Programozás Alapjai 1
Nagy Házi Feladat
Megoldási vázlat

Karácsonyi közlekedés

Szeretnénk megtudni, hogy karácsonykor hogyan tudunk Budapesten belül eljutni a szeretteinkhez tömegközlekedéssel, amikor a tömegközlekedési eszközök megváltozott menetrenddel járnak. A program futásához szükségünk lesz az indulási időre és pozícióra, továbbá az érkezési pozícióra és a program megkeresi a számunkra az időben a legrövidebb utat.

Forrásfájlok

A különböző viszonylatok, amin a járatok járnak a **routes.txt** nevű szöveges fájlban tároljuk, melyben pontos vesszővel elválasztva szerepelnek a viszonylatok azonosítói (4 karakterből áll), a viszonylatok neve (max 4 karakter), valamint a viszonylat típusának a kódja (pl.: 1 – metró).

```
0075;7E;3
6300;H7;109
8120;D12;4
VP24;24;3
```

A különböző járatokat a **trips.txt** nevű szöveges fájlban tároljuk, melyben pontos vesszővel elválasztva szerepelnek az adott járathoz tartozó viszonylat azonosítója (4 karakterből áll), a járat azonosítója (10 karakterből áll), valamint a járat végállomásának a neve (max 45 karakter).

```
1500;B5018683;Újbuda-központ M
3030;B72450241J;Nagykőrösi út / Határ út
1380;B8116150;Csepel, Szent Imre tér
1820;B837866086;Alacskai úti lakótelep
```

A megállók adatait a **stops.txt** nevű szöveges fájlban tároljuk, melyben pontos vesszővel elválasztva szerepelnek a megállók azonosítói (6 karakterből áll), a megállók neve (max 45 karakter), valamint a megállók pozíciójára vonatkozó GPS koordinálták.

```
008593;Móricz Zsigmond körtér M;47,476836;19,047188
008548;Gyál, Bem József utca;47,372935;19,216779
008461;Erzsébet királyné útja / Róna utca;47,521615;19,101317
008086;Széll Kálmán tér M;47,507350;19,026352
```

Az adott járatnak az egyes megállóit a **stop_times.txt** nevű szöveges fájlban tároljuk, melyben pontos vesszővel elválasztva szerepelnek az adott járaton az aktuális megálló azonosítója (6 karakterből áll), az adott járat azonosítója (10 karakterből áll), az a idő amikor az adott állomásra megérkezik és elindul a járat, valamint, hogy hányadik megálló a járaton.

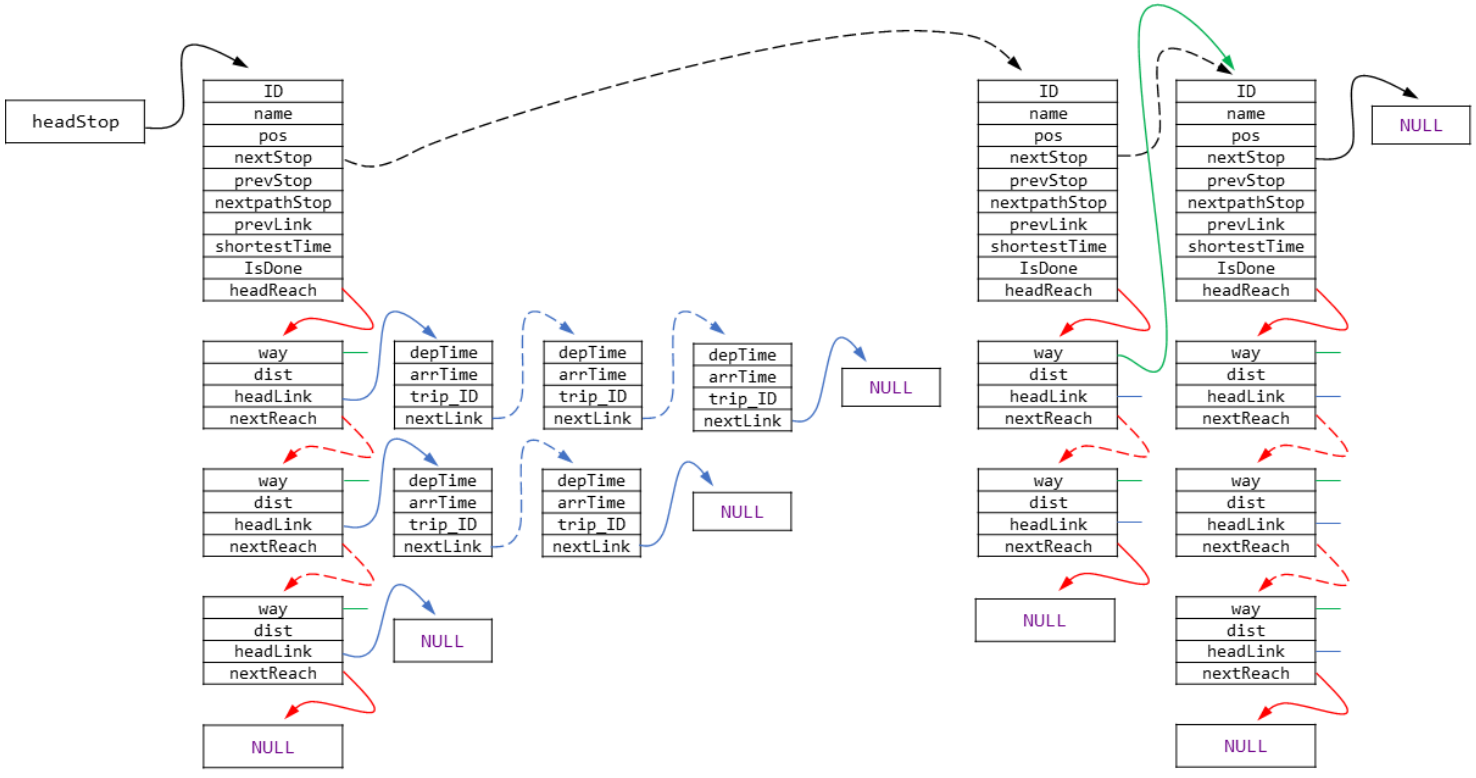
```
F01749;B1962215;09:58:45;09:58:45;0
F01742;B1962215;10:00:41;10:00:52;1
F01324;B1962215;10:02:47;10:02:57;2
F04181;B0571716;09:33:00;09:33:00;0
```

A program ennek a négy forrásfájlnak a felhasználásával építi fel az adatszerkezet. A különböző állományok között vannak kapcsolatok. A **`stops.txt`** és a **`stop_times.txt`** között a megállók azonosítója a közös kulcs, a **`stop_times.txt`** és a **`trips.txt`** között az adott járat azonosítója a közös kulcs, valamint a **`trips.txt`** és a **`routes.txt`** között a viszonylat azonosítója a közös kulcs.

Adatszerkezet

Az adott állományok alapján felrajzolható egy irányított gráf. Ahol a csúcsokat a megállók jelentik, az éleket pedig az, hogy el lehet-e jutni gyalog (nyilván nem akármekkora távolságot teszünk meg gyalog) vagy valamely közlekedési eszközzel a két megálló között. Mivel ugye a program célja, hogy minél gyorsabban eljussunk egyik pontból egy másik pontba, adott indulási idővel, szükségünk van rá, hogy tudjuk az adott élen mikor indul el a járat és mikor érkezik meg. Ez nyilvánvalóan két csúcs között több párhuzamos irányított élt fog eredményezni.

Mivel nem tudjuk mennyi adatot fogunk kapni emiatt mindenképp dinamikus adatszerkezetre van szükség. A rendelkezésre álló adatokat olyan adatszerkezetben érdemes tárolni, mely megkönnyíti az egyes megállók között való haladás megtervezését. Sok esetben egy szomszédsági mátrix jó választás lenne, de az esetünkben a csúcsok nagy száma miatt, valamint az indulási és érkezési idők tárolási szempontjából ez nem lenne túl logikus. Ehelyett olyan adatszerkezet használok, ahol a megállókat egy láncolt listában tárolom. Az adott megállókban pedig egy olyan pointert tárolok, ami egy láncolt lista kezdő címére mutat, amit azt tudja, hogy melyik megállókba lehet eljutni az adott megállókból. Ezekben az elérési elemekben eltárolom azt a mutatót, ami arra a csúcsra mutat, amit abból el tudok érni, valamint egy olyan pointer, ami egy láncolt lista kezdő címére mutat. Ez a láncolt lista az indulási és érkezési időket, valamint a járatot fogja eltárolni egy-egy struktúrában.



Az egyes megállókhoz tartalmazó láncolt lista, ami a gerinc elemet tartalmazza, `Stop` struktúrában tárolom.

```
typedef struct Stop        //állomás
{
  char ID[7];              //állomás ID-je, 6 karakter
  char name[45];           //állomás neve
  Position pos;            //állomás GPS koordinátái
  struct Stop* nextStop;   //adatszerkezetben a következő elem címe
  struct Stop* prevStop;
  struct Link* prevLink;
  struct Stop* nextpathStop;
  int shortestTime;
  int IsDone;
  struct Reach* headReach; //az adatszerkezetben az általa elérhető csúcsok címeit
} Stop;                      tartalmazó lista kezdőcíme
```

A `struct Stop* nextStop` mutat a következő megállóra a láncolt-listában, a `struct Reach* headReach` az elérési láncolt-lista kezdőcímét tárolja el. A `struct Stop* prevStop`, a `struct Link* prevLink`, a `struct Stop* nextpathStop`, az `int shortestTime` és az `int IsDone` a bejáró algoritmus futása szempontjából a különböző adatok tárolására szolgál, az adatszerkezet szempontjából nincs rá szükség. A megállók helyzetét a `Position` struktúra tárolja, aminek a tagjai két darab GPS koordináta, a szelességi és a hosszúsági kör.

```
typedef struct Position    //GPS koordináták
{
  double lat;              //szélességi kör
  double lon;              //hosszúsági kör
} Position;
```

A `Reach` struktúra felelős azért, hogy felépítsük az elérési-listát.

```
typedef struct Reach       //elérési elem
{
  Stop* way;               //amit elérek
  int dist;                //gyalog milyen távol van
  struct Link* headLink;   //hogyan jutok el a megállóba, lácolt-listára mutat
  struct Reach* nextReach; //az adatszerkezetben a következő elem címe
}Reach;
```

A `Stop* way` mutat arra a megállóra, amit ebből az elérésből el tudunk érni. A `int dist` szolgál arra, hogy eltároljuk hogy gyalog el lehet-e érni a megállót és ha igen akkor méterben milyen távol van. A `struct Link* headLink` mutat arra láncolt-lista kezdőcímére ami eltárolja, hogy tudok eljutni az egyik megállóból a másikba. A `struct Reach* nextReach` mutat a következő elérési elemre a láncolt-listában.

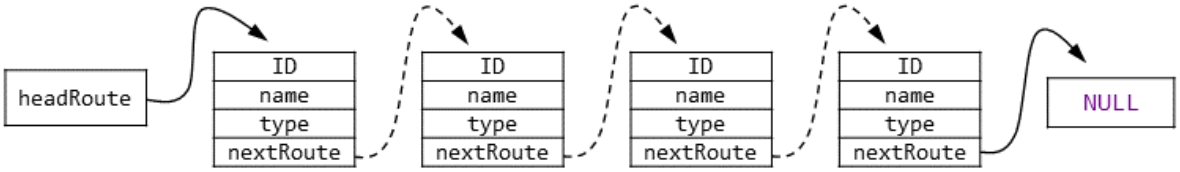
A `Link` struktúra felelős azért, hogy eltároljuk hogy jutunk el az egyik megállóból a másikba.

```
typedef struct Link        //egy él (járat)
{
  int depTime;             //indulási idő
  int arrTime;             //érkezési idő
  char trip_ID[11];        //járat azonosítója
  struct Link* nextLink;   //az adatszerkezetben a következő elem címe
}Link;
```

Az `int depTime` felel az adott jármű indulási ideéjért az adott megállóból, az `int arrTime` pedig az érkezési ideéjért fele az adott megállóba. A `char trip_ID[11]` tárolja el a járat azonosítóját. A `struct Link* nextLink` mutat a következő elemre a láncolt-listában.

Ezt a fő adatszerkezetet a **`stops.txt`** és a **`stop_times.txt`** szöveges állomány adatai alapján fogom felépíteni, aminek a folyamatát a későbbiekben le is fogom leírni.

A viszonylatokat egy láncolt-listában fogom eltárolni, amit **`routes.txt`**-ből fogok felépíteni.



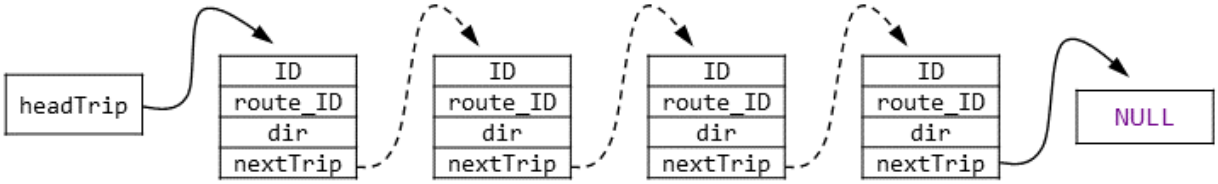
Ez a láncolt-lista fogja a viszonylatokat eltárolni és `Route` struktúrákból fog állni.

```
typedef struct Route        //viszonylat
{
  char ID[5];               // viszonylat ID-je, 4 karakter
  char name[5];             // viszonylat neve, max 4 karakter
  RouteType type;           // viszonylat típus
  struct Route* nextRoute; //az adatszerkezetben a következő elem címe
} Route;
```

A `struct Route* nextRoute` mutat a következő viszonylatra a láncolt-listában. A `RouteType type` mutatja meg, hogy milyen fajta ez a viszonylat, amit egy `RouteType` felsorolt típussal oldottam meg.

```
typedef enum RouteType      //viszonylat típus
{
      Tram, Metro, Bus, Trolleybus, Ferry, HÉV
} RouteType;
```

A járatokat egy láncolt-listában fogom eltárolni, amit **`trips.txt`**-ből fogok felépíteni.




```
typedef struct Trip               //egy járat
{
  char ID[11];                    //út ID-je, max 10 karakter
  char route_ID[5];               //az adott viszonylat
  char dir[45];                   //az járat iránya
  struct Trip* nextTrip;          //az adatszerkezetben a következő elem címe
} Trip;
```

A `char route_ID` tárolja el a viszonylat azonosítóját, ezt azért nem pointerrel csinálom mert akkor mindegyik `Trip`-hez meg kellett volna keresni a hozzátartozó `Route`-ot, de nekem csak a kiírásnál van szükségem rá és ott is csak azoknál amelyik járat rajta van a megtalált úton. Ez lehet, hogy több memóriát használ, de nem veszi el a processzor időt a többi függvénytől. A `struct Trip* nextTrip` mutat a következő járatra a láncolt-listában.

Adatok beolvasása

Kezdetben a megállókat kell beolvasni, amelyet a **`stops.txt`** állományban találjuk, hogy aztán később arra rá tudjuk építeni a többi adatot. A program megnyitja a feldolgozandó szöveges fájlt, majd az adatok beolvasását sorról sorra végzi. Tehát egy adott sort az új-sor karakterig olvasunk, majd folytatjuk a beolvasást a következő sor elején. Mivel ugye minden sor egy-egy megállót tartalmaz, amint beolvastunk egy sort azt bele rakjuk egy `Stop` struktúrába, majd hozzá is adjuk a láncolt-listához verem módszerrel. Végül miután a teljes **`stops.txt`** állományt beolvasta a program, a beolvasó program visszaadja, az utoljára beolvasott megállóra mutató pointert.

Miután már eltároltuk a megállókat, után már beolvashatjuk a **`stop_times.txt`** állományt, ami azt tárolja el, hogy mikor és hogyan juthatunk el az egyik megállóból a másikba. A program megnyitja a feldolgozandó szöveges fájlt, majd az adatok beolvasását sorról sorra végzi. Mivel ugye egy sorban nincs eltárolva, hogy melyik megállóból melyik megállóba megyünk ezért szükségünk van az előző sor tartalmára is. Amikor az adott beolvasott sort hozzá akarjuk adni az adatszerkezethez, akkor először megkeressük az előzőleg beolvasott megállót a megállók között mert ugye az mutatja meg hogy honnan megyünk. A már megtalált megálló elérési listájában megkeressük, hogy előzőleg már van-e olyan elérés, ami az újonnan beolvasott sor megállójára mutat. Ha nincs akkor létre hozunk egy új elérést. A megtalált vagy létrehozott elérésnél, az a láncolt-lista, ami megmutatja, hogy milyen módon jutunk el egyikből a másik megállóba (`Link`), ahhoz a láncolt-listához egyszerű verem módszerrel hozzá adjuk azokat az adatokat, amit beolvastunk.

A viszonylatokat a **`routes.txt`** állományban találjuk. A program megnyitja a feldolgozandó szöveges fájlt, majd az adatok beolvasását sorról sorra végzi. Tehát egy adott sort az új-sor karakterig olvasunk, majd folytatjuk a beolvasást a következő sor elején. Mivel ugye minden sor egy-egy viszonylatot tartalmaz, amint beolvastunk egy sort azt bele rakjuk egy `Route` struktúrába, majd hozzá is adjuk a láncolt-listához verem módszerrel. Végül miután a teljes **`routes.txt`** állományt beolvasta a program, a beolvasó program visszaadja, az utoljára beolvasott viszonylatra mutató pointert.

A járatokat a **`trips.txt`** állományban találjuk. A program megnyitja a feldolgozandó szöveges fájlt, majd az adatok beolvasását sorról sorra végzi. Tehát egy adott sort az új-sor karakterig olvasunk, majd folytatjuk a beolvasást a következő sor elején. Mivel ugye minden sor egy-egy járatot tartalmaz, amint beolvastunk egy sort azt bele rakjuk egy `Trip` struktúrába, majd hozzá is adjuk a láncolt-listához verem módszerrel. Végül miután a teljes **`trips.txt`** állományt beolvasta a program, a beolvasó program visszaadja, az utoljára beolvasott járatra mutató pointert.

Algoritmusok

A program bemenetként szolgál két pozíciót megfeleltetjük, két megállónak majd ezeket hozzá adjuk a többi megállóhoz verem módszerel. Miután ezt a két csúcsot hozzá adtuk a gráfhoz, a GPS koordináták alapján megvizsgáljuk, hogy mely megállók érthetők el gyalog is, ezt hozzá adjuk az adott megállok elérési eleméhez. Ha már valamilyen járattal el lehet jutni

az egyik megállóból a másikba akkor simán a stukatúrában a `int dist` változót átállítjuk a két megállónak a távolságára, ha meg még nincs kapcsolat a két megálló között akkor létre hozunk egy új elérési elemet, amit hozzá adunk az elérési listához, amelyben eltároljuk a távolságot.

A kitűzött feladat megvalósításához szükségem van egy gráf bejáró algoritmusra, amely egy legrövidebb utak feszitőfáját határozza meg. Számomra Dijkstra-algoritmus tűnt a legmegfelelőbbnek a probléma megvalósításához. Az algoritmus inputja egy súlyozott gráf és a gráf egy csúcsa. Jelen esetben ez az a megálló, ahonnan mi elindulunk a gráf éleinek a súlya pedig az, hogy mennyi idő eljutni az egyik megállóból a másik megállóba. Viszont ez ugye függ attól, hogy mi az indulási idő. Kezdetben az elért megállók halmaza csak a kezdő csúcsot tartalmazza és az elérési ideje az indulás idő. Az összes többi csúcs elérési ideje végtelen. A kezdő csúcsból megpróbáljuk javítani az elérési időket és miután az aktuális csúcsból az összes elérhető csúcsot megnéztük, végig megyünk a kész halmazba be nem vett csúcsok halmazán és amelyiknél az elérési idő a legkisebb azt bele vesszük a kész halmazban, majd a halmazba bele vett csúcsból javítjuk az elérési időket. Az algoritmus kimenetele egy legrövidebb utak feszitőfája lesz. Ezután pedig abból amelyik csúcsba el akarunk jutni visszafejtük a legrövidebb utat egészen a kezdő csúcsig, majd végül kiírjuk az eredményt.