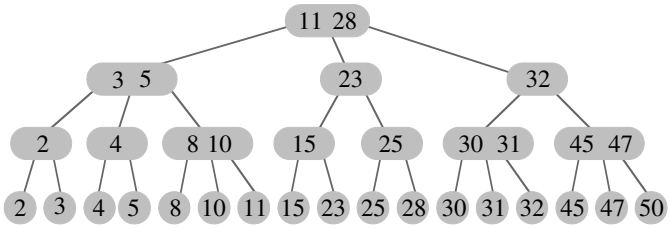

Figura 6.20 Esempio di albero 2-3. I nodi interni con due figli mantengono il massimo del sottoalbero sinistro (S); i nodi interni con tre figli mantengono il massimo del sottoalbero sinistro e di quello centrale (S ed M).

6.4 Alberi 2-3

Nei Paragrafi 6.2 e 6.3 abbiamo visto come usare rotazioni per implementare le operazioni su dizionari in tempo logaritmico (nel caso peggiore o ammortizzato). In questo paragrafo studieremo una tecnica alternativa alle rotazioni, basata sull'idea di permettere una maggiore flessibilità nel grado dei nodi. Se il grado non è vincolato ad essere 2, il bilanciamento può infatti essere mantenuto tramite opportune separazioni (`split`) e fusioni (`fuse`) di nodi.

Definizione 6.6 (Albero 2-3) *Un albero* 2-3 *è un albero che in cui ogni nodo interno ha* 2 *o* 3 *figli e tutti i cammini radice-foglia hanno la stessa lunghezza.*

Dimostriamo innanzitutto una limitazione sull'altezza degli alberi 2-3.

Lemma 6.5 *Sia T un albero* 2-3 *con n nodi, f foglie ed altezza h. Le seguenti disuaglianze sono soddisfatte da n, f e h: $2^{h+1} - 1 \leq n \leq (3^{h+1} - 1)/2$ e $2^h \leq f \leq 3^h$.*

Dimostrazione. Mostriamo la limitazioni contemporaneamente usando induzione su h. Se $h = 0$, l'albero consiste di un singolo nodo, che è anche foglia, e le disuguaglianze sono banalmente verificate. Supponiamo ora che l'ipotesi induttiva sia verificata sino ad altezza h e consideriamo un albero 2-3 T di altezza $h+1$. Sia T' ottenuto da T eliminando l'ultimo livello e siano n' e f' il numero di nodi e di foglie di T', rispettivamente. Per ipotesi induttiva, $2^{h+1} - 1 \leq n' \leq (3^{h+1} - 1)/2$ e $2^h \leq f' \leq 3^h$. Poiché ogni foglia di T' ha almeno due ad al più tre figli in T, avremo $2 \cdot 2^h \leq f \leq 3 \cdot 3^h$, ovvero $2^{h+1} \leq f \leq 3^{h+1}$. Poiché $n = n' + f$, è facile completare la dimostrazione sfruttando l'ipotesi induttiva su n'. □

Usando la limitazione sul numero di nodi dimostrata nel Lemma 6.5, e passando al logaritmo, otteniamo che l'altezza di un albero 2-3 è $\Theta(\log n)$.

Le chiavi e gli elementi del dizionario sono assegnati alle foglie dell'albero, in modo che le chiavi appaiano in ordine crescente da sinistra verso destra. Ogni nodo interno v mantiene invece due informazioni supplementari: $S[v]$ e $M[v]$. $S[v]$ è la massima chiave nel sottoalbero radicato nel figlio sinistro di v, e $M[v]$ è

```
    algoritmo search(radice r di un albero 2-3, chiave x) → elem
1.      if ( r è una foglia ) then
2.          if ( x = chiave(r) ) then return elem(r)
3.          else return null
4.      v_i ← i-esimo figlio di r
5.      if ( x ≤ S[r] ) then return search(v_1, x)
6.      else if ( r ha due figli oppure x ≤ M[r] ) then return search(v_2, x)
7.           else return search(v_3, x)
```

Figura 6.21 Implementazione dell'operazione `search` in un albero 2-3.

la massima chiave nel sottoalbero radicato nel figlio centrale di v. Un esempio è mostrato in Figura 6.20. Grazie alle informazioni S ed M, una ricerca può essere implementata in una maniera simile agli alberi binari di ricerca classici.

`search`$(chiave\ k) \rightarrow elem$

Confrontiamo la chiave cercata k sia con $S[v]$ che con $M[v]$. Se $k \leq S[v]$ proseguiamo nel sottoalbero sinistro; se $S[v] < k \leq M[v]$ proseguiamo nel sottoalbero centrale; altrimenti proseguiamo nel sottoalbero destro. Lo pseudocodice è mostrato in Figura 6.21. Il tempo richiesto da `search` è proporzionale all'altezza dell'albero, che è $\Theta(\log n)$ per il Lemma 6.5.

6.4.1 Fusioni e separazioni di nodi

In questo paragrafo mostreremo come sfruttare le possibili variazioni di grado per aggiornare un albero 2-3 a fronte di inserimenti e cancellazioni di elementi.

`insert`$(elem\ e, chiave\ k)$

Creiamo un nuovo nodo u con elemento e e chiave k. Localizziamo la corretta posizione per l'inserimento di u ricercando la chiave k nell'albero. Identifichiamo così un nodo v, sul penultimo livello, che dovrebbe diventare genitore di u. Abbiamo ora due casi:

- v ha due figli: possiamo aggiungere u come nuovo figlio di v, inserendolo opportunamente come figlio sinistro, centrale, o destro in modo da mantenere l'ordinamento crescente delle chiavi. Questo può comportare dover aggiornare i valori S ed M del nodo v e dei suoi antenati.
- v ha tre figli: non potendo aggiungere un ulteriore figlio a v, separiamo il nodo in due, con un'operazione chiamata `split`. Creiamo un nuovo nodo w e calcoliamo la posizione corretta che u dovrebbe avere rispetto ai tre figli di v. Rendiamo le due foglie con chiavi minime figlie di w e le rimanenti due figlie di v. Attacchiamo poi w come figlio del padre di v immediatamente precedente a v. Se il padre di w aveva due figli, possiamo fermarci. Altrimenti, dobbiamo

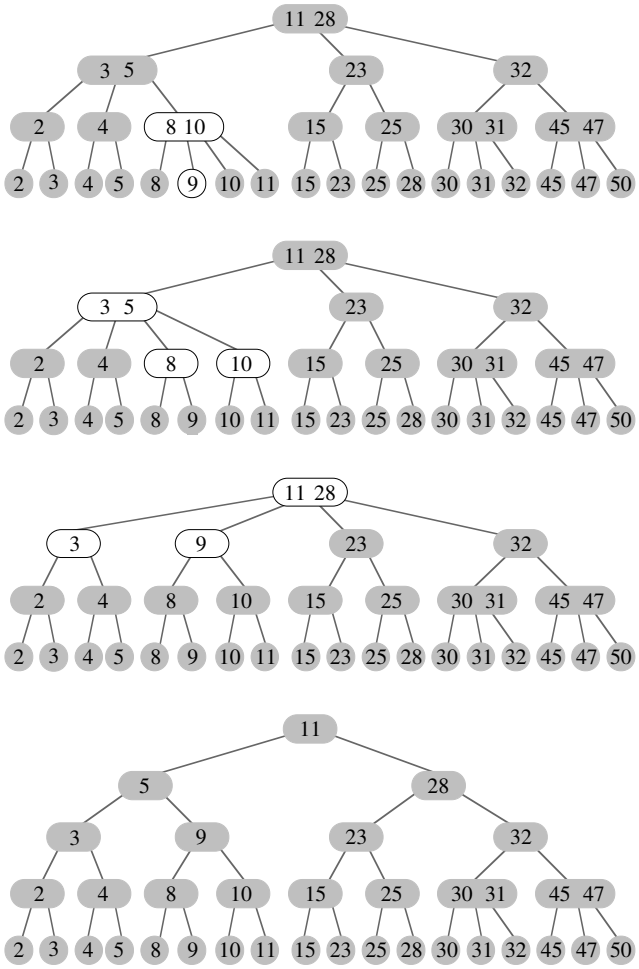

Figura 6.22 Inserimento del nodo con chiave 9 in un albero 2-3.

eseguire un nuovo `split` sul padre di v, procedendo nell'albero verso l'alto. Nel caso peggiore, quando tutti i nodi lungo il cammino avevano già tre figli, aggiungeremo all'albero una nuova radice. I valori S e M dei nodi incontrati lungo la risalita vanno anch'essi aggiornati opportunamente.

Un esempio di inserimento è illustrato in Figura 6.22. Lo pseudocodice della procedura `split`, richiamata su un nodo v con 4 figli, è inoltre mostrato in Figura 6.23 (lo pseudocodice non mostra come aggiornare i campi S ed M di ciascun nodo: aggiungere le opportune istruzioni può essere un utile esercizio). Poiché ciascuno split richiede tempo costante e l'altezza dell'albero è $\Theta(\log n)$, nel caso peggiore l'inserimento richiede tempo $O(\log n)$.

```
    algoritmo split(nodo v)
1.      crea un nuovo nodo w
2.      sia v_i l'i-esimo figlio di v in T, 1 ≤ i ≤ 4
3.      rendi v_1 e v_2 figli sinistro e destro di w
4.      if ( parent[v] = null ) then
5.          crea un nuovo nodo r
6.          rendi w e v figli sinistro e destro di r
7.      else
8.          aggiungi w come figlio di parent[v] immediatamente precedente a v
9.          if ( parent[v] ha quattro figli ) then split(parent[v])
```

Figura 6.23 Implementazione della procedura ausiliaria `split` usata per implementare l'inserimento in un albero 2-3.

`delete`$(elem\ e)$

L'operazione di cancellazione è simmetrica a quella di inserimento. Sia v il nodo contenente l'elemento e da eliminare. Abbiamo tre casi:

- v è la radice: basta rimuoverla ottenendo un albero vuoto.
- Il padre di v ha tre figli: è possibile rimuovere v, aggiornando eventualmente i campi S e M di v e dei suoi antenati (vedi Figura 6.24a).
- Il padre di v ha due figli: se il padre di v è la radice, basta eliminare v e suo padre, lasciando l'altro figlio come radice. Altrimenti, eseguiamo una operazione simmetrica allo `split`, che chiameremo `fuse`. Sia w il padre di v; assumiamo che w abbia un fratello l alla sua sinistra (nessun nodo può infatti essere figlio unico, e il caso in cui w abbia un unico fratello a destra viene trattato in modo

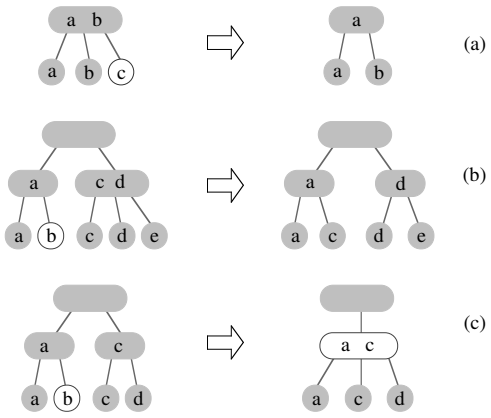

Figura 6.24 Cancellazione da un albero 2-3: vari casi.

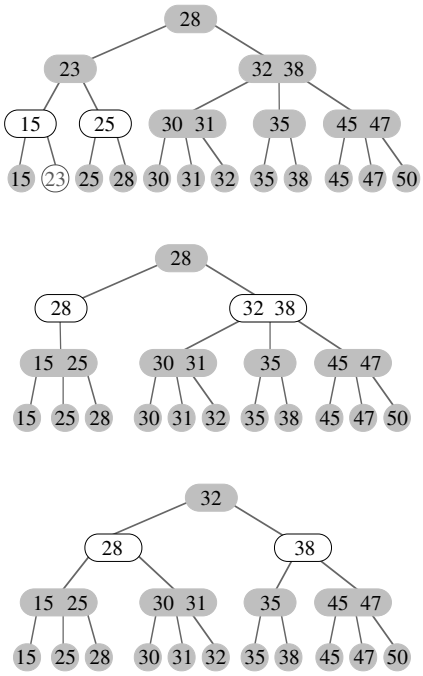

Figura 6.25 Cancellazione del nodo con chiave 23 da un albero 2-3.

simile). Se l ha tre figli, spostiamo il figlio destro di l come figlio sinistro di w e cancelliamo v (vedi Figura 6.24b). Altrimenti, dopo aver rimosso v, attacchiamo l'unico figlio rimanente di w come figlio destro di l e richiamiamo ricorsivamente la procedura per cancellare w (vedi Figura 6.24c).

Un esempio di cancellazione è mostrato in Figura 6.25. Anche in questo caso, poiché ciascuna `fuse` richiede tempo costante e l'altezza dell'albero è $\Theta(\log n)$, nel caso peggiore la cancellazione richiede tempo $O(\log n)$.

Riassumiamo i dettagli sull'implementazione della classe `Albero23` in Figura 6.26 e le prestazioni degli alberi 2-3 nel seguente teorema.

Teorema 6.5 *Un albero* 2-3 *con* n *nodi supporta operazioni* `search`*,* `insert` *e* `delete` *in tempo* $O(\log n)$ *nel caso peggiore.*

6.5 B-alberi

In questo paragrafo affronteremo il problema di come realizzare un dizionario in memoria secondaria. Come abbiamo visto nel Paragrafo 2.8 del Capitolo 2, la

classe `Albero23` **implementa** `Dizionario`:
dati: $S(n) = O(n)$
un albero 2-3 T con n nodi: le foglie mantengono le chiavi e gli elementi del dizionario, mentre i nodi interni mantengono le informazioni supplementari S e M.

operazioni:
`search`$(chiave\ k) \rightarrow elem$ $T(n) = O(\log n)$
traccia un cammino nell'albero usando la proprietà di ricerca e le informazioni S e M per decidere se proseguire nel sottoalbero sinistro, centrale (se esiste) o destro.

`insert`$(elem\ e, chiave\ k)$ $T(n) = O(\log n)$
crea un nuovo nodo v con elemento e e chiave k, e lo aggiunge all'albero come foglia mantenendo la proprietà di ricerca. Se il padre di v aveva già tre figli, separa il nodo in due tramite uno `split` e propaga le separazioni verso l'alto, fino al primo nodo con due figli o fino alla creazione di una nuova radice.

`delete`$(elem\ e)$ $T(n) = O(\log n)$
elimina il nodo v con elemento e. Se il padre w di v aveva solo due figli, uniscilo al fratello tramite un'operazione `fuse` e propaga le fusioni verso l'alto, fino al primo nodo con tre figli o fino alla cancellazione della radice.

Figura 6.26 Dizionario realizzato mediante alberi 2-3.

memoria secondaria è tipicamente molto più grande di quella principale, ma accedervi è molto più costoso in termini di tempo. È quindi importante minimizzare il numero di letture e scritture su memoria esterna (I/O). Per ammortizzare il tempo speso in ogni accesso, i dati sono gestiti in blocchi di dimensione B: per poter sfruttare questo fatto, è importante che un algoritmo o una struttura dati esibisca località nell'accesso ai dati.

Le implementazioni dei dizionari proposte nei paragrafi precedenti non esibiscono buona località: intuitivamente, non offrono nessuna garanzia che i nodi su un certo cammino di ricerca risiedano nello stesso blocco, e quindi si potrebbero avere $\Theta(\log n)$ I/O nel caso peggiore. Come vedremo, è possibile fare molto meglio. L'idea è di aumentare l'informazione a carico di ciascun nodo ed il grado del nodo rendendoli proporzionali a B. In questo modo, potremo usare tutta l'informazione contenuta in ogni blocco che porteremo in memoria principale, e, poiché diminuiremo considerevolmente l'altezza dell'albero, avremo un numero minore di accessi a memoria secondaria. I B-alberi che presentiamo in questo paragrafo sono dovuti a Bayer e McCreight [4] e sono un'estensione naturale degli alberi 2-3 che abbiamo descritto nel Paragrafo 6.4.