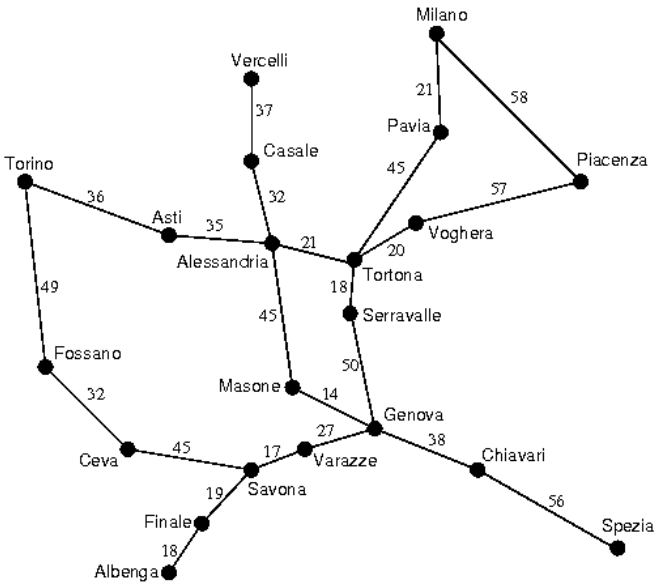
1 Introduzione

Scopo di questa esercitazione è scrivere un ***programma che simuli, in maniera semplificata, il movimento di N automezzi lungo una rete stradale segnalando quando un automezzo transita attraverso una città.***

1.1 Input

Abbiamo in input ***un insieme di tragitti di veicoli relativi ad una mappa stradale*** (come modellare una mappa e come calcolare un tragitto in essa è stato oggetto del precedente laboratorio sui grafi).

Questo laboratorio ne rappresenta un'evoluzione e potrebbe essere collegato al precedente chiedendo anche di calcolare i tragitti a partire dalla mappa, ma per semplicità assumiamo che i tragitti siano già stati calcolati e partiamo da essi.
Ciascun veicolo è identificato da una targa e ha una velocità media espressa in km/h. Il veicolo viaggia da una città di partenza ad una città di destinazione, attraversando una serie di città intermedie.



1.2 Definizione di Tragitto

Il tragitto del veicolo è dato dalla sequenza $C_1, C_2, \ldots, C_k$ di città attraversate, e dalla lunghezza d_i di ogni tratta stradale compresa fra due città consecutive C_i, C_{i+1}.
Quindi, possiamo rappresentare un tragitto con una sequenza alternata di città e di distanze in km:

$$d_1\ C_1\ d_2\ C_2\ \ldots\ d_k\ C_k$$

dove, per $i = 1 \ldots k-1$, d_i è la lunghezza della tratta stradale da C_i a C_{i+1}.

Per esempio, relativamente alla mappa stradale mostrata sopra, sono tragitti:

```
Partenza = Tortona    Tragitto = 18 Serravalle 50 Genova 27 Varazze
Partenza = Pavia      Tragitto = 45 Tortona 20 Voghera 57 Piacenza
```

1.3 Simulazione

Ciascun tragitto, da solo, mostra le tappe di un veicolo ma non dice nulla sullo svolgimento temporale dei suoi movimenti. Tuttavia, conoscendo la velocità media del veicolo è possibile calcolare in quali istanti di tempo il veicolo raggiunge le varie città previste dal suo tragitto.
Per esempio, consideriamo un veicolo che percorre il primo tragitto (da Tortona a Varazze) alla velocità media di 80 km/h (pari a 1.33 km/min). Tale veicolo si troverà a Tortona all'istante t=0, a Serravalle all'istante t=14, a Genova a t=51, e infine a Varazze a t=71 (tempi in minuti, arrotondati). Naturalmente si può estendere a N veicoli.

2 Obiettivo

Si realizzi un programma in C++ che ***simuli il movimento di N veicoli su strada***, seguendo l'approccio indicato (vedi dopo).

2.1 Input del programma

I dati sono acquisiti in input (da file) nel formato:

```
N
targa1 locationIniziale1 tragitto1 velocita1
...
targaN locationIniziale2 tragittoN velocitaN
```

Per esempio (due veicoli che percorrono i due tragitti esemplificati sopra):

```
2
AB123CD Tortona 18 Serravalle 50 Genova 27 Varazze 80
PQ789RS Pavia 45 Tortona 20 Voghera 57 Piacenza 100
```

Dunque, per ogni riga di input, la stringa iniziale rappresenta la targa; poi la località di partenza, il tragitto e l'ultimo numero in fondo rappresenta la velocità espressa in km/h.

2.2 Output del programma

L'output della simulazione deve mostrare le varie tappe *nell'ordine temporale con cui esse vengono raggiunte* dai vari mezzi. Nel nostro esempio, sarà:

```
AB123CD 0 Tortona
PQ789RS 0 Pavia
AB123CD 14 Serravalle
PQ789RS 27 Tortona
PQ789RS 39 Voghera
AB123CD 51 Genova
AB123CD 71 Varazze
PQ789RS 73 Piacenza
```

Le informazioni sul movimento, qui rappresentate in modo testuale, potrebbero essere inviate a una apposita interfaccia grafica per la visualizzazione effettiva della flotta di veicoli sulla mappa stradale. Questa visualizzazione potrebbe essere di grande interesse per corrieri quali DHL, FedEx, etc, ma esula dagli scopi delle nostre esercitazioni.

3 Implementazione: Coda con Priorità realizzata come Heap Binario

Uno dei modi più semplici per realizzare la simulazione consiste nell'utilizzare una **coda con priorità, realizzata come heap binario**, in cui la radice è etichettata con il valore minimo (NB: nelle slide di teoria la maggior parte degli esempi sono invece con radice etichettata da valore massimo), e seguire l'approccio seguente (che trovate già implementato nel main):

- Si acquisisce il numero N di veicoli
- Per ciascun veicolo vengono acquisite targa, la citta di partenza, il tragitto e velocità media in km/h
- Si prende la prima tappa da ciascuno degli N tragitti, che deve coincidere con la città di partenza; la si marca con l'indicazione temporale (timestamp) `t=0`, e si inserisce nello heap; abbiamo così N elementi nello heap, tutti con timestamp `t=0`.
- Finché lo heap non risulta vuoto, si itera nel modo seguente:
  - si estrae dallo heap l'elemento E con timestamp minimo (cioè quello in posizione radice) e lo si elimina dallo heap,
  - si deduce il veicolo V al quale tale elemento fa riferimento,
  - si visualizza in output la targa del veicolo insieme al timestamp e alla tappa raggiunta (vedi esempio sopra),
  - si calcola la successiva tappa di V (se tale tappa esiste) e il tempo T che occorre per raggiungerla,
  - si somma T al timestamp di E creando così un nuovo elemento che va inserito nello heap.
- Quando lo heap è vuoto, tutti i veicoli hanno raggiunto le loro destinazioni e la simulazione è terminata. Si noti che, con questo approccio, la dimensione massima dello heap non è mai superiore a N (perché abbiamo un elemento per ogni veicolo).

Lo heap binario **deve essere realizzato mediante un array**. Nei vari elementi dell'array possono essere inserite tutte le informazioni necessarie, tra cui ovviamente il timestamp che ha il ruolo di chiave.
Si ricorda che rappresentare un albero binario quasi completo con un array è molto semplice: i figli dell'elemento di posto i si trovano infatti nelle posizioni $2i$ e $2i+1$ se trascuriamo la posizione 0, mentre il padre (se esiste) si trova in posizione $\lfloor (i-2)/2 \rfloor + 1$ se trascuriamo la posizione 0. Se consideriamo anche la posizione 0, si ha che i figli dell'elemento di posto i si trovano nelle posizioni $2i+1$ e $2i+2$, mentre il padre (se esiste) si trova in posizione $\lfloor (i-1)/2 \rfloor$.
Non è dunque necessario usare puntatori o indici che riferiscano ai figli o al padre. Oltre a non essere necessario, è vietato, così come l'uso dei vector.

4 Funzioni da implementare

La simulazione è già implementata nel main, voi dovrete quindi progettare e implementare il codice delle funzioni nei file `tragitto.cpp` (4 funzioni) e `priority_queue.cpp` (5 funzioni) che troverete, assieme ad altri file, all'interno del file .zip scaricabile da Aulaweb nella sezione relativa al laboratorio 9. Più in dettaglio le funzioni da implementare sono le seguenti:

```
//********************** tragitto.h **********************
#include <iostream>
#include <string>
#include <sstream>
#include <fstream>

using namespace std;

namespace tragitto {

struct Tappa {
  float distanza;  // distanza di questa tappa dalla precedente in km
  string citta;    // luogo di questa tappa
};

struct TragittoElem {
  Tappa tappa;
  TragittoElem* next;
};

// Il tragitto di un veicolo e' una lista con le varie tappe in ordine,
// cosi' come spiegato nel testo del laboratorio
typedef TragittoElem* Tragitto;

const Tragitto tragittoVuoto = NULL;

// verifica se un Tragitto e vuoto
bool isEmpty(Tragitto);

// inserisce una Tappa in un Tragitto
void insert(Tragitto&, Tappa);

// ritorna FALSE se il Tragitto e vuoto
// altrimenti rimuove la prima Tappa del Tragitto
//    e la assegna al secondo parametro e ritorna TRUE
bool extract(Tragitto&, Tappa&);
}

// stampa un Tragitto (formato: N volte [distanza citta'])
void printTragitto(tragitto::Tragitto);
```

```
//********************** priority_queue.h **********************
#include <iostream>
#include <string>
#include <sstream>
#include <fstream>

using namespace std;

/*************************************************************************/
/*          Coda con priorita' implementata come heap binario            */
/*************************************************************************/

#include "tragitto.h"  // serve per associare un tragitto a ciascun veicolo
using namespace tragitto;

namespace priorityQueue {

// Ciascun veicolo e' descritto con una serie di info
struct InfoVeicolo {
  string plate = "";                        // targa
  tragitto::Tragitto path = tragittoVuoto;  // tragitto da compiere
  float speed = 0;                          // velocita' media in km/h
  string location = "";                     // dove si trova attualmente
};

struct PQVeicoloElem {
  // nel dominio del problema affrontato, un elemento di una priority queue
```

```
  // e' dato dalle informazioni del veicolo e dal tempo corrente
  InfoVeicolo *veicolo;     // informazioni sul veicolo
  float timeStamp;          // tempo attuale in minuti
};

typedef PQVeicoloElem Elem;
// il tipo Elem coincide con PQVeicoloElem (la natura dell'elemento della priority
// queue potrebbe cambiare in altre implementazioni e in altri domini;
// in questo modo le definizioni continueranno a fare riferimento a Elem)

struct dynamicArray {
    // struct "standard" per la rappresentazione
    // del TDD lista mediante array dinamico
    Elem* data;
    int size;
    int maxsize;
};

typedef dynamicArray PriorityQueue;
// una coda con priorita' realizzata con uno heap si implementa con un array dinamico,
// ma possiamo per semplicita' trascurare il problema di allocazione e deallocazione:
// assumiamo che lo heap venga creato con una certa dimensione passata come argomento
// a createEmptyHeap e quella rimanga per tutta l'esecuzione del programma

// crea una PriorityQueue di dimensione dim
PriorityQueue createEmptyPQ(int);

// verifica se una PriorityQueue e' vuota
bool isEmpty(const PriorityQueue&);

// inserisce un Elem in una PriorityQueue (nella posizione corretta) e ritorna TRUE
// nel caso in cui la PriorityQueue sia piena ritorna FALSE
// suggerimento: inserire in fondo e poi chiamare la funzione ausiliaria moveUp(...)
//   per ripristinare proprieta' dell'ordinamento a heap
bool insert(PriorityQueue&, const Elem&);

// ritorna FALSE se PriorityQueue e' vuota
// altrimenti TRUE e l'Elem minimo nella PriorityQueue in res
// (cioe' quello radice in posizione 0)
bool findMin(const PriorityQueue&, Elem&);

// ritorna FALSE se PriorityQueue e' vuota
// altrimenti TRUE ed elimina l'Elem minimo nella PriorityQueue
// (cioe' quello radice in posizione 0)
// suggerimento: sostituiamo la radice dell'albero con l'ultimo elemento
//   e poi chiamare la funzione ausiliaria moveDown(...)
//   per ripristinare proprieta' dell'ordinamento a heap
bool deleteMin(PriorityQueue&);
}
```

E' possibile chiaramente inserire altre funzioni ausiliarie utili per l'implementazione.

5 Tests manuali

Il file `main.cpp` contiene il `main` di un programma per aiutarvi a svolgere dei tests, in modo simile a quanto fatto nei precedenti laboratori.
Per potere usare questo programma con la vostra nuova implementazione, potete compilarlo così:
`g++ -std=c++11 -Wall *.cpp -o main`
e poi eseguirlo con `./main`.

Nella traccia trovate anche 10 file di input che riportano le informazioni per eseguire 10 simulazioni nel formato descritto sopra. Vi consigliamo di crearvi anche dei file con tragitti e velocità semplificati in modo da verificare il funzionamento della vostra implementazione in modo semplice.
Inoltre, per alcuni di questi file vi riportiamo anche il rispettivo file di output contenente il risultato (atteso) della simulazione stampato dal programma.

6 Consegna

Per la consegna, creare uno `zip` con tutti i file forniti.