



Eötvös Loránd Tudományegyetem
Informatikai Kar
Algoritmusok és Alkalmazásai Tanszék

Rendezési algoritmusok szemléltetése

Veszprémi Anna
mestertanár

Márföldi Péter Bence
programtervező informatikus BSc

Budapest, 2015

Tartalomjegyzék

1. Bevezetés	1
1.1. A feladat és annak értelmezése	1
1.2. Alkalmazott technológiák	1
1.2.1. Java	2
1.2.2. JavaFX	2
1.2.3. JUnit	2
2. Felhasználói dokumentáció	4
2.1. Rendszerkövetelmények	4
2.1.1. Minimális rendszerkövetelmények	4
2.1.2. Ajánlott rendszerkövetelmények	5
2.1.3. Telepítés és eltávolítás	5
Telepítés natív telepítővel	5
Hagyományos telepítés	6
Eltávolítás	6
2.2. Felhasználói felület bemutatása	7
2.2.1. Főmenü	7
Eszköztár	7
Központi panel	8
2.2.2. Bemenet megadása panel	8
Manuális	8
Generálás	8
Fájlból beolvasás	9
2.2.3. Főpanel	10
Eszköztár	11
Panelválasztó	11
Panel	11
2.2.4. Megfigyelés panel	11
Algoritmus lista	12
Állapotjelző táblázat	12
Gombok	12

TARTALOMJEGYZÉK

Állapotjelző felület	13
2.2.5. Összehasonlítás panel	14
Elemzés táblázat	14
Elemzés diagram	14
2.3. A vizsgált algoritmusok	15
2.3.1. Buborékrendezés	15
2.3.2. Beszúró rendezés	16
2.3.3. Shell rendezés	18
2.3.4. Gyorsrendezés	19
2.3.5. Kupacrendezés	21
2.3.6. Versenyrendezés	22
2.3.7. Radix "előre"	24
2.3.8. Radix "vissza"	26
3. Fejlesztői dokumentáció	28
3.1. Tervezés	28
3.1.1. Alapelvek	28
3.1.2. Használt fejlesztőeszközök	29
3.1.3. Felhasználói felület	29
Képernyőtervezek	29
Főmenü	29
Manuális bemenet	29
Bemenet generálása	30
Fájl beolvasása	30
Főpanelek	30
Megfigyelés panel	30
Összehasonlítás panel	30
Felületek közötti navigáció lehetőségek	30
3.1.4. Használati esetek	31
3.1.5. Csomagszerkezet	32
Modell csomag	33
algorithm alcsomag	34
algorithm.raw alcsomag	34
algorithm.raw.test alcsomag	34
info alcsomag	34
Megjelenítő csomag	34
graph alcsomag	35
Vezérlő csomag	35
3.1.6. Osztályszerkezet	35

TARTALOMJEGYZÉK

alapcsomag osztálya	36
controller csomag osztályai	36
algorithm csomag osztályai	37
alogirtmh.raw csomag osztályai	38
info csomag osztályai	39
3.2. Megvalósítás	39
3.2.1. Az eredeti terv módosítása	39
Szálkezelés	40
A JavaFX kibővítése	40
3.2.2. A megvalósítás menete	40
3.2.3. FXML állományok	41
Néhány fontosabb FXML állomány	42
BaseController	42
OverviewLayout	42
3.2.4. Nem forrásfájl állományok	42
3.2.5. Osztályok leírása	42
Modell réteg osztályai	42
SortingAlgorithm absztrakt osztály	43
ChartAlgorithm absztrakt osztály	43
GraphAlgorithm absztrakt osztály	43
RadixAlgorithm absztrakt osztály	44
SortingAlgorithmFactory osztály	44
CounterData osztály	45
RecursiveParameter osztály	45
BubbleSort osztály	46
InsertionSort osztály	46
ShellSort osztály	47
QuickSort osztály	47
HeapSort osztály	48
TournamentSort osztály	49
ForwardRadix osztály	49
BackwardRadix osztály	50
BenchmarkData osztály	50
SortingThread absztrakt osztály	51
BubbleSortThread osztály	51
InsertionSortThread osztály	52
ShellSortThread osztály	52
QuickSortThead osztály	52
HeapSortThread osztály	52

TARTALOMJEGYZÉK

TournamentSortThread osztály	53
InfoFlyWeight osztály	53
InfoFlyWeightFactory osztály	53
Kontroller réteg osztályai	54
MainApplication osztály	54
MenuController osztály	54
ManualInputController osztály	54
RandomInputController osztály	55
FileInputController osztály	56
Megjelenítési réteg osztályai	56
Vertex osztály	56
Edge osztály	57
Graph osztály	57
3.2.6. Tesztelés	57
4. Irodalomjegyzék	58

1. fejezet

Bevezetés

Az eddigi egyetemen töltött éveim során az **Algoritmusok és adatszerkezetek** kurzus foglalkozott mélyrehatóan a rendezési algoritmusokkal. Számonomra a tananyagból talán ez a témakör volt a legnehezebben elsajátítható. Ez sarkallt arra, hogy a szakdolgozatom témáját adják a rendezési algoritmusok, szerettem volna biztos tudással rendelkezni ezen a területen.

Az bizonyos, hogy minden informatikus - beleértve a leendőeket is - tanulmányaik kezdetén találkoztak ezen eljárásokkal. Nagyszerű terület arra, hogy megérthessük mi a műveletigény, hogy mi számít igazán sok adatnak, vagy, hogy mit értünk egy algoritmus stabilitásán.

1.1. A feladat és annak értelmezése

A feladat egy oktatóprogram létrehozása, melynek segítségével az érdeklődő megértheti néhány rendezési algoritmus működését egy egyszerű, letisztult felületen. Lehetőséget kell adni az algoritmusok lépésekénti vizsgálatára, továbbá a műveletek számának összehasonlítására. A felhasználó több módon megadhatja a rendezendő pozitív egész számokat:

- begépeléssel
- generálással
- fájlból betöltéssel

1.2. Alkalmazott technológiák

A Következőkben röviden összefoglaljuk a *Java*[8], *JavaFX*[9] és *JUnit*[10] jellegzetességeit.

1.2.1. Java

A *Java* egy általános célú, objektumorientált programozási nyelv, melyet 2009-ig a *Sun Microsystems* fejlesztett, ezt követően pedig az *Oracle*. A szakdolgozatban használt 1.8-as verziót már az *Oracle* adta ki 2014-ben. A *Java* nyelv a szintaxisát a *C* és *C++* nyelvektől örökölte, azonban utóbbitól eltérően egyszerű objektummóddel rendelkezik.

A *Java* platformra készült programok túlnyomó többsége asztali alkalmazás. Ma-napság egyre több helyen találkozhatunk a *Java* nyelven írt programokkal, például mobil eszközökön, banki rendszereknél vagy akár egy szórakoztató elektronikai eszközön. Nagy előnye, hogy sok nyelvvel ellentétben platformfüggetlen, azaz egy adott platformról egy program minimális változtatással átültethető egy másik platformra.

A *Java* legfontosabb része a *Java virtuális gép (JVM)*. A *JVM*-et sokféle berendezés és szoftvercsomag tartalmazza, így a nyelv egyaránt platformként és középszintként is működik. Összefoglalva a *Java* program három fontos szerepet tölt be:

- programozási nyelv
- köztes réteg (middleware)
- platform

1.2.2. JavaFX

Olyan szoftverplatform, amelynek célja, hogy gazdag internetes alkalmazást lehessen készíteni és futtatni eszközök széles skáláján. Eredetileg a *Swing* könyvtárat váltotta fel, azonban jelenleg mindenki része a *Java SE*-nek.

A 2.0-ás verzióig a fejlesztők egy külön nyelvet használtak, amelyet *JavaFX Script*-nek neveznek. Azonban mivel ez szintén *Java* bájtkódot generál a későbbieken megadatott a lehetőség, hogy a programozók *Java* kódot használjanak helyette. A *JavaFX* egyik legnagyobb előnye, hogy egy egyszerű *XML* struktúrában leírhatók a program grafikus felületének összetevői, melyhez ezt követően elegendő az egyes interakciókhöz tartozó funkciókat implementálni.

Az elterjedtebb operációs rendszerek mindegyikét támogatja. Ahogyan előnye, úgy hátránya is a *Swing*-hez képest az, hogy jelenleg is folyik a fejlesztése, ezért olykor csak hosszas utánjárást követően sikerül megoldást találni egy-egy problémára.

1.2.3. JUnit

Egy egységeszt keretrendszer a *Java* programozási nyelvhez. Az egységesztek karbantartására, és futtatására kínál szolgáltatást. Gyakran a verzió kiadási folyamat részeként szokták beépíteni, azaz egy kiadás akkor hibátlan, ha ezen tesztek

1.2. ALKALMAZOTT TECHNOLÓGIÁK

mindegyike hibátlanul lefut. Egy 2013-as felmérésben[12] tízezer Java technológiát használó *GitHub* projektet vizsgáltak. A projektek csaknem harmadánál használták a *JUnit*-ot, ezzel az egyik leggyakrabban használt függvénykönyvtár volt a felmérés során.

2. fejezet

Felhasználói dokumentáció

2.1. Rendszerkövetelmények

Az elkövetkezőkben ismertetésre kerülnek a minimális és az ajánlott rendszerkövetelmények.

2.1.1. Minimális rendszerkövetelmények

Mivel a program *Java* nyelven íródott, ezért elengedhetetlen, hogy a felhasználó számítógépen lehetőség legyen *Java* alkalmazások futtatására. Az alábbi operációs rendszereken érhető el a *Java Runtime Environment 8u45*-ös verziója:

- *Windows 8*
- *Windows 7*
- *Windows Vista SP2*
- *Windows Server 2008 R2 SP1 (64-bit)*
- *Windows Server 2012 (64-bit)*
- *Mac OS X 10.8.3 vagy újabb*
- *Suse Linux Enterprise Server 10 SP2+, 11.x*
- *Ubuntu Linux 12.04 vagy újabb*
- *Red Hat Enterprise Linux 5.5+, 6.x*
- *Oracle Linux 5.5+; 6.x; 7.x*

Hardverkövetelmények tekintetében a *Java JRE 8* futtatásához szükséges minimum követelményei az irányadóak. Azonban a program bizonyos esetekben több erőforrást is igényelhet, ezért ajánlott nagyobb memóriával és erősebb processzor rendelkező rendszer használata. A követelmények a következő táblázatban találhatóak:

2.1. RENDSZERKÖVETELMÉNYEK

Memória	128 MB
Szabad lemezterület	124 (+2) MB
Processzor	<i>Pentium 2 266 MHz</i>

2.1.2. Ajánlott rendszerkövetelmények

A szoftver tökéletes működéséhez a legelterjedtebb operációs rendszer, a *Windows* ajánlott. Továbbá követelmény 16:9-es képaránnyal rendelkező monitor, és legalább 1366×768 képernyőfelbontás használata.

A rendszer fejlesztése a következő ajánlott konfigurációra történt:

Operációs rendszer	<i>Windows 7</i>
Memória	4 GB
Processzor	<i>Intel Core i5-2467M, 2000 MHz</i>

2.1.3. Telepítés és eltávolítás

Alapvetően elegendő a *Java* futtatási környezet telepítéséről gondoskodnia a felhasználónak, ezen felül más program telepítésére nincs szükség. Azonban, mivel elsősorban *Windows* operációs rendszeren történő futtatásra lett felkészítve a program, ezt az operációs rendszert használók választhatják a kényelmesebb, natív telepítéses megoldást. Elsőként a programhoz készült telepítővel történő konfigurálást vesszük végig, majd ezt követően a *Java* programokra inkább jellemzőbb, ám kissé körülményesebb telepítési mód kerül bemutatásra. Végül röviden összefoglaljuk a program eltávolításához szükséges lépéseket.

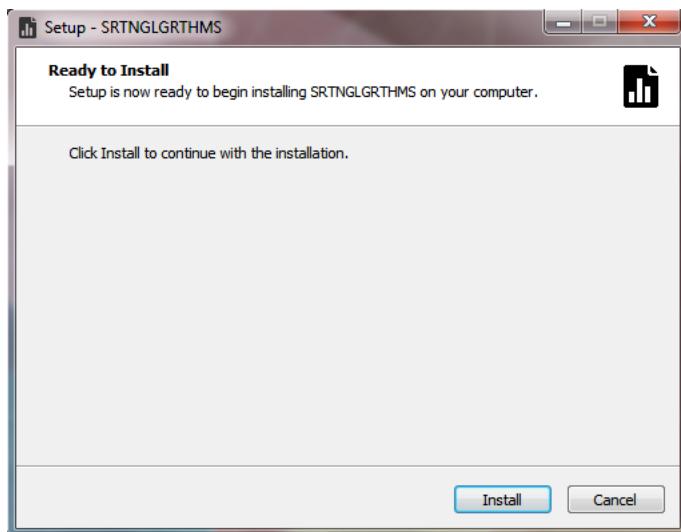
Bármely módszert is szándékozik követni a felhasználó, először győződjön meg róla, hogy az előzőekben ismertetett rendszerkövetelményeknek megfelel a számítógépe

Telepítés natív telepítővel

Ez a telepítési mód csak a *Windows*-t használók számára érhető el.

Az első lépés a telepítési varázsló elindítása. A megjelenő párbeszédpanelon kattintsunk a **Telepítés** gombra. Ezt követően elindul a telepítés, ami körülbelül fél percet vesz igénybe.

2.1. RENDSZERKÖVETELMÉNYEK



2.1. ábra. A telepítési párbeszédpanel

A telepítési panel bezárása után márás megjelenik a program főmenüje, így megkezdheti a felhasználó a használatát. A későbbiekben történő futtatáshoz a következő könyvtárba szükséges navigálni: C:\Felhasználók\{Felhasználói név}\AppData\Local\SRT

A fenti útvonal akkor érvényes, ha C:\ meghajtón található az operációs rendszer, néhány rendszeren más lehet ennek a meghajtónak a betűjele. Továbbá egyes számítógépeken az **AppData** mappa rejtett lehet, így érdemes valamilyen fájlbön-gészről, például *Total Commander*-t használni.

Hagyományos telepítés

A most ismertetésre kerülő telepítési mód minden operációs rendszeren elérhető.

Az első feladat a *Java* virtuális gép telepítése, amely letölthető a következő web-címről: <http://java.com/inc/BrowserRedirect1.jsp>

Fogadjuk el a licencszerződést, töltük le a *JRE* telepítőfájlt. Ha frissíteni szeretnénk a jelenlegi *Java* verziót a rendszerünkön, akkor előbb célszerűbb eltávolítani a régi verziót. Miután feltelepítettük a futtatási környezetet, készen áll a szoftver futtatására a rendszerünk. A program könyvtárában található **SRTNGLGRTHMS.jar** fájl elindításával kezdhetjük meg a szoftver használatát.

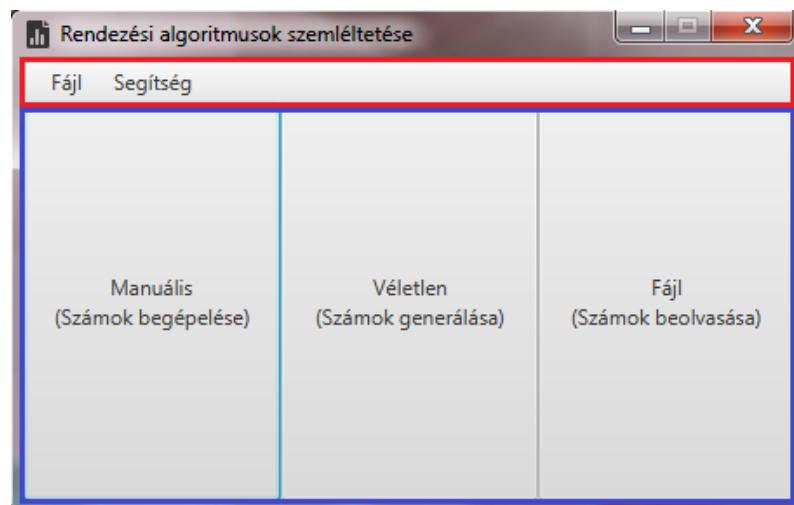
Eltávolítás

A program törlése az utóbb ismertetett telepítési mód esetében mindenössze annyiból áll, hogy a **SRTNGLGRTHMS.jar** fájlt eltávolítjuk, valamint amennyiben a jövőben nincs igény a *Java* futtatási környezet használatára, úgy a felhasználó eltávolíthatja azt. *Windows*-on történő natív telepítést követően a program telepítési könyvtárában található **unins000.exe** fájlt futtatva, majd az **Igen** gombra kattintva a program törlődik a számítógépről.

2.2. Felhasználói felület bemutatása

2.2.1. Főmenü

A program indítását követően megjelenik a főmenü, mely két komponensből áll.

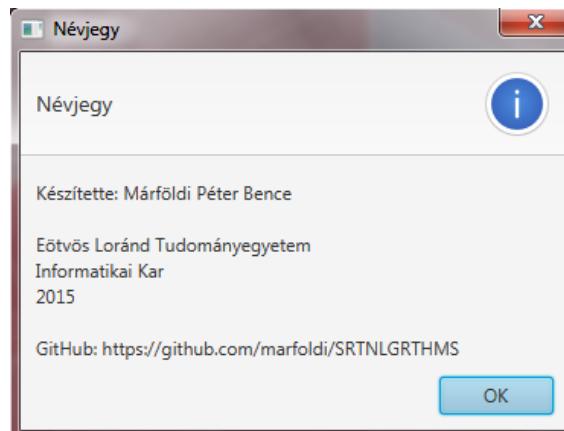


2.2. ábra. A program főmenüje

Az ablak felső részén található piros színnel jelölt rész az eszköztárat foglalja magában. A kék szín jelöli a főmenü központi paneljét, mely három gombból tevődik össze. Ezen gombokra történő kattintás után lehetőség nyílik a rendezendő számok megadására.

Eszköztár

Az eszköztárat két menüpont alkotja, **Fájl** és **Segítség** címszóval ellátva. Az előzőben a program bezárásának lehetősége kapott helyet, míg utóbbiban a szoftver névjegye tekinthető meg. Az **Ok** gomb lenyomásával bezárható a névjegy.



2.3. ábra. A program névjegye

Központi panel

A három gombból áll:

- Manuális (számok begépelése)
- Generálás (számok generálása)
- Fájl (számok beolvasása)

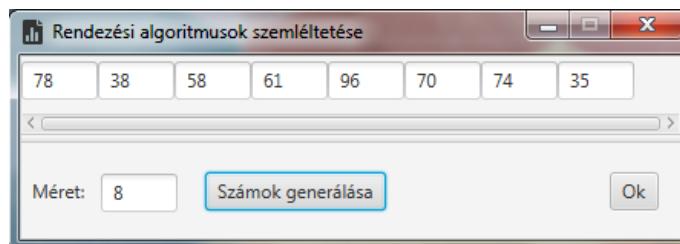
Ezek közül bármelyre kattintva átnavigálhatunk a bement megadását lehetővé tévő felületekre.

2.2.2. Bemenet megadása panel

Az előzőekben említett három lehetőség közül választhat a felhasználó.

Manuális

A megjelenő panelen két gomb található, melyek kezdetben inaktívak. A "Méret:" címke után található beviteli mezőbe egy pozitív egész szám megadásával és az **ENTER** billentyű leütésével megjelennek a számok bevitelére lehetőséget adó mezők. Ezt követően a panelen található két gomb már kattintható, a **Számok generálása** gombra kattintva 0 és 100 közötti véletlen számokkal töltődnek fel a mezők. Az **Okra** történő kattintás után megjelenik a program **Főpanelje**.



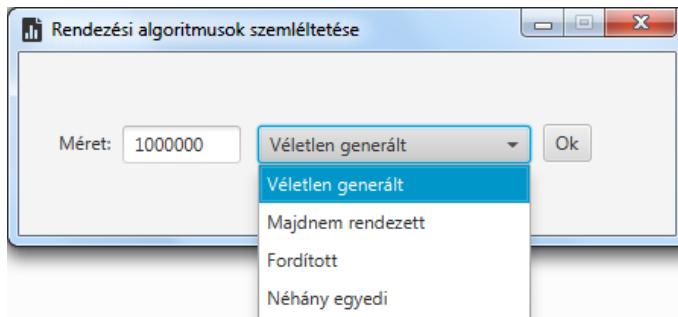
2.4. ábra. 8 elemű manuálisan megadott bemenet

Meg kell jegyezni, hogy ebben a módban legfeljebb száz érték adható meg, ennél nagyobb méretű bemenet manuális feltöltése túl körülményes lenne. Amennyiben több számot szándékozik megadni a felhasználó, válasszon a számok generálása vagy a fájlból történő beolvasás lehetőségek közül.

Generálás

A "Méret:" címke mellett megadva a bemeneti számok mennyiségét, és a legörökülő menüből kiválasztva a generálás módját az **OK** gomb kattinthatóvá válik.

2.2. FELHASZNÁLÓI FELÜLET BEMUTATÁSA



2.5. ábra. Egymillió véletlen generált érték megadása

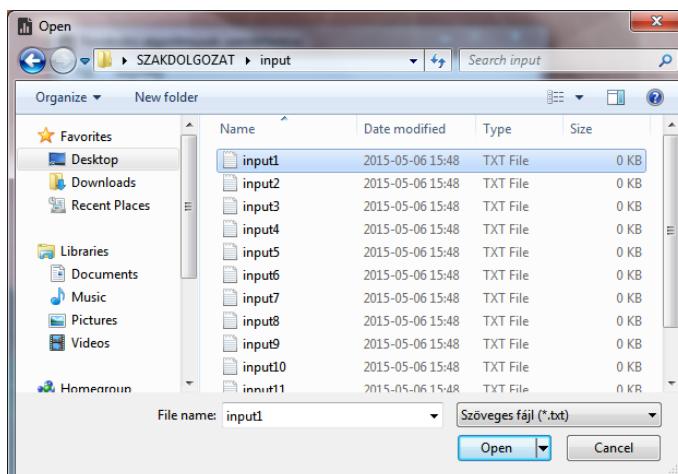
A legördülő menüből a következő négy típus közül lehet választani, azaz a generált tömb legyen:

- Véletlen generált - véletlenszerűen választott számokból álljon
- Majdnem rendezett - a tömb 80%-a már rendezve legyen
- Fordított - a tömb legyen csökkenőleg rendezett
- Néhány egyedi - a tömb elemei között sok azonos érték szerepeljen

Itt megjegyzendő, hogy amennyiben az input mérete az $[1,100]$ intervallumban van, akkor a rendezendő számok a 0 és 100 közötti értékek közül kerülnek kiválasztásra. Ennek oka, hogy a túl nagy differencia az egyes értékek között sokat rontana az oszlopdiagramok megjelenésén. Ellenkező esetben a *Java* nyelv által definiált egész típus(*Integer*) maximális értékéig terjedhet a generált számok nagysága.

Fájlból beolvasás

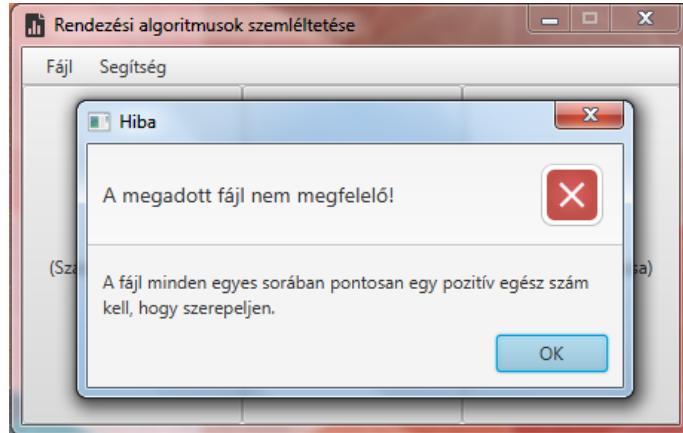
A gombra történő kattintás után megjelenik egy fájltallázó. A tallázóban csak szöveges(*txt* kiterjesztésű) fájl választására van lehetőség.



2.6. ábra. Fájltallázó

2.2. FELHASZNÁLÓI FELÜLET BEMUTATÁSA

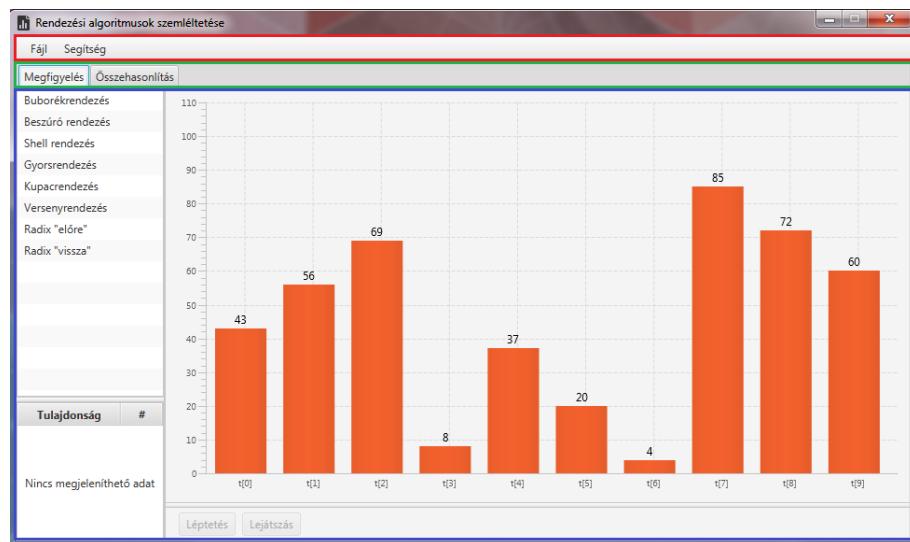
A fájl tartalmára történő megszorítások szigorúak. minden sora legfeljebb egy pozitív egész számot tartalmazhat, ellenkező esetben a program hibaüzenet kíséretében visszatér a főmenübe. Fontos, hogy az előzőek értelmében az üres sorok sem megengedettek.



2.7. ábra. Nem megfelelő fájl esetén a hibaüzenet

2.2.3. Főpanel

A rendezendő számok sikeres megadása után megjelenik a program főpanelje, mely három logikai részből áll. A főpanel magában foglalja a **Megfigyelés** és **Összehasonlítás** paneleket, melyek összetettségükön külön alfejezetekben kerülnek részletezésre.



2.8. ábra. A program főpanelja

Eszköztár

A **Főmenü**höz hasonlóan itt is jelen van a piros színnel jelzett eszköztár sáv. Itt fontos kiemelni, hogy a **Fájl** és **Segítség** pontokon belül további alpontok is elérhetőek:

A **Fájl** menüpont bővül a **Vissza a főmenübe** lehetőséggel, melynek segítségével a program újraindítása nélkül lehetőség van újabb rendezendő számsorozat megadására.

A **Segítségre** kattintva további lehetőségekkel választható az **Algoritmusról** pont. Itt elolvasható a rövid szöveges ismertetője a **Megfigyelés** panel listájából kiválasztott elemek. Amennyiben pedig az **Összehasonlítás** panel aktív, akkor e panel táblázatából kiválasztott algoritmus leírása tekinthető meg. Az aktuális panelen ha nem került kiválasztásra sor, akkor a menüpontra történő kattintás után felugró ablak figyelmezteti a felhasználót arról, hogy e menüpont használatához előbb ki kell választani egy algoritmust.

Panelválasztó

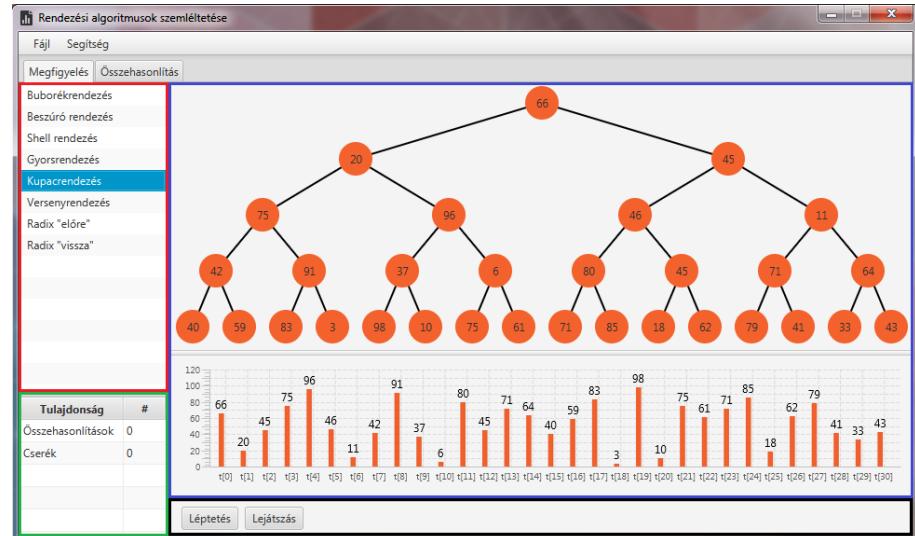
Amennyiben a bemenet megadásánál már ismertetett [1,100] intervallum magában foglalja a bemenet hosszát, akkor két elem látható a zöld jelölt részen. Így lehetősége van a felhasználónak navigálni a **Megfigyelés** és az **Összehasonlítás** panel között. Ellenkező esetben csak az utóbbi panel jelenik meg. Az aktuálisan megjelenített felület neve a listában kék kerettel jelenik meg, valamint a másik elemhez képest világosabb szürke színnel.

Panel

A harmadik logikai egységet alkotják a panelek - kék színnel jelölve -, melyek közötti váltást a **Panelválasztó** teszi lehetővé. Mivel részletesebb leírást kíván a panelek ismertetése, ezért egy-egy külön alfejezetben kerülnek bemutatásra.

2.2.4. Megfigyelés panel

A főpanel középső területén foglal helyet, négy komponensből tevődik össze. A felhasználónak itt nyílik lehetősége az egyes algoritmusok megfigyelésére, tanulmányozására.



2.9. ábra. Megfigyelés panel

Algoritmus lista

Az ábrán piros színnel jelölt rész, melynek elemeire kattintva kiválasztható, hogy mely algoritmust szeretné a felhasználó vizsgálni. A kiválasztott elem kék háttérszínt kap, alapértelmezett esetben nincs kijelölt elem.

Állapotjelző táblázat

A kiválasztott algoritmus aktuális állapotához tartozó információk jelennek meg a táblázatban. Az ábrán zöld kerettel van jelölve a komponens. Amennyiben nincs kiválasztott elem a **Nincs megjeleníthető adat** szöveg jelenik meg.

Két oszlopa a **Tulajdonság** és a **#**, mely utóbbi az értéket jelöli. Alapvetően az összehasonlításon alapuló algoritmusoknál megjelenő adatok az összehasonlítások és cserék vagy mozgatások száma. Az edényrendezésekben pedig az aktuálisan vizsgált bit indexe és a vizsgálatok száma jelenik meg. Egyes algoritmusokhoz a jobb megérhetőség miatt további állapotjelző értékek is tartoznak, melyek a következők:

Algoritmus	Tulajdonság
Shell rendezés	Lépésköz
Gyorsrendezés	Vezérelem
Radix "előre"	Cserék száma

Gombok

Alaphelyzetben a **Léptetés** és **Lejátszás** gombok inaktívak. A képen fekete keretet jelzi a helyüket. Amennyiben a felhasználó kiválaszt egy elemet az algoritmus listáról kattinthatóvá válnak. A léptetéssel egy következő állapotot tekinthet meg

a felhasználó. A **Lejátszás** gombra kattintva a program bemutatja az algoritmus működését. A felhasználó bármikor megállíthatja az animációt a **Lejátszás** gomb helyén található **Megállítás** gombra kattintva, majd ha kívánja innen folytathatja a vizsgálatot. Amennyiben a rendezés lezajlott, megjelenik az **Újraindítás** gomb, melynek megnyomásával az értékek visszakerülnek az eredeti helyükre.

Állapotjelző felület

A fenti ábrán kék szín jelöli ezt a területet. Az állapotjelző felületen minden esetben legalább egy oszlopdiagram foglal helyet. Ettől eltérően a **Kupac-** és **Versenyrendezés** kiválasztásakor egy gráf is helyet kap. Továbbá a **Radix "vissza"** rendezés második tömbjének reprezentálásához egy plusz oszlopdiagram is megjelenik. Itt megjegyzendő, hogy a szemléltetéshez használt bináris fa legfeljebb 31 csúcsot tartalmazhat. Hosszabb bemenet megadásakor csak az első 31 elem vesz részt a megjelenítésben, erről felugró ablak tájékoztatja a felhasználót:



2.10. ábra. Megváltozott bemenetről a figyelmeztetőablakok

Az egyes műveleteket különböző színek is jelölik. Alapértelmezetten a rendezendő elemek színe a következőket jelentik:

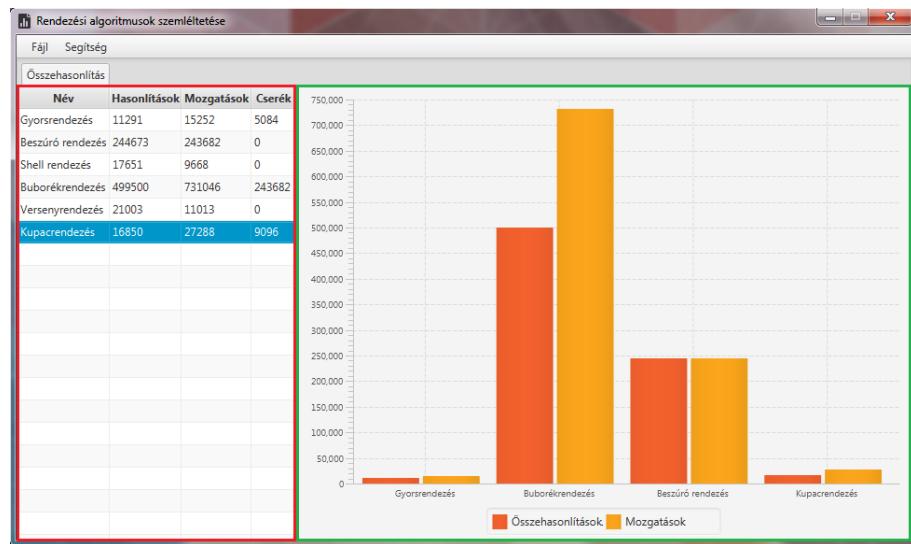
- Az alapértelmezett szín, nincs kijelölve az elem.
- Az elem cserére vagy mozgatásra van kijelölve.
- Az elemnek kitüntetett szerepe van.
- Az elem már a végleges helyére került.

A ● jelölés némi magyarázatra szorul. A kitüntetett szerepű elemek számít például a gyorsrendezés vezéreleme, vagy a versenyrendezés fájának felépítésekor egy belső csúcsba kerülő elem.

Némely algoritmusnál a fentiekkel eltérő lehet az egyes színek jelentése. A következő fejezetben(2.3), az algoritmusok ismertetésénél jelölve van minden ilyesfajta különbség.

2.2.5. Összehasonlítás panel

A **Megfigyelés panel**lel megegyezően a főpanel középső részén található, két logikai egységből épül fel. Lehetőséged ad az összehasonlításon alapuló rendezések műveletigényeinek a vizsgálására.



2.11. ábra. Összehasonlítás panel

Elemzés táblázat

Piros kerettel jelölt rész az ábrán, mely táblázatnak négy oszlopa van:

- Név - az algoritmus neve
- Hasonlítások - a rendezés során végzett összehasonlítások szummája
- Mozgatások - az elemmozgatások számának összege
- Cserék - a végzett cserék szummája

A **Mozgatások** oszlop minden esetben kitöltésre kerül, még ha az algoritmus nem is mozgatásokat használ. Ekkor a mozgatások oszlopban a cserék számának háromszorosa jelenik meg. Ennél fogva egyszerűbb az algoritmusok vizsgálata.

Elemzés diagram

A táblázathoz képest balra helyezkedik el a zöld kerettel jelölt oszlopdiagram. Kezdetben teljesen üres, csak a jelölések jelentése látható. Az **Elemzés táblázat** egy során történő dupla kattintás következtében megjelenik az algoritmus összehasonlításainak és mozgatásainak a száma. Előbbi narancs- utóbbi citromsárga színnel. Amennyiben olyan sorra kattint a felhasználó, mely már látható a diagramon, azon algoritmus adatai eltűnnek a felületről.

2.3. A vizsgált algoritmusok

2.3.1. Buborékrendezés

Leírás

A legrégebbi és a legegyszerűbb rendezési algoritmus. Mindemellett a legtöbb esetben a leglassabb is. Már az 1965-ös évben megjelent egy teljes körű elemzése[4].

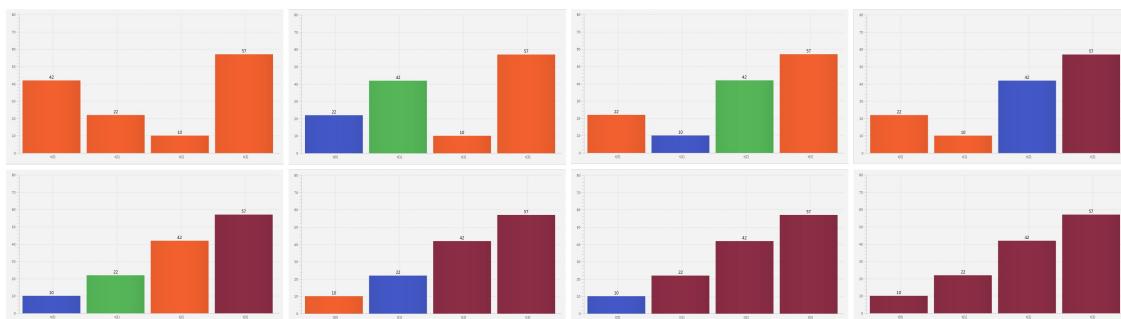
Működés

A rendezés minden egyes elemet összehasonlít a rakkövetkező elemmel, és ha szükséges megcseréli őket. Ez azt eredményezi, hogy lépésenként a maximális elem "buborék" szerűen a lista végére kerül, ezzel egyidejűleg a kisebb elemek "lesüllyednek" a tömb elejére. Amennyiben egy menetben a maximális eleme elérte a helyét visszavezetjük a problémát az eggyel "rövidebb" rendezési feladatra[1]. Az algoritmus javítható azzal, ha figyeljük, hogy az egyes menetekben történt-e csere. Amennyiben egy olyan menet végére értünk, amelyben egy elem helye sem változott, akkor a tömb már rendezve van. A program az eredeti, nem javított verziót mutatja be.

Példa

A rendezendő számok: 42, 22, 10, 57.

Az első ábrán szerepel a kezdeti állapot, a másodikon az első két értéket cseréje látható. A 42 ismét fentebb kerül egy pozícióval a harmadik ábrán. A negyedik ábrán csupán egy összehasonlítás történik, mivel az 57 nagyobb mint az előző érték. Az ötödik ábrán az előző állapot első két értékének(22 és 10) a cseréje látható. Az ezt követő összehasonlítás eredménye, hogy a második legnagyobb érték(42) a helyére került. Ezt követően már csak egy összehasonlítást történik, mely után az adatsor rendezve lesz.



2.12. ábra. Példa a buborékrendezésre

Műveletigény

Párosával haladunk végig az elemeken, és így vizsgáljuk őket. Mivel minden menet végén visszavezetjük a problémát az eggyel rövidebb feladatra, ezért az összehasonlítások száma n hosszú bemenetre:

$$\ddot{O}(n) = (n - 1) + (n - 2) + \dots + 1 = \frac{n \cdot (n - 1)}{2} = \frac{n^2}{2} - \frac{n}{2} = \Theta(n^2)$$

A cserék száma már nem állandó, a bemenő adatok inverziószámával egyezik meg. Belátható, hogy minden egyes cserével egy inverzió szüntethető meg két elem között. A legtöbb cserét akkor szükséges eszközölni, ha a rendezendő elemek minden egyike inverzióban áll a rákövetkező elemmel, azaz a tömb nagyság szerint csökkenő sorrendben rendezett. Ekkor a cserék száma:

$$MCs(n) = \frac{n \cdot (n - 1)}{2} = \Theta(n^2)$$

Amennyiben a tömb elemei már rendezettek, akkor egyetlen cserét sem szükséges végrehajtani. Az átlagos csereszám a maximális cserék számának fele[1], ám nagyságrendileg még ez is $\Theta(n^2)$

Jelölések az állapotjelző felületen

Kitüntetettnek ( színnel) jelöljük azt az elemet, amely egy csere következtében feljebb került. Nincs egyéb eltérés az eredeti jelölésekhez képest.

2.3.2. Beszúró rendezés

Leírás

A legtöbb esetben akár kétszer hatékonyabb a buborékrendezéshez képest[1]. Az elve egyszerűen megérthető egy hétköznapi példán keresztül: A kezünkben tartunk kártyalapokat, majd egy pakliból húzva az új lapot beszúrjuk a kezünkben lévő, már rendezett lapok közé. Ennél fogva szokás kártyás rendezésnek is nevezni.

Működés

Két részre bontjuk a tömböt, az egyik részén - a tömb bal széle - a már rendezett, míg a másikon a még nem vizsgált elemek szerepelnek. Kezdetben a tömb első elemét tekintjük a rendezett résznek.

Lényege, hogy a soron következő elemet - a második elemtől kezdve - egy ideiglenes változóba mentjük, és a rendezett tömbrész elemeit jobbra csúsztatjuk minden-

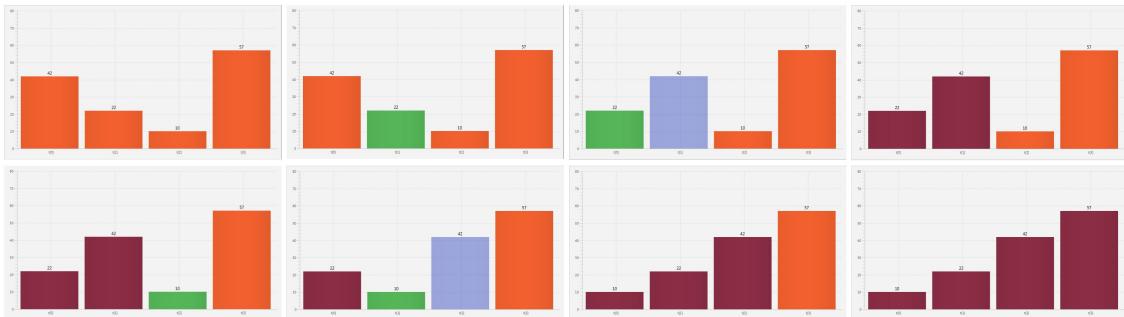
2.3. A VIZSGÁLT ALGORITMUSOK

addig, amíg a kiválasztott érték nem kerül a helyére. Ezt n-1 alkalommal ismételve megkapjuk a rendezett tömböt.

Példa

A rendezendő számok megegyeznek a buborékrendezés bemutatásánál használtakkal, azaz: 42, 22, 10, 57.

Az első ábrán szerepel a kezdeti állapot. A következő képen jelöljük azt, hogy a második elemet fogjuk beszúrni. A harmadik ábrán felcsúsztatjuk a 42 értéket, majd a negyedik képen már látható, hogy a tömb első két eleme alkotja rendezett tömbrészt. Ezt követően a harmadik elem kerül kiválasztásra. Az összehasonlítások eredményeképp a teljes rendezett tömbrészt jobbra tolódik, így az 7. képen már a tömb első három eleme megfelelő sorrendben van. Végül a tömb utolsó eleme kerül összehasonlításra az ōt megelőzövel, mivel nagyobb tőle, így kész a rendezési feladat.



2.13. ábra. Példa a beszúró rendezésre

Műveletigény

Legjobb esetben elegendő a második elemtől minden egyik elemet az előtte található értékkel összehasonlítani, vagyis a tömb már eleve rendezett[3]. Ez pontosan n-1 összehasonlítást jelent, azaz:

$$m\ddot{O}(n) = (n - 1) = \Theta(n)$$

Ebben az esetben pedig egyetlen elemet sem kell mozgatni.

A legrosszabb eset akkor áll fenn, ha a beszúrandó elem minden alkalommal kisebb a már beszúrt elemknél, azaz a tömb elemei csökkenőleg rendezettek[1]. Ekkor az összehasonlítások száma:

$$M\ddot{O}(n) = \sum_{i=1}^n i = \frac{(1 + (n - 1)) \cdot (n - 1)}{2} = \frac{n \cdot (n - 1)}{2} = \Theta(n^2)$$

Továbbá ekkor a mozgatások száma:

$$MM(n) = \Theta(n^2)$$

Jelölések az állapotjelző felületen

A már rendezett tömbrészt ● szín jelöli. Az aktuálisan beszúrandó elem pedig a ● jelölést kapja. A kiválasztott elemmel összehasonlított értékek ● színt kapnak.

Megjegyzendő, hogy a kitüntetett szerepű vagy beszúrandó elem nem szerepel tömbelemként az összehasonlításkor, csupán a felületen van jelölve a jobb megértetőség okán.

2.3.3. Shell rendezés

Leírás

Donald Shell nevéhez fűződik, a legtöbb esetben a leggyorsabb négyzetes idejű algoritmus. Az elve az, hogy célszerű lehet előbb a "távolabb" lévő elemeket használni és mozgatni, mivel így az elemek hamarabb közel kerülhetnek a végleges helyükhez.

Működés

Többször vizsgálja a tömböt, és minden alkalommal egy részén beszúró rendezést hajt végre. Arra, hogy mekkora méretű résztömböt vizsgáljon az egyes lépésekben az algoritmus több javaslat is található. A teljesség igénye nélkül néhány ajánlás[5] erre vonatkozóan:

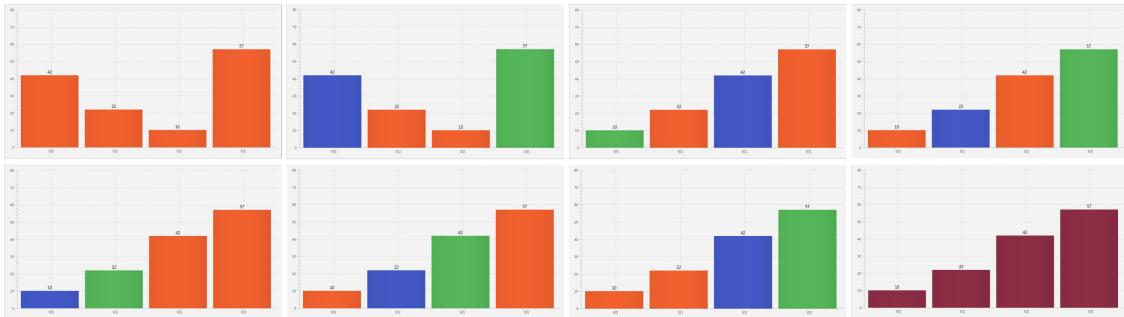
Szabály (k=1...)	Konkrét értékek	Műveletigény	Szerző
$\lfloor n/2^k \rfloor$	$\left\lfloor \frac{n}{2} \right\rfloor, \left\lfloor \frac{n}{4} \right\rfloor, \dots, 1$	$\Theta(n^2)$	Shell, 1959
$2^k - 1$	1, 3, 7, 11, ...	$\Theta(n^{\frac{3}{2}})$	Hibbard, 1963
$2^p 3^q$ váltakozva	1, 2, 3, 4, ...	$\Theta(n \log^2 n)$	Pratt, 1971

Példa

Az előző példákban használt számokat kívánjuk rendezni ismét, azaz a bemenet: 42, 22, 10, 57.

2.3. A VIZSGÁLT ALGORITMUSOK

Az első képen látható a kezdeti állapot. A következő ábrán jelöljük azt, hogy a negyedik elemet fogjuk mozgatni, ekkor a lépésköz három. A harmadik képen már kettő a vizsgált elemek távolsága. A harmadik, kiválasztott elemet beszúrjuk az első elem helyére. A negyedik képen vesszük a következő elemet, továbbra is 2 lépésközzel, ekkor nem szükséges mozgatni. Végül az ötödik megjelenített lépéstől kezdve beszúró rendezést alkalmazunk.



2.14. ábra. Példa a Shell rendezésre

Műveletigény

A fentebbi táblázatból látható, hogy az algoritmus sebessége nagyban függ a lépésköz megválasztásától. A program a **Vaughan Pratt** által javasolt értékeket[6] használja, így legrosszabb esetben $\Theta(n \log^2 n)$ a műveletek száma.

A legjobb eset akkor áll fenn, ha a tömb elemei már rendezettek, ekkor nincs szükség mozgatásra, az algoritmus futási ideje $\Theta(n)$ nagyságrendű.

Jelölések az állapotjelző felületen

A felület bemutatásában szereplőkhöz képest nincs eltérés.

2.3.4. Gyorsrendezés

Leírás

C.A.R. Hoare[7] alkotta meg 1965-ben. Az egyik leggyorsabb rendezési eljárás, ezért rendkívül gyakran alkalmazzák.

Működés

Helyben rendező, oszd meg és uralkodj[2] elven működő rekurzív algoritmus. A következő négy lépésre bontható fel az rendezés:

- Ha csak egy vagy nulla elemű az elemzett rész, akkor ne tegyünk semmit.
- Válasszunk egy vezérelementet (legjobb oldalibb elem).

2.3. A VIZSGÁLT ALGORITMUSOK

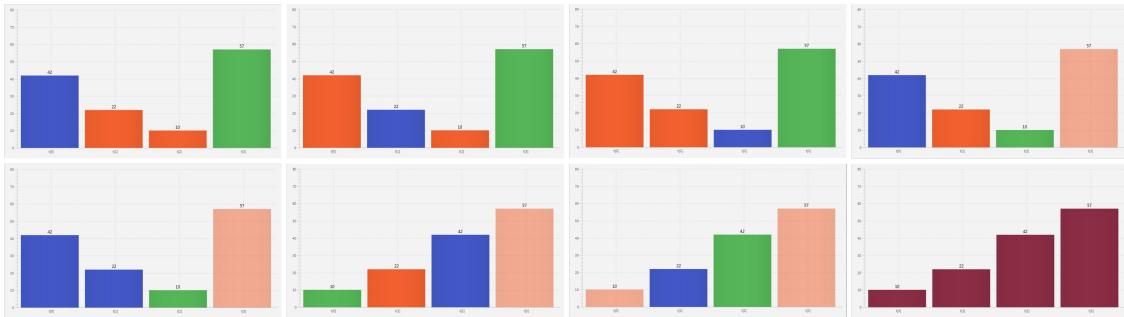
- Osszuk két részre a rendezendő részt, mozgassuk a vezérelemtől kisebb elemeket, míg a nagyobbakat a tömb végébe[3].
- Rekurzívan ismételjük meg az előbbi lépéseket a résztömbökön.

Gyakorlatilag szétválasztást eszközölünk, melynek eredményeképp a kiválasztott elemtől kisebb értékek tőle balra, a nagyobbak pedig jobbra helyezkednek el. Az egyszerűbb megérthatóséget szem előtt tartva a program minden a legjobboldalibb elemet szelekáltja.

Példa

Az eddigi példákkal megegyezően a bemenet: 42, 22, 10, 57.

Az első képen látható a már kiválasztott vezérelem. A negyedik ábráig csak összehasonlítások történnek, mivel minden elem kisebb, mint a kiválasztott érték. A negyedik képen a három elemű résztömbön végezzük el a rendezést, a vezérelem ismét a legjobboldalibb érték. Az ötödik képen az első érték nagyobb mint a vezérelem, így ezt az elemet a hatodik képen megcseréljük a vezérelemmel. Ezt követően a középső két elem kerül összehasonlításra, mivel nincs szükség cserére így a rendezés befejeződött.



2.15. ábra. Példa a gyorsrendezésre

Műveletigény

A rendezés műveletigényét befolyásolja, hogy hogyan választjuk meg a vezérelemet. Például a legrosszabb futási időt ($\Theta(n^2)$) eredményezi, ha minden a legjobboldalibb elemet választjuk vezérelemnek, és a tömb elemei csökkenő sorrendben vannak[2]. Éppen ezért a gyakorlatban javasolt ezen elem véletlenszerű megválasztása.

Ha feltételezzük azt, hogy minden rekurzív lépés felezi a tömböt, akkor egy $n \log(n)$ magasságú fával lehet ábrázolni a rekurziót. Továbbá, mivel minden szinten az elemek száma n , és ezek particionálásához használt lépésszám $\Theta(n)$ így a legjobb esetben a futási idő $\Theta(n \cdot \log(n))$.

A gyorsrendezés a legtöbb esetben(közepes és nagy méretű bemenetre) a legmegfelelőbb választás ha számít a rendezés sebessége, mivel az átlagos futási ideje - $\mathcal{O}(n \cdot \log(n))$ - közel áll a legjobb futási időhöz[2].

Amennyiben a tömb elemei már eleve rendezettek vagy esetleg fordított sorrendben szerepelnek nem hatékony az eljárás.

Jelölések az állapotjelző felületen

A vezérelement  szín jelöli. Annak érdekében, hogy jobban átlátható legyen, hogy éppen melyik résztömbön folyik a vizsgálat az éppen nem vizsgált rész haloványabb sárga színnel van jelölve. A többi jelölés a felület bemutatásában leírtaknak megfelelően történik.

2.3.5. Kupacrendezés

Leírás

A módszert **J. W. J. Williams** és **R. W. Floyd** javasolták 1964-ben[3]. Az $\mathcal{O}(n \log(n))$ algoritmusok közül az egyik leglassabb, azonban előnye a gyorsrendezéssel szemben, hogy nem erőteljesen rekurzív. Ennél fogva jól alkalmazható milliós nagyságrendű bemenetre. Ahogyan a neve is sugallja, a rendezéshez egy kupac adatszerkezetet használ. Az algoritmus ismertetése előtt definiáljuk a kupac fogalmát[1]: Olyan bináris fa, amelyre a következők teljesülnek:

- Kizárolag a levelek szintjén hiányozhat csúcs, azaz "majdnem teljes".
- A levélszint csúcsai balra tömörítettek.
- minden belső csúcs értéke nagyobb vagy egyenlő, mint a gyerekeinek értékei.

A második pont értelmében egyetlen olyan csúcs lehet, amelynek csak egy gyereke van, és az közvetlenül a levélszint felett kell, hogy elhelyezkedjen.

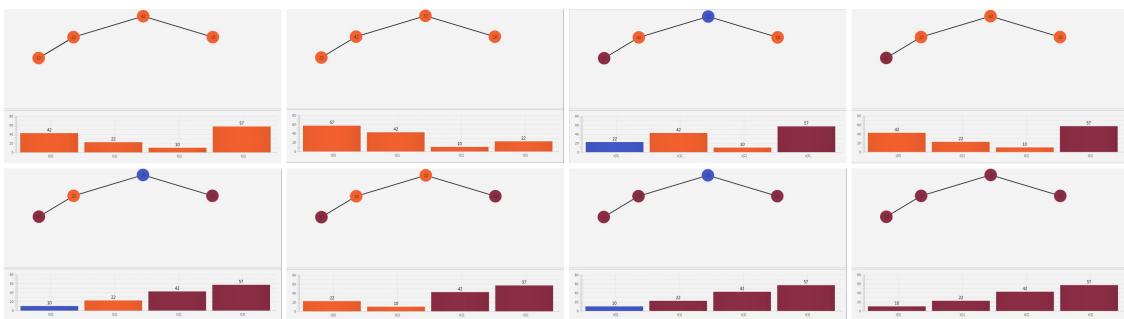
Működés

A rendezés az előbbi tulajdonságokra támaszkodik. Az algoritmus a bemeneti elemekből kupacot épít, majd a legfelsőbb elemét áthelyezi a kupac "végére". Ezt követően ellenőrzi a kupac tulajdonságokat, ahol szükséges cseréket hajt végre, hogy helyreálljon a kupac adatszerkezet, ekkor már az utolsó, legjobboldalibb levélelem nem vesz részt a kupacépítésben. A gyökérben található elemet a legjobboldalibb levélelem elé helyezi, és újraépíti a kupacot. Ezen lépések addig ismétlődnek, amíg már a maximális elem áthelyezése nem lehetséges, a kupac "végére" helyezési művelet elérte a gyökeret.

Példa

Az eddigi példákkal azonosan a rendezendő számsorozat: 42, 22, 10, 57.

Az első kép a kezdeti állapotot mutatja. Ezt követően a bemeneti adatokból kupacot építünk, melynek eredménye a második ábra. A gyökérelemet a levélszint utolsó elemével megcseréljük a harmadik képen. Ellenőrizzük, hogy teljesülnek-e a kupac tulajdonságok, amennyiben nem cseréket hajtunk végre, ennek eredménye látható a 4. képen. Ismét a gyökérelemet lesüllyesztjük, majd ellenőrizzük a kupac tulajdonságot. Az előző két lépést megismételve rendezett tömböt kapunk.



2.16. ábra. Példa a kupacrendezésre

Műveletigény

A kupacrendezés egyik további, hogy még a gyorsrendezésnek legrosszabb esetben a futási ideje $\mathcal{O}(n^2)$, addig itt a futási idő továbbra is $\mathcal{O}(n \cdot \log(n))$. Ennél fogva azon rendszereknél, ahol a négyzetes futási idő elfogadhatatlan inkább kupacrendezést alkalmaznak.

Jelölések az állapotjelző felületen

Az éppen összehasonlított értékeket ● jelöli. Amennyiben csere történt a belső csúcsba kerülő érték háttérszíne ● lesz. Ez alól kivétel, ha a gyökérbe került új érték, mivel ekkor már tudjuk az új legnagyobb értéket, ennek a színe ● lesz. Továbbá akkor is ezt a színt használjuk, ha már a tényleges, végleges helyre került egy elem.

2.3.6. Versenyrendezés

Leírás

A maximum-kiválasztó rendezések közé tartozik, minden egyes menetben kiválasztja a legnagyobb elemet, kiírja és végül eltávolítja. A maximum kiválasztásnak a gyakorlati háttérét a sportesemények lebonyolítási rendje adja, azaz meghatározza

2.3. A VIZSGÁLT ALGORITMUSOK

az elemek között a "nyertest"[1]. A módszert $n=2^k$ inpushossz esetén érdemes alkalmazni, mivel ettől értérő bemenetre sokkal kedvezőbb eredményt lehet elérni a kupacrendezéssel[1].

Működés

Az algoritmus által használt adatszerkezet egy teljes bináris fa. A bináris fa leveleiben szerepelnek a rendezendő elemek. Az első speciális menetben a fa belső pontjait kitöljtük, úgy, hogy a pontba a gyerekei közüli nagyobb érték kerül.

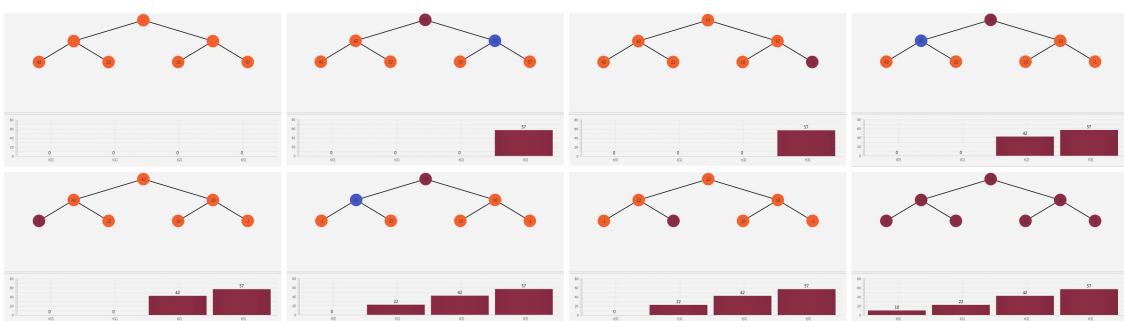
Ezt követően kerül sor az $(n - 1)$ egyszerűbb menetre: A gyökérben található elemet keresve "lefelé" haladunk a bináris fában, majd megtalálva azt a levelet amelyben a gyökér értéke szerepel egy abszolút vesztest állítunk a helyére. Ez az érték a programban -1, mivel csak pozitív egészeket használunk a rendezések szemléltetésére. Ezzel ellentétben a gyakorlatban ez az érték $-\infty$. Majd ezen az "ágon" úrajátsszuk a mérkőzéseket.

Amennyiben a bemenet hossza nem kettő hatvány a program -1 értékekkel tölti fel a bináris fa további leveleit, amíg a bemenet hossza nem lesz megfelelő.

Példa

Az eddigi példákkal megegyezően a bemenet: 42, 22, 10, 57.

Az első ábra a kezdőállapotot mutatja. A második kép a versenyfa kitöltését követő állapot, melynek eredményeképp megjelenik az abszolút maximum érték a diagramon. Ezt követően a harmadik képen látható a maximális értékhez tartozó levélelem megtalálásának állapota. A negyedik ábra szemlélteti az úrajátszás eredményét. Az előzőekhez hasonlóan szemléltetik a további képek a maximum érték megjelenítését, a levélelem megkeresését majd az úrajátszás eredményét.



2.17. ábra. Példa a versenyrendezésre

Műveletigény

A rendezés egyetlen hátránya a tárigénye, n szám esetén további $n - 1$ mezőre szükség van a versenyfa elkészítéséhez. Éppen emiatt a gyakorlatban nem sűrűn

használt eljárás. Az első, speciális menet, a versenyfa kitöltése $n - 1$ összehasonlítást és mozgatást használ ($n - 1$ a belső csúcsok száma). minden további menetben a fán kétszer kell végigmenni, melynek magassága $\log_2(n)$. Egyszer a maximális levél megtalálásához, majd az úrajátszáshoz, így a ezen menetek $2 \log_2(n)$ összehasonlítást végeznek. Mozgatás csak a második, úrajátszási művelethez tartozik. Így a műveletigények:

$$\ddot{O}(n) = n - 1 + (n - 1) \cdot 2 \cdot \log_2(n) = \Theta(n \cdot (\log(n)))$$

$$M(n) = n - 1 + (n - 1) \cdot \log_2(n) = \Theta(n \cdot (\log(n)))$$

Jelölések az állapotjelző felületen

Az összehasonlításokat, valamint a maximális levélelem megkereséséhez bejárt utat ● szín jelöli. A meccsek lejátszásakor a belső csúcsba kerülő érték ● jelölést kap. Ez alól kivétel, amikor a gyökérbe kerül egy meccs győztese, ekkor ● lesz az elem színe, továbbá akkor is ez a jelölés, amikor megtaláltuk a maximális levélcsúcsot.

2.3.7. Radix "előre"

Leírás

Az előzőekben ismertetett algoritmusok mindegyike összehasonlításon alapuló rendezés. A radix rendezés viszont az edényrendezések közé tartozik. Ezen rendezések nem hasonlítják össze az elemeket, hanem az elemek a rendezés során az értéküknek megfelelő edényekbe kerülnek. Az edényrendezések eredményeként rendezett adatsorozatot kapunk lineáris időben[3]. A radix rendezés egy rekurzív algoritmus, melynek minden szintjén létrejönnek az edények.

Az általános edényrendezés egy speciális változata a radix előre rendezés, bináris, d hosszú számokra.

Működés

Az első menetben a rendezendő elemek első bitjét vizsgálja az algoritmus. A vizsgálat két mutatóval történik, melyek a tömb két végéről indulnak. A tömb elején addig halad a mutató, amíg a vizsgált elem első jegye nem 1, ezzel párhuzamosan a tömb végén olyan elemet keres a másik, melynek első jegye 0. Amennyiben talált ilyen elemeket megcseréli őket. Ezt mindaddig folytatódik, amíg a két mutató nem

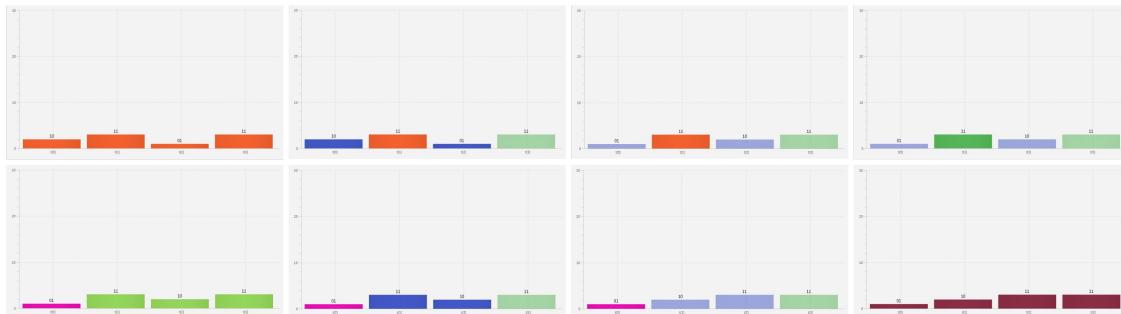
2.3. A VIZSGÁLT ALGORITMUSOK

találkozik. Ekkor kialakul két edény, az elsőben a 0-ás kezdőbittel rendelkező számok, míg a másodikban az 1-essel kezdődő elemek foglalnak helyet. Ezt követően a második bit kerül vizsgálatra az "aledényekben", az előzővel azonos módon. A rendezés befejeződött, ha minden számjegy szerinti vizsgálat megtörtént, vagy ha mindegyik, a futás alatt kialakult edény már csak egy elemet tartalmaz.

Példa

Az rendezendő számok: 2, 3, 1, 3, azaz binárisan 10, 11, 01, 11.

Az első ábra a kezdőállapotot mutatja. Mivel az első érték valószínűleg nem megfelelő helyen van, ezért a tömb végéről kezdve keresünk egy olyan értéket, amely 0-ás jeggyel kezdődik. A harmadik elem pont ilyen, a második képen ez a két érték cserére van kijelölve, majd a negyedik ábrán megtörténik a két elem cseréje. Az ötödik képen látható a menet során kialakult két edény. Mivel az első edény egy elemű, így nem szükséges a vizsgálata. A második edényben az 10 értéket szeretnénk cserélni, így az edény végétől haladunk előrefelé, amíg nem találunk olyan elemet, melynek második bitje egy. A 11 érték éppen ilyen, ezért megcseréljük őket. Az utolsó ábrán látható a rendezett sorozat.



2.18. ábra. Példa a radix "előre" rendezésre

Műveletigény

A rendezés lineáris időben történik, a számjegyek számát jelölje d , ekkor belátható, hogy a legrosszabb esetben is, azaz ha az összes szám minden bitjét meg kell vizsgálnunk az algoritmus futási ideje:

$$T(n) = \Theta(d \cdot n) = \Theta(n)$$

Jelölések az állapotjelző felületen

Az éppen vizsgált elem színnel van jelölve. Amennyiben két elemet fel kell cserálni háttérszínt kapnak. Továbbá a már vizsgált elemek háttérszíne fakóbb lesz. Amennyiben egy új edény keletkezik annak a színe véletlenszerűen választódik ki.

2.3.8. Radix "vissza"

Leírás

Az algoritmus rövid ismertetője megegyezik a **Radix "előre"** rendezés leírásával.

Működés

Az előző algoritmustól eltérően már nem helyben rendez, az eljárásnak két tömbre van szüksége. További különbség, hogy a kisebb helyiértéktől a nagyobb felé halad a vizsgálat.

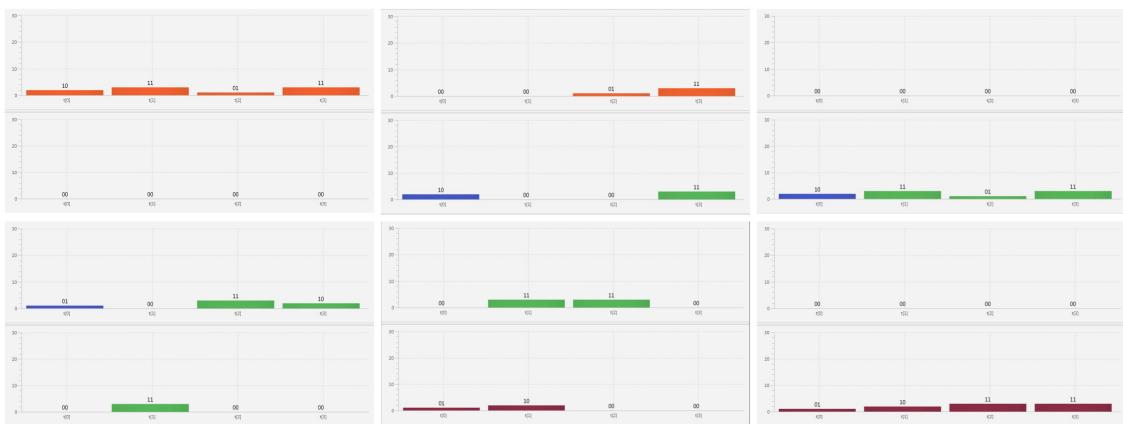
Amennyiben az aktuálisan vizsgált bit értéke 0, akkor a "második" tömb elejére, ellenkező esetben a tömb végére töltjük át az aktuális bináris számot. Ezáltal minden egyes menetben két edény keletkezik: egy, melyben található számok aktuális bitje 0, s egy másik, melyek vizsgált jegye 1-es. Ezt követően a 0-ás edényt az elejéről olvasva feltöltjük újra az eredeti tömböt az előbb ismertetett módon, majd a 1-es edényt elemeit az utolsó elemtől visszafelé haladva töltjük át az értékeket. Amennyiben a rendezendő bináris számok d hosszúak, úgy $d+1$ áttöltést követően rendezett tömböt kapunk.

Példa

A bemenet megegyezik a radix "előre" példájában használt számokkal, azaz: 2, 3, 1, 3, amelyek binárisan a 10, 11, 01, 11 értékek.

Az első ábra a kezdeti állapotot mutatja. A második képen a tömb első két eleme már áttöltésre került, a 10 az alsó tömb elejére, míg a 11 a végére. A harmadik ábrán látható a teljes áttöltött állapot. A következő ábrán visszatöltésre került az első érték, melynek első bitje 1, így az eredeti tömb végére került. Ekkor a második edény értékeit hátulról előrefelé vizsgáljuk, így a 10 érték kerül először az eredeti tömb elejére, majd a 11 a 10 előtti helyre. Végül az utolsó 11-es érték is bekerül az eredeti tömbbe. Ezt követően az 5. ábrán már csak átmásolásra kerülnek az értékek a megfelelő sorrendben, azaz az egyes edény tartalma előről olvasva, míg a második edény értékei hátulról előrefelé haladva.

2.3. A VIZSGÁLT ALGORITMUSOK



2.19. ábra. Példa a radix "vissza" rendezésre

Műveletigény

Lineáris idejű a rendezés, amennyiben d jegyűek a rendezendő számok az algoritmus futási ideje:

$$T(n) = \Theta(d \cdot n) = \Theta(n)$$

Jelölések az állapotjelző felületen

Az aktuálisan vizsgált bit értékének megfelelően, amely bináris szám a 0-ás edénybe kerül ●, míg amely az 1-es edénybe kerül az ● háttérszínt kap.

3. fejezet

Fejlesztői dokumentáció

3.1. Tervezés

A dolgozat fő célja egy olyan elsősorban hallgatóknak szánt program létrehozása, amellyel néhány rendezési algoritmus működése egy letisztult és egyszerű felhasználó felületen keresztül tanulmányozható.

A fejlesztés során több szempontot is figyelembe kell venni, úgy mint: műveletigény, memóriaigény, jó megjelenés, egyszerű kezelhetőség, és átlátható-, bővíthető kód készítése. Mivel ezen kritériumok közül több is csak egy másik rovására javítható, ezért a tervezés során kompromisszumokat kell kötni. Továbbá fel kell készülni arra, hogy az eredeti terven a fejlesztés során módosításokat kell végezni, mivel egy-egy probléma megoldása más megközelítést kívánhat.

3.1.1. Alapelvek

A programnak három jól elkülönülő komponensből kell állnia: Egy logikai(modell) részből, ami gyakorlatilag a rendszer "motorja", itt kell, hogy történjen mindenféle számítási és adattárolási művelet. Egy megjelenítési rétegből, amely a logikai rész eredményeit jeleníti meg a felhasználó számára. Végül pedig egy kontroller szintből, amely kapcsolatot teremt a logikai- és a megjelenítési réteg között. A gyakorlatban ezt a fajta tagolást nevezik Modell-Nézet-Vezérlő (*MVC*) tervezési mintának.

Az elsődleges szempont az, hogy a felhasználó könnyedén tudja kezelní a programot, és segítségével megérte az algoritmusok működését. Így a felhasználói felület áttekinthetőségrére és letisztultságára nagy hangsúlyt kell fektetni. Továbbá fontos az is, hogy a jövőben további rendezési eljárásokat is könnyedén meg lehessen jeleníteni a jelenlegiek mellett, így fontos a kód egyszerű bővíthetősége.

3.1.2. Használt fejlesztőeszközök

A fejlesztés *Eclipse SDK 4.4* fejlesztői környezet keretei között történik. A program grafikus fejlesztői felületet ad alkalmazások készítéséhez.

A kódolást segítő funkció volt a kódkiegészítés, továbbá az egyik beépített projektmenedzsment eszköz(*EGit*).

A fejlesztéshez elengedhetetlen a *Java SE 8u40* vagy magasabb verziójú szoftver. Továbbá a fejlesztést nagyban elősegíti a *JavaFX Scene Builder 2.0*, melynek segítségével egyszerűen megtervezhetővé válnak a grafikus felület komponensei.

Végül a telepítési környezet létrehozásához *Ant* és *InnoSetup* eszközök kerülnek felhasználásra. Az egész projekt, beleértve e dokumentumot is megtalálható, és az egyes verziók visszakövethetők a *GitHub*-on: <https://github.com/marfoldi/SRTNGLGRTHMS>

A program fejlesztése során egyedüli külső függvénykönyvtár a *JUnit*, melynek segítségével egységesek készülnek.

3.1.3. Felhasználói felület

Képernyőtervezek

A program főmenüjében három gomb foglal helyet, melyekre kattintva megadhatók a rendezendő számok. Továbbá az eszköztárban a **Fájl** és **Segítség** menüpontok foglalnak helyet, előbbire kattintva lehetőség nyílik a program bezárására, utóbbiban pedig megtekinthető a szoftver névjegye.



3.1. ábra. Főmenü látványterv

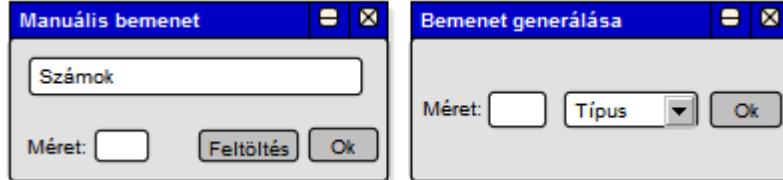
Főmenü.

Manuális bemenet. Az ablakon a **Méret:** címke mellett található mezővel definiálható a bemenet hossza. A megadását követően megjelennek a hossznak megfelelően beviteli mezők jelennek meg a **Számok** címkével jelölt részen. A **Feltöltés** gomb segítségével véletlen számokkal töltődnek fel az előbb említett beviteli mezők. Az **Ok** gombra történő kattintás után megjelennek a **Megfigyelés** és **Összehasonlítás** panelek.

3.1. TERVEZÉS

Bemenet generálása. A **Méret** címke után megadható a kívánt bemenet hossza. A **Típus** legördülő listából különböző generálási beállítások választhatóak ki. Az **Ok** gomb megnyomásával megjelennek a program főpaneljei.

Fájl beolvasása. A fájlból történő beolvasáshoz nem készült látványterv, mivel egy egyszerű fájtallázó segítségével van lehetősége a felhasználónak számokat ilyen módon megadni.



3.2. ábra. Bemenet megadása látványtervek

Főpanelek. A főpanelek eszköztárában megtalálhatóak a **Főmenü** pontjai, továbbá lehetőség van az előbb említett felületre történő visszatérésre. A **Segítség** menüpontban pedig lehetőség van az algoritmusok rövid leírásának megtekintésére. Valamint két gomb segítségével lehet navigálni a **Megfigyelés-** és **Összehasonlítás** panelek között.



3.3. ábra. Főpanelek képernyőterve

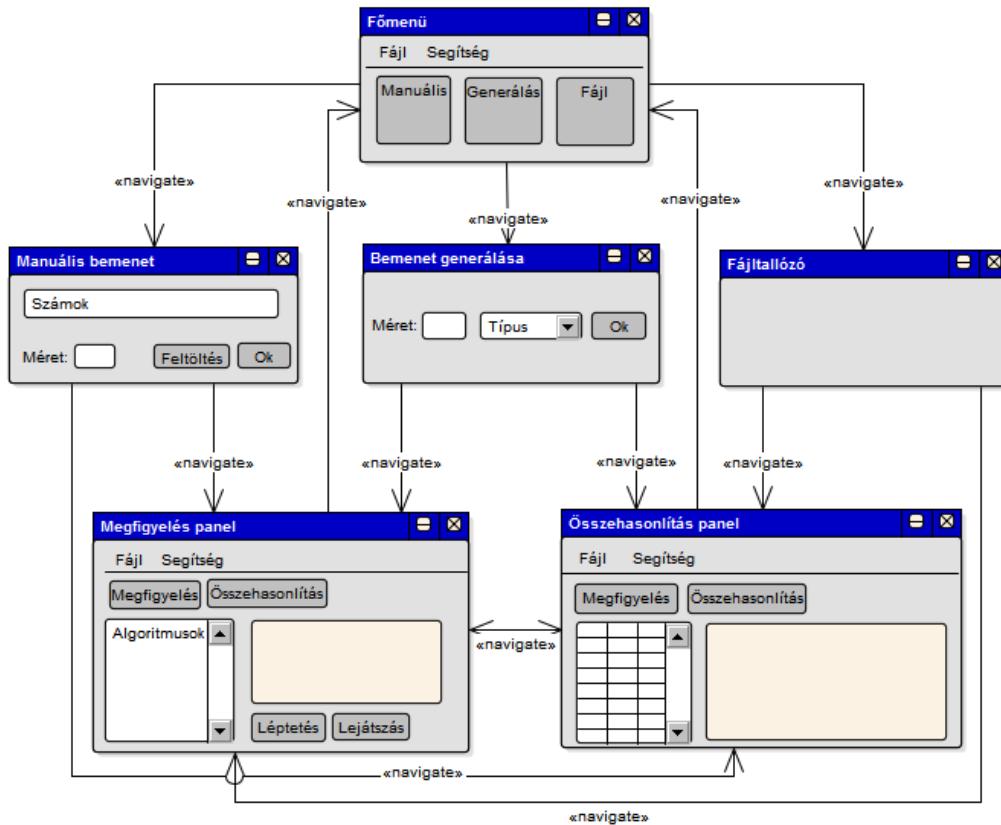
Megfigyelés panel. A bal szélen egy algoritmus lista található, melyből kiválasztható a megfigyelni kívánt eljárás. A **Léptetés** és **Lejátszás** gombokkal a sárgás színnel jelzett diagramon az algoritmus egy újabb állapota figyelhető meg.

Összehasonlítás panel. A bal oldalon található táblázatban szerepelnek az algoritmusok által végzett műveletek összegei. A sárga színnel jelölt diagramon megjelennek a kiválasztott algoritmus által végzett összehasonlítások és mozgatások összege.

Felületek közötti navigálási lehetőségek

A felületek közti navigálási irányokat a következő diagram szemlélteti:

3.1. TERVEZÉS



3.4. ábra. Navigálási lehetőségek

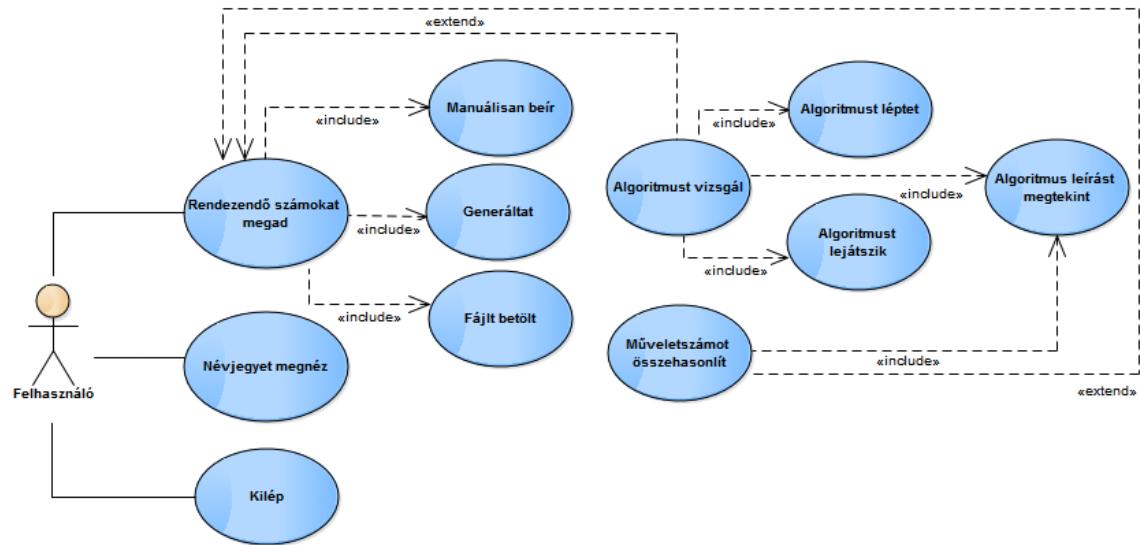
A főmenüből valamely gombra kattintva elérhetővé válik a kiválasztott típusú bemenet megadása panel. A számok megadásával megjelennek a **Megfigyelés** és **Összehasonlítás** panelek. A két panel közötti átjárás egy fülön történő kattintással tehető meg. Ezen paneleken pedig az eszköztár segítségével lehetőség van a **Főmenü**be való visszatérésre.

3.1.4. Használati esetek

A következőkben a főbb használati esetek kerülnek bemutatásra. A felhasználónak tudnia kell:

- rendezendő számok sorozatát megadni;
- algoritmust lejátszani;
- műveletszámokat összehasonlítani;
- rendezés leírását elolvasni;
- program névjegyet megtekinteni;
- programot bezárni;
- esetleges hibákról értesülni (például nem megfelelő fájl).

3.1. TERVEZÉS

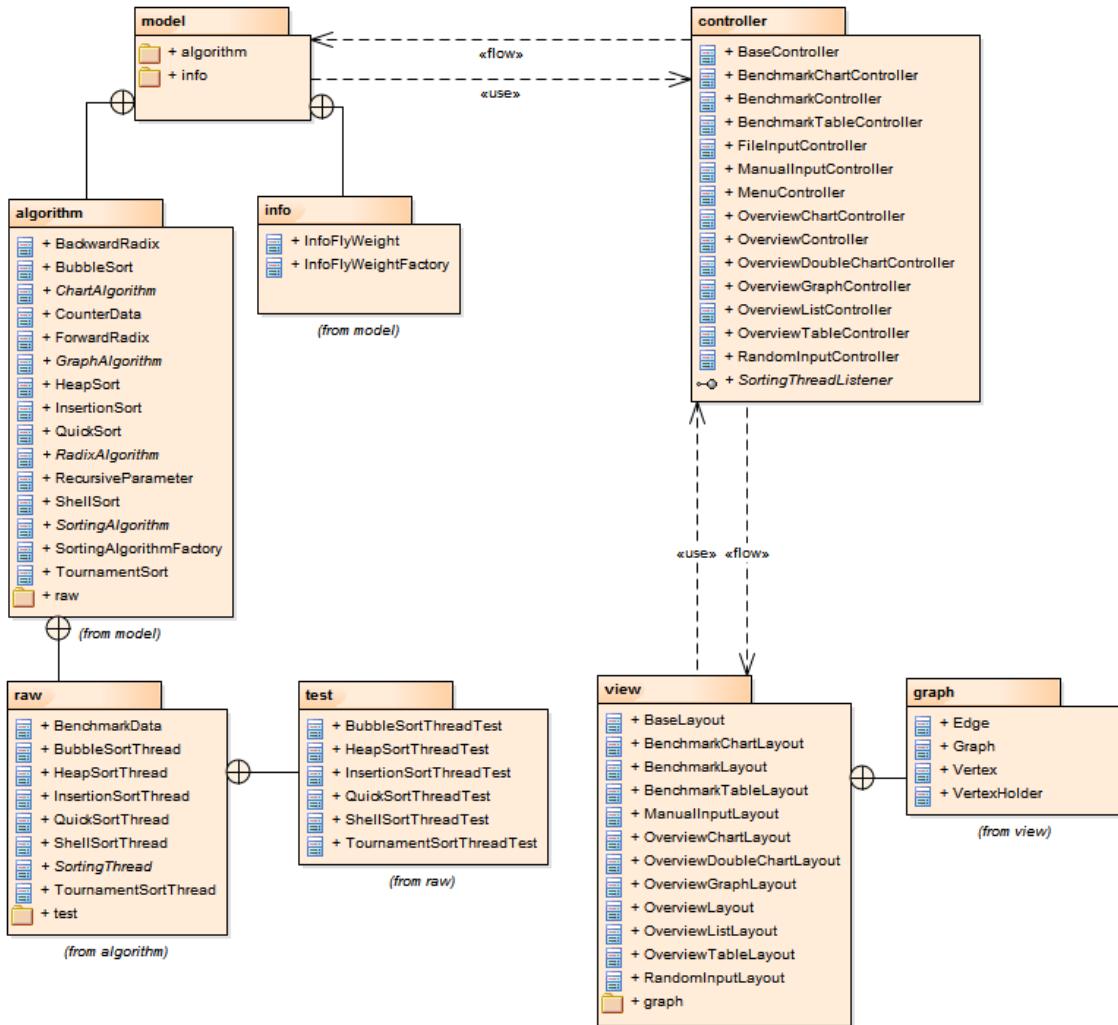


3.5. ábra. Használati esetek

3.1.5. Csomagszerkezet

Az egyik alapelve, hogy a grafikus megjelenítés működés szintjén minél inkább különüljön el a rendszerlogikától. A grafikus felület már csak a modell programésh által számított, a vezérlő programrésznek átadott adatokat jelenítse meg. Továbbá lehetőséget biztosítson a felhasználó számára, hogy vezérelje a program rendszerlogikáját a kontroller rétegen keresztül. Ez által, hogy a rendszerlogika lényegében független a grafikus interféstől. Így a szoftver magasabb fokú bővíthetőséget, a kód pedig jobb átláthatóságot nyer.

3.1. TERVEZÉS



3.6. ábra. A program csomagszerkezete

Ennek tekintetében a programot három fő csomagra kell bontani: A **modell** csomag, melyben a logikai osztályok találhatóak, a **view**, melyben a grafikai interfést leíró *FXML* fájlok és osztályok foglalnak helyet, végül a **controller** csomag, melynek osztályai kapcsolatot teremtenek a logikai és a megjelenítési réteg között.

Modell csomag

A modell osztályok a **modell** csomag foglalja magában. A pontosabb rendszerezést szem előtt tartva további alcsomagok kerülnek létrehozásra, így jobban áttekinthető az osztályhierarchia. Összességében ebben a csomagban foglalnak helyet azon osztályok, amelyek a rendszer adattároló, rendszerező és számítási feladatait ellátják. Az alábbi táblázat ad áttekintést az előzőleg említett alcsomagokról:

Név	Leírás
algorithm	A lejátszható algoritmusok implementációját tartalmazza
algorithm.raw	Az "egyszerű" rendezést megvalósító osztályok csomagja
algorithm.raw.test	A raw csomag osztályaihoz tartozó egységtesztek
info	"Az algoritmusról" felugróablakhoz tartozó adatbetöltő

algorithm alcsomag. A csomag osztályainak többsége azon algoritmusimplementációkat tartalmazza, melyek a **Megfigyelés** panelen megjelennek, azaz számon tartják az algoritmus aktuális állapotát. A csomag további osztályainak a fő funkciója, hogy azon műveleteket, amelyeket több algoritmus is használ legyenek kiemelve, ezáltal megakadályozhatóvá válik a kódismétlés.

algorithm.raw alcsomag. Azon algoritmusok megvalósítását tartalmazza, amelyekben nem kerül elmentésre a rendezés aktuális állapota. Az általuk szolgáltatott információ minden osztályt tartalmaz a csomag, amely ezen adatokat rendezett formában át tudja adni a controller rétegnek, melynek segítségével megjelennek az egyes értékek a felhasználói felületen.

algorithm.raw.test alcsomag. Az előbbi csomag osztályaihoz tartozó egységtesztek osztályait tartalmazza. Néhány egyszerűbb eset kerül tesztelésre, például: üres- és egy elemű tömb, száz véletlen generált szám rendezésének eredménye.

info alcsomag. A felületen egy algoritmust kiválasztva megtekinthető az eljárás rövid leírása. A leírások betöltése fájlból történik. A már lekérdezett szövegeket eltároljuk, így ha a felhasználó ismét lekérdezi az algoritmus leírását már nem szükséges újból beolvasni a fájlt. A csomag e funkciót megvalósító osztályokat tartalmazza.

Megjelenítő csomag

A grafikai interfész leíró *FXML* fájlokat tartalmazza. Ezen fájlokban a felület elemei kerülnek definiálásra, azonosítókkal ellátva, melyek segítségével a **Vezérlő csomag** osztályai tudnak az egyes komponensekre hivatkozni. Ezen fájlok mindegyikének neve a **Layout** szuffixet kapta, ezáltal kikövetkeztethető, hogy egy komponens elrendezésének az "tervét" tartalmazzák.

3.1. TERVEZÉS

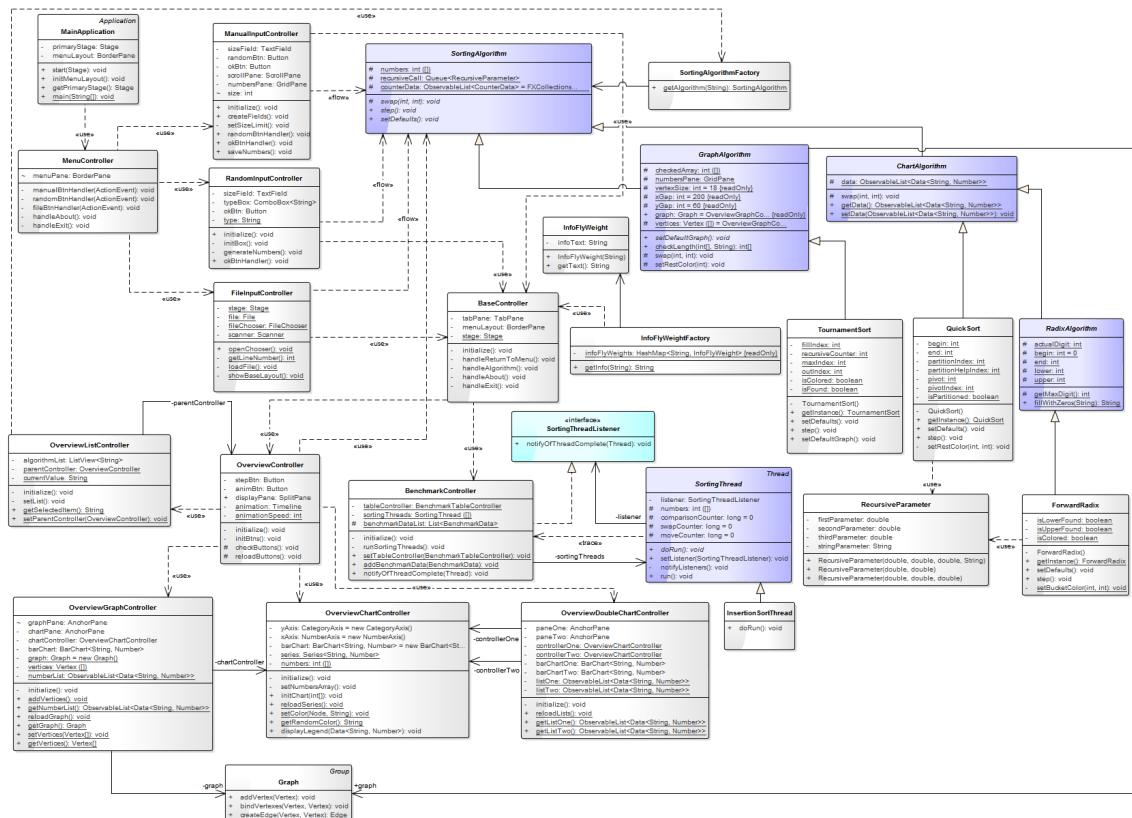
graph alcsomag. A gráf grafikus implementációját tartalmazó csomag.

Vezérlő csomag

Osztályai főleg a megjelenítő csomag komponenseihez tartoznak. Ezt az osztályok neve is jelölik, mivel csak a végződésükben(**Controller**) térnek el az *FXML* fájlok nevétől. Felhasználói interakcióra a modell osztály különböző metódusait hívja meg. Így kapcsolatot teremt az előbb definiált két csomag között. Továbbá tartalmaz egy interfész, melynek felhasználásával az **alrogithm.raw** osztály algoritmusai miután befejezték a futásukat figyelmeztetik az interfész megvalósító osztályt, ami jelen esetben a **Összehasonlítás** panel táblázatának vezérlő osztálya.

3.1.6. Osztályszerkezet

A következőkben bemutatásra kerülnek a csomagok főbb osztályai, és ezek fontosabb eljárásai. A bemutatásban szereplő osztályok sorrendje többé-kevésbé megegyezik a példányosulásuk sorrendjével. Az ábrán kék színnel vannak jelölve az absztrakt osztályok, míg zöld háttteret kapott az egyik fontosabb interfész.



3.7. ábra. A főbb osztályok kapcsolata és kommunikációjuk

A megjelenítési réteg fájljai nem kerülnek bemutatásra, mivel ezek szerkezetüket tekintve azonosak, csak a grafikus interfész elemeit definiálják. Amennyiben egy fájl

meghívásra kerül a kontroller réteg egyik osztályában az ismertetve lesz.

alapcsomag osztálya

MainApplication. A `strnglgrthms` csomag egyetlen osztálya, melyben a program `main` függvénye található. Létrehozza az ablakot, betölti a menü elrendezését tartalmazó *FXML* fájlt(**MenuLayout.fxml**).

controller csomag osztályai

A felhasználói interakciók hatását ez a réteg érvényesíti, figyeli a felület változásait és adatokat jelenít meg a megjelenítési rétegen.

MenuController. Az osztályban található metódusok kezelik a menü eszköztárán való kattintást (`handleAbout()` és `handleExit()` függvények). Továbbá a gombokon történő kattintás következtében betöltenek a bemenet megadására lehetőséget adó panelekhez tartozó *FXML* fájlokat.

ManualInputController. A menüben a **Manuális** gombra történő kattintás után példányosul az osztály. A sikeres bemenet megadását követően a `saveNumbers()` metódus átadja a számokat a **SortingAlgorithm** osztályának, ahol egy statikus tömbben kerülnek eltárolásra. Ezt követően betöltenek a program főpaneljeit tartalmazó **BaseLayout.fxml** fájlt.

RandomInputController. A **Véletlen** gombra kattintás következtében példányosul. A kontroller osztály végzi el a számok generálását, mely ezt követően az előbbivel azonosan átadásra kerül a **SortingAlgorithm** osztálynak. Szintén betöltenek a főpaneleket tartalmazó *FXML* fájlt.

FileInputController. A harmadik gombra kattintás követően keletkezik belőle objektum. Beállítja a fájlkiterjesztés szűrőt és megjeleníti a fájltallázót. Továbbá itt történik meg a fájl tartalmának az ellenőrzése. Az előző két osztályval azonosan végzi el a bemenő adatok átadását, valamint a főpanelek betöltését.

BaseController. A főpanelekhez tartozó kontroller osztály. Figyeli, hogy 100 elemnél hosszabb-e a bemenet, amennyiben igen eltünteti az **Megfigyelés** panelt. Valamint metódusai kezelik az eszköztár egy elemén való kattintást.

OverviewController. A **Megfigyelés** panel gombjait figyeli és módosítja, például ha befejeződött egy algoritmus, akkor az **Újraindítás** gombot jeleníti meg a

3.1. TERVEZÉS

felületen. Továbbá az ezeken történő kattintást kezeli. A **Léptetés** vagy a **Lejátszás** gomb megnyomásakor lekérdezi a kiválasztott algoritmust a **OverviewListController** osztálytól, majd a modell réteg **SortingAlgorithmFactory** osztályán keresztül elkéri a lejátszandó algoritmus példányát. Végül meghívja a megfelelő **algorithm** csomagban lévő osztály **step()** metódusát.

OverviewListController. A vizsgálható algoritmusok listáját tartalmazza, amennyiben a felületen módosítás történik a kiválasztott értéken újratölti a megfigyelés panelen látható diagramot, és meghívja a kiválasztott elem **setDefaults()** függvényét, amellyel az aktuális algoritmus állapotát tároló változók felveszik az alapértéküket.

OverviewChartController. A **Megfigyelés** panelen található diagram kontroller osztálya. A **SortingAlgorithm** osztály által eltárolt számokat betölti egy listába, majd ezeket megjeleníti az oszlopdiagramon. Az oszlopok színének megváltoztatását is ez az osztály végzi el. A **OverviewDoubleChartController**

OverviewDoubleChartController. Az előbb említett osztálynak két példányát tartalmazza, a **Radix "vissza"** rendezsnél tölti fel az első példány listáját a rendezendő számokkal, míg a másodikét nullákkal.

OverviewGraphController. Az előző két osztályal megegyező a funkciója. Továbbá "karbantartja" a megjelenített gráfot.

BenchmarkController. Példányosulását követően elindítja a **algorithm.raw** csomag algoritmusait. Megvalósítja a **SortingThreadListener** interfészét, ezáltal figyelmeztést tudnak küldeni neki a **SortingThread** osztályok ha befejezték futásukat. Feladata az **Összehasonlítás** felületen a táblázat frissítése, ha egy rendezési feladat megoldódott.

BenchmarkChartController. Az **OverviewChartController** osztályhoz hasonlóan egy diagram kontroller osztálya. A **Összehasonlítás** panel táblázatának egy elemén történő kattintást kezeli, amennyiben a kiválasztott elem még nem jeleinik meg a grafikonon hozzáadja, ellenkező esetben eltávolítja a grafikonról.

algorithm csomag osztályai

Főként a lejátszható algoritmusokat tartalmazza. Ezen osztályok mindegyike egyike, azaz a futás során csak egyetlen példány keletkezik belőlük.

TournamentSort. Mivel mindegyik rendezést megvalósító osztály kevéssé tér el egymástól, ezért csak egy kerül rövid ismertetésre.

Alapértelmezetten minden osztály tartalmaz egy `step()` metódust, amely egy következő állapotba lépteti a rendezést. minden osztályhoz a rendezést megvalósító változókon felül tartoznak olyan logikai változók, amelyekben az aktuális állapot elmentésre kerül. Ilyen változóban tárolódik például az, hogy a **Versenyrendezés** bináris fájában megtaláltuk-e a maximális elemhez tartozó levéleemet.

SortingAlgorithm. A rendezést megvalósító osztályok őse a **SortingAlgorithm** absztrakt osztály. Néhány olyan metódust valósít meg, amelyek minden rendezésnél azonosak.

ChartAlgorithm. Az előbb említett osztály absztrakt "gyereke", továbbá azon rendezést megvalósító osztályok ősosztálya, amelyek megjelenítéskor csak oszlopdiagramokat használnak.

RadixAlgorithm. A **ChartAlgorithm** osztály absztrakt leszármazottja. A radix rendezések ősosztálya. Néhány olyan metódust tartalmaz, amelyeket minden rendezés használ.

GraphAlgorithm. Szintén a **SortingAlgorithm** absztrakt "gyereke", és azon osztályok őse, melyek a grafikus megjelenítéskor oszlopdiagramot és gráfot is használnak.

SortingAlgorithmFactory. A **SortingAlgorithm** osztályok gyártó osztálya, az egyetlen `getAlgorithm(String algorithmName)` metódusa a paraméter tartalmától függően visszaadja a kért algoritmus példányát.

RecursiveParameter. Néhány adat eltárolására használt osztály. Például a gyorsrendezés megvalósításakor a rekurzív hívások elmentése ilyen objektumokat tartalmazó listába történik.

alogirtmh.raw csomag osztályai

Azon rendezések implementációját tartalmazza, amelyek megjelennek az **Összehasonlítás** panelen. Ezen osztályok mindenike egy külön szálat hoz létre, amely szálak befejeződésüket követően figyelmeztetik a már ismertetett **BenchmarkController** osztályt.

InsertionSortThread. Az előző csomag osztályaihoz hasonlóan, mivel ezen osztályok minimálisan térnek el felépítésüket tekintve csak egy osztály kerül rövid jellemzésre. A `sort()` metódus hívására egy programszálat indít el, amely elvégzi a rendezést majd átadja a műveletszámot a **BenchmarkController** osztálynak.

SortingThread. Az előbb említett osztálytípusok őse. Néhány olyan metódust valósít meg, amelyek minden rendezésnél azonosak. Továbbá tartalmazza a műveletek összegzéséhez használt változókat.

BenchmarkData. Egy rendezés a befejeződését követően példányosít belőle egy objektumot. Eltárolja az algoritmus által végzett összehasonlítások és mozgatások vagy cserék számát. Ilyen példány kerül átadásra a **BenchmarkController** osztály számára, amely ezekből listát készít.

info csomag osztályai

A két osztálya megvalósítja a pehelysúlyú programtervezési mintát. A minta használatának oka az, hogy a felhasználó egy adott algoritmushoz többször is lekérheti a leírást. Ekkor felesleges ismételten fájlból beolvasni a szöveget, mivel az nem változott. Célszerű ilyenkor a már lekért adatokat elmenteni, és azt visszaadni ha újra hivatkozás történik rá.

InfoFlyWeight. Az osztály egyetlen adattagja egy `String`. Ebben a változóban tárolódik egy algoritmus leírásának a szövege.

InfoFlyWeightFactory. Az osztály eltárolja a már betöltött szövegeket. A **BaseController** osztály `handleAbout()` függvénye az algoritmus listából aktuálisan kiválasztott elem szöveges reprezentációjával meghívja az osztály `getInfo(String algorithmName)` függvényét. A függvény megnézi, hogy már tárolásra került-e az algoritmushoz tartozó szöveg. Amennyiben igen, azt az objektumot adja vissza. Ellenkező esetben beolvassa a fájlt, és elmenti a szöveget egy InfoFlyWeight objektumba. Ezzel egyidejűleg az "új" algoritmus szövege is tárolásra kerül.

3.2. Megvalósítás

3.2.1. Az eredeti terv módosítása

A megvalósítás során néhány ponton módosítást kellett eszközölni az eredeti terveken.

Szálkezelés

Eredetileg egy-egy külön szálon futottak volna az algoritmusok, és a felhasználói interakció hatására ezek állapota változott volna. Azonban a *JavaFX* szálkezelése jelentősen eltér az szokványos szálkezelésétől, ezért járhatóbb útnak bizonyult az ha kétszer kerüljenek implementálásra az algoritmusok. Így került kialakításra a **algorithm.raw** csomag.

Az egyik implementációban elmentjük az interakciót követő állapotot, és ez jele-nik meg a felhasználói felületen. A másik megvalósításban pedig a rendezések azonnal lezajlanak, így képet kaphatunk arról, hogy mennyi műveletre volt szükség az egyes eljárások során. Ezen utóbbi implementációk mindegyike külön szálon fut, és ahogy valamelyik befejeződik figyelmezteti a főprogramot, hogy jelenítse meg a műveletek számát.

A JavaFX kibővítése

A szoftver erősen épít a *JavaFX* adta felületi komponensekre. Azonban akad olyan összetevő, amelyet ki kell bővíteni, vagy létre kell hozni az áttekinthetőség érdekében.

Ilyen például az oszlopdiagram, amely ugyan beépített komponens, de alapértelmezetten nem jelennek meg a diagramon az oszlopokhoz tartozó értékek számszerűen, csak a tengelyszámokból lehet kikövetkeztetni őket. Ez nyilvánvalóan így nem megfelelő, az oszlopok értékét valahogy jelölni a megjelenítéskor. A legegyszerűbb megoldás az, ha minden oszlopérték felé egy feliratot (*Label-t*) helyezünk el. Figyelni kell a szöveghez tartozó oszlop méretét. Amint megváltozik az diagram egy elemének a mérete módosítani kell a felette lévő szövegdobozban tárolt számot is.

Továbbá a *JavaFX* egyelőre nem kínál lehetőséget gráfok ábrázolására, itt megoldás lehet az, hogy egy külső függvénykönyvtárat felhasználva - például az *yFiles for JavaFX*-et - ábrázoljuk a bináris fákat. Azonban ezáltal a programnak több függősége lenne, és a könyvtárnak csupán néhány elemére van szükség a gráf megjelenítéséhez. Ezért célszerűbb egy saját implementációt készíteni, amely csupán azokat a műveleteket tartalmazza, amiket a szoftver használ.

3.2.2. A megvalósítás menete

Az első lépés volt a rendezési algoritmusok implementálása. Itt az előző alfejezetben leírtaknak megfelelően kétszer kell elkészíteni a rendezési eljárásokat. Ezzel egyidejűleg minden "egyszerű" rendezést megvalósító szálhoz egységeszteket kell írni.

Ezt követhette egyszerűbb felhasználói felület létrehozása. Kezdetben elegendő, ha csak egy grafikon jelenik meg, amely reprezentálja a tömbben található számokat.

3.2. MEGVALÓSÍTÁS

Két algoritmushoz szükséges a gráfos megjelenítés, így a következő lépés egy gráf grafikus implementálása. Ezt követően a cél, hogy néhány "beégetett" elemre a rendezések lejátszhatóak legyenek, és az aktuális állapota a tömbnek szinkronban legyen a diagrammal valamint a gráffal. Később az egyes lépésekben történő összehasonlításokat/vizsgálatokat, mozgatásokat, cseréket kell különböző színekkel jelölni az állapotjelző felületeken és számon tartani ezen műveletek összegeit.

Miután a program alapjai elkészültek kezdetét veheti a a felhasználói felület részletes kialakítása. Elsőként a diagram elhelyezése egy panelen, amely tartalmaz továbbá egy listát a választható algoritmusokról. Illetve egy táblázatot, melyben szerepelnek az aktuális állapot egyes tulajdonságai.

A programnak egy fontos szolgáltatása az, hogy a felhasználó különböző adatbeviteli mód közül választhat. A logikai réteget ki kell bővíteni ezekkel az esetekkel, továbbá a felhasználói felületen lehetőséget adni ezen módok kiválasztására.

Ezután az eszköztár kerül a helyére, mellyel párhuzamosan megtörténik az egyes műveletekhez tartozó eljárások implementálása.

Az utolsó felületi módosítás egy rendezések összehasonlítására lehetőséget adó panel létrehozása, táblázattal, benne az algoritmusok műveletigényével. Továbbá egy diagram hozzáadása, amin megjelenik a táblázatból kiválasztott sor összehasonlításainak és mozgatásainak a száma.

Végül, hogy a program egyszerűen használható legyen *Windows* környezetben egy telepítő fájl készítése, amellyel az előbb említett operációs rendszert használóknak nem szükséges külön *Java*-t telepíteniük.

3.2.3. FXML állományok

Az *FXML* az *Oracle* által fejlesztett dekleratív XML alapú nyelv[11]. Ezen fájlok definiálják a grafikus felület komponenseit. Egy ilyen fájl a következőképp épül fel:

- Az első sor minden az XML tulajdonságokat definiáló sor
- A következő sorokban található azon könyvtárak importálása, amelyekben megtalálhatók a felhasznált komponensek
- Ezt követi az ablak elrendezési módjának a megadása, ekkor több beépített megvalósítás közül lehet választani, többek között:
 - ◊ *BorderPane* - 5 pozícióba helyezhetők elemek
 - ◊ *GridPane* - táblázatszerűen jelennek meg az elemei
 - ◊ *AnchorPane* - egy "szimpla" felület, nincs megkötés

Továbbá ebben a sorban kell megadni az *FXML* állományhoz tartozó kontroller osztály elérési útvonalát. Az vezérlőosztály tud hivatkozni az egyes összetevőkre

az azonosítójukon keresztül.

- Végül a komponensek definiálása: méretük, azonosítójuk megadásával.

Néhány fontosabb FXML állomány

A következőkben bemutatásra kerülnek a projekt főbb *FXML* fájljai.

BaseController. Az egyik "összefogó" *FXML* fájl. A rendezendő számok megadása után minden esetben ez az állomány töltődik be. Alapvetően a fejlécet és egy **TabPane** komponenst tartalmaz, ami a két panel közötti navigálást teszi lehetővé. Ezeken felül importálva van két további állomány is: az **OverviewLayout** és a **BenchmarkLayout**. Ennek következtében, amikor a **BaseController** betöltődik vele együtt az előbb említett két fájl is beolvásásra kerül minden további komponensükkel.

OverviewLayout. Szintén több komponenst foglal magában, a **Megfigyelés** panelt gombjait és a más fájlok ból betöltött komponensek elhelyezkedését definíálja. Az általa importált további állományok: **OverviewListLayout**, **OverviewTableLayout**, továbbá az aktuálisan kiválasztott algoritmus típusától függően a **OverviewChartLayout**, **OverviewDoubleChartLayout** vagy **OverviewGraphLayout** valamelyike.

BenchmarkLayout. Az **Összehasonlítás** panel részeinek elhelyezkedését írja le. A következő *FXML* fájlokat hívja meg betöltésekor: **BenchmarkTableLayout**, **BenchmarkChartLayout**.

3.2.4. Nem forrásfájl állományok

A program által használt nem forrás- vagy *FXML* fájlok a **resources** mappában találhatóak. Ezen belül az **images** könyvtárban található meg a szoftver ikonja. A **text** mappán belül találhatóak azon fájlok, amelyeket az **InfoFlyWeight** osztály betölt. Azaz ezekben az állományokban található meg az egyes algoritmusok rövid ismertetése.

3.2.5. Osztályok leírása

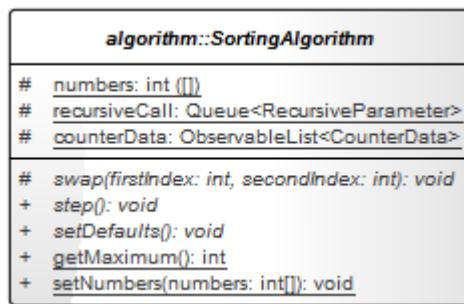
Az előző fejezetben(3.1) ismertetésre kerültek a főbb osztályok és azok fontosabb eljárásai, a következőkben nagyobb részletességgel kerülnek jellemzésre a szoftver osztályai.

Modell réteg osztályai

Elsőként a **algorithm** csomag főbb osztályai kerülnek ismertetésre.

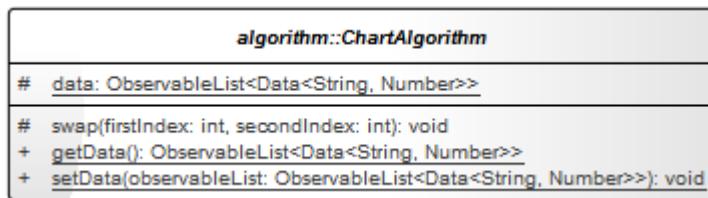
SortingAlgorithm absztrakt osztály. A **Megfigyelés** panelen megjelenő algoritmusok osztályainak őse. Az osztály **numbers** statikus adattagjában kerül el-tárolásra a bemenetkor megadott számsorozat. További két adattagja egy rekurzív hívások mentésére szolgáló- és egy **CounterData** objektumokat tartalmazó lista.

A **getMaximum()** eljárás visszaadja a **numbers** tömb maximális elemét. A függvénynek a megjelenítéshez használt oszlopdiagram lépésközének definiálásánál van szerepe. Továbbá tartalmaz három absztrakt eljárást, amelyeket a leszárma-zott osztályai valósítanak meg.



3.8. ábra. A SortingAlgorithm osztálydiagramja

ChartAlgorithm absztrakt osztály. Az előző osztály leszárma-zottja, megvalósítja a **swap(int firstIndex, int secondIndex)** metódusát. A diagram kontroller osztálya a rendezendő számokat egy **ObservableList** adatszerkezetben tárolja, így ennek megfelelően került az eljárás implementálásra. A kontroller réteg előbb említtet listáját statikus változóként elmenti, így a leszárma-zott osztályai egyszerűbben tudnak hivatkozni az egyes értékekre.

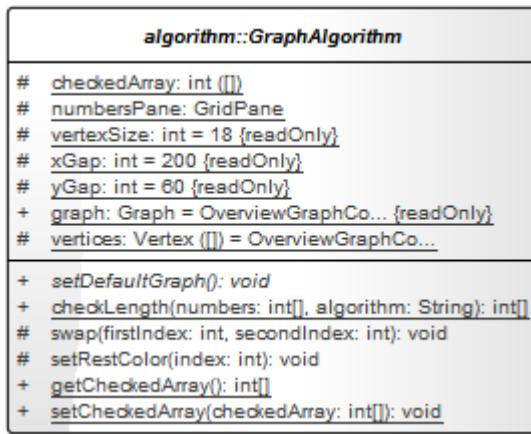


3.9. ábra. A ChartAlgorithm osztálydiagramja

GraphAlgorithm absztrakt osztály. Szintén a **SortingAlgorithm** leszárma-zott absztrakt osztálya. A **GraphController** osztály gráfját és csúcsait tárolja, így a leszárma-zott osztályai egyszerűbb elérik ezeket az adattagokat.

3.2. MEGVALÓSÍTÁS

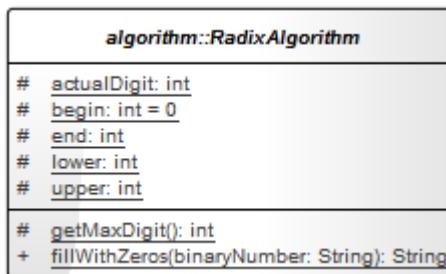
Továbbá ellenőrzi a bemenet hosszát, és ha szükséges "csonkolja" vagy megnöveli az eredeti tömb hosszát a **checkLength(int[] numbers, String algorithm)** metódusa. Megvalósítja az előző osztályhoz hasonlóan a **swap(int firstIndex, int secondIndex)** metódust. Valamint a **setRestColor(int index)** metódusa a paraméterül kapott indexig visszaállítja az eredeti színűre a **Megfigyelés** panelen megjelenített gráfcsúcsokat, valamint a oszlopokat.



3.10. ábra. A GraphAlgorithm osztálydiagramja

RadixAlgorithm absztrakt osztály. A **ChartAlgorithm** leszármazott absztrakt osztálya. Továbbá a radix eljárásokat megvalósító osztályok őse. A két származtatottjának adattagjai már itt definiálásra kerülnek, azaz négy "pointer", valamint az aktuálisan vizsgált számjegy indexe(**actualDigit**).

A **getMaxDigit()** metódusa visszaadja a leghosszabb bináris számot, amelyet a másik **fillWithZeros(String binaryNumber)** metódus fel is használ, mivel ennek megfelelően a rövidebb számokat nullákkal konkatenálja.



3.11. ábra. A GraphAlgorithm osztálydiagramja

SortingAlgorithmFactory osztály. Létrehozásakor a gyártó metódus programtervezési minta lett alapul véve. Ezen az osztályon keresztül kérhetők el az algoritmusok példányai, függetlenül attól, hogy az aktuális algoritmus őse a **ChartAlgorithm** vagy a **GraphAlgorithm** osztály.

3.2. MEGVALÓSÍTÁS

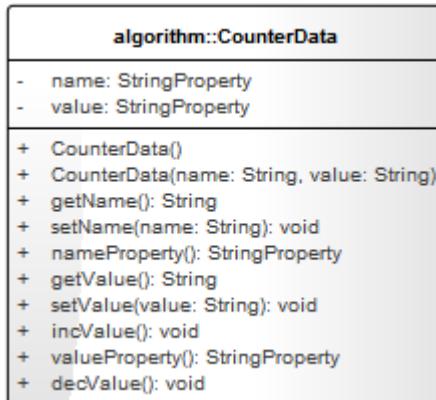
Egyetlen metódusának paramétere egy algoritmus szöveges reprezentációja, például: "Gyorsrendezés". Az eljárás visszaadja a rendezést megvalósító osztály példányát. Elsősorban a kontroller réteg **AlgorithmList** osztálya használja.



3.12. ábra. A SortingAlgorithmFactory osztálydiagramja

CounterData osztály. Az osztály segítségével adatok tárolhatóak kulcs-érték párból. Az implementációval kapcsolatban felmerülhet a kérdés, hogy miért nem a Java nyelv egy beépített adatszerkezete került felhasználásra. Ennek oka, hogy a felhasználói felületen lévő táblázatok csak olyan adatszerkezetet tudnak megjeleníteni, amelynek adattagjainak a típusa **Property** végződésű, és implementálva vannak ezekhez a mezőkhöz a **getter** metódusok.

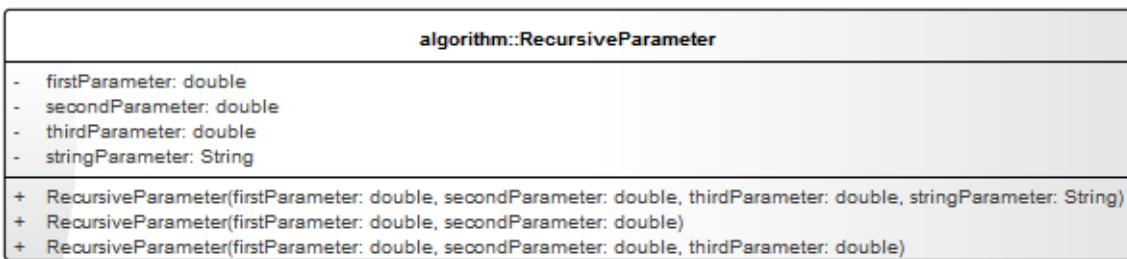
Az algoritmusokat megvalósító osztályok ilyen adatokból álló listában tárolják az egyes "tulajdonságaiat", és adják át a vezérlőrétegnek. A vezérlőréteg pedig a **Megfigyelés** panel táblázatában jeleníti meg az értékeket.



3.13. ábra. A CounterData osztálydiagramja

RecursiveParameter osztály. Funkciója hasonló az előbb ismertetett osztályéhez, azaz adatok tárolására lett létrehozva. Legfeljebb négy érték elmentésére alkalmas. Azon algoritmusok, amelyek rekurziót használnak ezen objektumokból álló listába mentik el a rekurzív hívások paramétereit. A konstruktora túlterhelt, azaz legalább kettő, maximum négy paraméterrel hívható.

3.2. MEGVALÓSÍTÁS



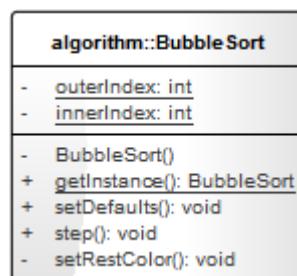
3.14. ábra. A RecursiveParameter osztálydiagramja

A következőkben az algoritmusok lejátszását megvalósító konkrét osztályokat jellemzzük, melyek szintén az **algorithm** csomagban találhatóak. Az osztályok mindegyike egyke, azaz a program futása során egyetlen példány keletkezik belőlük.

BubbleSort osztály. A buborékrendezést megvalósító egyke osztály. Ősosztálya a **ChartAlgorithm** Két változója megfelel az algoritmus két ciklusváltozójának. Konstruktora privát, a `getInstance()` metódusán keresztül kérhető le a példánya. Amennyiben nincs még létrehozva példány elkészíti, ellenkező esetben visszaadja a már létrehozott objektumot.

Az egyke tulajdonságából következően azokat a változókat melyekben tárolódik a rendezés aktuális állapota minden futtatás után vissza kell állítani az eredeti értékre, ezt a `setDefaults()` függvény teszi meg. Továbbá ez a metódus hozza létre a tulajdonságokat tartalmazó **CounterData** objektumokat.

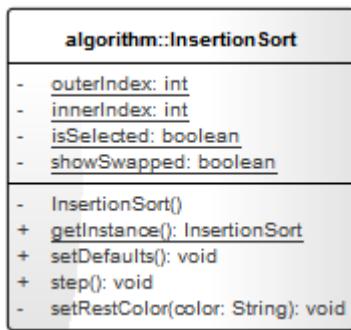
A `step()` örökölt eljárás egy rendezési lépést hajt végre. A `setDefaults()` metódus pedig visszaállítja a **Megfigyelés** panelen látható diagramok színét az alapértékre.



3.15. ábra. A BubbleSort osztálydiagramja

InsertionSort osztály. A beszúró rendezést megvalósító osztály. Alapvetően felépítését és függvényeit tekintve megegyezik a **BubbleSort** osztálynál leírtakkal. Különbség, hogy két további logikai változóval rendelkezik. Az `isSelected` mezőben tárolódik az az információ, hogy már kiválasztásra került-e egy elem, mivel a `step()` metódus ennek megfelelően hajtja végre a következő lépést. Továbbá a `showSwapped` változó értéke adja meg, hogy a felületen megjelenítésre került-e egy mozgatás.

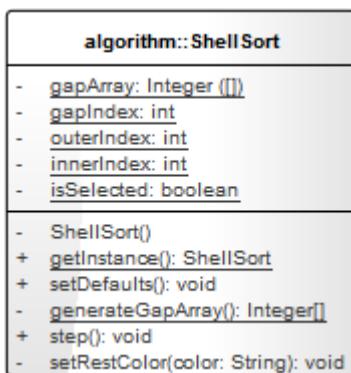
3.2. MEGVALÓSÍTÁS



3.16. ábra. Az InsertionSort osztálydiagramja

ShellSort osztály. A Shell rendezést megvalósító osztály. Őse, akárcsak az eddigi osztályoknak a **ChartAlgorithm**. A **BubbleSort**-nál leírtakkal lehet jellemezni, viszont tartalmaz néhány további változót és eljárást. A **gapArray** tömbben tárolódnak a lépésközök, az aktuális lépésköz a **gapIndex** változó segítségével kérhető le a tömbből. Az **InsertionSort**-nál leírtakkal megegyező szerepet tölt be a **isSelected** változó.

A **generateGapArray()** eljárás feltölti a lépésközöket tároló tömböt a bemenet méretétől függően a **Vaughan Pratt** által javasolt értékekkel[6]. Az implementációban a legutóbbi tömbhöz adott számot kettővel és hárommal szorozzuk, és számon tartjuk, hogy egy lépésben melyik szorzást végezük el. Amikor elérünk azt az értéket, amely nagyobb vagy egyenlő a tömb hosszával, akkor befejezzük a számok generálását, az utolsó érték nem lesz eleme a tömbnek.



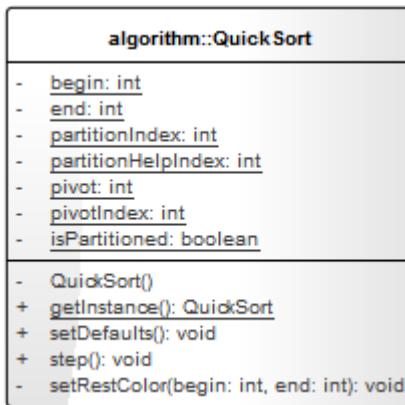
3.17. ábra. A ShellSort osztálydiagramja

QuickSort osztály. A gyorsrendezés implementációja. Őse szintén a **ChartAlgorithm** osztály. A **begin** és **end** változói jelölik az aktuálisan rendezendő tömbrész első és utolsó elemét. A **partitionIndex** és **partitionHelpIndex** mezőknek a tömb felosztásánál van szerepük. A **pivot** változóban tárolódik a vezérelem értéke, melynek indexét a **pivotIndex** tartalmazza. A **isPartitioned** logikai változó értéke mutatja, hogy megtörtént-e a tömb felosztása. Az osztály használja a **SortingAlgo-**

3.2. MEGVALÓSÍTÁS

rithm űsének a **recursiveCall** listáját, ide menti el a rekurzív hívásokhoz szükséges adatokat.

A **setDefaults()** metódus működése megegyezik az eddigiekkel. A **setRestColor()** eljárás viszont két egész számot vár paraméterként, és a két szám által meghatározott intervallumon kívül eső részen az oszlopdiagramok színét fakóra állítja át a vezérlő rétegen keresztül.



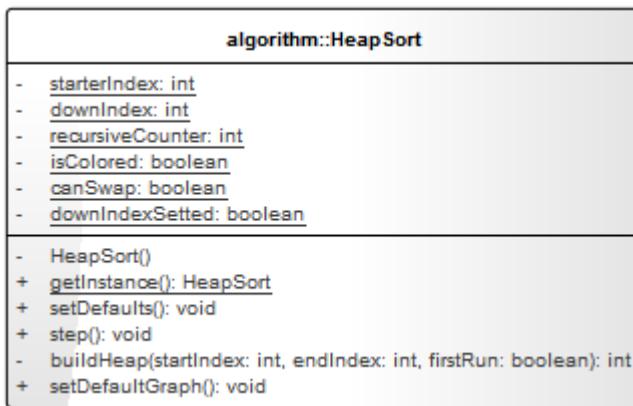
3.18. ábra. A QuickSort osztálydiagramja

HeapSort osztály. Őse a **GraphAlgorithm** osztály, a kupacrendezést valósítja meg.

A **starterIndex** változó tárolja azon elemét indexét, melynek vizsgáljuk a testvér-szülő "viszonyát". A **downIndex** segédváltozóban tárolódik a kupac tulajdonság ellenőrzésekor azon elem indexe, amelyet következőleg vizsgálni kell. A **recursiveCounter** megfeleltethető a kupacrendezés eljárás ciklusváltozójának. Továbbá három logikai mezővel rendelkezik az osztály, melyek közül a **isColored** jelöli, hogy már a diagramon megváltozott-e a vizsgált elemek színe. A **canSwap** változó értékétől függően hajtja végre az algoritmus a cseréket, mivel ez csak azt követően eszközölhető, ha már kijelöldtek a cserélendő elemek a **Megfigyelés** panelen. A **downIndexSetted** segédváltozó amennyiben igaz, akkor már megtörtént a kupac felépítése az első menetben, innentől kezdve a gyökérelemtől kell vizsgálni a kupac-tulajdonságot, így ennek megfelelően átállítódik a **downIndex** értéke 0-ra.

Az átláthatóságot szem előtt tartva a **step()** metódusból kiemelésre került a kupac(újra)építés eljárása, melyet a **buildHeap(int startIndex, int endIndex, boolean firstRun)** metódus valósít meg. A harmadik paramétere határozza meg, hogy alapkupac építése-e a feladat vagy a kupactulajdonság ellenőrzése. Az eljárás visszatérési értéke a következő vizsgálandó elem. A **setDefaultGraph()** függvény építi fel a bináris fát a rendezendő számokból, majd adja át a vezérlőrétegnek megjelenítésre.

3.2. MEGVALÓSÍTÁS



3.19. ábra. A HeapSort osztálydiagramja

TournamentSort osztály. A versenyrendezést megvalósító egyke osztály, melynek ōse az előző osztályhoz hasonlóan a **GraphAlgorithm**.

A **fillIndex** a versenyfa felépítésekor tárolja azon elem indexét, amelynek gyerekeit "versenyeztetjük". Akárcsak az előbb jellemzett osztálynál, itt is megfeleltethető a **recursiveCounter** változó az algoritmus ciklusváltozójának. A **maxIndex** mező tárolja a maximális levélelem keresésekor az egyik megtalált belső csúcsot. Az **outIndex** segítségével pedig elérjük egy maximális elem megtalálásakor a **Megfigyelés** panel oszlopdiagramjának egy adott oszlopát.

A **setDefaultGraph()** és **step()** eljárás funkciója megegyezik a **HeapSort**-nál leírtakkal.



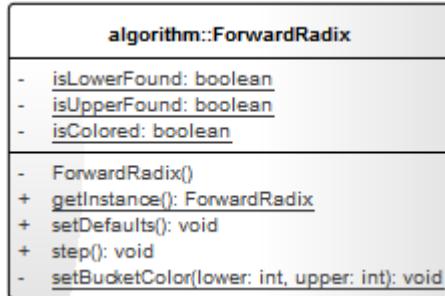
3.20. ábra. A TournamentSort osztálydiagramja

ForwardRadix osztály. Őse a **RadixAlgorithm** osztály, a radix "előre" rendezést valósítja meg. Az **isLowerFound** és **isUpperFound** változók tárolják, hogy egy adott lépéssben talált-e az algoritmus olyan értéket, amely nem a megfelelő helyen van. Továbbá az **isColored** logikai mező értéke megadja, hogy egy cserélendő elem színe megváltozott-e már a felületen.

A **setBucketColor(int lower, int upper)** eljárás a két paraméterül kapott

3.2. MEGVALÓSÍTÁS

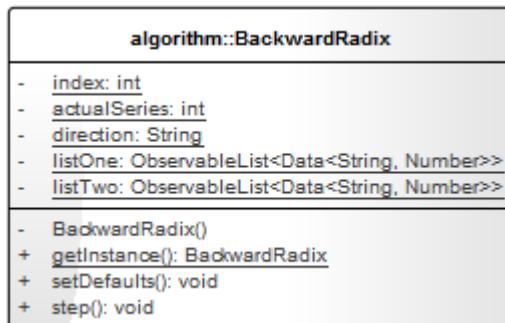
intervallumon beállítatja a kontroller rétegben az oszlopok színét egységesre, így kijelöl egy edényt. A **step()** metódus használja az örökölt **recursiveCall** listát.



3.21. ábra. A ForwardRadix osztálydiagramja

BackwardRadix osztály. A radix "vissza" rendezést valósítja meg, őosztálya a **RadixAlgorithm**. Az **index** változó tárolja az aktuálisan vizsgált elem indexét. Az **actualSeries** azt a tömböt, amelyben vizsgálni kell az elemeket. A **direction** változó megadja, hogy az aktuális résztömbön mely irányból kell végezni a vizsgálatot. A **listOne** és **listTwo** változók megfeleltethetőek az algoritmus által használt két tömbnek.

A **step()** metódus használja az örökölt **recursiveCall** listát.

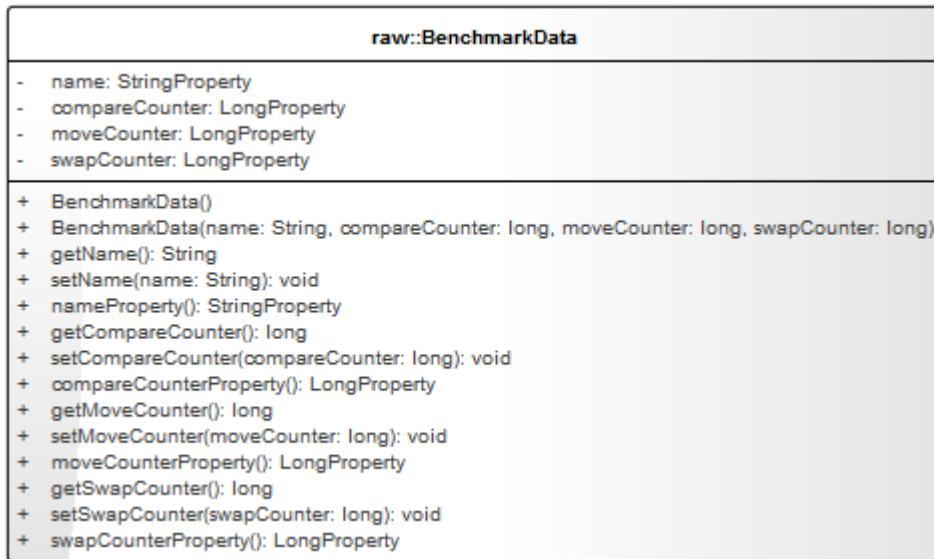


3.22. ábra. A BackwardRadix osztálydiagramja

Most az **algorithm.raw** csomag főbb osztályai kerülnek jellemzésre.

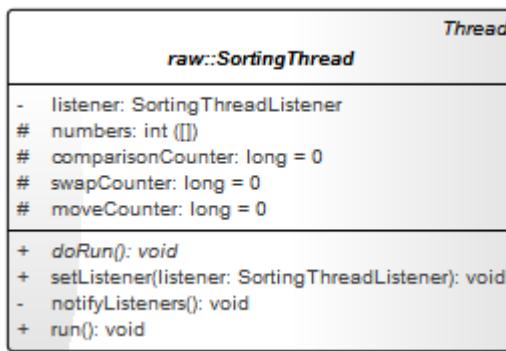
BenchmarkData osztály. A **CounterData** osztályhoz hasonló a funkciója és felépítése. Az algoritmus nevét és a rendezés során végzett műveletek összegét tárolja. A kontroller réteg használja fel a **Összehasonlítás** panel táblázatának a kitöltséhez. minden rendezést végző programszál létrehoz belőle egy objektumot, majd hozzáadja egy listához, amit a kontroller réteg lekér egy szál befejeződésekor.

3.2. MEGVALÓSÍTÁS



3.23. ábra. A BenchmarkData osztálydiagramja

SortingThread absztrakt osztály. A csomagban található rendezést megvalósító osztályok szülője. Őse a `Thread` osztály. A `SortingThreadListener` mezője tárolja azt az objektumot(`BenchmarkController`), amelyet a szál befejeződése után figyelmeztet. Továbbá adattagjai tárolják a rendezés során végzett műveletek számát. A leszármazott osztályai a `doRun()` metódusát valósítják meg, amely a programszál elindításakor hívódik meg.



3.24. ábra. A SortingThread osztálydiagramja

A következőkben az **Összehasonlítás** panelen megjelenő információkhöz adatokat szolgáltató konkrét osztályok kerülnek jellemzésre.

BubbleSortThread osztály. Megvalósítja az előbb említett `doRun()` metódust. Elvégzi a rendezést, majd a feladat befejeztével a kontroller réteg `BenchmarkController` osztályában található `BenchmarkDataList` listához hozzáfűz egy új `BenchmarkData` objektumot.

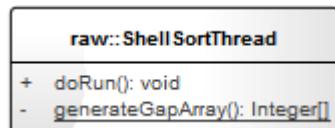
3.2. MEGVALÓSÍTÁS



3.25. ábra. A BubbleSortThread osztálydiagramja

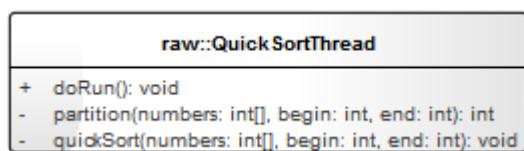
InsertionSortThread osztály. Őse a **SortingThread** osztály. Felépítése teljesen azonos a **BubbleSortThread** jellemzésénél leírtakkal. Ennél fogva az osztálydiagramban sincs eltérés.

ShellSortThread osztály. Az egyetlen különbség az előző osztályokhoz képest, hogy rendelkezik egy további metódussal. A `generateGapArray()` metódus meggyezik a **ShellSort** osztály jellemzésében szereplő azonos nevű függvényteljesítményét.



3.26. ábra. A ShellSortThread osztálydiagramja

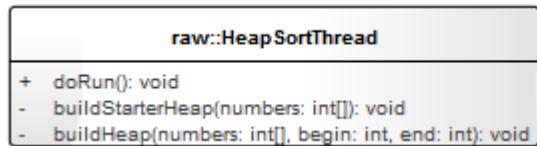
QuickSortThread osztály. Két további metódussal rendelkezik, ezektől eltekintve megegyezik az **BubbleSortThread** felépítésével. A `doRun()` metódusból kiemelésre került a tömb felosztását végző függvény(`partition(int[] numbers, int begin, int end)`), valamint a rekurzív hívásokat végző `quickSort(int[] numbers, int begin, int end)` eljárás.



3.27. ábra. A QuickSortThread osztálydiagramja

HeapSortThread osztály. A **QuickSortThread**-hez hasonlóan két metódus került kiemelésre a `doRun()` metódusból. Az egyik a `buildStarterHeap(int[] numbers)` eljárás, amely felépíti a kezdőkupacot. Továbbá a `buildHeap(int[] numbers, int begin, int end)` függvény, amely ellenőrzi a kupactulajdonságot.

3.2. MEGVALÓSÍTÁS

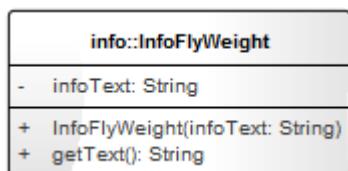


3.28. ábra. A HeapSortThread osztálydiagramja

TournamentSortThread osztály. Felépítése teljes mértékben megegyezik az **BubbleSortThead**-nél leírtakkal, ennél fogva az osztálydiagramja is.

Az **info** alcsomag osztályai kerülnek bemutatásra.

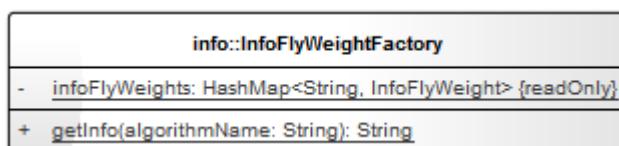
InfoFlyWeight osztály. Egyetlen adattagja egy szöveges mező. Ebben a változóban tárolódik egy algoritmushoz tartozó leírás.



3.29. ábra. Az InfoFlyWeight osztálydiagramja

InfoFlyWeightFactory osztály. Az előző osztály közreműködésével megvalósítja a pehelysúlyú programtervezési mintát. Mivel a felhasználó többször is lekérheti egy algoritmus leírását, ezért célszerű egy objektumban tárolni ezt az adatot, így nem szükséges minden lekérésnél újra betölteni a leírást tartalmazó fájlt. Egy **HashMap** adatszerkezetben kulcs-érték párokként tárolódnak a már betöltött algoritmusleírások.

A `String getInfo(String algorithmName)` metódusa ellenőrzi, hogy a paraméterként kapott algoritmus név (például: Gyorsrendezés) kulcs szerepel-e a **HashMap**-ben. Amennyiben igen, visszaadja a kulcshoz tartozó **InfoFlyWeight** objektumot. Ellenkező esetben betölti az algoritmus leírását, példányosít egy új **InfoFlyWeight**-et, majd hozzáfűzi a **HashMap**-hez.



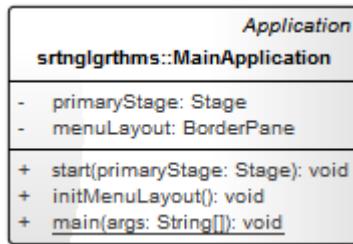
3.30. ábra. Az InfoFlyWeightFactory osztálydiagramja

Kontroller réteg osztályai

MainApplication osztály. Ugyan az alapcsomag része, funkcionalitását tekintve mégis a kontroller osztályok közé sorolható. Az osztály tartalmazza a program **main()** metódusát. A szoftver indításakor ez a metódus hívódik meg.

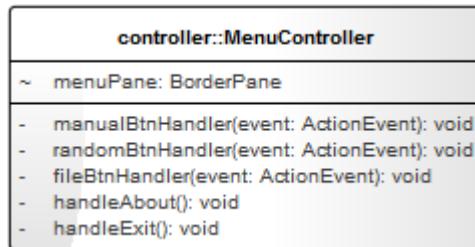
A **start()** függvény létrehoz egy "színpadot" (**Stage**), amely az ablaknak felel meg. Beállítja az ablak címsorát, betölti az alkalmazás ikonját. Majd meghívja a **InitMenuLayout** metódust.

Az imént említett eljárás betölti a főmenü elrendezését tartalmazó *FXML* fájlt, azaz a **MenuLayout.fxml**-t.



3.31. ábra. A MainApplication osztálydiagramja

MenuController osztály. A **MenuLayout.fxml** fájl betöltésével egyidejűleg példányosul. A felület eszköztárán a **Segítség** -> **Névjegy** pontjára kattintva fut le a **handleAbout()** metódus. Megjelenít egy **alertBoxot**, benne a program névje- gyének szövegével. Továbbá a **Fájl** -> **Bezárás** eszköztár elemet kiválasztva meghívódik a szoftver leállását eredményező **handleExit()** eljárás. A további függvényei a **MenuLayout.fxml** fájlban definiált gombokon történő kattintást kezelik, és betöltik a kiválasztott bemenettípushoz tartozó *FXML* fájlt.



3.32. ábra. A MenuController osztálydiagramja

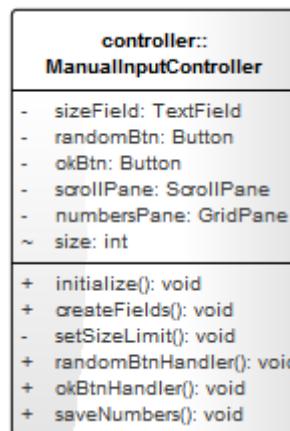
ManualInputController osztály. A **ManualInputLayout.fxml** fájl betölté- sével példányosul, ekkor lefut az **initialize()** metódus, mely letiltja a kattintást a felület két gombján.

A **setSizeLimit()** metódus figyeli a méret mező tartalmát, amennyiben nem pozitív egész szám a gombok továbbra is tiltva maradnak. Továbbá, ha a megadott

3.2. MEGVALÓSÍTÁS

hossz nagyobb száznál, akkor automatikusan átírja az értéket 100-ra. Amennyiben a méret megadása sikeres meghívódik a **createFields()** eljárás, mely létrehozza a számok beírására lehetőséged adó mezőket.

A **randomBtnHandler()** metódus a **Feltöltés** gombon történő kattintást követően fut le. Ekkor feltölti a létrehozott mezőket 0 és 100 közötti véletlen számokkal. A **saveNumbers()** metódus átadja a rendezendő számokat a **SortingAlgorithm** osztályának. Ezt követően betölti a program főpaneljeit tartalmazó **BaseLayout.fxml** fájlt.



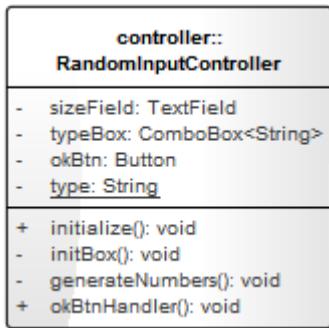
3.33. ábra. A ManualInputController osztálydiagramja

RandomInputController osztály. A **RandomInputLayout.fxml** fájl betöltésekor jön létre belőle egy példány. Ekkor az **initialize()** metódus eljárás letiltja az **Ok** gombon való kattintást, továbbá meghívja az **initBox()** függvényt.

Az imént említett eljárás feltölti a generálás módjára lehetőséget adó legördülő listát az opciókkal. Az **okBtnHandler()** kezeli az **Ok**-on történő kattintást. Itt történik meg a bemenet méretét tartalmazó mező ellenőrzése.

Amennyiben megfelelő a hossz formátuma, azaz egy pozitív egész szám meghívódik a **generateNumbers()** eljárás. A **typeBox** változóban tárolt legördülőlista kiválasztott elemétől függően kerülnek a számok generálásra. A kigenerált tömböt átadja **SortingAlgorithm** osztályának, majd betölti a program főpaneljeit tartalmazó **BaseLayout.fxml** fájlt.

3.2. MEGVALÓSÍTÁS

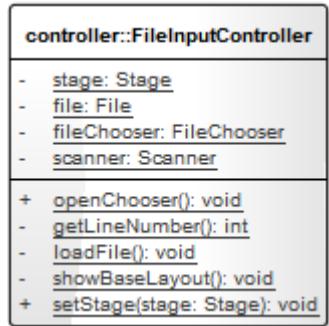


3.34. ábra. A RandomInputController osztálydiagramja

FileInputController osztály. A **FileInputLayout.fxml** betöltésekor példányosul. Az **openChooser()** eljárás beállítja a formátumszűrőt (*.txt), valamint megjeleníti a fájltallázó párbeszédpantelt.

A fájl kiválasztását követően meghívja a **loadFile()** függvényt. Kezdetben lekéri a fájl sorainak a számát a **getLineNumber()** eljárás segítségével, majd létrehoz egy ennek megfelelő nagyságú tömböt. Soronként végigolvassa a fájlt, amennyiben egy nem megfelelő sort talál (azaz egy sor üres vagy nem egy pozitív egész számot tartalmaz) befejezi a beolvasást, és hibaüzenetet jelenít meg.

Amennyiben a beolvasás sikeres volt átadja a tömböt a **SortingAlgorithm** osztálynak, és megnyitja a főpaneleket tartalmazó *FXML* fájlt.



3.35. ábra. A FileInputController osztálydiagramja

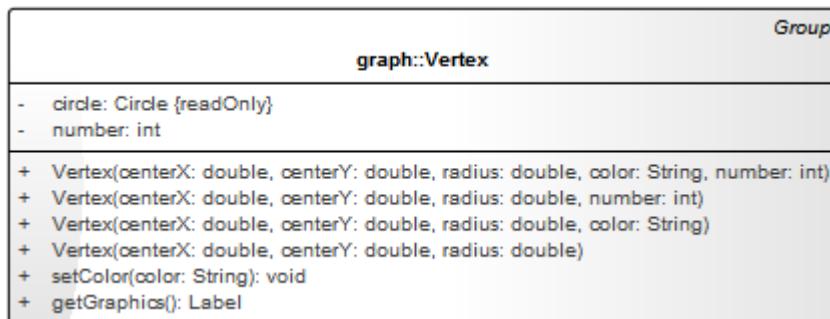
Megjelenítési réteg osztályai

A megjelenítésért felelős komponensek főként *FXML* fájlokban kerülnek tárolásra. A gráf implementációja azonban ilyen módon nem készíthető el. Ezért került kialakításra a **view.graph** csomag. Röviden most e csomag osztályai kerülnek jellemzésre.

Vertex osztály. A gráf egy csúcsát megvalósító osztály. Adattagja két *JavaFX* komponens, egy címke(**Label**) és egy kör(**Circle**). Konstruktora megkapja a kör **X** és **Y** pozícióját az ablakon, továbbá opcionálisan a címke vagy szín értékét.

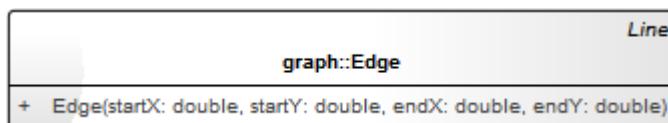
3.2. MEGVALÓSÍTÁS

A **getGraphics()** metódus visszaadja a csúcs grafikus reprezentációját. A **setColor(String color)** metódus segítségével megváltoztatható a kör adattag színe.



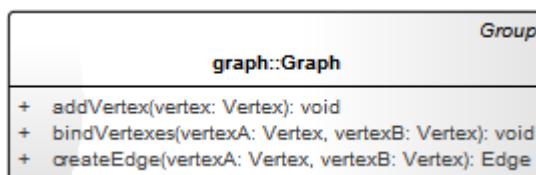
3.36. ábra. A Vertex osztálydiagramja

Edge osztály. Konstruktora a paraméterül kapott koordináták felhasználásával létrehoz egy vonal(**Line**) objektumot.



3.37. ábra. Az Edge osztálydiagramja

Graph osztály. Őse a **Group** osztály, ezáltal hozzáadható a felhasználó felülethez. Az **addVertex(Vertex vertex)** metódusával új csúcs adható hozzá a gráfhoz. A **bindVertices(Vertex vertexA, Vertex vertexB)** eljárás a két paraméterül kapott csúcsot köti össze a **createEdge(Vertex vertexA, Vertex vertexB)** metódus segítségével.



3.38. ábra. A Graph osztálydiagramja

3.2.6. Tesztelés

4. fejezet

Irodalomjegyzék

- [1] Dr. Fekete István: *Algoritmusok és adatszerkezetek I. jegyzet*, [ONLINE] [Hivatkozva: 2015.04.20] http://people.inf.elte.hu/fekete/algoritmusok_bsc/alg_1_jegyzet/
- [2] Thomas H. Cormen - Charles E. Leiserson - Ronald L. Rivest - Clifford Stein: *Új algoritmusok*, Scolar kiadó, 2003, [992], 9789639193901
- [3] Rónyai Lajos - Ivanyos Gábor - Szabó Réka: *Algoritmusok*, Typotex Elektronikus Kiadó Kft., 2000, [350], 9789632790145
- [4] Demuth, H.: *Electronic Data Sorting*, PhD thesis, Stanford University, 1956, [184]
- [5] *Shellsort*, Wikipedia the free encyclopedia. [ONLINE] [Hivatkozva: 2015.04.25] <http://en.wikipedia.org/wiki/Shellsort/>
- [6] Vaughan Ronald Pratt: *Shellsort and Sorting Networks (Outstanding Dissertations in the Computer Sciences)* Dissertations-G, 1980, [59], 0824044061
- [7] C.A.R. Hoare: *Algorithm 64: Quicksort* Communications of the ACM, 4, 7, 1961
- [8] *Java (programming language)*, Wikipedia the free encyclopedia. [ONLINE] [Hivatkozva: 2015.04.21] [http://en.wikipedia.org/wiki/Java_\(programming_language\)/](http://en.wikipedia.org/wiki/Java_(programming_language))
- [9] *JavaFX*, Wikipedia the free encyclopedia. [ONLINE] [Hivatkozva: 2015.04.21] <http://en.wikipedia.org/wiki/JavaFX/>
- [10] *JUnit*, Wikipedia the free encyclopedia. [ONLINE] [Hivatkozva: 2015.05.01] <http://en.wikipedia.org/wiki/JUnit>

-
- [11] *FXML*, Wikipedia the free encyclopedia. [ONLINE] [Hivatkozva: 2015.05.10] <http://en.wikipedia.org/wiki/FXML/>
 - [12] The Takipi Blog: *We Analyzed 30,000 GitHub Projects – Here Are The Top 100 Libraries in Java, JS and Ruby*, [ONLINE] [Hivatkozva: 2015.04.30] <http://blog.takipi.com/we-analyzed-30000-github-projects-here-are-the-top-100-libraries-in-java-js-and-ruby/>