Útvonalválaszt-O

Készítette Doxygen 1.8.20

1. fejezet

Útvonalválaszt-O Programozói Dokumentáció

1.1. Bevezető

Ezen programozói dokumentáció célja az, hogy a program belső felépítését, működését bemutassa. Ha a játékszabályok érdekelnek, akkor ajánlom a felhasználói dokumentációt. Itt a főoldalon el fogom magyarázni a megvalósított módszereket, hogy miért és milyen adatstruktúrákat használtam, a program által olvasott vagy írt fájlok szintaktikai követelményeit, valamint felsorolom azokat a hibaüzeneteket, amiket a program kiírhat, és hogy hogyan javítsd ki őket. Az egyes modulok, függvények, adatszerkezetek, változók és paraméterek részletes leírásait a fájlok aloldalain (vagy pdf-es verzió esetén a következő fejezetekben) találod.

1.2. A program belső működése

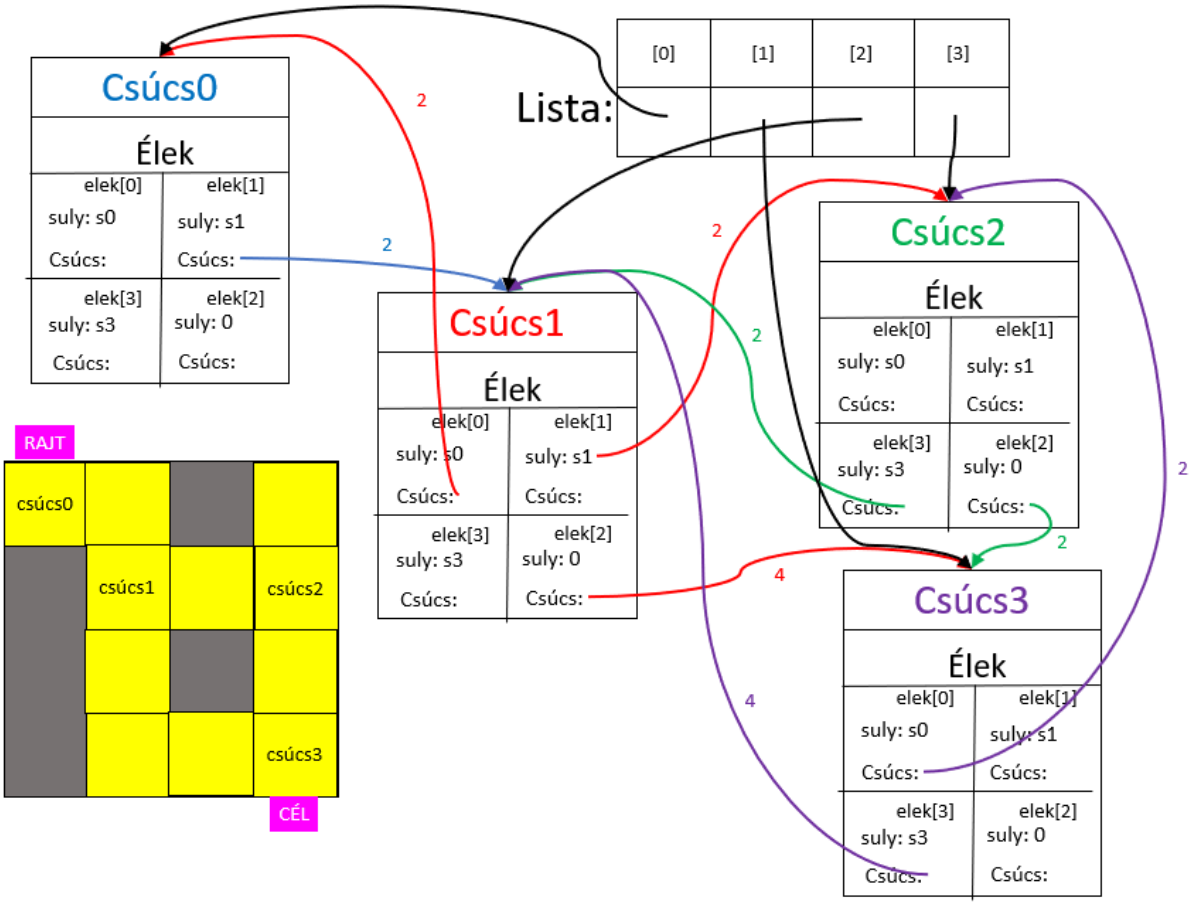
A program konzolban fut, így esztétikailag nem a legszebb, de funkcionalitásából emiatt nem vesztett semmit. A menü irányítása egyszerűen a felhasználó által begépelt számokkal történik, a szintek futása közben azonban a c-econio modul segítségével kezelem a felhasználói inputot. (A c-econio modul a githubon: https://github.com/czirkoszoltan/c-econio) Ennek a modulnak a segítségével tudom a konzolban a szöveg és a háttér színét változtatni, tetszőleges helyre szöveget írni, valamint felhasználói inputot kapni anélkül, hogy neki mindig enter-t kéne nyomnia. Ez a könyvtár is a forrásfájlok között van, így a program fordításához csak a szabványos könyvtárakra és a forrásfájlokra van szükség.

1.2.1. A program adatszerkezetei

A legnagyobb kihívás a program fejlesztése közben egyértelműen a legrövidebb útvonalat megtaláló függvények megírása volt. Ehhez szükség volt egy olyan adatstruktúrára, ami egy súlyozott gráfhoz hasonlóan működik, de az összes elemén is könnyű végigmenni.

Ehhez egy különleges láncolt listát készítettem: A megszokott láncolt listától eltérően, itt minden elem akár 4 másikhoz is kapcsolódik, sőt minden kapcsolatnak (élnek a gráfban) van súlya is. A gráfot reprezentáló adatszerkezetet egy kicsi, 4x4-es pálya példáján keresztül fogom bemutatni, és hogy könnyebb legyen megérteni, készítettem hozzá egy ábrát is. Az ábra bal alján látható a pálya. A gráf csúcsai minden pályán a rajt, a cél, valamint az elágazások (de a sima kanyarok nem!). Minden csúcs struktúrában van egy 4 elemű lista, aminek minden eleme egy él struktúra. Egy élnek két mezője van: az él súlya, és annak a csúcsnak a pointere, ahová az él mutat. Mivel a pointereken csak egyik irányban tudunk mozogni, ezért a gráfban valójában ha két csúcs között van kapcsolat, akkor köztük

két, egymással ellentétes irányítású él van, melyek súlya megegyezik, és ez a két csúcs távolsága. Az éleket tartalmazó tömb sorrendje megegyezik az Irany enum-mal, tehát pl. az elek[2] a lefele mutató élet jelenti. Ha egy csúcsnak egy adott irányban nincs szomszédja, akkor annak az élnek a súlya 0, pointere pedig NULL-ra mutat. Ebben az adatstruktúrában könnyedén eljuthatunk bármelyik csúcsból a szomszédjaihoz, ami nagyon fontos lesz a Dijkstra-algoritmus során (erről később lesz szó). Azonban nehézkes a gráf összes csúcsát sorba venni, hiszen mivel a gráfban vannak körök, könnyen végtelen ciklusba keveredhetünk. Ezért készítünk egy olyan tömböt (listát), aminek minden eleme egy csúcsra mutat. Amikor ezt a tömböt megépítjük, figyelünk arra, hogy minden csúcs pontosan egyszer szerepeljen a tömbben, és így ezután könnyen sorra vehetjük az összes csúcsot, amire szintén szükségünk lesz később. Fontos megemlíteni a tömbbel kapcsolatban, hogy a csúcsok sorrendje nem érdekel minket, bár a kezdő csúcs mindig a nulladik helyen lesz. Fontos megemlíteni még, hogy a gráf vagy a tömb a program futása során soha nem fog átméreteződni. A láncolt listához hasonló adatszerkezetre azért volt szükség, hogy egy csúcs szomszédjait könnyen elérjük. Ezen a különleges láncolt listán viszont nem tudunk csak egyszerűen végig menni, és minden elemét vizsgálni, ezért van szükség a sima tömbre. A játékot kb. 20 cella magas és 20 cella hosszú szinten érdekes játszani, és egy ilyen szinten nem nagyon lesz 20-30-nál több csúcs. Így igaz, hogy minden csúcsot gyakorlatilag több helyen is tárolunk, ez összeségében nem fog sok helyet foglalni, viszont a további algoritmusok megírását nagyban segíti. A csúcsok száma akkor lehet sok, ha a szinten egymás mellett több oszlop vagy sor is utca, hiszen ekkor tulajdonképpen minden cella egy elágazás. A program ilyenkor is megfelelően működik, azonban ilyen szinteket nincs értelme készíteni, elvégrre csak négy irányban lehet mozogni, így csak sok, egyenlő hosszúságú útvonal-lehetőséget jönne létre, ami az útvonalválasztási feladatot nem nehezítené.



A gráfon kívül a program többször használ még kétdimenziós tömböt, amelynek minden értéke a pálya egy mezőjére mutat, valamilyen formában. Ezek közül a leggyakrabban használt a Szintek struktúra terkep mezője, ahova fájlból töltődnek be a pályák (itt a szintek száma még egy harmadik dimenziót is létrehoz). Ezen kívül szükség van még egy ilyen tömbre a gráf megépítésekor. Itt miután egy csúcsnak elkészítettük a struktúráját és összekötöttük a szomszédjaival, egy ilyen tömbbe berakjuk a rá mutató pointert, arra az x-y helyre, ahol a pályán elhelyezkedik. Így amikor a többi csúcs vizsgálásánál újra visszajutunk ehhez a csúcshoz, már tudjuk, hogy ezt a csúcsot nem

kell újra elkészíteni, sőt össze is tudjuk kötni a másik csúccsal. Amikor a gráfból egy sima, egydimenziós tömböt csinálunk, akkor is egy ugyanilyen tömböt használunk, ám annak értekei boolean értékek, hiszen itt a pointerre nincs szükségünk. Az eddig felsorolt tömbök mind dinamikusan foglaltak voltak, ami kézen fekvő, hiszen méretük állandó.

Van azonban a programban egy olyan tömb is, aminek mérete nem állandó, ez pedig az eredményeket tároló tömb, a toplista. Azonban ez is dinamikus tömbként van kialakítva, hiszen mérete csak akkor változik, ha új eredményt kell betenni a listába, és annak hossza nem érte még el a tízet. Így maximum kilenc eredményt kell egy ilyen realloc()-kor másolni, és azt is csak ritkán.

1.2.2. A program legfontosabb algoritmusai

A program legfontosabb algoritmusai mind a legrövidebb útvonal meghatározáshoz szükségesek.

1.2.2.1. A gráf építése

Az algoritmust megvalósító függvény: graf_epit()

Az első lépés az ideális útvonal meghatározásában az egy gráf építése, amire majd használható lesz a Dijkstra-algoritmus. A feladat tehát az, hogy egy olyan kétdimenziós tömbből, amiben minden mező utca vagy épület, készítsünk egy olyan gráf adatstruktúrát, amiben azok a mezők lesznek a csúcsok, ahol legalább 3 utca találkozik, az élek pedig az őket összekötő utcák. Erre egy rekurzív algoritmus használtam. Habár a rajtban és a célban csak két utca találkozik, ezek is csúcsok lesznek a gráfban, hiszen ezen két pont között keressük a legrövidebb utat. Az algoritmus pszeudokódja egy adott csúcsból (a rajtból indítjuk):

1. Próbáljunk meg elindulni a gráfból egy irányba!
2. Ha arra utca van, akkor kövessük az utcát egészen addig, amíg olyan mezőhöz nem érünk, aminek 3 szomszédja is utca, azaz itt elágazás lesz. Közben számoljuk hányat lépünk! Ha arra épület volt, akkor ebben az irányban a csúcsnak nincs éle.
3. Amikor egy csúcshoz jutottunk, vizsgáljuk meg, hogy jártunk-e már ebben a csúcsban!
4. Ha nem, akkor készítsük el a csúcsot, és határozzuk meg, melyik irányból érkeztünk ide.
5. Állítsuk be a megfelelő éleket mindkét csúcson, hogy egymásra mutassanak, a súly pedig a lépések száma legyen.
6. Jegyezzük meg, hogy jártunk már ebben a csúcsban is, hogy többször ne vizsgáljuk meg.
7. Alkalmazzuk ugyanezt az algoritmust most erre a csúcsra is!
8. Ha a 3. pontban igen volt a válasz, akkor is készítsük el a megfelelő éleket, azonban más dolgunk nincs, mert az a csúcs már vizsgálva volt (vagyis vizsgálat alatt van, csak közben a függvény többször meghívta magát, így még nem érte körbe az összes irányon.) Az algoritmus közben ne felejtsük el számolni a csúcsok számát sem, hiszen ez alapján kell a megfelelő méretű memóriaterületet foglalni ahhoz a listához, amiben a csúcsokra mutató pointereket tároltuk.

1.2.2.2. A gráf "kilapítása"

Az algoritmust megvalósító függvény: graf_kilapit()

Miután meg van a gráfunk, ami gyakorlatilag egy kétdimenziós láncolt lista, hiszen négyféle irányban tudunk benne mozogni, szeretnénk egy egydimenziós tömböt is. Ennek nem kell láncolt listának lennie, ennek csak az lesz a szerepe, hogy könnyen és gyorsan végig tudjunk menni az összes csúcson, szomszédságtól függetlenül. Ehhez az előzőhöz hasonló rekurzív algoritmust használunk. Elindulunk a kezdő csúcsból mind a négy irányba, és ha olyan csúcsot találunk, amit még nem raktunk bele a listába, akkor belerakjuk.

1.2.2.3. A Dijkstra-algoritmus

Az algoritmust megvalósító függvény: dijkstra()

Most hogy megvan a gráf és a lista is, alkalmazhatjuk a híres Dijkstra-algoritmust, ami egy gráfban megtalálja két csúcs között a legrövidebb utat. Ennek az algoritmusnak a pszeudókódja röviden:

1. A gráf csúcsait rendezzük két halmazba: vizsgált és nem vizsgált. Először legyen minden a nem vizsgált halmazban.
2. Minden csúcshoz tartozzon egy mező, ami azt adja meg, hogy milyen messze van a kezdő csúcstól. Ezt először állítsuk minden csúcsnál végtelenre (ez esetben 10 000-re állítottam, ezt sem fogja semmilyen szint túllépni), kivéve a kezdő csúcsnál, ott legyen 0.
3. Minden csúcshoz tartozzon egy előző mező is, ami rámutat az előző csúcsra, vagyis hogy ha a kezdő csúcsból ebbe a csúcsba a legrövidebb úton jövünk, akkor melyik csúcsból érkeznénk ide. Ez a mező legyen üres minden csúcsnál kezdésnek.
4. A következő részt addig ismételjük, amíg a meg nem vizsgált csúcsok halmaza nem üres.
5. Válasszuk ki a meg nem vizsgált csúcsok közül azt, amelyiknek a távolsága a kezdő csúcstól a legnagyobb.
6. Ha ez a csúcs a cél, akkor végeztünk, kiléphetünk. (Így ugyan nem lesz meg minden pontnak a távolsága a kezdő ponttól, de erre nincs is szükség ebben a programban. A cél távolsága a kezdő csúcstól pedig ezután már nem változna az algoritmus futása során.) A keresett távolság a cél csúcs távolság mezőjében van, az előző mezőket követve pedig megkapjuk az útvonalat a kezdő csúcsig.
7. Keressük fel a csúcs összes, eddig még nem vizsgált szomszédját! Minden szomszédos csúcs esetén számoljuk ki, hogy milyen hosszú úton jutunk el oda, ha az éppen vizsgált csúcson keresztül megyünk. Ezt úgy kapjuk meg, hogy a vizsgált csúcs távolságához hozzáadjuk a szomszédos csúcsba vezető él súlyát. Ha ez az érték kisebb, mint a szomszédos csúcs távolság mezője, akkor frissítsük azt, és az előző mezőt állítsuk be, hogy a vizsgált csúcsra mutasson. Ezt a lépést ismételjük a vizsgált csúcs összes, eddig még nem vizsgált szomszédjára.
8. A vizsgált csúcsot rakjuk át a nem vizsgált halmazból a vizsgált halmazba.
9. Ugorjunk vissza a 4. lépéshez.

Habár a pszeudókódból kevésbé látszik, nagy szükség volt a sima, egydimenziós listára is az algoritmus közben, hiszen nagyban leegyszerűsítette azt, hogy megmondjuk, hány csúcsot nem vizsgáltunk még (van_nem_latogatott()) és azt is, hogy megmondjuk, melyik a legközelebbi csúcs (legkozelebbi()).

1.2.2.4. Bool mátrix létrehozása

Az algoritmust megvalósító függvény: matrix_letrehoz()

Az algoritmus futása után még fontos, hogy az ideális útvonalat olyan formában kell átadni a függvénynek, ami a képernyőre rajzolja a térképet, azaz minden mezőről egyértelműen el lehessen dönteni, arra ment-e az ideális útvonal. Az algoritmus futása után csak egymásra mutató csúcsokat kapunk, amik a térképen elágazásokat jelentenek, még ezeket is össze kell kötni. Tehát egy olyan kétdimenziós tömböt fogunk létrehozni, aminek egy mezője pontosan akkor igaz, ha arra megy az ideális útvonal. Ehhez a célból indulunk, és követjük a csúcs struktúrájában, a Dijsktra-algoritmus által meghatározott előző mezőt, azaz hogy melyik csúcs felé kell indulnunk. Ahhoz, hogy minden egyes mezőt a két csúcs között megjelöljünk, újra felhasználjuk a gráf építésénél már használt függvényeket. (utak_szama(), kovi_utca())

És máris készen vagyunk az ideális útvonal meghatározásával, amit már egy egyszerű nyomtató függvénnyel meg tudunk mutatni a játékosnak.

1.2.3. A program tesztelése

A program tesztelését főleg kézzel végeztem, azaz minél többször, minél többféle szinttel próbáltam. A memóriakezeléshez viszont használtam a Valgrind nevű programot. Mivel ez a program Windowson nem működik, WSL-ben, a Clion-on keresztül tudtam futtatni, és a program jelenlegi verziójára nem jelez semmilyen hibát.

1.3. A program által olvasott vagy írt fájlok

A program négy fájlt kezel, amiknek a szerkesztése nem szükséges a program futásához, de mivel komoly tartalmi megkötések vannak, ha egyszer belenyúlsz, könnyen hibákba ütközhetsz. Ebben a fejezetben leírom, hogy írj olyan fájlokat, amiket a program elfogad, a következőben pedig abban segítek, hogy megpróbálom megmondani, hol rontottad el, a hibaüzenet alapján.

A program három fő fájlja ".fs" kiterjesztést alkalmaz, amit csak azért használtam, hogy a kevésbé hozzáértők ne merjenek, vagy tudjanak belenyúlni.

1.3.1. "levels.fs"

Ez a fájl tartalmazza a szintek szöveges reprezentációját, és ez talán az egyetlen, amibe egy felhasználónak érdemes lehet belenyúlnia, ha már megunta az alap szinteket. Természetesen ha más szinteket töltesz be, akkor a toplistának nem lesz sok értelme, de a program ezt nem fogja tudni.

A fájl első sorában három egész számnak kell állnia, szóközzel elválasztva.

1. A fájlban szereplő szintek száma. Ha kevesebb szint van a fájlban, hibát fog dobni a program, ha több, akkor a maradékot figyelmen kívül hagyja.
2. A szintek magassága (azaz hány karakter magasak) Fontos, hogy ez az összes szintre vonatkozik! Ha egy szint is nem ilyen magas, hibát fog dobni a program.
3. A szintek hosszúsága (ugyanez igaz erre is, mint a magasságra) Ebből tehát egyértelmű, hogy a játék által egyszerre betöltött szinteknek a mérete meg kell hogy egyezzen.

Ezután jönnek a szintek sorai. Minden szinthez annyi sor és a sorokban annyi karakter tartozik, amennyit az első sorban meghatározott a fájl. Minden karakter vagy "U" (utca) vagy "E" (épület) kell hogy legyen, majd egy sortörés. A szinteket egy üres sor válassza el egymástól. Ha a fájl bármilyen más karaktert tartalmaz, hibát fog dobni a program.

1.3.2. "toplista.fs"

Ez a fájl tartalmazza az aktuális toplistát. Belenyúlni és átírni az eredményedet csalás! Ha valamit átírtál benne, inkább töröld a fájlt, és az első alkalomkor a program el fogja készíteni az üres toplistát. De azért leírom, hogy kell használni...

A fájl első sorában egyetlen számnak kell lennie, ami azt jelenti, hogy hány eredmény van a fájlban. Ez a szám nem lehet nagyobb tíznél, hiszen a program a legjobb 10 eredményt tárolja. Ezután minden egyes sor egy eredményt jelent. Egy sor két szám kezd:

1. Az adott eredmény helyezése, 0-tól indexelve (a holtversenyek miatt fontos)
2. Az adott eredmény pontja Ezeket szóközzel kell elválasztani, majd újabb szóköz után jön a játékos neve, egészen a sor végéig (szóközt is tartalmazhat)

1.3.3. "savegame.fs"

Na ebbe a fájlba aztán végképp csalás belenyúlni, ugyanis itt tárolódik egy lementett játéknak az adatai. A fájl mindösszesen két számot tartalmazhat, szóközzel elválasztva:

1. A teljesített szintek száma (azaz a következő szint indexe)
2. Az eddig elért pontok összege

1.3.4. "errorlog.txt"

Ha a program futása során valamilyen hiba lép fel, akkor a program a képernyőre írás mellett ide is menti a hibakódot és -üzenetet. Az előző hibaüzenet akkor törlődik, amikor következőre szabályosan lépünk ki a programból. Ennek a fájlnak a szerkesztése teljesen felesleges, a program nem olvas belőle. A következő fejezet fog foglalkozni az egyes hibaüzenetek pontos jelentésével.

1.4. Hibakódok és hibaüzenetek

Ha a program futása során valamilyen hiba lép fel, akkor a program a képernyőre írás mellett ide az errorlogs.txt fájlba menti a hibakódot és -üzenetet. Itt le fog írni, mi okozhatja az egyes hibákat, és hogy javítsuk őket. Ha bármi egyéb hiba lépne fel, vagy nem sikerül megjavítani, keress engem bátran: feketesamu(kuakc)gmail(pont)com

1.4.0.1. 10, Érvénytelen toplista fájl!

A program nem tudta megnyitni a "toplista.fs" fájlt se olvasó, se író módban. Lehetséges, hogy nincs jogosultsága ehhez ennek a programnak?

1.4.0.2. 11, Érvénytelen toplista fájl!

A toplista.fs fájl első sora érvénytelen, mindössze egy számot kellene tartalmaznia! Lásd az előző fejezetet a pontos leírásért!

1.4.0.3. 12, Érvénytelen toplista fájl!

A toplista.fs fájl valamelyik sorában nem sikerült két szám beolvasása, valószínűleg valamelyik hiányzik. Lásd az előző fejezetet a pontos leírásért!

1.4.0.4. 13, A játékmentés fájlja érvénytelen!

A program nem tudta megnyitni a "savegame.fs" fájlt író módban. Lehetséges, hogy nincs jogosultsága ehhez ennek a programnak?

1.4.0.5. 14, A szinteket tartalmazó fájl érvénytelen vagy nem létezik!

A program nem tudta olvasó módban megnyitni a "levels.fs" fájlt. Lehetséges, hogy nem létezi? A nhf.exe fájlal egy mappában kell lennie!

1.4.0.6. 15, A szinteket tartalmazó fájl érvénytelen!

A "levels.fs" fájl első sora érvénytelen, nem sikerült a 3 szám beolvasása. Lásd az előző fejezetet a pontos leírásért!

1.4.0.7. 16, A szinteket tartalmazó fájl érvénytelen!

A "levels.fs" fájl valamelyik sorában érvénytelen karakter van. Lásd az előző fejezetet a pontos leírásért!

1.4.0.8. 2, Nincs elég memória!

A program futása során memóriát kért az operációs rendszertől, de az nem volt képes teljesíteni a kívánságát. Mivel a mai számítógépek memóriái elég nagyok az én programomnak szükséges memóriaterületéhez képest, csak úgy tudom elképzelni ezt a hibaüzenetet, hogy valamelyik fájl első sorába egy irdatlan nagy számot írtál.

1.5. Futtatás Linuxon

A programot Windows-on fejlesztettem, de Windows Subsystem for Linux-on tesztelve lett. Mivel a két operációs rendszer máshogy kezeli a fájlokban a sorköz jelet, az alap kód windowson fog jól működni. A Linuxon futtatáshoz (legalábbis nekem csak így működött) ki kell venni a kommentet jelző két "/" jelet a következőre helyekről: szintek.c, 31., 48. és 51. sor. Ezután újra kell fordítani a programkódot, és elvileg működni fog a szintek beolvasása. A toplista fájlnál is hasonló hiba léphet fel.

2. fejezet

Adatszerkezet-mutató

2.1. Adatszerkezetek

Az összes adatszerkezet listája rövid leírásokkal:

3. fejezet

Fájlmutató

3.1. Fájllista

Az összes fájl listája rövid leírásokkal:

4. fejezet

Adatszerkezetek dokumentációja

4.1. Csucs struktúrareferencia

Egy csúcsot reprezentáló struktúra.

`#include <utvonalkereso.h>`

Adatmezők

- int x

  A csúcs x koordinátája.
- int y

  A csúcs y koordinátája.
- int tavolsag

  A csúcs távolsága a kezdő csúcstól.
- bool vizsgalt

  Vizsgáltuk-e már a csúcsot a Dijkstra-algoritmusban?
- int elozo

  Az elek[elozo] adja meg azt az élt, amivel eljuthatunk az előző csúcshoz, ha a legrövidebb úton akarunk visszajutni a kezdő csúcsba.
- El elek [4]

  A csúcsból kiinduló élek tömbje.
- bool cel

  Ez a csúcs a cél-e?
- bool rajt

  Ez a csúcs a rajt-e?

4.1.1. Részletes leírás

Egy csúcsot reprezentáló struktúra.

4.1.2. Adatmezők dokumentációja

4.1.2.1. cel

```
bool cel
```

Ez a csúcs a cél-e?

4.1.2.2. elek

```
El elek[4]
```

A csúcsból kiinduló élek tömbje.

A sorrend az Irany enummal megegyező (fel, balra, le, jobbra)

4.1.2.3. elozo

```
int elozo
```

Az elek[elozo] adja meg azt az élt, amivel eljuthatunk az előző csúcshoz, ha a legrövidebb úton akarunk visszajutni a kezdő csúcsba.

Ha a csúcs nem része a legrövidebb útnak a cél és a rajt között, akkor az algoritmus futása után se biztos, hogy helyes az értéke. Amíg nem tudjuk, melyik az előző csúcs, addig értéke 5. (de így sosem használjuk)

4.1.2.4. rajt

```
bool rajt
```

Ez a csúcs a rajt-e?

4.1.2.5. tavolsag

```
int tavolsag
```

A csúcs távolsága a kezdő csúcstól.

Amennyiben a csúcs nem a cél csúcs, akkor a Dijkstra-algoritmus futása után se biztos, hogy helyes érték, de ezeknek a csúcsoknak a távolsága nem is érdekel bennünket.

4.1.2.6. vizsgalt

```
bool vizsgalt
```

Vizsgáltuk-e már a csúcsot a Dijkstra-algoritmusban?

4.1.2.7. x

```
int x
```

A csúcs x koordinátája.

4.1.2.8. y

```
int y
```

A csúcs y koordinátája.

Ez a dokumentáció a struktúráról a következő fájl alapján készült:

- utvonalkereso.h

4.2. El struktúrareferencia

Egy csúcs egy élét reprezentáló struktúra.

```
#include <utvonalkereso.h>
```

Adatmezők

- int suly

  Az él súlya, azaz a két csúcs távolsága (mezőkben)
- Csucs * csucs

  Pointer a csúcsra, ahova az él mutat.

4.2.1. Részletes leírás

Egy csúcs egy élét reprezentáló struktúra.

4.2.2. Adatmezők dokumentációja

4.2.2.1. csucs

```
Csucs* csucs
```

Pointer a csúcsra, ahova az él mutat.

4.2.2.2. suly

```
int suly
```

Az él súlya, azaz a két csúcs távolsága (mezőkben)

Ez a dokumentáció a struktúráról a következő fájl alapján készült:

- utvonalkereso.h

4.3. Eredmeny struktúrareferencia

Egy eredmény struktúrája, ami a toplistára kerülhet.

```
#include <toplista.h>
```

Adatmezők

- int hely
  Helyezés, 0-tól indexelve.
- int pont
  Pontszám.
- char nev [50]
  Név (maximum 50 karakter)

4.3.1. Részletes leírás

Egy eredmény struktúrája, ami a toplistára kerülhet.

4.3.2. Adatmezők dokumentációja

4.3.2.1. hely

```
int hely
```

Helyezés, 0-tól indexelve.

4.3.2.2. nev

```
char nev[50]
```

Név (maximum 50 karakter)

4.3.2.3. pont

```
int pont
```

Pontszám.

Ez a dokumentáció a struktúráról a következő fájl alapján készült:

- toplista.h

4.4. Pozicio struktúrareferencia

Egy jelenlegi pozíció a cellák mátrixában.

```
#include <szintek.h>
```

Adatmezők

- int x
  X koordináta (0-tól indexelve)
- int y
  Y koordináta (0-tól indexelve)

4.4.1. Részletes leírás

Egy jelenlegi pozíció a cellák mátrixában.

Lehet játékos vagy "képzeletbeli" pont.

4.4.2. Adatmezők dokumentációja

4.4.2.1. x

```
int x
```

X koordináta (0-tól indexelve)

4.4.2.2. y

```
int y
```

Y koordináta (0-tól indexelve)

Ez a dokumentáció a struktúráról a következő fájl alapján készült:

- szintek.h

4.5. Szintek struktúrareferencia

A szintek tömbjét és méreteit tartalmazó struktúra.

```
#include <szintek.h>
```

Adatmezők

- Cella *** terkep
  A háromdimenziós, dinamikusan foglalt tömb pointere.
- int szintszam
  A szintek száma.
- int mag
  A szintek magassága.
- int hossz
  A szintek hossza.
- int aktiv_szint
  A jelenleg aktív szint sorszáma, 0-tól indexelve.
- int iranykonstansok [4][2]
  Azok a konstansok, amit egy pont koordinátáihoz adva azt a megadott irányba mozdítjuk el eggyel.

4.5.1. Részletes leírás

A szintek tömbjét és méreteit tartalmazó struktúra.

Háromdimenziós dinamikus tömb. Ez a dinamikus tömb nem lesz átméretezve a program futása során, maximum felszabadítva majd újra feltöltve.

Utolsó elem: terkep[szintszam][mag][hossz]

Gyakori használat: terkep[aktiv_szint][y][x]

4.5.2. Adatmezők dokumentációja

4.5.2.1. aktiv_szint

```
int aktiv_szint
```

A jelenleg aktív szint sorszáma, 0-tól indexelve.

4.5.2.2. hossz

```
int hossz
```

A szintek hossza.

4.5.2.3. iranykonstansok

```
int iranykonstansok[4][2]
```

Azok a konstansok, amit egy pont koordinátáihoz adva azt a megadott irányba mozdítjuk el eggyel.

Az irányok sorrendje megegyezik az Irany enum-mal. Pl. ha balra akarjuk mozdítani a pontot, akkor: p.x += iranykonstansok[3][0]; (-1) p.y += iranykonstansok[3][1]; (0) Az útvonalkereső függvények közül sokan használják, ezért célszerű a szintek adataival egy struktúrába tenni, így minden függvényben elérhető, ahol szükség van rá.

4.5.2.4. mag

```
int mag
```

A szintek magassága.

4.5.2.5. szintszam

```
int szintszam
```

A szintek száma.

4.5.2.6. terkep

```
Cella*** terkep
```

A háromdimenziós, dinamikusan foglalt tömb pointere.

Ez a dokumentáció a struktúráról a következő fájl alapján készült:

- szintek.h

4.6. Toplista struktúrareferencia

A toplista dinamikus (egydimenziós) tömbbje.

`#include <toplista.h>`

Adatmezők

- int meret

  A tömb mérete, maximum 10.
- Eredmeny * hs

  A dinamikusan foglalt tömb.

4.6.1. Részletes leírás

A toplista dinamikus (egydimenziós) tömbbje.

4.6.2. Adatmezők dokumentációja

4.6.2.1. hs

```
Eredmeny* hs
```

A dinamikusan foglalt tömb.

4.6.2.2. meret

```
int meret
```

A tömb mérete, maximum 10.

Ez a dokumentáció a struktúráról a következő fájl alapján készült:

- toplista.h

5. fejezet

Fájlok dokumentációja

5.1. egyeb.c fájlreferencia

Egyéb függvényeket tartalmazó modul. Ide tartozik többek között a megkezdett játékok mentése és betöltése, a dinamikus tömbök foglalása és szabdítása, valamint a kilépő függvény.

```
#include "egyeb.h"
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include "debugmalloc.h"
```

Függvények

- int szam_beolvas (int min, int max)

  Bekér a felhasználótól egy számot.
- void jatek_ment (int pont, Szintek meretek)

  Fájlba ment egy játékot.
- int jatek_betolt (int *szint, int *pont)

  Betölt egy korábban lementett játékot fájlból, majd törli a fájlból ezt a mentést.
- void segitseg (void)

  Kiírja a segítséget a képernyőre.
- Cella *** cella_tomb_foglal (Szintek meretek)

  Foglal egy dinamikus, háromdimenziós tömböt.
- Csucs *** csucs_tomb_foglal (Szintek meretek)

  Foglal egy dinamikus, kétdimenziós tömböt, aminek minden eleme egy csúcsra mutató pointer lesz, ezeket mind NULL-ra állítja.
- bool ** bool_tomb_foglal (Szintek meretek)

  Foglal egy dinamikus, kétdimenziós tömböt, aminek minden eleme egy boolean érték, ezeket mind hamisra állítja.
- void cella_tomb_szabadit (Szintek meretek)

  Felszabadítja a meretek struktúra dinamukis tömbjét.
- void kilep (int code, char mes[100], Szintek meretek)

  Kilép a programból a megadott hibakóddal és hibaüzenettel, és azt az errorlog.txt fájlba is menti.
- int menu (void)

  Kiírja a menü opcióit, majd bekéri a felhasználó választását a szam_beolvas() függvénnyel.

5.1.1. Részletes leírás

Egyéb függvényeket tartalmazó modul. Ide tartozik többek között a megkezdett játékok mentése és betöltése, a dinamikus tömbök foglalása és szabdítása, valamint a kilépő függvény.

5.1.2. Függvények dokumentációja

5.1.2.1. bool_tomb_foglal()

```
bool** bool_tomb_foglal (
            Szintek meretek )
```

Foglal egy dinamikus, kétdimenziós tömböt, aminek minden eleme egy boolean érték, ezeket mind hamisra állítja.

Paraméterek

meretek	A Szintek méreteit és tömbjét tartalmazó struktúra

Visszatérési érték

A foglalt tömb, amit a hívónak kell felszabadítania

5.1.2.2. cella_tomb_foglal()

```
Cella*** cella_tomb_foglal (
            Szintek meretek )
```

Foglal egy dinamikus, háromdimenziós tömböt.

Ha nincs elég memória, kilép, az eddig foglalt területeket felszabadítva.

Paraméterek

meretek	A Szintek méreteit és tömbjét tartalmazó struktúra

Visszatérési érték

A dinamikus tömb, amit a meghívónak kell felszabadítania a cella_tomb_szabadit() függvénnyel.

5.1.2.3. cella_tomb_szabadit()

```
void cella_tomb_szabadit (
            Szintek meretek )
```

Felszabadítja a meretek struktúra dinamukis tömbjét.

Paraméterek

meretek	A Szintek méreteit és tömbjét tartalmazó struktúra

5.1.2.4. csucs_tomb_foglal()

```
Csucs*** csucs_tomb_foglal (
            Szintek meretek )
```

Foglal egy dinamikus, kétdimenziós tömböt, aminek minden eleme egy csúcsra mutató pointer lesz, ezeket mind NULL-ra állítja.

Paraméterek

meretek	A Szintek méreteit és tömbjét tartalmazó struktúra

Visszatérési érték

A foglalt tömb, amit a hívónak kell felszabadítania

5.1.2.5. jatek_betolt()

```
int jatek_betolt (
            int * szint,
            int * pont )
```

Betölt egy korábban lementett játékot fájlból, majd törli a fájlból ezt a mentést.

Paraméterek

szint	Ebbe a változóba menti a teljesített szintek számát
pont	Ebbe a változóba menti az eddigi pontszámot

Visszatérési érték

0, ha sikeres a betöltés, 1 ha sikertelen (nem volt mentés, vagy érvénytelen)

5.1.2.6. jatek_ment()

```
void jatek_ment (
            int pont,
            Szintek meretek )
```

Fájlba ment egy játékot.

Paraméterek

pont	Eddig szerzett pontok
meretek	A Szintek méreteit és tömbjét tartalmazó struktúra

5.1.2.7. kilep()

```
void kilep (
            int code,
            char mes[100],
            Szintek meretek )
```

Kilép a programból a megadott hibakóddal és hibaüzenettel, és azt az errorlog.txt fájlba is menti.

Felszabadítja a dinamikus tömböt, hacsak annak pointere nem NULL. Ha úgy akarjuk meghívni a függvényt, hogy a meretek.terkep már fel lett szabadítva, vagy még nincs lefoglalva, akkor a meretek paraméternek adjunk (Szintek) {NULL}-t! Mivel 2-es (memóriahiány) kóddal sokszor van meghívva a függvény, ennek a hibaüzenete itt van megadva, ilyenkor mes lehet NULL. Figyelem! Ez a függvény egyes IDE-kben sok figyelmeztetést okozhat. Ha ez a függvény valahol meg lett hívva, akkor ott biztosan véget ér a program futása, hiszen ennek a függvénynek exit() a vége. Így ezen függvény meghívása után fellépő figyelmeztetések nem okozhatnak problémát, hiszen oda már nem is fog eljutni a program.

Paraméterek

code	A hibakód
mes	A hibaüzenet, maximum 100 karakter.
meretek	A Szintek méreteit és tömbjét tartalmazó struktúra

5.1.2.8. menu()

```
int menu (
            void  )
```

Kiírja a menü opcióit, majd bekéri a felhasználó választását a szam_beolvas() függvénnyel.

Visszatérési érték

a felhasználó választása az opciók közül

5.1.2.9. segitseg()

```
void segitseg (
            void  )
```

Kiírja a segítséget a képernyőre.

5.1.2.10. szam_beolvas()

```
int szam_beolvas (
            int min,
            int max )
```

Bekér a felhasználótól egy számot.

Ha az nincs [min, max]-ban, akkor újra kér. Egyébként visszaadja a számot. Az InfoC-s tutorial alapján.

Paraméterek

min	Várt szám alsó korlátja
max	Várt szám felső korlátja

Visszatérési érték

Felhasználó választása, ami garantáltan [min, max]-ban van

5.2. errorlog.txt fájlreferencia

5.3. main.c fájlreferencia

A program fő fájlja, ami a főmeüt kezeli és meghívja a megfelelő modul megfelelő függvényét.

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include "econio.h"
#include "szintek.h"
#include "egyeb.h"
#include "toplista.h"
#include "debugmalloc.h"
```

Függvények

- int main ()

  Beállítja a konzol kódolását, kezeli a főmenüt és meghívja a megfelelő függvényeket.

5.3.1. Részletes leírás

A program fő fájlja, ami a főmeüt kezeli és meghívja a megfelelő modul megfelelő függvényét.

5.3.2. Függvények dokumentációja

5.3.2.1. main()

```
int main ( )
```

Beállítja a konzol kódolását, kezeli a főmenüt és meghívja a megfelelő függvényeket.

Visszatérési érték

Kilépési kód, de valójában soha nem ezzel lép ki a program, hanem a kilep() függvénynek megadott kóddal.

5.4. szintek.c fájlreferencia

A szintek betöltését és kezelését leíró függvényeket tartlmazó modul.

```
#include "szintek.h"
#include "egyeb.h"
#include "toplista.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "econio.h"
#include "debugmalloc.h"
```

Függvények

- Szintek szintek_betolt (void)

  Betölti a "levels.fs" fájlból a szinteket egy dinamikusan foglalt, háromdimenziós tömbbe.
- void jatek_indul (int szint, int pont)

  Elindítja a játékot a megadott szintről és a megadott kezdőponttal.
- static int kovi_szint (Szintek meretek, int ∗idealis)

  A paraméterben megadot szint indítása.
- static void palya_nyomtat (Pozicio p, Szintek meretek, int time)

  Konzolba nyomtatja az aktuális pályát.
- static void palya_vegso_nyomtat (Pozicio p, Szintek meretek, int time, bool ∗∗idealis)

  A palya_nyomtat() függvényhez hasonlóan kinyomtatja a képernyőre a pályát, de nem csak a játkos útvonalával, hanem az ideális útvonallal is.
- static bool checkif_building (Pozicio p, Szintek meretek)

  Megnézi, hogy a játékos éppen épület mezőn áll-e, vagy a pályán kívül van-e.
- static bool checkif_finish (Pozicio p, Szintek meretek)

  Megnézi, hogy a játékos éppen a cél mezőn áll-e (bal alsó sarok)

5.4.1. Részletes leírás

A szintek betöltését és kezelését leíró függvényeket tartlmazó modul.

5.4.2. Függvények dokumentációja

5.4.2.1. checkif_building()

```
static bool checkif_building (
            Pozicio p,
            Szintek meretek )  [static]
```

Megnézi, hogy a játékos éppen épület mezőn áll-e, vagy a pályán kívül van-e.

Akkor ad vissza igazat, ha a játékos olyan mezőn áll, ahol nem állhatna.

Paraméterek

p	Játékos jelenlegi pozíciója
meretek	A Szintek méreteit és tömbjét tartalmazó struktúra

Visszatérési érték

Igaz, ha épület mezőn vagy a pályán kívül áll a játékos, különben hamis

5.4.2.2. checkif_finish()

```
static bool checkif_finish (
            Pozicio p,
            Szintek meretek )  [static]
```

Megnézi, hogy a játékos éppen a cél mezőn áll-e (bal alsó sarok)

Paraméterek

p	Játékos jelenlegi pozíciója
meretek	A Szintek méreteit és tömbjét tartalmazó struktúra

Visszatérési érték

Igaz, ha a játékos a célban van, egyébként hamis.

5.4.2.3. jatek_indul()

```
void jatek_indul (
            int szint,
            int pont )
```

Elindítja a játékot a megadott szintről és a megadott kezdőponttal.

Végigviszi a játékost az összes szinten a kovi_szint() függvénnyel, ha a játékos nem lép ki közben. Miután az összes szinttel végzett, betölti a toplistát és új eredményt vesz fel az uj_eredmeny() függvénnyel. Végül kiírja a toplistát a képernyőre, és felszabadítja az elfoglalt memóriaterületeket a toplista_nyomtat() függvénnyel.

Paraméterek

szint	Erről a szintről fog indulni a játék, 0-tól indexelve
pont	Ennyi ponttal kezdi a játékos a játékot. Mindkettő 0, ha új játékot kezd a felhasználó.

5.4.2.4. kovi_szint()

```
static int kovi_szint (
            Szintek meretek,
            int * idealis )  [static]
```

A paraméterben megadot szint indítása.

Folyamatosan nyomtatja a térképet, számolja az eltelt időt és a lépéseket, valamit mozgatja a játékos karakterét. Egészen addig, amíg az idő lejár vagy a játékos eléri a célt. Ekkor meghatározza a legrövidebb útvonalat a legrovidebb() fügvénnyel, és meg is mutatja a játékosnak. A szint futása közben ESC nyomására kilép a programból a kilep() fuggvénnyel.

Paraméterek

meretek	A Szintek méreteit és tömbjét tartalmazó struktúra
idealis	Pointer egy integerre, ahova menti a függvény az adott szint ideális útvonalának hosszát.

Visszatérési érték

A szerzett pontok száma

5.4.2.5. palya_nyomtat()

```
static void palya_nyomtat (
            Pozicio p,
            Szintek meretek,
            int time )  [static]
```

Konzolba nyomtatja az aktuális pályát.

Paraméterek

p	Pozicio struktúr, a játékos jelenlegi x és y pozíciója.
meretek	A Szintek méreteit és tömbjét tartalmazó struktúra
time	Hátralevő idő másodpercben

5.4.2.6. palya_vegso_nyomtat()

```
static void palya_vegso_nyomtat (
            Pozicio p,
            Szintek meretek,
            int time,
            bool ** idealis )  [static]
```

A palya_nyomtat() függvényhez hasonlóan kinyomtatja a képernyőre a pályát, de nem csak a játkos útvonalával, hanem az ideális útvonallal is.

Paraméterek

p	A játékos pozíciója
meretek	A Szintek méreteit és tömbjét tartalmazó struktúra
time	Hátralévő idő másodpercben (ilyenkor ez már fix, nem változik)
idealis	Kétdimenziós bool tömb, aminek minden mezője pontosan akkor igaz, ha az ideális útvonal átmegy azon a mezőn.

5.4.2.7. szintek_betolt()

```
Szintek szintek_betolt (
            void  )
```

Betölti a "levels.fs" fájlból a szinteket egy dinamikusan foglalt, háromdimenziós tömbbe.

Ha a fájlt nem tudja megnyitni, vagy érvénytelen, akkor kilép a programból a kilep() függvénnyel.

Visszatérési érték

A Szintek méreteit és tömbjét tartalmazó struktúra

5.5. szintek.h fájlreferencia

```
#include <stdio.h>
#include <stdbool.h>
```

Adatszerkezetek

- struct Pozicio

  Egy jelenlegi pozíció a cellák mátrixában.
- struct Szintek

  A szintek tömbjét és méreteit tartalmazó struktúra.

Enumerációk

- enum Cella { epulet, utca, u_jart }

  Egy cella a pályán.

5.5.1. Enumerációk dokumentációja

5.5.1.1. Cella

`enum Cella`

Egy cella a pályán.

A képernyőn 3 karakter hosszú és 1 karakter magas.

Enumeráció-értékek

epulet	Épület.
utca	Utca, amin nem járt még a játékos.
u_jart	Utca, amin járt már a játékos.

5.6. toplista.c fájlreferencia

A toplista betöltését és szerkesztését leíró függvényeket tartlmazó modul.

```
#include "toplista.h"
#include "egyeb.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "debugmalloc.h"
```

Függvények

- void toplista_betolt (Toplista *tl, Szintek meretek)

Fájlból betölti a lementett toplistát.

- void uj_eredmeny (Toplista *tl, int pont, Szintek meretek)

  Megállapítja egy új eredményről, hogy az felkerül-e a toplistára.
- static void eredmeny_felvesz (Toplista *tl, Eredmeny new, Szintek meretek)

  Egy eredményt berak a dinamikus tömbbe.
- void toplista_nyomtat (Toplista tl)

  Kinyomtatja a képernyőre az aktuális toplistát, majd felszabadítja a dinamikus tömböt.
- static void toplista_fajlba (Toplista tl)

  Fájlba írja az toplistát.

5.6.1. Részletes leírás

A toplista betöltését és szerkesztését leíró függvényeket tartlmazó modul.

5.6.2. Függvények dokumentációja

5.6.2.1. eredmeny_felvesz()

```
static void eredmeny_felvesz (
            Toplista * tl,
            Eredmeny new,
            Szintek meretek )  [static]
```

Egy eredményt berak a dinamikus tömbbe.

Ha szükséges, meg is nyújtja a tömböt (egyébként az utolsó eredményt eldobja). Habár a dinamius tömb megnyújtása hosszú művelet is lehetne, itt maximum 9 elemű tömböt kell másolni, ami nem probléma a mai gépeknek.

Paraméterek

tl	Dinamikus tömb
new	Az eredmény, amit berak a tömbbe
meretek	A Szintek méreteit és tömbjét tartalmazó struktúra

5.6.2.2. toplista_betolt()

```
void toplista_betolt (
            Toplista * tl,
            Szintek meretek )
```

Fájlból betölti a lementett toplistát.

Ha nem találja, akkor létrehozza. Ha nem tudja megynitni a fájlt, vagy az érvénytelen, kilép a kilep() függvénnyel.

Paraméterek

tl	Dinamikus tömb, amibe betölti az eredményeket.
meretek	A Szintek méreteit és tömbjét tartalmazó struktúra

5.6.2.3. toplista_fajlba()

```
static void toplista_fajlba (
            Toplista tl )  [static]
```

Fájlba írja az toplistát.

Paraméterek

tl	A toplista dinamikus tömbje

5.6.2.4. toplista_nyomtat()

```
void toplista_nyomtat (
            Toplista tl )
```

Kinyomtatja a képernyőre az aktuális toplistát, majd felszabadítja a dinamikus tömböt.

A toplista nyomtatása után már sosincs szükségünk a toplistára, ezért innen is fel lehet szabadítani.

Paraméterek

tl	A toplista dinamikus tömbje

5.6.2.5. uj_eredmeny()

```
void uj_eredmeny (
            Toplista * tl,
            int pont,
            Szintek meretek )
```

Megállapítja egy új eredményről, hogy az felkerül-e a toplistára.

Ha igen, bekéri a felhasználótól a nevét, és meghívja az eredmeny_felvesz() függvényt, majd az új toplistát fájlba írja a toplista_fajlba() függvénnyel.

Paraméterek

tl	Dinamikus tömb
pont	Az új eredmény pontszáma
meretek	A Szintek méreteit és tömbjét tartalmazó struktúra

5.7. toplista.h fájlreferencia

```
#include "szintek.h"
```

Adatszerkezetek

- struct Eredmeny

  Egy eredmény struktúrája, ami a toplistára kerülhet.
- struct Toplista

  A toplista dinamikus (egydimenziós) tömbbje.

5.8. utvonalkereso.c fájlreferencia

Ez a modul tartalmazza a legrövidebb útvonal megtalálásához szükséges függvényeket.

```
#include <stdlib.h>
#include "utvonalkereso.h"
#include "egyeb.h"
#include "debugmalloc.h"
```

Függvények

- bool ∗∗ legrovidebb (Szintek meretek, int ∗idealis)

  Megkeresi a legrövidebb útat az aktív szinten a rajt és a cél között (a fájl többi függvényének segítségével).
- static void graf_epit (Csucs ∗cs, Csucs ∗∗∗vizsgalt, Szintek meretek, int ∗meret)

  Rekurzív függvény, ami felépíti a gráf adatstruktúrát.
- static void graf_kilapit (Csucs ∗cs, Csucs ∗∗∗lista, int ∗n, bool ∗∗vizsgalt, int hossz)

  Rekurzív függvény, ami csinál egy egydimneziós tömböt a gráfhoz, aminek minden eleme egy csúcsra mutat.
- static void dijkstra (Csucs ∗∗lista, int meret)

  Függvény, ami megvalósítja a Dijkstra algoritmust a rajt és cél között.
- static void laposgraf_szabadit (Csucs ∗∗∗lista, int meret)

  Felszabadítja a gráf minden csúcsát, majd a tömböt amiben tároltuk őket.
- static bool utca_teszt (Pozicio p, Szintek meretek)

  Függvény, ami megvizsálja hogy az adott koordináta a pályán belül van-e és a mező utca-e.
- static int utak_szama (Pozicio p, Szintek meretek)

  Függvény, ami meghatározza, hogy az adott koordináták által meghatározott mezőnek hány szomszédja utca.
- static void kovi_utca (Pozicio ∗regi, Pozicio ∗uj, Szintek meretek)

Ha egy mezőnek két szomszédja utca, akkor ez a függvény meghatározza, hogy melyik irányba kell tovább menni, hogy az utcán maradjunk.

- static Irany irany_hataroz (Pozicio p)

  Függvény, ami meghatározza, hogy az adott x-y elmozdulással melyik irányba haladunk.
- static void csucs_init (Csucs ∗cs, Pozicio p, Szintek meretek)

  Kezdeti állapotba állít egy csúcsot.
- static bool van_nem_latogatott (Csucs ∗∗lista, int meret)

  Függvény, ami meghatározza, hogy van-e még meg nem látogatott csúcs a gráfban.
- static int legkozelebbi (Csucs ∗∗lista, int meret)

  Függvény, ami meghatározza azt a csúcsot, ami még nem volt vizsgálva, és a távolsága a legkisebb a kezdő csúcstól.
- static bool ∗∗ matrix_letrehoz (Csucs ∗∗lista, int meret, Szintek meretek, int ∗idealis)

  Létrehoz egy dinamikusan foglalt kétdimenziós tömböt, aminek mérete megegyezik a pálya méretével, és mezői pontosan akkor igazak, ha a gráfban az ideális útvonal a rajt és a cél között átmegy az adott mezőn.

5.8.1. Részletes leírás

Ez a modul tartalmazza a legrövidebb útvonal megtalálásához szükséges függvényeket.

Többek között olyan függvények, amik gráfot építenek, alkalmazzák a Dijkstra-algoritmust, és az eredményt egy kétdimenziós tömbbé alakítják, hogy könnyebben lehessen a képernyőre nyomtatni. A főbb függvények algoritmusairól az első fejezetben részletesen is írok.

5.8.2. Függvények dokumentációja

5.8.2.1. csucs_init()

```
static void csucs_init (
            Csucs * cs,
            Pozicio p,
            Szintek meretek )  [static]
```

Kezdeti állapotba állít egy csúcsot.

Pointereit lenullázza, megvizsgálja hogy a rajt vagy esetleg a cél-e. Ha a csúcs nem a rajt, akkor a távolságot egy olyan nagy értékre állítja, amilyen távolságok a gráfban biztos nem lesznek. (A Dijkstra-algoritmus szerint végtelenre kellene állítani, de ilyen lehetőség nincs C-ben)

Paraméterek

cs	A beállítani kíívánt csúcs pointere
p	A csúcs x és y koordinátája Pozíció struktúrában
meretek	Betöltött szintek méretei és tömbje

5.8.2.2. dijkstra()

```
static void dijkstra (
            Csucs ** lista,
            int meret )  [static]
```

Függvény, ami megvalósítja a Dijkstra algoritmust a rajt és cél között.

(Azaz meghatározza a legrövidebb útat a két pont között.)

Paraméterek

lista	Tömb, aminek minden elem a gráf egyik csúcsára mutat.
meret	Csúcsok száma a gráfba, azaz a tömb mérete.

5.8.2.3. graf_epit()

```
static void graf_epit (
            Csucs * cs,
            Csucs *** vizsgalt,
            Szintek meretek,
            int * meret )  [static]
```

Rekurzív függvény, ami felépíti a gráf adatstruktúrát.

Az elágazások lesznek a gráf csúcsai, az őket összekötő utcák pedig az élek. Egy futása egy adott csúcsot köt össze szomszédjaival, majd mindegyik szomszédjára meghívja magát. Közben számolja, hogy összesen hány csúcsot talál.

Paraméterek

cs	Annak a csúcsnak a pointere, amelyiket össze szeretnénk kötni a szomszédaival.
vizsgalt	Kétdimenziós tömb (méretei megegyeznek a szint méreteivel), aminek minden mezője NULL, ha ahhoz a mezőhöz még nem csináltunk csúcsot. Amikor egy új csúcsot készítünk, ebben a tömbben a megfelelő mezőt a csúcs pointerére állítjuk.
meretek	Betöltött szintek méretei és tömbje Az irányok sorrendje megegyezik enum Irany-nyal.
meret	Ebbe a változóba menti a csúcsok számát.

5.8.2.4. graf_kilapit()

```
static void graf_kilapit (
            Csucs * cs,
            Csucs *** lista,
            int * n,
            bool ** vizsgalt,
            int hossz )  [static]
```

Rekurzív függvény, ami csinál egy egydimneziós tömböt a gráfhoz, aminek minden eleme egy csúcsra mutat.

Ez később segíteni fogja az eligazodást a gráfban.

Paraméterek

cs	Kiinduló csúcs
lista	Az épülő tömb
n	Index, amit a rekurzív függvények mindegyike elér. Számlálja, hogy eddig hány csúcsot tettünk a tömbbe.
vizsgalt	Kétdimenziós tömb, aminek minden mezője pontosan akkor igaz, ha az ahhoz a mezőhöz tartozó csúcs már vizsgálva volt.
hossz	Csúcsok száma a gráfban

5.8.2.5. irany_hataroz()

```
static Irany irany_hataroz (
            Pozicio p )  [static]
```

Függvény, ami meghatározza, hogy az adott x-y elmozdulással melyik irányba haladunk.

A két koordináta közül pontosan egynek 0-nak, egynek pedig 1-nek vagy -1-nek kell lennie. Más x-y párra nem feltétlenül ad helyes eredményt.

Paraméterek

p	Az x és y koordináta Pozíció struktúrában

Visszatérési érték

Az irány enum-ja

5.8.2.6. kovi_utca()

```
static void kovi_utca (
            Pozicio * regi,
            Pozicio * uj,
            Szintek meretek )  [static]
```

Ha egy mezőnek két szomszédja utca, akkor ez a függvény meghatározza, hogy melyik irányba kell tovább menni, hogy az utcán maradjunk.

Paraméterek

regi	Az előző pozíció x és y koordinátája Pozíció struktúrában (erre nem szeretnénk visszamenni)
uj	Meghíváskor a jelenlegi pozíció koordinátái, ami frissítve lesz a következő koordinátákra, úgy hogy az utca maradjon, de ne abba az irányba lépjünk, ahonnan jöttünk.
meretek	Betöltött szintek méretei és tömbje Az irányok sorrendje megegyezik enum Irany-nyal.

5.8.2.7. laposgraf_szabadit()

```
static void laposgraf_szabadit (
            Csucs *** lista,
            int meret )  [static]
```

Felszabadítja a gráf minden csúcsát, majd a tömböt amiben tároltuk őket.

Paraméterek

lista	A csúcsok tömbje
meret	A tömb mérete

5.8.2.8. legkozelebbi()

```
static int legkozelebbi (
            Csucs ** lista,
            int meret )  [static]
```

Függvény, ami meghatározza azt a csúcsot, ami még nem volt vizsgálva, és a távolsága a legkisebb a kezdő csúcstól.

Paraméterek

lista	A gráf csúcsait tartalmazó tömb.
meret	A tömb mérete, azaz csúcsok száma

Visszatérési érték

A megfelelő csúcs indexe a tömbben

5.8.2.9. legrovidebb()

```
bool** legrovidebb (
            Szintek meretek,
            int * idealis )
```

Megkeresi a legrövidebb útat az aktív szinten a rajt és a cél között (a fájl többi függvényének segítségével).

Minden ilyen segédfüggvényhez elkészíti a szükséges segédtömböket, majd azokat felszabadítja.

Paraméterek

meretek	Betöltött szintek méretei és tömbje
idealis	Ebbe a változó menti a függvény az ideális útvonal hosszát Az irányok sorrendje megegyezik enum Irany-nyal.

Visszatérési érték

Egy olyan kétdimenziós tömb, aminek azon mezői igazak, amin átmegy az ideális útvonal, a többi hamis.

5.8.2.10. matrix_letrehoz()

```
static bool** matrix_letrehoz (
            Csucs ** lista,
            int meret,
            Szintek meretek,
            int * idealis )  [static]
```

Létrehoz egy dinamikusan foglalt kétdimenziós tömböt, aminek mérete megegyezik a pálya méretével, és mezői pontosan akkor igazak, ha a gráfban az ideális útvonal a rajt és a cél között átmegy az adott mezőn.

Paraméterek

lista	A gráf csúcsait tartalmazó tömb.
meret	A tömb mérete, azaz csúcsok száma
meretek	Betöltött szintek méretei és tömbje
idealis	Pointer, ahova a függvény elmenti az ideális útvonal hosszát

Visszatérési érték

A dinamikusan foglalt kétdimenziós bool tömb, amit a hívónak fel kell szabadítania.

5.8.2.11. utak_szama()

```
static int utak_szama (
            Pozicio p,
            Szintek meretek )  [static]
```

Függvény, ami meghatározza, hogy az adott koordináták által meghatározott mezőnek hány szomszédja utca.

(Akkor fogunk egy mezőt csúcsnak tekinteni, ha legalább 3 utca szomszédja van). Mivel a rajtot és a célt is csúcsnak szeretnénk tekinteni (hiszen közöttük keressük a legrövidebb utat), náluk automatikusan 3-at fog visszaadni a függvény.

Paraméterek

p	A pont x és y koordinátja Pozíció struktúrában
meretek	Betöltött szintek méretei és tömbje

Visszatérési érték

Szomszédos utca mezők száma

5.8.2.12. utca_teszt()

```
static bool utca_teszt (
            Pozicio p,
            Szintek meretek )  [static]
```

Függvény, ami megvizsálja hogy az adott koordináta a pályán belül van-e és a mező utca-e.

Paraméterek

p	A pont x és y koordinátja Pozíció struktúrában
meretek	Betöltött szintek méretei és tömbje

Visszatérési érték

Igaz, ha a koordináták által mutatott mező utca

5.8.2.13. van_nem_latogatott()

```
static bool van_nem_latogatott (
            Csucs ** lista,
            int meret )  [static]
```

Függvény, ami meghatározza, hogy van-e még meg nem látogatott csúcs a gráfban.

Paraméterek

lista	A gráf csúcsait tartalmazó tömb.
meret	A tömb mérete, azaz csúcsok száma

Visszatérési érték

Igaz, ha van még meg nem látogatott csúcs, egyébként hamis

5.9. utvonalkereso.h fájlreferencia

```
#include <stdbool.h>
#include "szintek.h"
```

Adatszerkezetek

- struct El

  Egy csúcs egy élét reprezentáló struktúra.
- struct Csucs

  Egy csúcsot reprezentáló struktúra.

Típusdefiníciók

- typedef struct Csucs Csucs

Enumerációk

- enum Irany {
  fel = 0, jobb = 1, le = 2, bal = 3,
  egyikse = 4 }

  A négy lehetséges irány a játékban.

5.9.1. Típusdefiníciók dokumentációja

5.9.1.1. Csucs

```
typedef struct Csucs Csucs
```

5.9.2. Enumerációk dokumentációja

5.9.2.1. Irany

```
enum Irany
```

A négy lehetséges irány a játékban.

Minden függvény, ami egy csúcs éleit veszi sorba, ebben a sorrendben halad.

Enumeráció-értékek

fel	
jobb	
le	
bal	
egyikse	

Tárgymutató