# 8 Dizionari

**Sommario**: *In questo capitolo si descrivono le strutture dati astratte più adatte a memorizzare un dizionario; particolare attenzione viene dedicata allo studio del costo computazionale per eseguire ciascuna operazione fondamentale sui dizionari.*

Un ***dizionario*** è una struttura dati che permette di gestire un insieme dinamico di dati, che di norma è un *insieme totalmente ordinato*, tramite queste tre sole operazioni:

- ***insert***: si inserisce un elemento;
- ***search***: si ricerca un elemento;
- ***delete***: si elimina un elemento.

Abbiamo già visto nel capitolo precedente come queste tre operazioni si possano implementare sulle strutture dati analizzate.
A ben vedere però fra le strutture dati che abbiamo descritto, le uniche che supportano in modo semplice (anche se non efficiente) tutte queste tre operazioni sono gli array, le liste e le liste doppie. Infatti, le code (con o senza priorità, inclusi gli heap) e le pile non consentono né la ricerca né l'eliminazione di un arbitrario elemento, mentre negli alberi l'eliminazione di un elemento comporta la disconnessione di una parte dei nodi dall'altra (cosa che può accadere anche nei grafi) e quindi è un'operazione che in genere richiede delle successive azioni correttive.
Di conseguenza, quando l'esigenza è quella di realizzare un diziona-

rio, ossia una struttura dati che rispetti la definizione data sopra, si ricorre a soluzioni specifiche. Noi ne illustreremo tre:

tabelle ad indirizzamento diretto;
tabelle hash;
alberi binari di ricerca.

Nel seguito della trattazione supponiamo che $U$, l'insieme dei valori delle chiavi, sia costituito da valori interi, che $m$ sia il numero delle posizioni a disposizione nella struttura dati, che $n$ sia il numero degli elementi da memorizzare nel dizionario e che i valori delle chiavi degli elementi da memorizzare siano tutti diversi fra loro.

## 8.1 Tabelle ad indirizzamento diretto

Questo metodo, molto semplice, consiste nel ricorrere a un array nel quale ogni indice corrisponde al valore della chiave dell'elemento da memorizzare in tale posizione.
Ci poniamo nell'***ipotesi*** $n \leq |U| = m$; allora un array $V$ di $m$ posizioni assolve perfettamente il compito di dizionario, e per giunta con grande efficienza. Infatti, tutte e tre le operazioni hanno costo computazionale $\Theta(1)$:

```
def Insert_Indirizz_Diretto (V; k):
   V[k] = dati dell'elemento di chiave k
   return

def Search_Indirizz_Diretto (V; k):
   return dati dell'elemento di chiave k
```

```
def Delete_Indirizz_Diretto (V; k):
    V[k] = None
    return
```

Purtroppo le cose non sono così semplici nel caso dei problemi reali, poiché:

- l'insieme $U$ può essere enorme, tanto grande da rendere impraticabile l'allocazione in memoria di un array V di sufficiente capienza;
- il numero delle chiavi effettivamente utilizzate può essere molto più piccolo di $|U|$: in tal caso vi è un rilevante spreco di memoria, in quanto la maggioranza delle posizioni dell'array $V$ resta inutilizzata.

Per queste ragioni, in generale, si ricorre a differenti implementazioni dei dizionari, a meno che non ci si trovi in quelle rare condizioni così specifiche da permettere l'uso dell'indirizzamento diretto sotto le quali questo è il metodo più efficiente in assoluto per memorizzare i dizionari.

## 8.2 Tabelle hash

Quando l'insieme $U$ dei valori che le chiavi possono assumere è molto grande e l'insieme $K$ delle chiavi da memorizzare effettivamente è invece molto più piccolo di $U$, allora esiste una soluzione detta ***Tabella Hash*** (lett.: carne tritata) che richiede molta meno memoria rispetto all'indirizzamento diretto.

L'idea è quella di utilizzare di nuovo un array di $m$ posizioni, ma questa volta non è possibile mettere in relazione direttamente la chiave con l'indice corrispondente, poiché le possibili chiavi sono molte di più rispetto agli indici. Allora si definisce una opportuna funzione $h$, detta ***funzione hash***, che viene utilizzata per calcolare la posizione in cui va inserito (o recuperato) un elemento sulla base del valore della sua chiave.

Supponendo che la tabella hash $T$ contenga $m$ posizioni, i cui indici vanno da $0$ ad $(m - 1)$, la funzione $h$ mappa $U$ nelle posizioni della tabella:

$$h: U \rightarrow \{0, 1, 2, \ldots, m - 1\}$$

Diciamo che $h(k)$ è il valore hash della chiave $k$.

Questa idea presenta un problema di fondo: anche se le chiavi da memorizzare sono meno di $m$, non si può escludere che due chiavi $k_1 \neq k_2$ siano tali per cui $h(k_1) = h(k_2)$, ossia la funzione hash restituisce lo stesso valore per entrambe le chiavi che quindi andrebbero memorizzate nella stessa posizione della tabella. Tale situazione viene chiamata ***collisione***, ed è un fenomeno che va evitato il più possibile e, altrimenti, risolto.

A tal fine, una *buona funzione hash* deve essere tale da rendere il più possibile *equiprobabile* il valore risultante dall'applicazione della funzione, ossia tutti i valori fra $0$ ed $(m - 1)$ dovrebbero essere prodotti con uguale probabilità. In altre parole, la funzione dovrebbe far apparire come "casuale" il valore risultante, disgregando qualunque regolarità della chiave (da questa considerazione deriva il nome della funzione). Inoltre, la funzione deve essere *deterministica*, ossia

se applicata più volte alla stessa chiave deve fornire sempre lo stesso risultato.

La situazione ideale è quella in cui ciascuna delle *m* posizioni della tabella è scelta con la stessa probabilità, ipotesi che viene detta ***uniformità semplice della funzione hash:***

$$\sum_{k:h(k)=j} P(k) = \frac{1}{m} \text{ per } j=0,1,\ldots,m-1$$

Purtroppo nella realtà ciò non sempre è vero, perché spesso la distribuzione di probabilità dei dati effettivamente presenti nel dizionario non è nota.

**Numeri reali come chiavi**

Supponendo che le chiavi siano numeri reali casuali distribuiti in modo uniforme nell'intervallo *[0, 1)*, allora la funzione:

$$h(k) = \lfloor km \rfloor$$

è una buona funzione hash. Questa assunzione ha il difetto di essere molto forte e di non essere sempre applicabile nelle situazioni reali.

**Numeri interi come chiavi**

Quando le chiavi sono stringhe di caratteri possono essere fatte corrispondere a numeri interi, anche molto grandi, ad esempio interpretando la codifica ASCII delle stringhe stesse come numero binario.

In questo caso la funzione:

$$h(k)=k \bmod m$$

è una buona funzione hash.

In questo caso però deve essere posta cura nella scelta di *m*. In particolare, vanno evitati i valori $m = 2^p$, ossia valori di *m* che siano

potenze di due, poiché in tal caso l'operatore modulo restituirebbe come valore l'esatta sequenza dei $p$ bit meno significativi della chiave, il che di fatto farebbe dipendere $h(k)$ da un sottoinsieme dei bit della chiave $k$.

Facciamo un esempio con dei numeri piccolissimi per capire: sia $m = 2^2 = 4$. Allora, la chiave 10 (in binario 1010) viene mappata in 2 (in binario 10, proprio come le ultime due cifre di 1010), la chiave 8 (1000) in 0 (00), la chiave 6 (0110) in 2 (10), la chiave 5 (0101) in 1 (01). Il lato negativo di questo comportamento è che si tralascia una parte dell'informazione, costruendo il valore hash solo sulla rimanente.

Buone scelte per $m$ sono valori primi non troppo vicini a potenze di due. Ad esempio, $m = 701$ è primo e lontano sia da $512$ che da $1024$. Analogamente, se si lavora con l'aritmetica decimale anziché con quella binaria, vanno evitati i valori di $m$ che siano potenze di 10; buone scelte sono i numeri primi non troppo vicini a potenze di 10.

**Risoluzione delle collisioni**

Si può anche scegliere una funzione hash più complicata, ma qualunque funzione si scelga il problema rimane: l'incidenza delle collisioni deriva in ogni caso dalle specifiche chiavi che devono essere memorizzate. Si potrebbe pensare di scegliere casualmente, ad ogni operazione di hashing, una funzione da una classe di funzioni, ma questo approccio ha lo svantaggio che il calcolo può ovviamente produrre output diversi per la stessa chiave, violando il principio del determinismo sopra citato.

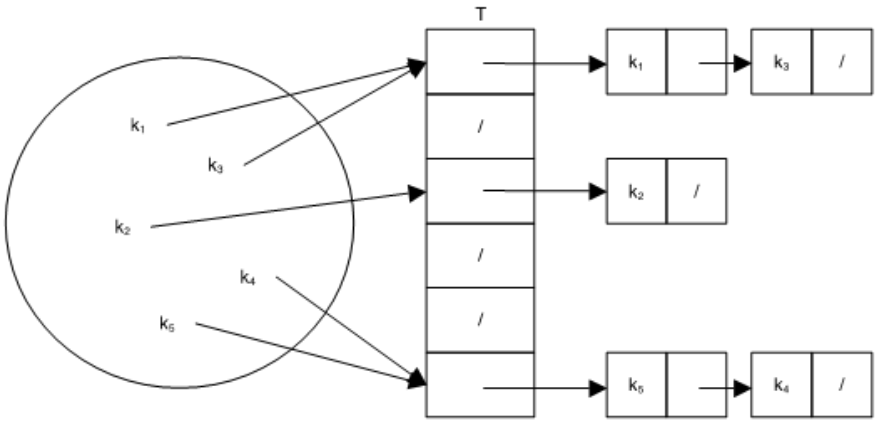
Ad ogni modo, per quanto bene sia progettata la funzione hash, è impossibile evitare del tutto le collisioni perché se $|U| > m$ è inevitabile che esistano chiavi diverse che producono una collisione.
Non potendole evitare, bisogna decidere come risolvere le collisioni.

### 8.2.1 Risoluzione delle collisioni mediante liste di trabocco

Questa tecnica prevede di inserire tutti gli elementi le cui chiavi mappano nella stessa posizione in una lista concatenata, detta *lista di trabocco*.
Possiamo definire le operazioni elementari come segue.
Per inserire un record puntato da x nella tabella T, usiamo l'algoritmo di inserimento in testa in una lista:

```
def Insert_Liste_Di_Trabocco (T, x):
   inserisci_in_testa(T[h(x.key)],x)
   return
```

Questa operazione ha sempre costo computazionale $\Theta(1)$, anche nel caso peggiore.



Assumendo ora di avere la funzione Cerca(L,k), che cerca nella lista concatenata puntata da L il record contenente la chiave k, e restituisce un puntatore a quel record se esso esiste, e None altrimenti:

```
def Search_Liste_Di_Trabocco (T, k):
    return Cerca(T[h(k)],k)
```

Questa operazione ha costo computazionale *O(lunghezza della lista puntata da T[h(k)])* il che, nel caso peggiore, diviene *O(n)* quando tutti gli *n* elementi memorizzati nella tabella hash mappano nella medesima posizione.
Utilizziamo ora la funzione Cancella(L,x), che cancella il record puntato da x dalla lista puntata da L, e poi restituisce il puntatore alla testa della nuova lista:

```
def Delete_Liste_Di_Trabocco (T; x):
    return Cancella(T[h(chiave(x))],x)
```

In questo caso il costo computazionale dipende dall'implementazione delle liste di trabocco e valgono, pertanto, tutte le osservazioni fatte per il costo dell'operazione di cancellazione nelle liste.
Domandiamoci ora quale sia il comportamento di una tabella hash, costituita di *m* posizioni e contenente *n* elementi, nel caso medio. Poiché tale comportamento dipende da quanto bene la funzione hash distribuisce l'insieme delle chiavi sulle *m* posizioni a disposizione, dobbiamo fare alcune assunzioni:

- assumiamo che la funzione hash goda della proprietà di uniformità semplice;

- supponiamo che la funzione hash sia calcolata in un tempo costante;
- definiamo $\alpha = n/m$ come il ***fattore di carico*** della tabella hash, che rappresenta la media dei numeri di elementi memorizzati in ciascuna delle liste di trabocco.

Sotto queste ipotesi, si può enunciare il seguente teorema:

***Teorema.*** In una tabella hash in cui le collisioni sono risolte tramite liste di trabocco, nell'ipotesi di uniformità semplice, il **numero atteso** di accessi effettuati da una ricerca senza successo è $\Theta(1 + \alpha)$.

***Dimostrazione.*** Nell'ipotesi di uniformità semplice, una chiave $k$ corrisponde ad una delle $m$ posizioni in modo equiprobabile. Quindi il numero medio di accessi in una ricerca senza successo è pari a uno per l'accesso alla lista stessa più la lunghezza media delle liste di trabocco, che è $\alpha$, da cui il valore complessivo $\Theta(1 + \alpha)$. CVD

### 8.2.2 Risoluzione delle collisioni mediante indirizzamento aperto

Questa tecnica prevede di inserire tutti gli elementi direttamente nella tabella, senza far uso di strutture dati aggiuntive. Essa è applicabile nelle seguenti condizioni:

- $m$ è maggiore o uguale al numero $n$ di elementi da memorizzare (quindi il fattore di carico non è mai maggiore di 1);
- $|U| >> m$.

L'idea è la seguente: invece di seguire dei puntatori, calcoliamo (nei modi che vedremo fra breve) la sequenza delle posizioni da esami-

nare. Questo perché i puntatori occupano molto spazio, che potrebbe invece essere utilizzato per memorizzare altre chiavi consentendo così potenzialmente di ridurre il numero di collisioni a parità di memoria utilizzata e quindi velocizzare le operazioni.

**Inserimento**

Se la posizione iniziale relativa alla chiave *k* è occupata, si scandisce la tabella fino a trovare una posizione libera nella quale l'elemento con chiave *k* può essere memorizzato. La scansione è guidata da una sequenza di funzioni hash ben determinata: *h(k, O), h(k, 1), ..., h(k, m – 1).*

**Ricerca**

Si scandisce la tabella mediante la stessa sequenza di funzioni hash utilizzata per l'inserimento fino a quando si trova l'elemento o si incontra una casella vuota, nel qual caso si deduce che l'elemento non è presente.

**Eliminazione**

Questa operazione è piuttosto critica. Infatti, se si elimina l'elemento lasciando la casella vuota, ciò impedisce da quel momento in poi di recuperare qualunque elemento sia stato memorizzato in caselle visitate dalla sequenza di funzioni hash dopo quella dalla quale si è eliminato l'elemento (si ricordi che una ricerca viene definita senza successo quando si incontra una casella vuota). D'altronde, se si marca (ad es. con un apposito valore *deleted*) la casella da cui è stato cancellato l'elemento, si risolve il problema della ricerca di elementi successivi ma se ne introduce un altro: il costo computazionale della ricerca non dipende più esclusivamente dal fattore di carico poiché

è influenzata anche dal numero delle posizioni precedentemente occupate da elementi e successivamente marcate. Per queste ragioni di solito *la cancellazione non è supportata con l'indirizzamento aperto.*

## Scansione lineare, scansione quadratica e hashing doppio

La sequenza di funzioni hash da utilizzare nell'indirizzamento aperto dovrebbe possedere due caratteristiche importanti. Da un lato dovrebbe consentire sempre di visitare tutte le $m$ caselle (e quindi produrre sempre una permutazione della sequenza degli $m$ indici), dall'altro dovrebbe essere in grado di produrre tutte le $m!$ permutazioni degli indici per garantire l'uniformità semplice.

Ciò però è molto difficile, ed infatti le tre funzioni hash più comunemente usate:

- scansione lineare,
- scansione quadratica,
- hashing doppio,

garantiscono tutte di generare sempre una permutazione degli indici ma nessuna delle tre è in grado di produrre più di $m^2$ differenti permutazioni. L'hashing doppio è quello che produce il maggior numero di permutazioni e di conseguenza, come ci si può aspettare, è quello che esibisce le migliori prestazioni.

**Scansione lineare**

Data una funzione hash $h'(k)$, la successione di funzioni hash definita dalla *scansione lineare* è la seguente:

$$h(k, i) = (h'(k) + i) \bmod m \text{ per } i = 0, 1, \ldots, m - 1.$$

In altre parole, la scansione percorre in modo circolare la tabella hash partendo da una posizione iniziale che è il risultato del calcolo di $h'(k)$.

Questa tecnica produce solamente $m$ permutazioni diverse, una per ogni distinto valore di $h'(k)$. Soffre di un problema chiamato ***agglomerazione primaria***, poiché tendono a formarsi lunghe sequenze ininterrotte di caselle occupate che aumentano il tempo di ricerca. Di solito, gli agglomerati si verificano perché se una posizione vuota è preceduta da $i$ posizioni piene, allora la probabilità che la posizione vuota sia la prossima ad essere riempita è $(i+1)/m$, in confronto a una probabilità di $1/m$ se la precedente posizione era vuota. Per questo, tratti lunghi di posizioni occupate tendono a diventare ancora più lunghi.

**Scansione quadratica**

Data una funzione hash $h'(k)$, la successione di funzioni hash definita dalla *scansione quadratica* è la seguente:

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m \text{ per } i = 0, 1, \ldots, m - 1.$$

In questo tipo di scansione l'incremento ad ogni passo ha una componente che cresce linearmente ed una che cresce quadraticamente, ciascuna delle quali è corredata di un opportuno coefficiente. Se i valori di $c_1$, $c_2$ ed $m$ sono scelti in modo opportuno le prestazioni sono molto migliori di quelle della scansione lineare.

Anche questa tecnica comunque produce solamente $m$ permutazioni diverse, una per ogni distinto valore di $h'(k)$. Inoltre, soffre di un problema chiamato ***agglomerazione secondaria*** (meno grave dell'agglomerazione primaria) poiché se due chiavi producono lo stesso

valore iniziale anche le successive posizioni visitate sono le stesse per entrambe.

**Hashing doppio**

L'aspetto fondamentale dell'hashing doppio è che la sequenza delle posizioni visitate dipende in due modi diversi dal valore della chiave $k$. Dunque, risolve il problema delle scansioni lineare e quadratica per cui le sequenze di celle visitate per due chiavi $k_1$ e $k_2$ diverse, che iniziano dalla stessa casella, proseguono identiche (ciò accadrà solo ove entrambe le funzioni hash producano una collisione per quella coppia di chiavi, eventualità estremamente rara).

Alla base di questa tecnica vi è l'utilizzo di due diverse funzioni hash anziché una sola. Più formalmente, date due funzioni hash $h_1(k)$ e $h_2(k)$, la successione di funzioni hash definita dall'hashing doppio è la seguente:

$$h(k, i) = (h_1(k) + i\ h_2(k)) \bmod m \text{ per } i = 0, 1, \ldots, m - 1$$

La posizione iniziale dipende dalla prima funzione hash $h_1(k)$ mentre le successive posizioni sono distanziate ciascuna di $h_2(k)$ dalla precedente, ossia il passo di scansione è governato dalla seconda funzione hash. In proposito si noti che il valore di $h_2(k)$ deve essere, per ogni $k$, primo con $m$, altrimenti la scansione non visita tutte le celle. Un modo per garantire questa proprietà è, ad esempio, porre $m$ pari a una potenza di 2 e progettare $h_2(k)$ in modo che fornisca sempre un valore dispari.

Per questa ragione il numero di permutazioni generate dall'hasing doppio è $m^2$ poiché ogni distinta coppia $(h_1(k), h_2(k))$ produce una differente permutazione.

Per queste ragioni il comportamento dell'hashing doppio nella pratica si avvicina molto all'ideale proprietà dell'uniformità semplice della funzione hash.

## Analisi

Valutiamo ora le prestazioni dell'indirizzamento aperto, assumendo che valga la proprietà dell'uniformità semplice, ovverosia che la sequenza di celle visitate sia, per ogni valore $k$ della chiave, una qualunque delle $m!$ permutazioni degli indici. Abbiamo visto che questa ipotesi non è facile da ottenere, ma ci permette di effettuare l'analisi formalmente.

Sia $\alpha = n/m < 1$ il fattore di carico della tabella hash.

**Ricerca senza successo**

***Teorema.*** In una tabella hash in cui le collisioni sono risolte tramite indirizzamento aperto, nell'ipotesi di uniformità semplice, il numero atteso di accessi effettuati da una ricerca senza successo è al più $1/(1-\alpha)$.

***Dimostrazione.*** In una ricerca senza successo tutte le caselle esplorate tranne l'ultima contengono una chiave diversa da quella cercata e l'ultima è vuota.

Sia $p_i = Pr\{\text{esattamente } i \text{ accessi a caselle occupate}\} \leq n$ per $i = 0, 1, \ldots, i$.

Per $i > n$ si ha sempre $p_i = 0$, poiché al massimo $n$ caselle possono essere occupate; il numero atteso di accessi è quindi:

$$1+\sum_{i=0}^{\infty} ip_i$$

Ricordando che per una variabile aleatoria $X$ con valori nei naturali $\mathbb{N} = \{1, 2, 3, \ldots\}$ esiste una semplice formula per il suo valore atteso:

$$E(X)=\sum_{i=0}^{\infty} iPr\{X=i\} =\sum_{i=0}^{\infty} i(Pr\{X \geq i\} - Pr\{X \geq i+1\})=\sum_{i=1}^{\infty} Pr\{X \geq i\}$$

e definendo:

$q_i$ = *Pr {almeno i accessi a caselle occupate} per* $i = 1, \ldots, i \leq n$

possiamo usare la formula di cui sopra per scrivere la seguente identità:

$$\sum_{i=0}^{\infty} ip_i = \sum_{i=1}^{\infty} q_i$$

Ora, la probabilità $q_1$ che il primo accesso avvenga su una casella occupata è $q_1 = n/m$. Il secondo accesso, se necessario, avviene su una delle rimanenti $m - 1$ caselle, $n - 1$ delle quali sono occupate. Effettuiamo un secondo accesso solo se la prima casella è occupata, e dunque la probabilità che il secondo accesso avvenga anch'esso su una casella occupata è:

$$q_2 = (\tfrac{n}{m})(\tfrac{n-1}{m-1})$$

In generale, l'$i$-esimo accesso si effettua solo se i precedenti $(i - 1)$ accessi sono avvenuti tutti su caselle occupate, ed avviene su una delle $(m - i + 1)$ caselle rimanenti, $(n - i + 1)$ delle quali sono occupate.

Dunque:

$$q_i = (\tfrac{n}{m})(\tfrac{n-1}{m-1})(\tfrac{n-2}{m-2}) \ldots (\tfrac{n-i+1}{m-i+1}) \leq (\tfrac{n}{m})^i = \alpha^i$$

poiché $(n - j)/(m - j)$ è minore o uguale di $n/m$ se $n \leq m$ e $j \geq 0$.

Nell'assunzione che sia $\alpha < 1$ possiamo quindi scrivere che il numero atteso di accessi senza successo è:

$$1 + \sum_{i=0}^{\infty} ip_i = 1 + \sum_{i=1}^{\infty} q_i \leq 1 + \sum_{i=1}^{\infty} \alpha^i = 1 + \alpha + \alpha^2 + \alpha^3 + \ldots = \tfrac{1}{1-\alpha}$$

CVD

L'interpretazione intuitiva di questo risultato è che un accesso è sempre necessario, poi con probabilità $\alpha$ ne servono due, con probabilità $\alpha^2$ ne serve un terzo, e così via.

**Inserimento**

Per inserire un elemento si deve scandire la tabella fino a trovare una casella vuota, quindi il numero di accessi è identico a quello della ricerca senza successo:

***Teorema.*** In una tabella hash in cui le collisioni sono risolte tramite indirizzamento aperto, nell'ipotesi di uniformità semplice, il numero atteso di accessi effettuati da un inserimento è al più $1/(1-\alpha)$.

**Ricerca con successo**

***Teorema.*** In una tabella hash in cui le collisioni sono risolte tramite indirizzamento aperto, nell'ipotesi di uniformità semplice, il numero di accessi atteso per una ricerca con successo (ossia la ricerca di un elemento presente nella tabella) è:

$$\frac{1}{\alpha} log_e \frac{1}{1-\alpha} + \frac{1}{\alpha}$$

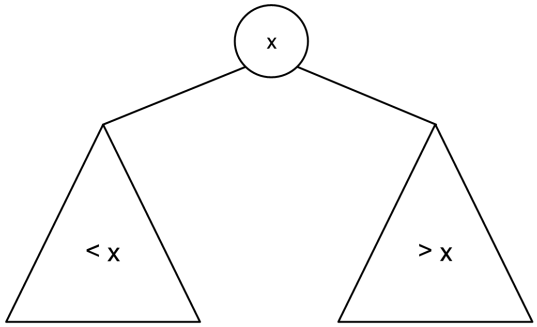
Non dimostriamo questo risultato, ma diamo alcuni esempi per far capire quanto bene si comporta l'indirizzamento aperto nell'ipotesi di uniformità semplice: con una tabella piena al 50% il numero di accessi atteso è meno di 3,387; con una tabella piena al 90% il numero di accessi atteso è meno di 3,67; tutto questo indipendentemente dalle dimensioni della tabella!

I dizionari del Python sono estremamente efficienti **in media**, ma le operazioni su di essi hanno i costi appena descritti, che in generale NON sono costanti nel **caso peggiore**.

## 8.3 Alberi binari di ricerca

Gli ***alberi binari di ricerca*** (***ABR***) sono alberi binari nei quali vengono mantenute le seguenti proprietà:

- ogni nodo dell'albero contiene una chiave;
- il valore della chiave contenuta in ogni nodo è maggiore al valore della chiave contenuta in ciascun nodo del suo sottalbero sinistro (se esso esiste);
- il valore della chiave contenuta in ogni nodo è minore del valore della chiave contenuta in ciascun nodo del suo sottalbero destro (se esso esiste).




Gli ABR sono strutture dati che supportano tutte le operazioni già definite in relazione agli insiemi dinamici più alcune altre.

Operazioni di interrogazione:

- Search(T, k): vogliamo sapere se l'elemento con chiave di valore *k* è presente in *T*;

- Minimum(T): vogliamo recuperare l'elemento di minimo valore presente in *T*;
- Maximum(T): vogliamo recuperare l'elemento di massimo valore presente in *T*;
- Predecessor(T, p): vogliamo recuperare l'elemento presente in *T* con valore precedente al valore nel nodo puntato da *p*;
- Successor(T, p): vogliamo recuperare l'elemento presente in *T* con valore successivo al valore nel nodo puntato da *p*.

Operazioni di manipolazione:

- Insert(T, k): vogliamo inserire un elemento di valore *k* in *T*;
- Delete(T, p): vogliamo eliminare da *T* l'elemento puntato da *p*.

Per inciso, un ABR può essere usato sia come dizionario che come coda di priorità: il minimo è sempre nel nodo più a sinistra, il massimo in quello più a destra. Si noti che il nodo più a sinistra non necessariamente è una foglia: può anche essere il nodo più a sinistra che non ha un figlio sinistro; analoga considerazione per il nodo più a destra.
Per elencare tutte le chiavi in ordine crescente basta compiere una visita inordine. Dunque un ABR può anche essere visto come una struttura dati su cui eseguire un algoritmo di ordinamento, costituito di due fasi:

1. inserimento di tutte le *n* chiavi da ordinare in un ABR, inizialmente vuoto;

2. visita inordine dell'ABR appena costruito.

Il costo computazionale di tale algoritmo sarà:

$$T(\text{costruzione ABR}) + T(\text{visita}) = T(\text{costruzione ABR}) + \Theta(n)$$

Più avanti determineremo il tempo necessario alla costruzione di un ABR.

### 8.3.1 Ricerca in un albero binario di ricerca

Il procedimento di ricerca su un ABR è concettualmente simile alla ricerca binaria ed è molto semplice: si esegue una discesa dalla radice che viene guidata dai valori memorizzati nei nodi che si incontrano lungo il cammino. Quando si trova in un nodo il valore ricercato si termina con successo restituendo il puntatore al nodo. Se invece si arriva ad una foglia che non lo contiene si termina senza successo restituendo *NULL*. Si osservi che ogni qual volta si scende verso destra (sinistra), automaticamente si decide di escludere l'intero sottoalbero di sinistra (destra), e quindi lo spazio di ricerca si riduce via via che si discende verso le foglie.

```
def ABR_search_ricorsiva (p, k):
  if ((p == None) or (p.key == k)):
    return p
  if (k < p.key):
    return ABR_search (p.left, k)
  else:
    return ABR_search (p.right, k)
```

Per quanto riguarda il costo computazionale della funzione di ricerca, intuitivamente, dato che la ricerca percorre un singolo cammino dalla

radice ad una foglia, ci aspettiamo che esso sia legato all'altezza $h$ dell'albero.

Più precisamente, la funzione ricorsiva compie un numero costante di operazioni sulla radice dell'albero e poi richiama se stessa sulla radice di uno dei due sottoalberi, che al massimo ha altezza $h - 1$; possiamo quindi scrivere:

$$T(h) = \Theta(1) + T(h - 1) = \Theta(h)$$

nel caso peggiore, che è quello della ricerca senza successo che termini in una foglia a distanza massima dalla radice. Nel caso base, che si ha quando $h=0$ (caso in cui la chiave cercata sia nella radice), il tempo di esecuzione è costante, cioè $T(0) = \Theta(1)$.

La ricerca su un ABR può essere espressa in modo semplice anche in forma iterativa:

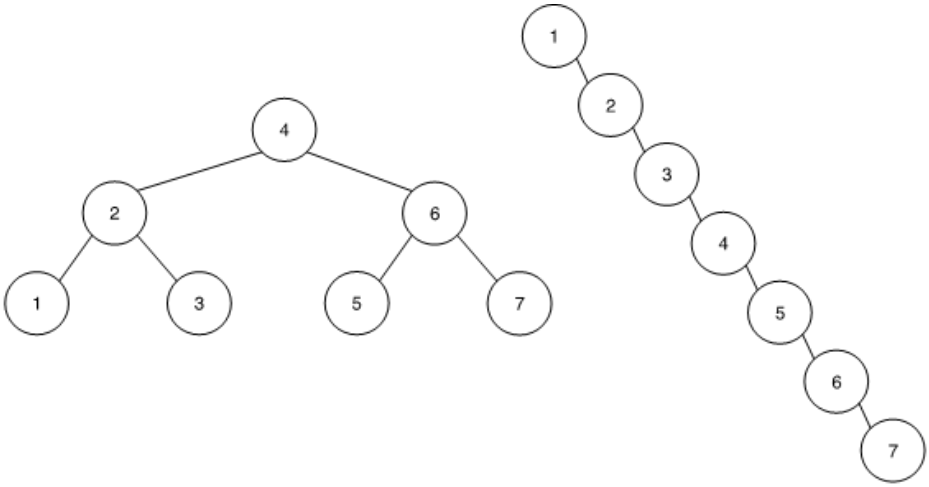
```
def ABR_search_iterativa (p, k):
  while ((p != None) and (p.key != k)):
    if (k < p.key):
      p = p.left
    else:
      p = p.right
  return p
```

In questo caso, il costo computazionale dipende dal numero di volte che viene eseguito il ciclo while. Nel caso peggiore (ricerca senza successo che termini in una foglia a distanza massima dalla radice) esso viene eseguito un numero di volte pari all'altezza dell'albero e quindi il costo è, ovviamente come per la versione ricorsiva, $O(h)$.

Considerando che la ricerca su un ABR ricorda molto da vicino la

ricerca binaria già descritta nel par. 3.2, per la quale il costo computazionale è $O(\log n)$, sorge spontanea una domanda: come mai non riusciamo a garantire un costo logaritmico anche per la ricerca su un ABR?

La ragione risiede nel fatto che l'ABR non include fra le sue proprietà alcunché in relazione alla sua altezza. Ad esempio, entrambi gli ABR illustrati nella figura seguente sono legittimi:

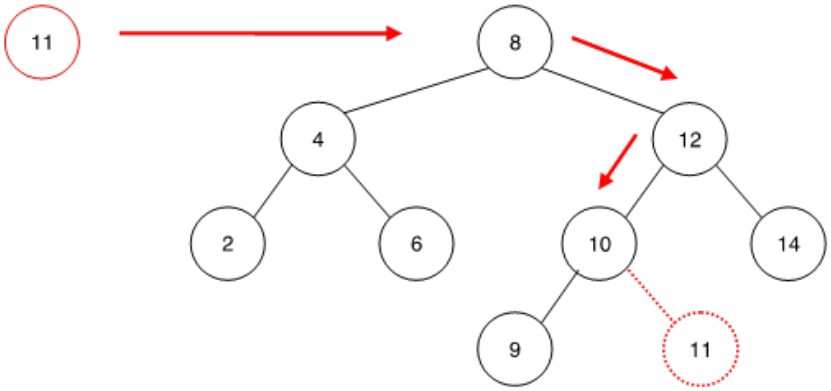



Chiaramente, nell'ABR di sinistra l'altezza è $\Theta(\log n)$ mentre in quello di destra (detto ***albero degenere***) è $\Theta(n)$; la stessa differenza fra tali due situazioni quindi si manifesta nel costo della ricerca nel caso pessimo.

Questo ci fa capire che, se vogliamo garantire che il costo computazionale della ricerca su ABR sia limitata superiormente da un logaritmo, dovremo preoccuparci di mettere in campo qualche tecnica che ci permetta di tenere sotto controllo la crescita dell'altezza: questo tipo di tecniche prendono il nome di tecniche di ***bilanciamento in altezza*** e saranno descritte nel seguito.

### 8.3.2 Inserimento in un albero binario di ricerca

Anche il procedimento di inserimento su un ABR è molto semplice: si esegue una discesa che, come nel caso della ricerca, viene guidata dai valori memorizzati nei nodi che si incontrano lungo il cammino. Quando si arriva al punto di voler proseguire la discesa verso un puntatore vuoto (NULL) allora, in quella posizione, si aggiunge un nuovo nodo contenente il valore da inserire. Il padre di tale nuovo nodo potrebbe essere una foglia (entrambi i suoi figli sono NULL), ma, più in generale, è un nodo a cui manca il figlio corrispondente alla direzione presa.




Per convenienza introduciamo una modifica alla struttura dati utilizzata per rappresentare un nodo che, sebbene non strettamente necessaria per l'inserzione, lo sarà per poter gestire l'eliminazione di un nodo.

In ogni nodo, oltre ai due puntatori ai figli sinistro e destro, aggiungiamo un campo in cui viene memorizzato il puntatore al padre (*None* nella radice dell'albero).

Chiameremo *parent* tale campo.

Supponiamo, come già fatto in altri casi analoghi, che sia disponibile un nodo già costruito puntato da z e contenente nel campo *key* il valore della chiave e nei campi *parent, left* e *right* il valore *NULL*.

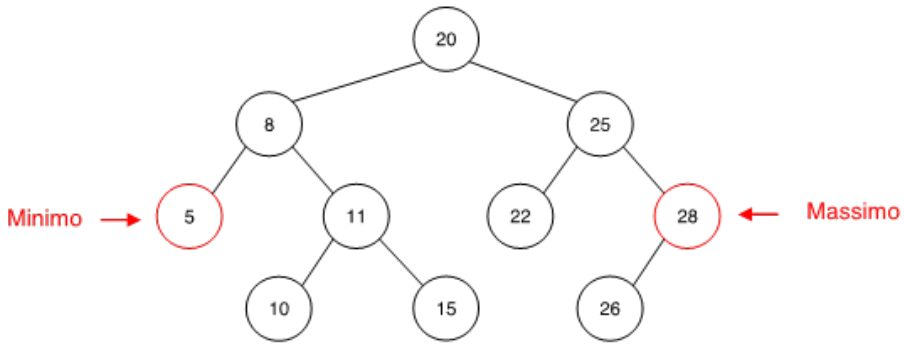
Così come nel caso della ricerca, anche qui il costo computazionale dipende dal numero di volte in cui viene eseguito il ciclo while. Esso viene eseguito un numero di volte che è al massimo pari all'altezza dell'albero e quindi il costo è *O(h)*.

Lo pseudocodice (iterativo) è il seguente:

```
def ABR_insert (p, z):
  x, y = p, None
  while (x != None)  # discesa alla prima pos. disponibile
    y = x              # y punta sempre al padre di x
    if z.key < x.key
      x = x.left
    else
      x = x.right
  z.parent = y        # colleghiamo al padre il nodo da inserire
  if (y == None):     # se albero inizialmente vuoto
    p = z
  else:
    if (z.key < y.key):
      y.left = z
    else:
      y.right = z
  return p            # restituiamo p, che potrebbe essere cambiato
```

### 8.3.3 Ricerca di minimo, massimo, predecessore e successore

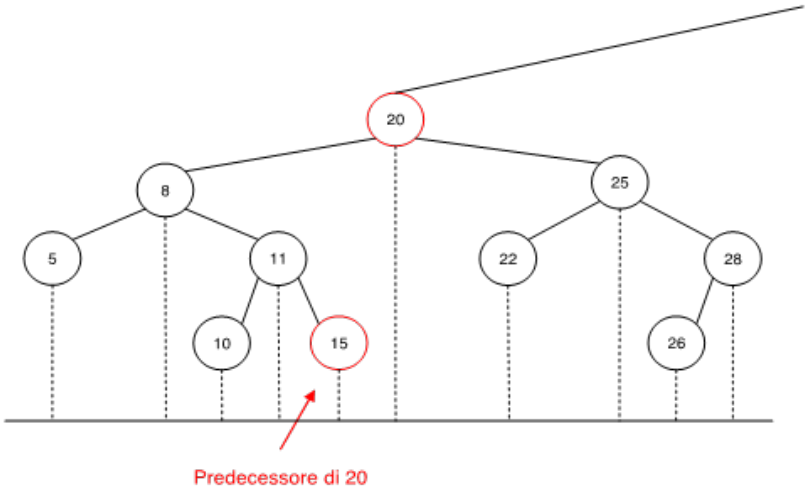
Consideriamo il problema di reperire il minimo e il massimo dei valori contenuti in un ABR. La soluzione è semplice: poiché, come già detto, il minimo (massimo) si trova nel nodo più a sinistra, per trovarlo si scende sempre a sinistra (destra) a partire dalla radice. Ci si ferma quando si arriva a un nodo che non ha figlio sinistro (destro): quel nodo contiene il minimo (massimo).
Si lascia lo pseudocodice delle due funzioni come esercizio.



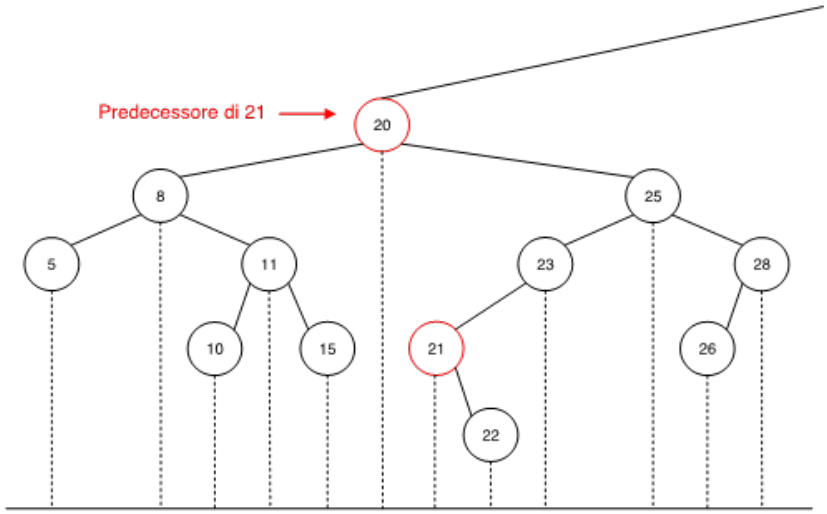

Entrambe queste operazioni richiedono di scendere dalla radice lungo un singolo cammino, e quindi hanno costo computazionale limitato superiormente dall'altezza dell'albero: $O(h)$.
Più interessante è il problema di determinare il predecessore o il successore di una chiave $k$ contenuta nell'ABR. Ricordiamo che per predecessore si intende il nodo dell'albero contenente la chiave che precederebbe immediatamente $k$ se le chiavi fossero ordinate; il successore è il nodo che contiene la chiave che seguirebbe immediatamente $k$ se le chiavi fossero ordinate; pertanto, la ricerca del prede-

cessore (successore) non ha nulla a che fare con relazioni padre-figlio sull'albero.

Concentriamoci dapprima sulla ricerca del predecessore.




**Caso 1:** se la chiave *k* è contenuta in un nodo che ha un sottoalbero sinistro (ad es. *k=20* nella figura della pagina precedente), allora il predecessore di *k* è il massimo delle chiavi contenute nel sottoalbero sinistro, e quindi il nodo che la contiene è quello più a destra del sottoalbero sinistro.




**Caso 2:** viceversa, se il nodo che contiene *k* non ha sottoalbero sinistro (ad es. *k=21* nella figura precedente) vuol dire che esso è il nodo più a sinistra di un certo sottoalbero, e quindi il minimo di tale sottoalbero. Per trovare il predecessore di *k* bisogna quindi risalire alla radice di quel sottoalbero, il che significa salire a destra finchè è possibile. Una volta giunti nella radice del sottoalbero, si risale (con un singolo passo di salita a sinistra) a suo padre che è il predecessore di *k*. Si noti che è sempre possibile fare tale ultimo passo, a meno che *k* non sia il minimo.

Una situazione perfettamente simmetrica esiste per il problema di trovare il successore di un nodo. Entrambe tali operazioni richiedono una discesa lungo un singolo cammino a partire dalla radice oppure una singola risalita verso la radice. Per entrambe le funzioni il costo computazionale è limitato superiormente dall'altezza dell'albero: $O(h)$. Si lascia lo pseudocodice come esercizio.

### 8.3.4 Eliminazione in un albero di ricerca

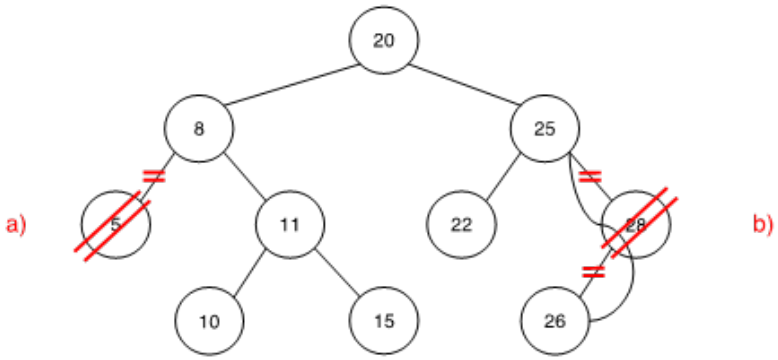
Il problema di eliminare una chiave contenuta in un ABR è il più complicato, poiché se la chiave da eliminare è contenuta in un nodo che ha entrambi i figli è necessario riaggiustare l'albero dopo l'eliminazione per evitare che si disconnetta, il che produrrebbe di conseguenza due ABR separati.
Riaggiustare l'albero significa trovare un nodo da collocare al posto del nodo che va eliminato, così da mantenere l'albero connesso. Naturalmente, poiché si tratta di un ABR, il nodo da mettere al posto di quello eliminato deve essere tale da garantire il mantenimento della proprietà fondamentale degli ABR, e quindi può essere solamente il predecessore o il successore del nodo da eliminare.
Qundi, per eliminare un nodo in un ABR:

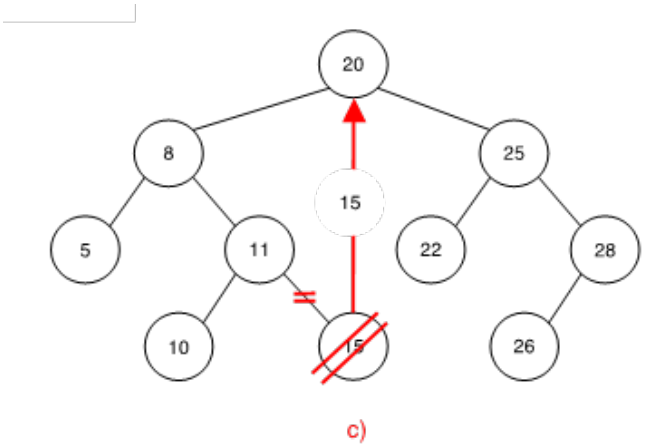
a) se il nodo è una foglia lo si elimina, ponendo a *None* l'opportuno campo nel nodo padre;

b) se il nodo ha un solo figlio lo si "cortocircuita", cioè si collegano direttamente fra loro suo padre e il suo unico figlio, indipendentemente che sia destro o sinistro;

c) se il nodo ha entrambi i figli lo si sostituisce col predecessore (o col successore), che va quindi tolto (ossia eliminato) dalla sua posizione originale. Si noti che, per trovare il predecessore (o il successore) del nodo da cancellare, si userà sicuramente il caso 1 dell'algoritmo di ricerca del predecessore visto precedentemente, infatti il nodo da cancellare in questo caso ha due figli, e quindi non può essere che gli manchi il figlio si-

nistro. D'altra parte, il predecessore (successore) non può avere entrambi i figli poiché è il minimo (massimo) del suo sottoalbero. Di conseguenza, la sua eliminazione si effettua con le modalità previste per i casi a) e b) precedenti.




Lo pseudocodice presuppone l'esistenza della funzione ABR_successor(p) che individua il successore del nodo puntato da p ed è il seguente. Il nodo da eliminare è puntato dal puntatore z, che è supposto essere diverso da None.

```
def ABR_delete (p, z):
```

```
  if((z.left==None) or (z.right==None)):# dopo queste
    y = z                               # istruzioni y
  else:                                 # punta al nodo da
    y = ABR_successor(z)                # eliminare fisicamente
#-------------------------------------------------------------------------------
  if (y.left != None):                  # y non può avere
    x = y.left                          # due figli
  else:                                 # x punterà al figlio
    x = y.right                         # oppure sarà NULL
#-------------------------------------------------------------------------------
  if (x != None):                       # se y ha il figlio
    x.parent = y.parent                 # cortocircuitiamo y
#-------------------------------------------------------------------------------
  if (y.parent == None):                # y è la radice?
    p = x                               # si: x è nuova radice
  else:                                 # no:
    if (y == y.parent.left):            # y è un figlio sinistro
      y.parent.left = x                 # cortocircuitiamo y
    else:                               # y è un figlio destro
      y.parent.right = x                # cortocircuitiamo y
#-------------------------------------------------------------------------------
  if (y != z):                          # siamo nel caso c)
    z.key = y.key                       # sovrascriviamo la chiave
#-------------------------------------------------------------------------------
  return p
```

Si noti che sovrascrivere la chiave (mediante l'struzione z.key = y.key)

è accettabile solo se non vi sono dati satellite, altrimenti diviene necessario aggiustare i puntatori per fare in modo che il nodo puntato da y sostituisca nell'albero a tutti gli effetti il nodo puntato da z.

La funzione effettua un numero costante di operazioni e una chiamata alla ricerca del successore, quindi la complessità è $\Theta(1) + \Theta(h) = \Theta(h)$, dove $h$ è l'altezza dell'ABR.

## 8.4 Alberi Red-Black

Come abbiamo visto, un ABR di altezza $h$ contenente $n$ nodi supporta le operazioni fondamentali dei dizionari in tempo $O(h)$, ma purtroppo non si può escludere che $h$ sia $O(n)$, con conseguente degrado delle prestazioni.

Viceversa, tutte le operazioni restano efficienti se si riesce a garantire che l'altezza dell'albero sia piccola, in particolare sia $O(\log n)$.

Per raggiungere tale scopo esistono varie tecniche, dette di ***bilanciamento***, tutte basate sull'idea di riorganizzare la struttura dell'albero se essa, a seguito di un'operazione di inserimento o di eliminazione di un nodo, viola determinati requisiti. In particolare, il requisito da controllare è che, per ciascun nodo dell'albero, l'altezza dei suoi due sottoalberi non sia "troppo differente". Ciò che rende non banali queste tecniche è che si vuole aggiungere agli alberi di ricerca una proprietà (il bilanciamento) senza peggiorare il costo computazionale delle operazioni.

Illustreremo ora una di tali tecniche, quella degli ***RB-alberi*** (***alberi Red-Black***).

**Definizione.** Un *RB-albero* è un ABR i cui nodi hanno un campo aggiuntivo, il ***colore***, che può essere solo rosso o nero. Alla struttura si aggiungono foglie fittizie (che non contengono chiavi) per fare in modo che tutti i nodi "veri" dell'albero abbiano ***esattamente due figli***.

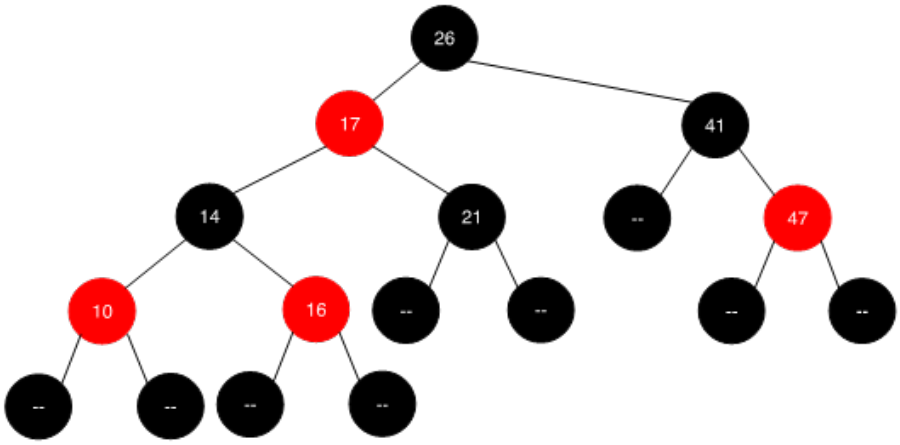
Un RB-albero soddisfa, inoltre, le seguenti proprietà aggiuntive:

1. ciascun nodo è rosso o nero;
2. ciascuna foglia fittizia è nera;
3. se un nodo è rosso i suoi figli sono entrambi neri;
4. ogni cammino da un nodo a ciascuna delle foglie del suo sottoalbero contiene lo stesso numero di nodi neri.

Col termine ***b-altezza di x*** (***black height*** di *x*, ***bh(x)***) si indica il numero di nodi neri sui cammini dal nodo x (non incluso) alle foglie sue discendenti (è uguale per tutti i cammini, proprietà 4).

La *b-altezza* di un RB-albero è la b-altezza della sua radice.

La figura seguente illustra un esempio di RB-albero.




Dalla combinazione delle proprietà 3. e 4. discende un fatto interessante: nessun cammino dalla radice ad una foglia può essere lungo più del doppio di un cammino dalla radice ad una qualunque altra. Questo perché:

- il numero di nodi neri è il medesimo lungo tutti i cammini (proprietà 4);
- il numero di nodi rossi lungo un cammino non può essere maggiore del numero di nodi neri lungo lo stesso cammino (proprietà 3)

e quindi un cammino che contiene il massimo numero possibile di nodi rossi non può essere lungo più del doppio di un cammino composto solo di nodi neri.

Questa proprietà, che può sembrare piuttosto debole, è sufficiente a dimostrare che l'altezza di un RB-albero di $n$ nodi è $O(\log n)$, come mostra il seguente teorema.

**Teorema.** *Un RB-albero con n nodi interni ha altezza h al più 2 log(n+1).*

**Dimostrazione.** Proviamo prima il seguente risultato:

*Il sottoalbero radicato in un qualsiasi nodo x contiene almeno $2^{bh(x)}-1$ nodi interni.*

Ragioniamo per induzione sulla distanza di $x$ dalle foglie, sia essa $d(x)$.

Se $d(x)=0$, allora $x$ è una foglia ed il suo sottoalbero contiene almeno $2^{bh(x)}-1=2^0-1=0$ nodi interni.

Sia ora $x$ un nodo interno con $d(x)>0$. I suoi figli hanno b-altezza pari o a $bh(x)$ o a $bh(x)-1$, a seconda che siano rossi o neri.

Su tali figli si può applicare l'ipotesi induttiva, poiché hanno distanza dalle foglie pari a $d(x)-1$, così il sottoalbero di ciascuno di essi contiene almeno $2^{bh(x)-1}-1$ nodi interni. Quindi, i nodi interni del sottoalbero radicato in $x$ sono almeno $2(2^{bh(x)-1}-1)+1=2^{bh(x)}-1$. Abbiamo così dimostrato l'asserto di cui avevamo bisogno per concludere la dimostrazione del teorema.

Sia ora $h$ l'altezza dell'RB-albero ed $r$ la sua radice. Sappiamo già che $bh(r) \geq h/2$. Dal risultato precedente, il numero di nodi interni dell'RB-albero, pari al numero di nodi interni nel sottoalbero radicato in $r$, è $n \geq 2^{bh(r)}-1 \geq 2^{h/2}-1$ da cui $n+1 \geq 2^{h/2}$, cioè $2\ log(n+1) \geq h$. **CVD**

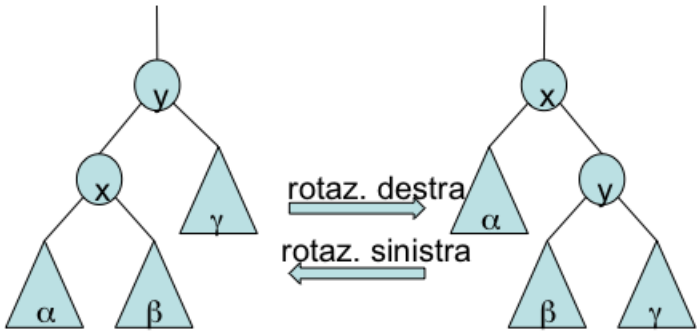
Il teorema precedente garantisce che le operazioni di ricerca di una chiave, del massimo o del minimo, del predecessore o del successore, siano tutte eseguite con un costo computazionale $O(log\ n)$.

Discorso a parte va fatto invece per gli inserimenti e le cancellazioni.

Infatti, l'esigenza di mantenere le proprietà di RB-albero implica che, dopo un inserimento o una cancellazione, la struttura dell'albero possa dover essere riaggiustata, in termini di:

- colori assegnati ai nodi;
- struttura dei puntatori (ossia, collocazione dei nodi nell'albero).

A tal fine sono definite apposite operazioni, dette ***rotazioni***, che permettono di ripristinare in tempo *O(log n)* le proprietà del RB-albero dopo un inserimento o un'eliminazione.
Le rotazioni, che possono essere destre o sinistre, sono operazioni locali che non modificano l'ordinamento delle chiavi secondo la visita in-ordine, e possono essere schematizzate come riportato nella figura seguente.




Possiamo osservare che effettuare ciascuna rotazione implica un costo costante poiché coinvolge un numero costante di puntatori, essendo un'operazione locale. Ciò si evidenzia nel seguente pseudocodice, nel quale p punta alla radice dell'albero ed x al nodo intorno al quale avviene la rotazione.

```
def RBA_rotaz_sinistra (p, x):
```

```
y = x.right
x.right = y.left
if y.left != None:
   y.left.parent = x
y.parent = x.parent
if x.parent==None:
   p = y
else:
   if x==x.parent.left:
      x.parent.left = y
   else:
      x.parent.right = y
y.left = x
x.parent = y
```

Lo pseudocodice della rotazione a destra è analogo.

Possiamo ora descrivere l'algoritmo di inserimento di una chiave in un albero Red & Black.
Si utilizza l'algoritmo di inserimento in un ABR, colorando il nodo sempre di rosso.
Così facendo, le regole 1. (ciascun nodo è rosso o nero) e 2. (le foglie fittizie sono nere) si mantengono sempre vere nella struttura; anche la regola 4. (ogni cammino da un nodo a ciascuna foglia ha lo stesso numero di nodi neri) rimane vera dopo un inserimento, visto che il nuovo nodo è sempre colorato di rosso. Dunque, l'unica regola che può essere infranta è la 3. (entrambi i figli di un nodo rosso sono

neri), infatti il nuovo nodo (rosso) potrebbe essere inserito come figlio di un nodo rosso.

Dobbiamo dunque procedere ad aggiustare la struttura solo in questo caso. Si presentano 3 possibili eventualità da gestire in modo diverso:

Caso 1. Il nodo inserito è figlio di un nodo rosso e lo zio è rosso.

Caso 2. Il nodo inserito è figlio destro di un nodo rosso e lo zio è nero.

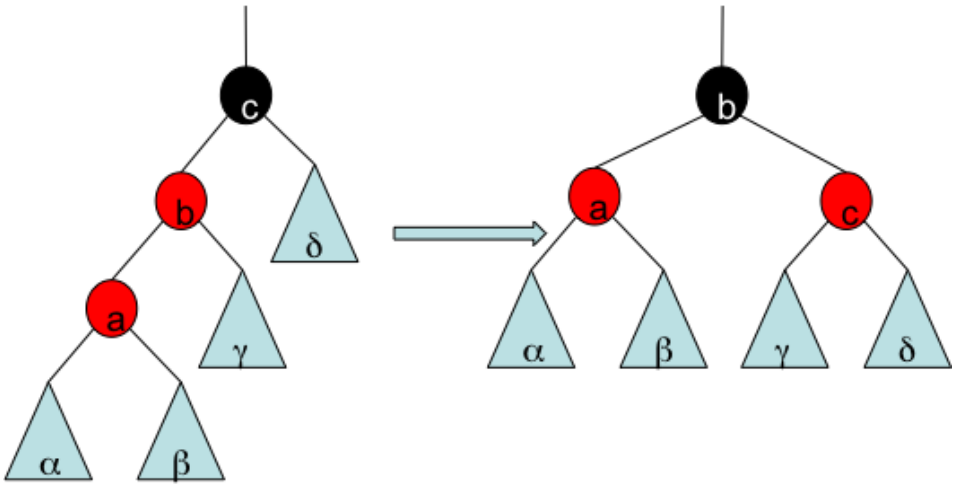
Caso 3. Il nodo inserito è figlio sinistro di un nodo rosso e lo zio è nero.

Si osservi che in tutti e tre i casi, il nonno del nuovo nodo è nero, poiché l'unico punto dell'albero in cui la regola 3. non è rispettata è quello in cui il nodo è stato inserito.

Nel Caso 1. si cambiano i colori di alcuni nodi: detto $x$ il nuovo nodo appena inserito, si colorano di nero il padre di $x$ e lo zio di $x$, di rosso il nonno di $x$. A questo punto, o la violazione è stata eliminata (in tal caso l'operazione di inserimento si conclude), o è stata portata più in alto, e si ripresenta di nuovo il Caso 1. a partire dal nonno di $x$, o infine la violazione è stata portata più in alto ma si presenta il Caso 2. o il Caso 3..

Se si presenta il Caso 2., si effettua una rotazione a sinistra imperniata sul padre di $x$, e questo conduce sempre al Caso 3.

Infine, se si presenta il Caso 3., si effettua una rotazione e si cambiano alcuni colori secondo la figura seguente, il che garantisce sempre la soluzione della violazione.

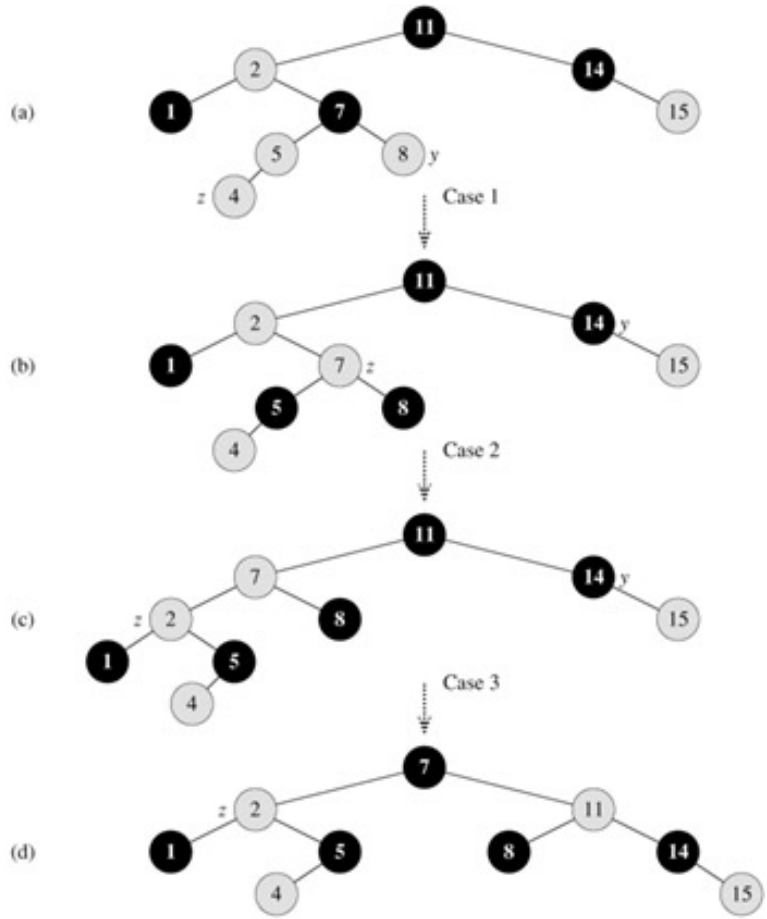



Ne consegue che, durante un inserimento, è possibile entrare più volte nel Caso 1., eventualmente nel Caso 2., e si conclude sempre con il Caso 3.

Un esempio di funzionamento dell'operazione di inserimento (viene inserito la chiave 4) è riportato in figura a pagina seguente.

Poiché ogni volta che si presenta un nuovo caso da risolvere, la violazione si è spostata più in alto, avremo al più *O(log n)* iterazioni di costo costante, provando così che in un albero Red & Black si può inserire una nuova chiave in tempo *O(log n).*

Lo stesso si può dire dell'operazione di cancellazione, che però decidiamo di omettere qui.


(a)
11
2
14
1
7
15
5
8
y
z
4
Case 1
(b)
11
2
14
y
1
7
15
z
5
8
4
Case 2
(c)
11
7
14
y
z
2
8
15
1
5
4
Case 3
(d)
7
z
2
11
1
5
8
14
4
15