UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE ED ELETTRICA
E MATEMATICA APPLICATA

Corso:
DESIGN AND ANALYSIS OF ALGORITHMS

APPUNTI

Carmine Terracciano

Mat. IE22700109

ANNO ACCADEMICO 2025/2026

Sommario

1 Binary Search Tree (BST) . . . 5
1.1 BST Insert . . . 5
1.2 BST Delete . . . 6
1.2.1 Cancellazione di un nodo con al massimo un figlio . . . 6
1.2.2 Cancellazione di un nodo con due figli . . . 6
1.3 Analisi delle prestazioni . . . 7
1.4 Richiami: Tipi di alberi binari . . . 8
2 AVL Tree (AVLT) . . . 9
2.1 Alberi bilanciati e Balance Factor . . . 9
2.2 Definizione di Albero AVL . . . 10
2.3 Mantenimento della Proprietà AVL: Ristrutturazione . . . 10
2.3.1 Caso 1 e 2: Rotazioni Singole (Casi Esterni) . . . 11
2.3.2 Caso 3 e 4: Rotazioni Doppie (Casi Interni) . . . 11
2.4 AVL Insert . . . 13
2.5 AVL Delete . . . 14
2.6 Analisi delle prestazioni . . . 14
3 Multi-Way Search Tree (MWST) . . . 15
3.1 (a, b)-Tree . . . 16
3.2 Insert . . . 18
3.3 Delete . . . 19
3.4 B-Tree . . . 21
4 Red-Black Trees e (2, 4) Trees . . . 23
4.1 (2, 4) Trees . . . 23
4.2 Red-Black Trees . . . 24
4.3 Dal (2, 4)-Tree al Red-Black Tree . . . 25
4.4 Dal Red-Black Tree al (2, 4)-Tree . . . 25
4.5 Altezza di un Red-Black Trees . . . 26
4.6 Insert . . . 26
4.6.1 Come risolvere un double red . . . 26
4.6.2 Complessità dell'inserimento . . . 27
4.7 Delete . . . 28
4.7.1 Caso 1: Delete di un nodo rosso . . . 28

4.7.2 Caso 2: Delete di un nodo nero con un solo figlio (rosso) 28
4.7.3 Caso 3: Delete di un nodo nero senza figli 29
4.7.4 Complessità della cancellazione 34
5 Hash Tables 35
5.1 Hash Functions 36
5.1.1 Esempi di Hash Code 38
5.1.2 Esempi di Funzione di Compressione 38
5.2 Gestione delle Collisioni 39
5.2.1 Separate Chaining 39
5.2.2 Open Addressing 40
5.3 Load Factor 43
5.4 Analisi delle prestazioni 43
6 Priority Queues 45
6.1 Chiavi (Priorità) 45
6.2 Possibili implementazioni di Priority Queue 46
6.3 Heap 47
6.4 Implementazione di una Priority Queue attraverso Heap 47
6.5 Implementazione Array-based di un Heap 52
6.6 Analisi delle prestazioni 53
6.7 Costruzione di un Heap da una lista di elementi: Heapify 53
6.8 Heap-Sort 56
6.9 Adaptable Priority Queue 59
7 Pattern Matching 61
7.1 Brute Force 62
7.2 L'algoritmo di Boyer-Moore 64
7.3 L'algoritmo di Knuth-Morris-Pratt 69
7.3.1 Failure-Function 69
8 Tries 73
8.1 Standard Tries 73
8.2 Compressed Tries 76
8.3 Suffix Tries 78
8.4 Pattern Matching con Suffix Tries 79

Capitolo 1

Binary Search Tree (BST)

Definizione: La struttura dati **albero binario di ricerca (BST)** è un albero binario i cui nodi contengono elementi del tipo chiave-valore, e soddisfa le seguenti proprietà:

- La chiave di ogni nodo è maggiore di tutte le chiavi nel suo sottoalbero sinistro.
- La chiave di ogni nodo è minore di tutte le chiavi nel suo sottoalbero destro.
- Tutti i nodi esterni non contengono elementi e sono considerati come nodi nulli (None) o nodi foglia.

Questa struttura (che non ammette chiavi duplicate) consente di eseguire le operazioni fondamentali di ricerca, inserimento e cancellazione in tempo proporzionale all'altezza dell'albero ($O(h)$).

1.1 BST Insert

L'inserimento di una coppia chiave-valore *(k, v)* in un BST avviene seguendo questi passaggi:

- Per prima cosa si esegue una ricerca per la chiave k nell'albero, per verificare se k esiste già nell'albero o in alternativa per trovare la posizione corretta per l'inserimento.
  - Partendo dalla radice, si confronta k con la chiave del nodo corrente:
  - Se k è minore, si procede nel sottoalbero sinistro.
  - Se k è maggiore, si procede nel sottoalbero destro.
  - Questo processo continua fino a raggiungere un nodo *p* che sarà il padre del nuovo nodo o fino a trovare k.
- Se k è già presente, si aggiorna il valore associato a k con v.
- Se k non è presente, si crea un nuovo nodo con la coppia (k, v) e lo si inserisce come figlio di p.
  - Se k < p.key(), il nuovo nodo viene inserito come figlio sinistro di p.
  - Se k > p.key(), viene inserito come figlio destro di p.

1.2 BST Delete

Per prima cosa si esegue una ricerca per la chiave k per verificare che k esista nell'albero.

- Se k non è presente, l'operazione termina senza modifiche all'albero.
- Se k è presente, si procede con la cancellazione facendo una distinzione tra due casi:
  - Il nodo da cancellare ha al massimo un figlio.
  - Il nodo da cancellare ha due figli.

1.2.1 Cancellazione di un nodo con al massimo un figlio

In questo caso, possiamo semplicemente rimuovere il nodo da cancellare p e collegare il suo unico figlio r (se esiste) al padre.

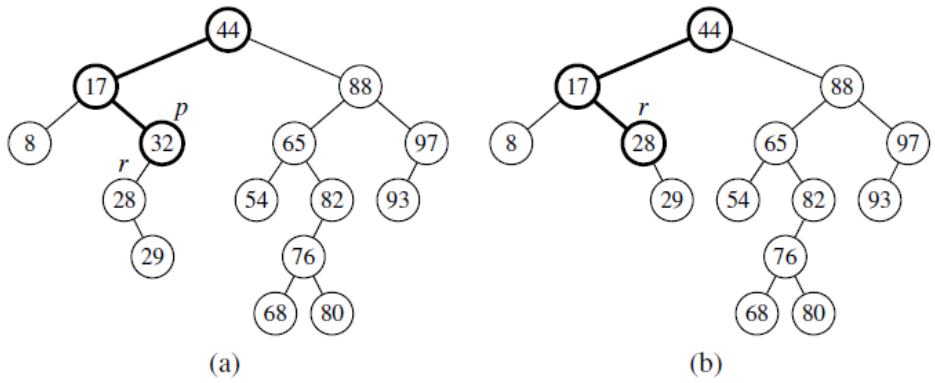


Figure 11.5: Deletion from the binary search tree of Figure 11.4b, where the item to delete (with key 32) is stored at a position p with one child r: (a) before the deletion; (b) after the deletion.

1.2.2 Cancellazione di un nodo con due figli

In questo caso, dobbiamo trovare un nodo sostituto per mantenere le proprietà del BST. Il nodo sostituto può essere (arbitrariamente) il *successore* (il nodo con la chiave più piccola [il nodo più a sinistra] nel sottoalbero destro) o il *predecessore* (il nodo con la chiave più grande [il nodo più a destra] nel sottoalbero sinistro).

Ipotizziamo di utilizzare il predecessore: $r = before(p)$. Una volta trovato il nodo sostituto r, copiamo la sua chiave e valore nel nodo da cancellare p e poi cancelliamo il nodo r, che avrà al massimo un figlio (quello sinistro) in quanto è il nodo più a destra nel sottoalbero sinistro. Dunque, dopo aver scambiato i valori, possiamo procedere con la cancellazione di r come nel primo caso, collegando il suo figlio (se esiste) al padre di r.

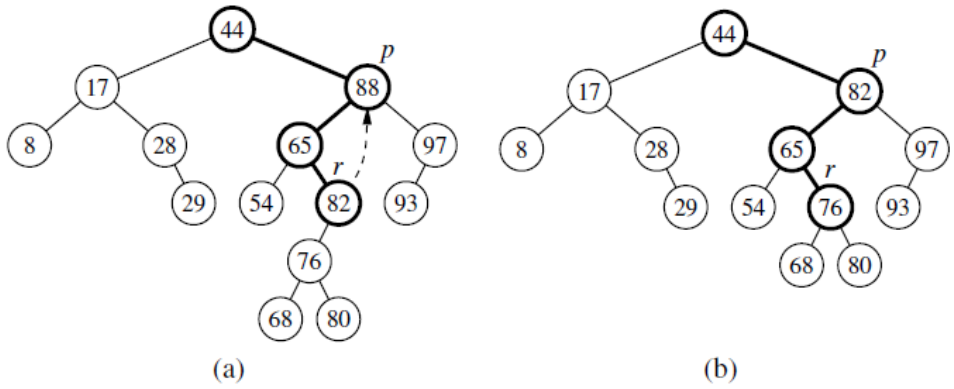


Figure 11.6: Deletion from the binary search tree of Figure 11.5b, where the item to delete (with key 88) is stored at a position p with two children, and replaced by its predecessor r: (a) before the deletion; (b) after the deletion.

1.3 Analisi delle prestazioni

Operation	Running Time
k in T	$O(h)$
T[k], T[k] = v	$O(h)$
T.delete(p), del T[k]	$O(h)$
T.find_position(k)	$O(h)$
T.first( ), T.last( ), T.find_min( ), T.find_max( )	$O(h)$
T.before(p), T.after(p)	$O(h)$
T.find_lt(k), T.find_le(k), T.find_gt(k), T.find_ge(k)	$O(h)$
T.find_range(start, stop)	$O(s+h)$
iter(T), reversed(T)	$O(n)$

Table 11.1: Worst-case running times of the operations for a TreeMap T. We denote the current height of the tree with h, and the number of items reported by find_range as s. The space usage is $O(n)$, where n is the number of items stored in the map.

Un BST è un'implementazione efficiente di un Map solo quando la sua altezza h è piccola. Partiamo dalla relazione che lega il numero di nodi n all'altezza h nel caso migliore (un albero completo). Un albero di altezza h può contenere al massimo $2^{h+1} - 1$ nodi.

Quindi, per contenere n nodi, l'altezza h deve soddisfare:

$$\begin{aligned}
n &\leq 2^{h+1} - 1 && \text{(Nodi massimi per altezza } h\text{)} \\
n + 1 &\leq 2^{h+1} && \text{(Aggiungo 1 a entrambi i lati)} \\
\log_2(n+1) &\leq \log_2(2^{h+1}) && \text{(Applico } \log_2 \text{ a entrambi i lati)} \\
\log_2(n+1) &\leq h + 1 && \text{(Semplifico } \log_2(2^x) = x\text{)} \\
\log_2(n+1) - 1 &\leq h && \text{(Sottraggo 1)}
\end{aligned}$$

Dato che h deve essere il più piccolo intero che soddisfa questa disequazione, applichiamo la funzione **ceiling** (soffitto, $\lceil \dots \rceil$) al lato sinistro:

$$h = \lceil \log_2(n+1) - 1 \rceil$$

Che è matematicamente equivalente a scrivere:

$$h = \lceil \log_2(n+1) \rceil - 1$$

Per definizione, la funzione **floor** (pavimento, $\lfloor \dots \rfloor$) restituisce l'unico intero h che soddisfa $h \leq x < h + 1$. Di conseguenza, possiamo scrivere direttamente:

$$h = \lfloor \log_2(n) \rfloor$$

1.4 Richiami: Tipi di alberi binari

Esistono diverse definizioni precise per classificare gli alberi binari in base alla loro struttura e al loro bilanciamento. Queste distinzioni sono fondamentali, perché la forma dell'albero determina l'efficienza delle operazioni su di esso.[1]

La differenza tra un albero "completo" e un albero "perfetto" è particolarmente importante.

[1] Ecco le definizioni chiave (spesso confuse tra loro) dei vari tipi di alberi:

- **Albero Binario Pieno (Full Binary Tree):** Un albero in cui ogni nodo ha **zero o due figli**. Non sono ammessi nodi con un solo figlio.
- **Albero Binario Perfetto (Perfect Binary Tree):** Un albero *pieno* in cui tutte le **foglie si trovano allo stesso livello** (stessa profondità). È la forma "perfetta" che massimizza il numero di nodi per una data altezza h.
- **Albero Binario Completo (Complete Binary Tree):** Un albero in cui tutti i livelli, **tranne eventualmente l'ultimo**, sono completamente pieni. Se l'ultimo livello non è pieno, i suoi nodi sono "impacchettati" il più possibile a sinistra. Questa è la struttura usata dagli *Heap*.
- **Albero Bilanciato (Balanced Binary Tree):** Termine generico per un albero la cui altezza è garantita essere $O(\log n)$. Esistono diverse definizioni specifiche:
  - **Bilanciato in altezza (es. Albero AVL):** Un albero binario di ricerca in cui, per *ogni nodo*, la differenza tra l'altezza del sottoalbero sinistro e quella del sottoalbero destro è al massimo 1.
  - **Perfettamente bilanciato (o bilanciato in peso):** Un albero in cui, per *ogni nodo*, il numero di nodi nel sottoalbero sinistro differisce al massimo di 1 dal numero di nodi nel sottoalbero destro. Questa definizione è molto più restrittiva e raramente usata.
- **Albero Degenerato (o Sbilanciato):** Il caso peggiore. Un albero in cui ogni nodo genitore ha un solo nodo figlio, trasformandosi essenzialmente in una lista concatenata. L'altezza è $O(n)$.

Capitolo 2

AVL Tree (AVLT)

Inserire n elementi casuali in un BST produce, in media, un albero di altezza $O(\log n)$. Tuttavia, nel caso peggiore (ad esempio, inserendo gli elementi in ordine crescente), l'altezza dell'albero può diventare $O(n)$, degradando le prestazioni delle operazioni di ricerca, inserimento e cancellazione a tempo lineare.

Per evitare questo problema, vogliamo modificare le operazioni di inserimento e cancellazione in modo da mantenere l'altezza dell'albero sempre logaritmica rispetto al numero di nodi n, ossia $h = O(\log n)$, garantendo così prestazioni logaritmiche anche nel caso peggiore. Gli **alberi AVL** sono una delle strutture dati che implementano questa idea di bilanciamento automatico.

2.1 Alberi bilanciati e Balance Factor

Per misurare quantitativamente il bilanciamento di un albero, introduciamo una metrica chiamata **Balance Factor**.

Il balance factor di un nodo v, che indichiamo con $\beta(v)$, è definito come la differenza tra l'altezza del suo sottoalbero sinistro e l'altezza del suo sottoalbero destro.

La formula per il calcolo è:

$$\beta(v) = height(\text{left}(v)) - height(\text{right}(v))$$

Dove $height(\dots)$ è la funzione che calcola l'altezza di un sottoalbero. Per convenzione, **l'altezza di un sottoalbero nullo (inesistente) è** -1.

Il valore del balance factor ci dice lo stato del nodo:

- $\beta(v) = 0$: Il nodo è bilanciato (i due sottoalberi hanno la stessa altezza).
- $\beta(v) = +1$: Il sottoalbero sinistro è più alto di 1 (l'albero è "pendente a sinistra").
- $\beta(v) = -1$: Il sottoalbero destro è più alto di 1 (l'albero è "pendente a destra").

2.2 Definizione di Albero AVL

Un albero **AVL** è un albero binario di ricerca (BST) che soddisfa una specifica proprietà di bilanciamento basata sul *balance factor*.

Definizione: Un albero binario di ricerca è un **albero AVL** se, **per ogni nodo** v appartenente all'albero, il valore assoluto del suo balance factor $\beta(v)$ è al più 1.

$$|\beta(v)| = |height(\text{left}(v)) - height(\text{right}(v))| \leq 1$$

Questo vincolo, se mantenuto dopo ogni operazione, è sufficiente a garantire che l'altezza totale dell'albero h rimanga sempre logaritmica ($h = O(\log n)$). Se un'operazione di inserimento o cancellazione viola questa proprietà (creando un nodo con $\beta(v) = +2$ o -2), l'albero esegue delle specifiche operazioni chiamate **rotazioni** per ripristinare il bilanciamento.

2.3 Mantenimento della Proprietà AVL: Ristrutturazione

Come abbiamo visto, un albero AVL è un BST che deve obbedire alla proprietà di bilanciamento. Le operazioni standard di inserimento e cancellazione di un BST possono violare questa proprietà, creando un nodo con fattore di bilanciamento +2 o −2.

Quando questo accade, l'albero deve essere "riparato". L'operazione di riparazione è chiamata **ristrutturazione** (o ribilanciamento) e viene implementata attraverso una o più operazioni primitive chiamate **rotazioni**.

Dopo un inserimento (o una cancellazione), risaliamo dall'elemento inserito p (o dal padre p dell'elemento cancellato) verso la radice per aggiornare i fattori di bilanciamento. Chiamiamo z il **primo nodo antenato che incontriamo che risulta sbilanciato**, ovvero con $\beta(z) = +2$ o $\beta(z) = -2$.

Una volta identificato z, definiamo:

- y: il **figlio di** z **con altezza maggiore** (e nota che y deve essere un antenato di p).
- x: il **figlio di** y **con altezza maggiore**. (non può esserci un pareggio e la posizione x deve anche essere un antenato di p, possibilmente p stesso).

L'operazione di ribilanciamento, chiamata **ristrutturazione trinodale**, coinvolge sempre e solo questi tre nodi (x, y, z) e i loro 4 possibili sottoalberi. L'obiettivo è riordinare x, y e z in modo da ottenere un albero binario di ricerca bilanciato. Si identificano i tre nodi (in ordine crescente di chiave) come a, b, c. Il nodo con la chiave mediana (b) diventerà la nuova radice, a diventerà il suo figlio sinistro e c il suo figlio destro. I 4 sottoalberi vengono poi riagganciati ordinatamente. Questo processo logico unificato si traduce in quattro diversi casi, che richiedono due tipi di operazioni meccaniche: **le rotazioni singole** e **le rotazioni doppie**.

Un'operazione di rotazione singola o doppia ha sempre una complessità temporale di $O(1)$, poiché coinvolge solo un numero costante di nodi e puntatori.

2.3.1 Caso 1 e 2: Rotazioni Singole (Casi Esterni)

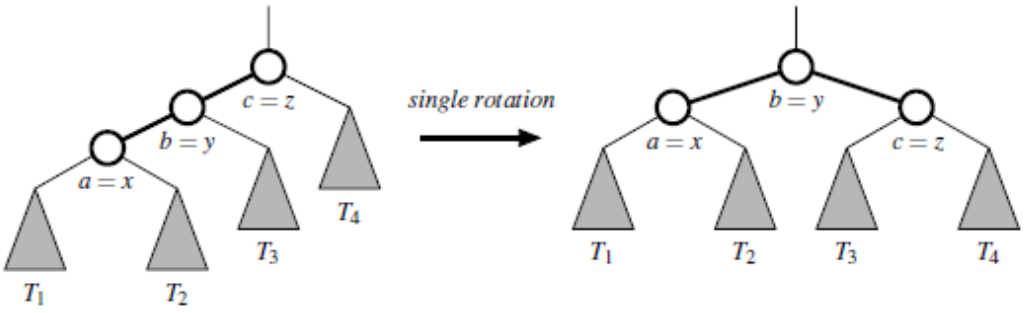
Si ha una rotazione singola quando x, y e z sono allineati sullo stesso lato.

Caso Sinistra-Sinistra (LL)

Questo caso si verifica quando y è il figlio sinistro di z e x è il figlio sinistro di y.

- z: Nodo sbilanciato ($\beta(z) = +2$)
- y: Figlio sinistro di z
- x: Figlio sinistro di y

Soluzione: Rotazione Singola a Destra (su z) Il nodo b diventa la nuova radice del sottoalbero. Il nodo a rimane il figlio sinistro di b, mentre c diventa il figlio destro di b. Inoltre, il sottoalbero destro di b viene agganciato come nuovo sottoalbero sinistro di c.



Caso Destra-Destra (RR)

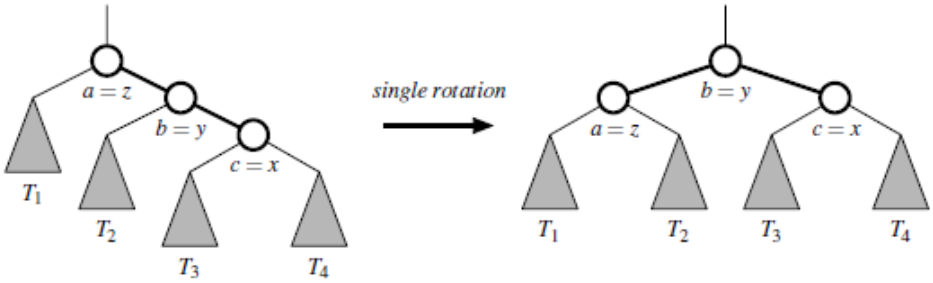
Questo è il caso speculare. y è il figlio destro di z e x è il figlio destro di y.

- z: Nodo sbilanciato ($\beta(z) = -2$)
- y: Figlio destro di z
- x: Figlio destro di y

Soluzione: Rotazione Singola a Sinistra (su z) Il nodo b diventa la nuova radice del sottoalbero. Il nodo a diventa il figlio sinistro di b, mentre c rimane il figlio destro di b. Inoltre, il sottoalbero sinistro di b viene agganciato come nuovo sottoalbero destro di a.

2.3.2 Caso 3 e 4: Rotazioni Doppie (Casi Interni)

Si ha una rotazione doppia quando x, y e z formano un "gomito" (o "zig-zag").

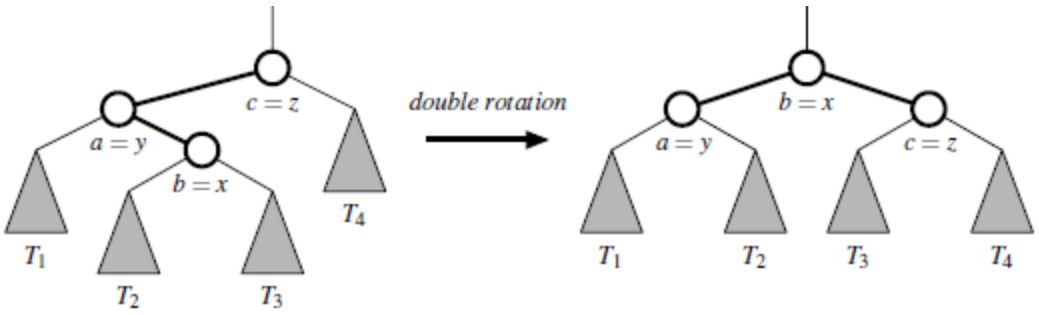


Caso Sinistra-Destra (LR)

Questo caso si verifica quando y è il figlio sinistro di z, ma x è il figlio destro di y.

- z: Nodo sbilanciato ($\beta(z) = +2$)
- y: Figlio sinistro di z
- x: Figlio **destro** di y

Soluzione: Doppia Rotazione (Sinistra-Destra) È sempre b a diventare la nuova radice del sottoalbero, con a come figlio sinistro e c come figlio destro. Inoltre, il sottoalbero sinistro di b viene agganciato come nuovo sottoalbero destro di a, e il sottoalbero destro di b viene agganciato come nuovo sottoalbero sinistro di c.

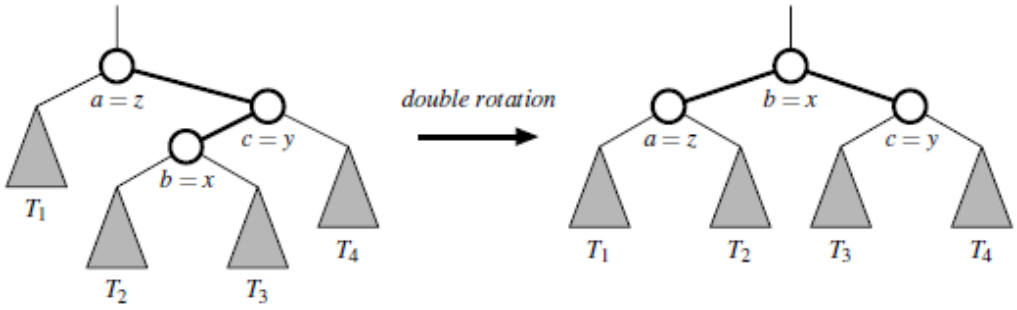


Caso Destra-Sinistra (RL)

Questo è il caso speculare. y è il figlio destro di z, ma x è il figlio *sinistro* di y.

- z: Nodo sbilanciato ($\beta(z) = -2$)
- y: Figlio destro di z
- x: Figlio **sinistro** di y

Soluzione: Doppia Rotazione (Destra-Sinistra) È sempre b a diventare la nuova radice del sottoalbero, con a come figlio sinistro e c come figlio destro. Inoltre, il sottoalbero sinistro di b viene agganciato come nuovo sottoalbero destro di a, e il sottoalbero destro di b viene agganciato come nuovo sottoalbero sinistro di c.



2.4 AVL Insert

L'inserimento in un albero AVL segue le stesse regole di un normale albero binario di ricerca, con l'aggiunta della necessità di mantenere l'equilibrio dell'albero. Dopo aver inserito un nuovo nodo, denotiamo questo nodo come p, e si risale lungo il cammino verso la radice, aggiornando i fattori di bilanciamento e applicando le rotazioni necessarie per ripristinare l'equilibrio al primo nodo z che risulta essere non bilanciato.

In particolare, in un albero AVL, l'applicazione di una singola ristrutturazione è sufficiente per ripristinare l'equilibrio dell'intero albero dopo un inserimento problematico.

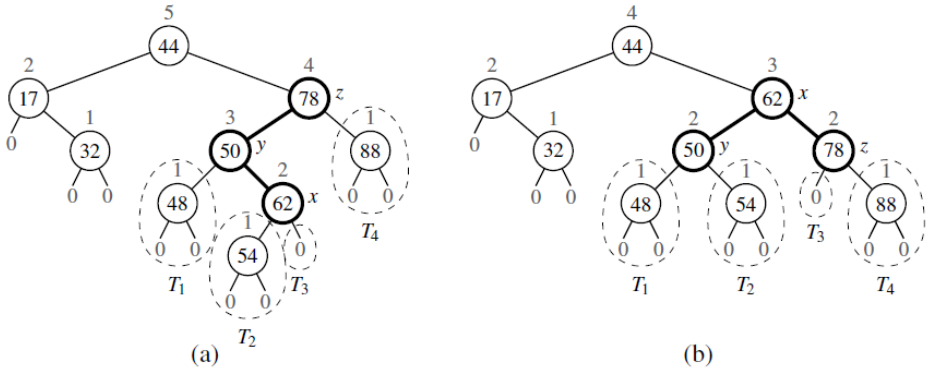


Figure 11.12: An example insertion of an item with key 54 in the AVL tree of Figure 11.11: (a) after adding a new node for key 54, the nodes storing keys 78 and 44 become unbalanced; (b) a trinode restructuring restores the height-balance property. We show the heights of nodes above them, and we identify the nodes x, y, and z and subtrees T_1, T_2, T_3, and T_4 participating in the trinode restructuring.

2.5 AVL Delete

La cancellazione in un albero AVL segue le stesse regole di un normale albero binario di ricerca, con l'aggiunta della necessità di mantenere l'equilibrio dell'albero. Dopo aver cancellato un nodo, denotiamo il padre del nodo cancellato come p, e si risale lungo il cammino verso la radice, aggiornando i fattori di bilanciamento e applicando le rotazioni necessarie per ripristinare l'equilibrio a ogni nodo z che risulta essere non bilanciato.

In particolare, in un albero AVL, può essere necessario eseguire fino a $O(\log n)$ ristrutturazioni per ripristinare l'equilibrio dell'intero albero dopo una cancellazione problematica.

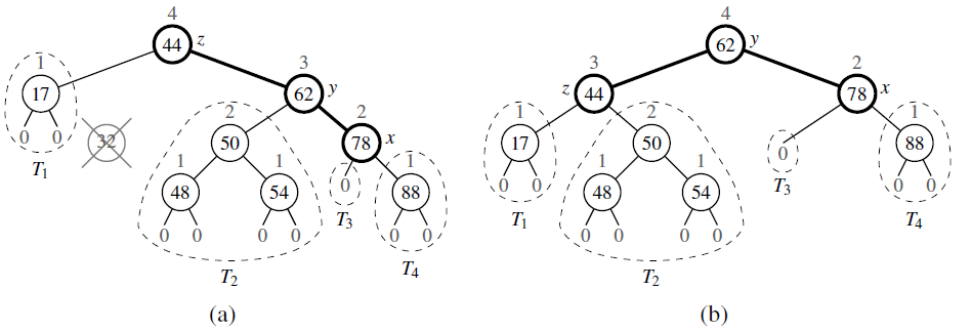


Figure 11.14: Deletion of the item with key 32 from the AVL tree of Figure 11.12b: (a) after removing the node storing key 32, the root becomes unbalanced; (b) a (single) rotation restores the height-balance property.

2.6 Analisi delle prestazioni

Operation	Running Time
k in T	$O(\log n)$
T[k] = v	$O(\log n)$
T.delete(p), del T[k]	$O(\log n)$
T.find_position(k)	$O(\log n)$
T.first( ), T.last( ), T.find_min( ), T.find_max( )	$O(\log n)$
T.before(p), T.after(p)	$O(\log n)$
T.find_lt(k), T.find_le(k), T.find_gt(k), T.find_ge(k)	$O(\log n)$
T.find_range(start, stop)	$O(s+\log n)$
iter(T), reversed(T)	$O(n)$

Table 11.2: Worst-case running times of operations for an n-item sorted map realized as an AVL tree T, with s denoting the number of items reported by find_range.

L'altezza di un albero AVL con n elementi è garantita essere $O(\log n)$. Poiché l'operazione standard di ricerca binaria su albero aveva tempi di esecuzione limitati dall'altezza e poiché il lavoro aggiuntivo per mantenere i fattori di bilanciamento e ristrutturare un albero AVL può essere limitato dalla lunghezza di un percorso nell'albero, le operazioni di mappatura tradizionali vengono eseguite nel caso peggiore in tempo logaritmico con un albero AVL.

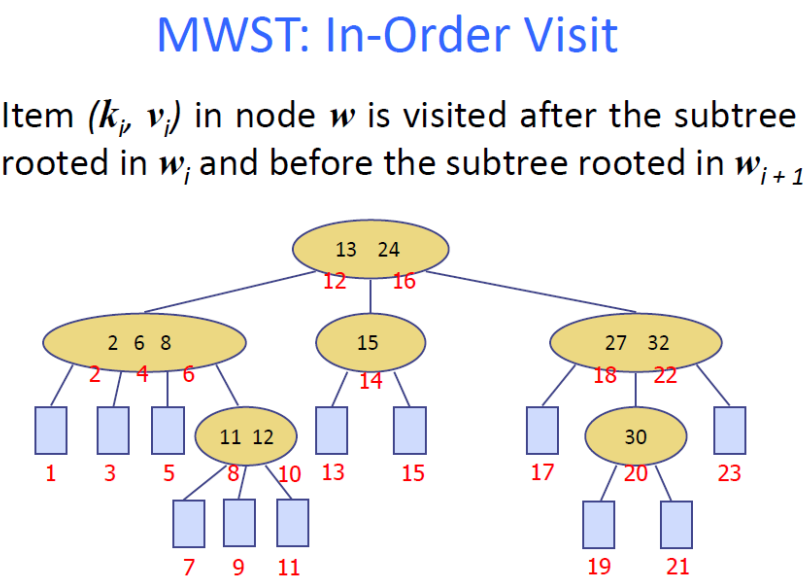
Capitolo 3

Multi-Way Search Tree (MWST)

Definizione: Un **Multi-Way Search Tree (MWST)** è un albero ordinato in cui:

- Ogni nodo dell'albero (d-nodo) ha $d \geq 2$ figli e contiene $d - 1$ elementi chiave valore (k_i, v_i) ordinati in modo crescente per chiave.
- Sia w un nodo con figli $w_1, w_2, \ldots, w_d$ e con chiavi $k_1, k_2, \ldots, k_{d-1}$. Allora:
  - Tutte le chiavi nel sottoalbero radicato in w_1 sono minori di k_1.
  - Tutte le chiavi nel sottoalbero radicato in w_i sono comprese tra k_{i-1} e k_i $(i = 2, \ldots, d - 1)$.
  - Tutte le chiavi nel sottoalbero radicato in w_d sono maggiori di k_{d-1}.
- I nodi foglia (None) non contengono elementi.

Si osservi come un MWST contenente n elementi abbia $n + 1$ nodi foglia (None).

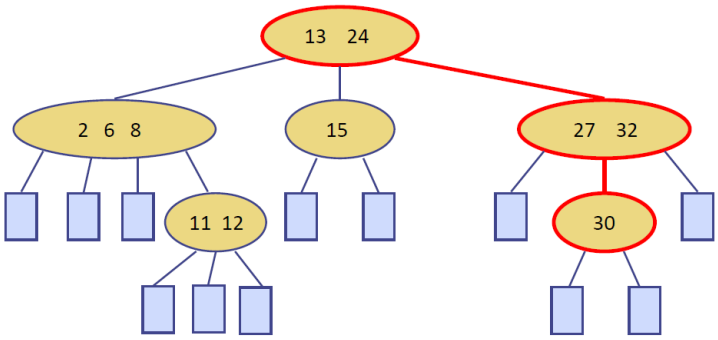


MWST: Search

- Looking for key k
  - Consider an internal node w with children $w_1\, w_2 \ldots w_d$ with keys $k_1\, k_2 \ldots k_{d-1}$
  - If $k = k_i$ (i = 1, 2, ... d-1), search ends successfully
  - If $k < k_1$ search in the subtree rooted in w_1
  - If $k_{i-1} < k < k_i$ search in the subtree rooted in w_i (i = 2,... d-1)
  - If $k > k_{d-1}$ search in the subtree rooted in w_d
  - If we reach a leaf, search ends unsuccessfully

MWST Search: Example

Find(30)



3.1 (a, b)-Tree

Definizione: Gli **(a, b)-Tree** sono dei *MWST* che soddisfano le seguenti proprietà:

- $2 \leq a \leq \lceil \frac{b-1}{2} \rceil$
- **Root Property:** La radice ha almeno 2 figli e al più b figli.
- **Node-Size Property:** Ogni nodo diverso dalla radice ha almeno a figli e al più b figli.
- **Depth Property:** Tutti i nodi foglia (None) sono allo stesso livello.

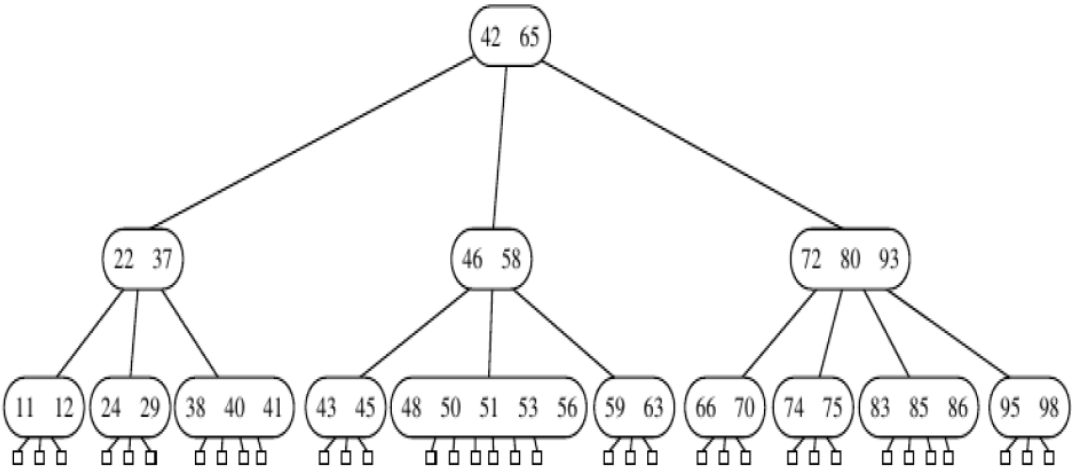


Figura 3.1: Esempio di un (a, b)-Tree con $a = 3$ e $b = 6$.

Altezza di un (a, b)-Tree e Ricerca di un elemento

L'altezza di un (a, b)-Tree con n elementi è:

- $\Omega(\log n/\log b)$ [== $\Omega(\log_b n)$].
- $O(\log n/\log a)$ [== $O(\log_a n)$].

Per ogni nodo, è necessario eseguire una ricerca tra tutti i suoi elementi. Un nodo è una *Map* di al più $b - 1$ elementi, per cui possiamo denotare con $f(b)$ il costo della ricerca in un nodo. Se l'elemento non è stato trovato è necessario scendere in uno dei figli, per cui il costo totale della ricerca in un (a, b)-Tree è:

$$O(f(b) \cdot \log n/\log a)$$

Se $f(b)$ è costante, il costo della ricerca è migliore di $O(\log n)$.

3.2 Insert

Per inserire un elemento (k, v) in un (a, b)-Tree, si procede come segue:

1. Si esegue una ricerca per trovare la posizione corretta in cui inserire l'elemento: supponendo che l'albero non presenti già un elemento con chiave k, la ricerca termina senza successo restituendo il nodo foglia (None) z. Sia w il genitore di z, inseriamo il nuovo elemento in w e aggiungiamo una nuova foglia (None) y come figlio di w.

2. A questo punto, se w ha meno di $b - 1$ elementi (cioè ha meno di b figli), l'inserimento è terminato.

3. Altrimenti, se w ha già $b - 1$ elementi (cioè b figli), si verifica un **overflow** e dobbiamo eseguire un **split** di w:

    - Consideriamo il nodo w con i suoi $b - 1$ elementi, più il nuovo elemento appena inserito.
    - Si esegue uno **split** di w in tre parti: gli elementi minori di k', l'elemento mediano k' e gli elementi maggiori di k'.
    - Si crea un nuovo nodo che conterrà gli elementi minori di k'. (Questo nodo è automaticamente valido perchè contiene almeno $\lceil (b-1)/2 \rceil - 1 \geq a - 1$ elementi e sicuramente meno di $b - 1$ elementi).
    - Vale lo stesso per il nodo che conterrà gli elementi maggiori di k'.
    - L'elemento mediano k' viene promosso al genitore p di w e diventa un nuovo elemento di p, con i due nodi risultanti dallo split come suoi figli.
        - Se p ha ora b elementi, si ripete la procedura di split su p.
        - Altrimenti, l'inserimento è terminato.
        - Se il nodo splittato era la radice, si crea una nuova radice che contiene solo l'elemento mediano m e ha come figli i due nodi risultanti dallo split.

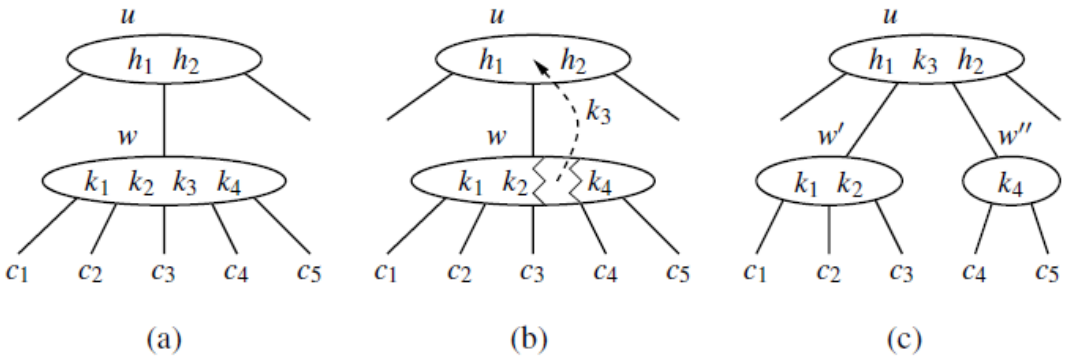


Figura 3.2: Esempio di inserimento con overflow in un (a, b)-Tree con $a = 2$ e $b = 4$. (a) overflow in un 5-nodo w [il massimo è un 4-nodo]; (b) e (c) split di w.

3.3 Delete

Per eliminare un elemento (k, v) in un (a, b)-Tree, si procede come segue:

1. Si esegue una ricerca per trovare il nodo w contenente l'elemento da eliminare k. Se il nodo non esiste, l'operazione termina.

2. Se il figlio a sinistra o a destra di k non è vuoto, oppure w è la radice, allora:

   - Si trova il predecessore o il successore di k (cioè l'elemento più grande del sottoalbero sinistro o l'elemento più piccolo del sottoalbero destro) e lo indichiamo con k'.
   - Si scambia l'elemento con chiave k con l'elemento con chiave k'.
   - Si elimina l'elemento con chiave k' dal sottoalbero (l'elemento originario con chiave k').

3. Altrimenti, se entrambi i figli di k sono vuoti, e:

   - il nodo w ha $> a-1$ elementi, eliminiamo l'elemento con chiave k da w e l'operazione termina.
   - il nodo w ha esattamente il numero minimo di $a-1$ elementi, si verifica un **underflow**.

4. Gestione dell'underflow:

   - Se il nodo x fratello sinistro di w (con lo stesso genitore p) ha più di $a-1$ elementi, si esegue un **transfer**:
     - Sia k' la chiave salvata nel genitore p "che sta tra il puntatore a x e il puntatore a w".
     - Sia k'' la chiave più grande nel nodo x(ricordiamo che x è il fratello a sinistra di w).
     - Cancella k da w, cancella k'' da x e sostituisci k' con k'' in p, e aggiungi k' in w.
   - Se il nodo x fratello destro di w (con lo stesso genitore p) ha più di $a-1$ elementi, si esegue un **transfer**:
     - L'opposto del caso precedente: si prende la chiave più piccola dal fratello destro x e la si sposta in w, aggiornando di conseguenza il genitore p.
   - Altrimenti, se entrambi i fratelli di w hanno esattamente $a-1$ elementi, si esegue una **fusion**:
     - Sia x un fratello di w (a sinistra o a destra) con lo stesso genitore p.
     - Sia k' la chiave salvata nel genitore p "che sta tra il puntatore a x e il puntatore a w".
     - Si crea un nuovo nodo che contiene tutti gli elementi di w eccetto l'elemento con chiave k, tutti gli elementi di x e l'elemento con chiave k'.

∗ È un nodo valido perchè contiene un numero di elementi pari a:

$$a - 1 \leq (a - 2) + (a - 1) + 1 = 2a - 2 \leq b - 1$$

– Si elimina k' da p.

∗ Se p è la radice e k' è il suo unico elemento, il nuovo nodo creato diventa la radice.

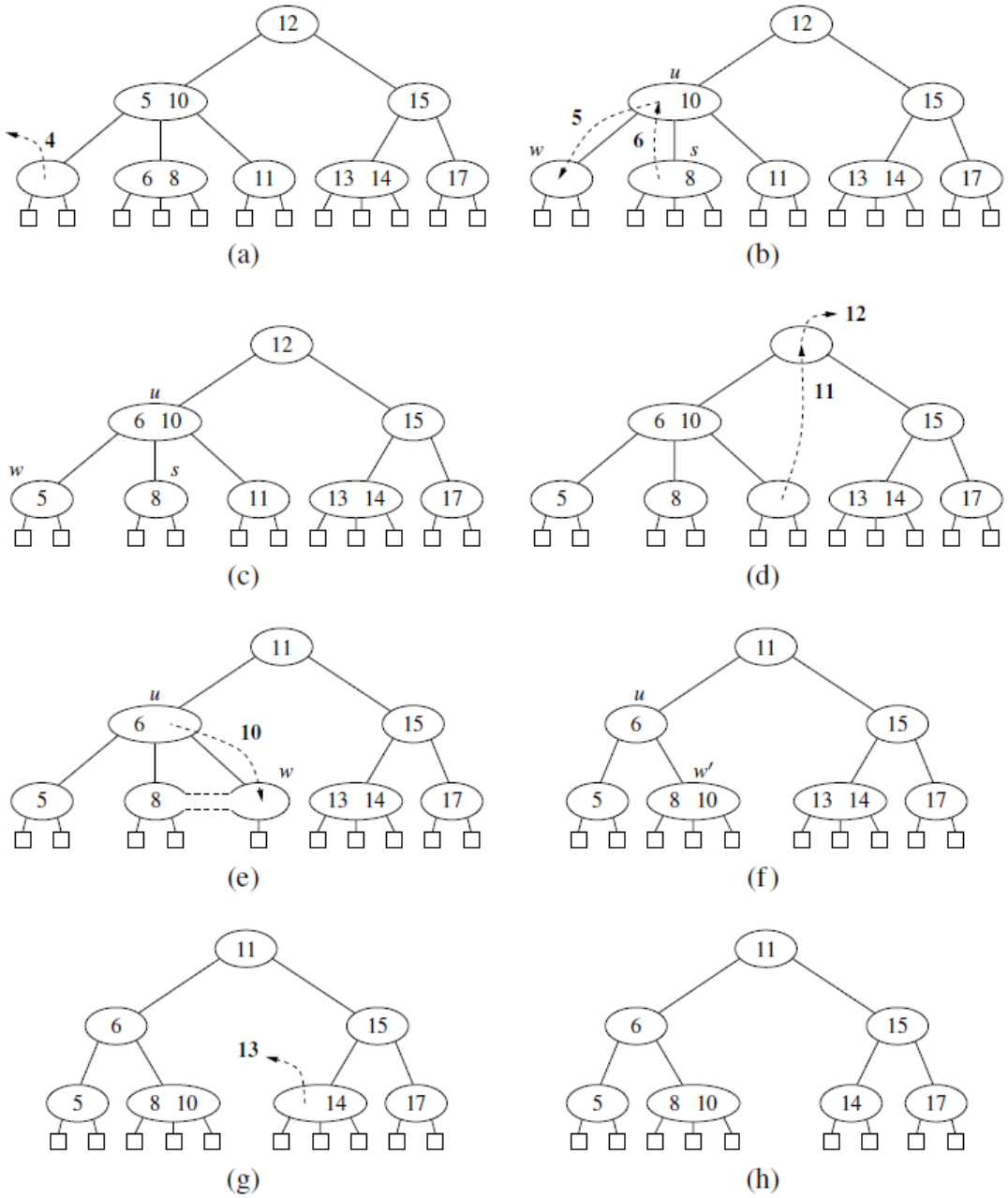


Figura 3.3: Una sequenza di rimozioni in un (a, b)-Tree con $a = 2$ e $b = 4$. (a) rimozione di 4, che causa un underflow; (b) un'operazione di transfer; (c) dopo l'operazione di transfer; (d) rimozione di 12, che causa un underflow; (e) un'operazione di fusion; (f) dopo l'operazione di fusion; (g) rimozione di 13; (h) dopo la rimozione di 13.

Complessità delle operazioni di inserimento e cancellazione

Si tenga presente che la ricerca di un nodo, come visto prima, impiega un tempo $O(f(b) \cdot \log n/\log a)$. Supponendo che la gestione di un overflow/underflow richieda al più $g(b)$ [$g(b)$ dipende dall'implementazione del nodo], e considerando che potrebbe essere necessario ripetere al più tali operazioni dal livello $h-1$ fino alla radice. Quindi, il costo totale dell'inserimento in un (a, b)-Tree è:

$$O\left((f(b)+g(b))\log n/\log a\right)$$

Come scegliere a e b?

Se i nodi sono troppo piccoli, l'albero sarà essenzialmente simile ad un albero binario di ricerca bilanciato. Nonostante ciò, vedremo che i (2, 4)-Tree sono particolarmente importanti perchè possono essere trasformati in degli alberi particolari chiamati **Red-Black Tree**.

Se i nodi sono troppo grandi, l'albero diventa meno efficiente in termini di spazio e di tempo per le operazioni di ricerca, inserimento e cancellazione. Dipende da come è implementata la Map all'interno del nodo.

- Se si tratta di Hash Tables, la ricerca e gli aggiornamenti sono $O(1)$, ma il calcolo della mediana è $O(b)$.
- Se si tratta di BST bilanciati, la ricerca e gli aggiornamenti sono $O(\log b)$ e il calcolo della mediana è $O(b)$.
- Se si tratta di vettori ordinati, la ricerca è $O(\log b)$, gli aggiornamenti sono $O(b)$ e il calcolo della mediana è $O(1)$.

Idealmente, vorremmo che tutte queste operazioni vadano come $O(1)$, ma ciò non è possibile. In pratica, si cerca di minimizzare b in modo che le operazioni siano efficienti, ma abbastanza grande da ridurre l'altezza dell'albero.

Se a e b sono troppo distanti tra loro, la complessità delle operazioni interne al nodo cancellano il vantaggio dato dalla riduzione dell'altezza [$f(b)$ e $g(b)$ crescono troppo rapidamente fino a diventare di gran lunga più pesanti di $\log a$].

3.4 B-Tree

Un **B-Tree** è un (a, b)-Tree con $a = \lceil (b-1)/2 \rceil$ e $b = d$.
Per le considerazioni fatte sugli (a, b)-Tree, si ha che:

- La ricerca ha una complessità di $O(f(d) \cdot \log n/(\log d - 1))$.
- L'inserimento e la cancellazione hanno una complessità di $O(g(d) \cdot \log n/(\log d - 1))$.

I/O Complexity

Consideriamo il problema della gestione di una grande raccolta di elementi che non rientrano nella memoria principale, come un tipico database. In questo contesto, ci riferiamo ai blocchi di memoria secondaria come *disk blocks*. Allo stesso modo, ci riferiamo al trasferimento di un blocco tra la memoria secondaria e la memoria primaria come *disk transfer*. Ricordando la grande differenza di tempo che esiste tra gli accessi alla memoria principale e gli accessi al disco, l'obiettivo principale della gestione di tale raccolta nella memoria esterna è quello di ridurre al minimo il numero di trasferimenti su disco necessari per eseguire una query o un aggiornamento. Ci riferiamo a questo numero come **I/O complexity** dell'algoritmo coinvolto.

Il modo migliore di ridurre al minimo il numero di trasferimenti su disco è quello di massimizzare il numero di elementi che possono essere memorizzati in un singolo blocco. Supponiamo che ogni blocco di disco possa contenere B elementi. Nel caso di un B-Tree, scegliamo il suo *ordine* d (il numero massimo di figli per nodo) il più grande possibile, in modo tale che un nodo – contenente $O(d)$ elementi e puntatori – occupi al massimo un singolo blocco di disco. Si ottiene così una relazione diretta tra d e B, ovvero $d = \Theta(B)$. Di conseguenza, ogni accesso a un nodo del B-Tree corrisponde a un singolo trasferimento su disco. Scegliendo d così grande, si massimizza il fattore di diramazione (branching factor) e si minimizza l'altezza dell'albero. Poiché l'altezza di un B-Tree con n elementi è $O(\log_d n)$, e dato che $d = \Theta(B)$, la **I/O complexity** per la **ricerca** è $O(\log_B n)$. Anche l'inserimento e la cancellazione mantengono questa complessità, poiché, oltre alla ricerca iniziale, richiedono un numero di operazioni di modifica (come divisioni o fusioni di nodi) proporzionale all'altezza dell'albero. In conclusione, i B-Tree sono una struttura dati molto efficiente per la gestione di grandi raccolte di elementi nella memoria esterna.

Capitolo 4

Red-Black Trees e (2, 4) Trees

Abbiamo già parlato di alberi bilanciati nel Capitolo 2, in particolare degli alberi AVL. Questo tipo di alberi consentiva di mantenere l'altezza dell'albero logaritmica rispetto al numero di nodi presenti, garantendo così operazioni di ricerca, inserimento e cancellazione efficienti. Tuttavia, una cancellazione in un albero AVL poteva richiedere molte rotazioni per mantenere l'equilibrio, rendendo l'operazione più costosa in termini di tempo.

4.1 (2, 4) Trees

I (2, 4) Trees sono semplicemente degli (a, b) Trees con $a = 2$ e $b = 4$. Ciò significa che ereditano tutte le proprietà degli (a, b) Trees discusse nel Capitolo 3, e dal punto di vista di un'analisi asintotica, le performance di un (2, 4) Tree sono equivalenti a quelle di un albero AVL. Per un (2, 4) Tree:

- L'altezza è $O(\log n)$.
- Le operazioni di split, transfer e fusion hanno una complessità di $O(1)$.
- Le operazioni di ricerca, inserimento e cancellazione hanno una complessità di $O(\log n)$.

All'interno di un (2, 4) Tree, ogni nodo può contenere da 1 a 3 chiavi e può avere da 2 a 4 figli. Questo consente di distinguere tra 2-nodi, 3-nodi e 4-nodi, sulla base del numero di figli (numero di chiavi + 1). Questo tipo di alberi è particolarmente interessante per la sua relazione che ha con un tipo particolare di alberi, i **Red-Black Trees**.

4.2 Red-Black Trees

Definizione: Un Red-Black Tree è un albero binario di ricerca in cui ogni nodo ha un colore: rosso o nero, e l'abero soddisfa le seguenti proprietà:

- **Root Property:** La radice deve essere nera.
- **External Property:** Tutti i nodi foglia (None) sono neri.
- **Internal Property:** I figli di un nodo rosso sono neri.
- **Depth Property:** Tutti i nodi foglia (None) hanno la stessa *black-depth*, ovvero il numero di antenati neri.

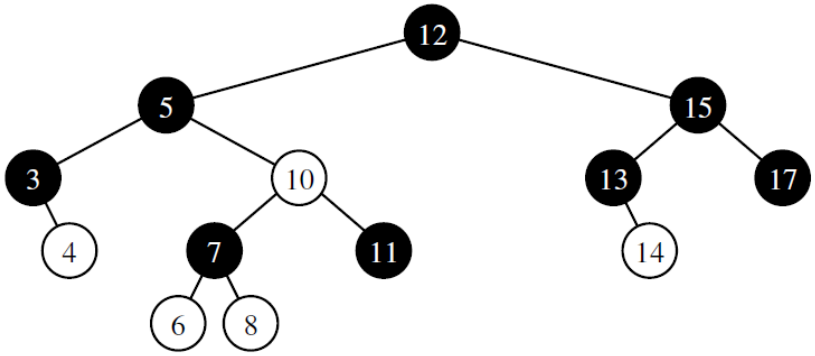


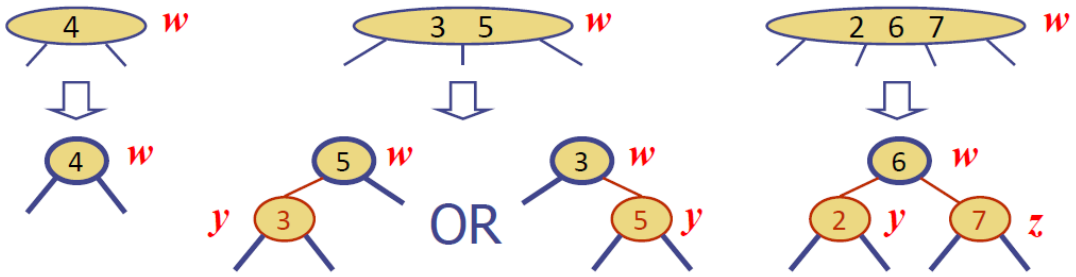
Figura 4.1: Esempio di Red-Black Tree, con i nodi rossi disegnati in bianco. La black-depth di ogni nodo foglia è 3. Da notare che non sono disegnati i nodi foglia (None), che sono tutti neri.

Come abbiamo detto, i Red-Black Trees sono strettamente correlati ai (2, 4) Trees. Infatti, ogni (2, 4) Tree può essere rappresentato come un Red-Black Tree e viceversa. Un Red-Black Tree può essere visto come una rappresentazione binaria di un (2, 4) Tree, dove i nodi rossi rappresentano i nodi con più di una chiave nel (2, 4) Tree. Proprio per questo motivo, i Red-Black Trees mantengono le stesse performance dei (2, 4)-Trees, con il beneficio aggiuntivo di una implementazione più semplice e di una maggiore efficienza nelle operazioni di inserimento e cancellazione, che richiedono al massimo una o due rotazioni per mantenere l'equilibrio dell'albero.

4.3 Dal (2, 4)-Tree al Red-Black Tree

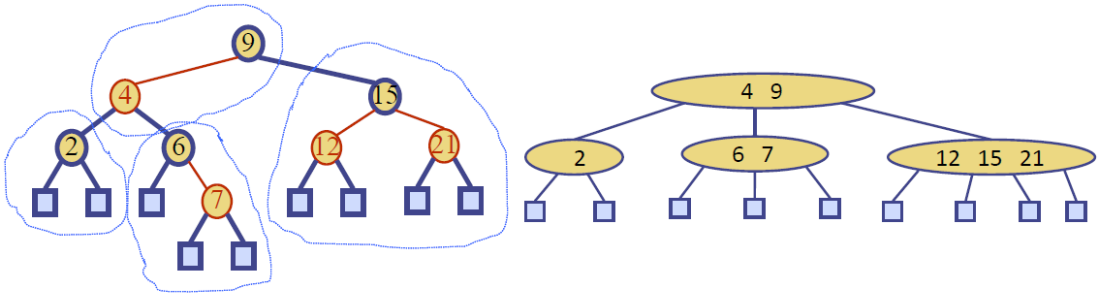
- Colora di nero tutti i nodi del (2, 4) Tree.
- Per ogni nodo w:
  - Se w è un 2-nodo, mantieni i figli (neri) di w così come sono.
  - Se w è un 3-nodo, crea un nuovo nodo rosso y, figlio destro (o sinistro) di w, e fai in modo che gli ultimi due (o i primi due) figli di w diventino figli di y, e il primo figlio (o l'ultimo) di w rimanga figlio di w.
  - Se w è un 4-nodo con chiavi k_1, k_2 e k_3, rappresentalo come un nodo nero con due figli rossi contenenti le chiavi k_2 e k_3.

Da notare che seguendo l'algoritmo sopra descritto, un nodo rosso avrà sempre un genitore nero.



4.4 Dal Red-Black Tree al (2, 4)-Tree

- Ogni nodo rosso w viene unito con il suo genitore nero p per formare un unico nodo del (2, 4) Tree.
  - L'elemento in w viene aggiunto alle chiavi in p.
  - I figli di w diventano figli di p.



4.5 Altezza di un Red-Black Trees

Sia T un Red-Black Tree con n nodi interni e altezza h, allora vale la seguente disuguaglianza:

$$\log(n+1) - 1 \leq h \leq 2\log(n+1) - 2$$

Sia d la black-depth di T. Sia T' il (2, 4)-Tree associato a T, e sia h' l'altezza di T'. Per via della corrispondenza tra T e T', vale la relazione $h' = d$. Quindi, si ha che $d = h' \leq \log(n+1) - 1$, da cui si ricava che $h \leq 2d \leq 2\log(n+1)$. Sappiamo inoltre che vale la seguente proprietà: $h' \leq 2d$. Quindi, otteniamo $h \leq 2\log(n+1) - 2$. L'altra disuguaglianza, $\log(n+1) - 1 \leq h$ deriva dalle proprietà di un qualsiasi albero binario.

4.6 Insert

Per inserire un nuovo nodo in un Red-Black Tree, si segue lo stesso procedimento di un normale BST:

- Se il nuovo nodo z è la radice, coloralo di nero.
- Altrimenti, inseriscilo come un nodo rosso.

L'inserimento eseguito in questo modo mantiene di già le Root, External, e Depth Properties. Tuttavia, potrebbe violare la Internal Property in alcuni casi:

- Se il genitore di z è nero, anche la Internal Property è mantenuta.
- Se il genitore di z è rosso, la Internal Property viene violata in quanto si ottiene una sequenza di due nodi rossi, un **double red**. In questo caso, bisogna ristrutturare l'albero per ripristinare le proprietà dei Red-Black Trees.

4.6.1 Come risolvere un double red

Siano z e il suo genitore v entrambi rossi, e sia w il fratello di v (lo zio di z).

- Se w è nero (o None), il double-red corrisponde ad una trasformazione sbagliata di un 4-nodo nel (2, 4)-Tree corrispondente.
  - Si esegue una **ristrutturazione** (Three-node restructuring, singola o doppia rotazione) su z.
  - Dopo la ristrutturazione, il nodo che diventa la radice della porzione ristrutturata viene colorato di nero, mentre i suoi due figli vengono colorati di rosso.
  - Una sola ristrutturazione è sufficiente per risolvere il problema del double-red.
- Se w è rosso, il double-red corrisponde ad un overflow e quindi in un 5-nodo nel (2, 4)-Tree corrispondente.
  - Si esegue una **ricolorazione** (recolouring) colorando v e w di nero, mentre il loro genitore u (genitore di v e w, e nonno di z) viene colorato di rosso (se u non è la radice).
  - In questo caso il double-red può propagarsi verso l'alto, quindi potrebbe essere necessario ripetere la procedura sul nodo u e il suo genitore.

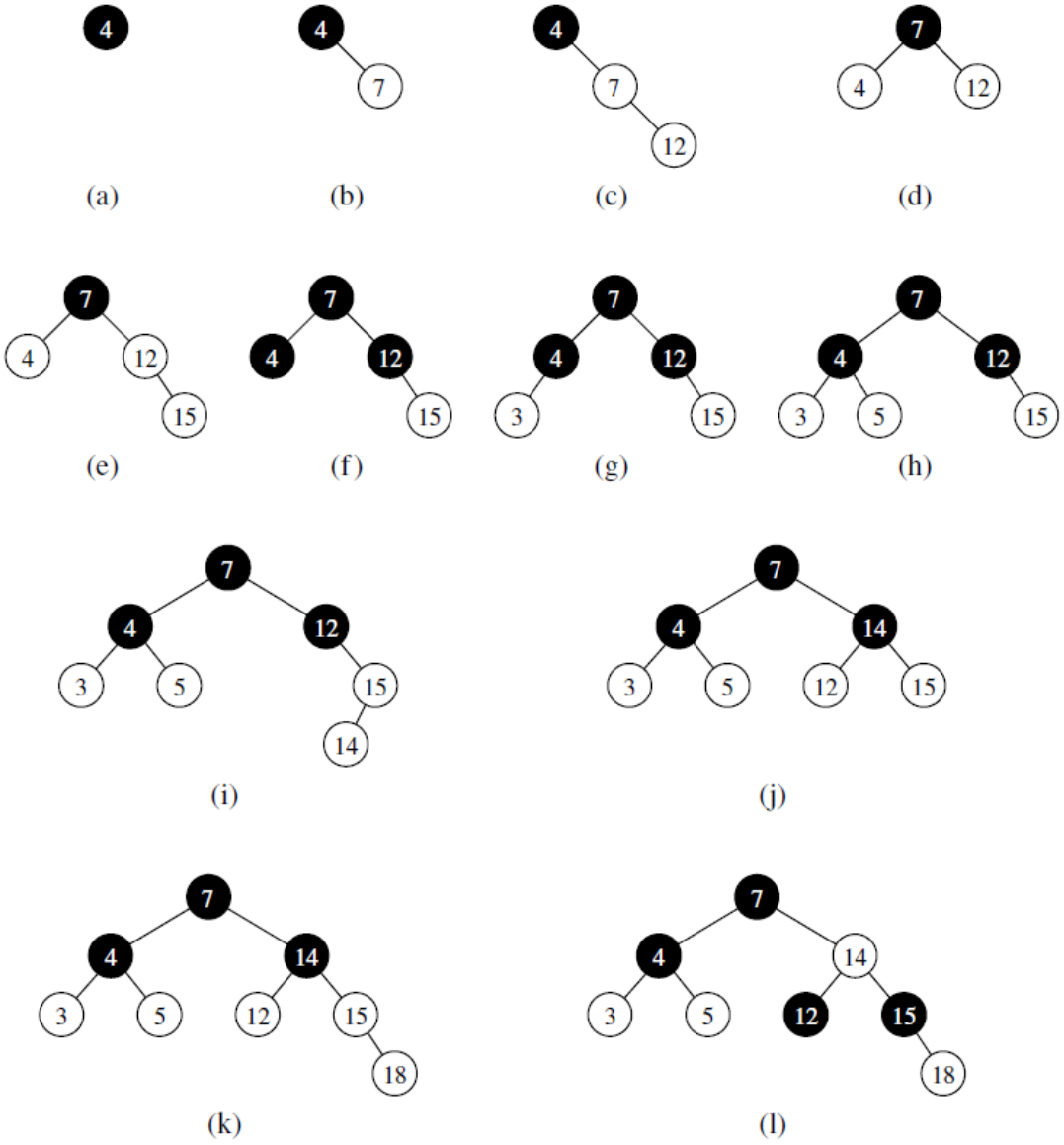


Figura 4.2: Una sequenza di inserimenti in un albero rosso-nero: (a) albero iniziale; (b) inserimento di 7; (c) inserimento di 12, che causa un double red; (d) dopo la ristrutturazione; (e) inserimento di 15, che causa un double red; (f) dopo la ricolorazione (la radice rimane nera); (g) inserimento di 3; (h) inserimento di 5; (i) inserimento di 14, che causa un double red; (j) dopo la ristrutturazione; (k) inserimento di 18, che causa un double red; (l) dopo la ricolorazione.

4.6.2 Complessità dell'inserimento

Come abbiamo visto, l'inserimento in un Red-Black Tree richiede una prima operazione di ricerca di $O(\log n)$ per trovare la posizione corretta del nuovo nodo, la creazione di un nuovo nodo in $O(1)$, e infine possono essere necessarie al massimo $O(\log n)$ ricolorazioni (ciascuna impiega $O(1)$) e al più una sola ristrutturazione ($O(1)$) per mantenere le proprietà dell'albero.

Pertanto, la complessità totale dell'inserimento in un Red-Black Tree è $O(\log n)$.

4.7 Delete

Per eliminare un nuovo nodo con chiave k in un Red-Black Tree, si segue lo stesso procedimento di un normale BST:

- Ciò vuol dire che eliminiamo sempre un nodo con al più un solo figlio. Il nodo eliminato contiene una chiave k o il suo predecessore/successore (in base all'implementazione) in ordine. Il nodo figlio di quello eliminato (se esiste) viene promosso a figlio del genitore del nodo eliminato.

4.7.1 Caso 1: Delete di un nodo rosso

Se il nodo eliminato è rosso, tutte le proprietà dei Red-Black Trees rimangono valide, poiché la rimozione di un nodo rosso non altera la black depth di alcun percorso dalla radice a una foglia, e poiché questa operazione non può introdurre un double red.

Nel corrispondente (2, 4)-Tree, la rimozione di un nodo rosso equivale alla rimozione di una chiave da un 3-nodo o da un 4-nodo, il che è sempre consentito senza ulteriori modifiche.

4.7.2 Caso 2: Delete di un nodo nero con un solo figlio (rosso)

Ricordiamo che stiamo trattando la rimozione di nodi con al più un figlio (per via della delete in un BST). Se il nodo da eliminare è nero e ha un figlio, questo sarà sicuramente rosso (altrimenti la black depth property non sarebbe soddisfatta e non si avrebbe un Red-Black Tree valido). In questo caso, possiamo semplicemente rimuovere il nodo nero e promuovere il figlio rosso al suo posto, colorandolo di nero, ristabilendo tutte le proprietà dei Red-Black Trees.

Nel corrispondente (2, 4)-Tree, questa operazione equivale alla rimozione del nodo nero da un 3-nodo.

Infine, consideriamo il caso più complesso, in cui il nodo da eliminare è un nodo nero senza figli.

4.7.3 Caso 3: Delete di un nodo nero senza figli

Il caso più complesso si verifica quando il nodo da eliminare è un nodo nero senza figli. Nel corrispondente (2, 4)-Tree, questa situazione equivale alla rimozione di una chiave da un 2-nodo. Senza un ribilanciamento, una modifica del genere comporta un deficit di uno per la black depth lungo il percorso che porta al nodo eliminato, violando così la Depth Property dei Red-Black Trees.

Per rimediare a questo scenario, consideriamo un contesto più generale con un nodo z che è noto per avere due sottoalberi, T_{heavy} e T_{light}, tale che la radice di T_{light} (se presente) è nera e tale che la black depth di T_{heavy} è esattamente uno in più rispetto a quella di T_{light}, come illustrato in Figura 4.3. Nel caso di una foglia nera rimossa, z è il genitore di quella foglia e T_{light} è banalmente il sottoalbero vuoto che rimane dopo la cancellazione. Descriviamo il caso più generale di un deficit perché il nostro algoritmo per il ribilanciamento dell'albero, in alcuni casi, spingerà il deficit più in alto nell'albero (proprio come la risoluzione di una cancellazione in un (2,4) tree a volte si propaga verso l'alto). Indichiamo con y la radice di T_{heavy} (Un tale nodo esiste perché T_{heavy} ha altezza nera almeno uno).

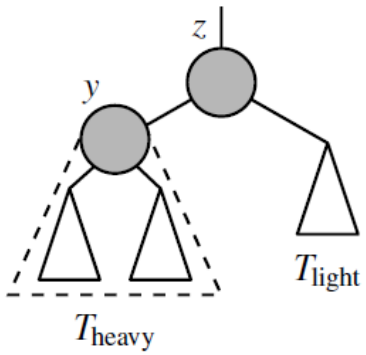


Figura 4.3: Illustrazione di un deficit tra le altezze nere dei sottoalberi del nodo z. Il colore grigio nell'illustrare y e z denota il fatto che questi nodi possono essere colorati sia di nero che di rosso.

Ricapitolando:

- z è un nodo con due sottoalberi T_{heavy} e T_{light}, tali che:
  - T_{heavy} ha altezza nera d.
  - T_{light} ha altezza nera $d - 1$.
- y è la radice di T_{heavy}, la quale esiste sempre.
- z e y possono essere sia rossi che neri.

Distinguiamo tre possibili casi:

- Nodo y nero con (almeno) un figlio rosso x.
- Nodo y nero e entrambi i figli di y sono neri (o None).
- Nodo y rosso.

Caso 3.1: Nodo y nero con (almeno) un figlio rosso x

Soluzione: Si esegue una *ristrutturazione* `restructure(x)` sui tre nodi x, il suo genitore y, e il nonno z, rinominati temporaneamente come a, b, e c in ordine di chiave. Sostituiamo z con il nodo etichettato b, rendendolo il genitore degli altri due. Coloriamo a e c di nero, e diamo a b il colore precedente di z.

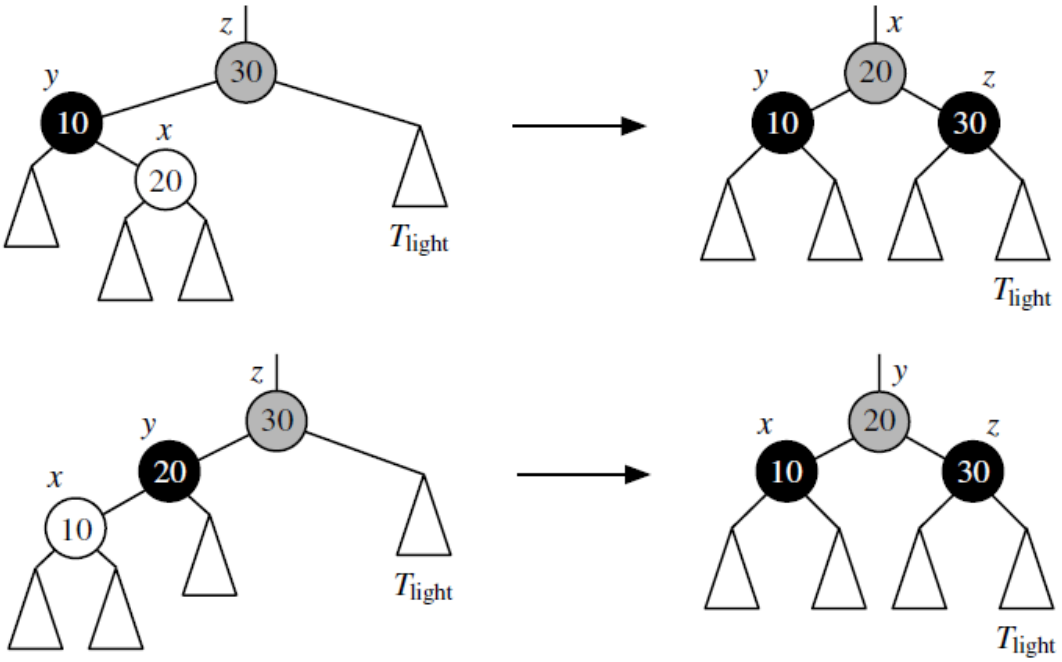


Figura 4.4: Risoluzione di un deficit nero in T_{light} attraverso una ristrutturazione su tre nodi `restructure(x)`. Sono mostrate due possibili configurazioni (le altre due sono simmetriche). Il colore grigio di z nelle figure a sinistra denota il fatto che questo nodo può essere colorato sia di rosso che di nero. La radice della porzione ristrutturata assume lo stesso colore, mentre i figli di quel nodo sono entrambi colorati di nero nel risultato.

Nel caso in cui y abbia entrambi i figli rossi, possiamo scegliere arbitrariamente uno dei due come x. Altrimenti, scegliamo l'unico figlio rosso di y come x. Da notare che il percorso verso T_{light} include un nodo nero aggiuntivo dopo la ristrutturazione, risolvendo così il suo deficit. Al contrario, il numero di nodi neri sui percorsi verso ciascuno degli altri tre sottoalberi illustrati in Figura 4.4 rimane invariato.

Risolvere questo caso corrisponde a un'operazione di *transfer* nell'albero (2,4) T' tra i due figli del nodo con z. Il fatto che y abbia un figlio rosso ci assicura che rappresenta o un 3-nodo o un 4-nodo. In effetti, l'elemento precedentemente memorizzato in z viene declassato per diventare un nuovo 2-nodo per risolvere la carenza, mentre un elemento memorizzato in y o nel suo figlio viene promosso per prendere il posto dell'elemento precedentemente memorizzato in z.

Caso 3.2: Nodo y nero e entrambi i figli di y sono neri (o None)

Soluzione: Si esegue una *ricolorazione*, per cui coloriamo y di rosso e, se z è rosso, lo coloriamo di nero.

- Se z era originariamente rosso, questa ricolorazione risolve il deficit.
- Se z era originariamente nero, la ricolorazione non risolve il deficit, ma lo propaga più in alto nell'albero; dobbiamo ripetere la considerazione di tutti e tre i casi sul genitore di z come rimedio.

Questa ricolorazione non introduce alcun double-red, poiché y non ha figli rossi.

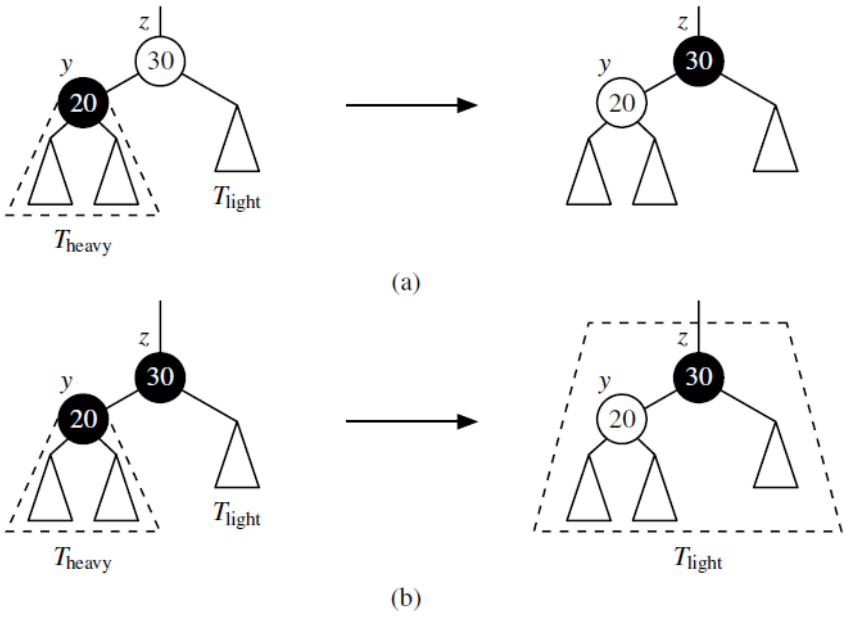


Figura 4.5: Risoluzione di un deficit nero in T_{light} tramite una ricolorazione. (a) quando z è originariamente rosso, si invertono i colori di y e z per risolvere il deficit nero in T_{light}, terminando il processo; (b) quando z è originariamente nero, la ricolorazione di y causa un deficit nero nell'intero sottoalbero di z, trasportando il problema ad un livello superiore che andrà risolto seguendo uno dei tre casi descritti.

La soluzione in questo caso corrisponde all'operazione di *fusion* nell'albero (2, 4) T', poiché y deve rappresentare un 2-nodo. Nel caso in cui z era originariamente rosso, e quindi il genitore nel corrispondente albero (2,4) è un 3-nodo o un 4-nodo, questa ricolorazione risolve il deficit. (Vedi Figura 4.5a.) Il percorso che porta a T_{light} include un nodo nero aggiuntivo nel risultato, mentre la ricolorazione non ha influenzato il numero di nodi neri sul percorso verso i sottoalberi di T_{heavy}. Nel caso in cui z fosse originariamente nero, e quindi il genitore nel corrispondente albero (2, 4) è un 2-nodo, la ricolorazione non ha aumentato il numero di nodi neri sul percorso verso T_{light}; in effetti, ha ridotto il numero di nodi neri sul percorso verso T_{heavy}. (Vedi Figura 4.5b.) Dopo questo passaggio, i due figli di z avranno la stessa altezza nera. Tuttavia, l'intero albero radicato in z è diventato carente, propagando così il problema più in alto nell'albero; dobbiamo ripetere la considerazione di tutti e tre i casi sul genitore di z come rimedio.

Caso 3.3: Nodo y rosso

Soluzione: Si esegue una *rotazione* su y e z, seguita da una ricolorazione di y in nero e di z in rosso. Inoltre, poichè y era originariamente rosso, il nuovo sottoalbero di z deve avere una radice nera y' e deve avere un'altezza nera uguale a quella originale di T_{heavy}. Pertanto, un deficit nero rimane nel nodo z dopo la trasformazione, e quindi riapplichiamo l'algoritmo per risolvere il deficit in z, sapendo che il nuovo figlio y', che è la radice di T_{heavy} è ora nero, e quindi che si applica o il Caso 3.1 o il Caso 3.2. Inoltre, la prossima applicazione sarà l'ultima, perché il Caso 3.1 è sempre terminale e il Caso 3.2 sarà terminale dato che z è rosso.

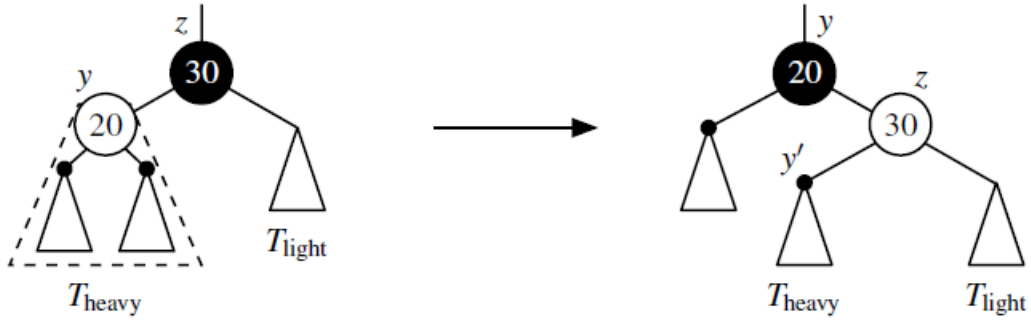


Figura 4.6: Una rotazione e una ricolorazione su un nodo rosso y e un nodo nero z, assumendo un deficit nero in z. Questo equivale a un cambiamento di orientamento nel corrispondente 3-nodo di un albero (2,4). Questa operazione non influisce sull'altezza nera di alcun percorso attraverso questa porzione dell'albero. Inoltre, poiché y era originariamente rosso, il nuovo sottoalbero di z deve avere una radice nera y' e deve avere un'altezza nera uguale a quella originale di T_{heavy}. Pertanto, un deficit nero rimane nel nodo z dopo la trasformazione.

Da notare che inizialmente y è rosso e T_{heavy} ha altezza nera almeno 1, z deve essere nero e i due sottoalberi di y devono avere ciascuno una radice nera e un'altezza nera uguale a quella di T_{heavy}.

Le prime operazioni di rotazione e ricolorazione denotano una riorientazione di un 3-nodo nel corrispondente albero (2,4) T'.

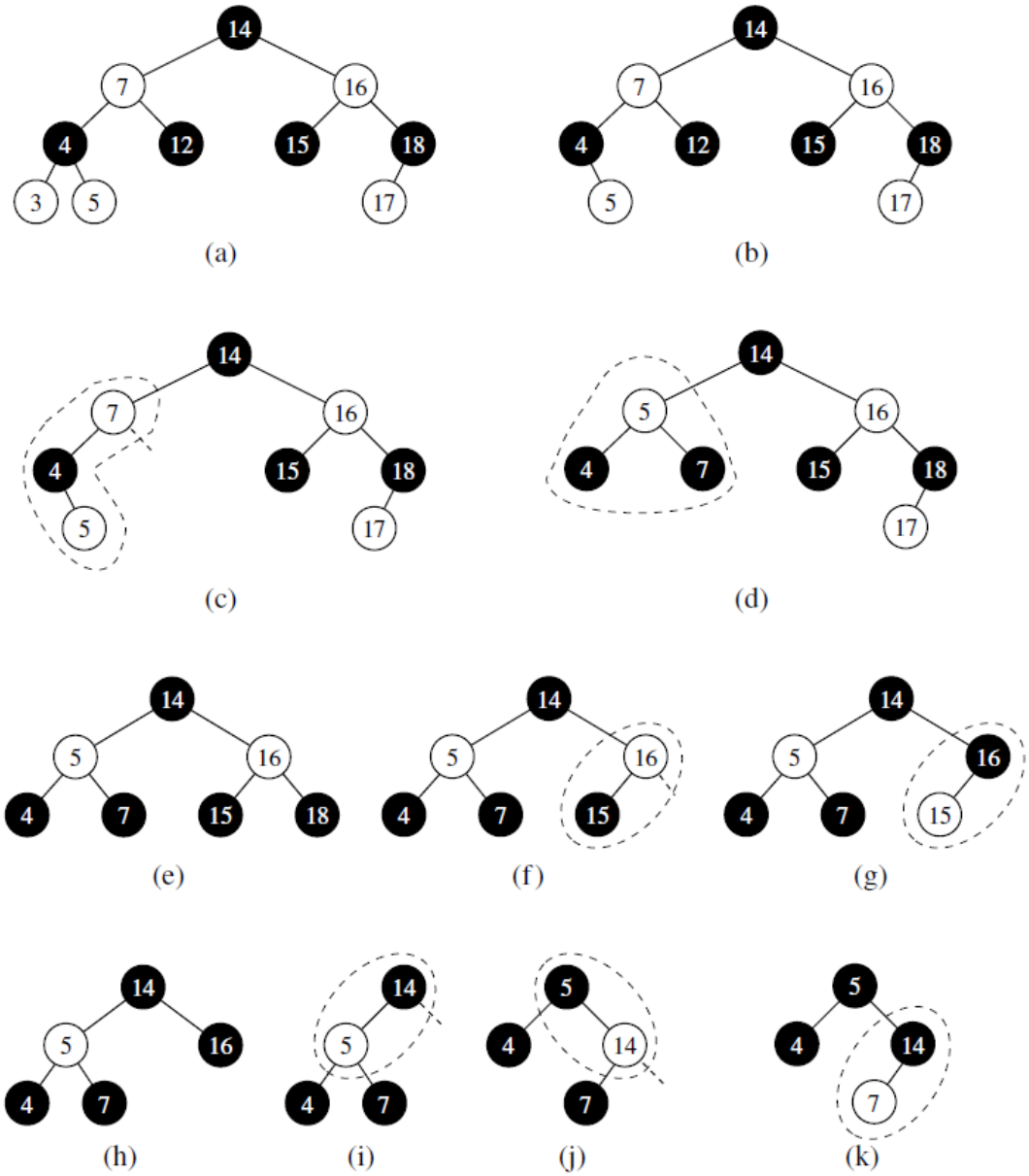


Figura 4.7: Una sequenza di cancellazioni da un Red-Black Tree: (a) albero iniziale; (b) rimozione di 3; (c) rimozione di 12, che causa un deficit nero a destra di 7 (risolto tramite ristrutturazione); (d) dopo la ristrutturazione; (e) rimozione di 17; (f) rimozione di 18, che causa un deficit nero a destra di 16 (risolto tramite ricolorazione); (g) dopo la ricolorazione; (h) rimozione di 15; (i) rimozione di 16, che causa un deficit nero a destra di 14 (risolto inizialmente tramite una rotazione); (j) dopo la rotazione il deficit nero deve essere risolto tramite una ricolorazione; (k) dopo la ricolorazione.

4.7.4 Complessità della cancellazione

L'algoritmo per eliminare un elemento da un Red-Black Tree con n elementi richiede $O(\log n)$ tempo e esegue $O(\log n)$ ricolorazioni e al massimo due operazioni di ristrutturazione.

Riepilogo Delete

Insert	Solving a double red	
RB-Tree	(2, 4)-Tree	result
Restructuring	Change the representation of the 4-node	Double red solved
Recoloring	Split	Double red solved or propagated

Delete	Rebalancing the BD	
RB-Tree	(2, 4)-Tree	result
Restructuring	transfer	BD balanced
Recoloring	fusion	BD balanced or unbalancing propagated
Rotation	Change the representation of the 3-node	Needs a recoloring or a restructuring

Figura 4.8: Riepilogo dei casi di delete in un Red-Black Tree.

Capitolo 5

Hash Tables

Come visto finora, le possibili implementazioni dell'*ADT Map* sono molteplici, da una semplice lista ordinata o non ordinata, a strutture dati più complesse come tutte le tipologie di alberi presentate nei capitoli precedenti. Tuttavia, queste implementazioni hanno tutte in comune il fatto che le operazioni di ricerca, inserimento e cancellazione hanno una complessità temporale di almeno $O(\log n)$ nel caso migliore (per alberi bilanciati) e $O(n)$ nel caso peggiore (per semplici liste). Ciascuna implementazione ha i suoi pro e contro, e la scelta di quale utilizzare dipende da vari fattori, tra cui la frequenza delle operazioni di inserimento, cancellazione e ricerca, la necessità di ordinare le chiavi, ecc. In questo capitolo introduciamo una delle strutture dati più popolari per l'implementazione di una mappa, e quella utilizzata dalla stessa implementazione di Python per la classe `dict`. Questa struttura è nota come **hash table** (tabella hash).

Intuitivamente, una mappa M supporta l'astrazione di utilizzare le chiavi come indici con una sintassi del tipo $M[k]$. Consideriamo inizialmente un contesto ristretto in cui una mappa con n elementi utilizza chiavi che si presuppone essere interi in un intervallo da 0 a $N-1$ per qualche $N \geq n$. In questo caso, possiamo rappresentare la mappa utilizzando una *lookup table* di lunghezza N, come illustrato in Figura 5.1.

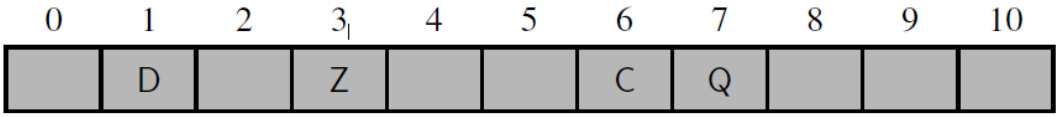


Figura 5.1: Una lookup table di lunghezza $N = 11$ per una mappa contenente $n = 4$ elementi: (1,D), (3,Z), (6,C), (7,Q).

Ci sono due aspetti che vogliamo estendere rispetto a questa semplice rappresentazione per arrivare alla definizione di *hash table*:

- Come prima cosa, potremmo non voler utilizzare un array di lunghezza N nel caso in cui $N \gg n$.
- In secondo luogo, non è necessario che le chiavi di una mappa siano interi.

Una tabella hash, per un dato tipo di chiave, consiste in:

- Un array di bucket di lunghezza N.
- Una funzione di hash h.
- Un metodo per gestire le collisioni all'interno di ciascun bucket.

5.1 Hash Functions

Una Hash Table utilizza una **funzione di hash** per mappare una chiave generica in un indice della tabella. Idealmente, le chiavi saranno ben distribuite nell'intervallo da 0 a $N-1$ dalla funzione di hash, ma in pratica potrebbero esserci due o più chiavi distinte che vengono mappate allo stesso indice. Di conseguenza, concettualizzeremo la nostra tabella come un array di bucket, come mostrato in Figura 5.2, in cui ogni bucket può gestire una collezione di elementi che vengono inviati a uno specifico indice dalla funzione di hash.

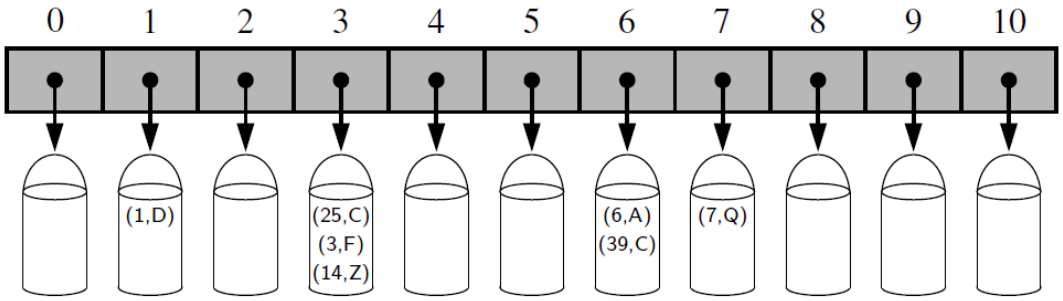


Figura 5.2: Un bucket array di lunghezza $N = 11$. Ogni bucket può contenere zero, uno, o più elementi.

Formalmente, una funzione di hash è una funzione $h : K \to \{0, 1, \ldots, N-1\}$ che mappa ogni chiave $k \in K$ in un indice della tabella. La scelta di una buona funzione di hash è cruciale per le prestazioni della hash table, poiché una cattiva distribuzione delle chiavi può portare a un numero elevato di collisioni, ovvero situazioni in cui più chiavi vengono mappate allo stesso indice, quindi allo stesso bucket.

L'idea principale è quella di usare la funzione di hash h per determinare il valore $h(k)$, che useremo come indice nel bucket array, A, al posto del valore di chiave k stesso (come abbiamo detto k non è necessariamente un intero, per cui potrebbe essere poco adatto come indice). Quindi, memorizziamo l'elemento formato dalla coppia chiave-valore (k, v) nel bucket $A[h(k)]$.

Se ci sono due o più chiavi con lo stesso valore di hash, allora due elementi diversi verranno mappati allo stesso bucket in A. In questo caso, diciamo che si è verificata una **collisione**. Ci sono modi per gestire le collisioni, di cui discuteremo più avanti, ma la strategia migliore è cercare di evitarle in primo luogo. Diciamo che una funzione di hash è "buona" se mappa le chiavi nella nostra mappa in modo da minimizzare sufficientemente le collisioni.

È comune pensare ad una funzione di hash come la composizione di due diverse funzioni:

- Hash code: una funzione $h_1 : K \to \mathbb{Z}$ che mappa una chiave k in un intero (positivo o negativo).
- Compression function: una funzione $h_2 : \mathbb{Z} \to \{0, 1, \ldots, N-1\}$ che mappa l'intero risultante in un indice della tabella hash.

$$h(x) = h_2(h_1(x))$$

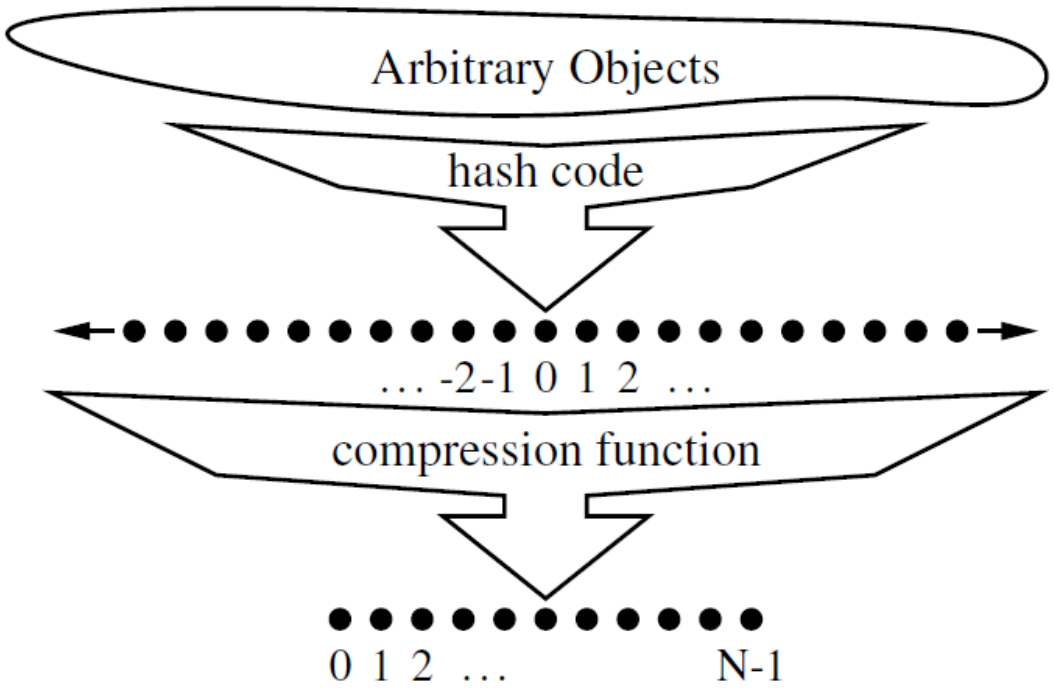


Figura 5.3: Le due parti di una funzione hash: hash code e funzione di compressione.

La separazione di queste due funzioni consente l'utilizzo di hash codes che sono validi per uno specifico insieme di chiavi, indipendentemente dalla dimensione della tabella hash (la cui dimensione può variare dinamicamente nel tempo).

5.1.1 Esempi di Hash Code

Come abbiamo detto la funzione di hash code mappa una generica chiave in un intero. Una *collisione* si verifica quando due chiavi distinte producono lo stesso hash code, per cui la funzione di compressione non può fare nulla per evitarla e memorizzerà entrambe le chiavi nello stesso bucket. Di conseguenza, è importante scegliere una buona funzione di hash code che minimizzi le collisioni per il tipo di chiavi che ci aspettiamo di utilizzare nella nostra mappa.

Presentiamo alcuni tra i metodi più comuni per il calcolo dell'hash code:

- **Memory address**: Si utilizza l'indirizzo di memoria (un intero) della chiave come hash code.
- **Rappresentazione in bit**: Si interpreta la rappresentazione in bit della chiave come un intero.
- **Composition**: Si partizionano i bit della chiave in gruppi di dimensione fissa, si sommano ignorando l'overflow (XOR).
- **Polinomial accumulation**: A differenza dei metodi precedenti che funzionano bene per chiavi numeriche, questo metodo è adatto per chiavi di tipo stringa, o più in generale per oggetti di lunghezza variabile in cui l'ordine degli elementi è importante. Si partizionano i bit della chiave in gruppi di dimensione fissa $(a_0, a_1, \ldots, a_{k-1})$ e si calcola il polinomio $p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{k-1} z^{k-1}$ per un opportuno valore di z. L'hash code è quindi il valore di $p(z)$ calcolato in un intero di dimensione fissa, ignorando l'overflow.

5.1.2 Esempi di Funzione di Compressione

La funzione di compressione mappa un intero (l'hash code) in un indice della tabella hash, ovvero in un intero nell'intervallo $[0, N-1]$, dove N è la dimensione della tabella hash. Una buona funzione di compressione deve riuscire a minimizzare il numero di collisioni a partire da hash codes differenti (poiché se due hash codes sono uguali, non c'è modo di evitare la collisione).

- **Modulo**:
  - $h_2(y) = y \mod N$
  - Questa funzione può portare a molte collisioni se N ha fattori comuni con gli hash codes, per cui N è tipicamente un numero primo.
- **MAD (Multiply-Add-Divide)**:
  - $h_2(y) = [(ay + b) \mod p] \mod N$
  - $p > N$ è un numero primo, e a e b sono interi scelti casualmente in $[0, p-1]$ con $a > 0$.
  - La probabilità che due hash codes distinti y_1 e y_2 causino una collisione è al più $1/N$.

5.2 Gestione delle Collisioni

L'esistenza delle collisioni ci impedisce di memorizzare semplicemente un elemento (k, v) nel bucket $A[h(k)]$. Inoltre complica in modo significativo le operazioni di ricerca, inserimento e cancellazione. In questa sezione presentiamo due approcci comuni per gestire le collisioni: *separate chaining* e *open addressing*.

5.2.1 Separate Chaining

Una soluzione semplice ed efficiente per gestire le collisioni è quella di far sì che ogni bucket $A[j]$ memorizzi un riferimento ad una struttura dati secondaria, che contiene gli elementi (k, v) tali che $h(k) = j$. Una scelta naturale per la struttura dati secondaria è una semplice linked list. Questa regola di risoluzione delle collisioni è nota come **separate chaining**, ed è illustrata in Figura 5.4.

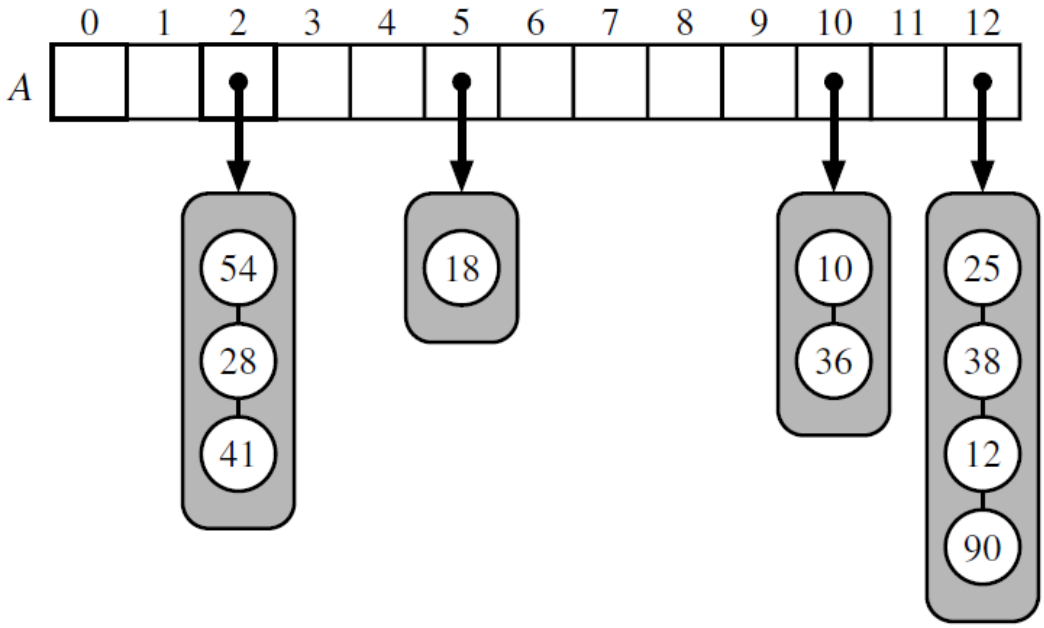


Figura 5.4: Hash Table di dimensione 13, che memorizza 10 elementi con chiavi intere, con collisioni risolte tramite separate chaining. La funzione di compressione è $h(k) = k \mod 13$. Per semplicità, non mostriamo i valori associati alle chiavi.

Nel caso peggiore, le operazioni su di uno specifico bucket impiegano tempo proporzionale alla dimensione del bucket stesso. Assumendo di utilizzare una buona funzione di hash per indicizzare gli n elementi della nostra mappa in un bucket array di capacità N, la dimensione attesa di ciascun bucket è n/N. Dunque, con una buona funzione di hash, le operazioni principali sulla mappa richiedono un tempo atteso di $O(n/N)$. Il rapporto $\lambda = n/N$, chiamato **load factor** (fattore di carico) della tabella hash, dovrebbe essere limitato da una piccola costante, preferibilmente inferiore a 1. Finché λ è $O(1)$, le operazioni principali sulla tabella hash vengono eseguite in tempo atteso $O(1)$.

5.2.2 Open Addressing

La tecniche di *separate chaining* è effiente e facile da implementare, ma richiede inevitabilmente spazio aggiuntivo per le strutture dati ausiliarie. Se si vuole evitare di utilizzare spazio aggiuntivo per gestire le collisioni, l'unico approccio possibile è quello di cercare uno slot alternativo, sempre all'interno del nostro array, in cui memorizzare l'elemento che causa la collisione.

Ci sono diverse varianti che utilizzano questo approccio, noti come tecniche di **open addressing**. Queste tecniche richiedono che il load factor $\lambda = n/N$ sia sempre inferiore a 1, ovvero che $n < N$, poiché ogni elemento deve essere memorizzato in uno slot dell'array.

Linear Probing

Un metodo semplice per gestire le collisioni con l'open addressing è il *linear probing*. Con questo approccio, se si tenta di inserire un elemento (k, v) in un bucket già occupato, si cerca il successivo bucket libero nell'array, procedendo in modo circolare, "sondando" linearmente uno slot per volta. Da qui il nome di *linear probing* (probe = sondare).

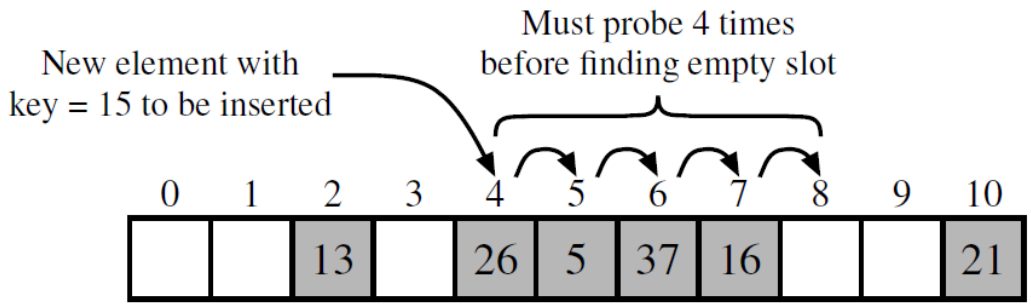


Figura 5.5: Inserimento di elementi con chiavi intere utilizzando linear probing per gestire le collisioni. La funzione di compressione è $h(k) = k \mod 11$. Per semplicità, non mostriamo i valori associati alle chiavi.

Un problema può sorgere quando, facendo riferimento alla Figura 5.5, si elimina l'elemento con chiave 37 e successivamente si tenta di ricercare l'elemento con chiave 15. Poichè l'elemento con chiave 15 è stato memorizzato, a seguito di una collisione, dopo l'elemento 37, la ricerca di 15 fallirà se ci si ferma al primo slot vuoto incontrato (quello lasciato libero da 37).

Per risolvere questo problema, quando si elimina un elemento in una tabella hash che utilizza linear probing, si può marcare lo slot come "AVAILABLE" invece di lasciarlo vuoto. In questo modo, durante la ricerca, si continuerà a sondare gli slot successivi anche se si incontra uno slot marcato, garantendo così che tutti gli elementi possano essere trovati correttamente, mentre durante l'inserimento, gli slot marcati possono essere riutilizzati per nuovi elementi. Di fatto, uno slot marcato come "AVAILABLE" differenzia uno slot precedentemente occupato da uno slot mai usato prima, e viene trattato come occupato durante la ricerca, ma come vuoto durante l'inserimento.

Il problema del Clustering causato da Linear Probing

Un problema noto con il linear probing è il fenomeno del **clustering lineare**. Quando si verifica una collisione e si utilizza il linear probing per trovare uno slot libero, gli elementi tendono a raggrupparsi in cluster contigui all'interno dell'array. Questi cluster possono crescere nel tempo, aumentando la probabilità di ulteriori collisioni e rallentando le operazioni di ricerca, inserimento e cancellazione. Questo perché, quando si cerca uno slot libero, si potrebbe dover sondare attraverso un intero cluster, aumentando il tempo necessario per completare l'operazione, specialmente quando il load factor λ inizia ad essere maggiore di $1/2$.

Esistono altre tecniche di open addressing che cercano di mitigare il problema del clustering lineare:

- **Quadratic Probing**: Evita il clustering lineare utilizzando una funzione di probing quadratica per trovare lo slot successivo, ma crea cluster di forma più complessa (clustering secondario). Funziona bene solo se il load factor λ è mantenuto al di sotto di $1/2$.
- **Double Hashing**: Si utilizza una seconda funzione di hash per calcolare l'offset in caso di collisione. Questo metodo riduce significativamente il clustering e offre prestazioni migliori rispetto al linear e quadratic probing, specialmente a load factor più elevati.

Double Hashing

La tecnica del *double hashing* utilizza una seconda funzione di hash $d(k)$ per ricalcolare l'offset in caso di collisione. Invece di sondare linearmente o quadraticamente, si calcola un offset basato sulla chiave stessa, il che aiuta a distribuire gli elementi in modo più uniforme nell'array e riduce il clustering.

La procedura di inserimento con double hashing funziona come segue:

- $(h(k) + j \cdot d(k)) \mod N$, per $j = 0, 1, \ldots, N-1$
- Si tenta di inserire l'elemento nel bucket calcolato. Se il bucket è occupato, si incrementa j e si ricalcola l'indice fino a trovare uno slot libero.
- Se si esauriscono tutti gli N tentativi senza trovare uno slot libero, la tabella è piena e l'inserimento fallisce.
- La funzione di offset $d(k)$ non può restituire zero.
- N dovrebbe essere un numero primo per garantire che tutti gli slot vengano sondati.
- Una scelta comune per $d(k)$ è $d(k) = q - (h_1(k) \mod q)$, dove q è un numero primo più piccolo di N.

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

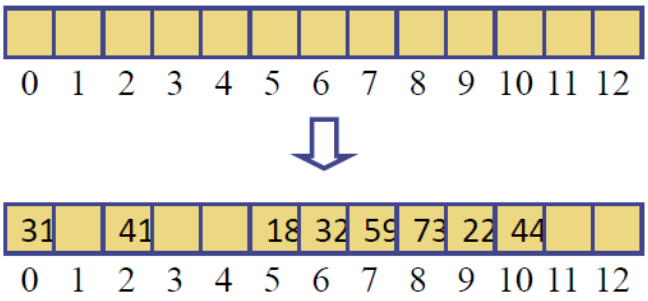


Figura 5.6: Inserimento di elementi con chiavi intere: 18, 41, 22, 59, 32, 31, 73. $N = 13$, $h(k) = k \mod 13$, $d(k) = 7 - (k \mod 7)$. Per semplicità, non mostriamo i valori associati alle chiavi.

5.3 Load Factor

Abbiamo visto come il **load factor** (fattore di carico) di una tabella hash è definito come il rapporto tra il numero di elementi memorizzati nella tabella n e la capacità totale della tabella N, $\lambda = n/N$. Questo parametro condizione fortemente le performance di una tabella hash. Secondo la letteratura, per mantenere prestazioni ottimali, il load factor dovrebbe essere mantenuto sempre al di sotto di una certa soglia, in base al tipo di gestione delle collisioni utilizzata:

- *separate chaining*: $\lambda \leq 0.9$.
- *open addressing*: $\lambda \leq 0.5$ (per linear probing e quadratic probing), $\lambda \leq 0.7$ (per double hashing).

5.4 Analisi delle prestazioni

- **Worst case**: Il caso peggiore si ha quando tutte le chiavi collidono, per cui si ha un tempo $O(n)$ per tutte le operazioni principali (ricerca, inserimento, cancellazione). Se i valori di hash sono distribuiti uniformemente, il numero di probe necessarie per trovare uno slot libero è in media $1/(1-\lambda)$.
- **Average case**: Il tempo medio per tutte le operazioni principali è $O(1)$. In pratica, con una buona funzione di hash, una hash table è estremamente efficiente se il load factor è molto minore di 1.

Capitolo 6

Priority Queues

Una **coda con priorità** (*Priority Queue*) è una struttura dati astratta fondamentale, simile a una coda standard (*Queue*), ma con una differenza cruciale nel criterio di estrazione. Mentre una coda standard opera secondo la logica **FIFO** (First-In, First-Out), rimuovendo l'elemento che è in attesa da più tempo, una coda a priorità rimuove sempre l'elemento con la **priorità** più alta (o più bassa, a seconda di quale logica si vuole utilizzare), indipendentemente dall'ordine di inserimento.

È importante non confondere una coda a priorità con una mappa (Map) o dizionario. Lo scopo di una **Map** è l'associazione e la ricerca rapida: memorizza coppie $\langle Chiave, Valore \rangle$ e risponde efficientemente alla domanda: "Qual è il valore associato a questa specifica chiave?". Al contrario, lo scopo di una **Priority Queue** è l'estrazione efficiente dell'elemento più importante. Sebbene gli elementi in una coda a priorità siano spesso implementati come coppie, ad esempio $\langle Priorità, Dato \rangle$ (a cui spesso ci si riferisce comunque come (chiave, valore)), questa coppia ha una funzione diversa: la $Priorità$ non serve per cercare il $Dato$ (come farebbe una chiave in una mappa), ma serve solo alla struttura interna della coda (spesso uno *Heap*) per determinare l'ordine di estrazione. In sintesi, si usa una mappa per *trovare* un elemento tramite un identificatore unico (la chiave), mentre si usa una coda a priorità per *estrarre* l'elemento con il grado di urgenza massimo o minimo.

Definizione: Una **Priority Queue** è una raccolta di elementi con priorità che consente l'inserimento arbitrario di elementi e la rimozione dell'elemento con priorità primaria. Quando un elemento viene aggiunto in una Priority Queue questo assume una determinata priorità rappresentata dalla chiave ad esso associata. L'elemento con la chiave minima sarà il successivo a essere rimosso dalla coda (quindi, a un elemento con chiave 1 verrà data priorità su un elemento con chiave 2).

6.1 Chiavi (Priorità)

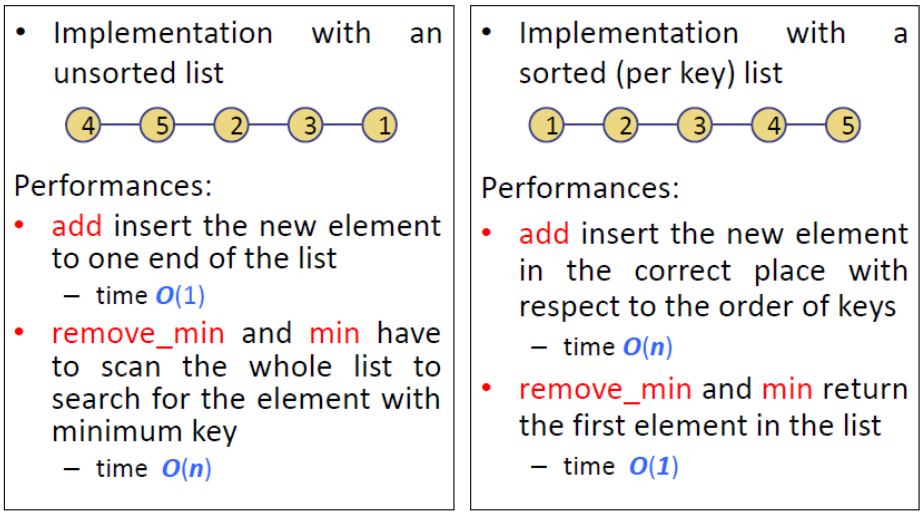
In una Priority Queue, ogni elemento è associato a una chiave che determina la sua priorità. Le chiavi in una Priority Queue possono essere oggetti di qualunque tipo tale che sia possibile

definire un ordinamento totale su di esse. Con tale generalità, le applicazioni possono sviluppare la propria nozione di priorità per ciascun elemento.

A marcare la differenza con la struttura dati Map, in una Priority Queue **le chiavi non sono necessariamente uniche**. È possibile avere più elementi con la stessa chiave (priorità). In questi casi, la politica di estrazione per gli elementi con chiavi identiche può variare a seconda dell'implementazione specifica della Priority Queue. Alcune implementazioni potrebbero adottare una politica FIFO per gli elementi con la stessa priorità, mentre altre potrebbero non garantire alcun ordine specifico tra di essi.

6.2 Possibili implementazioni di Priority Queue

Esistono diverse strategie per implementare una Priority Queue. Due approcci semplici utilizzano liste (array o liste collegate) per memorizzare gli elementi, con differenze significative nelle prestazioni delle operazioni di inserimento e rimozione.



Le due strategie per implementare una Priority Queue ADT dimostrano un interessante compromesso. Quando si utilizza una lista non ordinata per memorizzare gli elementi, possiamo eseguire inserimenti in tempo O(1), ma trovare o rimuovere un elemento con chiave minima richiede un ciclo in tempo O(n) attraverso l'intera collezione. Al contrario, se si utilizza una lista ordinata, possiamo banalmente trovare o rimuovere l'elemento minimo in tempo O(1), ma aggiungere un nuovo elemento alla coda potrebbe richiedere tempo O(n) per ripristinare l'ordinamento.

Esistono però implementazioni più efficienti, seppur più complesse, che consentono di eseguire sia inserimenti che rimozioni in tempo logaritmico. Una di queste implementazioni si basa su una struttura dati chiamata **heap binario**, la quale utilizza la struttura di un albero binario per trovare un compromesso tra elementi completamente non ordinati e perfettamente ordinati.

6.3 Heap

Definizione: Un **Heap** è un albero binario che soddisfa le seguenti proprietà:

- **Albero binario completo**: L'albero è un albero binario completo (vedi pag.6), cioè tutti i livelli dell'albero sono completamente riempiti, tranne eventualmente l'ultimo livello, che è riempito da sinistra a destra.
  - Sia h l'altezza dell'albero, per $i = 0, 1, \ldots, h-1$, il livello i-esimo contiene esattamente 2^i nodi. Se il livello $h-1$ (cioè nel livello più basso) non è pieno, tutti i suoi nodi sono riempiti da sinistra a destra.
- **Heap-Order**: Per ogni posizione (nodo) p, la chiave di p è minore o uguale alle chiavi dei suoi figli.
  - Questo implica che la chiave minima si trova sempre nella radice dell'albero.

Dal momento che un Heap è un albero binario completo, la sua altezza è sempre logaritmica rispetto al numero di nodi nell'albero. Questo fatto è cruciale per garantire che le operazioni di inserimento e rimozione possano essere eseguite in tempo $O(\log n)$. Si ha quindi:

$$h = \lfloor \log_2(n) \rfloor$$

6.4 Implementazione di una Priority Queue attraverso Heap

Un Heap può essere efficacemente utilizzato per implementare una Priority Queue, sfruttando le sue proprietà di albero binario completo e heap-order. Le operazioni principali di una Priority Queue, ovvero l'inserimento di un elemento e la rimozione dell'elemento con priorità minima, possono essere eseguite in tempo logaritmico grazie alla struttura dell'Heap.

- In ogni nodo dello Heap viene memorizzata una coppia chiave-valore, dove la chiave rappresenta la priorità dell'elemento.
  - N.B.: è possibile avere più nodi con la stessa chiave, in quanto si tratta di un'implementazione di una Priority Queue.
- Manteniamo un riferimento all'ultimo nodo dello Heap, cioè il nodo più a destra sull'ultimo livello dell'albero.

Inserimento `add(k,v)`

Si vuole inserire una nuova coppia chiave-valore (k, v) nell'Heap. Per mantenere la proprietà di albero binario completo, dobbiamo inserire il nuovo nodo nella posizione corretta, ovvero appena oltre il nodo più a destra al livello più basso dell'albero, o come posizione più a sinistra di un nuovo livello, se il livello più basso è già pieno (o se l'heap è vuoto).

La posizione del nodo nel quale inserire il nuovo elemento può essere trovato in un tempo $O(\log n)$ a partire dal riferimento all'ultimo nodo:

- Si parte dal riferimento all'ultimo nodo.
- Si risale l'albero fino alla radice o ad un nodo che è figlio sinistro del suo genitore.
- Se il nodo in cui si è giunti è il figlio sinistro del suo genitore, vai al nodo fratello.
- Infine, scendi sempre a sinistra fino a raggiungere un nodo foglia (None). Questa sarà la posizione in cui inserire il nuovo nodo.

Up-Heap Bubbling dopo l'inserimento

Dopo aver inserito il nuovo nodo nella posizione corretta al fine di mantenere la proprietà di albero binario completo, è possibile che la proprietà di heap-order venga violata, poiché la chiave del nuovo nodo potrebbe essere minore della chiave del suo genitore. Per ripristinare la proprietà di heap-order, si esegue un processo chiamato **up-heap bubbling**.

- Partendo dal nuovo nodo con chiave k, e risalendo l'albero verso la radice, si scambia il nodo corrente con il suo genitore fino a quando la heap-order property non è ristabilita.
- L'algoritmo termina quando la chiave k raggiunge la radice dell'albero o quando la chiave k si trova in un nodo il cui padre ha una chiave minore o uguale a k.

Dal momento che l'altezza dell'Heap è $O(\log n)$, l'operazione di up-heap bubbling richiede un tempo $O(\log n)$ nel caso peggiore.

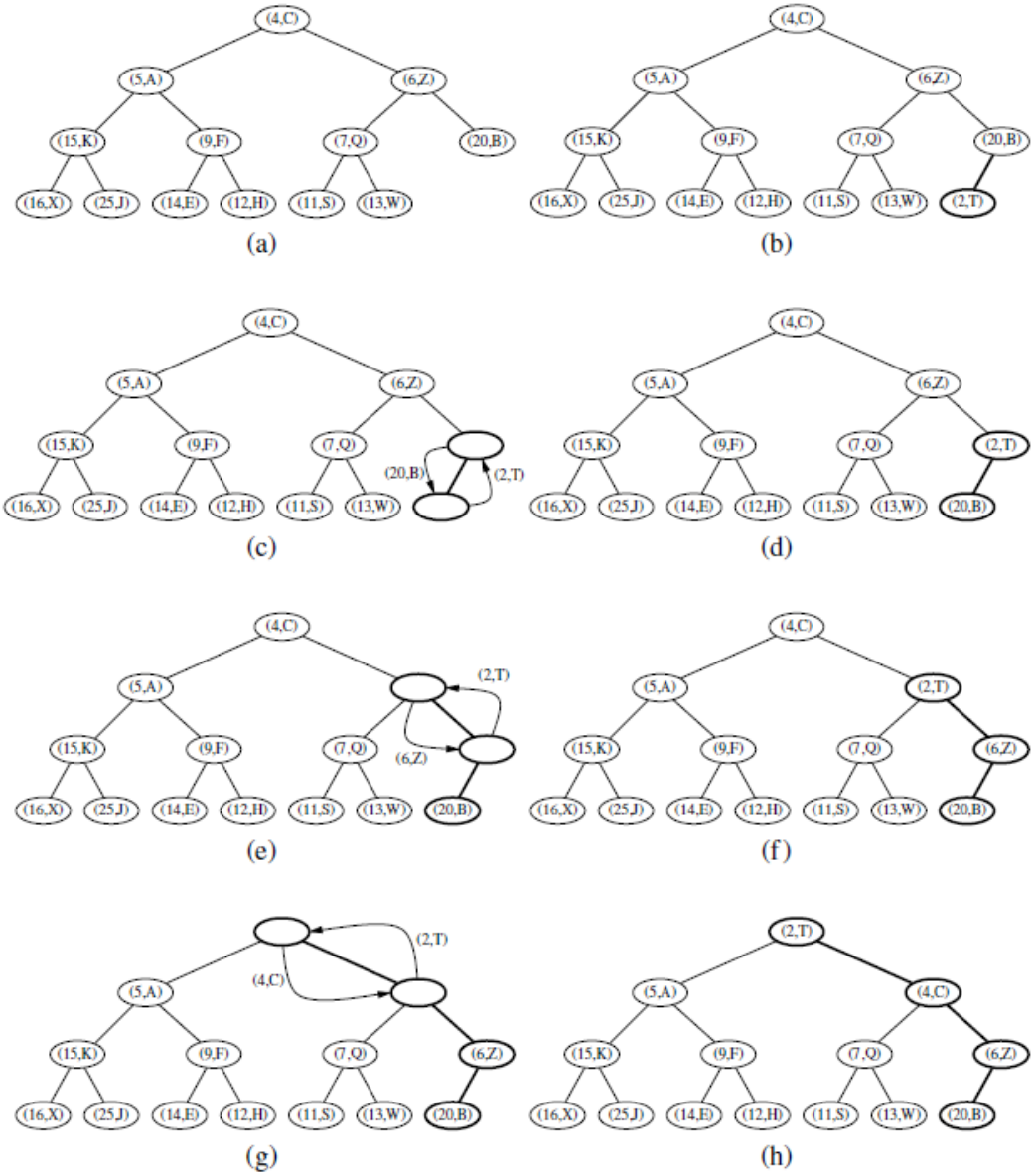


Figure 9.2: Insertion of a new entry with key 2 into the heap of Figure 9.1: (a) initial heap; (b) after performing operation add; (c and d) swap to locally restore the partial order property; (e and f) another swap; (g and h) final swap.

Cancellazione `remove_min()`

Il metodo `remove_min()` rimuove e restituisce l'elemento con la chiave minima dalla Priority Queue. In un Heap, l'elemento con la chiave minima si trova sempre nella radice dell'albero. Per mantenere la proprietà di albero binario completo dopo la rimozione della radice, dobbiamo sostituire la radice con l'ultimo nodo dell'Heap (cioè il nodo più a destra nell'ultimo livello dell'albero) e poi rimuovere l'ultimo nodo.

Down-Heap Bubbling dopo la cancellazione

Dopo aver effettuato la sostituzione della radice con l'ultimo nodo, è possibile che la proprietà di heap-order venga violata, poiché la chiave del nuovo nodo radice potrebbe essere maggiore della chiave di uno o entrambi i suoi figli. Per ripristinare la proprietà di heap-order, si esegue un processo chiamato **down-heap bubbling**.

- Partendo dalla radice, si scambia il nodo corrente con il figlio che ha la chiave minima, a condizione che la chiave del figlio sia minore della chiave del nodo corrente.
- Questo processo continua fino a quando la chiave del nodo corrente è minore o uguale alle chiavi dei suoi figli, o fino a quando il nodo corrente diventa un nodo foglia (None), per cui la heap-order property è ristabilita.

Dal momento che l'altezza dell'Heap è $O(\log n)$, l'operazione di down-heap bubbling richiede un tempo $O(\log n)$ nel caso peggiore.

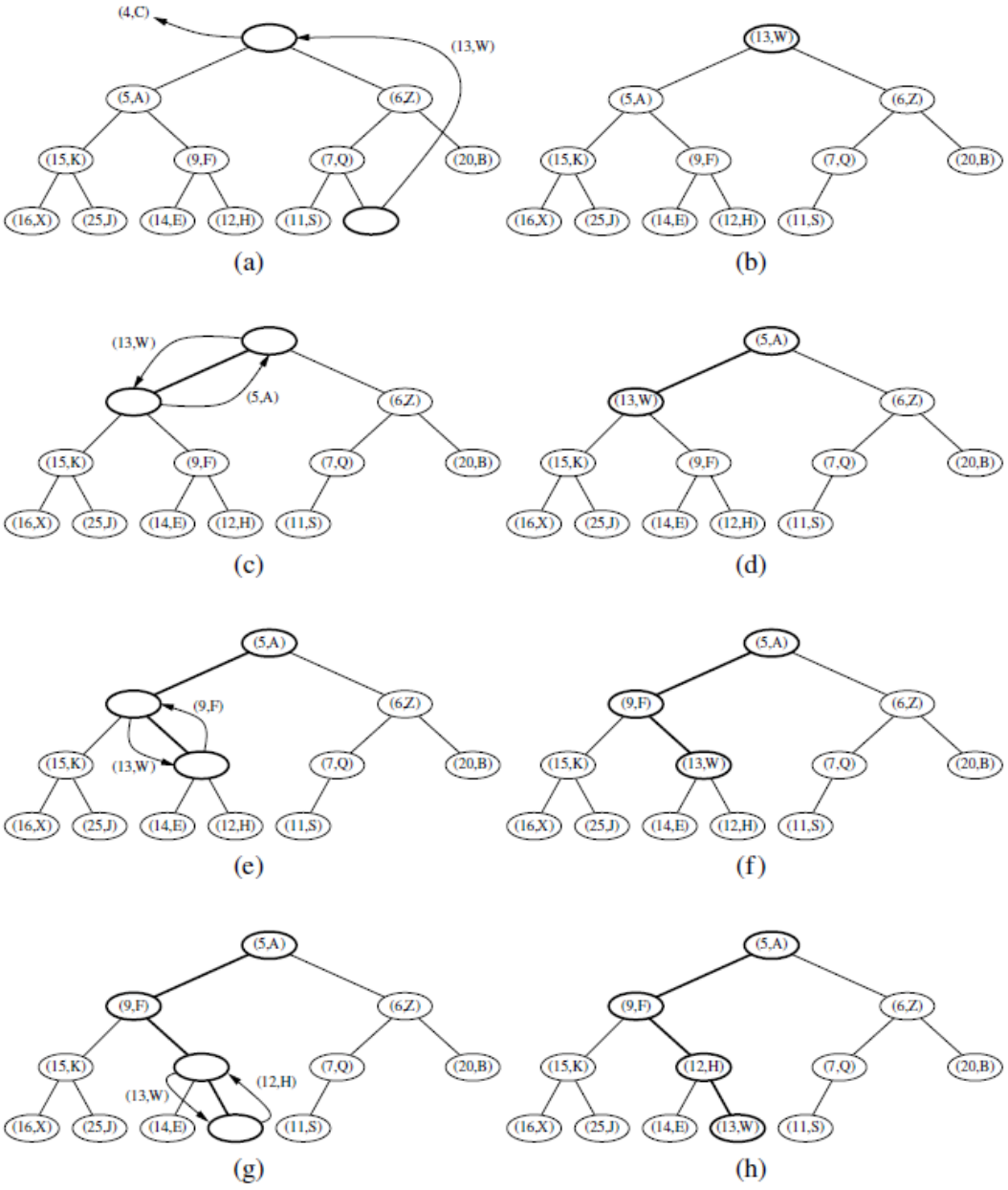


Figure 9.3: Removal of the entry with the smallest key from a heap: (a and b) deletion of the last node, whose entry gets stored into the root; (c and d) swap to locally restore the heap-order property; (e and f) another swap; (g and h) final swap.

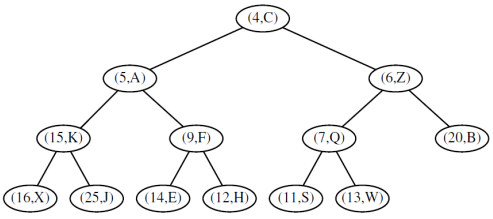
6.5 Implementazione Array-based di un Heap

È sempre possibile rappresentare un albero binario utilizzando un array, sfruttando la relazione tra gli indici dei nodi genitori e figli. In generale questa rappresentazione è meno efficiente in termini di spazio rispetto a una rappresentazione basata su nodi collegati, poiché richiede spazio per tutti i nodi, compresi quelli vuoti.

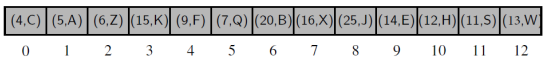
Tuttavia, per un albero binario *completo* come un Heap, questa rappresentazione è particolarmente efficiente, poiché non ci sono nodi vuoti tra i nodi effettivamente presenti nell'albero. Sia n il numero di nodi nell'albero, nel caso di un albero binario qualsiasi l'array può avere nel caso peggiore $N = 2^n - 1$ elementi, mentre nel caso di albero binario completo l'array avrà esattamente $N = n$ elementi.

Questo procedimento vale per ogni albero binario, ma è particolarmente efficiente per un Heap, poiché l'albero è completo. La mappatura tra i nodi dell'albero e gli indici dell'array avviene seguendo queste regole:

- Il nodo radice dell'albero viene memorizzato all'indice 0 dell'array.
- Per un nodo situato all'indice i nell'array:
  - Il figlio sinistro del nodo si trova all'indice $2i + 1$.
  - Il figlio destro del nodo si trova all'indice $2i + 2$.
  - Il genitore del nodo si trova all'indice $\lfloor (i - 1)/2 \rfloor$, a condizione che $i > 0$.



(a) Esempio di uno Heap con 13 elementi. L'ultima posizione è occupata dal nodo con chiave 13.



(b) Sua rappresentazione come array.

6.6 Analisi delle prestazioni

Operation	Running Time
len(P), P.is_empty( )	$O(1)$
P.min( )	$O(1)$
P.add( )	$O(\log n)^*$
P.remove_min( )	$O(\log n)^*$

*amortized, if array-based

In figura sono riassunte le complessità temporali delle operazioni principali di una Priority Queue implementata tramite un Heap binario, assumendo che due chiavi possano essere confrontate in tempo costante $O(1)$, e che l'Heap sia implementato come array-based o linked-based tree. L'analisi è basata sulle seguenti considerazioni:

- L'Heap ha n nodi, ognuno dei quali contiene una coppia chiave-valore.
- L'altezza dell'Heap è $O(\log n)$ [In particolare, $h = \lfloor \log_2 n \rfloor$], dal momento che si tratta di un albero binario completo.
- L'operazione `min()` viene eseguita in $O(1)$ perché la radice dell'albero contiene tale elemento.
- Nel caso peggiore, up-heap e down-heap eseguono un numero di scambi pari all'altezza dell'Heap.

Concludiamo che la struttura dati Heap è una realizzazione molto efficiente dell'ADT Priority Queue, indipendentemente dal fatto che l'heap sia implementato come array-based o linked-based tree. L'implementazione basata su heap raggiunge tempi di esecuzione rapidi sia per l'inserimento che per la rimozione, a differenza delle implementazioni basate sull'utilizzo di una sorted-list o unsorted-list.

6.7 Costruzione di un Heap da una lista di elementi: Heapify

A partire da un Heap vuoto, n inserimenti successivi al suo interno richiederebbero un tempo complessivo di $O(n \cdot \log n)$, poiché ogni inserimento richiede un tempo $O(\log n)$. Tuttavia, se le n coppie chiave-valore sono note a prescindere, esiste un algoritmo più efficiente chiamato **Heapify** che consente di costruire un Heap a partire da una lista di n elementi in tempo lineare $O(n)$.

Fusione di due Heap

Supponiamo di avere due Heap della stessa altezza h, e un nuovo elemento con chiave k. Vogliamo creare un nuovo Heap di altezza $h + 1$ che contenga tutti gli elementi dei due Heap e l'elemento con chiave k. Per fare ciò, possiamo seguire questi passaggi:

- Creiamo un nuovo nodo radice con chiave k.
- Assegniamo i due Heap esistenti come figli sinistro e destro della nuova radice.
- Eseguiamo l'operazione di down-heap bubbling a partire dalla radice per ripristinare la proprietà di heap-order.

Heapify

Per semplicità ipotizziamo che il numero di elementi n sia un intero tale che $n = 2^{h+1} - 1$ per qualche intero $h \geq 0$, in modo che l'Heap risultante sia un albero binario completo, con anche l'ultimo livello completamente pieno, per cui l'Heap ha altezza $h = \log_2(n + 1) - 1$. L'algoritmo Heapify può essere visto come una sequenza di $h + 1 = \log_2(n + 1)$ step [dato dal fatto che in un albero di altezza h ci sono $h + 1$ livelli, numerati da 0 a h]:

1. Nel primo step (Figura 9.5b), costruiamo $(n + 1)/2$ Heap di altezza 0 (cioè nodi singoli).
2. Nel secondo step (Figura 9.5c-d), costruiamo $(n + 1)/4$ Heap di altezza 1, ciascuno contenente tre nodi, unendo coppie di Heap elementari (ottenuti al passo precedente) e aggiungendo un nuovo elemento. Il nuovo nodo viene posizionato alla radice e potrebbe dover essere scambiato con un elemento figlio per preservare la heap-order property.
3. Nel terzo step (Figura 9.5e-f), costruiamo $(n + 1)/8$ Heap di altezza 2, ciascuno contenente sette nodi, unendo coppie di Heap (ottenuti al passo precedente) e aggiungendo un nuovo elemento. Il nuovo nodo viene posizionato alla radice e potrebbe dover essere scambiato attraverso un down-heap bubbling per preservare la heap-order property.

⋮

i. Nel generico i-esimo step, con $2 \leq i \leq h$, costruiamo $(n + 1)/2^i$ Heap di altezza $i - 1$, ciascuno contenente $2^i - 1$ nodi, unendo coppie di Heap (ottenuti al passo precedente) e aggiungendo un nuovo elemento. Il nuovo nodo viene posizionato alla radice e potrebbe dover essere scambiato attraverso un down-heap bubbling per preservare la heap-order property.

⋮

$h + 1$. Nell'ultimo step (Figura 9.5g-h), costruiamo un singolo Heap di altezza h, contenente tutti e n gli elementi, unendo gli ultimi due Heap che risultano essere di $(n - 1)/2$ elementi e di altezza $h - 1$ (ottenuti al passo precedente) e aggiungendo un nuovo elemento. Il nuovo nodo viene posizionato alla radice e potrebbe dover essere scambiato attraverso un down-heap bubbling per preservare la heap-order property.

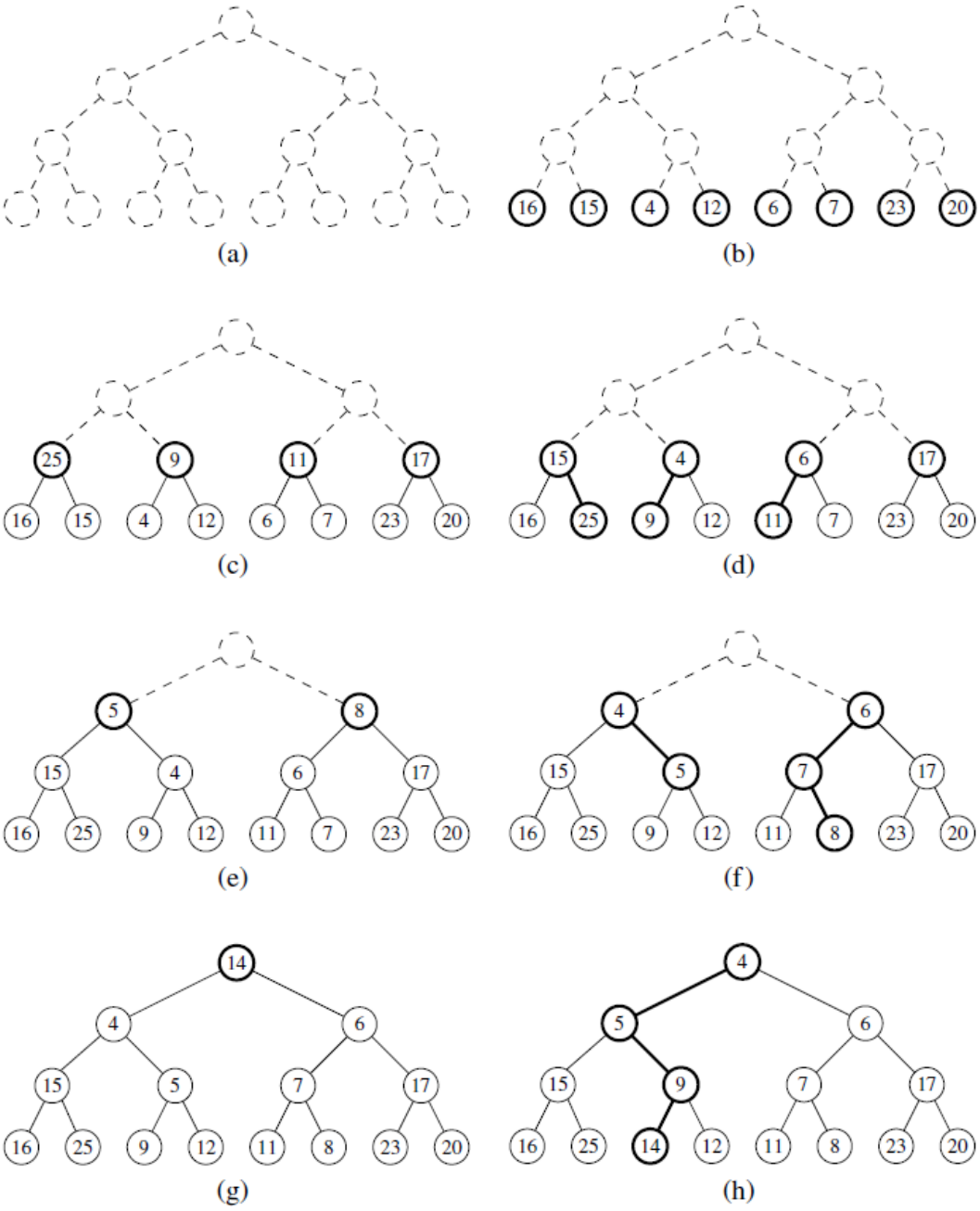


Figura 6.2: Costruzione bottom-up di un Heap con 15 elementi: (a, b) iniziamo costruendo Heap elementari di un solo elemento; (c, d) combiniamo questi Heap in Heap di 3 elementi, e poi (e, f) in Heap di 7 elementi, fino a (g, h) dove otteniamo l'Heap finale di 15 elementi. Il path del down-heap bubbling è evidenziato in (d, f, h). Per semplicità è mostrata solamente la chiave in ogni nodo anziché la coppia chiave-valore.

6.8 Heap-Sort

L'algoritmo **Heap-Sort** sfrutta la struttura dati Heap: data la lista di partenza, inserisco tutti gli elementi in un **Max-Heap** (al fine di avere gli elementi in ordine crescente alla fine dell'algoritmo) e poi estraggo ripetutamente l'elemento massimo (la radice del Max-Heap) per costruire la lista ordinata. Questo algoritmo viene eseguito già in tempo $O(n \cdot \log n)$, poiché l'inserimento di n elementi in un Heap richiede un tempo $O(n)$ (grazie all'algoritmo Heapify) e ogni estrazione dell'elemento massimo richiede un tempo $O(\log n)$, per un totale di n estrazioni.

L'idea per un ordinamento in-place

L'algoritmo che presentiamo di seguito è una versione più raffinata dello stesso Heap-sort che anziché utilizzare memoria aggiuntiva permette di avere un **ordinamento in-place** (cioè occupa al più memoria aggiuntiva $O(1)$) dell'Heap-sort.

L'idea chiave è dividere l'array C in due parti contigue:

1. **La parte sinistra (Heap):** Da $C[0]$ a $C[i-1]$.
2. **La parte destra (Sequenza):** Da $C[i]$ a $C[n-1]$.

Durante l'algoritmo, il confine i tra queste due parti si sposta.

Fase 1: costruzione di un Max Heap

Obiettivo: Trasformare l'array C in un unico Max-Heap.

In questa fase, si parte da un heap vuoto e si sposta il confine i **da sinistra verso destra**, da 1 a n. La parte sinistra (l'Heap) cresce, mentre la parte destra (la Sequenza di input) si riduce.

Per ogni passo i, da $i = 1$ fino a n:

1. **Azione (Espansione):** Il confine si sposta. L'elemento $C[i-1]$ (il primo della Sequenza) viene aggiunto all'Heap, diventandone l'ultima foglia.

2. **Aggiustamento (Up-Heap Bubbling):** L'aggiunta di $C[i-1]$ potrebbe violare la proprietà del Max-Heap. Si esegue quindi un **Up-Heap Bubbling** a partire da $C[i-1]$. Questo elemento viene scambiato con il suo genitore finché non è più piccolo del genitore o non raggiunge la radice $C[0]$.

Alla fine della Fase 1, $i = n$. La parte Heap occupa l'intero array $C[0 \ldots n-1]$ e la Sequenza è vuota. L'intero array è ora un Max-Heap valido.

Fase 2: estrazione degli elementi in ordine

Obiettivo: Estrarre gli elementi dall'Heap in ordine decrescente per costruire l'array ordinato.

In questa fase, si parte con tutti gli elementi nell'Heap e la Sequenza vuota. Si sposta il confine **da destra verso sinistra**.

Per ogni passo i, da $i = 1$ fino a n:

1. **Azione (Estrazione):** L'elemento massimo dell'Heap si trova sempre alla radice, $C[0]$. Questo elemento deve essere spostato nella sua posizione ordinata finale, che in questo passo è $C[n-i]$ (la prima posizione libera a sinistra della Sequenza ordinata). Si **scambiano** $C[0]$ e $C[n-i]$.

2. **Aggiustamento (Down-Heap Bubbling):** Dopo lo scambio, $C[n-i]$ contiene il massimo ed è ora "bloccato" (fa parte della Sequenza ordinata). L'Heap si è ridotto (ora va da $C[0]$ a $C[n-i-1]$). L'elemento che è finito in $C[0]$ (quello che era $C[n-i]$) è probabilmente fuori posto e viola la proprietà del max-Heap. Si esegue quindi un **Down-Heap Bubbling** a partire dalla radice $C[0]$. Questo elemento "scende" scambiandosi con il suo figlio *maggiore*, finché non è più grande di entrambi i figli o non raggiunge una foglia dell'Heap.

Alla fine della Fase 2, $i = n$. La parte "Heap" è vuota e la "Sequenza" (ora ordinata in senso crescente) occupa l'intero array.

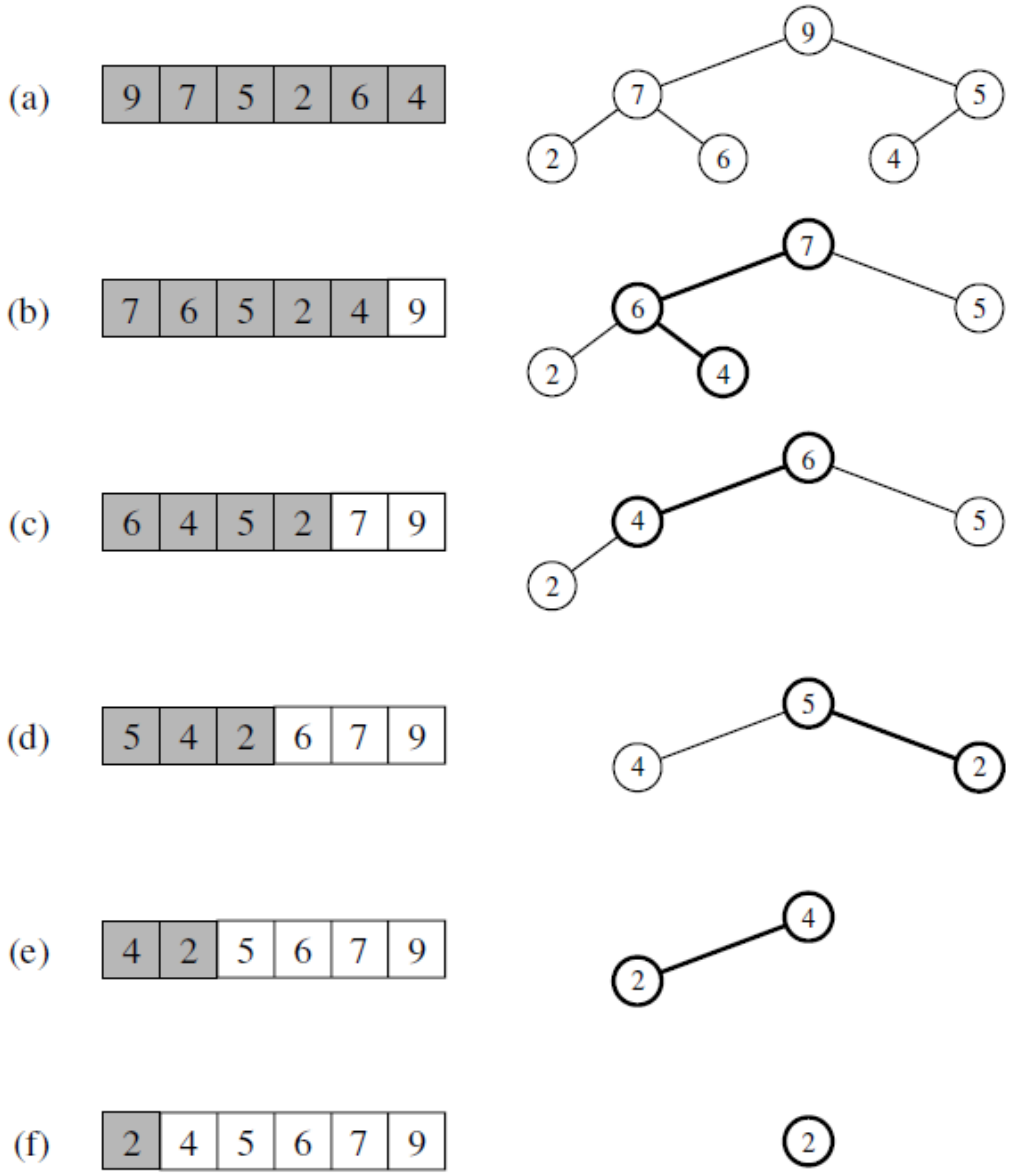


Figura 6.3: Fase 2 di un Heap-Sort in-place. La porzione Heap è evidenziata in grigio all'interno dell'array, per ogni iterazione dell'algoritmo. L'albero binario equivalente alla porzione Heap per ogni iterazione è rappresentato graficamente con il percorso più recente di Down-Heap Bubbling evidenziato.

6.9 Adaptable Priority Queue

L'ADT Priority Queue è sufficiente per molte applicazioni, ma in alcuni casi è indispensabile avere delle funzionalità aggiuntive per gestire in modo più flessibile gli elementi all'interno della coda, per esempio modificare la chiave (priorità) di un elemento esistente o rimuovere un elemento specifico (non necessariamente quello con priorità minima). Per questo motivo, presentiamo una variante chiamata **Adaptable Priority Queue** che estende l'ADT Priority Queue con queste funzionalità aggiuntive.

Locators

Per poter implementare in modo efficiente le nuove operazioni di modifica e cancellazione, è necessario trovare un meccanismo che ci permetta di trovare uno specifico elemento all'interno della coda senza dover scorrere l'intera struttura dati. Per questo motivo, quando un elemento viene inserito nella coda, viene restituito uno speciale oggetto chiamato **locator** al chiamante. Di conseguenza, ogni volta che si desidera modificare o rimuovere un elemento specifico di una coda a priorità P è necessario utilizzare il locator associato a quell'elemento per accedervi direttamente.

`P.update(loc, k, v)` : Sostituisce la chiave k e il valore v dell'elemento associato al locator loc nella coda a priorità P.

`P.remove(loc)` : Rimuove l'elemento associato al locator loc dalla coda a priorità P e lo restituisce.

L'astrazione del *locator* è in qualche modo simile all'astrazione della *position*. Tuttavia, facciamo una distinzione tra un locator e una position perché un locator per una coda prioritaria non rappresenta una collocazione tangibile di un elemento all'interno della struttura. Nella nostra coda prioritaria, un elemento può essere ricollocato all'interno della nostra struttura dati durante un'operazione che non sembra direttamente rilevante per quell'elemento. Un locator per un elemento rimarrà valido fintanto che quell'elemento rimarrà da qualche parte nella coda.

Implementazione di un Adaptable Priority Queue

L'implementazione della classe Locator estende la classe _Item dell'ADT Priority Queue per includere un campo aggiuntivo che tiene traccia della posizione corrente dell'elemento all'interno della rappresentazione basata su array del nostro Heap. Questo campo aggiuntivo consente di accedere rapidamente alla posizione dell'elemento nell'Heap, facilitando le operazioni di aggiornamento e rimozione, come mostrato Nella Figura 6.4.

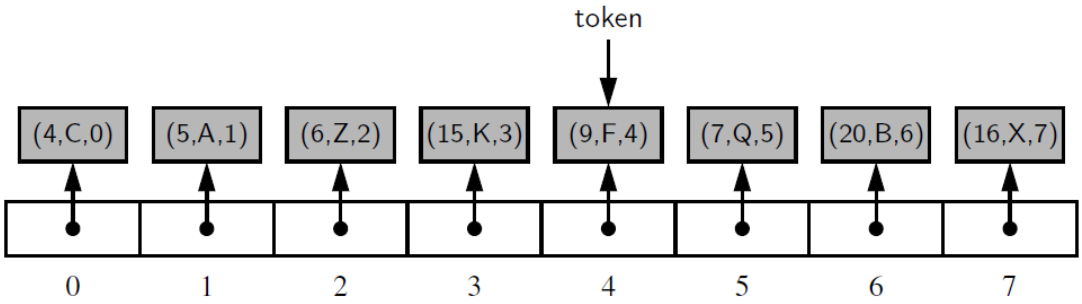


Figura 6.4: Rappresentazione di un Heap utilizzando una sequenza di Locator. Il terzo elemento di ciascuna istanza di Locator corrisponde all'indice dell'elemento all'interno dell'array. Si presume che l'identificatore "token" sia un riferimento al localizzatore nello scope dell'utente.

La lista è una sequenza di riferimenti a istanze di Locator, ognuna delle quali memorizza una chiave, un valore, e l'indice corrente dell'elemento all'interno dell'array che rappresenta l'Heap. All'utente verrà fornito un riferimento al Locator corrispondente per ciascun elemento inserito, come illustrato dall'identificatore "token" nella Figura 6.4.

Quando eseguiamo delle operazioni sulla Priority Queue che potrebbero alterare la posizione di un elemento all'interno dell'Heap (come l'inserimento che comporta un Up-Heap Bubbling, la rimozione che comporta un Down-Heap Bubbling o l'aggiornamento della chiave), dobbiamo assicurarci di aggiornare il campo dell'indice all'interno del Locator corrispondente. Questo garantisce che il Locator rimanga valido e punti sempre alla posizione corretta dell'elemento nell'Heap.

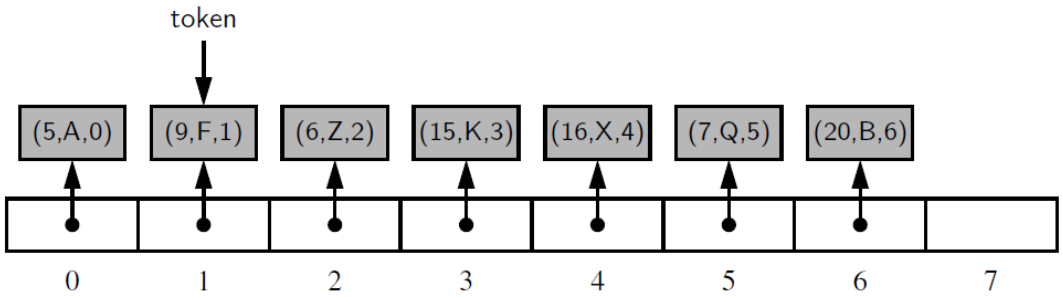


Figura 6.5: È possibile osservare lo stato dell'Heap dopo aver eseguito una `remove_min()`. Questa operazione causa un Down-Heap Bubbling che ricolloca gli elementi all'interno dell'Heap, e di conseguenza aggiorna gli indici nei Locator associati agli elementi coinvolti nel bubbling.

Operation	Running Time
len(P), P.is_empty( ), P.min( )	$O(1)$
P.add(k,v)	$O(\log n)^*$
P.update(loc, k, v)	$O(\log n)$
P.remove(loc)	$O(\log n)^*$
P.remove_min( )	$O(\log n)^*$

*amortized with dynamic array

Capitolo 7

Pattern Matching

Introduciamo di seguito la terminologia di base:

- Σ : l'alfabeto, ovvero l'insieme di caratteri possibili.
- $|\Sigma|$: la dimensione dell'alfabeto.
- Stringa S : una sequenza finita di caratteri appartenenti all'alfabeto Σ, di lunghezza m.
- $S[i]$: il carattere alla posizione i della stringa S.
- $S[i..j]$: la sottostringa di S che va dall'indice i all'indice j.
  - In Python: S[i:j+1]
- $S[0..k]$: prefisso di lunghezza $k + 1$ della stringa S.
  - In Python: S[:k+1]
- $S[j..m - 1]$: suffisso di lunghezza $m - j$ della stringa S.
  - In Python: S[j:]

Nel classico problema di pattern matching, ci viene data una stringa di testo T di lunghezza n e una stringa di pattern P di lunghezza m, e vogliamo scoprire se P è una sottostringa di T. In tal caso, potremmo voler trovare l'indice più basso j all'interno di T in cui inizia P, in modo che $T[j..j + m - 1]$ sia uguale a P, o forse trovare tutti gli indici di T in cui inizia il pattern P.

7.1 Brute Force

Il metodo più semplice per risolvere il problema del pattern matching è il metodo *brute force*. L'idea alla base di questo metodo è di confrontare il pattern P con ogni possibile sottostringa di T di lunghezza m. In particolare, per ogni indice i da 0 a $n - m$, confrontiamo la sottostringa $T[i..i + m - 1]$ con il pattern P. Se troviamo una corrispondenza, restituiamo l'indice i.

```
def find_brute(T, P):
    """Return the lowest index of T at which substring P begins (or else
        -1)."""
    n, m = len(T), len(P)  # introduce convenient notations
    for i in range(n-m+1):  # try every potential starting index within T
        k = 0  # an index into pattern P
        while k < m and T[i + k] == P[k]:  # kth character of P matches
            k += 1
        if k == m:  # if we reached the end of pattern,
            return i  # substring T[i:i+m] matches P
    return -1  # failed to find a match starting with any i
```

Performance

L'algoritmo consiste in due cicli annidati, con il ciclo esterno che scorre tutti i possibili indici iniziali del pattern nel testo T, e il ciclo interno che scorre ogni carattere del pattern P, confrontandolo con il suo potenziale carattere corrispondente nel testo. Pertanto, la correttezza dell'algoritmo deriva direttamente da questo approccio di ricerca esaustiva.

Il tempo di esecuzione del pattern matching tramite *Brute Force* nel caso peggiore non è buono poiché per ogni indice candidato in T, possiamo eseguire fino a m confronti di caratteri per scoprire che P non corrisponde a T all'indice corrente. Dal blocco di codice si può osservare che il ciclo for esterno viene eseguito al massimo $n - m + 1$ volte e il ciclo $while$ interno viene eseguito al massimo m volte. Pertanto, il tempo di esecuzione nel caso peggiore è $O(n \cdot m)$.

Esempio

Supponiamo di avere un testo

$$T = \text{"abacaabaccabacabaabb"}$$

e un pattern

$$P = \text{"abacab"}$$

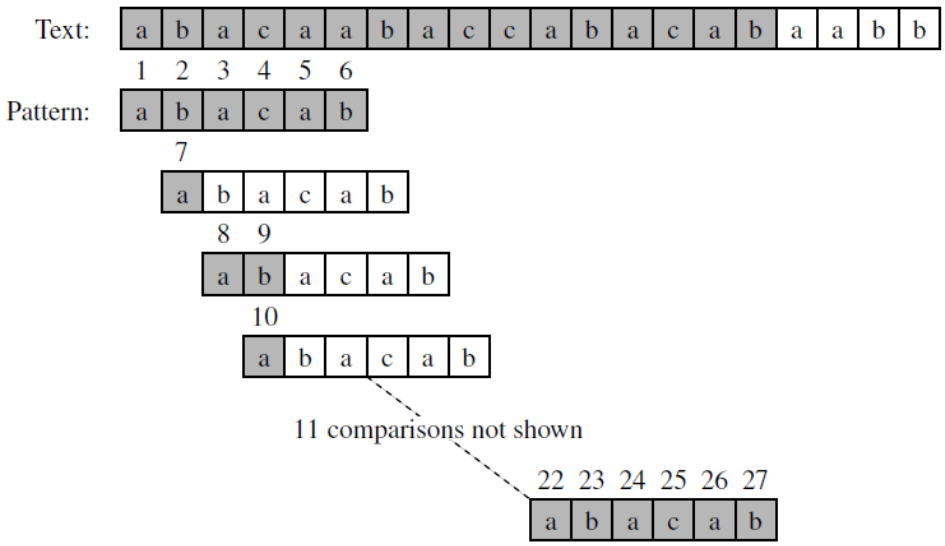


Figura 7.1: Esempio di Pattern Matching con algoritmo Brute Force. L'algoritmo esegue 27 confronti tra caratteri, numerati in figura.

7.2 L'algoritmo di Boyer-Moore

Come vedremo tra poco, non è sempre necessario confrontare ogni carattere del pattern con il testo. L'algoritmo di *Boyer-Moore* sfrutta questa osservazione per saltare alcune posizioni nel testo, riducendo così il numero di confronti necessari.

L'idea principale dell'algoritmo di *Boyer-Moore* è di migliorare l'efficienza dell'algoritmo *Brute Force* utilizzando due tecniche (euristiche) principali:

- **Looking-Glass Heuristic**: Quando si confrontano i caratteri del pattern con il testo, si inizia dal carattere più a destra del pattern e si procede verso sinistra.
- **Character-Jump Heuristic**: Durante la verifica di un possibile piazzamento di P in T, un mismatch tra $T[i] = c$ e $P[k]$ viene gestito come segue:

  Supponiamo che $T[i] \neq P[k]$ e $T[i] = c$.

  - Se c non appare in p, p può essere "spostato" completamente oltre $T[i]$ ($P[0]$ viene allineato con $T[i+1]$).
  - Altrimenti, $T[i]$ viene allineato con l'ultima occorrenza di c in P.

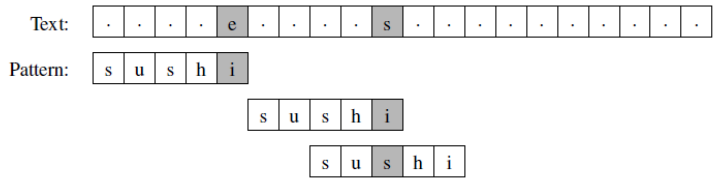


Figura 7.2: Una semplice dimostrazione dell'algoritmo di Boyer-Moore. Nel primo confronto si ha $T[4] \neq P[4]$ con $T[4] =$ 'e' che non è presente in P, per cui spostiamo P oltre $T[4]$. Nel secondo confronto si ha $T[9] \neq P[4]$ con $T[9] =$'s' che è presente in P, in particolare l'ultima occorrenza di 's' è in $P[2]$, per cui allineiamo $P[2]$ con $T[9]$.

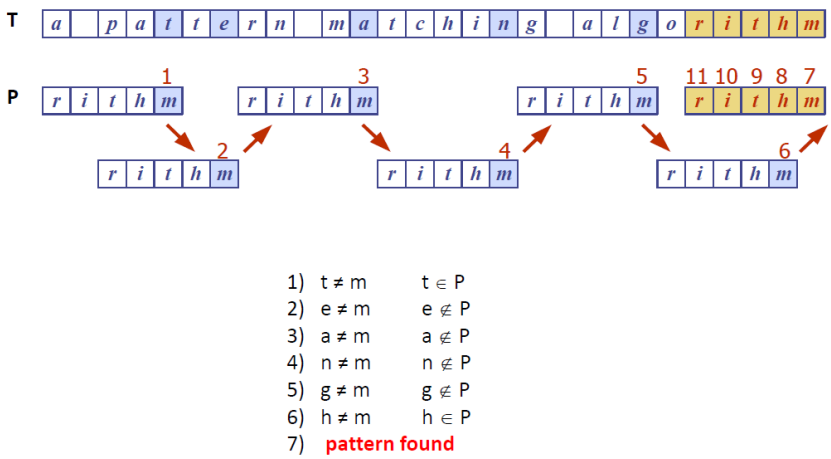


Figura 7.3: Esempio completo che mostra il numero di confronti.

Per formalizzare l'algoritmo di *Boyer-Moore* possiamo generalizzare il funzionamento come di seguito:

- Quando viene trovata una corrispondenza (a partire dall'ultimo carattere del pattern), l'algoritmo continua cercando di estendere la corrispondenza con il penultimo carattere del pattern nel suo allineamento corrente. Questo processo continua fino a quando tutti i caratteri del pattern sono stati confrontati con esito positivo o fino a quando si verifica un mismatch.

- Quando si verifica un mismsatch, e il carattere del testo che ha causato il mismatch non è presente nel pattern, il pattern viene spostato completamente oltre quel carattere del testo. Se il carattere del testo è presente da qualche altra parte nel pattern, dobbiamo considerare due diversi casi a seconda che la sua ultima occorrenza sia (a) precedente o (b) successiva al carattere del pattern che era allineato con il carattere del testo che ha causato il mismatch.

  Questi due casi sono rappresentati in figura 7.4 e approfonfiti di seguito:

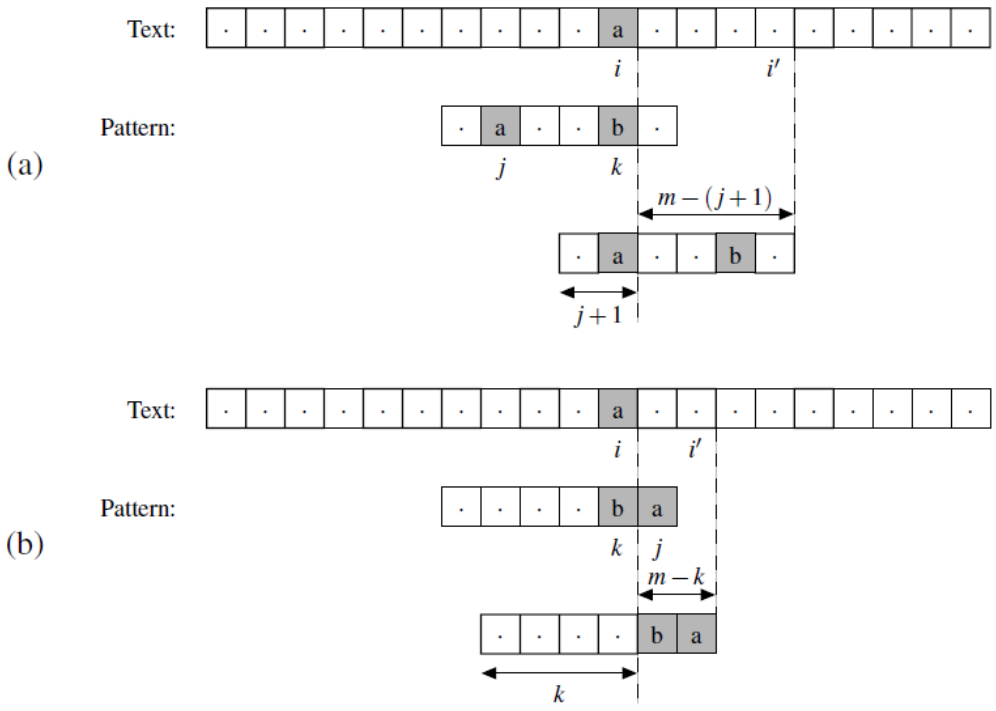


Figura 7.4: Indichiamo con i l'indice del carattere non corrispondente nel testo, con k l'indice corrispondente nel pattern e con j l'indice dell' ultima occorrenza di $T[i]$ all'interno del pattern. Distinguiamo due casi: **(a)** $j < k$, nel qual caso spostiamo il pattern di $k - j$ unità, e quindi l'indice i avanza di $m - (j + 1)$ unità; **(b)** $j > k$, nel qual caso spostiamo il pattern di un'unità, e l'indice i avanza di $m - k$ unità. N.B. m è la lunghezza del pattern.

Nel caso di 7.4(b), il pattern viene spostato di una sola unità a destra. Sarebbe sicuramente più produttivo spostare il pattern a destra fino a quando l'ultima occorrenza di $T[i]$ nel pattern non è allineata con $T[i]$, ma questo richiederebbe un ulteriore calcolo per la ricerca della nuova occorrenza.

L'efficienza dell'algoritmo di *Boyer-Moore* risiede nell'utilizzo di una funzione di pre-elaborazione, $L(c)$, chiamata **last occurrence function** che consente di determinare rapidamente l'ultima occorrenza di un carattere nel pattern. Questa funzione viene calcolata una sola volta prima dell'inizio del processo di ricerca e viene utilizzata ogni volta che si verifica un mismatch.

$$\forall c \in \Sigma, \quad L(c) = \begin{cases} \max\{i \mid P[i] = c\} & \text{se } c \in P \\ -1 & \text{se } c \notin P \end{cases}$$

Per esempio, se abbiamo Σ = {'a', 'b', 'c', 'd'} e P = "abacab" :

c	**'a'**	**'b'**	**'c'**	**'d'**
$L(c)$	4	5	3	-1

Possiamo modellare L come una *Map* che ha per chiavi i caratteri dell'alfabeto e per valori gli indici delle loro ultime occorrenze nel pattern (ad esempio, in Python possiamo usare un dizionario). La Map L può essere costruita in tempo $O(m + |\Sigma|)$ dove m è la lunghezza del pattern e $|\Sigma|$ è la dimensione dell'alfabeto.

```
def last_occurrence(p, sigma):
    """Return the last-occurrence map L for pattern p over alphabet
       sigma."""
    L = {c:-1 for c in sigma}  # initialize all characters to -1
    for i in range(len(p)):
        L[p[i]] = i  # update with last occurrence of character p[i]
    return L
```

```python
def find_boyer_moore(T, P):
    """Return the lowest index of T at which substring P begins (or else
        -1)."""
    n, m = len(T), len(P)   # introduce convenient notations
    if m == 0:
        return 0            # trivial search of empty pattern

    # last occurrence function
    last = {}               # build the last-occurrence map
    for k in range(m):
        last[P[k]] = k      # later occurrence overwrites

    # align end of pattern at index m-1 of text
    i = m-1               #an index into T
    k = m-1                 #an index into P
    while i < n:
        if T[i] == P[k]:    # a matching character
            if k == 0:
                return i    # pattern begins at index i of text
            else:
                i -= 1      # examine previous character
                k -= 1      # of both T and P
        else:
            j = last.get(T[i], -1)  # last(T[i]) is -1 if not found
            i += m-min(k, j+1)      # case analysis for jump step
            k = m-1                 # restart at end of pattern
    return -1
```

Performance

La versione semplificata dell'algoritmo di Boyer-Moore presentata qui ha un tempo di esecuzione di $O(n \cdot m + |\Sigma|)$, in quanto la last-occurrence function richiede $O(m + |\Sigma|)$ tempo per essere costruita e la ricerca del pattern richiede $O(n \cdot m)$.

Il caso peggiore si verifica quando si ha una coppia del tipo:

$$T = \text{"aaa...aaa"} \quad P = \text{"baa...aaa"}$$

In questo caso, l'algoritmo di Boyer-Moore si comporta come l'algoritmo Brute Force, eseguendo $O(n \cdot m)$ confronti tra caratteri.

Tuttavia, l'algoritmo originale di Boyer-Moore utilizza euristiche più avanzate ed efficienti che consentono di ottenere un tempo di esecuzione medio di $O(n + m + |\Sigma|)$.

Esempio

Supponiamo di avere un testo

$$T = \text{"abacaabadcabacabaabb"}$$

e un pattern

$$P = \text{"abacab"}$$

c	a	b	c	d
$L(c)$	4	5	3	-1

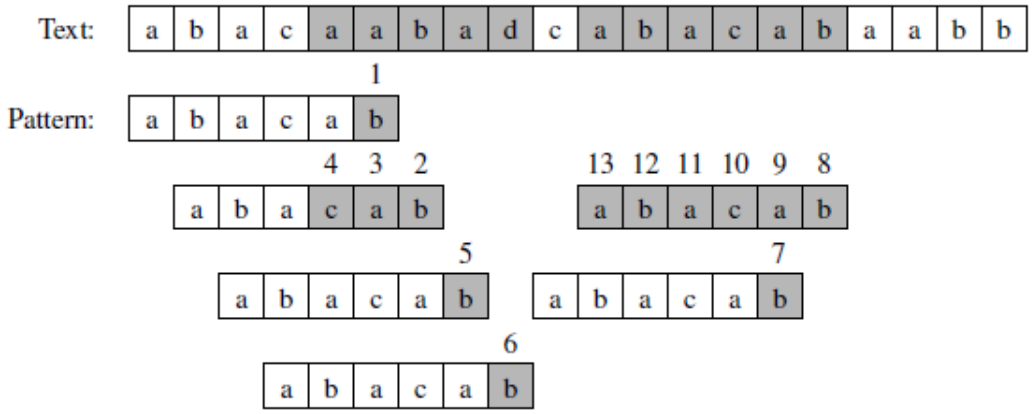


Figura 7.5: Esempio di Pattern Matching con algoritmo di Boyer-Moore, con anche la tabella delle last occurrence. L'algoritmo esegue 13 confronti tra caratteri, numerati in figura.

7.3 L'algoritmo di Knuth-Morris-Pratt

Analizzando le prestazioni nel caso peggiore degli algoritmi di pattern-matching *brute-force* e *Boyer-Moore* su istanze specifiche del problema possiamo notare una notevole inefficienza. Per un certo allineamento del pattern, se troviamo diversi caratteri corrispondenti ma poi rileviamo un mismatch, ignoriamo tutte le informazioni ottenute dai confronti andati a buon fine.

L'algoritmo di *Knuth-Morris-Pratt* (o "KMP") evita questo spreco di informazioni e, facendo così, raggiunge un tempo di esecuzione di $O(n+m)$, $O(m)$ per il pre-processing del pattern, e $O(n)$ per i confronti tra testo e pattern. Nel caso peggiore qualsiasi algoritmo di pattern-matching dovrà esaminare tutti i caratteri del testo e tutti i caratteri del pattern almeno una volta. L'idea principale dell'algoritmo KMP è quella di pre-calcolare le sovrapposizioni del pattern su se stesso. In questo modo, quando si verifica un mismatch in una certa posizione, sappiamo immediatamente qual è lo spostamento massimo che possiamo applicare al pattern prima di continuare la ricerca.

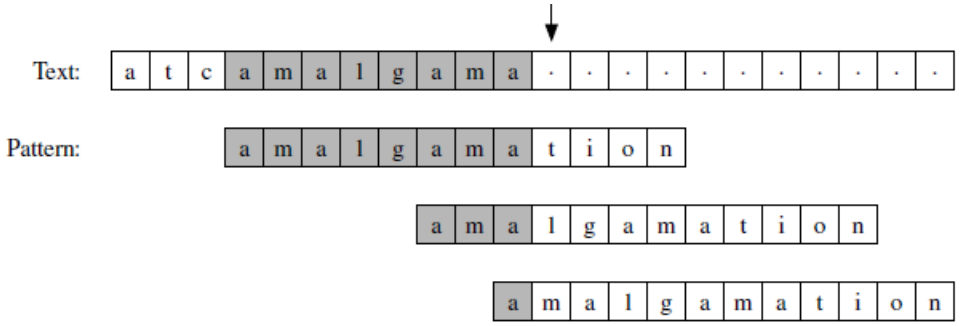


Figura 7.6: KMP non ripete i confronti sui primi tre caratteri del pattern dal momento che si ripetono e sono già stati confrontati con successo.

7.3.1 Failure-Function

L'algoritmo KMP si basa sul calcolo della **failure function** f, che indica il corretto spostamento di P in caso di confronto fallito. In particolare, la funzione di fallimento $f(k)$ è definita come la **lunghezza del più lungo prefisso di P che è anche un suffisso di $P[1:k+1]$**[1]. Intuitivamente, se troviamo un mismatch sul carattere $P[k+1]$, la funzione $f(k)$ ci dice quanti dei caratteri immediatamente precedenti possono essere riutilizzati per riavviare il pattern.

Operativamente, la funzione viene utilizzata nel seguente modo: se durante il confronto si verifica un *mismatch* all'indice j del pattern (cioè $P[j] \neq T[i]$), significa che i caratteri precedenti $P[0 \ldots j-1]$ corrispondevano al testo. Invece di ripartire da zero, l'algoritmo consulta il valore $f(j-1)$, il quale ci indica quanti caratteri del prefisso possiamo "salvare". Il confronto riprenderà quindi confrontando il carattere del testo $T[i]$ con il nuovo indice del pattern $j' = f(j-1)$.

[1] P[1:k+1] comprende i caratteri da indice 1 a indice k, $k+1$ non è incluso. Nota che non abbiamo incluso $P[0]$ qui, dal momento che una stringa è suffisso di sè stesso; così facendo imponiamo che sia un **suffisso proprio**, cioè diverso dalla stringa stessa

Per esempio, consideriamo il pattern P = "amalgamation". La failure function f per questo pattern è mostrata nella tabella seguente:

k	0	1	2	3	4	5	6	7	8	9	10	11
$P[k]$	a	m	a	l	g	a	m	a	t	i	o	n
$f[k]$	0	0	1	0	0	1	2	3	0	0	0	0

Implementazione

L'implementazione dell'algoritmo KMP si basa su una funzione di utilità, `compute_kmp_fail`, per calcolare efficientemente la funzione di fallimento. La parte principale dell'algoritmo KMP è il suo ciclo `while`, che per ogni iterazione esegue un confronto tra il carattere $T[j]$ e il carattere $P[k]$. Se l'esito di questo confronto è una corrispondenza, l'algoritmo procede ai caratteri successivi sia in T che in P (o segnala una corrispondenza completa se si raggiunge la fine del pattern). Se il confronto fallisce, l'algoritmo consulta la funzione di fallimento per un nuovo carattere candidato in P, o ricomincia con il prossimo indice in T se si fallisce sul primo carattere del pattern (dato che nulla può essere riutilizzato).

```
def find_kmp(T, P):
    """Return the lowest index of T at which substring P begins (or else
        -1)."""
    n, m = len(T), len(P)   # introduce convenient notations
    if m == 0:
        return 0            # trivial search of empty pattern

    fail = compute_kmp_fail(P)  # rely on utility to precompute fail
        function
    j = 0                   # index into text
    k = 0                   # index into pattern
    while j < n:
        if T[j] == P[k]:    # P[0:k] matched thus far
            if k == m-1:    # match is complete
                return j - m + 1   # pattern begins at index (j-m+1) of text
            j += 1          # try to extend match
            k += 1
        elif k > 0:
            k = fail[k-1]   # reuse suffix of P[0:k-1]
        else:
            j += 1          # no match at P[0], try next character in T
    return -1
```

```python
def compute_kmp_fail(P):
    """Utility that computes and returns KMP fail list."""
    m = len(P)
    fail = [0] * m      # by default, presume overlap of 0 everywhere
    j = 1
    k = 0
    while j < m:        # compute f(j) during this pass, if nonzero
        if P[j] == P[k]:  # k+1 characters match thus far
            fail[j] = k + 1
            j += 1
            k += 1
        elif k > 0:     # k follows a matching prefix
            k = fail[k-1]
        else:           # no match found starting at j
            j += 1
    return fail
```

Performance

Tralasciando momentaneamente il calcolo della failure function, il tempo di esecuzione dell'algoritmo KMP è chiaramente proporzionale al numero di iterazioni del ciclo while. Ai fini dell'analisi, definiamo $s = j - k$ come la quantità totale di spostamento del pattern P rispetto al testo T. Notiamo che durante l'esecuzione dell'algoritmo, abbiamo sempre $s \leq n$. Ad ogni iterazione del ciclo si verifica uno dei seguenti tre casi:

- Se $T[j] = P[k]$, allora sia j che k aumentano di 1, e quindi s non cambia.
- Se $T[j] \neq P[k]$ e $k > 0$, allora j non cambia e s aumenta di almeno 1, poiché in questo caso s cambia da $j - k$ a $j - f(k-1)$, che è un'aggiunta pari a $k - f(k-1)$, che è positiva perché $f(k-1) < k$.
- Se $T[j] \neq P[k]$ e $k = 0$, allora j aumenta di 1 e s aumenta di 1, poiché k non cambia.

Quindi, in ogni iterazione del ciclo, o j o s vengono incrementati di almeno 1 (eventualmente entrambi); pertanto, il numero totale di iterazioni del ciclo while nell'algoritmo KMP è al massimo $2n$. Per rendere tutto ciò possibile si presuppone che la failure function di P sia già stata precedentemente calcolata.

L'algoritmo per il calcolo della failure function ha una complessità di $O(m)$. La sua analisi è analoga a quella del principale algoritmo KMP, ma esegue dei confronti tra il pattern di lunghezza m con se stesso.

Dunque, combinando entrambi i risultati, otteniamo che l'algoritmo KMP ha una complessità temporale complessiva di $O(n + m)$. La correttezza dell'algoritmo KMP deriva direttamente dalla definizione della failure function. Qualsiasi confronto che viene saltato è in realtà superfluo, poiché la funzione di fallimento garantisce che tutti i confronti ignorati siano ridondanti.

Esempio

Supponiamo di avere un testo

$$T = \text{"abacaabaccabacabaabb"}$$

e un pattern

$$P = \text{"abacab"}$$

k	0	1	2	3	4	5
$P[k]$	a	b	a	c	a	b
$f(k)$	0	0	1	0	1	2

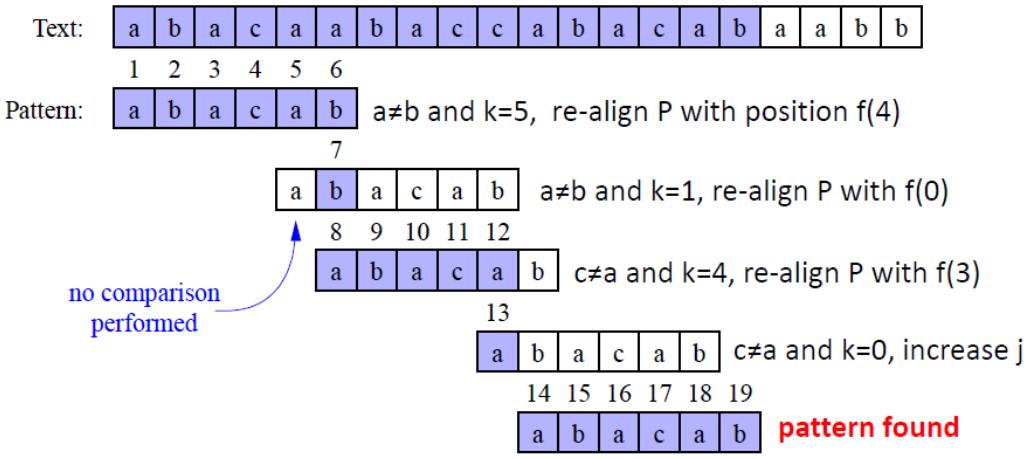


Figura 7.7: Esempio di Pattern Matching con algoritmo KMP, con anche la tabella della failure function. L'algoritmo esegue 19 confronti tra caratteri, numerati in figura (sono necessari ulteriori confronti per il calcolo della failure function non presenti in figura).

Capitolo 8

Tries

Il problema del *pattern matching* presentato nel capitolo 7 velocizza la ricerca in un testo effettuando una pre-elaborazione del pattern (per calcolare la funzione di fallimento nell'algoritmo di Knuth-Morris-Pratt o la funzione last nell'algoritmo di Boyer-Moore). In questa sezione, adottiamo un approccio complementare, presentando algoritmi di ricerca di stringhe che effettuano una pre-elaborazione del testo. Questo approccio è adatto per applicazioni in cui viene eseguita una serie di query su un testo fisso, in modo che il costo iniziale della pre-elaborazione del testo sia compensato da un'accelerazione in ogni query successiva. A questo proposito introduciamo i *tries* (pronunciato "try"), una struttura dati basata su alberi per memorizzare stringhe al fine di supportare un rapido pattern matching. La principale applicazione dei tries è nel recupero delle informazioni, da cui il nome "trie" che deriva dalla parola "retrieval".

8.1 Standard Tries

Sia S un insieme di s stringhe su un alfabeto Σ tale che nessuna stringa in S sia prefisso di un'altra stringa. Uno **Standard Trie** per S è un albero ordinato T con le seguenti proprietà:

- Ogni nodo di T, eccetto la radice, è etichettato con un carattere di Σ.
- I figli di un nodo interno di T hanno etichette distinte.
- T ha s foglie, ciascuna associata a una stringa di S, tale che la concatenazione delle etichette dei nodi sul percorso dalla radice a una foglia v di T dia la stringa di S associata a v.

Dunque, un trie T rappresenta le stringhe di S tramite i percorsi dalla radice alle foglie di T. È importante assumere che nessuna stringa in S sia prefisso di un'altra sintra poichè ciò garantisce che ogni stringa di S sia univocamente associata a una foglia di T. Possiamo sempre soddisfare questa assunzione aggiungendo un carattere speciale che non è nell'alfabeto originale Σ alla fine di ogni stringa.

Uno standard trie che memorizza una collezione S di s stringhe di lunghezza totale n da un alfabeto Σ ha le seguenti proprietà:

- L'altezza di T è uguale alla lunghezza della stringa più lunga in S.
- Ogni nodo interno di T può avere da 1 a $|\Sigma|$ figli.
- T ha s foglie.
- Il numero di nodi di T è al più $n + 1$.
  - Infatti, il caso peggiore per il numero di nodi di un trie si verifica quando nessuna coppia di stringhe condivide un prefisso non vuoto; cioè, ad eccezione della radice, tutti i nodi interni hanno un solo figlio.

Ricerca

Un trie T per un insieme S di stringhe permette di implementare una mappa in cui le chiavi sono le stringhe stesse; la ricerca di una stringa X avviene tracciando dalla radice il percorso indicato dai caratteri di X e, se tale percorso termina in un nodo foglia, la stringa è presente nella mappa, mentre se il percorso si interrompe o termina in un nodo interno la stringa non è una chiave valida.

Il tempo di esecuzione per cercare una stringa di lunghezza m è limitato superiormente da $O(m \cdot |\Sigma|)$, in quanto possiamo visitare al più $m + 1$ nodi di T e spendiamo $O(|\Sigma|)$ tempo in ogni nodo per determinare il figlio che ha come etichetta il carattere successivo. Tuttavia, possiamo migliorare il tempo speso in un nodo a $O(\log |\Sigma|)$ o atteso $O(1)$, mappando i caratteri ai figli utilizzando una tabella hash in ogni nodo, oppure utilizzando una lookup table diretta di dimensione $|\Sigma|$ in ogni nodo, se $|\Sigma|$ è sufficientemente piccolo (come nel caso delle stringhe di DNA). Per questi motivi, ci aspettiamo tipicamente che una ricerca per una stringa di lunghezza m venga eseguita in tempo $O(m)$.

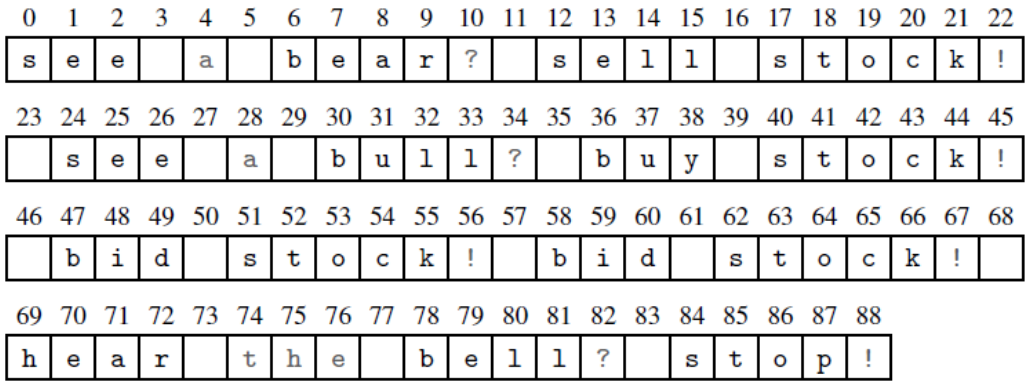
Word Matching

Grazie a queste caratteristiche, il trie è adatto per il **word matching** esatto e per le query sui prefissi (per via dell'operazione di ricerca appena descritta), ma non per il pattern matching di sottostringhe arbitrarie. Il *word matching* è un caso particolare di pattern matching in cui vogliamo determinare se un dato pattern corrisponde esattamente a una delle parole del testo. Il word matching si differenzia dal pattern matching standard analizzato nel capitolo 7 perché in questo caso il pattern non può corrispondere a una sottostringa arbitraria del testo, ma solo a una delle sue parole.

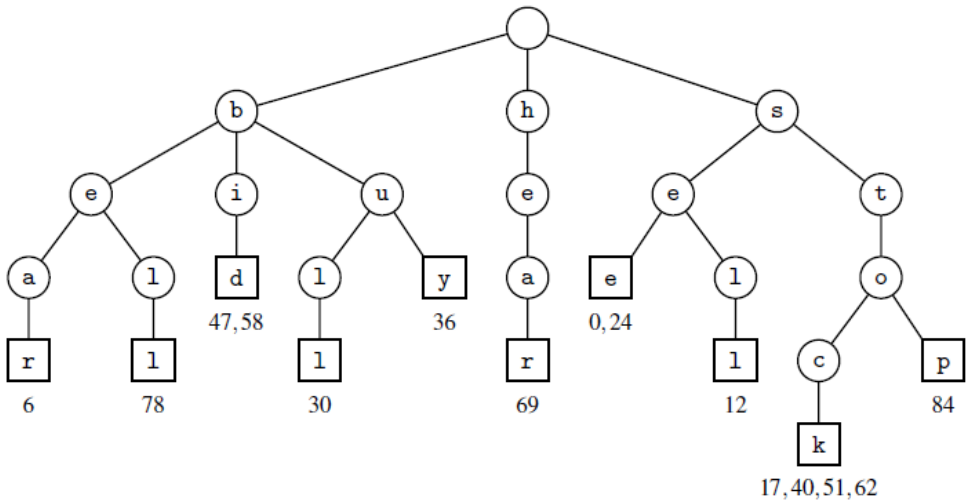
Costruzione di uno Standard Trie

Per costruire uno standard trie per un insieme S di stringhe, possiamo utilizzare un algoritmo incrementale che inserisce le stringhe una alla volta. Ricordiamo l'assunzione che nessuna stringa di S sia prefisso di un'altra stringa. Per inserire una nuova stringa X nel trie, seguiamo il percorso dei suoi caratteri finché esistono nodi corrispondenti. Nel punto in cui il percorso esistente si interrompe (ovvero non troviamo il carattere successivo), creiamo una nuova sequenza

di nodi per tutti i caratteri restanti di X. Il tempo di esecuzione per inserire X di lunghezza m è simile a una ricerca, con prestazioni nel caso peggiore di $O(m \cdot |\Sigma|)$, o attese $O(m)$ se si utilizzano tabelle hash secondarie in ogni nodo. Pertanto, la costruzione dell'intero trie per l'insieme S richiede un tempo atteso di $O(n)$, dove n è la lunghezza totale delle stringhe di S.



(a)



(b)

Figura 8.1: Word Matching tramite uno Standard Trie: (a) testo da cercare (articoli e preposizioni, noti anche come stop words, esclusi); (b) standard trie per le parole nel testo, con le foglie che mantengono l'informazione relativa all'indice in cui la data parola inizia nel testo. Ad esempio, la foglia per la parola *stock* indica che la parola inizia agli indici 17, 40, 51 e 62 del testo.

Come si può notare anche dall'esempio in Figura 8.1, c'è una potenziale inefficienza di spazio nello standard trie dovuta alla presenza di molti nodi che hanno un solo figlio, e l'esistenza di tali nodi rappresenta uno spreco. La ricerca di una soluzione a questo problema ha portato allo sviluppo del *Compressed Trie*, o *Patricia Trie*.

8.2 Compressed Tries

Un **Compressed Trie** è una variante dello standard trie che garantisce che ogni nodo interno del trie abbia almeno due figli. Questa regola viene applicata comprimendo le catene di nodi con un solo figlio.

Sia T un trie standard. Diciamo che un nodo interno v di T è **ridondante** se v ha un solo figlio e non è la radice. Diciamo anche che una *catena* di $k \geq 2$ archi,

$$(v_0, v_1)(v_1, v_2) \cdots (v_{k-1}, v_k)$$

è **ridondante** se:

- v_i è ridondante per $i = 1, \ldots, k-1$.
- v_0 e v_k non sono ridondanti.

Si può trasformare T in un *Compressed Trie* sostituendo ogni catena ridondante $(v_0, v_1) \cdots (v_{k-1}, v_k)$ di $k \geq 2$ archi con un singolo arco (v_0, v_k), rietichettando v_k con la concatenazione delle etichette dei nodi $v_1, \ldots, v_k$.

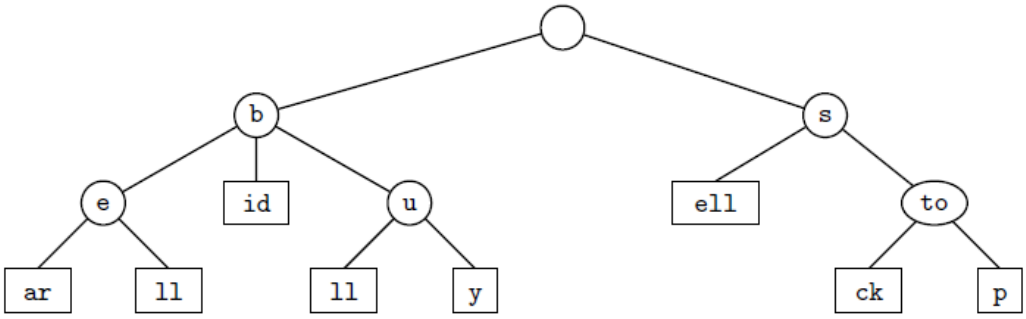


Figura 8.2: Compressed Trie per le stringhe {bear, bell, bid, bull, buy, sell, stock, stop}. (Confronta questo con lo standard trie mostrato in Figura 8.1.) Oltre alla compressione ai nodi foglia, nota il nodo interno con etichetta "to" condivisa dalle parole "stock" e "stop".

I nodi di un compressed trie sono etichettati con delle stringhe, che sono sottostringhe delle stringhe nella collezione, piuttosto che con singoli caratteri. Il vantaggio di un compressed trie rispetto a uno standard trie è che il numero di nodi del compressed trie è proporzionale al numero di stringhe e non alla loro lunghezza totale.

Un compressed trie che memorizza una collezione S di s stringhe in un alfabeto di dimensione d ha le seguenti proprietà:

- Ciascun nodo interno di T ha almeno due figli e al massimo d figli.
- T ha s nodi foglia.
- Il numero di nodi di T è $O(s)$.

Il *Compressed Trie* offre un reale vantaggio quando viene utilizzato come struttura di indicizzazione ausiliaria per una collezione di stringhe già memorizzate in una struttura primaria.

Supponiamo, ad esempio, che la collezione S di stringhe sia un array di stringhe $S[0], S[1], \ldots, S[s-1]$. Invece di memorizzare esplicitamente l'etichetta X di un nodo, la rappresentiamo implicitamente mediante una combinazione di tre interi $(i, j : k)$, tali che $X = S[i][j : k]$; cioè, X è la porzione di $S[i]$ costituita dai caratteri dalla posizione j fino a, ma non includendo, la posizione k.

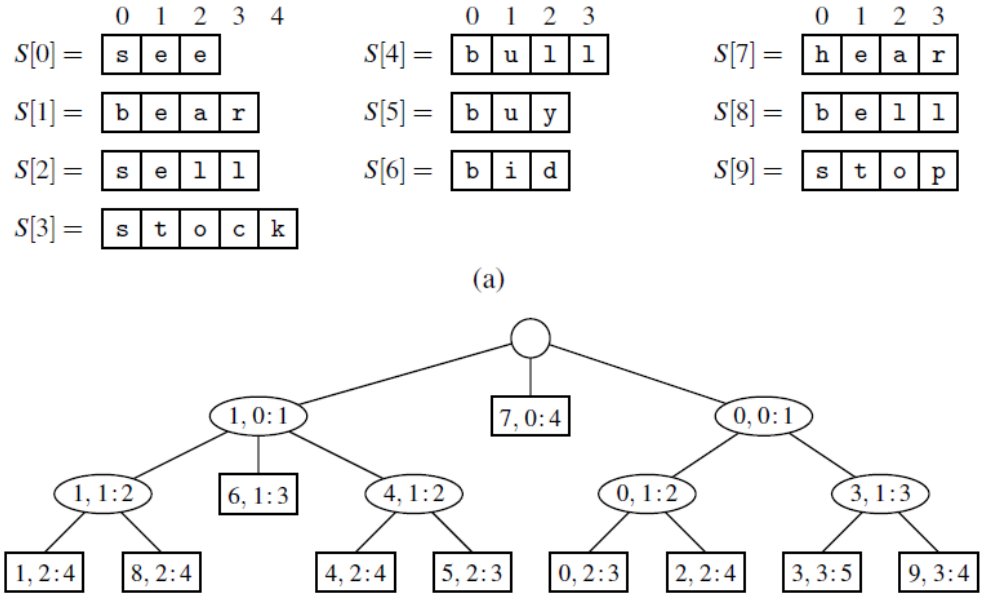


Figura 8.3: (a) Collezione S di stringhe memorizzata in un array. (b) Rappresentazione compatta del compressed trie per S.

In questo modo è possibile ridurre lo spazio totale per il trie stesso da $O(n)$ per lo standard trie a $O(s)$ per il compressed trie, dove n è la lunghezza totale delle stringhe in S e s è il numero di stringhe in S. Naturalmente, dobbiamo comunque memorizzare le diverse stringhe in S, ma riduciamo comunque lo spazio per il trie.

La ricerca in un compressed trie non è necessariamente più veloce che in un albero standard, poiché è ancora necessario confrontare ogni carattere del pattern desiderato con le etichette, potenzialmente multi-carattere, durante la traversata dei percorsi nel trie.

8.3 Suffix Tries

Un **Suffix Trie** per una stringa X è un trie che contiene tutti i suffissi di X.

Per un Suffix Trie, la rappresentazione compatta presentata nella sezione precedente può essere ulteriormente semplificata. In particolare, l'etichetta di ogni vertice è una coppia (j, k) che indica la stringa $X[j : k]$. Al fine di soddisfare la regola che nessun suffisso di X sia prefisso di un altro suffisso, possiamo aggiungere un carattere speciale, denotato con \$ (che non è nell'alfabeto originale Σ) alla fine di X (e quindi a ogni suffisso).

Al netto di tutto ciò, se la stringa X ha lunghezza n, costruiamo un trie per l'insieme di n stringhe $X[j : n]$, per $j = 0, \ldots, n-1$.

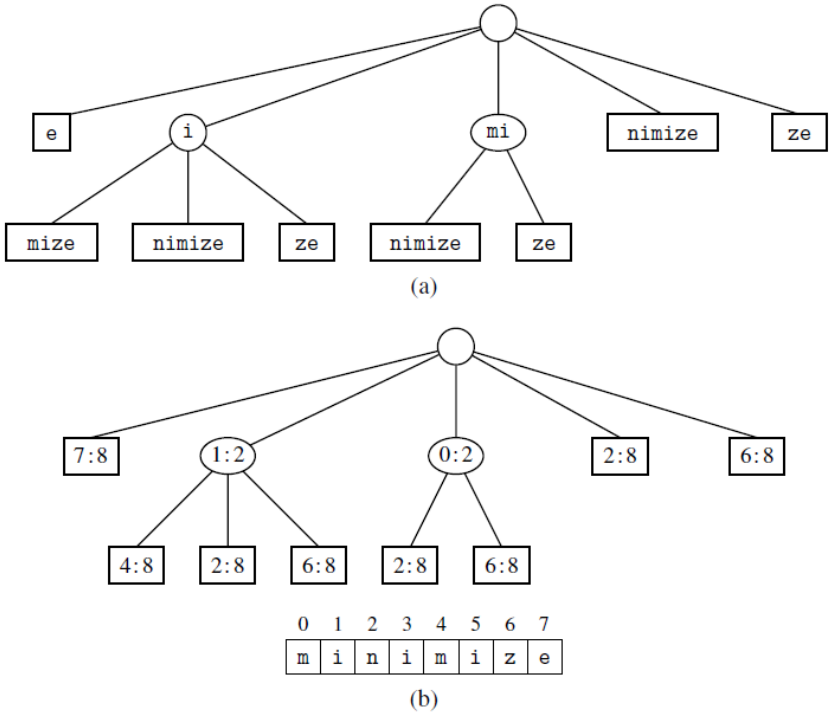


Figura 8.4: (a) Suffix trie T per la stringa X = "minimize". (b) Rappresentazione compatta di T, dove la coppia $(j : k)$ denota la porzione $X[j : k]$ nella stringa di riferimento.

Per costruire un Suffix Trie possiamo procedere in modo simile alla costruzione di uno Standard Trie (compresso). In breve, si segue il cammino esistente finché coincide; dove il cammino finisce o diverge si crea una diramazione.

La lunghezza totale dei suffissi di una stringa X di lunghezza n è pari a:

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2} = O(n^2)$$

Nonostante ciò, il suffix trie può essere costruito in spazio O(n) poiché molte porzioni dei suffissi condividono prefissi comuni. La costruzione di un Suffix Trie richiede un tempo $O(n^2)$, anche se in realtà esiste un algoritmo più complesso (che non approfondiremo) che costruisce un Suffix Trie in tempo $O(n)$.

8.4 Pattern Matching con Suffix Tries