IP a.a. 2022/23 - Esame del 16 giugno 2023

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 indicati in ciascuna parte. 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. Non è consentito modificare queste informazioni. Potete invece fare quello che volete all'interno del corpo delle funzioni: in particolare, se contengono già una istruzione `return`, questa è stata inserita provvisoriamente per rendere compilabili i file ancora vuoti, e **dovrete modificarla in modo appropriato**.

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. Attenzione però che **usare una funzione di libreria per evitare di scrivere del codice richiesto viene contato come errore** (esempio: se è richiesto di scrivere una funzione di ordinamento, usare la funzione `std::sort()` dal modulo di libreria standard `algorithm` è un errore).

Il programma principale, che esegue il test, è dato in ognuna delle sezioni.

1 Sezione 1 - Array - (max 7 punti)

Per questa parte lavorate nella cartella `Sezione1`. Per ogni esercizio dovete scrivere una funzione nel file specificato, fornito nel file zip, completandolo secondo le indicazioni.

Materiale dato

Nel file zip trovate

- Un file `array.h` contenente le definizioni di tipo dato e le intestazioni delle funzioni ⟵ NON MODIFICARE
- un file `mainTestArray.cpp` contenente un main da usare per fare testing ⟵ NON MODIFICARE
- un file `Cognome1.cpp` ⟵ MODIFICARE IL SUO CONTENUTO

1.1 Funzione stampa - (2 punti) - OBBLIGATORIA

```
std::string stampa(int arrayInteri[], int dimensione)
```

- INPUT:
  - `arrayInteri[]`: l'array di interi da stampare,
  - `dimensione`: la dimensione dell'array (si assume >= 0)
- OUTPUT: stringa che rappresenta contenuto dell'array
- Comportamento:

  la funzione deve ritornare una stringa con il contenuto dell'array di interi. Ogni intero e' separato dal successivo da una virgola; in caso di array vuoto, la funzione deve ritornare la stringa vuota.

  NB: per trasformare un intero in string e concaternarlo ad una stringa out esistente scrivere:

  `out = out + std::to_string(intero);`
- Esempi:

  INPUT => OUTPUT

  `arrayInteri`=[] , `dimensione`=0 => `""`

  `arrayInteri`=[77] , `dimensione`=1 => `"77"`

  `arrayInteri`=[77,56] , `dimensione`=2 => `"77,56"`

  `arrayInteri`=[77,56,104] , `dimensione`=3 => `"77,56,104"`

1.2 Funzione stampaKelementi - (2 punti)

```
std::string stampaKelementi(int arrayInteri[], int dimensione, int k)
```

- INPUT:
  - `arrayInteri[]`: l'array di interi da stampare,
  - `dimensione`: la dimensione dell'array (si assume >= 0);
  - `K`: il numero di elementi da stampare partendo dall'inizio (si assume >= 0)
- OUTPUT: stringa che rappresenta i primi K elementi dell'array
- Comportamento:

  la funzione deve ritornare una stringa contenente i primi K elementi dell'array di interi. Ogni intero e' separato dal successivo da una virgola; in caso di array vuoto ritorna la stringa vuota. Se K e' maggiore o uguale alla dimensione dell'array viene stampato l'intero array.
- Esempi:

  INPUT => OUTPUT

  `arrayInteri`=[77,56,104], `dimensione`=3, `k`=0 => `""`

  `arrayInteri`=[77,56,104], `dimensione`=3, `k`=1 => `"77"`

  `arrayInteri`=[77,56,104], `dimensione`=3, `k`=2 => `"77,56"`

  `arrayInteri`=[77,56,104], `dimensione`=3, `k`=3 => `"77,56,104"`

  `arrayInteri`=[77,56,104], `dimensione`=3, `k`=4 => `"77,56,104"`

1.3 Funzione sort - (3 punti)

```
void sort(int arrayInteri[], int dimensione)
```

- INPUT:
  - `arrayInteri[]`: l'array di interi da ordinare,
  - `dimensione`: la dimensione dell'array (si assume >= 0)
- OUTPUT: nessuno
- Comportamento:

  la funzione deve ordinare l'array di interi preso in input usando un algoritmo di sort tra quelli visti a lezione.

2 Sezione 1 - Liste - (max 9 punti)

Per questa parte lavorate nella cartella `Sezione2`, nel file `Cognome2.cpp`.

Una lista doppiamente linkata è una lista in cui ogni elemento, in aggiunta al puntatore al prossimo elemento (`next`), possiede anche un puntatore all'elemento precedente (`prev`). Una lista doppiamente linkata si può quindi scorrere sia in avanti che indietro.

Vogliamo modellare tramite una lista doppiamente linkata una lista di citta' da visitare in un percorso. Ogni citta' (escluse la prima e l'ultima) e' collegata ad una citta' precedente (`prev`) e una successiva (`next`).

Di ogni citta' memorizziamo il nome come stringa. Esempio di lista: (Genova, Milano, Venezia)

NB: Come in una lista semplice, il membro `next` vale `NULL` nell'ultimo elemento (coda); a differenza di una lista semplice, il membro `prev` vale `NULL` nel primo elemento (testa). Una lista con un solo elemento ha `next == prev == NULL`.

Siano date le seguenti definizioni:

```
// tipo contenuto della Cella
typedef std::string City;

// tipo Cell di una lista doppiamente linkata
struct Cell {
   City city;
   struct Cell* next;
```

```
    struct Cell* prev;
};

// tipo lista
typedef Cell * List;
```

Si richiede di implementare le funzioni descritte nel seguito.

Materiale dato

Nel file zip trovate

- un file `list.h` contenente le definizioni di tipo dato e le intestazioni delle funzioni ⟵ NON MODIFICARE
- un file `mainTestList.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 ⟵ MODIFICARE IL SUO CONTENUTO

2.1 Funzione inserimentoInTesta - (2 punti) - OBBLIGATORIA

```
void inserimentoInTesta(List &list, City newCity)
```

- INPUT:
  - `list`: la lista a cui aggiungere il nuovo elemento,
  - `newCity`: l'elemento da inserire
- OUTPUT: nessuno
- Comportamento:

  la funzione deve inserire l'elemento `newCity` in testa alla lista doppiamente linkata `list`. `list` puo' contenere gia' valori oppure essere vuota (cioe' uguale a `NULL`): gestire entrambi i casi; `newCity` invece si suppone sempre valorizzata correttamente.
- Esempi:

  fornendo come parametri `newCity` = Pisa e `list` = (Genova, Milano, Venezia) dopo l'esecuzione della funzione si avra' `list` = (Pisa, Genova, Milano, Venezia)

2.2 Funzione stampaAndata - (2 punti)

```
std::string stampaAndata(List list)
```

- INPUT:
  - `list`: la lista da visitare
- OUTPUT: una stringa che rappresenta l'intera lista (dal primo all'ultimo elemento)
- Comportamento:

  la funzione deve ritornare in una stringa la lista delle Citta' da visitare contenuta nella lista in ordine dalla prima all'ultima seguendo i campi `next` della lista.

  NB: per concatenare una virgola alla stringa out scrivere: `out = out + ",";`
- Esempi:

  - fornendo in input `list` = (Genova, Milano, Venezia) la funzione ritorna la stringa `"Genova,Milano,Venezia"`

  - fornendo in input `list` = () la funzione ritorna la stringa `""`

2.3 Funzione stampaRitorno - (3 punti)

```
std::string stampaRitorno(List list)
```

- INPUT:
  - `list`: la lista da visitare
- OUTPUT: una stringa che rappresenta l'intera lista (dall'ultimo al primo elemento)
- Comportamento:
  la funzione deve ritornare in una stringa la lista delle Citta' da visitare contenuta nella lista in ordine dall'ultima alla prima seguendo i campi `prev` della lista.
- Esempi:
  - fornendo in input `list` = (Genova, Milano, Venezia) la funzione ritorna la stringa `"Venezia,Milano,Genova"`
  - fornendo in input `list` = () la funzione ritorna la stringa `""`

2.4 Funzione stampaAdiacenti - (5 punti)

```
std::string stampaAdiacenti(List list, City newCity)
```

- INPUT:
  - `list`: la lista da visitare
  - `newCity`: l'elemento Y di cui cercare gli adiacenti X e Z
- OUTPUT: una stringa contenente una tripla X,Y,Z dove X è il nome del prev di Y e Z il nome del next di Y.
- Comportamento:
  la funzione trova le citta' adiacenti X,Z nel percorso alla citta Y fornita in input. Usare i campi `next` e `prev` della citta' Y.
  Quindi la funzione ritona una stringa contenente una tripla X,Y,Z dove X è il nome del `prev` di Y e Z il nome del `next` di Y.
  NB: e' necessario creare una stringa appropriata in ciascuno dei casi
  - (1) lista vuota: valore di ritorno stringa vuota `""`
  - (2) citta' Y non trovata: valore di ritorno stringa `"-,-,-"`
  - (3) `prev == NULL` (e next no): valore di ritorno stringa `"-,Y,Z"`
  - (4) `next == NULL` (e prev no): valore di ritorno stringa `"X,Y,-"`
  - (5) `prev == next == NULL`: valore di ritorno stringa `"-,Y,-"`

  Si assume che la citta' NON possa essere presente piu' volte nel percorso