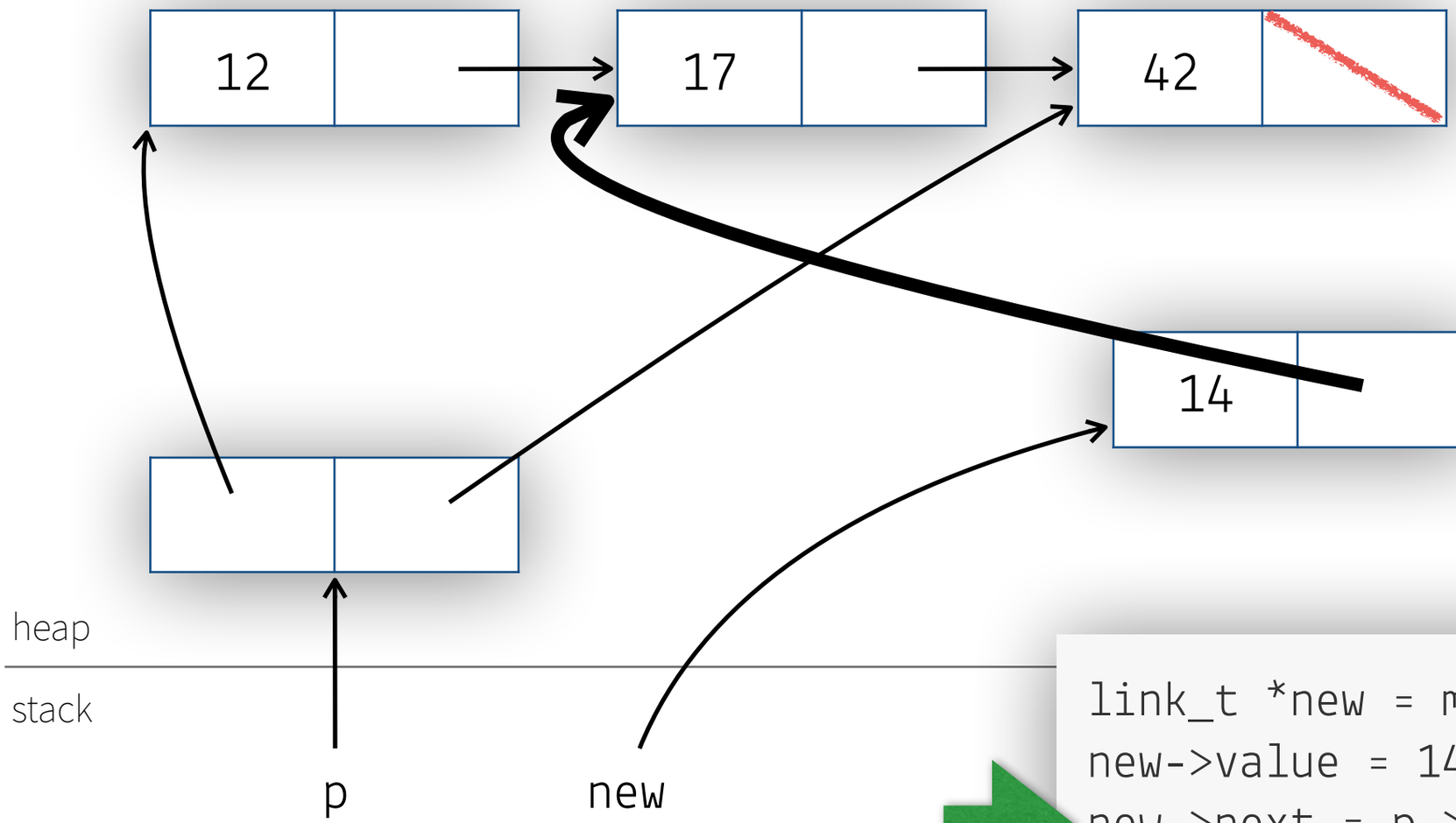

```
link_t *new = malloc(sizeof(link_t));
new->value = 14;
new->next = p->first->next;
p->first->next = new;
```

Lägg till ett element i listan [steg 2]



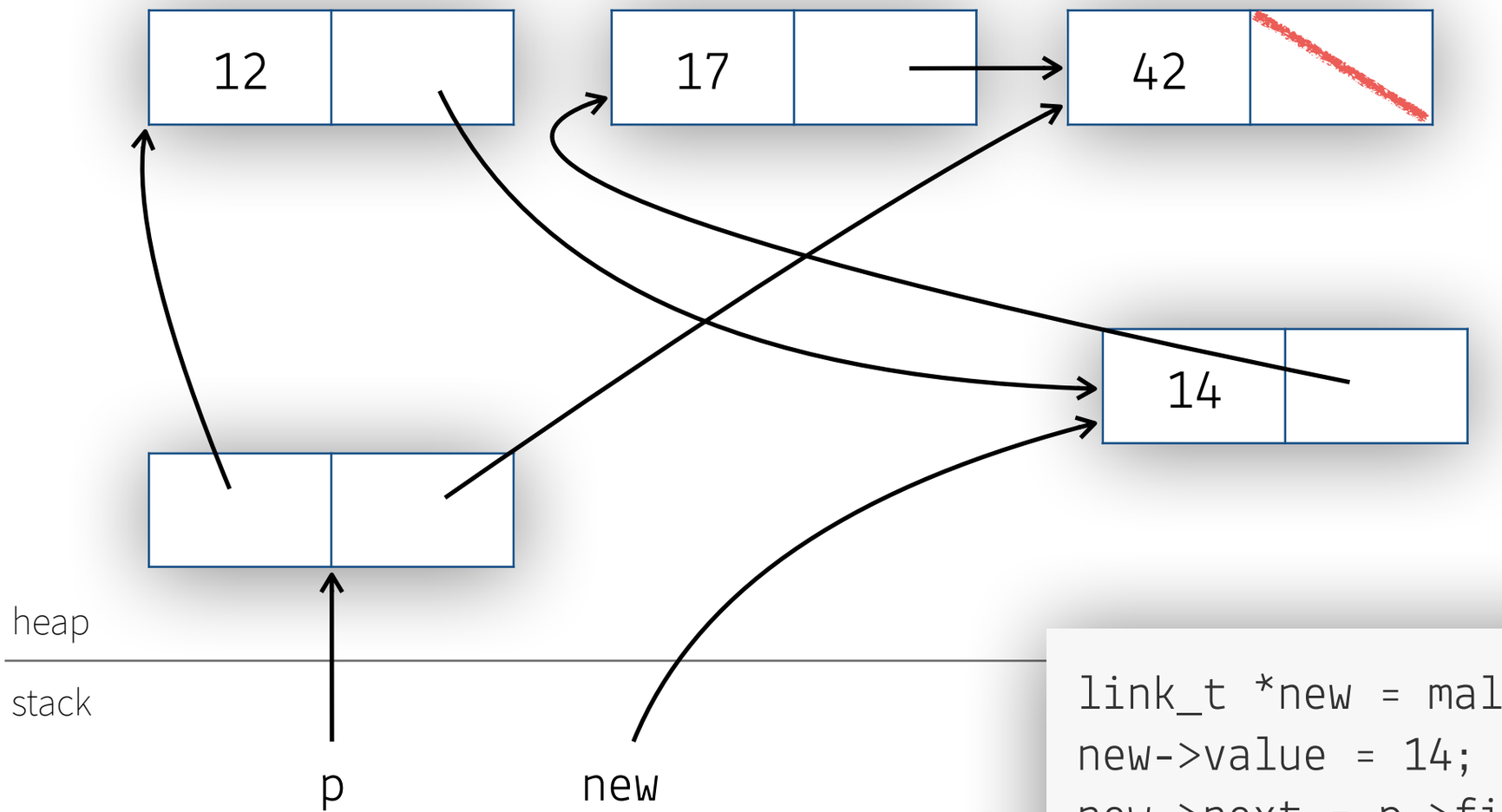

```
link_t *new = malloc(sizeof(link_t));
new->value = 14;
new->next = p->first->next;
p->first->next = new;
```

Lägg till ett element i listan [steg 3]



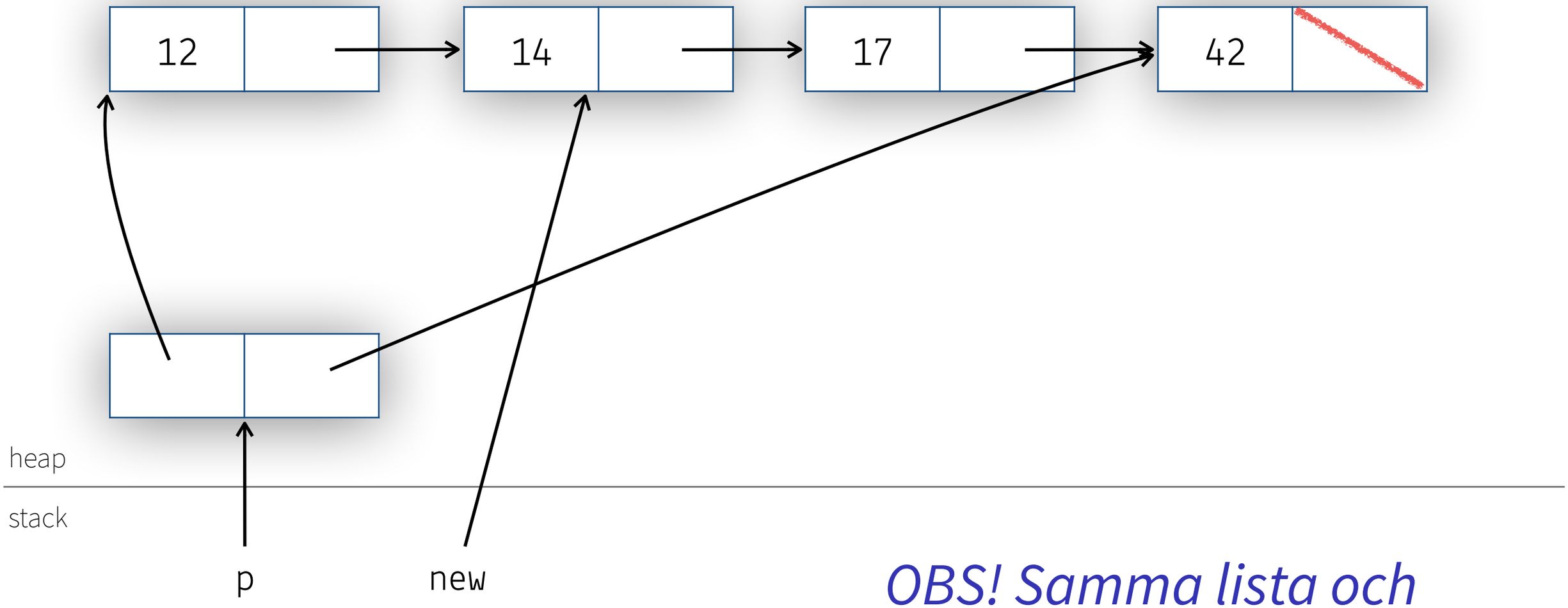

```
link_t *new = malloc(sizeof(link_t));
new->value = 14;
new->next = p->first->next;
p->first->next = new;
```

Lägg till ett element i listan [steg 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

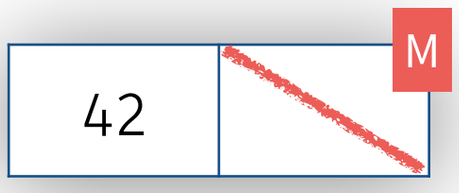
  Utgå från programmet nedan som loopar genom kommandoradsargumenten och skriver ut dem (utan en länkad lista)

- 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

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

Att ta bort ett element ur en lista.

- Ny notation:



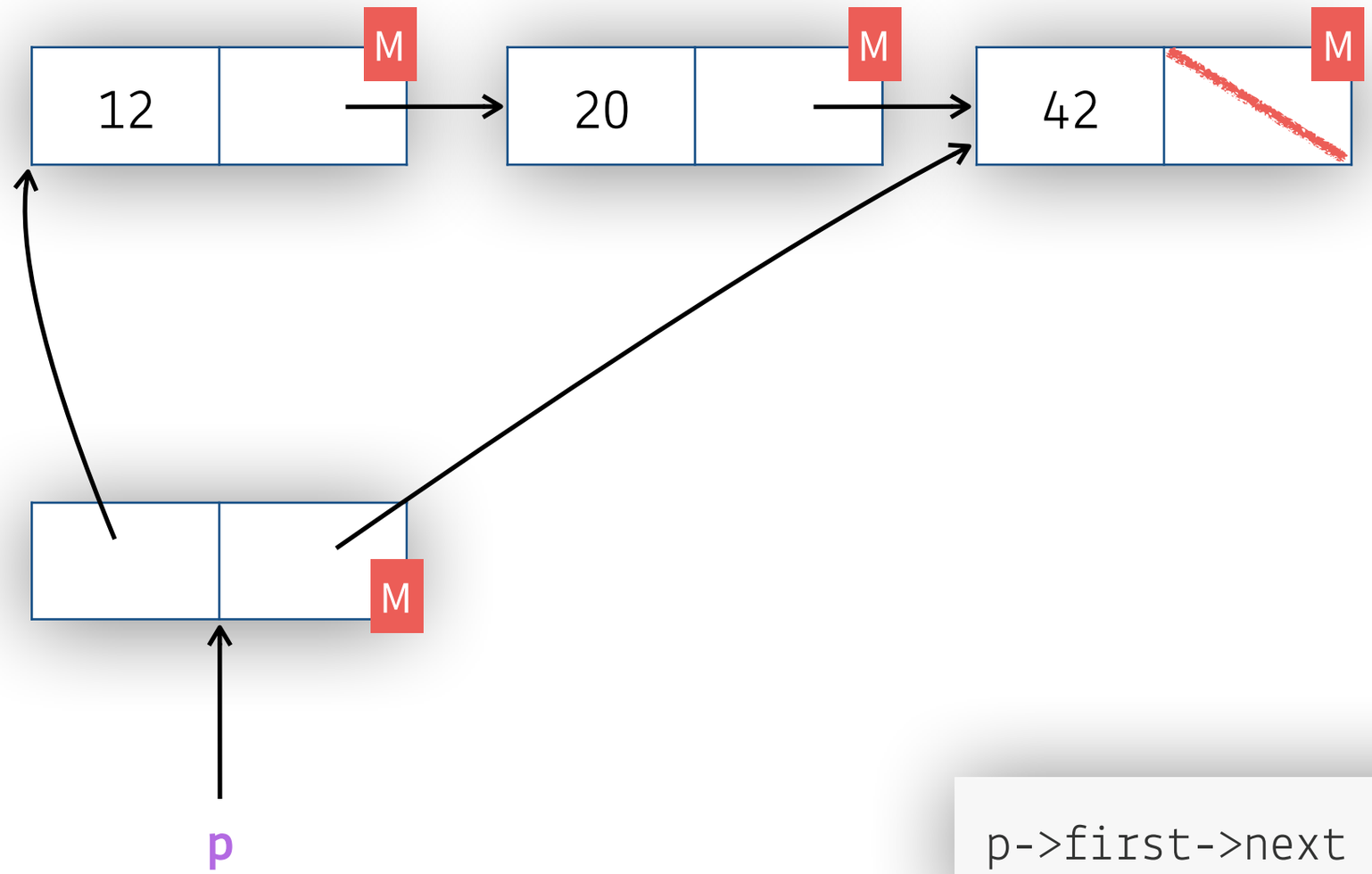
M betyder att `malloc` anser att strukten används

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

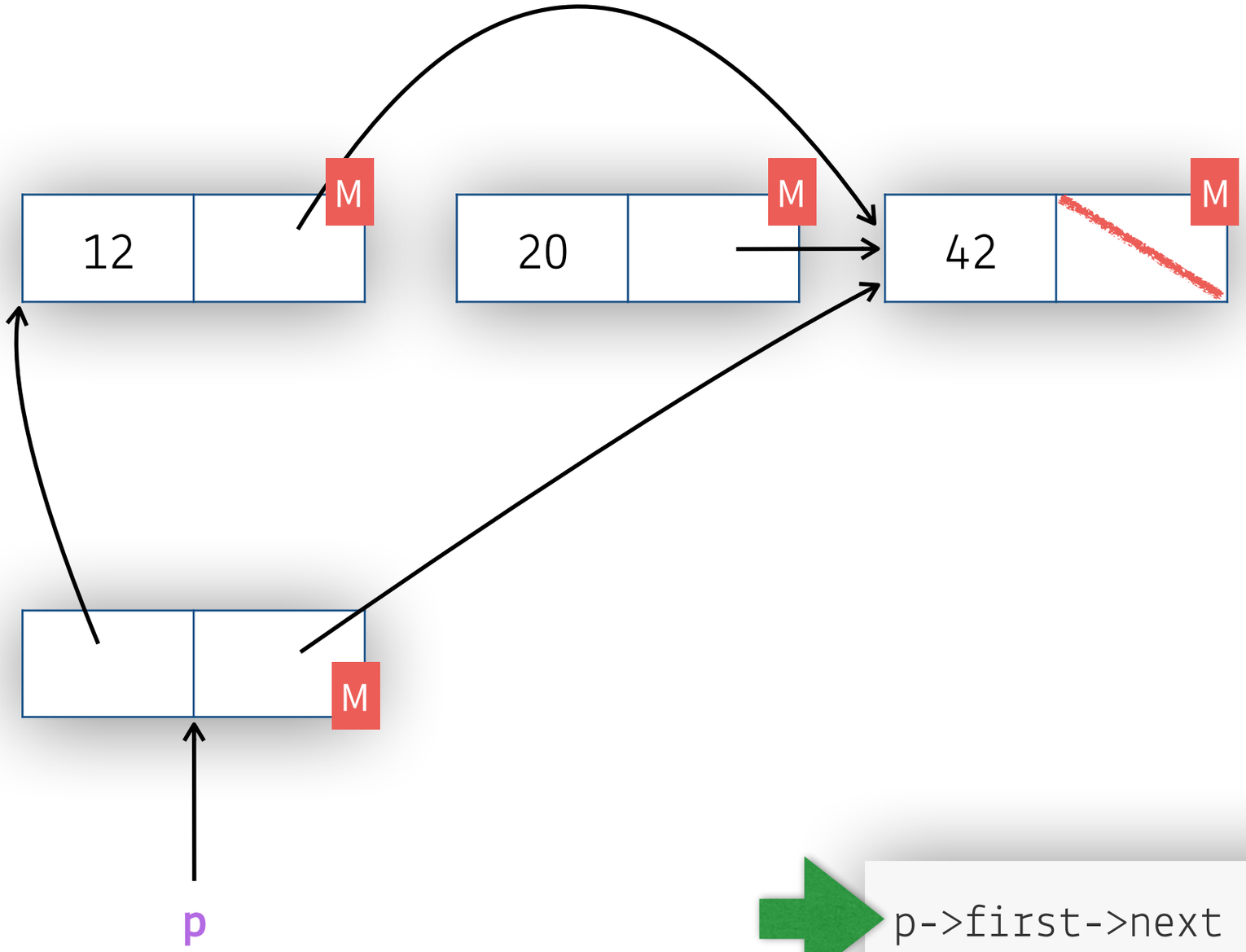
Röda värden i strukten betyder att de kanske inte finns längre

Tag bort ett element ur listan — försök 1 (1/2) [OBS! Fel]



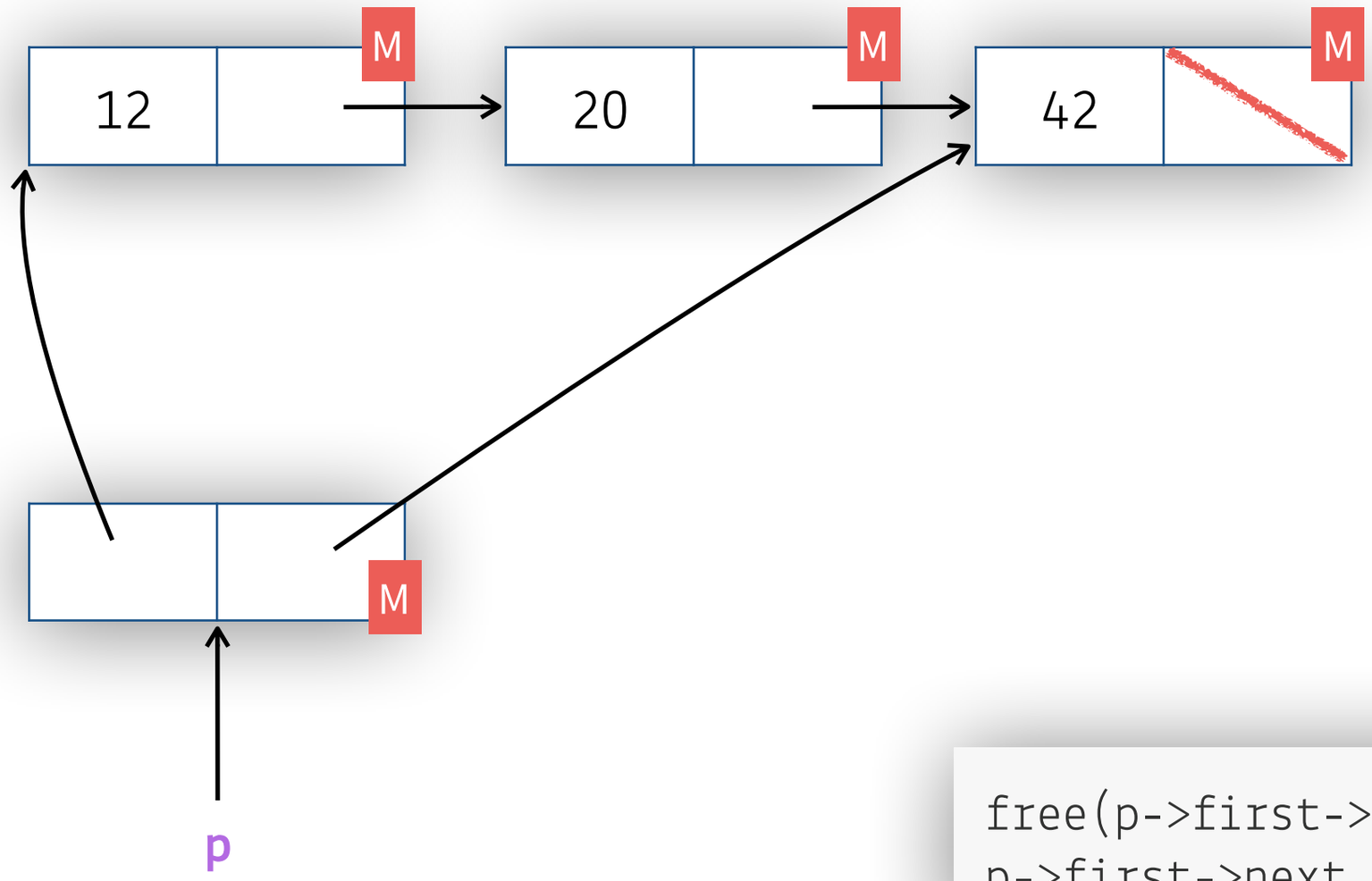

```
p->first->next = p->first->next->next;
```

Tag bort ett element ur listan — försök 1 (2/2) [OBS! Fel]



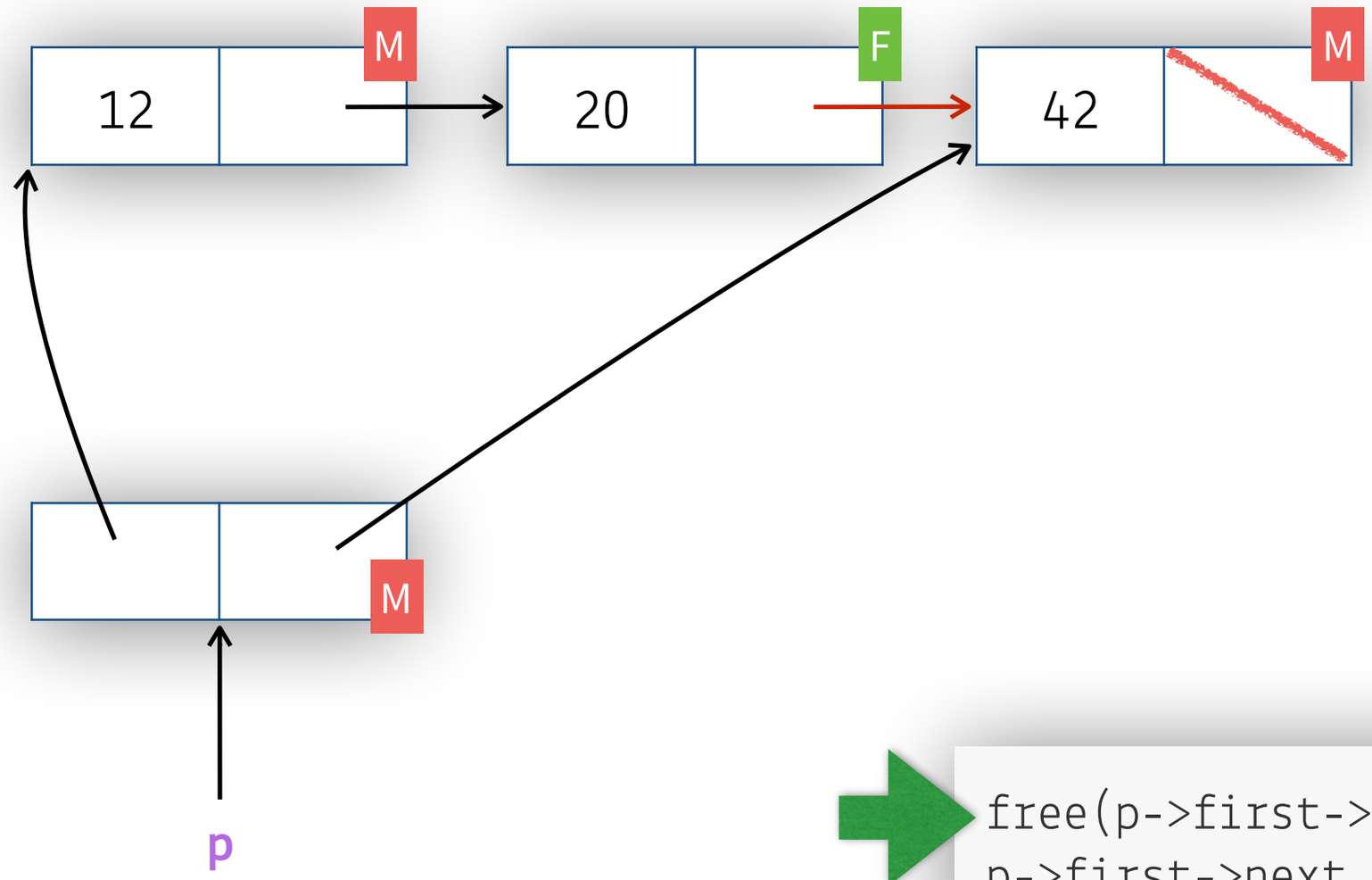

```
p->first->next = p->first->next->next;
```

Tag bort ett element ur listan — försök 2 (1/3) [OBS! Fel]



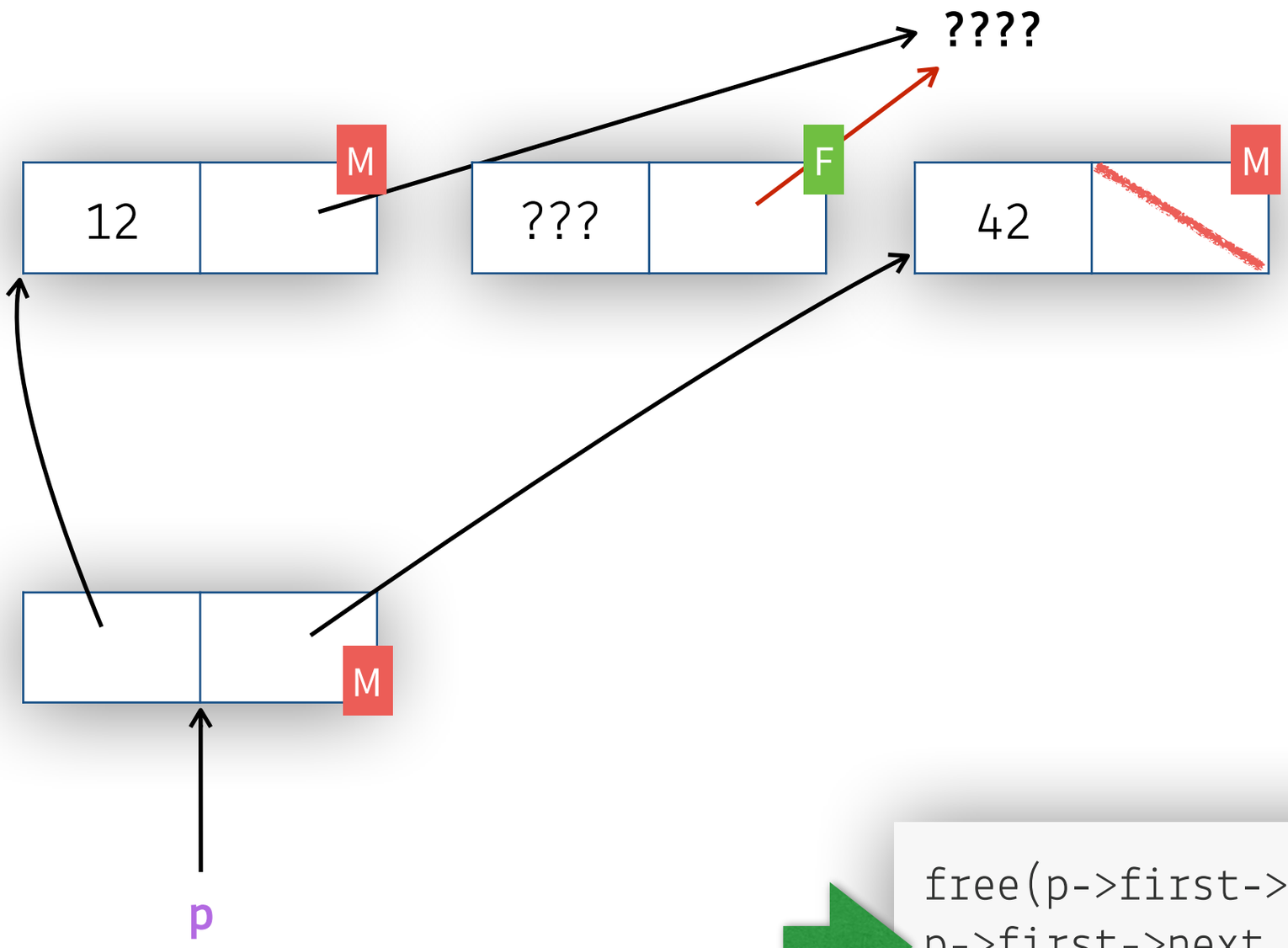

```
free(p->first->next);
p->first->next = p->first->next->next;
```

Tag bort ett element ur listan — försök 2 (2/3) [OBS! Fel]



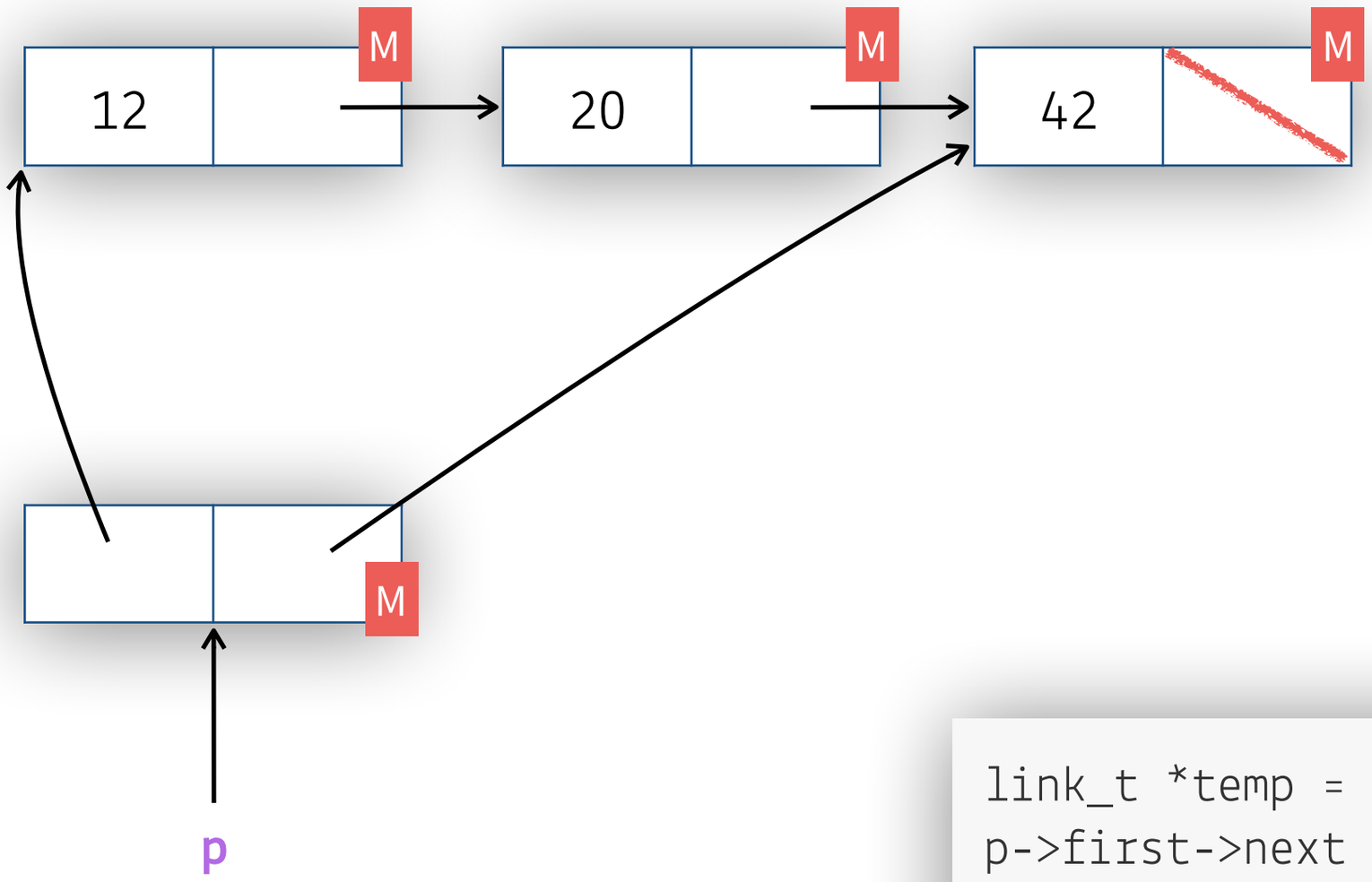

```
free(p->first->next);
p->first->next = p->first->next->next;
```

Tag bort ett element ur listan — försök 2 (3/3) [OBS! Fel]



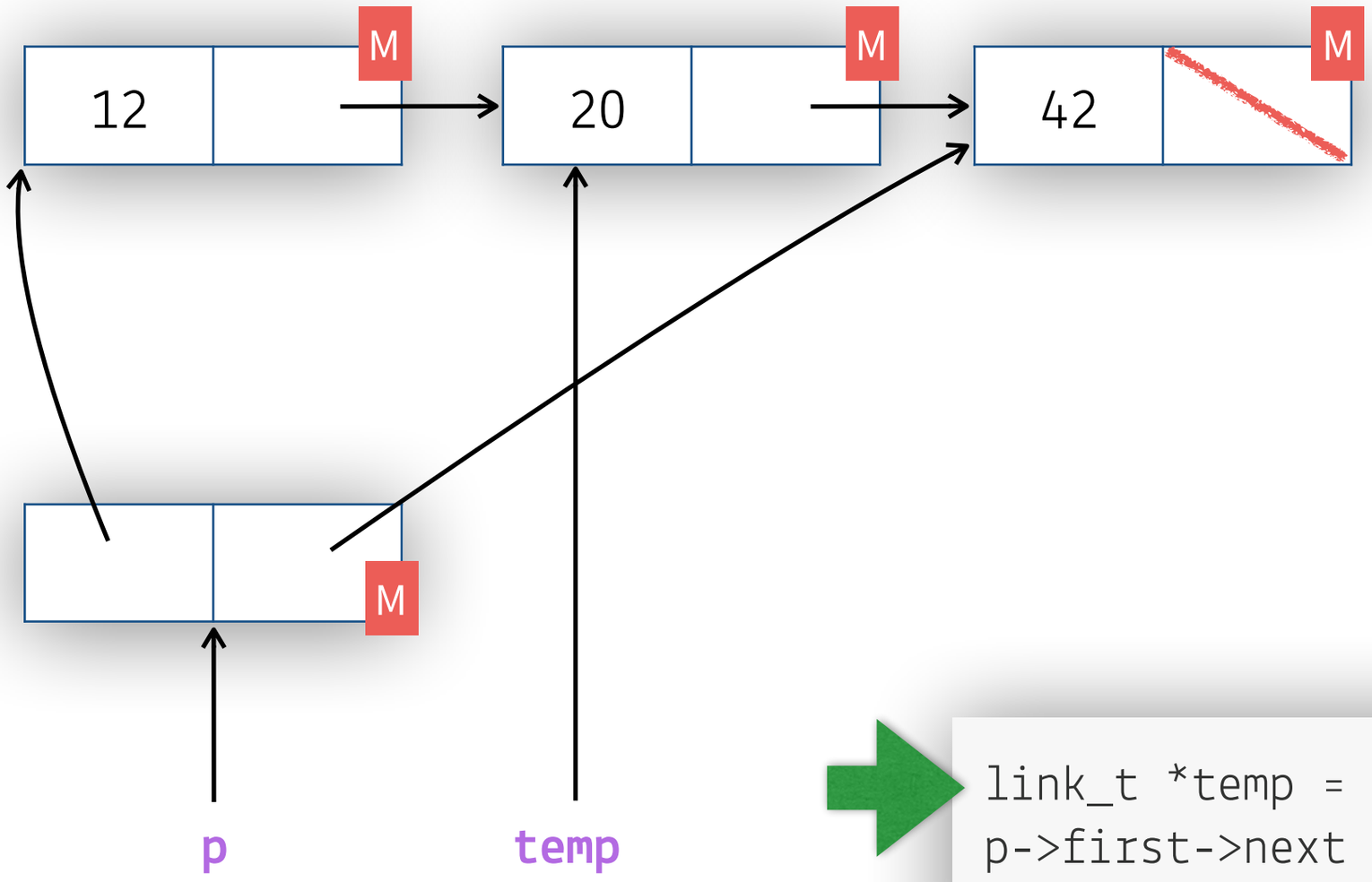

```
free(p->first->next);
p->first->next = p->first->next->next;
```

Tag bort ett element ur listan — försök 3 (1/4) [OBS! Rätt]



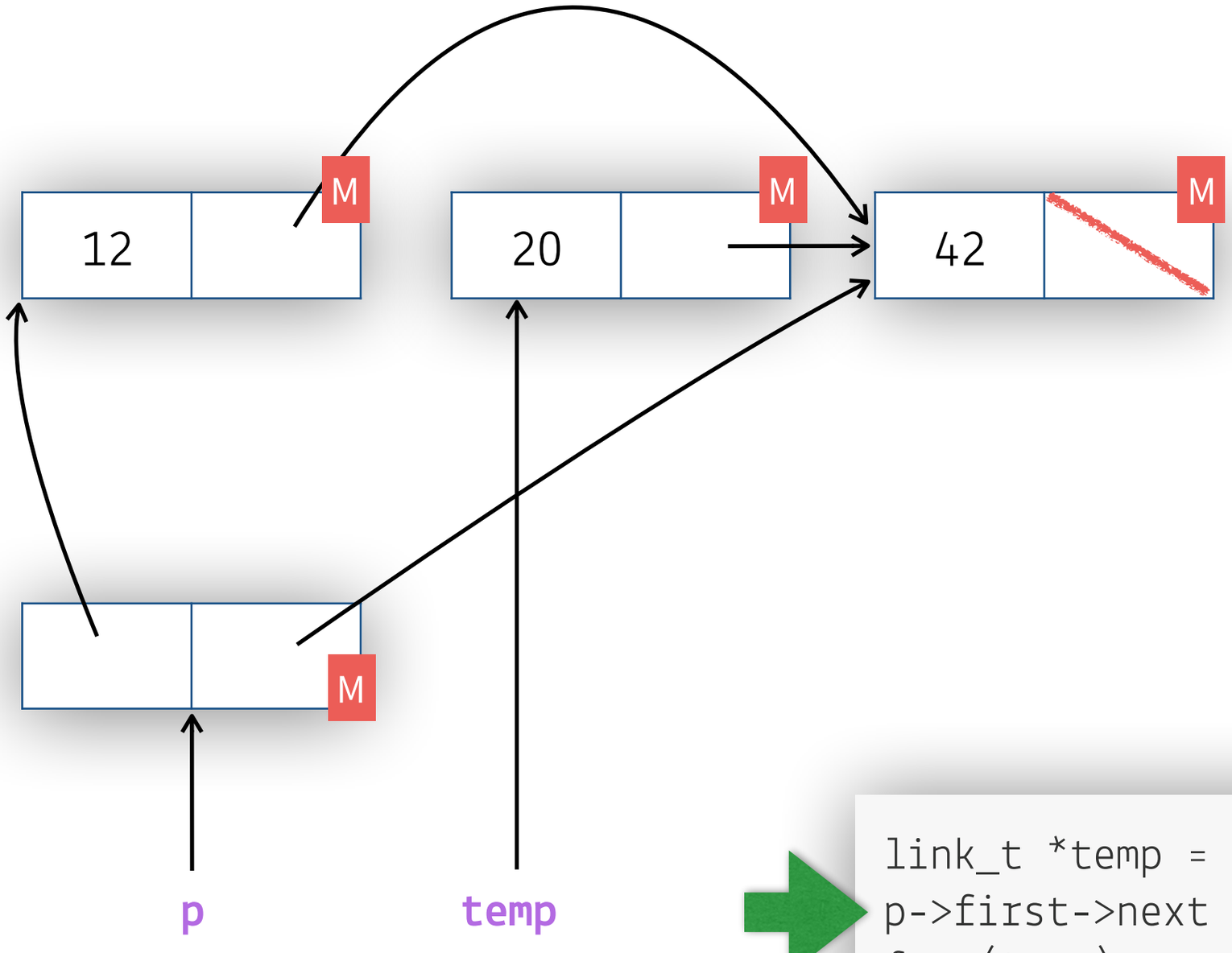

```
link_t *temp = p->first->next;
p->first->next = temp->next;
free(temp);
```

Tag bort ett element ur listan — försök 3 (2/4) [OBS! Rätt]



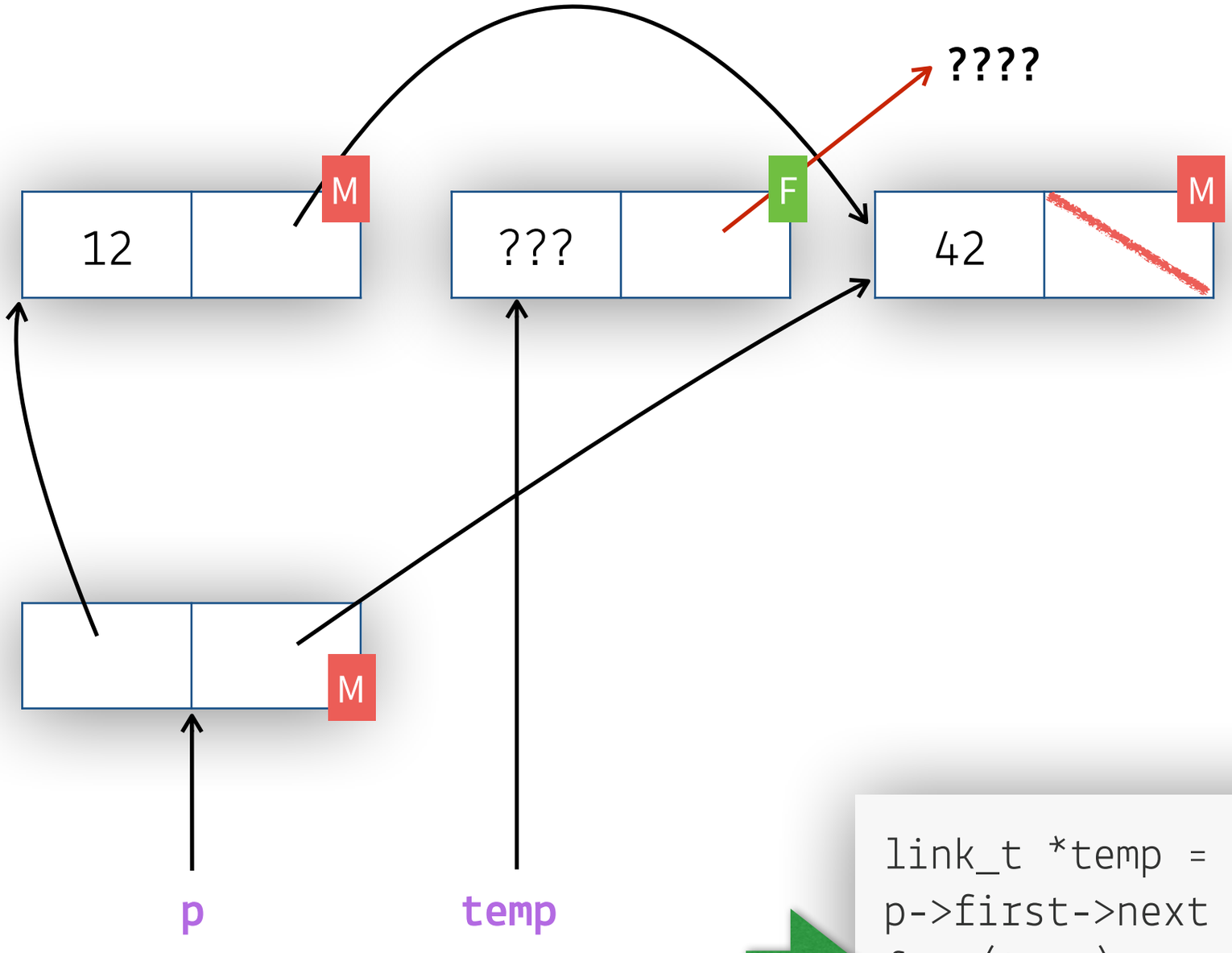

```
link_t *temp = p->first->next;
p->first->next = temp->next;
free(temp);
```

Tag bort ett element ur listan — försök 3 (3/4) [OBS! Rätt]



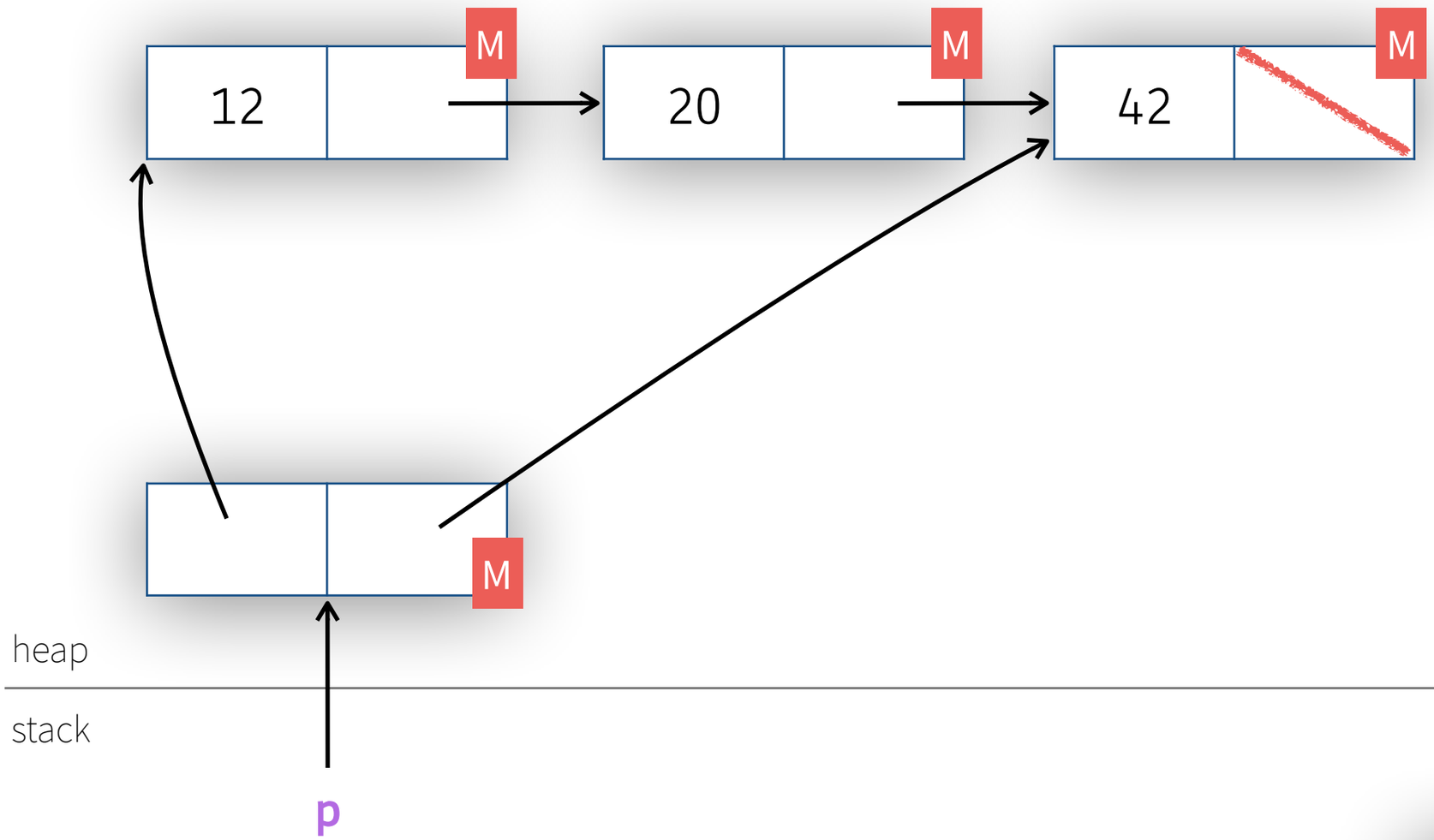

```
link_t *temp = p->first->next;
p->first->next = temp->next;
free(temp);
```

Tag bort ett element ur listan — försök 3 (4/4) [OBS! Rätt]



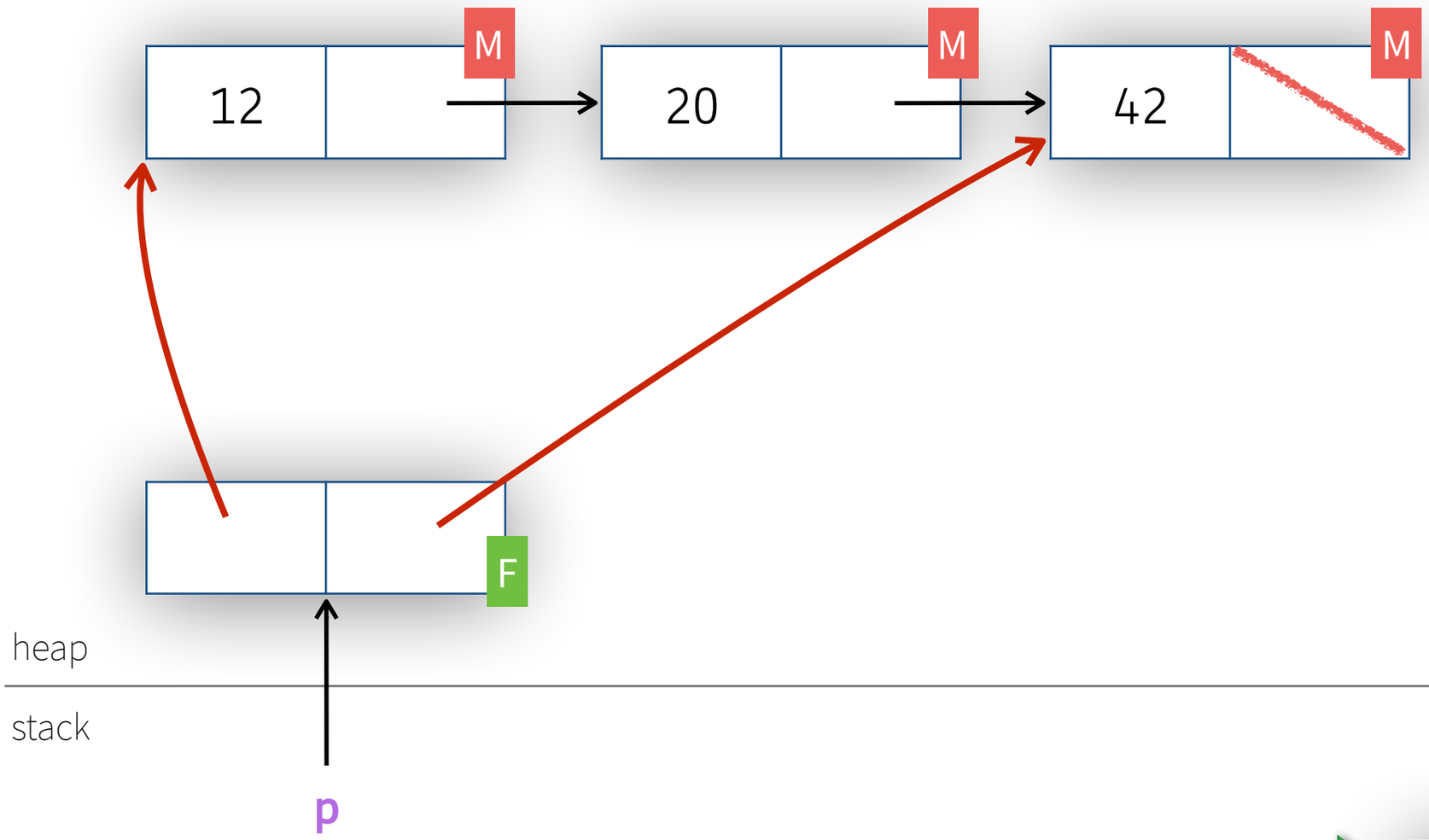

```
link_t *temp = p->first->next;
p->first->next = temp->next;
free(temp);
```

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




```
free(p);
```

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




```
free(p);
```

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:

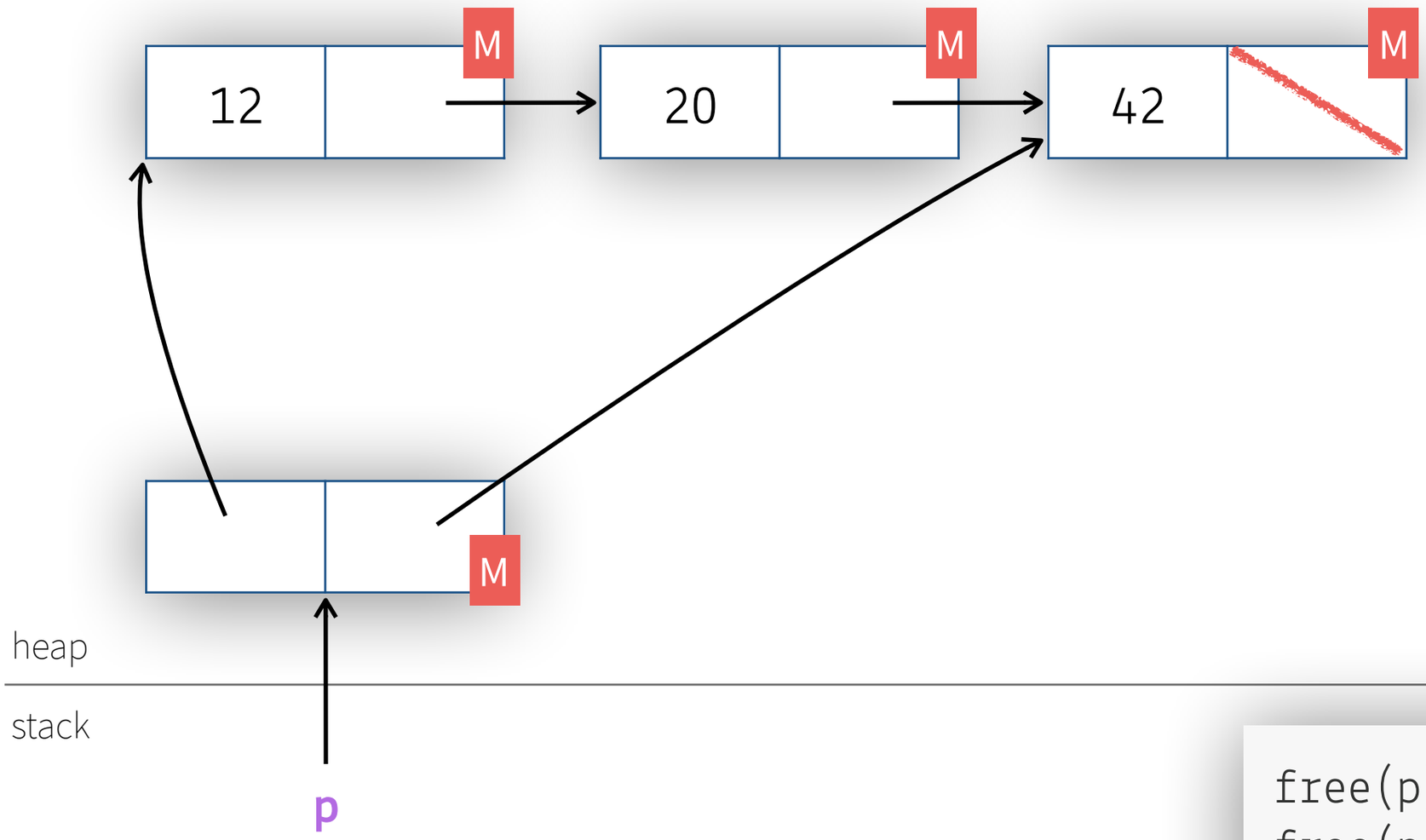
  efter att du gjort `free(p)` får inte inte använda p igen förrän du har pekat om `p` att peka på något data som du vet är validt

- Konsekvens av `free(p)` blir därför

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

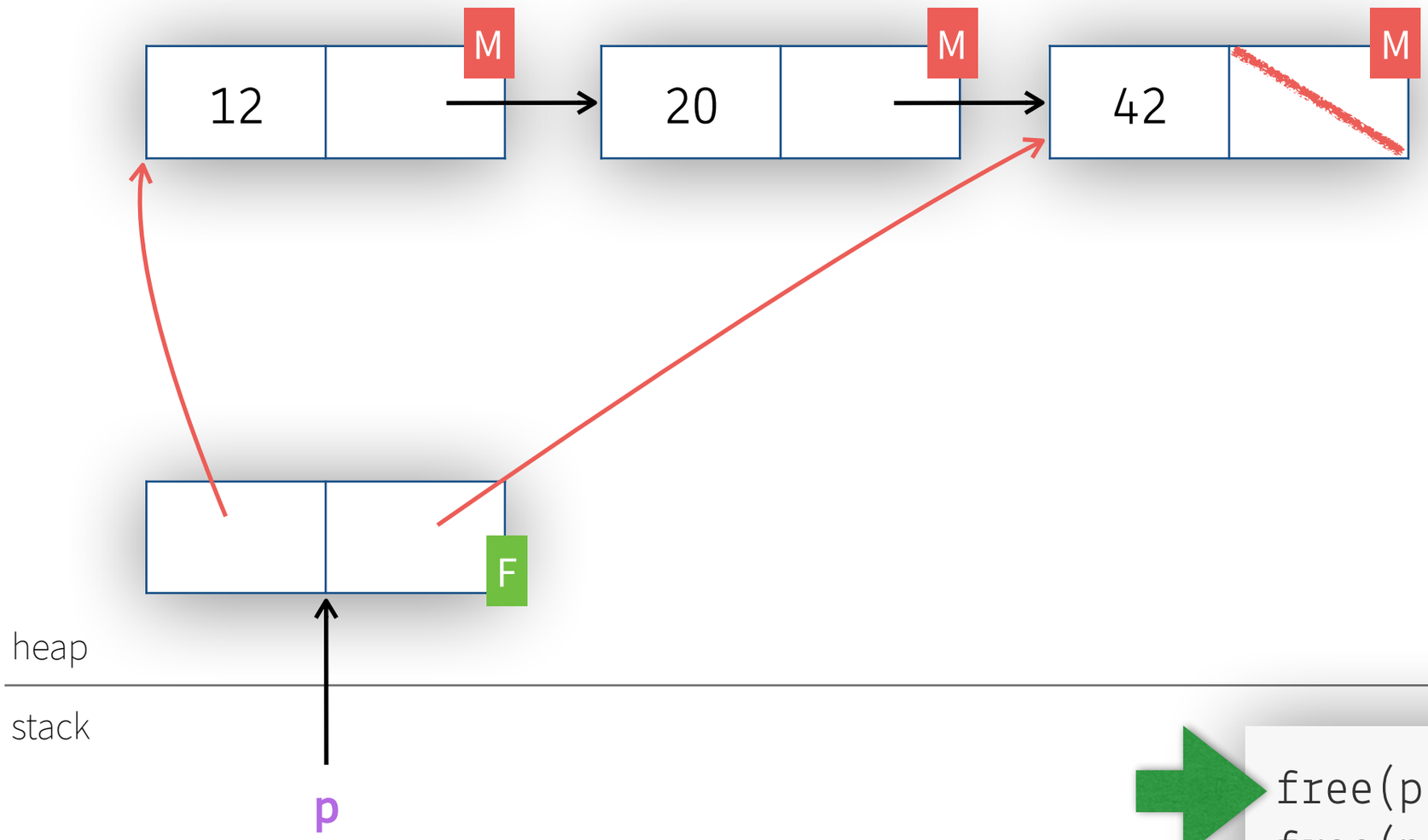
  …vilket i förlängningen betyder att vi läcker minne

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



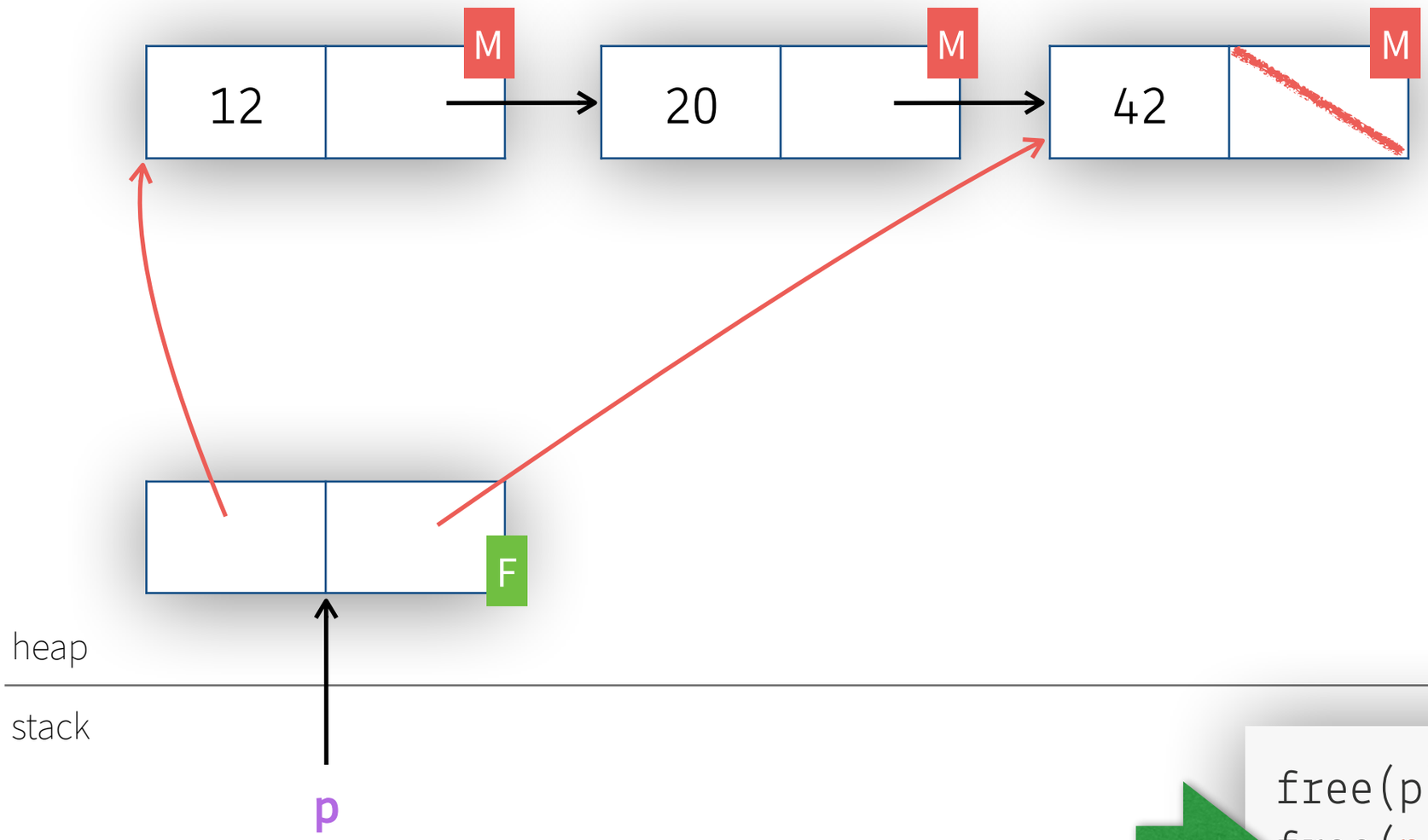

```
free(p);
free(p->first);
free(p->first->next);
free(p->first->next->next);
```

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




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

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);`

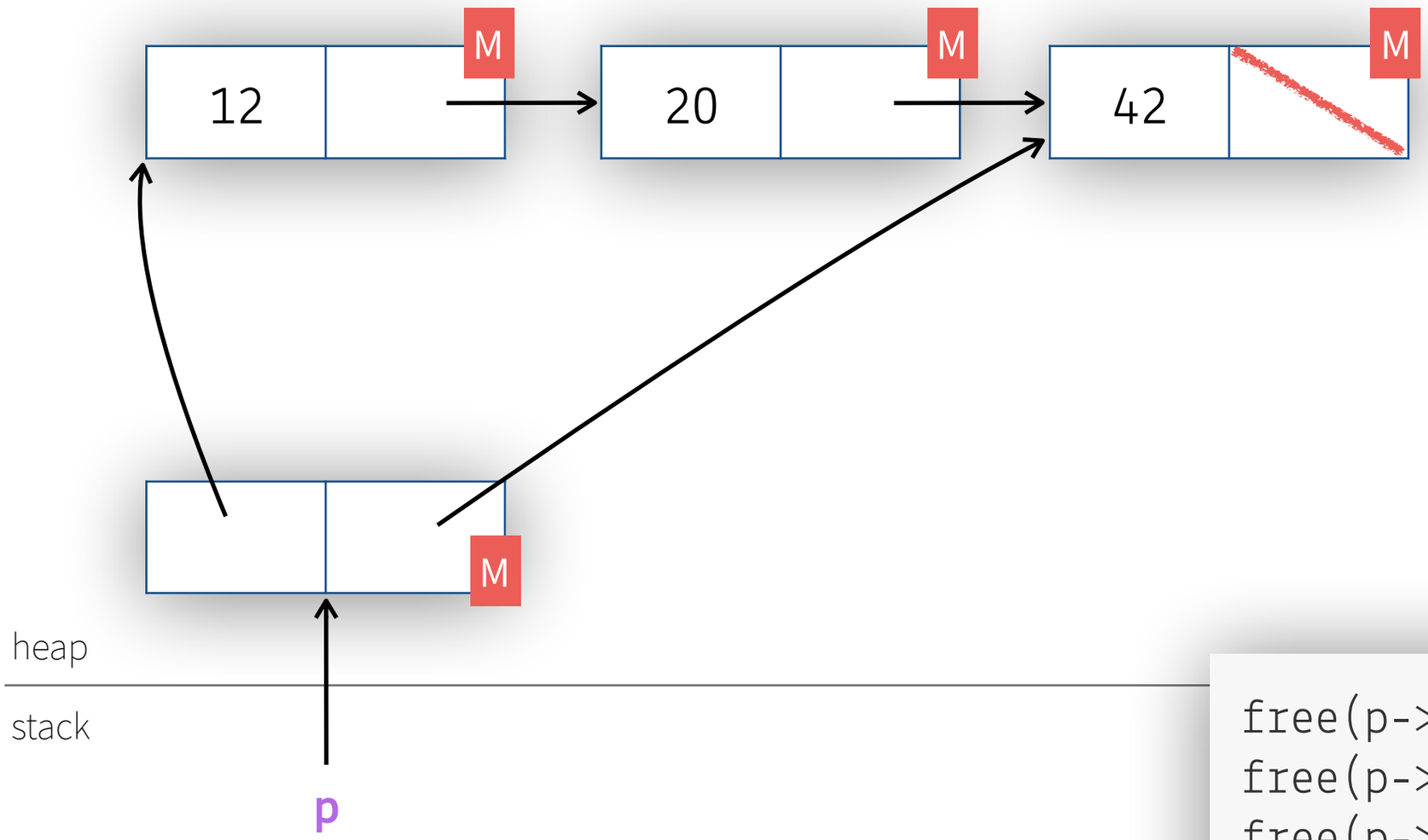
  …men det är ändå ett felaktigt program

- 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

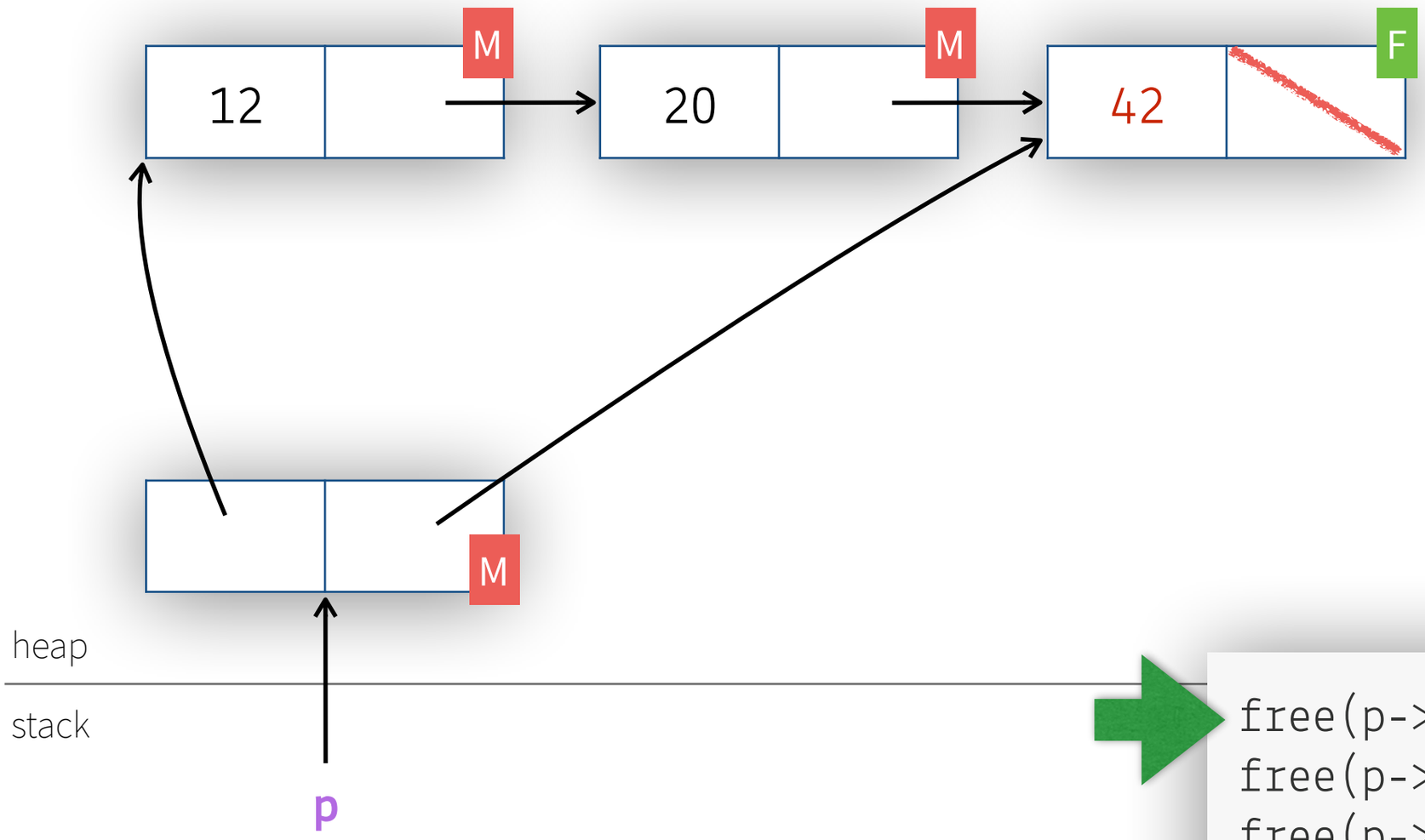
- Efter `free(p)` blir `p` en s.k. **dangling pointer** — en pekare till ett minnesblock som inte längre finns

Frigöra en länkad lista — försök 3 (1/5) [OBS! Korrekt]



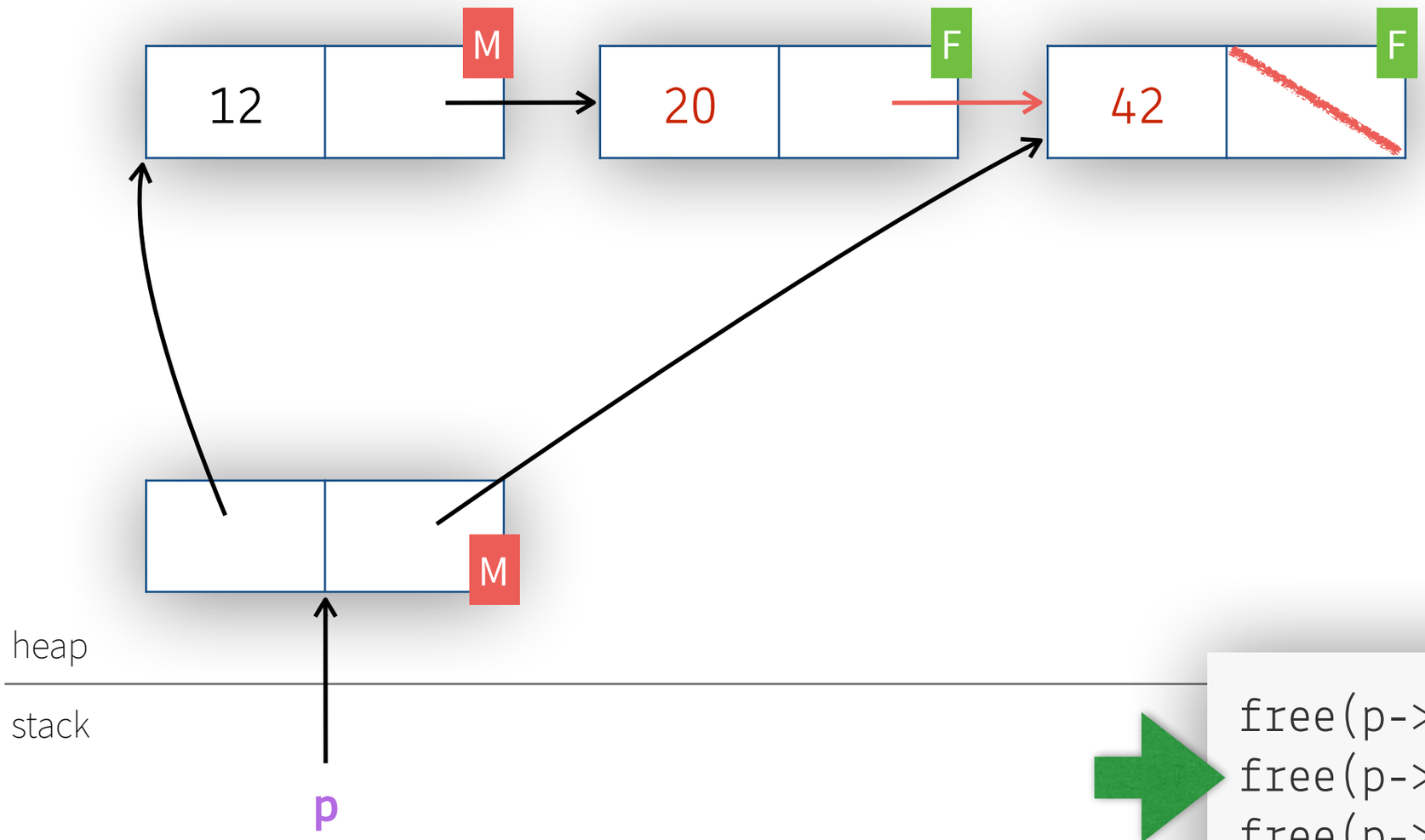

```
free(p->first->next->next);
free(p->first->next);
free(p->first);
free(p);
```

Frigöra en länkad lista — försök 3 (2/5) [OBS! Korrekt]



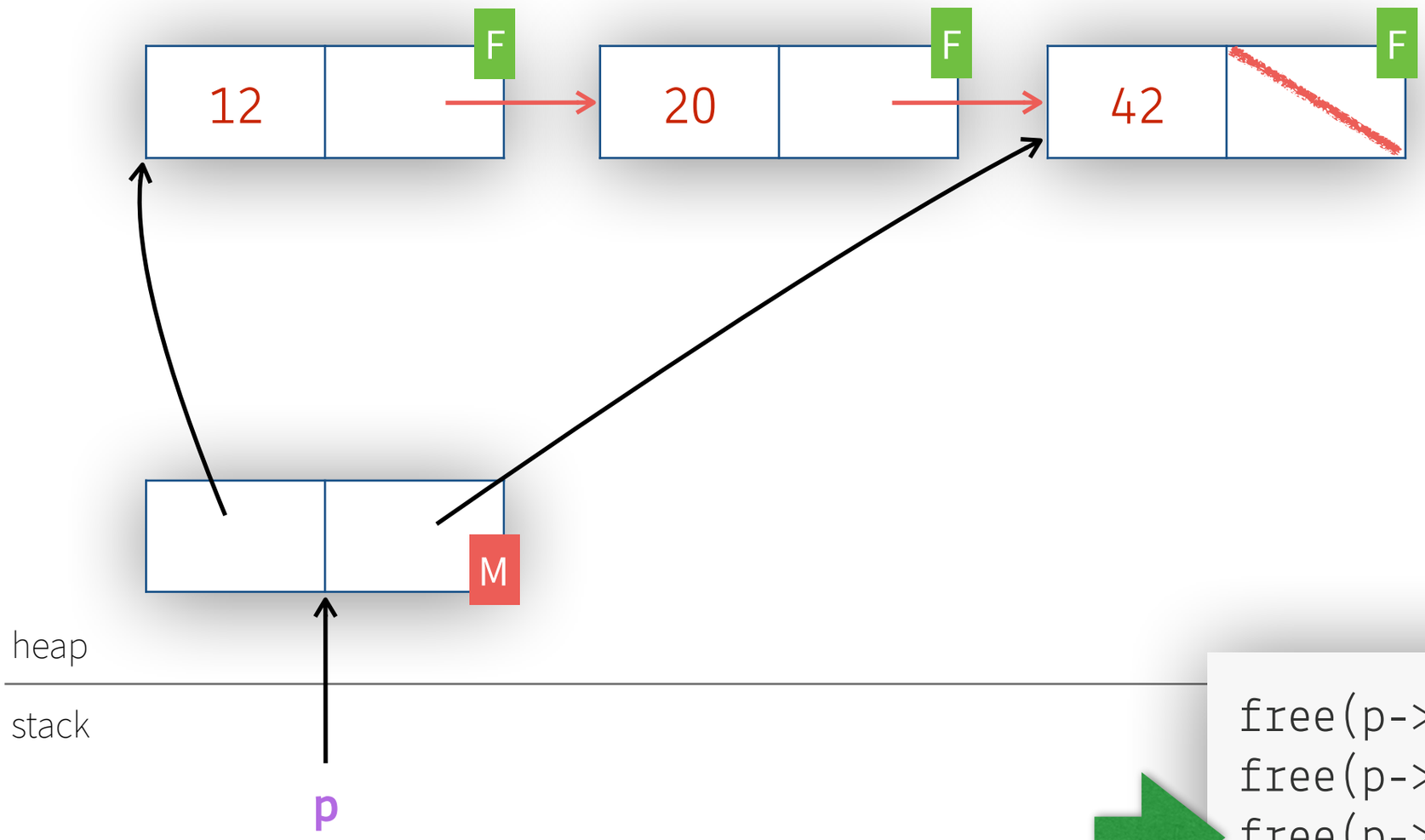

```
free(p->first->next->next);
free(p->first->next);
free(p->first);
free(p);
```

Frigöra en länkad lista — försök 3 (3/5) [OBS! Korrekt]



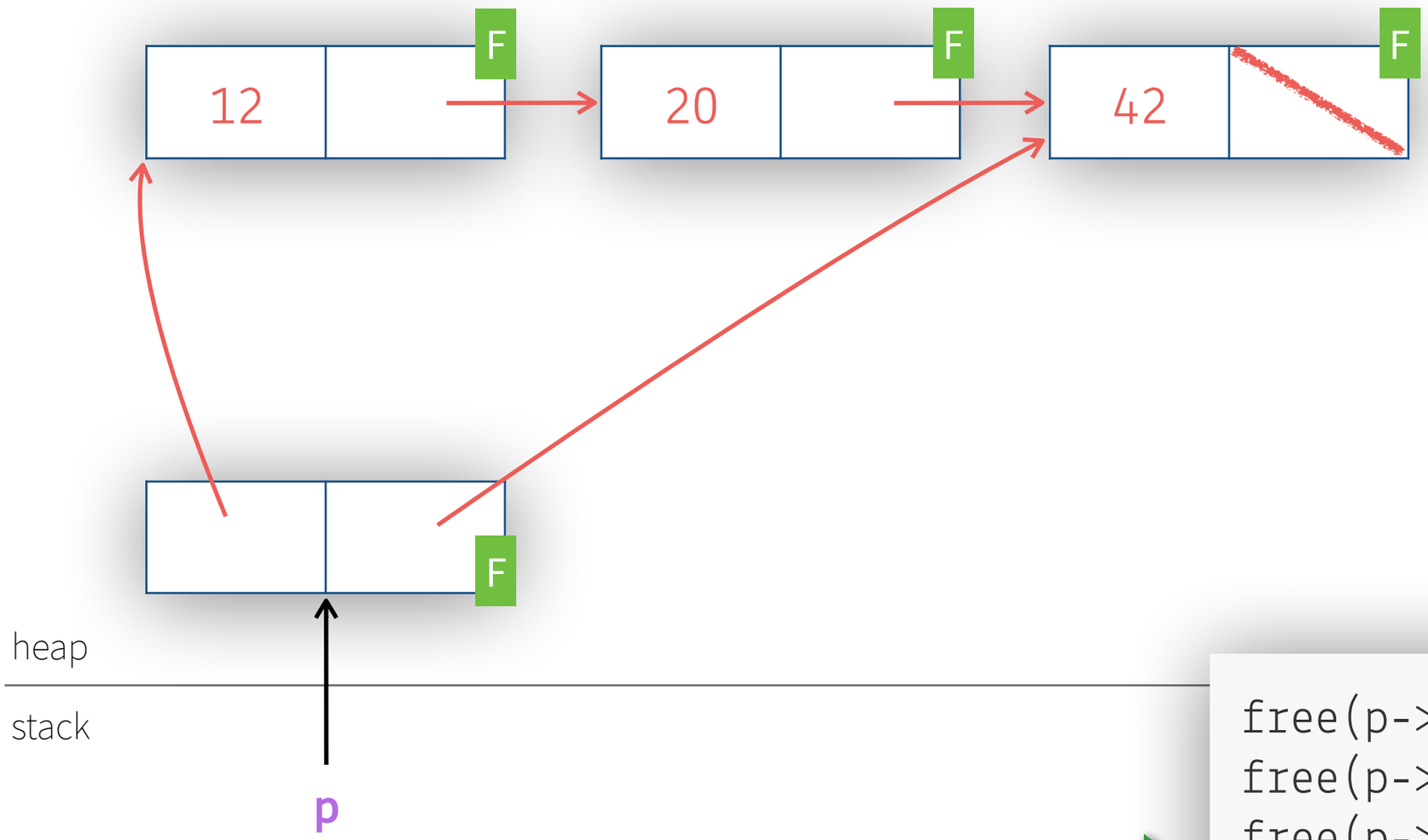

```
free(p->first->next->next);
free(p->first->next);
free(p->first);
free(p);
```

Frigöra en länkad lista — försök 3 (4/5) [OBS! Korrekt]




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

Frigöra en länkad lista — försök 3 (5/5) [OBS! Korrekt]




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

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

Några ord om modularisering och "ägarskap"

- Program som använder med listor arbetar i praktiken aldrig själv med länkarna och länkpekarna.
- Tillsammans med datastrukturerna för listorna definierar man funktioner som utför grundläggande funktioner på listor (skapa, sätt in element, tag bort element etc.)

```
list_t *list_new();
void list_insert(list_t *list, int index, int element);
bool list_remove(list_t *list, int index);
void list_delete(list_t *list);
.... // med mera
```

- Det gör att pekare till länkar aldrig skall finnas utanför listorna själva.
- Endast listfunktionerna får skapa och frigöra länkar.
- list-strukturen "äger" länkarna.
- Se http://wrigstad.com/ioopm19/assignments/lists.html för en mera ingående diskussion.

Iteratorer

- För att kunna gå igenom en listor utan att behöva komma åt länkarna kan man arbeta med *iteratorer* – datastrukturer som håller reda på en plats i listan.

```
struct list_iterator {
   link_t *current;
}

typedef struct list_iterator list_iter_t;

list_iter_t *list_iter_new(list_t *l);
bool list_iter_more(list_iter_t *i);
int  list_iter_next(list_iter_t *i);
void list_iter_free(list_iter_t *i);
```

```
list_t *p = ..... // Access the list

iter_t *i = list_iter_new(p);

while (list_iter_more(i)) {
  int e = list_iter_next(i);
  .... // Process the element e
}

list_iter_free(i);
```

Hur fungerar iteratorer?

```
struct list_iterator {
   link_t *current;
}

typedef struct list_iterator list_iter_t;
```

```
list_iter_t *list_iter_new(list_t *l) {
  list_iter_t *i =
     malloc(sizeof(list_iter_t));
  i->current = l->first;
  return i;
}

bool list_iter_more(list_iter_t *i) {
  return (i->current != NULL);
}

int list_iter_next(list_iter_t *i) {
  int e = i->current->value;
  i->current = i->current->next;
  return e;
}

void list_iter_free(list_iter_t *i) {
  free(i);
}
```

Generella listor

- Vill man ha listor som kan användas med olika slags data kan värdet i länkarna vara en unionstyp eller en void-pekare.
- Void-pekare kan peka på godtyckliga data.

```
typedef struct link link_t;

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

- Innan man använder en void-pekare måste man tala om vad den pekar på.

```
link_t *l = malloc(sizeof(link_t));
l->value = "Hello, world!"

printf("%s\n", (char*) l->value);
```