IP a.a. 2021/22 - Esame del 18 gennaio 2022

Prima di cominciare lo svolgimento leggete attentamente tutto il testo.

Questa prova è organizzata in due sezioni, in cui sono dati alcuni elementi e voi dovete progettare ex novo tutto quello che manca per arrivare a soddisfare le richieste del testo. Per ciascuna sezione nel file zip del testo trovate una cartella contenente i file da cui dovete partire. Dovete lavorare solo sui file Cognome1.cpp e Cognome2.cpp, rinominandoli rispettivamente con il vostro cognome seguito dal numero 1 o 2. Modificare gli altri file è sbagliato (ovviamente a meno di errata corrige indicata dai docenti).

In questi file dovete realizzare le funzioni richieste, esattamente con la *segnatura* con cui sono indicate: nome, tipo restituito, tipo degli argomenti nell'ordine in cui sono dati. Potete inoltre realizzare altre funzioni in tutti i casi in cui lo ritenete appropriato. Potete inserirvi tutti gli `#include` che vi servono oltre a quello relativo allo header con le funzioni da implementare.

1 Bag of words

Rappresentazione sintetica del contenuto di testi.

Per analizzare in modo automatico il contenuto di documenti testuali, una struttura dati semplice ma sorprendentemente efficace è "bag of words", "sacco di parole".

Una struttura dati "bag", anche detta "multiset", somiglia a un insieme ("set"), nel senso che gli elementi che contiene non hanno un ordine significativo; si può aggiungere un elemento, chiedere se un elemento è presente, eliminare un elemento, ma non si può chiedere "in che posizione" si trova un elemento.

A differenza di un insieme, gli elementi in un "bag" possono comparire più volte. Quindi, nella struttura dati, a ogni elemento è associato il numero di volte che questo è presente, e si può chiedere "quante volte" l'elemento è stato inserito.

Realizzazione del tipo bag of words

Vedere il file `bow.h` per le definizioni effettive.

Useremo una `struct` di tipo `Entry` per ogni elemento, contenente un membro di tipo `std::string` per memorizzare una parola e uno di tipo `int` per memorizzare il relativo numero di occorrenze.

Una bag-of-words sarà realizzata come un `std::vector` di elementi di tipo `Entry`, ridefinito con il nome `BagOfWords`.

Materiale dato

Nel file zip trovate

- un file `bow.h` contenente le definizioni di tipo date e le intestazioni delle funzioni ⟵ NON MODIFICARE
- un file `bowtest.cpp` contenente un main da usare per fare testing delle vostre implementazioni e la realizzazione (corpo) delle funzioni fornite da noi. ⟵ NON MODIFICARE
- un file `Cognome1.cpp` contenente lo scheletro delle funzioni che dovete realizzare voi ⟵ DOVETE MODIFICARE SOLO QUESTO, INCLUSO IL SUO NOME
- un file `testo.txt` contenente un testo di prova

Funzione `add` (DA FARE)

```
void add(BagOfWords & b, std::string w);
```

Aggiunge la parola `w` al BagOfWords `b`:

- se `w` non è presente, crea una nuova Entry con conteggio (`num`) pari a 1 e la aggiunge a b;
- se `w` è già presente, incrementa il suo conteggio.

Funzione `del` (DA FARE)

```
void del(BagOfWords & b, std::string w);
```

Toglie la parola `w` dal BagOfWords `b`:

- se `w` è presente con conteggio > 1, decrementa il conteggio;
- se `w` è presente con conteggio $== 1$, elimina l'elemento e scala tutti i successivi indietro di una posizione, poi fa il resize di b;
- se `w` non è presente, non fa niente.

Funzione `count` (DA FARE)

```
int count(const BagOfWords & b, std::string w);
```

Restituisce il conteggio della parola `w` nel BagOfWords `b`, oppure restitruisce 0 se `w` non è presente.

Funzione `print` (FORNITA)

```
void print(const BagOfWords & b);
```

Stampa b su `std::cout`.

NOTA: Questo è solo un esercizio, un vector è tutt'altro che la scelta migliore per questo scopo. Come minimo, per rendere piu efficiente l'accesso, il vector andrebbe ordinato per poter fare una ricerca binaria, cosa che qui non è richiesta; comunque la cancellazione resta poco efficiente.

2 Undo stack

Memorizzare un elenco di movimenti in modo da essere in grado di tornare indietro.

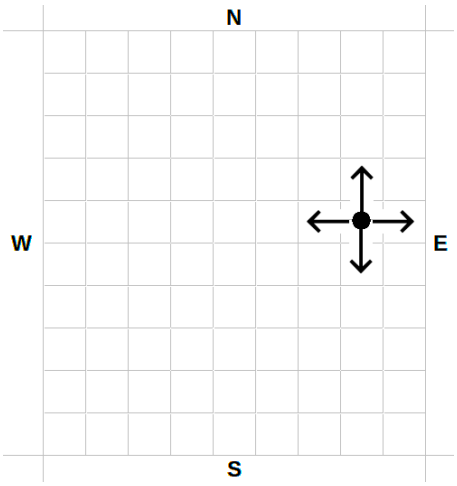
Un veicolo programmabile svolge un compito prestabilito, muovendosi in un ambiente descritto da una "griglia" di posizioni discrete, così che in ogni posizione sono possibili solo singoli passi verso nord, sud, est, ovest.

Vogliamo che il veicolo sia in grado di effettuare i movimenti all'indietro in modo da rpercorrere la traiettoria in senso inverso.

Per fare questo, la struttura dati ideale è uno **stack**. In uno stack sono possibili due operazioni:

1. Aggiunta di un elemento in cima (`push`);
2. Rimozione dell'elemento in cima (`pop`).

Ogni passo viene aggiunto in cima allo stack con l'operazione di push; quando è il momento di rifare la stessa traiettoria all'indietro, si fa una serie di pop e si ottengono tutti i passi fatti, ma in ordine invertito (l'ultimo aggiunto viene preso per primo, last in first out).



Realizzazione del tipo stack di movimenti

Vedere il file `movestack.h` per le definizioni effettive.

I movimenti sono codificati con quattro comandi di un singolo carattere, più un quinto che indica il tornare indietro:

`N`	Un passo verso l'alto
`S`	Un passo verso il basso
`E`	Un passo a destra
`W`	Un passo a sinistra
`B`	Annulla l'ultimo passo e torna alla posizione precedente

Realizziamo il tipo `MoveStack` come una lista collegata semplice i cui elementi (di tipo struct `Move`) contengono un campo carattere che memorizza la mossa fatta. Vanno memorizzate solo `NSEW`; quando viene incontrato `B` si annulla invece l'ultimo passo fatto, quindi non si memorizza niente.

Teniamo anche conto della posizione ad ogni passo, in modo da poter verificare che, facendo l'undo (anche detto "backtracking") di un intero percorso, si ritorna al punto di partenza.

Per memorizzare la posizione corrente si usano due variabili intere `x` e `y`. Per semplicità consideriamo una griglia illimitata, quindi qualunque valore negativo o positivo va bene, senza dover fare alcun controllo.

La posizione orizzontale `x` si incrementa a ogni `E` e si decrementa a ogni `W`; la posizione verticale `y` si incrementa a ogni `S` e si decrementa a ogni `N`.

Materiale dato

Nel file zip trovate

- un file `movestack.h` contenente le definizioni di tipo date e le intestazioni delle funzioni ⟵ NON MODIFICARE
- un file `movestacktest.cpp` contenente un main da usare per fare testing delle vostre implementazioni e la realizzazione (corpo) delle funzioni fornite da noi. ⟵ NON MODIFICARE
- un file `Cognome2.cpp` contenente lo scheletro delle funzioni che dovete realizzare voi ⟵ DOVETE MODIFICARE SOLO QUESTO, INCLUSO IL SUO NOME
- tre file `path1`, `path1` e `path1`, contenenti sequenze di comandi di prova.

Funzione `push`

```
void push(MoveStack & ms, char cmd);
```

Aggiunge un elemento in testa, contenente il comando (singolo carattere) `cmd`.

Funzione `pop`

```
char pop(MoveStack & ms);
```

Legge il comando contenuto nell'elemento in testa, rimuove l'elemento e restituisce il comando.
Attenzione: la funzione non deve tenere conto del caso in cui lo stack delle mosse risulta vuoto. Se volete potete sollevare una eccezione in questo caso, ma è responsabilità del programma chiamante assicurarsi che lo stack non sia vuoto.

Funzione `doMove`

```
void doMove(int &x, int &y, char cmd);
```

Riceve le coordinate della posizione corrente, e le aggiorna in base al comando di movimento `cmd`.

Note

- Come al solito siete autorizzati a creare funzioni ausiliarie da definire nel file `Cognome⟨N⟩.cpp`.
- Quando usate la memoria dinamica, ricordatevi di allocare e soprattutto deallocare correttamente.