1. Alberi bilanciati di ricerca

Gli alberi sono una delle strutture di dati più efficienti per realizzare dizionari, in modo particolare quando si vuole mantenere la possibilità di scorrere le chiavi presenti nella struttura secondo un qualche ordinamento. Affinché questo possa accadere, i nodi dell'albero devono essere dotati di informazioni di chiave e di valore, e la struttura dell'albero deve permettere di effettuare le ricerche. Vedremo quindi le caratteristiche minime necessarie per la ricerca, e studieremo poi come rendere efficienti ricerche, inserzioni e cancellazioni.

1.1 Alberi binari di ricerca

Le associazioni chiave-valore di un dizionario D possono essere contenute nei nodi di un albero binario B, detto *albero binario di ricerca* o ABR, che soddisfa le seguenti proprietà:

(1) per ogni nodo u, tutte le chiavi contenute nel sottoalbero radicato nel figlio sinistro di u sono minori della chiave contenuta in u;

(2) per ogni nodo u, tutte le chiavi contenute nel sottoalbero radicato nel figlio destro di u sono maggiori della chiave contenuta in u.

Esempio (Albero binario di ricerca) Un ABR è mostrato nella Figura 1.1. Si noti che con una visita simmetrica dell'albero si esaminano le chiavi in ordine crescente. ■

Le proprietà (1) e (2) hanno lo scopo di agevolare la ricerca di un elemento. È sufficiente confrontare l'elemento cercato con quello contenuto nella radice e, se sono diversi, riapplicare il procedimento al sottoalbero radicato nel figlio sinistro o destro, secondo l'esito del confronto. Questo procedimento è molto simile a quello effettuato nella ricerca binaria. Come il nome suggerisce, tale ricerca genera infatti partizioni successive degli elementi secondo uno schema coincidente con un albero binario.

Assumendo che l'albero binario sia realizzato con puntatori [cfr. Capitolo 5], le procedure lookupNode(), insertNode() e removeNode() cercano, inseriscono e rimuovono nodi da alberi binari di ricerca, e possono essere utilizzate per realizzare le relative operazioni degli insiemi e dei

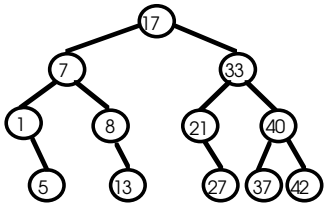

Figura 1.1: Un albero binario di ricerca.

dizionari. Per illustrare un approccio complementare rispetto a quello utilizzato introducendo gli alberi binari, proponiamo una versione iterativa delle procedure suddette.

La procedura lookupNode() cerca il nodo contenente la chiave x nell'albero binario di ricerca radicato in T; restituisce **nil**, nel caso che la chiave non sia presente, il puntatore al nodo contenente x, altrimenti. La complessità della procedura è $O(h)$, dove h è l'altezza dell'albero.

```
TREE lookupNode(TREE T, item x)
  while T ≠ nil and T.key ≠ x do
    T = iif(x < T.key, T.left, T.right)
  return T
```

Per inserire associazioni chiavi-elemento, la procedura insertNode() effettua innanzitutto una ricerca, in modo simile a quanto visto per lookupNode(), per localizzare il nodo u contenente l'elemento (in caso di modifica dell'associazione corrente) o il nodo p che diventerà il padre del nuovo nodo contenente l'elemento (in caso di inserzione). Nel caso il nodo sia trovato, viene aggiornato solo il campo *value*; altrimenti viene creato un nuovo nodo, e questo viene agganciato come figlio sinistro o destro di p, a seconda del valore della chiave, dalla procedura di appoggio link(). insertNode() restituisce un puntatore all'albero, che corrisponde all'albero esistente, se questo conteneva già almeno un nodo, o corrisponde al nodo creato, se questo è il primo a popolare l'albero (ovvero se $p =$ **nil**). Come prima la complessità è $O(h)$, ma si noti che l'altezza può crescere di uno.

```
TREE insertNode(TREE T, item x, item v)
  TREE p = nil                                          % Padre
  TREE u = T
  while u ≠ nil and u.key ≠ x do                        % Cerca posizione inserimento
    p = u
    u = iif(x < u.key, u.left, u.right)
  if u ≠ nil and u.key = x then
    u.value = v                                         % Chiave già presente
  else
    TREE n = Tree(x, v)          % Crea un nodo contenente coppia chiave-valore
    link(p, n, x)
    if p = nil then
      return n                                          % Primo nodo ad essere inserito
  return T                                              % Restituisce albero non modificato
```

```
link(TREE p, TREE u, item x)
if u ≠ nil then u.parent = p
if p ≠ nil then
    if x < p.key then p.left = u          else p.right = u
```

La procedura link(), utilizzata anche dalla procedura removeNode(), gestisce le due azioni necessarie per collegare un nodo padre p ad un nodo figlio u: modificare il puntatore al padre di u, e inserire u come figlio sinistro o destro di p, a seconda del valore del parametro x che viene confrontato con il contenuto del padre. Come casi particolari, p può essere **nil**, che significa che u è la nuova radice; oppure u può essere **nil**, che significa che un nodo è stato cancellato e **nil** sostituisce il figlio corrispondente nel padre. link() ha costo $O(1)$.

Per realizzare removeNode() si procede in modo analogo all'inserzione, con una complicazione in più data dal fatto che l'elemento x da cancellare si trova in un nodo u che non necessariamente è una foglia. Possono sorgere tre casi, illustrati nella Figura 1.2.

(1) Se u non ha figli, allora u è una foglia che viene semplicemente rimossa. C'è da gestire tuttavia il caso particolare in cui u sia l'unico nodo dell'albero.
(2) Se u ha un solo figlio f, allora u è rimosso rendendo f figlio del padre di u.
(3) Se u ha due figli, ci si può ridurre al caso (1) o (2) sostituendo l'elemento da cancellare x con il più piccolo elemento y tale che $y > x$, che si trova nel nodo s raggiungibile scendendo sempre a sinistra nel sottoalbero radicato nel figlio destro di u. L'elemento y prende il posto di x nel nodo u mentre viene rimosso il nodo s che, per definizione, non può avere un figlio sinistro.

```
TREE removeNode(TREE T, item x)
TREE u = lookupNode(T,x)
if u ≠ nil then
    if u.left ≠ nil and u.right ≠ nil then                    % Caso 3
        TREE s = u.right
        while s.left ≠ nil do s = s.left
        u.key = s.key
        u.value = s.value
        x = s.key
        u = s
    TREE t
    if u.left ≠ nil and u.right = nil then                    % Caso (2) - Solo figlio sx
        t = u.left
    else                                                      % Caso (2) - Solo figlio dx / Caso (1)
        t = u.right
    link(u.parent,t,x)
    if u.parent = nil then T = t
    delete u
return T
```

La procedura removeNode() restituisce il puntatore alla radice dell'albero, che può cambiare se il nodo cancellato è proprio la radice. La procedura inizia cercando tramite lookupNode() il nodo da cancellare u; la radice viene mantenuta nel puntatore T. Se u non è stato trovato, la procedura termina restituendo la radice inalterata.

A questo punto, la procedura è organizzata nel modo seguente:

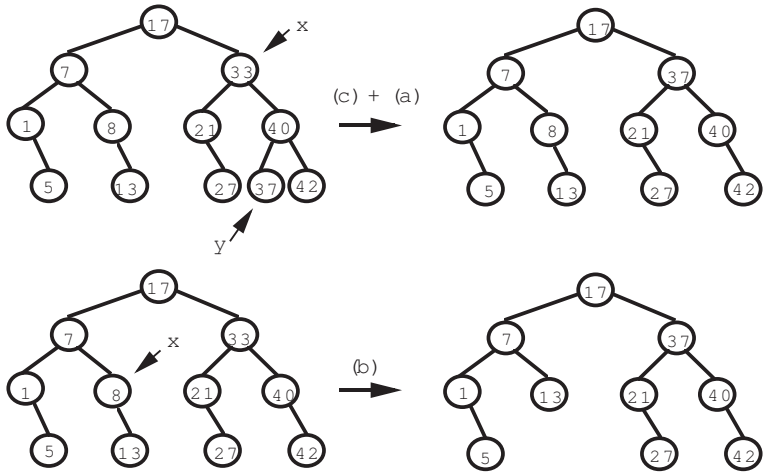

Figura 1.2: I tre casi di cancellazione di un elemento x da un albero binario di ricerca. **TODO**: "(c)+(a)" → "(3)+(1)"; "(b)" → "(2)", "x" → "u", "y" → "s"

- il caso (3) appare per primo, in quanto si riduce al caso (1) o (2); innanzitutto viene identificato il nodo s e la sua associazione chiave-valore viene ricopiata nel nodo u; a questo punto, il puntatore u assume il valore di s e si passa a gestire la cancellazione di s tramite i casi (1) e (2).
- il caso (2) con solo un figlio sinistro è semplice; il figlio sinistro viene collegato al padre di u; se il padre è assente, u era radice e il suo figlio sinistro diventa la nuova radice.
- il caso (2) con solo un figlio destro è simmetrico; il figlio destro viene collegato al padre di u; se il padre è assente, u era radice e il suo figlio destro diventa la nuova radice. Si noti che in pratica questo codice gestisce anche il caso (1), dove però il figlio destro di u è **nil**. Se sia il padre di u che il suo figlio destro sono **nil**, l'albero è ora vuoto e la nuova radice diventa per l'appunto **nil**.

La memoria associata al nodo u viene rimossa e la radice (potenzialmente modificata) viene restituita al chiamante. Richiedendo al massimo una discesa radice-foglia, anche questa procedura ha costo $O(h)$.

A differenza delle tabelle hash che verranno presentate nel prossimo capitolo, gli alberi di ricerca mantengono l'ordinamento fra i valori. Un operatore min() che restituisca il nodo contenente il minimo può essere realizzato facilmente in tempo $O(h)$. Basta infatti effettuare una ricerca in cui si segue sempre il figlio sinistro, fino a trovare un nodo che non ha figlio sinistro. L'operatore max() è simmetrico, basta seguire sempre il figlio destro.

```
TREE min(TREE T)
    while T.left ≠ nil do
        T = T.left
    return T
```

```
TREE max(TREE T)
    while T.right ≠ nil do
        T = T.right
    return T
```

```
TREE successorNode(TREE t)
    if t = nil then
        return t
    if t.right ≠ nil then
        return min(t.right)
    TREE p = t.parent
    while p ≠ nil and t = p.right do
        t = p
        p = p.parent
    return p
```

```
TREE predecessorNode(TREE t)
    if t = nil then
        return t
    if t.left ≠ nil then
        return max(t.left)
    TREE p = t.parent
    while p ≠ nil and t = p.left do
        t = p
        p = p.parent
    return p
```

Oltre a questi operatori, dato un nodo t (non necessariamente la radice), può essere interessante ottenere il successore/predecessore di t nell'albero, ovvero il nodo la cui chiave segue/precede immediatamente $t.key$ nell'ordinamento. Per trovare il successore di t, sono dati due casi:

- Se t ha un figlio destro, il successore è il minimo u del sottoalbero destro;
- altrimenti, il successore è il primo antenato u di t per cui t sta nel sottoalbero sinistro di u.

Il codice di successorNode() segue semplicemente i due casi; il codice di predecessorNode() è simmetrico.

Con l'operatore min() e la procedura successorNode() è possibile scorrere tutte le chiavi e/o i valori presenti nell'albero T secondo l'ordinamento utilizzando codice come questo:

```
TREE u = min(T)
while u ≠ nil do
    { Elabora u }
    u = successorNode(u)
```

Questo codice ha complessità ottima $O(n)$, in quanto tutti i nodi vengono visitati al massimo due volte: una volta scendendo lungo la direzione radice-foglia, alla ricerca di un minimo; e una volta salendo lungo la direzione foglia-radice, alla ricerca di un antenato.

Usando un ABR, la complessità delle singole operazioni lookupNode(), insertNode(), removeNode() è sempre $O(h)$. Purtroppo, h è $O(n)$ nel caso pessimo, perché l'albero può "allungarsi" inserendo e cancellando nodi. Per fortuna, è possibile modificare la struttura introducendo opportuni metodi di "bilanciamento" che permettono di contenere h entro $O(\log n)$. Esistono diversi approcci di questo tipo, che qui semplicemente elenchiamo:

- Alberi AVL (Adelson-Velskii e Landis, 1962)
- B-alberi (Bayer, McCreight, 1972)
- Alberi Red-Black (Bayer, 1972)
- Alberi 2-3 (Hopcroft, 1983)
- B-alberi binari (Andersson, 1993)

Tutte queste strutture sono caratterizzate dalla stessa limitazione superiore $O(\log n)$ per l'altezza dell'albero. Nel seguito presenteremo solo gli alberi Red-Black, che sono fra quelli più utilizzati in pratica.

1.2 Alberi Red-Black

Un albero è *perfettamente bilanciato* se tutte le foglie si trovano allo stesso livello. Fissato un numero n di nodi e un numero massimo d di figli per nodo (ad esempio $d = 2$ per gli alberi binari), l'albero perfettamente bilanciato è quello con altezza minima tra tutti gli alberi con le stesse caratteristiche. Esempi di alberi perfettamente bilanciati sono i B-alberi e gli Alberi 2-3. Un albero

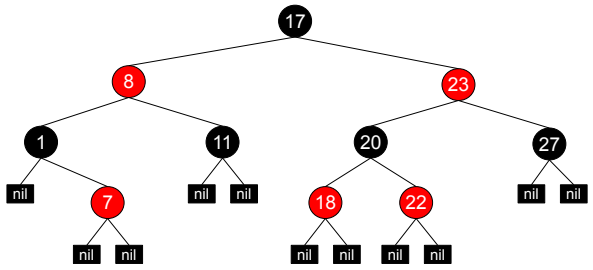

Figura 1.3: Un albero Red-Black con 10 nodi interni e 11 foglie.

è invece *k-bilanciato* se, per ogni nodo u, la massima differenza di altezza fra tutti i sottoalberi di u è pari a k. Gli alberi AVL sono un esempio di alberi 1-bilanciati.

Tutte queste strutture di dati cercano di minimizzare l'altezza dell'albero, al fine di limitare il numero di livelli che devono essere esplorati per trovare una chiave. Ma la rigidità delle loro strutture rende più costose le operazioni di inserimento e cancellazione, che richiedono complesse ristrutturazioni dell'albero per evitare sbilanciamenti. Gli alberi Red-Black in questa situazione rilassano i vincoli di bilanciamento, riducendo le operazioni necessarie di ristrutturazione, senza tuttavia rinunciare ad un'altezza limitata superiormente da $O(\log n)$.

Un albero Red-Black è un ABR in cui ogni nodo ha un attributo aggiuntivo *color*, che può essere rosso (RED) o nero (BLACK). Oltre ai requisiti soliti degli alberi binari di ricerca, per gli alberi Red-Black devono valere i seguenti vincoli:

(V_1) la radice è nera;
(V_2) tutte le foglie sono nere;
(V_3) entrambi i figli di un nodo rosso sono neri;
(V_4) tutti i cammini semplici da ogni nodo u ad una delle foglie contenute nel sottoalbero radicato in u hanno lo stesso numero di nodi neri.

Le foglie di un albero Red-Black non contengono alcun dato. In realtà, non è necessario memorizzare esplicitamente queste foglie nella memoria del computer – un puntatore **nil** scritto in *left* (*right*) sarebbe sufficiente a codificare che il figlio sinistro (destro) è una foglia. Tuttavia, gli algoritmi e le dimostrazioni risultano semplificate se le foglie sono memorizzate esplicitamente. Per risparmiare memoria, a volte viene utilizzato un nodo sentinella che assume il ruolo di tutte le foglie; tutti i puntatori a foglie puntano quindi allo stesso nodo. La Figura 1.3 mostra un albero Red-Black con 10 nodi interni e 11 foglie.

Vediamo ora come i vincoli Red-Black limitino il grado di sbilanciamento dell'albero e garantiscano un'altezza limitata superiormente. Il numero di nodi neri lungo ogni percorso da un nodo u (escluso) ad una qualsiasi foglia è detto *altezza nera* di u ed è indicato con $b(u)$. Questo valore è ben definito in quanto tutti i percorsi di questo tipo hanno lo stesso numero di nodi neri per il vincolo (V_4). L'altezza nera della radice è detta altezza nera dell'albero.

Teorema 1.1 Il sottoalbero con radice u contiene almeno $2^{b(u)} - 1$ nodi interni.

Dimostrazione. Si procede per induzione sull'altezza h dell'albero (non sull'altezza nera).

Nel *caso base*, $h = 0$. Allora u deve essere una foglia **nil** e il sottoalbero con radice in u contiene esattamente almeno $2^{b(u)} - 1 = 2^0 - 1 = 0$ nodi interni.

Nel *passo induttivo*, $h > 1$. Allora u è un nodo interno con due figli. Ogni figlio v ha un'altezza nera $b(v)$ pari a $b(u)$ o a $b(u) - 1$, a seconda del suo colore (se rosso: $b(u)$, se nero: $b(u) - 1$). Usando l'ipotesi induttiva, possiamo dire che ogni figlio ha almeno $2^{b(u)-1} - 1$ nodi interni. Quindi,

il sottoalbero con radice in u ha almeno:

$$2^{b(u)-1} - 1 + 2^{b(u)-1} - 1 + 1 = 2^{b(u)} - 1$$

nodi interni. Questo completa la dimostrazione. ■

Teorema 1.2 La lunghezza del più lungo cammino semplice radice-foglia non supera il doppio della lunghezza del più corto cammino semplice radice-foglia. Più formalmente, detto $\ell(u)$ il livello di un nodo u, vale la seguente proprietà:

$$\max\{\ell(u) : u.left = u.right = \textbf{nil}\} \leq 2 \cdot \min\{\ell(v) : v.left = v.right = \textbf{nil}\}$$

Dimostrazione. Si noti innanzitutto che nessun percorso può avere due nodi rossi in fila, a causa del vincolo (V_3). Tutti i cammini semplici radice-foglia hanno lo stesso numero di nodi neri per il vincolo (V_4); il più corto di essi è composto da soli nodi neri, mentre il più lungo alterna nodi rossi e neri, e quindi è lungo al massimo il doppio. ■

L'albero risultante non è quindi perfettamente bilanciato, tuttavia l'altezza dell'albero resta comunque limitata:

Teorema 1.3 L'altezza di un albero rosso-nero con n nodi interni è al più $2\log(n+1)$.

Dimostrazione. Per il Teorema 1.1, l'albero ha almeno $n \geq 2^{b(r)} - 1$, dove r è la radice. L'altezza dell'albero è al più il doppio dell'altezza nera della radice (come detto sopra, ogni nodo rosso deve essere seguito da uno nero), e quindi $n \geq 2^{h/2} - 1$. Semplificando, si ottiene $n+1 \geq 2^{h/2}$ e applicando il logaritmo ad entrambi i lati della disequazione si ottiene $h \leq 2\log(n+1)$. ■

Questo garantisce che tutte le operazioni di ricerca, inserimento e cancellazione possono essere completate in tempo $O(\log n)$, dove n è il numero di nodi.

Le operazioni lookupNode(), successorNode(), predecessorNode(), min() e max() non richiedono alcuna modifica rispetto al normale comportamento degli alberi binari di ricerca. Tuttavia, gli inserimenti o le cancellazioni possono violare i vincoli Red-Black; ripristinare tali vincoli può richiedere una quantità $O(\log n)$ di modifiche all'albero, ognuna delle quali con costo $O(1)$. Nel seguito illustreremo come effettuare le modifiche, illustrando solo informalmente la loro correttezza: per i dettagli precisi e le dimostrazioni formali (molto lunghe e macchinose) si faccia riferimento agli articoli originali.

Le modifiche possono essere di due tipi:

- *cambi di colore*, in cui la variabile *color* viene mutata;
- *rotazioni*, in cui la struttura dell'albero binario di ricerca viene cambiata, senza tuttavia violare le proprietà di ordinamento di tali alberi.

Le rotazioni muovono un nodo "verso l'alto" e un nodo "verso il basso"; lo scopo è ridurre lo sbilanciamento dell'albero, scegliendo una coppia di nodi tali per cui il nodo spostato in alto abbia un sottoalbero con altezza maggiore del sottoalbero del nodo spostato in basso.

Si parla di rotazione sinistra o destra, a seconda della direzione in cui i nodi vengono spostati. La Figura 1.4 illustra una rotazione sinistra a partire dal nodo x; la rotazione destra è speculare. L'idea è spostare x verso il basso e y verso l'alto; per farlo, bisogna:

(1) spostare il sottoalbero B come figlio destro di x;
(2) far diventare x il figlio sinistro di y;
(3) far diventare y figlio di p, il vecchio padre di x.

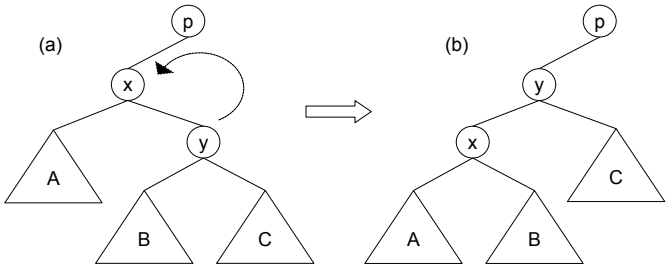

Figura 1.4: Rotazione a sinistra a partire dal nodo x. (a) Situazione iniziale. (b) Situazione finale.

Si noti che le proprietà dell'ABR non vengono violate. Infatti, i sottoalberi A e C restano figli sinistro di x e destro di y, rispettivamente; x diventa figlio sinistro di y, senza causare problemi in quanto la chiave in x è minore della chiave in y; infine, le chiavi contenuti in B sono maggiori della chiave in x, e vengono correttamente spostate nel sottoalbero destro di x.

Il codice associato a queste operazioni è illustrato nella procedura rotateLeft(); le righe sono marcate con (1), (2) e (3) per identificare le corrispondenti operazioni. La procedura restituisce il nodo che ha sostituito x, ovvero y. Il costo di una rotazione è ovviamente $O(1)$; la rotazione destra rotateRight() può essere ottenuta semplicemente scambiando fra loro ogni occorrenza delle variabili *left* e *right*.

```
TREE rotateLeft(TREE x)
    TREE y = x.right
    TREE p = x.parent
(1) x.right = y.left                    % Il sottoalbero B diventa figlio destro di x
(1) if y.left ≠ nil then y.left.parent = x
(2) y.left = x                          % x diventa figlio sinistro di y
(2) x.parent = y
(3) y.parent = p                        % y diventa figlio di p
(3) if p ≠ nil then
        if p.left = x then p.left = y
        else p.right = y
    return y
```

La procedura di inserimento insertNode() va modificata come segue. Nel caso il nuovo elemento inserito causi la creazione di un nuovo nodo, una procedura di bilanciamento balanceInsert() viene chiamata su di esso. Al termine del bilanciamento, è possibile che sia stata creata una nuova radice, che va restituita al chiamante. Il modo più semplice per identificare la nuova radice è risalire l'albero a partire dal nuovo nodo, seguendo i puntatori ai padri. Tali modifiche sono evidenziate in nero nell'algoritmo seguente, dove la parte inalterata viene mostrata in grigio.

La procedura di inserimento è simile alla insertNode() degli alberi binari di ricerca. Una volta creato e inserito il nuovo nodo nell'albero, tuttavia, è necessario assicurarsi che i vincoli Red-Black siano mantenuti, ed eventualmente operare le necessarie correzioni. La chiamata balanceInsert(n) avviene subito dopo la chiamata a link() nella procedura insertNode().

Il nodo appena inserito (da qui in poi chiamato t) viene colorato di rosso; questo garantisce che il vincolo (V_4) sia comunque rispettato, perché le altezze nere delle foglie non cambiano; ma è possibile che il vincolo (V_3) non sia più vero, in quanto il padre di t potrebbe essere rosso anch'esso; oppure è possibile che la radice sia rossa, se t è il primo nodo ad essere inserito (ovvero la radice); in questo caso viene violato il vincolo (V_1).

```
TREE insertNode(TREE T, item x, item v)
    TREE p = nil                                          % Padre
    TREE u = T
    while u ≠ nil and u.key ≠ x do                        % Cerca posizione inserimento
        p = u
        u = iif(x < u.key, u.left, u.right)
    if u ≠ nil and u.key = x then
        u.value = v                                       % Chiave già presente
    else
        TREE n = Tree(x, v)                               % Crea un nodo contenente coppia chiave-valore
        link(p, n, x)
        balanceInsert(n)
    while n.parent ≠ nil do
        n = n.parent
    return n                                              % Restituisce radice corrente dell'albero
```

La procedura balanceInsert() è organizzata in sette diversi casi, illustrati in Figura 1.5, ognuno dei quali corregge un possibile problema. I casi (1), (2), (5a), (5b) sono terminali, ovvero al completamento della loro esecuzione l'albero rispetta i vincoli Red-Black; il caso (3) risolve il problema localmente, ma il mancato rispetto dei vincoli si può ripresentare più in alto nell'albero, con uno qualunque dei 7 casi; infine, i casi (4a) e (4b) portano ai casi (5a) e (5b), rispettivamente. La procedura balanceInsert() è quindi organizzata tramite un ciclo **while**, che viene ripetuto finché $t \neq$ **nil**, ovvero fino a quando non si finisce nei casi (1), (2), (5a) e (5b).

Per comodità di esposizione, all'inizio di ogni iterazione del ciclo **while** vengono inizializzate tre variabili p, n e z, che corrispondono al padre, al "nonno" (padre del padre) e allo "zio" (fratello del padre). Per i casi (1) e (2) è possibile che siano pari a **nil**, ovvero che tali nodi non esistano. Vediamo quindi i casi:

(1) Se il nuovo nodo t non ha padre, è il primo ad essere inserito e quindi è la radice. È sufficiente colorare questo nodo di nero e terminare.

(2) Se il padre di t esiste ed è nero, non c'è da fare alcun ulteriore aggiornamento: né il vincolo (V_1) né il vincolo (V_3) sono violati e si può terminare.

Nei casi seguenti il padre è rosso. Possiamo quindi dedurre che esiste il nonno, in quanto un nodo rosso non può essere radice per il vincolo (V_1); per il vincolo (V_4), esiste anche lo zio.

(3) Se lo zio è rosso, è possibile colorare di nero il padre e lo zio, e di rosso il nonno. Poiché tutti i cammini che passano per lo zio e il padre devono passare necessariamente per il nonno, la lunghezza dei cammini neri non è cambiata. Ma il problema può essersi spostato sul nonno: può essere violato il vincolo (V_1), ovvero il nonno può essere una radice rossa; oppure il vincolo (V_3), ovvero il nonno rosso può avere un padre rosso. Poniamo $t = n$, e il ciclo continua.

Nei quattro casi seguenti, lo zio è nero. I quattro casi si distinguono per la struttura dei rapporti destro/sinistro delle coppie (p, t) e (n, p).

(4a),(4b) Si assuma che t sia figlio destro di p e che p sia figlio sinistro di n. Una rotazione a sinistra a partire dal nodo p scambia i ruoli di t e p, ottenendo una situazione simile a (5a), dove i nodi rossi in conflitto sul vincolo (V_3) sono entrambi figli sinistri dei loro padri. Dobbiamo quindi porre $t = p$ ed eseguire un'ulteriore (ultima) iterazione del ciclo. Si noti che alcuni cammini radice-foglia sono ora instradati diversamente; ma i nodi coinvolti nel cambiamento

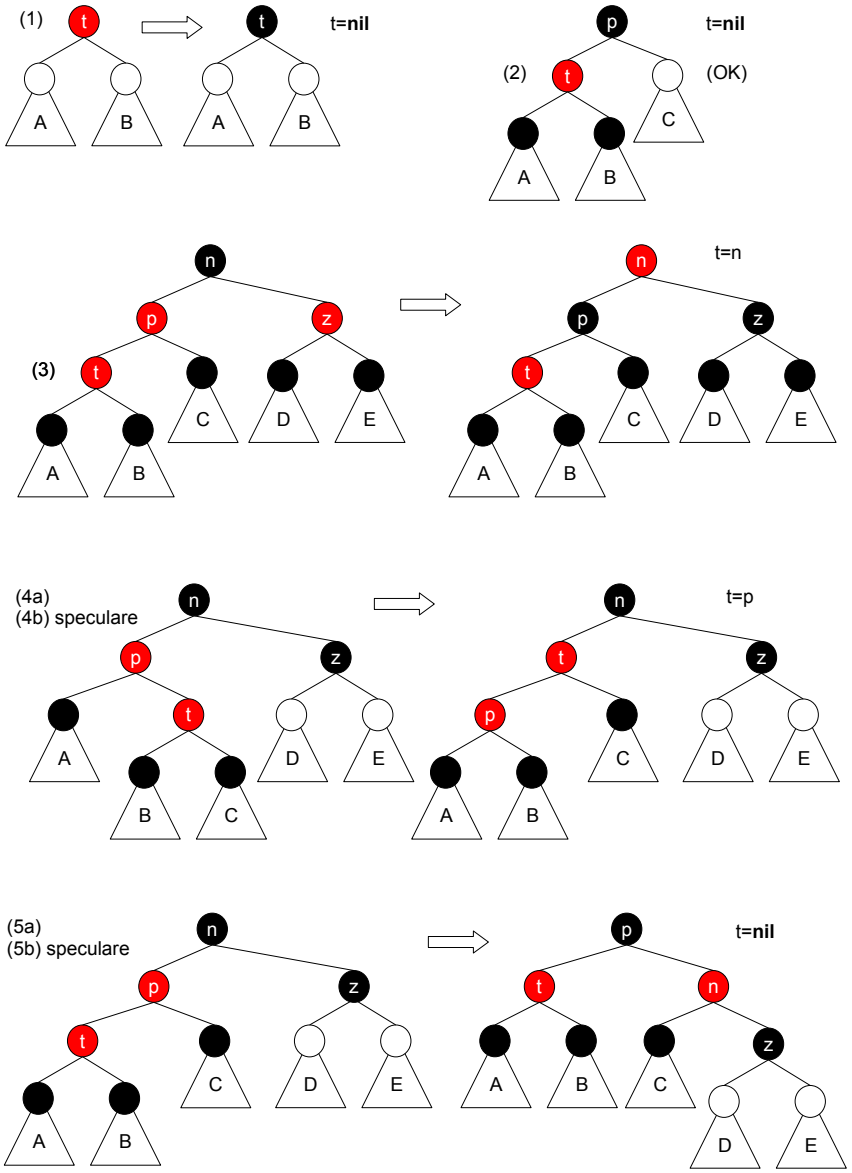

Figura 1.5: Cinque dei sette possibili casi di inserimento. I casi (4b) e (5b) non sono rappresentati, in quanto speculari ai casi (4a) e (5a). Il caso (2) non richiede alcuna modifica, mentre tutti gli altri casi collegano la situazione iniziale e la situazione finale con una freccia.

sono p e t, entrambi rossi, quindi la lunghezza dei cammini neri non cambia. Il caso (4b) è speculare a (4a), con t figlio sinistro di p e p figlio destro di n; in questo caso è necessaria una rotazione a destra.

(5a),(5b) Si assuma infine che t sia figlio sinistro di p e p sia figlio sinistro di n. Una rotazione a destra a partire da n ci porta ad una situazione in cui t e n sono figli di p; colorando n di rosso e p di nero ci troviamo in una situazione in cui tutti i vincoli Red-Black sono rispettati; in particolare, la lunghezza di tutti i cammini neri che passano per la radice è uguale alla situazione iniziale. Il caso (5b) è speculare a (5a), con t figlio destro di p e p figlio destro di n; in questo caso è necessaria una rotazione a sinistra.

```
balanceInsert(TREE t)
t.color = RED
while t ≠ nil do
    TREE p = t.parent                                              % Padre
    TREE n = iif(p ≠ nil, p.parent, nil)                           % Nonno
    TREE z = iif(n = nil, nil, iif(n.left = p, n.right, n.left))   % Zio
    if p = nil then                                                % Caso (1)
        t.color = BLACK
        t = nil
    else if p.color = BLACK then                                   % Caso (2)
        t = nil
    else if z.color = RED then                                     % Caso (3)
        p.color = z.color = BLACK
        n.color = RED
        t = n
    else
        if (t = p.right) and (p = n.left) then                     % Caso (4.a)
            rotateLeft(p)
            t = p
        else if (t = p.left) and (p = n.right) then                % Caso (4.b)
            rotateRight(p)
            t = p
        else
            if (t = p.left) and (p = n.left) then                  % Caso (5.a)
                rotateRight(n)
            else if (t = p.right) and (p = n.right) then           % Caso (5.b)
                rotateLeft(n)
            p.color = BLACK
            n.color = RED
            t = nil
```

La complessità della procedura balanceInsert() è $O(h)$, dove h è l'altezza dell'albero. Infatti, l'esecuzione di ognuno dei singoli casi è $O(1)$. Il caso (3) sposta il problema a livello del nonno, e quindi il suo codice può essere ripetuto al massimo $h/2$ volte. Tutti gli altri casi possono essere eseguiti al massimo una volta. Poiché h è $O(\log n)$, l'operazione di inserimento ha costo logaritmico nel numero di nodi.

Per quanto riguarda la cancellazione, ricordiamo che la procedura removeNode() è suddivisa in tre casi. Quando si cancella un nodo u con due figli (nel caso degli alberi Red-Black, con due figli non-foglie), si cerca l'elemento minimo s del sottoalbero destro di u (il successore) e si copia

la coppia chiave-valore da s in u. Il colore del nodo u può rimanere inalterato, e quindi i vincoli Red-Black vengono rispettati, ma il problema si sposta nel nodo s, che deve essere rimosso e può avere un figlio destro. Si ricade quindi negli altri due casi.

Da questo punto in avanti, possiamo quindi assumere che stiamo cancellando un nodo u con al massimo un figlio t non foglia. La chiamata balanceDelete(T,t), dove T è la radice dell'albero, avviene subito dopo la chiamata a link() nella procedura removeNode(). Al termine della chiamata, poichè la radice può essere cambiata, è necessario identificare la nuova radice, così come avevamo fatto con la balanceInsert() e restituirla al chiamante.

```
TREE removeNode(TREE T, item x)
    TREE u = lookupNode(T,x)
    if u ≠ nil then
        if u.left ≠ nil and u.right ≠ nil then                    % Caso 3
            TREE s = u.right
            while s.left ≠ nil do s = s.left
            u.key = s.key
            u.value = s.value
            x = s.key
            u = s
        TREE t
        if u.left ≠ nil and u.right = nil then                    % Caso (2) - Solo figlio sx
            t = u.left
        else                                                      % Caso (2) - Solo figlio dx / Caso (1)
            t = u.right
        link(u.parent,t,x)
        if u.color = BLACK then
            balanceDelete(T,t)
        if u.parent = nil then T = t
        delete u
    while T.parent ≠ nil do
        T = T.parent
    return T
```

La chiamata ha luogo se e solo se il nodo cancellato (e sostituito da t) ha colore nero. In questo caso, possiamo avere violato uno dei seguenti vincoli:

- la radice può ora essere rossa, violando il vincolo (V_1), in quanto il nodo t che ha sostituito u può avere questo colore;
- se il padre di u e t sono rossi, il fatto che ora siano uno padre dell'altro viola il vincolo (V_3);
- rimuovendo un nodo nero, abbiamo ridotto di uno l'altezza nera di tutte le foglie i cui cammini verso la radice passavano per u (e ora passano per t), violando quindi il vincolo (V_4).

Se invece il nodo u era rosso, la sua rimozione non cambia l'altezza nera delle foglie, non può creare due nodi rossi consecutivi, e non può mutare il colore della radice in rosso: in altre parole, non c'è alcun aggiornamento da fare.

La procedura balanceDelete() è organizzata in otto possibili casi; i primi quattro, illustrati in Figura 1.6, assumono che il nodo su cui opera la balanceDelete() sia figlio sinistro di suo padre; gli altri casi sono speculari, con t figlio destro di suo padre e tutti i riferimenti sinistro-destro scambiati.

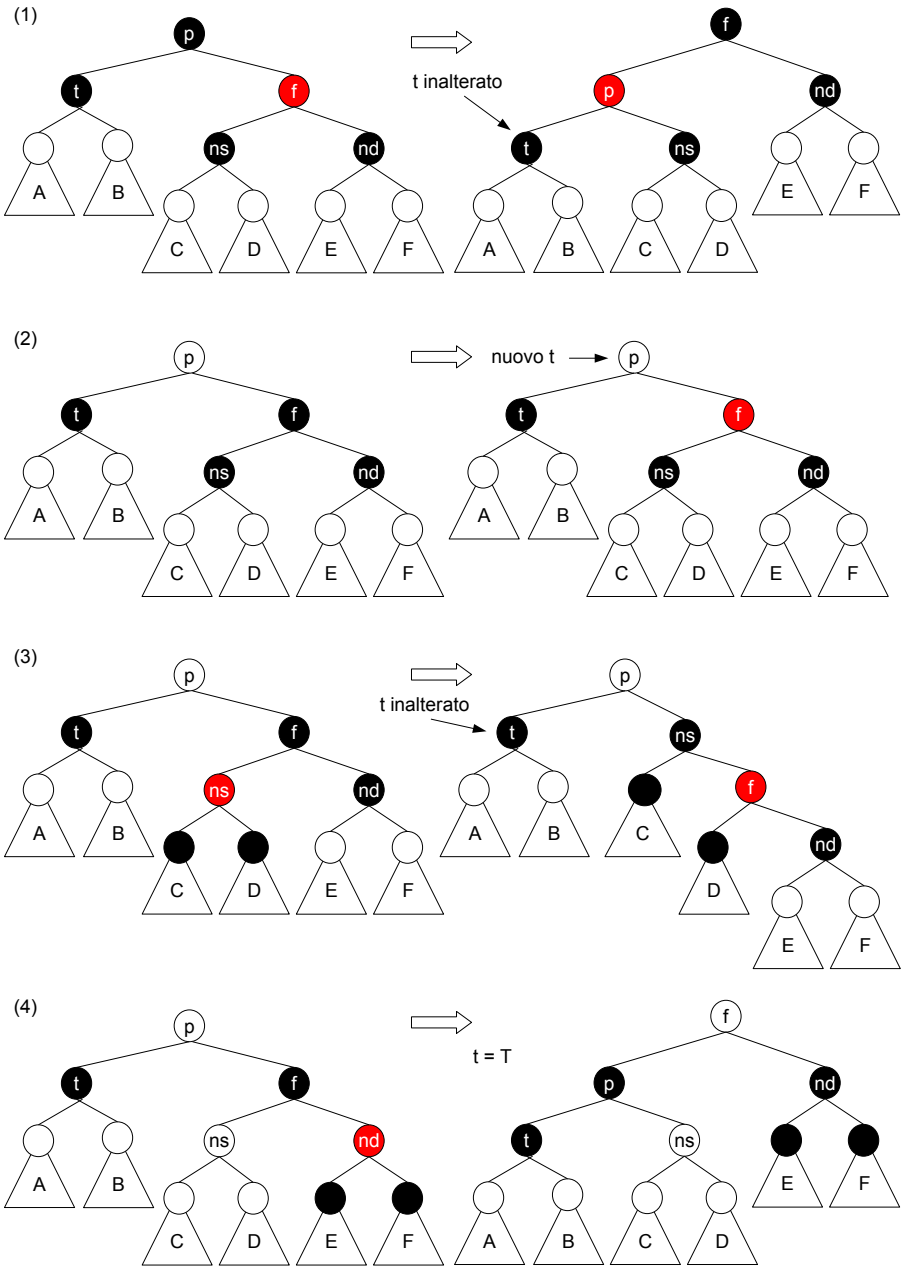

Figura 1.6: Quatto degli otto possibili casi di cancellazione, gli altri sono speculari. Tutti i casi collegano la situazione iniziale e la situazione finale con una freccia.

Come nel caso della balanceInsert(), il passaggio fra i vari casi è gestito tramite un ciclo **while**. Ad ogni iterazione del ciclo, partendo dal nodo t si ottengono i riferimenti al padre p, al fratello f e ai nipoti destro e sinistro *nd* e *ns*. Il ciclo termina quando il nodo t è la radice, oppure se è colorato di rosso. Al termine del ciclo, il nodo t viene colorato di nero. Questa operazione garantisce il rispetto del vincolo (V_1) (se il nodo colorato di nero è la radice) o del vincolo (V_3) (colorando un nodo nero, "spezziamo" il caso in cui il nodo e uno dei suoi figli siano entrambi rossi). Vedremo nella discussione dei casi come venga ripristinato il vincolo (V_4).

(1) Se il fratello f è rosso, i suoi figli *ns* e *nd* e il padre p devono essere necessariamente neri. I percorsi radice-foglia che passano per t hanno perso un nodo nero, quindi il vincolo (V_4) è stato violato. Scambiando i colori di f e p ed applicando una rotazione sinistra, lasciando invariato il puntatore t, ci ritroviamo in uno dei casi (2), (3), (4), in quanto i percorsi radice-foglia che passano per t hanno ancora un nodo nero mancante, ma ora il fratello di t non è più rosso. Gli altri percorsi radice-foglia soggetti alla rotazione ed ai cambi di colore non subiscono variazioni nella lunghezza nera.

Nei casi che seguono, il colore di f è sempre nero.

```
balanceDelete(TREE T, TREE t)
while t ≠ T and t.color = BLACK do
    TREE p = t.parent                                          % Padre
    if t = p.left() then
        TREE f = p.right                                       % Fratello
        TREE ns = f.left                                       % Nipote sinistro
        TREE nd = f.right                                      % Nipote destro
        if f.color = RED then                                  % (1)
            p.color = RED
            f.color = BLACK
            rotateLeft(p)
            % t viene lasciato inalterato, quindi si ricade nei casi (2),(3),(4)
        else
            if ns.color = nd.color = BLACK then                % (2)
                f.color = RED
                t = p
            else if ns.color = RED and nd.color = BLACK then   % (3)
                ns.color = BLACK
                f.color = RED
                rotateRight(f)
                % t viene lasciato inalterato, quindi si ricade nel caso (4)
            else if nd.color = RED then                        % (4)
                f.color = p.color
                p.color = BLACK
                nd.color = BLACK
                rotateLeft(p)
                t = T
    else
        % Casi (5)-(8) speculari a (1)-(4)
if t ≠ nil then t.color = BLACK
```

(2) *ns* e *nd* sono entrambi neri, mentre p può assumere un qualunque colore. Come prima, i percorsi radice-foglia che passano per t hanno perso un nodo nero, quindi il vincolo (V_4) è stato violato; colorando di rosso il nodo f, anche tutti i percorsi radice-foglia che passano per esso perdono un nodo nero. Questo significa che tutti i percorsi che passano per p (padre di t e f) sono nelle stesse condizioni. Il puntatore t assume quindi il valore p, e il ciclo ricomincia. A questo punto, se il nuovo nodo t ha colore rosso, il ciclo termina e t viene colorato di nero; questo aggiunge un nodo nero a tutti i percorsi radice-foglia che passano per t, ripristinando il vincolo (V_4). Se invece t è già nero, il ciclo continua.

(3) Nel caso *ns* sia rosso e *nd* sia nero, si colora il nodo f di rosso, il nodo *ns* di nero e si applica una rotazione destra a f. Lasciando t inalterato, ci siamo spostati nel caso (4). Nel sottoalbero radicato in f non si sono violate le regole (V_3) e (V_4); il problema resta nel nodo t.

(4) Nel caso *nd* sia rosso, applichiamo una rotazione sinistra a p; coloriamo f con il colore del padre, coloriamo *nd* di nero e coloriamo p di nero. In questo modo, abbiamo aggiunto un nodo nero ai cammini radice-foglia che passano per t, senza modificare la lunghezza nera dei cammini che passavano per f. Ponendo $t = T$, terminiamo l'algoritmo e ci salvaguardiamo anche dal caso in cui f sia diventato radice e sia rosso.

Per riassumere, dal caso (1) si passa ad uno dei casi (2), (3), (4); dal caso (2) si torna ad uno qualunque dei casi, ma risalendo di un livello l'albero. Dal caso (3) si passa al caso (4), mentre nel caso (4) si termina. In altre parole, è possibile visitare al massimo un numero $O(\log n)$ di casi, ognuno dei quali è gestito in tempo $O(1)$. In altre parole, la complessità di balanceDelete() e quindi di delete() è pari a $O(\log n)$.

1.3 Reality check

Fra gli alberi bilanciati, l'approccio Red-Black è quello più diffuso: lo troviamo, ad esempio, nelle strutture di dati `TreeMap` e `TreeSet` delle Java Collections , così come nei container `std::set<Value>` e `std::map<Key,Value>` della Standard Template Library del C++.

Un approccio diverso al bilanciamento degli alberi è quello dei *B-alberi* ed in particolare della variante B^+. Il loro nome deriva forse dall'iniziale di Bayer che li ha inventati insieme a McCreight negli anni '70, oppure da Boeing, l'azienda presso cui Bayer e McCreight lavoravano, o piuttosto dall'iniziale del termine "balanced".

A differenza degli alberi Red-Black, i B-alberi sono ottimizzati per sistemi di memorizzazione orientata alle pagine, quali database e file system. In questi sistemi, i dati sono memorizzati in memoria secondaria, divisi in pagine di grandi dimensioni (e.g., 4KB). Il tempo di accesso a tali pagine è misurato in millisecondi e non in nanosecondi, come avviene in memoria principale; inoltre, ogni lettura comporta l'acquisizione di un'intera pagina, anche se è necessario accedere ad un singolo dato. Una struttura di dati ottimizzata per questi sistemi deve quindi minimizzare il numero di letture di pagine diverse, cercando di sfruttare il più possibile tutti i dati contenuti in una pagina. Nella versione B^+, i B^+-alberi permettono non solo di eseguire ricerche, inserimenti e cancellazioni in tempo logaritmico, ma anche di accedere sequenzialmente ai dati in modo facile ed efficiente.

I B^+-alberi sono implementati in un discreto numero di file system, fra i quali i più famosi sono ReiserFS, XFS, JFS e NTFS, e nei database relazionali quali IBM DB2, Oracle 8, e Microsoft SQL Server. Vediamo rapidamente i principi del loro funzionamento, senza entrare nei loro complessi dettagli implementativi.

In un B^+-albero, ogni nodo è memorizzato in una pagina di memoria secondaria. Ogni nodo contiene una sequenza di chiavi e puntatori, assieme ad un contatore che indica il numero di chiavi presenti e un flag booleano che dice se il nodo è interno oppure una foglia. I puntatori dei nodi interni puntano ad altri nodi memorizzati in memoria secondaria; nei nodi foglia, invece, puntano

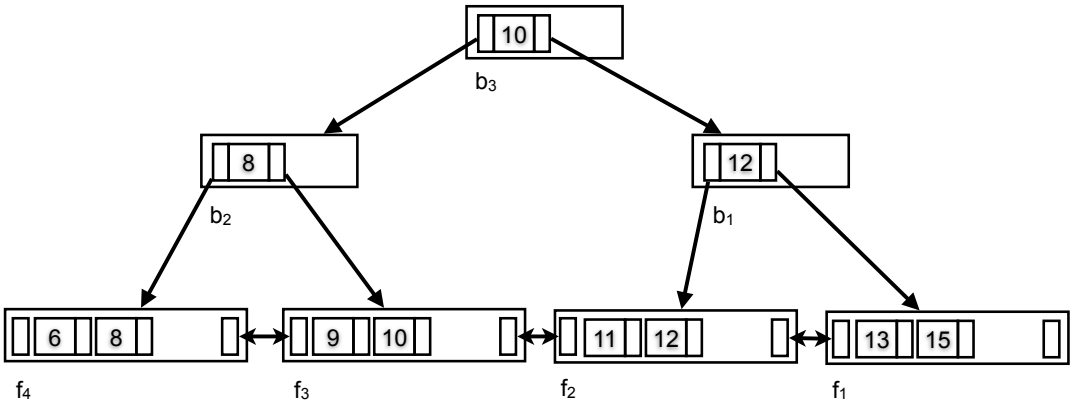

Figura 1.7: Un B$^+$-albero di ordine $m = 3$ contenente 3 nodi interni e 5 foglie con 8 chiavi.

ai dati associati alla chiave, come ad esempio un file oppure una riga di database. In altre parole, i nodi interni non contengono valori, ma permettono solo l'esplorazione dell'albero verso le foglie, dove i dati sono memorizzati. Inoltre, i B$^+$-alberi si contraddistinguono dal fatto che le foglie si trovano tutte allo stesso livello e sono collegate in una lista doppia, per permettere interrogazioni su intervalli di dati in maniera efficiente.

Ogni B$^+$-albero è caratterizzato da un parametro $m > 2$, detto anche *ordine* del B$^+$-albero, che rappresenta il numero massimo di figli che un nodo interno può avere; questo valore può essere molto elevato, ben superiore al valore 2 che si osserva negli alberi binari. Ogni nodo interno deve essere pieno almeno per metà; ovvero, detto f il numero di figli di un nodo, $\lceil m/2 \rceil \leq f \leq m$. Fa eccezione la radice, che può avere anche due figli (o essere una foglia).

La struttura di un B-albero è illustrata in Figura 1.7. I nodi interni contengono fino ad $m-1$ chiavi $k[1,\ldots m-1]$ e fino ad m puntatori $p[1,\ldots,m]$. Ogni puntatore $p[i]$, con $1 < i < m$, punta al sottoalbero contenente le chiavi comprese fra $k[i-1]$ (estremo escluso) e $k[i]$ (estremo incluso). Il puntatore $p[1]$ punta al sottoalbero contenente le chiavi minori o uguali a $k[1]$, mentre il puntatore $p[m]$ contiene le chiavi maggiori di $k[m-1]$. Nei nodi foglia, invece, sono presenti fino ad m coppie chiavi-puntatori, dove il puntatore punta al dato indicizzato, come un file in un file system o una riga in un database.

In base a queste considerazioni, l'algoritmo per la ricerca di una coppia chiave-valore parte dalla radice e seleziona il primo puntatore la cui corrispondente chiave è maggiore o uguale alla chiave cercata, oppure $p[m]$ se nessuna chiave memorizzata nel nodo soddisfa questo requisito. Si sposta quindi nel sottolbero indicato dal puntatore e continua così ricorsivamente nei sottoalberi analizzati. Una volta raggiunta una foglia, la chiave viene cercata fra quelle memorizzate all'interno ed eventualmente viene restituito il valore associato.

Esempio (Ricerca in un B$^+$-albero) Volendo cercare la chiave 12 nell'albero di Figura 1.7, si caricherà innanzitutto il nodo radice b_3 dalla memoria secondaria in memoria principale. Poichè 12 è maggiore di 10, si utilizzera il secondo puntatore di b_3 e si caricherà in memoria principale la pagina b_1. Poichè 12 è minore o uguale a 12, si utilizzerà il primo puntatore e si caricherà in memoria la pagina f_2. Poichè f_2 è una foglia e contiene 12, si utilizzerà il puntatore associato a 12 per raggiungere il valore cercato. ■

La procedura di inserimento in un B$^+$-albero è logicamente semplice, ma complessa da programmare. Ne vediamo quindi solo a grandi linee il funzionamento. Per comodità di presentazione,

assumiamo che m sia dispari, ovvero sia uguale ad un valore $2d+1$, e assumiamo che non siano ammesse chiavi ripetute.

Innanzitutto, si sfrutta il meccanismo di ricerca per individuare la foglia f_i in cui la chiave dovrebbe essere collocata. Se f_i non è ancora piena (il numero di elementi è inferiore a m), la chiave e il relativo puntatore vengono aggiunti ad essa. Per costruzione della procedura di ricerca, le chiavi e i puntatori del nodo padre della foglia non richiedono di essere modificati e la procedura di inserimento termina.

Altrimenti, se f_i è piena (contiene già $m = 2d+1$ chiavi), si crea una nuova foglia f_j e si dividono equamente le $2d+2$ chiavi (quelle contenute in f_i e quella da inserire) fra f_j (le prime $d+1$) e f_i (le seconde $d+1$). Il valore $d+1$ è pari a $\lceil m/2 \rceil$, il numero minimo di figli che devono essere contenuti in un nodo.

A questo punto, la chiave massima contenuta in f_j e il puntatore a f_j devono essere inseriti nel nodo interno b_k padre di f_i. Se b_k ha meno di $m = 2d+1$ figli, la coppia chiave-puntatore viene semplicemente inserita (in ordine) nella pagina e la procedura di inserimento termina.

Altrimenti, se b_k ha $m = 2d+1$ figli, si genera un ulteriore pagina b_l e i $2d+2$ puntatori vengono divisi fra b_l e b_k, usando lo stesso principio utilizzato per suddividere le foglie. Questo genera un nuovo inserimento al livello superiore, che può arrivare fino alla radice. Nel caso venga suddivisa la radice, si genera una nuova radice che inizialmente ha due figli.

Come ultimo caso particolare, si consideri il B$^+$-albero quando non è presente alcuna chiave. I primi m inserimenti vengono effettuati direttamente nella radice, che inizialmente è una foglia. Non appena questa foglia viene divisa in due foglie, viene generato un nodo interno che agisce come radice e ha le due foglie come figli.

Esempio (Ricerca in un B$^+$-albero) Vediamo una sequenza di inserimento di chiavi che genera il B$^+$-albero di Figura 1.7. Si assuma che il parametro m sia uguale a 3, un valore basso ma necessario per motivi grafici. Si osservi la Figura 1.8. Inizialmente (parte (a)), vengono inserite le chiavi 10,12,13, che vengono inserite nella foglia che costituisce la radice. Nella parte (b), viene inserita la chiave 15, che forza la creazione di f_2 e la suddivisione delle chiavi fra f_1 e f_2, oltre che la generazione del primo nodo interno, la radice b_1. L'inserimento della chiave 8 nella parte (c) non crea alcuna suddivisione, ma riempe la foglia f_2. Nella parte (d), l'inserimento della chiave 11 comporta la creazione di f_3 e la relativa suddivisione di chiavi; la chiave più alta inserita in f_3 viene inserita in b_1, che quindi ora è completo. Nella parte (e), la chiave 9 viene inserita in f_3, senza modifiche ai nodi interni. Infine, l'inserimento della chiave 6 forza la suddivisione di f_3 in f_3 ed f_4; l'inserimento della chiave 8 in b_1 comporta la suddivisione anche di questa pagina, con la conseguente creazione di una nuova radice, mostrata in Figura 1.7. ■

La procedura di cancellazione comporta una serie di casi simili, che possono portare all'eliminazione della radice e alla riduzione dell'altezza dell'albero.

A questo punto, è necessario fare alcune considerazioni sull'efficienza di questo approccio. Il valore $m = 3$ mostrato qui non è realistico; se si considerano pagine di 4KB, e coppie chiave-puntatore di 16 byte (due interi a 64 bit), è possibile ottenere un valore $m = 256$. Il fatto che le pagine debbano avere almeno $\lceil m/2 \rceil$ figli, indica che nel caso pessimo tutti i nodi sono riempiti con almeno 128 puntatori. Questo significa che l'albero risultante avrà altezza pari a $O(\log_{128} n)$, dove n è il numero di chiavi inserite. Sebbene questo valore sia comunque logaritmico, la base molto alta fa sì che il numero di nodi da caricare in memoria principale sia molto basso, ottimizzando così il tempo di accesso alla memoria secondaria. Dal punto di vista dell'occupazione di memoria, poiché i nodi possono essere mezzi pieni, questo comporta un utilizzo al più doppio della memoria

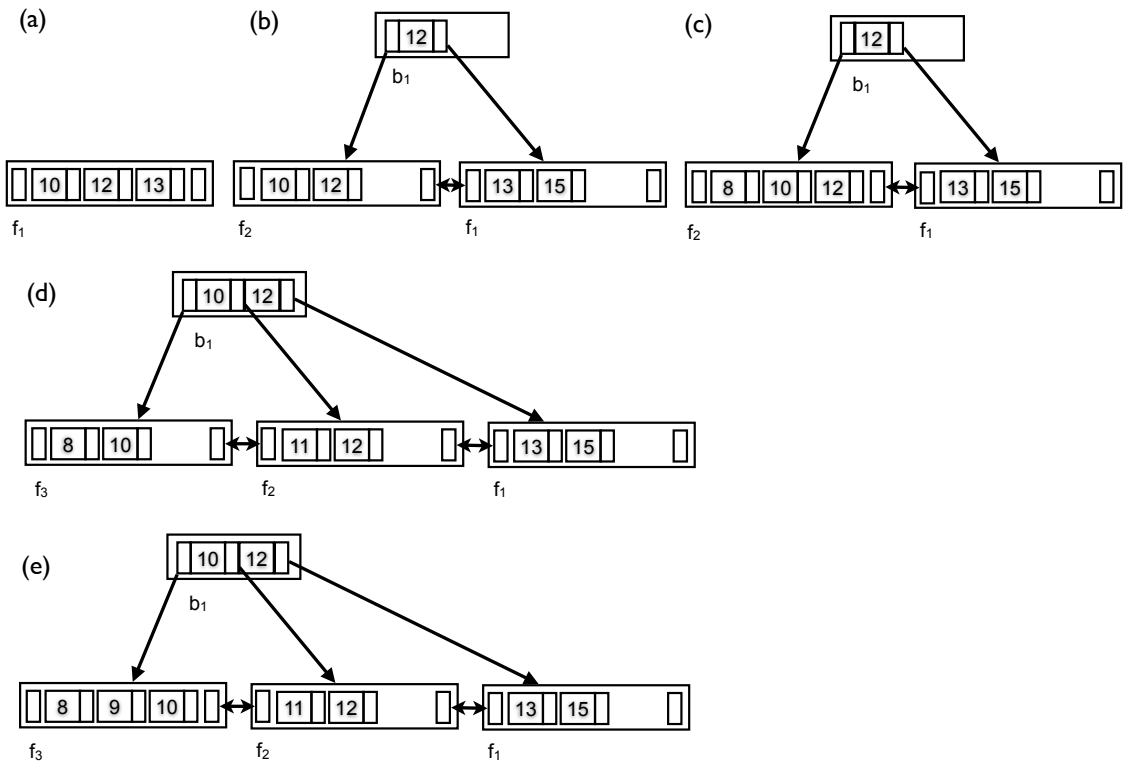

Figura 1.8: Inserimento di chiavi in un B^+-albero di ordine $m = 3$. (a) Vengono inserite le chiave 10, 12, 13; (b) viene inserita la chiave 15; (c) viene inserita la chiave 8; (d) viene inserita la chiave 11; (e) viene inserita la chiave 9. Infine, con l'inserimento della chiave 6, si genera il B^+-albero di Figura 1.7.

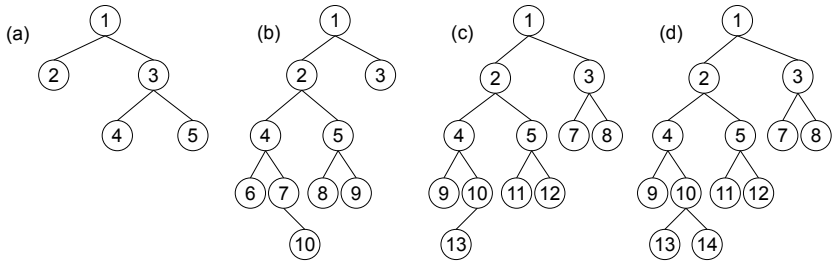

Figura 1.9: Alberi da colorare

necessaria per memorizzare tale albero. Questo "spreco" è giustificato dalla maggiore efficienza della struttura.

1.4 Esercizi

Esercizio 1.1. Dimostrare che, poiché l'ordinamento di n elementi basato su confronti richiede tempo $\Omega(n \log n)$, allora qualunque algoritmo basato su confronti per costruire un ABR contenente n valori è $\Omega(n \log n)$.

Esercizio 1.2. Si scriva una versione ricorsiva della procedura insertNode().

Esercizio 1.3. Si scriva una funzione che verifichi se un albero binario è un albero di ricerca usando gli operatori degli alberi binari.

Esercizio 1.4 (Stampa intervallo)**.** Dato un albero binario di ricerca i cui nodi contengono interi positivi, si scriva una procedura efficiente che usi gli operatori degli alberi binari e stampi, in ordine crescente, tutte le chiavi che appartengono ad un dato intervallo $[A,B]$.

Esercizio 1.5 (Invisita iterativa)**.** Dato un albero binario di ricerca realizzato con puntatori e i cui nodi contengono interi, si scriva una versione iterativa della visita simmetrica senza usare una pila.

Esercizio 1.6 (Concatenazione)**.** Dati due alberi binari di ricerca T_1 e T_2 tali che le chiavi in T_1 sono tutte minori delle chiavi in T_2, scrivere una procedura che restituisce un albero di ricerca contenente tutte le chiavi in tempo $O(h)$, dove h è l'altezza massima dei due alberi.

Esercizio 1.7 (Realizzazione di insiemi con ABR)**.** Si assuma di rappresentare gli insiemi con ABR. Quali sono le complessità delle operazioni union(), difference() e intersection()?

Esercizio 1.8 (Realizzazione di insiemi con alberi Red-Black)**.** Si assuma di rappresentare gli insiemi dell'esercizio precedente con alberi Red-Black. Vi sono dei cambiamenti da apportare alla vostra soluzione? La complessità cambia?

Esercizio 1.9 (Colorazione Red-Black)**.** Quali degli alberi in Figura 1.9 possono essere colorati rispettando i vincoli degli alberi Red-Black, e quali no?

Esercizio 1.10 (Dimensioni degli alberi Red-Black)**.** Qual è il numero minimo di nodi non foglie di un albero Red-Black di altezza nera b? Qual è il numero massimo?

1.5 Soluzioni

Soluzione Esercizio 1.1

Supponiamo per assurdo che esista un algoritmo basato su confronti che permetta di costruire l'albero in tempo $\Theta(g(n))$, e che $g(n)$ non sia $\Omega(n \log n)$. Allora sarebbe possibile ordinare n numeri costruendo l'albero, e poi visitando lo stesso in ordine simmetrico (in tempo $\Theta(n)$). Questo algoritmo di ordinamento avrebbe costo $\Theta(g(n)+n)$ che non è $\Omega(n \log n)$, il che è impossibile vista la dimostrazione della limitazione inferiore per l'ordinamento.

Soluzione Esercizio 1.4

L'algoritmo si basa su una visita simmetrica in cui vengono scartati i sottoalberi con chiavi non comprese nell'intervallo $[A,B]$. La sua complessità è $O(h+k)$, dove h è l'altezza dell'albero e k è il numero di chiavi che appartengono ad $[A,B]$. Infatti, occorre tempo $O(h)$ per individuare la chiave più piccola e quella più grande in $[A,B]$ e tempo $O(k)$ per visitare tutte le chiavi in $[A,B]$. La procedura assume che l'albero non sia vuoto.

```
printNodes(TREE t, int A, int B)
    int v = t.read()
    if t.left() ≠ nil and A < v then printNodes(t.left(),A,B)
    if A ≤ v and v ≤ B then print v
    if t.right() ≠ nil and B > v then printNodes(t.right(),A,B)
```

Soluzione Esercizio 1.5

Si usano le funzioni min() e successorNode(), poiché una visita simmetrica di un ABR T visita le chiavi in ordine crescente. Una versione iterativa si ottiene immediatamente visitando per primo il nodo restituito da min() e poi richiamando iterativamente la successorNode() finché non si giunge al nodo che contiene la chiave massima (e che ha **nil** come successore).

```
in-visit(TREE t)
    TREE u = t.min()
    while u ≠ nil do
        {esamina il nodo u}
        u = u.successorNode()
```

Nonostante in-visit() chiami per ogni nodo una procedura di complessità $O(h)$, la complessità della visita è $\Theta(n)$ e non $O(nh)$. Infatti la funzione in-visit() attraversa il minimo numero di nodi necessari a raggiungere il successore di u. Pertanto, la sequenza di n chiamate nella in-visit() attraversa ciascuna coppia padre-figlio esattamente una volta scendendo dal padre al figlio e ciascuna coppia figlio-padre esattamente una seconda volta risalendo dal figlio al padre.

Soluzione Esercizio 1.7

Siano A e B, con $|A| = n_A$ e $|B| = n_B$, gli insiemi in ingresso e C l'insieme da fornire come risultato, e denotiamo con T_X un albero bilanciato di ricerca che memorizza un generico insieme X. Per realizzare union() e difference(), occorre innanzitutto costruirsi T_C, facendosi in tempo $O(n_A)$ una copia di T_A. Per l'union(), si visita T_B e si aggiunge a T_C ogni chiave k di T_B che non appartiene a T_A. Dato che la visita richiede tempo $O(n_B)$, mentre A.contains(k) e C.insert(k) ne consumano $O(\log n_A)$ e $O(\log(n_A+n_B))$, rispettivamente, la complessità della union() è $O(n_A + n_B \log(n_A + n_B))$. Un tempo $O(n_A + n_B \log n_A)$ è invece richiesto dalla difference(), dato che occorre cancellare da T_C ogni chiave k di T_B che appartiene a T_A e C.remove(k) consuma in tal caso tempo O($\log n_A$). Per intersection() invece non è necessario ricopiarsi T_A in T_C, ma si parte con T_C inizialmente vuoto, e vi si aggiunge ogni chiave k di T_B che appartiene a T_A, per un totale di $O(n_B \log n_A)$ tempo. Si noti che in quest'ultimo caso si possono eventualmente scambiare i ruoli di T_A e T_B in modo

da effettuare la visita sull'albero col minor numero di chiavi. Si può quindi risparmiare tempo, rispetto alla realizzazione con liste ordinate, nel caso particolare che la cardinalità di un insieme sia molto minore della cardinalità dell'altro. Analoghe considerazioni valgono anche per la union(), ove conviene che $n_A > n_B$. Infine, è interessante il caso in cui si voglia modificare direttamente uno dei due insiemi in input, per esempio A, perché in questa eventualità, essendo $C = A$, non occorre ricopiarsi T_A in T_C, e le complessità scendono a $O(n_B \log n_A)$ per difference() e intersection() e a $O(n_B \log(n_A + n_B))$ per union(); quindi si potrebbe di nuovo risparmiare tempo rispetto alla realizzazione con liste ordinate nel caso particolare che le cardinalità dei due insiemi sono molto diverse tra loro.

Soluzione Esercizio 1.10

L'albero Red-Black di altezza nera b con il minimo numero di nodi è l'albero in cui in tutti i cammini radice-foglia non si incontrano nodi rossi, e tutte le foglie hanno lo stesso livello $h+1$. L'altezza è quindi $h = b$. Il numero totale di nodi interni è allora $2^b - 1$

L'albero Red-Black di altezza nera b con il massimo numero di nodi è l'albero perfetto che alterna livelli rossi a livelli neri. Se l'altezza nera è b, l'altezza dell'albero è $h = 2b$. Il numero totale di nodi interni è quindi $2^{2b} - 1$.