TRIE (albero digitale di ricerca)

Struttura dati impiegata per memorizzare un insieme S di n stringhe (il vocabolario V).

Tabelle hash

le operazioni di dizionario hanno costo O(m) al caso medio per una stringa di m caratteri

AVL

costo O(m log n)

- O(log n): numero dei confronti
- O(m): costo dei confronti

TRIE

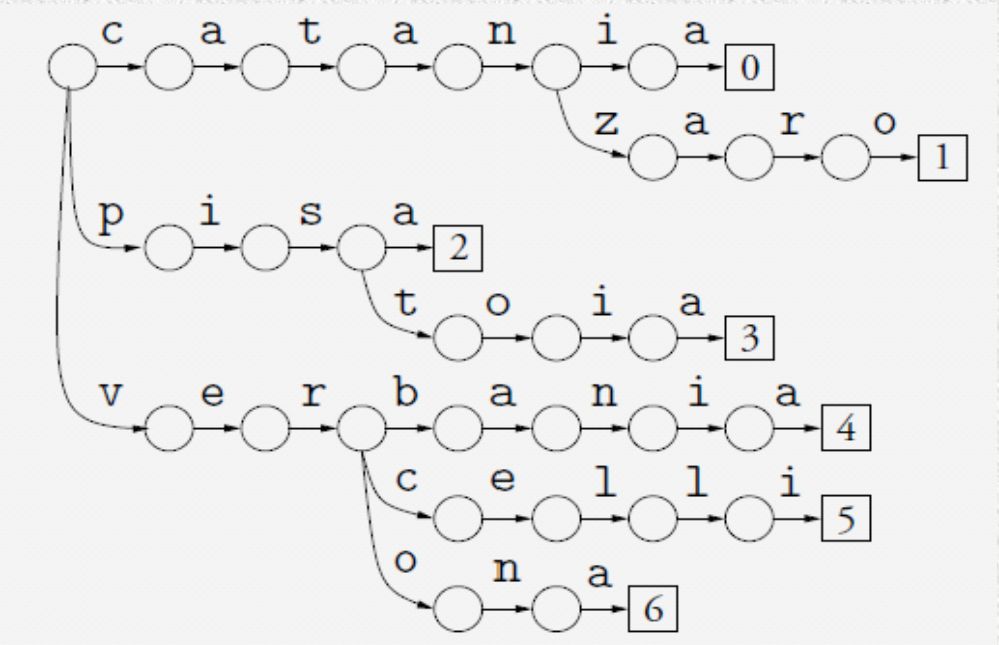
costo O(m)

TRIE: dizionario di stringhe

- I TRIE permettono di:
  - verificare se una stringa P appare in S
  - trovare il più lungo prefisso di P che appare come prefisso di una delle stringhe in S
- Prefisso i di una stringa P, $|P| = m$:

  $P[0, i]$ $\quad 0 \leq i \leq m-1$

TRIE: dizionario di stringhe



Definizione ricorsiva di TRIE per un insieme S di stringhe

TRIE(S) è un *albero cardinale σ-ario*, dove σ = numero di simboli dell'alfabeto $\Sigma = \{0, 1, ..., \sigma\}$ delle stringhe in S:

- S è vuoto ➔ TRIE(S) è NULL
- S non è vuoto ➔ crea radice r e per ogni $c \in \Sigma$
  - S_c = {stringhe in S che iniziano con il simbolo c}
  - Rimuovi il simbolo iniziale c dalle stringhe in S_c (se ≠ vuoto)
  - Poni ricorsivamente r.figlio[c] = TRIE(S_c)

TRIE

- Ciascun nodo u in TRIE(S) contiene un campo u.dato per memorizzare un elemento del dizionario
- Ogni stringa in S corrisponde a un nodo u, ed è memorizzata in u.dato.chiave
- Usando un terminatore di fine stringa per ciascuna stringa è possibile associare in modo univoco le stringhe alle foglie del trie (non è necessario se nessuna stringa è prefisso di un'altra).
- I nodi interni rappresentano prefissi.
- Altezza: lunghezza massima delle stringhe.
- Dimensione: al più N+1

  N= lunghezza totale delle n stringhe

  la dimensione massima si raggiunge se tutte le stringhe hanno carattere iniziale diverso

Ricerca in un trie

- Stringa *pattern* P di ricerca con m simboli, in O(m) tempo indipendentemente da |S|

```
Ricerca( radiceTrie, P ):
  u = radiceTrie;
  FOR (i = 0; i < m; i = i+1) {
    IF (u.figlio[ P[i] ] != null) {
      u = u.figlio[ P[i] ];
    } ELSE {
      RETURN null;
    }
  }
  RETURN u.dato;
```

- Il tempo di ricerca è O(p+1), dove p ≤ m è la lunghezza del prefisso trovato

Ricerca per prefissi

Identifica il nodo u che corrisponde al più lungo prefisso di P che appare nel trie.

```
RicercaPrefissi( radiceTrie, P ):
  u = radiceTrie;
  fine = false;
  FOR (i = 0; !fine && i < m; i = i+1) {
    IF (u.figlio[ P[i] ] != null) {
      u = u.figlio[ P[i] ];
    } ELSE {
      fine = true;
    }
  }
  numStringhe = 0;
  Recupera( u, elenco );
  RETURN elenco;
```

Tutte le stringhe in S che hanno il prefisso corrispondente a u possono essere recuperate con una semplice visita (numStringhe è una variabile globale).

```
Recupera( u, elenco ):
  IF (u != null) {
    IF (u.dato != null) {
      elenco[numStringhe]= u.dato;
      numStringhe = numStringhe + 1;
    }
    FOR (c = 0; c < sigma; c = c + 1)
      Recupera( u.figlio[c], elenco );
  }
```

Inserimento

- Si cerca nel trie il prefisso x più lungo della stringa P da inserire, che occorre in un nodo u del trie;
- Si scompone P come xy:
  - Se y non è vuoto, si sostituisce il sottoalbero vuoto raggiunto con la ricerca di x, mettendo al suo posto il trie per y;
  - Altrimenti si associa P al nodo u identificato.
- Il costo è O(m).
- Non occorrono operazioni di ribilanciamento, infatti la forma del trie è determinata univocamente dalle stringhe in esso contenute.

Applicazione: ordinamento di stringhe in tempo lineare

Il trie mantine l'ordinamento lessicografico….

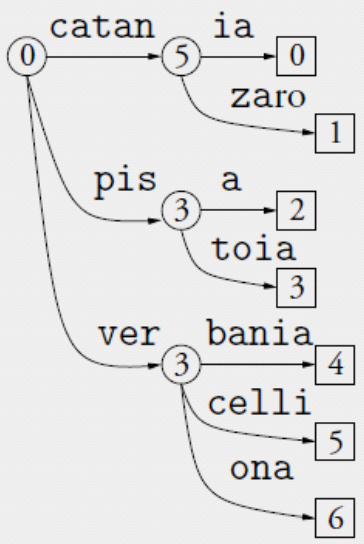
```
OrdinaLessicograficamente( S ):
  radiceTrie = null;
  elemento.sat = null;
  FOR (i = 0; i < n; i = i+1) {
    elemento.chiave = S[i];
    radiceTrie = Inserisci( radiceTrie, elemento );
  }
  numStringhe = 0;
  Recupera( radiceTrie, S );
  RETURN S;
```

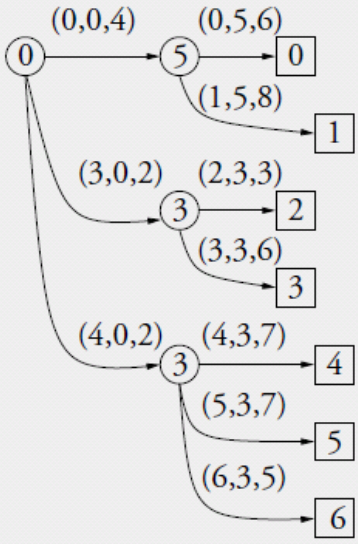
Si effettua, con la funzione Recupera, una visita simmetrica del trie costruito con l'inserimento iterativo delle stringhe contenute nell'array S.

Costo: O(N), se N è la somma delle n stringhe in S

Trie compatto

- Ha solo O(|S|) nodi (al più n nodi interni e n foglie).
- Non esistono nodi (eccetto la radice) con un solo figlio
- Archi etichettati con descrittori di sottostringhe:
  (stringa, inizio, fine)





Applicazioni: albero dei suffissi (*suffix tree*)

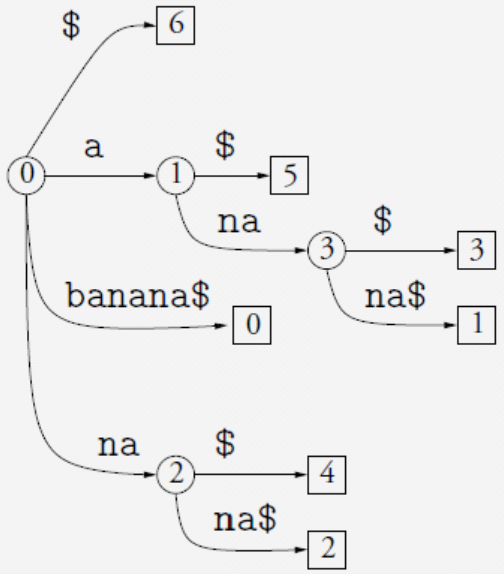
- Più potente delle liste invertite: consente la ricerca anche in testi che non possono essere partizionati in termini (es. sequenze biologiche, testi cinesi, ecc.)

- Trie compatto dove

  S = { suffissi T[i,n] di T}

- Esempio: T = **banana$**

```
banana$ [0]
anana$ [1]
nana$ [2]
ana$ [3]
na$ [4]
a$ [5]
$ [6]
```



Applicazioni: albero dei suffissi (*suffix tree*)

- Non è necessario dividere un testo in termini: ogni segmento è un potenziale termine.
- $O(n^2)$ termini, ma spazio O(n).
- Usando il vocabolario V delle liste invertite per memorizzare questi termini si avrebbe spazio quadratico o superiore.
- E' possibile costruire l'albero dei suffissi in O(n) tempo.

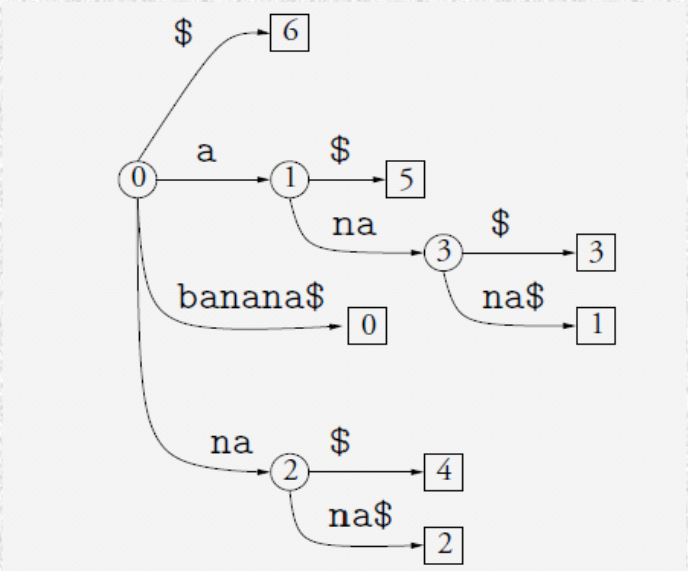
Albero dei suffissi

- Proprietà utilizzata dalla ricerca:
  P = T[i, i+|P|-1] sse P è prefisso del suffisso T[i,n]
- Una volta individuato il nodo u che corrisponde a P, le foglie i discendenti da u corrispondono esattamente alle posizioni i del testo in cui P occorre.
  1. Effettua una ricerca per prefissi di P
  2. Se P appare interamente in un cammino che dalla radice termina in u, recupera i suffissi T[i,n] nel sottoalbero di u.
  3. Per ogni suffisso T[i,n] così recuperato, restituisci la posizione i come occorrenza

Albero dei suffissi

Esempio: ricerca di P = ana

conduce al nodo le cui foglie discendenti sono i = 1 e i = 3, che sono anche le occorrenze di P



T = **banana$**

0 1 2 3 4 5 6

Albero dei suffissi

- Il problema di cercare P in un testo T si riduce al problema di cercare P nel trie compatto.
- Il costo della ricerca è indipendente dalla lunghezza del testo
- L'algoritmo è output-sensitive: tempo O(|P| + occ)
- Volendo usare l'albero per una collezione di documenti $D = \{T_0, T_1, \ldots, T_{s-1}\}$, basta costruirlo sul testo $T = T_0\$T_1\$\ldots\$T_{s-1}\$$ ottenuto concatenando i testi in D separati dal simbolo speciale $.

FINE

Lucidi tratti da
Crescenzi • Gambosi • Grossi,
Strutture di dati e algoritmi
Progettazione, analisi e visualizzazione
Addison-Wesley, 2006
http://algoritmica.org