20 Alberi

Indice

Introduzione

In questo capitolo introduciamo gli "alberi", strutture dati molto importanti in informatica e nella scienza della programmazione. A differenza degli array e delle liste, gli alberi sono strutture *non lineari*. Essi sono molto utilizzati, per esempio per rappresentare formule algebriche, aumentare la velocità dell'estrazione di dati da archivi, nell'intelligenza artificiale e nella crittografia. I "file system" dei sistemi operativi sono organizzati ad albero, e alberi vengono pure utilizzati nella progettazione di compilatori, elaboratori di testo e algoritmi di ricerca.

20.1 Gli alberi

Si dice "grafo diretto" un insieme di nodi (che contengono informazioni, come nelle liste) tutti legati "a due a due" da archi direzionati (puntatori). Un *albero* è un grafo diretto in cui ogni nodo può avere un solo arco entrante e un qualunque numero di archi uscenti. Se un nodo non ha archi uscenti si dice "foglia". Se un nodo non ha archi entranti si dice "radice". Poiché in un albero non ci sono nodi con due o più archi entranti, per ogni albero vi deve essere una e una sola radice.

Probabilmente l'esempio più rappresentativo del concetto di albero è l'"albero genealogico" (Figura 20.1 E Figura 20.2).

Per analogia con l'albero genealogico, il nodo da cui un arco parte si dice *padre*, un nodo a cui questo arriva si dice *figlio*. Per esempio, in Figura 20.3 il nodo `B` è il padre dei figli `C` e `D`. Due nodi con lo stesso padre sono detti fratelli. Da ogni nodo non-foglia di un albero si dirama un sottoalbero, quindi si intuisce la natura ricorsiva di questa struttura dati. Dato un nodo, i nodi che appartengono al suo sottoalbero si dicono suoi *discendenti*. Dato un qualunque nodo, i nodi che si trovano nel *cammino* dalla radice a esso sono i suoi *ascendenti*. Per esempio, `B` e `A` sono ascendenti di `C`.

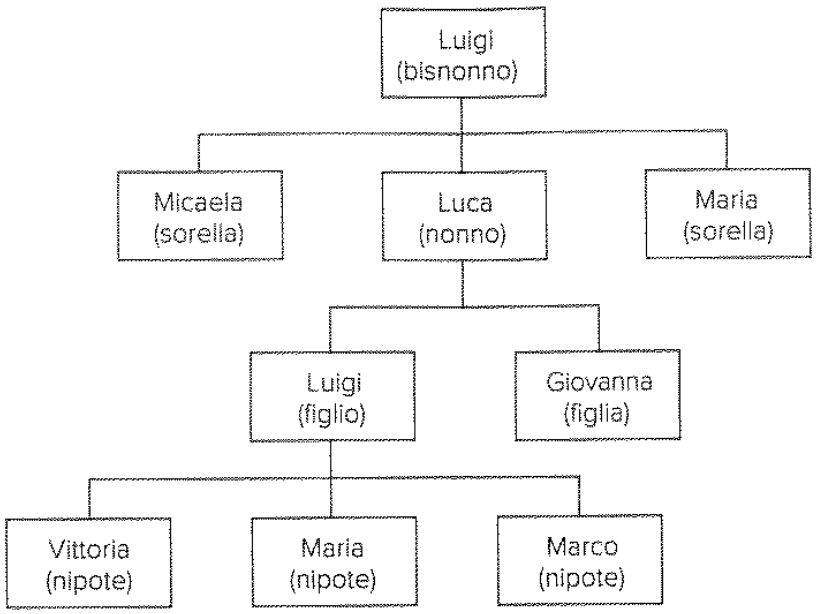


Figura 20.1
Albero genealogico.

Il livello di un nodo è la sua distanza dalla radice; la radice ha livello 0, i suoi figli livello 1, i suoi nipoti livello 2 e così via. I fratelli hanno lo stesso livello, ma non tutti i nodi dello stesso livello sono fratelli.

Nella Figura 20.4, il cammino dalla radice alla foglia I si rappresenta con la sequenza dei nomi dei nodi AFI. La profondità di un albero è la lunghezza del suo cammino più lungo aumentata di uno (per definizione la profondità di un albero vuoto è 0), ovvero è il livello della sua foglia più profonda. La Figura 20.4 contiene nodi su tre livelli: 0, 1 e 2, e ha profondità 3.

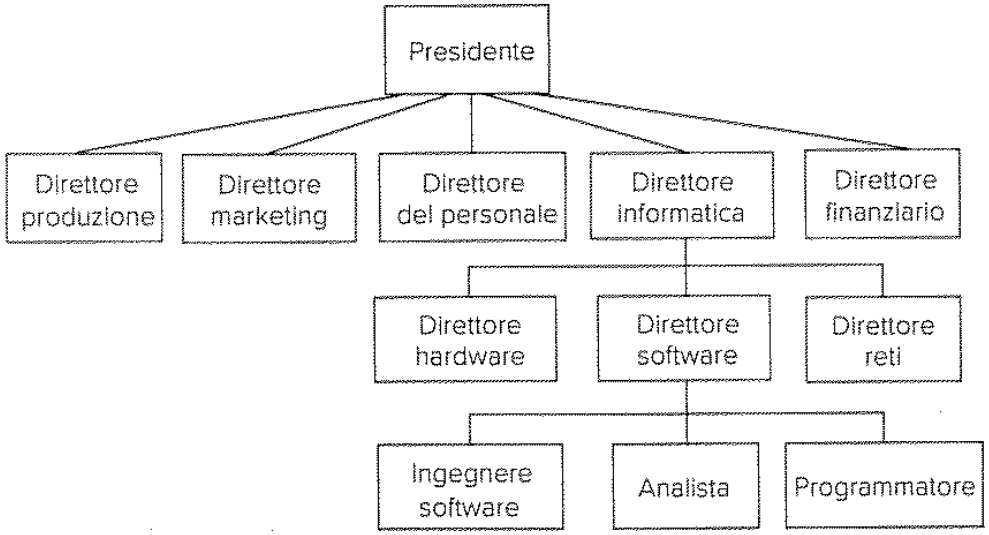


Figura 20.2
Struttura gerarchica ad albero.

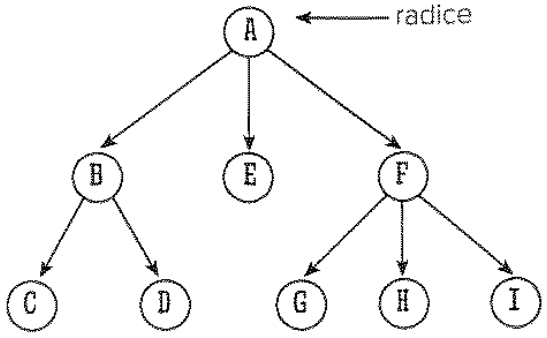


Figura 20.3
Albero.

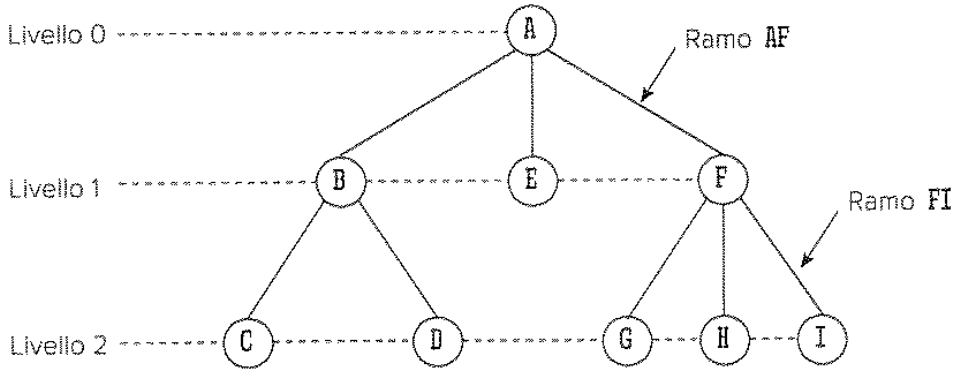
DEFINIZIONE

Il livello di un nodo è la sua distanza dalla radice. La profondità di un albero è la lunghezza del suo cammino più lungo aumentata di uno.

Come già osservato, il concetto di sottoalbero mette in luce la struttura intrinsecamente *ricorsiva* dell'albero. Un albero è un insieme di nodi tale per cui:

1. o è vuoto;
2. oppure ha un nodo denominato *radice* da cui discendono zero o più sottoalberi che sono essi stessi alberi (Figura 20.5).

Un albero di profondità h è *equilibrato* quando, dato un numero massimo di k figli per ciascun nodo, ogni nodo di livello $l < h - 1$ ha esattamente k figli. Un albero di profondità h è *perfettamente equilibrato* quando ciascun nodo di livello $l < h$ ha esattamente k figli (Figura 20.6).



Padri: A, B, F
Figli: B, E, F, C, D, G, H, I
Fratelli: {B, E, F}, {C, D}, {G, H, I}
Foglie: C, D, E, G, H, I
Nodi interni: B, F

Figura 20.4
Terminologia degli alberi.

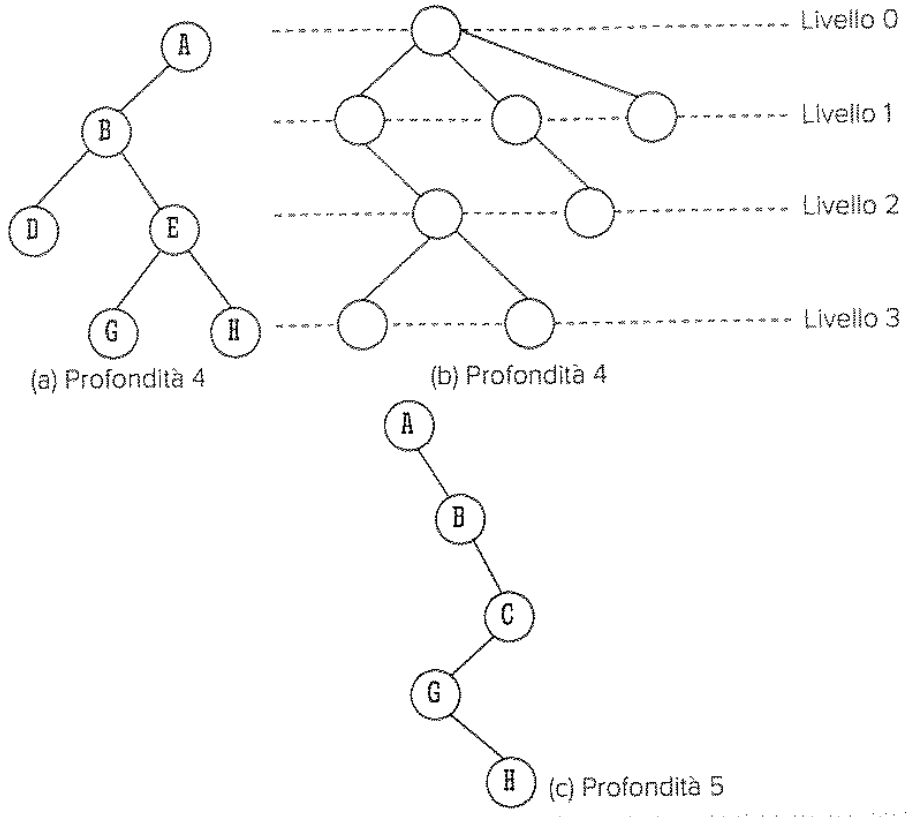


Figura 20.5
Alberi di varie profondità.

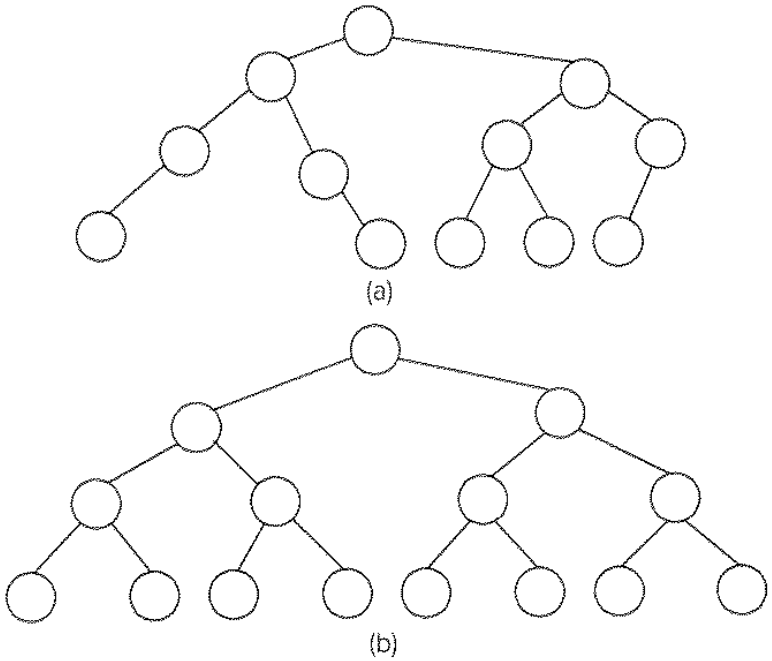


Figura 20.6
(*a*) Un albero equilibrato. (*b*) Un albero perfettamente equilibrato.

20.2 Alberi binari

Un albero è *binario* se ogni nodo non ha più di due figli (sottoalberi), un sinistro e un destro (Figura 20.7.)

A ogni livello *n*, un albero binario può contenere da 1 a 2^n nodi.

Nella la Figura 20.8 (*a*) l'albero A contiene 8 nodi e ha Profondità 3, mentre b contiene 5 nodi e ha Profondità di 4. Questo secondo caso è un albero "degenerato" in una lista semplicemente collegata, perché c'è solo un nodo foglia (E) e ogni nodo non-foglia ha solo un figlio.

20.2.1 Equilibrio

Il livello di un nodo determina l'efficienza con la quale esso può essere localizzato. Una caratteristica importante di un albero binario è l'*equilibrio*. Si

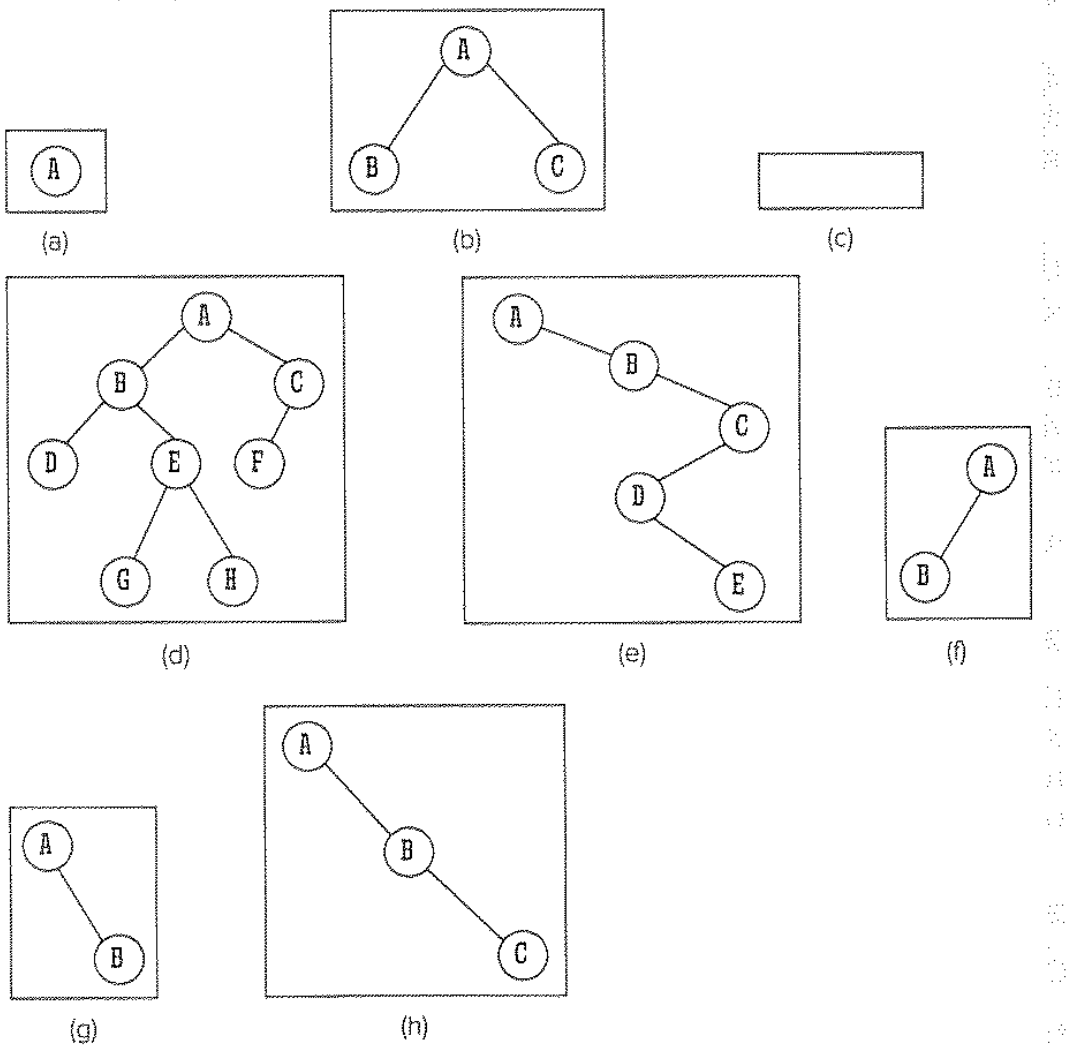


Figura 20.7
Alberi binari.

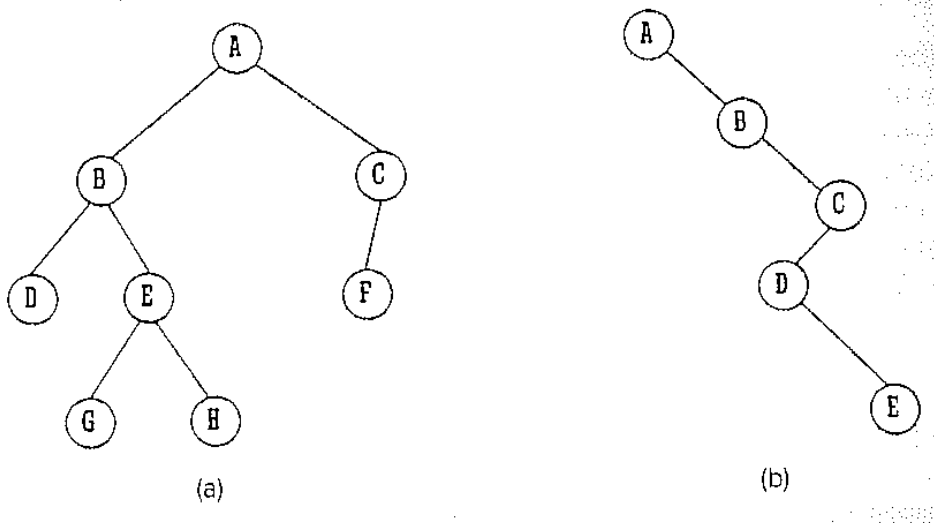


Figura 20.8
Alberi binari: (a) profondità 4; (b) profondità 5.

definisce *fattore di equilibrio* di un albero binario la differenza di profondità fra i suoi due sottoalberi. Denominate rispettivamente H_s e H_D le altezze dei sottoalberi sinistro e destro dell'albero B, il suo fattore di equilibrio è:

$$B = H_D - H_s$$

Utilizzando questa formula agli otto alberi della Figura 20.7 si ha (a) 0, (b) 0, (c) 0, (d) −1, (e) 4, (f) −1, (g) 1, (h) 2.

Un albero si dice *equilibrato* se sia lui sia i suoi sottoalberi hanno equilibrio 0. Poiché ciò capita di rado, si utilizza una definizione alternativa: un albero si dice *equilibrato* se sia lui sia i suoi sottoalberi hanno equilibrio 1 oppure 0 oppure −1.

Un albero binario si dice *degenerato* se ha una sola foglia. Un albero degenerato è una lista semplice.

20.3 Struttura di un albero binario

La struttura di un albero, in particolare di un albero binario, è simile a quella delle liste concatenate, con la differenza che i nodi hanno più di un puntatore all'elemento successivo, perché ogni nodo può avere più successivi. Nel caso dell'albero binario i puntatori si dicono *destro* e *sinistro* (Figura 20.9).

20.3.1 Operazioni sugli alberi binari

Operazioni tipiche sugli alberi binari sono:

- determinarne l'altezza;
- determinarne il numero di elementi;
- copiarli;
- visualizzarli;

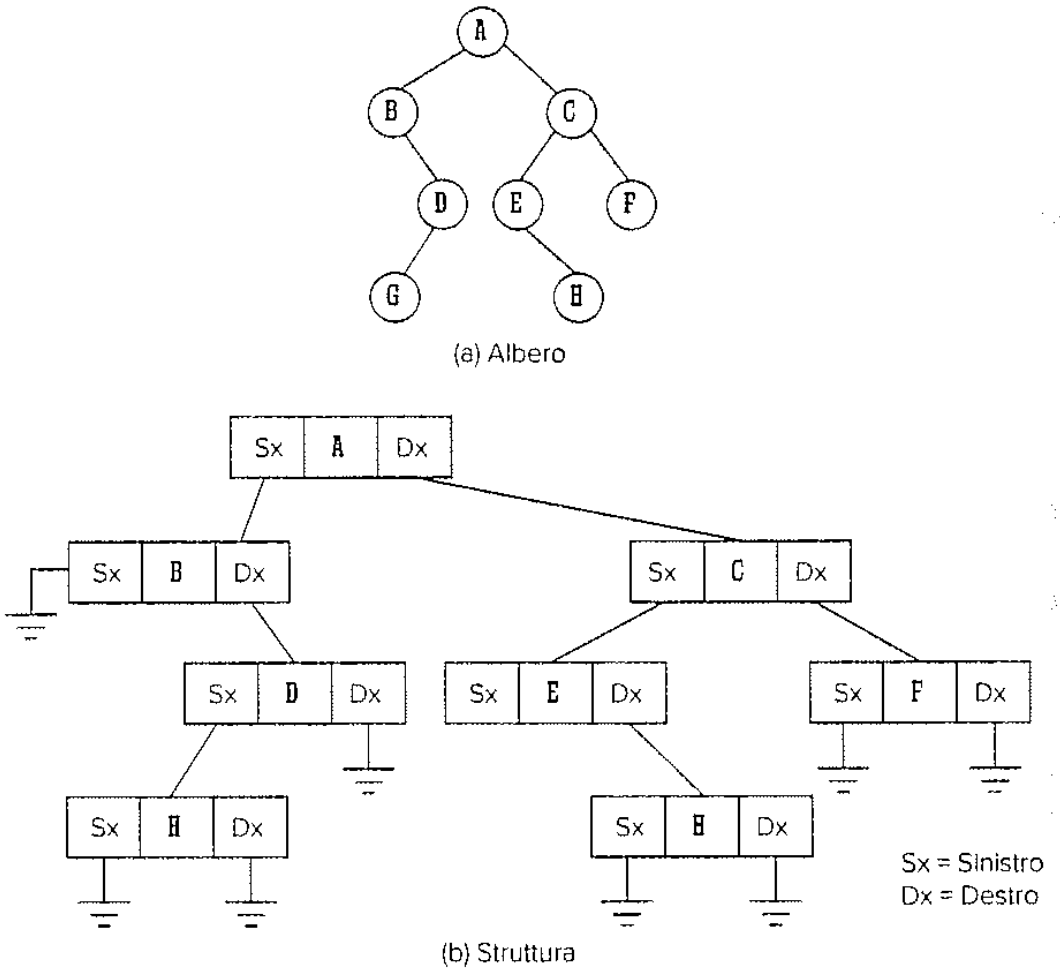


Figura 20.9
(a) Albero binario e (b) relativa struttura in nodi.

- confrontarli;
- eliminarli;
- valutare l'espressione algebrica che essi eventualmente rappresentano.

Come vedremo, queste operazioni si possono compiere visitando l'albero in maniera ricorsiva.

20.3.2 Rappresentazione di un albero binario

In C++ la struttura di un albero binario si può rappresentare così.

1. La classe nodo è annidata dentro la classe albero.

```
class albero
{
  class nodo
  {
    public:
```

```
        char* dati;
      private:
        nodo* destro;
        nodo* sinistro;
      friend class albero;
   };
public:
      nodo* radice;                    // radice dell'albero
      albero()  {radice = NULL; };
                                       // ... altre funzioni membro
};
```

2. questa classe AlberoBin non contiene la classe NodoAlbero, però la dichiara come classe amica.

```
template <typename T> class AlberoBin;

// dichiara un oggetto nodo di un albero binario
template < typename T> class NodoAlbero
{
   private:
      NodoAlbero <T>* sinistro;
      NodoAlbero <T>* destro;
      T dati;
   public:
      NodoAlbero(T item, NodoAlbero <T> *ps, NodoAlbero <T> *pd) {
         dati  = item;
         sinistro = ps;
         destro = pd;
      }
// metodi per accedere agli attributi puntatore
      NodoAlbero <T>* OSinistro() {return sinistro;}
      NodoAlbero <T>* ODestro(void) {return destro;};
      void PSinistro (NodoAlbero <T>* Sx) { sinistro = Sx;}
      void PDestro (NodoAlbero <T>* Dx) { destro = Dx;}
// metodi per accedere all'attributo dato
      T Odati() { return dati;}
      void Pdati( T e){ dati = e;}
      friend class AlberoBin <T>;
};

template <class T> class AlberoBin
{
   private:
      NodoAlbero <T> * p;
   public:
      AlberoBin() { p = NULL;}
// altre funzioni membro
};
```

20.4 Visita di un albero

Ci sono vari modi per visitare i nodi di un albero; ciò che si richiede in generale è che ogni nodo sia esaminato una sola volta e l'attraversamento dell'albero avvenga secondo un criterio predeterminato. Partendo dalla radice, in linea generale un albero può essere visitato o "in profondità" oppure "in ampiezza".

Nella visita in profondità, dopo il nodo corrente viene esaminato un suo figlio, quale dipende dal particolare algoritmo; se non c'è alcun figlio si ritorna dal padre ("backtracking" semplice) verso un eventuale altro figlio e così via. Così facendo tutti i discendenti di ogni nodo vengono visitati prima dei suoi eventuali fratelli o pari livello.

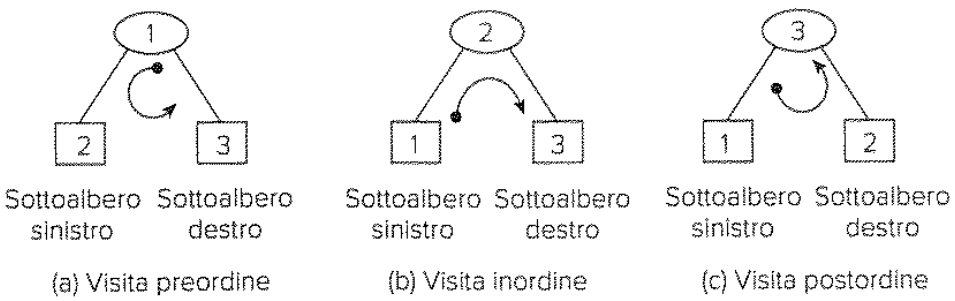
Nella visita in ampiezza, dopo il nodo corrente viene visitato un altro nodo di pari livello. Così facendo tutti i nodi di un livello verranno visitati prima di tutti i nodi del livello successivo.

Focalizzando l'attenzione sugli alberi binari, esistono tre strategie di visita notevoli (Figura 20.10), l'inordine, la preordine e la postordine.

20.4.1 Visita *in preordine*

La visita in preordine visita prima la radice, quindi il sottoalbero sinistro e da ultimo quello destro. L'algoritmo è quindi ricorsivo: sia per processare il sottoalbero sinistro sia quello destro si richiama lo stesso procedimento di visita in preordine. Per esempio, nel caso dell'albero della Figura 20.11 si visita prima la radice A, quindi il sottoalbero sinistro in preordine, cioè prima B, poi D e quindi E. Successivamente si visita in preordine il sottoalbero destro, cioè la sua radice C, poi F e quindi G. Di conseguenza la visita in preordine dell'albero della Figura 20.11 restituisce nell'ordine A-B-D-E-C-F-G. In C++ l'algoritmo potrebbe essere espresso così:

```
void PreOrdine(Nodo* p)
{
   if (p)
   {
      cout << p -> dati << ";     // visita la radice
      PreOrdine(p -> sinistro);   // visita il sottoalbero sinistro
```



(a) Visita preordine (b) Visita inordine (c) Visita postordine

Figura 20.10
Visita di alberi binari.

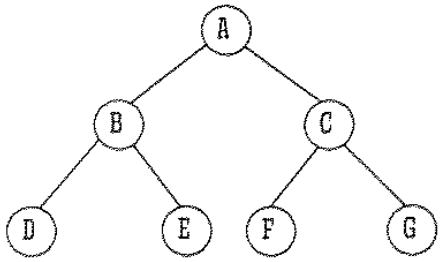


Figura 20.11
Visita preordine di un albero binario.

```
    PreOrdine(p -> destro);      // visita il sottoalbero destro
  }
}
```

dove l'istruzione di stampa del dato nel nodo radice può essere rimpiazzata da qualunque tipo di manipolazione dello stesso.

20.4.2 Visita *inordine*

La visita inordine processa prima il sottoalbero sinistro, quindi la radice e infine il sottoalbero destro.

Nell'albero della Figura 20.11 bisogna quindi visitare inordine il sottoalbero con radice B, cosa che richiede di visitare prima il sottoalbero con radice D. Quest'ultimo non ha sottoalberi quindi il primo nodo visitato è proprio la sua radice D. Dopodiché si visita la radice del sottoalbero con radice B, cioè il nodo B stesso, e poi il suo sottoalbero destro, cioè il nodo E. Quindi si risale alla radice A e poi al suo sottoalbero destro. La visita di quest'ultimo richiede prima la visita del suo sottoalbero sinistro, cioè F, la sua radice, cioè C, e da ultimo il suo sottoalbero destro G. Quindi la visita inordine dell'albero produce D-B-E-A-F-C-G.

La seguente funzione C++ schematizza l'algoritmo di visita inordine di un albero binario.

```
void InOrdine(Nodo *p)
{
  if (p)
  {
    InOrdine(p -> sinistro);   // visita il sottoalbero sinistro
    cout << p -> dati << ' ';  // visita la radice
    InOrdine(p -> destro);     // visita il sottoalbero destro
  }
}
```

20.4.3 Visita *postordine*

La visita in postordine processa prima il sottoalbero sinistro, poi quello destro e infine la radice.

Se si utilizza la visita in postordine dell'albero della Figura 20.11, si visita prima il sottoalbero con radice B, cosa che, ricorsivamente, richiede la visita del suo sottoalbero sinistro D, quindi il sottoalbero destro E, poi la sua radice B. Quindi si passa al sottoalbero con radice C, che richiede la visita del sottoalbero sinistro F, poi quello destro G, e la radice C. Da ultimo si visita la radice dell'albero A. Quindi la visita in postordine dell'albero produce la sequenza: D-E-B-F-G-C-A.

La seguente funzione C++ schematizza l'algoritmo di visita in postordine di un albero binario.

```
void PostOrdine(Nodo *p)
{
   if (p)
   {
      PostOrdine(p -> sinistro);    // visita il sottoalbero sinistro
      PostOrdine(p -> destro);      // visita il sottoalbero destro
      cout << p -> dati << '';      // visita la radice
   }
}
```

Esempio 20.1

Rappresentare sia il nodo sia le funzioni ricorsive mediante template.

Classe nodo

```
template <typename T> class NodoAlberoBin {
public :
   NodoAlberoBin() {FiglioSinistro = FiglioDestro = 0;}
   NodoAlberoBin(const T& e)
   {
      info = e;
      FiglioSinistro = FiglioDestro = 0;
   }
   NodoAlberoBin(const T& e, NodoAlberoBin <T>* s, NodoAlberoBin <T>* d)
   {
      info = e;
      FiglioSinistro = s;
      FiglioDestro = d;
   }
private :
   T info;
   NodoAlberoBin <T>   *FiglioSinistro,      // sottoalbero sinistro
                       *FiglioDestro;        // sottoalbero destro
};
```

***Visita PreOrdine di** *t*

```
template <typename T> void PreOrdine(NodoAlberoBin<T> *t)
```

```
{
   if (t) {
      Visita(t);
      PreOrdine(t -> FiglioSinistro);
      PreOrdine(t -> FiglioDestro);
   }
}
```

Visita InOrdine di *t

```
template <typename T> void InOrdine(NodoAlberoBin<T> *t)
{
   if (t) {
      InOrdine(t -> FiglioSinistro);
      Visita(t);
      InOrdine(t -> FiglioDestro);
   }
}
```

Visita PostOrdine di *t

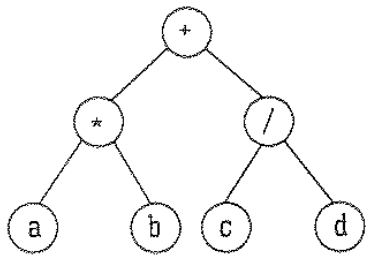
```
template <typename T> void postordine(NodoAlberoBin<T> *t)
{
   if (t) {
      postordine(t ->FiglioSinistro);
      postordine(t ->FiglioDestro);
      Visita(t);
   }
}
```

dove la funzione Visita() può essere rimpiazzata con:

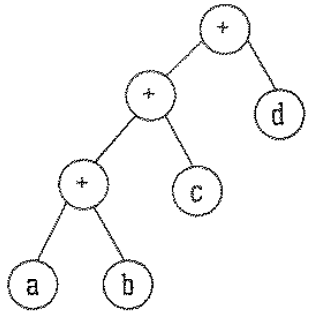
```
cout << t->info;
```

Esempio 20.2

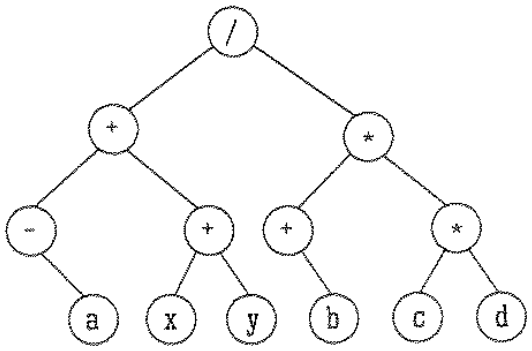
Indicare le sequenze di visita dei seguenti alberi binari.



(A)



(B)



(C)

Sequenze prodotte dalle visite preordine, inordine e postordine.

	Albero A	*Albero B*	*Albero C*
PreOrdine	+*ab/cd	+++abcd	/*-a+xy*+ b* cd
InOrdine	a*b+c/d	a+b+c+d	-a+x+y/+ b* c*d
PostOrdine	ab*cd/+	ab+c+d+	a-xy++b+cd**/

20.4.4 Profondità di un albero binario

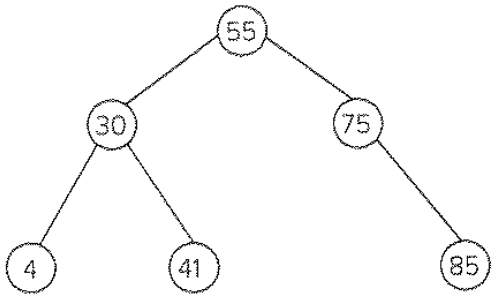
La funzione Profondita valuta la *profondità* dell'albero binario passatole mediante un puntatore al nodo radice. Il caso più semplice è quello di un albero vuoto, la cui profondità è 0. Se l'albero non è vuoto bisogna valutare separatamente le profondità ProfonditaSx e ProfonditaDx di ogni sottoalbero. L'algoritmo consiste nelle chiamate ricorsive della funzione Profondita con i due puntatori ai due sottoalberi come parametro. La funzione Profondita restituisce la profondità del sottoalbero più profondo aumentata di 1 (quella della radice).

```
int Profondita(Nodo *p)
{
   if (!p) return 0;
   else
   {
      int ProfonditaSx = Profondita(p->sx);
      int ProfonditaDx = Profondita(p-> dx);
      if (ProfonditaSx > ProfonditaDx)
         return ProfonditaSx + 1;
      else
         return ProfonditaDx + 1;
   }
}
```

20.5 Albero binario di ricerca

Gli alberi visti finora non hanno alcun ordine definito riguardo i valori presenti dentro i nodi.

In un albero binario *di ricerca*, dato un qualunque nodo, tutti i nodi del suo sottoalbero sinistro hanno valori minori (o maggiori) del suo, mentre tutti i nodi di quello destro li hanno maggiori (o minori) del suo. Il seguente è un esempio di albero binario di ricerca.



4 minore di 30
30 minore di 55
41 maggiore di 30
75 maggiore di 55
85 maggiore di 75

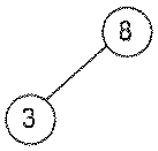
20.5.1 Creazione di un albero binario di ricerca

Supponiamo di dover memorizzare i seguenti numeri in un albero binario di ricerca.

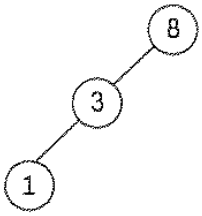
8 3 1 20 10 5 4

Inizialmente bisogna inserire il valore 8 in un albero vuoto, quindi l'unica scelta è metterlo nella radice.

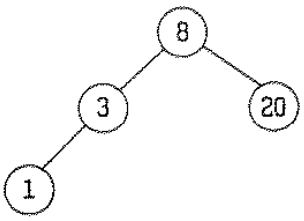
Poiché il successivo 3 è minore di 8, esso deve andare nel sottoalbero sinistro.



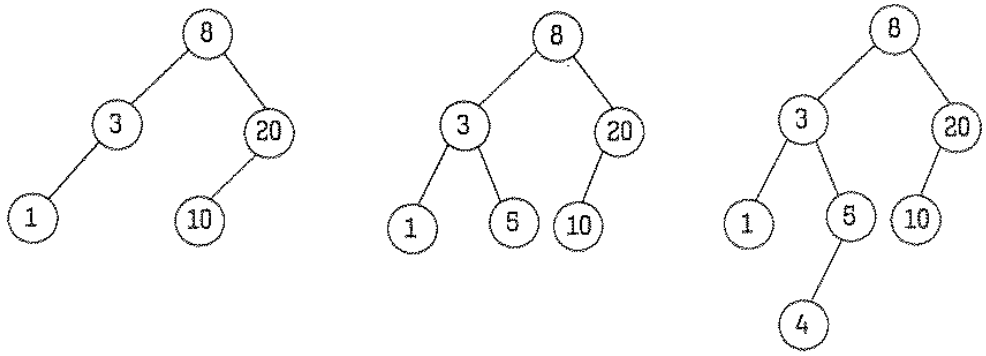
Quindi bisogna inserire 1 che è minore sia di 8 y che di 3, quindi deve andare nel sottoalbero sinistro di 3.



Il successivo 20, maggiore di 8, deve andare alla sua destra.



e così via. Ogni nuovo elemento viene inserito come foglia dell'albero corrente.



20.5.2 Implementazione di un nodo di un albero binario di ricerca

Gli alberi binari di ricerca servono a ritrovare velocemente i dati in esso memorizzati. Supponiamo di avere un albero binario in cui ciascun nodo contiene il nome e il numero di matricola di uno studente.

```
class Nodo {
public :
   int numero_matricola;
   char cognome_nome[30];
   Nodo *sinistro, *destro;
   Nodo (int id = 0, char *n = 0, Nodo *i = 0, Nodo *d = 0)
      : numero_matricola(id), sinistro(i), destro(d)
      { strcpy(cognome_nome,n) ; }
};
```

20.5.3 Ricerca

La ricerca di un nodo inizia dalla radice e segue questi passi:

1. *si confronta la chiave cercata con quella della radice;*
2. *se coincidono s'e' trovato l'elemento cercato;*
3. *se la chiave cercata è maggiore di quella della radice la ricerca viene avviata ricorsivamente nel sottoalbero destro, altrimenti in quello sinistro.*

Nel nostro caso la funzione di ricerca ha due argomenti, il puntatore all'albero da esaminare e il numero di matricola dello studente da cercare. Se lo trova, la funzione restituisce il puntatore al nodo dello studente, altrimenti restituisce il valore 0. Il codice C++ è:

```
Nodo* ricerca(Nodo* p, int cercato)
{
   if (!p) return 0;
      else if (cercato == p->numero_matricola) return p;
         else if (cercato < p->numero_matricola)
            return ricerca(p->sinistro, cercato);
            else return ricerca(p->destro, cercato);
}
```

20.5.4 Inserire un nodo

L'inserimento di un nuovo elemento in un albero di ricerca deve essere fatto in maniera tale che il risultato dell'operazione sia ancora un albero di ricerca. Si visita l'albero alla ricerca dell'elemento da inserire, e se lo si trova non si fa nulla (perché l'elemento c'è già); altrimenti si inserisce l'elemento nel posto dove la ricerca è terminata (senza trovare nulla), perché quello è il posto dove l'elemento sarebbe dovuto stare se fosse stato presente (Figura 20.12).

Per esempio, nella Figura 20.12 per inserire l'elemento 8 si parte dal nodo radice 25; l'8 dovrebbe stare nel sottoalbero sinistro perché 8<25. Rispetto poi al nodo 10, l'8 dovrebbe stare nel sottoalbero sinistro che al momento è vuoto. Quindi il nodo 8 sarà il figlio sinistro del nodo 10.

L'inserimento di un nodo inizia quindi dalla radice e segue questi passi:

1. *assegnare dinamicamente memoria per il nuovo nodo;*
2. *cercare l'elemento nell'albero per inserirlo come foglia;*
3. *appendere il nuovo elemento all'albero.*

La funzione inserire ha tre argomenti: il puntatore alla radice dell'albero, il puntatore al nuovo elemento e il numero di matricola dello studente. L'algoritmo crea un nodo per il nuovo studente e lo va a inserire nel posto corretto come indicato:

```
void inserire(Nodo* p, int nuova_mat, char* nuovo_studente)
{
  if (!p) p = new Nodo(nuova_mat, nuovo_studente);
    else if (nuova_mat < p->numero_matricola);
      inserire(p->sinistro, nuova_mat, nuovo_studente);
        else inserire(p->destro, nuova_mat, nuovo_studente);
}
```

Si noti che se l'albero corrente è vuoto (!p) il nuovo nodo ne diventa la radice.

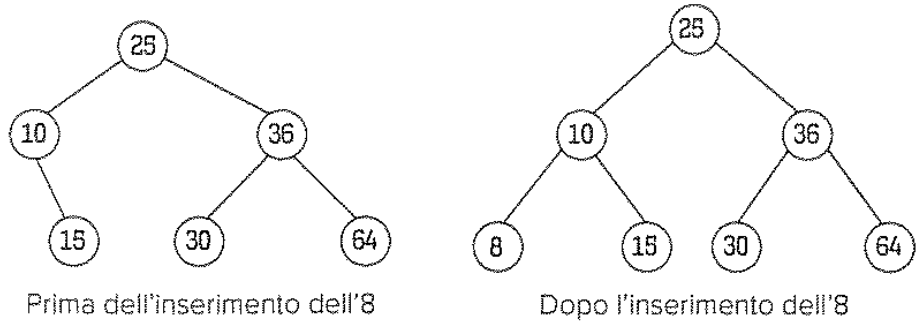


Figura 20.12
Inserimento in un albero binario di ricerca.

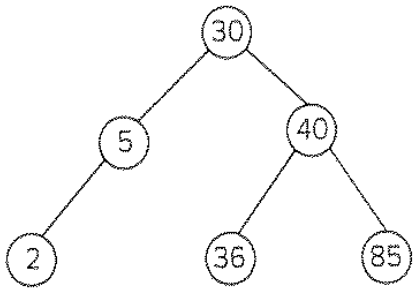
20.5.5 Eliminazione di un nodo

Anche l'eliminazione di un nodo da un albero binario è un'estensione dell'algoritmo di ricerca. Ovviamente l'eliminazione deve produrre comunque un albero di ricerca. I passi da seguire sono:

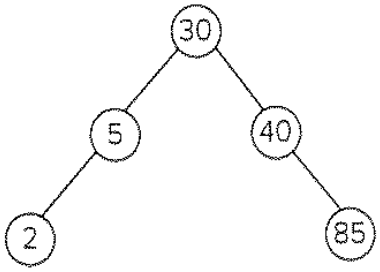
1. *cercare l'elemento da eliminare;*
2. *aggiustare i puntatori dei suoi ancestori se il nodo da eliminare ha meno di due figli, oppure sostituirlo con il nodo con chiave più vicina in modo da conservare la natura di albero di ricerca binario.*

Esempio 20.3

Sopprimere l'elemento con chiave 36 nel seguente albero di ricerca:

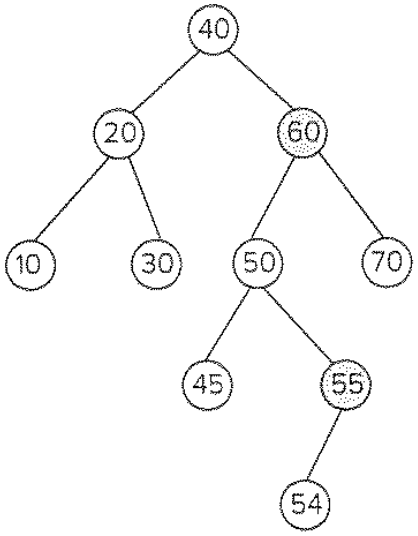


L'albero risultante è

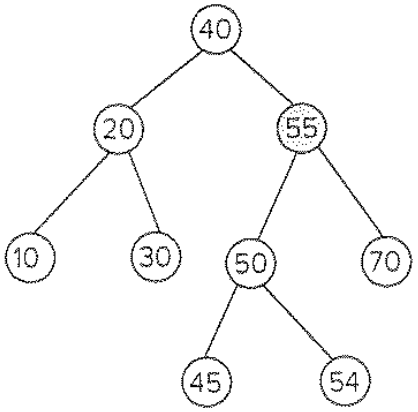


Esempio 20.4

Eliminare l'elemento con chiave 60 nel seguente albero:



Si rimpiazza 60 con l'elemento maggiore del suo sottoalbero sinistro (55) o con l'elemento più piccolo del suo sottoalbero destro (70). Se si opta per il primo caso, si sposta il 55 alla radice del sottoalbero e si riaggiusta l'albero.



SOMMARIO

In questo capitolo abbiamo visto cosa sono e come si gestiscono gli alberi. Strutture dati dinamiche, sono particolari grafi molto utilizzati per varie applicazioni. In particolare abbiamo introdotto l'*albero binario* come quello in cui ogni elemento possiede al massimo due figli. Essi sono utilizzati per rappresentare espressioni logico-aritmetiche: i nodi interni contengono gli operatori, e le foglie gli operandi. Gli alberi binari di ricerca sono invece caratterizzati da un ordinamento: dato un qualunque nodo, tutti i nodi del suo sottoalbero sinistro hanno valori minori del suo, mentre tutti i nodi del suo sottoalbero destro hanno valori maggiori. Questi alberi di ricerca sono fondamentali per memorizzare grandi quantità di dati che devono poi essere ritrovati in maniera veloce.

CONCETTI CHIAVE

- Albero
- Albero binario
- Albero di ricerca binario
- Visita inordine
- Nodo
- Visita postordine
- Visita preordine
- Sottoalbero