Programozás Alapjai 1
Nagy Házi Feladat
Dokumentáció

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.

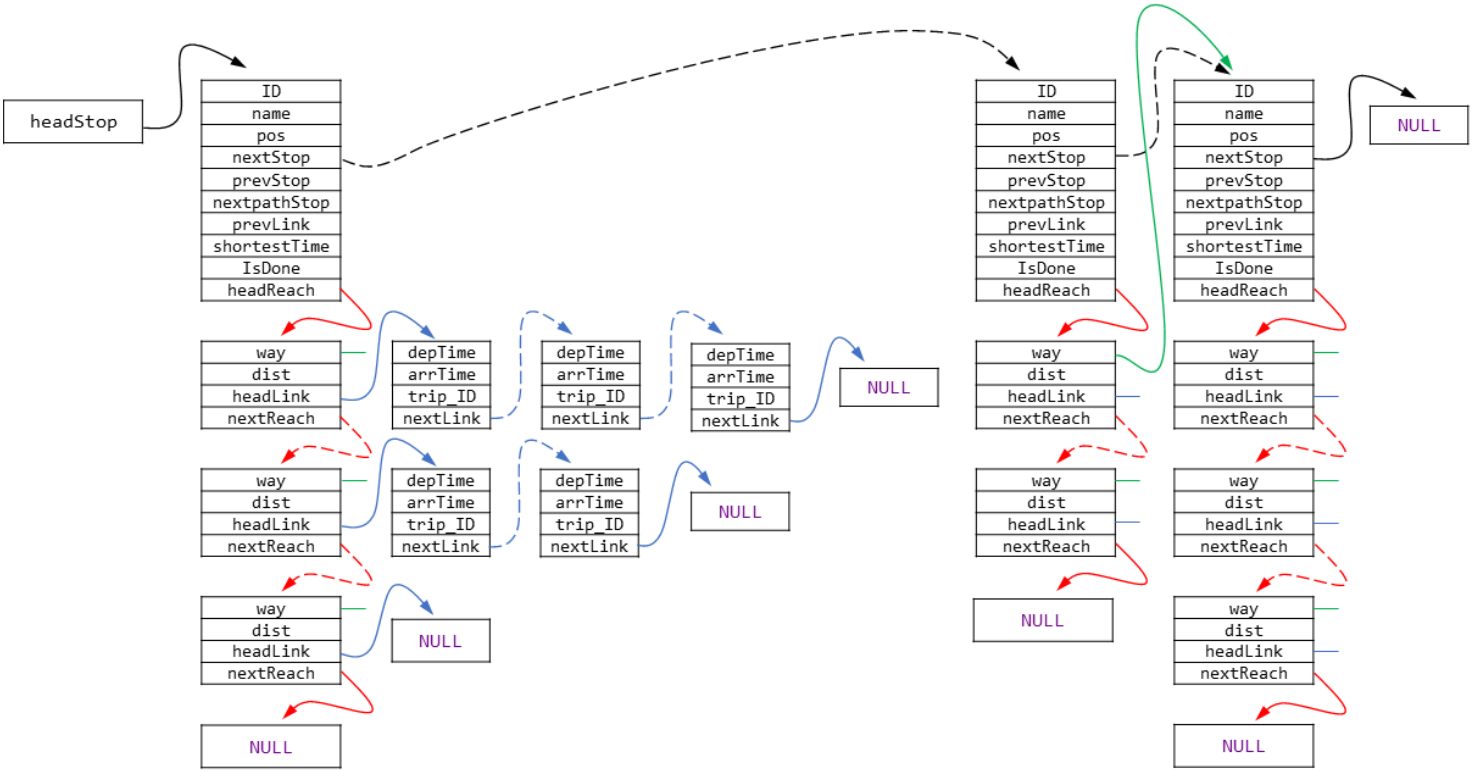
```
F01749;B1962215;09:58:45;09:58:45;0
F01742;B1962215;10:00:41;10:00:52;1
F01324;B1962215;10:02:47;10:02:57;2
F04181;B0571716;09:33:00;09:33:00;0
```

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.

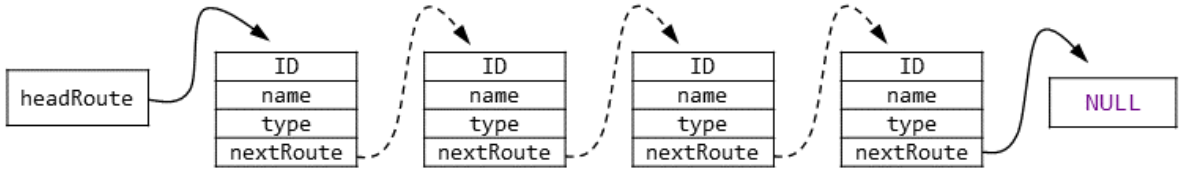
A `Link` struktúra felelős azért, hogy eltároljuk hogy jutunk el az egyik megállóból a másikba.

```
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.



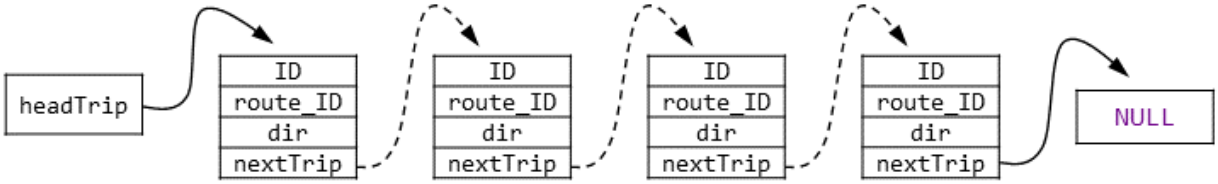
Ez a láncolt-lista fogja a viszonylatokat eltárolni és `Route` struktúrákból fog állni.

```
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.

Programszerkezet

A program futásához elengedhetetlen, hogy még a program elején meghívjuk a használt segéd könyvtárakat.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <wchar.h>
#include <locale.h>
```

A `<stdio.h>` a standard ki- és bementen olvasásához és írásához, valamint az állományokból való beolvasáshoz szükséges. A `<stdlib.h>` könyvtár, a dinamikus memóriakezeléshez szükséges. A `<string.h>` könyvtár, a string-ek másolásához, összehasonlításához szükségesek. A `<math.h>` könyvtár a matematikai függvények használatához szükségesek, amelyeket a távolság számításakor használunk főként. A `<wchar.h>` és a `<locale.h>` könyvtárak a magyar karakterek kiírásáért feltétlenül szükségesek.

A main függvény elején fontos változók kerülnek deklarálásra.

```
int main(void)
{
	setlocale(LC_ALL, "");
	double walk_speed = 1.4;
	int max_walk_dist = 500;
	int new_search = 1;
	Stop* headStop = readStops();
	readStopTimes(headStop);
	Trip* headTrip = readTrips();
	Route* headRoute = readRoutes();
	CreateWalk(headStop, max_walk_dist);
	Stop* startStop = NULL,* endStop = NULL;
	Time startTime;

	while (new_search)
	{
		startStop = (Stop*)malloc(sizeof(Stop));
		endStop = (Stop*)malloc(sizeof(Stop));
		Input(startStop, endStop, &startTime);
		headStop = AddPoint(headStop, startStop);
		headStop = AddPoint(headStop, endStop);
		CreateStartEndWalk(headStop, max_walk_dist);

		Dijkstra(headStop, startStop, GetTimeInSec(startTime), walk_speed);

		if (endStop->IsDone == 0)
			PrintNotFound();
		else
			PrintPath(startStop, endStop, headTrip, headRoute, walk_speed);

		FreeDijkstra(headStop);
		new_search = ISNewSearch();
	}
	FreeStops(headStop);
	FreeTrips(headTrip);
	FreeRoutes(headRoute);
	return 0;
}
```

A `setlocale(LC_ALL, "");` felelős azért, hogy az állományok beolvasásakor és a standard kimentre való írásakor a magyar karakterek rendesen jelenjenek meg.

A `double walk_speed` változó értéke alapján számolja a program, hogy egy adott távolságot milyen sebességgel tudjuk megtenni. A `int max_walk_dist` változó azért felelős, hogy tudjuk, hogy az egyes megállók hány méteres körzetében vizsgáljuk meg, hogy ott található-e másik megálló és ha igen akkor létre hoz közöttük egy-egy kapcsolatot, amely azt mutatja meg ha gyalog akarjuk megtenni a két megálló közti távolságot akkor hány métert kell gyalogolnunk.

A `int new_search` változó felel azért, hogy eltárolja, hogy miután bekértük az adatokat és lefutott a keresés a felhasználó szeretne-e új keresést indítani vagy bezárja a programot. Ez egy logikai értéket tárol, amely kezdetben `1` (igaz) értéket tartalmaz, amely ahhoz szükséges, hogy először mindenképpen belépjen a `while` ciklusba.

A `Stop* headStop = readStops();` olvassuk be a megállókat és a láncolt-lista kezdő címét eltároljuk. A `readStopTimes(headStop);` olvassuk be a járatok megállóit, tartalmazó állományokat és ezzel egyidőben a felépítjük a fésűs adatszerkezetet, melynek a gerinc eleme a megállók láncolt-listája, emiatt van szükség arra, hogy az argumentumban megadjuk ennek a láncolt-listának a kezdőcímét. A `Trip* headTrip = readTrips();` olvassuk be a járatokat és a láncolt-lista kezdő címét eltároljuk. A `Route* headRoute = readRoutes();` olvassuk be a viszonylatokat és a láncolt-lista kezdő címét eltároljuk.

A megállók beolvasása a `readStops()` függvénnyel történik, amely a **`stops.txt`** állomány adatait olvassa be, majd azokat tárolja el.

```
Stop* readStops(void)
{
	Stop* headStop = NULL;
	Stop* buff;
	FILE* stops_fp = fopen("stops.txt", "r");
	if (stops_fp == NULL)			//ha nem tudja megnyitni a fájlt
	{
		printf("\nError! stops.txt not found!\n");
		exit(-1);
		return NULL;
	}
	buff = (Stop*)malloc(sizeof(Stop));
	while (readStop(stops_fp, buff)) //addig olvas ameddig a végéig nem ér a fáljnak
	{
		buff->headReach = NULL;
		buff->prevStop = NULL;
		buff->prevLink = NULL;
		buff->nextpathStop = NULL;
		buff->shortestTime = -1;
		buff->IsDone = 0;
		buff->nextStop = headStop;
		headStop = buff;
		buff = (Stop*)malloc(sizeof(Stop));
	}
	free(buff);
	fclose(stops_fp);
	printf("Stops are ready!\n");
	return headStop;
}
```

Látszik, hogy a függvény folyamatosan előre lefoglalja a helyet a memóriában az újonnan beolvasott megállóknak, hogy legyen hova a beolvasó függvénynek beolvasni. Ezeket a megállókat a függvény verem szerűen feltölti a már meglévő láncolt-listába. Miután a beolvasó függvény már nem tud többet beolvasni, az utoljára foglalt memóriát felszabadítja és a fájlt, amiből olvasott bezárja. Végül pedig a láncolt-lista kezdő címére mutató pointert visszaadja. A megálló beolvasását a `readStop()` függvény végzi el.

```
int readStop(FILE* fp, Stop* buffer)
{
	for (int i = 0; i < 6; i++)
		if (fscanf(fp, "%c", &buffer->ID[i]) != 1)   //ha már nem tud beolvasni
			return 0;
	buffer->ID[6] = '\0';
	while (fgetc(fp) != ';');
	char name[45];
	fscanf(fp, "%[^;]", name);
	strcpy(buffer->name, name);
	Position pos;
	while (fgetc(fp) != ';');
	fscanf(fp, "%lf", &pos.lat);
	while (fgetc(fp) != ';');
	fscanf(fp, "%lf", &pos.lon);
	buffer->pos = pos;
	while (fgetc(fp) != '\n');
	return 1;
}
```

A függvény megkapja paraméterként az adott állományra mutató pointert, valamint az aktuális megálló címét. Az állomány egy sorát olvassa be, ahol először beolvassa a megálló azonosítóját. Ha már itt nem tudott beolvasni, tehát az fájl végéhez ért akkor visszaad a függvény egy 0 értéket. Ezután beolvassa a legfeljebb 45 karakteres megálló nevét, majd a megálló GPS koordinátáját. És ezzel egyidőben be is illeszti az adatokat a `buffer`-be. Végül visszaad egy 1 értéket, tehát hogy a beolvasás sikeres volt.

A megállók adatait tartalmazó lista, azaz a fésű gerince el is készült. A fésűk fogainak, azaz az elérési lista, valamint azoknak a fogainak a feltöltése még hátra van. Ezt a folyamatot a `readStopTimes()` függvény végez, aminek a paraméterének tudnunk kell a megállók tartalmazó láncolt-listának a kezdő címét. A beolvasást a **`stop_times.txt`** állományból történik.

```
void readStopTimes(Stop* headStop)
{
	FILE* stops_times_fp = fopen("stop_times.txt", "r");
	if (stops_times_fp == NULL)		//ha nem tudja megnyitni a fájlt
	{
		printf("\nError! stop_times.txt not found!\n");
		exit(-1);
		return NULL;
	}

	StopTime* buff = (StopTime*)malloc(sizeof(StopTime));
	Stop* from = NULL;			//melyik megálló elérésébe fogja belerakni
	char trip_ID[11];
	int depTime;
	while (readStopTime(stops_times_fp, buff))	//addig olvas ameddig csak tud
	{
```

```
            if (buff->seq == 0)        //ha a járat most indul el a végállomásról
            {
                  from = searchStop(buff->stop_ID, headStop);
                  strcpy(trip_ID, buff->trip_ID);
                  depTime = buff->depTime;
            }
            else if(strcmp(buff->trip_ID, trip_ID) == 0)    //ha egy járat
            {
                  Reach* reach = searchReach(buff->stop_ID, from, headStop);
                  Link* link = (Link*)malloc(sizeof(Link));
                  link->depTime = depTime;
                  link->arrTime = buff->arrTime;
                  strcpy(link->trip_ID, buff->trip_ID);
                  link->nextLink = reach->headLink;
                  reach->headLink = link;
                  from = searchStop(buff->stop_ID, headStop);
                  depTime = buff->depTime;
            }
            else                  //helyes adatok esetén nem fog ide jutni a program
                  printf("\nReading ERROR in stop_times.txt!");
      }
      free(buff);
      fclose(stops_times_fp);
      printf("Reaches are ready!\n");
      printf("Links are ready!\n");
}
```

Látszik, hogy a függvény folyamatosan előre lefoglalja a helyet a memóriában az újonnan beolvasott járatok megállóinak, amely tartalmazza, hogy az adott megállóba mikor érkezik a járat és onnan mikor indul el. Viszont nekünk ezt módosítani kell nekünk olyan formátumra, hogy azt kell eltárolnunk, hogy melyik megállóból melyik megállóba megy az adott járat és a megállóból mikor indul el és a másikba mikor érezik meg. Emiatt meg kell vizsgálnunk, hogy az adott járaton az épp beolvasott megálló hányadik a sorban mert, ha a 0.-ik akkor nem kell még eltárolni semmit, csak meg kell jegyezni, hogy onnan mikor indul el és melyik megálló az. Különben pedig az épp beolvasott sorból azt tároljuk el, hogy hova mentünk az előzőleg beolvasottból. Ezt egészen addig csináljuk ameddig az előzőleg és a most beolvasott járat azonosító megegyezik. Ezután újra a 0.-ik elemhez ugrik. A beolvasás addig történik ameddig a beolvasó függvény tud beolvasni a fájlból. Miután a beolvasó függvény már nem tud többet beolvasni, az utoljára foglalt memóriát felszabadítja és a fájlt, amiből olvasott bezárja. A járat megállója beolvasását a `readStopTime()` függvény végzi el.

```
int readStopTime(FILE* fp, StopTime* buffer)
{
      Time arrTime;
      Time depTime;
      for (int i = 0; i < 6; i++)
            if (fscanf(fp, "%c", &buffer->stop_ID[i]) !=1)//ha már nem tud beolvasni
                  return 0;
      buffer->stop_ID[6] = '\0';
      while (fgetc(fp) != ';');
      fscanf(fp, "%[^;]", buffer->trip_ID);
      while (fgetc(fp) != ';');
      fscanf(fp, "%d:%d:%d", &arrTime.h, &arrTime.m, &arrTime.s);
      while (fgetc(fp) != ';');
      fscanf(fp, "%d:%d:%d", &depTime.h, &depTime.m, &depTime.s);
      while (fgetc(fp) != ';');
      buffer->arrTime = GetTimeInSec(arrTime);
```

```
	buffer->depTime = GetTimeInSec(depTime);
	fscanf(fp, "%d", &buffer->seq);
	while (fgetc(fp) != '\n');
	return 1;
}
```

A függvény megkapja paraméterként az adott állományra mutató pointert, valamint az aktuális járat megállója címét. Az állomány egy sorát olvassa be, ahol először beolvassa a megálló azonosítóját. Ha már itt nem tudott beolvasni, tehát az fájl végéhez ért akkor visszaad a függvény egy 0 értéket. Ezután beolvassa a járat azonosítóját, valamint az érkezési és indulási időt. És ezzel egyidőben be is illeszti az adatokat a `buffer`-be. Végül visszaad egy 1 értéket, tehát hogy a beolvasás sikeres volt.

Ahhoz, hogy tudjuk hova illesszük be az eléréseket, szükségünk van arra, hogy az azonosítója alapján megkeressük a megállót. Ezt a `searchStop()` függvény végzi el.

```
Stop* searchStop(char ID[7], Stop* headStop)
{
	for (; headStop != NULL; headStop = headStop->nextStop)
		if (strcmp(ID, headStop->ID) == 0)
			return headStop;			//ha talált olyan megállót
	return NULL;					//ha nem talált olyan megállót
}
```

Végigmegy a megállókat tartalmazó láncolt-listán és egyenként megvizsgálja, hogy a paraméterként megadott azonosító megegyezik-e az épp aktuálisan vizsgált megálló azonosítójával. Ha megegyezik visszaadja a megállóra mutató pointert. Ha végig ment az összes megállón és nem talált egyezést akkor visszaad egy `NULL` pointert.

Amikor eltárolnánk a kapcsolatot a két megálló között akkor nem akarunk mindegyikre létrehozni új elérési elemet, hanem megnézzük, hogy van-e ilyen elérés, ha van akkor visszaadjuk azt, ha nincs akkor létrehozunk egy újat és azt adjuk vissza. Ezt az algoritmust végzi el a `searchReach()` függvény.

```
Reach* searchReach(char ID[7], Stop* stop, Stop* headStop)
{
	for (Reach* i = stop->headReach; i != NULL; i = i->nextReach)
		if (strcmp(ID, i->way->ID) == 0)
			return i;			//ha talált olyan elérést
	Reach* reach = (Reach*)malloc(sizeof(Reach));	//ha nem talált olyan elérést
	reach->way = searchStop(ID, headStop);
	reach->dist = -1;
	reach->headLink = NULL;
	reach->nextReach = stop->headReach;
	stop->headReach = reach;
	return reach;
}
```

A viszonylatok beolvasása a `readRoutes()` függvénnyel történik, amely a **`routes.txt`** állomány adatait olvassa be, majd azokat tárolja el.

```
Route* readRoutes(void)
{
	Route* headRoute = NULL;
	Route* buff;
	FILE* routes_fp = fopen("routes.txt", "r");
```

```
	if (routes_fp == NULL)			//ha nem tudja megnyitni a fájlt
	{
		printf("\nError! routes.txt not found!\n");
		exit(-1);
		return NULL;
	}
	buff = (Route*)malloc(sizeof(Route));
	while (readRoute(routes_fp, buff))	//addig olvas ameddig csak tud
	{
		buff->nextRoute = headRoute;
		headRoute = buff;
		buff = (Route*)malloc(sizeof(Route));
	}
	free(buff);
	fclose(routes_fp);
	printf("Routes are ready!\n");
	return headRoute;
}
```

Látszik, hogy a függvény folyamatosan előre lefoglalja a helyet a memóriában az újonnan beolvasott viszonylatnak, hogy legyen hova a beolvasó függvénynek beolvasni. Ezeket a viszonylatokat a függvény verem szerűen feltölti a már meglévő láncolt-listába. Miután a beolvasó függvény már nem tud többet beolvasni, az utoljára foglalt memóriát felszabadítja és a fájlt, amiből olvasott bezárja. Végül pedig a láncolt-lista kezdő címére mutató pointert visszaadja. A viszonylat beolvasását a `readRoute()` függvény végzi el.

```
int readRoute(FILE* fp, Route* buffer)
{
	for (int i = 0; i < 4; i++)
		if (fscanf(fp, "%c", &buffer->ID[i]) != 1) //ha már nem tud beolvasni
			return 0;
	buffer->ID[4] = '\0';
	while (fgetc(fp) != ';');
	fscanf(fp, "%[^;]", buffer->name);
	while (fgetc(fp) != ';');
	fscanf(fp, "%d", &buffer->type);
	while (fgetc(fp) != '\n');
	return 1;
}
```

A függvény megkapja paraméterként az adott állományra mutató pointert, valamint az aktuális viszonylat címét. Az állomány egy sorát olvassa be, ahol először beolvassa a viszonylat azonosítóját. Ha már itt nem tudott beolvasni, tehát az fájl végéhez ért akkor visszaad a függvény egy 0 értéket. Ezután beolvassa a viszonylat nevét, majd a viszonylat típusát. És ezzel egyidőben be is illeszti az adatokat a `buffer`-be. Végül visszaad egy 1 értéket, tehát hogy a beolvasás sikeres volt.

A járatok beolvasása a `readTrips()` függvénnyel történik, amely a **`trips.txt`** állomány adatait olvassa be, majd azokat tárolja el.

```
Trip* readTrips(void)
{
	Trip* headTrip = NULL;
	Trip* buff;
	FILE* trips_fp = fopen("trips.txt", "r");
	if (trips_fp == NULL)			//ha nem tudja megnyitni a fájlt
	{
```

```
		printf("\nError! trips.txt not found!\n");
		exit(-1);
		return NULL;
	}
	buff = (Trip*)malloc(sizeof(Trip));
	while (readTrip(trips_fp, buff)) //addig olvas ameddig a végéig nem ér a fájlnak
	{
		buff->nextTrip = headTrip;
		headTrip = buff;
		buff = (Trip*)malloc(sizeof(Trip));
	}
	free(buff);
	fclose(trips_fp);
	printf("Trips are ready!\n");
	return headTrip;
}
```

Látszik, hogy a függvény folyamatosan előre lefoglalja a helyet a memóriában az újonnan beolvasott járatoknak, hogy legyen hova a beolvasó függvénynek beolvasni. Ezeket a járatokat a függvény verem szerűen feltölti a már meglévő láncolt-listába. Miután a beolvasó függvény már nem tud többet beolvasni, az utoljára foglalt memóriát felszabadítja és a fájlt, amiből olvasott bezárja. Végül pedig a láncolt-lista kezdő címére mutató pointert visszaadja. A járat beolvasását a `readTrip()` függvény végzi el.

```
int readTrip(FILE* fp, Trip* buffer)
{
	for (int i = 0; i < 4; i++)
		if (fscanf(fp, "%c", &buffer->route_ID[i]) != 1)
			return 0;		//ha már nem tud beolvasni
	buffer->route_ID[4] = '\0';
	while (fgetc(fp) != ';');
	fscanf(fp, "%[^;]", buffer->ID);
	while (fgetc(fp) != ';');
	fscanf(fp, "%[^\n]", buffer->dir);
	while (fgetc(fp) != '\n');
	return 1;
}
```

A `CreateWalk(headStop, max_walk_dist);` felelős azért, hogy beállítsa a kapcsolatokat azok a megállók között, ahol a két megálló távolsága legfeljebb `int max_walk_dist`. Ez azért szükséges, hogy át tudjunk szállni az egyik járatról a másikra.

```
void CreateWalk(Stop* headStop, int maxDist)
{
	for (Stop* i = headStop; i != NULL; i = i->nextStop)
	{
		for (Stop* j = i->nextStop; j != NULL; j = j->nextStop)
		{
			int dist = StopsDist(i, j);
			if (dist <= maxDist)
			{
				SetWalkDist(i, j, dist);
				SetWalkDist(j, i, dist);
			}
		}
	}
	printf("Pathways are ready!\n");
}
```

Látszik, hogy megvizsgálja bármely két megállót, hogy a távolsága kisebb, mint az `int maxDist`. A távolságot a `StopsDist()` függvénnyel vizsgálja, majd a `SetWalkDist()` függvénnyel állítja be a kettő távolságát.

```
void SetWalkDist(Stop* stop, Stop* way, int dist)
{
	for (Reach* i = stop->headReach; i != NULL; i = i->nextReach)
		if (i->way == way)				//ha már szerepel az elérései között
		{
			i->dist = dist;
			return;
		}
	Reach* reach = (Reach*)malloc(sizeof(Reach));	//ha még nincs ilyen elérés
	reach->way = way;
	reach->dist = dist;
	reach->headLink = NULL;
	reach->nextReach = stop->headReach;
	stop->headReach = reach;
}
```

Látszik, hogy ha talál olyan elérést, aminél a távolságot be kell állítani akkor be is állítja majd vissza is tér a függvény. Ha végig ment az összes elérésen, de egyik sem egyezett meg akkor létrehoz egy új elérést.

A `Stop* startStop = NULL;` a `Stop* endStop = NULL;` és a `Time startTime;` azért felelősek, amikor beolvassuk a felhasználó által megadott adatokat, azt függvénnyel végezzük. Mivel a három érték közül maximum egyet tudnánk visszatérési értékként visszaadni, ehelyett a függvényeknek a változók pointerét adjuk meg, amit a függvény megtud hívni és módosítani az értékeket.

```
void Input(Stop* startStop, Stop* endStop, Time* startTime)
{
	system("cls");
	PrintLogo();
	printf("\nPlease enter the start time (hh:mm): ");
	scanf("%d:%d:%d", &startTime->h, &startTime->m);
	startTime->s = 0;
	printf("\nPlease enter the start position!");
	printf("\nLatitude: ");
	scanf("%lf", &startStop->pos.lat);
	printf("Longitude: ");
	scanf("%lf", &startStop->pos.lon);
	strcpy(startStop->ID, "000000");
	strcpy(startStop->name, "Start position");
	printf("\nPlease enter the end position!");
	printf("\nLatitude: ");
	scanf("%lf", &endStop->pos.lat);
	printf("Longitude: ");
	scanf("%lf", &endStop->pos.lon);
	strcpy(endStop->ID, "999999");
	strcpy(endStop->name, "End position");
}
```

Látszik, hogy először letörli a standard kimenten lévő szöveget és utána kirajzolja a logót. Aztán bekéri az indulási időt, majd az indulási pozíciót végül pedig az érkezési pozíciót.

Miután belettek olvasva az indulási és érkezi pozíciók, amik megállóként vannak értelmezve hozzá is adjuk a megállókat tartalmazó láncolt-listához. Ezután hasonló módon, mint az előbb csak erre a kettő pozícióra létrehozzuk a kapcsolatot a közeli megállókkal.

Ezek után már minden készen áll arra, hogy meghívjuk a legrövidebb utak feszítőfáját készítő Dijkstra algoritmust. Ehhez szükségünk van arra, hogy megadjuk a megállók kezdőcímére mutató pointert, a kiindulási megállót, az indulási időt, valamint a gyaloglás sebességét.

```
void Dijkstra(Stop* headStop, Stop* startStop, int startTime, double walk_speed)
{
  int IsDone = 0;
  startStop->shortestTime = startTime;
  Stop* current = GetMinUnfinshed(headStop);
  if (current == NULL)
    IsDone = 1;
  else
    current->IsDone = 1;
  while (!IsDone)
  {
    for (Reach* i = current->headReach; i != NULL; i = i->nextReach)
    {
       Link* firstLink = GetTheFirstLink(i, current->shortestTime);
       if (firstLink != NULL && i->dist == -1)
       {
         if (firstLink->arrTime <= i->way->shortestTime || i->way->shortestTime == -1)
         {
              i->way->shortestTime = firstLink->arrTime;
              i->way->prevStop = current;
              i->way->prevLink = firstLink;
         }
       }
       else if(firstLink != NULL && i->dist != -1)
       {
         if (firstLink->arrTime <= (int)(i->dist / walk_speed) + current->shortestTime)
         {
              if (firstLink->arrTime <= i->way->shortestTime||i->way->shortestTime==-1)
              {
                i->way->shortestTime = firstLink->arrTime;
                i->way->prevStop = current;
                i->way->prevLink = firstLink;
              }
         }
         else if (firstLink->arrTime > (int)(i->dist/walk_speed)+current->shortestTime)
         {
              if ((int)(i->dist/walk_speed)+current->shortestTime<=i->way->shortestTime
                                                    || i->way->shortestTime == -1)
              {
                i->way->shortestTime = (int)(i->dist/walk_speed)+current->shortestTime;
                i->way->prevStop = current;
                Link* prevLink = (Link*)malloc(sizeof(Link));
                prevLink->nextLink = NULL;
                strcpy(prevLink->trip_ID, "0000000000");
                prevLink->depTime = current->shortestTime;
                prevLink->arrTime = (int)(i->dist/walk_speed) + current->shortestTime;
                i->way->prevLink = prevLink;
              }
         }
       }
       else if (firstLink == NULL && i->dist != -1)
       {
```

```
        if ((int)(i->dist / walk_speed) + current->shortestTime < i->way->shortestTime
                                                        || i->way->shortestTime == -1)
        {
             i->way->shortestTime = (int)(i->dist/walk_speed) + current->shortestTime;
             i->way->prevStop = current;
             Link* prevLink = (Link*)malloc(sizeof(Link));
             prevLink->nextLink = NULL;
             strcpy(prevLink->trip_ID, "0000000000");
             prevLink->depTime = current->shortestTime;
             prevLink->arrTime = (int)(i->dist / walk_speed) + current->shortestTime;
             i->way->prevLink = prevLink;
        }
      }
      else {}
  }
  current = GetMinUnfinshed(headStop);
  if (current == NULL)
       IsDone = 1;
  else
       current->IsDone = 1;
  }
}
```

Látszik, hogy először beállítjuk a kezdőcsúcs elérési idejét az indulási időre. Így a `GetMinUnfinshed()` függvény, ami visszaadja a leghamarabb elért csúcsot, ami még nincs benne a kész halmazba az a kezdőcsúcsot fogja visszaadni. Amit visszaadott csúcsot aza abból elérhető csúcsokkal javításokat végzünk az elérési időre. Kezdetben minden csúcs elérési ideje -1 (az eredeti algoritmus szerint végtelen, de azt nem tudnánk eltárolni így a -1-et elnevezzük végtelennek) és a javítások közül négy különböző esetet különböztetünk meg. Az adott elérésnél, amit épp vizsgálunk a `GetTheFirstLink()` függvény visszaadja leghamarabbi kapcsolatot. Az első eset amikor a visszaadott kapcsolat értelmezve van és gyalog nem lehetne elérni. A második eset amikor a visszaadott kapcsolat értelmezve van és gyalog is ellehet érni, de járattal elérni gyorsabb. A harmadik abban különbözik a másodiktól, hogy gyalog gyorsabb megtenni az adott megállóból eljutni a másikba, mint egy valamilyen járattal. Az utolsó eset pedig amikor csak gyalog lehet eljutni az egyik megállóból a másikba. Akármelyik esetben akkor van javítás, ha az így elért idő jobb lenne, mint az eredeti elérési idő. Ez a folyamat addig ismétlődik ameddig az összes lehetséges javítást meg nem vizsgáltuk.

Miután végeztünk a Dijkstra algoritmussal, meg kell vizsgálnunk, hogy a kezdő és a végpont között van-e út, ezt pedig azzal vizsgáljuk meg, hogy a végpont belekerült-e a kész halmazba tehát `endStop->IsDone == 1`. Ha az érték igaz akkor kiírathatjuk az utat amit a kereső függvény megtalált. A kiírást a .. függvénnyel történik.

```
void PrintPath(Stop* startStop, Stop* endStop, Trip* headTrip, Route* headRoute,
double walk_speed)
{
  printf("\n-----------------------------------------------------------------");
  SetPathNextStops(endStop);
  printStop(startStop);
  int n = 0;
  int time = startStop->shortestTime;
  int dif_dist = 0;
  for (Stop* i = startStop->nextpathStop; i != NULL; i = i->nextpathStop)
  {
```

```
      if (i->nextpathStop == NULL || strcmp(i->prevLink->trip_ID,
                                        i->nextpathStop->prevLink->trip_ID) != 0)
      {
        if (strcmp(i->prevLink->trip_ID, "0000000000") != 0)
        {
             Trip* trip = searchTrip(i->prevLink->trip_ID, headTrip);
             Route* route = searchRoute(trip->route_ID, headRoute);
             printf("\n\t    \t|");
             printf("\n\t    \t|");
             printf("\n\t%4s\tO\t%d stops, Direction: %s",route->name,n+1, trip->dir);
             printf("\n\t    \t|");
             printf("\n\t    \t|");
             printStop(i);
             time = i->shortestTime;
             n = 0;
        }
        else if(i->nextpathStop != NULL)
        {
             Time t = GetTimeinTime(i->nextpathStop->prevLink->depTime);
             if (t.s >= 30)
               t.m++;
             dif_dist += walk_speed * (i->shortestTime - time);
             printf("\n\t    \t|");
             printf("\n\t    \tO\t%dm walk", dif_dist);
             printf("\n\t    \t|");
             printf("\n%02d:%02d #%s %s", t.h, t.m, i->ID, i->name);
             dif_dist = 0;
        }
        else
        {
             Time t = GetTimeinTime(i->shortestTime);
             if (t.s >= 30)
               t.m++;
             dif_dist += walk_speed * (i->shortestTime - time);
             printf("\n\t    \t|");
             printf("\n\t    \tO\t%dm walk", dif_dist);
             printf("\n\t    \t|");
             printf("\n%02d:%02d #%s %s", t.h, t.m, i->ID, i->name);
             dif_dist = 0;
        }
      }
      else if (strcmp(i->prevLink->trip_ID, "0000000000") == 0 && strcmp("0000000000",
                                        i->nextpathStop->prevLink->trip_ID) == 0)
      {
        dif_dist += walk_speed * (i->shortestTime - time);
      }
      else
        n++;
  }
  printf("\n-----------------------------------------------------------------");
}
```

Látszik, hogy először a végponttól kezdve visszafejtve eltárolja előre felé a kezdőponttól a bejárás útvonalát a `SetPathNextStops()` függvénnyel. Majd után végig megy az útvonal minden egyes megállóján és esetekre bontva csak akkor van kiírás amikor vagy kezdő- vagy végpontja van az útvonalnak vagy az útvonalon gyaloglás után felszál egy járatra vagy leszál egy járatról.

Miután kiírtunk egy útvonalat a `ISNewSearch()` függvénnyel megkérdezzük a felhasználót, hogy szeretne-e új útvonalat keresni és ha igen akkor a beolvasás utáni részre vissza ugrunk, ha pedig nem akkor felszabadítjuk a memóriát.

Először a fésűs listát szabadítjuk fel a `FreeStops()` függvénnyel.

```
void FreeStops(Stop* headStop)
{
	Stop* stop;
	while (headStop != NULL)
	{
		stop = headStop;
		headStop = stop->nextStop;
		FreeReachs(stop);
		free(stop);
	}
}
```

Ezen belül minden megállónál felszabadítjuk az elérési listáját a `FreeReachs()` függvénnyel.

```
void FreeReachs(Stop* stop)
{
	Reach* reach;
	while (stop->headReach != NULL)
	{
		reach = stop->headReach;
		stop->headReach = reach->nextReach;
		FreeLink(reach);
		free(reach);
	}
}
```

Ezen belül minden elérésnél felszabadítjuk a kapcsolati listát a `FreeLink()` függvénnyel.

```
void FreeLink(Reach* reach)
{
	Link* link;
	while (reach->headLink != NULL)
	{
		link = reach->headLink;
		reach->headLink = link->nextLink;
		free(link);
	}
}
```

Miután felszabadítottuk a fésűs listát, után fel kell szabadítani a viszonylatokat a `FreeRoutes()` függvénnyel.

```
void FreeRoutes(Route* headRoute)
{
	Route* route;
	while (headRoute != NULL)
	{
		route = headRoute;
		headRoute = route->nextRoute;
		free(route);
	}
}
```

Ezután pedig a járatokat kell felszabadítani a `FreeTrips()` függvénnyel.

```
void FreeTrips(Trip* headTrip)
{
	Trip* trip;
	while (headTrip != NULL)
	{
		trip = headTrip;
		headTrip = trip->nextTrip;
		free(trip);
	}
}
```

Tesztek, tapasztalatok

Mivel a program kereső algoritmusa mindig az időben a legrövidebb útvonalat keresi ezért mindig legalább olyan gyors útvonalat talál, mint az eredeti BBK Futár. Az eltérés több magyarázat van, először is amikor kiszámoljuk két megálló távolságát nem vesszük figyelembe a földrajzi környezetet mivel nincs róla adatunk, hanem csak a légvonalban való távolságukat tudjuk meghatározni, emiatt keletkezhet különbség két megálló távolsága között. Másodszor is az a algoritmus, amit én használok mindig időben a legrövidebbet keresi meg és nem vizsgálja meg, hogy minél kevesebb átszálással.

Az alábbiakban lehet látni egy tesztet, ahol BME I épületéből szeretnénk eljutni a Vasút történeti parkba 10:38-as indulási idővel. Látható az eredeti BBK futár már alapvetően a 4-es villamosra szál fel, holott, ha először a 212 buszra szálunk fel és utána a Boráros téren átszálunk a 6-os villamosra akkor gyorsabbak voltunk igaz, hogy plusz egy átszállással. És mivel az út első részét hamarabb tettük meg ezért a későbbiekben máshol kell átszállni az egyik buszról a másikra. És összeségében a program szerint 7 perccel hamarabb érnénk oda a végponthoz.