Föreläsning 5

Tobias Wrigstad

Lite om länkade strukturer — notation, insättning, frigöra…

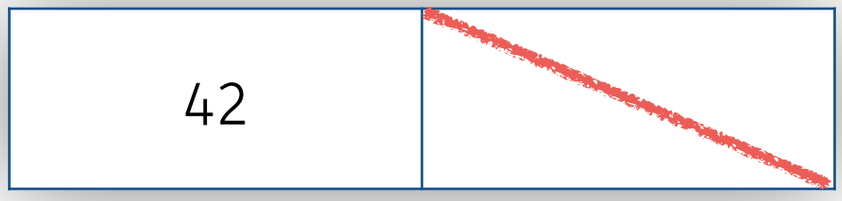
Notation

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

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

Struktarna som bygger upp en länkad lista

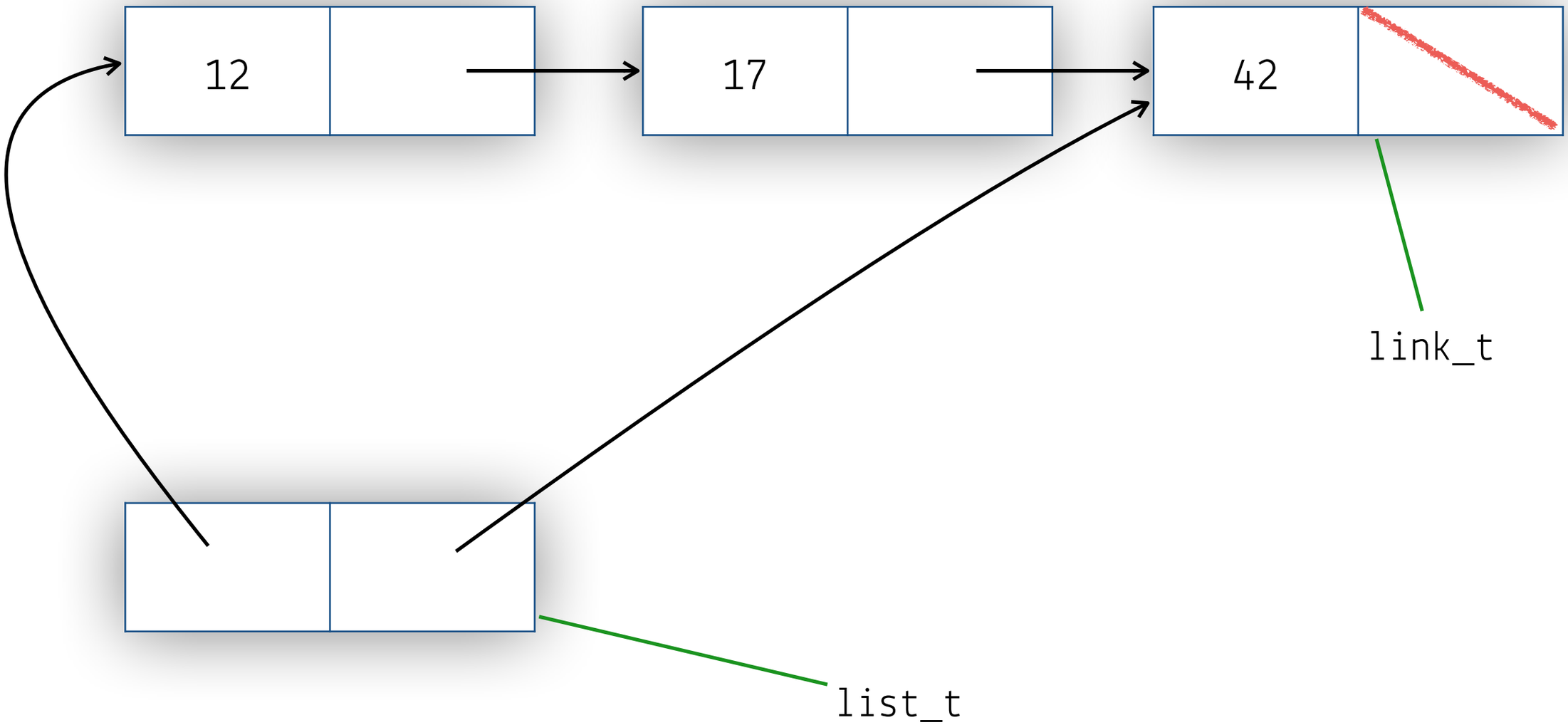
”Själva listan”

```
typedef struct list list_t;

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

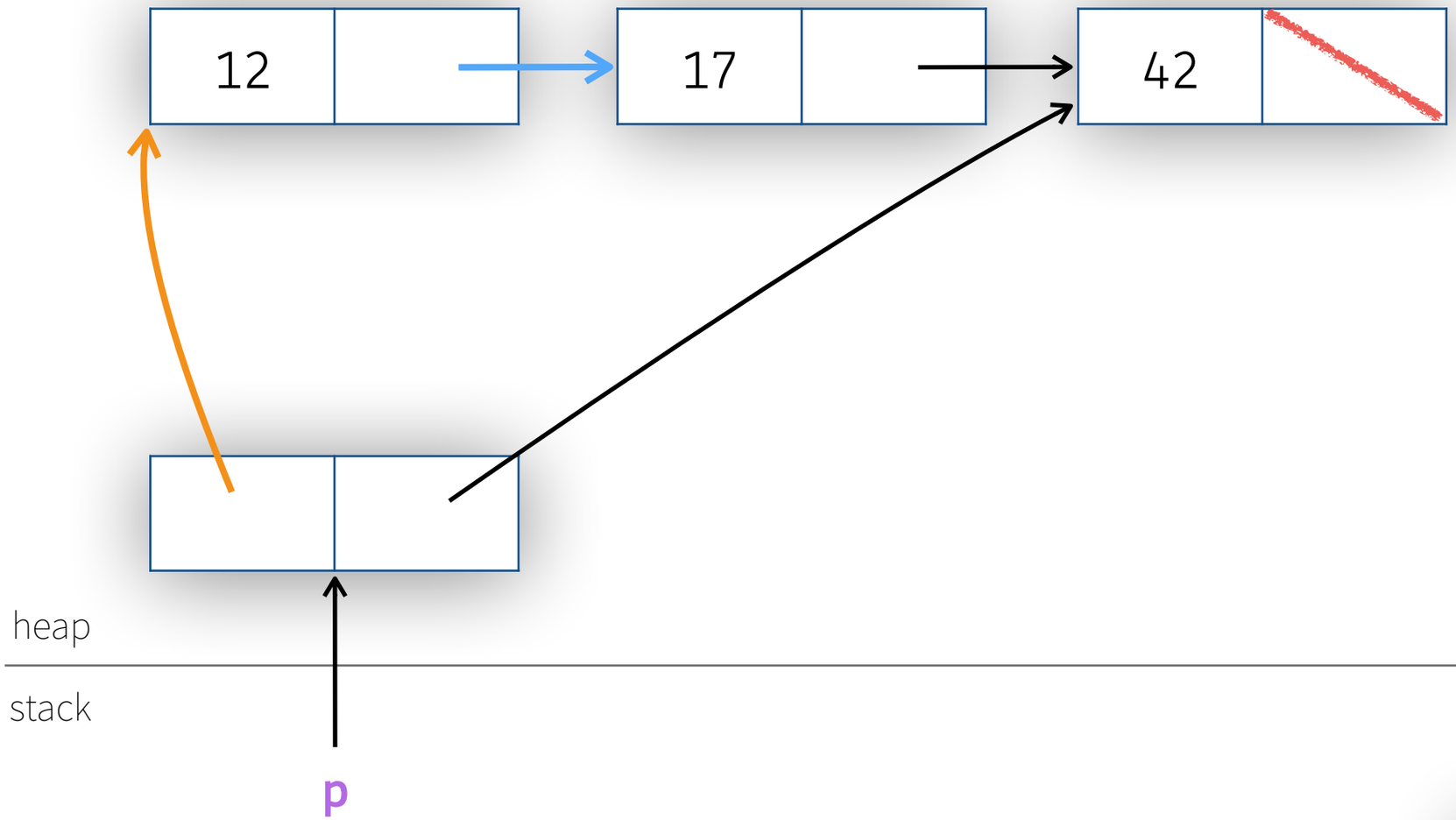
Länkarna

```
typedef struct link link_t;

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

En komplett länkad lista

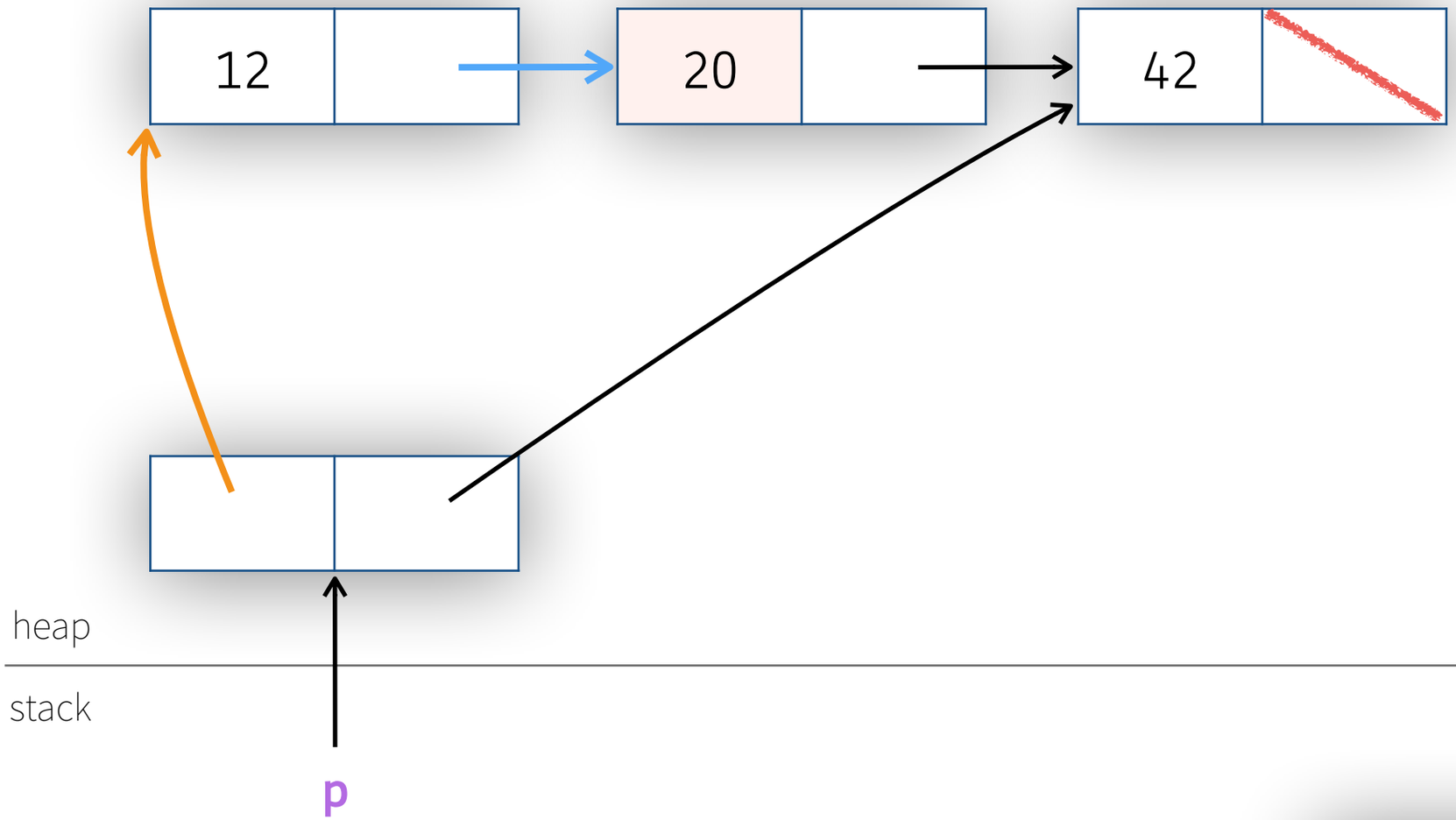


De olika delarna och hur man kommer åt dem



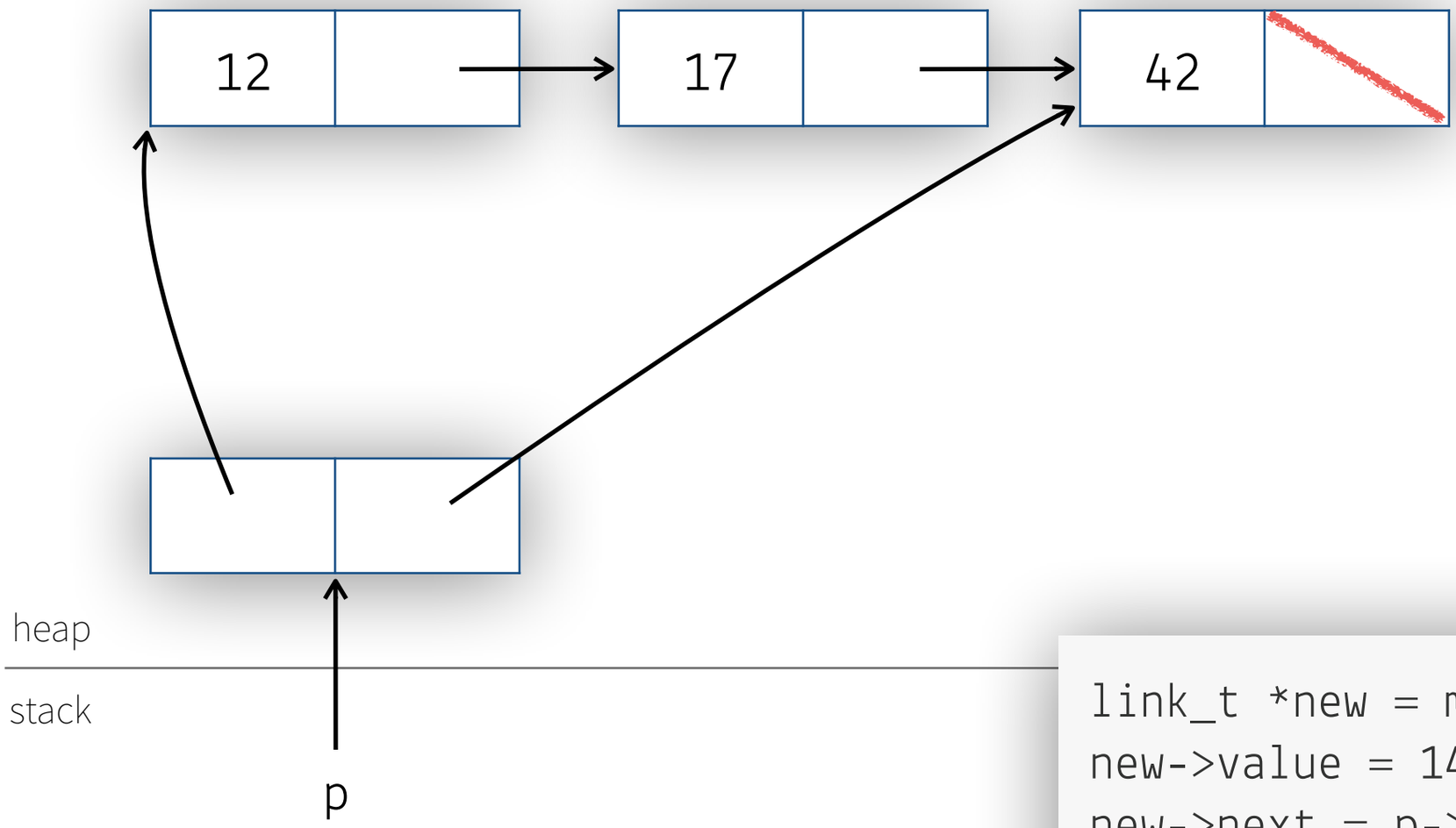

```
p->first->next
```

De olika delarna och hur man kommer åt dem



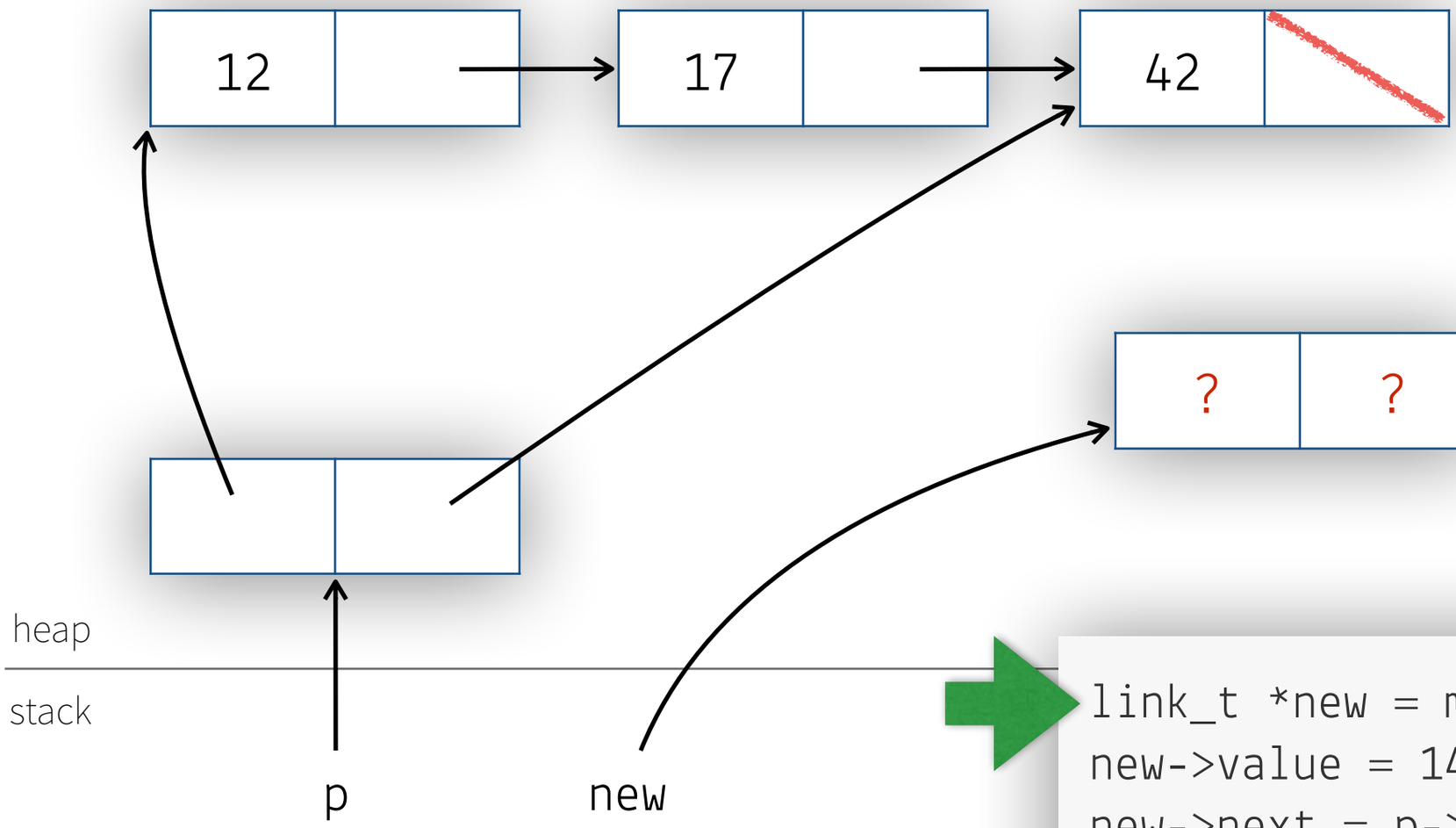

```
p->first->next->value = 20;
```

Lägg till ett element i listan



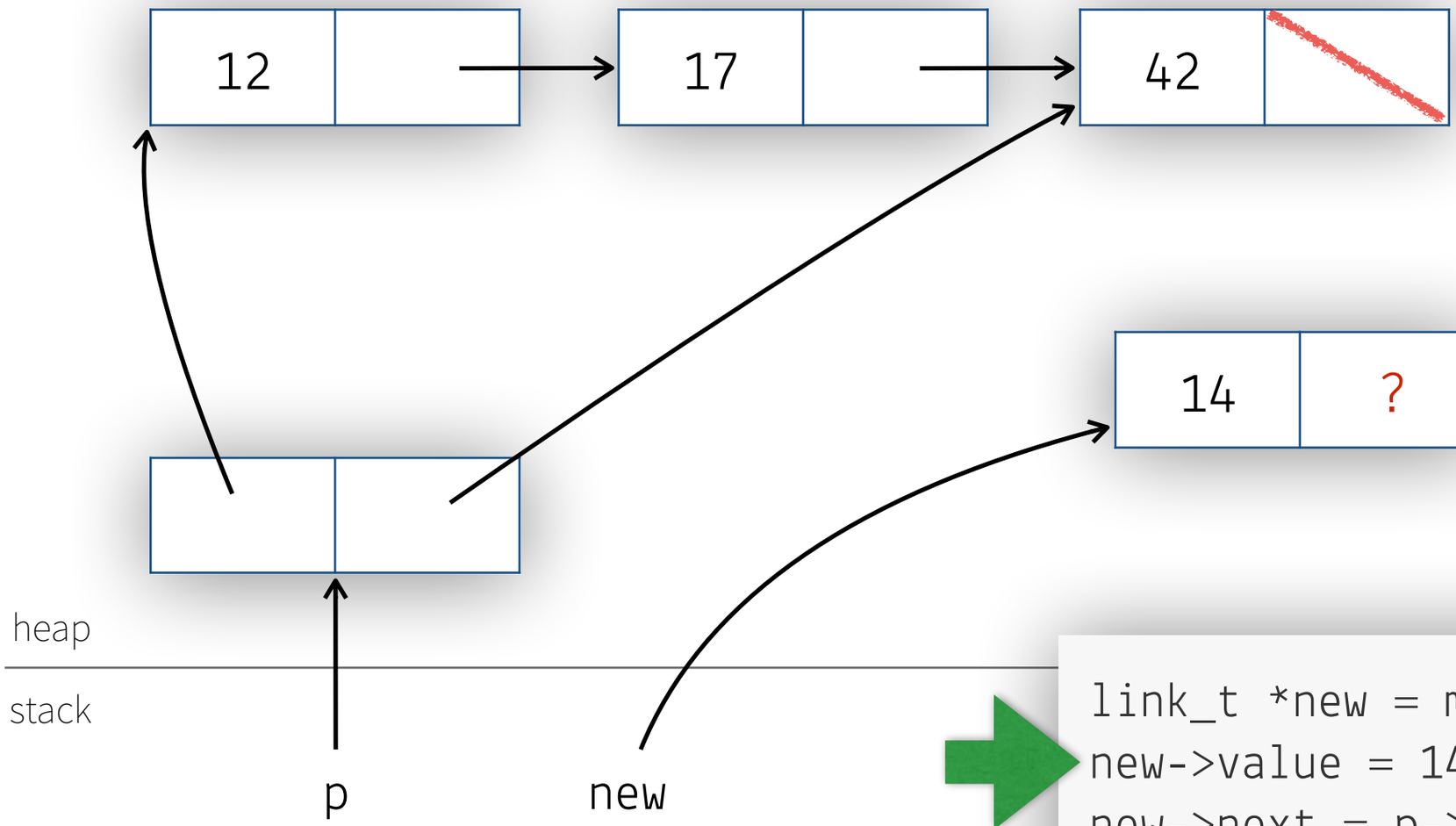

```
link_t *new = malloc(sizeof(link_t));
new->value = 14;
new->next = p->first->next;
p->first->next = new;
```

Lägg till ett element i listan [steg 1]



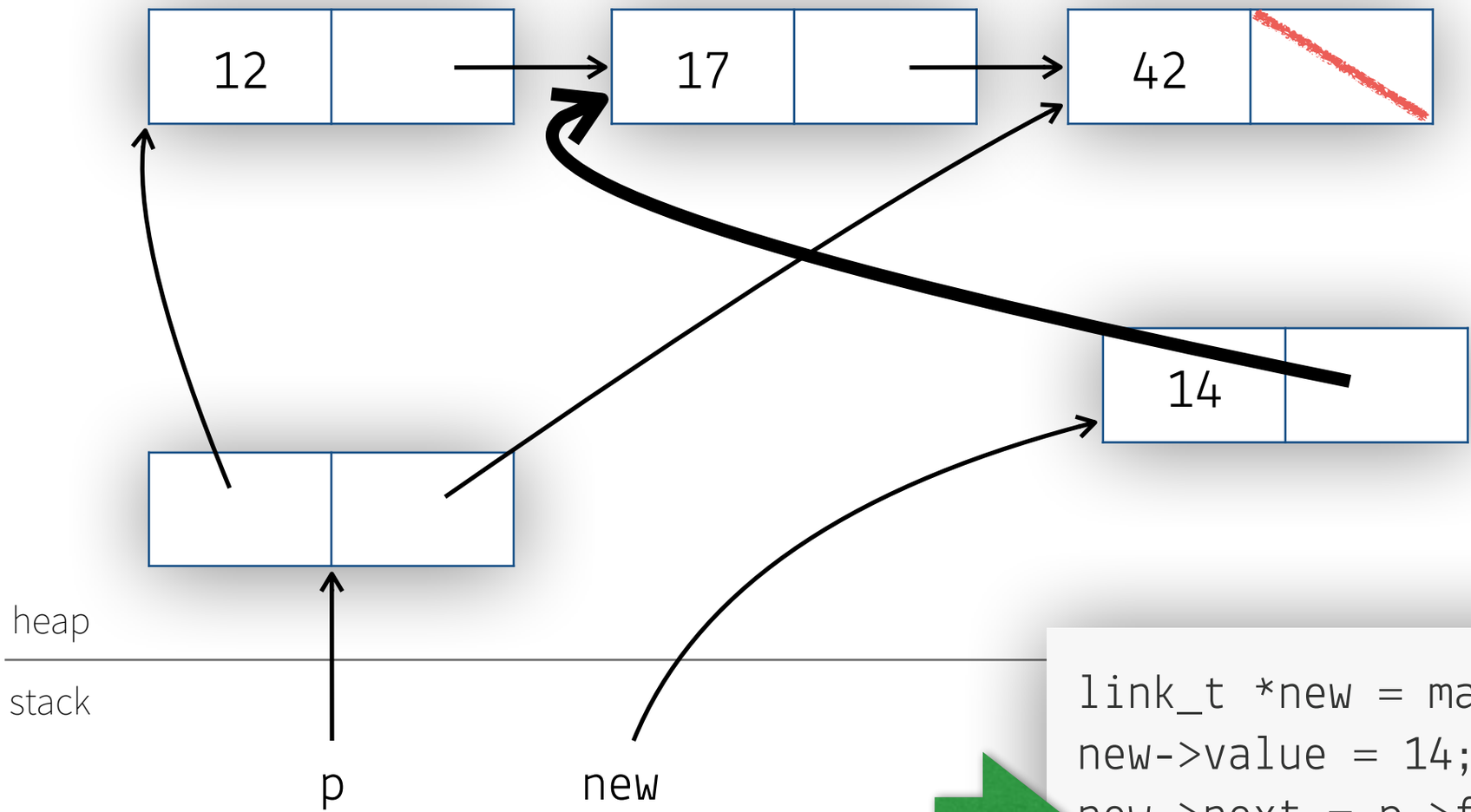

```
link_t *new = malloc(sizeof(link_t));
new->value = 14;
new->next = p->first->next;
p->first->next = new;
```

Lägg till ett element i listan [steg 2]



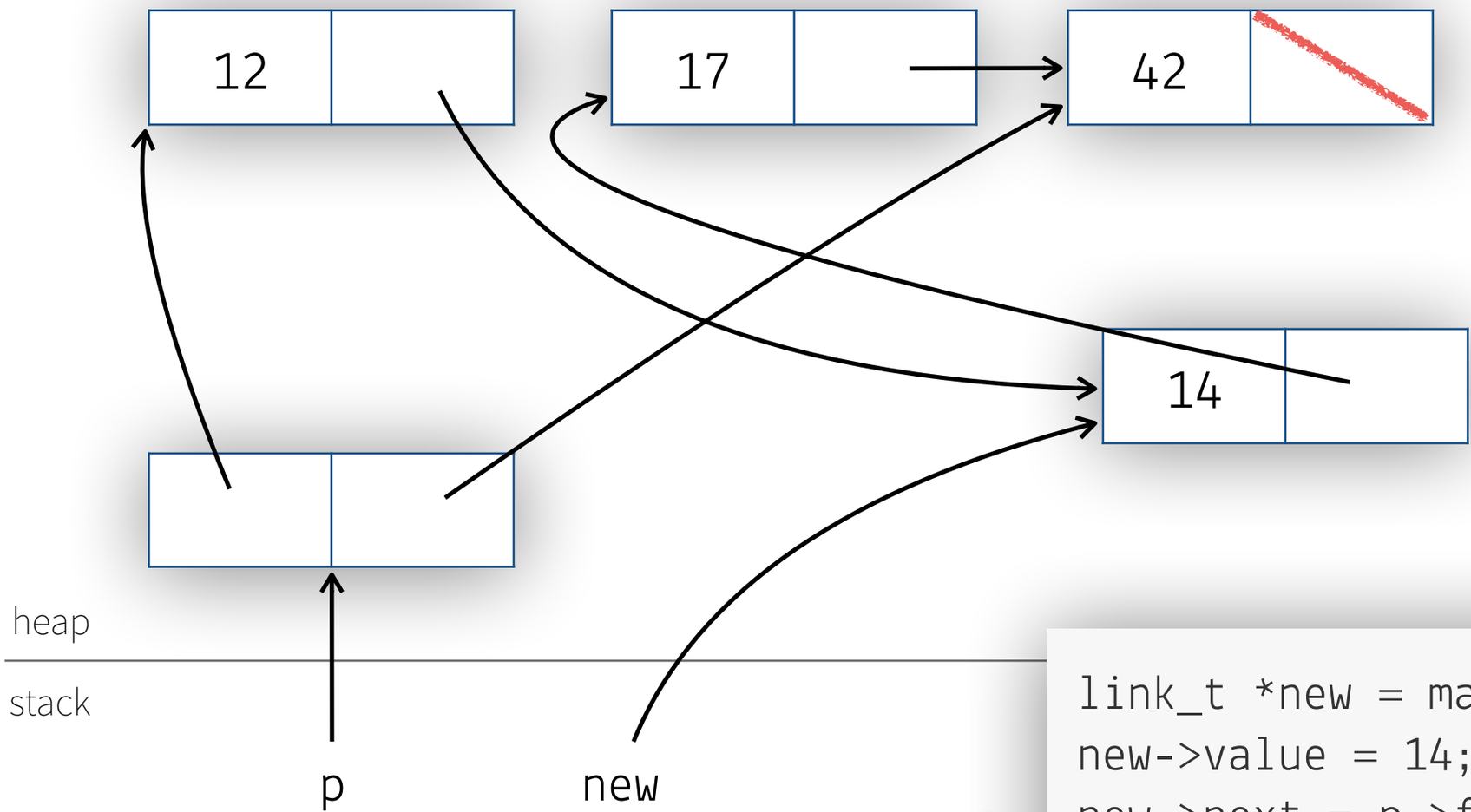

```
link_t *new = malloc(sizeof(link_t));
new->value = 14;
new->next = p->first->next;
p->first->next = new;
```

Lägg till ett element i listan [steg 3]



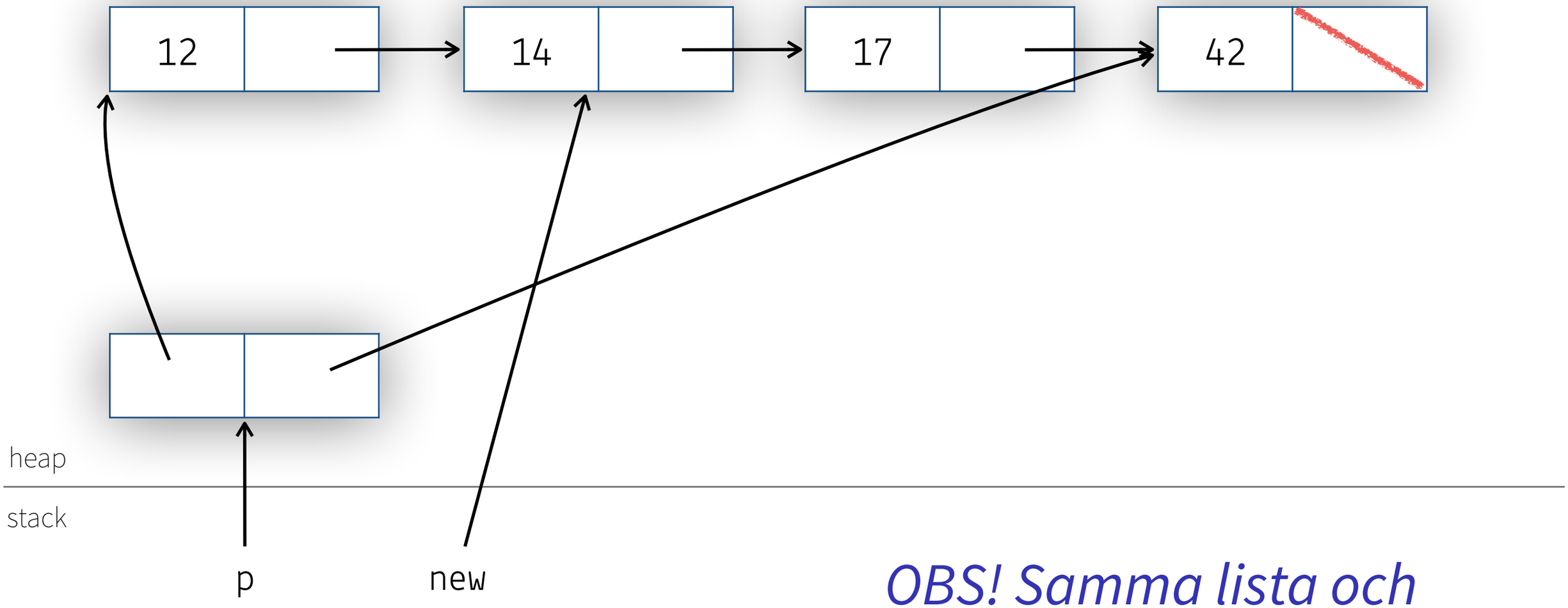

```
link_t *new = malloc(sizeof(link_t));
new->value = 14;
new->next = p->first->next;
p->first->next = new;
```

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

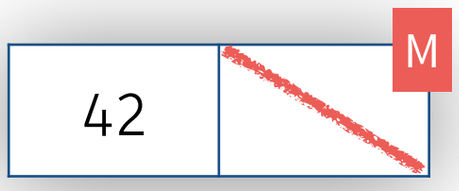
  Utgå från programmet nedan som loopar genom kommandoradsargumenten och skriver ut dem (utan en länkad lista)

- 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 frigöra en länkad lista

- Ny notation:



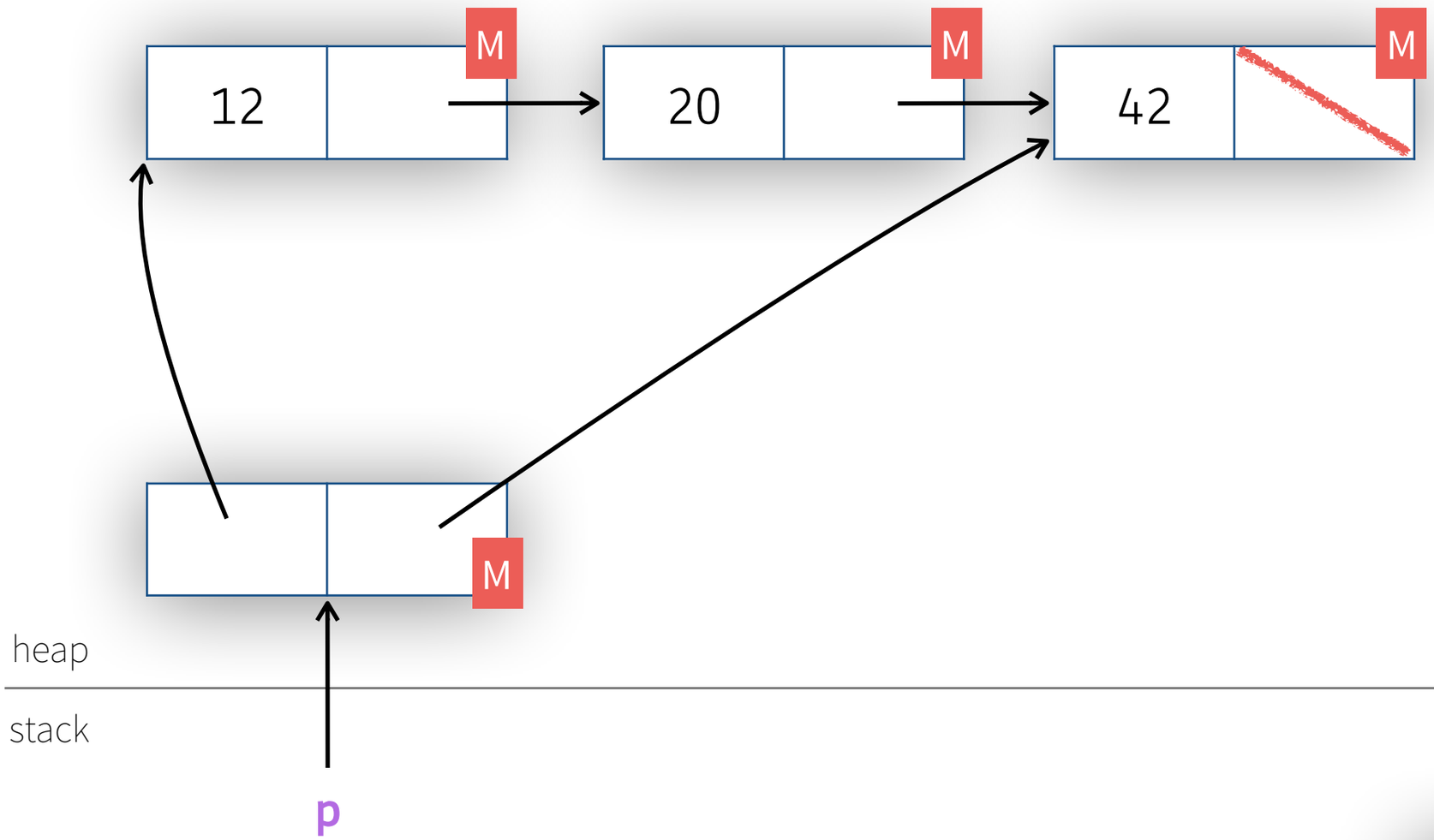
M betyder att `malloc` anser att strukten används

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

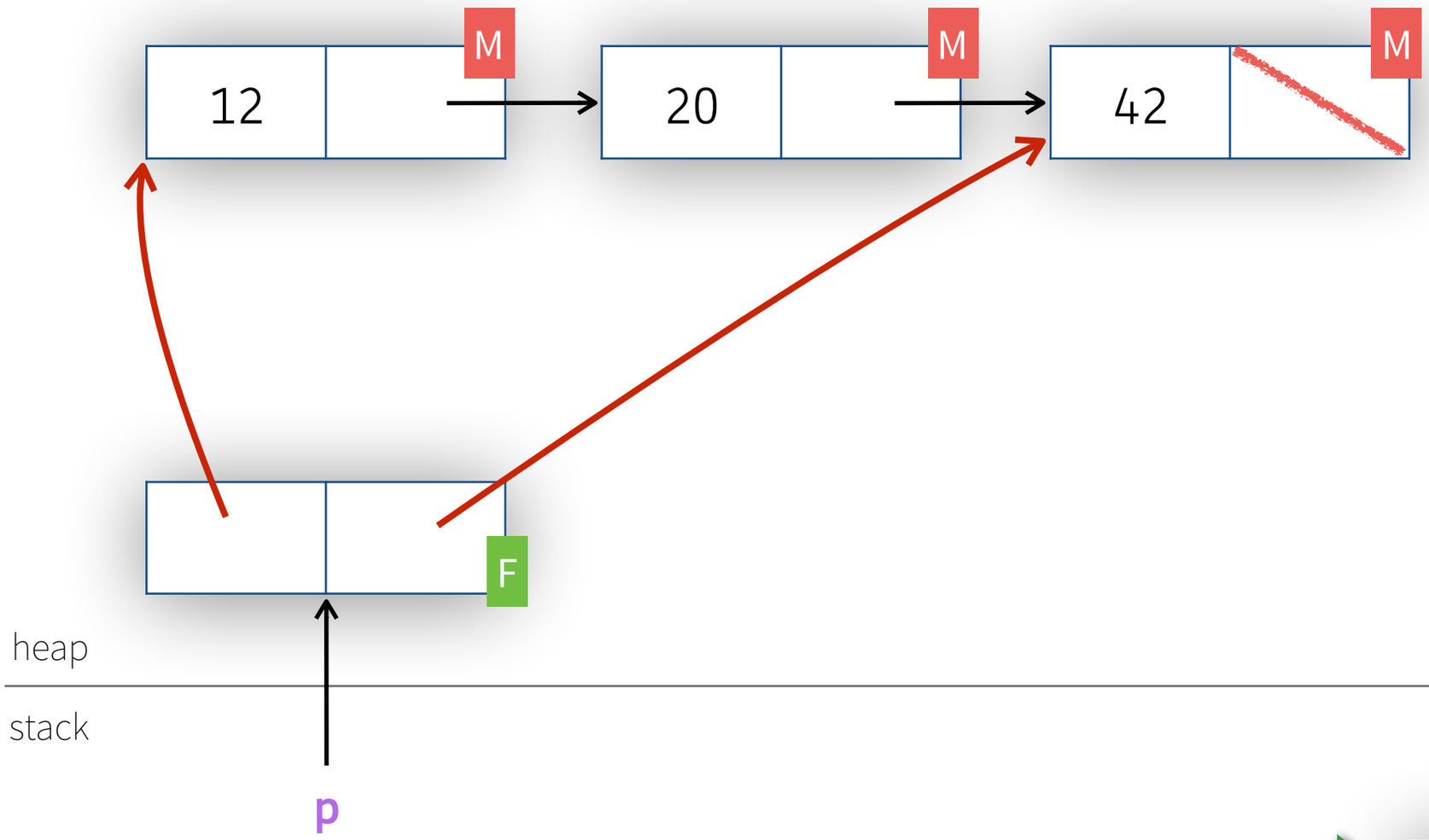
Röda värden i strukten betyder att de kanske inte finns längre

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




```
free(p);
```

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




```
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:

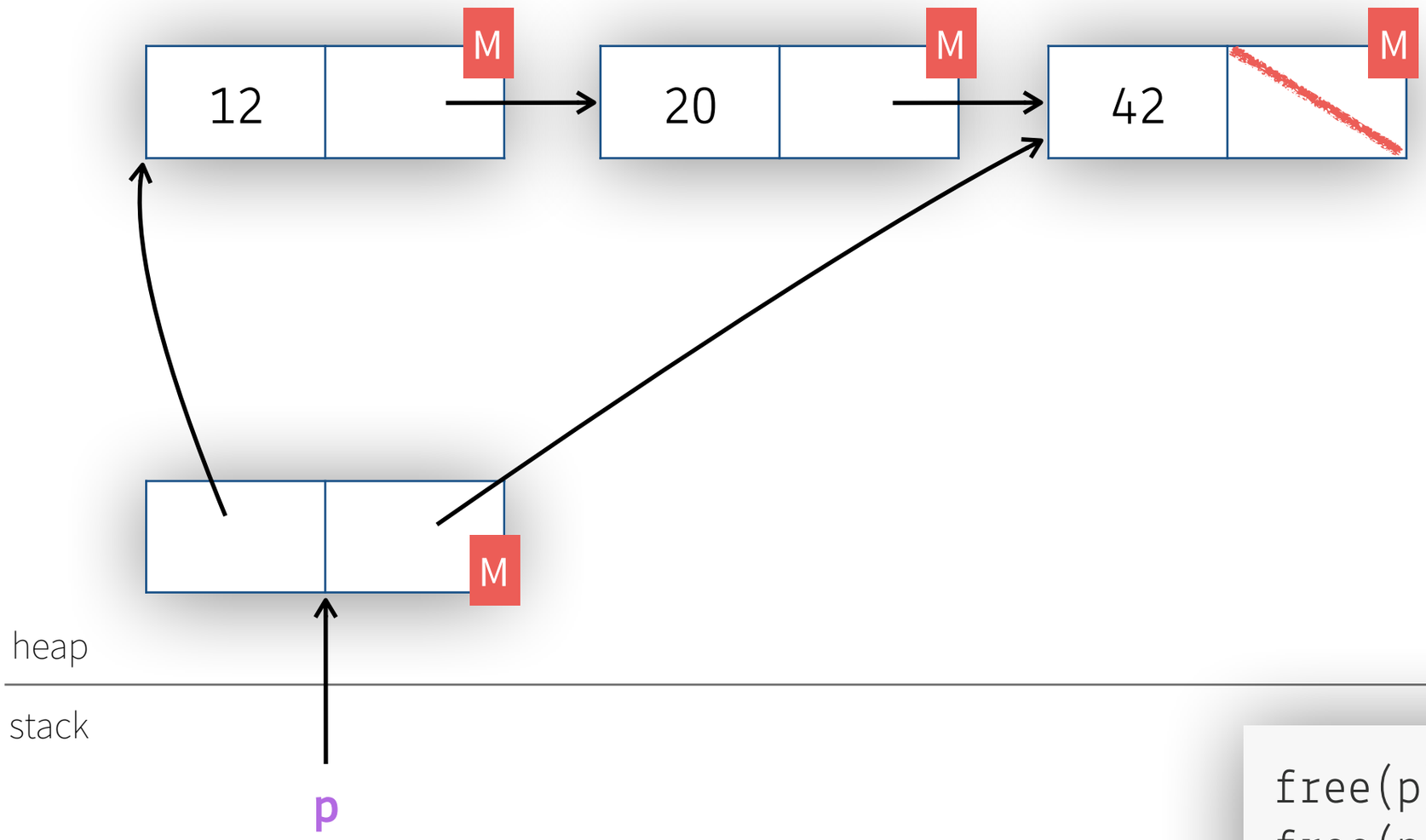
  efter att du gjort `free(p)` får inte inte använda p igen förrän du har pekat om `p` att peka på något data som du vet är validt

- Konsekvens av `free(p)` blir därför

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

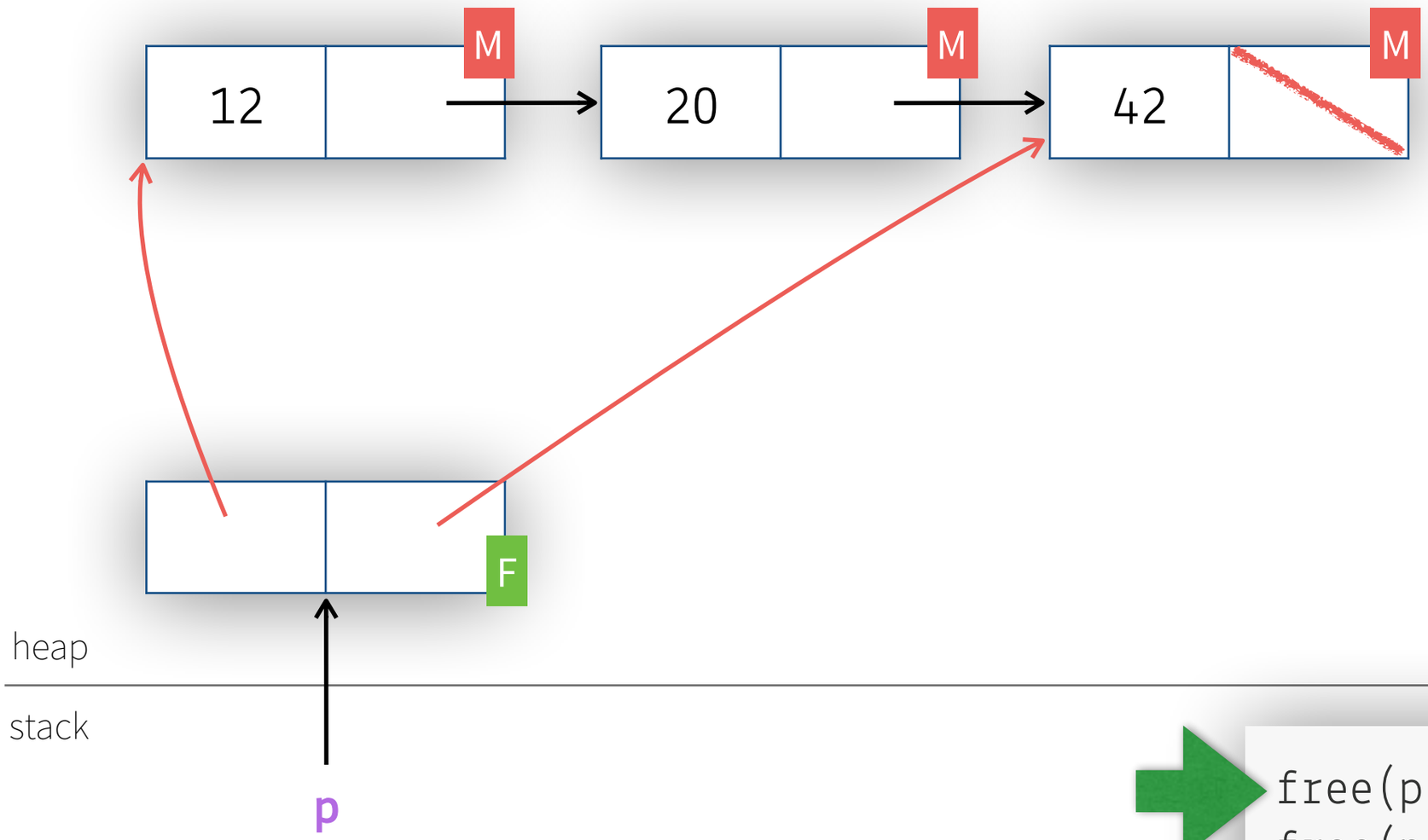
  …vilket i förlängningen betyder att vi läcker minne (3 `link_t`-struktar för att vara exakt)

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



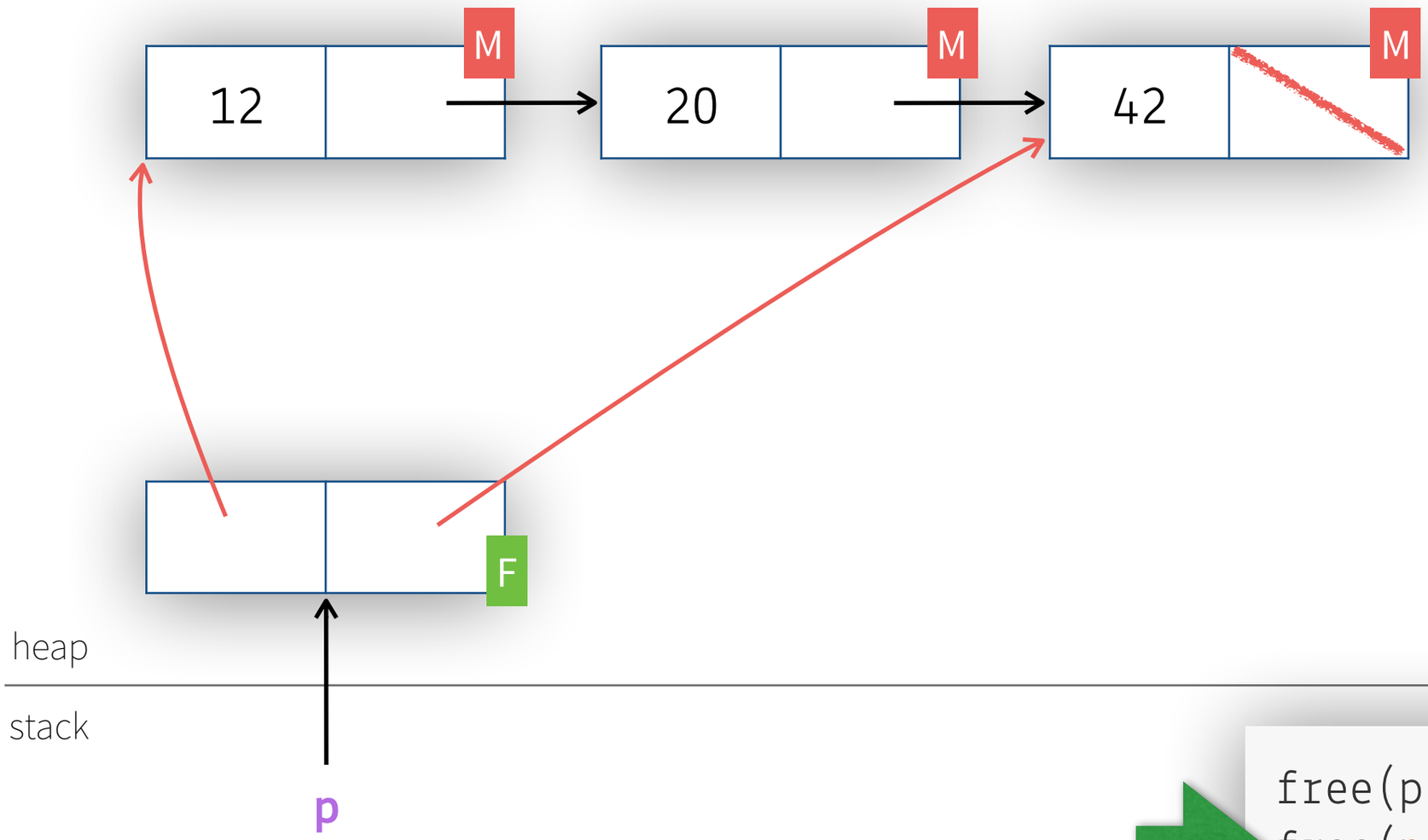

```
free(p);
free(p->first);
free(p->first->next);
free(p->first->next->next);
```

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




```
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);`

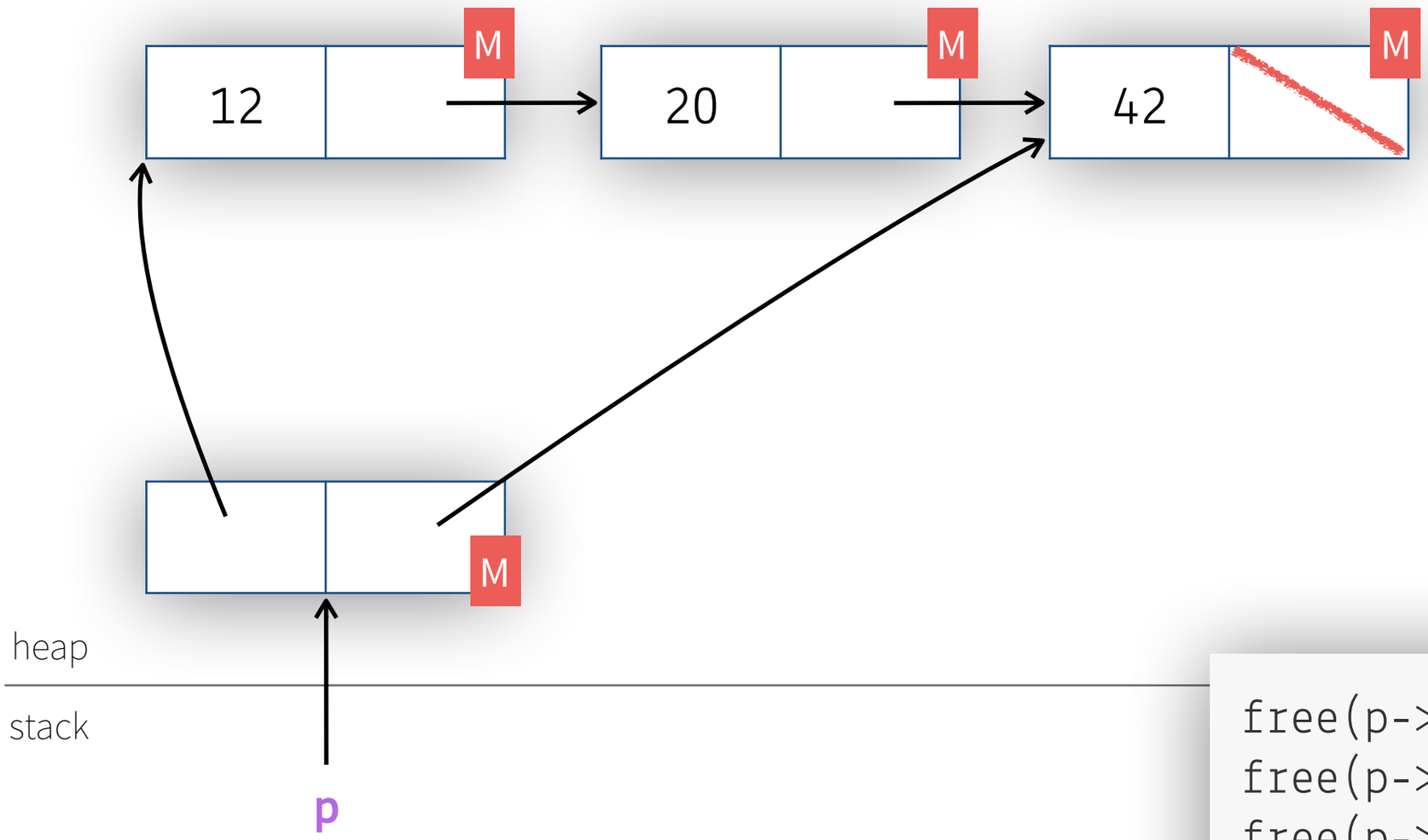
  …men det är ändå ett felaktigt program

- 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

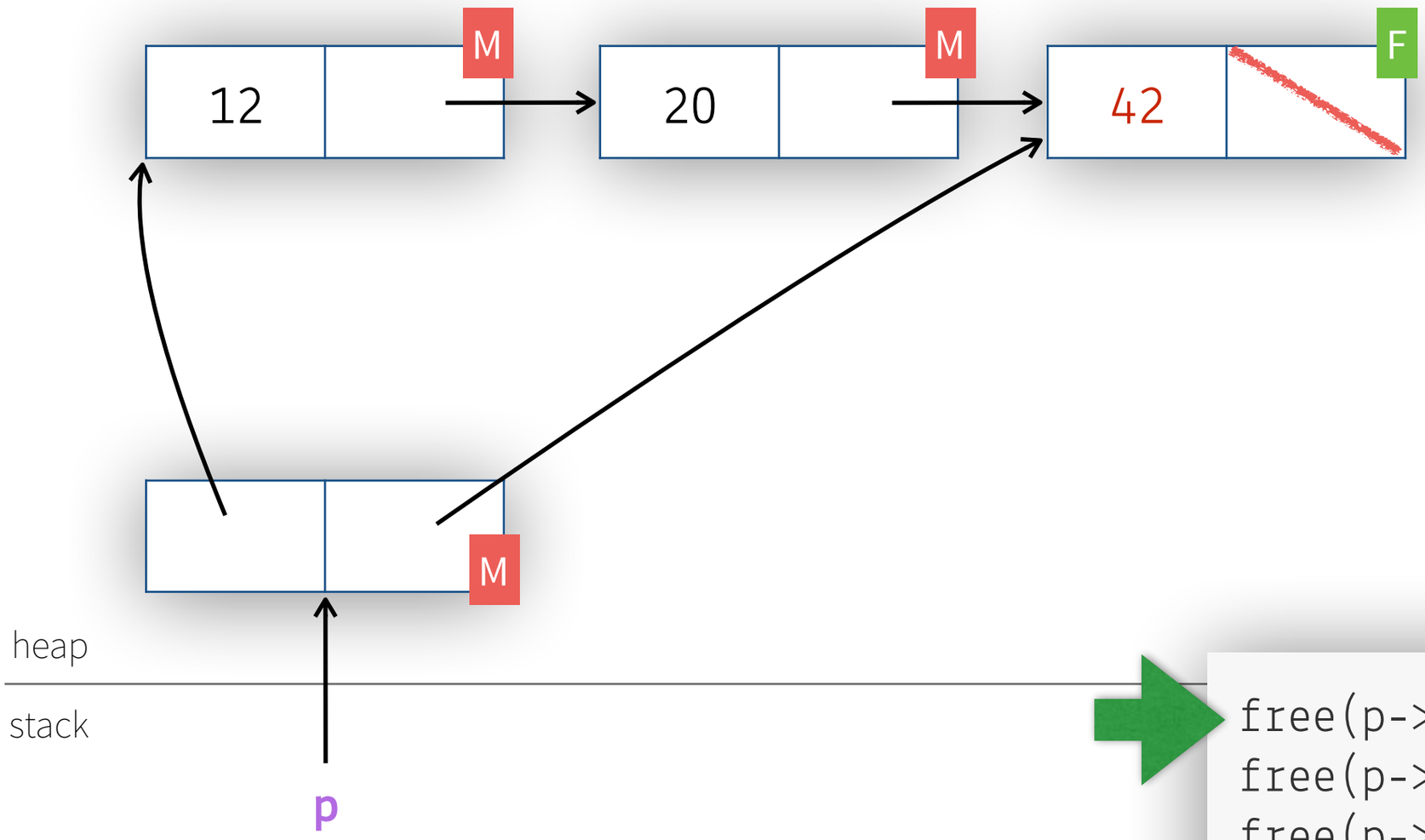
- Efter `free(p)` blir `p` en s.k. **dangling pointer** — en pekare till ett minnesblock som inte längre finns

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



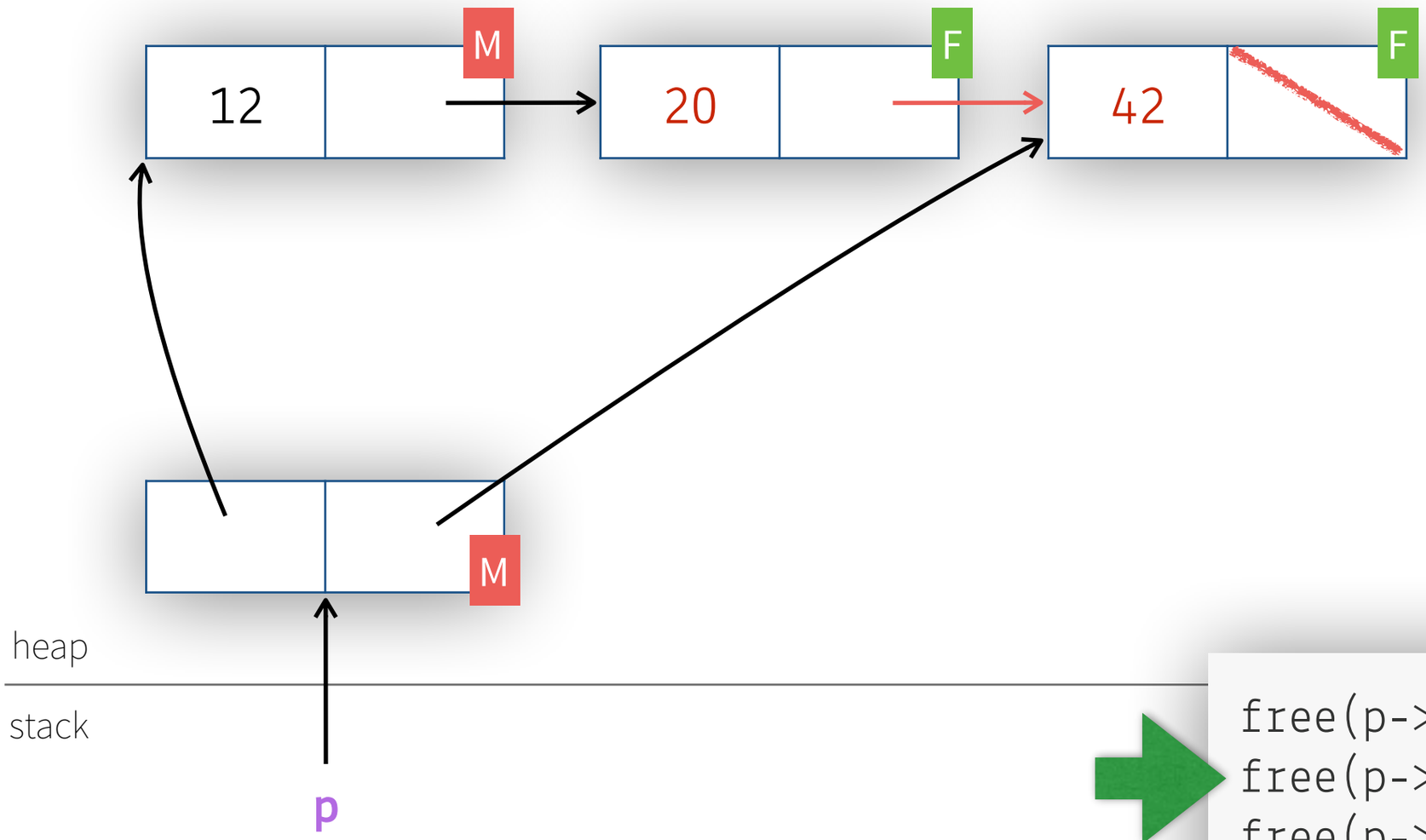

```
free(p->first->next->next);
free(p->first->next);
free(p->first);
free(p);
```

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



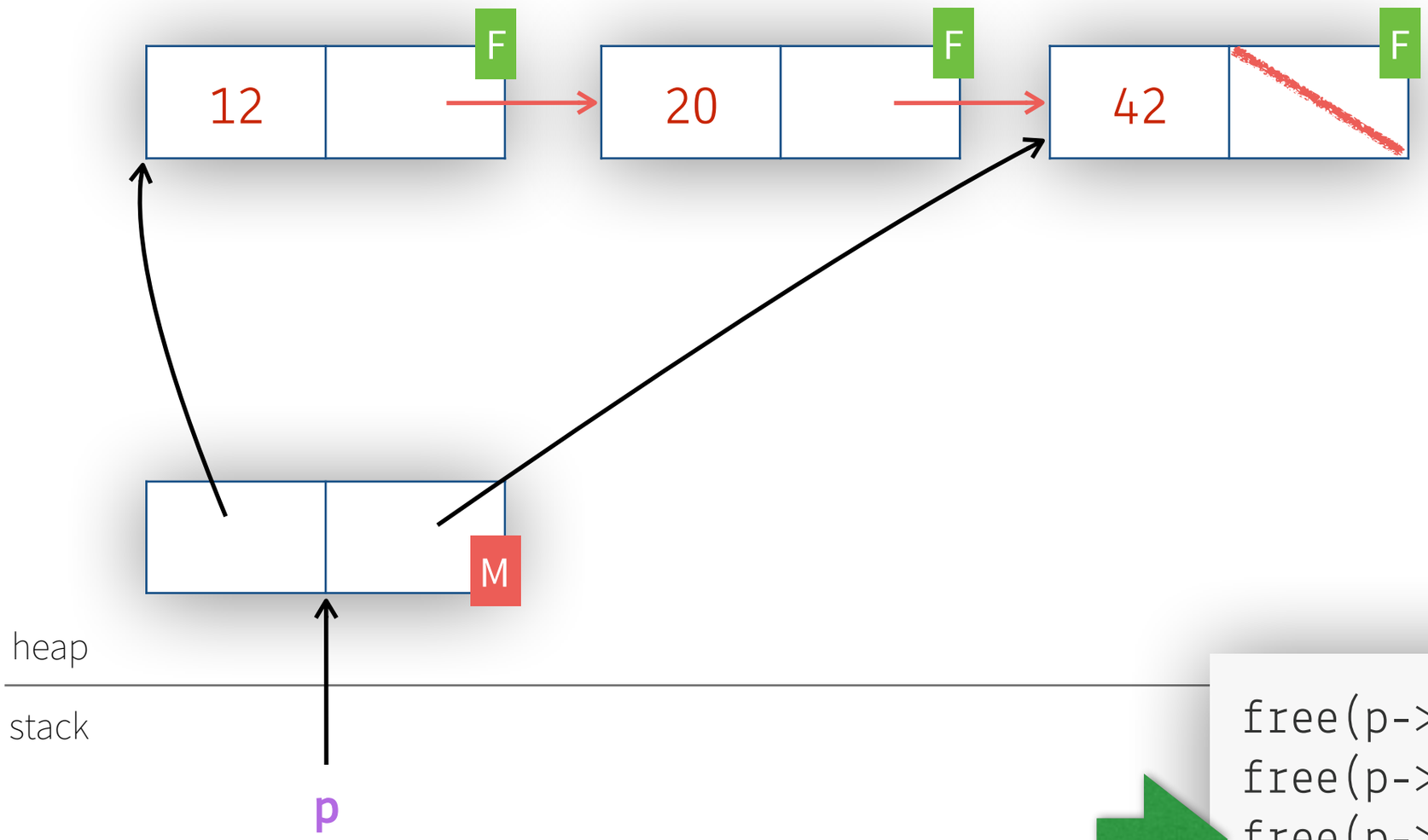

```
free(p->first->next->next);
free(p->first->next);
free(p->first);
free(p);
```

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



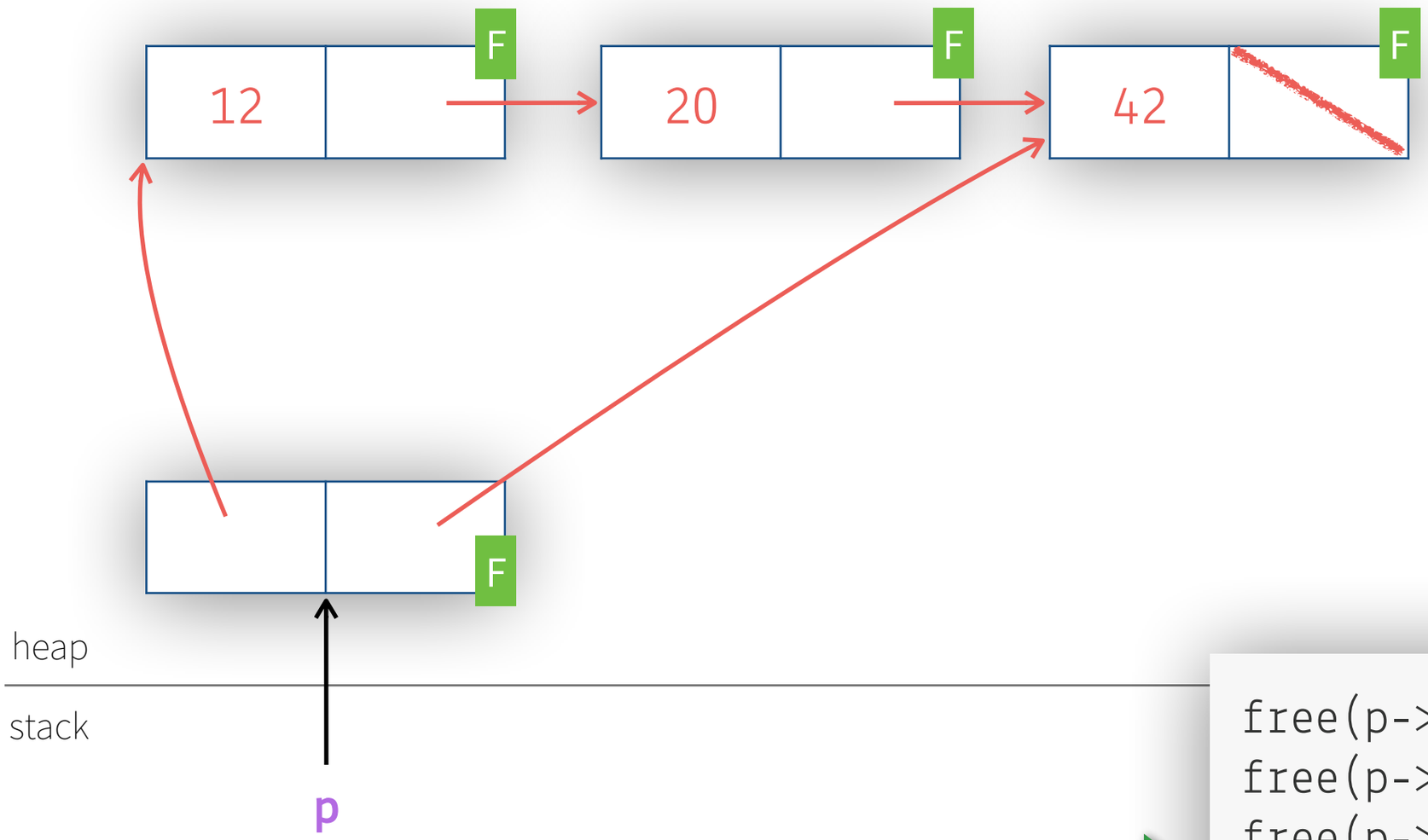

```
free(p->first->next->next);
free(p->first->next);
free(p->first);
free(p);
```

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




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

Frigöra en länkad lista — försök 3 [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

Livekodning

Iteratorer