15.6 Implementazione di alberi

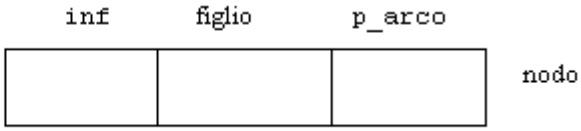
Per memorizzare gli alberi esistono varie possibilità. Una prima opzione potrebbe consistere nella definizione di una struttura simile a quella usata per l'albero binario, dove, oltre alla parte informazione, sono presenti tanti campi puntatore quanti ne sono necessari per far riferimento alle radici dei sottoalberi del nodo che ne ha di più:

```
struct nodo {
       char inf;
       struct nodo *p_1sott_albero;
       struct nodo *p_2sott_albero;
       ...
       struct nodo *p_nsotto_albero;
     };
```

Questa ipotesi presuppone la conoscenza del massimo numero di archi che si possono staccare da un nodo, e comporta inoltre uno spreco di memoria.

Un'altra possibilità è utilizzare liste multiple, in cui per ogni nodo dell'albero si forma una lista: il primo elemento della lista contiene l'etichetta del nodo, gli elementi successivi sono in numero uguale ai nodi figli. In ciascuno di essi viene memorizzato il puntatore alle liste di tali nodi.

```
struct nodo {
   char inf;
   struct nodo *figlio;
   struct nodo *p_arco;
};
```



Il campo `inf` viene utilizzato soltanto per il primo degli elementi di ogni lista, in cui il campo `p_arco` punta al secondo elemento della lista e `figlio` non viene utilizzato. Per gli altri elementi della lista `figlio` viene utilizzato per far riferimento alla lista del figlio e `p_arco` per puntare al successivo elemento della lista. In Figura 15.4 viene presentata la memorizzazione dell'albero di Figura 15.3.

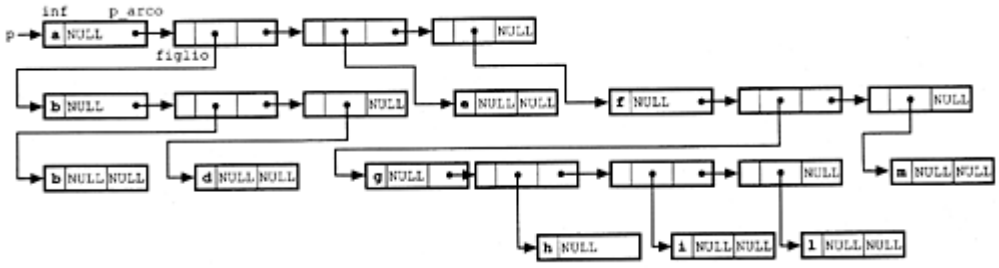


Figura 15.4 Memorizzazione dell'albero di Figura 15.3

Naturalmente questa scelta comporta un certo spreco di memoria, poiché soltanto due dei tre campi vengono utilizzati da ogni elemento allocato in memoria. Una soluzione alternativa è utilizzare due tipi di strutture: uno per il primo nodo di ogni lista, composta da un campo `inf` per l'etichetta del nodo e da un campo puntatore al secondo elemento della lista; un secondo per gli elementi successivi di ogni lista, composta da un campo puntatore alla lista di ogni figlio e da un campo puntatore al successivo elemento della lista:

```
struct nodo {
       char inf;
       struct lista *lista_figli;
     };
```

```
struct lista {
       struct nodo  *figlio;
       struct lista *successore;
     };
```

Per semplificare gli algoritmi di creazione e visita scegliamo di usufruire di un solo tipo di elemento.

Consideriamo ora il problema di memorizzare un albero immesso dall'utente in forma parentetica anticipata in una lista multipla e visitare in ordine anticipato l'albero costruito.

Nel Listato 15.4 è riportato il programma completo. Supponiamo per esempio che la stringa in ingresso sia quella relativa all'albero di Figura 15.3. La funzione `albero` legge il primo carattere della stringa in ingresso, che

necessariamente è una parentesi tonda aperta e chiama `crea_albero`. La funzione `crea_albero` alloca spazio per il primo nodo, legge il secondo carattere immesso e lo inserisce.
Viene poi letto un ulteriore carattere. Successivamente inizia un ciclo che va avanti finché il carattere in esame è uguale a parentesi tonda aperta, in pratica finché ci sono sottoalberi. Se l'albero fosse composto solamente dalla radice, a questo punto il procedimento avrebbe termine con il bloccaggio della lista inserendo `NULL` nel campo `p_arco` dell'elemento creato.
Nel ciclo viene allocato spazio per un successivo elemento, che questa volta verrà utilizzato per puntare al figlio, e viene chiamata ricorsivamente `crea_albero` passandole in ingresso il campo figlio dell'elemento creato. La nuova chiamata della funzione legge il successivo carattere della stringa, che necessariamente è la radice del sottoalbero, e così via finché ci sono figli.

```
/* Creazione di un albero e visita in ordine anticipato.
   L'albero viene immesso dall'utente informa parentetica
   anticipata. L'etichetta dei nodi è un carattere.
   L'albero è implementato con liste multiple             */

#include <stdio.h>
#include<stddef.h>

struct nodo {
  char inf;
  struct nodo *figlio;
  struct nodo *p_arco;
};

struct nodo *albero();
struct nodo *crea_albero(struct nodo *);
void anticipato(struct nodo *);

main()
{
struct nodo *radice;
radice = albero();                /* Creazione dell'albero */
printf("\nVISITA IN ORDINE ANTICIPATO\n");
anticipato(radice);
}


/* Legge il primo carattere della rappresentazione parentetica
   e invoca la funzione crea_albero()                         */
struct nodo *albero()
{
struct nodo *p = NULL;
char car;

printf("\nInserisci la rappresentazione dell'albero: ");
scanf("%c", &car);
p = crea_albero(p);
return(p);   /* Ritorna il puntatore alla radice al chiamante */
}


/* Crea un nodo e vi inserisce la relativa etichetta. Per ogni figlio crea un
arco. Invoca ricorsivamente se stessa          */
struct nodo *crea_albero(struct nodo *p)
{
struct nodo *paus;
char car;
```

```
/* crea il nodo */
p = (struct nodo *) malloc( sizeof( struct nodo ) );
scanf("%c", &p->inf);       /* Inserimento del valore nel nodo */
paus = p;
scanf("%c", &car);          /* Lettura del carattere successivo */

while(car=='(') {           /* Per ogni figlio ripeti */
  /* crea l'arco */
  paus->p_arco = (struct nodo *) malloc(sizeof(struct nodo));
  paus = paus->p_arco;
 /* Invoca se stessa passando il campo figlio dell'elem. creato */
  paus->figlio = crea_albero(paus->figlio);

  scanf("%c", &car);        /* Lettura del carattere successivo */
}

paus->p_arco = NULL;
return( p );   /* Ritorna il puntatore alla radice al chiamante */
}


/* Visita ricorsivamente l'albero in ordine anticipato */
void anticipato(struct nodo *p)
{
printf("(%c", p->inf);
p = p->p_arco;

while(p!=NULL) {
  anticipato(p->figlio);
  p = p->p_arco;
}
printf(")");
}
```

Listato 15.4 Creazione di un albero a partire dalla sua rappresentazione parentetica