Liste, Pile, Code

Salvatore Filippone
salvatore.filippone@uniroma2.it

Liste

Le *liste* sono delle strutture dati che realizzano il tipo di dato astratto "Sequence".

Liste con puntatori singoli

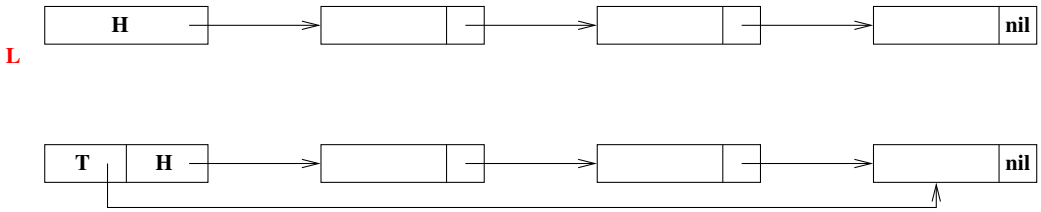
La lista con puntatori singoli consente di realizzare gli operatori `empty`, `head`, `next` e `insert` con una complessità $O(1)$; la complessità degli operatori `prev` e `remove` è $O(n)$, quella dell'operatore `tail` dipende dalla implementazione.

Si noti che le aree di memoria occupate dai dati non hanno in generale alcuna relazione ovvia tra loro.
Ciascun nodo della lista ha due componenti:

1. Un campo `value`;
2. Un campo `next` che contiene il puntatore al prossimo elemento;

La lista stessa è realizzata con un puntatore al primo elemento (se esiste, oppure **nil**); l'ultimo elemento della lista ha un campo `next` che vale **nil**.

Liste con puntatore singolo



Nella seconda variante presentata si mantiene esplicitamente un puntatore alla ultima posizione della lista

Metodi per LISTA

boolean empty()

Input: *LIST L*
Risultato $L.H == \textbf{nil}$

Pos head(LIST L)

Risultato $L.H$

Pos insert(Pos P, Item v)

Pos *new* T;
$T.value \leftarrow v$;
$T.next \leftarrow P.next$;
$P.next \leftarrow T$;
Risultato T

Pos next(Pos P)

$T \leftarrow P$;
if $T \neq \textbf{nil}$ **then**
  $T \leftarrow T.next$;
Risultato T

Pos prev(LIST L, Pos P)

Pos $T \leftarrow L$;
while $T \neq \textbf{nil}$ **and** $T.next \neq P$ **do**
  $T \leftarrow T.next$;
Risultato T

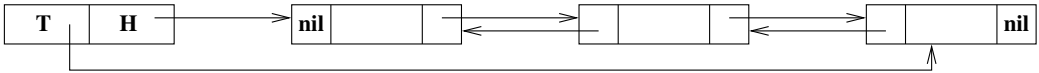
Pos remove(LIST L, Pos P)

Pos $T \leftarrow prev(L, P)$;
$T.next \leftarrow P.next$;
delete P;
Risultato T

Liste con puntatore doppio

In questo caso ciascun nodo della lista ha tre componenti:

1. Un campo `value`;
2. Un campo `next` che contiene il puntatore al prossimo elemento;
3. Un campo `prev` che contiene il puntatore all'elemento precedente;



Il vantaggio evidente è che il metodo `prev` ha ora costo $O(1)$.

Metodi per LISTA

Pos prev(Pos P)

Pos $T \leftarrow P$;
if $T \neq$ **nil then**
  $T \leftarrow T.prev$;
Risultato T

Pos remove(LIST L, Pos P)

Pos $T \leftarrow prev(L, P)$;
if $T \neq$ **nil then**
  $T.next \leftarrow P.next$;
else
  $L.H \leftarrow P.next$;
if $P.next =$ **nil then**
  $L.T \leftarrow T$;
delete P;
Risultato T

Pos insert(Pos P, Item v)

Pos *new* T;
$T.value \leftarrow v$;
$T.next \leftarrow P.next$;
Pos $X \leftarrow prev(P)$;
$P.next \leftarrow T$;
$T.next \leftarrow X$;
if $L.H =$ **nil then**
  $L.H \leftarrow T$;
if $T.next =$ **nil then**
  $L.T \leftarrow T$;
Risultato T

Liste con vettori

Se implementiamo una lista con un vettore, la posizione corrisponde all'indice dell'elemento nel vettore.
In questo caso la lista conterrà:

1. Un vettore V atto a contenere i valori;
2. La dimensione del vettore D;
3. Il numero di elementi attualmente presenti K, ovvero l'indice dell'ultimo elemento attualmente occupato S (con valore iniziale -1).

Con questa implementazione, si può rappresentare la *coda* della lista come

---

if $L.S \geq 0$ **then**
  Risultato $L.V[L.S]$
else
  Risultato nil

Implementazione con vettori

Problema fondamentale N. 1: *L'operazione di inserimento in una posizione arbitraria costa* $O(N)$

Infatti occorre traslare i contenuti del vettore

Function Pos insert(List L, Pos P, Item v)

for $i = P$ **to** $L.S$ **do**
  $L.V[i+1] \leftarrow L.V[i]$;
$L.V[P] \leftarrow v$;
$L.S \leftarrow L.S + 1$;
Risultato P

in dipendenza dal valore di P.

Liste con vettori

Supponiamo allora che gli inserimenti avvengano sempre *in coda*, ossia $P = L.S + 1$; in questo caso sembrerebbe che il costo possa essere $O(1)$.

Supponiamo allora che gli inserimenti avvengano sempre *in coda*, ossia $P = L.S + 1$; in questo caso sembrerebbe che il costo possa essere $O(1)$.

Implementazione con vettori

Problema fondamentale N. 2: *Occorre gestire la dimensione del vettore V*

Infatti abbiamo

Function Pos insert(List L, Pos P, Item v)

if $L.S = L.D - 1$ **then**
    choose NS;
    $L.V \leftarrow \text{realloc}(L.V, L.D, NS)$;
    $L.D \leftarrow NS$;
$L.V[L.S] \leftarrow v$;
$L.S \leftarrow L.S + 1$;
Risultato $L.S - 1$

In assenza di eventi di riallocazione, il costo è effettivamente $O(1)$.

Consideriamo ora l'operazione di `realloc`.

Function Vector realloc(Vector V, int *oldsize*, int *newsize*)

Vector new T, *newsize*;
for $i = 0$ **to** $\min(oldsize, newsize)$ **do**
  $T[i] \leftarrow V[i]$;
Risultato T

Il costo è lineare in $\min(newsize, oldsize)$.

Liste con vettori

Consideriamo ora l'operazione di `realloc`.

Function Vector realloc(Vector V, int *oldsize*, int *newsize*)

Vector new T, *newsize*;
for $i = 0$ **to** $\min(oldsize, newsize)$ **do**
  $T[i] \leftarrow V[i]$;
Risultato T

Il costo è lineare in $\min(newsize, oldsize)$.

Problema: come scegliere *newsize*?

La scelta più ovvia è

$newsize \leftarrow oldsize + d$;

Consideriamo ora il costo medio di N inserimenti con incremento d costante:

- gli eventi di riallocazione avverranno ogni d inserimenti;
- Ad ogni evento di riallocazione $i = d \cdot k$ si pagherà un costo lineare $d \cdot (k-1)$;

Pertanto il costo di $N = d \cdot P$ inserimenti sarà

$$\sum_{k=1}^{P} d \cdot (k-1) = d \sum_{k=0}^{P-1} k = d \cdot \frac{P \cdot (P-1)}{2} \approx d \cdot \frac{P^2}{2} = \frac{N^2}{2d} = O(N^2)$$

e quindi il costo *medio* del singolo inserimento è

$$O(N).$$

Una alternativa

Scegliamo ora

$$newsize = 2 \cdot oldsize.$$

In questo caso il numero di riallocazioni k dovrà essere tale che

$$2^k \geq N \Rightarrow k \geq \log(N);$$

ogni evento di riallocazione avrà un costo lineare, certamente non superiore a N, quindi il costo totale sarà non superiore a

$$O(N \log(N)),$$

e il costo *medio* di ciascun inserimento sarà

$$\log(N) \ll N.$$

Pile

Una *pila* (in inglese: *stack*) è un particolare tipo di lista in cui

Sia gli inserimenti che le cancellazioni avvengono ad un estremo della struttura dati, ovvero:

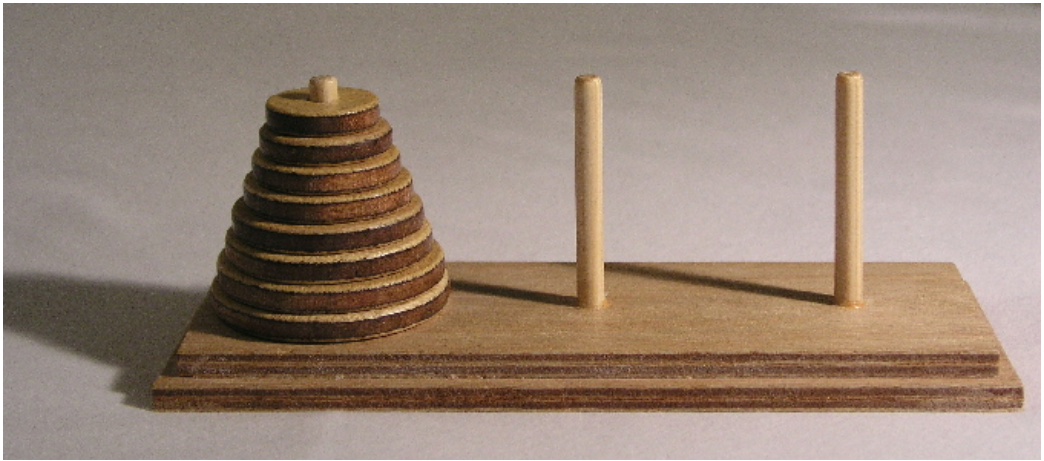
- *Gli inserimenti avvengono solo dopo l'ultimo elemento;*
- *La cancellazione avviene solo sull'ultimo elemento.*

Un esempio di uso delle pile si ha nella gestione delle procedure ricorsive.
Nella terminologia comune, quando si ha a che fare con pile si usano i nomi:

- `push`(P, v) per l'operazione di inserimento;
- `pop`(P) per l'operazione di cancellazione (che restituisce il contenuto dell'elemento cancellato);
- `top`(P) per l'operazione che legge l'elemento in cima alla pila.

Esercizio: scrivere le procedure `push` e `pop` adattando le procedure di lista `insert` e `delete`.

Pile: torre di Hanoi



Function hanoi-ricorsiva(int n, int *orig*, int *dest*, int *med*)

if $n = 1$ **then**
    % trasferisci un disco da *orig* a *dest*;
else
    hanoi-ricorsiva($n-1$,*orig*,*med*,*dest*);
    % trasferisci un disco da *orig* a *dest*;
    hanoi-ricorsiva($n-1$,*med*,*dest*,*orig*,);

Questo algoritmo ha una complessità esponenziale (per il teorema sulle ricorrenze lineari):

$$T(n) = \begin{cases} 1 & \text{per } n = 1, \\ 2T(n-1)+1 & \text{per } n > 1. \end{cases}$$

Pile: torre di Hanoi

Function hanoi-iterativa(int n, int *orig*, int *dest*, int *med*)

```
Stack S ← Stack();
boolean finedelmondo ← false;
while not finedelmondo do
    while n ≠ 1 do
        S.push(n, orig, dest, med);
        n ← n − 1;
        dest ↔ med;
    %Trasferisci un disco da orig a dest;
    if S.isEmpty() then
        finedelmondo ← true;
    else
        (n, orig, dest, med) ← S.pop();
        %Trasferisci un disco da orig a dest;
        n ← n − 1;
        orig ↔ med;
```

Code

Il tipo di dato *coda* (in inglese *queue*) è una specializzazione della lista in cui

- Gli eventi di inserimento avvengono solo ad un estremo della lista (prima della testa, o dopo la coda);
- Gli eventi di cancellazione avvengono solo all'altro estremo (alla coda, oppure alla testa).

In questo contesto le procedure prendono i nomi

- `enqueue`(Q, v) per l'operazione di inserimento;
- `dequeue`(Q) per l'operazione di cancellazione (che restituisce il contenuto dell'elemento cancellato).

Esercizio: scrivere le procedure `enqueue` e `dequeue` adattando le procedure di lista `insert` e `delete`.

Nota bene: in inglese non ci sono ambiguità potendosi usare le parole *queue* e *tail* per due concetti che in italiano vengono entrambi resi con *coda*