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:

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

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.

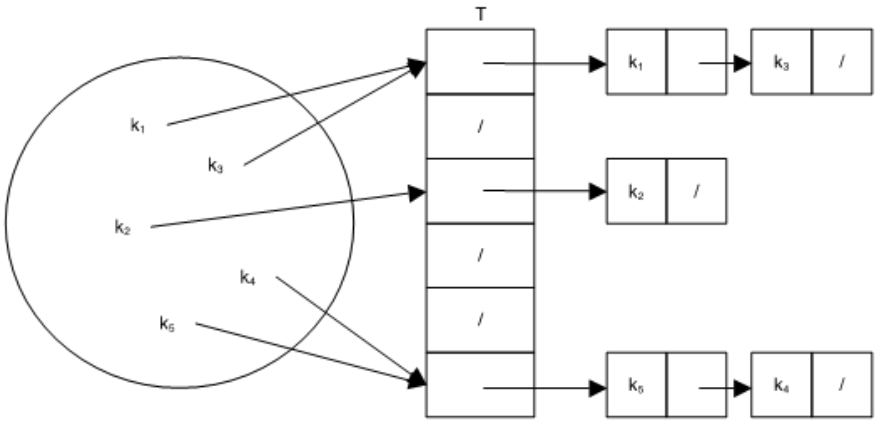
Ad ogni modo, per quanto bene sia progettata la funzione hash, è impossibile evitare del tutto le collisioni perché se $|U| > m$ è inevitabile che esistano chiavi diverse che producono una collisione.
Non potendole evitare, bisogna decidere come risolvere le collisioni.

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 α, 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:

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

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:

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

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:

$$q_2 = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right)$$

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:

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

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:

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

CVD

L'interpretazione intuitiva di questo risultato è che un accesso è sempre necessario, poi con probabilità α ne servono due, con probabilità α^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:

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

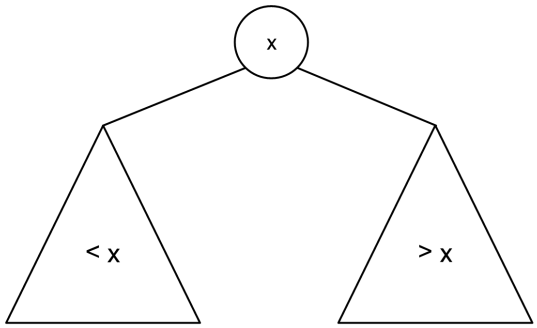
Non dimostriamo questo risultato, ma diamo alcuni esempi per far capire quanto bene si comporta l'indirizzamento aperto nell'ipotesi di uniformità semplice: con una tabella piena al 50% il numero di accessi atteso è meno di 3,387; con una tabella piena al 90% il numero di accessi atteso è meno di 3,67; 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 e, nel **caso peggiore**, arrivano anche ad $O(n)$.

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 costo della 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:

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

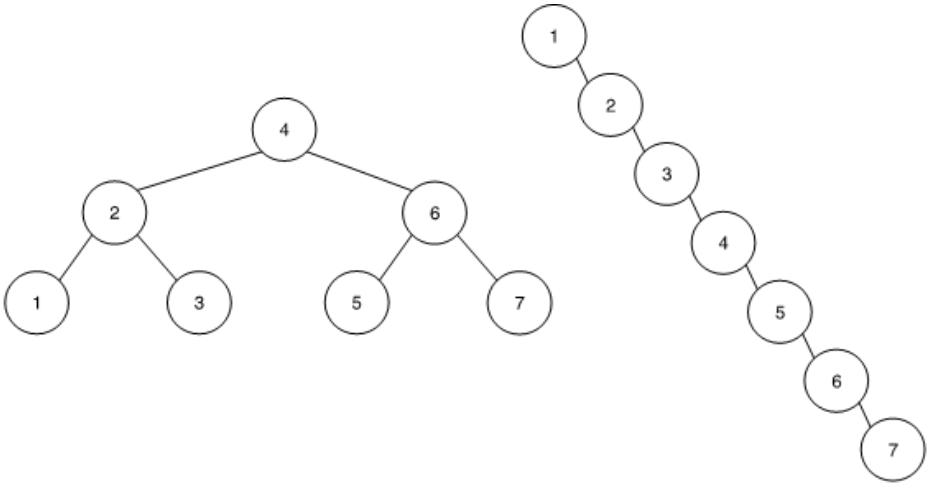
nel caso peggiore, che è quello della ricerca senza successo che termini in una foglia a distanza massima dalla radice. Nel caso base, che si ha quando $h=0$ (caso in cui la chiave cercata sia nella radice), il tempo di esecuzione è costante, cioè $T(0) = \Theta(1)$.

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(logn)*, 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:

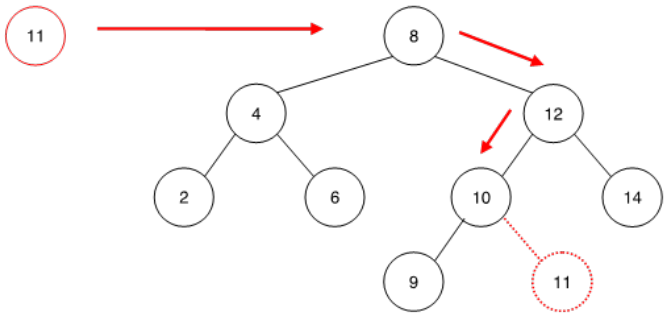


Chiaramente, nell'ABR di sinistra l'altezza è $\Theta(log\ n)$ mentre in quello di destra (detto ***albero degenere***) è $\Theta(n)$; la stessa differenza fra tali due situazioni quindi si manifesta nel costo della ricerca nel caso pessimo.
Questo ci fa capire che, se vogliamo garantire che il costo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

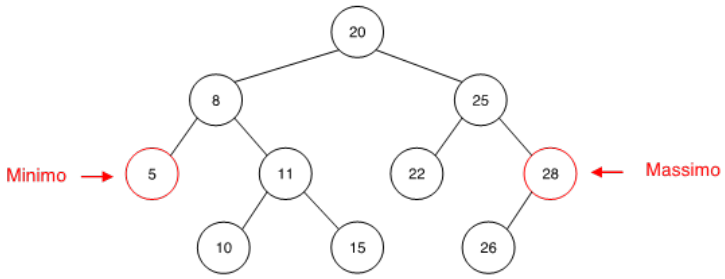
Supponiamo, come già fatto in altri casi analoghi, che sia disponibile un nodo già costruito puntato da z e contenente nel campo *key* il valore della chiave e nei campi *parent, left* e *right* il valore *NULL*. Così come nel caso della ricerca, anche qui il costo computazionale dipende dal numero di 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 ins.
  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, forse cambiato
```

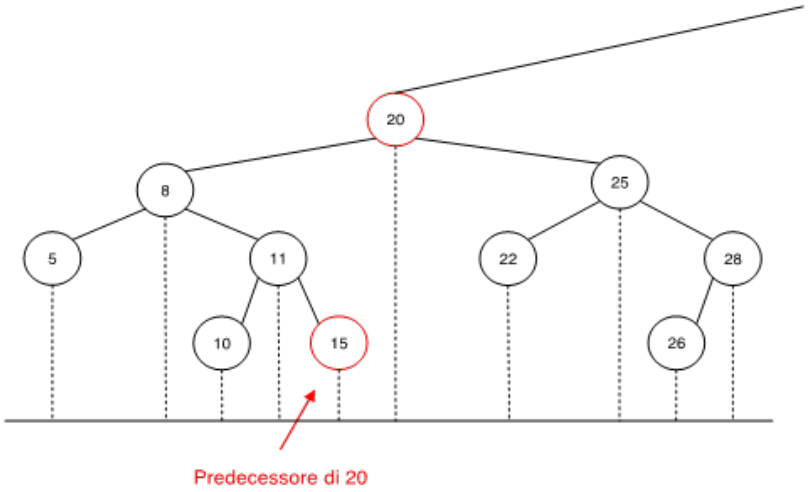
8.3.3 Ricerca di minimo, massimo, predecessore e successore

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.

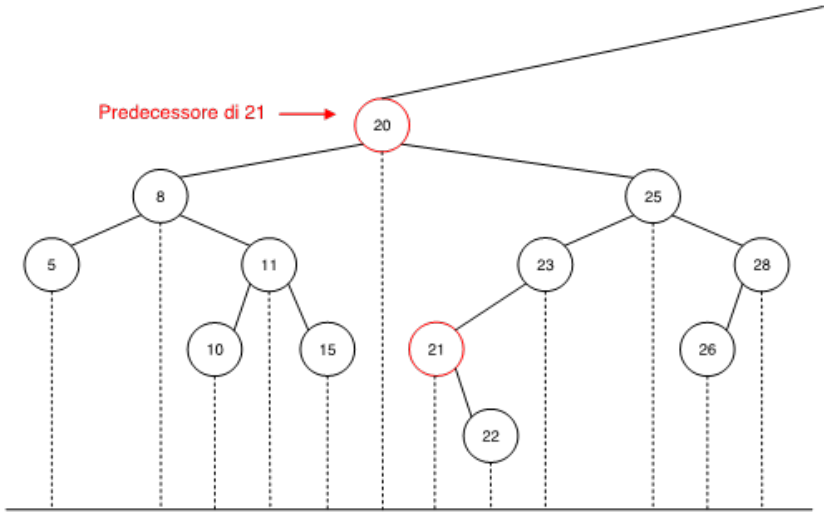


Entrambe queste operazioni richiedono di scendere dalla radice lungo un singolo cammino, e quindi hanno costo computazionale limitato superiormente dall'altezza dell'albero: $O(h)$.
Più interessante è il problema di determinare il predecessore o il successore di una chiave k contenuta nell'ABR. Ricordiamo 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 predecessore (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

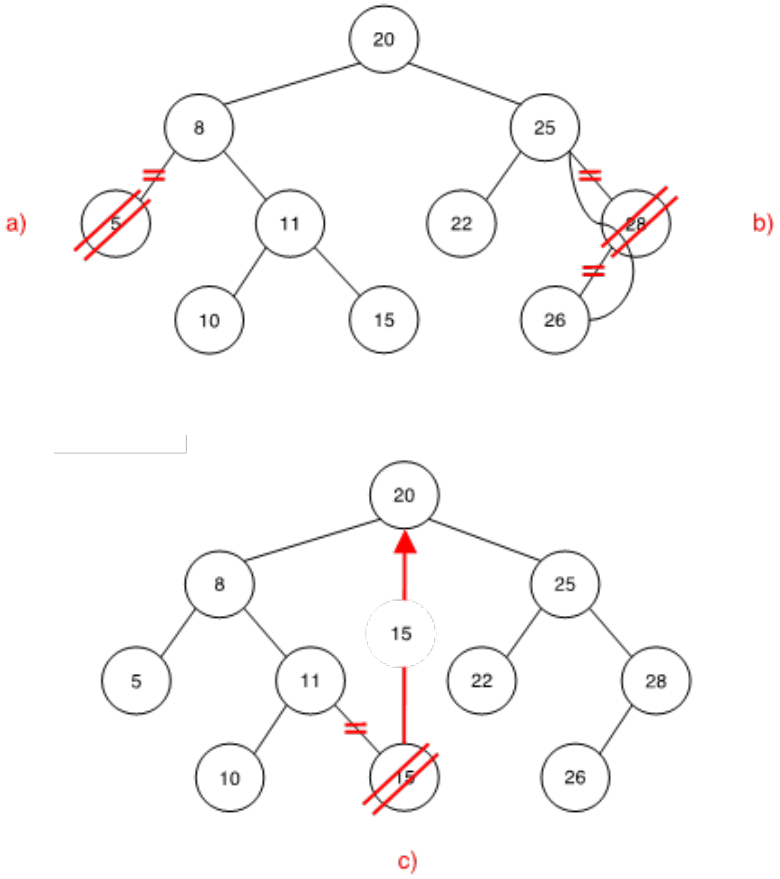
Il problema di eliminare una chiave contenuta in un ABR è il più complicato, poiché se la chiave da eliminare è contenuta in un nodo che ha entrambi i figli è necessario riaggiustare l'albero dopo l'eliminazione per evitare che si disconnetta, il che produrrebbe di conseguenza due ABR separati.
Riaggiustare 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 sinistro. 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 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                       # sovrascrivi 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 AVL

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.
Per motivi di efficienza, è evidente l'importanza di mantenere h il più piccola possibile.

Sappiamo che per un albero con n elementi, l'altezza minima possibile è $\lfloor \log_2 n \rfloor$. Definiamo **bilanciato** un albero per il quale l'altezza h sia $\Theta(\log n)$.
Per garantire l'efficienza delle operazioni, è necessario che l'albero rimanga bilanciato anche dopo le operazioni di inserimento e cancellazione (le uniche che modificano la struttura dell'albero aggiungendo o rimuovendo nodi). Studieremo ora una condizione che garantisce il bilanciamento e che può essere preservata in tempo $O(\log n)$ dopo ciascuna di queste due operazioni.

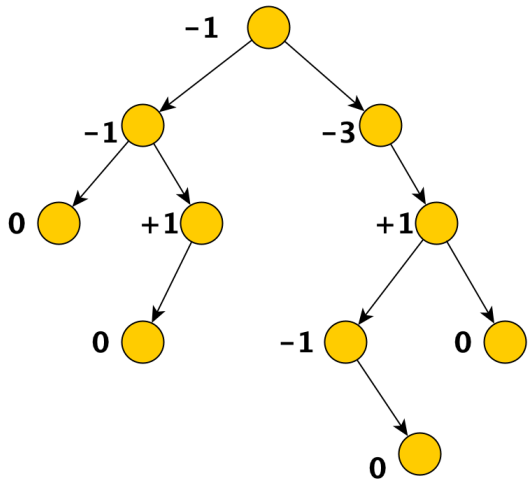
Questa condizione è alla base degli **alberi di ricerca AVL** (o semplicemente **alberi AVL**), la struttura dati che analizzeremo nel dettaglio nel seguito.

8.4.1 Definizione di albero AVL

In un albero binario, il **fattore di bilanciamento** di un nodo è dato dalla differenza tra l'altezza del suo sottoalbero sinistro e l'altezza del suo sottoalbero destro:

bilanciamento(nodo)=altezza(figlio sinistro)-altezza(figlio destro).

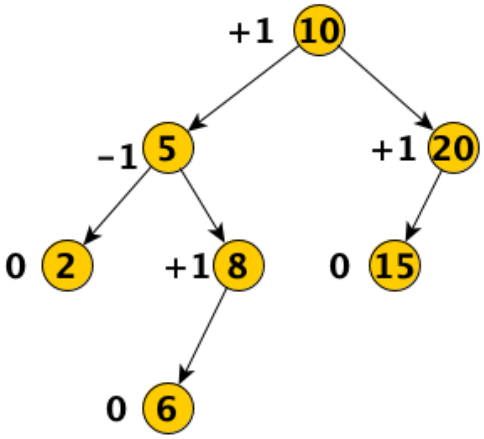
Una foglia, in questo contesto, ha fattore di bilanciamento 0. Nella figura che segue è mostrato come esempio un albero binario in cui, per ogni nodo, è indicato il suo fattore di bilanciamento.



Un **albero AVL** è un albero binario di ricerca in cui il fattore di bilanciamento di ciascuno dei suoi nodi sia compreso tra -1 ed 1. Formalmente, per ogni nodo nell'albero, vale la seguente condizione:

$$|\ \text{altezza(figlio sinistro)} - \text{altezza(figlio destro)}\ | \leq 1.$$

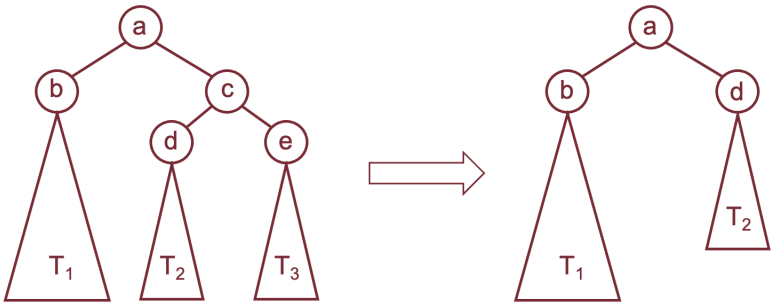
Nella figura che segue è riportato un esempio di albero AVL.



8.4.2 Gli alberi AVL sono alberi bilanciati

Dimostriamo ora che gli alberi AVL sono bilanciati cioè che, per un qualsiasi albero AVL con $n \geq 2$ nodi e altezza h vale la relazione $h \leq 2\log_2 n$. Con un'analisi più approfondita, si può dimostrare un risultato ancora più preciso: per gli alberi AVL, vale infatti che $h \leq 1.44\log_2 n$.

Iniziamo calcolando il numero minimo di nodi, denotato con $N(h)$, che può avere un albero AVL di altezza h. È evidente che $N(0)=1$ e $N(1)=2$. Per $h>2$, possiamo dire che i due sottoalberi della radice dell'albero di altezza h con il numero minimo di nodi non possono avere la stessa altezza. Per convincersene, basta osservare la seguente figura, in cui si mostra come -a partire da un qualunque albero AVL di altezza h in cui la radice ha sottoalberi sinistro e destro di pari altezza- sia sempre possibile ricavare un altro albero di pari altezza ma con un numero inferiore di nodi.



Nella parte sinistra della figura precedente, senza perdere di generalità, assumiamo che il sottoalbero radicato in *b* abbia altezza almeno pari a quella del sottoalbero radicato in *c* ed il sottoalbero radicato in *d* abbia altezza almeno pari a quella del sottoalbero radicato in *e*. Se la radice dell'albero AVL ha fattore di sbilanciamento 0, allora il sottoalbero T_1 radicato in *b* deve avere altezza *h*-1 ed il sottoalbero T_2 radicato in *d* altezza *h*-2. L'albero nella parte destra della figura non solo è un albero binario di ricerca valido ma è anche un AVL (l'unico fattore di sbilanciamento cambiato rispetto all'albero di sinistra è quello della radice che è diventato +1).

Per calcolare il numero minimo di nodi in un albero AVL di altezza *h* dobbiamo, quindi, massimizzare la differenza di altezza tra i due sottoalberi mantenendo le proprietà degli AVL. Ne consegue che l'albero che ha il numero minimo di nodi è quello in cui:

- un sottoalbero ha altezza *h*-1;
- l'altro sottoalbero ha altezza *h* - 2;
- ciò avviene ricorsivamente in tutti i sottoalberi.

Possiamo dunque esprimere il numero minimo di nodi tramite una ricorrenza:

$N(h)=N(h-1)+N(h-2)+1$ per $h \geq 2$

$N(0)=1;\ N(1)=2$.

Per risolverla, applichiamo il metodo iterativo:

$$N(h) = N(h-1) + N(h-2) + 1 = 2N(h-2) + N(h-3) + 1 \geq 2N(h-2) + 1 \geq$$

$$\geq \ldots \geq 2^i N(h-2i) + \sum_{j=0}^{i-1} 2^j = 2^i N(h-2i) + 2^i - 1 = 2^i(N(h-2i)+1) - 1.$$

Per $i = \left\lfloor \frac{h}{2} \right\rfloor$ si ha:

$$N(h) \geq 2^{\left\lfloor \frac{h}{2} \right\rfloor} \left(N\left(h - 2\left\lfloor \frac{h}{2} \right\rfloor\right) + 1 \right) - 1.$$

Distinguiamo i due possibili casi per h.

- Se h è pari:

$$N(h) \geq 2^{\frac{h}{2}}(N(0)+1) - 1 = 2^{\frac{h}{2}}(2) - 1 = 2^{\frac{h}{2}+1} - 1.$$

- Se h è dispari:

$$N(h) \geq 2^{\frac{h-1}{2}}(N(1)+1) - 1 = 2^{\frac{h-1}{2}}(3) - 1 = 2^{\frac{h}{2}+1-\frac{1}{2}+\log_2 3} - 1 > 2^{\frac{h}{2}+1} - 1$$

poiché $-\frac{1}{2} + \log_2 3 > 0$ (infatti $\log_2 3 \approx 1.58\ldots$).

Abbiamo quindi dimostrato che per il numero n di nodi presenti in un albero AVL di altezza h vale:

$$N(h) \geq 2^{\frac{h}{2}+1} - 1.$$

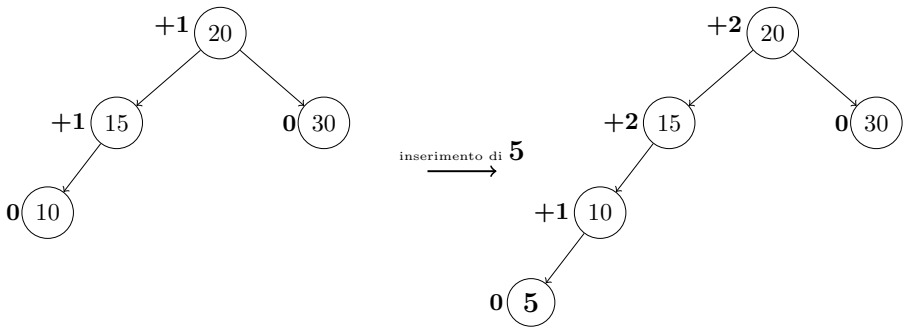
Da questa relazione, con semplici passaggi algebrici, otteniamo:

$$h \leq 2\log_2(n+1) - 2 \leq 2\log_2(2n) - 2 = 2\log_2 n$$

Poiché $h = O(\log n)$, concludiamo che gli alberi AVL sono bilanciati.

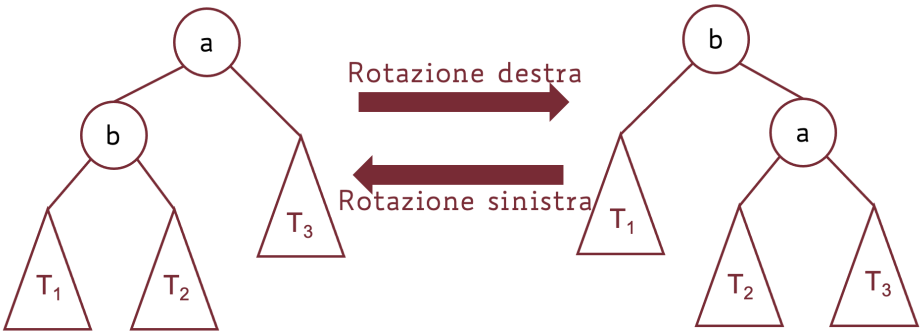
8.4.3 Ribilanciamento tramite rotazioni

Inserimento e cancellazione in un albero AVL sono operazioni che richiedono tempo $O(\log n)$ trattandosi di alberi binari di ricerca tuttavia, a seguito di una di queste operazioni, l'albero può sbilanciarsi, contenere cioè nodi con fattore di sbilanciamento pari a ± 2. Ad esempio, nella figura che segue si mostra come l'inserimento del nodo con chiave 5 in un albero AVL produca un albero di ricerca che non è più AVL, perché alcuni fattori di bilanciamento passano a +2.



Se, a seguito dell'inserimento, il fattore di bilanciamento dei nodi non è più compreso tra -1 e $+1$, allora l'albero deve essere ribilanciato per ripristinare l'equilibrio. Entrano quindi in gioco le **rotazioni**, operazioni che modificano la struttura dell'albero senza

alterare l'ordinamento dei nodi. Considereremo due tipi principali di rotazioni: la rotazione destra e la rotazione sinistra.
Nella figura che segue sono illustrate le rotazioni destra e sinistra, rispettivamente:

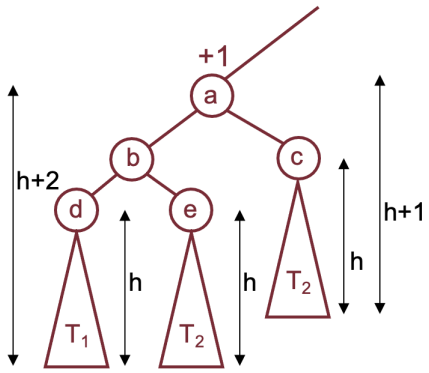


La prima cosa da notare è che, quando applicate, queste rotazioni non alterano l'ordinamento relativo tra le chiavi né l'ordine che ne risulta dopo una visita in-ordine. Di conseguenza, se utilizzate su un albero di ricerca, questo mantiene la sua proprietà di ordinamento orizzontale.
Il tipo di rotazioni da applicare per ri-bilanciare l'albero dipenderà dal tipo di sbilanciamento causato dall'inserimento o dalla cancellazione dell'elemento, come meglio specificato nel seguito.

8.4.4 Inserimento in alberi AVL e ribilanciamento

L'inserimento di una nuova chiave in un albero AVL è, nella sua prima parte, identico all'inserimento in un semplice albero binario di ricerca, con l'unica differenza che, una volta inserita la nuova chiave, il fattore di bilanciamento di tutti i nodi che sono stati interessati da questa operazione viene aggiornato, se necessario. Osserviamo

che, per come è fatto l'algoritmo di inserimento in un albero binario di ricerca, solo i nodi lungo il cammino dalla radice al nodo appena inserito possono subire una revisione ed un eventuale aggiornamento del loro fattore di bilanciamento. Sia *x* uno di questi nodi; se il nuovo nodo è stato inserito nel suo sottoalbero sinistro, il fattore di bilanciamento di *x* potrà aumentare di 1, mentre se il nodo è stato inserito nel suo sottoalbero destro, il fattore di bilanciamento di *x* potrà essere decrementato di 1. Se l'inserimento causa uno sbilanciamento, sia *a* il primo nodo in cui si riscontra un fattore di bilanciamento pari a ± 2 che incontriamo risalendo il cammino dal nodo inserito alla radice, ed analizziamo come ripristinare l'equilibrio, supponendo che *a* assuma fattore di bilanciamento +2 (il caso in cui a assume fattore di bilanciamento -2 è del tutto simmetrico). Nella figura seguente è rappresentata la situazione del sottoalbero radicato in *a*, **prima** dell'inserimento del nodo nel suo sottoalbero sinistro.



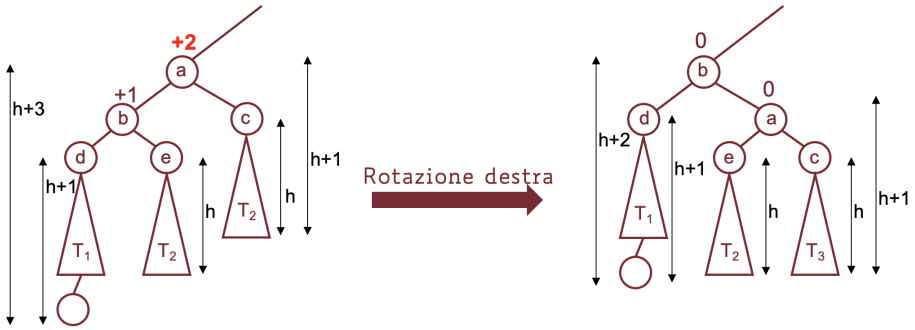
Preliminarmente, osserviamo che, per produrre uno sbilanciamento di +2, il nodo *a* deve inizialmente avere un fattore di bilanciamento

pari a +1, e i tre sottoalberi radicati in *d*, *e* e *c* devono avere la stessa altezza *h*. Infatti, se l'altezza del sottoalbero radicato in *e* non fosse uguale a quella del sottoalbero radicato in *d*, l'inserimento del nodo nel sottoalbero di altezza maggiore avrebbe causato uno sbilanciamento pari a +2 già nel nodo *b*, mentre l'inserimento nel sottoalbero di altezza minore non avrebbe modificato lo sbilanciamento di *a*. Infine, poiché il sottoalbero sinistro di *a* ha altezza h+2 e lo sbilanciamento di *a* è +1, anche il sottoalbero radicato in *c* ha altezza *h*.

Distinguiamo ora due sottocasi possibili e mostriamo che al più due opportune rotazioni saranno sufficienti a ribilanciare l'albero.

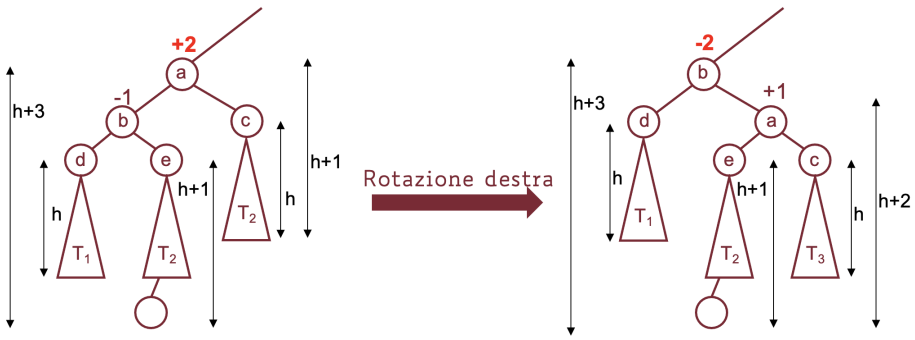
1. Il nodo viene inserito nel sottoalbero sinistro del figlio sinistro di *a* (T_1 nella figura):

   In questo caso, per ribilanciare, è sufficiente eseguire una sola rotazione destra del nodo *a* (vedi figura seguente). Dopo la rotazione, il sottoalbero radicato in *b* avrà la stessa altezza h+1 che aveva il sottoalbero radicato in *a* prima dell'inserimento. Di conseguenza, i nodi antenati di b manterranno lo stesso fattore di bilanciamento che avevano prima dell'inserimento. Gli unici nodi che avranno un fattore di bilanciamento diverso dal precedente saranno *b* ed *a*, che assumono fattore di bilanciamento 0 e pertanto risultano anch'essi bilanciati. In questo modo l'intero albero risulta bilanciato.

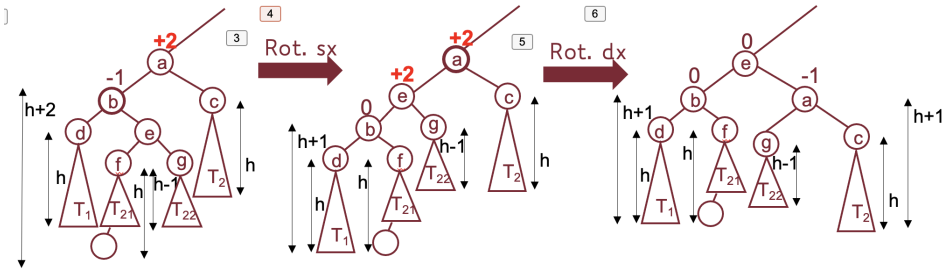


2. Il nodo viene inserito nel sottoalbero destro del figlio sinistro di *a* (T_2 nella figura):

   In questo caso, come illustrato nella seguente figura, una semplice rotazione a destra non è sufficiente:



   Invece, bisogna far precedere una rotazione destra del nodo *a* da una rotazione sinistra del nodo *b*, come si fede in figura:



Qui abbiamo a sinistra la situazione sbilanciata, al centro l'effetto della prima rotazione che coinvolge il nodo b ed a destra la situazione finale dopo la rotazione di a. Al termine delle due rotazioni il sottoalbero radicato in e avrà la stessa altezza $h+2$ che aveva il sottoalbero radicato in a prima dell'inserimento. Gli unici nodi il cui fattore di bilanciamento cambia a seguito dell'inserimento sono a, b ed e che avranno ora i fattori di bilanciamento: -1, 0 e 0, rispettivamente. In questo modo l'albero risulta nuovamente bilanciato.

Nella figura precedente si è considerato il caso in cui il nuovo nodo venga inserito nel sottoalbero sinistro di e. Si può verificare come le stesse due rotazioni ribilancino l'albero anche nel caso in cui il nuovo nodo venga inserito nel sottoalbero destro di e. L'unica variazione rispetto al caso precedente riguarda i fattori di bilanciamento dei nodi a e b, che risultano ora rispettivamente 0 e $+1$. Questi cambiamenti assicurano comunque che l'albero sia nuovamente bilanciato mantenendo perciò intatte le proprietà di un albero AVL.

Come già accennato, i casi da considerare per un nodo da ribilancia-

re con fattore di bilanciamento pari a -2 sono simmetrici a quelli analizzati per il caso $+2$.

Per tutto quanto visto, l'algoritmo per inserire una chiave x in un albero AVL, con eventuale ribilanciamento, può essere schematizzato come segue:

1. **Creazione nodo e suo inserimento:**
   Si crea un nuovo nodo con chiave x e lo si inserisce nell'albero T come foglia, posizionandolo nel punto corretto secondo l'ordinamento dell'albero binario di ricerca.

2. **Ricalcolo dei fattori di bilanciamento:**
   Si aggiornano i fattori di bilanciamento dei nodi il cui bilanciamento può essere variato a seguito dell'inserimento, cioè quelli sul cammino che va dal nodo inserito alla radice dell'albero. Ciò può essere fatto risalendo lungo il cammino dalla foglia verso la radice.

3. **Verifica dell'eventuale sbilanciamento e ribilanciamento:**
   Durante la risalita, se si incontra un nodo a con un fattore di bilanciamento pari a ± 2, che chiameremo *nodo critico*, è necessario eseguire un ribilanciamento del sottoalbero radicato nel nodo critico tramite rotazioni. I casi da considerare sono 4:

   - **se il fattore di bilanciamento è $+2$ e il nodo è stato inserito nel sottoalbero sinistro del sottoalbero sinistro** si esegue una rotazione sul nodo critico *a* verso destra;

- **se il fattore di bilanciamento è $+2$ e il nodo è stato inserito nel sottoalbero destro del sottoalbero sinistro** si eseguono due rotazioni: la prima verso sinistra sul nodo *b*, figlio sinistro del nodo critico, e la seconda verso destra sul nodo critico;
- **se il fattore di bilanciamento è -2 e il nodo è stato inserito nel sottoalbero destro del sottoalbero destro** si esegue una rotazione verso sinistra sul nodo critico;
- **se il fattore di bilanciamento è -2 e il nodo è stato inserito nel sottoalbero sinistro del sottoalbero destro** si eseguono due rotazioni: la prima verso destra sul figlio destro del nodo critico e la seconda verso sinistra sul nodo critico.

Costo computazionale dell'inserimento: Vediamo passo per passo il contributo al costo computazionale.

Passo 1 (Creazione nodo e suo inserimento): Il nuovo nodo viene inserito in tempo $O(h) = O(\log n)$ dato che l'altezza di un albero AVL con n nodi è $O(\log n)$.

Passo 2 (Ricalcolo dei fattori di bilanciamento): Questo passo richiede anch'esso $O(h) = O(\log n)$, poiché coinvolge solo i nodi lungo il cammino dalla radice alla foglia.

Passo 3 (Verifica di un eventuale sbilanciamento e ribilanciamento): Una volta identificato il nodo critico, il ribilanciamento richiede $O(1)$, in quanto le rotazioni sono eseguite in tempo costante modificando solo alcuni puntatori.

Sommando i tre contributi, l'inserimento di una chiave con eventuale ribilanciamento in un albero AVL ha costo complessiv $O(\log n)$.

Nell'implementazione Python dell'algoritmo proposta di seguito, si assume che ogni nodo dell'albero sia rappresentato da quattro campi:

- chiave: il valore del nodo,
- sinistro: il puntatore al figlio sinistro,
- destro: il puntatore al figlio destro,
- altezza: l'altezza del nodo all'interno dell'albero.

In altre parole, definiamo la classe del nodo come segue:

```
class Nodo:
      def __init__(self, chiave):
          self.chiave = chiave
          self.sinistro = None
          self.destro = None
          self.altezza = 1  # Altezza iniziale è 1
                                    # (una foglia)
```

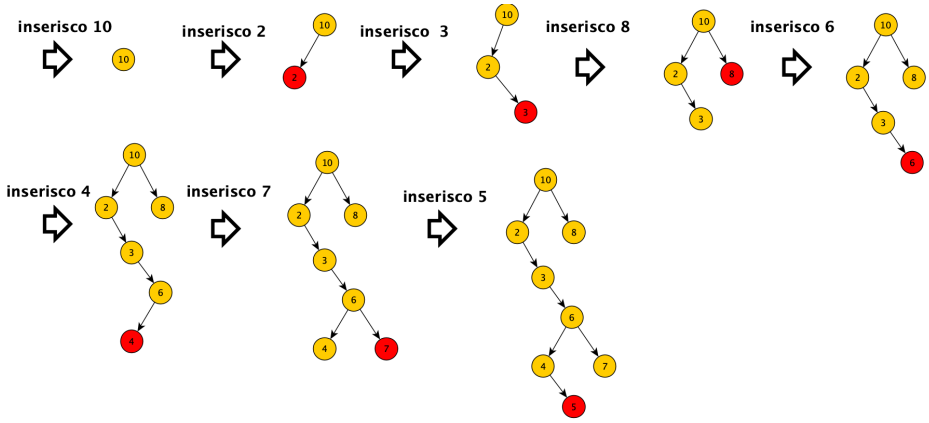
Ecco la parte principale dell'algoritmo:

```
def inserisci(nodo, chiave):
        # 1. Esegui l'inserimento normale in un ABR
        if nodo is None:
            return Nodo(chiave)
        if chiave < nodo.chiave:
            nodo.sin = inserisci(nodo.sin, chiave)
        else:
            nodo.des = inserisci(nodo.des, chiave)
        # 2. Aggiorna l'altezza del nodo corrente
        nodo.altezza = 1 + max(alt(nodo.sin), alt(nodo.des))
        # 3. Calcola il fattore di bilanciamento
        fattore = alt(nodo.sin) - alt(nodo.des)
        # 4. Se il nodo è sbilanciato, ci sono 4 casi:
        # Caso 1: Rotazione a destra
        if fattore > 1 and chiave < nodo.sin.chiave:
            return rotazione_destra(nodo)
        # Caso 2: Rotazione sinistra-destra
        if fattore > 1 and chiave > nodo.sin.chiave:
            nodo.sin = rotazione_sinistra(nodo.sin)
            return rotazione_destra(nodo)
        # Caso 3: Rotazione a destra
        if fattore < -1 and chiave > nodo.des.chiave:
            return rotazione_sinistra(nodo)
        # Caso 4: Rotazione destra-sinistra
        if fattore < -1 and chiave < nodo.des.chiave:
            nodo.des = rotazione_destra(nodo.des)
            return rotazione_sinistra(nodo)
        # Ritorna il nodo (modificato o no)
        return nodo
```

Infine di seguito sono riportate le funzioni necessarie per eseguire le rotazioni e calcolare l'altezza alt dei nodi.

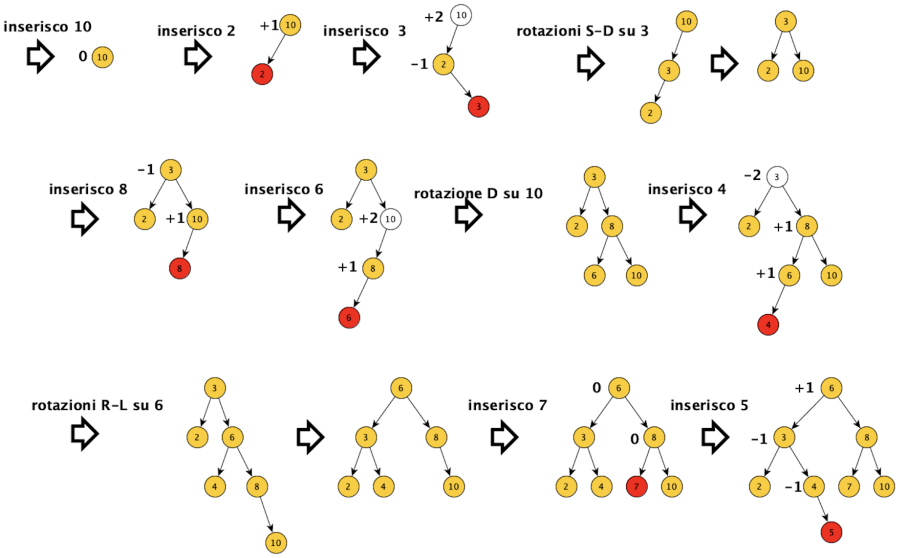
```
    def rotazione_sinistra(nodo):  # Rotaz. sinistra
        nuova = nodo.des
        nodo.des = nuova.sin
        nuova.sin = nodo
        # Aggiorna l'altezza dei nodi
        nodo.altezza = \
              max(alt(nodo.sin), alt(nodo.des)) + 1
        nuova.altezza = \
              max(alt(nuova.sin), alt(nuova.des) ) + 1
        return nuova

    def rotazione_destra(nodo):     # Rotaz. destra
        nuova = nodo.sin
        nodo.sin = nuova.des
        nuova.des = nodo
        # Aggiorna l'altezza dei nodi
        nodo.altezza = \
              max(alt(nodo.sin),alt(nodo.des)) + 1
        nuova.altezza = \
              max(alt(nuova.sin), alt(nuova.des) ) + 1
        return nuova

    def alt(nodo):     # Calcola l'altezza di un nodo
        if nodo == None: return -1. #Alt. di nodo nullo= -1
    return nodo.altezza
```

Concludiamo questa sezione con un esempio.



Nella precedente figura è mostrato un esempio dell'evoluzione di un albero binario di ricerca (sbilanciato) a seguito all'inserimento della sequenza di chiavi 10, 2, 3, 8, 6, 4, 7 e 5.

Invece, nella prossima figura viene illustrato come evolve l'albero di ricerca a seguito dell'inserimento della stessa sequenza di chiavi, quando si richieda di mantenerlo bilanciato, cioè che sia un AVL.



8.4.5 Cancellazione in alberi AVL e ribilanciamento

Così come l'inserimento, anche la cancellazione in un albero AVL segue le stesse regole che in un Albero Binario di Ricerca, ma è seguita da una fase di ribilanciamento necessaria per correggere eventuali sbilanciamenti causati dalla rimozione del nodo. Questo ribilanciamento viene eseguito utilizzando gli stessi schemi di rotazione descritti per il caso dell'inserimento.
L'algoritmo per cancellare una chiave x può essere schematizzato come segue:

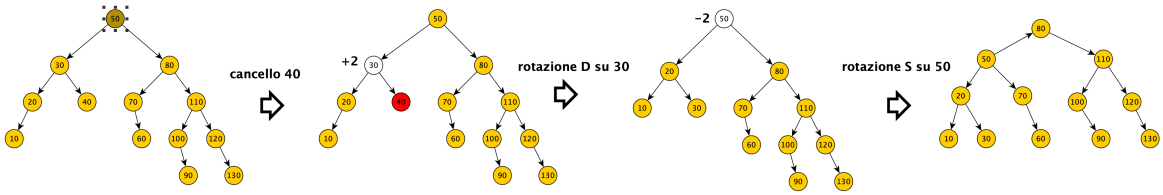
1. **Rimozione della chiave:**
   La cancellazione della chiave x avviene come in un normale Albero Binario di Ricerca:
   - Si cerca il nodo con chiave x
   - Se il nodo da eliminare è una foglia, lo si rimuove direttamente.
   - Se il nodo ha un solo figlio, lo si sostituisce con il figlio stesso.
   - Se il nodo ha due figli, lo si sostituisce con il suo successore in ordine (il nodo con il valore minimo nel sottoalbero destro) e si elimina il successore.
2. **Ricalcolo dei fattori di bilanciamento:**
   Dopo la rimozione, si aggiornano i fattori di bilanciamento risalendo il cammino dai nodi interessati, vale a dire il cammino che va dal padre del nodo eliminato alla radice dell'albero.

3. **Verifica e ribilanciamento:** Durante la risalita, se un nodo a diventa critico, ovvero con un fattore di bilanciamento pari a ± 2 è necessario eseguire un ribilanciamento del sottoalbero radicato in a. I casi possibili sono simili a quelli dell'inserimento:

   - **Se il fattore di bilanciamento è $+2$ e il sottoalbero sinistro è in equilibrio o ha un fattore di bilanciamento** ≥ 0, si esegue una rotazione destra su a.
   - **Se il fattore di bilanciamento è $+2$ e il sottoalbero sinistro ha un fattore di bilanciamento** < 0**,** si esegue una rotazione sinistra sul figlio sinistro di a, seguita da una rotazione destra su a
   - **Se il fattore di bilanciamento è -2 e il sottoalbero destro è in equilibrio o ha un fattore di bilanciamento** ≤ 0, si esegue una rotazione sinistra su a
   - **Se il fattore di bilanciamento è -2 e il sottoalbero destro ha un fattore di bilanciamento** > 0, si esegue una rotazione destra sul figlio destro di a, seguita da una rotazione sinistra su a

4. **Propagazione del ribilanciamento:** A differenza dell'inserimento, dopo il ribilanciamento locale, l'altezza dell'albero può ridursi di un'unità. Pertanto, potrebbe essere necessario propagare il ribilanciamento verso i nodi antenati lungo il cammino fino alla radice. Si veda, ad esempio, la figura seguente, che

illustra un caso di cancellazione in un albero AVL in cui sono necessarie due rotazioni consecutive a sinistra.



Costo computazionale dell'algoritmo

- **Passo 1 (cancellazione nodo)**: La cancellazione richiede $O(\log n)$ tempo, poiché l'altezza di un albero AVL con n nodi è $\Theta(\log n)$.
- **Passo 2 (Ricalcolo dei fattori di bilanciamento)**: Questo richiede $O(\log n)$ perché coinvolge solo i nodi lungo il cammino tra la radice ed il nodo rimosso.
- **Passo 3 (Verifica e ribilanciamento)**: Ogni ribilanciamento richiede $\Theta(1)$, poiché una singola rotazione o una doppia rotazione avviene in tempo costante.
- **Passo 4 (Propagazione del ribilanciamento)**: Può essere necessario ripetere il Passo 3 lungo tutto il cammino fino alla radice, eseguendo quindi $O(\log n)$ rotazioni in totale, per un costo totale di $O(\log n)$.

In conclusione la cancellazione con eventuale ribilanciamento in un albero AVL ha costo complessivo $O(\log n)$.

Ecco di seguito il codice Python:

```
def cancella(nodo, chiave):
        # 1. la chiave non è presente nell'albero
        if nodo is None:
            return nodo
        if chiave < nodo.chiave:
            nodo.sin = cancella(nodo.sin, chiave)
        elif chiave > nodo.chiave:
            nodo.des = cancella(nodo.des, chiave)
        else:
            # Nodo con una sola foglia o nessuna
            if nodo.sin is None:
                return nodo.des
            elif nodo.des is None:
                return nodo.sin
            # Nodo con due foglie: trova il minimo
            # nel sottoalbero destro
            temp = nodo_minimo(nodo.des)
            # sostituisci la chiave del nodo con il
            # minimo nel sottoalbero destro
            nodo.chiave = temp.chiave
            #cancella il minimo dal sottoalbero destro
            nodo.des = cancella(nodo.des, temp.chiave)

        # continua...
```

```
        # 2. Aggiorna l'altezza del nodo corrente
        nodo.altezza = \
            max(alt(nodo.sin), alt(nodo.des)) + 1
        # 3. Calcola il fattore di bilanciamento
        fattore = alt(nodo.sin) - alt(nodo.des)
        # Caso 1: Rotazione a destra
        if fattore > 1 and alt(nodo.sin.sin) >= \
            alt(nodo.sin.des):
            return rotazione_destra(nodo)
        # Caso 2: Rotazione  sinistra-destra
        if fattore > 1 and alt(nodo.sin.sin) < \
            alt(nodo.sin.des):
            nodo.sin = rotazione_sinistra(nodo.sin)
            return rotazione_destra(nodo)
        # Caso 3: Rotazione sinistra
        if fattore < -1 and alt(nodo.des.des) >= \
            alt(nodo.des.sin):
            return rotazione_sinistra(nodo)
        # Caso 4:  Rotazione destra-sinistra
        if fattore < -1 and alt(nodo.des.des) < \
            alt(nodo.des.sin):
            nodo.destro = rotazione_destra(nodo.des)
            return rotazione_sinistra(nodo)
        return nodo
```

```
# Funzione per trovare il nodo con il valore minimo
    def nodo_minimo(nodo):
        corrente = nodo
        while corrente.sin:
            corrente = corrente.sin
        return corrente
```

8.5 Alberi Red-Black (argomento facoltativo)

Gli (***alberi Red-Black***) sono un'alternativa agli AVL, infatti sono anch'essi degli alberi binari di ricerca la cui altezza è logaritmica rispetto al numero di nodi. Ciò viene garantito non più tramite i fattori di bilanciamento ma attraverso un colore (rosso o nero) che viene assegnato a ciascun nodo, insieme ad alcune proprietà che questi colori devono rispettare.

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***.

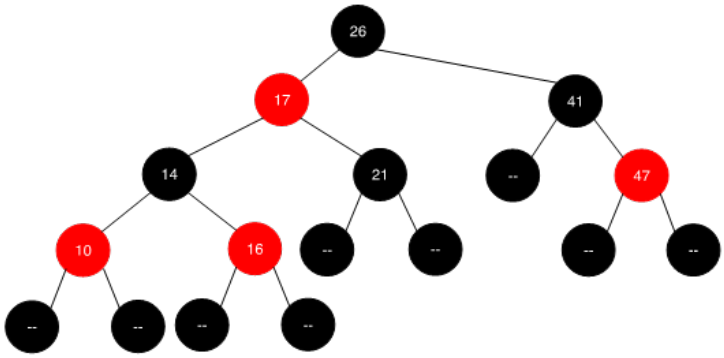
Un RB-albero soddisfa, inoltre, le seguenti proprietà aggiuntive:

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).

Anche in questo caso, vengono usate le ***rotazioni***, che possono essere destre o sinistre, già definite per gli AVL. Ricordiamo che si tratta di operazioni locali con costo costante che non modificano l'ordinamento delle chiavi secondo la visita in-ordine.

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.

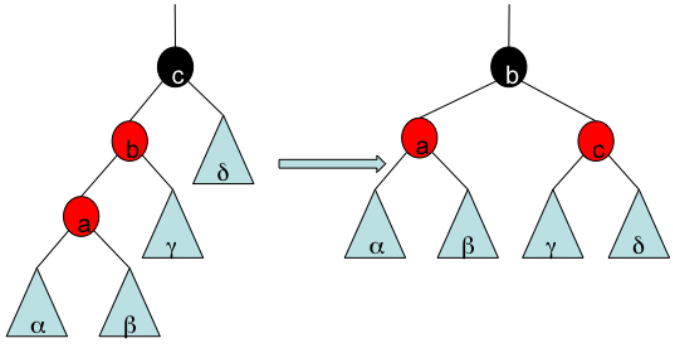
Caso 3. Il nodo inserito è figlio sinistro di un nodo rosso e lo zio è nero.

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.

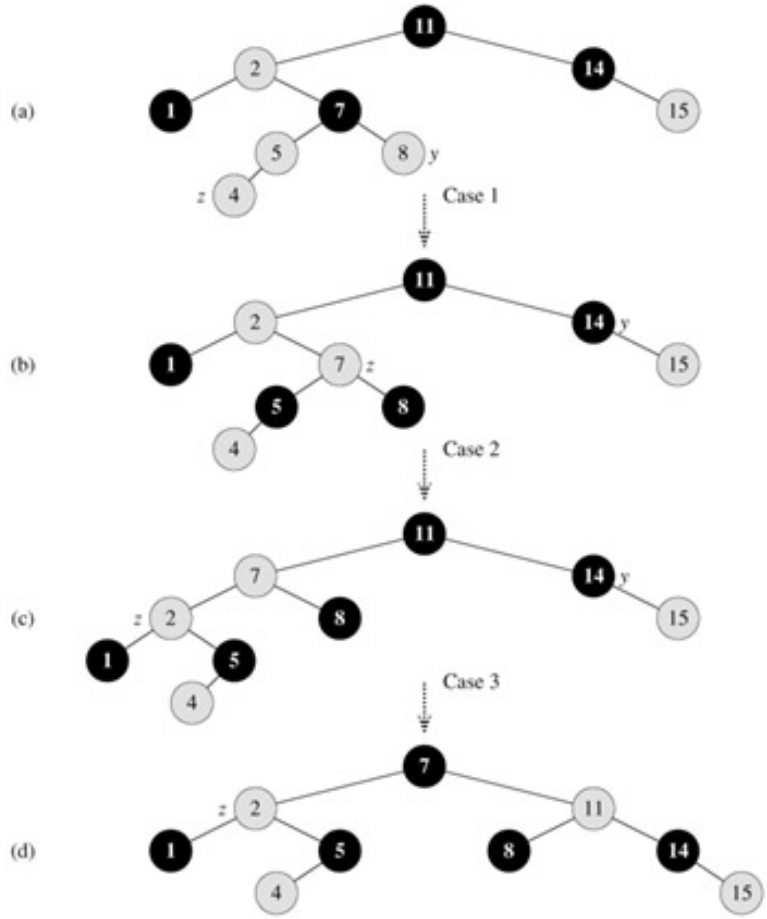


Ne consegue che, durante un inserimento, è possibile entrare più volte nel Caso 1., eventualmente nel Caso 2., e si conclude sempre con il Caso 3.

Un esempio di funzionamento dell'operazione di inserimento (viene inserito la chiave 4) è riportato nella figura qui di seguito. 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.



8.6 B-alberi

Gli alberi AVL sono ideali per garantire accessi efficienti ai dati grazie al loro bilanciamento rigoroso, che assicura un costo computazionale di $O(\log n)$ per le operazioni di ricerca, inserimento e cancellazione. Tuttavia, presentano alcune limitazioni che emergono in presenza di dati di grandi dimensioni, in cui i dati non risiedono interamente in memoria principale, ma sono invece memorizzati su dispositivi di archiviazione secondaria:

- **Elevato numero di accessi a memoria secondaria**: Ogni nodo di un albero AVL contiene un singolo valore e due puntatori. Quando la quantità di dati è elevata, non si riesce a sfruttare appieno il blocco di memoria letto con un accesso a memoria esterna (che di solito è più grande di un nodo). Questo comporta un numero elevato di accessi alla memoria secondaria, rallentando le operazioni.
- **Altezza dell'albero**: Quando si esegue un algoritmo su alberi AVL, ad esempio di ricerca, il numero di accessi alla memoria secondaria è proporzionale all'altezza; nonostante questa sia logaritmica, il numero di accessi risultante potrebbe risultare troppo elevato.

I *B-alberi* sono stati progettati per mitigare queste problematiche, ottimizzando le operazioni su dispositivi di archiviazione secondaria. Le loro caratteristiche principali includono:

1. **Nodi con più chiavi**: Ogni nodo di un B-albero può contenere diverse chiavi, riducendo il numero complessivo di nodi e quindi

l'altezza dell'albero. Questo consente di lavorare ai dati con un minor numero di accessi a memoria secondaria.

2. **Bilanciamento**: Come gli alberi AVL, i B-alberi mantengono il bilanciamento dopo ogni operazione di inserimento o cancellazione. Questo garantisce un'altezza logaritmica nel numero di nodi (che è molto meno del numero di chiavi).

3. **Efficienza nell'accesso a memoria secondaria**: I nodi dei B-alberi sono progettati per sfruttare al massimo il blocco di memoria letto dal disco. Un singolo accesso a memoria secondaria carica un intero nodo in memoria principale, permettendo di esaminare più chiavi in un solo passaggio.

Concludendo, gli alberi AVL sono ottimi per dati in memoria principale, ma per grandi dataset memorizzati su disco, i B-alberi rappresentano una scelta più efficiente. Nel seguito approfondiremo la struttura dei B-alberi e i loro principali vantaggi.

Definizione. Un *B-albero* di grado minimo $t \geq 2$ ha le seguenti proprietà:

- Il nodo radice contiene un numero s di chiavi memorizzate in ordine crescente e $0 \leq s \leq 2t-1$;
- Ogni nodo diverso dalla radice (sia interno che foglia) contiene un numero di chiavi s memorizzate in ordine crescente e $t-1 \leq s \leq 2t-1$;
- Un nodo interno contenente s chiavi, $k_1 < k_2 < \ldots < k_s$ ha esattamente $s+1$ figli, e per le chiavi negli $s+1$ sottoalberi T_i si verifica che:

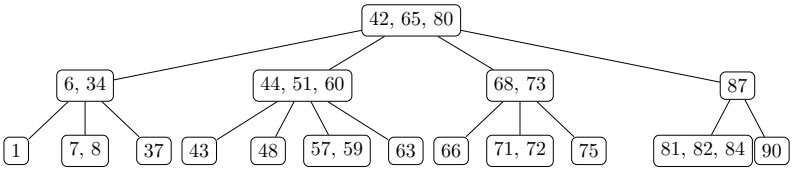
  - le chiavi in T_1 sono minori di k_1;
  - le chiavi in T_{s+1} sono maggiori di k_s;
  - le chiavi in T_i, con $1 < i < s + 1$, sono comprese tra k_{i-1} e k_i.

· i nodi foglia si trovano tutti alla stessa profondità.

In base alla definizione il valore t corrisponderà al grado minimo dei nodi interni del B-albero diversi dalla radice.
Di seguito è riportata la figura che mostra un esempio di B-albero con grado minimo $t = 2$. La figura evidenzia anche la struttura di uno dei suoi nodi interni, dove ogni nodo contiene:

· **Chiavi:** Valori ordinati utilizzati per separare i sottoalberi.

· **Puntatori ai figli:** Riferimenti ai sottoalberi associati agli intervalli determinati dalle chiavi.



La struttura dei nodi multi-chiave e multi-figlio, l'altezza ridotta (ad esempio, con nodi contenenti 100 chiavi, tre livelli possono rappresentare fino a circa $100^3 = 1000000$ di chiavi), e la disposizione ordinata delle chiavi nei nodi consentono ricerche rapide (es. ricerca binaria) e operazioni sequenziali (come una scansione ordinata dell'albero) efficienti, poiché beneficiano della disposizione compatta delle chiavi nei nodi, riducendo la necessità di passare da un nodo

all'altro. I B-alberi, inoltre, possono gestire la crescita dinamica dei dati senza richiedere riorganizzazioni complete della struttura: le operazioni di inserimento e cancellazione sono progettate per mantenere la struttura bilanciata con un costo estremamente esiguo, non solo in termini asintotici, ma anche di numero di accessi a memoria esterna. I B-alberi, quindi, minimizzano il numero di pagine da leggere o scrivere, divenendo così ideali per database e sistemi di file su memoria esterna.

In particolare, come vedremo nelle sezioni seguenti, in un B-albero di grado t contenente n chiavi, le tre operazioni fondamentali di un dizionario — *ricerca*, *inserimento* e *cancellazione* — possono essere implementate con un costo computazionale di $O(\log n)$, $O(t \log n)$ e $O(t \log n)$, rispettivamente. Considerando che t è generalmente molto più piccolo di n e che la struttura dell'albero rimane stabile durante l'esecuzione di inserimenti e cancellazioni, possiamo considerare t come una costante. Di conseguenza, tutte e tre le operazioni possono essere eseguite in tempo logaritmico rispetto al numero di chiavi presenti nell'albero.

Vediamo ora la classe BTreeNode che definisce un nodo del B-albero in Python.

```
class BTreeNode:
    def __init__(self, t, is_leaf):
        self.t = t  # Grado minimo
        self.is_leaf = is_leaf  # foglia?
        self.chiavi = [ ]  # chiavi nel nodo
        self.figli = [ ]  # figli (vuota se è foglia)
```

Ecco una descrizione dettagliata della classe **BTreeNode**:

t: rappresenta il grado minimo del B-albero. Ogni nodo (tranne la radice) deve contenere almeno $t-1$ chiavi e può contenerne al massimo $2t-1$;

is_leaf: è una variabile booleana che indica se il nodo è una foglia.

chiavi: è un array che memorizza le chiavi presenti nel nodo, mantenute in ordine crescente.

figli: è un array che memorizza i puntatori ai figli del nodo. Ha $\text{len}(\text{chiavi})+1$ elementi se il nodo non è una foglia, è vuota altrimenti.

8.6.1 I B-alberi sono alberi bilanciati

Dimostriamo ora che i B-alberi sono bilanciati, ovvero che l'altezza è $h=\Theta(\log_t n)$ studiando i casi estremi, cioè quelli in cui ciascun nodo contiene rispettivamente il minimo ed il massimo numero di chiavi possibili.

Caso 1. Minimo numero di chiavi per nodo: In questo caso, la radice contiene una chiave, ed ogni altro nodo contiene esattamente t-1 chiavi. A parità di altezza, un tale B-albero contiene il minimo numero di nodi possibile n_{min}.
Osserviamo che, in questo caso particolare, la radice ha esattamente due figli mentre gli altri nodi ne hanno t, il B-albero è costituito da due copie identiche di un albero in cui ogni nodo ha esattamente t-1 figli e le cui radici sono figli della radice del B-albero. Allora, a livello $i \geq 1$ abbiamo $2t^{i-1}(t-1)$ chiavi ed Il numero minimo di chiavi in un B-albero di altezza h è dato da:

$$n_{\text{min}} = 1 + 2(t-1) \cdot \sum_{j=0}^{h-1} t^j = 1 + 2(t-1) \cdot \frac{t^h - 1}{t-1} > 2(t^h - 1).$$

Quindi, $n \geq n_{min} > 2(t^h - 1)$ e l'altezza h risulta limitata superiormente da:

$$h > \log_t(n/2 + 1).$$

Caso 2. Massimo numero di chiavi per nodo: In questo caso, ogni nodo contiene il numero massimo di chiavi, $2t-1$. A parità di altezza, un tale B-albero contiene il massimo numero di nodi possibile n_{max}. A livello $i \geq 0$ abbiamo $(2t-1)(2t)^i$ chiavi ed il numero massimo di chiavi in un B-albero di altezza h è dato da:

$$n_{\text{max}} = (2t-1) \cdot \sum_{i=0}^{h} (2t)^i = (2t-1) \cdot \frac{(2t)^{h+1} - 1}{2t-1} = (2t)^{h+1} - 1.$$

Quindi, $n \leq (2t)^{h+1} - 1$ e l'altezza h risulta limitata inferiormente da:

$$h \geq \log_{2t}(n+1) - 1.$$

Combinando i due risultati, otteniamo:

$$\log_{2t}(n+1) - 1 < h \leq \log_t(n/2+1)$$

Poiché le due basi dei logaritmi differiscono per una costante, possiamo concludere che:

$$h = \Theta(\log_t n).$$

Nei B-alberi, il grado t influisce sulla loro struttura, determinando quanti figli e nodi può avere ciascun nodo. Tuttavia, t non varia al crescere del numero delle chiavi. Nel caso in cui possiamo dire che $t=\Theta(1)$, l'altezza dell'albero cresce in modo logaritmico rispetto al numero di chiavi, ossia $h = O(\log n)$. Inoltre, se consideriamo un albero con N nodi ed n chiavi, vale la relazione:

$$(t-1)(N-1) \leq n \leq 2(t-1)N$$

che lega i due numeri; in altri termini, $n = \Theta(t \cdot N)$.

8.6.2 Ricerca nei B-alberi

L'algoritmo di ricerca in un B-albero è basato sui seguenti passaggi principali:

1. **Partenza dalla radice:** La ricerca inizia dal nodo radice dell'albero. Se l'albero è vuoto (ossia se la radice è None), ci troviamo nel caso base e la ricerca termina immediatamente con un insuccesso.

2. **Ricerca nel nodo corrente:** Ogni nodo di un B-albero contiene un insieme di chiavi ordinate. Per cercare una chiave, si effettua una scansione delle chiavi presenti nel nodo distinguendo i casi in cui il nodo è una foglia o meno.

   Se il nodo è una foglia:

   - Si scorre l'insieme delle chiavi;
   - Se si trova la chiave cercata, la ricerca termina con successo;
   - Se si incontra una chiave maggiore della chiave cercata, o se si arriva alla fine della lista senza aver trovato la chiave, la ricerca termina con insuccesso.

   Se il nodo è un nodo interno:

   - Se la chiave cercata è **uguale** a una delle chiavi nel nodo, la ricerca termina con successo;
   - Se la chiave cercata è **minore** della più piccola chiave del nodo, la ricerca continua nel sottoalbero che corrisponde alla parte sinistra di quella chiave;

- Se la chiave cercata è **maggiore** della più grande chiave del nodo, la ricerca prosegue nel sottoalbero che corrisponde alla parte destra (ossia, nel figlio che rappresenta i valori maggiori di tutte le chiavi del nodo);
- Se la chiave cercata è **minore** di una chiave c_i del nodo e **maggiore** della chiave successiva c_{i+1}, la ricerca prosegue nel sottoalbero tra le due chiavi (ossia, nel figlio che rappresenta i valori maggiori della chiave c_i ma minori della chiave c_{i+1}).

Intuitivamente, il tempo per esplorare un nodo richiede tempo $O(t)$ per scorrere al massimo le $2t-1$ chiavi presenti, e l'algoritmo visita al più un solo nodo per ognuno degli $h = \Theta\log_t n$ livelli; pertanto il costo è $O(t \log_t n)$.
Più formalmente, la ricerca di una chiave in un B-albero con n chiavi ed altezza $h = \Theta(\log n)$ è un algoritmo ricorsivo il cui costo computazionale è dato dall'equazione di ricorrenza: $T(h) = O(t) + T(h-1)$ e $T(0)=\Theta(1)$, la cui soluzione è $T(h) = \Theta(ht)=\Theta(t \log_t n)$.
Si noti che, nel caso in cui si sostituisca all'interno dei nodi la ricerca sequenziale con la ricerca binaria, il costo computazionale diventa $O(\log t \log_t n)$. Applicando la formula di cambio di base per i logaritmi (i.e. $\log_t n = \frac{\log_2 n}{\log_2 t}$), otteniamo un costo $O(\log n)$.

Ecco il codice in Python dell'algoritmo di ricerca:

```
def cerca_chiave(nodo, chiave):
    # Cerca una chiave all'interno di un B-albero,
    # a partire da un nodo dato. Restituisce
    # True se la chiave è trovata, False altrimenti
    i = 0
    # Troviamo la posizione in cui la chiave
    # potrebbe trovarsi nel nodo
    while i < len(nodo.chiavi) and
                        chiave > nodo.chiavi[i]:
        i += 1
    # Verifichiamo se la chiave è presente nel nodo
    if i < len(nodo.chiavi) and
                        chiave == nodo.chiavi[i]:
        return True
    # Se il nodo è una foglia, la chiave non è
    # presente
    if nodo.is_leaf:
        return False
    # Altrimenti cerchiamo ricorsivamente nel figlio
    return cerca_chiave(nodo.figli[i], chiave)
```

8.6.3 Inserimento in B-alberi

L'algoritmo di inserimento di una chiave in un B-albero è progettato per mantenere l'albero bilanciato e ordinato. Ecco i suoi passaggi principali:

Ricerca del nodo foglia e inserimento della chiave:

- Si inizia dalla radice e, come se si stesse seguendo un algoritmo di ricerca, si percorre l'albero seguendo i figli appropriati fino a raggiungere un nodo foglia in cui inserire la chiave;
- la chiave viene inserita nella posizione corretta rispetto alle altre chiavi, cioè mantenendone l'ordine crescente all'interno del nodo;

Funzione di split:

- Se, dopo l'inserimento, la foglia risulta troppo piena (cioè contiene $2t$ chiavi, superando quindi la capacità massima consentita di $2t-1$ chiavi), è necessario applicare la funzione di **split** che spezza il nodo foglia in due: la chiave che si trova nella posizione centrale viene spostata nel nodo genitore, mentre le chiavi rimanenti vengono divise tra il nodo originale e un nuovo nodo;
- se il nodo padre diventa troppo pieno (cioè contiene $2t$ chiavi), la divisione si propaga verso l'alto e il processo di split viene ripetuto ricorsivamente per il nodo padre;
- se il nodo diviso non è una foglia, anche i puntatori ai figli vengono separati;

Ripetizione del processo fino alla radice:

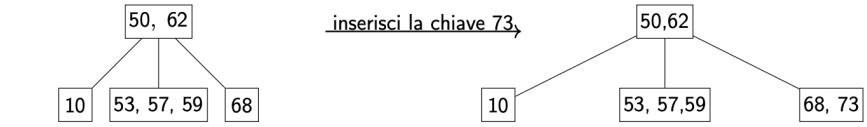
- La divisione si ripete fino a quando non si raggiunge un nodo che non è troppo pieno; questo nodo diventerà il punto finale del processo di inserimento;
- se la divisione raggiunge la radice e anche questa diventa troppo piena, la radice stessa deve essere divisa; in questo caso l'altezza del B-albero aumenta.

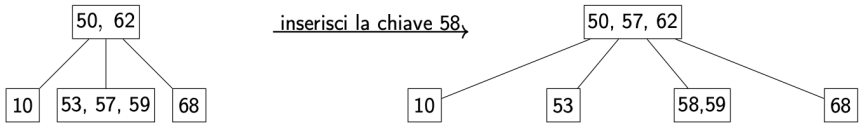
Anche il costo computazionale dell'inserimento di una chiave in un B-albero è $O(t \log_t n)$, infatti:

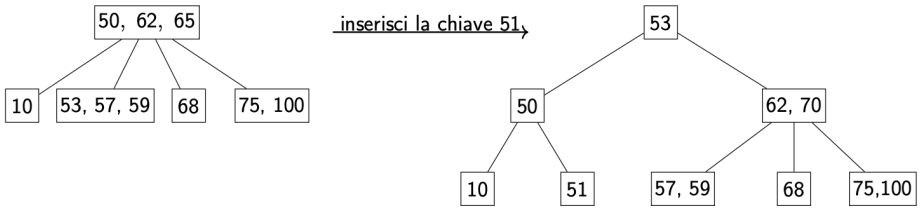
- La ricerca del nodo foglia in cui inserire la chiave richiede un tempo $O(t \log_t n)$ (algoritmo di ricerca).
- L'inserimento in un nodo non pieno o la divisione di un nodo avviene in tempo $O(t)$.
- La propagazione delle divisioni lungo l'albero avviene al più una volta per ciascun livello, e quindi non più di $O(\log_t n)$ volte.

Nella figura che segue viene mostrata l'evoluzione della struttura di un B-albero con $t = 2$ in seguito all'inserimento di alcune chiavi: l'inserimento della chiave 73 non provoca la sovradimensione del nodo; l' inserimento della chiave 58, invece, provoca la sovradimensione del nodo foglia in cui è inserita, e quindi si procede con uno split; l' inserimento della chiave 51, infine, provoca la sovradimensione del nodo foglia in cui è inserita per cui si procede con un primo split; a seguito di questo, il nodo padre (nel nostro caso la radice) risulta a sua volta sovradimensionato quindi segue l'ulteriore split

della radice per cui l'albero risultante avrà l'altezza incrementata di uno.







Segue una possibile implementazione dell'algoritmo di inserimento di una chiave in un B-albero in Python. E' riportato nel dettaglio anche il codice della funzione di split.

```
def inserisci_chiave(radice, chiave):
    # Inserisce la chiave nell'albero a partire
    #  dalla radice. Se essa è troppo piena, viene
    #  creata una nuova radice e l'altezza aumenta.
    t=radice.t
    aggiungi_chiave(radice,chiave,t)
    # Se la radice si riempie, gestiamo il caso
    # speciale creando una nuova radice
    if len(radice.chiavi)==2*t:
        nuovo_nodo = BTreeNode(t, False)
        nuovo_nodo.figli.append(radice)
        split(nuovo_nodo, 0, t)
    # La nuova radice è restituita
        return  nuovo_nodo
    return radice
```

```
def aggiungi_chiave(nodo, chiave, t):
    # Cerca il nodo foglia in cui inserire.
    # Se, dopo l'inserimento, un nodo
    # risulta troppo pieno (ha 2t chiavi), esegue
    # lo split di quel nodo.
    i = 0
    # Troviamo la posizione della chiave nel nodo
    while i < len(nodo.chiavi)
                    and chiave > nodo.chiavi[i]:
        i += 1
    # Se la chiave è già presente non facciamo nulla
    if i < len(nodo.chiavi)
                    and chiave == nodo.chiavi[i]:
        return
    # Se il nodo è una foglia, inseriamo la chiave
    if nodo.is_leaf:
        nodo.chiavi.insert(i, chiave)
        return
    # Se il nodo non è una foglia, cerchiamo
    # ricorsivamente nel figlio appropriato
    aggiungi_chiave(nodo.figli[i], chiave, t)
    # Dopo il ritorno, controlliamo se il figlio ha
    # troppe chiavi
    if len(nodo.figli[i].chiavi)==2*t:
        split(nodo,i,t)
```

```
def split(nodo, i, t):
    # Esegue lo split di un nodo pieno spostando
    # la chiave mediana nel nodo padre
    nodo_diviso = nodo.figli[i]
    nuovo_nodo = BTreeNode(t, nodo_diviso.is_leaf)
    # Spostiamo la metà destra delle chiavi nel
    # nuovo nodo
    nodo_chiavi = nodo_diviso.chiavi
    s = nodo_chiavi[t-1] # La chiave mediana
    nuovo_nodo.chiavi = nodo_chiavi[t:]
    nodo_diviso.chiavi = nodo_chiavi[:t-1]
    # Se il nodo diviso non è una foglia,
    # spostiamo anche i figli
    if not nodo_diviso.is_leaf:
        nuovo_nodo.figli = nodo_diviso.figli[t:]
        nodo_diviso.figli = nodo_diviso.figli[:t]
    # Inseriamo la chiave mediana nel nodo padre
    nodo.chiavi.insert(i, s)
    nodo.figli.insert(i + 1, nuovo_nodo)
```

8.7 Cancellazione in B-alberi

Anche l'algoritmo di cancellazione di una chiave in un B-albero è progettato per mantenere l'albero bilanciato e ordinato, rispettando le proprietà fondamentali di questa struttura. Si articola nei seguenti passaggi principali:

- **Ricerca della chiave da cancellare:**
  - Si inizia dalla radice e si percorre l'albero seguendo i figli appropriati fino a individuare il nodo contenente la chiave da cancellare.
  - La chiave può trovarsi in un nodo interno o in un nodo foglia. La gestione della cancellazione varia a seconda di dove si trova la chiave.
  - *Cancellazione in un nodo interno:*
    La chiave da cancellare viene sostituita con il suo predecessore (che è la chiave più a destra nella foglia più a destra del sottoalbero immediatamente alla sinistra). Dopo la sostituzione, il problema si riduce alla cancellazione in un nodo foglia.
  - *Cancellazione nel nodo foglia:*
    La chiave da cancellare viene semplicemente rimossa dalla lista delle chiavi del nodo.
    Se il nodo foglia contiene ora meno di $t-1$ chiavi (violando il requisito del grado minimo), è necessario applicare

un processo di **spostamento chiavi** o **fusione nodi** per ristabilire le proprietà del B-albero.

- **Spostamento chiavi o fusione nodi:**

  Si tenta di ristabilire la proprietà dei B-alberi tramite un'operazione di spostamento delle chiavi; se questa si rivela impossibile, allora si procede con un'operazione di fusione dei nodi.

  - *Spostamento chiavi:*

    Se il nodo ha un fratello sinistro con almeno t chiavi, la chiave più grande del fratello viene spostata nel nodo genitore, mentre una chiave del genitore viene trasferita nel nodo sottodimensionato (**spostamento di chiavi verso destra**). In alternativa, se il fratello destro ha almeno t chiavi, la chiave più piccola del fratello viene spostata nel nodo genitore, e una chiave del genitore viene trasferita nel nodo sottodimensionato (**spostamento di chiavi verso sinistra**). Questo è sufficiente a ristabilire le proprietà di B-albero.

  - *Fusione di nodi:*

    Se nessun fratello ha abbastanza chiavi, il nodo sottodimensionato viene fuso con un fratello adiacente, dando la preferenza a quello sinistro, includendo una chiave dal nodo genitore.

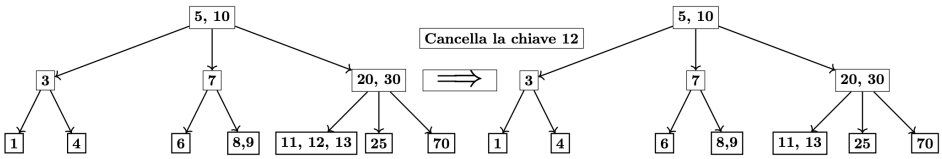
    Dopo la fusione, il nodo genitore potrebbe necessitare a sua volta di spostamento o fusione.

Questo processo può propagarsi verso l'alto, raggiungendo eventualmente la radice. Se la radice perde tutte le sue chiavi, l'albero si riduce di altezza.

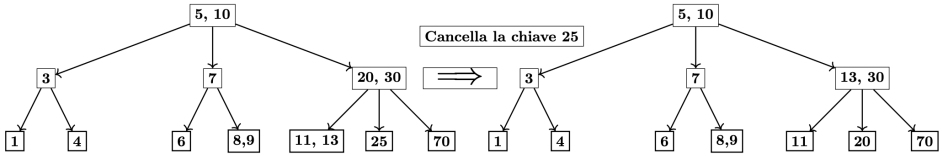
Il costo computazionale della cancellazione di una chiave in un B-albero è $O(t \log_t n)$, infatti:

- La ricerca della chiave da cancellare richiede un tempo $O(t \log_t n)$, grazie alla struttura bilanciata del B-albero, la cui altezza è proporzionale a $\log_t n$;
- Le operazioni di spostamento di chiavi o di fusione di nodi avvengono a livello di un singolo nodo e richiedono tempo $O(t)$ (ricordare che l'operazione di fusione di due sequenze ordinate si esegue in tempo lineare rispetto al numero di chiavi);
- La propagazione verso l'alto delle operazioni di ribilanciamento può coinvolgere al più $O(\log_t n)$ livelli, poiché questa è l'altezza massima del B-albero.

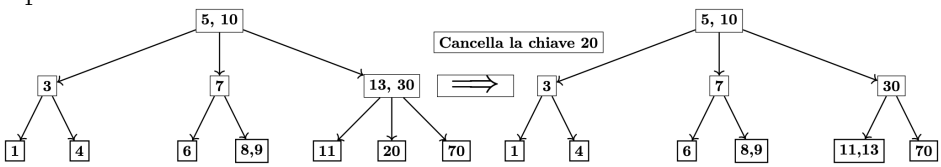
Nelle figure che seguono, viene mostrata l'evoluzione della struttura di un B-albero con $t = 2$ a seguito della cancellazione di alcune delle



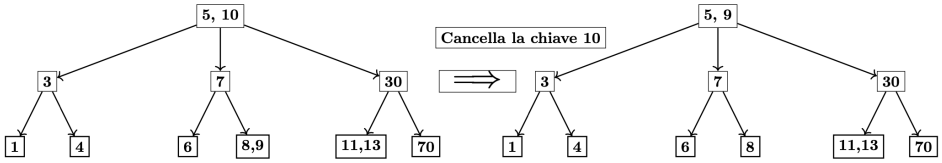
La chiave da cancellare (12) si trova in un nodo foglia e la sua rimozione non provoca la sottodimensione del nodo.



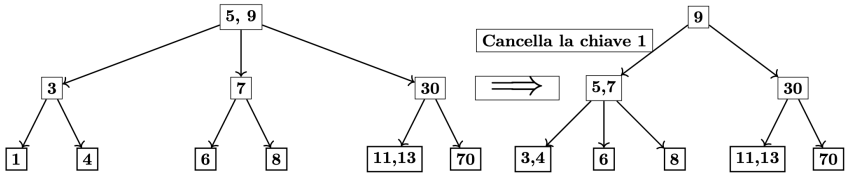
La chiave da cancellare (25) provoca la sottodimensione del nodo foglia che la contiene. Il problema viene risolto mediante uno spostamento di chiavi dal fratello sinistro.



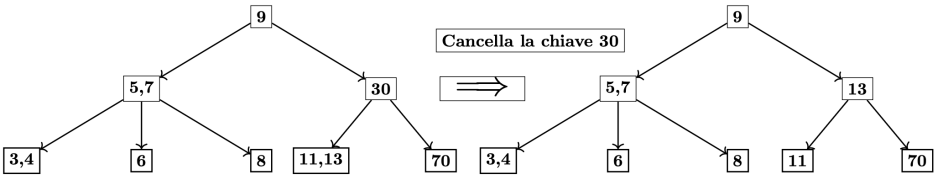
La rimozione della chiave 20 provoca la sottodimensione del nodo foglia che la contiene. Poiché non è possibile effettuare uno spostamento di chiavi tra fratelli, il problema viene risolto tramite una fusione del nodo con il fratello sinistro.



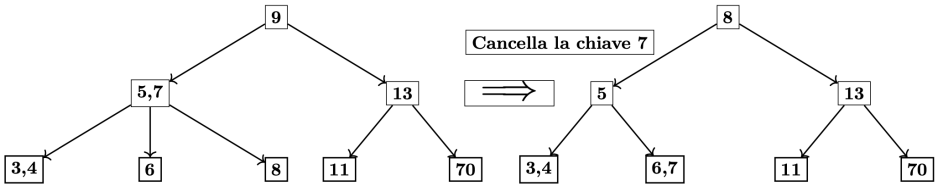
La chiave 10 da cancellare non si trova in un nodo foglia, quindi viene sostituita con il suo predecessore massimo. Successivamente, si procede con la cancellazione di quest'ultimo dalla sua foglia, senza che tale operazione comporti il sottodimensionamento della foglia.



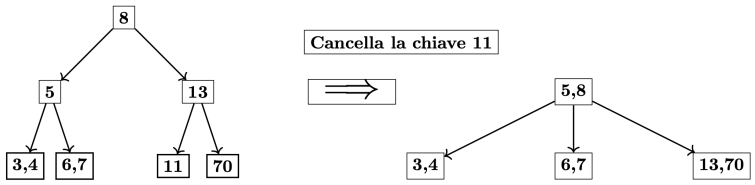
La rimozione della chiave 1 provoca la sottodimensione del nodo foglia che la contiene. Poiché non è possibile effettuare uno spostamento di chiavi tra fratelli, si procede con la fusione del nodo con il fratello destro. Il nodo padre diventa sottodimensionato e, non essendo possibile lo spostamento tra fratelli, si procede con una fusione tra nodo padre e fratello destro.



La chiave da cancellare (30) non si trova in un nodo foglia, quindi viene sostituita con il suo predecessore. Successivamente, si procede con la cancellazione di quest'ultimo dalla sua foglia, senza che tale operazione comporti il sottodimensionamento della foglia.



La chiave 9 da cancellare non si trova in un nodo foglia, quindi viene sostituita con il suo predecessore. Successivamente, si procede con la cancellazione di quest'ultimo dalla sua foglia. Questa operazione provoca il sottodimensionamento della foglia, non essendo possibile lo spostamento tra fratelli si procede con la fusione fusione del nodo con il fratello sinistro.



La rimozione della chiave 11 provoca la sottodimensione del nodo foglia che la contiene. Poiché non è possibile lo spostamento di chiavi tra nodi fratelli, si procede con la fusione del nodo con il fratello destro. Il nodo padre diventa sottodimensionato e, non essendo possibile lo spostamento tra fratelli, si procede con la fusione con il fratello sinistro. La radice diventa sottodimensionata e viene

cancellata. La nuova radice dell'albero è il nodo puntato dal suo unico figlio. In questo caso, a seguito della cancellazione si ha anche un decremento dell'altezza dell'albero.

Per completezza, di seguito riportiamo il codice Python necessario all'implementazione della funzione di cancellazione di una chiave nel B-albero. La funzione cancella si avvale delle sottofunzioni gestisci_chiave e elimina_da_foglia. Seguono infine i listati con i codici per le funzioni sposta e fondi necessarie allo spostamento delle chiavi e alla fusione dei nodi.

```
def cancella(radice,chiave):
    # elimina la chiave dal B-albero se presente e
     # restituisce la radice che può  variare
     # a seguito della cancellazione'''
     chiave = gestisci_chiave(radice, chiave)
     if chiave:
         elimina_da_foglia(radice, chiave)
     # restituisce la nuova radice nel caso in cui
     # la vecchia radice non abbia più chiavi
     if len(radice.chiavi)==0 and len(radice.figli)!=0:
         radice = radice.figli[0]
     return radice
```

```
def gestisci_chiave(nodo, chiave):
   # cerca una chiave in un B-albero, gestendo tre casi:
   #     - Se la chiave si trova in una foglia:
   #        restituisce la chiave stessa.
   #      - Se la chiave si trova in un nodo interno:
   #           sostituisce la chiave con il suo predec.
   #              e lo restituisce.
   #       - Se la chiave non c'è: restituisce None.
    while not nodo.is_leaf:
        i = 0
        while i < len(nodo.chiavi) and
                      chiave > nodo.chiavi[i]:
            i += 1
        # Se la chiave è trovata nel nodo corrente
        if i < len(nodo.chiavi) and
                    chiave == nodo.chiavi[i]:
            # Trova il predecessore e sostituisce
            nodo1 = nodo.figli[i]
            while not nodo1.is_leaf:
                nodo1 = nodo1.figli[-1]
            nodo.chiavi[i] = nodo1.chiavi[-1]
            return nodo1.chiavi[-1]
        nodo = nodo.figli[i]
    # Se in una foglia, verifica se la chiave c'è
    return chiave if chiave in nodo.chiavi else None
```

```
def elimina_da_foglia(nodo, chiave):
    # cancella la chiave situata in una foglia:
    # ricorsivamente localizza la foglia con la chiave
    #  da cancellare. La elimina e risale ricorsivamente
    # nell'albero; se il nodo è sottodimensionato tenta
    # uno spostamento di chiavi; se non è possibile,
    # procede con la fusione con un fratello.
    t = nodo.t
    if nodo.is_leaf:
        nodo.chiavi.remove(chiave)
    else:
        i=0
        while i<len(nodo.chiavi) and
                        chiave > nodo.chiavi[i]:
            i+=1
        elimina_da_foglia(nodo.figli[i], chiave)
        if len(nodo.figli[i].chiavi)<t-1:
            # prova  ad eseguire la funziona sposta,
            # se non è possibile esegui la fusione
            if not sposta(nodo,i):
                fondi(nodo, i)
```

```
def sposta(nodo, i):
    # cerca di ribilanciare con spostam. nodo.figlio[i].
    # Dà True se il ribilanciamento è riuscito,
    # False altrimenti.
    f=nodo.figli[i]
    # Prova a spostare una chiave dal fratello sinistro
    if i>0 and len(nodo.figli[i-1].chiavi)>t-1:
        fs = nodo.figli[i-1]
        f.chiavi.insert(0, nodo.chiavi[i-1])
        # La chiave del nodo padre scende
        # La chiave del fratello sale
        nodo.chiavi[i-1]=fs.chiavi.pop()
        #Se non è una foglia, sposta il figlio corrisp.
        if not f.is_leaf:
            f.figli.insert(0, fs.figli.pop())
        return True
    # Prova a spostare una chiave dal fratello destro
    if i+1<len(nodo.figli) and
              len(nodo.figli[i+1].chiavi)>t-1:
        fd = nodo.figli[i+1]
        f.chiavi.append(nodo.chiavi[i])
        nodo.chiavi[i]=fd.chiavi.pop(0)
        if not f.is_leaf:
            f.figli.append(fd.figli.pop(0))
        return True
    return False
```

```
def fondi(nodo, i):
    # Fonde  nodo.figli[i] con un suo fratello,
    # dando la preferenza al fratello sinistro
    f=nodo.figli[i] # Il figlio da fondere
    if i==0:
        # Fondi con il fratello destro
        fd=nodo.figli[i+1]
        # Porta la chiave dal nodo corrente al figlio
        f.chiavi.append(nodo.chiavi[i])
        # Aggiungi le chiavi e i figli del fratello des.
        f.chiavi.extend(fd.chiavi)
        f.figli.extend(fd.figli)
        # Rimuovi la chiave dal nodo corrente
        nodo.chiavi.pop(i)
        # Rimuovi il fratello des. dalla lista dei figli
        nodo.figli.pop(i+1)
    else:
        # Fondi con il fratello sinistro
        fs=nodo.figli[i-1]
        # Porta la chiave dal nodo corrente
        # al fratello sinistro
        fs.chiavi.append(nodo.chiavi[i-1])
        # Aggiungi le chiavi e i figli del figlio corr.
        fs.chiavi.extend(f.chiavi)
        fs.figli.extend(f.figli)
        # Rimuovi la chiave dal nodo corrente
        nodo.chiavi.pop(i-1)
        # Rimuovi il figlio corr. dalla lista dei figli
        nodo.figli.pop(i)
```