Alberi binari

Un albero binario è un albero con radice in cui ogni nodo ha al massimo due figli, chiamati figlio sinistro e figlio destro.

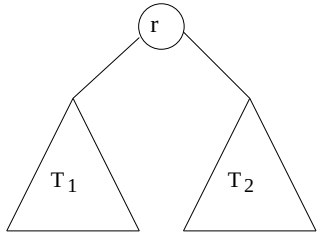
Un albero binario T i cui nodi appartengono a un insieme V può essere definito induttivamente come segue:

Definizione

T è un albero binario se e solo se:

- T coincide con $\emptyset$ (= albero *vuoto*).
- $T = \langle r, T_1, T_2 \rangle$, dove $r \in V$, T_1 e T_2 sono alberi binari.

Nel secondo caso, r è la radice di T, T_1 e T_2 sono rispettivamente il sottoalbero sinistro e il sottoalbero destro di r.



Gli elementi di V che compaiono in T sono chiamati nodi (o vertici) di T.

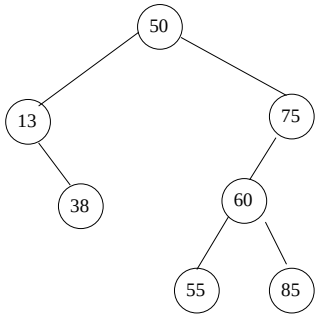
Un nodo è detto foglia se non ha figli, altrimenti è chiamato nodo interno.

Esempio 1

Esempio di albero binario i cui nodi appartengono all'insieme dei numeri naturali:

$$\bar{T} = \langle 50\ ,\ \langle 13, \emptyset, \langle 38, \emptyset, \emptyset \rangle \rangle\ ,\ \langle 75,\ \langle 60, \langle 55, \emptyset, \emptyset \rangle, \langle 85, \emptyset, \emptyset \rangle \rangle,\ \emptyset \rangle \rangle$$

$\bar{T}$ può essere rappresentato come un *grafo non orientato aciclico*:



Dato un albero T e un nodo v di T:

- L'altezza di v è la lunghezza del più lungo cammino da v a una foglia di T.
- La profondità (o livello) di v è la lunghezza del cammino (che, per definizione, è unico) da v alla radice di T.

L'altezza di T è l'altezza della sua radice (equivalentemente, la massima profondità di una foglia).

Nell'esempio precedente:

- 50 ha altezza 3 e profondità 0;
- 75 ha altezza 2 e profondità 1;
- 38 ha altezza 0 e profondità 2;
- 55 ha altezza 0 e profondità 3.

L'albero ha altezza 3.

Esercizi

1. Verificare che l'altezza $height(T)$ di un albero binario T può essere definita induttivamente come segue:

$$height(T) = \begin{cases} -1 & \text{se } T = \emptyset; \\ \max(height(T_1), height(T_2)) + 1 & \text{se } T = \langle r, T_1, T_2 \rangle \end{cases}$$

2. Definire induttivamente la funzione $belongs(v, T)$ che vale 1 se v è un nodo dell'albero T, 0 altrimenti. Usare uno schema induttivo simile a quello dell'es. 1.

3. Definire induttivamente la funzione $subtree(v, T)$ che determina il sottoalbero T_v di T avente come radice v (se v non appartiene a T, T_v è l'albero vuoto). Usare uno schema induttivo simile a quello dell'es. 1.

4. Definire induttivamente la funzione $depth(v, T)$ che calcola la profondità del nodo v nell'albero T (se v non appartiene a T, $depth(v, T) = -1$). Usare uno schema induttivo simile a quello dell'es. 1.

5. Definire induttivamente la funzione $sum(T)$ che calcola la somma dei nodi dell'albero T. Usare uno schema induttivo simile a quello dell'es. 1.

Rappresentazione di un albero binario

Per rappresentare un nodo di un albero binario i cui nodi sono interi si può usare la seguente struttura ricorsiva:

```
struct node {
    int key; // valore del nodo
    struct node* left;  // radice del sottoalbero sinistro
    struct node* right; // radice del sottoalbero destro
};

typedef struct node node;
```

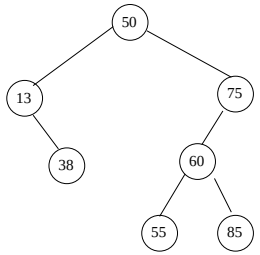
- Il campo `left` contiene l'indirizzo della radice del sottoalbero sinistro del nodo (`NULL` se il sottoalbero è vuoto).
- Analogamente, `right` è la radice del sottoalbero destro.

Per rappresentare un albero occorre definire una variabile `root` di tipo `node*` (*radice* dell'albero).

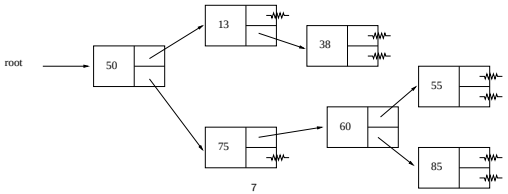
- Se `root` vale `NULL`, `root` rappresenta l'albero vuoto.
- Se `root` è diverso da `NULL`, `root` è l'indirizzo al nodo radice dell'albero.

Esempio 2

L'albero



viene rappresentato come



Per costruire l'albero della figura precedente si possono usare le seguenti istruzioni:

```
node* root; //  radice dell'albero
root = malloc(sizeof(node));
root->key = 50;
root->left = malloc(sizeof(node));
root->right = malloc(sizeof(node));
root->left->key = 13;
root->left->left = NULL;
root->left->right = malloc(sizeof(node));
root->right->key = 75;
root->right->left = malloc(sizeof(node));
root->right->right = NULL;
root->left->right->key = 38;
  .....
```

Esercizi

Per risolvere gli esercizi 1,2 e 3 occorre aver svolto gli esercizi a pag. 5.

1. Scrivere il codice di una funzione ricorsiva di intestazione

`int height(node *root)`

che calcola l'altezza dell'albero di radice `root`.

2. Scrivere il codice di una funzione ricorsiva di intestazione

`int belongs(node *v, node *root)`

che restituisce 1 se il nodo (il cui indirizzo è) `v` appartiene all'albero di radice `root`, 0 altrimenti.

3. Scrivere il codice di una funzione ricorsiva di intestazione

`int depth(node *v, node *root)`

che restituisce la profondità del nodo `v` nell'albero di radice `root`, -1 se `v` non appartiene all'albero.

4. Scrivere il codice di una funzione ricorsiva di intestazione

`int height(node *root)`

che calcola la somma dei nodi dell'albero di radice `root`.

Alberi binari di ricerca

Permettono di rappresentare in modo *efficiente* insiemi dinamici su cui sono richieste frequenti operazioni di ricerca.

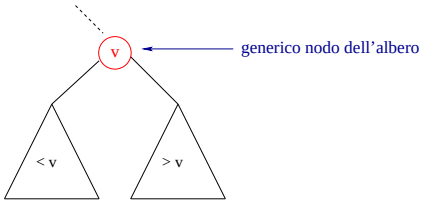
Definizione

Sia T un albero binario i cui nodi sono elementi di un insieme V su cui è definito una *relazione d'ordine lineare* $<$.

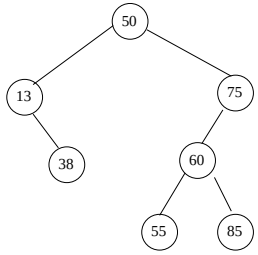
T è un albero binario di ricerca se e solo se, per ogni coppia di nodi u e v di T, vale:

- se u appartiene al sottoalbero sinistro di v, allora $u < v$
- se u appartiene al sottoalbero destro di v, allora $v < u$.

Non è considerato il caso di nodi uguali perché per rappresentare insiemi è sufficiente rappresentare ogni elemento una sola volta.



L'albero qui sotto non è una albero di ricerca. Perché?



Relazione d'ordine lineare

Sia V un insieme e sia $<$ un relazione binaria $<$ su V (ossia, $<$ è un sottoinsieme di $V \times V$).

- $<$ è una relazione d'ordine (stretto) su V se e solo se:

  (1) Per ogni $x, y \in V$, $x < y$ implica $y \not< x$
  (cioè, non vale $y < x$).

  (2) Per ogni $x, y, z \in V$, $x < y$ e $y < z$ implica $x < z$.
  (*transitività* di $<$).

  Da 1 segue che, per ogni $x \in V$, $x \not< x$.

- $<$ è una relazione d'ordine (stretto) lineare su V se e solo se:

  (1) $<$ è una *relazione d'ordine (stretto)* su V

  (2) Per ogni $x, y \in V$, $x < y$ oppure $y < x$ oppure $x = y$.

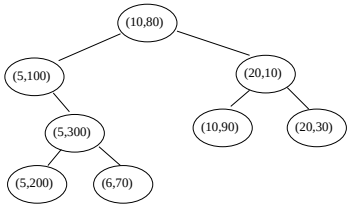
  Dalla definizione di relazione d'ordine, si deduce che *una sola* delle tre alternative è verificata.

Esempi

1. Supponiamo che V sia un insieme di numeri (naturali, interi, reali). Allora l'usuale relazione di ordinamento $<$ è un ordine lineare.

2. Sia V l'insieme dei punti nello spazio (coppie di interi).
Si può usare l'ordinamento fra coppie definito come segue:

$$(x_1, y_1) < (x_2, y_2) \iff (x_1 < x_2) \text{ oppure } (x_1 = x_2 \text{ e } y_1 < y_2)$$



L'albero di ricerca della figura rappresenta l'insieme contenente i punti (elencati in ordine crescente)

$$(5,100),\ (5,200),\ (5,300),\ (6,70),\ (10,80),\ (10,90),\ (20,10),\ (20,30)$$

3. Dato un alfabeto Σ su cui è definita una relazione d'ordine lineare $<_\Sigma$ (ad esempio, Σ è l'alfabeto $\{a, b, c, \ldots z\}$ e $<_\Sigma$ è definita dall'ordine in cui le lettere sono elencate, quindi $a <_\Sigma b <_\Sigma c <_\Sigma \cdots <_\Sigma z$), sia V l'insieme Σ^* delle parole costruite su Σ.

Date due parole x e y di Σ^*, $x < y$ (*ordinamento lessicografico*) se e solo se si verifica una delle seguenti condizioni:

- x è *prefisso proprio* di y.

  Questo significa che y è ottenibile concatenando a x una parola z diversa dalla parola nulla ϵ. La restrizione su z serve ad evitare che una parola sia prefisso proprio di se stessa.
  Ad esempio, `cane` è prefisso proprio di `canestro`, mentre `cane` non è prefisso proprio di `cane`.

- Le parole x e y hanno un *prefisso comune* w e, detta a la lettera in x subito dopo w e b la lettera in y subito dopo w, si ha che $a <_\Sigma b$.

  Se Σ è l'alfabeto italiano, `candela` $<$ `cane` (il prefisso w comune è `can` e `d` $<_\Sigma$ `e`).

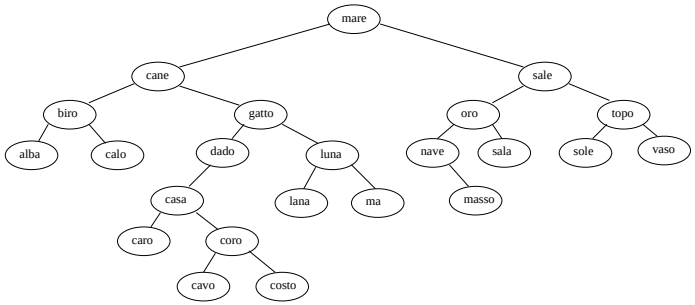
Formalmente:

$$x < y$$
$$\iff$$
$$(y = xz) \text{ oppure } (x = waz_1 \text{ e } x = wbz_2 \text{ e } a <_\Sigma b)$$

dove $w, z_1, z_2 \in \Sigma^*$, $a, b \in \Sigma$, $z \in \Sigma^+$ (insieme delle parole sull'alfabeto Σ diverse da ϵ).

L'ordinamento lessicografico è una relazione d'ordine lineare su Σ^*.

Esempio di albero binario di ricerca contenente parole definite sull'alfabeto $\{a, b, c, \ldots, z\}$



Nota

Quando si usano gli alberi binari di ricerca, sui nodi deve essere definita una relazione che soddisfa *tutte* le proprietà della definizione di ordine lineare.

Ad esempio, la relazione $<$ fra punti definita da

$$(x_1, y_1) < (x_2, y_2) \iff (x_1 < x_2) \text{ oppure } (y_1 < y_2)$$

non è una relazione d'ordine lineare. Infatti, secondo tale definizione di ha

$$(1, 10) \ < \ (10, 5) \qquad (10, 5) \ < \ (1, 10)$$

che contraddice la proprietà 1 della definizione di ordine stretto.

Esercizi

1. Dire quali fra le seguenti relazioni fra punti sono ordini lineari. In caso di risposta affermativa dimostrare che le proprietà della definizione valgono, altrimenti dare dei controesempi che mostrino quale proprietà è falsificata.

$$\begin{aligned}
(a, b) < (c, d) &\iff a < c \\
(a, b) < (c, d) &\iff a < c \text{ e } b < d \\
(a, b) < (c, d) &\iff (a + b) < (c + d) \\
(a, b) < (c, d) &\iff (a + b) < (c + d) \text{ e } b < d \\
(a, b) < (c, d) &\iff a \text{ è multiplo di } c \\
(a, b) < (c, d) &\iff a < d \text{ oppure } (a = d \text{ e } b < c)
\end{aligned}$$

2. Definire una relazione d'ordine stretto fra n-ple $(a_1, \ldots, a_n)$ di interi che estenda l'ordinamento fra coppie di interi definito prima.

Rappresentazione di insiemi dinamici mediante alberi binari di ricerca

Gli alberi binari di ricerca permettono di usare strategie *efficienti* per le operazioni di:

- Inserimento di un elemento
- Ricerca di un elemento
- Ricerca del minimo e del massimo
- Cancellazione di un elemento

La complessità di queste operazioni è proporzionale all'altezza dell'albero che rappresenta l'insieme.

Altezza di un albero binario di ricerca

L'altezza h dell'albero binario di ricerca che rappresenta un insieme I di n elementi dipende dall'ordine in cui gli elementi di I sono inseriti.

Vale:

$$\lfloor \log_2 n \rfloor \ \leq \ h \ \leq \ n - 1$$

- Se ogni nodo interno ha 2 figli, allora $h = \lfloor \log_2 n \rfloor$;
- se invece ogni nodo ha un solo figlio, allora $h = n - 1$.

Nota

L'intervallo entro cui può variare l'altezza è molto ampio (si tenga presente che $n = 2^{\log_2 n}$).

Si dimostra che, se l'inserimento avviene in modo casuale, l'altezza si mantiene vicino al valore minimo (teorema enunciato più avanti).

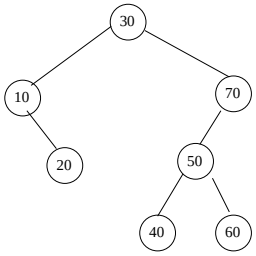
Esempio

Si vuole rappresentare mediante un albero di ricerca l'insieme $I = \{10, 20, 30, 40, 50, 60, 70\}$ contenente 7 elementi.

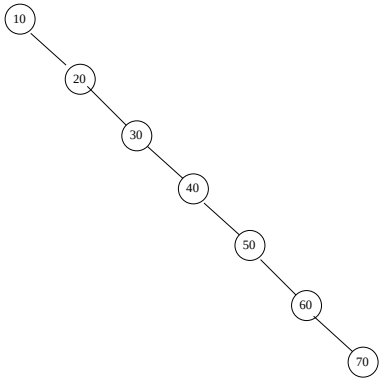
Se l'ordine con cui gli elementi sono inseriti è

30 10 70 20 50 40 60

l'insieme è rappresentato dall'albero di altezza 3 in figura.



Se invece gli elementi sono inseriti in ordine crescente si ottiene un albero di altezza 6 (lista).



Esercizio

Quali sequenze generano alberi di ricerca di altezza 2 (minima altezza richiesta per l'insieme I)?

Sequenze che generano alberi di altezza massima

Supponiamo che I contenga n elementi distinti $k_1, \ldots, k_n$ e di costruire un albero binario di ricerca inserendo gli elementi uno alla volta, partendo dall'albero vuoto. Le possibili sequenze di inserimento sono date dalle permuatazioni di I, quindi sono $n!$.

Si osserva che:

1. Diverse permutazioni possono generare lo stesso albero (il primo albero dell'esempio precedente è generato anche dalla sequenza $30, 70, 10, 50, 60, 40, 20$).
2. Le permutazioni che generano l'albero di altezza massima $n-1$ sono 2^{n-1} (vedi discussione nel prossimo lucido).

Poiché

$$\lim_{n\to\infty} \frac{2^{n-1}}{n!} = 0$$

si può affermare che le sequenze che producono alberi degeneri sono "poche" rispetto alle possibili $n!$ sequenze di input.

Quindi, se le sequenze di input sono equiprobabili, la probabilità di ottenere un albero degenere è bassa.

Dimostrazione del punto 2

Basta osservare che la sequenza

$$k_1\ ,\ \ldots\ ,\ k_n$$

genera un albero di altezza $n-1$ (altezza massima) se e solo se, per ogni $1 \le j \le n-1$, vale:

$$(k_j = \min\{k_j, \ldots, k_n\}) \quad \text{oppure} \quad (k_j = \max\{k_j, \ldots, k_n\})$$

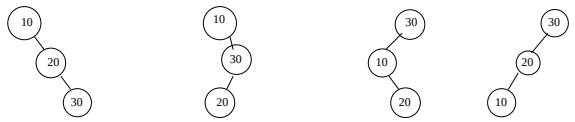
cioè, k_j è il minimo o il massimo fra gli elementi ancora da inserire.

Quindi, per ogni $1 \le j \le n-1$, l'elemento j-esimo della sequenza di input può essere scelto in 2 modi, e questo dà origine complessivamente a 2^{n-1} possibili sequenze che producono alberi di altezza massima.

Esempio

Se $I = \{10, 20, 30\}$, le sequenze $S_1, \ldots, S_4$ che generano alberi di altezza massima 2 sono:

$S_1 : 10, 20, 30 \quad S_2 : 10, 30, 20, \quad S_3 : 30, 10, 20 \quad S_4 : 30, 20, 10$



Sul libro (Capitolo 13) è dimostrato il seguente importante risultato che afferma che mediamente l'altezza di un albero generato casualmente è prossima al valore minimo:

Teorema

L'altezza media di un albero di ricerca costruito in modo casuale contenente n nodi è $O(\log n)$.

Conclusioni

In un albero di ricerca contenente n nodi:

- La ricerca ha complessità $O(h)$, dove h è l'*altezza* dell'albero.
- L'altezza h dell'albero dipende dalla sequenza di inserimento dei nodi dell'albero.

  Nel "caso medio" $h = O(\log n)$, quindi la ricerca è $O(\log n)$.

Se le sequenze che generano alberi degeneri (o, più in generale, alberi di altezza prossima a n) hanno probabilità alta occorre usare alberi binari di ricerca bilanciati (esempio, *RB-alberi*), in cui è garantito che l'altezza è $O(\log n)$ indipendentemente dal modo in cui i nodi sono inseriti.

Rappresentazione di un albero binario di ricerca

Per semplificare l'operazione di cancellazione, la struttura `struct node` utilizzata per rappresentare un nodo contiene anche l'indirizzo del nodo padre (vedi file `tree.h` e `tree.c`).

```
struct node {
    int key; // valore del nodo
    struct node* up; // nodo padre
    struct node* left; // nodo radice del sottoalbero sinistro
    struct node* right; // nodo radice del sottoalbero destro

};

typedef struct node node;
```

Il campo `up` contiene l'indirizzo del nodo padre oppure `NULL` se il nodo è la radice.

Creazione albero vuoto

In genere è sempre conveniente avere una apposita funzione per creare una struttura dati vuota.

La funzione `newTree()` restituisce un elemento di tipo `node*` che rappresenta l'albero vuoto.

```
node* newTree() {
    return NULL;
}
```

Inserimento di un elemento

La funzione `newNode()` crea un nuovo nodo contenente il valore `n` e restituisce l'indirizzo del nodo creato.

```
node *newNode(int n){
    node *new;
    new = malloc(sizeof(node));
    new->key = n;
    new->left = new->right = new->up = NULL;
    return new;
}
```

Ricordarsi di inizializzare *esplicitamente* i campi che devono valere `NULL`.

Se il nodo contiene elementi di tipo diverso (punti, parole, ...), l'istruzione

```
new->key = n;
```

va sostituita in modo opportuno.

Il nuovo nodo deve diventare una foglia dell'albero.

Quindi:

1. Occorre attraversare l'albero per trovare la foglia che deve diventare il padre del nuovo nodo.

   Nel codice, la variabile `q` percorre l'albero dalla radice alle foglie, `pq` è il padre di `q`.

   Il tempo richiesto è $O(h)$, dove h è l'altezza dell'albero.

2. Si chiama `newNode()` per creare un nuovo nodo contenente `n` che diventerà il figlio (sinistro o destro) della foglia individuata al punto 1.

   Serve un numero costante di operazioni.

Se l'albero è inizialmente vuoto, il nuovo nodo diventerà la radice dell'albero, altrimenti la radice dell'albero non cambia.

La complessità dell'inserimento è $O(h)$.

La funzione `treeInsert()` inserisce un nodo di valore `n` nell'albero `r` e restituisce la radice del nuovo albero.

```
node *treeInsert(int n, node  *r){
    node  *q, *pq;
    if(r == NULL) // l'albero e' vuoto
        return newNode(n);
    else{
//1.  q attraversa l'albero fino a trovare il punto di inserimento
            q = r;
            while(q != NULL){
                    pq = q;
                    q = (n < q->key) ?  q->left :  q->right;
                }
//2.  pq e' il padre del nuovo nodo
            q = newNode(n);
            if (n < pq ->key)
                pq->left = q;
            else
                pq->right = q;
            q->up = pq;
            return r;
        }
}
```

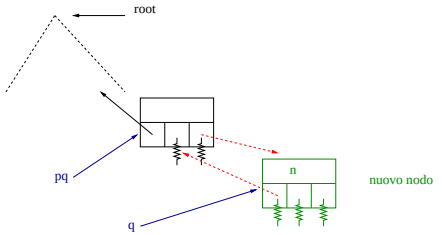

Nel ciclo `while`, `q` attraversa l'albero dalla radice alle foglie.

Con l'istruzione

```
q = (n < q->key) ?  q->left :  q->right;
```

- se vale `n < q->key`, `q` assume il valore di `q->left` (l'attraversamento prosegue nel sottoalbero sinistro di `q`);
- altrimenti, `q` diventa `q->right` (l'attraversamento dell'albero prosegue a destra).

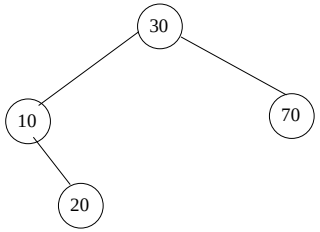
All'inizio del punto 2, `q` è l'indirizzo del nuovo nodo da inserire, `pq` è l'indirizzo della foglia che deve diventare il padre di `q`.



Esempio

```
int main(){
 node *root;
 root = newTree(); // creazione albero vuoto
 root = treeInsert(30, root);
 root = treeInsert(10, root);
 root = treeInsert(70, root);
 root = treeInsert(20, root);
 return 0;
}
```

L'albero ottenuto è:



Più in generale...

Si può scrivere la funzione `treeInsert()` in una forma più generale, in modo che sia corretta qualunque sia l'insieme V dei nodi.

Occorre sostituire l'espressione

```
n < q->key
```

con

```
minore(n, q->key)
```

dove `minore()` è una funzione per il confronto fra due elementi di V che restituisce 1 se il primo argomento è minore del secondo, 0 altrimenti.

Se V è l'insieme degli interi, la funzione `minore()` è

```
int minore(int a, int b){
   return a < b;
}
```

Infatti, l'espressione `a<t` ha tipo `int` e valore 1 se $a < b$, 0 altrimenti.

Supponiamo invece che V sia l'insieme delle parole, che la relazione d'ordine sia l'ordine lessicografico e che il campo `key` di ogni nodo (tipo `char*`) contenga (l'indirizzo a una stringa contenente) una parola.

Allora come funzione `minore()` si può usare

```
int minore(char* w1, char* w2){
   return strcmp(w1,w2) < 0;
}
```

dove `strcmp()` è una funzione standard per il confronto fra stringhe (vedere i manuali).

Esercizio

Come è stato fatto per le liste, scrivere una funzione

```
void  treeInsert1(node** r, int n)
```

che inserisce un nuovo nodo contenente il valore `n` nell'albero la cui radice è nella locazione di memoria `*r` (in altri termini, la radice dell'albero è passata "per indirizzo").

Ricerca di un elemento

Per cercare un elemento in un albero di ricerca non occorre esaminare *tutti* i nodi dell'albero, ma al massimo $h + 1$ nodi, dove h è l'*altezza* dell'albero.

La funzione `treeFind()` cerca l'elemento `n` nell'albero `r`
Se un nodo contenente `n` esiste restituisce il suo indirizzo, altrimenti restituisce `NULL`.

```
node  *treeFind(int n, node *r){
    while (r != NULL &&  r->key != n)
         r = (n < r->key) ? r->left : r->right;
    return r;
}
```

La complessità nel caso medio è $O(h)$, dove h è l'altezza dell'albero.

Si osservi l'ordine con cui è scritta la condizione

```
r != NULL  &&  r->key != n
```

Infatti la condizione `r != NULL` va valutata *prima* della condizione `r->key != n`; in caso contrario si ha errore in esecuzione (segmentation fault) quando `r` vale `NULL`.

Più in generale, si possono sostituire le espressioni

```
r->key != n              n < r->key
```

rispettivamente con

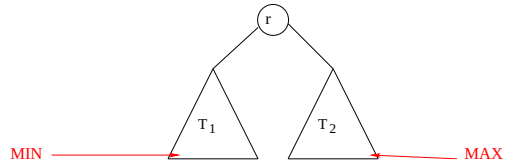
```
!uguale(r->key, x)       minore(x, r->key)
```

dove le funzioni `uguale()` e `minore()` vanno definite opportunamente in base ai valori V dei nodi.

Ricerca del minimo e del massimo

Il nodo contenente il minimo valore si trova scendendo nell'albero a sinistra fino a quando si raggiunge `NULL`, il massimo invece scendendo a destra.

La complessità è $O(h)$, dove h è l'altezza dell'albero (mentre in una lista la complessità è $O(n)$, dove n è la dimensione della lista).



La funzione `treeMin()` restituisce l'indirizzo del nodo di valore minimo dell'albero `r`.

```
node* treeMin(node *r){
    for( ; r->left != NULL; r = r->left);
    return r;
}
```

La funzione `treeMax()` restituisce l'indirizzo del nodo di valore massimo dell'albero di radice `r`.

```
node* treeMax(node *r){
    for( ; r->right != NULL; r = r->right);
    return r;
}
```

Nota

In `treeMin()` e in `treeMax()` non viene mai usato il valore del campo `key`. Quindi entrambe le funzioni possono essere usate anche quando i nodi contengono elementi di tipo diverso dagli interi.

Cancellazione di un elemento

```
node *treeDelete(node *x, node *r)
```

cancella il nodo di indirizzo `x` dall'albero `r` e restituisce l'indirizzo della radice dell'albero ottenuto.

Prima di cancellare `x`, viene scelto un nodo `s` che andrà a sostituire `x`.

(1) `x` *non ha figli*.

`x` è sostituito dall'albero vuoto, quindi `s` vale `NULL`.

(2) `x` *ha un unico figlio*.

`s` è il figlio di `x`.

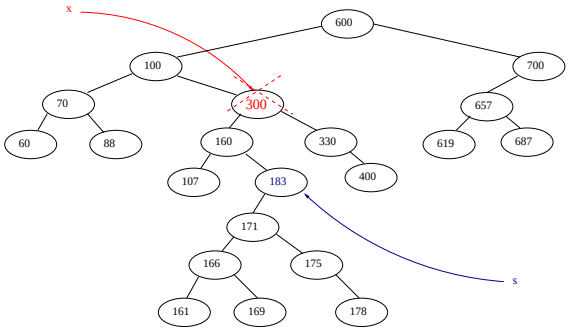
(3) `x` *ha due figli*.

`s` è il maggiore fra i nodi minori di `x`; questo significa che `s` è il nodo più a destra nel sottoalbero sinistro di `x`.
Si noti che `s` non ha il figlio destro, ma può avere il figlio sinistro. In questo caso, prima di sostituire `x` con `s`, il figlio sinistro di `s` diventa figlio del padre di `s` (in altri termini, il figlio sinistro di `s` prende il posto di `s`).

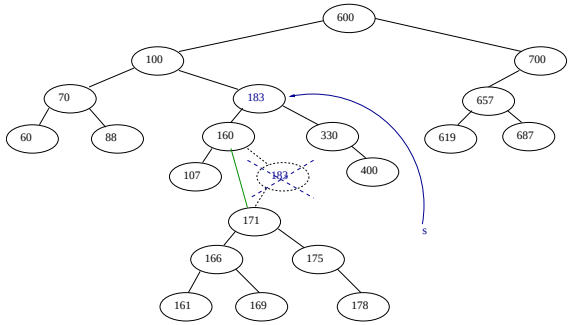
Si sostituisce `x` con `s` e si elimina `x`. Se `x` era la radice dell'albero, `s` diventa la nuova radice.

Esempio caso 3

Si vuole cancellare il nodo 300 dall'albero qui sotto.
Il nodo `s` è 183 (maggiore fra i nodi nel sottoalbero sinistro di 300).



Dopo la sostituzione di `x` con `s` e la cancellazione di `x`, l'albero diventa:



Verifica della correttezza

Occorre controllare che in tutti e tre i casi l'albero ottenuto sia ancora un albero binario di ricerca.

Per i primi due casi la verifica è immediata.

Per quanto riguarda il terzo caso, dopo la sostituzione di `x` con `s` vale:

- Se `v` è un nodo del sottoalbero sinistro di `s` (ad esempio, i nodi 160 e 175 dell'esempio precedente), allora `v` prima della sostituzione si trovava nel sottoalbero sinistro di `x`. Poiché `s` è il maggiore fra i nodi del sottoalbero sinistro di `x`, si ha che `v < s`.
- Se `v` è un nodo del sottoalbero destro di `s` (nell'esempio precedente, i nodi 330 e 400), allora `v` prima della sostituzione si trovava nel sottoalbero destro di `x`, quindi `x < v`. D'altra parte `s < x` (`s` si trovava nel sottoalbero sinistro di `x`), quindi, per la transitività di $<$, si ha `s < v`.

La complessità dell'operazione di cancellazione è proporzionale all'altezza h dell'albero, quindi è $O(h)$.

Nota

La funzione non utilizza mai il valore `key` di un nodo. Quindi, anche in questo caso la funzione può essere usata, senza modifiche, anche nel caso in cui V contenga elementi di tipo diverso degli interi.

Per cancellare il nodo `x` viene fatta la chiamata

```
freeNode(x);
```

dove la funzione `freeNode()` va definita opportunamente.

La struttura di `treeDelete()` (vedere il codice completo in `tree.c`) è:

```
node *treeDelete(node *x, node *r){
    node *s;  // nodo che sostituisce x

/*** Trova  il nodo s  che deve sostituire x  ***/

    if(x->left==NULL && x->right == NULL)
        /**  CASO 1: x non ha figli  **/
            s = NULL;
    else if (x->left == NULL || x->right == NULL)
        /**  CASO 2: x  ha un unico figlio   **/
            s = (x->left != NULL) ? x->left : x->right;
    else {
       /**   CASO 3: x ha due figli   **/
         ....
        }
```



```
/***  Il nodo  x viene sostituito con s.
      Se x e' la radice,  s diventa la radice del nuovo albero  ***/

    if(x == r){ // x e' la radice
            r = s; // s e' la nuova radice
            if( r!= NULL) // il nuovo albero non e' vuoto
                r->up = NULL;
        }
    else if (x->up->left == x){ // x e' figlio sinistro
            x->up->left = s;
            if(s != NULL) // s non e' l'albero vuoto
                s->up = x->up;
        }
    else{   // x e' figlio  destro
            x->up->right = s;
            if(s != NULL) // s non e' l'albero vuoto
                s->up = x->up;
        }

/***  Eliminazione del nodo x   ***/

    freeNode(x); // da definire opportunamente
    return r;
}
```


Nel caso V sia l'insieme degli interi, la funzione `freeNode()` da usare è:

```
void freeNode(node *p){
  free(p);
}
```

Se `p->key` è l'indirizzo di locazioni di memoria allocate dinamicamente, prima di eseguire `free(p)` occorre deallocare la memoria utilizzata da `p->key`.

Visita di un albero binario

L'obiettivo è quello di scrivere delle funzioni per attraversare tutti i nodi di un *generico* albero binario (anche non di ricerca), dove su ogni nodo `r` visitato viene eseguita la chiamata

```
visita(r)
```

e la funzione `visita()` è passata come parametro.

Per definire un parametro formale di tipo funzione, occorre specificare il tipo della funzione e degli argomenti della funzione (in realtà il parametro è un *puntatore a funzione*; vedere i dettagli sui manuali).

Nei prossimi esempi viene usato il parametro formale `visita()`, che è (il puntatore a) una funzione che ha tipo `void` e che ha un argomento di tipo `node*`.

Va dichiarato come

```
void visita(node *)
```

oppure, equivalentemente,

```
void (*visita)(node *)
```

Le parentesi tonde attorno a `*visita` sono necessarie, senza di esse la dichiarazione è interpretata come

```
(void *) visita(node *)
```

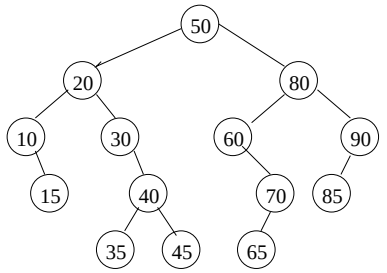
e significa che la funzione `visita()` restituisca un valore di tipo `void*` (puntatore "generico").

Un esempio di funzione che può istanziare il parametro `visita()` è la seguente funzione che stampa il valore del nodo `p`.

```
void printNode(node  *p){
    printf("%d ", p->key);
}
```


Visita in ordine anticipato (preorder)

Viene visitata prima la radice dell'albero, poi il sottoalbero sinistro e poi il sottoalbero destro. Nell'albero binario



l'ordine in cui i nodi sono visitati è:

```
50 20 10 15 30 40 35 45 80 60 70 65 90 85
```


La soluzione più naturale è di scrivere una funzione ricorsiva.

```
void preorder(node *r , void visita(node*)){
    if(r != NULL){
          visita(r);
          preorder(r->left, visita);
          preorder(r->right, visita);
    }
}
```

Esercizio

Scrivere una versione non ricorsiva della funzione `preorder()`.
Fare la stessa cosa per le altre funzioni di visita.

Esempio

Scrivere una funzione

```
void printPreorder(node *r)
```

che stampa in ordine anticipato il valore dei nodi di un albero binario di radice `r`.

Soluzione 1

Si usa uno schema ricorsivo analoga a quello della funzione `preorder()`.

```
void printPreorder(node *r){
    if(r != NULL){
            printf("%d ", r->key);
            printPreorder(r->left);
            printPreorder(r->right);
       }
}
```

Soluzione 2

Si chiama la funzione `preorder()` passando come primo argomento `r` e come secondo argomento la funzione `printNode()` definita prima.

```
void printNode(node  *p){
    printf("%d ", p->key);
}


void printPreorder(node *r) {
  preorder(r, printNode);
}
```


Visita in ordine simmetrico (inorder)

Viene visitato prima il sottoalbero sinistro, poi la radice, poi il sottoalbero destro.

Nell'esempio di prima, l'ordine di visita è:

```
10 15 20 30 35 40 45 50 60 65 70 80 85 90

void inorder(node* r, void visita(node *)){
    if(r != NULL){
            inorder(r->left,visita);
            visita(r);
            inorder(r->right,visita);
        }
}
```

Se l'albero `r` è un albero binario *di ricerca* (come nell'esempio), i nodi vengono visitati in ordine crescente rispetto all'ordine lineare definito sui nodi.

Visita in ordine differito (postorder)

Viene visitato prima il sottoalbero sinistro, poi il sottoalbero destro, poi la radice.

Nell'esempio di prima, l'ordine di visita è:

```
15 10 35 45 40 30 20 65 70 60 85 90 80 50

void postorder(node * r, void visita(node *)) {
    if(r != NULL){
            postorder(r->left,visita);
            postorder(r->right,visita);
            visita(r);
        }
}
```


Cancellazione di un albero

La funzione

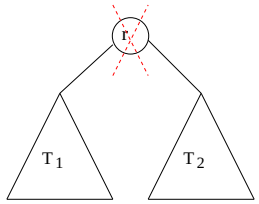
`void treeDestroy(node * r)`

cancella tutti i nodi di un albero la cui radice è `r`.

Attenzione all'*ordine* in cui i nodi vanno eliminati:

- *prima* vanno eliminati i nodi nei sottoalberi sinistro e destro di `r`,
- *poi* il nodo `r`.

Se si elimina per prima il nodo `r`, si perdono gli indirizzi dei sottoalberi.



È quindi un tipico esempio in cui occorre fare una visita in ordine differito.

```
void treeDestroy(node *r){
    if(r != NULL){
            treeDestroy(r->left);
            treeDestroy(r->right);
            freeNode(r);
        }
}
```

Oppure:

```
void treeDestroy(node * r){
  postorder(r, freeNode);
}
```


Rappresentazione di insiemi dinamici mediante alberi binari di ricerca

L'obiettivo è di riscrivere il programma per gestire insiemi dinamici più efficiente della versione basata sulle liste. Il programma deve leggere da standard input una sequenza di istruzioni secondo il formato nella tabella (dove n è un intero) e, quando un'istruzione è letta, eseguire l'operazione corrispondente.

Si assume che l'input sia inserito correttamente.

Conviene scrivere le istruzioni di input in un file `in.txt` ed eseguire il programma redirigendo lo standard input.

Istruzione in input	Operazione
`+` n	Se n non appartiena all'insieme lo inserisce, altrimenti non compie alcuna operazione
`-` n	Se n appartiene all'insieme lo elimina, altrimenti non compie alcuna operazione
`?` n	Stampa un messaggio che dichiara se n appartiene all'insieme
`c`	Stampa il numero di elementi dell'insieme
`p`	Stampa gli elementi dell'insieme
`o`	Stampa gli elementi dell'insieme in ordine crescente
`d`	Cancella tutti gli elementi dell'insieme
`f`	Termina l'esecuzione

Struttura dati

La versione basata sulle liste, pur essendo ottimale nella gestione dello spazio, non è efficiente in quanto la *ricerca*, e quindi anche le operazioni di *inserimento* e *cancellazione* (che richiedono delle ricerche), ha tempo lineare.

Rappresentando l'insieme mediante un albero binario di ricerca, le suddette operazioni possano essere eseguite in tempo logaritmico.

Struttura del programma

Si può suddividere il programma su tre file.

- Il file `tree.c` contiene le funzioni per gestire gli alberi binari di ricerca di interi.
- Il file `tree.h` contiene la definizione del tipo `node` e i prototipi delle funzioni definite in `tree.c`.
- Il file `mainTree.c` (da scrivere per esercizio) contiene solo la definizione della funzione `main()` che è analoga a quella dell'esempio con le liste.

  Per contare il numero dei nodi dell'albero, usare una variabile `count` locale a `main()`.

```
 /*****    mainTree.c    *****/
#include "tree.h"
....
int main(){
  int c, n, count;
  node *root; // radice dell'albero

  root = newTree(); // creazione albero vuoto
  count = 0; // numero nodi nell'albero
  while((c=getchar()) != 'f'){
    switch(c){
      case '+':  // aggiunta di un elemento
      ....
      break;
      case '-':  // eliminazione di un elemento
      ....
      break;
      // ALTRI CASI
     } // end switch
   } // end while
  treeDestroy(root);
  return 0;
}
```



```
.....

case '+':  // aggiunta di un elemento
   scanf("%d", &n);
   if(treeFind(n,root) == NULL){  // n non e' nell'albero
        root = treeInsert(n,root);  // inserisci n
        count++;
    }
break;

...
```

Per stampare i nodi in ordine crescente, è sufficiente una stampa dell'albero in ordine simmetrico.

Compilazione

Per creare l'eseguibile:

```
gcc mainTree.c tree.c
```

Il programma funziona come quello che usa le liste e si può usare lo stesso file di input `in.txt`:

```
a.exe < in.txt
```


Esercizi

1. Modificare le procedure di visita in modo che restituiscano il numero dei nodi dell'albero.

2. Scrivere una funzione ricorsiva che visita i nodi di un albero binario di ricerca in ordine decrescente.

3. Scrivere una funzione ricorsiva che stampa gli n nodi più piccoli di un albero binario di ricerca (n è un parametro della funzione).

4. Scrivere una funzione ricorsiva che stampa gli n nodi più grandi di un albero binario di ricerca (n è un parametro della funzione).