Struct e liste concatenate

Alessio Orlandi

20 aprile 2010

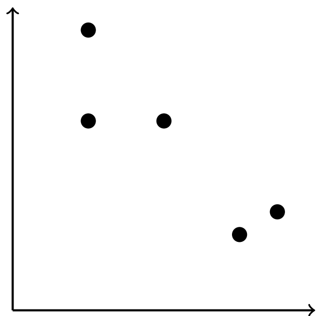
Tipi complessi

- Tipi scalari: *int*, *float*, ... : singolo elemento
- Contenitori per collezioni di oggetti: array.
- Quindi: array di interi, array di float, etc..

Tipi complessi

- Tipi scalari: *int*, *float*, ... : singolo elemento
- Contenitori per collezioni di oggetti: array.
- Quindi: array di interi, array di float, etc..
- E per i tipi complessi?

Esempio

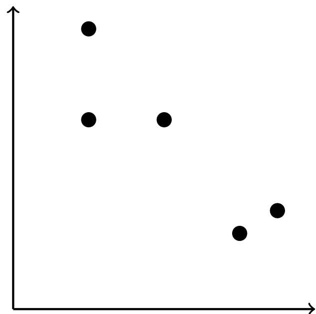


Si consideri una sequenza di punti sul piano e si determini tutte le coppie di punti che sono collineari su uno degli assi.

Punto = (x, y). In C:

```
int x[MAXPUNTI], y[MAXPUNTI]; // Brutto!
```

Anche peggio..



Si mantenga in memoria, per ogni punto, qual è il punto a destra più vicino.

```
int x[MAXPUNTI], y[MAXPUNTI];
int destra_x[MAXPUNTI], destra_y[MAXPUNTI];
```

Il riferimento non può avvenire con un puntatore!

Soluzione: *struct*

La solzione consiste nel creare un nuovo tipo, *punto*, il quale aggreghi le due coordinate, attraverso *struct*.

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

Ogni struttura è composta da campi. Essi sono contigui in memoria, ma a differenza degli array possono avere tipi *differenti*:

```
struct cerchio {
  int xcentro, ycentro;
  float raggio;
};
```

Uso

```
#include <stdio.h>
struct punto {
 int x, y;
};

main() {
  struct punto p1, p2;
  struct punto array[100];

  p1.x = 34; p1.y = 18;
  array[0].x = 99;
}
```

Uso con typedef

```
#include <stdio.h>
typedef struct s_punto {
  int x, y;
} punto;

main() {
   punto p1, p2;
   punto array[100];

   p1.x = 34; p1.y = 18;
   array[0].x = 99;
}
```

Puntatori a strutture

```c
#include <stdio.h>
typedef struct s_punto {
 int x, y;
 struct s_punto *ptr; // forzato
} punto;

main() {
  punto *p1, *p2;
  //...
  p1->x = 34; // (*p1).x = 34;
  p2->ptr = p1;
}
```

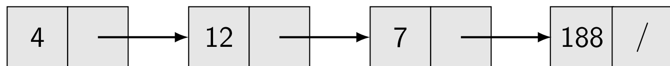
Puntatori a strutture

```
#include <stdio.h>
typedef struct s_punto {
 int x, y;
 struct s_punto *ptr; // forzato
} punto;

main() {
  punto *p1, *p2;
  //...
  p1->x = 34; // (*p1).x = 34;
  p2->ptr = p1;
}
```

Ma in pratica, qual è l'utilizzo primo? Liste!

Liste linkate semplici

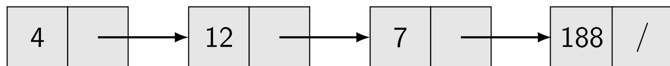


Una lista linkata è una struttura dati che contiene chiavi di un determinato tipo. Arbitrario di volta in volta. In una lista linkata semplice ogni elemento contiene:

- La chiave dell'elemento (int, in questo caso)
- Un puntatore al prox elemento (o un *tappo*).

Una lista è determinata dalla sua testa.
Vantaggi delle liste? Inserimento e cancellazioni nel mezzo in $O(1)$!

Liste semplici in C: definizione




```
typedef struct s_elemento {
    int chiave;
    struct s_elemento *prox;
} elemento;

..

elemento *testa = ...;
if ( testa->prox == NULL )
   printf("1 elem\n");
else
   printf("+ di 1 elem\n");
```

Rappresentazione in memoria

0x34:	4	0x41
0x39:	8	NULL
0x41:	1	0x39

```
elemento *testa = 0x34;
```

Lista per $4, 1, 8$.

Costruire un elemento

Siccome tutti gli elementi vengono riferiti per puntatore, ogni singolo elemento deve essere allocato a se stante.
Il puntatore *NULL* funge da tappo.

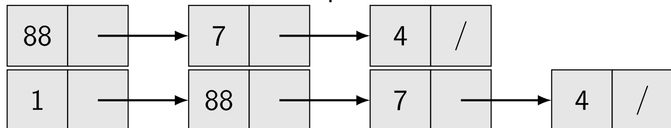
```
typedef struct s_elemento {
    int chiave;
    struct s_elemento *prox;
} elemento;

elemento *e1, *e2;
e1 = malloc(sizeof(elemento));
e2 = malloc(sizeof(elemento));

e1->chiave = 4;
e1->prox = e2;
e2->chiave = 8;
e2->prox = NULL; // tappo
```

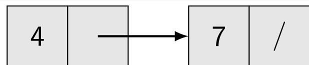
Inserimento in testa

Inserimento in testa è semplice:



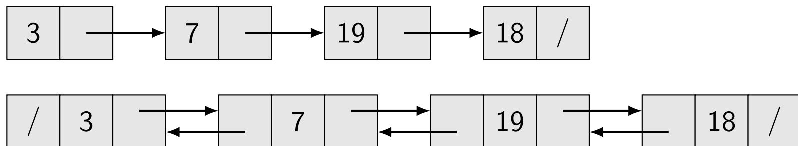

```
elemento *insTesta( elemento *testa, int c)
{
 elemento *nuova =  malloc(sizeof(elemento));
 nuova->chiave = c;
 nuova->prox = testa;
 return nuova;
}
...
elemento *lista = NULL;
lista = insTesta ( lista, 4 );
lista = insTesta ( lista, 8 );
```

Inserimento in coda



```
elemento *insCoda( elemento *p, int c) {
 elemento *nuova =  malloc(sizeof(elemento));
 nuova->prox = NULL; nuova->chiave = c;
 // Scorriamo fino a terminare la lista
 while ( p != NULL && p->prox != NULL )
   p = p->prox;
 if ( p == NULL ) // Lista vuota
   return nuova;
 else {
   p->prox = nuova;
   return p;
 } }
```

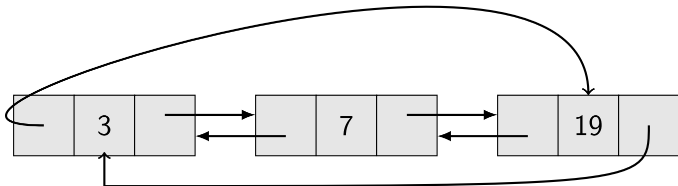
Liste doppiamente linkate




```
typedef struct s_elemento2 {
  struct s_elemento2 *prec;
  int dato;
  struct s_elemento2 *prox;
};
```

prec è NULL in testa, *prox* è NULL in coda.
Vantaggi: Scorrimento doppio. Svantaggi: Codice più convoluto.

Liste circolari



I concetti di *testa* e *coda* sfumano. La lista circolare può essere utilizzata come buffer circolare.

Esercizio 1

Questo esercizio serve per impratichirsi con la gesitone base delle liste, utilizzando gli interi. Scrivete un programma che:

1. Legga N, il numero di interi da leggere.
2. Legga N interi, e li inserisca in una nuova lista in coda.
3. Stampi la lista
4. Legga il valore x da tastiera, e cancelli dalla lista la prima occorrenza di x, se essa esiste.
5. Stampi la lista e torni al punto precedente, finchè la lista non è vuota.

Dopo ogni passo, provate il programma!

N.B, Quando i nodi vengono cancellati la loro memoria deve essere liberata (free)

Esercizio 2

Scrivete mergesort utilizzando le liste linkate semplici.
In particolare, notate che l'unica procedura da riscrivere veramente è *Fondi*: scrivete una procedura di fusione che operi solamente su liste.

Domanda: cosa notate rispetto alla versione con array? Quale porzione degli elementi in gioco è scomparso?

Esercizio 3

Scrivete un programma che implementi la strategia MTF (Move-To-Front) che riorganizza le liste (è anche sul libro). Si consiglia di usare liste *doppiamente* linkate.
Scrivete un programma che:

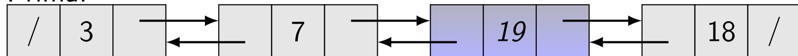
1. Legga N da tastiera, la dimensione della lista.
2. Legga N interi distinti e costruisca la lista inserendo in coda (*).
3. Legga l'intero x e stampi la posizione dall'inizio della lista in cui esso si trova, partendo dalla testa, oppure -1 se esso non è presente.
4. Prenda l'elemento della lista contenete x, lo distacchi dalla sua attuale posizione e lo faccia diventare la testa della lista
5. Torni al punto 3 infinitamente (per uscire: CTRL+C)

(*) potete anche evitare di usare l'inserimento in coda chiave per chiave e costruirla con un'unica procedura.

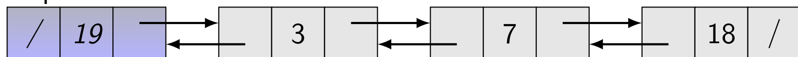
Esempio per esercizio 3

Input: 19
Output: 2 (contando da 0)

Prima:



Dopo:



Input: 14
Output: -1

La lista NON si modifica in questo caso.