Laboratorio di algoritmi e strutture dati

Docente: Violetta Lonati

mercoledì 10 gennaio 2018

L'obiettivo dell'esercitazione di oggi è scrivere un programma che sappia fare alcune semplici operazioni riguardanti le occorrenze di parole nelle righe di un testo.

A differenza di quanto fatto nell'esercitazione sugli alberi binari di ricerca, useremo una *tabella di hash* per implementare un *dizionario*.

Il programma deve leggere da standard input una sequenza di istruzioni, scritte una per riga, secondo il formato nella seguente tabella. In particolare il comando `?` *word* deve restituire l'elenco di tutti e soli i numeri di riga in cui compare la parola *word*. Nella lettura vanno ignorati i segni di punteggiatura e non si differenziano maiuscole e minuscole; vanno numerate soltanto le righe corrispondenti al comando + *word*.

Istruzione in input	**Operazione**
`+` *line*	Legge la riga *line* e memorizza nel dizionario le parole che la costituiscono
`?` *word*	stampa i numeri di riga in cui è comparsa la parola *word*
`l`	Stampa il numero di righe lette
`n`	Stampa il numero di parole distinte lette
`p`	Stampa le parole presenti nel dizionario, con l'elenco dei numeri di riga di cui compaiono
`f`	Termina l'esecuzione

Esempio di funzionamento

Sul seguente input:

```
+ Try not to become a man of success, but rather try to become a man of value.
+ Look deep into nature, and then you will understand everything better.
l
n
+ The true sign of intelligence is not knowledge but imagination.
+ We cannot solve our problems with the same thinking we used when we created them.
+ Weakness of attitude becomes weakness of character.
+ You can't blame gravity for falling in love.
+ The difference between stupidity and genius is that genius has its limits.
l
n
? become
? the
f
```

il programma produce il seguente output:

```
[l] Lette 2 righe
[n] Lette 22 parole distinte
[l] Lette 7 righe
[n] Lette 61 parole distinte
[?] become: 1
[?] the: 3 4 7
```

1 Strutture dati

1.1 Parole e numeri di riga

Occorre rappresentare un insieme di parole, ciascuna associata all'insieme del numero di righe in cui la parola compare.

Definite oportunamente un tipo strutturato per rappresentare una parola e le righe in cui compare. Poiché non sappiamo a priori quanti sono le righe in cui compare una parola, è necessario usare un array allocato dinamicamente (che chiameremo `ln_arr`), ed è utile prevedere anche dei membri che tengano traccia del relativo spazio allocato e di quello occupato.

```
typedef struct {
        char *word;
        ... ln_arr;
        ...
        ...
} Item;
```

1.2 Dizionario

Per implementare in maniera efficiente il comando di ricerca delle stringhe e considerando il fatto che non ci interessa mantenere le parole in ordine (nessun comando lo richiede), è utile usare la struttura dati *dizionario*, in cui la *chiave* è la parola, e il *valore* è l'elenco dei numeri di riga.

Implementiamo il dizionario con una *tabella di hash*, e gestiamo le collisioni con il meccanismo chiamato di *chaining*: la posizione i-esima della tabella di hash contiene una *catena* con tutti gli elementi che hanno i come valore di hash; le catene sono implementate come semplici liste concatenate. Quando si vuole inserire nel dizionario un nuovo elemento di chiave w, si calcola il valore di hash $h(w)$ della chiave, e si inserisce l'elemento in testa alla catena contenuta nella posizione $h(w)$ della tabella di hash.

Dunque il dizionario è costituito sostanzialmente da un vettore di catene; la dimensione di tale array dipende da una macro `HASHSIZE`. Può essere utile aggiungere anche altre informazioni come ad esempio il numero di parole già inserite:

```
typedef struct ht {
        Chain array[HASHSIZE];
        int count; // parole inserite
} *Dict;
```

1.3 Catene di collisione

Implementate le catene di collisione con liste concatenate: usate un tipo strutturato ricorsivo per gli elementi della catena (con un membro che punta ad un Item e un puntatore al prossimo elemento della catena) e definite un tipo che indica la testa di una catena.

```
struct element {
        Item *item;
```

```
        struct element *next;
};

typedef struct element *Chain;
```

2 Funzioni da implementare

Il programma conterrà:

- funzioni per la gestione degli Item
- funzioni per la gestione delle catene di collisione
- funzioni per la gestione della tabella di hash
- funzioni per la lettura delle parole e delle righe
- la funzione `main`

2.1 Funzioni per la gestione degli Item

Scrivete una funzione per stampare gli Item, una funzione per crearli e una funzione per modificarli.

La funzione di creazione, con prototipo `Item *item_new( char *w, int ln )`, deve allocare lo spazio per un nuovo Item e restituirne l'indirizzo dopo averlo inizializzato con la chiave `w` e il numero di riga `ln`.

Tra i comandi nella tabella sopra non si prevedono cancellazioni, quindi basta che la funzione di modifica sia in grado di aggiungere un nuovo numero di riga `ln` al vettore `ln_arr`, riallocandolo opportunamente se necessario. Il prototipo della funzione di modifica potrebbe essere `void item_modify( Item *p, int ln )`.

2.2 Funzioni per la gestione delle catene di collisione

Riadattate il codice già scritto per le liste di interi, sostituendo gli interi con il tipo `Item` sopra definito. In particolare potranno servire funzioni di questo tipo:

`Item *chain_find( Chain c, char *w )`
cerca nella catena un Item con chiave `w` e ne restituisce il puntatore, oppure `NULL` se `w` non è nella catena.

`Chain chain_insert( Chain c, Item *p )`
crea un nuovo elemento contenente l'Item puntato da p, lo inserisce in testa alla catena e restituisce l'indirizzo della nuova testa.

`void chain_print( Chain h )`
stampa gli elementi della catena.

2.3 Funzioni per la gestione della tabella di hash

Potranno servire funzioni di questo tipo:

`Dict dict_init()`
alloca lo spazio per il dizionario e inizializza i suoi membri; in particolare ricordate di assegnare a `NULL` tutti gli elementi del vettore.

`Item *dict_lookup( Dict t, char *w )`
La funzione `dict_lookup` restituisce l'Item contenente la chiave `w`, oppure `NULL` se `w` non è nel dizionario.

void `dict_add( Dict t, Item *p )`
La funzione `dict_add` aggiunge l'Item nel dizionario assumendo che la chiave `p -> w` non ci sia già.

void `dict_print( Dict h )`
stampa il contenuto del dizionario.

Servirà inoltre definire una *funzione di hash*. Ad esempio potete provare questa funzione (molto semplice e popolare anche se non particolarmente efficace nel limitare le collisioni):

```
unsigned long hash( char *str )
{
        unsigned long hash = 5381;
        int c;

        while ( (c = *str++) )
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

        return hash % HASHSIZE;
}
```

2.4 Funzioni per la lettura delle parole e delle righe

Per implementare il comando di lettura delle righe è necessario innanzitutto dividuare le singole parole che compaiono nella riga e cercarle nel dizionario. Inoltre, per ogni parola *w*:

- se *w* è già presente nel dizionario, ma non è ancora comparsa nella riga corrente, bisogna modificare il relativo Item aggiungendo il numero di riga corrente a quelli già presenti;
- se *w* non è già presente nel dizionario, bisogna aggiungerla, con la riga corrente.

Questo compito può essere svolto da una funzione con prototipo `void line( Dict t, int ln )`.

Tuttavia, memorizzare un'intera riga in una stringa e poi esaminarla per individuare le singole parole potrebbe essere scomodo. Si consiglia invece di scrivere una funzione `read_line` che legga una parola, ovvero una sequenza di caratteri alfabetici (usate il comando `isalpha` contenuto in `ctype.h`). Potete adattare la funzione già scritta come esercizio sull'allocazione dinamica della memoria; per incrementare correttamente il numero di riga è però importante individuare le parole che terminano la riga (ovvero quelle terminate da `'\n'`), sarà dunque necessario fare delle modifiche alla funzione: ad esempio può essere utile passare per riferimento un argomento di tipo carattere per ricordare il carattere terminatore.

La funzione `read_word` può essere inoltre usata per leggere da standard input l'argomento del comando `?`.

2.5 La funzione `main`

Si tratterà sostanzialmente di uno switch per la lettura e l'esecuzione delle istruzioni riportate nella tabella sopra, preceduto dall'opportuna dichiarazione di una variabile di tipo `Dict` e di altre variabili ausiliare. In particolare sarà comodo usare una variabile per ricordare il numero di riga corrente.

NB: attenzione alla gestione del carattere a-capo durante la lettura dei comandi; notate che la funzione `read_line` consuma anche il carattere che termina la stringa (quindi eventualmente l'a-capo).

```
int main() {

        Dict dict = dict_init();
        int ln;
        ...
```

```
        while(( com = getchar()) != 'f'){
                switch(com){
                        case '+': // legge una riga e inserisce parole in dict
                        ln++;
                        line( dict, ln );
                        break;

                        case '?':  // stampa num di riga in cui compare la parola
                        ...
                        break;

                        case 'n':  //  stampa il numero di parole lette
                        ....
                        break;

                        case 'l':  //  stampa il numero di righe lette
                        ...
                        break;

                        case 'p':  //  stampa il dizionario
                        dict_print( dict );
                        break;
                }

        }
        return 0;

}
```

3 Test del programma

Oltre all'esempio di esecuzione presentato all'inizio, potete usare la coppia di file `input_promessiSposi` e `output_promessiSposi` o trovare altri testi liberamente usabili sul sito del progetto Gutemberg `https://www.gutenberg.org`.