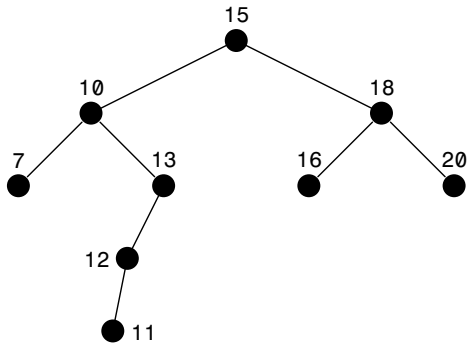
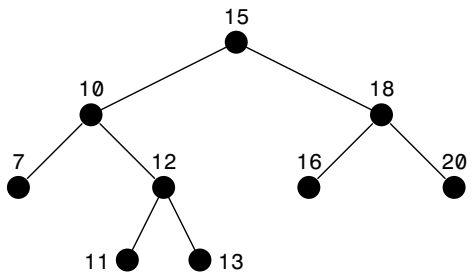


Segue una rotazione oraria su u (riga 8).



Quest'ultima operazione restituisce l'albero 1-bilanciato.

Per la cancellazione, possiamo trattare dei casi simili a quelli dell'inserimento, solo che possono esserci più nodi critici tra gli antenati del nodo cancellato: preferiamo quindi marcare logicamente i nodi cancellati, che vengono ignorati ai fini della ricerca. Quando una frazione costante dei nodi sono marcati come cancellati, ricostruiamo l'albero con le sole chiavi valide ottenendo un costo ammortizzato di `O(log n)`. Possiamo trasformare il costo ammortizzato in un costo al caso pessimo interlacciando la ricostruzione dell'albero, che produce una copia dell'albero attuale, con le operazioni normalmente eseguite su quest'ultimo.

Nonostante siano stati concepiti diverso tempo fa, gli alberi AVL sono tuttora molto competitivi rispetto a strutture di dati più recenti: la maggior parte delle rotazioni avvengono negli ultimi due livelli di nodi, per cui gli alberi AVL risultano molto efficienti se implementati opportunamente (all'atto pratico, la loro forma è molto vicina a quella di un albero completo e quasi perfettamente bilanciato).

4.5 Opus libri: liste invertite e trie

Nei sistemi di recupero dei documenti (**information retrieval**), i dati sono documenti testuali e il loro contenuto è relativamente stabile: pertanto, in tali sistemi lo scopo principale è quello di fornire una risposta veloce alle numerose interrogazioni degli utenti (contrariamente alle basi di dati che sono progettate per garantire

un alto flusso di transazioni che ne modificano i contenuti). In tal caso, la scelta adottata è quella di elaborare preliminarmente l'archivio dei documenti per ottenere un indice in cui le ricerche siano molto veloci, di fatto evitando di scandire l'intera collezione dei documenti a ogni interrogazione (come vedremo, i motori di ricerca sul Web rappresentano un noto esempio di questa strategia). Infatti, il tempo di calcolo aggiuntivo che è richiesto nella costruzione degli indici, viene ampiamente ripagato dal guadagno in velocità di risposta alle innumerevoli richieste che pervengono a tali sistemi.

Le **liste invertite** (chiamate anche file invertiti, file dei *posting* o concordanze) costituiscono uno dei cardini nell'organizzazione di documenti in tale scenario. Le consideriamo un'organizzazione logica dei dati piuttosto che una specifica struttura di dati, in quanto le componenti possono essere realizzate utilizzando strutture di dati differenti. La nozione di liste invertite si basa sulla possibilità di definire in modo preciso la nozione di **termine** (inteso come una parola o una sequenza massimale alfanumerica), in modo da partizionare ciascun documento o **testo** `T` della collezione di documenti `D` in segmenti disgiunti corrispondenti alle occorrenze dei vari termini (quindi le occorrenze di due termini non possono sovrapporsi in `T`). La struttura delle liste invertite è infatti riconducibile alle seguenti due componenti (ipotizzando che il testo sia visto come una sequenza di caratteri):

- il vocabolario o lessico `V`, contenente l'insieme dei termini distinti che appaiono nei testi di `D`;
- la lista invertita L_P (detta dei *posting*) per ciascun termine $P \in V$, contenente un riferimento alla posizione iniziale di ciascuna occorrenza di `P` nei testi $T \in D$: in altre parole, la lista per `P` contiene la coppia $\langle T, i \rangle$ se il segmento `T[i, i + |P| - 1]` del testo è proprio uguale a `P` (ogni documento $T \in D$ ha un suo identificatore numerico che lo contraddistingue dagli altri documenti in `D`).

Notiamo che le liste invertite sono spesso mantenute in forma compressa per ridurre lo spazio a circa il 10-25% del testo e, inoltre, le occorrenze sono spesso riferite al numero di linea, piuttosto che alla posizione precisa nel testo. Solitamente, la lista invertita L_P memorizza anche la sua lunghezza in quanto rappresenta la frequenza del termine `P` nei documenti in `D`. Per completezza, osserviamo che esistono metodi alternativi alle liste invertite come le *bitmap* e i *signature file*, ma osserviamo anche che tali metodi non risultano superiori alle liste invertite come prestazioni.

La realizzazione delle liste invertite prevede l'utilizzo di un dizionario (tabella hash o albero di ricerca) per memorizzare i termini $P \in V$: nello specifico, ciascun elemento `e` memorizzato nel dizionario ha un distinto termine $P \in V$ contenuto nel campo `e.chiave` e ha un'implementazione della corrispondente lista L_P nel campo `e.sat`. Nel seguito, ipotizziamo quindi che `e.sat` sia una lista doppia che fornisce le operazioni descritte nel Paragrafo 4.2; inoltre, ipotizziamo che un termine sia una sequenza massimale alfanumerica nel testo dato in ingresso.

Il Codice 4.9 descrive come costruire le liste invertite usando un dizionario per memorizzare i termini distinti, secondo quanto abbiamo osservato sopra. Lo scopo è quello di identificare una sequenza alfanumerica massima rappresentata dal segmento di testo `T[i, j – 1]` per `i < j` (righe 4-8): tale termine viene cercato nel dizionario (riga 10) e, se appare in esso, la coppia ⟨`T, i`⟩ che ne denota l'occorrenza in `T` viene posta in fondo alla corrispondente lista invertita (riga 12); se invece `T[i, j – 1]` non appare nel dizionario, tale lista viene creata e memorizzata nel dizionario (righe 14-17).

ALVIE **Codice 4.9** Costruzione di liste invertite di un testo `T` ∈ `D` (`elemento` indica un nuovo elemento).

```
1  CostruzioneListeInvertite( T ):      ⟨pre: T ∈ D è un testo di lunghezza n⟩
2    i = 0;
3    WHILE ( i < n ) {
4      WHILE (i < n && !AlfaNumerico( T[i] ))
5        i = i+1;
6      j = i;
7      WHILE (j < n && AlfaNumerico( T[j] ))
8        j = j+1;
9        IF ( i < n ) {
10         e = dizionarioListeInvertite.Ricerca( T[i,j-1] );
11         IF (e != null) {
12           e.sat.InserisciFondo( <T,i> );
13         } ELSE {
14           elemento.chiave = T[i,j-1];
15           elemento.sat = NuovaLista( );
16           elemento.sat.InserisciFondo( <T,i> );
17           dizionarioListeInvertite.Inserisci( elemento )
18         }
19         i = j;
20       }
21   }
```

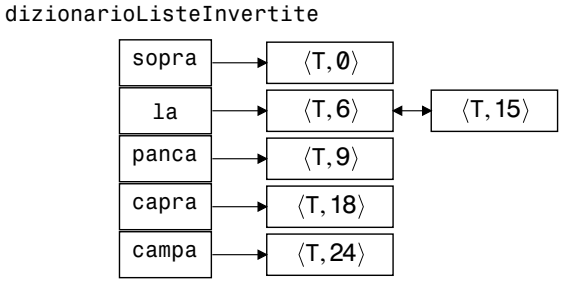
```
1  AlfaNumerico( c ):
2    RETURN ('a' <= c <= 'z' || 'A' <= c <= 'Z' || '0' <= c <= '9');
```

ESEMPIO 4.5

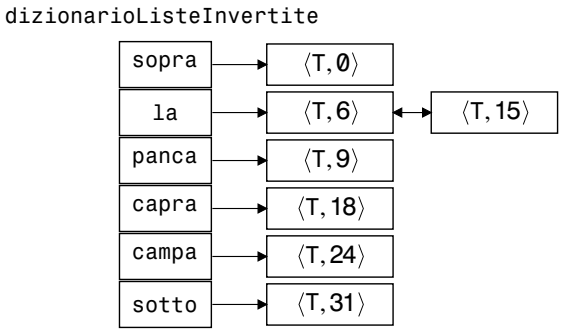
Si consideri il testo T riportato di seguito, i numeri rappresentano la posizione della lettera corrispondente all'interno di T.

```
0     6  9     15 18    24     31    37 40    46 49    55
SOPRA LA PANCA LA CAPRA CAMPA, SOTTO LA PANCA LA CAPRA CREPA.
```

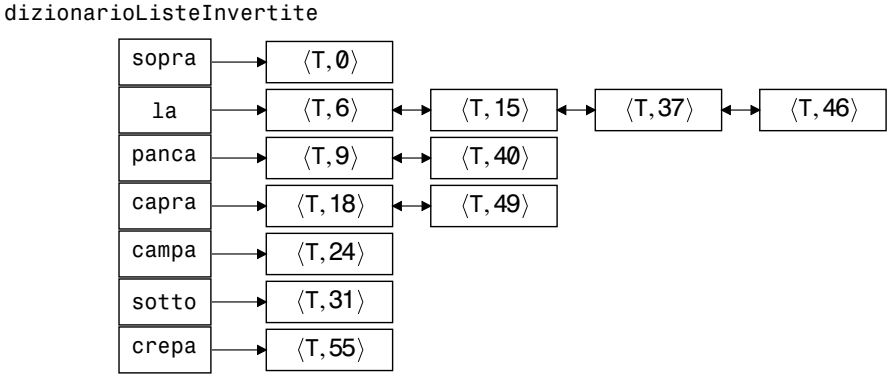
Supponiamo di aver già inserito nel dizionario le prime 6 parole. La situazione di `dizionarioListeInvertite` è rappresentata nella figura che segue.



Dopo aver inserito `campa`, `i = j = 29`. Alla fine del ciclo della riga 4, `i` e `j` puntano al primo carattere alfanumerico di `T` che segue, ovvero `i = j = 31`. Il ciclo successivo sposta `j` nella posizione 36 e quindi viene considerata la sequenza alfanumerica `T[31, 35]`, vale a dire la parola `sotto`. Essa non è presente nel dizionario (riga 9), quindi vi viene aggiunta (righe 13-16) inserendo nel dizionario una nuova lista composta da un unico elemento contenente il valore di ⟨T, i⟩



I quattro termini che seguono sono già presenti nel dizionario, pertanto si aggiungono in fondo alle liste corrispondenti le informazioni sulle posizioni di queste nuove occorrenze. Infine l'ultimo termine, `crepa`, viene aggiunto al dizionario e viene creata la lista composta da un elemento contenente l'informazione su questa unica occorrenza.



Supponendo che n sia il numero totale di caratteri esaminati, il costo della costruzione descritta nel Codice 4.9 è pari a O(n) al caso medio usando le tabelle hash e O(n log n) usando gli alberi di ricerca bilanciati. Nei casi reali, la costruzione è concettualmente simile a quella descritta nel Codice 4.9 ma notevolmente differente nelle prestazioni. Per esempio, dovremmo aspettarci di applicare tale codice a una vasta collezione di documenti che, per la sua dimensione, viene memorizzata nella memoria secondaria.

Ne risulta che, per l'analisi del costo finale, possiamo utilizzare il modello di memoria a due livelli valutando le varie decisioni progettuali: di solito, il vocabolario V è mantenuto nella memoria principale, mentre le liste e i documenti risiedono nella memoria secondaria; tuttavia, diversi motori di ricerca memorizzano anche le liste invertite (ma non i documenti) nella memoria principale utilizzando decine di migliaia di macchine in rete, ciascuna dotata di ampia memoria principale a basso costo.

Per quanto riguarda le interrogazioni effettuate in diversi motori di ricerca, esse prevedono l'uso di termini collegati tra loro mediante gli operatori booleani: l'operatore di congiunzione (AND) tra due termini prevede che entrambi i termini occorrano nel documento; quello di disgiunzione (OR) prevede che almeno uno dei termini occorra; infine, l'operatore di negazione (NOT) indica che il termine non debba occorrere.

È possibile usufruire anche dell'operatore di vicinanza (NEAR) come estensione dell'operatore di congiunzione, in cui viene espresso che i termini specificati occorrano a poche posizioni l'uno dall'altro. Tali interrogazioni possono essere composte come una qualunque espressione booleana, anche se le statistiche mostrano che la maggior parte delle interrogazioni nei motori di ricerca consiste nella ricerca di due termini connessi dall'operatore di congiunzione.

Le liste invertite aiutano a formulare tali interrogazioni in termini di operazioni elementari su liste, supponendo che ciascuna lista L_P sia ordinata. Per esempio, l'interrogazione (P AND Q) altro non è che l'intersezione tra L_P e L_Q, mentre (P OR Q) ne è l'unione senza ripetizioni: entrambe le operazioni possono essere svolte efficientemente con una variante dell'algoritmo di fusione adoperato per il *mergesort*. L'operazione di negazione è il complemento della lista e, infine, l'interrogazione (P NEAR Q) è anch'essa una variante della fusione delle liste L_P e L_Q, come discutiamo ora in dettaglio a scopo illustrativo (le interrogazioni AND e OR sono trattate in modo analogo): a tale scopo, ipotizziamo che le coppie in L_P e L_Q siano in ordine lessicografico crescente.

Il Codice 4.10 descrive come procedere per trovare le occorrenze di due termini P e Q che distano tra di loro per al più maxPos posizioni, facendo uso del Codice 4.11 per verificare tale condizione evitando di esaminare tutte le $O(|L_P| \times |L_Q|)$ coppie di occorrenze: le occorrenze restituite in uscita sono delle triple composte da T, i e j, dove $0 \leq j - i \leq \texttt{maxPos}$, a indicare che $T[i, i+|P|-1] = P$ e $T[j, j+|Q|-1] = Q$ oppure $T[i, i+|Q|-1] = Q$ e $T[j, j+|P|-1] = P$. Notiamo che tali triple sono fornite

in uscita in ordine lessicografico da `VerificaNear`, considerando nell'ordinamento delle triple prima il valore di `T`, poi il minimo tra `i` e `j` e poi il loro massimo: tale ordine lessicografico permette di esaminare ogni testo in ordine di identificatore e di scorrere le occorrenze al suo interno riportate nell'ordine simulato di scansione del testo, fornendo quindi un utile formato di uscita all'utente dell'interrogazione (in quanto non deve saltare da una parte all'altra del testo).

ALVIE Codice 4.10 Algoritmo per la risoluzione dell'interrogazione (`P NEAR Q`), in cui specifichiamo anche il massimo numero di posizioni in cui `P` e `Q` possono distare.

```
1  InterrogazioneNear( P, Q, maxPos ):                ⟨pre: maxPos ≥ 0⟩
2    LP = Ricerca( dizionarioListeInvertite, P );
3    LQ = Ricerca( dizionarioListeInvertite, Q );
4    IF (LP != null && LQ != null) {
5      listaP = LP.sat.inizio;
6      listaQ = LQ.sat.inizio;
7      WHILE (listaP != null && listaQ != null) {
8        <testoP, posP> = listaP.dato;
9        <testoQ, posQ> = listaQ.dato;
10       IF (testoP < testoQ) {
11         listaP = listaP.succ;
12       } ELSE IF (testoP > testoQ) {
13         listaQ = listaQ.succ;
14       } ELSE IF (posP <= posQ) {
15         VerificaNear( listaP, listaQ, maxPos );
16         listaP = listaP.succ;
17       } ELSE {
18         VerificaNear( listaQ, listaP, maxPos );
19         listaQ = listaQ.succ;
20       }
21     }
22   }
```

La funzione `InterrogazioneNear` nel Codice 4.10 provvede quindi a cercare `P` e `Q` nel vocabolario utilizzando il dizionario e supponendo che i testi `T` nella collezione sia identificati da interi (righe 2 e 3). Nel caso che le liste invertite L_P e L_Q siano entrambe non vuote, il codice provvede a scandirle come se volesse eseguire una fusione di tali liste (righe 4-22): ricordiamo che ciascuna lista è composta da una coppia che memorizza l'identificatore del testo e la posizione all'interno del testo dell'occorrenza del termine corrispettivo (`P` o `Q`).

Di conseguenza, se i testi sono diversi nel nodo corrente delle due liste, la funzione avanza di una posizione sulla lista che contiene il testo con identificatore minore (righe 10-13). Altrimenti, i testi sono i medesimi per cui essa deve produrre

tutte le occorrenze vicine usando `VerificaNear` e distinguendo il caso in cui l'occorrenza di `P` sia precedente a quella di `Q` o viceversa (righe 14-19).

ALVIE **Codice 4.11** Algoritmo di verifica di vicinanza per l'interrogazione (`P NEAR Q`).

```
1  VerificaNear( listaX, listaY, M ):                ⟨pre: posX ≤ posY⟩
2    <testoX, posX> = listaX.dato;
3    <testoY, posY> = listaY.dato;
4    WHILE (listaY != null && testoX == testoY && posY-posX <= M) {
5      PRINT testoX, posX, posY;
6      listaY = listaY.succ;
7      <testoY, posY> = listaY.dato;
8    }
```

ESEMPIO 4.6

Consideriamo il dizionario delle liste invertite dell'Esempio 5.5 ed eseguiamo `InterrogazioneNear` cercando i termini `la` e `panca` a distanza al più 6. Dopo aver individuato le liste relative ai due termini (righe 5-6), poiché il testo `T` è lo stesso per tutti i termini del dizionario, viene eseguita la riga 15 in quanto la prima occorrenza del termine `la` precede la prima occorrenza del termine `panca`. La funzione `VerificaNear` cerca nella lista invertita relativa a `panca` tutte le occorrenze a distanza al più 6 dalla prima occorrenza di `la`: in questo caso stampa solo la tripla `T, 6, 9` ed esce perché la successiva occorrenza del temine `panca` (posizione 40) è a distanza maggiore di 6 dal termine `la`. Quando il controllo torna alla funzione `InterrogazioneNear`, viene fatto avanzare il puntatore sulla lista invertita relativa al primo termine `la` e viene invocata per la seconda volta la funzione `VerificaNear`: questa volta, dato che l'occorrenza attuale del termine `panca` precede l'occorrenza attuale del termine `la`, si ricercano sulla lista relativa al termine `la` tutte le occorrenze a distanza al più 6 dall'occorrenza attuale di `panca`, ovvero verrà stampata la tripla `T, 9, 15`. Procedendo con l'algoritmo, verranno stampate anche le triple `T, 37, 40` e `T, 40, 46`.

Per la complessità notiamo che, se `r` indica il numero di triple che soddisfano l'interrogazione di vicinanza e che, quindi, sono fornite in uscita dal Codice 4.10, il tempo totale impiegato da esso è pari a $O(|L_P| + |L_Q| + r)$ e quindi dipende dalla quantità `r` di risultati forniti in uscita (*output sensitive*). Tale codice può essere modificato in modo da sostituire la verifica di distanza sulle posizioni con quella sulle linee del testo.

Gli altri tipi di interrogazione sono trattati in modo analogo a quella per vicinanza. Per esempio, l'interrogazione con la congiunzione (`AND`) calcolata come intersezione di L_P e L_Q richiede tempo $O(|L_P| + |L_Q|)$: da notare però, che se memorizziamo tali liste usando degli alberi di ricerca, possiamo effettuare l'intersezione attraverso una ricerca di ogni elemento della lista più corta tra L_P e L_Q nell'albero che memorizza la più lunga. Quindi, se $m = \min\{|L_P|, |L_Q|\}$ e $n = \max\{|L_P|, |L_Q|\}$, il costo è pari a $O(m \log n)$ e che, quando m è molto minore di n,

risulta inferiore al costo $O(m + n)$ stabilito sopra. In generale, il costo ottimo per l'intersezione è pari $O(m \log (n/m))$ che è sempre migliore sia di $O(m \log n)$ che di $O(m + n)$. Possiamo ottenere tale costo utilizzando gli alberi che permettono la *finger search*, in quanto ognuna delle m ricerche richiede $O(\log d)$ tempo invece che $O(\log n)$ tempo, dove $d < n$ è il numero di chiavi che intercorrono, in ordine di visita anticipata, tra il nodo a cui perveniamo con la chiave attualmente cercata e il nodo a cui siamo pervenuti con la precedente chiave cercata: al caso pessimo, abbiamo che le m ricerche conducono a m nodi equidistanti tra loro, per cui $d = \Theta(n/m)$ massimizza tale costo cumulativo fornendo $O(m \log (n/m))$ tempo.

Tale costo motiva la strategia di risoluzione delle interrogazioni che vedono la congiunzione di $t > 2$ termini (invece che di soli due): prima ordiniamo le t liste invertite dei termini in ordine crescente di frequenza (pari alla loro lunghezza); poi, calcoliamo l'intersezione tra le prime due liste e, per $3 \leq i \leq t$, effettuiamo l'intersezione tra l'i-esima lista e il risultato calcolato fino a quel momento con le prime $i - 1$ liste in ordine non decrescente di frequenza. Nel considerare anche le espressioni in disgiunzione (OR), procediamo come prima per ordine di frequenza delle liste, utilizzando una stima basata sulla somma delle loro lunghezze.

Notiamo, infine, che alternativamente le liste invertite possono essere mantenute ordinate in base a un ordine o rango di importanza delle occorrenze. Se tale ordine è preservato in modo coerente in tutte le liste, possiamo procedere come sopra con il vantaggio di dover esaminare solo i primi elementi di ciascuna lista invertita, in quanto la scansione può terminare non appena viene raggiunto un numero sufficiente di occorrenze. Queste, infatti, avranno necessariamente importanza maggiore di quelle che questo metodo tralascia. In tal modo, possiamo mediamente evitare di esaminare tutti gli elementi delle liste invertite.

4.5.1 Trie o alberi digitali di ricerca

Per realizzare il vocabolario V con delle liste invertite abbiamo utilizzato finora le tabelle hash oppure gli alberi di ricerca. Nel seguito modelliamo i termini da memorizzare in V come stringhe di lunghezza variabile, ossia come sequenze di simboli o caratteri presi da un alfabeto Σ di σ simboli, dove σ è un valore prefissato che non dipende dalla lunghezza e dal numero delle stringhe prese in considerazione (per esempio, $\sigma = 256$ per l'alfabeto ASCII mentre $\sigma = 2^{32}$ per l'alfabeto UNICODE). Ne deriva che ciascuna delle operazioni di un dizionario per una stringa di m caratteri richiede tempo medio $O(m)$ con le tabelle hash oppure tempo $O(m \log n)$ con gli alberi di ricerca, dove $n = |V|$: in quest'ultimo caso, abbiamo $O(\log n)$ confronti da eseguire e ciascuno di essi richiede tempo $O(m)$.

Mostriamo come ottenere un dizionario che garantisce tempo $O(m)$ per operazione utilizzando una struttura di dati denominata trie o albero digitale di ricerca, largamente impiegata per memorizzare un insieme $S = \{a_0, a_1, \ldots, a_{n-1}\}$ di n stringhe. Il termine trie si pronuncia come la parola inglese *try* e deriva dalla

parola inglese *retrieval* utilizzata per descrivere il recupero delle informazioni. I trie hanno innumerevoli applicazioni in quanto permettono di estendere la ricerca in un dizionario ai prefissi della stringa cercata: in altre parole, oltre a verificare se una data stringa `P` appare in `S`, essi permettono di trovare il più lungo **prefisso** di `P` che appare come prefisso di una delle stringhe in `S` (tale operazione non è immediata con le tabelle hash e con gli alberi di ricerca). A tal fine, definiamo il prefisso `i` della stringa `P` di lunghezza `m`, come la sua parte iniziale `P[0, ..., i]`, dove $0 \leq i \leq m - 1$. Per esempio, la ricerca del prefisso `ve` nell'insieme `S` composto dai nomi di alcune province italiane, ovvero `catania`, `catanzaro`, `pisa`, `pistoia`, `verbania`, `vercelli` e `verona`, fornisce come risultato le stringhe `verbania`, `vercelli` e `verona`.

Il **trie** per un insieme $S = \{a_0, a_1, \ldots, a_{n-1}\}$ di n stringhe definite su un alfabeto Σ è un albero cardinale σ-ario (Paragrafo 1.4.2), i cui nodi hanno σ figli (vuoti o meno), e la cui seguente definizione è ricorsiva in termini di `S`:

1. se l'insieme `S` delle stringhe è vuoto, il trie corrispondente a `S` è vuoto e viene rappresentato con `null`;
2. se `S` non è vuoto, sia S_c l'insieme delle stringhe in `S` aventi `c` come carattere iniziale, dove $c \in \Sigma$ (ai soli fini della definizione ricorsiva del trie, il carattere iniziale `c` comune alle stringhe in S_c viene ignorato e temporaneamente rimosso): il trie corrispondente a `S` è quindi composto da un nodo `u` chiamato radice tale che `u.figlio[c]` memorizza il trie ricorsivamente definito per S_c (alcuni dei figli possono essere vuoti).

Per poter realizzare un dizionario, ciascun nodo `u` di un trie è dotato di un campo `u.dato` in cui memorizzare un elemento `e`: ricordiamo che `e` ha un campo di ricerca `e.chiave`, che in questo scenario è una stringa, e un campo `e.sat` per i dati satellite, come specificato in precedenza. Mostriamo, nella parte sinistra della Figura 4.4, il trie corrispondente all'insieme `S` di stringhe composto dai sette nomi di provincia menzionati precedentemente.

Ogni stringa in `S` corrisponde a un nodo `u` del trie ed è quindi memorizzata nel campo `u.dato.chiave` (e gli eventuali dati satellite sono in `u.dato.sat`): per esempio, la foglia 5 corrisponde a `vercelli` come mostrato nella Figura 4.4, dove le foglie del trie sono numerate da `0` a `6`, e la foglia numero `i` memorizza il nome della `(i+1)`-esima provincia ($0 \leq i \leq 6$).

In generale, estendendo tutte le stringhe con un simbolo speciale, da appendere in fondo a esse come terminatore di stringa, e memorizzando le stringhe così estese nel trie, otteniamo la proprietà che queste stringhe sono associate in modo univoco alle foglie del trie (questa estensione non è necessaria se nessuna stringa in `S` è a sua volta prefisso di un'altra stringa in `S`).

Notiamo che l'altezza di un trie è data dalla lunghezza massima tra le stringhe. Nel nostro esempio, l'altezza del trie è `9` in quanto la stringa più lunga nell'insieme è `catanzaro`. Potando i sottoalberi che portano a una sola foglia, come mostrato

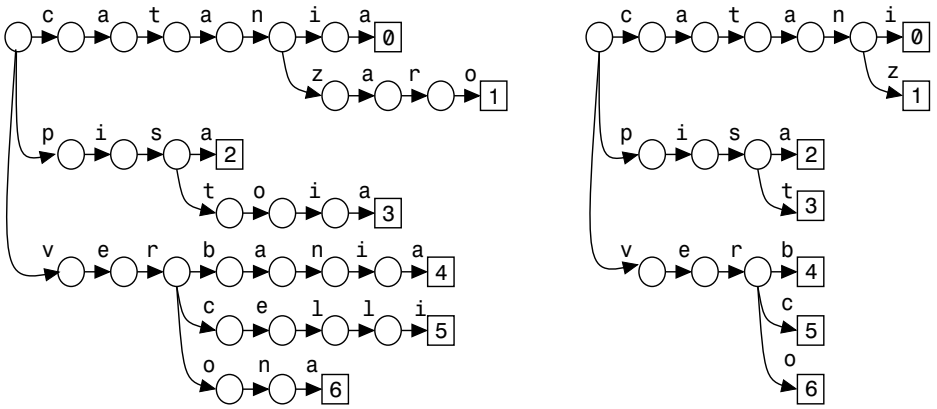


Figura 4.4 Trie per i nomi di alcune province (a sinistra) e sua versione potata (a destra).

nella parte destra della Figura 4.4, l'altezza media non dipende dalla lunghezza delle stringhe, ma è limitata superiormente da $2 \log_\sigma n + O(1)$, dove n è il numero di stringhe nell'insieme S. Tale potatura presume che le stringhe siano memorizzate altrove, in modo da poter ricostruire il sottoalbero potato in quanto è un filamento di nodi contenente la sequenza di caratteri finali di una delle stringhe (alternativamente, tali caratteri possono essere memorizzati nella foglia ottenuta dalla potatura).

Per quanto riguarda la dimensione del trie, indicando con N la lunghezza totale delle n stringhe memorizzate in S, ovvero la somma delle loro lunghezze, abbiamo che la dimensione di un trie è al più $N + 1$. Tale valore viene raggiunto quando ciascuna stringa in S ha il primo carattere differente da quello delle altre, per cui il trie è composto da una radice e poi da $|S|$ filamenti di nodi, ciascun filamento corrispondente a una stringa. Tuttavia, se si adotta la versione potata del trie, la dimensione al caso medio non dipende dalla lunghezza delle stringhe e vale all'incirca $1{,}44n$.

Come possiamo notare dall'esempio nella Figura 4.4, i nodi del trie rappresentano prefissi delle stringhe memorizzate, e le stringhe che sono memorizzate nei nodi discendenti da un nodo u hanno in comune il prefisso associato a u. Nel nostro esempio, le foglie discendenti dal nodo che memorizza il prefisso `pi` hanno associate le stringhe `pisa` e `pistoia`.

Sfruttiamo questa proprietà per implementare in modo semplice la ricerca di una stringa di lunghezza m in un trie utilizzando la sua definizione ricorsiva, come mostrato nel Codice 4.12: partiamo dalla radice per decidere quale figlio scegliere a livello $i = 0$ e, al generico passo in cui dobbiamo scegliere un figlio a livello i del nodo corrente u, esaminiamo il carattere $P[i]$ della stringa da cercare. Osserviamo che, se il corrispondente figlio non è vuoto, continuiamo la ricerca portando il livello a $i + 1$. Se invece tale figlio è vuoto, possiamo concludere che P non è

memorizzata nel dizionario. Quando `i = m`, abbiamo esaminato tutta la stringa `P` con successo pervenendo al nodo `u` di cui restituiamo l'elemento contenuto nel campo `u.dato`: in tal caso, osserviamo che `P` è memorizzato nel dizionario se e solo se tale campo è diverso da `null`.

ALVIE Codice 4.12 Algoritmo di ricerca in un trie.

```
1  Ricerca( radiceTrie, P ):          ⟨pre: P è una stringa di lunghezza m⟩
2    u = radiceTrie;
3    FOR (i = 0; i < m; i = i+1) {
4      IF (u.figlio[ P[i] ] != null) {
5        u = u.figlio[ P[i] ];
6      } ELSE {
7        RETURN null;
8      }
9    }
10   RETURN u.dato;
```

La parte interessante della ricerca mostrata nel Codice 4.12 è che, a differenza della ricerca mostrata per le tabelle hash e per gli alberi di ricerca, essa può facilmente identificare il nodo `u` che corrisponde al *più lungo* prefisso di `P` che appare nel trie: a tal fine, possiamo modificare la riga 7 del codice in modo che restituisca il nodo `u` (al posto di `null`).

Di conseguenza, tutte le stringhe in `S` che hanno tale prefisso di `P` come loro prefisso possono essere recuperate nei nodi che discendono da `u` (incluso) attraverso una semplice visita, ottenendo il Codice 4.13: notiamo che le ricerche di stringhe lunghe, quando queste ultime non compaiono nel dizionario, sono generalmente più veloci delle corrispondenti ricerche nei dizionari implementati con le tabelle hash, poiché la ricerca nei trie si ferma non appena trova un prefisso di `P` che non occorre nel trie, mentre la funzione hash è comunque calcolata sull'intera stringa prima di effettuare la ricerca.

Non è difficile analizzare il costo della ricerca, in quanto vengono visitati `O(m)` nodi: poiché ogni nodo richiede tempo costante, abbiamo `O(m)` tempo di ricerca.

ALVIE Codice 4.13 Algoritmo di ricerca per prefissi in un trie (`numStringhe` è una variabile globale).

```
1  RicercaPrefissi( radiceTrie, P ):  ⟨pre: P è una stringa di lunghezza m⟩
2    u = radiceTrie;
3    fine = false;
4    FOR (i = 0; !fine && i < m; i = i+1) {
5      IF (u.figlio[ P[i] ] != null) {
6        u = u.figlio[ P[i] ];
```

```
7      } ELSE {
8        fine = true;
9      }
10   }
11   numStringhe = 0;
12   Recupera( u, elenco );
13   RETURN elenco;
```

```
1  Recupera( u, elenco ):
2  IF (u != null) {
3    IF (u.dato != null) {
4      elenco[numStringhe]= u.dato;
5      numStringhe = numStringhe + 1;
6    }
7    FOR (c = 0; c < sigma; c = c + 1)
8      Recupera( u.figlio[c], elenco );
9  }
```

ESEMPIO 4.7

Come esempio quantitativo sulla velocità di ricerca dei trie, supponiamo di volere memorizzare i codici fiscali in un trie: ricordiamo che un codice fiscale contiene 9 lettere prese dall'alfabeto A … Z di 26 caratteri, e 7 cifre prese dall'alfabeto 0 … 9 di 10 caratteri, per un totale di $26^9 \times 10^7$ possibili codici fiscali. Cercare un codice fiscale in un trie richiede di attraversare al più 16 nodi, *indipendentemente* dal numero di codici fiscali memorizzati nel trie, in quanto l'altezza del trie è 16. In contrasto, la ricerca binaria o quella in un albero AVL, per esempio, avrebbe una dipendenza dal numero di chiavi: nel caso estremo, memorizzando metà dei possibili codici fiscali in un array ordinato, tale ricerca richiederebbe circa $\log(26^9 \times 10^7) - 1 \geq 64$ confronti tra chiavi. Il trie è quindi una struttura di dati molto efficiente per la ricerca di sequenze.

L'inserimento di un nuovo elemento nel dizionario rappresentato con un trie segue nuovamente la sua definizione ricorsiva, come mostrato nel Codice 4.14: se il trie è vuoto viene creata una radice (righe 2-7) e, quindi, dopo aver verificato che la chiave dell'elemento non appaia nel dizionario (riga 8), il trie viene attraversato fino a trovare un figlio vuoto in cui inserire ricorsivamente il trie per il resto dei caratteri (righe 14-18) oppure il nodo esiste già ma l'elemento da inserire deve essergli associato (riga 21).

ALVIE **Codice 4.14** Algoritmo di inserimento di un elemento in un trie.

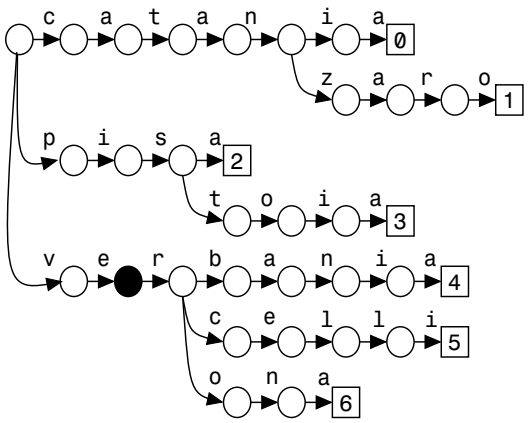
```
1  Inserisci( radiceTrie, e ):              ⟨pre: e.chiave ha lunghezza m⟩
2    IF (radiceTrie == null) {
3      radiceTrie = NuovoNodo( );
4      radiceTrie.dato = null;
5      FOR (c = 0; c < sigma; c = c + 1)
6        radiceTrie.figlio[c] = null;
7    }
8    IF (Ricerca( radiceTrie, e.chiave ) == null) {
9      u = radiceTrie;
10     FOR (i = 0; i < m; i = i+1) {
11       IF (u.figlio[ e.chiave[i] ] != null) {
12         u = u.figlio[ e.chiave[i] ];
13       } ELSE {
14         u.figlio[ e.chiave[i] ] = NuovoNodo( );
15         u = u.figlio[ e.chiave[i] ];
16         u.dato = null;
17         FOR (c = 0; c < sigma; c = c + 1)
18           u.figlio[c] = null;
19       }
20     }
21     u.dato = e;
22   }
23   RETURN radiceTrie;
```

In altre parole, l'inserimento della stringa P cerca prima il suo più lungo prefisso x che occorre in un nodo u del trie, analogamente alla procedura Ricerca, e scompone P come xy: se y non è vuoto sostituisce il sottoalbero vuoto raggiunto con la ricerca di x, mettendo al suo posto il trie per y; altrimenti, semplicemente associa P al nodo u identificato.

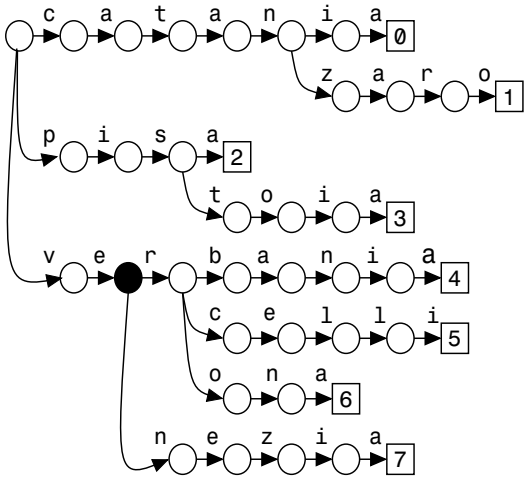
Il costo dell'inserimento è tempo O(m) in accordo a quanto discusso per la ricerca (e la cancellazione può essere trattata in modo analogo): da notare che non occorrono operazioni di ribilanciamento (rotazioni o divisioni) come succede negli alberi di ricerca. Infatti, un'importante proprietà è che la forma del trie è determinata univocamente dalle stringhe in esso contenute e non dal loro ordine di inserimento (contrariamente agli alberi di ricerca).

ESEMPIO 4.8

Consideriamo l'operazione di inserimento del termine `venezia` nel trie rappresentato nella parte sinistra della Figura 4.4. Dopo aver raggiunto in nodo `u` corrispondente al prefisso `ve` (rappresentato da un cerchio pieno nella figura che segue) viene riscontrato che `u.figlio[n]` è `null` (riga 11).



Quindi viene creato un nuovo filamento contenente i caratteri di `nezia` collegato al nodo `u` (righe 14-18).



Inoltre, poiché i trie preservano l'ordine lessicografico delle stringhe in esso contenute, possiamo fornire un semplice metodo per ordinare un array `S` di stringhe come mostrato nel Codice 4.15, in cui adoperiamo la funzione `Recupera` del Codice 4.13 per effettuare una visita anticipata del trie costruito attraverso l'inserimento iterativo delle stringhe contenute nell'array `S`.

ALVIE **Codice 4.15** Algoritmo di ordinamento di stringhe (fa uso di una variabile globale `numStringhe` e del Codice 4.13).

```
1  OrdinaLessicograficamente( S ):          ⟨pre: S è un array di n stringhe⟩
2    radiceTrie = null;
3    elemento.sat = null;
4    FOR (i = 0; i < n; i = i+1) {
5      elemento.chiave = S[i];
6      radiceTrie = Inserisci( radiceTrie, elemento );
7    }
8    numStringhe = 0;
9    Recupera( radiceTrie, S );
10   RETURN S;
```

La complessità dell'ordinamento proposto nel Codice 4.15 è $O(N)$ tempo se la somma delle lunghezze delle n stringhe in S è pari a N. Un algoritmo ottimo per confronti, come lo heapsort, impiegherebbe $O(n \log n)$ confronti, dove però ciascun confronto richiederebbe di accedere mediamente a N/n caratteri di una stringa, per un totale di $O\left(\frac{N}{n} n \log n\right) = O(N \log n)$ tempo: l'ordinamento di stringhe con un trie è quindi più efficiente in tal caso. Analogamente a quanto discusso per la ricerca in tabelle hash, il limite così ottenuto non contraddice il limite inferiore (in questo caso sull'ordinamento) poiché i caratteri negli elementi da ordinare sono utilizzati per altre operazioni oltre ai confronti.

ESEMPIO 4.9

La visita anticipata del trie finale dell'Esempio 5.8 produce la seguente sequenza: `catania`, `catanzaro`, `pisa`, `pistoia`, `venezia`, `verbania`, `vercelli` e `verona`. Si osservi che la visita anticipata, una volta giunta sul nodo nero, segue prima l'arco etichettato con `n` e poi quello etichettato con `r`.

Nati per ricerche come quelle discusse finora, i trie sono talmente flessibili da risultare utili per altri tipi di ricerche più complesse. Inoltre, sono utilizzati nella compressione dei dati, nei compilatori e negli analizzatori sintattici. Servono a completare termini specificati solo in parte; per esempio, i comandi nella shell, le parole nella composizione dei testi, i numeri telefonici e i messaggi SMS nei telefoni cellulari, gli indirizzi del Web o della posta elettronica. Permettono la realizzazione efficiente di correttori ortografici, di analizzatori di linguaggio naturale e di sistemi per il recupero di informazioni mediante basi di conoscenza. Forniscono operazioni di ricerca più complesse di quella per prefissi, come la ricerca con espressioni regolari e con errori. Permettono di individuare ripetizioni nelle stringhe (utili, per esempio, nell'analisi degli stili di scrittura di vari autori)

e di recuperare le stringhe comprese in un certo intervallo. Le loro prestazioni ne hanno favorito l'impiego anche nel trattamento di dati multidimensionali, nell'elaborazione dei segnali e nelle telecomunicazioni. Per esempio sono utilmente impiegati nella codifica e decodifica dei messaggi, nella risoluzione dei conflitti nell'accesso ai canali e nell'instradamento veloce dei router in Internet. A fronte della loro duttilità, i trie hanno una struttura sorprendentemente semplice.

Purtroppo essi presentano alcuni svantaggi dal punto di vista dello spazio occupato per alfabeti grandi, in quanto ciascuno dei loro nodi richiede l'allocazione di un array di σ elementi: inoltre, le loro prestazioni possono peggiorare se il linguaggio di programmazione adottato non rende disponibile un accesso efficiente ai singoli caratteri delle stringhe. Esistono diverse alternative per l'effettiva rappresentazione in memoria di un trie che sono basate sulla rappresentazioni dei suoi nodi mediante strutture di dati alternative agli array come le liste, le tabelle hash e gli alberi binari.

4.5.2 Trie compatti

I trie discussi finora hanno una certa ridondanza nel numero dei nodi, in quanto quelli con un solo figlio non vuoto rappresentano una scelta obbligata e non raffinano ulteriormente la ricerca nei trie, al contrario dei nodi che hanno due o più figli non vuoti. Tale ridondanza è rimossa nel **trie compatto**, mostrato nella parte sinistra della Figura 4.5, dove i nodi con un solo figlio non vuoto sono altrimenti rappresentati per preservare le funzionalità del trie: a tal fine, gli archi sono etichettati utilizzando le sottostringhe delle chiavi appartenenti agli elementi dell'insieme `S`, invece che i loro singoli caratteri.

Formalmente, dato il trie per le chiavi contenute negli elementi dell'insieme `S`, classifichiamo un nodo del trie come **unario** se ha esattamente un figlio non vuoto. Una *catena* di nodi unari è una sequenza massimale $u_0, u_1, \ldots, u_{r-1}$ di $r \geq 2$ nodi nel trie tale che ciascun nodo u_i è unario per $1 \leq i \leq r-2$ (notiamo che u_0 potrebbe essere la radice oppure u_{r-1} potrebbe essere una foglia) e u_{i+1} è figlio di u_i per $0 \leq i \leq r-2$. Sia c_i il carattere per cui $u_i = u_{i-1}.figlio[c_i]$ e $\beta = c_1 \cdots c_{r-1}$ la sottostringa ottenuta dalla concatenazione dei caratteri incontrati percorrendo la catena da u_0 a u_{r-1}: definiamo l'operazione di *compattazione* della catena sostituendo l'intera catena con la coppia di nodi u_0 e u_{r-1} collegati da un *singolo* arco (u_0, u_{r-1}), che è concettualmente etichettato con β.

Notiamo infatti che l'esplicita memorizzazione di β non è necessaria se associamo, a ciascun nodo `u`, il prefisso ottenuto percorrendo il cammino dalla radice fino a `u` (la radice ha quindi un prefisso vuoto): in tal modo, indicando con α il prefisso nel padre di `u` e con γ quello in `u`, possiamo ricavare β per differenza in quanto $\alpha\beta = \gamma$ e la sottostringa β è data dagli ultimi $r - 1 = |\beta|$ caratteri di γ.

Il trie compatto per l'insieme `S` è ottenuto dal trie costruito su `S` applicando l'operazione di compattazione a tutte le catene presenti nel trie stesso. Ne risulta che i nodi del trie compatto sono foglie oppure hanno almeno due figli non vuoti.

Per implementare un trie compatto, estendiamo l'approccio adottato per i trie, utilizzando la rappresentazione degli alberi cardinali σ-ari (Paragrafo 1.4.2): ciascun nodo `u` di un trie compatto è dotato di un campo `u.dato` in cui memorizzare un elemento `e` $\in$ `S` (quindi i campi di `e` sono indicati come `u.dato.chiave` e `u.dato.sat`) a cui aggiungiamo un campo `u.prefisso` per memorizzare il prefisso associato a `u`.

Tale prefisso è memorizzato mediante una coppia $\langle$`e`, `j`$\rangle$ per indicare che esso è dato dai primi `j` caratteri della stringa contenuta nel campo `e.chiave`: il vantaggio è che rappresentiamo ciascun prefisso con soli due interi indipendentemente dalla lunghezza del prefisso stesso, perché lo spazio richiesto da ciascun nodo rimane $O(\sigma)$ (purché i campi chiave degli elementi siano memorizzati a parte). La parte destra della Figura 4.5 mostra un esempio di tale rappresentazione, dove possiamo osservare che un nodo interno `u` appare nel trie compatto se e solo se, prendendo il prefisso α associato a `u`, esistono almeno due caratteri `c` $\neq$ `c'` dell'alfabeto Σ tali che entrambi α`c` e α`c'` sono prefissi delle chiavi di alcuni elementi in `S`.

La presenza di due chiavi, una prefisso dell'altra, potrebbe introdurre nodi unari che non possiamo rimuovere. Per tale ragione, estendiamo tutte le chiavi degli elementi in `S` con un ulteriore carattere `$`, che è un simbolo speciale da usare come terminatore di stringa (analogamente al carattere '`\0`' nel linguaggio C). In tal modo, poiché `$` appare solo in fondo alle stringhe, nessuna può essere prefisso dell'altra e, come osservato in precedenza, esiste una corrispondenza biunivoca tra le `n` chiavi e le `n` foglie del trie compatto costruito su di esse: quindi, presumiamo che ciascuna delle `n` foglie contenga un distinto elemento `e` $\in$ `S` (in particolare, illustriamo questa corrispondenza nella Figura 4.5 etichettando con `i` la foglia

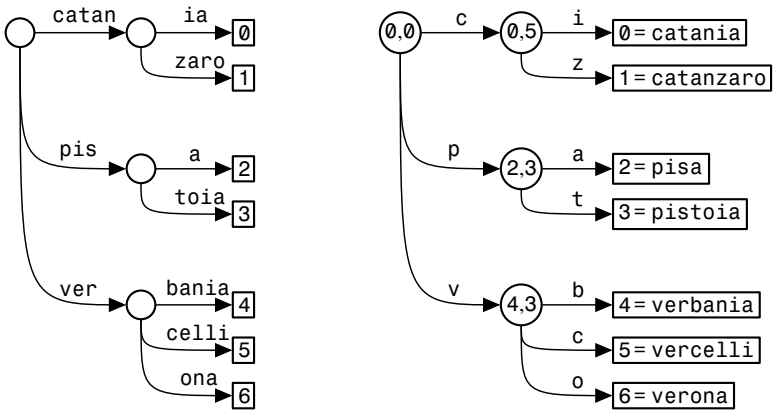


Figura 4.5 A sinistra, il trie compatto per memorizzare i nomi di alcune province; a destra, la versione con le sottostringhe rappresentate mediante coppie. Ogni foglia è associata a un elemento, identificato da un intero da `0` a `n` = 6, il cui campo chiave è una stringa del dizionario. Ogni nodo interno contiene la coppia $\langle$`e`, `i`$\rangle$ che rappresenta i primi `i` caratteri di `e.chiave`. L'etichetta `i` sull'arco `(u, v)` serve a sottolineare che `v` è il figlio `i` di `u`.

contenente e_i). Utilizzando il simbolo speciale e la rappresentazione dei prefissi mediante coppie, il trie compatto ha al più n nodi interni e n foglie, e quindi richiede $O(n\sigma)$ spazio, dove $\sigma = O(1)$ per le nostre ipotesi: lo spazio dipende quindi solo dal numero n delle stringhe nel caso pessimo e non dalla somma N delle loro lunghezze, contrariamente al trie.

La rappresentazione compatta di un trie non ne pregiudica le caratteristiche discusse finora. Per esempio la ricerca per prefissi in un trie compatto simula quella per prefissi in un trie (Codice 4.13) ed è mostrata nel Codice 4.16: la differenza risiede nelle righe 8-11 dove, dopo aver raggiunto il nodo u, ne prendiamo il prefisso a esso associato e ne confrontiamo i caratteri con P, dalla posizione i in poi, fino alla fine di una delle due stringhe oppure quando troviamo due caratteri differenti (riga 9). Analogamente alla ricerca nei trie, terminiamo di effettuare confronti quando tutti caratteri di P sono stati esaminati con successo oppure troviamo il suo più lungo prefisso che occorre nel trie compatto, e il costo del Codice 4.16 rimane pari a tempo O(m) più il numero di occorrenze riportate. L'operazione Ricerca è realizzata in modo analogo a quella per prefissi e mantiene la complessità di tempo O(m).

ALVIE Codice 4.16 Algoritmo di ricerca per prefissi in un trie compatto (fa uso di una variabile globale numStringhe e della funzione Recupera del Codice 4.13).

```
1  RicercaPrefissi( radiceTrieCompatto, P ):   ⟨pre: P contiene m caratteri⟩
2    u = radiceTrieCompatto;
3    fine = false;
4    i = 0;
5    WHILE (!fine && i < m) {
6      IF (u.figlio[ P[i] ] != null) {
7        u = u.figlio[ P[i] ];
8        <e, j> = u.prefisso;
9        WHILE ((i < m) && (i < j) && (P[i] == e.chiave[i]))
10         i = i + 1;
11       fine = (i < m) && (i < j);
12     } else {
13       fine = true;
14     }
15   }
16   numStringhe = 0;
17   Recupera( u, elenco );
```

Analogamente alla ricerca, l'inserimento di un nuovo elemento e nel trie compatto richiede tempo O(m) come mostrato nel Codice 4.17. Dopo aver creato la radice (righe 2-8), se necessario, a cui associamo il prefisso vuoto (di lunghezza 0), verifichiamo che l'elemento non sia già nel dizionario. A questo punto, procediamo

come nel caso della ricerca per prefissi per identificare il più lungo prefisso x della chiave di e che occorre nel trie compatto, dove P = xy. Sia u il nodo raggiunto e v il nodo calcolato nelle righe 11-23 e sia i = |x|: se x coincide con il prefisso associato a u, allora v = u; se invece, x è più breve del prefisso associato a u, allora v è il padre di u. Invochiamo ora CreaFoglia, descritta nel Codice 4.18, che fa la seguente cosa: se u ≠ v, spezza l'arco (u, v) in due creando un nodo intermedio a cui associa x come prefisso (corrispondente ai primi i caratteri di e.chiave) e a cui aggancia la nuova foglia che memorizza l'elemento e, la cui chiave ne diventa il prefisso di lunghezza m (in quanto prendiamo tutti i caratteri di e.chiave); se invece u = v, crea soltanto la nuova foglia come descritto sopra, agganciandola però a u. Ricordiamo che, non essendoci una chiave prefisso di un'altra, ogni inserimento di un nuovo elemento crea sicuramente una foglia.

ALVIE **Codice 4.17** Algoritmo per l'inserimento di un elemento in trie compatto.

```
1  Inserisci( radiceTrieCompatto, e ):    ⟨pre: e.chiave ha lunghezza m⟩
2    IF (radiceTrieCompatto == null) {
3      radiceTrieCompatto = NuovoNodo( );
4      radiceTrieCompatto.prefisso = <e, 0>;
5      radiceTrieCompatto.dato = null;
6      FOR (c = 0; c < sigma; c = c + 1)
7        radiceTrieCompatto.figlio[c] = null;
8    }
9    IF (Ricerca( radiceTrieCompatto, e.chiave ) == null) {
10     u = radiceTrieCompatto; fine = false; i = 0;
11     WHILE (!fine && i < m) {
12       v = u;
13       indice = i;
14       IF (u.figlio[ e.chiave[i] ] != null) {
15         u = u.figlio[ e.chiave[i] ];
16         <p, j> = u.prefisso;
17         WHILE (i < m && i < j && p.chiave[i] == e.chiave[i])
18           i = i + 1;
19         fine = (i < m) && (i < j);
20       } ELSE {
21         fine = true;
22       }
23     }
24     IF (fine) CreaFoglia( v, u, indice, i );
25   }
26   RETURN radiceTrieCompatto;
```

ALVIE Codice 4.18 Algoritmo per la creazione di una foglia e di un eventuale nodo (suo padre).

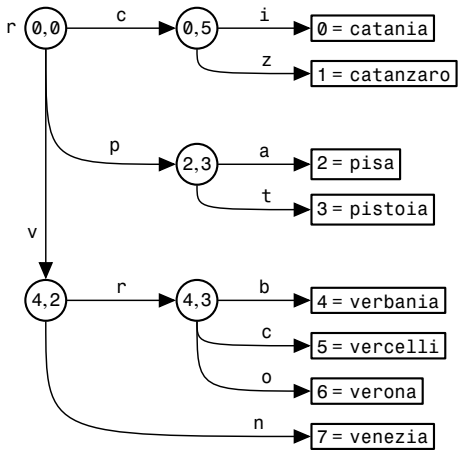
```
1  CreaFoglia ( v, u, indice, i ):        ⟨pre: e.chiave ha lunghezza m⟩
2    IF (v != u) {
3      v.figlio[ e.chiave[indice] ] = NuovoNodo( );
4      v = v.figlio[ e.chiave[indice] ];
5      v.prefisso = <e, i>;
6      v.dato = null;
7      FOR (c = 0; c < sigma; c = c + 1)
8        v.figlio[c] = null;
9      <p, j> = u.prefisso;
10     v.figlio[ p.chiave[i] ] = u;
11     u = v;
12   }
13   u.figlio[ e.chiave[i] ] = NuovoNodo( );
14   u = u.figlio[ e.chiave[i] ];
15   u.prefisso = <e, m>;
16   u.dato = e;
17   FOR (c = 0; c < sigma; c = c + 1)
18     u.figlio[c] = null;
```

Infine, la cancellazione dell'elemento avente chiave uguale a P, specificata in ingresso, richiede la rimozione della foglia raggiunta con la ricerca di P, nonché dell'arco che collega la foglia a suo padre u. Se u diventa unario, allora dobbiamo rimuovere u, il suo arco entrante e il suo arco uscente, sostituendoli con un unico arco la cui etichetta è la concatenazione della sottostringa nell'arco entrante con quella nell'arco uscente.

Tuttavia dobbiamo stare attenti a non usare elementi cancellati per i prefissi associati ai nodi u del trie compatto: se e viene cancellato e un antenato u della corrispondente foglia contiene il prefisso $\langle e, j \rangle$, allora è sufficiente individuare un altro elemento e' contenuto in una foglia che discende da u e sostituire quel prefisso con $\langle e', j \rangle$. Il costo della cancellazione è $O(m)$ poiché $\sigma = O(1)$.

ESEMPIO 4.10

Riprendiamo il trie della Figura 4.4 e consideriamo l'operazione di inserimento dell'elemento avente chiave venezia di lunghezza m = 7. Nel ciclo delimitato dalle righe da 11 a 23 u viene spostato su r.figlio['v'] (si veda la Figura 4.4). Ora, la coppia ⟨4, 3⟩ si riferisce ai primi 3 caratteri della stringa identificata dall'intero 4 (ovvero verbania). Il ciclo nella riga 18 calcola la lunghezza del massimo prefisso comune tra la stringa venezia e ver: questa informazione è contenuta nella variabile i. Quindi viene invocata la funzione Creafoglia con input v = r, u = r.figlio['v'], indice = 0 e i = 2.



Questa crea un nuovo nodo figlio `'v'` della radice con campo `prefisso` dato dalla coppia $\langle 4, 2\rangle$ (il primo elemento della coppia è l'identificativo di e mentre il secondo è il valore di `i`). Il nodo u diventa figlio `'r'` del nodo appena inserito (`r` è il carattere in posizione `i` di `verbania`). Infine viene creata una nuova foglia per il nuovo elemento con chiave `venezia`.

4.6 Esercizi

4.1 Discutere la complessità in tempo delle operazioni dei dizionari quando questi sono realizzati mediante array, array ordinati, liste e liste ordinate.

4.2 Mostrare come estendere i dizionari discussi nel capitolo in modo che possano gestire multi-insiemi con chiavi eventualmente ripetute.

4.3 Inserire la sequenza di chiavi `S = (5, 3, 4, 6, 7, 10)` in una tabella hash inizialmente vuota di dimensione `m = 7`, con indirizzamento aperto e sequenza di probing basata su hash doppio `Hash[i](k) = (Hash(k) + i × Hash′(k)) % m`, specificando quali funzioni `Hash` e `Hash′` si intende utilizzare.

4.4 Descrivere la cancellazione fisica da tabelle `hash` a indirizzamento aperto.

4.5 Si consideri la seguente variante della tabella hash a indirizzamento aperto e scansione lineare che utilizza due funzioni hash $\texttt{Hash}_1\texttt{()}$ e $\texttt{Hash}_2\texttt{()}$ invece che una singola funzione: per l'inserimento di una chiave `k`, se la posizione $\texttt{Hash}_1\texttt{(k)}$ nella tabella è libera, `k` viene inserita in tale posizione; se invece risulta occupata da un'altra chiave `k′`, allora `k` prende il posto di `k′` in tale posizione e l'inserimento continua con l'inserimento di `k′` utilizzando $\texttt{Hash}_2$ (come nella solita scansione lineare). Supponendo di utilizzare una tabella di dimensione `m`:

(a) scrivere lo pseudocodice per l'inserimento descritto sopra;

(b) discutere come cambia l'algoritmo di ricerca.