

Algoritmi e Strutture Dati

Schifano S. Fabio `schifano@fe.infn.it`

Strutture dati: Alberi

La struttura dati **albero** o **tree** organizza i dati in modo non sequenziale ma gerarchico, cioè tra i componenti della struttura detti **nodi** possiamo stabilire una relazione di **parentela**, per cui possiamo identificarli come **nodi padre**, **nodi figlio** e **nodi fratello**.

La struttura dati albero è rappresentata da un insieme di **nodi** tra i quali distinguiamo:

- ▷ il nodo **radice** caratterizzato dal fatto di **non** essere figlio di nessuno
- ▷ i nodi **intermedi** i quali sono figli di altri nodi ed a loro volta possono avere altri figli
- ▷ i nodi **foglia** caratterizzati dal fatto di **non** avere alcun figlio

La terminologia usata per trattare gli alberi è la seguente:

- ▷ sia T un albero di n nodi di radice r
- ▷ siano T_1, \dots, T_k i k insiemi disgiunti in cui sono partizionati i rimanenti nodi
- ▷ siano r_1, \dots, r_k le k radici di tali insiemi

allora diremo che:

- ▷ r_1, \dots, r_k sono tra di loro **fratelli**
- ▷ r_1, \dots, r_k sono **figli** di r
- ▷ r è il **padre** di r_1, \dots, r_k

Inoltre:

- ▷ un **discendente** di un nodo n é o un figlio di n stesso o un figlio di un discendente di n
- ▷ un **antenato** di un nodo n é il padre di n stesso o il padre di un antenato di n
- ▷ un nodo senza figli é detto **nodo foglia**
- ▷ un nodo senza padre é detto **nodo radice**

In un albero possiamo quindi distinguere due livelli di **ordinamento**:

- ▷ quello **verticale**, cioè in base alla distanza dei nodi dal nodo radice. Quindi:
 - ▷ la radice é a livello 0
 - ▷ i figli del nodo radice sono a livello 1,
 - ▷ i figli dei figli sono a livello 2, . . .
- ▷ quello **orizzontale** per cui possiamo distinguere tra il **primo** (o **figlio maggiore**), **secondo**, . . ., ultimo figlio (o **figlio minore**)

Un albero può essere pensato come una **generalizzazione** di un di una struttura dati sequenziale tipo **lista**:

- ▷ una lista é un particolare albero in cui ogni nodo ha un solo figlio
- ▷ un albero é una generalizzazione di una lista in cui ogni elemento ha un predecessore, il padre, ma ha più di un successore.

Alberi: Specifica

Un tipico insieme di operazioni per gli alberi comprende:

- ▷ CREAALBERO, crea un albero vuoto
- ▷ RADICE, per accedere direttamente alla radice dell'albero
- ▷ PADRE, per accedere al padre di un nodo
- ▷ PRIMOFIGLIO, per accedere al primo figlio di un nodo
- ▷ SUCCFRATELLO, per accedere al successivo fratello di un nodo
- ▷ ALBEROVUOTO, per verificare se un albero contiene nodi o meno
- ▷ FOGLIA, per verificare se un nodo é un nodo foglia
- ▷ FINEFRATELLI, per verificare, dati un nodo, se non ci sono piú fratelli
- ▷ INSRADICE, per inserire la radice in un albero vuoto
- ▷ CANCSOTTOALBERO, per cancellare un sottoalbero, se il sottoalbero é composto di un solo nodo cancella un nodo
- ▷ INSSOTTOALBERO, per inserire un sottoalbero, se il sottoalbero é composto di un solo nodo aggiunge un nodo

Indicando con **albero** il nome di tipo della struttura dati albero, con **nodo** il nome di tipo della struttura nodo di un albero, e con T un generico albero, possiamo dare la seguente specifica sintattica e semantica

- ▷ CREALABERO: $() \longrightarrow \text{albero}$
 $\text{CREALABERO}() = T'$
Pre:
Post $T' = \Lambda$
- ▷ ALBEROVUOTO: $(\text{albero}) \longrightarrow \text{boolean}$
 $\text{ALBEROVUOTO}(T) = b$
Pre:
Post: $b = \text{True}$ se $T = \Lambda$, $b = \text{False}$ altrimenti
- ▷ INSRADICE: $(\text{nodo}, \text{albero}) \longrightarrow \text{albero}$
 $\text{INSRADICE}(u, T) = T'$
Pre: $T = \Lambda$
Post: T' é un albero con il solo nodo radice u
- ▷ RADICE: $\text{albero} \longrightarrow \text{nodo}$
 $\text{RADICE}(T) = u$
Pre: $T \neq \Lambda$
Post: u é la radice di T
- ▷ PADRE: $(\text{nodo}, \text{albero}) \longrightarrow \text{nodo}$
 $\text{PADRE}(u, T) = v$
Pre: $T \neq \Lambda$ e u non sia la radice di T
Post: v é il padre di u

- ▷ FOGLIA: (nodo, albero) \longrightarrow boolean
FOGLIA(u, T) = b
Pre: $T \neq \Lambda$ e $u \in T$
Post: $b = \text{True}$ se u non ha figli, $b = \text{False}$ altrimenti

- ▷ PRIMOFIGLIO: (nodo, albero) \longrightarrow nodo
PRIMOFIGLIO(u, T) = v
Pre: $T \neq \Lambda$ e u sia nodo intermedio di T
Post: v é il primo figlio di u secondo la relazione di ordinamento dei figli di un nodo

- ▷ FINEFRATELLI: (nodo, albero) \longrightarrow boolean
FINEFRATELLI(u, T) = b
Pre: $T \neq \Lambda$ e u sia un nodo di T o un particolare nodo s detto **sentinella** che segue l'ultimo figlio
Post: $b = \text{True}$ se $u = s$, $b = \text{False}$ altrimenti

- ▷ SUCCFRATELLO: (nodo, albero) \longrightarrow nodo
SUCCFRATELLO(u, T) = v
Pre: $T \neq \Lambda$ e u sia un nodo di T
Post: v é il fratello minore di u seconda la relazione di precedenza stabilita sui fratelli.
 $v = s$ se u é l'ultimo fratello

▷ $\text{INSSOTTOALBERO}(\text{nodo}, \text{nodo}, \text{albero}, \text{albero}) \longrightarrow \text{albero}$

$\text{INSSOTTOALBERO}(n, m, T, U) = T'$

Pre:

▷ $T \neq \Lambda, U \neq \Lambda$

▷ n, m due nodi di T tali che m é figlio di n oppure $m = n$

Post:

▷ se $m \neq n$, quindi m figlio di n per ipotesi, allora T' é ottenuto aggiungendo il sottoalbero U a all'albero T in modo tale che:

▷ la radice u dell'albero U sia un figlio di n

▷ la radice u dell'albero U sia il fratello successivo di m

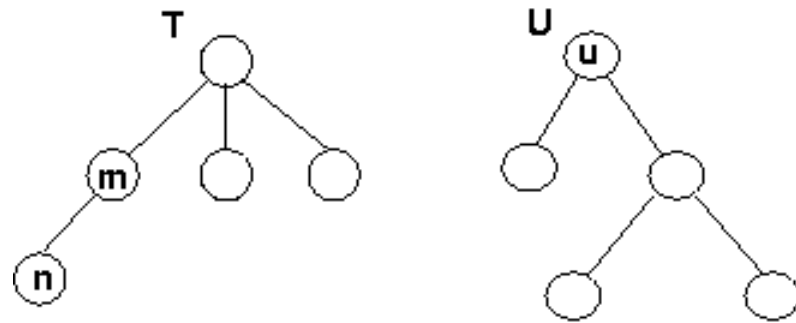
▷ se $m = n$ allora T' é ottenuto come prima ma la radice u dell'albero U diviene il primo figlio di n .

▷ $\text{CANCSOTTOALBERO}(\text{nodo}, \text{albero}) \longrightarrow \text{albero}$

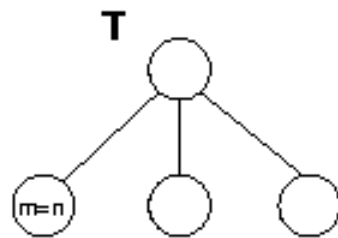
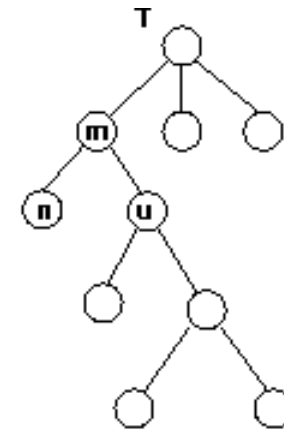
$\text{CANCSOTTOALBERO}(u, T) = T'$

Pre: $T \neq \Lambda$ e u sia un nodo di T

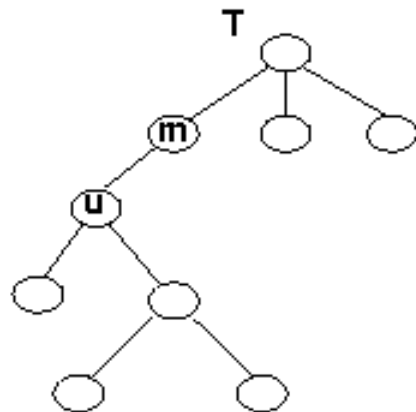
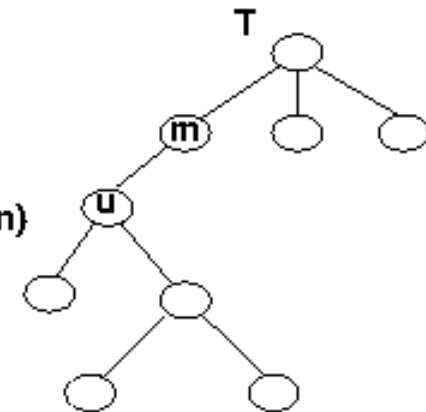
Post: T' é ottenuto da T togliendo il sottoalbero di radice u



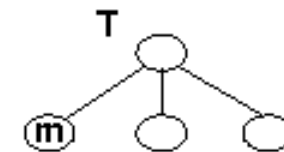
INSSOTTOALBERO(T,U,m,n)



INSSOTTOALBERO(T,U,m,m)



CANCELLASOTTOALBERO(u,T)



Algoritmi sugli alberi: Visite

L'algoritmo di **visita** di un albero consiste nell'esaminare ogni nodo dell'albero esattamente una volta.

La visita dei nodi può essere eseguita in vari modi detti **ordine di visita**.

Sia T un albero di radice r . Se r ha k figli, indichiamo con T_1, \dots, T_k i k sottoalberi aventi come radice i k figli del nodo r .

Possiamo allora definire tre ordini di visita principali:

- ▷ algoritmo di visita in ordine **anticipato** o **previsita**, che consiste dei seguenti passi:
 1. esamina la radice r
 2. effettua ricorsivamente la **previsita** dei sottoalberi T_1, \dots, T_k
- ▷ algoritmo di visita in ordine **differito** o **postvisita**, che consiste dei seguenti passi:
 1. effettua ricorsivamente la **postvisita** dei sottoalberi T_1, \dots, T_k
 2. esamina la radice r
- ▷ algoritmo di visita in ordine **simmetrico** o **invisita**, che consiste dei seguenti passi:
 1. fissato $i \geq 1$
 2. effettua ricorsivamente la **invisita** dei sottoalberi T_1, \dots, T_i
 3. esamina la radice r
 4. effettua la **invisita** dei sottoalberi T_{i+1}, \dots, T_k

Algoritmi sugli alberi: Previsita

```
void PREVISITA ( nodo u, albero T ) {  
  
    < esamina il nodo u >  
  
    if ( ! FOGLIA(u, T) ) {  
  
        v = PRIMOFIGLIO(u, T);  
  
        while ( ! FINEFRATELLI(v, T) ) {  
  
            PREVISITA ( v, T );  
  
            v = SUCCFRATELLO(v, T);  
  
        }  
    }  
}
```

La procedura **PREVISITA** é innescata dalla seguente chiamata:

```
PREVISITA( RADICE(T), T );
```

Algoritmi sugli alberi: Postvisita

```
void POSTVISITA ( nodo u, albero T ) {  
  
    if ( ! FOGLIA(u, T) ) {  
  
        v = PRIMOFIGLIO(u, T);  
  
        while ( ! FINEFRATELLI(v, T) ) {  
  
            POSTVISITA ( v, T );  
  
            v = SUCCFRATELLO(v, T);  
  
        }  
    }  
  
    < esamina il nodo u >  
  
}
```

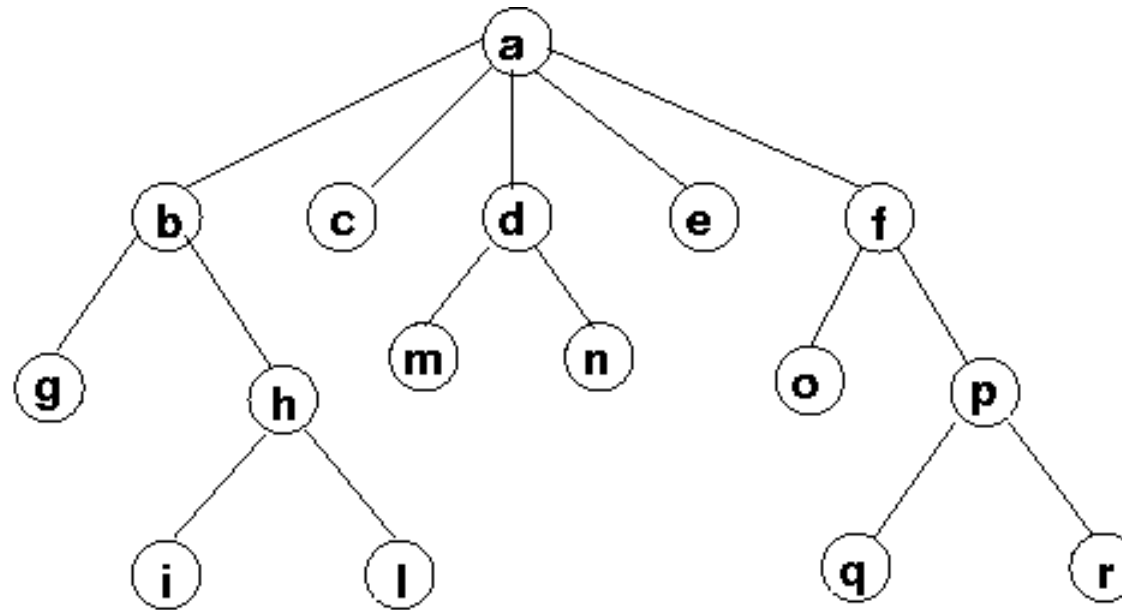
La procedura **POSTVISITA** é innescata dalla seguente chiamata:

```
POSTVISITA( RADICE(T), T );
```

Algoritmi sugli alberi: Invisita

Supponendo i , la procedura di invisita é la seguente:

```
void INVISITA ( nodo u, albero T ) {  
  
    if ( FOGLIA(u, T) ) {  
  
        < esamina il nodo u >  
  
    } else {  
  
        v = PRIMOFIGLIO(u, T);  
  
        INVISITA(v, T);  
  
        < esamina il nodo u >  
  
        v = SUCCFRATELLO(v, T);  
  
        while ( ! FINEFRATELLI(v, T) ) {  
            INVISITA(v, T);  
            v = SUCCFRATELLO(v, T);  
        }  
  
    }  
}
```



previsita(a)
 marca a con 1
 previsita(b)
 marca b con 2
 previsita(g)
 marca g con 3
 previsita(h)
 marca h con 4
 previsita(i)
 marca i con 5
 previsita(l)

postvisita(a)
 postvisita(b)
 postvisita(g)
 marca g con 1
 postvisita(h)
 postvisita(l)
 marca l con 2
 postvisita(l)
 marca l con 3
 marca h con 4
 marca b con 5
 postvisita(e)
 marca e con 6
 postvisita(d)
 postvisita(m)
 ...

INVISITA(i=1)
 invisita(a)
 invisita(b)
 invisita(g)
 marca g con 1
 marca b con 2
 invisita(n)
 invisita(i)
 marca i con 3
 marca h con 4
 invisita(l)
 marca l con 5
 marca a con 6

Algoritmi sugli alberi: Visite

Analisi di complessità:

I tre algoritmi appena descritti effettuano una chiamata ricorsiva per ogni nodo dell'albero.

Pertanto, se l'albero contiene n nodi la complessità di ciascun algoritmo di visita é $\mathcal{O}(n)$ purché si fornisca una implementazione della struttura dati **albero** per cui gli operatori utilizzati abbiano complessità $\mathcal{O}(1)$.

Osserviamo inoltre che un limite inferiore alla complessità delle procedure di visita é $\Omega(n)$ con n numero dei nodi: infatti per visitare tutti i nodi non posso impiegare meno di n passi.

Dunque sotto l'ipotesi che gli operatori hanno complessità $\mathcal{O}(1)$ possiamo concludere che la complessità degli algoritmi di visita é $\Theta(n)$ cioè gli algoritmi di visita sono **ottimi**.

Esercizio: Altezza Alberi

L'**altezza** o **profondità** di un albero é per definizione il massimo della lunghezza di tutti i cammini radice-foglia.

Scrivere una funzione **ottima** **ALTEZZA** che calcola l'altezza di un albero T di n nodi.

Per calcolare la distanza massima radice-foglia, bisogna visitare tutto l'albero in modo da misurare il numero di nodi attraversati in ogni cammino dalla radice alla foglia.

Quindi una limitazione inferiore alla complessità del problema é $\Omega(n)$.

La funzione **ALTEZZA** per essere **ottima** deve richiedere tempo $\mathcal{O}(n)$.

Per calcolare l'altezza di un albero possiamo usare uno schema di visita **post-visita** ponendo che:

- ▷ l'altezza di un nodo foglia é 0
- ▷ l'altezza di un nodo intermedio u é 1 piú l'altezza del sottoalbero radicato in (che ha come radice il nodo)
 u

Funzione Altezza:

```
int ALTEZZA ( nodo u, albero T ) {  
    nodo v;  
    int tmp, altezza;  
  
    if ( FOGLIA(u, T) )  
        altezza = 0;  
    else {  
        altezza = 0;  
        v = PRIMOFIGLIO(u, T);  
  
        while ( ! FINEFRATELLI (v, T) ) {  
            tmp = ALTEZZA(v, T);  
  
            if ( tmp > altezza )  
                altezza = tmp;  
  
            v = SUCCFRATELLO(v, T);  
        }  
  
        altezza++;  
    }  
    return altezza;  
}
```


Realizzazione di alberi: Vettore dei padri

Una semplice implementazione in C del tipo di dato albero potrebbe essere la seguente:

- ▷ supponiamo che i nodi dell'albero siano etichettati con i numeri da $0 \dots n - 1$, ove n é il numero dei nodi dell'abero, con la convenzione che:
 - ▷ ogni figlio ha numero maggiore del padre
 - ▷ ogni fratello maggiore ha un numero più piccolo del fratello minore

Un esempio di numerazione è il seguente:

- ▷ il nodo radice é etichettato con il numero 0
- ▷ il nodo padre è etichettato con un numero minore di quello dei suoi figli
- ▷ i figli di ogni nodo sono etichettati rispettivamente, dal maggiore al minore, con numeri crescenti.

In questo modo possiamo individuare il PRIMOFIGLIO di un nodo come colui che ha il numero più piccolo.

Tale sistema di etichettatura corrisponde ad una visita dell'abero secondo lo schema dell'algoritmo di **pre-visita**.

- ▷ un vettore di interi detto **vettore dei padri** contiene la seguente informazione:
 - ▷ $T[0] = -1$, il padre del nodo radice non é definito
 - ▷ $T[i] = j$, se j é il padre del nodo i

Realizzazione in C

```
typedef int      nodo;  
typedef nodo[n]  albero;
```

L'implementazione degli alberi tramite vettore dei padri permette di calcolare facilmente il padre di un nodo, per cui la funzione $\text{PADRE} \in \mathcal{O}(1)$.

Viceversa, l'operazione $\text{PRIMOFIGLIO} \in \mathcal{O}(n)$, infatti:

```
nodo PRIMOFIGLIO ( nodo u, albero T ) {  
    int found = 0;  
    int i      = 0  
  
    while ( ! found && ( i < n ) ) {  
        if ( T[i] = u )  
            found = 1;  
        else  
            i++;  
    }  
  
    return i;  
}
```

Esercizio: utilizzando l'implementazione tramite vettore dei padri, realizzare tutti gli operatori definiti per gli alberi.

Realizzazione di alberi: Liste dei figli

Una seconda implementazione degli alberi prevede l'uso di un vettore in cui ogni elemento é associato ad un nodo dell'albero, e contiene il riferimento (puntatore, cursore o altro) ad una lista contenente l'insieme dei figli del nodo.

Supponendo che i nodi siano etichettati con indici $0 \dots n - 1$, una realizzazione in C potrebbe essere la seguente:

```
typedef int nodo;
struct structalbero {
    lista testa[n];
    nodo radice;
};

typedef structalbero * albero
```

In questa implementazione é facile calcolare i figli di un nodo. Infatti, facendo riferimento all'implementazione delle liste già vista, la realizzazione della funzione PRIMOFIGLIO é immediata:

```
nodo PRIMOFIGLIO (nodo u, albero T) {
    return LEGGILISTA ( PRIMOLISTA(T->testa[u]) );
}
```

Supponendo una implementazione efficiente delle liste (eg: lista circolare) la funzione PRIMOFIGLIO ha ordine di complessità:

$$\text{PRIMOFIGLIO}() \in \mathcal{O}(1).$$

Purtroppo però tale implementazione non permette di realizzare in modo efficiente altre operazioni sugli alberi, come ad esempio la funzione PADRE.

Infatti, utilizzando la definizione del tipo di dato albero tramite liste di figli, la realizzazione della funzione PADRE(u, T) precede di cercare la lista in cui il nodo u compare, ovvero, consiste dei seguenti passi:

1. scandire tutte le liste dei figli di ciascun nodo dell'albero
2. per ogni lista L verificare se il nodo $u \in L$

Poiché ogni nodo compare una sola volta in ogni lista, nel caso pessimo bisogna esaminare tutti i figli di tutti i nodi, ovvero tutti i nodi dell'albero, quindi la complessità é $\mathcal{O}(n)$.

Le stesse considerazioni valgono per la funzione SUCCFRATELLO.

Funzione PADRE:

```
nodo PADRE ( nodo u, albero T ) {  
    int found = 0;  
    int i      = 0;  
    posizione p;  
    lista      L;  
  
    while ( ! found && ( i < n ) ) {  
        L = T->testa[i];          // lista dei figli del nodo i  
        p = PRIMOLISTA ( L );  
  
        while ( ! found && FINELISTA (p, L) )  
            if ( LEGGILISTA(p) == u )  
                found = 1;  
            else  
                p = SUCCLISTA(p);  
  
        if ( ! found )  
            i++;  
    }  
    return i;  
}
```

Esercizio realizzare tutti gli operatori degli alberi supponendo che questi ultimi siano realizzati tramite liste dei figli.

Realizzazione con puntatori padre, primofiglio, fratello

Una realizzazione che permette di implementare quasi tutti gli operatori con un ordine di complessità $\mathcal{O}(n)$, cioè in modo costante, é quella in cui ogni nodo contiene il riferimento al padre, al primo-figlio ed al fratello.

```
typedef nodestruct * albero;
typedef nodestruct * nodo;

struct nodestruct {
    tipoelem valore;
    nodo      padre;
    nodo      primofiglio;
    nodo      fratello;
}
```

Utilizzando tale realizzazione ogni operatore tranne CANCSOTTOALBERO, richiede $\mathcal{O}(1)$.

In questa implementazione CANCSOTTOALBERO non può avere costo $\mathcal{O}(1)$, poiché per aggiornare il campo fratello dei record corrispondenti ai fratelli maggiori dei nodi cancellati, occorre risalire più volte l'albero.

Per ottenere una implementazione di CANCSOTTOALBERO in tempo $\mathcal{O}(1)$, occorre che ogni nodo contenga anche un puntatore al fratello precedente (maggiore).

Comunque, anche con quest'ultima informazione in più, se si vuole effettuare una rimozione **definitiva** del sottoalbero (**free**), il tempo di eliminazione rimane comunque lineare nel numero dei nodi da eliminare.

Alberi Binari

Un albero **binario** é un particolare albero in cui ogni nodo ha al piú due figli rispettivamente **figlio sinistro** ed il **figlio destro**.

La proprietà di distinguere tra figlio sinistro e figlio destro implica che dati due alberi aventi gli stessi nodi, sono detti uguali se in entrambi ogni nodo ha gli stessi nodi figlio sinistro e destro.

Un caso particolare ed interessante degli alberi binari é rappresentato dagli **alberi binari di decisione**.

Supponiamo che un algoritmo A per risolvere un problema deve compiere una successione di scelte. La successione di scelte può essere visualizzata tramite un albero binario di decisione.

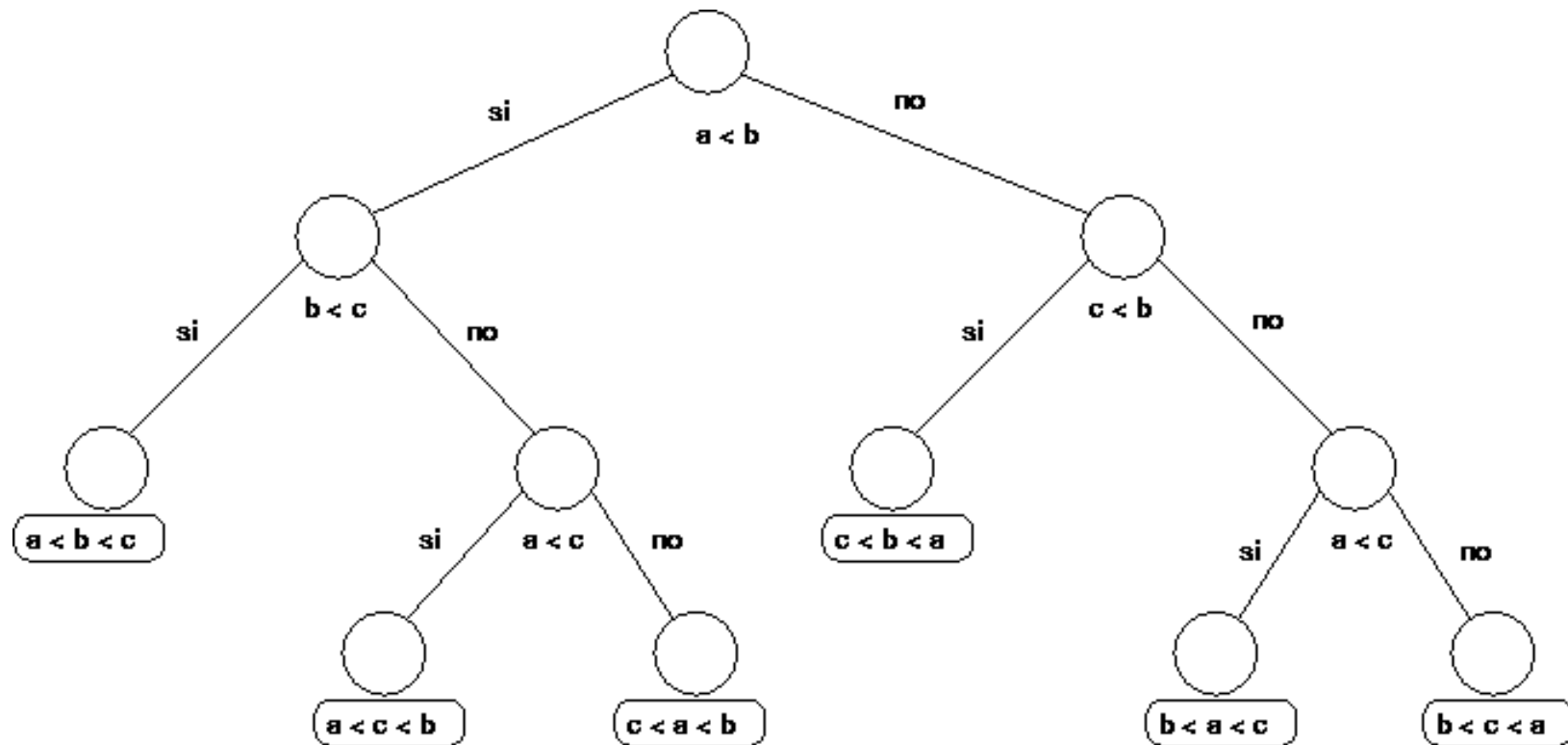
In tale albero, i nodi intermedi rappresentano confronti tra elementi del problema, ovvero scelte.

Un cammino radice foglia rappresenta la sequenza di scelte o confronti che l'algoritmo deve effettuare per costruire una soluzione corrispondente ad una particolare istanza del problema.

Il cammino di lunghezza massima rappresenta il numero di scelte da effettuare nel caso pessimo.

Albero di decisione

Esempio: albero di decisione per un il problema di ordinamento di tre numeri.



Ad esempio, dati tre numeri interi a, b, c , la sequenza di scelte $a < b$ e $b < c$ porta alla soluzione che $a < b < c$.

L'algoritmo di ordinamento di 3 numeri basato su un albero binario di decisione è il seguente:

```
void ordina3 ( int a, int b, int c ) {  
    if ( a < b )  
        if ( b < c )  
            return (a,b,c);  
        else  
            if ( a < c )  
                return (a,c,b)  
            else  
                return (c,a,b)  
    else  
        if ( c < b )  
            return (c,b,a)  
        else  
            if ( a < c )  
                return (b,a,c)  
            else  
                return (b,c,a)  
}
```

L'algoritmo, dati 3 numeri, genera tutte le possibili **permutazioni** dei 3 numeri, corrispondenti a possibili soluzioni di una istanza del problema.

In generale, se indichiamo con n il numero dei dati da ordinare, la complessità di un algoritmo basato su un albero decisionale è $\mathcal{O}(n!)$, ovvero **ordine fattoriale**.

L'algoritmo di ordinamento deve individuare, tramite una sequenza di scelte, la permutazione dei dati corrispondente a quella in cui gli elementi sono ordinati nel modo voluto.

Ogni nodo non terminale dell'albero di decisione rappresenta una scelta, mentre ogni foglia rappresenta una possibile permutazione dei dati di ingresso.

Se, data una istanza di dimensione n , l'algoritmo compie, nel caso pessimo, h scelte, allora:

- ▷ il numero delle foglie dell'albero sarà $n_f = 2^h$
- ▷ il numero dei nodi non foglia sarà $2^h - 1$
- ▷ il numero totale dei nodi dell'albero sarà $N = 2^{h+1} - 1$

Infatti, possiamo dimostrare per induzione che un albero di altezza h contiene $k = 2^h$ foglie:

- ▷ $h = 0$, $n_f = 1 = 2^0 = 2^h$
- ▷ supponiamo (**ipotesi induttiva**) che la relazione sia vera per $h - 1$ e dimostriamo che é vera per h :
passando dal livello $h - 1$ al livello h il numero delle foglie al più raddoppia, quindi $n_f = 2 \cdot 2^{h-1} = 2^h$

Il numero totale N dei nodi dell'albero é dato dalla somma di tutti i nodi presenti ad ogni livello $i = 0 \dots h$, quindi:

$$N = \sum_{i=0}^h 2^i = 1 + 2 \frac{1-2^h}{1-2} = 2^{h+1} - 1$$

Il numero dei nodi intermedi possiamo calcolarlo come differenza di $N - n_f = 2^{h+1} - 1 - 2^h = 2^h - 1$.

Data una istanza del problema di ordinamento di dimensione n , il numero delle possibili soluzioni ad essa corrispondente é $n!$, cioè il numero delle possibili permutazioni di n dati.

Se ad ogni passo l'algoritmo effettua una scelta binaria, allora la sequenza di scelte può essere rappresentata su un albero binario di decisione avente $n!$ foglie ed una altezza (distanza radice-foglia) pari a $h = \log_2 n!$

Quindi nel caso pessimo la soluzione può essere determinata effettuando $h = \log_2 n!$ scelte o confronti.

Poiché valgono le seguenti maggiorazioni:

$$n! \geq n \cdot (n-1) \cdot \dots \cdot (n/2) \geq (n/2)^{n/2}$$

si deduce che:

$$h = \log_2 n! \geq (n/2) \cdot \log_2(n/2)$$

Pertanto una **limitazione inferiore** alla complessità del problema dell'ordinamento, intesa come numero di confronti in funzione di n , é $\Omega(n \log n)$.

Se riusciamo a determinare un algoritmo di complessità $\mathcal{O}(n \log n)$ abbiamo determinato un algoritmo di ordinamento **ottimo**.

Alberi Binari: Specifica

La specifica degli alberi binari include gli operatori visti per gli alberi ordinari piú i seguenti operatori:

- ▷ FIGLIOSINISTRO: (nodo, binalbero) \longrightarrow nodo
FIGLIOSINISTRO (u , T) = v
Pre: $T \neq \Lambda$, $u \in T$ u ha un figlio sinistro
Post: v é il figlio sinistro di u
- ▷ FIGLIODESTRO: (nodo, binalbero) \longrightarrow nodo
FIGLIODESTRO (u , T) = v
Pre: $T \neq \Lambda$, $u \in T$ u ha un figlio sinistro
Post: v é il figlio destro di u
- ▷ SINISTROVUOTO(nodo, binalbero) \longrightarrow boolean
SINISTROVUOTO(u , T) = b
Pre: $T \neq \Lambda$, $u \in T$
Post: $b = \text{True}$ se u non ha figlio sinistro, $b = \text{False}$ altrimenti
- ▷ DESTROVUOTO(nodo, binalbero) \longrightarrow boolean
DESTROVUOTO(u , T) = b
Pre: $T \neq \Lambda$, $u \in T$
Post: $b = \text{True}$ se u non ha figlio destro, $b = \text{False}$ altrimenti

▷ COSTRBINALBERO: (binalabero, binalbero) \longrightarrow binalbero

$\text{COSTRBINALBERO}(T, U) = T'$

Pre:

Post: T' é un albero binario con radice un nuovo nodo u il cui figlio sinistro é l'albero T ed il figlio destro é l'albero U . Se uno dei due tra T ed U é vuoto, allora il corrispondente figlio di u non esiste.

Alberi Binari: Algoritmi di visita

Utilizzando gli operatori appena definiti, gli algoritmi di visita, **pre-visita**, **in-visita** e **post-visita** degli alberi binari possono essere definiti nel seguente modo:

```
void BINVISITA ( nodo u, binalbero T ) {  
  
    < esame anticipato del nodo u > // pre-visita  
  
    if ( ! SINISTROVUOTO(u, T)  
        BINVISITA ( FIGLIOSINISTRO(u, T), T);  
  
    < esame simmetrico del nodo u > // in-visita  
  
    if ( ! DESTROVUOTO(u, T)  
        BINVISITA ( FIGLIODESTRO(u, T), T);  
  
    < esame differito del nodo u > // post-visita  
  
}
```

Lo schema sopra raccoglie tutti e tre gli algoritmi di visita.

Alberi Binari: Realizzazione

```
typedef bincella * binalbero;  
typedef bincella * nodo;  
  
struct bincella {  
    tipoelem: valore;  
    nodo sinistro, destro, genitore;  
};
```

```
binalbero CREABINALBERO () {  
    binalbero T = NULL;  
    return T;  
}
```

```
int BINALBEROVUOTO(binalbero T) {  
    return ( T == NULL ) ? 1 : 0;  
}
```

```
nodo BINRADICE ( binalbero T ) {  
    return T;  
}
```

```
nodo FIGLIOSINISTRO ( nodo u ) {  
    return u->sinistro;  
}
```

```
nodo FIGLIODESTRO ( nodo u ) {  
    return u->destro;  
}
```

```
int SINISTROVUOTO ( nodo u ) {  
    return ( u->sinistro == NULL ) ? 1 : 0;  
}
```

```
int DESTROVUOTO ( nodo u ) {  
    return ( u->destro == NULL ) ? 1 : 0;  
}
```


Procedura per la costruzione di binalberi:

```
// CONSTRBINALBERO: costruisce un albero binario a partire da due alberi binari T ed U.  
// L'albero binario T e' passato per variabile poiche' T viene aggiornato con  
// il riferimento alla radice del nuovo albero  
void CONSTRBINALBERO ( binalbero * T, binalbero U ) {  
    nodo u;  
  
    u = (nodo) malloc ( sizeof(bincella) );  
  
    u->sinistro = * T;  
    u->destro   = U;  
    u->genitore = NULL;  
  
    if ( *T != NULL )  
        T->genitore = u;  
  
    if ( U != NULL )  
        U->genitore = u;  
  
    *T = u; // modifica l'albero T  
}
```

La procedura **CONSTRBINALBERO** costruisce alberi binari:

- ▶ **CONSTRBINALBERO(T, U), T = NULL, U = NULL**: costruisce un albero composto da un solo nodo ed il riferimento a tale nodo é assegnato a T
- ▶ **CONSTRBINALBERO(T, U), T != NULL, U = NULL**: costruisce un albero binario con un nodo il cui sottoalbero sinistro é l'albero T.
- ▶ **CONSTRBINALBERO(T, U), T != NULL U != NULL**: costruisce un albero binario con un nodo il cui sottoalbero sinistro é l'albero T ed il sottoalbero destro é U.

Ad esempio il seguente frammento di codice C costruisce un albero binario di tre nodi:

```
CONSTRBINALBERO ( &T, NULL ); // crea un nodo con valore 100
n = BINRADICE(T);
SCRIVINODO(100, n);

CONSTRBINALBERO ( &U, NULL ); // crea un nodo con valore 101
n = BINRADICE(U);
SCRIVINODO(101, n);

CONSTRBINALBERO ( &T, U );      // crea un nodo con valore 102
n = BINRADICE(T);
SCRIVINODO(102, n);
```

Procedura Ricorsiva per la Creazione di Alberi Binari

Supponiamo di voler costruire un albero binario di altezza h ed $n = 2^{h+1} - 1$ nodi.

La costruzione dell'albero può essere effettuata per valori di h decrescenti secondo il seguente algoritmo:

1. se $h = 0$ crea un solo nodo
2. se $h > 0$ crea **ricorsivamente** due sottoalberi di altezza $h - 1$ e poi **uniscili** in un unico albero binario

```
void INITBINALBERO( int h, binalbero * T ) { // h = altezza
    binalbero U = CREABINALBERO();

    if ( h == 0 )                // crea un nodo
        CONSTRBINALBERO ( T, NULL );
    else {                        // inizializza due sottoalberi T ed U di altezza h-1
        INITBINALBERO ( h-1, T );
        INITBINALBERO ( h-1, &U );
        CONSTRBINALBERO ( T, U ); // unisci i due sotto-alberi
    }
}
```

L'albero T é passato per variabile in modo tale che alla fine della procedura T sia aggiornato con il valore del riferimento al nuovo albero.

Rappresentazione Binaria di Alberi Ordinati

Un qualsiasi albero ordinato può essere rappresentato tramite un albero binario avente gli stessi nodi e la stessa radice.

L'algoritmo di trasformazione di un albero ordinato T in un albero binario B è il seguente:

1. per ogni nodo n di T , inserisci n in B
2. dichiara come figlio sinistro di n il primo figlio che n ha in T
3. dichiara come figlio destro di n il fratello successivo che n ha in T

La sequenza di nodi esaminati di T e B sarà la stessa se:

- ▶ T e B sono visitati in ordine anticipato (pre-visita o pre-order)
- ▶ T è visitato in ordine differito (post-visita o post-order) e B in ordine simmetrico (invisita o in-order)

Esercizio: scrivere l'algoritmo di trasformazione da un albero ordinato ad un albero binario.

Alberi Binari di Ricerca

Un caso particolarmente interessante, per l'uso in campo informatico, é rappresentato dagli **alberi binari di ricerca**.

Possiamo definire gli alberi binati di ricerca come alberi binari in cui é possibile definire una relazione di ordinamento tra gli elementi contenuti nei nodi dell'albero.

Esempio: un albero binario i cui nodi contengono numeri interi é un allbero binario di ricerca se:

per ogni nodo dell'albero il valore in esso contenuto é:

- ▷ **maggiore o uguale** (**minore o uguale**) dei valori dei nodi contenuti nel sottoalbero sinistro
- ▷ **minore o uguale** (**maggiore o uguale**) dei valori dei nodi del sottoalbero destro.

Gli alberi binari di ricerca sono interessanti poiché la relazione di ordinamento che occorre tra i nodi dell'albero permette di **guidare** la ricerca di un particolare elemento.

Un algoritmo che usa un albero binario di ricerca, nel caso pessimo ha complessità $\mathcal{O}(h) \sim \leq \mathcal{O}(\log_2 n)$, ove n è il numero complessivo dei nodi dell'albero, ovvero, ha complessità **logaritmica**.

Esercizio: procedura ricorsiva di creazione ed inizializzazione di un albero binario di ricerca

```

/*****
/* procedura di inizializzazione di un albero binario di ricerca.
/* Ogni nodo ha un valore:
/*     maggiore di tutti i nodi del sottoalbero sinistro
/*     minore   di tutti i nodi del sottoalbero destro
/* l = numero di livelli dell'albero, e = valore iniziale con cui etichettare i nodi*/
*****/
int INITBINALBERO_RICERCA ( int l, int e, binalbero * T ) {
    binalbero U = CREABINALBERO();
    int a;

    if ( l == 1 ) {                // crea una foglia
        CONSTRBINALBERO ( T, NULL );
        SCRIVINODO ( e, BINRADICE(*T) );
        e++;
    } else {                       // inizializza due sottoalberi
        e = INITBINALBERO_RICERCA(l-1, e, T );
        a = e;                     // memorizzo il valore da assegnare alla radice
        e = INITBINALBERO_RICERCA(l-1, e+1, &U);
        CONSTRBINALBERO ( T, U ); // unisci i due sotto-alberi
        SCRIVINODO ( a, BINRADICE(*T) );
    }
    return e;
}

```

Esercizio

Scrivere una procedura C **ottima** che, dato un albero ordinato i cui nodi contengono valori interi, si vogliono cancellare tutte le foglie per le quali la somma dei valori contenuti nei nodi del cammino radice foglia abbia lunghezza k .

Soluzione: Sia n il numero dei nodi dell'albero ordinato.

Per realizzare l'operazione proposta bisogna percorrere tutti i cammini radice foglia esaminando di conseguenza tutti i nodi dell'albero.

Quindi una limitazione inferiore alla complessità del problema é $\Omega(n)$.

L'algoritmo deve effettuare una visita dei nodi dell'albero, e per ogni nodo u calcolare una valore **sum** uguale al somma del valore del nodo u e la somma dei valori dei nodi incontrati nel cammino dalla radice al nodo u .

- ▷ $sum_0 = \text{LEGGINODO}(\text{RADICE}(T))$
- ▷ $sum_1 = \text{LEGGINODO}(\text{PRIMOFIGLIO}(\text{RADICE}(T))) + sum_0$
- ▷ ...

Poiché per ogni nodo u bisogna prima calcolare il valore di **sum** e poi eventualmente cancellare la foglia, l'algoritmo effettuerà una visita **anticipata**.

```
void CANCELLAFOGLIEK ( nodo u, albero T, int k, int somma ) {  
  
    if ( ! FOGLIA(u, T) ) {  
  
        v = PRIMOFIGLIO(u, T);  
  
        while ( ! FINEFRATELLI (v, T) ) {  
            CANCELLAFOGLIEK ( v, T, k, somma + LEGGINODO(u, T), k );  
            v = SUCCFRATELLO(v, T);  
        }  
  
    } else {  
  
        if ( somma + LEGGINODO(u, T) == k )  
            CANCSOTTOALBERO(u, T);  
  
    }  
}
```

Se assumiamo una implementazione efficiente degli alberi, ad esempio puntatore padre, primo-figlio, fratello, allora la procedura, essendo basata su una procedura di visita ha complessità $\mathcal{O}(n)$, ove n è il numero dei nodi. Poiché la complessità del caso pessimo è uguale alla limitazione inferiore, la procedura è di complessità **ottima**.

che succede se valutiamo $\Omega(\dots)$ per eccesso ?

Esercizio

Dato un albero binario B si vuole memorizzare in ogni nodo v il numero dei nodi del sottoalbero radicato in v . Scrivere una procedura C ricorsiva di complessità **ottima**.

Soluzione: per ogni nodo v bisogna visitare tutti i nodi del sottoalbero radicato in v , quindi in definitiva bisogna visitare tutti i nodi dell'albero. Perciò una limitazione inferiore alla complessità del problema è $\Omega(n)$ ove n è il numero dei nodi dell'albero.

L'algoritmo effettuerà una visita dell'albero:

- ▷ se il nodo visitato è una foglia allora si scrive 1 nel campo valore del nodo
- ▷ altrimenti si scrive 1 più la somma dei valori memorizzati rispettivamente nel figlio sinistro e nel figlio destro

Poiché il valore da scrivere in ogni nodo è uguale al valore del nodo più il valore del figlio sinistro e del figlio destro, il tipo di visita da utilizzare è la visita **differita** o **posticipata**.

```
void CONTANODI ( nodo u, binalbero T ) {  
    int s = 0;  
    int d = 0;  
  
    if ( FOGLIA(u) )  
        SCRIVINODO(1, u);  
    else {  
        if ( ! SINISTROVUOTO(u) ) {  
            CONTANODI( FIGLIOSINISTRO(u), T );  
            s = LEGGINODO(FIGLIOSINISTRO(u), T);  
        }  
        if ( ! DESTROVUOTO(u) ) {  
            CONTANODI( FIGLIODESTRO(u), T );  
            d = LEGGINODO(FIGLIODESTRO(u), T);  
        }  
        SCRIVINODO( 1+s+d, u );  
    }  
}
```

Poiché l'algoritmo é basato su un schema di visita, la sua complessità é $\mathcal{O}(n)$ e quindi **ottimo**.

La procedura viene innescata dalla chiamata **CONTANODI(BINRADICE(B), T)**.

Esercizio

Siano dati un albero binario B , non vuoto, di n nodi contenenti valori interi, ed una lista L contenente $k \in \mathcal{O}(n)$ valori interi.

Scrivere in C una procedura ricorsiva \mathcal{P} che costruisca una lista M i cui elementi sono tutti gli elementi dell'albero B che non sono presenti nella lista L .

Studiare la complessità della procedura della procedura \mathcal{P} .

Dire qual'è la realizzazione delle strutture dati e la semantica degli operatori utilizzati per definire la procedura \mathcal{P} .

Soluzione:

La procedura \mathcal{P} deve:

1. visitare l'albero B utilizzando un qualsiasi schema di visita
2. per ogni nodo v dell'albero B controllare se il valore in esso contenuto appartiene alla lista L
3. se non appartiene alla lista L inserire il valore del nodo v nella lista M

Assumendo una rappresentazione degli alberi binari tramite puntatori padre, figlio sinistro e destro, ed una rappresentazione delle lista tramite puntatori, possiamo fare le seguenti considerazioni di costo:

- ▷ il passo 1 ha un costo lineare nel numero dei nodi dell'albero
- ▷ il passo 2 ha un costo lineare nel numero dei dati contenuti nella lista L
- ▷ il passo 3 ha un costo costante

Poiché il passo 2 é eseguito per ogni nodo visitato e poiché si assume che il numero degli elementi della lista L sia ordine n , possiamo concludere che la procedura $\mathcal{P} \in \mathcal{O}(n^2)$.

```
void P ( nodo v, lista L, lista M ) {  
  
    if ( ! APPARTIENELISTA(LEGGINODO(v), L ) ) {  
        posizione p = PRIMOLISTA(M);  
        INSLISTA(LEGGINODO(v), &p);  
    }  
  
    if ( ! SINISTROVUOTO(v) )  
        P ( FIFLIOSINISTRO(v), L, M );  
  
    if ( ! DESTROVUOTO(v) )  
        P ( FIFLIODESTRO(v), L, M );  
  
}
```

La procedura viene innescata dalla chiamata $P(\text{BINRADICE}(B), L, M)$.

Esercizio

Sia dato un albero binario i cui nodi contengono valori interi. Supponendo che il tipo di dato albero binario sia realizzato mediante puntatori padre figlio sinistro e destro, scrivere in C una procedura ricorsiva **ottima**, **aggiunginodo**, che **estenda** l'implementazione del tipo di dato albero binario.

La procedura **aggiunginodo** deve aggiungere ad ogni foglia v un nodo contenente come valore la somma di tutti i valori contenuti nel cammino dalla radice alla foglia v .

Scrivere un **main** che:

1. inizializza l'albero binario con valori random
2. stampa l'albero (visita l'albero e stampa i valori contenuti in ogni nodo)
3. chiama la procedura **aggiunginodo**
4. ristampare l'albero

La procedura **aggiunginodo** può assumere di conoscere la realizzazione C dell'albero binario.

Studiare la complessità della procedura **aggiunginodo**.