Dizionari

- Un dizionario memorizza una collezione di elementi e ne fornisce le operazioni di
  - Ricerca
  - Inserimento
  - Cancellazione
- Ciascun elemento contiene una chiave di ricerca
- Oltre alla chiave, contiene dei dati satellite

Dizionari

- **Universo U** delle chiavi di ricerca
- Un dizionario memorizza un insieme S di n elementi (n è la **dimensione** di S)
- Hp.: e ∈ S ha il campo **e.chiave** ∈ U e quello dei dati satellite **e.sat** (che dipende dall'applicazione)
- Operazioni di base su S per una chiave k ∈ U:

Dizionario statico

  - **Ricerca**(k) trova e t.c. **e.chiave** = k (**null** se non esiste)

dinamico

  - **Inserisci**(e) esegue S = S ∪ {e} (hp. chiavi distinte in S, cioè **e.chiave** non è già in S)
  - **Cancella**(k) esegue S = S - {e}, tale che k=**e.chiave**

Liste invertite e trie

- **Information retrieval**: recupero di documenti (testi)
- Molte query sugli stessi documenti: costruzione di un indice (es. motori di ricerca)
- **Liste invertite** (file invertite, file dei *posting*, concordanze): organizzazione logica dei dati
  - **termine** = parola o sequenza alfanumerica massimale
  - **documento** = testo partizionato in una sequenza di termini
  - **vocabolario V** = termini distinti che appaiono nei documenti
  - **lista invertita** per ogni termine P ∈ V (*posting*)
    - Riferimento alla posizione iniziale di ciascuna occorrenza di P nel testo T (a cui è associato un identificatore)
    - L_P = { < T, i > t.c. T=ID(documento) e T[i,i+|P|-1] = P }

Un'implementazione delle liste invertite

- Dizionario per memorizzare il vocabolario V
  - Per esempio, tabella hash o albero binario di ricerca
- Ciascun P ∈ V è memorizzato come un elemento *e* nel dizionario t.c.
  - e.chiave = P
  - e.sat = L_P
- Si parte con il dizionario vuoto

```
CostruzioneListeInvertite( T ):
  i = 0;
  WHILE (i < n ) {
    WHILE (i < n && !AlfaNumerico( T[i] ))
      i = i+1;
    j = i;
    WHILE (j < n && AlfaNumerico( T[j] ))
      j = j+1;
    e = dizionarioListeInvertite.Ricerca( T[i,j-1] );
    IF (e != null) {
      e.sat.InserisciFondo( <T,i> );
    } ELSE {
      elemento.chiave = T[i,j-1];
      elemento.sat = NuovaLista( );
      elemento.sat.InserisciFondo( <T,i> );
      dizionarioListeInvertite.Inserisci( elemento )
    }
    i = j;
  }
```

n è la lunghezza del testo T

identificazione di un termine

il termine è già nel dizionario: si aggiunge la nuova occorrenza

il termine non è nel dizionario: si crea una nuova lista, si aggiunge l'occorrenza e si memorizza il termine nel dizionario

Alfanumerico (c)

return ('a' ≤ c ≤ 'z' || 'A' ≤ c ≤ 'Z' || '0' ≤ c ≤ '9');

Costo

n: numero dei caratteri esaminati

- O(n) al caso medio con tabelle hash
- O(n log n) con AVL

Interrogazioni

- In genere i motori di ricerca permettono di effettuare ricerche su termini legati da operatori booleani
  - P AND Q: entrambi i termini devo apparire in T
  - P OR Q: uno dei due termini deve apparire in T
- Come possiamo fare a rispondere a queste query su un certo testo T?
  - P AND Q $\Leftrightarrow$??
  - P OR Q $\Leftrightarrow$??
  - NOT P $\Leftrightarrow$??

Operazioni booleane

- P AND Q $\Leftrightarrow L_P \cap L_Q$
- P OR Q $\Leftrightarrow L_P \cup L_Q$
- NOT P $\Leftrightarrow$ complemento$\{L_P\}$

- Conviene mantenere le liste **ordinate,** analogamente alla fusione nel mergesort

Operatore NEAR

- P NEAR Q

  i termini P e Q devono occorrere vicini (entro maxPos posizioni)

- È possibile farlo in $O(|L_P|+|L_Q|+occ)$ tempo invece che $O(|L_P| \times |L_Q|)$

  occ = numero di occorrenze.

- Le occorrenze restituite sono triple $< T, i, j >$:

  $0 \le j - i \le maxPos$

  $T[i, i+|P|-1] = P$ e $T[j, j+|Q|-1] = Q$

  oppure $T[i. i+|Q|-1] = Q$ e $T[j, j+|P|-1] = P$

```
InterrogazioneNear( P, Q, maxPos ):
  LP = Ricerca( dizionarioListeInvertite, P );
  LQ = Ricerca( dizionarioListeInvertite, Q );
  IF (LP != null && LQ != null) {
    listaP = LP.sat.inizio;
    listaQ = LQ.sat.inizio;
    WHILE (listaP != null && listaQ != null) {
      <testoP, posP> = listaP.dato;
      <testoQ, posQ> = listaQ.dato;
      IF (testoP < testoQ) {
        listaP = listaP.succ;
      } ELSE IF (testoP > testoQ) {
        listaQ = listaQ.succ;
      } ELSE IF (posP <= posQ) {
        VerificaNear( listaP, listaQ, maxPos );
        listaP = listaP.succ;
      } ELSE {
        VerificaNear( listaQ, listaP, maxPos );
        listaQ = listaQ.succ;
      }
    }
  }
```

stesso testo

Ignorando **`VerificaNear`**, la scansione delle liste è come quella della fusione nel mergesort $O(|L_P|+|L_Q|)$ tempo

VerificaNear

```
VerificaNear( listaX, listaY, M ):                    ⟨pre: posX ⩽ posY⟩
  <testoX, posX> = listaX.dato;
  <testoY, posY> = listaY.dato;
  WHILE (listaY != null && testoX == testoY && posY-posX <= M) {
    PRINT testoX, posX, posY;
    listaY = listaY.succ;
    <testoY, posY> = listaY.dato;
  }
```

- Ogni volta che avanza su una delle due liste, controlla che le posizioni differiscano di al più M
- Fornisce un'occorrenza a ogni iterazione del corpo del **while**
- Viene invocata $O(|L_P|+|L_Q|)$ volte in tutto e il numero totale di iterazioni è occ.
- Il contributo risultante è pari a $O(|L_P|+|L_Q|+occ)$ tempo
- il costo dipende dall'output: il codice è *output-sensitive*

FINE

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