22.1 Alberi binari

Strutture non lineari

Le strutture dati che abbiamo fin qui esaminato sono lineari: ogni elemento della sequenza ha un successore e un predecessore, fatti salvi il primo e l'ultimo. Possiamo però pensare a qualcosa di più generale, in cui la relazione tra gli elementi della struttura non sia lineare.

Nodi Archi Etichette

Nel seguito utilizzeremo il termine *nodo* con il significato di elemento e il termine *arco* per indicare una connessione tra due nodi; con *etichetta* faremo invece riferimento al valore rappresentativo di ogni nodo.

Alberi binari

Una delle strutture dati più note è l'*albero binario*, definito come un insieme B di nodi con le seguenti proprietà:

Radice

- B è vuoto oppure un nodo di B è scelto come radice;
- i rimanenti nodi possono essere ripartiti in due insiemi disgiunti B_1 e B_2, essi stessi definiti come alberi binari.

Sottoalberi

B_1 e B_2 sono detti *sottoalberi* della radice. È importante sottolineare che il sottoalbero sinistro B_1 è distinto dal sottoalbero destro B_2 e questa distinzione permane anche se uno dei due è vuoto. La ricorsività della definizione stessa è evidente.

Un esempio di albero binario è presentato in **Figura 22.1**. Il nodo designato come radice ha etichetta 104, e da esso si dipartono il sottoalbero sinistro con radice 32 e il sottoalbero destro con radice 121. Si dice che i nodi con etichette 32 e 121 sono *fratelli* e sono rispettivamente il figlio sinistro e il figlio destro del nodo con etichetta 104. L'albero che ha come radice 32 ha ancora due sottoalberi con radici 23 e 44, i quali non hanno figli o, in altre parole, hanno come figli alberi vuoti. L'albero con radice 121 non ha figlio sinistro e ha come figlio destro il sottoalbero con radice 200, il quale a sua volta ha come figlio sinistro il sottoalbero con radice 152 e non ha figlio destro. Il sottoalbero che ha come radice 152 non ha figli.

Foglie

I nodi da cui non si dipartono altri sottoalberi (non vuoti) sono detti *nodi terminali* o *foglie*. Nell'esempio sono quelli etichettati con 23, 44 e 152. Si chiamano *visite* le scansioni dell'albero che portano a percorrerne i vari nodi. L'ordine in cui questo avviene distingue differenti tipi di visite.

Visite

La visita *in ordine anticipato* di un albero binario viene effettuata con il seguente algoritmo:

Ordine anticipato

```
anticipato( radice dell'albero )
Se l'albero non è vuoto:
    Visita la radice
    anticipato( radice del sottoalbero sinistro )
    anticipato( radice del sottoalbero destro )
```

in cui abbiamo evidenziato la ricorsività della visita. Nel caso dell'albero di **Figura 22.1** la visita in ordine anticipato dà la sequenza: 104, 32, 23, 44, 121, 200, 152.

La visita *in ordine differito* invece è così descritta:

Ordine differito

```
differito( radice dell'albero )
Se l'albero non è vuoto:
    differito( radice del sottoalbero sinistro )
    differito( radice del sottoalbero destro )
    Visita la radice
```

Applicando il procedimento sull'albero di esempio abbiamo: 23, 44, 32, 152, 200, 121, 104.

Concludiamo con un terzo tipo di visita, *in ordine simmetrico*, il cui algoritmo è:

Ordine simmetrico

simmetrico(radice dell'albero)
Se l'albero non è vuoto:
  simmetrico(radice del sottoalbero sinistro)
  Visita la radice
  simmetrico(radice del sottoalbero destro)

che restituisce, con riferimento all'esempio di ***Figura 22.1***: 23, 32, 44, 104, 121, 152, 200.

22.2 Implementazione di alberi binari

Consideriamo ora il seguente problema: memorizzare i dati interi immessi dall'utente in un albero binario tale che il valore dell'etichetta di un qualsiasi nodo sia maggiore di tutti i valori presenti nel suo sottoalbero sinistro e minore di tutti i valori del suo sottoalbero destro. Per esempio, se l'utente immette in sequenza 104, 32, 44, 121, 200 152, 23, un albero conforme con la definizione è quello di ***Figura 22.1***. La sequenza di immissione termina quando l'utente inserisce il valore zero; in caso di immissione di più occorrenze dello stesso valore, soltanto la prima verrà inserita nell'albero. Si chiede di visitare l'albero in ordine anticipato.

Una soluzione è ottenuta definendo ciascun nodo come una struttura costituita da un campo informazione, contenente l'etichetta, e due puntatori al sottoalbero sinistro e al sottoalbero destro:

```
struct nodo {
   int inf;
   struct nodo *albSin;
   struct nodo *albDes;
};
```

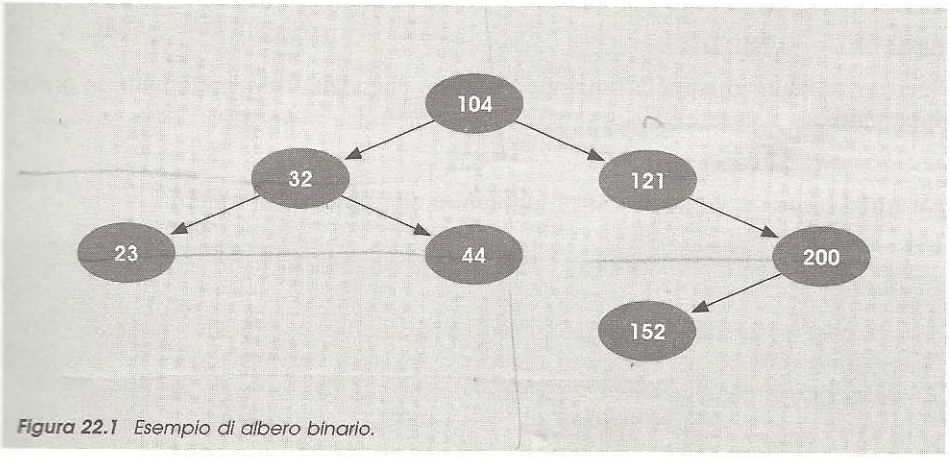

Figura 22.1 *Esempio di albero binario.*

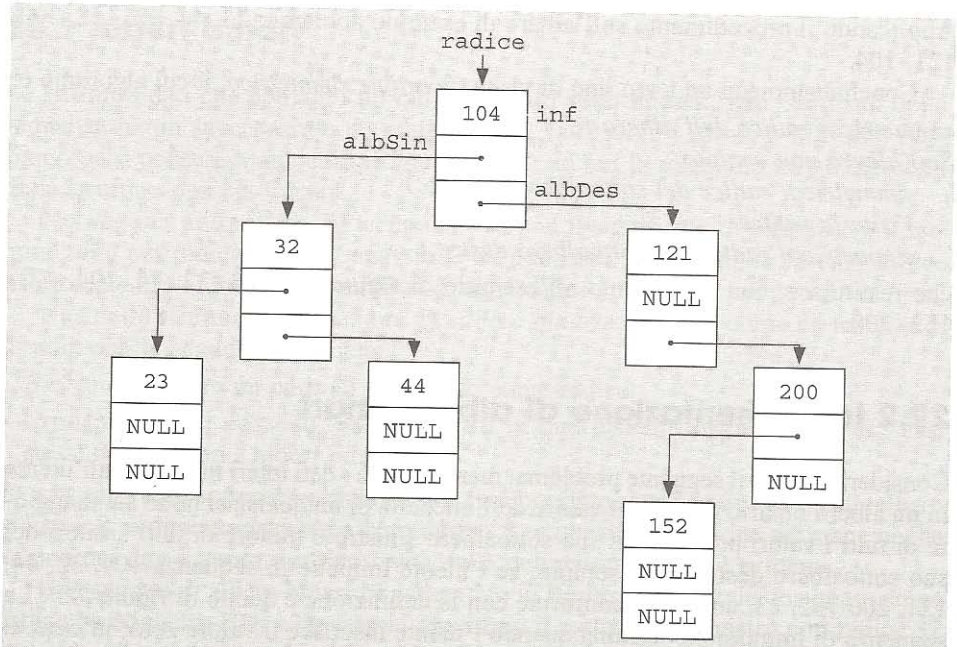

Figura 22.2 *Esempio di memorizzazione dell'albero binario di Figura 22.1 in una lista doppia.*

L'albero binario di ***Figura 22.1*** verrebbe così memorizzato come in ***Figura 22.2***. I puntatori che non fanno riferimento ad alcun nodo devono essere messi a valore `NULL`, in modo da permettere una corretta terminazione delle visite. Il programma completo è presentato nel ***Listato 22.1***.

Nel `main` è dichiarata la variabile `radice` che conterrà il riferimento alla radice dell'albero; essa è un puntatore a oggetti di tipo `nodo`:

```
struct nodo *radice;
```

Il `main` invoca la funzione `albBin` che crea l'albero e ritorna il puntatore alla radice:

```
radice = albBin();
```

Successivamente chiama la funzione di visita, che visualizza le etichette in ordine anticipato:

```
anticipato(radice);
```

Ad `anticipato` viene passato il puntatore alla radice dell'albero.

Listato 22.1 *Creazione e visita in ordine anticipato di un albero binario.*

```
/* Creazione di un albero binario e visita in ordine
   anticipato. L'etichetta dei nodi è un valore intero,
   le occorrenze multiple dello stesso valore non
   vengono memorizzate                                  */
```

```
#include <stdio.h>
#include <malloc.h>

struct nodo {
  int inf;
  struct nodo *albSin;
  struct nodo *albDes;
};
struct nodo *albBin(void);
struct nodo *creaNodo(struct nodo *, int);
void anticipato(struct nodo *);

main()
{
struct nodo *radice; /* puntatore alla radice dell'albero */

radice = albBin(); /* invoca la funzione per la creazione
                      dell'albero binario                  */
printf("\nVISITA IN ORDINE ANTICIPATO\n");
anticipato(radice);
}

/* Crea l'albero binario. Per ogni etichetta immessa
   dall'utente, invoca la funzione creaNodo.
   Ritorna al chiamante la radice dell'albero          */

struct nodo *albBin(void)
{
struct nodo *p = NULL;
struct nodo x;

do {
printf("\nInserire una informazione (0 per finire): ");
  scanf("%d", &x.inf);

  if(x.inf!=0)
    p = creaNodo(p, x.inf);     /* invoca creaNodo() */
}
while(x.inf!=0);

return(p);                      /* ritorna la radice */
}

/* Visita ricorsivamente l'albero alla ricerca del punto
   di inserimento. Quando trova la posizione, crea un
   nodo, vi inserisce l'etichetta e ritorna il puntatore
   a tale nodo.
```

```
   Parametri in ingresso:
        p      e' il puntatore alla radice
        val    e' l'etichetta da inserire nel nodo     */

struct nodo *creaNodo(struct nodo *p, int val)
{
if(p==NULL) { /* il punto di inserimento e' stato reperito */
   /* Creazione del nodo */
   p = (struct nodo *) malloc(sizeof(struct nodo));
   p->inf = val;       /* inserimento di val in elemento */
   p->albSin = NULL;   /* marca di albero sinistro vuoto */
   p->albDes = NULL;   /* marca di albero destro vuoto   */
}
else {             /* ricerca del punto di inserimento */
   if(val > p->inf)
     /* Visita il sottoalbero destro */
     p->albDes = creaNodo(p->albDes, val);
   else
     if(val < p->inf)
       /* Visita il sottoalbero sinistro */
       p->albSin = creaNodo(p->albSin, val);
}
return(p);         /* ritorna il puntatore alla radice */
}

/* Visita l'albero binario in ordine anticipato */

void anticipato(struct nodo *p)
{
if(p!=NULL) {
   printf("%d ", p->inf); /* visita la radice */
   anticipato(p->albSin); /* visita il sottoalbero sinistro */
   anticipato(p->albDes); /* visita il sottoalbero destro   */
}
}
```

Albero vuoto

La funzione `albBin` crea l'albero vuoto inizializzando a `NULL` il puntatore alla radice p. Il resto della procedura è costituito da un ciclo in cui si richiedono all'utente i valori della sequenza finché non viene immesso zero. A ogni nuovo inserimento viene chiamata la funzione `creaNodo` la quale pensa a inserirlo nell'albero:

creaNodo

```
p = creaNodo(p, x.inf);
```

A ogni chiamata `creaNodo` restituisce la radice dell'albero stesso. La dichiarazione di `creaNodo` è la seguente:

```
struct nodo *creaNodo(struct nodo *p, int val);
```

Come abbiamo già visto per le liste, il caso della creazione del primo nodo deve essere trattato a parte. La funzione `albBin` ha provveduto a inizializzare la radice a valore `NULL`; in questo modo un test su `p` in `creaNodo` dirà se l'albero è vuoto, nel qual caso verrà creato il nodo radice e inizializzato il campo `inf` con il valore immesso dall'utente, passato alla funzione nella variabile `val`:

```
/* Creazione del nodo */
p = (struct nodo *) malloc(sizeof(struct nodo));
p->inf = val;
p->albSin = NULL;
p->albDes = NULL;
```

Prima di far ritornare il controllo ad `albBin` viene assegnato `NULL` ai puntatori sinistro e destro del nodo.

Inserimento nel sottoalbero destro o sinistro

Nel caso esista almeno un nodo già costruito (`p` non è uguale a `NULL`), ci si domanda se `val` sia maggiore del campo `inf` della radice, nel qual caso viene richiamata ricorsivamente `creaNodo` passandole il puntatore al sottoalbero destro:

```
p->albDes = creaNodo(p->albDes, val);
```

Se al contrario `val` è minore di `inf`, viene richiamata ricorsivamente la funzione sul sottoalbero sinistro:

```
p->albSin = creaNodo(p->albSin, val);
```

Se nessuno dei due casi risulta vero, significa che `val` è uguale a `inf` e dunque non deve essere inserito nell'albero perché, come specificava il testo del problema, le occorrenze multiple devono essere scartate.

Si noti come, nel caso di un valore non ancora memorizzato nell'albero, il procedimento ricorsivo termini sempre con la creazione di una foglia, corrispondente alle istruzioni di "creazione del nodo" che abbiamo elencato in precedenza. La connessione del nodo creato al padre avviene grazie al valore di ritorno delle chiamate ricorsive, che è assegnato a `p->albDes` o a `p->albSin` secondo il caso.

Visita in ordine anticipato

La funzione `anticipato` corrisponde in maniera speculare alla definizione di visita in ordine anticipato data precedentemente. L'intestazione della definizione della funzione è la seguente:

```
void anticipato(struct nodo *p);
```

Il parametro attuale `p` assume il valore della radice; se non è uguale a `NULL` (cioè se l'albero non è vuoto) ne viene stampata l'etichetta e viene invocata ricorsivamente `anticipato` passandole la radice del sottoalbero sinistro:

```
anticipato(p->albSin);
```

Successivamente viene richiamata la funzione passandole la radice del sottoalbero destro:

```
anticipato(p->albDes);
```

22.3 Visita in ordine simmetrico

Il problema trattato in questo paragrafo è: ampliare il programma del paragrafo precedente in modo che venga effettuata anche la visita in ordine simmetrico e inoltre sia ricercato nell'albero un valore richiesto all'utente. Nel ***Listato 22.2*** sono riportate le modifiche da apportare al programma e le nuove funzioni.

La funzione di visita `simmetrico` è analoga ad `anticipato` esaminata precedentemente. La differenza sta nel fatto che prima viene visitato il sottoalbero sinistro, poi viene visitata la radice e infine viene visitato il sottoalbero destro.

La variabile `trovato` del `main` è di tipo puntatore a una struttura `nodo`:

```
struct nodo *trovato;
```

Essa viene passata per indirizzo alla funzione `ricerca` che vi immette, se reperito, il puntatore al nodo che contiene il valore ricercato. La funzione `ricerca` si comporta come la visita in ordine anticipato: prima verifica se l'elemento ricercato è nella posizione corrente, poi lo ricerca nel sottoalbero sinistro, infine lo ricerca nel sottoalbero destro.

Listato 22.2 *Visita in ordine simmetrico e ricerca di un valore nell'albero binario.*

```
/* DA AGGIUNGERE ALLE DICHIARAZIONI INIZIALI
   DEL LISTATO 22.1 */

void simmetrico(struct nodo *);
void ricerca(struct nodo *, int, struct nodo **);

/* DA AGGIUNGERE AL MAIN DEL LISTATO 22.1 */

struct nodo *trovato;
int chi;
...
printf("\nVISITA IN ORDINE SIMMETRICO\n");
simmetrico(radice);

printf("\nInserire il valore da ricercare: ");
scanf("%d", &chi);

printf("\nRICERCA COMPLETA");
trovato = NULL;
ricerca(radice, chi, &trovato);
if(trovato != NULL)
  printf("\n Elemento %d presente \n", trovato->inf);
else
  printf("\n Elemento non presente\n");
```

```
/* Funzione che visita l'albero binario in ordine simmetrico.
   DA AGGIUNGERE AL LISTATO 22.1                                */

void simmetrico(struct nodo *p)
{
if(p!=NULL) {
   simmetrico(p->albSin);
   printf("%d   ", p->inf);
   simmetrico(p->albDes);
}
}

/* Funzione che ricerca un'etichetta nell'albero binario.
   DA AGGIUNGERE AL LISTATO 22.1.
   Visita l'albero in ordine anticipato                         */

void ricerca(struct nodo *p, int val, struct nodo **pEle)
{
if(p!=NULL)
   if(val == p->inf) /* La ricerca ha dato esito positivo */
      *pEle = p;     /* pEle è passato per indirizzo
                        per cui l'assegnamento di p
                        avviene sul parametro attuale   */
   else {
      ricerca(p->albSin, val, pEle);
      ricerca(p->albDes, val, pEle);
}
}
```

22.4 Alberi binari di ricerca

L'algoritmo di ricerca che abbiamo implementato non è dei più veloci. Infatti, per accorgersi che un valore non è presente si deve visitare l'intero albero, realizzando cioè quella che abbiamo definito nel Capitolo 12 come una ricerca completa. Osserviamo però che, preso un qualsiasi nodo, il suo valore è maggiore di tutti i nodi presenti nel suo sottoalbero sinistro e minore di tutti i nodi del suo sottoalbero destro. Una ricerca che utilizzi questa informazione porta più velocemente al risultato (si veda il ***Listato 22.3***).

Listato 22.3 *Ricerca di un valore nell'albero binario*

```
/* Ricerca ottimizzata */

void ricBin(struct nodo *p, int val, struct nodo **pEle)
```

```
{
if(p!=NULL)
   if(val == p->inf) {
      printf("  trovato ");
      *pEle = p;
   }
else
   if(val < p->inf) {
      printf("  sinistra");
      ricBin(p->albSin, val, pEle);
   }
   else {
      printf(" destra");
      ricBin(p->albDes, val, pEle);
   }
}
```

RIFERIMENTO

Complessità computazionale degli algoritmi di ordinamento

Capitolo 3
Paragrafo 6
Capitolo 12
Paragrafo 3 e 4

Se il valore ricercato non è quello presente nel nodo attuale lo andiamo a ricercare nel sottoalbero sinistro se risulta esserne maggiore, altrimenti nel suo sottoalbero destro. Considerando l'albero di ***Figura 22.1*** la ricerca del valore 23 o 200 si conclude in tre chiamate; se ricerchiamo un valore non presente la ricerca si chiude al massimo in cinque chiamate (il numero di livelli più 1).

In pratica abbiamo realizzato un algoritmo che ha prestazioni analoghe a quello di ricerca binaria. L'idea che sta dietro a questo modo di procedere è utilizzata per creare gli indici dei file.

APPROFONDIMENTI

Prestazioni degli alberi binari di ricerca

Osserviamo che la ricerca ha le stesse prestazioni di quella binaria solamente se l'albero è *bilanciato*, cioè se, preso un qualsiasi nodo, il numero di nodi dei suoi due sottoalberi differisce al più di una unità. Il numero di livelli dell'albero, infatti, costituisce un limite superiore al numero di accessi effettuati dalla ricerca.

La struttura dell'albero creato con la funzione `creaNodo` dipende dalla sequenza con cui vengono immessi i valori in ingresso; il caso peggiore si presenta quando questi sono già ordinati. Per esempio, se l'utente inserisse: 23, 32, 44, 104, 121, 152, 200 l'albero si presenterebbe totalmente sbilanciato, un albero cioè dove il numero di livelli è uguale al numero di elementi inseriti. Altre sequenze danno vita ad alberi più bilanciati.

Si potrebbero scrivere algoritmi che bilanciano l'albero mentre lo creano, ma spesso nella realtà – specialmente se si sta lavorando con la memoria secondaria – questo risulta pesante in termini di tempi di risposta. Un'altra soluzione è quella di prevedere funzioni che trasformino alberi qualsiasi in alberi bilanciati, per cui l'inserimento di elementi avviene così come l'abbiamo visto e, a tempi determinati, viene lanciata la procedura di bilanciamento.