



UNIVERSITÀ DEGLI STUDI  
DI GENOVA



Dibris ► I miei corsi ► Informatica (Scuola di Scienze MFN) ► Laurea in Informatica ► Anno Accademico 2013/14 ► L31 - a.a. precedenti ► IP-1213 ► 17 dicembre - 23 dicembre ► Martedì 18 e giovedì 20 dicembre - Esercitazione c...

Sei collegato come **Leonardo Siri.** (Esci)

## Navigazione



### Dibris

#### My home

Dibris

Il mio profilo

Corso in uso

IP-1213

Partecipanti

Badge

Introduzione

ESAMI

Prove d'esame  
degli anni  
precedenti

24 settembre - 30  
settembre

1 ottobre - 7  
ottobre

8 ottobre - 14  
ottobre

15 ottobre - 21  
ottobre

22 ottobre - 28  
ottobre

29 ottobre - 4  
novembre

5 novembre - 11  
novembre

12 novembre - 18  
novembre

19 novembre - 25  
novembre

26 novembre - 2  
dicembre

3 dicembre - 9  
dicembre

10 dicembre - 16  
dicembre

17 dicembre - 23  
dicembre

**Martedì 18 e  
giovedì 20  
dicembre -  
Esercitazione  
C...**

ESAMI

I miei corsi

Corsi



## Amministrazione

Amministrazione del  
corso

Impostazioni profilo

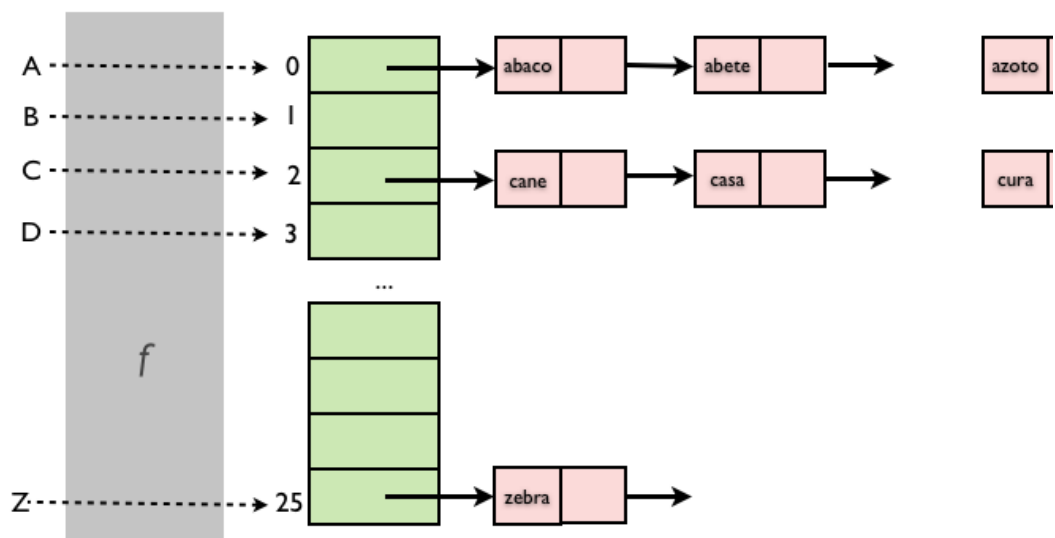
## Martedì 18 e giovedì 20 dicembre - Esercitazione con valutazione: dizionario indicizzato

### Dizionario indicizzato

Scopo dell'esercitazione di questa settimana è l'implementazione di un dizionario che permetta di cercare velocemente gruppi di parole che iniziano per una data lettera (prima parte) o per una coppia di lettere (seconda parte).

### Prima parte

La struttura dati che utilizziamo è descritta nella figura



Un array ci permette di tenere in memoria l'indirizzo di partenza di una linked list che memorizza in modo ordinato tutte le parole inserite che iniziano per una certa lettera.

La funzione  $f$  associa ad una data lettera, l'indice opportuno dell'array.

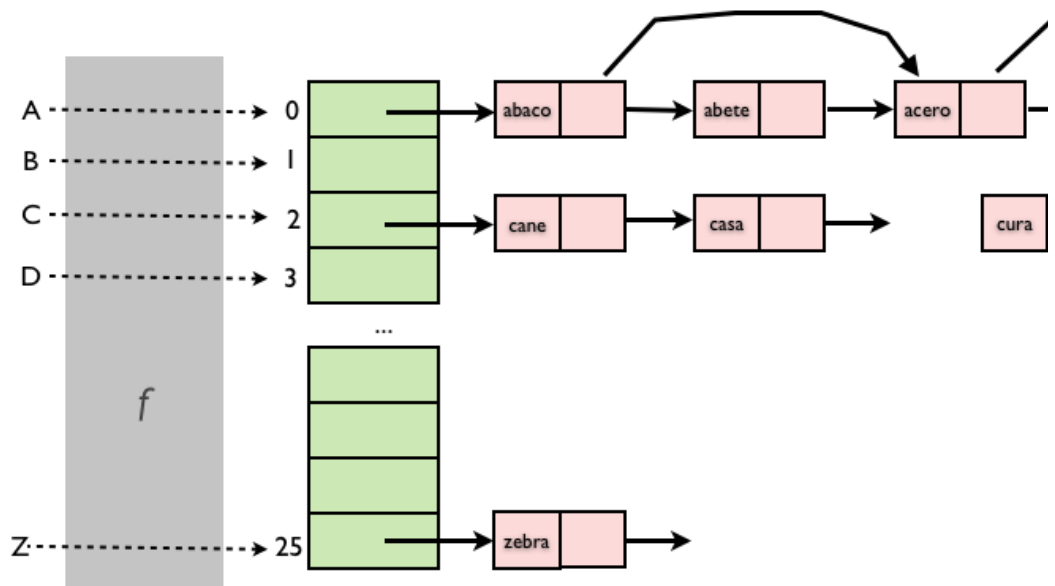
In questa prima parte dovete prevedere almeno due moduli:

- **word\_list**: linked list di parole (stringhe) che ci permette di memorizzare una lista ordinata di parole. Minimalmente questo modulo deve implementare le operazioni:
  - **create\_list**: creazione di una lista vuota,
  - **add\_word**: inserimento ordinato di un elemento nella lista,
  - **contains\_word**: verifica della presenza di una parola nella lista.
- **dictionary**: array di puntatori. Prevedere le funzioni:
  - **create\_dictionary**: inizializza un dizionario vuoto (tutti gli elementi = NULL) della lunghezza opportuna (26 elementi)
  - **add\_word**: inserisce una parola
  - **contains\_word**: verifica la presenza di una parola nel dizionario
  - **print\_word\_by\_letter**: stampa tutte le parole presenti nel dizionario che iniziano per una data lettera

Prevedere un opportuno file principale che permetta di verificare tutte le funzioni del dizionario tramite menù

### Seconda parte

In questa seconda parte vogliamo rendere più efficiente la ricerca di parole che cominciano per una data coppia di lettere. Per questo motivo implementiamo la struttura dati descritta in figura:



Osserviamo che la struttura del dizionario non è cambiata, ma lo è quella della lista. Per questo motivo implementiamo un nuovo modulo, `another_list`, dove ogni elemento contiene un doppio puntatore, il primo punta all'elemento successivo nella lista (come sempre), il secondo, nel caso l'elemento in questione sia il primo di un gruppo di parole che iniziano con una data coppia di caratteri, punta al primo elemento con la coppia lessicografica immediatamente successiva. Questo nuovo modulo dovrà implementare le stesse funzioni di `word_list`.

Aggiungiamo al modulo `dictionary` una funzione per la stampa delle parole che iniziano con una data coppia di caratteri.

Prevedere un opportuno file principale che permetta di verificare tutte le funzioni del dizionario esteso tramite menù.

### Dettagli pratici

- le parole con lunghezza < 2 possono essere eliminate
- considerate solo parole scritte in caratteri minuscoli

Il menu del programma principale deve apparire come segue:

```
a)aggiungi una parola
l)leggi tutte le parole di un file
c)erca una parola
s)tampa tutte le parole che iniziano per...
e)sci
```

Per scegliere una funzione si usa la corrispondente lettera iniziale (minuscola).

Il programma dovrà occuparsi di effettuare le operazioni necessarie per preparare i dati da memorizzare, ossia:

- leggere una parola (da terminale o da file - nel caso del file dovrà anche leggere il nome del file e occuparsi di aprirlo come `istream`)
- garantire che le parole siano tutte in minuscolo (potete convertire tutti i caratteri usando la funzione **`tolower`** dichiarata in **`cctype`**)
- eliminare eventuale punteggiatura e tutto quello che non è alfabetico all'inizio e alla fine di ciascuna parola letta (potete creare una funzione **`trim`** che prende la stringa grezza e la restituisce ripulita). Questa funzione rende il programma più tollerante ai miei errori, da terminale, e a contenuti indesiderati, da file. Inoltre evita di memorizzare più versioni diverse della stessa parola solo perché seguita da punteggiature diverse (quindi riduce la quantità di dati da trattare).

Queste operazioni richiedono qualche linea di codice in più, ma risparmiano molto lavoro nella fase di prova del programma.

Un file da utilizzare per i vostri test: [mobydick](#)

### Riflessione

Osserviamo che gli obiettivi della parte 2 avrebbero potuto essere raggiunti in modo molto più agevole allungando l'array del dizionario in modo opportuno, come? In questo caso avremmo dovuto concepire una funzione `f` di accesso all'elemento dell'array più elaborata. Discutere questa possibilità, eventualmente con l'aiuto di parti di codice, all'interno di un file di testo "riflessione.txt" che accluderete al codice consegnato.

**Nota finale.** Non è importante ai fini dell'esercitazione, ma per conoscenza menzioniamo che la struttura dati implementata nella prima parte è una versione semplice di una hash table (vedete la [hash table](#) su wikipedia).

## Valutazione


Il codice che non compila potrebbe non venire corretto.

Il punteggio che otterrete sarà nel range [0,4].

- Per ottenere il punteggio massimo dovete completare tutta l'esercitazione, e produrre codice leggibile ben strutturato, ben commentato, che rispetti tutte le specifiche. Il file "riflessione.txt" dovrà essere chiaro e corretto.
- 
- Per ottenere un punteggio buono (2 o 3) dovete produrre codice leggibile ben strutturato, ben commentato, che rispetti le specifiche e sia in grado di gestire in modo opportuno i casi indicati nel testo. La sola prima parte, anche se corretta, non permette di ottenere più di 1.5 punti. Con la prima parte e la riflessione, anche se corrette, potete ottenere al più 2.5 punti.
- Comunque se non riuscite a completare tutte le richieste, confinate la parte mancante in modo che il vostro lavoro sia completo nella parte che consegnate. In questo caso vi consigliamo vivamente di includere un file "readme.txt" che descriva quali sono le parti mancanti.



## Stato consegna

|                        |  |
|------------------------|--|
| Stato consegna         | Consegnato per la valutazione  |
| Stato valutazione      | Non valutata   |
| Termine consegne       | venerdì, 4 gennaio 2013, 23:55   |
| Tempo rimasto          | Il compito è stato consegnato 2 giorni 23 ore in anticipo  |
| Ultima modifica        | mercoledì, 2 gennaio 2013, 00:03   |
| Consegna file          |  <a href="#">Siri_Lab12.tgz</a> |
| Commenti alle consegne | <a href="#">► Commenti (0)</a>   |