Le strutture informative: gli alberi

di Anna Pugliese

seconda parte

Dopo aver dedicato il precedente numero della rubrica agli aspetti topologici delle strutture dati ad albero, eccoci a parlare di quegli aspetti più algoritmici che riguardano le strategie di allocazione in memoria e gli esempi pratici di utilizzo. In considerazione dell'ampio spazio dedicato la volta scorsa alla teoria degli alberi, sarà il caso di dare a questa trattazione un taglio molto pratico

Vedremo dei precisi metodi di memorizzazione e riporteremo algoritmi di visita, e di uso in genere, degli alberi, esprimendo questi ultimi in un linguaggio di programmazione particolarmente adatto allo scopo: il linguaggio C. Non si scoraggino i non particolarmente ferrati nella programmazione in C: cercheremo di spiegare, dove sarà il caso di farlo, cosa si nasconde dietro tutto ciò che, in C, appare sintatticamente mostruoso (nota di adp: bene, finalmente qualcuno che ha avuto il coraggio di gridarlo in pubblico; aggiungerei che talvolta il «C» riesce ad essere perfino «semanticamente mostruoso»!).

Alcune strategie di memorizzazione

Dovrebbe essere sufficientemente chiaro, per coloro che hanno seguito fin qui la trattazione delle strutture informative presentata negli ultimi numeri della rubrica e/o per coloro i quali possiedono comunque delle basi in materia, che il concetto fondamentale sul quale poggiano le tecniche di allocazione degli alberi, e più in generale quelle delle strutture a lista, è il puntatore. Dietro questo concetto è nascosta la possibilità di trattare come «dato» qualcosa che sta invece ad un livello di astrazione inferiore a quello dei dati in sé, e cioè l'indirizzo cui i dati risiedono, in memoria. Vediamo allora come, mediante l'impiego di puntatori, è possibile organizzare in memoria l'informazione presente in un albero.

Sia A il nostro albero. Un qualsiasi nodo N di A, può essere memorizzato in una lista il cui primo elemento contiene l'informazione associata al nodo N e gli elementi successivi, in numero pari ai nodi figli di N, contengono puntatori alle liste in cui ognuno di tali figli è memorizzato. La figura 1 mostra l'applicazione di quanto detto, su un nodo E avente quattro figli.

A questo punto è evidente che la lista contenente l'intero albero A, può essere considerata quella che si ottiene a partire dalla lista associata alla sua radice R, che chiameremo L(R). In figura 2b è riportata l'intera lista multipla contenente l'albero di figura 2a.

La strategia di allocazione proposta può essere perfezionata per dar vita ad una nuova strategia che permette un discreto risparmio di occupazione di memoria. L'idea è quella di risparmiare un elemento di lista per ogni nodo foglia dell'albero, inserendo l'informazione contenuta in tale nodo foglia, direttamente nell'elemento della lista del padre che in figura 2b contiene invece il puntatore alla sua lista di figlio. La cosa è esemplificata in figura 3, dove è ripor-

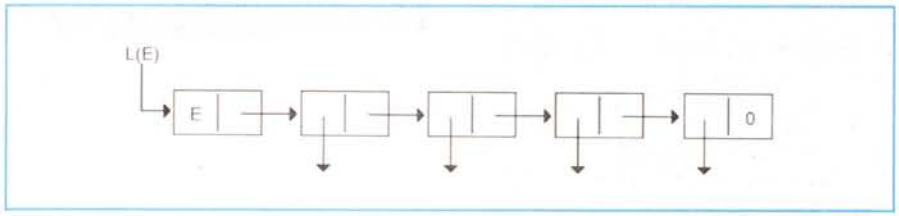

Figura 1 - Lista contenente un nodo di un albero.

tata la lista multipla contenente l'albero di figura 2a memorizzato con la nuova strategia.

Al fine di dimostrare che le strategie di memorizzazione degli alberi possono essere inventate a chili, diamo un'occhiata alla figura 4. La prima cosa che salta all'occhio, da una pur fugace osservazione, è che i singoli elementi della lista multipla di figura 4 sono strutturalmente diversi da quelli visti sinora: essi sono strutture con tre campi, invece di due, due dei quali contengono elementi di tipo puntatore. Il significato dei tre campi è il seguente: il primo campo contiene, al solito, l'informazione associata al nodo che l'elemento rappresenta, il secondo campo contiene un puntatore all'elemento rappresentante il suo primo figlio ed, infine, il terzo campo punta al resto dei suoi fratelli, dove per resto si intendono i fratelli che seguono (ad esempio da sinistra a destra) il nodo in questione. Cerchiamo di capire in base a che cosa le diverse strategie vanno confrontate. Chiamiamo A la strategia esemplificata dalle figure 1 e 2b, B e C le strategie di figura 3 e 4 rispettivamente. Abbiamo già detto dell'evidente risparmio in termini di occupazione di memoria, ottenibile con la strategia B rispetto alla A. La strategia C presenta, con riferimento al solo esempio in questione, un risparmio di memoria ulteriore, ma solo fittizio. In altri termini essa impiega 11 elementi di lista contro i 15 della B ed i 21 della A. Ma gli 11 elementi comprendono in tutto 22 puntatori ed 11 informazioni (essendo 3 i campi associati ad ogni elemento di lista), contro i 15 puntatori e le 15 informazioni della strategia B. È chiaro che una corretta valutazione della memoria occupata nei due casi deve tener conto del tipo di informazione associata ad ogni nodo, della grandezza dei puntatori, ed infine della struttura dell'albero da memorizzare: probabilmente troppi fattori per esprimere un valido giudizio in forma generale. Allora, se ci trovassimo nella necessità di dover scegliere fra le strategie B e C, come agiremmo? La risposta giusta è quella che non si limita a considerare il problema dal punto di vista dell'occupazione di memoria, ma tiene conto degli altri fattori in gioco; il più importante fra i fattori che probabilmente staranno giocando quando ci troveremo a scegliere come memorizzare i nostri alberi, è senza dubbio il tipo di operazioni che vogliamo eseguire sull'albero stesso: queste influenzano la complessità dello scrivere programmi che realizzano tali operazioni. Spieghiamoci meglio.

Supponiamo di voler costruire la funzione È_UNA_FOGLIA(p), che restituisce 0 se l'elemento di lista puntato da p corrisponde ad un nodo intermedio dell'albero ed un numero diverso da 0 se corrisponde invece ad un elemento foglia. Se la strategia adottata è la C questa funzione non deve far altro che leggere il valore del secondo elemento della struttura puntata da p, se questi è 0 la risposta è diversa da zero, se questi è diverso da zero la risposta è zero, vale a dire che la risposta è l'opposto del valore di questo elemento. Guardando la figura 3 e la figura 2b, non ci vuole molto a capire che l'implementazione di È_UNA_FOGLIA(p) è più complessa con le strategie B e A, di quanto non lo sia quella appena vista. Nella successiva sezione mostreremo l'effettiva implementazione in linguaggio C di ognuna delle tre versioni della funzione per poter meglio confrontare i diversi casi. Ma allora, la strategia C è la migliore? Se dovessimo usare gli alberi solo per interrogarci su quali nodi sono foglie e quali no, probabilmente sì. Ma pensiamo un attimo ad un algoritmo di visita dell'albero in ordine differito; come la mettiamo? Ad essere sinceri non lo so

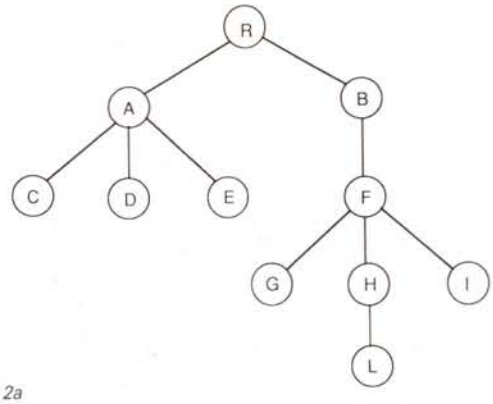

Figura 2a

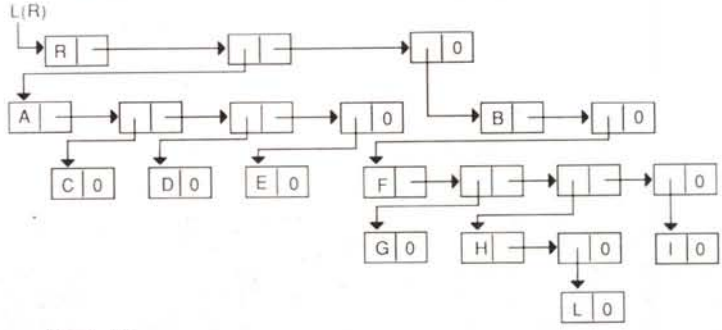

Figura 2 b

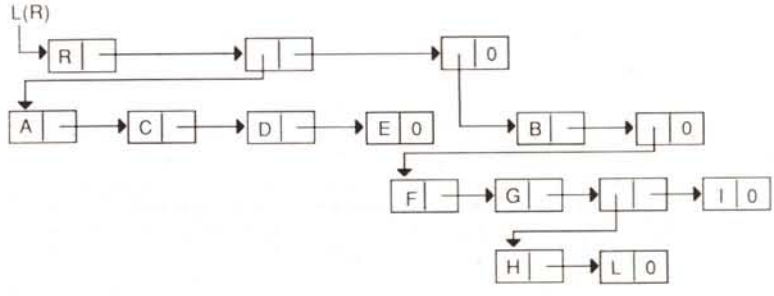

Figura 3

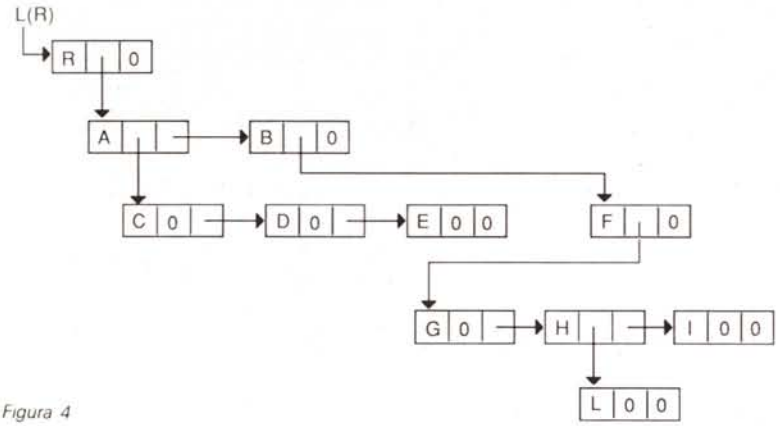

Figura 4

neanche io, visto che non ho ancora provato a scriverne gli algoritmi, ma siccome conto di farlo prima di sera, ci sono buone possibilità che voi possiate trovare abbozzi di risposta nella successiva sezione, dove, credo, andrò a riportarli. Prima di abbandonare la presente sezione dedicata alle strategie di memorizzazione, è il caso di parlare un attimo del caso degli alberi binari, esistendo per essi una particolare strategia che, sfruttando la caratteristica di tali alberi, ne permette una efficiente gestione. Ricordiamo che gli alberi binari, sono dei casi (quasi) particolari di alberi sottostanti al vincolo che ogni nodo deve avere esattamente due figli: il figlio destro ed il figlio sinistro (quindi sono alberi non liberi ed ordinati), oltre al fatto che un albero binario può anche essere vuoto, mentre un normale albero no, il che è pura speculazione accademica in quanto, come vedremo, gli alberi sono sempre implementati prevedendo l'albero vuoto come caso particolare. La strategia di memorizzazione di alberi binari, sopra citata, è quella schematizzata mediante la lista multipla, riportata in figura 5b, che corrisponde all'albero binario di figura 5a.

Esempi di utilizzo degli alberi

Per ragioni, perdonatemi il termine, didattiche, è conveniente che la presentazione degli algoritmi di manipolazione degli alberi, venga fatta cominciando dal caso degli alberi binari, per poi passare agli alberi in genere. Detto questo, cominciamo illustrando le definizioni necessarie. Innanzitutto occorre definire il tipo cui appartengono gli elementi della lista multipla rappresentante un generico albero binario (vedi tabella A).

Ogni elemento contiene dunque tre campi. Sui campi «left» e «right», indicanti inequivocabilmente i due figli del nodo in questione, c'è poco da dire: essi sono dichiarati essere di tipo «nodo *» che nel linguaggio C sta ad indicare i puntatori ad elementi di tipo nodo. Il campo «info» è invece stato definito di un tipo ragionevolmente scelto fra i tanti, e può essere chiaramente sostituito con il tipo che più ci aggrada; per ora, consideriamolo come una stringa, che in C è costituita da un puntatore a carattere, lunga 20.

Tanto per assaggiare, diamo un'occhiata alla funzione seguente: ▼

```
int E'_UNA_FOGLIA (NODO * x)
{
  return (x->left==0)&&(x->right==0);
}
```

che è sicuramente più comprensibile dopo aver saputo che in C l'operatore di uguaglianza si indica con il simbolo «==», che il simbolo «&&» rappresenta l'operazione logica «and» e che dato un puntatore «p» ad una struttura «s», «p->field» indica il campo field di s, senza trascurare, infine, che in C le operazioni di confronto restituiscono un intero, vale a dire 0 per falso e non 0 (solitamente -1 oppure 1) per vero.

Supponiamo adesso di voler contare il numero di nodi presente in un certo sottoalbero «subtree», dove evidentemente, se subtree coincide con la radice dell'albero, il risultato è il numero di nodi dell'intero albero. Riflettiamo allora sul fatto che problematiche di questo genere, presuppongono l'abilità di considerare ogni nodo come «già contato» oppure no, il che può essere risolto solo se siamo capaci di girare fra i nodi di un albero sapendo se il nodo sul quale ci troviamo è già stato incontrato o se è la prima volta che ci finiamo sopra. Molti avranno già capito che la soluzione a questo problema richiede l'uso di un algoritmo di «visita» dell'albero. Mi viene in mente allora, che potrebbe essere molto utile, e non solo per risolvere il problema del conteggio, una procedurina che ricevuti come parametri il puntatore all'albero ed un puntatore ad una generica funzione, di tipo ben definito si capisce, visita l'albero, diciamo in ordine anticipato, e chiama su ogni nodo visitato la funzione che gli è stata passata.

Vediamo un po' (figura A). In tutta onestà, non immaginavo proprio che ne uscisse fuori una funzione così semplice, avevo dimenticato evidentemente la potenza della ricorsione. Eh sì! Gli alberi

```
typedef struct nodo {   char *          info[20];
                        nodo *          left;
                        nodo *          right;   } NODO;
```

Tabella A

```
void exec_fun_on_tree (NODO *subtree, void (*fun)())
{
              /* Visita l'albero puntato da subtree **
              **   in ordine binario anticipato     **
              **    chiamando su ogni nodo la       **
              **     funzione passatagli fun        */

   fun(subtree);
   if (subtree->left!=0) exec_on_tree(subtree->left,fun);
   if (subtree->right!=0) exec_on_tree(subtree->right,fun);
}
```

Figura A

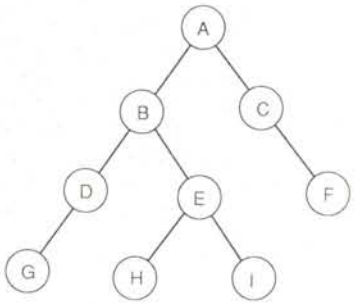

Figura 5a

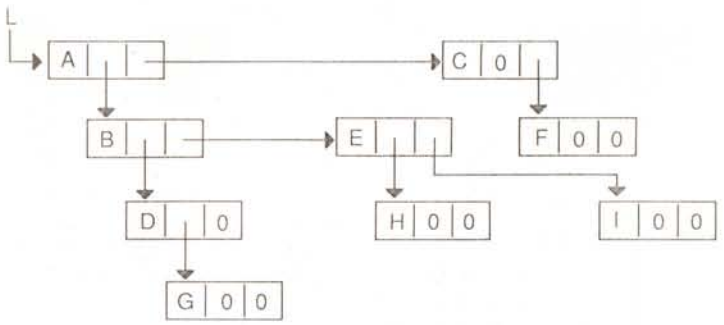

Figura 5b

sono strutture estremamente ricorsive, ed è con procedure ricorsive che conviene andarci ad operare. Ma bando alle ciance. Non vi vedo tutti convinti della funzionalità di exec_on_tree. Parliamone. Cominciamo col dire che il simbolo «!» indica l'operazione logica «not» e che, quindi «!=» significa «diverso da».

I parametri che la funzione ha a disposizione, sono un puntatore ad un nodo (un elemento di lista multipla: vale a dire una struttura di tipo NODO), ed un puntatore ad una funzione. In C, i nomi delle funzioni denotano essi stessi dei puntatori alla funzione. L'istruzione «fun(subtree)» è una più che legittima invocazione della funzione fun su subtree. Certo, a giudicare dal tipo cui appartiene «fun» per quanto si evince dalla sua dichiarazione come parametro formale di exec_on_tree, non si direbbe che essa abbia ulteriori parametri, ma in C non è necessario specificare il tipo dei parametri di una funzione quando si sta definendo la funzione stessa come parametro. È evidente che l'effettiva dichiarazione della funzione che verrà data come parametro formale ad exec_on_tree, deve dichiarare che il suo unico parametro è di tipo NODO *.

Vediamo allora la dichiarazione della funzione conta_nodi che volevamo implementare inizialmente:

```
void conta_nodi (NODO *p)
{
Totale++;
}
```

Probabilmente a qualcuno di voi potrà sembrare che io stia abusando della sua pazienza, ma si sbaglia. Il corpo di conta_nodi è esattamente quello, posto ovviamente che la variabile Totale sia stata definita come variabile statica ed inizializzata a zero, il che è realizzato mediante:

```
static int Totale;
```

e tramite la funzione:

```
void AZZERA()
{ Totale=0; }
```

Se volessimo allora contare i nodi dell'albero puntato da:

```
AZZERA();
exec_on_tree(radice_albero, conta_nodi);
```

Qualcuno potrebbe allora osservare che sarebbe stato molto più semplice inserire l'operazione di incremento della variabile statica Totale, direttamente all'interno della funzione exec_on_tree; ma quando dicevo che exec_on_tree ci sarebbe stata utile non solo per il problema di contare i nodi, mi riferivo proprio alla sua generalità. Consideriamo al proposito, la funzione seguente:

```
void visualizza_informazione(NODO *p)
{
    printf("Nominativo :  %s ",p->info);
}
```

che chiama la funzione C «printf», di scrittura formattata, che stampa a video la stringa passatagli come primo parametro sostituendo tutte le occorrenze di «%s» con la stringa puntata dal secondo parametro. In pratica scrive a video il contenuto del campo info della struttura *p che gli viene passata. Bene, la seguente chiamata:

```
exec_on_tre(radice_albero,visualizza_informazione);
```

fa sì che vengano visualizzate tutte le informazioni contenute nell'albero puntato da radice_albero.

Dovrebbe risultare chiaro che, con la funzione exec_on_tree a disposizione, è proprio il caso di dire «chi più ne ha più ne metta» riferendoci alle funzioni che gli possono essere passate. Chi di voi ha già in mente di andare a provare sul suo computer la funzionalità di questi esempi sugli alberi, può verificare direttamente la generalità della funzione exec_on_tree. Prima, però, che sia possibile rendere operativi gli esempi riportati, è necessario spendere due parole su come si alloca memoria per gli elementi della lista multipla che implementa l'albero binario. A venirci incontro, sono le due seguenti funzioni del linguaggio C. La prima serve a richiedere l'allocazione dinamica di un certo numero di byte di memoria:

```
char * malloc (unsigned size);
```

che restituisce il puntatore alla memoria allocata, visto come puntatore ad una sequenza di caratteri. La seconda, che in realtà è un operatore del C e non una funzione, serve a calcolare, in byte, la grandezza di una variabile o di un tipo:

```
sizeof(object);
```

Vediamo allora, a mo' di esempio, come può essere realizzata una funzione che aggiunge elementi ad un albero binario (vedi figura B). La funzione ha l'evidente handicap di costruire alberi completamente sbilanciati, ma funziona. Due parole per spiegare le novità:

```
void append_to_tree (NODO *p, char *inf)
{
        NODO *pun;
        pun=p;
        while (pun->left!=0)   { pun=pun->left; }
        pun->left = (NODO *) malloc(sizeof(NODO));
        pun=pun->left;
        strcpy(pun->info,inf);
}
```

Figura B

«(NODO *)» è stato preposto all'invocazione della malloc per cambiare il tipo di oggetto da assegnare a pun; malloc infatti restituisce un puntatore a carattere mentre pun è un puntatore a NODO. Questo tipo di forzatura dei tipi, in C prende il nome di «cast». La funzione strcpy è anch'essa una funzione standard C, come printf, ed è definita come:

```
char strcpy (char *s1, char *s2);
```

essa copia la stringa puntata da s2 nella memoria puntata da s1 e restituisce la stringa s1 (della quale non sappiamo che farcene e la lasciamo al compilatore che si «arrabbierà» con un messaggio di «WARNING»).

Per deallocare la memoria nel caso in cui dovessimo rimuovere un elemento da un albero, basta usare la funzione:

```
void free (char *ptr);
```

che libera, restituendola al sistema operativo, la memoria puntata da ptr, che era precedentemente stata allocata con malloc (che ne ricorda la grandezza).

Siamo così giunti alla fine di questo articolo, e non c'è rimasto il tempo di dare un'occhiata agli alberi non binari; ma ogni promessa è un debito, ed avendovi promesso di riportare l'algoritmo di visita differita di un albero allocato tramite la strategia di memorizzazione illustrata in figura 4, eccomi costretta ad oltrepassare la mezzanotte per poter scrivere la seguente procedura (confido nella potenza della ricorsione che mi farà risparmiare tempo e... sonno). Ma ad un patto. Le spiegazioni un'altra volta (questo articolo non può diventare un manuale C!).

```
typedef struct nodo2 {
            int       numeric_info;
            nodo *    primo_figlio;
            nodo *    lista_dei_fratelli;
} NODO2;
```

```
void exec_fun_on_tree (NODO2 *subtree, void (*fun)())
{
    if (subtree->primo_figlio!=0)
            exec_on_tree(subtree->primo_figlio,fun);
    if (subtree->lista_dei_fratelli!=0)
            exec_on_tree(subtree->lista_dei_fratelli,fun);
    fun(subtree);
}
```

Figura C

È la nuova struttura che ci serve per gli elementi della lista multipla.

Mentre la funzione exec_on_tree (che incorpora, questa volta l'algoritmo di visita differita) sugli alberi non binari memorizzati come in figura 4 è rappresentata in figura C.

Speriamo che funzioni!

A risentirci.

MC