Implementazione non ricorsiva

Utilizzando esplicitamente uno stack, invece di affidarsi allo stack dei record di attivazione gestito dal sistema C, è possibile riformulare le funzioni ricorsive precedenti in modo iterativo.

```
void preorderit(inttree *p,void (*op)(inttree *))
{
    struct astack *s = createastack();

    apush(s,p);
    while(!aempty(s)) {
        p = apop(s);
        (*op)(p);
        if(p->right)
            apush(s,p->right);
        if(p->left)
            apush(s,p->left);
    }
    destroyastack(s);
}
```

Sostituendo lo stack `astack` con la corrispondente implementazione di coda `aqueue`, è immediato implementare la visita *level–order* dell'albero:

```
void levelorderit(inttree *p,void (*op)(inttree *))
{
    struct aqueue *s = createaqueue();

    aenqueue(s,p);
    while(!aemptyq(s)) {
        p = adequeue(s);
        (*op)(p);
        if(p->left)
            aenqueue(s,p->left);
        if(p->right)
            aenqueue(s,p->right);
    }
    destroyaqueue(s);
}
```

(Vedi `inttreeit.c` per la definizione di `struct astack`, etc.)

Alberi Binari di Ricerca

Gli alberi binari di ricerca (o, **alberi di ricerca binaria**) sono strutture dati che consentono, su un insieme di elementi con un ordine totale le operazioni di:

– **ricerca** di un elemento, **verifica** dell'appartenenza di un elemento a un sottoinsieme.
– **inserimento** e **cancellazione** di elementi.
– reperimento del **minimo** e del **massimo** elemento di un sottoinsieme.
– determinazione del **successore** e del **predecessore** di un elemento in un sottoinsieme.

Definizione:

Sia $< A, \leq >$ un insieme totalmente ordinato e sia $S \subseteq A$.

Un **albero di ricerca binaria** per S è un albero binario T i cui nodi sono etichettati da elementi di S.
Inoltre, detta $E(v)$ l'etichetta del nodo v di T valgono le seguenti:

– per ogni elemento s di S esiste uno e un solo v tale che $E(v) = s$.
– per ogni nodo v di T, se u appartiene al sottoalbero destro di v, allora $E(v) < E(u)$.
– per ogni nodo v di T, se u appartiene al sottoalbero sinistro di v, allora $E(v) > E(u)$.

Implementazione

Assumiamo che l'insieme A delle etichette (*chiavi*) siano gli interi.

Collochiamo nel file `searchtree.h` la seguente `typedef`

```
typedef int key;
```

Rappresenteremo i nodi dell'albero utilizzando una `struct` con tre campi puntatore: oltre ai puntatori `left` e `right` ai sottoalberi sinistro e destro, memorizzeremo nel puntatore `up` l'indirizzo del nodo *padre*.

```
struct searchtree {
    key v;
    struct searchtree *left, *right, *up;
};

typedef struct searchtree searchtree;
```

Ovviamente, oltre alla chiave, un nodo può contenere altre informazioni, che possono essere rappresentate aggiungendo nuovi campi membro o campi puntatore alla struttura `searchtree`.

Ad esempio, se ogni nodo deve essere associato a una stringa `nome`, modificheremo come segue la nostra definizione della struttura `searchtree`:

```
struct searchtree {
    key v;
    char *nome;
    struct searchtree *left, *right, *up;
};
```

NOTA: l'implementazione discussa gestisce "senza problemi" (quando l'unica cosa da gestire è il valore della chiave) alberi binari di ricerca in cui i valori per il campo `key v` possano trovarsi ripetuti in nodi distinti.

Operazioni sugli alberi binari di ricerca

Essendo a tutti gli effetti degli alberi binari, l'implementazione delle visite **inorder**, **preorder**, **postorder**, **level-order** può essere mutuata dalle corrispondenti funzioni per alberi binari generici:

```
void inorder(searchtree *p, void (*op)(searchtree *))
{
    if(p) {
        inorder(p->left,op);
        (*op)(p);
        inorder(p->right,op);
    }
}
```

La visita in ordine simmetrico di un albero di ricerca binaria produce un elenco ordinato delle etichette (*chiavi*) dell'albero.

Chiaramente, la visita in ordine di un albero con n nodi richiede $\Theta(n)$ passi.

Ricerca di una chiave in un albero:

```
searchtree *search(searchtree *p, key k)
{
    if(!p || k == p->v)
        return p;
    return search(k < p->v ? p->left : p->right, k);
}
```

`search` ritorna `NULL` se l'albero con radice `p` non contiene alcun nodo con chiave `k`, altrimenti ritorna un puntatore al (nodo più a sinistra) con chiave `k`.

`search` sfrutta in modo essenziale le proprietà definitorie di un albero di ricerca binaria: ricorsivamente viene esaminato il sottoalbero sinistro se `k < p->v`, il sottoalbero destro altrimenti.

Il tempo di esecuzione è $O(h)$ dove h è l'altezza dell'albero.

Un esercizio di ripasso sull'operatore ternario `? :`
Versioni alternative di `search`:

```
searchtree *search2ndvers(searchtree *p, key k)
{
    if(!p || k == p->v)
        return p;
    return k < p->v ? search2ndvers(p->left,k) : search2ndvers(p->right,k);
}

searchtree *search3rdvers(searchtree *p, key k)
{
    if(!p || k == p->v)
        return p;
    if(k < p->v)
        return search3rdvers(p->left,k);
    else
        return search3rdvers(p->right,k);
}
```

Eccone una versione iterativa:

```
searchtree *itsearch(searchtree *p, key k)
{
    while(p && k != p->v)
        p = k < p->v ? p->left : p->right;
    return p;
}
```

Anche le operazioni di estrazione del **minimo** e del **massimo** elemento richiedono $O(h)$ passi, per h l'altezza dell'albero:

Semplicemente, per trovare il minimo si seguono solo i puntatori `left`:

```
searchtree *treemin(searchtree *p)   /* si assume p != NULL */
{
    for(;p->left;p = p->left);
    return p;
}
```

mentre per trovare il massimo si seguono solo i puntatori `right`:

```
searchtree *treemax(searchtree *p)  /* si assume p != NULL */
{
    for(;p->right;p = p->right);
    return p;
}
```

Successore e **Predecessore**
Per trovare il **successore** di una chiave in un albero, non è necessario passare la radice dell'albero stesso alla funzione: l'unico parametro è un puntatore al nodo di cui si vuole ottenere il successore:

```
searchtree *treesucc(searchtree *q) /* si assume q != NULL */
{
    searchtree *qq;

    if(q->right)
        return treemin(q->right);
    qq = q->up;
    while(qq && q == qq->right) {
        q = qq;
        qq = qq->up;
    }
    return qq;
}
```

`treesucc` restituisce il puntatore al nodo successore oppure `NULL` se tale nodo non esiste.

Il codice di `treesucc` è suddiviso in due casi:

```
if(q->right)
    return treemin(q->right);
```

Se il sottoalbero destro di `q` non è vuoto, il successore di `q` è il nodo più a sinistra del sottoalbero destro: tale nodo viene restituito da `treemin(q->right)`.

```
qq = q->up;
while(qq && q == qq->right) {
    q = qq;
    qq = qq->up;
}
return qq;
```

Altrimenti, se il successore esiste, allora deve essere l'antenato più "giovane" di `q` il cui figlio sinistro è pure antenato di `q`. Per determinarlo, `qq` punta al padre di `q`; la scansione fa risalire `q` e `qq`, fino a quando non si trova un antenato con le proprietà desiderate: essere figlio sinistro del padre.

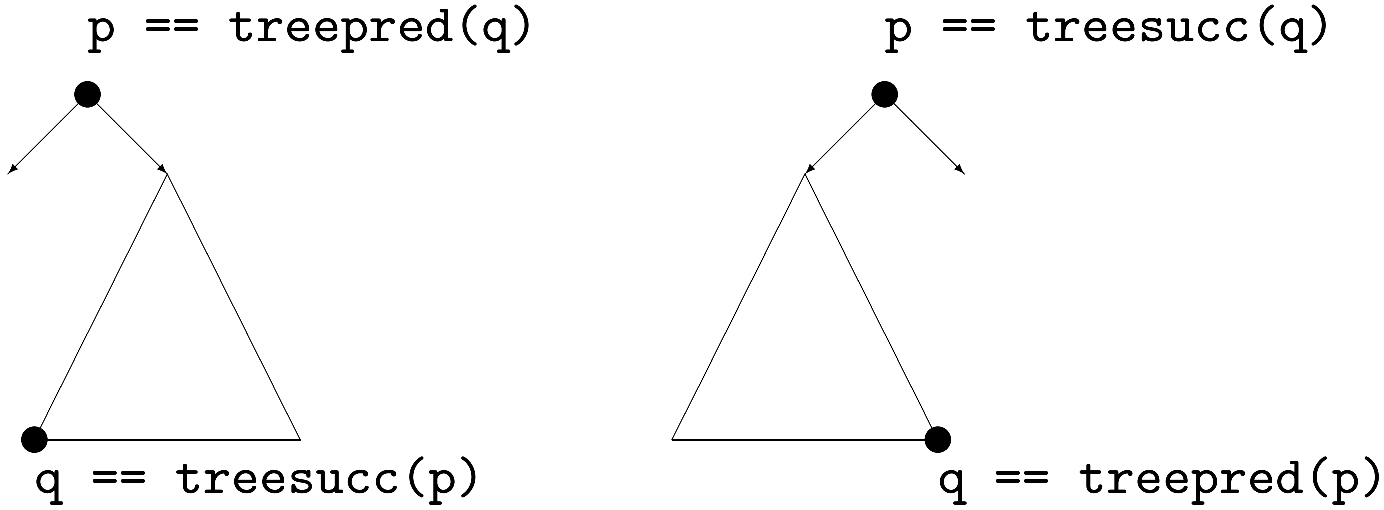
La funzione `treepred` che restituisce, se esiste, il puntatore al predecessore del nodo passato come parametro, è la *duale* di `treesucc`.

```
searchtree *treepred(searchtree *q) /* si assume q != NULL */
{
    searchtree *qq;

    if(q->left)
        return treemax(q->left);
    qq = q->up;
    while(qq && q == qq->left) {
        q = qq;
        qq = qq->up;
    }
    return qq;
}
```

`treesucc` e `treepred` richiedono tempo $O(h)$.

NOTA: Se un nodo ha due figli, allora il suo successore non ha figlio sinistro e il suo predecessore non ha figlio destro.



Nella prima figura `q` non ha figlio sinistro: `q == treemin(p->right)`.

Nella seconda figura `q` non ha figlio destro: `q == treemax(p->left)`.

Inserimento

La funzione `insert` crea un nodo con la chiave `k` e lo introduce "al posto giusto" nell'albero. Viene restituita la nuova radice.

```
searchtree *insert(searchtree *p, key k) {
    searchtree *q = malloc(sizeof(searchtree));
    searchtree *r = p;
    searchtree *s = NULL;

    if(!q) { fprintf(stderr,"Errore di allocazione\n"); exit(-1); }
    q->v = k;
    q->left = q->right = NULL;
    while(r) {
        s = r;
        r = k < r->v ? r->left : r->right;
    }
    q->up = s;
    if(!s)          return q;
    if(k < s->v)    s->left = q;
    else            s->right = q;
    return p;
}
```

```
searchtree *q = malloc(sizeof(searchtree));
...
if(!q) { fprintf(stderr,"Errore di allocazione\n"); exit(-1); }
q->v = k;
q->left = q->right = NULL;
```

In primo luogo viene allocato il nuovo nodo puntato da `q` nella memoria dinamica.

```
while(r) {
    s = r;
    r = k < r->v ? r->left : r->right;
}
```

`r` scende nell'albero scegliendo il ramo sinistro o destro a seconda dell'esito del confronto. `s` punta al padre di `r`.

```
q->up = s;
if(!s)          return q;
if(k < s->v)    s->left = q;
else            s->right = q;
return p;
```

Si collega il nodo puntato da `q` nel punto individuato da `s`. Se `s` è `NULL`, l'albero è vuoto e viene restituita la nuova radice `q`, altrimenti `q` viene agganciato come figlio sinistro o destro a seconda dell'esito del confronto fra le chiavi.

Cancellazione

La funzione `delete` cancella il nodo puntato da `q` dall'albero. Poiché tale nodo può essere la radice, `delete` restituisce la nuova radice.

```
searchtree *delete(searchtree *p, searchtree *q) /* si assume q != NULL */
{
    searchtree *r, *s, *t = NULL;

    if(!q->left || !q->right)   r = q;
    else                        r = treesucc(q);
    s = r->left ? r->left : r->right;
    if(s)                       s->up = r->up;
    if(!r->up)                  t = s;
    else
        if(r == r->up->left)    r->up->left = s;
        else                    r->up->right = s;
    if(r != q)                  q->v = r->v;
    free(r);
    return t ? t : (p != r ? p : NULL);
}
```

`delete` prende in esame tre casi:

1. Se `q` non ha figli, semplicemente si modifica il nodo padre, ponendo a `NULL` il puntatore che puntava a `q`.

2. Se `q` ha un unico figlio, si crea un collegamento tra il padre di `q` e il figlio di `q`.

3. Se `q` ha due figli, si determina il successore `r` di `q`: esso non ha figlio sinistro (vedi NOTA pag. 374). Si elimina `r` come nei casi 1 e 2, previa la sostituzione delle info contenute in `q` con quelle contenute in `r`.

```
searchtree *r, *s, *t = NULL;
```

`r` sarà uguale a `q` nei casi 1 e 2, sarà il successore di `q` nel caso 3. `s` sarà il figlio di `r` oppure `NULL` se `r` non ha figli. `t` sarà diverso da `NULL` solo se la radice dovrà essere modificata.

```
    if(!q->left || !q->right)   r = q;
    else                        r = treesucc(q);
    s = r->left ? r->left : r->right;
```

Viene determinato in quale dei tre casi siamo, e assegnati i valori corrispondenti a `r` e `s`.

```
    if(s)                       s->up = r->up;
    if(!r->up)                  t = s;
    else
        if(r == r->up->left)    r->up->left = s;
        else                    r->up->right = s;
```

Il nodo `r` viene estratto dall'albero, modificando opportunamente il padre di `r` e `s`. Si gestisce il caso in cui `r` è la radice e il caso in cui `s` è `NULL`.

```
    if(r != q)                  q->v = r->v;
    free(r);
    return t ? t : (p != r ? p : NULL);
```

Viene deallocato il nodo puntato da `r`. Se `r != q` siamo nel caso 3 e si copiano le info di `r` in `q`. L'ultima riga ritorna `t` se la radice è da modificare, `p` se non è da modificare, `NULL` se `q` era l'unico nodo dell'albero.

Anche le operazioni di inserimento e cancellazione dell'albero richiedono tempo $O(h)$, per h l'altezza dell'albero.

Distruzione di un albero di ricerca binaria:

(Esercizio: dire perché il codice seguente effettua troppe scansioni di rami dell'albero, e migliorarlo)

```
void destroysearchtree(searchtree *p) /* si assume p != NULL */
{
    while(p = delete(p,p));
}
```

Creazione di un albero di ricerca binaria:

```
searchtree *createsearchtree(void)
{
    return NULL;
}
```

Riepilogo delle Prestazioni

Le operazioni di ricerca, inserimento, cancellazione, minimo, massimo, predecessore, successore, richiedono tempo $O(h)$ per h l'altezza dell'albero.

Nel caso peggiore in cui l'albero degenera in una lista ho $h = n$.

Nel caso medio di un albero di ricerca binaria costruito in modo casuale con n chiavi (distinte) l'altezza h è $h \in O(\log_2 n)$.

Provare `rndtree.c`, per la generazione di permutazioni casuali di n chiavi e per la costruzione degli alberi binari di ricerca mediante inserimenti successivi nell'ordine specificato dalle permutazioni.

Potrete verificare sperimentalmente quale è l'altezza media degli alberi così costruiti.

Alberi Bilanciati

Le operazioni fondamentali sugli alberi di ricerca binaria:

Ricerca, inserimento, cancellazione, min, max, predecessore, successore,

richiedono tempo $O(h)$ dove h è l'altezza dell'albero, vale a dire la lunghezza del suo ramo più lungo.

Se potessimo garantire che ogni nodo ripartisca i suoi discendenti in modo tale che il suo sottoalbero sinistro contenga lo stesso numero di nodi del suo sottoalbero destro, allora avremmo un albero perfettamente *bilanciato*, la cui altezza sarebbe

$$h = 1 + \lfloor \log_2 n \rfloor,$$

dove n è il numero totale di nodi nell'albero.

Non si può richiedere che l'albero soddisfi perfettamente questa condizione.

Ma condizioni un poco più "rilassate" permettono implementazioni efficienti nella gestione della costruzione e della manipolazione di alberi sufficientemente bilanciati.

Agli effetti pratici parleremo di alberi **bilanciati** quando si garantisce che l'altezza dell'albero sia $O(\log_2 n)$.

Vi sono vari criteri che garantiscono questa proprietà: in dipendenza dal criterio definitorio utilizzato abbiamo vari tipi di alberi bilanciati.

Ad esempio possiamo richiedere che per ogni nodo le altezze h_l e h_r dei due sottoalberi sinistro e destro del nodo possano differire al più di 1.

Ogni albero che soddisfi questo criterio è bilanciato poiché ha altezza $O(\log_2 n)$.

Gli alberi che soddisfano questo criterio sono chiamati *Alberi AVL* (Adel'son, Vel'skiĭl, Landis). Gli alberi AVL sono uno dei primi esempi in letteratura di alberi bilanciati gestibili con algoritmi dalle prestazioni accettabili.

Vi sono numerosi tipi di alberi bilanciati: AVL, 2-3, 2-3-4, Red-Black, Splay, B-alberi, etc.

In genere, gli algoritmi per la manipolazione degli alberi bilanciati sono complicati, così come la loro implementazione in C.

Alberi Red-Black

Un albero *Red-Black* è un albero di ricerca binaria in cui ogni nodo contiene come informazione aggiuntiva il suo *colore*, che può essere **red** oppure **black**, e che soddisfa alcune proprietà aggiuntive relative al colore:

1) Ciascun nodo è **red** o è **black**.

2) Ciascuna foglia è **black**.

3) Se un nodo è **red** allora entrambi i figli sono **black**.

4) Ogni cammino da un nodo a una foglia sua discendente contiene lo stesso numero di nodi **black**.

Negli alberi Red-Black una foglia non contiene informazione sulla chiave, ma serve solo a segnalare la terminazione del ramo che la contiene.

Implementativamente, useremo un unico nodo ”dummy” (o ”sentinella”) per rappresentare tutte le foglie.

Per semplicità implementativa è utile imporre un'ulteriore proprietà:

5) la radice è **black**.

Il numero di nodi **black** su un cammino da un nodo p (escluso) a una foglia sua discendente è detta la *b-altezza* del nodo $bh(p)$.

Un sottoalbero con radice p di un albero Red-Black contiene almeno $2^{bh(p)} - 1$ nodi interni.

Per induzione: base: se p è la foglia allora non vi e' alcun nodo interno da conteggiare e infatti $2^0 - 1 = 0$: Ok.
Passo: I figli di p hanno b-altezza $bh(p)$ o $bh(p) - 1$ a seconda del loro colore. Quindi: il numero di nodi interni al sottoalbero di radice p deve essere
$\geq (2^{bh(p)-1} - 1) + (2^{bh(p)-1} - 1) + 1 = 2^{bh(p)} - 1$: Ok.

Poiché su ogni ramo almeno metà dei nodi sono black per la proprietà 3, allora la b-altezza della radice deve essere $\geq h/2$, per h l'altezza dell'albero.
Quindi il numero di nodi interni n dell'albero è tale che:

$$n \geq 2^{bh(root)} - 1 \geq 2^{h/2} - 1.$$

Si conclude:

Un albero Red-Black con n nodi (interni) ha altezza

$$h \leq 2 \log_2(n+1).$$

Dunque, gli alberi Red-Black sono alberi bilanciati, e le operazioni fondamentali di ricerca, min, max, predecessore,successore sono eseguite in tempo $O(\log_2 n)$.

Per quanto riguarda inserimento e cancellazione: dobbiamo vedere come implementare queste operazioni.

Implementazione di alberi Red-Black.

Nell'header file `rbtree.h` riportiamo le definizioni di tipo necessarie:

```
typedef int key;

typedef enum { red, black } color;

struct rbnode {
    key v;
    color c;
    struct rbnode *left, *right, *up;
};

typedef struct rbnode rbnode;

typedef struct {
    rbnode *root, *nil;
} rbtree;
```

DIGRESSIONE: Tipi enumerativi

In C è possibile definire *tipi enumerativi*:

```
enum giorno { lun, mar, mer, gio, ven, sab, dom };

enum giorno lezione;
lezione = mer;
```

Si possono dichiarare variabili di un tipo enumerativo e assegnare loro i valori del tipo stesso.

I valori vengono implicitamente interpretati come costanti di tipo `int`. Per default il primo di questi valori vale 0, e gli altri assumono via via i valori successivi (`lun == 0`, `mar == 1`, `...` , `dom == 6`). Si può altresì specificare esplicitamente il valore numerico dei simboli:

```
typedef enum { lun = 1, mar, mer, gio, ven, sab = 10, dom } giorno;

giorno lezione = mer;  /* mer == 3  */
giorno riposo = dom;   /* dom == 11 */
```

```
struct rbnode {
    key v;
    color c;
    struct rbnode *left, *right, *up;
};
```

Modifichiamo la definizione di nodo di albero binario di ricerca aggiungendo il campo `color c`.

```
typedef struct {
    rbnode *root, *nil;
} rbtree;
```

Una variabile di tipo `rbtree` costituisce un *header* per un albero Red-Black: contiene due puntatori, `root` punterà alla radice dell'albero, mentre `nil` rappresenterà *ogni* foglia dell'albero: è il nodo sentinella.

Creazione dell'albero Red-Black:

```
rbtree *createrbtree(void)
{
    rbtree *t = malloc(sizeof(rbtree));

    if(!t) {
        fprintf(stderr,"Errore di allocazione A\n");
        exit(-1);
    }
    if(!(t->root = malloc(sizeof(rbnode)))) {
        fprintf(stderr,"Errore di allocazione B\n");
        exit(-2);
    }
    t->nil = t->root;
    t->nil->left = t->nil->right = t->nil->up = t->nil;
    t->nil->c = black;
    return t;
}
```

`createrbtree` in primo luogo alloca l'header `t` e il nodo sentinella, controllando l'esito delle allocazioni.

All'inizio l'albero è vuoto, quindi esiste solo il nodo sentinella: `t->root` e `t->nil` punteranno alla sentinella, così come tutti i puntatori della sentinella stessa.

Il colore della sentinella (che rappresenta ogni foglia) è `black`.

Rotazioni

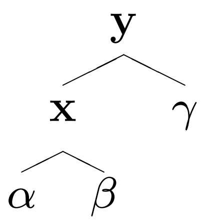
Poiché gli alberi Red-Black sono una versione specializzata degli alberi di ricerca binaria, tutte le operazioni fondamentali implementate per gli alberi di ricerca binaria possono essere applicate agli alberi Red-Black.

Però, le operazioni che modificano l'albero, *Inserimento* e *Cancellazione*, in genere violano le proprietà Red-Black.

Potremo ristabilire le proprietà violate modificando localmente la struttura dell'albero e la colorazione di alcuni nodi senza intaccare l'ordine di visita degli elementi.

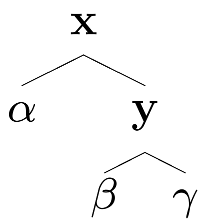
Per questo ci dotiamo delle operazioni di *rotazione sinistra* e *rotazione destra*.

Le rotazioni non modificano l'ordinamento delle chiavi secondo la visita inorder.



`rightrotate(r,y)` $\longrightarrow$

$\longleftarrow$ `leftrotate(r,x)`




```
void leftrotate(rbtree *r, rbnode *x)
{
    rbnode *y = x->right;

    x->right = y->left;
    if(y->left != r->nil)
        y->left->up = x;
    y->up = x->up;
    if(x->up == r->nil)
        r->root = y;
    else
        if(x == x->up->left)
            y->up->left = y;
        else
            y->up->right = y;
    y->left = x;
    x->up = y;
}
```

```
void rightrotate(rbtree *r, rbnode *x)
{
    rbnode *y = x->left;

    x->left = y->right;
    if(y->right != r->nil)
        y->right->up = x;
    y->up = x->up;
    if(x->up == r->nil)
        r->root = y;
    else
        if(x == x->up->right)
            y->up->right = y;
        else
            y->up->left = y;
    y->right = x;
    x->up = y;
}
```

Inserimento

La procedura di inserimento di un elemento nell'albero Red-Black richiede tempo $O(\log_2 n)$.

In primo luogo si alloca il nodo `x` contenente la chiave `k` e si inserisce `x` nell'albero usando la normale procedura **`insert`** per alberi di ricerca binaria (qui è chiamata **`simpleinsert`**).
Il nodo inserito è colorato **`red`**.

Quali proprietà Red-Black possono essere violate ?
1) NO. Il nodo `x` è **`red`**.
2) NO. Il nodo `x` non è una foglia (non è **`*nil`**).
3) SI'. Dipende da dove è stato inserito il nodo.
4) NO. Si è sostituita una foglia **`black`** con un nodo **`red`** che ha due figli foglie **`black`** (**`*nil`**).

La proprietà 3) è violata quando il padre di `x` è `red`.

Per ristabilire la proprietà 3) faremo risalire la "violazione" nell'albero, preservando le altre proprietà: all'inizio di ogni iterazione `x` punterà a un nodo `red` con padre `red`.

Vi sono sei casi da considerare, ma i 3 casi "destri" sono simmetrici ai 3 casi "sinistri".

I 3 casi sinistri (risp., destri) corrispondono a situazioni in cui il padre di `x` è un figlio sinistro (risp., destro).

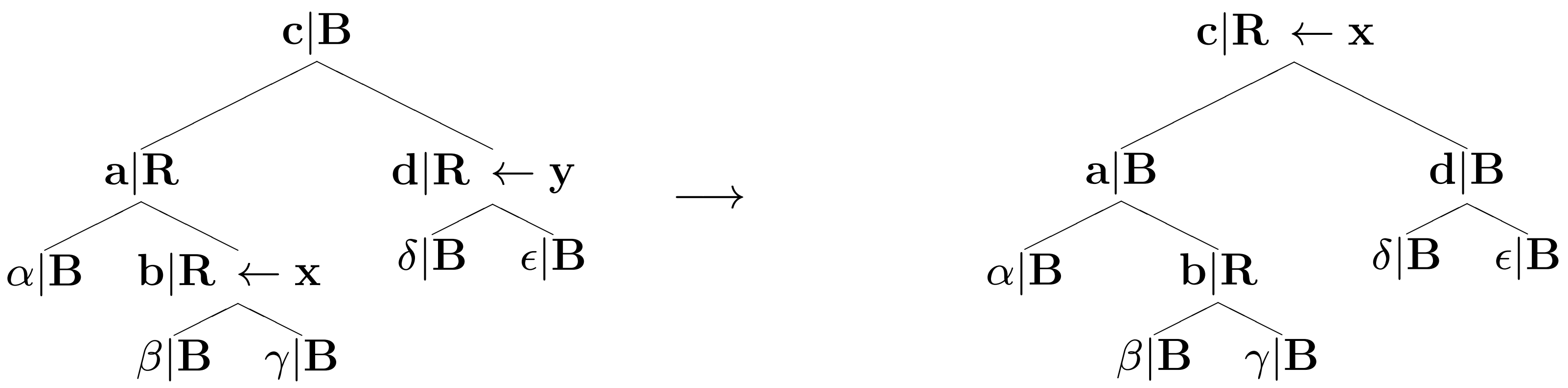
La proprietà 5) garantisce che il nonno di `x` esista sempre.

Assumiamo che il padre `x->up` di `x` sia figlio sinistro.
Il nonno di `x` è `black` perchè vi è una sola violazione nell'albero.
Sia `y = x->up->up->right` lo "zio destro" di `x`.

Caso 1) Lo zio `y` è `red`.
Si colorano `black` i nodi `x->up` e `y` e si colora `red` il nonno di `x`.
A questo punto può essere il nonno `x->up->up` l'unico nodo che viola RB3). Si compie un'altra iterazione.
Caso 2) Lo zio `y` è `black` e `x` è figlio destro.
Una `leftrotate(x->up)` ci porta al caso 3) senza modificare le b-altezze dei nodi ne' violare RB4) poiché sia `x` che `x->up` sono `red`.
Caso 3) Lo zio `y` è `black` e `x` è figlio sinistro.
Una `rightrotate(x->up->up)` e le ricolorazioni del padre e del nonno terminano l'esecuzione, in quanto non ci sono più nodi rossi adiacenti.

Commenti supplementari alla procedura di inserimento.
Ripristino di RB3): i tre casi.

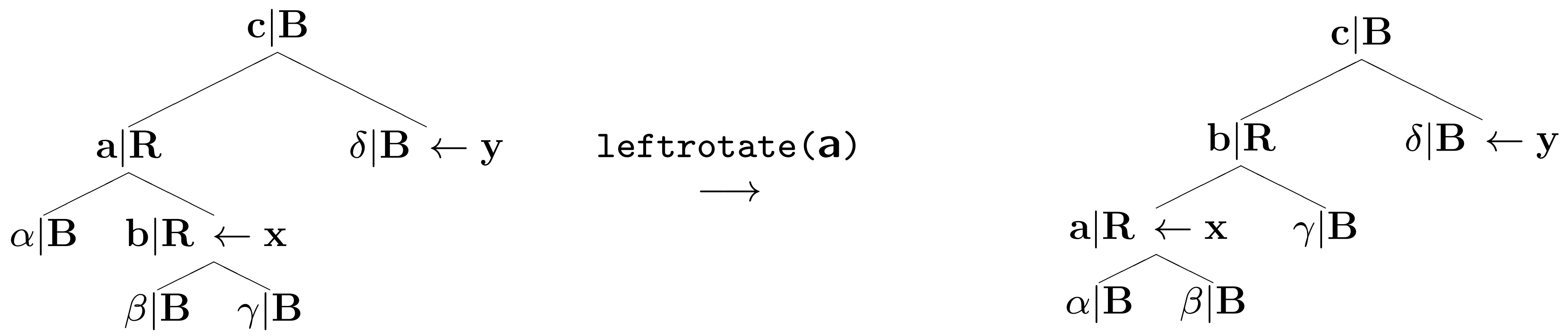
Caso 1)



Si applica anche quando `x` è figlio sinistro del padre.

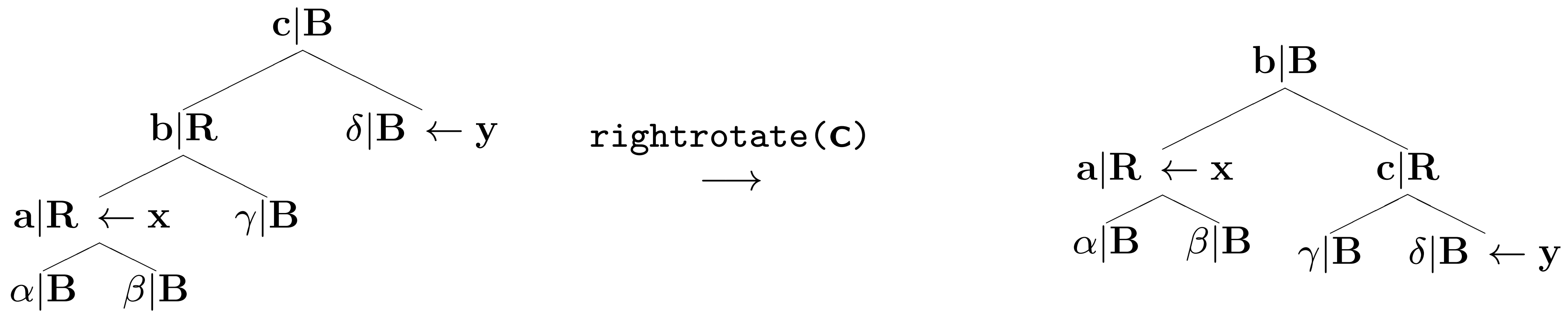
Si itera col nuovo valore di `x`.

Caso 2)



Porta al caso 3).

Caso 3)



La proprietà RB3) è ripristinata.

```
void rbinsert(rbtree *tree, key k)
{
    rbnode *x = simpleinsert(tree, k);    /* inserisce k come in albero binario */
    rbnode *y;                            /* y sara' lo zio di x */

    while(x != tree->root && x->up->c == red) {
        if(x->up == x->up->up->left) {            /* caso L */
            y = x->up->up->right;
            if(y->c == red) {
                x->up->c = black;                 /* caso 1L */
                y->c = black;                     /* caso 1L */
                x->up->up->c = red;               /* caso 1L */
                x = x->up->up;                    /* caso 1L */
            } else {
                if(x == x->up->right)             /* caso 2L */
                    leftrotate(tree,x = x->up);   /* caso 2L */
                x->up->c = black;                 /* caso 3L */
                x->up->up->c = red;               /* caso 3L */
                rightrotate(tree,x->up->up);      /* caso 3L */
            }
        } else {                                  /* caso R */
            ...
        }
    }
    tree->root->c = black;         /* colora black la radice */
}
```

```
rbnode *simpleinsert(rbtree *tree, key k)
{
    rbnode *q = malloc(sizeof(rbnode));
    rbnode *r = tree->root;
    rbnode *s = tree->nil;

    if(!q) {
        fprintf(stderr,"Errore di allocazione C\n");
        exit(-4);
    }
    q->v = k;
    q->left = q->right = tree->nil;
    q->c = red;                      /* colora red il nodo da inserire */
    while(r != tree->nil) {
        s = r;
        r = k < r->v ? r->left : r->right;
    }
    q->up = s;
    if(s == tree->nil)
        return tree->root = q;
    if(k < s->v)
        s->left = q;
    else
        s->right = q;
    return q;                        /* ritorna un puntatore al nodo inserito */
}
```

Commenti sulle prestazioni.

Le procedure di rotazione `leftrotate` e `rightrotate` richiedono tempo $O(1)$.

Nella funzione `rbinsert` il ciclo `while` è ripetuto solo se si esegue il caso 1 con conseguente spostamento verso la radice del puntatore `x`.

Tale ciclo può essere eseguito al più $O(\log_2 n)$ volte, dunque `rbinsert` richiede tempo $O(\log_2 n)$.

Si noti che `rbinsert` effettua al più due rotazioni, poiché se si esegue il caso 2 o 3 la funzione termina.

Cancellazione
La procedura di cancellazione di un nodo da un albero Red-Black richiede tempo $O(\log_2 n)$.
E' però considerevolmente più complicata dell'operazione di inserimento.

In primo luogo si richiama la procedura di cancellazione per nodi da alberi di ricerca binaria, modificata leggermente:
-Non c'e' bisogno di gestire alcuni casi terminali grazie all'uso della sentinella.
-Se il nodo estratto è `black`, viene violata la proprietà RB4).
-Se il figlio del nodo estratto è `red`, può essere momentaneamente violata RB3): la si ripristina subito colorando `black` il figlio.

Prima di uscire da `delete` si invoca un'apposita funzione `fixup` sul figlio del nodo estratto, che ripristinerà RB4) (e la momentanea violazione di RB3)) preservando le altre proprietà Red-Black.

```
void rbdelete(rbtree *tree, rbnode *q)
{
    rbnode *r, *s;

    if(q->left == tree->nil || q->right == tree->nil)
        r = q;
    else
        r = treesucc(tree,q);
    s = r->left != tree->nil ? r->left : r->right;
    s->up = r->up;           /* non si controlla che s non sia nil */
    if(r->up == tree->nil)
        tree->root = s;
    else
        if(r == r->up->left)
            r->up->left = s;
        else
            r->up->right = s;
    if(r != q)
        q->v = r->v;
    if(r->c == black)          /* se il nodo estratto e' black */
        fixup(tree, s);        /* richiama fixup sul figlio s  */
    free(r);
}
```

```
void fixup(rbtree *tree, rbnode *x) {
    rbnode *w;
    while(x != tree->root && x->c == black) {
        if(x == x->up->left) {                                   /* caso L */
            if((w = x->up->right)->c == red) {
                w->c = black;   x->up->c = red;                  /* caso 1L */
                leftrotate(tree,x->up);                          /* caso 1L */
                w = x->up->right;                                /* caso 1L */
            }
            if(w->left->c == black && w->right->c == black) {
                w->c = red;                                      /* caso 2L */
                x = x->up;                                       /* caso 2L */
            } else {
                if(w->right->c == black) {
                    w->left->c = black; w->c = red;              /* caso 3L */
                    rightrotate(tree,w);                         /* caso 3L */
                    w = x->up->right;                            /* caso 3L */
                }
                w->c = x->up->c;    x->up->c = black;            /* caso 4L */
                w->right->c = black;                             /* caso 4L */
                leftrotate(tree,x->up);                          /* caso 4L */
                x = tree->root;                                  /* caso 4L */
            }
        } else { /* caso R */ }
    }
    x->c = black;                               /* colora black il nodo rosso o la radice */
}
```

Cancellazione.
Analisi della funzione `fixup(rbtree *tree, rbnode *x)`

La procedura `fixup` ripristina le proprietà Red-Black dopo la cancellazione di un nodo colorato **black**.

Al momento dell'eliminazione del nodo, l'unica proprietà che può essere violata è RB4) (NB: anche RB3 se il nodo `x` e suo padre sono **red**, ma questo si risolve subito colorando `x` **black**):

RB4) Ogni cammino da un nodo a una foglia sua discendente contiene lo stesso numero di nodi **black**.

Dobbiamo ripristinarla senza violare le altre proprietà.

RB4) è violata:

Ogni cammino che conteneva il nodo **black** estratto, ora ha un nodo **black** di meno. Ogni antenato del nodo estratto viola RB4).

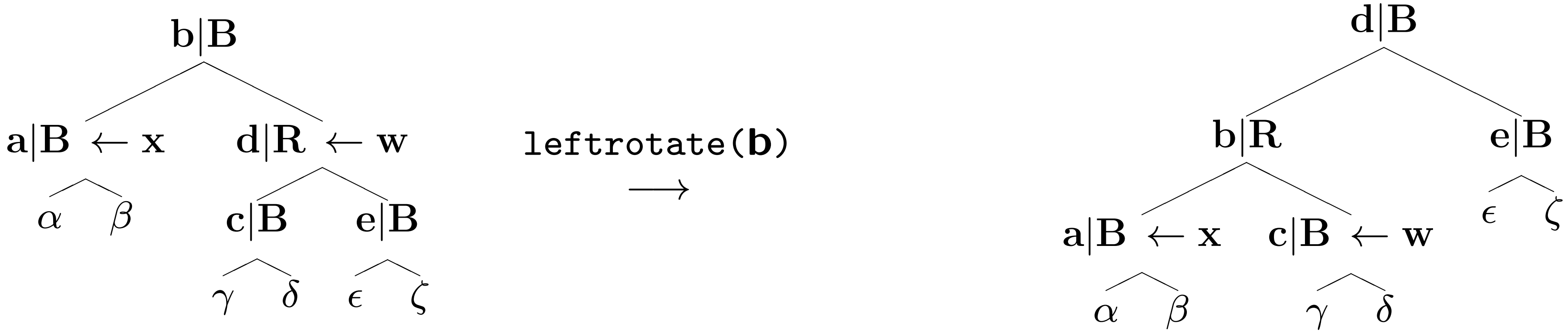
Se (l'unico) figlio del nodo estratto è **red** lo ricoloriamo **black** e così facendo abbiamo ristabilito RB4).

Se invece il figlio era **black**, si risale nell'albero fino a quando, o si trova un nodo **red** da poter colorare **black**, o si raggiunge la radice, o si possono eseguire opportune rotazioni e ricolorazioni che risolvano il problema.

A ogni iterazione `x` punta a un nodo **black**.
Se `x` è figlio sinistro vi sono 4 casi da trattare.
Dualizzando si ottengono i 4 casi per `x` figlio destro.

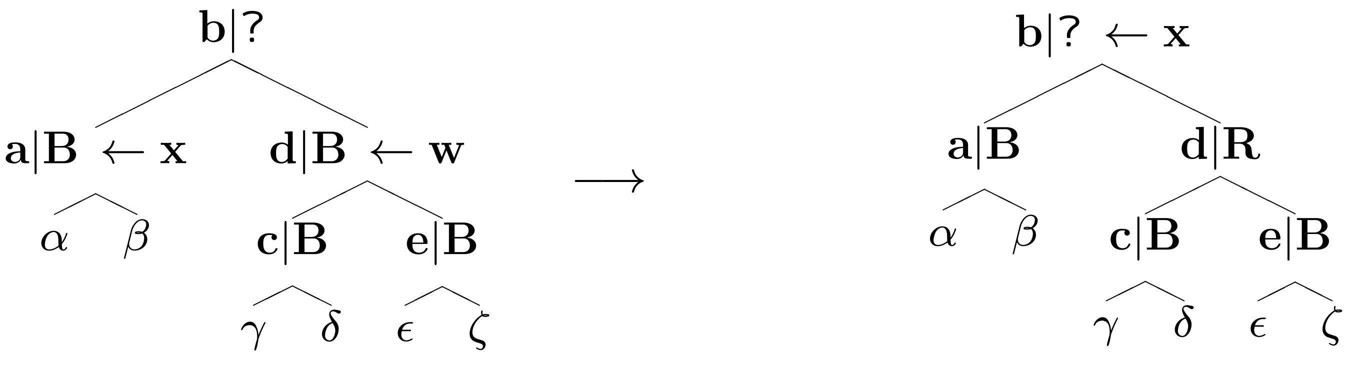
Sia `w` il fratello di `x`. `w` non è foglia, altrimenti RB4) era già violata prima della cancellazione.

Caso 1) `w` è **red** (il padre di `x` deve quindi essere **black**).



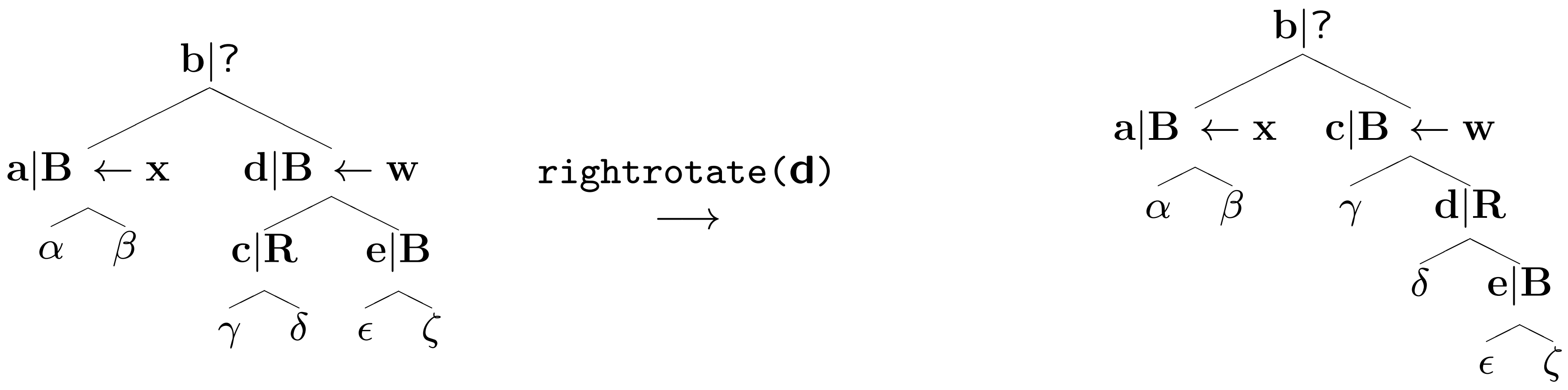
Porta a uno dei casi 2), 3), 4).

Caso 2) `w` è **black**. Entrambi i figli di `w` sono **black**.
Si può colorare `w` **red**. Il problema della violazione di RB4) si sposta sul padre di `x`.



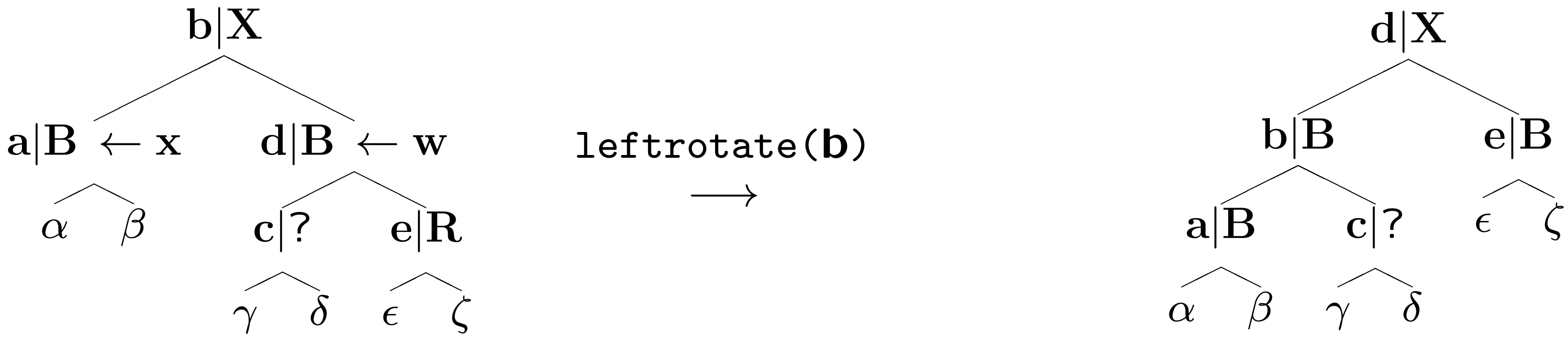
Si itera sul nuovo valore di `x`.

Caso 3) `w` è **black**. il suo figlio sinistro è **red** e invece il suo figlio destro è **black**.



Porta al caso 4).

Caso 4) `w` è **black**. il suo figlio destro è **red**.



`x` viene posto = `root`.

La proprietà RB4) è ripristinata.

Infatti ogni cammino che passa per il nodo che era puntato da `x` incontra un `black` in più. Il numero di nodi `black` su ogni cammino che non tocca `x` non è stato modificato.

Commenti sulle prestazioni.

La cancellazione del nodo, senza `fixup` richiede $O(\log_2 n)$.
Per quanto riguarda `fixup`:
– Ogni rotazione richiede tempo $O(1)$.
– I casi 3) e 4) eseguono al più due rotazioni, e poi la procedura termina.
– Il caso 1) richiede una rotazione, poi:
Se il caso 1) porta al caso 2) non si verifica alcuna ulteriore iterazione poiché il padre di `x` sarà sicuramente **red**.
Altrimenti il caso 1) porta al caso 3) o 4).
– Il caso 2) può dare luogo a una serie di iterazioni che portano il puntatore `x` a risalire nell'albero. Al massimo si raggiunge la radice dopo $O(\log_2 n)$ iterazioni.

Dunque la cancellazione di un elemento da un albero Red-Black richiede $O(\log_2 n)$ passi e al più 3 rotazioni.

Rappresentazione di Grafi (non orientati)

Vi sono molti modi di rappresentare grafi non orientati.

Quando si deve scegliere come implementare grafi (orientati o meno) in una certa applicazione, bisogna considerare gli eventuali vincoli — impliciti o espliciti — e le possibili operazioni che caratterizzano la classe di grafi adatta all'applicazione.

Consideriamo due rappresentazioni alternative per grafi non orientati generici:
– con *matrice* d'adiacenza.
– con *liste* d'adiacenza.

Matrice d'adiacenza.

Consideriamo un grafo non orientato $G = (V, E)$. Numeriamo i vertici di V: Indichiamo con v_i il vertice di V la cui etichetta è $i \in \{1, 2, \ldots, |V|\}$.

La **matrice di adiacenza** di G è la matrice simmetrica `M` con $|V| \times |V|$ valori booleani in cui `M[x][y] == 1` se e solo se l'arco $(v_{\mathtt{x}}, v_{\mathtt{y}}) \in E$.

La rappresentazione con matrici di adiacenza sono accettabili solamente se i grafi da manipolare *non* sono *sparsi* (A G mancano "pochi" archi per essere il grafo completo su V).

Lo spazio necessario con questa rappresentazione è $O(V^2)$. Anche il tempo richiesto da molti degli algoritmi fondamentali è $O(V^2)$ con questa rappresentazione.

Liste d'adiacenza.

Quando il grafo è sparso, diciamo $|E| = O(|V| \log |V|)$, la rappresentazione attraverso **liste di adiacenza** è spesso da preferire.

In questa rappresentazione vi è una lista per ogni vertice $v_i \in V$.
Ogni elemento di una tale lista è a sua volta un vertice $v_j \in V$.
v_j appartiene alla lista associata a v_i se e solo se $(v_i, v_j) \in E$.

I dettagli implementativi variano a seconda delle necessità applicative.
Ad esempio, le liste di adiacenza possono essere contenute in un array statico o dinamico, una lista concatenata, una tabella hash, un albero di ricerca, etc...
Anche le informazioni contenute nei nodi delle liste di adiacenza dipendono dalle esigenze dell'applicazione.

Consideriamo una semplice implementazione con un array allocato in memoria dinamica di liste d'adiacenza.

Questa implementazione non è adatta ad applicazioni che modifichino frequentemente i grafi.

```
struct node {                /* nodo di lista di adiacenza */
    int v;
    struct node *next;
};

struct graph {               /* struttura associata a ogni grafo */
    int V;                   /* numero nodi  */
    int E;                   /* numero archi */
    struct node **A;         /* array di liste di adiacenza */
};
```

Creazione del grafo

```
struct graph *creategraph(int nv, int ne)
{
    struct graph *g = malloc(sizeof(struct graph));

    if(!g) {
        fprintf(stderr,"Errore di Allocazione\n");
        exit(-1);
    }
    g->E = ne;
    g->V = nv;
    if(!(g->A = calloc(nv,sizeof(struct node *))))  {
        fprintf(stderr,"Errore di Allocazione\n");
        exit(-2);
    }
    return g;
}
```

La funzione `creategraph` richiede il numero di vertici e il numero di nodi del grafo da creare. Alloca spazio per ogni lista di adiacenza, inizialmente vuota.

Lettura del grafo:

Dopo aver creato la struttura delle liste di adiacenza, bisogna inserire gli archi nelle liste stesse.

```
void readgraph(struct graph *g, FILE *fp) {
    int i,v1, v2;

    for(i = 0; i < g->E; i++) {
        fscanf(fp,"%d %d",&v1,&v2);
        g->A[v1-1] = vertexinsert(g->A[v1-1],v2);
        g->A[v2-1] = vertexinsert(g->A[v2-1],v1);
    }
}
```

Usiamo questa funzione che legge da un file gli archi. Nel file ogni riga contiene un unico arco specificato come coppia di vertici.

La funzione per inserire i vertici nelle liste di adiacenza è una normale funzione per l'inserzione in testa a liste concatenate:

```
struct node *vertexinsert(struct node *p, int k) {
    struct node *q = malloc(sizeof(struct node));

    if(!q) { fprintf(stderr,"Errore di Allocazione\n"); exit(-3); }
    q->v = k;
    q->next = p;
    return q;
}
```

Si noti che per ogni arco si devono eseguire due chiamate a `vertexinsert` su due liste diverse dell'array `g->A`.
Dato un arco $(v, w) \in E$, si deve inserire v nella lista associata a w, così come w nella lista associata a v.

Attraversamento in Profondità

L'attraversamento in profondità (**depth-first search, DFS**) di un grafo non orientato consiste nel visitare ogni nodo del grafo secondo un ordine compatibile con quanto qui di seguito specificato:

Il prossimo nodo da visitare è connesso con un arco al nodo più recentemente visitato che abbia archi che lo connettano a nodi non ancora visitati.

L'attraversamento DFS, fra le altre cose, permette l'individuazione delle componenti connesse di un grafo.

Implementiamo la visita DFS tramite una semplice funzione ricorsiva:

```
void dfs1(struct graph *g, int i, int *aux) {
    struct node *t;
    aux[i] = 1;
    for(t = g->A[i]; t; t = t->next)
        if(!aux[t->v - 1]) {
            printf("%d,",t->v);
            dfs1(g,t->v-1,aux);
        }
}

void dfs(struct graph *g) {
    int i, *aux = calloc(g->V,sizeof(int));
    if(!aux) { fprintf(stderr,"Errore di Allocazione\n"); exit(-4); }
    for(i = 0; i < g->V; i++)
        if(!aux[i]) {
            printf("\n%d,",i+1);
            dfs1(g,i,aux);
        }
    free(aux);
}
```

La procedura ricorsiva implementata per la visita DFS usa un array di appoggio per memorizzare quando un vertice è già stato incontrato.

Quando `dfs1` è richiamata da `dfs` si entra in una nuova componente connessa.

`dfs1` richiamerà se stessa ricorsivamente fino a quando tutta la componente è stata visitata.

Poiché `dfs1` contiene un ciclo sulla lista di adiacenza del nodo con cui è richiamata, ogni arco viene esaminato in totale due volte, mentre la lista di adiacenza di ogni vertice è scandita una volta sola.

La visita DFS con liste di adiacenza richiede $O(|V| + |E|)$.

Attraversamento in Ampiezza

L'attraversamento in ampiezza (**breadth-first search, BFS**) è un modo alternativo al DFS per visitare ogni nodo di un grafo non orientato.

Il prossimo nodo da visitare lo si sceglie fra quelli che siano connessi al nodo visitato meno recentemente che abbia archi che lo connettano a nodi non ancora visitati.

Vediamone un'implementazione non ricorsiva che memorizza in una coda i nodi connessi al nodo appena visitato.

Sostituendo la coda con uno stack si ottiene una (leggera variante della) visita DFS.

```
void bfs1(struct graph *g, int i, int *aux) {
    struct node *t;
    intqueue *q = createqueue();
    enqueue(q,i);
    while(!emptyq(q)) {
        i = dequeue(q);
        aux[i] = 1;
        for(t = g->A[i]; t; t = t->next)
            if(!aux[t->v - 1]) {
                enqueue(q,t->v -1);
                printf("%d,",t->v);
                aux[t->v-1] = 1;
            }
    }
    destroyqueue(q);
}

void bfs(struct graph *g) {
    int i, *aux = calloc(g->V,sizeof(int));
    if(!aux) { fprintf(stderr,"Errore di Allocazione\n"); exit(-4); }
    for(i = 0; i < g->V; i++)
        if(!aux[i]) {
            printf("\n%d,",i+1);
            bfs1(g,i,aux);
        }
    free(aux);
}
```

La procedura implementata per la visita BFS usa un array di appoggio per memorizzare quando un vertice è già stato incontrato.

Quando `bfs1` è richiamata da `bfs` si entra in una nuova componente connessa.

`bfs1` usa una coda per memorizzare da quali vertici riprendere la visita quando la lista di adiacenza del vertice corrente è stata tutta esplorata.

Ogni lista è visitata una volta, e ogni arco due volte.

La visita BFS con liste di adiacenza richiede $O(|V| + |E|)$.

Complementi: Qualche nozione ulteriore di Input/Output e gestione di stringhe.

Nel file di intestazione `stdio.h` si trovano i prototipi delle funzioni della famiglia di `scanf` e `printf`.

Ad esempio vi sono le funzioni `sscanf` e `sprintf`.
Funzionano come le rispettive funzioni per l'input/output, ma il loro primo argomento è una stringa di caratteri che sostituisce lo stream di input/output.

```
char s[80];
sprintf(s,"%d, %d, %d, stella!",1,2,3); /* s contiene 1, 2, 3, stella! */

sscanf(s,"%d",&i);                      /* i contiene 1 */
```

Accesso diretto ai file.
Il prototipo delle seguenti funzioni è in `stdio.h`.

```
int fseek(FILE *fp, long offset, int place)
```

Riposiziona il cursore per la prossima operazione sul file binario puntato da `fp`.
La nuova posizione del cursore è `offset` byte da:
– Dall'inizio se `place` è `SEEK_SET`
– Dalla posizione corrente prima di `fseek` se `place` è `SEEK_CUR`
– Dalla fine del file se `place` è `SEEK_END`

```
long ftell(FILE *fp)
```

Restituisce la posizione attuale del cursore associato al file `fp` come numero di byte dall'inizio del file stesso.

```
void rewind(FILE *fp)
```

Riporta il cursore all'inizio del file puntato da `fp`.

`size_t fread(void *a_ptr, size_t el_size, size_t n, FILE *fp)`

Legge al più `n * el_size` byte dal file puntato da `fp` e li pone nell'array puntato da `a_ptr`.
Viene restituito il numero di elementi effettivamente scritti con successo in `a_ptr`.
Se si incontra la fine del file, il valore restituito è minore di `n * el_size`.

`size_t fwrite(const void *a_ptr, size_t el_size, size_t n, FILE *fp)`

Scrive `n * el_size` byte nel file puntato da `fp`, prelevandoli dall'array puntato da `a_ptr`.
Viene restituito il numero di elementi scritti con successo nel file.
In caso di errore viene restituito un valore minore di `n * el_size`.

Consultare il libro, e/o il sito del corso (o altre fonti) per la documentazione relativa alle funzioni di libreria standard i cui prototipi sono in `stdio.h`, `stdlib.h`, `ctype.h`, `string.h`.

Altri file di intestazione interessanti: `assert.h`, `time.h`.