Problema della Ricerca

Un dizionario rappresenta un insieme di informazioni suddiviso per elementi ad ognuno dei quali è associata una chiave. Esempio di dizionario è l'elenco telefonico dove la chiave è costituita dalla coppia cognome, nome e gli elementi dalle informazioni quali numero telefonico ed indirizzo. Altro esempio classico di dizionario è rappresentato dal vocabolario della lingua italiana dove ad ogni parola, chiave del dizionario, è associato un significato.

L'insieme minimo di operazioni attuabili su un dizionario è costituito dalle operazioni di ricerca, inserimento e cancellazione dove, la ricerca restituisce un elemento associato ad una chiave, l'inserimento inserisce all'interno del dizionario una nuova coppia, chiave ed elemento, e la cancellazione cancella la coppia, chiave ed elemento.

Tratteremo in questa sezione alcune possibili implementazioni di un dizionario che si distinguono nell'organizzazione delle informazioni e quindi nella complessità delle tre operazioni.

Vedremo ad esempio che una possibile soluzione di implementazione di un dizionario è la lista ordinata e che in tal caso l'operazione di ricerca ha una complessità O(log n) mentre le operazioni di inserimento e cancellazione hanno una complessità O(n). Così come vedremo che gli alberi AVL costituiscono anch'essi una valida alternativa di implementazione di un dizionario e che in tal caso, invece, le tre operazioni hanno tutte complessità O(log n).

E' bene evidenziare, sin da ora, che non esiste in assoluto la migliore soluzione per l'implementazione di un dizionario. La scelta della struttura dati dipende infatti da molteplici fattori ed emerge solo a seguito di un analisi volta ad individuare sia le operazioni maggiormente ricorrenti che il grado di complessità che si vuol affrontare in fase di realizzazione. Ad esempio, se a seguito di un analisi emergesse che durante il proprio ciclo di vita il dizionario non sarà particolarmente dinamico nelle sue componenti, potrebbe risultare preferibile penalizzare le operazioni di cancellazione ed

inserimento a scapito di una maggiore semplicità di realizzazione e quindi magari ricorrere alla implementazione tramite lista ordinata. Caso opposto, invece, è quello in cui ricerca, cancellazione ed inserimento hanno simile grado di ricorrenza. Potrebbe in tal caso risultare opportuno ricorrere ad una implementazione mediante alberi AVL, affrontando quindi una fase di realizzazione più laboriosa ma assicurandosi nel contempo per tutte e tre le operazioni una complessità logaritmica nel numero degli elementi.

Prima di introdurre le possibili implementazioni descriviamo l'ADT Dizionario.

Tipo di dati: Dizionario
Insieme di coppie (Chiave, Elemento)

Operazioni: Inserisci(Chiave k, Elemento e)
Aggiunge una coppia (Chiave, Elemento) al dizionario

Cancella(Chiave k)
Cancella dal dizionario la coppia (Chiave, Elemento) individuata dalla chiave k

Cerca(Chiave k)→Elemento
Restituisce l'elemento del dizionario al quale è associata la chiave k

1. Liste e Bit Vector

Le liste ed i bit vector rappresentano due soluzioni facilmente attuabili per l'implementazione dell'ADT Dizionario. Come vedremo, i bit vector sono applicabili solo in un contesto ben definito.

1.1 Liste

Possiamo implementare un dizionario mediante l'utilizzo dell'ADT lista ordinata o dell'ADT lista non ordinata. In ambedue i casi, il record rappresentante un elemento della lista sarà formato da un campo chiave, da un campo elemento ed infine dal campo rappresentante il puntatore all'elemento successivo.

Nel seguente prospetto sono riportati i tempi di esecuzione delle operazioni di ricerca, inserimento e cancellazione.

	Lista Ordinata	**Lista non ordinata**
Ricerca	Logaritmica nel numero di elementi O(log n)	Lineare nel numero di elementi O(n)
Inserimento	Lineare nel numero di elementi O(n)	Costante O(1)[1]
Cancellazione	Lineare nel numero di elementi O(n)	Lineare nel numero di elementi O(n)

1.2 Bit Vector

I bit vector rappresentano una soluzione applicabile per l'implementazione di un dizionario quando le chiavi assumono valori appartenenti ad un sottoinsieme limitato dell'intero insieme universo ed inoltre tali valori sono rappresentabili mediante numeri interi.

L'implementazione mediante bit vector richiede l'utilizzo di un array di booleani, bit vector, dove lo *i*-esimo bit è posto a *true* se lo *i*-esimo elemento è presente nel dizionario altrimenti è posto a *false*.

[1] L'operazione di inserimento può avvenire inserendo ogni nuova coppia come ultimo elemento della lista

Avremo quindi che per:

- ricercare l'elemento *i*, valutiamo semplicemente se lo *i*-esimo elemento del bit vector è true o false;
- inserire l'elemento *i*: poniamo lo *i*-esimo elemento del bit vector uguale a true;
- cancellare l'elemento *i*, poniamo lo *i*-esimo elemento del bit vector uguale a false.

Considerando che l'accesso allo *i*-esimo elemento di un array ha tempo costante rispetto al numero di elementi in esso contenuti, abbiamo che le operazioni di ricerca, inserimento e cancellazione hanno tutte complessità costante.

Esempio

Si prendi in considerazione la gestione delle prenotazioni delle camere di un albergo. Ad ogni camera è associato un valore intero rappresentante il numero di stanza. E' possibile creare, quindi, un vettore di bit avente dimensione pari al numero di stanze dell'albergo e:

- *quando inseriamo una prenotazione per la stanza i poniamo lo i-esimo elemento del vettore uguale a true;*
- *quando cancelliamo una prenotazione per la stanza i poniamo lo i-esimo elemento del vettore uguale a false;*
- *quando si deve verificare se la stanza i è prenotata valutiamo il valore assunto dallo i-esimo elemento del vettore booleno.*

2. Alberi Binari di Ricerca (BST)

Vedremo in questo paragrafo come organizzare le informazione del dizionario mediante la struttura dati Albero Binario di Ricerca.

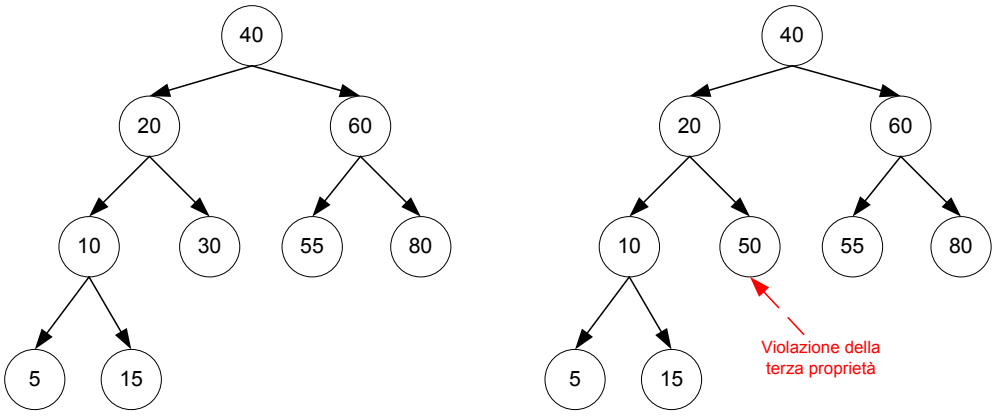
Crediamo opportuno sottolineare che, anche se all'interno del corso tratteremo gli alberi binari di ricerca solamente nello studio dell'ADT Dizionario, non bisogna ad ogni modo commettere l'errore di confinare il loro utilizzo esclusivamente all'interno di tale ambito. Gli alberi binari di ricerca sono strutture dati valide oltre che per la realizzazione degli operatori base dell'ADT Dizionario, ricerca, inserimento e cancellazione, anche per l'implementazione di altri operatori quali la ricerca del successivo, la ricerca del predecessore, la ricerca del massimo o la ricerca del minimo, operatori applicabili ad ADT di altro tipo quali ad esempio l'ADT Coda a Priorità.

Definizione: un ***albero binario di ricerca*** è un albero binario che soddisfa le seguenti proprietà:

1. in ogni nodo dell'albero è contenuta la coppia (chiave, elemento);
2. le chiavi sono estratte da un insieme totalmente ordinato;
3. ogni chiave del sottoalbero sinistro del nodo *v* è minore o al massimo uguale alla chiave contenuta nel nodo *v*;
4. ogni chiave del sottoalbero destro del nodo *v* è maggiore o al massimo uguale alla chiave contenuta nel nodo *v*.

Esempio

Solo la prima raffigurazione rappresenta un albero binario di ricerca.



Nota

Una caratteristica dell'albero di ricerca è che il path della visita inorder è rappresentato dalle chiavi dell'albero riportate in ordine crescente. Ad esempio si consideri l'albero binario di ricerca dell'esempio sopra si ha che il path della visita inorder è: 5, 10, 15, 20, 30, 40, 55, 60, 80.

Ricerca

Facendo leva sulle proprietà 3 e 4, appunto chiamate proprietà di ricerca, ricerchiamo una chiave all'interno di un albero binario di ricerca seguendo una metodologia che richiama l'algoritmo di ricerca binaria.

Sia *v* un nodo dell'albero e *k* il valore della chiave da ricercare, diamo luogo ai seguenti passi:

1. se la chiave di *v* è uguale a *k* allora stop: chiave trovata!
2. se la chiave di *v* è maggiore di *k* allora continuiamo la ricerca sottoalbero sinistro
3. se la chiave di *v* è minore di *k* allora continuiamo la ricerca sottoalbero destro

Alberi binari di Ricerca- Algoritmo di ricerca chiave

```
Cerca( Chiave k, Nodo v ) → Nodo
{
        Nodo    r

        se  ( v == NULL ) allora ritorna NULL

        se (  v.chiave ==  k ) allora ritorna v

        se ( v.chiave >  k )
            allora r =  v.sinistro
            altrimenti r =v.destro

        ritorna Cerca( k, r )
}
```

Analizziamo la complessità dell'algoritmo. Esso scorre di livello in livello l'albero binario di ricerca. Il peggiore dei casi è rappresentato dalla discesa lungo un intero verso dell'albero, fino ad arrivare ad una foglia. Tale caso occorre quando l'elemento non è trovato ed al suo verificarsi l'algoritmo esegue quindi un numero di passi direttamente proporzionali con l'altezza dell'albero. Pertanto se indichiamo con *h* l'altezza dell'albero binario di ricerca, possiamo dire che l'algoritmo nel caso pessimo, elemento non trovato, ha una complessità lineare rispetto ad *h*: *O(h)*.

Inserimento

In un albero binario, inseriamo un nuovo nodo sempre come foglia. Diamo luogo ad un inserimento in considerazione dei seguenti macro passi:

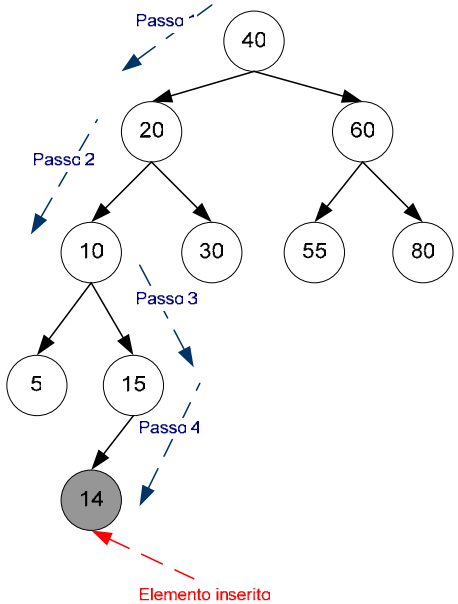
Ricerca del giusto padre, individuiamo il nodo *v* che possiede le proprietà adatte per essere genitore del nuovo nodo. La ricerca del giusto padre avviene scendendo lungo l'albero, con la stessa metodologia utilizzata per l'operazione di ricerca, fin quando non si incontra un nodo *v* che non ha figlio sinistro ed ha la chiave maggiore di quella dell'elemento da inserire oppure un nodo *v* che non ha figlio destro ed ha la chiave minore di quella dell'elemento da inserire.

Inserimento nodo: trovato il giusto padre inseriamo il nuovo nodo come figlio destro o sinistro nel rispetto delle proprietà di ricerca

Analizziamo la complessità. Poiché l'algoritmo scorre lungo un intero verso l'albero binario di ricerca, si ha che la complessità anche in questo caso è lineare rispetto all'altezza dell'albero: *O(h)*.

Esempio

Inserimento elemento con chiave 14.



Cancellazione

Prima di procedere nella cancellazione di un nodo all'interno di un albero binario di ricerca vediamo le seguenti due procedure poste al servizio dell'operazione di cancellazione.

Ricerca del massimo

Le proprietà di ricerca garantiscono che per l'individuazione della chiave più grande presente in un albero binario di ricerca dobbiamo scendere, quanto più possibile, lungo il verso destro dell'albero.

Alberi binari di Ricerca- Ricerca del massimo

```
max( Nodo u ) →Nodo
{
    Nodo v

    v = u
    while( v.destro != NULL )
        v = v.destro

    ritorna v
}
```

L'algoritmo di ricerca del massimo ha, nel caso pessimo, una complessità lineare rispetto all'altezza dell'albero.

Ricerca del predecessore

Indichiamo con il termine di predecessore di un nodo *v* quel nodo contenente come chiave il massimo dell'insieme delle chiavi più piccole di quella contenuta in *v*.
In un albero binario di ricerca il predecessore di un nodo *v* è:

1. il massimo del sottoalbero sinistro, se *v* ha un figlio sinistro;
2. il più basso antenato di *u* il cui figlio destro è anch'esso antenato di *u* ("*risali l'albero fin quando non incontri una svolta a sinistra*"), altrimenti;

Alberi binari di Ricerca- Ricerca del predecessore

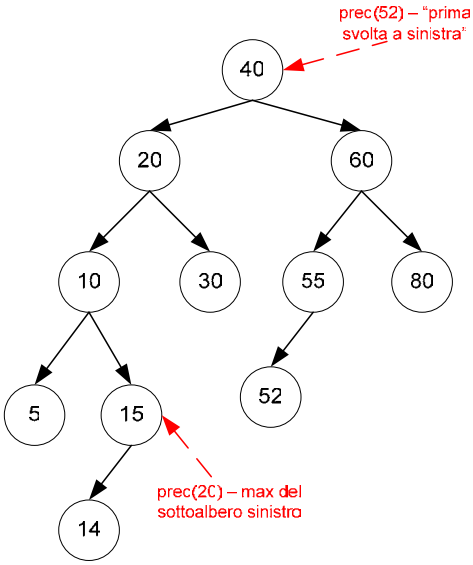
```
predecessore( Nodo u ) →Nodo
{
    se ( u ha figlio sinistro )
        allora ritorna max( u )

    while( padre(u) != NULL  e u e figlio sinistro del padre )
        u = padre u

    ritorna il padre u
}
```

L'algoritmo di ricerca del predecessore risale l'albero e pertanto esso, nel caso pessimo, ha complessità lineare rispetto all'altezza dell'albero .

Esempio – Ricerca del predecessore



Siamo pronti per analizzare l'operazione di cancellazione. Sia *u* il nodo dell'albero di ricerca da cancellare, si possono verificare i seguenti tre casi:

1. *u* è una foglia: in tal caso distacchiamo semplicemente la foglia dall'albero;
2. *u* ha un unico figlio (figura 2.1) e sia *v* l'unico figlio di *u*
   a. se *u* è radice allora *v* diviene la nuova radice dell'albero;
   b. se *u* non è radice allora sia *w* il padre di *u*, sostituiamo l'arco (*w*,*u*) con l'arco (*w*,*v*);

3. *u* ha due figli (figura 2.2): in tal caso sia *v* il predecessore di *u*. Poiché *u* ha due figli il predecessore sarà il massimo del sottoalbero sinistro di *u* e pertanto *v* sarà una foglia oppure un nodo interno avente al max un solo figlio: quello sinistro[2]. La cancellazione di *u* pertanto può avvenire copiando la chiave di *v* in *u* ed eliminando *v* cadendo per quest'ultima operazione nei due casi precedenti.

La cancellazione al pari delle altre due operazioni, ricerca ed inserimento, ha una complessità proporzionale all'altezza dell'albero poiché nel caso pessimo potremmo muoverci su l'intera altezza dell'albero alla ricerca del predecessore.

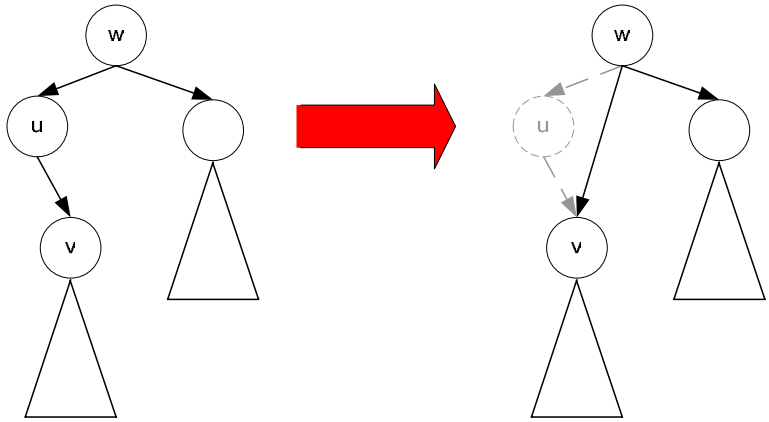


Figura 2.1 –Eliminazione nodo con un solo figlio

[2] Se *v* avesse una figlio destro esisterebbe una chiave più grande di *v* e quindi *v* non rappresenterebbe il massimo del sottoalbero sinistro di *u*.

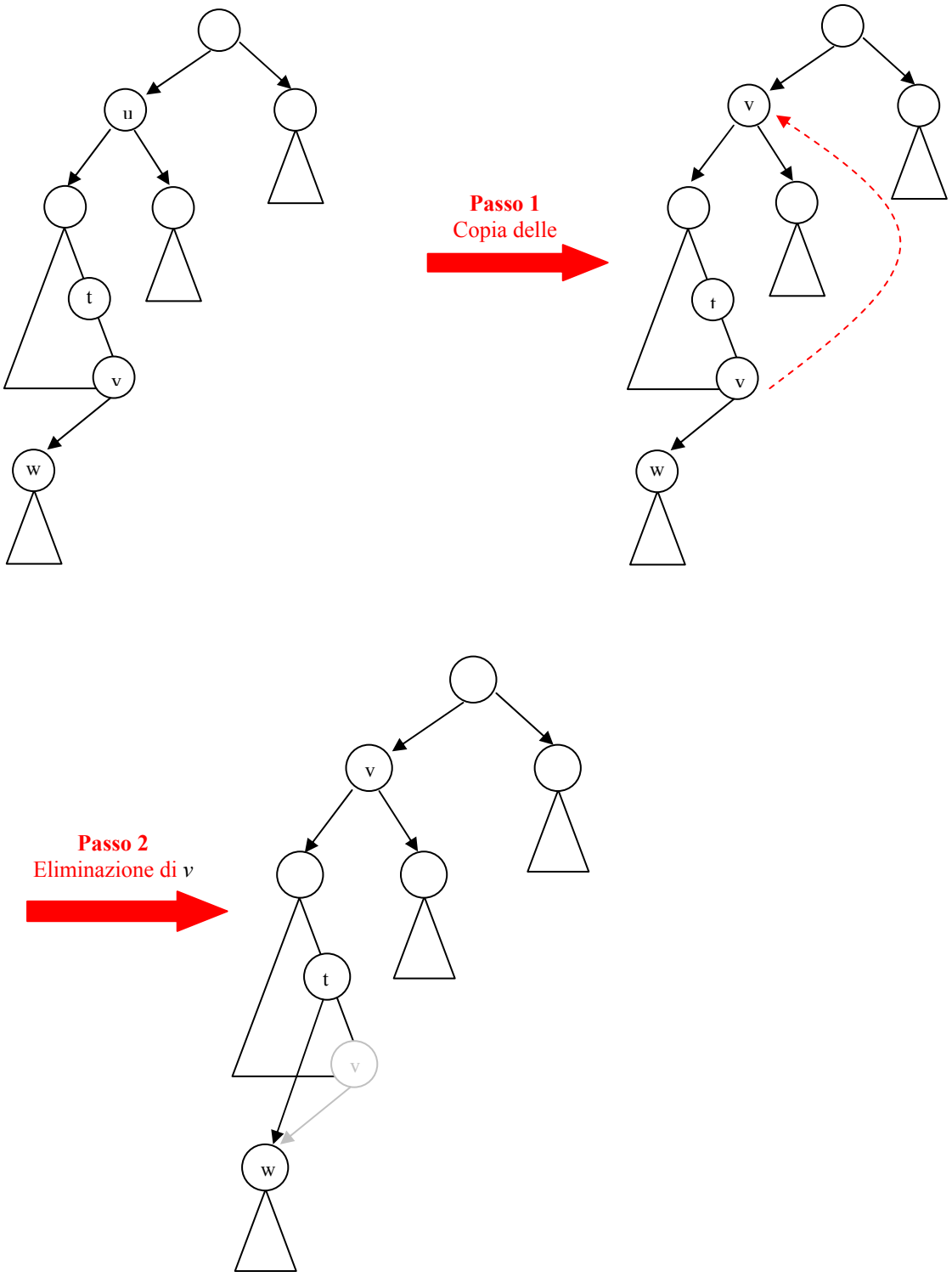
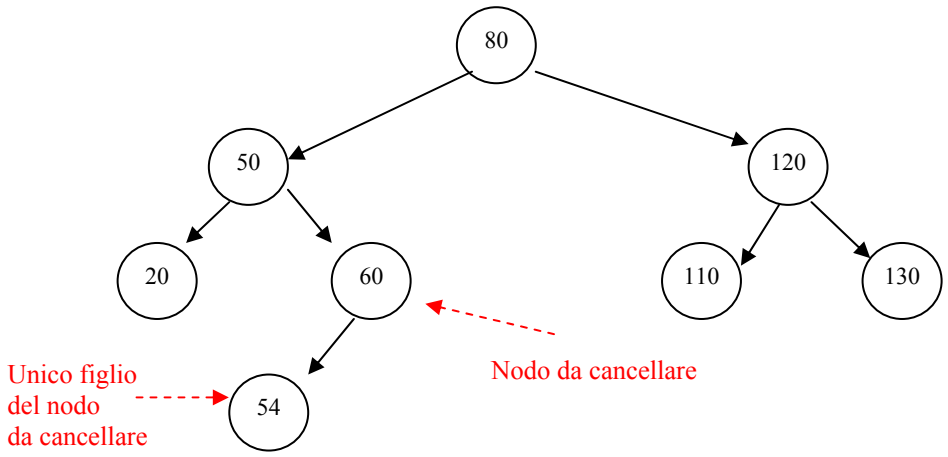
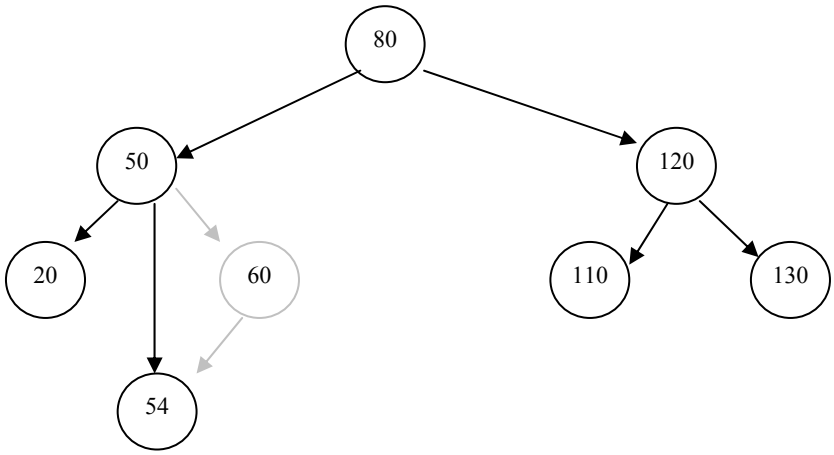


Figura 2.2 – Eliminazione nodo con due figli

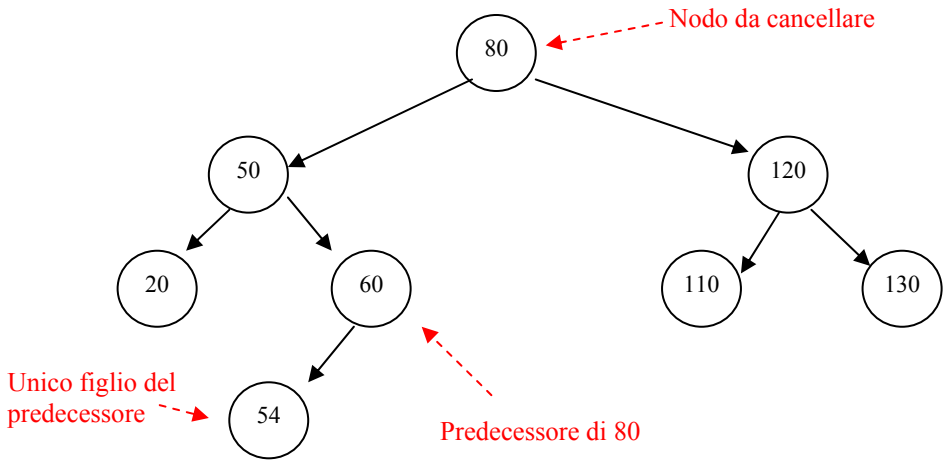
Esempio

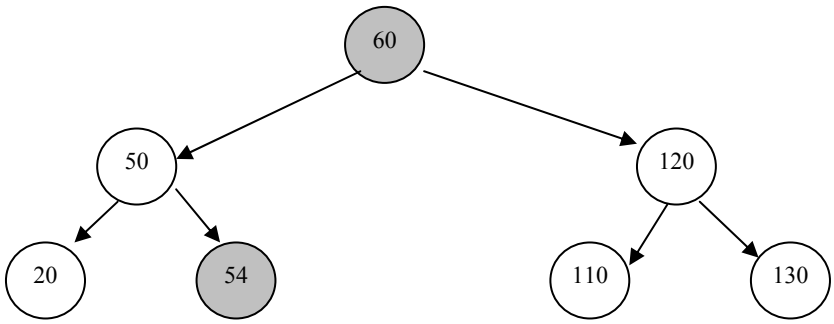
Raffiguriamo di seguito la cancellazione di un nodo con un figlio.





Vediamo infine la cancellazione di un nodo con due figli



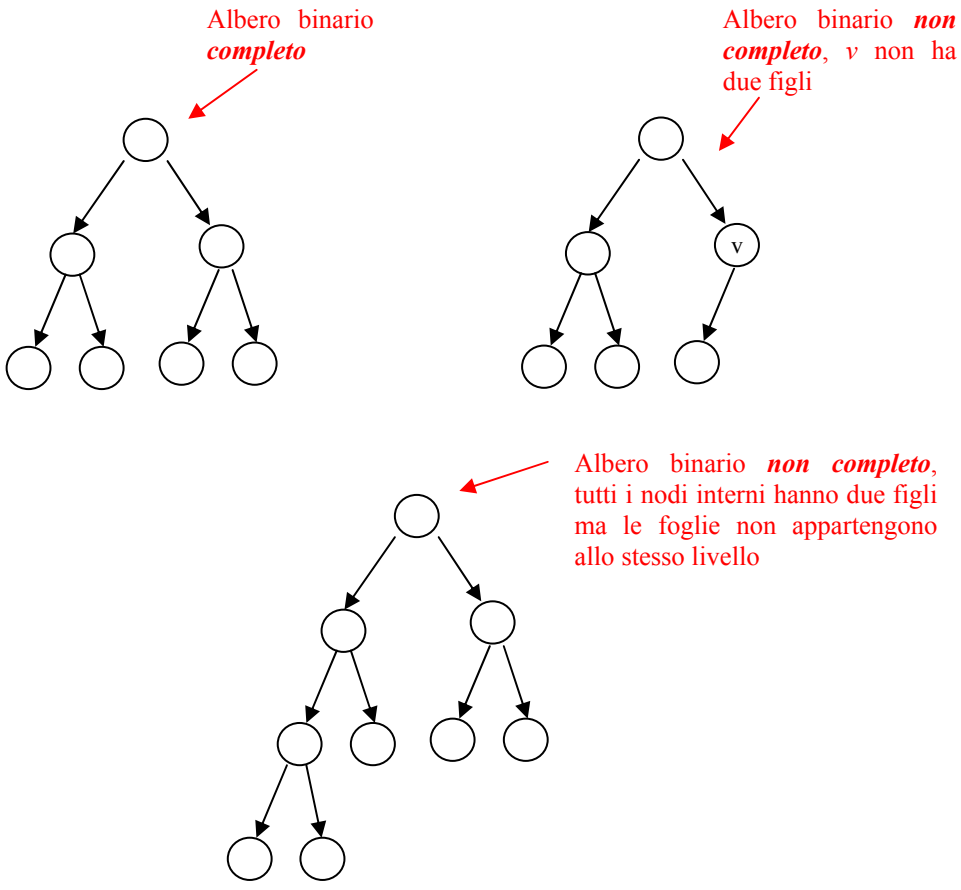


2.1 Analisi dell'altezza di un albero binario di ricerca

Come visto precedentemente, mediante la struttura dati alberi binari di ricerca siamo in grado di implementare algoritmi di inserimento, cancellazione e ricerca aventi complessità lineare con l'altezza dell'albero. L'analisi che ci accingiamo a svolgere all'interno del presente paragrafo ha come obiettivo quello di esprimere tale complessità in termini di numero di nodi del grafo ovvero in termini di numero di elementi presenti nel dizionario. Vediamo prima alcune nozioni propedeutiche all'analisi.

Definizione: un albero binario si definisce ***completo*** se ogni nodo interno ha due figli e tutte le foglie appartengono allo stesso livello.

Esempio



Si dimostra il seguente teorema valido per gli alberi binari completi.

Teorema 2.1.1. *Sia T un albero binario completo. Si indichi con $n_{foglie}(T)$ il numero di foglie dell'albero T e con h l'altezza dell'albero, allora si ha che: $n_{foglie}(T) = 2^h$.*

Dimostrazione. Il teorema si dimostra per induzione. Sia $h=0$, allora T è formato dalla sola radice che è pertanto anche l'unica foglia per cui: $n_{foglie}(T) = 1 = 2^0$.
Poniamo quindi vero il teorema per un albero completo, che indichiamo con T_1, avente altezza $h-1$. Allora $n_{foglie}(T_1) = 2^{h-1}$. Costruiamo quindi, a partire da T_1, un albero completo di altezza h che indichiamo con T. Affinché T sia completo ad ogni foglia di T_1 dobbiamo aggiungere due figli. Tali figli rappresenteranno le foglie di T. Pertanto le foglie di T saranno: $n_{foglie}(T) = 2 * n_{foglie}(T_1) = 2 * 2^{h-1} = 2^h$.

Estendendo il risultato del teorema a tutti i nodi dell'albero abbiamo che:

Teorema 2.1.2. *Sia T un albero binario completo. Sia n il numero di nodi di T e h l'altezza di T allora $n \geq 2^h$ cioè $h \leq \log n$.*

Analisi della complessità - Caso 1: albero di altezza minima

La definizione di albero binario completo ci aiuta nella nostra analisi quando l'albero binario di ricerca ha altezza minima. Siano n il numero di elementi del dizionario, l'albero binario di ricerca contenente le n coppie ed avente altezza minima è dato da un albero binario di ricerca di altezza h completo almeno fino a livello $h-1$. In tal caso abbiamo, dal teorema 2.1.2 che: $h = (h-1)+1 \leq \log(n-1)+1$.

Pertanto, avendo le operazioni di ricerca, inserimento e cancellazione tutte complessità $O(h)$ possiamo affermare che tale complessità espressa in termini di numero di nodi è $O(\log n)$ nel caso in cui l'albero ha altezza minima.

Analisi della complessità - Caso 2: albero di altezza massima

Caso opposto a quello appena esaminato è dato da un albero completamente sbilanciato, un albero avente altezza uguale al numero di nodi meno uno[3].
In tal caso la complessità degli algoritmi di ricerca, cancellazione ed inserimento è banalmente proporzionale al numero di elementi del dizionario: $O(n)$.

[3] Si cade in tale ipotesi quando l'inserimento degli elementi all'interno dell'albero è ordinato.

Analisi della complessità - Caso 2: altezza media

Presupposto dell'analisi dell'altezza media di un albero binario di ricerca è l'inserimento in ordine puramente casuale dei nodi all'interno dell'albero. Tutti gli elementi del dizionario hanno uguale probabilità di essere scelti e quindi inseriti.

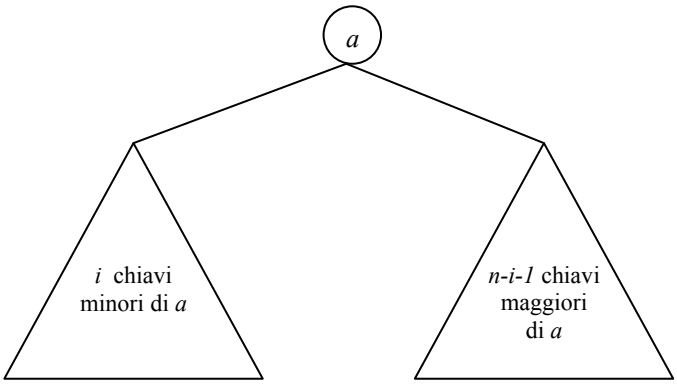
Sulla base di tale ipotesi dimostreremo che l'altezza dell'albero è *O(log n)* e pertanto nel caso medio le tre operazioni del dizionario hanno tutte complessità *O(log n)*.

Teorema 2.1.3. *sia T un albero binario di ricerca ottenuto mediante una serie di inserimenti di n coppie (chiave, elemento). Sia inoltre l'ordine di inserimento degli elementi puramente casuale. Si indichi infine con P(n) l'altezza media dell'albero T, allora si ha che:*

$$P(n) = O(\log n)$$

Dim: indichiamo con $\{a_1, ..., a_n\}$ l'insieme degli elementi da inserire all'interno dell'albero. Affinchè l'ordine di inserimento sia ritenuto puramente casuale dobbiamo avere che la probabilità di scelta degli elementi sia equidistribuita, ovvero che essa sia uguale ad 1/n per ogni elemento.

Indichiamo con *a* il primo elemento che viene prelevato dalla lista per essere inserito all'interno dell'albero. Essendo la probabilità di scelta equi distribuita abbiamo che la probabilità che *a* sia uguale ad un qualunque a_i è pari ad *1/n* per ogni $i=1,...,n$. Al termine dell'inserimento di tutti gli elementi è lecito raffigurare l'albero binario di ricerca nel seguente modo:



dove nel sottoalbero sinistro sono contenuti gli elementi associati alle chiavi più piccole di *a* e nel sottoalbero destro gli elementi associati alle chiavi più grandi di *a*.

Indichiamo quindi con *i* il numero di chiavi più piccole di *a* e con *n-i-1* quelle più grandi di *a*. Indichiamo inoltre con *P(i)* e con *P(n-i-1)* rispettivamente le altezze medie del sottoalbero sinistro e destro quando l'elemento a_i è radice dell'albero.

Abbiamo che l'altezza media dell'intero albero, *P(n)*, sarà calcolabile a partire dalla seguente formula:

$$P_i(n) = \frac{i}{n}(P(i)+1) + \frac{n-i-1}{n}(P(n-i-1)+1) + \frac{1}{n}$$

dove:

$P_i(n)$: è l'altezza media dell'intero albero contenente n elementi ed avente la radice $a= a_i$.

i/n: è la probabilità di aver "pescato" i elementi più piccoli di a dopo l'inserimento di a;

$P(i)+1$: è l'altezza media del sottoalbero sinistro più il nodo a;

$(n-i-1)/n$: è la probabilità di aver "pescato" $n-i-1$ elementi più grandi di a dopo l'inserimento di a;

$P(n-i-1)$: è l'altezza media del sottoalbero destro più il nodo a;

$1/n$: è la probabilità di aver "pescato" a come primo elemento.

Pertanto considerando che la probabilità che un elemento sia radice dell'albero è pari ad $1/n$ abbiamo:

$$P(n) = \frac{1}{n}\sum_{i=0}^{n-1} P_i(n) = \frac{1}{n}\sum_{i=o}^{n-1} \frac{i}{n}(P(i)+1) + \frac{n-i-1}{n}(P(n-i-1)+1) + \frac{1}{n} =$$

$$= \frac{1}{n}\sum_{i=o}^{n-1} \frac{i}{n}P(i) + \frac{i}{n} + \frac{n-i-1}{n}P(n-i-1) + \frac{n-i-1}{n} + \frac{1}{n} =$$

$$= \frac{1}{n}\sum_{i=o}^{n-1} \frac{i}{n}P(i) + \frac{n-i-1}{n}P(n-i-1) + \frac{i+n-i-1+1}{n} =$$

$$= \frac{1}{n}\sum_{i=o}^{n-1} \frac{i}{n}P(i) + \frac{n-i-1}{n}P(n-i-1) + 1 =$$

$$= \frac{1}{n}\left(n + \sum_{i=o}^{n-1} \frac{i}{n}P(i) + \frac{n-i-1}{n}P(n-i-1) \right) =$$

$$= \frac{1}{n}\left(n + \frac{1}{n}\sum_{i=o}^{n-1} i\,P(i) + (n-i-1)\,P(n-i-1) \right) =$$

$$= 1 + \frac{1}{n^2}\sum_{i=o}^{n-1} i\,P(i) + (n-i-1)\,P(n-i-1) =$$

$$= 1 + \frac{1}{n^2}\sum_{i=o}^{n-1} 2\,i\,P(i) =$$

$$= 1 + \frac{2}{n^2}\sum_{i=o}^{n-1} i\,P(i) = 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} i\,P(i)$$

Quest'ultima uguaglianza è giustificata dal fatto che per i uguale a zero $iP(i)=0$.
Ricapitolando abbiamo che:

$$P(n) = 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} i\,P(i)$$

Ora, procedendo per induzione su n, dimostriamo che:

$$P(n) \le 1 + 4\log n \qquad (1)$$

Poniamo *n=1*. In tal caso l'altezza media di un albero formato da un nodo è obbligatoriamente uguale ad uno e pertanto P(1) = 1 = 1 + 4 log1.

Poniamo vera l'ipotesi per tutti i valori minori di *n*, avremo quindi che:

$$P(n) = 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} i\,P(i) \quad \leq$$

$$\leq \quad 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} i\,(1 + 4\log i) \quad =$$

$$= 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} i + 4i\,\log i \quad =$$

$$= 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} i + \frac{2}{n^2}\sum_{i=1}^{n-1} 4i\,\log i$$

Ora essendo $\sum_{i=1}^{n-1} i \quad = \quad \frac{n\,(n-1)}{2} \quad = \quad \frac{n^2 - n}{2} \leq \frac{n^2}{2}$ si ha che:

$$P(n) \leq 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} 4i\,\log i + \frac{2}{n^2}\,\frac{n^2}{2} \quad =$$

$$= \quad 2 + \frac{2}{n^2}\sum_{i=1}^{n-1} 4i\,\log i \quad =$$

$$= \quad 2 + \frac{8}{n^2}\sum_{i=1}^{n-1} i\,\log i$$

Analizziamo ora gli indici della sommatoria, dividiamoli in due range: (1, ..., $\lceil n/2 \rceil$-1) e ($\lceil n/2 \rceil$, ..., n-1) e riscriviamo la disuguaglianza.

$$P(n) \leq 2 + \frac{8}{n^2}\left(\sum_{i=1}^{\left\lceil \frac{n}{2} \right\rceil - 1} i\,\log i + \sum_{i=\left\lceil \frac{n}{2} \right\rceil}^{n-1} i\,\log i \right)$$

Essendo ora i due logaritmi calcolati sul valore incrementale di *i* possiamo affermare che:

i) $\sum_{i=1}^{\left\lceil \frac{n}{2} \right\rceil - 1} i\,\log i \leq \sum_{i=1}^{\left\lceil \frac{n}{2} \right\rceil - 1} i\,\log \frac{n}{2}$ poichè il valore di *i* è sempre minore di *n/2*

ii) $\sum_{i=\left\lceil \frac{n}{2} \right\rceil}^{n-1} i\,\log i \quad \leq \quad \sum_{i=\left\lceil \frac{n}{2} \right\rceil}^{n-1} i\,\log n$ poichè il valore di *i* è sempre minore di *n*

e pertanto riscrivere la disuguaglianza come:

$$P(n) \le 2 + \frac{8}{n^2}\left(\sum_{i=1}^{\left\lceil \frac{n}{2} \right\rceil - 1} i \log \frac{n}{2} + \sum_{i=\left\lceil \frac{n}{2} \right\rceil}^{n-1} i \log n \right)$$

Analizziamo ora separatamente le due sommatorie considerando i due diversi casi: n è pari ed n è dispari.

n pari

$$\sum_{i=1}^{\left\lceil \frac{n}{2} \right\rceil - 1} i = \sum_{i=1}^{\frac{n}{2}-1} i = \frac{\left(\frac{n}{2}-1\right)\left(\frac{n}{2}\right)}{2} \le \frac{\left(\frac{n}{2}\right)\left(\frac{n}{2}\right)}{2} = \frac{n^2}{8}$$

$$\sum_{i=\left\lceil \frac{n}{2} \right\rceil}^{n-1} i = \sum_{i=\frac{n}{2}}^{n-1} i = \sum_{i=1}^{n-1} i - \sum_{i=1}^{\frac{n}{2}-1} i = \frac{n(n-1)}{2} - \frac{\frac{n}{2}\left(\frac{n}{2}-1\right)}{2} =$$

$$= \frac{n(n-1)}{2} - \frac{n}{4}\left(\frac{n-2}{2}\right) = \frac{n^2-n}{2} - \frac{n^2-2n}{8} = \frac{4n^2-4n-n^2+2n}{8} =$$

$$= \frac{3n^2-2n}{8} \le \frac{3n^2}{8}$$

n dispari

$$\sum_{i=1}^{\left\lceil \frac{n}{2} \right\rceil - 1} i = \sum_{i=1}^{\frac{n+1}{2}-1} i = \sum_{i=1}^{\frac{n-1}{2}} i = \frac{\left(\frac{n-1}{2}\right)\left(\frac{n-1}{2}+1\right)}{2} = \frac{\left(\frac{n-1}{2}\right)\left(\frac{n+1}{2}\right)}{2} = \frac{(n-1)(n+1)}{8} = \frac{n^2-1}{8} \le \frac{n^2}{8}$$

$$\sum_{i=\left\lceil \frac{n}{2} \right\rceil}^{n-1} i = \sum_{i=\frac{n+1}{2}}^{n-1} i = \sum_{i=1}^{n-1} i - \sum_{i=1}^{\frac{n+1}{2}-1} i = \sum_{i=1}^{n-1} i - \sum_{i=1}^{\frac{n-1}{2}} i = \frac{n(n-1)}{2} - \frac{\left(\frac{n-1}{2}+1\right)\left(\frac{n-1}{2}\right)}{2} =$$

$$= \frac{n(n-1)}{2} - \frac{\left(\frac{n+1}{2}\right)\left(\frac{n-1}{2}\right)}{2} = \frac{n(n-1)}{2} - \frac{(n+1)(n-1)}{8} =$$

$$= \frac{4n(n-1)-(n+1)(n-1)}{8} = \frac{4n^2-4n-n^2+1)}{8} = \frac{3n^2-4n+1}{8} \le \frac{3n^2}{8}$$

Dall'analisi fatta sopra sulle sommatorie ne deriva che:

$$P(n) \le 2 + \frac{8}{n^2}\left(\sum_{i=1}^{\left\lceil \frac{n}{2} \right\rceil - 1} i \log \frac{n}{2} + \sum_{i=\left\lceil \frac{n}{2} \right\rceil}^{n-1} i \log n \right) \le 2 + \frac{8}{n^2}\left(\frac{n^2}{8} \log \frac{n}{2} + \frac{3n^2}{8} \log n \right) =$$

$$= 2 + \log \frac{n}{2} + 3 \log n = 2 + \log n - \log 2 + 3 \log n = 2 + \log n - 1 + 3 \log n = 1 + 4 \log n$$

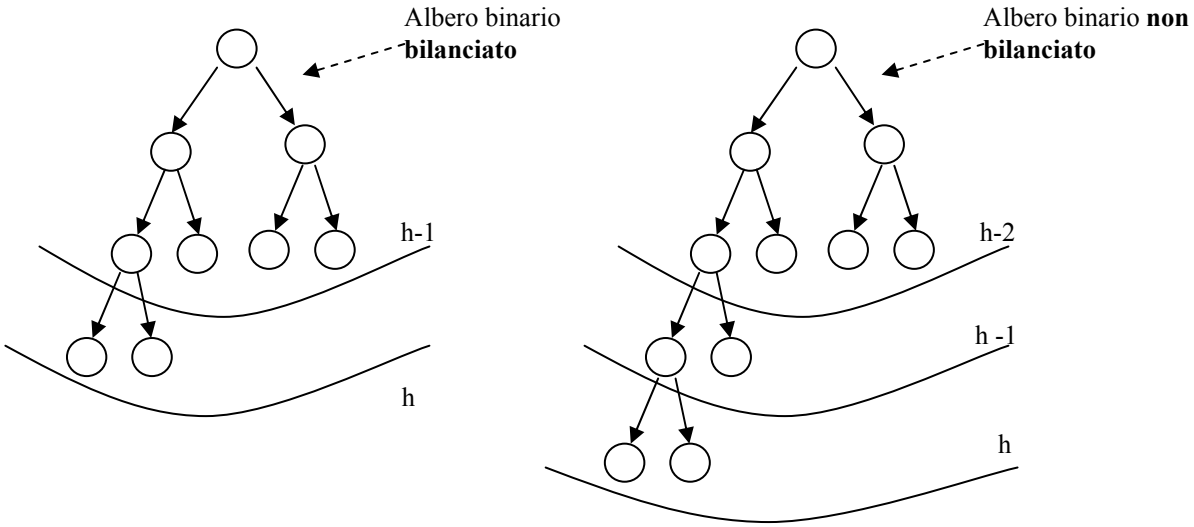
3. Alberi AVL

Nel precedente capitolo abbiamo visto come il bilanciamento di un albero influenzi le operazioni di ricerca, cancellazione e inserimento. Un albero perfettamente bilanciato garantisce per ognuna di queste operazioni una complessità logaritmica nel numero di elementi del dizionario.

Sulla base di questa considerazione, Adel'son, Veliskii e Landis (A.V.L.) hanno ideato una famiglia di alberi, gli Alberi AVL. Caratteristica di tali alberi è quella di mantenere un perfetto bilanciamento anche dopo aver dato luogo ad operazioni di inserimento e cancellazione, ovvero dopo aver apportato modifiche alla struttura dell'albero. Vedremo come l'esecuzione di opportune operazioni, che chiameremo rotazioni di ribilanciamento, rappresenterà la chiave di volta per il mantenimento del bilanciamento dell'albero.

Definizione: un albero binario è ***bilanciato in altezza*** se le altezze del sottoalbero sinistro e destro differiscono al più di una unità.

Esempio – Alberi bilanciati



Definizione: sia v un nodo, definiamo ***fattore di bilanciamento*****,** $\beta(v)$, del nodo v la differenza tra l'altezza dei sottoalberi sinistro e destro radicati in v:

$$\beta(v) = h(sin(v)) - h(des(v)).$$

Notiamo che in un albero binario di ricerca, più il fattore di bilanciamento dei nodi è basso è maggiormente efficienti sono le operazioni di ricerca, inserimento e cancellazione.

Definizione. Un ***Albero AVL*** è un albero binario di ricerca in cui:

1. ogni nodo dell'albero contiene oltre alla coppia (chiave, elemento) informazioni sul bilanciamento del nodo;
2. per ogni nodo v si ha che: $|\beta(v)| \leq 1$;

Un Albero AVL è pertanto un albero binario che mantiene le proprietà di ricerca ed inoltre è bilanciato. Si dimostra il seguente:

Teorema fondamentale degli alberi AVL: *sia T un Albero AVL. Indichiamo con n il numero di nodi dell'albero T e con h l'altezza dell'albero T.*
Si ha che:

$$h = O(\log n).$$

Dim: omessa.

3.1 Rotazioni per il ribilanciamento dell'albero

Le rotazioni di ribilanciamento rappresentano un insieme di tecniche applicabili agli Alberi AVL ogni qual volta, dopo aver dato luogo ad operazioni che apportano modifiche alla struttura dell'albero, il fattore di bilanciamento di un nodo risulti non essere più minore o uguale ad uno.

Le rotazioni di ribilanciamento mirano a ripristinare il fattore di bilanciamento dei nodi dell'Albero AVL. Vediamo come.

Sia v un nodo sbilanciato, ovvero un nodo tale che $\beta(v)$ è maggiore o uguale a due. Sia u il nodo figlio di v in cui è radicato l'albero avente altezza maggiore. Siano infine $T(u)_{sin}$ e $T(u)_{des}$ rispettivamente il sottoalbero sinistro e destro di u. Si possono verificare i seguenti quattro casi:

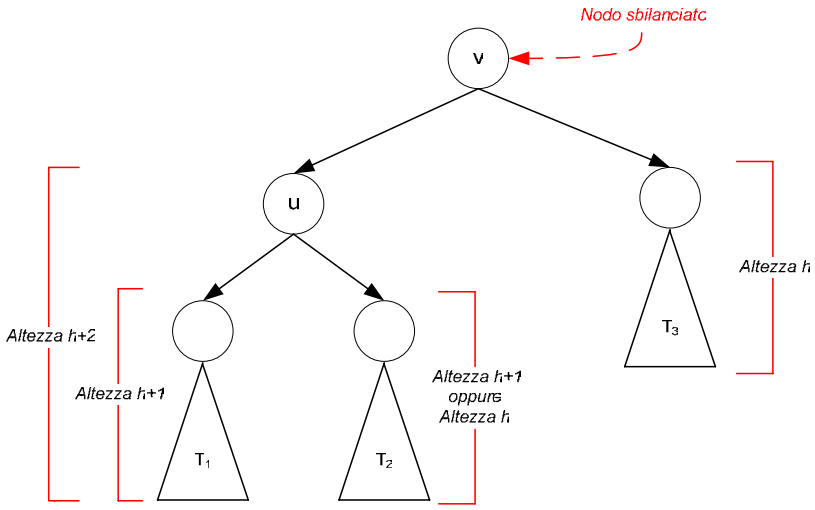
1. u è figlio sinistro di v e $T(u)_{sin}$ ha altezza maggiore o uguale a $T(u)_{des}$.
2. u è figlio sinistro di v e $T(u)_{des}$ ha altezza maggiore di $T(u)_{sin}$.
3. u è figlio destro di v e $T(u)_{des}$ ha altezza maggiore o uguale a $T(u)_{sin}$.
4. u è figlio destro di v e $T(u)_{sin}$ ha altezza maggiore $T(u)_{des}$.

Per facilità di esposizione, considereremo nella nostra analisi solo i primi due casi, poiché i rimanenti casi sono simmetrici rispetto ad essi.

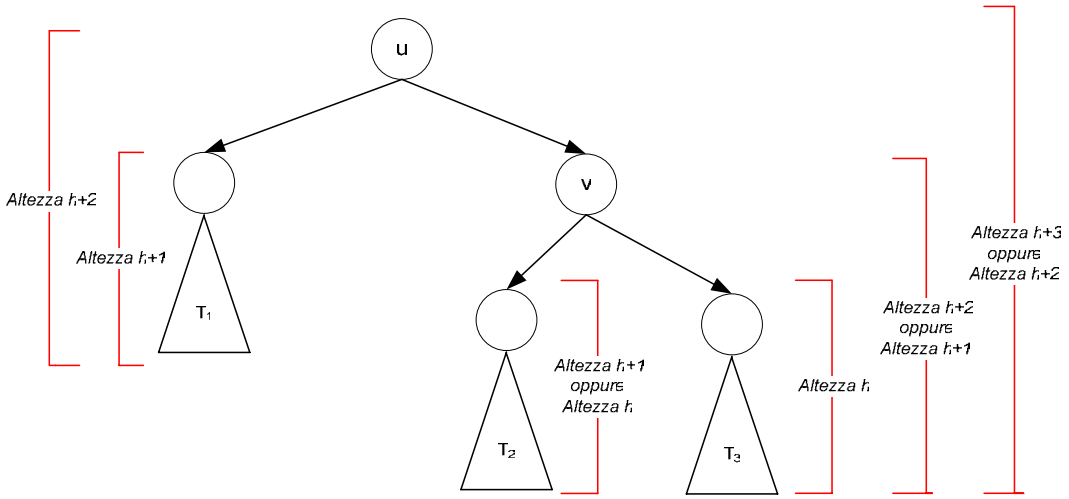
Caso 1

Osserviamo la figura in basso. Indichiamo con v il nodo avente fattore di bilanciamento maggiore di uno, con u il figlio sinistro di v ed inoltre con T_3 l'albero destro radicato in v. Siano infine T_1 e T_2 rispettivamente il sottoalbero sinistro e destro di u.

Assumiamo quindi che l'altezza dell'albero radicato in u è maggiore di due unità rispetto all'altezza del sottoalbero T_3. Inoltre che l'altezza di T_1 è maggiore o uguale all'altezza di T_2. Tale differenza non può superare il valore uno, altrimenti il nodo u sarebbe un nodo sbilanciato.

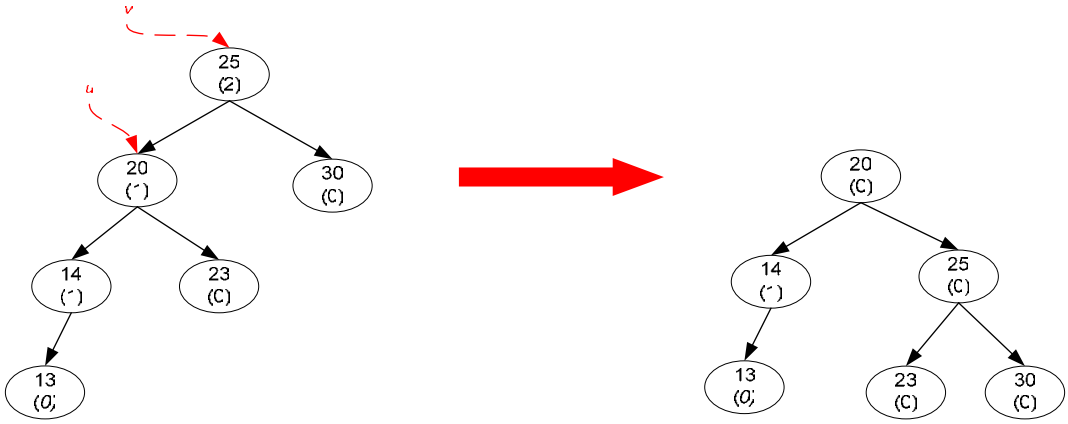


In tal caso possiamo ribilanciare l'albero sul nodo *v* dando luogo ad una rotazione destra su *v*, ovvero poniamo *u* padre di *v* e spostiamo il sottoalbero T_2 come sottoalbero sinistro radicato in *v*.



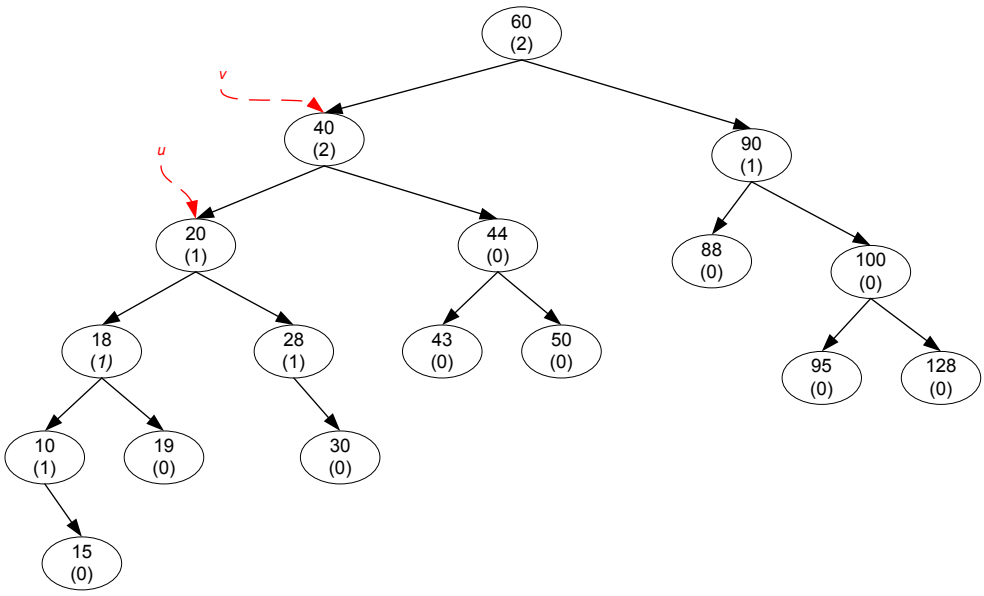
Esempio 3.1.1

Consideriamo il seguente Albero AVL dove tra parentesi riportiamo il fattore di bilanciamento di ogni nodo.
La radice dell'albero rappresenta un nodo sbilanciato. Operiamo quindi la rotazione come descritto nel caso 1.

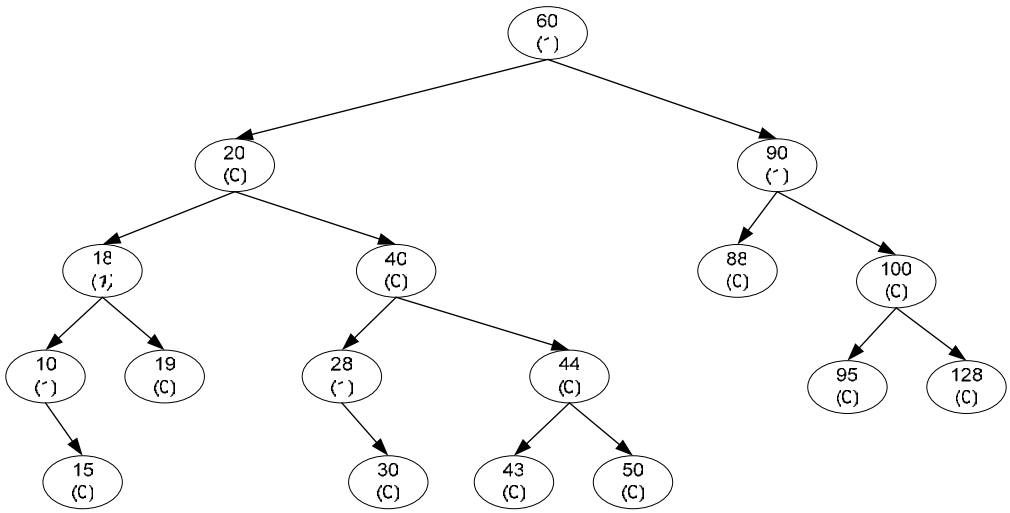


Esempio 3.1.2

Consideriamo il seguente Albero AVL.



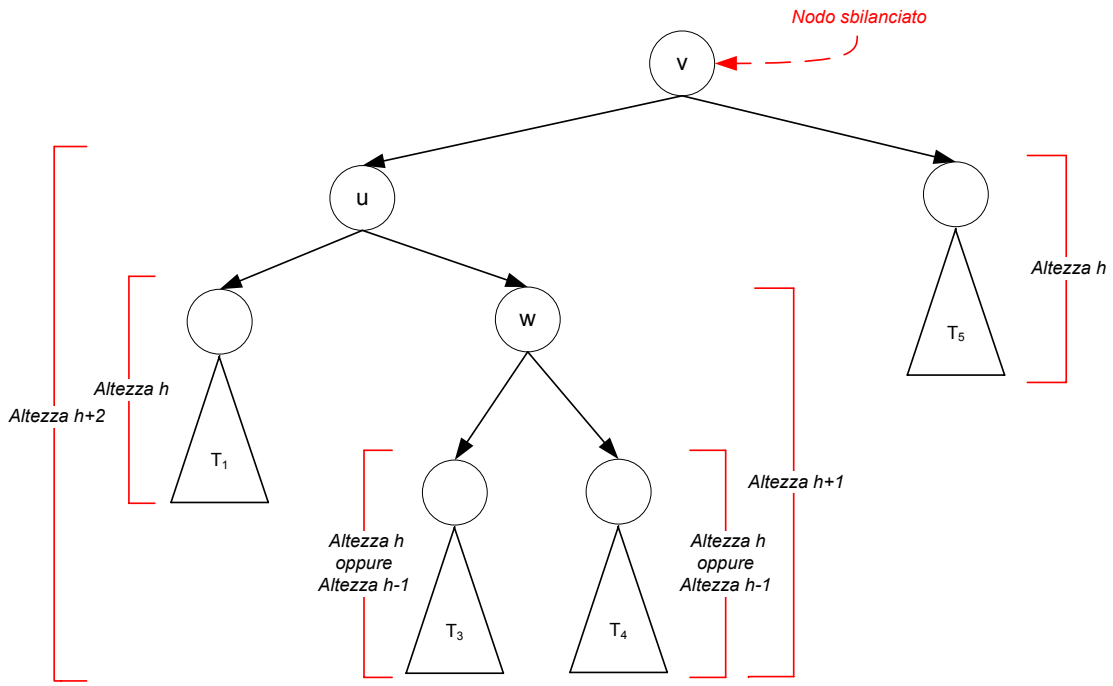
Anche in questo caso ricorriamo alla rotazione verso destra per ristabilire gli equilibri di bilanciamento.
Notiamo come l'operazione di bilanciamento applicata al nodo v ristabilisce anche il fattore di bilanciamento della radice dell'albero.



Caso 2

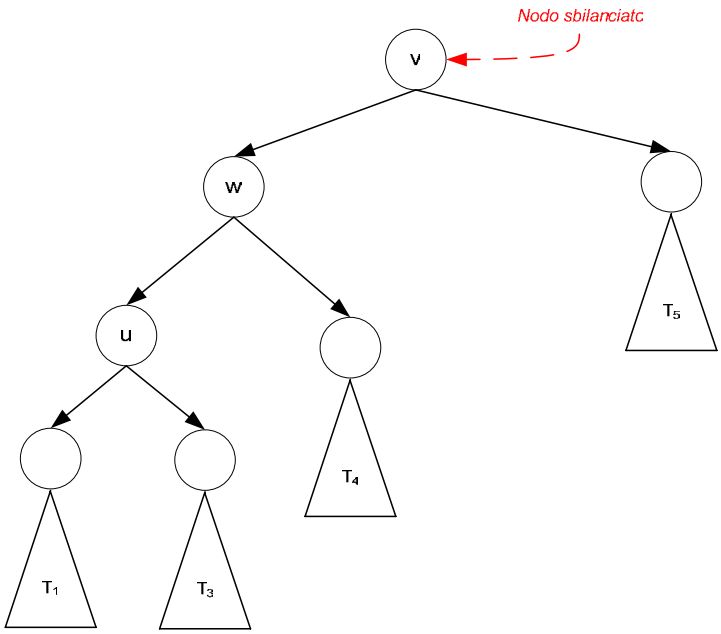
Similmente a quanto fatto sopra, osserviamo la figura in basso dove indichiamo con *v* il nodo avente fattore di bilanciamento maggiore di uno, con *u* il figlio sinistro di *v* ed inoltre con T_5 l'albero destro radicato in *v*. Siano infine T_1 il sottoalbero sinistro radicato in *u*, *w* il figlio destro di *u* e T_3, T_4 rispettivamente i sottoalberi sinistro e destro radicati in *w*.

Assumiamo quindi che l'altezza dell'albero radicato in *u* sia maggiore di due unità rispetto all'altezza del sottoalbero T_5. Inoltre che l'altezza di T_1 è minore dell'altezza del sottoalbero radicato in *w*. Notiamo, similmente al caso sopra affrontato che tale differenza non può superare il valore uno, altrimenti il nodo *u* sarebbe un nodo sbilanciato.

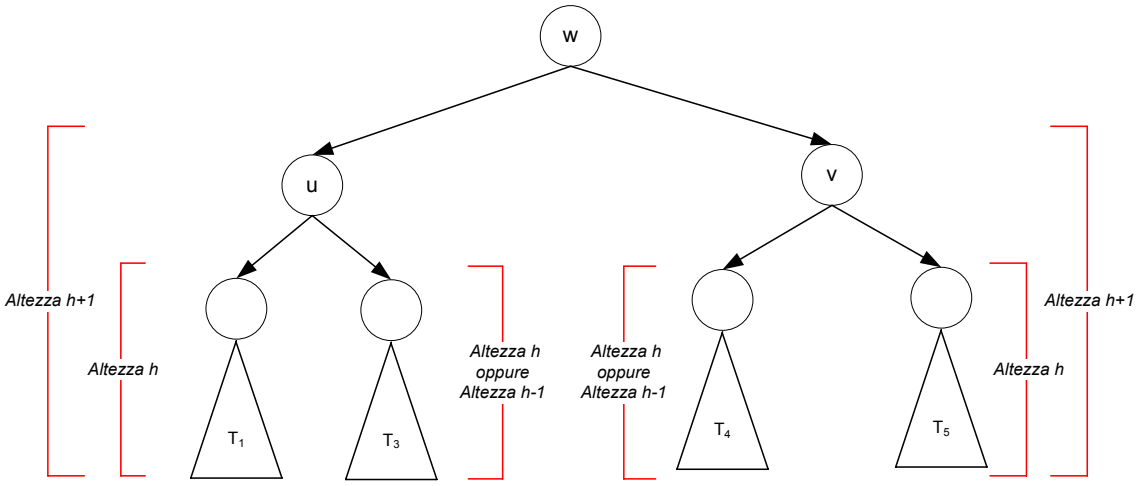


In tal caso ribilanciamo l'albero sul nodo *v* eseguendo consecutivamente le seguenti due rotazioni:

Rotazione 1: rotazione verso sinistra su *u*, poniamo *w* padre di *u* ed il sottoalbero T_3 come sottoalbero destro radicato in *u*

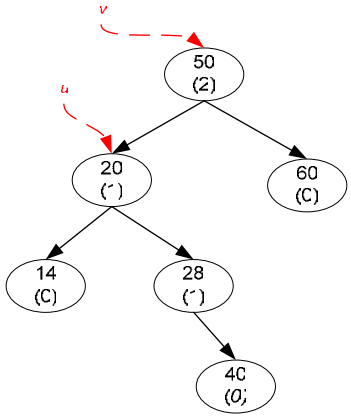


Rotazione 2: rotazione verso destra su v, poniamo *w* padre di *v* ed il sottoalbero *T4* come sottoalbero sinistro radicato in *v*.



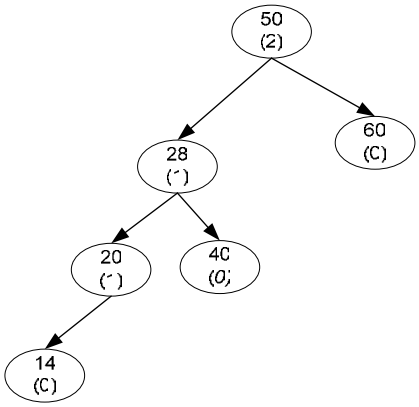
Esempio 3.1.3

Consideriamo il seguente Albero AVL.

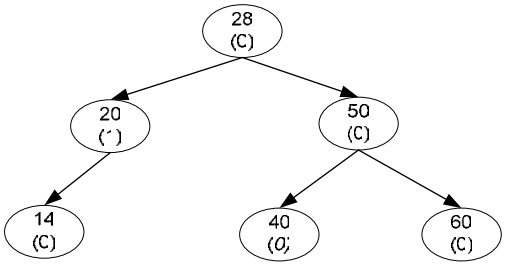


Ricadiamo nel caso 2 e pertanto per ristabilire gli equilibri di bilanciamento eseguiamo la doppia rotazione:

Prima rotazione



Seconda rotazione



3.2 Gli operatori del dizionario

Prima di focalizzare la nostra attenzione sugli operatori del dizionario, osserviamo dapprima che le singole tecniche di ribilanciamento descritte nel Paragrafo 3.1 hanno una complessità indipendente dal numero di elementi presenti nel dizionario. La loro complessità è pertanto costante: O(1).

Ricerca

Poichè in un Albero AVL sono comunque rispettate le proprietà di ricerca introdotte per i BST, l'algoritmo di ricerca di un elemento all'interno di un Albero AVL è identico all'algoritmo *CercaChiave* visto per i BST. La complessità dell'algoritmo rimane pertanto pari a *O(h)* e sulla base del teorema fondamentale degli alberi AVL possiamo affermare che tale complessità in termini di nodi è pari a *O(log n)*.

Inserimento

L'inserimento in un albero AVL prevede l'esecuzione di tre macro passi distinti:

1. inserimento del nodo come foglia, in modalità identica a quella già descritta nell'albero binario di ricerca;
2. aggiornamento dei fattori di bilanciamento di tutti i nodi presenti nel cammino che collega il nodo appena inserito con la radice dell'albero, *cammino critico*;
3. ribilanciamento dell'albero, mediante le tecniche di rotazione sopra descritte. Tecniche da applicare solo qualora almeno un nodo dell'albero presentasse un fattore di bilanciamento maggiore di uno.

Analizziamo l'ultimo passo. Sia *n* il primo nodo di livello più basso che incontriamo lungo il cammino critico e avente un fattore di bilanciamento maggiore di uno. Chiamiamo tale nodo "*nodo critico"*. Come abbiamo già visto nell'esempio 3.1.2 si ha che:

Lemma: *una rotazione semplice o doppia sul nodo critico è sufficiente a ribilanciare in altezza un albero AVL dopo l'inserimento di un nuovo elemento*.

Pertanto dopo aver applicata l'opportuna tecnica di bilanciamento al nodo critico non occorre più risalire l'albero ed effettuare ulteriori bilanciamenti. I fattori di bilanciamento dei nodi più in alto vengono di conseguenza ristabiliti.

Il costo dell'operazione di inserimento è pertanto proporzionale all'altezza dell'albero, *O(h)*, e sulla base del teorema fondamentale degli alberi AVL possiamo affermare che tale complessità in termini di nodi è pari a *O(log n)*.

Cancellazione

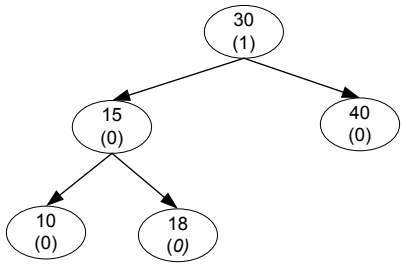
Similmente all'inserimento la cancellazione di un nodo avviene in tre passi distinti:

1. cancellazione del nodo come descritto per gli alberi binari di ricerca
2. aggiornamento dei fattori di bilanciamento dei nodi presenti lungo il cammino critico, rappresentato questa volta dal cammino che collega il padre del nodo cancellato e la radice dell'albero;
3. ribilanciamento dell'albero, mediante le tecniche di rotazione sopra descritte. Tecniche da applicare solo qualora almeno un nodo dell'albero presentasse un fattore di bilanciamento maggiore di uno.

Diversamente dall'inserimento, nella cancellazione di un elemento l'algoritmo di ribilanciamento potrebbe essere applicato più di una volta. Possiamo incorrere in uno sbilanciamento a cascata che nel caso pessimo obbligherebbe l'esecuzione di *O(h)* ribilanciamenti. Ad ogni modo tale inconveniente rende particolarmente laborioso la procedura di cancellazione, ma in termini di complessità possiamo comunque affermare che la cancellazione di un nodo ha complessità pari a *O(h)* e quindi dal teorema fondamentale pari a *O(log n)*.

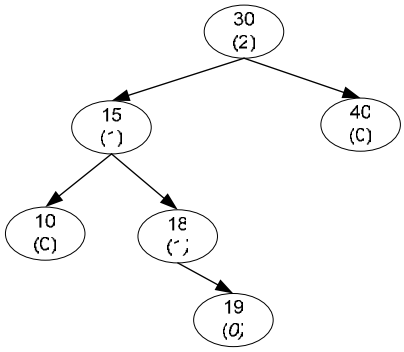
Esempio 3.2.1

Consideriamo il seguente albero AVL.

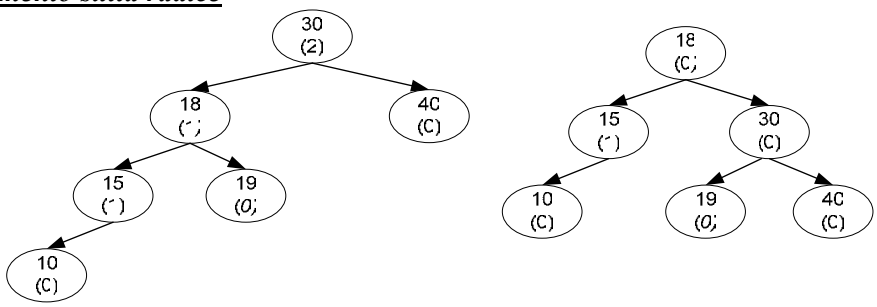


Rappresentiamo tutte le modifiche apportate all'albero in seguito all'applicazione della seguente sequenza di operazioni: insert(19), insert(42), insert(53), insert(41).

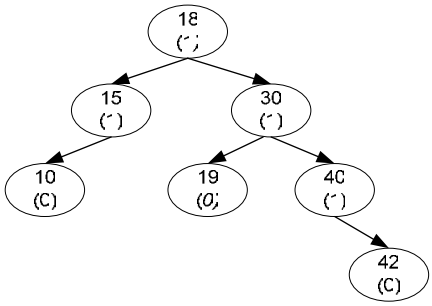
Insert 19



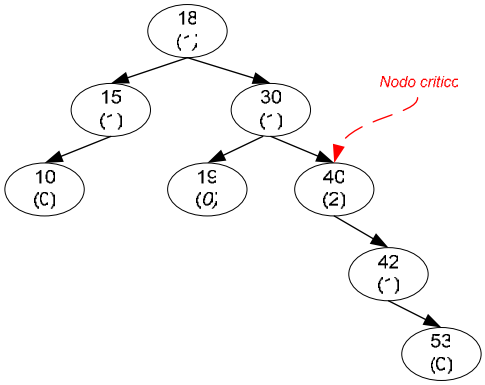
Ribilanciamento sulla radice



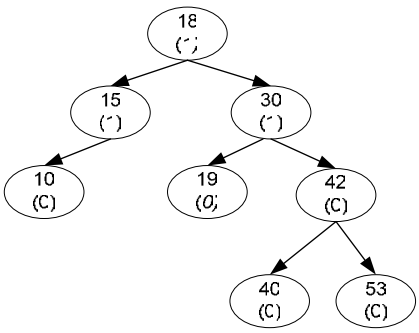
Insert 42



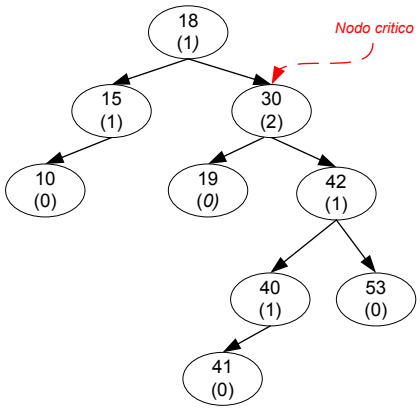
Insert 53



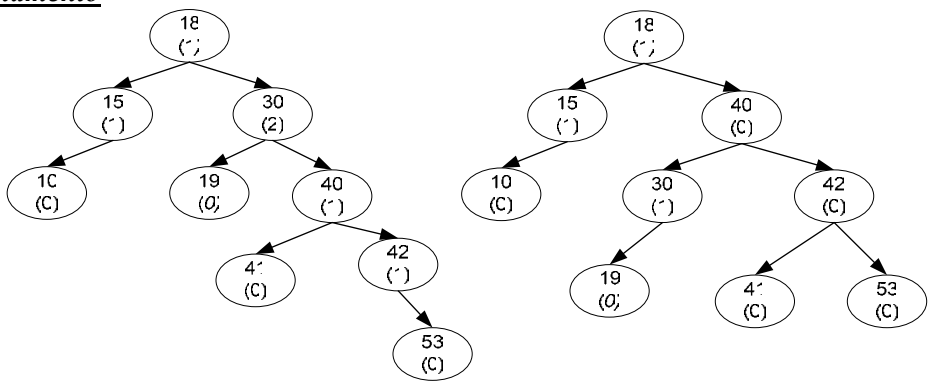
Ribilanciamento



Insert 41



Ribilanciamento



4. Realizzazione mediante Alberi 2-3

Studieremo in questo capitolo una organizzazione dati che fornisce maggiore libertà sul grado uscente dei nodi: l'Albero 2-3.

Dimostreremo che, similmente agli Alberi AVL, un Albero 2-3 possiede tra le sue peculiarità anche quella di avere una altezza limitata superiormente, a meno di una costante, dal logaritmo del numero dei nodi: *O(log n)*.

Vedremo come tale caratteristica rappresenti una garanzia per una efficiente implementazione degli operatori di inserimento, ricerca e cancellazione, che continueranno ad avere anche in questo caso complessità pari a *O(log n)*.

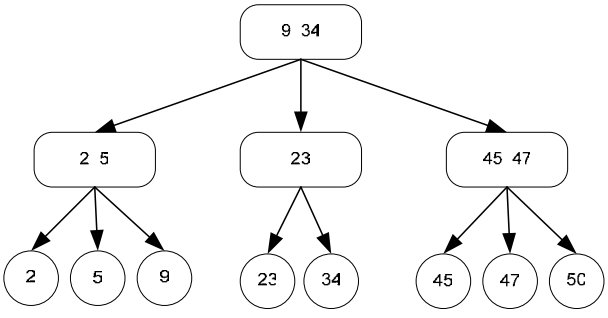
Vedremo inoltre come tale garanzia potrebbe venir meno a seguito di una operazione di inserimento o cancellazione e, in tal caso, a quali operazioni dovremo necessariamente dar luogo al fine di ristabilire un equilibrio sulla distribuzione in altezza dei nodi. Chiameremo tali operazioni *split* e *fuse*.

Definizione. Un Albero 2-3 è un albero in cui:

1. ogni nodo interno ha due o tre figli;
2. tutte le foglie appartengono allo stesso livello;
3. le chiavi e gli elementi del dizionario sono contenute esclusivamente all'interno dei nodi foglia:
4. le chiavi, presenti nei nodi foglia, appaiono in ordine crescente da sinistra verso destra;
5. ogni nodo interno *v* mantiene la chiave massima del sottoalbero sinistro e la chiave massima del sottoalbero centrale se ha tre figli. Altrimenti mantiene solamente la chiave massima del sottoalbero sinistro.

Esempio

In basso una raffigurazione dei un Albero 2-3.



Il seguente teorema introduce una importante proprietà di un albero così strutturato: l'altezza, a meno di una costante, è limitata superiormente dal numero di nodi presenti nell'albero.

Teorema Fondamentale Alberi 2-3. *Sia T un Albero 2-3. Indichiamo con n il numero dei nodi, con f il numero delle foglie ed infine con h l'altezza. Si ha che:*

$$a) \quad 2^{h+1} - 1 \leq n \leq \frac{3^{h+1} - 1}{2}$$

$$b) \quad 2^h \leq f \leq 3^h$$

Dim. Procediamo per induzione sull'altezza dell'Albero 2-3. Per h uguale a zero abbiamo che esso è composto dalla sola radice. Tale nodo è anche foglia e quindi le due asserzioni a) e b) sono banalmente verificate.
Poniamo vere le due asserzioni anche per un'altezza *h-1*. Indichiamo con T_1 l'Albero 2-3 di altezza *h-1*, con n_1 i nodi dell'albero T_1 e con f_1 il numero delle foglie di T_1. Avremo che:

$$i) \quad 2^h - 1 \leq n_1 \leq \frac{3^h - 1}{2}$$

$$ii) \quad 2^{h-1} \leq f_1 \leq 3^{h-1}$$

Costruiamo ora un Albero 2-3 di altezza h, che chiameremo T, sulla base dell'Albero 2-3 T_1.
Aggiungiamo quindi un ulteriore livello a T_1 inserendo ad ogni foglia di T_1 minimo due o massimo tre figli.
Otteniamo già un primo risultato. Dalla *ii*) abbiamo che il numero di foglie di T, che indichiamo con f, sarà nel caso minimo maggiore o uguale $2*2^{h-1} = 2^h$ o nel caso massimo minore o uguale a $3*3^{h-1} = 3^h$, cioè:

$$iii) \quad 2^h \leq f \leq 3^h.$$

Indichiamo ora con n il numero di nodi di T.
Abbiamo che $n = n_1 + f$. Cioè li numero di nodi di T è pari al numero di nodi dell'albero T_1 più il numero di foglie che ad esso abbiamo aggiunto per aumentarne l'altezza. Pertanto dalla *i)* e dalla *iii)* abbiamo prima che:

$$n_1 + f \geq 2^h - 1 + 2^h = 2(2^h) - 1 = 2^{h+1} - 1$$

e quindi che:

$$n_1 + f \leq \frac{3^h - 1}{2} + 3^h = \frac{3^h - 1 + (2 * 3^h)}{2} = \frac{3\,3^h - 1}{2} = \frac{3^{h+1} - 1}{2}$$

Come già ampiamente detto in precedenza, una struttura dati ha influenza sulla complessità degli operatori. Vediamo quindi come l'organizzazione dati suggerita dall'Albero 2-3 influenza gli operatori del dizionario: ricerca, inserimento e cancellazione.

Ricerca

La ricerca di un elemento avviene sulla base delle informazioni contenute all'interno dei nodi interni. Indichiamo con *v* un nodo interno dell'Albero 2-3, indichiamo con *S[v]* il massimo del sottoalbero sinistro radicato in *v*, mentre indichiamo con *M[v]* il massimo del sottoalbero centrale radicato in *v*. Ambedue i valori sono memorizzati all'interno del nodo *v*. Possiamo descrivere, in maniera del tutto generica, l'algoritmo di ricerca di una chiave mediante il seguente elenco di passi:

1. confrontiamo la chiave da ricercare, *k*, con *S[v]* e *M[v]*;
2. se *k* è minore o uguale a *S[v]* proseguiamo nel sottoalbero sinistro;
3. se *k* è compreso tra *S[v]* e *M[v]* proseguiamo nel sottoalbero centrale;
4. se i casi 2 e 3 non sono verificati proseguiamo nel sottoalbero di destra.

Alberi 2-3, ricerca di una chiave

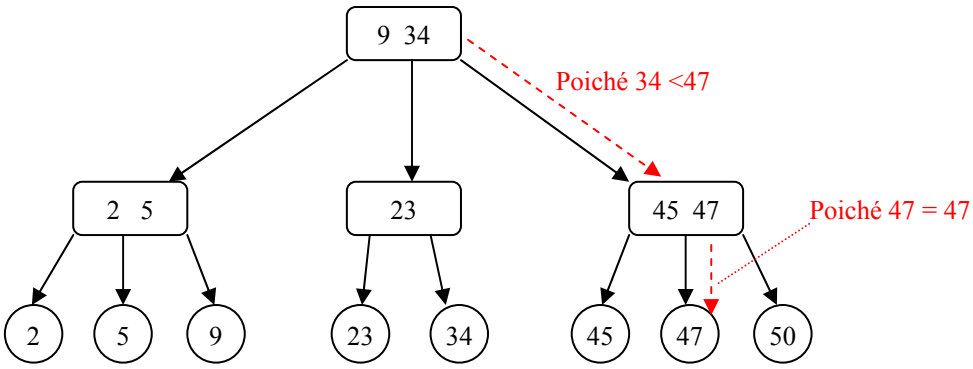
```
Search( Nodo v, chiave k) →Nodo
{
    se v è foglia
        allora
        {
            se k = v.chiave allora ritorna v
            altrimenti ritorna null
        }

    se k <= S[v]
        allora Search(  v.sinistro, k )
        altrimenti
        {
            se v ha tre figli e  S[v] <  k <= M[v]
            allora Search(  v.centro, k )
            altrimenti Search( v.destro, k )
        }
}
```

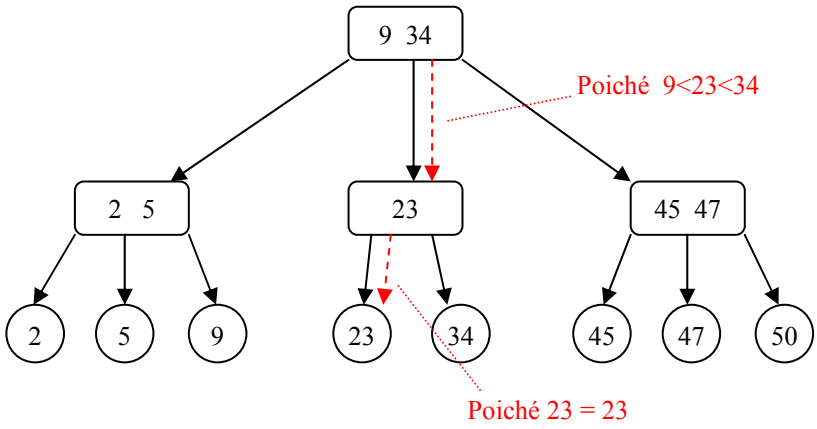
Nel caso pessimo, elemento non trovato, l'algoritmo di ricerca scenderà lungo un intero verso dell'Albero 2-3. Possiamo quindi affermare che la complessità dell'operatore è pari a *O(h)*. Da qui, e sulla base del teorema fondamentale per gli Alberi 2-3, possiamo concludere che in termini di nodi dell'albero, ovvero di numero di elementi del dizionario, la complessità dell'operatore di ricerca è *O(log n)*.

Esempio

Ricerca della chiave 47

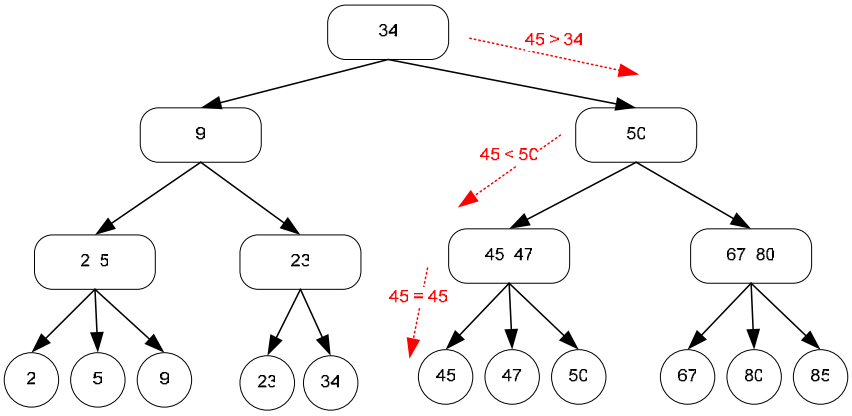


Ricerca della chiave 23.

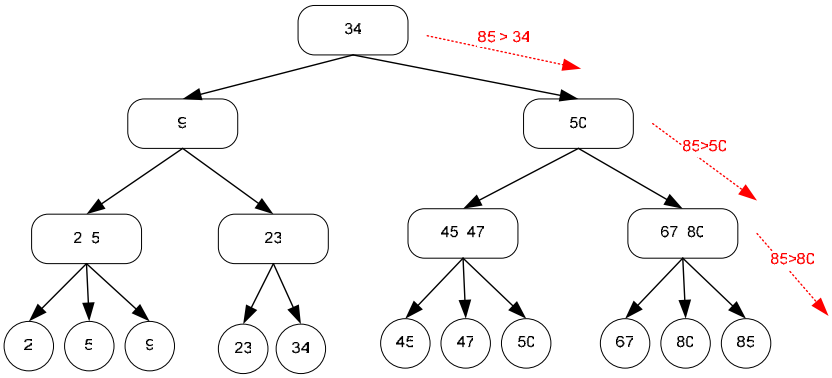


Esempio

Ricerca dell'elemento 45.



Ricerca dell'elemento 85



Inserimento

Poiché l'operatore di inserimento apporta una modifica alla struttura dell'albero, dobbiamo essere certi che tale operazione venga eseguita nel rispetto delle proprietà di un Albero 2-3. Come vedremo, ci garantiremo la piena compatibilità con tali proprietà eseguendo, qualora necessario, una serie di operazioni volte a riequilibrare la distribuzione in altezza dei nodi.

Per inserire in un Albero 2-3 un nuovo nodo, *u*, avente chiave *k*, ricerchiamo prima il nodo interno che potrebbe rappresentare il giusto genitore di *u*. Diamo luogo a tale operazione applicando i medesimi passi visti nell'algoritmo di ricerca di una chiave. Applicheremo la ricerca sulla chiave *k* e ci fermeremo solo quando giungeremo ad un nodo interno avente altezza *h-1*.

Sia quindi *v* il nodo che potrebbe rappresenta il giusto genitore del nodo da inserire, *u*. Si possono verificare i seguenti due casi:

1. ***v ha due figli,*** in tal caso il nuovo nodo verrà inserito come figlio di *v*, facendo attenzione a mantenere inalterato l'ordine crescente delle foglie.
2. ***v ha tre figli:*** in tal caso non è possibile aggiungere altri figli a *v*. Ricorriamo pertanto alla tecnica della suddivisione (*split*): creiamo un nuovo nodo *w* avente come figli le due foglie minime di *v*, poniamo *w* come figlio del padre di *v* e aggiorniamo le informazioni contenute in *v*. Infine, se il padre di *v* aveva in precedenza due figli ci fermiamo, altrimenti eseguiamo nuovamente lo *split* sul padre di *v*. Applicheremo quest'ultimo passo anche al padre di *v* e così man mano che risaliamo l'albero.

Alberi 2-3, operazione di split

```
Split ( Nodo v)
{
      Nodo w
      Sia  v_i l'i-esimo figlio di v,  1≤ i ≤ 4

      poni  v_1  v_2  figli di w
      se ( padre v è NULL )
      allora
      {
            crea un nuovo nodo n
            poni sia v che w figli di n
      }
      altrimenti
      {
            poni anche w come figlio del padre di v
            se ( padre di v ha quattro figli ) allora split( padre di v)
}
```

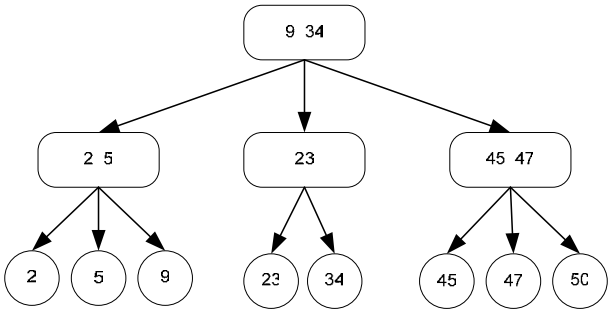
Analizziamo la complessità dell'operazione di inserimento. Essa si sviluppa essenzialmente nell'applicazione di due macro passi: ricerca del giusto genitore e collocazione del nuovo nodo.

La ricerca del giusto genitore, come abbiamo già visto, ha una complessità pari a *O(h)*.

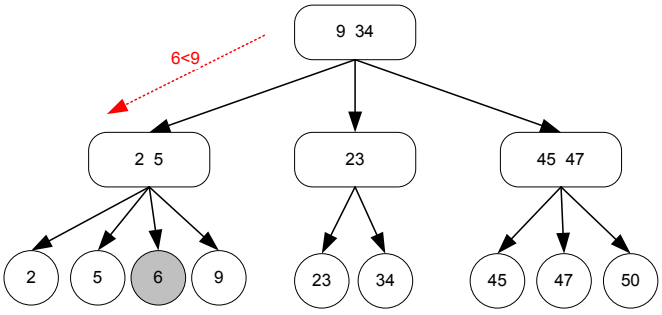
Analizziamo pertanto la collocazione del nuovo nodo. Valutiamo il caso pessimo: dobbiamo dar luogo ad una successione di operazioni di split. Se consideriamo che un'operazione di split ha complessità costante rispetto al numero di nodi presenti nell'albero, *O(1)*, ed il numero di operazioni di split è al massimo pari all'altezza dell'albero[4], *h*, possiamo dire che l'inserimento nel caso pessimo ha anch'esso complessità pari a *O(h)*, cioè in considerazione del teorema principale pari a *O(log n)*.

Esempio

Inseriamo la chiave 6 all'interno del seguente Albero 2-3

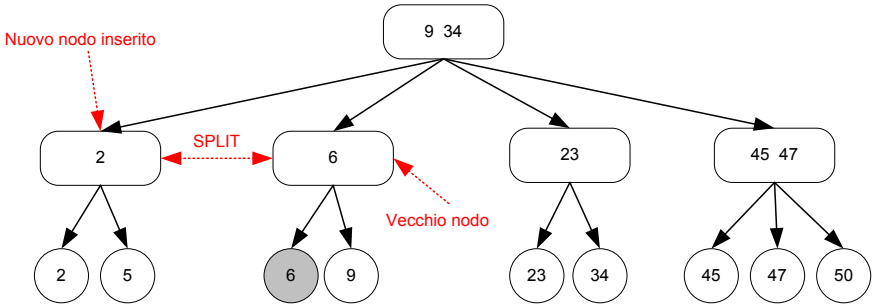


Il giusto genitore è rappresentato dal nodo contenente i valori 2 e 5.

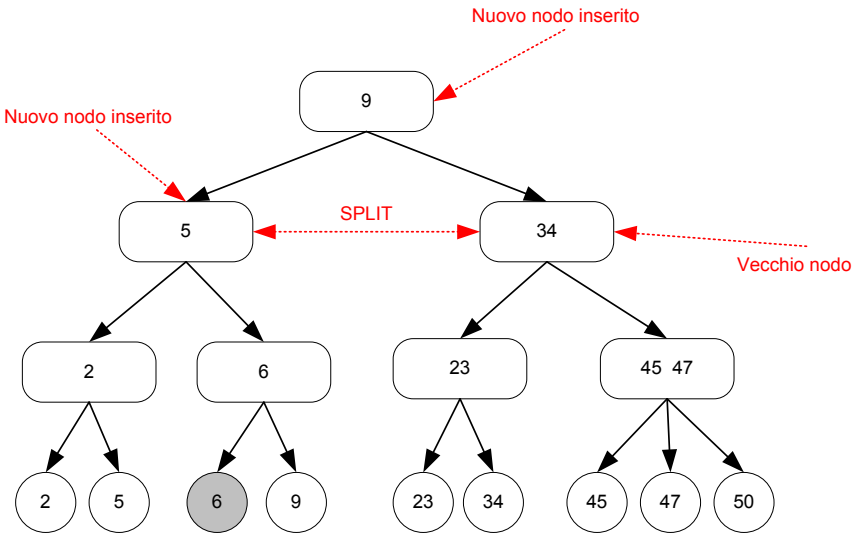


Esso però già possiede tre figli. Dobbiamo quindi ricorrere ad uno split.

[4] Una volta arrivati alla radice ci fermiamo comunque

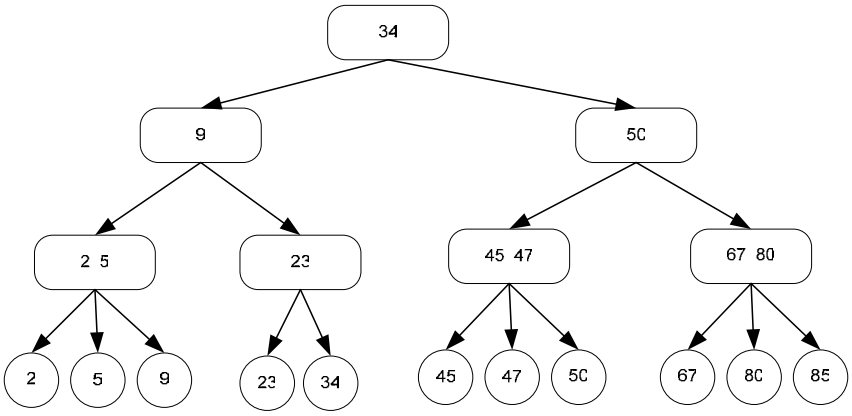


La radice dell'albero possiede, a questo punto, quattro figli. Dobbiamo dar luogo ad una ulteriore operazione di split

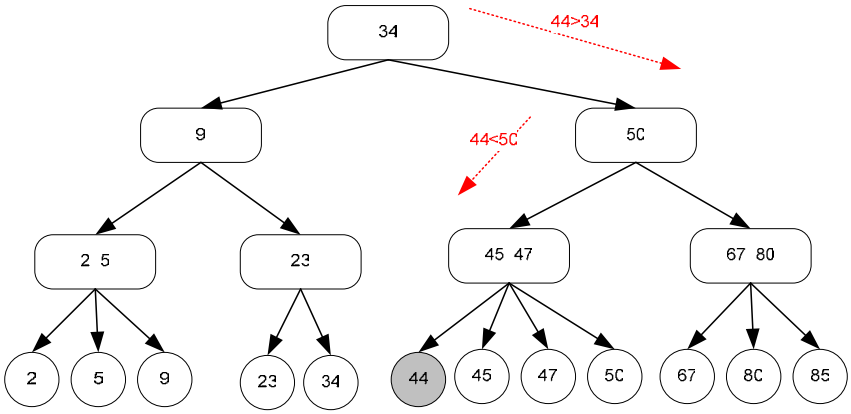


Esempio

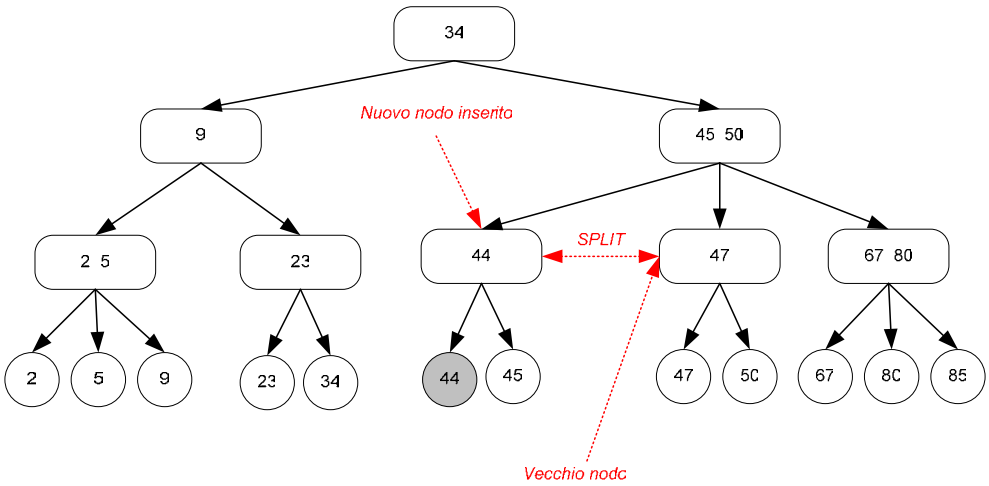
Inseriamo la chiave 44 all'interno del seguente Albero 2-3.



Il giusto genitore è rappresentato dal nodo contenente i valori 45 e 47



Il nodo ha già tre figli, ricorriamo allo split.



Cancellazione

Similmente all'operatore di inserimento, quello di cancellazione apporta una modifica alla struttura dell'albero. Pertanto, al fine di garantire il rispetto delle proprietà di un Albero 2-3 eseguiremo, anche questa volta, una serie di operazioni volte a riequilibrare la distribuzione in altezza dei nodi.

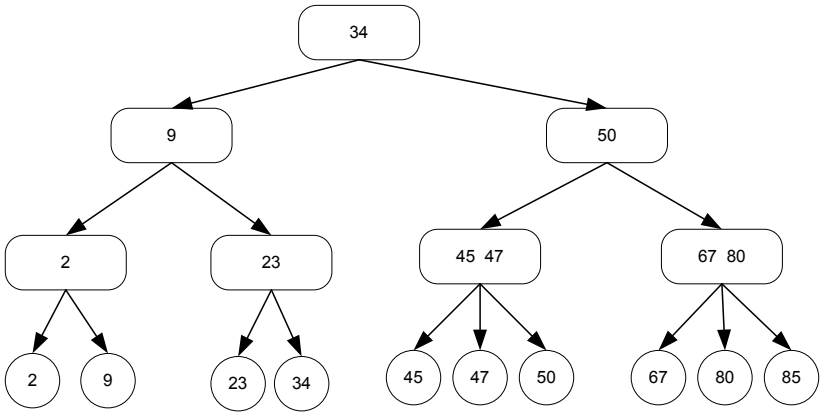
Indichiamo con *v* il nodo da eliminare, si possono verificare i seguenti tre casi:

1. ***v è radice***, rimuoviamo il nodo ottenendo un albero vuoto
2. ***il padre di v ha tre figli***, rimuoviamo semplicemente *v* e aggiorniamo le informazioni contenute nei nodi presenti lungo il percorso che collega il padre di *v* e la radice;
3. ***il padre di v ha due figli.*** Se il padre di *v* è radice del sottoalbero si eliminano sia *v* che il padre di *v*. Definiamo in tal modo un albero la cui radice sarà l'altro figlio del padre di *v*. Altrimenti, sia *w* il padre di *v*, eseguiamo un controllo sui fratelli di *w*. Si possono verificare i seguenti due sottocasi:
    a. ***w ha un fratello sinistro (destro) avente tre figli***. Sia *z* tale nodo. Spostiamo il figlio più a destra (sinistra) di *z* rendendolo figlio sinistro (destro) di *w*.
    b. ***nessun fratello di w ha tre figli.*** In tal caso eseguiamo la procedura di fusione (*fuse*): spostiamo il rimanente figlio di *w* ponendolo figlio di uno dei fratelli di *w* ed infine richiamiamo la cancellazione su *w*.

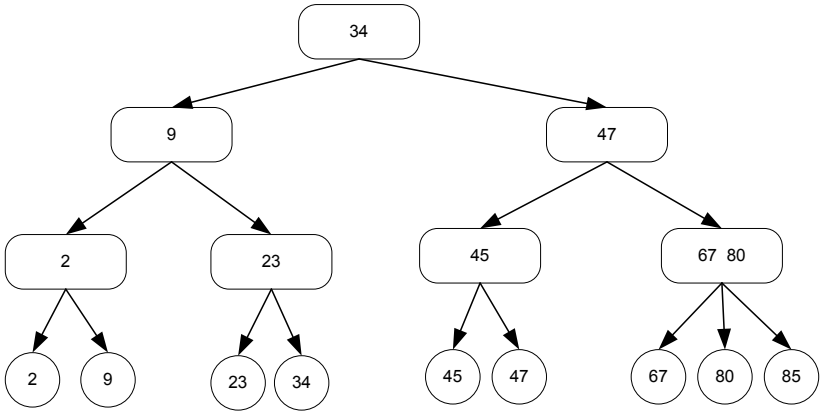
Analizziamo la complessità dell'operatore di cancellazione. Il caso pessimo è rappresentato dal caso *3b*, dobbiamo dar luogo ad un'operazione di fusione che potrà generare a sua volta una successione di operazioni di cancellazione. Considerando che l'operazione di fusione, anche se laboriosa, ha una complessità costante nel numero di nodi dell'albero, *O(1)*, ed inoltre che il numero massimo di operazioni di cancellazione da eseguire in successione è pari all'altezza dell'albero, possiamo affermare che l'operatore di cancellazione ha una complessità pari a *O(logn)*, cioè in considerazione del teorema principale pari a *O(log n)*.

Esempio

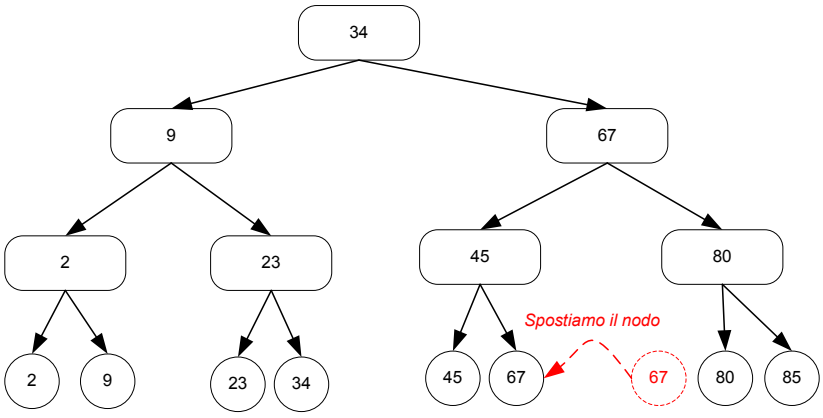
Consideriamo il seguente Albero 2-3. Eliminiamo la chiave 50.



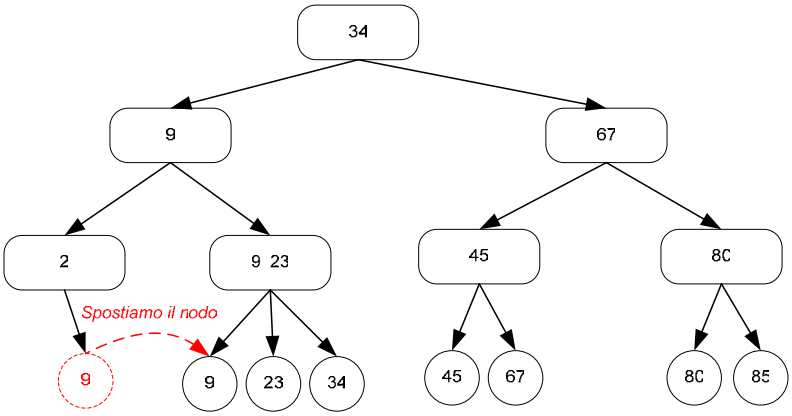
Poiché il padre del nodo contenente la chiave 50 possiede tre figli, eliminiamo semplicemente il nodo ed otteniamo quindi il seguente Albero 2-3.



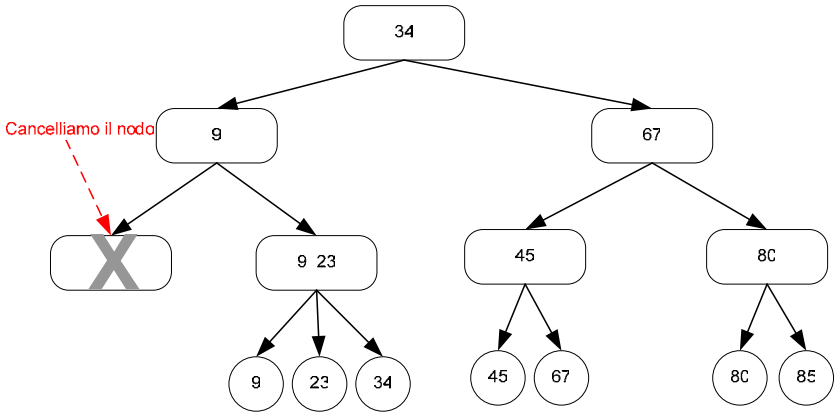
Eliminiamo ora la chiave 47. Poichè il padre del nodo contenente la chiave 47 ha un fratello con tre figli, quello avente i valori 67 e 80, spostiamo il figlio più a sinistra di quest'ultimo, 67, al posto del nodo 47. Otteniamo il seguente albero.



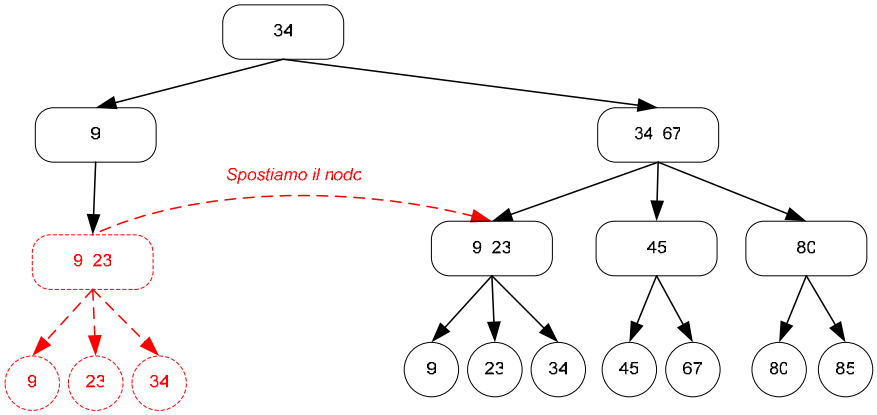
Eliminiamo ora la chiave 2. Poichè il padre del nodo contenente la chiave 2 non possiede fratelli aventi tre nodi dobbiamo ricorrere prima ad una fusione ottenendo quindi il seguente albero



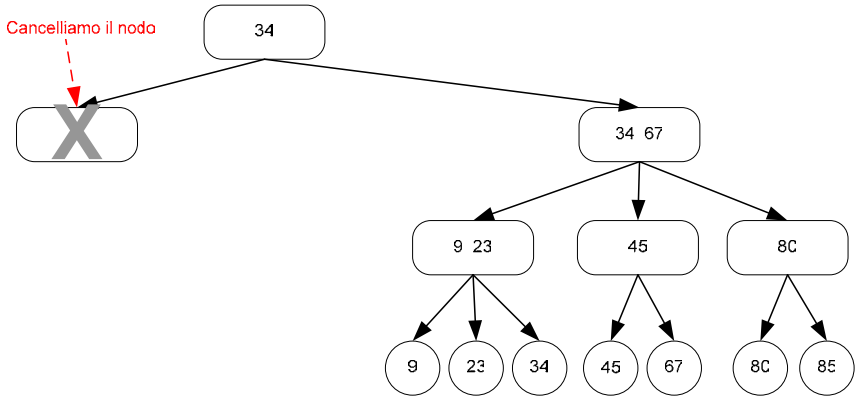
Ed infine dobbiamo richiamare la cancellazione sul nodo rimasto privo di figli



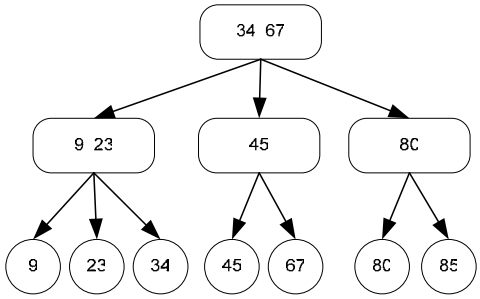
Ora poiché il nodo da cancellare ha un padre che non possiede fratelli con tre figli dobbiamo ricorrere nuovamente prima ad una fusione.



Ed infine richiamare la cancellazione sul nodo rimasto privi di figli.

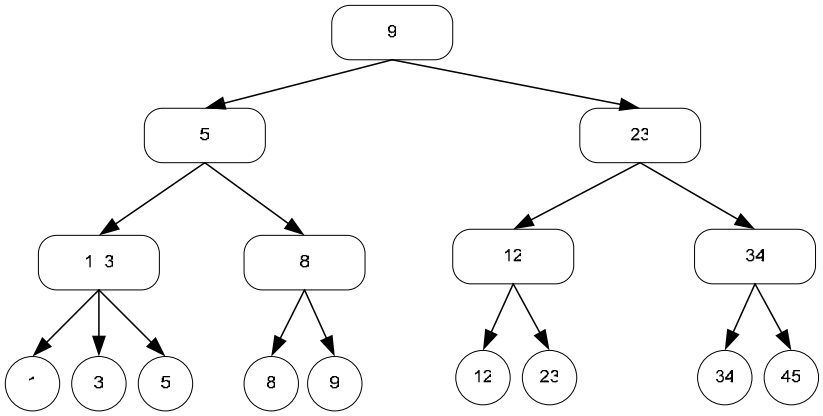


Il padre del nodo da cancellare è una radice per cui eliminiamo sia il nodo che il padre, ottenendo quindi il seguente Albero 2-3

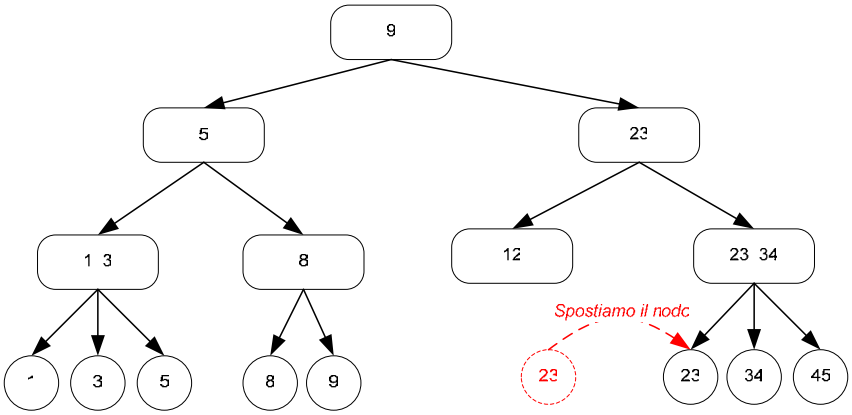


Esempio

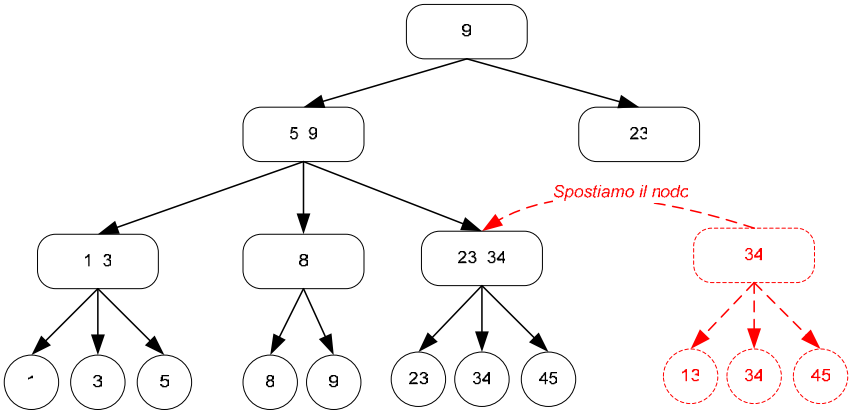
Consideriamo il seguente Albero 2-3. Eliminiamo la chiave 12.



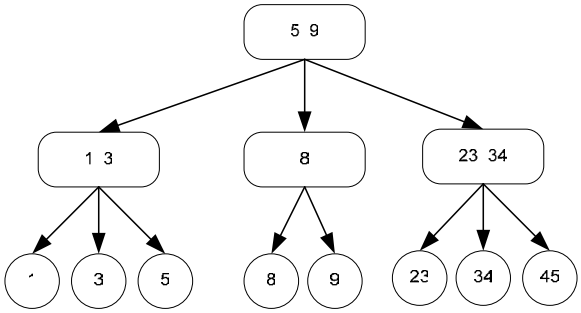
Poiché il padre del nodo contenente la chiave 12 non ha fratelli aventi tre figli dobbiamo ricorrere prima ad una operazione di fusione.



Ed infine richiamare la cancellazione sul nodo contenente il valore 12. Siccome il padre del nodo da cancellare non possiede fratelli con tre figli operiamo nuovamente una fusione



E successivamente la cancellazione sul nodo rimasto privo di figli. Otteniamo quindi il seguente albero 2-3



.

7. B-Alberi

Vedremo in questo capitolo una generalizzazione degli alberi binari di ricerca: la struttura dati B-Albero.

Il B-Albero nasce dalla necessità di disporre di una struttura dati che operi al meglio sui dispositivi di memorizzazione di massa, in cui, al fine di ammortizzare il tempo di latenza dell'accesso fisico, le operazioni di I/O avvengono mediante la lettura/scrittura di blocchi di informazioni di dimensione fissata, le pagine.

Vedremo, infatti, che in un B-Albero le informazioni del dizionario sono raggruppate in blocchi di dati ed ognuno di questi blocchi rappresenta un nodo dell'albero. La definizione di un B-Albero avente i nodi di dimensione proporzionale al blocco di lettura/scrittura comporta un numero di accessi minori all'unità fisica qualora memorizziamo le informazioni del B-Albero all'interno dei dispositivi di memoria di massa.

Un B-Albero, quindi, potrebbe rappresentare una struttura dati valida per l'implementazione di un dizionario quando per la memorizzazione delle coppie *(chiave,elemento)* facciamo uso di tali dispositivi meccanici.

Un esempio di applicazione pratica dei B-Alberi è dato dalla metodologia di memorizzazione delle informazioni adottata dai Database Server.

Definizione. Un B-Albero non vuoto di grado minimo t è un albero in cui:

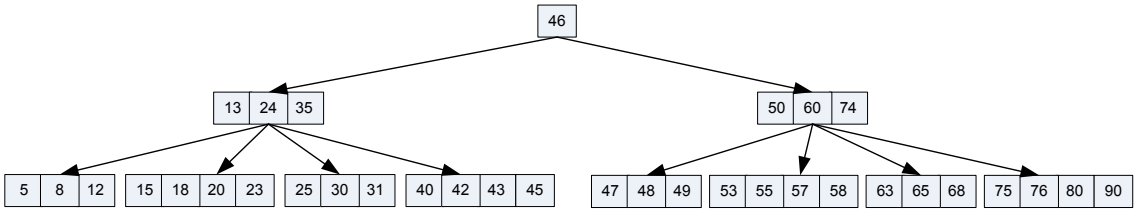
1. tutte le foglie appartengono allo stesso livello;
2. nella radice sono memorizzate in ordine crescente un numero di chiavi pari ad uno o al massimo pari a $2t-1$;
3. in tutti i nodi diversi dalla radice, v, sono memorizzate in ordine crescente un numero di chiavi $Key(v)$ tali che $t-1 \leq Key(v) \leq 2t-1$;
4. ogni nodo interno, v, ha $Key(v)+1$ figli;
5. ogni chiave di un nodo interno separa le chiavi dei sottoalberi del nodo, cioè indichiamo con $k_i(v)$ una chiave qualsiasi contenuta nello i-esimo sottoalbero di v e con $Key_i(v)$ la i-esima chiave contenuta nel nodo v si deve avere che:

$$k_1(v) \leq Key_1(v) \leq k_2(v) \leq Key_2(v) \leq \ldots \leq Key_{Key(v)}(v) \leq k_{Key(v)+1}(v)$$

Si dice che un nodo è ***quasi vuoto*** se ha un numero di chiavi pari a *t-1*. Si dice che un nodo è ***pieno*** se ha un numero di chiavi pari a *2t-1*.

Esempio

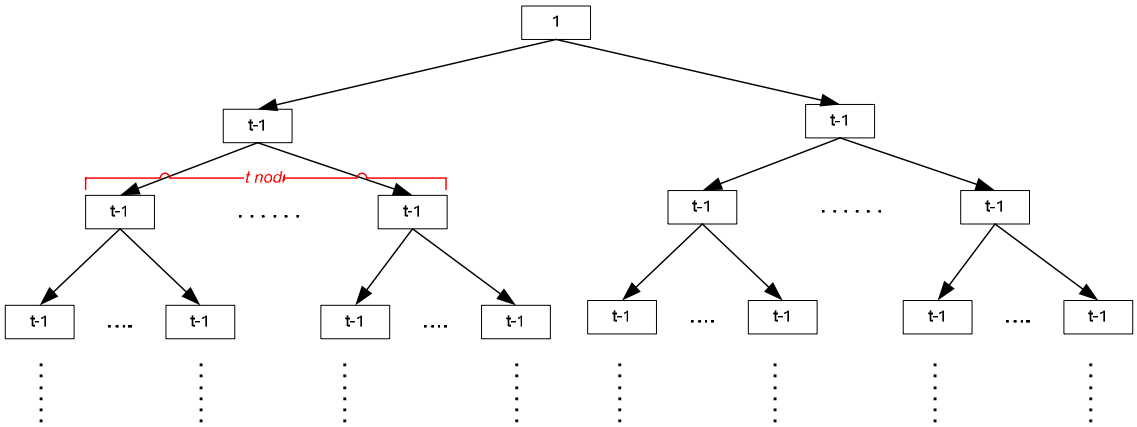
Segue una raffigurazione di un B-Albero di grado t=3.



Teorema. *Sia h l'altezza di un B-Albero, T, di grado minimo t. Sia n il numero di chiavi contenute nel B-Albero T. Si ha che:*

$$h \leq \log_t\left(\frac{n+1}{2}\right)$$

Dim. Un B-Albero, T_1, di grado *t* ed altezza *h* contenente il minor numero possibile di chiavi è un B-Albero in cui tutti i nodi sono quasi vuoti, ossia contengono *t-1* chiavi, e la radice contiene una sola chiave.



In tal caso:

- il numero di nodi di T_1 ad altezza zero è 1;
- il numero di nodi di T_1 ad altezza uno è due: i due figli della radice;
- il numero di nodi di T_1 ad altezza due è $2t$: t figli per ogni figlio della radice;
- il numero di nodi di T_1 ad altezza tre è $2t*t=2t^2$

Più genericamente, il numero di nodi di T_1 ad altezza i è, $2t^{i-1}$.

Considerando ora che ogni nodo possiede *t-1* chiavi e che la radice possiede una sola chiave abbiamo che il numero di chiavi presenti all'interno di T_1 è:

$$key(T_1) = 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1}$$

Sia ora T un generico B-Albero, di grado t e di altezza h. Abbiamo che il numero di chiavi di T, che indichiamo con n, è maggiore o al massimo uguale al numero di chiavi di T_1, che come abbiamo assunto rappresenta un B-Albero di grado t ed altezza h contenente il minor numero possibile di chiavi.
Quindi:

$$n \geq key(T_1) = 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1}$$

ed essendo vera la seguente uguaglianza:

$$\sum_{i=1}^{h} t^{i-1} = \sum_{i=0}^{h-1} t^{i} = \frac{t^h - 1}{t-1}$$

otteniamo che:

$$n \geq 1 + 2(t-1)\sum_{i=1}^{h} t^{i-1} = 1 + 2(t-1)\frac{t^h - 1}{t-1} = 1 + 2t^h - 2 = 2t^h - 1 \Rightarrow$$

$$\Rightarrow 2t^h \leq n + 1 \quad \Rightarrow$$

$$\Rightarrow t^h \leq \frac{n+1}{2} \quad \Rightarrow$$

$$\Rightarrow \quad h \leq \log_t\left(\frac{n+1}{2}\right)$$

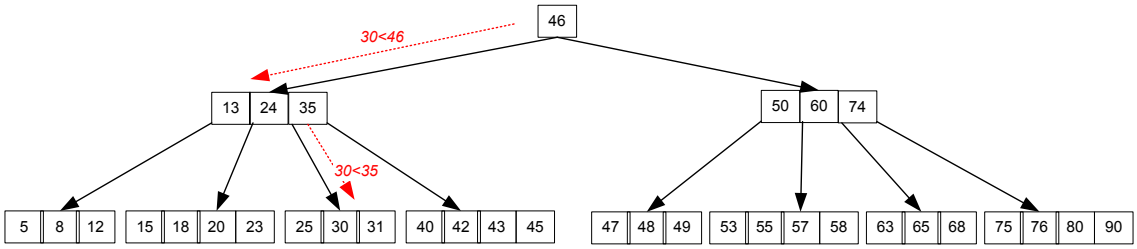
Vediamo ora le operazioni del dizionario: ricerca, inserimento e cancellazione.

Ricerca

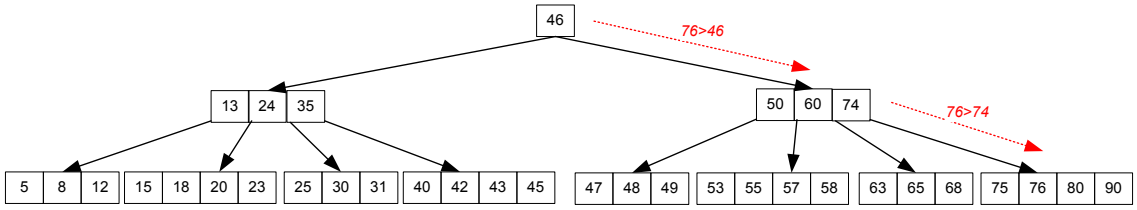
Ricerchiamo una chiave all'interno di un B-Albero mediante l'applicazione di una metodologia simile a quella sviluppata per la ricerca di una chiave all'interno di un albero binario di ricerca. In un B-Albero la decisione non sarà più binaria ma Θ(t)-aria, poiché ogni nodo ha un numero di figli compreso tra t e $2t$. Sia k una chiave da ricercare, navighiamo all'interno dell'albero identificando per ogni nodo v la più piccola chiave maggiore di k, $Key_i(v)$. Se $Key_i(v)$ esiste proseguiamo nella navigazione attraversando il sottoalbero $T_i(v)$ radicato in v, altrimenti proseguiamo attraversando il sottoalbero più a destra radicato in v.

Esempio

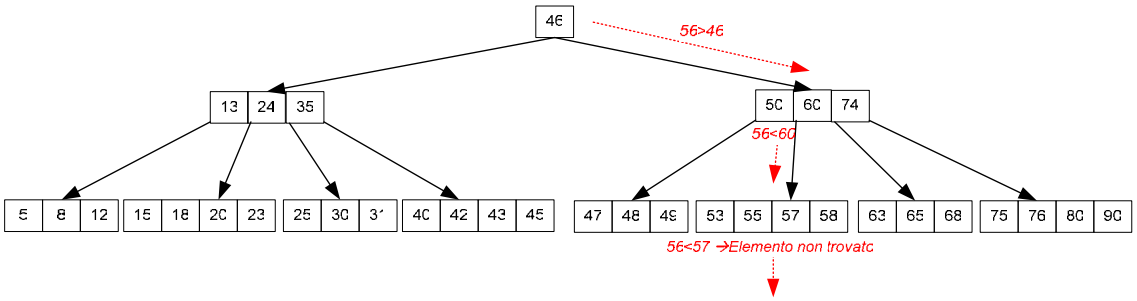
Sia considerato li seguente B-Albero.Ricerchiamo la chiave 30.



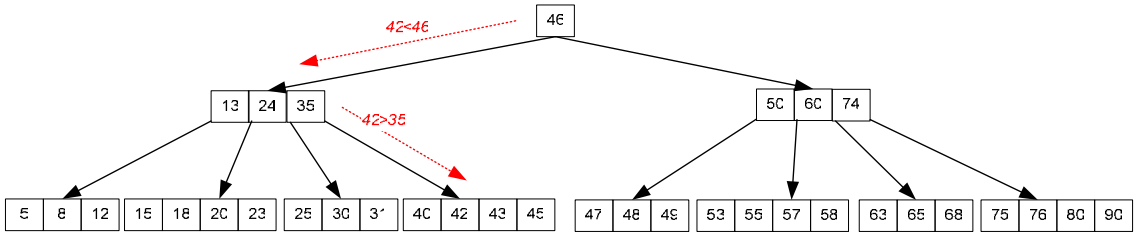
Ricerchiamo la chiave 76



Ricerchiamo la chiave 56



Ricerchiamo la chiave 42



Valutiamo la complessità dell'operazione di ricerca. Nel caso peggiore scendiamo lungo un intero verso del B-Albero, fino a raggiungere una foglia. Il numero di passi da eseguire è quindi chiaramente proporzionale con l'altezza dell'albero: *O(h)*. In virtù del teorema sopra dimostrato possiamo affermare che in termini di numero di nodi eseguiamo $O(\log_t n)$ passi di discesa. Ad ogni passo di discesa, dobbiamo dar luogo ad un confronto tra la chiave da ricercare e le chiavi contenute nel nodo. Facendo leva sull'ordinamento delle chiavi, applichiamo l'algoritmo di ricerca binaria e realizziamo tale fase con complessità pari a *O(log t)*.
Pertanto, la complessità dell'intera operazione di ricerca è:

$$O(\log_t n) * O(\log t) = O(\log_t n * \log t)$$

applicando la regola di trasformazione della base di un logaritmo abbiamo *O(log n)*.

Inserimento

Inseriamo sempre una nuova chiave in un B-Albero all'interno di una foglia. La procedura di inserimento richiede pertanto dapprima l'identificazione del nodo foglia che potrebbe rappresentare il giusto contenitore. Attuiamo questa fase applicando la medesima metodologia di navigazione vista nell'operazione di ricerca di una chiave.
Sia ora *v* il nodo foglia che rappresenta il giusto contenitore. Possiamo incontrare i seguenti due casi:

- *v* è non pieno, *v* ha meno di *2t-1* figli e pertanto inseriamo semplicemente la nuova chiave all'interno del nodo mantenendo però inalterate le proprietà di ordinamento.

- *v* è pieno, *v* possiede già *2t-1* chiavi. Non possiamo semplicemente aggiungere un altro figlio. Dobbiamo ricorrere pertanto alla tecnica della suddivisione (*split*): creiamo un nuovo nodo *w*, inseriamo in esso le prime *t-1* chiavi di *v* ed infine promuoviamo la *t-esima* chiave di *v* spostandola all'interno del padre di *v*. Le rimanenti *t* chiavi continueranno ad appartenere al nodo *v*.
  Se il padre di *v* non era precedentemente pieno, l'operazione di inserimento si arresta. Altrimenti richiamiamo l'operazione di split sul padre di *v*.

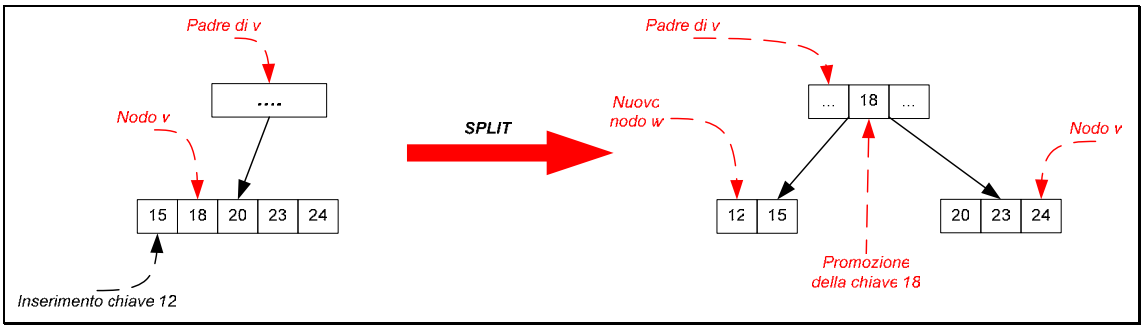
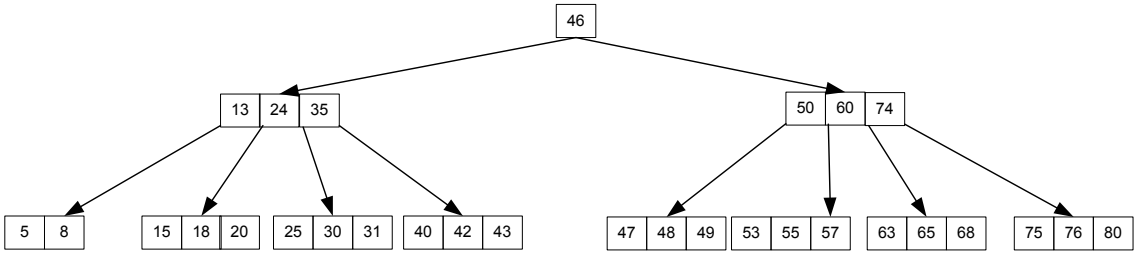


Figura - Raffigurazione dell'operazione di split in un B-Albero di grado minimo 3

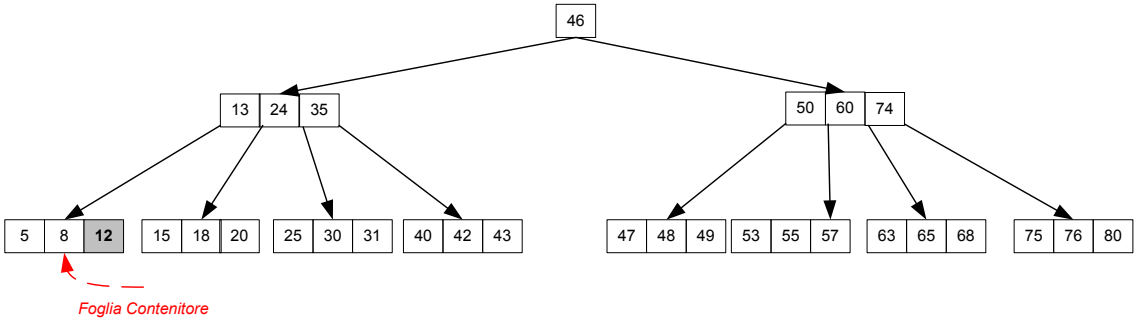
Valutiamo la complessità dell'operazione di inserimento. La ricerca del contenitore foglia e l'inserimento della chiave in esso ha, sulla base di quanto detto nell'analisi dell'operazione di ricerca di una chiave, complessità nel caso pessimo pari a $O(\log n)$. Relativamente alle operazioni di ripristino della struttura dell'albero osserviamo dapprima che una singola operazione di split ha banalmente complessità costante rispetto al numero delle chiavi presenti nell'albero. Sulla base di questa osservazione possiamo affermare che nel caso pessimo, esecuzione delle operazioni di split per ogni nodo incontrato lungo il cammino che collega la foglia e la radice, la complessità dell'intera operazione di ripristino della struttura dell'albero è $O(h)$, ossia $O(\log_t n)$. Essendo t un valore maggiore o al massimo uguale a due, abbiamo che l'operazione di inserimento nel caso pessimo ha una complessità pari a $O(\log n)$.

Esempio

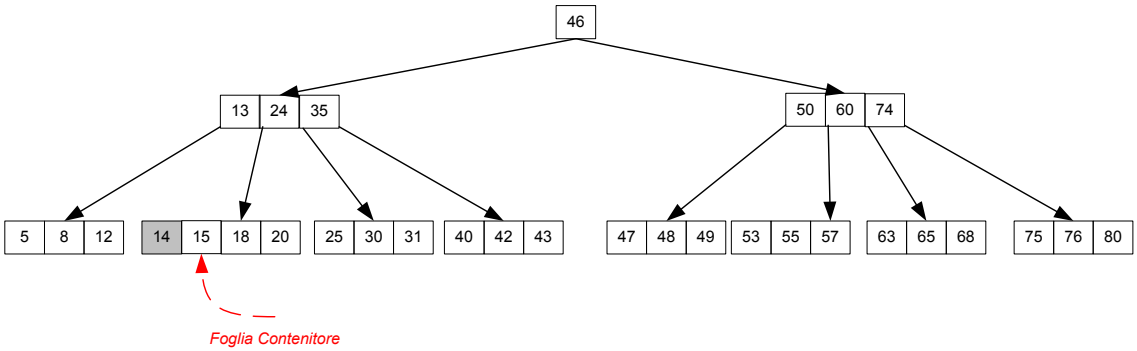
Sia considerato il B-Albero di grado minimo t=2 raffigurato in basso.



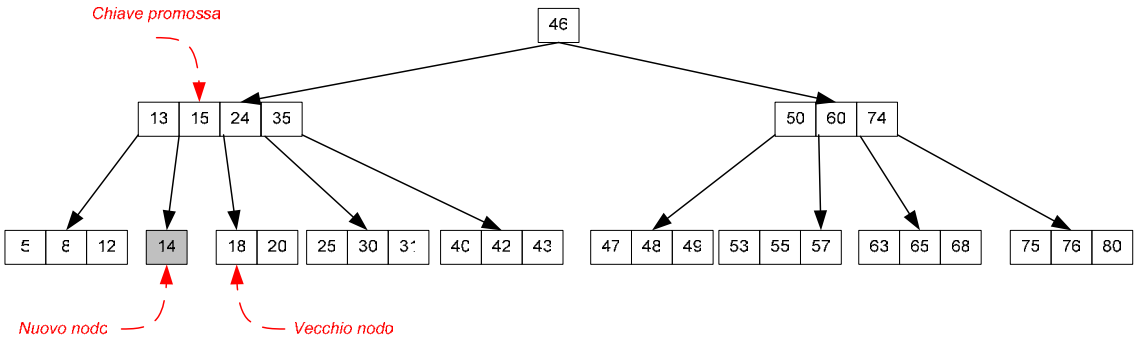
Inseriamo la chiave 12



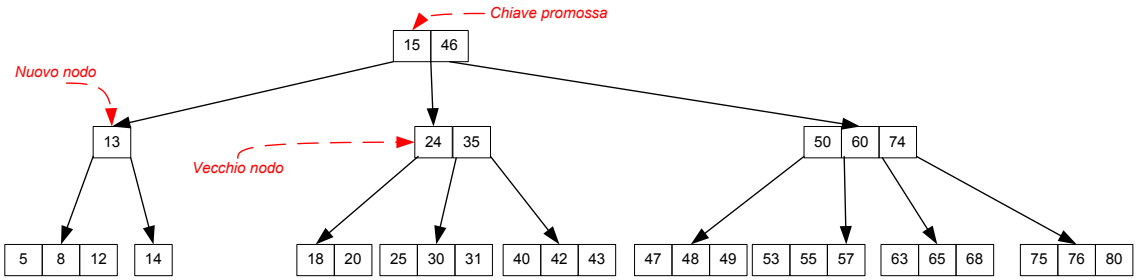
La foglia contenitore non era precedentemente piena. L'intera operazione si risolve quindi in un semplice inserimento della chiave all'interno della foglia contenitore.
Inseriamo la chiave 14.



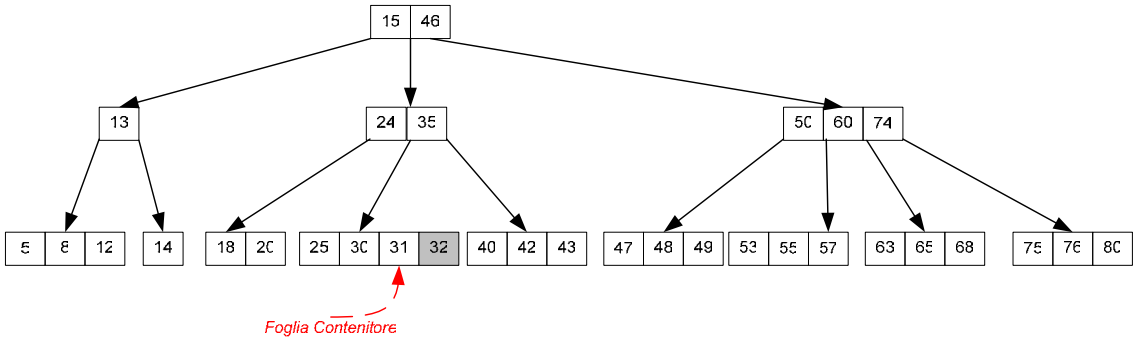
Questa volta la foglia contiene un numero di chiavi non lecite:4 > 2t-1 = 3. Dobbiamo pertanto ricorrere allo split del nodo.



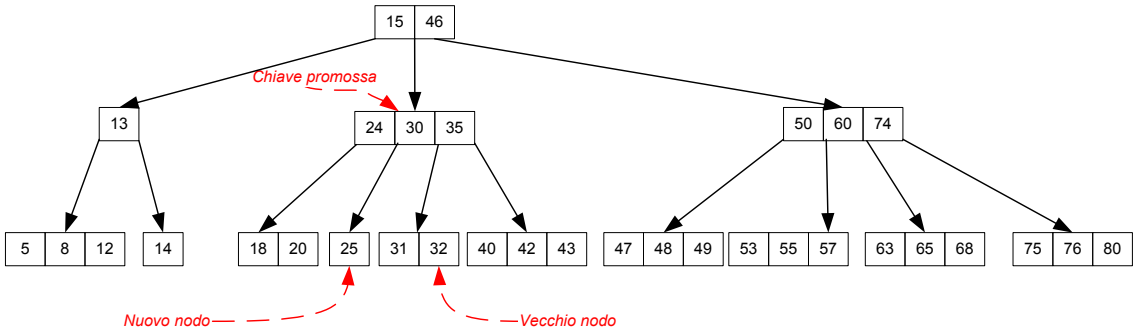
Il nodo all'interno del quale è stata collocata la chiave promossa contiene a sua volta un numero di chiavi non lecite. Ricorriamo nuovamente allo split.



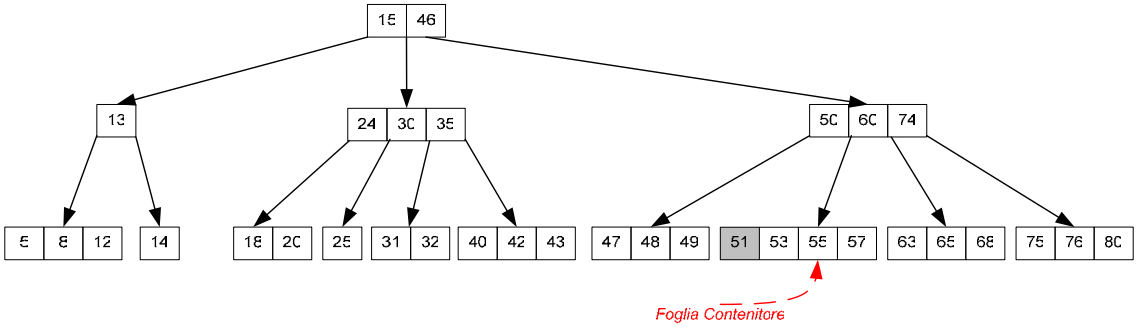
Inseriamo la chiave 32



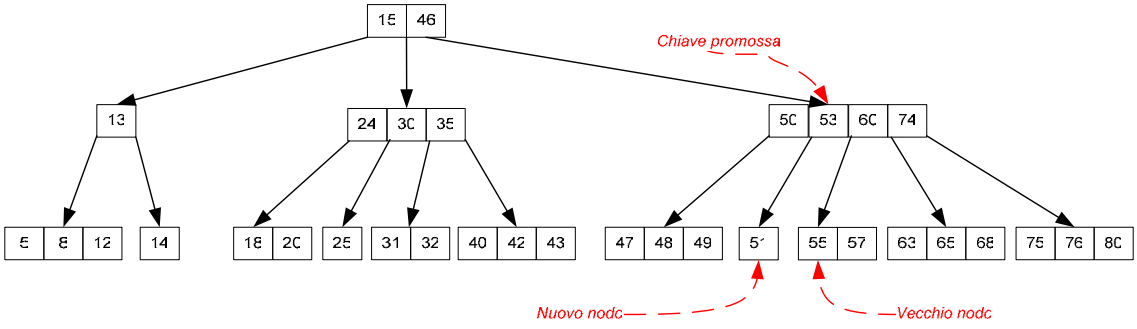
La foglia contenitore possiede un numero di chiavi non più lecito. Ricorriamo alla operazione di split.



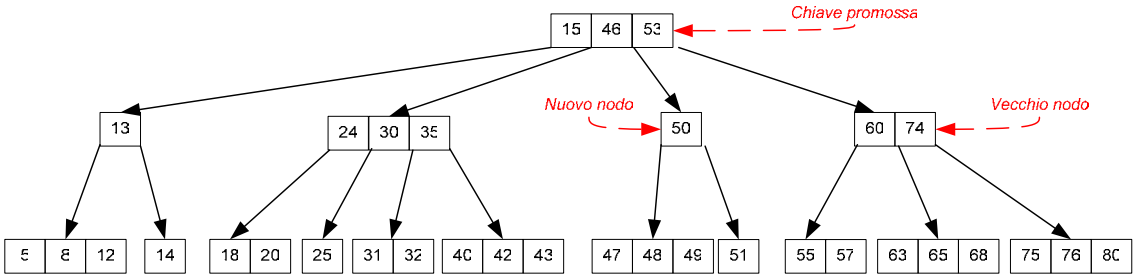
Il nodo in cui inseriamo la chiave promossa continua a possedere un numero di chiavi lecito. L'operazione di inserimento termina. Inseriamo ora la chiave 51.



Ricorriamo nuovamente alla operazione di split.



L'operazione di inserimento della chiave 51 ancora non termina poiché si rende necessaria una ulteriore operazione di split da applicare sul nodo interno in cui abbiamo inserito la chiave promossa.



Cancellazione

Quando cancelliamo un nodo da un B-Albero dobbiamo verificare se al termine dell'operazione l'albero continua a possedere le proprietà elencate ad inizio capitolo. In caso non affermativo, similmente agli alberi 2-3, si potrebbe rendere necessaria l'esecuzione della procedura di fusione.

Indichiamo con *v* il nodo contenente la chiave da cancellare. Si possono verificare i seguenti casi:

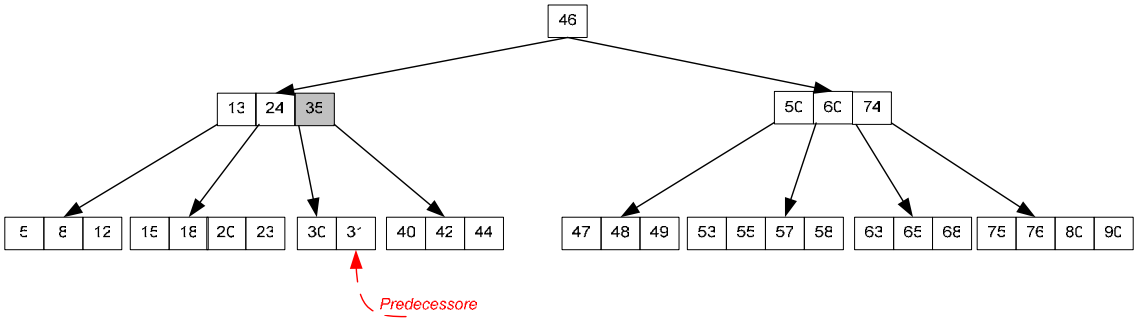
1. *v* è un nodo interno, in tal caso cerchiamo il nodo contenente il predecessore di *k*: *pred(k)*. Dopodiché copiamo *pred(k)* all'interno di *v* ed al posto di *k* e richiamiamo la cancellazione del *pred(k)* che è sicuramente contenuto in una foglia. Ricadiamo, pertanto, in questa seconda cancellazione in uno dei casi a seguire;
2. *v* è una foglia non quasi vuota, in tal caso *v* contiene almeno *t* chiavi. Pertanto cancelliamo semplicemente la chiave posta all'interno di *v*;
3. *v* è una foglia quasi vuota, in tal caso *v* contiene *t-1* chiavi. Esaminiamo i fratelli di *v*. Si possono verificare i seguenti sottocasi:
   a. il fratello sinistro (destro) di *v* è non quasi vuoto. Inseriamo la chiave massima (minima) del fratello sinistro (destro) di *v* come chiave separatrice dei due nodi all'interno del nodo padre. Infine inseriamo la chiave separatrice del padre come elemento minimo (massimo) di *v*.
   b. tutti i fratelli di *v* sono quasi vuoti. Copiamo all'interno di uno dei fratelli di *v*, che indichiamo con *w*, e rispettando l'ordine crescente delle chiavi, la chiave separatrice dei nodi *v* e *w* e tutte le chiavi contenute in *v* (*fuse*). Infine cancelliamo il nodo *v*. In tal modo il padre di *v* perde una chiave e contemporaneamente perde anche un figlio, la proprietà 4 è rispettata.

L'operazione di cancellazione termina se il padre di *v* non diventa un nodo quasi vuoto. Altrimenti essa continua con la propagazione verso l'alto dell'operazione di fuse.

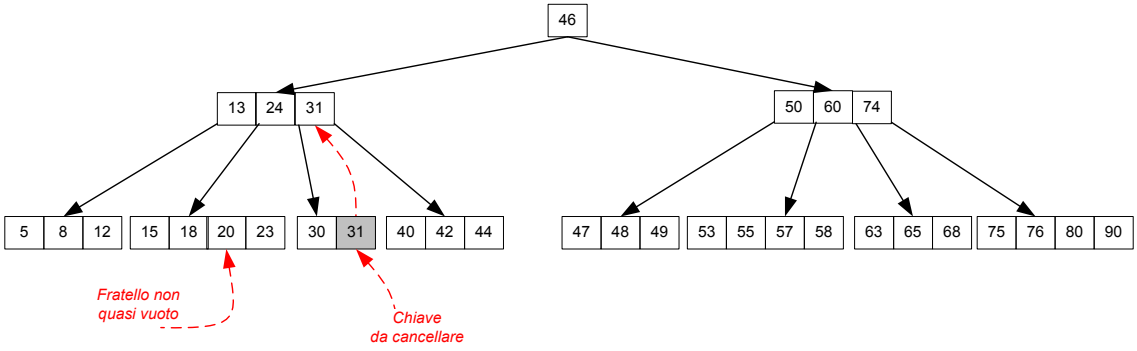
Compiendo una analisi simile a quella fatta per l'operazione di inserimento e considerando che l'operazione di fusione anche se laboriosa è comunque costante rispetto al numero di chiavi presenti nel B-Albero, possiamo affermare che la complessità dell'intera procedura di cancellazione è pari a *O(log n)*.

Esempio

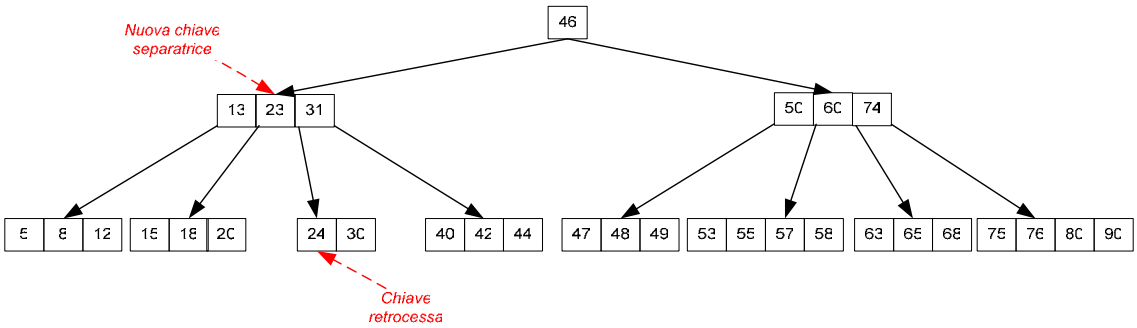
Consideriamo il B-Albero di grado t=3 di seguito raffigurato. Procediamo nella cancellazione della chiave35.



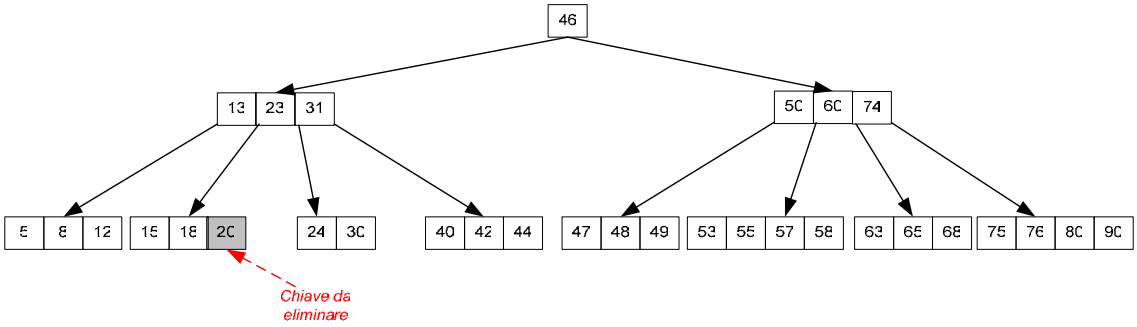
Il predecessore di 35 è la chiave 31. Per cui sostituiamo la chiave 35 con la chiave 31 e cancelliamo la chiave 31 presente nella foglia.



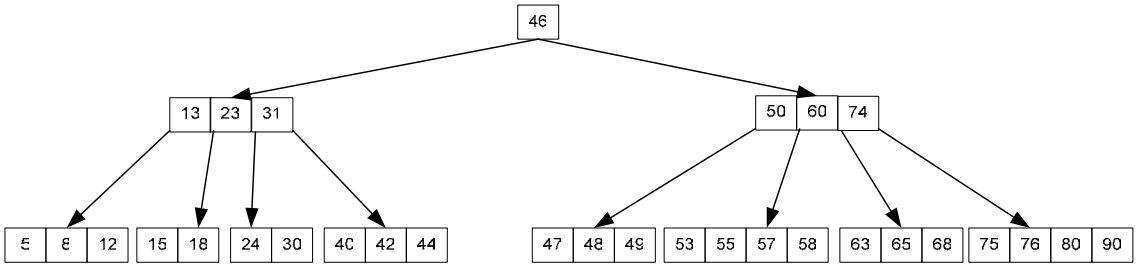
Il nodo foglia contenente la chiave 31 possiede un fratello non quasi vuoto. Promuoviamo quindi la chiave più grande presente in tale nodo, 23, e retrocediamo la chiave 24



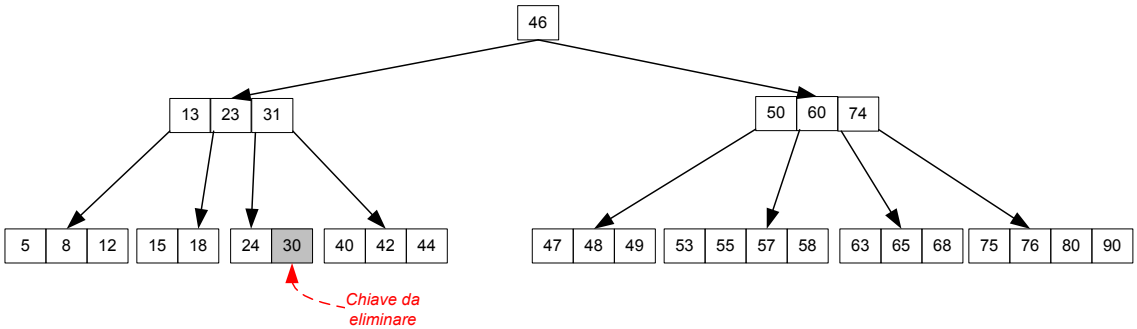
Cancelliamo ora la chiave 20.



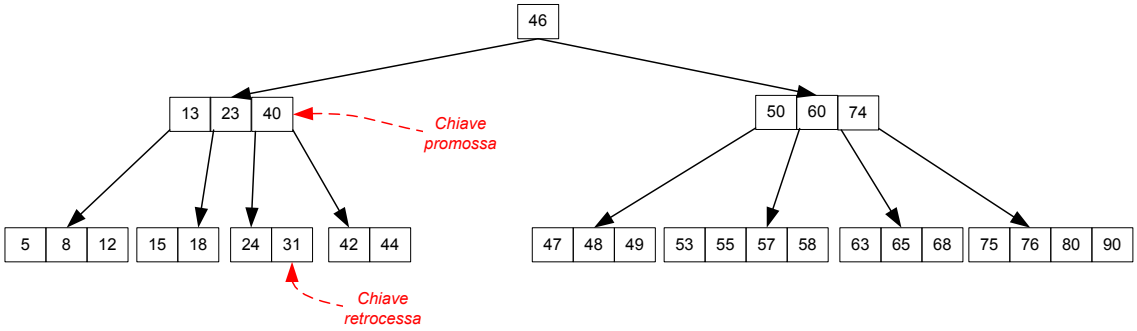
Il nodo contenente tale chiave è non quasi vuoto. L'operazione di cancellazione consiste quindi in una semplice rimozione del valore.



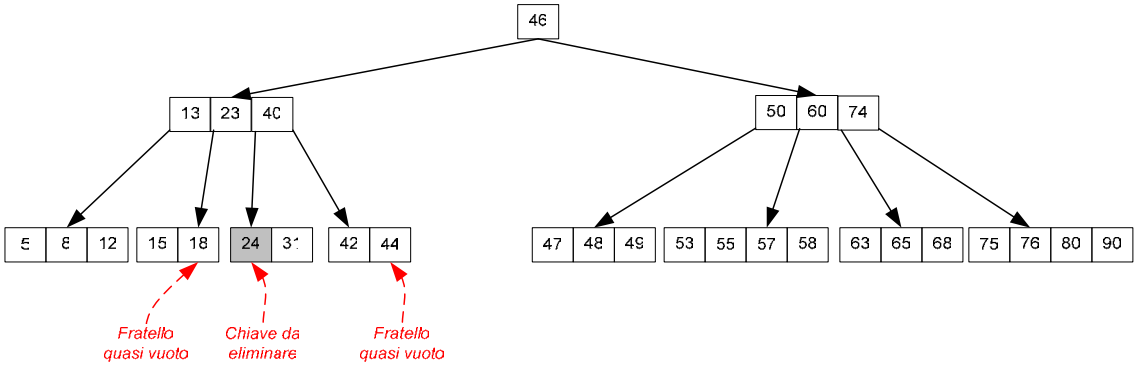
Cancelliamo la chiave 30.



Il fratello destro del nodo contenente la chiave 30 è non quasi vuoto, pertanto promuoviamo la chiave minima del nodo non quasi vuoto e retrocediamo il separatore dei due nodi.



Cancelliamo la chiave 24



Il nodo contenente la chiave 24 ha entrambi i fratelli quasi vuoti. Consideriamo il fratello sinistro e procediamo nella fusione. Diamo luogo quindi ad una retrocessione della chiave separatrice dei due nodi e spostiamo le chiavi del nodo da eliminare all'interno del fratello sinistro.