

Introduzione alla programmazione in C

Appunti delle lezioni di
Tecniche di programmazione

Giorgio Grisetti

Luca Iocchi

Daniele Nardi

Fabio Patrizi

Alberto Pretto

Dipartimento di Ingegneria Informatica, Automatica e Gestionale

Facoltà di Ingegneria dell'Informazione, Informatica, Statistica

Università di Roma "La Sapienza"

Edizione 2018/2019



2019

Indice

12. Alberi binari	309
12.1. Alberi	309
12.2. Alberi binari	310
12.2.1. Definizione induttiva di albero binario	311
12.2.2. Alberi binari completi	311
12.3. Tipo astratto AlberoBin	312
12.4. Rappresentazione indicizzata di alberi binari	312
12.4.1. Rappresentazione indicizzata di alberi binari completi	312
12.4.2. Rappresentazione indicizzata di alberi binari non completi	314
12.4.3. Rappresentazione indicizzata generale di alberi binari	314
12.5. Rappresentazione collegata di alberi binari	315
12.5.1. Implementazione del tipo astratto AlberoBin	318
12.6. Rappresentazione parentetica di alberi binari	320
12.7. Costruzione di un albero binario da rappresentazione parentetica	321
12.8. Rappresentazione di espressioni aritmetiche mediante alberi binari	322
12.9. Visita in profondità di alberi binari	324
12.9.1. Proprietà della visita in profondità	324
12.9.2. Tipi diversi di visite in profondità: in preordine, simmetrica, in postordine	325
12.10. Realizzazione della visita in profondità	326
12.10.1. Rappresentazione tramite array	326
12.10.2. Rappresentazione tramite puntatori	326

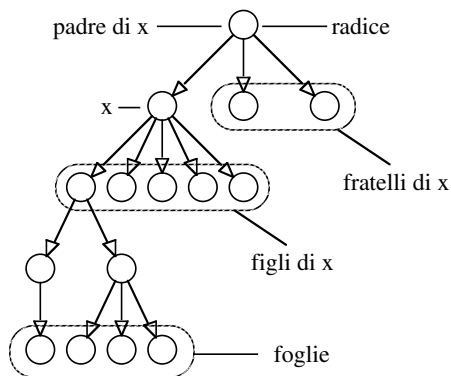
12.11	Applicazioni delle visite	327
12.11.1.	Calcolo della profondità di un albero	327
12.11.2.	Verifica della presenza di un elemento nell'albero	328
12.12	Accesso all'albero per livelli	329
12.12.1.	Stampa dei nodi di un livello	329
12.12.2.	Insieme dei nodi di un livello	329
12.13	Visita in ampiezza di alberi binari	331
12.13.1.	Esempio di visita in ampiezza di alberi binari	332
12.13.2.	Proprietà della visita in ampiezza	332
12.13.3.	Realizzazione della visita in ampiezza	333
12.14	Applicazioni della visita in ampiezza	334
12.14.1.	Ricerca di un elemento a profondità minima	334
12.15	Visita in profondità iterativa	335
12.16	Operazioni di modifica di alberi	336
12.16.1.	Inserimento di un nodo come foglia in posizione data	337
12.16.2.	Inserimento di un nodo interno in posizione data	338
12.16.3.	Cancellazione di un nodo	340
12.17	Alberi binari di ricerca	342
12.17.1.	Verifica della presenza di un elemento in un albero binario di ricerca	343

12. Alberi binari

12.1. Alberi

In questo capitolo viene presentata la struttura dati *albero*, che rappresenta un esempio particolarmente significativo di struttura non lineare. Infatti, prendendo a modello la definizione induttiva della struttura dati, la definizione delle funzioni che operano su alberi binari risulta immediata, mentre le implementazioni con i costrutti di ciclo richiedono un uso di strutture dati di appoggio.

L'albero è un tipo di dato non lineare utilizzato per memorizzare informazioni in modo gerarchico. Un esempio di albero è il seguente:



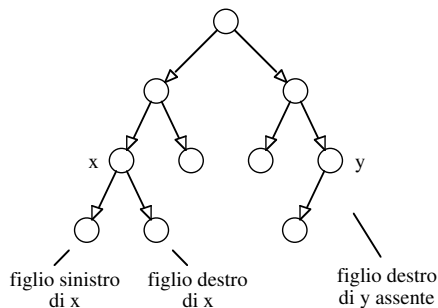
Osserviamo che un albero è costituito da **nodi** (indicati graficamente come cerchi), a cui è tipicamente associato un contenuto informativo, e **archi** (indicati graficamente come frecce), che rappresentano relazioni di discendenza diretta tra nodi. Useremo la seguente terminologia standard:

- **padre** di un nodo x : nodo che ha un collegamento in uscita verso il nodo x . In un albero ogni nodo ha al più un padre;
- **nodo radice** dell'albero: nodo che ha solo collegamenti in uscita verso altri nodi, cioè non ha nessun padre;
- **fratelli** di un nodo x : nodi che hanno lo stesso padre del nodo x ;
- **figli** di un nodo x : nodi verso cui x ha collegamenti in uscita;
- **foglie** dell'albero: nodi che non hanno nessun collegamento in uscita verso altri nodi.

Diremo inoltre che la radice è a **livello** zero nell'albero e che ogni altro nodo si trova ad un livello uguale al livello del padre più uno. Ad esempio, il nodo x è a livello 1 nell'albero della figura precedente. La **profondità** di un albero è pari al massimo livello su cui si trova almeno un nodo dell'albero. Ad esempio, l'albero della figura precedente ha profondità 4.

12.2. Alberi binari

In questo capitolo tratteremo il caso particolare di alberi binari, dove ogni nodo ha al più due figli: un **figlio sinistro** ed un **figlio destro**, come mostrato in figura:



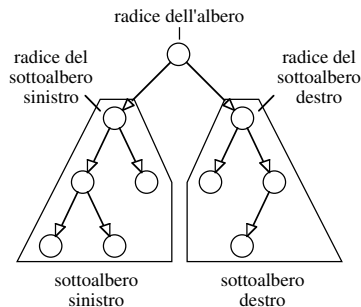
Si noti che il nodo x ha un figlio sinistro ed un figlio destro, mentre il nodo y ha solo un figlio sinistro.

12.2.1. Definizione induttiva di albero binario

Una classica definizione induttiva di albero binario è la seguente:

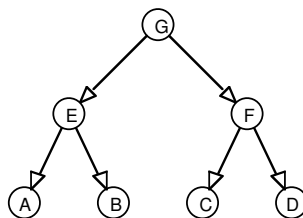
- l'albero vuoto, è un albero binario;
- se T_s e T_d sono due alberi binari, allora l'albero che ha un nodo radice e T_s e T_d come sottoalberi è un albero binario;
- nient'altro è un albero binario.

I due sottoalberi T_s e T_d di un albero binario non vuoto T vengono detti rispettivamente **sottoalbero sinistro** e **sottoalbero destro** di T .



12.2.2. Alberi binari completi

Un albero binario viene detto **completo** se ogni nodo che non sia una foglia ha esattamente due figli e le foglie sono tutte allo stesso livello nell'albero:



È facile verificare che un albero binario completo non vuoto di profondità k ha esattamente $2^{k+1} - 1$ nodi: infatti, vi sono esattamente 2^k foglie a livello k e $2^k - 1$ nodi che non sono foglie, e la somma di queste due quantità dà proprio $2^{k+1} - 1$.

12.3. Tipo astratto **AlberoBin**

Il tipo di dato astratto **AlberoBin** può essere definito come segue.

TipoAstratto **AlberoBin(T)**

Domini

AlberoBin : dominio di interesse del tipo

T : dominio dei valori associati ai nodi degli alberi binari

Funzioni

albBinVuoto() \mapsto **AlberoBin**

pre: nessuna

post: **RESULT** è l'albero binario vuoto (ossia senza nodi)

creaAlbBin(T r, AlberoBin s, AlberoBin d) \mapsto **AlberoBin**

pre: nessuna

post: **RESULT** è l'albero binario che ha **r** come radice, **s** come sottoalbero sinistro, e **d** come sottoalbero destro

estVuoto(AlberoBin a) \mapsto **Boolean**

pre: nessuna

post: **RESULT** è true se **a** è vuoto, false altrimenti

radice(AlberoBin a) \mapsto **T**

pre: **a** non è vuoto

post: **RESULT** è il valore associato al nodo che è radice di **a**

sinistro(AlberoBin a) \mapsto **AlberoBin**

pre: **a** non è vuoto

post: **RESULT** è il sottoalbero sinistro di **a**

destro(AlberoBin a) \mapsto **AlberoBin**

pre: **a** non è vuoto

post: **RESULT** è il sottoalbero destro di **a**

FineTipoAstratto

12.4. Rappresentazione indicizzata di alberi binari

Vi sono diversi modi per rappresentare alberi binari nei linguaggi di programmazione. Uno dei più semplici è la rappresentazione indicizzata, realizzabile mediante array.

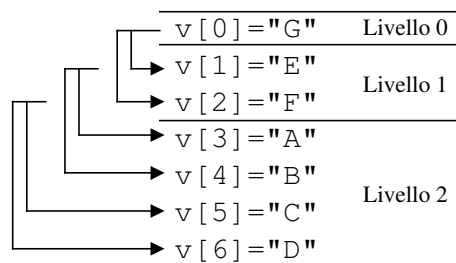
12.4.1. Rappresentazione indicizzata di alberi binari completi

La rappresentazione indicizzata mediante array è particolarmente adatta per rappresentare alberi binari completi. Dato un array v di dimensione n :

- se $n = 0$, l'array v rappresenta l'albero vuoto;

- se $n > 0$, l'array v rappresenta un albero binario completo i cui nodi corrispondono agli indici nell'intervallo $[0..n-1]$ di v . In particolare si ha che:
 - l'albero rappresentato ha n nodi;
 - il contenuto informativo del nodo i è $v[i]$;
 - la radice corrisponde all'indice 0;
 - il figlio sinistro di i ha indice $2i + 1$;
 - il figlio destro di i ha indice $2i + 2$.

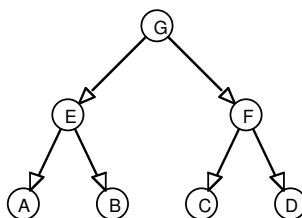
Ad esempio, l'albero binario completo mostrato nella figura precedente può essere rappresentato mediante l'array v di dimensione $n = 7$ come mostrato sotto:



Si noti che i figli del nodo 'F', che corrisponde all'indice 2, sono 'C' e 'D', e questi corrispondono agli indici $5 = 2 \cdot 2 + 1$ e $6 = 2 \cdot 2 + 2$.

Un modo pratico per costruire una rappresentazione indicizzata di un albero binario consiste nel disporre nell'array, a partire da $v[0]$, i valori dei nodi presi da sinistra verso destra su ogni livello a partire dal livello zero fino al livello massimo: questo appare evidente se si osservano le due figure precedenti.

Esempio: L'albero binario completo mostrato nella figura seguente:



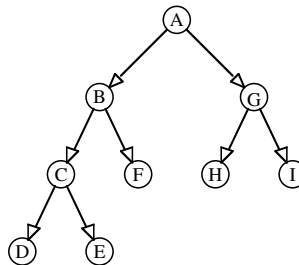
può essere rappresentato in modo indicizzato come array di caratteri:

```
char [] v = { 'G', 'E', 'F', 'A', 'B', 'C', 'D' };
```

12.4.2. Rappresentazione indicizzata di alberi binari non completi

È possibile utilizzare la rappresentazione indicizzata anche per alberi binari non completi, assumendo sia indicato esplicitamente se un nodo i è presente nell'albero. Tuttavia, in questo modo è necessario fare spazio nell'array anche per nodi non effettivamente presenti nell'albero, e questo può essere estremamente dispendioso in termini di memoria richiesta.

Esempio: L'albero binario non completo mostrato nella figura seguente:



può essere rappresentato in C in modo indicizzato come segue:

```
char [] v = { 'A', 'B', 'G', 'C', 'F', 'H', 'I', 'D',
              'E', '0', '0', '0', '0', '0', 'L', '0' };
```

Si dove '0' viene usato per indicare che il nodo non esiste.

12.4.3. Rappresentazione indicizzata generale di alberi binari

La rappresentazione generale di un albero binario sotto forma di array richiede la dichiarazione di un tipo array di nodi dell'albero:

```

#define MaxNodiAlbero 100

typedef ... TipoInfoAlbero;

struct StructAlbero {
    TipoInfoAlbero info;
    bool esiste;
};

typedef struct StructAlbero TipoNodoAlbero;

typedef TipoNodoAlbero TipoAlbero[MaxNodiAlbero];

```

La scelta in questo caso è stata quella di definire staticamente la dimensione dell'array e di usare un campo booleano `esiste` per indicare se un elemento dell'array rappresenta un nodo dell'albero o meno.

Esempio: Il seguente frammento di codice indica come stampare i figli di un nodo `i` assumendo di avere un albero binario rappresentato in modo indicizzato:

```

TipoAlbero albero;
...
// i: nodo di cui stampare i figli
if (i < 0 || i >= MaxNodiAlbero)
    printf("Indice errato...");
else {
    if (2*i+1 < MaxNodiAlbero && albero[2*i+1].esiste)
        printf("sinistro di %d -> %d\n", i, albero[2*i+1].info);
    else
        printf("%d non ha figlio sinistro\n", i);
    if (2*i+2 < MaxNodiAlbero && albero[2*i+2].esiste)
        printf("destro di %d -> %d\n", i, albero[2*i+2].info);
    else
        printf("%d non ha figlio destro\n", i);
}

```

Si noti che un sottoalbero per essere definito deve avere un indice all'interno del massimo consentito ed, in tal caso, il campo `esiste` deve essere `true`.

12.5. Rappresentazione collegata di alberi binari

Gli alberi binari possono essere rappresentati anche mediante la **rappresentazione collegata**. Usando questo metodo, ogni nodo viene

rappresentato mediante una struttura avente i seguenti campi:

1. informazione associata al nodo;
2. riferimento al figlio sinistro del nodo;
3. riferimento al figlio destro del nodo.

È possibile astrarre questo concetto in C definendo un tipo `TipoAbero` che è un puntatore ad un nodo di un albero binario:

```
struct StructAbero {
    TipoInfoAbero info;
    struct StructAbero *destro, *sinistro;
};

typedef struct StructAbero TipoNodoAbero;

typedef TipoNodoAbero* TipoAbero;
```

Osserviamo che:

- il campo `info` memorizza il contenuto informativo del nodo. Il tipo `TipoInfoAbero` verrà definito di volta in volta a seconda della natura delle informazioni che si vogliono memorizzare nei nodi dell'albero binario, analogamente a quanto visto per le strutture collegate lineari.
- i campi `sinistro` e `destro` sono riferimenti ai nodi figlio sinistro e destro del nodo, rispettivamente. Per indicare che il nodo non ha uno o entrambi i figli useremo il valore `NULL`. Un nodo foglia dovrà pertanto avere entrambi i campi `sinistro` e `destro` posti a `NULL`.

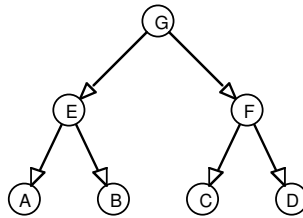
Nota: il tipo `TipoNodoAbero` è simile al tipo `TipoNodoSCL` visto a proposito delle strutture collegate lineari. Il fatto che ogni nodo dell'albero binario contiene due puntatori indica che si ha a che fare con una **struttura collegata non lineare**.

Un albero binario nella rappresentazione collegata sarà rappresentato quindi nel seguente modo:

- L'albero binario *vuoto* viene rappresentato usando il valore `NULL`.

- Un albero binario *non vuoto* viene rappresentato mantenendo il riferimento al nodo radice dell'albero. Osserviamo che partendo dal riferimento alla radice dell'albero è possibile utilizzare i collegamenti ai nodi figli per raggiungere ogni altro nodo dell'albero. In base a questa assunzione, ogni valore di tipo puntatore a *TipoNodoAlbero* consente di accedere a tutto il sottoalbero alla cui radice esso punta.

Esempio: L'albero binario completo mostrato nella figura seguente:



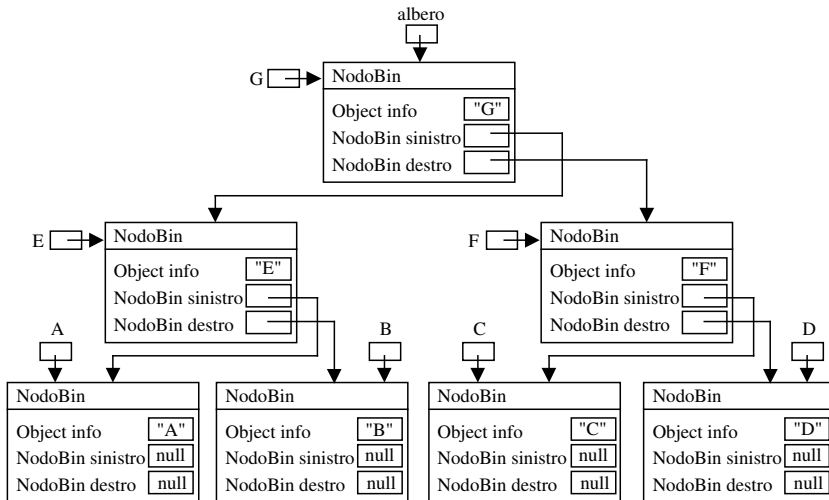
può essere rappresentato in C in modo collegato come illustrato nel seguente frammento di programma:

```

TipoNodoAlbero* nodoalb_alloc(TipoInfoAlbero e) {
    TipoNodoAlbero* r = (TipoNodoAlbero*) malloc(sizeof(
        TipoNodoAlbero));
    r->info = e; r->destrto = NULL; r->sinistro = NULL;
    return r;
}

TipoAlbero A = nodoalb_alloc('A');
TipoAlbero B = nodoalb_alloc('B');
TipoAlbero C = nodoalb_alloc('C');
TipoAlbero D = nodoalb_alloc('D');
TipoAlbero E = nodoalb_alloc('E');
TipoAlbero F = nodoalb_alloc('F');
TipoAlbero G = nodoalb_alloc('G');
E->sinistro = A;    E->destrto = B;
F->sinistro = C;    F->destrto = D;
G->sinistro = E;    G->destrto = F;
TipoAlbero albero = G;
  
```

Dapprima vengono creati i nodi dell'albero, e poi vengono inizializzati i campi corrispondenti ai sottoalberi, come illustrato nella seguente figura:



La variabile `albero` rappresenta quindi l'albero binario e contiene un riferimento alla radice dell'albero.

12.5.1. Implementazione del tipo astratto `AlberoBin`

Una realizzazione del tipo astratto `AlberoBin` usando la rappresentazione collegata è mostrata di seguito. Il tipo astratto `AlberoBin` viene implementato mediante la struttura C `TipoAlbero`, il tipo degli elementi `T` è definito in C con il tipo `TipoInfoAlbero`.

```

// l'albero vuoto e' rappresentato dal valore NULL
TipoAlbero albBinVuoto() {
    return NULL;
}

// costruttore di un albero a partire dai sottoalberi
// e info radice
TipoAlbero creaAlbBin(TipoInfoAlbero infoRadice,
    TipoAlbero sx, TipoAlbero dx) {
    TipoAlbero a = (TipoAlbero) malloc(sizeof(
        TipoNodoAlbero));
    a->info=infoRadice;
    a->sinistro=sx;
    a->destro=dx;
    return a;
}

```

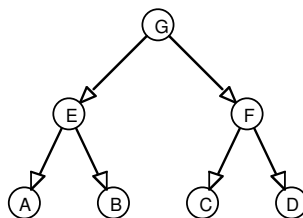
```
// predicato che verifica se l'albero e' vuoto
bool estVuoto(TipoAlbero a) {
    return (a == NULL);
}

// funzione che restituisce il contenuto della radice
TipoInfoAlbero radice(TipoAlbero a) {
    if (a==NULL) {
        printf("ERRORE accesso albero vuoto\n");
        return ERRORE_InfoAlbero;
    }
    else
        return a->info;
}

// funzione che restituisce il sottoalbero sinistro
TipoAlbero sinistro(TipoAlbero a) {
    if (a==NULL) {
        printf("ERRORE accesso albero vuoto\n");
        return NULL;
    }
    else
        return a->sinistro;
}

// funzione che restituisce il sottoalbero destro
TipoAlbero destro(TipoAlbero a) {
    if (a==NULL) {
        printf("ERRORE accesso albero vuoto\n");
        return NULL;
    }
    else
        return a->destro;
}
```

Esempio: L'albero binario:



può essere costruito come mostrato nel seguente programma:

```

TipoAlbero A = creaAlbBin('A', albBinVuoto(),
                          albBinVuoto());
TipoAlbero B = creaAlbBin('B', albBinVuoto(),
                          albBinVuoto());
TipoAlbero C = creaAlbBin('C', albBinVuoto(),
                          albBinVuoto());
TipoAlbero D = creaAlbBin('D', albBinVuoto(),
                          albBinVuoto());
TipoAlbero E = creaAlbBin('E', A, B);
TipoAlbero F = creaAlbBin('F', C, D);
TipoAlbero G = creaAlbBin('G', E, F);

```

12.6. Rappresentazione parentetica di alberi binari

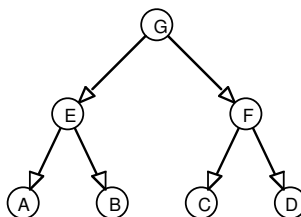
Un'altra possibile rappresentazione di un albero binario è la **rappresentazione parentetica**, basata su stringhe di caratteri. Assumiamo che $f(T)$ sia la stringa che denota l'albero binario T ;

1. l'albero vuoto è rappresentato dalla stringa " $()$ ", cioè $f(\text{albero vuoto}) = "()"$;
2. un albero binario con radice x e sottoalberi sinistro e destro T_1 e T_2 è rappresentato dalla stringa " $(\text{" " + } x + f(T_1) + f(T_2) + \text{" "})$ ".

Esempio: L'albero mostrato nella seguente figura può essere rappresentato mediante la stringa:

$$"(G (E (A () ()) (B () ())) " +$$

$$"(F (C () ()) (D () ())))".$$



12.7. Costruzione di un albero binario da rappresentazione parentetica

Vediamo ora un modo conveniente per costruire un albero binario data una sua rappresentazione parentetica. In particolare, vediamo come progettare una funzione [LeggiAlbero](#) che fornisca questa funzionalità.

```
TipoAlbero LeggiAlbero(char *nome_file) {
    TipoAlbero result;
    FILE *file_albero;
    file_albero = fopen(nome_file, "r");
    result = LeggiSottoAlbero(file_albero);
    fclose(file_albero);
    return result;
}
```

La funzione ha come parametro in ingresso una stringa che rappresenta il nome del file che contiene l'albero e, oltre a gestire le operazioni di apertura e chiusura chiama la funzione ausiliaria [LeggiSottoAlbero](#), che restituisce un puntatore al sottoalbero letto.

```
TipoAlbero LeggiSottoAlbero(FILE *file_albero) {
    char c;
    TipoInfoAlbero i;
    TipoAlbero r;
    c = LeggiParentesi(file_albero); /* legge la
    parentesi aperta */
    c = fgetc(file_albero); /* carattere successivo */
    if (c == ')')
        return NULL; /* se legge () allora l'albero e'
    vuoto */
    else {
        fscanf(file_albero, "%c", &i); /* altrimenti
        legge la radice */
        r = (TipoAlbero) malloc(sizeof(TipoNodoAlbero));
        r->info = i;
        /* legge i sottoalberi */
        r->sinistro = LeggiSottoAlbero(file_albero);
        r->destra = LeggiSottoAlbero(file_albero);
        c = LeggiParentesi(file_albero); /* legge la
        parentesi chiusa */
        return r;
    }
}
```

Si noti che:

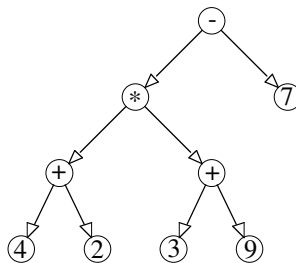
- quando l'albero è vuoto, il carattere che segue la parentesi aperta '(' deve necessariamente essere la parentesi chiusa ')' (senza spazi);
- al contrario quando il nodo non è l'albero vuoto, l'informazione di tipo carattere da associare al nodo deve essere preceduta da uno spazio ' '.
- Se le informazioni sono di tipo `int`, o altro tipo primitivo numerico il tipo della variabile usata per leggere l'informazione ed il formato della istruzione `fscanf` devono essere aggiornate coerentemente;
- se le informazioni sono di altro tipo, in particolare di tipo non primitivo per leggere l'informazione associata diventa necessario progettare una specifica funzione.

12.8. Rappresentazione di espressioni aritmetiche mediante alberi binari

Una espressione aritmetica può essere rappresentata mediante un albero binario in cui:

- i nodi non foglie rappresentano operazioni (es. `+`, `-`, `*`, `/`);
- le foglie rappresentano valori numerici (es. `2`, `7`, ecc.)

Esempio: L'espressione aritmetica $(4 + 2) \cdot (3 + 9) - 7$ può essere rappresentata mediante l'albero binario:



L'albero, a sua volta, può essere rappresentato in modo parentetico come:

```

( -
  ( *
    ( +
      ( 4 ( ) ( ) )
      ( 2 ( ) ( ) )
    )
    ( +
      ( 3 ( ) ( ) )
      ( 9 ( ) ( ) )
    )
  )
  ( 7 ( ) ( ) )
)

```

Sebbene la corrispondenza tra la struttura dell'espressione e quella dell'albero binario sia immediata, la definizione della struttura dati da utilizzare per memorizzare le informazioni nel nodo richiede qualche accorgimento dato che alcuni nodi contengono degli operandi numerici ed altri contengono invece l'indicazione dell'operatore da applicare. Si possono sviluppare diverse alternative (ad esempio utilizzare una struttura con due campi, `char` ed `int` usando un carattere convenzionale per il campo `char`, quando il nodo è una foglia e contiene una informazione di tipo numerico. Lo pseudo-codice di una funzione `valutaEspressione` che, data una espressione rappresentata come albero binario, ne restituisce il valore calcolato può essere definito come:

```

int valutaEspressione(TipoAlbero a) {
    // caso albero vuoto: errore
    if (a==NULL) Errore albero vuoto;
    // caso nodo foglia: restituisce valore
    if (a->sinistro==NULL && a->destra==NULL)
        return valore numerico;
    else { // caso nodo non foglia: calcola ricorsivamente
        // le sottoespressioni destra e sinistra
        int sin = valutaEspressione(a->sinistro);
        int des = valutaEspressione(a->destra);
        // applica operazione
        if (nodo-addizione) return sin + des;
        else ... analogamente per gli altri casi
        else Operazione non valida
    }
}

```

```
}
```

12.9. Visita in profondità di alberi binari

Diversamente dal caso delle strutture lineari, dove è possibile visitare tutti gli elementi in sequenza nell'ordine in cui sono rappresentati, nel caso di alberi binari vi sono molti modi diversi di esplorare i nodi. In particolare, vedremo alcuni algoritmi di visita che consentono di esplorare i nodi di un albero binario. Gli scopi di una visita possono essere molteplici: verificare la presenza di un dato elemento in un albero, calcolare proprietà globali di un albero, come profondità o numero totale di nodi, ecc.

La **visita in profondità** di un albero T può essere realizzata ricorsivamente come il seguente pseudo codice mostra:

```
if  $T$  non è vuoto {  
    analizza il nodo radice di  $T$   
    visita il sottoalbero sinistro di  $T$   
    visita il sottoalbero destro di  $T$   
}
```

L'operazione di "analisi" di un nodo può essere semplicemente la stampa dell'informazione associata al nodo, ma può essere qualunque elaborazione di quell'informazione.

12.9.1. Proprietà della visita in profondità

L'algoritmo di visita in profondità visto precedentemente ha le seguenti proprietà fondamentali:

- termina in un numero finito di passi: infatti ogni chiamata ricorsiva viene applicata su un sottoalbero che è strettamente più piccolo dell'albero corrente;
- visita tutti i nodi di T : per ipotesi induttiva, l'algoritmo visita tutti i nodi dei sottoalberi sinistro e destro di T , se non sono vuoti. Poiché la radice di T viene anch'essa visitata, allora tutti i nodi di T vengono visitati;

- richiede un numero totale di passi proporzionale ad n , dove n è il numero di nodi di **T**: infatti ogni nodo viene considerato al più una volta durante la visita (e questo avviene quando il nodo è la radice del sottoalbero corrente) e l'algoritmo effettua un numero costante di passi per passare da un nodo all'altro.

12.9.2. Tipi diversi di visite in profondità: in preordine, simmetrica, in postordine

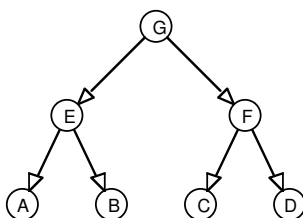
Le tre operazioni della visita in profondità:

- visita della radice;
- chiamata ricorsiva sul sottoalbero sinistro;
- chiamata ricorsiva sul sottoalbero destro;

possono essere realizzate in qualunque ordine. In particolare, consideriamo tre particolari ordini che danno luogo a tre diverse versioni di visita in profondità:

- **visita in preordine**: radice, sottoalbero sinistro, sottoalbero destro;
- **visita simmetrica**: sottoalbero sinistro, radice, sottoalbero destro;
- **visita in postordine**: sottoalbero sinistro, sottoalbero destro, radice.

Esempio: Consideriamo l'albero seguente:



Se l'operazione di visita è la stampa dell'informazione del nodo, si hanno i seguenti tre risultati applicando i diversi tipi di visita:

- visita in preordine: **GEABFC D**
- visita simmetrica: **AEBGCFD**
- visita in postordine: **ABECDFG**

Si noti che la radice **G** è la prima ad essere visitata nel caso di visita in preordine, è visitata a metà della visita nel caso di visita simmetrica, ed è visitata per ultima nel caso di visita in postordine.

12.10. Realizzazione della visita in profondità

Mostriamo ora come realizzare in C le operazioni di visita in profondità di un albero binario.

12.10.1. Rappresentazione tramite array

Consideriamo prima il caso della visita in preordine con la rappresentazione tramite array.

La visita inizia con la chiamata ad una funzione ausiliaria che ha come parametro ulteriore l'indice del nodo da visitare.

```
/* visita in preordine di un albero binario
   rappresentato con array */

void VisitaPreordine(TipoAlbero a)
{
    /* visita a partire dalla radice */
    VisitaIndicePre(a, 0);
}
```

La funzione ausiliaria è ricorsiva e l'ordine di esecuzione delle operazioni corrisponde a quello della visita in preordine.

```
/* visita in preordine */
void VisitaIndicePre(TipoAlbero a, int i) {
    if (i >= 0 && i < MaxNodiAlbero && a[i].esiste) {
        /* albero non vuoto */
        Analizza(a[i].info); // analizza la radice
        VisitaIndicePre(a, i * 2 + 1); // visita sinistro
        VisitaIndicePre(a, i * 2 + 2); // visita destro
    }
}
```

12.10.2. Rappresentazione tramite puntatori

Consideriamo ora la visita in profondità in postordine nella rappresentazione tramite struttura collegata:

```
void VisitaPostordine(TipoAlbero a) {
    if (a != NULL) {
        /* albero non vuoto */
        VisitaPostordine(a->sinistro); // visita sinistro
        VisitaPostordine(a->destra);   // visita destro
        Analizza(a->info);             // analizza la radice
    }
}
```

Nota: Semplicemente spostando la posizione dell'operazione di analisi si ottengono le tre varianti preordine, simmetrica, postordine.

12.11. Applicazioni delle visite

Mostriamo ora come adattare l'algoritmo di visita in profondità per realizzare alcune operazioni di interesse generale su alberi binari.

12.11.1. Calcolo della profondità di un albero

Si vuole definire una funzione che restituisce la profondità dell'albero passato come parametro:

```
/* calcola la profondita' di un albero binario */
int Profondita(TipoAlbero albero)
{
    int s, d;
    if (albero == NULL)
        return -1; // l'albero vuoto ha profondita' -1
    else {
        // calcola la profondita' dei sottoalberi
        s = Profondita(albero->sinistro);
        d = Profondita(albero->destra);
        // l'albero complessivo ha profondita' max(s,d) + 1
        return (Massimo(s, d) + 1);
    }
}
```

Il metodo verifica dapprima se l'albero (o sottoalbero) passato come parametro è vuoto, nel qual caso restituisce come profondità -1 . Questo è il passo base della ricorsione. Altrimenti, vengono calcolate ricorsivamente la profondità del sottoalbero sinistro e destro, e viene restituito il massimo di queste due quantità più uno.

```

/* determina il massimo fra due interi */
int Massimo(int a, int b) {
    if (a > b)
        return a;
    else
        return b;
}

```

12.11.2. Verifica della presenza di un elemento nell'albero

Una operazione di grande rilevanza applicativa è la ricerca di un dato valore in un albero binario. Si vuole definire una funzione che, dato un valore, restituisce **true** se il valore è presente nell'albero su cui è invocato, e **false** altrimenti. Nel caso in cui il valore venga trovato, la funzione deve restituire anche il puntatore al nodo corrispondente.

```

bool RicercaAlbero(TipoAlbero A, TipoInfoAlbero elem,
    TipoAlbero *posiz) {
    bool trovato = false; *posiz=NULL;
    if (A == NULL)
        trovato = false;
    else if (elem == A->info) { // elemento trovato
        *posiz = A; trovato = true;
    }
    else {
        /* cerca nel sottoalbero sinistro */
        trovato = RicercaAlbero(A->sinistro, elem, posiz);
        if (!trovato)
            /* cerca nel sottoalbero destro */
            trovato = RicercaAlbero(A->destra, elem, posiz);
    }
    return trovato;
}

```

La funzione **RicercaAlbero** verifica dapprima il caso base in cui l'albero è vuoto, nel qual caso la risposta è **false**. Se invece l'albero non è vuoto, il valore è presente se e solo se è verificato almeno uno dei tre casi seguenti ed assegna il valore **true** alla variabile **trovato**, che altrimenti rimane con il valore **false**:

1. il valore è contenuto nella radice correntemente visitata (ovvero, **elem == A->info** esegue **trovato=true**);
2. il valore è contenuto nel sottoalbero sinistro della radice (ovvero, **RicercaAlbero(A->sinistro, elem, posiz)** restituisce **true**);

3. il valore è contenuto nel sottoalbero destro della radice (ovvero `RicercaAlbero(A->destro, elem, posiz)` restituisce `true`).

Nota: La funzione `RicercaAlbero` potrebbe dover visitare *tutti* i nodi prima di stabilire con certezza se un valore dato è presente o meno nell'albero. Il costo computazionale è quindi $O(n)$. Vedremo più avanti che usando alberi con determinate proprietà (alberi binari di ricerca) è possibile usare un metodo diverso e molto più efficiente per ricercare un valore.

Nota: Se il tipo delle informazioni memorizzate nel nodo dell'albero non è un tipo primitivo occorre definire una opportuna funzione che verifica l'uguaglianza di due elementi del tipo delle informazioni contenute nell'albero.

12.12. Accesso all'albero per livelli

In diverse situazioni è necessario accedere ai nodi di un albero procedendo per livelli.

12.12.1. Stampa dei nodi di un livello

Mostriamo una funzione che, presi in input un albero a (contenente valori interi) ed un livello l , stampa l'insieme dei nodi di a che si trovano al livello l . La funzione implementa una visita in profondità ricorsiva in cui il livello d'interesse viene decrementato ogni volta che si scende di livello. Il contenuto della radice viene stampato solo quando tale livello è pari a 0.

```
void stampaNodiLivello(TipoAlbero a, int livello){
    if (estVuoto(a)) return;
    if (livello == 0){
        printf("%d ", radice(a));
        return;
    }
    stampaNodiLivello(sinistro(a), livello-1);
    stampaNodiLivello(destro(a), livello-1);
}
```

12.12.2. Insieme dei nodi di un livello

Mostriamo a seguire una funzione che restituisce l'insieme degli elementi presenti nei nodi di un dato livello.

```
Insieme* elementiLivello(TipoAlbero a, int livello) {
    Insieme* ris = insiemeVuoto();
    elementiLivelloRic(a, livello, ris);
    return ris;
}

void elementiLivelloRic(TipoAlbero a, int livello,
    Insieme* ins) {
    if (estVuoto(a)) return;
    if (livello == 0){
        inserisci(ins, radice(a));
    }
    elementiLivelloRic(sinistro(a), livello-1, ins);
    elementiLivelloRic(destro(a), livello-1, ins);
}
```

In questo caso viene usata una funzione ausiliaria per effettuare la ricorsione sui sottoalberi del nodo corrente. Anche questa funzione è essenzialmente una visita in profondità ricorsiva in cui viene usato un parametro per individuare il livello della radice del sottoalbero corrente.

La seguente variante non fa uso della funzione ausiliaria. Si noti che l'insieme `des` deve essere deallocato quando non più necessario (la deallocazione è omessa per brevità).

```

Insieme* elementiLivelloB(TipoAlbero a, int livello){
    if (estVuoto(a) || livello < 0){
        Insieme* ris = insiemeVuoto();
        return ris;
    }
    if(livello == 0){
        Insieme* ris = insiemeVuoto();
        inserisci(ris,radice(a));
        return ris;
    }
    if (livello > 0){
        Insieme* ris = elementiLivelloB(sinistro(a),
        livello-1);
        Insieme* des = elementiLivelloB(destro(a),livello
        -1);
        IteratoreInsieme* it = creaIteratoreInsieme(des);
        while(hasNext(it)){
            inserisci(ris, next(it));
        }
        // Deallocazione des (omessa per brevit )
        return ris;
    }
    return NULL; // non si verifica mai
}

```

12.13. Visita in ampiezza di alberi binari

Una modalit  alternativa di visita dei nodi di un albero   la cosiddetta visita in “ampiezza”. Con questo termine si intende una visita in cui un nodo al livello l viene visitato solo dopo che tutti i nodi al livello $l - 1$ sono stati visitati. Contrariamente a quanto accade per la visita in profondit , l’implementazione della visita in ampiezza risulta pi  agevole nella variante iterativa; questa variante sfrutta una coda per memorizzare i nodi da visitare in maniera tale da garantire che l’ordine di visita corrisponda a quello di inserimento.

La **visita in ampiezza** di un albero T pu  essere realizzata iterativamente come mostrato dal seguente pseudocodice:

```

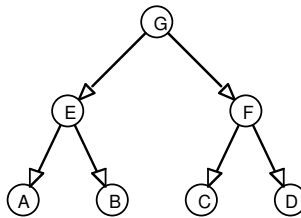
sia C una coda vuota di nodi di un albero
inserisci la radice di T in C
while(C non   vuota){
    estrai il nodo n in testa alla coda C
    visita n
    inserisci il figlio sinistro di n in C
}

```

```
    inserisci il figlio destro di n in C  
}
```

12.13.1. Esempio di visita in ampiezza di alberi binari

Esempio: Consideriamo l'albero seguente:



I nodi dell'albero vengono visitati nel seguente ordine:

G E F A B C D

12.13.2. Proprietà della visita in ampiezza

L'algoritmo di visita in ampiezza visto precedentemente ha le seguenti proprietà fondamentali:

- termina in un numero finito di passi: in un albero binario, nessun nodo è padre dei suoi antenati (incluso il nodo stesso) quindi, una volta estratto, non viene più inserito nella coda. Questo, insieme all'osservazione che un albero binario contiene un numero finito di nodi e ad ogni iterazione viene estratto esattamente un nodo, implica che la coda si svuota in un numero finito di passi, quindi l'algoritmo termina.
- visita tutti i nodi di **T**: è facile vedere che se un nodo viene inserito nella coda, sarà necessariamente visitato e tutti i suoi figli saranno inseriti nella coda a loro volta. Questo processo termina solo quando tutti i nodi sono stati inseriti, quindi estratti.
- richiede un numero totale di passi proporzionale ad n , dove n è il numero di nodi di **T**: ogni nodo viene inserito esattamente una volta nella coda, e l'algoritmo effettua un numero costante di passi per inserirne i figli.

12.13.3. Realizzazione della visita in ampiezza

Mostriamo ora l'implementazione in C della visita in ampiezza di un albero binario. Assumiamo che siano disponibili:

- un'implementazione con side-effect del tipo astratto Coda i cui elementi siano di tipo [TipoAlbero](#) (si faccia riferimento alle implementazioni dei capitoli precedenti). Non è necessaria alcuna assunzione circa la modalità di rappresentazione della coda.
- un'implementazione con rappresentazione collegata del tipo astratto Albero Binario contenente elementi di tipo [T](#) (si faccia riferimento all'implementazione sopra riportata).

```
void visitaAmpiezza(TipoAlbero a){
    if (estVuoto(a)) return;
    Coda* c = codaVuota();
    inCoda(c,a);
    while(!estVuoto(c)){
        TipoAlbero n = primo(c);
        analizza(n);
        outCoda(c);
        if (!estVuoto(sinistro(n)) inCoda(c,sinistro(n)));
        if (!estVuoto(destro(n)) inCoda(c,destro(n)));
    }
}
```

Nel caso in cui si dovesse realizzare la funzione di visita accedendo alla rappresentazione, sarebbe sufficiente rimpiazzare le funzioni del tipo astratto con le operazioni sulla rappresentazione. A titolo esemplificativo mostriamo il codice per la realizzazione della visita nel caso di rappresentazione collegata (riportata per comodità).

```

typedef struct StructAlbero {
    TipoInfoAlbero info;
    struct StructAlbero* destro = NULL;
    struct StructAlbero* sinistro = NULL;
} TipoNodoAlbero;

typedef TipoNodoAlbero* TipoAlbero;

void visitaAmpiezza(TipoAlbero a){
    if (a == NULL) return;
    Coda* c = codaVuota();
    inCoda(c,a);
    while(!estVuota(c)){
        TipoAlbero n = a;
        analizza(n);
        outCoda(c);
        if (n->sinistro != NULL) inCoda(c,n ->
sinistro);
        if (n->destro != NULL) inCoda(c,n -> destro);
    }
}

```

12.14. Applicazioni della visita in ampiezza

In molte situazioni le visite in profondità e in ampiezza possono essere utilizzate in maniera equivalente. Si pensi ad esempio al conteggio del numero di nodi di un albero o alla verifica della presenza di un elemento in un albero. Esistono tuttavia situazioni in cui la visita in ampiezza si presta in maniera più naturale a realizzare la funzione desiderata; ne mostriamo a seguire un esempio.

12.14.1. Ricerca di un elemento a profondità minima

Si vuole definire una funzione che, dati un albero binario a ed un elemento e , restituisca il riferimento ad un nodo di a di livello minimo contenente e ($NULL$ se non presente). Mostriamo l'implementazione della funzione nel caso di rappresentazione collegata dell'albero.

```

TipoNodoAlbero* nodoProfonditaMin(TipoAlbero a,
    TipoInfoAlbero e){
    if (a == NULL) return;
    TipoNodoAlbero* n;
    Coda* c = codaVuota(); // Coda di TipoAlbero (
    ovvero TipoNodoAlbero*)
    inCoda(c,a);
    while(!estVuota(c)){
        n = primo(c);
        if (n -> info == e) return n;
        outCoda(c);
        if (n->sinistro != NULL) inCoda(c,n ->
            sinistro);
        if (n->destro != NULL) inCoda(c,n -> destro);
    }
    return NULL;
}

```

La funzione restituisce il primo nodo visitato contenente e . Poiché essa implementa una visita in ampiezza (come si può vedere confrontando la struttura della funzione con quella della visita in ampiezza), tale nodo è necessariamente quello di profondità minima.

Si osservi che con questa implementazione non è sempre necessaria una visita esaustiva dell'albero, in quanto la funzione termina non appena viene trovato un nodo contenente e .

L'implementazione tramite visita in profondità, sebbene possibile, risulterebbe inutilmente complicata. Questa infatti richiederebbe di effettuare una visita esaustiva dell'albero, memorizzando tutti i nodi contenuti e e la relativa profondità (anch'essa da calcolare), a cui far seguire la selezione del nodo di profondità minima.

12.15. Visita in profondità iterativa

La visita in profondità sfrutta una pila per mantenere l'ordine di visita dei nodi. Nell'implementazione ricorsiva ciò avviene in maniera implicita in quanto ad ogni invocazione ricorsiva corrisponde l'allocazione di un nuovo record di attivazione in cima alla pila. Lo stesso approccio può essere implementato in maniera esplicita, tramite l'uso di una pila, procedendo iterativamente. Mostriamo a seguire tale implementazione.

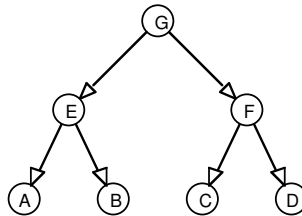
```

void visitaProfonditaIterativa(TipoAlbero a){
    if (a == NULL) return;
    Pila* p = pilaVuota();
    push(p,a);
    while(!estVuota(p)){
        TipoAlbero n = car(p);
        analizza(n);
        pop(p);
        if (n->destro != NULL) push(p,n -> destro);
        if (n->sinistro != NULL) push(p,n -> sinistro)
    }
}

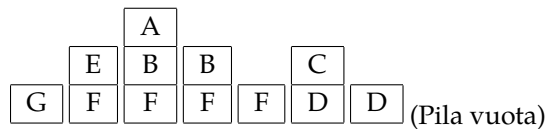
```

Si osservi che l'ordine di inserimento dei nodi nella pila è invertito rispetto all'ordine con cui vengono visitati i sottoalberi destro e sinistro nella visita ricorsiva.

Consideriamo ora l'evoluzione della pila **p** a seguito dell'invocazione della funzione **visitaProfonditaIterativa** sull'albero seguente:



Mostriamo lo stato della pila all'inizio di ogni iterazione, quando viene effettuato il test del ciclo **while**:



Poiché ad ogni iterazione la funzione visita l'elemento affiorante della pila (rimpiazzandolo con i suoi figli), è immediato verificare che l'ordine di visita dei nodi è il seguente:

G E A B F G D

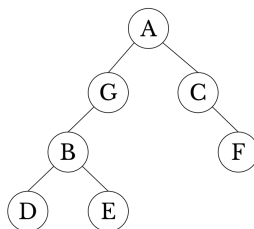
12.16. Operazioni di modifica di alberi

Le operazioni del tipo astratto **AlberoBin** sopra introdotte permettono la costruzione e la visita di un albero binario. Mostriamo ora, a titolo

esemplificativo, l'implementazione di alcune operazioni di inserimento e cancellazione.

12.16.1. Inserimento di un nodo come foglia in posizione data

Dato un albero a vogliamo inserire un nuovo nodo come foglia, in posizione specificata tramite una stringa opportuna. La stringa è una sequenza di caratteri 'd' ed 's', dove 's' indica il figlio sinistro e 'd' il figlio destro di un nodo. Partendo dalla radice dell'albero e dal primo carattere della stringa, ci si sposta sul figlio destro o sinistro a seconda del carattere incontrato, finché la stringa finisce o si è raggiunto un nodo privo del figlio specificato. Nel primo caso, l'albero non viene modificato, nel secondo si inserisce un nuovo nodo con l'elemento fornito in input nella posizione in cui ci si dovrebbe spostare. Ad esempio, dato il seguente albero:



la stringa "ssd" rappresenta il nodo **E**, mentre la stringa "sd" rappresenta il figlio destro (non presente) del nodo **G**. Nel primo caso, non deve essere effettuata nessuna modifica all'albero, mentre nel secondo caso, deve essere inserito un nuovo nodo come figlio destro di **G**. La funzione seguente inserisce un elemento come nodo in posizione data.

```

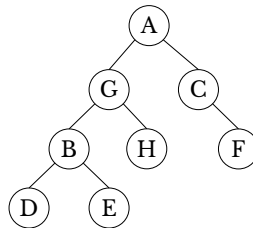
void inserisciFogliaInPosizione(TipoAlbero* a,
    TipoInfoAlbero e, char* p){
    if (a == NULL){
        printf("ERRORE: puntatore NULL")
        return;
    }
    if(estVuoto(*a)){
        *a = creaAlbBin(e, NULL, NULL);
        return;
    }
    if(*p=='\0') return;
    if(*p=='s') inserisciFogliaInPosizione(&((*a)->
        sinistro), e, p+1);
    if(*p=='d') inserisciFogliaInPosizione(&((*a)->
        destro), e, p+1);
}

```

Ad esempio se `a` è l'albero mostrato sopra, l'invocazione:

```
inserisciFogliaInPosizione(a, 'H', "sd")
```

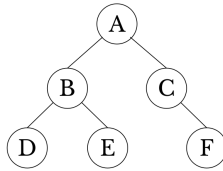
modifica `a` producendo il seguente risultato:



Si osservi che fornendo in input qualunque estensione della stringa "sd", ad esempio "sds" o "sdsssd", etc. si ottiene lo stesso risultato, in quanto il nodo viene aggiunto nel momento in cui non è più possibile proseguire lungo il percorso rappresentato dalla stringa.

12.16.2. Inserimento di un nodo interno in posizione data

L'inserimento di un nodo interno pone difficoltà aggiuntive rispetto all'inserimento come foglia, in quanto occorre gestire il posizionamento del sottoalbero che viene "staccato" per fare posto al nuovo nodo. Per vedere ciò, si consideri nuovamente il seguente albero:



Si supponga di voler inserire un nuovo nodo **G** nella posizione attualmente occupata dal nodo **B**. In che relazione deve essere il nuovo nodo con il sottoalbero avente **B** per radice? In generale, tale relazione dipende dal caso in esame.

Mostriamo a seguire un'estensione della funzione [inserisciFogliaInPosizione](#) che inserisce un nuovo nodo nella posizione identificata da una stringa fornita in input (secondo la convenzione mostrata sopra). La funzione prende in input un parametro aggiuntivo di tipo carattere, usato per determinare se il sottoalbero disconnesso per far posto al nuovo nodo debba essere figlio destro (carattere 'd') o sinistro ('s') del nuovo nodo. Nel caso la stringa identifichi una foglia, la funzione si comporta in maniera analoga a [inserisciFogliaInPosizione](#).

```

void inserisciInPosizione(TipoAlbero* a,
    TipoInfoAlbero e, char* p, char f){
    if (a == NULL){
        printf("ERRORE: puntatore NULL");
        return;
    }
    if(estVuoto(*a)){
        *a = creaAlbBin(e, NULL, NULL);
        return;
    }
    if(*p=='\0'){
        if (f == 'd'){
            *a = creaAlbBin(e, NULL, *a);
        }
        if (f == 's'){
            *a = creaAlbBin(e, *a, NULL);
        }
        return;
    }
    if(*p=='s') inserisciInPosizione(&((*a)->sinistro),
        e, p+1, f);
    if(*p=='d') inserisciInPosizione(&((*a)->destrto), e,
        p+1, f);
}
  
```

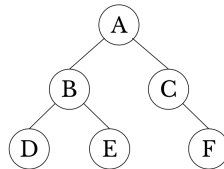
12.16.3. Cancellazione di un nodo

Quando si vuole cancellare un nodo n da un albero binario, possono verificarsi due casi:

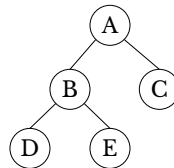
- n ha uno o nessun figlio
- n ha entrambi i figli

Nel primo caso si procede semplicemente eliminando il nodo n . Nel caso questo sia una foglia, nessuna ulteriore operazione è necessaria. Se invece n ha un figlio, viene usato per rimpiazzare n .

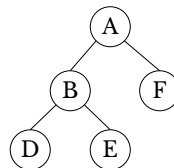
Nel seguente albero di esempio:



L'eliminazione del nodo **F** deve produrre il seguente risultato:

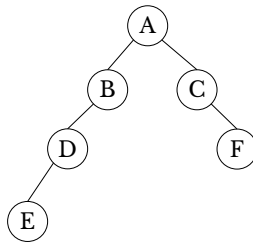


L'eliminazione del nodo **C** deve produrre il seguente risultato:

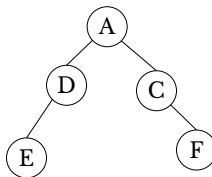


Nel secondo caso, invece, non esiste una soluzione unica e la scelta di come gestire i sottoalberi del nodo eliminato dipende dal caso in esame. A scopo illustrativo, mostriamo una possibile soluzione in cui il sottoalbero destro del nodo da eliminare viene prima reso sottoalbero sinistro della foglia più a sinistra del sottoalbero con radice il nodo da eliminare. Questo comporta che il nodo selezionato abbia un solo figlio, per cui l'eliminazione ricade in uno dei casi precedenti.

Ad esempio, l'eliminazione del nodo **B** dal primo albero viene effettuata rendendo dapprima il nodo **E** figlio sinistro della foglia **D**, ottenendo il seguente albero:



e successivamente eliminando il nodo **B**, ottenendo pertanto il seguente risultato:



Mostriamo di seguito una possibile implementazione della funzione. La funzione **eliminaNodo** elimina la prima occorrenza del nodo che incontra. Si osservi che la funzione restituisce un valore booleano per indicare se l'eliminazione abbia avuto luogo (valore *true*) o meno (*false*). Inoltre, viene usata la funzione ausiliaria **fogliaSinistra** per individuare la foglia più a sinistra di un albero.

```

TipoAlbero fogliaSinistra(TipoAlbero a){
    if (estVuoto(a)) return NULL;
    if (!estVuoto(sinistro(a))) return fogliaSinistra(
        sinistro(a));
    if (!estVuoto(destro(a))) return fogliaSinistra(
        destro(a));
    return a;
}

bool eliminaNodo(TipoAlbero* a, TipoInfoAlbero e){
    if (a == NULL){
        printf("ERRORE: puntatore NULL");
        return false;
    }
    if(estVuoto(*a)){
        return false;
    }
    if (radice(*a) == e){
        // Rende il sottoalbero destro sottoalbero
        // sinistro della foglia piu' a sinistra di *a
        TipoAlbero dx = destro(*a);
        if (!estVuoto(dx)){
            (*a) -> destro = NULL;
            fogliaSinistra(*a) -> sinistro = dx;
        }
        // Elimina la radice di *a
        TipoAlbero n = *a;
        *a = sinistro(*a);
        free(n); // Dealloca il nodo eliminato
        return true;
    }
    return (eliminaNodo(&((*a)->sinistro),e) ||
        eliminaNodo(&((*a)->destro),e));
}

```

Si osservi che la funzione mostrata gestisce correttamente anche il caso in cui il nodo da eliminare sia una foglia o abbia un solo figlio.

12.17. Alberi binari di ricerca

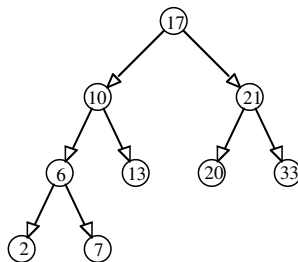
Concludiamo la trattazione sugli alberi binari discutendo un sottoinsieme degli alberi binari: gli **alberi binari di ricerca**. Gli alberi binari di ricerca sono strutture dati che consentono di cercare informazioni in modo molto efficiente. In un albero binario di ricerca, il dominio delle informazioni associabili ai nodi deve essere un insieme su cui è definita una relazione di ordine totale.

Un albero binario di ricerca può essere:

- un albero vuoto, oppure

- un albero binario non vuoto tale che:
 - il valore di ogni nodo nel sottoalbero sinistro è minore o uguale al valore della radice rispetto alla relazione d'ordine;
 - il valore di ogni nodo nel sottoalbero destro è maggiore o uguale al valore della radice rispetto alla relazione d'ordine;
 - i sottoalberi sinistro e destro sono a loro volta alberi binari di ricerca.

Esempio: L'albero mostrato nella figura seguente è un albero binario di ricerca dove il dominio delle informazioni associate ai nodi è l'insieme dei numeri interi.



12.17.1. Verifica della presenza di un elemento in un albero binario di ricerca

Mostriamo ora una variante della funzione vista precedentemente che sfrutta le proprietà peculiari degli alberi binari di ricerca per verificare la presenza di un elemento nell'albero in modo molto efficiente. Assumiamo che i nodi dell'albero contengano numeri interi come nell'esempio precedente.

```

bool presenteAlberoRicerca(TipoAlbero a,
    TipoInfoAlbero elem) {
    if (a==NULL)
        return false;
    else
        if (a->info == elem) // elemento nella radice
            return true;
        else if (a->info > elem) // l'elemento puo' essere
            // solo nel sottoalbero sin
            return presenteAlberoRicerca(a->sinistro, elem);
        else // l'elemento puo' essere solo nel
            sottoalbero des
            return presenteAlberoRicerca(a->destro, elem);
}

```

Se l'albero è vuoto, il metodo restituisce subito **false**. Altrimenti, l'elemento **elem** viene confrontato con la radice (**a->info == elem**):

- se il confronto ha successo, l'elemento cercato è nella radice (**return true**);
- se il valore nella radice dell'albero è maggiore di quello cercato, allora **elem** precede strettamente radice(), e quindi può solo essere presente nel sottoalbero sinistro (**return presenteAlberoRicerca(a->sinistro, elem)**);
- se il valore nella radice dell'albero è minore di quello cercato, allora **elem** segue strettamente radice(), e quindi può solo essere presente nel sottoalbero destro (**return presenteAlberoRicerca(a->destro, elem)**).

Nota: La funzione **presenteAlberoRicerca** effettua al più una chiamata ricorsiva, e non due come nella versione su alberi binari generali. In questo modo, scartando ad ogni chiamata ricorsiva un intero sottoalbero, il metodo esplorerà al massimo tanti nodi quanto è la profondità dell'albero, e non tutti i nodi dell'albero. Se il metodo viene applicato ad un albero binario di ricerca completo con n nodi, esso richiede un numero di passi proporzionale alla profondità dell'albero, che è $\log_2 n$. Questo metodo è pertanto sostanzialmente più efficiente ($O(\log(n))$), rispetto al procedimento generale che ha costo $O(n)$.